Series 2 Software Evolution

Mike van der Deure 14655837, Colin de Koning 13579991

December 2025

1 Introduction

For the second exercise of the Software evolution course, an algorithm was created that finds the type one clones and a visualization tool was created that include multiple visualizations to get insights in the clones. This exercise takes the front-end route. As an extra, type 2 clones are also detected using a similar algorithm to type 1 clones.

2 Clone detection

2.1 Explanation of the algorithm

2.1.1 Type 1 clones

The starting point of the exercise was finding all the type 1 clones. Type 1 clones are pieces of code that are entirely the same, besides white spaces and comments (Koschke, 2008). The algorithm is AST based, since the exercise required it. The starting point for finding all the type 1 clones was a pseudocode from Baxter et al. (1998). This pseudocode is shown below.

```
Algorithm:
1. Clones = None
2. For each subtree i:
     If mass(i)  MassThreshold
        hash i to bucket
3. For each subtree i and j in the same bucket:
     If CompareTree(i, j) > SimilarityThreshold:
        For each subtree s of i:
            If IsMember(Clones, s)
                RemoveClonePair(Clones, s)
        For each subtree s of j:
            If IsMember(Clones, s)
                RemoveClonePair(Clones, s)
        AddClonePair(Clones, i, j)
```

The base of this pseudo code is only taking the nodes that are larger than a certain mass, grouping them and then comparing those grouped nodes based on a similarity. The actual algorithm implemented is described below. Its a bit different from the pseudocode.

2.1.2 Actual algorithm

The code starts by ignoring all very small code nodes that are unlikely to be meaningful clones. this is done by measuring the mass, which is the total number of smaller elements (nodes) it contains. This is done in the calc_mass function. All the nodes that have a mass smaller than the threshold are ignored. The threshold is set on 25. According to Yang et al. (2015) setting a threshold of 30 is a common practice, mostly to save time. This algorithm had it on 25, close to 30 but a but lower, mostly in the hope to find a few more clones to test the visualizations. This is however a parameter to the function, so it's easy adjustable. Yang et al. (2015) mentioned that a threshold of 48 had almost the same result as 30, so it shouldn't make a big difference if it's 25 or 30. The algorithm was still good runnable at 25.

All the node higher than the threshold are seen as relevant and are grouped by their mass in a bucket. For example, all sub trees with 50 nodes go into the "50-Mass Bucket," all sub trees with 51 nodes go into the "51-Mass Bucket," and so on. This is done inside the find_clones_type1 function at the start.

Since Type-1 clones must be completely identical, only the identical subtrees within the same mass bucket have to be compared. If two segments have different masses, they cannot be Type-1 clones. However, before these nodes can be compared, first irrelevant information that intervenes in the comparison, has to be filtered. These are things like the file location, in Rascall called the src. his ensures that two segments are considered identical even if they are in different files. This can be found in the function strip_location.

From the next part the written algorithm is different from the shown pseudocode. The written ccode loops over all the buckets, checks the bucket size and if the size of the bucket is larger than 1, the bucket is looked at. If it has only 1 node, there is no possibility on any duplication or clones. Then there are checks on having a src and being more than 5 lines. If there is no src, the algorithm skips that node, as it can't do anything with it. Having more then 5 lines is based on the previous exercise, where for duplication 6 lines at a time was a good number of lines to check for duplication (Heitlager et al., 2007), as less than that number of lines can lead to irrelevant duplication to find. And as clones are about duplicated lines of code, it made sense to do use it here too.

If the node comes through the checks, it's put inside an exact bucket. This is a new map that will include all the nodes that come through the checks, and if a

node is the same as another node inside one of those buckets, it is placed inside the bucket with the same node. Before that can happen the sourc, in Rascall src, is stripped from the node, as otherwise two nodes that are the same, but on different locations, won't be seen as the same.

Then the algorithm loops over the exact buckets and if there are buckets larger than 1 item, that bucket is placed in a new list where all the clones are collected as a list. The bucket is placed in the list as a list, so clone classes, all clones with the same code, are kept together.

When this is done all the subnodes are removed to battle duplicated counting. When subnodes wouldn't be removed duplication percentages could be higher than 100 percent, which isn't possible. This is because a subnode of a clone, is partly the same code as its parent node. When not removed, those lines of code are counted twice or more.

2.1.3 Type 2 clones

Type 2 clones are nodes that are syntactically identical copies of each other (Koschke, 2008). This means that the code is the same, but the naming inside the code can be different. These clones are in this exercise detected on a similar way as the type 1 clones. They algorithm groups the same, and does everything else the same, until it comes at comparing. At that point there is an extra step to normalize the node. The node is changed, where all the names are changed to a constant name. This happens in the function normalize_node_to_type2. Then all the identical nodes for type 2 are gathered, just as well as all the nodes for type 1. Since the type 1 clones are not allowed to count. Then all the buckets of type 2 clones that have two or more items in it, which are not in type 1, are put in the output list. Again the children are removed and the output is given.

2.2 motivation

The reason the article called Clone Detection Using Abstract Syntax Trees (Baxter et al., 1998) was chosen as starting point, was that the exercise had to be AST based and this article describes a way to do it. At the same time is the paper cited a lot, which signals that it has been influential and well received by the research community. The high citation count also suggests that the paper is regarded as a reliable and authoritative source, making it a suitable and credible foundation for the base of this created algorithm in this exercise.

The reason the second part of the algorithm isn't in line with the pseudocode, is because another way had been seen, that was easier to program, and since for type 1 clone detection not all of the pseudo code was required, the choice had been made to use another route. A piece of the pseudocode that not necessary is for example the similarity. Since only type one and two clones are searched

for, its not relevant to calculate the similarity, when the current implementation looks only if two nodes are entirely the same.

2.3 Report created by algorithm

After the clones were found, they were written into another file. This made sense to do, to get a better overview into them, since it is easier readable then a written output in a terminal. There still is a written output in the terminal. This includes all the statistics about the complete project, but for the other things like example clone classes and number of clone lines per file, only the first six example were printed. The rest is only viewable in the report, to keep a better overview inside the terminal.

JSON has been chosen as format for the report. The JSON format is good readable and at the same time easy to put in Python. Python was used later on to make the visualizations. The written report exists of number of total lines in project, number of duplicated lines, which can also be seen as number of lines inside clones, duplicate percentage, number of found clones, number of clone classes, the number of lines of the biggest clones and all the examples of clones per class. The number of lines ignore white spaces and comments, as it is found with ASTs. For every file in the targeted project, the number of lines and the number of clones lines, as well as the last line number of every file is also given in the report. This information was necessary for the chosen visualizations. The line number was necessary for the dot plots to know how many line number should be plotted per file. The number of cloned lines and normal lines were necessary to calculate the clone percentage and to put in the tree maps. These visualizations are described later.

All the statistics that are put in the written report are insightful for clones or are used in the visualizations. For example the reason why also all the duplicated lines per file were given, is that it creates to make the tree maps in the way that is later described.

3 Visualizations

All the visualizations are made using the information of the JSON report. No other information is used, only what can be found in the report and that is a result of the Rascall code. This report is imported in python and transformed to a dictionary.

The visualizations are made in Python. The choice for python comes from a familiarity with the programming language. Besides that, python has a lot of good libraries for visualizations, like Matplotlib, Seaborn and Plotly. Also are there some libraries to help with for interaction, like the earlier named Plotly, but also Ipywidgets. The visualizations itself are made in Jupyter Notebook,

as its easy to interact with them. When the code block of the visualization is runned, the user can directly interact with it.

All the visualizations are interactive by having the option to pick the project. This is done, so all the reviewed projects can be shown. But at the same time, not at the same time. Since it is not really relevant to compare different projects with each other, as they are not related to each other. Having more than one project visualized at the same time, would only broaden the focus and make the plots too zoomed out, more messy and too unclear. Making this is done with ipywidgets library in python, so that only a few lines have to be added to add a new database. It works by having a choice parameter in the function, which sets at the start of the function which project the data should project. The user can select the preferred project on the left upper corner.
The types of visualizations are chosen based on giving an overview of clear information that is not easy readable in number form. For example, when someone want to know the number of clones in a project fo the duplication percentage of code, it would be easier and clearer to give only the number. While if someone want it per file, a visualization would be a good option, since its easier readable at a single glance. This is also the reason all the visualizations made here and described below, are using per file as a base.

3.1 Heatmap

The first visualization chosen is a heatmap where all the files that have clones are on both the x and y axis. The goal of this visualization is to give an insight in where the clones are inside a project. In this case in what files. Since there is no use in comparing two different projects, the visualization is made so that inside the visualization, the user can choose the project. Another interaction included in the visualization is zooming. When a project is bigger and includes more files that have clones, like the hsqldb project, the data can become to large to visualize good and the visualization can become too unclear. That's why the user has the ability to zoom in and out based on what the user want to see. This visualization isn't based on a paper, but it is useful in combination with the dot plots, which are based on a paper. This is explained in the text about the dot plots. An example of the visualization is given in figure 1.

3.2 Dot plots

The second visualization that is made is a dot plot. The plot is from the paper from Koschke (2008). The function of the dot plots, which are in the paper also scatter plots called, is to show the duplication lines of two units. According to the same paper has the visualization the problem of scalability.

With the dot plots made in this exercise, the visualization shows where the clones are. The dots are placed on the line number of the file, so the user knows

directly where to search for the clones. This means that all the lines are in the plot, also the empty lines and comments. Even though, these lines are irrelevant for clones, it gives more clarity for the user. When they would be ignored, the numbers of the lines would be off and it would be unclear where the clones really are. So in contrast to the way of Koschke (2008), not all similarity is shown, but only real clones between two files. This gives insight in where the clones are. Also only two files can be compared, not other units. MAYBE EXPLAIN! The plot works by having a file on the x-axis and a file on the y-axis, where for both all the lines are plotted out on the axis. On the lines where there are clones between those two files are dots plotted on the lines from both files.

The visualization is made so the user can pick the two files, both for the x-axis and the y-axis, using a drop down menu on the top left. This has the downside that it can become a bit unclear, just like Koschke (2008) has pointed out by saying its difficult to scale. Especially since not all files have clones with each other. To combat this limitation, several measures have been implemented. First is to show the user only files that have clones in the drop menu. It is useless to have files without clones inside the drop menu, as it give no insight in where the clones are. Second is to not show the plot for two files that have no clones with each other. Instead of a plot, a printed message is shown, that says that there are no clones for these combination of files. So can the user directly see that the current option combination isn't useful and go on to the next. The last thing is the heatmap. As discussed in the text of the heatmap, the heatmap can be used in combination with the dot plots. The user can see what combination of files have clones inside the heatmap, and then search those files in the drop down menu in the dot plots, to see on what lines in the files those clones are.

Since the files, which are compared, can sometimes be quite large, there is also a zoom option in the visualization. The user can zoom using the mouse or touch pad, by selecting what he wants to see, to see the dots of the clones from closer and to be able to read better the line numbers on which the clones are.

3.3 Tree maps

The third visualization that is made is a tree map. A tree map is a visualization that has a space filling approach in visualizing hierarchical data (Johnson & Shneiderman, 1991). Koschke (2008) also discusses this vizualization for clone representation. Here have the leaves sizes, which represent the number of clone lines. the hierarchy is the hierarchy in the folders of where the files are. So the map shows in what directory the different files are.

The goal of the implemented tree map is to show another perspective on where the clones are inside the directory. Besides just showing where the clones are, like the heatmap does, does the tree map show the amount of cloned lines per file, through the size of all the blocks. The tree maps also show the percentage

of cloned lines out of the total lines per file, by the color of the blocks. This can be seen by the shade of red a block has, which is also shown in the legend on the right of the tree map. The hierarchically nature of the tree maps show where in the project the files are, so the user knows where to search for them.

In this visualization, the software system is structured hierarchically ('Project' -¿ 'Module' -¿'File'). Each rectangle represents a source file. The area of a rectangle corresponds to the number of Type-1 cloned lines, making files with a high absolute clone volume immediately visible. In addition, color intensity encodes the relative clone density, i.e., the percentage of cloned lines compared to the total size of the file. Darker colors indicate files where cloning has a stronger impact.

By hovering over the blocks, the user can see the exact amount of lines of the file, the number of clone lines and percentage of cloned lines on the total of the file, rounded to 2 decimals. The user can also select a specific directory in the tree map, by clicking on it, to enlarge the view of the files in that directory. It can go back to the complete overview by clicking on it again.

Reflection

3.4 Clone detection

The type 1 clone detection that was written in rascal is tested manually with test files that are also submitted. The clone detection for type 1 works for those test files. The duplication percentage in the the projects smallsql and hsqldb appear to be a bit low. But since the code detection only looks at type 1, which means pieces of complete the same code, besides white lines and comments, the number are not entirely unrealistic. The duplication percentage of the hsqldb is higher than the same metric for smallsql. This makes sense as with a bigger project, there is more code that could potentially be a clone. For number of duplicated lines for type two detection, when the type 1 clones are not subtracted, are higher than the type 1 clones. This isn't submitted in the reports, but manually tested.

It would have been better to have used more tests, like unit tests, as its now only tested manually, with the test files, and a bit with the visualizations. With more time this would have been done.

Looking back, the clone detection algorithm would probably be more robust when it would follow the pseudocode more. Not that it would work better for type one clone and, but it would be easier to go further with the same code to create a algorithm that also detects other type of clones, like type three clones. For example, type 3 clones need a similarity calculation like in the earlier named pseudocode also used.

3.5 Visualizations

The visualizations try to keep all the information clear and to keep a good overview of everything. At some points, there is still some risk at unclearity. For example, the tree maps are with the current JSON reports very clear. However, if there would have been a project with way more files with clones, this would be less the case. The visualization would still be useful as it can zoom into submaps, but the first overview would be a bit messy. The same is the case for the heatmap. For the dot plots the same counts, but that is a known thing for the plots as written earlier and there are measurements to combat that quite well.

3.5.1 Possible improvements

There are some small point the algorithms are lacking in. For example, it would be better to have the files in the tree map that are in the project root, directly in the project root, instead of a piece called project root. This would signal better where those files are, but the library didn't let us to that. Another improvement could be in the dot plots. There are currently a lot of pairs of files that can be picked, that have no clones. For a better user experience, it would be better to solve that instead of relying on checking the heatmap every time. But the current implementation works as well. A last thing would also be at the dot plots. That is that all the duplicated lines are signaled in the dot plots. This is how the dot plots were intended according to the cited paper in the explanation of the dot plots. However, since this exercise was specifically about the clones and this would be a whole other implementation, the choice has been made to only visualize the clones. This decision was mostly time based.

3.5.2 Requirements

The plots all have their own use. The dot plot gives information in where inside the file the clones exist, so the maintainer can find the clones and delete, adjust or keep them as he wishes. The heatmap is useful as support for the dot plots. It also gives insight in what files have clones. The tree maps are there as a hierarchical overview with density of clones. The density and number of cloned lines could signal what files are possibly redundant and what files have a priority to change. The same count per directory.

There are some possible requirements a maintainer could currently be missing with the current tool. For example, to get a even more complete overview, the current visualizations misses a figure that informs about clone classes. This would also have been useful for the user or maintainer to get an insight in what code is more often cloned. Then could the user pay attention to it with writing new code or to put that code in a function somewhere central. Such a visualization would be a good improvement to the current tool. Another improvement would be to put the visualization in some sort of software, so the

user doesn't need to run it manually, eventhough it could be expected that a maintainer or user fo such a tool would be capable of using a jupyter notebook.

References

Baxter, I., Yahin, A., de Moura, L., Sant'Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. *Proc. of International Conference on Software Maintenance*, *368-377*, 368–377. https://doi.org/10.1109/ICSM.1998.738528

Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, 30–39. https://doi.org/10.1109/QUATIC.2007.8

Johnson, B., & Shneiderman, B. (1991). Tree-maps: A space-filling approach to the visualization of hierarchical information structures. *Proceeding Visualization '91*, 284–291. https://doi.org/10.1109/VISUAL.1991.175815

Koschke, R. (2008). Identifying and removing software clones. https://doi.org/10.1007/978-3-540-76440-3_2

Yang, J., Hotta, K., Higo, Y., Igaki, H., & Kusumoto, S. (2015). Classification model for code clones based on machine learning. *Empirical Software Engineering*, *20*(4), 1095–1125.