Software Evolution - Clone Detection

Anandan Krishnsamy (14874121), Nefeli Tavoulari (15043347)

December 2023

Universiteit van Amsterdam

Contents

1 Introduction

Cloning affects the software's evolvability. Detecting, managing, and removing software redundancy aids investigations into software reuse, refactoring, modularization, and parameterization. Reasons for duplication include code scavenging (frequent use of copy and paste), intentional duplication for efficiency concerns, and parallel or forked development. Understanding a system's clones is essential for making consistent changes during its evolution.[4]

Types of Clones:

1. Type I: An identical duplication without alterations, excluding whitespaces and comments.
2. Type II: A copy maintaining syntactical identity, with modifications limited to altering variable names, types, or function identifiers.
3. Type III: A duplicated version with additional modifications where statements are altered, added, or removed, diverging from the original structure.
4. Type IV: The functionality is the same, code may be completely different.

[6] provides detailed information on clone detection, including the process stages, various clone detection algorithms, and visualization methods. In this assignment, we examine and implement AST-based clone detection, using it to produce clones and gather statistics about clones for three Java projects: TestProject, smallsql, and hsqldb.

2 Project structure

```
/ src
  Algorithms
    SubtreeClones.rsc
    SequenceClones.rsc
    GeneralizeClones.rsc
  Tests
    SubtreeClonesTests.rsc
    SequenceClonesTests.rsc
  Visualization
    ExportJson.rsc
  Lib
    Utilities.rsc
    Statistics.rsc
  Main.rsc
/ results
/ clone-visualization
```

Algorithms folder: contains all the algorithm implementations based on the given literature.
Tests folder: contains all the tests on the algorithms, using the smallsql project, and a test project that we created, and that is included in our submission.
Lib folder: contains some commonly used functions, as well as some functions that calculate project volume.
Main file: script to run the algorithms on any project.
clone-visualization: a React Vite front end app and a Python spreadsheet visualization generaotor
results: json files containing the clone classes of each algorithm.

3 Instructions

To run the project from the Rascal terminal:

- import Main;
- choose configuration, e.g. massThreshold, cloneType, project.
- main();

To run tests:

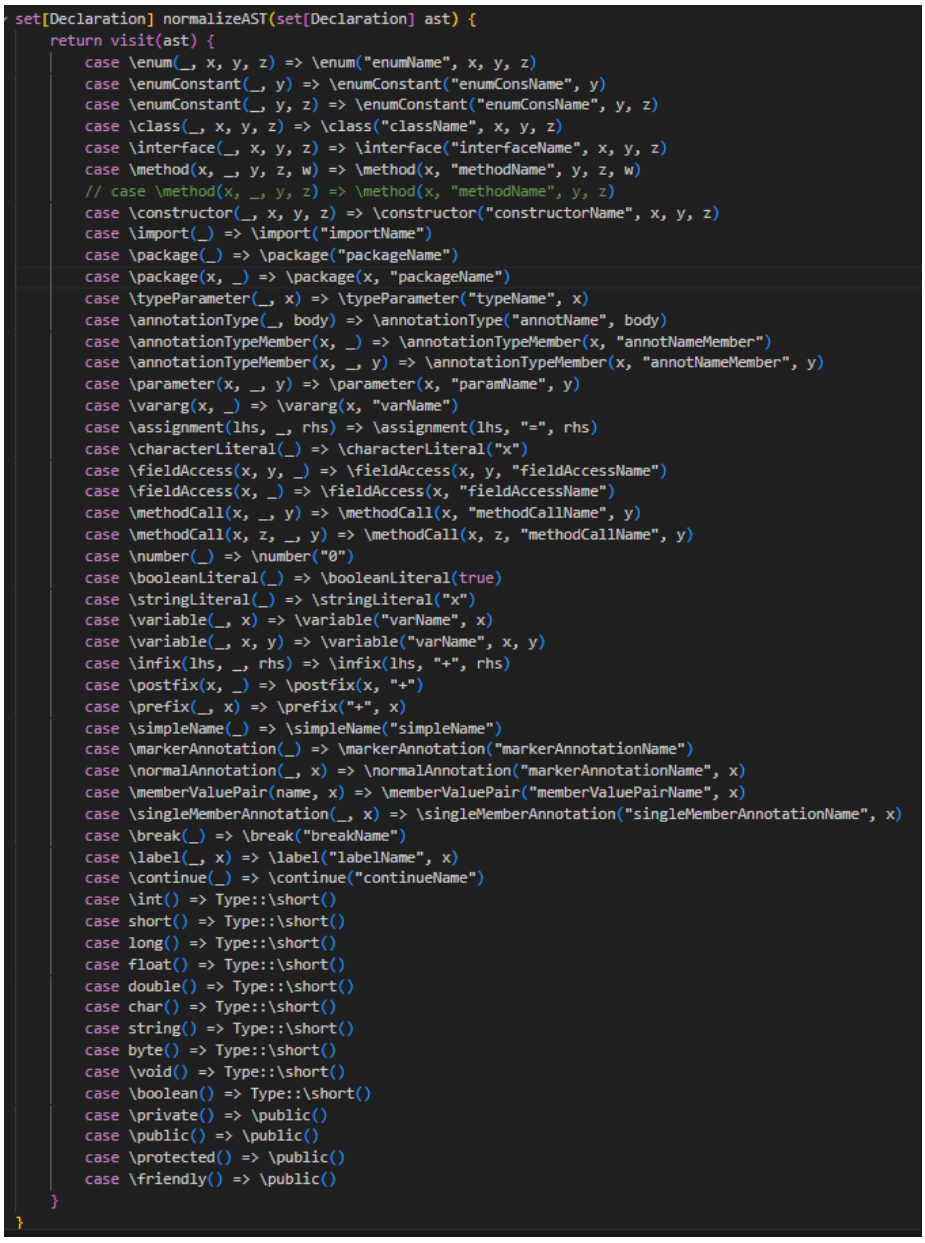
- import Main;
- import <testsFile>;
- :test

4 Algorithms

We implemented and compared the three algorithms described in the Clone Detection Using Abstract Syntax Trees paper [2] using rascal[5], subsequently we made necessary modifications to enhance their effectiveness and reduce duplicate clones.

4.1 Sub-tree Clones

This algorithm finds sub-tree clones. The stages followed:
1) **Preprocessing**
For clone type 2 and 3, we need to clean the AST of any identifiers. For that we went through the rascal documentation and put a standard value for every name or type.
2) **Hash Table Creation**
We store the potential clones in a hash table to decrease the number of comparisons. The hash function used is the md5Hash function, which seems to



```
set[Declaration] normalizeAST(set[Declaration] ast) {
    return visit(ast) {
        case \enum(_, x, y, z) => \enum("enumName", x, y, z)
        case \enumConstant(_, y) => \enumConstant("enumConsName", y)
        case \enumConstant(_, y, z) => \enumConstant("enumConsName", y, z)
        case \class(_, x, y, z) => \class("className", x, y, z)
        case \interface(_, x, y, z) => \interface("interfaceName", x, y, z)
        case \method(x, _, y, z, w) => \method(x, "methodName", y, z, w)
        // case \method(x, _, y, z) => \method(x, "methodName", y, z)
        case \constructor(_, x, y, z) => \constructor("constructorName", x, y, z)
        case \import(_) => \import("importName")
        case \package(_) => \package("packageName")
        case \package(x, _) => \package(x, "packageName")
        case \typeParameter(_, x) => \typeParameter("typeName", x)
        case \annotationType(_, body) => \annotationType("annotName", body)
        case \annotationTypeMember(x, _) => \annotationTypeMember(x, "annotNameMember")
        case \annotationTypeMember(x, _, y) => \annotationTypeMember(x, "annotNameMember", y)
        case \parameter(x, _, y) => \parameter(x, "paramName", y)
        case \vararg(x, _) => \vararg(x, "varName")
        case \assignment(lhs, _, rhs) => \assignment(lhs, "=", rhs)
        case \characterLiteral(_) => \characterLiteral("x")
        case \fieldAccess(x, y, _) => \fieldAccess(x, y, "fieldAccessName")
        case \fieldAccess(x, _) => \fieldAccess(x, "fieldAccessName")
        case \methodCall(x, _, y) => \methodCall(x, "methodCallName", y)
        case \methodCall(x, z, _, y) => \methodCall(x, z, "methodCallName", y)
        case \number(_) => \number("0")
        case \booleanLiteral(_) => \booleanLiteral(true)
        case \stringLiteral(_) => \stringLiteral("x")
        case \variable(_, x) => \variable("varName", x)
        case \variable(_, x, y) => \variable("varName", x, y)
        case \infix(lhs, _, rhs) => \infix(lhs, "+", rhs)
        case \postfix(x, _) => \postfix(x, "+")
        case \prefix(_, x) => \prefix("+", x)
        case \simpleName(_) => \simpleName("simpleName")
        case \markerAnnotation(_) => \markerAnnotation("markerAnnotationName")
        case \normalAnnotation(_, x) => \normalAnnotation("markerAnnotationName", x)
        case \memberValuePair(name, x) => \memberValuePair("memberValuePairName", x)
        case \singleMemberAnnotation(_, x) => \singleMemberAnnotation("singleMemberAnnotationName", x)
        case \break(_) => \break("breakName")
        case \label(_, x) => \label("labelName", x)
        case \continue(_) => \continue("continueName")
        case \int() => Type::\short()
        case short() => Type::\short()
        case long() => Type::\short()
        case float() => Type::\short()
        case double() => Type::\short()
        case char() => Type::\short()
        case string() => Type::\short()
        case byte() => Type::\short()
        case \void() => Type::\short()
        case \boolean() => Type::\short()
        case \private() => \public()
        case \public() => \public()
        case \protected() => \public()
        case \friendly() => \public()
    }
}
```

Figure 1: Normalization of AST to find near miss clones

perform as we wanted it, since it creates different buckets for different nodes. We iterate over the sub-trees of the tree and hash every one of them. We use the unsetRec function, that basically "cleans" the sub-tree off any locations, that would obviously make every node seem different from the others. We skip the nodes that have a mass smaller than the mass threshold, because that means they are probably close to the leaves and do not contain such important information. So, sub-trees with the same hash end up in the same bucket. We chose the massThreshold of 6, because after some tests, we realized that if we had a lower massThreshold we would do some redundant operations on leaves. Also, with a higher massThreshold we would miss many important clones. Plus, we tried using tha arity function, with which our program ran very fast, having less buckets. However, we thought that it is better having many buckets but catch more clones, so we used our custom function for finding children of a node.

Listing 1: Hash Table Creation

```
1  tuple[map[str, list[node]], map[node, list[value
       ]]] createSubtreeHashTable(list[Declaration]
       ast, int massThreshold, bool generalize) {
2  map[str, list[node]] hashTable = ();
3  map[node, list[value]] childrenOfParents = ();
4  visit (ast) {
5      case node n: {
6          if (generalize) {
7              // Storing parents child information
                   for future use
8              list[value] children = getChildren(n);
9              str childrenString = toString(children
                   );
10             if (startsWith(childrenString, "[[")
                   {
11                 childrenOfParents[n] = children
                       [0];
12             } else {
13                 childrenOfParents[n] = getChildren
                       (n);
14             }
15         }
16         // Hashing
17         if (subtreeMass(unsetRec(n)) >=
               massThreshold) {
18             str hash = md5Hash(unsetRec(n));
19             if (hash in hashTable) {
20                 hashTable[hash] += n;
21             } else {
22                 hashTable[hash] = [n];
23             }
```

```
            }
        }
    }
    return <hashTable, childrenOfParents>;
}
```

3) **Clone Detection**

For clone type 1, if two subtrees are in the same bucket, they must be equal, so we do not check anything else. For clone type 2 and 3, we iterate over each bucket and compare every pair of sub-trees, using the compareTree function. Similarity = 2 x S / (2 x S + L + R) where: S = number of shared nodes, L = number of different nodes in sub-tree 1, R = number of different nodes in sub-tree 2. If the clone type is 2, the result of compareTree should be 1.0, so that the two sub-trees are identical, with the difference that identifiers are ignored. if the clone type is 3, the similarity threshold can be lower, e.g. 0.8, so that we also include in our results non identical but still similar pieces of code.

4) **Clone Gathering**

At this stage, we add the clones into the final struct, a list of tuples, consisting of two nodes, that represent a clone pair. Before we do that though, we need to check if the current pair is a sub-clone of an existing pair in the struct. Because in that case, there is no reason in adding it, since we need the biggest sub-tree that is cloned. We do that by visiting the struct's pair's sub-trees and checking if these children match the given pair. For the case that the flipped pair is already in the struct, we do not add it. Also, we need to check that there are not sub-clones of the current pair in the struct. If there are, they should be removed, because they are no more needed, for the same reason as above. We do that by visiting the given pair's sub-trees and checking if these children match the clones' struct's pair.

5) **Statistics Calculation**

Regarding the statistics we are calculating, for performance purposes, we have a function that calculates all of them at the same time, saving some time.
The results we are printing on the screen are:
- an example of a clone pair
- number of clone pairs
- number of clone classes: we basically had to find the different code piece that is repeated every time. So we created a map that has as key a node and as value a set of all its clones. Of course we took great care so that not both members of a pair are added as a key to the map.
- biggest clone in lines: we used unitLOC from series-1[3].
- biggest clone class in members: having stored the classes in our map, it was easy to find the one that had the biggest size of the set.
- To calculate the percentage of duplicated lines, we first determined the total lines of code (LOC) for the entire project using data from Series-1. We then computed the LOC for each class, ensuring no repetition of lines. This precaution was necessary because a class can be nested within another class. To manage this, we created a map where the key was the filename and the value

was a set of its lines. By iterating over each class and adding its lines to the set, we effectively eliminated any duplicates, as sets inherently remove duplicate entries.

4.2 Sequence Clones

This algorithm finds sequence clones. The stages followed:
1) **Hash Table Creation**
We store the potential clones in a hash table to decrease the number of comparisons. The hash function used is the md5Hash function. We iterate over the subsequences of size at least as the minimumSequenceLengthThreshold and hash every node of them. We chose minimum length of 6, since that was suggested by SIG as we learnt in series-1. We use the unsetRec function, that basically "cleans" the node off any locations, that would obviously make every node seem different from the others. We accumulate the subsequence hashes in one string, which is then also hashed as a whole. This seems to work as expected, creating buckets for different subsequences and putting same ones in the same bucket. So, sequences with the same hash end up in the same bucket.

2) **Clone Detection**
We iterate over each bucket and compare every pair of sub-trees, using the compareSequences function. Similarity = 2 x S / (2 x S + L + R) where: S = number of shared nodes, L = number of different nodes in sub-tree 1, R = number of different nodes in sub-tree 2. If the clone type we are looking for is 1, the result of compareTree should be 1, so that the two sequences are identical. if the clone type is 2, it should again be 1, so that they are identical, with the only difference that we ignore the leaves in the hash function. if the clone type is 3, the similarity threshold can be lower, e.g. 0.8, so that we also include in our results non identical but still similar pieces of code.

3) **Clone Gathering**
At this stage, we add the clones into the final struct, a list of tuples, consisting of two lists of nodes, that represent a clone pair of sequences. Before we do that though, we need to check if the current pair is a sub-clone of an existing pair in the struct. Because in that case, there is no reason in adding it, since we need the biggest sequence that is cloned. For the case that the flipped pair is already in the struct, we do not add it. Also, we need to check that there are not sub-clones of the current pair in the struct. If there are, they should be removed, because they are no more needed, for the same reason as above.
4) **Statistics Calculation**
Regarding the statistics we are calculating, the same applies as for the above algorithm. The results we are printing on the screen are:
- an example of a clone pair
- number of clone pairs
- number of clone classes
- biggest clone class in lines
- biggest clone class in members
- percentage of duplicated lines

These functions are the same as for the sub-tree clones algorithm, but they receive as argument the clones struct with the sequences.

4.3 Clone Generalization

We provide the possibility of generalizing both the subtree and the sequences algorithm. But that is basically only worth running, with the paper[2] version of subtree and sequences algorithms. Since our own version takes care of subsumption when adding pairs to the clones struct(described later in design decisions). When going through the AST in the subtree and sequence algorithms, we create a map that has as a key a node and as a value its children. Afterwards, in order to find the parent of a node we have to look through the map where the value contains this node. Of course a node can have many parents and we have to check for all of them, whether they form clone pairs. For subsequences, their parents are not only the initial sequences but also other subsequences that contain them. In that way we don't miss any clones.

Listing 2: Clone Generalization Aogorithm

```
{
    set[tuple[node, node]] clonesToGeneralize =
       clonePairs;
    while (size(clonesToGeneralize) != 0) {
        for (pair <- clonesToGeneralize) {
            println("<size(clonesToGeneralize)>");
            clonesToGeneralize -= pair;
            list[node] parents0 = parentsOf(pair[0],
               childrenOfParents);
            list[node] parents1 = parentsOf(pair[1],
               childrenOfParents);
            for (i <- parents0, j <- parents1) {
                if (i != j) {
                    if (compareTree(i, j,
                        massThreshold) >=
                        similarityThreshold) {
                        clonePairs -= pair;
                        addSubtreeClone(clonePairs, i,
                            j, massThreshold);
                        addSubtreeClone(
                           clonesToGeneralize, i, j,
                           massThreshold);
                    }
                }
            }
        }
    }
    return clonePairs;}
```

5 Design Decisions

5.1 Algorithms

At first we followed the Subtree and Sequence algorithms exactly as described in the paper, e.g. :

Listing 3: Subtree Clone Detection Algorithm pseudocode as given in the paper

```
1. Clones=\empty
2. For each subtree i:
    If mass(i)>=MassThreshold
    Then hash i to bucket
3. For each subtree i and j in the same bucket
    If CompareTree(i,j) > SimilarityThreshold
        Then { For each subtree s of i
                If IsMember(Clones,s)
                Then RemoveClonePair(Clones,s)
               For each subtree s of j
                If IsMember(Clones,s)
                Then RemoveClonePair(Clones,s)
               AddClonePair(Clones,i,j)
               }
```

However, we soon realized the issues it has:
1) This algorithm removes subclones when a new clone is about to be added to the clone struct. However, it does not check whether the pair that is being added is itself a subclone of an already existing pair in the struct. This way, with this algorithm we end up with many subclones and duplicates, which is obviously problematic. Nevertheless, we realized that that is why the Generalization algorithm, given in this paper, is useful. Because it mostly helps perform subsumption.

2) Using this algorithm as is in combination with the generalization algorithm is still not giving correct results. And that is because of the way the subclone removal is done. Based on the pseudocode, when a subtree of i is found in the clones, this clone pair should be removed, and then the same applies for j. However, in this way many clone pairs are excluded, without them being subclones of the pair we want to add. That is why we concluded that a pair is considered a subclone if and only if both members of the pair are subtrees of i and j, at the same time. This caused us a lot of confusion, because even the Generalization algorithm could not fix this issue, since many useful clones have been removed, which could not be of course retrieved by clones' parents, as the Generalization algorithm describes. An example with the sequence algorithm and minimum sequence of 3, to show what we are describing:

Listing 4: Example

```
1     public void f() {
2          if (true) {
3               int a=1;
4               int b=2;
5               int c=3;
6          }
7          int a=1;
8          int b=2;
9          int c=3;
10    }
11    public void g() {
12         if (true) {
13              int a=1;
14              int b=2;
15              int c=3;
16         }
17         int a=1;
18         int b=2;
19         int c=3;
20    }
```

Here, for clone type 1, there are 3 clone pairs. Obviously the biggest clone pair is the whole body of f and the whole body of g. However, there are still two more clone pairs, which should not be excluded, lines 3-5 with 7-9 and lines 13-15 with 17-19. There are not any other clone pairs between the two functions, because they would be subclones of the biggest clone pair we said in the beginning. However, the other two we mentioned are important clone pairs that we made sure we capture with our algorithm.

Listing 5: Improved Clone Detection Algorithm

```
1. Clones=\empty
2. For each subtree i:
    If mass(i)>=MassThreshold
    Then hash i to bucket
3. For each subtree i and j in the same bucket
    If CompareTree(i,j) > SimilarityThreshold Then
    For each OldClonePair in Clones
         If (i,j) or (j,i) are subset of OldClonePair
              Then ignore (i,j)
         If OldClonePair is subset of (i,j)  or (j,i)
            RemoveOldClonePair
    AddClonePair(Clones,i,j)
```

5.2 Choice of Data Types and Data Handling

We were able to calculate results for most of the algorithms, including Types 1, 2, and 3, in a reasonable amount of time due to our design decisions and optimizations.

- The mass threshold and minimum sequence length are parameterized, allowing us to run the clone detection algorithm with any threshold or minimum sequence length values.
- We utilized sets wherever feasible, as they outperform lists in speed and automatically prevent duplication.
- Avoiding traversing loops redundant times .e.g.
  for (i <- [0 .. size(hashTable[bucket]) - 1], j <- [i+1..size(hashTable[bucket])])
  This way we also avoided checking flipped pairs or pairs where i and j are the same.
- To conserve resources and avoid expensive computations, we preserved necessary values in a Hashtable using appropriate keys for the future use. For instance, we stored Parent-Children relationships and subtrees of a node in a hashmap, compareTree/compareSequences result in a map, so that we do not calculate it every time. We also used as key to the map a list that contains the pair that was compared. We used a list, so that the order of the pair does not matter.
- We have enhanced and streamlined the comparisons for greater efficiency, though some operations remain inherently costly. For example, verifying whether each clone is a subset of an existing clone requires the computation $size(buckets) * 2^{size(bucket)}$.

  Instead of the lengthy check:

  ```
  if(isSubTree(pair[0], i) or isSubTree(pair[1], j) or
  isSubTree(pair[0], j) or isSubTree(pair[1], i))
  ```

  We obtained the subtrees of i and j for a more rapid comparison:

  ```
  IJSubTrees = subTree(i) + subTree(j)
  if (pair[0] in IJSubTrees or pair[1] in IJSubTrees)
  ```

- Duplicate lines are identified by mapping the line numbers to a hashmap where the file name serves as the key, and the values are sets of lines. This approach not only avoids duplication but also accelerates the process of calculating duplicated lines. The same data is employed in visualizations for further analysis.
- For Type 2 and Type 3 clones, we normalized identifiers across the entire subtree by processing the entire AST of the project, which significantly saved time.

- Considering the project's size and the number of buckets, we refactored to have dedicated functions for Type 1 and Types 2-3 clones. This change allows for a single check instead of multiple checks for every element in a bucket, reducing costly operations. We modularized and reused the remaining operations.
- Caching isSubset result for the subclone removal part.
- Instead of performing nested visits, run them in parallel while storing the nodes in lists.
- Using unsetRec, either as is or by selecting which fields to exclude (e.g., unsetRec(n, "decl", "messages", "typ")), saves time during the visits.
- Adding break and continue statements wherever possible enhances efficiency.

6 Java Test Project

We created a very small and simple Java project, to test that everything works as expected. We wrote also some tests on it and we discuss the results in comments in this project.

7 Results

7.1 Project 'Smallsql'

Type	**Clone pairs**	**Clone classes**	**Biggest members**	**Biggest in line**	**Duplicate %**	**Duration (hh:mm:ss)**
I	815	815	2	11	10%	00:02:56
II	1127	1127	2	122	17%	00:02:43
III	943	943	2	122	15%	00:02:53

Table 1: SubTree Algorithm, massThreshold 6 with Generalization

Type	**Clone pairs**	**Clone classes**	**Biggest members**	**Biggest in line**	**Duplicate %**	**Duration (hh:mm:ss)**
I	22254	816	138	11	17%	00:09:59
II	119052	1270	356	122	35%	02:35:02
III	122762	1214	357	122	36%	02:50:15

Table 2: Improved version of SubTree Algorithm, massThreshold 6

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	59	59	2	21	2%	00:11:47

Table 3: SubSequence Algorithm, min. sequence length 6 with Generalisation

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	30	22	4	21	2%	00:00:08

Table 4: Improved version of SubSequence Algorithm, min. sequence length 6

7.2 Project 'Hsqldb'

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	9366	9366	2	55	19%	00:08:10

Table 5: SubTree Algorithm, massThreshold 6

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	6996	6996	2	55	19%	02:17:47
II	6607	6607	2	82	21%	02:46:50

Table 6: SubTree Algorithm, massThreshold 6 with Generalisation

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	82808	7199	104	55	28%	02:25:48

Table 7: Improved version of SubTree Algorithm, massThreshold 6

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	397	397	2	67	2%	00:14:28

Table 8: SubSequence Algorithm, minimum sequence length 6

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	230	152	6	67	2%	00:05:24

Table 9: Improved version of SubSequence Algorithm, min. sequence length 6

7.3 Test Project

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	3	3	2	10	59%	00:00:01
II	4	4	2	1	11%	00:00:01
III	4	4	2	1	11%	00:00:01

Table 10: SubTree Algorithm, massThreshold 4 with Generalization

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	8	8	2	10	59%	00:00:01
II	33	5	6	1	38%	00:00:01
III	33	5	6	1	38%	00:00:01

Table 11: Improved version of SubTree Algorithm, massThreshold 4

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	1	1	2	8	43%	00:00:01
II	2	2	2	8	43%	00:00:01
III	2	2	2	8	43%	00:00:01

Table 12: SubSequence Algorithm, minSequenceLength 3 with Generalization

Type	Clone pairs	Clone classes	Biggest members	Biggest in line	Duplicate %	Duration (hh:mm:ss)
I	3	3	2	8	43%	00:00:01
II	3	3	2	8	43%	00:00:01
III	3	3	2	8	43%	00:00:01

Table 13: Improved version of SubSequence Algorithm, minSequenceLength 3

8 Accuracy

The Sequence algorithm identifies fewer clone pairs than the Subtree algorithm because it targets sequences of at least 6 lines, whereas the Subtree algorithm can detect clones as small as one line. Type 2 clones are more than Type 1 clones, which makes sense, since identifiers are not taken into consideration. The paper version yields fewer clone pairs compared to our version, as it omits many potential clones that are not detectable even with generalization.

9 Scalability

A significant aspect of the project involved enhancing our program's scalability. We implemented our algorithms and tested them on projects of varying sizes: 'Test Project' (small), 'smallSQL' (medium), and 'hsqldb' (large) for Types 1, 2, and 3. The results substantiate our solution's scalability.

10 Comparison

We have developed six variants of algorithms for analyzing Type 1, Type 2, and Type 3 clones:

1. SubTree Algorithm, as detailed in a referenced paper [2].
2. SubTree Algorithm with Generalization.
3. An Improved version of the SubTree Algorithm.
4. SubSequence Algorithm, as outlined in paper [2].
5. SubSequence Algorithm with Generalization.
6. An Improved version of the SubSequence Algorithm.

The SubTree algorithms demonstrate superior speed and scalability, enabling us to execute and obtain results for larger projects like 'hsqldb' for Type 1, Type 2, and Type 3 clones, outperforming the SubSequence algorithms in these aspects. However, a limitation of the SubTree algorithms is their tendency to identify one-line clones, which are often not very useful. To address this, we have implemented visualization techniques that allow users to filter these results and take appropriate action. Adjusting the threshold to target larger clones can also enhance the utility of these algorithms. Our visualizations are particularly effective in overseeing larger clones and monitoring the frequency of their occurrence. The improved version of our SubTree algorithm delivers more reliable results compared to the version mentioned in [2], and it maintains efficacy even with generalization.

In contrast, the SubSequence algorithms are slower and less scalable for larger projects compared to the SubTree algorithms. Despite this, they tend to yield more useful results as the number of lines directly correlates with the minimum sequence length. A notable drawback is their occasional misclassification of completely different code, such as differing method calls, as clones. Our improved SubSequence algorithm version generates more accurate clones than the version cited in [2]. Although it is the fastest for Type 1 clones, it struggles to scale for Type 2 and 3 clones. Nevertheless, it can be effectively applied in larger projects by increasing the minimum sequence length, thereby enhancing its utility.

11 Examples of Clones

11.1 Type I Clone

Listing 6: "file:///src/smallsql/database/LongTreeList.java— Line: 331-341"

```
//there are more entries in this node
result |= (((long)nextEntry) << shift);
if(listEnum.stack>=3){
        listEnum.offsetStack[listEnum.stack] = offset;
        return result;
}
listEnum.offsetStack[listEnum.stack] = offset+
   pointerSize;
offset = getPointer();
shift -= 16;
listEnum.stack++;
listEnum.resultStack[listEnum.stack] = result;
```

Listing 7: "file:///src/smallsql/database/LongTreeList.java— Line: 376-386"

```
// there are more entries in this node
result |= (((long)nextEntry) << shift);
if(listEnum.stack>=3){
        listEnum.offsetStack[listEnum.stack] = offset;
        return result;
}
listEnum.offsetStack[listEnum.stack] = offset+
   pointerSize;
offset = getPointer();
shift -= 16;
listEnum.stack++;
listEnum.resultStack[listEnum.stack] = result;
```

11.2 Type II and III Clones

Example 1:

Listing 8: "file:///src/smallsql/database/IndexDescriptions.java—Line: 82-86"

```
private final void resize(int newSize){
    IndexDescription[] dataNew = new IndexDescription[
       newSize];
    System.arraycopy(data, 0, dataNew, 0, size);
    data = dataNew;
}
```

Listing 9: "file:///src/smallsql/database/Expressions.java— Line: 130-134"

```
private final void resize(int newSize){
    Expression[] dataNew = new Expression[newSize];
    System.arraycopy(data, 0, dataNew, 0, size);
    data = dataNew;
}
```

Example 2:

Listing 10: "file:///src/smallsql/database/SortedResult.java—(Line: 226-230)"

```
while(newRow++ < 0){
    if(!previous()){
        return false;
    }
}
```

Listing 11: "file:///src/smallsql/database/SortedResult.java—(Line: 245-249)

```
while(rows++ < 0){
    if(!previous()){
        return false;
    }
}
```

12 Data export

We exported the classes in textual form for TestProject, smallsql and Hsqldb projects for clone types 1,2,3 and for subtree, sequence and generalization algorithms.

13 Visualization

We experimented with two types of visualization, one using python to write results in Excel and one using charts in a React web application. Some of them are mostly useful for us to compare our algorithms but others are useful also for a developer to find the biggest clones and their locations and fix them.

13.1 Spreadsheet visualization

We have created a spreadsheet visualization of clones, where the clone class, number of duplicated lines, number of clones, and locations of clones (file and line numbers) are plotted in a readable format. It is important to understand that not all cloning is harmful [1], and sometimes it might be necessary. This

visualization can help users identify large clones, most frequently repeated clones and helps them to take appropriate action if necessary.

Clone Class	No. of lines	No. of clones	Clone 1	Clone 2	Clone 3	Clone 4	Clone 5	Clone 6	Cl
|java+com|	11	2	|java+compilationUnit:///src/smallsql/database/ExpressionFunctionRound						
|java+com|	11	2	|java+compilationUnit:///src/smallsql/database/IndexScrollStatus.java|(4						
|java+com|	10	2	|java+compilationUnit:///src/smallsql/junit/TestFunctions.java|(22524,3						
|java+com|	9	2	|java+compilationUnit:///src/smallsql/database/Utils.java|(11150,303,<3						
|java+com|	9	2	|java+compilationUnit:///src/smallsql/database/TableView.java|(3908,20						
|java+com|	9	2	|java+compilationUnit:///src/smallsql/database/SSCallableStatement.java						
|java+com|	8	8	|java+com|	|java+com|	|java+com|	|java+com|	|java+com|	|java+com|	|jav
|java+com|	8	2	|java+compilationUnit:///src/smallsql/database/Utils.java|(6330,321,<20						
|java+com|	7	2	|java+com|ilationUnit:///src/smallsql/junit/TestAlterTable2.java|(297,18						
|java+com|	7	2	|java+compilationUnit:///src/smallsql/junit/TestAlterTable2.java|(319,16						
|java+com|	7	2	|java+compilationUnit:///src/smallsql/database/Table.java|(13602,350,<3						
|java+com|	7	2	|java+compilationUnit:///src/smallsql/junit/TestJoins.java|(10508,180,<2						

Figure 2: Spreadsheet Visualization.

13.2 Web Page with useful charts

We created a simple React project that uses a json with our data. We used the Chart.js library and plotted graphs for Clone Pairs, clone classes, Biggest Clone Class, Biggest clone in lines, Duplicated number of lines found by different algorithms.

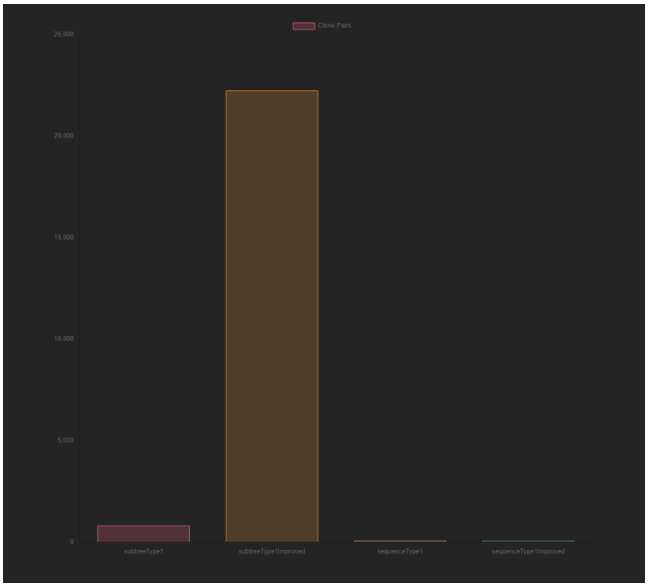


Figure 3: Spreadsheet Visualization

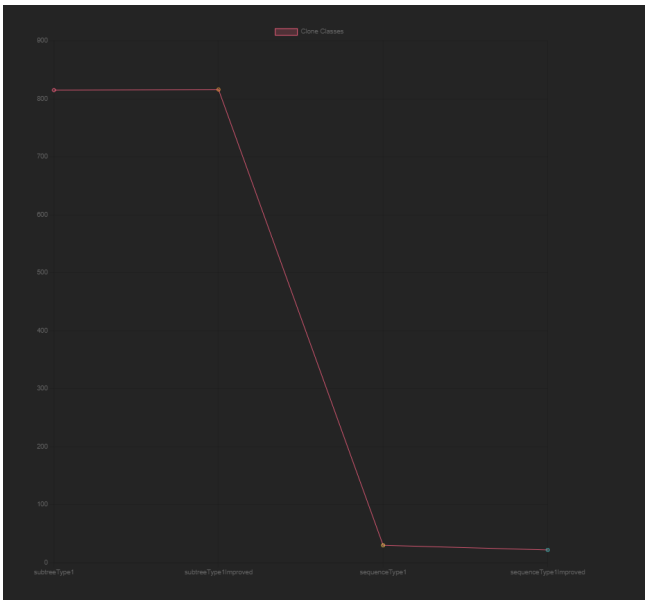


Figure 4: Number of clone classes found by different algorithms

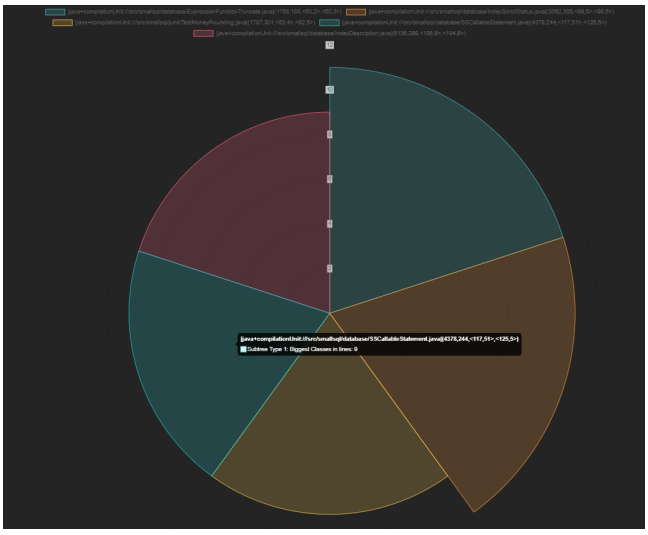


Figure 5: 5 Biggest Classes in lines found by the Subtree algorithm (paper version with generalization)

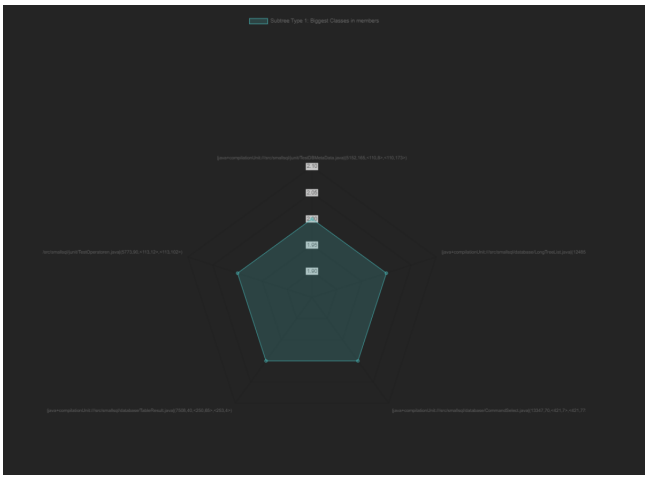


Figure 6: 5 Biggest Classes in members found by the Subtree algorithm (paper version with generalization)

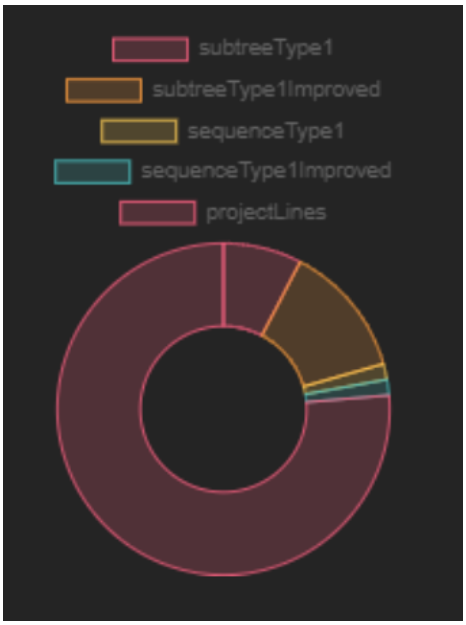


Figure 7: Number of duplicated lines found by different algorithms in comparison to the project lines. The chart can be hovered to see the exact numbers.

References

[1] KAPSER Cory. Cloning considered harmful'considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, 2006.

[2] Ira D. Baxter et al. *Clone Detection Using Abstract Syntax Trees*. 1998.

[3] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007.

[4] Tom Mens, Serge Demeyer, and Tom Mens. *Identifying and Removing Software Clones*. Springer, 2008.

[5] UseTheSource organization. The rascal meta programming language.

[6] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.