University of Passau
Faculty of Computer Science and Mathematics

Bachelor's Thesis

Sensitivity Analysis of Flexible AST Matching

Author Martin Bauer
bauermar@fim.uni-passau.de

Advisors Prof. Dr.-Ing. Sven Apel
Georg Seibt
Olaf Leßenich

Submission Date 2017/10/06

Bauer, Martin:
Sensitivity Analysis of Flexible AST Matching
Bachelor's Thesis, University of Passau, 2017.

Abstract

The syntactic merge tool for Java programs JDime provides a matching algorithm based on Flexible Tree Matching, which allows to successfully match even complex changes with far less conflicts than an unstructured merge would produce.

This thesis studies the influence of numerous parameters in this matching algorithm, such as weightings in a cost model or an argument for a probability distribution. The research could not only identify a relation better merging results and the individual weightings but also confirm that another parameter has been chosen reasonably.

Furthermore, a filtering mechanism for AST nodes based on syntactic categories is added to the algorithm's implementation. This allows the matchings to be computed up to four times faster than before.

Contents

List of Figures **V**

List of Tables **VI**

List of Listings **VII**

List of Algorithms **VIII**

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Structure of the Thesis . . . 2

2 Background **3**

2.1 Version Control System . . . 3

2.2 Merging Techniques . . . 4

2.3 JDime . . . 6

2.4 Decision Tree . . . 8

3 Weighting Edge Costs in Flexible Tree Matching **9**

4 Altering the Metropolis Algorithm **11**

4.1 Syntactic Categories . . . 11

4.2 Proposing Optimal Matchings . . . 12

5 Merge Tasks **14**

6 Results **19**

6.1 Weighted Edge Costs . . . 19

6.2 Selecting Matchings . . . 25

6.3 Seed Influence . . . 26

6.4 Syntactic Categories . . . 28

7 Conclusion and Future Work **32**

List of Figures

2.1 GNU MERGE conflict 4
2.2 Merge conflict after two independent changes 5
2.3 Simplified ASTs of Figure 2.2 7
2.4 A decision tree based on the data from Table 2.1 8

3.1 A bipartite graph constructed from two trees 9

4.1 The `Artifact` class structure 12

6.1 Distribution of weights and scores 19
6.2 Distribution of weights and scores (top 15 %) 20
6.3 The left AST in a bipartite graph after the matching 21
6.4 The right AST in a bipartite graph after the matching 22
6.5 Better matchings are produced by higher w_r, w_n and w_o values 23
6.6 Better matchings are produced by lower w_a and w_s values 24
6.7 Decision tree based on the mean scores over all merge tasks 24
6.8 Score against probability of success and configurations 25
6.9 Averaged score against probability of success and configurations 25
6.10 Range of the scores when evaluated with different seed values 26
6.11 Seed influence on every merge task 27
6.12 Distribution of weights and scores using syntactic categories 29
6.13 Distribution of weights and scores using syntactic categories (top 15 %) . . 29
6.14 Influence of different seed values on the score using syntactic categories . . 30

List of Tables

2.1 Weekend activities based on weather and money 8

5.1 Size of the bipartite graph in the test scenarios 14

6.1 Size of the bipartite graph with and without syntactic categories 28
6.2 Mean score of the merge tasks with and without categories 30
6.3 Runtime of the merge tasks with and without categories 31

List of Listings

1 Merge task MovedMethod . . . 15
2 Merge task Multiple . . . 15
3 Merge task RenamedMethod . . . 15
4 Merge task ShiftedCode . . . 16
5 Merge task SurroundWithLoop . . . 16
6 Merge task SurroundWithTry . . . 17
7 Merge task ExtractMethod . . . 17
8 Merge task ExtractMethod2 . . . 18

List of Algorithms

1 The assembling of the bipartite graph. 12

CHAPTER 1

Introduction

1.1 Motivation

In projects of reasonable size, there are always multiple different contexts where work happens. Each feature, experiment, or alternative of a product is actually a context of its own: it can be seen as its own “topic”, clearly separated from other topics. Most likely, there is at least one context for the “main” or “production” state, and another context for each feature, experiment, etc. In real-world projects, work always happens in multiple of these contexts in parallel. Keeping the work in those separate contexts is a huge help, but there will come a time when the changes from one context have to be integrated into another one. For example, when a feature has been completed, you usually want to integrate it into the “production” context [Gün14].

In terms of Version Control Systems, such a workflow is called *Feature-Branch-Workflow* with its *branches* referring to the individual contexts. When merging multiple changes together, almost every Version Control System performs a simple *line-based merge*, i.e., it monotonously compares files line by line without ever knowing what the content of those files means. If there are lines where the merge tool gets stuck, the user has to manually resolve the conflicts, even in situations that appear obvious to the user.

An alternative approach is a so-called *structured merge*. This technique tries to get a deep knowledge of the data it is merging, which means it can more easily resolve conflicts than a line-based approach can. Structured merging appears to be a good fit for preventing conflicts when source code is involved because its nature is based on structured abstract syntax trees (AST) [Sei16, Section 1.1].

The merging tool JDime[1] implements — among others — such a merging strategy, based on the *Flexible Tree Matching* algorithm, which contains many parameters that influence

[1] Sven Apel and Olaf Leßenich. *JDime: Structured Merge with Auto-Tuning.* URL: `http://www.infosun.fim.uni-passau.de/se/JDime/` (visited on 10/03/2017).

the quality of the merging outcome. The goal of this thesis is to investigate these influence factors on the effectiveness and efficiency of the merging process.

1.2 Structure of the Thesis

This thesis is organized as follows: The first chapter gives a brief overview over the concepts like Version Control Systems and their various merging strategies. Afterwards, the theoretical foundation of Flexible Tree Matching is outlined in Chapter 3. The next chapter proposes a change to JDime's implementation and presents its stochastical parts. Each investigation in this paper is based on numerous test scenarios, which are described in Chapter 5, followed by the discussion of the results in Chapter 6. Some conclusions and the outline for future work are drawn in the final chapter.

CHAPTER 2

Background

2.1 Version Control System

A Version Control System (VCS) is a software that helps developers work together and maintain a complete history of their source code over time. Developers can easily go back in time and compare earlier versions of their code while minimizing disruption to all team members because the VCS keeps track of every modification to the code in a special kind of database.

Today, version control is an essential instrument for working on a project with several developers, since every member is continually writing new and changing existing source code. Team members working concurrently and even individuals who work on their own can benefit from the ability to work on independent streams of changes. Creating a so-called "branch" keeps multiple streams of work independent from each other while also providing the facility to merge that work back together. In some cases, the merge can be performed automatically, because there is sufficient history information to reconstruct the changes and more importantly the changes do not conflict. In other cases, the changes made in one branch are incompatible with those currently present in the merge destination. A VCS should discover this problem and solve it in an orderly manner. Needless to say, overwriting is no option, as work is lost when using this strategy. That's why most modern version control systems accept a commit of a modified file to the repository only if its original local copy matches the latest revision available in the repository. If someone else has committed a newer version of the file to the repository in the meantime, the developer has to merge this newer file with the locally changed one before a commit [Atl].

2.2 Merging Techniques

When merging multiple artifacts of source code, their differences have to be identified. This separate process is crucial for the quality of the merge. A wrongly detected change of a program element by the compare process — although it is actually unchanged in both versions — might lead to an unnecessary conflict while assembling the unified output in a later step of the merge. Nowadays, Version Control Systems usually perform a textual, line-based merge with a popular example being GNU MERGE. If no conflict occurred during the merge process, the output equals the successfully merged file. In case a conflict was detected, the output is enriched in the relevant places as shown in Figure 2.1 [Inc08].

```
content of merged file
<<<<<<< file1
conflicting lines in file1
=======
conflicting files in file2
>>>>>>> file2
content of merged file
```

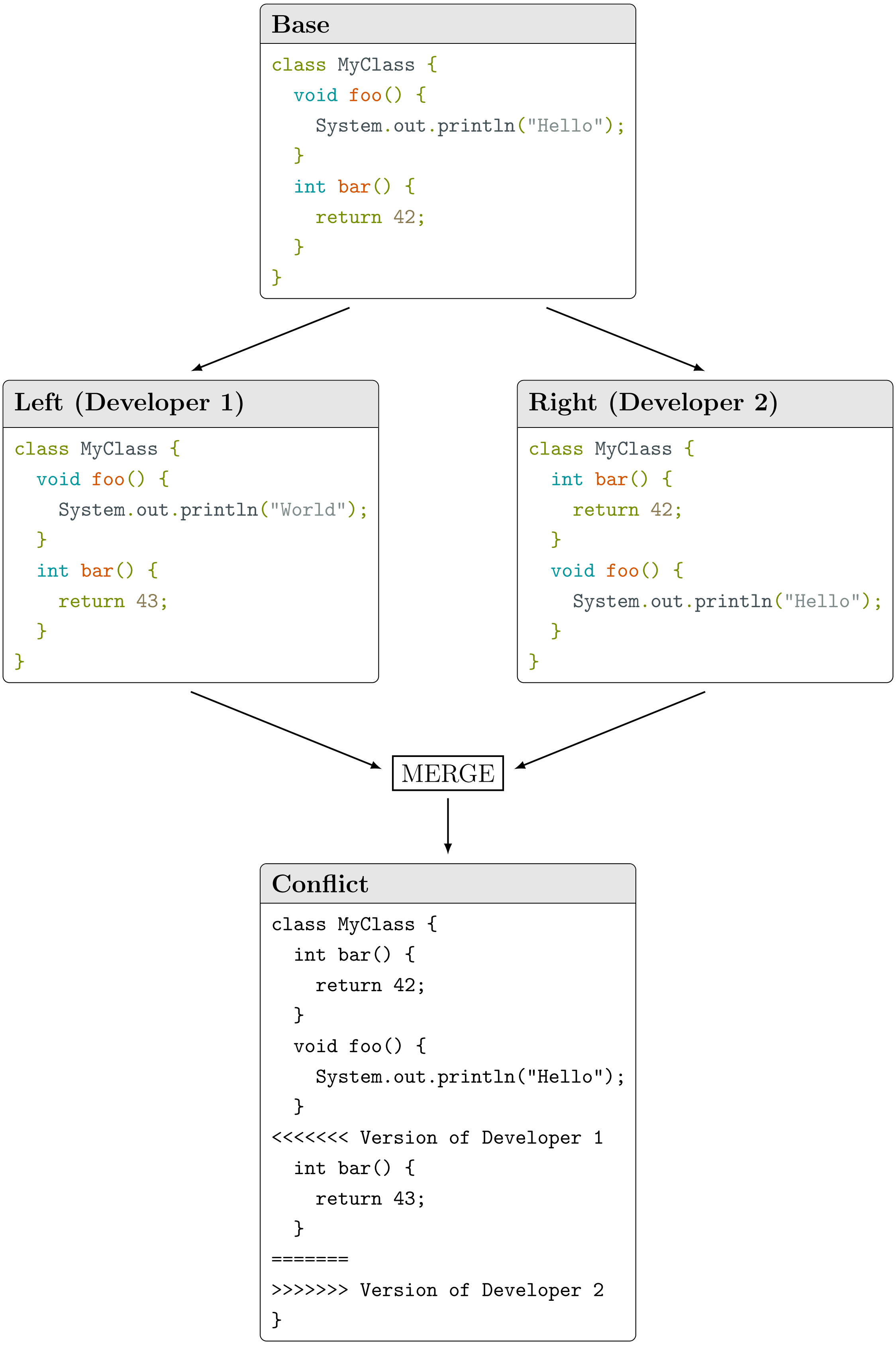
Figure 2.1. GNU MERGE conflict.

The reason for the success of textual merge operations in modern Version Control Systems is its universal applicability and speed. Such a merge can be applied to all non-binary files regardless in size and content. This means, that developers do not need different tools for each programming language to merge their work, but one — the textual merge tool.

However, the downside is, that textual merge is rather weak when it comes to handling merge conflicts. In Java, for example, a very simple change like reordering methods, which has no impact on the semantic behavior, can lead to a lot of conflicts. Those have to be manually resolved in order to complete the merge. Furthermore, (re)formatting of code often produces conflicts, because the position of brackets and the indentation style might be different according to the settings of the developers' editor [Leß12, Section 3.4].

Consider for example the simple Java class in its base version in Listing 2.2. Now, imagine two developers are working independently on this class. The first developer changes the content of the two methods `foo` and `bar`. The second developer doesn't like the order of the methods and swaps them, leaving their content unchanged. What happens, if both commit and merge their work using a textual merge tool? Although the changes are quite simple and reproducible, the merge results in a conflict, which has to be resolved manually.

To overcome the disadvantages of a textual merge, especially to reduce the number of conflicts a user has to resolve by hand, one could use syntactic merge. This structured merge



```
class MyClass {
  void foo() {
    System.out.println("Hello");
  }
  int bar() {
    return 42;
  }
}
```

```
class MyClass {
  void foo() {
    System.out.println("World");
  }
  int bar() {
    return 43;
  }
}
```

```
class MyClass {
  int bar() {
    return 42;
  }
  void foo() {
    System.out.println("Hello");
  }
}
```

```
class MyClass {
  int bar() {
    return 42;
  }
  void foo() {
    System.out.println("Hello");
  }
<<<<<<< Version of Developer 1
  int bar() {
    return 43;
  }
=======
>>>>>>> Version of Developer 2
}
```

Figure 2.2. Two developers are modifying a file at the same time. Once they have finished their work, they try to merge their changes back together, which in this case results in a conflict.

exploits language-specific knowledge and can, therefore, compare and merge software artifacts better than textual merges. The underlying data structure for a syntactic merge is usually an abstract syntax tree, which requires the merge tool to parse the programs in advance and generate the corresponding tree.

Figure 2.3 shows simplified abstract syntax trees for the modified versions of `MyClass` from Listing 2.2 (Left and Right).

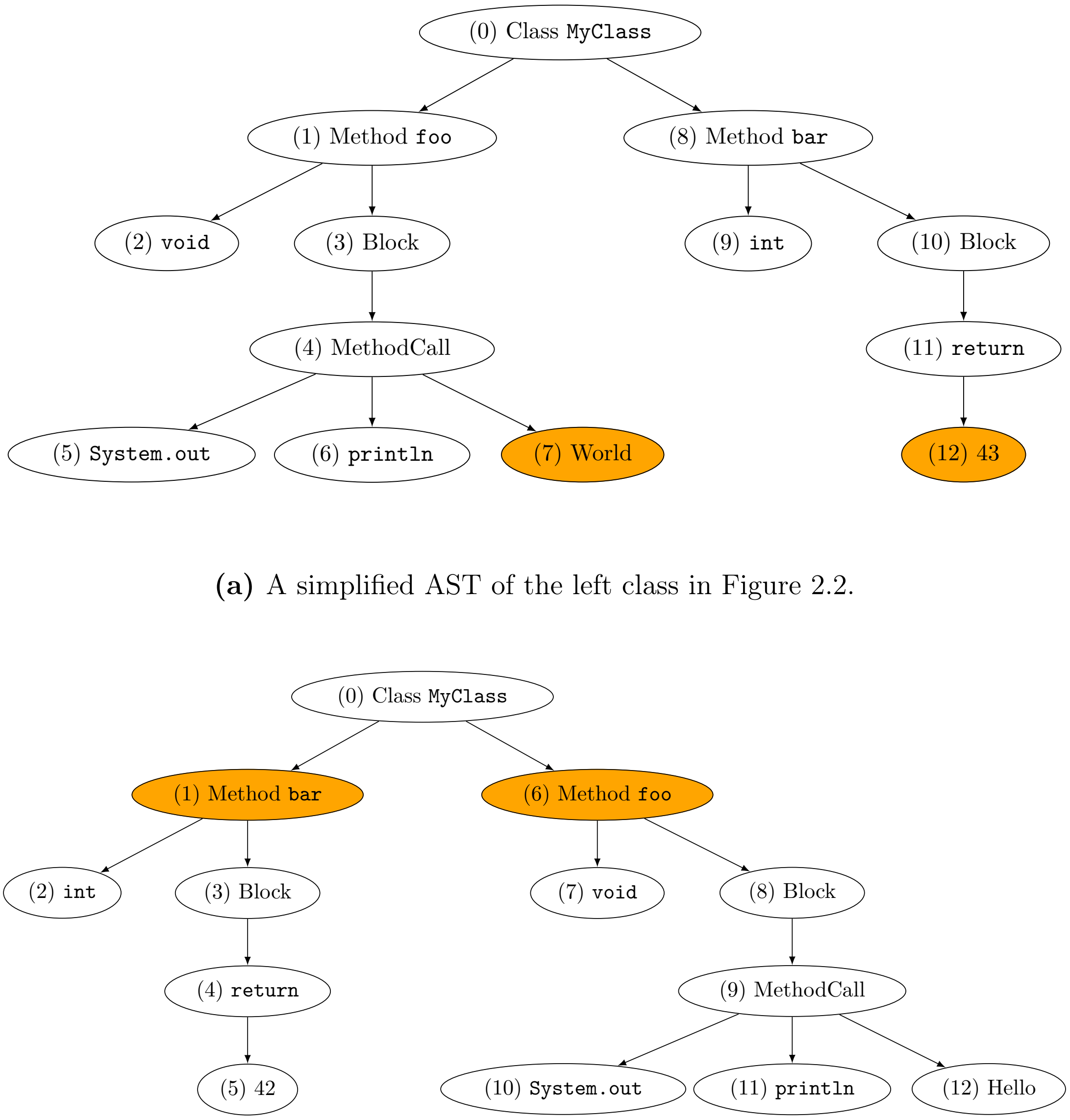
In order to compare two programs for this type of merge, both trees are traversed and the differing nodes are identified. Since the merge is applied to the trees, code formatting is no longer relevant. The previously mentioned ordering conflicts can also be detected as such, and in case of Java programs, simply ignored. Now, the output file can be generated by simply pretty-printing the abstract syntax tree.

This approach, however, has disadvantages as well: Syntactic merges are much slower than textual merges, mostly due to the complexity of their compare algorithms. A syntactic merge requires the input files to be syntactically correct, otherwise, the parser is not able to build the AST. Furthermore, it is restricted to certain file types since it uses syntactic knowledge of the programming languages in which the programs to be merged are written [Leß12, Section 3.5].

2.3 JDime

JDime is a structured merge tool that can perform a syntactic three-way merge on Java programs. It focuses on providing a structured merge that can resolve more conflicts than an unstructured merge, but still retains a similar runtime. As Leßenich, Apel, and Lengauer [LAL15] proved, JDime is a viable alternative to traditional line-based merging. An auto-tuning approach that first attempts a line-based merge and then a structured merge if there are conflicts, allows JDime to be up to 92 times faster than purely structured tools, 10 times on average.

In an earlier version, JDime did not have the ability to solve conflicts that involved nodes on different levels in the AST. While this had no effect on merging of reordered or reformatted code, it lacked the capability to merge situations like code being surrounded by constructs such as loops or conditional statements. The master's thesis of Seibt [Sei16], therefore, added an implementation of Flexible Tree Matching [Kum+11] to JDime, which relaxes the rigid requirements of the tree matching algorithm in favor of a tunable formulation in which the role of hierarchy can be controlled.



(a) A simplified AST of the left class in Figure 2.2.

(b) A simplified AST of the right class in Figure 2.2.

Figure 2.3. Simplified ASTs of Listing 2.2. Notice that the reordering of the methods resulted in a simple swap in the root node's children. The change in the value of the method parameters introduced two new nodes 7 and 12, that replace their predecessors.

2.4 Decision Tree

A Decision Trees is a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from data features [dev17]. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally created. The final result is a tree with *decision nodes* and *leaf nodes*. The deeper the tree, the more complex the decision rules, and the fitter the model [Say]. Consider the following example inspired by Hawks [Haw]: You are making your weekend plans, but there are a few unknown factors that will determine what you can and can't do. Depending on the weather and your current financial situation you come up the following table.

Table 2.1. Weekend activities based on weather and money.

Weather	Money	Activity
sunny	rich	play tennis
sunny	poor	play tennis
rainy	poor	stay in
windy	poor	cinema
windy	rich	shopping

The advantage of the decision tree is its graphical representation of possible solutions to a decision based on certain conditions. Rather studying a table with possibly thousands of entries the decision tree helps you "see" the (potential) structure behind the data. Figure 2.4 gives an example of a decision tree based on the data from Table 2.1. The decision nodes are represented by squares and leafs by ellipses.

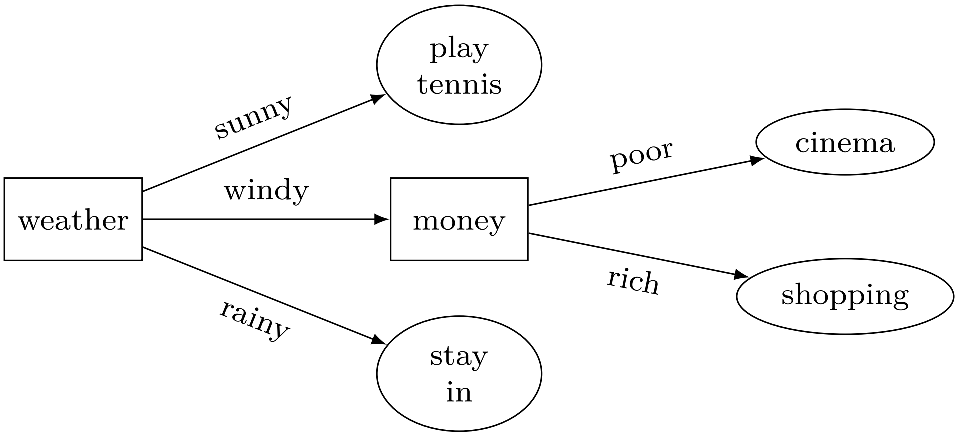


Figure 2.4. A decision tree based on the data from Table 2.1.

CHAPTER 3

Weighting Edge Costs in Flexible Tree Matching

In the structural merging domain, one has to find an injective binary relation M between two labeled trees L and R. That means M is a strict subset of the cartesian product of the trees' nodes because no node from L or R occurs in more than one element of M. The matching M is then interpreted as a bipartite graph between the nodes of the trees (Figure 3.1).

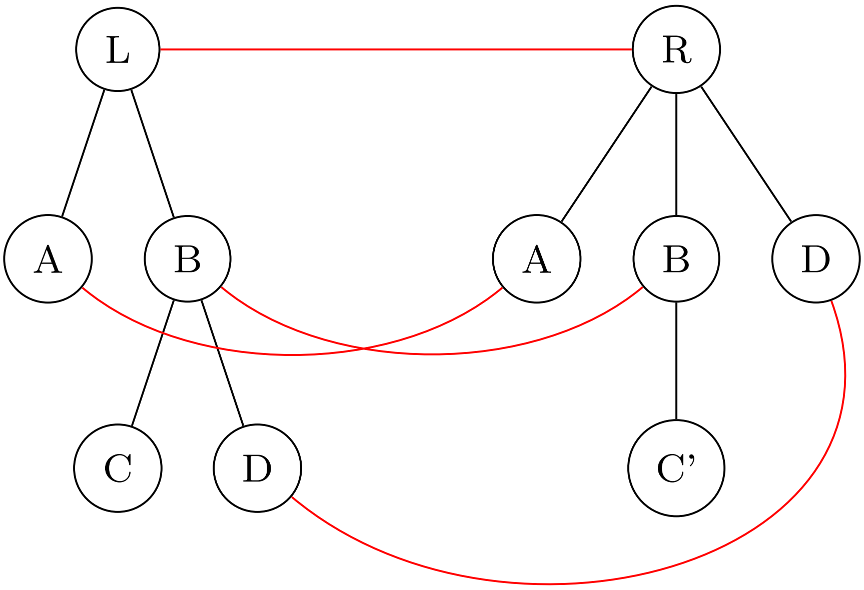


Figure 3.1. A bipartite graph constructed from two trees.

The edges represent operations transforming L into R. A matching between nodes with a different label corresponds to a renaming; nodes that are not mapped represent deletion and insertion depending on the tree, they are missing in.

The Flexible Tree Matching algorithm is based on a cost model which uses weighted costs to describe properties of a matching. Given a function $c(l, r)$ calculating the cost of a matching $(l, r) \in M$, i.e., the cost of an edge in the graph, the problem of tree matching is finding a set of matchings M that minimizes the sum of the costs defined as

$$c(M) = \frac{1}{|L| + |R|} \sum_{l,r \in M} c(l, r).$$

The cost calculation for a single matching (l, r) is split up into multiple parts:

$$c(l,r) = \begin{cases} w_n & \text{if } l = \bigotimes_L \text{ or } r = \bigotimes_R, \\ c_r(l,r) + c_a(l,r) + c_s(l,r) & \text{otherwise.} \end{cases}$$

$\bigotimes_L$ and $\bigotimes_R$ denote auxiliary no-match nodes and allow weighting the cost with w_n when a node could not be matched. If (l, r) represents an actual match, its cost equals the sum of all three c_r, c_a and c_s.

The relabeling term c_r represents the cost of matching two nodes that have different labels.

$$c_r(l,r) = \begin{cases} 0 & \text{if } l \text{ and } r \text{ match,} \\ w_r & \text{otherwise.} \end{cases}$$

The ancestry cost c_a describes violating ancestry relationships between the tree nodes in L and R. This cost function examines the children of L and R to count the ones not being matched to children of the opposite node.

The sibling cost c_s penalizes edges that fail to preserve sibling relationships between trees. The two costs c_a and c_s are internally weighted by w_a and w_s respectively. Since the actual cost calculation is not germane in this thesis, it is omitted here and simply denoted by $\hat{c}_a$ and $\hat{c}_s$. See [Kum+11, section 3] for the mathematical details.

$$c_a(l,r) = w_a \cdot \hat{c}_a(l,r,M)$$
$$c_s(l,r) = w_s \cdot \hat{c}_s(l,r,M)$$

Furthermore, in an AST, the order of a node's children may be relevant, which lead to the addition of a new ordering cost by Seibt [Sei16]. The function c_o and its weight w_o penalize matchings that introduce an ordering which is being violated by other matchings. See [Sei16, section 3.5.2].

$$c_o(l,r) = w_o \cdot \hat{c}_o(l,r,M)$$

Thus, the cost of a matching (l, r) can be expressed as

$$c(l,r) = \begin{cases} w_n & \text{if } l = \bigotimes_L \text{ or } r = \bigotimes_R, \\ c_r(l,r) + c_a(l,r) + c_s(l,r) + c_o(l,r) & \text{otherwise.} \end{cases}$$

To generate optimal matchings, the Flexible Tree Matching algorithm requires a domain-specific configuration via its parameters. The aim of this thesis is to analyze the influence of those weightings w_n, w_r, w_a, w_s and w_o on the matchings the algorithm produces.

CHAPTER 4

Altering the Metropolis Algorithm

In JDime, the Metropolis algorithm [CG95] is responsible for approximating the set of matchings with the lowest cost. The algorithm starts by assembling the complete bipartite graph G and calculates the cost bounds[1] for all matchings. These matchings M are then sorted by increasing bound. Every iteration of the Metropolis algorithm proposes a new set of matchings by choosing an index $j \in [1, |M|]$ and fixing the first j matchings in M. Those fixed matchings are then completed to a set that satisfies the constraint that for every node in the left and right tree there is exactly one matching containing that node.

4.1 Syntactic Categories

When JDime builds the bipartite graph, it will try to match every node from the left AST with every node from the right AST, even though in many cases, this is unnecessary. For instance, a node representing a class declaration does not need to be matched to variable declaration nodes. Therefore, a filtering mechanism is proposed such that JDime considers only matchings with the same syntactic category.

At the heart of the JDime architecture is the `Artifact` class, which — in the current version — represents either a file in a directory tree or a node in an AST. The matching algorithm operates on any kind of `Artifact`, which means that in order to support syntactic categories, each artifact has to be assigned a category. This is implemented via a new abstract method `categoryMatches(Artifact)` in the abstract base class `Artifact` (Figure 4.1) that is implemented by concrete artifacts with regards to their needs.

`ASTNodeArtifact` uses the Java class of the AST nodes created by ExtendJ[2] as its syntactic category. This leads to a high granularity in categories because those classes boil down to expression level (e.g. `MethodDecl`, `ThrowStmt`, `AndBitwiseExpr`, etc.).

[1] Sei16, Section 3.2.

[2] http://jastadd.org/web/extendj/

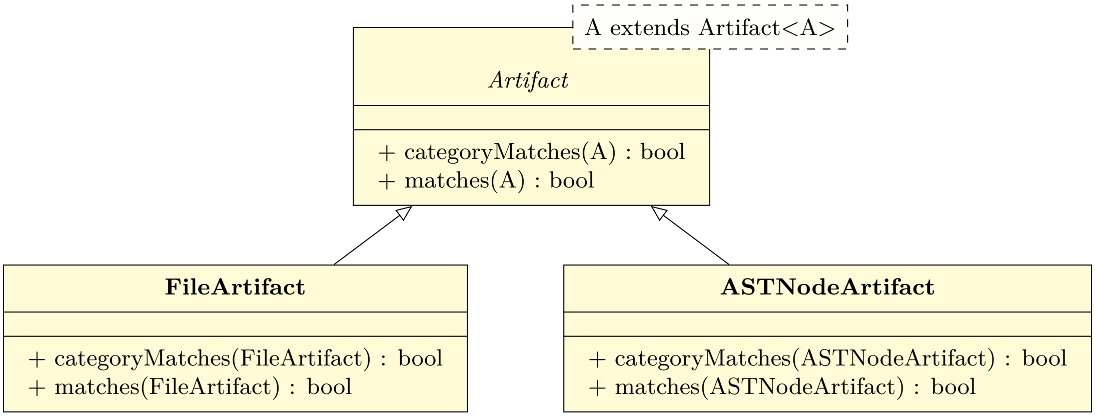

A extends Artifact<A>
Artifact
+ categoryMatches(A) : bool
+ matches(A) : bool
FileArtifact
+ categoryMatches(FileArtifact) : bool
+ matches(FileArtifact) : bool
ASTNodeArtifact
+ categoryMatches(ASTNodeArtifact) : bool
+ matches(ASTNodeArtifact) : bool

Figure 4.1. The `Artifact` class structure.

The `FileArtifact`'s categories are based on the type of the file, i.e., directories and regular files.

Since every node in the bipartite graph is an `Artifact`, the `CostModelMatcher` can identify compatible nodes and only add a matching to the bipartite graph, if the categories of those two nodes match or one of them is a no-match node, i.e., the edge represents an addition or deletion. Algorithm 1 shows the procedure for the assembling of the graph.

Algorithm 1: The assembling of the bipartite graph.

```
Input: left AST, right AST
Output: bipartite graph
bipartiteGraph ← empty graph;
foreach lNode in left AST including no-match node do
    foreach rNode in right AST including no-match node do
        edge ← (lNode, rNode);
        if isAddition(edge) or isDeletion(edge) or categoryMatches(lNode, rNode)
         then
            add(bipartiteGraph, edge);
        end
    end
end
```

4.2 Proposing Optimal Matchings

Once the bipartite graph is created, JDime can fix a portion of the matchings, which is defined by an index in the list of possible matchings. Instead of always fixing the matching with the lowest cost (index 0), a random index is chosen. This method ensures that less costly matchings are more likely to be fixed as they have a lower index in the sorted list, but still allows other matchings with a higher cost to be fixed. The randomness in JDime is implemented using a negative binomial distribution where $r = 1$. The probability function is defined as

$$P(\{X = k\}) = p(1 - p)^k.$$

This thesis investigates the choice of $p = 0.7$ to see, whether the correctness of the algorithm can be improved by using different values.
In addition to the probability of success, the implementation of Flexible Tree Matching uses a seed to initialize a pseudorandom number generator for the negative binomial distribution, which is hard-coded as 42. Again, the influence of different values on the correctness of the matchings is inspected.

CHAPTER 5

Merge Tasks

In order to evaluate a particular matching, multiple test scenarios — hereinafter referred to as merge tasks — have been created. Each merge task is associated with a "perfect" matching (reference matching), that is constructed by hand. To quantify the quality of computed matchings against their reference, the Jaccard similarity coefficient is used. For two sets A and B it is defined as

$$J(A,B) = \begin{cases} 1 & A = B = \emptyset, \\ \frac{|A \cap B|}{|A \cup B|} & \text{otherwise.} \end{cases}$$

The following listings each show two versions of a class, which then have to be matched by the Flexible Tree Matching algorithm. Seibt [Sei16] used an evolutionary algorithm to find an optimal weighting configuration based on several test scenarios. In addition to the six merge tasks, which have been part of the learning process, several new ones are added, essentially to test the algorithm on ASTs with more nodes. Table 5.1 gives an idea of the bipartite graphs' size corresponding to the merge tasks.

Table 5.1. Size of the bipartite graph in the test scenarios.

Merge Task	Left AST	Right AST	# Nodes	# Edges
MovedMethod	43	43	86	1,849
Multiple	55	55	110	3,025
RenamedMethod	23	23	46	529
ShiftedCode	62	52	114	3,224
SurroundWithLoop	72	51	123	3,672
SurroundWithTry	64	47	111	3,008
ExtractMethod	109	107	216	11,663
ExtractMethod2	165	141	306	23,265

The merge task MovedMethod rearranges two methods. Despite the semantic equivalence of the two files, traditional unstructured tools would produce a conflict when faced with the reordering of methods in a class. In the AST, this change is represented by the reordering of the children of the nodes representing the class declaration.

```java
public class MovedMethod {
  private void foo() {
    System.out.println("Hello");
  }

  public int bar() {
    return 42;
  }
}
```

```java
public class MovedMethod {
  public int bar() {
    return 42;
  }

  private void foo() {
    System.out.println("Hello");
  }
}
```

Listing 1. Merge task MovedMethod.

The merge task Multiple is a self-test, representing the easiest possible matching scenario. The class contains multiple declarations and should be matched against itself.

```java
class Multiple {
  void foo() {
    int i = 0;
    int j = 1;
    String s = "";
    i = 0;
    j = 1;
  }
}
```

Listing 2. Merge task Multiple.

In RenamedMethod the method cannot be matched by unstructured merging since the method declaration nodes have different labels. JDime should match the method declaration nodes and all their children.

```java
class Foo {
  int getAnswer() {
    return 42;
  }
}
```

```java
class Foo {
  int getResult() {
    return 42;
  }
}
```

Listing 3. Merge task RenamedMethod.

In the ShiftedCode task, the lines 9 and 10 are surrounded by an if block. The nodes previously representing the body of the `bar` method are now children of an if statement block. In addition to this conditional statement, another return point has been added.

```java
1  import java.util.List;
2
3  class Bar {
4    List<String> l;
5
6    int bar() {
7      String s = l.get(0);
8      return s.length();
9    }
10 }
```

```java
1  import java.util.List;
2
3  class Bar {
4    List<String> l;
5
6    int bar() {
7      if (l != null) {
8        String s = l.get(0);
9        return s.length();
10     }
11     return 0;
12   }
13 }
```

Listing 4. Merge task ShiftedCode.

In the SurroundWithLoop task, a code fragment is embedded in a for-loop construct. In the AST, this means removing several layers of nodes representing the loop before the subtree of the surrounded fragment.

```java
public class SWLoop {
  public boolean isOnline() {
    boolean online;
    online = check();
    return online;
  }

  public boolean check() {
    return true;
  }
}
```

```java
public class SWLoop {
  public boolean isOnline() {
    boolean online;
    for (int i = 0; i < 10; i++) {
      online = check();
    }
    return online;
  }

  public boolean check() {
    return true;
  }
}
```

Listing 5. Merge task SurroundWithLoop.

SurroundWithTry represents a similar problem as Listing 5 does, however with the added complexity of a catch block. JDime is expected to match the surrounded code but not the additional nodes of the catch block.

```java
public class SurroundWithTry {
  public void doSmth() {
    String s = ex();
  }

  private String ex() {
    throw new RuntimeException();
  }
}
```

```java
public class SurroundWithTry {
  public void doSmth() {
    try {
      String s = ex();
    } catch (RuntimeException e) {
      e.printStackTrace();
    }
  }

  private String ex() {
    throw new RuntimeException();
  }
}
```

Listing 6. Merge task SurroundWithTry.

In the ExtractMethod task two identical if statements are extracted into a new method. JDime has to match exactly one of the left AST subtrees against the same AST subtree in the right file, which is several layers deeper. The other AST subtree should not be matched.

```java
import java.util.List;
import java.util.LinkedList;

class Stack {
  List<Integer> stack = new LinkedList<>();

  public Integer pop() {
    if (stack.size() > 1) {
      System.out.println("output");
    }
    return stack.remove(stack.size()-1);
  }

  public void push(Integer elem) {
    if (stack.size() > 1) {
      System.out.println("output");
    }
    stack.add(elem);
  }
}
```

```java
import java.util.List;
import java.util.LinkedList;

class Stack {
  List<Integer> stack = new LinkedList<>();

  public Integer pop() {
    helperMethod();
    return stack.remove(stack.size()-1);
  }

  public void push(Integer elem) {
    helperMethod();
    stack.add(elem);
  }

  private void helperMethod() {
    if (stack.size() > 1) {
      System.out.println("output");
    }
  }
}
```

Listing 7. Merge task ExtractMethod.

The ExtractMethod2 task is similar to the previous merge task, except that the if statement is a lot bigger and a method parameter is added to the extracted method. This merge task serves as an example for ASTs with many nodes.

```java
import java.util.List;
import java.util.LinkedList;

class Stack {
  List<Integer> stack = new LinkedList<>();

  public Integer pop() {
    if (stack.size() > 1) {
      System.out.println("output");
    } else if (stack.size() > 10) {
      System.out.println("output2");
    } else {
      System.out.println("pop");
    }
    return stack.remove(stack.size()-1);
  }

  public void push(Integer elem) {
    if (stack.size() > 1) {
      System.out.println("output");
    } else if (stack.size() > 10) {
      System.out.println("output2");
    } else {
      System.out.println("push");
    }
    stack.add(elem);
  }
}
```

```java
import java.util.List;
import java.util.LinkedList;

class Stack {
  List<Integer> stack = new LinkedList<>();

  public Integer pop() {
    helper("pop");
    return stack.remove(stack.size()-1);
  }

  public void push(Integer elem) {
    helper("push");
    stack.add(elem);
  }

  private void helper(String msg) {
    if (stack.size() > 1) {
      System.out.println("output");
    } else if (stack.size() > 10) {
      System.out.println("output2");
    } else {
      System.out.println(msg);
    }
  }
}
```

Listing 8. Merge task ExtractMethod2.

CHAPTER 6

Results

6.1 Weighted Edge Costs

In total 3750 weighting configurations have been evaluated. Each parameter value for w_r, w_n, w_a, w_s and w_o was taken from a uniform distribution between 0 and 1. Then every configuration was evaluated with the merge tasks described in Chapter 5. The resulting data is presented in Figure 6.1. Since the weight values and the corresponding scores are in the range of 0 to 1, both can be shown in the same axis range.

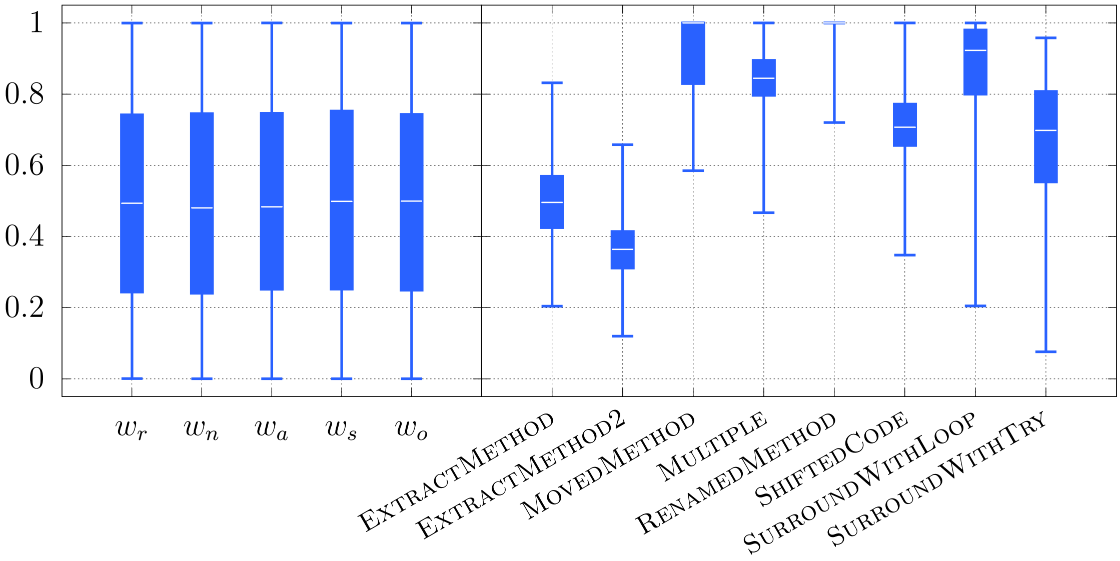


Figure 6.1. Distribution of weights and scores.

The first five boxes — the weight values — show indeed that every parameter value is uniformly distributed over $[0, 1]$. This guarantees that no accumulation in the space of the weights is investigated, but a representative subset of the entire five-dimensional one. When looking at the scores' boxes, you'll notice that RenamedMethod is correctly matched for almost every configuration. The actual box is hard to see, because it is so small, but it is located at the top of the upper whisker. In fact, this merge task is correctly matched in 91.8 % of the configurations. In this context, “correctly” means every single node of the ASTs is matched exactly as the reference matching specifies. MovedMethod also performs relatively well: in around 71.1 % of the cases, the rearranging has been fully detected and matched. The self test Multiple is matched at least to 50 % for every configuration. The other merge tasks show, that 50 % of the cases result in similar scores, but the other half is almost unpredictable. Especially SurroundWithLoop and SurroundWithTry take nearly every value between 0 and 1, meaning the matching can be good or completely useless, depending on the weighting parameters.

The most important statement from this figure, however, is taken from the boxes of ExtractMethod and ExtractMethod2. Despite that almost 4000 different weights have been tested, not a single one reached a score of 0.835 or above. ExtractMethod2 performs even worse: not even scores better than 0.659 are reached, regardless of the weighting parameters.

Since in reality, a merge tool like JDime needs to perform adequately in all these situations, the mean score matters as well. If we raise the bar and look only at the best[1] 15 % of the configurations, the distribution of the parameter values shifts (Figure 6.2).

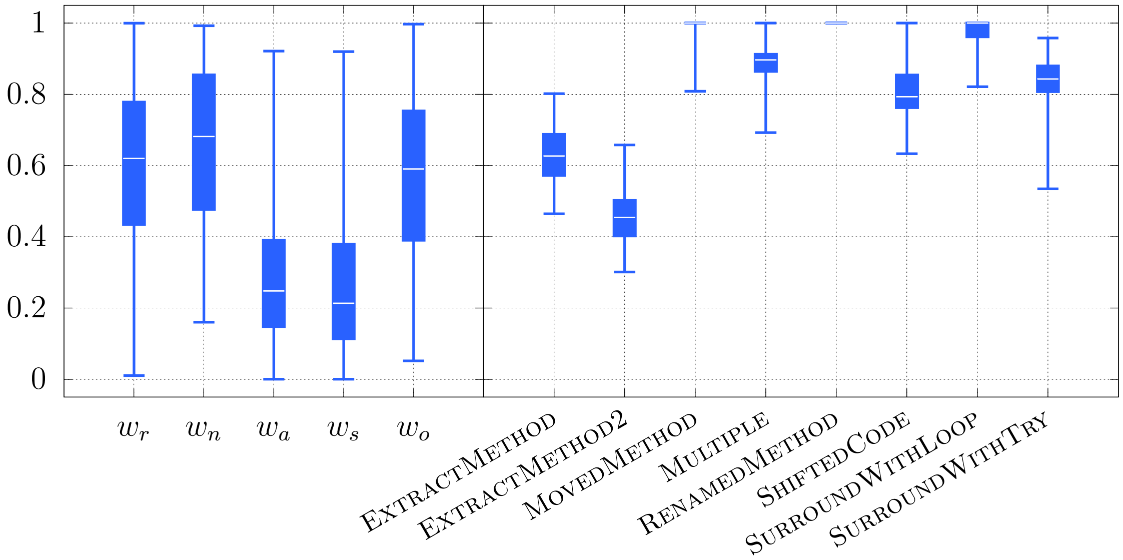
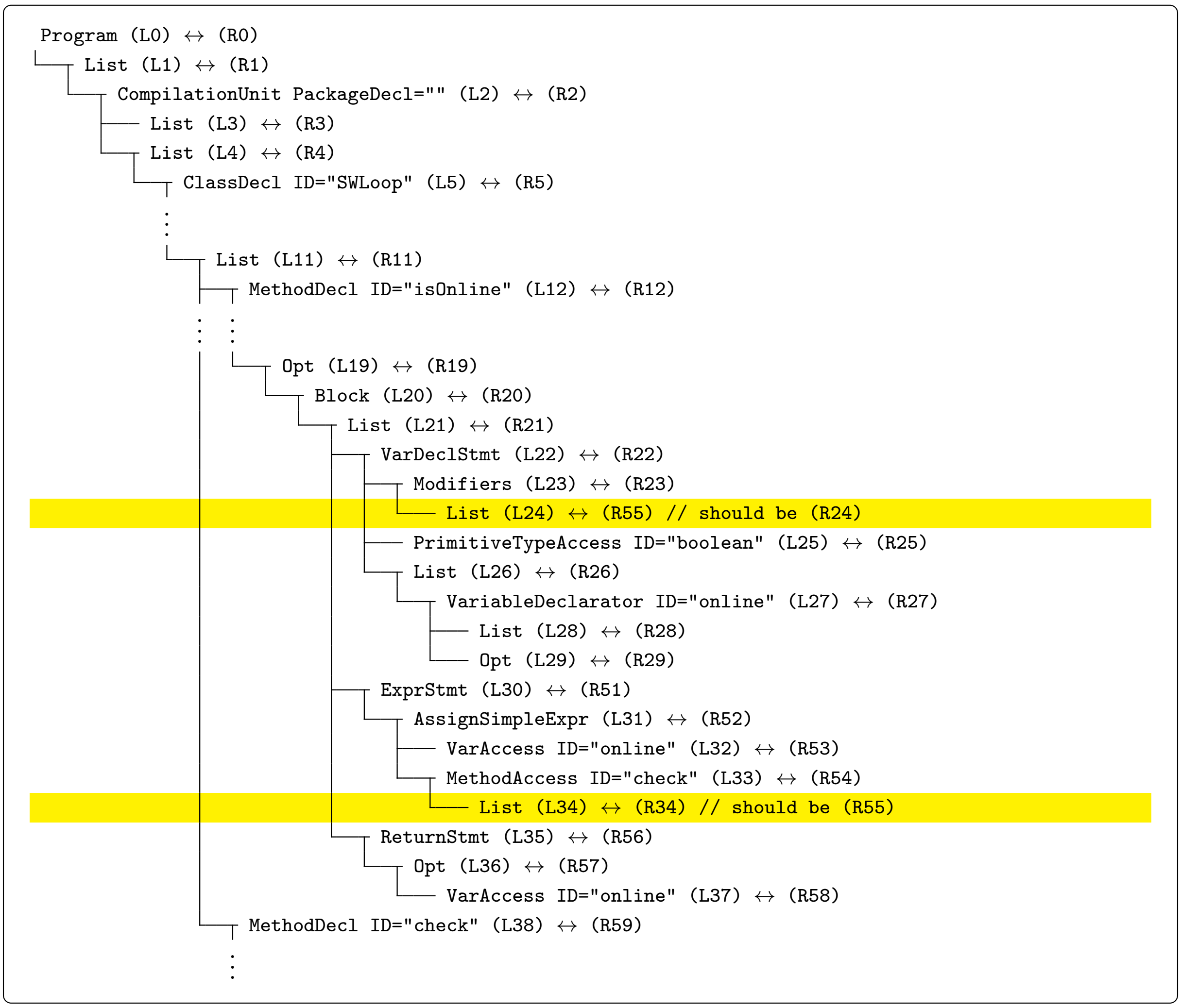


Figure 6.2. Distribution of weights and scores (top 15 %).

The fact that both ExtractMethod tasks perform worse than all other merge tasks manifests here even more. All other merge tasks lead to average scores better than 0.8 %, while ExtractMethod2 doesn’t even get near that with 75 % of its scores worse than 0.5. MovedMethod could improve to match correctly in 92.3 % of the cases. RenamedMethod even fully matched for all the configurations. However, the range of the resulting scores is still relatively large, e.g., about 0.424 for SurroundWithTry.

[1] based on the mean score over all 8 merge tasks

Sources of Error When studying these matchings specifically, one can observe different types of wrong matchings. Apart from simple incorrect matchings, e.g., a method declaration matching with a variable declaration, the troublemakers are to a large extent `List` nodes. For instance, a node representing a method declaration has a child node for its modifiers. As there can be several for one method, this node has multiple children, each representing one modifier. Of course, the same applies for class declarations, field declarations, etc. These `List` nodes are often matched among each other, leading to a decreased score. The figures 6.3 and 6.4 show an excerpt from the bipartite graph for the merge task SurroundWithLoop in plaintext, separated into the left and right ASTs. The highlighted lines mark nodes which are matched incorrectly.

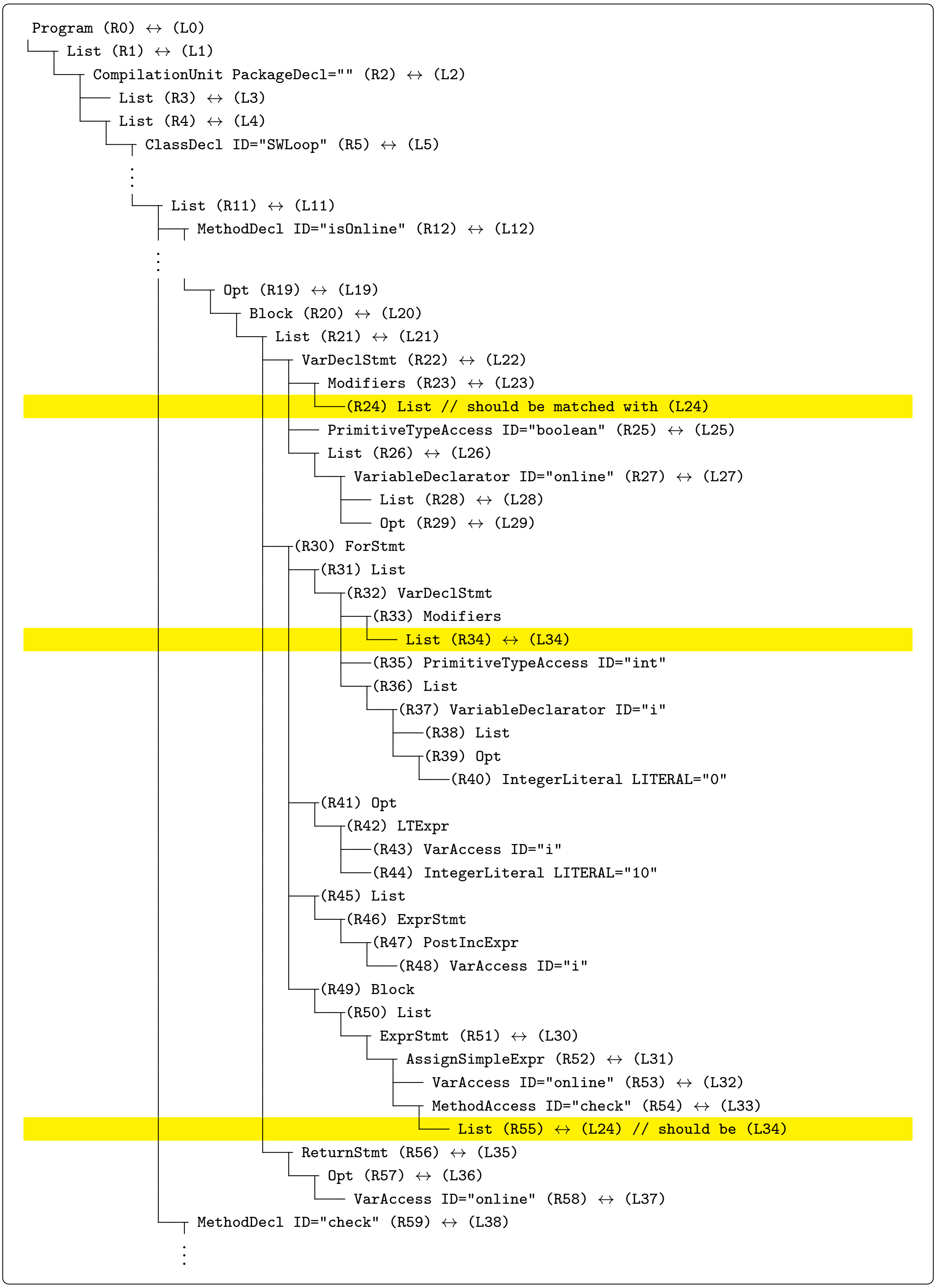


```
Program (L0) ↔ (R0)
└─┬ List (L1) ↔ (R1)
  └─┬ CompilationUnit PackageDecl="" (L2) ↔ (R2)
    ├── List (L3) ↔ (R3)
    └─┬ List (L4) ↔ (R4)
      └─┬ ClassDecl ID="SWLoop" (L5) ↔ (R5)
        ⋮
        └─┬ List (L11) ↔ (R11)
          ├─┬ MethodDecl ID="isOnline" (L12) ↔ (R12)
          ⋮ ⋮
          │ └─┬ Opt (L19) ↔ (R19)
          │   └─┬ Block (L20) ↔ (R20)
          │     └─┬ List (L21) ↔ (R21)
          │       ├─┬ VarDeclStmt (L22) ↔ (R22)
          │       │ ├─┬ Modifiers (L23) ↔ (R23)
          │       │ │ └── List (L24) ↔ (R55) // should be (R24)
          │       │ ├── PrimitiveTypeAccess ID="boolean" (L25) ↔ (R25)
          │       │ └─┬ List (L26) ↔ (R26)
          │       │   └─┬ VariableDeclarator ID="online" (L27) ↔ (R27)
          │       │     ├── List (L28) ↔ (R28)
          │       │     └── Opt (L29) ↔ (R29)
          │       ├─┬ ExprStmt (L30) ↔ (R51)
          │       │ └─┬ AssignSimpleExpr (L31) ↔ (R52)
          │       │   ├── VarAccess ID="online" (L32) ↔ (R53)
          │       │   └─┬ MethodAccess ID="check" (L33) ↔ (R54)
          │       │     └── List (L34) ↔ (R34) // should be (R55)
          │       └─┬ ReturnStmt (L35) ↔ (R56)
          │         └─┬ Opt (L36) ↔ (R57)
          │           └── VarAccess ID="online" (L37) ↔ (R58)
          └─┬ MethodDecl ID="check" (L38) ↔ (R59)
            ⋮
```

Figure 6.3. The left AST in a bipartite graph after the matching.

Node `(L24)` holds the modifiers of the method `isOnline` and is not changed when a statement inside this method is surrounded by a loop. Despite that, it is matched with `(R55)` — a `List` node holding the parameters for a `check` method call. Meanwhile, node `(R24)` is assigned to the no-match node.

Due to the injective relation of the graph, node `(R55)` cannot be used anymore for its



```
Program (R0) ↔ (L0)
└─List (R1) ↔ (L1)
  └─CompilationUnit PackageDecl="" (R2) ↔ (L2)
    ├─List (R3) ↔ (L3)
    └─List (R4) ↔ (L4)
      └─ClassDecl ID="SWLoop" (R5) ↔ (L5)
        ⋮
        └─List (R11) ↔ (L11)
          ├─MethodDecl ID="isOnline" (R12) ↔ (L12)
          ⋮
          │ └─Opt (R19) ↔ (L19)
          │   └─Block (R20) ↔ (L20)
          │     └─List (R21) ↔ (L21)
          │       ├─VarDeclStmt (R22) ↔ (L22)
          │       │ ├─Modifiers (R23) ↔ (L23)
          │       │ │ └─(R24) List // should be matched with (L24)
          │       │ ├─PrimitiveTypeAccess ID="boolean" (R25) ↔ (L25)
          │       │ └─List (R26) ↔ (L26)
          │       │   └─VariableDeclarator ID="online" (R27) ↔ (L27)
          │       │     ├─List (R28) ↔ (L28)
          │       │     └─Opt (R29) ↔ (L29)
          │       ├─(R30) ForStmt
          │       │ ├─(R31) List
          │       │ │ └─(R32) VarDeclStmt
          │       │ │   ├─(R33) Modifiers
          │       │ │   │ └─List (R34) ↔ (L34)
          │       │ │   ├─(R35) PrimitiveTypeAccess ID="int"
          │       │ │   └─(R36) List
          │       │ │     └─(R37) VariableDeclarator ID="i"
          │       │ │       ├─(R38) List
          │       │ │       └─(R39) Opt
          │       │ │         └─(R40) IntegerLiteral LITERAL="0"
          │       │ ├─(R41) Opt
          │       │ │ └─(R42) LTExpr
          │       │ │   ├─(R43) VarAccess ID="i"
          │       │ │   └─(R44) IntegerLiteral LITERAL="10"
          │       │ ├─(R45) List
          │       │ │ └─(R46) ExprStmt
          │       │ │   └─(R47) PostIncExpr
          │       │ │     └─(R48) VarAccess ID="i"
          │       │ └─(R49) Block
          │       │   └─(R50) List
          │       │     └─ExprStmt (R51) ↔ (L30)
          │       │       └─AssignSimpleExpr (R52) ↔ (L31)
          │       │         ├─VarAccess ID="online" (R53) ↔ (L32)
          │       │         └─MethodAccess ID="check" (R54) ↔ (L33)
          │       │           └─List (R55) ↔ (L24) // should be (L34)
          │       └─ReturnStmt (R56) ↔ (L35)
          │         └─Opt (R57) ↔ (L36)
          │           └─VarAccess ID="online" (R58) ↔ (L37)
          └─MethodDecl ID="check" (R59) ↔ (L38)
            ⋮
```

Figure 6.4. The right AST in a bipartite graph after the matching.

intrinsic value: to be the counterpart for node `(L34)`. There are two options left for this node: either matching with the no-match node or matching with another node from the right AST. As it appears, the algorithm chose the later and matched `(L34)` with the exact same position on the right — also a `List` node, but with a different meaning. Unfortunately, the nodes `(R30)` to `(R50)` belong to the added for-loop and therefore should not be matched at all.

Weighting Tendency When looking at the weightings themselves, one can see a tendency of increasing or decreasing. The following figures show the shifting of each weight, with the light blue area representing 50 % of the weightings and the dark blue line the average. The bottom axis shows, that the mean score over all merge tasks reaches from 0.535 to 0.859. Note that this axis has no standard scale (linear, logarithmic, etc.). The top axis displays the appropriate size of the dataset in linear scale. The peeks at the right-hand side of each plot are due to the smaller dataset and therefore lead to a bigger variance in the scores.

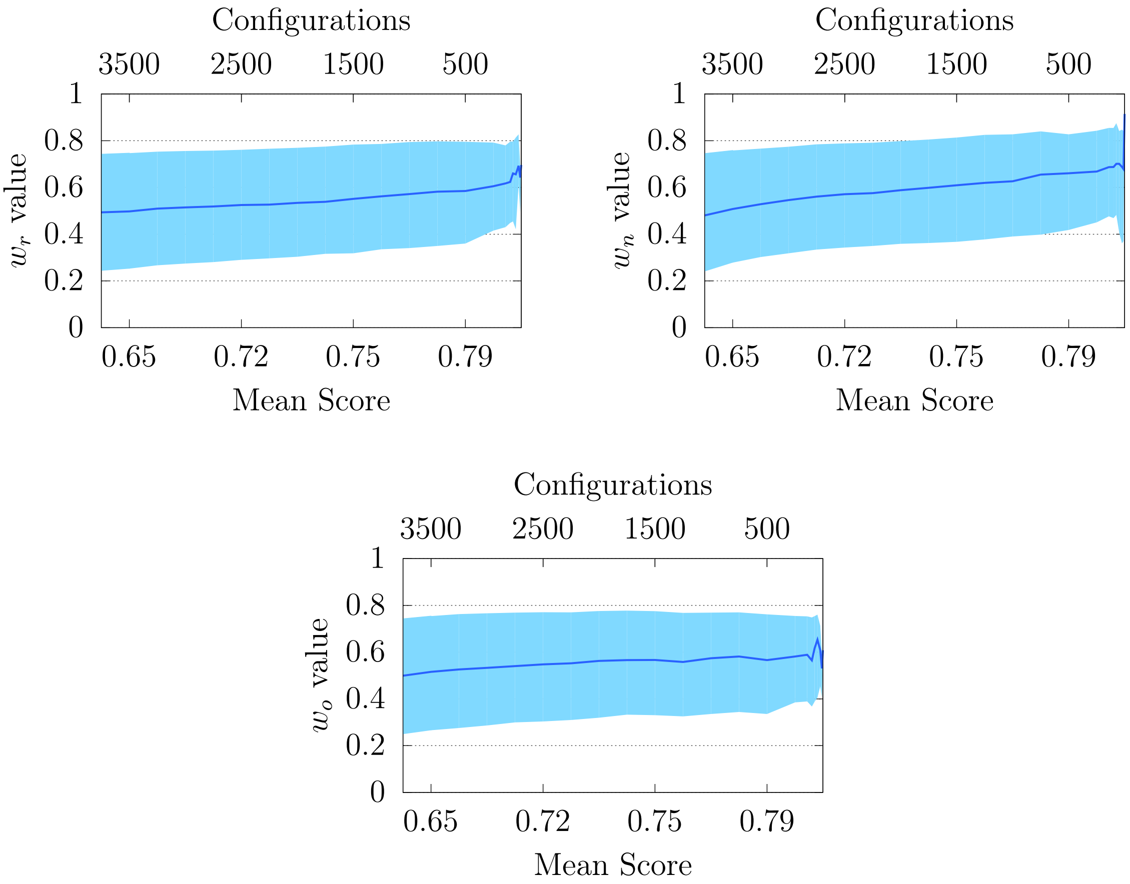


Figure 6.5. Better matchings are produced by higher w_r, w_n and w_o values.

While w_r, w_n and w_o slightly increase as the score get higher, w_a and w_s definitely produce better results with lower values. Also, the range of the first and third quartile becomes smaller and smaller.

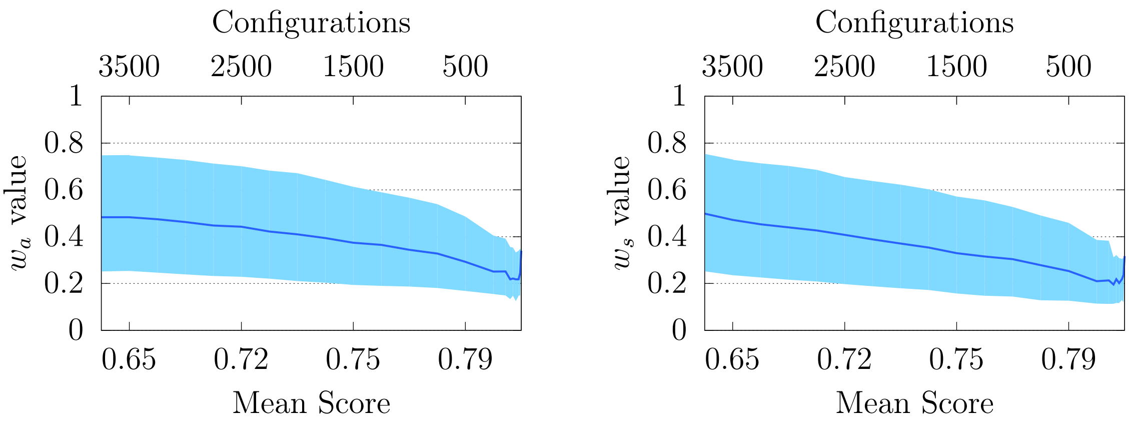


Figure 6.6. Better matchings are produced by lower w_a and w_s values.

Machine Learning To help identify a strategy most likely to reach a high score, a model has been learned by SCIKIT-LEARN [Ped+11]. The resulting decision tree is presented in Figure 6.7.

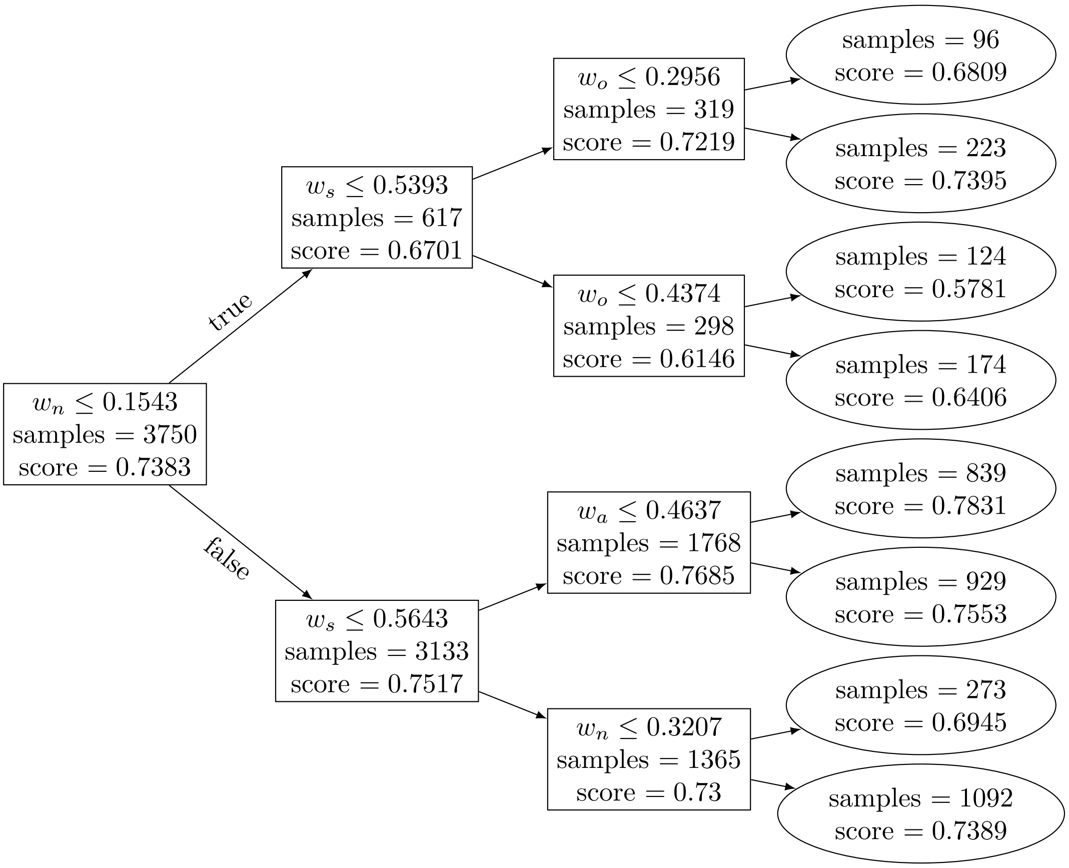


Figure 6.7. Decision tree based on the mean scores over all merge tasks.

This tree suggests that the highest score 0.7831 can be reached by choosing $w_n > 0.1543$, $w_s \leq 0.5643$ and $w_a \leq 0.4637$. Surprisingly not every weight participates in the decision for a high score. Nevertheless, the decisions are in accordance with the observations, when increasing the mean score, like in Figure 6.2.

6.2 Selecting Matchings

The evaluation of different probability values is based on 10 random weighting configurations. For every probability value, the ten configurations are evaluated with the merge tasks from Chapter 5. The resulting scores are presented in Figure 6.8 and Figure 6.9. The individual points in the plots represent the mean score over all merge tasks.

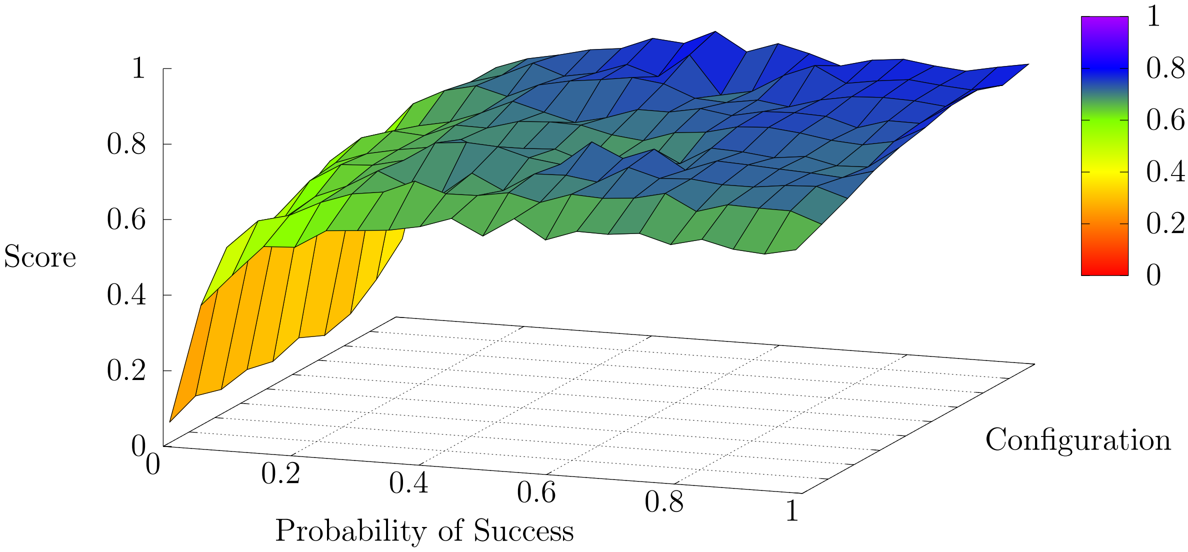


Figure 6.8. Score against probability of success and configurations.

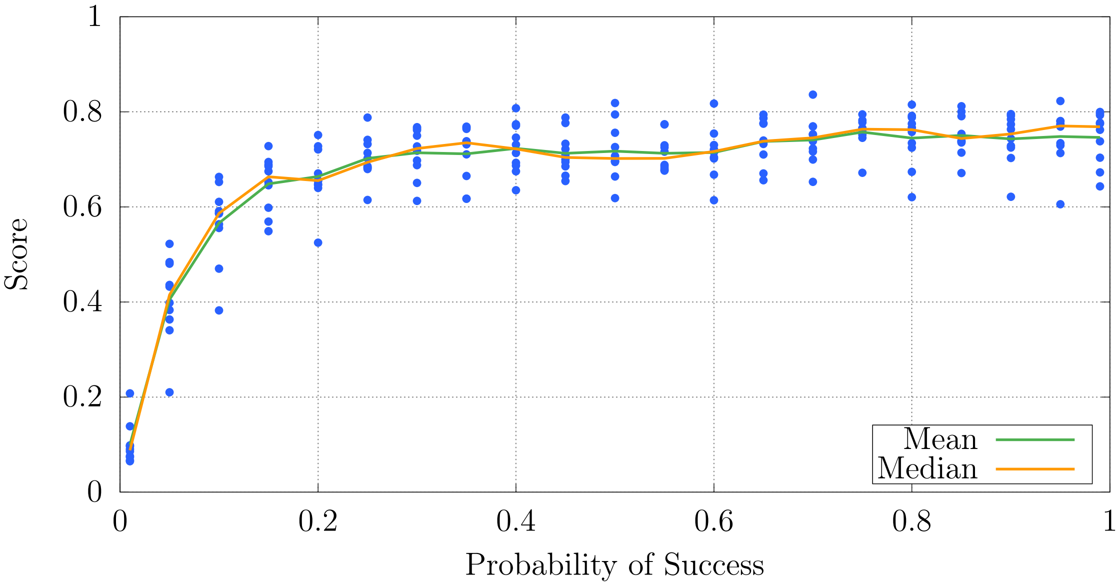


Figure 6.9. Averaged score against probability of success and configurations.

As both figures show, increasing the probability values above about 0.4 has no influence on the scores. However, when using lower values, the score rapidly decreases. The probability of success controls, how many costly matchings are fixed in one iteration of the Metropolis algorithm. The smaller the probability, the more times it is likely that non-ideal matchings are fixed. This leads to wrongly fixing edges in the bipartite graph that prevent other correct edges to be set. Nevertheless, the figures also show that using 0.7 in JDime is a good choice and leads to the best[2] possible scores.

6.3 Seed Influence

The seed should neither impact the performance nor the outcome of the matchings regardless of the particular weighting configuration. Each configuration has been evaluated with 10 different seed values and the range (difference between minimum and maximum) of the resulting scores is displayed in Figure 6.10.

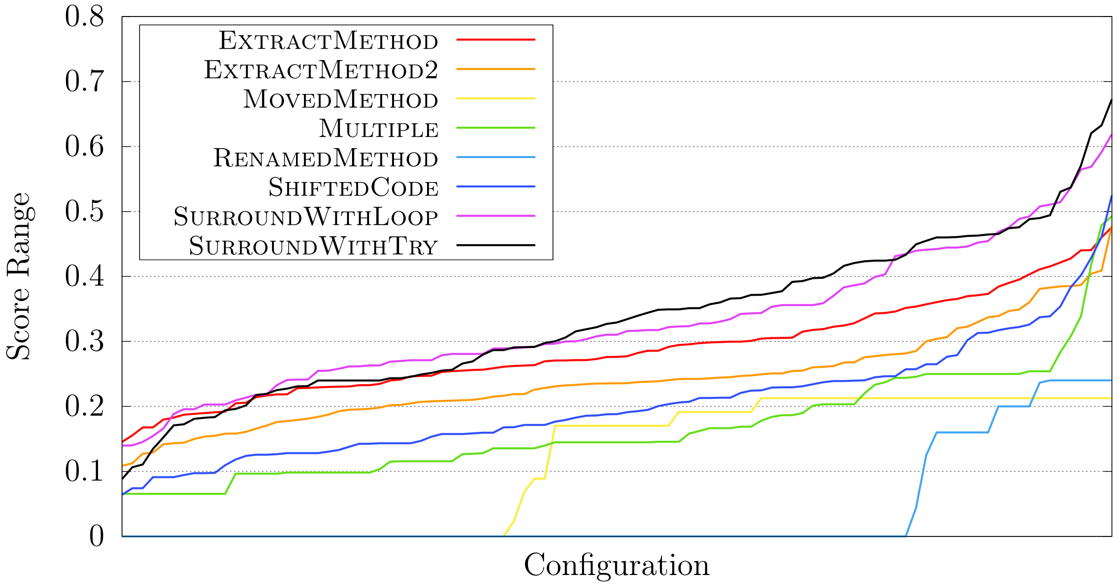


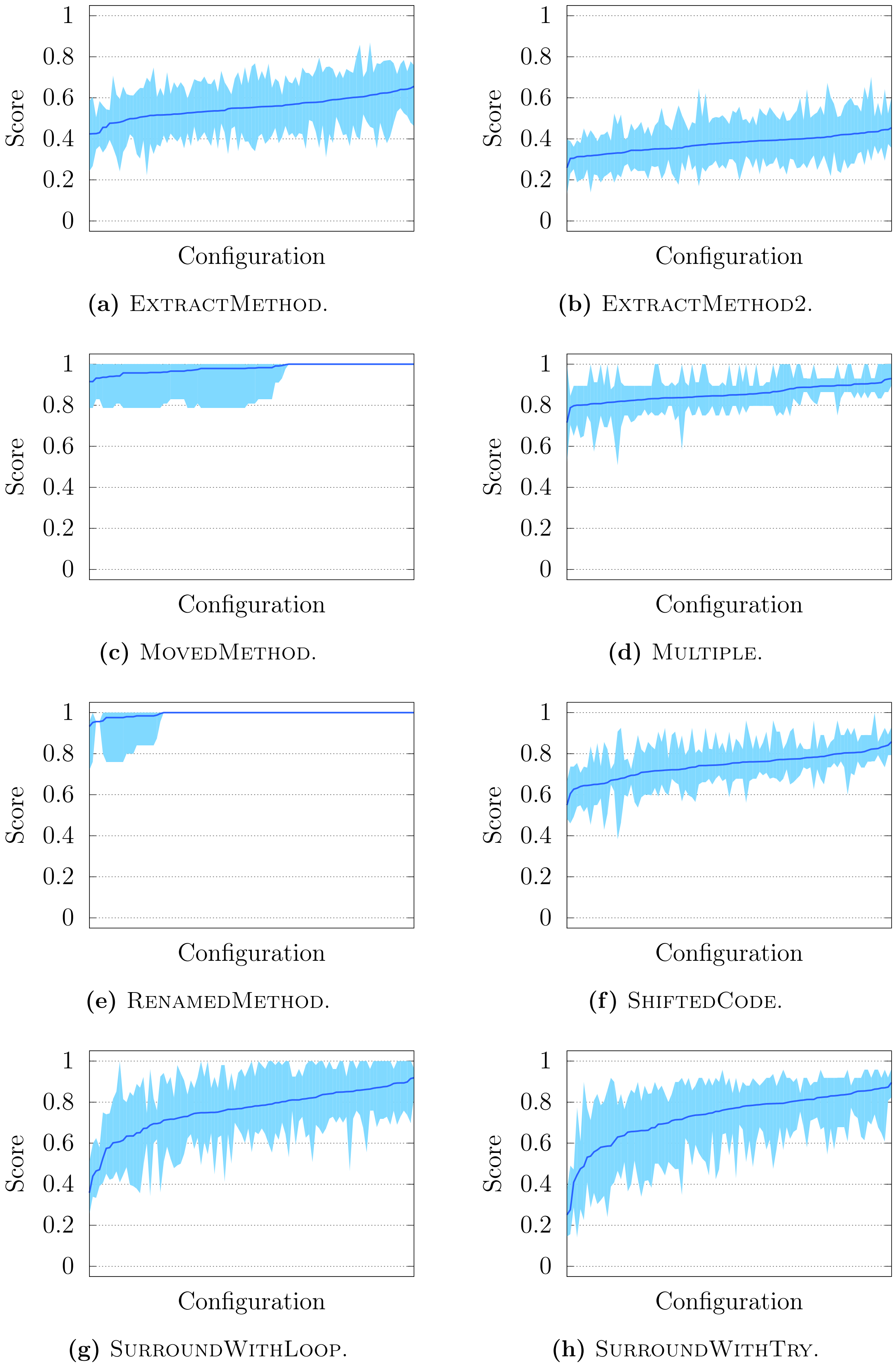
Figure 6.10. Range of the scores when evaluated with different seed values.

If the seed had no or little influence on the computation, then for each merge task a range of zero or at least close to zero has to be expected. As Figure 6.10 pinpoints, this is by no means the case. The only two merge tasks that comply the expectation to some extent are RenamedMethod and MovedMethod. Those two scenarios are the only ones that are not influenced by the seed — at least for some configurations — and therefore have a range of zero.

The following figures show the score of each configuration (light blue area) depending on the seed and the averaged score (blue line) over all seed values.

The staircase-like behavior of some plots can be explained with relatively small ASTs. RenamedMethod for example only has few nodes in its abstract syntax tree, which means the algorithm can only compute few different scores. Consequently, the discrete space of all the possible differences between minimum and maximum is small, too. Of course, the same applies to the other scenarios, but due to their larger ASTs, there are

[2] at least with regards to the probability value



(a) ExtractMethod.

(b) ExtractMethod2.

(c) MovedMethod.

(d) Multiple.

(e) RenamedMethod.

(f) ShiftedCode.

(g) SurroundWithLoop.

(h) SurroundWithTry.

Figure 6.11. Seed influence on every merge task.

many, many more possible range values. In combination with a small dataset, the plots do not have a staircase-like appearance.

Just like before, RenamedMethod and MovedMethod show, that the seed has no influence on the score, if all nodes are matched like the reference matching specifies. In all the other cases, however, the seed can manipulate the score values greatly, both in a positive and in a negative way. Interestingly, even when the average score over all seed values for one weighting configuration is about 0.6, a special seed allows the matching to be perfect (SurroundWithLoop) with a score of 1.0.

The reason for this rather contradictory result is still not clear and definitely needs more investigation. Causal could be the algorithm's sensitivity to changed parameters or a simple bug in its implementation. Either way, given the small sample size, future studies are suggested in order to validate the results on a larger dataset.

6.4 Syntactic Categories

Without using categories, the size of the resulting bipartite graph only depends on the size of both ASTs. When including the categories, another factor plays a great role: the structure of the ASTs. Matching a simple class with hundreds of variable declarations will not result in a reduced graph size because the syntactic category of those nodes is mostly the same. In "standard" classes, however, there are many different types of statements, e.g., method declarations, if statements, variable access, etc. This allows the filter to eliminate a bunch of nodes which aren't worth considering for matching.

Table 6.1 lists both the size of the bipartite graph before and after the introduction of the categories. All merge tasks have a greatly reduced graph size, but the biggest difference can be seen — as expected — in larger classes with many different categories.

Table 6.1. Size of the bipartite graph with and without syntactic categories.

		Valid Edges		
Merge Task	# Nodes	without	with	Change
		Categories		
ExtractMethod	218	11879	1530	-87.1 %
ExtractMethod2	308	23571	2710	-88.5 %
MovedMethod	88	1935	371	-80.8 %
Multiple	112	3135	583	-81.4 %
RenamedMethod	48	575	167	-71.0 %
ShiftedCode	116	3338	592	-82.3 %
SurroundWithLoop	125	3795	678	-82.1 %
SurroundWithTry	113	3119	660	-78.8 %

Weighting Configurations Every weighting configuration from Section 6.1 is evaluated again, but this time the particular ASTs are filtered such that the bipartite graph only contains edges connecting nodes with the same syntactic category or with the no-match nodes. The scores of the merge tasks are displayed in Figure 6.12 and Figure 6.13.

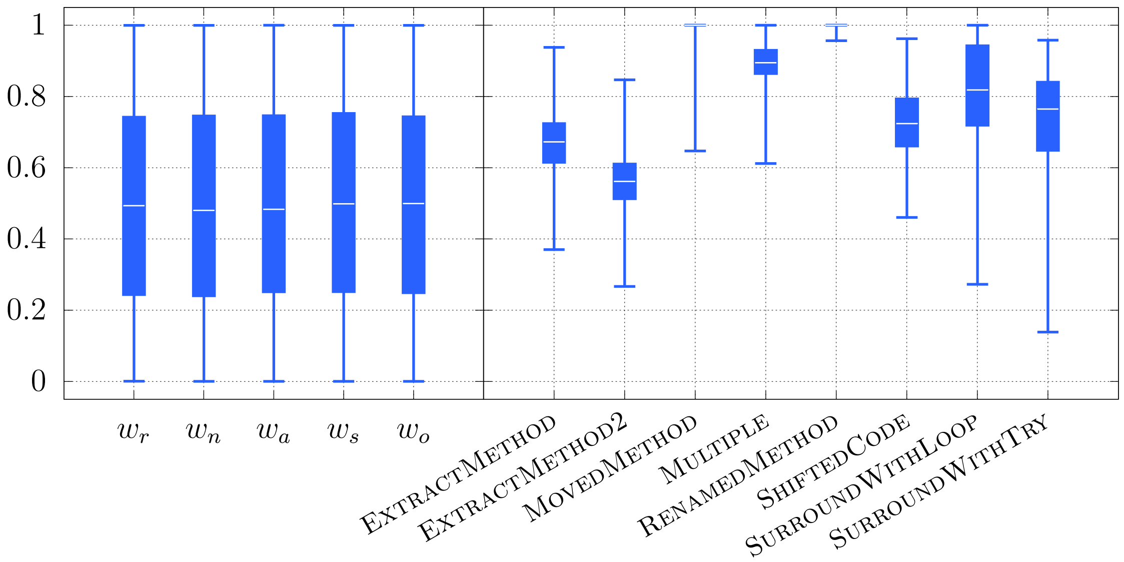


Figure 6.12. Distribution of weights and scores using syntactic categories.

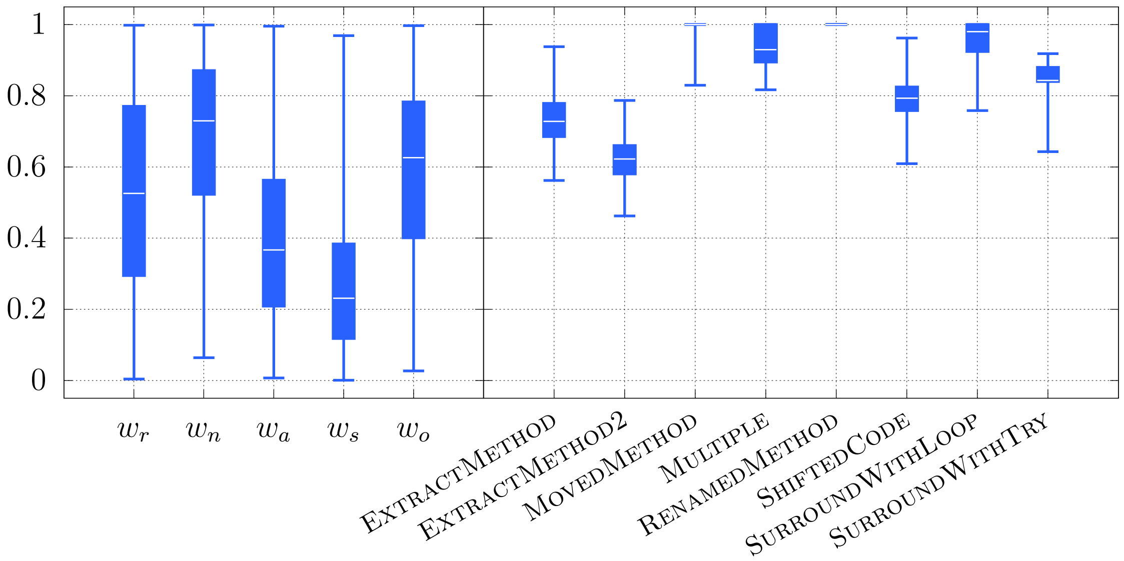


Figure 6.13. Distribution of weights and scores using syntactic categories (top 15 %).

Compared to the previous results without syntactic categories (Figure 6.1) almost every merge task produces slightly better results. Especially the scenarios with many AST nodes (ExtractMethod and ExtractMethod2) reach significant higher scores. The RenamedMethod task is matched correctly in 96.8 % and MovedMethod in 97.4 % of the cases.

Table 6.2 outlines the relative change in the score values. One can see very clearly, that tasks with many different syntactic categories (e.g. ExtractMethod2) benefit a lot from the filtering. Those with only a few categories (e.g. ShiftedCode) gain less to not much. It should be noted, that the reason for the poor result of SurroundWithLoop might be a stochastical anomaly.

Table 6.2. Mean score of the merge tasks with and without categories.

Merge Task	Mean Score without	 with	Change
	Categories		
ExtractMethod	0.498	0.670	+34.4 %
ExtractMethod2	0.365	0.559	+53.2 %
MovedMethod	0.945	0.996	+5.4 %
Multiple	0.834	0.903	+8.3 %
RenamedMethod	0.989	0.999	+1.0 %
ShiftedCode	0.713	0.717	+0.6 %
SurroundWithLoop	0.868	0.808	−6.9 %
SurroundWithTry	0.663	0.722	+8.6 %

Nevertheless, as Figure 6.12 shows, the big picture does not change: there are still many merge tasks that produce scores in almost the full range (e.g. SurroundWithTry) and bigger tasks still perform worse than smaller ones (with unacceptable results).

Seed Influence The same configurations from Section 6.3 are tested with the new implementation. As illustrated in Figure 6.14, no significant improvement was identified.

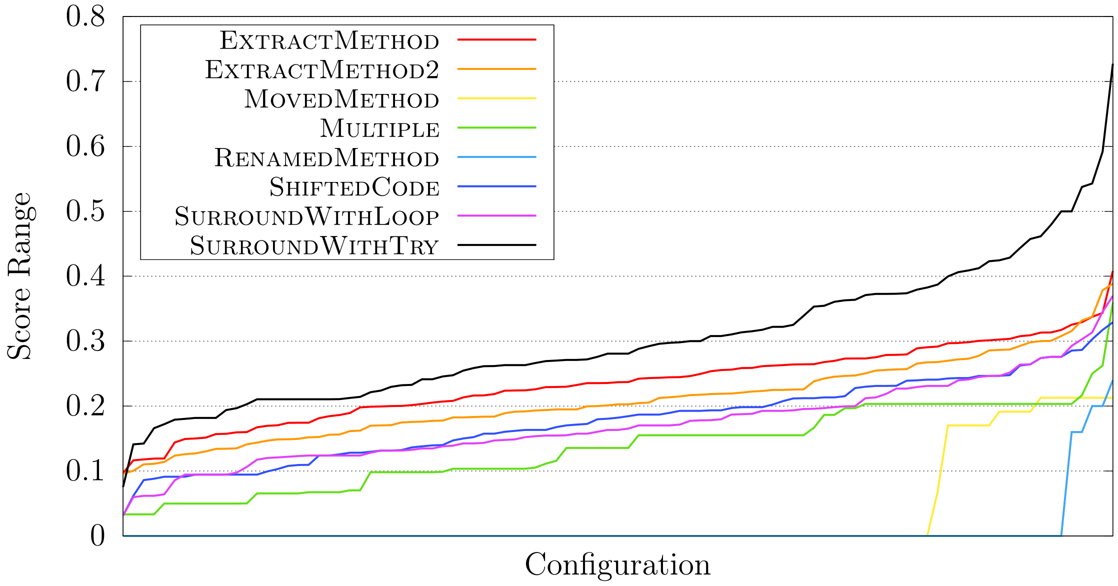


Figure 6.14. Influence of different seed values on the score using syntactic categories.

Runtime Improvement The most striking result to emerge from the data is the reduced runtime. The results were gathered on a machine with an Intel Core i7 Skylake clocked at 2.6 GHz with 16 GiB of RAM available. Table 6.3 shows the runtime of the different configurations for each merge task. As the application is running multithreaded inside the Java Virtual Machine, the runtime values were averaged over 10 runs, each with a random weighting configuration.

Table 6.3. Runtime of the merge tasks with and without categories.

	Runtime		
Merge Task	without	with	Change
	Categories		
ExtractMethod	144.26 s	48.26 s	−66.5 %
ExtractMethod2	402.61 s	114.36 s	−71.6 %
MovedMethod	13.61 s	6.43 s	−52.7 %
Multiple	23.57 s	11.03 s	−53.2 %
RenamedMethod	2.77 s	1.58 s	−42.9 %
ShiftedCode	20.80 s	9.49 s	−54.4 %
SurroundWithLoop	26.54 s	11.86 s	−55.3 %
SurroundWithTry	19.72 s	9.12 s	−53.7 %

As expected, the experiments show that filtering nodes lead to reduced runtime. In fact, the new implementation allows JDime to process the merge tasks about twice as fast than before. Especially those tasks with more nodes in the ASTs take drastically less time.

CHAPTER 7

Conclusion and Future Work

This thesis gave an overview over common merging techniques in Version Control Systems and examined the structured approach in particular. The de-facto standard in merging today is a line-based, textual method, which is applicable to all kinds of plain text files, because of its generality and speed. But on the other hand, by not using any information about the structure of a document at all, such tools tend to produce a large number of conflicts, which are in some cases hard to resolve manually. Structured algorithms provide much more power to match elements and detect conflicts in order to assist users while merging files, but they are slower and restricted to specific programming languages. One example is the AST-based implementation in JDime, which performs a syntactic merge on Java files. The fundamental part of this approach is the Flexible Tree Matching algorithm with its cost model containing many parameters that control the effectiveness and efficiency of the strategy.

This thesis investigated the influence of weightings on those costs using different merge tasks and could show, that there is a relation between better merging results and the individual weights. Nevertheless, bigger merge tasks with more nodes in the AST perform always worse than their smaller counterparts and do not even get in the range of "good" results. The research could possibly support the decision to use the Flexible Tree Matching approach in JDime *after* a textual merge has resulted in conflicts.

The experiments could confirm that the parameter controlling the proposal of matchings, which is hard-coded as $p = 0.7$ is indeed a good choice, considering that other values did not improve the quality of the matchings. This research has also raised many questions in need of further investigation regarding the influence of a seed on the correctness of the algorithm. Future studies should target this topic in particular.

The introduction of syntactic categories to the AST's nodes could improve the correctness of the particular merge scenarios, although the big picture did not change. However, the runtime of each scenario could be drastically reduced: a merge task runs up to four times and at least twice as fast as without categories, thanks to the filtering for suitable nodes. These findings suggest the opportunities for future research to optimize filtering even further. On a wider level, `List` nodes can be assigned different categories to differentiate a modifier list from a parameter list. This could reduce the runtime of a matching even more.

Bibliography

[AL] Sven Apel and Olaf Leßenich. *JDime: Structured Merge with Auto-Tuning.* URL: http://www.infosun.fim.uni-passau.de/se/JDime/ (visited on 10/03/2017).

[Atl] Atlassian-Tutorials. *What is version control.* URL: https://www.atlassian.com/git/tutorials/what-is-version-control (visited on 10/03/2017).

[CG95] Siddhartha Chib and Edward Greenberg. "Understanding the Metropolis-Hastings Algorithm". In: *The American Statistician* 49.4 (1995), pp. 327–335.

[dev17] scikit-learn developers. *Decision Trees.* 2017. URL: http://scikit-learn.org/stable/modules/tree.html (visited on 10/03/2017).

[Gün14] Tobias Günther. *Understanding the Concept of Branches in Git.* 2014. URL: https://www.git-tower.com/blog/understanding-branches-in-git/ (visited on 10/03/2017).

[Haw] Douglas Hawks. *What Is a Decision Tree? — Examples, Advantages & Role in Management.* URL: http://study.com/academy/lesson/what-is-a-decision-tree-examples-advantages-role-in-management.html (visited on 10/03/2017).

[Inc08] Free Software Foundation Inc. *Comparing and Merging Files.* Nov. 2008. URL: https://www.gnu.org/software/diffutils/manual/html_node/index.html (visited on 10/03/2017).

[Jas11] JastAdd-Team. *ExtendJ: The JastAdd Extensible Java Compiler.* 2011. URL: http://jastadd.org/web/extendj/ (visited on 10/03/2017).

[Kum+11] Ranjitha Kumar et al. "Flexible tree matching". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence-Volume Volume Three.* AAAI Press. 2011, pp. 2674–2679.

[LAL15] Olaf Leßenich, Sven Apel, and Christian Lengauer. "Balancing precision and performance in structured merge". In: *Automated Software Engineering* 22.3 (2015), pp. 367–397.

[Leß12] Olaf Leßenich. “Adjustable Syntactic Merge of Java Programs”. MA thesis. University of Passau, Feb. 2012.

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Say] Saed Sayad. *Decision Tree — Classification*. URL: `http://www.saedsayad.com/decision_tree.htm` (visited on 10/03/2017).

[Sei16] Georg Seibt. “Applying Flexible Tree Matching to Abstract Syntax Trees”. MA thesis. University of Passau, Sept. 2016.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Martin Bauer
Passau, den 06. Oktober 2017