Library updates, program errors, and maintenance tasks in general force developers to apply the same code change to different locations within their projects. If the locations are very different to each other, it is very time-consuming to identify all of them. Even with sufficient time, there is no guarantee that a manual search reveals all locations. If the change is critical, each missed location can lead to severe consequences. The manual application of the code change to each location can also get tedious. If the change is larger, developers have to execute several transformation steps for each code location. In the worst case, they forget a required step and thus add new errors to their projects.

To support developers in this task, this thesis presents the recommendation system ARES. It leads to more accurate recommendations compared to previous approaches. ARES achieves this by conserving variations in the training examples in more detail due to its pattern design and by an improved handling of code movements. With the tool C3, this thesis also presents an extension to ARES that allows the extraction of training examples from code repositories. In combination, both tools create a recommendation system that automatically learns code recommendation patterns from repositories.

ARES, C3, and similar tools rely on lists of edit operations to express code changes. However, creating compact (i.e., short) lists of edit operations from data in repositories is difficult. As previous approaches produce too long lists for ARES and C3, this thesis presents a novel tree differencing approach called MTDIFF. The evaluation shows that MTDIFF shortens the edit operation lists compared to other state-of-the-art approaches.
Georg Dotzler

Learning Code Transformations from Repositories
FAU Studien aus der Informatik

Band 5

Herausgeber der Reihe:
Björn Eskofier, Richard Lenz, Andreas Maier,
Michael Philippsen, Lutz Schröder,
Wolfgang Schröder-Preikschat, Marc Stamminger, Rolf Wanka
Georg Dotzler

Learning Code Transformations from Repositories

Erlangen
FAU University Press
2018
Learning Code
Transformations from Repositories

Erlernen von Code-Transformationen mit Hilfe von Projektarchiven

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.

vorgelegt von
Georg Dotzler
aus Nabburg
Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg.

Tag der mündlichen Prüfung: 19.04.2018
Vorsitzender des Promotionsorgans: Prof. Dr.-Ing. Reinhard Lerch

Gutachter: Prof. Dr. Michael Philippsen
Prof. Dr. Sven Apel (Universität Passau)
Coś się kończy, coś się zaczyna.
(English: Something ends, something begins.)
- Andrzej Sapkowski
Abstract

Library updates, program errors, and maintenance tasks in general force developers to apply the same code change to different locations within their projects. If the locations are very different to each other, it is very time-consuming to identify all of them. Even with sufficient time, there is no guarantee that a manual search reveals all locations. If the change is critical, each missed location can lead to severe consequences. The manual application of the code change to each location can also get tedious. If the change is larger, developers have to execute several transformation steps for each code location. In the worst case, they forget a required step and thus add new errors to their projects.

There are several example-based code recommendation systems that support developers in the execution of such repetitive code changes. These tools use one or more code changes as input to generate a pattern. With such a pattern, they search for code locations that are similar to the input examples. If the tools find a suitable location, they apply the pattern and present the transformed code as recommendation to the developer. The use of such tools automates the search and requires less time. The tools also perform the required code transformation automatically and thus cause less errors compared to the manual execution. However, current state-of-the-art tools have two drawbacks. First, the tools often generate recommendations that are syntactically or semantically wrong. Developers cannot copy such recommendations directly into their projects. The necessary adaptation of these recommendations means additional manual work. Second, developers have to provide the input examples. This creates an additional workload for developers and makes the use of such tools often too costly.

To address the first drawback, this thesis presents the recommendation system ARES. It uses a pattern design that leads to more accurate recommendations compared to previous approaches. ARES achieves this by conserving variations in the input examples in more detail due to its pattern design and by an improved handling of code movements. The evaluation of ARES uses historic data from code repositories. This makes it possible to compare the generated recommendations with real code changes. In this scenario, ARES achieves an average accuracy of 96%.

With the tool C3, this thesis presents also a solution to the second drawback. ARES requires groups of two or more similar code changes
as input. $C3$ extracts such groups from code repositories. To identify the
groups, $C3$ supports two different syntactic similarity metrics and two
different clustering algorithms that are all specialized for code changes.

ARES, $C3$, and similar tools rely on lists of edit operations to express
code changes. However, creating compact (i.e., short) lists of edit opera-
tions from data in repositories is difficult. As previous approaches pro-
duce too long lists for ARES and $C3$, this thesis presents a novel tree
differencing approach called MTDIFF. It includes six general optimiza-
tions that are also compatible with other tree differencing approaches.
The evaluation shows that the optimizations shorten the edit operation
lists of other state-of-the-art approaches. Compared to these optimized
approaches, MTDIFF further shortens these lists.
Kurzfassung


Diverse beispielbasierte Empfehlungssysteme unterstützen Entwickler bei der Durchführung solcher sich wiederholenden Änderungen. Diese Programmierwerkzeuge benutzen als Eingabe eine oder mehrere Quelltextänderungen und erzeugen daraus Muster. Mittels solcher Muster suchen die Programmierwerkzeuge nach Quelltextstellen, die Ähnlichkeiten mit den Eingabebeispielen aufweisen. Wenn die Programmierwerkzeuge eine passende Stelle finden, wenden sie das Muster an und präsentieren den Entwicklern den geänderten Quelltext als Vorschlag. Die Verwendung eines Empfehlungssystems automatisiert somit die Suche und benötigt weniger Zeit als eine manuelle Durchführung. Die Programmierwerkzeuge führen auch die Quelltextänderung automatisch durch und verursachen damit weniger Fehler als bei einer manuellen Einarbeitung. Die aktuellen Programmierwerkzeuge haben jedoch zwei Probleme. Erstens generieren sie häufig Vorschläge, die syntaktisch oder semantisch falsch sind, was bedeutet, dass Entwickler diese nicht direkt in ihr Projekt einbauen können. Die deshalb nötige Anpassung der Vorschläge bedeutet zusätzliche Arbeit. Zweitens müssen Entwickler die Eingabebeispiele selbst bereit stellen. Dies ist zusätzlicher Mehraufwand für Entwickler, was den breiten Einsatz dieser Programmierwerkzeuge zu aufwändig macht.

Als Lösung für das erste Problem präsentiert die vorliegende Arbeit das Empfehlungssystem ARES. Es verwendet ein neues Musterdesign, das im Vergleich zu früheren Ansätzen zu genauerer Empfehlungen führt. Sein Musterdesign erlaubt es ARES, Abweichungen in
den Eingabebeispielen sowie Quelltextverschiebungen genauer auszu-
drücken. Die ARES-Evaluierung verwendet historische Daten aus
Quelltextprojektarchiven. Dies ermöglicht es, die generierten Empfeh-
lungen mit realen Quelltextänderungen zu vergleichen. Dabei erreicht
ARES eine durchschnittliche Genauigkeit von 96%.

Mit dem Programmierwerkzeug C3 präsentiert diese Arbeit auch eine
Lösung für das zweite Problem. ARES benötigt als Eingabe Gruppen
aus zwei oder mehr einander ähnlichen Quelltextänderungen. C3 extra-
hiert solche Gruppen aus Quelltextprojektarchiven. Um die Gruppen zu
identifizieren, verwendet C3 zwei unterschiedliche syntaktische Ähn-
lichkeitsmaße und zwei unterschiedliche Clustering-Verfahren, die spe-
ziell auf Quelltextänderungen angepasst wurden.

ARES, C3 und ähnliche Programmierwerkzeuge basieren darauf,
dass Quelltextänderungen als Listen von Änderungsoperationen darge-
stellt werden. Die Erstellung von kompakten (genauer: kurzen) Listen
von Änderungsoperationen aus Projektarchivdaten benötigt komplexe
Lösungen. Da die bisher veröffentlichten Ansätze für ARES und C3
unzureichend waren, präsentiert diese Arbeit außerdem MTDIFF, einen
neuen Ansatz zur Ermittlung von Unterschieden in Bäumen. MTDIFF
beinhaltet sechs allgemeine Optimierungsschritte, die auch für andere
baumbasierte Verfahren zur Ermittlung von Unterschieden geeignet
sind. Die Evaluierung in dieser Arbeit zeigt, dass diese Optimierungen
die Änderungsoperationslisten anderer Ansätze verkürzen. Im Ver-
gleich zu anderen optimierten Verfahren erzeugt MTDIFF nochmals
kürzere Listen.
Acknowledgement

This work was only possible due to the support of many people, both in my professional and in my personal life. I thank each and every one of them for their support in the years gone by. My whole journey started with my supervisor Michael Philippsen. Dear Michael, thank you for the opportunity to work on all the exciting projects in the last eight years, for our early morning discussions and for all your help during my time at the university. I would also like to thank Sven Apel for volunteering to review my thesis and for his detailed and valuable feedback. Furthermore, I would like to thank Klaus Meyer-Wegener, Wolfgang Schröder-Preikschat, and Georg Fischer for their time and their challenging questions at my final examination. Additionally, I thank the Embedded Systems Institute for the financial support.

For a long time, the office rooms of the programming systems group were my home and I thank the complete team for the great time I had with them, both at the office and outside of it. From the team, I would like to thank Ronald Veldema in particular. He was always there to discuss ideas, to create new ones out of thin air and to identify weak points and opportunities in my research. I also thank Stefan Kempf and Marius Kamp for our relaxed and often cheerful time in our office. Marius not only shared the office with me, but he, Patrick Kreutzer, and Christoph Romstöck all spent countless hours on writing code for my project. You three have my deepest gratitude for all the help you gave me in the past years.

My parents and my brother Paul also supported me during the whole time. Thank you for helping me through the long working hours and taking care of me when I arrived hungry at your door step. To my friends, especially Shari, Lukas and Anne Sophie, I have to apologize for all the missed parties, birthdays and gaming nights. I thank you for your endless patience. I am also grateful that you always found time for me when I needed someone to talk to. Most of all, I thank Sarah for her support, for bringing me tea and coffee, for her encouragement and her love. Sarah, thank you for everything. You made this thesis possible.

Georg Dotzler
List of Publications

Parts of this work have been published in the following research papers:


# Contents

Abstract i

Kurzfassung iii

Acknowledgement v

List of Publications vii

List of Figures xv

List of Tables xvii

1 Introduction 1
  1.1 Systematic Edits 2
  1.2 Recommendation Systems for Systematic Edits 6
  1.3 Contributions of this Thesis 10
  1.4 Outline 12

2 State of the Art and Related Work 15
  2.1 Static Analysis Tools in Industry 17
  2.2 Code Search Engines 19
  2.3 Code Fragment Recommendation 22
  2.4 Code Fragment Mining 26
  2.5 Code Transformation 30
  2.6 Code Completion 33
  2.7 Refactoring 34
  2.8 API Centered Works 36
  2.9 Program Repair 40
  2.10 Summary 41

3 Tree Differencing 45
  3.1 Tree Differencing on ASTs 48
  3.2 Related Work 53
  3.3 General Optimizations 64
    3.3.1 Identical Subtree Optimization $\Theta_A$ 65
5 Clustering of Code Changes

5.1 Related Work

5.2 Similarity of Code Changes
5.2.1 Identification of Changed Methods
5.2.2 Line-based Representation
5.2.3 Tree-based Representation
5.2.4 Pairwise Similarity Values

5.3 Clustering
5.3.1 Preprocessing
5.3.2 Agglomerative Hierarchical Clustering
5.3.3 DBSCAN

5.4 Complexity Analysis

5.5 Evaluation
5.5.1 Execution Time
5.5.2 Relevance
5.5.3 Relationship to Code Clones
5.5.4 Case Study ARES
5.5.5 Recommendation Accuracy
5.5.6 Limitations and Threats to Validity

5.6 Summary

6 Conclusions and Future Work

A Tree Differencing Appendix

Bibliography
# List of Figures

1.1 Two code changes that form a systematic edit. 3
1.2 Recommendation comparison. 8
1.3 Combined workflow of ARES and C3. 12

3.1 Code change computed with `diff`. 46
3.2 Tree differencing introductory example. 48
3.3 Edit script for the tree differencing introductory example in Fig. 3.2. 52
3.4 RTED example for $\Theta_A$. 66
3.5 Identical Subtree Optimization, $\Theta_A$. 67
3.6 GT example for $\Theta_B$. 68
3.7 LCS Optimization, $\Theta_B$. 69
3.8 GT example for $\Theta_C$. 71
3.9 Unmapped Leaves Optimization, $\Theta_C$. 72
3.10 Simplified GT example for $\Theta_D$. 73
3.11 Inner Node Optimization, $\Theta_D$. 74
3.12 GT example for $\Theta_E$. 75
3.13 Leaf Move Optimization, $\Theta_E$. 77
3.14 GT example for $\Theta_F$. 78
3.15 Cross Move Optimization, $\Theta_F$. 79
3.16 MTDIFF example. 81
3.17 Tree mapping in MTDIFF. 82
3.18 Simple example to show the MTDIFF heuristics. 83
3.19 Computation of leaf pairs in MTDIFF. 85
3.20 Ambiguous leaf similarity matrix for Fig 3.18. 86
3.21 Algorithm to compute the similarity matrix S. 87
3.22 Handling of ambiguous leaves in MTDIFF. 89
3.23 Example for the assignment problem. 90
3.24 Fast computation of inner node pairs in MTDIFF for the ambiguity treatment. 90
3.25 Computation of inner node pairs in MTDIFF. 91
3.26 Comparison of the GT-AST and the JDime-AST. 98
3.27 Tough code changes for MTDIFF. 103
3.28 Edit script sizes for the 152,979 file changes grouped by algorithm. 104
3.29 Positive size differences between the baseline and the optimized versions. ................................................. 105
3.30 Positive size differences between the optimized algorithms and MTDIFF. ............................................. 107
3.31 Fractions of move-only scripts that are found on the respective ASTs. ..................................................... 109
3.32 Comparison between JDime$A_F$ and MTDIFF. .......................................................... 110
3.33 Runtimes for the 152,979 file changes. ................................................................. 111

4.1 Workflow of ARES for one input group of code changes. .... 125
4.2 Generalized pattern. ............................................................... 134
4.3 ARES grammar. ................................................................. 135
4.4 Changes $c_\alpha$ and $c_\beta$ for the ARES example. .................. 142
4.5 Differences between the code changes $c_\alpha$ and $c_\beta$. .................. 143
4.6 Assignment part of the edit script adjustment Rule #48. ........ 148
4.7 Identical call destination part of Rule #31. .................. 149
4.8 Simplified version of Rule #42. .................. 151
4.9 Third code change $c_\gamma$. .................. 158
4.10 Differences between the original parts of $c_\gamma$ and the pattern from the first iteration .................. 159
4.11 Differences between the modified parts of $c_\gamma$ and the pattern from the first iteration .................. 160
4.12 Simplified version of Rule #4. .................. 161
4.13 Code base for the search example. .................. 161
4.14 Algorithm to search for suitable code locations. .................. 163
4.15 Matched AST nodes during the search for code locations. .................. 164
4.16 Code recommendations for the ARES example. .................. 168
4.17 LASE context example. .................. 177
4.18 ARES pattern for Id 20. .................. 179
4.19 ARES pattern for Id 23. .................. 180
4.20 Time comparison between LASE and ARES. .................. 181

5.1 Workflow of C3. .................. 190
5.2 C3 code change example. .................. 205
5.3 Tree-based representation of the C3 example. .................. 207
5.4 A similarity matrix and the graph it induces. .................. 214
5.5 Kernighan-Lin algorithm. .................. 215
5.6 Agglomerative hierarchical clustering. .................. 217
5.7 Agglomerative hierarchical clustering on a graph. .................. 218
5.8 DBSCAN algorithm for $\rho = 1$ .......................... 221
5.9 DBSCAN example .............................................. 222
5.10 Work times per commit ........................................ 228
5.11 Sequential clustering times per repository .................. 228
5.12 Distribution of useful groups of at least size 3 for the 9 repositories .................................................. 238
5.13 Accuracy values per recommendation .......................... 239
5.14 Accuracy distribution for the 9 repositories .................... 241

A.1 GT results with and without $\Theta_{A-F}$ ........................................ 253
A.2 Comparison between GT$_{A-C}$ and GT$_{A-D}$ for $\Theta_D$ .................. 254
A.3 RTED results with and without $\Theta_{A-F}$ ................................. 256
A.4 RTED/JSync results with and without $\Theta_D$ ............................... 257
A.5 CD results with and without $\Theta_{A-F}$ ......................................... 260
A.6 Comparison between CD$_{A-C}$ and CD$_{A-D}$ for $\Theta_D$ ............... 261
A.7 JSync results with and without $\Theta_{A-F}$ ...................................... 263
A.8 Comparison between JSync$_{A-E}$ and JSync$_{A-F}$ for $\Theta_F$ ............... 264
A.9 JDime results with and without $\Theta_{A-F}$ ................................. 266
A.10 Comparison between JDime$_{A-C}$ and JDime$_{A-D}$ for $\Theta_D$ ......... 267
A.11 Comparison between JDime$_{A-E}$ and JDime$_{A-F}$ for $\Theta_F$ ....... 268
## List of Tables

3.1 Comparison of the baseline algorithms with their optimized versions. .............................................. 99  
3.2 Effects of $\Theta_{A-F}$ on the GT results. ................................................................. 101  
3.3 Effects of $\Theta_{A-F}$ on the JSync results. ................................................................. 102  
3.4 Performance of MTDIFF on changed files. ................................................................. 105  
3.5 Questionnaire input. ................................................................. 115  
3.6 Questionnaire results (1,920 answers). ................................................................. 116  
4.1 Edit distances between code changes $c_A$ ... $c_D$. .................................................. 141  
4.2 Edit distances between gp and the code changes $c_G$, $c_D$. .................................................. 141  
4.3 Comparison with LASE on 23 code changes from Eclipse JDT and Eclipse SWT. .................................................. 173  
4.4 Results of ARES for 23 groups using all code changes as input. .................................................. 176  
4.5 Accuracy on JUnit. ................................................................. 184  
5.1 Repository properties (end of 2015). ................................................................. 224  
5.2 Similarity computation times. ................................................................. 226  
5.3 Identified bugfix groups from the Eclipse JDT and Eclipse SWT repositories. .................................................. 231  
5.4 Identified groups and clone groups in the 9 repositories. .................................................. 234  
5.5 Useful groups in the 9 repositories. ................................................................. 236
1 Introduction


Brandon Sanderson, The Way of Kings

If it arrives at the right time, advice is a valuable part of people’s lives [270]. It has the potential to improve people’s decisions [331] and thus helps them achieving their goals. Advice comes in many different forms. Family members or friends have always been a source for advice [231]. Ratings on websites like Amazon [320] are a modern form of advice that helps costumers selecting products. Besides its role in personal life, advice plays also an important part in the working environment [247]. For example, software developers use Stack Overflow [311], a question and answer website, to get advice for their current development tasks. As another source of advice, developers also use tools (e.g., FindBugs [25]) that detect problems in the source code. The tools highlight code locations that contain problems. Some tools also propose solutions. The advice of such tools and websites is one way that allows software developers to handle the demands of modern software development. As software projects grow continuously and change over time [279], the maintenance of long-term projects never ceases to be a hard and difficult task [37]. The deployment of the software to several platforms [172], interactions between developers in global teams [140] and new technologies (e.g., CUDA [257]) are additional challenges that further increase the difficulty of the software development process. It is no longer possible for a single developer to face all these challenges alone and developers increasingly seek advice from websites, co-workers, or tools.
To support developers in the maintenance of software projects, this thesis takes a closer look at tools that provide advice during the development process. In general, such tools belong to the Recommendation Systems category in the Software Engineering (RSSE) field. Robillard et al. [278, 277] define a RSSE as “a software application that provides information items estimated to be valuable for a software engineering task in a given context”. The available RSSEs cover the complete development process, from requirements engineering to software debugging and maintenance. Some of these tools also address the handling of systematic edits. This is the central use case of this thesis and the following section takes a closer look at it.

1.1 Systematic Edits

Meng et al. [236] define systematic edits as “similar but not identical changes to many locations”. This definition does not clarify the meaning of “similar” and leaves room for interpretation. In general, developers have to decide whether two code changes are similar or not in their current context. This thesis also leaves this decisive power in the hands of developers and refrains from a formal definition of similarity that would tie it to a concrete metric and thresholds.

Although the term “similar” remains open for interpretation, it is still possible to define the difference between similar and identical code changes. Two changes are similar but not identical if they share “some” but not all edit operations. Edit operations are small building blocks that modify code parts. This thesis uses four different types of edit operations, namely updates, inserts, deletes, and moves. Based on the definition by Meng et al., a single code change to a single location is not a systematic edit. Only if there are at least two similar code locations in which the change is useful (i.e., fixes a problem, improves the readability, or improves another quality feature), the change is systematic. This definition sets no restrictions to the relationship of the code locations. They can be in the same method or in different projects. Furthermore, two similar code changes can appear at the same time or with a time gap. Still, both form a systematic edit as long as they are “similar”.

Fig. 1.1 holds a simple example with two code changes that further illustrate the meaning of “similar”. In the first code change, a developer (Alice) removed an assert statement and replaced it with an if statement. The statement checks whether or not foo is null. In the second
1.1 Systematic Edits

Original code:  
01: updateValue();
02: \textbf{assert} (foo != null);
03: foo.someMethod(23);
04:
05: return;

Modified code:  
01: updateValue();
02: \textbf{if} (foo != null) {
03: \textbf{foo.someMethod}(23);
04: \textbf{System.out.print}(foo);
05: return;

(a) First code change from Alice’s project.

Original code:  
01: updateValue();
02: 
03: foo.someMethod(42);
04: foo.send();
05: return;

Modified code:  
01: updateValue();
02: \textbf{if} (foo != null) {
03: \textbf{foo.someMethod}(42);
04: \textbf{foo.send}();
05: return;

(b) Second code change from Bob’s project.

Figure 1.1.: Two code changes (left \textarrow right) that form a systematic edit. The boxes show the differences between the original and the modified code: \textcolor{green}{green} for insertions, \textcolor{red}{red} for deletions, \textcolor{blue}{blue} for movements.

change, a developer (Bob) also added the check for \texttt{null} that prevents a \texttt{NullPointerException}. This part is similar to the first change. In contrast to the first example, two statements were moved to the body of the \texttt{if} statement. There is also no \texttt{assert} to remove. The insertion of the \texttt{print} statement is also only present in the first code change.

Both examples share the edit operation that inserts the \texttt{if} statement and the operation to move one statement into the body of the \texttt{if} statement. As there are three differences (the deleted \texttt{assert}, the added \texttt{print}, and the moved \texttt{send}), the changes are similar but not identical. Thus, both changes form a systematic edit, although one was made in Alice’s project and one in Bob’s. The time when both developers executed their changes is irrelevant.

As edit operations vary between code locations, standard development tools provide no support for developers in performing systematic edits. For example, basic search and replace approaches that Integrated Development Environments (IDEs) provide (e.g., Eclipse [99], IntelliJ IDEA [163]) do not work for systematic edits. It is impossible to express the differences that occur at each code location like the \texttt{assert} in the example with a simple search query. Regular expressions to search and replace the code are also insufficient as they ignore the tree-based structure of source code. This can lead to incorrect code if, for example, the
updateValue and the someMethod call appear in different code blocks. Refactoring wizards in the IDEs are also insufficient as they preserve the semantics of the code. For the examples in Fig. 1.1 and for bug fixes in general this is too restrictive. The added if statements in the examples alter the semantics because the code no longer produces a NullPointerException if foo is null. Thus, these code changes are not refactorings. However, as refactorings are a subcategory of systematic edits, refactoring wizards partially help for systematic edits.

As shown above, systematic edits do not only require a more sophisticated approach, they also happen often and developers spend a lot of time on them. For example, Kim and Notkin [187] analyzed commits in two open source projects with their tool LSdiff. They found that 75% of structural changes (e.g., field changes or interface changes) in their dataset were systematic edits. However, systematic edits do not only occur in structural changes. Nguyen et al. [256] performed a manual analysis of ~1,000 commits in five open-source projects. Their results show that 17-45% of all bug fixes in these commits were recurring, i.e., appeared more than once. Again, all recurring bug fixes that are not identical are part of a systematic edit. Molderez et al. [243] identified over 5,000 systematic edits in 43,756 commits of 51 software projects with closed itemset mining [335]. In their study, the identified systematic edits cover complete methods. This also shows that systematic edits are not restricted to bug fixes or structural edits. In a study with 317 libraries, Xavier et al. [321] found that 15% of all library changes break backwards compatibility. According to their results, this affects 2.5% of the examined 260,000 clients that use the libraries. Adapting all the affected client applications to the new library versions leads to systematic edits. There are also some library changes that affect up to 100% of the client applications. In this case, even more applications require the execution of systematic edits. Overall, these studies show the regular occurrence of systematic edits and their diversity. As a consequence, developers spend a lot of time on their execution.

Due to the above definition of systematic edits, they are connected to code clones, i.e., redundant code parts in software systems [177]. Removing these clones can reduce the number of required systematic edits. For that purpose, Meng et al. [235] developed RASE that removes code clones by applying refactoring operations (e.g., by extracting a method). With this approach it was possible to reduce the number of required systematic edits. However, they also discovered that 42%
of the systematic edits remain and cannot be handled with automatic clone removal. Kapser and Godfrey also found in a case study [177] that the use of code clones can be a valid design decision. For example, code clones can reduce the complexity of code that has to take different hardware or platforms into account. Handling all the different hardware or platform variants at one single location avoids code clones but requires many if statements that take care of the code variants at a fine-grained level. By contrast, code clones separate the different variants earlier and thus do not require such a fine-grained regime.

Besides the frequency of systematic edits and the associated time overhead, there is also another aspect that makes systematic edits an important research topic. Several recent studies (e.g., Costas and Kärreman [69]) show that boring tasks arise in knowledge work tasks (e.g., software development) and are not restricted to industrial or bureaucratic environments. According to Cummings et al. [76], boring tasks are monotonous, repetitive and either require constant attention or have a low task load (i.e., a low demand required by the work environment). Systematic edits share these properties. They provide no surprises to developers and thus are monotonous. Each location (see Fig. 1.1) is similar to the previous one and poses no special challenge for developers. The actual edit operations, for example, adding the if statement or moving the code in Fig. 1.1, have to be repeated at each location. This makes systematic edits repetitive. Despite that, there are some differences in the code locations (e.g., the assert statement) and the actual change requires constant attention. Due to these shared properties, systematic edits can be considered a boring task. Thus, they can also cause boredom. Fisher [115] defines boredom as an “unpleasant, transient affective state in which the individual feels a pervasive lack of interest in and difficulty concentrating on the current activity”. As a consequence, boredom causes distress in employees and also leads to counterproductive work behavior [153]. A recent experiment by Hunter and Eastwood [159] showed that boredom is also connected to attention failures. According to Ko and Myers [191], such attention failures cause program errors. Thus, attention failures can happen during the execution of systematic edits attention. This can lead to the introduction of additional bugs in a project during the execution of the code changes. To avoid such errors and to reduce the time overhead, this thesis introduces a novel RSSE system that helps developers in executing systematic edits.
1 Introduction

1.2 Recommendation Systems for Systematic Edits

Systematic edits occur regularly during the development process and can cause bugs due to attention failures. This has led to the development of several RSSEs that provide recommendations for systematic edits. Some tools only target specific systematic changes (e.g., Android changes [253]) whereas others are more general and learn new recommendation patterns from one or more input examples (e.g., LASE [236], REFAZER [281]). However, the current example-based tools have several limitations. This section uses the recommendation results by LASE and REFAZER to explain the limitations of current RSSEs in detail.

To train LASE for a new systematic edit, developers have to provide two or more code change examples of it. As LASE targets method changes, each such example has to consist of two versions of a method. One version is the original method before a change and the other is the modified method after a change. Based on these training examples, LASE creates a generalized pattern. Afterwards, LASE is able to search a code base for locations that allow the application of the created pattern. If it finds such a location, LASE applies the pattern to a copy of the code location and presents the transformed copy as a recommendation to developers.

Each generalized pattern of LASE consist of the common list of edit operations and the common context of the examples. LASE distinguishes between insert, delete, update and move edit operations. The common list of edit operations in the pattern transforms the abstract syntax trees (ASTs) of the methods before a change into the ASTs of the methods after a change. The used AST structure is very common in compilers and software engineering tools. It also plays an important role throughout this thesis. A formal definition is available in Sec. 3.1.

To create a pattern for each code change, LASE first generates the list of edit operations that transforms the original method into the modified method. Then LASE combines the edit operations of the training examples by determining their longest common subsequence (LCS) [38]. The common context that LASE uses consists of the common subtrees between the original methods of the input examples. To compute this common context, LASE uses the Maximum Common Subtree Extraction (MCESE) algorithm [218]. As the computed subtrees can contain code that is unrelated to the actual changes, LASE
1.2 Recommendation Systems for Systematic Edits

further refines the context with the help of a data dependency analysis. It identifies all unchanged AST nodes (i.e., nodes not affected by edit operations) that have no data dependencies to changed AST nodes. LASE excludes all such independent, unchanged AST nodes from the context.

Current state-of-the-art tools like LASE and REFAZER have three drawbacks. They produce inaccurate recommendations, do not help developers in the necessary validation of the recommendations, and also expect the developers to provide the input examples.

The first drawback is the accuracy. If a recommendation differs from the code a developer requires (or would have written), it is less accurate than manual changes. This forces developers to adapt the generated recommendations to their needs. As this adaptation process is monotonous, repetitive and requires constant attention, it also shares the characteristics of boring tasks and does not prevent attention failures that are error-prone. For accurate recommendations it is necessary that the created patterns preserve as much information from the input examples as possible. At the same time, the tools have to generalize the patterns. This generalization makes it possible to apply patterns to additional code locations, not only to the input examples. Each LASE pattern only includes the common list of edit operations of the input examples. This removes most of the variations in the input examples and all information about edit operations that only appear in some of the input examples is lost. For the systematic edit in Fig 1.1 this reduces the accuracy of LASE’s recommendations. As the delete of the assert statement is only present in one input example, this delete operation is not part of the pattern. This means that each recommendation of LASE retains the assert statement if it is present at a code location. Fig. 1.2 illustrates this problem with an example. The example uses the pattern that LASE learns from the two code changes in Fig. 1.1 and applies it to the code location in Fig. 1.2(a). The fact that the assert statement is still present in the recommendation (see Fig. 1.2(b)) is not the only problem. In a correct recommendation, foo.transfer has to be in the body of the if statement and not below the if statement, see Fig. 1.2(c). The recommendation of LASE still leads to a NullPointerException due to the wrong position of foo.transfer. In theory, the second input example in Fig 1.1 gives LASE the information that the statement after foo.someMethod also has to be moved to the body of the if statement. However, as it is only present in a single example,
1 Introduction

```java
updateValue();
assert (foo != null);
foo.someMethod(42);
foo.transfer();
return;
```

(a) Code location.

```java
updateValue();
assert (foo != null);
if (foo != null) {
    foo.someMethod(42);
}
foo.transfer();
return;
```

(b) Inaccurate recommendation by LASE.

```java
updateValue();
assert (foo != null);
foo.someMethod(42);
foo.transfer();
System.out.print(foo);
return;
```

(c) Expected recommendation.

Figure 1.2.: Recommendation comparison. Differences highlighted in blue.

LASE actually excludes this edit operation from the pattern. Again, the differences of the input examples cause this inaccuracy.

The inserted `print` statement in line 5 is also not present in both input examples. Only the first example contains this inserted `print` statement. As it is not part of the common edit operations, the LASE pattern does not consider this additional statement (see Fig. 1.2(b)). The domain specific language (DSL) in REFAZER also does not provide the means to encode this change in the pattern.

The accuracy of a recommendation is not only limited by the information preserved in the pattern, but also by the algorithms that are used during the pattern creation process. For example, LASE and REFAZER both use tree differencing algorithms to compute the differences of the input examples. LASE uses ChangeDistiller (CD) [119], a tree differencing algorithm that supports code movements. CD uses heuristics to determine a short list of edit operations that describe the code change. Finding an optimal solution without heuristics is infeasible as the computation of the optimal edit script that includes code movements is a NP-hard problem [41, 45]. Heuristics can lead to non-optimal results in some cases and thus to unnecessary edit operations. For example, if the heuristics do not detect a code movement, CD expresses the movement as a `delete` and an `insert` operation. Thus, it requires two instead of one edit operation. Such differences caused by the heuristics in CD
1.2 Recommendation Systems for Systematic Edits

can reduce the number of common edit operations that LASE identifies. As a consequence, LASE performs only a subset of the edit operations that the systematic edit requires. For example, if the heuristics detect the move of \texttt{foo.someMethod} only in one example and use a delete and insert in the second one, LASE excludes this movement from the pattern and the accuracy of the recommendations drops.

REFAZER uses the tree differencing algorithm developed by Zhang and Shasha [338] that does not consider code movements. This can also lead to recommendations that do not reflect the expectations of developers and require additional changes by them. For example, in Fig. 1.1 it is important to identify the movement of \texttt{foo.send} to a new location. If a tool does not detect this move, it has to express this change as delete and insert operations. This is less general and would lead to wrong results if the code location contains \texttt{foo.transfer} instead of \texttt{foo.send}.

As mentioned above, the second drawback of current example-based systems is the difficulty of the validation process. Developers must validate the correctness of the presented recommendations to determine whether it is safe and useful to add them to their code base. However, this is an open challenge [186]. One potential solution for this problem is the presentation of the patterns in a human readable way. In LASE, it is possible to review the list of edit operations. Similarly, REFAZER encodes the pattern in a human-readable DSL that is based on the edit operations. Although a readable presentation is available, the usefulness of both presentations is limited. Both require developers to imagine the ASTs and the relationship between AST nodes and the edit operations. This is a lot of effort for larger systematic edits. On the level of AST nodes, the simple code change in Fig. 1.1(a) already requires 11 edit operations and the modified part contains 18 AST nodes. Larger methods contain several hundred AST nodes and some systematic edits require over 100 edit operations. Understanding such graphs causes a large cognitive load and can overwhelm human viewers [136]. Huang et al. [157] showed that even in small graphs with only 25 nodes and 98 links human perception can be overburdened. Additionally, for a developer without knowledge of ASTs and tree transformations, lists of edit operations or a DSL based on edit operations are hard to understand. Thus, a compact and familiar pattern representation would be an improvement to the state-of-the-art because it can help developers understanding the presented recommendations.
The third drawback of state-of-the-art tools is their expectation that developers provide the input examples. If different developers perform the code changes that form a systematic edit (e.g., Alice and Bob), it is unlikely that they are aware that their current code changes are suitable training examples for a RSSE tool. According to Kim and Meng [186], this is one of the challenges for further research on the topic of RSSEs for systematic edits. Without an approach that identifies training examples automatically, it is impossible to build a proactive system that works in the background without requiring the help of developers. A fully automatic RSSE should only require the developers to accept or decline a recommendation.

## 1.3 Contributions of this Thesis

The goal of this thesis is to address the three drawbacks described in the previous section. Therefore, it presents three contributions that form the building blocks of an automatic RSSE. This new RSSE provides advice for developers and help them with systematic edits.

The first contribution of this thesis targets tree differencing algorithms. RSSE tools (e.g., LASE, REFAZER) use such algorithms to determine differences in the input examples. The more accurate the results of the tree differencing algorithms are, the more accurate is the generalized pattern. To improve the accuracy, this thesis introduces six optimizations that any tree differencing algorithm can use. The optimizations minimize the number of edit operations required to express code changes. As a consequence, the edit operations express the code changes more accurately. This thesis also discusses the novel Move-optimized Tree Differencing (MTDIFF) algorithm that uses the six optimizations and further minimizes the number of edit operations for code changes. It even requires fewer code changes than the state-of-the-art tools in combination with the six optimizations. To minimize the number of edit operations, the optimizations and MTDIFF both use novel heuristics that detect code movements more accurately than previous approaches. This also increases the accuracy of the recommendations. Five of the six optimizations and most heuristics of MTDIFF were published in 2016 at the Automatic Software Engineering conference [89].

The second contribution is an Accurate REcommendation System (ARES) that uses a novel pattern design and applies new algorithms
that create more accurate recommendations from such patterns. In fact, ARES creates the desired recommendation in Fig. 1.2(c) for the two code changes of Fig. 1.2. To achieve this result, the pattern design of ARES uses an original and a modified code part. Annotations added to these code parts express variations in the input examples (e.g., the `assert` or the `print` statements in Fig. 1.1). The annotations are more flexible than the pattern designs of previous approaches. The pattern design is also able to express code movements (e.g., `foo.send` in Fig. 1.1) that appear only in some input examples. Additionally, the pattern design is more compact compared to lengthy list of edit operations. The initial pattern design was published in 2012 at the Workshop on Recommendation Systems for Software Engineering [90]. An improved version was published at the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering in 2017 [88]. A pattern design is not very helpful if it is impossible to create the corresponding patterns from examples without human interaction. Therefore, ARES is able to use two or more input examples to create a pattern in the novel and more accurate pattern design. For that purpose it uses MTDIFF. ARES also includes algorithms to search in a code base for code locations for which a pattern is applicable and an algorithm to create suitable and accurate code recommendations for such locations. Fig. 1.3 shows the workflow of ARES. It uses groups of two or more code changes as input. For each such group, ARES creates a generalized pattern. Then it is able to search a Code Base for locations that allow the application of the created patterns. If ARES finds such a location, it generates an accurate recommendation. ARES was initially published at the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering in 2017 [88].

LASE, REFAZER, and other tools still require that developers provide the input examples. A proactive system that does not need to rely on developers has to identify suitable input examples automatically. Thus, the third contribution of this thesis is the tool $C^3$ that looks for similar code changes in code repositories (see Fig. 1.3). It does not only identify groups of syntactically identical code changes, but also identifies groups of similar changes that contain variations. To identify these groups, $C^3$ uses two different encodings for the code changes. Then it is able to apply two different clustering algorithms on the encoded changes to identify groups of similar code changes. These resulting groups of
1 Introduction

Figure 1.3.: Combined workflow of ARES and $C3$.

similar changes are ideal inputs for ARES. For each variation between code changes in the same group, ARES adds annotations that generalize the pattern. These annotations make it possible that the resulting patterns are not only applicable to the input examples, but also to several other code locations in a code base. $C3$ was first published at the Mining Software Repositories Conference in 2016 [196].

1.4 Outline

Chapter 2 gives a general overview of static analysis tools and state-of-the-art recommendation systems. The following three chapters also contain state-of-the-art sections that discuss the works most related to the contributions presented in them. Chapter 3 begins with a short introduction to tree differencing algorithms and edit operations. Then it discusses the six general optimizations that are applicable to current state-of-the-art tools. Afterwards, it illustrates the heuristics in MTDIFF and evaluates the six optimizations as well as MTDIFF. The evaluation shows that when the six optimizations are added to several state-of-the-art tree differencing algorithms they lead to shorter lists of edit operations for thousands of code changes. It also shows that
MTDIFF requires even fewer operations than the combination of state-of-the-art algorithms with the six optimizations. Additionally, the evaluation presents the results of a questionnaire that examines the relationship between the perception of helpfulness and edit script length.

Chapter 4 explains how the pattern design of ARES can improve the accuracy of a recommendation system. It also gives a formal definition of the different annotations that ARES supports and explains their uses. Afterwards, the chapter discusses the details of the algorithms in ARES. It explains the approach that creates the original and modified parts of the patterns and adds the different annotations. It also shows how ARES can use created patterns to find code locations that allow the application of the pattern. It also discusses how ARES applies the patterns to these locations to create recommendations. To evaluate ARES, the chapter compares ARES with LASE by applying both tools to systematic edits from different repositories.

Chapter 5 introduces $C^3$ that creates groups of code changes suitable as input for ARES. The chapter contains a detailed description of the two different encodings that $C^3$ uses to compute the similarity of code changes. It also sketches the two clustering algorithms that $C^3$ supports. The evaluation in Chapter 5 discusses the general results of $C^3$. It also combines ARES with $C^3$, applies the combination to different Java projects, and measures the accuracy of the created recommendations.

Chapter 6 summarizes the contributions and results of this thesis. It also gives an outlook on further research options.
2 State of the Art and Related Work

Over the course of the last couple years, I spent a lot of time with static software analysis tools. It’s been an eye opening experience for me.

John Carmack, QuakeCon 2011

The quote by John Carmack (co-founder of id Software, CTO of Oculus VR) shows the potential that developers in the software industry see in static analysis tools. The term “static analysis tool” subsumes all tools that analyze code fragments (e.g., expressions, code lines, methods, classes, programs) without executing the code or program. ARES falls into this category as it searches for systematic edits in a code base and recommends program changes without executing the program. An overview of all available static analysis tools is beyond the scope of this thesis. Therefore, Sec. 2.1 gives an overview of static analysis tools that are used in the industry. It illustrates that static analysis tools used in industry projects use predefined patterns and thus are not flexible enough for systematic edits.

ARES performs three tasks. It creates patterns from examples, it searches for code locations, and it creates recommendations. To generate the recommendations, ARES performs the transformations encoded in the patterns on the identified code locations. The following sections are all related to these tasks. Sec. 2.2 discusses search tools that require active participation of developers to identify code locations and code fragments. Sec. 2.3 takes a closer look at tools that search for code fragments in the background without input from developers. To create patterns, ARES requires similar code changes as input examples. C3 mines them from code repositories. Other related mining approaches identify similar code parts or other code fragments in repositories.
Sec. 2.4 discusses them and other mining approaches that address source code fragments.

ARES is not the only tool that performs code transformations to help developers in their work. There are even languages that developers can use to describe transformations. These and other ways to create and describe transformations are the focus of Sec. 2.5.

ARES also recommends code fragments that developers can use to speed up the execution of systematic edits. The size of the recommended code fragments ranges from a single line to complete methods. ARES recommends both refactorings that preserve the semantics of the code as well as bug fixes that change them. Besides ARES, there are many other tools that recommend code fragments. Code completion systems recommend the next identifier or the next expression to developers. Compared to ARES, code completion systems work on a smaller scale. Sec. 2.6 gives an overview of code completion tools. There are also tools that recommend refactoring opportunities to developers. Sec. 2.7 discusses them.

Many code changes also occur due to application programming interface (API) changes. For example, if the interface of a library is changed between versions, it is also necessary to change the code that uses the library. Due to the regularity of API changes, there are several tools that support developers in their execution. Some of them are also recommendation systems. Sec. 2.8 discusses some API tools that are related to ARES.

Closely related to code recommendation systems are approaches to program repair. Like ARES and other recommendation tools, they use patterns to search for code locations and they also transform these locations to create program patches. However, instead of striving to provide recommendations, program repair tools implement a direct solution to a problem. Their goal is that the developer does not need to get involved. To be able to change code directly, program repair tools have to ensure that their program change is correct. For that purpose, they need a test suite or another specification that allows them to determine whether or not a repair is correct. Nevertheless, program repair is closely related to ARES and therefore discussed in Sec. 2.9.

Whereas this chapter gives an overview about related static analysis tools, each of the following three chapters also contains an extra related work section. These three sections present works that are much closer to what is presented in this thesis. Sec. 3.2 discusses tree differencing
approaches that are related to MTDIFF. Sec. 4.1 addresses example-based recommendation systems and other tools that are most closely related to ARES. As C3 mines code changes and clusters them into groups, Sec. 5.1 discusses similar mining approaches.

2.1 Static Analysis Tools in Industry

This section gives an overview of static analysis tools that Microsoft, Google, and Facebook use in their projects. The list of discussed tools comes from the collection by Christakis and Bird [61]. All the following tools have in common that they rely on predefined patterns and are not designed for systematic edits. Thus, although they detect many serious problems, they are not flexible enough for the automatic handling of systematic edits.

Microsoft uses several static analysis tools. For example, the company develops and uses BinSkim [42]. This tool validates linker and compiler settings. Developers can add new validation patterns written in C#. FxCop [126] detects problems in .NET assemblies (e.g., tests for empty strings with string length). It is possible to extend the functionality by means of patterns which are written in C#. The tool PREfast [267] reports problems in C/C++ programs (e.g., buffer overruns). Currently, it is only able to detect predefined problems and users cannot add new functionalities. StyleCop [299] checks the style and consistency of C# files. Like BinSkim and FxCop, it is possible to provide additional checks in C#. This short selection of tools already shows that they do not target systematic edits, but instead focus on common problems. To address a specific systematic edit with such a tool a custom pattern is required. It is the task of the developer to write such a pattern manually. ARES only requires examples for the systematic edit and creates patterns automatically. Thus, although the tools provide help for developers in numerous cases, they offer no help for a specific systematic edit.

Microsoft also uses third-party plugins. CodeRush [65] is a plugin for Visual Studio. It detects problems in C# and Visual Basic (e.g., unused code parts). Developers can extend it with plugins written in C# and Visual Basic. Fortify [122] and Checkmarx [58] are two tools that identify security vulnerabilities. Both tools provide a DSL that developers can use to define custom rules. The static analysis tool Synopsis (Coverity) [71, 39] provides a complete Software Development Kit (SDK) that gives developers the ability to write custom analysis
checks. Cppcheck [72] is another static analysis tool with features like bounds checking. Developers can write simple rules for Cppcheck with regular expressions. If more sophisticated analysis techniques are necessary, developers have to change the code of Cppcheck directly. The disadvantages of these tools are the same as described above.

Google introduced Shipshape/Tricorder [294, 286] as a static analysis platform. This platform supports the use of different analysis tools as plugins. It supports Error Prone [107] for the detection of Java problems, Checkstyle [59] to validate the Java coding style, and also several other tools (e.g., ClangTidy [62]). Error Prone and Checkstyle both allow developers to write new checks in Java. It is possible to extend the functionality of ClangTidy with checks written in Python. Google also uses FindBugs [114, 297, 25], another analysis tool that detects common problems (e.g., null pointer exceptions) in Java code. For systematic edits, developers can add custom checks written in Java.

Facebook relies on Infer [162, 52, 53] to perform static analysis checks during the development process. Similar to other tools, Infer includes many predefined checks. For example, it can check for null pointer exceptions. If additional checks are necessary (e.g., for systematic edits), developers have to provide them written in OCaml.

All the tools above rely on predefined patterns to search and fix problems in a code base. To include new patterns, developers have to write the new functionality in a DSL (e.g., Fortify [122]) or a general-purpose programming language like C#. This can be a lot of effort. For some tools there is the additional problem that the programming language for the patterns (or checks) is different from the programming language of the code base. Thus, developers are forced to write patterns (or checks) in an unfamiliar language if they want to create a pattern for a specific systematic edit. Developers also have to know the API of the static analysis tool, e.g., to read code fragments and to display results in their patterns. It is also useful to know which common tasks the frameworks already provide to avoid duplicate work. For example, many frameworks already create the control flow graph (CFG) and offer access to it through their API. Due to the framework designs, the writing of new patterns or checks often requires knowledge about analysis techniques (e.g., about control flow and data flow analysis). This is not basic knowledge and makes the learning curve for developers even steeper.

Thus, writing a plugin for a systematic edit is often too much effort, especially since the tools do not provide support for creating patterns.
As a consequence, developers often perform systematic edits manually, despite the huge range of available tools. In contrast to the discussed tools, ARES only requires example changes as input. For its use, a deep knowledge about ASTs, analysis techniques, or tool APIs is unnecessary.

2.2 Code Search Engines

Searching is entangled with systematic edits. Developers have to search for code locations in their code base to be able to apply a systematic edit. Due to the lack of suitable tools, this is regularly done with textual queries. ARES automates this task as it discovers locations for systematic edits without human interaction. Searching manually costs more time and there is no guarantee that such a search has identified all code locations. Search tools also are only helpful if the developers are aware that there is a code location that has to be changed based on a specific systematic edit pattern. If the changes that form the systematic edit were made in another project or were executed by others, developers often do not know that a systematic edit is relevant for their code base.

As a consequence, search engines do not provide help to fix the problem. ARES can use changes from other projects and from other developers to automatically search for code locations. As this search for code locations is an internal part of ARES, it is also related to other search approaches. This makes it necessary to compare current approaches in this area with ARES. In general, these tools do not cover systematic edits directly. However, they can provide the necessary information to developers that prevents bugs or redundant code. This means they can reduce the number of systematic edits that developers have to apply to a code base due to a better code quality.

Čubranić et al. [74, 75] presented Hipikat. During the development process, it collects code fragments, the involved developers and a variety of links (e.g., between developers and code modified by them). Developers can use this collected information to find code fragments that are related to their current tasks. For example, they can select a code fragment and request similar code fragments from Hipikat. Bajracharya et al. [28] presented a technique that automatically creates a search index from API usages. Developers can execute keyword queries on this index. The system then creates a ranked list of code snippets (API usages) and presents it as result. This can be a short cut to identify code locations for systematic edits compared to a simple text
State of the Art and Related Work

search. McMillan et al. [233, 234] developed an approach that addresses features (e.g., MP3 support). To find modules that implement required features, developers can write feature descriptions and use them as input to the search engine. Hashimoto et al. [138] introduced the tree differencing tool Diff/TS [137] to analyze the differences in repository commits and add the results to a fact database. Developers can query this database to search for code changes, for example, code changes that contain a specific identifier name. Thus, this database can also help identifying code locations for systematic edits. The tool by Ichii et al. [160] uses the browsing history of developers. It recommends components that developers examined who have a similar browsing history. Recently, Martie et al. [227] presented two novel search engines tailored to the identification of source code examples. The search engines accept keywords or text queries. They also provide a user interface (UI) in which developers can easily add filters to the query. PROSPECTOR by Mandelin et al. [222] mines a code base for related objects. For example, it considers two classes \((c_1, c_2)\) to be related if \(c_1\) has a method that creates an instance of \(c_2\). In Java, this is the case for the class \(Input\) that creates instances of \(Buffer\). If a developer builds a query on such a pair of related objects, PROSPECTOR recommends example code in a code base (if available). It does not recommend code locations for a necessary systematic edit.

Other approaches search for bug fixes. If developers patch a bug in their code base, they can search for similar bugs and the corresponding fix that solves the problem. Thus, these tools help to identify systematic edits and are related to C3. DebugAdvisor by Ashok et al. [22] allows developers to search for bug fixes that are similar to the bug detected in their code base based with a search query. Queries in DebugAdvisor are not limited to a short search string. They can contain additional information like core dumps or functions to improve the results. Ye et al. [332] rank source files according to their relevance to a specific bug report. They train their system with previous bug reports and the fixes in software repositories. Almhana et al. [7] propose a similar approach. Their work tries to minimize the number of recommended classes with the help of a multi-objective optimizer. This makes it easier for developers to evaluate the results.

Some tools make it possible to search for code structures (e.g., dependency graphs, call graphs, type hierarchies) and thus they can make the search for code locations of systematic edits easier. With the tool Dora
by Hill et al. [145], developers can search for code structures with queries consisting of code parts. Dora then displays the code structure that is related to the query as a call graph. The tool by Robbilard [276] recommends items related to the items in a search query based on their structural relationships in the code base. Wang et al. [316] introduced a query language that allows developers to search for dependencies inside a code base. It transforms the input queries into graph reachability patterns and searches for them in the dependency graph of the code base. Mens et al. [238] presented a language to define programming patterns for Smalltalk. It is possible to build search tools that identify code locations matching such a pattern in a code base. Long et al. [215] use a different approach. Their tool identifies API methods that access and change the same data structures and clusters these API methods together. Developers can then use queries to identify such clusters. Although all these approaches can help discovering code locations for systematic edits, they still need active participation from developers. ARES searches automatically for such locations.

YooHoo by Holmes and Walker [151] informs developers about code changes in other repositories or branches that can have an impact on their code. For that purpose, developers have to identify code entities that are relevant for their work. YooHoo then uses a dependency analysis to find all other related code parts. If one of these other code parts changes, it informs the developer and shows other code locations that could require the execution of the same systematic edit. However, developers have to identify the relevant code locations beforehand.

Moha et al. [242] developed a DSL that allows the specification of code smells (i.e., code characteristics that indicate deeper code problems [125]) and a system that uses this DSL to detect code smells in a code base. It is possible to detect systematic edits if they have the purpose of removing code smells. However, there are systematic edits that occur independently from code smells. In these cases, this DSL provides no help. In contrast to ARES, it is also necessary that the developers create a search pattern manually in the DSL whereas ARES only requires training examples.

ReSharper for Visual Studio [274] detects problems in C# (e.g., possible NullReferenceExceptions), in JavaScript, in Visual Basic, and in other languages. Developers can specify search queries for ReSharper. To fix a problem detected with a custom query, developers can specify a code transformation that ReSharper afterwards executes on identified code
locations. With placeholders in the query developers can even generalize the process.

All the above approaches have in common that developers have to actively interact with them. Either developers have to specify a search query or they have to provide code fragments, core dumps, or bug reports as input. The combination of C3 and ARES creates a powerful automatic system that is able to create patterns and to search for possible applications without manual effort.

2.3 Code Fragment Recommendation

This section covers analysis tools that recommend code fragments. In contrast to the tools discussed in the previous section, such tools require no direct input (e.g., in form of a search query) from developers. Thus, they help developers even if they are unaware of bugs or readability problems by analyzing their projects in the background. However, this category of approaches is too general to address specific systematic edits. It is only possible to prevent systematic edits with these tools. As described above, if the tools improve the code quality and reduce redundant code, developers have to execute fewer systematic edits.

CodeBroker by Ye and Fischer [334, 333] uses comments and method signatures to search for similar code fragments to prevent code clones during their creation. For the identification, CodeBroker uses term frequencies and method signature similarities. The tool is able to prevent systematic edits that affect code clones, but it offers no direct solution for executing systematic edits. Barbosa et al. [31] presented a recommendation system that looks for similar code fragments in a code base. It takes types and method calls in the exception handling into account and uses exception details to measure the similarities between code fragments. Based on this information, it shows developers examples for similar code fragments. With these examples, developers can improve their exception handlers and thus prevent systematic edits. However, it is not a tool that helps executing systematic edits.

Other search tools rely on available test suites. Code Conjurer by Hummel et al. [158] uses the current test suite to find code fragments in a data set that satisfy the tests. The system by Janjic and Atkinson [165] uses recently written code and a set of initial tests to recommend other test cases for the new code. Both tools can identify problems during
2.3 Code Fragment Recommendation

the development process early and thus they can reduce the number of systematic edits that appear during the maintenance of the software.

Clever by Nguyen et al. [254] manages code clones. It generates feature vectors for the ASTs of code fragments. The numbers in the vectors specify the number of occurrences of a feature in an AST (e.g., the number of AST nodes with label \textit{identifier}). Then it identifies clones by identifying identical vectors. If a developer changes a group of clones inconsistently, Clever detects this by comparing the differences in the ASTs of two clones and proposes a consistent change if possible. The applicability of Clever to systematic edits depends on the similarity of the feature vectors. If a systematic edit applies to code locations that have large differences, the feature vectors of the code locations are different and therefore Clever (in contrast to ARES) cannot recommend the change.

Several systems also use the interactions of developers with the IDE to gather data about the currently developed code. Reverb by McIntyre and Walker [232] tracks recent method changes in IDEs. In particular, it tracks changes to the CFG, method calls, types, literals, and code properties. Reverb uses this information to detect methods for which similar changes (i.e., the same systematic edit) could be necessary. For that purpose, it extracts the original version of the changed code properties and searches for similar ones inside a code base. This approach differs from ARES as it does not use patterns to search for code locations. It only searches for similar properties. For example, if an identifier \textit{foo} was changed into \textit{bar}, Reverb searches for all code locations that contain the identifier \textit{foo}. The more code properties of a code location are identical to the original version of the changed method, the higher is the similarity of the code location. The output of Reverb is a list with methods and their similarity values to the initial code change. It is the task of a developer to examine each method as Reverb provides only the similarity values as guideline. Thus, the developer still has to look at several methods to identify code locations that allow the execution of a systematic edit. Reverb also does not recommend transformed code. Developers have to perform the edit operations manually. This can lead to new mistakes due to attention failures.

The recommendation tool by Antunes et al. [16] also builds a model from the interactions of the developer with the IDE, as well as structural elements in the code base like inheritance and call hierarchies. Based on this model and the context in the IDE, the tool recommends
related classes, interfaces and methods to the current task of a developer. It detects related code fragments based on similar structural properties and the code parts used in the last interactions with the IDE. As before, this can help to improve the quality of the code and thus can avoid some systematic edits. However, it offers no solution in handling systematic edits in general. Kobayashi et al. [192] presented another approach that uses IDE interactions. Based on the collected information during these interactions, it identifies code fragments that were changed together within a time frame in the past. If a developer performs another code change that belongs to such a group, their system recommends the other code fragments automatically. The actual execution of the code change remains the task of developers. With RASCAL, McCarey et al. [230] presented a user-centered recommendation system. It mines repositories for code fragment usages (e.g., methods or classes) categorized by users and looks for users that used the same methods and classes as the current developer. Based on this information, RASCAL recommends the next likely code fragment to be used. Such a system prevents systematic edits by avoiding bugs due to helpful recommendations, but gives developers no support in executing them.

Mendel by Lozano et al. [216] recommends code fragments based on code properties of classes in the same class hierarchy. For example, if a developer creates a new class and all other classes within the same class hierarchy implement the method `run`, Mendel recommends the implementation of the `run` method.

Critics by Zhang et al. [340] supports developers during the review process. It extracts a pattern from a code change, identifies similar changes and also detects divergences from this pattern. Developers can adjust the generated patterns in an IDE to make them more general.

CBCD by Li and Ernst [210] searches in the program dependency graph (PDG) of a project for subgraphs that are isomorphic to PDGs of bugs. If a systematic edit changes the PDG, this approach can detect code locations that are also applicable to this systematic edit. However, it is restricted to code locations that have an isomorphic PDG. This is not a commonly shared property of systematic edits.

FrUiT by Bruch et al. [50] is a rule-based recommendation system. It creates rules from code fragment usages in a code base. If a rule applies to the current code of a developer, it recommends the action associated with the rule. For example, after the instantiation of a class, it recommends the most frequently used methods of this class. Like other
approaches, FrUiT can increase the code quality and thus can reduce the required systematic edits that developers have to perform on a code base. However, it is not applicable if recommendations for a specific systematic edit are required.

Santos et al. [291] compare three different approaches that recommend code locations for systematic edits. Their information retrieval approach uses a single code change as input. It is a text-based approach that splits the original part of a code change into terms. Then it computes the similarity between the original part and other code fragments in the code base with these terms. For example, if the input is a method, it computes the similarity of the input to all methods. Then the information retrieval approach ranks the possible code locations in the code base according to their similarity. For the first $n$ items with the highest similarity it recommends the code locations. The AST-based approach by Santos et al. also requires only one code change as input. It computes the LCS between the input code change and similar fragments in the code base. Then the AST-based approach ranks the fragments according to the size of the LCS and recommends code locations for the first $n$ top-ranked ones. Their structural approach uses two or more code changes as input and extracts their common structural properties. The properties include method signatures, package names, class hierarchy details, and other structural items. Then the structural approach searches for fragments with similar properties. If one is found, the approach identifies this fragment as a possible code location for a recommendation. To further improve the results, Santos et al. also presented a variant of the three approaches. In this variant, their tool also examines whether the edit script of the input code changes is applicable and whether or not the code base after the application compiles. Only then, the system recommends the code location.

All approaches from Santos et al. have in common that they can lead to many false positives. The information retrieval and AST-based approaches both create a list of code fragments ranked by similarity. Depending on the threshold, a lot of code fragments can fulfill the conditions. The structural approach can also lead to a lot of false positives if the input examples share only very few properties. Under these circumstances, the pattern of the structural approach is very general and fits many code fragments. The search in ARES is more specific and produces fewer false positives as its patterns also consider the control flow. A straightforward application of the edit script also has disadvantages.
If there are variations between the edit examples (e.g., the assert statement from the introduction), the edit script does not fit to all locations of a systematic edit and useful recommendations get lost.

The solutions described above do not cover the complete range of systematic edits that consists of searching for code locations and executing the necessary code transformations. Many of them are also only useful for a subset of systematic edits occurring in large software projects. Some of them even offer no direct help for systematic edits, but only increase the code quality. This can prevent some systematic edits from happening, but is no general solution to the execution of them. Additionally, most of the approaches only identify the code locations and do not offer help in transforming code for systematic edits. Thus, a developer still has to execute them manually. As this remains a boring, monotonous, and repetitive task, these tools do not prevent attention failures and the introduction of additional bugs during the execution of systematic edits.

2.4 Code Fragment Mining

The code fragment recommendation systems in the previous section identify similar or related fragments during the development process. The tools in this section use a code base provided by a developer as input and extract relationships in it. These relationships can reveal identical or similar code locations. If developers execute a systematic edit on one of these locations, it is likely that they also have to execute it on code locations containing identical or nearly identical code. Thus, the identification of these locations provides help for systematic edits. However, some systematic edits do not require a high similarity of the code locations. In such cases, code clone detection is insufficient.

Dup by Baker [30] identifies identical or nearly identical code fragments (i.e., code clones) inside a code base with a textual and line-based approach. For the identification of identical lines, Dup removes whitespace and compares the remaining characters. To identify nearly identical lines, Dup also accepts different identifiers, field names, constants and macro names in a line. To speed up the search in large code bases, Dup uses parameterized suffix trees [29]. Although Dup was not created as a recommendation tool for systematic edits, it sometimes can help to identify code locations for them. This is possible if developers
have to execute the systematic edit at all identical or nearly identical code locations.

Baxter et al. [35] use an AST-based approach. Initially, their approach hashes all subtrees of the code base. To increase the probability of identifying similar trees, the hash value computation does not include the leaves of the subtrees. If the hashing produces subtrees with identical values, the approach further refines the results by applying a node-based similarity measurement that takes the leaves into account. This approach detects code clones. As discussed in Sec. 1.1, it is possible to remove some of the detected clones. This can prevent systematic edits from happening. Sometimes, the approach by Baxter et al. helps identifying code locations for systematic edits. In these cases, a systematic edit that affects code clones has to be applied to all clones at the same time. However, due to the hash value approach, the tool does not detect clones if they contain additional statements (e.g., the `assert` statement from the introductory example).

The tool CCFinder by Kamiya et al. [176] transforms a code base into a sequence of tokens and identifies identical subsequences in the resulting token stream. To identify not only identical clones, it uses several transformations that normalize the code. For example, one transformation surrounds single statements in the `then` and `else` parts of an `if` statement with a code block. Like the approach by Baxter et al., additional statements and larger changes make the detection of code clones less likely and thus CCFinder does not detect all code locations for a systematic edit. DECKARD by Jiang et al. [170] also uses an AST-based approach. It creates feature vectors for larger subtrees in the ASTs of a code base. Each element in the vector corresponds to a specific AST label (see Sec. 3.1) and holds the number of occurrences of this node type (e.g., identifier) in the subtree. Then DECKARD uses locality sensitive hashing [80] on the vectors. This hashing has the property that vectors with a small Euclidean distance have the same hash value. Thus, if two code fragments use similar AST node types, they have the same hash value. DECKARD presents all subtrees with the same hash value as potential code clones. Again, the potential to detect systematic edits depends on the variance of the code clones. If there are too many statements that differ between code locations, DECKARD does not identify the code locations as clones and thus provides no help for such systematic edits. In contrast to C3, it only detects similar code fragments and not similar code changes. CloneDetective by Juergens et al. [174] is a
Clone detection framework that can work on the granularity of lines, words or tokens. It converts a code base into a list of these items and then identifies identical subsequences of the items in the code base. CloneDetective presents these identical items to the developer. This only identifies identical code clones, in contrast to other more flexible approaches like SourcererCC [288, 289]. The latter is another clone detector that first extracts the code blocks from a code base and computes the token frequency for each block. Then it creates an index for each code block and the used tokens. This makes it possible to identify all blocks that share a large set of tokens. Compared to other clone detection tools, SourcererCC is more flexible but still requires a certain number of shared tokens between the code locations. As the purpose of SourcererCC is not the execution of systematic edits, but the identification of similar code locations, it does not offer help for the code transformations of systematic edits. However, it can help in the detection of code locations for systematic edits as long as the code locations are very similar.

In contrast to text-based, token-based and AST-based approaches, there are also works that focus on the extraction of rules. Engler et al. [106] presented such an approach. It infers logical rules with static analysis from a given code base. A simple logical rule from their approach is that a variable has to be not null at a specific code position. It warns developers if a code location violates these rules. It is able to discover code locations of systematic edits in this way. The precondition of this discovery is that it has to be possible to express the systematic edit with logical rules that are valid in the current code base. However, expressing refactorings that do not change the semantics is impossible with such rules. Thus, all systematic edits that only perform a refactoring are not covered. Again, the system offers no support in the execution of a systematic edit.

Arvocerde et al. [19] introduced a tool to recommend code locations that contain architectural anomalies. Similar to the work by Engler et al., this only identifies some of the systematic edits in a code base. Only if the systematic edits address such anomalies, it helps in the identification of corresponding code locations. Bugram by Wang et al. [315] splits a code base into tokens and trains an n-gram model with them. Then it computes the probability of all token sequences according to the trained model and reports the token sequences with the lowest probability as bugs. If Bugram reports several problems that require the
same code transformations to fix, it has identified the code locations of a systematic edit. However, systematic edits occur several times in a code base and thus the token sequence also occurs several times. This increases the probability of the token sequence and makes it less likely that Bugram detects it. Additionally, Bugram does not execute the code transformation. Similar to Bugram, the bug fixing tool by Ray et al. [272] detects rare token sequences. It uses the cache language model [308] to identify code lines with bugs. It uses the sequences of Java tokens that appear in a large training set as input. If a code line does not fit the trained model, the tool warns the developer of a problem. Similar to Bugram this can detect the code locations of a systematic edit. However, both tools were created for a different purpose and thus do not offer a full solution to systematic edits.

PR-Miner by Li and Zhou [212] identifies code fragments that appear frequently together. For example, it identifies method calls that frequently follow each other in a code base. Sets of such code fragments (e.g., method calls) are grouped together and form programming rules. With these rules, PR-Miner detects violations, i.e., code locations in which not all items of a set appear. If PR-Miner detects the same violation at different code locations and all require the same patch, it has identified a systematic edit. Thus, in such cases it offers help in their execution. In their approach, Lozano et al. [217] also mine for associative rules in a code base. In contrast to the other approaches described in this section, they focus on filtering and grouping of these rules. It is possible to use their techniques to optimize or enhance tools like PR-Miner. AntMiner by Liang et al. [213] also identifies rules in code bases. It first slices a code base into subparts based on critical operations. In the version from 2016, AntMiner supports two critical operations, error-prone function calls and improper return values. Then AntMiner normalizes the code and hashes the statements in the subparts. On the sequences of these hashed statements it uses frequent itemset mining. Each identified itemset forms a rule. If a sequence of statements in the code base violates such an itemset, AntMiner warns developers about it. Like other mining approaches, this helps in identifying code locations of systematic edits.

In contrast, the approach by Krinke [197] identifies similar PDGs. In this approach, two PDGs are considered similar if they share the same paths from the root to the nodes of depth $k$. Gabel et al. [127] proposed to use similar PDGs to identify semantic code clones. Both approaches
only detect certain types of code clones and are not applicable to all systematic edits.

As the information in a repository does not reflect intermediate states of the development process, Proksch et al. [269] presented a system that captures the development process in IDEs and also includes intermediate code snapshots. This data can be used by other tools to address systematic edits.

Whereas the approaches described above identified similar code locations, this is insufficient for all kinds of systematic edits. It is possible that the code locations that belong to the same systematic edit have very few code parts in common. For the decision whether two code locations require the execution of the same systematic edit or not, the applicability of the same code transformations is more important. This means that even if two code locations are very different, if they cause similar problems and require the same patch to fix it, they actually do form a systematic edit. To address this issue, this thesis presents C3 that mines repositories for similar code changes instead of similar code locations.

## 2.5 Code Transformation

The previous sections focused on the search for code locations. The search alone is insufficient for the execution of systematic edits. An all-encompassing tool for systematic edits has to find the associated code locations and has to perform the necessary code transformation automatically. The latter is necessary to avoid mistakes that can occur during the manual execution of edit operations. This section discusses techniques that allow the definition of code transformations and provide the means to apply them automatically to a code location.

Nix [258] proposed an approach that infers text change patterns in an editor from examples. This has the drawback that it ignores the hierarchical structure of code and thus the application of the pattern can lead to an incorrect syntax. It is also the task of the developer to specify the code locations for the transformation.

TXL by Cordy et al. [68], REFINE by Burson et al. [51], and A* by Ladd and Ramming [200] are DSLs to formally specify code transformations. Boekhold et al. [44] introduced a transformation DSL for ANSI C and the tool CCT to execute defined transformations. Manniesing et al. [224] use the same tool to vectorize program code. The HULA update
2.5 Code Transformation

language by Erwig et al. [108] allows the definition of code transformations for the Haskell programming language. All these approaches share the drawback that they require the developer to learn a new language to express code transformations. This can be a barrier that prevents developers from using such approaches for systematic edits. Additionally, this requires more effort compared to ARES that creates the transformation specification automatically. Due to a steep learning curve of the approaches mentioned above, it is also possible that the manual transformation of a systematic edit into the DSL equivalent is incorrect. This can lead to additional problems in the code and again increases the effort for developers.

Other tools learn code transformations from examples. Masui and Nakayama [229] presented a technique that learns from repeated text changes in an editor. If a user performs the same changes more than once, their technique creates a text editor macro that the user can invoke. The approach by Miller and Myers [241] allows developers to select several code locations for simultaneous editing. Then the presented tool performs each change in the editor at the same time at all locations. With Linked Editing by Toomim [305] it is also possible to change code clones at the same time with an editor. CloneTracker by Duala-Ekoko and Robillard [95] uses detected clones as input and helps developers in changing a set of clones simultaneously in an IDE. All of these tools have in common that they can only apply identical or nearly identical code transformations.

Some tools allow the definition of code transformations on ASTs. For example, Stratego/XT [313, 312] provides a language for AST transformations. As input it uses the language independent ATerm format [48]. Baxter et al. [34] introduced a program transformation toolkit that also works on ASTs. With this toolkit, developers can define transformation rules in a rule specification language and apply them to ASTs. In contrast to Stratego/XT, the toolkit also searches for applications of the defined rules in a code base. Similar to the DSLs and in contrast to an example-based system, it is the task of the developer to specify the transformation. Compared to ARES that learns transformations from examples, the manual writing of transformation is more effort. It is also possible that a manual transformation description introduces errors in the code as many developers are unfamiliar with transformation languages and ASTs.
Boshernitsan et al. [46] developed iXj, a graphical tool to define and execute complex code transformations. Again, this causes a higher workload on the developer compared to an example-based system as it still requires a manual definition of code transformations.

For Jigsaw by Cottrell et al. [70], developers have to specify a method as seed for the recommendation process. Then they can specify a target location at which Jigsaw inserts the functionality of the seed method. Based on this information, Jigsaw creates a recommendation that integrates the functionality of the seed method. Here the developer only has to give an example and the code location that has to contain the functionality from the example. However, in contrast to ARES, it does not search for code locations automatically.

With the aid of program metamorphosis by Reichenbach et al. [273], developers can perform five basic program transformations (rename, split, delegate, cut, and paste). In contrast to standard refactorings, these transformations may break the behavior of a program. For each such behavior altering step, the developers can choose between accepting this behavior or reverting the transformation. This approach is orthogonal to an example-based system. It is possible to break down the transformations from example-based systems into these 5 basic transformation steps. Thus, it is possible to combine this approach with ARES.

Tate et al. [302] developed a system that creates compiler transformations from an example. Their system first builds a program expression graph (PEG) from the original and the modified code in the example and then combines them in a PEG with equality information (E-PEG) to proof that they are equivalent. If the proof and thus the combination is successful, the tool by Tate et al. transforms the E-PEG into the most general one that still contains the proof that both PEGs are equivalent. Then it computes an optimization rule from this general E-PEG. In contrast to systematic edits, this approach is limited to semantics preserving transformations.

In general, the code transformation approaches lay the burden of the transformation definition on the developer. An example-based system like ARES has the advantage that it uses the provided examples to determine the transformation. It does not require that the developer learns a transformation DSL or acquires knowledge about ASTs or other compiler structures. As ARES transforms code automatically, developers do not have to spend time on the definition of code transformations.
2.6 Code Completion

Recommendation systems that address systematic edits usually recommend source code to support developers in their work. This is also true for code completion systems, although on a smaller scale. Often tightly integrated in an IDE, they propose the next method call or complete the current identifier. For example, the code completion system by Bruch et al. [49] uses machine learning on a code base to optimize the prediction mechanism. The system then orders the possible completion options based on the learned model. D'Souza et al. [94] developed a code completion system for Python. Their approach extracts API usages from Github projects and uses them to train a nearest neighbor classifier.

Abebe and Tonella [2] presented an ontology built from the identifiers in a code base to create a system that recommends identifiers to developers. It requires the last typed characters in the editor as input and uses the textual similarity to identifiers in the ontology to create identifier recommendations. Asaduzzaman et al. [21] propose a context sensitive method for code completion. As context their system extracts keywords, method names, class names, and interface names within four lines of previous API changes in repositories. Then it compares the extracted contexts with the last four lines in the current context of a developer. For this comparison, the approach uses string similarity measurements (e.g., the LCS algorithm). The completion system then recommends the most similar code fragments (e.g., method calls) to the current context.

Precise by Zhang et al. [337] recommends parameters for API usages. Similar to Bruch et al. [49], Precise mines a code base and then searches for usages similar to the context in the IDE. Nguyen et al. [253] presented a system that learns Android API usages from byte code and is able to recommend the next method in a method call sequence. Santos and Myers [290] propose annotations that help IDEs in recommending suitable code completions for API usages. API developers can use these annotations to provide more information about their methods (e.g., StaticFactory). The IDE can use this additional information to rank and filter the list of possible code completions.

Proksch et al. [268] provide a benchmark for recommendation systems. The benchmark data consists of recorded interactions of developers with an IDE. It targets code completion systems that recommend the next method call.
In contrast to recommendation systems for systematic edits, the code completion systems described above do not propose transformed code, but only the next useful code fragment. ARES targets larger changes.

2.7 Refactoring

Refactoring tools perform predefined transformations that preserve the semantics of changed code. In some cases, refactoring approaches also search for refactoring opportunities, i.e., code locations for which a refactoring is applicable. However, in contrast to more general recommendation systems, they often work with predefined patterns. Many of the following tools also recommend refactorings like extract method that removes redundant code (i.e., code clones). Using such refactorings reduces the number of identical or nearly identical code locations. This reduces the number of systematic edits that developers have to execute as fewer code locations require the same patch.

Atkinson and King [23] presented several algorithms that detect different refactoring opportunities by examining the ASTs of a code base. For example, one algorithm identifies all public fields and recommends encapsulating them with getters and setters. To execute the refactorings, this approach relies on the functionalities provided by the IDEs. There are also different tools that add functionality to current IDEs. Garrido and Johnson [129] proposed one for C that can handle preprocessor directives. Feldthaus et al. [113] presented a similar refactoring tool for JavaScript.

Aries by Higo et al. [143] detects code clones with CCFinder and then recommends clones suitable for refactoring based on coupling measurements. The approach by Hotta et al. [156] removes code clones for which the Form Template Method is applicable. This refactoring removes the clones and replaces them with a unified template method. To identify suitable code clones, this approach relies on program dependency graphs. Both approaches remove code clones and thus can prevent some systematic edits. They offer no further support for developers. CeDAR by Tairas and Gray [301] also combines clone detection with code refactoring techniques. It can use the output of different clone detection tools (e.g., CCFinder) to detect refactoring opportunities in the clones. In combination with the Eclipse IDE refactoring engine, it can change all clones at the same time. Therefore, CeDAR can perform
systematic edits if they apply to code clones and if their changes are only refactorings and do not affect the semantics of the code.

Several tools focus on class refactorings. Abdeen et al. [1] presented a technique to reduce coupling between packages. It uses simulated annealing to optimize the class distribution across packages. This can minimize the coupling and increase the maintainability of the code. Fokaefs et al. [121] presented an approach that identifies Extract Class refactoring opportunities and recommends these refactorings to the developer. To identify these opportunities, it uses hierarchical clustering [336] (see Sec. 5.3.2) on code fragments. R3 by Bavota et al. [33] recommends the movement of classes to different packages. To identify suitable classes, it analyzes the identifiers, comments and string literals and uses this information to identify semantic relationships. It combines this semantic information with class dependencies to find closely related classes and recommends grouping them together. However, all three approaches only cover systematic edits concerning classes.

Maruyama and Shima [228] presented an approach that performs refactorings for library developers. For that purpose, their approach analyzes the effect of past library uses. Based on this information, it refactors the library methods and separates code that library users did not modify from code that they modified frequently. Tsantalis and Chatzigeorgiou [307] created a system that recommends extract method refactorings. It uses block slicing to extract all statements that compute a variable. They also created a second system [306] that recommends move method refactorings. It analyzes the code fragments that a method accesses and determines whether or not the code fragments are part of the class surrounding the method. If the movement of this method into another class will increase the number of code fragments that access the new surrounding class, then the system recommends the refactoring. Methodbook by Oliveto et al. [260] is a similar approach. It searches for the feature envy code smell in a code base. Feature envy occurs if a method uses more fields and methods from a class that is not its own. With the help of these metrics Methodbook detects whether a class accesses more fields and methods from another class and proposes a refactoring if necessary. Such refactoring approaches increase the quality of the code base. This prevents errors and thus can reduce the number of occurring systematic edits.

O’Keeffe and Ó Cinnéide [259] proposed a tool that automatically performs refactorings on a project while monitoring its quality. The
goal of this tool is to change the project with refactorings to increase its quality. It can use different algorithms (e.g., hill climbing) to guide the search for a project with higher quality. For the quality evaluation it uses several predefined metrics. If the results of these metrics are positive, the quality search continues with the next refactoring. The approach by Seng et al. [292] detects refactoring opportunities with a genetic algorithm. As fitness function, it uses a combination of software quality values. Both tools improve the quality of the source code and thus can reduce the number of required edits. This is especially true for refactorings (e.g., extract method) that remove redundant code.

WitchDoctor by Foster et al. [123] detects if a developer starts a manual refactoring. It analyzes the changes of this developer in the IDE and compares these operations with a refactoring rule database. If the operations match such a refactoring rule, WitchDoctor applies the refactoring and presents it as a recommendation. In contrast to ARES, it requires predefined rule specifications and it is unable to learn rules from examples.

BeneFactor by Ge et al. [131] compares code changes in the IDE with predefined refactoring patterns. If it detects one, it informs the developer about the applicability of the pattern. Subsequently, it can perform the remaining code changes to complete the refactoring automatically. Mens et al. [239] presented a formal specification of refactorings that is based on graph transformations. This can be used to ensure behavior preservation of refactorings in tools. Both approaches provide a general improvement of code refactoring.

However, for systematic edits the tools described in this section are not general enough. Many systematic edits also change the semantics (e.g., bug fixes) and thus are not covered by refactoring techniques.

### 2.8 API Centered Works

New library versions regularly change their API. Such updates can make methods or classes deprecated and in some cases even break the code that uses these libraries (see Xavier et al. [321]). This is such a common occurrence that several tools specifically target these problems. In this narrow context, they also create recommendations for systematic edits. However, they provide only limited support for systematic edits that affect the control flow or that refactor code fragments.
2.8 API Centered Works

SemDiff by Dagenais and Robillard [78, 79] recommends new alternatives for framework methods that are no longer available in newer versions. To serve that purpose, SemDiff analyzes method changes and usages inside the frameworks. The resulting recommendation consists of alternative API calls and includes suitable parameters.

Zimmermann et al. [342] developed ROSE, a tool that learns associations between code fragments based on API changes from code repositories. The associations are sets of code fragments that were changed together in the past. For example, ROSE can learn that a change to a variable v was made together with a change to the method init in the past. With such associations, ROSE can react to code changes performed by developers. If this change has associations to other fragments, ROSE recommends them as code locations that might also require the executed changes. Similar to ROSE, TARMAQ by Rolfsnes et al. [280] uses rule mining to identify changes that were related in the past. With this information, their system reacts to new changes and recommends files that developers might have to change accordingly. In contrast to ROSE, TARMAQ also creates recommendations if the new changes affect files that were not present in or related to the historic data of the project. For example, if TARMAQ learned that the fragments a and b were changed together in the past and a developer changes the fragments b and c, it recommends to also change a. Due to the previously unknown change c, ROSE does not create a recommendation. LIBSYNC by Nguyen et al. [252] mines associations based on API usages and then recommends appropriate changes concerning such API usages. Their approach takes into account that many API changes affect only method calls and inheritances. Saied and Sahraoui [287] presented another approach for API usage mining to identify API calls that are used together. Their approach extracts API methods, used fields in the methods, parameters and local variables, encodes them in vectors and clusters them with DBSCAN [109] (see Sec. 5.3.3) to identify similar API methods. Additionally, their approach extracts API methods that are used together in applications. According to Saied and Sahraoui, this can help developers in understanding existing libraries. The approach by Azad et al. [26] learns API change patterns from StackOverflow, instead of code repositories. Similar to ROSE, it can recommend code locations that also require changes. These mining tools all focus solely on API changes. Control flow changes like the insertion of the if statement in the introductory example are not covered. Thus, they are applicable
to only a subset of systematic changes and not usable in general. They also do not recommend code changes, but only provide code locations.

In contrast to the aforementioned tools, CatchUp! by Henkel and Diwan [139] offers support for API changes in libraries. With this tool, library developers can record refactorings that affect the API. Then, a library user can replay the recorded refactorings to adapt their program to the new library version. This automatically performs the systematic edits that the library API change causes. However, the drawbacks of this approach are that the library developers have to record their changes and CatchUp! only supports API refactorings.

Several tools recommend the usage of APIs. For example, Holmes et al. [152, 150] presented an approach that analyzes the current source code and extracts the structural context of this code (e.g., used types and used method names). Their system then searches for similar structures in a dataset of API usages to create recommendations. The generated recommendations include a class diagram presentation, the similarities of the structures to the different API usages and source code examples from the repository. MAPO [322, 341] is also a tool recommending API usages. It first extracts API sequences from a code base and then uses hierarchical clustering to identify sequences with similar names and methods. Afterwards, it uses a frequent subsequent miner on each cluster to create patterns. Developers then can use MAPO on an API call and the system recommends patterns containing similar API calls. Duala-Ekoko and Robillard [96] presented API Explorer. It recommends API classes (and their methods) that have a relationship with the class of the current object in the IDE cursor position. To identify the related methods, API Explorer builds a dependency graph between API elements from the byte code of the API. Fowkes et al. [124] presented a probabilistic API miner. Compared to MAPO, this miner produces fewer redundant usage patterns, i.e., patterns that have a similar sequence of API calls. Ichinco et al. [161] introduced Example Guru, an API recommendation system for novice programmers. They implemented the rules that determine useful recommendations by hand. These kinds of approaches give developers additional information during the development process and thus only avoid systematic edits due to higher code quality.

Kawrykow and Robillard [180] presented a static analysis tool that recommends API usages. Their approach looks for methods in a code base that are a reimplementation of API methods. To identify methods
that are similar to each other, it uses abstract method bodies consisting of fields, methods and types referenced in the bodies. Therefore, their system identifies API related methods and can prevent systematic edits due to elimination of redundant code.

Kapur et al. [178] designed Trident to help developers in adapting method calls and type names to a changed API. Trident provides a sophisticated search-and-replace IDE for that purpose. Developers can specify search parameters to find specific invocations of the old API. Afterwards, they can define the refactoring that transforms the code at these search locations into usages of the new API with Trident’s graphical UI. Trident presents the transformed locations as recommendations to the user for review. As with the other tools presented above, it is again left to the developer to specify search queries and the transformation. Even with tool support, this is more effort compared to an example-based system as long as examples are available.

Hora et al. [154] address the problem of identifying common change rules by analyzing change patterns from code repositories. To create the actual rules from code changes, their approach examines deleted and inserted method invocations. If two rules only differ in one invocation, it merges them with the help of wildcards. Based on these learned rules, their approach browses a code base and recommends code locations that violate one of the learned rules. Due to the focus on method invocations, this approach is only applicable for systematic edits that involve deleted and inserted invocations.

With the help of mining API usages, the framework by Kim et al. [185] enriches API documentations with code examples. For that purpose, they extract usages from a code base and encode these usages with vectors (proposed by Jiang et al. [170]). Following this, the framework uses k-means, a ranking system, or a hybrid approach to identify relevant code examples that it embeds in the API documents.

In summary, the state-of-the-art API tools either focus on the detection of code locations that require an adaptation to a new API, or they support developers in using the API. Defining and executing the transformation that adapts the code to a new API remains the task of the developer. In contrast, ARES and other example-based tools present the transformed code automatically to developers as recommendations.
### 2.9 Program Repair

This section discusses program repair approaches that fix specific problems in a code base. As this is a large field, it is only possible to provide some examples that illustrate the differences to recommendation systems. In contrast to recommendation systems, program repair approaches require rule specifications or test cases. Based on this input, they can ensure that a program transformation fixes a rule violation or a failed test case. The disadvantage of these approaches is that developers have to provide such specifications or test cases. If they are unavailable, their creation causes considerable additional effort for developers. As the focus of this thesis is the recommendation of source code, the selected program repair examples below have the same goal. Other approaches [83, 104, 105] repair problems in running programs and thus are not discussed here.

GenProg by Le Goues et al. [206, 205, 318] automatically repairs program code with genetic programming. The genetic operations on the individuals perform AST transformations. As input, their system requires a set of test cases that are used as fitness function. The more test cases an individual (i.e., the patched code) passes, the better its fitness value. Kim et al. [184] first identified human bug fix patterns in patches. Based on these patterns, they created PAR. It identifies errors in programs with the help of correct and incorrect tests and code coverage. If a fix pattern is applicable for a code location, PAR applies the pattern automatically. Then it runs the test suite again and keeps this fix if it improves the test results. CFIX by Jin et al. [171] uses static analysis to detect concurrency bugs and patches them in LLVM bitcode. To validate the patches, it uses guided testing. All three approaches require either failing tests or a code location that reveals a concurrency bug in the case of CFIX. Both cases do not apply for systematic edits that contain refactoring transformations as such changes preserve the semantics and thus do not change the results of test cases.

Castro et al. [55] proposed to use logic adductive reasoning to diagnose and correct inconsistency in programs. A developer can enter a consistency check formulated in a Prolog dialect. If parts of the check fail, the system looks for predefined actions that are able to correct the failed parts and recommends them to the developer. FixMeUp by Son et al. [296] detects access control problems and recommends repairs. As input, it requires the access policy of the program. The policy has to specify a conditional check that validates the access rights and sensitive
calls. Based on this policy, FixMeUp uses static analysis to identify violations and recommends the conditional checks that a developer should add to the program. Specifying the access policy, for example with annotations, is again left to the developer. AutoFix-E \[317, 264\] makes use of the contracts available in Eiffel programs. These contracts specify pre-conditions, post-conditions, and intermediate conditions. If an automatic or manual test detects a contract violation, AutoFix-E generates a repaired program. To create the repaired program, it uses a behavioral model that describes the object state changes during successful runs of the program. With this model, AutoFix-E is able to create the appropriate fix. Castro, FixMeUp, and AutoFix-E all require a specification of consistency checks that they use to repair the program. Creating the specification is the task of the developer and thus means additional effort if they are unavailable beforehand. Automatic repair tools are also limited to systematic edits that influence the semantics.

### 2.10 Summary

This chapter gave an overview of state-of-the-art static analysis tools. However, the discussed tools cover only a subset of the requirements that a recommendation system for systematic edits has. Most tools either search for specific code locations or perform code transformations. Only a few tools cover both parts that are required to execute systematic edits. Some of the discussed approaches also have restrictions that an example-based system like ARES does not have. For example, most mining approaches expect a high similarity of the code locations to identify relationships between them. There is also a use case that is not covered by the state-of-the-art tools. Developers sometimes perform two or more similar code changes in a row. Thus, they automatically generate input data for an example-based recommendation system. However, the tools presented in this chapter cannot make use of these two similar code changes directly.

As this thesis focuses on static recommendation system tools, which are a subpart of general static analysis tools, this chapter first gave an overview of the static analysis tools that developers in large software companies use. As these tools rely on predefined patterns, their use for systematic edits is limited. It is possible to provide user-defined patterns, but this is additional manual effort for developers. The second section discussed code search approaches. The search engines are able
to discover code locations but have the disadvantage that developers
have to formulate queries or use other input mechanisms to identify
code locations for systematic edits. This can be complicated for some
systematic edits if they cover large differences in the code locations.
Tools in the code fragment recommendation category work without
direct input and require less effort. They search for specific code pat-
terns and properties in a code base. They rely on either predefined pat-
terns or learn patterns from other sources (e.g., a code base, an IDE,
repositories). However, in contrast to tools for systematic edits, they
do not take the code transformation into account. Additionally, code
locations of a systematic edits share the characteristics that the same
patch is applicable to them. It is unnecessary that code locations that
require the execution of the same systematic edit have a high similarity.
This limits also the applicable of the code mining techniques discussed
above. For systems like ARES similar code changes are more impor-
tant than similar code fragments. Therefore, Sec. 5.1 examines mining
approaches for code changes in detail.

This chapter also presented different techniques for code transfor-
mations. In contrast to approaches that use examples, they require that
a developer specifies the transformation manually, either in a language
or in an IDE. If examples are already available, the creation of a transfor-
mation is additional overhead for developers because example-based
systems can compute the transformations automatically. Nevertheless,
code transformations are still essential, even for example-based sys-
tems. Example-based systems create a pattern from the input examples.
The generated patterns contain the information required to search for
code transformation as well as the information how to transform them.
Considering static recommendation systems, the most fine-grained
ones are code completion tools. Although helpful during the develop-
ment process, they lack the ability to perform larger code transforma-
tions that are essential for systematic edits. The discussed refactoring
tools transform code on a larger scale. However, they are restricted to
changes that preserve the semantics of programs. Thus, they only sup-
port developers in a limited subset of occurring systematic edits. This
is also true for API centered techniques. They only provide support for
systematic edits if the similar code changes affect methods or inter-
faces. The last group of tools presented in this chapter were automatic
repair tools. As input, they require more than simple examples. They
also need test cases or provable specifications to solve problems. Thus, they also have additional preconditions compared to systematic edits.

In summary, this chapter showed that the current state-of-the-art tools are insufficient for the complete spectrum of systematic edits. ARES requires only examples to search for code locations and to transform them automatically. If $C3$ delivers the required examples, even the time-consuming example search works automatically without any involvement of developers.
3 Tree Differencing

Trees sprout up just about everywhere in computer science.

---

Donald Knuth, Combinatorial Algorithms

Trees play an essential role in example-based recommendation systems. LASE and ARES both use trees to represent the code changes they use as input. To identify common code and the differences between the input examples, both tools compute the differences on these trees and express them as lists of edit operations. Based on these lists, the recommendation tools create their patterns. However, creating a short list of edit operations that expresses the differences between two trees is computational expensive and complicated. To address this problem, this chapter presents MTDIFF. It is a novel tree differencing algorithm that is able to describe differences in trees with shorter lists of edit operations. Nevertheless, before this chapter focuses on trees and MTDIFF, it is useful to discuss text-based and token-based approaches to show limitations and disadvantages for tools like ARES.

Code repositories (e.g., Git [132], Mercurial [240]) or code repository hosting platforms (e.g., GitHub [133]) use a textual representation to describe the differences in commits (i.e., the file differences between two versions). The Unix tool `diff` also uses a textual representation for expressing differences in source code files. It executes Myers algorithm [246] on code lines to detect changed lines and thus changed code. For the code change in Fig. 3.1 `diff` detects two deleted and two inserted lines. However, the change in the first line just adds whitespace and the change in the second line just adds a pair of additional parentheses. For tools like ARES such differences are too verbose because they detect changes that are purely textual and thus do not influence
3 Tree Differencing

Original code:  Modified code:
01: f1();               f1();
02: f2((arg));       f2(arg);

(a) Simple example for $\text{diff}$.

(b) The output of $\text{diff}$ for the example.

Figure 3.1: Code change computed with $\text{diff}$. The boxes show the edit operations: green for insertions, red for deletions.

the semantics of the program. To avoid recommendations that do not influence the actual code, recommendation tools have to ignore such changes. The changes in Fig. 3.1 even affect more sophisticated text-based algorithms. LHDiff by Asaduzzaman et al. [20] tracks line changes across file versions. It initially uses $\text{diff}$ to identify unchanged lines. Then it computes possible matching candidates for each remaining line based on the similarity of their simhash [56] and other heuristics. This minimizes the number of differences per line, but still, the approach reacts to the additional parentheses in Fig. 3.1 and reports a deleted and inserted line. LDiff by Canfora et al. [54] is another approach that detects line changes between file versions. Like LHDiff, it first detects unchanged lines with $\text{diff}$. Then, it separates the code into hunks (i.e., sequences of adjacent lines). To identify moved code lines, it computes the Levenshtein distance [209] (LVD) between hunks in the original and the modified file. However, LDiff still creates delete and insert edit operations for both lines in Fig. 3.1 as the characters in them were changed.

An alternative are token-based approaches. These are unaware of the tree structure of the source code. As a consequence, they cannot determine whether or not the inserted parentheses in line 2 change the semantics. Skipping parentheses entirely is not an option for source code differencing as in many cases parentheses have an impact on the semantics (e.g., $(1 + 2) \cdot 3$). Thus, a token-based approaches often detects edit operations that do not affect the semantics like the parentheses in Fig. 3.1. This is unsuitable for ARES as such additional edit operations can make a pattern too general and thus useless (see Chapter 4). Therefore, token-based approaches are insufficient for
ARES. Nevertheless, plagiarism detection (e.g., JPlag [266]) or code clone detection (e.g., DECKARD [170]) often rely on tokens as the computation of code differences on token streams is faster compared to the computation on trees. The latter is more complex and more time consuming.

ARES uses a tree-based representation on ASTs. This has the advantage that it is independent of tokens that do not influence the semantics (e.g., some parentheses) and whitespace. However, detecting differences between two trees is complex. Computing the shortest edit scripts, i.e., the smallest number of edit operations that transform an original tree into a modified one is time consuming. If the algorithms detect code movements, it is even worse.

If only the edit operations update, delete or insert are used, RTED is currently the best available algorithm. It has a time complexity of $O(n^3)$ which is optimal for this problem. If the movement of tree nodes (or subtrees) is an allowed edit operation, the problem is NP-hard [41, 45]. Thus, an optimal solution is no longer feasible and the algorithms that solve these problems have to rely on heuristics.

This chapter introduces such an algorithm called MTDIFF. It uses heuristics to detect code movements. Additionally, MTDIFF makes use of six optimizations that are also compatible with other tree differencing approaches. MTDIFF and the optimizations are based on previous work [89]. After an introduction into tree differencing in Sec. 3.1, this chapter discusses the state-of-the-art approaches in detail in Sec. 3.2. The following section presents six optimizations that reduce the number of unnecessary edit operations in the results of current state-of-the-art approaches. From the six presented optimizations in Sec. 3.3, one works as pre-processing and five work as post-processing steps. They all do not depend on the tree differencing algorithms they optimize. Additionally, this chapter discusses the details of MTDIFF in Sec 3.6. MTDIFF is a novel tree differencing algorithm that focuses on the detection of code moments. Due to heuristics that detect such movements, MTDIFF further reduces the number of unnecessary edit operations and is able to express code changes with a shorter edit script. Shorter edit scripts for the same code change have the advantage that ARES creates less general patterns that have a higher precision and accuracy. Sec. 3.6 concludes this chapter with an evaluation of the six optimizations and MTDIFF. This evaluation also contains a short study of the connection between shorter edit scripts and helpfulness for developers.
3 Tree Differencing

3.1 Tree Differencing on ASTs

ARES relies on ASTs to represent code structures. The differences between the ASTs of the input examples are the base for the pattern creation process. Similar to compilers, ARES uses a parser to transform its input examples into ASTs. ASTs are rooted, ordered, labeled trees [5]. For example, \textit{Call}_o and \textit{Call}_m are the root nodes in the trees in Fig. 3.2. Parsers often add artificial root nodes for complete files. ASTs are also ordered. Changing the position of AST nodes (e.g., statements) corresponds to a syntactical change in the code. This can also change the semantics of the code. As shown in Fig. 3.2, each node in the ASTs has a label $l$ that corresponds to the type of the node (e.g., \textit{Name}). Additionally, some nodes have a value $v$. To simplify the tree diagrams in the remaining parts of this thesis, the tree nodes in the diagrams only contain a value if it is available. Otherwise, the nodes in the tree diagrams contain only the label. Nodes without children in the ASTs

Original code:  
\begin{verbatim}
01: foo( xy + 2, 42, 'q');
02: bar( 42, xyz + 2, foo, t);
\end{verbatim}

Modified code:  
\begin{verbatim}
01: foo( xy + 2, 42, 'q');
02: bar( 42, xyz + 2, foo, t);
\end{verbatim}

(a) Simple tree differencing example code.

(b) ASTs based on the source code.

Figure 3.2.: Tree differencing introductory example. The boxes show the edit actions that the tree differencing algorithms detect: \textcolor{yellow}{yellow} for updates, \textcolor{green}{green} for insertions, \textcolor{red}{red} for deletions, \textcolor{blue}{blue} for node movements. The dashed red arrows show the pairs in the mapping. Bold red arrows show pairs that lead to edit operations.
3.1 Tree Differencing on ASTs

are called leaves (e.g., \(xy_o, 2_m\)). Nodes that have children are called inner nodes (e.g., \(\text{Call}_o, \text{Args}_m\)).

ASTs also play an important role for ARES (see Chapter 4). ARES internally uses ASTs to represent the code changes it receives as inputs. To create patterns from these changes, ARES has to analyze the edit scripts between the different ASTs. To compute these edit scripts, ARES uses MTDIFF. In general, the heuristics in ARES assume that the computed edit scripts are as small as possible because only small edit scripts allow ARES to generate patterns that lead to accurate recommendations. This requirement connects ARES to the edit script size discussion that runs like a golden thread through this chapter.

Tree differencing algorithms like MTDIFF create an edit script, i.e., a list of edit operations that transforms the original \(AST_o\) into the modified \(AST_m\). In general, the goal of tree differencing algorithms is the reduction of the edit script cost. Each edit operation \(e_i\) has a cost \(\gamma(e_i)\). In an optimal edit script, the cost \(\sum_{i=1}^{k} \gamma(e_i)\) is minimal. Although this general definition allows different costs for each edit operation, many tools that use tree differencing algorithms (e.g., ARES, LASE [236]) just assume that all edit operations have the same cost (e.g., 1). In general, a shorter script produces a more accurate recommendation. The type of the operation has no impact on the recommendation results. This means that such tools do not prefer one edit operation over another. As the focus of this work lies on code recommendation systems, this and the following chapters also assume that each edit operation has a cost of 1. With identical costs for each edit operation, an edit script is optimal if the number of edit operations is minimal. A synonym for the cost of an optimal edit script is the edit distance \(\delta\) between two trees \(AST_o\) and \(AST_m\). Thus, an edit script is optimal if its cost is \(\delta\). The problem of finding this script is called edit distance problem [41]. Bille [41] defines \(\delta\) as:

\[
\delta(\text{AST}_o, \text{AST}_m) = \min\{\sum_{i=1}^{k} \gamma(e_i) | e_1..e_k \text{ transform } \text{AST}_o \text{ into } \text{AST}_m\}.
\]

There exist many approaches that address the problem of finding an optimal edit script. In general, tree differencing algorithms have to compare the nodes in both ASTs to identify it. Only a few approaches (e.g., brute force approaches, genetic algorithms) can create optimal edit scripts without comparing pairs of AST nodes. The result of the node comparison is the base for the edit script creation. During the
node comparison most state-of-the-art algorithms create a mapping \( M \). It consists of node pairs \((n_o, n_m)\) where \( n_o \) is part of \( AST_o \) and \( n_m \) is part of \( AST_m \). Each node must only be part of one pair, i.e., for two pairs \((n_{1o}, n_{1m}), (n_{2o}, n_{2m})\) \( \in M \) the condition \( n_{1o} = n_{2o} \text{ iff } n_{1m} = n_{2m} \) has to be true.

The following sections and chapters use the expression “\( n_{1o} \text{ is mapped to } n_{1m} \)”. This means that the pair \((n_{1o}, n_{1m})\) is part of the mapping \( M \). Most tree differencing algorithms that work on ASTs also enforce another restriction on all pairs \((n_o, n_m)\) that may be part of \( M \). For each pair \((n_o, n_m)\) in \( M \) the labels of \( n_o \) and \( n_m \) have to be identical. Our optimizations and MTDIFF also enforce this. In the domain of ASTs this is useful as there is no relationship between a Call node and a node for a numeric constant like 2.

As a mapping is an equivalent representation to an edit script (see below), state-of-the-art tree differencing algorithms do not search for an edit script with optimal cost, but for an optimal mapping. A mapping is optimal if its cost is minimal. Thus, the algorithms optimize the mapping instead of the edit script cost as this mapping is their source for creating the edit operations. For the definition of the cost \( \gamma \) of mapping \( M \), let \( U_o \) be the set of nodes from \( AST_o \) that do not appear in a pair in \( M \) and let \( U_m \) be the set of nodes from \( AST_m \) that do not appear in a pair in \( M \). Then the cost of a mapping \( M \) is:

\[
\gamma(M) = \sum_{(n_o, n_m) \in M} \gamma(n_o \to n_m) + \sum_{n_o \in U_o} \gamma(n_o \to \Lambda) + \sum_{n_m \in U_m} \gamma(\Lambda \to n_m).
\]

\( \gamma(n_o \to n_m) \) is the cost if both nodes have a partner in the mapping. \( \gamma(n_o \to \Lambda) \) is the cost if a node from the original tree has no partner in the mapping. \( \gamma(\Lambda \to n_m) \) is the cost if a node from the modified tree has no partner in the mapping. This cost definition makes it possible to specify the relationship between the mapping and the edit script. Given an edit script \( E \) with edit operations \( e_1 \ldots e_k \) from \( AST_o \) to \( AST_m \) then there exists a mapping \( M \) from \( AST_o \) to \( AST_m \) such that \( \gamma(M) \leq \gamma(E) \). Conversely, for any mapping \( M \) there exists an edit script \( E \) such that \( \gamma(E) = \gamma(M) \) [338]. This leads to an alternative definition of the edit distance:

\[
\delta(AST_o, AST_m) = \min\{\gamma(M)|M \text{ is a mapping from } AST_o \text{ to } AST_m\}.
\]

Thus, equivalent to finding edit scripts with minimal cost (i.e., the shortest edit script) is finding a mapping with minimal cost. Most tree
differencing algorithms rely on this equivalence and compute the edit script in two phases. Phase 1 computes a mapping whereas phase 2 creates edit operations based on this mapping. In this setting, Phase 1 remains NP-hard, but for phase 2 there exist fast optimal solutions [57, 137]. They compute the edit operations with a time complexity of $O(n^2)$, where $n$ is the number of nodes in the largest AST (i.e., either of AST$_o$ or AST$_m$). Thus, as there are fast solutions for phase 2, the focus of this thesis lies on the optimization of phase 1 to reduce the costs (i.e., size) of the created edit scripts.

The types of the edit operations that appear in an edit script depend on the algorithms used in phase 2. Phase 2 in this thesis uses the optimal algorithm by Chawathe et al. [57]. It differentiates between four edit operations, namely update, delete, insert, and move (see Fig. 3.2(b)). With these operations, the edit script transforms AST$_o$ into AST$_m$. The operation update changes the values of a node. For a pair (n$_o$, n$_m$) in the mapping in which the value $v_o$ of n$_o$ differs from the value $v_m$ of n$_m$, phase 2 creates an update($n_o$, $v_m$) operation. This update only changes the value of a node. The phase 2 algorithm does not support the change of labels with an update. This is not an issue as the tree differencing algorithms used in this thesis also do not use such pairs. In Fig. 3.2(b) the dashed arrows in red show the pairs in the mapping determined by phase 1 of a tree differencing algorithm (e.g., MTDIFF). In this example, an arrow connects the node $xy_o$ with the node $xyz_m$. As both have different values, phase 2 creates the edit operation update($xy_o$, $xyz_m$).

The delete operation removes nodes from AST$_o$ that are no longer present in AST$_m$. For each node of AST$_o$ that is not part of a pair in the mapping ($n_o \rightarrow \Lambda$), the algorithm of phase 2 adds a delete($n_o$) operation to the edit script. In the given example, this happens for node 'q'. The insert operation addresses nodes that are not present in AST$_m$. For each node of AST$_m$ that is not part of the mapping ($\Lambda \rightarrow n_m$), phase 2 creates an insert($n_m$, parent($n_m$), i) operation where $i$ specifies the $i^{th}$ child of the parent of $n_m$. Thus, for each of the nodes $bar_m$ and $t_m$ in the example, phase 2 creates an insert operation. The move operation changes the positions of AST nodes. This sometimes involves a change of their parents. Thus, there are two cases. In the first case, a node is moved to a new parent. For a pair (n$_o$, n$_m$) in the mapping, where the pair (parent(n$_o$), parent(n$_m$)) is not in the mapping, phase 2 adds a move($n_o$, parent($n_m$), i) operation where $i$ specifies the $i^{th}$ child of parent($n_m$). In the given example, the mapping contains the pair
update(xy_o, xyz)
delete('q'_o)
insert(bar_m, Call_m, 0)
insert(t_m, Args_m, 3)
move(42_o, Args_m, 0)
move(foo_o, Args_m, 2)

Figure 3.3.: Edit script for the tree differencing introductory example in Fig. 3.2.

(foo_o, foo_m). As there is no pair (no arrow) that connects their parents Call_o and Args_m, phase 2 puts a move operation into the edit script. The second case that leads to a move operation is the position change of a node within the children list of a parent node. Thus, in the second case the node keeps its parent. For a pair (n_o, n_m) in the mapping, where the pair (parent(n_o), parent(n_m)) also is in the mapping phase 2 adds a move operation if the nodes n_o and n_m have different child-indices. Thus, as the node 42_o has index 1 and the node 42_m has index 0 in the example and the parents are also a pair in the mapping, phase 2 adds another move to the edit script. Fig. 3.3 contains the complete edit script for this example.

With the help of the example in Fig. 3.2(b) it is also possible to explain the connection between the cost of the edit script E and the cost of the mapping M in more detail. As six edit operations (six colored boxes) are necessary to transform AST_o into AST_m and each edit operation has cost 1, the cost of the edit script is 6. This script is also minimal and thus the edit distance \( \delta(\text{AST}_o, \text{AST}_m) \) is also 6. The mapping cost \( \gamma(M) \) is based on three values. \( \sum \gamma(n_o \rightarrow n_m) \) computes the cost of the mappings. Unchanged nodes, i.e., nodes in the mapping that do not cause edit operations are without cost. Three mappings actually cause edit operations (the three bold red arrows in Fig. 3.2(b)) and thus \( \sum \gamma(n_o \rightarrow n_m) \) is 3. \( \sum \gamma(n_o \rightarrow \Lambda) \) computes the cost for all nodes from AST_o that are not in the mapping. If the cost for delete operations is 1, \( \sum \gamma(n_o \rightarrow \Lambda) \) is equal to the number of nodes from AST_o that are not in the mapping. It is 1 for the example due to the node 'q'_o. \( \sum \gamma(\Lambda \rightarrow n_m) \) computes the costs for all nodes from AST_m that are not in the mapping. The cost for \( \sum \gamma(\Lambda \rightarrow n_m) \) is 2 (assuming insert operations have cost 1) as there are two nodes (bar_m, t_m) in this set. Thus, the cost for the mapping is \( 3 + 1 + 2 = 6 \) which is equal to the cost of the edit script.
3.2 Related Work

In the last decades a lot of effort has been spent on finding algorithms that identify the shortest edit script. This happened because these algorithms have applications in several areas ranging from computer science to biology \([6, 339]\). The survey of Bille \([41]\) discusses tree differencing approaches up to 2005. It includes several algorithms described below. The survey also contains a detailed explanation of the edit distance and similar problems.

The first part of this section looks at tree differencing approaches that minimize the cost (i.e., the size) of edit scripts by minimizing the cost of the mapping. This section begins with a description of the long development of optimal tree differencing algorithms that do not detect movements. These algorithms always produce the shortest move-less edit script that is possible. The differences in the approaches lie in their time and memory complexities. Afterwards, this section discusses approaches that detect code movements. As this problem is NP-hard, the algorithms differ in the heuristics they use to minimize the size of edit scripts. In its last part, this section covers differencing algorithms for special purposes.

In 1979 Tai \([300]\) introduced one of the first optimal tree differencing algorithms. This tree differencing algorithm solves the edit distance problem with a space and time complexity of \(O(o^3m^3)\), where \(o\) is the number of nodes in the original tree and \(m\) is the number of nodes in the modified tree. Zhang and Shasha \([338]\) developed another optimal algorithm in 1989. It has a space complexity of \(O(o\cdot m)\) and a time complexity of \(O(o\cdot m\cdot min(l_o, d_o)\cdot min(l_m, d_m))\), where \(l\) is the number of leaves in the tree and \(d\) is the depth of the tree. Thus, in the worst case the algorithm has a time complexity of \(O(o^2 \cdot m^2)\). For balanced trees it is even faster and has a lower time complexity of \(O(o \cdot m \cdot \log o \cdot \log m)\). The approach by Zhang and Shasha uses a recursive dynamic programming approach that splits the larger edit distance problem between two trees \((T_o, T_m)\) into smaller subproblems. The subproblems are computations of the optimal (i.e., minimal) edit distances between the forests that appear due to the decomposition of the initial trees. As decomposition strategy, Zhang and Shasha always use the nodes on the path from the root to the leftmost leaf (or the rightmost leaf respectively). This leads to the low time complexity for balanced trees. As the decomposition strategy influences the time complexity of the algorithms, other strategies have been developed.
For example, ten years later, Klein [190] proposed an algorithm with a different decomposition strategy. His strategy uses the heavy path decomposition in $T_o$, which always selects the edge to the child with the most descendants. This gives an algorithm with a time complexity of $O(o^2 \cdot m \cdot o)$, where $m \geq o$. Thus, it is faster than the algorithm of Zhang and Shasha if there is a size difference between the trees (i.e., if the trees are not balanced). In 2005, Dulucq and Touzet [98] presented an algorithm that optimizes this decomposition strategy. Their algorithm has a time and space complexity of $O(o^2 \cdot m \cdot \log m)$. Demaine et al. [82] further improved their strategy. The algorithm by Demaine et al. not only uses the heavy path in $T_o$, but also examines the heavy path in $T_m$. It always selects the edge with the most descendants (in $T_o$ or $T_m$) for the next decomposition. This reduces the time complexity to $O(o^2 \cdot m \cdot (1 + \log o \cdot m))$ and keeps the space complexity at $O(o \cdot m)$. They also proved that this is worst case optimal for the edit distance problem.

The algorithm by Demaine et al. has the disadvantage that it is slower than the algorithm by Zhang and Shasha for many trees (e.g., balanced trees) due to the decomposition strategy. This disadvantage led to the creation of RTED by Pawlik and Augstein [262]. RTED computes the best strategy for a pair of trees and executes this strategy. Thus, it examines for each decomposition step, whether a decomposition based on the leftmost, the rightmost or the nodes on the heavy path in both $T_o$ and $T_m$ has the lowest computational cost. As a consequence, it has to examine six different possibilities for each decomposition step. With its strategy computation, RTED preserves the optimal worst case complexity of $O(o^2 \cdot m \cdot (1 + \log o \cdot m))$. At the same time, it computes the shortest edit distance for balanced trees in $O(o \cdot m \cdot \log o \cdot \log m)$. Thus, it combines the strength of the Zhang and Shasha algorithm with the strength of the algorithm by Demaine et al. and hence is faster than both on average. The main advantage of RTED is the identification of an optimal move-less edit script. However, for ARES this is insufficient. ARES can use move operations to increase the accuracy of the recommendations. RTED is part of the evaluation in Sec. 3.6.

Whereas the research on the shortest edit distance on trees without moves aims at optimal solutions, this is infeasible for edit scripts that contain moves. As a consequence, the research in this area focuses on the development of fast heuristics that approximate an optimal solution (i.e., a short edit script) as far as possible. One of the groundworks in this area is the work by Chawathe et al. [57]. Current tools [89, 112]
3.2 Related Work

still use their algorithm for phase 2 to create edit operations from a mapping. To create the required mapping (phase 1), Chawathe et al. developed an algorithm called **FastMatch**. **FastMatch** first serializes the trees AST\textsubscript{o} and AST\textsubscript{m} into lists and then computes the LCS on these lists. A node pair (n\textsubscript{o}, n\textsubscript{m}) can be part of the LCS if it fulfills the equality condition for leaves or inner nodes. For leaves, the equality condition of **FastMatch** is fulfilled if the labels are equal and the values are above a definable threshold \( f \) (\( 0 \leq f \leq 1 \)). **FastMatch** does not specify a value comparison function. In general, any string similarity measurement is possible. For an inner node pair (i\textsubscript{o}, i\textsubscript{m}), the labels have to be equal and the two inner nodes have to fulfill the equality condition

\[
sim\text{CHi} = \frac{|\text{common}(i\textsubscript{o}, i\textsubscript{m})|}{\max(|\text{leaves}(i\textsubscript{o})|, |\text{leaves}(i\textsubscript{m})|)} \geq t.
\]

The function \(	ext{common}\) counts all leaf pairs (l\textsubscript{o}, l\textsubscript{m}) in the mapping where the leaf l\textsubscript{o} is a descendant of i\textsubscript{o} and l\textsubscript{m} is a descendant of i\textsubscript{m}. For all leaves and inner nodes that are not part of the LCS, the algorithm looks for nodes that fulfill the equality condition with a linear search in the trees. Without the LCS optimization, this algorithm has a complexity of \( O(l^2 \cdot c + l \cdot i) \), where \( l \) is the maximal number of leaves in one of the trees (\( \max(l\textsubscript{o}, l\textsubscript{m}) \)) and \( i \) is the maximal number of inner nodes in one the trees (\( \max(i\textsubscript{o}, i\textsubscript{m}) \)). The value \( c \) is the average cost to compute the equality condition on leaves. This approach already has a lower complexity compared to the optimal algorithms. With the LCS optimization, **FastMatch** is even faster. It has a complexity of \( O((l \cdot e + e^2) \cdot c + 2 \cdot |\text{lbl}| \cdot l \cdot e) \) where \( e \) is the weighted edit distance between AST\textsubscript{o} and AST\textsubscript{m} (see Chawathe et al. [57]) and \( |\text{lbl}| \) is the number of unique inner node labels. Not only is this algorithm faster than RTED, it also supports move operations. However, it no longer guarantees optimal edit scripts. That means it is possible that there exist edit scripts that are shorter than the ones created with **FastMatch**.

Based on the work of Chawathe et al., Fluri et al. developed CD [119]. In contrast to other tree differencing algorithms, it is tailored to a specific task, namely the identification of change types in programs. Still, the approach of CD generally leads to shorter edit scripts compared to **FastMatch** for source code changes. Similar to **FastMatch**, CD uses an equality function for leaves and inner nodes. The differences to **FastMatch** lie in the examined node pairs and in the definition of the equality functions. CD first compares all leaves in AST\textsubscript{o} and AST\textsubscript{m} and looks for the leaf pairs that fulfill the equality condition and have the highest value similarity \( \text{sim}_{2g} \). In CD two leaves fulfill the equality
condition if they have the same label and the similarity measure of the values $\text{sim}_{2g}(v(l_o), v(l_m))$ is above 0.6. To compute the string similarity of the values, CD uses a similarity measurement $\text{sim}_{2g}$ that is based on n-grams [3] (see Sec. 3.4.1 on page 84 for an example). Its complexity is in $O(|v(l_o)| + |v(l_m)|)$ and thus it is faster than the also very common LVD with $O(|v(l_o)| \cdot |v(l_m)|)$. If there are several leaf pairs with the same similarity, CD selects the pair with the first node that appears during a post-order traversal of the trees (first of $\text{AST}_o$, second of $\text{AST}_m$). In these and other cases it is possible that CD selects leaf pairs that lead to an edit script that is longer than the optimal one. However, CD uses a very coarse-grained AST in which a complete statement (e.g., an assignment) forms a leaf node. As a consequence, such cases do not happen as often as on a fine-grained AST. In a fine-grained AST this is more common as a statement is an inner node and the leaves are less diverse.

After the matching of leaf nodes, CD forms pairs of inner nodes and adds the pairs to the mapping. To create the pairs, it traverses $\text{AST}_o$ in post-order and selects the first partner in $\text{AST}_m$ (also in post-order) that is not already mapped to another node and that fulfills the equality condition for inner nodes. For inner nodes, CD extends the equality conditions of $\text{FastMatch}$. To be added to the mapping, a pair of inner nodes has to meet three conditions: both nodes need the same label, the condition of $\text{sim}_{CHi} \geq t$ must be true, and also the similarity of the values $\text{sim}_{2g}(v(i_o), v(i_m))$ has to be larger than 0.6. If the similarity of the values is below the threshold of 0.6, the condition is still fulfilled if $\text{sim}_{CHi} \geq 0.8$. CD also uses two different values for the threshold $t$ depending on the size of $i_o$ and $i_m$. If the size is less or equal 4, $t$ is 0.4. It is 0.6 otherwise. As CD examines all possible leaf pairs and sorts them according to their similarities, the complexity of CD is above $\text{FastMatch}$. It is $O_{CD}(l^2 \cdot (c_L + 1 + \log l^2) + i \cdot (c_I + l))$. The advantage of CD is the smaller edit script for coarse-grained ASTs compared to $\text{FastMatch}$ although the minimization of the edit script was not the original purpose of the algorithm. The goal of Fluri et al. was an algorithm that identifies code changes for the identification of change types. The evaluation shows that the optimizations also improve the results of CD and reduce the sizes of the computed edit scripts (see Sec. 3.6).

The coarse-grained focus of CD is a drawback for applications that require more fine-grained ASTs. For example, compilers, many code analysis tools, and ARES use fine-grained ASTs. Using CD on such an AST leads to unnecessarily large edit scripts. To fill this gap, Falleri et
3.2 Related Work

al. [112] developed GumTree (GT). GT uses fine-grained ASTs. For example, it uses the AST of the Java Development Tools [166] for Java code. GT works in two phases, first a top-down, then a bottom-up phase. The top-down phase identifies identical subtrees in AST\textsubscript{o} and AST\textsubscript{m}. If there is more than one possible partner for such a tree, GT adds the possible pairs to a candidate list. Then it uses a function called dice to sort the candidate list. The function measures the ratio of common descendants. For each candidate pair (n\textsubscript{o}, n\textsubscript{m}), GT computes the dice value and orders the candidate list accordingly. Then it iterates over the candidates and if both n\textsubscript{o} and n\textsubscript{m} are not mapped to another node, GT adds the pair. Afterwards it executes the bottom-up phase. In this phase, GT first identifies unmatched inner nodes i\textsubscript{o}. For each such node i\textsubscript{o} it searches for inner nodes i\textsubscript{m} that have the same label and that are also not part of the mapping. If i\textsubscript{o} and i\textsubscript{m} have mapped children and there is no other node in AST\textsubscript{m} that fulfills these three conditions, GT adds the pair (i\textsubscript{o}, i\textsubscript{m}) to the mapping. If there are several such candidates, GT selects the pair with the highest dice value. After GT has added a pair of inner nodes to the mapping, it removes all descendants that are already in the mapping from the trees. If the reduced trees have fewer than maxSize = 100 nodes, GT executes RTED on the pair (i\textsubscript{o}, i\textsubscript{m}) to find additional pairs for the mapping. As this fixes the input size of the RTED algorithm, the complexity of GT is $O(n^2)$ where $n = \max(|AST\textsubscript{o}|, |AST\textsubscript{m}|)$. Although GT is a very fast algorithm and works on fine-grained AST, it still produces edit scripts that the optimizations in Sec. 3.3 can shrink. However, GT with optimizations still produces larger edit scripts than MTDIFF (see Sec. 3.6). In ARES, the GT edit scripts would cause patterns that are too general.

JSync by Nguyen et al. [251] is a clone management tool. To extract code changes, it also uses a tree differencing algorithm. This algorithm (called JSync below) works in three steps. First, JSync creates an initial mapping that contains pairs of leaf nodes. For that mapping JSync uses a text-based approach. It converts the ASTs into text and uses the LCS algorithm on the text lines to identify identical lines. Each leaf node in an unchanged line of AST\textsubscript{o} is paired with the corresponding leaf node in AST\textsubscript{m}. The identical lines also split the code into unchanged and changed fragments. This provides an assignment of changed fragments in AST\textsubscript{o} to changed fragments in AST\textsubscript{m}. Each changed fragment is surrounded by identical lines that are part of the LCS and thus have a corresponding partner in both ASTs. For each changed fragment in
AST$_o$, JSync creates a sequence of its leaves and also a sequence of leaves for the partner fragment of AST$_m$. Then JSync executes a variant of the LCS algorithm on both sequences. This identifies additional leaf pairs for the mapping. After this step, JSync traverses the ASTs bottom-up to find pairs of inner nodes. For each inner node $i_o$ that shares at least one child with another inner node $i_m$, JSync computes a similarity value (called Exas) that takes the nodes and their children into account. Then JSync adds the pair with the highest Exas value to the mapping. In the third step, JSync traverses the trees top-down. For each pair $(p_o, p_m)$ in the mapping, it searches for unmapped children in the subtrees of both nodes. As in the previous step, JSync uses the Exas similarity value to identify the most similar pair of inner nodes. It adds this most similar pair to the mapping. For leaf pairs, JSync computes the LVD on the values. It also adds the most similar leaf pairs to the mapping. The time complexity is $O(n^2)$ where $n = \max(|AST_o|, |AST_m|)$. The evaluation in Sec. 3.6.1 shows that JSync benefits from the optimizations in Sec. 3.3. However, MTDIFF produces even shorter edit scripts and thus is more useful for ARES.

It is possible to improve the results of merge tools with tree differencing algorithms. For example, JDime [17, 208, 207] is a structured merge tool for Java that uses a tree differencing algorithm to identify changes between Java files during a merge. Similar to CD, the purpose of the algorithm is not an optimal edit script. The goal of the tree differencing algorithm in JDime is the support of the merge process. This influenced the algorithm design. For example, JDime does not create pairs in the mapping that lead to update operations.

Internally, JDime combines different approaches. To reduce the execution time, it uses a variant of $\Theta_A$ (see Sec. 3.3.1) which identifies identical subtrees. The difference between $\Theta_A$ in Sec. 3.3.1 and the variant in JDime lies in the code parts $\Theta_A$ is applied to. This thesis proposes the use of $\Theta_A$ on the complete file. In JDime it is only applied to subtrees with identical root nodes. Generally, JDime traverses the ASTs top-down. As soon as the root nodes of two compared subtrees have different labels, the top-down traversal stops to increase the performance of JDime. To avoid an immediate abort if two root nodes do not match, JDime can also use a lookahead [207]. This is necessary to handle movements of root nodes to other subtrees. In the lookahead configuration, JDime searches for possible matches to a root node in relevant subtrees. As this has an impact on the performance, the depth of the lookahead is
3.2 Related Work

limited and depends on the label of the AST node. If the root nodes have the same label (or are an allowed label match), the top-down approach continues with an examination of their children. JDime uses two different algorithms for the children. One algorithm targets children for which their order within their parent does not matter. Reordering of such children has no effect on the semantics of the program (e.g., import statements in Java). The other algorithm handles children for which the order matters (e.g., statements in methods). If all the children fall into the unordered category, JDime uses a variant of Yang’s algorithm [330]. This variant recursively calls JDime’s top-down tree differencing algorithm on each children pair \((c_o, c_m)\) to get the mapping \(M_{(c_o, c_m)}\) for each combination. To identify the children pairs that have the largest combined mapping, JDime creates a matrix that holds the number of pairs in the mapping for each combination and executes the LCS algorithm on this matrix. Afterwards, it adds all pairs that are part of the LCS to the mapping. Additionally, for each added pair, JDime adds the node pairs from their mapping \(M_{(c_o, c_m)}\). For nodes that only have unordered children, JDime uses the Hungarian method [199] instead of the LCS algorithm. This has the advantage that it detects moves, but it is slower \(O(n^3)\) than the LCS algorithm \(O(n^2)\). Due to an early abort if two nodes have different labels, JDime can be faster than other algorithms. However, this abort can also increase the size of the edit scripts. The evaluation in Sec. 3.6 shows that the optimizations (see Sec. 3.3) have a large positive effect on the JDime results due to this early abort. They are able to reduce the edit script size for several thousand code changes.

Diff/TS by Hashimoto et al. [137, 138] is another tree differencing algorithm designed for ASTs. It works in three steps. The first step generates hashes for serialized representations [137] of certain tree nodes. For Java, Hashimoto et al. suggest classes and methods. If an hash of a subtree in AST\(_o\) also appears as hash of a subtree in AST\(_m\) and the hashes are unique in both ASTs, Diff/TS adds the subtree pair to the mapping. It also adds pairs for the corresponding children. This part is highly similar to \(\Theta_A\) (see Sec. 3.3.1), but less general as it only looks for hashes of certain tree nodes. After the first step, Diff/TS removes the unique identical subtrees from the original ASTs to reduce the problem size for the remaining steps. Afterwards, the first preprocessing step identifies unique nodes that appear in AST\(_o\) and AST\(_m\), but have different children. Diff/TS pairs these nodes together and adds them to the mapping. The second step uses a top-down approach. Diff/TS collapses all
remaining subtrees with a hash value (i.e., the classes and methods) into special nodes. The collapsed AST nodes bear their hashes as values. Then Diff/TS uses the algorithm by Zhang and Shasha to identify pairs for the mapping. If a pair consists of collapsed nodes, Diff/TS replaces the hash nodes with their original tree and executes the algorithm of Zhang and Shasha on the trees behind the collapsed nodes. This identifies additional pairs for the mapping in previously collapsed trees. To reduce the complexity of the algorithm, the replacement stops at a user-defined size threshold. The third step examines the matched node pairs and tries to reduce edit operations that affect moved subtrees, i.e., pairs in the mapping that lead to a move operation. On such pairs, Diff/TS executes the Zhang and Shasha algorithm. If the results of the Zhang and Shasha algorithm reduce the length of the edit script, Diff/TS replaces the original pairs with the pairs that lead to a shorter edit script.

As Diff/TS has a threshold on the Zhang and Shasha algorithm execution size, the time complexity of Diff/TS is $O(n^2)$. Although there is no public version of Diff/TS available, it is likely that the optimizations from Sec. 3.3 can improve the results of Diff/TS. As Diff/TS uses the Zhang and Shasha algorithm internally which does not support moves, Diff/TS probably does not detect all code movements. This leads to non-optimal edit scripts that the optimizations can improve.

The tool srcDiff by Decker [81] is another tree differencing approach that also uses heuristics to compute a list of edit operations. In contrast to other algorithms, the goal of srcDiff is not the minimization of the edit script, but to improve the change representation for developers. It uses a top-down approach that applies Myers algorithm [246] on the list of direct children of two matched nodes. A rule-based system decides whether two nodes can match during the top-down traversal. For example, there is a special rule that decides whether two if statements are similar enough to match. In contrast, single identifiers always match. The complexity is $O(n^2 \cdot n^k)$ where $k$ is the depth of an internal data structure (the srcML tree). According to the evaluation of srcDiff, developers found that the output of srcDiff is easier to read compared to the output of GT although its edit scripts are longer. Thus, as srcDiff produces longer edit scripts than GT, it also produces longer edit scripts than MTDIFF. For an automatic tool like ARES, a minimal edit script is more useful. An approach like srcDiff would lead to very general patterns and thus would reduce the accuracy of the recommendations.
Higo et al. [144] presented an extension for GT and the GT framework. This extension detects copied and inserted code in file changes and replaces all *insert* operations that add a code copy with a single *copy-and-paste* operation. This can reduce the edit script size in some cases. The introduction of this additional edit operation is orthogonal to the presented optimizations and MTDIFF. It is possible to include it in MTDIFF. This could even improve the results of ARES and C3.

There are several algorithms that target ASTs of XML documents. As the algorithms focus on XML ASTs, they can take the properties of XML ASTs into account. For example, it is often sufficient to compare only the labels and values of inner nodes to find suitable partners. In contrast to fine-grained ASTs, many XML leaves also have a unique value. This makes sophisticated algorithms to identify leaf pairs unnecessary. It is sufficient to search for leaves with identical or nearly identical values.

The diffX algorithm by Al-Ekram et al. [103] detects differences between XML documents. It traverses the XML ASTs top-down and looks for nodes with identical labels and values. If there exists such a partner $n_m$ for a node $n_o$, and there is no alternative, diffX adds the pair to the mapping. If there are several alternatives in AST$_m$, diffX executes its tree differencing approach on $n_o$ and each alternative and takes the alternative $a_m$ with the highest number of pairs. Based on $n_o$ and the selected $a_m$, diffX adds the matching nodes to the overall mapping. Afterwards, diffX continues the top-down traversal and examines the next node $n_o$ that is not already in a pair in the mapping. The time complexity of this algorithm is in $O(n^2)$. Like other XML focused algorithm, diffX expects that most nodes are unique in the compared ASTs. Thus, it does not handle ambiguities in the trees like MTDIFF does. Compared to MTDIFF, this can lead to longer edit scripts if there are many identical nodes in the trees.

XyDiff by Cobéna et al. [63] starts with a variant of optimization $\Theta_A$ (see Sec. 3.3.1) and also detects unique subtrees. Additionally, XyDiff uses the property of XML that there are nodes with unique ids. If such a node appears in AST$_o$ and AST$_m$, XyDiff adds this pair to the mapping. Afterwards, XyDiff examines the pairs in the mapping. If the parents of a pair in the mapping have the same type and both parents are not yet part of it, XyDiff also adds a pair of the parents to the mapping. In the next and final step, XyDiff examines the children of each pair in the mapping. If two children both have no partner in the mapping and they both have the same label, XyDiff also adds them to the mapping.
The time complexity of XyDiff is in $O(n \cdot \log(n))$. Like diffX, XyDiff also assumes that most of the nodes are unique. If this is not the case, it generates longer edit scripts as it does not have an ambiguity handling like MTDIFF.

For XML ASTs, in which the leaf nodes contain large text, Dohrn and Riehle presented HDDiff [87]. It combines XyDiff with text-based approaches to split large text leaves into smaller ones. This has the advantage that small text changes no longer affect the complete leaf text. The time complexity of HDDiff is in $O(n^2 + n \cdot \log^2(n))$.

A more recent tree differencing approach for XML is XCC by Rönnau and Borghoff [283]. This algorithm is part of an XML merging framework. It first uses the LCS algorithm to create leaf pairs for the mapping. A leaf pair can be part of the LCS if both their labels and their values are identical. Then XCC traverses the ASTs bottom-up to identify parents that share a leaf pair. XCC assumes that there is no large hierarchy in the AST as it focuses on the leaves. The time complexity is in $O(l \cdot e + i + e)$, where $l$ is the number of leaves, $i$ the number of inner nodes, and $e$ the size of the edit script. As XCC uses the LCS algorithm, it has the disadvantage that it cannot handle reordered nodes. As a consequence, it can benefit from the optimizations in Sec. 3.3 as they detect such reorderings. Similar to other algorithms for XML ASTs, it does not handle ambiguities. This can lead to longer edit scripts compared to MTDIFF if there are many identical nodes in the AST.

There are several other approaches that identify differences in models (e.g., UML diagrams). A description of all of them is out of scope of this thesis. However, Kolovos et al. [194] give an overview of current model differencing approaches. The approaches have in common that they rely on the fact that nearly all nodes and edges in a model are different. Thus, they mainly differ in the similarity measurements they use to compare nodes and edges. Due to the many identical nodes in ASTs, such similarity measurements are insufficient to compute short edit scripts. MTDIFF is better suited for trees with identical nodes.

There are also several approaches that identify semantic differences in models (e.g., Maoz et al. [225], Langer et al. [202]). However, as ARES and C3 both work on the syntactical level, such approaches are also out of scope of this thesis.

To take care of the specific requirements of the Verilog Hardware Description Language (VHDL), Duley et al. [97] proposed Vdiff. It traverses the VHDL ASTs top-down. On each AST level it applies the LCS
algorithm. A node pair can be part of the LCS if both have the same label. Afterwards, Vdiff adds all pairs that are part of the LCS to the mapping. Then Vdiff examines all nodes not already in the mapping. For each such node, Vdiff creates a textual representation that contains the complete subtree. Then it uses the UMLDiff text similarity [323] to find the pairs with the highest similarity. Similar to JDime, Vdiff also ignores the order of children for which the order does not influence the semantics. The time complexity is in $O(n^2)$. As discussed before, the LCS algorithm prevents an accurate detection of reordered nodes. MTdIFF can identify such reorderings.

JDiff [18] by Apiwattanapong et al. identifies differences in object-oriented programs. It adds pairs of classes that have the same qualified name to the mapping. Then it examines the methods of each class pair. If there are methods in both classes that have the same signature, it also adds them to the mapping. If there are still methods that are not part of the mapping, JDiff looks for methods with the same name. JDiff also adds them to the mapping (if their parent classes are also mapped). For method pairs in the mapping, JDiff computes the control flow graphs (CFGs). Then it identifies similar parts in their CFGs. For that purpose, JDiff iteratively computes the hammocks (single-entry, single-exit subgraphs) of the CFGs and collapses each hammock into a single node until each CFG of a method pair consists of a single hammock. Afterwards, JDiff expands the hammocks and thus replaces them with their subgraphs. The subgraphs may contain other hammocks of a lower level. On the expanded graph, JDiff performs a depth-first traversal, identifies nodes with identical labels, and adds them to the mapping. To compare two hammocks $h_o$ and $h_m$, JDiff recursively uses the complete algorithm on the hammock pair. If the number of resulting pairs in the mapping for two hammocks is above a certain threshold, JDiff adds the pair $(h_o, h_m)$ to the mapping. If the number is below the threshold and there is no suitable node for $h_o$ in the current subgraph, JDiff looks for possible partners to $h_o$ in the children of $h_m$. To speed up the differencing on the CFGs, JDiff contains a maximal lookahead that decides how deep the hammock expansion runs. Without this lookahead restriction, the complexity of JDiff is in $O(n^3)$. JDiff can benefit from the optimizations in Sec. 3.3 as it does not compare large code parts with each other to identify node pairs. For example, if methods have different signatures or names it does not look for suitable node
3 Tree Differencing

pairs shared by them. In contrast, MTDIFF detects movements of leaves and inner nodes across the complete AST.

Horwitz [155] presented an algorithm that detects differences in a program representation graph and thus in the semantics of two code parts. However, the approach is limited to a subset of the C language. According to Horwitz, this algorithm has an unrealistic upper bound of $O(n^2)$.

Dex [271] by Raghavan et al. is a graph differencing tool for abstract semantic graphs. This graph consists of an AST with type information and other semantic properties. Dex combines a top-down and bottom-up approach and executes both traversals several times. The top-down approach looks for identical nodes (same labels and values) that also have identical children. Dex adds such nodes to the mapping. If there is more than one match, Dex uses the information from the previous bottom-up traversal to find the best match. In the bottom-up traversal, Dex computes a cost matrix and adds the pairs with minimal costs. Thus, the goal in the bottom-up phase is the minimization of the cost for each added pair. Then Dex computes the edit operations for the current mapping and executes the complete algorithm again until there are no further changes to the edit operations. The complete process has a time complexity of $O(n^4)$ which is slower than other algorithms described above. All the heuristics are also tailored to the abstract semantic graph and are less suited for fine-grained ASTs with many identical nodes.

3.3 General Optimizations

Current tree differencing approaches for ASTs have problems to detecting move operations accurately. This led to the development of the six optimizations $\Theta_{A-F}$ as well as to the development of MTDIFF (see Sec. 3.4). This section describes all six optimizations $\Theta_{A-F}$. They compensate the limitations of current approaches and are compatible with standard tree differencing algorithms that generate a mapping. They can reduce the size of non-optimal edit scripts and thus optimize their costs. The optimizations are also the base for MTDIFF. To reduce the edit script size they replace other edit operations with moves. This makes them also an essential building block for ARES as it requires such short edit scripts to generate accurate recommendations. The following subsections introduce the different optimizations. To illustrate that
the optimizations are applicable to a variety of algorithms, this chapter evaluates them with five tree differencing algorithms (GT, RTED, CD, JSync, and JDime). As these five algorithms use different AST granularities and different assumptions, the actual examples differ from algorithm to algorithm. Except for Sec. 3.3.1 that uses the edit script from RTED, this section only uses edit scripts from GT to show the effects of the optimizations. Appendix A discusses examples for the other tree differencing algorithms.

This selection includes GT and CD because both are regularly used in research tools to compute differences on ASTs. JSync and JDime are examples of tree differencing algorithms used for special purposes. JSync is part of a clone management framework whereas JDime is part of a structural merge tool. All four algorithms also support moves and their source code is publicly available. RTED is a special case in this set of selected tree differencing approaches, because its intended use is the creation of an optimal edit script without moves. This is different to the non-optimal algorithms MTDIFF, GT, CD, JSync, and JDime. Nevertheless, it is possible to use the mapping from RTED as input for any phase 2 algorithm, even if this algorithm supports moves. This section and the evaluation both use phase 2 of the GT framework [135] in combination with RTED. In such a setting, it is also possible to combine the mapping from RTED with $\Theta_{A-F}$ and to examine the results. This reveals also the reason for the inclusion of RTED in the evaluation. It allows the evaluation of the effect that the optimizations have on an optimal algorithm.

The following sections also use notations like "GT$_{A-F}$". This is an abbreviation for GT in combination with the optimizations $\Theta_{A-F}$. After the presentation of $\Theta_A$ to $\Theta_F$, the end of this section discusses the effects that the optimization order has on the results.

### 3.3.1 Identical Subtree Optimization $\Theta_A$

$\Theta_A$ identifies identical subtrees in the compared ASTs and adds the node pairs for the identical subtrees and for their children to the mapping. This is crucial for a short edit script as it handles unchanged code parts. If a tree differencing algorithm maps nodes from unchanged to nodes from changed code parts, the edit script contains unnecessary operations. As $\Theta_A$ is an intrinsic part of GT, this optimization has no effect on the GT results. On the RTED mappings and thus the resulting edit scripts, $\Theta_A$ has a positive effect and shortens the edit scripts.
3 Tree Differencing

Original code:

```
01: moveA();
02: stmtA();
```

Modified code:

```
01: stmtA();
02: moveA();
```

(a) RTED requires 2 edit operations \(= 2 \) boxes.

(b) RTED\(_A\) requires 1 edit operation.

(c) ASTs of the \(\Theta_A\) example.

Figure 3.4.: RTED example for \(\Theta_A\). The boxes show the edit operations that the tree differencing algorithms detect: Yellow for updates, green for insertions, red for deletions, blue for node movements. The red arrows show the changed pairs in the mapping after the optimization. Gray arrows show the original pairs that were changed.

As RTED does not consider moved code, it is unable to handle moved and otherwise unchanged code with a short edit script. For example, instead of a single move operation, RTED requires two update operations to express the move of call moveA in Fig. 3.4. Other algorithms also create unnecessarily large edit scripts if they do not detect the movement of unchanged code fragments (see Appendix A).

A solution for such detection problems is the identification of identical subtrees in \(\text{AST}_o\) and \(\text{AST}_m\). The Identical Subtree Optimization \(\Theta_A\) handles this. It is an intrinsic part of GT and also appears in variants in other tree differencing algorithms (e.g., Diff/TS [137, 138], XyDiff [63]). The evaluation in Sec. 3.6 shows that RTED, CD, JSync, and JDime can benefit from this optimization.

Fig. 3.5 holds the pseudo code for this optimization. It starts with the computation of fingerprints for each subtree (line 2). Each fingerprint is a string representation of the respective subtree. The fingerprint has the property that it is only identical for identical subtrees. A fingerprint is unique in \(\text{AST}_o\) if there is no second identical subtree in \(\text{AST}_o\). The same applies for \(\text{AST}_m\). After the fingerprint computation, \(\Theta_A\) traverses
3.3 General Optimizations

```plaintext
01: function IDENTICALSUBTREEOPT(AST_o,AST_m)
02:  List_o ← createFingerprints(AST_o); List_m ← createFingerprints(AST_m)
03:  M ← ∅; list ← add(AST_o, list)
04:  while list ≠ ∅ do
05:  o ← head(list)
06:  F ← fingerprint(o)
07:  if unique(F, List_o, List_m) then
08:     m ← getPartner(o, AST_m)
09:     M ← M ∪ mapNodesInSubtrees(o, m)
10:  else
11:     for c ∈ children(o) do
12:         if depth(c) > 0 then
13:             list ← add(c, list)
14:     return M

Figure 3.5.: Identical Subtree Optimization, Θ_A.
```

AST_o top-down and looks for identical fingerprints (lines 4–9). If the fingerprint of a subtree is unique in AST_o and also unique in AST_m, Θ_A adds the corresponding subtree pairs. It also adds pairs for their children to the mapping (line 9). If there is no unique match, the top-down traversal continues. For that purpose, Θ_A adds all the children of the current node o to the work list if they are not leaves (depth > 0) (lines 11–13). As all algorithms handle leaves explicitly, it is unnecessary to handle them in the preprocessing step. This avoids duplicate work by Θ_A and the algorithms. Additionally, in fine-grained ASTs, unique leaves are a rare occurrence and thus the handling of leaves would increase the execution time unnecessarily.

In the RTED example, the fingerprints of the Call_αo and Call_βm subtrees are identical. As both subtrees only occur once in the original and once in the modified code, the fingerprints are also unique. Thus, Θ_A pairs the roots for the moveA calls (i.e., Call_αo and Call_βm) together and adds this pair to the mapping. It also adds the pairs for their children (i.e., (moveA_o, moveA_m)) to the mapping. Similarly, Θ_A handles the stmtA calls. These changes decrease the size of the edit script by one for the RTED example. For the other algorithms, Θ_A improves the edit script in a similar manner (see Appendix A).

There is also a positive side effect of Θ_A. Many code changes affect only small parts of a file and all other parts remain unchanged. This gives Θ_A the potential to add pairs for all nodes in large subtrees to the mapping. If the following tree differencing algorithm only examines the subset of nodes that is not part of the mapping, the execution time of the algorithm with Θ_A is smaller than without it.
3.3.2 LCS Optimization $\Theta_B$

$\Theta_B$ addresses unmapped inner and unmapped leaf nodes after the execution of the tree differencing algorithms. It adds all suitable pairs that are part of an LCS to the mapping. This is beneficial because the examined tree differencing algorithms (except RTED) use heuristics and thresholds that determine the node pairs in the mapping. As such heuristics and thresholds are not flexible enough for all situations, the heuristics miss some valid node pairs (i.e., pairs with the same label). $\Theta_B$ can add these pairs to lower the edit script size. For example, GT uses its dice similarity in combination with RTED to match nodes. This combination does not identify the movement of the $\text{condB}$ node in line 1 of Fig. 3.6. The other four analyzed algorithms also do not detect all movements to other subtrees (see Appendix A). This even happens if there is a clear, unambiguous partner to a node in the original AST (e.g., $\text{condB}_m$ in Fig. 3.6).

The LCS Optimization $\Theta_B$ fills these gaps in the mapping that the heuristics and thresholds in the tree differencing algorithms cause. It first serializes subtrees that form a pair in the mapping and then applies the LCS algorithm on these serialized trees. If a pair of nodes is part

Original code:  Modified code:

(a) GT$_A$ requires 3 edit operations.

(b) GT$_{A-B}$ requires 2 edit operations.

(c) ASTs of the $\Theta_B$ example.

Figure 3.6.: GT example for $\Theta_B$. Same legend as in Fig. 3.4.
function LCSOPT(root_o, root_m, M)
03: U_o ← unmappedNodesTopDown(root_o, M);
04: U_m ← unmappedNodesTopDown(root_m, M)
05: done ← ∅
06: for u_o ∈ U_o do
07: p_o ← parent(u_o)
08: if p_o ∈ done then
09: continue
10: done ← done ∪ {p_o}
11: p_m ← mappedNode(u_o, M)
12: if p_m ≠ null then
13: l_o ← nodesInPostOrder(p_o)
14: l_m ← nodesInPostOrder(p_m)
15: result ← LCS(l_o, l_m, U_o, U_m, M)
16: for (m_o, m_m) ∈ result do
17: if m_o ∈ U_o ∧ m_m ∈ U_m then
18: M ← M ∪ {(m_o, m_m)}
19: U_o ← U_o \ {m_o}; U_m ← U_m \ {m_m}
20: return M

Figure 3.7.: LCS Optimization, \( \Theta_B \).

of the LCS and both nodes are not already mapped to other nodes, \( \Theta_B \) adds the newly discovered pair to the mapping.

\( \Theta_B \) starts with a top-down search for nodes without partner (lines 2–3 in Fig. 3.7). This reduces the overhead of the optimization as it only applies the LCS algorithm if there are unmapped nodes in the tree. Using the optimization for nodes that are already part of the mapping would not change the results. Thus, if \( \Theta_B \) finds such a node (e.g., \texttt{condB} in the GT example of Fig. 3.6), it examines its parent. If the parent is part of the mapping, e.g., the argument list \texttt{Args}_o of the assert statement, the parent is a starting point for \( \Theta_B \). \( \Theta_B \) serializes the subtree of the parent (e.g., \texttt{Args}_o) and also the subtree of its partner \( p_m \) (e.g., \texttt{Args}_m) and computes the LCS (lines 12–14 in Fig. 3.7) on the serialized lists. To be part of the LCS, a node pair \((n_o, n_m)\) has to fulfill two conditions. First, both nodes require the same label. Second, either both nodes form a pair that is already part of the mapping or both are not mapped to another node. The second condition ensures that a node is at most part of one pair in the mapping. \( \Theta_B \) also avoids unnecessary moves across already mapped nodes because such pairs are also in the LCS. If \( \Theta_B \) finds an unmapped pair like \((\texttt{condB}_o, \texttt{condB}_m)\), it adds the pair to the mapping (lines 15–18). In the example in Fig. 3.6, this reduces the size of the edit script by one. To avoid another evaluation of nodes that are no longer unmapped, \( \Theta_B \) removes the new node pairs from \( U_o \) and \( U_m \) (line 18). Other nodes in Fig. 3.6 do not fulfill the required
conditions of $\Theta_B$. In a similar fashion, $\Theta_B$ also reduces the size of the edit scripts for the examples of the other algorithms (see Appendix A).

### 3.3.3 Unmapped Leaves Optimization $\Theta_C$

$\Theta_C$ identifies pairs of unmapped leaves for the mapping. This is necessary as often the heuristics in the algorithms or the LCS algorithm in $\Theta_B$ cannot detect arbitrary moves inside the ASTs. For example, in line 3 of Fig. 3.8 the node $1_o$ is part of the LCS whereas the nodes $2_o$ and $c_o$ are not. Due to the structure of the LCS, only one of the leaves ($1_o$, $c_o$, $2_o$) can be part of it. GT without optimizations also does not detect these move operations because it internally uses RTED to identify leaf pairs. As RTED does not consider movements, it cannot detect the reordering of the leaves. However, using RTED is faster than an optimal solution that takes reorderings into account. Other algorithms make a similar trade-off between performance and detection of moved leaves. They also use approaches (e.g., the LCS algorithm) that ignore reorderings to improve their performance (see Appendix A).

$\Theta_C$ can detect reordered leaves and thus is able to reduce the edit script sizes. It detects leaf movements by examining leaves that have no partner in the mapping. The limitation to leaves reduces the overhead of $\Theta_C$. The use of only unmapped leaves also lowers the risk that the optimization increases the edit script size. It is possible that $\Theta_C$ increases the edit script size as it only identifies a local optimum and not a global one. $\Theta_C$ starts with an iteration over all leaves without partner (see line 2 in Fig. 3.9) and searches for parents of leaves that have a partner in the mapping (lines 4–6 in Fig. 3.9). In the GT example in Fig. 3.8, both $2_o$ and $c_o$ are not part of the mapping but their parent ($Args_o$) is. If $\Theta_C$ finds such a leaf $u_o$ for which the pair ($p_o$, $p_m$) is in the mapping, the optimization uses the function $partnerChild$ to identify the best possible partner for $u_o$ in the children of $p_m$. For that purpose, $partnerChild$ checks eight conditions in the following order:

1. Select a node $x_m$ among the children of $mappedP_m$ that has the same label, the same value, and the same position as $u_o$. The node $x_m$ must not be in the mapping.

2. Select a node $x_m$ that is among the children of $mappedP_m$ and that has the same label, the same value, and the same position as $u_o$. The pair ($x_o$, $x_m$) is in the mapping and the value of $x_o$ and $x_m$ is different.
3.3 General Optimizations

Original code:
02: callC(1, 2, c);

Modified code:
callC(c, 2, 1);

(a) GT\textsubscript{A−B} requires 4 edit operations.
02: callC(1, 2, c);
callC(c, 2, 1);

(b) GT\textsubscript{A−C} requires 2 edit operations.

(c) ASTs of the $\Theta\textsubscript{C}$ example.

Figure 3.8.: GT example for $\Theta\textsubscript{C}$. Same legend as in Fig. 3.4.

3. Select a node $x\textsubscript{m}$ that is among the children of $\text{mappedP}\textsubscript{m}$ and has both the same label and the same position as $u\textsubscript{o}$. The node $x\textsubscript{m}$ must not be in the mapping.

4. Select a node $x\textsubscript{m}$ that is among the children of $\text{mappedP}\textsubscript{m}$ and has both the same label and the same position as $u\textsubscript{o}$. The pair $(x, x\textsubscript{m})$ is in the mapping and the value of $x$ and $x\textsubscript{m}$ is different.

5. Select the node $x\textsubscript{m}$ if $\text{mappedP}\textsubscript{m}$ has only one child and $x\textsubscript{m}$ has both the same label and the same value as $u\textsubscript{o}$. The node $x\textsubscript{m}$ must not be in the mapping.

6. Select the node $x\textsubscript{m}$ if $\text{mappedP}\textsubscript{m}$ has only one child, $x\textsubscript{m}$ has both the same label and the same value as $u\textsubscript{o}$. The pair $(x\textsubscript{o}, x\textsubscript{m})$ is in the mapping and the value of $x\textsubscript{o}$ and $x\textsubscript{m}$ is different.

7. Select a node $x\textsubscript{m}$ that is among the children of $\text{mappedP}\textsubscript{m}$ and that has both, the same label and the same value as $u\textsubscript{o}$. The node $x\textsubscript{m}$ must not be in the mapping.

8. Select a node $x\textsubscript{m}$ that is among the children of $\text{mappedP}\textsubscript{m}$ and has both, the same label and the same value as $u\textsubscript{o}$. The pair $(x\textsubscript{o}, x\textsubscript{m})$ is in the mapping and the value of $x\textsubscript{o}$ and $x\textsubscript{m}$ is different.
3 Tree Differencing

function \textsc{UnmappedLeavesOpt}(\textit{root}_o, \textit{root}_m, \textit{M})
02: \textit{U}_o \leftarrow \text{unmappedLeaves}(\textit{root}_o, \textit{M});
03: for \textit{u}_o \in \textit{U}_o do
04: \textit{p}_o \leftarrow \text{parent}(\textit{u}_o);
05: \textit{mappedP}_m \leftarrow \text{mappedNode}(\textit{p}_o, \textit{M})
06: if \textit{mappedP}_m \neq \text{null} then
07: \textit{x}_m \leftarrow \text{partnerChild}(\textit{u}_o, \textit{mappedP}_m)
08: if \textit{x}_m \neq \text{null} then
09: \textit{x}_o \leftarrow \text{mappedNode}(\textit{x}_m, \textit{M})
10: if \textit{x}_o \neq \text{null} then
11: \textit{M} \leftarrow \text{M} \setminus \{(\textit{x}_o, \textit{x}_m)\}
12: \textit{M} \leftarrow \text{M} \cup \{(\textit{u}_o, \textit{x}_m)\}
13: \textit{U}_m \leftarrow \text{unmappedLeaves}(\textit{root}_m, \textit{M});
14: for \textit{u}_m \in \textit{U}_m do
15: \textit{p}_m \leftarrow \text{parent}(\textit{u}_m);
16: \textit{mappedP}_o \leftarrow \text{mappedNode}(\textit{p}_m, \textit{M})
17: if \textit{mappedP}_o \neq \text{null} then
18: \textit{x}_o \leftarrow \text{partnerChild}(\textit{u}_m, \textit{mappedP}_o)
19: if \textit{x}_o \neq \text{null} then
20: \textit{x}_m \leftarrow \text{mappedNode}(\textit{x}_o, \textit{M})
21: if \textit{x}_m \neq \text{null} then
22: \textit{M} \leftarrow \text{M} \setminus \{(\textit{x}_o, \textit{x}_m)\}
23: \textit{M} \leftarrow \text{M} \cup \{(\textit{x}_o, \textit{u}_m)\}
24: return \textit{M}

Figure 3.9.: Unmapped Leaves Optimization, \(\Theta_C\).

For the node \textit{c}_o in the example, \textit{c}_m fulfills the seventh condition. Both have the same label and the same value. The node \textit{c}_m is also not mapped to another node. The same condition applies to \textit{2}_o and \textit{2}_m. For the GT example, this optimization shortens the edit script by 2. \(\Theta_C\) reduces the size of the edit scripts from other algorithms analogously.

The actual search for unmapped leaves happens two times in \(\Theta_C\). It occurs once for \(\text{AST}_o\) and once for \(\text{AST}_m\). The reason is that if \textit{p}_o has no unmapped leaf but \textit{p}_m does, \(\Theta_C\) has the possibility to optimize the mapping if there are leaf pairs with a better match.

### 3.3.4 Inner Node Optimization \(\Theta_D\)

\(\Theta_D\) identifies better suited partner for inner nodes that are already mapped to another node. This is useful as \(\Theta_B\) and \(\Theta_C\) influence the pairs in the mapping. In some cases, this makes the additional evaluation of the inner nodes worthwhile. It is possible, that different pairs of inner nodes reduce the edit script size by avoiding move operations. The heuristics of GT, JSync and CD all follow a similar principle regarding inner nodes. They pair inner nodes (\textit{i}_o, \textit{i}_m) together if their children are mapped to each other. This means that the heuristics pair \textit{i}_o with \textit{i}_m if there are children in \textit{i}_o paired together with children in \textit{i}_m. This
approach avoids unnecessary *moves* as paired children of paired parents do not generate *move* operations (as long as there is no reordering within the children). In general, the heuristics of the algorithms pair the inner nodes together that share the most children pairs. However, there are some cases in which the heuristics create inner node pairs that lead to unnecessary edit operations. Additionally, there is the influence of $\Theta_D$.

$\Theta_D$ takes care of such inner node pairs. As $\Theta_D$ mainly improves the results of $\Theta_B$ and $\Theta_C$, a larger example is necessary. Fig. 3.10 holds the simplified AST of the GT example of Fig. A.2. In this example, a developer removed an *if* statement and moved the call $q$ into the body of the *while* loop. GT alone takes 12 edit operations for this example, GT with $\Theta_{A-C}$ requires 9 edit operations. In this edit script of GT$_{A-C}$, the *move* operations of $q$ and $h$ are unnecessary. The cause of this is the node pair ($Block_{\gamma_0}$, $Block_{\beta_m}$) in the mapping. GT creates this pair because the *dice* similarity metric assigns the highest similarity value to this pairing. This happens although the *dice* similarity takes the children of the inner nodes into account and the pair ($Block_{\beta_0}$, $Block_{\beta_m}$) has more common children. Thus, the heuristics in GT pair two inner nodes together that share some children ($q()$ in the example), but they do not identify the optimal pair that shares more ($p()$ and $h()$ in the GT examples). The heuristics in the other analyzed algorithms share this problem (see Appendix A). As the algorithms do not identify a global optimum, they sometimes generate inner node pairs that share fewer children with each other than they would share with their optimal partners.
Although RTED identifies such optimal partners, \( \Theta_B \) and \( \Theta_C \) influence the children pairs in the RTED results. Thus, the original assumptions of RTED about shared children are no longer valid after their changes to the mapping. This makes it useful to combine \( \Theta_{B-D} \) even with RTED.

\( \Theta_D \) reduces the edit script size by evaluating the shared children of inner nodes. It examines all pairs of inner nodes \((i_o, i_m)\) in the mapping in their post-order appearance (see Fig. 3.11). To optimize the tree bottom-up, \( \Theta_D \) uses the post-order traversal of AST\(_o\). This is necessary as changes by \( \Theta_D \) on the lower levels can influence the results of higher levels of the tree. For each inner node \( i_o \), \( \Theta_D \) looks for a node \( \text{max}_m \) that shares the most children with it in the mapping. If such a node exists and it is not already mapped to \( i_o \) (i.e., it is equal to node \( i_m \)), \( \text{max}_m \) is a potential partner for \( i_o \). Line 9 contains the usual check that ensures that only nodes with identical labels can form a pair in the mapping. To avoid unnecessary moves, \( \Theta_D \) only changes node pairs if more than half of the children of \( i_o \) are mapped to children of \( \text{max}_m \).

In the example in Fig. 3.10, GT pairs the block of the if statement (Block\(_{\gamma o}\)) in the original code with the block of the while statement (Block\(_{\beta m}\)) . Both blocks share the subtree of the call \( q \), but the calls \( p \) and \( h \) belong to a different block. \( \Theta_D \) detects that the block of the while statement in the original code shares more children with the block of the while statement in the modified code. Thus, \( \Theta_D \) changes the mapping to the one shown by the red arrows in Fig. 3.10. This reduces the edit script by one. The edit script changes by \( \Theta_D \) for the other tree differencing algorithms are similar (see Appendix A).
3.3 General Optimizations

Original code:  
03: printE("42!" + "Answer:");        | Modified code:  
| printE("Answer:" + "42!" );

(a) GT$_{A-D}$ requires 2 edit operations.

03: printE("42!" + "Answer:");        | printE("Answer:" + "42!" );
(b) GT$_{A-E}$ requires 1 edit operation.

(c) ASTs for the $\Theta_E$ example.

Figure 3.12.: GT example for $\Theta_E$. Same legend as in Fig. 3.4.

3.3.5 Leaf Move Optimization $\Theta_E$

$\Theta_E$ evaluates the leaf nodes in the mapping to identify missed moves or unnecessary moves. Fig. 3.12 gives an example for GT. Instead of a single move, GT requires two update operations. Again, the combination of RTED with the dice heuristics inside GT causes this problem. As the other optimizations $\Theta_B$ and $\Theta_C$ address unmapped nodes and $\Theta_D$ only addresses inner nodes, the previous optimizations do not cover this issue. As discussed in Sec. 3.3.3 for $\Theta_C$, the other algorithms also use heuristics that often do not detect suitable partners for moved leaves (e.g., the "Answer:" nodes in the GT example). This is also the problem that $\Theta_E$ addresses. However, as $\Theta_E$ overrides decisions made by the original algorithms (i.e., it changes pairs in the mapping) its heuristics are more conservative. In contrast to $\Theta_B$, the optimization $\Theta_E$ has no effects on the CD results due to the way CD handles leaves of its coarse-grained AST.

In contrast to $\Theta_C$, $\Theta_E$ examines leaves $l_o$ that have a partner $l_m$ in the mapping. To avoid an increase of the edit script size by the optimization and to reduce the execution time, $\Theta_E$ only addresses pairs in which both nodes have different values (line 2 in Fig. 3.13). For the GT
example in Fig. 3.12 this method returns the pairs ("42!"o, "Answer:"m) and ("Answer:"o, "42!"m). \(\Theta_E\) looks for better fitting partners for the nodes in these pairs. A better fitting partners must have the same label, the same value and either be unique in the children list of their parent or be at the same position in the children list of the examined nodes. \(\Theta_E\) first examines the pairs inside the mapping to identify a partner \(mp_m\) for the parent of \(l_o\) (line 7). If this partner has a child \(c\) that has the same label and the same value as \(l_o\), \(c\) is a possible partner for the leaf \(l_o\). Using such a \(c\) either transforms an update into a move or, in the best case, removes the edit operations for \(l_o\) and \(c\). To avoid unnecessary moves, \(\Theta_E\) only changes the mapping if there is only one child \(c\) that has the same label and value as \(l_m\) (i.e., it is unique), or their position in the children list of their parents is equal (lines 11–12). In the GT example of Fig. 3.12, the parent of the node "42!"o is +o. This node is mapped to +m. In the children list of +m, only "42!"m has the same label and value. Thus, the unique condition in line 12 is fulfilled, and \(\Theta_E\) adds the new pair ("42!"o, "42!"m) to the mapping (lines 12–13). As "42!"m also has a partner in the mapping ("Answer:"o), \(\Theta_E\) also adds the previously paired nodes "Answer:"o and "Answer:"m to the mapping (lines 15–16). As the value of these nodes is identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair. If the leaves are not identical, \(\Theta_E\) can examine the next leaf pair.

If the nodes differ in AST\(_o\) and AST\(_m\) due to deletes and inserts, it is possible that there is no partner for "42!"o that fulfills the conditions of \(\Theta_E\) (same label, same value, unique or at the same position). Still, it is possible that there exists a partner for "Answer:"m. To find such a possible partner for "Answer:"m, \(\Theta_E\) uses the second loop in lines 24–35. This loop is identical to the first one, except that it looks for a partner to \(l_m\).

It is further possible that two leaves in a pair not only have a different value, but also two parents that are not a pair in the mapping. For such cases, \(\Theta_E\) executes the algorithm in Fig. 3.13 a second time. In the second run, \(\Theta_E\) uses the parents of the paired leaves (parent\((l_m)\) and parent\((l_o)\)) to optimize the mapping. This changes lines 7 and 22 in Fig. 3.13. This second run also has the potential to reduce the edit script size.
3.3 General Optimizations

3.3.6 Cross Move Optimization $\Theta_F$

$\Theta_F$ identifies identical nodes that switched places according to the mapping and optimizes the corresponding pairs. It is a novel optimization not published in the previous work of MTDIFF [89]. It is essential to create accurate recommendation patterns with ARES. $\Theta_F$ shortens the edit script in several cases. Fig. 3.14 gives an example for GT. Due to the deleted call, the dice similarity measurement in GT pairs two $F$ nodes together which creates two unnecessary move operations. These two operations switch the $F$ nodes in the ASTs in Fig. 3.14. The heuristics of the other algorithms also cause move operations that switch subtrees (like the $F$ nodes in the GT example) although the subtrees...
3 Tree Differencing

Original code:

04: `deletef []
05: F f = new F(1 , 2);

Modified code:

04: `f = new F (1);
05: F f = new F (1);

(a) GT

(b) GT

(c) ASTs for line 5 of the \( \Theta_F \) example.

Figure 3.14.: GT example for \( \Theta_F \). Same legend as in Fig. 3.4.

actually remain in place and are unaltered (see Appendix A). For RTED this only occurs, if \( \Theta_{A-E} \) were executed beforehand.

\( \Theta_F \) changes the mapping and eliminates such unnecessary moves. For that purpose, it examines all pairs \( (n_o, n_m) \) in the mapping and looks for better partners. It also assumes that the parents of the nodes in a pair do not change. To keep the parents fixed for each node pair, \( \Theta_F \) traverses the tree top-down in breadth-first search order (line 2 in Fig. 3.15) and thus never changes the parents of a previously optimized node pair. For each pair \( (n_o, n_m) \), \( \Theta_F \) looks for alternative pairs that can reduce the edit script size. The reduction is possible if the parents of the nodes \( (p_o, p_m) \) have partners \( (mp_o, mp_m) \) in the mapping and \( (p_o, p_m) \) is not identical to \( (mp_o, mp_m) \) (lines 7–11). It is also more likely that \( \Theta_F \) cannot improve the mapping if the values of the parents and partners are not identical. For example, it happens more rarely that developers move nodes of an addition to nodes of a Boolean comparison. To be conservative, \( \Theta_F \) ignores such pairs (lines 10–11). The \( F \) nodes and their parents in the GT example fulfill all these conditions.
3.3 General Optimizations

Possible partners for \( n_o \) and \( n_m \) that shorten the edit scripts are either children of \( m_p o \) (lines 12–22) or \( m_p m \) (lines 23–33). Pairing \( n_o \) with a node \( c_m \) can shorten the edit script if the parent of \( c_m \) is \( m_p m \) that is mapped to the parent of \( n_o \) (line 17). In the GT example the parent of \( n_o \) is the declaration node \( \text{Decl}_o \) in line 5 of the original code in the example. The node \( m_p m \) is the declaration node \( \text{Decl}_m \) of the modified code. As \( (\text{Decl}_o, \text{Decl}_m) \) is a pair in the mapping, the optimization \( \Theta_F \) can switch the pairs \((F_{\alpha_o}, F_{\beta_m}), (F_{\beta_o}, F_{\alpha_m})\) to \((F_{\alpha_o}, F_{\alpha_m}), (F_{\beta_o}, F_{\beta_m})\) in the mapping. In the GT example, this avoids the two extra *moves*.

### 3.3.7 Order of \( \Theta_{A-F} \)

There exist different valid execution orders for the optimizations. All can lead to shorter edit scripts. However, the optimizations were designed for the order \((\Theta_A; \text{Phase 1 - Mapping Creation}; \Theta_B; \Theta_C; \Theta_D; \Theta_F)\).
$\Theta_E; \Theta_F$, where \textit{Mapping Creation} is the part of the original algorithm that produces the mapping. There are several advantages of this order. $\Theta_A$ is designed as a preprocessing step that identifies unchanged code parts. This prevents the \textit{Mapping Creation} heuristics of the algorithms to pair nodes of unchanged code with nodes in changed code. Thus, $\Theta_A$ has a larger effect if it runs before the \textit{Mapping Creation} of the tree differencing algorithm. Additionally, the \textit{Mapping Creation} step can use the results of $\Theta_A$ to reduce the workload, as the \textit{Mapping Creation} can limit itself to the examination of the remaining unmapped nodes.

$\Theta_B$ inserts additional pairs in the mapping and hence reduces the number of unmapped nodes. The following optimizations can use these extra pairs as a foundation and produce better results with them. Some optimizations also work better with fewer unmapped nodes. For example, $\Theta_C$ examines each unmapped leaf isolated from most other nodes and does not take all other unmapped leaves into account. As a consequence, the more unmapped leaves exist, the higher the risk that $\Theta_C$ creates unnecessary edit operations. Thus, it is beneficial to run $\Theta_B$ before $\Theta_C$. As $\Theta_C$ changes unmapped leaf pairs, it influences the children of inner nodes that are part of the mapping and thus influences $\Theta_D$. $\Theta_D$ optimizes the pairs of inner nodes in the mapping. It can take the results of $\Theta_B$ and $\Theta_C$ into account to further reduce the size of the edit script. There are many cases in which the execution of $\Theta_D$ after the previous optimizations leads to a shorter edit script compared to the execution of $\Theta_D$ directly after the \textit{Mapping Creation}.

The change of inner node pairs in the mapping by $\Theta_D$ offers additional optimization possibilities. Due to the change of inner nodes, changing the pairs of their children can also reduce the edit script size. This is the task of $\Theta_E$. It re-examines the leaves and adjusts the leaf pairs based on the pairs of inner nodes in the mapping. It is also useful after the \textit{Mapping Creation} and $\Theta_B$. However, at any earlier position it cannot take the changes of $\Theta_D$ into account.

The results from the evaluation in Sec. 3.6 show that the optimization $\Theta_F$ either changes the pairs generated by the \textit{Mapping Creation} of the base algorithm or it changes the pairs that $\Theta_B$ created. Thus, the execution of $\Theta_F$ at any position after \textit{Mapping Creation} or $\Theta_B$ is useful. However, there is also no disadvantage in executing $\Theta_F$ last.

Except of $\Theta_A$, all other optimizations influence each other. Thus, it is possible to run them until a fix point is reached in which the optimizations no longer change the mapping. Currently, this is not used for
3.4 Move-Optimized Tree Differencing

GT, RTED, JSync and JDime have trouble detecting moves in changed subtrees. Fig. 3.16 gives an example for this problem. All four tree differencing algorithms generate unnecessary edit operations for this example. Due to the different ASTs, JDime in combination with \( \Theta_{A-F} \) requires 20 edit operations for this example. On their respective ASTs, GT, RTED, and JSync require 11 edit operations for the example. All algorithms and the optimizations are unable to detect the move of the declaration of integer \( a \) from line 3 to line 1 in the example. The additional code lines in Fig. 3.16 are necessary to trigger this problem. Due to the move detection problem, the edit script contains three delete and three insert operations instead of a single move in Fig. 3.16.

In contrast to the other three examined algorithms, CD is more flexible in detecting movements in changed subtrees. However, using CD for ARES or \( C3 \) is impossible because both tools require a fine-grained AST. CD has been designed for a coarse-grained AST and its heuristics do not apply to a fine-grained version. Therefore, this thesis presents MTDIFF, a novel tree differencing algorithm that uses the structure of CD as basis.
3 Tree Differencing

Figure 3.17.: Tree mapping in MTDIFF.

Similar to CD, the first part of MTDIFF only creates leaf pairs for the mapping (line 2 in Fig. 3.17). Then, also like CD, MTDIFF identifies pairs of inner nodes for the mapping. The heuristics that determine the pairs of inner nodes use the previously determined leaf pairs. In contrast to CD and all other algorithms, MTDIFF requires $\Theta_A$. Without $\Theta_A$, the leaf mapping heuristics in MTDIFF are too time consuming. The weights and thresholds in MTDIFF are also optimized for the use of MTDIFF in combination with $\Theta_B - \Gamma$. The optimizations also propagate information from the leaf pairs and the inner nodes back to each other. As for other algorithms (e.g., GT, RTED, CD, JSync, and JDime), this results in shorter edit scripts. Using the two-step CD/MTDIFF approach on a fine-grained AST solves the original problem in Fig. 3.16.

For a general use of this approach on a fine-grained AST, there are still two challenges to overcome:

1. In the coarse-grained CD-ASTs, leaves are complete statements. As there is a large diversity of statements, there are only a few leaves that have the same value (i.e., the same code) in the CD-AST. In fine-grained ASTs, many nodes have the same label and value. For example, all the int nodes in Fig. 3.18 have the same label and the same value. The following sections use this example to explain the properties of MTDIFF. To minimize the edit script for code with such identical leaves it is necessary to spend additional effort optimizing all pairs with identical nodes. Without such an optimization, the edit script would contain unnecessary moves. Sec. 3.4.1 shows how MTDIFF handles leaves with identical values.

2. In the CD-AST, most conditions are simply values attached to a node. For example, the condA is the value of the if node. In a fine-grained AST, such conditions are an attached subtree. Thus, heuristics that expect that conditions are values of inner nodes instead of subtrees do not work as intended and lead to larger edit scripts.
3.4 Move-Optimized Tree Differencing

Original code:
```
01: foo(xy + 2);
02: int a = 0;
03: int b = 1;
```

Modified code:
```
   foo((xyz + 2), t);
   int b = 1;
   int a = 0;
```

Figure 3.18: Simple example to show the MTDIFF heuristics. Same legend as in Fig. 3.8.
3 Tree Differencing

The following sections illustrate how MTDIFF handles these two challenges. Sec. 3.4.1 describes the leaf mapping step and the changes that are necessary for **Challenge 1**. With the leaf mapping results, Sec. 3.4.2 computes the inner node pairs for the mapping. This section also discusses how the inner node mapping step handles **Challenge 2** in MTDIFF. Sec. 3.4.3 describes how the configuration values for MTDIFF were determined. The configuration values are weights, thresholds and constants that MTDIFF uses in its heuristics.

### 3.4.1 Leaf Mapping

The leaf mapping step of MTDIFF (see Fig. 3.19) is responsible for identifying leaf pairs for the mapping. The second step of MTDIFF only builds pairs of inner nodes. As input, the leaf mapping requires AST\(_o\) (with root\(_o\)) and AST\(_M\) (with root\(_m\)). To take \(\Theta_A\) into account, LeafMapping removes all leaves that are already present in the mapping \(M\) (lines 2–3).

The actual algorithm starts with the computation of the leaf similarity for all unmapped leaves (lines 5–9). The similarity function \(\text{sim}_L\) is similar to the one in CD:

\[
\text{sim}_L(l_o, l_m) = \begin{cases} 
0 & \text{lbl}(l_o) \neq \text{lbl}(l_m) \\
\text{sim}_{2g}(v(l_o), v(l_m)) & \text{otherwise}
\end{cases}
\]

As before, node pairs with different labels (\(\text{lbl}\)) are not allowed and thus their similarity is zero. For all other leaves, MTDIFF uses the bigram similarity to compare the values. The bigram similarity showed better results than the LVD in the CD evaluation [119]. Adamson and Boreham [3] introduced the bigram similarity \(\text{sim}_{2g}\) to identify similar words in documents. Bigrams are two letter combinations that appear if one uses a sliding window on a string. For example, the bigrams of the word “node” are “no”, “od”, and “de”. For two string \(s_o, s_m\), Adamson and Boreham define the similarity measure as:

\[
\text{sim}_{2g}(s_o, s_m) = \frac{2|\text{bigrams}(s_o) \cap \text{bigrams}(s_m)|}{|\text{bigrams}(s_o) \cup \text{bigrams}(s_m)|}.
\]

Compared to the LVD, the bigram similarity can handle character movements better as it does not rely on the LCS algorithm. After computing the similarity for each pair of unmapped leaves, MTDIFF compares the value with threshold \(L\) (line 8). Only if the computed similarity is above this threshold, MTDIFF adds the node pair to the list of leaf pairs \(P\). Without a threshold, the mapping contains too many pairs that
move nodes across unrelated subtrees. This also influences the mapping of inner nodes. Abandoning the threshold would lead to unnecessary edit operations in the script. Additionally, adding arbitrary pairs of leaf nodes with different values is not beneficial for the edit script size. Such pairs simply replace the delete and insert operations with move and update operations.

After the similarity computation, LeafMapping sorts the list of leaf pairs $P$ first by decreasing similarity, second by the post-order node visiting sequence of AST$_o$, and third by the post-order node visiting sequence of AST$_m$ (line 10). This enforces a deterministic order on the pairs and thus MTDIFF is able to create deterministic edit scripts that are identical in each run. Executing these steps on the example in Fig. 3.18 leads to the following sorted list of leaf pairs $P$ (with a similarity above zero): $(\text{foo}_o, \text{foo}_m, 1), (2_o, 2_m, 1), (\text{int}_\alpha_o, \text{int}_\alpha_m, 1), (\text{int}_\beta_o, \text{int}_\beta_m, 1), (\text{a}_o, \text{a}_m, 1), (\text{1}_o, \text{1}_m, 1), (\text{int}_\beta_o, \text{int}_\alpha_m, 1), (\text{b}_o, \text{b}_m, 1), (\text{0}_o, \text{0}_m, 1), (\text{xyz}_o, \text{xyz}_m, 0.66)$. In the list, there are 7 pairs without ambiguities. However, the four int nodes are ambiguous as they all have the same similarity value.

In a coarse-grained AST, such ambiguities are very rare, and CD does not handle them explicitly. Instead, CD just selects the first pair in the

```plaintext
01: function LEAFMAPPING(root$_o$, root$_m$, M, resolveAmbiguity)
02:     L$_o$ ← unmappedLeaves(root$_o$, M)
03:     L$_m$ ← unmappedLeaves(root$_m$, M)
04:     U ← L$_o$ ∪ L$_m$; P ← ∅
05:     for $l_o$ ∈ L$_o$ do
06:         for $l_m$ ∈ L$_m$ do
07:             similarity ← sim$_L$(l$_o$, l$_m$)
08:             if similarity ≥ threshold$_L$ then
09:                 P ← P ∪ {(l$_o$, l$_m$, similarity)}
10:             P ← orderBySimilarityAndVisitOrder(P);
11:         C ← P; A ← ∅
12:     if resolveAmbiguity then
13:         C ← unambiguousPairs(P)
14:         A ← ambiguousPairs(P)
15:     for ($l_o$, $l_m$, s) ∈ C do
16:         if $l_o$ ∈ U ∧ $l_m$ ∈ U then
17:             M ← M ∪ {$l_o$, $l_m$}
18:             U ← U \ {$l_o$}; U ← U \ {$l_m$}
19:         A ← handleAmbiguities(A, M)
20:     for ($l_o$, $l_m$, s) ∈ A do
21:         if $l_o$ ∈ U ∧ $l_m$ ∈ U then
22:             M ← M ∪ {$l_o$, $l_m$}
23:             U ← U \ {$l_o$}; U ← U \ {$l_m$}
24:         return M
```

Figure 3.19.: Computation of leaf pairs in MTDIFF.
sorted list. In Fig. 3.16, this would be \((\text{int}_{\alpha_0}, \text{int}_{\alpha_m}, 1)\). LeafMapping displays the same behavior if resolveAmbiguity is set to false.

However, MTDIFF needs to handle ambiguities, as they are common in a fine-grained AST. Without ambiguity treatment, MTDIFF would add the pair \((\text{int}_{\alpha_0}, \text{int}_{\alpha_m}, 1)\) to the mapping for the example. This in turn would cause the inner node mapping step to add the barely similar pairs \((\text{Decl}_{\alpha_0}, \text{Decl}_{\alpha_m}, 0.5)\) and \((\text{Decl}_{\beta_0}, \text{Decl}_{\beta_m}, 0.5)\) to the mapping. The edit script based on these mappings would cause two move operations for the identifiers \(a\) and \(b\) instead of one for the assignment.

To handle the ambiguities, MTDIFF separates the unambiguous from the ambiguous pairs and puts the four \text{int} pairs into the Ambiguous list \(A\) and the remaining seven pairs into the \text{Clear match} list \(C\). Then MTDIFF adds all nodes with a clear partner to the mapping (lines 15–18). The check in line 16 makes sure that each node appears only once in the mapping. This is necessary because several pairs with the same node can exist in the list. However, all such pairs have different similarities in \(C\). To select the best pairs first, \(C\) is sorted by similarity and by the post-order position of the nodes in the trees \(\text{AST}_o\) and \(\text{AST}_m\). In the current example, MTDIFF adds all seven pairs in \(C\) to the mapping.

Afterwards, MTDIFF handles the ambiguous pairs in groups. In the example, all four \text{int} leaves belong to the same group as they have the same label and value. Table 3.20(a) shows their similarities. MTDIFF uses the additional information that the parents and their corresponding trees provide to treat such ambiguous leaves. The parents of the pair \((\text{int}_{\alpha_0}, \text{int}_{\alpha_m})\) are the declaration nodes \((\text{Decl}_{\alpha_0}, \text{Decl}_{\alpha_m})\). \text{Decl}_{\alpha_0} contains the identifier \(a\) as child, whereas \text{Decl}_{\alpha_m} contains \(b\). These different identifiers are the key for the mapping optimization. If the parents provide no additional information, MTDIFF continues the ambiguity treatment with the grandparents. If necessary, the ambiguity treatment goes up to the roots of the ASTs.

\[
\begin{array}{c|cc}
\text{int}_{\alpha_0} & \text{int}_{\beta_0} \\
\hline
\text{int}_{\alpha_m} & 1 & 1 \\
\text{int}_{\beta_m} & 1 & 1 \\
\end{array}
\quad
\begin{array}{c|cc}
\text{Decl}_{\alpha_0} & \text{Decl}_{\beta_0} \\
\hline
\text{Decl}_{\alpha_m} & \frac{2}{3} & 2 \\
\text{Decl}_{\beta_m} & 2 & \frac{5}{3} \\
\end{array}
\]

(a) Ambiguous leaf similarity \quad (b) Matrix \(S\) after handleAmbiguities.

Figure 3.20.: Ambiguous leaf similarity matrix for Fig 3.18.
3.4 Move-Optimized Tree Differencing

On the similarity values in Fig. 3.20, MTDIFF applies the algorithm in Fig. 3.21. First, MTDIFF initializes the matrix with the leaf pairs and their similarity values (lines 2–7). The information comes from the leaf pairs of the group $g$. Then the function iteratively adjusts the similarity values with the loop in lines 9–28. The adjustment process stops if there are no longer any changes made to the matrix values. This happens if the similarity computation reaches the root nodes (line 18). The adjustment also ends if each row and column has a maximum value to speed up the computation. Due to the assignment problem [199], it is possible that a single maximum in each row and column is insufficient to find the mapping for the shortest edit script. However, to reduce the execution time, MTDIFF aborts the similarity computation at this point.

To update the matrix $S$, MTDIFF examines each node pair $(n_o, n_m)$ (lines 11–28) and computes a new similarity value based on their parents. For the similarity, MTDIFF executes $treeMapping$ without ambiguity treatment on the trees of the parents $p_o$ and $p_m$ to get a mapping.
Then MTDIFF adds up the similarities for each node pair in \( M' \) (lines 23–24) and normalizes it to the interval \([0;1] \) (line 25). The last part of each iteration is an update of the similarity matrix (line 27). MTDIFF replaces the nodes with their parents in the matrix and also adds the new similarity value \( \text{sim}' \) to the original one \( \text{sim} \). MTDIFF adds the similarity values instead of replacing them because this can create several distinct similarity values within a row. This provides additional information for the cases in which a single maximum in each row is insufficient. Table 3.20(b) holds the similarity matrix after the first iteration for the example in Fig. 3.18. As this matrix has a maximum in each row and column, the adjustment process terminates with this updated version.

The updated similarity matrix solves the problem with the ambiguous \( \text{int} \) pairs in Fig. 3.18. However, there are cases in which this is insufficient, e.g., if the same statement appears several times in the same block. \( \Theta_A \) also does not help in these circumstances as it only adds pairs to the mapping if the subtrees are unique. The parent information also provides no insight as all subtrees up to the root provide the same similarity values for each ambiguous pair. To handle these cases, \( \text{handleAmbiguities} \) (see Fig. 3.22) also takes the leaf positions into account. MTDIFF uses a second matrix \( D \) (\text{Distance}) that holds similarity values based on the node positions.

Initially, \( \text{handleAmbiguities} \) splits all ambiguous nodes in list \( A \) into groups (line 2 in Fig. 3.22). Each group contains all leaves that have the same similarity values to each other (e.g., the four \( \text{int} \) nodes). After the similarity matrix computation (line 4), \( \text{handleAmbiguities} \) computes \( D \). To compute the distance between two leaves, \( \text{handleAmbiguities} \) uses their index in the post-order sorted list \( L_o' \) or \( L_m' \) (lines 5–14 in Fig. 3.22). Each distance is the absolute difference between the indices in the lists. This approach differs from previous work on MTDIFF [89]. Originally, MTDIFF used the index difference based on the post-order position of the complete AST. In contrast to the new approach, this leads to less accurate recommendations in ARES due to unnecessary large edit scripts. In general, the new approach is more robust against changes to \( \text{AST}_o \) and \( \text{AST}_m \).

MTDIFF has to consider both, the distance matrix \( D \) and the similarity matrix \( S \) to handle ambiguous leaves. To find the most suitable pairs for the mapping, MTDIFF combines both. For the combination, \( \text{handleAmbiguities} \) first normalizes both values to the interval \([0;1000] \). Then it
3.4 Move-Optimized Tree Differencing

```
01: function HANDLEAMBIGUITIES(A, M)
02: G ← createGroups(A, M); A' ← ∅
03: for g ∈ G do
04: S ← computeSimilarityMatrix(g, A, M)
05: L_o ← getOriginalLeaves(g)
06: L_m ← getModifiedLeaves(g)
07: L_o' ← sortByPostOrder(L_o)
08: L_m' ← sortByPostOrder(L_m)
09: D ← createMatrix(|L_o|, |L_m|)
10: for i ∈ 0..|L_o|−1 do
11: for j ∈ 0..|L_m|−1 do
12: l_o ← getNode(L_o, i)
13: l_m ← getNode(L_m, j)
14: D_{ij} ← abs(getIndex(l_o, L_o') − getIndex(l_m, L_m'))
15: S ← normalize(S)
16: D ← normalize(D)
17: S' ← createMatrix(|L_o|, |L_m|)
18: for i ∈ 0..|L_o|−1 do
19: for j ∈ 0..|L_m|−1 do
20: S'_{ij} ← w_{L0} · (1000 − S_{ij}) + w_{L1} · D_{ij}
21: H ← identifyBestPairsWithHungarianMethod(S')
22: A' ← A' ∪ H
23: return A'
```

Figure 3.22.: Handling of ambiguous leaves in MTDIFF.

combines both values into a matrix $S'$ with the weights $w_{L0}$ and $w_{L1}$ (line 20 in Fig. 3.22). It is necessary to use $1 − value$ for the similarity matrix entries because in the similarity matrix $S$, the highest value is preferable whereas in the distance matrix $D$ the lowest is preferable.

After the combination of the values, MTDIFF has a similarity matrix $S'$ for each group $g$ of ambiguous leaves. To finally identify the best leaf pairs, MTDIFF uses the Hungarian method [199] (line 21). This leads to shorter edit scripts compared to a naive approach. For example, a simple row by row selection of the maximum values in the matrix can lead to larger edit scripts. In Fig. 3.16 this would be sufficient, but for the similarity matrix in Fig. 3.23 it is not. Selecting only the maximum similarities from the top row to the last row in Fig. 3.23 leads to a combined similarity of 24. The Hungarian method, that solves this Assignment Problem, leads to a combined similarity of 30. As the Hungarian method originally was not designed for float values, rounding errors can lead to problems inside the algorithm. To avoid them, MTDIFF uses the interval $[0;1000]$ for $S'$. Due to the long execution time of the Hungarian method, MTDIFF uses the Hungarian method only for matrices with less than 1000 rows and less than 1000 columns. If the matrix is larger, MTDIFF just selects the maximum row by row. In summary, with the above similarity computation and the Hungarian method, MTDIFF is
3 Tree Differencing

![Table](image)

Figure 3.23.: Example for the assignment problem.

able to handle **Challenge 1**, i.e., the occurrence of ambiguous leaves in fine-grained ASTs.

### 3.4.2 Inner Node Mapping

Like CD, the `fastInnerNodeMapping` in Fig. 3.24 (called during the leaf ambiguity resolution) uses the first pair of inner nodes above a threshold. Both CD and MTDIFF use this selection scheme to reduce the execution time. However, as \( \Theta_A \) reduces the number of unmapped nodes, examining all unmapped inner nodes in MTDIFF is not as expensive as for CD. This allows MTDIFF to compare all pairs of unmapped inner nodes (see Fig. 3.25). The benefit of the comparison of all inner nodes is a reduction of the edit script size. As MTDIFF calls `innerNodeMapping` only once, this benefit outweighs the time overhead.

Due to **Challenge 2**, the actual computation of the inner node similarity differs from CD. In contrast to CD, inner nodes like `if` or `while` do not contain their condition as value. In fine-grained ASTs, the condition

```plaintext
01: function fastInnerNodeMapping(root_o, root_m, M)
02:     I_o ← unmappedInnerNodes(root_o, M)
03:     I_o ← sortByPostVisitOrder(I_o, root_o)
04:     I_m ← unmappedInnerNodes(root_m, M)
05:     I_m ← sortByPostVisitOrder(I_m, root_o)
06:     U ← I_o ∪ I_m; P ← ∅
07:     for i_o ∈ I_o do
08:         for i_m ∈ I_m do
09:             vsim ← sim_{IV}(i_o, i_m)
10:             csim ← sim_{IC}(i_o, i_m)
11:             similarity ← (w_{I0} · vsim + w_{I1} · csim) / 2
12:             if similarity ≥ threshold_I then
13:                 M ← M ∪ \{(i_o, i_m)\}
14:                 U ← U \ {i_o}; U ← U \ {i_m}
15:                 break
16:     return M
```

Figure 3.24.: Fast computation of inner node pairs in MTDIFF for the ambiguity treatment.
3.4 Move-Optimized Tree Differencing

is an extra subtree. Therefore, the inner node similarity of CD is not applicable. Instead, MTDIFF uses two similarity measurements, \(sim_{IV}\) for the values of inner nodes and \(sim_{IC}\) for the children of inner nodes. To combine the two similarities, MTDIFF uses the two weights \(w_{I0}\) (for \(sim_{IV}\)) and \(w_{I1}\) (for \(sim_{IC}\)) that lie in the interval \([0;1]\). MTDIFF does not require that \(w_{I0} + w_{I1}\) is below 1 as this gives more flexibility to the configuration. The inner node value similarity is defined as:

\[
sim_{IV}(i_o, i_m) =
\begin{cases}
0.0 & \text{lbl}(i_o) \neq \text{lbl}(i_m), \\
1.0 & v(i_o) = v(i_m) \land \text{discrete}(i_o), \\
\sim_d(v(i_o)) & v(i_o) \neq v(i_m) \land \text{discrete}(i_o), \\
\sim_{2g}(v(i_o), v(i_m)) & \text{otherwise}
\end{cases}
\]

The structure of \(sim_{IV}\) is similar to \(sim_{L}\). The difference lies in the second and third condition. There are several inner nodes (e.g., infix expressions like additions or multiplications) that have a small number of discrete values. For example, an assignment node can have the values =, +=, -=, etc. If two inner nodes have the same label, and the label falls into this discrete category, \(sim_{IV}\) uses the similarity 1.0 if the values are equal. If the values are not equal, it uses the constant value \(\sim_d\). Sec. 3.4.3 explains the computation of this constant. The actual set of discrete values depends on the AST. For the GT-AST, the \textit{discrete} label set contains the labels for \texttt{PREFIX_EXPRESSION}, \texttt{POSTFIX_EXPRESSION}, \texttt{INFIX_EXPRESSION}, and \texttt{ASSIGNMENT}. For all remaining inner nodes, MTDIFF compares the values with the

```plaintext
01: function INNERNODEMAPPING(root_o, root_m, M)
02:     I_o ← unmappedInnerNodes(root_o, M)
03:     I_m ← unmappedInnerNodes(root_m, M)
04:     U ← I_o \cup I_m; P ← ∅
05:     for i_o ∈ I_o do
06:         for i_m ∈ I_m do
07:             vsim ← sim_{IV}(i_o, i_m)
08:             csim ← sim_{IC}(i_o, i_m)
09:             similarity ← (w_{I0} \cdot vsim + w_{I1} \cdot csim) / 2
10:         if similarity ≥ threshold then
11:             P ← P \cup \{(i_o, i_m, similarity)\}
12:     P ← orderBySimilarityAndVisitOrder(P);
13:     for (i_o, i_m, s) ∈ P do
14:         if i_o ∈ U \land i_m ∈ U then
15:             M ← M \cup \{(i_o, i_m)\}
16:         U ← U \setminus \{i_o\}; U ← U \setminus \{i_m\}
17: return M
```

Figure 3.25.: Computation of inner node pairs in MTDIFF.
bigram similarity measurement $\text{sim}_{2g}$. This has the advantage that MTDIFF also works on the coarse-grained CD-AST. For discrete values, the bigram similarity is not an option as the similarity of "+" and "*" is just 0. Thus, the pair $(+, m)$ would not be part of the mapping. This would create a delete and insert operation instead of a single update and increase the edit script size.

The second similarity $\text{sim}_{IC}$ for inner nodes uses the children of inner nodes as base. Without this similarity, $\text{innerNodeMapping}$ would select the first node with equal label and value that appear during the post-order traversal of the trees. In the example of Fig. 3.16 this would create the pairs $(\text{Decl}_o, \text{Decl}_m)$ and $(=o, =m)$ for the mapping. As described above, this causes two additional move operations for the identifiers $a$ and $b$.

Fluri et al. [119] evaluated different options for this similarity. In their evaluation, the best results were obtained with an inner node similarity measurement that takes only the leaves into account. As a consequence, MTDIFF also relies on the leaves to compute $\text{sim}_{IC}$. Whereas CD uses the value to enforce the importance of the condition for if and other inner nodes, MTDIFF has to use a different approach to support fine-grained ASTs. For the actual similarity $\text{sumC}$ (part of $\text{sim}_{IC}$), MTDIFF uses:

$$\text{sumC}(i_o, i_m) = \sum_{q_o \in \text{directch}(i_o)} \frac{|\text{comp}(q_o, i_m)|}{|\text{leaves}(q_o)|} + \sum_{q_m \in \text{directch}(i_m)} \frac{|\text{comp}(q_m, i_o)|}{|\text{leaves}(q_m)|}.$$  

For each direct child $q_o$ of $i_o$ and for each direct child $q_m$ of $i_m$, $\text{sumC}$ computes the similarity of the direct children with the paired inner nodes. This similarity uses the leaf mapping from the previous step as input. Based on this mapping, $\text{comp}$ counts the number of leaves in $q_o$ that are part of the mapping and for which the mapping partner is a leaf in $i_m$. The $\text{comp}$ function uses the complete $i_m$ for this search. As the position of $q_o$ in $i_o$ is fixed, it would be possible to search only for partners in the corresponding subtree $q_m$ that has the same position in $i_m$. However, it is possible that a leaf in $q_o$ is moved to a different subtree in $i_m$. In this case, using only a subtree $q_m$ would decrease the similarity of $i_o$ and $i_m$. Using the complete tree of $i_o$ to search for partner has the advantage that the trees $i_o$ and $i_m$ remain similar even if the order of their leaves was changed. Thus, $\text{comp}$ uses the complete tree with $i_m$ as root for the search. To normalize the $\text{comp}$ results and constrain it to the interval $[0;1]$, MTDIFF divides it by the number of leaves in $q_o$. The similarity computation for the direct child $q_m$ of $i_m$ works analogously.
MTDIFF also normalizes \( \text{sumC} \) so that the actual size of the subtrees is no longer important. This normalization leads to the following definition of \( \text{sim}^{IC} \):

\[
\text{sim}^{IC}(i_o, i_m) = \begin{cases} 
0 & \text{lbl}(i_o) \neq \text{lbl}(i_m), \\
\frac{\text{sumC}(i_o, i_m)}{|\text{directch}(i_o)|+|\text{directch}(i_m)|} & \text{otherwise}
\end{cases}
\]

As before, nodes with different labels have a similarity of 0. Otherwise, MTDIFF computes the similarity of all direct children with \( \text{sumC} \) and divides it by the number of direct children of inner node \( i_o \) and the number of direct children of inner node \( i_m \). This division normalizes \( \text{sim}^{IC} \) to the interval \([0;1]\).

Args\(_o\) and Args\(_m\) in Fig. 3.16 provide a suitable example for \( \text{sim}^{IC} \). The direct child \(+_o\) of Args\(_o\) has the two children \(xy_o\) and \(2_o\). The mapping contains the pairs \((xy_o, xyz_m)\) and \((2_o, 2_m)\). \(xyz_m\) and \(2_m\) are both leaves of Args\(_m\). Thus, \(\text{comp} \) for \(+_o\) is 2. As \(+_o\) has two leaves, the computed similarity for the direct children in \(i_o\) is \(\frac{2}{2}\). As Args\(_m\) has two direct children, the second sum in \(\text{sumC}\) has two summands. The first belongs to \(+_m\) and is identical to \(+_o\). The second belongs to \(t_m\) and is \(\frac{0}{1} = 0\) as there is no partner node for \(t_m\) in AST\(_o\). For the normalization in \(\text{sim}^{IC}\), the direct children are important. Args\(_o\) has a single direct child, namely \(+_o\). Args\(_m\) has the two direct children, \(+_m\) and \(t_m\). The denominator of \(\text{sim}^{IC}\) is therefore \(|\text{directch}(i_o)|+|\text{directch}(i_m)| = 1 + 2 = 3\). The actual normalized similarity \(\text{sim}^{IC}\) of Args\(_o\) and Args\(_m\) is then \(\frac{1}{3}(1+1+0) = 0.67\). The use of \(\text{sim}^{IC}\) solves Challenge 2. Due to the division by the number of leaves in \(\text{sumC}\), all the subtrees have the same importance and the condition is no longer less important than the body of an if statement.

### 3.4.3 Thresholds and Weights

Internally, MTDIFF uses four weights \((w_{L0}, w_{L1}, w_{I0}, w_{I1})\), two thresholds \((\text{threshold}_L, \text{threshold}_I)\), and one constant \(\text{sim}_d\). All lie in the interval \([0;1]\). In the previous work on MTDIFF [89], Particle Swarm Optimization (PSO) [182] was used to determine these thresholds and weights. In this old version the constant \(\text{sim}_d\) had a fixed value of 0.2. For this thesis, Hill Climbing [285] was used to compute the values as it produced better results in shorter time. For both optimization methods, the fitness function was identical and based on a comparison with the results of GT\(_{A-F}\). In each iteration of the algorithms, the results
of MTDIFF and GT$_{A-F}$ were compared on the file changes in four smaller repositories (Checkstyle [60], Fitlibrary [116], JGraphT [168], and JUnit [175]) made before the end of 2015. If MTDIFF had an equal or a larger edit script compared to GT, the fitness value was decreased by one, otherwise it was increased by 1. A naive fitness function that only minimizes the number of edit operations across all files was not a suitable option to generate the configuration for MTDIFF. Such a naive fitness function would only focus on the edit script minimization of large files with many changes, as these files would have more weight on the results than smaller ones. As both ARES and C3 use method changes, the configuration values have to produce a short edit script even for small isolated changes.

For this thesis, the complete hill climbing algorithm was executed five times with different start values. Each time the execution used the step size 0.05 for each iteration. To create a start values for the five executions, all values were initialized with the same constant (0.0, 0.25, 0.5, 0.75, 1.0). Thus, for the first start value, all weights, all thresholds, and the constant were set to 0.0. The other four start values were set in a similar fashion. The best result from these five executions was the start value of a second run. In this second run, the step size was 0.01 and only this single start value was used. The fitness function was the same for both runs. The best configuration after the second run was $w_{L0}=0.77$, $w_{L1}=0.02$, $w_{I0}=0.15$, $w_{I1}=0.30$, $\text{threshold}_{L}=0.95$, $\text{threshold}_{I}=0.07$, and $\text{sim}_{d}=0.5$.

These values give a general insight into the mechanisms behind MTDIFF. The high leaf threshold and low threshold for inner nodes show that only nearly identical leaves avoid unnecessary moves and updates. For inner nodes, nearly all similarity values give acceptable pairs. Still, a value of zero decreases the fitness. The leaf mapping weights show that the leaf similarity of the parent hierarchy is much more important than the position of the nodes. The results for the inner node mapping weights indicate that the similarity of the children is only twice as important as the similarity of the node values. The high value for $\text{sim}_{d}=0.5$ shows that, even if inner nodes have different discrete values, their similarity should be above zero. Otherwise, the edit script grows for some files.
3.5 Complexity Analysis

This section discusses the time complexities of $\Theta_A$–$\Theta_F$ and MTDIFF. The focus of this section lies on the worst case complexities of the algorithms, because they illustrate the execution time on degenerated ASTs. However, due to the semantics behind ASTs, degenerated ones occur rarely in actual code. For example, it is unlikely that almost every subtree in a file has changed and $\Theta_A$ does not find pairs for the mapping. Regarding MTDIFF, the worst case occurs if almost all leaves and all statements in a program are identical. This also happens rarely and mostly in test files. Nevertheless, the complexities give an estimation of the execution times. The following discussion assumes that the access to a mapped node ($\text{mappedNode}$ in the algorithms described above) is in $O(1)$ due to hash maps.

$\Theta_A$ needs to create a fingerprint for each node. Then it looks for identical fingerprints in AST$_o$ and AST$_m$. With a hash map that stores the fingerprints, the complexity of $\Theta_A$ is in $\mathcal{O}_{\Theta_A}(n)$ [112], where $n = \max(\text{nodes}(\text{AST}_o), \text{nodes}(\text{AST}_m))$. The worst case complexity of $\Theta_B$ is in $\mathcal{O}(n^3)$, due to many possible executions of the LCS algorithm ($\mathcal{O}(n^2)$). However, as in other cases, this is rare (or non existent) in real code. $\Theta_B$ executes the LCS algorithm on the parents of all unmapped nodes that have a parent in the mapping. Again, in the worst case this means that $\frac{n}{2}$ unmapped nodes have a parent in the mapping and each LCS execution does not add additional pairs to the mapping.

$\Theta_C$ iterates over all unmapped leaves. Thus, in the worst case it has a complexity of $\mathcal{O}(l)$, where $l = \max(\text{leaves}(\text{AST}_o), \text{leaves}(\text{AST}_m))$. $\Theta_D$ compares all inner nodes with each other and thus its complexity lies in $\mathcal{O}(i^2)$, where $i = \max(\text{innerNodes}(\text{AST}_o), \text{innerNodes}(\text{AST}_m))$. $\Theta_E$ compares $l^2$ leaves in the first step and adds at most $\frac{l}{2}$ items to the work list. In the second step, $\Theta_E$ compares these $\frac{l}{2}$ with $l$ other leaves and adds at most $\frac{l}{4}$ items to the work list. The comparison ends in $\log l$ steps. As this forms a geometric series ($l^2 \cdot \sum \frac{1}{2^n}$) the worst case complexity of $\Theta_E$ is in $\mathcal{O}(l^2)$. $\Theta_F$ iterates over all $n$ nodes and examines at most $n$ children for each node. However, $\Theta_F$ visits each node only twice, once as parent and once as child. Thus, the complexity is in $\mathcal{O}(n)$.

MTDIFF starts with the leafMapping step. Without the ambiguity treatment, this step requires $l^2$ comparisons with cost $c_L$ to compute the similarity of the leaves. Then the leafMapping step sorts the $l^2$ resulting pairs in $\mathcal{O}(l^2 \cdot \log(l^2))$. In the worst case, the AST contains
3 Tree Differencing

\(l \cdot (l - 1)\) identical leaves. Thus, the similarity matrix \(S\) contains \(l^2\) entries. In the worst case, \(\text{computeSimilarityMatrix}\) has to update the similarity matrix \(i^2\) times until it reaches the root nodes. To update each entry requires the execution of \(\text{treeMapping}\) with \(\text{resolveAmbiguity}\) turned off. This execution uses the simplified \(\text{leafMapping}\) with a complexity in \(O(l^2 \cdot \log(l^2) \cdot c_L)\). The \(\text{fastInnerNodeMapping}\) needs to compare \(i^2\) inner nodes. The cost of \(\text{sim}_{IV}\) is \(c_I\). The cost of \(\text{sim}_{IC}\) is \(l\), as it examines all leaves in the inner nodes. Thus, the complexity of \(\text{fastInnerNodeMapping}\) is in \(O(l^2 \cdot \log(l^2) \cdot c_L)\) + \(i^2\cdot (c_I + l)\). This leads to the complexity \(\mathcal{O}(i^2 \cdot i^2 \cdot (l^2 \cdot \log(l^2) \cdot c_L) + i^2 \cdot (c_I + l))\) for \(\text{computeSimilarityMatrix}\). With dynamic programming it is possible to reduce this to \(\mathcal{O}(i^2 \cdot (l^2 \cdot \log(l^2) \cdot c_L) + i^2 \cdot (c_I + l))\), as it allows the reuse of the subtree comparison results. Additionally, it is possible to cache the similarity values of the leaves and \(\text{sim}_{IV}\) of the inner nodes. This further reduces the complexity to \(\mathcal{O}(i^2 \cdot (l^2 \cdot \log(l^2) + i^2 \cdot l) + l^2 \cdot c_L + i^2 \cdot c_I)\). The computation of the position value lies in \(\mathcal{O}_F(l^2)\), the Hungarian method for MTDIFF is in \(\mathcal{O}_H(l^3)\).

The \(\text{innerNodeMapping}\) requires to sort \(i^2\) pairs with a complexity of \(i^2 \cdot \log(i^2)\). Additionally, it needs the same operations as \(\text{fastInnerNodeMapping}\) and the complexity is \(\mathcal{O}_I(i^2 \cdot (c_I + l + \log(i^2)))\). If the comparison costs \(c_I, c_L\) are in \(\mathcal{O}(1)\), the complexity of MTDIFF is in

\[
\mathcal{O}_\text{MTDIFF} = \mathcal{O}_S + \mathcal{O}_P + \mathcal{O}_H + \mathcal{O}_I + \mathcal{O}_\Theta_{A-F} = \mathcal{O}(i^2 \cdot (l^2 \cdot \log(l^2) + i^2 \cdot l) + l^2 \cdot c_L + i^2 \cdot c_I + l^2 + l^3 + i^2 \cdot (c_I + l + \log(i^2))) + \mathcal{O}_\Theta_{A-F} = \mathcal{O}(i^2 \cdot (l^2 \cdot \log(l^2) + i^2 \cdot l) + l^3 + i^2 \cdot \log(i^2)) + \mathcal{O}_\Theta_{A-F} = \mathcal{O}(i^2 \cdot (l^2 \cdot \log(l^2) + i^2 \cdot l) + l^3 + i^2 \cdot \log(i^2))
\]

This is above the complexity of CD with \(\mathcal{O}_CD(l^2 \cdot (c_L + 1 + \log(l^2)) + i \cdot (c_I + l))\). Still, due to \(\Theta_A\), the following evaluation shows that for most file changes, MTDIFF with the built-in \(\Theta_{A-F}\) is faster than the algorithms CD, RTED and JSync.

### 3.6 Evaluation

The goal of this evaluation is to answer 5 questions:

1. Do the optimizations \(\Theta_{A-F}\) reduce the size of the edit scripts?

2. Does MTDIFF produce shorter edit scripts compared to the combination of other algorithms with \(\Theta_{A-F}\)?

3. Does MTDIFF detect more code parts that were moved by developers compared to other approaches?
4. How time consuming are the optimizations $\Theta_{A-F}$ and the MTDIFF heuristics?

5. Do developers perceive shorter edit scripts to be more helpful?

To answer these questions, the following sections analyze the results of different algorithms (including MTDIFF) with and without the optimizations $\Theta_{A-F}$ on different code changes. The basis for this evaluation is the GT framework [135]. For GT, RTED, and CD, this framework provides the implementations of phase 1 (Mapping Creation). In the framework, GT and RTED use the same fine-grained Java AST from the Java Development Toolkit (JDT) [166] (called GT-AST below). The CD implementation of this framework uses a different coarse-grained AST (called CD-AST below). Additionally, as part of the previous work for this thesis [89], a publicly available JSync version [173] was added to the GT framework. It uses a different AST (called JSync-AST below) that is similar to the GT-AST. One of the small differences is that the JSync-AST does not use import statements. The parser for the JSync-AST just ignores them. Furthermore, a publicly available version of JDime [167] was added to the GT framework. For the measurement, the lookahead of JDime was deactivated to speed up the execution time. JDime uses the AST of the ExtendJ compiler [110]. This AST is called JDime-AST below. It is fine-grained and very verbose (see Fig. 3.26). For example, the JDime-AST adds an extra List node if a node can have multiple children according to the syntax. It also requires two + nodes to express the computation $a + b + c$.

All implementations share the same implementation for the optimizations $\Theta_{A-F}$ and for phase 2 that generates the edit operations. Thus, phase 2 does not influence the results. As all algorithms use different ASTs, the following evaluation never compares the edit scripts of different algorithms directly, it only compares the algorithms with and without $\Theta_{A-F}$. The exception is MTDIFF, as it is configurable and supports all four AST types. This makes the comparison in Sec. 3.6.2 possible.

To generate the results, a 3.6 GHz Intel Xeon CPU equipped with 128 GB of RAM was used. The machine used Ubuntu in version 16.10 with a 4.8.0 Linux Kernel and OpenJDK in version 1.8.0. The dataset for the evaluation origins from the commits to nine open-source Java projects (Ant [15], Checkstyle [60], Cobertura [64], DrJava[93], Eclipse JDT Core [101], Eclipse SWT [102], Fitlibrary [116], JGraphT [168], and
3 Tree Differencing

(a) Simple code block.

```
01: {  
02:   foo(a + b + c, bar());  
03: }
```

(b) ASTs for the example.

GT-AST:  

```
Block
   Call
      foo
         +
         a
         b
         c
         bar
```

JDime-AST:  

```
Block
   Listα
      1: Call; v: foo
      Listβ
         1: Call; v: bar
```

JUnit [175]). 8 of the 9 projects (all except Fitlibrary) have commits in 2016 and all 9 provide a Git repository. 8 of the 9 projects are part of the Qualitas Corpus [303]. The exception is Eclipse SWT. It is not part of the Corpus, but was used in the evaluation of ARES and C3 (see Sec. 4.8 and Sec. 5.5).

From the repositories, the dataset uses all Java file changes in the commits to the master branches (without merges) before 2016-01-01. In these commits, 73,153 file changes affect only comments or whitespace. As such changes are not visible in ASTs, they are not part of the evaluation dataset. To identify these files, MTDIFF was run with the GT-AST. The 73,153 file changes all had an edit script of size zero.

The RTED implementation in the GT framework requires a lot of memory. For file changes with more than 20,000 AST nodes, RTED would require more than the 128 GB RAM that are available in the
workstation. Therefore, the data set also does not contain 8,849 file changes that have more than 20,000 AST nodes in the GT-AST.

After excluding all these files, the final dataset used in the evaluation consists of 152,979 file changes. For each of these file changes, the tree differencing algorithms were run on the file before the commit and the file after the commit. Excluding I/O overhead, the complete sequential time for the evaluation was 163 days.

### 3.6.1 Effects of the General Optimizations

ARES and C3 require a short edit script that expresses method changes accurately. This leads to the question whether or not the optimizations $\Theta_{A-F}$ have an effect on the edit script size. Table 3.1 holds the results for the 152,979 file changes from the 9 repositories. This table compares the baseline algorithms (GT, RTED, CD, JSync, and JDime) with their optimized versions (i.e., $\text{GT}_{A-F}$, $\text{RTED}_{A-F}$, $\text{CD}_{A-F}$, $\text{JSync}_{A-F}$, and $\text{JDime}_{A-F}$). It lists two comparisons. The first one is based on the number of pairs in the mapping ($\text{map.}$), the second one is based on the number of edit operations in the edit script ($\text{size}$). A mapping is defined as better if it contains more pairs compared to another mapping on the same file change. The edit script size is better if it contains fewer edit operations compared to another edit script on the same file change. Table 3.1 lists the mapping besides the edit script size as it is an indicator for the number of moves in a mapping. The more pairs in the mapping, the higher are the numbers of unchanged, updated or moved nodes. The mapping size also shows the effect of the optimizations $\Theta_{A-F}$ more directly, as they manipulate the mapping instead of the edit script. Nevertheless, the following discussion focuses on the edit script size as it is more relevant for ARES.
The column *Baseline better* of Table 3.1 holds the number of changed files (i.e., file revisions) for which the two measurements (mapping and edit script size) of the baseline algorithms are better compared to the optimized versions of the algorithms. The column *With Θ_{A−F} better* shows the number of file changes in which the optimized versions, i.e., the five baseline algorithms in combination with the optimizations Θ_{A−F} produced better results. The *Same Size* column lists the number of files for which both versions produced the same measurement results.

For all five algorithms (GT, RTED, CD, JSync, and JDime), the results of the optimized versions are better compared to the baseline version in both metrics. From 19% up to 56% of all file changes, the algorithms in combination with the six optimizations produce a shorter edit script. With a value below 1%, the negative effect of the optimizations is also negligible. Except for JDime, most file changes fall into the *Same Size* category. This is to be expected as a large portion of real code changes are simple or do not mix code change types (e.g., they contain only an added method). Such changes pose no problem for the baseline algorithms. JDime is an exception due to four reasons. First, it does not identify updates. Second, it only identifies moves for nodes in the unordered category. Third, it stops its top-down approach if labels of two nodes do not match. As a consequence, JDime does not compare two code parts for similarities if their parents are different. This can increase the edit script. Fourth, it does not execute Θ_{A} on the complete files, but only on some subtrees. This further increases the edit scripts compared to the optimized version in some cases. In summary, these four reasons cause the lower number of 66,516 file changes that have the same size in the JDime baseline version and its optimized variant.

Compared to the other four algorithms, the negative effect of the optimizations on the RTED results is higher. This highlights the special position of RTED in this setting. If a file change originally contains no code movements, RTED produces an optimal edit script. If the optimizations change such an optimal script, they create unnecessary moves. As a consequence, the edit scripts of RTED are shorter compared to the optimized versions in 1,381 cases. Nevertheless, the combination of RTED with Θ_{A−F} creates shorter edit scripts for 34,417 file changes and hence is a valid alternative to other tree differencing approaches that support code movements. A drawback of RTED_{A−F} is the high execution time (see Sec. 3.6.4).
Table 3.2 and Table 3.3 provide a closer look into the effect of the different optimizations. In contrast to Table 3.1, Table 3.2 contains a comparison of GT with different optimized versions and Table 3.3 contains a comparison of JSync with different optimized version. Each optimized version consists of the baseline algorithm plus a single additional optimization. As $\Theta_A$ is an intrinsic part of GT, it does not influence the GT results. For JSync it improves (i.e., shortens) the edit script for 10% of the file changes. In some cases, it has a negative effect on the edit script size. Responsible for this is its interaction with JSync. $\Theta_A$ accurately detects statements that remain in place between the original and the modified file version. However, JSync divides code into unchanged and changed parts. This prevents JSync from assigning nodes above the unchanged statements to nodes below them. $\Theta_B$ corrects such problems. As $\Theta_B$ and $\Theta_C$ only change unmapped nodes, they have no negative effect on the GT edit script size. Instead, they reduce the required edit operations for several thousand files. Both optimizations also have the largest impact on the GT results and reduce the edit script size for many changes. Table 3.3 shows a few file changes for which the edit script grows with $\Theta_B$ and $\Theta_C$. This is not caused by the algorithms, but by the JSync implementation that uses heuristics that are not deterministic for all input files.

$\Theta_D$ has the smallest effect compared to the baseline algorithm. As all of the baseline algorithms already optimize the pairs of inner nodes in the mapping, there is not much potential for improvement. However, $\Theta_D$ is able to improve the results after the execution of $\Theta_B$ and $\Theta_C$. Both optimizations influence the pairs of mapped children and thus make an evaluation of inner nodes by $\Theta_D$ useful. There are some file changes

Table 3.2.: Effects of $\Theta_A$−$F$ on the GT results.

<table>
<thead>
<tr>
<th></th>
<th>GT Baseline better</th>
<th>With $\Theta$ better</th>
<th>Same Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GT_A$ (GT-AST)</td>
<td># map.$\uparrow$</td>
<td>0 (00.0%)</td>
<td>0 (00.0%)</td>
</tr>
<tr>
<td></td>
<td># size.$\downarrow$</td>
<td>0 (00.0%)</td>
<td>0 (00.0%)</td>
</tr>
<tr>
<td>$GT_B$ (GT-AST)</td>
<td># map.$\uparrow$</td>
<td>0 (00.0%)</td>
<td>33,085 (21.6%)</td>
</tr>
<tr>
<td></td>
<td># size.$\downarrow$</td>
<td>0 (00.0%)</td>
<td>25,948 (17.0%)</td>
</tr>
<tr>
<td>$GT_C$ (GT-AST)</td>
<td># map.$\uparrow$</td>
<td>0 (00.0%)</td>
<td>8,097 (05.3%)</td>
</tr>
<tr>
<td></td>
<td># size.$\downarrow$</td>
<td>0 (00.0%)</td>
<td>8,414 (05.5%)</td>
</tr>
<tr>
<td>$GT_D$ (GT-AST)</td>
<td># map.$\uparrow$</td>
<td>0 (00.0%)</td>
<td>143 (01.0%)</td>
</tr>
<tr>
<td></td>
<td># size.$\downarrow$</td>
<td>143 (01.0%)</td>
<td>246 (01.6%)</td>
</tr>
<tr>
<td>$GT_E$ (GT-AST)</td>
<td># map.$\uparrow$</td>
<td>71 (00.6%)</td>
<td>10 (00.0%)</td>
</tr>
<tr>
<td></td>
<td># size.$\downarrow$</td>
<td>71 (00.6%)</td>
<td>857 (05.5%)</td>
</tr>
<tr>
<td>$GT_F$ (GT-AST)</td>
<td># map.$\uparrow$</td>
<td>0 (00.0%)</td>
<td>0 (00.0%)</td>
</tr>
<tr>
<td></td>
<td># size.$\downarrow$</td>
<td>423 (00.3%)</td>
<td>4,044 (02.6%)</td>
</tr>
</tbody>
</table>
Table 3.3.: Effects of $\Theta_{A-F}$ on the JSync results.

<table>
<thead>
<tr>
<th></th>
<th>JSync Baseline better</th>
<th>With $\Theta$ better</th>
<th>Same Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSync $A$ (JSync-AST)</td>
<td># map.↑ 381 (00.2%)</td>
<td># size↓ 343 (00.2%)</td>
<td>JSync $A$ (JSync-AST)</td>
</tr>
<tr>
<td>JSync $B$ (JSync-AST)</td>
<td># map.↑ 7 (00.0%)</td>
<td># size↓ 88 (00.1%)</td>
<td>JSync $B$ (JSync-AST)</td>
</tr>
<tr>
<td>JSync $C$ (JSync-AST)</td>
<td># map.↑ 27 (00.0%)</td>
<td># size↓ 94 (00.1%)</td>
<td>JSync $C$ (JSync-AST)</td>
</tr>
<tr>
<td>JSync $D$ (JSync-AST)</td>
<td># map.↑ 74 (00.0%)</td>
<td># size↓ 192 (00.3%)</td>
<td>JSync $D$ (JSync-AST)</td>
</tr>
<tr>
<td>JSync $E$ (JSync-AST)</td>
<td># map.↑ 1,379 (00.9%)</td>
<td># size↓ 147 (00.1%)</td>
<td>JSync $E$ (JSync-AST)</td>
</tr>
<tr>
<td>JSync $F$ (JSync-AST)</td>
<td># map.↑ 73 (00.0%)</td>
<td># size↓ 173 (00.3%)</td>
<td>JSync $F$ (JSync-AST)</td>
</tr>
</tbody>
</table>

for which $\Theta_{D}$ produces a longer edit script. The problem of $\Theta_{D}$ is its limited view on the mapping. It optimizes only one pair of inner nodes at the same time. If the optimized pair has partners, it is possible that for these partners the pairs of the baseline algorithm were better than the pairs produced by $\Theta_{D}$. As a consequence, $\Theta_{D}$ can increase the edit script size.

For 857 file changes in Table 3.2, $\Theta_{E}$ improves the results. If it is run after $\Theta_{B}$ and $\Theta_{C}$, it can even improve more file changes. Despite these improvements, there are a few cases in which $\Theta_{E}$ leads to the creation of longer edit scripts. Line 1 in Fig. 3.27 holds an example. GT uses three updates to express the code change. Due to $\Theta_{E}$, the optimized version requires four edit operations. The move of obj1 creates this additional operation. In this case, the detection of the movement is still useful for ARES, as it can use the detection of identical code parts to optimize the patterns for a higher accuracy. However, such a mapping increases the edit script size and can cause problems for other tools. The replacement of updates with moves is also the reason behind the 136 cases that have a shorter edit script for the baseline version compared to the version with $\Theta_{E}$ in Table 3.2.

$\Theta_{F}$ also reduces the edit script size for 4,044 file changes and only increases it for 423 changes. For $\Theta_{D}$, the narrow focus on single code pairs is the reason for the 423 file changes that have a longer edit script after the use of $\Theta_{F}$. The optimization traverses the ASTs top-down to identify more suitable partners for moved nodes. However, it is possible that it changes a node pair and as a consequence causes additional move operations in the children of the changed pair.
3.6 Evaluation

Original code:  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01:</td>
<td>obj1.obj2(obj3);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02:</td>
<td>l = m;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03:</td>
<td>n = o;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04:</td>
<td>int f = lock . unlock();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05:</td>
<td>lock . lock();</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Modified code:  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01:</td>
<td>obj4 . obj5 ( obj1);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02:</td>
<td>l = t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03:</td>
<td>n = m;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04:</td>
<td>int f = lock . lock();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05:</td>
<td>lock . unlock();</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) GT requires 7 edit operations.

(b) GT in combination with $\Theta_{A-C}$ requires 9 edit operations.

(c) MTDIFF requires 10 edit operations.

Figure 3.27.: Tough code changes for MTDIFF. Same legend as in Fig. 3.4.

Another aspect to discuss is the influence of the optimizations on the edit script size. For comparison, Fig. 3.28 shows the script sizes for the different algorithms on the evaluated file changes. Over 75% of all file changes require fewer than 50 edit operations. Due to the huge part of file changes with equal edit script size for the compared versions, the differences between algorithms are barely visible in this diagram. Therefore, Fig. 3.29 shows the edit script differences for the With $\Theta_{A-F}$ better column. Using GT as an example, the median size difference to GT$_{A-F}$ is 4. As 75% of all file changes in GT have a size below 50, the optimizations reduce the edit script size by at least 8% for the cases in which they have a positive effect. In many cases the improvement is even larger. Overall, this answers the initial question. The optimizations reduce the number of edit operations for several thousand files independent of the baseline algorithms. They also reduce the edit script size by 8% or more in most cases.

3.6.2 Performance of MTDIFF

Even with activated optimizations, the current state-of-the-art algorithms produce too many unnecessary edit operations for ARES. This led to the development of MTDIFF. Instead of a comparison with the
baseline algorithms, this section compares MTDIFF directly with the optimized versions. This increases the challenge for MTDIFF as for nearly all file changes the results of the optimized versions are either better than or equal to their baseline versions. Table 3.4 holds the results of the comparison with MTDIFF. The third column lists the number of file changes for which the results of the optimized versions \((\text{GT}_A - F, \ \text{RTED}_A - F, \ \text{CD}_A - F, \ \text{JSync}_A - F, \ \text{and JDime}_A - F)\) are better than MTDIFF. The fourth column holds all the file changes for which MTDIFF produces better results.

Even compared to the optimized versions, MTDIFF still produces shorter edit scripts for 17% up to 21% of all file changes. However, MTDIFF also produces longer scripts for 3% up to 17% of the file changes. An outlier in this comparison is the high value of 17% in
3.6 Evaluation

Figure 3.29.: Positive size differences between the baseline and the optimized versions. The boxes show the 25%/75% quartiles, the lines indicate the medians. The lower whiskers are at the minimum, the diamonds show the mean.

the comparison with JDime. Responsible for this high value are the List nodes in the JDime-AST. They cause problems for the MTDIFF heuristics because they have no value and often their subtrees have the same leaves. For example, List_α and List_β in Fig. 3.26 both have the same number of leaves and thus they always have the same similarity value in MTDIFF. Thus, only their position in the tree determines their partner in the mapping. In combination with deleted or inserted List nodes, this can lead to pairs in the mapping that create unnecessary moves. Additionally, List nodes can be inner nodes as well as leaves (see List_γ in Fig. 3.26). As the heuristics in MTDIFF do not identify pairs between inner nodes and leaves, a code change that converts a List node into a leaf or vice versa can also lead to an increase of the edit script size. The heuristics in JDime in combination with Θ_B are better suited to detect the List node pairs that lead to short edit scripts.

Table 3.4.: Performance of MTDIFF on changed files.

<table>
<thead>
<tr>
<th>Alg. with Θ_A−F</th>
<th>MTDIFF better</th>
<th>Same Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT_A−F (GT-AST)</td>
<td># map↑ 2,067 (01.4%)</td>
<td>25,526 (16.7%)</td>
</tr>
<tr>
<td></td>
<td># size↓9,190 (06.0%)</td>
<td>29,515 (09.3%)</td>
</tr>
<tr>
<td>RTED_A−F (GT-AST)</td>
<td># map↑ 1,484 (01.0%)</td>
<td>30,908 (20.2%)</td>
</tr>
<tr>
<td></td>
<td># size↓ 8,883 (05.8%)</td>
<td>32,129 (21.0%)</td>
</tr>
<tr>
<td>CD_A−F (CD-AST)</td>
<td># map↑ 3,784 (02.5%)</td>
<td>10,294 (06.7%)</td>
</tr>
<tr>
<td></td>
<td># size↓ 4,383 (02.9%)</td>
<td>25,913 (16.9%)</td>
</tr>
<tr>
<td>JSync_A−F (JSync-AST)</td>
<td># map↑ 1,721 (01.1%)</td>
<td>27,924 (18.3%)</td>
</tr>
<tr>
<td></td>
<td># size↓ 8,107 (05.3%)</td>
<td>31,789 (20.8%)</td>
</tr>
<tr>
<td>JDime_A−F (JDime-AST)</td>
<td># map↑ 2,148 (01.4%)</td>
<td>33,136 (21.7%)</td>
</tr>
<tr>
<td></td>
<td># size↓ 26,295 (17.2%)</td>
<td>31,704 (20.7%)</td>
</tr>
</tbody>
</table>
3 Tree Differencing

$\Theta_B$ offers no improvement for MTDIFF as MTDIFF creates pairs of List nodes and thus there are no unmapped nodes for $\Theta_B$ to optimize.

The main cause for longer scripts on the other ASTs is the aggressive detection of moves in MTDIFF. This is not critical for ARES as it can use correctly detected moves to create more accurate recommendations, even if they increase the edit script size. However, this is not true for all use cases of tree differencing algorithms. The lines 2–3 in Fig. 3.27 show an example that illustrates the problem. MTDIFF detects the movement of $m$. This replaces the two update operations with one move, one delete, and one insert and thus increases the edit script size by 1.

As the aggressive detection of moves is the main reason for the edit script size growth on most ASTs, MTDIFF only enlarges the edit script by a few edit operations. As a consequence, the median size increase is at a low value of 2. However, there are a few cases in which the heuristics of MTDIFF fail and create many unnecessary edit operations. In the worst case of the measurements for the GT-AST, MTDIFF creates 689 more edit operations than GT$_{A-F}$. There are several conditions that have to occur together to cause such a result. It is necessary that there are many changes in different methods and code blocks. The changed file also needs several hundred identical statements. In the evaluated repositories, this occurs in JUnit test cases, in configuration files or classes with many overloaded methods. In such a scenario, $\Theta_A$ offers no help. Due to the changes, it cannot add complete code blocks to the mapping. Due to the many identical statements, $\Theta_A$ is also unable to pair identical statements together. As the structure of the code blocks and statements is highly similar, the ambiguity treatment in MTDIFF does not help in identifying differences between identical leaves. Thus, only the positions of the leaves are the determining factor. If the position heuristics in MTDIFF fail to find the optimal partner for one leaf, this wrong leaf pair has the potential to trigger a cascade of other non-optimal pair. This propagates to the inner nodes. At the end, this cascade of non-optimal pairs leads to the large edit operation difference between GT$_{A-F}$ and MTDIFF in a few file changes. RTED and the dice heuristics in GT are more robust against such changes.

Compared to JDime$_{A-F}$, the median size growth is also at a low value of 2 as the node pairs that increase the edit script size are often only one or two List nodes. However, there are also some outliers. In the worst case of the measurements, MTDIFF creates 2,151 more edit operations than JDime$_{A-F}$. The corresponding file change contains many small
3.6 Evaluation

Figure 3.30.: Positive size differences between the optimized algorithms and MTDIFF. The boxes show the 25%/75% quartiles, the lines indicate the medians. The lower whiskers are at the minimum, the diamonds show the mean.

modified subtrees that have \textit{List} nodes as leaves. As a consequence, \( \Theta_A \) is unable to create many pairs for the mappings. Similar to the outliers in other trees, only the \textit{List} node positions are the determining factor for the mapping. In this scenario, a single non-optimal assignment of one \textit{List} node can trigger a cascade of other non-optimal assignments. This leads to the longer edit scripts in such outliers. The top-down approach in JDime in combination with \( \Theta_B \) is more suited for such a file change on the JDime-AST.

Although there are a few outliers on the negative side, there are also outliers on the positive side. Fig. 3.30 shows the edit script size differences between the optimized versions and MTDIFF. Even compared to the optimized versions, the median values show that MTDIFF shortens the edit script for the affected file changes by 5 or more edit operations on average. This reduces the edit script size by 10% for most files. In some cases, the reduction is even considerably larger. MTDIFF even achieves a reduction by 3,634 edit operations compared to the GT\(_{A−F}\) edit script for one change.

Such extreme cases occur if developers perform many different code changes in the same revision. In this particular case, a developer combined several refactorings. The refactoring types were identifier renaming, \textit{this} insertion, and method extraction. In such cases, \( \Theta_A \) is unable to detect large code movements due to the many changes. The \textit{dice} heuristics in GT also have trouble detecting code movements because
of the method extraction. As RTED also does not detect movements, its internal use in GT also does not offer help for such cases.

In summary, despite several negative results, MTDIFF still outperforms the optimized algorithms and creates shorter edit scripts for more file changes. On the JDime-AST the cause of longer script is often only a non-optimal pair of List nodes. An additional optimization could handle such cases. In other ASTs, the correct identification of moves by MTDIFF sometimes leads to longer edit scripts if there exists a shorter edit script with updates. In these cases, the detection of code movements can be beneficial for ARES and similar tools despite the larger size because it provides additional information about the code transformations made by developers.

3.6.3 Move Detection

So far the measurements have solely targeted the edit script size. However, this is not an indicator of how many code movements from developers a tree differencing algorithm detects. The example in Fig. 3.27 shows that an accurate detection of moves can even increase the edit script size, and thus the size is not an appropriate measurement.

To evaluate the move detection capabilities, this thesis follows the reasoning of Falleri et al. [112]. They argue that a tree differencing algorithm is more accurate in detecting moves and thus more likely to find the ideal number of moves for a code change, if the algorithm identifies more edit scripts that solely consist of moves.

Fig. 3.31 holds the results for this evaluation. An initial evaluation step determined the number of file changes for which at least one of the algorithms (GT, RTED, CD, JSync, JDime, GT\_A\_F, RTED\_A\_F, CD\_A\_F, JSync\_A\_F, JDime\_A\_F, or MTDIFF) created a move-only edit script. As in other parts of the evaluation, it is not valid to mix different AST types. MTDIFF was executed with different ASTs as input. It is possible that an edit script consists only of moves in one AST. For another AST, this can be impossible. For example, the JSync-AST does not include import statements. A file change that contains only code movements beside deleted import statements is a move-only edit script in the JSync-AST. However, in the GT-AST it is not a move-only edit script due to the deleted import statements. For the GT-ASTs, the algorithms identify 2,907 file changes that at least one algorithm can express with move-only edit scripts. For the CD-AST the total count is
3.6 Evaluation

Figure 3.31.: Fractions of move-only scripts that are found on the respective ASTs. The black bar belongs to the baseline algorithms, gray to the optimized versions with $\Theta_{A-F}$ added, and white to MTDIFF.

923, for the JSync-AST it is 2,939, and for the JDime-AST it is 2,924. The total count corresponds to the 100% in Fig. 3.31. The black bars show the percentage of file changes for which the baseline algorithms generate an edit script that solely consists of moves. The gray bars show the percentage of the optimized versions with $\Theta_{A-F}$, and the white bars the percentage of MTDIFF.

All the optimized algorithms create more move-only edit scripts for the dataset than their baseline counterparts. Unsurprisingly, RTED does not detect a move-only edit script without optimizations. MTDIFF even surpasses these values and creates more move-only scripts than the optimized algorithms. In fact, if GT$_{A-F}$, RTED, RTED$_{A-F}$, JSync, JSync$_{A-F}$, or JDime create a move-only edit script, MTDIFF also uses only moves. MTDIFF only misses 2 move-only file changes that GT identifies. The reason for this is $\Theta_F$ that changes a pair of inner nodes to avoid a move on a node nearer to the root. This causes additional edit operations in their children (see lines 4–5 in Fig. 3.27). MTDIFF also misses 2 move-only file changes that CD identifies and 8 move-only file changes that CD$_{A-F}$ identifies. In these 10 cases, the heuristics of MTDIFF for inner nodes create non-optimal pairs. A solution would be an adjustment of the weights in MTDIFF to a coarse-grained AST. In fact, the weight $w_{I1}$ for the children similarity for $\text{sim}_{IC}$ needs to be nearly zero to create a move-only script for these cases. MTDIFF also misses 36 move-only edit scripts on the JDime-AST. As before, the combination of the JDime-AST with the heuristics in MTDIFF causes these problems. In JDime, some identifiers are values of inner nodes
### 3 Tree Differencing

<table>
<thead>
<tr>
<th>Original code:</th>
<th>Modified code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: int a = 2;</td>
<td>int a = 1;</td>
</tr>
<tr>
<td>02: int b = 1;</td>
<td>int b = 2;</td>
</tr>
</tbody>
</table>

(a) JDime in combination with $\Theta_{A-F}$ requires 2 edit operations.

<table>
<thead>
<tr>
<th>Original code:</th>
<th>Modified code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: int a = 2;</td>
<td>int $\overline{a}$ = 1;</td>
</tr>
<tr>
<td>02: int b = 1;</td>
<td>int $\overline{b}$ = 2;</td>
</tr>
</tbody>
</table>

(b) MTDIFF requires 3 edit operations.

Figure 3.32.: Comparison between JDime$_{A-F}$ and MTDIFF. Same legend as in Fig. 3.2.

(see the method names in Fig. 3.26). Due to the weights in MTDIFF, the leaves have a higher impact than the values of inner nodes. This can lead to non-optimal node pairs that increase the size of the edit script and prevent a move-only edit script. Fig. 3.32 holds an example. In this example, MTDIFF uses the leaves 1 and 2 to determine the inner node pairs. The values of the declaration nodes $a$ and $b$ are less important as they are values on inner nodes that have a lower weight $w_{I0}$. The resulting inner node pairs avoid move operations for the leaves 1 and 2 but create two update operations for the declarations. The other cases of the 36 move-only edit scripts are similar.

Despite the 48 described outliers, MTDIFF identifies more move-only edit scripts in total. This indicates that it detects moves made by developers more accurately than other approaches.

#### 3.6.4 Execution Time

For many applications (e.g., code merges) it is essential that the tree differencing algorithms are fast. Therefore, this section takes a closer look at the execution time of the heuristics and MTDIFF. Fig. 3.33 holds the measured times. The measurements include only the time to execute phase 1 (*Mapping Creation*) and the time to execute the optimizations. Parsing the code into ASTs and the execution of phase 2 is not part of the measured times as both parts are independent of the algorithms. A special case is JSync. It requires a second parse step to identify changed and unchanged lines. The time of this line-based parsing is part of the measured JSync times.

The boxes in Fig. 3.33 show that for over 75% of the code changes, $\Theta_{A-F}$ only increase the execution times by a few milliseconds. Even on the largest files, the optimizations increase the execution time only
Figure 3.33.: Runtimes for the 152,979 file changes. The boxes show the 25%/75% quartiles, the lines indicate the medians. The lower whiskers are at the minimum, the diamonds show the mean.
by 4 s for GT and up to 40 s for RTED. This is an increase by at most 10%. There is no single optimization that is responsible for the slow execution times. The portion they contribute to the total execution time depends on the size of unmapped nodes and the number of pairs in the mapping in each file change.

In the results, CD$_{A-F}$ is even faster than the baseline CD. The reason is that the CD implementation makes use of the results from $\Theta_A$. It only examines nodes that are not part of the mapping after $\Theta_A$. For CD, it is easy to adapt the implementation in a suitable fashion. For RTED, JSync, and JDime this is more complicated. Therefore, to avoid unnecessary errors in the implementations, the use of $\Theta_A$ to speed up the computation was only included in the CD implementation for this measurement.

GT (with and without optimizations) is the fastest algorithm on the dataset, although it uses a fine-grained AST and thus also has to handle more nodes than CD or JSync. Due to the use of a coarse-grained AST, CD is faster than RTED, JSync, or JDime. As RTED does not rely on heuristics, its time consumption shows a cubic growth with the AST size. This leads to the higher range of the 75% quartile and the high worst case execution times of 806 s respectively 846 s.

The JDime-AST requires even more nodes than the fine-grained GT-AST. For the simple code fragment in Fig. 3.26, the JDime-AST contains 11 nodes whereas the GT-AST only has 9 nodes. These additional nodes are one of the reasons why JDime is slower than GT. Another reason is that JDime does not apply $\Theta_A$ on the complete file change. Thus, it has to examine more nodes than GT. This is also visible in the direct comparison of JDime with MTDIFF (on the JDime-AST).

JDime has an execution time of 682 s for a file change from the Eclipse JDT repository. For this change, MTDIFF has an execution time of 130 s although it has a higher complexity. The decisive factor for this time difference is the use of $\Theta_A$ by MTDIFF on the complete file. As a consequence, the execution times of JDime, JDime$_{A-F}$, and MTDIFF are similar.

MTDIFF is even faster than RTED and JSync. For over 75% of all files, its execution time is below 0.3 s despite its high complexity (see Sec. 3.5). There are two reasons for these results. First, $\Theta_A$ reduces the number of leaves and inner nodes that MTDIFF has to examine. Second, the implementation of MTDIFF uses dynamic programming. This avoids many identical similarity computations. As a consequence, its memory
consumption is above GT, CD, and JSync. However, it stays below the memory consumption of the RTED implementation.

Although MTDIFF is fast for most file changes, there are a few outliers in the measurements and the maximal measured execution time of MTDIFF on the GT-AST is 682 s. Besides the AST size, several properties of the file change cause this result. It is necessary that almost every method or code block is changed to prevent $\Theta_A$ from matching complete blocks. Additionally, each statement or expression has to appear at least twice. This prevents $\Theta_A$ from matching statements and expressions. In particular, the maximal measured execution time occurred on a file containing JUnit test cases. In this file, each test method was identical to all others except of one string constant per method. A refactoring that added `this` to all fields in a JUnit test caused the high execution time. The highly similar test methods and this large change caused the high execution time. In the future it is necessary to detect such cases in MTDIFF and to switch off the expensive leaf ambiguity treatment for such a change.

On the larger JDime-AST there are also several extreme outliers. For one such outlier on the JDime-AST, MTDIFF requires even 58,826 s ($\approx 16$ h) to compute the edit script (see Fig. 3.33). Due to the many code transformations in this file change, $\Theta_A$ does not reduce the amount of nodes that MTDIFF has to handle. Inside the original and the modified file there are also several hundred `List` nodes without children. MTDIFF handles them with its leaf ambiguity treatment. As this creates a large similarity matrix, its computation takes a lot of time. Additionally, the JDime-AST has more inner nodes than the GT-AST. This makes `fastInnerNodeMapping` slower and also increases the number of iterations in `computeSimilarityMatrix`. However, like for the GT-AST outliers, it is possible to detect such cases and to switch off the expensive leaf ambiguity treatment if needed.

Overall, for over 75% of the file changes in the dataset, the execution time of $\Theta_{A-F}$ is below a couple of milliseconds. This makes the optimizations a nearly cost free way to optimize tree differencing approaches. With an average execution time below 0.5 s for over 75% of the file changes, MTDIFF is also suitable for most applications including ARES and C3.
3.6.5 Script Size Questionnaire

This section is mostly based on previous work on MTDIFF [89]. It addresses the research question whether or not developers actually perceive shorter edit scripts to be more helpful. To answer this question, a questionnaire was presented to several participants. Each item in the questionnaire consisted of a code change for which the results of two tree differencing algorithms were presented next to each other. Instead of a list of edit operations, the participants evaluated a graphical representation similar to the one in Fig. 3.27. The only differences were the depictions of the updated and moved code parts. In contrast to Fig. 3.27, the updated and moved parts were highlighted in both the original and the modified code in the questionnaire.

Each code change that the participants evaluated was displayed in the same order: original code with the edit operation of algorithm one, modified code of algorithm one, original code of algorithm two, and modified code of algorithm two. For each of the questionnaire items, the participants had to answer the question: “Is the first or the second representation more helpful in understanding the intention of the change? [Allowed answers: First, Second, or Unclear]”.

The study consisted of 240 items taken randomly from the 126,162 file changes of the original MTDIFF evaluation [89]. Before the random selection, the file changes were filtered. Due to the filtering, the ballot contained only small changes (pairs in the mapping + edit operations ≤ 500). This was necessary as a preliminary study showed that it is hard for developers to evaluate large code changes. It was perceived especially hard if the code changes did not fit on one screen. Thus, selecting large code changes would simply increase the number of “Unclear” answers.

To produce results that are independent of the algorithms, the code changes were selected from 24 categories. The categories correspond to the edit script size cells in the Tables 3.1 and 3.4. If available, 10 changes per category were selected randomly.

In some cases, there were insufficient changes available. Thus, code changes in other categories of the same columns were randomly selected. For the categories in which the edit scripts had the same size, scripts with at least one different edit operation were chosen. For example, it is possible to express the move and update of a node with a delete and insert operation. In such a case, the size of the edit script is equal, but the edit operations differ. As the changes for the questionnaire were selected from the original MTDIFF evaluation [89], the changes were...
Table 3.5.: Questionnaire input (240 code changes).

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
<th>Algorithm with shorter script</th>
<th>Number of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GT</td>
<td>GT</td>
<td>GT</td>
<td>10</td>
</tr>
<tr>
<td>2 GT</td>
<td>GT</td>
<td>GT</td>
<td>10</td>
</tr>
<tr>
<td>3 GT</td>
<td>GT</td>
<td>Equal Size</td>
<td>11</td>
</tr>
<tr>
<td>4 RTED</td>
<td>RTED</td>
<td>RTED</td>
<td>14</td>
</tr>
<tr>
<td>5 RTED</td>
<td>RTED</td>
<td>Equal Size</td>
<td>10</td>
</tr>
<tr>
<td>6 RTED</td>
<td>RTED</td>
<td>Equal Size</td>
<td>10</td>
</tr>
<tr>
<td>7 CD</td>
<td>CD</td>
<td>CD</td>
<td>2</td>
</tr>
<tr>
<td>8 CD</td>
<td>CD</td>
<td>CD</td>
<td>10</td>
</tr>
<tr>
<td>9 CD</td>
<td>CD</td>
<td>Equal Size</td>
<td>9</td>
</tr>
<tr>
<td>10 JSync</td>
<td>JSync</td>
<td>JSync</td>
<td>14</td>
</tr>
<tr>
<td>11 JSync</td>
<td>JSync</td>
<td>JSync</td>
<td>10</td>
</tr>
<tr>
<td>12 JSync</td>
<td>JSync</td>
<td>Equal Size</td>
<td>10</td>
</tr>
<tr>
<td>13 GT</td>
<td>MTDIFF</td>
<td>GT</td>
<td>10</td>
</tr>
<tr>
<td>14 GT</td>
<td>MTDIFF</td>
<td>MTDIFF (GT-AST)</td>
<td>10</td>
</tr>
<tr>
<td>15 GT</td>
<td>MTDIFF</td>
<td>Equal Size</td>
<td>10</td>
</tr>
<tr>
<td>16 RTED</td>
<td>MTDIFF</td>
<td>RTED</td>
<td>10</td>
</tr>
<tr>
<td>17 RTED</td>
<td>MTDIFF</td>
<td>MTDIFF (GT-AST)</td>
<td>10</td>
</tr>
<tr>
<td>18 RTED</td>
<td>MTDIFF</td>
<td>Equal Size</td>
<td>10</td>
</tr>
<tr>
<td>19 CD</td>
<td>MTDIFF</td>
<td>CD</td>
<td>10</td>
</tr>
<tr>
<td>20 CD</td>
<td>MTDIFF</td>
<td>MTDIFF (CD-AST)</td>
<td>10</td>
</tr>
<tr>
<td>21 CD</td>
<td>MTDIFF</td>
<td>Equal Size</td>
<td>10</td>
</tr>
<tr>
<td>22 JSync</td>
<td>MTDIFF</td>
<td>JSync</td>
<td>10</td>
</tr>
<tr>
<td>23 JSync</td>
<td>MTDIFF</td>
<td>JSync</td>
<td>10</td>
</tr>
<tr>
<td>24 JSync</td>
<td>MTDIFF</td>
<td>Equal Size</td>
<td>10</td>
</tr>
</tbody>
</table>

computed without $\Theta_F$ and with an older version of MTDIFF. Table 3.5 lists the number of items in each category.

8 participants answered the questions to all code changes in the questionnaire. All hold a master or PhD degree in Computer Science or Mathematics and are familiar with Java and the line-based diff representation. It is possible that the participants grew more familiar with the graphical representation during the study. This could influence the results. To avoid such order effects, all 240 items were shown to the participants in a different random order. Additionally, the left and right tree differencing algorithms were switched randomly. This made sure that the shorter edit script of an item can occur on the left side for one participant, but on the right side for another participant.

Table 3.6 holds the result of the questionnaire. Under the condition that the first edit script was shorter, the majority of the participants also found this shorter edit script more helpful. Similar, if the edit script of the second algorithm was shorter, 354 times the participants found it more helpful. In both rows, the number of times the shorter version was found more helpful is above $\left\lfloor \frac{640}{3} \right\rfloor = 213$ of a completely random
3 Tree Differencing

Table 3.6.: Questionnaire results (1,920 answers).

<table>
<thead>
<tr>
<th></th>
<th>First more helpful</th>
<th>Second more helpful</th>
<th>Unclear</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>First shorter</td>
<td>270</td>
<td>180</td>
<td>190</td>
<td>640</td>
</tr>
<tr>
<td>Second shorter</td>
<td>106</td>
<td>354</td>
<td>180</td>
<td>640</td>
</tr>
<tr>
<td>Same Size</td>
<td>197</td>
<td>238</td>
<td>205</td>
<td>640</td>
</tr>
<tr>
<td>Sum</td>
<td>573</td>
<td>772</td>
<td>575</td>
<td>1,920</td>
</tr>
</tbody>
</table>

distribution. In contrast, if the edit scripts had the same size, the distance to a random distribution is lower.

The first step to answer the last research question of this chapter is an examination of the connection between shorter edit scripts and the more helpful answers of the participants. The null hypothesis is:

- The two nominal variables script size difference (in the displayed code change pairs) and the helpfulness perception by the participants are independent.

Pearson’s chi-squared test [263] examines such relationships between variables in large sample sets. For the test, the calculation of \( \chi^2 \) for the values in Table 3.6 is necessary:

\[ \chi^2 = \sum_{ij} \frac{(n_{ij} - E_{ij})^2}{E_{ij}} = \sum_{ij} \frac{\left( n_{ij} - \frac{1}{N} \sum_k (n_{ik}) \cdot \sum_k (n_{kj}) \right)^2}{\frac{1}{N} \sum_k (n_{ik}) \cdot \sum_k (n_{kj})} = \]

\[ = \left( \frac{n_{00} - \frac{1}{N} \cdot \sum_{k=0}^2 (n_{0k}) \cdot \sum_{k=0}^2 (n_{k0})}{\frac{1}{N} \cdot \sum_{k=0}^2 (n_{0k}) \cdot \sum_{k=0}^2 (n_{k0})} \right)^2 + \cdots + \left( \frac{n_{22} - \frac{1}{N} \cdot \sum_{k=0}^2 (n_{2k}) \cdot \sum_{k=0}^2 (n_{k2})}{\frac{1}{N} \cdot \sum_{k=0}^2 (n_{2k}) \cdot \sum_{k=0}^2 (n_{k2})} \right)^2 \]

\[ = \left( \frac{270 - \frac{1}{1920} \cdot 640 \cdot 573}{\frac{1}{1920} \cdot 640 \cdot 573} \right)^2 + \cdots + \left( \frac{205 - \frac{1}{1920} \cdot 640 \cdot 205}{\frac{1}{1920} \cdot 640 \cdot 205} \right)^2 = 133.34. \]

\( \chi^2 \) uses the expected value \( E_{ij} = N \cdot p_i \cdot p_j \) to determine the likelihood that an item is in a specific cell in Table 3.6. Here, \( p_i \) is the probability for a value to be in row \( i \) and \( p_j \) is the probability for a value to be in column \( j \). As \( \chi^2 \) is greater than the critical value 18.467 (4 degrees of freedom and probability 0.999) it is possible to reject the null hypothesis with \( p < 0.001 \). To show the effect of the edit script size on the helpfulness perception, it is necessary to exclude the results of the third row in Table 3.6, i.e., the Same Size categories. This reduces \( \chi^2 \) to \( \chi^2_{12} = 128.50 \) which is still significant.
3.6 Evaluation

For two nominal variables, Cramér [73] defines the association measurement \( V \) where 0 means no association and 1 is a full association:

\[
V = \sqrt{\frac{\chi^2}{N \cdot \min(rows-1, columns-1)}}
\]

With 2 rows, 3 columns and \( N = 1920 \) this gives \( V_{12} = 0.32 \) which indicates an association between the edit script size and the helpfulness. Cramér’s \( V \) gives also the means to compute Cohen’s effect size measurement which is \( w = V \cdot \sqrt{\text{rows} - 1} = 0.32 \) [66]. In the literature [66], this indicates a medium effect of the script size on the helpfulness. As \( w \) does not define whether the effect is positive or negative, an examination of Table 3.6 is necessary. The data shows that the participants perceived smaller edit scripts as more helpful instead of longer scripts. In combination with Cohen’s \( w \), this means that the reduction of the edit script size has a medium positive effect on the helpfulness. This answers the research question. The questionnaire shows that a smaller edit script was perceived more helpful by the participants.

3.6.6 Limitations and Threats to Validity

The evaluation only considered code changes in the Java language. This threatens the universal validity of the results. However, the optimizations \( \Theta_{A-F} \) do not use any language specific features, only general properties of ASTs like the label or the value. Thus, it is likely that they work with different programming languages. MTDIFF also does not rely on any language specific features. The evaluation even shows that MTDIFF works with ASTs of different granularities. This supports the assumption that both \( \Theta_{A-F} \) and MTDIFF work language-independently. However, the results on the JDime-AST show that there are still AST types that require an extension of the heuristics in MTDIFF.

Bugs in the implementation also remain a threat to the validity of the evaluation results. To avoid the influence of bugs on the results, the evaluation used the publicly available third-party source code from the GT framework [135] as far as possible. This framework contains implementations for GT, RTED, and CD. The creation of edit scripts from the mapping (phase 2) and the graphical representation of code changes that was used in the questionnaire are also part of this framework. If one assumes that phase 2 from the GT framework contains no severe bugs, the generated edit operations for these mappings were also correct. As a consequence, the evaluation shows that even with potential
bugs and limitations in the optimizations $\Theta_{A-F}$, they provide a benefit for tree differencing algorithms in general. They created mappings that reduced the size of the edit scripts. Similarly, despite bugs and limitations, MTDIFF produced mappings that led to shorter edit scripts compared to the other tree differencing algorithms.

However, despite the precautions and the use of tested third-party source code, there were still bugs that influenced the results of this work. In the CD implementation of GT, the sort order of leaves was wrong. This problem only became known during the work on this thesis. Fixing the sort order reduced the edit script size of CD considerably compared to the initial CD evaluation in the previous work on this topic [89]. Instead of the previously reported 98%, CD$_{A-F}$ only shortens the edit script for 43.7% of the analyzed file changes. The number of file changes for which MTDIFF produces a shorter edit script compared to CD$_{A-F}$ also dropped from 41% to 16.9%. Although the differences got smaller, there is still a measurable benefit of the optimizations and MTDIFF compared to CD.

For JSync, it was necessary to adapt a publicly available version to the GT framework. Assuming that phase 2 in the GT framework produced correct results, the improvements by $\Theta_{A-F}$ were in the same order of magnitude as for GT and CD. This again supports the hypothesis that $\Theta_{A-F}$ can reduce the edit script sizes for different algorithms. JSync also is non-deterministic for some file changes. However, this only affects a few files and increases or decreases the edit script only by one or two edit operations in most cases. Thus, it has no effect on general results due to the larger size reduction by the optimizations and the large number of file changes in the evaluation.

The conclusions that can be drawn from the edit script questionnaire are limited. The questionnaire used only a few participants and their programming experience was very homogeneous. All held at least a masters degree in computer science or mathematics and were members of the same university. Thus, the questionnaire is only the first step in the research about the perception of tree differencing results by developers. To facilitate further studies in this area, the questionnaire and anonymous raw data is publicly available (https://github.com/FAU-Inf2/tree-measurements).
3.7 Summary

Although widely used, line-based code change representations provided by `diff` are too coarse-grained for tools like ARES or C3. A more precise solution are tree-based algorithms that work on ASTs. In general, they work in two phases. Phase 1 identifies a mapping, i.e., node pairs in which the first node is part of the AST from the original source code and the second node is part of the AST from the modified source code. Phase 2 creates an edit script from this mapping that transforms the original AST into the modified AST. The challenge lies in determining the mapping that leads to the shortest edit script. This is important, as tools like ARES rely on a compact expression of code changes. It is possible to divide the tree differencing algorithms into two independent classes. The first class consists of optimal algorithms that identify the shortest (i.e., the optimal) edit script for a code change. The disadvantage of approaches in this class is that they are limited to the edit operations `insert`, `delete` and `update`. The second class of algorithms also supports `move` operations. Detecting moves is essential for ARES to create accurate recommendations. In contrast to the first class, the second class of algorithms is limited to the use of heuristics as the problem to find the shortest edit scripts with `move` operations is NP-hard.

As all algorithms from the second class use heuristics, there are many conditions under which the heuristics produce unnecessarily large edit scripts. This chapter examined the heuristics of the four algorithms GT, CD, JSync, and JDime. It also explored the possibility of using RTED from the optimal class in combination with a phase 2 implementation that supports `move` operations. The analysis of these five algorithms led to the development of the six optimizations $\Theta_{A-F}$. It is possible to combine these optimizations with tree differencing algorithms to create shorter edit scripts. The optimization $\Theta_A$ is a preprocessing step that was already used in combination with other tree differencing algorithms (e.g., GT) in the past. An evaluation of $\Theta_A$ on JSync showed that it is a valid addition to each new tree differencing approach that relies on heuristics. In the evaluation of this chapter, of $\Theta_A$ reduced the size of the edit scripts for 15,661 (10%) of the analyzed code changes and only increased the edit script size for 343 cases. The evaluation also showed that $\Theta_A$ can reduce the execution times of tree differencing algorithms (e.g., for CD). The other optimizations $\Theta_{B-F}$ are post-processing steps that improve the mapping after the run of the original
3 Tree Differencing

algorithms. As input they only require the current mapping and both ASTs. \( \Theta_B \) uses the LCS algorithm to pair nodes missed by the original algorithms. \( \Theta_C \) finds new pairs with previously unmapped leaves. \( \Theta_D \) updates the pairs of inner nodes based on the new information available after the execution of \( \Theta_B \) and \( \Theta_C \). \( \Theta_E \) analyzes all leaves and looks for non-optimal leaf movements. Finally, \( \Theta_F \) reduces movements that switch identical nodes in place. The evaluation in this chapter showed that the optimizations in combination with the considered algorithms reduce the edit script for 19\% up to 56\% of all examined file changes. The optimizations increased the execution time only slightly. For CD, \( \Theta_A \) even decreased the execution time.

The analyzed algorithms in combination with \( \Theta_{A-F} \) still create unnecessarily large edit scripts for many code changes. The problem is the inflexibility of GT, RTED, JSync, and JDime to detect moves across changed subtrees. CD is more flexible, but it was designed for a coarse-grained AST in which statements are leaf nodes. Thus, the CD heuristics are ill suited for a fine-grained one. However, such fine-grained ASTs are common in compilers and especially ARES requires them to create accurate recommendations. To address this issue, this chapter introduced the new tree differencing algorithm MTDIFF. Based on the CD structure, it uses a two-step approach to compute the mapping. The first step creates leaf pairs for the mapping. The second step uses these leaf pairs to identify pairs of inner nodes that lead to a short edit script. However, MTDIFF differs from CD in two aspects. First, it was designed with the expectation that \( \Theta_{A-F} \) are used. Especially the use of \( \Theta_A \) is crucial. Without it, the execution time of MTDIFF is too high. In contrast to CD, the heuristics of MTDIFF take fine-grained ASTs into account. MTDIFF uses the parent hierarchy during the creation of leaf pairs and also uses similarity heuristics for inner nodes that differ from CD. Additionally, MTDIFF uses other thresholds adapted to its heuristics.

The evaluation showed that the use of MTDIFF further reduced the size of the edit script for 17\% up to 21\% of all examined code changes. Thus, it creates shorter scripts than the optimized GT, RTED, CD, JSync, and JDime versions. Additionally, the evaluation showed that MTDIFF is more precise in detecting movements than GT, RTED, JSync, and JDime. Due to the higher complexity, MTDIFF is slower than GT. However, it still manages to compute the edit script for 75\% of the analyzed code changes within 0.206 seconds.
The evaluation concluded with a small questionnaire that examined the connection between shorter edit scripts and the helpfulness of these scripts for developers. The questionnaire showed that the reduction of the edit script size had a medium positive effect on the helpfulness. In general, the participants of the experiment found shorter scripts more helpful in understanding the intention of the changes compared to larger edit scripts.

This chapter showed the capabilities of MTDIFF, a new tree differencing algorithm that produces shorter edit scripts compared to previous approaches on fine-grained ASTs. It is a necessary prerequisite for the tools ARES and C3. MTDIFF also uses six general optimizations that are able to reduce the edit script size on their own and that are even compatible with other tree differencing approaches.
4 Accurate Code Recommendations

Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less.

Marie Curie

Code recommendation systems can help developers understanding more about their code. They can inform developers about potential errors, possible performance improvements, or missed systematic edits. In general, recommendation systems search inside a code base for code locations that fulfill the conditions of a pattern. If a recommendation system finds such a code location, it informs the developer about the identified code position. Often, it also gives hints about the changes that are necessary at this location. For example, such systems often name the type of error (e.g., Possible NullPointerException) they detected. More sophisticated ones provide the already changed source code as recommendation. To identify such code locations, the recommendation tools often rely on predefined patterns and search algorithms (e.g., Findbugs [25, 114]). The recommendation systems use the information in such patterns to search for suitable code locations and also to create the changed code they recommend. During the last years, a lot of research projects developed different ways to generate such patterns. As seen in Chapter 2, some approaches allow users to define patterns [34, 313]. Other approaches learn transformations from the interactions of the developer with the IDE [275], or they learn patterns from input examples. Besides the origin of the patterns, previous research efforts were focused on the optimization of precision and recall. The accuracy of the recommendation, i.e., the syntactical closeness of the recommendation to the code developers actually need, was not the main target of previous research efforts.
A recommendation for a systematic edit has a perfect accuracy of 100%, if the code of the recommendation is identical to the code that a developer would create manually for the application of the systematic edit at the recommended code location. Thus, only if it is unnecessary to adapt the recommendation in any way, it has a perfect accuracy. The more differences a recommendation has to the code of a manual execution, the less accurate it is. For example, if a developer has to change a single identifier in the recommendation before an application to the code base is possible, its accuracy is slightly lower than 100%. If a developer has to add several statements to a recommendation, the accuracy of the recommendation is lower than for a single identifier change. To measure the accuracy, the evaluation in Sec. 4.8 compares the AST nodes of the recommendation to the AST nodes of the manual applications of systematic edits. Only if both lists of AST nodes are identical, the recommendation is perfect and has an accuracy of 100%.

Addressing the accuracy of recommendations is an important aspect as a high accuracy reduces the number of manual adaptations. Fewer manual adaptations lowers the number of attention failures that can happen during this adaptation process. Such attention failures (see Sec. 1.1) occur during the manual adaptation of recommendations as this process has the same characteristics as boring tasks. Thus, a higher accuracy reduces the number of bugs developers accidentally add to a code base during the application of recommendations. Fig. 1.1 from the introduction (page 3) shows a recommendation that contains incorrect code parts. For the concrete code location, the recommendation requires two adaptation steps before it is correct and can be applied to the project. The first step is the deletion of the `assert` statement. The second step is the movement of the code after the `foo.someMethod` call into the body of the `if` statement. If developers miss one or both of these necessary adaptations and apply the incorrect recommendation directly, they add bugs to their projects. Both bugs only become apparent when the code runs and remain silent during the compilation of the code. Thus, they are hard to detect. If a test triggers the actually obsolete `assert` statement, the test results become useless. Even worse, at its original position, `foo.run` can cause a `NullPointerException` at runtime. This shows how important accurate code recommendations are.

To increase the accuracy for the introductory example and also for other systematic edits, a recommendation system has to address **three challenges:**
1. The created patterns have to conserve variances in the input examples as far as possible without becoming too specific.

2. The patterns have to express code movements.

3. The used algorithms and heuristics have to be able to handle the variances and the code movements.

The introductory example has shown the importance of challenge one and challenge two. Due to the difficult detection of code movements (see Chapter 3), challenge two is especially hard. Thus, many recommendation tools ignore code movements to simplify their patterns and algorithms. This means that they do not face challenge three. However, the handling of moves can increase the accuracy of the recommendations. As an answer to the three challenges, this chapter introduces ARES. The pattern design of ARES conserves more variations of the input examples compared to previous approaches. With the help of MTDIFF, ARES is also able to identify code movements and to include them in its patterns. The additional information in the patterns gives ARES the ability to generate more accurate code recommendations compared to other state-of-the-art approaches.

This chapter is based on previous work on pattern annotations [90] and ARES [88].

Sec. 4.1 discusses the state-of-the-art with respect to recommendation systems. It focuses on approaches that are close to ARES, i.e., that recommend source code or use examples as input. Sec. 4.2 describes the pattern design that ARES uses to address challenges one and two. Sec. 4.3 to 4.6 describe the workflow of ARES (see Fig. 4.1). The workflow starts with the Input Order Determination step that is the topic of Sec. 4.3. ARES supports two or more input examples with its workflow. As the order of the input examples influences the resulting pattern, the Input Order Determination step computes a suitable order that prevents an over-generalized pattern. A pattern is over-generalized (i.e.,

![Figure 4.1.: Workflow of ARES for one input group of code changes.](image-url)
too general), if it creates too many recommendations that are useless to developers. Such patterns reduce the precision of a recommendation tool. As the *Generalized Pattern* step only works with two changes as input, the upper row of the workflow contains a loop. In each loop iteration, the *Input Order Determination* step chooses the next input and the *Pattern Creation* merges the two input changes into one pattern. Sec. 4.4 explains this pattern creation process. The result of this process is a *Generalized Pattern* that incorporates the details of two or more input examples. ARES can use this pattern to search for applications in a code base. This is the focus of Sec. 4.5. For each identified code location, the *Recommendation Creation step* creates recommendations. Sec. 4.6 discusses this part of the workflow. After the presentation of the workflow, Sec. 4.7 discusses the complexities of its steps. This chapter concludes with an evaluation of ARES in Sec. 4.8. It compares ARES with LASE on systematic edits from the Eclipse JDT, Eclipse SWT and JUnit code repositories.

4.1 Related Work

Whereas Chapter 2 provides a general overview of static analysis tools, this section focuses on tools that are closer related to ARES. From all tools, LASE [236] is the most similar one. Like ARES, LASE uses two or more code changes as input to generate a pattern. It also searches for code locations in a code base that allow the application of the pattern. For the identified locations, LASE creates recommendations. LASE requires that the inputs are pairs of an original and a modified method. These pairs express the code changes. The actual pattern creation process differs from ARES. Initially, LASE computes the edit scripts of all input examples with CD. The CD algorithm uses a coarse-grained AST and thus, LASE cannot express changes in expressions, only changes of complete statements. This makes the recommendations from LASE’s pattern less accurate. After the computation of the edit scripts with CD, LASE merges pairs of edit scripts together with the LCS algorithm. Two edit operations may be part of the LCS if they have the same label and the bigram similarity of their textual representations is 0.6 or above. If LASE does not identify at least one common edit operation, it uses the LCS algorithm again on the input changes. On this second run, LASE replaces all types, methods and variables with abstract names. Then it executes the LCS algorithm again. If the abstract representation of two
4.1 Related Work

edit operations is identical, they can be part of the LCS in this second run. This LCS-based approach has the disadvantage that the patterns in LASE only include the edit operations that are shared by all input code changes. This can reduce the accuracy of the recommendations as it cannot cover variances in the input examples (e.g., the `assert` statement in the introductory example).

The patterns of LASE do not only contain the changed code parts, but also additional unchanged line as context. This is also true for ARES. However, the extraction of the context surrounding the code changes differs between both approaches. For the context, LASE identifies identical code that appears in all input code changes with the tool CCFinder and adds the nodes of this identical code to the context. As CCFinder works on tokens, it is unaware of the AST structure of the code. As a consequence, the identified identical code can be part of different subtrees. To include such tokens would violate the syntactical structure of the source code. Thus, LASE removes them from the context with the help of the Maximum Common Subtree Extraction (MCESE) [218]. Furthermore, LASE removes all nodes that are not part of the computed common subtrees. To further refine the edit context, LASE performs a control and data dependence analysis to determine whether the changed code and the unchanged context are related. It excludes all unchanged code parts from the context that have no data dependencies to the changed code. After this process, the context of LASE consists of a forest of ASTs. In summary, LASE is unable to handle differences among expressions in statements (e.g., call arguments) due to the use of CD and only uses edit operations that are shared by all input changes. ARES can handle such differences more accurately. However, the use of data dependencies to exclude unrelated code could also improve the results of ARES and has to be evaluated in the future.

RASE [235] uses LASE to identify code clones and replaces them with a unified method if possible. As RASE looks for refactoring opportunities, it offers no general solution to systematic edits. It is only of use if it is possible to replace similar code locations with a unified method. In this case, it can prevent some systematic edits from occurring. Cookbook by Jacobellis et al. [164] also builds on LASE and uses code changes as input. It analyzes the code changes made by developers in the IDE and compares them to the patterns that were generated with LASE in the past. If a change in the IDE is part of such a pattern, Cookbook recommends the complete change. As Cookbook uses LASE,
it has the same limitations. ARES can express more variances in the input examples and also supports expression differences. SYDIT [237] is the predecessor of LASE and only requires one input code change. It uses CD to extract the edit script of the single change and uses data dependencies to identify the context. This is similar to LASE. As the information in one example is limited, SYDIT replaces variables, methods and types with abstract identifiers to generalize its patterns. However, compared to LASE and ARES, the patterns are still very specific as the abstract versions cannot express additional statements or expressions. Thus, SYDIT can miss code locations that belong to the same systematic edit as the input example.

The approach by Kim et al. [188] builds a bug fix database consisting of code changes from code repositories. It extracts all changes that modify or delete code and that have “Fixed”, “Bug”, or a bug tracking number in the commit message. The approach divides the bug fix into smaller parts (e.g., conditions and statements). For each part, it stores the original version, a normalized original version, and the modified version in the database. With this database, the approach by Kim et al. can search for code locations in a code base that are identical to the unchanged or normalized versions of the bug fix part. It presents the modified version of the bug fix part as recommendation. Using two or more examples as input, ARES can create more general patterns that also identify code locations that have large code differences to the input examples (e.g., additional statements).

REFAZER [281] also learns program transformations from examples. For each input example, it computes the edit script with the tree differencing algorithm from Zhang and Sasha (see Sec. 3.2). Then it creates connected components from the edit operations in these edit scripts. For example, if several delete operations remove all the nodes in a subtree, REFAZER puts the operations into the same connected components. Then REFAZER uses DBSCAN (see Sec. 5.3) to identify similar connected components among the edit scripts. For each such cluster of changed components, REFAZER creates a rewrite rule in its own DSL that describes the AST transformations of a cluster. REFAZER uses PROSE [265] to process these sets of transformations. PROSE is an inductive programming framework and is able to synthesize an abstract transformation from an input set of transformations. As PROSE synthesizes different abstract transformations, REFAZER provides a filter function to reduce their number. The filtered abstract transformations
4.1 Related Work

are the patterns that REFAZER uses to search for code locations and to transform them into recommendations.

Compared to ARES, the REFAZER approach has several drawbacks. First, the current version of REFAZER is unable to learn transformations from code changes that have different operations in the edit script (e.g., the `assert` in the introductory example). Second, the DSL and the Zhang and Sasha tree differencing algorithm both do not support code movements. Thus, REFAZER cannot detect code movements in the input examples. This can lower the accuracy of the recommendations (see Sec. 1.2).

Coccinelle by Padioleau et al. [261] allows the generalization of code changes in patch format. With a generalized patch, Coccinelle identifies other code locations for which the patch is applicable and recommends the changed code to the developer. For the generalization, Padioleau et al. defined an own DSL called SmLP that enhances normal patches to semantic patches. Similar to ARES, the pattern design that Coccinelle uses is human readable and uses a DSL to describe generalizations. However, it is the task of the developer to generalize a patch with SmLP. To avoid this additional effort for developers, Andersen et al. [9, 10] introduced spdiff that creates a semantic patch from a set of code changes in patch format. To create a semantic patch, spdiff computes the largest common sub-patch that captures as many common changes in the input examples as possible. Similar to LASE this has the drawback that the pattern cannot express variances in the input examples. The pattern is also based on patches that consist of inserted and deleted lines. It is impossible to express the code movements from the introductory example with this approach. ARES can handle this movement with its pattern design. Using patches also has the disadvantage that there is less context available. ARES and LASE add unchanged code to the pattern to make them more precise.

Genesis by Long et al. [214] is an automatic program repair approach that uses examples as input and returns generated patches. The base for the patches are patterns (called transforms) that Genesis infers from the input examples. In contrast to ARES, Genesis does not generate a single transform for a set of input examples but several. Genesis also identifies generators in the input examples. These generators can add code to the patches that has no link to the original part of the pattern. This is similar to the choice annotation in ARES (see Sec. 4.2). Despite the similarities, the use case of Genesis is different from the use case of
ARES. The patterns that Genesis infers can be very general. As a consequence, it creates several patches for each code location. To filter all the patches, Genesis requires a test suite to evaluate their usefulness. Due to this filter, it can take hours to run Genesis on a complete code base for a single pattern. As the patterns in ARES are less general, they identify fewer code locations and a test suite to filter the results is unnecessary for a single pattern. However, the information about whether a recommendation does or does not pass the tests in a test suite is still valuable for developers. It is possible to add such a feature to ARES in the future.

VuRLE by Ma et al. [220] is another example-based approach that focuses on automatic program repair. Similar to Genesis, it does not generate one pattern from a set of input examples but creates several patterns. The input examples do not belong to a concrete systematic edit, but to a complete change class like “unchecked return value”. Initially, VuRLE performs a cluster analysis on the input examples and uses techniques from C3 (see Sec. 5.1 for details). For each pair of input changes that remain connected by the clustering (i.e., are members of the same cluster), VuRLE creates a pattern. Thus, it can generate several patterns for the same connected component. The actual pattern creation starts with the computation of the edit operations between the original and modified code with GT. Similar to LASE, VuRLE uses the LCS algorithm to compute the shared edit operations of two input examples that are part of the pattern. As above, the use of the LCS algorithm can reduce the accuracy of the recommendation. The patterns of VuRLE also include a context. Like LASE, VuRLE uses this context to search for code locations. To generate the context, VuRLE computes all the paths from the root node of two code changes to their respective leaves. Then it compares each path with the LCS algorithm. The shared parts form the context of the pattern. To further generalize the context, VuRLE replaces identifiers and types with placeholders. ARES is more specific and replaces identifiers only if they have different values in the input examples. As VuRLE replaces all of them, this can lead to many false positives.

Similar to other tools, the pattern design of VuRLE also cannot express arbitrary code movements. Instead, it relies on its ability to produce several patterns for a set of input examples. To avoid a huge number of recommendations for the same location, VuRLE first determines which connected component is the most similar one to a code location. For that purpose, it computes the similarity of the context of
a component to the code location and chooses the component with the highest similarity to the location. After the selection of the connected component, VuRLE sorts the context of all patterns of the connected component with respect to their similarity to the code location. Then it applies each pattern of the connected component to the code location until it finds one that does not produce redundant code (e.g., duplicate statements). It recommends the first repair without redundant code to the developer. Compared to ARES, the search and recommendation creation of VuRLE takes longer. In many cases it has to compare several contexts and has to apply several patterns to create a single recommendation. The filtering step also only avoids redundant code. In contrast to ARES, VuRLE does not optimize the accuracy of the recommendations.

FixWizard by Nguyen et al. [256] identifies code peers (i.e., similar code fragments), recognizes bug fix patterns of code peers, and recommends fixes for the affected code peers. To identify code peers, FixWizard uses a two-step approach. First, it detects code peer candidates based on their structural similarity [254] and their name similarity. Second, it computes a feature similarity between possible candidates. The feature similarity relies on the similarity of method names that appear in the candidates. Based on these results, it determines the code peers. To extract common fixes from a repository, FixWizard uses feature vectors. Sec. 5.1 discusses this step in comparison with C3. If FixWizard finds two or more similar code fixes, it adds them to a database. If a code base contains a code peer that is changed by a bug fix in the database, FixWizard recommends to apply this fix to all peers that were not changed accordingly. The recommendation consists of the code location and the list of edit operations on the statement level (including adopted variable and method names). In contrast to ARES, FixWizard only works for code peers and has no means to search for other code locations. It also does not provide complete recommendations. Instead, FixWizard recommends the code location and the list of edit operations for the transformation.

Nguyen et al. [13] presented TasC, a recommendation system that suggests statement-level source code changes for the current programming task. TasC extracts code changes from the repository by applying the JSync tree differencing algorithm on the code changes in the repository. Based on these AST changes, TasC extracts the changed tokens and normalizes them. Then TasC uses the Latent Dirichlet Allocation
Accurate Code Recommendations

(LDA) [43] on the changed tokens to create a task model. This is the base for the recommendation system. To create recommendations, TasC examines the current context in the IDE and compares the tokens in the current code fragment with tokens in its model. Based on the token probabilities in their model and the similarity of the tokens, TasC presents a ranked list with recommendations. In contrast to ARES, TasC works on the current context of the IDE and cannot recommend changes in other parts of a project's code base. The changes are also limited to statements and do not cover complete methods. Additionally, TasC cannot express code movements. If the accuracy of recommendations depends on code movements, it is possible that TasC creates recommendations that are not very accurate and thus are not helpful for the developers.

Thung et al. [304] created a recommendation system for backporting driver files in the Linux kernel. As input, the system uses the driver to be backported, the targeted older Linux version and the Linux source code archive. The system first performs a binary search on the repository and compiles the driver with the different repository versions to search for the most recent commit in which the driver causes a compilation error. As a limitation, the current system only addresses problems where a single code line triggers the error. Then the system uses GT to compare the added and deleted code lines in the commit with the line that caused the compilation error. For each possible line combination, the system computes a code transformation that has the potential to fix the error. Then it ranks all identified transformations according to their similarity to the line that caused the compilation error. In contrast to ARES, this system solves a very specific problem and includes steps (like the binary search in the repository) that are not required by a general recommendation system like ARES. The systematic edits that ARES addresses are also more general and include more than a single line of code. The recommendations of ARES also provide help for other problems like runtime exceptions or refactorings. However, a check whether a recommendation creates a compilation problem or not can be added to ARES as filter.

ChangeCommander by Fluri et al. [120] recommends the insertion of if statements around method invocations. This is often necessary to prevent bugs. Initially, ChangeCommander browses repositories to identify method invocations that were moved into an inserted if statement. This process uses CD to detect the movements. If such
4.2 Pattern Design

For accurate recommendations, the pattern design has to solve the first two challenges. It needs a way to handle variances in the input examples and it needs a method to identify and handle code movements. In both cases, the pattern design of ARES provides a solution. Each pattern consists of an original and a modified code part. Both parts are plain Java code with annotations. To avoid interference of the annotations

a movement happens several times for the same method invocation, ChangeCommander creates a pattern. ChangeCommander then uses these pattern to highlight method invocations in the IDE that frequently were moved into the body of an if. In contrast to ARES, ChangeCommander only supports the movement of method invocations and cannot handle other types of systematic edits.

Landauer and Hirakawa [201] presented a programming by demonstration (PBD) approach that allows users to automate text changes after the manual execution of one or more similar text changes. This only works for systematic edits, if the textual changes for each systematic edit are identical. ARES can handle variations like a different number of moved statements at each code location.

Robbes and Lanza [275] proposed an approach that creates a code transformation pattern from a single example. It records a code change performed by a developer in the IDE and creates a pattern from this record. A developer can further generalize and optimize the pattern in the IDE. Afterwards, the tool searches for applications of such a pattern. For each code location that allows the application of the pattern, the approach presents the modified code as recommendation. As this system relies on a single recorded change, it is less general than approaches that can use several input examples. The generalization of the pattern is also the task of developers and causes additional effort. The tool also has the disadvantage that it cannot use code changes from repositories. Developers have to record a code change in the IDE. ARES can use data from repositories to generate patterns and generalizes them automatically.

Mondal et al. [244] presented a ranking mechanism to order recommendations for the same code location. It is possible to combine this approach with ARES to filter less relevant recommendations. This is especially useful if ARES is used with several thousand patterns.
4 Accurate Code Recommendations

Original code:

```java
    // #match (original, (k)) {
    // #wildcard expr (ARES0, verbose, 1)
    this.init( verbose );
    this.shutdown();
    updateValue();

    k = 0;
    while (k < 10) {
        // #wildcard stmt (ARES1 );
        k++;
    }
    // #wildcard stmt (ARES2 );
    foo.someMethod (42);

    // # } }
```

Modified code:

```java
    // #match (modified) {
    // #use (ARES0, verbose, 1)
    this.init( verbose );
    updateValue();

    // #choice {
    // #case {
        assert (this.value == 23);
    // # }
    // # }
    if (foo != null) {
        foo.someMethod(42);
    }

    // #choice {
    // #case {
        System.out.print(foo );
    // # }
    // #case {
        this.print(foo );
    // # }
    // # }
    // # }
    this.shutdown();
    // # }
```

Figure 4.2.: Generalized pattern.

with other analysis tools and compilers, the annotations are simply line comments that start with “//#”. Using plain Java code and annotations makes it possible to refine patterns manually if desired. As done in previous work [90], it is also possible to write such patterns from scratch without input examples.

Fig. 4.2 shows a pattern that contains all annotation types that ARES generates from input code changes. The annotations in Fig. 4.2 are displayed in blue. Fig. 4.3 contains the corresponding formal grammar of the ARES annotations. The `match` annotation surrounds the pattern on the original and on the modified side and specifies the boundaries of the pattern. Using such a boundary allows the pattern to contain additional information that is not part of the token-based code location search. In the future, this will make it possible to perform additional checks (e.g., type checks) on the pattern. To make the inspection and the refinement of the patterns easier for developers, the `match` annotations contain the `AresMatchTypeIdentifier` (original or modified) that marks the original and the modified parts. The most important part of the `match` annotation is the `AresIdentifierList`. It contains all identifiers for which
4.2 Pattern Design

Statement:
StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement
AresStatement

AresStatement:
AresMatchStatement
AresWildcardStatement
AresUseStatement
AresChoiceStatement

AresToken:
'//#'

AresMatchTypeIdentifier:
original
modified

AresIdentifierList:
Identifier {, AresIdentifierList}

AresMatchType:
'(' AresMatchTypeIdentifier [ ',', '(' AresIdentifierList ')' ] ')'

AresMatchStatement:
AresToken match AresMatchType '{' BlockStatements AresToken '}'

AresWildcardStatement:
AresToken wildcard Identifier
'(' Identifier [, Expression , IntegerLiteral ] ')' [;]

AresUseStatement:
AresToken use
'(' Identifier [, Expression , IntegerLiteral ] ')' [;]

AresChoiceStatement:
AresToken choice '{' AresCaseList AresToken '}'

AresCaseList:
AresCaseStatement {, AresCaseStatement}

AresCaseStatement:
AresToken case '{' BlockStatements AresToken '}'

Figure 4.3.: ARES grammar.

ARES does not need to find an exact match (equal label and equal value) during the search for pattern applications (see Sec. 4.5). This has the advantage that it makes the pattern more general and allows it to be applicable to more code locations even if they have different identifiers.
Using the *wildcard* annotation (described below) for identifiers would be too verbose and make the patterns more complicated to read. For example, the *AresIdentifierList* in Fig. 4.2 contains the identifier \( k \). If ARES searches for matching AST tokens to \( k \), arbitrary identifiers are valid. In contrast to \( k \), the *init* identifier (line 3) is not part of the *match* annotation. As a consequence, a code location has to contain an identifier with *init* at the appropriate position during the search process.

To improve the accuracy of the recommendations, ARES replaces identifiers in the *AresIdentifierList* with the identifiers from the code location. This is possible with the help of MTDIFF (see Sec. 4.6). Other approaches like LASE often cannot infer the correct identifiers for the recommendations from the input changes and thus add placeholders (e.g., \( v0 \)). The recommendations of ARES are more accurate (see Sec. 4.6).

The *wildcard* annotations are the main tool of ARES for the pattern generalization. ARES currently uses two *wildcard* types (*expr* and *stmt*) distinguished by the identifier that follows the *wildcard* keyword. The *stmt-wildcard* consumes none or an arbitrary number of statements during the search for possible pattern applications. This means that a code location matches the pattern in Fig. 4.2 even if it has additional statements before *foo.someMethod* (line 16). ARES also uses *stmt-wildcards* to express differences in the input examples. For LASE, the *assert* statement in the introductory example poses a problem whereas ARES can handle it with a *stmt-wildcard*. As this *wildcard* type can match none or an arbitrary number of statements, it can match an *assert* before *foo.someMethod* at any examined code location.

The *wildcards* tagged with *expr* generalize the pattern on a finer level. They specify on which part of a statement none or (if applicable) an arbitrary number of expressions may occur. For example, the *wildcard* in line 2 of Fig. 4.2 specifies that at the first occurrence of *verbose* none or an arbitrary number of method arguments can exist at the code location. Thus, the pattern matches the code location even if *init* has none or more arguments. As the expression *verbose* can occur more than once in a statement, the *IntegerLiteral* after the expression (\( i \) in the example) specifies exactly which of the alternative expressions belong to the *wildcard* annotation. Whether the *expr-wildcard* may match none or an arbitrary number of expressions depends on the position of the expression in the AST. As *verbose* is part of an argument list of a method call, its associated *wildcard* can match none, one or an arbitrary number
of expressions. If the wildcard would use $expr(A1, this, 1)$ on the same statement, only a single (and not none) expression is a valid match. To increase the flexibility of the pattern design, several expr-wildcards can address the same statement. In general, each expr-wildcard always addresses the next non-wildcard statement in a code block.

In the ARES grammar, stmt and expr are only Identifiers, not keywords. This allows the handling of wildcards with a plugin system. It is possible to extend this plugin system to support other wildcard types. For example, the previous work on ARES [90] uses an effectFree-wildcard. This wildcard only matches expressions that have no side-effects at their execution. Whereas the expr and stmt plugins only work on the AST, the effectFree plugin also includes a static analysis of the code. Although the plugin system offers such additional options for patterns, ARES currently only generates wildcard annotations that use the stmt or expr plugins. Using more sophisticated plugins (e.g., the effect-Free plugin) would require additional analyses in ARES during the pattern creation. This is part of future work. The grammar in Fig. 4.3 allows a semicolon at the end of a wildcard. In fact, ARES adds a semicolon to stmt-wildcards and does not add one to expr-wildcards. The sole purpose of the semicolon is to emphasize for a human reader that the expr-wildcard addresses a following statement. Internally, ARES simply ignores the semicolon.

Whereas the original part of the pattern contains wildcards, the modified part contains use annotations. The use annotations specify where the Recommendation Creation step (see Sec. 4.6) has to add the code matched by the corresponding wildcard during the search for code locations. The corresponding wildcard to a use annotation is the wildcard with the same name. For example, the use in line 2 has the name $ARES0$ and thus the corresponding wildcard is the one in line 2 of the original part. As the use that corresponds to a wildcard can appear anywhere in the modified code, ARES supports the movement of arbitrary code. It is unnecessary to specify in the pattern how many statements or expressions are moved to a new location. This solves the problem of the moved $foo.run$ from the introductory example and increases the accuracy of the recommendations for such changes. As the wildcard in line 15 has no corresponding use annotation, ARES simply deletes the code matched by it. This design allows ARES to express deletes of one or more code lines. In the case of the introductory example, this can take care of the assert statement.
The combination of the wildcard and use annotations expresses code transformations that other approaches cannot handle. For example, LASE only uses the edit operations that are shared by all input examples. It cannot execute a delete operation that only occurs in one example. Thus, LASE keeps statements like the assert from Fig. 1.1 in the recommendation. ARES creates more accurate recommendations as the annotations can express such edit operations.

The pattern does not only encode movements of code matched by wildcards, it also encodes the movements of other code parts. For example, the pattern moves the statement foo.someMethod(42) into an if statement (lines 16 to 14 in Fig. 4.2). As the pattern encodes the movement and not an insert, ARES supports different constants inside the argument list of foo.someMethod. Thus, if a code location uses the statement foo.someMethod(99), the recommendation also contains the literal 99. Approaches that do not support code movements (e.g., REFAZER) are forced to add 42, as the pattern does not contain a move of the statement, but an insert operation of 42. The ability of ARES to include differing constants like 99 also leads to more accurate recommendations.

With wildcard and use annotations, ARES can express variations in the original parts of the input examples. However, it is also possible that the modified parts of the examples contain different inserted statements. For example, one code change adds System.out.print, the second one adds this.print. As both inserted statements have no corresponding code on the original side of the examples, there are no corresponding wildcard annotations in the original part of the pattern. To address this issue, ARES uses the choice annotation. If a pattern contains choice annotations, ARES creates several recommendation versions for each code location and shows them as variants of the same recommendation to the developer. Each variant contains a different combination of inserted statements (see Sec. 4.6). This increases the accuracy of the recommendation even further as it adds the System.out.print or the this.print statement. Both are present in the modified parts of input examples.

The match, wildcard, and use annotations handle variations in the original code of the input examples. The choice annotation is even able to handle variations in the inserted statements of the modified part. In combination, these four annotations solve challenge one.
As the wildcard and use annotations can have different positions, ARES can express movements of arbitrary code with its pattern design. Non-wildcard code can also have different positions in the original and the modified pattern parts. Thus, the patterns can also express specific code movements. In combination, this solves the second challenge, the accurate expression of code movements in the pattern. Thereby, the pattern design of ARES already solves the problems of other recommendation approaches like LASE. However, creating such a pattern based on input examples is complicated. Thus, ARES still has to solve challenge three, i.e., the use of algorithms that handle such a pattern design. The following two sections discuss this in detail.

4.3 Input Order Determination

As the Pattern Creation step makes heavy use of heuristics, the results depend on the order of the input examples. A wrong order can lead to very general patterns. To have consistent results for the same set of examples, a technique that imposes a consistent order and produces the same pattern for the same set of examples is desirable. As it is hard for developers to gauge the consequences of a certain input order, relying on their input is not an option. Additionally, the goal of this thesis is a recommendation system that requires little effort to use. If the input code changes come from other tools (e.g., C3), it is also unlikely that the order has been optimized for ARES beforehand. Consequently, it is the task of ARES to determine a suitable input order.

In each iteration of the workflow (see Fig. 4.1) the Input Order Determination step in ARES chooses the code change that has the highest similarity to the current generalized pattern. In the first iteration there is no current generalized pattern available. Instead, ARES chooses the two code changes that have the highest average similarity to each other and to all other input examples. To get a similarity value for two code changes, ARES uses MTDIFF to compute the edit operations between the original parts of both code changes and the edit operations between the two modified parts. The higher the sum of the number of edit operations of both parts the lower is the similarity value. Thus, the smaller the number of edit operations the more likely it is that ARES chooses the corresponding code change pair.

This selection scheme has two advantages. First, it is possible to stop the iteration process at different generalization levels. Second, it
allows a less complex rule-based system in the Pattern Creation step. Stopping the iteration process at a certain level is desirable, as some input code changes can reduce the complete pattern to a single wildcard in the Pattern Creation step (also called over-generalization below). Such a pattern is useless as it matches all code in a project. As the Input Order Determination step chooses the pattern with the highest similarity in each iteration, it steadily increases the generality of the pattern. If one iteration creates a single wildcard and thus leads to an over-generalization, ARES could select the pattern from the previous iteration. However, the evaluation in this thesis does not use this feature. The second advantage of this selection scheme is that a higher similarity also means that there are fewer differences between the pattern and the code change and thus fewer edit operations that express the differences. This can avoid complex sets of edit operations that are not covered by the current rule-based system of the Pattern Creation step. As discussed by Mandl [223], a rule-based system cannot cover all potential use cases if it was not designed and tested for the complete input data set. As this data set is nearly unlimited in size due to the variety of source code changes, creating all necessary rules manually is impossible. Thus, reducing the number of edit operations by selecting only very similar code changes decreases the number of complex edit scripts and thus reduces the likelihood that the rule-based system produces an over-generalization due to missing rules. However, even with this suitable selection scheme, the current rule-based system in ARES requires over 50 manually implemented rules (see https://github.com/FAU-Inf2/ARES).

To explain the order determination, Table 4.1 holds a set of artificial similarity values for the code changes $c_A \ldots c_D$ (the smaller the better). Initially, ARES selects the pair that has the lowest total number of edit operations between itself and all other code changes. Thus, the first iteration consists of $c_A$ and $c_B$ in the example (9 and 8 edit operations). Then ARES computes the similarities between $c_C$, $c_D$ and the generalized pattern $gp$ from the first iteration (see Table 4.2). As $c_C$ has the lowest total number of edit operations, ARES uses the pair ($c_C$, $gp$) in the next iteration, and it uses $c_D$ in the final iteration.

Similar to other parts of ARES, the Input Order Determination step also contributes to the accuracy of the recommendations. It lowers the number of edit operations and thus reduces the likelihood that a complex edit operation set is not fully covered by the rule-based system.
4.4 Pattern Creation

From the Input Order Determination step, the Pattern Creation step receives the two code changes \( c_\alpha \) and \( c_\beta \) in the first iteration and \( c_\gamma \) in the second iteration. ARES uses these three code changes to create the pattern in Fig. 4.2. Fig. 4.4 contains the code changes of the first iterations. The coloring is based on the output of MTDIFF. However, in contrast to Chapter 3, this chapter combines smaller AST changes (like the insert of the identifier `print`) into a single change (like `this.print`) for better readability. The following sections focus on the first iteration, while Sec. 4.4.7 discusses the second iteration that combines \( c_\gamma \) with the pattern from the previous iteration to create the pattern in Fig. 4.2. The section uses this example to illustrate the differences between the first iteration and the second iteration in the Pattern Creation step. All further iterations are similar to the second one.

For the generalized pattern in the first iteration, ARES has to determine the annotations based on the differences in the input examples. To identify the position and type of the wildcard annotations, ARES uses...
MTDIFF to determine the edit operations that transform the original part $o_\alpha$ of change $c_\alpha$ into the original part $o_\beta$ of change $c_\beta$. Fig. 4.5(a) contains a graphical representation of the differences. In general, ARES creates a wildcard annotation for each edit operation. Similarly, ARES
4.4 Pattern Creation

```java
01: { 
02:   d = 1.0;
03:   this.init(true);
04:   updateValue();
05:   this.shutdown();
06:   j = 0;
07:   while (j < 10) {
08:     String tmp = "bar";
09:     this.init(placeholder, tmp);
10:     j++;
11:   }
12:   assert (foo != null);
13:   foo.someMethod(23);
14: }
15: try {
16:   this.i = 5;
17:   this.init(placeholder);
18:   updateValue();
19:   printValue(placeholder); 
20: } 
```

(a) Differences between the original parts of the code changes $c_\alpha$ and $c_\beta$.

```java
21:   for (j = 0; j < 10; j++) {
22:     String tmp = "bar";
23:     this.init(placeholder, tmp);
24:     j++;
25:   }
26:   if (foo != null) {
27:     foo.someMethod(23);
28:   }
29:   this.print(foo);
30:   this.shutdown();
31: }
32: catch (Exception e) {
33: }
```

(b) Differences between the modified parts of the code changes $c_\alpha$ and $c_\beta$.

Figure 4.5: Differences between the original and modified parts of the code changes $c_\alpha$ and $c_\beta$. Same legend as in Fig. 3.2.

uses the operations that transform the modified part $m_\alpha$ of change $c_\alpha$ into the part $m_\beta$ (see Fig. 4.5(b)) to create use or choice annotations.

The pattern creation process uses six sub-steps in every iteration. The first step (Change Isolation) determines the actual size of the pattern and thus the scope of the match annotations. The second step,
the Edit Script Adjustment is responsible for the balance between over-fitting and over-generalization. An over-fitted pattern only matches the input examples but does not patch any other code locations. This does not offer help for systematic edits as it does not create new recommendations for developers. In the worst case, an over-generalized pattern is a single wildcard. This is useless for developers as such a pattern would match any code location and generate thousands of useless recommendations. To balance the pattern between these extremes, ARES uses a rule-based system that filters and adjusts edit scripts. ARES applies this rule-based system to the edit scripts that MTDIFF generates from the differences between $o_\alpha$ and $o_\beta$ and also to the edit script that MTDIFF generates between $m_\alpha$ and $m_\beta$. After the filtering and adjustment step, ARES inserts the match annotation in the Match Insertion step. In the Wildcard and Use Insertion step, ARES adds the wildcard and use annotations. To determine the names that specify the connections between wildcards and uses, ARES uses MTDIFF in the Wildcard Name Assignment step. The mapping between $o_\alpha$ and $m_\alpha$ and the mapping between $o_\beta$ and $m_\beta$ are additional inputs for this step. For a correct name assignment based on these mappings, an accurate detection of moves is essential. Otherwise, ARES is unable to identify the correct connections between the annotations. Therefore, ARES requires MTDIFF. The last sub-step of the Pattern Creation is the insertion of choice annotations.

4.4.1 Change Isolation

Similar to LASE, the patterns of ARES include unchanged code that surrounds the changed code parts. This is often called the context [236]. Including the surrounding code in the pattern is important as it makes the pattern more precise and avoids the creation of useless recommendations. However, including all the surrounding code of $c_\alpha$ and $c_\beta$ for the following steps in the Pattern Creation process is not always possible. Due the heuristics in ARES, including too much surrounding code can lead to an over-general pattern. For example, the method body of change $c_\beta$ contains a try-node whereas $c_\alpha$ does not. As this difference appears in the edit script of $(o_\alpha, o_\beta)$, ARES would add a wildcard for the try statement. This would reduce the pattern to a single wildcard and render it useless. Thus, before the next step, ARES has to remove surrounding code that would lead to over-general patterns.

144
For that purpose, the *Change Isolation* step uses two phases. The first phase marks all nodes that are affected by edit operations in the edit script for the pair \((o_\alpha, m_\alpha)\). It also marks all affected nodes in the edit script for the pair \((o_\beta, m_\beta)\). Then, for each of the four ASTs, ARES climbs bottom-up through the tree to identify the node (called change-root below) that is farthest away from the root and is still parent to all marked (i.e., changed) nodes. ARES always excludes all code that is not inside the subtrees of these four nodes. For the example in Fig. 4.4, the change root of \(o_\alpha\) is the complete method body and thus the code in lines 1–20. For \(m_\alpha\) it is also the complete method body. For \(o_\beta\) and \(m_\beta\) it is the *try* statement. In most cases, the computation and use of these change-roots is sufficient and the *Pattern Creation* step returns a pattern that contains at least one statement that is not a wildcard. If this selection still leads to an over-generalization, ARES executes phase 2 of the *Change Isolation* step. This phase uses several heuristics to avoid an over-generalized pattern. It tries out a combination of the heuristics until the first one leads to a pattern that is not over-generalized. The combinations in their execution order are:

1. Use the mapping from MTDIFF between \(o_\alpha\) and \(o_\beta\) to determine alternative change roots. If a direct child \(dch_1\) of the original change root in \(o_\alpha\) is similar to a child \(ch_2\) of the original change root in \(o_\beta\), use this pair as new change roots. The two nodes \(dch_1\) and \(ch_2\) are similar if they have common children. This means that there are pairs of children from \(ch_1\) and \(ch_2\) in the mapping. The new change roots \((dch_1, ch_2)\) are only valid selections if there is a partner to \(dch_1\) in the mapping from \(o_\alpha\) to \(m_\alpha\) and if there is also a partner to \(ch_2\) in the mapping from \(o_\beta\) to \(m_\beta\). This is necessary to determine the full set of four change-roots for the following steps. The change-roots also have to cover at least one change from \(o_\alpha\) to \(m_\alpha\), their counterparts at least one change from \(o_\beta\) to \(m_\beta\). This condition ensures that the patterns contain at least one edit operation from the input code changes.

2. Revert the role of \(o_\alpha\) and \(o_\beta\) and execute the first part of phase 2 again. This second search for change-roots uses the direct children of the change root in \(o_\beta\) as source. This covers the cases in which the original code \(o_\beta\) has an additional statement (like the *try* in Fig. 4.4).
3. Use the direct grandchildren instead of the direct children and try the previous steps of phase 2 again. This addresses cases in which one of the input examples has two additional nodes surrounding the important parts of the code change.

An application of these heuristics on the example in Fig. 4.4 shows that phase 1 on its own is insufficient due to the different labels of the change-roots. The change-roots for $o_\alpha$ and $m_\alpha$ are code blocks, the change-roots for $o_\beta$ and $m_\beta$ are try statements. Thus, the edit script between $o_\alpha$ and $o_\beta$ contains an insert operation for the try statement. This edit operation leads to a replacement of the complete code block with a single wildcard in the following Pattern Creation steps (see Sec. 4.4.4).

Phase 2 targets such problems. The first part of phase 2 examines all direct children from $o_\alpha$ and thus all statements from lines 2–14 in the example. The restriction that change-roots have to cover a change between $o_\alpha$ and $m_\alpha$ excludes most statements. Only the assert and while statement are possible options. But they do not fulfill all conditions. Both have no partners in the mapping from $o_\alpha$ to $m_\alpha$. Since there is no solution to be found in this part of phase 2, the Change Isolation continues with its second part.

For the example, part two of phase 2 gives suitable change-roots that fulfill all conditions and do not lead to an over-generalization. The block node that represents the body of the try statement (lines 1–18 in Fig. 4.4(b)) is a suitable change-root. It is mapped to the block in lines 1–20 of $o_\alpha$. The same applies to the try statement in $m_\beta$. The change roots also fulfill the last condition. They cover most of the of the changes between the original and the modified code parts. In fact, except for the change of the catch block in $c_\beta$, they cover all changes.

The missing change in the catch block is also an example that illustrates the limitations of this step. If ARES has to use the heuristics in phase 2, the pattern no longer covers all edit operations of the systematic edit. This reduces the accuracy of the recommendation. However, a correct recommendation with lower accuracy is still better than no recommendation at all. In future versions of ARES, it would be possible to break with the pattern design to handle this in a different way. AST-based data structures can express such variations. A disadvantage of such data structures is that they grow large and complicated. This makes it difficult for developers to understand them [136].
alternative is to use several original and modified code parts to express a single pattern. As one of the principal goals of this thesis is the creation of patterns that are readable for developers, this thesis does not explore these two possibilities.

4.4.2 Edit Script Adjustment

This step is the technical core of ARES. It uses the edit scripts from MTDIFF and the change-roots from the previous step as inputs. With this information, ARES balances the pattern between over-fitting and over-generalization. Over-fitting means that the pattern only matches the input examples and does not identify additional code locations that require the execution of the systematic edit. Over-generalization means that the pattern contains too many annotations and too few non-variable code statements and expressions. Thus, it fits too many code locations that are unrelated to the systematic edit. Another disadvantage of general patterns is their lower accuracy. If a pattern is more general it contains more wildcard annotations. Sometimes the Pattern Creation process replaces complete subtrees with such wildcards. This renders all edit operations that affect nodes inside these subtrees invalid as the nodes they affect are no longer present in the ASTs of the pattern. Each such invalid edit operation lowers the accuracy of the resulting pattern as these edit operations are part of the transformation from the systematic edit.

The following parts of the Pattern Creation process add annotations for each edit operation in the edit script from \( o_\alpha \) to \( o_\beta \) and in the edit script from \( m_\alpha \) to \( m_\beta \). To achieve a balance, the Edit Script Adjustment step removes some of the edit operations in the edit scripts or replaces them with artificial created operations. As the edit scripts reflect the complexity of the underlying code, there are countless possible combinations of edit operations that require different adjustments. Due to this complexity, ARES uses a rule-based system with over 50 rules. A complete overview over all rules is publicly available (see https://github.com/FAU-Inf2/ARES). This section describes only the rules that are relevant for the example in Fig. 4.4. Although the current rules cover thousands of examples (see Sec. 5.5.4), they cannot cover all existing combinations [223].

In general, the rules have to address two different issues. Both are related to the edit scripts on fine-grained ASTs that tree differencing algorithms generate. The first issue is the granularity of the edit
operations. On the one hand, the fine-grained edit script allows expr-wildcards that lead to more accurate recommendations compared to LASE which only works on the statement level. On the other hand, the fine-grained edit operation can lead to over-fitting and the resulting patterns only work for the input examples. Line 2 of the code changes in Fig. 4.4 illustrates this problem. To reduce the size of the edit script, a tree differencing algorithm keeps the assignment for this line although both the left-hand and the right-hand sides are different. The left-hand sides of the assignments hold a single identifier \( d \) in \( c_\alpha \) and a field access \( \text{this}.i \) in \( c_\beta \). The right-hand sides hold a floating point \( 1.0 \) and an integer \( 5 \) literal. A pattern without filter step would begin with an assignment and two expr-wildcards annotations for the left-hand and the right-hand side of the assignment. Such a pattern would not match a code location that starts with the declaration \( \text{float } d = 1.0; \). Thus, its applicability is very narrow.

To avoid this type over-fitting, ARES replaces the edit operations that affect (nearly) all expressions of a statement with an edit operation that affects the whole statement. In Fig. 4.4 this is relevant for the first statement in line 2. As the assignments have only the label in common, a pattern starting with an assignment would be too specific. Rule #48 (see Fig. 4.6) of the rule-based system handles this case. The rule iterates over each assignment pair in the mapping (lines 3–10). For each pair, it examines \( a_{\alpha\alpha} \) and checks whether a delete operation affects the left-hand side of \( a_{\alpha\alpha} \) and whether another one affects the right-hand side of \( a_{\alpha\alpha} \) (line 6). If this is the case, the rule also checks whether two insert operations cover the left- and right-hand side of \( a_{\beta\beta} \) (line 7). If both conditions are true, the rule removes all edit operations that cover the expressions of the two assignments. For example, this removes the insert operation of the field access \( \text{this}.i \) and also the insert operations.
4.4 Pattern Creation

of its children (this, i). Then rule #48 adds a single edit operation for the assignment in $c_β$. Adding a delete operation for the assignment in $c_α$ is unnecessary as both assignments are in the same position and the delete operation of the assignment would only create the same wildcard at the same position (see Sec 4.4.4).

The second issue concerns edit operations that over-generalize the pattern. The main cause of such operations is found in the heuristics of tree differencing algorithms. In many cases they do not lead to an optimal solution. This issue led to the development of the general optimizations and MTDIFF (see Chapter 3). However, even MTDIFF does not produce an optimal edit script for all changes. Additionally, MTDIFF only focuses on the edit script size and is unaware of the special needs of ARES. Therefore, ARES has to address this issue with additional rules. Still, the edit scripts of MTDIFF are shorter than those of other approaches and thus fewer rules are required.

Rule #31 (see Fig. 4.7) handles one of these cases by examining moved method calls. If two similar but not identical method calls appear in two nested blocks, MTDIFF creates pairs of inner nodes for the mapping based on the similarity of their leaves. This can create unnecessary move operations. For example, due to the argument verbose in
Fig. 4.5(a), MTDIFF moves the call to \textit{init} from line 9 in \(o_\alpha\) to line 3 in \(o_\beta\). This would cause two wildcards. As this would lead to a too general pattern, ARES detects such movements (lines 4–17 in Fig. 4.7) and changes the edit script. It replaces the \textit{move} of the call from line 9 to line 3 in Fig. 4.5(a) with a \textit{delete} operation in line 9 of \(o_\alpha\) and an \textit{insert} of \textit{verbose} in line 3 of \(o_\beta\).

Like other tree differencing algorithms, MTDIFF also generates edit operations for each node in a deleted or moved AST. This creates \textit{expr-wildcards} in addition to the \textit{wildcards} for the parent statements. Such \textit{expr-wildcards} are useless as ARES will replace the statement they affect with a \textit{stmt-wildcard} in the pattern. To prevent such useless \textit{expr-wildcards}, ARES uses several rules (\#42–\#46) to adjust the edit script. For each \textit{deleted} or \textit{inserted} node, the rules examine whether their parent is also \textit{deleted} or \textit{inserted}. In Fig. 4.5(a), this happens for the statements in lines 8, 9 and 12. Thus, instead of keeping the \textit{delete} operation for the leaves \textit{foo} and \textit{null} (line 12 in Fig. 4.5(a)), rule \#42 removes them (see Fig. 4.8).

### 4.4.3 Match Insertion

This step adds the \textit{match} annotations that surround the patterns. If the change roots from \(o_\beta\) and \(m_\beta\) are code blocks, the \textit{Match Insertion} step copies all statements from these blocks into the bodies of the \textit{match} annotations. This is the case for the example in Fig. 4.4. In all other cases, ARES moves the change roots directly into the body of the \textit{match} annotation. Besides specifying the pattern border, the \textit{match} annotation can also hold a list of identifiers. During the search for pattern applications (see Sec. 4.4.7), ARES makes use of this list. The search compares AST tokens from the pattern with tokens from the code location. If the pattern token is an identifier and this identifier is part of the \textit{match} identifier list, it can match any identifier token at the code location. If an identifier is not part of this list, their values (i.e., names) have to be identical. ARES uses the \textit{match} annotation instead of \textit{expr-wildcards} for identifiers to increase the readability of the pattern. As identifier differences are common, using \textit{wildcards} could cause many annotations, even for small patterns.

To create the identifier list, ARES examines all \textit{update} operations in the mapping between \(o_\alpha\) and \(o_\beta\). In the example in Fig. 4.4, there are only \textit{updates} that change the identifier \(j\) to \(k\). As ARES uses the right input example as pattern base, it adds the identifier \(k\) to the \textit{match} annotation.
4.4 Pattern Creation

```plaintext
01: function RULE42(editOps, mapping)
02:     for e ∈ editOps do
03:         if type(e) ≠ DELETE then
04:             continue
05:         n ← getNode(e)
06:         p_o ← parent(n)
07:         e ← getEditOperation(p_o, editOps)
08:         if e = null ∨ type(e) ≠ DELETE then
09:             continue
10:         removeOpForNode(n, editOps)
```

Figure 4.8.: Simplified version of Rule #42.

and removes the corresponding update operations from the edit script between $o_α$ and $o_β$ and from the edit script between $m_α$ and $m_β$. This has the effect that during the search for applications, ARES can accept any identifiers at the positions of $k$ in the pattern. As a consequence, the match list makes patterns more general. Whereas expr-wildcards only affect the following statement, the match identifier list affects the complete pattern. Thus, even if MTDIFF creates only one update for an identifier, it still has an effect on all occurrences of the identifier in the pattern. It also does not make a difference whether the update affects a method name, a field name, a parameter name, or a local variable name. In some cases, this can have a negative impact on the precision of ARES. However, as the precision is still at a high level (see Sec. 4.8.2) the impact is small and does not justify a more complicated pattern design.

For the Match Insertion the support of move operations in the edit script also holds an advantage. Alternatives like RTED often generate update operations instead of moves (e.g., lines 1–2 in Fig. A.3(a)). This would add the identifiers `stmtA` and `moveA` to the match annotation although this is unnecessary. The detection of moves reduces these update operations and thus reduces the identifier list on the match annotation. For such changes, MTDIFF reduces the generality of the pattern.

### 4.4.4 Wildcard and Use Insertion

After the adjustment of the edit script and the processing of update operations in the previous sections, this section deals with the wildcard and use annotations that further generalize the pattern. For each insert operation that remains in the edit script after the adjustment step, ARES adds a wildcard annotation. If the insert operation adds
Accurate Code Recommendations

a statement in $o_\beta$, ARES replaces the statement with a stmt-wildcard. If an insert operation adds an expression, ARES adds an expr-wildcard before the statement that contains the expression. A replacement of a code part makes no sense for expr-wildcard as they affect the following non-wildcard statement.

For each statement $s_1$ in $o_\alpha$ that is part of a delete operation, ARES also adds a wildcard in $o_\beta$. In contrast to wildcards for insert operations, this is more complicated as the edit operation does not specify the position of the deleted statement in relation to the code in $o_\beta$. The delete operation only defines the position of the deleted node in $o_\alpha$. This leaves the exact position of the wildcard unclear. To identify the correct position for the wildcard in the code fragment $o_\beta$, ARES uses the positions of the statements that surround $s_1$ and that are not part of an edit operation. For example, if $s_0$ is the statement in $o_\alpha$ before the deleted statement $s_1$ and $s_0$ is unchanged, ARES adds the wildcard after $s_0$ in $o_\beta$. However, these heuristics can fail if the edit script moves $s_0$ to a position below $s_2$. It is no longer possible that $s_1$ is positioned after $s_0$ but before $s_2$ in $o_\beta$. For such cases, ARES computes the most likely position of $s_1$ based on the edit script after the adjustment step, i.e., it examines the moves in the edit script for that purpose. If a move of $s_0$ below $s_2$ happens in an edit script, the heuristics place the wildcard for $s_1$ directly before the statement $s_2$. However, such heuristics cannot cover all cases. As a consequence, the heuristics sometimes add the wildcard annotation at an incorrect position, i.e., a position that prevents the pattern from being applicable to the input examples. If the heuristics in ARES use such wrong position, the pattern also leads to less accurate recommendations. However, the high accuracy values in the evaluation show that such incorrect positions are a rare occurrence.

The handling of deleted expressions is easier. For a deleted expression in $o_\alpha$ that affects a statement $s_1$, ARES adds an expr-wildcard in $o_\beta$ before the statement that forms a pair in the mapping with $s_1$.

For edit operations that affect the modified methods $m_\alpha$ and $m_\beta$ of the input examples ARES creates use annotations. Except for the created annotation type, it handles insert and delete operations in the same way as it handles them for the original methods. If they affect statements, ARES replaces the statements with a use annotation. If the insert and delete operations affect expressions, ARES adds a use annotation before the corresponding statement. Throughout the generation of annotations, ARES keeps track of the statements that have
been replaced by either wildcard or use annotations. It also memorizes the expressions that the expr annotations affect. This data is necessary for the following name assignment step in Sec. 4.4.5.

ARES also has to insert wildcard and use annotations for the move operations in the adjusted edit script. However, in contrast to other steps during the pattern creation, a special treatment of code movements does not offer a benefit for the annotation insertion. Instead, ARES makes use of the fact that a move originates from a node in the original part that is mapped to a node in the modified part. Thus, it is possible to express a move operation with a delete operation of the node in the original part and an insert operation of the node in the modified part. ARES uses this fact and replaces each move in the edit script from \( o_\alpha \) to \( o_\beta \) with a delete and insert operation for this step. This also happens for moves in the edit script from \( m_\alpha \) to \( m_\beta \). Afterwards, ARES adds annotations based on the delete and insert operations. As other parts in the Pattern Creation step handle update operations, this step simply ignores them.

The following paragraphs illustrate the effects of the Wildcard and Use Insertion step by applying it to the example in Fig. 4.4. After the adjustment step, the edit script of the original code parts from Fig. 4.4 contains the insert operation of verbose. For this insert ARES adds the expr-wildcard in line 2 of Fig. 4.2. The verbose on the added wildcard identifies the expression that is part of the insert operation and also the expression that the wildcard affects. As the insert operation replaces the first verbose expression of the following statement, ARES adds 1 to the wildcard annotation to specify that it targets the first occurrence of verbose. The identifier ARES0 is the name of the wildcard. ARES adds such names in the following step. There is also an insert of verbose in the edit script from \( m_\alpha \) to \( m_\beta \). For this insert operation, ARES adds the use in the modified part of Fig. 4.2.

Besides the insert of the verbose expression, the edit script of the example also contains operations that affect statements. ARES replaces the statements with wildcard and use annotations. From top to bottom, the first is the insert of the assignment in line 2 of the input examples in Fig. 4.4. Rule #48 generated this operation during the Edit Script Adjustment step. As the insert operation affects a statement, ARES replaces it with a stmt-wildcard. There is also a corresponding insert on the modified side. ARES replaces this statement with a use annotation.
The Wildcard and Use Insertion step also addresses adjacent wildcards. In the running example, ARES replaces the `updateValue` call in line 8 and the `printValue` call in line 9 with two stmt-wildcards. As both are stmt-wildcards, ARES combines them into a single stmt-wildcard annotation. For expr-wildcards this only happens if they affect the same expression. This can occur if a `delete` and an `insert` operation lead to the same expr-wildcard.

Adding wildcard annotations for the `delete` operations of the `tmp` declaration and the `init` call in lines 8 and 9 of $o_\alpha$ is more difficult. To determine the correct position, ARES takes the unchanged code, the position of the deleted nodes, and the edit script into account. Both, the declaration of `tmp` and the `init` call are before the increment $j++$ in $o_\alpha$. As the pair $(j++, k++)$ is part of the mapping produced by MTDIFF, ARES is able to determine that the wildcards for the deleted nodes have to be inserted before $k++$ in $o_\beta$. As the deleted nodes are the first two nodes in the code block, there is no additional information available. Thus, ARES just adds both wildcards before the increment $k++$. Afterwards, ARES combines all adjacent wildcards. This leads to the single wildcard in line 12 in the pattern (see Fig. 4.2). For the corresponding edit operations in the edit script from $m_\alpha$ to $m_\beta$ (e.g., the deleted declaration of `tmp` in $m_\alpha$), ARES creates use annotations and also combines adjacent uses into a single one.

The final `delete` operation in the edit script of the original parts is the `assert` in line 12 of $o_\alpha$. To identify its correct position, ARES examines the statements before and after the deleted `assert`. As its position is after the `while`-loop and before the `foo.someMethod` call, ARES adds the wildcard between those statements in $o_\beta$. This time there is no corresponding edit operation in the edit script of the modified code fragments. Thus, there is no use annotation to add.

Finally, ARES also adds use annotations for the two final edit operations in the edit script of the modified code parts. Both operations do not have counterparts in the original original parts. To be precise, the edit script from $m_\alpha$ to $m_\beta$ contains a `delete` operation of `this.print` and an `insert` operation of `System.out.println` (see Fig. 4.5(b)). For the `insert` operation, ARES inserts a use annotation below the `if` statement (lines 13–15) in $m_\beta$. For the `delete` operation, ARES also determines that the use has to be placed below the `if` statement and before the call to `shutdown`. This creates two adjacent use annotations that ARES combines into one.
4.4.5 Wildcard Name Assignment

This step assigns the names to the wildcard and use annotations. The names identify annotation pairs that belong together. These pairs allow the pattern to express movements of arbitrary code. This increases the accuracy of the recommendations.

To identify the wildcard and use pairs, ARES uses the information it gathered during the wildcard replacement. For example, ARES knows that the wildcard in line 12 replaced the inserted updateValue call, the inserted printValue call, the deleted declaration of tmp and the deleted init call. It also knows that the use annotation in line 12 replaced the inserted updateValue, printValue, the deleted declaration of tmp and the deleted init call. With this information and with the mappings from $o_\alpha$ to $m_\alpha$, and from $o_\beta$ to $m_\beta$ ARES can determine which wildcard annotation belongs to which use. The updateValue statement in line 8 of $o_\beta$ forms a pair with the updateValue statement in line 8 of $m_\beta$ in the mapping from $o_\alpha$ to $m_\alpha$. As the updateValue statements form a pair in this mapping, the wildcard and use annotations that replaced them are also related. Thus, the use annotation in line 12 is a suitable partner for the wildcard annotation in line 12 and ARES assigns the same name (ARES1) to both. For this assignment process, a tree differencing algorithm that identifies moves is essential. Only with an accurate detection of moved code parts, ARES is able to identify the correct wildcard and use pairs.

There are some corner cases in this Wildcard Name assignment step. If there are different use annotations that share replaced statements with a single wildcard, ARES chooses the use that shares the most replaced statements with the wildcard. If there are several use annotations that share the same number of replaced statements, ARES uses the first use that occurs during the post-order traversal of the AST of the modified pattern part.

After the annotation insertion step and the described name assignment, the first statement inside the match annotation is a stmt-wildcard due to the insert operation of this.i (not shown in Fig. 4.2). This wildcard would cause problems during the search for pattern applications. It would allow the pattern to start at any arbitrary code position and thus would lead to thousands of unnecessary recommendations. To avoid this, ARES deletes all wildcards at the beginning of the match annotation body in the original pattern part. If ARES removes such a wildcard, it also examines the use annotation with the same name (if available). If this use is also at the beginning of the match annotation body in the
4 Accurate Code Recommendations

modified code part, ARES also removes it. Otherwise, ARES removes the name of the use annotation.

Similarly, a stmt-wildcard at the end of the match annotation body would also match an arbitrary number of statements without additional information for the developer. Thus, ARES also removes wildcards at the end of the match annotation body. If there is a corresponding use, ARES removes it if it is also at the end of the match body. Otherwise, it removes the name of the use annotation.

4.4.6 Choice Insertion

This step adds choice annotations to the modified part of the pattern. They are responsible for expressing differences in $m_\alpha$ and $m_\beta$ that have no corresponding difference in the edit script between $o_\alpha$ and $o_\beta$. ARES expresses such differences in the pattern to increase the accuracy of the recommendations.

In general, it is possible that a code change from the input examples (e.g., $c_\alpha$) contains inserted statements. If these inserted statements are not part of the second code change (e.g., `this.print` in line 16 of Fig. 4.4(a)) the above annotation types cannot handle the differences. This makes the choice annotations necessary to increase the accuracy of the recommendations. After the Wildcard Name Assignment step, it is easy for ARES to identify the positions of the choice annotations. They have to replace all use annotations without assigned names. Each choice annotation added in this way consists of one or two case annotations. ARES determines the statements within these case annotations with the help of the corresponding use annotations. One of the case annotations contains all deleted statements that were originally replaced by the use annotation. The other case annotation contains all inserted statements that were originally replaced by the use annotation. It is possible that a single use annotation replaced deleted and inserted statements because the previous steps merge adjacent annotations.

The example of Fig. 4.4 leads to a use annotation without name below the if in the modified part of the pattern. ARES changes this use to a choice annotation. The use annotation replaced the deleted statement `this.print` and the inserted statement `System.out.println`. Thus, ARES places each of the two statements in an own case block.
4.4.7 Second Iteration of the Pattern Creation Process

As the Pattern Creation step of the first iteration differs slightly from other iterations, this section shows the difference by means of an example. The pattern in Fig. 4.2 is based on three code changes. After the first iteration discussed in the previous sections, a second iteration with the change $c_\gamma$ (see Fig. 4.9) creates the resulting pattern. Figs. 4.10 and 4.11 hold the differences between $c_\gamma$ and the generalized pattern of the first iteration. In the second iteration, $c_\gamma$ takes the place of $c_\alpha$ and the created generalized pattern of the first iteration takes the place of $c_\beta$. Again, phase 1 of the Change Isolation step is insufficient due to the changed exception of the try statement in $c_\gamma$. As the try statement is part of the first change but not of the second, the first heuristics of phase 2 identify the body of the try statement in $c_\gamma$ as change root.

The Edit Script Adjustment step only differs from the first iteration in the rules that are applicable to the edit scripts. The rule-based system removes all edit operations that affect the annotations from the first iterations. Additionally, Rule #04 removes the move of System.out.println as it is paired with a node in the choice annotation (see Fig. 4.12). The recognition of code lines that are moved into parts of the choice annotation requires a tree differencing algorithm like MTDIFF that detects such movements accurately.

After the Edit Script Adjustment step, ARES adds wildcards for all the additional statements in the loops in $o_\gamma$ and merges all of them together with the wildcard in the loop of the generalized pattern. It handles the use annotations for the additional statements in a similar fashion. Due to the additional assert statement in line 5 of the modified part of $c_\gamma$, ARES also adds another use annotation before the for loop. For this concrete example, the Wildcard Name Assignment does not induce any changes in the second iteration. In general, it works like the name assignment during the first iteration. Finally, the Choice Insertion step replaces the new use before the for loop with a choice annotation that holds the assert statement from $c_\gamma$. As the use annotation did not replace an inserted statement, the choice annotation contains only one case block. In general, the Choice Insertion step also merges adjacent choice annotations. If two choice annotations are next to each other, this step moves the case annotations from the second choice annotation into the first one. Afterwards, it removes the now empty second choice annotation from the pattern. After the described second iteration, the resulting pattern is the one shown in Fig. 4.2.
try
{
    this.i = 5;
    this.init(verbose);
    updateValue();
    l = 0;
    while(l < 10) {
        updateValue();
        printValue("foo");
        if (this.containsException()) {
            for (Watcher w : this.watchers) {
                Message m = createMessage();
                m.saveState(this);
                w.throwException();
                break;
            }
            l++;
        }
    }
    foo.someMethod(42);
}

catch (Exception e) {
    System.out.println("foo");
    this.shutdown();
}

Figure 4.9.: Third code change c• Same legend as in Fig. 3.2.
Figure 4.10.: Differences between the original parts of $c_\gamma$ and the pattern from the first iteration. Annotations are in blue. Otherwise, same legend as in Fig. 3.2.
\textbf{Figure 4.11:} Differences between the modified parts of $c^\gamma$ and the pattern from the first iteration. Annotations are in blue.

```
{ //
  this.shutdown();
  { //
    this.print( foo );
  } // case
  System.out.print( foo );
} // case

{ //
  foo.someMethod( k );
  if( foo != null ) {
    foo.someMethod( 42 );
  }
  this.print( foo );
  this.shutdown();
}
```

Otherwise, same legend as in Fig. 3.2.
4.5 Search for Applications

With the created patterns, ARES is able to search for code locations that allow the application of the patterns. Whereas other parts of ARES rely on tree-based algorithms to increase the accuracy, the Search for Applications step uses token-based algorithms as it has to be fast enough for large code bases. To be precise, ARES uses AST nodes as tokens and compares the list of AST nodes of a code location with the list of AST nodes of the original part of the pattern. Besides the detection of the location, the following Recommendation Creation step (see Sec. 4.6) requires an assignment of code location nodes to nodes of the original part of the pattern. This search step also takes care of these assignments. The node assignment also determines which code parts are matched by a wildcard. Afterwards, ARES can move tokens matched by a wildcard with name ARES? to the position of the use with the same name.

```java
1 Foo foo = Library.getObject();
2 this.init(getVerbose());
3 this.shutdown();
4 updateValue();
5 c = 0;
6 while (c < 10) {
7    System.out.println(c);
8    c++;
9 }
10 foo.someMethod(99);
11 return foo;
```

Figure 4.13.: Code base for the search example.
To give an example for the search algorithm of ARES, this section applies the pattern from Fig. 4.2 to the code location in Fig. 4.13. The Search for Applications step starts with the identification of valid starting points in the code base. The body of the match annotation from the original pattern part provides the input for this starting point search algorithm. The first non-wildcard statement in the body of the match annotation defines the node label of valid starting points. Thus, as the first statement node in the match body in Fig. 4.2 is a Call node, valid starting points for the patterns are all Call nodes in the code base. This also fulfills the complete condition for valid starting points that was used in the previous work on ARES [88]. However, this has been changed in the most recent ARES version as the old search was slower than the search in LASE.

In the new version of this thesis, valid starting points have to fulfill an extended condition to increase the performance of the search. Valid starting points have to begin a subsequence of statements that have the same labels as the sequence of statements defined by the original part of the pattern. This sequence from the pattern does not include annotations. The matching subsequence of statements at the code location does not have to include every statement from the code location, i.e., it can have gaps. Otherwise, wildcards in the pattern would have no effect. The statements in the subsequence of the code locations also do not have to be identical to the statements of the pattern. It is sufficient that they have the same label. However, the statements in the subsequence have to be in the same order as the non-wildcard statements in the pattern. For the pattern in Fig. 4.2, the sequence of statements consists of three calls (this.init, this.shutdown, updateValue), an assignment ($k = 0$), a while loop, and another call (foo.someMethod). Thus, a code location requires a subsequence of statements that have the same labels and appear in the same order as these six statements from the pattern. The code location in lines 2–10 in Fig. 4.13 fulfills this condition. Starting in line 2, the valid subsequence contains three calls (this.init, this.shutdown, updateValue), an assignment ($c = 0$), a while loop and another call (foo.someMethod). Thus, line 2 in Fig. 4.13 is a valid starting point.

After the identification of all valid starting points, ARES executes the search algorithm in Fig. 4.14 for each starting point in parallel. Besides a valid starting point, the search algorithm also requires the method body that contains the starting point and the original part of the generalized pattern as input.
4.5 Search for Applications

With lines 2–3 of the search algorithm in Fig. 4.14, ARES creates the serialized list of AST nodes for the code location and for the pattern. To serialize the ASTs, ARES uses in-order traversal. The list NL_{cl} of the code location starts at the given valid starting point and ends at the last node of the code block that contains the starting point. The list NL_{p} starts at the first node inside the body of the match annotation that is not part of an expr-wildcard subtree.

The loop in lines 6–29 determines whether or not the pattern is really applicable to the code location. In each loop iteration, the algorithm takes one node from the code location list (n_{cl}) and one node from the pattern list (n_{p}) and compares them. Lines 14–17 handle the standard case that occurs if no wildcard is involved. In the given example, the list
of AST nodes $\text{NL}_{cl}$ begins with a \textit{Call} node. The first node in the pattern list is also a \textit{Call} node. Thus, as both nodes have nothing to do with \textit{wildcards}, ARES executes the standard case and uses \textit{isMatch} to compare them. In general, \textit{isMatch} is true if both nodes have the same label. There are only two exceptions. If the nodes are both identifiers, ARES also checks their values. The values have to be equal or the identifier node $n_p$ from the pattern has to be in the list of arbitrary identifiers on the \textit{match} annotation. Allowing only identical identifiers if they are not in the \textit{match} annotation list reduces the number of wrong recommendations and thus increases the precision of a pattern. The second exceptions for the \textit{isMatch} function are the boolean constants \textit{true} and \textit{false}. Only if the value for the constants is identical, the \textit{isMatch} condition is fulfilled. The reason for treating boolean constants differently is their limited range. A different boolean value can make a large difference in the semantics of a code fragment. For all other nodes (e.g., numbers) or even inner nodes (e.g., assignment nodes like “\texttt{=}, “\texttt{>=}”), only the labels matter. This generalizes the pattern and thus can increase the recall (i.e., the number of correct recommendations).

If the \textit{isMatch} condition is fulfilled, ARES adds the pair $(n_{cl}, n_p)$ to the set of \textit{resultPairs}, increments the position counters, and continues with the next loop iteration. Fig. 4.15 illustrates the \textit{resultPairs} for the first nodes in the code location. During the subsequent Recommendation Creation, ARES uses these pairs to replace nodes in the pattern with nodes from the code location. In the example from Fig. 4.13, ARES pairs the constant node 99 with the node 23 of the pattern. With the
information in resultPairs, ARES can replace a 23 node in the recommendation with the more accurate node 99. If isMatch rejects a node pair \((n_{cl}, n_p)\), ARES tries to backtrack to the last valid position (line 31). The last valid position contains different position counters and snapshots of resultPairs and visited. Using backtracking is necessary as wildcards can match none or an arbitrary number of statements or expressions. As ARES does not know beforehand how many statements a wildcard may consume, it stores each valid intermediate state in resets.

After matching the argument list Args (see Fig. 4.15), ARES sets \(n_p\) to verbose which is associated with a expr-wildcard. This triggers a different behavior in the algorithm. First, it sets \(w\) to the wildcard node associated with verbose (lines 11–12). Second, it executes the wildcard handling in lines 19–29. If this is the first appearance of a wildcard, ARES creates a reset point for the backtracking. Each reset point contains the current resultPairs and visited sets. The initial reset point for each wildcard handles the case that the wildcard does not match any nodes. Thus, the pattern allows that the code location calls init without arguments. In this reset point, ARES keeps the code location \(pos_{cl}\). This ensures that after backtracking to the reset point, ARES continues with the current node at the code location. Whereas this position remains unchanged, ARES increments the pattern position \(pos_p\). This skips the verbose node that is associated with the wildcard if the backtracking algorithm goes back to the reset point. Without this increment, the search algorithm would be trapped in an endless loop. This first reset point is useful if the argument list in the code location is empty. After backtracking to this reset point, the search algorithm can continue with the Call node in the next line of the original code from pattern. Thus, ARES would no longer try to find matching nodes to verbose.

In the code location of the example in Fig. 4.13, init contains an argument, namely getVerbose(). Thus, in the first iteration for the verbose expr-wildcard ARES uses the method allowedNode to determine whether the call node (parent of getVerbose) fulfills the condition of the wildcard. For an expr-wildcard the allowedNode condition is fulfilled if the node is inside the scope of the wildcard. The scope is determined by the position of the wildcard expression. In the example, verbose is in the argument list of init and it is also the first argument. Thus, the scope of the wildcard includes the complete argument list of init. In contrast, any node that is not part of the argument list of init at the
4 Accurate Code Recommendations

code location is outside the scope. Besides `allowedNode`, ARES uses another method called `allowedReset` to further check whether the current node is a valid reset point. This is the case if the node completes the current expression or the current statement. The check ensures that the used reset points are complete expressions or statements. This is necessary because the code matched by a `wildcard` can be moved to a new location in the recommendation. Moving only parts of expressions or statements would create a recommendation with incorrect syntax.

If the previous checks determine that a node is a valid reset point, ARES uses the method `hasSufficientStatements` in lines 20 and 26. This is a novel feature published for the first time in this thesis. It was not included in the original publication of ARES [88]. This new method checks whether it is possible to match the remaining non-`wildcard` statements of the pattern, if the search would backtrack to this reset point. In the running example, ARES matches `foo.someMethod` of Fig. 4.15 with the `stmt-wildcard` that has the name `ARES2`. In this case, `allowedReset` is true, as the `stmt-wildcard` consumed a complete statement. However, backtracking to the reset point after `foo.someMethod` does not fulfill the `hasSufficientStatements` condition. Starting from the reset point, there is only a `return` statement left in the code location. The pattern, however, requires that a `Call` statement follows after the `wildcard` with name `ARES2`. Due to this failed check, ARES does not add another reset point. Together with the new extended starting point condition, this feature speeds up the search for pattern applications.

If `hasSufficientStatements` is also fulfilled, ARES creates a new reset point. Again, this reset point contains a snapshot of the `resultPairs` and `visited` set. In contrast to the first appearances of a `wildcard`, ARES increments both the pattern position and the position of the code location. This is necessary, as `allowedReset` takes the current node \( n_{cl} \) into account to check for complete statements. To avoid backtracking into the middle of a statement, ARES increments \( pos_{cl} \) for the reset point. For the `getVerbose()` call in the running example, the last node of the `Call` subtree (i.e., the identifier `getVerbose`) fullfills all necessary conditions and ARES creates a reset point.

After matching the last node in `getVerbose()`, the loop continues. The position \( pos_p \) still points to `verbose` and thus to its associated `wildcard`. However, the next node in the code location is the `Call` node in line 3. For this node, the `allowedNode` check fails and ARES backtracks to the most recent reset point. This reset point does not influence the
4.6 Recommendation Creation

currently examined code location node (i.e., the Call), but sets the position of the pattern $pos_p$ to the next node. This is also the Call node of the following statement in line 4 of the pattern. Then the loop in lines 6–31 continues with an examination of both Call nodes.

For the stmt-wildcards in lines 12 and 15 of the pattern in Fig. 4.2, the search execution is similar to the expr-wildcard handling described above. This time, however, allowedNode accepts all nodes until the end of the code block. For a stmt-wildcard, the check allowedReset is fulfilled if the tested node is the last node of a statement from the block that contains the first statement matched by the corresponding wildcard.

The loop of the search algorithm terminates if it reaches either the end of the token list from the code location or the end of the token list from the pattern. After its termination, ARES checks whether the loop reached the end of the pattern and thus whether each of the pattern nodes has a matching node in the code location. If this is the case (i.e., $pos_p = |NL_p|$), the search algorithm has successfully identified a code location and returns the set of matched pairs (lines 33–35) for the next step.

4.6 Recommendation Creation

For each identified code location, ARES creates a recommendation. Central to this step is the previously identified code location. At the beginning of this step, ARES uses MTDIFF to compute the edit script between the original and the modified part of the pattern. Then it executes the computed edit script on the code location to create the recommendation. This has the advantage that the approach preserves variations in the code location like the identifier $c$ (instead of $k$ in the pattern) and the constant $99$ (instead of $23$ in the pattern).

ARES starts the Recommendation Creation step by copying the identified code location. As it is unclear whether or not a developer will accept the recommendation, ARES must not change the code location directly. After the creation of a copy, ARES transforms it until it looks like the original part of the pattern (except of some node values). To achieve this, ARES removes all statements that it paired with a stmt-wildcard during the previous step and inserts the corresponding wildcard at their position. It also replaces the expression matched by a expr-wildcard with the expression specified on the wildcard annotation in the
Accurate Code Recommendations

Thus, for the example in Fig. 4.13 ARES replaces `getVerbose()` with `verbose`. Additionally, ARES inserts the `expr-wildcard` before the associated statement. After this process, the copy of the code location looks like the original part of the pattern except for the value of some nodes. In the running example these are the identifiers `c` and the constant 99.

The next part of the `Recommendation Creation` applies the edit script of the pattern to the modified copy. Afterwards, it replaces the `use` annotations. First, for each `use`, ARES determines the `wildcard` that bears the same name. Then it collects the statements that were paired with this `wildcard` during the search step. At the position of the `use` annotation ARES adds the statements collected from the `wildcard`. Thus, ARES

```
1 Foo foo = Library.getObject();
2 this.init(getVerbose());
3 updateValue();
4 assert (this.value == 23);  
5 for (k = 0; k < 10; k++) {
6   System.out.println(c);    
7 }                            
8 if (foo != null) {
9   foo.someMethod(99);
10 }                           
11 System.out.print(foo);      
12 this.shutdown();           
13 return foo ;               
```

(a) First recommendation variant.

```
1 Foo foo = Library.getObject();
2 this.init(getVerbose());
3 updateValue();
4 assert (this.value == 23);  
5 for (k = 0; k < 10; k++) {
6   System.out.println(c);    
7 }                            
8 if (foo != null) {
9   foo.someMethod(99);       
10 }                          
11 System.out.print(foo);      
12 this.shutdown();           
13 return foo ;               
```

(b) Second recommendation variant.

```
1 Foo foo = Library.getObject();
2 this.init(getVerbose());
3 updateValue();
4 assert (this.value == 23);  
5 for (k = 0; k < 10; k++) {
6   System.out.println(c);    
7 }                            
8 if (foo != null) {
9   foo.someMethod(99);       
10 }                          
11 System.out.print(foo);      
12 this.shutdown();           
13 return foo ;               
```

(c) Third recommendation variant.

Figure 4.16.: Code recommendations for the ARES example.
replaces the use annotation named ARES1 with the System.out.println call from the code location in the running example. For use annotations that affect expressions, ARES replaces the expression associated with the use annotation. Again, the replacements are the expressions that were paired with the wildcard bearing the same name. In the running example, ARES replaces verbose with getVerbose() from the code location. Afterwards, ARES deletes the use annotations for the expressions.

The final part of the recommendation creation handles the choice annotations. For each choice annotation, ARES has the option to replace it with the body of a case annotation and also the option to delete the choice without inserting any code. This means that for the first choice in the pattern (lines 6–10 in Fig. 4.2) ARES has two options, for the second choice in the pattern (lines 17–24) ARES has three options. To cover all possibilities, ARES would have to generate $2 \cdot 3 = 6$ variations of a single recommendation. This would lead to a large number of variants that all have to be examined by developers. To avoid this overhead, ARES only generates a subset of these possibilities. It determines the maximal number of cases $max$ on any choice annotation in the pattern (2 cases in Fig. 4.2). Then ARES generates $max + 1$ variations of the pattern. In the first variation, ARES always inserts the code of the first case of each choice annotation. In the second variation, ARES always inserts the code of the second case annotation of each choice. If a choice annotation only has one case annotation, ARES simply deletes the choice in this variation without adding any code. This reduces the size of the recommendation. For the final variation, ARES removes all choices completely without inserting any code. Fig. 4.16 holds all three variations for the code location in Fig. 4.13.

**4.7 Complexity Analysis**

This section discusses the worst case time complexities of the different workflow steps (see Fig. 4.1). The workflow starts with the Input Order Determination. In the first iteration, ARES compares each input example $m$ with all other $m - 1$ input examples. For each comparison, ARES uses MTDIFF to compute the edit script between the original parts and the edit script between the modified parts. Thus, the complexity of the first iteration is in $O(m^2 \cdot 2 \cdot O_{MTDIFF})$. For each of the following
Accurate Code Recommendations

\[ m - 2 \text{ iterations, the determination step compares the generalized pattern with the remaining input changes. Again, this comparison requires the computation of the edit script once for the original parts and once for the modified parts. Thus, the complexity for each following iteration lies in } O(m \cdot 2 \cdot O_{\text{MTDIFF}}) \text{. Overall, the time complexity of the first step in ARES is in } O(m^2 \cdot O_{\text{MTDIFF}}) \text{. As a consequence, the execution time of the Input Order Determination depends on the number of input examples and on the performance of MTDIFF.}

The Change Isolation step as well as most of the other steps require the results of MTDIFF from \( o_\alpha \) to \( o_\beta \), from \( o_\beta \) to \( o_\alpha \), from \( m_\alpha \) to \( m_\beta \), from \( m_\beta \) to \( m_\alpha \), from \( o_\alpha \) to \( m_\alpha \), and from \( o_\beta \) to \( m_\beta \). Overall the complexity is in \( O(6 \cdot O_{\text{MTDIFF}}) = O_{\text{MTDIFF}} \). As the Edit Script Adjustment step can reuse the results from the Change Isolation step, it does not need additional calls to MTDIFF. Thus, the complexity depends solely on the applied rules. In general, the most time consuming rules either compare all edit operations \( e \), where \( e = \max(e_{o_\alpha, o_\beta}, e_{m_\alpha, m_\beta}) \), or all pairs in the mapping \( p \) with each other (\( p = \max(|p_{o_\alpha, o_\beta}|, |p_{m_\alpha, m_\beta}|) \)). Some rules also compare all node pairs \( n \) in the trees, where \( n = \max(|n_{o_\alpha}|, |n_{o_\beta}|, |n_{m_\alpha}|, |n_{m_\beta}|) \). Thus, the complexity of the Edit Script Adjustment step is in \( O(e^2 + m^2 + n^2) \). As \( n \) is the upper boundary for the edit operations and mappings, the time complexity is in \( O(n^2) \).

The Match Insertion and also the Wildcard and Use Insertion steps both have to iterate over all edit operations. Their complexity is in \( O(n) \). As there are at most \( n \) use annotations and also at most \( n \) choice annotations, the complexity of the Wildcard Name Assignment and Choice Insertion steps is also in \( O(n) \).

With \( N \) AST nodes in the code base, there are at most \( N \) possible start nodes for a pattern. For each start node, the search algorithm has a worst case time complexity of \( O(N^w) \). In this case, \( w \) is the maximal number of wildcards within the same scope. The scope of each wildcard is their parent code block. Thus, if two wildcard annotations are in the same code block in a pattern, \( w \) is two. The complete complexity of the code location search is in \( O(N \cdot N^w) \). However, in many patterns there is only one wildcard within the same scope. Additionally, \( \text{hasSufficientStatements} \) reduces the backtracking locations. Thus, in many cases this degenerates to \( O(N \cdot N^1) \). Additionally, only a subset of the nodes are valid starting points for a pattern. Thus, for many patterns, the search complexity lies in \( O(N) \).
4.8 Evaluation

The Recommendation Creation requires a single execution of MTDIFF on the pattern. Afterwards it simply applies the edit script of size $e$. Thus, the time complexity of this step is in $O(O_{MTDIFF} + e) = O_{MTDIFF}$.

In summary, the worst case time complexity of the complete pattern creation process in ARES is in $O(m^2 \cdot O_{MTDIFF} + n^2)$. The generation of recommendations for a single pattern is in $O(N \cdot N^w + O_{MTDIFF})$.

4.8 Evaluation

The goal of this evaluation is to answer 4 questions:

1. Does ARES provide more accurate recommendations compared to the state-of-the-art tool LASE?

2. Are the precision and recall values of ARES on par with the state-of-the-art tool LASE?

3. Is ARES fast enough for large repositories?

4. Does ARES provide more accurate recommendations than LASE on a large dataset?

Each of the following sections targets one of these questions and compares ARES with LASE to answer them. As dataset, the first three sections use two groups of similar code changes from the Eclipse JDT project [101] and 21 groups from the Eclipse SWT project [102]. Each of the 23 groups consists of several code changes that are present in the respective repositories. All changes of a group belong to the same systematic edit and fix the same bug. The 23 groups were manually collected by Meng et al. [236] for their original publication and for their evaluation of LASE. Using these 23 groups in the following evaluation allows a fair comparison as these code changes were selected by other authors (i.e., Meng et al.). The different groups vary in their complexity and in the number of changes they contain. To answer the fourth research question, Sec. 4.8.4 uses the clustering results from $C3$ (see Chapter. 5) of the JUnit [175] repository. This dataset contains 3,904 groups of similar code changes. All four sections used a 3.6 GHz Intel Xeon CPU equipped with 128 GB of RAM, OpenJDK 8 and Ubuntu 16.10. This chapter concludes with a discussion of the limitations of ARES and the threats to the validity of this evaluation.
4 Accurate Code Recommendations

4.8.1 Recommendation Accuracy

Table 4.3 holds the evaluation results of LASE and ARES for the 23 groups of code changes from the Eclipse JDT and Eclipse SWT repositories. The first two rows (Ids: 1, 2) belong to code changes from the Eclipse JDT repository. Each row contains the Bugzilla Id [100] that Meng et al. used to identify the group of code changes in the repositories. Table 4.3 also provides the number of code changes \( m \) that each group contains. For almost all groups in the evaluation, \( m \) is equal to the number of code changes provided by Meng et al. However, for the two changes with Ids 20 and 21, \( m \) differs from the reported values, as it was impossible to identify the same number of changes in the repositories. For Id 20 a search for "79107" in the commit messages of the Eclipse SWT repository reveals 10 changes instead of 3. For Id 21 it was only possible to identify 3 instead of the 4 changes that were reported by Meng et al.

Besides the general data, Table 4.3 contains two distinct segments, namely LASE—Two Input Code Changes and ARES—Two Input Code Changes. They hold the evaluation results of the two tools. To create the results for both segments, two code changes from each group were used as inputs. For a fair comparison, the same two code changes were used as inputs for LASE and for ARES. Meng et al. also used two code changes from each group to generate their results. To identify the pair that Meng et al. used, all combinations within each group were evaluated with LASE. For each group, this thesis used the code change combination that reproduced the same precision and recall results that Meng et al. have measured. For 16 of the 23 groups, it was indeed possible to find a code change combination that creates the results measured by Meng et al. For 4 of the 23 groups (Ids 1, 12, 16, 23) all possible pairs within the groups led to lower precision and recall values. The most likely explanation for this difference is that the publicly available version of LASE that was used in this thesis is not the version that was used for the publication. Even to reproduce the 16 identical results, it was necessary to patch several bugs in the publicly available version. For one of the 23 groups (Id 2) all combinations of input changes led to better results compared to the publication by Meng et al. It is possible that the applied patches also improved the results for this group. The results for the groups with Id 20 and 21 differ due to different sizes of \( m \).

For the evaluation, the selected two input examples of each group were used as inputs for both LASE and ARES to create patterns. The
Table 4.3.: Comparison with LASE on 23 code changes from Eclipse JDT and Eclipse SWT.

<table>
<thead>
<tr>
<th>Id</th>
<th>Bugzilla Id</th>
<th>m</th>
<th>LASE—Two Input Code Changes</th>
<th>ARES—Two Input Code Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\Delta)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>1</td>
<td>77644</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>82429</td>
<td>16</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>140007</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>139329(^1)</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>142947(^1)</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>91937</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>103863</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>129314</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>134091</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>139329(^2)</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>139329(^3)</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>142947(^2)</td>
<td>9</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>76182</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>77194</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>86079(^1)</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>95409</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>97981</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>76391</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>89785</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>79107</td>
<td>10</td>
<td>26</td>
<td>10</td>
</tr>
<tr>
<td>21</td>
<td>86079(^2)</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>95116</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>98198</td>
<td>15</td>
<td>67</td>
<td>12</td>
</tr>
</tbody>
</table>

Mean: Available Locations; \(\Delta\): Generated Recommendations; \(\checkmark\): Correct Recommendations;
\(A_T\)%: Token Accuracy; \(A_C\)%: Character Accuracy; \(P\)%: Precision; \(R\)%: Recall;

4.8 Evaluation
code base for each group was the revision of the project before the commit of the group’s oldest code change. Thus, the code base contained all unmodified \(m\) code locations of each group.

For each group and each segment, the \(\Delta\)-columns in Table 4.3 hold the number of recommendations made by the respective tool with the pattern generated from the two input examples. The \(\checkmark\)-column lists how many of the recommended code locations are also in the corresponding set of manually identified code locations. The columns \(A_T\) and \(A_C\) hold the mean accuracy values of the correctly identified input changes. The higher the accuracy, the higher is the syntactical similarity of the recommendations to the code changes in the repositories. Thus, the higher this value is, the fewer adaptations are necessary by a developer. As measurement for the accuracy, this evaluation uses the Levenshtein distance (LVD) between the method body from the recommendation and the method body from the repository. To generate the method body of the recommendation, this evaluation applied each recommendation to the code base and extracted the body of the patched method. The method body from the repository is the modified method body of the corresponding change. The LVD computes how many inserts, updates and deletes are necessary to convert the method body from the recommendation into the method body from the repository. This evaluation distinguishes between two different accuracy values. It uses \(A_T\) to measure the token accuracy and \(A_C\) to measure the character accuracy. The token accuracy shows how similar the syntax of the recommendation method body is to the syntax of the modified method body of the code change in the repository. For example, its value drops if the two method bodies use a different number of declarations. To get a value between 0 and 1, the evaluation uses \(A_T = 1 - \frac{LVD_T}{max(|r_T|, |m_T|)}\), where \(|r_T|\) is the number of tokens (i.e., AST nodes) in the recommendation and \(|m_T|\) is the number of tokens (i.e., AST nodes) in the method body from the repository. \(LVD_T\) considers two tokens as equal if they have the same label and the same value. As comments are not part of the token stream that consists of AST nodes, they do not influence \(A_T\). To measure the accuracy of comments, whitespace and structure tokens (e.g., parenthesis), this evaluation uses \(A_C\). This accuracy value is defined as \(A_C = 1 - \frac{LVD_C}{max(|r_C|, |m_C|)}\), where \(|r_C|\) is the number of characters in the method body of the recommendation and \(|m_C|\) is the number of characters in the method body from the repository.
Across all groups, the mean token accuracy $A_T$ of ARES is higher than the accuracy of LASE. The accuracy is higher because ARES handles code variation in the input examples better. It is also able to execute code movements more accurately. In 11 of 23 groups ARES even achieves a perfect accuracy of 100%. For these 11 groups, developers can directly apply the recommendations without any further changes. For some of the remaining groups, ARES was unable to create recommendations with a perfect accuracy because the two input changes did not cover the complete code transformations of the group (e.g., Id 10). In some cases (e.g., Id 2), the Change Isolation step selected a sub-node of the original change-roots to prevent over-generalization. As a consequence, the created patterns did not express the complete transformations that were present in the input examples. It also happened that wildcards cover variations in the input code changes (e.g., Id 1). This led to a reduction of the accuracy results as the pattern could not express code transformations hidden in the wildcards annotations.

For the group with Id 7, the recommendations of LASE have the lowest accuracy values. In this particular case, LASE inserted a statement at the wrong position in the recommendation. As the complete method body of the recommendation consisted only of three statements, the wrong position of the inserted statement had a huge effect on the results.

In five cases, the accuracy columns of ARES contain two values instead of one (Ids 2, 8, 12, 16, 23). For each of these groups, the pattern created by ARES contained at least one choice annotation. The left value is the mean accuracy of the recommendation variants that all had the lowest measured accuracy. The right value is the mean accuracy of the recommendation variants that all had the highest measured accuracy. For three of the five groups (Ids 2, 12, 23), even the minimal mean accuracy is higher then the accuracy of LASE. For the remaining two groups (Ids 8, 16) the accuracy of LASE is higher compared to the minimal accuracy values. LASE achieves a higher accuracy as its recommendations contained unaltered code parts from the code location although these code parts have to be changed according to one of the input examples. As LASE executes only edit operations that are shared by all input examples, it simply left the code parts untouched. Due to the created patterns for the two groups, ARES removed the changed code parts. Their intended replacement was part of a choice annotation. However, for patterns with choice annotations, ARES always creates
4 Accurate Code Recommendations

Table 4.4.: Results of ARES for 23 groups using all code changes as input.

<table>
<thead>
<tr>
<th>Id</th>
<th>Bugzilla Id</th>
<th>m</th>
<th>△</th>
<th>✓</th>
<th>AT%</th>
<th>AC%</th>
<th>P%</th>
<th>R%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>77644</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>90</td>
<td>89</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>82429</td>
<td>16</td>
<td>25</td>
<td>16</td>
<td>54</td>
<td>61</td>
<td>64</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>114007</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>139329</td>
<td>6</td>
<td>35</td>
<td>6</td>
<td>80</td>
<td>82</td>
<td>17</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>142947</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>91937</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>103863</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>129314</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>95/100</td>
<td>95/100</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>134091</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>99</td>
<td>99</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>139329²</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>57/100</td>
<td>56/100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>139329³</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>12</td>
<td>142947²</td>
<td>9</td>
<td>12</td>
<td>9</td>
<td>78/90</td>
<td>75/89</td>
<td>75</td>
<td>100</td>
</tr>
<tr>
<td>13</td>
<td>76182</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>92</td>
<td>92</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>77394</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>96</td>
<td>97</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>15</td>
<td>86079¹</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>93</td>
<td>92</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>16</td>
<td>95409</td>
<td>9</td>
<td>18</td>
<td>9</td>
<td>45/68</td>
<td>47/69</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>17</td>
<td>97981</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>18</td>
<td>76391</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>67</td>
</tr>
<tr>
<td>19</td>
<td>89785</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>84/100</td>
<td>79/99</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>79107</td>
<td>10</td>
<td>27</td>
<td>10</td>
<td>98</td>
<td>97</td>
<td>37</td>
<td>100</td>
</tr>
<tr>
<td>21</td>
<td>86079²</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>73/100</td>
<td>75/100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>22</td>
<td>95116</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>23</td>
<td>98198</td>
<td>15</td>
<td>356</td>
<td>15</td>
<td>76/95</td>
<td>79/95</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>Avg.</td>
<td>6</td>
<td>24</td>
<td>6</td>
<td>37/94</td>
<td>88/94</td>
<td>82</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

*m*: Available Locations; △: Generated Recommendations; ✓: Correct Recommendations; AT%: Token Accuracy; AC%: Character Accuracy; P%: Precision; R%: Recall;

a recommendation variant that does not contain any code from the annotations (see Sec. 4.6). For the two groups in the evaluation, these variants have the lowest accuracy values. In the variants with higher accuracies, ARES replaced the deleted code parts with the appropriate code from the choice annotations. For these variants, the accuracies of ARES are higher than the accuracies of LASE.

For 11 of the 23 cases, ARES achieves a character accuracy AC of 100%. This means that for these groups, ARES creates recommendations that produce exactly the same code that the developers have added to the repository.

There are use cases, in which a high recall is more important than a high precision. This is the case if developers search for critical bugs in their code base. One way to increase the recall for such a scenario is the manual adaptation of the patterns. An alternative is the use of additional input examples as they can cause additional annotations. These annotations make a pattern more general and thus increase its recall.
4.8 Evaluation

(a) Original part of the first change.  
(b) Original part of the second change.

Figure 4.17.: LASE context example. The context is in blue.

However, more annotations also have the potential to lower the precision and accuracy values of the created recommendations. To determine whether patterns with a high recall can still lead to recommendations with high accuracy values, an evaluation of ARES with more general patterns is necessary. Thus, for this high recall scenario, the evaluation used all $m$ code changes in each group as input for ARES to create patterns. This maximizes the recall and at the same time minimizes the precision and accuracy. Table 4.4 holds the results. For 16 of the 23 groups, the minimal accuracy values $A_T$ are still above the accuracy values of LASE with two input changes (see Table 4.3). For the recommendation variants with the highest accuracy, ARES has a higher accuracy in 21 of the 23 groups compared to LASE. Thus, even if ARES uses all input changes and lowers its accuracy for a systematic edit, the values are still above the accuracy results of LASE.

Overall, independent of the recall value, ARES achieves higher accuracy values than the state-of-the-art tool LASE on this dataset with 23 groups. This answers the initial research question.

4.8.2 Precision and Recall

High accuracy values are useless without high precision and recall values. A tool with a low recall value generates too few recommendations and a tool with a low precision creates too many wrong recommendations. In both cases, the accuracy is of no concern to developers as low precision and recall values make a tool unusable for them.

In general, precision $P$ is defined as $\frac{tp}{tp + fp}$ where $tp$ are the true positives (here: correct recommendations) and $fp$ are the false positives (here: incorrect recommendations). Thus, based on the values in Table 4.3, the precision can be computed as $\frac{\checkmark}{\Delta}$. The results show that
for 20 of 23 groups, the precision of ARES is equal or higher than the precision of LASE. In particular, ARES is more precise in three groups (Ids 10, 12, 23). For these three groups, the general context of the LASE pattern included only few statements. Thus, the pattern was also applicable to additional locations in the code base. In contrast, ARES includes identical code parts from the input examples even if they have no data dependencies to the systematic edit. This led to less general patterns.

LASE is more precise than ARES in three groups (Ids 2, 8, 20). For the group with Id 2, LASE has a higher precision value due to the context it used to search for code locations. In general, the LASE context consists of a forest of trees computed with the MCESE. In contrast, ARES uses the AST of the original pattern as context which is more restrictive. Whereas a forest allows gaps within the hierarchy of tree nodes at a code location, the pattern in ARES has a fixed node hierarchy. Fig. 4.17 shows an example that illustrates this issue. In the original parts of the two code changes, the \textit{if} statement and \textit{try} statement are at different places in the hierarchy. As a consequence, the context in LASE consists of two trees, indicated by \textit{stmt1} and \textit{stmt2}. Due to its pattern design, ARES replaces the \textit{if} and \textit{try} statements with a \textit{wildcard}. Thus, the context of ARES consists only of \textit{stmt1}. Exploring alternative pattern designs that also support such input changes is future work.

The group with Id 8 is a special case. ARES allows string constants to have arbitrary values at the code location. LASE enforces a bigram similarity of 0.6. This similarity threshold avoids wrong recommendations and increases the precision. It is also the reason for the higher precision of LASE for this group.

In contrast to ARES, LASE identifies common method signature parts in the input examples as well as common method names. Thus, it can use the parts of the signature that are shared by all input examples to filter the code locations. This increased LASE’s precision for the group with Id 20.

Recall \( R \) is defined as \( \frac{tp}{tp + fn} \), where \( fn \) represents the false negatives (not identified correct recommendations). Thus, based on the values in Table 4.3, the recall is \( \frac{\sqrt{m}}{m} \). For 7 groups (Ids 8, 12, 14, 15, 16, 20, 23) the recall of ARES is lower than the recall of LASE. The reason lies in the context included in the pattern. LASE only includes code parts in the context if they have dependencies to the changed code parts. ARES includes all code parts from the change roots that are identical in the input examples.
4.8 Evaluation

In the used dataset, the groups with Ids 20 and 23 are outliers because their precision is lower compared to all other groups. The lower precision values are caused by the small patterns that both LASE and ARES create for them. The cause for the small patterns are the very few lines that the input changes have in common.

Figs. 4.18 and 4.19 show the patterns that ARES created for both groups. The original part of the pattern for Id 20 contains only four statements and many of the used identifiers are part of the match annotation. Even the for loop is not distinctive as the condition is part of an expr-wildcard. The pattern for the code change with Id 23 is even smaller. Such general patterns fit numerous patterns in the Eclipse SWT repository and thus lower the precision.

Overall, the precision of ARES and LASE is on par as both as ARES and LASE have the same precision for 17 groups and ARES has a higher precision in 3 of the 23 groups (evaluation question 2). With 90%, the mean precision of ARES is also similar to that of LASE (92%). The recall of ARES is lower than LASE in 7 of the 23 groups. The mean recall values of LASE (87%) and ARES (76%) also reflect this difference.
Figure 4.19.: ARES pattern for Id 23.

(a) Original part.

```java
if (handle == 0) SWT.error(SWT.ERROR_GRAPHIC_DISPOSED);
```

(b) Modified part.

```java
if (handle == 0) SWT.error(SWT.ERROR_GRAPHIC_DISPOSED);
```
4.8 Evaluation

Figure 4.20.: Time comparison between LASE and ARES. The boxes show the 25%/75% quartiles, the lines indicate the medians. The lower whiskers are at the minimum, the upper whiskers are at the maximum, and the diamonds show the mean.

However, developers preferred a high precision over a high recall in the study by Christakis and Bird [61]. Thus, at least for the more important value, ARES is on par with LASE. Still, as Table 4.4 shows, it is possible to achieve a perfect recall with ARES if more input examples are provided. The mean precision and accuracy values for patterns with a perfect recall remain high ($A_T$: 88/94%, $P$: 87%).

4.8.3 Time Measurements

This section addresses the question whether ARES is fast enough to handle large real-world repositories. For that purpose, this section analyzes the times that ARES required to create patterns and the time it took to use the patterns on the repositories. As above, this evaluation part uses the 23 groups from the Eclipse JDT and Eclipse SWT repositories (see Table 5.1 on page 224 for details about the repositories).

Fig. 4.20 shows the execution times for ARES and LASE on the 23 groups. The measurement in each row of the figure consists of 23 data points, one for each group. Thus, the box plot in the first row consists of the times that LASE required to generate the patterns for the 23 groups.
Accurate Code Recommendations

For two input files, the Pattern Creation time of ARES is faster compared to LASE. The main reason is the use of a CD version without optimizations in LASE. This version is slower than MTDIFF which ARES uses. Additionally, LASE performs a dependency analysis to identify the context for the patterns. ARES currently works purely syntactical and does not have such an additional overhead. Using all input code changes to create a pattern increased the Pattern Creation time of ARES considerably. The reason is that ARES requires $2 \cdot m^2$ executions of MTDIFF to determine a suitable input order ($m$ is the input size for each group). For the largest group with 16 code changes (Id 2), this are over 512 executions of MTDIFF. As a consequence, ARES also has the highest Pattern Creation for the group with Id 2. However, for a practical use of ARES this is not a concern, as it is only necessary to generate a pattern once. Due to its design, ARES can store a generated pattern anywhere as text. This makes it possible to precompute a large number of patterns on a compute cluster. This option further reduces the severity of the pattern creation time. Additionally, even for larger groups of input changes, the creation time is below one minute and thus still allows an interactive use of ARES. It would also be possible to use a less time consuming Input Order Determination, although this could reduce the accuracy.

Whereas the pattern creation happens only once, the search for code locations happens each time a developer deems it necessary. Thus, it is more crucial than the pattern creation time. The Pattern Use time in Fig. 4.20 shows the corresponding measurement. The displayed times include the time to parse the files in the code base, the time to search for pattern applications, and the time to create recommendations. The measured times do not include the time to create Eclipse projects although this is a prerequisite of LASE. In the results, the time to search for code application dominates the time to create recommendations for both tools. For the 23 groups, ARES is slightly faster than LASE. It is also faster than the previously published ARES version [88]. The new determination of valid starting points and the reset point reduction with hasSufficientStatements (see Sec. 4.5) are responsible for these results. The Pattern Use time of LASE is higher as it uses the MCESE [218] to identify code locations. This has a complexity of $O(n^2 \cdot \min(d, l)^2)$, where $n$ is the number of nodes, $d$ is the depth, and $l$ is the number of leaves. This is slower than $O(n^w)$ of ARES if the number of wildcards $w$ is below 2 in the same block (see Sec. 4.7). ARES is not only faster
than LASE, it even has the advantage that it can detect several code locations within the same method. LASE examines each method only once and thus detects only the first pattern occurrence in each method.

In contrast to LASE, ARES is unable to filter possible starting points based on parts of the method signatures. As ARES is restricted to method bodies, its patterns do not contain the necessary information. Despite this disadvantage, ARES is still faster than LASE. In fact, for the 23 code changes from the Eclipse repositories, an artificial filter for method signatures improved the results of ARES only by around 100 ms for each group. This filter was artificial as it was not generated from the input examples by ARES but it was implemented manually for each of the 23 groups.

LASE has the highest Pattern Use time for the group with Id 7. As the input examples in this group are identical and all examples have a large method body, the pattern context that LASE created is also large. Such a large context increases the execution time of the MCESE.

In the configuration with two input changes, ARES has the highest Pattern Use time for the group with Id 9. The created pattern for this group only consists of a single if statement inside the match annotation. Thus, each if in a code base is a valid starting point for this pattern. This increases the number of times, ARES has to use its token-based search algorithm.

The group with Id 23 has the highest Pattern Use time in the measurement in which ARES used all code changes in a group to create the pattern. The execution time is at such a high value as ARES creates 356 recommendations for this pattern. The creation of such a large number of recommendations takes time because ARES has to compute the edit script of the pattern with MTDIFF again after each recommendation. This is necessary because the implementation of the Recommendation Creation step in ARES modifies the edit script to create recommendations. In summary, the analysis of these outliers and the 23 groups shows that the number of wildcards is not the decisive factor for an increase in the execution time.

Besides being faster than LASE, ARES is able to browse the large Eclipse repositories for pattern applications within 15 s for the majority of all groups. Thus, for the use of one pattern at a time, this is sufficient. Overall, this part of the evaluation shows that ARES is fast enough to be used on large repositories with real-world patterns.
4.8.4 **Comparison on the JUnit Dataset**

In general, previous works [236, 281] on example-based code recommendation systems only used small datasets of real-world examples. This lowers the validity of the results as it is possible to optimize any tool for such small datasets. To address this issue, the current section shows the evaluation results of LASE and ARES on a larger dataset. The evaluation uses 3,904 groups of code changes that \( C_3 \) (see Chapter 5) extracted from the JUnit repository [175]. As the number of code locations in the repository \( m \) is unknown for each group, it is impossible to compute and compare the precision and recall results of both tools. However, as the input code changes are available, it is possible to measure the accuracy for each recommendation that corresponds to an input code change. In contrast to the evaluation above, all input code changes were used to create the pattern for each group. This maximizes the number of recommendations. As code base for each group, this evaluation used the most recent repository revision before the commit of the oldest code change in each group.

Table 4.5 shows the results for both tools. The 3,904 groups of code changes contain 12,528 input code changes. LASE creates recommendations for 490 and ARES creates recommendations for 2,247 of the input code changes. In theory, the value for both tools has to be 12,528. However, \( C_3 \) clusters code changes based on their similarity to each other. There is no guarantee that all members in a group belong to the same systematic edit. Thus, not all groups from \( C_3 \) are suitable inputs for both LASE and ARES. Additionally, LASE and ARES are still research prototypes and not mature enough to handle each group of input changes in such a large dataset.

482 recommendations are in the common intersection of both tools. For these recommendations it is possible to compare the accuracy

| Table 4.5.: Accuracy on JUnit. |
|----------------|----------------|----------------|
| LASE | ARES (Min/Max) |
| Groups of Code Changes | 3,904 | |
| Code Changes in these Groups | 12,528 | |
| Created Recommendations | 4,232 | 61,791 |
| Created Input Recommendations | 490 | 2,247 |
| Shared Recommendations | 482 | |
| Shared Recommendations \( A_T \) (Mean) % | 90 | 91/97 |
| Shared Recommendations \( A_T \) (Median) % | 100 | 100/100 |
| Shared Recommendations \( A_C \) (Mean) % | 76 | 90/95 |
| Shared Recommendations \( A_C \) (Median) % | 82 | 100/100 |
4.8 Evaluation

values of both tools. The last four rows of Table 4.5 hold the summary of this comparison. Again, the mean accuracy values of ARES are higher than the accuracy values of LASE. This is even true for the recommendations variants with the lowest accuracy values. Thus, ARES even produces recommendations with higher accuracy than LASE on a large dataset.

There are several outliers that have a very low accuracy for both LASE and ARES. However, these outliers are caused by the groups from C3 and not by the tools themselves. Sec. 5.5.5 discusses such outliers in more detail.

4.8.5 Limitations and Threats to Validity

Currently, ARES only creates patterns for method bodies or smaller code fragments. Due to this limitation, ARES cannot create recommendations for code changes that transform method signatures or class bodies, or for code changes that span multiple files. This limitation also prevents ARES from using common parts in the method signatures of the input examples to increase its precision. This also increases the execution time of the search for pattern applications. It also had consequences on the used dataset. The original evaluation by Meng et al. [236] used 24 instead of 23 groups. It was necessary to exclude the group with Bugzilla Id 74139 from the evaluation because the code changes in this group only share a signature and have no common statements in their method bodies. In ARES, such input changes lead to an over-generalized pattern consisting of a single wildcard.

Extending the pattern creation in ARES beyond the scope of method bodies requires further research. The support of larger code fragments may require additional annotations because the order of fields or methods has no influence on the semantics of a Java program in contrast to the order of statements in a code block. It also requires new rules that preserve the balance between over-fitting and over-generalization under extended conditions. In contrast to the pattern creation step, all other parts of ARES require only minor adjustments. The search for code locations is neutral to the labels of the nodes it compares. Even the wildcard plugin system requires only a few adjustments to take the additional scopes into account. The recommendation creation uses only the matches from the code search and the edit script. Both are also independent of the AST node labels.
4 Accurate Code Recommendations

ARES is also limited to the Java programming language. To support other languages a parser that transforms the code into an ARES-AST is required. ARES also needs code generators for statements and expressions in the target language. Additionally, an adaptation of the rule-based system to the new language is necessary.

LASE, ARES and most other research recommendation systems share a common limitation. They are only able to search for one pattern at the same time. Using a database with millions of patterns would lead to a dramatical increase in the execution time of the tools and make them unusable. A possible solution to this problem are suffix trees. Such trees are already used by code clone detection tools like CCFinder [176].

A threat to the validity of the evaluation is the implementation of LASE. It is built from a publicly available version [203]. However, during the evaluation process it was necessary to patch several bugs. Only after these patches it was possible to reproduce the majority of the results reported by Meng et al. [236]. For the larger JUnit evaluation, it was also necessary to bypass the Eclipse UI that LASE uses. Only this bypass allowed the execution of the measurements with shell scripts. However, despite these problems, the precision and recall results of 16 from the 23 groups were identical to the ones reported by Meng et al. Thus, even if new problems were introduced due to the UI removal and the bug fixes, both do not impact the validity of the general results.

Another threat to the validity of the results is the maturity of ARES. It produced only recommendations for 2,247 of the possible 12,528 input code changes. Partially, this happened because not all groups provided by C3 solely contain code changes that all belong to the same systematic edit. Additionally, ARES is a prototype and thus some of the groups do not work because of bugs in the implementation. The rule-based system also cannot cover all input scenarios (see Mandl [223]). Despite these limitations, ARES is still more mature than LASE and provides accurate recommendations for many groups of code changes. Thus, it can provide actual advice to help developers, although it covers only a part of the systematic edits that occur in real-world repositories.

4.9 Summary

This chapter presented the novel Accurate REcommendation System (ARES). Similar to other state-of-the-art recommendation tools like LASE, ARES creates patterns from two or more input code changes.
With this input, ARES is able to search for code locations that allow the application of the patterns. If ARES identifies such a code location, it executes the code transformation specified by the pattern and presents the transformed code as recommendation to developers. Therefore, ARES can provide help for the execution of systematic edits. With two or more code changes of the same systematic edit as inputs, ARES searches for code locations that allow the application of this systematic edit. Additionally, ARES executes the systematic edit on the identified locations and presents the results as recommendations. Thus, with ARES developers no longer have to perform the repetitive systematic edits manually.

In contrast to previous recommendation systems for systematic edits, the focus of ARES lies on the accuracy of the recommendations. Thus, ARES is designed to create recommendations that accurately reflect what a developer would have written. The evaluation shows that for 11 out of 23 groups of code changes from the Eclipse JDT and SWT repositories, the recommendations of ARES have a perfect accuracy of 100%. This means that for these 11 groups, developers can directly add the code recommended by ARES to their code base without any additional changes. This avoids mistakes that can occur during a manual execution of systematic edits. ARES does not only achieve a perfect accuracy for 11 patterns, it also has a high mean accuracy of over 92% for the 23 groups. In comparison, the state-of-the-art tool LASE achieves a mean accuracy of 78%.

To achieve these high accuracy values, ARES addresses three challenges that example-based recommendation systems for systematic edits have to face. The first challenge comes from the variations in the input examples. The second challenge are code movements that appear in the input examples. ARES solves these two challenges with its unique pattern design. Each pattern consists of an original and a modified code part with annotations. With the help of wildcard and use annotations, ARES can express variations of the original code parts of the input examples. Due to an assignment of wildcard annotations to use annotations, the pattern design can also express the movement of an arbitrary number of statements. With the choice annotation, ARES is even able to express variations in the modified code parts of the input examples.

Finally, to make such a pattern design useful, ARES has to solve the third challenge. It needs algorithms that create the patterns from two
Accurate Code Recommendations

or more input examples. The pattern creation process in ARES works iteratively on pairs. In the first iteration, ARES uses a pair of two code changes as input. In each following iteration this pair consists of one code change and the created pattern from the previous iteration. To order a set of input code changes for this iterative process, ARES computes the syntactical similarity of the input code changes. In the first iteration ARES chooses the two code changes that have the highest similarity to each other and to all other code changes. In the following iterations, ARES selects the code change that has the highest similarity to the pattern from the previous iteration. After the input pair selection, ARES computes the edit scripts between the original code parts of the pair and also the edit script between the modified code parts. To find a balance between over-fitting and over-generalization, ARES adjusts these edit scripts with a rule-based system. Currently this rule-based system uses over 50 manually implemented rules. After the adjustment of the two edit scripts, ARES adds annotations to the pattern for each edit operation that remains in the edit scripts. This leads to the patterns that ARES uses to identify code locations.

The evaluation in this chapter showed that ARES is more accurate than LASE on the manual created dataset with 23 groups (92% vs. 78%). For these 23 groups, the precision of ARES is 90% which is on par with LASE (precision: 92%). The recall of ARES (78%) is lower compared to LASE (87%) due to the additional context that ARES includes in the patterns. LASE reduces this context with the help of a data dependence analysis. To improve ARES, it is possible to include such an analysis in the future. Additionally, the time measurements showed that the pattern search in ARES is slightly faster than LASE due to the fast token-based search in ARES. ARES also creates more accurate recommendations than LASE on the automatically created JUnit dataset with 3,904 groups.

In summary, ARES uses MTDIFF to generate code patterns from examples and uses these patterns to provide accurate code recommendations for developers. All, the evaluation input data, the evaluation results, and the source code of ARES are publicly available (https://github.com/FAU-Inf2/ARES). This version also includes a description of the rules that ARES uses. It also contains the readable patterns that ARES creates from the input data to kindle further research on recommendation patterns.
5 Clustering of Code Changes

If I have seen further, it is by standing on the shoulders of giants.  

Sir Isaac Newton

Code repositories hold the distilled knowledge of thousands of developers. The change histories from the repositories give blue prints for bug fixes, show how to refactor code for better readability, and how to adjust code to new library versions. Learning from such a treasure of knowledge is the key for a flexible recommendation system. For example, ARES can help executing a systematic edit if it gets just two or more similar method changes as input. Thousands of such similar changes are hidden in the histories of repositories. The problem is finding them.

A manual search for all similar code changes is impossible. The Eclipse JDT repository contained over 106,000 changed methods at the end of 2015. A complete manual evaluation would require approximately $106,000^2 / 2 = 5,618,000,000$ comparisons for this repository alone. Data mining approaches are more suitable for such huge data sets. They can examine the code changes in the histories of the code repositories automatically to identify groups of code changes that ARES and similar tools can use. For such recommendation tools, a mining approach has to identify code changes that are similar or identical in contrast to changes to identical or nearly identical code fragments. Only if the original and modified parts between two or more input code change differ, ARES is able to create a general pattern that also finds code locations distinct from the code locations of the input examples. However, too many differences in the original and modified parts lead to over-generalized patterns. Such patterns are useless. Thus, in order to obtain the most useful pattern, a data mining approach has
to identify groups of code changes that have similar but not identical original and modified code parts. The edit operations of the code changes within a group also have to be similar enough so that they are part of the same systematic edit. Only patterns of this kind are helpful for developers, as they address repetitive changes. Similar original and modified methods alone are insufficient, as this condition would allow two code changes within a group to oppose each other. For example, a suitable mining approach does not put two code changes \( c_1 \) and \( c_2 \) to the same group if \( c_1 \) deletes the \texttt{assert a \neq null} \) and \( c_2 \) adds the \texttt{assert a \neq null}. The combination of both does not lead to useful patterns. Besides the identification of similar code changes, it is also important that the data mining approach is fast enough to work on huge data sets.

This chapter is based on previous work [196].

It presents the tool \textbf{C3} (\textit{Clustering of Code Changes}) that fulfills the conditions described above and is able to create suitable groups of code changes for ARES. It uses clustering algorithms to identify groups of similar code changes. In contrast to state-of-the-art approaches (see Sec. 5.1), \textbf{C3} ignores the original and the modified parts of the code changes and solely focuses on the edit operations. This excludes groups that contain code changes that oppose each other. The evaluation in Sec. 5.5 shows that the identified groups still contain groups of code changes that have similar original and modified methods. \textbf{C3} also performs filtering steps to reduce the execution time of the system.

Fig. 5.1 shows the workflow of \textbf{C3}. As input it uses one or several repositories. From the historical data in the repositories ARES extracts all file changes and splits them into method-level changes. Each one of

---

1 I had the initial idea to apply clustering algorithms to code changes in 2015. However, the selection of the algorithms as well as their implementations and preliminary evaluation were made by Patrick Kreutzer during his master's thesis [195]. Patrick Kreutzer also contributed the \textit{k}-nearest neighbors filtering, the Kernighan-Lin algorithm, the heuristics for hierarchical clustering, and the heuristics for DBSCAN. It is unavoidable to describe some of his contributions in this thesis as their discussion is necessary to understand \textbf{C3} and its interaction with ARES. To the initial publication of \textbf{C3}, I contributed the identification of changed methods, the parallelization of the similarity computation, the measurement tools, and the complete evaluation including the analysis of the results.
these changes only contains edit operations that affect a single method. This makes the changes more useful for ARES, as it currently only uses method bodies to create patterns. Sec. 5.2 discusses the decomposition into methods which is the first part of the Similarity Computation. Sec. 5.2 continues with a description of the similarity matrix computation. Each row and column in this matrix belongs to a method-level code change that \( C3 \) extracts from the repositories. The similarity in each matrix cell corresponds to the string similarity between two code changes. To create a string that corresponds to a code change, \( C3 \) allows the use of two different configurations. It is necessary to set one of the configurations at the start of \( C3 \). It is not useful to mix their results. The first configuration (called Line-based representation below) applies the Unix tool \texttt{diff} to each code change and each result as string for the following similarity computation. The second configuration (called Tree-based below) computes an edit script with MTDIFF for each code change and converts this list of edit operations into a string. Then \( C3 \) uses a string similarity metric to compute the pairwise similarities for each matrix cell. The resulting similarity matrix is the input for the two supported clustering approaches. Currently, \( C3 \) uses either hierarchical agglomerative clustering [245, 336] or DBSCAN [109]. Sec. 5.3 describes the two clustering approaches, while Sec. 5.4 discusses the complexities of the algorithms and heuristics in \( C3 \). The evaluation in Sec. 5.5 applies \( C3 \) to nine software repositories. It also discusses the strength and weaknesses of the string representations and clustering approaches. Additionally, it evaluates the combination of \( C3 \) and ARES.

5.1 Related Work

There are several tools that are related to \( C3 \). However, they use different similarity methods, different clustering algorithms, and often have a different purpose. For example, FixWizard by Nguyen et al. [256] identifies recurring bug fix patterns in repositories. It analyses all code changes in a repository and computes the edit operations with an earlier version of the JSync tree differencing algorithm (see Chapter 3). Besides the edit operations, FixWizard also builds a graph-based object usage model (groum) [255] for the original and the modified code of each code change. Then it maps the changes of the ASTs to the groums and extracts the sub-groums that the edit operations affect. FixWizard serializes these sub-groums and computes the pairwise similarity between
code changes based on them. Two nodes in the serialized sub-groups are similar if they have the same value (e.g., call the same method) or the used methods are code peers (i.e., their name and feature similarity is above a threshold). In contrast to many other approaches in this thesis, FixWizard does not use the LCS algorithm or the LVD on the serialized lists. It just searches for the maximum number of similar nodes. If the serialized sub-groups of two code changes have a similarity above a certain threshold, FixWizards puts them in the same group. In contrast to $C_3$, this similarity measurement requires the identification of code peers and thus does not necessarily identify similar changes that appear in completely different methods. However, it is possible to use this similarity measurement as alternative input for the clustering algorithms in $C_3$. This is also a possible extension to FixWizard, as it uses only a single threshold to divide the changes into groups and does not use a clustering algorithm.

Higo and Kusumoto [142] proposed an approach that detects code clones based on code changes in repositories. If two code fragments were changed in a similar way in the past, their approach identifies them as clones. To detect similar code changes, their approach looks for similar deleted or inserted lines. Initially, it hashes the statements of the original and the modified version of a file in a commit. This has the advantage that it is possible to identify identical statements based on the hash value which is faster than a string-based or tree-based comparison of two statements. Afterwards, it builds sequences of the hashes to represent the statements in both versions. By applying the LCS algorithm on the sequence of hashes from the original version and the modified version, it detects deleted and inserted statements. Afterwards, it converts the deleted and inserted statements back to tokens. Then it compares the tokens of the deleted and inserted statements of all file changes in a pairwise manner. If it identifies identical code changes, it marks the code fragments that were affected by the same change as code clones. Whereas the approach by Higo and Kusumoto detects similar code fragments, $C_3$ detects similar changes. Despite this different purpose, $C_3$ is also more general, as it also searches for similar code changes and not only for identical ones.

Nguyen et al. [250] also proposed a tool that identifies identical or nearly identical code changes. First, their approach extracts the original and the modified AST of a change and normalizes both ASTs. The normalization step replaces local variables and literals with generic values.
Then their approach uses JSync to identify the edit operations on the normalized ASTs. To compare code changes with each other, it creates a hash value of the edit script. The result of the approach are groups of code changes in which each code change within a group has the same hash value. This approach mostly detects identical or nearly identical code changes. Only the normalization gives the approach some flexibility. In contrast, $C^3$ can also group code changes together if there are some additional operations (e.g., an additional assert) in one of them.

Wang et al. \cite{314} also presented a system to identify similar code changes. Their system extracts the \textit{diff} representation for all commits in a repository. Then it further splits this representation into hunks, i.e., changed code regions provided by the output of \textit{diff}. Then it detects clones in the hunks with CCFinder \cite{176}. It considers two hunks to be similar if they share an identical token sequence of at least 10 tokens. In contrast, one of the $C^3$ configurations uses the LCS algorithm on the characters of the \textit{diff} output and thus can detect similarities between changes that do not share 10 identical tokens in a row. As long as there are sufficient similar tokens in the LCS, $C^3$ considers two code changes to be similar. $C^3$ also uses methods to divide file changes into smaller ones. This is more suitable for ARES, as there can be several hunks for one method. To create accurate recommendations for a systematic change that covers the whole method body, ARES also requires examples that cover the complete body.

Negara et al. \cite{249} track edit operations inside an IDE and use this data to identify similar code changes. Their approach combines several edit operations into a transaction. Then they look for similar transactions with a frequent code change pattern mining algorithm that was inspired by CHARM \cite{335}. The new mining algorithm by Negara et al. has two key differences to CHARM. It uses bags instead of sets, because an edit operation can occur several times in a transaction. Additionally, it handles overlapping edit operations with the help of overlapping transactions. The approach is orthogonal to $C^3$ because the input data is different. $C^3$ uses commits from code repositories as input. This has the advantage that there are thousands of code repositories publicly available. Datasets that contain logged edit operation from IDEs are rare.

The tools that are most similar to $C^3$ are SysEdMiner and VuRLE. Both were published after the initial publication of $C^3$ \cite{196}. The tool SysEdMiner by Molderez et al. \cite{243} looks for similar changed methods
in repositories with closed frequent itemset mining [4]. To get the items for each method, SysEdMiner computes the edit script with ChangeNode, a variant of CD that uses a fine-grained AST as input, but keeps the heuristics of CD. Then it divides the resulting list of edit operations by the method they affect. In contrast, \( C_3 \) first identifies pairs of methods and then computes the edit scripts for the method pairs. This avoids edit operations that move code between different methods within the same class. With the closed frequent itemset miner CHARM [335], SysEdMiner looks for identical subsequences of edit operations between the changed methods. For that purpose, it is necessary to define equivalence criteria for edit operations. In SysEdMiner, two edit operations are equivalent, only if they have the same type (e.g., update) and their changed ASTs are structurally equal. This means that SysEdMiner only considers two deletes equal if both delete identical ASTs with identical labels. Only the values in the ASTs may differ. The edit operations also have to affect an identical context. The context is identical if the path between the root node of a method and the root node of an edit operation is identical between two edit operations, i.e. if the paths have the same number of nodes and the nodes have the same labels. In the terminology of frequent itemset mining, support denotes the number of identical subsequences found in all analyzed sequences. This means that an edit operation subsequence that appears in two different methods has a support of 2. A subsequence is closed if it is not part of another subsequence with the same support. For the subsequences a:3, ac:3, d:4, de:2 with the character sequence before the colon and corresponding support values behind the colon, the closed itemsets are ac:3, d:4, de:2. Although SysEdMiner identifies similar code changes, it does not cluster them to create groups. It simply returns the detected closed itemsets and their support. However, it is possible to use the closed frequent itemset mining in SysEdMiner to create a similarity matrix that \( C_3 \) can use.

VuRLE by Ma et al. [220] applies clustering to identify similar change blocks in already similar code changes provided by developers. Each change block contains the edit operations of a sequence of changed code lines and an AST of the code before the changed lines that is shared between the original and the modified code versions. To identify the sequence of changed lines, VuRLE uses diff. Similar to the tree-based representation in \( C_3 \), it first computes the edit operations for each sub-change. In contrast to \( C_3 \), it does not use an encoding but compares
the edit operations directly. This takes additional time and also is more sensitive to identifier names. Sec. 5.2.3 discusses the problem with identifiers in more detail. On the similarity values between change blocks, VuRLE uses DBSCAN and applies the heuristics of \textbf{C3} (see the previous work on \textbf{C3} [196]) to identify suitable parameters for DBSCAN. Whereas \textbf{C3} searches for similar code changes in a repository, VuRLE already works on similar code changes that solve the same problem. The purpose of the clustering in VuRLE is solely responsible for dividing already similar changes further into smaller groups. As a consequence, it does not work on larger data sets. This allows the more time-consuming direct comparison of the items in the edit scripts.

Approaches that split larger code bases into smaller code fragments are closely related to \textbf{C3} because they use similar algorithms and also work on source code parts. Wiggerts [319] gives an introduction to the topic and discusses several approaches up to 1997. The approaches include hierarchical agglomerative clustering that \textbf{C3} also uses. Sec. 5.3 compares different clustering categories and discusses their suitability for \textbf{C3}. Anquetil and Lethbridge [14] as well as Maqbool and Babri [226] compare different hierarchical clustering algorithms and analyze their usefulness for the decomposition of large projects. Andritsos and Tzerpos [12] present LIMBO, a hierarchical clustering approach that uses weights to combine different attributes for a better decomposition. \textbf{C3} currently uses only the similarity of the string representations as single attribute and thus does not require such a combination approach. All the decomposition strategies have in common that they split a component into similar code fragments. However, ARES not only requires similar fragments but also similar or identical changes to these fragments. Thus, it is impossible to use such approaches directly. It is still necessary to search inside the history of projects for similar changes.

The work by Beyer and Noack [40] uses co-change graphs to display related code fragments. The nodes in the co-change graph represent code fragments, the edges represent commits. If an edge connects two code fragments, both were changed within the same commit. The system by Beyer and Noack applies an energy model on this co-change graph. This model changes the distances between nodes and draws nodes that share several edges closer together. This graph can help developers to identify related code fragments. However, it is not an approach that provides an automatic separation of code fragments into
groups. It also does not address code changes. Nevertheless, a similar approach could be used to display the $C^3$ results for an examination by developers or researchers. The approach by Vanya et al. [310] also identifies code fragments that were changed together in the past. It computes pairwise similarities between files and then applies hierarchical agglomerative clustering. The more often a pair of files was changed by the same author in a timespan, the more similar the files are. This structure is similar to $C^3$. However, $C^3$ uses a different similarity function to identify similar code changes. It is more flexible as it also identifies similarities between code changes of items that were not changed together in the past.

Besides the approach by Higo and Kusumoto, there are also several approaches that identify code clones (i.e., similar code fragments). However, whereas Higo and Kusumoto present an approach that relies on code changes, the following approaches use other means. For example, Corazza et al. [67] propose the use of lexical information (e.g., parameter names, class names, comments) to identify similar files. Their approach builds a vector representation for each file. This representation holds the frequencies of lexical tokens in the file compared to the frequency of the tokens in the code base. Then their system uses hierarchical agglomerative clustering on the vectors to find similar files.

CP-Miner by Li et al. [211] identifies similar basic blocks. For that purpose, it converts statements into hash values and represents each basic block as a sequence of hash values. This makes it possible to search for identical subsequences between blocks with frequent itemset mining [4]. CP-Miner works with closed subsequences to identify only the most similar basic blocks. It is possible to use a similar approach in $C^3$ to identify frequent sequences of edit operations. However, this does not work if there are slight differences (e.g., different identifier names) in two edit operations. The string similarity in $C^3$ is more flexible as partly similar edit operations also match.

Yamaguchi et al. [327, 328] detect similar methods based on API usages. Their approach first builds vectors that represent the methods in a code base. In the vectors each item corresponds to a called method, a used type, or a complete subtree up to a predefined depth. For example, if vector item 42 stands for function $foo$ and two methods $m_1$ and $m_2$ have 1 at item 42 in their vectors, both call $foo$. Then the approach uses the Principal Component Analysis (PCA) to reduce the dimensionality of the vector space. Afterwards, methods with similar
5.1 Related Work

function usage lie next to each other in the computed space. The purpose of this work by Yamaguchi et al. is vulnerability detection. As a consequence they assume that developers want to evaluate the similarity graph and they do not propose an automatic clustering in this space. Instead, developers can query the results to get all methods $A_i$ that have a high similarity to a vulnerable method $B$. Yamaguchi et al. further extended their idea of identifying code vulnerabilities [329, 325]. In their more recent approach, their system computes paths from a potential vulnerable method call (e.g., memcpy) to the definition of its input values with static code analysis. Then it searches for similar paths in this dataset with agglomerative hierarchical clustering. For each group of similar paths, it creates a pattern to use it to search for similar code fragments in a Code Property Graph [326]. This is another example for a different use case of similar algorithms. The pattern creation and search for applications in the approach by Yamaguchi et al. is related to ARES. However, ARES also creates recommendations besides providing code locations. It also is not limited to patterns that require similar paths. C3 does not search for similar paths but for similar code changes.

Nguyen et al. [253] presented an approach that also mines software repositories for similar methods. Their approach uses a program dependency graph (PDG) including API calls. Then their approach encodes these PDGs as vectors and looks for identical vectors or identical vector parts. Nguyen et al. found that within a project 12% of the methods appear more than once and thus are repeated. Across projects this number drops to 3%. Still they have shown that there are several methods inside a project that require systematic edits.

Another set of tools uses clustering techniques to untangle changes. Herzig and Zeller [141] found that 15% of analyzed bug fixes contain tangled changes, i.e., changes in the same commit that are unrelated to the bug fix. They propose an untangling algorithm that separates the code changes into parts that are related and unrelated to the bug fix. The algorithm uses different voters that decide whether or not there is a relationship between two changes. Herzig and Zeller proposed 5 voters. One voter examines the distance of the changes in the code file, one voter examines the package distance, and one voter deals with the call graph distance. The fourth voter uses ROSE [342] which provides information about code parts that were frequently changed together in the past. The fifth voter examines the data dependencies between
the code changes and considers the code changes to be unrelated if they do not share variables. Similar to the work by Herzig and Zeller, EpiceaUntangler by Dias et al. [84] untangles commits with the help of voters. However, EpiceaUntangler does not use commits in repositories but events in the IDE recorded by Epicea [85]. ClusterChanges by Barnett et al. [32] also untangles changes. To decompose a code change into unrelated ones, ClusterChanges analyses the definition of code fragments (e.g., fields, methods) and their uses inside the code changes. If there are no shared uses or shared definitions, ClusterChanges splits the code fragments.

Kawrykow [179] introduced DiffCat, an approach that combines CD with partial program analysis (PPA) [77] to divide changes into subtasks. Initially, DiffCat removes non-essential parts from a code change. For example, an insert of this often does not change the semantics of the code. Thus, it is a non-essential change as it only improves the readability. To divide changes into subtasks, DiffCat computes a weighted similarity between changed nodes based on identifier names, data flow dependencies, inheritance relations, and hierarchical dependencies based on declarations and variable uses. With this similarity information, DiffCat builds a graph. The changed AST nodes are the nodes of the graph and the edges hold the weighted similarities. Then DiffCat removes all edges below a threshold. Afterwards, DiffCat returns the subtask which consists of changed AST nodes that are connected by an edge.

All the techniques described above try to untangle changes and do not identify similar changes that were made in different commits. Untangling changes is still an important part for C3. It already divides file changes into method changes. A further untangling into related and unrelated change parts can avoid over-generalized patterns. Patterns from tangled code changes also lead to less accurate recommendations as they contain more wildcards. Such wildcards can hide transformations that are necessary for a complete systematic edit. Thus, including more sophisticated untangling techniques can improve the accuracy of C3 in the future. The disadvantage is an increase in the execution time of C3 as it has to handle more code changes in the similarity matrix.

Fluri et al. [118] presented an approach to identify change types (e.g., insertion/deletion of a parameter) that frequently occur together. Initially, for each commit in a repository, their approach computes the edit operations with CD. Then it assigns a change type to the edit
operation from CD. Fluri et al. defined these change types in their previous work [117]. Central to the approach are feature vectors for each change type. Each element of the vector corresponds to a code change (i.e., a commit). The value of the element specifies how often the change type occurs in each code change. With agglomerative hierarchical clustering, the approach computes a dendrogram in which each change type is a leaf node. Based on a threshold, this dendrogram allows the determination of change types that frequently occur together. An example from the evaluation of Fluri et al. is the pair (return type change, method renaming) that frequently appears together in the data set. In contrast to C3, the goal of this clustering is not the identification of similar code changes but the discovery of change types that occur together. As the change types provide additional information, C3 also uses them in the tree-based representation (see Sec. 5.2.3).

Besides clustering, there are also other approaches that gather information about code changes and commits. Hindle et al. [146] compare different machine learning algorithms in their applicability for categorizing code changes. The categories include “non-functional comment change” or “corrective bug fix”. Dragan et al. [91] add different labels (called “stereotypes”) to commits. Their approach uses static analyses to extract the stereotypes of the methods [92] and combines these method stereotypes into stereotypes of commits. For instance, the approach uses Structure Modifier or Behavior Modifier.

Kim et al. [189] train a support vector machine to identify bug fixes that add new errors to the code. This allows the detection of hidden bugs in the project. The feature vectors that the approach uses contain meta data (like the author) and also parts of the code (like identifier names). In contrast to C3, the trained classifier only determines whether or not a code change is similar to a buggy code change. It does not create groups of similar code changes.

The tool JUnit/CIA by Stoerzer et al. [298] determines whether or not a changed method in a commit improves the quality of the project code. An improvement means that more tests run successfully. Initially, JUnit/CIA executes the test suite. Then it examines each method change and analyzes whether or not it affects a failed test. Based on this analysis, JUnit/CIA reports which method changes likely have caused additional bugs and which have not.

All the information provided by these approaches can help to refine the results of C3 in the future. For example, to avoid patterns that
worsen the quality of the code, \textit{C3} could also use a test suite to examine whether or not a method change affects tests negatively.

Instead of creating groups of similar code changes, many of the discussed state-of-the-art approaches apply mining techniques on code changes for different purposes (e.g., for their untangling). Even if the approaches identify groups of similar code changes, they often only identify groups of nearly identical ones (e.g., CCFinder). The exception is SysEdMiner that uses closed frequent itemset mining on code changes. However, SysEdMiner is still limited to the identification of equal subsequences and thus is less flexible than \textit{C3}.

5.2 Similarity of Code Changes

This section describes how \textit{C3} computes the similarity matrix needed for the clustering algorithms. Source of this step are the commits in code repositories. These commits consist of one or more file changes. However, this is too coarse-grained for ARES as it works on the granularity of method bodies or below (e.g., code blocks, statements). The disadvantage of a granularity below method bodies is that it causes the computational cost to grow. Each changed method consists of at least one or several changed code blocks or statements. Since \textit{C3} computes the pairwise similarities, the computation time grows quadratic with the number of considered code changes. Due to this growth, \textit{C3} uses only method bodies. This constitutes \textit{C3}'s initial task, the extraction of changed methods from repositories. Sec. 5.2.1 discusses the details.

After this initial task, \textit{C3} requires a method to compute the similarity between code changes. In general, clustering algorithms require a distance or a similarity value to compare two input items with each other. To keep the similarity method fast and simple, \textit{C3} relies on a standard string similarity measure based on the LCS. This makes it necessary to convert the method changes into strings beforehand. Other similarity measurements between edit operations are possible. However, as some edit operations contain text (e.g., update operations on identifiers) and also represent source code (e.g., the delete of an assert statement), other similarity measurements still have to compare strings between different edit operations to identify similar ones.

To simplify its approach, \textit{C3} transforms the changes into strings beforehand. As the created strings have a high influence on the clustering results, \textit{C3} currently supports two independent conversions. The
first method converts the method changes into a textual line-based diff representation (see Fig. 3.1). The second one uses the edit script from MTDIFF and encodes it into short strings. Sec. 5.2.2 discusses the line-based representation and Sec. 5.2.3 the tree-based one. The evaluation in Sec. 5.5 discusses their strengths and weaknesses.

The following more formal definition of the similarity computation described above clarifies the terms for the subsequent sections. From one or more code repositories, C3 extracts the set C of method changes. Then C3 converts all these changes into one of the two supported string representations. On all possible string pairs, C3 applies a function \( \text{sim}_{C3} : C \times C \rightarrow [0, 1] \) to compute the similarity between the two method changes in the pair. The result is a similarity matrix \( S_{C3} \) of size \( |C| \times |C| \). Sec. 5.2.4 defines \( \text{sim}_{C3} \) and explains filters that C3 uses to speed up the computation of \( S_{C3} \).

5.2.1 Identification of Changed Methods

There is no straightforward solution that splits file changes into parts that affect only a single method pair. Modifications of method signatures, reordering of methods, insertion of methods, and the deletion of methods can all happen in the same file change. Additionally, edit scripts can also move code parts from one method to another. All these changes make it difficult to identify pairs of methods \((M_o, M_m)\) from the original and the modified file that belong together. Pairing the wrong methods leads to large edit scripts that do not reflect the actual changes that were made in the file.

For these reasons, C3 relies on heuristics to identify method pairs. The heuristics use the results of MTDIFF as input. Thus, for each file change between a file \( F_o \) before the commit and a file \( F_m \) after the commit, C3 executes MTDIFF on \((F_o, F_m)\). Then it uses several heuristics to create pairs \((M_o, M_m)\) of changed methods based on the resulting edit script. C3 applies the method identification heuristics iteratively in the following order:

1. Add the pair \((M_o, M_m)\) to the result if the nodes that represent their method bodies form a pair \((\text{body}_o, \text{body}_m)\) in the mapping of \((F_o, F_m)\). This pairing is immune against signature changes. Additionally, MTDIFF adds \((\text{body}_o, \text{body}_m)\) to the mapping if they share similar statements. Using the method bodies reduces the risk that the edit script between \( M_o \) and \( M_m \) contains edit operations
5 Clustering of Code Changes

that are not part of the edit script between $F_o$ and $F_m$. For example, this could happen if $C_3$ would use only the signature. Thus $C_3$ would pair the two methods $m_{1o}$ and $m_{1m}$ together although the code from the method body of $m_{1o}$ was moved to the body of $m_{2m}$. This would create several delete operations in the edit script between $m_{1o}$ and $m_{1m}$ although the file change between $F_o$ and $F_m$ contains only a single move.

2. For all remaining methods that are not yet part of the result, add the pair $(M_o, M_m)$ to the result if the nodes that represent their signature form a pair $(\text{sig}_o, \text{sig}_m)$ in the mapping of $(F_o, F_m)$. This identifies methods with identical signatures and many modified statements.

3. For all remaining methods that are not part of a pair after the previous selections, add the pair $(M_o, M_m)$ to the result if the method names are identical and unique. This helps identifying methods with signature and statement changes. It is still not a solution for all occurring cases, as the uniqueness requirement prevents its use for overloaded methods. Nevertheless, a selection of methods with an identical and unique name reduces the risk of method pairs that lead to edit operations that are not part of the edit script between $F_o$ and $F_m$.

4. For all remaining methods that are not part of a pair after the first three heuristics, add the pair $(M_o, M_m)$ to the result if they are at the same position within a file. Thus, if $M_o$ is the $i^{th}$ method of $F_o$ and $M_m$ is also the $i^{th}$ of $F_m$, $C_3$ adds the pair to the result. This also works for overloaded methods with changed signatures and statements. However, it cannot handle reordered methods correctly. Thus, there are some cases in which $C_3$ is unable to detect suitable method pairs.

The heuristics automatically discard deleted or inserted methods. This is an advantage because such changes are useless for ARES. Deletion or insertions of complete methods changes the class whereas ARES only works on method body changes. As MTDIFF also moves code parts between methods to reduce the size of the edit script, it is possible that the heuristics select method pairs $(M_o, M_m)$ that are identical, but some of their nodes are changed in the edit script between $F_o$ and $F_m$. To exclude all such pairs, $C_3$ applies MTDIFF to each pair $(M_o, M_m)$. If
5.2 Similarity of Code Changes

The resulting edit script is empty, $C3$ removes the pair from the result. The use of MTDIFF instead of a textual comparison has the advantage that it ignores whitespace or comment changes. After removing the unchanged methods, $C3$ passes the remaining pairs $(M_o, M_m)$ on to its next step for further processing.

5.2.2 Line-based Representation

To measure the similarity of code changes with the LCS algorithm, $C3$ has to convert the changes into a string beforehand. The string of the line-based representation consists of the diff output of the method change. Chapter 3 provides a more detailed description of diff. To get the line-based diff representation, $C3$ executes the diff command provided by Git [132] on the complete file pair $(F_o, F_m)$. The context is set to zero lines, thus only the changed code lines appear in the output of the diff execution. Then $C3$ searches the deleted lines that belong to $M_o$ and the inserted lines that belong to $M_m$ based on the line numbers of the methods. Both sets of lines ordered by their appearance in the diff output of the complete file form the string representation. $C3$ excludes header lines and line offsets. Using diff on the complete file can be less precise than an execution of diff separately for each method pair because this can create representations that do not reflect the change of the method pair if the file contains reordered methods. However, as $C3$ relies on an external tool to provide the diff output, this reduces the execution time.

An advantage of the line-based diff representation is that the changed lines can contain identifiers that reveal parts of the semantics of a code location. Some research results [204, 24] indicate that identifiers can help programmers in comprehending code better. In such cases, the identifiers actually reflect the semantics of the code. As the line-based representation contains complete identifiers, the identifiers have a high influence on the clustering results. The more similar the identifiers are, the more likely it is that the clustering algorithms add them to the same group. As long as the identifiers are similar and have the same semantics this is an advantage. In other cases, this is a disadvantage. The identifiers $exists$ and $notExists$ have a character-based similarity of 55% although they mean the opposite. Thus, although the semantics are different, the identifiers have similarities above zero which increases the likelihood that the clustering algorithms put changes with $exists$ in the same group as changes with $notExists$. 


Based on diff, the line-based representation also does not encode code movements and thus identical edit scripts that consist of many moves do not necessarily have a similar line-based representation. The move edit operation in a tree-based edit script only specifies the position change and the root node of the moved subtree. Representing a move with delete and insert operations requires two edit operations for each node in the moved tree. As a consequence, the similarity of code changes with code movements no longer depends on the similarity between moved root nodes in such a representation. Instead, the similarity values depend on the similarities of the deleted and inserted children.

Another drawback of the line-based representation is the length of the string representation. It requires more characters than the tree-based representation and thus needs more time to compare two method changes.

5.2.3 Tree-based Representation

An alternative configuration to the line-based one is the tree-based representation. It is based on the previous work by Romstöck [282] and relies on the edit scripts computed with MTDIFF. To exclude method pairs \((M_o, M_m)\) with an empty edit script, \(C^3\) already computes the edit script between \(M_o\) and \(M_m\) during the Identification of Changed Methods. It reuses this edit script for the tree-based representation. Initially, the edit script contains only the four edit operations delete, insert, update, and move. \(C^3\) raises the level of abstraction and replaces these edit operations with higher-level ones that include AST-specific information for a tighter encoding. Afterwards, \(C^3\) encodes the list of edit operations into a string. These strings are the input for the following computation of the similarity matrix. During the complete conversion into encoded strings, \(C^3\) also keeps the order from the edit operations provided by MTDIFF. This also ensures that identical edit scripts have an identical encoding.

To raise the level of abstraction of the edit operations, \(C^3\) first converts the move operations that only reorder nodes inside the same parent into align operations. Thus, \(C^3\) replaces each move of node \(n_o\) to \(n_m\) with an align operation if the pair \((\text{parent}(n_o), \text{parent}(n_m))\) is in the mapping. As a consequence, the edit script no longer contains four, but five basic edit operations.
5.2 Similarity of Code Changes

Then, it replaces the five basic operations with an appropriate higher-level operation if it is available. C3 uses 20 higher-level operations. 16 of these 20 operations were presented by Fluri and Gall [117]. From these 16, five address statements, namely `AlignStatement`, `DeleteStatement`, `InsertStatement`, `MoveStatement`, and `UpdateStatement`. Two higher-level operations address the `final` modifier (`InsertFinal`, `DeleteFinal`). Two other higher-level operations address the accessibility of methods and fields (`IncreaseAccessibility`, `DecreaseAccessibility`) and replace the

<table>
<thead>
<tr>
<th>Original code:</th>
<th>Modified code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void method()</td>
<td>void method()</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>foo = Lib. getObject();</td>
<td>foo = Lib. <code>getObjectOrNull</code>();</td>
</tr>
<tr>
<td>foo.someMethod(23);</td>
<td>if (foo != null) foo.someMethod(23);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

(a) MTDIFF requires 6 edit operations.

(b) AST of the original code.

(c) AST of the modified code.

Figure 5.2.: C3 code change example. Same legend as in Fig. 3.4.
delete or insert of public, protected, and private. RenameMethod replaces an update of the method name. Similarly, UpdateReturnType replaces a change of the return type of a method. Finally, five higher-level operations address parameters, namely the operations AlignParameter, DeleteParameter, InsertParameter, TypeChangeParameter, and the operation RenameParameter. Besides the 16 operations by Fluri and Gall, C3 also uses four novel higher-level operations for variable declarations (DeleteDeclaration, InsertDeclaration, TypeChangeDeclaration, and RenameDeclaration).

Fig. 5.2 holds an example. The example contains a small code change and the corresponding ASTs. One edit operation of the corresponding edit script is the insert of if\(_m\). In the domain of programming languages, if is a statement. Thus, C3 adds this information to the edit script and replaces the insert operation of the if statement with a StatementInsert operation.

To further increase the abstraction of the edit script, C3 builds a forest out of the edit operations. To connect the operations it uses their parent-child relationships. For example, the insert operations of if\(_m\) and !=\(_m\) have a relationship because if\(_m\) is the parent of !=\(_m\). For such inserts with a parental relationship, C3 adds an edge to the forest. If there are multiple children, their order is equal to the order in AST\(_m\). In a similar fashion, C3 handles delete operations based on their order in AST\(_o\). Fig. 5.3(a) shows the resulting forest.

The final part of the conversion process is the encoding of the higher-level edit script into characters. The encoding has to fulfill several requirements. It has to be short to speed up the computation of the pairwise similarity values. It also has to be identical for identical code changes. Finally, it has to reduce the similarity between unrelated parts of the edit script. For example, using delete and insert as names in the encoding is not a good option because both share the characters 'e' and 't'. Thus, using the words delete and insert directly would cause unwanted similarities in the edit script. The same problem occurs for identifiers. Identifiers with different meaning should not have any similarity despite some shared characters.

To avoid the edit operation similarity, C3 encodes each edit operation as single character. The characters 0x01 – 0x05 encode the five basic operations (e.g., 0x01 for insert). The characters 0x06 – 0x19 encode the higher-level edit operations (e.g., 0x06 for InsertStatement). After the encoded edit operation type, C3 adds a character that encodes the
5.2 Similarity of Code Changes

(a) Edit operations as ordered script trees.

```
InsertStatement (if \(m\), Block\(m\), 1)
insert (\(\neq m\), \(\neq m\), 0)
insert (foo\(\neq m\), \(\neq m\), 0)
insert (null\(\neq m\), \(\neq m\), 1)
```

(b) Higher-level edit script.

```
InsertStatement (if \(m\), \(\neq m\), foo\(\neq m\), null\(\neq m\), Block\(m\), 1)
update (getObject\(o\), getObjectOrNull)
MoveStatement (Call\(\beta o\), if \(m\), 1)
```

(c) Encoded edit script.

```
0x06 0x23 0x24 0x25 0xCAFE 0x26
0x03 0xABCD
0x07 0x29 0xAB 0xAA
```

Figure 5.3.: Tree-based representation of the C\(3\) example.

labels of the edit operation node (e.g., 0x29 for the moved Call\(\beta o\)). Currently, C\(3\) uses 96 different labels, one for each AST node label that occurs in the output of MTDIFF. Thus, the characters 0x26 – 0x79 encode the node labels. C\(3\) also encodes trees with more than one edit operation in the forest differently. Instead of encoding each edit operation separately, C\(3\) combines them into one operation. Thus, C\(3\) only encodes one InsertStatement operation for the four inserted nodes in the example (see Fig. 5.3(c)). The final encoded character of the InsertStatement operation of the example specifies the node label of the parent from the inserted node and thus the destination of the insert operation.

The encoding in C\(3\) also takes special care of identifiers to avoid the influence of similar but not identical identifiers with different semantics (e.g., exists and notExists). C\(3\) applies a hash algorithm that converts identifiers and values into a two character long hash. Using this 32-bit hash allows the distinction between different identifier names and still has a 100% similarity for identical ones. For example, foo gets the hash 0xCAFE, getObject gets the hash 0xABCD, ba gets 0x1234, and b gets 0xFFFF. As a consequence, the hash-values of the identifiers ba
and \( b \) have a character similarity of 0. Thus, character similarities of different identifiers no longer influence the edit script similarities. The hash values also make it possible to distinguish between expressions like \( a + a \), \( a + b \), and \( b + b \). An evaluation with a predecessor of \( C3 \) on a small dataset by Romstöck [282] showed that a longer hash with more than 32-bit does not improve the results.

To express the edit scripts of MTDIFF, the encoding also requires the inclusion of code movements. To distinguish between different move operations, \( C3 \) puts the relative position change into the tree-based representation. Fig. 5.2 contains the positions \( p \) of the AST nodes to illustrate the computation. Each position consists of a tuple \((x,y)\), where \( x \) is the position in the list of children of their parents. The value \( y \) holds the distance from the root node. The higher-level edit script in Fig. 5.3(a) contains a move of \( \text{Call}_{\beta_0} \) to \( \text{Call}_{\beta_m} \). The position of \( \text{Call}_{\beta_0} \) is \((1,1)\). The position of \( \text{Call}_{\beta_m} \) in the modified AST\(_m\) is \((1,2)\). The relative position change of this move is its old position \((1,1)\) minus its new position \((1,2)\). This gives a relative position change of \((0,-1)\). To encode the position, \( C3 \) starts with a fixed offset to avoid collisions with edit operations or node labels. For example, \( C3 \) encodes 0 as \( 0xAB \) and -1 as \( 0xAA \). This finishes the encoding of the MoveStatement operation (see Fig. 5.3(c)). Compared to absolute positions, relative positions have the advantage that they are more robust to size differences between \( M_o \) and \( M_m \). Thus, the encoding works if the code change starts at different depth inside the methods and thus increases the similarity of similar code changes.

Fig. 5.3 holds the encoding of the complete edit script from the example. It also shows that the encoding omits some less relevant information. The move operation does not encode the node label. The encoded script also does not contain a character representation of the values specified by the update operation. Additionally, the encoded edit script also does not specify the position of the insert operations. All these omitted parts shorten the edit script and speed up the similarity computation. As \( C3 \) does this for all code changes, identical changes remain identical. These omissions also have the advantage that they increase the similarities between code changes that belong to the same systematic edit. This makes it more likely that the clustering algorithms put them in the same group. If this happens, it increases the variance inside a group of code changes and allows ARES to create more general patterns. In some cases the omissions can have a negative effect if they
increase the similarity of unrelated changes. However, the evaluation of the encoding by Romstöck [282] showed that the omissions are beneficial in more cases.

### 5.2.4 Pairwise Similarity Values

The line-based or tree-based representations are the input for the next step of $C_3$ that computes the similarity matrix $S_{C_3} \in \mathbb{R}^{||C|| \times ||C||}$ for the clustering algorithms. The similarities depend on the representations $r_i$ created from the code changes $c_i \in C$. The similarity function is:

$$
\text{sim}_{C_3}(c_j, c_k) := 1 - \frac{\text{LCS}(r_j, r_k)}{\max(|r_j|, |r_k|)} \leq 1 - \frac{\min(|r_j|, |r_k|)}{\max(|r_j|, |r_k|)} =: \text{sim}^{\text{max}}_{C_3}(c_j, c_k).
$$

It uses the LCS algorithm [38] and divides it by the maximal number of characters in either $r_j$ or $r_k$. Together with the subtraction, this normalizes the similarity to the interval $[0; 1]$, where 1 is a perfect similarity. The similarity also fulfills the two criteria of a similarity function [324] due to the properties of the LCS. It is symmetric as $\text{sim}_{C_3}(c_j, c_k) = \text{sim}_{C_3}(c_k, c_j)$ and it is positive. Due to the symmetric property, it is sufficient to compute only the triangular matrix for the clustering. Additionally, $\text{sim}_{C_3}$ fulfills the triangle inequality $\text{sim}_{C_3}(c_j, c_l) \leq \text{sim}_{C_3}(c_j, c_k) + \text{sim}_{C_3}(c_k, c_l)$ due to the LCS [47]. However, it is not a full metric as it does not fulfill the reflexivity criteria $\text{sim}_{C_3}(c_j, c_k) = 1$ iff $c_j = c_k$ for all representations. The tree-based representation omits some information and also uses hash values that can have collisions. Thus, it is possible that two different edit scripts have the same string representation. This violates the reflexivity. For $C_3$ this is not an issue as its goal is the identification of similar code changes. The missing information only increases the similarity of some changes and makes it more likely that $C_3$ assigns them to the same group.

There are valid alternatives to the LCS-based $\text{sim}_{C_3}$. For example, it is possible to use the LVD. A possible disadvantage of the LVD is the support of character substitutions. Converting a $- \rightarrow +$ in the line-based representation or a delete $(0x01)$ into an insert $(0x02)$ operation can increase the similarity between code changes that do exactly the opposite. The LCS in $C_3$ does not allow such substitutions and thus the pair $(-, +)$ is not part of the determined subsequence. However, only an extra evaluation with the LVD can show the effects of such substitutions. This is future work. Other alternatives are bigrams or n-grams (see Chapter 3). The complexity of them is below that of the...
LCS algorithm with $\mathcal{O}(|r_j| + |r_k|)$. A possible disadvantage of this similarity is that it ignores the order in the string representations. This can make edit scripts too similar that affect a different order of statements. However, similar to the LVD, only a future evaluation can give certainty. A preliminary study of $C_3$ on a small data set by Romstöck [282] showed that the results with the Needleman-Wunsch algorithm [248] are similar to the LCS version with a fixed gap penalty. The use of a variable gap penalty is too time consuming. Other alternatives like the Smith–Waterman algorithm [295] with complexity $\mathcal{O}(|r_j|^2 \cdot |r_k|)$ are also too time consuming.

Crucial for a fast computation of the similarity matrix is not only a fast similarity function. It is also essential to compute only values that influence the clustering results. In a large dataset most code changes have little or no similarity to each other. Thus, for many entries in the matrix $S_{C_3}$ the similarity is almost zero. Computing such entries takes unnecessary amounts of time without improving the clustering results. To avoid the computation of such values, $C_3$ uses $\text{sim}_{C_3}^{\text{max}}$ to approximate the similarity results. The function $\text{sim}_{C_3}^{\text{max}}(c_j, c_k)$ can compute the maximal possible similarity between two code changes $c_j$ and $c_k$ solely based on the length of their string representations $r_j$ and $r_k$. It has a complexity of $\mathcal{O}(1)$.

There are two common filtering mechanisms that can make use of such an upper boundary of the similarity. One is a simple threshold filtering. This filter sets all similarity values to zero if their maximal similarity $\text{sim}_{C_3}^{\text{max}}(c_j, c_k)$ is below a threshold $t$. The alternative is a k-nearest neighbors filtering added by Kreutzer [195] to $C_3$. This filter keeps the highest $k$ similarity values for each node, i.e., it keeps the similarity values to its $k$ nearest neighbors. A preliminary evaluation of $C_3$ on a small dataset by Kreutzer showed that using k-nearest neighbors filtering leads to more useful groups for ARES compared to a threshold filtering. Again, a group is useful for ARES if it does not lead to an over-generalized pattern. Additionally, the evaluation by Kreutzer also showed that a k-nearest neighbors filtering with $k = 1$ leads to the best results. Due to this result, the current version of $C_3$ also uses a k-nearest neighbors filtering with $k = 1$. Thus, instead of computing all entries in the similarity matrix it is sufficient to compute only the nearest neighbors $N(c_j)$ of a code change $C_j$. $C_3$ sets all other similarity
values to zero. Nearest neighbors are all code changes $c_k$ for which there is no other code change $c_l$ that is more similar to $c_j$:

$$c_k \in N(c_j) \iff \forall c_l \in C \setminus \{c_j\} : s(c_j, c_l) \leq s(c_j, c_k).$$

The k-nearest neighbor filtering requires $sim_{C^3}^{max}(c_j, c_k)$ to avoid unnecessary similarity computations during the iterative work of row $j$ in the matrix. For each column $col_k$ in this row, it is only necessary to compute $sim_{C^3}(c_j, c_k)$ if $sim_{C^3}^{max}(c_j, c_k) \geq \max_{[0; k-1]}$. Thus, $C^3$ only computes $sim_{C^3}$ if the upper boundary $sim_{C^3}^{max}(c_j, c_k)$ of $col_k$ is above or equal to the maximum of the columns in the interval $[0; k - 1]$. If the maximum is below it, $C^3$ sets the value to zero. The computation of the rows is also embarrassingly parallel. In combination with the k-nearest neighbor filtering this allows a fast computation of the similarity matrix.

5.3 Clustering

This section discusses the clustering algorithms that $C^3$ uses. Backer and Jain [27] give a general definition for clustering (i.e., cluster analysis): “In cluster analysis, a [set] of objects is split up into a number of more or less homogeneous [groups] on the basis of an often subjectively chosen measure of similarity, such that the similarity between objects within a [group] is larger than the similarity between objects belonging to different subgroups.”

In relation to $C^3$, this definition means that the goal of the Clustering step is to split up the method changes into groups such that the code changes in a group belong to the same systematic edit. $C^3$ uses the similarity $sim_{C^3}$ to ensure that the similarity between method changes within a group is larger than the similarity to other code changes. The cluster analysis in $C^3$ is a partitioning of the $n$ elements of a set of code changes $C$ into $m$ disjoint subsets $G_i$, called clusters or groups:

$$C \supseteq (G_1 \cup G_2 \cup \cdots \cup G_m)$$

$$\forall 1 \leq i, j \leq m, i \neq j : G_i \cap G_j = \emptyset$$

$$\forall 1 \leq i \leq m : |G_i| > 1.$$

These three equations define the clustering of $C^3$. In contrast to other formal cluster analysis definitions in the literature (see Xu and Wunsch [324]), $C^3$ requires that a group contains at least two code
changes. This ensures that each group is a suitable input for ARES which requires at least two changes. As there exist code changes that are different from all others in a dataset, $C_3$ also does not require that each code change is part of a group $G_i$. Thus, the changes that form groups with only one member are called outliers.

The definition already shows that the needs of $C_3$ differ partially from standard use cases. In fact, $C_3$ has three requirements that a clustering approach has to fulfill to provide good results. The first requirement is the acceptance of outliers. A clustering algorithm has to be robust against outliers and has to exclude them from every group if necessary. The second requirement is a low time complexity as the datasets for $C_3$ can be large. Otherwise, it is impossible to run on $C_3$ on large datasets in an acceptable time frame. Finally, a suitable algorithm for $C_3$ has to work on different datasets without manual configurations. In general, clustering algorithms have one or more parameters that influence the results and often depend on characteristics of the input data. Setting these parameters manually for $C_3$ would require additional effort from developers. Finding suitable parameters manually requires also a deep knowledge about $C_3$ and ARES. This is usually not the case. Instead, $C_3$ uses only clustering techniques that allow an automatically determination of the parameters based on the similarity matrix.

Fahad et al. [111] give a taxonomy of clustering algorithms. They distinguish between the five categories Hierarchical-based, Model-based, Partitioning-based, Density-based, and Grid-based clustering algorithms. In a preliminary work on $C_3$, Kreutzer [195] compared several approaches from these five categories on a small dataset. The main criteria in his evaluation was the number of groups that lead to useful ARES patterns. Kreutzer considered a pattern useful if it consists of more than a single wildcard. This evaluation was essential in the selection of the two algorithms that $C_3$ currently supports.

From the Hierarchical-based category, the preliminary work evaluated Agglomerative Hierarchical Clustering [336] with single-linkage. From the different available versions, Kreutzer selected single-linkage as it is faster than other hierarchical clustering implementations (see Sec. 5.3.2). It is one of the two clustering algorithms that $C_3$ supports as it fulfills all requirements and identified many groups that ARES can combine into useful patterns.

The preliminary work by Kreutzer also evaluated the Markov Cluster Algorithm [309] from the Model-based category. This algorithm often
created large groups that ARES combines into over-generalized patterns. The Markov Cluster Algorithm also has a high time complexity of $O(n^3)$, where $n$ is the number of code changes. Due to these two reasons, it is not part of the cluster algorithms that $C3$ supports.

Whereas some algorithms like the above-mentioned Agglomerative Hierarchical Clustering and Markov Cluster Algorithm can work directly on the similarity matrix, others require low dimensional spatial data. This makes it necessary to embed the similarity matrix into a d-dimensional space. To optimize the results of the clustering, it is necessary that this embedding respects the similarity values in the matrix and puts code changes with a high similarity next to each other. Generating such an embedding is time consuming. The approach that $C3$ uses has a time-complexity of $O(n^3)$. Sec. 5.3.3 discusses the details.

From the Partition-based category, the preliminary work evaluated k-Means [336] as it is the best-known squared error-based clustering algorithm [324]. The evaluation revealed several disadvantages of k-Means. The clustering algorithm forces each item into a group and thus does not handle outliers in a suitable fashion. Kreutzer showed that this leads to fewer useful groups since groups with outliers often lead to over-generalized patterns.

From the Density-based category, Kreutzer selected DENCLUE [147]. It has the advantage that it handles outliers in the input data. However, there is currently no heuristics that configures the parameters for different datasets automatically. Although the heuristics by Gan and Li [128] gave suitable configurations in the evaluation by Kreutzer, their computation was too time consuming even on small datasets. An alternative from the same category is DBSCAN [109]. For DBSCAN, Kreutzer also found suitable heuristics that compute the parameter automatically. Therefore, DBSCAN is part of the supported clustering algorithms of $C3$. Sec. 5.3.3 contains a detailed description of the algorithm.

In general, Grid-based algorithms also work on spatial data and thus require an embedding. Due to the high complexity of the embedding computation in $C3$, this destroys their greatest advantage, the fast processing time [111]. As a consequence, the preliminary evaluation of $C3$ did not consider Grid-based algorithms. However, it is possible that some grid-based clustering algorithm like OptiGrid [148] work without an embedding. To analyze these alternatives is future work.
To further lower the execution time of the clustering algorithms, C3 also includes a preprocessing step that splits the dataset into smaller subsets. This is the topic of the following section.

### 5.3.1 Preprocessing

It is unnecessary to run the clustering algorithm on the whole dataset at once. Due to the k-nearest neighbor filter on the similarity matrix, the dataset consists of several connected components. Fig. 5.4 shows a similarity matrix $S_{C3}$ and the graph it induces. The example contains three connected components, one with the nodes $(c_1, c_2, c_3)$ and one with the nodes $(c_4, c_5, c_6)$. The edges $E$ hold the similarity values from the matrix. Thus, an edge $e_{jk} \in E$ from $c_j$ to $c_k$ holds the weight $\text{sim}_{C3}^{\text{max}}(c_j, c_k)$. Nodes that have a similarity of 0 have no connection and thus no edge in the graph. Reflexive edges with a similarity of 1 are irrelevant for the clustering and are not displayed. As there is no edge in the graph that connects nodes from different components, the similarity of nodes from different components is low. It is likely that combining code changes from different connected components leads to the creation of an over-generalized pattern. Instead, C3 executes the clustering algorithms for each connected component separately. This has also the advantage that it lowers the execution times of the clustering step as the complexities of both clustering algorithms grow superlinearly.

However, large connected components remain a problem. In large datasets there are connected components with over 10,000 nodes. Applying the clustering algorithms and especially the embedding with $\mathcal{O}(n^3)$ is still too time consuming for a connected component of this size. As a solution to this issue, Kreutzer [195] integrated the

Figure 5.4.: A similarity matrix and the graph it induces.
5.3 Clustering

Kernighan-Lin [183] algorithm in C3. With this algorithm, C3 splits connected components into smaller ones until each connected component has less than 10,000 nodes. The threshold of 10,000 was also determined in the preliminary work on C3 by Kreutzer [195].

The Kernighan-Lin algorithm divides a graph into two equally sized parts by removing edges. As it predominantly removes edges of low similarity, this has a low impact on the clustering results in general. Fig. 5.5 shows the pseudo code for the Kernighan-Lin algorithm. It splits the connected component CC into two smaller components CC1 and CC2. Lines 2–8 create an initial separation of the nodes into CC1 and CC2. Whereas Kernighan and Lin proposed a random distribution of the nodes, C3 uses the initialization proposed by Kreutzer [195] as this leads to a faster termination of the Kernighan-Lin algorithm. Instead of a random selection, it iterates over the pairs in the connected

```plaintext
function KERNIGHANLIN(CC, S3)

01: Pairs ← getPairs(CC)
02: CC1 ← ∅; CC2 ← ∅
03: for (c1, c2) ∈ Pairs do
04:     if getSim(c1, c2, S3) = 0 then
05:         if c1 /∈ CC1 ∧ c2 /∈ CC2 ∧ c1 /∈ CC1 ∧ c2 /∈ CC2 then
06:             CC1 ← CC1 ∪ {c1}; CC2 ← CC2 ∪ {c2};
07:     distributeRemainingNodesEvenly(CC1, CC2, Pairs)
08: V ← 1
09: while V > 0 do
10:     CC1' ← CC1; CC2' ← CC2
11:     E ← ∅; I ← ∅
12:     for c ∈ nodes(CC) do
13:         e ← externalWeight(c, CC1, CC2, S3); i ← internalWeight(c, CC1, CC2, S3)
14:             E ← E ∪ (c, e); I ← I ∪ (c, i)
15: X ← []
16:     E' ← E; I' ← I
17:     while CC1' ≠ ∅ ∧ CC2' ≠ ∅ do
18:         (c1, c2) ← choosePair(CC1', CC2', S3, E', I')
19:         E' ← updateExternalWeight(c1, c2, S3); I' ← updateInternalWeight(c1, c2, S3)
20:         e1' ← getV(E', c1); e2' ← getV(E', c2);
21:         i1' ← getV(I', c1); i2' ← getV(I', c2);
22:         x ← (e1' − i1') + (e2' − i2') − 2 · getSim(c1, c2, S3)
23:         X ← addElementToList(X, (x, c1, c2))
24:     (k, V) ← findMaximalSum(X)
25:     if V > 0 then
26:         for i ∈ 0..k do
27:             (x, c1, c2) ← getElement(i, X)
28:         CC1 ← CC1 \ {c1}; CC2 ← CC2 \ {c2}.
29:         CC1 ← CC1 ∪ {c2}; CC2 ← CC2 ∪ {c1}
30: return (CC1, CC2)
```

Figure 5.5.: Kernighan-Lin algorithm. From the work by Kreutzer [195].
component and puts $c_j$ into $CC_1$ and $c_k$ into $CC_2$ if both have a similarity of 0 (line 5) and both are not part of either $CC_1$ or $CC_2$ (line 6). Then $C_3$ distributes the remaining nodes randomly so that the pairs $CC_1$ and $CC_2$ have equal size. If $CC$ has an uneven number of nodes, the Kernighan-Lin algorithm adds a pseudo node that is removed at the end of the algorithm (not shown in Fig. 5.5).

The algorithm switches nodes between $CC_1$ and $CC_2$. This happens as long as the change increases the sum of the weights from edges that connect nodes in the new components and decreases the sum of weights from edges that connect nodes between $CC_1$ and $CC_2$. For that purpose, the algorithm computes an external and internal weight $i$ for each node (lines 13–15). The external weight $e$ for a node $c$ is the sum of all weights on edges $(c, c_k)$ that connect $c$ with a node $c_k$ in the other connected component. Analogously, the internal weight for a node $c$ is the sum of all weights on edges $(c, c_k)$ that connect $c$ with a node $c_k$ in the same connected component. Lines 18–26 identify suitable node pairs $(c_1, c_2)$ that decrease the external weights. The loop examines all nodes in the connected components. For choosePair, $C_3$ uses a fast version proposed by Kernighan and Lin [183]. Compared to the normal variant, it only examines the edges of 20 nodes instead of all edges. This reduces the execution time of the algorithm. The choosePair method in $C_3$ first identifies 10 nodes in $CC_1$ and 10 nodes in $CC_2$ that have the highest difference between their external and internal weights. Then it examines all edges of these 20 nodes. It looks for the edge $(c_1, c_2)$ between $CC_1$ and $CC_2$ that allows the highest reduction of the external weight. To estimate this value it computes $x = (e_1 - i_1) + (e_2 - i_1) - 2 \cdot \text{sim}_{C_3}(c_1, c_2)$. To access the internal and external values, $C_3$ uses getV. The best candidate edge is the edge with the highest $x$. After choosePair provides the pair $(c_1, c_2)$ with the highest $x$, $C_3$ updates the weights of all nodes that have an edge to $c_1$ and $c_2$ (lines 20–21). This update assumes that $c_1$ and $c_2$ switched place between $CC_1$ and $CC_2$.

After examining all nodes in $CC_1$ and $CC_2$, the algorithm has a list $X$. Each element in this list gives the potential weight reduction $x$ that occurs after $c_1$ and $c_2$ have switched places. To identify the best pairs to change, the algorithm looks for the index $k$, where $V = \sum_0^k x_i$ is maximal. The values $x_i$ are the weight reductions in $X$. If the maximum $V$ is greater than zero, it is possible to further optimize the current split into $CC_1$ and $CC_2$ (lines 28–31). To optimize a solution from a previous
5.3 Clustering

Agglomerative Hierarchical Clustering \([336]\) is a bottom-up approach that merges groups together in a stepwise fashion. Fig. 5.6 holds the pseudo code of the algorithm. Initially, all nodes in the connected component form a group of their own (lines 4–5). The loop in lines 6–20 merges groups until there are no longer any clusters that have a similarity above \(\gamma\) (lines 16–17). As a metric to compute the similarity between groups, \(C_3\) uses the so-called single linkage. In this metric, the similarity between two groups \(G_1\) and \(G_2\) is equal to the highest similarity between a node in \(c_1 \in G_1\) and a node \(c_2\) in \(G_2\):

\[
sim_{SL}(G_1, G_2) := \max\{ \sim_{C_3}(c_1, c_2) \mid c_1 \in G_1 \land c_2 \in G_2 \}.
\]
In Fig. 5.6, lines 12–15 compute this similarity value. If the computed edge \( \text{max} \) between two groups \( \text{CMax}_1 \) and \( \text{CMax}_2 \) has a similarity value above \( \gamma \), \( \mathcal{C}3 \) merges the two groups (lines 18–20). The use of single linkage has the advantage that it lowers the complexity of the clustering compared to other similarity computations between groups. In general, the agglomerative hierarchical clustering has a complexity of \( \mathcal{O}(n^3) \). However, due to single linkage, the agglomerative hierarchical clustering becomes equivalent to the search of a maximum spanning tree of the connected component [134]. This makes it possible to compute the clustering with Kruskal’s algorithm [198]. The algorithm has a complexity of \( \mathcal{O}(e \cdot \log(n)) \), where \( e \) is the number of edges in the graph. Applied to \( \mathcal{C}3 \), the computation of the spanning tree just has to ignore all edges with a weight below \( \gamma \). As groups with only one member do not form a systematic edit, \( \mathcal{C}3 \) discards them (lines 21–23 in Fig. 5.6).
Fig. 5.7 gives an example for the agglomerative hierarchical clustering. The dendrogram illustrates the different merge steps of the algorithm. The first merge happens between \( c_4 \) and \( c_5 \) as they share the highest edge in the spanning tree. In the next iteration, Kruskal’s algorithm adds the edge between \( c_3 \) and \( c_4 \) to the spanning tree and creates a group with three nodes. With a \( \gamma \) of 0.9 this remains the only group in the result set. \( C3 \) discards \( c_1 \), \( c_2 \), and \( c_6 \) as outliers.

Essential for an automatic execution of this clustering algorithm are the heuristics created by Kreutzer that compute \( \gamma \) automatically. A description of them is available in the initial works on \( C3 \) [195, 196]. To create additional groups usable for ARES, \( C3 \) uses an iterative approach. Each iteration uses a different \( \gamma_i \). The first iteration starts with the complete connected component. \( C3 \) computes \( \gamma_1 \) for this graph and applies the hierarchical clustering to identify the groups for this \( \gamma \) value. Afterwards, it removes the groups that have more than one member. Then it uses the \( \gamma \)-heuristics to generate \( \gamma_2 \) for this smaller graph. With the new \( \gamma_2 \), it applies the hierarchical clustering algorithm again on the remaining graph. This continues until the clustering no longer identifies groups with more than one member for a threshold \( \gamma_k \). In the example in Fig. 5.7, \( \gamma_1 = 0.9 \) identifies the group \( (c_3, c_4, c_5) \). \( C3 \) removes it from the connected component and creates a smaller graph (see Fig. 5.7(c)). With \( \gamma_2 = 0.6 \) (computed with the heuristics), \( C3 \) also identifies the group \( (c_1, c_2) \) in this graph. This iterative process produces more groups that lead to useful patterns than a single \( \gamma \). A single \( \gamma \) value is too limited as there is not a single optimal value for all types of code changes. Due to the different sizes of the edit scripts, one differing edit operation can have a small or a large impact on the similarity values. For example, if two edit script have only two input code changes, one differing edit operations between the two scripts would lower the similarity value to 50%. If two edit scripts have 1,000 edit operations, a single differing operation does not have such a large impact. To take groups with fewer edit operations into account, the iterative process is necessary. Using a single \( \gamma \) with a low value instead of the iterative process is not an option as this would lead to many over-generalized patterns.

### 5.3.3 DBSCAN

The DBSCAN [109] algorithm clusters items based on their position in a \( d \)-dimensional space. Using the similarity matrix directly for
Clustering of Code Changes

DBSCAN does not lead to useful results as it is not suited for such high-dimensional data. Thus, $C3$ embeds the code changes $c_i$ of each connected component separately into a $d$-dimensional space. This lower dimensional space allows the use of DBSCAN. It assigns nodes to the same group if they have a small distance to each other. Therefore, it is important that the nodes with a high similarity to each other are also close in the $d$-dimensional space. In contrast, nodes with a lower similarity have to be separated to avoid groups that cause over-generalized patterns. The *Laplacian Eigenmap* [36] gives an embedding that fulfills this requirement. To compute $d$ for this algorithm, it is possible to apply the *Eigengap heuristics* [219] to the similarity matrix. This returns a suitable $d$ for $C3$. However, the preliminary evaluation of $C3$ on small datasets showed that a 2-dimensional space is a good choice for $C3$. It leads to more useful groups than the use of $d$ from the *Eigengap heuristics*.

The *Laplacian Eigenmap* assigns a position $y_i \in \mathbb{R}^d$ to each code change $c_i$. To place nodes next to each other, it minimizes the *Euclidean distance* between similar nodes. It converts $S$ into a matrix $Y$ of size $|C| \times d$, where $|C|$ is the number of code changes in the connected component. Each row $i$ in $Y$ gives the coordinates $(y_1(i), y_2(i))$ of a code change $c_i$ in the 2-dimensional space. To optimize the positions for the clustering, the *Laplacian Eigenmap* algorithm minimizes the weighted distance between nodes. In $C3$, the weights are the similarity values. For an optimal embedding, the algorithm has to solve the following equation:

$$
\min \left( \sum_{i,j} \| y^i - y^j \|^2 \cdot \text{sim}_{C3}(c_i, c_j) \right) = \min(\text{tr}(Y^TLY)).
$$

In this equation, $\text{tr}$ denotes the trace of the matrix. The term $\| y^i - y^j \|^2$ is the euclidean distance between $y^i = [y_1(i), y_2(i)]^T$ and $y^j = [y_1(j), y_2(j)]^T$. Finding $\arg\min \text{tr}(Y^TLY)$ gives the desired positions for the matrix $Y$. The minimal solution also has to satisfy the side condition $Y^T D Y = I$. This condition is necessary to ensure that the algorithm does not map all nodes to the same point. According to Belkin and Niyogi [36], the eigenvectors $Y$ corresponding to the lowest eigenvalues of the generalized eigenvalue problem $LY = \lambda DY$ provide the solution to the minimization problem ($\arg\min \text{tr}(Y^TLY)$). $D$ in this eigenvalue problem is the diagonal similarity matrix with the
5.3 Clustering

Definition $D_{i,i} = \sum_j \text{sim}_{C3}(c_i, c_j)$. $L = D - S$ is the Laplacian matrix and gives the algorithm its name. The eigenvectors to the $0 = \lambda_0 \leq \lambda_1 \leq \cdots \leq \lambda_{n-1}$ eigenvalues solve the equation of the optimization problem. $C3$ selects the $d = 2$ eigenvectors that belong to the two smallest distinct, non-zero eigenvalues as column vectors of $Y$.

Spectral Clustering [293, 219, 336] approaches also use embeddings based on eigenvectors. However, they differ in the Laplacian matrix they use. They also combine the embedding with a clustering technique. For example, the Shi-Malik algorithm [293] uses the symmetric normalized Laplacian to compute the eigenvectors. Then it uses the eigenvectors iteratively to bipartition the graph into groups. Such an algorithm is not suited for the use case of $C3$ as it does not handle outliers.

Instead, $C3$ uses the embedding as base for the DBSCAN algorithm. The DBSCAN algorithm divides the data points into core-points, reachable points, and outliers. A point is a core-point if it has at least $\rho$ other points within the Euclidean distance with value $\epsilon$ (also called $\epsilon$-neighborhood). A point that is density-reachable from a core-point is a reachable point [109]. All other points are outliers. As $C3$ accepts groups of size two, each point only requires another point within the euclidean distance $\epsilon$ to become a core-point. This means that $\rho = 1$. This condition divides the points into core-points and outliers. There are no reachable points for $\rho = 1$.

```
01: function DBSCAN(CC, Y)
02:     nodes ← getNodes(CC)
03:     GRes ← ∅
04:     marked ← ∅
05:     for n ∈ nodes do
06:         if n ∈ marked then
07:             continue
08:         if |neighbors_\epsilon(n, Y)| = 0 then
09:             continue
10:     marked ← marked ∪ n
11:     G ← ∅
12:     W ← neighbors_\epsilon(n, Y)
13:     while |W| > 0 do
14:         n' ← getAnyElement(W)
15:         W ← W \ n'
16:         marked ← marked ∪ n'
17:         G ← G ∪ n'
18:         W ← W ∪ (neighbors_\epsilon(n', Y) \ G)
19:     GRes ← GRes ∪ G
20:     return GRes
```

Figure 5.8.: DBSCAN algorithm for $\rho = 1$. 
Fig. 5.8 holds the pseudo code for the DBSCAN version in \( \mathcal{C}_3 \). The code iterates over all nodes in a connected component \( \mathcal{C} \) (lines 5–19). For each unmarked point that is not an outlier (lines 6–10), \( \mathcal{C}_3 \) creates a group with the neighbors that are within distance \( \epsilon \) (lines 12–18). The algorithm uses a work list \( W \) to create a complete group. Initially, the work list contains all neighbors of \( n \). If these neighbors also have nodes within the \( \epsilon \)-distance they have to be part of the same group and \( \mathcal{C}_3 \) adds them to \( W \) for further processing (line 18).

Whereas \( \rho = 1 \) for \( \mathcal{C}_3 \), \( \epsilon \) is a free parameter. As for the hierarchical agglomerative clustering, Kreutzer introduced heuristics to compute \( \epsilon \) for a connected component [195]. The previous works on \( \mathcal{C}_3 \) hold the details [195, 196]. Similar to the hierarchical clustering heuristics in \( \mathcal{C}_3 \), an iterative computation of \( \epsilon \) is also beneficial for DBSCAN. Fig. 5.9 gives an example. Initially, the heuristics identify an \( \epsilon_1 \) that puts the code changes \( c_3, c_4, \) and \( c_5 \) in the same group. A larger \( \epsilon \) that would also group \( c_1 \) and \( c_2 \) together is not an option. Such a large \( \epsilon \) would add \( c_6 \) to the group (\( c_3, c_4, c_5 \)) and create an over-generalized pattern in the worst case. An iterative application of DBSCAN solves this problem. \( \mathcal{C}_3 \) first uses \( \epsilon_1 \) to identify groups. Then it removes the code changes that are in groups with a size \( \geq 1 \) from the 2-dimensional space (i.e., from \( Y \)). A second iteration of the heuristics provides \( \epsilon_2 \) that puts \( c_1 \) and \( c_2 \) in the same group. The code change \( c_6 \) remains an outlier. \( \mathcal{C}_3 \) repeats this process until an iteration no longer identifies a group with two or more code changes.
5.4 Complexity Analysis

C3 starts with the identification of changed methods in repositories. This step requires the execution of MTDIFF on f file changes. Thus, the time complexity is in $\mathcal{O}(f \cdot \mathcal{O}_{\text{MTDIFF}})$. With this information, C3 extracts $|C|$ method changes. To identify empty edit scripts, C3 executes MTDIFF on all $|C|$ method changes. This gives a combined complexity of $\mathcal{O}((f + |C|) \cdot \mathcal{O}_{\text{MTDIFF}})$.

The line-based representation requires the execution of diff on the file changes. For the $|C|$ method changes this has a time complexity of $\mathcal{O}(f \cdot L \cdot e)$, where $L$ is the number of lines and $e$ is the size of the edit script (i.e., the number of changed lines). This gives a combined time complexity of $\mathcal{O}((f + |C|) \cdot \mathcal{O}_{\text{MTDIFF}} + f \cdot L \cdot e)$. The tree-based representation executes MTDIFF again on the $|C|$ method changes. The encoding is in $\mathcal{O}(e)$, where $e$ is the size of the edit script. This gives a combined time complexity of $\mathcal{O}((f + |C|) \cdot \mathcal{O}_{\text{MTDIFF}} + e)$.

The similarity matrix computation requires the computation of $(|C| \cdot |C - 1|)/2$ comparisons in the worst case. The real number of computations is lower as C3 uses a k-nearest neighbor filtering. Each similarity computation requires the computation of the LCS. C3 computes the LCS with Hirschberg’s algorithm [149]. This algorithm has a time complexity of $\mathcal{O}_H(|r_j| \cdot |r_k|)$, where $|r_j|$ is the number of characters in the string representation of $c_j$ and $|r_k|$ is the number of characters in the string representation of $c_k$. Thus, this part of C3 has a worst case time complexity of $\mathcal{O}(|C|^2 \cdot |r|^2)$, where $|r|$ is the length of the longest string representation in $C$.

Before the actual clustering algorithm, C3 divides the graph into connected components. The identification of connected components is in $\mathcal{O}(\max(|C|, |E|))$, where $|C|$ is the number of method changes and $|E|$ is the number of non-null entries in the similarity matrix [183]. If necessary, C3 also executes the Kernighan-Lin algorithm to split large connected components. Its time complexity lies in $\mathcal{O}(n^2 \cdot \log(n))$, where $n$ is the number of nodes in the connected component. However, as C3 uses a fast version of choosePair and a non-random initialization, the execution time is faster in practice.

Due to the use of single-linkage, the agglomerative hierarchical clustering is equivalent to the computation of the spanning tree. This allows the computation of the clustering results with Kruskal’s algorithm [198]. It has a time complexity of $\mathcal{O}(e \cdot \log(n)) \subset \mathcal{O}(n^2 \cdot \log(n))$, where $e$ is the number of edges in the connected component and $n$ is the number...
of nodes. Thus, it can be faster than the general hierarchical clustering with $O(n^3)$.

As DBSCAN works in a $d$-dimensional space, it requires an embedding of the similarity graph into such a low-dimensional space. $C3$ uses the Laplacian Eigenmap for this purpose. This has a complexity of $O(n^3)$ [293], where $n$ is the number of nodes in the embedded component. In contrast, the actual DBSCAN algorithm has a lower complexity of $O(n^2 \cdot d)$, where $d$ denotes the dimensionality of the target space.

### 5.5 Evaluation

This evaluation follows the structure of its predecessors and answers 5 questions:

1. Is $C3$ fast enough for a practical use on large datasets?
2. Does $C3$ detect known systematic edits in a large dataset?
3. Is there a relationship between groups of $C3$ and code clones?
4. Does $C3$ provide suitable groups for ARES?
5. Does ARES produce accurate recommendations for large datasets?

In contrast to the previous chapters, the measurements used a computer cluster with 64 nodes. Each node was equipped with two 2.66 GHz Xeon 5650 Westmere chips (12 cores and simultaneous multi-threading) and 24 GB of RAM. Like before, the input for the evaluation were 9 open-source Java projects (Ant [15], Checkstyle [60], Cobertura [64], DrJava[93], Eclipse JDT Core [101], Eclipse SWT [102], Fitlibrary [116], JGraphT [168], and JUnit [175]).

As in Chapter 3, the

<table>
<thead>
<tr>
<th>Repository</th>
<th>LOC</th>
<th>Commits</th>
<th>Size</th>
<th>Changed Files</th>
<th>Modified Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cobertura</td>
<td>43,586</td>
<td>335</td>
<td>small</td>
<td>998</td>
<td>3,327</td>
</tr>
<tr>
<td>Fitlibrary</td>
<td>74,366</td>
<td>151</td>
<td>small</td>
<td>2,185</td>
<td>4,350</td>
</tr>
<tr>
<td>JGraphT</td>
<td>33,554</td>
<td>517</td>
<td>small</td>
<td>2,048</td>
<td>3,262</td>
</tr>
<tr>
<td>JUnit</td>
<td>33,871</td>
<td>1,064</td>
<td>small</td>
<td>3,446</td>
<td>4,784</td>
</tr>
<tr>
<td>Ant</td>
<td>246,426</td>
<td>6,340</td>
<td>medium</td>
<td>19,082</td>
<td>33,965</td>
</tr>
<tr>
<td>Checkstyle</td>
<td>132,091</td>
<td>3,001</td>
<td>medium</td>
<td>10,978</td>
<td>18,060</td>
</tr>
<tr>
<td>DrJava</td>
<td>224,863</td>
<td>2,759</td>
<td>medium</td>
<td>16,801</td>
<td>45,289</td>
</tr>
<tr>
<td>Eclipse JDT</td>
<td>1,334,938</td>
<td>15,497</td>
<td>large</td>
<td>53,299</td>
<td>106,697</td>
</tr>
<tr>
<td>Eclipse SWT</td>
<td>378,560</td>
<td>18,373</td>
<td>large</td>
<td>41,473</td>
<td>91,533</td>
</tr>
<tr>
<td>Total</td>
<td>2,502,255</td>
<td>47,947</td>
<td>-</td>
<td>190,310</td>
<td>311,267</td>
</tr>
</tbody>
</table>
5.5 Evaluation

evaluation dataset contains all file changes present in the commits to the master branches of the 9 repositories. The dataset also uses only commits that were made before 2016. Table 5.1 holds the properties of the different repositories. The nine repositories are divided into three artificial categories (small, medium, large) based on their lines of code (LOC) and the number of modified methods. The first four small repositories have fewer than 100,000 LOC and fewer than 10,000 changed methods. The medium sized repositories have fewer than 300,000 LOC and fewer than 50,000 changed methods. The two large repositories have more than 300,000 LOC and over 90,000 changed methods. The execution times analyzed in the next section reflect these differences.

There is also a fundamental difference between this evaluation and the evaluation in Chapter 3. C3 uses the AST design of ARES. This AST differs slightly from the GT-AST. These differences cause a different number of changed files in the C3 dataset, as some changes are not visible in the ARES-AST. For example, the GT-AST and the ARES-AST express \( a + b + c \) differently. In the GT-AST, \( a, b, \) and \( c \) form a single list and are children of a single addition node (+). In the ARES-AST, there are two addition (+) nodes. Both ASTs react differently to a code changes that insert parentheses. Changing the above expression to \((a + b) + c\) does not change the ARES-AST. However, the parentheses change the GT-AST and create a second + node to express the new evaluation order, enforced by the parentheses. Thus, this code change is visible in the GT-AST, but not in the ARES-AST.

The following sections deal with the research questions in detail. They also compare the line-based with the tree-based representation as well as the agglomerative hierarchical clustering with DBSCAN. This gives 4 configurations of C3 that all have strength and weaknesses. After the evaluation of the configurations, this chapter ends with a discussion of the limitations of C3.

5.5.1 Execution Time

This section discusses the execution times of the different parts of C3 and analyses their applicability for practical use on large repositories. Table 5.2 holds the measured times for the similarity matrix computation. The left columns list the times for the line-based representation, whereas the right columns list the times for the tree-based representation. The Work columns hold the sequential execution times whereas the Wall columns list the elapsed real time of the parallel similarity
matrix computation. For the four smaller repositories (Cobertura, Fitlibrary, JGraphT, JUnit), one cluster node with 24 threads was used. For the medium and large repositories, 64 nodes with 24 threads per node (= 1,536 threads) computed the results. The Increment columns contain the median time it took to compute one commit sequentially. The measured times shown in all the columns contain all parts of the similarity computation. They include the time to extract the methods from the repository, the time to create the string representations, and the time to compute the similarity matrix entries. As it is possible to compute the similarity matrix incrementally, the Increment columns are of special interest. After an initial computation of the similarity matrix for the repository history, only the Increment time values are of relevance. In theory, the time per commit grows with the number of commits in the repository. However, due to the k-nearest neighbor filtering there are factors with more influence. The number of changed files inside a commit and the file size have a larger impact. The effect of the number of commits on the Increment time is minor, even for the large repositories. It lies in the range of milliseconds.

Across all repositories, the larger size of the strings from the line-based representations causes a higher CPU time compared to the tree-based representation (1,525 h vs. 102 h in total). The string size has such a large impact because it has a quadratic effect on the LCS computation time. However, the elapsed real times in the Wall columns show a different picture. Both variants have a total Wall time of 17 h. The main reason is the input/output (I/O) overhead that makes the CPU time difference irrelevant. For example, an unavoidable overhead is the extraction of the original and modified file versions from the repository on the hard drive. As C3 is a research tool, it also saves intermediate results

Table 5.2.: Similarity computation times.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Line-based Representation</th>
<th></th>
<th>Tree-based Representation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Work</td>
<td>Wall</td>
<td>Increment</td>
<td>Work</td>
</tr>
<tr>
<td>Cobertura</td>
<td>1.2h</td>
<td>1.1h</td>
<td>0.95s</td>
<td>0.43h</td>
</tr>
<tr>
<td>Fitlibrary</td>
<td>0.50h</td>
<td>0.12h</td>
<td>3.1s</td>
<td>0.09h</td>
</tr>
<tr>
<td>JGraphT</td>
<td>0.52h</td>
<td>0.13h</td>
<td>0.78s</td>
<td>0.21h</td>
</tr>
<tr>
<td>JUnit</td>
<td>0.91h</td>
<td>0.14h</td>
<td>0.64s</td>
<td>0.13h</td>
</tr>
<tr>
<td>Ant</td>
<td>44h</td>
<td>0.37h</td>
<td>2.8s</td>
<td>6.3h</td>
</tr>
<tr>
<td>Checkstyle</td>
<td>12h</td>
<td>1.1h</td>
<td>2.0s</td>
<td>6.5h</td>
</tr>
<tr>
<td>DrJava</td>
<td>13h</td>
<td>3.0h</td>
<td>43s</td>
<td>11h</td>
</tr>
<tr>
<td>Eclipse JDT</td>
<td>928h</td>
<td>6.3h</td>
<td>39s</td>
<td>46h</td>
</tr>
<tr>
<td>Eclipse SWT</td>
<td>407h</td>
<td>4.7h</td>
<td>22s</td>
<td>31h</td>
</tr>
<tr>
<td>Total</td>
<td>1,525h</td>
<td>17h</td>
<td>-</td>
<td>102h</td>
</tr>
</tbody>
</table>
like the edit scripts or the rows of the similarity matrix to the hard drive. This limits the possible speedup in both versions (see Amdahl’s law [8]). The I/O overhead explains also the higher Wall time on the Cobertura repository compared to the actual Work time. Due to the large commits in the Cobertura repository, the CPU was idle several times and had to wait until Git extracted the necessary file changes from the hard drive. Besides the I/O overhead, the static assignment of code changes to threads in C3 also prevented a better speedup. As a consequence, the slowest thread determines the Wall time. In contrast to the other repositories, on the Eclipse JDT repository the tree-based representation was slower than the line-based one. Again the static assignment caused the higher Wall time. Due to chance, one of the threads had several commits with a high MTDIFF execution time. Thus, the Wall time was unnecessarily large.

Whereas Table 5.2 shows only the median times of the commits in the Increment column (including the I/O times), Fig. 5.10 illustrates the complete sequential times per commit, measured without I/O overhead. Each data point in the boxplot belongs to one of the 47,947 commits in the dataset. The data shows that C3 is able to process most commits within minutes. This time includes the extraction of the methods in the commit, the creation of the string representations and the computation of the similarity matrix rows. As seen in the summary above, the tree-based representation requires less CPU time than the line-based representation. To give a fair comparison between line-based and tree-based representations, the tree-based representation did not reuse the MTDIFF results from the extracting method step. Instead, it used MTDIFF to create the edit script for each method change in a commit. Even with this additional overhead, the computation of the tree-based representation is faster than the line-based representation. The line-based representation is slower because it uses an extra process to execute the diff command of Git. This has the advantage that it avoids a reimplementation of diff, but also the disadvantage that starting the extraction process causes a constant overhead.

In the dataset, there are several commits that take longer than 10 minutes across all steps. This is a contrast to the majority of the other measured commits. The main reason is the commit size. A commit in Eclipse JDT with 1,678 file changes caused the highest Extracting Method time with 3.4 h. However, this means that the extraction (including the calls to MTDIFF) takes only 7.3 s per file sequentially.
**5 Clustering of Code Changes**

![Diagram of execution times per commit](image)

**Figure 5.10.** Work times per commit. The boxes show the 25%/75% quartiles, the lines indicate the medians. The lower whiskers are at the minimum, the diamonds show the mean.

This execution time per file is fast enough for a call to C3 with each normal sized commit. Thus, it is unnecessary to update the similarity matrix over night or over the weekend (when CPU cycles are available).

The highest computation time for the tree-based representation is also caused by a large commit. This commit to Eclipse JDT has 1,766 file changes and each of the files contains a large number of modified methods. Due to Myer’s algorithm, the execution time from the line-based representation for huge commits is lower as its complexity is also lower than the one of MTDIFF. However, due to the more compact size of the tree-based representation, the tree-based similarity computation is faster than the line-based one.

Fig. 5.11 shows the execution times of the sequential clustering algorithms. Currently, a parallel implementation of the clustering algorithms and a parallel implementation of the embedding does not exist.

![Diagram of sequential clustering times per repository](image)

**Figure 5.11.** Sequential clustering times per repository. The boxes show the 25%/75% quartiles, the lines indicate the medians. The lower whiskers are at the minimum, the diamonds show the mean.
Each data point in the plot corresponds to a repository. As the goal of this evaluation is the comparison of the hierarchical clustering and DBSCAN, the plot combines the results from the line-based representation and the tree-based representation. Thus, the hierarchical clustering box consists of 18 measurements. Overall, the hierarchical clustering is faster than DBSCAN. The root cause is the high time complexity of the embedding that is a requirement for DBSCAN. Large connected components cause high execution times for both clustering algorithms. 

Overall, the similarity computation in C3 scales well. The current version is fast enough to process a commit immediately and even does not require a computing cluster for single commits. However, to create a similarity matrix from scratch, a computing cluster is preferable.

The clustering algorithms in C3 (especially DBSCAN) are more time consuming. C3 does not support a parallel execution of the clustering and it is also impossible to compute the results of the clustering approaches incrementally. They always have to compute the groups for the complete similarity matrix. Additional research is necessary to find incremental clustering approaches that do not have negative impact on the quality of the C3 results. Thus, as the clustering takes a lot of time, it is necessary to execute it overnight or on weekends in a real-world scenario. As the execution is sequential, a computing cluster is unnecessary. Whereas the time of DBSCAN is sufficient to run on a single large repository, it is too high for a dataset that combines several repositories. With such a large dataset (e.g., the combination of all 9 repositories), it takes DBSCAN weeks to compute the clustering results in the current version. Overall, the similarity computation is fast enough for large repositories, but the clustering algorithms require additional performance improvements.

5.5.2 Relevance

In combination, the four different configurations of C3 (line-based vs. tree-based, hierarchical clustering vs. DBSCAN) create 89,775 + 119,096 + 46,286 + 69,327 = 324,484 groups from the 311,267 modified
5 Clustering of Code Changes

methods. Evaluating such a huge result set manually is impossible. Instead, this section evaluates whether known groups of similar code changes appear in the results. If this is the case, it is also likely that C3 finds other groups of similar code changes that belong together. Again, this thesis uses the manually identified groups of code changes by Meng et al. [236]. In contrast to the ARES evaluation, the bug fix 74139 that ARES could not handle (see Sec. 4.8.5) is part of the C3 evaluation. This increases the total number of code change groups to 24.

Table 5.3 holds the results. Each C3 configuration has an own segment in the table. The columns under LH give the results for the combination of the Line-based representation and the Hierarchical clustering. The columns under LD hold the results from the combination of the Line-based representation with DBSCAN. The columns under TH summarize the results of the Tree-based representation in combination with Hierarchical clustering. Finally, the columns under TD show the results for the Tree-based representation in combination with DBSCAN. For each configuration, Table 5.3 shows the parts of the 24 groups of code changes that the different configurations identified in the 106,697 + 91,533 modified methods from the Eclipse repositories (see Table 5.1). The ✓ columns indicate whether or not the C3 configuration created a group solely consisting of members from the corresponding manually defined group. Thus, if there are code changes inside a group that do not belong to the manually identified set for the corresponding bug fix the complete group is not added to the shown results in Table 5.3. Counting such a group as positive result is not useful as such groups often lead to over-generalized patterns. The △ columns list the number of code changes that are in groups that solely belong to the corresponding manually identified code changes.

A suitable example to explain the △ columns is the bug fix with Id 12. The △ column for Id 12 holds the value 7 for the LH configuration. This means that in the results from the LH configuration there are 7 code changes from the manually defined group with Id 12. It also means that all 7 are in groups without code changes that are not part of the manually defined set. The Groups columns list the distribution of the code changes across groups. Thus, 2;2;3 in the corresponding LH column means that the 7 code changes from this bug fix are found in three distinct groups. Two groups contain two code changes and one group contains three such code changes.
### Table 5.3: Identified bugfix groups from the Eclipse JDT and Eclipse SWT repositories.

<table>
<thead>
<tr>
<th>Id</th>
<th>Bugzilla Id</th>
<th>LH</th>
<th>Groups</th>
<th>LD</th>
<th>Groups</th>
<th>TH</th>
<th>Groups</th>
<th>TD</th>
<th>Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>77044</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>82420</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>114007</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>139329</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>142947</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>91937</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>103863</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>129314</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>9</td>
<td>134091</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>10</td>
<td>139329</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>11</td>
<td>139329</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>12</td>
<td>142947</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>13</td>
<td>97981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>14</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>15</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>17</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>18</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>19</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>20</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>21</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>22</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>23</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>24</td>
<td>197981</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td></td>
<td>24</td>
<td>129</td>
<td>39</td>
<td>116</td>
<td>51</td>
<td>110</td>
<td>35</td>
<td>113</td>
</tr>
<tr>
<td><strong>%</strong></td>
<td></td>
<td>92</td>
<td>80</td>
<td>88</td>
<td>76</td>
<td>88</td>
<td>74</td>
<td>96</td>
<td>76</td>
</tr>
</tbody>
</table>

*m*: # Code Changes for the Systematic Edit; ✓: Systematic Edit Found; △: # Changes in Groups;
The results show that all four configurations identify most of the 24 manually defined groups. \( LD \) and \( TH \) both identify 21 groups, \( LH \) identifies 22 groups, and \( TD \) even identifies 23 groups. The configurations also put most (110–119) of the 149 code changes in groups that solely consist of code changes that are in the same set of manually identified groups. Compared to the other three configuration, \( LH \) puts the most code changes (119 from 149) in groups that belong to a single bug fix. It also identifies the most complete groups. For eight bug fixes (Ids 5, 6, 9, 10, 11, 17, 21, 22), it finds groups that have the same code changes as the manually identified input set. The \( TD \) version, however, creates groups for 23 of the 24 bug fix groups and thus handles one additional systematic edits, compared to the \( LH \) configuration. In general, the hierarchical clustering configurations (\( LH \) and \( TH \)) build larger groups and fewer groups of size two. Thus, they only require 35/39 groups for the 24 manually identified code change groups. Similarly, the groups of the tree-based configurations are slightly larger. Depending on the use case, either one of them is the best choice. Only \( LD \) offers no special advantage based on the data in this part of the evaluation.

There are some particularities in the results. None of the configurations identify any groups for the bug fix with Id 18. In fact, the configurations mix the code changes from this bug fix with others and create groups with over 60 code changes. The reason for this behavior is the small number of edit operations in the edit script to the bug fix with id 18. The script only contains an insert of an if. Such a change is highly similar to all other code changes that insert similar if checks to methods in a similar way. The small size of the edit script also reduces the detection rate for the bug fix with Id 19. This bug fix adds only one or two statements (according to the input examples) to a method and performs no other changes. As a result, the heuristics of the hierarchical clustering are unable to identify a suitable \( \gamma \) that isolates the changes from bug fix with Id 19.

Sometimes, the heuristics split a manually identified group across several automatically computed ones (e.g., Id 2). Such a split happens if there are many differences in the code changes. For a use with ARES this is a minor problem. The patterns for each of the subgroups are less general, but together the patterns express all the variations of the bug fix group they belong to. Only if an edit location requires a pattern from the combination of two subgroups, the use of several groups for the same bug fix fails.
In summary, \textit{C3} can isolate groups of code changes that belong to the same systematic edit even in large datasets. \textit{C3} is even able to identify groups that are equal to the manually identified ones. Thus, \textit{C3} can replace time consuming manual identification of systematic edits in repositories. This also means that \textit{C3} is capable of producing useful input for tools that target systematic edits.

\subsection*{5.5.3 Relationship to Code Clones}

\textit{C3} only uses the similarity between method changes to identify groups. Thus, the clustering results can contain groups with changes that have very different original and modified methods. However, as developers applied similar changes to these methods, it is still possible that the methods are similar and thus all groups of \textit{C3} could be groups of identical or nearly identical methods. This would mean that \textit{C3} only identifies groups of changed code clones.

Roy et al. \cite{284} distinguish between four types of code clones. Type-1 clones are identical code fragments that only have differences in whitespace, layout, or comments. Thus, they have the same ASTs. Type-2 clones are code fragments that have the same ASTs, but their nodes have different values. The definition of Type-3 clones is more vague and requires knowledge about the origin of the similar code fragments. According to Roy et al., Type-3 clones are copied code fragments that underwent different modifications. Thus, they can have different statements. They may also have different node values in their ASTs. Based on this definition, any two code fragments could be clones. To be able to distinguish between Type-3 clones and other non-clone fragments, the origin of the code fragments (i.e., the copy process) has to be known. As this is not known for every changed method in the evaluation, this section only focuses on Type-1 and Type-2 clones. Type-4 clones are syntactically different code fragments that have the same semantics. As both ARES and \textit{C3} currently only work on the syntactical level of code, Type-4 clones are out of scope of this thesis.

The first line in Table 5.4 holds the number of groups that the different configurations extracted from the dataset. The second line shows the number of code changes in these groups and the third line holds the percentage of the code changes compared to the 311,267 method changes in the complete dataset (see Table 5.1). Similar to the results in Sec. 5.5.2, the hierarchical clustering and the tree-based representation both lead to larger groups (\textit{Mean Size} in Table 5.4). The main reason
5 Clustering of Code Changes

Table 5.4.: Identified groups and clone groups in the 9 repositories.

<table>
<thead>
<tr>
<th></th>
<th>LH</th>
<th>LD</th>
<th>TH</th>
<th>TD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups</td>
<td>89,775</td>
<td>119,096</td>
<td>46,286</td>
<td>69,327</td>
</tr>
<tr>
<td>...Code Changes</td>
<td>261,009</td>
<td>282,436</td>
<td>288,411</td>
<td>299,761</td>
</tr>
<tr>
<td>...Code Changes (%)</td>
<td>84</td>
<td>91</td>
<td>93</td>
<td>96</td>
</tr>
<tr>
<td>...Mean Number of Members</td>
<td>2.9</td>
<td>2.4</td>
<td>6.2</td>
<td>4.3</td>
</tr>
<tr>
<td>Type-1 Clone Groups</td>
<td>11,201</td>
<td>11,814</td>
<td>6,147</td>
<td>5,445</td>
</tr>
<tr>
<td>...Code Changes</td>
<td>24,835</td>
<td>24,619</td>
<td>13,769</td>
<td>11,502</td>
</tr>
<tr>
<td>...Code Changes (%)</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Type-2 Clone Groups</td>
<td>14,600</td>
<td>19,545</td>
<td>3,765</td>
<td>5,976</td>
</tr>
<tr>
<td>...Code Changes</td>
<td>36,430</td>
<td>42,504</td>
<td>10,803</td>
<td>14,607</td>
</tr>
<tr>
<td>...Code Changes (%)</td>
<td>12</td>
<td>14</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Other Groups</td>
<td>63,980</td>
<td>87,740</td>
<td>36,385</td>
<td>57,919</td>
</tr>
<tr>
<td>...Code Changes</td>
<td>199,996</td>
<td>215,434</td>
<td>264,301</td>
<td>274,034</td>
</tr>
<tr>
<td>...Code Changes (%)</td>
<td>64</td>
<td>69</td>
<td>85</td>
<td>88</td>
</tr>
</tbody>
</table>

for the larger groups of the tree-based representation is that identifiers have less influence on the results, as they are only encoded in two characters. This increases the similarity of code changes with different identifiers. Additionally, code that is unrelated to the change is not part of the tree-based representation. As the line-based representation contains complete lines the unchanged context in these lines is also part of the string representation and thus all the identifiers of these lines are part of the representation. This increases the similarity of code changes with identical context and similar identifiers.

As the tree-based representation and the hierarchical clustering both lead to larger groups, the combination TH has the fewest groups of the four configurations. Despite this, these groups still contain more code changes (i.e., 288,411) than the results from the LH (261,009) and LD (282,436) configurations. The exception is TD that has even more code changes (299,761) in its groups than the TH configuration. TD also has fewer groups than LH and LD. Overall, the numbers do not give a real indication on which configuration is better. It depends on the use case, whether larger groups are preferable or whether it is more important to maximize the total number of code changes in groups. The next section examines how these differences affect the recommendation results of ARES.

Table 5.4 also lists the number of groups that consist of Type-1 or Type-2 clones. In a Type-1 group, all code changes have an original method with an ARES-AST that is identical to the ARES-ASTs of the other methods. Similarly, Type-2 groups have original methods where the ASTs are identical except for their values. All four configurations
built groups with Type-1 and Type-2 clones. Again, the mean size of the groups influence the results. The tree-based representation as well as the hierarchical clustering create larger groups, which makes it less likely that all their members are code clones. However, most groups in the results do not solely consist of Type-1 or Type-2 clones. Instead, they have original methods that differ in at least one AST node. As such groups are harder to detect and lead to more general patterns, they are more valuable. They can also lead to recommendations for code fragments that are Type-3 clones or no clones at all.

The groups with Type-1 and Type-2 clones reveal methods that are not only code clones but were also changed in identical ways. This means that they are candidates for refactoring. Removing them, prevents the regular overhead of applying the same systematic edits to all members of the groups.

In summary, there is a connection to code clones, but it is not a strong one. Although C^3 detects some groups that only contain changes to Type-1 and Type-2 clones, the vast majority of the detected groups contain code changes to methods with more variety.

### 5.5.4 Case Study ARES

There are fast algorithms that detect Type-1 and Type-2 clones with token-based or hash-based techniques. Tools like Clever by Nguyen et al. [254] can also keep track of changes to code clones. As the transformations on such clones are identical there are also tools that explicitly handle such identical changes. An example is the simultaneous editing approach by Miller and Myers (see Sec. 2.5).

Identifying systematic edits that change code locations with more variance is more difficult and requires tools like C^3. Code changes to Type-1 and Type-2 clones also pose no challenge for ARES as they neither require wildcard nor use annotations.

Thus, this part of the evaluation focuses on the Other groups from above that do not solely contain Type-1 or Type-2 clones. To identify whether or not these groups belong to the same systematic edit, they were used as input for ARES to create patterns. In contrast to other parts of this thesis, all group members were used as input. The evaluation discarded over-generalized patterns and used all remaining patterns to search for code applications with ARES. To get the code base for the application search, the evaluation identified the revision $r$ that contains the oldest code change in each group. For the search, it used
the predecessor of this revision \( r \). Thus, if \( c_1 \) is the oldest change in a group of code changes \( (c_1..c_n) \) of pattern \( p \) and \( r_1 \) is the revision of \( c_1 \), the most recent revision before \( r_1 \) was used as code base for the search. If this search leads to one or more recommendations for the code locations of \( (c_1..c_n) \), all changes probably belong to the same systematic edit. Thus, if \( \mathcal{C}3 \) detected such a group, it is is rated *useful*. As this group allows ARES to create suitable recommendations it is useful for the execution of systematic edits. However, whether or not a group actually belongs to a systematic edit has to be decided by human judges.

The definition of *useful* requires only that a pattern identifies one location that belongs to any of the code change in the group that created it. It does not require that a pattern identifies all code locations of a group. The reason for this less strict definition is the incompleteness of the rule-based system. Thus, in some cases, ARES creates fewer *wildcards* than necessary to identify all \( (c_1..c_n) \) code changes. Additionally, it is possible that a code change \( c_n \) was made years after \( c_1 \) and the original method of \( c_n \) is not part of the revision that is used as code base. As the method is not present, ARES is unable to create recommendations for it. Applying ARES to all possible revisions of the code changes \( (c_1..c_n) \) is too time consuming for the large dataset used in the evaluation. Even with only one revision it takes weeks to compute the results in parallel.

Table 5.5 holds the number of groups that are not Type-1 or Type-2 clones (copied from Table 5.4) and the number of these groups that are *useful* according to the above definition. It also lists the number of *useful* groups that have two members, i.e., have size 2. The results show that up to \( 40,458 \) *useful* groups can be found in the repositories. This means that \( \mathcal{C}3 \) found up to \( 40,458 \) different systematic edits in the repositories. All groups have at least two members and all

<table>
<thead>
<tr>
<th></th>
<th>LH</th>
<th>LD</th>
<th>TH</th>
<th>TD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other Groups</td>
<td>63,980</td>
<td>87,740</td>
<td>36,385</td>
<td>57,919</td>
</tr>
<tr>
<td>...Code Changes</td>
<td>199,996</td>
<td>215,434</td>
<td>264,301</td>
<td>274,034</td>
</tr>
<tr>
<td>...Code Changes (%)</td>
<td>64</td>
<td>69</td>
<td>85</td>
<td>88</td>
</tr>
<tr>
<td>Useful Groups</td>
<td>31,338</td>
<td>40,458</td>
<td>15,261</td>
<td>19,900</td>
</tr>
<tr>
<td>...Code Changes</td>
<td>77,401</td>
<td>89,630</td>
<td>40,556</td>
<td>48,008</td>
</tr>
<tr>
<td>...Code Changes (%)</td>
<td>25</td>
<td>29</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>Useful Groups (Size 2)</td>
<td>24,817</td>
<td>35,421</td>
<td>11,234</td>
<td>16,069</td>
</tr>
<tr>
<td>...Code Changes</td>
<td>49,634</td>
<td>70,842</td>
<td>22,468</td>
<td>32,138</td>
</tr>
<tr>
<td>...Code Changes (%)</td>
<td>16</td>
<td>23</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>
members in the *useful* groups have a sufficient similarity so that ARES is able to create recommendations that are based on the pattern of the group. Thus the code changes in each group are similar enough to be systematic edits. However, the number of identified *useful* groups is only an approximation for the number of systematic edits. For example, sometimes several groups belong to the same systematic edit (see Sec. 5.5.2). The *useful* groups also contain identical code changes. Meng et al. exclude such changes from their definition of systematic edits, although the recommendation of such changes is still useful for developers. Additionally, C3 may group code changes together that developers do not consider similar enough to be part of the same systematic edits. The number of *useful* groups is also higher than the 5,000 systematic edits Molderetz et al. [243] reported for a similar number of commits (43,756). Besides the reasons discussed before, their approach is also responsible for the lower number. C3 creates string representations from the edit script and then applies a string similarity metric. As a consequence, edit operations can have some similarities if some of the characters in the string representation are identical. This is very common in the line-based representation with the full identifier names. In contrast, SysEdMiner has very strict similarity rules for edit operations. This makes it less flexible than C3.

The maximum number of code changes in *useful* groups of LD in Table 5.5 reveals that 29 % of all method changes are systematic as they are similar enough for C3 to move them into a group. This shows that a large amount of similar changes occur during the life-time of software projects. Due to this number, tools like ARES have the potential to save developers a lot of time during the maintenance of their projects.

Compared to the other three configurations, the LD configuration has more code changes in *useful* groups. However, most of them are in groups of size 2. For such small groups, ARES creates fewer wildcards and thus ARES generates fewer over-generalized patterns. The disadvantage of these groups is that the patterns are less general and prevent the identification of code locations that differ from the input examples. LH also has more code changes in *useful* groups, compared to the tree-based configuration. Thus, based on the total number of code changes in *useful* groups the line-based representations perform better than the tree-based ones.
5 Clustering of Code Changes

Figure 5.12.: Distribution of useful groups of at least size 3 for the 9 repositories.

The DBSCAN configurations also put more code changes in useful groups than the hierarchical ones. Again, this is due to the smaller groups that the DBSCAN configurations identify.

A look at Fig. 5.12 shows that for groups with size 4 or more the performance of the algorithms changes. For sizes greater or equal 4, the hierarchical configurations create more useful groups. TH even creates more useful groups than LD. The best configuration for such large useful groups is LH. The larger groups also allow an estimation of the number of times ARES would have been of help for developers. The analysis of ARES in Sec. 4.8.2 shows that for many systematic edits, two input examples are sufficient for a high recall value. Thus, it is possible to assume that this is also true for the 4,317 groups in the LH configuration of a size greater or equal 3. This means that for each of those groups, ARES would have helped developers for at least one code change.
5.5 Evaluation

Overall, this section showed that for 15,261 up to 40,458 groups ARES creates patterns from the C3 results that provide useful recommendations. Thus, these groups are indeed suited for the use in ARES.

5.5.5 Recommendation Accuracy

This section takes a closer look at the recommendations that ARES produces for the useful groups. Fig. 5.13 shows the accuracy values of ARES for these recommendations. As in Chapter 4, the token accuracies are based on the LVD between the tokens of the recommendation and the tokens of the modified method body from the repository. Due to the choice annotations, ARES sometimes creates more than one recommendation variation for a code location. The ARES min box in Fig. 5.13 corresponds to the lowest accuracy values of all the variations. The ARES max box uses the highest accuracy values. The median of ARES max is at a perfect accuracy of 100%. This shows that even for several thousand input examples, ARES creates very accurate recommendations. If one included the trivial recommendations for Type-1 or Type-2 clones, the results would be even better.

However, the pure accuracy values do not reflect whether ARES executes complex transformations or whether the transformations are so small that even the original methods have a high accuracy to their modification. In theory, it is possible that the recommendations have a lower similarity to the modified methods than their original counterparts. To measure the accuracy of the unchanged code, the tokens of the original
method bodies were compared to the tokens of the modified method bodies with the LVD. The *Unchanged* box displays the results.

In general, the accuracy of the recommendations is higher than the accuracy of the *Unchanged* method body. This is even true for the recommendation variants with the lowest accuracy (*ARES min*). However, there are some outliers with a similarity of nearly zero for *ARES min* and *ARES max*. This is also visible in the accuracy distribution in Fig. 5.14. The *ARES min* accuracy is low if a pattern contains many statements in choice annotations and the code outside choices is small. Under such circumstances, the recommendation variant that does not add any code for the choice annotations contains only a few statements. If the code in the choice annotations is large, the recommendation has fewer statements than the modified method in the repository. This can reduce the accuracy considerably. The groups that do not belong to any systematic edit are the root cause for the low accuracy values in the *ARES max* category. If C3 assigns the wrong original method to the wrong modified method during the method change extraction, ARES creates patterns that do not reflect actual code changes in the repository. As a consequence, the recommendations have no similarity to the modified code of the methods.

Table 4.5 shows that LASE only generates recommendations for a small subset of the groups from C3. The reason is that LASE is not mature enough to be used on such a large dataset. This is especially true if LASE has to use more than two input examples to generate patterns. Thus, LASE would only create recommendations for less complicated input groups. This would provide no new insights beyond the ones discussed in Sec. 4.8.4. Thus, this section examined only ARES.

### 5.5.6 Limitations and Threats to Validity

A limitation of C3 is the slow execution time of the clustering algorithms (especially of DBSCAN). To reduce this time it is foremost necessary to parallelize their implementations. Additionally, for a practical use of C3, the development of an incremental clustering approach is necessary. This avoids a re-computation of the complete clusters for each new commit. It is possible to use existing approaches to modify the current clustering algorithms for C3. For example, Jia et al. [169] present an approach that allows the incremental computation of eigenvalues. This is the main performance hot spot in the DBSCAN configuration of
5.5 Evaluation

Figure 5.14.: Accuracy distribution for the 9 repositories.

\( C_3 \). Such an approach, however, also influences other heuristics like the \( \epsilon \) computation and requires further research.

As the similarity computation and the clustering part of \( C_3 \) are independent of the origin of the code changes, \( C_3 \) works also across repositories. However, the slow execution time of DBSCAN prevents a cross-repository evaluation of \( C_3 \) on two or more large repositories. Thus, it is unclear whether \( C_3 \) produces suitable results for datasets that span multiple large repositories. After a parallelized version of the clustering step in \( C_3 \) is available, it is possible to explore this further.

The evaluation above also showed no single optimal configuration for all use cases. This makes it difficult for developers to decide which version of \( C_3 \) fits their needs. A solution that avoids such considerations is the combination of the different clustering results into one. There exist ensemble methods for this purpose [86]. However, this requires an additional evaluation of ensemble methods in combination with the heuristics of \( C_3 \).

A threat to the external validity of the results is the focus on one programming language. However, most parts of \( C_3 \) are language independent. It is only necessary to replace the parser front-end. Even the
higher-level edit operations (e.g., *StatementInsert*) are general enough so that they are applicable to most programming languages. Still, this part also requires an additional evaluation.

The small number of systematic edits for the relevance analysis in Sec. 5.5.2 is another threat to the validity of the results. However, as \textit{C3} also works in the large case study of Sec. 5.5.4, it is unlikely that it identifies only 23 groups of relevant code changes. To increase the confidence in this aspect, a larger manual evaluation of \textit{C3} groups with developers is necessary.

The incompleteness of the heuristics in ARES also impacts the generality of the results given in Sec. 5.5.4 and in Sec. 5.5.5. As ARES is a research prototype, the rule-based system and the other heuristics in ARES cover only a small part of the variances that occur in real code examples. This reduces the number of useful patterns in Sec. 5.5.4. Thus, in some cases, ARES unnecessarily creates an over-generalized pattern or an over-fitted pattern that is not even applicable to the input examples. However, as even the current version creates thousands of useful recommendations, the general implication that both \textit{C3} and ARES can help developers in executing systematic edits remains.

Finally, it is still unclear whether \textit{C3} is able to provide suitable input groups for other example-based tools like REFAZER. However, as \textit{C3} creates useful groups for ARES as well as LASE (see the previous work on \textit{C3} [196]), this it is likely.

5.6 Summary

This chapter presented \textit{C3}, a tool that identifies groups of similar code changes in code repositories. These groups are ideal inputs for ARES. They allow ARES to create patterns that can support developers in the execution of systematic edits.

Initially, \textit{C3} computes the edit script for all file changes in the repository with MTDIFF. Based on these scripts, \textit{C3} determines all method changes that the repository contains. Then \textit{C3} converts each method change into one of two available string representations. The line-based representation consists of the deleted and inserted lines computed with Myer's algorithm (i.e., *diff*). The tree-based representation is a short encoding of the edit script of the method change. To create this encoded string, \textit{C3} transforms the script into one with higher-level edit operations like *StatementInsert*. It also creates trees from edit operations.
that have a child-parent relationship. Then $C_3$ converts the trees and
the higher-level edit operations into characters. This conversion into
characters uses a two character-wide hash for identifiers and relies on
relative positions to encode moves. After the creation of the string repre-
sentations, $C_3$ computes the similarity matrix for them and thus for
the code changes they represent. $C_3$ uses the LCS algorithm to compute
the similarities between each code change pair. To reduce the overhead
of the matrix computation, $C_3$ uses a k-nearest neighbor filtering.

This similarity matrix describes a graph in which the edges hold the
similarity values. As there are node pairs without edges due to the
k-nearest neighbor filtering, the graph consists of several independent
connected components. If there are connected components with more
than 10,000 nodes, $C_3$ uses the Kernighan-Lin algorithm to split them
into two components to reduce the execution time of the subsequent
clustering algorithms. On each of the components, $C_3$ executes one
of two supported clustering algorithms. The first supported algorithm
is agglomerative hierarchical clustering. It works directly on the con-
nected components. The second supported clustering algorithm is
DBSCAN. It requires that the data points (i.e., the code changes) lie
in a $d$-dimensional space of low dimension. To embed the connected
components into such a low dimensional space, $C_3$ uses the Laplacian
Eigenmap. After the embedding, $C_3$ applies DBSCAN on the embedded
code changes.

The evaluation of $C_3$ shows that the similarity matrix computation
is fast enough for practical use, even for large datasets. It takes only
seconds for most commits. Its design also allows an incremental com-
putation. The agglomerative hierarchical clustering is also fast and
computes the clustering results in under an hour for large repositories.
In contrast, DBSCAN takes 13 hours for the largest repository due to
the time-consuming eigenvalue computation for the embedding. For a
practical use of $C_3$, it is necessary to reduce this time.

The two supported string representations and the two supported
clustering algorithms result in four $C_3$ configurations. In the set of
code changes from the 9 analyzed repositories, all four configurations
identify thousands of groups that are suitable inputs for ARES. In fact,
inside the $106,697 + 91,533$ method changes from the Eclipse reposi-
tories, $C_3$ is able to identify 23 of the 24 manually composed groups
from Meng et al. [236]. This shows that $C_3$ is able to identify a large
portion of given systematic edits even in large datasets. An evaluation
Clustering of Code Changes

of the C3 results with ARES also shows that up to 40,458 groups lead to patterns that give useful recommendations. As they are similar enough for such recommendations, it is likely that each group consists of code changes that belong to the same systematic edits. Otherwise, ARES would be unable to generate useful patterns. Additionally, most of the groups of code changes that C3 identifies affect methods with different statements. This means that the original methods of the code changes are not Type-1 or Type-2 clones. Thus, C3 identifies many non-trivial groups of similar code changes. Finally, an analysis of the combination of C3 with ARES shows that ARES generates recommendations with a high accuracy even for such a large dataset. As all 4 configurations show different strengths and weaknesses there is no clear optimal configuration. However, the line-based representation in combination with the hierarchical clustering is a good compromise for most applications. It is faster than the DBSCAN versions and also identifies the most useful groups of size 4 and greater.

This chapter completes the contributions of this thesis. ARES in combination with C3 is able to learn code transformation patterns from repositories automatically. These patterns give recommendations to code changes that can help developers in executing systematic edits. To allow further research on the clustering of code changes, the results of C3 are publicly available (https://github.com/FAU-Inf2/cthree).
6 Conclusions and Future Work

As names have power, words have power. Words can light fires in the minds of men. Words can wring tears from the hardest hearts.

Patrick Rothfuss, The Wise Man’s Fear

As words have power, recommendations have power. They can change the perspective of developers and they can reveal hidden problems in code bases. This can prevent serious consequences. To provide recommendations for developers, this thesis presented ARES, an example-based recommendation tool. For each group of similar code changes, it creates a pattern that describes the properties shared by all group members. The created pattern gives ARES the power to search for code locations that contain the problem addressed by the input changes. If ARES identifies such a code location, it can use the code transformation described by the pattern to automatically generate recommendations for developers. Compared to previous approaches, the recommendations by ARES are more accurate and have the advantage that developers often can add them directly to their code base. This avoids mistakes that occur if developers have to adapt a recommendation to the individual needs of their code base. It also reduces the time developers have to spend on executing boring, repetitive systematic edits. ARES achieves its accurate recommendations via its unique pattern design that captures more details of the input code changes compared to previous approaches. Its design also expresses code movements, a basic code transformation that other approaches often neglect. Encoding movements in the pattern has the advantage that this further increases the accuracy of the recommendations. In contrast to patterns of other approaches, the patterns created by ARES have a readable, textual
Conclusions and Future Work

representation. This also has the benefit that developers can review the patterns and even adapt them to their needs if necessary. Although the adaptation of a pattern is overhead, it often is less effort and less boring than the manual execution of a systematic edit.

MTDIFF is the heart of ARES. It is a novel tree differencing algorithm that produces shorter edit scripts compared to other state-of-the-art algorithms (GT, RTED, CD, JSync, JDime). ARES requires these short edit scripts to compute patterns with high accuracy, precision, and recall values. To create shorter edit scripts than the other examined algorithms, MTDIFF uses a two-step approach. The first step creates leaf pairs. These leaf pairs give MTDIFF the information it needs to compute pairs of inner nodes in the second step. With these pairs, MTDIFF is able to generate the edit script. MTDIFF inherits its two-step structure from CD. Despite this relationship, MTDIFF uses different heuristics to compute the leaf and inner node pairs. The heuristics of CD rely on a coarse-grained AST and work best if the leaves are complete statements instead of expressions. However, ARES requires a more fine-grained AST to handle differences within statements. This increases the accuracy of the recommendations. As its heuristics for fine-grained ASTs make MTDIFF more time-consuming, it requires the preprocessing optimization $\Theta_A$. This optimization creates pairs for all unchanged code parts beforehand and reduces the number of nodes that MTDIFF has to handle. Additionally, MTDIFF makes use of five post-processing optimizations $\Theta_{B-F}$ to further shorten the edit scripts for ARES.

Standalone, ARES is unable to provide recommendations automatically. Similar to other tools, developers have to provide input examples for the automatic pattern creation. To lift the heavy burden of input example gathering from the shoulders of developers, this thesis also presented $C3$. It uses one or more code repositories as input and extracts groups of similar code changes from them. These groups are fitting inputs for ARES. Groups from previous extraction approaches are less suited for it because they mostly focus on the detection of similar code fragments. Using such groups would lead to very specific patterns that do not identify additional code locations and thus are not general enough to provide helpful recommendations for developers. $C3$ differs from previous approaches and searches for similar code changes in code repositories. Initially, $C3$ extracts all method changes from the master branches in the repositories. Afterwards, it converts each method
change into a string to allow the use of fast string algorithms. To be precise, \textit{C3} uses the LCS of the strings to compute the pairwise similarities between the strings. Currently, \textit{C3} supports two different string representations, a line-based and a tree-based one. The line-based \textit{diff} representation has the advantage that it allows \textit{C3} to identify more similar code changes in the repositories. The alternative tree-based representation is faster and leads to larger groups of similar changes. Larger groups are a double-edged sword as they more often lead to overgeneralized patterns. However, if this does not happen, the patterns are more general and thus provide more recommendations than patterns from smaller groups.

The result of the pairwise similarity computation is a matrix that \textit{C3} passes on to its clustering algorithms. It is possible to use either agglomerative hierarchical clustering or DBSCAN to split the extracted method changes into groups. The hierarchical clustering is faster and leads to larger groups. DBSCAN is more time-consuming, but identifies more similarities between code changes, creates more groups and thus has the potential to generate more recommendations for developers. The evaluation of \textit{C3} showed that the four possible combinations of two string representations and two clustering algorithms have different strengths and weaknesses. However, all have in common that they provide groups of code changes that ARES can use to create helpful recommendations.

In combination, MTDIFF, ARES, and \textit{C3} create a powerful recommendation system. With only repositories as input, this tool combination automatically provides helpful recommendations for developers without any additional human effort. In fact, together the three tools form a system that learns code transformations from repositories without supervision. The learned code transformations are visible in the patterns delivered by the tool combination. Together, MTDIFF, ARES, and \textit{C3} break the limitation of other available static analysis tools. For example, they are no longer bound by predefined patterns. The combination is also flexible enough to help developers in executing a wide range of different systematic edits and thus provides a solution to the problems discussed in the introduction of this thesis. Due to the use of \textit{C3} on repositories, the combination of the three tools even reacts to changes added by other developers. This allows them to identify problems that have been previously unknown to developers. Overall, MTDIFF, ARES, and \textit{C3} make the execution of systematic edits faster.
and thus they avoid boring, repetitive manual code transformations. This can decrease attention failures and thus errors caused by developers.

The algorithms and heuristics presented in this thesis are also not restricted to the area of recommendation tools for systematic edits. They have a broader impact beyond this narrow field. For example, the optimizations $\Theta_{A-F}$ are compatible with different tree differencing approaches. Thus, they can be of use in any area that requires an accurate identification of differences between trees. In fact, there are already some existing research tools [207, 221] that include the optimizations for better results. As the optimizations and MTDIFF do not use AST specific features, it is even possible to use both for trees that have no relation to programming languages. For example, the detection of short edit scripts could be useful for biology projects [6, 339].

All the contributions of this thesis open doors for further research. As the GT framework provides different parsers, an extension of the MTDIFF evaluation to other programming languages can reveal language specific change patterns that the current optimizations $\Theta_{A-F}$ and MTDIFF do not cover. This can lead to the development of additional optimizations.

The data of the current MTDIFF evaluation also contains examples that can benefit from additional optimizations. In several cases, GT and RTED create shorter edit scripts by using update operations instead of moves. It is possible to define an additional optimization that takes care of such cases. However, as such an optimization has no benefit for ARES, it was not developed and evaluated. In general, the preference of moves over a shorter edit script is application specific. Other applications, like the visual presentation of code changes, have other requirements. As shown in the questionnaire of Chapter 3, the edit script size has only a medium effect on the perceived helpfulness. Thus, it is likely that there are other edit script characteristics that influence the comprehension of code change representations and their helpfulness. This is in line with the study by Decker [81]. The results of the study show that most developers preferred the more coarse-grained representation of Decker’s srcDiff, instead of the fine-grained representation by GT. However, as in other cases, the study does not reveal the underlying principles leading to this result. Thus, it is still unclear what code change visualization characteristics influence the perception of developers. Whereas the size has only a medium effect, the
number of different visualized code changes within a code line or the
used colors could have a greater impact. To have tight control over such
variables, research efforts have to leave beaten paths. A focus on tool-
independent research is more beneficial for such questions. Only with
artificial examples that are independent of specific tools it is possible
to sufficiently control the variables of the visual representation for a
deeper analysis.

The evaluation of ARES exposed several limitations. Finding solu-
tions to these issues requires further research. In particular, ARES is
only able to recommend changes within method bodies in its current
version. Extending the pattern scope above the granularity of method
bodies creates new challenges as the code outside methods follows dif-
ferent rules. For example, the method order in Java code does not influ-
ence the semantics. However, if the order of method was changed in
several input examples to increase the readability, ARES should include
this change in the recommendation. Thus, a future version of ARES
should ignore the method order if it varies between the input exam-
pies. However, ARES should enforce it, if all input examples change the
method order in the same fashion. Expressing such code characteris-
tics requires an extension of the current pattern design. Similar to the
scope extension, an application to other programming languages can
also reveal the need for additional features. Currently, the automatically
created patterns also use only two wildcard types, namely expr-wildcards
and stmt-wildcards. Better accuracy, precision and recall values may
become possible by extending the supported types. For example, wild-
cards that restrict the data types of the code locations can increase
the precision. Besides extensions to the pattern design, the search for
multiple patterns in parallel requires more research. There is also the
possibility that ARES can improve its own heuristics automatically. At
the moment, it is limited by the manually created rule-based system.
Replacing these rules with a trained neural network would allow ARES
to cover more diverse input example groups. With millions of input
eamples from public repositories provided by C3, there is sufficient
data to train such a neural network. It can train without human super-
vision by comparing the recommendations with the code changes in the
repositories.

The execution time of ARES and similar recommendation systems
is another open research topic. Although ARES needs only seconds
to search for applications of a single pattern, searching for millions
Conclusions and Future Work

of patterns still takes weeks. To use ARES with such a huge pattern database, other search strategies like suffix trees [176] have to be explored.

ARES also has the potential to provide recommendations beyond the context of systematic edits. With suitable input examples, ARES can provide recommendations for special topics like energy efficient programming or parallelization. Using ARES in such special environments can reveal new requirements for a more general recommendation system.

As for ARES, the limitations of \( C^3 \) give directions for additional research efforts. The execution time of \( C^3 \) is still an open topic. Whereas it is no problem to add a new row to the similarity matrix with each new commit, the current version of \( C^3 \) has to execute the clustering analysis on the complete dataset to consider data from a new commit. Incremental clustering techniques may be able to lower this overhead considerably. The central structure of \( C^3 \) also allows the exploration of other similarity metrics and other clustering algorithms.

For example, Kehrer et al. [181] present an approach that lifts basic edit operations to a higher abstraction level. It is possible to include such edit operations in the tree-based representations in \( C^3 \). Other clustering techniques, e.g., self organizing maps that use neural networks [193] can also lead to useful groups for ARES. In its current version, \( C^3 \) solely uses the similarity between code changes. A combination of different metrics can further optimize the clustering results, e.g., it can be beneficial to include similarities of messages, used types, etc.

An important aspect concerning recommendation systems and static analysis tools was not covered in this thesis as it was beyond its scope. To be precise, this thesis does not address the interaction of developers with ARES. However, this is an important topic for future research as there are known challenges in the interaction of developers with recommendation systems. In their study, Christakis and Bird [61] identified 15 pain points that prevent developers from using static analysis tools. These 15 points include too many false positives, bad visualizations, no ranking mechanisms, etc. With additional research effort, it is possible to extend ARES to lessen the pain caused by such issues. For example, despite ARES's high precision, millions of patterns still create thousands of false positives. Reducing them requires the development of additional filter mechanisms. For example, it is possible that ARES checks whether or not an applied recommendation would lead to
a compilation error. It is also possible to determine whether or not an applied recommendation fulfills the available tests. Such filter mechanisms can reduce false positives.

Gašparič and Janes [130] also encourage authors of recommendation systems to add visualizations and extend the provided information. According to them, it is important that the recommendations help developers in comprehending the reasons that lead to a recommendation. For that purpose, it can be helpful to include the code changes that formed the pattern and their authors in the recommendations. The authors of the commercial static analysis tool Coverty [39] also stress that it is crucial that developers understand the generated recommendations. A community-based rating system for patterns can be a solution. It allows a ranking of patterns (including their corresponding recommendations) and reduces the information overhead.

All possible improvements to the presentation of recommendations share the restriction that only developers can provide a useful evaluation. Only with their feedback it is possible to separate useful features from irrelevant ones. In line with more research on code change representations, this research can provide more insights if it is done independently of a specific recommendation tool.

The datasets created in the course of this thesis are a treasure for future research projects. The input data definition and results of the MTDIFF evaluation are publicly available on github (https://github.com/FAU-Inf2/tree-measurements) and can serve as benchmark for tree differencing algorithms. Similarly, the ARES input data and results are also publicly available (https://github.com/FAU-Inf2/ARES) and can serve as accuracy benchmark for code recommendation systems. This data can also be the source for a general recommendation tool benchmark that allows the evaluation of precision and recall on a larger scale. However, this requires a human evaluation of the respective input groups from C3 beforehand. Additionally, a human examination of the code locations that belong to the created recommendations is necessary for a full benchmark. Both topics are still future work.

The dataset from C3 (https://github.com/FAU-Inf2/cthree) and the corresponding ARES patterns can also offer new insights concerning the type of changes that occur regularly during the development process. Such insights can lead to the development of novel static analysis tools that prevent these kinds of problems from happening. As for the benchmarks, this requires a human evaluation of the results beforehand.
6 Conclusions and Future Work

To allow the reproducibility of the results and kindle further research, ARES (https://github.com/FAU-Inf2/ARES) and MTDIFF (https://github.com/FAU-Inf2/treedifferencing) are also available under an open-source license.
A  Tree Differencing Appendix

This section presents code change examples for GT, RTED, CD, JSync, and JDime for which the $\Theta_{A-F}$ reduce the size of the edit script. Due to the different AST designs and the different heuristics used by the five approaches, the examples vary from approach to approach.

A.1  GT Examples

For the sake of completeness, Fig. A.1 shows the combined code of the GT examples for $\Theta_B$, $\Theta_C$, $\Theta_E$, and $\Theta_F$. Fig. A.2 holds the code for the $\Theta_D$ example. The details were discussed in Sec. 3.3.

Original code:                       Modified code:

01: assert (condB);                  assert(!condB);  
02: callC(1, 2, c);                  callC(c, 2, 1);  
03: printE("42!" + "Answer:"");     printE("Answer:" + "42!");  
04: deleteF();                       F f = new F(1);  
05: F f = new F(1, 2);               f = new F (1);

(a) GT requires 14 edit operations $\equiv$ 14 boxes.

01: assert (condB);                  assert(!condB);  
02: callC(1, 2, c);                  callC(c, 2, 1);  
03: printE("42!" + "Answer:"");     printE("Answer:" + "42!");  
04: deleteF();                       F f = new F(1);  
05: F f = new F(1, 2);               f = new F (1);

(b) GT in combination with all optimizations requires 8 edit operations.

Figure A.1.: GT results with and without $\Theta_{A-F}$. Same legend as in Fig. 3.2.
Figure A.2.: Comparison between GT\textsubscript{A−C} and GT\textsubscript{A−D} for Θ\textsubscript{D}. Same legend as in Fig. 3.2.
A.2 RTED Examples

Throughout Chapter 3, RTED plays a special role as it does not detect code movements, in contrast to the other four evaluated tree differencing algorithms. However, it is compatible with $\Theta_{A-F}$. This makes it possible to evaluate the effect that these optimizations have on a move-less edit script.

Fig. A.3 holds an example for RTED that the optimizations can improve. Sec. 3.3 already discussed the movements in lines 1 and 2. For line 3, RTED creates an unnecessarily large edit script. Partially, the switched argument order is responsible. However, the restriction of the GT framework play also a role. The GT-AST differentiates between the field access of this and super. Thus, both have different labels. The RTED algorithm internally pairs such nodes together. However, the GT implementation of RTED removes them from the mapping as it does not allow pairs of nodes with different labels. The combination of the reordered arguments and the restrictions of the GT framework leads to the 6 delete and 6 insert operations. As the this node pair is part of the LCS that $\Theta_B$ computes, it adds the pair to the mapping together with the corresponding access nodes. This reduces the edit script size for this line from 12 to 7.

As RTED does not consider moves, it also does not recognize the optimal node pairs for line 4. The GT framework also drops the pairs with different labels (e.g., i and c) that RTED generates from the mapping. As a consequence, the edit script from RTED contains the four edit operations in line 4 of Fig. A.3. $\Theta_C$ can replace them with two move operations.

For the example in Fig. A.4, the combination of RTED with $\Theta_B$ leads to a mapping that $\Theta_D$ can further improve. The baseline RTED algorithm assigns no partner to the code block in lines 4–7 of the modified code. $\Theta_B$ finds a partner (the block in line 3 of the original code) and adds this pair to the mapping. However, due to zoo, the code block in line 9 of the original code is a better choice. $\Theta_D$ changes the pairs accordingly and reduces the edit script size by 1.

As RTED is unaware of moves, it adds the pair (k, t) to the mapping in line 5 of Fig. A.3. This causes one update and one move operation for this pair of nodes. However, the RTED algorithm does not cause this move directly. RTED maps the parent of k to the parent of t. As both parents have different labels, the GT framework removes this pair from the mapping. Only this removal causes the additional move operation.
$\Theta_E$ can reduce the edit script for this example by 1 as it pairs the two $t$ nodes in the AST together.

A pair of nodes with different labels causes also the unnecessary edit operations for line 6. RTED pairs the nodes $\mathtt{this}$ and $\mathtt{fieldF}$ together. The GT framework discards it as the nodes have different labels. This leads to the 12 edit operations in line 6. The combination of RTED with $\Theta_{A-E}$ creates a shorter edit script, but the move and update operations for $\mathtt{classf}$ remain. $\Theta_F$ changes the pairs in the mapping and avoids the unnecessary move.

Original code:  
Modified code:  
\begin{verbatim}
01: moveA();  
02: stmtA();  
03: callB(this, B, super, B);  
04: callC(1, 2, c);  
05: E( callE, k("Sauron", t));  
06: ClassF classF = a(this, Q);  
\end{verbatim}  
\begin{verbatim}
01: moveA();  
02: stmtA();  
03: callB(this, B, super, B);  
04: callC(1, 2, c);  
05: E( callE, k("Sauron", t));  
06: ClassF classF = a(this, Q);  
\end{verbatim}

(a) RTED requires 39 edit operations.

(b) RTED in combination with all optimizations requires 24 edit operations.

Figure A.3.: RTED results with and without $\Theta_{A-F}$. Same legend as in Fig. 3.2.
A.2 RTED Examples

Original code:

```java
if (condD) {
    callD();
    for (Item i : l) {
        if (A) {
            for (Element e : s) {
                zoo();
            }
        }
        if (condD) {
            callD();
            for (Item i : l) {
                if (A) {
                    for (Element e : s) {
                        Item i;
                        zoo();
                    }
                }
                if (Element.class) {
                    e();
                }
            }
            if (Element.class) {
                e();
            }
        }
    }
}
```

Modified code:

```java
if (condD) {
    callD();
    for (Item i : l) {
        for (Element e : s) {
            Item i;
            zoo();
        }
    }
    if (Element.class) {
        e();
    }
}
```

(a) RTED/JSync require 32 edit operations.

(b) RTED/JSync in combination with $\Theta_{A-C}$ require 23 edit operations.

(c) RTED/JSync in combination with $\Theta_{A-D}$ require 22 edit operations.

Figure A.4.: RTED/JSync results with and without $\Theta_D$. Same legend as in Fig. 3.2.
A.3 CD Examples

CD uses a coarse-grained AST in which full statements are leaves. As a consequence, this section requires larger examples. $\Theta_A$ can improve the edit script from CD for lines 2–4 in Fig. A.5. For multiple identical leaves in an AST (e.g., methodA in Fig. A.5), CD selects the first leaf pair that appears during the post-order traversal of the ASTs. Thus, due to the movement of the if statement, CD maps the node methodA outside the if statement in the original code to the node methodA inside the if statement in the modified code. As these non-optimal leaf pairs force the similarity of the if statements below the CD threshold for inner nodes, CD also does not add the pair (if_o, if_m) to the mapping. This increases the size of the edit script unnecessarily and leads to four additional edit operations. $\Theta_A$ detects the identical moved if statement and replaces the edit operations with a single move.

$\Theta_B$ is able to optimize the edit script for the code in lines 5–6. Due to the change from result to wrongName, the similarity of the return statement dropped below the CD threshold for leaf pairs. This pushed the similarity of the if statements in line 5 below the CD threshold for inner nodes. In combination, these missing pairs in the mapping cause the four edit operations in lines 5–6. $\Theta_B$ adds the missing pairs as they are part of its computed LCS and thus requires only a single update operation to express this code change.

In combination with CD, $\Theta_C$ mainly improves the edit script for parameters and only after $\Theta_A$ and $\Theta_B$. Lines 13–16 in Fig. A.5 show an example. The results from $\Theta_A$ and $\Theta_B$ offer a chance for $\Theta_C$ to lower the edit script size. After the first two optimizations, the parameter Object value is mapped to the parameter C expected. At the same time, Object is mapped to String. This creates an unnecessary move in the edit script. $\Theta_C$ analyzes the unmapped leaf C and detects a suitable match. It adds the pair (Object, C) to the mapping and reduces the edit script by 1.

The coarse-grained CD-AST makes an improvement by $\Theta_D$ only visible in an even larger code change. Fig. A.6 shows a suitable example. $\Theta_B$ added two pairs of code blocks. This is shorter than the two delete and the two insert operations by CD, but it is still unnecessarily large. $\Theta_D$ further optimizes these pairs and reduces the size of the edit script by 2.

Due to the way CD handles leaves in combination with its coarse-grained AST, $\Theta_E$ has no effect on the CD results.
Fig. A.5 also holds an example for $\Theta_F$. In lines 7–10, two statements were modified. Due to their textual similarity, CD maps the statement in line 8 to the statement in line 10 and vice versa. This causes two update and two move operations. $\Theta_F$ detects these useless move operations and maps the statement in line 8 in the original code to the statement in line 8 of the modified code. Additionally, it adds the statements in line 10 as a pair to the mapping. This drops the two move operations from the edit script.
A Tree Differencing Appendix

Original code:

```java
Obj cd() {
  methodA();
  if (condA)
    methodA();
  if (condB)
    return result;
  if (condF)
    fTimeout = timeout;
  for (F f: valuesF)
    fTimeout = t.fTimeout;
  return null;
}

void methodC(Object value) {
  }
```

Modified code:

```java
Obj cd() {
  if (condA)
    methodA();
  methodA();
  if (condB)
    return wrongName;
  if (condF)
    this.timeout = timeout;
  for (F f: valuesF)
    timeout = t.timeout;
  return null;
}

void methodC(String message, C expected) {
  }
```

(a) CD requires 25 edit operations.

(b) CD<sub>A−B</sub> requires 11 edit operations.

(c) CD<sub>A−F</sub> requires 8 edit operations.

Figure A.5.: CD results with and without Θ<sub>A−F</sub>. Same legend as in Fig. 3.2.
A.3 CD Examples

Original code:

```java
01: class C {
02:   boolean c() {
03:     if (l) {
04:         }
05:     D d = l;
06:     if (d) {
07:       return false;
08:     }
09:     if (d) {
10:       return isOverridingMethod();
11:     }
12:     }
13:     if (d) {
14:       return;
15:     }
16:     }
17:     }
18:     }
19:   }
20:   void h() { return; }
21:   }
22: }
```

Modified code:

```java
class D {
  boolean c() {
    if (1) {
      final D d = l;
      if (d) {
        }
      return;
    }
  }
  boolean o() {
    if (d) {
      isOverridingMethod();
    }
    if (d) {
      return inOverridingMethod;
    }
  }
  void h() { return; }
  private void i(D a) { }
}
```

(a) CD in combination with $\Theta_{A-C}$ requires 28 edit operations.

(b) CD in combination with $\Theta_{A-D}$ requires 26 edit operations.

Figure A.6.: Comparison between CD$_{A-C}$ and CD$_{A-D}$ for $\Theta_D$. For this example, only the relevant moves are in blue.
A.4 JSync Examples

$\Theta_A$ is able to reduce the edit script size for the code in Line 1 in Fig. A.7. In this line, a developer switched the addition with the subtraction. For changed lines, JSync uses the LCS algorithm to identify leaf pairs for the mapping. Its use of the LCS maps the first occurrence of $a$ and the first occurrence $i$ to their first counterparts in the modified code. Based on these leaf pairs, JSync computes the pairs of inner nodes and maps the addition to the subtraction. As a consequence, the edit script of JSync contains two update operations instead of a single move. $\Theta_A$ detects the identical subtrees beforehand and adds the corresponding nodes to the mapping. This reduces the size of the edit script by 1.

The LCS variant of JSync takes the direct neighbors of the leaves into account. This can lead to longer edit scripts. For example, in lines 1–4 of Fig. A.7 too many neighbors of $\text{condB}$ were changed. As a consequence, the $\text{condB}$ nodes in the original and in the modified version do not fulfill the match condition of JSync’s LCS variant. Therefore, JSync does not add the $\text{condB}$ pair to the mapping. This increases the edit script size by one. The LCS algorithm in $\Theta_B$ detects the pair of $\text{condB}$ nodes and adds it to the mapping. This replaces the delete and insert operations with a single move.

Due to the restriction of the LCS algorithm, the LCS variant of JSync and $\Theta_B$ are both unable to detect the move of $c$ in line 3 of Fig. A.7. $\Theta_C$ examines the previously unmapped $c$ and detects a suitable partner. It adds the pair $(c_o, c_m)$ to the mapping. This reduces the size of the edit script by 2 as this pair eliminates the delete and insert operations for the $c$ nodes.

Like RTED, JSync creates unnecessary edit operations for the code change in Fig. A.4. In this code change, JSync does not find a suitable partner for the code block in lines 4–7. Internally, JSync uses the Exas similarity value to identify partners for inner nodes. However, this value is based on shared leaf nodes. As the code block in lines 4–7 only shares children with the for loop in line 3 of the original code part (e.g., $\text{Item}$), it has an Exas similarity value of 0 to other unmapped blocks. However, the optimization $\Theta_B$ finds a partner (the block in line 3 of the original code) and adds this pair to the mapping. Due to $\text{zoo}$ the code block in line 7 is a better choice. $\Theta_D$ recognizes this and changes the pairs in the mapping. This makes the move operation for $\text{zoo}$ obsolete and reduces the edit script size by 1.
Due to its LCS variant, JSync also does not detect the movement of t in line 5 of Fig. A.7. Again, there are too many changes in the adjacent leaves of t. This prevents the LCS variant from adding the pair of t identifiers to the mapping. Thus, the t nodes remain without partners in the mapping. As the t nodes are unmapped, $\Theta_B$ can add the pair $(f_o, t_m)$ to the mapping. However, this creates an unnecessary update operation. $\Theta_E$ detects that there is a leaf $t_o$ that has the same label and value as $t_m$ and adds the pair to the mapping. This removes the update operation from the edit script and reduces its size by 1.

Compared to the other algorithms, JSync offers fewer opportunities for $\Theta_F$ to optimize due to its Exas similarity metric. As a consequence, $\Theta_F$ improves the JSync results only in large code changes. For smaller changes, $\Theta_F$ can only improve the edit script if JSync is combined with $\Theta_B$. Fig. A.8 shows the original and modified code of such a smaller example. In this example, $\Theta_B$ pairs the node $l_o_1$ from the left-hand side of the assignment in line 5 together with the node $l_m$ in line 2 of the modified code. This causes two additional move operations. $\Theta_F$ recognizes that the two moves are unnecessary and changes the mapping accordingly. This reduces the edit script size by 2.

Figure A.7.: JSync results with and without $\Theta_{A−F}$. Same legend as in Fig. 3.2.
A Tree Differencing Appendix

Original code:   Modified code:

void m1();       void m1() {
void m2();       c = [[line]];  
void m3() {
  l = l(e());    void m2() {
  if (cond) {
    e.p();         e.p();
    }              }
  e.p();         void m3() {
  }              exit();
}

(a) JSync in combination with $\Theta_{A-E}$ requires 18 edit operations.

void m1();       void m1() {
void m2();       c = l( line );
void m3() {
  l = l(e());    void m2() {
  if (cond) {
    e.p();         e.p();
    }              }
  e.p();         void m3() {
  }              exit();
  }
}

(b) JSync in combination with $\Theta_{A-F}$ requires 16 edit operations.

Figure A.8.: Comparison between JSync$_{A-E}$ and JSync$_{A-F}$ for $\Theta_F$. For this example, only the relevant moves are in blue.
A.5 JDime Examples

JDime already uses a variant of $\Theta_A$. However, this variant does not search for identical subtrees in the complete file. Thus, an execution of $\Theta_A$ on the complete file still can improve the results. JDime also uses the LCS algorithm to detect statement pairs. Due to these two properties, JDime does not detect the movement of the call moveA in Fig. A.9. This leads to four additional edit operations compared to the single move operation the combination of JDime and $\Theta_A$ creates.

For the assert statement in Fig. A.9, the top-down approach of JDime prevents the generation of a short edit script. During its top-down walk, JDime encounters the identifier condB in the original AST. On the modified AST it encounters a negation (!) node. As a consequence, the top-down traversal of JDime stops as both nodes have different labels. This causes a delete and an insert operation for condB. As the condB node pair is part of its computed LCS, $\Theta_B$ adds the pair to the mapping and replaces the two edit operations with a single move operation.

Similar to other tree differencing approaches, JDime handles argument nodes with the LCS algorithm. As described above, the LCS algorithm is unable to detect the move of c in Fig. A.9 and thus does not add the pair of c identifier nodes to the mapping. $\Theta_C$ examines the unmapped leaves, detects this pair and adds it to the mapping. In a similar fashion it identifies the pair $(2_o, 2_m)$. This replaces the two delete operations for line 3 and the two insert operations for line 2 of Fig. A.9 with two move operations.

Whereas the heuristics in JDime do not lead to mappings that $\Theta_D$ can improve, the combination of JDime with $\Theta_B$ creates such mappings. Fig. A.10 holds an example for JDime. After the optimization by $\Theta_B$, the parent of return r2 (a List node in the JDime-AST) in AST$_o$ is mapped to the parent of return r1 in AST$_m$. This leads to an unnecessary move operation. To remove this operation, $\Theta_D$ maps the parent of return r1 in AST$_o$ to the parent of return r1 in AST$_m$ as these two parents share more children. This reduces the edit script size by 1.

The assignment to e in Fig. A.9 gives an example in which the combination of JDime with $\Theta_B$ leads to unnecessary edit operations. In the example, the top-down approach of JDime stops at the call s as there is no identical node in the modified part of the example. Thus, as JDime does not add the pair $(s_o, str_m)$ in the mapping, the edit script contains several delete and update operations for the assignments to e. $\Theta_B$ optimizes this mapping and adds the pairs ("42!", "Answer:"), and
("Answer:”, ”42!"). However, this causes two update operations. $\Theta_E$ can replace them with a single move, because it pairs the identical string constants together.

$\Theta_F$ can reduce the edit script size for the example in Fig. A.11. After the initial execution of JDime, $\Theta_B$ mapped the call $c(d)$ to the call $\text{asTestCase}(d)$. However, the combination of JDime and the optimization $\Theta_B$ also mapped the argument $d$ in $c(d)$ to the argument $d$ in $c_2(d)$. This creates an unnecessary move operation. Due to the pair $(c(d), \text{asTestCase}(d))$ in the mapping, it is more suitable to map the argument $d$ in $c(d)$ to the $d$ in the $\text{asTestCase}$ call. $\Theta_F$ recognizes this, updates the pairs in the mapping and eliminates the unnecessary move operation. This reduces the edit script size by 1.

### Original code:

```plaintext
01: moveA ();
02: assert ( condB );
03: callC (1 , 2 , c);
04: e = s ("42!", "Answer:" );
```

### Modified code:

```plaintext
01: moveA ();
02: assert ( !condB );
03: callC (c , 2 , 1);
04: e = str ("Answer:" , "42!");
```

(a) JDime requires 19 edit operations.

```plaintext
01: moveA ();
02: assert ( condB );
03: callC (1 , 2 , c);
04: e = s ("42!", "Answer:" );
```

### Modified code:

```plaintext
01: moveA ();
02: assert ( !condB );
03: callC (c , 2 , 1);
04: e = str ("Answer:" , "42!");
```

(b) JDime in combination with all optimizations requires 7 edit operations.

Figure A.9.: JDime results with and without $\Theta_A - F$. Same legend as in Fig. 3.2.
A.5 JDime Examples

<table>
<thead>
<tr>
<th>Original code:</th>
<th>Modified code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: if (f) {</td>
<td>switch (i) {</td>
</tr>
<tr>
<td>02:</td>
<td>case A:</td>
</tr>
<tr>
<td>03: return r2;</td>
<td>Return r1;</td>
</tr>
<tr>
<td>04: } else {</td>
<td>case B:</td>
</tr>
<tr>
<td>05: return r1;</td>
<td>return r2;</td>
</tr>
<tr>
<td>06:</td>
<td>default:</td>
</tr>
<tr>
<td>07:</td>
<td>z();</td>
</tr>
<tr>
<td>08:</td>
<td>}</td>
</tr>
</tbody>
</table>

(a) JDime in combination with $\Theta_{A-C}$ requires 17 edit operations.

<table>
<thead>
<tr>
<th>Original code:</th>
<th>Modified code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: if (f) {</td>
<td>switch (i) {</td>
</tr>
<tr>
<td>02:</td>
<td>case A:</td>
</tr>
<tr>
<td>03: return r2;</td>
<td>return r1;</td>
</tr>
<tr>
<td>04: } else {</td>
<td>case B:</td>
</tr>
<tr>
<td>05: return r1;</td>
<td>return r2;</td>
</tr>
<tr>
<td>06:</td>
<td>default:</td>
</tr>
<tr>
<td>07:</td>
<td>z();</td>
</tr>
<tr>
<td>08:</td>
<td>}</td>
</tr>
</tbody>
</table>

(b) JDime in combination with $\Theta_{A-D}$ requires 16 edit operations.

Figure A.10.: Comparison between JDime$_{A-C}$ and JDime$_{A-D}$ for $\Theta_D$. For this example, only the relevant moves are in blue.
A Tree Differencing Appendix

Original code:  Modified code:
01: Object f1(F d) { 0bject f1() {
02:     if (cond2)  if (cond) {
03:         return c(d);  return c2;
04:     }  else {
05:         if (k(d))  return asTestCase(d);
06:             put(d, c(d));
07:         return d;  }
08:     }  return d;
09: }  return d;
10:  }
11:  }
12:  }
13:  }
14: void f4() {  void f3() {
15: }  }

(a) JDime in combination with $\Theta_{A-E}$ requires 29 edit operations.

(b) JDime in combination with $\Theta_{A-F}$ requires 28 edit operations.

Figure A.11.: Comparison between JDime$_{A-E}$ and JDime$_{A-F}$ for $\Theta_{F}$. For this example, only the relevant moves are in blue.
Bibliography


Bibliography


Bibliography


Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Conference/Journal</th>
<th>Page(s)</th>
<th>DOI</th>
</tr>
</thead>
</table>


Bibliography


Bibliography


Bibliography


Bibliography


[263] Karl Pearson, “X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling”, Philosophical Magazine Series 5, volume 50, number 302, pages 157–175, 1900. DOI: 10.1007/978-1-4612-4380-9_2.


Bibliography


Bibliography


Library updates, program errors, and maintenance tasks in general force developers to apply the same code change to different locations within their projects. If the locations are very different to each other, it is very time-consuming to identify all of them. Even with sufficient time, there is no guarantee that a manual search reveals all locations. If the change is critical, each missed location can lead to severe consequences. The manual application of the code change to each location can also get tedious. If the change is larger, developers have to execute several transformation steps for each code location. In the worst case, they forget a required step and thus add new errors to their projects.

To support developers in this task, this thesis presents the recommendation system ARES. It leads to more accurate recommendations compared to previous approaches. ARES achieves this by conserving variations in the training examples in more detail due to its pattern design and by an improved handling of code movements. With the tool C3, this thesis also presents an extension to ARES that allows the extraction of training examples from code repositories. In combination, both tools create a recommendation system that automatically learns code recommendation patterns from repositories.

ARES, C3, and similar tools rely on lists of edit operations to express code changes. However, creating compact (i.e., short) lists of edit operations from data in repositories is difficult. As previous approaches produce too long lists for ARES and C3, this thesis presents a novel tree differencing approach called MTDIFF. The evaluation shows that MTDIFF shortens the edit operation lists compared to other state-of-the-art approaches.