

---

# Linear Orderings of Sparse Graphs

---

## Dissertation

KATHRIN HANAUER

*Supervisor:*

PROF. DR. FRANZ J. BRANDENBURG

Fakultät für Informatik und Mathematik  
Universität Passau

July 2017



## Abstract

The LINEAR ORDERING problem consists in finding a total ordering of the vertices of a directed graph such that the number of backward arcs, i. e. , arcs whose heads precede their tails in the ordering, is minimized. A minimum set of backward arcs corresponds to an optimal solution to the equivalent FEEDBACK ARC SET problem and forms a minimum CYCLE COVER.

LINEAR ORDERING and FEEDBACK ARC SET are classic  $\mathcal{NP}$ -hard optimization problems and have a wide range of applications. Whereas both problems have been studied intensively on dense graphs and tournaments, not much is known about their structure and properties on sparser graphs. There are also only few approximative algorithms that give performance guarantees especially for graphs with bounded vertex degree.

This thesis fills this gap in multiple respects: We establish necessary conditions for a linear ordering (and thereby also for a feedback arc set) to be optimal, which provide new and fine-grained insights into the combinatorial structure of the problem. From these, we derive a framework for polynomial-time algorithms that construct linear orderings which adhere to one or more of these conditions. The analysis of the linear orderings produced by these algorithms is especially tailored to graphs with bounded vertex degrees of three and four and improves on previously known upper bounds. Furthermore, the set of necessary conditions is used to implement exact and fast algorithms for the LINEAR ORDERING problem on sparse graphs. In an experimental evaluation, we finally show that the property-enforcing algorithms produce linear orderings that are very close to the optimum and that the exact representative delivers solutions in a timely manner also in practice.

As an additional benefit, our results can be applied to the ACYCLIC SUBGRAPH problem, which is the complementary problem to FEEDBACK ARC SET, and provide insights into the dual problem of FEEDBACK ARC SET, the ARC-DISJOINT CYCLES problem.



## Preface

You don't do a PhD thesis all on your own. Certainly, you are the one who writes it and who agonizes about its content, but there are so many people without whom this project would not have been realizable. I wish to express my gratitude to everyone who helped me come this far.

Let me start with my supervisor, Prof. Dr. Franz J. Brandenburg. Thank you for "infecting" me with FEEDBACK ARC SET and LINEAR ORDERING and I mean this in a strictly positive sense! You piqued my curiosity in cycles and their breakup in the first place and, as a matter of fact, it lasted until now. Thank you also for introducing me to the world of papers, conferences, and journals, for relying on my abilities, and for your always valuable advice. I also would like to thank Prof. Dr. Ulrik Brandes for being the second referee for this dissertation, for his interest in my work, and for some very inspiring conversations.

During my time working as a teaching and research assistant at the Chair of Theoretical Computer Science, I was lucky to have great colleagues, some of whom have also become friends over time. Thank you, Christopher Auer, Christian Bachmaier, Wolfgang Brunner, Susanne Frühauf, Andreas Hofmeier, Marco Matzeder, Daniel Neuwirth, Josef Reislhuber, and Alexander Riemer, for sharing your experience and your advice, for many expert discussions, and simply for having made this chair a great place to work. Special thanks go to my office mates, Christopher Auer and Josef Reislhuber, whom I naturally spent most of the time with. I am also incredibly grateful to Christian Bachmaier for carefully proof-reading this thesis and for his very helpful comments.

On the private side, I want to thank my partner, Olaf. If it were not for you, this project would probably never have been finished. You were my rock in turbulent waters, you cheered me up when I had found myself once more in a dead end with a proof or with something else, you were always there when I needed you. Thank you for believing in me. Aside from that, I am highly grateful for your professional opinion on this thesis. I also wish to especially thank my parents, Christiana and Reiner: Thank you for your understanding, your enduring support, for helping me through difficult episodes,

for accommodating me, and for covering my back such that I can focus on this thesis. Cordial thanks also to my sister, Christina, for being a great and appreciated dialogue partner, for taking work off me, and for some very cheerful recreational activities! I am also truly grateful to my partner's parents, Erika and Heinz, for their encouragement and for providing me several times a place to work, but also to rest.

Passau, in July 2017

Kathrin Hanauer

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>The Linear Ordering Problem: An Outline</b>	<b>5</b>
2.1	The Linear Ordering Problem and its Kin . . . . .	6
2.1.1	Problem Statements . . . . .	6
2.1.2	Linear Programming . . . . .	9
2.1.3	Dual Problems . . . . .	11
2.1.4	The Acyclic Subgraph Polytope . . . . .	13
2.1.5	Reductions to and from the LINEAR ORDERING Problem . . . . .	14
2.2	Complexity . . . . .	18
2.2.1	$\mathcal{NP}$ -Completeness Results . . . . .	18
2.2.2	Approximability and Approximations . . . . .	21
2.2.3	Parameterized Complexity . . . . .	23
2.2.4	Polynomially Solvable Instances . . . . .	25
2.3	The Cardinality of Optimal Feedback Arc Sets . . . . .	26
2.3.1	Existential Bounds . . . . .	26
2.3.2	Algorithms with Absolute Performance Guarantees . . . . .	27
2.4	Linear Orderings in Practice . . . . .	28
2.4.1	Heuristics . . . . .	28
2.4.2	Applications . . . . .	30
<b>3</b>	<b>Preliminaries: Definitions and Preparations</b>	<b>35</b>
3.1	Basics . . . . .	35
3.1.1	Sets and Multisets . . . . .	35
3.1.2	Minimal and Maximal versus Minimum and Maximum . . . . .	37
3.1.3	Computational Complexity . . . . .	37
3.1.4	Data Structures . . . . .	37
3.2	General Graph Theory . . . . .	38
3.2.1	Graphs, Vertices, and Arcs . . . . .	38

3.2.2	Vertex Degrees . . . . .	39
3.2.3	Paths, Cycles, and Walks . . . . .	40
3.2.4	Connectivity and Acyclicity . . . . .	41
3.3	Feedback Arc Sets and Linear Orderings . . . . .	43
3.3.1	Feedback Arc Sets . . . . .	43
3.3.2	Linear Orderings . . . . .	44
3.3.3	Forward Paths and Layouts . . . . .	47
3.4	Preprocessing and Default Assumptions . . . . .	48
3.4.1	Loops and Anti-Parallel Arcs . . . . .	48
3.4.2	Strong Connectivity . . . . .	49
3.4.3	Biconnectivity . . . . .	50
<b>4</b>	<b>Properties of Optimal Linear Orderings: A Microscopic View</b>	<b>55</b>
4.1	General Framework . . . . .	55
4.2	Algorithmic Setup . . . . .	58
4.2.1	Graphs . . . . .	58
4.2.2	Linear Orderings . . . . .	59
4.2.3	Vertex Layouts . . . . .	59
4.2.4	Initializing the Data Structures . . . . .	60
4.2.5	General Remarks . . . . .	63
4.3	Nesting Property . . . . .	63
4.3.1	A 1-opt Algorithm . . . . .	63
4.3.2	Nesting Arcs . . . . .	68
4.3.3	A Graph's Excess . . . . .	71
4.4	Path Property . . . . .	73
4.4.1	Forward Paths for Backward Arcs . . . . .	73
4.4.2	Establishing Forward Paths . . . . .	74
4.4.3	Minimal Feedback Arc Sets . . . . .	76
4.5	Blocking Vertices Property . . . . .	77
4.5.1	Left- and Right-Blocking Vertices . . . . .	78
4.5.2	Vertical Splits . . . . .	82
4.5.3	Establishing Non-Blocking Forward Paths . . . . .	84
4.6	Multipath Property . . . . .	87
4.6.1	Arc-Disjoint Forward Paths . . . . .	87
4.6.2	Analyzing the Flow Network Approach . . . . .	91



4.6.3	Arc-Disjoint Cycles . . . . .	94
4.6.4	An $\mathcal{NP}$ -hard Extension . . . . .	95
4.7	Multipath Blocking Vertices Property . . . . .	97
4.7.1	Non-Blocking Multipaths . . . . .	98
4.7.2	Flow Networks for Split Graphs . . . . .	103
4.7.3	Again an $\mathcal{NP}$ -hard Extension . . . . .	107
4.8	Eliminable Layouts Property . . . . .	112
4.8.1	Eliminable Layouts . . . . .	112
4.8.2	The Elimination Operation . . . . .	117
4.8.3	Eliminating Eliminable Layouts . . . . .	122
4.9	A PsiOpt-Algorithm . . . . .	126
4.9.1	A Cascading Meta-Algorithm . . . . .	127
4.9.2	Establishing the Necessary Properties Simultaneously . . . . .	129
4.10	Manipulations and Meta-Properties . . . . .	132
4.10.1	Basic Operations on Linear Orderings and Graphs . . . . .	133
4.10.2	Fusion Property . . . . .	143
4.10.3	Reduction Property . . . . .	146
4.10.4	Arc Stability Property . . . . .	149
<b>5</b>	<b>Maximum Cardinality of Optimal Feedback Arc Sets of Sparse Graphs</b>	<b>151</b>
5.1	Auxiliary Graphs . . . . .	152
5.1.1	The Forward Path Graph . . . . .	152
5.1.2	The Pooled Forward Path Graph . . . . .	154
5.1.3	The Polarized Forward Path Graph . . . . .	156
5.1.4	The Truncated Forward Path Graph . . . . .	157
5.2	Subcubic Graphs . . . . .	159
5.2.1	A Tight Bound . . . . .	159
5.2.2	On the Approximation Ratio . . . . .	163
5.3	From Subcubic to General Graphs . . . . .	167
5.3.1	Pebble Transportation in Supercubic Graphs . . . . .	168
5.3.2	A General Assignment Scheme . . . . .	169
5.4	Subquartic and Subquintic Graphs . . . . .	173
5.4.1	Two Special Cases of One-Arc Stability . . . . .	173
5.4.2	A Tight Bound for Subquartic Graphs . . . . .	177
5.4.3	Subquintic Graphs . . . . .	193

<b>6</b>	<b>Exact and Fast Algorithms for Linear Ordering</b>	<b>197</b>
6.1	Partial Layouts and Incomplete Linear Orderings . . . . .	198
6.2	Exact Algorithms for Optimization and Decision . . . . .	201
6.3	Branch and Bound with Integrated Partial Layouts . . . . .	208
6.4	Fine-Tuning . . . . .	218
6.5	Runtime Comparison for Sparse Graphs . . . . .	219
<b>7</b>	<b>Experimental Evaluation</b>	<b>221</b>
7.1	The Algorithm Test Suite . . . . .	222
7.1.1	Algorithms . . . . .	222
7.1.2	Input Instances . . . . .	224
7.1.3	Technical Setup . . . . .	226
7.1.4	Evaluation . . . . .	227
7.2	Sparse Regular Graphs . . . . .	228
7.2.1	Selection and Configuration of Algorithms . . . . .	228
7.2.2	Performances and Running Times . . . . .	229
7.2.3	Summary . . . . .	237
7.3	Large Graphs . . . . .	239
7.3.1	Fences, Ladders, and their Composites . . . . .	239
7.3.2	Performances and Running Times . . . . .	240
7.3.3	Summary . . . . .	243
7.4	The LOLIB Graph Library . . . . .	244
7.4.1	Sets of LOLIB Instances . . . . .	244
7.4.2	Performances and Running Times . . . . .	246
7.4.3	Comparison to Other Approaches . . . . .	249
7.5	Threats to Validity . . . . .	249
7.5.1	Construct Validity . . . . .	249
7.5.2	Internal Validity . . . . .	250
7.5.3	External Validity . . . . .	250
7.6	Summary . . . . .	250
<b>8</b>	<b>Conclusion and Future Work</b>	<b>253</b>
	<b>Bibliography</b>	<b>255</b>
	<b>Notation Index</b>	<b>267</b>

**Algorithm Index**

**273**

**Subject Index**

**277**



To put it in extreme terms, one might say that cycles appear in our everyday life. There even is the term “life cycle”, which appears in various contexts: Products have one, software, organisms. We are also confronted on a daily basis with other types of cycles. Soon after getting up, we make use of the water cycle. When we need to go somewhere, we find that the time-tables of public transportation have cycles: If we wait only long enough, there always is a next bus, train, subway to come. Some of them even operate themselves on a circle route. Large cities often have ring roads and in rural areas, drivers are frequently confronted with roundabouts, which provide the advantage that, while going in circles, one has all the time in the world to decide which exit to take. At least in theory. A cycle basically describes a permanent reoccurrence of events or a process that, instead of terminating, always starts anew. All of the aforementioned sounds very positive—as if cycles were in general a nice thing. Sometimes, however, cycles are not good. We can be trapped in a vicious cycle and need to break it. We may want to have a clear beginning and a clear ending, or something which is first and something which is last. It may be important to know with what to start.

When modeling relationships between entities as a directed graph, cyclic structures occur rather frequently. All the same, they are undesired in many applications and have to be cut, usually however under the precondition that their breakup changes the graph as little as possible. Such situations naturally include different types of ranking problems, for example in conjunction with elections, votings, or sports tournaments. In the late 18th century, the Marquis de Condorcet observed what is known today as the Condorcet paradox [dC85]: Given three or more options, the majority preferences can be cyclic, even though the individual preferences are not. Cycles also hamper the analysis of systems, such as in electronic circuits testing or software verification. Not least, cyclic dependencies are often the cause for bootstrapping problems in various fields of computing.

The problem of breaking cycles by removing a minimum cardinality set of arcs from a directed graph  $G$  is known as the FEEDBACK ARC SET problem. As the thereby

obtained acyclic subgraph always admits a topological sorting, the problem can be formulated equivalently as the LINEAR ORDERING problem, which consists in finding a total ordering  $\pi$  of the vertices of  $G$  that minimizes the number of “backward” arcs  $(u, v)$ , where  $\pi(u) > \pi(v)$ . In an optimal solution, there is a one-to-one correspondence between the backward arcs with respect to a linear ordering and the removed arcs in an optimal feedback arc set. This perspective on FEEDBACK ARC SET also underlines its connection to ranking problems. Another benefit of the LINEAR ORDERING problem is that it can be visualized nicely. To this end, the vertices are placed on a horizontal line in the order of  $\pi$ . “Forward” arcs, which are those that agree with the linear ordering, are then drawn from left to right above the line and backward arcs from right to left below.

FEEDBACK ARC SET is a relatively old problem with respect to computational complexity. Accordingly, there are many results concerning it and which equally apply to its sibling, the LINEAR ORDERING problem. We therefore start with a comprehensive survey on the latter, which includes FEEDBACK ARC SET as well as further related problems. The chapter stretches from the discussion of computational complexity over worst-case cardinalities of backward arc sets to practical considerations such as heuristics and applications.

Afterwards, we define some mathematical and graph-theoretical notation and terms that enable us to deal with the LINEAR ORDERING problem formally. At the end of this chapter, we also stipulate some basic assumptions regarding the input graphs under consideration.

Next, we study optimal linear orderings thoroughly and with an attentive eye for details. Whereas this may at first seem like a waste of time and effort in light of the fact that LINEAR ORDERING belongs to the class of  $\mathcal{NP}$ -hard problems, we later demonstrate the opposite for sparse instances in various respects. In the course of this chapter, we derive a number of properties that are characteristic for optimal linear orderings and show for each of them how it can be established algorithmically and, with few minor exceptions, in a time-efficient manner. In result, we develop a polynomial-time algorithm that constructs a linear ordering compliant to all major properties. As an addendum, the last section here also briefly touches some high-level properties, the so-called meta-properties.

Subsequent to this, we devote ourselves to an in-depth analysis of linear orderings that adhere to properties of the previous chapter and consider the cardinality of their induced set of backward arcs. For graphs having  $n$  vertices and a maximum vertex degree of three, i. e., subcubic graphs, we derive an improved upper bound of  $\lfloor \frac{n}{3} \rfloor$  on

---

the cardinality of an optimal feedback arc set and show that it is tight. By extending our approach, we are also able to assess vertices of degree four. This yields an improved upper bound of  $\lfloor \frac{2n}{3} \rfloor$  for the cardinality of an optimal feedback arc set of a subquartic graph having  $n$  vertices. Again, the new result is best possible. Furthermore, we conjecture that a very similar approach can be applied to show that an optimal feedback arc set of a subquintic graph has cardinality at most  $\lfloor \frac{2.5n}{3} \rfloor$ .

In the following chapter, we make use of some of the properties to develop new exact algorithms which are fast on sparse graphs and at the same time space-efficient. For cubic graphs, they have a running time of only  $\mathcal{O}^*(\sqrt{2}^n)$  and thereby outperform even other candidates that require exponential space.

The penultimate chapter contains an experimental evaluation of algorithms that establish a set of properties efficiently as well as of one of the new exact algorithms. For comparison, they also compete against standard algorithms that have been used in the past to tackle the FEEDBACK ARC SET problem and the LINEAR ORDERING problem. Here, the property-enforcing routines show their superiority on sparse graphs and still very good performance on other graphs.

We conclude in the last chapter and pick up on some open problems and suggestions for future work.





## 2

# The Linear Ordering Problem: An Outline

First references to the LINEAR ORDERING problem date back to the 1950s, when Chenery and Watanabe [CW58] studied methods to analyze the interdependencies among different productive sectors in economic sciences. What is more, they already suggested a simple and straightforward algorithmic approach to obtain a good or even an optimal solution in this context. Since then, the LINEAR ORDERING problem has found comprehensive consideration in different branches of mathematics and computer science and has been encountered in many and varied applications.

This chapter briefly surveys known facts about LINEAR ORDERING. It starts with different equivalent statements of the problem that can be found in literature as well as closely related problems and their mutual reductions. This also includes the formulation of the LINEAR ORDERING problem as linear program and the introduction of the linear ordering polytope. Next, we address its computational complexity, how it has been tackled algorithmically, and compile results regarding the cardinality of optimal solutions. The chapter concludes with an overview of heuristics and practical applications. Additionally, we summarize in [Table 2.1](#) a selection of the algorithms that have been mentioned.

In order to provide a concise description of studied graphs and graph classes, let us briefly agree on terminology. Unless indicated otherwise, the term graph, denoted by  $G$ , always refers to a directed, unweighted multigraph with vertex set  $V$  and arc set  $A$ , which is a multiset of ordered pairs of vertices. In particular, this includes the possibility for  $G$  to contain parallel arcs as well as loops. The cardinalities of  $V$  and  $A$  are denoted by  $n$  and  $m$ , respectively. An arc  $(u, v)$  is said to leave vertex  $u$  and enter vertex  $v$ . The number of arcs leaving a vertex is called its *outdegree*, and the number of arcs entering it accordingly its *indegree*. The *degree* of a vertex is the sum of its out- and indegree. A graph  $G$  is called *Eulerian* if for every vertex  $v$  of  $G$ , the number of arcs entering  $v$  equals the number of arcs leaving  $v$ . The *complete* graph on  $n$  vertices is the graph that has an

arc between every ordered(!) pair of vertices, i. e., for  $u, v \in V, u \neq v, (u, v) \in A$  and  $(v, u) \in A$ . In contrast, a *tournament* is a graph that has exactly one arc between every unordered pair of vertices, i. e., for  $u, v \in V, u \neq v$ , either  $(u, v) \in A$  or  $(v, u) \in A$ , but neither both nor none. A graph is called *bipartite*, if its set of vertices can be partitioned into sets  $V' \dot{\cup} V''$  such that there is no arc connecting two vertices contained in the same partition. A *bipartite tournament* is a bipartite graph with an arc between every unordered pair of vertices  $v', v''$  such that  $v' \in V'$  and  $v'' \in V''$ . The definition of bipartite can be generalized from two partitions to an arbitrary number  $c$ , thus resulting in the notion of  $c$ -partite graphs and tournaments.

Throughout this chapter, we use big O notation (see [Section 3.1.3](#)) to specify the computational complexity of problems and algorithms.

## 2.1 The Linear Ordering Problem and its Kin

Different perspectives on the LINEAR ORDERING problem, resulting from its appearance in multifarious kinds of application as well as its being researched in the field of pure (graph theory) and applied mathematics (optimization) as well as computer science have led to a number of formulations stating more or less the same problem. In the following, we summarize the most frequently used definitions of the LINEAR ORDERING problem and list some relevant reductions to and from other problems.

### 2.1.1 Problem Statements

A *linear ordering*  $\pi$  of  $G$  is a sequential ordering of its vertex set, which is modeled as a bijection that assigns to each vertex  $v$  a unique integer from the range  $\{0, \dots, |V| - 1\}$ . Given a graph with vertex set  $\{a, b, c, d, e\}$ , e. g., a possible linear ordering could be as follows:

$$\pi(a) = 0, \quad \pi(b) = 4, \quad \pi(c) = 3, \quad \pi(d) = 1, \quad \pi(e) = 2.$$

The order of the vertices defined by  $\pi$  here is  $(a, d, e, c, b)$ . Given a specific linear ordering  $\pi$ , an arc  $(u, v)$  of  $G$  is a forward arc if its orientation strictly conforms with the linear ordering, i. e., if  $\pi(u) < \pi(v)$ . Otherwise, if  $\pi(u) \geq \pi(v)$ ,  $(u, v)$  is a backward arc. We denote the multiset consisting of all forward arcs by  $\mathcal{F}_\pi$ , and the multiset consisting of all backward arcs by  $\mathcal{B}_\pi$ . Continuing the above example, an arc directed from  $a$  to  $c$ , e. g., would agree with  $\pi$  and thus be part of  $\mathcal{F}_\pi$ . However, an arc directed from  $c$  to  $a$  would contradict  $\pi$  and therefore be contained in  $\mathcal{B}_\pi$ . Note that also an arc directed from  $a$  to

$a$ , i. e. , a loop, is considered contradictory. For a visual representation, the vertices are usually placed on a horizontal line in the order of  $\pi$ . Forward arcs are then drawn in the upper half-plane and backward arcs in the lower. Examples are shown in [Figure 3.6](#).

The “classic” LINEAR ORDERING problem now consists in finding a linear ordering  $\pi$  of a given graph that minimizes the number of arcs contradicting it, i. e. , to minimize the cardinality of  $\mathcal{B}_\pi$ :

---



---

### The Linear Ordering Problem (LO)

**Instance:** directed graph  $G$

**Question:** What linear ordering  $\pi$  of  $G$  minimizes  $|\mathcal{B}_\pi|$ ?

---

At times, the arcs of the directed graph under consideration have attributes assigned, such as a weight function  $w$ . In this case, it may be desirable not to minimize the number of contradictory arcs, but instead the sum of their weights. This leads to the weighted version of the LINEAR ORDERING problem:

---



---

### The Weighted Linear Ordering Problem (wLO)

**Instance:** directed graph  $G$ , arc weight function  $w$

**Question:** What linear ordering  $\pi$  of  $G$  minimizes  $\sum_{a \in \mathcal{B}_\pi} w(a)$ ?

---

The WEIGHTED LINEAR ORDERING problem generalizes the LINEAR ORDERING problem by using a constant arc weight function such as  $w : A \rightarrow \{1\}$ . Conversely, however, the WEIGHTED LINEAR ORDERING problem with integer or rational arc weights may also be transformed back to the LINEAR ORDERING problem by translating the weight of each arc to an adequate number of parallel arc copies. Arcs with negative weights can be handled by reversing their direction and proceeding with the absolute value of the weight.

Both the LINEAR ORDERING problem and the WEIGHTED LINEAR ORDERING problem have been stated as optimization problems. The respective decision versions are obtained by additionally supplying an upper bound  $k$ . Thus, the counterpart of the LINEAR ORDERING problem is:

---



---

**The Linear Ordering Decision Problem (dLO)**

**Instance:** directed graph  $G$ , positive integer  $k$

**Question:** Is there a linear ordering  $\pi$  of  $G$  such that  $|\mathcal{B}_\pi| \leq k$ ?

---

If we again regard arc weights, we obtain the decision version of the WEIGHTED LINEAR ORDERING problem:

---



---

**The Weighted Linear Ordering Decision Problem (wdLO)**

**Instance:** directed graph  $G$ , arc weight function  $w$ , positive integer  $k$

**Question:** Is there a linear ordering  $\pi$  of  $G$  such that  $\sum_{a \in \mathcal{B}_\pi} w(a) \leq k$ ?

---

For an arbitrary linear ordering  $\pi$  of a graph  $G$ , the subgraph of  $G$  that consists only of the forward arcs can never have a cycle. This implies that every cycle in  $G$  must contain at least one arc of  $\mathcal{B}_\pi$ . For this reason, the multiset  $\mathcal{B}_\pi$  is also called a feedback arc set of  $G$ . Indeed, this view on the LINEAR ORDERING problem is very common in the field of graph theory:

---



---

**The Feedback Arc Set Problem (FAS)**

**Instance:** directed graph  $G$  with arc set  $A$

**Question:** What is the smallest subset  $\mathcal{B} \subseteq A$  that contains at least one arc of every cycle in  $G$ ?

---

The according decision problem here reads:

---



---

**The Feedback Arc Set Decision Problem (dFAS)**

**Instance:** directed graph  $G$  with arc set  $A$ , positive integer  $k$

**Question:** Is there a subset  $\mathcal{B} \subseteq A$ ,  $|\mathcal{B}| \leq k$ , that contains at least one arc of every cycle in  $G$ ?

---

The definitions of the weighted versions of FAS and dFAS, the WEIGHTED FEEDBACK ARC SET problem (wFAS) and the WEIGHTED FEEDBACK ARC SET DECISION problem (wdFAS), can be obtained straightforwardly. The formal equivalence of LO and FAS has already been established by Younger [You63] in 1963 and implies the equivalence of the respective weighted and/or decision versions.

The “FEEDBACK ARC SET perspective” also leads to three further names for the LINEAR ORDERING problem, namely FEEDBACK CUTSET, CYCLE COVER, and CYCLE HITTING SET. Yet another term is based on the observation that if the problem were considered on an undirected graph instead, the subgraph consisting only of forward arcs would be a tree, which motivated the name CHORD SET [You63].

The adjacency matrix of a directed graph  $G$  on  $n$  vertices is an  $n \times n$  matrix  $H$  with entries  $h_{ij}$ ,  $0 \leq i, j < n$  such that  $h_{ij} = 1$  if there is an arc from the  $i$ th to the  $j$ th vertex and  $h_{ij} = 0$  otherwise. In case that  $G$  is weighted,  $h_{ij}$  instead equals the respective arc’s weight, if the arc exists. Using this graph representation, the LINEAR ORDERING problem is equivalent to the following statement:

---



---

### **The Matrix Triangulation Problem**

**Instance:**  $n \times n$  matrix  $H$

**Question:** What simultaneous permutation of  $H$ ’s rows and columns minimizes the sum of the entries (on and) below the diagonal?

---

Note that the entries on an adjacency matrix’s diagonal correspond to loops of the graph and remain on the diagonal for every simultaneous permutation of the rows and columns.

## 2.1.2 Linear Programming

By switching back to the FEEDBACK ARC SET perspective, the LINEAR ORDERING problem can straightforwardly be formulated as an integer 0-1 linear program. Let  $\mathcal{C}$  be the set of all cycles of a graph  $G$  with  $n$  vertices and arc set  $A$ . For a cycle  $C \in \mathcal{C}$ , denote by  $a \in C$  that the arc  $a$  is contained in cycle  $C$ . For each arc  $a$ , we use a variable  $x_a$  to specify whether  $a$  is part of the feedback arc set ( $x_a = 1$ ) or not ( $x_a = 0$ ). Then, the corresponding linear program reads:

$$\text{minimize } \sum_{a \in A} x_a \quad (2.1a)$$

subject to

$$\sum_{a \in C} x_a \geq 1, \quad \forall C \in \mathcal{C}, \quad (2.1b)$$

$$x_a \in \{0, 1\}, \quad \forall a \in A. \quad (2.1c)$$

For the weighted version, the objective function has to be replaced by

$$\text{minimize } \sum_{a \in A} w(a)x_a. \quad (2.1a')$$

As graphs are likely to have an exponential number of cycles, this modeling is rather unsuitable in practice. Instead, similar as in the MATRIX TRIANGULATION problem, a weighted graph  $G^+$  with the same number  $n$  of vertices is considered that has an arc between each ordered(!) pair of vertices. The arcs' weights are denoted and determined as for the adjacency matrix, i. e., if there is an arc from vertex  $i$  to vertex  $j$ ,  $h_{ij}$  is the weight of this arc (or 1 in case of an unweighted graph), and  $h_{ij} = 0$  otherwise. For any linear ordering of  $G^+$ , both the subgraph consisting of forward arcs only and the subgraph consisting of backward arcs only are tournaments. If the arc directed from vertex  $i$  to vertex  $j$  is forward, then the arc from  $j$  to  $i$  is backward and vice versa. Furthermore, a tournament is acyclic if and only if it does not contain a cycle of length three. Replacing the variables  $x_a$  by  $x_{ij}$ , the LINEAR ORDERING problem and the WEIGHTED LINEAR ORDERING problem can thus be rewritten as follows:

$$\text{minimize } \sum_{0 \leq i, j < n} h_{ij}x_{ij} \quad (2.2a)$$

subject to

$$x_{ij} + x_{ji} = 1, \quad \forall 0 \leq i < j < n, \quad (2.2b)$$

$$x_{ij} + x_{jk} + x_{ki} \geq 1, \quad \forall 0 \leq i < \{j, k\} < n, j \neq k, \quad (2.2c)$$

$$x_{ij} \in \{0, 1\}, \quad \forall 0 \leq i < j < n. \quad (2.2d)$$

In contrast to the original linear program, this formulation only has  $\binom{n}{2} + 2\binom{n}{3}$  constraints—a polynomial.

The canonical linear programming relaxation is obtained in each case by replacing Equation (2.1c) and Equation (2.2d) with

$$0 \leq x_a \leq 1, \quad \forall 0 \leq i < j < n \quad (2.1c^*)$$

and

$$0 \leq x_{ij} \leq 1, \quad \forall 0 \leq i < j < n, \quad (2.2d^*)$$

respectively. The value of the objective function of the 0-1 integer linear programs is the same for both Equation (2.1a) and Equation (2.2a) and usually denoted as  $\tau_G$  or simply  $\tau$ . Nutov and Penn [NP95] showed that the equality also holds for the objective function value of the relaxed versions. The relaxation's objective function value is usually denoted

by  $\tau_G^*$  or  $\tau^*$ . As every solution of the integer linear program is also a feasible, though not necessarily optimal, solution of its relaxation,  $\tau^* \leq \tau$ . The solution associated with the relaxation, i. e., the assignments of the variables  $x_a$  or  $x_{ij}$ , respectively, is also referred to as a *fractional solution*.

### 2.1.3 Dual Problems

Basically, the dual of a problem can be thought of as its counterpart in some respect. For LO, there are two problems that are commonly considered “dual” to it.

In the first case, we only switch view in that we are no longer seeking a linear ordering that minimizes the number of arcs contradicting it. Instead, we aim at maximizing the complementary arc set, i. e., the number of arcs that agree. The solution remains the same: Any linear ordering that maximizes the number of forward arcs also minimizes the number of backward arcs and vice versa. Transferred to the FEEDBACK ARC SET view, the corresponding problem is known as the ACYCLIC SUBGRAPH problem:

---

#### **The Acyclic Subgraph Problem (AS)**

**Instance:** directed graph  $G$  with arc set  $A$

**Question:** What is the largest subset  $\mathcal{F} \subseteq A$  such that the subgraph of  $G$  restricted to  $\mathcal{F}$  is acyclic?

---

The corresponding weighted and/or decision versions are derived in an analogous manner as in [Section 2.1.1](#). Similarly, a 0-1 integer linear program for ADC can be formulated. For the cycle set version (cf. [Equation \(2.1\)](#)), the corresponding linear program reads

$$\text{maximize } \sum_{a \in A} x_a \quad (2.3a)$$

subject to

$$\sum_{a \in \mathcal{C}} x_a \leq |\mathcal{C}| - 1, \quad \forall \mathcal{C} \in \mathcal{C}, \quad (2.3b)$$

$$x_a \in \{0, 1\}, \quad \forall a \in A, \quad (2.3c)$$

where  $|C|$  denotes the length of cycle  $C$ , i. e., its number of arcs. For the three-cycle version (cf. Equation (2.2)), the according linear program is

$$\text{maximize } \sum_{0 \leq i, j < n} h_{ij} x_{ij} \quad (2.4a)$$

subject to

$$x_{ij} + x_{ji} = 1, \quad \forall 0 \leq i < j < n, \quad (2.4b)$$

$$x_{ij} + x_{jk} + x_{ki} \leq 2, \quad \forall 0 \leq i < \{j, k\} < n, j \neq k, \quad (2.4c)$$

$$x_{ij} \in \{0, 1\}, \quad \forall 0 \leq i < j < n. \quad (2.4d)$$

Another notion of “dual” originates from mathematical optimization theory and is closely linked to linear programming. In short, both linear programs introduced in Section 2.1.2 could be written as

$$\text{minimize } \mathbf{h}^\top \mathbf{x} \quad (2.5a)$$

subject to

$$\mathbf{C}\mathbf{x} \geq \mathbf{b}, \quad (2.5b)$$

$$\mathbf{x} \geq \mathbf{0}. \quad (2.5c)$$

Let  $m$  denote the number of arcs in the graph under consideration and  $\mathcal{C}$  be again the set of all cycles. For the (relaxation of the) first linear program in Section 2.1.2,  $\mathbf{x}$  is an  $m \times 1$  vector whose entries correspond to the variables  $x_a$  and  $\mathbf{h}$  is a vector of the same dimension with all entries set to 1.  $\mathbf{C}$  is a  $|\mathcal{C}| \times m$  matrix with  $\mathbf{C}_{ij} = 1$  if the  $j$ th arc is part of the  $i$ th cycle and  $\mathbf{C}_{ij} = 0$  otherwise.  $\mathbf{b}$  is a constant-1 vector of size  $|\mathcal{C}| \times 1$ .

The appertaining dual of this linear program is:

$$\text{maximize } \mathbf{b}^\top \mathbf{y} \quad (2.6a)$$

subject to

$$\mathbf{C}^\top \mathbf{y} \leq \mathbf{h}, \quad (2.6b)$$

$$\mathbf{y} \geq \mathbf{0}. \quad (2.6c)$$

Here,  $\mathbf{y}$  is a vector of size  $|\mathcal{C}| \times 1$  and selects a maximum subset of  $\mathcal{C}$  with the restriction imposed by Equation (2.6b) that the  $j$ th arc may occur in at most  $\mathbf{h}_j$  of the selected cycles, i. e., in case of an unweighted graph, the cycles must be arc-disjoint.

If we treat both programs as 0-1 integer linear programs for unweighted graphs, the dual linear program is equivalent to the following graph problem:



---



---

### The Arc-Disjoint Cycles Problem (ADC)

**Instance:** directed graph  $G$  with arc set  $A$

**Question:** What is the largest collection of pairwise arc-disjoint cycles?

---

The objective function value of the dual program with the requirement that all variables are integral is usually denoted by  $\nu_G$  or  $\nu$  and is also used in general for the cardinality of an optimal solution to ADC. The objective function value of the canonical relaxation is denoted as  $\nu_G^*$  or  $\nu^*$  and, as it is a maximization problem,  $\nu_G^* \geq \nu_G$ . Furthermore, the strong duality theorem implies that  $\tau_G^* = \nu_G^*$ , whereas the weak duality theorem yields  $\nu_G \leq \tau_G$ .

#### 2.1.4 The Acyclic Subgraph Polytope

One approach to studying the LINEAR ORDERING and the FEEDBACK ARC SET problem results from a closer examination of the structure of the linear programs. Commonly, the perspective of the ACYCLIC SUBGRAPH problem is taken here.

Consider a graph  $G$  with arc set  $A$  of cardinality  $m$ . Then, every subset  $Y \subseteq A$  can be represented by an  $m$ -dimensional vector  $\mathbf{x}^Y$  that has an entry for every arc  $a \in A$ . The value  $\mathbf{x}_a^Y = 1$  indicates that  $a \in Y$ , whereas  $\mathbf{x}_a^Y = 0$  means that  $a \notin Y$ . The *acyclic subgraph polytope*  $\mathbb{P}_{AS}(G)$  of  $G$  can then be defined as the polytope in  $\mathbb{R}^m$  that is the convex hull of all vectors that represent an acyclic arc set of  $G$ . With this notion, the linear program in Equation (2.3) could be rewritten succinctly as

$$\text{maximize } \mathbf{c}^\top \mathbf{x}, \quad \mathbf{x} \in \mathbb{P}_{AS}(G), \quad (2.7)$$

where  $\mathbf{c}$  is the  $m$ -dimension 1-vector in the unweighted case and otherwise  $\mathbf{c}_a = w(a)$ . The (vector representing the) optimal solution to this linear program is always integral due to the modeling and, if  $G$  is unweighted, equals  $m - \tau_G$  and otherwise the respective arc weight sums. The problem, however, lies in the definition of  $\mathbb{P}_{AS}(G)$ , which does not allow for an application of standard linear programming techniques. Thus, one focus of interest lies in a description of  $\mathbb{P}_{AS}(G)$  by an (ideally polynomially sized) system of inequalities.

The inequalities given in Equation (2.3) also define a polytope in  $\mathbb{R}^m$ , which is commonly referred to as  $\mathbb{P}_C(G)$ . As every acyclic arc set must satisfy the inequality constraints of Equation (2.3), the polytope  $\mathbb{P}_C(G)$  contains  $\mathbb{P}_{AS}(G)$ . In contrast to  $\mathbb{P}_{AS}(G)$ , however,  $\mathbb{P}_C(G)$ 's extreme points are not necessarily integral. Instead,  $\mathbb{P}_{AS}(G)$  equals the

convex hull of all integral points of  $\mathbb{P}_C(G)$  [GJR85b]. Grötschel *et al.* [GJR85b] termed a graph *weakly acyclic*, if  $\mathbb{P}_{AS}(G) = \mathbb{P}_C(G)$ . Thus,  $\tau_G = \tau_G^* = \nu_G^* = \nu_G$  in this case. Graphs with  $\mathbb{P}_{AS}(G) \neq \mathbb{P}_C(G)$  are conversely said to be *strongly cyclic* [GJR85b]. For such graphs, the description of  $\mathbb{P}_{AS}(G)$  requires further, *facet-defining* inequalities. Such inequalities are obtained from certain graphs including, e. g., 2- and 3-cycles,  $k$ -fences, Möbius ladders, and several more [GJR85b, GJR85a, GKN98, MR11]. If  $G$  is not triconnected, a description of  $\mathbb{P}_{AS}(G)$  can be obtained from the description of the acyclic subgraph polytopes of its triconnected components and if every triconnected component  $H$  satisfies  $\tau_H = \nu_H$ , then so does  $G$  [BFM94, NP95].

A slightly different polytope that is often studied in the same context is the *linear ordering polytope*  $\mathbb{P}_{LO}^n$ , which is defined as the convex hull of all acyclic subtournaments of the complete graph on  $n$  vertices, i. e., it has an arc between every ordered pair of vertices. For the sake of brevity, we will denote this graph with  $G_n$  in the following. The polytope  $\mathbb{P}_C^n$  is the polytope defined by Equation (2.3) with respect to  $G_n$ , i. e.,  $\mathbb{P}_C^n = \mathbb{P}_C(G_n)$  and thus contains in particular  $\mathbb{P}_{LO}^n$ . As we only consider acyclic subgraphs that are tournaments here,  $\mathbb{P}_{LO}^n$  is a face of  $\mathbb{P}_{AS}(G_n)$ . Dridi [Dri80] proved that  $\mathbb{P}_{LO}^n = \mathbb{P}_C^n$  for  $n \leq 5$ , whereas for  $n > 5$ ,  $\mathbb{P}_{LO}^n \subsetneq \mathbb{P}_C^n$ . Reinelt [Rei93] explicitly listed the facets for  $G_6$  and  $G_7$ .

Although the ACYCLIC SUBGRAPH problem is primarily considered here, the results can be transferred straightforwardly to the LINEAR ORDERING and the FEEDBACK ARC SET problem. As a matter of fact, there is a one-to-one correspondence between the vertices of the polytopes obtained from the respective definitions with regard to LO/FAS and the acyclic subgraph and linear ordering polytope [GJR85b].

### 2.1.5 Reductions to and from the Linear Ordering Problem

Apart from the dual problems, LINEAR ORDERING is unmediatedly linked to a number of other well-known problems via polynomial time reductions. In many cases, these relationships also allow to deduce facts for LO/FAS concerning, e. g., complexity or approximability. For this reason, we list a selection of the most relevant or most frequently considered ones here. In order to keep the problem descriptions concise, we do not explicitly state the respective decision versions; they can be obtained similar as in Section 2.1.1 by introducing a positive integer  $k$  as supplementary parameter that serves as an upper bound. All problems considered here, except for the last, reference primarily the FEEDBACK ARC SET view.

The problem with the intuitively closest connection to FAS emerges by substituting the quest for arcs by a quest for vertices:

---



---

### The Feedback Vertex Set Problem (FVS)

**Instance:** directed graph  $G$  with vertex set  $V$

**Question:** What is the smallest subset  $X \subseteq V$  that contains at least one vertex of every cycle in  $G$ ?

---

The corresponding WEIGHTED FEEDBACK VERTEX SET problem (wFVS) is formulated by additionally specifying a weight function  $w$  on the vertices and minimizing  $\sum_{v \in X} w(v)$  instead of  $|X|$ .

There are approximation-preserving reductions between FAS and FVS in both directions (see, e. g., [ENSS98]):

For  $FAS \preceq FVS$ , we need the notion of a graph's *line graph*. Let  $G$  be a directed graph with vertex set  $V$  and arc set  $A$ . The line graph of  $G$ , which we denote as  $L(G)$ , is a graph with vertex set  $A$  and an arc from a vertex  $a_1 \in A$  to a vertex  $a_2 \in A$  if the head of the arc  $a_1$  in  $G$  coincides with the tail of the arc  $a_2$  in  $G$ , i. e., there is a vertex  $v \in V$  such that  $a_1$  enters  $v$  and  $a_2$  leaves  $v$ . Let now  $G$  be an instance of FAS. Then,  $L(G)$  is an instance of FVS. Moreover,  $\mathcal{B}$  is a subset of  $G$ 's arcs covering all cycles of  $G$  if and only if  $\mathcal{B}$  is a subset of  $L(G)$ 's vertices covering all cycles of  $L(G)$ . In case that the weighted versions are considered, i. e.,  $wFAS \preceq wFVS$ , every vertex of the line graph directly inherits the respective arc's weight.

For  $FVS \preceq FAS$ , we employ a routine called vertex splitting: Let  $G$  with vertex set  $V$  and arc set  $A$  be an instance of FVS. Construct a new graph  $G'$  from  $G$  by replacing each vertex  $v \in V$  by two vertices  $v_{in}$  and  $v_{out}$  along with an arc  $a_v$  that is directed from  $v_{in}$  to  $v_{out}$ . For every arc from  $u$  to  $v$  in  $G$ ,  $G'$  contains an arc from  $u_{out}$  to  $v_{in}$ . Then, if  $X$  is a subset of  $G$ 's vertex set covering all cycles of  $G$ , then the set of arcs  $\{a_x \mid x \in X\}$  covers all cycles in  $G'$ . Conversely, if  $\mathcal{B}$  is a subset of  $G'$ 's arc set covering all cycles of  $G'$ , then a subset of  $G$ 's vertices covering all cycles of  $G$  can be obtained as follows: Substitute every arc  $a$  in  $\mathcal{B}$  that did not result from the splitting of a vertex by the arc  $a_v$ , where  $v$  is either the head or the tail of the arc corresponding to  $a$  in  $G$ . Compared to  $\mathcal{B}$ , this modified arc set is equal or less in size and still covers all cycles of  $G'$ . The vertex set covering all cycles of  $G$  then consists of all vertices  $v \in V$  such that  $a_v$  is in the modified arc set. For  $wFVS \preceq wFAS$ , the arcs resulting from the splitting operation inherit the respective vertex's weight, whereas the weight of all other arcs of  $G'$  is set to infinity.

A classic problem with a close linkage to FAS that is concerned with undirected graphs is VERTEX COVER:

---



---

### **The Vertex Cover Problem (VC)**

**Instance:** undirected graph  $U$  with vertex set  $V$

**Question:** What is a smallest subset  $X \subseteq V$  such every edge has at least one end vertex in  $X$ ?

---

As its name suggests, VC is a covering problem, as is FAS. However, the task here is to cover all edges of an undirected graph by selecting at least one of its end vertices.

The reduction  $VC \preceq FAS$  is approximation-preserving [Kan92] and has been described first in [Kar72]: Consider an undirected graph  $U$  with vertex set  $V$  and edge set  $E$  as an instance of VC. Similar to the reduction from FVS, we construct a directed graph  $G$  from  $U$  that contains two vertices  $v_{in}, v_{out}$  for every vertex  $v \in V$  along with an arc  $a_v$  that is directed from  $v_{in}$  to  $v_{out}$ . Furthermore, for every undirected edge  $e \in E$  connecting vertices  $u$  and  $v$  in  $U$ ,  $G$  contains an arc from  $u_{out}$  to  $v_{in}$  as well as an arc from  $v_{out}$  to  $u_{in}$ . Note that this construction yields a directed cycle of length four for every edge in  $U$ . A subset of vertices  $X \subseteq V$  now covers all edges of  $U$  if and only if the arc set  $\{a_x \mid x \in X\}$  covers all cycles of  $G$ . A vertex cover  $X$  of  $U$  can be constructed from any feedback arc set  $\mathcal{B}$  of  $G$  and such that  $|X| \leq |\mathcal{B}|$  in the same way as in the reduction  $FVS \preceq FAS$ .

A generalization [ENSS98] of FEEDBACK ARC SET results from the specification of an additional set of “interesting” vertices (a definition via arcs is also possible) for the problem instance and the subsequent restriction of the task to only cover cycles containing at least one interesting vertex or arc:

---



---

### **The Subset Feedback Arc Set Problem (SUBSET-FAS)**

**Instance:** directed graph  $G$  with vertex set  $V$  and arc set  $A$ ,  $I \subseteq V$

**Question:** What is the smallest subset  $\mathcal{B} \subseteq A$  that contains at least one arc of every cycle of  $G$  that contains an element of  $I$ ?

---

Considering  $I = V$ , the reduction  $FAS \preceq SUBSET-FAS$  becomes immediately apparent as does the fact that approximations are preserved. SUBSET FEEDBACK VERTEX SET (SUBSET-FVS) and the respective weighted versions, SUBSET WEIGHTED FEEDBACK ARC SET (SUBSET-wFAS) and SUBSET WEIGHTED FEEDBACK VERTEX SET (SUBSET-wFVS), can be defined accordingly.

Next, we want to instance a problem that FAS has an approximation-preserving reduction to, but as yet there is none known for the opposite direction:

---



---

### **The Directed Multicut Problem (DMC)**

**Instance:** directed network  $N$  with arc capacities  $c$ ,  $2k$  vertices forming source-sink pairs  $s_i, t_i, 0 \leq i < k$

**Question:** What is a minimum capacity set of arcs whose removal disconnects each source  $s_i$  from the sink  $t_i$ ?

---

In fact, DMC is equivalent to SUBSET-wFAS [ENSS98], which is demonstrated by the following transformations:

For SUBSET-wFAS  $\preceq$  DMC, construct a directed network  $N$  from a graph  $G$  that contains a source  $s_v$  and a sink  $t_v$  for every vertex  $v$  contained in the set  $I$  that defines the interesting cycles, i. e.,  $k = |I|$ . All vertices not contained in  $I$  have a single counterpart in  $N$ . Furthermore, there is an edge with infinite capacity from  $t_v$  to  $s_v, \forall v \in I$ . Every arc directed from vertex  $u$  to vertex  $v$  in  $G$  is represented by an arc emanating from  $s_u$ , if  $u \in I$ , otherwise the counterpart of  $u$ , and entering  $t_v$ , if  $v \in I$ , otherwise  $v$ 's counterpart. The capacity of every such arc in  $N$  equals the arc's weight in  $G$ . A minimum capacity set of arcs that separates all source-sink pairs in  $N$  then directly corresponds to a minimum weighted set of arcs covering all interesting cycles of  $G$ .

For DMC  $\preceq$  SUBSET-wFAS, the directed graph  $G$  is constructed from a directed network  $N$  by identifying each source  $s_i$  with its sink  $t_i$  and adopting all arcs such that the weight of each corresponds to its capacity in  $N$ . The set  $I$  consists of all vertices resulting from the identification of the source-sink pairs.

In Section 2.1.2, we have already seen how to cast LO/FAS as an integer linear program, which is not purely of theoretical interest, but in particular also enables us to find a solution to a concrete instance in practice by means of LP solvers. Another tool that is widely used to solve complex problems are so-called SAT solvers. To round off, we therefore also show how to construct a boolean formula for an instance of dLO, or, more formally, how to reduce dLO to SAT.

---



---

### **The Boolean Satisfiability Problem (SAT)**

**Instance:** boolean expression  $F$

**Question:** Is there an assignment of true and false to the variables in  $F$  such that  $F$  evaluates to true?

---

The reduction  $\text{dLO} \preceq \text{SAT}$  works on the linear ordering perspective: Given a graph  $G$  with vertex set  $V$  and arc set  $A$  and an integer  $k$ , we construct a boolean expression  $F$  that is satisfiable if and only if  $G$  has a linear ordering with at most  $k$  backward arcs. To this end, we create variables  $v_i, \forall v \in V \forall 0 \leq i < |V|$ , that serve the representation of a linear ordering  $\pi$ : If  $v_i = \text{true}$ , then  $\pi(v) = i$ . Thus,  $F$  must ensure that for every vertex  $v$  exactly one of the variables  $v_i$  is  $\text{true}$ . In order to restrict the number of backward arcs to at most  $k$ , we also create variables  $a_l, \forall a \in A \forall 0 \leq l < k$ . Similar as for the vertices,  $F$  has to ensure that for every value of  $l$  at most one arc  $a \in A$  exists such that  $a_l = \text{true}$ . Finally, the linear ordering represented via the variables  $v_i$  and the selection of the backward arcs represented via the variables  $a_j$  must be linked with each other by adding clauses to  $F$  that require for every arc  $a \in A$  that is directed from vertex  $u$  to vertex  $v$  and every position  $i, 0 \leq i < |V|$  that if  $u_i = \text{true}$  and there is no position  $j, i < j$  such that  $v_j = \text{true}$ , then there must be an  $l, 0 \leq l < k$ , such that  $a_l = \text{true}$ , i. e.,  $a$  must be selected as one of the at most  $k$  backward arcs. The interested reader is referred to [SW93] for a more detailed description of the construction of  $F$ .

## 2.2 Complexity

Having stated the problems under consideration and to some extent also discussed important interrelationships, we now turn to the classification of LO and FAS with respect to their computational complexity. In this context, we also consider some special classes of graphs as input instances.

### 2.2.1 $\mathcal{NP}$ -Completeness Results

The decision version of FEEDBACK ARC SET is one of Karp's 21  $\mathcal{NP}$ -complete problems [Kar72] that were published in 1972. Its  $\mathcal{NP}$ -hardness follows from the reduction from VERTEX COVER, which has already been stated in Section 2.1.5. dFAS is also listed as "Problem [GT8]" in Garey and Johnson's seminal book *Computers and Intractability: A Guide to the Theory of  $\mathcal{NP}$ -Completeness* [GJ79]. The  $\mathcal{NP}$ -hardness of FAS immediately implies the same hardness result for ACYCLIC SUBGRAPH as well as for all problems that have a polynomial-time reduction from FAS, in particular FVS and SUBSET-FAS. Also the ARC-DISJOINT CYCLES problem belongs in this complexity class [BG02].

There are a couple of algorithms with exponential running time that compute an optimal solution to the LINEAR ORDERING and FEEDBACK ARC SET problem. A large

family consists in so-called branch-and-cut or cutting plane approaches, which operate on the linear programming perspective and seek to move from a fractional to an integral solution. The monograph by Martí and Reinelt [MR11] provides a detailed overview. These approaches constantly undergo further improvements, cf. [BSN15]. Using a cutting plane algorithm, Grötschel *et al.* [GJR84a] solved concrete LO instances obtained from input-output matrices (cf. Section 2.4) of size up to  $60 \times 60$  as well as from the German Soccer Championship of 1981/82. Mushi [Mus05] used input-output matrices of size up to  $41 \times 41$  and compared the performance with respect to different configurations of the linear programming solver. With a combinatorial branch-and-bound approach, Kaas [Kaa81] was able to solve instances of 25 to 34 vertices by enumerating permutations. Note that the straightforward approach of testing all possible feedback arc sets requires  $\mathcal{O}^*(2^m)$  time, which is worse than  $\mathcal{O}^*(n!)$  for dense graphs. Flood [Flo90] developed an exact “screening” algorithm that iteratively constructs permutations and compared it experimentally to an approach that solves the more general quadratic assignment problem using instances with up to 20 vertices. Using dynamic programming, Raman and Saurabh [RS07] show that a minimum feedback arc set can be computed in  $\mathcal{O}^*(2^n)$  time and  $\mathcal{O}^*(2^n)$  space or, alternatively, in polynomial space and  $\mathcal{O}^*(4^{n+o(n)})$  time. Fomin and Kratsch [FK10] devise an algorithmic framework for solving various ordering problems, which is also based on the dynamic programming approach. In case of the LINEAR ORDERING problem, it yields a running time of  $\mathcal{O}(nm \cdot 2^n)$ , however again at the expense of exponential space.

The fact that FEEDBACK ARC SET is  $\mathcal{NP}$ -hard for general graphs immediately raises the question whether under special conditions, e. g., for certain classes of graphs, a solution may nevertheless be computable efficiently. In many cases, however, the answer is in the negative. Note that the reductions between FAS and FVS do not necessarily preserve the characteristics of a class.

One example are tournaments, which occur rather frequently in real-world application scenarios (cf. Section 2.4). In 1992, Bang-Jensen and Thomassen [BT92] conjectured that FAS remains  $\mathcal{NP}$ -hard if the problem instance is a tournament. Only in 2005, more than one decade later, Ailon *et al.* [ACN08] were able to give a randomized reduction from the general FEEDBACK ARC SET problem to FEEDBACK ARC SET on tournaments, thus proving that FAS on tournaments remains  $\mathcal{NP}$ -hard unless  $\mathcal{NP} \subseteq \mathcal{BPP}$ . Soon afterwards, Alon [Alo06] as well as Charbit *et al.* [CTY07] independently provided derandomized reductions, thus finally settling the conjecture. A different approach was taken by Conitzer [Con06] in an essay on Slater rankings (cf. Section 2.4), who



obtained the same hardness result using a reduction from the canonical decision version of MAX-SAT to dFAS on tournaments. Interestingly, the question of whether FVS remains  $\mathcal{NP}$ -hard on tournaments could already be confirmed in 1989 by Speckenmeyer [Spe89]. Bang-Jensen and Thomassen [BT92] also provide a different proof for this result.

An exact algorithm for FAS on tournaments has been developed by Raman *et al.* [RSS07] with a running time of  $\mathcal{O}(n^{\mathcal{O}(1)} \cdot 1.5541^m)$ . For FVS, Dom *et al.* [DGH<sup>+</sup>10] presented an algorithm with a time complexity of  $\mathcal{O}(1.709^n)$ . In both cases, fixed-parameter approaches (cf. Section 2.2.3) were the basis.

Another prominent class of graphs are bipartite tournaments. Just as in the previous case, the  $\mathcal{NP}$ -hardness of FVS, shown by Cai *et al.* [CDZ02] in 2002, had already been established when in 2006, Guo *et al.* [GGH<sup>+</sup>06] were able to also close the gap for FAS. Inspired by Conitzer's proof for tournaments, they gave a reduction from SAT and thereby showed not only that FAS remains  $\mathcal{NP}$ -hard on bipartite tournaments, but also for the more general case of  $c$ -partite tournaments, for any fixed  $c \geq 2$ .

Surprisingly on first sight, FAS also remains  $\mathcal{NP}$ -hard on Eulerian graphs. Every Eulerian graph can be decomposed into a set of pairwise arc-disjoint cycles by a simple greedy algorithm. This stirs up hope that the maximum number of pairwise arc-disjoint cycles might equal the size of a minimum feedback arc set, which, as both cardinalities must be integral, would imply that these graphs are weakly acyclic (cf. Section 2.1.2) and hence allow for a polynomial-time solution of FAS and ADC using LP solvers. This hope is dashed by Borobia *et al.* [BNP96], who provide a Eulerian graph on seven vertices with an optimal fractional solution of 4.5. Seymour [Sey96] showed, however, that equality indeed holds if the underlying undirected graph is linklessly embeddable in three-dimensional space or, equivalently, if it does not have a minor in the Petersen family. Finally, Perrot and Van Pham [PVP13] gave a reduction for FAS on general graphs to FAS on Eulerian graphs, thus settling  $\mathcal{NP}$ -hardness decisively.

For two further graph classes, the  $\mathcal{NP}$ -hardness of the FEEDBACK ARC SET problem has been known for a relatively long time: directed line graphs and graphs where every vertex has outdegree and indegree at most three [Gav77]. The situation is again similar for FVS:  $\mathcal{NP}$ -hardness has been shown for graphs where every vertex has outdegree and indegree at most two, planar graphs with outdegree and indegree at most three at each vertex [GJ79], as well as line graphs [Gav77].



### 2.2.2 Approximability and Approximations

Intractability, i. e.,  $\mathcal{NP}$ -hardness, of a problem almost automatically creates need for good approximation algorithms and simultaneously poses a new question: How close to the optimum solution can we get?

Let us start with the bad news: For general graphs, Kann [Kan92] showed that the FEEDBACK ARC SET problem is  $\text{MaxSNP}$ -hard, which also implies  $\mathcal{APX}$ -hardness. Consequently, unless  $\mathcal{P} = \mathcal{NP}$ , no polynomial-time approximation scheme for FAS can exist, which would allow to construct a solution in polynomial time that is at most  $(1 + \varepsilon)$  times bigger than the optimum, for any  $\varepsilon > 0$ . As a matter of principle,  $\mathcal{APX}$ -hardness still leaves the possibility to find a constant-factor approximation algorithm, i. e., an algorithm that guarantees that the quotient of the size of the delivered solution divided by that of the optimum is bounded by some constant  $c$ . Unfortunately, no such algorithm for FAS on general graphs is known presently. Furthermore, as VERTEX COVER  $\preceq$  FAS (cf. Section 2.1.5) via an approximation-preserving reduction, the fact that it is  $\mathcal{NP}$ -hard to approximate VC better than to a factor of  $10\sqrt{5} - 21 = 1.3606\dots$  [DS05] also applies to FAS. However, VERTEX COVER has a relatively simple factor-two approximation [GJ79, PS98] and is thus  $\mathcal{APX}$ -complete, whereas  $\mathcal{APX}$ -completeness is still an open question for FAS.

It is important to note here that the closely related ACYCLIC SUBGRAPH problem has a constant-factor approximation with ratio  $\frac{1}{2}$ : Take an arbitrary linear ordering of the vertices of the input graph. If the number of backward arcs induced thereby exceeds the number of forward arcs, reverse the ordering. Then, if the graph has  $m$  arcs, the forward arcs form an acyclic subgraph containing at least  $\frac{m}{2}$  arcs. As the optimum solution is at most  $m$ , we obtain a guaranteed approximation ratio of  $\frac{1}{2}$ . Thus, AS is  $\mathcal{APX}$ -complete. Due to a result by Papadimitriou and Yannakakis [Pap91], AS is also complete for  $\text{MaxSNP}$ .

In the context of hardness of approximation, a conjecture postulated by Khot [Kho02] in 2002, which is widely known as the *Unique Games Conjecture*, plays an influential role. One of several equivalent formulations is based on the so-called LABEL COVER problem:

---



---

### The Unique Label Cover Problem

**Instance:** undirected, edge-weighted, complete bipartite graph  $U$  with edge set  $E$ , a set of vertex labels  $M$ , for every edge  $e \in E$  a bijection  $p_e : M \rightarrow M$

**Question:** What label assignment maximizes the total weight of those edges  $e = \{u, v\}$  with the property that  $p_e$  maps the label of  $u$  to the label of  $v$ ?

---

The *Unique Games Conjecture* now states that for a pair of constants  $(c_1, c_2)$  sufficiently small, there is a constant  $k$  depending on  $c_1$  and  $c_2$  such that it is  $\mathcal{NP}$ -hard to distinguish whether an instance of UNIQUE LABEL COVER with  $|M| = k$  has an optimum value of at least  $1 - c_1$  or at most  $c_2$ . If this conjecture could be confirmed, it would prove a number of classic  $\mathcal{NP}$ -hard optimization problems to be inapproximable below some specific bound [Kho10]. Among these problems is not only VC (with a bound of 2 [KR08]), but most notably also the FEEDBACK ARC SET problem. In particular, Guruswami *et al.* [GMR08] prove that on condition of the Unique Games Conjecture, FAS is inapproximable within a constant-factor. They also show that if the conjecture is true, ACYCLIC SUBGRAPH cannot be approximated within a factor better than  $\frac{1}{2}$ .

On the positive side, Even *et al.* [ENSS98] showed that FAS (in fact, SUBSET-wFAS) can be approximated to a factor of  $\mathcal{O}(\min\{\log \tau^* \log \log \tau^*, \log n \log \log n\})$  in polynomial time, where  $\tau^*$  denotes again the optimum fractional solution (cf. Section 2.1.2) and  $n$  the number of vertices. Their algorithm is based on a result by Seymour [Sey95], who proved that the integrality gap, i. e., the maximum ratio between the optimum integral and the optimum fractional solution, is at most  $\mathcal{O}(\log \tau^* \log \log \tau^*)$  in the unweighted case. Alternatively, the authors provide an  $\mathcal{O}(\log^2 |X|)$ -approximation algorithm for SUBSET-wFAS, where  $X$  identifies the set of interesting cycles (cf. Section 2.1.5). For FAS, Even *et al.* improve a result by Leighton and Rao [LR88], who obtained a factor of  $\mathcal{O}(\log^2 n)$  via the computation of approximately sparsest cuts using linear programming. Klein *et al.* [KST90] replaced this dependency with a randomized algorithm. Demetrescu and Finocchi [DF03] were able to show that the straightforward approach of identifying and destroying cycles step by step yields an  $\mathcal{O}(m \cdot n)$ -time algorithm which guarantees a solution that exceeds the optimum by a factor of at most the length of the longest cycle in the graph.

There are further encouraging results if we consider FAS again on special classes of graphs. For tournaments, Ailon *et al.* [ACN08] presented a randomized 3-approximation algorithm called KwikSort that constructs a linear ordering in a similar manner as

the well-known *QuickSort* algorithm sorts numbers. The authors also extended this approach to tackle *wFAS* on tournaments with various kinds of weight constraints, which led to different approximation ratios. Van Zuylen and Williamson [vZW09] were able to obtain respective derandomized versions while preserving approximation guarantees. Shortly after the conference version of [ACN08], Kenyon-Mathieu and Schudy [KS06, KS07] designed a polynomial-time approximation scheme (PTAS) for *WEIGHTED FEEDBACK ARC SET* on tournaments. Their algorithm is based on earlier results by Arora *et al.* [AFK02] as well as Frieze and Kannan [FK99], who obtained polynomial-time approximation schemes for the *ACYCLIC SUBGRAPH* problem on  $n$ -vertex graphs whose number of arcs is in  $\Omega(n^2)$ . To complete the picture, we note that for *FVS* on tournaments, a 3-approximation algorithm has been known since 1989 and is due to Speckenmeyer [Spe89]. In 2000, Cai *et al.* [CDZ00] were able to improve the ratio down to 2.5 and, only recently, Mnich *et al.* [MWV15] even achieved  $\frac{7}{3}$ .

Finally, there are also approximation results for bipartite and multipartite tournaments: Gupta [Gup08] presented both randomized and deterministic 4-approximation algorithms for *FAS* on these classes of graphs; van Zuylen [vZ11], however, pointed out an error in the correctness proof. Nevertheless, she was able to confirm the findings for *wFAS* on bipartite tournaments by designing a different algorithm. Her work also includes a 2-approximation algorithm for *wFVS* on bipartite tournaments, thereby improving on earlier results by Cai [CDZ02] and Sasatte [Sas08a], who achieved a factor of 3.5 and 3, respectively.

### 2.2.3 Parameterized Complexity

Besides studying approximability, an alternative approach to dealing with  $\mathcal{NP}$ -hard problems consists in their classification with respect to *parameterized complexity*. To this end, an instance  $x$  of the decision problem under consideration is equipped with an additional parameter  $k$ . The complexity of deciding whether  $(x, k)$  is a YES- or a NO-instance is then measured with regard to both the size of  $x$  and the parameter  $k$ . Such a parameterized problem is said to be *fixed-parameter tractable* and belongs to the complexity class  $\mathcal{FPT}$  if there is an algorithm which solves the problem in time  $|x|^{\mathcal{O}(1)} \cdot f(k)$ , where  $f$  is some function depending only on  $k$ , but not on  $x$ . Self-evidently, the parameterized complexity of a problem is strongly dependent on the choice of the parameter  $k$ .

In the case of dFAS, it seems natural to use the upper bound on the size of the feedback arc set as parameter  $k$  and the sum of the feedback arcs' weights for wdFAS. Indeed, Chen *et al.* [CLL07] showed in 2007 that dFAS is in  $\mathcal{FPT}$  with this choice of  $k$  by constructing an algorithm with a running time of  $\mathcal{O}(n^{\mathcal{O}(1)} \cdot (1.48k)^k)$ <sup>1</sup> on an  $n$ -vertex graph. A superset of the authors [CLL<sup>+</sup>08] also published a faster algorithm that runs in  $\mathcal{O}(n^{\mathcal{O}(1)} \cdot 4^k k^3 k!)$ <sup>2</sup> time. Both algorithms were originally designed for dFVS, but can be transferred to dFAS<sup>3</sup> using the reduction described in Section 2.1.5. Raman and Saurabh [RS06] proved that also dAS is fixed-parameter tractable with  $k$  being the number of arcs in the acyclic subgraph and that it can be solved in time  $\mathcal{O}(4^k k + m)$ .

Further results again target specific classes of graphs. Already in 2003, Raman and Saurabh [RS03] constructed an  $\mathcal{O}(n^\omega \log n \cdot (c\sqrt{k/e})^k)$ -time algorithm for dFAS on tournaments, where  $\omega$  is the exponent in the running time of the best matrix multiplication algorithm,  $e$  is the base of the natural logarithm, and  $c$  some constant. In a later version, Raman and Saurabh [RS06] were able to improve the running time further to  $\mathcal{O}(n^\omega \cdot 2.415^k)$ , now also including the weighted version, wdFAS, if all arcs have weight at least one. The currently best known  $\mathcal{FPT}$ -algorithms for both unweighted and weighted (with weights at least one) FAS on tournaments are due to Alon *et al.* [ALS09] with a running time of  $2^{\mathcal{O}(\sqrt{k} \log^2 k)} + n^{\mathcal{O}(1)}$  and Karpinski and Schudy [KS10] with a running time of  $2^{\mathcal{O}(\sqrt{k})} + n^{\mathcal{O}(1)}$ . Bessy *et al.* [BFG<sup>+</sup>11] showed how to compute a kernel with  $\mathcal{O}(k)$  vertices in polynomial time.

If we consider the vertex version, i. e., dFVS, on tournaments, an  $\mathcal{O}(n^3 \cdot 3^k)$ -time algorithm can be obtained straightforwardly from the observation that a graph is acyclic if and only if it does not contain a directed triangle. Dom *et al.* [DGH<sup>+</sup>10] showed that dFVS on tournaments can be solved in time  $\mathcal{O}(n^2(\log \log n + k) \cdot 2^k)$ . For wdFVS, the best result is an  $\mathcal{O}(n^\omega \cdot 2.4143^k)$ -time algorithm by Raman and Saurabh [RS06].

On bipartite tournaments, Dom *et al.* [DGH<sup>+</sup>10] also provided a  $\mathcal{O}(n^6 \cdot 3.373^k)$ -time algorithm for dFAS. For dFVS, there is an  $\mathcal{O}(n^2 \cdot 3^k + n^3)$ -time algorithm due to Sasatte [Sas08b].

<sup>1</sup>The exact running time given is  $\mathcal{O}(n^4 \cdot (1.48k)^k k^{2.5})$  for dFVS, which can be improved to  $\mathcal{O}(n^3 \cdot (1.48k)^k k^{3.5} \log k \log \log k)$  using the approximation by [ENSS98].

<sup>2</sup>The exact running time given is  $\mathcal{O}(n^4 \cdot 4^k k^3 k!)$  for dFVS, which can be improved to  $\mathcal{O}(n^3 \cdot 4^k k^4 k! \log k \log \log k + n^{4.376} \log^2 n)$  using the approximation by [ENSS98].

<sup>3</sup>The authors claim in both cases that the algorithm for dFAS, obtained via the reduction, has the same running time as for dFVS. As the line graph needed for this purpose contains  $m$  instead of  $n$  vertices, this seems incorrect in the general case. As  $m \in \mathcal{O}(n^2)$ , the exponent in the polynomial depending on  $n$  doubles for dFAS.

### 2.2.4 Polynomially Solvable Instances

Fortunately, there are also a couple of graph classes that admit the solution of the FEEDBACK ARC SET problem in polynomial time.

One of the oldest results dates back to 1976 and relates to the well-studied class of planar graphs. Lucchesi and Younger [Luc76, LY78] showed that the cardinality of a minimum dijoin, i. e. , a set of arcs covering all directed cuts of a given graph, equals the size of a maximum set of pairwise disjoint directed cuts. This minimax equality has an immediate effect for FAS on planar graphs, as there is a one-to-one correspondence between a cycle in the graph and directed cut in its planar dual. A minimum dijoin in the planar dual graph thus translates directly to a minimum feedback arc set in the original graph. The former can be computed efficiently in time  $\mathcal{O}(n^4)$  for both unweighted and weighted graphs via an algorithm by Frank [Fra81]. Targeting especially its application with respect to the FEEDBACK ARC SET problem, de Mendonça Neto and Eades [dMNE99] developed a set of improvements that lead to a significant speedup in practice. A very comprehensible and nicely illustrated review of Frank's algorithm can be found in [Eck15]. Alternatively, there also is an algorithm by Karzanov [Kar79] that computes a minimum dijoin in polynomial time. Stamm [Sta90] additionally developed a factor-2 approximation algorithm for FAS on planar graphs that runs in time  $\mathcal{O}(n \log n)$ .

In 1990, Ramachandran [Ram90] proved the minimax equality also for reducible flow graphs. A graph is called a *reducible flow graph* if it contains a vertex  $v$  such that a depth-first search traversal starting at  $v$  is unique. The proposed algorithm runs in time  $\mathcal{O}(\min(m^2, mn^{5/3}))$  and operates on weighted and unweighted graphs. By combining an earlier approach of Ramachandran [Ram88] with contraction operations, Koehler [Koe05] was able to improve the running time to  $\mathcal{O}(m \log n)$ .

A result that has already been mentioned in Section 2.2.1 is the polynomial-time tractability of FEEDBACK ARC SET on Eulerian graphs that are linklessly embeddable in three-dimensional space [Sey96]. A graph is said to be *linklessly embeddable*, if it can be embedded such that for any two closed undirected paths, there is a disk that bounds one path, but does not contain the other.

A further class of graphs, which actually comprises the above mentioned classes, constitute all graphs with the property  $\tau = \nu$ . Here, the ellipsoid method [GJR85b] can be applied to efficiently solve the LINEAR ORDERING and the FEEDBACK ARC SET problem. Alternatively, we can use the formulation given in Equation (2.4) together with any polynomial-time linear programming algorithm [NP95]. This class also contains all

graphs whose underlying undirected graph does not have the “utility graph”  $K_{3,3}$  as minor [BFM94, NP95].

Regarding FEEDBACK VERTEX SET, we observe a considerable difference in complexity for planar graphs: FVS remains  $\mathcal{NP}$ -hard [Yan78]. For reducible flow graphs, however, there is even a linear-time algorithm for unweighted graphs [Sha79].

## 2.3 The Cardinality of Optimal Feedback Arc Sets

Given an unweighted graph on  $n$  vertices and  $m$  arcs, how large can an optimal feedback arc set at most be? This question has been posed relatively early in the history of FAS. For an answer, one can either try to construct or prove the existence of graphs whose feedback arc set has a particular size, or via the analysis of an algorithm’s performance. The absolute lower bound here is trivial: A graph may be acyclic and thus its optimal feedback arc set is the empty set.

We may, however, also ask the question in a slightly different way: Let  $\mathcal{G}_{n,m}$  denote the set of all unweighted graphs with  $n$  vertices and  $m$  arcs and consider

$$\tilde{\tau}_{n,m} = \max_{G \in \mathcal{G}_{n,m}} \tau_G,$$

where  $\tau_G$  denotes the cardinality of an optimal feedback arc set of  $G$ . How large is  $\tilde{\tau}_{n,m}$ ?

### 2.3.1 Existential Bounds

The question was first addressed in 1965 by Erdős and Moon [EM65], who showed that in case of tournaments,

$$\frac{m}{2} - c_0(n^3 \log n)^{\frac{1}{2}} < \tilde{\tau}_{n,m} \leq m - \lfloor \frac{n}{2} \rfloor \cdot \lfloor \frac{n+1}{2} \rfloor, \quad (2.8)$$

where  $c_0$  is some constant. Furthermore, they showed for general graphs that

$$\text{if } \lim_{n,m \rightarrow \infty} \frac{n \log n}{m} = 0, \text{ then } \lim_{n \rightarrow \infty} \frac{\tilde{\tau}_{n,m}}{m} = \frac{1}{2}. \quad (2.9)$$

A couple of years later, Spencer [Spe71] was able to establish again for tournaments that

$$\frac{m}{2} - c_1 n^{\frac{3}{2}} \sqrt{\log n} \leq \tilde{\tau}_{n,m} \leq \frac{m}{2} - c_2 n^{\frac{3}{2}}, \quad (2.10)$$

for two constants  $c_1$  and  $c_2$ , and finally close the gap asymptotically by showing [Spe80]

$$\frac{m}{2} - c_3 n^{\frac{3}{2}} \leq \tilde{\tau}_{n,m} \quad (2.11)$$

for another constant  $c_3$ . Furthermore, Spencer obtained  $c_2 = 0.1577989 \dots$ , whereas de la Vega [dlV83] proved a value of 1.73 for  $c_3$ .

### 2.3.2 Algorithms with Absolute Performance Guarantees

The results presented in the previous subsection more or less settle the question for tournaments. What remains open, however, is how  $\tilde{\tau}_{n,m}$  behaves on sparser graphs.

In this context, an  $\mathcal{O}(m+n)$ -time algorithm by Berger and Shor [BS90] reveals interesting facts: Process the vertices of a graph in random order. For each vertex, take the smaller set of either all incoming or all outgoing arcs and add it to the feedback arc set, then remove the vertex along with all incident arcs from the graph and proceed. Berger and Shor discovered that this algorithm can be derandomized and yields

$$\pi_G \leq \frac{m}{2} - c_4 \frac{m}{\sqrt{\Delta_G}}, \quad (2.12)$$

where  $\Delta_G$  denotes the maximum degree of a vertex in  $G$  and  $c_4$  is again some constant. As  $m = \frac{n(n-1)}{2}$  and  $\Delta_G = n-1$  on tournaments, this result generalizes the upper bound in Equation (2.10). Furthermore, they provide concrete bounds for small values of  $\Delta_G$  and show that

$$\tilde{\tau}_{n,m} \leq \frac{5}{18}m \quad (2.13)$$

in case of  $\Delta_G = 3$  and

$$\tilde{\tau}_{n,m} \leq \frac{11}{30}m \quad (2.14)$$

if  $\Delta_G = 4$  or  $5$ . The derandomized version has a running time of  $\mathcal{O}(m \cdot n)$ .

Shortly thereafter, Eades *et al.* [ELS93] developed a faster,  $\mathcal{O}(m)$ -time algorithm and proved

$$\tilde{\tau}_{n,m} \leq \frac{m}{2} - \frac{n}{6}. \quad (2.15)$$

Depending on the maximum vertex degree, this bound is more precise or even better than that in Equation (2.12) for those sparse graphs having  $m \in \mathcal{O}(n)$ . This algorithm constructs two linear orderings vertex by vertex in a manner similar to a generalized *TopSort* or the *SelectionSort* algorithm for numbers: If a vertex has no incoming arcs, it is appended to the first ordering; if it has no outgoing arcs, it is prepended to the second ordering. Otherwise, the algorithm chooses the vertex whose difference of outgoing and incoming arcs is a minimum and appends it to the first ordering. In all cases, the processed vertex is removed from the graph and the procedure continues with the remainder. Finally, the second linear ordering is appended to the first.

Eades and Lin [EL95] later tailored the algorithm in [ELS93] specifically to cubic graphs, i. e., graphs where every vertex has degree 3, and were thereby able to improve



the bound in Equation (2.13) further. Their new algorithm runs in time  $\mathcal{O}(m \cdot n)$  and yields

$$\tilde{\tau}_{n,m} \leq \frac{m}{4}. \quad (2.16)$$

## 2.4 Linear Orderings in Practice

Until now, our perspective on the LINEAR ORDERING problem has been almost purely theoretical. We conclude this chapter by taking the practical view and mention algorithms that do not offer guarantees, but perform well in experiments. Furthermore, we also show in which application scenarios the LINEAR ORDERING problem occurs.

### 2.4.1 Heuristics

The motivation to design an algorithm for LINEAR ORDERING can be multifarious: Obtaining approximate solutions with guaranteed maximum deviation from the optimum, a worst-case upper bound, or being satisfied with a decisive answer of whether a concrete (small) quantity of backward arcs suffice, which would be an application for *FPT*-algorithms have already been covered earlier. For some practical scenarios, however, such guarantees are irrelevant. Instead, one may prefer algorithms that experimentally yield good results, are easy to implement, particularly fast, or a combination thereof.

The randomized algorithm by Berger and Shor [BS90] with a running time of  $\mathcal{O}(m)$  as well as the  $\mathcal{O}(m)$ -time, deterministic algorithm by Eades *et al.* [ELS93], which have already been mentioned in Section 2.3, are already two examples that meet all three requirements fairly well. The derandomized version of the former adds more complexity to the implementation and runs in time  $\mathcal{O}(m \cdot n)$ . Interestingly, both algorithms bear resemblance to a two-sided *SelectionSort*.

Another popular heuristic has been proposed by Chanas and Kobylański [CK96], which can be regarded as an adaptation of *InsertionSort* to the LINEAR ORDERING problem: It starts with an arbitrary linear ordering that specifies the processing order of the vertices and constructs a new linear ordering starting from an empty sequence. The currently processed vertex is inserted into the new ordering at its locally optimal position, i. e., such that a minimum of its incident arcs are backward. Once completed, the new linear ordering is used to specify the processing order and the procedure is repeated until the number of backward arcs is not reduced during a repetition. The



resulting linear ordering is then reversed and the algorithm starts anew, until also this step yields no further improvement. As the number of backward arcs can be reduced at most  $\mathcal{O}(m)$  times, the procedure can be implemented in time  $\mathcal{O}(m \cdot n^2)$ .

Coleman and Wirth [CW09] compared a number of algorithms, including those by Eades *et al.* as well as Chanas and Kobylański experimentally on tournaments of size 100. Their selection also contains adaptations of further sorting algorithms to the LINEAR ORDERING problem as well as several own variations. In conclusion, they find that the heuristic by Chanas and Kobylański performs best in practice, but also that it can be improved further by using other heuristics as preprocessing.

A similar study has been conducted on a broader set of input graphs, including also instances with up to 200 vertices as well as graphs that are significantly sparser than tournaments [Han10]. It comprises the algorithms mentioned above by Berger and Shor, Eades *et al.*, as well as Chanas and Kobylański. Additionally, it considers the algorithm by Demetrescu and Finocchi [DF03], which has already been mentioned in Section 2.2.2 and has a running time of  $\mathcal{O}(m \cdot n)$ , as well as an algorithm by Saab [Saa01], who developed a divide-and-conquer algorithm based on the minimum graph bisection problem. Similar as in the study by Coleman and Wirth, the algorithm by Chanas and Kobylański outperforms all others except in a benchmarking setup with cubic graphs, where the algorithms by Eades *et al.* and Saab achieve slightly better results.

Inspired by these findings, a study with different variations of *InsertionSort*-like heuristics on graphs of size up to 1,000 vertices has been carried out [BH11]. The authors also considered a 1-opt algorithm that repeatedly removes a vertex from the linear ordering and reinserts it at its locally optimal position. Additionally, its combination with a reversal operation as in Chanas and Kobylański's heuristic has been included. These new approaches turned out to produce the best results in the setup and also show the fastest convergence rates, i. e., the number of backward arcs reduces very quickly during the execution.

In their monograph on the LINEAR ORDERING problem on tournaments, Martí and Reinelt [MR11] survey further heuristics and report on their experimental performance. The authors make a distinction between constructive approaches, local search, and multi-start procedures and cover a great variety of algorithms. Their conclusion confirms the superiority of *InsertionSort* variants in the first category. Among the local search procedures, again insertion-based heuristics like that by Chanas and Kobylański and the 1-opt algorithm show to come in average closer to the optimum than their competitors. However, they reach the best objective value noticeably less often than, e. g., a heuristic

based on local enumeration, which computes an optimal solution for short, fixed-length subsequences of the linear ordering. Martí and Reinelt also dedicated an entire chapter to meta-heuristics like the greedy randomized adaptive search procedure (GRASP), tabu search, simulated annealing, and genetic algorithms. In the analysis, a memetic algorithm, which is a combination of a genetic algorithm and an insertion-based local search, outperforms all others.

For the FEEDBACK VERTEX SET problem, Lemaic [Lem08] proposed an interesting set of heuristics which is based on Markov chains.

## 2.4.2 Applications

The LINEAR ORDERING problem has a broad variety of applications, either directly or via one of its related problems. In the following, we list a small selection.

**Input-Output Analysis** In the introduction to this chapter, the origin of LO in the field of economics has already been mentioned. In particular, it arises in the analysis of so-called input-output matrices, which are used to represent the interdependencies, e. g., between different sectors of a national economy. Here, an  $n \times n$  matrix  $\mathbf{H}$  models the inputs and outputs of  $n$  sectors, where the entry  $h_{ij}$  in the  $i$ th row and the  $j$ th column tells the amount of units sector  $i$  requires from sector  $j$  in order to produce one unit itself. One approach to study these interdependencies consists in establishing a hierarchy of the sectors according to their subjection to the production of other sectors. Technically, this corresponds exactly to the perspective taken in the MATRIX TRIANGULATION problem. The linear ordering of the sectors can then be compared for instance to the respective result obtained for other countries [CW58]. Exact computations have been reported for a set of input-output matrices of size up to  $60 \times 60$  [GJR84b]. Input-output matrices also provide a good source for the generation of benchmarking instances for LINEAR ORDERING and FEEDBACK ARC SET algorithms and are part of the LOLIB library, which is also used in Chapter 7.

**Ranking and Rank Aggregation** Another vast area of applications originates from the close linkage between linear orderings and rankings, due to the fact that an arc in the input graph can be interpreted as a “preference” relation. Moreover, arc weights even allow to express nuances.

The WEIGHTED LINEAR ORDERING problem finds a relatively close correspondence in a variant of the RANK AGGREGATION problem which is also known as KEMENY RANKING [Kem59] and probably the second-oldest reference to LINEAR ORDERING:

Given a set of  $n$  items that are ranked by  $k$  voters with possible ties, find a consensus ranking that minimizes the sum of the distances to all  $k$  input rankings. The distance between two rankings is computed by considering the items pairwise and assigning each pair a value of 0, 1, or 2, depending on whether both rankings agree on their relative order, one sees them tied and the other has a preference, or their opinions are conflictive, respectively, and then taking the sum over all pairs. The task of finding a consensus ranking can be modeled as an instance of the LINEAR ORDERING problem: Construct a graph with a vertex for every item along with an arc  $(i, j)$  for every pair of items  $i, j$  whose weight equals the number of voters who prefer  $i$  to  $j$ . An optimal linear ordering of this weighted graph then yields a totally ordered consensus ranking whose sum of distances is a minimum. Due to the nature of the LINEAR ORDERING problem, however, consensus rankings with ties are ignored. If, however, the order of any two items is uniquely determined, the size of the corresponding feedback arc set equals the pairwise sum of the Kendall tau distances of the consensus ranking and all input rankings. The *Kendall tau distance* of two permutations is defined as the number of their pairwise disagreements and equals half of their Kemeny distance, where every disagreement scores 2.

Whereas a KEMENY RANKING asks to minimize the total number of disagreements over all voters, a SLATER RANKING aims at minimizing the number of disagreements over a pairwise majority decision, i. e., if a majority of the voters prefers  $i$  to  $j$ , then the consensus ranking should ideally rank  $i$  before  $j$ . This ranking rule originates from a problem studied by Slater [Sla61], who investigated inconsistencies in pairwise comparisons in psychological experiments. In order to find a consensus ranking, we can construct an unweighted graph having again a vertex for every item and an arc  $(i, j)$  for two items  $i, j$  if the majority of the voters ranked  $i$  before  $j$ . Every optimal linear ordering then is a consensus ranking.

Ranking and rank aggregation problems naturally occur in voting systems, sports competitions, and scheduling tasks and imply in turn vast and manifold applications of the LINEAR ORDERING problem. One of them is metasearch, which combines the rankings of different web search engines in order to either simply obtain a better result, taking multiple selection criteria into account, or to combat spam [DKNS01, YXS06]. Rank aggregation is also employed when users of a website can express their individual preferences and an overall opinion needs to be compiled, e. g., for movie databases or travel portals. In sports tournaments, Kemeny and Slater rankings or variations thereof

may be used to obtain a ranking of the candidates, however with the drawback that the optimal solution often is not unique.

Even though the LINEAR ORDERING problem is able to grasp both Kemeny and Slater rankings entirely if all rankings are permutations, research is dedicated specifically to these scenarios due to the special nature of the input graph: In contrast to the LINEAR ORDERING problem in general it is always built itself from rankings.

**Penalty Approaches** A solution to LO or FAS always produces a consistent set of arcs, which builds the acyclic subgraph. This characteristic is the key to its application in penalty approaches: Assume that we are given a set of items. Along with it comes a set of item pairs  $i, j$  specifying that  $i$  should precede  $j$  and a penalty  $p_{ij}$  that becomes due if this order is violated. We now need to find a set of item pairs such that the sum of its penalties is a minimum and all remaining pairs are not conflictive. In fact, this scenario can be modeled as an instance of WEIGHTED FEEDBACK ARC SET or WEIGHTED LINEAR ORDERING, if we are seeking a total order of the items. The input graph has a vertex for every item and an arc  $(i, j)$  whose weight equals the penalty  $p_{ij}$  for every given pair of items  $i, j$ .

The penalty approach is used in practice to solve the ONE-SIDED CROSSING MINIMIZATION problem: Consider a bipartite graph with vertex set  $U \dot{\cup} V$  and assume that the elements of  $U$  are totally ordered and placed on a straight line  $l_1$ . The task now is to find an ordering of the vertices in  $V$  such that if they are placed on a second straight line  $l_2$  parallel to  $l_1$ , the number of edge crossings is minimized. We can cast this as a WEIGHTED LINEAR ORDERING problem by creating a graph with vertex set  $V$  and an arc  $(u, v)$  for two vertices  $u, v \in V$  such that the penalty  $p_{uv}$  equals the number of crossings between arcs incident to  $u$  and arcs incident to  $v$  that occur if  $v$  precedes  $u$  in the resulting linear ordering. ONE-SIDED CROSSING MINIMIZATION appears, e. g. , as a subproblem in the graph drawing framework for directed graphs proposed by Sugiyama [STT81].

Further scenarios for the penalty approach are again scheduling problems as well as tasks that require to resolve cyclic dependencies and specify penalties for the non-observance of a dependency.

**FEEDBACK VERTEX SET: Deadlock Recovery and Circuit Testing** Due to the mutual reducibility of FEEDBACK ARC SET and FEEDBACK VERTEX SET, we also mention some applications for the latter.

A typical example is the recovery from a deadlock. Deadlocks arise in operating systems, e. g. , if a process waits for a resource that is currently held by another process, while the other process itself requests a resource that is held by the first process.

Similar situations may occur upon the concurrent access of a database and also in non-computational setups. These cases can be modeled again as a graph, where each process or client is represented by a (weighted) vertex and arcs are used to express dependencies. A minimum feedback vertex set then tells which processes or clients need to be killed in order to resolve the deadlock while keeping the damage to a minimum.

Another interesting application for FVS appears in the context of testing electronic circuits. Here, feedback cycles have shown to significantly complicate the task. However, the test generation complexity is greatly reduced if such cycles are broken while still yielding good fault coverage [CA90]. An exact algorithm for this problem has shown to work reasonably fast also for large instances [CA95].

**Further Applications** There are many more application scenarios for both FAS and FVS which are similar to those mentioned above. Furthermore, if we consider extensions of both problems such as additional constraints on the linear ordering, weights on both vertices and arcs, or a non-linear objective function, their number increases even further [MR11].

Table 2.1: Algorithms for the LINEAR ORDERING problem.

Algorithm	Graph Class	Performance <sup>a</sup>	Time Complexity	References
ILP/Branch & Cut/Cutting Plane	general	exact	exponential	cf. Section 2.2.1
Enumeration via Branch & Bound	general	exact	$\mathcal{O}(n!)$	cf. Section 2.2.1
Dynamic Programming	general	exact	$\mathcal{O}^*(2^n)^b$ , $\mathcal{O}^*(4^{n+o(n)})$	[RS07], [FK10]
Extension of $\mathcal{FPT}$ algorithm	tournaments	exact	$\mathcal{O}^*(1.5541^m)$	[RSS07]
Approximation	general	$\mathcal{O}(\log n \log \log n) \cdot \tau$	$\mathcal{P}$	[ENSS98]
Greedy Cycle Destruction	general	$ \text{longest cycle}  \cdot \tau$	$\mathcal{O}(m \cdot n)$	[DF03]
KwikSort	tournaments	$3 \cdot \tau$	$\mathcal{P}$	[ACN08, vZW09]
PTAS	tournaments	$(1 + \varepsilon) \cdot \tau$	$\mathcal{P}$	[KS06, KS07]
Approximation ( $\mathcal{APX}$ )	bipartite tournaments	$4 \cdot \tau$	$\mathcal{P}$	[vZ11]
$\mathcal{FPT}$ algorithm	general	exact	$\mathcal{O}^*(4^k k^3 k!)$	[CLL+08]
$\mathcal{FPT}$ algorithm	tournaments	exact	$\mathcal{O}(2^{\mathcal{O}(\sqrt{k} \log^2 k)})$	[KS10]
$\mathcal{FPT}$ algorithm	bipartite tournaments	exact	$\mathcal{O}^*(3.373^k)$	[DGH+10]
via Minimum Dijoins	planar graphs	exact	$\mathcal{O}(n^4)$	[Luc76, LY78, Fra81]
Contraction algorithm	reducible flow graphs	exact	$\mathcal{O}(m \log n)$	[Ram88, Koe05]
via LP Solver	graphs with $\tau = \nu$	exact	$\mathcal{P}$	[GJR85b, BFM94, NP95]
Selection by Degree	general	$\frac{m}{2} - \frac{m \cdot c}{\sqrt{\Delta_G}}$	$\mathcal{O}(m \cdot n)$	[BS90]
Generalized TopSort	general / cubic graphs	$\frac{m}{2} - \frac{n}{6} / \frac{m}{4}$	$\mathcal{O}(m)$ / $\mathcal{O}(m \cdot n)$	[ELS93] / [EL95]
InsertionSort-like Heuristics	general	–	$\mathcal{O}(m \cdot n^2)$	[CK96, BH11]

<sup>a</sup>Worst-case upper bound for result of computation.  $\tau$ : Value of optimal solution.<sup>b</sup>Space complexity:  $\mathcal{O}^*(2^n)$

# 3

## Preliminaries: Definitions and Preparations

The first two sections of this chapter summarize mathematical notations and basic definitions that are used in this thesis and introduce some technical terms. In the third section, we look at some simple graph manipulations and briefly study their impact on the LINEAR ORDERING problem. Among other things, these contribute to being able to state a couple of default assumptions in the last section.

### 3.1 Basics

We start by introducing a set of very basic terms and definitions. Although most readers will be familiar with them, we list them here for the sake of completeness.

#### 3.1.1 Sets and Multisets

A *set* is an unordered collection of pairwise distinct elements, e. g. ,  $\{e_0, e_1, e_2\}$ . A set  $Y$  is a *subset* of another set  $X$ , denoted by  $Y \subseteq X$ , if every element of  $Y$  is also contained in  $X$ , but not necessarily vice versa. A subset  $Y$  of  $X$  is called *proper* and denoted by either  $Y \subsetneq X$  or  $Y \subset X$ , if  $Y \subseteq X$ , but  $Y \neq X$ . In this thesis, the former notation,  $Y \subsetneq X$  is used to emphasize the properness of a subset relation, e. g. , if this fact is important, while in the latter,  $Y \subset X$ , it can be regarded as an additional, but not relevant information. If  $Y \subseteq X$ , then  $X$  conversely is a *superset* of  $Y$ , which is denoted as  $X \supseteq Y$ . Analogously, a superset is said to be *proper*, denoted as  $Y \supsetneq X$  and unstressed as  $Y \supset X$ , if equality of the sets is ruled out. The number of elements of a set  $X$  is called the *cardinality* of  $X$  and denoted by  $|X|$ . The cardinality of a set can be finite or infinite. To distinguish sets from multisets, which will be introduced later in this subsection, we also call them *simple* sets.

The set of natural numbers is denoted by  $\mathbb{N}$ , which by default includes the element 0. The fact that  $\mathbb{N}$  contains 0 can be emphasized by writing  $\mathbb{N}_0$ . To specify the natural numbers that are greater than or at least a specific value  $k \in \mathbb{N}$ , we use the abbreviations

$\mathbb{N}_{>k}$  and  $\mathbb{N}_{\geq k}$ , respectively. These definitions can be made analogously for the set  $\mathbb{R}$  of real numbers.

In contrast to a set, a *sequence*  $S$  is an ordered collection of elements and allows for duplicates. A sequence can be specified by  $(s_i)_{i=0}^k$  together with a definition of each element  $s_i$ . The number of elements of  $S$  is called the *length* of  $S$ , which may be finite or infinite.

An *n-tuple* is an ordered collection of exactly  $n$  elements, which are usually listed within parenthesis.  $(e_1, e_2, e_3)$ , for instance, is a 3-tuple. Like sequences, *n-tuples* may contain duplicates.

If a set's restriction of being distinct is put aside, we obtain a *multiset*. To distinguish multisets from sets, we denote them by  $[e_0, e_0, e_1, e_2]$ . Formally, a multiset  $X$  is defined by a 2-tuple  $(U, m)$ , where  $U$  is called the underlying set of elements and  $m : U \rightarrow \mathbb{N}_{\geq 1}$  specifies the number of occurrences of an element  $u \in U$  in  $X$ . If  $X = (U, m)$  and  $U \subseteq U'$ , we introduce a term and shorthand notation for this relation by calling  $X$  a *multisubset* of  $U'$  and denoting it by  $X \subseteq^* U'$ . For  $U' = \{e_0, e_1, e_2\}$ , e. g.,  $X = [e_0, e_0, e_1]$  is a multisubset of  $U'$ , as the underlying set of elements of  $X$  is  $U = \{e_0, e_1\}$  and  $U \subseteq U'$ . Note that every simple set  $X$  is also a multiset  $(X, m)$ , where  $m : X \rightarrow \{1\}$  is the constant-one function that maps every element to 1. A multiset  $Y = (U_Y, m_Y)$  is a subset of a multiset  $X = (U_X, m_X)$ , denoted by  $X \subseteq Y$ , if  $U_Y \subseteq U_X$  and  $\forall u \in U_Y : m_Y(u) \leq m_X(u)$ . As in the case of subsets, most relations defined on sets carry over accordingly to multisets. To avoid confusion, however, we briefly address unions and multiset sums. Given two multisets  $X = (U_X, m_X)$  and  $Y = (U_Y, m_Y)$ , the union  $X \cup Y$  is defined as the multiset  $(U_{X \cup Y}, m_{X \cup Y})$ , where  $U_{X \cup Y} = U_X \cup U_Y$  and for all  $u \in U_{X \cup Y}$ ,

$$m_{X \cup Y}(u) = \begin{cases} m_X(u), & \text{if } u \in U_X \wedge u \notin U_Y, \\ m_Y(u), & \text{if } u \in U_Y \wedge u \notin U_X, \\ \max \{m_X(u), m_Y(u)\}, & \text{else.} \end{cases}$$

In contrast, the *multiset sum*  $X \uplus Y$  yields the multiset  $(U_{X \uplus Y}, m_{X \uplus Y})$ , where  $U_{X \uplus Y} = U_X \cup U_Y$  and for all  $u \in U_{X \uplus Y}$ ,

$$m_{X \uplus Y}(u) = \begin{cases} m_X(u), & \text{if } u \in U_X \wedge u \notin U_Y, \\ m_Y(u), & \text{if } u \in U_Y \wedge u \notin U_X, \\ m_X(u) + m_Y(u), & \text{else.} \end{cases}$$

Observe that a multiset sum only differs from an ordinary union if  $X$  and  $Y$  are not disjoint. With these definitions, we obtain for  $X = [e_0, e_0, e_1]$  and  $Y = [e_0, e_1, e_1, e_2]$ ,



e. g.,  $X \cup Y = [e_0, e_0, e_1, e_1, e_2]$  and  $X \uplus Y = [e_0, e_0, e_0, e_1, e_1, e_1, e_2]$ . As might have been expected, the intersection  $X \cap Y$  is defined as the multiset  $(U_{X \cap Y}, m_{X \cap Y})$ , where  $U_{X \cap Y} = U_X \cap U_Y$  and for all  $u \in U_{X \cap Y}$ ,

$$m_{X \cap Y}(u) = \min \{m_X(u), m_Y(u)\}.$$

The *cardinality* of a multiset  $X = (U, m)$  equals again the number of elements contained in  $X$  and is denoted by  $|X|$ , i. e.,  $|X| = \sum_{u \in U} m(u)$ .

To simplify notation, we write  $x \in X$  for a multiset  $X = (U, m)$ , to express that  $x \in U$ , and  $x \notin X$  if  $x \notin U$ . Furthermore, in case that no ambiguity arises, we use the term set from here on as an umbrella term for both simple sets and multisets.

### 3.1.2 Minimal and Maximal versus Minimum and Maximum

As the terms minimal and maximal and their differentiation from minimum and maximum often lead to misunderstandings, we define them here explicitly. Let  $X$  be a set of elements,  $Y \subseteq X$  be a subset thereof and  $p$  be an arbitrary property such that  $Y$  fulfills  $p$ . Under these preconditions, we say that  $Y$  is *minimal* with respect to  $p$ , if for every element  $y \in Y$  holds that the set  $Y \setminus \{y\}$  does not fulfill  $p$ . Likewise,  $Y$  is *maximal* with respect to  $p$ , if for every element  $x \in X \setminus Y$  holds that  $Y \cup \{x\}$  does not fulfill  $p$ .

On the contrary,  $Y$  is *minimum* with respect to  $p$ , if there is no other subset  $Z \subseteq X$  such that  $Z$  also fulfills  $p$  and  $|Z| < |Y|$ . Analogously,  $Y$  is *maximum* with respect to  $p$ , if there is no other subset  $Z \subseteq X$  such that  $Z$  also fulfills  $p$  and  $|Z| > |Y|$ .

### 3.1.3 Computational Complexity

We use standard big O notation, i. e.,  $\mathcal{O}$ ,  $o$ ,  $\Omega$ , and  $\Theta$ , to describe the growth of functions that express the time or space complexity of an algorithm or a subroutine. When dealing with very fast growing functions such as exponential functions, polynomially bounded factors are often negligible. The starred variant of  $\mathcal{O}$  accounts for these situations and is defined as  $\mathcal{O}^*(g) = \mathcal{O}(g \cdot p)$ , where  $p$  is any polynomial. This notation is often used for exact or *FPT* algorithms.

### 3.1.4 Data Structures

Many algorithms that are described in this thesis use lists to store different types of elements. We therefore briefly introduce some notations in this context.

Lists represent the mathematical concept of finite sequences, i. e. , they maintain the order of their elements and allow for duplicates. A list  $L$  with elements  $e_0, e_1,$  and  $e_2$  can be specified by  $\langle e_0, e_1, e_2 \rangle$ . Subsequently,  $\langle \rangle$  denotes the empty list.

We use the operator  $\diamond$  to express that two lists are concatenated. For two lists  $L_1 = \langle e_0, e_1 \rangle$  and  $L_2 = \langle e_2 \rangle$ , e. g. ,  $L_1 \diamond L_2 = \langle e_0, e_1, e_2 \rangle$  and  $L_2 \diamond L_1 = \langle e_2, e_0, e_1 \rangle$ .

The  $i$ th element of a list  $L$  can be queried by  $L[i]$ . We assume that lists are zero-based, i. e. , the first element of the list has index 0. For  $L = \langle e_0, e_1, e_2 \rangle$ , e. g. , this implies that  $L[0] = e_0, L[1] = e_1,$  and  $L[2] = e_2$ , while for another list  $L' = \langle e_2, e_0, e_1 \rangle$ , e. g. ,  $L'[0] = e_2, L'[1] = e_0,$  and  $L'[2] = e_1$ .

The *length* of a list is denoted by  $|L|$  and indicates the number of elements it contains. Thus, for  $L = \langle e_0, e_1, e_2 \rangle$ ,  $|L| = 3$ .

## 3.2 General Graph Theory

We now turn to graphs. The definitions introduced here refer without exception to *directed graphs (digraphs)*. In contrast to *undirected graphs*, the relation between two vertices is not necessarily symmetric.

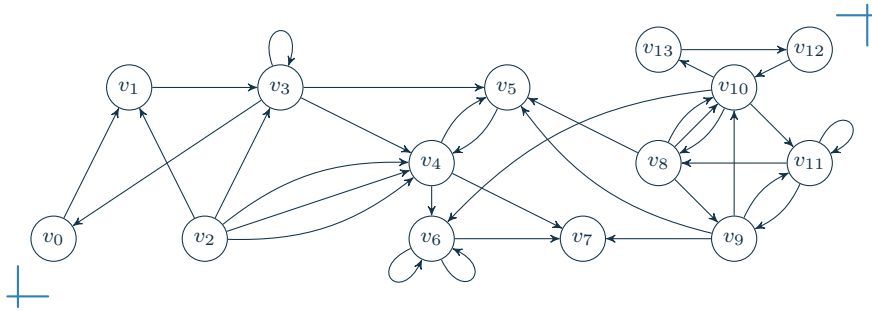
### 3.2.1 Graphs, Vertices, and Arcs

A *graph*  $G$  is a tuple  $(V, A)$  with a finite *vertex set*  $V$  and a finite *arc set*  $A \subseteq^* V \times V$ , where  $A$  is a multiset  $(U, m)$  with  $U \subseteq V \times V$ . Unless indicated otherwise, we assume this naming for the vertex and arc set of a graph  $G$  in the remainder of this thesis.

The size of  $G$  is measured by the cardinalities  $n = |V|$  and  $m = |A|$ . An arc  $a$  with *tail*  $u$  and *head*  $v$  is specified by  $a = (u, v)$ . In this case,  $a$  is said to be *incident* to both vertices  $u$  and  $v$  and  $a$  is an *outgoing* arc of  $u$  and an *incoming* arc of  $v$ . Two vertices  $u, v$  are called *adjacent*, if  $(u, v) \in A$  or  $(v, u) \in A$ . For an arc  $a = (u, v)$ , we say that  $a$  is an incoming arc from  $u$  at  $v$  and an outgoing arc to  $v$  at  $u$ .

The *reverse* of an arc  $(u, v)$ , denoted by  $(u, v)^R$ , is an arc with transposed head and tail, i. e. ,  $(u, v)^R = (v, u)$ . For a (multi-)set  $Y \subseteq A$  of arcs,  $Y^R = [y^R \mid y \in Y]$  and consequently,  $|Y| = |Y^R|$ . The *reverse* of a graph  $G = (V, A)$  is the graph  $G^R = (V, A^R)$ . Observe that reversion is an involution, i. e. ,  $(u, v)^{RR} = (u, v)$ ,  $Y^{RR} = Y$ , and  $G^{RR} = G$ .

A *loop* is an arc where head and tail coincide, e. g. ,  $a = (u, u)$ . An arc  $a = (u, v)$  has *parallel arcs*, if  $m(a) \geq 2$  and one or more *anti-parallel arcs*, if  $(v, u) \in A$ . The multiset containing all parallel arcs of an arc  $a$ , including  $a$  itself, is denoted by  $[a]_{\parallel}$ . We say that



**Figure 3.1:** A multigraph with loops at  $v_3$ ,  $v_6$ , and  $v_{11}$ , three parallel arcs  $(v_2, v_4)$ , two parallel arcs  $(v_8, v_{10})$ , as well as anti-parallel arcs  $\{(v_4, v_5), (v_5, v_4)\}$ ,  $\{(v_8, v_{10}), (v_{10}, v_8)\}$ , and  $\{(v_9, v_{11}), (v_{11}, v_9)\}$ .

a graph has parallel or anti-parallel arcs, if one of its arcs has parallel or anti-parallel arcs, respectively. A graph with loops, parallel, or anti-parallel arcs is called a *multigraph*. Otherwise, the graph is said to be *simple*. In the latter case, it suffices to define  $A$  as a simple set, i. e.,  $A \subseteq V \times V$ . **Figure 3.1** provides an example of a multigraph that will serve as a running example for the remainder of this chapter.

A graph  $G' = (V', A')$  is a *subgraph* of  $G$ , denoted by  $G' \subseteq G$ , if  $V' \subseteq V$  and  $A' \subseteq A$  as well as  $A' \subseteq^* V' \times V'$ . A subgraph  $G'$  is *spanning* if  $V' = V$ . For a set of vertices  $X \subseteq V$ , the *induced* subgraph  $G_X$  is the subgraph of  $G$  with vertex set  $X$  and arc set  $(U_X, m_X)$ , where  $U_X = \{(u, v) \in U \mid u \in X \wedge v \in X\}$  and  $\forall a \in U_X : m_X(a) = m(a)$ . Using a set of arcs  $Y$ , we define the spanning subgraph  $G|_Y$  of  $G$  that is *restricted* to  $Y$  as  $G|_Y = (V, A \cap Y)$ .

A *tournament* is a simple graph  $G = (V, A)$  such that for each distinct pair of vertices  $u, v \in V$ ,  $u \neq v$ , exactly one of the arcs  $(u, v)$  and  $(v, u)$  is in  $A$ . A graph is *bipartite* if its vertex set  $V$  can be partitioned into two sets  $V' \dot{\cup} V'' = V$  such that for all arcs  $(u, v) \in A$ , either  $u \in V'$  and  $v \in V''$  or  $u \in V''$  and  $v \in V'$ . A *bipartite tournament* is a simple bipartite graph  $G = (V' \dot{\cup} V'', A)$  with exactly one of  $(u, v)$  and  $(v, u)$  in  $A$  for each pair of vertices  $u, v$  such that  $u \in V'$  and  $v \in V''$ .

### 3.2.2 Vertex Degrees

For a vertex  $v \in V$ , we denote by  $N^+(v)$  the simple set of vertices that  $v$  has an outgoing arc to, i. e.,  $N^+(v) = \{u \in V \mid (v, u) \in U\}$ . Likewise,  $N^-(v)$  denotes the simple set of vertices that  $v$  has an incoming arc from, i. e.,  $N^-(v) = \{u \in V \mid (u, v) \in U\}$ . The *outdegree*  $d^+(v)$  of  $v$  is defined as the number of arcs that are outgoing from  $v$ , also

counting parallel arcs, i. e.,  $d^+(v) = \sum_{u \in N^+(v)} m((v, u))$ . Equivalently,  $d^-(v)$  denotes  $v$ 's *indegree*, i. e.,  $d^-(v) = \sum_{u \in N^-(v)} m((u, v))$ . Finally, the (total) *degree*  $d(v)$  of  $v$  is defined as  $d(v) = d^+(v) + d^-(v)$  and the *delta degree*  $\delta(v)$  as  $\delta(v) = d^+(v) - d^-(v)$ . Note that for every graph,  $\sum_{v \in V} \delta(v) = \sum_{v \in V} d^+(v) - \sum_{v \in V} d^-(v) = 0$ .

A vertex  $v$  is called a *source* if  $d^+(v) \geq 1$  and  $d^-(v) = 0$  and a *sink* if  $d^+(v) = 0$  and  $d^-(v) \geq 1$ . In case  $d(v) = 0$ ,  $v$  is said to be *isolated*. The maximum degree of a vertex in a graph is denoted by  $\Delta_G$ , i. e.,  $\Delta_G = \max_{v \in V} d(v)$ .

A graph  $G$  is *k-regular*, if  $\forall v \in V : d(v) = k$  and *k-subregular*, if  $\forall v \in V : d(v) \leq k$ , or equivalently,  $\Delta_G = k$ . For  $k = 3$ , we also refer to such a graph as *cubic* or *subcubic*, respectively. For  $k = 4$ , the terms *quartic* and *subquartic* are used as an alternative. In case of  $k = 5$ , a graph is called *quintic* or *subquintic*, respectively.

### 3.2.3 Paths, Cycles, and Walks

A *path*  $P$  of length  $k$  is an alternating sequence of  $k + 1$  vertices and  $k$  connecting arcs  $P = \langle v_0, (v_0, v_1), v_1, (v_1, v_2), v_2, \dots, v_{k-1}, (v_{k-1}, v_k), v_k \rangle$ . In case no ambiguity arises, we may also specify a path by giving only either its vertices or arcs. As a short form, we write  $v_0 \rightsquigarrow v_k$  to denote an unspecified path from  $v_0$  to  $v_k$ . A path is *simple*, if it does not include a vertex more than once. Let  $V_P = \{v_0, v_1, \dots, v_k\}$  be the set of vertices of  $P$  and  $A_P = \{(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)\}$  the set of arcs of  $P$ . With a slight abuse of notation, we write  $v \in P$  for a vertex  $v \in V$  to express that  $v \in V_P$ , and  $a \in P$  for an arc  $a \in A$  to express that  $a \in A_P$ .

A path  $P'$  is a *subpath* of a path  $P = \langle v_0, (v_0, v_1), v_1, (v_1, v_2), v_2, \dots, v_{k-1}, (v_{k-1}, v_k), v_k \rangle$ , if  $P' = \langle v_i, (v_i, v_{i+1}), v_{i+1}, \dots, v_{j-1}, (v_{j-1}, v_j), v_j \rangle$ , for some  $i, j$  with  $0 \leq i \leq j \leq k$ , i. e.,  $P'$  is a continuous subsequence of vertices and arcs of  $P$ .

Two paths  $P$  and  $P'$  are *vertex-disjoint* if they do not share a common vertex. Likewise,  $P$  and  $P'$  are *arc-disjoint*, if they do not share a common arc. Every vertex-disjoint pair of paths is also arc-disjoint.

A *cycle*  $C$  of length  $k$  is a path of length  $k$  where the first and the last vertex coincide, i. e.,  $v_0 = v_k$ . For a cycle to be *simple*, all vertices apart from  $v_0, v_k$  must be distinct. Whenever a graph contains a cycle  $C$  that is not simple, it also contains a simple cycle  $C'$  that consists of the shortest subpath in  $C$  that begins and ends at the same vertex.

The notion of vertex- and arc-disjointness is transferred from paths to cycles analogously, i. e., two cycles  $C$  and  $C'$  are *vertex-disjoint* (*arc-disjoint*) if no vertex (arc) is contained in both  $C$  and  $C'$ .

Note that paths and cycles of length  $k = 1$  are single arcs and loops, respectively.

Similar to a path, a *walk*  $W$  of length  $k$  is an alternating sequence of  $k + 1$  vertices and  $k$  arcs  $W = \langle v_0, a_{\{v_0, v_1\}}, v_1, a_{\{v_1, v_2\}}, v_2, \dots, v_{k-1}, a_{\{v_{k-1}, v_k\}}, v_k \rangle$  with the vital difference that  $a_{\{v_i, v_j\}}$  may either be the arc  $(v_i, v_j)$  or its reverse, i. e.,  $(v_j, v_i)$ . Informally, a walk is a path where the arcs' directions are ignored. Consequently, every path is also a walk, but not vice versa.

### 3.2.4 Connectivity and Acyclicity

A graph  $G$  is said to be *connected*, if there is a walk from  $u$  to  $v$  for every pair of vertices  $u, v \in V$ .  $G$  is said to be  $k$ -vertex-connected or simply  $k$ -connected, if  $n > k$  and for any subset of at most  $k - 1$  vertices  $V' \subsetneq V$ ,  $G_{V \setminus V'}$  is connected, i. e., the removal of at most  $k - 1$  arbitrary vertices does not disconnect  $G$ . A 2-connected graph is synonymously also called *biconnected*. A *connected component* of a graph is a maximal connected subgraph, i. e., for  $X \subseteq V$ ,  $G_X$  is a connected component if  $G_X$  is connected if there is no superset  $X' \supsetneq X$  such that  $G_{X'}$  is connected. Analogously, a *biconnected component* or *block* of a graph is a maximal biconnected subgraph. A vertex  $v$  is called a *cut vertex*, if  $G_{V \setminus \{v\}}$  has more connected components than  $G$ . In particular, a graph is biconnected if and only if it does not have a cut vertex. The definition implies that every cut vertex of a graph is part of at least two blocks. The *block-cut tree* of a graph  $G$  is an unrooted tree  $\mathcal{T}$  whose nodes each represent either a block or a cut vertex of  $G$ . Two nodes of  $\mathcal{T}$  are adjacent if and only if one of them represents a block, the other a cut vertex, and the block contains the cut vertex.

The graph depicted in [Figure 3.1](#), e. g., is not biconnected due to the presence of the (only) cut vertex  $v_{10}$ . It contains two blocks: one consisting of all vertices  $v_i$  with  $0 \leq i \leq 10$ , and second consisting of all vertices  $v_j$  with  $10 \leq j \leq 13$ . Observe that  $v_{10}$  is contained in both blocks. The block-cut tree of this example graph thus consists of three nodes representing two blocks and one cut vertex.

For two vertices  $u, v$  of a graph  $G$ ,  $v$  is said to be *reachable* from  $u$  if there is a path  $u \rightsquigarrow v$  in  $G$ . The vertices  $u$  and  $v$  are called *strongly connected*, if they are mutually reachable, i. e., there are paths  $u \rightsquigarrow v$  and  $v \rightsquigarrow u$  in  $G$ . By transferring this definition to the whole graph, we obtain that a graph is *strongly connected*, if there is a path between each ordered pair of vertices.

A *strongly connected component*, SCC for short, of a graph is a strongly connected subgraph that is maximal with respect to its vertex set. Formally, the property of being

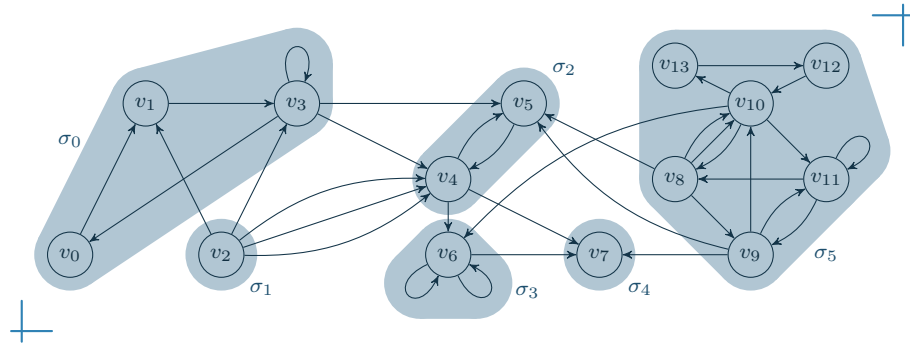


Figure 3.2: Strongly connected components of the example graph.

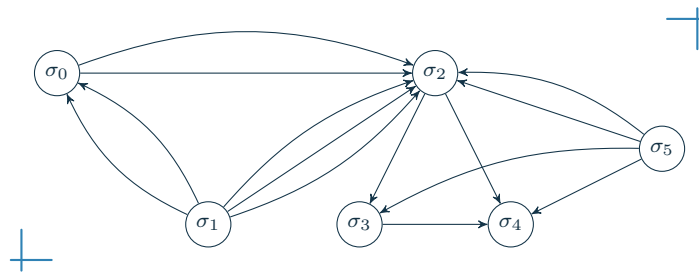


Figure 3.3: The condensation of the example graph.

strongly connected defines an equivalence relation on the set of vertices such that each equivalence class corresponds to the vertex set of exactly one SCC. This also implies that any two vertices  $v_i, v_j$  of a cycle  $C$  are strongly connected, as  $C$  can be split into two paths  $v_i \rightsquigarrow v_j$  and  $v_j \rightsquigarrow v_i$ . Therefore, for every cycle in  $G$ , its vertices and arcs must be part of the same SCC. Figure 3.2 shows the SCCs of the example graph.

In general, the number of SCCs in a graph may range between 1 and  $n$ . In the former case, the graph itself is strongly connected, while in the latter, every SCC consists of a single vertex only. If so, the graph does not contain a cycle of length  $k > 1$ . A graph is said to be *acyclic*, if it does not contain a cycle of length  $k > 0$ . In consequence, every loop-free graph with  $n$  SCCs is acyclic. Contracting each SCC of  $G$  to a single vertex always yields an acyclic graph that is called the *condensation* of  $G$ . See Figure 3.3 for the condensation of the example graph.

In an acyclic graph, the reachability relation defines a partial order on the set of its vertices. Every linear extension of this relation (i. e., to a total order) corresponds to a topological sorting of the acyclic graph. A *topological sorting* of  $G = (V, A)$  is a bijective mapping  $\xi : V \rightarrow \{0, \dots, n - 1\}$  that assigns a *topsort number*  $\xi(v)$  to every vertex  $v$  such

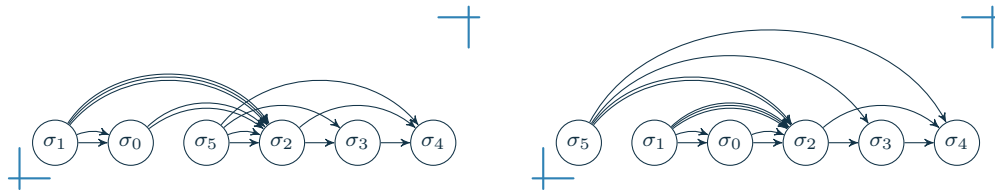


Figure 3.4: Two possible topological sortings of the condensed example graph.

that  $\forall (u, v) \in A : \xi(u) < \xi(v)$ . Descriptively, every arc must point from a vertex with a lower topsort number to a vertex with a higher topsort number. A graph is acyclic if and only if it can be sorted topologically. Just as there may be a number of different linear extensions to a partial order, there may be equally many topological sortings of an acyclic graph. For illustration, Figure 3.4 shows two possible topological sortings of the condensed example graph.

Computing a topological sorting and inherently checking whether a graph is acyclic can be accomplished in time  $\mathcal{O}(n + m)$  by repeatedly removing a source  $v$  from the graph and assigning it the topsort number  $\xi(v)$  in increasing order. We refer to the algorithm that returns a topological sorting of an acyclic graph  $G$  as  $TopSort(G)$ .

The strongly connected components of a graph can also be retrieved in time  $\mathcal{O}(n + m)$  using, e. g., Tarjan's algorithm [Tar72], which is based on depth-first search. At the same time, the graph's condensation may be constructed.

### 3.3 Feedback Arc Sets and Linear Orderings

This section is to introduce notation that relates especially to the mathematical handling of feedback arc sets and linear orderings.

#### 3.3.1 Feedback Arc Sets

A *feedback arc set*  $\mathcal{B}$ , sometimes also called a *cycle cover*, for a graph  $G = (V, A)$  is a set of arcs  $\mathcal{B} \subseteq A$  such that the subgraph restricted to  $A \setminus \mathcal{B}$  is acyclic. The set  $\mathcal{F} = A \setminus \mathcal{B}$  is referred to as the *acyclic arc set* of  $G$  with respect to  $\mathcal{B}$  and the subgraph restricted to  $\mathcal{F}$ ,  $G|_{\mathcal{F}}$ , as the according *acyclic subgraph*.

Given a set of arcs  $\mathcal{B} \subseteq A$ , we say that  $\mathcal{B}$  is *feasible*, if it forms a valid feedback arc set. In conformance with the definitions in Section 3.1, a feedback arc set is called *minimal*, if no arc can be removed from it such that it remains feasible. Analogously, a feedback arc

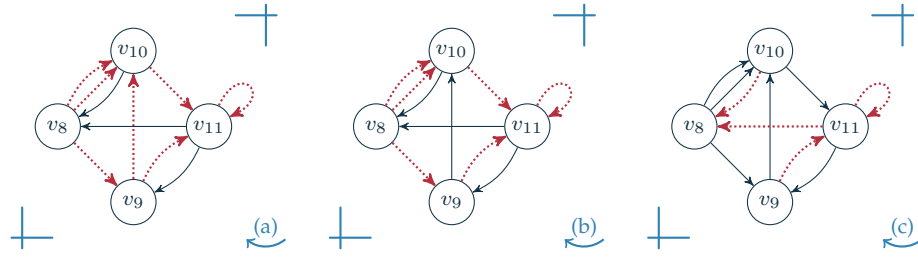


Figure 3.5: A feasible feedback arc set (a), a minimal feedback arc set (b), and an optimal (minimum) feedback arc set (c) for the subgraph  $\sigma_5$ , highlighted by dotted red arcs.

set is *minimum*, if its cardinality is minimum among all feedback arc sets of  $G$ . In this case, the feedback arc set is also said to be *optimal*.

Figure 3.5 shows three different feedback arc sets for the strongly connected subgraph  $\sigma_5$  of the example graph. In the first example, Figure 3.5(a), one of the arcs  $(v_8, v_9)$ ,  $(v_9, v_{10})$ , or  $(v_{10}, v_{11})$  could be removed from the feedback arc set while still maintaining feasibility. This is not the case for the feedback arc set depicted in Figure 3.5(b), however, its cardinality is not minimum among all feedback arc sets, as Figure 3.5(c) testifies. In fact, the last figure shows an optimal feedback arc set.

We denote the cardinality of an optimal feedback arc set by  $\tau_G$  or simply  $\tau$ , if no ambiguity arises.

If  $\mathcal{B}$  is an (optimal) feedback arc set for a graph  $G$ , then  $\mathcal{B}^R$  is an (optimal) feedback arc set for the reverse graph  $G^R$ . In particular, this implies that  $\tau_G = \tau_{G^R}$ .

The optimal feedback arc set is the empty set if and only if the graph is acyclic.

### 3.3.2 Linear Orderings

A *linear ordering*  $\pi_G$  of  $G = (V, A)$  is a bijective mapping  $\pi_G : V \rightarrow \{0, \dots, n-1\}$  that assigns a unique *LO position*  $\pi_G(v)$  to every vertex  $v \in V$ . An arc  $(u, v)$  is called *forward*, if  $\pi_G(u) < \pi_G(v)$ , and otherwise *backward*. Two vertices  $u, v \in V$  are said to be *consecutive* with respect to  $\pi_G$ , if their LO positions differ by exactly one, i. e.,  $|\pi_G(u) - \pi_G(v)| = 1$ . More generally, a set  $X \subseteq V$  of vertices is called consecutive with respect to  $\pi_G$ , if  $\max_{u, v \in X} |\pi_G(u) - \pi_G(v)| = |X| - 1$ .

Linear orderings are usually visualized by aligning the vertices horizontally and sorting them according to their LO positions in increasing order from left to right. Forward arcs are then drawn as arches above or on the same level as the vertices, while backward arcs appear below it. We denote by  $\mathcal{F}_{\pi_G}$  and  $\mathcal{B}_{\pi_G}$  the set of arcs that are



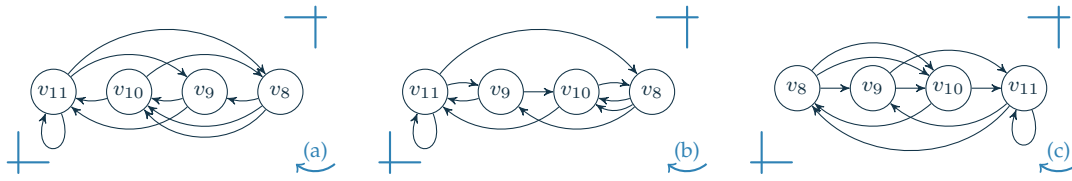


Figure 3.6: Linear orderings of the subgraph  $\sigma_5$ , such that the backward arcs in (a), (b), and (c) each correspond to the respective feedback arc set shown in Figure 3.5.

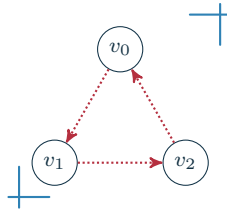


Figure 3.7: A feasible feedback arc set without corresponding linear ordering.

classified by  $\pi_G$  as forward and backward, respectively. We omit the subscript and simply write  $\pi$  instead of  $\pi_G$  if the graph it refers to is clear from the context.

In fact, linear orderings bear resemblance to topological sortings, but with the important distinction that the latter do not allow for backward arcs. If we consider the subgraph restricted to the forward arcs, however, then  $\pi$  indeed is a topological sorting of this subgraph. Consequently,  $\pi$  defines an acyclic subgraph  $G|_{\mathcal{F}_\pi}$  of  $G$  along with a feedback arc set, which corresponds exactly to the set of arcs classified as backward,  $\mathcal{B}_\pi$ . Note that if we consider instead the subgraph restricted to the backward arcs excluding loops, then the reversal of  $\pi$  again is a topological sorting. Hence,  $G|_{\mathcal{B}_\pi}$  may contain cycles of length at most one. We say that a vertex  $v \in V$  is a *pseudosource* with respect to a linear ordering  $\pi$  if  $v$  is a source in  $G|_{\mathcal{F}_\pi}$ . Analogously, a vertex  $v \in V$  is called a *pseudosink* with respect to a linear ordering  $\pi$  if  $v$  is a sink in  $G|_{\mathcal{F}_\pi}$ .

Figure 3.6 uses the subgraph  $\sigma_5$  of the example graph again and shows three linear orderings. The set of backward arcs in each of them corresponds to the feedback arc set depicted in the respective subfigure of Figure 3.5.

A linear ordering uniquely identifies a feedback arc set  $\mathcal{B}_\pi$  and an acyclic arc set  $\mathcal{F}_\pi$ , whereas the opposite does not hold true in general. On the contrary, every topological sorting of  $G|_{\mathcal{F}_\pi}$  yields a linear ordering  $\pi'$  such that  $\mathcal{B}_{\pi'} \subseteq \mathcal{B}_\pi$ , i. e., the topological sorting may “accidentally” have classified arcs of  $\mathcal{B}_\pi$  as forward. This implies that, first, there may be multiple linear orderings that define the same feedback arc set and, second, not every feedback arc set  $\mathcal{B}$  has a corresponding linear ordering  $\pi$  such that  $\mathcal{B} = \mathcal{B}_\pi$ . For a

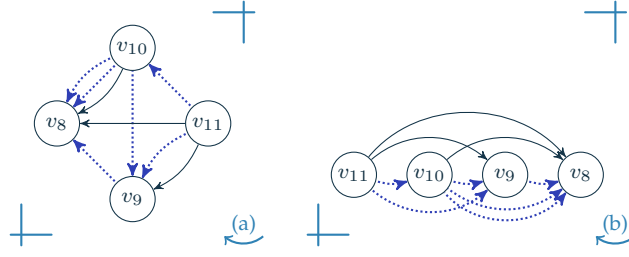


Figure 3.8: Construction of linear ordering that corresponds exactly to the feedback arc set depicted in Figure 3.5(a): reversal of non-loop backward arcs (a) and topological sorting (b). For illustration, reversed backward arcs are drawn below the vertex level.

graph consisting only of a cycle of length  $k > 1$ , for example, such as the graph shown in Figure 3.7, the set of all arcs is feasible, but we have already argued in the paragraph before last that for any linear ordering  $\pi$ , the subgraph  $G|_{\mathcal{B}_\pi}$  must be acyclic except for loops. If this requirement is met, however, a construction of  $\pi$  is possible, which is also exemplified in Figure 3.8.

---

#### Observation

If there is an acyclic arc set  $\mathcal{F} \subseteq A$  of a graph  $G = (V, A)$  and the set  $\mathcal{B} = A \setminus \mathcal{F}$  does not contain a cycle of length  $k \geq 2$ , a corresponding linear ordering  $\pi$  can be obtained such that  $\mathcal{B} = \mathcal{B}_\pi$  by topologically sorting the graph that consists of  $G|_{\mathcal{F}}$  plus the reverse of each arc in  $\mathcal{B}$  that is not a loop.

---

The characteristic of inducing the same set of backward arcs yields an equivalence relation  $\sim_{\mathcal{B}}$  on the set of linear orderings of a graph. The equivalence class of  $\pi$  hence is the set containing all linear orderings  $\pi'$  that induce exactly the same set of backward arcs as  $\pi$ , i. e.,  $[\pi]_{\sim_{\mathcal{B}}} = \{\pi' \mid \mathcal{B}_{\pi'} = \mathcal{B}_\pi\}$ . Using the above observation,  $[\pi]_{\sim_{\mathcal{B}}}$  can be constructed explicitly by compiling all topological sortings.

To ease notation, we define  $|\pi| = |\mathcal{B}_\pi|$ . We also adopt the terminology from sets and say that a linear ordering  $\pi$  is *optimal*, if its induced feedback arc set  $\mathcal{B}_\pi$  is. To express that a linear ordering  $\pi$  is optimal, we denote it by  $\pi^*$ , i. e., we always have  $|\pi_G^*| = \tau_G$ . Note that all linear orderings in the equivalence class  $[\pi^*]_{\sim_{\mathcal{B}}}$  of an optimal linear ordering  $\pi^*$  are also optimal.

For a linear ordering  $\pi$ , we define the *reverse linear ordering*  $\pi^R$  as  $\pi^R : V \rightarrow \{0, \dots, n-1\}$  where  $\pi^R(v) = n-1 - \pi(v)$  for every vertex  $v \in V$ . This definition complies well with the definition of the reverse graph: If  $\pi_G$  is a linear ordering of  $G$ , then  $\pi_G^R$  forms the

corresponding linear ordering for  $G^R$  such that  $\mathcal{B}_{\pi_G^R}$  equals the set of backward arcs induced by  $\pi_{G^R}$  on  $G^R$ , i. e.,  $\mathcal{B}_{\pi_G^R} = \mathcal{B}_{\pi_{G^R}}$ . In particular, this implies that  $\pi_G$  is an optimal linear ordering of  $G$  if and only if  $\pi_{G^R}$  is an optimal linear ordering of  $G^R$ . Note that also the reversion of a linear ordering is an involution, i. e.,  $\pi_{G^{RR}} = \pi_G$ .

Finally, we simplify  $\mathcal{B}_\pi$  and  $\mathcal{F}_\pi$  to  $\mathcal{B}$  and  $\mathcal{F}$ , respectively, if the linear ordering that these sets are obtained from is unambiguous. In the context of a linear ordering  $\pi$ ,  $\mathcal{B}$  and  $\mathcal{F}$  always refer to  $\mathcal{B}_\pi$  and  $\mathcal{F}_\pi$ .


### 3.3.3 Forward Paths and Layouts

Given a linear ordering  $\pi$  of a graph  $G = (V, A)$ , a path  $P$  is called *forward path* if every arc in  $P$  is forward.

For every vertex  $v \in V$ ,  $\pi$  defines four (multi-)sets of arcs:  $\mathcal{F}^-(v)$ ,  $\mathcal{F}^+(v)$ ,  $\mathcal{B}^-(v)$ ,  $\mathcal{B}^+(v)$ , which contain the incoming forward, outgoing forward, incoming backward, and outgoing backward arcs of  $v$  according to  $\pi$ . Additionally, we use  $\mathcal{F}(v) = \mathcal{F}^-(v) \cup \mathcal{F}^+(v)$  and  $\mathcal{B}(v) = \mathcal{B}^-(v) \cup \mathcal{B}^+(v)$  if we do not want to distinguish between incoming and outgoing arcs. We denote the cardinalities by  $f^-(v) = |\mathcal{F}^-(v)|$ ,  $f^+(v) = |\mathcal{F}^+(v)|$ ,  $b^-(v) = |\mathcal{B}^-(v)|$ , and  $b^+(v) = |\mathcal{B}^+(v)|$ . Analogously, we set  $f(v) = f^-(v) + f^+(v) = |\mathcal{F}(v)|$ , and  $b(v) = b^-(v) + b^+(v) = |\mathcal{B}(v)|$  to express the total number of forward and backward arcs incident to  $v$ , respectively. For ease of notation, let  $\varphi[X]$ , for  $\varphi \in \{\mathcal{F}^-, \mathcal{F}^+, \mathcal{B}^-, \mathcal{B}^+, f^-, f^+, b^-, b^+\}$ , be the canonical extension of the function  $\varphi$  to sets of elements, e. g.,  $\mathcal{F}^-[X] = \bigcup_{v \in X} \mathcal{F}^-(v)$  and  $f^-[X] = |\mathcal{F}^-[X]|$ . The linear ordering  $\pi$  induces a *layout*  $\mathcal{L} : V \rightarrow \mathbb{N}_0^4$  such that for each vertex  $v \in V$ ,  $\mathcal{L}(v)$  is the 4-tuple  $(f^-(v), f^+(v), b^-(v), b^+(v))$ .

Note that if we consider the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^R$ , then for every vertex  $v \in V$ , its incoming forward arcs according to  $\pi$  in  $G$  correspond to its outgoing forward arcs according to  $\pi^R$  in  $G^R$  and its outgoing forward arcs according to  $\pi$  in  $G$  correspond to its incoming forward arcs according to  $\pi^R$  in  $G^R$ . Likewise, its incoming backward arcs according to  $\pi$  in  $G$  correspond to its outgoing backward arcs according to  $\pi^R$  in  $G^R$  and its outgoing backward arcs according to  $\pi$  in  $G$  correspond to its incoming backward arcs according to  $\pi^R$  in  $G^R$ .

In order to improve comprehensibility, we use pictograms that visualize the layout as a shorthand notation for the 4-tuples, depending on the degree of accuracy needed. In a pictogram, a simple arc ( $\nearrow$ ) always signifies exactly one arc, whereas an arc with a double shaft ( $\rightrightarrows$ ) serves as a representative for a set of arcs. We use a dotted version

( $\nabla$ ), if the represented arc set may be empty, otherwise, the lines are drawn solid. In the latter two cases, identical colors and drawing styles of two double shaft arcs indicate identical arc set cardinalities, provided that they are incident to the same vertex. (We would run out of drawing styles if we also kept this invariant across different vertices.) The pictogram , for instance, represents a vertex  $v$  with  $\mathcal{L}(v) = (i, j, k, l)$ , where  $i \geq 0, j = k + 1, k \geq 1$ , and  $l = 2$ .

### 3.4 Preprocessing and Default Assumptions

The LINEAR ORDERING problem consists in finding an optimal linear ordering  $\pi^*$  of a graph  $G$ . Prior to tackling this task, there are a few preprocessing steps which can be taken to facilitate the handling of  $G$ .

In this section, we will introduce them briefly and see how they enable us to set up some default assumptions on the input graphs. Unless indicated otherwise, the restrictions listed here shall be effective for the remainder of this thesis.

#### 3.4.1 Loops and Anti-Parallel Arcs

Consider a graph  $G = (V, A)$  as input to the LINEAR ORDERING problem.

By the definition of a linear ordering, a loop is always classified as a backward arc. Let  $l$  denote the number of loops of the input graph  $G$  and let  $G'$  be the subgraph of  $G$  that contains all arcs but loops. Then, every linear ordering of  $G'$  is also a linear ordering of  $G$  and vice versa, whose set of backward arcs differ by exactly the  $l$  loops.

The same applies to pairs of anti-parallel arcs. Let  $\{(u, v), (v, u)\} \subseteq A$  be such a pair and let  $G'$  be the subgraph of  $G$  restricted to  $A \setminus \{(u, v), (v, u)\}$ . Note that parallel arcs of  $(u, v)$  or  $(v, u)$ , if existent, remain in  $G'$ . Again, every linear ordering of  $G'$  is also a linear ordering of  $G$  and vice versa, whose set of backward arcs differ by exactly 1: In case that  $u$  has a smaller LO position than  $v$ ,  $G$  has  $(v, u)$  as an additional backward arc, otherwise, the additional backward arc is  $(u, v)$ .

In consequence, loops and anti-parallel arcs can be deemed irrelevant for the construction of a linear ordering. We can therefore remove all loops from the input graph as well as anti-parallel arcs pairwise and proceed with the resulting graph, which is simple up to parallel arcs. After the construction of the linear ordering, these arcs are reinserted.

The manipulation of graphs and their effects on the LINEAR ORDERING problem are also discussed in more detail in [Section 4.10.1](#).

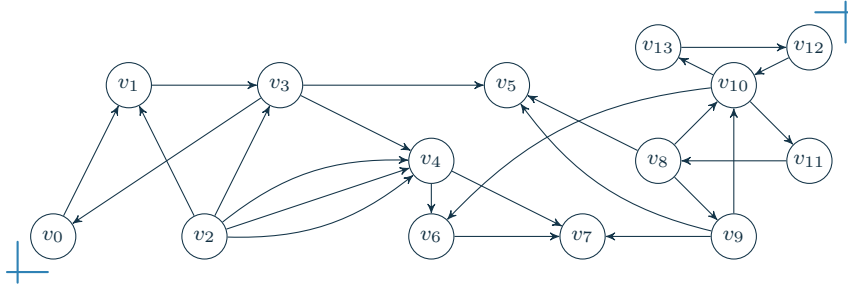


Figure 3.9: The example graph of the previous section after the removal of loops and anti-parallel arcs.

In the following, we confine ourselves to considering only graphs without loops and anti-parallel arcs:

---

**Assumption 3.1**

The input graph is free of loops and anti-parallel arcs.

---

The result of the application of the preprocessing steps described above to the example graph of the previous section is shown in Figure 3.9.

### 3.4.2 Strong Connectivity

Consider now a graph  $G = (V, A)$  that is not strongly connected. In Section 3.2.4, we have already seen that no cycle of the graph can contain vertices of different strongly connected components. It therefore suffices to solve the problem on the SCCs separately and combine the results afterwards:

Let  $\sigma_0, \dots, \sigma_k$  be the strongly connected components of  $G$  and let  $\pi_0, \dots, \pi_k$  be linear orderings of them, respectively. Denote by  $\mathcal{B}_0, \dots, \mathcal{B}_k$  the according induced sets of backward arcs. Then,  $\mathcal{B} = \bigcup_{0 \leq i \leq k} \mathcal{B}_i$  is a feedback arc set of  $G$  with  $|\mathcal{B}| = \sum_{0 \leq i \leq k} |\mathcal{B}_i|$ .

A corresponding linear ordering  $\pi$  of  $G$  can be obtained by either sorting  $G|_{\mathcal{F}}$  topologically (cf. Section 3.3) or by “concatenating” the linear orderings of the SCCs in the order of the topological sorting of the condensation of  $G$ , i. e., if  $\xi$  is a topological sorting of the condensation of  $G$  and  $u, v$  are two vertices of  $G$ , then  $\pi(u) < \pi(v)$  if and only if either  $u$  and  $v$  are vertices of the same SCC  $\sigma_i$  and  $\pi_i(u) < \pi_i(v)$  or  $u$  belongs to SCC  $\sigma_i$  and  $v$  belongs to SCC  $\sigma_j$ ,  $i \neq j$ , and  $\xi(\sigma_i) < \xi(\sigma_j)$ . Note that for the sake of simplicity, the SCCs and the vertices of the condensation graph have been identified in this definition.

This allows us to impose the following restriction:

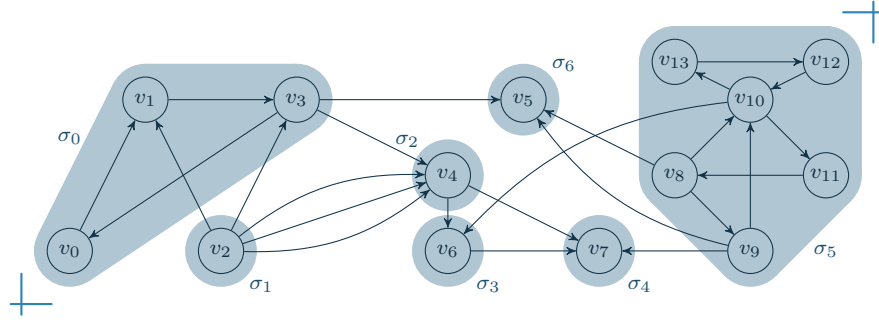


Figure 3.10: The strongly connected components of the graph depicted in Figure 3.9.

---



---

### Assumption 3.2

The input graph is always strongly connected.

---

For illustration, Figure 3.10 depicts the strongly connected components of the graph shown in Figure 3.9, i. e., the example graph of the previous section after the removal of loops and anti-parallel arcs.

### 3.4.3 Biconnectivity

Finally, consider the case that  $G = (V, A)$  is strongly connected, but not biconnected. The following fact is commonly known (cf. [Sta90]):

---



---

### Proposition 3.1

A set of arcs  $\mathcal{B} \subseteq A$  is a feedback arc set for a graph  $G = (V, A)$  if and only if  $\mathcal{B} = \bigcup_i \mathcal{B}_i$  such that  $\mathcal{B}_i$  is a feedback arc set of  $\beta_i$ , for every block  $\beta_i$  of  $G$ .

---

As this may not be entirely obvious at first glance, we provide a short proof here.

*Proof.* Let  $\beta_0, \dots, \beta_k$  denote the blocks of  $G$  and let  $\mathcal{T}$  be the block-cut tree of  $G$ . Consider two vertices  $u, v \in V$  that are not cut vertices and contained in blocks  $\beta_u$  and  $\beta_v$ , respectively, such that  $\beta_u \neq \beta_v$ . As  $G$  is strongly connected, there must be paths  $P = u \rightsquigarrow v$  and  $P' = v \rightsquigarrow u$  in  $G$ . Both paths contain all cut vertices  $X$  on the path from  $\beta_u$  to  $\beta_v$  in  $\mathcal{T}$ . Let  $C$  denote the cycle consisting of  $P$  and  $P'$ . In this case,  $C$  contains all vertices in  $X$  at least twice, i. e.,  $C$  is not simple. Instead,  $C$  consists of a set of simple cycles formed by the subpaths between two cut vertices. The vertices of every such simple cycle are subsequently contained in a common block of  $G$ . Furthermore,  $C$  is

covered by a feedback arc set if and only if every cycle of this set of simple cycles is covered.  $\square$

Hence, it suffices to solve the LINEAR ORDERING separately on each block and combine the result afterwards, which allows us assume:

---



---

**Assumption 3.3**

The input graph is biconnected.

---

Let  $\beta_0, \dots, \beta_k$  again denote the blocks of  $G$  and let  $\pi_0, \dots, \pi_k$  be their respective linear orderings. A linear ordering  $\pi$  of the graph  $G$  can be obtained by sorting the acyclic subgraph  $G|_{\mathcal{F}}$  that results from removing all arcs that have been classified as backward by one of  $\pi_0, \dots, \pi_k$  topologically, as [Proposition 3.1](#) immediately suggests. Alternatively,  $\pi$  can be constructed directly from the  $\pi_0, \dots, \pi_k$  as follows: Traverse the block-cut tree  $\mathcal{T}$  of  $G$  by a depth-first search to obtain a preorder of the blocks and sort the linear orderings of the blocks according to this preorder. For simplicity, assume that  $\pi_0, \dots, \pi_k$  is already such an ordering. We incrementally construct a combined linear ordering: Let  $\pi^{(0)} = \pi_0$ . For  $0 < i \leq k$ , let  $v_i$  denote the only common vertex of  $\pi_i$  and  $\pi^{(i-1)}$ . This vertex must be a cut vertex, as it is part of at least two blocks, and, as  $\mathcal{T}$  is a tree, there can only be one such vertex. Construct  $\pi^{(i)}$  from  $\pi^{(i-1)}$  by replacing  $v_i$  in  $\pi^{(i-1)}$  with the linear ordering  $\pi_i$ . More formally, for two distinct vertices  $u \neq w$  contained in one of the blocks  $\beta_0, \dots, \beta_i$ ,  $\pi^{(i)}(u) < \pi^{(i)}(w)$  if and only if either both are contained in  $\beta_i$  and  $\pi_i(u) < \pi_i(w)$ , or both are not contained in  $\beta_i$  and  $\pi^{(i-1)}(u) < \pi^{(i-1)}(w)$ , or  $u$  is contained in  $\beta_i$ ,  $w$  is not contained in  $\beta_i$ , and  $\pi^{(i-1)}(v_i) < \pi^{(i-1)}(w)$ , or  $u$  is not contained in  $\beta_i$ ,  $w$  is contained in  $\beta_i$ , and  $\pi^{(i-1)}(u) < \pi^{(i-1)}(v_i)$ . Then,  $\pi = \pi^{(k)}$  is a linear ordering of  $G$  that conforms with every linear ordering  $\pi_i$ ,  $0 \leq i \leq k$ .

With respect to the aim of using all assumptions concurrently, let us briefly address the question of whether a block of a strongly connected graph is again strongly connected.

---



---

**Proposition 3.2**

Every block  $\beta$  of a strongly connected graph  $G$  is itself strongly connected.

---

*Proof.* Let  $\mathcal{T}$  be the block-cut tree of  $G$  and let  $u, v$  be two vertices of the same block  $\beta$  of  $\mathcal{T}$ . As  $G$  is strongly connected, there is a path  $P = u \rightsquigarrow v$  in  $G$ . If  $P$  contains only vertices of  $\beta$ ,  $P$  is also a path in  $\beta$ . Suppose  $P$  contains a vertex that does not belong to  $\beta$ . In this case,  $P$  must also pass through at least one cut vertex. Let  $w$  be the first cut vertex

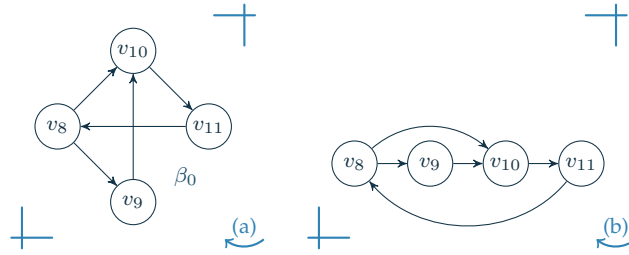


Figure 3.11: The block  $\beta_0$  of  $\sigma_5$  along with an (optimal) linear ordering.

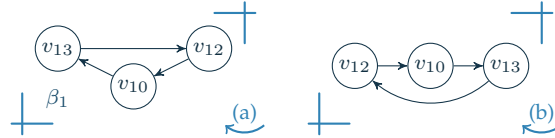


Figure 3.12: The block  $\beta_1$  of  $\sigma_5$  along with an (optimal) linear ordering.

encountered on a traversal of  $P$ . Then,  $w$  is contained in  $\beta$  and  $P$  leaves  $\beta$  at  $w$ . However,  $P$  must end at  $v$ , which is also contained in  $\beta$  and, as  $\mathcal{T}$  is a tree and  $w$  is a cut vertex,  $P$  must enter  $\beta$  again at  $w$ . Subsequently,  $\beta$  contains a path  $u \rightsquigarrow w$  as well as a path  $w \rightsquigarrow v$ . Hence,  $\beta$  contains a path  $u \rightsquigarrow v$  for every pair of vertices  $u, v$ , which implies that  $\beta$  is strongly connected.  $\square$

We can thus continue the preprocessing of the input graph after the removal of loops and pairs of anti-parallel arcs as well as the partition into strongly connected components by simply splitting these SCCs up into blocks.

The strongly connected components of the example graph, as visualized in Figure 3.10 are all biconnected except for  $\sigma_5$ , which has a cut vertex  $v_{10}$ . Figure 3.11 and Figure 3.12 show the two blocks of  $\sigma_5$  along with an optimal linear ordering for each block. The block-cut tree of  $\sigma_5$  is depicted in Figure 3.13(a). The linear ordering for  $\sigma_5$  obtained using the procedure described above is given in Figure 3.13(b).

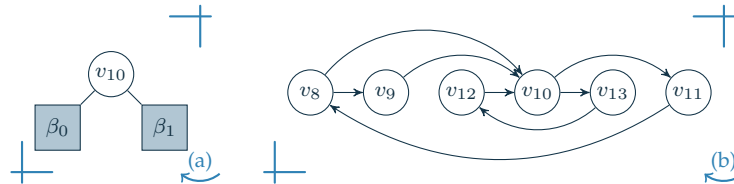


Figure 3.13: The block-cut tree of  $\sigma_5$  and the linear ordering constructed from those of  $\beta_0$  and  $\beta_1$ .



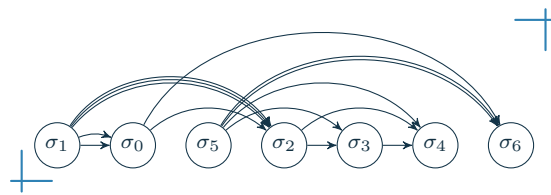


Figure 3.14: A topological sorting of the SCCs depicted in Figure 3.10.

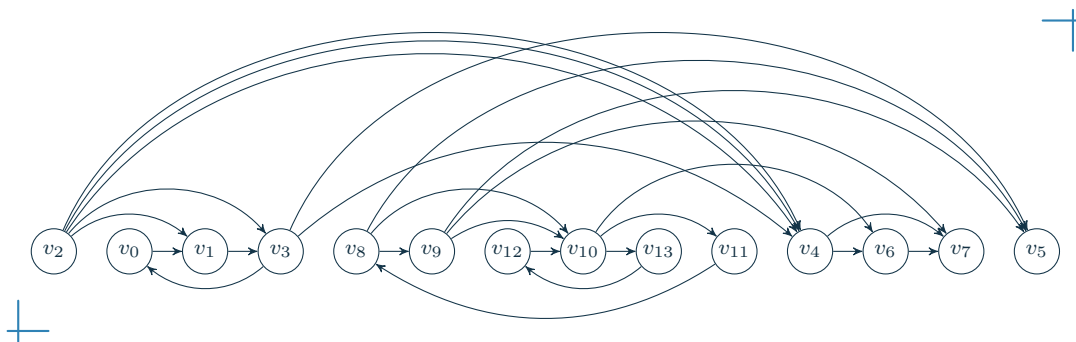


Figure 3.15: An (optimal) linear ordering of the graph depicted in Figure 3.10, obtained by concatenating the linear orderings of the SCCs in the order of the topological sorting as shown in Figure 3.14.

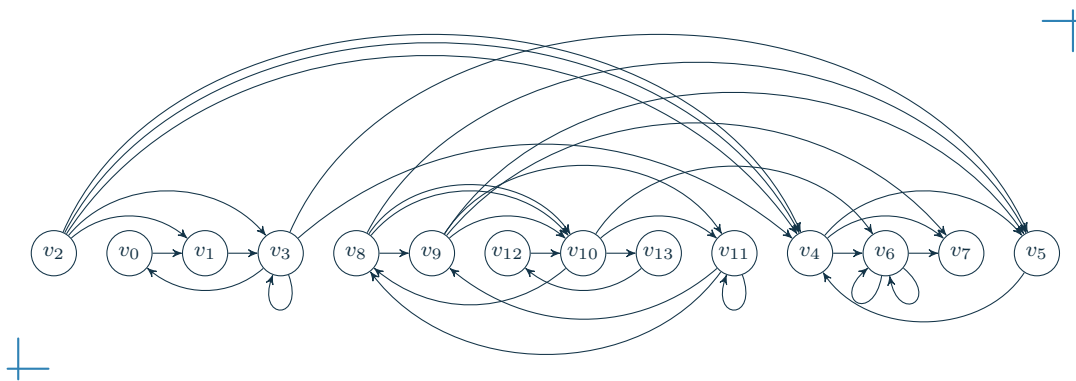


Figure 3.16: An optimal linear ordering of the example graph after the reinsertion of loops and pairs of anti-parallel arcs. The linear ordering corresponds to that depicted in Figure 3.15.

In order to finish the example, we next obtain a topological sorting of the condensation of the graph shown in [Figure 3.10](#), which is depicted in [Figure 3.14](#). Afterwards, the separately computed, here even optimal, linear orderings of the SCCs can be “concatenated” in the order of the topological sorting (cf. [Figure 3.15](#)).

Finally, the loops and anti-parallel arcs are reinserted into the graph, thus leading to a linear ordering of the original input graph. [Figure 3.16](#) provides the optimal linear ordering of the example graph that has arisen from these steps.

## 4 | Properties of Optimal Linear Orderings: A Microscopic View

Finding the absolutely best solution to a problem is often time consuming, and, if the instance is big enough, at times even impossible. In this chapter, we specifically study optimal solutions to the `LINEAR ORDERING` problem to gain more insight into their structure. In doing so, we apply a kind of microscopic view with the emphasis being placed primarily on single arcs and vertices, thus opposing more “macroscopic” approaches that operate on higher abstraction levels. Whereas this limitation might appear disadvantageous at first sight, it turns out to be powerful in sum due to synergy effects.

We benefit algorithmically in two respects: First, approximation algorithms and heuristics can be improved to yield better solutions while still running in polynomial time. In the course of this chapter, algorithms are developed that monotonically improve a given linear ordering. Second, exact algorithms can take advantage of these findings and be sped up. A third aspect consists in the ability to prove new bounds on the cardinality of an optimal solution for different kinds of graphs.

As a consequence, the contents provided here form the basis for all subsequent chapters.

### 4.1 General Framework

A solution to a problem is optimal if and only if it is at least as good as every other solution. This is the common definition for optimality. As `LINEAR ORDERING` is a minimization problem, the part “is as good as” translates to “induces at most as many backward arcs as” here. In order to obtain a more formal definition, we introduce a predicate `Opt` which expresses whether a linear ordering  $\pi$  is optimal, i. e.,  $\text{Opt}(\pi) =$

true, or not, i. e. ,  $\text{Opt}(\pi) = \text{false}$  or  $\neg\text{Opt}(\pi) = \text{true}$ . Then, we can express optimality as follows:

$$\text{Opt}(\pi) \Leftrightarrow \forall \pi' : |\pi| \leq |\pi'|$$

As mentioned at the beginning, finding such a solution for  $\mathcal{NP}$ -hard optimization problems such as LO and FAS is in any case resource consuming and sometimes impossible in practice. Instead, one tries to find good solutions, where “good” may be interpreted in different ways. A “good” solution may, e. g. , be one that is provably close to the optimum, with the downside that something about the optimal solution must be known. A different approach consists in optimizing a given solution as far as possible.

Optimality is unquestionably a very ambitious property for a linear ordering to achieve. In the following sections, we therefore establish a more accessible set of properties with the characteristic that if a linear ordering disrespects one of them, it is not optimal and can be improved in a prespecified way. A special class of such properties consists in those who require a coupling with another property such that the compliance of a linear ordering depends on the choice of the partner. We call such properties *meta-properties* and address a selection of them in the last section of this chapter.

Just as  $\text{Opt}$  indicates whether or not a linear ordering is optimal, we define a predicate for every introduced property that can be established efficiently and that is not a meta-property. Furthermore, we define a set  $\Psi$  that encompasses all these predicates. The “superpredicate”  $\Psi_{\text{opt}}$  then expresses that a linear ordering has all properties defined by the predicates in  $\Psi$ :

$$\Psi_{\text{opt}}(\pi) \Leftrightarrow \forall \psi \in \Psi : \psi(\pi)$$

We call a linear ordering  $\pi$   *$\Psi$ -optimal* if and only if  $\Psi_{\text{opt}}(\pi) = \text{true}$ . As a reminder, we point out that  $\pi^*$  always denotes an optimal linear ordering, i. e. ,  $\text{Opt}(\pi^*)$  is always true (cf. [Section 3.3](#)).

In this chapter, we prove the following two theorems:

---



---

**Theorem 4.1**

For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \Psi_{\text{opt}}(\pi)$ .

---

A characteristic of the properties in  $\Psi$  is that all can be established concurrently and in polynomial time, which allows us to implement a  $\Psi_{\text{opt}}$ -algorithm for LINEAR ORDERING.

---



---

**Theorem 4.2**

There is an  $\mathcal{O}(n \cdot m^2 \cdot \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ -time algorithm that constructs a  $\Psi$ -optimal linear ordering  $\pi$ .

---

In the individual sections of the chapter, we will introduce each property separately, prove its validity for optimal linear orderings, and, where applicable, provide a property-establishing algorithm. A common feature of these algorithms is that they can be conveniently formulated by means of the routine *Iterate* (cf. [Algorithm 4.1](#)): Let  $E(G, \pi)$  be an algorithm that takes a graph  $G$  and a linear ordering  $\pi$  as input and returns  $\pi$  if it respects the respective property. Otherwise, it applies an improvement to  $\pi$  and returns this new linear ordering. Then,  $\text{Iterate}(G, \pi, E)$  establishes the property by calling  $E(G, \pi)$  until no further improvement is possible, i. e.,  $E$ 's return value equals its argument  $\pi$ .

---

**Algorithm 4.1** *Iterate*


---

**Require:** graph  $G$ , linear ordering  $\pi$ , property-enforcing routine  $E(G, \pi)$

**Return:** a linear ordering obtained from  $\pi$  that respects the property enforced by  $E(G, \pi)$

```

1: procedure Iterate( $G, \pi, E$ )
2:    $\pi' \leftarrow \pi$ 
3:   repeat
4:      $\pi \leftarrow \pi'$ 
5:      $\pi' \leftarrow E(G, \pi)$ 
6:   until  $\pi = \pi'$ 
7:   return  $\pi$ 

```

---

A crucial characteristic of every property-enforcing routine  $E(G, \pi)$  is that it never returns a linear ordering  $\pi'$  such that  $|\pi'| > |\pi|$ . This monotonicity in combination with the wrapper algorithm *Iterate* yields a local search heuristic for every routine  $E$  which employs a technique that is also known as hill climbing. In contrast to simulated annealing or tabu search, no worsening of the solution, even if only temporarily, can occur.

The local search terminates as soon as  $E$  no longer modifies the linear ordering, i. e.,  $\pi = \pi'$  in the pseudocode. Prior to this, the linear ordering must have been improved in each call to the property-enforcing routine, i. e.,  $|\pi|$  decreases by at least one per loop

iteration. As  $1 \leq |\pi| \leq m$ , this enables us to bound the running time of  $\text{Iterate}(G, \pi, E)$  depending on  $E$  as follows<sup>1</sup>:

---



---

**Proposition 4.1**

$\text{Iterate}(G, \pi, E)$  runs in  $\mathcal{O}(m)$  times the running time of  $E(G, \pi)$ .

---

## 4.2 Algorithmic Setup

In the current chapter, several algorithms are devised that construct and manipulate linear orderings. The data structures given in this section serve as a basis for future time complexity analyses. Along with these, two initialization algorithms are introduced that may be used as ingredients to other algorithms later.

### 4.2.1 Graphs

We start with a suitable data structure to represent the input graph. Taking into account that the algorithms studied in the following sections construct or manipulate different linear orderings of the input graph, but not the graph itself, there is no need to attach great importance to the cost of vertex or arc additions and deletions. We are, however, interested in being able to efficiently obtain the vertices and arcs that are linked with a vertex. Furthermore, this thesis focuses especially on sparse graphs, hence, space complexity may be worth a consideration. For these reasons, we assume an adjacency list representation here, or, more precisely, an incidence list representation, i. e., every vertex stores a list of incident incoming and outgoing arcs. For an efficient handling of parallel arcs in multigraphs, we assume that they are represented blockwisely in the incidence lists, such that the parallel copies of an arc can be skipped in  $\mathcal{O}(1)$  in the traversal of an incidence list where required.

Additionally, every arc  $a$  knows its head and tail, which can be accessed by  $\text{head}(a)$  and  $\text{tail}(a)$  in constant time. Accessing the reverse of a graph does not cause more than constant overhead if only the roles of the lists of incoming and outgoing arcs at each vertex are swapped and heads are treated as tails and vice versa.

We further maintain  $\mathcal{F}$  and  $\mathcal{B}$  as sets and additionally a flag attached to each arc specifying whether this arc is classified as forward or backward according to the current

---

<sup>1</sup>See also [Section 4.2.5](#) for further remarks on an efficient implementation of  $\text{Iterate}$ .

linear ordering. Consequently, computing, e. g., a topological sorting of an acyclic subgraph can be accomplished by ignoring all arcs with flag “backward”.

### 4.2.2 Linear Orderings

There are essentially two possibilities of how a linear ordering can be implemented. Mathematically, it is modeled as a bijective function from the set of vertices  $V$  to the set of natural numbers ranging from 0 to  $n - 1$ . A data structure that comes relatively close to this is the map, which is sometimes also called an associative array or a dictionary. A different approach consists in storing the actual ordering of the vertices in a linear data structure such as an array or a linked list.

Both map and array share the advantage that the position of every vertex is stored explicitly (although in the case of an array, the mapping is technically from  $\{0, \dots, n - 1\}$  to  $V$ ). A drawback is, however, that even small changes to the relative ordering of the vertices, e. g., the movement of a single vertex to another position, requires up to  $\mathcal{O}(n)$  updates. In case of a linked list, the situation is precisely the reverse. Here, such a move operation can be accomplished in constant time, provided that the insertion position is known.

In anticipation of the algorithms introduced in the following, we opt for a representation of a linear ordering as a doubly-linked list, which allows for an efficient traversal of the ordering in both directions. If needed, the exact position of a vertex within a linear ordering must be computed explicitly as shown in [Section 4.2.4](#).

Whenever the reverse of a linear ordering is accessed or modified by an algorithm, we assume that only a different interface is used, such that there is no additional overhead and changes to the reverse linear ordering also take effect directly on the original linear ordering.

### 4.2.3 Vertex Layouts

A linear ordering induces a layout on each vertex. In [Section 3.3.3](#), we introduced the sets  $\mathcal{F}^-(v)$ ,  $\mathcal{F}^+(v)$ ,  $\mathcal{B}^-(v)$ , and  $\mathcal{B}^+(v)$  for a vertex  $v$ . As discussed earlier, we assume that the graph is represented using incidence lists. In order to also maintain these special sets, we add four additional incidence lists to each vertex. Every such list is sorted according to the positions of the tails in the current linear ordering for  $\mathcal{F}^-(v)$  and  $\mathcal{B}^-(v)$  and likewise according to the positions of the heads for  $\mathcal{F}^+(v)$  and  $\mathcal{B}^+(v)$ , in each case in

---

**Algorithm 4.2** Compute LO positions and forward/backward arc sets.

---

**Require:** graph  $G = (V, A)$ , linear ordering  $\pi$

**Return:** LO positions w. r. t.  $\pi$ , sets of backward and forward arcs  $\mathcal{B}, \mathcal{F}$

```

1: procedure ComputePositionsAndArcSets( $G, \pi$ )
2:    $i \leftarrow 0$ 
3:   for all  $v$  in the order of  $\pi$  do
4:      $\pi(v) \leftarrow i$             $\triangleright$  allow for fast “reverse lookup”:  $v$  has position  $i$  in  $\pi$ 
5:      $i \leftarrow i + 1$ 
6:    $\mathcal{F} \leftarrow \emptyset; \mathcal{B} \leftarrow \emptyset$ 
7:   for all  $(u, v) \in A$  do
8:     if  $\pi(u) < \pi(v)$  then
9:        $\mathcal{F} \leftarrow \mathcal{F} \cup \{(u, v)\}$ 
10:    else
11:       $\mathcal{B} \leftarrow \mathcal{B} \cup \{(u, v)\}$ 
12:   return  $\pi, \mathcal{B}, \mathcal{F}$ 

```

---

ascending order (with respect to the bijection  $\pi$ ). We assume that the lists are represented as doubly-linked lists in any implementation.

To denote the sorted list of arcs in  $\mathcal{F}^-(v)$  according to a linear ordering  $\pi$ , we introduce a notation that is based on the definition of lists in [Section 3.1](#) and write  $\langle \mathcal{F}^- \rangle_\pi(v)$ . Likewise, we obtain  $\langle \mathcal{F}^+ \rangle_\pi(v)$ ,  $\langle \mathcal{B}^- \rangle_\pi(v)$ , and  $\langle \mathcal{B}^+ \rangle_\pi(v)$  for  $\mathcal{F}^+(v)$ ,  $\mathcal{B}^-(v)$ , and  $\mathcal{B}^+(v)$ , respectively. If the linear ordering that these ordered lists refer to is clear from the context, the subscript may also be omitted.

#### 4.2.4 Initializing the Data Structures

Whereas we may readily assume that the graph is given with incidence lists, this is not a reasonable approach for the sorted lists introduced above. They depend heavily on a specific linear ordering, which is, by nature of this chapter, subject to change. The same applies to the exact position of a vertex within a linear ordering and the classification of arcs as forward or backward. Furthermore, algorithms may or may not make use of these additional structures. Therefore, it is desirable to be able to list their construction explicitly as a statement in pseudocode if needed.

To this end, we provide two initializing routines, *ComputePositionsAndArcSets*( $G, \pi$ ) as well as *ComputeLayoutLists*( $G, \pi$ ), that take a graph  $G$  and a linear ordering  $\pi$  as



**Algorithm 4.3** Compute vertex layouts.**Require:** graph  $G = (V, A)$ , linear ordering  $\pi$  with precomputed LO positions**Return:** sorted layout lists  $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle$ 


---

```

1: procedure ComputeLayoutLists( $G, \pi$ )
2:   for all  $v \in V$  do initialize  $\langle \mathcal{F}^- \rangle(v), \langle \mathcal{F}^+ \rangle(v), \langle \mathcal{B}^- \rangle(v), \langle \mathcal{B}^+ \rangle(v)$  with  $\langle \rangle$ 
3:   for all  $v$  in the order of  $\pi$  do
4:     for all incoming arcs  $(u, v)$  of  $v$  do
5:       if  $\pi(u) < \pi(v)$  then
6:          $\langle \mathcal{F}^+ \rangle(u) \leftarrow \langle \mathcal{F}^+ \rangle(u) \diamond \langle (u, v) \rangle$ 
7:       else
8:          $\langle \mathcal{B}^+ \rangle(u) \leftarrow \langle \mathcal{B}^+ \rangle(u) \diamond \langle (u, v) \rangle$ 
9:     for all outgoing arcs  $(v, u)$  of  $v$  do
10:      if  $\pi(v) < \pi(u)$  then
11:         $\langle \mathcal{F}^- \rangle(u) \leftarrow \langle \mathcal{F}^- \rangle(u) \diamond \langle (v, u) \rangle$ 
12:      else
13:         $\langle \mathcal{B}^- \rangle(u) \leftarrow \langle \mathcal{B}^- \rangle(u) \diamond \langle (v, u) \rangle$ 
14:   return  $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle$ 

```

---

input and perform these tasks. [Algorithm 4.2](#) and [Algorithm 4.3](#) show their respective details.

We are able to identify two separate blocks in the listing of *ComputePositionsAndArcSets*: In [lines 3–5](#), the algorithm iterates over all vertices in the order of their appearance within  $\pi$ . Concurrently, it maintains a counter variable  $i$ , which in each iteration corresponds to the position of the current vertex. For every vertex, the value of  $i$  is stored explicitly as an attribute. Next, in [lines 7–11](#), the previously computed vertex positions are used to construct the sets  $\mathcal{F}$  and  $\mathcal{B}$ .

*ComputeLayoutLists* starts by initializing the sorted lists  $\langle \mathcal{F}^- \rangle(v), \langle \mathcal{F}^+ \rangle(v), \langle \mathcal{B}^- \rangle(v)$ , and  $\langle \mathcal{B}^+ \rangle(v)$  with empty lists. In the loop spanning [lines 3–13](#), the algorithm again iterates over all vertices in the order they appear in  $\pi$ . For every vertex  $v$ , first all incoming arcs are considered. Let  $(u, v)$  be such an arc. If  $(u, v)$  is forward, then at the arc's tail,  $u$ ,  $(u, v)$  must be added to the end of  $u$ 's sorted list of outgoing forward arcs: The heads of all outgoing forward arcs of  $u$  that are not in the list yet must have a greater position in  $\pi$  than  $v$ , due to the fact that the algorithm iterates over the vertices

in the order of  $\pi$ . The same argument applies if  $(u, v)$  is backward, and likewise for all outgoing arcs of  $v$ .

---



---

**Lemma 4.1**

The initialization routines  $\text{ComputePositionsAndArcSets}(G, \pi)$  and  $\text{ComputeLayoutLists}(G, \pi)$  run in time  $\mathcal{O}(m)$  each.

---

*Proof.* Consider [Algorithm 4.2](#) first. The first block, [lines 3–5](#), iterates over all vertices of  $V$ . For each vertex, storing its position and incrementing the counter variable  $i$  can be accomplished constant time. Hence, the first block requires  $\mathcal{O}(n)$  steps. The second block, [lines 7–11](#), consists in iterating over all arcs of the graph, each time comparing two previously computed values and adding an arc to either  $\mathcal{F}$  or  $\mathcal{B}$ . For every arc, this can be done in constant time, so the running time of the second block is in  $\mathcal{O}(m)$ .

Consider now [Algorithm 4.3](#) and assume that the linear ordering  $\pi$  is already equipped with precomputed LO positions. The initialization with empty lists requires  $\mathcal{O}(n)$  time. The remainder of the listing contains three nested loops. The outer loop iterates again over all vertices. In its body, there are two more loops that iterate over the incoming and outgoing arcs, respectively, of the current vertex  $v$ . The body of every inner loop consists in querying the LO positions of an arc's end vertices and appending the arc to the end of a list, both of which can be accomplished in constant time if the LO positions have been precomputed. The running time of the inner loops is therefore in  $\mathcal{O}(d^-(v))$  and  $\mathcal{O}(d^+(v))$ , respectively. Consequently, this block requires  $\sum_{v \in V} d^-(v) + d^+(v) = 2m$  steps, which is in  $\mathcal{O}(m)$ .

In conclusion, with  $n \in \mathcal{O}(m)$  due to  $G$  being strongly connected, the running time of both  $\text{ComputePositionsAndArcSets}(G, \pi)$  and  $\text{ComputeLayoutLists}(G, \pi)$  is in  $\mathcal{O}(m)$ .  $\square$

We stipulate that every algorithm that needs to access these additional structures is required to call the routines  $\text{ComputePositionsAndArcSets}$  and, if necessary,  $\text{ComputeLayoutLists}$  prior to querying them. After manipulating a linear ordering, it is the algorithm's obligation to update the structures accordingly if it requires further access to them.

### 4.2.5 General Remarks

All algorithms presented in this chapter adhere to the concept of referential transparency, i. e. , calling a function or an algorithm with the same parameters will always produce the same result and there are no side-effects. For this reason, a property-enforcing routine  $E$ , e. g. , explicitly returns a linear ordering. In a faithful implementation in real code, this may result in additional overhead for copying the linear orderings back and forth and checking explicitly whether the linear ordering has actually been changed by  $E$  requires another  $\mathcal{O}(n)$  steps. By waiving referential transparency and modifying the linear ordering passed as argument directly or an intelligent copy-on-write implementation, these expenses can be saved.

To avoid fundamental misconception of the presented algorithms, however, we stick to referential transparency in pseudocode, but assume an efficient realization for the runtime analyses. This is to apply to all algorithms, functions, and (sub-) routines used in this chapter.

As noted before, we also assume that accessing the reverse of a graph or linear ordering is only a change of the interface and thus produces no additional overhead.

## 4.3 Nesting Property

*Local search* is a widely-used postprocessing scheme for optimization problems that iteratively improves a solution by applying local changes. The first property that we consider here, the Nesting Property, can be regarded as the result of an algorithm that belongs in this class.

### 4.3.1 A 1-opt Algorithm

In general,  $k$ -opt describes a class of heuristics that perform a local search by modifying  $k$  elements of the current solution at a time. Large values of  $k$  are typically borne by running times that are exponential in  $k$ . We consider a setting where  $k = 1$ . For a linear ordering  $\pi$  of a graph  $G$ , a vertex  $v$ , and a position  $p \in \{0, \dots, n - 1\}$ , we define the operation  $\text{Move}(\pi, v, p)$ , which returns the linear ordering obtained from  $\pi$  by moving  $v$  to position  $p$ , as follows:

Let  $v_i$  denote the vertex at position  $i$  within  $\pi$ , i. e. ,  $\pi(v_i) = i$ . The input linear ordering thus reads  $\pi = (v_0, \dots, v_{q-1}, v_q, v_{q+1}, \dots, v_{n-1})$ . Calling  $\text{Move}(\pi, v_q, p)$  then

**Algorithm 4.4** A 1-opt Procedure**Require:** graph  $G$ , linear ordering  $\pi$ **Return:**  $\pi$  if it is 1-opt, otherwise an improved linear ordering  $\pi'$ 


---

```

1: procedure OneOpt( $G, \pi$ )
2:   for all  $v \in V$  do
3:      $b_o \leftarrow b^-(v) + b^+(v)$  according to current position of  $v$ 
4:      $p \leftarrow$  position of  $v$  that minimizes  $b^-(v) + b^+(v)$ 
5:      $b_n \leftarrow b^-(v) + b^+(v)$  according to position  $p$ 
6:     if  $b_n < b_o$  then
7:        $\pi' \leftarrow \text{Move}(\pi, v, p)$ 
8:     return  $\pi'$ 
9:   return  $\pi$ 

```

---

yields the linear ordering  $\pi' = (v_0, \dots, v_{p-1}, v_q, v_p, \dots, v_{q-1}, v_{q+1}, \dots, v_{n-1})$ , if  $p < q$ , and  $\pi' = (v_0, \dots, v_{q-1}, v_{q+1}, \dots, v_{p-1}, v_p, v_q, \dots, v_{n-1})$ , if  $p > q$ . In case  $p = q$ ,  $\pi' = \pi$ .

Using this operation, we can implement a 1-opt algorithm for LO using the procedure *OneOpt* shown in [Algorithm 4.4](#): For a vertex  $v$  of the graph, consider all positions within  $\pi$  and select the position  $p$  that minimizes the number of backward arcs incident to  $v$ , i. e.,  $b^-(v) + b^+(v)$  (cf. [line 4](#)). Move  $v$  to  $p$  if this reduces the number of backward arcs and return, otherwise continue with the next vertex. In fact, a very similar procedure that is known as “Sifting” is used in the context of crossing minimization [[BB04](#)]. By combining *OneOpt*( $G, \pi$ ) with *Iterate*, we obtain a 1-opt algorithm for LO. However, *OneOpt*, as it is listed in [Algorithm 4.4](#), needs  $\mathcal{O}(n^2)$  steps if, for every vertex  $v \in V$ , it naively examines every position within  $\pi$  in [line 4](#).

For this reason, we devise an enhanced implementation of this 1-opt algorithm that is listed as *EnforceNesting*( $G, \pi$ ) in [Algorithm 4.5](#). As we will see in the next subsection, the arcs of a linear ordering that is 1-opt form a sort of nesting, hence the name. This algorithm makes use of the additional data structures described in [Section 4.2](#). In the first two steps of the algorithm, it therefore calls *ComputePositionsAndArcSets* and *ComputeLayoutLists* to set these up.

The improvement with regard to the routine *OneOpt* in [Algorithm 4.4](#) is based on the following observation: Let  $v \in V$  be a vertex. Consider the number of backward arcs incident to  $v$  while  $v$  is moved to positions  $0, 1, \dots, n - 1$  in turn. At position 0,  $b^-(v) + b^+(v) = d^-(v)$ . More precisely,  $b^+(v) = 0$ , as  $v$  is already at the smallest position within  $\pi$ , and  $b^-(v) = d^-(v)$ . By moving  $v$  to position 1,  $v$  “skips” the vertex

**Algorithm 4.5** Nesting Property**Require:** graph  $G$ , linear ordering  $\pi$ **Return:**  $\pi$  if  $\text{Nest}(\pi)$ , otherwise an improved linear ordering  $\pi'$ 


---

```

1: procedure EnforceNesting( $G, \pi$ )
2:    $\pi, \mathcal{B}, \mathcal{F} \leftarrow \text{ComputePositionsAndArcSets}(G, \pi)$ 
3:    $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle \leftarrow \text{ComputeLayoutLists}(G, \pi)$ 
4:   for all  $v \in V$  do
5:      $R(v) \leftarrow \text{Merge}(\langle \mathcal{F}^+ \rangle(v), \pi \circ \text{head}, \langle \mathcal{B}^- \rangle(v), \pi \circ \text{tail})$ 
6:      $L(v) \leftarrow \text{Merge}(\langle \mathcal{F}^- \rangle(v), \pi \circ \text{tail}, \langle \mathcal{B}^+ \rangle(v), \pi \circ \text{head})$ 
7:      $c \leftarrow 0; \text{cmin} \leftarrow 0; p \leftarrow \pi(v)$ 
8:     for all  $a$  in the order of  $R(v)$  do
9:       if  $a \in \mathcal{F}$  then  $c \leftarrow c + 1$ 
10:      else
11:         $c \leftarrow c - 1$ 
12:        if  $c < \text{cmin}$  then  $\text{cmin} \leftarrow c; p \leftarrow \pi(\text{tail}(a))$ 
13:       $c \leftarrow 0$ 
14:      for all  $a$  in the reverse order of  $L(v)$  do
15:        if  $a \in \mathcal{F}$  then  $c \leftarrow c + 1$ 
16:        else
17:           $c \leftarrow c - 1$ 
18:          if  $c < \text{cmin}$  then  $\text{cmin} \leftarrow c; p \leftarrow \pi(\text{head}(a))$ 
19:      if  $\text{cmin} < 0$  then
20:         $\pi' \leftarrow \text{Move}(\pi, v, p)$ 
21:      return  $\pi'$ 
22:   return  $\pi$ 

```

---

that was previously at position 1. Whenever  $v$  is moved to the next position, it skips another vertex. Now consider how  $b^-(v) + b^+(v)$  changes while  $v$  hops through the linear ordering. Whenever  $v$  skips a vertex  $u$  such that  $(v, u) \in A$ , then  $(v, u)$  changes from outgoing forward to outgoing backward, i. e.,  $b^+(v)$  increases. On the other hand, if  $v$  skips a vertex  $u$  such that  $(u, v) \in A$ , then  $(u, v)$  changes from incoming backward to incoming forward, i. e.,  $b^-(v)$  decreases. If  $v$  skips a vertex that is not adjacent to  $v$ , then  $b^-(v) + b^+(v)$  remains unchanged.

*EnforceNesting* exploits this by only considering those positions where  $v$  skips an adjacent vertex. In order to find the interesting positions, it constructs two lists  $R(v)$  and  $L(v)$  for the current vertex  $v$ : In [line 5](#),  $R(v)$  is obtained by merging the sorted lists  $\langle \mathcal{F}^+ \rangle(v)$  and  $\langle \mathcal{B}^- \rangle(v)$  using a routine *Merge*.  $\text{Merge}(L_1, f_1, L_2, f_2)$  takes two sorted lists as well as two functions  $f_1$  and  $f_2$  as input and returns a list with the elements of  $L_1$  and  $L_2$  that is sorted using the results of  $f_1$  and  $f_2$  applied to the elements of  $L_1$  and  $L_2$ , respectively. As  $R(v)$  is constructed from  $\langle \mathcal{F}^+ \rangle(v)$  with function  $\pi \circ \text{head}$  and  $\langle \mathcal{B}^- \rangle(v)$  with function  $\pi \circ \text{tail}$ , it contains the outgoing forward and incoming backward arcs of  $v$  that are sorted using the position of the heads for the arcs in  $\mathcal{F}^+(v)$  and the position of the tails for the arcs in  $\mathcal{B}^-(v)$ . In [line 6](#),  $L(v)$  is obtained analogously by merging  $\langle \mathcal{F}^- \rangle(v)$  and  $\langle \mathcal{B}^+ \rangle(v)$ . In contrast to the consideration we just made, *EnforceNesting* starts at the current position of every vertex instead of position 0. With the arcs in  $R(v)$ , positions greater than  $v$ 's current position are considered, while examining  $L(v)$  results in considering the positions smaller than  $v$ 's.

In [line 7](#), three variables are initialized: The variable  $c$  can be thought of as representing the balance between additional forward and backward arcs. Whenever skipping an adjacent vertex implies that the number of backward arcs increases, it is incremented, if the number of forward arcs increases, it is decremented. The actual number of incident backward arcs of  $v$  at a position thus equals  $b_{\pi}^-(v) + b_{\pi}^+(v) + c$ . Initially,  $c$  is set to 0. The variable  $\text{cmin}$  maintains the minimum value of  $c$  encountered so far and  $p$  stores the corresponding position of the adjacent vertex within  $\pi$ . They are initialized with 0 and the current position of  $v$ , respectively.

In [lines 8–12](#), the algorithm iterates over the arcs in  $R(v)$  in order. Let  $a$  be an arc of  $R(v)$ . If  $a$  is a forward arc with respect to the current linear ordering, then  $a$  becomes backward as soon as  $v$  skips  $a$ 's head. Otherwise, if  $a$  is backward,  $a$  becomes a forward arc once  $v$  skips  $a$ 's tail. In the former case,  $c$  must be incremented, because the number of incident backward arcs increases. In the latter,  $c$  must be decremented. [Line 9](#) and [line 11](#) contain exactly this update of  $c$ . If this leads to a new minimum value of  $c$ , the variables  $\text{cmin}$  and  $p$  are updated accordingly in [line 12](#).

In the loop spanning [lines 8–12](#), only positions greater than the current position of  $v$  are considered. Therefore, in [lines 14–18](#), the arcs contained in  $L(v)$  are considered. For that purpose,  $c$  is reset to 0. Note that this does not apply to  $\text{cmin}$  and  $p$ . As  $L(v)$  contains the arcs sorted in ascending order with regard to  $\pi$ , but  $c$  stores relative changes to  $b_{\pi}^-(v) + b_{\pi}^+(v)$  starting from the current position of  $v$ , the arcs must be considered in descending order, i. e.,  $L(v)$  needs to be traversed backwards. The body of this loop

corresponds to that of the first loop, with the only exception that  $p$  must be obtained from the position of the arc's head.

Finally, if  $cmin$  is negative, moving  $v$  to the corresponding position reduces the number of incident backward arcs. This is checked, and, if applicable, realized in [lines 19–21](#). Observe that if  $p$  is the original position of a vertex  $u$  and  $p$  is greater than  $v$ 's original position, then  $Move(\pi, v, p)$  moves  $v$  to the position right behind  $u$ . Otherwise, if  $p$  is smaller than  $v$ 's original position,  $Move(\pi, v, p)$  moves  $v$  right before  $u$ . In both cases, the vertex  $u$  is skipped as intended.

---



---

**Lemma 4.2**

$EnforceNesting(G, \pi)$  runs in time  $\mathcal{O}(m)$ .

---

*Proof.* Consider the procedure  $EnforceNesting$  as given in [Algorithm 4.5](#). The calls to  $ComputePositionsAndArcSets$  in [line 2](#) and  $ComputeLayoutLists$  in [line 3](#) run in time  $\mathcal{O}(m)$  by [Lemma 4.1](#). Next, in [lines 4–21](#), a loop follows that iterates over all vertices of the graph. Let  $v$  be one of the considered vertices.

The loop's body starts with computing the sorted lists  $R(v)$  and  $L(v)$  by merging  $\langle \mathcal{F}^+ \rangle(v)$  and  $\langle \mathcal{B}^- \rangle(v)$  as well as  $\langle \mathcal{F}^- \rangle(v)$  and  $\langle \mathcal{B}^+ \rangle(v)$ . This is realized using the routine  $Merge(L_1, f_1, L_2, f_2)$ , which takes two sorted lists and two functions as input.  $Merge$  can be implemented in time  $\mathcal{O}(|L_1| + |L_2|)$  analogously to the merge step in the  $MergeSort$  sorting algorithm, assuming that each call to  $f_1$  or  $f_2$  returns in constant time. To this end,  $Merge(L_1, f_1, L_2, f_2)$  uses two iterators, one for  $L_1$  and the other for  $L_2$ , which both start at the first element of the list, respectively. Now the elements at the current iterator positions are passed as arguments to  $f_1$  and  $f_2$  and the results are compared. The element that yields the smaller value is then appended to the output list and the iterator of the input list which the element was taken from is advanced. Subsequently, the statements in [line 5](#) and [line 6](#) can be carried out in time  $\mathcal{O}(d(v))$ .

In [line 7](#), the variables  $c$ ,  $cmin$ , and  $p$  are set up. Due to the call to  $ComputePositionsAndArcSets$  at the beginning of the algorithm, the value of  $\pi(v)$  can be obtained in time  $\mathcal{O}(1)$ . The same applies to the updates of  $p$  in [line 12](#) and [line 18](#).

In the following, there are two loops spanning [lines 8–12](#) and [lines 14–18](#), respectively, whose bodies consist only of statements that can be carried out in constant time: the forward flag of an arc  $a$  is queried in the condition  $a \in \mathcal{F}$  of the if-clause and the variables  $c$ ,  $cmin$ , and  $p$  are updated. Hence, the running time of these loops is in  $\mathcal{O}(|R(v)| + |L(v)|) = \mathcal{O}(d(v))$ .

Finally, [line 19](#) checks whether a better position than the current one was found for  $v$ , and, if so, moves  $v$  there. If  $p$  is actually a natural number storing a concrete position within  $\pi$ ,  $\text{Move}(\pi, v, p)$  runs in time  $\mathcal{O}(n)$ . The move operation becomes more efficient, however, if  $p$  is merely a pointer to a position in the doubly-linked list representing  $\pi$  (cf. [Section 4.2](#)). In this case,  $\text{Move}(\pi, v, p)$  can be carried out in time  $\mathcal{O}(1)$ <sup>1</sup>. Due to the fact that *EnforceNesting* terminates immediately afterwards, *Move* is called at most once during the whole execution of *EnforceNesting*.

Hence, we have up to  $n$  iterations of the outer loop, and each vertex  $v$  that is considered has a cost of  $\mathcal{O}(d(v))$ , which yields a total of  $\mathcal{O}(\sum_{v \in V} d(v)) = \mathcal{O}(m)$  for all vertices. Additionally, we need  $\mathcal{O}(m)$  steps for the calls to *ComputePositionsAndArcSets* in [line 2](#) and *ComputeLayoutLists* in [line 3](#) as well as another  $\mathcal{O}(1)$  or  $\mathcal{O}(n)$  steps for the call to *Move* in [line 20](#).

In conclusion, as  $n \in \mathcal{O}(m)$ , *EnforceNesting* runs in time  $\mathcal{O}(m)$ , irrespective of the realization of  $p$ . □

In order to obtain a 1-opt algorithm, we combine *EnforceNesting* again with *Iterate*. Define  $\text{EstablishNesting}(G, \pi)$  as a synonym for  $\text{Iterate}(G, \pi, \text{EnforceNesting})$ . The combination of [Lemma 4.2](#) with [Proposition 4.1](#) then yields:

---

**Corollary 4.1**  
 $\text{EstablishNesting}(G, \pi)$  runs in time  $\mathcal{O}(m^2)$ .

---

### 4.3.2 Nesting Arcs

As has already been indicated, a linear ordering that is 1-opt contains a kind of nesting structure. We want to explore this a bit further.

Hence, we may ask ourselves, what does a linear ordering  $\pi$  “look like” after the execution of  $\text{EstablishNesting}(G, \pi)$ ? *EstablishNesting* terminates as soon as *EnforceNesting* cannot find an improved position for any vertex. Let us therefore consider such a vertex  $v$  and the listing in [Algorithm 4.5](#). The variable  $cmin$  indicates whether there is a position for  $v$  such that  $v$  has less incident backward arcs than before. In this case,  $cmin < 0$  (cf. [line 19](#)). Consequently, if no improvement is possible,  $cmin = 0$ , as  $cmin$  was initialized in [line 7](#) with 0 and only decreases in the course of the algorithm. If  $cmin$  always remains at 0, however, then  $c$  must always remain non-negative. This implies that

---

<sup>1</sup>See also [Section 4.2.5](#).



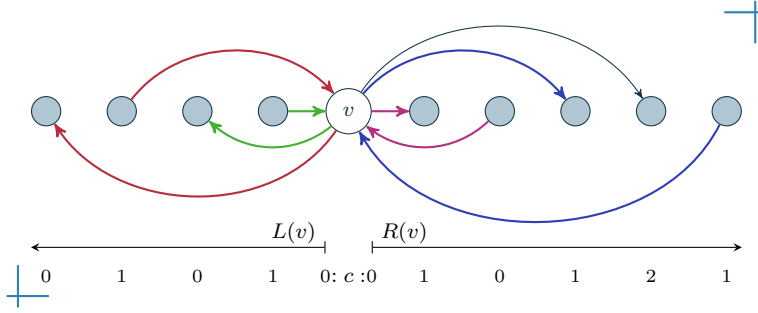


Figure 4.1: A vertex  $v$  whose position within the linear ordering cannot be improved by *EnforceNesting*.

during the iteration over  $R(v)$  in the first loop, the number of backward arcs encountered up to any element of  $R(v)$  never exceeds the number of processed forward arcs. The same holds for the second loop, which iterates over  $L(v)$ . In consequence, for every encountered backward arc, there must be a matching forward arc that was processed earlier. This forward arc must then be nested inside the corresponding backward arc. Figure 4.1 provides an example and also visualizes the values of  $c$  during the algorithm's iteration over  $R(v)$  and  $L(v)$ , respectively. The colors highlight the nesting pairs of arcs.

By expressing this nesting structure as property of a linear ordering, we obtain<sup>1</sup>:

---

**Lemma 4.3** NESTING PROPERTY

For every optimal linear ordering  $\pi^*$  of a graph  $G$ , there are two injective mappings  $\mu_h, \mu_t : \mathcal{B} \rightarrow \mathcal{F}$  such that

$$\begin{aligned} \mu_h((u, v)) = (v, x) &\Rightarrow \pi^*(x) < \pi^*(u) \quad \text{and} \\ \mu_t((u, v)) = (y, u) &\Rightarrow \pi^*(v) < \pi^*(y). \end{aligned}$$


---

Descriptively, the mapping  $\mu_h$  assigns each backward arc a nesting forward arc at its head and  $\mu_t$  likewise at its tail.

*Proof.* Let  $\pi$  be an arbitrary linear ordering of  $G$ . Consider the movement  $\text{Move}(\pi, v, p)$  of a vertex  $v$  to a new position  $p$  within  $\pi$ . If  $p > \pi(v)$ , then all outgoing forward arcs of  $v$  whose heads are at a position  $< p$  are turned into backward arcs and all incoming backward arcs with tail in the same range become forward. Likewise, if  $p < \pi(v)$ , all

<sup>1</sup>This result has also been published in an earlier conference article [HBA13].

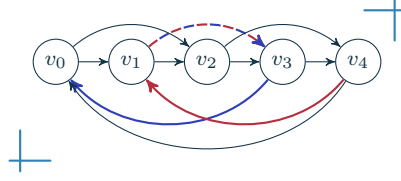


Figure 4.2: Defining  $\mu_h((v_4, v_1)) = (v_1, v_3) = \mu_t((v_3, v_0))$  conforms with Lemma 4.3 and thereby shows that  $\mu_h[\mathcal{B}] \cap \mu_t[\mathcal{B}] \neq \emptyset$  is possible.

incoming forward arcs become backward and outgoing backward arcs become forward if the other end vertex lies at a position  $> p$ .

Let  $\pi'$  be the linear ordering that is obtained after running *EstablishNesting* on  $\pi$ . Define the injective mapping  $\mu_h$  as follows: For every vertex  $v$ , order the outgoing forward arcs  $(v, x_0), (v, x_1), \dots, (v, x_{k-1})$  increasing in “length”, i. e., such that  $\pi'(x_0) \leq \pi'(x_1) \leq \dots \leq \pi'(x_{k-1})$ . Do the same for the incoming backward arcs  $(u_0, v), (u_1, v), \dots, (u_{s-1}, v)$ , with  $\pi'(u_0) \leq \pi'(u_1) \leq \dots \leq \pi'(u_{s-1})$ . Note that if  $v$  is incident to parallel arcs, the vertices  $x_i, x_j$  or  $u_i, u_j$  for  $i \neq j$  need not necessarily be distinct. We have  $s \leq k$ , otherwise moving  $v$  to position  $\pi'(u_{s-1})$  decreases the number of backward arcs incident to  $v$  by  $s - k$ . For each  $i, 0 \leq i < s$ , set  $\mu_h((u_i, v)) = (v, x_i)$ .

Consider the backward arc  $(u_i, v)$  with  $\mu_h((u_i, v)) = (v, x_i)$ . Suppose  $\pi'(x_i) > \pi'(u_i)$ . Then, moving  $v$  to position  $\pi'(u_i)$  would turn  $i$  backward arcs into forward arcs, but at most  $i - 1$  forward arcs would become backward. This contradicts the assumption that there is no position for  $v$  in  $\pi'$  that decreases the number of incident backward arcs further.

The injective mapping  $\mu_t$  can be obtained by processing the outgoing backward arcs and the incoming forward arcs for each vertex, again both increasing in length. The argument can be applied likewise.

In particular, this guarantees the Nesting Property for every optimal ordering  $\pi^*$ , because by definition, no ordering with strictly less backward arcs exists.  $\square$

A similar observation was independently made by [Hel64].

Observe that  $\mu_h[\mathcal{B}]$  and  $\mu_t[\mathcal{B}]$  need not be disjoint. For an example, consider the graph depicted in Figure 4.2. Setting  $\mu_h((v_4, v_1)) = (v_1, v_3)$  as well as  $\mu_t((v_3, v_0)) = (v_1, v_3)$  meets all requirements given in Lemma 4.3.

Let  $\text{Nest}(\pi)$  be the predicate that indicates whether  $\pi$  respects the Nesting Property. By Lemma 4.3 follows:

---



---

**Corollary 4.2**

For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \text{Nest}(\pi)$ .

---

### 4.3.3 A Graph's Excess

**Lemma 4.3** backs up a quite intuitive relationship between the cardinalities of  $\mathcal{F}^-(v)$ ,  $\mathcal{F}^+(v)$ ,  $\mathcal{B}^-(v)$ , and  $\mathcal{B}^+(v)$  of a vertex  $v$ , as the defined mappings  $\mu_h, \mu_t$  are injective:

---



---

**Corollary 4.3**

Let  $\pi$  be a linear ordering of a graph  $G = (V, A)$  that respects the Nesting Property. Then,

$$\forall v \in V : b^-(v) \leq f^+(v) \wedge b^+(v) \leq f^-(v).$$


---

With [Corollary 4.2](#), we immediately have:

---



---

**Corollary 4.4**

Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V, A)$ . Then,

$$\forall v \in V : b^-(v) \leq f^+(v) \wedge b^+(v) \leq f^-(v).$$


---

Furthermore, we can deduce:

---



---

**Corollary 4.5**

Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V, A)$ . Then,

$$\forall v \in V : b(v) \leq \min \{d^-(v), d^+(v)\}.$$


---

*Proof.* By [Corollary 4.4](#), we have

$$b^-(v) + b^+(v) \leq b^-(v) + f^-(v) = d^-(v),$$

$$b^-(v) + b^+(v) \leq f^+(v) + b^+(v) = d^+(v).$$

Hence,  $b(v) = b^-(v) + b^+(v) \leq \min \{d^-(v), d^+(v)\}$ . □

Although [Corollary 4.4](#) essentially only implies that there are at most as many backward arcs as forward arcs and therefore a weak bound of  $\frac{m}{2}$  for  $|\pi^*|$ , it helps in strengthening it for certain graphs. To this means, we define the *excess*  $exc(G)$  of a graph  $G = (V, A)$  as follows:

$$exc(G) = \sum_{v \in V} \max\{0, \delta(v)\}$$

---



---

**Lemma 4.4**

Let  $\pi^*$  be an optimal linear ordering of a graph  $G$ . Then,

$$|\pi^*| \leq \frac{m}{2} - \frac{exc(G)}{2}.$$


---

*Proof.* Let  $\pi$  be a linear ordering that respects the Nesting Property. Recall that  $\delta(v)$  is defined as  $d^+(v) - d^-(v)$ , which in turn equals  $f^+(v) + b^+(v) - f^-(v) - b^-(v)$ . By rearranging the terms, we obtain

$$\delta(v) = (f^+(v) - b^-(v)) - (f^-(v) - b^+(v)).$$

Define  $x(v) = f^+(v) - b^-(v)$  and  $y(v) = f^-(v) - b^+(v)$ , such that

$$\delta(v) = x(v) - y(v).$$

By [Corollary 4.4](#), both  $y(v) \geq 0$  and  $x(v) \geq 0$ , so we derive from the former that  $\delta(v) \leq x(v)$ , and from the latter,

$$\max\{0, \delta(v)\} \leq x(v).$$

We now associate each forward arc with its tail and each backward arc with its head in order to ensure that every arc is counted exactly once. Then, every vertex  $v$  is accountable for  $f^+(v)$  forward arcs and  $b^-(v)$  backward arcs. In consequence,

$$\begin{aligned} m &= \sum_{v \in V} (f^+(v) + b^-(v)) \\ &= \sum_{v \in V} (2 \cdot b^-(v) + x(v)) \\ &= \sum_{v \in V} (2 \cdot b^-(v)) + \sum_{v \in V} x(v) \\ &\geq \sum_{v \in V} (2 \cdot b^-(v)) + exc(G), \end{aligned}$$

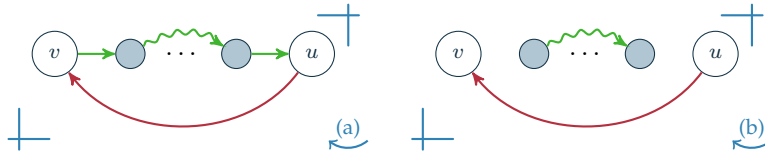


Figure 4.3: A backward arc  $b = (u, v)$  along with a forward path  $P_b = v \rightsquigarrow u$  (a), and the corresponding cropped forward path  $P_{b_{\text{crop}}}$  (b).

because  $x(v) \geq \max\{0, \delta(v)\}$ . With  $|\pi| = \sum_{v \in V} b^-(v)$ ,

$$m \geq 2 \cdot |\pi| + \text{exc}(G),$$

hence,

$$|\pi| \leq \frac{m}{2} - \frac{\text{exc}(G)}{2}.$$

With [Corollary 4.2](#), the claim follows.  $\square$

As it is evident from the definition of the excess, the strength of [Lemma 4.4](#) depends highly on the imbalance between the in- and outdegrees of a graph's vertices.

## 4.4 Path Property

The second property introduces the vital concept of a backward arc's forward path, which provides the basis for the so-called forward path graph used in [Chapter 5](#). Furthermore, it characterizes a linear ordering that induces a minimal feedback arc set.

### 4.4.1 Forward Paths for Backward Arcs

In [Chapter 3](#), a forward path has been introduced as a path that consists only of forward arcs. As a first step, we extend this definition as follows:

Let  $b = (u, v)$  be a backward arc according to some linear ordering  $\pi$ . A path  $P_b$  is a *forward path for b*, if it consists only of forward arcs and  $P_b = v \rightsquigarrow u$ , i. e.,  $P_b$  starts at  $b$ 's head and ends at  $b$ 's tail. [Figure 4.3\(a\)](#) provides a schematic drawing of a forward path for a backward arc. The cropped version of a forward path, which is shown in [Figure 4.3\(b\)](#), will only be introduced in [Section 4.5](#). As we only consider graphs without anti-parallel arcs, every forward path for a backward arc has length at least two.

We obtain the following property<sup>1</sup>:

<sup>1</sup>This result has also been published in an earlier conference article [[HBA13](#)].

**Lemma 4.5**

## PATH PROPERTY

Let  $\mathcal{B}$  be the set of backward arcs according to an optimal linear ordering  $\pi^*$ . For every backward arc  $b = (u, v) \in \mathcal{B}$ , there is a forward path  $P_b = v \rightsquigarrow u$ .

*Proof.* Let  $\pi^*$  be an optimal linear ordering of a graph  $G$ . Suppose there is a backward arc  $b = (u, v)$  in  $\pi^*$  that has no forward path  $P_b$ . Consider the feedback arc set  $\mathcal{B}$  induced by  $\pi^*$ . By assumption,  $\mathcal{B}$  is optimal. Remove  $b$  and all its parallel arcs from  $\mathcal{B}$  and insert it into  $\mathcal{F}$ . As there is no forward path  $v \rightsquigarrow u$ ,  $G|_{\mathcal{F}}$  remains acyclic.

Subsequently,  $\mathcal{B} \setminus [b]_{\parallel}$  is feasible, a contradiction to the optimality of  $\mathcal{B}$  and therefore also to the optimality of  $\pi^*$ .  $\square$

Note that [Lemma 4.5](#) does not require the forward paths of two backward arcs to be arc-disjoint. In particular, this also applies to parallel arcs. All parallel arcs of a backward arc may, e. g., be associated with the same forward path here, and in case of parallel forward arcs, all forward paths can use the same copy. Subsequently, we may neglect parallel arcs with respect to the Path Property and only have to ensure that they are classified alike.

Let  $\text{Path}(\pi)$  be a predicate such that  $\text{Path}(\pi) = \text{true}$  if and only if  $\pi$  respects the Path Property.

**Corollary 4.6**

For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \text{Path}(\pi)$ .

**4.4.2 Establishing Forward Paths**

The proof of [Lemma 4.5](#) already suggests how the Path Property can be established for a linear ordering. This is also the approach of the procedure *EnforceForwardPaths*( $G, \pi$ ) shown in [Algorithm 4.6](#).

As explained above, parallel arcs can largely be ignored with respect to the Path Property. Hence, we first create the *simple subgraph*  $G' \subseteq G$  which retains only one of multiple parallel arcs. More formally, if  $G = (V, A)$  with  $A = (U, m)$ , then  $G' = (V, A')$  with  $A' = (U, m')$  and  $m' : U \rightarrow \{1\}$  is the constant-one function. The graph  $G'$  is computed in [line 2](#) by the routine *SimplifyGraph*( $G$ ). We denote the sets of backward and forward arcs corresponding to  $G'$  by  $\mathcal{B}'$  and  $\mathcal{F}'$ , respectively.

**Algorithm 4.6** Path Property**Require:** graph  $G$ , linear ordering  $\pi$ **Return:**  $\pi$  if  $\text{Path}(\pi)$ , otherwise an improved linear ordering  $\pi'$ 1: **procedure** *EnforceForwardPaths*( $G, \pi$ )2:  $G' \leftarrow \text{SimplifyGraph}(G)$   $\triangleright G'$  retains only one of multiple parallel arcs3:  $\pi, \mathcal{B}', \mathcal{F}' \leftarrow \text{ComputePositionsAndArcSets}(G', \pi)$ 4:  $T \leftarrow \text{TransitiveClosure}(G'|_{\mathcal{F}'})$ 5: **for all**  $b \in \mathcal{B}'$  **do**6:     **if**  $\text{head}(b) \rightsquigarrow \text{tail}(b) \notin T$  **then**7:          $\mathcal{F}' \leftarrow \mathcal{F}' \cup \{b\}$ 8:          $\pi' \leftarrow \text{TopSort}(G'|_{\mathcal{F}'})$ 9:         **return**  $\pi'$ 10: **return**  $\pi$ 

For reasons of efficiency, the transitive closure  $T$  of  $G|_{\mathcal{F}'}$  is obtained in the next step. To this end, the algorithm employs a routine *TransitiveClosure*( $G$ ), which computes and stores for each ordered pair of vertices  $u, v$  whether there is a path  $u \rightsquigarrow v$  in  $G$ . The routine is called with argument  $G|_{\mathcal{F}'}$  and the result stored in a variable  $T$ . To test whether a backward arc  $b$  has a forward path then is only a query to  $T$ . If the check is negative for a backward arc  $b$ ,  $b$  is added to the set of forward arcs and *TopSort*( $G|_{\mathcal{F}'}$ ) yields a topological sorting of the acyclic subgraph, which is used as the new linear ordering. This implicitly also guarantees that parallel arcs in  $G$  receive the same classification.

Let  $\omega$  be the exponent in the running time of fast matrix multiplication algorithms.

---



---

**Lemma 4.6**

*EnforceForwardPaths*( $G, \pi$ ) runs in time  $\mathcal{O}(\min\{n \cdot m, n^\omega\})$ .

---

*Proof.* Consider the procedure *EnforceForwardPaths* in [Algorithm 4.6](#). Due to the blockwise representation of parallel arcs (cf. [Section 4.2](#)), *SimplifyGraph* can construct the simple subgraph  $G'$  in  $\mathcal{O}(n + m') = \mathcal{O}(m')$  steps, where  $m' = |A'|$  denotes the number of arcs in  $G' = (V, A')$ . Second, the routine *ComputePositionsAndArcSets* is called, which also has a running time of  $\mathcal{O}(m')$  by [Lemma 4.1](#).

Next, in [line 4](#), the transitive closure  $T$  of  $G|_{\mathcal{F}'}$  is obtained. This can be accomplished, e. g., by a depth-first search or a breadth-first search starting from every vertex of  $G'$

and using only forward arcs. Every depth-first search or breadth-first search requires  $\mathcal{O}(m')$  steps. Subsequently, the transitive closure can be computed in time  $\mathcal{O}(n \cdot m')$ .

Alternatively, the transitive closure of a graph can be obtained via matrix multiplication in time  $\mathcal{O}(n^\omega)$ , where  $\omega$  equals the exponent in the running time of fast matrix multiplication algorithms [Ski08]. As multiplying two  $n \times n$  matrices requires to consider all entries of the matrices, it has an asymptotic lower bound of  $\Omega(n^2)$ . Consequently,  $\omega \geq 2$ , which in turn implies that this approach yields a better running time only for sufficiently dense graphs.

Having already constructed  $T$ , testing whether a backward arc has a forward path in [line 6](#) then requires only constant time. The number of backward arcs is in  $\mathcal{O}(m')$ . Finally, the time needed to compute a topological sorting in [line 8](#) if the check fails is in  $\mathcal{O}(m')$  as mentioned in [Chapter 3](#). Note that the procedure terminates in this case immediately afterwards. Hence, a topological sorting is computed at most once during the execution of *EnforceForwardPaths*.

To sum up, we have a running time of  $\mathcal{O}(m') + \mathcal{O}(\min\{n \cdot m', n^\omega\}) + \mathcal{O}(m')$  for the initialization, the computation of the transitive closure, and the check of all backward arcs. If necessary, we need additionally  $\mathcal{O}(m')$  steps to compute the improved linear ordering at the very end. As  $G'$  is simple,  $m' \in \mathcal{O}(n^2)$ . Furthermore,  $\omega \geq 2$ , which implies  $m' \in \mathcal{O}(n^\omega)$ . Consequently, the running time of *EnforceForwardPaths* is in  $\mathcal{O}(\min\{n \cdot m', n^\omega\})$ . Observing that  $m' \leq m$  finally yields  $\mathcal{O}(\min\{n \cdot m, n^\omega\})$ .  $\square$

At present, the best known value for  $\omega$  is 2.3728639 due to Le Gall [LG14].

By a combination of *EnforceForwardPaths* and *Iterate*, we obtain an algorithm that repeats these steps until every backward arc has a forward path. We refer to this algorithm as *EstablishForwardPaths*( $G, \pi$ ), i. e.,  $\text{EstablishForwardPaths}(G, \pi) := \text{Iterate}(G, \pi, \text{EnforceForwardPaths})$ .

Applying [Proposition 4.1](#) to [Lemma 4.6](#) then yields:

---

**Corollary 4.7**

*EstablishForwardPaths*( $G, \pi$ ) runs in time  $\mathcal{O}(m \cdot \min\{n \cdot m, n^\omega\})$ .

---

#### 4.4.3 Minimal Feedback Arc Sets

*EstablishForwardPaths* can be regarded as a minimization procedure for the set of backward arcs that is associated with the linear ordering that it receives as input: After



it has terminated, no backward arc can be removed from the feedback arc set without destroying feasibility. Consequently, we obtain the following characterization<sup>1</sup>:

---



---

**Corollary 4.8**

The feedback arc set  $\mathcal{B}$  induced by a linear ordering  $\pi$  is minimal if and only if  $\pi$  respects the Path Property.

---

Recall from [Chapter 3](#) that for an arbitrary feedback arc set  $\mathcal{B}$ , a topological sorting of the corresponding acyclic subgraph  $G|_{\mathcal{F}}$  yields a linear ordering whose induced set of backward arcs  $\mathcal{B}'$  is only a subset of  $\mathcal{B}$ . With the Path Property we obtain:

---



---

**Theorem 4.3**

Let  $\pi$  be a linear ordering of a graph  $G$ . Every topological sorting of  $G|_{\mathcal{F}}$  yields a set of backward arcs  $\mathcal{B}' = \mathcal{B}_\pi$  if and only if  $\pi$  fulfills the Path Property.

---

*Proof.* Let  $\pi$  be a linear ordering of a graph  $G$  and let  $\xi$  be a topological sorting of  $G|_{\mathcal{F}}$ , which we also interpret as a linear ordering of  $G$ . If an arc  $b$  is backward with respect to  $\xi$ , then  $b \notin \mathcal{F}$ , i. e., it must also be backward with respect to  $\pi$ . This implies  $\mathcal{B}_\xi \subseteq \mathcal{B}_\pi$ .

Let us now assume that  $\pi$  respects the Path Property. Suppose  $\mathcal{B}_\xi \subsetneq \mathcal{B}_\pi$  and let  $b$  be an arc with  $b \in \mathcal{B}_\pi$ , but  $b \notin \mathcal{B}_\xi$ . Then,  $\xi$  witnesses that  $\mathcal{B}_\pi \setminus \{b\}$  is feasible, a contradiction to [Corollary 4.8](#), which states that  $\mathcal{B}_\pi$  is minimal.

For the converse, assume that  $\pi$  does not respect the Path Property, i. e., there is an arc  $b \in \mathcal{B}_\pi$  that has no forward path  $P_b$  in  $G|_{\mathcal{F}}$ . Let  $\mathcal{F}'_\pi = \mathcal{F}_\pi \cup \{b\}$ . Then,  $G|_{\mathcal{F}'_\pi}$  is acyclic (cf. proof of [Lemma 4.5](#)) and  $G|_{\mathcal{F}_\pi} \subsetneq G|_{\mathcal{F}'_\pi}$ . Compute a topological sorting  $\xi'$  of  $G|_{\mathcal{F}'_\pi}$ , which is also a topological sorting of its subgraph  $G|_{\mathcal{F}_\pi}$ . By construction,  $b \notin \mathcal{B}_{\xi'}$ , but  $b \in \mathcal{B}_\pi$ . Hence, there is at least one topological sorting  $\xi'$  of  $G|_{\mathcal{F}_\pi}$  such that  $\mathcal{B}_{\xi'} \neq \mathcal{B}_\pi$ .  $\square$

## 4.5 Blocking Vertices Property

The Path Property introduced in [Section 4.4](#) requires for every backward arc  $b = (u, v)$  a forward path  $P_b = v \rightsquigarrow u$ . In this section, we show that on the basis of their layout, certain vertices can be excluded from being part of such a path.

---

<sup>1</sup>This result has also been published in an earlier conference article [[HBA13](#)].

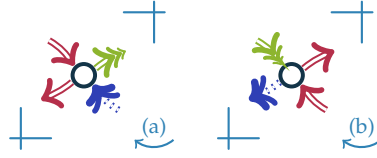




Figure 4.4: Pictograms for a left-blocking vertex (a) and a right-blocking vertex (b).

#### 4.5.1 Left- and Right-Blocking Vertices

We start by defining the term of a *cropped* forward path of a backward arc: Let  $\pi$  be a linear ordering of a graph  $G$  and let  $P_b = \langle x_0 = v, x_1, \dots, x_k = u \rangle$  be a forward path for a backward arc  $b = (u, v)$ . Then,  $P_{b_{\text{crop}}} = \langle x_1, \dots, x_{k-1} \rangle$  is called a cropped forward path for  $b$ . Descriptively,  $P_{b_{\text{crop}}}$  is the subpath of  $P_b$  obtained by removing its first and last vertex, i. e.,  $P_b$  without  $b$ 's head and tail. Figure 4.3(b) illustrates the definition once more. As a forward path for a backward arc has at least two arcs and three vertices, its cropped equivalent consists of at least one vertex.

Next, consider the layout of a vertex  $v$  that is induced by a linear ordering. We say that  $v$  is *left-blocking* if  $f^-(v), f^+(v), b^+(v) \geq 1$ , and  $f^-(v) = b^+(v)$ . Likewise,  $v$  is *right-blocking* if  $f^-(v), f^+(v), b^-(v) \geq 1$ , and  $f^+(v) = b^-(v)$ . The pictograms,  for left-blocking and  for right-blocking, illustrate the choice of naming: the blocking side is the “side” of the vertex where the number of incoming and outgoing arcs are equal. Figure 4.4 additionally shows enlarged versions of both pictograms. Observe that every vertex that is left-blocking in  $G$  according to a linear ordering  $\pi$  is right-blocking in  $G^R$  according to the linear ordering  $\pi^R$ . Likewise, every right-blocking vertex according to  $\pi$  in  $G$  is left-blocking according to  $\pi^R$  in  $G^R$ .

Note that for two distinct vertices  $v \neq v'$  in any linear ordering always holds that

$$\mathcal{F}^-(v) \cap \mathcal{F}^-(v') = \mathcal{F}^+(v) \cap \mathcal{F}^+(v') = \mathcal{B}^-(v) \cap \mathcal{B}^-(v') = \mathcal{B}^+(v) \cap \mathcal{B}^+(v') = \emptyset.$$

In case of blocking vertices, this and the fact that an arc can either be forward or backward leads us to the following observation:

---

#### Proposition 4.2

For every set  $Z_l$  of left-blocking vertices,  $f^-[Z_l] = b^+[Z_l]$  and  $\mathcal{F}^-[Z_l] \cap \mathcal{B}^+[Z_l] = \emptyset$ . Likewise, for every set  $Z_r$  of right-blocking vertices,  $f^+[Z_r] = b^-[Z_r]$  and  $\mathcal{F}^+[Z_r] \cap \mathcal{B}^-[Z_r] = \emptyset$ .

---

Moreover, due to the equal number of incoming and outgoing arcs on their “blocking side”, left-blocking and right-blocking vertices show an interesting feature:

---

**Lemma 4.7**

Let  $\pi$  be a linear ordering of a graph  $G = (V, A)$  and let  $\mathcal{B} \subseteq A$  the set of backward arcs induced by  $\pi$ . If  $Z_l \subseteq V$  is a set of left-blocking vertices, then  $\mathcal{B}' = \mathcal{B} \setminus \mathcal{B}^+[Z_l] \cup \mathcal{F}^-[Z_l]$  is feasible and every path  $u \rightsquigarrow v$  that consists only of arcs in  $A \setminus \mathcal{B}'$  and contains a vertex in  $z \in Z_l$  must have a subpath  $u \rightsquigarrow z$  consisting only of vertices in  $Z_l$ . Likewise, if  $Z_r \subseteq V$  is a set of right-blocking vertices, then  $\mathcal{B}' = \mathcal{B} \setminus \mathcal{B}^-[Z_r] \cup \mathcal{F}^+[Z_r]$  is feasible and every path  $u \rightsquigarrow v$  that consists only of arcs in  $A \setminus \mathcal{B}'$  and contains a vertex in  $z \in Z_r$  must have a subpath  $z \rightsquigarrow v$  consisting only of vertices in  $Z_r$ . Furthermore,  $|\mathcal{B}| = |\mathcal{B}'|$  in both cases.

---

*Proof.* Consider first the case that  $Z_l \subseteq V$  is a set of left-blocking vertices with respect to  $\pi$ . Let  $\mathcal{F} = A \setminus \mathcal{B}$  denote the set of forward arcs induced by  $\pi$ . Note that  $\mathcal{B}^+[Z_l] \cap \mathcal{B}^-[Z_l]$  need not necessarily be empty: There may be two vertices  $z, z' \in Z_l$  such that  $(z', z) \in \mathcal{B}$ , so  $(z', z) \in \mathcal{B}^-(z)$  and  $(z', z) \in \mathcal{B}^+(z')$ . Due to  $G$  being free of loops by [Assumption 3.1](#),  $z \neq z'$ , so  $\pi(z) < \pi(z')$ .

Consider a path  $P = u \rightsquigarrow v$  that uses only arcs in

$$\mathcal{F}' = A \setminus \mathcal{B}' = \mathcal{F} \setminus \mathcal{F}^-[Z_l] \cup \mathcal{B}^+[Z_l].$$

and contains at least one vertex  $z \in Z_l$ . If  $z$  is not the first vertex on  $P$ , i. e.,  $z \neq u$ ,  $P$  must contain an incoming arc  $a \in \mathcal{F}'$  of  $z$ . As  $\mathcal{F}^-(z) \subseteq \mathcal{F}^-[Z_l]$  and  $\mathcal{F}' \cap \mathcal{F}^-[Z_l] = \emptyset$ ,  $a \notin \mathcal{F}^-(z)$ . Thus,  $a \in \mathcal{B}^-(z) \cap \mathcal{F}'$ . This implies that there is a vertex  $z' \in Z_l$  such that  $a \in \mathcal{B}^+(z')$ , i. e.,  $a = (z', z)$ , and  $P$  contains  $z'$ . Furthermore,  $\pi(z) < \pi(z')$ , because  $a$  is backward. If also  $z' \neq u$ ,  $P$  must contain an incoming arc  $a' \in \mathcal{F}'$  of  $z'$ . For the same reason as in case of  $z$ , there is hence a vertex  $z'' \in Z_l$  such that  $a' = (z'', z')$  and  $\pi(z') < \pi(z'')$ . We can repeat this argument until we reach the first vertex of  $P$ ,  $u$ . Thus, every vertex preceding  $z$  in  $P$  is in  $Z_l$ , which implies that  $P$  has a subpath  $u \rightsquigarrow z$  consisting only of arcs in  $Z_l$ .

Suppose that

$$\mathcal{B}' = \mathcal{B} \setminus \mathcal{B}^+[Z_l] \cup \mathcal{F}^-[Z_l]$$

is not feasible. Then, there is a cycle  $C$  consisting only of arcs in  $\mathcal{F}'$ . In case that  $C$  is not simple,  $G|_{\mathcal{F}'}$  also contains a simple cycle consisting of the shortest subpath of  $C$  whose first and last vertex are identical (cf. [Section 3.2](#)). We therefore readily assume that  $C$  is simple. As  $G|_{\mathcal{F}}$  is acyclic,  $C$  must contain at least one arc of  $\mathcal{F}' \setminus \mathcal{F} = \mathcal{B}^+[Z_l]$ .

Let  $k \geq 1$  and  $Y = \{(z_i, h_i) \mid 0 \leq i < k\} \subseteq \mathcal{B}^+[Z_l]$  be the set of arcs in  $C$  that are also in  $\mathcal{B}^+[Z_l]$ . Then,  $C$  consists of  $Y$  and a set of paths  $\mathcal{P} = \{h_i \rightsquigarrow z_j \mid 0 \leq i, j < k\}$  that use only arcs in  $\mathcal{F}'$  and  $z_j \in Z_l$  for all  $0 \leq j < k$ . As we have shown above, this implies that all paths of  $\mathcal{P}$  consist only of vertices in  $Z_l$ . In consequence, all vertices of  $C$  are in  $Z_l$  and all arcs of  $C$  are backward arcs with respect to  $\pi$ , a contradiction to  $\mathcal{B}$  being acyclic (cf. Section 3.3). Hence,  $\mathcal{B}'$  is feasible. Furthermore, Proposition 4.2 implies that  $\mathcal{B}^+[Z_l] = \mathcal{F}^-[Z_l]$  and  $\mathcal{B}^+[Z_l] \cap \mathcal{F}^-[Z_l] = \emptyset$ . We can therefore conclude that  $|\mathcal{B}| = |\mathcal{B}'|$ .

For the proof concerning a set of right-blocking vertices  $Z_r$ , the same arguments as above are applicable. Alternatively, we may consider the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^R$ . In  $G^R$ , the vertices in  $Z_r$  are left-blocking and the statement follows immediately.  $\square$

With this in mind, we can now turn to the Blocking Vertices Property. In its definition, also the reason why these vertices are called “blocking” becomes apparent:

---



---

**Lemma 4.8**

BLOCKING VERTICES PROPERTY

Let  $\pi^*$  be an optimal linear ordering of a graph  $G$ . For every backward arc  $b \in \mathcal{B}$  induced by  $\pi^*$  there is a cropped forward path that does not contain a left-blocking vertex and there is a cropped forward path that does not contain a right-blocking vertex.

---

*Proof.* We first show that every backward arc has a forward path without left-blocking vertices. Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V, A)$  and let  $\mathcal{B}, \mathcal{F}$  be the respective sets of backward and forward arcs. Suppose, for the sake of contradiction, that there is a backward arc  $b = (u, v) \in \mathcal{B}$  such that all cropped forward paths of  $b$  contain at least one left-blocking vertex. If applicable, the same immediately holds for all parallel arcs of  $b$ . In consequence of the Path Property, there is at least one forward path for  $b$ . Construct the set  $Z \subset V$  as follows: For every cropped forward path of  $b$ , place the first vertex that occurs on a traversal of the path and is left-blocking in  $Z$ . Note that this implies in particular that  $v \notin Z$ . By Lemma 4.7,

$$\mathcal{B}' = \mathcal{B} \setminus \mathcal{B}^+[Z] \cup \mathcal{F}^-[Z]$$

is feasible and  $|\mathcal{B}'| = |\mathcal{B}|$ . Let

$$\mathcal{F}' = \mathcal{F} \setminus \mathcal{B}' = \mathcal{F} \setminus \mathcal{F}^-[Z] \cup \mathcal{B}^+[Z].$$



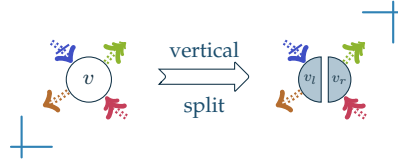


Figure 4.6: Vertical split operation on vertex  $v$ .

Note that the Blocking Vertices Property does not—and cannot—guarantee the existence of a forward path for a backward arc that contains neither left- nor right-blocking vertices: Let  $Z_l$  and  $Z_r$  denote the sets of left- and right-blocking vertices, respectively, as they are constructed in the proof of Lemma 4.8. Then,  $\mathcal{B}^+[Z_l] \cap \mathcal{B}^-[Z_r]$  may be non-empty and, hence, the set  $\mathcal{B}'$  in the proof may be equal in cardinality or even greater than  $\mathcal{B}$ . Figure 4.5 provides further evidence by depicting a backward arc in an optimal linear ordering that either has a forward path via a left-blocking vertex or a forward path via a right-blocking vertex, but none else. The optimality has been verified by an exact algorithm.

As before, we define a predicate  $\text{NoBlock}(\pi)$  that expresses whether a linear ordering  $\pi$  respects the Blocking Vertices Property and obtain:

---



---

**Corollary 4.9**

For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \text{NoBlock}(\pi)$ .

---

#### 4.5.2 Vertical Splits

The Blocking Vertices Property is stated in Lemma 4.8 as an extension, or more precisely, restriction of the Path Property. Alternatively, the graph can be modified such that the Blocking Vertices Property can be traced back to the plain Path Property on the modified graph.

A *vertical split* of a vertex  $v$  is a graph operation that replaces  $v$  by two vertices  $v_l, v_r$  such that  $v_l$  inherits all arcs in  $\mathcal{F}^-(v) \cup \mathcal{B}^+(v)$  and  $v_r$  inherits all arcs in  $\mathcal{F}^+(v) \cup \mathcal{B}^-(v)$ . In effect,  $v_l$  is a pseudosink and  $v_r$  is a pseudosource. Figure 4.6 gives an illustration of this modification. For ease of handling, we identify the arcs incident to  $v$  before and after the split, i. e., if  $a = (v, u)$  is an arc with  $a \in \mathcal{F}^+(v)$  before the split, then  $a = (v_r, u)$  with  $a \in \mathcal{F}^+(v_r)$  after the split. This holds likewise for arcs in  $\mathcal{B}^-(v)$  as well as for arcs in  $\mathcal{F}^-(v)$  and  $\mathcal{B}^+(v)$ , which are inherited by  $v_l$ .

---

<sup>1</sup>Verified by an exact algorithm.

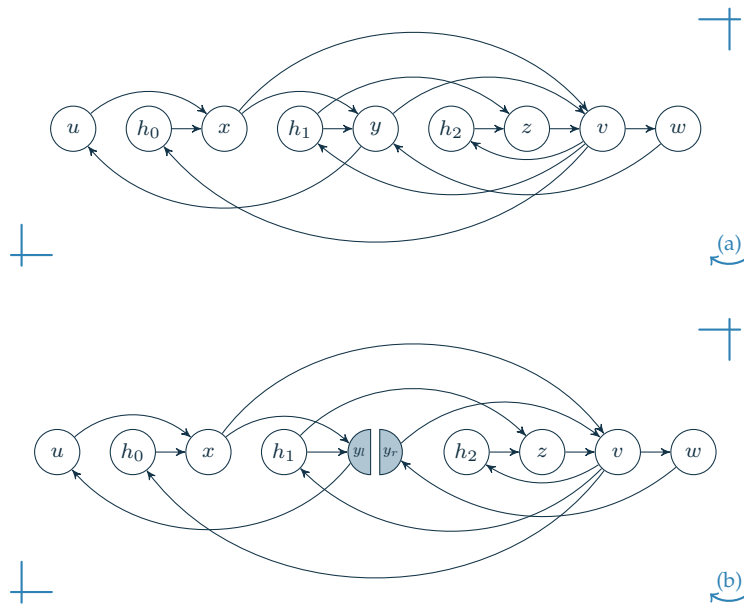


Figure 4.7: A linear ordering of a graph with right-blocking vertex  $y$  (a) and the corresponding linear ordering of its right-blocking split graph (b).

For a graph  $G$  with linear ordering  $\pi$ , we obtain the *left-blocking split graph*  $G_{sp/l}$  along with a linear ordering  $\pi_{sp/l}$  by applying a vertical split to every left-blocking vertex and keeping every other vertex as it is.  $\pi_{sp/l}$  is constructed from  $\pi$  such that the split vertices  $v_l, v_r$  of  $v$  in  $\pi$  and  $\pi_{sp/l}(v_l) < \pi_{sp/l}(v_r)$ . Likewise, we obtain the *right-blocking split graph*  $G_{sp/r}$  with linear ordering  $\pi_{sp/r}$  by applying a vertical split to every right-blocking vertex and keeping every other vertex as it is. Figure 4.7 shows the linear ordering of a graph and its right-blocking split graph with corresponding linear ordering.

By making oneself aware of the fact that these splits exactly inhibit forward paths to pass through, but not start or end at the split vertices, we obtain:

---

**Corollary 4.10**

Let  $\pi$  be a linear ordering of a graph  $G$ . Then,  $\pi$  respects the Blocking Vertices Property if and only if both  $\pi_{sp/l}$  and  $\pi_{sp/r}$  respect the Path Property.

---

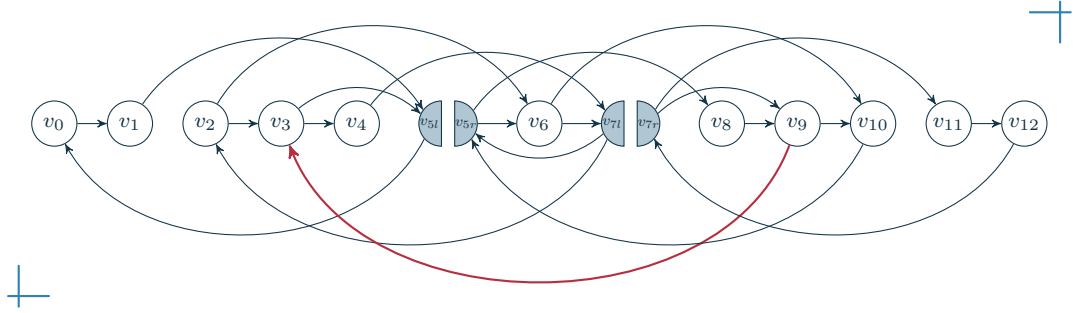


Figure 4.8: The linear ordering obtained from the optimal<sup>2</sup> linear ordering shown in Figure 4.5 by splitting both left- and right-blocking vertices. There is no forward path for the backward arc  $(v_9, v_3)$ .

In consequence, we can reformulate Lemma 4.8 to<sup>1</sup>:

---

**Corollary 4.11**

Let  $\pi^*$  be an optimal linear ordering of a graph  $G$ . Then,  $\pi_{\text{sp}/l}^*$  and  $\pi_{\text{sp}/r}^*$  respect the Path Property.

---

Note that in parallel to Lemma 4.8, Corollary 4.11 does not apply if both left- and right-blocking vertices are split simultaneously. Figure 4.8 demonstrates this by showing the linear ordering obtained from the optimal<sup>2</sup> linear ordering given in Figure 4.5 by splitting both left- and right-blocking vertices.

### 4.5.3 Establishing Non-Blocking Forward Paths

Finally, we want to deal with the algorithmic aspect of the Blocking Vertices Property. Let  $\text{SplitVertically}(G, \pi, X)$  be a function that returns the graph  $G_{\text{sp}}$  that is obtained from  $G$  by applying a vertical split to all vertices in  $X \subseteq V$ . Along with  $G_{\text{sp}}$ , it returns the corresponding linear ordering  $\pi_{\text{sp}}$ .

Using this routine, the Blocking Vertices Property can be established efficiently by means of the two procedures shown in Algorithm 4.7. *EnforceNoBlockLeft* enforces the left-sided Blocking Vertices Property, i. e., it only considers left-blocking vertices. Like all property-enforcing algorithms before, it receives a graph  $G$  and a linear ordering  $\pi$  as input. Due to Corollary 4.11, the Blocking Vertices Property can be reduced to the Path

<sup>1</sup>A significantly weaker version of this result has also been published in an earlier conference article [HBA13].

<sup>2</sup>Verified by an exact algorithm.



**Algorithm 4.7** Blocking Vertices Property**Require:** graph  $G$ , linear ordering  $\pi$ **Return:**  $\pi$  if NoBlock( $\pi$ ), otherwise an improved linear ordering  $\pi'$ 


---

```

1: procedure EnforceNoBlocking( $G, \pi$ )
2:    $\pi' \leftarrow$  EnforceNoBlockLeft( $G, \pi$ )
3:   if  $\pi' \neq \pi$  then return  $\pi'$  else return EnforceNoBlockLeft( $G^R, \pi^R$ )R
4: procedure EnforceNoBlockLeft( $G, \pi$ )
5:    $\pi, \mathcal{B}, \mathcal{F} \leftarrow$  ComputePositionsAndArcSets( $G, \pi$ )
6:    $Q \leftarrow$  set of left-blocking vertices in  $G$  according to  $\pi$ 
7:    $G_{\text{sp}/l}, \pi_{\text{sp}/l} \leftarrow$  SplitVertically( $G, \pi, Q$ )
8:    $G' \leftarrow$  SimplifyGraph( $G_{\text{sp}/l}$ )
9:    $\pi_{\text{sp}/l}, \mathcal{B}', \mathcal{F}' \leftarrow$  ComputePositionsAndArcSets( $G', \pi_{\text{sp}/l}$ )
10:   $T \leftarrow$  TransitiveClosure( $G'|_{\mathcal{F}'}$ )
11:  for all  $b \in \mathcal{B}'$  do
12:    if  $\text{head}(b) \rightsquigarrow \text{tail}(b) \notin T$  then
13:       $Z \subseteq Q$  as in the proof of Lemma 4.8
14:       $\mathcal{F}'' \leftarrow (\mathcal{F} \setminus \mathcal{F}^-[Z]) \cup \mathcal{B}^+[Z] \cup [b]_{\parallel}$ 
15:       $\pi' \leftarrow$  TopSort( $G|_{\mathcal{F}''}$ )
16:    return  $\pi'$ 
17:  return  $\pi$ 

```

---

Property on the left-blocking split graph. The necessary steps to obtain the improved linear ordering efficiently, however, differ slightly from those in the Path Property, which is why *EnforceForwardPaths* is not reused here. Instead, the modification of the linear ordering is included directly.

In order to identify the set of left-blocking vertices, the algorithm needs to call *ComputePositionsAndArcSets* on the input graph. The set of left-blocking vertices as well as the left-blocking split graph  $G_{\text{sp}/l}$  and the corresponding linear ordering  $\pi_{\text{sp}/l}$  are obtained in [line 6](#) and [line 7](#). Next, just as in *EnforceForwardPaths*, the simple subgraph  $G'$ , here of  $G_{\text{sp}/l}$ , is constructed ([line 8](#)). As we also need to access the set of forward and backward arcs in  $G'$ , the algorithm again calls *ComputePositionsAndArcSets* on  $G'$  and the linear ordering  $\pi_{\text{sp}/l}$ . The transitive closure  $T$  of the subgraph of  $G'$  induced by the set of forward arcs  $\mathcal{F}'$  is computed explicitly in [line 10](#). Afterwards, each backward arc is checked for a forward path. This task reduces again to querying  $T$  in [line 12](#). If negative, the improved set of forward arcs is constructed as shown in the proof of

**Lemma 4.8** and a new linear ordering  $\pi'$  is computed (lines 14–15). In this case, the routine cancels and returns  $\pi'$ . Otherwise, if the check for all backward arcs was positive, *EnforceNoBlockLeft* returns the unmodified original linear ordering  $\pi$ .

For the right-blocking vertices, *EnforceNoBlockLeft* is called with the reverse graph  $G^R$  and the reverse linear ordering  $\pi^R$ . For simplicity, we assume an implementation such that changes to  $\pi^R$  are immediately reflected in  $\pi$ . Eventually, *EnforceNoBlocking* combines both calls.

Let  $\omega$  be the exponent in the running time of fast matrix multiplication algorithms.

---



---

**Lemma 4.9**

*EnforceNoBlocking*( $G, \pi$ ) runs in time  $\mathcal{O}(\min\{n \cdot m, n^\omega\})$ .

---

*Proof.* The analysis of *EnforceNoBlockLeft* does not differ much from that of *EnforceForwardPaths* conducted in **Lemma 4.6**. For the identification of the left-blocking vertices, however, the algorithm needs to call *ComputePositionsAndArcSets* immediately on the input graph here. As a result of the blockwise representation of parallel arcs (cf. **Section 4.2**), *ComputePositionsAndArcSets* can be implemented nonetheless in time  $\mathcal{O}(m')$ , where  $m'$  again denotes the number of arcs in the simple subgraph of  $G$ . Computing the set of left-blocking vertices in **line 6** can then be achieved in time  $\mathcal{O}(n)$ , because only the length of the respective incidence lists at each vertex need to be compared. For the same reason as above and utilizing the fact that parallel arcs are always classified identically, the split graph and the corresponding linear ordering in **line 7** can be obtained in time  $\mathcal{O}(m')$ ,

Comparing the size of  $G_{\text{sp}/l}$  and  $\pi_{\text{sp}/l}$  to  $G$  and  $\pi$ , respectively, we find that  $G_{\text{sp}/l}$  and  $\pi_{\text{sp}/l}$  have at most twice as many vertices as  $G$  and  $\pi$ . The number of arcs remains the same. Consequently,  $n_{G_{\text{sp}/l}} \in \Theta(n_G)$  and  $m_{G_{\text{sp}/l}} = m_G$ . Therefore, we do not differentiate between  $n_G$  and  $n_{G_{\text{sp}/l}} = n_{G'}$  and simply use  $n$ . For the arcs, we write  $m = m_G = m_{G_{\text{sp}/l}}$  if we include parallel arcs, and  $m' = m_{G'}$  for the arcs of the simple subgraph  $G' \subseteq G_{\text{sp}/l}$ .

As in *EnforceForwardPaths*, *SimplifyGraph*( $G_{\text{sp}/l}$ ) can be implemented to run in time  $\mathcal{O}(m')$ , which equally applies to the second call to *ComputePositionsAndArcSets* in **line 9**. The transitive closure of the subgraph induced by the set of forward arcs in **line 10** takes time  $\mathcal{O}(\min\{n \cdot m', n^\omega\})$ , as has already been discussed in the proof of **Lemma 4.6**.

In the following, the algorithm loops over all arcs that are currently classified as backward, whose number is in  $\mathcal{O}(m')$ . For each backward arc  $b$ , the existence of a

forward path is tested in  $\mathcal{O}(1)$ . If necessary, the set  $Z$  of blocking vertices that occur first on any cropped forward path of  $b$  can be obtained by a breadth-first search in time in [line 13](#) in accordance with the definition in the proof of [Lemma 4.8](#). Afterwards, the set of forward arcs is recomputed in [line 14](#) and a new linear ordering obtained in [line 15](#). All of these statements can be executed in time  $\mathcal{O}(m')$  if parallel arcs are again treated blockwisely. Like in *EnforceForwardPaths*, these steps are carried out at most once during the execution of *EnforceNoBlockLeft*.

Consequently, we obtain the same running time for *EnforceNoBlockLeft* as for *EnforceForwardPaths*, which is  $\mathcal{O}(m') + \mathcal{O}(\min\{n \cdot m', n^\omega\})$ , and can be bounded from above by  $\mathcal{O}(\min\{n \cdot m, n^\omega\})$ . As *EnforceNoBlocking* consists of exactly two calls to *EnforceNoBlockLeft*, we obtain the same running time. See [Section 4.2.5](#) for the reason why we neglect the cost of comparing  $\pi$  to  $\pi'$  and of the reverse operations.  $\square$

We employ again *Iterate* to obtain an algorithm that establishes the Blocking Vertices Property and define *EstablishNoBlocking*( $G, \pi$ ) as *Iterate*( $G, \pi, \text{EnforceNoBlocking}$ ).

By [Lemma 4.9](#) and [Proposition 4.1](#) follows:

---



---

**Corollary 4.12**

*EstablishNoBlocking*( $G, \pi$ ) runs in time  $\mathcal{O}(m \cdot \min\{n \cdot m, n^\omega\})$ .

---

## 4.6 Multipath Property

With the introduction of the Blocking Vertices Property in the previous section, one possibility for tightening the Path Property has already been shown. The Multipath Property constitutes another one, which is aimed at arcs shared by forward paths.

### 4.6.1 Arc-Disjoint Forward Paths

We cannot assume in general that, even for an optimal linear ordering, it is possible to find a forward path for every arc such that all forward paths are pairwise arc-disjoint. For an example, consider the graph shown in [Figure 4.9](#). The linear ordering depicted here is known to be optimal<sup>1</sup>, yet the forward paths for the backward arcs  $(u_0, v_0)$  and  $(u_2, v_2)$  have arc  $(u_1, v_1)$  in common.

---

<sup>1</sup>see also [Section 5.2.1](#)

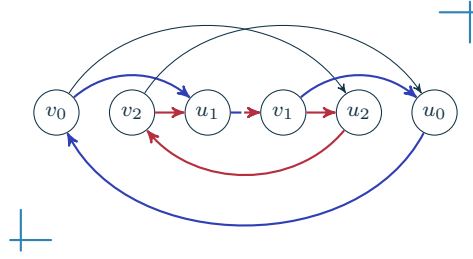


Figure 4.9: Two backward arcs  $(u_0, v_0)$  and  $(u_2, v_2)$  with non-disjoint forward paths.

On the other hand, there are clearly graphs where arc-disjointness of all or some of the forward paths is achievable. This raises a question: What is a sufficient condition for a set of backward arcs to have arc-disjoint forward paths in an optimal linear ordering? The following lemma provides one such requirement:

---

**Lemma 4.10** MULTIPATH PROPERTY  
 Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V, A)$ . For every vertex  $v \in V$ , there is a set of pairwise arc-disjoint forward paths that contains a distinct forward path for every  $b \in \mathcal{B}_{\pi^*}^+(v)$ .

---

*Proof.* We consider an arbitrary linear ordering  $\pi$  of a graph  $G = (V, A)$  that respects the Path Property and show that if it does not fulfill Lemma 4.10, then  $\pi$  is not optimal. Let  $\mathcal{B}, \mathcal{F}$  be the sets of backward and forward arcs according to  $\pi$  and  $v \in V$ . Note that  $\mathcal{B}(v) = \mathcal{B}^+(v) \cup \mathcal{B}^-(v)$  and a forward path for a backward arc  $b \in \mathcal{B}^+(v)$  can only consist of vertices at positions  $\leq \pi(v)$ , whereas a forward path for a backward arc  $b \in \mathcal{B}^-(v)$  can only consist of vertices at positions  $\geq \pi(v)$ . Hence, every forward path for a backward arc in  $\mathcal{B}^+(v)$  is arc-disjoint to every forward path for a backward arc in  $\mathcal{B}^-(v)$ .

We can regard the problem of finding disjoint forward paths for  $\mathcal{B}^+(v)$  as a maximum flow problem with unit capacities on the graph obtained from  $G|_{\mathcal{F}}$  by adding a new vertex  $s$  and unit-capacity arcs from  $s$  to every head  $h$  of a backward arc in  $\mathcal{B}^+(v)$ . Consider the maximum flow  $f$  from  $s$  to  $v$ . If  $f = b^+(v)$ , then there are arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^+(v)$ . Otherwise, we can find a minimum cut  $D$  that consists of  $f < b^+(v)$  arcs and separates  $s$  from  $v$ . See Figure 4.10(a) and Figure 4.10(b) for an illustration of this setup. Let  $X = D \cap \mathcal{F}$  and  $Y = \mathcal{B}^+(v) \setminus \{(v, h) \mid (s, h) \in D\}$ , i. e., we ignore arcs in the minimum cut that are incident to  $s$  and, at the same time, their respective backward arcs, so the fact that  $|D| < |\mathcal{B}^+(v)|$  implies that  $|X| < |Y|$ . Figure 4.10(c) visualizes  $X$  and  $Y$  once more. Let  $\mathcal{B}' = \mathcal{B} \setminus Y \cup X$  and

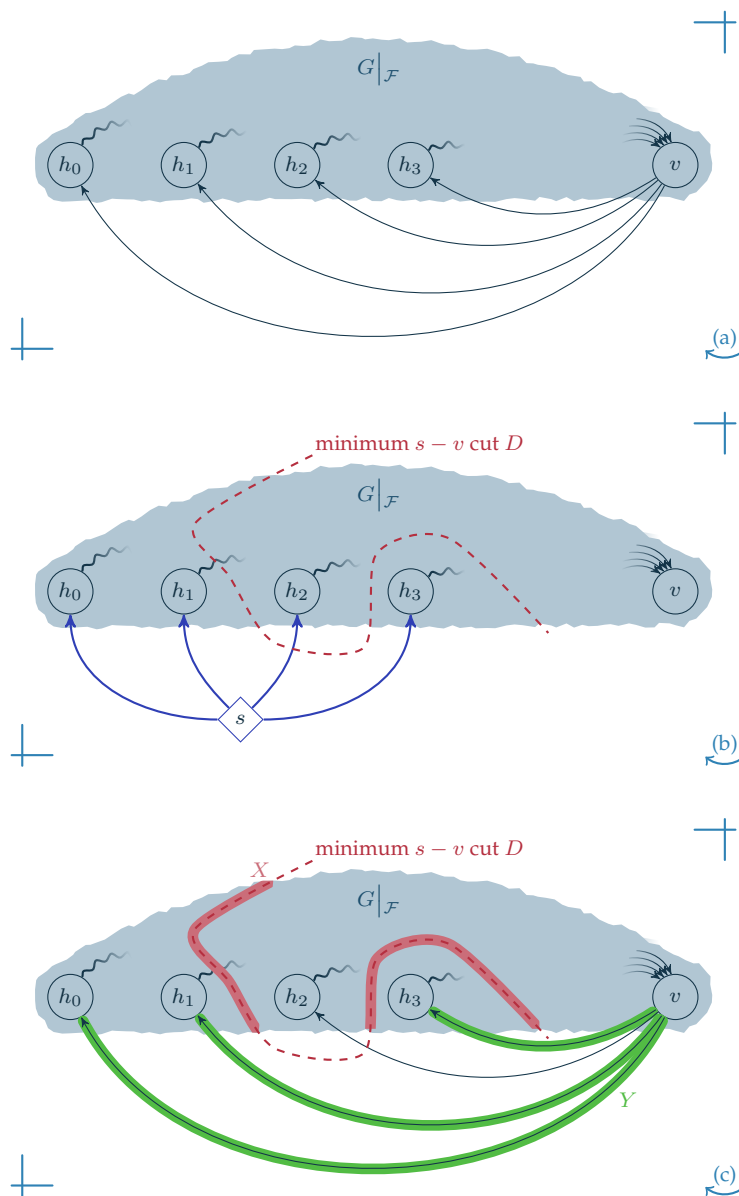


Figure 4.10: Proof of the Multipath Property:

A vertex  $v$  with outgoing backward arcs and their respective forward paths (a). Introduce a source vertex  $s$  with unit-capacity arcs to every head of an arc in  $\mathcal{B}^+(v)$  and consider the maximum  $s-v$  flow/minimum  $s-v$  cut (b). Finally, identify the sets  $X = D \cap \mathcal{F}$  (red) and  $Y = \mathcal{B}^+(v) \setminus \{(v, h) \mid (s, h) \in D\}$  (green) (c).

$\mathcal{F}' = \mathcal{F} \setminus X \cup Y$ . Suppose, for contradiction, that  $\mathcal{F}'$  contains a cycle  $C$ . As  $G|_{\mathcal{F}}$  is acyclic,  $C$  includes at least one arc  $b = (v, u) \in Y$ , so  $C$  passes through  $v$ . In order to close the cycle, there must be a path  $u \rightsquigarrow v$  in  $\mathcal{F}'$ . Observe that  $u \rightsquigarrow v$  may contain further arcs from  $Y$ . In this case,  $v$  appears more than once in  $u \rightsquigarrow v$ . Consider the subpath from  $u$  to the first occurrence of  $v$ , which is a forward path in  $\mathcal{F} \setminus X$ . As  $(v, u) \in Y$ ,  $(s, u)$  cannot have been part of the minimum cut, which implies that there also is a path from  $s$  to  $v$  via  $u$  in the maximum flow graph that is not covered by  $D$ , a contradiction. Hence,  $G|_{\mathcal{F}'}$  is acyclic, so  $\mathcal{B}'$  is feasible and with  $|\mathcal{B}'| < |\mathcal{B}|$ ,  $\pi$  cannot be optimal.

The proof for the existence of arc-disjoint forward paths for  $\mathcal{B}^-(v)$  follows by considering the outgoing backward arcs  $v$  in the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^R$  (cf. Section 3.3.3).  $\square$

As before, we introduce a predicate  $\text{MPath}(\pi)$  that expresses whether  $\pi$  respects the Multipath Property and obtain:

---



---

**Corollary 4.13**

For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \text{MPath}(\pi)$ .

---

**Lemma 4.10** immediately implies the “simple” Path Property and also the Nesting Property. For the latter, it suffices to set  $\mu_h((u, v))$  to the first arc of the forward path  $v \rightsquigarrow u$  that is determined by computing arc-disjoint forward paths for the incoming backward arcs at  $v$ , and  $\mu_t((u, v))$  analogously to the last arc of the forward path  $v \rightsquigarrow u$  that is determined by computing arc-disjoint forward paths for the outgoing backward arcs at  $u$ .

---



---

**Corollary 4.14**

For every linear ordering  $\pi$  holds:  $\text{MPath}(\pi) \Rightarrow \text{Nest}(\pi) \wedge \text{Path}(\pi)$ .

---

Note that for a backward arc  $(u, v)$ , the Multipath Property only demands a forward path  $P = v \rightsquigarrow u$  such that  $P$  is arc-disjoint with all other forward paths incident to  $v$ , and a possibly different forward path  $P' = v \rightsquigarrow u$  that is arc-disjoint with the forward paths of all backward arcs incident to  $u$ . This is no deficiency of the Multipath Property: The graph depicted in [Figure 4.11](#) testifies that these two forward paths cannot always be selected coincidentally, i. e., such that  $P = P'$ . Here, for the backward arc  $(v_8, v_0)$ , the forward path  $\langle v_0, v_3, v_7, v_8 \rangle$  is selected at  $v_0$ , while at  $v_8$ , the selected forward path is  $\langle v_0, v_1, v_8 \rangle$  and there is no possibility to find two that match each other without creating

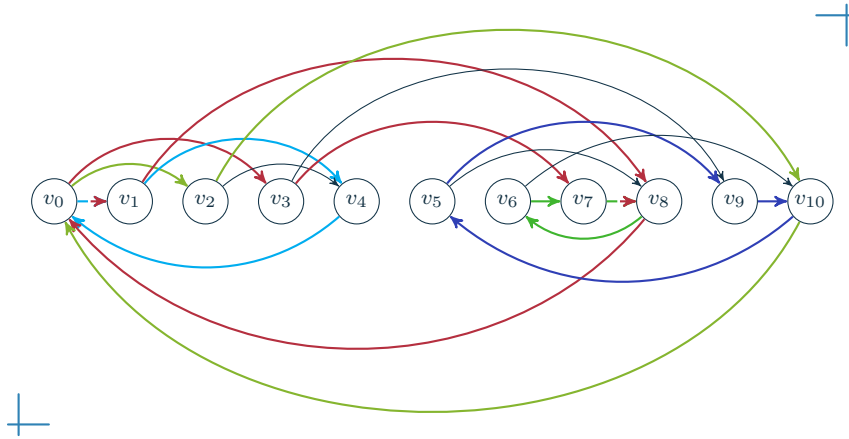


Figure 4.11: Linear ordering where the forward paths for  $(v_8, v_0)$  at  $v_0$  and  $v_8$  and  $(v_{10}, v_0)$  at  $v_0$  and  $v_{10}$  cannot be selected coincidentally.

the analogous conflict for the forward path for  $(v_{10}, v_0)$  at  $v_0$  and  $v_{10}$ . The depicted linear ordering is optimal<sup>1</sup>.

#### 4.6.2 Analyzing the Flow Network Approach

The argument via flow networks in the proof of the Multipath Property readily suggests the use of maximum flow/minimum cut algorithms to establish the property efficiently. To this effect, let  $MinCut(G, s, t)$  be an algorithm that computes a minimum  $s - t$  cut in a graph  $G$  with unit-capacity arcs and returns the set of arcs forming the cut-set. Then, a property-enforcing procedure  $EnforceMultiPaths(G, \pi)$  for the Multipath Property can be implemented as shown in Algorithm 4.8.

We take again advantage of the fact that establishing pairwise arc-disjoint forward paths for the incoming backward arcs of each vertex can be reduced to establishing pairwise arc-disjoint forward paths for the outgoing backward arcs in the reverse graph using the reverse linear ordering. Consequently,  $EnforceMultiPaths(G, \pi)$  simply calls the subroutine  $EnforceMPathsOutgoing(G, \pi)$  twice: once with  $G$  and  $\pi$ , and, if the linear ordering is unchanged, once with  $G^R$  and  $\pi^R$ .

Let us therefore consider  $EnforceMPathsOutgoing(G, \pi)$ . The first step after the initialization is to obtain the flow network  $N$  from the acyclic subgraph  $G|_{\mathcal{F}}$  with unit capacities on all arcs (line 6). Then, for every vertex  $v$  of the input graph, the algorithm checks whether there are arc-disjoint forward paths for all outgoing backward

<sup>1</sup>Verified by an exact algorithm.

**Algorithm 4.8** Multipath Property**Require:** graph  $G = (V, A)$ , linear ordering  $\pi$ **Return:**  $\pi$  if  $\text{MPPath}(\pi)$ , otherwise an improved linear ordering  $\pi'$ 


---

```

1: procedure EnforceMultiPaths( $G, \pi$ )
2:    $\pi' \leftarrow \text{EnforceMPathsOutgoing}(G, \pi)$ 
3:   if  $\pi' \neq \pi$  then return  $\pi'$  else return  $\text{EnforceMPathsOutgoing}(G^R, \pi^R)^R$ 
4: procedure EnforceMPathsOutgoing( $G, \pi$ )
5:    $\pi, \mathcal{B}, \mathcal{F} \leftarrow \text{ComputePositionsAndArcSets}(G, \pi)$ 
6:    $N \leftarrow G|_{\mathcal{F}}$  ▷ construct flow network  $N$  with unit capacities
7:   for all  $v \in V$  do
8:     add new vertex  $s$  to  $N$  ▷ consider forward paths for arcs in  $\mathcal{B}^+(v)$ 
9:     for all  $(v, u) \in \mathcal{B}^+(v)$  do add unit-capacity arc  $(s, u)$  to  $N$ 
10:     $D \leftarrow \text{MinCut}(N, s, v)$ 
11:    if  $|D| < b^+(v)$  then
12:       $X \leftarrow D \cap \mathcal{F}$ 
13:       $Y \leftarrow \mathcal{B}^+(v) \setminus \{(v, u) \mid (s, u) \in D\}$ 
14:       $\mathcal{F}' \leftarrow \mathcal{F} \setminus X \cup Y$ 
15:       $\pi' \leftarrow \text{TopSort}(G|_{\mathcal{F}'})$ 
16:      return  $\pi'$ 
17:    remove  $s$  and all incident arcs from  $N$ 
18:  return  $\pi$ 

```

---

arcs (lines 8–16). To this end, it adds a new vertex  $s$  to the flow network  $N$  (line 8) and arcs from  $s$  to the head of every backward arc (line 9). Next,  $\text{MinCut}(N, s, v)$  is called, which returns a minimum set  $D$  of arcs whose removal disconnects  $s$  from  $v$ . In case that its cardinality equals the number of outgoing backward arcs  $b^+(v)$ , the pairwise arc-disjoint forward paths for  $\mathcal{B}^+(v)$  correspond to the flow paths that can be computed by a maximum flow algorithm. Note that the cardinality of  $D$  cannot exceed  $b^+(v)$ , because  $s$  has only  $b^+(v)$  outgoing arcs and each arc has unit capacity. Hence, if there are not enough pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^+(v)$ , the arc sets  $X$  and  $Y$  are obtained as described in the proof of Lemma 4.10 and an improved set of forward arcs is constructed. Topologically sorting the subgraph of  $G$  restricted to the new set of forward arcs then yields an improved linear ordering  $\pi'$ . At this point, *EnforceMPathsOutgoing* cancels all further examination of forward paths and returns



$\pi'$ , as the given linear ordering  $\pi$  did not respect the Multipath Property. Otherwise, the newly added vertex  $s$  is removed from the flow network  $N$  and the algorithm proceeds with the next vertex.

Let  $\kappa(n, m)$  be the time complexity of computing a minimum cut in a unit-capacity network.

---



---

**Lemma 4.11**

*EnforceMultiPaths*( $G, \pi$ ) runs in time  $\mathcal{O}(n \cdot \kappa(n, m))$ .

---

*Proof.* We start by analyzing *EnforceMPathsOutgoing*, which is shown in [Algorithm 4.8](#). The first step consists in calling the routine *ComputePositionsAndArcSets*, which has a running time of  $\mathcal{O}(m)$  by [Lemma 4.1](#). The second step is to construct a flow network  $N$  from  $G$  with unit capacities. This can be accomplished in constant time by not storing the capacity of each arc explicitly, but rather providing a capacity function that returns 1 for each arc.

Next, consider the statements within the loop beginning in [line 7](#). Adding a vertex to the flow network in [line 8](#) along with an arc for each backward arc in  $\mathcal{B}^+(v)$  in [line 9](#) can be accomplished in time  $\mathcal{O}(m)$ , because we assume an incidence list representation of  $G$  (cf. [Section 4.2](#)). Due to the initialization phase in [line 5](#), we can iterate over  $\mathcal{B}^+(v)$  without additional effort, e. g., via the sorted list  $\langle \mathcal{B}^+ \rangle(v)$ .

Let  $n_G$  and  $m_G$  denote the number of vertices and arcs of  $G$ , respectively, and let analogously  $n_N$  and  $m_N$  denote the number of vertices and arcs of  $N$ . Then,  $n_N = n_G + 1$  and  $m_N \leq m_G - |\mathcal{B}| + \Delta_G$ . We immediately derive  $n_N \in \Theta(n_G)$ . As to  $m_N$ , we obtain with  $\Delta_G \leq n_G$  that  $m_N \leq m_G - |\mathcal{B}| + n_G \leq 2m_G$ , i. e.,  $m_N \in \mathcal{O}(m_G)$ . In consequence, we can safely use  $n = n_G$  and  $m = m_G$  to estimate the time complexity of routines executed on  $N$ .

Let  $\mathcal{O}(\kappa(n, m))$  be the running time of an algorithm that computes a minimum  $s - v$  cut in a unit-capacity network in [line 10](#). The following check in the condition of the if-clause requires only constant time, provided that the minimum cut algorithm stores the cardinality of  $D$  explicitly. The statements in the body of the if-clause are carried out at most once during the execution of *EnforceMPathsOutgoing*, because at its end, the algorithm terminates prematurely. Computing the sets of arcs  $X$  and  $Y$  as well as computing the new set of backward arcs  $\mathcal{F}'$  in [lines 12–14](#) and finally obtaining an improved linear ordering in [line 15](#) can be accomplished in time  $\mathcal{O}(m)$ .

The time complexity of the statements in [lines 8–11](#) is therefore  $\mathcal{O}(m + \kappa(n, m)) = \mathcal{O}(\max\{m, \kappa(n, m)\})$ . Additionally,  $\mathcal{O}(m)$  steps are needed if the condition of the if-clause holds. Otherwise,  $s$  is again removed from  $N$  along with all incident arcs, which requires at most  $\mathcal{O}(m)$  steps. Taking these statements together, we obtain the body of the loop that starts at [line 7](#) and which iterates over all vertices of  $G$ . Consequently, the time complexity of the loop is  $\mathcal{O}(n \cdot (\max\{m, \kappa(n, m)\}) + m)$ . We may assume that computing a minimum cut requires at least one traversal of the graph, which is why  $\kappa(n, m) \in \Omega(m)$ . Hence,  $\mathcal{O}(n \cdot (\max\{m, \kappa(n, m)\}) + m)$  can be simplified to  $\mathcal{O}(n \cdot \kappa(n, m) + m) = \mathcal{O}(n \cdot \kappa(n, m))$ .

With *EnforceMultiPaths* consisting of exactly two calls to *EnforceMPathsOutgoing*, its running time is the same. See again [Section 4.2.5](#) for the reason why we neglect the cost of comparing  $\pi$  to  $\pi'$  and of the reverse operations.  $\square$

We combine again the property-enforcing procedure with *Iterate* and obtain an algorithm *EstablishMultiPaths*( $G, \pi$ ) that establishes the Multipath Property and is defined as *Iterate*( $G, \pi, \text{EnforceMultiPaths}$ ). In consequence of [Lemma 4.11](#) and [Proposition 4.1](#), we obtain:

---



---

**Corollary 4.15**

*EstablishMultiPaths*( $G, \pi$ ) runs in time  $\mathcal{O}(n \cdot m \cdot \kappa(n, m))$ .

---

The currently best known algorithm for solving the maximum flow/minimum cut problem on unit-capacity networks is due to Dinic [[Din70](#), [Din06](#)] and has a time complexity of  $\mathcal{O}(m \cdot \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ .

### 4.6.3 Arc-Disjoint Cycles

Interestingly, the Multipath Property also lets us gain new insights into the dual problem of the FEEDBACK ARC SET problem, the ARC-DISJOINT CYCLES problem.

To this end, consider a vertex  $v$  in a linear ordering of a graph  $G$  that respects the Multipath Property. By [Lemma 4.10](#), there are pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^+(v) \cup \mathcal{B}^-(v)$ . These forward paths form together with their respective backward arcs a set of pairwise arc-disjoint cycles in  $G$ .

Thus, the Multipath Property implies a lower bound on an optimal solution to the ARC-DISJOINT CYCLES problem:

---



---

**Corollary 4.16**

If  $\pi$  is a linear ordering of a graph  $G$  that respects the Multipath Property, then the number of arc-disjoint cycles in  $G$  is at least  $\max_{v \in V} b^-(v) + b^+(v)$ .

---

#### 4.6.4 An $\mathcal{NP}$ -hard Extension

Earlier in this section, [Figure 4.9](#) has shown us an example for an optimal linear ordering of a graph where the two induced backward arcs have non-disjoint forward paths. Consequently, we asked for sufficient conditions for a set of backward arcs to have pairwise arc-disjoint forward paths. The Multipath Property provided one such condition by stating that a common head or tail suffices. In fact, this condition can be strengthened even more with the help of the following observation:

---



---

**Lemma 4.12**

Let  $\mathcal{B}$  be the set of backward arcs induced by a linear ordering  $\pi$  of a graph  $G$  such that  $\pi$  respects the Multipath Property, let  $b \in \mathcal{B}$ , and let  $v$  be vertex on a forward path  $P_b$  for  $b$ . Let  $Y \subseteq \mathcal{B}^+(v)$  or  $Y \subseteq \mathcal{B}^-(v)$  be a subset of either the incoming or the outgoing backward arcs of  $v$  and let  $X$  be a set of forward arcs induced by  $\pi$  such that every forward path of  $b$  or a backward arc in  $Y$  contains an arc in  $X$  and  $|X| \leq |Y|$ . Then,  $\mathcal{B}' = \mathcal{B} \setminus (Y \cup [b]_{\parallel}) \cup X$  is feasible.

---

*Proof.* Let  $G = (V, A)$  and consider a backward arc  $b = (u, w) \in \mathcal{B}$  and a vertex  $v$  that lies on a forward path  $P_b$  for  $b$ . This implies that  $P_b$  consists of two subpaths  $P'_b = w \rightsquigarrow v$  and  $P''_b = v \rightsquigarrow u$ . Note that  $v = u$  or  $v = w$  is not precluded. In this case, one of both subpaths is empty and  $b \in \mathcal{B}(v)$ . Observe, however, that if  $b \in \mathcal{B}(v)$ , but  $[b]_{\parallel} \not\subseteq Y$ , then the existence of a set of forward arcs  $X$  that covers all forward paths in  $Y \cup [b]_{\parallel}$  with  $|X| \leq |Y|$  contradicts the Multipath Property. Hence, either  $b \notin \mathcal{B}(v)$  or  $[b]_{\parallel} \subseteq Y$ . Assume that  $Y \subseteq \mathcal{B}^+(v)$  (cf. [Figure 4.12](#)).

As  $\pi$  respects the Multipath Property, there must be pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}(v) = \mathcal{B}^+(v) \cup \mathcal{B}^-(v)$ . In particular, there must be pairwise arc-disjoint forward paths for the backward arcs in  $Y$ , due to  $Y \subseteq \mathcal{B}^+(v)$ . Subsequently, every set of forward arcs that covers all forward paths of the backward arcs in  $Y$  has cardinality at least  $|Y|$ . Hence,  $|X| = |Y|$  and every arc  $x \in X$  is part of at least one forward path of a backward arc in  $Y$ .

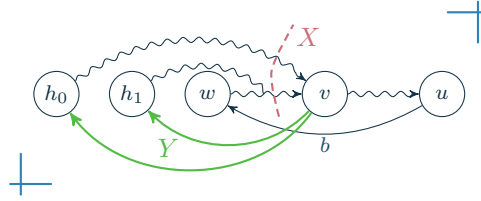


Figure 4.12: Schematic drawing of the situation in Lemma 4.12 with  $Y \subseteq \mathcal{B}^+(v)$ .

Let  $\mathcal{F} = A \setminus \mathcal{B}$  and  $\mathcal{F}' = A \setminus \mathcal{B}' = \mathcal{F} \setminus X \cup (Y \cup [b]_{\parallel})$ . Suppose that  $\mathcal{B}'$  is not feasible. Then, there must be a cycle  $C$  consisting only of arcs in  $\mathcal{F}'$ . As  $\mathcal{F}$  is feasible,  $C$  must contain at least two arcs of  $Y \cup [b]_{\parallel}$ : If  $C$  contained just one arc  $y \in (Y \cup [b]_{\parallel})$ , all other arcs must be in  $\mathcal{F} \setminus X$ , which implies that there is a forward path for  $y$  in  $G|_{\mathcal{F}}$  that is not covered by  $X$ , a contradiction. In particular, this also implies that  $C$  cannot only contain arcs in  $[b]_{\parallel} \cup \mathcal{F} \setminus X$ , i. e.,  $C$  must contain at least one arc from  $Y$ .

Consider the possibility that  $C$  does not contain  $b$  or one of its parallel arcs. In this case, there is a set of  $k \geq 2$  arcs  $Y' = \{(v, h_i) \mid 0 \leq i < k\} \subseteq Y$  such that  $C$  is composed of this set of arcs plus a set of paths  $\mathcal{P} = \{h_i \rightsquigarrow v \mid 0 \leq i < k\}$  which contain only arcs of  $\mathcal{F} \setminus X$ . Hence, with respect to  $\pi$ , the paths in  $\mathcal{P}$  are forward paths for the backward arcs in  $Y'$  that are not covered by  $X$ , a contradiction.

Consequently,  $C$  must contain  $b = (u, w)$  or one of its parallel arcs and  $k \geq 1$  further arcs  $(v, h_i) \in Y$ , where  $0 \leq i < k$ . Thus,  $C$  must have a subpath  $P = w \rightsquigarrow v$  which contains only arcs of  $\mathcal{F} \setminus X$ . As every arc  $x \in X$  is contained in at least one forward path of a backward arc in  $Y$  and all of these forward paths end at  $v$ , no arc of  $X$  can be part of the subpath  $P'_b = v \rightsquigarrow u$  of  $P_b$ . Subsequently,  $P$  and  $P'_b$  form a path  $w \rightsquigarrow v \rightsquigarrow u$  that uses only arcs in  $\mathcal{F} \setminus X$  and is a forward path for  $b$ , a contradiction to the definition of  $X$ .

In conclusion,  $\mathcal{F}'$  must be acyclic, so  $\mathcal{B}'$  is feasible. The analogous statement for the case that  $Y \subseteq \mathcal{B}^-(v)$  follows immediately by considering the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^R$ .  $\square$

The extension of the Multipath Property is now straightforward:

---

**Lemma 4.13** EXTENDED MULTIPATH PROPERTY

Let  $\pi^*$  be an optimal linear ordering of a graph  $G$ , let  $b$  be a backward arc induced by  $\pi^*$ , and let  $v$  be vertex on a forward path  $P_b$  for  $b$ . Then, there is a set of pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^+(v) \cup [b]_{\parallel}$  as well as one for the backward arcs in  $\mathcal{B}^-(v) \cup [b]_{\parallel}$ .

---

*Proof.* Let  $G = (V, A)$  and denote by  $\mathcal{B}$  and  $\mathcal{F}$  the set of backward and forward arcs induced by  $\pi^*$ , respectively. Consider a backward arc  $b \in \mathcal{B}$ . By [Lemma 4.5](#), there must be a forward path  $P_b$  for  $b$ . Let  $v$  be a vertex on  $P_b$ . If  $b \in \mathcal{B}(v)$ , then the statement immediately follows from [Lemma 4.10](#). Hence, assume that  $b \notin \mathcal{B}(v)$ , i. e.,  $[b]_{\parallel} \cap \mathcal{B}(v) = \emptyset$ .

Suppose that there is no set of forward paths for  $\mathcal{B}^+(v) \cup [b]_{\parallel}$  that are pairwise arc-disjoint. Then, there must be a subset of backward arcs  $Y \subseteq \mathcal{B}^+(v)$  and a directed multicut  $X \subseteq \mathcal{F}$  such that every forward path for  $b$  and every forward path for a backward arc in  $Y$  contains an arc from  $X$  and  $|X| < |Y| + |[b]_{\parallel}|$ , i. e.,  $|X| \leq |Y|$ .

Let  $\mathcal{B}' = \mathcal{B} \setminus (Y \cup [b]_{\parallel}) \cup X$ . As  $Y \cap [b]_{\parallel} = \emptyset$  and  $|[b]_{\parallel}| \geq 1$ ,  $|\mathcal{B}'| < |\mathcal{B}|$ . In consequence of [Lemma 4.10](#),  $\pi^*$  respects the Multipath Property, so we can conclude from [Lemma 4.12](#) that  $\mathcal{B}'$  is feasible, a contradiction to the optimality of  $\pi^*$ .

To prove the existence of a set of pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^-(v) \cup [b]_{\parallel}$ , we consider the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^{*R}$  instead.  $\square$

Note that in contrast to the Multipath Property, the proof of [Lemma 4.13](#) does not suggest an efficient algorithm to ensure that an arbitrary linear ordering  $\pi$  respects this property. The DIRECTED MULTICUT problem that appears as a subroutine in the proof is  $\mathcal{NP}$ -hard in general (cf. [Section 2.1.5](#)) and even remains so in case of two terminal pairs [[KPPW15](#)], which would otherwise have been sufficient here for a polynomial-time implementation: The first terminal pair  $(s_1, t_1)$  consists in the head  $(s_1)$  and tail  $(t_1)$  of the considered backward arc  $b$ . For the second terminal pair  $(s_2, t_2)$ , we introduce an artificial source  $s$  as in the proof of [Lemma 4.10](#), where  $s$  has an arc to every head of an outgoing backward arc of  $v$ , and then use  $s_2 = s$  and  $t_2 = v$ .

## 4.7 Multipath Blocking Vertices Property

With the Blocking Vertices Property and the Multipath Property, two independent extensions of the Path Property have been introduced in the preceding sections. It would be desirable, however, to be able to establish a combination of both, i. e., to have a Blocking Vertices Property for the arc-disjoint forward paths of the Multipath Property. This is the aim of this section.

### 4.7.1 Non-Blocking Multipaths

In analogy to [Corollary 4.11](#) for the Blocking Vertices Property and simple forward paths, we can formulate the existence of non-blocking multipaths as follows:

---

**Lemma 4.14** MULTIPATH BLOCKING VERTICES PROPERTY  
 Let  $\pi^*$  be an optimal linear ordering of a graph  $G$ . Then,  $\pi_{\text{sp}/l}^*$  and  $\pi_{\text{sp}/r}^*$  respect the Multipath Property.

---

*Proof.* Just as the Multipath Blocking Vertices Property is obtained from a combination of the Blocking Vertices Property and the Multipath Property, so is its proof. Let  $\pi^*$  be an optimal linear ordering of  $G = (V, A)$  and suppose, for the sake of contradiction, that  $\pi^*$  does not respect the Multipath Blocking Vertices Property.

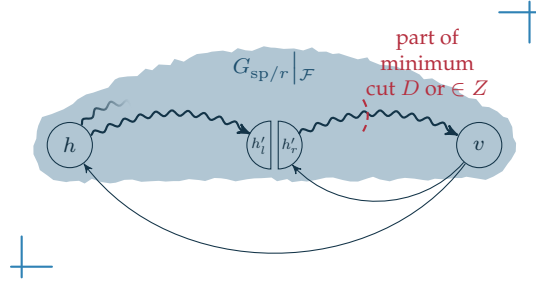
**The Right-Blocking Split Graph and Outgoing Backward Arcs** We assume first that  $\pi_{\text{sp}/r}^*$  does not respect the Multipath Property. In this context, let  $v$  be a vertex of  $G$ , and suppose that the Multipath Property is violated for the set of outgoing backward arcs  $\mathcal{B}^+(v)$  of  $v$ . As in the proof of the Multipath Property, the forward paths for the outgoing and incoming backward arcs of a vertex can be considered separately. The argument for the set of incoming backward arcs  $\mathcal{B}^-(v)$  is symmetric and will be addressed at the end of the proof.

In case that  $v$  itself is right-blocking and therefore split into  $v_l$  and  $v_r$ , substitute  $v$  by  $v_l$  in the following paragraphs. For the correctness, recall that  $v_l$  inherits  $\mathcal{F}^-(v)$  and  $\mathcal{B}^+(v)$ , so  $\mathcal{F}^-(v_l)$  corresponds to  $\mathcal{F}^-(v)$  and  $\mathcal{B}^+(v_l)$  corresponds to  $\mathcal{B}^+(v)$ . Furthermore, arcs in  $\mathcal{F}^+(v)$  cannot be part of forward paths for backward arcs in  $\mathcal{B}^+(v)$ .

By assumption, there are no pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^+(v)$  in the right-blocking split graph with linear ordering  $\pi_{\text{sp}/r}^*$ . Let  $s$ ,  $D$ ,  $X$ , and  $Y$  be defined as in the proof of [Lemma 4.10](#), i. e.,  $s$  is the newly introduced source with an outgoing arc to every head of a backward arc in  $\mathcal{B}^+(v)$ ,  $D$  is a minimum cut with  $|D| < b^+(v)$  that separates  $s$  from  $v$ ,  $X = D \cap \mathcal{F}$ , and  $Y = \mathcal{B}^+(v) \setminus \{(v, h) \mid (s, h) \in D\}$  (cf. [Figure 4.10\(b\)](#)).

Because we considered the split graph when computing the minimum cut,  $X$  only covers forward paths that are preserved during the splitting of right-blocking vertices. In consequence, there may be uncovered forward paths in the unsplit graph  $G$  that contain right-blocking vertices.

To overcome this issue, construct a set  $Z$  of right-blocking vertices as follows: For each backward arc in  $Y$ , consider all forward paths according to  $\pi^*$  in the unsplit graph  $G$



**Figure 4.13:** Proof of Multipath Blocking Vertices Property: A forward path  $h \rightsquigarrow v$  for the backward arc  $(v, h)$  in  $G$  (but not in  $G_{sp/r}$ ) that contains the head of another outgoing backward arc  $(v, h')$  (respectively  $(v, h'_r)$  in  $G_{sp/r}$ ) of  $v$ . If all forward paths for  $(v, h')$  are covered, so are all forward paths for  $(v, h)$  via  $h'$ .

that do not contain an arc of the reduced minimum cut  $X$ . Place the last right-blocking vertex that occurs on a traversal of the respective cropped forward paths in  $Z$ . Because  $X$  is a minimum cut for all forward paths for arcs in  $Y$  according to  $\pi_{sp/r}^*$  in the split graph, every cropped forward path not passing through an arc in  $X$  must contain a right-blocking vertex.

The intention now is to additionally exchange the incoming backward arcs for the outgoing forward arcs of the right-blocking vertices in  $Z$  in analogy to the proof of the Blocking Vertices Property. Let  $H = \{h \in V \mid (v, h) \in Y\}$  be the set of all heads of the backward arcs in the reduced set  $Y$  of outgoing backward arcs of  $v$ . Suppose that some vertices in  $Z$  are at the same time the head of an outgoing backward arc of  $v$ , i. e.,  $H \cap Z \neq \emptyset$ . As we only consider cropped forward paths, there must hence be two backward arcs  $b = (v, h), b' = (v, h') \in Y$  such that a forward path  $P_b$  for  $b$  in the unsplit graph  $G$  passes  $h'$  and  $h'$  is right-blocking, i. e.,  $b' = (v, h') = (v, h'_r)$  (see also [Figure 4.13](#)). Note, however, that this also implies that every forward path for  $b'$  in  $G$  is a subpath of a forward path for  $b$  and that destroying all forward paths of  $b'$  also cuts all forward paths of  $b$  that contain  $h'$ . As we placed the last right-blocking vertex that occurs on a traversal of every cropped forward path for  $b$  in  $Z$  and every subpath  $h' \rightsquigarrow v$  either contains a right-blocking vertex or an arc of  $X$ ,  $h' \notin Z$ . Subsequently,  $Z \cap H = \emptyset$ , and, as all forward paths end at  $v$ ,  $v \notin Z$  either.

Let now

$$\mathcal{B}' = \mathcal{B} \setminus \mathcal{B}^- [Z] \cup \mathcal{F}^+ [Z]$$

and

$$\mathcal{F}' = \mathcal{A} \setminus \mathcal{B}' = \mathcal{F} \setminus \mathcal{F}^+ [Z] \cup \mathcal{B}^- [Z].$$

By Lemma 4.7,  $\mathcal{B}'$  is feasible and  $|\mathcal{B}'| = |\mathcal{B}|$ . Furthermore, every path that consists only of arcs in  $\mathcal{F}'$  and contains a vertex in  $Z$  must end within  $Z$ . Next, consider

$$\mathcal{B}'' = \mathcal{B}' \setminus Y \cup X = \mathcal{B} \setminus (\mathcal{B}^-[Z] \cup Y) \cup \mathcal{F}^+[Z] \cup X$$

and

$$\mathcal{F}'' = A \setminus \mathcal{B}'' = \mathcal{F}' \setminus X \cup Y = \mathcal{F} \setminus (\mathcal{F}^+[Z] \cup X) \cup \mathcal{B}^-[Z] \cup Y.$$

Note that  $Z \cap H = \emptyset$  immediately implies that  $\mathcal{B}^-[Z] \cap Y = \emptyset$ . Furthermore,  $\mathcal{F}^+[Z] \cap X = \emptyset$  because  $X$  covers exactly all forward paths in  $\pi_{\text{sp}/r}^*$  and no forward path in  $\pi_{\text{sp}/r}^*$  can contain a vertex of  $Z$ . Hence, if the tail  $t$  of an arc in  $X$  were in  $Z$ , then  $t$  must be the head of an outgoing backward arc of  $v$ , but  $Z \cap H = \emptyset$ . As shown in the proof of Lemma 4.10,  $|X| < |Y|$ . Thus,  $|\mathcal{B}''| < |\mathcal{B}'| = |\mathcal{B}|$ .

It remains to show that  $\mathcal{B}''$  is feasible, i. e.,  $G|_{\mathcal{F}''}$  is acyclic. Suppose, for contradiction, that  $G|_{\mathcal{F}''}$  contains a simple cycle  $C$ , i. e., all arcs of  $C$  are in  $\mathcal{F}''$ . Due to  $\mathcal{B}'$  being feasible,  $C$  must contain exactly one arc  $(v, h) \in Y$ . Observe that if  $C$  contained more than one arc of  $Y$ , it would contain  $v$  at least twice and would therefore not be simple. Then, the remaining arcs of the cycle must constitute a path  $P = h \rightsquigarrow v$  and consist solely of arcs in  $\mathcal{F}' \setminus X$ . As every path in  $\mathcal{F}'$  that contains a vertex of  $Z$  must also end in  $Z$  and  $v \notin Z$  by construction,  $P$  cannot contain a vertex of  $Z$ . Thus,  $P$  is a forward path for  $(v, h)$  in  $G|_{\mathcal{F}}$  and contains neither an arc of  $X$  nor a vertex of  $Z$ , a contradiction. Subsequently,  $C$  cannot exist, so  $\mathcal{B}''$  is feasible. Because  $|\mathcal{B}''| < |\mathcal{B}|$ , this, however, contradicts the optimality of  $\pi^*$ .

**The Left-Blocking Split Graph and Outgoing Backward Arcs** Let us now suppose that  $\pi_{\text{sp}/l}^*$  does not respect the Multipath Property and consider again the outgoing backward arcs  $\mathcal{B}^+(v)$  of a vertex  $v$ .

The proof follows largely that for the right-blocking split graph, i. e., we add a source vertex  $s$  with an outgoing arc to every head of a backward arc in  $\mathcal{B}^+(v)$  and consider the minimum  $s - v$  cut  $D$ . If the Multipath Property is violated, then  $|D| < b^+(v)$  and we define the sets  $X$  and  $Y$  as in the right-blocking case.

There is a difference in the construction of the set  $Z$  in that it contains left- instead of right-blocking vertices: For each backward arc in  $Y$ , consider again all forward paths according to  $\pi^*$  in the unsplit graph  $G$  that are not covered by an arc of the reduced cut  $X$ , but now place the last left-blocking vertex that occurs on a traversal of the respective cropped forward path in  $Z$ .



As we are dealing with left-blocking vertices, we are interested in the arc sets  $\mathcal{F}^- [Z]$  as well as  $\mathcal{B}^+ [Z]$  here. By applying again [Lemma 4.7](#), we obtain that

$$\mathcal{B}' = \mathcal{B} \setminus \mathcal{B}^+ [Z] \cup \mathcal{F}^- [Z]$$

is feasible,  $|\mathcal{B}'| = |\mathcal{B}|$  and that every path that consists only of arcs in

$$\mathcal{F}' = A \setminus \mathcal{B}' = \mathcal{F} \setminus \mathcal{F}^- [Z] \cup \mathcal{B}^+ [Z]$$

and contains a vertex in  $Z$  must start in  $Z$ .

With the same argument as above, we obtain that  $H \cap Z = \emptyset$ , which implies that  $Y \cap \mathcal{B}^+ [Z] = \emptyset$ . Furthermore,  $X \cap \mathcal{F}^- [Z] = \emptyset$ , because all vertices in  $Z$  are left-blocking and therefore split vertically in  $\pi_{\text{sp}/l}$ . This implies that all forward paths using an arc in  $\mathcal{F}^- [Z]$  must end there, but  $X$  is a minimum cut for forward paths ending at  $v$ . We obtain the improved set of backward and forward arcs as

$$\mathcal{B}'' = \mathcal{B}' \setminus Y \cup X = \mathcal{B} \setminus (\mathcal{B}^+ [Z] \cup Y) \cup \mathcal{F}^- [Z] \cup X$$

and

$$\mathcal{F}'' = A \setminus \mathcal{B}'' = \mathcal{F}' \setminus X \cup Y = \mathcal{F} \setminus (X \cup \mathcal{F}^- [Z]) \cup \mathcal{B}^+ [Z] \cup Y.$$

It remains again to show that  $\mathcal{B}''$  is feasible, i. e.,  $G|_{\mathcal{F}''}$  is acyclic. Hence, suppose that  $G|_{\mathcal{F}''}$  contains a simple cycle  $C$ . As in the right-blocking case,  $C$  must then contain exactly one arc  $(v, h) \in Y$  and all other arcs of  $C$  form a path  $P = h \rightsquigarrow v$  that uses only arcs in  $\mathcal{F}' \setminus X$ . Here, every path in  $\mathcal{F}'$  that contains a vertex of  $Z$  must also start in  $Z$ , but  $h \in H$  and  $H \cap Z = \emptyset$ , so  $P$  again cannot contain a vertex of  $Z$ . Subsequently,  $P$  is a forward path for  $(v, h)$  in  $G|_{\mathcal{F}}$  that is covered neither by  $X$  nor by  $Z$ , a contradiction. Hence,  $\mathcal{B}''$  is feasible with  $|\mathcal{B}''| < |\mathcal{B}|$  a contradiction to  $\pi^*$  being optimal.

**Incoming Backward Arcs** The respective proofs for the incoming backward arcs in the left-blocking as well as the right-blocking case follow by considering the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^{*R}$ . Note that the combination of incoming backward arcs and the left-blocking case corresponds to the outgoing backward arcs and the right-blocking case in the reverse graph and linear ordering, and the combination of incoming backward arcs and the right-blocking case corresponds to the outgoing backward arcs and the left-blocking case.  $\square$

[Figure 4.14](#) continues the example given in [Figure 4.7](#) and shows how this linear ordering instance is improved by enforcing the Multipath Blocking Vertices Property for the outgoing backward arcs of vertex  $v$ : The first subfigure, [Figure 4.14\(a\)](#), depicts

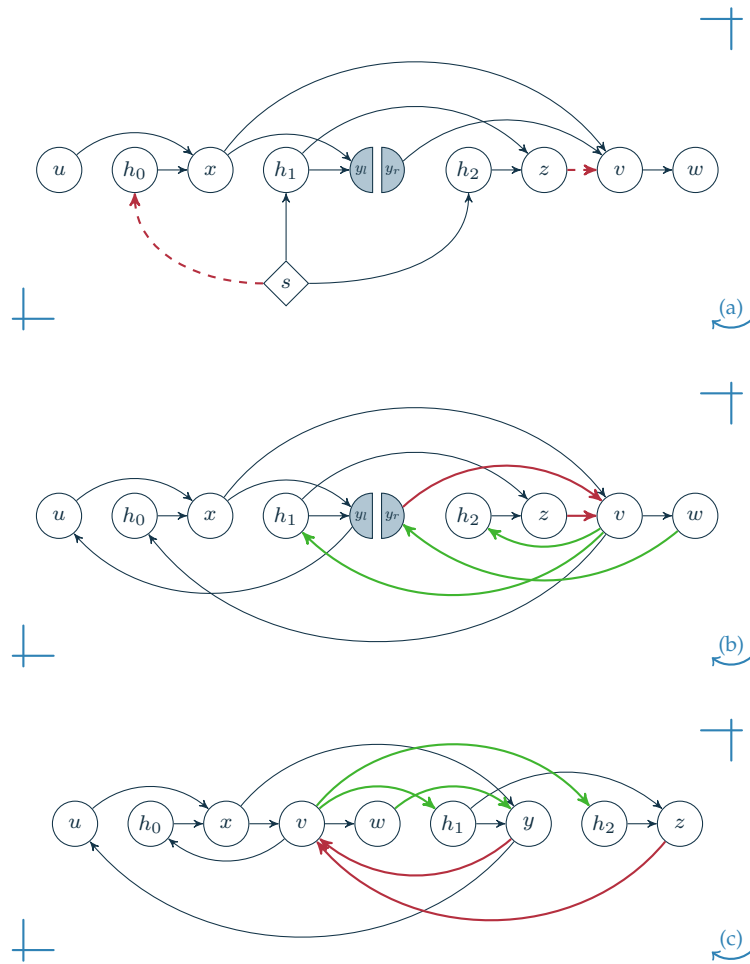


Figure 4.14: Enforcing the Multipath Blocking Vertices Property on the instance introduced in Figure 4.7 for the outgoing backward arcs of vertex  $v$ : Find a minimum  $s - v$  cut (a), identify the backward and forward arcs to exchange with each other (b), and obtain the improved linear ordering (c).

the acyclic subgraph of the right-blocking split graph with an additional source  $s$  and arcs to every head of an outgoing backward arc of  $v$ . The dashed red arcs highlight a minimum  $s - v$  cut, which consists of the two arcs  $(s, h_0)$  and  $(z, v)$ . As  $(s, h_0)$  is incident to  $s$ , the reduced minimum cut  $X$  contains only  $(z, v)$ , and the subset  $Y \subseteq \mathcal{B}^+(v)$  of exchangeable backward arcs equals  $\{(v, h_1), (v, h_2)\}$ . Additionally, we construct the set of right-blocking vertices  $Z$  with  $Z = \{y\}$ , because  $y$  is the last right-blocking vertex encountered on a traversal of the cropped forward path  $\langle y \rangle$  of  $\langle h_1, y, v \rangle$  in the unsplit acyclic subgraph  $G|_{\mathcal{F}}$ . We thus have identified two sets of arcs that switch their roles from backward to forward and vice versa: The arcs in  $Y \cup \mathcal{B}^-(y) = \{(v, h_1), (v, h_2), (w, y)\}$  will become forward arcs, whereas the arcs in  $X \cup \mathcal{F}^+(y) = \{(z, v), (y, v)\}$  will become backward arcs. In [Figure 4.14\(b\)](#) these two arc sets are highlighted. As the right-blocking split graph is shown, however,  $(w, y)$  must be translated to  $(w, y_r)$  and  $(y, v)$  to  $(y_r, v)$ . Finally, [Figure 4.14\(c\)](#) depicts a linear ordering obtained by topologically sorting the improved acyclic subgraph.

We introduce a predicate  $\text{MNoBlock}(\pi)$  that expresses whether a linear ordering  $\pi$  respects the Multipath Blocking Vertices Property and derive from [Lemma 4.14](#):

---



---

**Corollary 4.17**

For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \text{MNoBlock}(\pi)$ .

---

The definition of the Multipath Blocking Vertices Property immediately implies that  $\forall \pi : \text{MNoBlock}(\pi) \Rightarrow \text{NoBlock}(\pi) \wedge \text{MPath}(\pi)$ . With [Corollary 4.14](#), we thus obtain:

---



---

**Corollary 4.18**

For every linear ordering  $\pi$  holds:

$$\text{MNoBlock}(\pi) \Rightarrow \text{Nest}(\pi) \wedge \text{Path}(\pi) \wedge \text{NoBlock}(\pi) \wedge \text{MPath}(\pi).$$


---

## 4.7.2 Flow Networks for Split Graphs

In [Algorithm 4.8](#) it has been shown how to enforce the Multipath Property using a flow network. We extend this algorithm here to also incorporate blocking vertices, such that we finally obtain an algorithm for the Multipath Blocking Vertices Property.

To this end, consider the procedure  $\text{EnforceMNSBOut}(G, \pi, \sigma)$  listed in [Algorithm 4.9](#). Like all property-enforcing algorithms, it takes a graph  $G$  and a linear ordering  $\pi$

**Algorithm 4.9** Multipath Blocking Vertices Property**Require:** graph  $G = (V, A)$ , linear ordering  $\pi$ , (side  $\sigma = \text{left}$  or  $\sigma = \text{right}$ )**Return:**  $\pi$  if  $\text{MNoBlock}(\pi)$ , otherwise an improved linear ordering  $\pi'$ 


---

```

1: procedure EnforceMultiPathsNoBlocking( $G, \pi$ )
2:    $\pi' \leftarrow \text{EnforceMNSBOut}(G, \pi, \text{left})$ 
3:   if  $\pi' \neq \pi$  then return  $\pi'$ 
4:   else  $\pi' \leftarrow \text{EnforceMNSBOut}(G, \pi, \text{right})$ 
5:     if  $\pi' \neq \pi$  then return  $\pi'$ 
6:     else  $\pi' \leftarrow \text{EnforceMNSBOut}(G^R, \pi^R, \text{left})^R$ 
7:       if  $\pi' \neq \pi$  then return  $\pi'$  else return  $\text{EnforceMNSBOut}(G^R, \pi^R, \text{right})^R$ 
8: procedure EnforceMNSBOut( $G, \pi, \sigma$ )
9:    $\pi, \mathcal{B}, \mathcal{F} \leftarrow \text{ComputePositionsAndArcSets}(G, \pi)$ 
10:   $Q \leftarrow$  set of  $\sigma$ -blocking vertices in  $G$  according to  $\pi$ 
11:   $G_{\text{sp}}, \pi_{\text{sp}} \leftarrow \text{SplitVertically}(G, \pi, Q)$ 
12:   $\pi_{\text{sp}}, \mathcal{B}', \mathcal{F}' \leftarrow \text{ComputePositionsAndArcSets}(G_{\text{sp}}, \pi_{\text{sp}})$ 
13:   $N \leftarrow G_{\text{sp}}|_{\mathcal{F}'}$   $\triangleright$  construct flow network  $N$  with unit capacities as default
14:  for all  $q \in Q$  do add zero-capacity arc  $(q_l, q_r)$  to  $N$ 
15:  for all  $v \in V$  do
16:    add new vertex  $s$  to  $N$   $\triangleright$  consider forward paths for arcs in  $\mathcal{B}_{\pi_{\text{sp}}}^+(v)$ 
17:    for all  $(v, u) \in \mathcal{B}_{\pi_{\text{sp}}}^+(v)$  do add arc  $(s, u)$  to  $N$ 
18:     $D \leftarrow \text{MinCut}(N, s, v)$   $\triangleright$  compute minimum cut closest to  $v$ 
19:    if  $\text{size}(D) < b_{\pi_{\text{sp}}}^+(v)$  then
20:       $Z \leftarrow \{q \in Q \mid (q_l, q_r) \in D\}$ 
21:       $X \leftarrow D \cap \mathcal{F}'$ 
22:       $Y \leftarrow \mathcal{B}^+(v) \setminus \{(v, u) \mid (s, u) \in D\}$ 
23:      if  $\sigma = \text{right}$  then  $\mathcal{F}'' \leftarrow \mathcal{F} \setminus (X \cup \mathcal{F}^+[Z]) \cup Y \cup \mathcal{B}^-[Z]$ 
24:      else  $\mathcal{F}'' \leftarrow \mathcal{F} \setminus (X \cup \mathcal{F}^-[Z]) \cup Y \cup \mathcal{B}^+[Z]$ 
25:       $\pi' \leftarrow \text{TopSort}(G|_{\mathcal{F}''})$ 
26:      return  $\pi'$ 
27:    remove  $s$  and all incident arcs from  $N$ 
28:  return  $\pi$ 

```

---

as input. In this case, however, there is a third parameter  $\sigma$  specifying whether to enforce the Multipath Blocking Vertices Property with respect to the left- or to the right-blocking vertices. The algorithm follows largely that of *EnforceMPathsOutgoing* shown in [Algorithm 4.8](#), but adds a couple of statements. Like *EnforceMPathsOutgoing*, *EnforceMNSBOut* only considers the set of outgoing backward arcs for every vertex.

In [line 10](#), a set  $Q$  of vertices is obtained that either contains all left- or all right-blocking vertices in  $G$ , depending on whether  $\sigma$  is set to `left` or `right`. This set is then used in the following statement to obtain, respectively, the left- or right-blocking split graph  $G_{\text{sp}}$  along with linear ordering  $\pi_{\text{sp}}$ . The initialization step in the following line sets the additional data structures up. In contrast to *EnforceMPathsOutgoing*, the flow network here is constructed from the split graph restricted to the set of forward arcs (cf. [line 13](#)). Unless specified otherwise, all arcs of  $N$  have a default capacity of one unit. The following line, [line 14](#), describes the addition of zero-capacity arcs between the two split components of a vertex in  $Q$  and thereby represents the key idea to handle the blocking vertices: As their capacity is set to zero, they do not influence the value of a maximum flow or minimum cut between a source and a sink, but if they are on a path connecting these two vertices, they may be part of the cut set.

Next, the algorithm loops over all vertices  $v$  of the input graph. Recall that when the vertical split operation was introduced in [Section 4.5](#), we identified every arc of the unsplit graph with its counterpart in the split graph. Therefore, we do not have to make a distinction between whether an arc is incident to a split or an unsplit vertex.

The loop ([lines 15–27](#)) encompasses the check for all outgoing backward arcs for each vertex. To this end, a new vertex  $s$  is added to the flow network ([line 16](#)) along with an arc from  $s$  to the (split or unsplit) head of every outgoing backward arc of  $v$ . In the next step, the minimum  $s - v$  cut closest to  $v$  is obtained ([line 18](#)). As already mentioned, this cut set,  $D$ , may contain zero-capacity arcs that originate from the blocking vertices. The choice of the minimum cut as the one that is closest to  $v$  implies that the zero-capacity arc belonging to last blocking vertex on each forward path, if existing, is in  $D$ . Furthermore, we must compare the size of  $D$ , i. e., the sum of capacities of arcs in  $D$ , to  $b^+(v)$  in [line 19](#) rather than  $D$ 's cardinality, in order to know whether the set of backward arcs can be improved.

If the test is positive, the set  $Z$  of blocking vertices is extracted from  $Q$  as those whose zero-capacity arcs are part of the minimum cut  $D$ . As mentioned above, we assume here that  $\text{MinCut}(N, s, v)$  yields a set of arcs that is closest to  $v$  and minimal with respect to the definition given in [Section 3.1](#), as can be obtained, e. g., by a backward-looking

breadth-first search starting from  $v$ . Then,  $Z$  matches the definition in the proof of [Lemma 4.14](#).

Next, the reduced cut set  $X$  and the corresponding reduced set of backward arcs  $Y$  are obtained in [line 21](#) like in [Algorithm 4.8](#). Note, however, that intersecting  $D$  and the set of forward arcs in  $G$ ,  $\mathcal{F}$ , not only eliminates arcs incident to the new vertex  $s$ , but also zero-capacity arcs originating from  $Q$ .

The improved set of forward arcs is obtained for the right-blocking case in [line 23](#) and for the left-blocking case in [line 24](#), as described in the proof of [Lemma 4.14](#). Topologically sorting the subgraph of  $G$  restricted to this new set of forward arcs then yields the improved linear ordering  $\pi'$ , which is returned immediately. Otherwise, if the test in [line 19](#) was negative, the algorithm continues with the next vertex.

The procedure *EnforceMultiPathsNoBlocking*( $G, \pi$ ), which is also listed in [Algorithm 4.9](#), calls *EnforceMNSBOut* four times: once with side  $\sigma$  set to `left`, once with  $\sigma$  set to `right` with parameters  $G$  and  $\pi$ , and the same with parameters  $G^R$  and  $\pi^R$  to ensure the Multipath Blocking Vertices Property also for the incoming backward arcs of every vertex.

Let  $\kappa(n, m)$  be the time complexity of computing a minimum cut in a unit-capacity network.

---



---

**Lemma 4.15**

*EnforceMultiPathsNoBlocking*( $G, \pi$ ) runs in time  $\mathcal{O}(n \cdot \kappa(n, m))$ .

---

*Proof.* We start again by analyzing *EnforceMNSBOut*. *EnforceMNSBOut* differs from *EnforceMPathsOutgoing*, which was analyzed in the proof of [Lemma 4.11](#), only in few statements. The construction of the set of blocking vertices  $Q$  in [line 10](#) as well as the call to *SplitVertically* in [line 11](#) and the insertion of the zero-capacity arcs in [line 14](#) can be accomplished in time  $\mathcal{O}(m)$ . In comparison to  $G$ , the split graph  $G_{\text{sp}}$  has at most twice as many vertices and the same number of arcs, hence, their size is asymptotically equal. Thus, the initialization steps in [line 9](#) and [line 12](#) require  $\mathcal{O}(m)$  time each.

An important point in the algorithm is the call to *MinCut* in [line 18](#). In the proof of [Lemma 4.11](#), we introduced a function  $\kappa(n, m)$  to express the running time of an algorithm that computes a minimum cut in a unit-capacity network as  $\mathcal{O}(\kappa(n, m))$ . Here, the flow network additionally contains exactly the zero-capacity arcs, which are, however, irrelevant for the computation of the maximum  $s - v$  flow. The set of arcs that form a minimum cut can be obtained by a backward breadth-first search on the residual

graph that starts from  $v$  and that also takes the zero-capacity arcs with residual capacity zero into account. This does not increase the asymptotic running time of *MinCut* and is compliant with the demands on the cut set stated in the description of [Algorithm 4.9](#). The effort to compute an improved set of forward arcs stays asymptotically the same in comparison to *EnforceMPathsOutgoing*.

In conclusion, as  $\kappa(n, m) \in \Omega(m)$  as argued in the proof of [Lemma 4.11](#), the time needed to compute the minimum cut still dominates the asymptotic time complexity of the statements inside the loop. For the total running time of the loop, this yields  $\mathcal{O}(n \cdot \kappa(n, m))$ , which in turn supersedes the additional effort of  $\mathcal{O}(m)$  at the beginning of the procedure.

*EnforceMultiPathsNoBlocking* consists of four calls to *EnforceMNSBOut* and therefore has the same asymptotic running time. Due to the general assumptions regarding the implementation (cf. [Section 4.2.5](#)), we neglect again the cost of comparing  $\pi$  to  $\pi'$  and of the reverse operations.  $\square$

By passing *EnforceMultiPathsNoBlocking* as parameter to *Iterate*, we obtain an algorithm that establishes the Multipath Blocking Vertices Property. For easier reference, we set *EstablishMultiPathsNoBlocking*( $G, \pi$ ) to *Iterate*( $G, \pi, \text{EnforceMultiPathsNoBlocking}$ ).

[Lemma 4.15](#) in combination with [Proposition 4.1](#) yields:

---



---

**Corollary 4.19**

*EstablishMultiPathsNoBlocking*( $G, \pi$ ) runs in time  $\mathcal{O}(n \cdot m \cdot \kappa(n, m))$ .

---

As already mentioned in [Section 4.6](#),  $\kappa(n, m) \in \mathcal{O}(m \cdot \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ .

### 4.7.3 Again an $\mathcal{NP}$ -hard Extension

At the end of the previous section, we introduced the Extended Multipath Property, which is stronger than the Multipath Property, but not efficiently enforceable anymore. A similar extension is possible for the Multipath Blocking Vertices Property, which uses the following statement:

---



---

**Lemma 4.16**

Let  $\mathcal{B}$  be the set of backward arcs induced by a linear ordering  $\pi$  of a graph  $G$  such that  $\pi$  respects the Multipath Blocking Vertices Property and  $b = (u, w) \in \mathcal{B}$ . Let  $v$  be a vertex on a forward path  $P_b$  for  $b$  that contains no left-blocking vertex and let  $Y \subseteq \mathcal{B}^+(v)$  or  $Y \subseteq \mathcal{B}^-(v)$  be a set of either incoming or outgoing backward arcs of  $v$ . Furthermore, let  $Z \subseteq V$  be a set of left-blocking vertices that contains neither  $w$  nor, if  $Y \subseteq \mathcal{B}^+(v)$ , the head of any arc in  $Y$ . If there is a set  $X$  of forward arcs induced by  $\pi_{\text{sp}/l}$  that covers every forward path of a backward arc in  $Y \cup [b]_{\parallel}$  in  $\pi_{\text{sp}/l}$ ,  $Z$  covers every forward path of a backward arc in  $Y \cup [b]_{\parallel}$  in  $\pi$  that contains a left-blocking vertex, and  $|X| \leq |Y|$ , then  $\mathcal{B} \setminus (\mathcal{B}^+[Z] \cup Y \cup [b]_{\parallel}) \cup \mathcal{F}^-[Z] \cup X$  is feasible.

Likewise, let  $v$  be a vertex on a forward path  $P_b$  for  $b$  that contains no right-blocking vertex and let  $Y \subseteq \mathcal{B}^+(v)$  or  $Y \subseteq \mathcal{B}^-(v)$  be a set of either incoming or outgoing backward arcs of  $v$ . Furthermore, let  $Z \subseteq V$  be a set of right-blocking vertices that contains neither  $u$  nor, if  $Y \subseteq \mathcal{B}^-(v)$ , the tail of any arc in  $Y$ . If there is a set  $X$  of forward arcs induced by  $\pi_{\text{sp}/r}$  that covers every forward path of a backward arc in  $Y \cup [b]_{\parallel}$  in  $\pi_{\text{sp}/r}$ ,  $Z$  covers every forward path of a backward arc in  $Y \cup [b]_{\parallel}$  in  $\pi$  that contains a right-blocking vertex, and  $|X| \leq |Y|$ , then  $\mathcal{B} \setminus (\mathcal{B}^-[Z] \cup Y \cup [b]_{\parallel}) \cup \mathcal{F}^+[Z] \cup X$  is feasible.

---

*Proof.* Let  $G = (V, A)$  and denote by  $\mathcal{F} = A \setminus \mathcal{B}$  the set of forward arcs induced by  $\pi$ . We consider first the left-blocking case. As  $v$  is on the forward path  $P_b = w \rightsquigarrow u$  in  $\pi_{\text{sp}/l}$ ,  $P_b$  consists of two subpaths  $P'_b = w \rightsquigarrow v$  and  $P''_b = v \rightsquigarrow u$ . Furthermore,  $v$  is not left-blocking by the lemma's preconditions.

As  $\pi$  respects the Multipath Blocking Vertices Property,  $\pi_{\text{sp}/l}$  respects the Multipath Property by [Lemma 4.14](#). Consequently, there must be pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}(v) = \mathcal{B}^+(v) \cup \mathcal{B}^-(v)$  in  $\pi_{\text{sp}/l}$ , which in particular also holds for those in  $Y$ . Hence, every set of forward arcs that covers all forward paths for the backward arcs in  $Y$  must have cardinality at least  $|Y|$ . As  $|X| \leq |Y|$ , we can conclude that  $|X| = |Y|$  and that every arc in  $X$  is part of at least one forward path for a backward arc in  $Y$  in  $\pi_{\text{sp}/l}$ .

Next, consider the set of left-blocking vertices  $Z$ . By [Lemma 4.7](#),

$$\mathcal{B}' = \mathcal{B} \setminus \mathcal{B}^+[Z] \cup \mathcal{F}^-[Z]$$



is feasible,  $|\mathcal{B}| = |\mathcal{B}'|$  and every path consisting only of arcs in

$$\mathcal{F}' = A \setminus \mathcal{B}' = \mathcal{F} \setminus \mathcal{F}^-[Z] \cup \mathcal{B}^+[Z]$$

that contains a vertex in  $Z$  must also start at a vertex in  $Z$ . Let  $(t, h) \in Y \cup [b]_{\parallel}$  and suppose there is a path  $P = h \rightsquigarrow t$  that uses only arcs in  $\mathcal{F}'$ . If  $P$  contains a vertex in  $Z$ , then, due to Lemma 4.7,  $h \in Z$ , a contradiction to our precondition that  $Z$  contains neither  $w$  nor the head of any arc in  $Y$  if  $(t, h) \in \mathcal{B}^+(v)$  and to  $v$  being not left-blocking if  $(t, h) \in \mathcal{B}^-(v)$ , i. e.,  $h = v$ . Consequently,  $P$  contains only arcs in  $\mathcal{F}$  and no vertex in  $Z$ , which in turn implies that  $P$  is covered by  $X$ .

Let now

$$\mathcal{B}'' = \mathcal{B}' \setminus (Y \cup [b]_{\parallel}) \cup X$$

and

$$\mathcal{F}'' = A \setminus \mathcal{B}'' = \mathcal{F}' \setminus X \cup Y \cup [b]_{\parallel}.$$

Suppose there is a cycle  $C$  in  $G|_{\mathcal{F}''}$ . W.l.o.g., we assume that  $C$  is simple. As  $\mathcal{B}'$  is feasible,  $G|_{\mathcal{F}'}$  is acyclic, so  $C$  cannot only consist of arcs in  $\mathcal{F}' \setminus X$ . Hence,  $C$  must contain at least two arcs from  $Y \cup [b]_{\parallel}$ : If  $C$  contained just one arc  $(t, h) \in [b]_{\parallel} \cup Y$ , then there must be a path  $h \rightsquigarrow t$  in  $\mathcal{F}' \setminus X$ , a contradiction to our conclusion in the previous paragraph. Furthermore, if  $C$  contains two or more parallel arcs in  $[b]_{\parallel}$ , it is not simple. Hence,  $C$  must use at least one arc of  $Y$ . Suppose  $C$  uses neither  $b$  nor one of its parallel arcs. In case that  $Y \subseteq \mathcal{B}^+(v)$ , this implies that  $C$  consists of a set of arcs  $Y' = \{(v, h_i) \mid 0 \leq i < k\} \subseteq Y$  as well as a set of paths  $\mathcal{P} = \{h_i \rightsquigarrow v \mid 0 \leq i < k\}$ , where  $k \geq 2$ . Otherwise, if  $Y \subseteq \mathcal{B}^-(v)$ ,  $C$  analogously consists of a set of  $k \geq 2$  arcs  $Y' = \{(t_i, v) \mid 0 \leq i < k\} \subseteq Y$  as well as a set of paths  $\mathcal{P} = \{v \rightsquigarrow t_i \mid 0 \leq i < k\}$ . In both cases, the paths in  $\mathcal{P}$  use only arcs in  $\mathcal{F}' \setminus X$ , which once more contradicts our conclusion in the previous paragraph.

Hence, exactly one arc of  $[b]_{\parallel}$  and  $k \geq 1$  arcs of  $Y$  must be part of  $C$ . Consider the case that  $Y \subseteq \mathcal{B}^+(v)$  and let  $Y' = \{(v, h_i) \mid 0 \leq i < k\} \subseteq Y$  be the subset of arcs in  $Y$  that is used in  $C$ . Then,  $C$  must also contain subpaths  $h_j \rightsquigarrow u$  for some  $j \in \{0, \dots, k-1\}$  and, more importantly, a subpath  $P' = w \rightsquigarrow v$ , such that all use only arcs in  $\mathcal{F}' \setminus X$ . Recall that every arc in  $X$  is also part of a forward path for a backward arc in  $Y$  in  $\pi_{\text{sp}/l}$  and all of these forward paths end at  $v$ . Hence,  $P''_b = v \rightsquigarrow u$  uses neither an arc in  $X$  nor does it contain a vertex in  $Z$ . The same applies for  $P'$ , which yields that they together form a path  $w \rightsquigarrow v \rightsquigarrow u$  that contains neither a left-blocking vertex nor an arc in  $X$  and is a forward path for  $b$  in  $\pi$ , a contradiction. For the alternative case that  $Y \subseteq \mathcal{B}^-(v)$ , let  $Y' = \{(t_i, v) \mid 0 \leq i < k\} \subseteq Y$  be the subset of arcs in  $Y$  that is used in  $C$ . As above,  $C$

must then contain a subpath  $w \rightsquigarrow t_j$ , for some  $j \in \{0, \dots, k-1\}$ , as well as a subpath  $P' = v \rightsquigarrow u$ , and both use only arcs in  $\mathcal{F}' \setminus X$ . Every arc in  $X$  must again also be part of a forward path for a backward arc in  $Y$  in  $\pi_{\text{sp}/l}$ , all of which start at  $v$ . Thus,  $P'_b = w \rightsquigarrow v$  can neither contain an arc in  $X$  nor a vertex in  $Z$ . Hence, the combination of  $P'_b$  and  $P'$  results in a path  $w \rightsquigarrow v \rightsquigarrow u$ , which is a forward path for  $b$  in  $\pi$  that is not covered by  $X$  and does not pass through a left-blocking vertex, a contradiction.

Consequently,  $G|_{\mathcal{F}''}$  is acyclic, which yields that

$$\begin{aligned} \mathcal{B}'' &= \mathcal{B}' \setminus (Y \cup [b]_{\parallel}) \cup X \\ &= (\mathcal{B} \setminus \mathcal{B}^+[Z] \cup \mathcal{F}^-[Z]) \setminus (Y \cup [b]_{\parallel}) \cup X \\ &= \mathcal{B} \setminus (\mathcal{B}^+[Z] \cup Y \cup [b]_{\parallel}) \cup \mathcal{F}^-[Z] \cup X \end{aligned}$$

is feasible.

The analogous statement using right-blocking vertices follows once more by considering the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^R$ .  $\square$

Note that for an application of [Lemma 4.12](#) in the proof of [Lemma 4.16](#), we would have needed a linear ordering that induces the intermediate set of backward arcs  $\mathcal{B}'$  and, most notably, respects the Multipath Property, which we cannot guarantee in general.

Using [Lemma 4.16](#), we can strengthen the Multipath Blocking Vertices Property as follows:

---

**Lemma 4.17**      EXTENDED MULTIPATH BLOCKING VERTICES PROPERTY

Let  $\pi^*$  be an optimal linear ordering of a graph  $G$  and let  $b = (u, w)$  be a backward arc induced by  $\pi^*$ . If  $v$  is a vertex on a forward path  $P_b$  for  $b$  in  $\pi_{\text{sp}/l}^*$  ( $\pi_{\text{sp}/r}^*$ ) and  $u$  and  $w$  are not themselves left-blocking (right-blocking), then there is a set of pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^+(v) \cup [b]_{\parallel}$  as well as one for the backward arcs in  $\mathcal{B}^-(v) \cup [b]_{\parallel}$  in  $\pi_{\text{sp}/l}^*$  ( $\pi_{\text{sp}/r}^*$ ).

---

*Proof.* Let  $G = (V, A)$  and denote by  $\mathcal{B}$  and  $\mathcal{F}$  the set of backward and forward arcs induced by  $\pi^*$ , respectively. Note that if  $b \in \mathcal{B}(v)$ , then the pairwise arc-disjoint forward paths for  $\mathcal{B}^+(v) \cup [b]_{\parallel} = \mathcal{B}^+(v)$  and  $\mathcal{B}^-(v) \cup [b]_{\parallel} = \mathcal{B}^-(v)$  are already implied by [Lemma 4.14](#). Hence, assume in the following that  $b \notin \mathcal{B}(v)$ .

As  $\pi^*$  is optimal and hence respects the Multipath Blocking Vertices Property, both  $\pi_{\text{sp}/l}^*$  and  $\pi_{\text{sp}/r}^*$  respect the Multipath Property. Recall that, due to the identification of

arcs before and after a vertical split,  $\pi_{\text{sp}/l}^*$  and  $\pi_{\text{sp}/r}^*$  induce the same sets of backward and forward arcs as  $\pi^*$ .

Suppose there is a backward arc  $b = (u, w) \in \mathcal{B}$  and a vertex  $v$  on a forward path  $P_b$  of  $b$  in  $\pi_{\text{sp}/l}$  such that  $u$  and  $w$  are not left-blocking and no set of pairwise arc-disjoint forward paths for the backward arcs in  $\mathcal{B}^+(v) \cup [b]_{\parallel}$  or for the backward arcs in  $\mathcal{B}^-(v) \cup [b]_{\parallel}$  exists. Then, there must be a set  $Y \subseteq \mathcal{B}^+(v)$  or  $Y \subseteq \mathcal{B}^-(v)$ , respectively, as well as a directed multicut  $X \subseteq \mathcal{F}$  such that  $|X| \leq |Y| + |[b]_{\parallel}|$ , and  $X$  covers all forward paths for the backward arcs in  $Y \cup [b]_{\parallel}$  in  $\pi_{\text{sp}/l}^*$ .

Identify a set  $Z$  of left-blocking vertices by traversing all cropped forward paths for the backward arcs in  $Y$  that do not contain an arc of  $X$  and placing the first left-blocking vertex that occurs in  $Z$ . If  $Y \subseteq \mathcal{B}^+(v)$ , we traverse all forward paths backwards, i. e., starting at  $v$ , whereas if  $Y \subseteq \mathcal{B}^-(v)$ , we use the usual direction (and thereby also start at  $v$ ). Proceed likewise in both cases for all cropped forward paths for  $b$  that do not contain an arc in  $X$  and add the left-blocking vertices encountered first to  $Z$ . As  $u$ ,  $v$ , and  $w$  are not left-blocking, they cannot be in  $Z$ . Assume that  $Y \subseteq \mathcal{B}^+(v)$ . Then, there also is no head of a backward arc in  $Y$  contained in  $Z$  (cf. also the proof of [Lemma 4.14](#)): In this case, there would have been backward arcs  $(v, h)$  and  $(v, h')$  such that, w. l. o. g.,  $h'$  is left-blocking and part of a cropped forward path  $P$  for  $(v, h)$ . Then, however,  $P$  has a subpath which is a cropped forward path for  $(v, h')$ , which either contains an arc of  $X$  or a left-blocking vertex  $z \in Z$ . In the former case,  $P$  also contains an arc of  $X$  and is therefore not considered, whereas in the latter,  $z$  must have been encountered before  $h'$  on a backward traversal of  $P$  and we only collect the first left-blocking vertex that occurs to create  $Z$ . The analogous argument yields that if  $Y \subseteq \mathcal{B}^-(v)$ , then no tail of a backward arc in  $Y$  can be contained in  $Z$ .

As  $|X| \leq |Y|$ , we can apply [Lemma 4.16](#) both if  $Y \subseteq \mathcal{B}^+(v)$  and if  $Y \subseteq \mathcal{B}^-(v)$ , which yields that

$$\mathcal{B}' = \mathcal{B} \setminus \left( \mathcal{B}^+[Z] \cup Y \cup [b]_{\parallel} \right) \cup \mathcal{F}^-[Z] \cup X$$

is feasible. As  $|[b]_{\parallel}| \geq 1$ ,  $\mathcal{B}^+[Z] = \mathcal{F}^-[Z]$  by [Proposition 4.2](#), and all sets are pairwise disjoint,  $|\mathcal{B}^+[Z] \cup Y \cup [b]_{\parallel}| > |\mathcal{F}^-[Z] \cup X|$ , which implies that  $|\mathcal{B}'| < |\mathcal{B}|$ , a contradiction to  $\pi^*$  being optimal.



The analogous statement for right-blocking vertices follows immediately from the left-blocking case by considering the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^R$  instead.  $\square$

For the same reason as for the Extended Multipath Property, the proof of the Extended Multipath Blocking Vertices Property does not suggest an efficient algorithm to check or enforce it, due to the  $\mathcal{NP}$ -hardness of the DIRECTED MULTICUT problem (cf. Section 4.6.4). In particular their preparatory lemmas, Lemma 4.12 and Lemma 4.16, may however be useful in case that a directed minimum cut has already been obtained by some other procedure. In this case, weaker versions of the Extended Multipath Property and the Extended Multipath Blocking Vertices Property may be implemented in polynomial time. In fact, two such properties are employed in Section 5.4.1.

## 4.8 Eliminator Layouts Property

The next property that is discussed in this chapter differs from all previous ones in an important aspect: If it is violated, then the linear ordering is not necessarily improved during the establishment; the cardinality of the induced set of backward arcs may also remain constant. There is some revenue after all: First, enforcing it may destroy one of the aforementioned properties and thereby indirectly yield an improvement of the linear ordering. Second, it can also be applied to linear orderings that are already optimal and thus limits the number of orderings to consider when looking for an optimal solution (cf. Chapter 5, Chapter 6).

### 4.8.1 Eliminator Layouts

An eliminator layout can be regarded as a special form of the layout of a blocking vertex. Consider the layout  $\mathcal{L}(v)$  of a vertex  $v$  according to a linear ordering. Recall that  $v$  is called left-blocking, if  $f^-(v) = b^+(v)$  and  $f^-(v), f^+(v), b^+(v) \geq 1$ . Accordingly,  $v$  is called right-blocking, if  $f^+(v) = b^-(v)$  and  $f^-(v), f^+(v), b^-(v) \geq 1$ . We now add a further constraint. A vertex  $v$  has an *eliminator* layout, if either  $f^-(v), f^+(v), b^+(v) \geq 1$  with  $f^-(v) = b^+(v)$  and  $b^-(v) = 0$ , or  $f^-(v), f^+(v), b^-(v) \geq 1$  with  $f^+(v) = b^-(v)$ , and  $b^+(v) = 0$ . More descriptively, if  $v$ 's layout is eliminator, then  $v$  is either left-blocking with additionally  $b^-(v) = 0$ , or  $v$  is right-blocking with  $b^+(v) = 0$ . In pictograms, an eliminator layout is depicted as  or , respectively.

As the name suggests, we want to eliminate these layouts, i. e., we aim at constructing a linear ordering that does not induce an eliminator layout on any vertex, and at the same time without increasing the number of backward arcs. An advantage of this

task over the Blocking Vertices Property is that we do not have to decide between two versions (left and right), but can eliminate both types simultaneously<sup>1</sup>:

---

**Lemma 4.18** ELIMINABLE LAYOUTS PROPERTY  
 For every optimal linear ordering  $\pi^*$  of  $G$  there is a linear ordering  $\pi^{*'}$  with  $|\pi^{*'}| = |\pi^*|$  such that  $\pi^{*'}$  does not induce an eliminable layout on any vertex.

---

*Proof.* Let  $\pi$  be a linear ordering of  $G$  that respects the Nesting Property (cf. Lemma 4.3). Recall from Section 3.3 that the layout  $\mathcal{L}(v)$  of a vertex  $v$  is defined as the four-tuple  $(f^-(v), f^+(v), b^-(v), b^+(v))$ . We will make use of this notation in the proof.



Consider a vertex  $v$  with layout  $\mathcal{L}(v) = (x, y, y, 0)$  for some  $x, y \geq 1$  (). Then,  $v$  is right-blocking and its layout is eliminable by the above definition. For easier reference, we name the tails of  $v$ 's incoming backward arcs  $u_i$  in ascending order with respect to the linear ordering, i. e., we obtain an indexed family  $\{u_i\}_{0 \leq i < y}$  with  $\pi(u_i) \leq \pi(u_{i+1})$ ,  $\forall 0 \leq i < y - 1$ , and  $(u_i, v) \in \mathcal{B}^-(v)$ ,  $\forall 0 \leq i < y$ . In consequence of  $\pi$  adhering to the Nesting Property, there is an injective mapping  $\mu_h$  that assigns a forward arc  $(v, w_i) \in \mathcal{F}^+(v)$  to each backward arc  $(u_i, v) \in \mathcal{B}^-(v)$ ,  $\forall 0 \leq i < y$ . Furthermore,  $\forall 0 \leq i < y$ ,  $\pi(v) < \pi(w_i) < \pi(u_i)$ . As  $b^-(v) = f^+(v)$  and the mapping  $\mu_h$  is injective, every arc in  $\mathcal{F}^+(v)$  is assigned to a unique backward arc.

Figure 4.15(a) shows a subsequence of a linear ordering that we will use as an example in the following. It is important to note that the figure does not show a complete linear ordering. There must be vertices to the left of  $v$  because  $v$  has incoming forward arcs and there may also be vertices between those depicted or to the right of the rightmost vertex.

Consider now the following operation: Modify  $\pi$  by moving  $v$  to some position at least  $\pi(u_{y-1})$ , e. g., by applying the routine  $Move(\pi, v, \pi(u_{y-1}))$ . In the example,  $y = 2$ , so we obtain the situation depicted in Figure 4.15(b) after moving  $v$  to  $\pi(u_1)$ .

We can now observe that in the new linear ordering, let us call it  $\pi'$  to distinguish it from the initial linear ordering  $\pi$ , all arcs in  $\mathcal{B}^-(v)$  count as forward arcs whereas all arcs in  $\mathcal{F}^+(v)$  have become backward arcs. From now on, we indicate with subscript  $\pi$  or  $\pi'$  which linear ordering a set or quantity refers to.

As both  $\mathcal{B}_\pi^-(v)$  and  $\mathcal{F}_\pi^+(v)$  contain  $y$  arcs, the cardinality of the induced set of backward arcs remains unchanged in  $\pi'$ . The layout of  $v$  in  $\pi'$  is  $\mathcal{L}_{\pi'}(v) = (x + y, 0, 0, y)$  () and is therefore no longer eliminable. However, also the layouts of  $u_i$  and  $w_i$ ,  $\forall 0 \leq i < y$ , have changed, so we investigate these next.

<sup>1</sup>A weaker version of this result has also been published in an earlier conference article [HBA13].

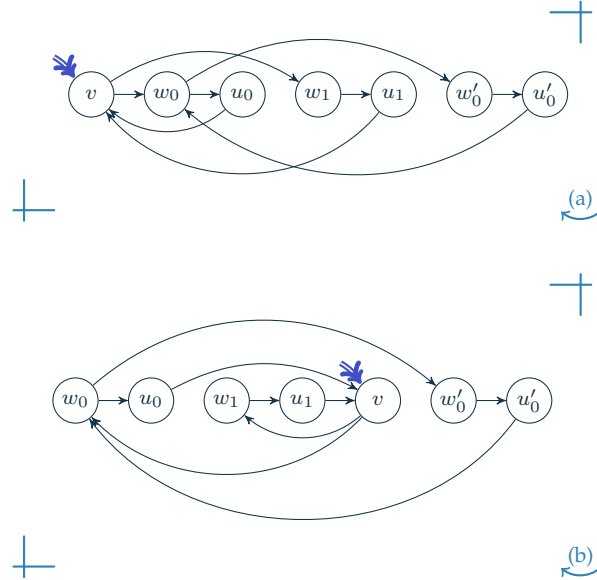









Figure 4.15: Excerpt from a linear ordering that induces an eliminable layout on vertex  $v$  (a) and the linear layout resulting from moving  $v$  (b).




Let us first consider some vertex  $u \in \{u_i\}_{0 \leq i < y}$ . Suppose that the layout of  $u$  changed to  $\begin{smallmatrix} \swarrow & \circ & \searrow \\ \nwarrow & & \nearrow \end{smallmatrix}$  in  $\pi'$ , i. e.,  $\mathcal{L}_{\pi'}(u) = (x_u, y_u, y_u, 0)$  for  $x_u, y_u \geq 1$ . The movement of  $v$  can only affect the classification of arcs incident to  $v$ , hence, it suffices to consider the only arc incident to both  $u$  and  $v$ , which is the former backward arc  $(u, v)$ . As  $(u, v) \in \mathcal{B}_{\pi}^+(u)$  before and  $(u, v) \in \mathcal{F}_{\pi'}^+(u)$  afterwards,  $b_{\pi}^+(u) = b_{\pi'}^+(u) + 1$  and  $f_{\pi}^+(u) = f_{\pi'}^+(u) - 1$ . Consequently, the layout of  $u$  with respect to the initial linear ordering  $\pi$  is  $\mathcal{L}_{\pi}(u) = (x_u, y_u - 1, y_u, 1)$  ( $\begin{smallmatrix} \swarrow & \circ & \searrow \\ \nwarrow & & \nearrow \end{smallmatrix}$ ). Then, however,  $b_{\pi}^-(u) = y_u > y_u - 1 = f_{\pi}^+(u)$ , a contradiction to [Corollary 4.3](#) and the assumption that  $\pi$  respects the Nesting Property. So  $\pi'$  cannot induce the eliminable layout type  $\begin{smallmatrix} \swarrow & \circ & \searrow \\ \nwarrow & & \nearrow \end{smallmatrix}$  on any vertex  $u \in \{u_i\}_{0 \leq i < y}$ .


Suppose that the layout of  $u$  is  $\begin{smallmatrix} \swarrow & \circ & \searrow \\ \nwarrow & & \nearrow \end{smallmatrix}$  in  $\pi'$ , i. e.,  $\mathcal{L}_{\pi'}(u) = (y_u, x_u, 0, y_u)$  for any  $u \in \{u_i\}_{0 \leq i < y}$  and  $x_u, y_u \geq 1$ . With the same argument, we obtain  $b_{\pi}^+(u) = b_{\pi'}^+(u) + 1$  and  $f_{\pi}^+(u) = f_{\pi'}^+(u) - 1$ , so the layout of  $u$  with respect to the initial linear ordering  $\pi$  is  $\mathcal{L}_{\pi}(u) = (y_u, x_u - 1, 0, y_u + 1)$  ( $\begin{smallmatrix} \swarrow & \circ & \searrow \\ \nwarrow & & \nearrow \end{smallmatrix}$ ). Then,  $b_{\pi}^+(u) = y_u + 1 > y_u = f_{\pi}^-(u)$ , again a contradiction to [Corollary 4.3](#) and  $\pi$  respecting the Nesting Property. Consequently,  $\pi'$  also cannot induce the eliminable layout  $\begin{smallmatrix} \swarrow & \circ & \searrow \\ \nwarrow & & \nearrow \end{smallmatrix}$  on any vertex  $u \in \{u_i\}_{0 \leq i < y}$ .

Next, we turn our attention to some vertex  $w \in \{w_i\}_{0 \leq i < y}$ . Here, the only arc whose classification is affected by the movement of  $v$  is the arc  $(v, w)$ . For this arc holds that  $(v, w) \in \mathcal{F}_{\pi}^-(w)$  and  $(v, w) \in \mathcal{B}_{\pi'}^-(w)$ . Hence, after the movement of  $v$ ,  $w$  has at least one

incoming backward arc, so  as layout of  $w$  with respect to  $\pi'$  can be discarded. It may be the case, however, that  $\mathcal{L}_{\pi'}(w) = (x_w, y_w, y_w, 0)$  (). As  $(v, w)$  changes from incoming forward to incoming backward with respect to  $w$ , we have  $b_{\pi'}^-(w) = b_{\pi'}^-(w) - 1$  and  $f_{\pi'}^-(w) = f_{\pi'}^-(w) + 1$ . Then, the layout of  $w$  according to the initial linear ordering  $\pi$  would have been  $\mathcal{L}_{\pi}(w) = (x_w + 1, y_w, y_w - 1, 0)$  () , which does not contradict the Nesting Property. Subsequently, the elimination of the layout  at  $v$  may in turn produce this layout on vertices in  $\{w_i\}_{0 \leq i < y}$ . We therefore have to iterate the movement process in order to eliminate the layout also on these vertices until the constructed linear ordering either no longer induces an elimidable layout of type  on any vertex or it violates [Corollary 4.3](#).

Assume that the elimination of  is implemented as a sweep line algorithm that starts at the first vertex of  $\pi$ , i. e.,  $\pi^{-1}(0)$ , and processes the vertices in ascending order according to  $\pi$ . Whenever the Nesting Property is violated during the procedure it may be re-established, which produces a linear ordering that induces strictly fewer backward arcs. In this case, the sweep line algorithm starts anew. So if the procedure reaches the  $q$ th vertex in  $\pi$ , we have as an invariant that none of the vertices  $v'$  with  $\pi(v') < q$  have layout . Observe that if the vertex at position  $q$  is moved, the vertex that formerly was at position  $q + 1$  takes position  $q$  afterwards, so the sweep line does not necessarily advance its position in every step. Once the sweep line reaches the vertex at the last position, the elimination process is finished.

Suppose that the procedure does not terminate. Then, the sweep line is stuck at some position  $q < n - 1$ , i. e., the vertex at position  $q$  has layout , and each time after eliminating the layout on this vertex, there is another vertex at the same position  $q$  with layout . As this process continues forever, but there are only  $n$  vertices, there must be a vertex  $v$  that reoccurs infinitely often at position  $q$ . Consider a linear ordering  $\pi^{(v \rightarrow q)}$  that is obtained from  $\pi$  by move operations, such that  $v$  is at position  $q$  and the sweep line cannot advance beyond  $q$ . For the sake of clarity, note that  $\pi^{(v \rightarrow q)}$  is not unique, i. e., there may be more than one such linear ordering. We simply pick one of them here. As already argued,  $v$  must have an elimidable layout with  $\mathcal{L}_{\pi^{(v \rightarrow q)}}(v) = (x, y, y, 0)$  (). Let  $\{u_i\}_{0 \leq i < y}$  and  $\{w_i\}_{0 \leq i < y}$  be defined with respect to  $v$  as at the beginning of the proof.

Consider again the elimination operation, which moves  $v$  beyond all vertices that have an outgoing backward arc to  $v$  or an incoming forward arc from  $v$ . If the sweep line cannot advance to a position greater than  $q$  and if  $v$  reoccurs at  $q$  infinitely often, then all vertices  $\{u_i\}_{0 \leq i < y}$  and  $\{w_i\}_{0 \leq i < y}$  must also take position  $q$  with layout  between any two reoccurrences of  $v$  at  $q$ . Let  $X_v = \{u_i\}_{0 \leq i < y} \cup \{w_i\}_{0 \leq i < y}$ .



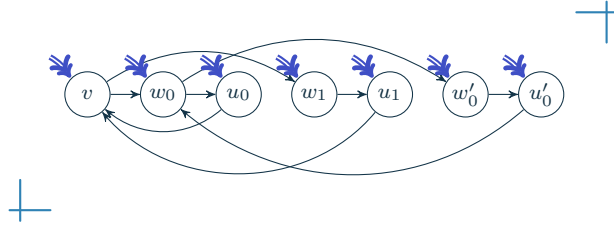


Figure 4.16: Excerpt from a linear ordering where the elimination process does not terminate.

Next, consider a vertex  $v' \in X_v$ . As we just argued, also  $v'$  occurs infinitely often at position  $q$  with eliminable layout  $\begin{smallmatrix} \bullet & \rightarrow \\ \circ & \\ \bullet & \leftarrow \end{smallmatrix}$ . Let  $\pi^{(v' \rightarrow q)}$  be a linear ordering obtained from  $\pi^{(v \rightarrow q)}$  by move operations and such that  $v'$  is at position  $q$ . Let  $\mathcal{L}_{\pi^{(v' \rightarrow q)}}(v') = (x', y', y', 0)$  ( $\begin{smallmatrix} \bullet & \rightarrow \\ \circ & \\ \bullet & \leftarrow \end{smallmatrix}$ ). Denote again by  $\{u'_i\}_{0 \leq i < y'}$  and  $\{w'_i\}_{0 \leq i < y'}$  the set of adjacent vertices of  $v'$  in analogy to the naming when considering  $v$ . As  $v'$  also reoccurs at position  $q$ , so must  $\{u'_i\}_{0 \leq i < y'}$  and  $\{w'_i\}_{0 \leq i < y'}$  by the same argument as above.




We can describe the set  $X$  of vertices that reoccur at position  $q$  as a transitive closure of  $X_v$ , i. e.,  $X = \{v\} \cup \bigcup_{v' \in X} X_{v'}$ . Figure 4.16 depicts an excerpt of a linear ordering where the sweep line is stuck at the leftmost position that is shown. In this example,  $X = \{v, u_0, u_1, w_0, w_1, u'_0, w'_0\}$ .

Next, we want to analyze the subgraph  $G_X$  of  $G$  induced by  $X$ . Consider some vertex  $v \in X$ . By definition,  $v$  takes position  $q$  infinitely often and with eliminable layout  $\begin{smallmatrix} \bullet & \rightarrow \\ \circ & \\ \bullet & \leftarrow \end{smallmatrix}$ . Let  $\pi^{(v \rightarrow q)}$  be defined as above. Then,  $\mathcal{B}_{\pi^{(v \rightarrow q)}}^+(v) = \emptyset$  and  $\{w \mid (v, w) \in \mathcal{F}_{\pi^{(v \rightarrow q)}}^+(v)\} \subset X$ . Consequently, the head of every outgoing arc of  $v$  is also in  $X$ . As for the incoming arcs of  $v$ , we find that because  $v$  has layout  $\begin{smallmatrix} \bullet & \rightarrow \\ \circ & \\ \bullet & \leftarrow \end{smallmatrix}$  in  $\pi^{(v \rightarrow q)}$ ,  $\mathcal{F}_{\pi^{(v \rightarrow q)}}^-(v) \neq \emptyset$ . Hence, there must be at least one vertex  $z$  with  $\pi^{(v \rightarrow q)}(z) < q$ . This implies that the sweep line has already passed  $z$ , so  $z \notin X$ . Therefore,  $X$  must be a proper subset of  $V$ . Furthermore, there is a path  $z \rightsquigarrow v$  in  $G$  consisting of the arc  $(z, v) \in \mathcal{F}_{\pi^{(v \rightarrow q)}}^-(v)$ , but there is no path  $v \rightsquigarrow z$  in the opposite direction:  $v \in X$  and the head of every outgoing arc of a vertex in  $X$  is also in  $X$ , but  $z \notin X$ . However,  $G$  is strongly connected, a contradiction. Subsequently, there must be at least one vertex in  $X$  that has not a layout of type  $\begin{smallmatrix} \bullet & \rightarrow \\ \circ & \\ \bullet & \leftarrow \end{smallmatrix}$  when taking position  $q$ , at which point the sweep line can advance.

Finally, suppose that the initial linear ordering  $\pi$  was optimal. Recall that if during the elimination process a linear ordering is obtained that contradicts Corollary 4.3, the Nesting Property is violated and its re-establishment yields a linear ordering  $\pi'$  with  $|\pi'| < |\pi|$ . If  $\pi$  was optimal, this would be a contradiction. Hence, Corollary 4.3 holds for



all linear orderings throughout the elimination process and it terminates with a linear ordering  $\pi''$  such that  $|\pi''| = |\pi|$ .

For vertices with layout , the statement follows as in the case of the Blocking Vertices Property by considering the vertices of type  in the reverse graph along with the reverse linear ordering. Alternatively, the layout  could be eliminated analogously using a sweep line that processes the vertices in descending order according to the linear ordering  $\pi$ . Here, for a vertex  $v$  with eliminable layout, the vertices  $\{u_i\}_{0 \leq i < y}$  must be defined as the heads of  $v$ 's outgoing backward arcs and  $\{w_i\}_{0 \leq i < y}$  as the tails of  $v$ 's corresponding incoming forward arcs. If we again assume a sorting such that  $\pi(u_i) \leq \pi(u_{i+1})$  for all  $0 \leq i < y - 1$ , then  $v$  must be moved to a position less than or equal to  $\pi(u_0)$ .  $\square$

Let  $\text{Elim}(\pi)$  the predicate that indicates whether a linear ordering  $\pi$  respects the Eliminator Layouts Property. By [Lemma 4.18](#), we have:

---



---



**Corollary 4.20**


For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \text{Elim}(\pi)$ .

---

From here on, we stipulate that if a linear ordering  $\pi$  respects the Eliminator Layouts Property, then it does not induce an eliminable layout on any vertex.

### 4.8.2 The Elimination Operation

In the proof of [Lemma 4.18](#), an elimination operation has been described and it has been shown that this process always terminates. However, none has been said so far about the number of steps it takes, which will be the subject of analysis in this section. For ease of notation and understanding, the following definitions all refer to eliminable layouts of type , but can be made likewise for layouts of type .

Let  $\pi$  be a linear ordering of a graph  $G$  and  $0 \leq q \leq n - 1$  be a position within  $\pi$ . We define the elimination operation on the vertex at position  $q$  as a function  $\text{elim}_q(\pi)$ . Assume that  $v$  is the vertex at position  $q$ , i. e.,  $\pi(v) = q$ . Then,  $\text{elim}_q(\pi)$  returns the linear ordering obtained from moving  $v$  to a certain position  $p$  (which we will address shortly), provided that  $\pi$  induces a layout of type  on  $v$ . Otherwise,  $\text{elim}_q(\pi) = \pi$ . In the proof of [Lemma 4.18](#),  $p$  was defined to be such that  $p \geq \max_{(u,v) \in \mathcal{B}^-(v)} \pi(u)$  and it was suggested to actually use the smallest possible value of  $p$ . For reasons that will become clear later in this subsection, we refrain from adopting this proposal and simply set

$p = n - 1$ , which always satisfies the condition. We can hence define  $\text{elim}_q(\pi)$  formally as:

$$\text{elim}_q(\pi) = \begin{cases} \text{Move}(\pi, v_q, n - 1), & \text{if } \mathcal{L}_\pi(v_q) \text{ is of type } \begin{array}{c} \text{blue star} \\ \text{red arrow} \end{array}, \\ \pi, & \text{else.} \end{cases}$$

It has already been pointed out in the proof of [Lemma 4.18](#) that the position  $q$  in the linear ordering serves as a barrier in the elimination operation: neither can a vertex that resides at a position smaller than  $q$  in  $\pi$  be moved to a position at least  $q$  in  $\text{elim}_q(\pi)$ , nor vice versa. More precisely, every vertex  $v$  with  $\pi(v) < q$  remains at its position during the elimination operation:

---

**Proposition 4.3**

Let  $\pi$  be a linear ordering of a graph  $G$  and  $q \in \{0, \dots, n - 1\}$ . For every vertex  $v \in V$  holds: If  $\pi(v) < q$ , then  $\text{elim}_q(\pi)(v) = \pi(v)$ , otherwise,  $\text{elim}_q(\pi)(v) \geq q$ .

Using functional powers, we are able to describe any number  $i$  of successive elimination operations at position  $q$  as the  $i$ th iterate of  $\text{elim}_q(\pi)$ . This implies that we obtain for  $i = 0$  the identity function, i. e. ,

$$\text{elim}_q^0(\pi) = \pi,$$

for  $i = 1$ ,

$$\text{elim}_q^1(\pi) = \text{elim}_q(\pi),$$


and for any  $i \geq 2$ ,

$$\text{elim}_q^i(\pi) = (\text{elim}_q \circ \text{elim}_q^{i-1})(\pi),$$

where  $f \circ g$  denotes the standard function composition of two functions  $f$  and  $g$ .

In the next step, we introduce a notation to list the vertices occurring at a specific position within  $\pi$ . To this end, let  $\mathcal{I}_\pi(q)$  be a sequence of vertices  $s_i$  such that  $s_i$  takes position  $q$  after the  $i$ th elimination operation on  $q$ . As shown in the proof of [Lemma 4.18](#), this sequence must eventually encounter a vertex whose layout is not eliminable at position  $q$ . From this point on,  $\text{elim}_q$  is the identity function. In order to avoid a sequence of infinite length, we additionally require that the linear ordering changes after every elimination operation. More formally,

$$\begin{aligned} \mathcal{I}_\pi(q) = (s_i)_{i=0}^k, \text{ s. t. } & \quad \text{elim}_q^i(\pi)(s_i) = q, \\ & \quad \forall i \in \{0, \dots, k - 1\} : \text{elim}_q^i(\pi) \neq \text{elim}_q^{i+1}(\pi), \\ & \quad \text{elim}_q^k(\pi) = \text{elim}_q^{k+1}(\pi). \end{aligned}$$

With this definition, the number of steps needed to eliminate all layouts of type  at position  $q$  in  $\pi$  equals exactly the length of  $\mathcal{I}_\pi(q)$  minus one. To facilitate notation, we treat a linear ordering in the following as a permutation of the vertices of  $G$ . A linear ordering is thus represented as an  $n$ -tuple such that the vertex at position  $i$  is listed as the  $i$ th entry.

---



---

**Lemma 4.19**

Let  $\mathcal{I}_\pi(q) = (s_i)_{i=0}^k$  be the sequence of vertices taking position  $q$  during the elimination operation applied to a linear ordering  $\pi$  of a graph  $G$ . Then,  $k \leq n - q - 1$ .

---

*Proof.* Let  $\pi = (v_0, \dots, v_{n-1})$  be a linear ordering of a graph  $G$  and  $q \in \{0, \dots, n-1\}$  be a position within  $\pi$ . Consider the elimination operation at position  $q$ . Let  $s_i$  be a vertex that appears in  $\mathcal{I}_\pi(q) = (s_i)_{i=0}^k$  and assume for the moment that  $i < n - q - 1$ . By definition of  $\mathcal{I}_\pi(q)$ ,  $\text{elim}_q^i(\pi)(s_i) = q$ . If  $i = 0$ , then  $\text{elim}_q^0(\pi) = \pi$ , so  $s_0 = v_q$ . Otherwise, all vertices  $s_j$  with  $0 \leq j < i$  have already been moved successively to position  $n - 1$  of the linear ordering. The linear ordering obtained after the first iteration of  $\text{elim}_q$  is  $\text{elim}_q^1(\pi) = \text{elim}_q(\pi) = (v_0, \dots, v_{q-1}, v_{q+1}, \dots, v_{n-1}, v_q)$ . After the second iteration of  $\text{elim}_q$ , we have that  $\text{elim}_q^2(\pi) = (v_0, \dots, v_{q-1}, v_{q+2}, \dots, v_{n-1}, v_q, v_{q+1})$ , and the linear ordering after the  $i$ th iteration of  $\text{elim}_q$  must therefore be  $\text{elim}_q^i(\pi) = (v_0, \dots, v_{q-1}, v_{q+i}, \dots, v_{n-1}, v_q, \dots, v_{q+i-1})$ . Hence,  $s_i = v_{q+i}$ , still under the assumption that  $i < n - q - 1$ .

If we leave aside that  $i < n - q - 1$  and consider  $i = n - q - 1$ , then  $s_i = v_{n-1}$ , i. e.,  $\text{elim}_q^{n-q-1}(\pi) = (v_0, \dots, v_{q-1}, v_{n-1}, v_q, \dots, v_{n-2})$ . This implies that in the  $(n - q)$ th iteration of  $\text{elim}_q$ ,  $\text{elim}_q^{n-q}(\pi) = (v_0, \dots, v_{q-1}, v_q, \dots, v_{n-1})$ , which is  $\pi$  again. Consequently, from this point on, the elimination process would reconsider only vertices that have already taken position  $q$  in an earlier step, namely in the  $(i - (n - q))$ th iteration, and, what is more, the linear ordering would be the same as in the  $(i - (n - q))$ th iteration. Hence, if the vertex at position  $q$  has an eliminable layout in the  $i$ th iteration, it would also be in position  $q$  and have an eliminable layout in the  $(i + (n - q))$ th iteration. Then, however, the elimination process would not terminate, a contradiction to the proof of [Lemma 4.18](#).

Finally, as  $i \leq k$ , we conclude that  $k \leq n - q - 1$ . □

For the analysis here, we defined  $\text{elim}_q(\pi)$  such that a vertex  $v$  at position  $q$  with eliminable layout is moved to the very end of the linear ordering, whereas in the proof of


**Lemma 4.18**, it has been shown that any position  $p$  with  $p \geq \max_{(u,v) \in \mathcal{B}^-(v)} \pi(u)$  suffices to prove its correctness.

To justify our decision, let us briefly consider the consequences for the length of  $\mathcal{I}_\pi(q)$  if  $\text{elim}_q(\pi)$  would have been defined to move a vertex  $v$  only to position  $\max_{(u,v) \in \mathcal{B}^-(v)} \pi(u)$ . In this case, it is possible that vertices appear in  $\mathcal{I}_\pi(q)$  multiple times. **Figure 4.17** shows an example of such a situation. Due to reoccurring vertices, the length of  $\mathcal{I}_\pi(q)$  here can easily exceed  $n - q$ , which is by **Lemma 4.19** the maximum length of  $\mathcal{I}_\pi(q)$  if we use  $p = n - 1$ . In the example depicted in **Figure 4.17**, the length of  $\mathcal{I}_\pi(q)$  with  $\pi(v_0) = q$  grows quadratically with the number of vertices involved. More precisely, if  $x$  is the (odd) number of vertices in the example and  $f(x)$  denotes the length of  $\mathcal{I}_\pi(q)$ , then  $f(x) = f(x - 2) + x - 1$ , because the vertices at the last two positions are moved for the first time after the vertex  $v_{x-3}$ , i. e., the last vertex in a sequence of length  $x - 2$ , had reached position  $q$  and was as a consequence moved to position  $x - 1$ . Note that in **Figure 4.17**,  $v_2$  changes its position for the first time only in consequence of the elimination of  $v_0$ 's layout, and  $v_4$  remains at its position until the elimination of  $v_2$ 's layout. As  $f(1) = 1$ ,  $f(x) = \frac{1}{4}(x^2 + 3)$ .

With regard to an efficient establishment of the Eliminateable Layouts Property, the possibility of vertices reoccurring in  $\mathcal{I}_\pi(q)$  is undesirable. Moving vertices to a position greater than or equal to the minimum position required but smaller than  $n - 1$  therefore seems not reasonable, as it cannot avoid legitimate reoccurrences of vertices. Setting  $p = n - 1$ , on the other hand, guarantees that this never happens.

If, instead of eliminable layouts of type , those of type  are considered, we can explicitly define  $\text{elim}_q(\pi)$  in an analogous fashion:

$$\text{elim}_q(\pi) = \begin{cases} \text{Move}(\pi, v_q, 0) & \text{if } \mathcal{L}_\pi(v_q) \text{ is of type } \img alt="red arrow pointing left, blue arrow pointing right, red arrow pointing left" data-bbox="725 570 765 590"/>, \\ \pi, & \text{else.} \end{cases}$$

This corresponds exactly to the effect of the *Move* operation carried out on the reverse graph for the vertices with induced layout  according to the reverse linear ordering. In this case, **Proposition 4.3** changes accordingly to:

---



---

**Proposition 4.4**

Let  $\pi$  be a linear ordering of a graph  $G$  and  $q \in \{0, \dots, n - 1\}$ . For every vertex  $v \in V$  holds: If  $\pi(v) > q$ , then  $\text{elim}_q(\pi)(v) = \pi(v)$ , otherwise,  $\text{elim}_q(\pi)(v) \leq q$ .

With  $\mathcal{I}_\pi(q)$  being obtained for this definition of  $\text{elim}_q(\pi)$ , the proof of **Lemma 4.19** can be conducted analogously, so the statement remains valid.

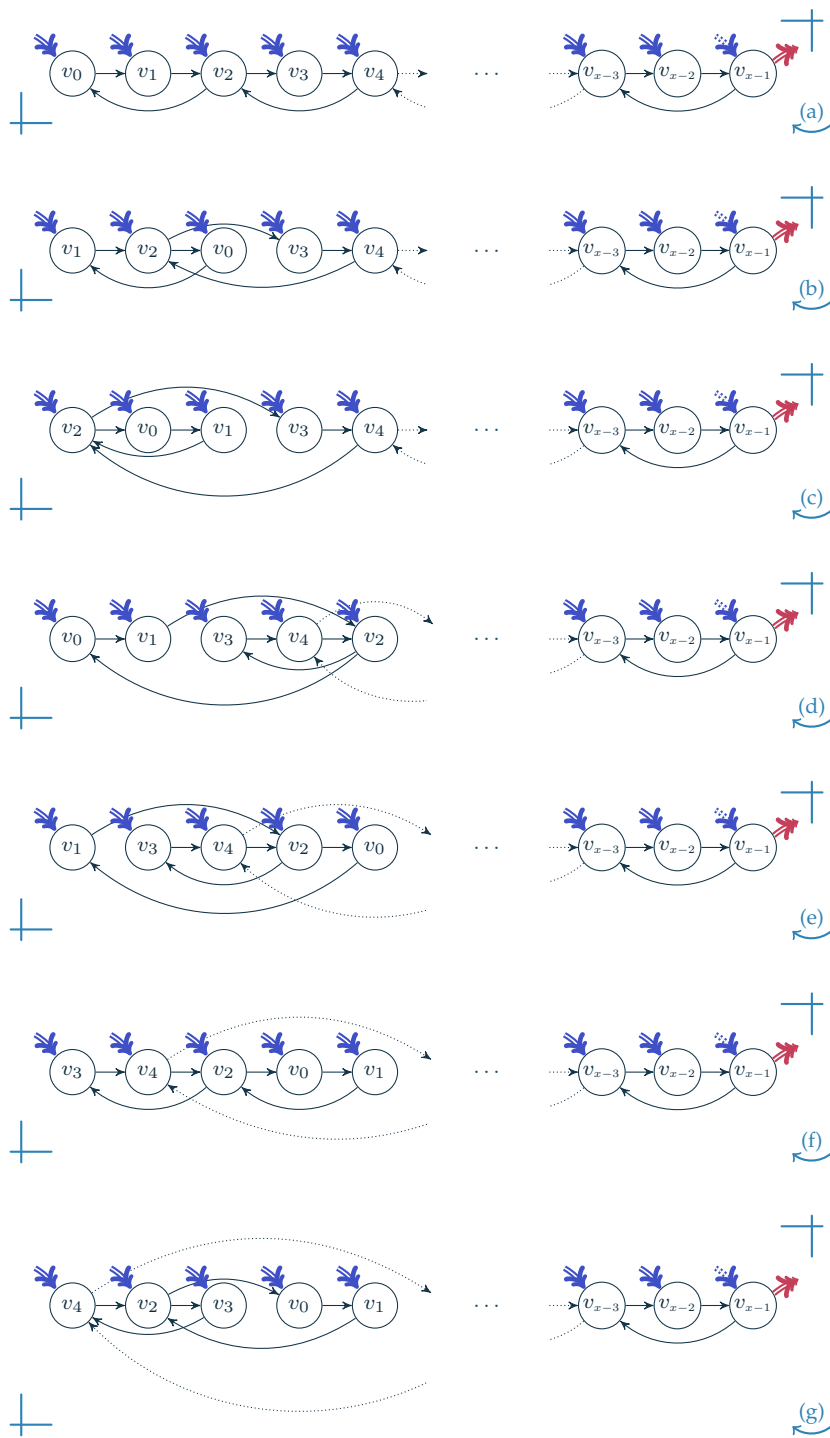









Figure 4.17: First steps of an elimination operation where the elimination of layout  $\rightarrow$  on a vertex  $v$  is implemented such that  $v$  is only moved to  $\max_{(u,v) \in \mathcal{B}^-(v)} \pi(u)$ .

### 4.8.3 Eliminating Elimenable Layouts

The proof of [Lemma 4.18](#) and the discussion in the previous subsection already indicate how the Elimenable Layouts Property can be established efficiently. The listing in [Algorithm 4.10](#) provides the corresponding pseudocode.


Like in the establishment of the Blocking Vertices Property and in accordance with the proof of the Elimenable Layouts Property, the routine  $EliminateLayouts(G, \pi)$  consists of two calls to a subroutine  $EliminateRightLayouts(G, \pi)$ , which takes only care of the elimination of the layout . In the second call,  $EliminateRightLayouts$  therefore obtains the reverse graph along with the reverse linear ordering as arguments instead. As shown in the proof of [Lemma 4.18](#), the elimination operation applied to vertices with layouts of type  cannot produce a layout of  and, because of their symmetry, also vice versa.

The procedure  $EliminateRightLayouts(G, \pi)$  takes a graph  $G$  and a linear ordering  $\pi$  as input. Here, we additionally require  $\pi$  to respect the Nesting Property. Similar to the property-enforcing algorithms we encountered in the previous sections,  $EliminateRightLayouts$  returns  $\pi$  without modification if the initial linear ordering adheres to the Elimenable Layouts Property with respect to the layout . There is, however, a slight difference in the interpretation if  $EliminateRightLayouts$  returns a different linear ordering  $\pi'$ : In this case, the linear ordering either contains a vertex for which [Corollary 4.3](#) does not hold, so  $\pi'$  does not respect the Nesting Property and re-establishing it yields an improved linear ordering; or some vertices have been moved during the execution of this algorithm without violating [Corollary 4.3](#). As [Corollary 4.3](#) is only implied by the Nesting Property but not logically equivalent, the corollary may hold even though the Nesting Property does not. For reasons of efficiency,  $EliminateRightLayouts$ , and thus indirectly also  $EliminateLayouts$ , only checks the weaker restriction. Hence, returning a different linear ordering only indicates that an improvement of the linear ordering may be possible.


For the elimination of layout , the sweep line starts at position 1 ([line 7](#)), because the vertex at position 0 certainly has  $f^-(v) = 0$ , so its layout cannot be of type . Unless the sweep line has advanced beyond the third but last vertex in the linear ordering, which sits at position  $n - 3$ , the algorithm applies two checks to the vertex  $v$  at the current sweep line position. Considering only positions smaller than or equal to  $n - 3$  suffices because a vertex at position  $n - 2$  or  $n - 1$  cannot have both an outgoing forward arc and an incoming backward arc, so its layout cannot be of type .


**Algorithm 4.10** Elimidable Layouts Property**Require:** graph  $G = (V, A)$ , linear ordering  $\pi$  respecting the Nesting Property**Return:**  $\pi$  if  $\text{Elim}(\pi)$ , otherwise a linear ordering  $\pi'$  that can possibly be improved by re-establishing the Nesting Property

```

1: procedure EliminateLayouts( $G, \pi$ )
2:    $\pi' \leftarrow \text{EliminateRightLayouts}(G, \pi)$ 
3:   if  $\pi' \neq \pi$  then return  $\pi'$  else return  $\text{EliminateRightLayouts}(G^R, \pi^R)^R$ 
4: procedure EliminateRightLayouts( $G, \pi$ )
5:    $\pi, \mathcal{B}, \mathcal{F} \leftarrow \text{ComputePositionsAndArcSets}(G, \pi)$ 
6:    $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle \leftarrow \text{ComputeLayoutLists}(G, \pi)$ 
7:    $s \leftarrow 1$  ▷ start sweep line at position 1 for elimination of 
8:   while  $s \leq n - 3$  do
9:      $v \leftarrow \pi^{-1}(s)$  ▷ consider vertex at position  $s$ 
10:    if  $f^+(v) < b^-(v)$  or  $f^-(v) < b^+(v)$  then
11:      return  $\pi$  ▷ violation of Corollary 4.3
12:    else if  $f^+(v) \geq 1$  and  $f^+(v) = b^-(v)$  and  $f^-(v) \geq 1$  and  $b^+(v) = 0$  then
13:       $\pi \leftarrow \text{Move}(\pi, v, n - 1)$ 
14:       $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle \leftarrow \text{Update}(v, \langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle)$ 
15:    else
16:       $s \leftarrow s + 1$  ▷ advance sweep line
17:  return  $\pi$ 

```

If  $v$ 's layout violates [Corollary 4.3](#) (line 10), *EliminateRightLayouts* immediately stops and returns the modified linear ordering  $\pi$ , because the linear ordering is guaranteed to be improvable. Otherwise, in line 12, it checks whether  $v$ 's layout is of type  and, if this is the case, it applies the move operation as defined in [Section 4.8.2](#), i. e.,  $v$  is always moved to the last position  $n - 1$ . To avoid visual clutter, the variable  $\pi$  is simply overwritten here. The algorithm then calls a helper routine  $\text{Update}(v, \langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle)$ , whose task is to update the sorted lists representing the vertices' layouts such that their cardinalities can be queried in subsequent iterations. The implementation of *Update* will be addressed after the description of *EliminateRightLayouts*.

In case that  $v$  neither contradicts [Corollary 4.3](#) nor has a layout of type , the sweep line advances to the next position (line 16). If the algorithm finishes the loop without

**Algorithm 4.11** Helper Routine for *EliminateRightLayouts*( $G, \pi$ )**Require:** moved vertex  $v$ , layout lists  $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle$ **Return:** updated lists  $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle$ 


---

```

1: procedure Update( $v, \langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle$ )
2:   for all  $(v, u)$  in the order of  $\langle \mathcal{F}^+ \rangle(v)$  do
3:      $\langle \mathcal{B}^+ \rangle(v) \leftarrow \langle \mathcal{B}^+ \rangle(v) \diamond \langle (v, u) \rangle$ 
4:     remove  $(v, u)$  from  $\langle \mathcal{F}^- \rangle(u)$ 
5:      $\langle \mathcal{B}^- \rangle(u) \leftarrow \langle \mathcal{B}^- \rangle(u) \diamond \langle (v, u) \rangle$ 
6:   for all  $(u, v)$  in the order of  $\langle \mathcal{B}^- \rangle(v)$  do
7:      $\langle \mathcal{F}^- \rangle(v) \leftarrow \langle \mathcal{F}^- \rangle(v) \diamond \langle (u, v) \rangle$ 
8:     remove  $(u, v)$  from  $\langle \mathcal{B}^+ \rangle(u)$ 
9:      $\langle \mathcal{F}^+ \rangle(u) \leftarrow \langle \mathcal{F}^+ \rangle(u) \diamond \langle (u, v) \rangle$ 
10:   $\langle \mathcal{F}^+ \rangle(v) \leftarrow \langle \rangle; \langle \mathcal{B}^- \rangle(v) \leftarrow \langle \rangle$ 
11:  return  $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle$ 

```

---

cancelling prematurely, the algorithm returns the linear ordering  $\pi$ , which may however be different from the original one in [line 17](#).

Let us now turn to the realization of the helper routine *Update*. Naturally, calling the initializing routines *ComputePositionsAndArcSets* as well as *ComputeLayoutLists* instead would serve the same purpose, which is the update of the representation of the vertex layouts. As the change to the linear ordering is comparatively small, however, we expect a better running time of *Update* in comparison to those, which rebuild all structures from scratch. Furthermore, this helper routine only needs to maintain the structures used by *EliminateRightLayouts*, which are the (cardinalities of) the sorted lists representing the vertex layouts.

[Algorithm 4.11](#) shows the implementation of this routine in pseudocode. *Update* receives the vertex  $v$  that has been moved as well as the layout lists  $\langle \mathcal{F}^- \rangle, \langle \mathcal{F}^+ \rangle, \langle \mathcal{B}^- \rangle, \langle \mathcal{B}^+ \rangle$  as input. Recall from the proof of [Lemma 4.18](#) and from the discussion in the previous subsection that the move operation only affects arcs incident to  $v$  whose head or tail, respectively, has a position greater than  $v$ 's original one, which are exactly the arcs in  $\mathcal{F}^+(v)$  and  $\mathcal{B}^-(v)$ . The classification of these arcs switches from backward to forward and vice versa. In [lines 2–5](#), *Update* traverses the outgoing forward arcs of  $v$  in the order they appear in  $\langle \mathcal{F}^+ \rangle(v)$ , i. e., such that the position of their heads does not decrease. Every such arc is appended to the sorted list of outgoing backward arcs of  $v$ . As  $\langle \mathcal{F}^+ \rangle(v)$



is traversed in order, every arc whose head has a greater position than the currently considered arc will be processed afterwards. This approach guarantees a correct sorting of  $\langle \mathcal{B}^+ \rangle (v)$ . For the head  $u$  of the current arc, similar updates are conducted. Here, the arc is removed from the list  $\langle \mathcal{F}^- \rangle (u)$  of incoming forward arcs and then appended to  $\langle \mathcal{B}^- \rangle (u)$ . As  $v$  is moved to the greatest position within the linear ordering,  $(v, u)$  must be the last arc in the sorted list  $\langle \mathcal{B}^- \rangle (u)$  of incoming backward arcs. Analogously, the incoming backward arcs of  $v$  are processed in lines 6–9. Finally, the sorted lists  $\langle \mathcal{F}^+ \rangle (v)$  and  $\langle \mathcal{B}^- \rangle (v)$  are cleared, because  $v$  has no outgoing forward or incoming backward arcs after being moved. Note that *Update* does not update the attribute of a vertex that explicitly stores its position within  $\pi$ .

---



---

**Lemma 4.20**

*EliminateLayouts*( $G, \pi$ ) runs in time  $\mathcal{O}(n \cdot m)$ .

---

*Proof.* Consider the listing in Algorithm 4.10. First, the algorithm calls *ComputePositionsAndArcSets* as well as *ComputeLayoutLists*, which set up the sorted lists representing the vertex layouts and run in time  $\mathcal{O}(m)$  each by Lemma 4.1. The next statement of *EliminateRightLayouts* is an initialization only and requires time  $\mathcal{O}(1)$ . Consider the body of the loop, which spans lines 8–16. The sweep line itself may be thought of as a pointer to a position in the list that can access the element residing there in constant time. Due to the fact that the algorithm maintains the data structures representing the vertex layouts explicitly, the conditions of the if and else-if clause can also be checked in constant time.

The move operation in line 13 consists of removing  $v$  from its current position in the linear ordering and appending it to the end, which can be accomplished in time  $\mathcal{O}(1)$ , because by assumption,  $\pi$  is represented as a doubly-linked list (cf. Section 4.2). The updates to  $\langle \mathcal{F}^- \rangle$ ,  $\langle \mathcal{F}^+ \rangle$ ,  $\langle \mathcal{B}^- \rangle$ , and  $\langle \mathcal{B}^+ \rangle$  for  $v$  as well as for the head of every arc in  $\langle \mathcal{F}^+ \rangle (v)$  and the tail of every arc in  $\langle \mathcal{B}^- \rangle (v)$  are realized in the subroutine *Update*. Advancing the sweep line in line 16 requires only constant time.

Let us now turn to analyzing the time complexity of *Update*. To this end, consider the listing in Algorithm 4.11. The routine consists of two loops that iterate over  $\langle \mathcal{F}^+ \rangle (v)$  and  $\langle \mathcal{B}^- \rangle (v)$ , respectively. In their bodies, an arc is once removed from a list and appended to a list twice. These three statements can be carried out in time  $\mathcal{O}(1)$ . As to the removal of an arc from a list, we assume that each arc stores altogether two references to its positions within the two layout lists it is contained in, where one of them belongs to

its tail and the other to its head. Then, also removals can be handled in constant time. Clearing the lists  $\langle \mathcal{F}^+ \rangle(v)$  and  $\langle \mathcal{B}^- \rangle(v)$  in [line 10](#) requires at most  $\mathcal{O}(d(v))$  steps. As the sum of the number of iterations of both loops is at most  $d(v)$ , we can conclude that the running time of *Update* is in  $\mathcal{O}(d(v))$ .

Using this result, we find that one iteration of the loop in *EliminateRightLayouts* takes time  $\mathcal{O}(d(v))$  if  $v$  is the currently considered vertex. Next, the number of iterations needs to be addressed. By [Lemma 4.19](#), the elimination process at position  $q$  needs to consider at most  $n - q$  vertices. As  $q$  ranges from 1 to  $n - 3$ , we have at most  $\sum_{q=1}^{n-3} (n - q)$  iterations, which is in  $\mathcal{O}(n^2)$ . Consequently, the running time of the loop dominates that of the initialization steps at the beginning of the algorithm. By using  $\Delta_G$  as an upper bound for the degree  $d(v)$  of any vertex  $v$ , we obtain a running time of  $\mathcal{O}(n^2 \cdot \Delta_G)$ .

This can be slightly improved by the following consideration: For each iteration, we have a time complexity of  $\mathcal{O}(d(v))$ , where  $v$  is the currently considered vertex. The number of iterations  $\mathcal{O}(n^2)$  is asymptotically equivalent to assuming that every vertex of  $G$  appears at every position within  $\pi$ . By [Lemma 4.19](#), no vertex is considered twice at the same position  $q$ , i. e., the time complexity for one position is in  $\mathcal{O}(\sum_{v \in V} d(v)) = \mathcal{O}(m)$ . As there are  $\mathcal{O}(n)$  positions, *EliminateRightLayouts*( $G, \pi$ ) runs in time  $\mathcal{O}(n \cdot m)$ .

We neglect again the time required to compare  $\pi$  to  $\pi'$  in *EliminateLayouts* as well as the effort to obtain the reverse graph and linear orderings due to the preconditions set in [Section 4.2.5](#). As *EliminateLayouts* calls *EliminateRightLayouts* twice, we obtain the same asymptotic running time.  $\square$

Note that *EliminateLayouts* is only useful in combination with a procedure that re-establishes the Nesting Property, e. g., *EstablishNesting*. Hence, we refrain from defining a routine here that only establishes the Elimination Property and head on to further properties. Nonetheless, we do not treat the Elimination Property as a meta-property, because its own behavior is unaffected by the concrete routine that re-establishes the Nesting Property.

## 4.9 A $\Psi_{\text{opt}}$ -Algorithm

Each of the past sections of this chapter introduced a new property that provides a necessary condition for a linear ordering to be optimal. Additionally, algorithms have been given to establish almost each individual property. Our aim now is to combine

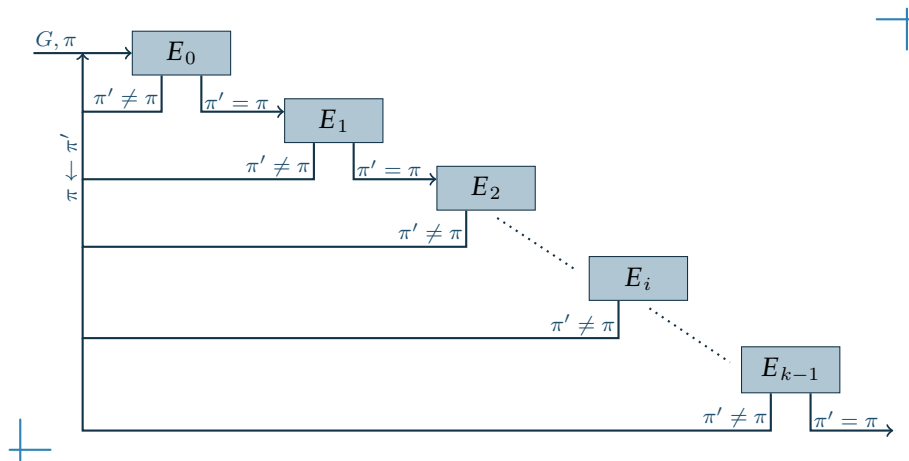


Figure 4.18: Operation principle of the meta-algorithm *Cascade*.

these algorithms into one that efficiently constructs a linear ordering adhering to all properties at the same time.

#### 4.9.1 A Cascading Meta-Algorithm

Let  $G$  be a graph and  $\pi$  be an arbitrary linear ordering of  $G$ . We assume that for every property we want to establish, there is a property-enforcing routine  $E(G, \pi)$  that applies an improvement to the input linear ordering  $\pi$  if  $\pi$  did not adhere to the respective property and otherwise leaves  $\pi$  unchanged. In the former case,  $E(G, \pi)$  returns an improved linear ordering  $\pi'$ , in the latter, the return value equals the argument,  $\pi$ .

At the beginning of this chapter, we introduced a meta-algorithm *Iterate* that takes one such property-enforcing routine as input and calls it repeatedly until the property has finally been established. In order to achieve the same for multiple properties, we devise a meta-algorithm  $Cascade(G, \pi, L = \langle E_0, \dots, E_{k-1} \rangle)$ , which takes an entire list of property-enforcing routines as input. Figure 4.18 illustrates the operation principle of this procedure.

The algorithm passes the graph  $G$  and the current linear ordering  $\pi$  to each property-enforcing routine of the list in turn, starting with  $E_0$ . If a property-enforcing routine  $E_i$  returns  $\pi$  unmodified, then the linear ordering passed to  $E_i$  already respected the corresponding property. If  $E_i$  is the last property-enforcing routine in the list, the algorithm terminates. Otherwise, the next step consists in passing the linear ordering to  $E_{i+1}$ . Whenever a routine returns a different linear ordering than it has been given as argument, the algorithm jumps back to the first element of the list,  $E_0$ .

**Algorithm 4.12** Cascade

**Require:** graph  $G$ , linear ordering  $\pi$ , list  $L = \langle E_0, \dots, E_{k-1} \rangle$  of property-enforcing routines  $E(G, \pi)$

**Return:** a linear ordering that respects all properties enforced by the elements of  $L$

```

1: procedure Cascade( $G, \pi, L = \langle E_0, \dots, E_{k-1} \rangle$ )
2:    $c \leftarrow 0$ 
3:   while  $c < k$  do
4:      $\pi' \leftarrow E_c(G, \pi)$ 
5:     if  $\pi' = \pi$  then
6:        $c \leftarrow c + 1$  ▷ continue to next routine
7:     else
8:        $c \leftarrow 0$  ▷ restart from  $E_0$  with linear ordering  $\pi'$ 
9:        $\pi \leftarrow \pi'$ 
10:  return  $\pi$ 

```

Algorithm 4.12 lists the implementation of *Cascade* in more detail. The counter variable  $c$  is initialized with 0 in line 2. In the following loop, the property-enforcing procedure  $E_c$  is called with parameters  $G$  and  $\pi$ . In case that the returned linear ordering  $\pi'$  equals  $\pi$ , the variable  $c$  is incremented, otherwise, it is reset to 0 and  $\pi$  becomes  $\pi'$ . The loop terminates once  $c$  reaches  $k$ .

---

**Lemma 4.21**

Let  $\pi$  be a linear ordering of a graph  $G$  and let  $L = \langle E_0, \dots, E_{k-1} \rangle$  be a list of property-enforcing routines that either return an improvement of the given linear ordering, or return the linear ordering that has been passed as argument. Then,  $\text{Cascade}(G, \pi, L)$  returns a linear ordering respecting all properties enforced by  $E_0, \dots, E_{k-1}$  and runs in time  $\mathcal{O}(m)$  times the maximum running time of  $E_i(G, \pi)$ , where  $1 \leq i < k$ .

---

*Proof.* Let  $E_i$  be an arbitrary property-enforcing routine. By contract,  $E_i$  returns its argument,  $\pi$ , if and only if  $\pi$  respects the property to enforce. Otherwise, it returns a modified linear ordering  $\pi'$  whose cardinality of the induced set of backward arcs is at least one less than that of  $\pi$ .

Consider the listing in Algorithm 4.12. Let  $r$  denote the number of times that the variable  $c$  is reset to 0 in line 9. Every reset coincides with a property-enforcing routine

returning a new linear ordering, which in turn implies that the cardinality of the induced set of backward arcs has decreased by at least one. In consequence,  $r$  cannot exceed  $|\pi|$ , where  $\pi$  here denotes the initial linear ordering that was passed to *Cascade* as parameter. More precisely,  $r < |\pi|$ , because we only consider strongly connected graphs and  $r = |\pi|$  would imply that the set of backward arcs induced by the linear ordering at termination was empty, i. e., that the graph  $G$  was acyclic.

Next, we want to analyze the steps taken between two resets of the variable  $c$ , including its initialization in [line 2](#). Let  $j$  denote the value of  $c$  immediately before one such reset. Then, since the previous reset, all property-enforcing routines  $E_i$  with  $0 \leq i \leq j$  have been called and all except for  $E_j$  returned the linear ordering  $\pi$  unmodified. The number of steps taken between two resets of the variable  $c$  equals the sum of the steps required for each  $E_i$ ,  $0 \leq i \leq j$ , which is asymptotically equivalent to the maximum of the time complexity of the involved property-enforcing routines. As  $j < k$  and  $r \in \mathcal{O}(m)$ , *Cascade* runs in time  $\mathcal{O}(m)$  times the maximum running time of  $E_i(G, \pi)$ , where  $0 \leq i < k$ .

As to the correctness of *Cascade*, consider the sequel of the algorithm starting with the last reset of the variable  $c$ . Here, the algorithm passes the current linear ordering to each property-enforcing routine of the list. Each routine must return  $\pi$  unmodified, because if not, there would be another reset of the variable  $c$ . Consequently, the current linear ordering respects all properties enforced by  $E_0, \dots, E_{k-1}$  and, as there can be no further reset, *Cascade* terminates.  $\square$

#### 4.9.2 Establishing the Necessary Properties Simultaneously

We now turn to proving the two theorems stated at the very beginning of this chapter. There, we introduced the predicate  $\Psi_{\text{opt}}$  for linear orderings with the following definition:  $\Psi_{\text{opt}}(\pi) \Leftrightarrow \forall \psi \in \Psi : \psi(\pi)$ , where  $\Psi$  denotes the set of predicates corresponding to the properties introduced earlier in this chapter. Hence,

$$\Psi = \{\text{Nest}, \text{Path}, \text{NoBlock}, \text{MPath}, \text{MNoBlock}, \text{Elim}\}.$$

As announced at the beginning of this chapter, we show:

---



---

#### **Theorem 4.1** (restated)

For every linear ordering  $\pi$  holds:  $\text{Opt}(\pi) \Rightarrow \Psi_{\text{opt}}(\pi)$ .

---

*Proof.* Let  $\pi$  be a linear ordering. By [Corollary 4.17](#),  $\text{Opt}(\pi)$  implies  $\text{MNoBlock}(\pi)$  and by [Corollary 4.18](#), if  $\text{MNoBlock}(\pi)$  holds, then also  $\text{Nest}(\pi)$ ,  $\text{Path}(\pi)$ ,  $\text{NoBlock}(\pi)$ , and  $\text{MPath}(\pi)$ . Finally, [Corollary 4.20](#) yields that  $\text{Opt}(\pi)$  also implies  $\text{Elim}(\pi)$ , which concludes the proof of [Theorem 4.1](#).  $\square$

The results of the previous subsection immediately enable us to construct a  $\Psi_{\text{opt}}$ -algorithm  $\text{PsiOpt}(G, \pi)$ , which establishes the Nesting Property, the Path Property, the Blocking Vertices Property, the Multipath Property, the Multipath Blocking Vertices Property, and the Eliminate Layouts Property simultaneously. As to this, we use the meta-algorithm *Cascade* and set the list of property-enforcing routines to  $L := \langle \text{EnforceNesting}, \text{EnforceForwardPaths}, \text{EnforceNoBlocking}, \text{EnforceMultiPaths}, \text{EnforceMultiPathsNoBlocking}, \text{EliminateLayouts} \rangle$ . Then, it is convenient to define  $\text{PsiOpt}(G, \pi)$  as  $\text{Cascade}(G, \pi, L)$ . Let  $\kappa(n, m)$  again be the time complexity of computing a minimum cut in a unit-capacity network.

---



---

**Lemma 4.22**

$\text{PsiOpt}(G, \pi)$  establishes the Nesting Property, the Path Property, the Blocking Vertices Property, the Multipath Property, the Multipath Blocking Vertices Property, and the Eliminate Layouts Property simultaneously on a linear ordering  $\pi$  of a graph  $G$  and runs in time  $\mathcal{O}(n \cdot m \cdot \kappa(n, m))$ .

---

*Proof.* If for all property-enforcing routines contained in the list  $L$  held that they either leave the linear ordering unchanged or improve it, the claim would immediately follow by [Lemma 4.21](#). Unfortunately, this is not the case for *EliminateLayouts*. It is known, however, that the elimination process applied within this routine never constructs a linear ordering whose induced set of backward arcs has greater cardinality than the one it received as an input.

Therefore, let us briefly consider the situation when *EliminateLayouts* returns a new linear ordering  $\pi'$ , but the cardinality of the induced set of backward arcs remains constant, i. e.,  $|\pi| = |\pi'|$ . Whenever a routine returns a new linear ordering, *Cascade* reinvokes all property-enforcing routines of the list  $L$ . Observe that *EliminateLayouts* is the last item in  $L$  and that *EliminateLayouts* is the only property-enforcing routine in  $L$  that may return a new linear ordering that is neither an improvement nor a worsening.

There are two possibilities: Either the modification applied by *EliminateLayouts* to the linear ordering that it has been passed as argument destroyed one of the properties

established earlier by one of the other property-enforcing routines or the new linear ordering still respects all of them despite the modification. In the former case, the property-enforcing routine for the destroyed property must apply an improvement. As shown in the proof of [Lemma 4.21](#), there are at most  $\mathcal{O}(m)$  improvements of the linear ordering possible. In the latter case, the property-enforcing routines preceding *EliminateLayouts* in  $L$  do not change the linear ordering and *Cascade* terminates with one additional run through the elements of  $L$ .

Consequently, the number of resets of the variable  $c$  in [Algorithm 4.12](#) is still bounded by  $\mathcal{O}(m)$ . The running times of the elements of  $L$  are as follows:  $\mathcal{O}(m)$  for *EnforceNesting*,  $\mathcal{O}(\min\{n \cdot m, n^\omega\})$  for *EnforceForwardPaths*,  $\mathcal{O}(\min\{n \cdot m, n^\omega\})$  for *EnforceNoBlocking*,  $\mathcal{O}(n \cdot \kappa(n, m))$  for *EnforceMultiPaths*,  $\mathcal{O}(n \cdot \kappa(n, m))$  for *EnforceMultiPathsNoBlocking*, and  $\mathcal{O}(n \cdot m)$  for *EliminateLayouts*, where  $\omega$  denotes the exponent in the running time of fast matrix multiplication algorithms and  $\omega \geq 2$  (cf. [Section 4.4](#)), and  $\kappa(n, m)$  represents the time complexity of computing a minimum cut in a unit-capacity network with  $\kappa(n, m) \in \Omega(m)$  (cf. [Section 4.6](#)).

With  $\mathcal{O}(\min\{n \cdot m, n^\omega\}) \subseteq \mathcal{O}(n \cdot m)$  and  $\mathcal{O}(n \cdot m) \subseteq \mathcal{O}(n \cdot \kappa(n, m))$ , the maximum running time of a property-enforcing routine in  $L$  equals  $\mathcal{O}(n \cdot \kappa(n, m))$ . Hence,  $\text{PsiOpt}(G, \pi)$  runs in time  $\mathcal{O}(n \cdot m \cdot \kappa(n, m))$ .  $\square$

By choosing an arbitrary initial linear ordering to be passed as an argument to  $\text{PsiOpt}(G, \pi)$ , we obtain an algorithm that constructs a  $\Psi$ -optimal linear ordering  $\pi$ . This concludes the proof of [Theorem 4.2](#) if we substitute  $\mathcal{O}(m \cdot \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$  for  $\kappa(n, m)$ , which is the running time of Dinic's algorithm (cf. [Section 4.6.2](#)):

---

**Theorem 4.2** (restated)

There is an  $\mathcal{O}(n \cdot m^2 \cdot \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ -time algorithm that constructs a  $\Psi$ -optimal linear ordering  $\pi$ .

---

Utilizing the result of [Corollary 4.18](#), which states that the Multipath Blocking Vertices Property implies the Nesting Property, Path Property, Blocking Vertices Property, and Multipath Property, the algorithm  $\text{PsiOpt}(G, \pi)$  can alternatively also be defined as  $\text{Cascade}(G, \pi, \langle \text{EnforceMultiPathsNoBlocking}, \text{EliminateLayouts} \rangle)$ . This neither affects the validity of [Lemma 4.22](#) nor the proof of [Theorem 4.2](#).

For subcubic graphs, we obtain a slightly more efficient  $\Psi_{\text{opt}}$ -algorithm by observing that due to [Corollary 4.3](#), the Nesting Property ensures that every vertex can be incident to at most one backward arc. In consequence, the Path Property together with the

Nesting Property implies the Multipath Property. Furthermore, the Eliminate Layouts Property ensures that every vertex that has an incident backward arc has only either incoming or outgoing forward arcs, but not both. Consequently, if a linear ordering  $\pi$  of a subcubic graph respects the Nesting Property, the Path Property, and the Eliminate Layouts Property, then no cropped forward path can contain a left-blocking or right-blocking vertex, thus,  $\pi$  also respects the Blocking Vertices Property, which is, for the same reason as above, equivalent to the Multipath Blocking Vertices Property. Therefore, it suffices to define the  $\Psi_{\text{opt}}$ -algorithm for subcubic graphs,  $\text{PsiOptCubic}(G, \pi)$ , as  $\text{Cascade}(G, \pi, \langle \text{EnforceNesting}, \text{EnforceForwardPaths}, \text{EliminateLayouts} \rangle)$ .

For subcubic graphs, we have that  $m = \frac{3}{2}n$ , i. e.,  $m \in \mathcal{O}(n)$ . The time complexities of  $\text{EnforceNesting}$ ,  $\text{EnforceForwardPaths}$ , and  $\text{EliminateLayouts}$  therefore all reduce to  $\mathcal{O}(n^2)$ . With the same argument regarding the handling of  $\text{EliminateLayouts}$  as in the proof of [Lemma 4.22](#), we derive from [Lemma 4.21](#):

---



---

**Corollary 4.21**

There is an  $\mathcal{O}(n^3)$ -time algorithm that constructs a  $\Psi$ -optimal linear ordering  $\pi$  of a subcubic graph.

---

## 4.10 Manipulations and Meta-Properties

Eventually, we attend to some further properties of optimal linear orderings that are not beneficial on their own, but rather in conjunction with other LINEAR ORDERING algorithms or, particularly with regard to the earlier results of this chapter, property-enforcing routines. That is why we also call them “meta-properties”. In this section, we introduce three major representatives of this kind: the Fusion Property, the Reduction Property, and the Arc Stability Property. As each of them is  $\mathcal{NP}$ -hard in its most general form, we additionally suggest some weaker, polynomial-time establishable, variants.

A characteristic that is common to all of these meta-properties is that they are built on manipulations of the input graph and, accordingly, its current linear ordering. Therefore, we first lay the foundations for these manipulations in a separate subsection and afterwards turn our attention to the above mentioned meta-properties.



### 4.10.1 Basic Operations on Linear Orderings and Graphs

We start by studying a number of simple graph operations and their impact on a linear ordering. Although the motivation for this is provided by the meta-properties that will be presented later in this section, the following results are also of interest in themselves and can be considered independently. Our goal is that, given a graph  $G$  along with a linear ordering  $\pi_G$ , the manipulation of  $G$  can be transferred simultaneously to a manipulation of  $\pi_G$  such that we obtain a new graph  $H$  along with a linear ordering  $\pi_H$  and  $\pi_H$  is derived from  $\pi_G$ .

In the following, let  $G = (V_G, A_G)$  be a graph with vertex set  $V_G$  and arc set  $A_G = (U_G, m_G)$  and let the graph  $H = (V_H, A_H)$  with vertex set  $V_H$  and arc set  $A_H = (U_H, m_H)$  be obtained from  $G$  by some manipulation. For the benefit of a conciser notation, we generalize  $m_G$  and  $m_H$  such that for some tuple of vertices  $(x, y)$  with  $(x, y) \notin U_G$ ,  $m_G((x, y)) = 0$ . The same applies analogously to  $m_H$ . Except for the penultimate graph operation, we assume that  $G$  is free of loops and pairs of anti-parallel arcs. The latter does not necessarily also apply to  $H$ .

Note that the operations that are introduced to describe the manipulations behave like functions, i. e., they have a return value and do not change the graphs, vertices, arcs, or linear orderings that are passed to them as arguments. We nevertheless stick to the term “operation” as graph manipulations are widely known also as “graph operations”.

The implications of the operations for the respective linear ordering are in general not difficult to see and have to some extent also been observed in the same or a similar fashion elsewhere, e. g., by Younger [You63]. For the sake of completeness and to avoid the need to bother the reader with subtle differences, we provide all necessary proofs.

#### Manipulations of Linear Orderings

For a concise description of how a linear ordering changes, we define three basic operations: If both  $G$  and  $H$  have the same set of vertices, i. e.,  $V_G = V_H$ , then any linear ordering  $\pi_G$  of  $G$  can also be interpreted as a linear ordering of  $H$ . For the sake of clarity, we denote such a reinterpretation explicitly by the operation  $\text{reinterpret}(\pi_G, H)$ , which returns the corresponding linear ordering for  $H$ .

Otherwise, if  $H$  was obtained from  $G$  by adding a new vertex  $u$ , i. e.,  $V_H = V_G \cup \{u\}$ , then a linear ordering of  $H$  may be obtained by inserting  $u$  at some position  $q$ , where

$0 \leq q < |V_H|$ , in a linear ordering  $\pi_G$  of  $G$ . This is the definition of the operation  $\text{insert}(\pi_G, u, q)$ . More formally, for every vertex  $v \in V_H$ ,

$$\text{insert}(\pi_G, u, q)(v) = \begin{cases} \pi_G(v), & \text{if } \pi_G(v) < q, \\ q, & \text{if } v = u, \\ \pi_G(v) + 1, & \text{if } \pi_G(v) \geq q. \end{cases}$$

For convenience, we also provide a version of the operation which takes a graph as an additional argument and performs a reinterpretation by defining  $\text{insert}(\pi_G, u, q, H) = \text{reinterpret}(\text{insert}(\pi_G, u, q), H)$ .

Finally,  $H$  may be the result of the removal of a vertex  $u$  from  $G$ , i. e.,  $V_H = V_G \setminus \{u\}$ . In this case, the construction of a linear ordering of  $H$  from a linear ordering  $\pi_G$  of  $G$  by simply leaving out  $v$  suggests itself. We denote this operation by  $\text{skip}(\pi_G, u)$ . Formally, the definition reads

$$\text{skip}(\pi_G, u)(v) = \begin{cases} \pi_G(v), & \text{if } \pi_G(v) < \pi_G(u), \\ \pi_G(v) - 1, & \text{if } \pi_G(v) > \pi_G(u), \end{cases}$$

where  $v \in V_H$ . As above, we define a convenience operation which also performs a reinterpretation by setting  $\text{skip}(\pi_G, u, H) = \text{reinterpret}(\text{skip}(\pi_G, u), H)$ .

Having this settled, we are ready to turn to graph manipulations.

### Vertex Removal

We start with the very simple graph operation of *vertex removal*. For a vertex  $v \in V_G$ , we denote by  $\text{remove}(G, v)$  the removal of  $v$  from  $G$ , i. e., if  $H = \text{remove}(G, v)$ , then  $H = G|_{V_G \setminus \{v\}}$ .

Consider a vertex  $v$  whose incident number of backward arcs with respect to a linear ordering  $\pi_G$  of a graph  $G$  is the maximum possible in an optimal linear ordering by [Corollary 4.5](#), i. e.,  $b_{\pi_G}(v) = \min \{d^+(v), d^-(v)\}$ . We call such a vertex *maximal* in  $\pi_G$ . To distinguish such vertices further, we say that a maximal vertex  $v$  is *out-maximal* in  $\pi_G$  if  $b_{\pi_G}(v) = d^+(v)$  and *in-maximal* in  $\pi_G$  if  $b_{\pi_G}(v) = d^-(v)$ .

---

#### Proposition 4.5

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G = (V_G, A_G)$ . A vertex  $v \in V_G$  is

- (i) out-maximal in  $\pi_G^*$  if and only if  $v$  is either a pseudosink or right-blocking,
  - (ii) in-maximal in  $\pi_G^*$  if and only if  $v$  is either a pseudosource or left-blocking.
-

*Proof.* Let  $v \in V_G$  be a maximal vertex in  $\pi_G^*$ . Then,  $b_{\pi_G^*}(v) = b_{\pi_G^*}^-(v) + b_{\pi_G^*}^+(v) = \min \{d^+(v), d^-(v)\}$ . Assume for (i) that  $b_{\pi_G^*}(v) = d^+(v)$ . Then,  $b_{\pi_G^*}^-(v) = d^+(v) - b_{\pi_G^*}^+(v) = f_{\pi_G^*}^+(v)$ . Thus,  $v$  is a pseudosink if  $b_{\pi_G^*}^-(v) = f_{\pi_G^*}^+(v) = 0$  and otherwise right-blocking. In case (ii), where  $b_{\pi_G^*}(v) = d^-(v)$  instead, an analogous argument yields that  $v$  is a pseudosource if  $b_{\pi_G^*}^+(v) = f_{\pi_G^*}^-(v) = 0$  and otherwise left-blocking.

For the respective converse, assume first that  $v$  is either a pseudosink or right-blocking, i. e.,  $f_{\pi_G^*}^+(v) = b_{\pi_G^*}^-(v)$ . Note that if  $v$  is a pseudosink, then  $f_{\pi_G^*}^+(v) = b_{\pi_G^*}^-(v) = 0$ . Hence,  $b_{\pi_G^*}^-(v) + b_{\pi_G^*}^+(v) = f_{\pi_G^*}^+(v) + b_{\pi_G^*}^+(v) = d^+(v)$ . Likewise, if  $v$  is either a pseudosource or left-blocking, then  $f_{\pi_G^*}^-(v) = b_{\pi_G^*}^+(v)$  and  $b_{\pi_G^*}^-(v) + b_{\pi_G^*}^+(v) = b_{\pi_G^*}^-(v) + f_{\pi_G^*}^-(v) = d^-(v)$ . As  $b_{\pi_G^*}(v) = b_{\pi_G^*}^-(v) + b_{\pi_G^*}^+(v) \leq \min \{d^+(v), d^-(v)\}$  by Corollary 4.5,  $v$  is maximal in  $\pi_G^*$ .  $\square$

Let us now study the implications on a linear ordering  $\pi_G$  of  $G$  upon the removal of a maximal vertex.

---

#### Lemma 4.23

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G = (V_G, A_G)$  and let  $v \in V_G$  be maximal in  $\pi_G^*$ . Then,  $\pi_H = \text{skip}(\pi_G^*, v, H)$  is an optimal linear ordering of  $H = \text{remove}(G, v)$  and  $|\pi_G^*| = |\pi_H| + b_{\pi_G^*}(v)$ .

---

*Proof.* Let  $H = (V_H, A_H)$ . The removal of  $v$  from  $G$  also implies the removal of all arcs incident to  $v$ . As  $v$  is maximal,  $|\pi_G^*| = |\pi_H| + b_{\pi_G^*}(v) = |\pi_H| + \min \{d^+(v), d^-(v)\}$ .

Suppose that  $\pi_H$  is not optimal and let  $\pi_H^*$  be an optimal linear ordering of  $H$ . Then,  $|\pi_H^*| < |\pi_H|$ . Construct a new linear ordering  $\pi_G$  of  $G$  such that

$$\pi_G = \begin{cases} \text{insert}(\pi_H^*, v, 0, G), & \text{if } d^-(v) \leq d^+(v), \\ \text{insert}(\pi_H^*, v, |V_H|, G), & \text{else,} \end{cases}$$

i. e.,  $v$  is inserted at the very beginning of  $\pi_H^*$  if it has at most as many incoming as outgoing arcs, and otherwise at the very end. Consequently,  $v$  is incident to exactly  $\min \{d^-(v), d^+(v)\}$  backward arcs in  $\pi_G$ . Thus,

$$|\pi_G| = |\pi_H^*| + \min \{d^+(v), d^-(v)\} < |\pi_H| + \min \{d^+(v), d^-(v)\} = |\pi_G^*|,$$

a contradiction to the optimality of  $\pi_G^*$ .  $\square$

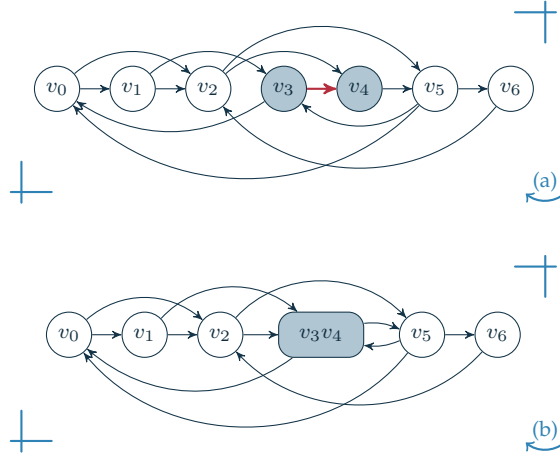


Figure 4.19: A linear ordering  $\pi_G$  of a graph  $G$  (a) and the result of  $\text{contract}(G, (v_3, v_4))$  with linear ordering  $\text{insert}(\text{skip}(\text{skip}(\pi_G, v_3), v_4), v_3v_4, \pi_G(v_3), H)$  (b).

### Arc Contraction

Let  $(u, v)$  be an arc of  $G$ . By *contracting*  $(u, v)$ , we obtain a new graph  $H$  such that the vertices  $u$  and  $v$  of  $G$  are replaced by a new vertex “ $uv$ ” which inherits all arcs incident to  $u$  and  $v$  in  $G$  except for the arc  $(u, v)$  itself and, if existent, all parallel arcs of  $(u, v)$ . If  $u$  and  $v$  are adjacent to the same vertex in  $G$ , the contraction may create additional parallel or anti-parallel arcs.

We denote this operation by  $\text{contract}(G, (u, v))$ . Formally, if  $H = \text{contract}(G, (u, v))$ , then  $V_H = V_G \setminus \{u, v\} \cup \{uv\}$  and  $A_H = (U_H, m_H)$ , where

$$\begin{aligned} U_H = & \{(x, y) \in U_G \mid x, y \in V_H\} \\ & \cup \{(x, uv) \in U_G \mid x \in (\mathbb{N}_G^-(u) \cup \mathbb{N}_G^-(v)) \setminus \{u, v\}\} \\ & \cup \{(uv, y) \in U_G \mid y \in (\mathbb{N}_G^+(u) \cup \mathbb{N}_G^+(v)) \setminus \{u, v\}\} \end{aligned}$$

and for all  $(x, y) \in U_H$ ,

$$m_H((x, y)) = \begin{cases} m_G((x, y)), & \text{if } x, y \notin \{u, v\}, \\ m_G((x, u)) + m_G((x, v)), & \text{if } y = uv, \\ m_G((u, y)) + m_G((v, y)), & \text{if } x = uv. \end{cases}$$

Consider now a linear ordering  $\pi_G$  of  $G$  and assume that  $\pi_G(v) = \pi_G(u) + 1$ . As  $u$  and  $v$  are consecutive with respect to  $\pi_G$ , setting the corresponding linear ordering  $\pi_H$  of  $H$  to  $\text{insert}(\text{skip}(\text{skip}(\pi_G, u), v), uv, \pi_G(u), H)$  suggests itself.

In Figure 4.19, the effect of an arc contraction is visualized. Figure 4.19(a) highlights the arc  $(v_3, v_4)$  of the graph  $G$  in the linear ordering  $\pi_G$ , which is contracted in Figure 4.19(b). The linear ordering  $\pi_H$  used for the graph resulting from the arc contraction,  $H$ , is  $\pi_H = \text{insert}(\text{skip}(\text{skip}(\pi_G, v_3), v_4), v_3v_4, \pi_G(v_3), H)$ .

---



---

**Lemma 4.24**

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G = (V_G, A_G)$ ,  $u, v \in V_G$ ,  $(u, v) \in A_G$ , and  $\pi_G^*(v) = \pi_G^*(u) + 1$ . Then,  $\pi_H = \text{insert}(\text{skip}(\text{skip}(\pi_G^*, u), v), uv, \pi_G^*(u), H)$  is an optimal linear ordering of  $H = \text{contract}(G, (u, v))$  and  $|\pi_G^*| = |\pi_H|$ .

---

*Proof.* Denote by  $\mathcal{B}_{\pi_G^*}$  and  $\mathcal{B}_{\pi_H}$  the set of backward arcs in  $G$  induced by  $\pi_G^*$  and the set of backward arcs in  $H$  induced by  $\pi_H$ , respectively. The contraction of arc  $(u, v)$  in  $G$  produces counterparts of all arcs in  $H$  except for  $(u, v)$  and its parallel arcs. By construction of  $\pi_H$ , the correspondents of all arcs of  $H$  that are backward arcs in  $\pi_H$  are also backward arcs in  $\pi_G^*$  and vice versa. The arc  $(u, v)$  and its parallel arcs are forward arcs with respect to  $\pi_G^*$ . Subsequently,  $|\mathcal{B}_{\pi_G^*}| = |\mathcal{B}_{\pi_H}|$ , which yields  $|\pi_G^*| = |\pi_H|$ .

Suppose for the sake of contradiction that  $\pi_H$  is not an optimal linear ordering of  $H$ . Then, there is an optimal linear ordering  $\pi_H^*$  of  $H$  such that  $|\pi_H^*| < |\pi_H|$ . Obtain a linear ordering  $\pi_G$  of  $G$  from  $\pi_H^*$  by applying the corresponding inverse operations, i. e.,  $\pi_G = \text{insert}(\text{insert}(\text{skip}(\pi_H^*, uv), u, \pi_H^*(uv)), v, \pi_H^*(uv) + 1, G)$ . Note that this implies in turn that  $\pi_H^* = \text{insert}(\text{skip}(\text{skip}(\pi_G, u), v), uv, \pi_G(u), H)$ . With the same argument as above, we can conclude that  $|\pi_G| = |\pi_H^*|$ . By assumption, however,  $|\pi_H^*| < |\pi_H|$ . Hence, also  $|\pi_G| < |\pi_G^*|$ , a contradiction, because  $\pi_G^*$  is an optimal linear ordering of  $G$ .  $\square$

From Lemma 4.24 and its proof we can derive that for every linear ordering  $\pi_H$  of a graph  $H$  that is obtained from a graph  $G$  by arc contraction, there is a corresponding linear ordering  $\pi_G$  of  $G$  that induces as many backward arcs on  $G$  as  $\pi_H$  induces on  $H$ , because  $\pi_G$  can always be constructed from  $\pi_H$  such that the contracted arc is a forward arc with respect to  $\pi_G$ . In particular, this also applies to optimal linear orderings of  $H$  and yields the following result:

---



---

**Corollary 4.22**

Let  $(u, v)$  be an arc of a graph  $G$  and  $H = \text{contract}(G, (u, v))$ . Then,  $\tau_G \leq \tau_H$ .

---

## Arc Insertion

Next, we consider the effects of connecting two non-adjacent vertices in a graph by inserting a new arc. Let  $u, v$  be two distinct vertices of  $G$ , i. e.,  $u \neq v$ . The operation  $\text{connect}(G, u, v)$  manipulates  $G$  by adding a new arc  $(u, v)$ . Formally, if  $H = \text{connect}(G, u, v)$ , then  $V_H = V_G$  and  $A_H = A_G \uplus [(u, v)]$ . Note that  $u$  and  $v$  may already be adjacent in  $G$ . We immediately make the following observation:

---

**Proposition 4.6**

Let  $\pi_G$  be a linear ordering of a graph  $G$ ,  $H = \text{connect}(G, u, v)$ , and  $\pi_H = \text{reinterpret}(\pi_G, H)$ . Then,  $|\pi_G| = |\pi_H|$ , if  $\pi_G(u) < \pi_G(v)$ , and  $|\pi_H| = |\pi_G| + 1$ , otherwise.

---

*Proof.* If  $\pi_G(u) < \pi_G(v)$ , then the new arc  $(u, v)$  is a forward arc with respect to  $\pi_H$ . In this case,  $|\pi_H| = |\pi_G|$ . Otherwise,  $\pi_G(u) > \pi_G(v)$ , so  $(u, v)$  counts as a backward arc and  $|\pi_H| = |\pi_G| + 1$ .  $\square$

Let us now consider optimal linear orderings and arc insertions for two consecutive vertices:

---

**Lemma 4.25**

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G = (V_G, A_G)$ ,  $u, v \in V_G$ , and  $\pi_G^*(v) = \pi_G^*(u) + 1$ . Then,  $\pi_H = \text{reinterpret}(\pi_G^*, H)$  is an optimal linear ordering of  $H = \text{connect}(G, u, v)$  and  $|\pi_G^*| = |\pi_H|$ .

---

*Proof.* As  $\pi_G^*(u) < \pi_G^*(v)$ , [Proposition 4.6](#) immediately implies  $|\pi_G^*| = |\pi_H|$ .

Suppose for the sake of contradiction that  $\pi_H$  is not an optimal linear ordering of  $H$ . Then, there is an optimal linear ordering  $\pi_H^*$  of  $H$  with  $|\pi_H^*| < |\pi_H|$ . Let  $\pi_G = \text{reinterpret}(\pi_H^*, G)$  be the corresponding linear ordering of  $G$ . By [Proposition 4.6](#),  $|\pi_G| \leq |\pi_H^*|$ , which yields  $|\pi_G| \leq |\pi_H^*| < |\pi_H| = |\pi_G^*|$ . Hence,  $|\pi_G| < |\pi_G^*|$ , a contradiction to  $\pi_G^*$  being an optimal linear ordering of  $G$ .  $\square$

Observe that the graph obtained from  $G$  by inserting an arc always has the same set of vertices as  $G$ . Consequently, every linear ordering of one graph can always be reinterpreted as a linear ordering of the other graph. If we consider optimal linear orderings of both graphs, then this observation in combination with [Proposition 4.6](#) immediately yields:

---



---

**Corollary 4.23**

Let  $u, v$  be two vertices of a graph  $G$  and  $H = \text{connect}(G, u, v)$ . Then,  $\tau_G \leq \tau_H \leq \tau_G + 1$ .

---

### Arc Subdivision

The *subdivision* can be considered the converse operation of an arc contraction. Let  $(u, v)$  be again be an arc of  $G$ . Then, subdividing  $(u, v)$  yields a new graph  $H$  that removes the arc  $(u, v)$  and adds a new vertex “ $uv$ ” along with two arcs  $(u, uv)$  and  $(uv, v)$ .

We denote this operation by  $\text{subdivide}(G, (u, v))$ . Formally, if  $H = \text{subdivide}(G, (u, v))$ , then  $V_H = V_G \cup \{uv\}$  and  $A_H = A_G \setminus \{(u, v)\} \cup \{(u, uv), (uv, v)\}$ . Applying this operation to any arc of a graph does not have great influence on its linear orderings:

---



---

**Proposition 4.7**

Let  $\pi_G$  be a linear ordering of a graph  $G$ ,  $H = \text{subdivide}(G, (u, v))$ , and  $\pi_H = \text{insert}(\pi_G, uv, \pi_G(u) + 1, H)$  or  $\pi_H = \text{insert}(\pi_G, uv, \pi_G(v), H)$ . Then,  $|\pi_G| = |\pi_H|$ .

---

*Proof.* If  $(u, v)$  is a forward arc, then  $\pi_G(u) < \pi_G(v)$  and subsequently,  $\pi_H(u) < \pi_H(uv) < \pi_H(v)$  for both choices of  $\pi_H$ . Hence,  $(u, uv)$  and  $(uv, v)$  are also forward arcs and  $|\pi_G| = |\pi_H|$ .

Otherwise, if  $(u, v)$  is a backward arc, then  $\pi_G(v) < \pi_G(u)$ . In case that  $\pi_H = \text{insert}(\pi_G, uv, \pi_G(u) + 1, H)$ ,  $\pi_H(v) < \pi_H(u) < \pi_H(uv)$ , i. e.,  $(u, uv)$  is forward and  $(uv, v)$  is backward. On the other hand, if  $\pi_H = \text{insert}(\pi_G, uv, \pi_G(v), H)$ , then  $\pi_H(uv) < \pi_H(v) < \pi_H(u)$ , which implies that  $(u, uv)$  is backward and  $(uv, v)$  is forward. As the backward arc  $(u, v)$  is removed by the subdivision operation and replaced by  $(u, uv)$  and  $(uv, v)$ , and exactly one of them is backward and the other is forward, we again obtain  $|\pi_G| = |\pi_H|$ .  $\square$

In particular, the above observation can be transferred to the optimality of linear orderings as well:

---



---

**Lemma 4.26**

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G$ ,  $H = \text{subdivide}(G, (u, v))$ , and  $\pi_H = \text{insert}(\pi_G, uv, \pi_G(u) + 1, H)$  or  $\pi_H = \text{insert}(\pi_G, uv, \pi_G(v), H)$ . Then,  $\pi_H$  is an optimal linear ordering of  $H$ .

---

*Proof.* Towards a contradiction, suppose that  $\pi_H$  is not an optimal linear ordering of  $H$  and let  $\pi_H^*$  be an optimal linear ordering of  $H$ . Then,  $|\pi_H^*| < |\pi_H|$ . Consider the classification of  $(u, uv)$  and  $(uv, v)$  with respect to  $\pi_H^*$ . If both are backward, then the vertex  $uv$  is incident to two backward arcs, but no forward arcs. However, this contradicts the Nesting Property and thereby also the optimality of  $\pi_H^*$ . Hence, at least one of  $(u, uv)$  and  $(uv, v)$  must be a forward arc. Furthermore, if one of them is a backward arc, then the other arc must be the corresponding nesting arc at  $uv$ . Subsequently, if  $(u, uv)$  is backward, then  $\pi_H^*(uv) < \pi_H^*(v) < \pi_H^*(u)$ , and if  $(uv, v)$  is backward, then  $\pi_H^*(v) < \pi_H^*(u) < \pi_H^*(uv)$ .

Let  $\pi_G = \text{skip}(\pi_H^*, uv, G)$  be the linear ordering of  $G$  obtained from  $\pi_H^*$  by simply ignoring  $uv$ . In case that both  $(u, uv)$  and  $(uv, v)$  are forward arcs,  $\pi_H^*(u) < \pi_H^*(uv) < \pi_H^*(v)$ . Hence,  $\pi_G(u) < \pi_G(v)$  and the sets of backward arcs induced by  $\pi_H^*$  and  $\pi_G$  are identical. Otherwise, if one of  $(u, uv)$  and  $(uv, v)$  is backward with respect to  $\pi_H^*$ , then  $\pi_H^*(v) < \pi_H^*(u)$  as shown above. Thus, also  $\pi_G(v) < \pi_G(u)$  and  $(u, v)$  is backward with respect to  $\pi_G$ . Then, the set of backward arcs induced by  $\pi_H^*$  and  $\pi_G$  differ only in that the former contains either  $(u, uv)$  or  $(uv, v)$ , whereas the latter contains  $(u, v)$  instead. Subsequently,  $|\pi_H^*| = |\pi_G|$ .

By [Proposition 4.7](#),  $|\pi_G^*| = |\pi_H|$ . Hence,  $|\pi_G| = |\pi_H^*| < |\pi_H| = |\pi_G^*|$ , a contradiction to the optimality of  $\pi_G^*$ . In consequence,  $\pi_H$  must be an optimal linear ordering of  $H$ .  $\square$

### Neutralization

So far, we have put the possibility that an arc contraction produces one or more pairs of anti-parallel arcs aside, although we stipulated in [Section 3.4](#) that we assume all input graphs to be free of anti-parallel arcs. To also allow for the handling of graphs resulting from arc contractions, we introduce a graph operation called *neutralization*, which removes anti-parallel arcs pairwise. More precisely, we show that an arc and its reverse neutralize each other with respect to their impact on an optimal linear ordering and thereby provide the formal proof for our assumption in [Section 3.4](#) as a side line.

Let  $u, v \in V_G, u \neq v$ , be two distinct vertices and  $(u, v), (v, u)$  be a pair of anti-parallel arcs of  $G$ . The operation  $\text{neutralize}(G, (u, v), (v, u))$  yields a new graph by removing one copy of each of  $(u, v)$  and  $(v, u)$  from  $G$ . Formally, if  $H = \text{neutralize}(G, (u, v), (v, u))$ , then  $V_H = V_G$  and  $A_H = A_G \setminus [(u, v), (v, u)]$ . Subsequently, with  $A_H = (U_H, m_H)$ ,  $(u, v) \in U_H \Leftrightarrow m_G((u, v)) \geq 2$  and  $(v, u) \in U_H \Leftrightarrow m_G((v, u)) \geq 2$ . If  $(u, v) \in U_H$ , then  $m_H((u, v)) = m_G((u, v)) - 1$ . Likewise, if  $(v, u) \in U_H$ , then  $m_H((v, u)) = m_G((v, u)) - 1$ .



---



---

**Proposition 4.8**

Let  $u, v \in V_G$ ,  $u \neq v$ ,  $(u, v), (v, u) \in A_G$ , and  $H = \text{neutralize}(G, (u, v), (v, u))$ .

Then, for any linear ordering  $\pi_G$  of  $G$  holds:  $|\pi_G| = |\text{reinterpret}(\pi_G, H)| + 1$ .

---

*Proof.* Let  $\pi_H = \text{reinterpret}(\pi_G, H)$ . As either  $\pi_G(u) < \pi_G(v)$  or  $\pi_G(v) < \pi_G(u)$ , one of  $(u, v)$  and  $(v, u)$  must be a forward arc and the other one must be a backward arc with respect to  $\pi_G$ . Observing that exactly these arcs are removed in  $H$  then yields that  $|\pi_H| = |\pi_G| - 1$ .  $\square$

As in the case of arc insertion, every linear ordering of a graph can be reinterpreted as a linear ordering of the graph after applying a neutralization operation and vice versa.

---



---

**Corollary 4.24**

Let  $u, v \in V_G$ ,  $u \neq v$ , and  $(u, v), (v, u) \in A_G$ . If  $H = \text{neutralize}(G, (u, v), (v, u))$ , then  $\tau_G = \tau_H + 1$ .

---

With regard to the improvment of linear orderings, the following result is especially of interest:

---



---

**Lemma 4.27**

Let  $\pi_G$  be a linear ordering of a graph  $G$ ,  $u \neq v \in V_G$ ,  $(u, v), (v, u) \in A_G$ , and  $H = \text{neutralize}(G, (u, v), (v, u))$ . Then,  $\text{Opt}(\pi_G) \Leftrightarrow \text{Opt}(\text{reinterpret}(\pi_G, H))$ .

---

*Proof.* Consider an optimal linear ordering  $\pi_G^*$  of  $G$  and let  $\pi_H = \text{reinterpret}(\pi_G^*, H)$ . By [Proposition 4.8](#),  $|\pi_G^*| = |\pi_H| + 1$ . Suppose that  $\pi_H$  is not an optimal linear ordering of  $H$ . Then, there is an optimal linear ordering  $\pi_H^*$  of  $H$  such that  $|\pi_H^*| < |\pi_H|$ . Let  $\pi_G = \text{reinterpret}(\pi_H^*, G)$ . By [Proposition 4.8](#),  $|\pi_G| = |\pi_H^*| + 1$ . Consequently,  $|\pi_G| - 1 = |\pi_H^*| < |\pi_H| = |\pi_G^*| - 1$ , which implies that  $|\pi_G| < |\pi_G^*|$ , a contradiction to  $\pi_G^*$  being optimal.

By switching the roles of  $G$  and  $H$ , the same argument can be used to show that if  $\pi_H^* = \text{reinterpret}(\pi_G, H)$  is an optimal linear ordering of  $H$ , then  $\pi_G = \text{reinterpret}(\pi_H^*, G)$  is an optimal linear ordering of  $G$ .  $\square$

For convenience, we define a graph operation that eliminates all pairs of anti-parallel arcs by an exhaustive application of the neutralization operation as long as a pair of anti-parallel arcs exists. To this end, we “overload” the operation `neutralize` and use

$H = \text{neutralize}(G)$  with the just described semantics. Formally, the graph  $H$  that is obtained by  $\text{neutralize}(G)$  has the vertex set  $V_H = V_G$  and the arc set  $A_H = (U_H, m_H)$ , where

$$U_H = \{(u, v) \in U_G \mid (v, u) \notin U_G \vee ((v, u) \in U_G \wedge m_G((u, v)) > m_G((v, u)))\}$$

and for every arc  $(u, v) \in U_H$ ,

$$m_H((u, v)) = \begin{cases} m_G((u, v)), & \text{if } (v, u) \notin U_G, \\ m_G((u, v)) - m_G((v, u)), & \text{else.} \end{cases}$$

We can immediately derive from [Lemma 4.27](#):

---



---

**Corollary 4.25**

Let  $\pi_G$  be a linear ordering of a graph  $G$  and  $H = \text{neutralize}(G)$ . Then,  $\text{Opt}(\pi_G) \Leftrightarrow \text{Opt}(\text{reinterpret}(\pi_G, H))$ .

---

### Vertex Fusion

Finally, by combining three of the aforementioned operations, namely arc connection, arc contraction, and neutralization, we obtain what we call a *vertex fusion*. A specialty here is that we use the involved vertices' positions to specify where the fusion shall occur. The vertex fusion applied to a graph  $G$  and a linear ordering  $\pi_G$  at position  $q$ ,  $0 \leq q < n - 1$ , is denoted by  $\text{fuse}(G, \pi_G, q)$  and yields a tuple  $(H, \pi_H)$ , where

$$H = \text{neutralize}(\text{contract}(\text{connect}(G, v_q, v_{q+1}), (v_q, v_{q+1})))$$

and

$$\pi_H = \text{insert}(\text{skip}(\text{skip}(\pi_G, v_q), v_{q+1}), v_q v_{q+1}, \pi_G(v_q), H).$$

In other words, a vertex fusion replaces two consecutive vertices  $u$  and  $v$  by a new vertex " $uv$ " and afterwards removes anti-parallel arcs pairwise.

From our study of the single operations that it is compiled from, we immediately obtain:

---



---

**Lemma 4.28**

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G$  on  $n$  vertices and  $q$  be a position with  $\pi_G^*$  with  $0 \leq q < n - 1$ . Then,  $(H, \pi_H) = \text{fuse}(G, \pi_G^*, q)$  such that  $\pi_H$  is an optimal linear ordering of  $H$ .

---

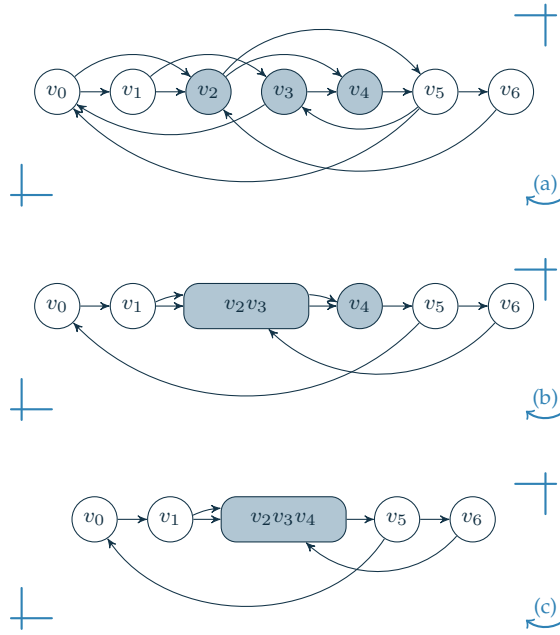


Figure 4.20: A linear ordering  $\pi_G$  of a graph  $G$  (a) along with the result of  $\text{fuse}(G, \pi_G, 2)$  (b) and the second iterate  $\text{fuse}^2(G, \pi_G, 2)$  (c).

*Proof.* Let  $v_q$  denote the vertex at position  $q$ , i. e.,  $\pi_G^*(v_q) = q$ . As  $\pi_G^*$  is an optimal linear ordering of  $G$  and  $v_q, v_{q+1}$  are two consecutive vertices with respect to  $\pi_G^*$ ,  $\pi_{H'} = \text{reinterpret}(\pi_G^*, H')$  is an optimal linear ordering of  $H' = \text{connect}(G, v_q, v_{q+1})$  by Lemma 4.25. Furthermore, the linear ordering  $\pi_{H''} = \text{insert}(\text{skip}(\text{skip}(\pi_{H'}, v_q), v_{q+1}), v_q v_{q+1}, \pi_G^*(v_q), H'')$  then also is an optimal linear ordering of  $H'' = \text{contract}(H', (v_q, v_{q+1}))$  by Lemma 4.24. Finally, Lemma 4.27 implies that  $\pi_H = \text{reinterpret}(\pi_{H''}, H)$  is an optimal linear ordering of  $H = \text{neutralize}(H'')$ , which concludes the proof.  $\square$

#### 4.10.2 Fusion Property

The Fusion Property and its variants, as the name already suggests, are based on the graph operation that we just introduced, the vertex fusion. Consider a linear ordering  $\pi_G = (v_0, v_1, \dots, v_{n-1})$  of some graph  $G = (V_G, A_G)$  with  $A_G = (U_G, m_G)$ . As hitherto, we treat a linear ordering as a permutation of  $V$ . With a slight abuse of notation, we can use functional powers to generalize the fusion of two consecutive vertices in  $\pi_G$  to a fusion of an arbitrary set of consecutive vertices:  $\text{fuse}^0(G, \pi_G, q)$  then yields  $(G, \pi_G)$  and for any natural number  $i$ ,  $1 \leq i < n - q$ ,  $\text{fuse}^i(G, \pi_G, q) = \text{fuse}(\text{fuse}^{i-1}(G, \pi_G, q), q)$ , which is equivalent to a fusion of the vertices with position at least  $q$  and at most  $q + i$ .

Figure 4.20 exemplifies the fusion of a set of consecutive vertices: The initial graph  $G$  with linear ordering  $\pi_G$  is shown in Figure 4.20(a). A vertex fusion at position 2, i. e.,  $\text{fuse}(G, \pi_G, 2)$ , yields the graph and linear ordering depicted in Figure 4.20(b). The arcs  $(v_1, v_2)$  and  $(v_1, v_3)$  of  $G$  result in a pair of parallel arcs  $(v_1, v_2v_3)$ , whereas the correspondents of the arcs  $(v_0, v_2)$  and  $(v_3, v_0)$  of  $G$  now neutralize each other, and likewise do  $(v_2, v_5)$  and  $(v_5, v_3)$ . Another pair of parallel arcs originates from  $(v_2, v_4)$  and  $(v_3, v_4)$ . The graph and linear ordering in Figure 4.20(c) is obtained by another application of the vertex fusion operation and is equivalent to  $\text{fuse}^2(G, \pi_G, 2)$ .

Using the above definition, we can state the Fusion Property as follows:

---

**Lemma 4.29** FUSION PROPERTY

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G$  on  $n$  vertices and  $\mathcal{B}$  its induced set of backward arcs. For every linear ordering  $\pi_G \in [\pi_G^*]_{\sim_{\mathcal{B}}}$ , every position  $q$  with  $0 \leq q < n$ , and every iterate  $i$  with  $0 \leq i < n - q$ , the vertex fusion  $\text{fuse}^i(G, \pi_G, q)$  yields a graph  $H$  and a linear ordering  $\pi_H$  such that  $\pi_H$  is an optimal linear ordering of  $H$ .

---

*Proof.* Let  $G$  be a graph that is free of anti-parallel arcs with an optimal linear ordering  $\pi_G^*$ . Consider any linear ordering  $\pi_G \in [\pi_G^*]_{\sim_{\mathcal{B}}}$ . As  $\pi_G^*$  is optimal, so is  $\pi_G$ .

Let now  $q$  be a position within  $\pi_G$  with  $0 \leq q < n$  and  $0 \leq i < n - q$ . We prove that  $\text{fuse}^i(G, \pi_G, q) = (H, \pi_H)$  such that  $\pi_H$  is an optimal linear ordering of  $H$  by induction on  $i$ .

**Base case  $i = 0$ :** As  $\text{fuse}^0(G, \pi_G, q) = (G, \pi_G)$ , the claim follows immediately.

**Base case  $i = 1$ :** In this case,  $\text{fuse}^1(G, \pi_G, q)$  equals  $\text{fuse}(G, \pi_G, q)$  and the claim follows immediately from Lemma 4.28.

**Inductive step  $i \Rightarrow i+1$ :** Let  $(H, \pi_H) = \text{fuse}^i(G, \pi_G, q)$ . By induction hypothesis,  $\pi_H$  is an optimal linear ordering of  $H$ . Let  $(H', \pi_{H'}) = \text{fuse}^{i+1}(G, \pi_G, q)$ . As  $\text{fuse}^{i+1}(G, \pi_G, q) = \text{fuse}(\text{fuse}^i(G, \pi_G, q), q)$ ,  $(H', \pi_{H'}) = \text{fuse}(H, \pi_H, q) = \text{fuse}^1(H, \pi_H, q)$  and the claim follows again from Lemma 4.28. □

At the beginning of this section, the Fusion Property was called a meta-property. Indeed, given a linear ordering  $\pi$  of a graph, we cannot derive benefit from Lemma 4.29 alone. Such a benefit could be, e. g., the ability to tell whether  $\pi$  can be improved, similar to the properties discussed earlier in this chapter. If we additionally consider one of these “beneficial” properties, however, things change.

Let  $\pi_G$  be again a linear ordering of a graph  $G$  and let  $\psi$  be such a beneficial property. We may even assume that  $\pi_G$  respects  $\psi$ . If there is a set of consecutive vertices with respect to  $\pi_G$ , however, such that their fusion yields a linear ordering not respecting  $\psi$ , [Lemma 4.29](#) immediately implies that  $\pi_G$  cannot be optimal. Moreover, the Fusion Property enables us to actually construct an improved linear ordering of  $G$  by first enforcing  $\psi$  on the linear ordering obtained from the vertex fusion, and then re-expanding the fused vertices while preserving their original relative order.

As the Fusion Property essentially makes a statement about all topological sortings of the acyclic subgraph  $G|_{\mathcal{F}}$  of a given linear ordering  $\pi_G$  in combination with the fusion of all sets of consecutive vertices with respect to  $\pi_G$ , the number of resulting linear orderings which have to be checked is superpolynomial in general. For this reason, the formulation of relaxations seems appropriate. We exemplarily state three below:

All relaxations relinquish to consider all possible topological sortings of  $G|_{\mathcal{F}}$  and only use the current linear ordering  $\pi_G$ . The first weaker version of the Fusion Property additionally restricts itself to the sets of consecutive vertices that include the vertex at position zero in the linear ordering:

---

**Corollary 4.26** PREFIX FUSION PROPERTY

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G$ . For every iterate  $i$  with  $0 \leq i < n$ , the vertex fusion  $\text{fuse}^i(G, \pi_G^*, 0)$  yields a graph  $H$  and a linear ordering  $\pi_H$  such that  $\pi_H$  is an optimal linear ordering of  $H$ .

---

Note that it may be the case that the  $i$ -th iterate of the fusion operation yields a linear ordering that respects some property, whereas the  $(i - 1)$ -th iterate does not and vice versa.

Analogously, another relaxation of the Fusion Property consists in considering only sets of consecutive vertices that include the vertex at position  $n - 1$  in the linear ordering:

---

**Corollary 4.27** SUFFIX FUSION PROPERTY

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G$ . For every position  $q$  with  $0 \leq q < n$ , the vertex fusion  $\text{fuse}^{n-q-1}(G, \pi_G^*, q)$  yields a graph  $H$  and a linear ordering  $\pi_H$  such that  $\pi_H$  is an optimal linear ordering of  $H$ .

---

Both the Prefix Fusion Property and the Suffix Fusion Property “generate”  $n$  linear orderings that can be passed on to other property-enforcing routines. Note that the

Suffix Fusion Property is in fact equivalent to the Prefix Fusion Property on the reverse graph along with the reverse linear ordering.

Finally, a reasonable assumption would be that the implications of the Fusion Property are especially strong if every backward arc in the original linear ordering also has a correspondent in the linear ordering resulting from the vertex fusions. To this end, we define a function  $\text{maxIterate}(q)$  that yields the largest iterate  $i$ ,  $0 \leq i < n - q$ , of the vertex fusion at position  $q$  such that every backward arc has a correspondent in the resulting graph and linear ordering. Then, the greedy vertex fusion operation  $\text{fusegreedy}(G, \pi_G)$  yields a graph  $H$  and a linear ordering  $\pi_H$  that can be obtained as follows: Initially, set  $q \leftarrow 0$ ,  $i \leftarrow \text{maxIterate}(q)$ , and let  $(H', \pi_{H'}) = \text{fuse}^i(G, \pi_G, q)$ . Next, set  $q \leftarrow q + i + 1$ ,  $i \leftarrow \text{maxIterate}(q)$  using the new value of  $q$ , and reapply the vertex fusion operation using  $H'$  and  $\pi_{H'}$  instead of  $G$  and  $\pi_G$  as input. The last step is repeated as long as  $q$  does not exceed  $n - 1$ .

---

**Corollary 4.28** GREEDY FUSION PROPERTY

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G$  and let  $H$  and  $\pi_H$  be the graph and linear ordering resulting from  $\text{fusegreedy}(G, \pi_G^*)$ . Then,  $\pi_H$  is an optimal linear ordering of  $H$ .

---

As in case of the Prefix Fusion Property and the Suffix Fusion Property, the linear ordering obtained thereby can be passed to other property-enforcing routines. Under the assumption that a vertex fusion can be computed in time that is proportional to the degrees of the two involved vertices, i. e.,  $\mathcal{O}(d(u) + d(v))$  for vertices  $u$  and  $v$ , the time required for a multi-vertex fusion is in  $\mathcal{O}(n)$ , due to the increasing vertex degree of the fused vertices. Enforcing the Prefix Fusion Property or the Suffix Fusion Property in combination with another algorithm with running time in  $\Omega(n)$  therefore increases it by a factor of  $\mathcal{O}(n)$ , whereas enforcing the Greedy Fusion Property generates just one linear ordering and hence always only requires  $\mathcal{O}(n)$  additional steps for its construction.

### 4.10.3 Reduction Property

For the second meta-property, we reduce the input graph by removing a set of vertices. To simplify notation, we introduce a generalized version of the function  $\text{skip}$  which creates a new linear ordering by skipping a set of vertices instead of only a single one. For a linear ordering  $\pi_G$  of a graph  $G = (V_G, A_G)$ , it is formally defined by

$$\text{skip}(\pi_G, U)(v) = \pi_G(v) - |\{u \in U \mid \pi_G(u) < \pi_G(v)\}|,$$

where  $v$  is a vertex of  $G$  and  $U \subseteq V$ . As for the original definition of skip, we define the convenience function  $\text{skip}(\pi_G, U, H) = \text{reinterpret}(\text{skip}(\pi_G, U), H)$ .

---

**Lemma 4.30**
**REDUCTION PROPERTY**

Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V_G, A_G)$  and let  $U$  be a set of pseudosources, pseudosinks, and either left- or right-blocking vertices, but not both. Then,  $\pi_H = \text{skip}(\pi_G^*, U, H)$  is an optimal linear ordering of  $H = G_{V_G \setminus U}$ .

---

*Proof.* The statement follows trivially if  $U = \emptyset$ . Thus, we assume in the remainder that  $|U| \geq 1$ . First, consider an out-maximal vertex  $v$  in  $\pi^*$ . As the name suggests,  $b_{\pi^*}(v) = d^+(v)$ . Furthermore, by [Proposition 4.5](#),  $v$  is either a pseudosink or right-blocking. Suppose that  $v$  loses one of its outgoing backward arcs. Then, both its outdegree and its number of incident backward arcs decrease by one. Subsequently,  $v$  is still out-maximal afterwards. Likewise, if we consider an in-maximal vertex  $v$  in  $\pi^*$ , then  $v$  must be a pseudosource or left-blocking. Here, the same applies for the removal of an incoming backward arc:  $v$  remains in-maximal.

Assume that  $U$  contains no left-blocking vertices and order the vertices  $u_i \in U$ ,  $0 \leq i < k = |U|$ , such that  $\pi^*(u_i) < \pi^*(u_{i+1})$  for every  $0 \leq i < k - 1$ . By [Lemma 4.23](#),  $\pi_G^{(0)} = \text{skip}(\pi_G^*, u_0, G^{(0)})$  is an optimal linear ordering of  $G^{(0)} = \text{remove}(G, u_0)$ . If  $k = 1$ , this already proves the statement for the case that  $U$  contains only pseudosources, pseudosinks, or right-blocking vertices. Otherwise, the removal of  $u_0$  from  $G$  and its incident arcs may have resulted in the loss of either an incoming forward arc or an outgoing backward arc of another vertex  $u_j \in U$ . In consequence,  $u_j$  cannot be a pseudosource with respect to  $\pi^*$ , but must be a pseudosink or right-blocking, i. e.,  $u_j$  is out-maximal. As  $\pi_G^{(0)}$  is optimal,  $f_{\pi_G^{(0)}}^-(u_j) \geq b_{\pi_G^{(0)}}^+(u_j)$ . Furthermore, due to our above considerations,  $u_j$  is out-maximal with respect to  $\pi_G^{(0)}$ . Subsequently, every vertex in  $U \setminus \{u_0\}$  is either a pseudosource, a pseudosink, or right-blocking with respect to  $\pi_G^{(0)}$ . We can therefore apply [Lemma 4.23](#) once more and obtain that  $\pi_G^{(1)} = \text{skip}(\pi_G^{(0)}, u_1, G^{(1)})$  is an optimal linear ordering of  $G^{(1)} = \text{remove}(G^{(0)}, u_1)$ . As all remaining vertices in  $U$ —should they exist—can again only have lost an incoming forward arc or an outgoing backward arc, the reasoning from above can be continued for all vertices of  $U$ . This finally yields that  $\pi_G^{(k-1)} = \text{skip}(\pi_G^{(k-2)}, u_{k-1}, G^{(k-1)}) = \pi_H$  is an optimal linear ordering of  $G^{(k-1)} = H$  and thereby concludes this part of the proof.

The argument for the case that  $U$  contains no right-blocking vertices follows immediately by considering the reverse graph  $G^R$  along with the reverse linear ordering  $\pi_G^{*R}$ .

Alternatively, we can prove the statement directly by ordering the vertices  $u_i \in U$ ,  $0 \leq i < k = |U|$ , such that  $\pi^*(u_i) > \pi^*(u_{i+1})$  for every  $0 \leq i < k - 1$ , i. e., in descending order with respect to  $\pi^*$ . Then, the removal of a vertex  $u_i$  can only cause a vertex  $u_j$ ,  $j > i$ , to lose either its outgoing forward arc or its incoming backward arc. Thus,  $u_j$  cannot be a pseudosink, but must be in-maximal. The optimality of the corresponding linear ordering in consequence of [Lemma 4.23](#) again guarantees that  $f_{\pi_G^{(i)}}^+(u_j) \geq b_{\pi_G^{(i)}}^-(u_j)$  and our consideration at the beginning of the proof implies that  $u_j$  is in-maximal with respect to  $\pi_G^{(i)}$ . By continuing the argument, we obtain that  $\pi_G^{(k-1)} = \text{skip}(\pi_G^{(k-2)}, u_{k-1}, G^{(k-1)}) = \pi_H$  is an optimal linear ordering of  $G^{(k-1)} = H$ .  $\square$

As the vertex set  $U$  in [Lemma 4.30](#) is not limited in size, enforcing the Reduction Property as it is stated would require to consider all suitable subsets of vertices, which in general is a superpolynomial amount. Instead of providing a weaker, but efficiently establishable version of this property here, we hint at its close relationship to the Blocking Vertices Property and the Multipath Blocking Vertices Property. These also deal with left- and right-blocking vertices, but, in contrast to the Reduction Property, they do not remove them, but split them vertically and thereby preserve in particular their incident backward arcs. Furthermore, both of them can be enforced in polynomial time.

Note that a key insight in the proof of [Lemma 4.30](#) was that if we stick to the prescribed ordering of the vertices of  $U$ , the removal of a vertex did not affect the out- or in-maximality of the remaining vertices. If the set  $U$  were to contain both left- and right-blocking vertices, however, this guarantee would be void. Interestingly, if we consider the example given in [Figure 4.5](#) and [Figure 4.8](#) with this in mind, we observe that after the removal of the right-blocking vertex  $v_5$ , the hitherto left-blocking vertex  $v_7$  is no longer left-blocking, and vice versa.

The following version of a Reduction Property hence follows immediately from [Lemma 4.23](#):

---

**Corollary 4.29** INDEPENDENT SET REDUCTION PROPERTY  
 Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V_G, A_G)$  and let  $U$  be an independent set of maximal vertices. Then,  $\pi_H = \text{skip}(\pi_G^*, U, H)$  is an optimal linear ordering of  $H = G_{V_G \setminus U}$ .

---



## 4.10.4 Arc Stability Property

When we introduced the first property of this chapter, the Nesting Property, we observed that it is related to a family of algorithms that are subsumed under the term local search. In fact, the algorithm that enforces the Nesting Property is a representative of the class of  $k$ -opt heuristics, whose principle of design is to improve a given solution by simultaneously modifying (up to)  $k$  of its components. *EnforceNesting* does so by moving one vertex at a time to a new position within a given linear ordering  $\pi$  and is therefore actually a 1-opt algorithm.

We will now bring in another implementation of a  $k$ -opt algorithm that also operates on linear orderings, but in a different way. In result, we obtain a further property of optimal linear orderings. In contrast to the Nesting Property, however, it is a meta-property:

**Lemma 4.31**

## ARC STABILITY PROPERTY

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G = (V, A)$  and let  $F \subseteq \mathcal{F}_{\pi_G^*}$  be a subset of the forward arcs induced by  $\pi_G^*$ . Then,  $|\pi_G^*| \leq |\pi_H| + |F|$  for every linear ordering  $\pi_H$  of the spanning subgraph  $H = (V, A \setminus F)$ .

*Proof.* Suppose there is a linear ordering  $\pi'_H$  of  $H$  such that  $|\pi'_H| < |\pi_G^*| - k$ , where  $k = |F|$ . As  $G$  and  $H$  share the same set of vertices,  $\pi'_G = \text{reinterpret}(\pi'_H, G)$  is a linear ordering of  $G$ . Recall that  $H = G|_{A \setminus F}$ . In the worst case,  $\pi'_G$  classifies every arc in  $F$  as backward, which yields  $|\pi'_G| \leq |\pi'_H| + k$  and hence,  $|\pi'_G| - k \leq |\pi'_H| < |\pi_G^*| - k$ . Consequently,  $|\pi'_G| < |\pi_G^*|$ , a contradiction to the optimality of  $\pi_G^*$ .  $\square$

Apparently, enforcing the Arc Stability Property even in combination with a polynomial-time heuristic on a graph with  $m$  arcs requires  $\binom{m}{k}$  calls to this subroutine for each  $k \in \{1, \dots, m - |\pi| - 1\}$  and is hence unattractive for the design of fast algorithms. On the other hand, the running time increases only by a polynomial factor of  $\mathcal{O}(m^k)$  if the cardinality  $k$  of  $F$  is fixed.

The following trivial version hence only has a multiplicative overhead of  $\mathcal{O}(m)$ :

**Corollary 4.30**

## ONE-ARC STABILITY PROPERTY

Let  $\pi_G^*$  be an optimal linear ordering of a graph  $G = (V, A)$  and let  $a$  be a forward arc with respect to  $\pi_G^*$ . Then,  $|\pi_G^*| \leq |\pi_H| + 1$  for every linear ordering  $\pi_H$  of the spanning subgraph  $H = (V, A \setminus \{a\})$ .



## 5 | Maximum Cardinality of Optimal Feedback Arc Sets of Sparse Graphs

In the preceding chapter, we compiled a collection of properties that every optimal linear ordering must respect. What is more, we showed that the larger part of them can be established efficiently, both separately and collectively. For the latter, we developed an algorithm *PsiOpt* for general graphs as well as its sibling *PsiOptCubic* for subcubic graphs, which construct a compliant solution, i. e., a  $\Psi$ -optimal linear ordering, in polynomial time.

We now turn to an analysis of these algorithms with respect to their qualitative performance on sparse graphs. More precisely, we study in particular graphs whose maximum vertex degree is three or four, and also suggest an approach for graphs with a maximum vertex degree of five. In doing so, we obtain improved upper bounds for the cardinality of minimum feedback arc sets for these classes.

The chapter is structured as follows: In the first section, we define some auxiliary graphs. These graphs are successively derived from an input graph  $G$  and a concrete linear ordering  $\pi$  of  $G$  and will aid in the analysis of the cardinality of the backward arc set induced by  $\pi$ . Next, we attend to subcubic and cubic graphs and prove a tight upper bound of  $\lfloor \frac{n}{3} \rfloor$  for the cardinality of a minimum feedback arc set of a graph with  $n$  vertices. This section also contains a short discussion of the quality of a  $\Psi$ -optimal linear ordering in comparison to an optimal one. Afterwards, we extend the scheme used for subcubic graphs such that we can essentially analyze graphs with arbitrary vertex degrees. The application of this general approach to subquartic graphs yields a tight upper bound of  $\lfloor \frac{2n}{3} \rfloor$  for a graph with  $n$  vertices, which, if all vertices have degree exactly four, equals  $\lfloor \frac{m}{3} \rfloor$  with  $m = 2n$  being the number of arcs. Finally, we briefly consider subquintic graphs and arrive at the conjecture that the upper bound here is  $\lfloor \frac{2.5n}{3} \rfloor$  for a graph on  $n$  vertices. If the graph is quintic, then the number of arcs is  $m = 2.5n$ , which would again imply an upper bound of  $\lfloor \frac{m}{3} \rfloor$ .

Throughout this chapter, we assume that the input graph is simple.

## 5.1 Auxiliary Graphs

In Chapter 4, we introduced the concept of a forward path. The Path Property uses forward paths in a very basic manner in that it only demands their existence for every backward arc. The Blocking Vertices Property poses further restrictions, and the Multipath Property and finally the Multipath Blocking Vertices Property require compliance with other backward arcs' forward paths.

We will now use forward paths to obtain a few auxiliary graph structures. The constructions build on each other in the sense that the first auxiliary graph forms the basis for the second, the second for the third, and so on.

### 5.1.1 The Forward Path Graph

Let  $\pi$  be a  $\Psi$ -optimal linear ordering of a graph  $G = (V, A)$ . In tangible terms, the Multipath Blocking Vertices Property guarantees that if we take the left-blocking split graph  $G_{\text{sp}/l}$  of  $G$  along with the corresponding linear ordering  $\pi_{\text{sp}/l}$ , then  $\pi_{\text{sp}/l}$  respects the Multipath Property, i. e., for every vertex  $v$  in  $G_{\text{sp}/l}$ , there is a set of pairwise arc-disjoint forward paths that contains a distinct forward path  $P_b$  for every backward arc  $b$  that is incident to  $v$  according to  $\pi_{\text{sp}/l}$ . Furthermore, the same holds for the right-blocking split graph  $G_{\text{sp}/r}$  together with the corresponding linear ordering  $\pi_{\text{sp}/r}$ .

On the basis of the latter, we define the term of a forward path graph. By definition of  $\Psi$ -optimality,  $\pi$  must respect the Multipath Blocking Vertices Property. A *forward path graph*  $\tilde{G} = (\tilde{V}, \tilde{A})$  of  $G$  with respect to  $\pi$  is a spanning subgraph of  $G$ 's right-blocking split graph  $G_{\text{sp}/r}$ , i. e.,  $\tilde{G} \subseteq G_{\text{sp}/r}$ , where  $\tilde{G}$  is compiled from the backward arcs induced by  $\pi_{\text{sp}/r}$  plus for every vertex  $v \in \tilde{V}$  a set of pairwise arc-disjoint forward paths  $\mathcal{P}_v$  such that  $\mathcal{P}_v$  contains a distinct forward path for every incoming backward arc of  $v$  with respect to  $\pi_{\text{sp}/r}$ . Figure 5.1 provides an example for a graph and a forward path graph of it. The colors indicate the respective forward paths that have been chosen for each backward arc. Along with the forward path graph  $\tilde{G}$ , we maintain a linear ordering  $\tilde{\pi}$ , whose vertex ordering equals that of  $\pi_{\text{sp}/r}$ , i. e.,  $\tilde{\pi} = \text{reinterpret}(\pi_{\text{sp}/r}, \tilde{G})$ . Note that because all backward arcs are preserved and the arcs before and after the split are identified,  $\mathcal{B}_{\tilde{\pi}} = \mathcal{B}_{\pi_{\text{sp}/r}} = \mathcal{B}_{\pi}$  and thus,  $|\tilde{\pi}| = |\pi_{\text{sp}/r}| = |\pi|$ .

In contrast to  $G_{\text{sp}/r}$ , the forward path graph  $\tilde{G}$  is not uniquely determined. Indeed, there may be more than one suitable set of arc-disjoint forward paths for the incoming backward arcs of a vertex, and we select one arbitrarily. What is more, arc-disjointness is only required for forward paths of backward arcs with a common head. The forward

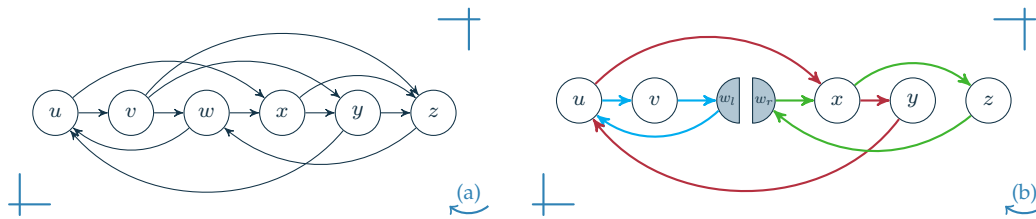


Figure 5.1: A graph (a) along with one of its forward path graphs (b).

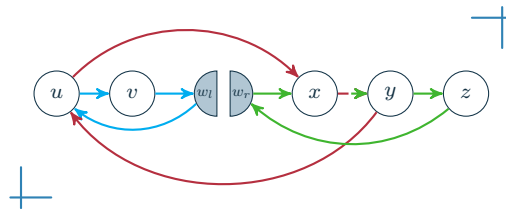


Figure 5.2: An alternative forward path graph for the graph in Figure 5.1(a).

path graph depicted in Figure 5.2 thus is another possible forward path graph for the graph and linear ordering shown in Figure 5.1(a). Also note that for the set of selected forward paths, arc-disjointness is only required for those belonging to the incoming backward arcs of each vertex, but not necessarily for those belonging to its outgoing backward arcs. The latter are chosen at their respective heads. In Section 4.6.1 we have already seen that a coincident selection is not always possible. Figure 5.3 provides another example of the construction of a forward path graph that demonstrates these facts.

Note that  $\tilde{\pi}$  respects the Path Property but neither the Nesting Property nor the Elimenable Layouts Property, due to  $\tilde{G}$  being only a subgraph of  $G$ . More precisely,  $\tilde{\pi}$  respects only “half” of the Nesting Property, namely the part that concerns the backward arcs’ heads, i. e., the mapping  $\mu_h$  in Lemma 4.3. It does, however, not hold for the

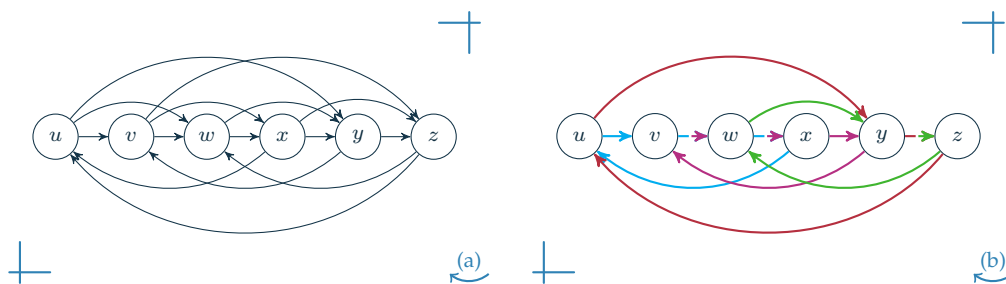


Figure 5.3: Another graph (a) along with one of its forward path graphs (b).

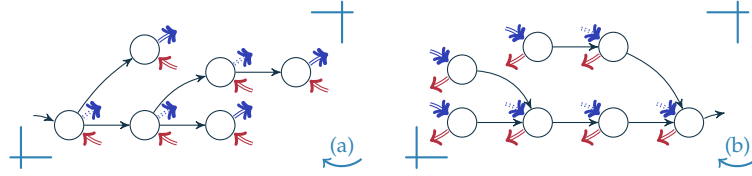


Figure 5.4: An example for a for-out-tree on vertices with layout  $\rightarrow\circ\leftarrow$  (a) and an example for a for-in-tree on vertices with layout  $\leftarrow\circ\rightarrow$  (b). The dotted blue arcs indicate that there may, but need not, be more incident forward arcs.

backward arcs' tails, which is expressed via the mapping  $\mu_t$ . In particular, we still have  $f_{\tilde{\pi}}^+(v) \geq b_{\tilde{\pi}}^-(v)$  for every vertex  $v$  within  $\tilde{\pi}$ . If  $b_{\tilde{\pi}}^+(v) \geq 1$ , however, then it is only guaranteed that  $f_{\tilde{\pi}}^-(v) \geq 1$ , but not necessarily  $f_{\tilde{\pi}}^-(v) \geq b_{\tilde{\pi}}^+(v)$ .

### 5.1.2 The Pooled Forward Path Graph

For a further analysis of the forward path graph  $\tilde{G}$ , we pool vertices with certain layouts in  $\tilde{\pi}$ . Consider a vertex  $v \in \tilde{V}$  such that  $f_{\tilde{\pi}}^-(v) = 1$ ,  $b_{\tilde{\pi}}^-(v) \geq 1$ , and  $b_{\tilde{\pi}}^+(v) = 0$ . We represent the layout of vertices like  $v$  by the pictogram  $\rightarrow\circ\leftarrow$ . Observe that every such vertex has exactly one incoming forward arc and that these arcs alone naturally cannot form a cycle, because they are all forward arcs. Hence, the vertices with layout  $\rightarrow\circ\leftarrow$  together with their incoming forward arcs define a subgraph of  $\tilde{G}$  that is a forest of out-trees. We call these out-trees the for-out-trees on vertices with layout  $\rightarrow\circ\leftarrow$  in  $\tilde{\pi}$ . Analogously, we can identify a forest of for-in-trees in  $\tilde{\pi}$  that consist of vertices  $v \in \tilde{V}$  such that  $f_{\tilde{\pi}}^+(v) = 1$ ,  $b_{\tilde{\pi}}^+(v) \geq 1$ , and  $b_{\tilde{\pi}}^-(v) = 0$  together with their only outgoing forward arc. The layout of these vertices is depicted by the pictogram  $\leftarrow\circ\rightarrow$ . Figure 5.4 provides examples for both kinds of trees.

Based on these two structures, we now simplify the forward path graph  $\tilde{G}$  by contracting the tree arcs and thereby *pooling* all vertices that belong to the same for-out-tree or for-in-tree in  $\tilde{\pi}$ . Let  $(u, v)$  be a forward arc in  $\tilde{\pi}$  such that  $u$  and  $v$  both have layout  $\rightarrow\circ\leftarrow$ . Then, we contract  $(u, v)$  as described in Section 4.10.1, which yields again a vertex  $uv$  with layout  $\rightarrow\circ\leftarrow$ . Likewise, if  $(u, v)$  is a forward arc in  $\tilde{\pi}$  such that  $u$  and  $v$  both have layout  $\leftarrow\circ\rightarrow$ , then the contraction of  $(u, v)$  produces in turn a vertex  $uv$  with layout  $\leftarrow\circ\rightarrow$ . We call the graph obtained after an exhaustive application of these operations a *pooled forward path graph* and denote it by  $\tilde{G}^\circ = (\tilde{V}^\circ, \tilde{A}^\circ)$ . We also derive a corresponding linear ordering  $\tilde{\pi}^\circ$  for  $\tilde{G}^\circ$  from  $\tilde{\pi}$  as follows: Before contracting an arc  $(u, v)$  connecting two vertices  $u, v$  with layout  $\rightarrow\circ\leftarrow$ , we modify  $\tilde{\pi}$  by moving  $v$  to the position  $\tilde{\pi}(u) + 1$ .

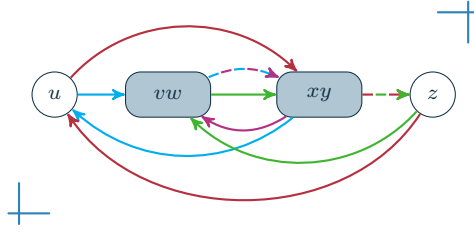


Figure 5.5: The linear ordering of the pooled forward path graph derived from the forward path graph shown in Figure 5.3(b).

Analogously, before contracting an arc  $(u, v)$  connecting two vertices  $u, v$  with layout  $\begin{smallmatrix} \rightarrow & \circ & \rightarrow \\ \leftarrow & \circ & \rightarrow \end{smallmatrix}$ , we modify  $\tilde{\pi}$  by moving  $u$  to the position  $\tilde{\pi}(v) - 1$ . Note that this neither affects the layouts of  $u$  and  $v$  nor the classification of any arc. Then,  $u$  and  $v$  are consecutive and a linear ordering for the resulting graph can be obtained as described in Section 4.10.1. Observe that in contrast to an actual vertex fusion, we do not perform a neutralization, i. e., except for the contracted arcs, all arcs incident to the involved vertices are preserved. As a consequence,  $\tilde{G}^\circ$  is not necessarily a simple graph. For ease of handling, we identify the arcs incident to the vertices before the contraction with their counterparts afterwards. The sets of backward arcs induced by  $\tilde{\pi}$  and  $\tilde{\pi}^\circ$  are thus the same.

Whereas the pooled forward path graphs of the forward path graphs depicted in Figure 5.1(b) and Figure 5.2 remain as they are, because the former has no vertex with layout  $\begin{smallmatrix} \rightarrow & \circ & \rightarrow \\ \leftarrow & \circ & \rightarrow \end{smallmatrix}$  or  $\begin{smallmatrix} \rightarrow & \circ & \rightarrow \\ \leftarrow & \circ & \rightarrow \end{smallmatrix}$  and the latter has a one-vertex for-in-tree consisting only of  $y$ , a difference is visible for the example shown in Figure 5.3(b): Here, the vertices  $v$  and  $w$  have layout  $\begin{smallmatrix} \rightarrow & \circ & \rightarrow \\ \leftarrow & \circ & \rightarrow \end{smallmatrix}$  and form a degenerate for-out-tree which is a path. Similarly, the vertices  $x$  and  $y$  both have layout  $\begin{smallmatrix} \rightarrow & \circ & \rightarrow \\ \leftarrow & \circ & \rightarrow \end{smallmatrix}$  and constitute a degenerate for-in-tree. To obtain the corresponding pooled forward path graph, the arcs  $(v, w)$  and  $(x, y)$  are contracted, such that new vertices  $vw$  and  $xy$  emerge. The result is depicted in Figure 5.5.

Consider a set of vertices  $V_{T^+}$  that forms a connected for-out-tree  $T^+$  in  $\tilde{\pi}$  and let  $v^\circ \in \tilde{V}^\circ$  be the vertex that pools all vertices in  $V_{T^+}$ . As  $T^+$  has exactly  $|V_{T^+}| - 1$  arcs, we obtain

$$\begin{aligned}
 f_{\tilde{\pi}^\circ}^-(v^\circ) &= 1, \\
 f_{\tilde{\pi}^\circ}^+(v^\circ) &= \sum_{v \in V_{T^+}} f_{\tilde{\pi}}^+(v) - (|V_{T^+}| - 1) = \sum_{v \in V_{T^+}} f_{\tilde{\pi}}^+(v) - |V_{T^+}| + 1, \\
 b_{\tilde{\pi}^\circ}^-(v^\circ) &= \sum_{v \in V_{T^+}} b_{\tilde{\pi}}^-(v), \\
 b_{\tilde{\pi}^\circ}^+(v^\circ) &= 0.
 \end{aligned} \tag{5.1}$$

Analogously, if  $T^-$  is a connected for-in-tree in  $\tilde{\pi}$  with vertex set  $V_{T^-}$  and  $v^\circ \in \tilde{V}^\circ$  is the vertex within  $\tilde{\pi}^\circ$  that pools all vertices in  $V_{T^-}$ , then

$$\begin{aligned} f_{\tilde{\pi}^\circ}^-(v^\circ) &= \sum_{v \in V_{T^-}} f_{\tilde{\pi}}^-(v) - (|V_{T^-}| - 1) = \sum_{v \in V_{T^-}} f_{\tilde{\pi}}^-(v) - |V_{T^-}| + 1, \\ f_{\tilde{\pi}^\circ}^+(v^\circ) &= 1, \\ b_{\tilde{\pi}^\circ}^-(v^\circ) &= 0, \\ b_{\tilde{\pi}^\circ}^+(v^\circ) &= \sum_{v \in V_{T^-}} b_{\tilde{\pi}}^+(v). \end{aligned} \tag{5.2}$$

### 5.1.3 The Polarized Forward Path Graph

Starting from the pooled forward path graph  $\tilde{G}^\circ$ , we obtain the so-called *polarized forward path graph*  $\tilde{G}^\circ = (\tilde{V}^\circ, \tilde{A}^\circ)$ . To this end, we *polarize* every vertex  $v \in \tilde{V}^\circ$  by splitting it into up to three component vertices or simply *components*  $v_s$ ,  $v_p$ , and  $v_t$ , such that  $v_s$  inherits all incoming and  $v_t$  inherits all outgoing backward arcs of  $v$ . All forward arcs of  $v$  are inherited by  $v_p$ . Additionally, we add  $b_{\tilde{\pi}}^-(v)$  parallel arcs  $(v_s, v_p)$  as well as  $b_{\tilde{\pi}}^+(v)$  parallel arcs  $(v_p, v_t)$ . For simplicity's sake, each component vertex of  $v$  is only present if it inherits at least one arc from  $v$  with a single exception: In case that  $v$  is isolated in  $\tilde{G}^\circ$ ,  $\tilde{G}^\circ$  contains only an isolated vertex  $v_p$ . The definition in particular also implies that if  $v$  is a pseudosource or a pseudosink or has no incident backward arcs, then  $v$  is not split at all. In these cases, we have  $v_s = v$  or  $v_t = v$  or  $v_p = v$ , respectively. Note that the vertical split of a right-blocking vertex  $v$ , which already occurred for the construction of the forward path graph  $\tilde{G}$ , produces a pseudosink  $v_l$  and a pseudosource  $v_r$  according to the definition in Section 4.5.2. To conform with the above naming scheme, we set  $v_s = v_r$  and  $v_t = v_l$  in this case. Figure 5.6 depicts the general case of a vertex  $v \in \tilde{V}^\circ$  that is split into three component vertices  $v_s, v_p, v_t \in \tilde{V}^\circ$  and the slightly different situation if  $v \in V$  is right-blocking in  $\pi$ .

Along with  $\tilde{G}^\circ$ , we maintain a corresponding linear ordering  $\tilde{\pi}^\circ$  of  $\tilde{G}^\circ$ , which is derived from  $\tilde{G}^\circ$  such that  $v_s$ ,  $v_p$ , and  $v_t$  take the relative position of  $v$  in  $\tilde{\pi}^\circ$  and  $\tilde{\pi}^\circ(v_s) \leq \tilde{\pi}^\circ(v_p) \leq \tilde{\pi}^\circ(v_t)$ , where applicable. Observe that if  $v$  is right-blocking in  $\pi$ , then  $\tilde{\pi}^\circ(v_l) \leq \tilde{\pi}^\circ(v_r)$  already, so we keep the ordering  $\tilde{\pi}^\circ(v_t) \leq \tilde{\pi}^\circ(v_s)$  in this exceptional case. Consequently, if existing,  $v_s$  always is a pseudosource in  $\tilde{\pi}^\circ$ ,  $v_t$  is a pseudosink, and  $v_p$  has only forward paths passing through it, but no incident backward arcs and is therefore called a *passage* vertex or simply a *passage*. Note that by definition, the passage vertex may also be isolated. Indeed, every vertex  $v \in \tilde{V}^\circ$  is either a



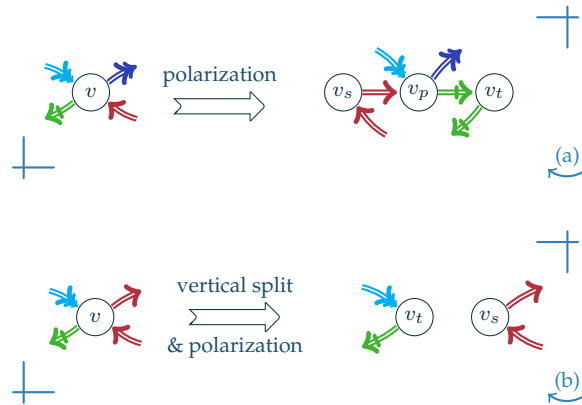


Figure 5.6: Polarization of a vertex  $v$  in  $\tilde{G}^\omega$  with incoming and outgoing forward and backward arcs into its component vertices  $v_s, v_p,$  and  $v_t$  in  $\tilde{G}^{\omega\tilde{}}$  (a) as well as the special case where  $v$  is right-blocking in  $G$  and therefore only has component vertices  $v_l = v_t$  and  $v_r = v_s$  in  $\tilde{G}^{\omega\tilde{}}$  (b).

pseudosource, a pseudosink, or a passage in  $\tilde{\pi}^\omega$ . Note that the polarization preserves all backward arcs as well as all forward paths selected during the construction of  $\tilde{G}$ . The additional arcs added between  $v_s$  and  $v_p$  as well as  $v_p$  and  $v_t$  may be regarded as an extension of the forward paths of the respective incident backward arcs.

Figure 5.7 shows two examples for polarized forward path graphs: The graph in Figure 5.7(a) is obtained from the forward path graph in Figure 5.2, which is the same as its pooled forward path graph except for the relabelling of the split vertices  $w_l$  and  $w_r$  into  $w_t$  and  $w_s$ , respectively, as described above. The second graph, shown in Figure 5.7(b), is constructed from the pooled forward path graph that served as an example in the previous subsection.

For a better visualization, we use special pictograms for the components of the pooled vertices that were introduced in  $\tilde{G}^\omega$ : If  $v$  has layout  $\rightarrow\circ\leftarrow$  in  $\tilde{\pi}^\omega$ , then  $v$  is split into  $v_s$  and  $v_p$  during the construction of  $\tilde{G}^{\omega\tilde{}}$ , which have combined layout  $\rightarrow\circ\leftarrow$  in  $\tilde{\pi}^{\omega\tilde{}}$ . Analogously, if  $v$  has layout  $\leftarrow\circ\rightarrow$  in  $\tilde{\pi}^\omega$ , then  $v$  is split into its component vertices  $v_p$  and  $v_t$ , which have combined layout  $\leftarrow\circ\rightarrow$  in  $\tilde{\pi}^{\omega\tilde{}}$ .

### 5.1.4 The Truncated Forward Path Graph

The last auxiliary graph that we use is constructed in turn from the polarized forward path graph  $\tilde{G}^{\omega\tilde{}}$ . The *truncated forward path graph*  $\tilde{G}_{tr}^{\omega\tilde{}} = (\tilde{V}_{tr}^{\omega\tilde{}}, \tilde{A}_{tr}^{\omega\tilde{}})$  is the subgraph obtained from  $\tilde{G}^{\omega\tilde{}}$  by removing all vertices that are pseudosinks with respect to  $\tilde{\pi}^{\omega\tilde{}}$ . In other words,

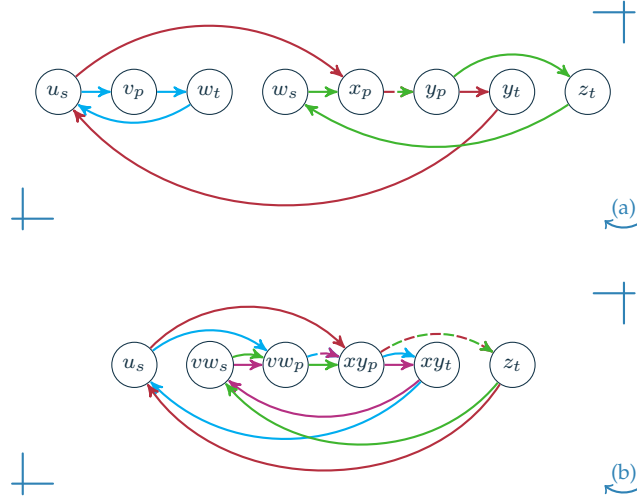


Figure 5.7: The linear orderings of the polarized forward path graphs derived from the (pooled) forward path graphs shown in Figure 5.2 in (a) and Figure 5.5 in (b).

$\tilde{G}_{\text{tr}}^{\emptyset} = \tilde{G}_{S \cup P}^{\emptyset}$ , where  $S$  is the set of pseudosources and  $P$  is the set of passage vertices with respect to  $\tilde{\pi}^{\emptyset}$ . As the backward arcs are removed together with their tails,  $\tilde{G}_{\text{tr}}^{\emptyset}$  is acyclic. Furthermore, every vertex  $v$  that is a pseudosource in  $\tilde{\pi}^{\emptyset}$  is a source in  $\tilde{G}_{\text{tr}}^{\emptyset}$  with  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^{+}(v) = b_{\tilde{\pi}^{\emptyset}}^{-}(v)$ , because, first, the forward paths chosen for  $\tilde{G}$  are pairwise arc-disjoint at the backward arcs' head. Second, no pseudosource can have a forward arc to a pseudosink in  $\tilde{G}^{\emptyset}$ : Otherwise, this forward arc's tail must be a pseudosource and its head must be a pseudosink in  $\tilde{G}^{\emptyset}$ , which would imply that  $\tilde{G}$  and hence also  $G$  had a pair of anti-parallel arcs. In consequence, only passage vertices in  $\tilde{\pi}^{\emptyset}$  may "lose" outgoing (forward) arcs during the transition from  $\tilde{G}^{\emptyset}$  to  $\tilde{G}_{\text{tr}}^{\emptyset}$ , which is also why  $\tilde{G}_{\text{tr}}^{\emptyset}$  contains exactly the same isolated vertices as  $\tilde{G}$  and there are no sources in  $\tilde{G}_{\text{tr}}^{\emptyset}$  that were not pseudosources in  $\tilde{G}^{\emptyset}$ . To facilitate indication, we call a spanning subgraph  $H \subseteq \tilde{G}_{\text{tr}}^{\emptyset}$  of  $\tilde{G}_{\text{tr}}^{\emptyset}$  *source-preserving* if there is a one-to-one correspondence between the sources in  $H$  and the pseudosources in  $\tilde{\pi}^{\emptyset}$  and  $d_H^{+}(v) = b_{\tilde{\pi}^{\emptyset}}^{-}(v)$  for every pseudosource  $v \in \tilde{V}^{\emptyset}$ . Hence,  $\tilde{G}_{\text{tr}}^{\emptyset}$  is itself source-preserving.

For visualization purposes, we maintain a linear ordering  $\tilde{\pi}_{\text{tr}}^{\emptyset}$  of  $\tilde{G}_{\text{tr}}^{\emptyset}$ , which is obtained from  $\tilde{\pi}^{\emptyset}$  by skipping all pseudosinks. Figure 5.8 shows the truncated forward path graphs of all forward path graphs that were given exemplarily in Section 5.1.1. The forward path graph depicted in Figure 5.1(b) is at the same time also a pooled and a polarized forward path graph, except for the relabelling of the vertices. Figure 5.8(a) displays the corresponding truncated forward path graph. For the polarized forward

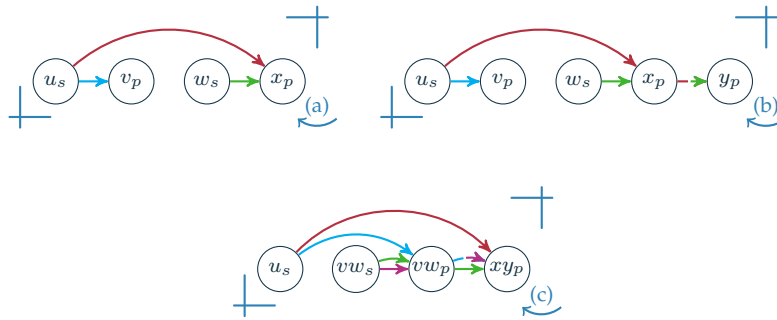


Figure 5.8: The linear orderings of the truncated forward path graphs derived from the (polarized) forward path graphs shown in Figure 5.1(b) in (a), Figure 5.7(a) in (b), and Figure 5.7(b) in (c).

path graphs shown in Figure 5.7, their truncated counterparts are given in Figure 5.8(b) and Figure 5.8(c).

## 5.2 Subcubic Graphs

We start our analysis of the cardinality of optimal feedback arc sets with the sparsest type of graphs, which are the subcubic ones. It might be objected here that there are even sparser graphs with a maximum vertex degree of two or one. For a graph to be strongly connected, however, every vertex needs at least one incoming and one outgoing arc. Hence, a vertex degree of one is never possible, and a graph whose vertices each have degree exactly two and which is strongly connected is a simple cycle with an optimal feedback arc set of cardinality one.

### 5.2.1 A Tight Bound

In order to estimate the number of backward arcs in dependency on the size of the input graph, we will use the following notion: Think of a graph along with a linear ordering and imagine that every backward arc produces a pebble that is initially located at its head. This pebble is then transported along a forward path to the backward arc's tail and put down on one of its vertices that is neither incident to another backward arc nor has another pebble already been deposited there. This vertex can then uniquely be assigned to the backward arc that produced the pebble. We will show that such an assignment is possible for every subcubic graph, and that we can also assign each backward arc its head and tail.

---

**Theorem 5.1**

Every  $\Psi$ -optimal linear ordering of a subcubic graph having  $n$  vertices induces at most  $\lfloor \frac{n}{3} \rfloor$  backward arcs.

---

*Proof.* Consider a  $\Psi$ -optimal linear ordering  $\pi$  of a subcubic graph  $G = (V, A)$ . As  $\pi$  respects the Nesting Property and the Eliminateable Layouts Property, it induces one of the seven different layouts shown in Table 5.1 on every vertex  $v \in V$ . Note that the layouts  $\rightarrow\circ\leftarrow$  and  $\rightarrow\circ\rightarrow$  are eliminateable and therefore beyond consideration.

We first construct a forward path graph  $\vec{G} = (\vec{V}, \vec{A})$  of  $G$  with respect to  $\pi$  as described in Section 5.1.1. Let  $\vec{\pi}$  denote the corresponding linear ordering of  $\vec{G}$  and observe that  $\vec{V} = V_{\text{sp}/r} = V$ , because no vertex can have both an incoming and an outgoing backward arc in  $\pi$  and  $\pi$  respects the Eliminateable Layouts Property. With every vertex being incident to at most one backward arc, we can assume that  $\vec{G}$  contains exactly one forward path per backward arc. Due to the limited number of possible vertex layouts,  $\vec{G}$  cannot contain a vertex with layout  $\rightarrow\circ\leftarrow$  or  $\leftarrow\circ\rightarrow$ , which implies that the corresponding pooled forward path graph  $\vec{G}^\circ$  equals  $\vec{G}$ . Moreover, every vertex in  $\vec{G}$  already is either a pseudosource, a pseudosink, or a passage vertex. Hence,  $\vec{G}^\circ = \vec{G}^\circ = \vec{G}$ . Note that no vertex with an outgoing backward arc can be part of a forward path that does not end at it. We conduct the analysis of  $\vec{G}$  using the truncated forward path graph  $\vec{G}_{\text{tr}}^\circ = (\vec{V}_{\text{tr}}^\circ, \vec{A}_{\text{tr}}^\circ)$ , which can in this case be obtained directly from  $\vec{G}$  by removing all pseudosinks (cf. Section 5.1.4).

To estimate the number of backward arcs in  $\pi$ , we look at the delta degrees of the vertices in  $\vec{G}_{\text{tr}}^\circ$ . Recall that the delta degree  $\delta(v)$  of a vertex  $v$  is defined as  $d^+(v) - d^-(v)$ . To ease notation, we stipulate that all denominations of vertex degrees, i. e.,  $\delta(v)$ ,  $d^+(v)$ , and  $d^-(v)$ , used in the remainder of this proof only refer to the truncated forward path graph  $\vec{G}_{\text{tr}}^\circ$ . Figure 5.9 provides an example of a graph, a  $\Psi$ -optimal (here even optimal) linear ordering, and the corresponding truncated forward path graph.

**Table 5.1:** Layouts induced by a  $\Psi$ -optimal linear ordering on a vertex  $v$  of a subcubic graph as pictograms and 4-tuples.

---

$d(v) = 2:$	$\leftarrow\circ\rightarrow$ (0, 1, 1, 0),	$\rightarrow\circ\leftarrow$ (1, 0, 0, 1),	$\rightarrow\circ\rightarrow$ (1, 0, 1, 0)
$d(v) = 3:$	$\leftarrow\circ\leftarrow$ (0, 2, 1, 0),	$\rightarrow\circ\leftarrow$ (2, 0, 0, 1),	$\rightarrow\circ\rightarrow$ (1, 2, 0, 0), $\rightarrow\circ\rightarrow$ (2, 1, 0, 0)

---

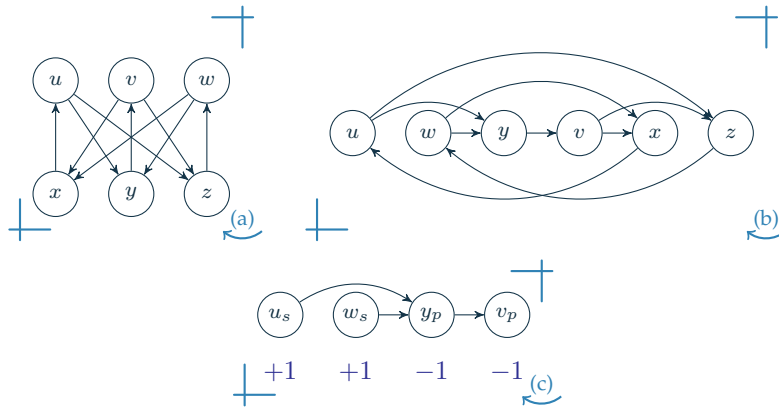


Figure 5.9: The 3-fence (a), an optimal linear ordering of it (b), and the (in this case unique) truncated forward path graph with delta degrees (c).

Consider a vertex  $v \in V$  that has an incoming backward arc in  $\pi$ . Looking up the table of possible layouts of vertices with an incident backward arcs, we find that the layout of  $v$  must be either  $\alpha_{\leftarrow}^{\rightarrow}$  or  $\alpha_{\leftarrow}^{\leftarrow}$ , i. e.,  $v$  is a pseudosource and  $b_{\pi}^{-}(v) = b_{\pi}^{-\rightarrow}(v) = 1$ . Furthermore,  $v = v_s$  in  $\tilde{G}_{\text{tr}}^{\emptyset}$ . As  $\tilde{G}_{\text{tr}}^{\emptyset}$  is source-preserving,  $v_s$  is a source in  $\tilde{G}_{\text{tr}}^{\emptyset}$  with  $d^{+}(v_s) = 1$  and thus,  $\delta(v_s) = 1$ . Besides, there are no other sources in  $\tilde{G}_{\text{tr}}^{\emptyset}$ .

Consider now a non-isolated vertex  $v_p \in \tilde{V}_{\text{tr}}^{\emptyset}$  whose original vertex  $v$  has no incident backward arcs in  $\tilde{G}$  and suppose that in the truncated forward path graph,  $\delta(v_p) \leq -2$ . Recall that  $\tilde{G}$  is a union of backward arcs and corresponding forward paths, i. e.,  $v$  has at least one incoming and one outgoing arc in  $\tilde{G}$ . Furthermore, it can be incident to at most three arcs, hence, if  $\delta(v_p) \leq -2$ , the layout induced on  $v$  by  $\tilde{\pi}$  must be  $\beta_{\rightarrow}^{\rightarrow}$  and  $v$ 's outgoing forward arc  $(v, t) \in \tilde{A}$  is not contained in  $\tilde{A}_{\text{tr}}^{\emptyset}$ . This in turn implies that  $t$  is a pseudosink with  $b_{\tilde{\pi}}^{+}(t) = b_{\tilde{\pi}}^{+\leftarrow}(t) = 1$  and is therefore not part of  $\tilde{G}_{\text{tr}}^{\emptyset}$ . Moreover, both incoming arcs of  $v$  must be contained in  $\tilde{G}_{\text{tr}}^{\emptyset}$  to obtain  $\delta(v_p) = -2$  and no smaller delta degree is possible. In consequence,  $\tilde{G}$  contains two forward paths for the single backward arc whose tail is  $t$ , a contradiction to the fact that the forward path graph for a subcubic graph contains exactly one forward path per backward arc. Thus, we obtain for every vertex  $v_p$  of  $\tilde{V}_{\text{tr}}^{\emptyset}$  a delta degree of  $\delta(v_p) \geq -1$ , because an isolated vertex always has a delta degree of zero. As  $\tilde{G}_{\text{tr}}^{\emptyset}$  contains no other vertices, we can therefore conclude that  $\forall v \in \tilde{V}_{\text{tr}}^{\emptyset} : \delta(v) \in \{-1, 0, 1\}$ .

Recall from [Section 3.2](#) that for every graph the sum of the delta degrees of its vertices equals zero. Thus,

$$\sum_{v \in \tilde{V}_{\text{tr}}^{\emptyset}} \delta(v) = 0.$$

Consequently, for every vertex  $v$  with  $\delta(v) = +1$ ,  $\tilde{G}_{\text{tr}}^{\emptyset}$  must contain another vertex  $u$  with  $\delta(u) = -1$ . Furthermore, the vertex set of  $\tilde{G}^{\emptyset} = \tilde{G}$  and  $G$  is exactly the same, whereas that of  $\tilde{G}_{\text{tr}}^{\emptyset}$  differs only in the removal of all pseudosinks. This enables us to assign each backward arc  $(u, v)$  three exclusive vertices of  $G$ : Its tail  $u$ , its head  $v$ , which is preserved as the source  $v_s$  in  $\tilde{G}_{\text{tr}}^{\emptyset}$ , and a passage vertex  $w$  of  $\tilde{G}$  such that the delta degree of  $w_p$  in  $\tilde{G}_{\text{tr}}^{\emptyset}$  is  $\delta(w_p) = -1$ , which thereby counterbalances the delta degree  $\delta(v_s) = +1$  of  $v_s$ . Hence,

$$|\tilde{\pi}| \leq \lfloor \frac{n}{3} \rfloor,$$

as the number of backward arcs is always integer. □

So the pebble that each backward arc produces and that must be transported along a forward path until it is dropped on one of its vertices corresponds in the proof to a source in the truncated forward path graph with delta degree  $+1$ , whereas the deposit vertex is the balancing one with delta degree  $-1$ .

By [Corollary 4.21](#), we can construct a  $\Psi$ -optimal linear ordering of a subcubic graph in  $\mathcal{O}(n^3)$  time. Thus, we obtain<sup>1</sup>:

---

**Corollary 5.1**

There is an  $\mathcal{O}(n^3)$ -time algorithm to construct a feedback arc set with cardinality at most  $\lfloor \frac{n}{3} \rfloor$  for a subcubic graph.

---

A directed triangle or the graph depicted in [Figure 5.9](#), which has an optimal fractional feedback arc set of size  $\frac{3}{2}$  using one half of each of the arcs  $(x, u)$ ,  $(y, v)$ , and  $(z, w)$  already testify that the bound of  $\lfloor \frac{n}{3} \rfloor$  is tight for  $n = 3$  and  $n = 6$ , respectively. In fact, there is a subcubic graph that meets this bound for all values of  $n \geq 3$ :

---

**Lemma 5.1**

For every  $n \geq 3$  there is a subcubic graph on  $n$  vertices whose optimal feedback arc set has cardinality at least  $\lfloor \frac{n}{3} \rfloor$ .

---

<sup>1</sup>This result has also been published in an earlier conference article [[HBA13](#)].

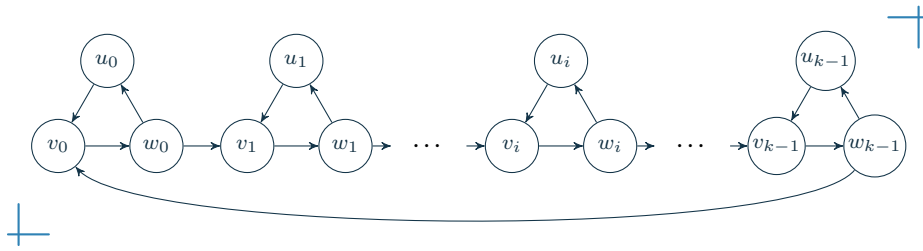


Figure 5.10: Construction of subcubic graphs with  $n$  vertices and  $\tau = \lfloor \frac{n}{3} \rfloor$ .

*Proof.* Let  $k = \lfloor \frac{n}{3} \rfloor$  and construct a graph  $G_n$  that consists of  $k$  vertex-disjoint directed triangles such that the  $i$ th triangle uses the vertices  $\{u_i, v_i, w_i\}$  and the arcs  $\{(u_i, v_i), (v_i, w_i), (w_i, u_i)\}$ , where  $0 \leq i < k$ . Additionally,  $G_n$  contains an arc  $(w_i, v_{i+1})$  for every  $0 \leq i < k - 1$  and an arc  $(w_{k-1}, v_0)$ . The construction is also visualized in Figure 5.10. Finally, perform an arc subdivision on  $n - 3k$  of the arcs as described in Section 4.10.1. Then,  $G_n$  is a strongly connected, subcubic graph on  $n$  vertices with  $k$  vertex-disjoint and hence also arc-disjoint cycles. Consequently,  $\tau_{G_n} \geq k = \lfloor \frac{n}{3} \rfloor$ .  $\square$

Note that the graphs constructed in Lemma 5.1 can be easily augmented such that at most one vertex has a degree less than three by adding, e. g., arcs that connect two vertices  $u_i$  and  $u_{\lfloor \frac{k}{2} \rfloor + i}$  for  $0 \leq i < \frac{k}{2}$ . As every optimal linear ordering is also  $\Psi$ -optimal by Theorem 4.1, the combination of Theorem 5.1 and Lemma 5.1 yields<sup>1</sup>:

---

**Theorem 5.2**

The cardinality of an optimal feedback arc set of a subcubic graph having  $n$  vertices is at most  $\lfloor \frac{n}{3} \rfloor$  and this bound is tight for all values of  $n \geq 3$ .

---

### 5.2.2 On the Approximation Ratio

In the previous subsection, we showed that every optimal linear ordering of a cubic graph induces a feedback arc set of cardinality at most one third of the number of vertices, and that this bound can be achieved by an efficient algorithm, *PsiOptCubic*. A natural question to ask here is: How close is the solution of *PsiOptCubic* to an optimal solution? Or, maybe more importantly, how bad can it be?

The attentive reader may have noticed that the proof of the bound in the previous subsection was based only on two properties: the Path Property and the Elimenable Layouts

<sup>1</sup>A slightly weaker version of this result has also been published in an earlier conference article [HBA13].

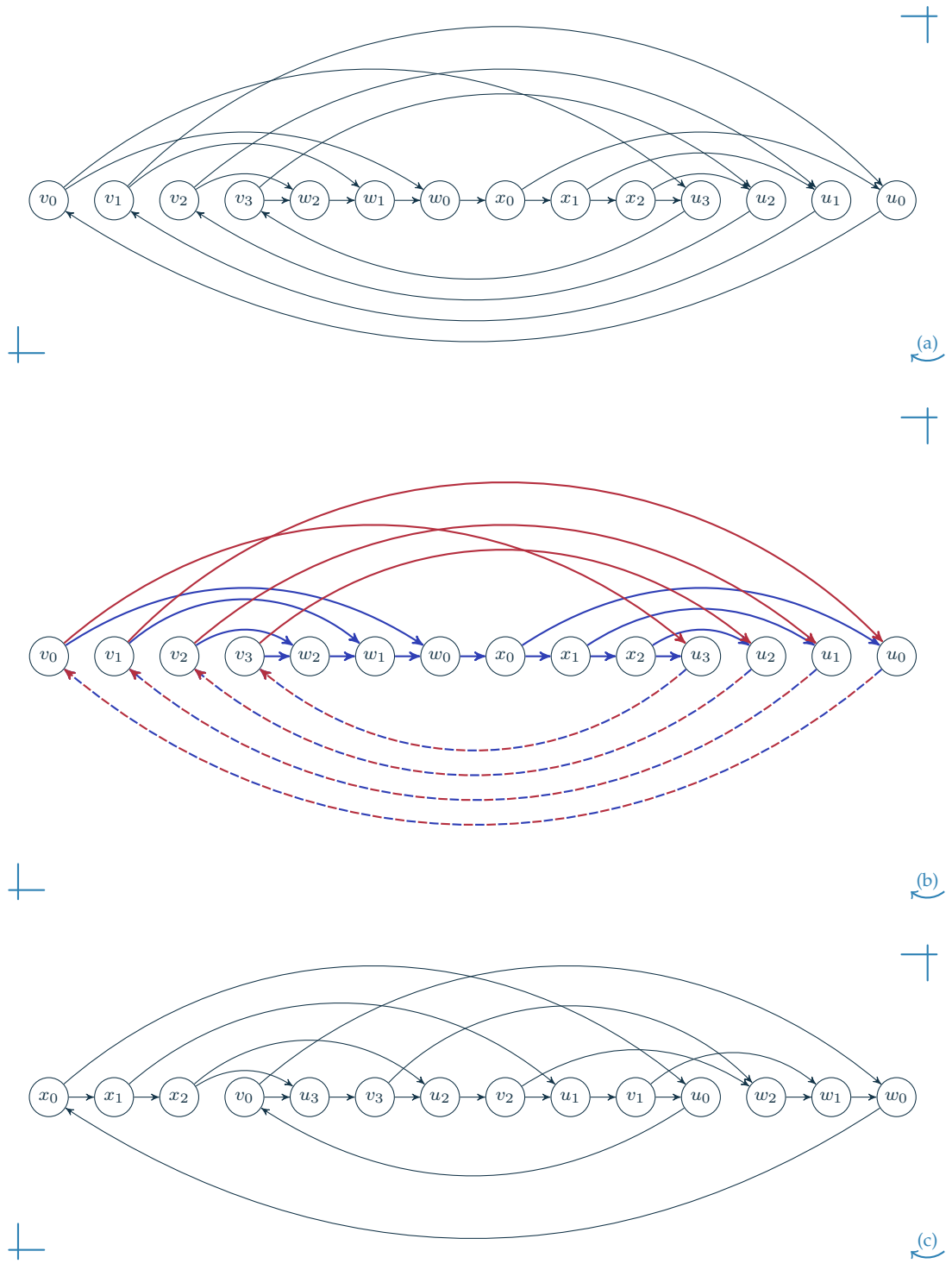


Figure 5.11: A cubic graph with a  $\Psi$ -optimal linear ordering of size  $\frac{n+2}{4}$  (a), a visualization of some of its cycles (b), and an optimal solution needing only two backward arcs (c).



Property. Indeed, these two properties do not suffice to construct linear orderings that are guaranteedly close to an optimal solution:

---



---

**Lemma 5.2**

For every  $k \geq 2$ , there is a cubic graph on  $n = 4k - 2$  vertices with a  $\Psi$ -optimal linear ordering  $\pi$  such that  $\frac{|\pi|}{\tau} \in \Theta(n)$ .

---

*Proof.* We construct a graph  $G = (V, A)$  with  $n = 4k - 2$  vertices as it is depicted in [Figure 5.11\(a\)](#) for  $k = 4$ : For every  $0 \leq i < k$ ,  $G$  has two vertices  $u_i$  and  $v_i$  that are connected by an arc  $(u_i, v_i)$ . Additionally, there is an arc  $(v_i, u_{i-1})$  for every  $1 \leq i < k$  and there is an arc  $(v_0, u_{k-1})$ . Furthermore, for every  $0 \leq i < k - 1$ ,  $G$  has two vertices  $w_i$  and  $x_i$  as well as the arcs  $(v_i, w_i)$  and  $(x_i, u_i)$ . Next, add the arcs  $(w_{i+1}, w_i)$  and  $(x_i, x_{i+1})$  for every  $0 \leq i < k - 2$ . Finally, we insert the arcs  $(v_{k-1}, w_{k-2})$ ,  $(w_0, x_0)$ , and  $(x_{k-2}, u_{k-1})$ . In result,  $G$  is cubic.

Let  $\pi = (v_0, v_1, \dots, v_{k-1}, w_{k-2}, w_{k-3}, \dots, w_0, x_0, x_1, \dots, x_{k-2}, u_{k-1}, u_{k-2}, \dots, u_0)$  be the linear ordering which is already depicted in [Figure 5.11\(a\)](#) for  $k = 4$ . Then,  $\pi$  is a  $\Psi$ -optimal linear ordering of  $G$  and  $|\pi| = k$ . Observe that all forward paths with respect to  $\pi$  use the arc  $(w_0, x_0)$ . Furthermore, there is a cycle consisting only of the vertices  $u_i$  and  $v_i$ ,  $0 \leq i < k$ . [Figure 5.11\(b\)](#) highlights these cycles: The arcs of the cycles containing the forward paths are colored blue, whereas the cycle induced by the vertex set  $\bigcup_{0 \leq i < k} \{u_i, v_i\}$  is colored red. The backward arcs are part of the blue cycles as well as the red cycle and are therefore bicolored. Note that there are further cycles that use both pure red and pure blue arcs and, because of the latter, necessarily also  $(w_0, x_0)$ . Subsequently, however, the arc  $(w_0, x_0)$  together with one of the current backward arcs,  $(u_0, v_0)$ , e. g., cover all cycles in  $G$  and removing them yields an acyclic subgraph of  $G$ . A corresponding linear ordering is depicted in [Figure 5.11\(c\)](#). In conclusion,  $\tau = 2$  for all graphs constructed like  $G$ , which implies that  $\frac{|\pi|}{\tau} = \frac{k}{2} = \frac{n+2}{8} \in \Theta(n)$ .  $\square$

Another property that has been introduced in [Chapter 4](#) but has not played a role so far is the Fusion Property. As a matter of fact, if we choose the vertices to be fused carefully, we can use it to construct a better solution for the graphs constructed in the proof of [Lemma 5.2](#): Let  $G = (V, A)$  be a graph constructed according to the description in the proof of [Lemma 5.2](#). We partition the vertices in  $Y = \{v_i \mid 0 \leq i < k\}$  into up to three sets  $Z_1, Z_2, Z_3$  such that two vertices  $v_j, v_l \in Y$ ,  $v_j \neq v_l$ , are in different partitions if there is a vertex  $u \in V$  such that both  $(v_j, u), (u, v_l) \in A$ . More descriptively,  $v_j$  and  $v_l$

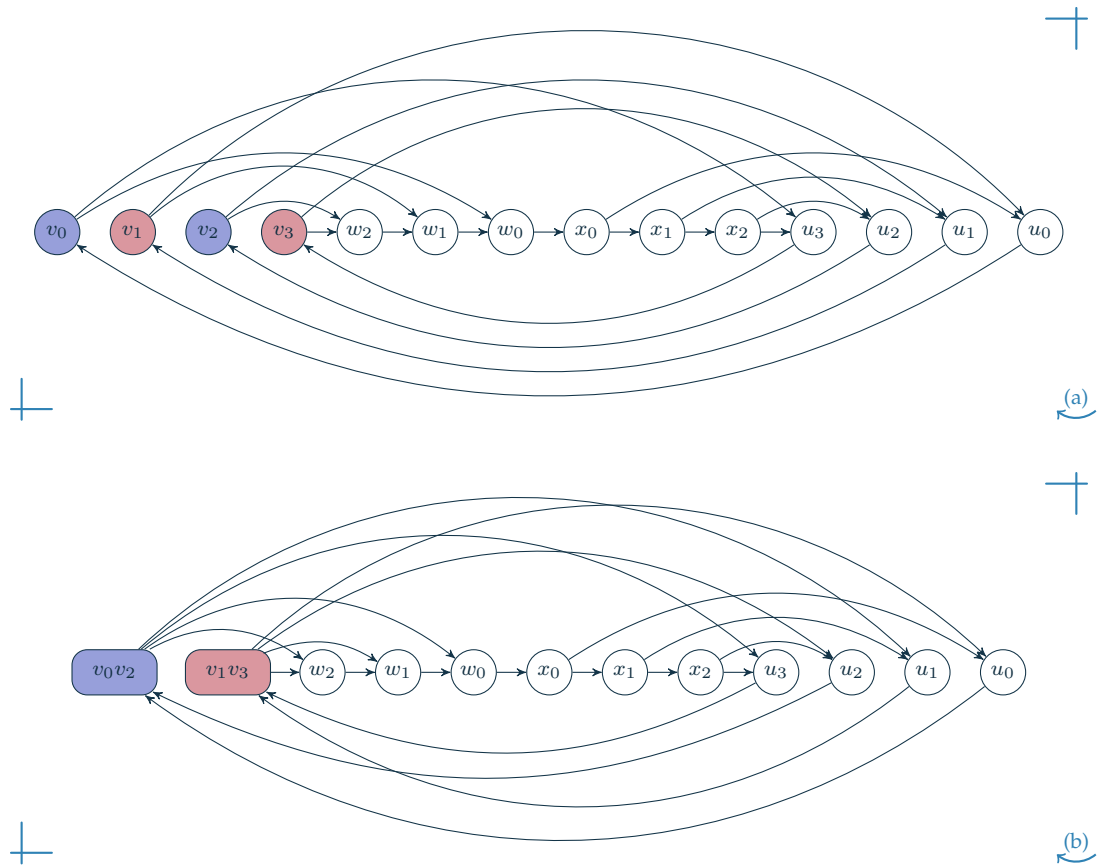


Figure 5.12: A suitable partition of the vertices  $\{v_i \mid 0 \leq i < k\}$  (a) and the graph and linear ordering obtained after the fusion of all vertices that are in the same partition (b).

reside in different partitions if they have a common neighbor  $u$  such that one of them has an outgoing arc to  $u$  and the other has an incoming arc from  $u$ . A suitable partition can be found efficiently by constructing an undirected helper graph  $G_H$  that has  $Y$  as vertex set and an edge between two vertices in  $Y$  if they may not be contained in the same partition. In case of graphs constructed like  $G$ , every vertex in  $G_H$  has degree at most two. Hence,  $G_H$  is a collection of simple paths and cycles and is thus 3-colorable. In fact,  $G_H$  is only a simple cycle of length  $k$  here. Now, any 3-coloring of  $G_H$  such that two adjacent vertices receive different colors corresponds to a suitable partition of the vertices in  $Y$ . [Figure 5.12\(a\)](#) depicts such a coloring for  $G$ . As the cycle in  $G_H$  for the example graph has length  $k = 4$  and is thus even, two colors suffice. Next, we obtain a new linear ordering  $\pi'$  from  $\pi$  by reordering the vertices in  $Y$  such that vertices in the same partition (i. e., of the same color) are consecutive. Note that  $G$  does not have any arcs between two vertices of  $Y$ , which is why  $\mathcal{B}_{\pi'} = \mathcal{B}_{\pi}$ . Finally, fuse all vertices of  $Y$  that are in the same partition. Due to the construction of  $Z_1$ ,  $Z_2$ , and  $Z_3$ , there is a one-to-one correspondence between the arcs in the fused graph and the arcs in  $G$ , i. e., no neutralization of arcs occurs. The fused graph along with the respective linear ordering is shown in [Figure 5.12\(b\)](#) and we observe that neither  $v_0v_2$  nor  $v_1v_3$  has two arc-disjoint forward paths for its incident backward arcs, i. e., the linear ordering of the fused graph does not respect the Multipath Property and hence can be improved.

In case of the example graph, re-establishing the Multipath Property on the fused graph, undoing the fuse operation, and re-establishing the Path Property indeed yields an optimal linear ordering. However, the above construction of the fused graph fails already if we replace the arcs  $(v_i, u_{i-1})$ , for every  $1 \leq i < k$  as well as  $(v_0, u_{k-1})$  by dedicated paths of length at least two. Then, all vertices can be in the same partition and the fused graph has forward paths that use these new paths. It therefore remains an interesting open question of whether the Fusion Property or any other property can be used in general to efficiently construct a linear ordering of a cubic graph that improves the gap shown in [Lemma 5.2](#).

### 5.3 From Subcubic to General Graphs

Subcubic graphs certainly constitute the class of the sparsest interesting graphs to consider as input to the LINEAR ORDERING problem. As we have seen in [Chapter 4](#), however, powerful properties such as the Blocking Vertices Property, the Multipath Property, and the Multipath Blocking Vertices Property, grasp at nothing on these graphs

because they are just too sparse: For cubic graphs, an optimal linear ordering cannot induce blocking vertices with a non-eliminable layout and as every vertex can be incident to at most one backward arc, the pairwise arc-disjoint forward paths demanded by the Multipath Property are trivially guaranteed by the combination of the much simpler Nesting Property and Path Property.

In this section, we raise the maximum vertex degree and propose a suitable extension of the schema used for the analysis of cubic graphs.

### 5.3.1 Pebble Transportation in Supercubic Graphs

Remember the idea about the pebble that is produced by each backward arc and transported along one of its forward path, which we used in the previous section to prove a tight bound on subcubic graphs? There, we were able to exclusively assign a backward arc its tail, its head, and one further vertex on its forward path. Looking at vertices of degree four or more, however, the same procedure is no longer possible, because a vertex may have multiple incoming and outgoing backward arcs. On the other hand, the Nesting Property also limits the number of incident backward arcs to at most half of a vertex's degree.

Motivated by this observation, we will extend the notion of "pebble transportation" as follows: Every backward arc  $b$  still produces a pebble that initially lies at its head and must be transported along a forward path of  $b$  until it can be deposited on one of the forward path's vertices. However, a vertex is no longer exclusively associated with one backward arc, but must be shared.

Let  $G = (V, A)$  be a graph along with a linear ordering  $\pi$ . To keep track of a vertex's assignments, we introduce three mappings  $\alpha_h, \alpha_t, \alpha_d : V \rightarrow \mathbb{N}$ , where  $\alpha_h(v)$  denotes the number of backward arcs whose head is  $v$ ,  $\alpha_t(v)$  denotes the number of backward arcs whose tail is  $v$ , and  $\alpha_d(v)$  denotes the number of pebbles that are deposited on  $v$ . Furthermore, let  $\alpha(v) = \alpha_h(v) + \alpha_t(v) + \alpha_d(v)$  be the vertex-wise sum of these values. The value of  $\alpha(v)$  may be regarded as a fee or charge that is demanded by a vertex  $v$ .

In case of subcubic graphs, we found that  $\alpha(v) \leq 1$  for each vertex  $v$  and hence, every backward arc can be associated with three exclusive vertices. For higher vertex degrees, we may think of halves and thirds of vertices, e. g., if  $\alpha(v) = 2$  or  $\alpha(v) = 3$ , respectively.

Let us now consider the necessary conditions for these mappings to be useful for the estimation of  $|\pi|$ . Whereas the definition of  $\alpha_h$  and  $\alpha_t$  immediately imply that

$$\sum_{v \in V} \alpha_h(v) = \sum_{v \in V} \alpha_t(v) = |\pi|, \quad (5.3)$$

we must take care that

$$\sum_{v \in V} \alpha_d(v) \geq |\pi|. \quad (5.4)$$

We say that an assignment scheme  $\alpha$  that consists of the mappings  $\alpha_h$ ,  $\alpha_t$ , and  $\alpha_d$  is *admissible* with respect to a linear ordering  $\pi$  if it fulfills Equation (5.3) and Equation (5.4).

Employing this to count the number of backward arcs then yields:

---



---

**Proposition 5.1**

Let  $\pi$  be a linear ordering of a graph  $G = (V, A)$  along with an admissible assignment scheme  $\alpha$ . Then,

$$\tau_G \leq |\pi| \leq \lfloor \frac{1}{3} \sum_{v \in V} \alpha(v) \rfloor.$$


---

### 5.3.2 A General Assignment Scheme

Based on the prerequisites that we just developed and the proof strategy for the subcubic case, we propose the following general assignment scheme:

---



---

**Lemma 5.3**

Let  $G = (V_G, A_G)$  be a graph with linear ordering  $\pi$  that respects the Multipath Blocking Vertices Property, let  $\tilde{G}^\circ$  be a corresponding polarized forward path graph of  $G$  with linear ordering  $\tilde{\pi}^\circ$ , and let  $H = (V_H, A_H) \subseteq \tilde{G}_{\text{tr}}^\circ$  be a spanning, source-preserving subgraph of the respective truncated forward path graph. Then,  $\alpha(v) = \alpha_h(v) + \alpha_t(v) + \alpha_d(v)$  such that



$$\alpha_h(v) = b_\pi^-(v),$$

$$\alpha_t(v) = b_\pi^+(v), \text{ and}$$

$$\alpha_d(v) = \begin{cases} \max\{-\delta_H(v_p), 0\}, & \text{if } v_p \in V_H \text{ is } v\text{'s passage component,} \\ \frac{1}{k} \cdot \max\{-\delta_H(u_p), 0\}, & \text{if } v \text{ and } k-1 \text{ other vertices are pooled} \\ & \text{within a vertex } u \text{ and } u_p \in V_H \text{ is} \\ & \text{} u\text{'s passage component,} \\ 0, & \text{otherwise.} \end{cases}$$

for each vertex  $v \in V_G$  is an admissible assignment scheme.

---

*Proof.* Recall from Section 5.1 that the polarized forward path graph  $\tilde{G}^\circ$  is obtained from the right-blocking split graph  $G_{\text{sp}/r}$  of  $G$  by first constructing a forward path graph  $\tilde{G} = (\tilde{V}, \tilde{A}) \subseteq G_{\text{sp}/r}$  along with a linear ordering  $\tilde{\pi}$  such that every set of backward arcs with a common head has pairwise arc-disjoint forward paths. Afterwards, the pooling of vertices with layout  and  in  $\tilde{\pi}$  yields the pooled forward path graph  $\tilde{G}^\circ = (\tilde{V}^\circ, \tilde{A}^\circ)$  together with the linear ordering  $\tilde{\pi}^\circ$ . By splitting every vertex  $v$  in  $\tilde{G}^\circ$  into up to three components  $v_s, v_p,$  and  $v_t$  such that  $v_s$  is a pseudosource,  $v_p$  is a passage, and  $v_t$  is a pseudosink, we obtain the polarized forward path graph  $\tilde{G}^\circ = (\tilde{V}^\circ, \tilde{A}^\circ)$  and finally the truncated forward path graph  $\tilde{G}_{\text{tr}}^\circ = (\tilde{V}_{\text{tr}}^\circ, \tilde{A}_{\text{tr}}^\circ)$  by removing all pseudosinks in  $\tilde{G}^\circ$ .

As has already been noted in Section 5.1.3, all backward arcs as well as the forward paths chosen for  $\tilde{G}$  are preserved in  $\tilde{G}^\circ$ . Furthermore, every vertex  $v \in \tilde{V}^\circ$  with an incident backward arc is either a pseudosource or a pseudosink. As  $H$  is source-preserving, every pseudosource  $v_s$  in  $\tilde{G}^\circ$  is a source in  $H$  such that  $d_H^+(v_s) = b_{\tilde{\pi}^\circ}^-(v_s)$  and every source in  $H$  originates from a pseudosource in  $\tilde{G}^\circ$ . Hence, for each source  $v_s \in V_H$  that originates from the polarization of the vertex  $v \in \tilde{V}^\circ$ ,  $\delta_H(v_s) = b_{\tilde{\pi}^\circ}^-(v_s) = b_{\tilde{\pi}^\circ}^-(v)$ . Observe that, as in every graph,  $\sum_{x \in V_H} \delta_H(x) = 0$ . Let  $S \subseteq V_H$  be the set of all sources in  $H$  and let  $v_s$  be the respective pseudosource component of a vertex  $v \in \tilde{V}^\circ$ . As  $S \subseteq \tilde{V}^\circ$  also equals exactly the set of pseudosources in  $\tilde{G}^\circ$ , we obtain

$$\sum_{v_s \in S} \delta_H(v_s) = \sum_{v_s \in S} b_{\tilde{\pi}^\circ}^-(v_s) = \sum_{v \in \tilde{V}^\circ} b_{\tilde{\pi}^\circ}^-(v) = \sum_{v \in V} b_{\tilde{\pi}^\circ}^-(v) = |\pi|.$$

Consequently, the delta degrees of all non-sources in  $H$  must sum up to the negative of this value, i. e.,

$$\sum_{x \in V_H \setminus S} \delta_H(x) = -|\pi|.$$

Equivalently,

$$\sum_{x \in V_H \setminus S} -\delta_H(x) = |\pi|,$$

which yields

$$\sum_{x \in V_H \setminus S} \max\{-\delta_H(x), 0\} \geq |\pi|. \quad (5.5)$$

Consider now the mappings  $\alpha_h, \alpha_t,$  and  $\alpha_d$  as defined in the statement of the lemma, but initially for the vertices of  $\tilde{G}^\circ$  instead of  $G$ , as follows: For a vertex  $x \in \tilde{G}^\circ$ , we set  $\alpha_h(x) = b_{\tilde{\pi}^\circ}^-(x)$  and  $\alpha_t(x) = b_{\tilde{\pi}^\circ}^+(x)$ . Furthermore, if  $x$  is a passage, we derive the value of  $\alpha_d(x)$  from its negative delta degree in  $H$  and clip it at zero, i. e.,  $\alpha_d(x) =$

$\max\{-\delta_H(x), 0\}$ . Keep in mind that every vertex of  $\tilde{G}^\circ$  is either a pseudosource or a pseudosink or a passage and that  $V_H$  contains exactly the pseudosources and the passages. To make  $\alpha_h, \alpha_t$ , and  $\alpha_d$  total, we set  $\alpha_h(x) = 0$  for pseudosinks and passages,  $\alpha_t(x) = 0$  for pseudosources and passages, and  $\alpha_d(x) = 0$  for pseudosources and pseudosinks. Let  $P$  denote the set of passages in  $\tilde{G}^\circ$ . Equation (5.5) then yields that

$$\sum_{x \in \tilde{V}^\circ} \alpha_d(x) = \sum_{x \in P} \max\{-\delta_H(x), 0\} = \sum_{x \in V_H \setminus S} \max\{-\delta_H(x), 0\} \geq |\pi|, \quad (5.6)$$

because for every pseudosource or pseudosink  $x$ ,  $\alpha_d(x) = 0$  by the above definition. Furthermore,  $V_H$  consists exactly of the pseudosources and passages of  $\tilde{G}^\circ$  and every pseudosource in  $\tilde{G}^\circ$  corresponds to a source in  $S \subseteq V_H$ .

The mappings  $\alpha_h, \alpha_t$ , and  $\alpha_d$  for the vertices of  $G$  are now obtained from their corresponding component vertices in  $\tilde{G}^\circ$ , i. e., for  $v \in V_G$ ,  $\alpha_h(v) = \alpha_h(v_s)$ ,  $\alpha_t(v) = \alpha_t(v_t)$ , and  $\alpha_d(v) = \alpha_d(v_p)$ . If a component does not exist, we set the respective value to zero. In case that  $v$  is pooled within a vertex  $u$  in  $\tilde{G}^\circ$  along with  $k - 1$  further vertices, we have  $b_{\tilde{\pi}^\circ}^-(u) = \sum_{v \in \tilde{V}(u)} b_{\tilde{\pi}^\circ}^-(v) = \sum_{v \in \tilde{V}(u)} b_{\tilde{\pi}^\circ}^-(v)$  and  $b_{\tilde{\pi}^\circ}^+(u) = \sum_{v \in \tilde{V}(u)} b_{\tilde{\pi}^\circ}^+(v) = \sum_{v \in \tilde{V}(u)} b_{\tilde{\pi}^\circ}^+(v)$ , where  $\tilde{V}(u)$  is used here to denote the set of vertices that  $u$  pools. Note that  $\tilde{V}(u) \subseteq V$ . We set  $\alpha_h(v) = b_{\tilde{\pi}^\circ}^-(v)$ ,  $\alpha_t(v) = b_{\tilde{\pi}^\circ}^+(v)$ , and  $\alpha_d(v) = \frac{1}{k} \cdot \alpha_d(u_p)$ , such that the componentwise sum of all vertices pooled within  $u$  yields exactly  $\alpha_h(u)$ ,  $\alpha_t(u)$ , and  $\alpha_d(u)$ . Recall from Section 5.1.3 that for a right-blocking vertex  $v$ ,  $v_s = v_r$ ,  $v_t = v_l$ , and there is no passage component  $v_p$ .

Hence, this definition yields for every vertex  $v \in V_G$  that










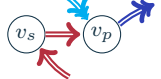




$$\begin{aligned} \alpha_h(v) &= \alpha_h(v_s) = b_{\tilde{\pi}^\circ}^-(v_s) = b_{\tilde{\pi}^\circ}^-(v) = b_{\tilde{\pi}^\circ}^-(v) = b_{\tilde{\pi}^\circ}^-(v) \quad \text{and} \\ \alpha_t(v) &= \alpha_t(v_t) = b_{\tilde{\pi}^\circ}^+(v_t) = b_{\tilde{\pi}^\circ}^+(v) = b_{\tilde{\pi}^\circ}^+(v) = b_{\tilde{\pi}^\circ}^+(v) \end{aligned}$$

and thereby corresponds to the statement in the lemma. Furthermore,


$$\begin{aligned} \sum_{v \in V_G} \alpha_h(v) &= \sum_{v \in V_G} b_{\tilde{\pi}^\circ}^-(v) = |\pi|, \\ \sum_{v \in V_G} \alpha_t(v) &= \sum_{v \in V_G} b_{\tilde{\pi}^\circ}^+(v) = |\pi|, \text{ and, due to Equation (5.6),} \\ \sum_{v \in V_G} \alpha_d(v) &= \sum_{v \in V_G} \alpha_d(v_p) = \sum_{x \in \tilde{V}^\circ} \alpha_d(x) \geq |\pi|, \end{aligned}$$


which shows that the assignment scheme is admissible and thereby concludes the proof.  $\square$

Table 5.2: Definition of the mappings  $\alpha_h$ ,  $\alpha_t$ , and  $\alpha_d$  according to Lemma 5.3.

$\mathcal{L}_\pi(v)$	$\mathcal{L}_{\tilde{\pi}^\emptyset}(v_s) / \mathcal{L}_{\tilde{\pi}^\emptyset}(v_p) / \mathcal{L}_{\tilde{\pi}^\emptyset}(v_t)^a$	$\alpha_h(v)$	$\alpha_t(v)$	$\alpha_d(v)$
		$b_\pi^-(v)$	0	0
		0	$b_\pi^+(v)$	0
		$b_\pi^-(v)$	$b_\pi^+(v)$	$\max\{-\delta_H(v_p), 0\}$
		$b_\pi^-(v)$	$b_\pi^+(v)$	0
		$b_\pi^-(v)$	0	$\max\{-\delta_H(v_p), 0\}^b$
		0	$b_\pi^+(v)$	$\max\{-\delta_H(v_p), 0\}^c$
		0	0	$\max\{-\delta_H(v_p), 0\}$

<sup>a</sup>As  $\tilde{G} \subseteq G$  and  $\tilde{G}^\emptyset$  is obtained in turn from  $\tilde{G}$ ,  $f_{\tilde{\pi}^\emptyset}^-(v_p) \leq f_\pi^-(v)$ ,  $f_{\tilde{\pi}^\emptyset}^+(v_p) \leq f_\pi^+(v)$ , and, if  $v_p$  does not exist,  $1 \leq f_{\tilde{\pi}^\emptyset}^-(v_t) \leq f_\pi^-(v)$  in general. For the sake of readability, the pictograms do not reflect this fact if compared to the leftmost column.

<sup>b</sup>If  $v$  has layout  in  $\tilde{\pi}$  and is pooled within a vertex  $u$  along with  $k - 1$  further vertices, then the second column shows the components  $u_s$  and  $u_p$  of  $u$  and  $\alpha_d(v) = \frac{1}{k} \max\{-\delta_H(u_p), 0\}$ .

<sup>c</sup>If  $v$  has layout  in  $\tilde{\pi}$  and is pooled within a vertex  $u$  along with  $k - 1$  further vertices, then the second column shows the components  $u_p$  and  $u_t$  of  $u$  and  $\alpha_d(v) = \frac{1}{k} \max\{-\delta_H(u_p), 0\}$ .



The assignment scheme presented in [Lemma 5.3](#) is once again subsumed in [Table 5.2](#) for the different vertex layouts. Here, the first column shows the layout of a vertex  $v$  as induced by the linear ordering  $\pi$  that is to be analyzed. In the second column, the components of  $v$  or, if  $v$  is pooled within a vertex  $u$  in  $\tilde{G}^\circ$ , those of  $u$  are depicted. The last three columns specify  $v$ 's assignments. The table also shows the implications in case that  $v$  is right-blocking in the fourth row.

## 5.4 Subquartic and Subquintic Graphs

We now apply the generalized assignment scheme developed in the previous section to subquartic graphs, i. e., graphs where every vertex has a degree of at most four. Prior to this, we introduce two properties that basically constitute a manifestation of the Arc Stability Property, but are especially tailored to graphs of this density. The analysis of  $\Psi$ -optimal linear orderings that additionally adhere to them allows us then to derive a tight bound. Finally, we will also make a suggestion how this proof can be extended to also accommodate for graphs with vertex degree at most five, i. e., subquintic graphs.

### 5.4.1 Two Special Cases of One-Arc Stability

In preparation for our estimate of the upper bound of an optimal feedback arc set of a subquartic graph, we introduce two properties of optimal linear orderings. Interestingly, their generalization yields the Extended Multipath Blocking Vertices Property, which has already shown not to be establishable efficiently, whereas the following weaker versions only require a special implementation of the One-Arc Stability Property.

---

**Lemma 5.4** ALTERNATIVE FORWARD PATHS PROPERTY  
 Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V, A)$ . Let  $u, v \in V$  such that  $u$  has exactly one incoming backward arc  $b_u$  and  $v$  has exactly one outgoing backward arc  $b_v$  with respect to  $\pi^*$  and let  $(u, v) \in A$  be a forward arc in  $\pi^*$ . Then, at least one of  $b_u$  and  $b_v$  has a forward path that contains neither  $(u, v)$  nor a left-blocking (right-blocking) vertex.

---

*Proof.* Let  $b_u = (t, u)$  and  $b_v = (v, h)$  denote the incoming backward arc of  $u$  and the outgoing backward arc of  $v$ , respectively, and set  $a = (u, v)$ . By [Lemma 4.14](#),  $\pi^*$  respects the Multipath Blocking Vertices Property. Consider first the left-blocking case and

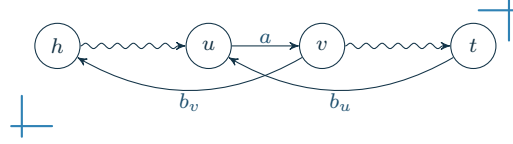


Figure 5.13: Initial situation in Lemma 5.4.

suppose that in  $\pi_{\text{sp}/l}^*$ , all forward paths for  $b_u$  contain  $a$  as first arc and all forward paths for  $b_v$  contain  $a$  as last arc, as is also depicted schematically in Figure 5.13.

Identify a set  $Z$  of left-blocking vertices by traversing all cropped forward paths of  $b_u$  and  $b_v$  in  $\pi^*$  and placing the first left-blocking vertex that is encountered thereby in  $Z$ . Then, every forward path for  $b_u$  or  $b_v$  in  $\pi^*$  uses either  $a$  or contains a vertex in  $Z$ . Moreover,  $u, v \notin Z$  because with all forward paths for  $b_u$  and  $b_v$  in  $\pi_{\text{sp}/l}^*$  passing through  $a$ , neither  $u$  nor  $v$  can be left-blocking. Consider  $h$  and  $t$ . As all forward paths for  $b_u$  have  $a$  as their first arc and all forward paths for  $b_v$  have  $a$  as their last arc,  $\pi^*(h) < \pi^*(u) < \pi^*(v) < \pi^*(t)$ . Hence,  $h$  cannot be part of a forward path of  $b_u$  and  $t$  cannot be part of a forward path for  $b_v$ . Furthermore, we only considered cropped forward paths for the creation of  $Z$  and all forward paths for  $b_v$  start at  $h$ , whereas all forward paths for  $b_u$  end at  $t$ . Thus,  $h, t \notin Z$  either.

Let  $\mathcal{B}$  and  $\mathcal{F}$  denote the set of backward and forward arcs induced by  $\pi^*$ ,  $Y = \{b_v\} \subseteq \mathcal{B}^+(v)$ , and  $X = \{a\}$ . Observe that  $v$  is on a forward path for  $b_u$  with respect to  $\pi_{\text{sp}/l}^*$ . By Lemma 4.16,

$$\begin{aligned} \mathcal{B}' &= \mathcal{B} \setminus \left( \mathcal{B}^+[Z] \cup Y \cup [b_u]_{\parallel} \right) \cup \mathcal{F}^-[Z] \cup X \\ &= \mathcal{B} \setminus \left( \mathcal{B}^+[Z] \cup \{b_u, b_v\} \right) \cup \mathcal{F}^-[Z] \cup \{a\} \end{aligned}$$

is feasible. Furthermore,  $\mathcal{B}^+[Z] = \mathcal{F}^-[Z]$  by Proposition 4.2, and  $\mathcal{B}^+[Z] \cap \{b_u, b_v\} = \emptyset$  due to  $v, t \notin Z$ . Subsequently,  $|\mathcal{B}'| < |\mathcal{B}|$ , a contradiction to the optimality of  $\pi^*$ .

For the right-blocking case, the statement follows by an analogous argument. Alternatively, we can instead consider the reverse graph  $G^R$  along with the reverse linear ordering  $\pi^{*R}$ . Then,  $u$  and  $v$  swap roles and every right-blocking vertex in  $\pi^*$  is left-blocking with respect to  $\pi^{*R}$  and vice versa.  $\square$

The second property has a similar setup:

**Lemma 5.5**

## TAIL ON FORWARD PATH PROPERTY

Let  $\pi^*$  be an optimal linear ordering of a graph  $G = (V, A)$ . Let  $u, u', v \in V$  such that  $u$  and  $u'$  are not left-blocking (right-blocking) and have exactly one incoming backward arc  $b_u$  and  $b_{u'}$ , respectively,  $v$  has a single outgoing forward arc, and  $(u, v), (u', v) \in A$  are forward arcs with respect to  $\pi^*$ . If a forward path for  $b_u$  in  $\pi_{\text{sp}/l}^*$  ( $\pi_{\text{sp}/r}^*$ ) contains the tail of  $b_{u'}$  or vice versa, then at least one of  $b_u$  and  $b_{u'}$  has a forward path in  $\pi_{\text{sp}/l}^*$  ( $\pi_{\text{sp}/r}^*$ ) that does not contain  $v$ .

*Proof.* Let  $\mathcal{B}$  and  $\mathcal{F}$  denote the set of backward and forward arcs induced by  $\pi^*$  and let  $b_u = (w, u)$  and  $b_{u'} = (w', u')$ . By Lemma 4.14,  $\pi^*$  respects the Multipath Blocking Vertices Property. If  $w = w'$ , then the statement immediately follows in consequence thereof. Hence, we only consider the case that  $w \neq w'$ .

W.l.o.g., assume that  $w'$  is part of a forward path  $P_{b_u}$  for  $b_u$  in  $\pi_{\text{sp}/l}^*$  or in  $\pi_{\text{sp}/r}^*$ , respectively. Consequently,  $\pi^*(u) < \pi^*(w') < \pi^*(w)$ . Let  $a$  be  $v$ 's only outgoing forward arc, i. e.,  $\mathcal{F}^+(v) = \{a\}$ . Note that if a forward path for  $b_u$  or  $b_{u'}$  contains  $v$ , then it must also contain  $a$ . Set  $Y = \{b_{u'}\} \subseteq \mathcal{B}^+(w')$  and  $X = \{a\}$ .

Suppose that in  $\pi_{\text{sp}/l}^*$  all forward paths for  $b_u$  and  $b_{u'}$  contain  $v$ . This situation is also depicted in Figure 5.14. Construct a set of left-blocking vertices  $Z$  by traversing every cropped forward path for  $b_u$  and  $b_{u'}$  in  $\pi^*$  and collecting the first left-blocking vertex that occurs during the traversal in  $Z$ . Due to the precondition in the statement of the lemma, neither  $u$  nor  $u'$  are left-blocking. Hence  $u, u' \notin Z$ . The same applies for  $w$  and  $w'$ : Based on our assumption,  $w'$  is part of a forward path for  $b_u$  in  $\pi_{\text{sp}/l}^*$ . Hence,  $w'$  cannot be left-blocking, so  $w' \notin Z$ . As  $\pi^*(w') < \pi^*(w)$ ,  $w$  cannot be part of a forward path for  $b_{u'}$  and, due to the consideration of cropped forward paths only,  $w \notin Z$ .

Every forward path for  $b_u$  or  $b_{u'}$  in  $\pi^*$  then contains either  $a$  or a vertex of  $Z$ . Due to Lemma 4.16,

$$\begin{aligned} \mathcal{B}' &= \mathcal{B} \setminus \left( \mathcal{B}^+[Z] \cup Y \cup [b_u]_{\parallel} \right) \cup \mathcal{F}^-[Z] \cup X \\ &= \mathcal{B} \setminus \left( \mathcal{B}^+[Z] \cup \{b_{u'}, b_u\} \right) \cup \mathcal{F}^-[Z] \cup \{a\} \end{aligned}$$

is feasible. By Proposition 4.2,  $\mathcal{B}^+[Z] = \mathcal{F}^-[Z]$ . Moreover, the fact that  $w, w' \notin Z$  yields that  $\mathcal{B}^+[Z] \cap \{b_{u'}, b_u\} = \emptyset$ , which in turn implies that  $|\mathcal{B}'| < |\mathcal{B}|$ , a contradiction to the optimality of  $\pi^*$ .

The proof for the right-blocking case is similar. To this end, suppose that all forward paths for  $b_u$  and  $b_{u'}$  in  $\pi_{\text{sp}/r}^*$  contain  $v$ . Let  $Z$  be a set of right-blocking vertices that is

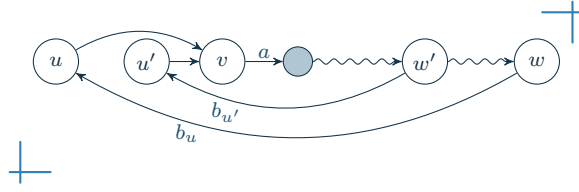


Figure 5.14: Initial situation in Lemma 5.5.

obtained by traversing every cropped forward path for  $b_u$  and  $b_{u'}$  in  $\pi^*$  and putting the first right-blocking vertex that occurs during the traversal in  $Z$ . For the same reasons as in the left-blocking case,  $u, u', w, w' \notin Z$ . Lemma 4.16 here yields that

$$\begin{aligned} \mathcal{B}' &= \mathcal{B} \setminus \left( \mathcal{B}^-[Z] \cup Y \cup [b_u]_{\parallel} \right) \cup \mathcal{F}^+[Z] \cup X \\ &= \mathcal{B} \setminus \left( \mathcal{B}^-[Z] \cup \{b_{u'}, b_u\} \right) \cup \mathcal{F}^+[Z] \cup \{a\} \end{aligned}$$

is feasible. Again by Proposition 4.2,  $\mathcal{B}^-[Z] = \mathcal{F}^+[Z]$  and  $\mathcal{B}^-[Z] \cap \{b_{u'}, b_u\} = \emptyset$  because  $u, u' \notin Z$ . Thus,  $|\mathcal{B}'| < |\mathcal{B}|$ , a contradiction to the optimality of  $\pi^*$ .  $\square$

In contrast to the Extended Multipath Blocking Vertices Property, the properties shown in Lemma 5.4 and Lemma 5.5 can be enforced efficiently in both cases by a variant of the One-Arc Stability Property:

---



---

#### Lemma 5.6

There is a  $\mathcal{O}(m^2)$ -time algorithm that tests whether a given linear ordering  $\pi$  respects the Alternative Forward Paths Property and, if negative, applies an improvement.

---

*Proof.* First, we obtain the set of forward and backward arcs induced by  $\pi$  by calling the initialization routine *ComputePositionsAndArcSets*, which was introduced in Section 4.2.4 and runs in  $\mathcal{O}(m)$  time by Lemma 4.1. Next, consider each forward arc  $a = (u, v)$  and check whether  $u$  has exactly one incoming backward arc and  $v$  has exactly one outgoing backward arc. This can be accomplished in constant time. If positive, remove  $a$  and test for both  $\pi_{\text{sp}/l}$  and  $\pi_{\text{sp}/r}$  if at least one of the backward arcs incident to  $u$  and  $v$  has a forward path. The latter requires at most two traversals of the graph per backward arc and can thus be accomplished in  $\mathcal{O}(m)$ . If both backward arcs have no forward path, we compute a set  $Z$  of left-blocking or right-blocking vertices, respectively,

apply the modification given in the proof of [Lemma 5.4](#) to the set of backward arcs, and obtain an improved linear ordering by topologically sorting the new acyclic subgraph of  $G$  in time  $\mathcal{O}(m)$ . Hence, the running time is in  $\mathcal{O}(m^2 + m) = \mathcal{O}(m^2)$ .  $\square$

For the Tail On Forward Path Property, we obtain:

---



---

**Lemma 5.7**

There is a  $\mathcal{O}(m^2)$ -time algorithm that tests whether a given linear ordering  $\pi$  respects the Tail On Forward Path Property and, if negative, applies an improvement.











---

*Proof.* We start again with the computation of the set of forward and backward arcs induced by  $\pi$  by calling the initialization routine *ComputePositionsAndArcSets*, which was introduced in [Section 4.2.4](#) and runs in  $\mathcal{O}(m)$  time by [Lemma 4.1](#). Then, the algorithm loops over all vertices  $v \in V$  and checks whether  $v$  has a single outgoing forward arc  $a$  and whether there is a set of vertices  $U$  with  $|U| \geq 2$  such that each vertex  $u \in U$  has an outgoing forward arc to  $v$ , is not left-blocking or right-blocking, respectively, and has exactly one incoming backward arc  $b_u$ . In case of success, we remove  $a$  and test for every vertex  $u \in U$  whether its incident backward arc  $b_u$  has a forward path in  $\pi_{\text{sp}/l}$  ( $\pi_{\text{sp}/r}$ ). The algorithm creates a set  $U' \subseteq U$  that contains all vertices that failed this test. The time needed to construct  $U$  and  $U'$  is in  $\mathcal{O}(f^-(v) \cdot m) \subseteq \mathcal{O}(d^-(v) \cdot m)$ . For every vertex  $u \in U'$ , we have to see whether there is a vertex  $u' \in U'$  such that  $u$  is on a forward path for  $u'$ . More precisely, it suffices to check whether there is a forward path from the tail of  $b_{u'}$  to the tail of  $b_u$ , because they all have a forward path via  $a$ . This can be accomplished by traversing the forward path graph with respect to  $\pi_{\text{sp}/l}$  ( $\pi_{\text{sp}/r}$ ) starting at the tails and requires hence  $\mathcal{O}(d^-(v) \cdot m)$  time. If there are two such vertices  $u \neq u' \in U'$ , we obtain a set  $Z$  of left-blocking or right-blocking vertices, respectively, apply the modification given in the proof of [Lemma 5.5](#) to the set of backward arcs, and obtain an improved linear ordering by topologically sorting the new acyclic subgraph of  $G$  in time  $\mathcal{O}(m)$ . In sum, we hence obtain a running time of  $\mathcal{O}(\sum_{v \in V} (2 \cdot d^-(v) \cdot m) + m) \subseteq \mathcal{O}(m^2)$ .  $\square$

### 5.4.2 A Tight Bound for Subquartic Graphs

With these two new properties at hand, we are now ready to apply the generalized assignment scheme to subquartic graphs. Due to the Nesting Property, a vertex of degree four can have up to two incident backward arcs. For such a vertex  $v$ , we then

**Table 5.3:** Layouts induced by a  $\Psi$ -optimal linear ordering on a vertex  $v$  with  $d(v) = 4$  as pictograms and 4-tuples.

$b(v) = 2:$		$(0, 2, 2, 0),$		$(2, 0, 0, 2),$		$(1, 1, 1, 1)$		
$b(v) = 1:$		$(0, 3, 1, 0),$		$(3, 0, 0, 1),$		$(1, 2, 1, 0),$		$(2, 1, 0, 1)$
$b(v) = 0:$		$(1, 3, 0, 0),$		$(2, 2, 0, 0),$		$(3, 1, 0, 0)$		

already have  $\alpha_h(v) + \alpha_t(v) = 2$ . By showing that for every vertex  $v$  in a subquartic graph,  $\alpha(v) \leq 2$  on average, we obtain:

---





---

**Lemma 5.8**

Let  $\pi$  be a  $\Psi$ -optimal linear ordering of subquartic graph  $G$  such that  $\pi$  additionally respects the Alternative Forward Paths Property and the Tail On Forward Path Property. Then, there is an admissible assignment scheme  $\alpha$  with respect to  $\pi$  such that for every vertex  $v$  of  $G$ ,  $\alpha(v) \leq 2$  on average.

---

*Proof.* Let  $G = (V_G, A_G)$  be a subquartic graph with a  $\Psi$ -optimal linear ordering  $\pi$  that additionally respects the Alternative Forward Paths Property and the Tail On Forward Path Property. Then,  $\pi$  also respects the Nesting Property and the Eliminateable Layouts Property and hence induces one of the seventeen different layouts shown in Table 5.1 and Table 5.3 on every vertex of  $G$ . In particular, every vertex is incident to at most two backward arcs. Let  $n = |V_G|$  and  $m = |A_G|$  denote, as usual, the number of vertices and arcs of  $G$ . The graph depicted in Figure 5.15 will serve as a running example.

In analogy to the proof for subcubic graphs, we obtain a forward path graph  $\tilde{G} = (\tilde{V}, \tilde{A})$  of  $G$  along with the linear ordering  $\tilde{\pi}$  as a first step. Recall that  $\tilde{G} \subseteq G_{sp/r}$  and, by construction,  $\tilde{G}$  contains two pairwise arc-disjoint forward paths for every pair of backward arcs that have a common head. For the forward paths of a pair of backward arcs with a common tail, arc-disjointness is not guaranteed though. Hence, it may be the case that a vertex with layout  in  $\pi_{sp/r}$  has just one incoming forward arc in  $\tilde{\pi}$ , i. e., its layout is . Consider a forward path  $P = u \rightsquigarrow v$  such that every vertex besides  $u$  has an outgoing backward arc and  $v$  is a pseudosink in  $\pi$ . We call such a path a *pseudosink path*. In case that there are multiple ways to select a forward path for a backward arc  $b$  and  $b$ 's head has just one incoming backward arc, we choose a forward path for  $b$  that is not a pseudosink path wherever possible. As  $G$  is subquartic, this affects in particular

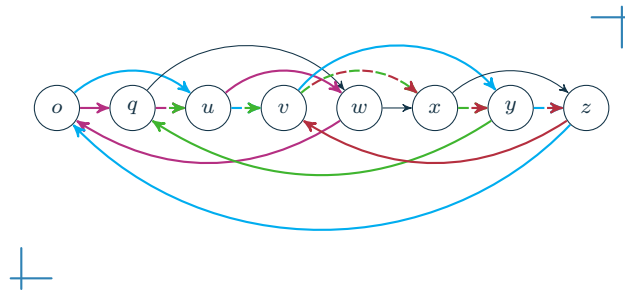


Figure 5.15: A  $\Psi$ -optimal linear ordering that respects the Alternative Forward Paths Property and the Tail On Forward Path Property of a quartic graph. For each backward arc, one possible forward path is highlighted, with forward paths of backward arcs with a common head being arc-disjoint.

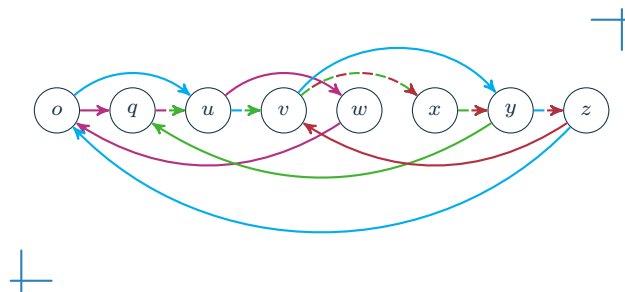


Figure 5.16: The forward path graph of the graph in Figure 5.15 corresponding to the selection of the forward paths, which at the same time also is its pooled forward path graph.

backward arcs whose heads have layout  $\rightarrow\alpha\leftarrow$ . This forward path selection policy has already been paid regard to in Figure 5.15. Figure 5.16 shows the corresponding forward path graph  $\tilde{G}$ .

We now obtain the pooled forward path graph  $\tilde{G}^\circ = (\tilde{V}^\circ, \tilde{A}^\circ)$  by contracting arcs as described in Section 5.1.2. As  $G$  is subquartic, this only concerns vertices with layout  $\rightarrow\alpha\leftarrow$  and  $\rightarrow\alpha\leftarrow$  as well as vertices with layout  $\rightarrow\circ\rightarrow$  and  $\leftarrow\circ\rightarrow$  in  $\tilde{\pi}$ . Along with  $\tilde{G}^\circ$ , we also obtain the corresponding linear ordering  $\tilde{\pi}^\circ$ . In case of the example graph, every for-out-tree and every for-in-tree consists of a single vertex only. The pooled forward path graph therefore equals the forward path graph. Let  $u \in \tilde{V}^\circ$  be a vertex with layout

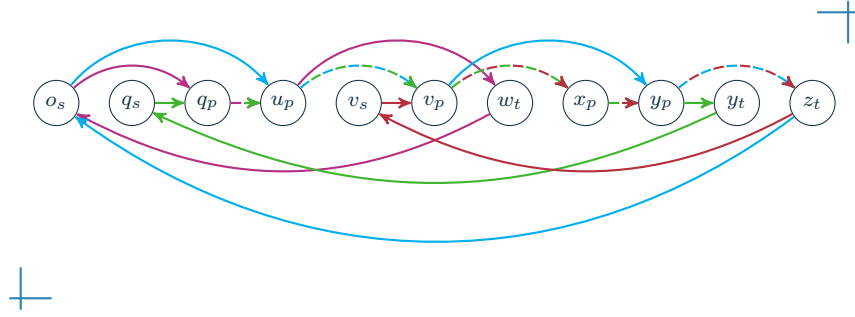


Figure 5.17: The polarized forward path graph obtained from the pooled forward path graph in Figure 5.16.

$\rightarrow \circ \leftarrow$  in  $\tilde{\pi}^\circ$  such that  $u$  pools  $k$  vertices, i. e., it is the result of  $k - 1$  arc contractions. We can specify the cardinalities given in Equation (5.1) more precisely here and obtain

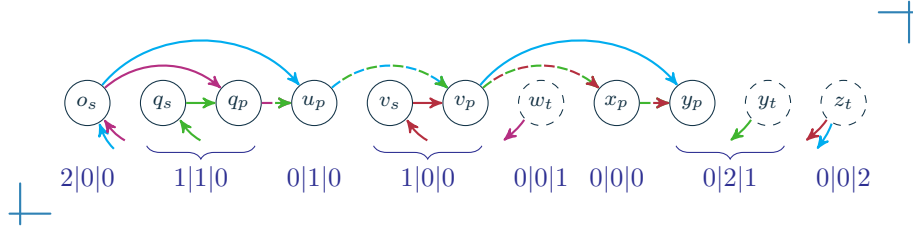
$$\begin{aligned} f_{\tilde{\pi}^\circ}^-(u) &= 1, \\ 1 &\leq f_{\tilde{\pi}^\circ}^+(u) \leq k + 1, \\ b_{\tilde{\pi}^\circ}^-(u) &= k, \\ b_{\tilde{\pi}^\circ}^+(u) &= 0. \end{aligned} \tag{5.7}$$

Similarly, if  $u$  has layout  $\leftarrow \circ \rightarrow$  in  $\tilde{\pi}^\circ$  and pools  $k$  vertices, we can derive from Equation (5.2) that

$$\begin{aligned} 1 &\leq f_{\tilde{\pi}^\circ}^-(u) \leq k + 1, \\ f_{\tilde{\pi}^\circ}^+(u) &= 1, \\ b_{\tilde{\pi}^\circ}^-(u) &= 0, \\ b_{\tilde{\pi}^\circ}^+(u) &= k. \end{aligned} \tag{5.8}$$

In the next step, we consider the polarized forward path graph  $\tilde{G}^\circ = (\tilde{V}^\circ, \tilde{A}^\circ)$  and the corresponding linear ordering  $\tilde{\pi}^\circ$ , which are derived from  $\tilde{G}^\circ$  and  $\tilde{\pi}^\circ$  as shown in Section 5.1.3. This auxiliary graph again only differs from the previous one in case of vertices with layout  $\rightarrow \circ \leftarrow$  and  $\leftarrow \circ \rightarrow$ , because all other vertices already are either pseudosources, pseudosinks, or passages. Recall that a vertex  $u$  with layout  $\rightarrow \circ \leftarrow$  in  $\tilde{\pi}^\circ$  is split into two vertices  $u_s$  and  $u_p$  with combined layout  $\circ \leftarrow \rightarrow$  and that a vertex  $u$  with layout  $\leftarrow \circ \rightarrow$  in  $\tilde{\pi}^\circ$  is split into two vertices  $u_p$  and  $u_t$  with combined layout  $\leftarrow \circ \rightarrow$  in  $\tilde{\pi}^\circ$ . Hence, if  $u$  has layout  $\rightarrow \circ \leftarrow$  in  $\tilde{\pi}^\circ$  and pools  $k$  vertices, then  $u_s$  has  $k$  incoming backward arcs and there are  $k$  parallel arcs  $(u_s, u_p)$ . Likewise,  $u_t$  has  $k$  outgoing backward arcs and there are  $k$  parallel arcs  $(u_p, u_t)$  in case that  $v$  has layout  $\leftarrow \circ \rightarrow$  in  $\tilde{\pi}^\circ$  and pools  $k$  vertices.





**Figure 5.18:** The truncated forward path graph obtained from the polarized forward path graph in Figure 5.17 along with the assignments of each vertex according to Equation (5.9). The assignments of each vertex  $v$  of the original graph are represented as  $\alpha_h(v)|\alpha_d(v)|\alpha_t(v)$ . Vertices with a dashed border, i. e.,  $w_t$ ,  $y_t$ , and  $z_t$ , are not contained in the truncated forward path graph and only shown here to be able to conveniently display their assignments.

Figure 5.17 depicts the polarized forward path graph for the example graph. As we already noticed in Section 5.1.3, all backward arcs as well as all forward paths selected during the construction of  $\tilde{G}$  have been preserved during the transformation of  $\tilde{G}$  to  $\tilde{G}^\circ$  and  $\tilde{G}^\circ$ . Furthermore, every vertex  $v \in \tilde{V}^\circ$  with an incident backward arc is either a pseudosource or a pseudosink.

From  $\tilde{G}^\circ$  we obtain the truncated forward path graph  $\tilde{G}_{tr}^\circ = (\tilde{V}_{tr}^\circ, \tilde{A}_{tr}^\circ)$  by removing all pseudosinks as described in Section 5.1.4. As  $\tilde{G}_{tr}^\circ$  is source-preserving, the following definition of  $\alpha_h$ ,  $\alpha_t$ , and  $\alpha_d$  yields an admissible assignment scheme by Lemma 5.3:

$$\begin{aligned} \alpha_h(v) &= b_\pi^-(v), \\ \alpha_t(v) &= b_\pi^+(v), \text{ and} \\ \alpha_d(v) &= \begin{cases} \max\{-\delta_{\tilde{G}_{tr}^\circ}(v_p), 0\}, & \text{if } v_p \in \tilde{V}_{tr}^\circ \text{ is } v\text{'s passage component,} \\ \frac{1}{k} \cdot \max\{-\delta_{\tilde{G}_{tr}^\circ}(u_p), 0\}, & \text{if } v \text{ and } k-1 \text{ other vertices are pooled} \\ & \text{within a vertex } u \text{ and } u_p \in \tilde{V}_{tr}^\circ \text{ is} \\ & \text{ } u\text{'s passage component,} \\ 0, & \text{otherwise.} \end{cases} \quad (5.9) \end{aligned}$$

Figure 5.18 shows the truncated forward path graph for the running example. For each vertex  $v \in V_G$ , the assignments are additionally given using the format  $\alpha_h(v)|\alpha_d(v)|\alpha_t(v)$ .

We now show that for every vertex  $v \in V_G$ ,  $\alpha(v) = \alpha_h(v) + \alpha_t(v) + \alpha_d(v) \leq 2$  on average, with a single exception, which we will address in the last part of the proof. Depending on  $v$ 's layout within  $\pi$ , we distinguish the following cases:

*v* is a pseudosource or a pseudosink. If *v* is a pseudosource or a pseudosink in  $\pi$ , then we immediately obtain from the definition that  $\alpha(v) = \alpha_h(v) = b_{\pi}^-(v) \leq 2$  in the former and  $\alpha(v) = \alpha_t(v) = b_{\pi}^+(v) \leq 2$  in the latter case.

*v* has both an incoming and an outgoing backward arc. If *v* has layout  $\curvearrowright \circ \curvearrowleft$  in  $\pi$ , then it is right-blocking. Subsequently,  $\alpha_h(v) = b_{\pi}^-(v) = 1$ ,  $\alpha_t(v) = b_{\pi}^+(v) = 1$ , and hence,  $\alpha(v) = \alpha_h(v) + \alpha_t(v) = 2$ .

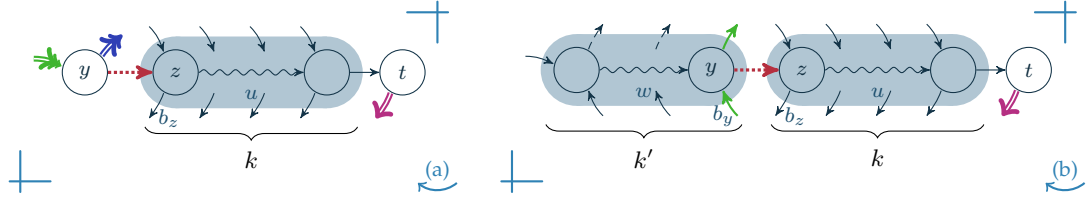
*v* has no incident backward arcs. In this case, *v* is neither pooled nor polarized and thus has only a (possibly isolated) passage component  $v_p$  in  $\tilde{G}_{\text{tr}}^{\emptyset}$ . By the definition in the previous step,  $\alpha(v) = \alpha_d(v) = \max\{-\delta_{\tilde{G}_{\text{tr}}^{\emptyset}}(v_p), 0\}$ . As *G* is subquartic,  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}(v_p) \leq d_G(v) \leq 4$ . Hence,  $1 \leq d_G^+(v) \leq 3$  and  $1 \leq d_G^-(v) \leq 3$ . Regarding  $\tilde{G}_{\text{tr}}^{\emptyset}$ , however, we only have that  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^+(v_p) \leq d_G^+(v)$  and  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^-(v_p) \leq d_G^-(v)$ . In particular, *v* may be an isolated vertex if no forward path selected during the construction of  $\tilde{G}$  contains *v*. Otherwise, *v* has at least one incoming and one outgoing arc in  $\tilde{G}$ . In consequence of the removal of pseudosinks,  $v_p$  may be a sink in  $\tilde{G}_{\text{tr}}^{\emptyset}$ , but not a source. Thus, we either have that  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^+(v_p) = d_{\tilde{G}_{\text{tr}}^{\emptyset}}^-(v_p) = 0$  or  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^-(v_p) \geq 1$ . Assume that  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^+(v_p) = 1$  and  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^-(v_p) = 0$ . Then, *v* has an outgoing forward arc whose head *h* is a pseudosink in  $\tilde{G}$ , which is removed during the construction of  $\tilde{G}_{\text{tr}}^{\emptyset}$ . Furthermore, *v* must be contained in one or more forward paths of *h*'s outgoing backward arcs. Observe that *h* is the tail of these backward arcs and that the forward paths for  $\tilde{G}$  are selected such that they are pairwise arc-disjoint at the backward arcs' heads, but not necessarily at their tails. As *G* is subquartic,  $b_{\pi}^+(h) \leq 2$ , so *v* can be part of at most two forward paths, which implies that  $f_{\pi}^-(v) \leq 2$  and thus,  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^-(v_p) \leq 2$ . Note that if  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^+(v_p) \geq 2$ , then  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^-(v_p) \leq 2$  due to *G* being subquartic. Hence, if  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^+(v_p) = 0$ , then  $d_{\tilde{G}_{\text{tr}}^{\emptyset}}^-(v_p) \leq 2$ . Consequently,  $\delta_{\tilde{G}_{\text{tr}}^{\emptyset}}(v_p) \geq -2$  in any case, so  $\alpha(v) \leq 2$ .

*v* has an incoming, but not an outgoing backward arc and is not a pseudosource. Assume now that *v* has layout  $\curvearrowleft \circ \curvearrowleft$  in  $\pi$ . Let *b* denote *v*'s only incident backward arc, i. e.,  $\mathcal{B}_{\pi}^-(v) = \{b\}$ . There are two possibilities: Either *v* is not contained in any forward path selected for  $\tilde{G}$  besides the one for *b*. In this case, *v* is a pseudosource in  $\tilde{G}$  and therefore also in  $\tilde{G}^{\circ}$  and  $\tilde{G}^{\emptyset}$ . Then,  $\alpha(v) = \alpha_h(v) = 1$ . Otherwise, *v* has layout  $\curvearrowleft \circ \curvearrowright$  or  $\curvearrowleft \circ \curvearrowleft$  in  $\tilde{\pi}$ . Let  $u \in \tilde{V}^{\circ}$  be the vertex with layout  $\curvearrowleft \circ \curvearrowright$  in  $\tilde{\pi}^{\circ}$  that pools *v* and  $k - 1$  further vertices. By Equation (5.7),  $b_{\tilde{\pi}^{\circ}}^-(u) = k$ ,  $1 \leq f_{\tilde{\pi}^{\circ}}^+(u) \leq k + 1$ , and  $f_{\tilde{\pi}^{\circ}}^-(u) = 1$ . In  $\tilde{G}^{\emptyset}$ , *u* is split into two vertices  $u_s$  and  $u_p$  with combined layout  $\curvearrowleft \circ \curvearrowright$  and such that  $b_{\tilde{\pi}^{\emptyset}}^-(u_s) = f_{\tilde{\pi}^{\emptyset}}^+(u_s) = k$ ,  $f_{\tilde{\pi}^{\emptyset}}^-(u_p) = k + 1$ , and  $1 \leq f_{\tilde{\pi}^{\emptyset}}^+(u_p) \leq k + 1$ . Consider  $u_p$ 's delta degree in  $\tilde{G}_{\text{tr}}^{\emptyset}$ . As a

forward path in  $\tilde{G}$  has length at least two, not every head of an outgoing forward arc of  $u_p$  can be a pseudosink in  $\tilde{G}$  and hence be removed to obtain  $\tilde{G}_{\text{tr}}^{\circ}$ . Thus,  $d_{\tilde{G}_{\text{tr}}^{\circ}}^{+}(u_p) \geq 1$  and, subsequently,  $\delta_{\tilde{G}_{\text{tr}}^{\circ}}(u_p) \geq -k$ . Hence,  $\max\{-\delta_{\tilde{G}_{\text{tr}}^{\circ}}(u_p), 0\} \leq k$ , which in turn yields that  $\alpha_d(v) = \frac{1}{k} \cdot \max\{-\delta_{\tilde{G}_{\text{tr}}^{\circ}}(u_p), 0\} \leq 1$ . Furthermore,  $v$  has  $\alpha_h(v) = b_{\pi}^{-}(v) = 1$ , so  $\alpha(v) = \alpha_h(v) + \alpha_d(v) \leq 2$  on average.

*v* has an outgoing, but not an incoming backward arc and is not a pseudosink. Finally, let us consider the case that  $v$  has layout  $\rightarrow \circ \rightarrow$  in  $\pi$  and let  $\mathcal{B}_{\pi}^{+}(v) = \{b\}$ . Similar to the previous case, there are two possibilities: If no forward path for a backward arc different from  $b$  contains  $v$ , then  $v$  is a pseudosink in  $\tilde{G}$  and remains such during the construction of  $\tilde{G}^{\circ}$  and  $\tilde{G}^{\circ}$ . Consequently,  $\alpha(v) = \alpha_t(v_t) = 1$ . In case that at least one forward path actually passes through  $v$ ,  $v$  has layout  $\rightarrow \circ \rightarrow$  or  $\rightarrow \circ \rightarrow$  in  $\tilde{G}$  and is subject to the pooling and polarization employed to obtain first  $\tilde{G}^{\circ}$  and then  $\tilde{G}^{\circ}$ . Consider the vertex  $u \in \tilde{V}^{\circ}$  with layout  $\rightarrow \circ \rightarrow$  in  $\tilde{\pi}^{\circ}$  that pools  $v$  and  $k - 1$  further vertices. Then,  $u$  is split into two vertices  $u_p$  and  $u_t$  with combined layout  $\rightarrow \circ \rightarrow$  for the construction of  $\tilde{G}^{\circ}$  and  $\tilde{\pi}^{\circ}$ . By Equation (5.8),  $1 \leq f_{\tilde{\pi}^{\circ}}^{-}(u) \leq k + 1$ ,  $b_{\tilde{\pi}^{\circ}}^{+}(u) = k$ , and  $f_{\tilde{\pi}^{\circ}}^{+}(u) = 1$ , which implies that after the split,  $1 \leq f_{\tilde{\pi}^{\circ}}^{-}(u_p) \leq k + 1$ ,  $f_{\tilde{\pi}^{\circ}}^{+}(u_p) = k + 1$  and  $f_{\tilde{\pi}^{\circ}}^{-}(u_t) = b_{\tilde{\pi}^{\circ}}^{+}(u_t) = k$ . Consider  $u$ 's passage component  $u_p$ . Note that  $u_t$  is a pseudosink and therefore removed during the construction of  $\tilde{G}_{\text{tr}}^{\circ}$ . Thus,  $1 \leq d_{\tilde{G}_{\text{tr}}^{\circ}}^{-}(u_p) \leq k + 1$  and  $0 \leq d_{\tilde{G}_{\text{tr}}^{\circ}}^{+}(u_p) \leq 1$ . In particular,  $d_{\tilde{G}_{\text{tr}}^{\circ}}^{+}(u_p) = 0$  if and only if the head of  $u$ 's outgoing forward arc is a pseudosink in  $\tilde{G}^{\circ}$  with respect to  $\tilde{\pi}^{\circ}$ . If this is not the case, then  $\delta_{\tilde{G}^{\circ}}(u_p) \geq -k$ . We can draw the same conclusion if  $d_{\tilde{G}^{\circ}}^{-}(u_p) \leq k$ . This yields  $\alpha_t(v) = b_{\pi}^{+}(v) = 1$  and  $\alpha_d(v) = \frac{1}{k} \cdot \max\{-\delta_{\tilde{G}_{\text{tr}}^{\circ}}(u_p), 0\} \leq 1$  on average, i. e.,  $\alpha(v) \leq 2$  on average for every vertex  $v$  that is pooled within  $u$ .

To complete this case, assume that the head of  $u$ 's outgoing forward arc in  $\tilde{\pi}^{\circ}$  is a pseudosink  $t$  and  $d_{\tilde{G}_{\text{tr}}^{\circ}}^{+}(u_p) = 0$  as a result. Furthermore, let  $d_{\tilde{G}_{\text{tr}}^{\circ}}^{-}(u_p) = k + 1$ , which yields  $\delta_{\tilde{G}_{\text{tr}}^{\circ}}(u_p) = -(k + 1)$  and subsequently,  $\alpha_d(v) = \frac{1}{k} \cdot (k + 1)$  with the current attribution scheme. As  $\alpha_t(v) = 1$ , we only have  $\alpha(v) = 1 + \frac{k+1}{k} > 2$  on average. We will show that this common surplus of one for all vertices pooled within  $u$  can be carried over to another vertex such that on average, every vertex is still charged at most two. To this end, we will modify  $\tilde{G}_{\text{tr}}^{\circ}$  a posteriori by removing further arcs and thereby obtain a spanning subgraph  $H = (V_H, A_H) \subseteq \tilde{G}_{\text{tr}}^{\circ}$ . In doing so, however, we will prove that the above assumptions and conclusions for  $\tilde{G}_{\text{tr}}^{\circ}$  also remain valid for  $H$ . Furthermore, the handling of each vertex like  $u$  requires the removal of exactly one arc.



**Figure 5.19:** Case I:  $y$  has no incident backward arcs (a). Case II(a):  $y$  has an incoming backward arc, but its forward path does not use  $(y, z)$  (b). In both cases, the arc corresponding to  $(y, z)$  can be removed a posteriori.

Recall that the pooling of vertices with layout  $\begin{array}{c} \rightarrow \circ \rightarrow \\ \rightarrow \circ \rightarrow \end{array}$  and  $\begin{array}{c} \rightarrow \circ \rightarrow \\ \rightarrow \circ \rightarrow \end{array}$  was defined via the contraction of the vertices' outgoing forward arcs. The fact that  $u_p$  has indegree  $d_{G_{tr}^{\circ}}^-(u_p) = k + 1$  immediately implies that  $f_{\tilde{\pi}}^-(u) = k + 1$ . Subsequently,  $u$  emerged from the pooling of  $k \geq 1$  vertices of  $\tilde{G}$  such that each vertex has layout  $\begin{array}{c} \rightarrow \circ \rightarrow \\ \rightarrow \circ \rightarrow \end{array}$  in the forward path graph  $\tilde{\pi}$  (and none has layout  $\begin{array}{c} \rightarrow \circ \rightarrow \\ \rightarrow \circ \rightarrow \end{array}$ ). Moreover, the  $k + 1$  incoming forward arcs of  $u$  are part of the  $k$  forward paths selected for the backward arcs incident to  $u$  plus the at most two further forward paths for the backward arcs incident to the pseudosink  $t$ . Let  $z$  be a leaf in the for-in-tree that is pooled within  $u$ , i. e., none of both tails of  $z$ 's incoming forward arcs has layout  $\begin{array}{c} \rightarrow \circ \rightarrow \\ \rightarrow \circ \rightarrow \end{array}$  or  $\begin{array}{c} \rightarrow \circ \rightarrow \\ \rightarrow \circ \rightarrow \end{array}$  in  $\tilde{\pi}$ , and let  $\mathcal{B}_{\tilde{\pi}}^+(z) = \mathcal{B}_{\tilde{\pi}}^+(z) = \{b_z\}$ . Consider the penultimate vertex  $y \in \tilde{V}$  on the forward path  $P_{b_z}$  selected for  $b_z$  during the construction of  $\tilde{G}$ . Then,  $(y, z)$  must be a forward arc in  $\tilde{G}$ . More precisely,  $(y, z)$  is an incoming forward arc of  $z$  in  $\tilde{G}$ , so there also is a corresponding incoming forward arc of  $u$  in  $\tilde{G}$ . As  $y$  is the penultimate vertex on a forward path, it cannot be a pseudosource in  $\tilde{\pi}$ , which implies that it can neither be one in  $\pi$  nor have layout  $\begin{array}{c} \rightarrow \circ \rightarrow \\ \rightarrow \circ \rightarrow \end{array}$  in  $\pi$ .

**Case I** Consider the case that  $y$  has no incident backward arcs (cf. Figure 5.19(a)). As we have argued above,  $1 \leq d_{\tilde{G}}^-(y) \leq 3$ . Suppose for the sake of contradiction that by the previous argumentation,  $\alpha(y) = \alpha_d(y) = 2$ . Then,  $\delta_{G_{tr}^{\circ}}(y_p) = -2$ , which implies that  $d_{G_{tr}^{\circ}}^-(y_p) = 3$  and  $d_{G_{tr}^{\circ}}^+(y_p) = 1$ , because  $(y_p, z_p)$  is an outgoing arc of  $y_p$  in  $\tilde{G}_{tr}^{\circ}$ . Consequently, the at most  $k + 2$  forward paths entering  $u$  use the  $k$  other incoming forward arcs of  $u$  besides  $(y, z)$  plus the three incoming forward arcs of  $y$ . This yields  $k + 3$  different forward arcs in total, where no pair of these arcs can be part of the same forward path, a contradiction. Hence,  $d_{G_{tr}^{\circ}}^-(y_p) \leq 2$  and  $\alpha(y) \leq 1$ . We can therefore modify  $H$  in comparison to  $\tilde{G}_{tr}^{\circ}$  by additionally removing the arc corresponding to  $(y, z)$ , i. e., one of the possibly multiple parallel arcs  $(y_p, u_p)$ , without affecting the validity and correctness of the previous arguments. Note that if  $y_p$  has outgoing forward arcs to two

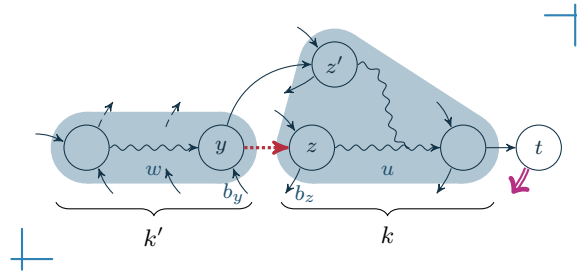


Figure 5.20: Case II(b): There is another forward path for  $b_y$  not using  $(y, z)$ , so the arc corresponding to  $(y, z)$  can be removed.

or three vertices with layout  $\rightarrow \circ \rightarrow$  and a posteriori, i. e., for  $H$ , loses two or three outgoing arcs as a result of this carryover, then  $y_p$  must have had at least as many outgoing arcs before the modification, which reduces the number of possible incoming arcs due to  $G$  being subquartic. Thus, its delta degree would be at least  $-2$  in the first case and exactly  $-1$  in the second, which implies that  $\delta_H(y_p) \geq -2$  and hence  $\alpha(y) \leq 2$  in any case.

**Case II** Otherwise, assume that  $y$  has layout  $\rightarrow \circ \rightarrow$  or  $\rightarrow \circ \rightarrow$ . Then,  $u$  is incident to a vertex  $w \in \tilde{V}^\circ$  with layout  $\rightarrow \circ \rightarrow$ , which pools  $k' \geq 1$  vertices including  $y$ . Note that  $w$  is split into  $w_s$  and  $w_p$  during the construction of  $\tilde{G}^\circ$  with combined layout  $\rightarrow \circ \rightarrow$  in  $\tilde{\pi}^\circ$ . Let  $b_y$  denote  $y$ 's incoming backward arc in  $\pi$ .

**Case II(a)** If the forward path  $P_{b_y}$  selected for  $b_y$  during the construction of  $\tilde{G}$  does not contain  $z$  (cf. Figure 5.19(b)), then we can again modify  $H$  in comparison to  $\tilde{G}_{tr}^\circ$  by additionally removing the arc that corresponds to  $(y, z)$  without affecting the correctness of the above analysis: As the first arc of  $P_{b_y}$  is an outgoing arc of  $w_p$  in both  $\tilde{G}_{tr}^\circ$  and  $H$ ,  $d_H^+(w_p) \geq 1$  and thus,  $\delta_H(w_p) \leq k$  is still guaranteed.

Otherwise,  $P_{b_y}$  is a pseudosink path and both  $P_{b_y}$  and  $P_{b_z}$  contain  $(y, z)$ . Hence,  $b_y$ 's tail is either pooled within  $u$  or equals the pseudosink  $t$ . In consequence of the Alternative Forward Paths Property, at least one of  $b_y$  or  $b_z$  must have another forward path in  $\pi_{sp/r}$  that does not contain  $(y, z)$ . Furthermore, we stipulated that the forward paths starting at a vertex with layout  $\rightarrow \circ \rightarrow$  in  $\pi$  are chosen such that pseudosink paths are avoided wherever possible.

**Case II(b)** Assume that  $b_y$  has another forward path  $P'_{b_y}$  not containing  $(y, z)$  in  $\pi$  and let  $z'$  denote the head of  $y$ 's second outgoing forward arc in  $\pi$ , i. e.,  $\mathcal{F}_\pi^+(y) = \{(y, z), (y, z')\}$  (cf. Figure 5.20). Then,  $P'_{b_y}$  would have been chosen during the construction of  $\tilde{G}$  unless it also was a pseudosink path. Subsequently,  $z'$  must have layout  $\rightarrow \circ \rightarrow$  in  $\pi$  as well as in  $\tilde{\pi}$  and be pooled within  $u$ . As  $u_p$  has the maximum of  $k + 1$  incoming arcs in  $\tilde{G}_{tr}^\circ, \tilde{A}_{tr}^\circ$

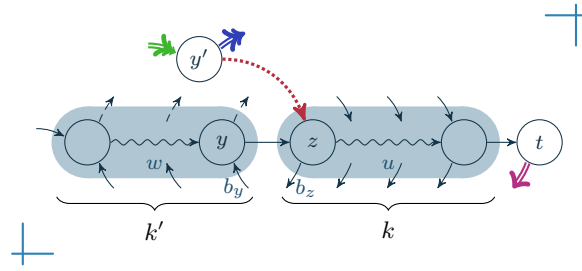


Figure 5.21: Case II(c): There is another forward path for  $b_z$  via  $y'$  and  $y'$  has no incident backward arcs, so the arc corresponding to  $(y', z)$  can be removed.

must in particular contain the arc corresponding to  $(y, z')$ , i. e.,  $\tilde{G}_{tr}^\emptyset$  and  $H$  currently have at least two parallel arcs  $(w, u)$ . Hence, we can again remove the arc corresponding to  $(y, z)$  a posteriori from  $\tilde{G}_{tr}^\emptyset$  to obtain  $H$  and  $d_H^+(w_p) \geq 1$  is still guaranteed. Note that  $u$  is processed after this step, so the arc  $(w_p, u_p)$  that corresponds to  $(y, z')$  cannot be removed later during the processing of another vertex like  $u$  and therefore certainly remains in  $H$ .

Otherwise, if  $P'_{b_y}$  does not exist, then  $b_z$  must have a second forward path  $P'_{b_z}$  that does not contain  $(y, z)$ , but uses  $z$ 's other incoming forward arc  $(y', z)$ . Note that  $(y', z) \in \tilde{A}$  because  $d_{\tilde{G}_{tr}^\emptyset}^-(u_p) = k + 1$ . For the same reason as above,  $y'$  can neither be a pseudosource in  $\pi$  nor have layout  $\rightarrow\alpha\leftarrow$ .

**Case II(c)** If  $y'$  has no incident backward arcs, then we can reuse the arguments that we used for  $y$  in this case and conclude that the additional removal of the arc corresponding to  $(y', z)$  in  $H$  in comparison to  $\tilde{G}_{tr}^\emptyset$  preserves that  $\alpha(y') \leq 2$  (cf. Figure 5.21).

Otherwise,  $y'$  has layout  $\rightarrow\alpha\leftarrow$  in  $\pi$  and is hence pooled within a vertex  $w'$  with layout  $\rightarrow\alpha\leftarrow$  in  $\tilde{\pi}^\circ$ , which is split into two vertices  $w'_s$  and  $w'_p$  with combined layout  $\rightarrow\alpha\leftarrow$  in  $\tilde{\pi}^\emptyset$ . Let  $b_{y'}$  denote the incoming backward arc of  $y'$ .

**Case II(d)** If the forward path  $P_{b_{y'}}$  selected for  $b_{y'}$  does not contain  $(y', z)$ , then its first arc guarantees again that  $d_H^+(w'_p) \geq 1$  even after the belated removal of the arc corresponding to  $(y', z)$  (cf. Figure 5.22(a)).

**Case II(e)** Consider the case that  $P_{b_{y'}}$  contains  $(y', z)$  (cf. Figure 5.22(b)). Then, it is a pseudosink path and  $b_{y'}$ 's tail is either pooled within  $u$  or the pseudosink  $t$ . If there is a second forward path  $P'_{b_{y'}}$  for  $b_{y'}$  that does not use  $(y', z)$ , then it must be a pseudosink path, too, due to the preferential forward path selection policy. As  $f_{\tilde{\pi}}^-(u) = d_{\tilde{G}_{tr}^\emptyset}^-(u_p) = k + 1$ , there is at least one further arc  $(w', u)$  in  $\tilde{G}^\circ$  that corresponds to the first arc of  $P'_{b_{y'}}$ , and after the removal of the arc corresponding to  $(y', z)$  to obtain  $H$ ,  $d_H^+(w'_p) \geq 1$  is still guaranteed. Observe that as in Case II(b),  $u$  is processed after this step, so the arc

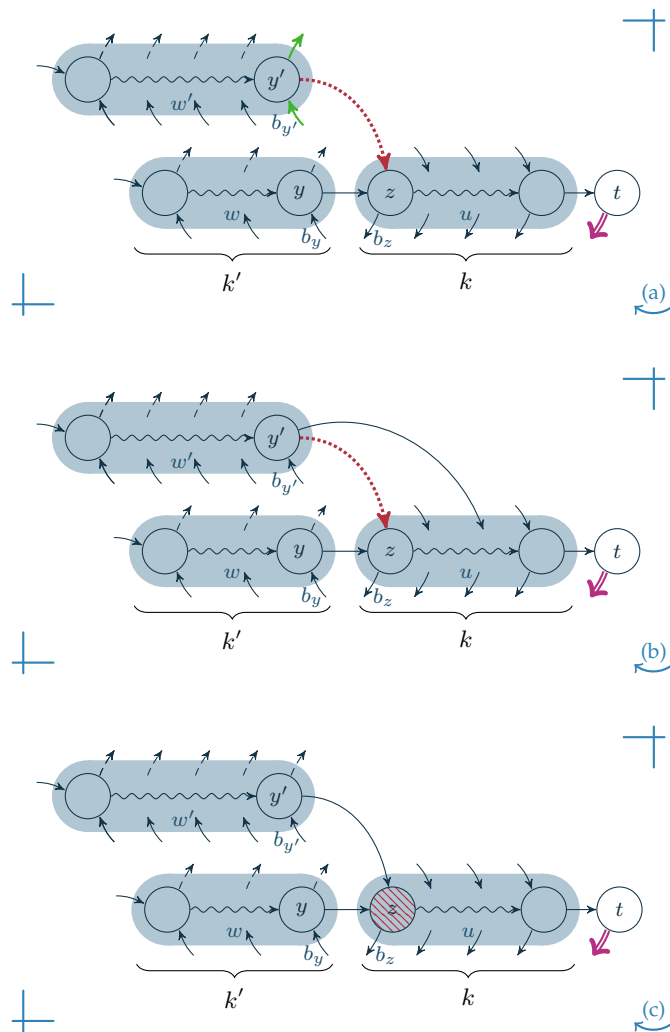


Figure 5.22: Case II(d): The forward path selected for  $b_{y'}$  does not use  $(y', z)$ , so the arc corresponding to  $(y', z)$  can be removed a posteriori (a). Case II(e): There is another forward path for  $b_{y'}$  not using  $(y', z)$ , so the arc corresponding to  $(y', z)$  can be removed a posteriori (b). Case II(f): All forward paths for  $b_y$  and  $b_{y'}$  contain  $z$  and hence in particular  $z$ 's only outgoing forward arc, a contradiction to either the Multipath Blocking Vertices Property if  $b_y$  and  $b_{y'}$  have a common tail, or to the Tail On Forward Path Property otherwise (c).

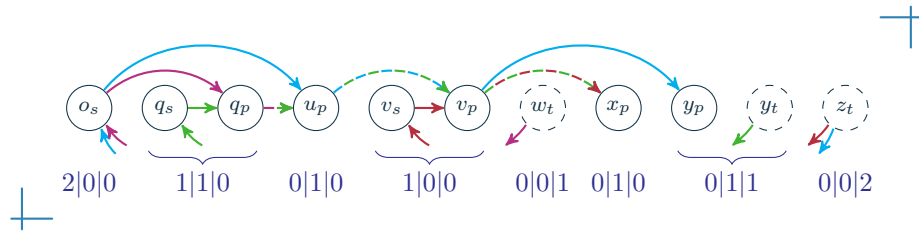


Figure 5.23: The modified truncated forward path graph obtained from the truncated forward path graph in Figure 5.18 by additionally removing  $(x_p, y_p)$  in result of the application of Case I. The new assignments according to Equation (5.10) for each vertex  $v$  of the original graph are again represented as  $\alpha_h(v)|\alpha_d(v)|\alpha_t(v)$ . Vertices with a dashed border, i. e. ,  $w_t, y_t,$  and  $z_t,$  are not contained in the modified truncated forward path graph and only shown here to be able to conveniently display their assignments.







$(w'_p, u_p)$  that corresponds to the second outgoing forward arc of  $y'$  cannot be removed later during the processing of another vertex like  $u$  but certainly remains in  $H$ .

Case II(f) Eventually, suppose that  $b_{y'}$  has only forward paths via  $(y', z)$  (cf. Figure 5.22(c)). Then, the forward paths of both  $b_y$  and  $b_{y'}$  pass through  $z$  and, in particular, use  $z$ 's outgoing forward arc in  $\tilde{G}$ . This implies that  $b_y$  and  $b_{y'}$  either have a common tail or that the tail of one is part of the forward path of the other. In consequence of the Multipath Blocking Vertices Property,  $b_y$  and  $b_{y'}$  must have arc-disjoint forward paths in the former case, whereas in the latter, at least one of them must have a forward path not containing  $z$  by Lemma 5.5, both times a contradiction.

Hence, all possible cases showed that it is safe to remove the arc corresponding to one of  $z$ 's incoming forward arcs a posteriori, which yields that  $d_{\bar{H}}(u_p) = k$  and, subsequently,  $\delta_H(u_p) = -k$ . As a result, for each vertex  $v$  of the  $k$  vertices pooled within  $u$  holds that  $\alpha_t(v) = 1$  and  $\alpha_d(v) \leq 1$  on average. Furthermore, to obtain the modified



Table 5.4: Mappings  $\alpha_h, \alpha_t, \alpha_d$ , and their sum  $\alpha$  for  $v \in V_G$ .

$\mathcal{L}_\pi(v)$	$\alpha_h(v)$	$\alpha_t(v)$	$\alpha_d(v)$	$\alpha(v)$
	$\leq 2$	0	0	$\leq 2$
	0	$\leq 2$	0	$\leq 2$
	1	1	0	2
	1	0	$\leq 1^a$	$\leq 2^a$
	0	1	$\leq 1^a$	$\leq 2^a$
	0	0	$\leq 2$	$\leq 2$

<sup>a</sup>on average

graph  $H$ , we neither remove an arc that is incident to a source in  $\tilde{G}_{tr}^\emptyset$  nor can  $H$  contain a source that was not already one in  $\tilde{G}_{tr}^\emptyset$ . Thus  $H$  is source-preserving and

$$\begin{aligned}
 \alpha_h(v) &= b_\pi^-(v), \\
 \alpha_t(v) &= b_\pi^+(v), \text{ and} \\
 \alpha_d(v) &= \begin{cases} \max\{-\delta_H(v_p), 0\}, & \text{if } v_p \in V_H \text{ is } v\text{'s passage component,} \\ \frac{1}{k} \cdot \max\{-\delta_H(u_p), 0\}, & \text{if } v \text{ and } k-1 \text{ other vertices are pooled} \\ & \text{within a vertex } u \text{ and } u_p \in V_H \text{ is} \\ & u\text{'s passage component,} \\ 0, & \text{otherwise.} \end{cases} \quad (5.10)
 \end{aligned}$$

is an admissible assignment scheme with respect to  $\pi$  that guarantees  $\alpha(v) \leq 2$  on average for every vertex  $v \in V_G$ . Table 5.4 itemizes the assignments for each vertex  $v \in V_G$  once again.

The modified truncated forward path for the running example graph is depicted in Figure 5.23. Here, the vertex  $y$  required a special treatment. As the forward path selected for  $y$ 's outgoing backward arc  $(y, q)$  in the original linear ordering contains  $(x, y)$  as last arc and  $x$  has no incident backward arcs, Case I applies.  $\square$

Together with Proposition 5.1, Lemma 5.8 immediately implies the equivalent of Theorem 5.1 for subquartic graphs:

---



---

**Theorem 5.3**

Every  $\Psi$ -optimal linear ordering  $\pi$  of a subquartic graph having  $n$  vertices such that  $\pi$  additionally respects the Alternative Forward Paths Property and the Tail On Forward Path Property induces at most  $\lfloor \frac{2n}{3} \rfloor$  backward arcs.

---

Recall that the algorithm *PsiOpt*, which we introduced in Section 4.9.2 of the previous chapter and which constructs a  $\Psi$ -optimal linear ordering, uses the meta-algorithm *Cascade* together with subroutines that enforce the individual properties that are grouped together for  $\Psi$ -optimality. By Lemma 4.21, the running time of *Cascade* is  $\mathcal{O}(m)$  times the maximum running time of one of its subroutines. As pointed out in the proof of Lemma 4.22, this also applies to *PsiOpt*, even though the last of its subroutines, *EliminateLayouts*, does not guarantee a strict improvement in case of failure. Furthermore, the most time-consuming subroutine of *PsiOpt* has a running time of  $\mathcal{O}(n \cdot \kappa(n, m))$ , where  $\kappa(n, m)$  represents the time complexity of computing a minimum cut in a unit-capacity network with  $\kappa(n, m) \in \Omega(m) \cap \mathcal{O}(m \cdot \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$  (cf. Section 4.6). By Lemma 5.6 and Lemma 5.7, the Alternative Forward Paths Property and the Tail On Forward Path Property, which are in addition to  $\Psi$ -optimality required for the proof of the subquartic bound, can both be enforced in  $\mathcal{O}(m^2)$ . Hence, if we insert these two further subroutines prior to *EliminateLayouts* in *PsiOpt*, we obtain a new cascading algorithm that establishes all necessary properties. Moreover, with  $m \in \Theta(n)$  for subquartic graphs and hence  $\mathcal{O}(n^2) \subseteq \mathcal{O}(n \cdot \kappa(n, m))$ , we obtain together with an overhead of a factor of  $\mathcal{O}(m) = \mathcal{O}(n)$  for the cascading<sup>1</sup>:

---



---

**Corollary 5.2**

There is an  $\mathcal{O}(n^2 \cdot \kappa(n, m)) \subseteq \mathcal{O}(n^{3.5})$ -time algorithm to construct a feedback arc set with cardinality at most  $\lfloor \frac{2n}{3} \rfloor$  for a subquartic graph.

---

Similar as in the subcubic case, we are again able to show that the bound in Theorem 5.3 is tight:

---

<sup>1</sup>In [HBA13], there is a similar statement that claims a running time of at most  $\mathcal{O}(n^{3.38})$ . However, the underlying linear ordering there does not respect the Multipath Blocking Vertices Property, but only a considerably weaker property.

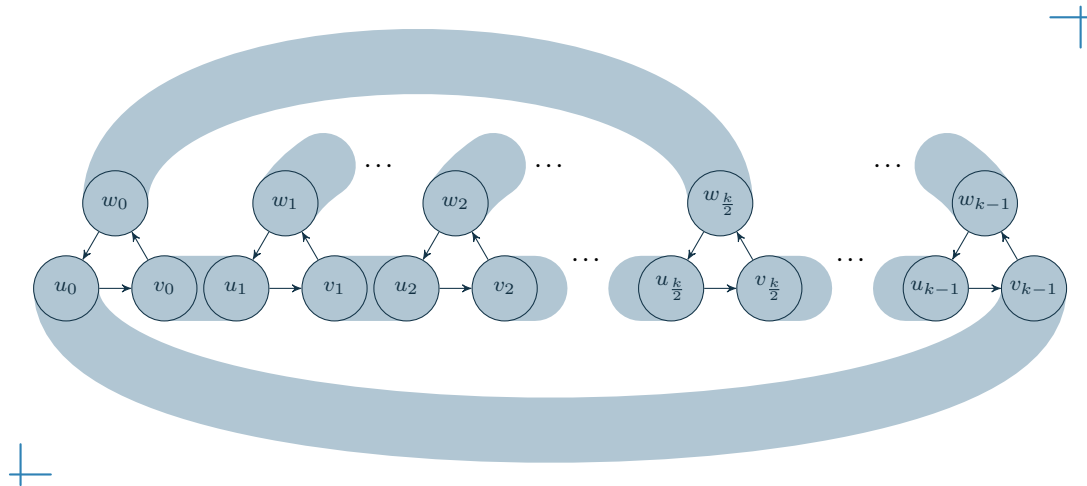


Figure 5.24: Construction of subquartic graphs with  $n$  vertices and  $\tau = \lfloor \frac{2n}{3} \rfloor$ . The shaded regions indicate vertex fusions.

**Lemma 5.9**

For every  $n \geq 6$  there is a subquartic graph on  $n$  vertices whose optimal feedback arc set has cardinality at least  $\lfloor \frac{2n}{3} \rfloor$ .

*Proof.* Let  $k = \lfloor \frac{2n}{3} \rfloor$ . Observe that if  $n \bmod 3 = 0$ , then  $k = 2 \cdot \frac{n}{3}$ , whereas  $k = 2 \cdot \frac{n-1}{3}$  if  $n \bmod 3 = 1$ , and  $k = 2 \cdot \frac{n-2}{3} + 1$  if  $n \bmod 3 = 2$ . We construct a graph  $G_n$  on  $n$  vertices that has exactly  $k$  arc-disjoint cycles of length 3. To this end, let  $T$  be a set of  $k$  vertex-disjoint, directed triangles  $t_0, \dots, t_{k-1}$ , where  $t_i = \langle u_i, (u_i, v_i), v_i, (v_i, w_i), w_i, (w_i, u_i), u_i \rangle$ , i. e.,  $t_i$  consists of the vertices  $u_i, v_i$ , and  $w_i$  and the arcs  $(u_i, v_i), (v_i, w_i)$ , and  $(w_i, u_i)$ . Note that this yields exactly  $3k$  vertices and  $3k$  arcs.

The vertex set  $V_n$  of  $G_n$  is formed by the pairwise fusion of the triangle vertices as follows: For every  $0 \leq i < k - 1$ ,  $V_n$  has a vertex  $v_i u_{i+1}$ , which emerges from the fusion of the triangle vertices  $v_i$  and  $u_{i+1}$ . Additionally, there is a vertex  $v_{k-1} u_0$ . Furthermore, for every  $0 \leq i < \lfloor \frac{k}{2} \rfloor$ ,  $V_n$  contains a vertex  $w_i w_{\lfloor \frac{k}{2} \rfloor + i}$ , and, if  $k$  is odd, also the vertex  $w_{k-1}$ . Figure 5.24 gives an outline of this construction and visualizes the fusions of the triangle vertices. The arc set of  $G_n$  corresponds exactly to the triangles' arcs. In case that  $n \bmod 3 = 1$ , we additionally perform an arc subdivision on an arbitrarily selected arc.

Consequently, if  $n \bmod 3 = 0$ , then  $k$  is even and  $G_n$  has exactly  $k + \frac{k}{2} = \frac{3}{2} \cdot 2 \cdot \frac{n}{3} = n$  vertices and  $3k = 2n$  arcs. Otherwise, if  $n \bmod 3 = 1$ , then  $k$  is also even and  $G_n$  has  $k + \frac{k}{2} + 1 = \frac{3}{2} \cdot 2 \cdot \frac{n-1}{3} + 1 = n - 1 + 1 = n$  vertices and  $3k + 1 = 3 \cdot 2 \cdot \frac{n-1}{3} + 1 = 2n - 1$  arcs.

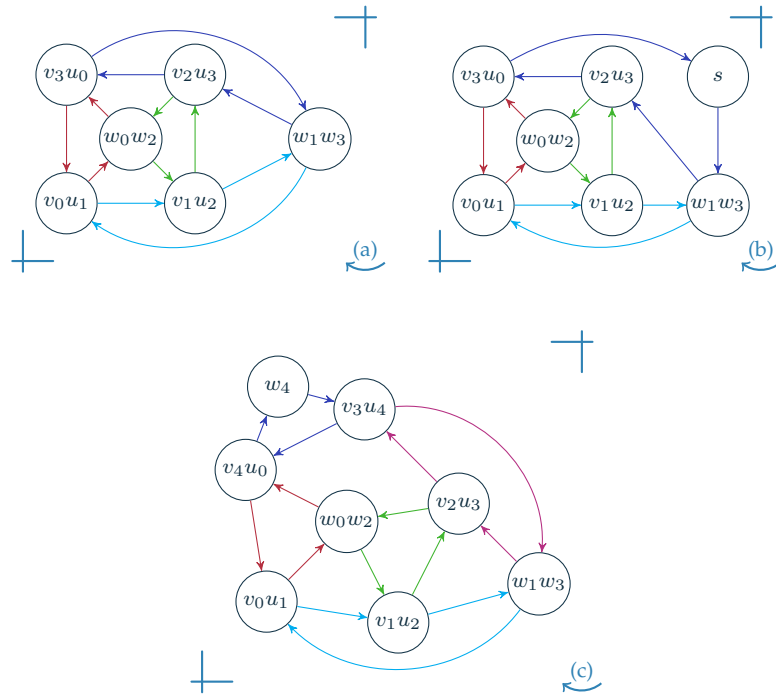


Figure 5.25: Subquartic graphs with  $\tau = \lfloor \frac{2n}{3} \rfloor$  for  $n = 6$  (a),  $n = 7$  (b), and  $n = 8$  (c). Colors highlight the arc-disjoint cycles originating from the directed triangles during the construction. The vertex  $s$  in (b) results from the arc subdivision.

Finally, if  $n \bmod 3 = 2$ , then  $k$  is odd and  $G_n$  has exactly  $k + \lceil \frac{k}{2} \rceil = 2 \cdot \frac{n-2}{3} + 1 + \frac{n-2}{3} + 1 = n - 2 + 2 = n$  vertices and again  $3k = 3 \cdot (2 \cdot \frac{n-2}{3} + 1) = 2n - 1$  arcs. In Figure 5.25, the resulting graphs for  $n = 6$ ,  $n = 7$ , and  $n = 8$  are shown in an exemplary way. By construction,  $G_n$  is strongly connected, subquartic, and has  $k$  arc-disjoint cycles. Furthermore,  $G_n$  is simple for  $n \geq 6$ . Thus,  $\tau_{G_n} \geq k = \lfloor \frac{2n}{3} \rfloor$ .  $\square$

Due to every optimal linear ordering being  $\Psi$ -optimal by Theorem 4.1 and respecting the Alternative Forward Paths Property and the Tail On Forward Path Property by Lemma 5.4 and Lemma 5.5, we obtain<sup>1</sup> from Theorem 5.3 and Lemma 5.9:

---

**Theorem 5.4**















The cardinality of an optimal feedback arc set of a subquartic graph having  $n$  vertices is at most  $\lfloor \frac{2n}{3} \rfloor$  and this bound is tight for all values of  $n \geq 6$ .

---

Note that in case of a quartic graph,  $m = 2n$ .

<sup>1</sup>A similar result has also been published in an earlier conference article [HBA13].



**Table 5.5:** Layouts induced by a  $\Psi$ -optimal linear ordering on a vertex  $v$  with  $d(v) = 5$  as pictograms and 4-tuples.

$b(v) = 2:$	 (0, 3, 2, 0),	 (1, 2, 1, 1),	
	 (3, 0, 0, 2),	 (2, 1, 1, 1)	
$b(v) = 1:$	 (0, 4, 1, 0),	 (1, 3, 1, 0),	 (2, 2, 1, 0),
	 (4, 0, 0, 1)	 (3, 1, 0, 1),	 (2, 2, 0, 1)
$b(v) = 0:$	 (1, 4, 0, 0),	 (2, 3, 0, 0),	
	 (3, 2, 0, 0),	 (4, 1, 0, 0)	

### 5.4.3 Subquintic Graphs

Upon considering vertices of degree five, we observe that the Nesting Property here implies that the number of incident backward arcs in any optimal linear ordering cannot exceed two, as is the case for subquartic graph. Table 5.5 subsumes all possible layouts induced by a  $\Psi$ -optimal linear ordering on a vertex of degree five. Elimenable layouts are omitted.

Let  $\pi$  be a  $\Psi$ -optimal linear ordering of a subquintic graph  $G = (V, A)$  and assume that  $\pi$  also respects the two new properties introduced in Section 5.4.1. Consider a forward path graph  $\vec{G}$  with respect to  $\pi$ , as well as the corresponding pooled, the polarized, and the truncated forward path graph  $\vec{G}^\circ$ ,  $\vec{G}^\partial$ , and  $\vec{G}_{tr}^\partial$ , respectively.

By applying the general assignment scheme from Section 5.3.2 in analogy to the proof of Lemma 5.8 with  $H = \vec{G}_{tr}^\partial$ , we immediately obtain  $\alpha(s) = \alpha_h(s) \leq 2$  for every pseudosource  $s$  and  $\alpha(t) = \alpha_t(t) \leq 2$  for every pseudosink  $t$ . Whereas a vertex  $v$  with layout  is right-blocking and, in consequence of the vertical split, is counted as a pseudosink and a pseudosource, such that  $\alpha(v) = \alpha_h(v) + \alpha_t(v) = 2$ , this does not apply to its left-blocking equivalent, because  $\vec{G} \subseteq G_{sp/r}$ , but  $\vec{G} \not\subseteq G_{sp/l}$  in general. If a vertex  $v$  has layout  in the forward path graph  $\vec{G}$ , then it has three component vertices  $v_s$ ,  $v_p$ , and  $v_t$  in  $\vec{G}^\partial$ , where  $\delta(v_p) \leq 1$ . Subsequently, we obtain only  $\alpha(v) = \alpha_h(v) + \alpha_t(v) + \alpha_d(v) \leq 1 + 1 + 1 = 3$ .



A vertex  $v$  with layout  or  in  $\pi$  is subject to pooling in  $\vec{G}^\circ$  unless there is no forward path passing through it. In this case, it is a pseudosource or a pseudosink in  $\vec{G}$ , which implies that  $\alpha(v) = \alpha_h(v) = 1$  and  $\alpha(v) = \alpha_t(v) = 1$ , respectively. Otherwise, let

Table 5.6: Mappings  $\alpha_h, \alpha_t, \alpha_d$ , and their sum  $\alpha$  for  $v \in V_G$ .

$\mathcal{L}_\pi(v)$	$\alpha_h(v)$	$\alpha_t(v)$	$\alpha_d(v)$	$\alpha(v)$
	$\leq 2$	0	0	$\leq 2$
	0	$\leq 2$	0	$\leq 2$
	1	1	0	2
	1	1	$\leq 1$	$\leq 3$
	1	0	$\leq 1^a$	$\leq 2^a$
	0	1	$\leq 3^a$	$\leq 4^a$
	1	0	$\leq 2$	$\leq 3$
	0	1	$\leq 2$	$\leq 3$
	0	0	$\leq 1$	$\leq 1$
	0	0	$\leq 2$	$\leq 2$
	0	0	$\leq 3$	$\leq 3$

<sup>a</sup>on average

$u$  be the vertex that pools  $k$  vertices with layout , including  $v$ . Here, we can argue similar as in the proof of Lemma 5.8 and thereby obtain  $\delta_{\tilde{G}_{tr}^\emptyset}(u_p) \geq -k$ . Hence,  $\alpha_d(v) \leq 1$  on average. If  $u$  pools  $k$  vertices with layout , then  $d_{\tilde{G}_{tr}^\emptyset}^-(v_p) \leq 3k - k + 1 = 2k + 1$  by Equation (5.2). In consequence, we only obtain  $\alpha_d(v) \leq 3$  on average.

For a vertex  $v$  with layout or , the passage component vertex  $v_p$  may in the worst case inherit both incoming forward arcs and have no outgoing arcs in  $\tilde{G}_{tr}^\emptyset$ . Thus,  $\alpha_d(v) \leq 2$  here.

Finally, the situation in case that  $v$  is a passage with layout , , or is straightforward and yields  $\alpha(v) \leq 1$ ,  $\alpha(v) \leq 2$ , and  $\alpha(v) \leq 3$ , respectively. If  $v$  has layout ,  $d_{\tilde{G}_{tr}^\emptyset}^+(v_p) = 0$  may occur. Then, however,  $v_p$ 's only outgoing forward arc in  $\tilde{G}_{tr}^\emptyset$  is incident to a pseudosink  $t$ , where  $t$  can have at most two outgoing backward arcs. In consequence,  $d_{\tilde{G}_{tr}^\emptyset}^-(v_p) \leq 2$ . If  $d_{\tilde{G}_{tr}^\emptyset}^+(v_p) = 1$ , then we only have  $d_{\tilde{G}_{tr}^\emptyset}^-(v_p) \leq 3$ , which yields  $\delta_{\tilde{G}_{tr}^\emptyset}(v_p) \geq -3$  and hence  $\alpha(v) \leq 3$  in summary. Table 5.6 subsumes these vertex assignments once more.

By modifying  $\tilde{G}_{\text{tr}}^{\emptyset}$  carefully towards a subgraph  $H \subseteq \tilde{G}_{\text{tr}}^{\emptyset}$  and considering vertices with different layouts in unison, we conjecture that it is possible to obtain  $\alpha(v) \leq 2.5$  on average in all of the above cases. Due to the same maximum number of incident backward arcs for subquintic as for subquartic graphs, the arguments for the subquartic case conducted in the proof of [Lemma 5.8](#) should be applicable here for vertices of degree at most four.

---



---

**Conjecture 5.1**

Let  $\pi$  be a  $\Psi$ -optimal linear ordering of subquintic graph  $G$  such that  $\pi$  additionally respects the Alternative Forward Paths Property and the Tail On Forward Path Property. Then, there is an admissible assignment scheme  $\alpha$  with respect to  $\pi$  such that, on average, for every vertex  $v$  in  $G$ ,  $\alpha(v) \leq 2.5$  if  $d(v) = 5$  and  $\alpha(v) \leq 2$  otherwise.

---

If this conjecture holds true, we immediately obtain:

---



---

**Conjecture 5.2**

The cardinality of an optimal feedback arc set of a subquintic graph having  $n$  vertices is at most  $\lfloor \frac{2.5n}{3} \rfloor$ .

---

Note that in case of a quintic graph,  $m = 2.5n$ , so we would obtain an upper bound of  $\frac{m}{3}$  in consequence.





## 6

Exact and Fast Algorithms for  
Linear Ordering

In computational complexity theory,  $\mathcal{NP}$ -hard problems such as the LINEAR ORDERING problem are called “intractable”, which is due to their lack of a polynomial-time algorithm. This shortcoming implies that every algorithm that guarantees to return an optimal solution for arbitrary graphs has a quite long worst-case execution time and is thus usually of little interest in practice—in particular, if the input instances are large. For this reason, a number of heuristics and approximations for the LINEAR ORDERING problem have been developed to obtain good solutions in reasonable time (cf. [Chapter 2](#), [Chapter 4](#)). By contrast, exact approaches are few and far between.

To recapitulate briefly, a naive search for an optimal feedback arc set requires to test up to  $2^m$  possible arc subsets. In case that the input graph is dense, it is more efficient to enumerate all  $n!$  possible linear orderings, though. By means of dynamic programming, an optimal solution can be obtained in as little as  $\mathcal{O}^*(2^n)$  time [[RS07](#), [FK10](#)]. In contrast to the above approaches, however, this procedure also requires exponential space. Alternatively, the algorithm can be brought down to polynomial space, which is accompanied by an increase in running time to  $\mathcal{O}^*(4^{n+o(n)})$ . With the continued improvement of linear (and non-linear) program solvers and the development of new cutting plane strategies that specifically target the LINEAR ORDERING problem [[MR11](#), [BSN15](#)], these specializations of a multi-purpose tool suite seem to have become the de-facto standard to obtain optimal solutions.

Nevertheless, the ever increasing computational power of modern devices also changes the size at which an input instance becomes effectively intractable. This in turn stimulates demand for exact algorithms, whose number and variety can thus be expected to grow at about the same rate as real-world problems become “tractable” in practice. What is more, the quality of non-optimal results for larger instances improve further if heuristics are employed that combine exact solutions of subproblems, like local enumeration (cf. [Section 2.4.1](#)).

In this chapter, we devise a new exact algorithm that performs in particular very well on sparse graphs. Furthermore, it is easy to implement and competitive with existing approaches in that it runs in time  $\mathcal{O}^*(\min\{(\frac{m}{n} + 1)^n, (\lfloor \frac{\Delta_G}{2} \rfloor + 1)^n, \sqrt{2}^{\binom{-n}{-n}} \cdot n!\})$  in general and in  $\mathcal{O}^*(\min\{(\frac{1}{2}\sqrt{k^2 - 1})^n, \sqrt{2}^{\binom{-n}{-n}} \cdot n!\})$  on  $k$ -regular graphs with odd  $k$ , and requires only  $\mathcal{O}(n \cdot m)$  space. Prior to this, we do some groundwork which already yields a simpler version of the aforementioned algorithm as well as one for the corresponding decision problem.

In addition to the default assumptions postulated in Section 3.4.1, we only deal with graphs that are simple, i. e. , we do not allow for multiple parallel arcs. In compensation, the algorithms presented in the following can be adapted to weighted graphs in a straightforward manner, thus providing in turn a means to handle parallel arcs.

---



---

**Assumption 6.1**

The input graph is free of parallel arcs.

---

## 6.1 Partial Layouts and Incomplete Linear Orderings

In Section 3.3.3, we stated that a linear ordering  $\pi$  of a graph  $G = (V, A)$  induces a layout  $\mathcal{L}$  that assigns a 4-tuple to every vertex  $v \in V$  specifying its number of incoming and outgoing forward and backward arcs. More precisely,  $\mathcal{L}(v) = (f^-(v), f^+(v), b^-(v), b^+(v))$ . Let us now reverse our perspective and suppose that we are solely given a layout  $\mathcal{L}$ . What can we tell about those linear orderings inducing  $\mathcal{L}$ ? Evidentially, all of them must imply the same number of backward arcs, as for every linear ordering  $\pi$ ,

$$|\pi| = \sum_{v \in V} b^-(v) = \sum_{v \in V} b^+(v).$$

In fact, it suffices here to know only one of  $b^-$  or  $b^+$ . Let us hence assume that we are just given  $b^-(v)$  for every vertex  $v \in V$ . We call the function  $\lambda : V \rightarrow \mathbb{N}_0$  which assigns each vertex  $v \in V$  a number of incoming backward arcs  $\lambda(v)$  a *partial layout* of  $G$ . Certainly, every linear ordering  $\pi$  unambiguously defines a partial layout  $\lambda_\pi$  such that  $\lambda_\pi(v) = b_\pi^-(v)$ , but not vice versa. A linear ordering  $\pi$  is said to *realize* a partial layout  $\lambda$  if  $\lambda = \lambda_\pi$ . A partial layout is *realizable*, if there is a linear ordering that realizes it. Not every partial layout  $\lambda$  is also realizable, e. g. , if  $\lambda(v) > d^-(v)$ .

Despite its name, a partial layout that is realizable by an optimal linear ordering in fact defines the layout of each vertex entirely.

**Lemma 6.1**

If two linear orderings  $\pi, \pi'$  of a graph  $G = (V, A)$  that both respect the Nesting Property and the Path Property realize the same partial layout  $\lambda$ , then  $\mathcal{B}_\pi = \mathcal{B}_{\pi'}$ .

*Proof.* Consider the set  $X$  of all vertices  $v \in V$  such that  $\lambda(v) = d^-(v)$ . Note that this always applies to the vertex at the smallest position within any linear ordering and thus in particular also to  $\pi^{-1}(0)$  and  $\pi'^{-1}(0)$ . Hence,  $X \neq \emptyset$ . With  $b_\pi^-(v) = b_{\pi'}^-(v) = d^-(v)$ , all incoming arcs of a vertex  $v \in X$  are backward with respect to both  $\pi$  and  $\pi'$ , i. e.,  $v$  has no incoming forward arcs. Subsequently,  $\mathcal{B}_\pi^-(v) = \mathcal{B}_{\pi'}^-(v)$  and  $\mathcal{F}_\pi^-(v) = \mathcal{F}_{\pi'}^-(v) = \emptyset$ . As  $\pi$  and  $\pi'$  respect the Nesting Property,  $v$  cannot have an outgoing backward arc due to [Corollary 4.3](#), so  $\mathcal{B}_\pi^+(v) = \mathcal{B}_{\pi'}^+(v) = \emptyset$  and  $\mathcal{F}_\pi^+(v) = \mathcal{F}_{\pi'}^+(v)$ . Let  $B = \mathcal{B}_\pi^-[X] = \mathcal{B}_{\pi'}^-[X] \subseteq \mathcal{B}_\pi \cap \mathcal{B}_{\pi'}$  and  $F = \mathcal{F}_\pi^+[X] = \mathcal{F}_{\pi'}^+[X] \subseteq \mathcal{F}_\pi \cap \mathcal{F}_{\pi'}$ . As  $\mathcal{F}_\pi^-[X] = \mathcal{F}_{\pi'}^-[X] = \emptyset$ , there is no arc  $(u, v)$  in neither  $\mathcal{F}_\pi$  nor  $\mathcal{F}_{\pi'}$  such that  $u \notin X$  and  $v \in X$ .

Let  $w$  be the vertex at the smallest position within  $\pi$  that is not in  $X$ . Then,  $\mathcal{F}_\pi^-(w) \subseteq \mathcal{F}_\pi^+[X] \subseteq F$ , which implies that the same arcs are also incoming forward arcs of  $w$  in  $\pi'$ , i. e.,  $\mathcal{F}_\pi^-(w) \subseteq \mathcal{F}_{\pi'}^-(w)$ . Furthermore,  $\lambda(w) = d^-(w) - f_\pi^-(w) = d^-(w) - f_{\pi'}^-(w)$ , so  $f_\pi^-(w) = f_{\pi'}^-(w)$  and, as a result,  $\mathcal{F}_\pi^-(w) = \mathcal{F}_{\pi'}^-(w)$  and  $\mathcal{F}_{\pi'}^-(w) \subseteq F$ . Consequently,  $\mathcal{B}_\pi^-(w) = \mathcal{B}_{\pi'}^-(w)$ . Moreover,  $\mathcal{B}_\pi^+(w) \subseteq \mathcal{B}_\pi^-[X] \subseteq B$ , which in analogy to the above implies that the same arcs are also outgoing backward arcs of  $w$  with respect to  $\pi'$ , i. e.,  $\mathcal{B}_\pi^+(w) \subseteq \mathcal{B}_{\pi'}^+(w)$ . Suppose there is an arc  $(w, y) \in \mathcal{B}_\pi^+(w) \setminus \mathcal{B}_{\pi'}^+(w)$ . Then,  $(w, y) \in \mathcal{F}_\pi^+(w)$ . As  $\mathcal{F}_\pi^+(w) \subseteq \mathcal{F}_\pi$  and  $w \notin X, y \notin X$  either. Due to  $\pi'$  respecting the Path Property, there must be a path  $P = y \rightsquigarrow w$  consisting only of arcs in  $\mathcal{F}_{\pi'}$ . In particular,  $P$  must use an arc  $(x, w) \in \mathcal{F}_{\pi'}^-(w)$ . As  $y \notin X$  and  $x \in X$ ,  $P$  contains at least one arc  $(u, v) \in \mathcal{F}_{\pi'}$  such that  $u \notin X$  and  $v \in X$ , a contradiction. Hence,  $\mathcal{B}_\pi^+(w) = \mathcal{B}_{\pi'}^+(w)$  and subsequently,  $\mathcal{F}_\pi^+(w) = \mathcal{F}_{\pi'}^+(w)$ .

Let  $X' = X \cup \{w\}$ ,  $B' = B \cup \mathcal{B}_\pi^-(w) = B \cup \mathcal{B}_{\pi'}^-(w)$ , and  $F' = F \cup \mathcal{F}_\pi^+(w) = F \cup \mathcal{F}_{\pi'}^+(w)$ . Then again,  $B' = \mathcal{B}_\pi^-[X'] = \mathcal{B}_{\pi'}^-[X'] \subseteq \mathcal{B}_\pi \cap \mathcal{B}_{\pi'}$  and  $F' = \mathcal{F}_\pi^+[X'] = \mathcal{F}_{\pi'}^+[X'] \subseteq \mathcal{F}_\pi \cap \mathcal{F}_{\pi'}$ . As the tail of every arc in  $\mathcal{F}_\pi^-(w) = \mathcal{F}_{\pi'}^-(w)$  is in  $X$ , there is no arc  $(u, v)$  in neither  $\mathcal{F}_\pi$  nor  $\mathcal{F}_{\pi'}$  such that  $u \notin X'$  and  $v \in X'$ .

The statement follows by repeating the argument with  $X = X', B = B'$ , and  $F = F'$  until  $X$  contains all vertices of  $G$  and  $B = \mathcal{B}_\pi = \mathcal{B}_{\pi'}$ .  $\square$

As every optimal linear ordering respects the Nesting Property by [Lemma 4.3](#) and the Path Property by [Lemma 4.5](#), we immediately obtain:

---



---

**Corollary 6.1**

If two optimal linear orderings  $\pi^*$ ,  $\pi'^*$  of a graph  $G = (V, A)$  realize the same partial layout  $\lambda$ , then  $\mathcal{B}_\pi = \mathcal{B}_{\pi'}$ .

---

As a benefit, it suffices to check whether a partial layout is realizable by any linear ordering if we are only interested in those that are optimal.

In this chapter, we will also deal with linear orderings of a graph  $G = (V, A)$  that do not (yet) assign a LO position to each vertex  $v \in V$ . Such a linear ordering  $\pi$  is said to be *incomplete* and defined as a mapping  $\pi : V \rightarrow \{0, \dots, n-1\} \cup \{\perp\}$  which either assigns a unique LO position to a vertex  $v \in V$  or  $\perp$  if the LO position is undefined. Besides, there may be no “gaps” within the incomplete linear ordering. More formally, if  $\pi(v) \neq \perp$ , then  $\forall i \in \{0, \dots, \pi(v)\} \exists u \in V : \pi(u) = i$ . With a slight abuse of terminology, we say that a vertex  $v$  is *contained* in  $\pi$  if  $\pi(v)$  is defined, i. e.,  $\pi(v) \neq \perp$ . The *length*  $l(\pi)$  of an (incomplete) linear ordering  $\pi$  is the number of vertices contained in  $\pi$ , i. e.,  $l(\pi) = |\{v \in V \mid \pi(v) \neq \perp\}|$ . Furthermore, the *extension* of an incomplete linear ordering  $\pi$  by a vertex  $v$  with  $\pi(v) = \perp$  yields an (incomplete) linear ordering  $\pi'$  that assigns each vertex contained in  $\pi$  the same LO position and  $\pi'(v) = l(\pi)$  additionally.

Let  $\pi$  be an incomplete linear ordering. In favor of a concise notation, we adapt some definitions for conventional linear orderings to incomplete ones. As a matter of principle, every vertex  $v$  that is not contained in  $\pi$  is treated as if it extended  $\pi$ , i. e., as if we considered the extension of  $\pi$  by  $v$ . Hence, if a vertex  $v$  has an arc to or from a vertex not contained in  $\pi$ , then this arc is considered as an outgoing forward arc or an incoming backward arc of  $v$ , respectively. More formally,

$$\begin{aligned} \mathcal{F}_\pi^+(v) &= \{(v, u) \mid \pi(u) = \perp \vee (\pi(v) \neq \perp \wedge \pi(v) < \pi(u))\}, \\ \mathcal{B}_\pi^-(v) &= \{(u, v) \mid \pi(u) = \perp \vee (\pi(v) \neq \perp \wedge \pi(v) < \pi(u))\}, \\ \mathcal{F}_\pi^-(v) &= \{(u, v) \mid \pi(u) \neq \perp \wedge (\pi(v) = \perp \vee \pi(u) < \pi(v))\}, \\ \mathcal{B}_\pi^+(v) &= \{(v, u) \mid \pi(u) \neq \perp \wedge (\pi(v) = \perp \vee \pi(u) < \pi(v))\}. \end{aligned}$$

The definitions of  $f^-$ ,  $f^+$ ,  $b^-$ ,  $b^+$ , and  $\mathcal{L}$  follow accordingly. Nevertheless, the number of backward arcs induced by  $\pi$  is defined only via the vertices contained in  $\pi$ , i. e.,

$$|\pi| = \sum_{v \in V : \pi(v) \neq \perp} b^-(v).$$

In subsequence, the extension of a linear ordering never decreases the number of induced backward arcs.

## 6.2 Exact Algorithms for Optimization and Decision

We now cast the results from [Section 6.1](#) and in particular [Lemma 6.1](#) and [Corollary 6.1](#) in a relatively simple, yet surprisingly fast algorithm which solves the LINEAR ORDERING problem to optimality. Afterwards, we straightforwardly derive an algorithm for the corresponding decision problem. The procedure described in the following lemma provides the basis for both:

---



---

### Lemma 6.2

Let  $\lambda : V \rightarrow \mathbb{N}_0$  be a partial layout of graph  $G = (V, A)$ . There is an  $\mathcal{O}(m)$ -time and  $\mathcal{O}(n)$ -space algorithm that either guarantees that no optimal linear ordering realizing  $\lambda$  exists or returns a candidate linear ordering that realizes  $\lambda$ .

---

*Proof.* Consider the function  $CheckPartialLayout(G, \lambda)$  listed in [Algorithm 6.1](#), which obtains a simple graph  $G = (V, A)$  along with the specification of  $\lambda$  as arguments and operates similar to an algorithm for topological sorting. As we are given the number of incoming backward arcs  $\lambda(v) = b^-(v)$  for every vertex  $v \in V$ , we can immediately derive the number of incoming forward arcs  $f^-(v)$  as  $d^-(v) - b^-(v)$  (cf. [line 5](#)). Furthermore, we maintain a bucket for every possible number of incoming forward arcs and pigeon-hole the vertices accordingly (cf. [line 2](#) and [line 6](#)). Observe that we stipulated in [Assumption 3.2](#) that  $G$  is strongly connected, which implies at least one incoming and one outgoing arc per vertex and hence  $\forall v \in V : d^-(v) < \Delta_G$ .

After these preprocessing steps, we start to iteratively construct a linear ordering  $\pi$  by extending  $\pi$  by a vertex  $v$  if and only if exactly  $f^-(v)$  vertices from  $N^-(v)$  are already contained in  $\pi$ . The algorithm returns  $\pi$  in [line 21](#) if  $\pi$  could be constructed such that it induces  $\lambda$  and if  $\pi$  is not glaringly non-optimal, e. g., because it violates the Nesting Property. In the latter case,  $\lambda$  is rejected by returning  $\perp$ .

For the construction of  $\pi$ ,  $CheckPartialLayout$  makes use of the buckets to ensure that exactly  $f^-(v)$  vertices from  $N^-(v)$  are already contained in the linear ordering before extending it by  $v$ . To this end, it maintains as invariant that if a vertex  $v$  is contained in bucket  $S[j]$ , then  $j$  vertices having an arc to  $v$  still need to be processed before  $v$ . Thus, a vertex may only extend the linear ordering if it currently resides in the bucket  $S[0]$ .

Let us briefly consider the relationship between two vertices  $u, v \in S[0]$ ,  $u \neq v$ , for any point in time during the execution of the algorithm. Suppose  $(v, u) \in A$ . If  $CheckPartialLayout$  processes  $u$  before  $v$ ,  $u$ 's number of incoming backward arcs would

---

**Algorithm 6.1** Check a partial layout and construct a linear ordering inducing it.

---

**Require:** simple graph  $G = (V, A)$ , partial layout  $\lambda : V \rightarrow \mathbb{N}_0$

**Return:** reject  $\lambda$  by returning  $\perp$  or return a linear ordering realizing  $\lambda$

```

1: procedure CheckPartialLayout( $G, \lambda$ )
2:   for all  $j \in \{0, \dots, \Delta_G - 1\}$  do  $S[j] \leftarrow \emptyset$ 
3:    $b^- \leftarrow \lambda$ 
4:   for all  $v \in V$  do
5:      $f^-(v) \leftarrow d^-(v) - b^-(v)$             $\triangleright$  compute number of incoming forward arcs
6:      $S[f^-(v)] \leftarrow S[f^-(v)] \cup \{v\}$         $\triangleright$  sort vertex into bucket
7:    $\pi \leftarrow \perp^V$ 
8:    $i \leftarrow 0$ 
9:   while  $S[0] \neq \emptyset$  do
10:    remove one vertex  $v$  from  $S[0]$ 
11:    if  $2 \cdot b^-(v) \geq n - i - |S[0]|$  then return  $\perp$             $\triangleright \pi$  cannot be optimal
12:     $\pi \leftarrow \text{insert}(\pi, v, i)$ 
13:     $i \leftarrow i + 1$ 
14:    for all outgoing arcs  $(v, u)$  of  $v$  do            $\triangleright$  update all outgoing neighbors  $u$  of  $v$ 
15:      if  $\pi(u) = \perp$  then
16:        let  $S[j]$  be the bucket containing  $u$ 
17:        if  $j = 0$  then return  $\perp$             $\triangleright b^-(u)$  is less than  $\lambda(u)$ 
18:         $S[j] \leftarrow S[j] \setminus \{u\}$             $\triangleright$  update bucket of  $u$ 
19:         $S[j - 1] \leftarrow S[j - 1] \cup \{u\}$ 
20:    if  $i < n$  then return  $\perp$             $\triangleright \lambda$  unrealizable for at least one vertex
21:    return  $\pi$ 

```

---

match  $\lambda(u)$  exactly. Thus,  $(v, u)$  must be a backward arc in order to comply with  $\lambda(u)$ . Furthermore, if  $\pi$  is optimal, it must respect the Nesting Property by [Lemma 4.3](#). Hence, there must be a vertex  $w$  such that  $\pi(u) < \pi(w) < \pi(v)$  and  $(w, v)$  is an incoming forward arc of  $v$ . In consequence,  $w \in N^-(v)$  and, as  $v \in S[0]$ ,  $\pi$  induces strictly less incoming backward arcs on  $v$  than specified by  $\lambda(v)$ . Conversely, if *CheckPartialLayout* processes  $v$  before  $u$ , then  $(v, u)$  is a forward arc. As  $u \in S[0]$ ,  $b_{\pi}^-(u) < \lambda(u)$ . Subsequently, no two vertices in  $S[0]$  may be adjacent.

At the beginning of the main part of the algorithm, the linear ordering  $\pi$  is “empty”, i. e., the LO position is undefined for all vertices (cf. [line 7](#)). We also use a variable

$i$  to specify the current number of vertices that have already been ordered, which is initialized with 0 in [line 8](#). In [lines 9–19](#), *CheckPartialLayout* loops over the vertices contained in bucket  $S[0]$ . In each iteration, it removes one arbitrary vertex  $v$  from  $S[0]$  and checks whether it can have  $\lambda(v)$  incoming backward arcs and at the same time have position  $i$  in  $\pi$ . Note that if  $v$  has  $b^-(v)$  incoming backward arcs and  $f^+(v)$  outgoing forward arcs, then at least  $f^+(v) + b^-(v)$  vertices must have a position strictly greater than  $i$  in  $\pi$ , as we declared  $G$  to be simple in [Assumption 6.1](#). Furthermore, we have already argued that if  $\pi$  is optimal, then  $v$  cannot be adjacent to another vertex in  $S[0]$ . In consequence of the Nesting Property and in particular [Corollary 4.4](#),  $f^+(v) \geq b^-(v)$ , which yields that  $2 \cdot b^-(v) < n - i - |S[0]|$ . Hence, *CheckPartialLayout* rejects  $\lambda$  and returns  $\perp$  in [line 11](#) if this inequality is violated. Otherwise, we extend  $\pi$  by  $v$ . More formally,  $v$  is inserted into  $\pi$  at position  $i$  and  $i$  is incremented. The algorithm now updates the buckets of all vertices having an incoming arc from  $v$  in [lines 14–19](#) that are not contained in  $\pi$  yet. Let  $u$  be one such vertex and  $u \in S[j]$ . As argued above,  $\pi$  cannot be optimal if  $u \in S[0]$ . Thus, *CheckPartialLayout* rejects  $\lambda$  in this case (cf. [line 17](#)). Observe that  $v \in N^-(u)$ . As  $v$  is now contained in  $\pi$ ,  $u$  can be moved from bucket  $S[j]$  to  $S[j - 1]$  in [lines 18–19](#). Once  $S[0]$  is empty, the algorithm checks whether  $\pi$  contains all vertices and terminates.

For the analysis of the time complexity, we assume that a vertex's addition to or removal from a bucket requires constant time. As  $\Delta_G < n$ , the preprocessing steps in [lines 2–6](#) can be accomplished in time  $\mathcal{O}(n)$ . For every vertex that is processed in the loop spanning [lines 9–19](#), the algorithm needs to consider at most all outgoing arcs and sort the heads into a new bucket. In consequence, the number of steps required in the main part is  $\sum_{v \in V} d^+(v) = m$ , which yields a time complexity of  $\mathcal{O}(m)$ . Due to  $G$  being simple and strongly connected,  $n \in \mathcal{O}(m)$ , so the algorithm runs in time  $\mathcal{O}(m)$ . The space complexity follows by observing that at any point during the algorithm, all buckets together contain at most  $n$  elements.  $\square$

With the help of [Lemma 6.2](#), we are able to design an exact algorithm by enumerating all contemplable partial layouts:

---

### **Theorem 6.1**

There is an  $\mathcal{O}(n)$ -space algorithm that constructs an optimal linear ordering  $\pi^*$  of a graph  $G$  with maximum vertex degree  $\Delta_G$  in time  $\mathcal{O}(m \cdot (\frac{1}{2}\sqrt{k^2 - 1})^n)$  if  $G$  is  $k$ -regular and  $k$  is odd, and  $\mathcal{O}(m \cdot (\min\{\frac{m}{n}, \lfloor \frac{\Delta_G}{2} \rfloor\} + 1)^n)$  otherwise.

---

---

**Algorithm 6.2** A simple, exact algorithm.

---

**Require:** simple graph  $G = (V, A)$

**Return:** the returned linear ordering is optimal

1: **procedure** *ExactLOSimple*( $G$ )

2:    $\lambda \leftarrow 0^V$

3:   **repeat**

4:      $\lambda \leftarrow \text{next}(G, \lambda)$     $\triangleright$  construct next  $\lambda$  such that  $\sum_{v \in V} \lambda(v)$  is non-decreasing

5:      $\pi \leftarrow \text{CheckPartialLayout}(G, \lambda)$

6:   **until**  $\pi \neq \perp$

7:   **return**  $\pi$

---

*Proof.* Consider the algorithm *ExactLOSimple*( $G$ ) listed in [Algorithm 6.2](#), which obtains a graph  $G = (V, A)$  as parameter. It first initializes a partial layout  $\lambda$  in [line 2](#) with the constant-zero function. Then, it enters a loop that enumerates partial layouts in some non-decreasing order with respect to  $\sum_{v \in V} \lambda(v)$  in [line 4](#) and passes each to *CheckPartialLayout* in [line 5](#). As soon as this subroutine returns a linear ordering  $\pi \neq \perp$  and thereby reports that the current partial layout is realizable, *ExactLOSimple* abandons the search and returns  $\pi$ .

Before giving our attention to the question how many interesting partial layouts exist and how they can be enumerated, let us briefly address the overall correctness of *ExactLOSimple*. First, we may note that the initial value of  $\lambda$ , which assigns each vertex zero incoming backward arcs, is never checked. However, this choice implies that for any linear ordering  $\pi$  realizing  $\lambda$ ,  $|\pi| = 0$ , i. e.,  $G$  is acyclic, a contradiction to [Assumption 3.2](#), which states that all graphs are strongly connected. Second, the algorithm returns immediately if a check was successful and does not consider any further partial layouts. As the invariant during the enumeration of partial layouts is that  $\sum_{v \in V} \lambda(v)$  is non-decreasing, the number of induced backward arcs in the next iteration must be either the same or greater. As we are interested in an optimal solution, i. e., one with the minimum number of induced backward arcs, this behavior is correct. Furthermore, by [Corollary 6.1](#), it suffices to test for each partial layout only whether there is an arbitrary linear ordering that realizes it.

How many partial layouts of  $G$  do we need to enumerate in the worst case? In consequence of the Nesting Property (cf. [Lemma 4.3](#), [Corollary 4.4](#)), an optimal linear



ordering  $\pi^*$  of  $G$  induces a layout on every vertex  $v \in V$  such that  $b^-(v) \leq f^+(v)$ . Hence, with  $b^-(v) + f^+(v) \leq d(v)$ ,

$$b^-(v) \leq \lfloor \frac{d(v)}{2} \rfloor.$$

The number of incoming backward arcs  $b^-(v)$  for every vertex  $v$  thus ranges between 0 and  $\lfloor \frac{d(v)}{2} \rfloor$ , which yields a total of  $\lfloor \frac{d(v)}{2} \rfloor + 1$  possibilities per vertex and at most

$$\prod_{v \in V} \left( \lfloor \frac{d(v)}{2} \rfloor + 1 \right)$$

possible partial layouts  $\lambda$ . Due to the inequality of the arithmetic and geometric means<sup>1</sup> and the monotonicity of the exponentiation we can bound this value from above by

$$\begin{aligned} \prod_{v \in V} \left( \lfloor \frac{d(v)}{2} \rfloor + 1 \right) &= \left( \sqrt[n]{\prod_{v \in V} \left( \lfloor \frac{d(v)}{2} \rfloor + 1 \right)} \right)^n \\ &\leq \left( \frac{1}{n} \cdot \sum_{v \in V} \left( \lfloor \frac{d(v)}{2} \rfloor + 1 \right) \right)^n \\ &= \left( \frac{1}{n} \cdot \left( \sum_{v \in V} \lfloor \frac{d(v)}{2} \rfloor + n \right) \right)^n \\ &= \left( \frac{1}{n} \cdot \sum_{v \in V} \lfloor \frac{d(v)}{2} \rfloor + 1 \right)^n \\ &\leq \left( \frac{1}{n} \cdot \sum_{v \in V} \frac{d(v)}{2} + 1 \right)^n \\ &= \left( \frac{1}{2n} \cdot \sum_{v \in V} d(v) + 1 \right)^n \\ &= \left( \frac{1}{2n} \cdot 2m + 1 \right)^n \\ &= \left( \frac{m}{n} + 1 \right)^n. \end{aligned}$$

Depending on the vertices' degree distribution in  $G$ , a better upper bound may be obtained by observing that

$$\lfloor \frac{d(v)}{2} \rfloor \leq \lfloor \frac{\Delta_G}{2} \rfloor.$$

In combination, the number of interesting partial layouts of a graph  $G$  is therefore at most

$$\left( \min \left\{ \frac{m}{n}, \lfloor \frac{\Delta_G}{2} \rfloor \right\} + 1 \right)^n.$$

<sup>1</sup>The inequality states that  $\sqrt[n]{x_1 \cdot x_2 \cdot \dots \cdot x_n} \leq \frac{x_1 + x_2 + \dots + x_n}{n}$ .

In case that  $G$  is  $k$ -regular and  $k$  is odd, the Eliminateable Layouts Property imposes a further restriction: Consider a vertex  $v$  that has the maximum of  $\lfloor \frac{k}{2} \rfloor = \frac{k-1}{2}$  incoming backward arcs, i. e.,  $b^-(v) = \frac{k-1}{2}$ . The Nesting Property again implies that  $f^+(v) \geq b^-(v)$ , i. e.,  $f^+(v) \geq \frac{k-1}{2}$ . If  $f^+(v) = \frac{k-1}{2}$ , however, then  $f^-(v) = 1$  and  $b^+(v) = 0$ , which is an eliminateable layout. By [Lemma 4.18](#), there is an optimal linear ordering  $\pi^*$  of  $G$  such that  $\pi^*$  does not induce an eliminateable layout on any vertex  $v \in V$ . Subsequently, it suffices to assume that  $f^+(v) = \frac{k+1}{2}$ . But then again,  $v$ 's outdegree  $d^+(v) = f^+(v) = \frac{k+1}{2}$  and  $v$ 's indegree  $d^-(v) = b^-(v) = \frac{k-1}{2}$ . Due to the handshaking lemma, at most half of all vertices of  $G$  can have  $\frac{k+1}{2}$  outgoing and  $\frac{k-1}{2}$  incoming arcs, which in turn implies that at most  $\frac{n}{2}$  vertices can have the maximum of  $\frac{k-1}{2}$  incoming backward arcs, and the other  $\frac{n}{2}$  vertices can have at most  $\frac{k-3}{2}$  incoming backward arcs. Thus, the number of partial layouts to consider is

$$\begin{aligned} \left(\frac{k-1}{2} + 1\right)^{\frac{n}{2}} \cdot \left(\frac{k-3}{2} + 1\right)^{\frac{n}{2}} &= \left(\frac{k+1}{2}\right)^{\frac{n}{2}} \cdot \left(\frac{k-1}{2}\right)^{\frac{n}{2}} \\ &= \left(\frac{k+1}{2} \cdot \frac{k-1}{2}\right)^{\frac{n}{2}} \\ &= \left(\frac{k^2-1}{4}\right)^{\frac{n}{2}} \\ &= \left(\frac{1}{2}\sqrt{k^2-1}\right)^n. \end{aligned}$$

Note that in the listing of the algorithm, this behavior is hidden in the functionality of  $\text{next}(G, \lambda)$ .

If  $G$  is  $k$ -regular and  $k$  is even, then every vertex may have  $\frac{k}{2}$  outgoing and  $\frac{k}{2}$  incoming arcs and  $b^-(v) = \frac{k}{2}$  is possible for every  $v \in V$ . As  $m = \frac{kn}{2}$  and hence,  $k = \frac{2m}{n}$ , the number of possible partial layouts in this case is

$$\left(\frac{k}{2} + 1\right)^n = \left(\frac{2m}{2n} + 1\right)^n = \left(\frac{m}{n} + 1\right)^n,$$

which meets exactly the previously established bound for general graphs.

Irrespective of whether  $G$  is  $k$ -regular or not, the contemplable partial layouts can be enumerated as requested in a non-decreasing fashion with respect to  $\sum_{v \in V} \lambda(v)$ . Furthermore, we assume that the order of the enumeration is fixed. Then, given a partial layout  $\lambda$ , the next partial layout in order can be constructed in situ and in at most  $\mathcal{O}(n)$  time. Furthermore, by [Lemma 6.2](#),  $\text{CheckPartialLayout}$  runs in time  $\mathcal{O}(m)$  and needs  $\mathcal{O}(n)$  space. As  $n \in \mathcal{O}(m)$ , each iteration of the loop hence requires  $\mathcal{O}(m)$  time and  $\mathcal{O}(n)$  space. In total,  $\text{ExactLOSimple}$  has a space complexity of  $\mathcal{O}(n)$  and a running time of

---

**Algorithm 6.3** An algorithm for the LINEAR ORDERING decision problem.

---

**Require:** simple graph  $G = (V, A)$ , upper bound  $k$

**Return:** true if  $G$  has a linear ordering  $\pi$  such that  $|\pi| \leq k$ , otherwise false

```

1: procedure DecideLO( $G, k$ )
2:    $\lambda \leftarrow \text{next}(G, 0^V)$  ▷ obtain first  $\lambda$  with  $\sum_{v \in V} \lambda(v) = 1$ 
3:   while  $\sum_{v \in V} \lambda(v) \leq k$  do
4:      $\pi \leftarrow \text{CheckPartialLayout}(G, \lambda)$ 
5:     if  $\pi \neq \perp$  then return true
6:      $\lambda \leftarrow \text{next}(\lambda)$  ▷ construct next  $\lambda$  such that  $\sum_{v \in V} \lambda(v)$  is non-decreasing
7:   return false

```

---

$\mathcal{O}(m \cdot (\min\{\frac{m}{n}, \lfloor \frac{\Delta_G}{2} \rfloor\} + 1)^n)$  for general graphs and  $\mathcal{O}(m \cdot (\frac{1}{2}\sqrt{k^2 - 1})^n)$  for  $k$ -regular graphs with  $k$  odd. □

The fact that *ExactLOSimple* enumerates partial layouts in a non-decreasing fashion allows us to turn it straightforwardly into an effective algorithm for the decision problem.

---

**Theorem 6.2**

There is an  $\mathcal{O}(n)$ -space algorithm that decides in  $\mathcal{O}(m \cdot (n + k - 1)^k)$  time whether a given graph  $G$  has a linear ordering  $\pi$  such that  $|\pi| \leq k$ .

---

*Proof.* Consider the modification of *ExactLOSimple* as listed in [Algorithm 6.3](#). In contrast to the original, *DecideLO*( $G, k$ ) receives an additional parameter  $k$  that serves as an upper bound on the cardinality of the induced set of backward arcs of the linear orderings to consider. Like *ExactLOSimple*, it initializes  $\lambda$  as a first step in [line 2](#), here however by setting it to the first partial layout whose sum of function values equals one, which is obtained by calling *next* on the constant-zero partial layout  $0^V$ . Afterwards, the algorithm enters a while loop that has as invariant that  $\sum_{v \in V} \lambda(v)$  may not exceed  $k$ , i. e., the upper bound passed as parameter must be observed. In [line 4](#), it then checks whether  $\lambda$  can be realized by a linear ordering and if so, returns true, because this linear ordering induces at most  $k$  backward arcs. Otherwise, the algorithm constructs the next partial layout such that the sum of its function values is not less than the current one and the statements in the loop's body are repeated. This continues until either a partial layout is realizable or the upper bound  $k$  is exceeded. In the latter case, the loop

terminates regularly and the algorithm returns `false`, because every linear ordering of  $G$  must induce strictly more than  $k$  backward arcs.

Naturally, every partial layout considered by *DecideLO* would also have been considered by *ExactLOSimple*. For this reason, its running time can be at most that of the algorithm solving the optimization problem and the space complexity is in fact the same. However, we can now additionally bound the running time of *DecideLO* in terms of the parameter  $k$ . To this end, let us consider the number of partial layouts whose sum of function values equals an integer  $i$ . This corresponds to the number of possibilities of distributing  $i$  identical balls in  $n$  bins and hence is exactly  $\binom{n+i-1}{i}$ . As *DecideLO* considers only partial layouts such that  $i \leq k$ , we obtain a total of  $\sum_{1 \leq i \leq k} \binom{n+i-1}{i} \in \mathcal{O}((n+k-1)^k)$  partial layouts. With a cost of  $\mathcal{O}(m)$  for checking each partial layout, the statement follows.  $\square$

In consequence, we can decide in polynomial time whether a graph has a linear ordering inducing at most  $k$  backward arcs if  $k$  is fixed. Unfortunately, the degree of the polynomial depends directly on  $k$ .

### 6.3 Branch and Bound with Integrated Partial Layouts


Especially in case that  $|\pi^*|$  is large, one major drawback of the approach developed in the previous section for solving the optimization problem consists in the construction of an expectably huge number of partial layouts  $\lambda$  that are rejected by *CheckPartialLayout*. In a more sophisticated approach, we therefore seek to overcome this issue by generating only those partial layouts  $\lambda$  that are also realizable, i. e., that have a candidate linear ordering inducing them. We will have to trade this, however, for the invariant that the partial layouts are considered in a monotonically increasing order with respect to the number of induced backward arcs.

Furthermore, we address another important shortcoming of *ExactLOSimple*: If  $G$  is dense enough, its worst-case running time may exceed the effort of simply enumerating all  $n!$  possible linear orderings. To this end, we modify *ExactLOSimple* such that the construction of the linear ordering is integrated in the enumeration of all possible partial layouts  $\lambda$ .

The algorithm we employ here as a subroutine, *ExtendLO*, is recursive in nature and builds a linear ordering stepwise by extending it by vertices, similar to a simple and straightforward brute-force search. In doing so, however, it guarantees that it never


generates two linear orderings that induce the same partial layout  $\lambda$ . [Algorithm 6.4](#) outlines  $ExtendLO(G, \pi, t, E, Z, D)$ . Apart from the graph  $G$ , the routine receives an incomplete linear ordering  $\pi$ , a value  $t$  telling the size of the best solution found so far, as well as three sets  $E, Z, D$  as arguments, the latter of which are used to classify vertices and thereby keep the above guarantee of not testing the same partial layout more than once: Every vertex  $v$  belongs in exactly one of four categories depending on  $v$ 's current status and, if applicable, the effect on  $\pi$  if  $v$  would extend the incomplete ordering  $\pi$  in the next step. Additionally, we assume that for each partial layout  $\lambda$  that is still to be considered by the algorithm, there is an individual upper bound  $\lambda_{\max}$  for each vertex  $v$ , i. e.,  $\lambda(v) \leq \lambda_{\max}(v)$ .

**processed:**  $\pi(v) \neq \perp$ , i. e.,  $\pi(v)$  is defined.

**eligible (E):**  $\pi(v) = \perp$ ,  $1 \leq b^-(v) \leq f^+(v)$ , the arcs in  $\mathcal{B}^+(v)$  and  $\mathcal{F}^-(v)$  nest as required by the Nesting Property,  $b^-(v) \leq \lambda_{\max}(v)$ , and  $v$ 's layout is not .

**zero cost (Z):** as in case of "eligible", but with  $b^-(v) = 0$ .

**deferred (D):**  $\pi(v) = \perp$  and  $v \notin E \cup Z$ , i. e., either  $\mathcal{B}^+(v)$  and  $\mathcal{F}^-(v)$  violate the Nesting Property,  $\mathcal{L}(v)$  violates the Elimination Property, or  $b^-(v) > \lambda_{\max}(v)$ .

Note that for every vertex  $v$  that is not yet contained in  $\pi$ , the value of  $b^-(v)$  equals its indegree  $d^-(v)$  minus the number of incident incoming arcs whose tail is already contained in  $\pi$ . Likewise, the value of  $f^+(v)$  can be obtained from  $v$ 's outdegree  $d^+(v)$  minus the number of incident outgoing arcs whose head is already contained in  $\pi$ . As we consider simple graphs by [Assumption 6.1](#), the algorithm only has to keep track of the number of vertices in  $N^-(v)$  and  $N^+(v)$ , respectively, that are already contained in  $\pi$ . Alternatively, it can store the values  $b^-(v)$  and  $f^+(v)$  explicitly by initializing them with  $b^-(v) = d^-(v)$  and  $f^+(v) = d^+(v)$  and decrementing them whenever a neighbor of  $v$  has been processed. This also already suffices to check whether  $v$ 's layout would be of eliminable type  if  $v$  extended  $\pi$ .

To assess the linear ordering's compliance with the Nesting Property, the algorithm maintains an equilibrium or *balance*  $\mathcal{E}(v)$  for each vertex  $v$  that is initialized with zero. Whenever  $\pi(v) = \perp$  and a vertex from  $N^+(v)$  extends  $\pi$ ,  $\mathcal{E}(v)$  is decremented to indicate that there is an outgoing backward arc of  $v$  whose nesting incoming forward arc is still missing. Consequently,  $\mathcal{E}(v)$  is incremented if a vertex from  $N^-(v)$  extends  $\pi$  and  $\mathcal{E}(v)$  was negative before. Otherwise, the balance remains unchanged. Note that an incoming forward arc of  $v$  cannot be the nesting forward arc of backward arc whose head has a greater position than this forward arc's tail. Thus,  $v$  may extend  $\pi$  only if  $\mathcal{E}(v) = 0$  at this

point of time. If  $\pi$  already contains  $v$ ,  $\mathcal{E}(v)$  may be incremented for each vertex in  $N^+(v)$  that extends  $\pi$  and decremented accordingly for each vertex in  $N^-(v)$ . Then, a negative value of  $\mathcal{E}(v)$  always indicates an unresolvable violation of the Nesting Property at  $v$ .

*ExtendLO*( $G, \pi, t, E, Z, D$ ) returns an extension of  $\pi$  if and only if the number of backward arcs induced by the extension is strictly less than  $t$ . Otherwise, it indicates that no such extension is possible and returns  $\perp$ . To keep the algorithms' listing concise, we assume that the values  $b^-(v)$ ,  $f^+(v)$ , as well as  $\mathcal{E}(v)$  are stored within the data structure representing  $\pi$  for each vertex  $v \in V$ . Furthermore, the maintenance of the sets  $E$ ,  $Z$ , and  $D$  obviates the need to represent  $\lambda$  or  $\lambda_{\max}$  explicitly.

The algorithm proceeds as follows: First, it checks whether the number of backward arcs induced by the incomplete linear ordering  $\pi$  already meets or exceeds  $t$  and, if positive, discards  $\pi$  and returns  $\perp$  in [line 2](#). Otherwise,  $\pi$  is extended by all vertices classified as “zero cost” in [lines 4–9](#). For every such vertex  $v$  holds that  $\pi$  induces no incoming backward arcs, i. e., all neighbors having an arc to  $v$  must already be contained in  $\pi$ .  $v$  is inserted at the position beyond the last within  $\pi$ , such that  $\pi(v)$  is maximum among all vertices that  $\pi$  is defined for (cf. [line 6](#)). As  $v$  is contained in  $\pi$  now, the classification of all unprocessed neighbors  $u$  of  $v$  may require an update, which is handled by the subroutine *Reclassify*( $G, u, v, \pi, E, Z, D$ ). It will be reviewed later.

After all zero cost vertices have been processed, *ExtendLO* checks whether  $\pi$  is already complete in [line 10](#). If so,  $|\pi| < t$  and the routine returns it. Otherwise, as  $Z = \emptyset$  now,  $\pi$  must be extended by an “eligible” vertex, which also implies that this vertex has at least one incident incoming backward arc. Thus, neither may  $E$  be empty nor may the number of backward arcs already induced by  $\pi$  plus the minimum of the incoming backward arcs of a vertex in  $E$  equal or exceed the currently best value  $t$ . Observe that if  $|\pi| + b_{\pi}^-(v) \geq t$  for all  $v \in E$ , then  $\pi$  cannot be extended such that the number of induced backward arcs is strictly less than  $t$ . Therefore, *ExtendLO* aborts in these cases and returns  $\perp$  in [line 11](#).

In [line 12](#), the algorithm chooses and removes one vertex  $v$  from  $E$  such that  $t$  could still be undercut. Recall that  $b_{\pi}^-(v)$  is the number of incoming backward arcs of  $v$  if  $v$  would extend  $\pi$  in the next step. Trivially, there are two possibilities for  $v$  in the optimal linear ordering that is to be constructed (provided that it exists): Either  $v$  has exactly  $b^-(v)$  incoming backward arcs, i. e.,  $\lambda(v) = b^-(v)$ , or strictly less than  $b^-(v)$ , i. e.,  $\lambda(v) < b^-(v)$ . To cover the former case, the algorithm appends  $v$  to  $\pi$  in [line 13](#). In contrast to the processing of zero cost vertices,  $\pi$  is not modified, but the resulting linear ordering is stored as  $\pi'$  instead and the sets  $E, Z, D$  are copied to  $E', Z', D'$  in

---

**Algorithm 6.4** Extend an incomplete linear ordering optimally.

---

**Require:** simple graph  $G$ , incomplete linear ordering  $\pi$ , currently best result  $t$ , vertex sets  $E$  (eligible),  $Z$  (zero cost),  $D$  (deferred)

**Return:** a completion of  $\pi$  of size less than  $t$ , if possible, otherwise  $\perp$

```

1: procedure ExtendLO( $G, \pi, t, E, Z, D$ )
2:   if  $|\pi| \geq t$  then return  $\perp$ 
3:    $l \leftarrow |\{v \in V \mid \pi(v) \neq \perp\}|$ 
4:   while  $Z \neq \emptyset$  do
5:     remove one vertex  $v$  from  $Z$ 
6:      $\pi \leftarrow \text{insert}(\pi, v, l)$ 
7:      $l \leftarrow l + 1$ 
8:     for all neighbors  $u$  of  $v$  do
9:        $E, Z, D \leftarrow \text{Reclassify}(G, u, v, \pi, E, Z, D)$ 
10:    if  $l = n$  then return  $\pi$ 
11:    else if  $E = \emptyset$  or  $\min_{v \in E} b_{\pi}^{-}(v) \geq t - |\pi|$  then return  $\perp$ 
12:    remove one vertex  $v$  from  $E$  with  $|\pi| + b^{-}(v) < t$  ▷ branch on  $\lambda(v)$ 
13:     $\pi' \leftarrow \text{insert}(\pi, v, l)$ 
14:     $E' \leftarrow E; Z' \leftarrow Z; D' \leftarrow D$ 
15:    for all neighbors  $u$  of  $v$  do
16:       $E', Z', D' \leftarrow \text{Reclassify}(G, u, v, \pi', E', Z', D')$ 
17:     $\pi' \leftarrow \text{ExtendLO}(G, \pi', t, E', Z', D')$ 
18:    if  $\pi' \neq \perp$  then  $t \leftarrow |\pi'|$ 
19:    add  $v$  to  $D$ 
20:     $\pi \leftarrow \text{ExtendLO}(G, \pi, t, E, Z, D)$ 
21:    if  $\pi = \perp$  then return  $\pi'$ 
22:  return  $\pi$ 

```

---

**line 14.** Similar as above, the insertion of  $v$  to the linear ordering requires an update on all neighbors of  $v$ , which is taken care of in **lines 15–16**, but now reflected in the new sets  $E', Z', D'$ . *ExtendLO* then recurses on the extended linear ordering  $\pi'$  with  $E', Z', D'$  as arguments and stores the result again in  $\pi'$  (cf. **line 17**). As the recursive call returns only a linear ordering if  $\pi'$  could be extended such that it induces less than  $t$  backward arcs, the value of  $t$  is updated in **line 18** in preparation of another recursive call. Here, the algorithm tests the second possibility for  $v$ . To this end, it defers  $v$  and stores the result

**Algorithm 6.5** Reclassify a vertex.

**Require:** graph  $G = (V, A)$ , vertices  $u, v$ , incomplete linear ordering  $\pi$ , vertex sets  $E$  (eligible),  $Z$  (zero cost),  $D$  (deferred)

**Return:**  $E, Z, D$  where  $u$  is reclassified correctly

```

1: procedure Reclassify( $G, u, v, \pi, E, Z, D$ )
2:   if  $\pi(u) \neq \perp$  then return  $E, Z, D$ 
3:   if  $(v, u) \in A$  then  $b^-(u) \leftarrow b^-(u) - 1; \mathcal{E}(u) \leftarrow \min\{0, \mathcal{E}(u) + 1\}$ 
4:   else  $f^+(u) \leftarrow f^+(u) - 1; \mathcal{E}(u) \leftarrow \mathcal{E}(u) - 1$ 
5:   if  $f^+(u) < b^-(u)$  or  $\mathcal{E}(u) \neq 0$  or  $\mathcal{L}(u) = \text{✂}$  then
6:     ensure that  $u \notin E$  and  $u \in D$ 
7:   else if  $b^-(u) = 0$  then
8:     move  $u$  to  $Z$ 
9:   else if  $(v, u) \in A$  and  $u \notin E$  then
10:    move  $u$  from  $D$  to  $E$ 
11:  return  $E, Z, D$ 


```

of the recursive call back in  $\pi$  (cf. lines 19–20). As before, the return value is different from  $\perp$  if and only if the linear ordering could be extended such that it induces less backward arcs than  $t$ . *ExtendLO* accounts for this and returns  $\pi'$  in case that the second recursive call failed, and otherwise  $\pi$ . Note that if both  $\pi = \perp$  and  $\pi' = \perp$ , the algorithm returns  $\pi' = \perp$ , which indicates that the incomplete linear ordering passed as argument could not be extended such that it undercuts  $t$ . The binary choice in the second part of the algorithm is one of two characteristics of *ExtendLO* to ensure that no partial layout is considered twice.

Let us finally consider *Reclassify*( $G, u, v, \pi, E, Z, D$ ), which is listed in Algorithm 6.5. This subroutine is called whenever the classification of a vertex  $u$  may require an update due to the insertion of its neighbor  $v$  in  $\pi$ . In case that  $u$  has already been processed, it immediately returns its arguments  $E, Z, D$ , as no reclassification of  $u$  is necessary. Otherwise, consider a vertex  $u$  such that  $(v, u) \in A$  or  $(u, v) \in A$  and  $u$  is not yet contained in  $\pi$ . Let  $b^-(u)$  and  $f^+(u)$  here denote the number of incoming backward and outgoing forward arcs of  $u$ , respectively, if  $u$  would extend the incomplete linear ordering in the next step. As defined in Section 6.1, we consider an arc  $(u, w)$  or  $(w, u)$  to be in  $\mathcal{B}^+(u)$  or  $\mathcal{F}^-(u)$  if and only if  $\pi(w) \neq \perp$ . We assume that  $b^-(u)$  and  $f^+(u)$  are initialized with  $d^-(u)$  and  $d^+(u)$ , respectively, at the beginning of the algorithm and  $u$ 's



balance with  $\mathcal{E}(u) = 0$ , as described earlier. Then, if  $(v, u) \in A$ ,  $(v, u)$  is an incoming forward arc of  $u$ . Hence in [line 3](#),  $b^-(u)$  is decremented and  $(v, u)$  can be the nesting forward arc of an outgoing backward arc at  $u$ , i. e.,  $\mathcal{E}(u)$  is incremented if it was negative before. Otherwise, if  $(u, v) \in A$ , then  $(u, v)$  is an outgoing backward arc of  $u$ , so  $f^+(u)$  and  $\mathcal{E}(u)$  are both decremented in [line 4](#).

Next, we revalidate the classification of  $u$  with respect to  $E$ ,  $Z$ , and  $D$ . If  $u$  does not meet the requirements for being “eligible” as defined above, which demand that  $b^-(u) \leq f^+(u)$ , there is a proper nesting of the arcs in  $\mathcal{B}^+(u)$  and  $\mathcal{F}^-(u)$  that complies with the Nesting Property, and  $u$ ’s layout is not of eliminable type , then  $u$  must be classified as “deferred” (cf. [lines 5–6](#)). Note that the heads of all arcs in  $\mathcal{B}^+(u)$  as well as the tails of all arcs in  $\mathcal{F}^-(u)$  are already contained in  $\pi$ . Hence, their nesting is already determined and we can check it as described earlier in this section by testing whether  $\mathcal{E}(u) = 0$ . Otherwise, if  $b^-(u) = 0$ , then  $u$  now is a “zero cost” vertex and therefore moved to  $Z$  (cf. [lines 7–8](#)). In case that  $u$  meets the requirements for being “eligible” and  $u$  is considered because of an incoming arc, i. e., because  $(v, u) \in A$ , then  $u$  is effectively also reclassified as eligible and moved from  $D$  to  $E$  if necessary in [lines 9–10](#). Thus,  $u$  can only change from “deferred” to “eligible” if  $b^-(u)$  has decreased by at least one since its last inspection. This is the second part that ensures that no partial layout is considered twice. As to the correctness, observe that if  $u$  was deferred because it did not meet all requirements of being “eligible”, then only a decrease of  $b^-(u)$  can resolve this. On the other hand, if  $u$  was “deferred” deliberately for the second recursive call in *ExtendLO*, then  $u$  may still meet the requirements of being “eligible” at the next inspection even though  $b^-(u)$  has not decreased, which is why this reclassification may only occur if a further incoming arc of  $u$  has been classified as forward, i. e., if  $b^-(u)$  certainly has decreased.

Together with a proper initialization of  $E$ ,  $Z$ , and  $D$ , we can use *ExtendLO* to construct an alternative exact algorithm that performs asymptotically at least as good as *ExactLOSimple*:

---

### Theorem 6.3

There is an  $\mathcal{O}(n \cdot m)$ -space algorithm that constructs an optimal linear ordering  $\pi^*$  of a graph  $G$  with maximum vertex degree  $\Delta_G$  in time  $\mathcal{O}(m \cdot \min\{(\frac{1}{2}\sqrt{k^2 - 1})^n, \sqrt{2}^{(-n)} \cdot n!\})$  if  $G$  is  $k$ -regular and  $k$  is odd, and  $\mathcal{O}(m \cdot \min\{(\frac{m}{n} + 1)^n, (\lfloor \frac{\Delta_G}{2} \rfloor + 1)^n, \sqrt{2}^{(-n)} \cdot n!\})$  otherwise.

---

---

**Algorithm 6.6** A more sophisticated exact algorithm.

---

**Require:** simple graph  $G$

**Return:** the returned linear ordering is optimal

```

1: procedure ExactLOIntegrated( $G$ )
2:    $\pi \leftarrow$  linear ordering of  $G$  obtained by some heuristic
3:    $E \leftarrow \emptyset; Z \leftarrow \emptyset; D \leftarrow \emptyset$ 
4:   for all  $v \in V$  do
5:     if  $d^+(v) \geq d^-(v)$  then  $E \leftarrow E \cup \{v\}$  else  $D \leftarrow D \cup \{v\}$ 
6:    $\pi' \leftarrow \text{ExtendLO}(G, \perp^V, |\pi|, E, Z, D)$ 
7:   if  $\pi' \neq \perp$  then
8:     return  $\pi'$ 
9:   return  $\pi$ 

```

---

*Proof.* Consider algorithm *ExactLOIntegrated*( $G$ ) as given in [Algorithm 6.6](#). In order to obtain a linear ordering to start with, it runs a heuristic on  $G$  in [line 2](#). Afterwards, the sets  $E$ ,  $Z$ , and  $D$  are constructed in preparation for the call to *ExtendLO* with an “empty” linear ordering  $\perp^V$ , i. e., one that does not define a LO position for any vertex. For the classification of each vertex  $v \in V$  in [lines 4–5](#), the algorithm needs to consider only its out- and indegree: Recall that a vertex is “eligible” if it can extend the incomplete linear ordering in the next step such that its layout is in accordance with the Nesting Property and the Eliminateable Layouts Property. As the incomplete linear ordering is initially empty, all outgoing arcs of  $v$  would become outgoing forward arcs and all incoming arcs would become incoming backward arcs if  $v$  is appended in the next step. The algorithm therefore classifies  $v$  as “eligible” if and only if  $d^-(v) \leq d^+(v)$ . Note that an eliminateable layout is impossible at this stage because  $f^-(v) = 0$ . Moreover,  $Z = \emptyset$  because  $G$  is strongly connected by [Assumption 3.2](#).

Next, *ExactLOIntegrated* calls *ExtendLO*( $G, \perp^V, |\pi|, E, Z, D$ ). As mentioned above, the incomplete linear ordering here is  $\perp^V$ . We also assume that  $b^-$ ,  $f^+$ , and  $\mathcal{E}$  are initialized accordingly as described earlier, i. e., with  $b^-(v) = d^-(v)$ ,  $f^+(v) = d^+(v)$ , and  $\mathcal{E}(v) = 0$  for each vertex  $v \in V$ . The parameter  $t$  is set to  $|\pi|$ , because we are only interested in linear orderings that are strictly better than the solution returned by the heuristic algorithm. Thus, if this call returns a linear ordering  $\pi'$  such that  $\pi' \neq \perp$ , then  $|\pi'| < |\pi|$  and, as *ExtendLO* tested all contemplable partial layouts,  $\pi'$  is an optimal linear ordering of  $G$ . Subsequently, the algorithm returns  $\pi'$  in [line 8](#). Otherwise, the

solution found by the heuristic already is an optimal solution and is hence returned in [line 9](#).

Even though *ExtendLO* enumerates the partial layouts only implicitly and in a different order, it does not consider more partial layouts than *ExactLOSimple* due to the vertex classifications and the argumentation at the beginning of this section. For every vertex  $v$  that is appended to the incomplete linear ordering, the classification of its neighbors are updated in *Reclassify*, which can be done in constant time per neighbor if we assume that the values necessary to compute its layout and balance are stored explicitly. Hence, the effort for all reclassifications during one completion of a linear layout is in  $\mathcal{O}(\sum_{v \in V} d(v)) = \mathcal{O}(m)$ . Furthermore, each recursive call of *ExtendLO* needs to copy the sets  $E$ ,  $Z$ , and  $D$  once be able to branch on  $\lambda(v)$  for some vertex  $v$ , which requires  $\mathcal{O}(n)$  steps each time.

As in case of *ExactLOSimple*, the number of complete linear orderings does not exceed  $(\min\{\frac{m}{n}, \lfloor \frac{\Delta_G}{2} \rfloor\} + 1)^n$  for general graphs and  $(\frac{1}{2}\sqrt{k^2 - 1})^n$  for  $k$ -regular graphs with  $k$  odd. Furthermore, *ExactLOIntegrated* never constructs the same linear ordering twice, but on the contrary: Suppose that *ExactLOIntegrated* obtains a complete or incomplete linear ordering  $\pi$  in the course of the algorithm. Then, for every pair of consecutive vertices  $u, v$  with respect to  $\pi$ , *ExactLOIntegrated* does not construct the linear ordering that emerges when  $u$  takes the LO position of  $v$  and vice-versa. If either  $(u, v)$  or  $(v, u)$  exists, then the local ordering of  $u$  and  $v$  in an optimal linear ordering is uniquely determined by the fact that the arc cannot be backward in consequence of the Nesting Property. Otherwise, if no such arc exists, then both linear orderings induce the same (partial) layout on  $u$  and  $v$  and *ExactLOIntegrated* enumerates each only at most once. Consequently, *ExactLOIntegrated* constructs at most

$$\frac{n!}{2^{\frac{n}{2}}} = \sqrt{2}^{(-n)} \cdot n!$$

linear orderings.

In summary, *ExactLOIntegrated* traverses a recursion tree having  $\mathcal{O}(\min\{(\frac{1}{2}\sqrt{k^2 - 1})^n, \sqrt{2}^{(-n)} \cdot n!\})$  leaves if  $G$  is  $k$ -regular and  $\mathcal{O}(\min\{(\frac{m}{n} + 1)^n, (\lfloor \frac{\Delta_G}{2} \rfloor + 1)^n, \sqrt{2}^{(-n)} \cdot n!\})$  in general, and in both cases asymptotically the same number of inner nodes. With each leaf being associated with a cost of  $\mathcal{O}(m)$  for the reclassifications and each inner node with a cost of  $\mathcal{O}(n) \subseteq \mathcal{O}(m)$  for the set copying, the time complexity of *ExactLOIntegrated* follows as stated in the theorem.

Finally, let us address the algorithm's space complexity. For each recursive call, the space required to store the incomplete linear ordering as well as  $t$ ,  $E$ ,  $Z$ , and  $D$  is in

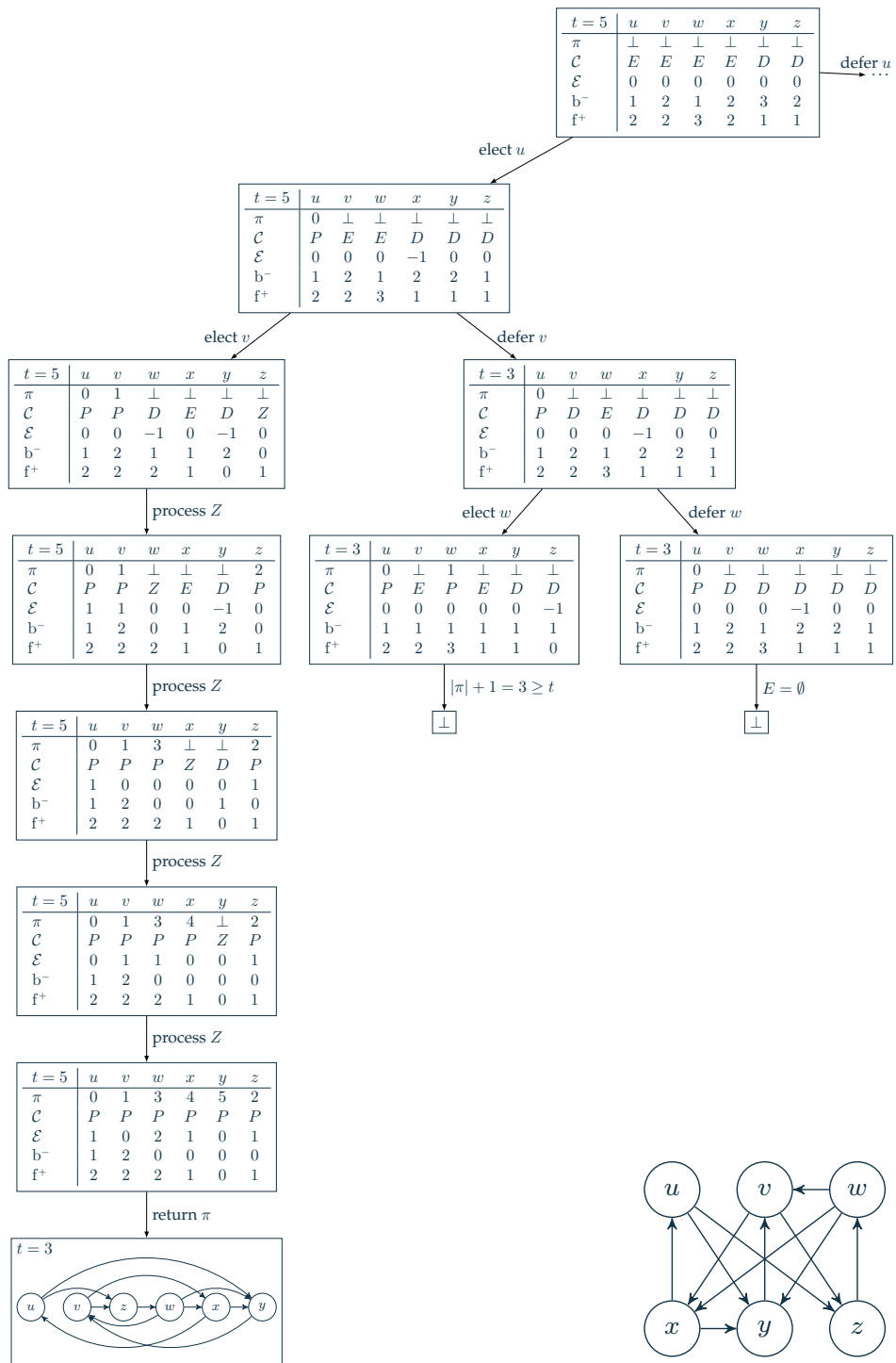


Figure 6.1: *ExactLOIntegrated*'s proceeding on the graph depicted in the lower right corner (part one, continued in Figure 6.2).

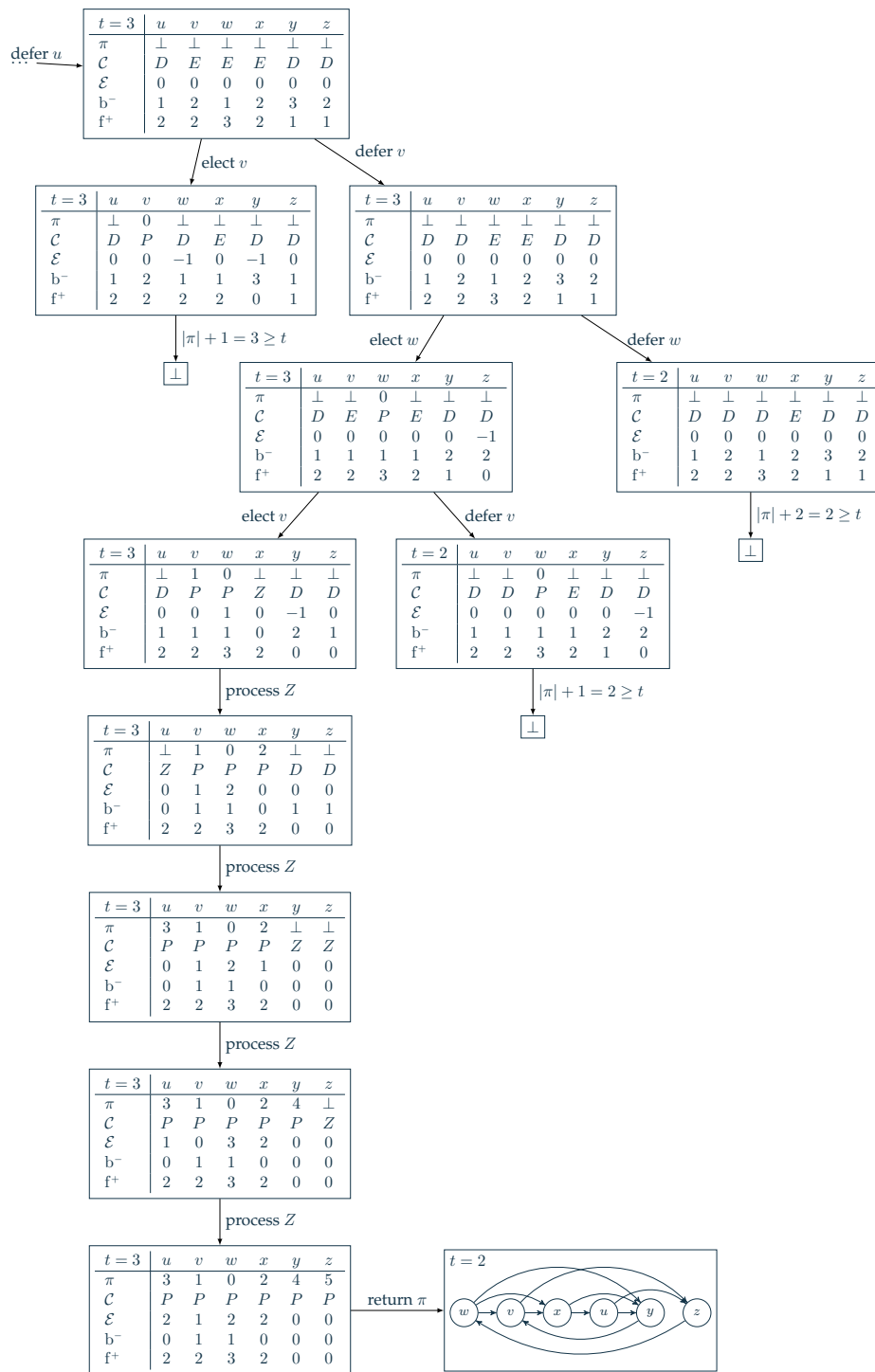


Figure 6.2: Continuation of Figure 6.1.

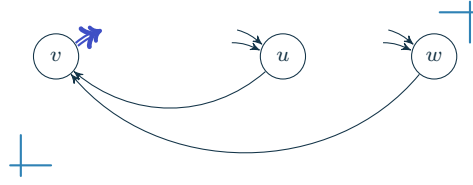


Figure 6.3: An “eligible” vertex  $v$  where the tails of its incoming arcs have degree three.

$\mathcal{O}(n)$ . The maximum recursion depth of *ExtendLO* is given by the number of times that every vertex may change its classification, which is in  $\mathcal{O}(d(v))$  per vertex  $v$  and hence in  $\mathcal{O}(\sum_{v \in V} d(v)) = \mathcal{O}(m)$  in total. This yields an overall space complexity of  $\mathcal{O}(n \cdot m)$ .  $\square$

Figure 6.1 and Figure 6.2 exemplarily visualize the proceeding of *ExactLOIntegrated* on a small input graph. The variables maintained by the algorithm are represented in tabular form, where  $\mathcal{C}$  indicates the classification of each vertex as  $P$  (processed),  $E$  (eligible),  $Z$  (zero cost), or  $D$  (deferred). The initial upper bound  $t$  was set to the trivial upper bound of  $\lfloor \frac{m}{2} \rfloor = 5$ . Observe that  $t$  declines in the course of the algorithm down to  $t = 2$  and that the recursive call corresponding to the election of a vertex always precedes that for its deferral.

## 6.4 Fine-Tuning

In order to keep the description of the algorithm *ExactLOIntegrated* concise, some possible improvements have been omitted in the previous sections, as they do not influence the estimation of the worst-case running time except in that the effort comprised in the polynomial factor may increase. Nevertheless, they may speed up the observed execution time for some input graphs. The additions suggested in the following, however, are neither exhaustive nor do they actually guarantee a decrease in running time. On the contrary, it may be the case that the extra checks cost more than they save. Hence, their employment must be carefully considered.

An addition that barely increases the time effort consists in checking the adherence to the Nesting Property not only for vertices that are about to be inserted in the incomplete linear ordering, but also for those that are already contained. Effectively, this implies that the proper nesting of the arcs in  $\mathcal{B}^-(v)$  and  $\mathcal{F}^+(v)$  of an already processed vertex  $v$  is verified if one of its neighbors is reclassified. This can be accomplished easily in the helper routine *Reclassify*, which must then have the possibility to report a violation with the result that the current incomplete linear ordering is rejected.

Apart from the Nesting Property and the Eliminateable Layouts Property, also other properties of optimal linear orderings may be considered during the construction of a linear ordering, such as the Path Property, the Blocking Vertices Property, the Multipath Property, or the Multipath Blocking Vertices Property. In these cases, however, the additional effort required for the check should be balanced against the effort of enumerating perhaps only few more discardable linear orderings. As the incomplete linear ordering is constructed “from left to right”, i. e., with increasing LO positions, the application of the Suffix Fusion Property suggests itself. For the examination of the fused graphs along with the fused linear orderings, however, the benefits should again be weighed carefully against the additional time effort.

In case that the graph contains vertices of degree three, it is already possible to reduce the number of “eligible” vertices during the initialization phase in *ExactLOIntegrated*: Such a degree-three vertex may have an outgoing(!) backward only if it also has two incoming arcs, otherwise, its layout would be eliminable. Hence, a vertex  $v$  can be “eligible” in this phase only if all tails of its incoming arcs either do not have vertex degree three or also have two incoming arcs, as depicted in [Figure 6.3](#). This check may also be reapplied if the reclassification of  $v$  is considered in *Reclassify*.

## 6.5 Runtime Comparison for Sparse Graphs

With  $\mathcal{O}(m \cdot \min\{(\frac{m}{n} + 1)^n, (\lfloor \frac{\Delta_G}{2} \rfloor + 1)^n, \sqrt{2}^{(-n)} \cdot n!\})$ , the runtime analysis of *ExactLOIntegrated* in [Theorem 6.3](#) shows a strong dependency on the input graph’s density. This is in sharp contrast to almost all of the previously known exact algorithms for solving the LINEAR ORDERING problem: For the enumeration of all vertex permutations, approaches based on dynamic programming, as well as those employing linear program solvers, the number of vertices  $n$  of the input graph has great influence on the running time, whereas the graph’s density is sometimes even completely ignored. Only the brute-force algorithm that tests all  $2^m$  possible subsets of arcs for whether they are a feasible feedback arc set takes advantage if the input graph has few arcs. This fact could be taken as evidence that there has not been placed great importance in the development of exact algorithms that are tailored to sparse graphs in the past.

In this chapter, we made a first attempt to close this gap. The comparison of running times provided in [Table 6.1](#) shows that the newly developed algorithm outperforms the brute-force subset testing approach easily. Moreover, for graphs with a maximum vertex degree of at most seven, it is also able to keep up with or even outrace the

Table 6.1: Running Times for Sparse Graphs.

	<i>ExactLOIntegrated</i>		<b>subset testing</b>
$\Delta_G = k$		<i>k</i> -regular	
$k = 3$ :	$\mathcal{O}^*(2^n)$	$\mathcal{O}^*(\sqrt{2}^n) \subseteq \mathcal{O}^*(1.4143^n)$	$\mathcal{O}^*(2^{3n/2}) \subseteq \mathcal{O}^*(2.8285^n)$
$k = 4$ :	$\mathcal{O}^*(3^n)$	$\mathcal{O}^*(3^n)$	$\mathcal{O}^*(4^n)$
$k = 5$ :	$\mathcal{O}^*(3^n)$	$\mathcal{O}^*(\sqrt{6}^n) \subseteq \mathcal{O}^*(2.4495^n)$	$\mathcal{O}^*(2^{5n/2}) \subseteq \mathcal{O}^*(5.6569^n)$
$k = 6$ :	$\mathcal{O}^*(4^n)$	$\mathcal{O}^*(4^n)$	$\mathcal{O}^*(8^n)$
$k = 7$ :	$\mathcal{O}^*(4^n)$	$\mathcal{O}^*((2\sqrt{3})^n) \subseteq \mathcal{O}^*(3.4642^n)$	$\mathcal{O}^*(2^{7n/2}) \subseteq \mathcal{O}^*(11.31134^n)$
$k = 8$ :	$\mathcal{O}^*(5^n)$	$\mathcal{O}^*(5^n)$	$\mathcal{O}^*(16^n)$
$k = 9$ :	$\mathcal{O}^*(5^n)$	$\mathcal{O}^*((2\sqrt{5})^n) \subseteq \mathcal{O}^*(4.4722^n)$	$\mathcal{O}^*(2^{9n/2}) \subseteq \mathcal{O}^*(22.6275^n)$

dynamic programming algorithm, which has a running time of  $\mathcal{O}^*(4^{n+o(n)})$  if the space complexity is to be kept polynomial. Last, but not least, *ExactLOIntegrated* is easy to implement and requires only standard data structures.



It sounds at first like a paradox that in practice, algorithms with (good) performance guarantees are often inferior to theoretically weaker or not-guaranteeing-anything heuristics. Indeed, a series of papers have shown that the situation is no different for FEEDBACK ARC SET and LINEAR ORDERING: As has already been discussed in [Section 2.4.1](#), in particular algorithms that mimic an “insertion sort” of the input graph’s vertices have turned out to be very effective—both standalone and as postprocessing after other heuristics or meta-heuristics [[CW09](#), [Han10](#), [BH11](#), [MR11](#)]. By contrast, a 3-approximation algorithm for tournaments, e. g., was shown to be significantly weaker in experiments [[CW09](#), [Han10](#)].

The study conducted by Martí and Reinelt [[MR11](#)] for the ACYCLIC SUBGRAPH problem on very dense and weighted graphs also includes an evaluation of the performance of (meta-)heuristics in comparison to the optimum solution—wherever this value was known. Their findings suggest that the candidates perform surprisingly well on the used test instances with very small relative errors. For sparse and unweighted graphs, however, no similar comparison of solutions found by non-exact algorithms to the optimum has been published so far.

In the following, we report on an experimental evaluation of the algorithms introduced in this thesis with a focus on sparse and regular graphs. Besides the assessment of the solution quality, i. e., the size of the produced feedback arc set, we also consider the execution times of the individual candidates. Due to the relatively “efficient” exact algorithm presented in [Chapter 6](#) as well as a specific selection of test instances, we can use the optimum solution value in many cases to gauge the candidates’ performances.

With the bounds obtained in [Section 4.3](#) and [Chapter 5](#), the set of tested algorithms also contain candidates that offer performance guarantees—albeit not with respect to the optimum solution. Nevertheless, their behavior will especially be of interest.

## 7.1 The Algorithm Test Suite

We start with a short motivation and overview of the tested algorithms. Afterwards, we introduce different sets of input instances that were used to evaluate the candidates both with regard to their performance and their running time. The section concludes with a description of the implementations in source code as well as the technical setup and the execution environment.

### 7.1.1 Algorithms

In [Chapter 4](#), we proved a collection of properties that any optimal linear ordering has to respect and showed how a better solution can be obtained in case that one of these properties is violated. This finally led to an algorithm called *PsiOpt* in [Section 4.9.2](#), which constructs a linear ordering that simultaneously adheres to the Nesting Property, the Path Property, the Blocking Vertices Property, the Multipath Property, the Multipath Blocking Vertices Property, and the Eliminateable Layouts Property. As the Multipath Blocking Vertices Property implies all other properties except for the Eliminateable Layouts Property, an alternative implementation of *PsiOpt* could only enforce these two properties, while preserving the same theoretical solution quality as *PsiOpt* as well as the same asymptotic running time. With the property-enforcing routine for the Multipath Blocking Vertices Property being relatively costly in comparison to those for the Nesting Property and the Path Property (cf. [Section 4.9.2](#)), this raises an interesting question of which variant of *PsiOpt* consumes more computation time in practice.

One further property that has been introduced in [Chapter 4](#), the Fusion Property, was described as a kind of meta-property and is only applicable in combination with another algorithm. Furthermore, only weaker versions, such as the Prefix Fusion Property, the Suffix Fusion Property, and the Greedy Fusion Property, can be established in polynomial time, provided that the combined algorithm is also efficient. We chained two of these weaker versions with *PsiOpt* for this evaluation, also with the intention of comparing the performances and running times to those of the “standalone” *PsiOpt* algorithm.

Ensuring only the Nesting Property by means of *EstablishNesting* as described in [Section 4.3](#) yields a 1-opt algorithm that resembles *InsertionSort* on the set of vertices and is closely related to heuristics like *Moves* [[CW09](#)] and *Sifting* [[Han10](#), [BH11](#)] or the approach taken by Chanas and Kobylański [[CK96](#)]. In contrast to *EstablishNesting*, *Sifting* may modify the linear ordering also without strictly improving it and terminates

as soon as the size of the induced backward arc set has not changed during one iteration. With respect to this, *EstablishNesting* behaves similar to a stable sorting algorithm and continues until the linear ordering itself remains without modification. The algorithm by Chanas and Kobylański starts in each iteration with an “empty” linear ordering and inserts the vertices one by one at their locally best position. The vertices are processed in the order specified by the linear ordering obtained in the previous iteration or its reverse until again the size of the induced set of backward arcs has not decreased during one run. *EstablishNesting* can hence be expected to produce similar results as these competitors.

Another sorting-like heuristic was proposed by Eades *et al.* [ELS93], which also turned out to be a good “preprocessor” for *InsertionSort*-like heuristics [CW09]. What is more, it additionally offers a performance guarantee in terms of an absolute upper bound on the cardinality of the induced set of backward arcs. With a linear running time on unweighted graphs, it is also well suited to produce good results very quickly.

We chose the algorithm by Eades *et al.* as well as *EstablishNesting* as a yardstick to assess both the quality and the running time of the algorithms developed in this thesis. Finally, we introduced an exact algorithm in Chapter 6, which is suited to yield optimum values for small enough graphs and whose running time—although exponential in theory—is to be evaluated in practice.

It would have been interesting to also compare the running time of *ExactLOIntegrated* to the exact algorithm based on dynamic programming [RS07, FK10], which runs in only  $\mathcal{O}^*(2^n)$  time, but requires exponential space. Unfortunately, the memory usage of the implementation (cf. Section 7.1.3) on input instances with as few as 30 vertices already exceeded our memory limit of 120GB, which made this part of the study impracticable.

The test suite includes the following algorithms:

**Nest:** *EstablishNesting* as described in Section 4.3.

**ELS:** The linear-time algorithm by Eades *et al.* [ELS93].

**Psi:** The  $\Psi_{\text{opt}}$ -algorithms *PsiOpt* and *PsiOptCubic* for cubic input graphs, as introduced in Section 4.9.2.

**PsiAlt:** *Cascade* with *EnforceMultiPathsNoBlocking* and *EliminateLayouts*, which as an alternative to *PsiOpt* also yields a  $\Psi_{\text{opt}}$  linear ordering (cf. Section 4.9.2).

**Fusion:** The algorithm obtained by applying *Cascade* on an enforcement of both the Prefix Fusion Property and the Greedy Fusion Property, each combined with *PsiOpt*.

**ExIn:** The exact algorithm *ExactLOIntegrated* as described in Section 6.3 with **ELS** as the heuristic that provides the initial upper bound.

From the set of possible tuning options described in [Section 6.4](#) for *ExIn*, we implemented the additional check of the nesting balance for already processed vertices as well as the extension in the assessment of a vertex’s eligibility if the tails of its incoming arcs have degree three.

The Suffix Fusion Property has been omitted in the test suite for reasons of symmetry with the Prefix Fusion Property. We also decided against the inclusion of a polynomial-time variant of the Reduction Property or the Independent Set Reduction Property due to their close relationship with the Blocking Vertices Property and the Multipath Blocking Vertices Property (cf. [Section 4.10.3](#)). In a pre-test, the enforcement of the additional properties introduced specifically for the proof of the bound for subquartic graphs in [Chapter 5](#), i. e., the Alternative Forward Paths Property and the Tail On Forward Path Property, yielded very few to no improvements in comparison with the “standard”  $\Psi_{\text{opt}}$ -algorithms. To keep the number of algorithms and their variants manageable, the results for these versions are not listed here separately. The same applies for their generalization, the One-Arc Stability Property, and further polynomial-time variants of the Arc Stability Property.

With the exception of *ELS* and *ExIn*, all algorithms can also be used as postprocessing routines and therefore be instantiated with an input linear ordering. To also assess the variability introduced therewith, these candidates are tested with the following initializers:

- Any**: an arbitrary but fixed linear ordering,
- ELS**: the linear ordering produced by *ELS*,
- Random**: a random linear ordering.

The input linear ordering for an algorithm is hence determined by the result produced by the respective initializer. For the initializer **Any**, we used the linear ordering corresponding to the order the vertices were given in the definition of the input graph. In case of **Random**, the initial linear ordering was obtained in each run from a random permutation of the list of vertices. We refer to the combination of an algorithm **A** with an initializer **I** as **A | I**. *Nest | ELS*, e. g., hence means that the algorithm *Nest* was started from an initial linear ordering obtained by *ELS*.

### 7.1.2 Input Instances

The input instances for the evaluation can be subdivided into three categories:

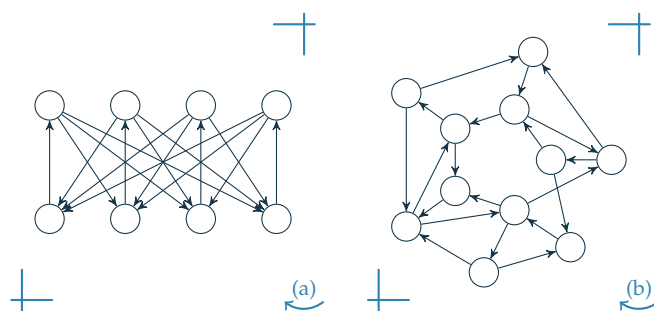


Figure 7.1: A 4-fence (a) and a Möbius ladder (b).

**Cubic, Quartic, Quintic** As the focus of this evaluation lies especially on sparse and regular graphs, our first three sets of instances comprise strongly connected, regular graphs of degrees three, four, and five, respectively, i. e., cubic, quartic, and quintic graphs. In view of the exact algorithm, the number of vertices of the instances was kept small and ranges between 20 and 200 in intervals of 10 in each case, with 100 graphs per size.

The graphs were constructed as follows: First, a random undirected  $k$ -regular graph was generated using the algorithm by Steger and Wormald [SW99] for  $k \in \{3, 4, 5\}$ . Afterwards, each graph was oriented such that it is strongly connected by producing a randomized ear decomposition [Sch13] for 50% of the instances and a randomized decomposition into cycles and ears for the remaining 50%. Due to the sparsity of the graphs, a random orientation obtained by independently choosing the direction of each arc with a certain probability and filtering those graphs which did not turn out strongly connected had a reject rate of close to 100% and was hence impracticable. We also decided against a random orientation and later decomposition into strongly connected components due to the varying and expectably small sizes of the single SCCs.

**Facet** To also benchmark graphs with more vertices, another set of input instances was compiled with the number of vertices ranging between approximately 100 and 1000 in intervals of 100. These graphs were generated as two-clique-sums of so-called  $k$ -fences and Möbius ladders. In short, a  $k$ -fence is a bipartite tournament on  $k + k$  vertices such that the vertices in the first partition have exactly one incoming arc and the vertices in the second partition have exactly one outgoing arc, whereas a *Möbius ladder* is obtained by a specific cyclic conjunction of an odd number of cycles of length three and four such that the resulting graph can be embedded on a Möbius strip [MR11]. Figure 7.1 depicts an example for each of them. The graphs of both classes produce facet-defining inequalities

of the acyclic subgraph polytope [MR11] (cf. Section 2.1.4). Furthermore, the value of an optimal solution to the LINEAR ORDERING problem is known for these graphs, and, if two such graphs are combined into a larger graph  $H$  by means of a two-clique-sum, then also  $H$  yields a facet-defining inequality and the value of an optimal solution of  $H$  can be obtained from those of the two combined graphs [BFM94, NP95].

We utilize this fact to generate sparse test instances of size between 100 and 1000. To this end, all  $k$ -fences with  $k \in \{3, \dots, 30\}$  as well as 2640 ladders consisting each of three to eleven 3- and 4-cycles were constructed. Then, for each graph with target size  $n$ , we drew graphs randomly from the set of all graphs producing facet-defining inequalities and attached them to one another until no graph could be added without exceeding  $n$ . The set contains 50 instances per target size.

**LOLIB** For an evaluation of our algorithms in comparison to other heuristic approaches and to also investigate their performance on dense graphs, we included the LOLIB<sup>1</sup> library [MR11], a collection of graphs that are in general very dense and nearly tournaments. For part of the graphs contained therein, the value of an optimum solution is known, whereas for others, only the best value found so far is available. The library consists of eight sets of graphs, some compiled from real-world data, such as input-output matrices, others generated according to some random distribution. In contrast to the other test instances, the graphs contained in the LOLIB library have integer arc weights, which have been translated to their corresponding unweighted multigraphs, i. e., an arc with weight  $k$  was replaced by  $k$  parallel unweighted arcs.

### 7.1.3 Technical Setup

All algorithms as well as the data structures for graphs and linear orderings have been implemented in C++ and as described in the respective sections of this thesis. Basic data structures such as lists, sets, maps, etc. were taken as provided by the standard template library (STL). The algorithms to enforce the Multipath Property and the Multipath Blocking Vertices Property, which employ a network flow algorithm as a subroutine, make use of the implementation available as part of the COIN-OR LEMON graph library<sup>2</sup>.

The generators for the input instances **Cubic**, **Quartic**, **Quintic**, and **Facet** were realized in Python on the basis of the NetworkX software package<sup>3</sup>.

<sup>1</sup><http://www.optsi.com.es/lolib/>, last accessed on July 5, 2017.

<sup>2</sup><http://lemon.cs.elte.hu/trac/lemon>, last accessed on July 5, 2017.

<sup>3</sup><https://networkx.github.io/>, last accessed on July 5, 2017.

Both the graphs and the results produced by the implementation were stored in a MySQL database. The algorithms were run on machines with an Intel® Xeon® E5-2650 v2 processor with 2.60GHz and 128GB memory. Each test scenario, i. e., the execution of each algorithm on each graph, was repeated at least five times and the median was taken to obtain unbiased running times. In case of *ExIn* on input instances from *Cubic*, *Quartic*, or *Quintic* as well as in case of *Fusion* on input instances from *Facet*, we additionally set a time limit of 4 hours.

The implementation as well as the generated input graphs used in the evaluation are available on a supplementary web page<sup>1</sup>. The statistical evaluation was conducted using the statistics module shipped with Python.

#### 7.1.4 Evaluation

For the evaluation of the performances and running times, we gathered a number of key figures. Let  $\mathcal{B}$  and  $\mathcal{F}$  denote the set of backward and forward arcs, respectively, that are induced by the linear ordering  $\pi$  constructed by an algorithm for a given graph  $G$ . The assessed values of  $\pi$  are the cardinalities of the multisets  $\mathcal{B}$  and  $\mathcal{F}$ , i. e.,  $|\mathcal{B}|$  and  $|\mathcal{F}|$ . We denote by  $\tau$  again the value of the optimum solution to the LINEAR ORDERING problem. For the ACYCLIC SUBGRAPH problem, the optimum solution value, denoted by  $\bar{\tau}$  here, can then be obtained as the difference between the number of arcs  $|A| = m$  and the optimum solution to the LINEAR ORDERING problem, i. e.,  $\bar{\tau} = m - \tau$ . If these values are neither known nor could they be determined, we used the best known solution as approximative value instead. As mentioned above, all experiments were repeated between 5 and 10 times.

The figures used in the evaluation of an algorithm on a set of graphs are:

**mean  $|\mathcal{B}|$ :** the mean of  $|\mathcal{B}|$

**dev. % ( $\mathcal{B}$ ):** the mean percentage deviation from the optimum or best known solution  
as  $\frac{|\mathcal{B}| - \tau}{\tau}$

**dev. % ( $\mathcal{F}$ ):** the mean percentage deviation from the optimum or best known solution  
as  $\frac{\bar{\tau} - |\mathcal{F}|}{\bar{\tau}}$

**#hits of OPT/best:** the number of times an algorithm found the optimum or best known solution for a graph compared to the total number of graphs in the set

**time:** the mean of all graphs in the set of the median of all execution times for the algorithm on the same graph

<sup>1</sup><https://algo-rhythmics.org/sparselo>



In case of **Random** as initial linear ordering, we collected two values for the mean result and the deviations: for the first one, we used the average over all repetitions as the result of the algorithm, whereas for the second, we picked the best solution obtained during all repetitions. The value for #hits of OPT/best was determined here by considering the solutions of all repetitions. For **Any** and **ELS** as initializers, the algorithms are deterministic and hence produce the same solution in each repetition.

With regard to the deviations from the optimum or best known solutions, recall that  $\bar{\tau} = m - \tau$ . Thus,

$$\bar{\tau} - |\mathcal{F}| = m - \tau - |\mathcal{F}| = m - \tau - (m - |\mathcal{B}|) = |\mathcal{B}| - \tau.$$

Subsequently, the value of the absolute deviation is independent from whether  $\mathcal{B}$  or  $\mathcal{F}$  is considered. As  $\tau \leq \bar{\tau}$ , however, the mean percentage deviation for the set of forward arcs can be at most the mean percentage deviation for the set of backward arcs.

## 7.2 Sparse Regular Graphs

The first collection of benchmark instances consists of the sets **Cubic**, **Quartic**, and **Quintic**, which contain 3-, 4-, and 5-regular graphs, respectively. The graphs generated for these sets are (at least partly) small enough for the exact algorithm **ExIn** to finish in acceptable time. For these instances, we are hence able to compare the performance of the heuristic and approximative approaches to the optimum solution.

### 7.2.1 Selection and Configuration of Algorithms

We benchmarked all algorithms listed in [Section 7.1.1](#). For instances of **Cubic**, **Psi** refers to the specially crafted version of *PsiOpt* on cubic graphs, *PsiOptCubic*. In this case, there also exists no alternative implementation of *PsiOpt*, which is why no performance or runtime results were gathered for **PsiAlt** on the **Cubic** set.

To obtain an upper bound on the size of graphs such that **ExIn** terminates in acceptable time, we set an initial limit of 4 hours. All instances with up to 140 vertices from **Cubic** remained below this threshold. Out of 50 cubic graphs with 150 vertices, **ExIn** still terminated on 44 of them in less than 4 hours. For **Quartic** and **Quintic**, the time limit could be kept strictly only on instances with 40 vertices or less. Among the 100 quartic and 100 quintic graphs with 50 vertices, however, there were only one quartic and two quintic graphs that needed more time. We hence softened the 4 hour limit slightly to also make for a comparison against exact values for these instances.



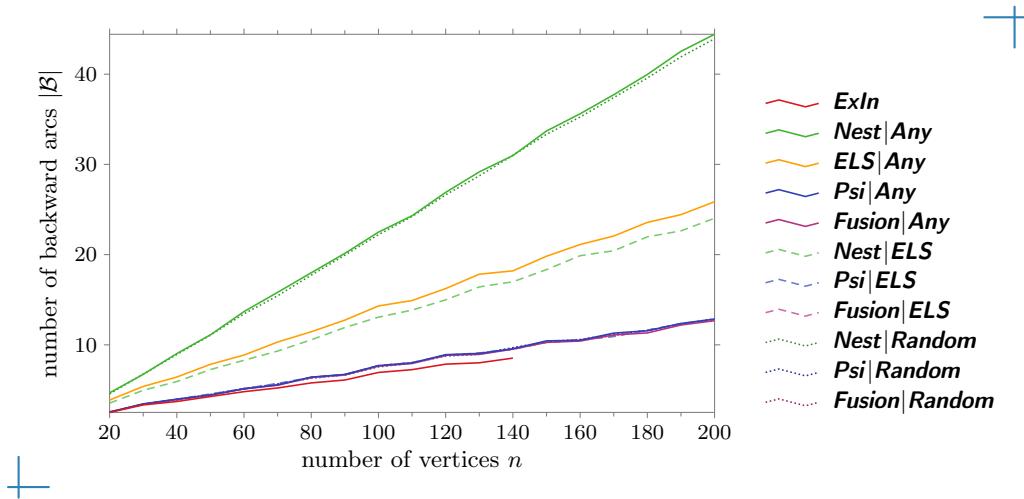


Figure 7.2: Comparison of absolute performances on **Cubic** instances.

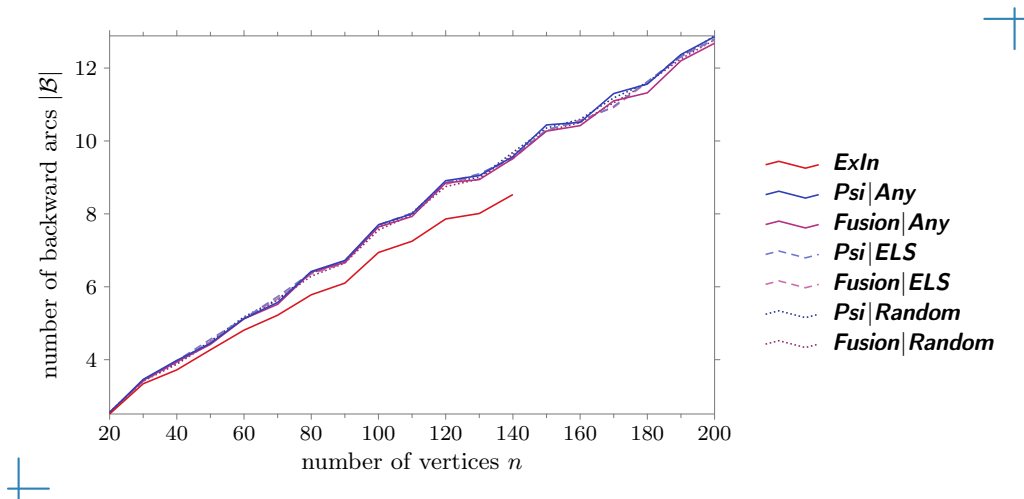


Figure 7.3: Absolute performances of **ExIn**, **Psi**, and **Fusion** on **Cubic** instances.

### 7.2.2 Performances and Running Times

We repeated the execution of each algorithm on a graph five times for **ExIn** and ten times for all other algorithms. The running time of an algorithm on a graph always corresponds to the median of the measured values.

Figure 7.2 displays the absolute performances on **Cubic** as the mean number of induced backward arcs of all candidates graphically. For those using a random initial linear ordering, the plot shows the average cardinality of the backward arc set over all ten repetitions.

In regard of the results obtained in other studies [CW09, BH11], which show a certain superiority of algorithms like *Nest*, its performance compared to *ELS* here is surprisingly bad on first sight. However, the graphs tested there were significantly denser with probabilities between 0.5 and 0.95 for two vertices to be connected via an arc. A small evaluation including *ELS* and an approach similar to *Nest* on cubic graphs of size 10 to 150 [Han10], by contrast, revealed similar relative performance tendencies. The plot shows that if *Nest* was run on a linear ordering produced by *ELS*, it was able to achieve a small improvement.

Equally striking, but now in the positive sense, is the performance of *Psi* and *Fusion*, which is once again considerably better than *ELS* or *Nest* | *ELS* and comes very close to the optimum solutions produced by *ExIn*. There is no visible difference between *Psi* and *Fusion* in this plot. Figure 7.3 therefore shows only the performances of *Psi*, *Fusion*, and *ExIn* again in a separate plot. Here, a very tiny gap between the graphs corresponding to *Psi* | *Any* and *Fusion* | *Any* can be perceived, with an advantage for the latter. As in Figure 7.2, the plot shows a small deviation of *Psi* and *Fusion* from the optimum solution that grows with the number of vertices.

For a more detailed evaluation of performances, all figures have been compiled in Table 7.1 for instances of *Cubic* with 20, 80, 140, and 200 vertices. It shows that on the set of graphs with 20 vertices, both *Psi* and *Fusion* found the optimum solution for each graph starting from 10 random linear orderings. If there was no variation in the input linear ordering during the repetitions, i. e., in case of *Any* and *ELS*, the optimum hit rate is still 96 out of 100. With increasing graph size, the number of times an optimum linear ordering was found declines and the mean deviation from the optimum grows, as was already indicated by the visualizations given in Figure 7.2 and Figure 7.3. Nevertheless, the optimum hit rate on graphs of size 140 is still between 31 and 34 for *Psi* and *Fusion* with *Any* and *ELS* as initializers, and even 76 and 79 for *Psi* | *Random* and *Fusion* | *Random*. The absolute mean deviation from the optimum is around one arc for these algorithms, which results in a mean percentage deviation of about 0.5% with respect to the set of forward arcs and less than 14% with respect to the set of backward arcs. Observe that for *Cubic-200*, i. e., cubic graphs with 200 vertices, no optimum solutions were available. For this reason, the mean deviations as well as the hit rate relate to the best solution found by any of the algorithms, which is however not necessarily the optimum. Interestingly, the number of best solution hits does not exceed 86 out of 100 for any candidate, which implies that the best solutions were obtained in some sort of “collaboration”.

Table 7.1: Performances and running times on *Cubic* instances.

<i>ExIn</i>	<i>Nest</i>   <i>Any</i>	<i>ELS</i>   <i>Any</i>	<i>Psi</i>   <i>Any</i>	<i>Fusion</i>   <i>Any</i>	<i>Nest</i>   <i>ELS</i>	<i>Nest</i>   <i>ELS</i>	<i>Psi</i>   <i>ELS</i>	<i>Fusion</i>   <i>ELS</i>	<i>Nest</i>   <i>Random</i>	<i>Psi</i>   <i>Random</i>	<i>Fusion</i>   <i>Random</i>	
<b>Cubic-20</b>												
mean $ \beta $	4.68	3.88	2.55	2.55	3.54	2.55	2.55	2.55	(4.58, 3.30)	(2.57, 2.51)	(2.55, 2.51)	
dev. % ( $\beta$ )	97.82	63.87	1.83	1.83	47.75	2.17	2.17	2.17	(94.28, 36.42)	(2.40, 0.00)	(1.92, 0.00)	
dev. % ( $\mathcal{F}$ )	7.87	4.96	0.14	0.14	3.73	0.14	0.14	0.14	(7.48, 2.85)	(0.21, 0.00)	(0.16, 0.00)	
#hits of OPT	100 / 100	21 / 100	96 / 100	96 / 100	29 / 100	96 / 100	96 / 100	96 / 100	34 / 100	100 / 100	100 / 100	
time	0.69 ms	1.12 ms	0.14 ms	2.84 ms	49.28 ms	0.21 ms	1.61 ms	44.97 ms	1.04 ms	2.70 ms	47.60 ms	
<b>Cubic-80</b>												
mean $ \beta $	17.99	11.45	6.42	6.40	10.56	6.39	6.39	6.35	(17.73, 14.69)	(6.37, 5.83)	(6.29, 5.82)	
dev. % ( $\beta$ )	220.88	104.00	11.60	11.24	87.81	10.88	10.88	10.18	(216.60, 161.77)	(10.76, 0.87)	(9.21, 0.70)	
dev. % ( $\mathcal{F}$ )	10.69	4.96	0.56	0.54	4.18	0.53	0.50	0.50	(10.45, 7.80)	(0.51, 0.04)	(0.45, 0.04)	
#hits of OPT	100 / 100	0 / 100	54 / 100	54 / 100	0 / 100	56 / 100	57 / 100	57 / 100	0 / 100	95 / 100	96 / 100	
time	823.61 ms	18.92 ms	0.34 ms	37.02 ms	889.92 ms	1.01 ms	14.22 ms	767.62 ms	10.85 ms	38.52 ms	793.13 ms	
<b>Cubic-140</b>												
mean $ \beta $	30.98	18.20	9.57	9.52	16.99	9.54	9.51	9.51	(31.01, 26.88)	(9.69, 8.78)	(9.63, 8.75)	
dev. % ( $\beta$ )	269.55	115.96	12.55	11.85	101.56	12.14	11.80	11.80	(269.79, 220.16)	(13.86, 2.93)	(13.11, 2.59)	
dev. % ( $\mathcal{F}$ )	11.14	4.80	0.52	0.49	4.20	0.50	0.49	0.49	(11.16, 9.11)	(0.57, 0.12)	(0.55, 0.11)	
#hits of OPT	100 / 100	0 / 100	31 / 100	32 / 100	0 / 100	32 / 100	34 / 100	34 / 100	0 / 100	76 / 100	79 / 100	
time	20 min 48.07 s	49.94 ms	0.56 ms	92.87 ms	3.41 s	1.98 ms	34.92 ms	3.39 s	29.83 ms	90.48 ms	3.36 s	
<b>Cubic-200 (11.31<sup>a</sup>)</b>												
mean $ \beta $	44.42	25.86	12.86	12.68	24.03	12.80	12.76	12.76	(43.93, 39.18)	(12.88, 11.60)	(12.78, 11.46)	
dev. % ( $\beta$ ) <sup>b</sup>	299.21	131.39	14.17	12.61	114.70	13.72	13.42	13.42	(294.81, 251.92)	(14.26, 2.70)	(13.25, 1.34)	
dev. % ( $\mathcal{F}$ ) <sup>b</sup>	11.47	5.04	0.54	0.47	4.41	0.52	0.50	0.50	(11.30, 9.65)	(0.54, 0.10)	(0.51, 0.05)	
#hits of best	0 / 100	0 / 100	17 / 100	23 / 100	0 / 100	22 / 100	22 / 100	22 / 100	0 / 100	75 / 100	86 / 100	
time	99.28 ms	0.84 ms	152.67 ms	9.42 s	3.91 ms	61.28 ms	9.93 s	9.93 s	68.11 ms	153.37 ms	9.50 s	

<sup>a</sup>mean of best solutions<sup>b</sup>with respect to the best solution

Comparing the performance of *Psi* and *Fusion* in terms of figures, we observe a small improvement towards the latter for larger graphs, which increases slowly as the number of vertices grows. The mean deviations from the optimum or best known solution show only marginal differences with respect to the initializers: If *ELS* was used instead of *Any* or *Random*, the obtained solutions are on average slightly closer to the optimum in many cases.

Looking at the running times, we can identify two main groups among the non-exact algorithms: Whereas *Nest*, *ELS*, and *Psi* terminated on average within at most a few milliseconds on graphs with 20 vertices and still within roughly 150 milliseconds on graphs with 200 vertices, *Fusion* needed around 50 milliseconds on average on the set of the smallest instances, and almost 10 seconds on that of the largest ones. It is worth noting that the running time of *Psi* was reduced if it started from a linear ordering produced by *ELS*. *ExIn*'s average computation time ranged between 0.7 milliseconds on graphs with 20 vertices and 21 minutes on graphs with 140 vertices. Interestingly, the mean time consumed by running *ExIn* on a graph with 20 vertices undercuts the mean time consumed by running any other non-exact algorithm except for *ELS* alone and *Nest* | *ELS*. For instances from *Cubic-80*, the average running time of *ExIn* was still on about the same level as that of *Fusion*. The running time of *ExIn* on a graph with 80 vertices varied between 14 milliseconds and 6 seconds, and between 2.5 seconds and 3 hours for graphs with 140 vertices. The one-hour-barrier was broken first for a graph with 120 vertices, and also the maximum running time of 3 hours and 49 minutes was measured for a graph from this set.

The situation is very similar for quartic graphs, as [Figure 7.4](#) shows. Again, *Nest* | *Any* and *Nest* | *Random* are clearly outperformed by *ELS*. Running *Nest* on the linear ordering obtained by *ELS* seems to increase the solution quality slightly more than for *Cubic*, the discrepancy in performance between *Nest* | *ELS* and that of *Psi* and *Fusion* is however still markedly. For a better visualization of the differences between the variants of *Psi* and *Fusion*, [Figure 7.5](#) provides again a separate plot. As in the case of cubic graphs, a small gap between the performance graphs of *Psi* | *Any* and *Fusion* | *Any* are discernible. The plot also shows that the obtained solutions are on average close to the optimum, with a deviation that increases with the number of vertices.

For a better assessment of the solution qualities and running times, [Table 7.2](#) shows the key figures for *ExIn*, *Psi*, *PsiAlt*, and *Fusion* for different initial linear orderings on quartic instances of size 20, 50, 140, and 200. On graphs with 20 vertices, i. e., *Quartic-20*, *Fusion* was still able to find an optimum solution if the algorithm was started ten

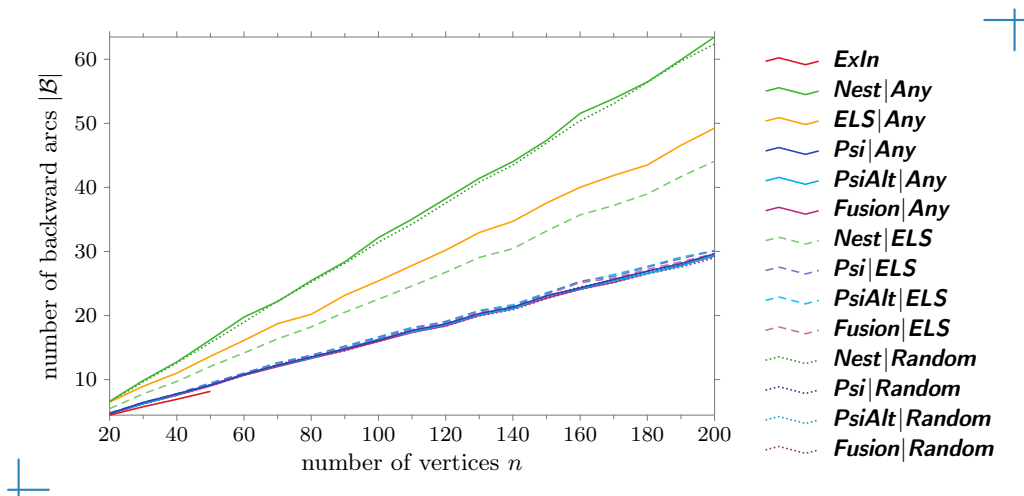


Figure 7.4: Comparison of absolute performances on *Quartic* instances.

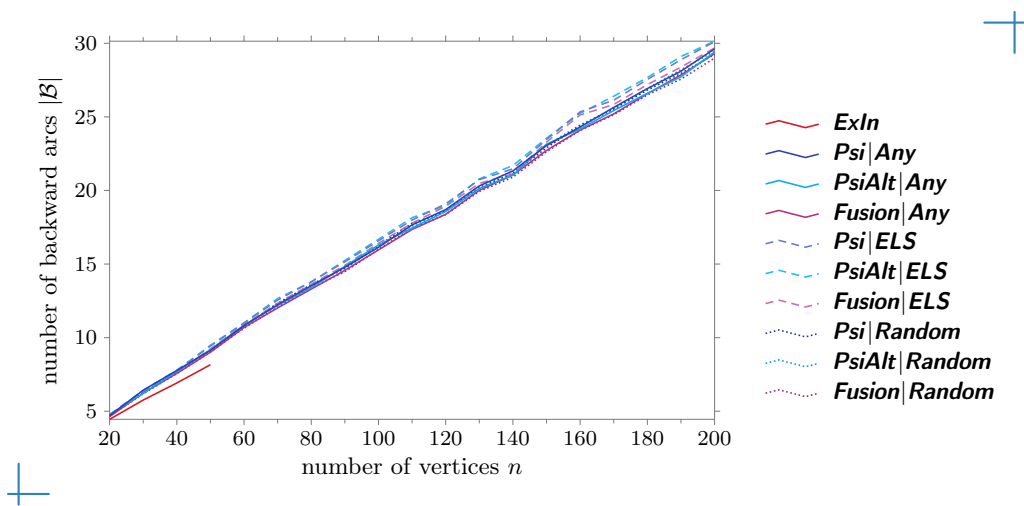


Figure 7.5: Absolute performances of *ExIn*, *Psi*, *PsiAlt*, and *Fusion* on *Quartic* instances.

Table 7.2: Performances and running times on *Quartic* instances.

	<i>ExIn</i>	<i>Psi</i>   <i>Any</i>	<i>PsiAlt</i>   <i>Any</i>	<i>Fusion</i>   <i>Any</i>	<i>Psi</i>   <i>ELS</i>	<i>PsiAlt</i>   <i>ELS</i>	<i>Fusion</i>   <i>ELS</i>	<i>Psi</i>   <i>Random</i>	<i>PsiAlt</i>   <i>Random</i>	<i>Fusion</i>   <i>Random</i>
<b>Quartic-20</b>										
mean   <i>B</i>	4.45	4.70	4.77	4.64	4.76	4.77	4.65	(4.75, 4.46)	(4.72, 4.46)	(4.66, 4.45)
dev. % ( <i>B</i> )	0.00	5.93	7.08	4.42	7.69	7.73	5.18	(7.07, 0.25)	(6.09, 0.33)	(4.79, 0.00)
dev. % ( <i>F</i> )	0.00	0.70	0.91	0.54	0.87	0.90	0.56	(0.85, 0.03)	(0.76, 0.03)	(0.58, 0.00)
#hits of OPT	100 / 100	78 / 100	73 / 100	81 / 100	75 / 100	73 / 100	83 / 100	99 / 100	99 / 100	100 / 100
time	13.05 ms	7.34 ms	5.33 ms	58.90 ms	5.03 ms	2.47 ms	50.85 ms	6.70 ms	4.96 ms	56.19 ms
<b>Quartic-50</b>										
mean   <i>B</i>	8.16	9.14	9.18	9.01	9.43	9.48	9.24	(9.17, 8.32)	(9.13, 8.30)	(9.02, 8.27)
dev. % ( <i>B</i> )	0.00	12.22	12.69	10.67	15.65	16.42	13.23	(12.57, 1.90)	(11.99, 1.63)	(10.80, 1.37)
dev. % ( <i>F</i> )	0.00	1.07	1.11	0.93	1.39	1.44	1.18	(1.09, 0.17)	(1.06, 0.15)	(0.94, 0.12)
#hits of OPT	100 / 100	41 / 100	32 / 100	44 / 100	26 / 100	24 / 100	32 / 100	84 / 100	86 / 100	89 / 100
time	7.0 min 4.15 s	33.76 ms	29.08 ms	385.12 ms	19.12 ms	10.60 ms	293.93 ms	31.56 ms	26.14 ms	363.43 ms
<b>Quartic-140 (18.56<sup>(a)</sup>)</b>										
mean   <i>B</i>	21.33	21.04	21.04	21.05	21.47	21.69	21.13	(21.15, 19.14)	(21.20, 19.19)	(20.90, 18.89)
dev. % ( <i>B</i> ) <sup>b</sup>	15.15	13.67	13.67	13.65	15.92	17.09	14.15	(14.24, 3.12)	(14.47, 3.44)	(12.83, 1.85)
dev. % ( <i>F</i> ) <sup>b</sup>	1.06	0.95	0.95	0.95	1.11	1.20	0.98	(0.99, 0.22)	(1.01, 0.24)	(0.90, 0.13)
#hits of best	6 / 100	8 / 100	8 / 100	8 / 100	3 / 100	3 / 100	6 / 100	56 / 100	54 / 100	72 / 100
time	123.10 ms	100.38 ms	100.38 ms	4.83 s	73.52 ms	49.32 ms	4.65 s	137.49 ms	86.65 ms	4.71 s
<b>Quartic-200 (25.72<sup>(a)</sup>)</b>										
mean   <i>B</i>	29.62	29.40	29.40	29.29	30.11	30.14	29.68	(29.25, 26.56)	(29.46, 26.68)	(28.99, 26.45)
dev. % ( <i>B</i> ) <sup>b</sup>	15.27	14.55	14.55	14.00	17.20	17.30	15.54	(13.94, 3.32)	(14.71, 3.78)	(12.88, 2.90)
dev. % ( <i>F</i> ) <sup>b</sup>	1.04	0.98	0.98	0.95	1.17	1.18	1.06	(0.94, 0.22)	(1.00, 0.26)	(0.87, 0.19)
#hits of best	3 / 100	7 / 100	7 / 100	6 / 100	1 / 100	2 / 100	3 / 100	49 / 100	42 / 100	54 / 100
time	229.45 ms	211.29 ms	211.29 ms	14.98 s	129.25 ms	77.04 ms	14.45 s	250.50 ms	181.30 ms	15.85 s

<sup>a</sup>mean of best solutions<sup>b</sup>with respect to the best solution

times from a random initial linear ordering, and also *Psi* and *PsiAlt* missed just one. The deterministic variants reached the optimum less often with 73 to 83 out of 100. The hit rate declines rapidly as the size of the graphs grows and drops below 10% already for the non-randomized algorithms on instances with 140 vertices. Observe that for *Quartic-140* and *Quartic-200*, the reference for the mean deviations as well as the hit rate is the best solution that was found by any of the candidates, which is not necessarily the optimum solution. The fact that the best solutions were not produced by a single algorithm alone, which has already been observed on *Cubic*, becomes even more apparent here: *Fusion* | *Random* as the most successful candidate found in only 54 out of 100 cases the best solution.

As the plots suggested, the mean percentage deviation from the optimum or best known solution increases with the number of vertices. For *Quartic-50*, the solutions produced by *Psi*, *PsiAlt*, and *Fusion* differ from the optimum by little more than one arc on average, which results in a relative deviation of at most 1.44% and 16.42% with respect to the set of forward and backward arcs, respectively.

Even though *Psi* and *PsiAlt* establish the same set of properties on a given linear ordering, their performances differ slightly: In case of *Any* as initial linear ordering, *Psi* seems to outperform *PsiAlt* on smaller instances, whereas this goes into reverse on larger graphs. Exactly the opposite can be observed for *Random*, whereas for *ELS* as initial linear ordering, *Psi* shows the lower mean deviations. Hence, with respect to their performances, no clear winner can be determined. The figures also show that *Fusion* yields the best results among all non-exact candidates. In comparison to the results for cubic graphs, the performance gap between *Fusion* and *Psi* or *PsiAlt* emerges even more clearly.

The running time of *Psi* and *PsiAlt* ranged on average between a few milliseconds on instances with 20 vertices and 250 milliseconds on graphs from *Quartic-200*. By contrast, the execution of *Fusion* required around 50 milliseconds to 16 seconds. Again, *ExIn* ran on average faster than *Fusion* on small instances, with about 13 milliseconds on the graphs in *Quartic-20*. However, in terms of the mean running time, *Fusion* outran *ExIn* already for instances of size 30, with around 150 milliseconds for the former in comparison to 400 milliseconds for the latter candidate. The maximum computation time consumed by *ExIn* remained below three minutes for instances of up to 40 vertices, but jumped to 4.5 hours for *Quartic-50*. Nevertheless, the minimum time for a graph with 50 vertices was still below 3 seconds. The comparison of *Psi* and *PsiAlt* with respect to

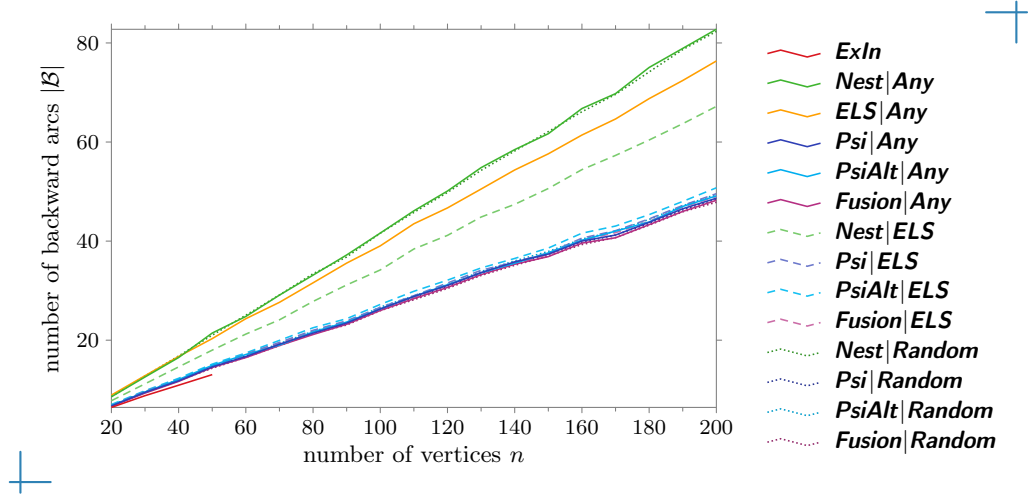


Figure 7.6: Comparison of absolute performances on *Quintic* instances.

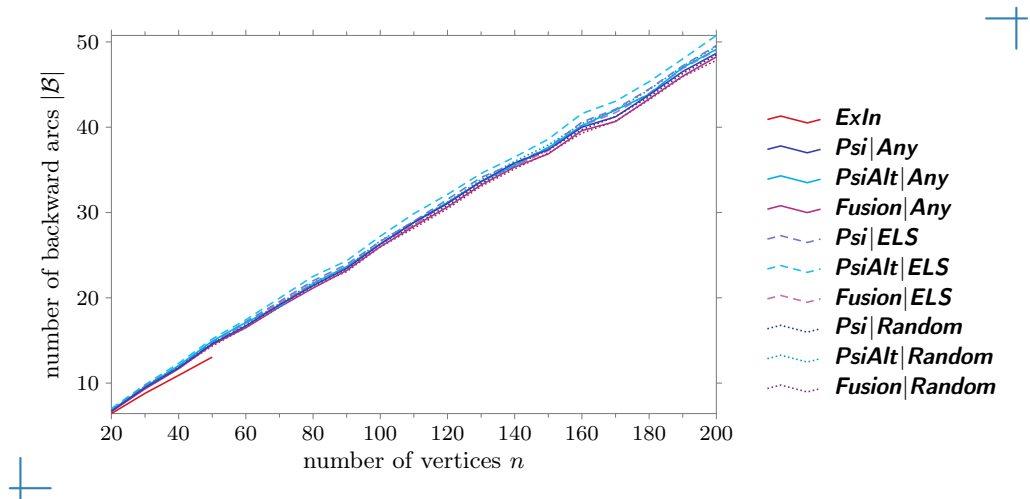


Figure 7.7: Absolute performances of *ExIn*, *Psi*, *PsiAlt*, and *Fusion* on *Quintic* instances.

their running time shows that *PsiAlt* terminated faster on average, in particular if both algorithms started from a linear ordering computed by *ELS*.

The benchmark results for quintic graphs do not differ significantly from those for quartic graphs. Figure 7.6 displays the average number of backward arcs induced by the solutions obtained from the various algorithms. The performance of *Nest* was again the poorest, whereas *Psi*, *PsiAlt*, and *Fusion* yielded results close to the optimum. For a better resolution of the graphs corresponding to the latter three, a separate plot is



provided in Figure 7.7. Interestingly, *PsiAlt* | *ELS* seems to perform marginally worse than its competitors.

A more detailed comparison of the approaches can be made from the figures compiled in Table 7.3, which reports on the results for *ExIn*, *Psi*, *PsiAlt*, and *Fusion* on instances with 20, 50, 140, and 200 vertices. Similar to *Quartic*, the best hit rate is close to 100% for all algorithms if they were started ten times from a random initial linear ordering, and declines rapidly as the size of the graphs grows. The deviations also confirm that *PsiAlt* | *ELS* is outperformed by its competitors, although the differences are rather small. What is remarkable, however, is that *PsiAlt* is in almost all cases outmatched by *Psi*, which also becomes evident in the optimum/best hit rate. Again, *Fusion* appears superior to both *Psi* and *PsiAlt*, with an even greater distinction than could be observed on *Quartic* and *Cubic*. The mean percentage deviation from the optimum on graphs with 50 vertices is still below 1.9% and 16.38% with respect to the set of forward and backward arcs, respectively.

The measured running times on quintic graphs range between hardly 10 milliseconds on graphs with 20 vertices and a little more than 300 milliseconds on graphs with 200 vertices for *Psi* and *PsiAlt*, whereas *Fusion* needed between about 60 milliseconds and 16 seconds. *ExIn*, in comparison, terminated within 15 milliseconds on average on graphs with 20 vertices, and thereby once again outraced *Fusion*. For graphs with 50 vertices, however, the mean execution time was already beyond 17 minutes. The maximum running time of *ExIn* on a quintic graph was approximately 5 hours and 10 minutes on a graph with 50 vertices. In contrast, *ExIn* terminated within 4 minutes on all graphs with 40 vertices or less. The minimum time needed by *ExIn* on an instance from *Quintic-50* was less than 10 seconds.

### 7.2.3 Summary

The algorithms *Psi*, *PsiAlt*, and *Fusion* show very good performances on the test sets containing 3-, 4, and 5-regular graphs with low mean deviations from the optimum solution. Here, they stand out clearly against approaches like *Nest* or *ELS*.

The comparison of *Psi* and *PsiAlt* reveals slightly better performance results for the former on quintic graphs, whereas the picture is rather unclear on quartic instances. The additional effort to establish the Prefix Fusion Property and the Greedy Fusion Property only seems to pay off for larger and slightly denser than cubic graphs, however at the expense of a faster termination. Furthermore, the measured execution times of

Table 7.3: Performances and running times on *Quintic* instances.

	<i>ExIn</i>	<i>Psi</i>   <i>Any</i>	<i>PsiAlt</i>   <i>Any</i>	<i>Fusion</i>   <i>Any</i>	<i>Psi</i>   <i>ELS</i>	<i>PsiAlt</i>   <i>ELS</i>	<i>Fusion</i>   <i>ELS</i>	<i>Psi</i>   <i>Random</i>	<i>PsiAlt</i>   <i>Random</i>	<i>Fusion</i>   <i>Random</i>
<b><i>Quintic-20</i></b>										
mean $ B $	6.43	6.72	6.80	6.66	6.78	7.02	6.70	(6.80, 6.45)	(6.83, 6.46)	(6.71, 6.45)
dev. % ( $B$ )	0.00	4.93	5.94	3.77	5.67	9.39	4.29	(5.87, 0.28)	(6.39, 0.44)	(4.48, 0.28)
dev. % ( $\mathcal{F}$ )	0.00	0.66	0.85	0.53	0.80	1.35	0.62	(0.84, 0.05)	(0.93, 0.07)	(0.64, 0.05)
#hits of OPT	100 / 100	75 / 100	67 / 100	78 / 100	69 / 100	58 / 100	76 / 100	98 / 100	97 / 100	98 / 100
time	14.61 ms	9.39 ms	7.11 ms	65.53 ms	6.48 ms	3.05 ms	58.92 ms	8.83 ms	6.18 ms	62.87 ms
<b><i>Quintic-50</i></b>										
mean $ B $	13.05	14.61	14.92	14.50	14.80	15.17	14.65	(14.56, 13.30)	(14.68, 13.41)	(14.34, 13.25)
dev. % ( $B$ )	0.00	12.25	14.50	11.36	13.86	16.38	12.57	(11.75, 1.96)	(12.73, 2.70)	(10.04, 1.54)
dev. % ( $\mathcal{F}$ )	0.00	1.39	1.67	1.29	1.56	1.90	1.43	(1.35, 0.22)	(1.46, 0.32)	(1.15, 0.18)
#hits of OPT	100 / 100	16 / 100	13 / 100	20 / 100	12 / 100	8 / 100	15 / 100	76 / 100	66 / 100	81 / 100
time	17.0 min 46.61 s	42.70 ms	34.16 ms	384.88 ms	24.57 ms	12.90 ms	329.01 ms	39.35 ms	31.75 ms	366.84 ms
<b><i>Quintic-140</i> (32.01<sup>a</sup>)</b>										
mean $ B $	35.80	35.54	35.33	35.79	36.50	35.43	(35.52, 32.80)	(36.01, 33.02)	(35.12, 32.45)	
dev. % ( $B$ ) <sup>b</sup>	12.01	11.12	10.54	11.97	14.12	10.84	(11.13, 2.51)	(12.62, 3.20)	(9.84, 1.40)	
dev. % ( $\mathcal{F}$ ) <sup>b</sup>	1.19	1.11	1.04	1.19	1.41	1.07	(1.10, 0.25)	(1.26, 0.32)	(0.98, 0.14)	
#hits of best	6 / 100	9 / 100	10 / 100	3 / 100	1 / 100	9 / 100	46 / 100	38 / 100	68 / 100	
time	156.18 ms	126.98 ms	5.16 s	95.16 ms	46.77 ms	4.43 s	151.62 ms	108.73 ms	4.80 s	
<b><i>Quintic-200</i> (44.04<sup>a</sup>)</b>										
mean $ B $	48.66	49.13	48.24	49.57	50.76	49.03	(48.48, 45.09)	(49.45, 45.47)	(47.84, 44.54)	
dev. % ( $B$ ) <sup>b</sup>	10.71	11.60	9.72	12.72	15.33	11.49	(10.24, 2.43)	(12.43, 3.33)	(8.78, 1.17)	
dev. % ( $\mathcal{F}$ ) <sup>b</sup>	1.01	1.12	0.92	1.21	1.47	1.09	(0.97, 0.23)	(1.19, 0.31)	(0.83, 0.11)	
#hits of best	6 / 100	2 / 100	7 / 100	3 / 100	1 / 100	3 / 100	45 / 100	34 / 100	66 / 100	
time	306.26 ms	259.06 ms	16.11 s	174.35 ms	77.98 ms	14.16 s	301.17 ms	230.41 ms	15.09 s	

<sup>a</sup> mean of best solutions<sup>b</sup> with respect to the best solution

the exact algorithm **ExIn** remained below those of **Fusion** on small instances, while naturally providing the better results. In general, the execution times of **ExIn** varied widely between a few milliseconds or seconds and several hours, even if the graphs were equal in size.

## 7.3 Large Graphs

The experimental results on sparse regular graphs revealed a number of interesting facts with regard to the algorithms' performances. To gain a better overview of the candidates' behavior on sparse graphs, we conducted another evaluation on larger graphs which are still sparse, but no longer regular.

### 7.3.1 Fences, Ladders, and their Composites

We generated 28  $k$ -fences for  $k = 3, \dots, 30$  as well as 2640 Möbius ladders. For the latter, we used three to eleven cycles per graph. The maximum vertex degree of the obtained graphs was ten, whereas the minimum was three. All  $k$ -fences and Möbius ladders have been put together in the sets **Fences** and **Möbius Ladders**, respectively. The graphs in these sets also form the initial population of a set of graphs  $S$  that we used for the construction of the actual test instances.

To generate large sparse graphs, we applied the following procedure: Let  $G$  be a graph that is randomly chosen from  $S$ . We denote by  $n_G$  the number of  $G$ 's vertices and by  $N$  the target size, i. e., the number of vertices that  $G$  should finally have. Consider a subset  $S'$  of  $S$  that contains only graphs whose number of vertices plus  $n_G$  does not exceed  $N + 2$ . If  $S'$  is not empty, choose a graph  $H$  from  $S'$  at random with uniform distribution and attach  $H$  to  $G$  by means of a two-clique-sum: Select one arc of  $G$  and  $H$ , respectively, identify the head of the former with the tail of the latter and vice versa, and remove both arcs. This yields a composite graph with  $n_{G+H} = n_G + n_H - 2$  vertices, where  $n_H$  denotes the number of vertices in  $H$ . Let  $G$  now in turn be this composite and repeat the steps until no further graph can be attached such that  $n_G \leq N$ . As the smallest graph in  $S$  has six vertices, the size of  $G$  differs from  $N$  by at most 5. For the construction of the next graph, we added  $G$  to  $S$ .

The set of benchmark instances **Facet** consists of 50 graphs for each target size between 100 and 1,000 in intervals of 100. The average vertex degree of a graph in **Facet** is slightly less than four, however with an average median of three. For all graphs, the minimum

vertex degree is three, whereas the maximum ranges between six on smaller instances and up to 48 on larger ones.

The optimum solution value of a graph can be obtained in unison with its construction by summing up the optimum solution values of the two combined graphs and subtracting one in each step i. e.,  $\tau_{G+H} = \tau_G + \tau_H - 1$  [BFM94].

Due to the long computation time of *Fusion*, we decided again upon a time limit of 4 hours. In consequence, *Fusion* was run only on graphs with at most 700 vertices.

### 7.3.2 Performances and Running Times

As in case of the sparse regular instances, we ran every algorithm ten times on every graph. The evaluation of the running time always uses the median of all repetitions.

For a better understanding of the graphs contained in *Facet* and their “difficulty” with respect to the LINEAR ORDERING problem, we first executed the candidates on *Fences* and *Möbius Ladders*. Table 7.4 lists the obtained results. In view of the distinctive in- and outdegree distribution of fence graphs, it does not come as a great surprise that every algorithm found an optimum solution on all 28 instances. In case of Möbius ladders, differences in performance become visible. The mean optimum number of backward arcs for the graphs in *Möbius Ladder* is 5.69. If started ten times from a random linear ordering, *Psi*, *PsiAlt*, and *Fusion* were able to find the optimum solution on every instance. *Psi | Any* and *Fusion | Any* still show a very good optimum solution hit rate and only low mean percentage deviations from the optimum. Similar values were obtained by *Fusion | Random* if we consider the mean performance of all repetitions. *Nest | Any* and *Nest | Random* (with regard to the average performance) produced the worst results, followed by *ELS | Any* and *Nest | ELS*. The picture is hence similar to that seen on the sets *Cubic*, *Quartic*, and *Quintic*.

We now take a look at the performances on *Facet*, as depicted in Figure 7.8. As before, the plot shows the mean number of backward arcs obtained by the algorithms. For those candidates starting from a random linear ordering, we used the average value. The observable results are consistent with those for *Cubic*, *Quartic*, and *Quintic*. The deviations from the optimum solution are again very small, in particular for *Psi* and *Fusion*, and seem to grow with a similar rate as for smaller instances. Unfortunately, the evaluation of *Fusion* on graphs with 800 vertices and more had to be aborted due to running times of several hours.

Table 7.4: Performances and running times on *Fences* and *Möbius Ladders* instances.

	<i>Nest</i>   <i>Any</i>	<i>ELS</i>   <i>Any</i>	<i>Psi</i>   <i>Any</i>	<i>PsiAlt</i>   <i>Any</i>	<i>Fusion</i>   <i>Any</i>	<i>Nest</i>   <i>ELS</i>	<i>Psi</i>   <i>ELS</i>	<i>PsiAlt</i>   <i>ELS</i>	<i>Fusion</i>   <i>ELS</i>
<b>Fences</b> (15.50 <sup>a</sup> )									
mean   $\beta$	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50
dev. % ( $\beta$ )	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
dev. % ( $\mathcal{F}$ )	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
#hits of OPT	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28
time	22.94 ms	0.35 ms	54.55 ms	38.27 ms	625.96 ms	0.75 ms	20.60 ms	7.02 ms	193.50 ms
<b>Möbius Ladders</b> (5.69 <sup>a</sup> )									
mean   $\beta$	6.33	5.85	5.69	5.74	5.69	5.81	5.71	5.73	5.70
dev. % ( $\beta$ )	11.10	2.71	0.07	0.92	0.06	2.08	0.40	0.67	0.23
dev. % ( $\mathcal{F}$ )	3.10	0.75	0.02	0.26	0.02	0.58	0.11	0.18	0.06
#hits of OPT	1109 / 2640	2221 / 2640	2629 / 2640	2497 / 2640	2631 / 2640	2316 / 2640	2577 / 2640	2535 / 2640	2604 / 2640
time	0.73 ms	0.12 ms	5.56 ms	3.43 ms	38.05 ms	0.14 ms	3.02 ms	1.11 ms	31.96 ms
<b>Fences</b> (15.50 <sup>a</sup> )									
mean   $\beta$	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50
dev. % ( $\beta$ )	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
dev. % ( $\mathcal{F}$ )	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
#hits of OPT	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28	28 / 28
time	19.09 ms	43.74 ms	32.50 ms	940.53 ms	43.74 ms	32.50 ms	940.53 ms	43.74 ms	32.50 ms
<b>Möbius Ladders</b> (5.69 <sup>a</sup> )									
mean   $\beta$	(6.21, 5.69)	(5.70, 5.69)	(5.73, 5.69)	(5.69, 5.69)	(5.69, 5.69)	(5.69, 5.69)	(5.69, 5.69)	(5.69, 5.69)	(5.69, 5.69)
dev. % ( $\beta$ )	(9.05, 0.06)	(0.23, 0.00)	(0.80, 0.00)	(0.07, 0.00)	(0.07, 0.00)	(0.07, 0.00)	(0.07, 0.00)	(0.07, 0.00)	(0.07, 0.00)
dev. % ( $\mathcal{F}$ )	(2.54, 0.02)	(0.06, 0.00)	(0.22, 0.00)	(0.02, 0.00)	(0.02, 0.00)	(0.02, 0.00)	(0.02, 0.00)	(0.02, 0.00)	(0.02, 0.00)
#hits of OPT	2630 / 2640	2640 / 2640	2640 / 2640	2640 / 2640	2640 / 2640	2640 / 2640	2640 / 2640	2640 / 2640	2640 / 2640
time	0.62 ms	4.87 ms	2.89 ms	37.83 ms	4.87 ms	2.89 ms	37.83 ms	4.87 ms	2.89 ms

<sup>a</sup>mean of OPT solutions

Table 7.5: Performances and running times on *Facet* instances with initializers *Any* and *ELS*.

	<i>Nest</i>   <i>Any</i>	<i>ELS</i>   <i>Any</i>	<i>Psi</i>   <i>Any</i>	<i>PsiAlt</i>   <i>Any</i>	<i>Fusion</i>   <i>Any</i>	<i>Nest</i>   <i>ELS</i>	<i>Psi</i>   <i>ELS</i>	<i>PsiAlt</i>   <i>ELS</i>	<i>Fusion</i>   <i>ELS</i>
<b>Facet-100 (34.74<sup>a</sup>)</b>									
mean   $\mathcal{B}$	40.60	38.52	36.24	36.48	35.84	37.84	36.28	36.38	36.08
dev. % ( $\mathcal{B}$ )	16.98	10.90	4.32	5.03	3.16	8.93	4.43	4.72	3.86
dev. % ( $\mathcal{F}$ )	4.19	2.67	1.07	1.23	0.78	2.18	1.09	1.16	0.94
#hits of OPT	0 / 50	0 / 50	9 / 50	3 / 50	15 / 50	0 / 50	6 / 50	5 / 50	8 / 50
time	29.79 ms	0.52 ms	92.49 ms	56.35 ms	3.28 s	1.82 ms	50.39 ms	20.52 ms	2.60 s
<b>Facet-500 (174.24<sup>a</sup>)</b>									
mean   $\mathcal{B}$	208.24	197.68	182.40	184.12	181.16	193.72	183.26	184.64	181.78
dev. % ( $\mathcal{B}$ )	19.53	13.46	4.69	5.67	3.98	11.19	5.18	5.97	4.33
dev. % ( $\mathcal{F}$ )	4.51	3.10	1.08	1.30	0.92	2.58	1.19	1.37	1.00
#hits of OPT	0 / 50	0 / 50	0 / 50	0 / 50	0 / 50	0 / 50	0 / 50	0 / 50	0 / 50
time	494.07 ms	2.50 ms	962.86 ms	1.10 s	17.0 min 9.93 s	18.56 ms	453.76 ms	215.10 ms	18.0 min 43.38 s
<b>Facet-1000 (351.14<sup>a</sup>)</b>									
mean   $\mathcal{B}$	418.38	398.34	368.12	371.74		389.64	367.90	370.60	
dev. % ( $\mathcal{B}$ )	19.15	13.45	4.83	5.87		10.97	4.77	5.54	
dev. % ( $\mathcal{F}$ )	4.24	2.97	1.06	1.29		2.42	1.05	1.22	
#hits of OPT	0 / 50	0 / 50	0 / 50	0 / 50		0 / 50	0 / 50	0 / 50	
time	2.11 s	5.50 ms	4.16 s	5.25 s		69.55 ms	1.19 s	654.62 ms	

<sup>a</sup> mean of OPT solutions

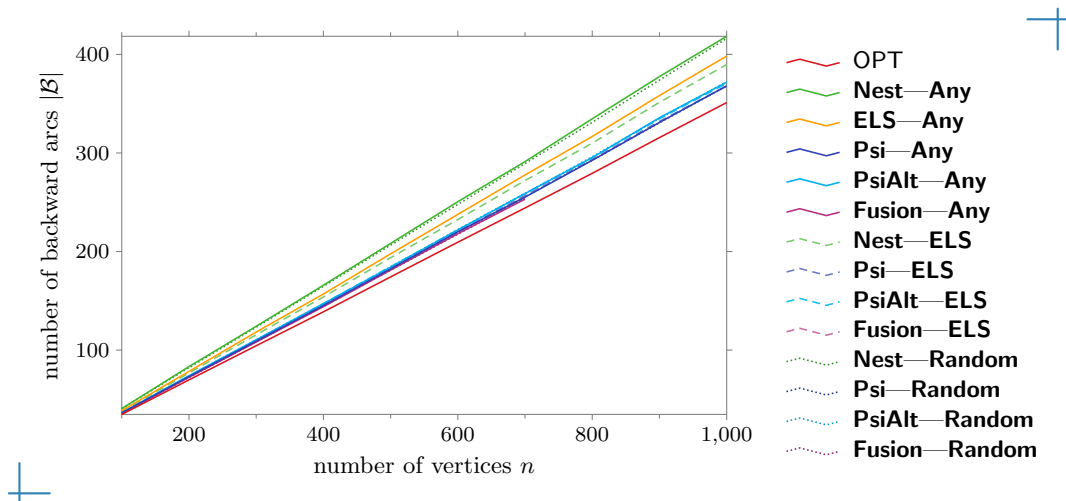


Figure 7.8: Comparison of absolute performances on *Facet* instances.

The compilation of detailed figures in Table 7.5 and Table 7.6 once again confirms the results from the previous section. The deviations of *Psi* and *PsiAlt* from the optimum are in the region of 5% with respect to the set of backward arcs and 1% with respect to the set of forward arcs. If started ten times from a random linear ordering, their performance improved notably. Again, *Psi* outperforms *PsiAlt* by presenting lower mean percentage deviations. *Fusion* came even closer to the optimum than *Psi*, however at the expense of a markedly increased running time, whereas those of *Psi* and *PsiAlt* still did not exceed a couple of seconds on instances with 1000 vertices. The solutions obtained by *Nest* and *ELS* are on average worse than those of its competitors. However, *ELS* remained below 10 milliseconds even on the largest graphs in the set. The OPT hit ratio can be observed to decline further as the size of the input instances grows. On *Facet-500*, none of the algorithms was able to find an optimum solution for a graph.

### 7.3.3 Summary

The evaluation of the benchmarking results on large and non-regular graphs reaffirms the impression from Section 7.2: *Fusion* yielded on average the best results and came very close to the optimum, but required comparatively long running times. *Psi* performed slightly worse than *Fusion*, but terminated within a few seconds even on graphs with 1,000 vertices. The solutions obtained by *PsiAlt* were on average not as good as those by *Psi* and the gap also seems to increase marginally with the size of the input

Table 7.6: Performances and running times on *Facet* instances with initializer *Random*.

	<i>Nest</i>   <i>Random</i>	<i>Psi</i>   <i>Random</i>	<i>PsiAlt</i>   <i>Random</i>	<i>Fusion</i>   <i>Random</i>
<b>Facet-100</b> (34.74 <sup>a</sup> )				
mean $ \mathcal{B} $	(40.59, 38.48)	(36.18, 35.20)	(36.55, 35.30)	(35.95, 35.08)
dev. % ( $\mathcal{B}$ )	(16.93, 10.81)	(4.16, 1.34)	(5.23, 1.63)	(3.46, 0.96)
dev. % ( $\mathcal{F}$ )	(4.16, 2.65)	(1.02, 0.33)	(1.28, 0.40)	(0.84, 0.24)
#hits of OPT	0 / 50	27 / 50	24 / 50	33 / 50
time	27.74 ms	90.87 ms	51.80 ms	2.79 s
<b>Facet-500</b> (174.24 <sup>a</sup> )				
mean $ \mathcal{B} $	(206.48, 200.96)	(182.30, 179.38)	(184.54, 181.18)	(181.01, 178.90)
dev. % ( $\mathcal{B}$ )	(18.53, 15.36)	(4.63, 2.95)	(5.92, 3.99)	(3.89, 2.68)
dev. % ( $\mathcal{F}$ )	(4.28, 3.56)	(1.06, 0.68)	(1.36, 0.92)	(0.89, 0.61)
#hits of OPT	0 / 50	0 / 50	0 / 50	0 / 50
time	474.54 ms	976.50 ms	957.37 ms	15.0 min 33.85 s
<b>Facet-1000</b> (351.14 <sup>a</sup> )				
mean $ \mathcal{B} $	(416.15, 407.86)	(367.77, 363.48)	(372.39, 367.68)	
dev. % ( $\mathcal{B}$ )	(18.52, 16.16)	(4.74, 3.51)	(6.05, 4.71)	
dev. % ( $\mathcal{F}$ )	(4.11, 3.58)	(1.04, 0.77)	(1.33, 1.04)	
#hits of OPT	0 / 50	0 / 50	0 / 50	
time	2.15 s	4.11 s	4.52 s	

<sup>a</sup>mean of OPT solutions

instances. Both *Nest* and *ELS* are clearly inferior to the other candidates with regard to the performance, but are in the lead with respect to the running time.

## 7.4 The LOLIB Graph Library

The last category of test instances used in this evaluation consists of graphs provided by the LOLIB graph library, which has been used by Martí and Reinelt [MR11] to evaluate a number of heuristics and meta-heuristics. The library was included in this study for one thing to also assess the algorithms' performance on dense graphs, for another thing to broaden the number of competitors and obtain further results to compare against.

### 7.4.1 Sets of LOLIB Instances

The LOLIB graph library consists of eight different sets of input instances, most of which are relatively dense and weighted. For this study, loops as well as two-cycles have been removed from the graphs in a preprocessing step. In the following, we give a



short summary of the sets of instances based on the description provided by Martí and Reinelt [MR11]. As usual,  $n$  denotes a graph's number of vertices.

The set **IO** is compiled from different input-output matrices and contains 50 graphs with 44 to 79 vertices and 270 to 2705 arcs with a median vertex degree between  $0.32n$  and  $0.95n$ . The total weight of the arcs ranges between 16,938 and 296,170,682.

The instances in **RandA1**, **RandA2**, **RandB** were generated randomly. For **RandA1**, the number of vertices is 100, 150, 200, and 500, with 25 instances for each size. The arc weights of the graphs were generated by a  $(0, 100)$  uniform distribution, which results in a minimum vertex degree of  $0.93n$ . The number of arcs for graphs in this set lies between 4,892 and 123,581, the total arc weight between 161,639 and 4,222,286. **RandA2** contains graphs with 100, 150, and 200 vertices and again 25 instances per graph size. They were obtained from  $\frac{n}{2}$  random permutations for each size  $n$  from the number of times that each relative order of a pair of vertices appeared in one of the permutations. The graphs in this set have between 4,338 and 18,369 arcs with a total arc weight of 25,696 to 163,470. The vertex degrees range between  $0.73n$  and  $0.96n$ . The 90 graphs in **RandB** were generated with the intent to create difficult instances for the LINEAR ORDERING problem. Their number of vertices is 40, 44, and 50, and the arc weights were drawn uniformly from different, specifically chosen ranges. The resulting graphs have 766 to 1,219 arcs and total arc weights of 38,165 to 70,498. Also these graphs are very dense with the minimum relative vertex degree being  $0.89n$ .

The set **SGB** contains 25 graphs that originate from the Stanford GraphBase [Knu09] and have a size of 75 vertices. The arc weights were again drawn uniformly from  $\{0, \dots, 25000\}$ . Graphs in this set have between 2,432 and 2,550 arcs with a total arc weight of 3,543,100 to 3,792,880. The vertex degree in this set ranges between  $0.31n$  and  $0.99n$ .

**XLOLIB** is a set of 78 benchmark instances by Schiavinotto and Stützle [SS04] that were generated from enlarged input-output matrices. The graphs have 150 or 250 vertices, their number of arcs is between 3,949 and 29,548. The total arc weight includes values from 286,139 to 140,281,693. The density of the graphs varies with vertex degrees of  $0.25n$  to  $0.99n$ .

The instances in **MB** originate from Mitchell and Borchers [MB00] and were obtained from random matrices where all arc weights were drawn uniformly from either  $\{0, \dots, 99\}$  or  $\{0, \dots, 39\}$ . Afterwards, some percentage of the arcs were removed again. The resulting 30 graphs have 100 to 250 vertices and 4,671 to 30,824 arcs. The vertex degree ranges between  $0.87n$  and  $0.97n$ , the total arc weight is 164,363 to 1,100,843.

Finally, LOLIB contains a set of graphs the authors called **Special**, which is itself a collection of instances used in other publications. The 37 graphs have varying sizes of as few as 11 up to 452 vertices and 55 to 2,996 arcs with a total arc weight between 55 and 2,884,574. Also the vertex degrees spread widely from  $0.002n$  up to  $0.99n$ .

#### 7.4.2 Performances and Running Times

We ran the algorithms **Psi** and **Fusion** on all instances of LOLIB. Heuristics similar to **Nest** have already been covered in the study by Martí and Reinelt [MR11]. **ELS** was only implemented for simple graphs. For this reason, we also omitted **ELS** as initial linear ordering and started either from an arbitrary, but fixed linear ordering (**Any**) or a random one (**Random**). The resulting four algorithm configurations **Psi | Any**, **Psi | Random**, **Fusion | Any**, and **Fusion | Random** were repeated five times on each graph.

Table 7.7 reports both on the algorithms' performances and running times. For each set of instances, we gathered four key figures: the mean deviation from the optimum or best known value for the set of backward arcs (dev. % ( $\mathcal{B}$ )), the same value for the set of forward arcs (dev. % ( $\mathcal{F}$ )), the number of times an algorithm found an optimum linear ordering or one matching the best known value of a graph in relation to the total number of graphs in the set, and the mean time the execution of the algorithm on a graph from the respective set needed. Due to the large diversity of the graphs even within the same test set and the concomitant variations in their respective linear orderings, the mean value of the obtained solutions is not very meaningful and was therefore omitted.

For the set **IO**, **Fusion | Random** was able to retrieve an optimum or best solution for 90% of the instances in the set, **Psi | Random** still achieved 84%. The deviations from the optimum or best known solutions are rather small and show a distinct improvement if the algorithms started from a random linear ordering instead of the one that was fixed arbitrarily. The performance results of **Psi | Any** and **Fusion | Any** are identical, and also differ only slightly for **Psi | Random** and **Fusion | Random**.

None of the algorithms was able to find an optimum or best solution for a graph of **RandA1**. Yet, the deviation from the optimum solution is less than 1.7% for the set of backward arcs and less than 1.1% for the set of forward arcs. The closeness of the respective deviations for the set of backward arcs and the set of forward arcs suggest that  $\tau$  is not too far from half of the total number of arcs. Again, the performances of **Psi** and **Fusion** are almost equal. The comparatively large size of the graphs in this set with up to 500 vertices together with a high density results in running times that are on

Table 7.7: Performances and running times on LOLIB instances.

	<i>Psi</i>   <i>Any</i>	<i>Fusion</i>   <i>Any</i>	<i>Psi</i>   <i>Random</i>	<i>Fusion</i>   <i>Random</i>
<b>LOLIB-IO</b>				
dev. % ( $\mathcal{B}$ )	1.73	1.73	(1.02, 0.16)	(0.92, 0.10)
dev. % ( $\mathcal{F}$ )	0.09	0.09	(0.05, 0.01)	(0.04, 0.00)
#hits of OPT/best	29 / 50	29 / 50	42 / 50	45 / 50
time	291.50 ms	985.28 ms	301.97 ms	1.01 s
<b>LOLIB-RandA1</b>				
dev. % ( $\mathcal{B}$ )	1.69	1.69	(1.63, 1.21)	(1.66, 1.24)
dev. % ( $\mathcal{F}$ )	1.03	1.03	(1.00, 0.75)	(1.02, 0.77)
#hits of OPT/best	0 / 100	0 / 100	0 / 100	0 / 100
time	7 min 12.81 s	50 min 19.77 s	6 min 30.58 s	48 min 58.73 s
<b>LOLIB-RandA2</b>				
dev. % ( $\mathcal{B}$ )	1.48	1.38	(1.54, 0.98)	(1.39, 0.82)
dev. % ( $\mathcal{F}$ )	0.11	0.10	(0.11, 0.07)	(0.10, 0.06)
#hits of OPT/best	0 / 75	0 / 75	0 / 75	0 / 75
time	18.97 s	1 min 46.26 s	13.60 s	1 min 34.12 s
<b>LOLIB-RandB</b>				
dev. % ( $\mathcal{B}$ )	2.29	2.27	(2.01, 0.85)	(2.05, 0.83)
dev. % ( $\mathcal{F}$ )	0.91	0.91	(0.80, 0.34)	(0.81, 0.34)
#hits of OPT/best	1 / 90	1 / 90	4 / 90	11 / 90
time	202.96 ms	952.24 ms	203.90 ms	937.40 ms
<b>LOLIB-SGB</b>				
dev. % ( $\mathcal{B}$ )	0.18	0.18	(0.48, 0.07)	(0.17, 0.01)
dev. % ( $\mathcal{F}$ )	0.07	0.07	(0.18, 0.03)	(0.07, 0.00)
#hits of OPT/best	2 / 25	2 / 25	7 / 25	12 / 25
time	1.02 s	4.49 s	1.23 s	5.42 s
<b>LOLIB-MB</b>				
dev. % ( $\mathcal{B}$ )	0.06	0.06	(0.09, 0.04)	(0.08, 0.03)
dev. % ( $\mathcal{F}$ )	0.01	0.01	(0.01, 0.00)	(0.01, 0.00)
#hits of OPT/best	1 / 30	3 / 30	6 / 30	5 / 30
time	28.53 s	2 min 40.38 s	20.92 s	2 min 31.99 s
<b>LOLIB-XLOLIB</b>				
dev. % ( $\mathcal{B}$ )	4.98	4.86	(4.82, 3.39)	(4.70, 3.34)
dev. % ( $\mathcal{F}$ )	1.42	1.39	(1.36, 0.98)	(1.34, 0.96)
#hits of OPT/best	0 / 78	0 / 78	0 / 78	0 / 78
time	2 min 38.0 s	6 min 52.15 s	2 min 32.1 s	6 min 36.53 s
<b>LOLIB-Special</b>				
dev. % ( $\mathcal{B}$ )	4.35	4.17	(3.72, 1.79)	(3.44, 1.45)
dev. % ( $\mathcal{F}$ )	1.00	0.96	(0.88, 0.49)	(0.85, 0.40)
#hits of OPT/best	4 / 37	4 / 37	6 / 37	6 / 37
time	646.07 ms	5.74 s	671.16 ms	5.80 s

average around 7 minutes for *Psi* and 50 minutes for *Fusion*. A maximum of 5 hours and 20 minutes was measured during the execution of *Fusion* | *Any* on a single instance.

For the graphs in *RandA2*, the deviation for the set of backward arcs is at most 1.54% and not far below those for *RandA1*, whereas it does not exceed 0.11% for the set of forward arcs. Even though, no algorithm produced an optimal or best solution. There also is a small improvement of performance visible between *Psi* and *Fusion*.

Despite the fact that the set *RandB* was compiled specifically to obtain “difficult” instances for the LINEAR ORDERING problem, both *Psi* and *Fusion* were at least able to reach the optimum or best solution on a few graphs. Regarding the deviation from the optimum or best solution, however, the observed values are among the larger ones. The performances of *Psi* and *Fusion* are almost equal.

The results on the instances from both *SGB* and *MB* show the smallest mean deviations of all sets in LOLIB. Nonetheless, the optimum or best hit rate remains below 50% for *SGB* and even below 20% for *MB*. Interestingly, the mean deviations for *IO* graphs are noticeably larger, whereas all algorithms found an optimum or best solution for at least 58% of the graphs.

Finally, the instances in the set *XLOLIB* and *Special* seem to have been the most challenging for the tested set of algorithms. The deviations from the optimum or best solution in case of the set of backward arcs reach values of almost 5% and 1.5 % in case of the set of forward arcs. For the graphs contained in *XLOLIB*, no algorithm could find an optimum or best solution. The ratio of optimum or best solutions found for instances from *Special* range between 10% and a little more than 16%.

All in all, the results show relatively small mean deviations from the optimum or best known solutions for *Psi* and the same or only slightly better ones for *Fusion*, both for *Any* and *Random* as initial linear ordering strategy. The mean deviation for the set of backward arcs is less than 5% on all sets, and in many cases even far below 2%. For the set of forward arcs, the mean deviation is for all sets except *RandA1* and *XLOLIB* at most 1%, and never exceeds 1.42%. In contrast to the benchmark results on sparser graphs, the observed performance of both *Psi* and *Fusion* shows a dependency on the initial linear ordering: The mean deviations from the optimum value differ—at times even considerably—for *Any* and *Random*, in most cases in favor of *Random*. By contrast, the performance gaps between *Psi* and *Fusion* are barely noticeable. Considering all sets of LOLIB, the mean deviations of the solutions obtained by *Psi* | *Any*, *Psi* | *Random*, *Fusion* | *Any*, and *Fusion* | *Random* for the set of forward arcs are 0.72%, 0.67%, 0.71%, and 0.67%, respectively.

### 7.4.3 Comparison to Other Approaches

In comparison to the results of the study reported on by Martí and Reinelt [MR11], we can observe that both *Psi* and *Fusion* perform in most cases markedly better than the set of heuristics included there, which comprises a number of construction heuristics as well as various local search approaches. On the other hand, some representatives of the meta-heuristics tested in their study like tabu search, variable neighborhood search, scatter search, or GRASP show mean deviations of 0.45% or less on average across all LOLIB instances with regard to the set of forward arcs. These findings suggest that the performance of *Psi* and *Fusion* resides somewhere between conventional insertion and local search heuristics and the mentioned meta-heuristics.

The performance figures on LOLIB instances show no distinct superiority of *Fusion* over the simpler and considerably less time-consuming *Psi* algorithm. As a clear dependency of the solution quality from the initial linear ordering was observable, it would be interesting to apply *Psi* as a local search heuristic in combination with the above mentioned meta-heuristics.

## 7.5 Threats to Validity

Assessing the performance of graph algorithms in practice calls for a number of decisions on the realization that may have major or minor influence on the results obtained.

### 7.5.1 Construct Validity

We mainly assessed the mean performance of the algorithms on a set of graphs. In consequence, differences in the quality of individual solutions may get lost in the evaluation of average values. To counteract this, we also included the best or OPT hit ratio of an algorithm, which reflects whether an algorithm often came very close to the optimum, but always missed it, or whether it really obtained optimal solutions. Furthermore, the quality of the input linear ordering naturally has an enormous impact on both the quality of the final solution and the running time of the algorithms *Nest*, *Psi*, *PsiAlt*, and *Fusion*. To preempt too much influence from this side, we used three different initializers. In this context, we also covered the performance of the algorithms if they start from different random linear orderings. Due to the large number of  $n!$  possible linear orderings for a graph of size  $n$ , five or ten repetitions may however well be too few to judge the performance, especially on large graphs.

### 7.5.2 Internal Validity

Both the algorithms and the specific data structures were not trimmed excessively to get the shortest possible running times. Thus, speedups of some computation times, accompanied by an equalization or a further diversification among the candidates, are possible. Moreover, there may be replacements for the standard data structures as well as the helper algorithms used which change the picture. This in principle also concerns the realization of weighted arcs as multiple parallel arcs. However, the overhead should be small due to their blockwise representation (cf. [Section 4.2](#)).

### 7.5.3 External Validity

Naturally, there must be a set of instances to run the algorithms on. As sparse real-world graphs are scarce and suitable data is difficult to find, we opted for a combination of different random generation processes on the one hand and the adaptation of a graph library that has already been used in a similar study on the other hand. Due to the method of construction, the graphs in the test sets **Cubic**, **Quartic**, and **Quintic** as well as **Facet** may show a bias with respect to certain properties such as in- and outdegree distributions or the intertwining of cycles, which may influence the performance of individual algorithms positively or negatively. In an attempt to reduce this risk, we implemented two different random orientators of an undirected regular graph for **Cubic**, **Quartic**, and **Quintic**. Furthermore, we included the set **Facet** of graphs that are both larger and were constructed by a radically different process. With regard to the adaptation of the LOLIB graph library, the graphs contained therein have been collected from a number of very different sources [[MR11](#)]. As such, even the instances in the same subset partly vary widely with respect to many properties such as size, density, or degree distributions.

## 7.6 Summary

In the previous sections, we compared the performances of the algorithms developed in this thesis to other established approaches as well as to the optimum solution. Especially on the tested sets of sparse input instances, **Psi**, **PsiAlt**, and **Fusion** were able to produce results that are on average better than those of their heuristic competitors and very close to the best ones possible. On dense and weighted graphs, their performance with respect to the percentage deviation from the optimum is numerically similar to that

on sparse graphs. However, the values of  $\tau$  and  $\bar{\tau}$  on dense and weighted instances are considerably larger, which makes the same absolute deviation appear smaller in relation. In consequence, low percentage deviations on sparse graphs are more difficult to achieve, a fact that speaks once again for the performance of *Psi*, *PsiAlt*, and *Fusion*.

In the comparison of the two variants of *PsiOpt*, *Psi* shows on average better results than *PsiAlt*, which might be due to the larger number of improving routines involved, which in turn could be speculated to have led to a wider search space. It would be interesting to see a detailed analysis of the reasons for the observable performance gap, though. The establishing of the Prefix Fusion Property and the Greedy Fusion Property in conjunction with *PsiOpt*, as realized in the algorithm *Fusion*, improved the results on sparse graphs once more in comparison to *Psi*, however only marginally. On some test sets from LOLIB, the performances of both algorithms were even equivalent. In view of this and the markedly increased time effort for *Fusion*, the practical usefulness must be weighed carefully according to the operational scenario.

Finally, the exact algorithm developed in [Chapter 6](#) has proven itself to be applicable in practice with very fast execution times on small instances and still acceptable ones for larger graphs. On cubic graphs, even input instances with far beyond 100 vertices did not pose a problem.





It comes to no surprise that the single chapters of this thesis answered some questions and in most cases immediately posed new ones. The survey in [Chapter 2](#) serves well as a starting point to walk briefly through the results. For instance, it lists a number of algorithms and heuristics for the LINEAR ORDERING problem that have been known for some time along with their performance guarantees. In [Chapter 4](#), we derived a series of sophisticated properties that led to a collection of new algorithms that solve the LINEAR ORDERING problem approximately ([Chapter 4](#)) and exactly ([Chapter 6](#)). Having a running time of  $\mathcal{O}(n \cdot m^2 \cdot \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$  for general graphs, the algorithm *PsiOpt*, which efficiently establishes all major properties concurrently, is asymptotically not much slower than pure *InsertionSort*-based approaches and undoubtedly fast enough to be run also on large instances, as demonstrated in [Chapter 7](#).

Concerning the maximum size of an optimal feedback arc set for a graph having  $n$  vertices and  $m$  arcs, which has also been covered in [Chapter 2](#), Eades *et al.* [[ELS93](#)] showed that it is at most  $\frac{n}{2} - \frac{n}{6}$  for general graphs and it is at most  $\frac{5}{18}m$  and  $\frac{11}{30}m$  for graphs with maximum vertex degree three and four, respectively, due to a result by Berger and Shor [[BS90](#)]. In [Chapter 4](#), we pointed out that the establishment of the Nesting Property alone already ensures that  $\frac{m}{2} - \frac{exc(G)}{2}$  is not exceeded, where  $exc(G)$  is the so-called excess of the input graph  $G$ , which is the larger the bigger the imbalance between the vertices' in- and outdegrees. What is more, we proved in [Chapter 5](#) with the help of *PsiOpt* that in case of subcubic graphs, i. e., those with a maximum vertex degree of three, an optimal feedback arc set cannot contain more than  $\lfloor \frac{n}{3} \rfloor$  arcs and that this is best possible. With a small extension of the set of properties enforced by *PsiOpt*, we also derived a tight upper bound of  $\lfloor \frac{2n}{3} \rfloor$  for the backward arc set for subquartic graphs, where the maximum vertex degree is four. Both results respectively improve the previously known upper bounds of  $\frac{5}{18}m = \frac{5}{12}n$  in the cubic case and  $\frac{11}{30}m = \frac{22}{30}n$  in the quartic one. Apart from the obvious open question about tight bounds for graphs with larger vertex degrees, [Chapter 5](#) also asks what can be achieved in comparison to the

optimal solution if we additionally take more high-level properties into account. Can this yield some kind of approximation algorithm?

In [Chapter 6](#), we devised two exact algorithms for the LINEAR ORDERING optimization problem and one for the decision problem, all of which are tailored specifically to graphs with very small maximum vertex degrees and require only relatively short execution times and polynomial space. In fact, they constitute the first exact algorithms for LINEAR ORDERING and FEEDBACK ARC SET that are designed for sparse graphs. On cubic graphs, the achieved running time is only  $\mathcal{O}^*(\sqrt{2}^n)$ . It would be interesting to see whether these algorithms can be sped up further by mixing them with the approaches used in other exact algorithms, such as dynamic programming or divide and conquer.

Finally, we showed in [Chapter 7](#) that the algorithms developed in this thesis are competitive with existing ones both with regard to performance and running time. As expected, their strengths lie in particular in the processing of sparse instances. Here, the exact representative exhibited fast execution times and sometimes even outran its polynomial-time competitors. It was able to process cubic input instances with 140 vertices in about 20 minutes on average. Conversely, the latter produced results that came very close to the optimum. On denser instances, *PsiOpt* and its siblings were no longer superior to other algorithms, but also not overly inferior. Upon considering that the best results on these graphs were obtained by local search and other meta-heuristics, the combination of *PsiOpt* with these paradigms may be promising. Furthermore, due to time and space constraints, we could not evaluate the algorithmic strengths of all properties introduced in this thesis and their combinations. Thus, a more large-scale evaluation of algorithms for the LINEAR ORDERING problem on an extended set of input instances suggests itself. Furthermore, an in-depth analysis of hill climbing algorithms such as *PsiOpt*, its alternative implementation that enforces only the Multipath Blocking Vertices Property and the Eliminator Layouts Property, as well as further representatives of this kind, which also includes the assessment of the decline rate of the solution size in the course of their operation, might reveal interesting facts.

## Bibliography

- [ACN08] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: Ranking and clustering. *Journal of the ACM*, 55(5):23:1–23:27, October 2008.
- [AFK02] Sanjeev Arora, Alan Frieze, and Haim Kaplan. A new rounding procedure for the assignment problem with applications to dense graph arrangement problems. *Mathematical Programming*, 92(1):1–36, March 2002.
- [Alo06] Noga Alon. Ranking tournaments. *SIAM Journal on Discrete Mathematics*, 20(1):137–142, 2006.
- [ALS09] Noga Alon, Daniel Lokshtanov, and Saket Saurabh. Fast FAST. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikolettseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, volume 5555 of *Lecture Notes in Computer Science*, pages 49–58. Springer, 2009.
- [BB04] Michael Baur and Ulrik Brandes. Crossing reduction in circular layouts. In Juraj Hromkovic, Manfred Nagl, and Bernhard Westfechtel, editors, *Graph-Theoretic Concepts in Computer Science, 30th International Workshop, WG 2004, Bad Honnef, Germany, June 21-23, 2004, Revised Papers*, volume 3353 of *Lecture Notes in Computer Science*, pages 332–343. Springer, 2004.
- [BFG<sup>+</sup>11] Stéphane Bessy, Fedor V. Fomin, Serge Gaspers, Christophe Paul, Anthony Perez, Saket Saurabh, and Stéphan Thomassé. Kernels for feedback arc set in tournaments. *Journal of Computer and System Sciences*, 77(6):1071–1078, 2011.
- [BFM94] Francisco Barahona, Jean Fonlupt, and Ali Ridha Mahjoub. Compositions of graphs and polyhedra IV: acyclic spanning subgraphs. *SIAM Journal on Discrete Mathematics*, 7(3):390–402, 1994.

- [BG02] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs - theory, algorithms and applications*. Springer, 2002.
- [BH11] Franz J. Brandenburg and Kathrin Hanauer. Sorting heuristics for the feedback arc set problem. Technical Report MIP-1104, Faculty of Computer Science and Mathematics, University of Passau, 2011.
- [BNP96] Alberto Borobia, Zeev Nutov, and Michal Penn. Doubly stochastic matrices and dicycle covers and packings in eulerian digraphs. *Linear Algebra and its Applications*, 246:361–371, 1996.
- [BS90] Bonnie Berger and Peter W. Shor. Approximation algorithms for the maximum acyclic subgraph problem. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California.*, pages 236–243. SIAM, 1990.
- [BSN15] Ali Baharev, Hermann Schichl, and Arnold Neumaier. An exact method for the minimum feedback arc set problem. [http://www.mat.univie.ac.at/~herman/fwf-P27891-N32/minimum\\_feedback\\_arc\\_set.pdf](http://www.mat.univie.ac.at/~herman/fwf-P27891-N32/minimum_feedback_arc_set.pdf), December 2015.
- [BT92] Jørgen Bang-Jensen and Carsten Thomassen. A polynomial algorithm for the 2-path problem for semicomplete digraphs. *SIAM Journal on Discrete Mathematics*, 5(3):366–376, August 1992.
- [CA90] Kwang-Ting Cheng and Vishwani D. Agrawal. A partial scan method for sequential circuits with feedback. *IEEE Trans. Computers*, 39(4):544–549, 1990.
- [CA95] Arun Chakradhar, Srimat T. and Balakrishnan and Vishwani D. Agrawal. An exact algorithm for selecting partial scan flip-flops. *Journal of Electronic Testing*, 7(1-2):83–93, 1995.
- [CDZ00] Mao-cheng Cai, Xiaotie Deng, and Wenan Zang. An approximation algorithm for feedback vertex sets in tournaments. *SIAM Journal on Computing*, 30(6):1993–2007, 2000.
- [CDZ02] Mao-Cheng Cai, Xiaotie Deng, and Wenan Zang. A min-max theorem on feedback vertex sets. *Mathematics of Operations Research*, 27(2):361–371, 2002.

- [CK96] Stefan Chanas and Przemysław Kobylański. A new heuristic algorithm solving the linear ordering problem. *Computational Optimization and Applications*, 6(2):191–205, 1996.
- [CLL07] Jianer Chen, Yang Liu, and Songjian Lu. Directed feedback vertex set problem is FPT. In Erik D. Demaine, Gregory Gutin, Dániel Marx, and Ulrike Stege, editors, *Structure Theory and FPT Algorithmics for Graphs, Digraphs and Hypergraphs, 08.07. - 13.07.2007*, volume 07281 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
- [CLL<sup>+</sup>08] Jianer Chen, Yang Liu, Songjian Lu, Barry O’Sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM*, 55(5), 2008.
- [Con06] Vincent Conitzer. Computing Slater rankings using similarities among candidates. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 613. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.
- [CTY07] Pierre Charbit, Stéphan Thomassé, and Anders Yeo. The minimum feedback arc set problem is  $\mathcal{NP}$ -hard for tournaments. *Combinatorics, Probability & Computing*, 16(1):1–4, 2007.
- [CW58] Hollis B. Chenery and Tsunehiko Watanabe. International comparisons of the structure of production. *Econometrica*, 26(4):487–521, 1958.
- [CW09] Tom Coleman and Anthony Wirth. Ranking tournaments: Local search and a new algorithm. *Journal of Experimental Algorithmics (JEA)*, 14:6, 2009.
- [dC85] M. le Marquis de Condorcet. Essai sur l’application de l’analyse à la probabilité des décisions rendues à la pluralité des voix. 1785.
- [DF03] Camil Demetrescu and Irene Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters*, 86(3):129–136, 2003.
- [DGH<sup>+</sup>10] Michael Dom, Jiong Guo, Falk Hüffner, Rolf Niedermeier, and Anke Truß. Fixed-parameter tractability results for feedback set problems in tournaments. *Journal of Discrete Algorithms*, 8(1):76–86, 2010.

- [Din70] E. A. Dinic. An algorithm for the solution of the max-flow problem with the polynomial estimation. *Doklady Akademii Nauk SSSR*, 194(4), 1970. In Russian. English translation: *Soviet Mathematics Doklady* 11, pages 1277–1280 (1970).
- [Din06] Yefim Dinitz. Dinitz' algorithm: The original version and even's version. In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman, editors, *Theoretical Computer Science: Essays in Memory of Shimon Even*, pages 218–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [DKNS01] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In Vincent Y. Shen, Nobuo Saito, Michael R. Lyu, and Mary Ellen Zurko, editors, *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*, pages 613–622. ACM, 2001.
- [dlV83] Wenceslas Fernandez de la Vega. On the maximum cardinality of a consistent set of arcs in a random tournament. *Journal of Combinatorial Theory, Series B*, 35(3):328–332, 1983.
- [dMNE99] Candido Ferreira Xavier de Mendonça Neto and Peter Eades. An improvement for an algorithm for finding a minimum feedback arc set for planar graphs. 21(4), 1999.
- [Dri80] T. Dridi. Sur les distributions binaires associées à des distributions ordinales. *Mathématiques et sciences humaines*, (69):15–31, 45, 1980.
- [DS05] Irit Dinur and Shmuel Safra. On the hardness of approximating Vertex Cover. *Annals of Mathematics*, 162(1):439–485, 2005.
- [Eck15] Barbara Eckl. Feedback Arc Set auf planaren Graphen. Bachelor's thesis, 2015.
- [EL95] Peter Eades and Xuemin Lin. A Heuristic for the Feedback Arc Set Problem. *Australasian Journal of Combinatorics*, 12:15–25, 1995.
- [ELS93] Peter Eades, Xuemin Lin, and William F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, October 1993.
- [EM65] Paul Erdős and J. W. Moon. On sets of consistent arcs in a tournament. *Canadian Mathematical Bulletin*, 8:269–271, 1965.

- [ENSS98] Guy Even, Joseph (Seffi) Naor, Baruch Schieber, and M. Sudan. Approximating minimum feedback sets and multicut in directed graphs. *Algorithmica*, 20(2):151–174, 1998.
- [FK99] Alan Frieze and Ravi Kannan. Quick approximation to matrices and applications. *Combinatorica*, 19(2):175–220, 1999.
- [FK10] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2010.
- [Flo90] Merrill M. Flood. Exact and heuristic algorithms for the weighted feedback arc set problem: A special case of the skew-symmetric quadratic assignment problem. *Networks*, 20(1):1–23, 1990.
- [Fra81] András Frank. How to make a digraph strongly connected. *Combinatorica*, 1(2):145–153, 1981.
- [Gav77] Fanica Gavril. Some  $\mathcal{NP}$ -complete problems on graphs. In *Proceedings of the 11th Conference on Information Sciences and Systems*, pages 91–95. Johns Hopkins University, 1977.
- [GGH<sup>+</sup>06] Jiong Guo, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *Journal of Computer and System Sciences*, 72(8):1386–1396, 2006.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of  $\mathcal{NP}$ -Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.
- [GJR84a] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32(6):1195–1220, 1984.
- [GJR84b] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. Optimal triangulation of large real world input-output matrices. *Statistische Hefte*, 25:261–295, 1984.
- [GJR85a] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. Facets of the linear ordering polytope. *Mathematical Programming*, 33(1):43–60, 1985.

- [GJR85b] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 33(1):28–42, 1985.
- [GKN98] E. Girlich, M. Kovalev, and V. Nalivaiko. A note on an extension of facet-defining digraphs. Technical report, 1998.
- [GMR08] Venkatesan Guruswami, Rajsekar Manokaran, and Prasad Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 573–582. IEEE Computer Society, 2008.
- [Gup08] Sushmita Gupta. Feedback arc set problem in bipartite tournaments. *Information Processing Letters*, 105(4):150–154, 2008.
- [Han10] Kathrin Hanauer. Algorithms for the feedback arc set problem. Master’s thesis, Faculty of Computer Science and Mathematics, University of Passau, Passau, Germany, 2010.
- [HBA13] Kathrin Hanauer, Franz J. Brandenburg, and Christopher Auer. Tight upper bounds for minimum feedback arc sets of regular graphs. In Andreas Brandstädt, Klaus Jansen, and Rüdiger Reischuk, editors, *Graph-Theoretic Concepts in Computer Science - 39th International Workshop, WG 2013, Lübeck, Germany, June 19-21, 2013, Revised Papers*, volume 8165 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2013.
- [Hel64] Ernst Helmstädter. Die Dreiecksform der Input-Output-Matrix und ihre möglichen Wandlungen im Wachstumsprozeß. In Fritz Neumark, editor, *Strukturwandlungen einer wachsenden Wirtschaft*. Schriften des Vereins für Socialpolitik, 1964.
- [Kaa81] R. Kaas. A branch and bound algorithm for the acyclic subgraph problem. *European Journal of Operational Research*, 8(4):355–362, 1981.
- [Kan92] Viggo Kann. *On the Approximability of  $\mathcal{NP}$ -complete Optimization Problems*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, May 1992.
- [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium*



- on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York., The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.*
- [Kar79] Alexander V. Karzanov. On the minimal number of arcs of a digraph meeting all its directed cutsets. *Graph Theory Newsletters*, 8(4), March 1979.
- [Kem59] John G. Kemeny. Mathematics without numbers. *Daedalus*, 88(4):577–591, 1959.
- [Kho02] Subhash Khot. On the power of unique 2-prover 1-round games. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 767–775. ACM, 2002.
- [Kho10] Subhash Khot. On the unique games conjecture (invited survey). In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity, CCC 2010, Cambridge, Massachusetts, June 9-12, 2010*, pages 99–121. IEEE Computer Society, 2010.
- [Knu09] Donald Ervin Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley Professional, 1st edition, 2009.
- [Koe05] Henning Koehler. A contraction algorithm for finding minimal feedback sets. In *Proceedings of the Twenty-eighth Australasian Conference on Computer Science - Volume 38, ACSC '05*, pages 165–173, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
- [KPPW15] Stefan Kratsch, Marcin Pilipczuk, Michał Pilipczuk, and Magnus Wahlström. Fixed-parameter tractability of multicut in directed acyclic graphs. *SIAM Journal on Discrete Mathematics*, 29(1):122–144, 2015.
- [KR08] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within  $2-\epsilon$ . *Journal of Computer and System Sciences*, 74(3):335–349, 2008.
- [KS06] Claire Kenyon-Mathieu and Warren Schudy. How to rank with few errors: A PTAS for weighted feedback arc set on tournaments. *Electronic Colloquium on Computational Complexity (ECCC)*, 13(144), 2006.

- [KS07] Claire Kenyon-Mathieu and Warren Schudy. How to rank with few errors. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 95–103. ACM, 2007.
- [KS10] Marek Karpinski and Warren Schudy. Faster algorithms for feedback arc set tournament, kemeny rank aggregation and betweenness tournament. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I*, volume 6506 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2010.
- [KST90] Philip Klein, Clifford Stein, and Éva Tardos. Leighton-Rao might be practical: Faster approximation algorithms for concurrent flow with uniform capacities. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing, STOC '90*, pages 310–321, New York, NY, USA, 1990. ACM.
- [Lem08] Mile Lemaic. *Markov-Chain-Based Heuristics for the Feedback Vertex Set Problem for Digraphs*. PhD thesis, 2008.
- [LG14] François Le Gall. Algebraic complexity theory and matrix multiplication. In Katsusuke Nabeshima, Kosaku Nagasaka, Franz Winkler, and Ágnes Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, page 23. ACM, 2014.
- [LR88] Thomas Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 422–431, Oct 1988.
- [Luc76] Cláudio Leonardo Lucchesi. *A minimax equality for directed graphs*. PhD thesis, 1976.
- [LY78] Cláudio Leonardo Lucchesi and Daniel Haven Younger. A minimax theorem for directed graphs. *Journal of the London Mathematical Society*, 17:369–374, 1978.
- [MB00] John E. Mitchell and Brian Borchers. Solving linear ordering problems with a combined interior point/simplex cutting plane algorithm. In H. L. Frenk,

- C. Roos, T. Terlaky, and S. Zhang, editors, *High Performance Optimization*, chapter 14, pages 349–366. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.
- [MR11] Rafael Martí and Gerhard Reinelt. *The Linear Ordering Problem. Exact and Heuristic Methods in Combinatorial Optimization*, volume 175 of *Applied Mathematical Sciences*. Springer Berlin Heidelberg, 2011.
- [Mus05] A. R Mushi. The linear ordering problem: An algorithm for the optimal solution. 6(1):51–64, June 2005.
- [MWV15] Matthias Mnich, Virginia Vassilevska Williams, and László A. Végh. A  $7/3$ -approximation for feedback vertex sets in tournaments. *CoRR*, abs/1511.01137, 2015.
- [NP95] Zeev Nutov and Michal Penn. On the integral dicycle packings and covers and the linear ordering polytope. *Discrete Applied Mathematics*, 60(1-3):293–309, 1995.
- [Pap91] Mihalis Papadimitriou, Christos H. and Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991.
- [PS98] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.
- [PVP13] Kévin Perrot and Trung Van Pham.  $\mathcal{NP}$ -hardness of minimum feedback arc set problem on Eulerian digraphs and minimum recurrent configuration problem of chip-firing game. *CoRR*, abs/1303.3708, 2013.
- [Ram88] Vijaya Ramachandran. Finding a minimum feedback arc set in reducible flow graphs. *Journal of Algorithms*, 9(3):299–313, 1988.
- [Ram90] Vijaya Ramachandran. A minimax arc theorem for reducible flow graphs. *SIAM Journal on Discrete Mathematics*, 3(4):554–560, 1990.
- [Rei93] Gerhard Reinelt. A note on small linear-ordering polytopes. *Discrete & Computational Geometry*, 10:67–78, 1993.
- [RS03] Venkatesh Raman and Saket Saurabh. Parameterized complexity of directed feedback set problems in tournaments. In Frank K. H. A. Dehne, Jörg-Rüdiger

- Sack, and Michiel H. M. Smid, editors, *Algorithms and Data Structures, 8th International Workshop, WADS 2003, Ottawa, Ontario, Canada, July 30 - August 1, 2003, Proceedings*, volume 2748 of *Lecture Notes in Computer Science*, pages 484–492. Springer, 2003.
- [RS06] Venkatesh Raman and Saket Saurabh. Parameterized algorithms for feedback set problems and their duals in tournaments. *Theoretical Computer Science*, 351(3):446–458, 2006. Parameterized and Exact Computation First International Workshop on Parameterized and Exact Computation 2004.
- [RS07] Venkatesh Raman and Saket Saurabh. Improved fixed parameter tractable algorithms for two "edge" problems: MAXCUT and MAXDAG. *Inf. Process. Lett.*, 104(2):65–72, 2007.
- [RSS07] Venkatesh Raman, Saket Saurabh, and Somnath Sikdar. Efficient exact algorithms through enumerating maximal independent sets and other techniques. *Theory Comput. Syst.*, 41(3):563–587, 2007.
- [Saa01] Youssef Saab. A fast and effective algorithm for the feedback arc set problem. *Journal of Heuristics*, 7(3):235–250, 2001.
- [Sas08a] Prashant Sasatte. Improved approximation algorithm for the feedback set problem in a bipartite tournament. *Operations Research Letters*, 36(5):602–604, 2008.
- [Sas08b] Prashant Sasatte. Improved {FPT} algorithm for feedback vertex set problem in bipartite tournament. *Information Processing Letters*, 105(3):79–82, 2008.
- [Sch13] Jens M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241 – 244, 2013.
- [Sey95] Paul D. Seymour. Packing directed circuits fractionally. *Combinatorica*, 15(2):281–288, 1995.
- [Sey96] Paul D. Seymour. Packing circuits in Eulerian digraphs. *Combinatorica*, 16(2):223–231, 1996.
- [Sha79] Adi Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal on Computing*, 8(4):645–655, 1979.

- [Ski08] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London, 2008.
- [Sla61] Patrick Slater. Inconsistencies in a schedule of paired comparisons. *Biometrika*, 48(3-4):303–312, December 1961.
- [Spe71] Joel Spencer. Optimal ranking of tournaments. *Networks*, 1(2):135–138, 1971.
- [Spe80] Joel Spencer. Optimally ranking unrankable tournaments. *Periodica Mathematica Hungarica*, 11(2):131–144, 1980.
- [Spe89] Ewald Speckenmeyer. On feedback problems in digraphs. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG '89, Castle Rolduc, The Netherlands, June 14-16, 1989, Proceedings*, volume 411 of *Lecture Notes in Computer Science*, pages 218–231. Springer, 1989.
- [SS04] Tommaso Schiavinotto and Thomas Stützle. The linear ordering problem: Instances, search space analysis and algorithms. *Journal of Mathematical Modelling and Algorithms*, 3(4):367–402, 2004.
- [Sta90] Hermann Stamm. On feedback problems in planar digraphs. In Rolf H. Möhring, editor, *Graph-Theoretic Concepts in Computer Science, 16rd International Workshop, WG '90, Berlin, Germany, June 20-22, 1990, Proceedings*, volume 484 of *Lecture Notes in Computer Science*, pages 79–89. Springer, 1990.
- [STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.
- [SW93] Hermann Stamm-Wilbrandt. Programming in propositional logic or reductions: Back to the roots (satisfiability). Technical report, Universität Bonn, 1993.
- [SW99] Angelika Steger and Nicholas C. Wormald. Generating random regular graphs quickly. *Comb. Probab. Comput.*, 8(4):377–396, July 1999.
- [Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

- [vZ11] Anke van Zuylen. Linear programming based approximation algorithms for feedback set problems in bipartite tournaments. *Theoretical Computer Science*, 412(23):2556–2561, 2011.
- [vZW09] Anke van Zuylen and David P. Williamson. Deterministic pivoting algorithms for constrained ranking and clustering problems. *Mathematics of Operations Research*, 34(3):594–620, 2009.
- [Yan78] Mihalis Yannakakis. Node- and edge-deletion NP-complete problems. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pages 253–264. ACM, 1978.
- [You63] Daniel Haven Younger. Minimum Feedback Arc Sets for a Directed Graph. *IEEE Transactions on Circuit Theory*, 10(2):238–245, June 1963.
- [YXS06] Yao Yu, Chen Xinmeng, and Zhu Shanfeng. Rank aggregation algorithms based on voting model for metasearch. In *Wireless Communications, Networking and Mobile Computing, 2006. WiCOM 2006. International Conference on*, pages 1–4. IEEE, 2006.

## Notation Index

- $(u, v)$   
an arc with tail  $u$  and head  $v$ , **38**
- $(u, v)^R$   
the reverse of  $(u, v)$ ,  $(u, v)^R = (v, u)$ ,  
**38**
- $u \rightsquigarrow v$   
a path from  $u$  to  $v$  (short form), **40**
- $X \subseteq^* U$   
 $X$  is a multisubset of  $U$ , i. e.,  
 $X = (U', m)$  and  $U' \subseteq U$ , **36**
- $X \uplus Y$   
the multiset sum of  $X$  and  $Y$ , **36**
- $\langle \mathcal{B}^+ \rangle_\pi(v)$   
a list containing the elements of  $\mathcal{B}^+(v)$   
sorted according to the position of  
their heads in  $\pi$ , **60**
- $\langle \mathcal{B}^- \rangle_\pi(v)$   
a list containing the elements of  $\mathcal{B}^-(v)$   
sorted according to the position of  
their tails in  $\pi$ , **60**
- $\langle \mathcal{F}^+ \rangle_\pi(v)$   
a list containing the elements of  $\mathcal{F}^+(v)$   
sorted according to the position of  
their heads in  $\pi$ , **60**
- $\langle \mathcal{F}^- \rangle_\pi(v)$   
a list containing the elements of  $\mathcal{F}^-(v)$   
sorted according to the position of  
their tails in  $\pi$ , **60**
- $\langle v_0, \dots, v_k \rangle$   
a path from  $v_0$  to  $v_k$ , **40**
- $[\pi]_{\sim_{\mathcal{B}}}$   
the set of linear orderings inducing the  
same set of backward arcs as  $\pi$ , **46**
- $A$   
the set of arcs, **38**
- $A^R$   
the set containing the reverse of each  
arc in  $A$ , **38**
- $\mathcal{B}$   
a feedback arc set, **43**  
often used as simplification for  $\mathcal{B}_\pi$ , **47**
- $\mathcal{B}(v)$   
the set of backward arcs incident to  $v$ ,  
**47**
- $b(v)$   
the total number of backward arcs  
incident to  $v$ , **47**
- $\mathcal{B}^+(v)$   
the set of backward arcs outgoing from  
 $v$ , **47**
- $b^+(v)$

- the number of backward arcs outgoing from  $v$ , [47](#)
- $\mathcal{B}^-(v)$   
the set of backward arcs incoming to  $v$ , [47](#)
- $b^-(v)$   
the number of backward arcs incoming to  $v$ , [47](#)
- $\mathcal{B}_\pi$   
the set of backward arcs induced by  $\pi$ , [44](#)
- NoBlock( $\pi$ )  
a predicate that indicates whether or not  $\pi$  respects the Blocking Vertices Property, [82](#)
- C  
a cycle, [40](#)
- connect( $G, u, v$ )  
the graph obtained by adding a new arc  $(u, v)$  to  $G$ , [138](#)
- $d(v)$   
 $v$ 's degree, [40](#)
- $d^+(v)$   
 $v$ 's outdegree, [39](#)
- $d^-(v)$   
 $v$ 's indegree, [40](#)
- $\Delta_G$   
the maximum degree of a vertex in a graph  $G$ , [40](#)
- $\delta(v)$   
 $v$ 's delta degree, [40](#)  
 $\delta(v) = d^+(v) - d^-(v)$ , see also  $d^+(v)$  and  $d^-(v)$
- Elim( $\pi$ )  
a predicate that indicates whether or not  $\pi$  respects the Eliminateable Layouts Property, [117](#)
- elim $_q(\pi)$   
function that applies the elimination operation to the vertex  $\pi^{-1}(q)$  and returns the resulting linear ordering if this vertex has an eliminateable layout, otherwise, it is the identity function, [117](#)
- exc( $G$ )  
the excess of  $G$ , [72](#)
- $\mathcal{F}$   
an acyclic arc set, [43](#)  
often used as simplification for  $\mathcal{F}_\pi$ , [47](#)
- $\mathcal{F}(v)$   
the set of forward arcs incident to  $v$ , [47](#)
- $f(v)$   
the total number of forward arcs incident to  $v$ , [47](#)
- $\mathcal{F}^+(v)$   
the set of forward arcs outgoing from  $v$ , [47](#)
- $f^+(v)$   
the number of forward arcs outgoing from  $v$ , [47](#)
- $\mathcal{F}^-(v)$   
the set of forward arcs incoming to  $v$ , [47](#)
- $f^-(v)$   
the number of forward arcs incoming to  $v$ , [47](#)
- $\mathcal{F}_\pi$   
the set of forward arcs induced by  $\pi$ , [44](#)



- $\text{fuse}(G, \pi, q)$   
*the graph and linear ordering obtained by a fusion of the vertices at position  $q$  and  $q + 1$  in  $\pi$ , 142*
- $G$   
*a graph, 38*  
 $G = (V, A)$ , see also  $A$  and  $V$
- $G^R$   
*the reverse of  $G, G^R = (V, A^R)$ , 38*
- $G|_{\mathcal{F}}$   
*the acyclic subgraph of  $G$ , 43*
- $G_{\text{sp}}$   
*a general split graph of  $G$ , without indicating what vertices are split, 84*
- $G_{\text{sp}/l}$   
*the left-blocking split graph of  $G$ , 83*
- $G_{\text{sp}/r}$   
*the right-blocking split graph of  $G$ , 83*
- $G_X$   
*the subgraph induced by the vertex set  $X \subseteq V$ , 39*
- $G|_Y$   
*the subgraph of  $G$  restricted to the arc set  $A \cap Y$ , 39*
- $\overset{\sim}{G}$   
*a forward path graph of  $G$  with respect to a  $\Psi$ -optimal linear ordering, 152*
- $\overset{\sim}{G}^\circ$   
*a pooled forward path graph of  $G$  with respect to a  $\Psi$ -optimal linear ordering, 154*
- $\overset{\sim}{G}^\emptyset$   
*a polarized forward path graph of  $G$  with respect to a  $\Psi$ -optimal linear ordering, 156*
- $\overset{\sim}{G}_{\text{tr}}$   
*a truncated forward path graph of  $G$  with respect to a  $\Psi$ -optimal linear ordering, 157*
- $\mathcal{I}_\pi(q)$   
*the sequence of vertices in the order they occur at position  $q$  starting from  $\pi$ , 118*
- $\text{insert}(\pi, u, q)$   
*the linear ordering obtained by inserting vertex  $u$  at position  $q$  in  $\pi$ , 134*
- $\text{insert}(\pi, u, q, H)$   
*= reinterpret(insert( $\pi, u, q$ ),  $H$ ), 134*
- $L(G)$   
*the line graph of  $G$ , 15*
- $l(\pi)$   
*the length of an (incomplete) linear ordering  $\pi$ , 200*
- $\mathcal{L}(v)$   
*the layout of vertex  $v$ , 47*
- $m$   
*the number of arcs, 38*  
 $m = |A|$ , see also  $A$
- $\text{MNoBlock}(\pi)$   
*a predicate that indicates whether or not  $\pi$  respects the Multipath Blocking Vertices Property, 103*
- $\text{MPath}(\pi)$

- a predicate that indicates whether or not  $\pi$  respects the Multipath Property, **90**
- $n$   
the number of vertices, **38**  
 $n = |V|$ , see also  $V$
- $N^+(v)$   
the set of vertices with an incoming arc from  $v$ , **39**
- $N^-(v)$   
the set of vertices with an outgoing arc to  $v$ , **39**
- $Nest(\pi)$   
a predicate that indicates whether or not  $\pi$  respects the Nesting Property, **70**
- $neutralize(G, (u, v), (v, u))$   
the graph obtained by removing the pair of anti-parallel arcs  $(u, v)$  and  $(v, u)$  from  $G$ , **140**
- $\nu_G$   
the cardinality of an optimal set of arc-disjoint cycles of  $G$ , **13**  
the objective function value of the ADC 0-1 integer linear program, **13**  
 $\nu_G \leq \tau_G$ , see also  $\tau_G$
- $\nu_G^*$   
the objective function value of the ADC linear programming relaxation, **13**  
 $\nu_G^* = \tau_G^*$ , see also  $\tau_G^*$   
 $\nu_G^* \geq \nu_G$ , see also  $\nu_G$
- $\mathcal{O}$   
 $f \in \mathcal{O}(g)$ :  $f$  asymptotically grows at most as fast as  $g$ , **37**
- $o$   
 $f \in o(g)$ :  $f$  asymptotically grows strictly slower than  $g$ , **37**
- $\mathcal{O}^*$   
as  $\mathcal{O}$ , but ignores polynomially bounded factors, **37**
- $\Omega$   
 $f \in \Omega(g)$ :  $f$  asymptotically grows at least as fast as  $g$ , **37**
- $Opt(\pi)$   
a predicate that indicates whether or not  $\pi$  is optimal, **56**
- $P$   
a path, **40**
- $P_b$   
a forward path for backward arc  $b$ , **73**
- $P_{b_{crop}}$   
a cropped forward path for backward arc  $b$ , **78**
- $Path(\pi)$   
a predicate that indicates whether or not  $\pi$  respects the Path Property, **74**
- $\pi$   
a linear ordering, **44**
- $|\pi|$   
the cardinality of the induced feedback arc set, **46**
- $\pi^R$   
the reverse of  $\pi$ , **46**
- $\pi_{sp}$   
the linear ordering belonging to  $G_{sp}$ , **84**
- $\pi_{sp/l}$

- the linear ordering belonging to  $G_{\text{sp}/l}$ ,  
83
- $\pi_{\text{sp}/r}$   
the linear ordering belonging to  $G_{\text{sp}/r}$ ,  
83
- $\Psi_{\text{opt}}(\pi)$   
a predicate that indicates whether or  
not  $\pi$  respects all properties of  
Chapter 4, 56
- $\text{reinterpret}(\pi, H)$   
the linear ordering obtained by  
interpreting  $\pi$  as a linear ordering  
of a graph  $H$ , 133
- $\text{skip}(\pi, u)$   
the linear ordering obtained by  
removing vertex  $u$  from  $\pi$ , 134
- $\text{skip}(\pi, u, H)$   
=  $\text{reinterpret}(\text{skip}(\pi, u), H)$ , 134
- $\tau_G$
- cardinality of an optimal feedback arc  
set of  $G$ , 44
- the objective function value of the FAS  
0-1 integer linear program, 10
- $\tau_G \geq \nu_G$ , see also  $\nu_G$
- $\tau_G^*$   
the objective function value of the FAS  
linear programming relaxation, 11
- $\tau_G^* = \nu_G^*$ , see also  $\nu_G^*$
- $\tau_G^* \leq \tau_G$ , see also  $\tau_G$
- $\Theta$   
 $f \in \Theta(g)$ :  $f$  asymptotically grows as  
fast as  $g$ , 37
- $\xi(v)$   
the topsort number of  $v$ , 42
- $V$   
the set of vertices, 38
- $W$   
a walk, 41



# Algorithm Index

*Cascade*( $G, \pi, L$ )

*calls the property-enforcing routines contained in  $L$  in a specific order until all properties are established, 128*

*CheckPartialLayout*( $G, \lambda$ )

*checks a partial layout  $\lambda$  and, if possible, constructs a linear ordering realizing it, 202*

*ComputeLayoutLists*( $G, \pi$ )

*computes the ordered lists of incoming/outgoing forward/backward arcs, 61*

*ComputePositionsAndArcSets*( $G, \pi$ )

*computes the sets of induced forward and backward arcs, 60*

*DecideLO*( $G, k$ )

*decides whether  $G$  has a linear ordering inducing at most  $k$  backward arcs, 207*

*EliminateLayouts*( $G, \pi$ )

*returns  $\pi$ , if  $\text{Elim}(\pi)$ , otherwise a possibly improvable linear ordering, 123*

*EnforceForwardPaths*( $G, \pi$ )

*returns  $\pi$ , if  $\text{Path}(\pi)$ , otherwise an improved linear ordering, 75*

*EnforceMultiPaths*( $G, \pi$ )

*returns  $\pi$ , if  $\text{MPath}(\pi)$ , otherwise an improved linear ordering, 92*

*EnforceMultiPathsNoBlocking*( $G, \pi$ )

*returns  $\pi$ , if  $\text{MNoBlock}(\pi)$ , otherwise an improved linear ordering, 104*

*EnforceNesting*( $G, \pi$ )

*returns  $\pi$ , if  $\text{Nest}(\pi)$ , otherwise an improved linear ordering, 65*

*EnforceNoBlocking*( $G, \pi$ )

*returns  $\pi$ , if  $\text{NoBlock}(\pi)$ , otherwise an improved linear ordering, 85*

*EstablishForwardPaths*( $G, \pi$ )

*establishes the Path Property on  $\pi$ , 76*

*EstablishMultiPaths*( $G, \pi$ )

*establishes the Multipath Property on  $\pi$ , 94*

*EstablishMultiPathsNoBlocking*( $G, \pi$ )

*establishes the Multipath Blocking Vertices Property on  $\pi$ , 107*

*EstablishNesting*( $G, \pi$ )

*establishes the Nesting Property on  $\pi$ , 68*

*EstablishNoBlocking*( $G, \pi$ )

*establishes the Blocking Vertices Property on  $\pi$ , 87*

*ExactLOIntegrated*( $G$ )

*computes an optimal linear ordering  $\pi$  of  $G$  (integrated version), 214*

*ExactLOSimple*( $G$ )

*computes an optimal linear ordering  $\pi$  of  $G$  (simple version), 204*

*ExtendLO*( $G, \pi, t, E, Z, D$ )

*extends an incomplete linear ordering  $\pi$  optimally, 211*

*Iterate*( $G, \pi, E$ )

*repeatedly calls a property-enforcing routine  $E(G, \pi)$  until the property is established, 57*

*Merge*( $L_1, f_1, L_2, f_2$ )

*merges two lists  $L_1$  and  $L_2$  as in the MergeSort algorithm by applying the functions  $f_1$  and  $f_2$  to the elements of  $L_1$  and  $L_2$ , respectively, and comparing the results., 66*

*MinCut*( $G, s, t$ )

*computes a minimum  $s - t$  cut in  $G$  with unit capacities and returns the arcs in the cut-set, 91*

*Move*( $\pi, v, p$ )

*moves vertex  $v$  to position  $p$  within  $\pi$ , 63*

*PsiOpt*( $G, \pi$ )

*establishes the Nesting Property, the Path Property, the Blocking Vertices Property, the Multipath Property, the Multipath Blocking Vertices Property, and the Eliminateable Layouts Property simultaneously on  $\pi$ , 130*

*PsiOptCubic*( $G, \pi$ )

*establishes all properties of Chapter 4 simultaneously on  $\pi$  if  $G$  is subcubic, 132*

*Reclassify*( $G, u, v, \pi, E, Z, D$ )

*extends an incomplete linear ordering  $\pi$  optimally, 212*

*SimplifyGraph*( $G$ )

*computes the simple subgraph of  $G$ , 74*

*SplitVertically*( $G, \pi, X$ )

*applies a vertical split to all vertices in  $X$  and returns the resulting graph and linear ordering, 84*

*TopSort*( $G$ )

*computes a topological sorting of an acyclic graph  $G$ , 43*

*TransitiveClosure*( $G$ )

*computes the transitive closure of  $G$ , 75*





# Subject Index

Numbers printed in **bold face** indicate the pages where the term is introduced.

- acyclic arc set, **43, 45, 46**
- acyclic graph, *see* graph ↪ acyclic
- ACYCLIC SUBGRAPH, **11, 18**
- Acyclic Subgraph (decision), **24**
- ACYCLIC SUBGRAPH (DECISION), **11**
- acyclic subgraph polytope, **13, 226**
- ADC, *see* ARC-DISJOINT CYCLES
- adjacent, *see* vertex ↪ adjacent
- admissible, **169**
- Alternative Forward Paths Property, **173**
- arc
  - anti-parallel, **38, 133**
  - backward, **44**
  - contraction, **136**
  - forward, **44, 47**
  - incident, **38**
  - incoming, **38**
  - insertion, **138**
  - neutralization, **140**
  - outgoing, **38**
  - parallel, **38**
  - reverse, **38, 46, 140**
  - subdivision, **139**
- arc contraction, *see* arc ↪ contraction
- arc insertion, *see* arc ↪ insertion
- arc neutralization, *see* arc ↪
  - neutralization
- arc set, **38, 39**
  - feasible, **43, 79, 95, 108**
- Arc Stability Property, **149**
- arc subdivision, *see* arc ↪ subdivision
- ARC-DISJOINT CYCLES, **13, 18**
- AS, *see* ACYCLIC SUBGRAPH
- balance, **209**
- biconnected component, *see* block, **41**
- block, **41, 50**
- block-cut tree, **41**
- Blocking Vertices Property, **80**
- BOOLEAN SATISFIABILITY, **17**
- cardinality, **35, 37, 38**
- CHORD SET, **9**
- component vertex, *see* vertex ↪
  - component
- condensation, **42, 49**
- connected component, **41**
- cubic graph, *see* graph ↪ cubic
- cut vertex, **41**
- cycle, **40**
  - arc-disjoint, **40**

- simple, [40, 50](#)
  - vertex-disjoint, [40](#)
- CYCLE COVER, [9](#)
- cycle cover, [43](#)
- CYCLE HITTING SET, [9](#)
- dAS, *see* ACYCLIC SUBGRAPH (DECISION)
- degree, [40](#)
  - delta degree, [40, 160](#)
  - indegree, [40](#)
  - outdegree, [39](#)
- delta degree, *see* degree ↪ delta degree
- dFAS, *see* FEEDBACK ARC SET (DECISION)
- digraphs, [38](#)
- directed graphs, [38](#)
- DIRECTED MULTICUT, [17, 97, 112](#)
- dLO, *see* LINEAR ORDERING (DECISION)
- DMC, *see* DIRECTED MULTICUT
- Eliminable Layouts Property, [113, 178](#)
- excess, [72](#)
- Extended Multipath Blocking Vertices Property, [110](#)
- Extended Multipath Property, [96](#)
- facet-defining, [14, 225](#)
- FAS, *see* FEEDBACK ARC SET
- feasible, *see* arc set ↪ feasible, [77](#)
- FEEDBACK ARC SET, [8](#)
- feedback arc set, [43, 45](#)
  - minimal, *see also* set ↪ minimal, [43, 73, 77](#)
  - minimum, *see also* set ↪ minimum, [44](#)
  - optimal, [44](#)
- FEEDBACK ARC SET (DECISION), [8](#)
- FEEDBACK CUTSET, [9](#)
- FEEDBACK VERTEX SET, [15, 18](#)
- fixed-parameter tractable, [23](#)
- forward path, [47, 73, 74, 77](#)
  - cropped, [78, 132](#)
  - for a backward arc, [73](#)
- forward path graph, [73, 152](#)
  - polarized, [156, 169, 180](#)
  - pooled, [154, 170, 179](#)
  - truncated, [157, 160, 169](#)
- FPT*, *see* fixed parameter tractable
- fractional solution, [11, 22](#)
- FVS, *see* FEEDBACK VERTEX SET
- graph, [38, 43](#)
  - acyclic, [42, 43](#)
  - biconnected, [41](#)
  - bipartite, [6, 32, 39](#)
  - bipartite tournament, [6, 20–24, 34, 39, 225](#)
  - complete, [5](#)
  - connected, [41](#)
  - cubic, [27, 29, 40, 225](#)
  - Eulerian, [5, 25](#)
  - linklessly embeddable, [20, 25](#)
  - multigraph, *see* multigraph
  - quartic, [40, 225](#)
  - quintic, [40](#)
  - regular, [40](#)
  - reverse, [38, 44, 46, 58](#)
  - simple, [39, 48, 151, 198](#)
  - strongly connected, [41, 50](#)
  - strongly cyclic, [14](#)

- subcubic, [2](#), [40](#), [131](#), [159](#)
  - subgraph, [39](#)
    - acyclic subgraph, [43](#)
    - induced, [39](#)
    - restricted, [39](#), [43](#)
    - simple, [74](#), [85](#)
    - source-preserving, [158](#), [169](#)
    - spanning, [39](#), [152](#)
  - subquartic, [3](#), [40](#)
  - subquintic, [3](#), [40](#)
  - subregular, [40](#)
  - tournament, [6](#), [10–14](#), [19–24](#), [26](#), [29–31](#), [34](#), [39](#)
  - weakly acyclic, [14](#), [20](#)
- head, [38](#)
- incident, *see* [arc](#) ↪ incident
- incoming, *see* [arc](#) ↪ incoming
- indegree, *see* [degree](#) ↪ indegree
- Independent Set Reduction Property, [148](#)
- induced subgraph, *see* [graph](#) ↪ subgraph  
↪ induced
- isolated vertex, *see* [vertex](#) ↪ isolated
- $k$ -fence, [14](#), [161](#), [225](#)
- $k$ -opt, [63](#), [149](#)
- $k$ -regular, *see* [graph](#) ↪ regular
- KEMENY RANKING, [30](#)
- Kendall tau distance, [31](#)
- LABEL COVER, [22](#)
- layout, [47](#)
  - eliminable, [112](#)
  - partial, [198](#)
- left-blocking split graph, [83](#)
- length (list), [38](#)
- length (sequence), [36](#)
- line graph, [15](#), [20](#)
- LINEAR ORDERING, [7](#), [203](#), [213](#)
- linear ordering, [6](#), [44](#)
  - incomplete, [200](#)
    - containment, [200](#)
    - extension, [200](#)
    - length, [200](#)
  - optimal, [46](#)
  - $\Psi$ -optimal, [56](#), [131–132](#), [160](#), [178](#), [195](#)
  - reverse, [46](#), [59](#)
- LINEAR ORDERING (DECISION), [8](#), [207](#)
- linear ordering polytope, [14](#)
- linear program, [9](#)
- linear programming relaxation, [10](#)
- LO, *see* LINEAR ORDERING
- local search, [63](#), [149](#)
- loop, [38](#), [45](#), [48](#), [133](#)
- LO position, [44](#), [200](#)
- Möbius ladder, [14](#), [225](#)
- MATRIX TRIANGULATION, [9](#)
- meta-property, [56](#)
- multigraph, [39](#)
- Multipath Blocking Vertices Property, [98](#)
- Multipath Property, [88](#)
- multiset, [36](#), [38](#)
- multiset sum, [36](#)
- multisubset, [36](#)
- Nesting Property, [63](#), [69](#), [178](#)
- neutralization, *see* [arc](#) ↪ neutralization
- One-Arc Stability Property, [149](#), [173](#), [176](#)

- ONE-SIDED CROSSING MINIMIZATION, **32**
- outdegree, *see* degree ↪ outdegree
- outgoing, *see* arc ↪ outgoing
- parameterized complexity, **23**
- partial
- layout
    - realizable, **198**
    - realization, **198**
- passage, *see* vertex ↪ passage
- path, **40**
- arc-disjoint, **40, 87**
  - forward, *see* forward path
  - pseudosink, **178, 185**
  - simple, **40**
  - subpath, **40**
  - vertex-disjoint, **40**
- Path Property, **74, 153**
- polarization, **156**
- pooling, **154**
- pseudosink, *see* vertex ↪ pseudosink
- pseudosource, *see* vertex ↪ pseudosource
- quartic graph, *see* graph ↪ quartic
- quintic graph, *see* graph ↪ quintic
- RANK AGGREGATION, **30**
- reducible flow graph, **25**
- Reduction Property, **147**
- regular graph, *see* graph ↪ regular
- restricted subgraph, *see* graph ↪ subgraph ↪ restricted
- right-blocking split graph, **83, 152**
- SAT, *see* Boolean Satisfiability
- SCC, *see* strongly connected component
- sequence, **36, 38**
- set, **35**
- maximal, **37, 41**
  - maximum, **37**
  - minimal, **37**
  - minimum, **37**
  - simple, **35, 36, 37, 39**
- simple cycle, *see* cycle ↪ simple
- simple graph, *see* graph ↪ simple
- simple path, *see* path ↪ simple
- simple subgraph, *see* graph ↪ subgraph ↪ simple
- sink, *see* vertex ↪ sink
- SLATER RANKING, **31**
- source, *see* vertex ↪ source
- spanning subgraph, *see* graph ↪ subgraph ↪ spanning
- strongly connected component, **41, 49**
- strongly connected graph, *see* graph ↪ strongly connected
- strongly cyclic, *see* graph ↪ strongly cyclic
- subcubic graph, *see* graph ↪ subcubic
- subgraph, *see* graph ↪ subgraph
- subpath, *see* path ↪ subpath
- subquartic graph, *see* graph ↪ subquartic
- subquintic graph, *see* graph ↪ subquintic
- subregular, *see* graph ↪ subregular
- subset, **35**
- proper, **35**
- SUBSET FEEDBACK ARC SET, **16, 18**
- SUBSET FEEDBACK VERTEX SET, **16**
- SUBSET WEIGHTED FEEDBACK ARC SET, **16**

- SUBSET WEIGHTED FEEDBACK VERTEX SET, [16](#)
- SUBSET-FAS, *see* SUBSET FEEDBACK ARC SET
- SUBSET-FVS, *see* SUBSET FEEDBACK VERTEX SET
- SUBSET-wFAS, *see* SUBSET WEIGHTED FEEDBACK ARC SET
- SUBSET-wFVS, *see* SUBSET WEIGHTED FEEDBACK VERTEX SET
- superset, [35](#)  
     proper, [35](#)
- tail, [38](#)
- Tail On Forward Path Property, [175](#)
- topological sorting, [42](#), [45](#), [75](#)
- topsort number, [42](#)
- tournament, *see* graph ↪ tournament  
     bipartite, *see* graph ↪ bipartite tournament
- tuple, [36](#)
- undirected graphs, [38](#)
- Unique Games Conjecture, [21](#), [22](#)
- VC, *see* VERTEX COVER
- vertex  
     adjacent, [38](#)  
     component, [156](#)  
     consecutive, [44](#), [136](#), [138](#), [143](#), [155](#)  
     degree, [5](#)  
     fusion, [142](#)  
     in-maximal, [134](#), [147](#)  
     indegree, [5](#)  
     isolated, [40](#), [161](#)  
     left-blocking, [78](#), [112](#), [134](#)  
     maximal, [134](#)  
     out-maximal, [134](#), [147](#)  
     outdegree, [5](#)  
     passage, [156](#)  
     pseudosink, [45](#), [82](#), [134](#)  
     pseudosource, [45](#), [82](#), [134](#)  
     reachable, [41](#)  
     removal, [134](#)  
     right-blocking, [78](#), [112](#), [134](#)  
     sink, [40](#)  
     source, [40](#), [43](#)  
     strongly connected, [41](#)
- VERTEX COVER, [16](#), [21](#)
- vertex fusion, *see* vertex ↪ fusion
- vertex removal, *see* vertex ↪ removal
- vertex set, [38](#), [39](#)
- vertex splitting, [15](#)
- vertical split, [82](#)
- walk, [41](#)
- wdFAS, *see* WEIGHTED FEEDBACK ARC SET (DECISION)
- wdLO, *see* WEIGHTED LINEAR ORDERING (DECISION)
- weakly acyclic, *see* graph ↪ weakly acyclic
- WEIGHTED FEEDBACK ARC SET, [8](#)
- WEIGHTED FEEDBACK ARC SET (DECISION), [8](#)
- WEIGHTED FEEDBACK VERTEX SET, [15](#)
- WEIGHTED LINEAR ORDERING, [7](#)
- WEIGHTED LINEAR ORDERING (DECISION), [8](#)
- wFAS, *see* WEIGHTED FEEDBACK ARC SET

wFVS, *see* WEIGHTED FEEDBACK  
VERTEX SET

wLO, *see* WEIGHTED LINEAR ORDERING