Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

STEPHAN SCHWARTZ, LEONARDO BALESTRIERI,
RALF BORNDÖRFER

On Finding Subpaths With High Demand

ZIB Report 18-27 (Juni 2018)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

On Finding Subpaths With High Demand

Stephan Schwartz, Leonardo Balestrieri,
Ralf Borndörfer

Abstract

We study the problem of finding subpaths with high demand in a given network that is traversed by several users. The demand of a subpath is the number of users who completely cover this subpath during their trip. Especially with large instances, an efficient algorithm for computing all subpaths' demands is necessary. We introduce a path-graph to prevent multiple generations of the same subpath and give a recursive approach to compute the demands of all subpaths. Our runtime analysis shows, that the presented approach compares very well against the theoretical minimum runtime.

1 Introduction

In this paper we consider the following *subpath load computation problem* (SLCP). Given a directed graph G and user demands in the form of weighted paths in G, compute the load of every (sub)path in G. The load of a subpath P is the sum of the weights of all user paths T which contain P as a subpath.

The problem has applications in toll billing where users of a given network are billed for certain subpaths, called segments, which they cover during their trip. This graph segmentation problem is described in detail in [3] where the problem is solved using a set-packing integer programming formulation. The information of all subpaths' loads serves as an input for the IP and is therefore crucial for the formulation.

The SLCP has connections to finding frequent subpaths. In [2], the problem of mining paths which are frequent subpaths of given trajectories is considered. There, as usual for mining freqent substructures of a graph, the term frequent is determined by a given threshold value. Consequently, the focus of the used algorithms is a bottom-up approach where frequent substructures are combined to larger substructures which are then pruned if they are not frequent themselves, see [1] for an overwiew. In contrast, we aim at computing the loads or frequencies of all possible subpaths, favoring a different approach.

While the SLCP can be solved in polynomial time, efficient computations become necessary with large networks and even larger numbers of user paths. We tackle the

problem in two steps. First, we construct a *subpath-graph* to better handle duplicate subpaths. In the second step, we employ a recursive approach on the subpath-graph to compute the loads of all subpaths. Our runtime analysis shows, that the presented approach compares very well against the theoretical minimum runtime.

2 The Subpath Load Computation Problem

Let $G = (V, E)$ be a directed graph with $|V| = n$ and let $\mathcal{P}$ denote the set of simple paths in G. Moreover, let $\mathcal{T} \subseteq \mathcal{P}$ be a set of *user trajectories* in G with $|\mathcal{T}| = t$ and a demand $d_T \in \mathbb{N}$ for every $T \in \mathcal{T}$. For a path $P \in \mathcal{P}$ we define the *load* of P as follows:

$$\ell(P) := \sum_{T \in \mathcal{T}:\, P \subseteq T} d_T.$$

In other words, the load of a path can be seen as the number of users covering the path during their trip. The subpath load computation problem (SLCP) is to compute the load of every possible path in G.

First, we can observe that $\ell(P) = 0$ if $P \not\subseteq T$ for all $T \in \mathcal{T}$. Consequently, we define

$$\mathcal{P}_{\mathcal{T}} := \{P \in \mathcal{P} \mid \exists\, T \in \mathcal{T} : P \subseteq T\}$$

and state that $|\mathcal{P}_{\mathcal{T}}| \leq t \binom{n}{2}$ since each user trajectory $T \in \mathcal{T}$ has at most $\binom{n}{2}$ subpaths. As a result, we only have to compute the loads for paths $P \in \mathcal{P}_{\mathcal{T}}$ and therefore avoid the exponential size of $|\mathcal{P}|$.

Now let us take a closer look at the size of $\mathcal{P}_{\mathcal{T}}$ and define $s := |\mathcal{P}_{\mathcal{T}}|$. While there are instances with $s \in \Theta(tn^2)$, e.g. with arc-disjoint user trajectories, in many cases we have $s \ll tn^2$ due to intersecting user trajectories. For example, consider a path graph on n nodes with every possible user trajectory $\big($i.e. $t \in \Theta(n^2)\big)$. Therefore, we have $t \binom{n}{2} \in \Theta(n^4)$ while on the other hand, we have $s \in \Theta(n^2)$.

A natural first approach is to consider every user trajectory $T \in \mathcal{T}$ and every possible subpath of T to collect the demand for all subpaths. As pointed out above, this algorithm runs in $\mathcal{O}(tn^2)$ since we consider every subpath of every user trajectory.

In particular, every subpath in the intersection of two trajectories is considered multiple times. For example, consider the instance given in Figure 1. Since the subpath $(2, 3)$ is part of every trajectory it is explored $|\mathcal{T}|$ times with the above algorithm. In particular, if trajectories share a longer subpath, e.g. $(1, 2, 3)$, all subpaths of this subpath are considered for each of those trajectories.

In order to avoid these multiple considerations we introduce a *subpath-graph* that ensures that every subpath is expanded only once.

3 Constructing the Subpath-Graph

In the following we describe a problem-specific construction of what we call the *subpath-graph*. For a given instance $(G, \mathcal{T}, d)$ of the SLCP, the corresponding subpath-graph $D = (W,\ A)$ is a directed graph, where each node $w \in W$ represents a path $w =$

$\boxed{(1,2,3,4,5)}$	$\boxed{(2,3,4,5)}$	$\boxed{(5,1,2,3)}$	$\boxed{(4,1,2,3)}$	$\boxed{(1,2,3)}$
$(1,2,3,4)$	$(2,3,4)$	$(5,1,2)$	$(4,1,2)$	$(1,2)$
$(2,3,4,5)$	$(3,4,5)$	$(1,2,3)$	$(1,2,3)$	$(2,3)$
$(1,2,3)$	$(2,3)$	$(5,1)$	$(4,1)$	
$(2,3,4)$	$(3,4)$	$(1,2)$	$(1,2)$	
$(3,4,5)$	$(4,5)$	$(2,3)$	$(2,3)$	
$(1,2)$				
$(2,3)$				
$(3,4)$				
$(4,5)$				

Figure 1: User trajectories and considered subpaths for an instance of SLCP with $G = K_5$, $d \equiv 1$ and $\mathcal{T} = \{(1,2,3,4,5), (2,3,4,5), (5,1,2,3), (4,1,2,3), (1,2,3)\}$.

$(v_1, \ldots, v_k)$ in G. More specifically, we have $W = \mathcal{P}_\mathcal{T}$, i.e. the nodes in D correspond to the subpaths of $\mathcal{T}$. For every node $w = (v_1, \ldots, v_k) \in W$ with $k \geq 3$ we introduce an arc (w, w_1) with $w_1 = (v_1, \ldots, v_{k-1})$ and another arc (w, w_2) with $w_2 = (v_2, \ldots, v_k)$. Figure 2 presents an exemplary subpath-graph on a small network.

First, we can observe that the subpath-graph is a directed acyclic graph, since the head of every arc represents a proper subpath of its tail. Moreover, every node in D not representing an arc in G has exactly two successors, namely the two subpaths obtained by removing the first and the last node, respectively, which implies that $|A| \leq 2|W|$.

Algorithm 3.1 specifies the construction of the subpath-graph. The set W contains nodes for which all outgoing arcs have been created while Q contains the candidates to be added to W. For every candidate w we check if it has already been considered (line 5). If it was not added to W before, we generate the successors of w as well as the corresponding arcs and add the successors as candidates to Q. The algorithm terminates if the set of candidates is empty.

With the observations above we can evaluate the runtime of this algorithm. First, note that in line 11 we only add node w to the candidate set Q if w is a head of an arc in the subpath-graph D. As we have $|A| \leq 2s$, the loop in line 3 is executed at most $2s + t$ times which lies in $\mathcal{O}(s)$. If the check in line 5 is implemented using a prefix tree with already added nodes (cf. [2]), the lookup can be done in $\mathcal{O}(n\ \Delta(G))$ where $\Delta(G)$ is the maximum degree of G. The total runtime of Algorithm 3.1 is then in $\mathcal{O}(s\, n\, \Delta(G))$.

4 Solving the SLCP Recursively

Now that we have constructed the subpath-graph, we will describe an algorithm to efficiently compute the loads of all nodes in D to solve the SLCP.

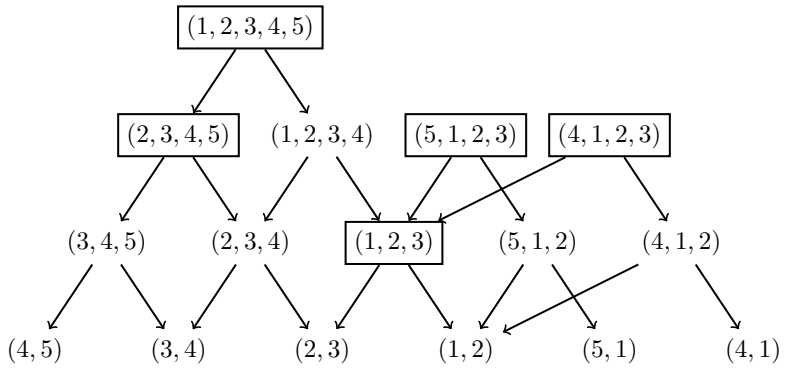

Figure 2: Examplary subpath-graph for $\mathcal{T} = \{(1,2,3,4,5), (2,3,4,5), (5,1,2,3), (4,1,2,3), (1,2,3)\}$ and $G = K_5$ as well as $d \equiv 1$.

Algorithm 3.1 construct subpath-graph

Input: user trajectories $\mathcal{T}$ of paths in G
Output: subpath-graph $D = (W, A)$

```
1: W, A := ∅
2: Q := T
3: while Q ≠ ∅ do
4:     w := Q.pop()        // w = (v_1, ..., v_k)
5:     if w ∉ W then
6:         W := W ∪ {w}
7:         if k ≥ 3 then
8:             w_1 := (v_1, ..., v_{k-1})
9:             w_2 := (v_2, ..., v_k)
10:            A := A ∪ {(w, w_1), (w, w_2)}
11:            Q := Q ∪ {w_1, w_2}
12: return (W, A)
```

Algorithm 4.1 Compute loads for all subpaths

Input: subpath-graph D, user trajectories $\mathcal{T}$ with demands (d_T)
Output: loads $L = \big(\ell(w)\big)_{w \in W}$

1: Compute $W_m := \{w = (v_1, \dots, v_m) \in W\}$ for $m = 2, \dots, n$
2: $\ell(w) := 0 \quad \forall w \in W$
3: $\ell(w) := d_T \quad \forall w = T \in \mathcal{T}$
4: **for** $m \in \{n, \dots, 2\}$ **do**
5: **for** $(v_1, \dots, v_m) \in W_m$ **do**
6: **if** $m \geq 3$ **then**
7: $\ell(v_1, \dots, v_{m-1}) := \ell(v_1, \dots, v_{m-1}) + \ell(v_1, \dots, v_m)$
8: $\ell(v_2, \dots, v_m) \quad := \ell(v_2, \dots, v_m) \quad + \ell(v_1, \dots, v_m)$
9: **if** $m \geq 4$ **then**
10: $\ell(v_2, \dots, v_{m-1}) := \ell(v_2, \dots, v_{m-1}) - \ell(v_1, \dots, v_m)$
11: **return** $L = \big(\ell(w)\big)_{w \in W}$

Algorithm 4.1 starts by partitioning the node set W into several level sets depending on the length of the path associated with each node (line 1). Afterwards, starting at the top level we descend the graph and for every node $w = (v_1, \dots, v_m)$ that we consider, we add the load of the current node to the load of both of its successors, given that w is not a leaf, i.e. $m \geq 3$. We also subtract the current load from the load of "the inner path" $(v_2, \dots, v_{m-1})$, if this is still a path, i.e. $m \geq 4$. We will see in a moment that this is necessary to respect the inclusion-exclusion priciple (cf. Theorem 1) and that the algorithm indeed computes the loads of all subpaths.

Let us first analyze the runtime of Algorithm 4.1. Computing the level sets W_m can be done in $\mathcal{O}(s)$ if we start at the leafs ($m = 2$) and traverse the graph with reversed arcs. In the main part we consider every node in W exactly once and since all other operations can be performed in $\mathcal{O}(1)$ the total runtime of this algorithm is in $\mathcal{O}(s)$.

In the following we prove the recursion which is implemented in Algorithm 4.1. We start by introducing further notation to simplify the illustration of the recursion. For $P = (v_1, \dots v_k) \in \mathcal{P}$ we define $N^-(P) := \{v_0 \in V \,|\, (v_0, v_1, \dots, v_k) \in \mathcal{P}\}$ and for $v \in N^-(P)$ we set $v \boxplus P := (v, v_1, \dots, v_k)$. This means that $N^-(P)$ are the predecessors of node v_1 which are not part of path P. Therefore, $(v, v_1, \dots, v_k)$ with $v \in N^-(P)$ is a simple path in G which we denote by $v \boxplus P$. Analogously, we define $N^+(P) := \{v_{k+1} \in V \,|\, (v_1, \dots, v_k, v_{k+1}) \in \mathcal{P}\}$ and set $P \boxplus v := (v_1, \dots, v_k, v)$ for $v \in N^+(P)$. Finally, for $P \in \mathcal{P}$ we write $\mathcal{T}(P) := \{T \in \mathcal{T} : P \subseteq T\}$ and obtain the following result.

Lemma 1. *For $P \in \mathcal{P}$ we have*

$$\mathcal{T}(P) = (\{P\} \cap \mathcal{T}) \cup \bigcup_{u \in N^-(P)} \mathcal{T}(u \boxplus P) \cup \bigcup_{v \in N^+(P)} \mathcal{T}(P \boxplus v). \tag{1}$$

Moreover, for arbitrary $u \in N^-(P)$ and $v \in N^+(P)$ we have

$$\mathcal{T}(u \boxplus P) \cap \mathcal{T}(P \boxplus v) \;=\; \mathcal{T}(u \boxplus P \boxplus v).$$

Proof. Let $P \in \mathcal{P}$ and $T \in \mathcal{T}$. We know that $P = T \iff (\{P\} \cap \mathcal{T}) = T$ and we can also observe that P is a proper subpath of T if and only if there is a node $u \in N^-(P)$ or $v \in N^+(P)$ such that $u \boxplus P \subseteq T$ or $P \boxplus v \subseteq T$. This proves the first equation. To prove the second equation let $P = (v_1, \ldots, v_k)$ and let $u \in N^-(P)$ and $v \in N^+(P)$. Now obviously $(u, v_1, \ldots, v_k) \subseteq T$ and $(v_1, \ldots, v_k, v) \subseteq T$ iff $(u, v_1, \ldots, v_k, v) \subseteq T$ which concludes the proof.

□ □

Now we extend the user demand to all paths by defining $d_P := 0 \; \forall P \notin \mathcal{T}$ to formulate the following recursion.

Theorem 1. *Let $P \in \mathcal{P}$, then*

$$\ell(P) \;=\; d_P + \sum_{u \in N^-(P)} \ell(u \boxplus P) + \sum_{v \in N^+(P)} \ell(P \boxplus v) - \sum_{u \in N^-(P)} \sum_{v \in N^+(P)} \ell(u \boxplus P \boxplus v).$$

Proof. We use Lemma 1 and the inclusion-exclusion principle. First note that for $u_1 \neq u_2 \in N^-(P)$ we have $\mathcal{T}(u_1 \boxplus P) \cap \mathcal{T}(u_2 \boxplus P) = \varnothing$. Analogously, for $v_1 \neq v_2 \in N^+(P)$ we have $\mathcal{T}(P \boxplus v_1) \cap \mathcal{T}(P \boxplus v_2) = \varnothing$. Inserting the identity from (1) into the definition of $\ell(P)$, the statement immediately follows using the inclusion-exclusion principle.

□ □

Theorem 2. *Algorithm 4.1 is correct.*

Proof. With Theorem 1 it is easy to prove the correctness of Algorithm 4.1. For any path $P \in \mathcal{P}_\mathcal{T}$ and for arbitrary $u \in N^-(P)$ we know that either $u \boxplus P \notin \mathcal{P}_\mathcal{T}$ or $u \boxplus P$ is a predecessor of P in the path-graph D. While the first implies that $\ell(u \boxplus P) = 0$, the latter ensures that the load is added to $\ell(P)$ in line 8 of the algorithm when the node $u \boxplus P$ and its successors are considered. Analogously, this holds for $v \in N^+(P)$ and the paths $P \boxplus v$ and $u \boxplus P \boxplus v$, proving the correctness of Algorithm 4.1.

□ □

We conclude that the subpath-graph can be constructed in $\mathcal{O}(s\, n\, \Delta(G))$. In many networks we can assume that the maximum degree is bounded, leading to a runtime of $\mathcal{O}(s\, n)$. If the subpath-graph is constructed, our recursive algorithm to solve the SLCP runs in $\mathcal{O}(s)$. Given that considering every subpath at least once leads to a minimum runtime of $\mathcal{O}(s)$, the presented algorithms are very well suited for solving the SLCP.

References

[1] Cook, D., Holder, L. (eds.): Mining Graph Data. John Wiley & Sons (2006)

[2] Guha, S.: Finding Frequent Subpaths in a Graph. International Journal of Data Mining & Knowledge Management Process **5** (2014), 35

[3] Schwartz, S., Borndörfer, R., Bartz, G.: The Graph Segmentation Problem. In Proceedings of INOC 2017, ENDM (to appear)