Visual Analysis of Source Code Similarities

Michael Burch, Julian Strotzer, and Daniel Weiskopf
VISUS, University of Stuttgart
Stuttgart, Germany
Email: {michael.burch, julian.strotzer, daniel.weiskopf}@visus.uni-stuttgart.de

***Abstract*—Software systems typically consist of many lines of source code organized in several files hierarchically structured into directories and packages. Since the code is the key data in software development, in many scenarios an overview of it is required, in particular for similar code passages. In this paper, we investigate the visual analysis of source code similarities for local as well as global code passages. To this end, we first compute all subsequence occurrence frequencies (support metric) and relative occurrence frequencies (confidence metric) in local as well as global code regions. The resulting textual data attached by its occurrence values is displayed in a triangular matrix. Several interaction techniques are integrated in our visualization tool which are illustrated in the corresponding case study illustrating similarities in source code written in Assembler consisting of 10,641 characters.**

I. Introduction

Today's software systems are large and typically developed over time spans of several years by many developers. This process produces vast amounts of data of various types. The key data in software development is the source code itself since it is responsible for a well-designed and properly running software in the end. Analyzing this textual data is a challenging task that may be supported by visualization techniques benefiting from the perceptual abilities of the human viewers due to the strengths of their visual system and fast pattern recognition.

To understand code, we usually have to view it directly. However, if someone is just interested in quickly finding similarities in large amounts of code globally as well as locally, text scrolling interaction alone is very time-consuming and not useful any more. An algorithmic solution is required that is able to uncover similar code regions; but the output data of such an algorithm is already that large that an exploration of it on a pure textual basis is again very difficult.

In many scenarios, we wish to first obtain an overview of similar code fragments and their regions of occurrence. To achieve this goal, we propose an approach that consists of two separate components: a source code similarity analyzer and a visualization that shows the similarity results together with the computed occurrence frequency values and the regions of those occurrences by either a pixel-based approach or by overplotting the display with color-coded most frequently occurring textual code fragments.

Generally, there are important questions to a software developer concerning source code similarities, focusing on either the complete code or parts of it. The following questions can be answered by our visual analysis technique:

- **Code regions:** Which regions in the source code contain many similar textual patterns? Are those most frequent code fragments occurring more locally, more globally, or both? Are there any empty regions?
- **Code fragment lengths:** What are the lengths (number of characters) of the frequently occurring code fragments? Are those equally distributed or are there regions containing longer frequently occurring code fragments while others contain shorter ones?
- **Occurrence frequencies:** If code fragments occur frequently, what are the occurrence frequency and relative occurrence frequency values? Are there any regions with similar values?
- **Similarity distributions:** How are the most frequently occurring code fragments distributed? Are there any differences in the regions or also between different code fragment lengths?
- **Code comparisons:** If there are several code files, are the visual patterns for the code fragment frequencies similar or do those behave totally different? Are there any differences or commonalities among many subsequent revisions of the same code fragment?

We illustrate the usefulness of the technique and its integrated interaction techniques by applying it to source code written in Assembler that contains 10,641 characters.

II. Related Work

When analyzing software systems, their structure, behavior, and evolution are of importance [1]. However, regardless of what is analyzed, source code can be considered as the key data for static software and also during software development. While the complete system is evolving, programmers are constantly changing code by adding, deleting, or replacing code passages. In some scenarios, it also comes to a more or less copy-and-paste behavior during implementation in which already implemented code is duplicated and added to a totally different part of the project or sometimes even in the same source code file. Such code clones [2] are of special interest for software development since those are problematic in further development and for reliably maintaining a growing project.

Code clone detection in general and the visualization of its results is a difficult discipline and has been researched intensively [3], [4]. The above mentioned scenario of copy and paste is the easiest one to uncover because the code is not changed but just copied as it is. More complex scenarios occur when the code is consistently changed, for example,

when variables are renamed or functionality is changed while still semantically doing the same as before. Such source code similarities are not in the focus of our present work, but instead, we are more interested in visually presenting all code fragments locally as well as globally which are direct multiple copies from other code regions. Such an overview is important to see where possible similarities occur in the code that then have to be analyzed further.

There is already some research on presenting source code by first providing an overview. The SeeSoft tool [5] for example, uses a line-based color-coded visualization of the code lines still showing the pretty printed structure [6], [7] of the code and additional information such as the age of the code. Lommerse et al. [8] propose a visual code navigator in which three views are provided showing large source code of software projects from three different perspectives: a symbol view, a syntactic view, and an evolution view. What is missing in this tool is a view showing source code similarities.

Beck et al. [9] visually augment source code by additional information such as performance data or they [10] use word-sized graphics for monitoring numeric variables. Although this integrates additional views on this textual data, it is still difficult to visually analyze the code for pure textual similarities. Moreover, this additional information requires a certain amount of display space and makes it problematic to show larger code passages as an overview, which we refer to as visual scalability issues. The analyst still has to interactively scroll the code to visually compare the values, but as a benefit the code is shown in its original form as pretty printed text.

Changes done to source code during its evolution are also analyzed and visualized. For example, Voinea et al. [11], [12] use a line-based representation to show the evolving code structures. However, in their work they try to give a global view on the code changes but do not consider the visual analysis of code similarities for a single version of it. Also the CodeFlows visualization [13] is based on showing source code evolution, but on a more structural basis.

In the field of bioinformatics, research on text sequence comparisons is of special interest, allowing one to identify common subsequences of DNA strands for example. Multiple sequence alignment techniques [14] are used to find out common substructures that are then visualized as color-coded lines of differently large sizes [15]. However, in this approach, several DNA strand subsequences are not compared several times with all the others, which is required in our work. The dot plot matrix [16] is a popular visualization technique using color-coded pixels to indicate where two sequences contain the same nucleotides. This technique is powerful, but as a drawback only single characters are compared and not differently long substrings. Moreover, occurrence frequencies and relative occurrence frequencies [17], [18] cannot be displayed.

III. Data Model and Transformation

In this work, we deal with textual data, i.e., source code generated by using any programming language. In this section, we first mathematically model this kind of data and then show how the data is transformed into a list of code fragments, code regions, and attached frequency values.

A. Source Code

Source code is a finite sequence of $n \in \mathbb{N}$ characters

$$S := \{\sigma_1, \ldots, \sigma_n\}$$

where each σ_i is an element of a finite alphabet Σ, i.e., $\sigma_i \in \Sigma, \forall 1 \leq i \leq n$. In the context of this work, we do not treat special characters such as 'empty space' or 'carriage return' differently from the others but we, instead, just read all characters in the same way, i.e., source code builds one long sequence of elements that is taken as input for our preprocessing algorithm.

In the visual representation of source code, some characters are already treated differently, e.g., source code is not written in one single line but starts in separate lines in order to visually represent the code structure more clearly. This concept is often referred to as pretty printing [6], [7] in programming which typically illustrates a hierarchical organization [19], [20]. Our work is freed from those structural aspects and hence, can easily be applied to any kind of code, even if those pretty printing rules are not used during implementing code.

B. Code Fragment Frequencies

Before starting the visualization tool, source code has to be preprocessed first which is then stored internally by the data analysis algorithm. This means, the source code is only analyzed once, making the visualization technique interactive.

The data analysis algorithm takes the sequence of characters S as input and first computes a list of all possible occurring substrings in the given sequence S. From the precomputed list, occurrence regions are computed by calculating start and end indices of the found regions. In the same algorithm loop, the occurrence frequencies are computed, which are later divided by the number of characters contained in each region, allowing us to compute the relative occurrence frequencies.

The final output of the preprocessing algorithm is a list of $k \in \mathbb{N}$ subsequences $L := \{S_1, \ldots, S_k\}$, where each subsequence S_i is attached by another list L_i containing the start and end indices of the corresponding occurrence regions together with both values for the occurrence frequencies in these regions:

$$L_i := \{I_1, \ldots, I_{n_i}\}$$

where each information quadruple can be described as

$$I_j := (s_j, e_j, f_{j_{all}}, f_{j_{rel}})$$

in which s_j expresses the start index of that code region, e_j the end index, $f_{j_{all}}$ the occurrence frequency (support), and $f_{j_{rel}}$ the relative occurrence frequency (confidence). The support metric values are computed by counting the number of word occurrences in that region, whereas the confidence metric values can be obtained by the support divided by the maximal number of possible occurrences of that substring in that region.

IV. Visualization Technique

The visual analysis of source code data comes in two separate steps. The preprocessing step consists of the text comparisons while simultaneously computing occurrence frequencies and relative occurrence frequencies, which is described in Section III. The preprocessed output data is then used as input for our interactive visualization tool; here, we still preserve a link to the original raw source code data to later facilitate details-on-demand and a direct linking of the visualization of the preprocessed data to the source code.

A. Visual Encoding of Source Code

In our approach, we need to visualize the code in a line-based representation because we need one single representative element on a horizontal line for intuitively attaching the triangular matrix with the additional information about the occurrence regions as well as the occurrence frequencies.

This simple visualization of the source code line can be seen as color-coded horizontal line on top of each triangular matrix. Each character is visually mapped to a unique color from a predefined color scale, i.e., it is treated as categorical data. The color coding is one straightforward way to visually encode similar code regions; however, due to perceptual problems and the fact that only a limited number of color hues can be separated by the human eye and the visual system [21], we have to add the triangular matrix to achieve a better representation of the similarities.

The visual representation can be used as interaction support when selecting code regions for filtering and zooming the triangular matrix. Moreover, the selected code region can be represented as original view on the source code in its pretty printed form. Brushing-and-linking features between the code and the similarity matrix and vice versa are possible, which is helpful for putting the visualization in context to the real code.

B. Triangular Matrix

The visual design of our approach uses a triangular matrix to encode support and confidence metric values for each substring and its similarities in several regions in a source code fragment—similar to the approach used in the saccade plots for displaying eye movement saccades [22]. Such a scenario is illustrated in Figure 2 for a larger example with real code. Here, we can see source code represented as a color-coded compressed line on the top which is used as basis for similarity analyses. The font sizes visually represent the support metric values of the text fragments under observation (in this case, ‘add’, ‘and’, and ‘sub’), whereas the color coding gives a hint about the confidence metric values.

The visual design of our approach is illustrated in Figure 1; the design can be described as follows:

- By placing a representative substring in the triangle, a corresponding subtriangular region is spanned that indicates the explored region in the source code.

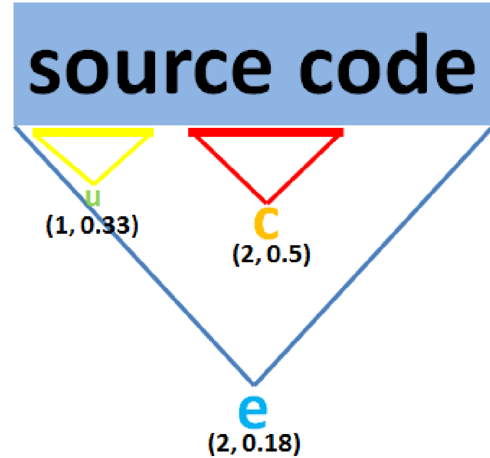


Fig. 1. An illustrative example of our approach showing similarities of substrings in the text ‘source code’.

- Gray-colored guiding lines can be added to the diagram to support the user to understand the region under exploration.
- We place substrings with higher confidence values on top of the others because the focus is on the relevant substrings.
- We visually encode one metric value in the color of the text labels, while the other is mapped to the font sizes.

The triangles are unique representations for each region, which can intuitively be used to interpret the visualization. The text-based diagram is similar to word clouds, but the position of each text fragment is fixed in this representation. In standard word clouds, a sophisticated layout algorithm can be used to produce a clutter-free visualization. It may be noted that color coding and font size can be exchanged interactively and it can be switched off completely.

C. Interaction Techniques

Although we provide the viewer with an overview first, our visualization technique benefits from several important interaction features with which one can easily manipulate the views and browse in the data. This supports deriving visual patterns that must again be remapped to the original data in order to derive meaningful insights. Our tool is designed in a way following the principles postulated in the Visual Information Seeking Mantra by Shneiderman [23]: Overview first, zoom and filter, then details on demand. For the interaction techniques, we were inspired by the work by Yi et al. [24], in which seven categories of interaction principles are described together with specific application examples.

- **Select — mark something as interesting:** The viewer is able to highlight substrings as being interesting for future investigations. This principle helps keep track of specific substrings while the view is changed. The highlighted substrings can still be perceived even if the view has changed and consequently support a better analysis of contextual information.

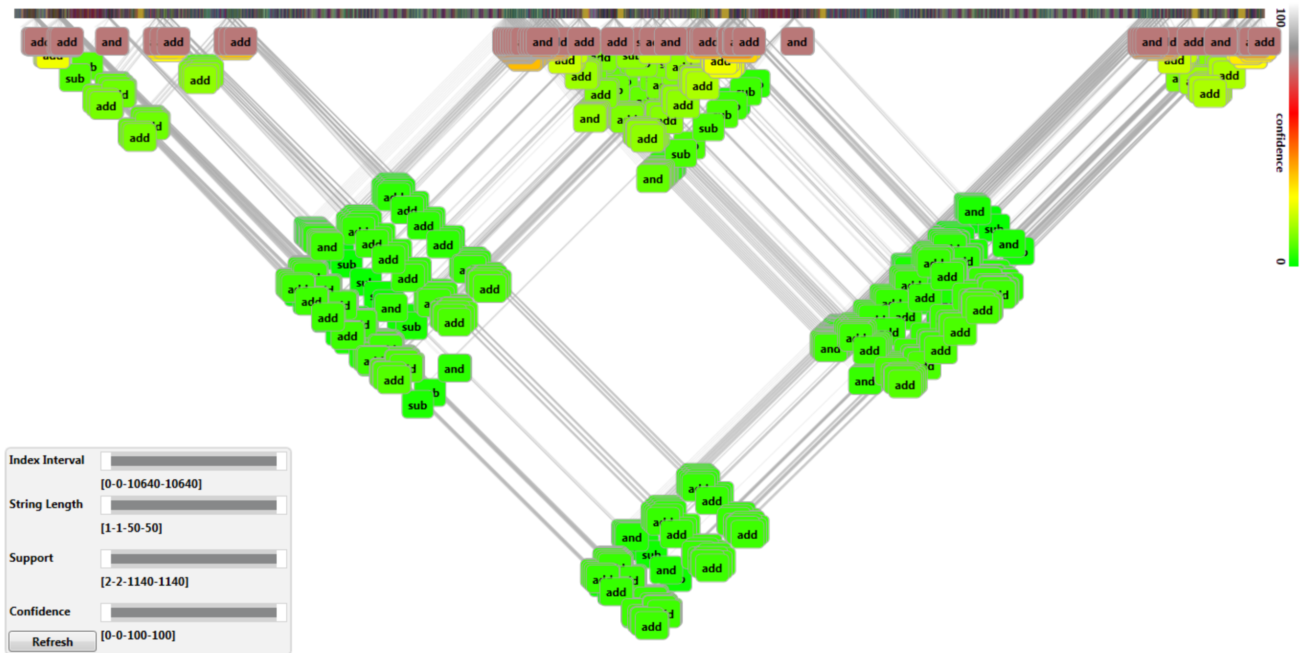


Fig. 2. A source code fragment with several thousand characters. Searching for occurrence frequencies of the words ‘add’, ‘and’, and ‘sub’ results in color-coded cluster-like word regions similar to word clouds.

- **Explore — show something else:** In particular, for local regions the explore interaction is of interest. One pans to another local region to see if there are similar visual patterns. Also parameter/slider changes have an impact on the view and may add additional substrings or some others disappear.
- **Reconfigure — show a different arrangement:** In the initial triangular matrix, all substrings are visually represented at the apex of the triangle that spans the analyzed subregion of the source code. Guiding lines can be added to support finding the spanned region in the code. The viewer is able to rearrange the positions of the displayed substrings by using mouse drag-and-drop operations. This would lead to misinterpretations when the triangle guiding lines would not be shown any more but also unclutters the display.
- **Encode — show a different representation:** A different representation of the data can help the user see the data from a different perspective. This is supported by either using a color-coded pixel-based representation or by showing the complete substring texts centered to the respective triangle apex. Also a view on the original pretty printed source code can be requested using the same color coding for the visualized code fragments.
- **Abstract/Elaborate — show more or less detail:** The user can zoom into local regions. Then, the focus is put on the shorter code fragment, which helps the user see more details about this specific piece of code. Overview and detail are supported by showing a large triangle for the detail view and a smaller one for the overview in which the detailed one is highlighted.
- **Filter — show something conditionally:** Filter functions can be applied to either the text lengths, the occurrence frequencies, or the relative occurrence frequencies. Also a filter function for specific text patterns can be applied, leading to a representation where only the substrings are displayed containing the filtered text. This can be done in the data preprocessing step and also interactively in the visualization. The filtered out substrings are either removed from the display or shown as grayed out texts for contextual information.
- **Connect — show related items:** Multiple views can be displayed, for example smaller detail triangles and a larger context with highlighted details, while brushing and linking between both can be applied. Real source code can be displayed in its original form when a source code fragment is selected.

V. Case Study

To illustrate the usefulness of our source code similarity visualization we applied it to source code written in Assembler, a low-level programming language for a computer. To illustrate the algorithmic and visual scalability of our approach, the source code is first preprocessed and the final output of this algorithm is visualized as color-coded triangle representations. For this reason, we analyzed source code consisting of 10,641 characters. We first compare the code fragments for similarities and show the output data. Since the resulting figure contains very many characters of length one, two, and three, we decide to filter out those from the beginning.

Figure 3 illustrates all code fragments of lengths between 5 and 8 which occur at least twice in the Assembler code. In

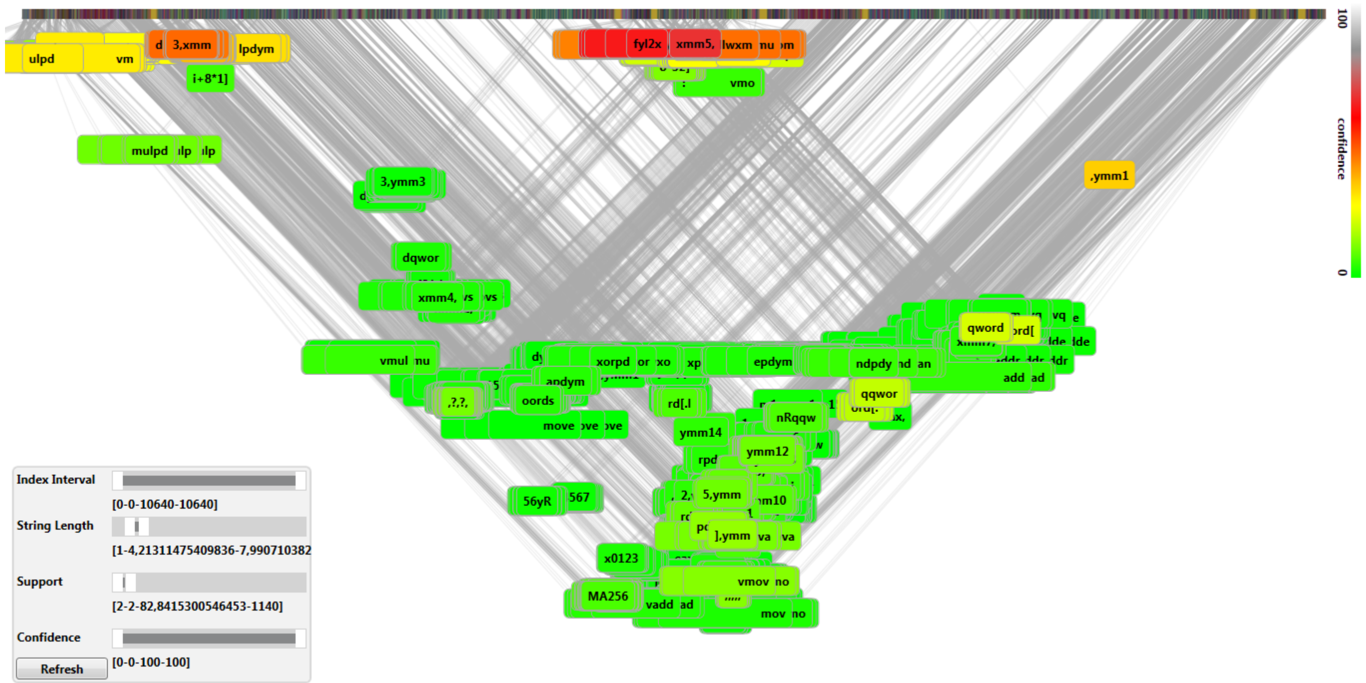


Fig. 3. Showing all code fragments of lengths between 5 and 8 occurring at least twice in the Assembler code containing 10,641 characters in total. The occurrence frequency is filtered for values between 2 and 82, where 1,140 is the maximum value. There is no restriction on relative occurrence frequencies.

the figure, we see that there are many empty regions in the diagram. But globally there are many similar words, which can be seen by the cluster of words close to the apex of the largest triangle. Those are colored green, indicating that their relative occurrence frequencies are not that large.

We can also see an outlier in this visualization. The code fragment ',ymm1' occurs very frequently in the rightmost larger code region. By the orange color coding, we can derive that it has a higher relative occurrence frequency than many other words. Moreover, it is the only frequently occurring word in this region, which is a strange phenomenon. Looking in the corresponding source code, we see that several variables are used, while many of them starting with ',ymm1' followed by another number, which leads to a higher frequency of the prefix ',ymm1' than for all the others. The word ',ymm' would occur more frequently but is filtered out by limiting the code fragment lengths between 5 and 8.

Another interesting fragment is 'qqword', which we already saw in the overview representation. The distribution of the occurrence frequencies of that code fragment is illustrated in Figure 4 (a) as a textual representation. What is also interesting here are the empty regions and the increasing values of the relative occurrence frequencies to the apex of the largest triangle.

There is also a clear structuring into two larger regions which are indicated by the two triangles close to the source code line. The larger square gives a hint about the overlapping regions.

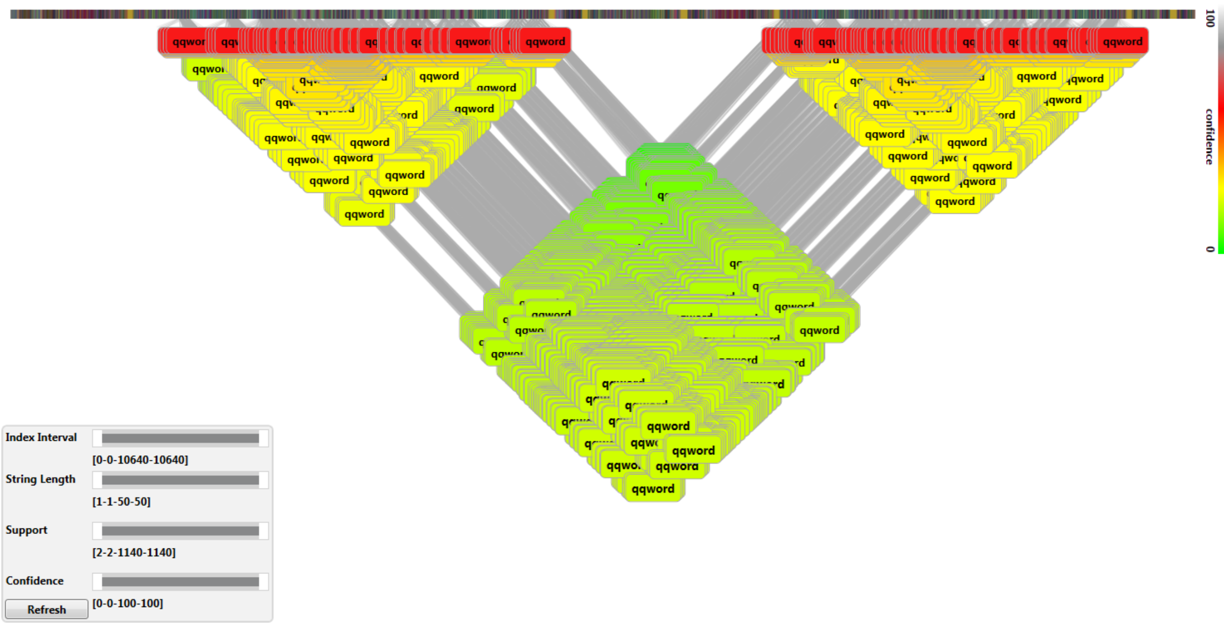
Requesting two words such as 'qqword' and 'sub' shows that both seem to be located in totally different regions in the code, while the code fragment 'sub' occurs less frequently than the fragment 'qqword', which can be seen in Figure 4 (b). The corresponding pixel-based and overlap-free representations can only serve as an overview but have to be changed to a text-based representation in order to visually distinguish the code fragment occurrence frequencies for more than one code fragment.

Selecting a code fragment opens a source code viewer showing the original form where the same color coding as in the triangle view is applied. This helps directly link the found observations in the visualization to the raw data, i.e., to the source code.

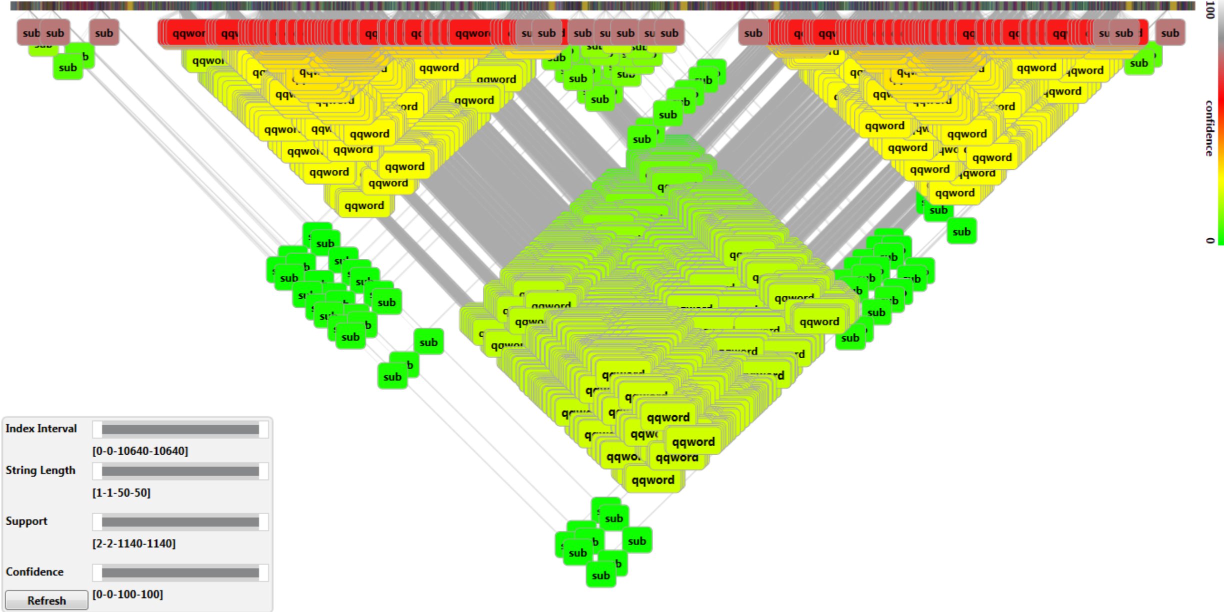
VI. Discussion and Limitations

Although we presented an intuitive visualization technique for source code similarities on a pure text comparison-based approach, we are aware of the fact that there are also various limitations of the approach which will be discussed in the following:

- **Algorithmic scalability:** The data preprocessing algorithm can be very time-consuming, depending on the length of the source code to be analyzed and also on the number of similar code fragments. This process is provided as a separate component allowing us to first preprocess source code and then independently inspect the output of the algorithm. This makes the visualization technique interactive because we do not need to algorithmically transform the original source code again but can look up requested information in the already generated output data.



(a)



(b)

Fig. 4. Filtering the source code for a specific word and visually inspecting the word similarities in the code: (a) The code fragment ‘qqword’. (b) Requesting two code fragments ‘qqword’ and ‘sub’ simultaneously.

- **Visual scalability:** The output of the similarity algorithm is already that large that much visual clutter caused by overplotting and occlusion can occur in the overview representation. Consequently, for some source code examples the number of generated similar code fragments can be too large to be analyzed directly in such an overview, which demands to already filter the preprocessed data before inspecting it visually.
- **Programming language independency:** Our similarity algorithm is totally independent of the programming language, which is advantageous. However, if someone is interested in seeing the results of more sophisticated code clone detection algorithms, we are aware of the fact that the programming language has an impact on the preprocessing algorithm.
- **Data aggregation:** If problems with visual scalability occur, one can apply aggregation techniques such as summing up values. This can also be applied in our visualization technique, leading to a more uncluttered display; however, as a drawback, the displayed values

cannot be interpreted that easily any more. Aggregation techniques should be applied after first having inspected the probably cluttered and overdrawn original generated data of the similarity algorithm.

- **Source code/software system structure:** We display the code as a line-based representation, which makes the code unreadable and not interpretable. This is due to having a representative element for each character in the code on a horizontal one-dimensional line. It may be beneficial to also visualize the source code structure (pretty printed form) or even the hierarchical organization of the complete software system. This might additionally provide a tool for navigating in the visualization, e.g., by collapsing or expanding source code.

VII. Conclusion and Future Work

In this paper, we introduced an overview-based visualization technique for visually analyzing textual data such as source code. Since source code can become very long, obtaining an overview of it is difficult and scrolling cannot really support a viewer or software developer to quickly get insights into this kind of data. This is particularly the case when the code must be analyzed for similarities, i.e., if regions in the code are of interest in which code is similar. Our visualization makes use of a triangular matrix representation and visually encodes occurrence frequencies and relative frequencies locally as well as globally as color-coded pixels that can also be overlaid by detailed textual information although it clutters the display. Interaction techniques are integrated, allowing one to adjust the occurrence frequency and relative frequency display intervals and the substring length of interest.

For future work, we plan to add further interaction features to the already existing palette in the tool. Moreover, we plan to apply the visualization to source code from different programming languages to see if those show similar or different visual patterns. The dynamics of source code might also be of interest, i.e., it may be useful to see how the similarities in source code evolve over time, for example, from revision to revision during the development of a project. This might give hints about similar text passages that are added by copy and paste and that were not present in a former version of the system. Also, an evaluation by an expert user study, possibly including eye tracking [25], is interesting to find out if our technique is easy to learn and if the visual patterns are useful to visually explore such data.

References

[1] S. Diehl, *Software Visualization – Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[2] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” Tech. Rep., 2007.

[3] Z. M. Jiang, A. E. Hassan, and R. C. Holt, “Visualizing clone cohesion and coupling,” in *Proceedings of the Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2006, pp. 467–476.

[4] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, “Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder,” in *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 106–115.

[5] S. G. Eick, J. L. Steffen, and E. E. Sumner, “Seesoft–a tool for visualizing line oriented software statistics,” *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[6] I. Goldstein, “Pretty printing: Converting list to linear structure,” in *Proceedings of Artificial Intelligence Memo 279, Massachusetts Institute of Technology*, 1973.

[7] R. C. Waters, “Using the new common Lisp pretty printer,” *ACM SIGPLAN Lisp Pointers*, vol. 5, no. 2, pp. 27–34, 1992.

[8] G. Lommerse, F. Nossin, L. Voinea, and A. Telea, “The visual code navigator: An interactive toolset for source code investigation,” in *Proceedings of the IEEE Symposium on Information Visualization*. IEEE Computer Society, 2005.

[9] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, “In situ understanding of performance bottlenecks through visually augmented code,” in *Proceedings of International Conference on Program Comprehension*, 2013, pp. 63–72.

[10] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf, “Visual monitoring of numeric variables embedded in source code,” in *Proceedings of the IEEE Working Conference on Software Visualization, VISSOFT*, 2013, pp. 1–4.

[11] L. Voinea and A. Telea, “Visual data mining and analysis of software repositories,” *Computers & Graphics*, vol. 31, no. 3, pp. 410–428, 2007.

[12] L. Voinea, A. Telea, and J. J. van Wijk, “CVSscan: visualization of code evolution,” in *Proceedings of the ACM Symposium on Software Visualization*. ACM, 2005, pp. 47–56.

[13] A. Telea and D. Auber, “Code flows: Visualizing structural evolution of source code,” *Computer Graphics Forum*, vol. 27, no. 3, pp. 831–838, 2008.

[14] D. Mount, *Bioinformatics: Sequence and Genome Analysis (2nd ed.)*. Cold Spring Harbor Laboratory Press: Cold Spring Harbor, NY, 2004.

[15] J. Slack, K. Hildebrand, T. Munzner, and K. S. John, “SequenceJuxtaposer: fluid navigation for large-scale sequence comparison in context,” in *Proceedings of the German Conference on Bioinformatics*, 2004, pp. 37–42.

[16] A. J. Gibbs and G. A. McIntyre, “The diagram, a method for comparing sequences. Its use with amino acid and nucleotide sequences,” *European Journal on Biochemics*, no. 16, pp. 1–11, 1970.

[17] M. Burch, S. Diehl, and P. Weißgerber, “EPOSee – A tool for visualizing software evolution,” in *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis*, 2005, pp. 127–128.

[18] ——, “Visual data mining in software archives,” in *Proceedings of the ACM 2005 Symposium on Software Visualization*, 2005, pp. 37–46.

[19] M. Burch, M. Raschke, and D. Weiskopf, “Indented Pixel Tree Plots,” in *Proceedings of 6th International Symposium on Advances in Visual Computing, ISVC*, 2010, pp. 338–349.

[20] M. Burch, H. Schmauder, and D. Weiskopf, “Indented pixel tree browser for exploring huge hierarchies,” in *Proceedings of the 7th International Symposium on Advances in Visual Computing, ISVC*, 2011, pp. 301–312.

[21] C. Ware, *Visual Thinking for Design*. Burlington, MA: Morgan Kaufman, 2004.

[22] M. Burch, H. Schmauder, M. Raschke, and D. Weiskopf, “Saccade plots,” in *Proceedings of Eye Tracking Research and Applications ETRA*, 2014, pp. 307–310.

[23] B. Shneiderman, “The eyes have it: A task by data type taxonomy for information visualizations,” in *Proceedings of the IEEE Symposium on Visual Languages*, 1996, pp. 336–343.

[24] J. S. Yi, Y. ah Kang, J. T. Stasko, and J. A. Jacko, “Toward a deeper understanding of the role of interaction in information visualization,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1224–1231, 2007.

[25] K. Kurzhals, B. D. Fisher, M. Burch, and D. Weiskopf, “Evaluating visual analytics with eye tracking,” in *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization, BELIV*, 2014, pp. 61–69.