2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Automatic Clustering of Code Changes

Patrick Kreutzer[1], Georg Dotzler[1], Matthias Ring[2],
Bjoern M. Eskofier[2], Michael Philippsen[1]
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
Programming Systems Group[1] and Digital Sports Group[2]
{patrick.kreutzer, georg.dotzler, ..., michael.philippsen}@fau.de

ABSTRACT

Several research tools and projects require groups of similar code changes as input. Examples are recommendation and bug finding tools that can provide valuable information to developers based on such data. With the help of similar code changes they can simplify the application of bug fixes and code changes to multiple locations in a project. But despite their benefit, the practical value of existing tools is limited, as users need to manually specify the input data, i.e., the groups of similar code changes.

To overcome this drawback, this paper presents and evaluates two syntactical similarity metrics, one of them is specifically designed to run fast, in combination with two carefully selected and self-tuning clustering algorithms to automatically detect groups of similar code changes.

We evaluate the combinations of metrics and clustering algorithms by applying them to several open source projects and also publish the detected groups of similar code changes online as a reference dataset. The automatically detected groups of similar code changes work well when used as input for LASE, a recommendation system for code changes.

CCS Concepts

•Information systems → Clustering; •Software and its engineering → *Software libraries and repositories;*

Keywords

Clustering; Code Changes; Software Repositories

1. INTRODUCTION

Similar code changes, like the two bug fixes shown in Fig. 1, are typical ingredients of software development. Similar changes occur, for example, if a certain bug has to be fixed in many spots in the project's code, or if a change of a third-party library causes modifications at all call sites. Usually, the contexts of similar changes differ slightly with respect to variable names (*foo* vs. *bar*), to instruction ordering, to interwoven unrelated statements (*someCall()*), etc.

MSR'16, May 14-15 2016, Austin, TX, USA

ISBN 978-1-4503-4186-8/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2901739.2901749

```
- foo = Lib.getObject();
- foo.someMethod(13, 3);
+ foo = Lib.getObjectOrNull();
+ if (foo != null) foo.someMethod(13, 3);
```

(a) Bug fix 1.

```
- bar = Lib.getObject();
+ bar = Lib.getObjectOrNull();
  someCall();
- bar.someMethod(12, 11);
+ if (bar != null) bar.someMethod(12, 11);
```

(b) Bug fix 2.

Figure 1: Similar changes that add a *null*-check.

Since carrying out repetitive code changes is both tedious and error-prone, some research projects exploit similar code changes to make developers more productive and to avoid unnecessary errors. For example, LASE [42] requires a set of syntactically similar code changes as input and generalizes them by abstracting away minor differences among the set. It then uses the generalizations to find other spots in the code that match. For such a spot, LASE recommends a code change that is adapted to the spot's context. Cookbook [30] uses the generalizations to suggest code completions while the developer is typing. RASE [41] also starts from a set of similar changes. After the generalization, it refactors the code without altering its semantics to replace all code changes with a single unifying clone. Critics [57] addresses the review process. The developer provides a generalization of a single code change as input. Critics then finds matching spots in the code where a similar change may have been forgotten or where an inconsistent modification may have occurred.

All these research projects require as input a set of similar code changes (or a generalization thereof). The decision whether code changes are similar relies on the subjective judgment of a user. For example, experienced developers might consider the changes in Fig. 1 as similar. In both cases, *getObject()* is replaced with *getObjectOrNull()* and the required *null*-check is added to the code. But consider the code change in Fig. 2. Here, instead of a *null*-check, a developer added a guard to a debug output. For an untrained eye the syntax of this code change looks highly similar to

```
- foo = Lib.getObject();
- System.out.println(foo);
+ foo = Lib.getObjectOrNull();
+ if (DEBUG) System.out.println(foo);
```

Figure 2: Code change to hide debug output.

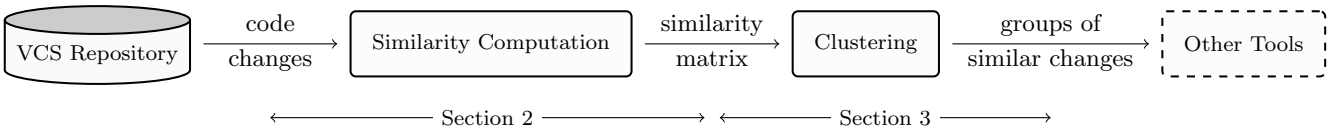


Figure 3: Workflow of $\mathcal{C}3$.

the bug fixes in Fig. 1, but not for experienced developers. Thus, in addition to compare any two code changes, a user of the above tools also has to manually perform a clustering to form the groups of similar changes and to decide when a change is *not similar enough* to be included in a group.

In real world projects all of this is too much manual work. Thus, the impact of tools that try to help developers with similar changes can be improved in practice. Our proposed approach $\mathcal{C}3$ (*Clustering of Code Changes*) overcomes this drawback as it identifies groups of similar changes in a fully automatic way. It puts the bug fixes in Fig. 1 into a group that does not contain the code change from Fig. 2.

Fig. 3 illustrates the workflow. First, $\mathcal{C}3$ mines the repository of a version control system (VCS) to extract code changes. Then it compares them pairwisely to compute a similarity matrix. We evaluate two different syntactical metrics that both rely on a fine-grained extraction of code changes. One metric is based on applying the Unix tool *diff* to methods, the novel and faster metric uses fine-grained edit scripts. For the metrics we show the trade-off between execution time and accuracy. This similarity computation is discussed in Sec. 2. A clustering follows in Sec. 3 that uses the similarity matrix to detect groups of similar changes. We present two algorithms that work on this matrix and we show how to automatically optimize their configuration parameters to different repositories. The detected groups can serve as input for recommendation and bug fixing tools. Sec. 4 evaluates the 4 combinations of metrics and clustering algorithms. It shows that the recommendation system LASE can work with the resulting groups of code changes as input. Before we conclude, Sec. 5 discusses related work.

2. SIMILARITY OF CODE CHANGES

This section is organized as follows: First, we define what $\mathcal{C}3$ considers to be *code changes* and how it detects them. We will argue that a code change cannot simply be the file that has been modified between two versions in the repository. Second, we show two options for $\mathcal{C}3$ to represent the extracted code changes. The common idea is that a code change is represented as a sequence of edit operations that turns an original code fragment into its modified version. Representations differ with respect to both their edit operations and the items they affect. Besides a traditional *diff*-based representation, we present a fine-grained representation that is based on the abstract syntax trees (ASTs) of the code. The final step takes these representations as input and compares them in a pairwise fashion. $\mathcal{C}3$ avoids to compute all pairwise similarity values to reduce the runtime.

In total, $\mathcal{C}3$ uses the above three steps to extract a set C of code changes from a software repository to be able to apply a similarity metric s, i.e., a symmetric function $C \times C \mapsto [0, 1]$ that assigns a *similarity value* $s(c_j, c_k) = s(c_k, c_j)$ to a pair of code changes $c_j, c_k \in C$. The more similar two changes c_j and c_k are, the higher is the value of $s(c_j, c_k)$. The result is a similarity matrix of size $|C| \times |C|$.

2.1 Granularity of Code Changes

When Unix *diff* is applied to the old and the new version of a file, it splits the modifications into blocks of code lines. The usefulness of these blocks varies. Sometimes, like in the examples in Figs. 1 and 2, the developer's modifications are easy to recognize and understand. If, however, the modifications do not stay local within a small area of the file, the output of *diff* can become hard to understand because they end up in a number of small blocks. The reason is that *diff* uses a certain number of untouched lines as threshold for splitting larger code changes into changes of finer granularity, i.e., the blocks of changed lines. The readability of the *diff* output is further deteriorated if developers move larger code fragments around. As *diff* cannot detect move operations, this creates many unintended inserted and deleted lines. This gets even worse if code is both moved and modified. Moreover, as blocks of changed code lines can span method boundaries, the *diff* output is sometimes also hard to map back to what was modified.

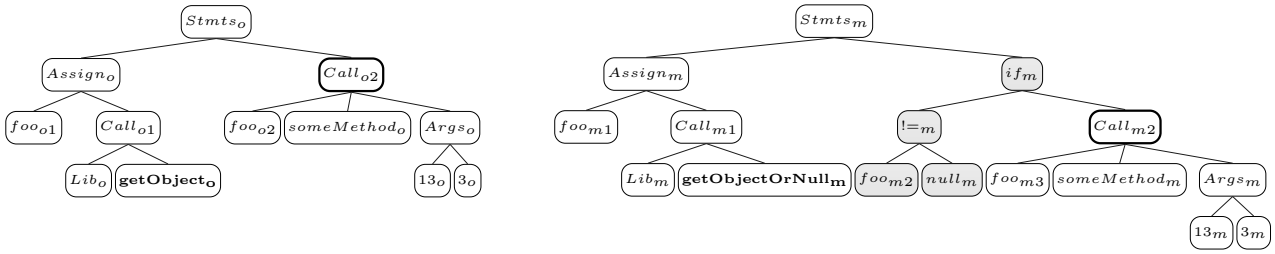
The challenge hence is to bundle edit operations in a more meaningful way: Changes should be aligned to the syntactical structure of the code and should be small enough not to be affected by moved code fragments in other parts of the file. These bundles cannot be too small because the size of the similarity matrix grows quadratically with their number. They cannot be too large, e.g., by bundling all the (possibly unrelated) modifications of a committed file into one big change, as this makes it unlikely to find similar changes in other commits. Also, for other tools like LASE that work with change sets it is better (or even required) to identify code changes with finer granularity.

As a compromise, we define a *code change* to be a pair (M_o, M_m) of changed methods, where M_o is a method from the original file and M_m is the corresponding method from the modified file. Thus, a modified and committed file may contain multiple code changes. If *diff* is used on the old version and the new version of a method, the range of code lines is much smaller and hence the output is often more reasonable. Because of the smaller scope, the user also often notices that some deletes and inserts are caused by a move.

Note that this definition excludes changes outside of methods (e.g. changes to member variables). This restriction has, however, little impact on the results, as these changes are less likely to be repetitive than changes within methods. Additionally, tools like LASE require a pair of methods as input, i.e., they can only deal with changes inside methods.

It is, however, easier said than done to split a modified file into its set of modified methods. For example, consider a method that is renamed, has been given an extra parameter, and has also been moved to the other end of the file. How much modification and code motion can there be so that it still can be detected as the same method, despite some modifications? Tools like *diff* that just work on lines of text cannot solve this problem.

To solve this problem, $\mathcal{C}3$ uses (a modified version of) the



(a) AST_o of the original code fragment *before* the change.

(b) AST_m of the modified code fragment *after* the change.

Figure 4: ASTs of the example code fragment in Fig. 1(a), before and after the change.

ChangeDistiller [18] tree differencing algorithm (CD) that works on the ASTs of the original and the modified file, see https://github.com/FAU-Inf2/treedifferencing. Such a tree differencing algorithm uses heuristics to compute a mapping of AST nodes to find those nodes in the two ASTs that best belong together. The AST from the method that got the extra parameter and has been moved within the file will still have many AST nodes that match their corresponding nodes in the original AST, especially as CD understands renaming of identifiers. Except for the 4 gray ones, for each of the nodes in AST_o of Fig. 4, CD finds the matching node in the other AST_m, e.g., $(Call_{o2}, Call_{m2})$.

As soon as the tree differencing algorithm returns the mapping of the AST nodes of the original and the modified file, it is possible to detect tuples of original and modified methods. There are four cases that $\mathcal{C}3$ uses to identify these tuples. (a) $\mathcal{C}3$ builds a tuple of an original method M_o and a modified method M_m if CD has matched their code-block nodes. This withstands changes to the signature of the methods. CD matches the code-block nodes if their subtrees have enough matching nodes. (b) For all remaining methods in both ASTs, $\mathcal{C}3$ examines the method signature nodes. If the CD heuristics considered these to match, $\mathcal{C}3$ builds a tuple from them. (c) To find partners for all methods that are still not paired into tuples, $\mathcal{C}3$ examines the method names. An original method is bundled with a modified method with the exact same unique name. (d) If there are still methods left, $\mathcal{C}3$ examines their positions, i.e., their order of appearance in a depth-first traversal of the ASTs. Thus, if M_o is the i^{th} method of AST_o and M_m is the i^{th} method of AST_m, $\mathcal{C}3$ yields a tuple (M_o, M_m).

In general, the mapping may contain pairs of methods that were not changed between the original file and the modified file. $\mathcal{C}3$ thus only generates a code change representation (Sec. 2.2) for a pair of methods (M_o, M_m), if the results of CD indicate a change between M_o and M_m. Since inserted and deleted methods cannot be part of a tuple, they are discarded by $\mathcal{C}3$. This keeps the similarity matrix small and has only little impact on the clustering results as deletions and insertions of full methods are rarely repetitive changes.

$\mathcal{C}3$'s modified CD has a worst case time complexity of $\mathcal{O}_{CD}(l^2 \cdot \log l^2 + i \cdot l)$ for comparing two ASTs with at most l leaf nodes and i inner nodes. Although this is expensive, it avoids the negative effects of wrongly detected code changes in later steps and in the clustering. Note that later on all modified methods can be processed concurrently.

2.2 Representation of Code Changes

A feature vector is one possible way to represent a code change, for example see Fluri et al. [17]. Instead, $\mathcal{C}3$ is based on the common idea to represent a code change as a list of edit operations (also called edit script) that turn an original code fragment into its modified version. Such representations differ with respect to both the set of available edit operations and the items that the edit operations affect. For every code change, $\mathcal{C}3$ computes either a traditional *diff*-based or a novel AST-based representation. $\mathcal{C}3$ then turns a code change into an *encoded script*, i.e., a string that holds all the necessary edit operations. These strings are later on compared in a pairwise way.

Line-based representation with diff on methods.

Diff uses the longest common subsequence (LCS) [8] to detect inserted and deleted lines. This takes $\mathcal{O}(d^2)$ time, where d denotes the length of the longer input. Examples of a *diff*-based representation can be found in Figs. 1 and 2.

According to Falleri et al. [15], this format has conceptual drawbacks. First, programmers do not think and work on the granularity of lines but instead they modify code structures which can either be part of a line or can span multiple lines. Second, programmers use more edit operations than just insert and delete. For example, they also move, rename, and reorder the code in the files.

Despite these conceptual problems, our evaluation in Sec. 4 shows that the clustering derived from method sized *diff* representations still finds useful groups. However, the performance suffers since the string representation, i.e., the concatenation of all the lines that form the output of *diff*, is too verbose. If a programmer changes only a single character in a line, this adds two lines to the code change representation in the edit script. Long edit scripts take a long time in the pairwise comparison and make it time-consuming to compute the similarity matrix.

AST-based representation.

To produce more concise and much shorter edit scripts for a faster pairwise comparison, $\mathcal{C}3$ uses the following key ideas: (a) It starts from the edit scripts that the tree differencing algorithm generates from its matching of AST nodes. (b) $\mathcal{C}3$ then raises the level of abstraction of the purely graph-oriented edit operations into the domain of the programming language. The result are fewer but more meaningful edit operations. Finally, (c) these edit scripts are encoded densely for a faster pairwise comparison.

Tree differencing. $\mathcal{C}3$'s tree differencing algorithm employs the edit operations *insert*, *delete*, *update*, *align* and *move* to specify what changes are necessary to transform AST_o of the original method into the modified AST_m.

For each of the four gray nodes in the modified AST_m of Fig. 4 that do not have a partner in the matching, there is an *insert*. For matching nodes with different positions

```
insert(if_m, Stmts_m, 1)
insert(!=_m, if_m, 0)
insert(foo_m2, !=_m, 0)
insert(null_m, =_m, 1)
update(getObject_o, getObjectOrNull)
move(Call_o2, if_m, 1)
```

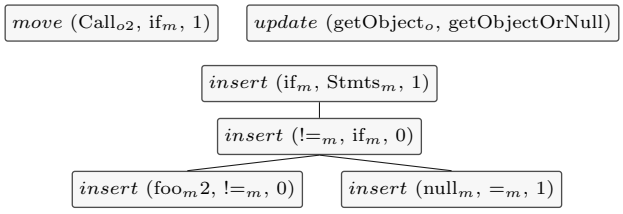
Figure 5: Edit script for the ASTs in Fig. 4.

there is a *move*. For a changed value (e.g. (*getObject*$_o$, *getObjectOrNull*$_m$)) there is an *update*. See Fig. 5 for all 6 edit operations. Compared to the line insertions and deletions in the *diff*-representation, this better describes the steps that the programmer took to change the code.

Raising the level of abstraction. To further shorten the script, $\mathcal{C}3$ bundles small edit operations into more compact programming-domain operations. In Fig. 5 it takes four basic *insert* operations to add the subtree rooted at *if*$_m$. Instead $\mathcal{C}3$ first builds ordered script trees from the basic operations. Edges in the script trees indicate that operations are related. In Fig. 6(a), *if*$_m$ is related to *!=*$_m$ since the former is a parent of the latter in the AST. Similarly, the insertions of *!=*$_m$, *foo*$_{m2}$, and *null*$_m$ are linked. In Fig. 6(b), $\mathcal{C}3$ bundles the complete tree into one higher-level *InsertStatement* operation. Thus, the edit script contains not only the information that a tree node is added to the code, but also that it is a statement node.

Including *InsertStatement*, $\mathcal{C}3$ uses 20 higher-level operations. Similar to Fluri and Gall [16], there are five operations for statements (*Align*, *Delete*, *Insert*, *Move*, *Update*), two for the final modifier (*Insert*, *Delete*), two for the accessibility (*Increase*, *Decrease*), one for methods (*Rename*), one for the return type (*Update*), and five for parameters (*Align*, *Delete*, *Insert*, *TypeChange*, *Rename*). Additionally, $\mathcal{C}3$ introduces four new operations for variable declarations (*Insert*, *Delete*, *Rename*, *TypeChange*).

Tight encoding. Up to here we presented the edit scripts in a human-readable form. For faster pairwise comparisons, it is crucial to encode them with only a few bytes. Besides the short size, it is also important to find an encoding that avoids similarities between unrelated AST nodes. In any naive character-based encoding, two different and overly long identifier names could easily hide that two edit scripts



(a) Basic edit operations as ordered script trees.

```
InsertStatement (if_m, !=_m, foo_m2, null_m, Stmts_m, 1)
update (getObject_o, getObjectOrNull)
MoveStatement (Call_o2, if_m, 1)
```

(b) Higher-level edit operations.

```
0x06 0x23 0x24 0x25 0xCAFE 0x26 0x27
0x03 0xABCD
0x04 0x29 0xAB 0xAA
```

(c) Encoded edit script.

Figure 6: Raising the level of abstraction for bug fix 1.

are almost identical, except for those two names. On the other hand, the type of the node affected by the edit operation has to be distinctive when assessing the similarity of two scripts.

$\mathcal{C}3$ therefore encodes all the basic edit operations in a single character, e.g. `0x01` represents an *insert* operation. A second character represents the node type (e.g. `0x29` for the moved $Call_{o2}$). If $\mathcal{C}3$ encounters an operation-tree similar to the *insert*-tree in Fig. 6(a), it encodes the operation type only once and appends the types. This again better represents what the programmer was doing when changing the code. The last encoded type of an *insert* operation is the type of the parent node (e.g. `0x27` in *InsertStatement*).

To make the length of the identifier name irrelevant, they are encoded as (unique) characters in addition to the node type, yielding different values for names like *foo* (`0xCAFE`), *ba* (`0x1234`) and *b* (`0xFFFF`). This avoids the 50%-similarity in a character-wise name comparison of *b* and *ba* and also keeps expressions like $a+a$, $a+b$, and $b+b$ distinguishable. Analyzing the results we found that a 32-bit hash is sufficient to distinguish identifiers even in large repositories.

Another important aspect of an encoding is to encode positions and distances of the *move* and *align* operation in a relative way. $\mathcal{C}3$ uses tuples like (x, y) where x corresponds to the index of a node in the children list of its parent. For example, $Stmts_o$ has the children list $(Assign_o, Call_o)$, i.e., $Assign_o$ is at index 0 and $Call_o$ is at index 1. The value y denotes the tree level and thus the vertical distance from the root node. In the example, the old position of $Call_{o2}$ is $(1, 1)$ and the new one is $(1, 2)$. To get the relative distance, $\mathcal{C}3$ subtracts the node's new position $(1, 2)$ from its old one $(1, 1)$, resulting in the relative move $(0, -1)$. For the encoding, $\mathcal{C}3$ assigns the value `0xAB` to position 0, leading to an encoded relative movement of (`0xAB`, `0xAA`). With relative positions, two *move* operations in two edit scripts are encoded in the same way, even if they affect slightly different spots of the ASTs. That makes the edit scripts more similar than with absolute positions.

Fig. 6(c) shows the encoded character arrays for the running example. Note that $\mathcal{C}3$ omits some less relevant information, like the position of the *insert* operation and the value of the *update* operation. This also shortens the scripts.

2.3 Pairwise Similarity Values

Let r_i denote the string representation of a code change c_i (i.e., a method pair with changes), and let $|r_i|$ denote its length. To compute the pairwise similarity value $s(c_j, c_k)$ of two code changes c_j and c_k, $\mathcal{C}3$ computes the length $LCS(r_j, r_k)$ of the *Longest Common Subsequence* [8] of their representations r_j and r_k. To normalize to the interval $[0, 1]$, $LCS(r_j, r_k)$ is divided by the maximal length of the strings:

$$s(c_j, c_k) := \frac{LCS(r_j, r_k)}{max(|r_j|, |r_k|)} \leq \frac{min(|r_j|, |r_k|)}{max(|r_j|, |r_k|)} =: s_{max}(c_j, c_k).$$

The more similar the two string representations r_j and r_k are, the higher is the value of $LCS(r_j, r_k)$ and the higher the resulting similarity value $s(c_j, c_k)$ of the code changes. Note that there is an upper bound $s_{max}(c_j, c_k)$ for each pair (r_j, r_k), since $LCS(r_j, r_k)$ can never be larger than the length of the shorter of the two representations r_j and r_k.

$\mathcal{C}3$ uses Hirschberg's algorithm [27] for LCS that has a time complexity of $\mathcal{O}(|r_j| \cdot |r_k|)$. With a suitable fixed gap penalty, the Needleman-Wunsch algorithm [43] produces sim-

ilar results. The time complexity of a dynamic gap penalty is too high. Thus, as the choice is arbitrary, $\mathcal{C}3$ uses LCS.

Clustering-Aware Similarity Matrix. Comparing the representation of each code change of a repository with that of every other code change is too time consuming for large repositories, even with a dense script encoding. Fortunately, our evaluation shows that for the clustering it is sufficient to only consider the nearest neighbors $N(c_j)$ of each code change c_j. A code change c_k is said to be a nearest neighbor of c_j if there is no other code change c_i that is more similar to c_j than is c_k:

$$c_k \in N(c_j) \;\Leftrightarrow\; \forall c_o \in C \setminus \{c_j\} : s(c_j, c_o) \leq s(c_j, c_k)\,.$$

Thus, for each row in the similarity matrix that belongs to a code change c_j, it is only necessary to find the entry (or entries) where $s(c_j, c_k)$ is maximal. All other entries are 0.

This makes it quick to iteratively compute a row j of the similarity matrix, column by column. For each column k it suffices to check (in $\mathcal{O}(1)$ time) the upper bound $s_{max}(c_j, c_k)$. It is only necessary to actually compute the Longest Common Subsequence for the exact value of $s(c_j, c_k)$, if this upper bound is above the maximal similarity value of the row that the iterative computation has seen so far.

Note that a parallel reduction can be used instead of the iteration. Moreover, all rows can be processed concurrently as there are no dependencies.

3. CLUSTERING

Let $C = \{c_1, \ldots, c_n\}$ denote the set of code changes, and let S denote the corresponding similarity matrix computed with one of the techniques introduced in Sec. 2. When interpreted as an adjacency matrix, S induces an undirected, weighted similarity graph $G = (C, E)$. The nodes of G correspond to the code changes. Each edge $e_{jk} \in E$ between two nodes c_j and c_k carries the similarity value $s(c_j, c_k)$ as weight. Two nodes with a similarity of 0 are unconnected in G. Reflexive edges are omitted from the graph as their similarity value of 1 is irrelevant. See Fig. 7 for an example.

This section explains how $\mathcal{C}3$ processes such a similarity graph to split C into groups of similar code changes. A *clustering* is a (not necessarily complete) partitioning of the n elements of C into m disjoint subsets C_i, called *clusters* or *groups*, with at least two members each:

$$C \supseteq (C_1 \cup C_2 \cup \cdots \cup C_m)$$
$$\forall\, 1 \leq i, j \leq m,\, i \neq j :\; C_i \cap C_j = \varnothing$$
$$\forall\, 1 \leq i \leq m :\; |C_i| > 1\,.$$

Changes that are not assigned to a cluster are called *outliers*.

To be of any use, the detected clusters have to consist of code changes that are similar enough to one another. On the other hand, we prefer larger clusters with many members to cover as much variance as possible. The algorithms that we use to compute a clustering take these requirements into account and provide parameters to control the granularity of the clusters. We present heuristics to automatically choose these parameter values to balance the conflicting goals.

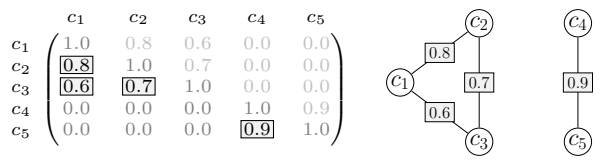


Figure 7: A similarity matrix and the graph it induces.

To compute a clustering, $\mathcal{C}3$ first splits the graph into its connected components in a pre-processing step (Section 3.1). A clustering algorithm is then applied to each connected component to assign its nodes to groups of similar changes (Sections 3.2 and 3.3). In preliminary research, we evaluated a selection of clustering algorithms [56] (some work on the similarity graph, others work on a d-dimensional embedding of it) to pick the ones that best suit the problem of clustering code changes.

The *Markov Cluster Algorithm* [51] is one of the graph-based algorithms. We found that it tends to produce larger groups than other algorithms. Even when considering only the code changes in the validation set used in Section 4, it combines several of the manually built clusters of similar changes into larger groups. This reduces their usability, as for example LASE cannot create generalized change patterns (and thus recommendations) from them because of too much variance among the group members. Another disadvantage of the algorithm is that its time complexity is in $\mathcal{O}(n^3)$. Due to these limitations, we dropped the Markov Clustering. Instead, we picked the *Agglomerative Hierarchical Clustering* [56] that produces good results with a better time complexity in $\mathcal{O}(n^2 \log(n))$, see Section 3.2.

In addition to the above graph-based algorithms, we considered three spatial clustering algorithms that work on an embedding of the similarity graph. Their common disadvantage is that the time complexity of computing the embedding is in $\mathcal{O}(n^3)$, which is already higher than that of the Agglomerative Hierarchical Clustering.

From the spatial clustering algorithms, *k-Means* [56] does not fit our purpose, as it inevitably assigns each code change to a cluster, i.e., it cannot deal with outliers and unique code changes in a software repository. We also dropped the *DENCLUE* algorithm [26], since we did not find heuristics to choose suitable parameter values that have an acceptable runtime. In our experiments, the heuristics presented by Gan and Li [20] led to promising results, but its computation turned out to be too costly, even for small datasets. Other heuristics that we tried did not yield satisfactory results. From the spatial clustering algorithms, we picked the *DBSCAN* algorithm [14], see Section 3.3. It explicitly considers outliers in the dataset and in combination with our novel heuristics does not suffer from overly large clusters (as the Markov Clustering does).

Hence, after a common pre-preprocessing $\mathcal{C}3$ uses either a fast graph-based clustering or a slower spatial clustering.

3.1 Pre-processing

The clustering algorithms and the heuristics to choose their parameters are geared towards connected graphs. Since the similarity graph may be disconnected, $\mathcal{C}3$ splits the graph into its connected components in a pre-processing step and computes a clustering for each of the components. Nodes in different components are not similar, as otherwise there would be an edge connecting the components. Hence, clusters cannot have nodes from more than one component.

Computing the connected components is fast and scales linearly with the number of nodes and edges [28]. As there are typically many small connected components, splitting the graph and processing the connected components individually reduces the overall runtime because the time com-

plexities of the clustering algorithms that follow scale superlinearly with the number of nodes.

The resulting connected components can still be too large for the DBSCAN algorithm (Section 3.3) to run fast enough. Thus, if $\mathcal{C}3$ encounters a connected component with more than 10,000 nodes, it recursively applies the *Kernighan-Lin* algorithm [35] to split it into components with at most that many nodes. The algorithm divides a graph into two equally sized components by removing edges from the graph. The loss of similarity information has little impact on the clustering, since only edges with small weights are pruned.

3.2 Agglomerative Hierarchical Clustering

Agglomerative Hierarchical Clustering [56] works on the similarity graph. Initially, each node c_i is assigned to its own cluster C_i. The algorithm then repeatedly identifies the two groups C_j and C_k with the highest similarity $s(C_j, C_k)$ between a pair of clusters and merges them into a single cluster C_{jk}. Merging of clusters is repeated until the two most similar clusters have a similarity below a threshold γ.

From the various ways the literature proposes to define the similarity $s(C_j, C_k)$ of a pair of clusters [56], $\mathcal{C}3$ uses a method known as *single linkage* that determines the similarity of C_j and C_k as the highest weighted edge that connects a node from C_j with a node from C_k:

$$s(C_j, C_k) := \max\{\, s(c_u, c_v) \mid c_u \in C_j \wedge c_v \in C_k \,\}\,.$$

$\mathcal{C}3$ uses single linkage instead of other techniques because of its low runtime. In particular, single linkage does not require a costly re-computation of similarity values whenever two clusters are merged. While in general the complexity of the hierarchical clustering algorithm is in $\mathcal{O}(n^3)$, using single linkage turns the algorithm into computing a maximum spanning tree of the similarity graph [21]. Thus, Kruskal's algorithm can compute a clustering with a time complexity of $\mathcal{O}(e \log(n)) \subset \mathcal{O}(n^2 \log(n))$, where e denotes the number of edges and n denotes the number of nodes [11]. Edges with a weight below γ are not considered for the spanning. The connected components of the spanning tree are the resulting clusters, those with one node are discarded as outliers.

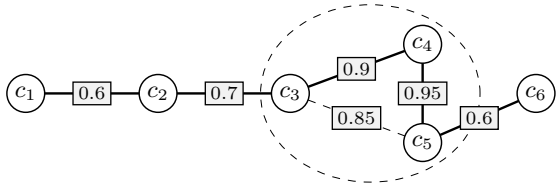
Heuristics to choose γ. A higher threshold γ leads to more and smaller clusters. To choose γ, $\mathcal{C}3$ adapts a heuristics by Fred and Jain [19]. The idea is to compute a spanning tree and to sort the weights of its edges in ascending order. γ is then determined by the largest gap between two consecutive values $s_{\langle i-1 \rangle}$ and $s_{\langle i \rangle}$, where $s_{\langle i \rangle}$ denotes the i^{th} value in the sorted sequence:

$$\gamma := s_{\langle i^* \rangle} \quad \text{with} \quad i^* := \operatorname{argmax}_i (s_{\langle i \rangle} - s_{\langle i-1 \rangle})\,.$$

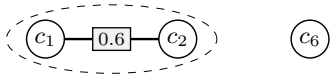
By removing all edges with a weight below γ the spanning tree is split into multiple connected components that correspond to the resulting clusters and outliers.

Consider the similarity graph in Fig. 8(a). The bold edges indicate its maximum spanning tree. The largest gap between two weights in the spanning tree is between the values $s_{\langle i^*-1 \rangle} = 0.7$ and $s_{\langle i^* \rangle} = 0.9$. Thus, we choose $\gamma = 0.9$ and remove all edges from the spanning tree with a weight below 0.9. As a result, we identify the set $\{c_3, c_4, c_5\}$ as the sole cluster. c_1, c_2, and c_3 are not connected by edges with sufficiently high weights and are hence marked as outliers.

Iterative application. After removing the clusters for one γ value from the similarity graph there may still be clusters for lower thresholds. See for example the cluster



(a) A similarity graph and its spanning tree.



(b) Remaining graph after removal of clusters for $\gamma = 0.9$.

Figure 8: Agglomerative Hierarchical Clustering on a graph.

$\{c_1, c_2\}$ in Fig. 8(b). Hence, we propose to apply the clustering algorithm iteratively. We remove code changes that are assigned to a cluster in the previous iteration (as well as their incident edges) from the similarity graph and apply the algorithm on the remaining graph. Re-applying the heuristics to choose a new value for γ isolates more clusters in the remaining graph. This is repeated until only unconnected outliers remain.

3.3 DBSCAN

The DBSCAN algorithm [14] clusters objects that are represented as points in a d-dimensional space. Thus, $\mathcal{C}3$ first computes a d-dimensional embedding of the similarity graph by mapping each code change c_i to a point $y_i \in \mathbb{R}^d$. To do so, $\mathcal{C}3$ computes the *Laplacian Eigenmap* [7] of the similarity graph that preserves its neighborhood structure, i.e., it maps pairs of nodes with a high similarity value to points that are located close together. Note that this closely resembles the technique introduced by Shi and Malik [48], which is also known as *Spectral Clustering* [53, 56]. According to our experiments, a fixed dimensionality of $d = 2$ is a good choice, whereas varying the dimensionality by means of the *Eigengap heuristics* [53] does not lead to better results. In its current implementation, the runtime primarily depends on the solution of an eigenvalue problem which, according to Shi and Malik [48], has a time complexity of $\mathcal{O}(n^3)$.

Let $Y = \{y_1, \ldots, y_n\}$ denote the set of the mappings of all n code changes. The DBSCAN algorithm defines the *ϵ-neighborhood* $N_\epsilon(y_i)$ of a point y_i to be the set of all points within a radius of ϵ around y_i:

$$N_\epsilon(y_i) := \{y_j \in Y \setminus \{y_i\} :\ \|y_i - y_j\| \leq \epsilon\}\,.$$

A point is called *core point* if its ϵ-neighborhood contains at least ρ other points. Since core points become the centers of the clusters and since our goal is to identify groups with at least two members, we set $\rho := 1$. A point y_j is *directly density-reachable* from a core point y_k if y_j lies in the ϵ-neighborhood of y_k. A point y_j is said to be *indirectly density-reachable* from a core point y_k if there is a sequence $y_j =: p_1 \to p_2 \to \cdots \to p_l := y_k$ of points p_i, such that each p_i is directly density-reachable from p_{i+1}. For $\rho = 1$, the clusters identified by the DBSCAN algorithm correspond to the maximal sets of indirectly density-reachable points. Without further optimizations, the DBSCAN algorithm computes these sets with a time complexity of $\mathcal{O}(n^2 d)$ [14].

Heuristics to choose ϵ. The larger the value of ϵ, the more points are assigned to the same group. If ϵ is chosen

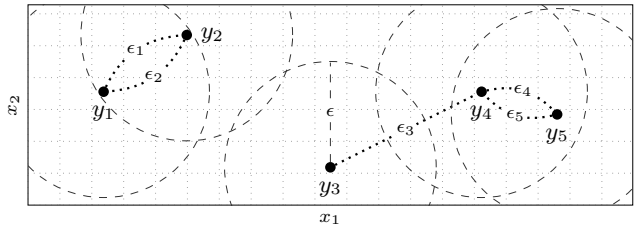


Figure 9: Five points y_i, their ϵ_i, and the resulting ϵ.

too small, the algorithm marks only few points as core points and thus only finds few, small clusters. On the contrary, if ϵ is chosen too large, points that originate from not-so-similar code changes are combined.

To automatically choose a suitable value for ϵ the following new heuristics works well. Let ϵ_i denote the distance from a point y_i to its nearest neighbor, and let $\epsilon_\downarrow := \min_i(\epsilon_i)$ denote the minimum and $\epsilon_\uparrow := \max_i(\epsilon_i)$ denote the maximum of all the ϵ_i. For the five points y_i and their respective distances ϵ_i depicted in Fig. 9, this results in $\epsilon_\downarrow = \epsilon_4 = \epsilon_5$ and $\epsilon_\uparrow = \epsilon_3$. Any value $\epsilon \geq \epsilon_i$ makes y_i a core point, so that the code change c_i is assigned to a cluster of at least two members. For $\epsilon < \epsilon_\downarrow$ no points are marked as core points, so that only outliers but no clusters are identified. For $\epsilon \geq \epsilon_\uparrow$ every point becomes a core point, so that no outliers are detected (which does not imply that all points are assigned to the same cluster). The heuristics thus assume that the "optimal" value of ϵ lies in the range $[\epsilon_\downarrow, \epsilon_\uparrow]$; manual experiments for several datasets confirmed this.

To choose ϵ, we define $\epsilon_{\langle j \rangle}$ to be the j-smallest value of the ϵ_i. We then determine the value of ϵ by means of the largest relative gap between two consecutive values $\epsilon_{\langle j+1 \rangle}$ and $\epsilon_{\langle j \rangle}$:

$$\epsilon := \epsilon_{\langle j^* \rangle} \quad \text{with} \quad j^* := \text{argmax}_j(\epsilon_{\langle j+1 \rangle}/\epsilon_{\langle j \rangle}) .$$

For the points in Fig. 9, a sorted sequence of the distances ϵ_i is $\langle \epsilon_4, \epsilon_5, \epsilon_1, \epsilon_2, \epsilon_3 \rangle$ with the largest gap between ϵ_2 and ϵ_3. Thus, the heuristics choose $\epsilon = \epsilon_2$. As shown in Fig. 9, this leads to two clusters $\{c_1, c_2\}$ and $\{c_4, c_5\}$ and an outlier c_3.

Outliers y_i may have a strong influence on the computation of ϵ, since their ϵ_i is typically high. $\mathcal{C}3$ therefore uses the *Box plot statistic* [50] to ignore the influence of extreme outliers when choosing ϵ. For this purpose, let $\epsilon_{\langle 25\% \rangle}$ denote the lower and $\epsilon_{\langle 75\% \rangle}$ denote the upper quartile of the ϵ_i. $\mathcal{C}3$ marks y_i as an extreme outlier if

$$\epsilon_i \ > \ \epsilon_{\langle 25\% \rangle} + 3(\epsilon_{\langle 75\% \rangle} - \epsilon_{\langle 25\% \rangle}) .$$

Iterative application. Similar to the Agglomerative Hierarchical Clustering, we propose to apply the DBSCAN algorithm iteratively. In each iteration $\mathcal{C}3$ removes all code changes from the embedding that are assigned to a cluster. The clustering algorithm then runs again on the remaining code changes until it cannot find any more clusters.

4. EVALUATION

We start this section with a description of our dataset and the execution environment. Then we answer the following research questions. Section 4.2: Does $\mathcal{C}3$ scale for large datasets? Section 4.3: Does $\mathcal{C}3$ identify relevant clusters? Section 4.4: Are the results useful for state-of-the-art tools? At the end we discuss limitations and threats to validity.

4.1 Dataset and Execution Environment

Our dataset is based on 9 *git*-repositories from active open-source projects that we arbitrarily classify into *small*, *medium*, and *large* projects depending on the respective number of modified methods (see Table 1 for details). We use all commits to the master branch (up to and including 2015) that affect Java code (excluding whitespace and comments) and that have a single parent commit (to skip merges and duplicates). Table 1 lists the Lines Of Code at the end of 2015, the number of analyzed commits, the number of changed Java files, and the number of modified methods in these files (according to Sec. 2.1).

We use 64 nodes equipped with two 2.66 GHz Xeon 5650 Westmere chips (12 cores + SMT) and 24 GB of RAM.

4.2 Scalability

Both of the similarity metrics and the clustering algorithms scale for large datasets.

Metrics. On the right, Table 1 holds the times measured for both the *diff*-metric and the AST-metric. All measurements include the times for the method extraction, the change representation, and the construction of the similarity matrix. The column *Work* shows how long this takes sequentially. Computing the AST-representation is faster than the *diff*-representation (1,525h vs. 102h) as the pairwise LCS computations take longer. With parallelization (1 node/24 threads for small repositories, 64 nodes/1,536 threads otherwise) the total *Wall* time for the completion of the matrix sums up to about 17h for all 9 repositories.

Two reasons cause the non-ideal speedup. First, the sequential file IO to read from the repositories and to write incremental results is a serious bottleneck that puts an upper bound on the speedup (according to Amdahl's law [1]). Second, individual commits have varying execution times as Fig. 10 shows for each of the steps of Sec. 2 (outliers are left out for readability). As $\mathcal{C}3$ statically assigns commits to threads, there is always one thread for a repository that is slower than the rest due to the varying costs. This slowest

Table 1: Dataset overview and runtimes for Sec. 2.

Repository	General Properties (end of 2015)					Time measurement for Sec. 2					
				Changed	Modified	*diff*-Representation			AST-Representation		
	LOC	Commits	Size	Files	Methods	Work	Wall	Increment	Work	Wall	Increment
Cobertura	43,586	335	small	998	3,327	1.2h	1.1h	0.95s	0.43h	0.49h	0.25s
Fitlibrary	74,366	151	small	2,185	4,350	0.50h	0.12h	3.1s	0.09h	0.081h	0.61s
JGraphT	33,554	517	small	2,048	3,262	0.52h	0.13h	0.78s	0.21h	0.17h	0.22s
JUnit	33,871	1,064	small	3,446	4,784	0.91h	0.14h	0.64s	0.13h	0.12h	0.17s
Ant	246,426	6,340	medium	19,082	33,965	44h	0.37h	2.8s	6.3h	0.32h	0.80s
Checkstyle	132,091	3,001	medium	10,978	18,060	12h	1.1h	2.0s	6.5h	1.1h	1.6s
DrJava	224,863	2,759	medium	16,801	45,289	131h	3.0h	43s	11h	2.9h	3.0s
Eclipse JDT	1,334,938	15,407	large	53,299	106,697	928h	6.3h	39s	46h	7.6h	1.5s
Eclipse SWT	378,560	18,373	large	41,473	91,533	407h	4.7h	22s	31h	4.6h	1.4s
Total	2,502,255	47,947	-	150,310	311,267	1525h	17h	-	102h	17h	-

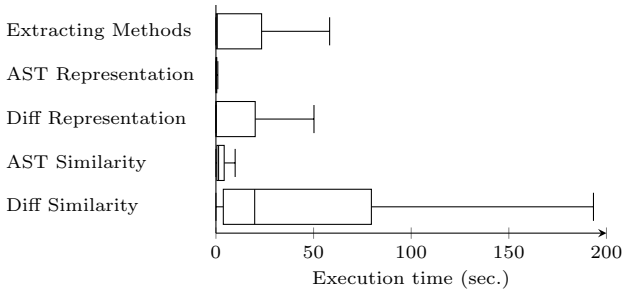


Figure 10: *Work* times per commit for 150,310 changed files (25%/75% quartiles, whiskers: ± 1.5 × interquartile range).

thread determines the *Wall* time. As cheking-out commits from the repository is sequential and larger commits take longer, the 24 threads cannot speed up the similarity computation of Cobertura, which suffers from a high fraction of large commits. Both reasons also strongly affect the parallel execution time of the AST-metric on the Eclipse JDT repository. This leads to a *Wall* time that is higher than for the *diff*-metric.

Both Eclipse repositories are extreme as they accumulate the outliers over all steps of Fig. 10. The outliers in *Extracting Methods* are due to files with more than 100,000 AST nodes (mainly JUnit-Tests) for which $\mathcal{C}3$ sometimes takes over 10 minutes. On large changed methods with several thousand AST nodes $\mathcal{C}3$'s tree differencing takes over 200s to create the edit scripts. For the *diff*-representation, the interprocess communication between Java and *diff* causes the outliers.

Despite those outliers, over 75% of all commits take below 100s with a median in the range of 1.4s to 39s, see the *Increment* column in Table 1 that shows how long it takes (median) to process a single commit for the repository including the time to fill the rows of its code changes in the similarity matrix.

Clustering. The box plots in Fig. 11 show the range of the 18 execution times for the two sequential clustering algorithms (9 repositories with 2 representations). Even for large repositories, the sequential hierarchical clustering only takes a couple of minutes. Due to the computation of the eigenvalues, DBSCAN requires several hours. Considering the large size of the Eclipse repositories, this is still an acceptable time. The connected components can be processed in parallel (future work) for both clustering algorithms.

4.3 Relevance

To show that $\mathcal{C}3$ finds relevant clusters we must evaluate whether it finds most of the similar code changes that are present in the repositories, and whether the resulting clusters both contain changes that are cohesive and exclude changes that are not similar enough. Unfortunately, there is

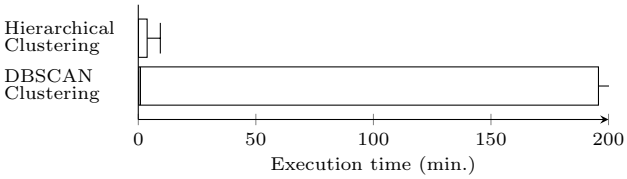


Figure 11: Sequential time of the clustering algorithms (25%/75% quartiles, whiskers: ± 1.5 × interquartile range).

Table 2: Clustering results measured within the 9 projects.

	AH	AD	dH	dD
Clusters	46,286	69,327	89,775	119,096
...Cluster Members	288,411	299,761	261,009	282,436

no test data set for code changes to answer these questions.

Table 2 shows that depending on the combinations of metric (AST vs. *diff*) and clustering algorithm (Hierarchical vs. DBSCAN) $\mathcal{C}3$ identifies a total of almost 300,000 unique changes and separates them into 46,286 to 119,096 clusters of similar changes. Regardless of the used metric, DBSCAN tends to find more clusters of similar changes and the absolute number of changes in those clusters is also larger than for the other combinations of metrics and clustering algorithms. These numbers not only show that developers perform many repetitive changes but also make it impractical to manually inspect the full results. Furthermore, it is hard to devise objective criteria for similarity that allow a concise manual evaluation of the clustering results by independent users of $\mathcal{C}3$.

Therefore, we take a different approach to evaluate the relevance of the detected clusters. Meng et al. [42] have manually identified 24 clusters of bug fixes in the two Eclipse repositories with a total of 149 similar changes. This gives us a manually labeled independent data set that we can use to discuss the relevance of the cluster that $\mathcal{C}3$ finds.

From these 24 *Manual* clusters, 23 are present in $\mathcal{C}3$'s clustering results, only a small change is not recognizable as separate cluster. As the clusters found in $\mathcal{C}3$'s results are similar to the *Manual* clusters, both with respect to their sizes and members, this is an indication that $\mathcal{C}3$ also finds many of the similar changes in the other repositories and that their clustering is also reasonable.

Table 3 holds the details. Some of the clusters that $\mathcal{C}3$ finds are *Identical* to the *Manual* clusters, i.e., the same set of similar changes is grouped together. In some cases (*Divided*), $\mathcal{C}3$ splits a *Manual* cluster into two or more smaller ones that together hold exactly the same code changes as the *Manual* cluster. In some cases (*Partial*), $\mathcal{C}3$ forms the same clusters except that one (*-1*), two (*-2*), or more (≥ *-2*) code changes are missing in the clusters. This is caused by larger variations between the changes within one *Manual* cluster. Due to these differences, either the metric causes a higher similarity of such a change to a different cluster or the cluster algorithm discards the change as an outlier. In general, the partial clusters only miss a few of the code changes that Meng et al. have grouped in their *Manual* clusters.

In total, depending on the combination of metric and clus-

Table 3: Performance on test data set.

		AH	AD	dH	dD
	Manual Cl.	24			
	...Members	149			
$\mathcal{C}3$	Identical Cl.	6	1	8	3
	...Members	31	12	44	9
	Divided Cl.	2	5	2	7
	...Members	11	24	13	52
	Partial Cl. (-1)	7	10	8	5
	...Members	23	43	41	19
	Partial Cl. (-2)	4	4	3	4
	...Members	36	12	13	18
	Partial Cl. (≥ -2)	2	3	1	2
	...Members	9	22	5	18
	Total	21(88%)	23(96%)	22(92%)	21(88%)
	...Members	110(74%)	113(76%)	116(78%)	116(78%)

tering algorithm, $\mathcal{C}3$ finds 21–23 of the 24 manually identified clusters of similar changes and it finds 110–119 of the 149 code changes that Meng et al. use. $\mathcal{C}3$ does not find any supersets of the *Manual* clusters and there is no cluster that contains changes from 2 or more different *Manual* clusters.

There is no longer a clear advantage for DBSCAN on the *Manual* clusters. The DBSCAN-versions only find a subset of the *Identical* clusters that the Hierarchical-versions find. The missed ones show up as *Divided* or *Partial*. 5 of the 6 *Identical* AH-clusters are also found by dH. The one *Identical* cluster from AH that dH misses contains an extra line in one of the five code changes in the *Manual* cluster. This line renders the *diff*-representations too dissimilar and so this cluster ends up as *Partial* in the dH results, missing one member.

There is only one *Manual* cluster that does not appear in any of the results of $\mathcal{C}3$'s four combinations. The code changes in this cluster only add a check of a condition to a method and thus is too similar to over 50 other changes. The clustering algorithms simply are not able to isolate it.

None of the four combinations of metrics and clustering algorithms is superior for this test data set. AD identifies the most clusters, dH the most *Identical* ones, and one particular *Identical* cluster is only found by AH. Only dD has no clear advantage over the other combinations.

Overall, all versions find a high percentage of the *Manual* clusters (88%–96%) including around 75% of their similar changes. Thus, at least for this test data set, $\mathcal{C}3$ can identify relevant clusters. Only one cluster contains code changes from the *Manual* set and code changes from the repositories not present in this set. This means that $\mathcal{C}3$ also separates clusters of code changes successfully and does not put unrelated changes together.

4.4 Case-Study with LASE

$\mathcal{C}3$ discovers the clusters from Meng et al., but $\mathcal{C}3$ finds many more clusters of similar changes. In this section, we examine the usefulness of the remaining clusters.

To evaluate the clusters we use the following methodology: We first feed each cluster of similar code changes found by $\mathcal{C}3$ into a publicly available implementation of LASE (see https://www.cs.utexas.edu/%7Emengna09/projects.html) and let LASE generalize the changes in that cluster. Only if the variation among the members of the clusters is not too broad, LASE generates a generalization that can be used to recommend code changes (e.g. bug fixes). Then we use the most recent version from the repository that has not yet seen any of the code changes in the cluster. If LASE finds recommendations for this version based on the generalization derived from the input cluster, we have found a cluster that is potentially useful for developers.

From all the clusters from Table 2, up to 37% are useful in this sense (the exact numbers are listed in Table 4). For 55% (AH) to 65% (dD) of these useful clusters, LASE is able to recommend *all* code changes contained in the respective cluster. For 85% (AH) to 90% (dD), LASE recommends

Table 4: Useful clusters (i.e. clusters that produce recommendations with LASE) found in the 9 repositories.

	AH	AD	dH	dD
Useful Clusters	16,909	20,940	33,456	41,909
Useful Clusters of size 2	11,667	15,780	25,273	35,708

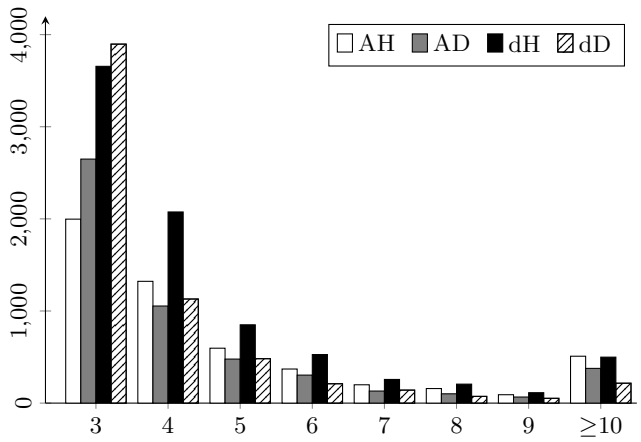


Figure 12: Distribution of useful clusters of at least size 3 for the 9 repositories.

at least *one* of these changes. Hence, LASE and similar tools can benefit from automatically derived inputs as it is unlikely that human developers would and could find so many clusters with so many similar code changes. Note that we do not evaluate LASE here. We (only) evaluate, whether LASE can use the clusters of similar code changes identified by $\mathcal{C}3$ as an input from which it can generate some useful results.

Moreover, one can reason that it is more likely for a change to be a recurrent one if it occurred more often in the repository. For a pair of two similar changes, it might be harder to find many more spots to which the bug fix is applicable. But if a bug fix has already been applied to, for example, 10 spots, it is more likely that there are additional spots that have been missed by the developers. Hence, the sizes of the clusters matter. Fig. 12 illustrates that $\mathcal{C}3$ not only finds useful clusters but also that up to 31% of them contain at least 3 code changes. Those are more likely to be repetitive.

As in all the previous subsections of the evaluation, we also evaluate whether one of the four combinations of metrics and clustering algorithms performs best. While dD identifies less *Identical* clusters in the test data set, this time dD works best as it again produces most clusters that also contain the highest total number of members. But dD also has two disadvantages. First, it does not only identify the most useful clusters, it also identifies the most clusters that are not useful. Second, a large portion of these clusters are just pairs (that are less likely to indicate repetitive changes).

In contrast, AH and DH identify fewer clusters, but these clusters tend to be of larger sizes. As above, the reason is that the heuristics of the hierarchical clustering lead to larger clusters.

4.5 Limitations

A threat to the external validity is the focus on Java. In the future, we plan to also analyze projects that use other programming languages by replacing the parser front-end. All other parts of $\mathcal{C}3$ are language independent.

A limitation of $\mathcal{C}3$ are the outliers of *Method Extraction* times for large commits. We plan to evaluate whether faster tree differencing algorithms (e.g. Gumtree [15]) or other approaches (e.g. OAT [46]) can be a remedy. Another current limitation is that both clustering implementations are not parallelized yet and do not compute the clustering incre-

mentally when a new commit is added to the repository. We are currently working on a solution to these problems. In particular, we plan to incrementally compute the embedding of the similarity graph to speed up the DBSCAN clustering [31].

None of the four combinations of metrics and clustering algorithms performs best in all cases. Since in practice applications most likely only use one combination to compute a single clustering, we plan to work on techniques to combine and integrate the results of the different combinations.

The evaluation in Section 4.3 that shows the relevance of the automatically detected clusters suffers from the fact that there is no large data set with manually picked clusters of similar changes in the literature. To increase the confidence in the results, we plan to recruit students to identify more clusters of similar code changes manually.

We claim that the detected clusters are useful for different applications. The case study illustrates this for LASE only, and thus there is the potential risk that other tools cannot make as much use of the clusters. In order to deal with this threat, more and different case studies would help.

5. RELATED WORK

Fundamentally different to this paper are approaches that cluster software systems into smaller, more comprehensible, and easier to maintain subsystems [2, 3, 9, 10, 40, 52, 55].

A lot closer is an area of work that does not deal with code changes, but addresses repetitive code clones. Tools like CBCD [38], CCFinder [33], CP-Miner [39], dup [4], or the one introduced by Baxter et al. [6] find syntactically identical (or "almost identical") code *fragments*. $\mathcal{C}3$ differs in two ways: It deals with code *changes* and it detects groups of *similar* instances instead of *identical* ones. Hence, $\mathcal{C}3$ requires similarity metrics and clustering algorithms. One exception is DECKARD [32] that detects similar fragments by encoding AST subtrees as feature vectors. These vectors cannot be used to encode the characteristics of code changes.

Developers often perform a few unrelated changes in a single commit. Several approaches untangle such changes and split commits into smaller subsets of related changes. Textually the code of these related changes may look completely different, but such changes typically affect related aspects of a program (data structures, variables, objects and their fields, call graph, flow of control, etc.). ClusterChanges [5] analyzes *def-use* relationships between changed code regions. Kawrykow [34] exploits overlapping sets of identifiers or changes with a common superclass. Herzig and Zeller [23] check whether changes access the same variables or how similar the package names of changed files are. EpiceaUntangler [12] analyzes if changed code regions call methods with the same name or if changed methods call each other. While works on change untangling extract relationship information and perform a clustering to find changes that belong together, $\mathcal{C}3$ finds similar and repetitive changes that not necessarily have a semantically close relationship and that can also be found across projects. As $\mathcal{C}3$ works on method granularity, a preceding untangling could improve the results if it splits unrelated changes within methods.

Closest to $\mathcal{C}3$ is work that extracts change patterns from software repositories and that detects recurring code changes. Approaches differ in their representation of code changes, in when they consider changes to be similar, and whether and how they build groups of similar changes.

BugMem [36] extracts *individual* bug fixes from repositories and represents them as filtered lists of (slightly abstracted) AST subtrees to warn developers if newly written code is likely to contain a bug. It does not cluster the bug fixes but considers each change as a single pattern. Similarly, Hashimoto et al. [22] create an edit script for each extracted code change individually and store it in a fact base. Manual queries against the fact base yield matching change patterns. Hora et al. [29] represent code changes as rules that describe which method invocation was replaced with which other. They do not form clusters of changes, but only merge *pairs* of changes whose rules only differ in certain parts. Similarly, FixWizard [47] detects *pairs* of recurring bug fixes to code peers in object-oriented programs, i.e., to classes or methods that serve a similar purpose or that have a similar object interaction pattern. Two changes are considered recurring if their impact to the interaction patterns is sufficiently similar. Our definition of similar changes is broader and $\mathcal{C}3$ finds larger clusters, nut just pairs of changes.

Higo and Kusumoto [24] represent code changes as sets of deleted and inserted statements and they merge *identical* representations into groups. Similarly, Nguyen et al. [45] detect groups of recurring code changes whose pairs of ASTs are *identical* after a normalization with respect to literals and local variables. Wang et al. [54] consider the change blocks generated by *diff* as code changes (which is affected by long distance moves, see Sec. 2). If the token streams of two or more changes have large *identical* parts, they are merged into a group. In contrast, $\mathcal{C}3$ allows more *variation* among the code changes due to the *LCS*-based metrics.

Two more lines of work are orthogonal to $\mathcal{C}3$: Negara et al. [44] detect groups of recurring code changes by inspecting the edit operations within an IDE instead of mining software repositories. Fluri et al. [17] represent code changes by counting the occurrences of change types (e.g. *statement insert*, *parameter delete*, etc.) and detect clusters of related, co-occurring change types instead of similar changes.

Finally, change classification assigns code changes to predefined classes, for example to categorize changes into maintenance categories [25], to characterize the impact of a change on a class [13], or to detect erroneous changes [37, 49]. In contrast, $\mathcal{C}3$ detects previously unknown classes.

6. CONCLUSION

$\mathcal{C}3$ is the first tool that automatically identifies and clusters similar code changes in source code repositories. To compare pairs of code changes from modified methods, $\mathcal{C}3$ uses either a *diff*-based metric or a novel AST-based metric. To automatically cluster code changes based on these similarity-values, $\mathcal{C}3$ applies two clustering algorithms.

The evaluation shows that, when executed on a cluster computer, $\mathcal{C}3$ is able to identify clusters of code changes even in large repositories within a couple of hours. It also shows that $\mathcal{C}3$ fully automatically detects up to 96% of a set of manually identified bug fix clusters. Additionally, $\mathcal{C}3$ extracts over 10,000 clusters from the repositories of 9 open source projects that work directly with LASE, a state-of-the-art code recommendation tool.

To provide an initial database for code change research, we open-source the identified clusters of code changes at https://github.com/FAU-Inf2/cthree.

7. REFERENCES

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *SJCC'67: Spring Joint Computer Conf.*, pages 483–485, Atlantic City, NJ, Apr. 1967.

[2] P. Andritsos and V. Tzerpos. Information-Theoretic Software Clustering. *IEEE Trans. on Softw. Eng.*, 31(2):150–165, Feb. 2005.

[3] N. Anquetil, C. Fourrier, and T. C. Lethbridge. Experiments with Clustering As a Software Remodularization Method. In *WCRE'99: Working Conf. on Reverse Eng.*, pages 235–255, Atlanta, GA, Oct. 1999.

[4] B. S. Baker. On Finding Duplication and Near-duplication in Large Software Systems. In *WCRE'95: Working Conf. on Reverse Eng.*, pages 86–95, Toronto, Canada, July 1995.

[5] M. Barnett, C. Bird, J. Brunet, and S. Lahiri. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *ICSE'15: Intl. Conf. on Softw. Eng.*, Florence, Italy, May 2015.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *ICSM'98: Intl. Conf. on Softw. Maintenance*, pages 368–377, Bethesda, MD, Nov. 1998.

[7] M. Belkin and P. Niyogi. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation*, 15(6):1373–1396, June 2003.

[8] L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *SPIRE'00: String Processing and Inf. Retrieval Symp.*, pages 39–48, A Coruna, Spain, Sep. 2000.

[9] D. Beyer and A. Noack. Clustering Software Artifacts Based on Frequent Common Changes. In *IWPC'05: Intl. Workshop on Program Comprehension*, pages 259–268, St. Louis, MO, May 2005.

[10] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello. Investigating the Use of Lexical Information for Software System Clustering. In *CSMR'11: European Conf. on Soft. Maintenance and Reengineering*, pages 35–44, Oldenburg, Germany, March 2011.

[11] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3rd edition, 2009.

[12] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling Fine-Grained Code Changes. In *SANER'15: Intl. Conf. on Softw. Analysis, Evolution, and Reengineering*, pages 341–350, Montréal, Canada, March 2015.

[13] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic. Using Stereotypes to Help Characterize Commits. In *ICSM'11: Intl. Conf. on Softw. Maintenance*, pages 520–523, Williamsburg, VA, Sep. 2011.

[14] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD'96: Intl. Conf. on Knowledge Discovery and Data Mining*, pages 226–231, Portland, OR, Aug. 1996.

[15] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ASE'14: Intl. Conf. Automated Softw. Eng.*, pages 313–324, Västerås, Sweden, Sep. 2014.

[16] B. Fluri and H. C. Gall. Classifying Change Types for Qualifying Change Couplings. In *ICPC'06: Intl. Conf. on Program Comprehension*, pages 35–45, Athens, Greece, June 2006.

[17] B. Fluri, E. Giger, and H. C. Gall. Discovering Patterns of Change Types. In *ASE'08: Intl. Conf. on Automated Softw. Eng.*, pages 463–466, L'Aquila, Italy, Sep. 2008.

[18] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[19] A. L. N. Fred and A. K. Jain. Combining Multiple Clusterings Using Evidence Accumulation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 27(6):835–850, June 2005.

[20] W. Gan and D. Li. Optimal Choice of Parameters for a Density-Based Clustering Algorithm. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, volume 2639 of *Lecture Notes in Computer Science*, pages 603–606. Springer Berlin Heidelberg, 2003.

[21] J. C. Gower and G. J. S. Ross. Minimum Spanning Trees and Single Linkage Cluster Analysis. *Applied Statistics*, 18(1):54–64, 1969.

[22] M. Hashimoto, A. Mori, and T. Izumida. A Comprehensive and Scalable Method for Analyzing Fine-Grained Source Code Change Patterns. In *SANER'15: Intl. Conf. on Softw. Analysis, Evolution and Reengineering*, pages 351–360, Montréal, Canada, March 2015.

[23] K. Herzig and A. Zeller. The Impact of Tangled Code Changes. In *MSR'13: Working Conf. on Mining Software Repositories*, pages 121–130, San Francisco, CA, May 2013.

[24] Y. Higo and S. Kusumoto. Identifying Clone Removal Opportunities Based on Co-evolution Analysis. In *IWPSE'13: Intl. Workshop on Principles on Software Evolution*, pages 28–37, Saint Petersburg, Russia, Aug. 2013.

[25] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic Classication of Large Changes into Maintenance Categories. In *ICPC'09: Intl. Conf. on Program Comprehension*, pages 30–39, Vancouver, Canada, May 2009.

[26] A. Hinneburg and H.-H. Gabriel. DENCLUE 2.0: Fast Clustering Based on Kernel Density Estimation. In *IDA'07: Intl. Conf. on Intelligent Data Analysis*, pages 70–80, Ljubljana, Slovenia, Sep. 2007.

[27] D. S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[28] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[29] A. Hora, N. Anquetil, S. Ducasse, and M. Valente. Mining System Specific Rules from Change Patterns. In *WCRE'13: Working Conf. on Reverse Eng.*, pages 331–340, Koblenz, Germany, Oct. 2013.

[30] J. Jacobellis, N. Meng, and M. Kim. Cookbook: In

Situ Code Completion Using Edit Recipes Learned from Examples. In *ICSE'14: Intl. Conf. Softw. Eng.*, pages 584–587, Hyderabad, India, May 2014.

[31] P. Jia, J. Yin, X. Huang, and D. Hu. Incremental Laplacian Eigenmaps by Preserving Adjacent Information Between Data Points. *Pattern Recogn. Lett.*, 30(16):1457–1463, Dec. 2009.

[32] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *ICSE'07: Intl. Conf. on Softw. Eng.*, pages 96–105, Minneapolis, MN, May 2007.

[33] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. on Softw. Eng.*, 28(7):654–670, July 2002.

[34] D. Kawrykow. Enabling Precise Interpretations of Software Change Data. Master's Thesis, School of Computer Science, McGill University, Montreal, Aug. 2011.

[35] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, Feb. 1970.

[36] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of Bug Fixes. In *SIGSOFT'06/FSE-14: Intl. Symp. on Foundations of Softw. Eng.*, pages 35–45, Portland, OR, Nov. 2006.

[37] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Trans. on Softw. Eng.*, 34(2):181–196, March 2008.

[38] J. Li and M. D. Ernst. CBCD: Cloned Buggy Code Detector. In *ICSE'12: Intl. Conf. on Softw. Eng.*, pages 310–320, Zürich, Switzerland, June 2012.

[39] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI'04: Symp. on Opearting Systems Design & Implementation*, pages 289–302, San Francisco, CA, Dec. 2004.

[40] O. Maqbool and H. Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Trans. on Softw. Eng.*, 33(11):759–780, Nov. 2007.

[41] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does Automated Refactoring Obviate Systematic Editing? In *ICSE'15: Intl. Conf. Softw. Eng. - Volume 1*, pages 392–402, Florence, Italy, May 2015.

[42] N. Meng, M. Kim, and K. S. McKinley. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *ICSE'13: Intl. Conf. Softw. Eng.*, pages 502–511, San Francisco, CA, May 2013.

[43] S. Needleman and C. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *J. Molecular Biol.*, 48(3):443–453, March 1970.

[44] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In *ICSE'14: Intl. Conf. on Softw. Eng.*, pages 803–813, Hyderabad, India, May 2014.

[45] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A Study of Repetitiveness of Code Changes in Software Evolution. In *ASE'13: Intl. Conf. on Automated Softw. Eng.*, pages 180–190, Palo Alto, CA, Nov. 2013.

[46] H. A. Nguyen, T. T. Nguyen, J. G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen. A Graph-based Approach to API Usage Adaptation. In *OOPSLA'10: Proc. Intl. Conf. Object-Oriented Progr., Systems, Languages & Appl.*, pages 302–321, Reno/Tahoe, NV, Oct. 2010.

[47] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring Bug Fixes in Object-oriented Programs. In *ICSE'10: Intl. Conf. on Softw. Eng. - Volume 1*, pages 315–324, Cape Town, South Africa, May 2010.

[48] J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 22(8):888–905, Aug. 2000.

[49] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding Failure-inducing Changes in Java Programs Using Change Classification. In *SIGSOFT'06/FSE-14: Intl. Symp. on Foundations of Softw. Eng.*, pages 57–68, Portland, OR, Nov. 2006.

[50] J. W. Tukey. *Exploratory Data Analysis.* Addison-Wesley, Reading, MA, 1977.

[51] S. Van Dongen. Graph Clustering Via a Discrete Uncoupling Process. *SIAM J. on Matrix Analysis and Applications*, 30(1):121–141, Feb. 2008.

[52] A. Vanya, L. Hofland, S. Klusener, P. Van De Laar, and H. Van Vliet. Assessing Software Archives with Evolutionary Clusters. In *ICPC'08: Intl. Conf. on Program Comprehension*, pages 192–201, Amsterdam, The Netherlands, June 2008.

[53] U. Von Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[54] S. Wang, D. Lo, and X. Jiang. Understanding Widespread Changes: A Taxonomic Study. In *CSMR'13: European Conf. on Softw. Maintenance and Reengineering*, pages 5–14, Genova, Italy, March 2013.

[55] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *WCRE'97: Working Conf. on Reverse Eng.*, pages 33–43, Amsterdam, The Netherlands, Oct. 1997.

[56] M. J. Zaki and W. Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms.* Cambridge University Press, New York, NY, 2014.

[57] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive Code Review for Systematic Changes. In *ICSE'15: Intl. Conf. Softw. Eng. - Volume 1*, pages 111–122, Florence, Italy, May 2015.