Additional Detection of Clones Using Locally Sensitive Hashing

Nataliia I. Pravorska

Khmelnitsky National University, Ukraine
E-mail: pravorskana@khmnu.edu.ua

Received 28 October 2022; Accepted 26 January 2023;
Publication 16 May 2023

Abstract

Today, there are many methods for detecting blocks with repetitions and redundancy in the program code. But mostly they turn out to be dependent on the programming language in which the software is developed and try to detect complex types of repeating blocks. Therefore, the goal of the research was to develop a language-independent repetition detector and expand its capabilities. In the development and operation of the language-independent incremental repeater detector, it was decided to conduct experiments for five open source systems for evaluation using the industrial detector SIG (Software Improvement Group), including the use of a tool syntactic analysis. But there was the question of extending the algorithm for additional detection of duplication and redundancy in the code, which was proposed by Hammel, and how improvements can be made to achieve independence from the programming language. Particular attention was paid to the empirical results presented in the original study, as their effectiveness is questionable. The main parameters that were considered when creating the index for LIIRD (Language-independent incremental repeat detector) and its expansion of the LSH (locally sensitive hashing): measuring time, memory and creating an incremental step. Based on the results of experiments conducted by the

Journal of Cyber Security and Mobility, Vol. 12_3, 367–388.
doi: 10.13052/jcsm2245-1439.123.6

authors of Hammel's work, there was a motivation to develop an extended approach. The idea of this approach is that according to the original study, the operation of calculating the entire block index with repeats and redundancy from scratch is very time consuming. Therefore, it is proposed to use LSH to obtain an effective assessment of the similarity of software project files.

Keywords: Language-independent incremental repeat detector, locally sensitive hashing, incremental approach, incremental step, experiment, hash segment, hash function, clone index, shingles, MinHashing, shingling, software engineering.

1 Introduction

Today, there are many automated methods that help to detect blocks with repetitions and redundancy in the program code. A negative feature of such methods is excessive dependence on the programming language in which the code is written. In an effort to improve code representation, such methods are used by language parsers to find blocks of code with repetitions of more complex types. Therefore, building or finding parsers will not be supported for languages that are not very popular.

The language-independent incremental repeat detector that was developed is able to detect precisely specified types of clones, and the language-independent approach of detecting blocks of code with repetitions and redundancy allows to uniformly detect such constructs in different programming languages.

During the development of a language-independent incremental repeat detector, the problem arose of ways to expand the algorithm for additional detection of repetitions and redundancy in the code, which was proposed by Hummel et al. [1] and how improvements can be made to achieve programming language independence. Particular attention was paid to the empirical results presented in the original study, as their effectiveness is questionable.

To solve the given task, it became expedient to consider the extension of LIIRD by the method better known as "Locally Sensitive Hashing" (LSH). A significant reduction in the computational time required for the search process [2] occurs using approximate schemes based on LSH. The basic idea of locally-sensitive hashing is to use different hash functions to hash basic data points multiple times. At the same time, it will be guaranteed that similar elements have a greater chance to meet and end up in the same hash segment, unlike heterogeneous elements [3]. Only then are the elements that

have entered the hash segment, also known as candidate pairs, passed the similarity check.

2 Priming the Expansion of the Expanded Approach

Based on the results of experiments conducted by the authors of Hammel's work [1], the motivation for developing an extended approach arose. During one of the experiments, the Eclipse SDK (v3.3) (a platform for programming and compiling applications in Java) is analyzed. The software project itself has more than 42 million lines of code and more than 200,000 source code files written in the Java programming language. The study measures the time required to create the initial clone index and the time it takes to query and update it. The results of the experiment are presented in Table 1.

Based on these measurements, it is easy to see that the time required to create an index of repeats (clones) is quite large (7 hours and 4 minutes). Despite the fact that the experiment was carried out on rather outdated equipment, as of today, the time spent turned out to be quite significant. The purpose of our study is to find out if it is possible to increase the productivity of this step using LSH, and ultimately reduce the time required to generate intermediate information. The idea behind this approach is that, according to the original study [1], the operation of calculating the entire cloning index (block index with repetitions and redundancy) from zero turns out to be very time-consuming. Therefore, it is suggested to use LSH to obtain an effective assessment of the similarity of software project files. Subsequently, for those files in which duplicates are found, the index elements are calculated on the fly and the detection operation is performed. When using this approach, a compromise is required. While it seems intuitive that there may be an improvement in detector performance when building the index, there is a negative impact on the following aspects:

- **Feedback (Recall).** If the files turned out to be similar, only for them, with a similar approach, the process of detecting repetitions in the program code takes place. It follows that there will be skipped blocks with repeated code. For example, if the files are very large, then most of

Table 1 Analytical measurements of Eclipse SDK (v3.3) [6]

Creating an index (full)	7 hours 4 min
Index query (per file)	0.21 sec. median
	0.91 sec. average
Update index (per file)	0.85 sec. average

them are different, but there are some identical pieces of code in them. To reduce the likelihood of skipping repetitions and redundancy in the code (although this possibility will not be completely excluded), it is necessary to choose a low threshold of file similarity when comparing them.

- **Query performance (QueryPerformance).** The time required to query and detect blocks of code with repeats will increase with each new commit, due to additional computational overhead. Such overhead occurs when computing index entries on the fly for pairs of similar files. However, if the index creation time will be significantly reduced, then such behavior can be tolerated. There is an allocation of two main work processes [4], similar to a clone detector. For reuse in versions, the initial intermediate information is also created in the first process. However, in this case, the information will differ from the cloning index of the LIIRD. On the other hand, the second process is restarted for each new version of the software system. When using LSH, files that have been affected are moved into segments along with existing files that are similar to each of them and above a previously defined similarity threshold.

3 The Working Process of Creating a Similarity Index Based on LSH

This process consists of a number of sub-steps, as well as the process of creating the "Clone Index" mentioned in the paper [4]. Workflow substeps will result in the creation of an initial state of intermediate information. As in the LIIRD, the starting point is the same, that is, the same preprocessing steps will be used. However, the remaining part of the process will have significant differences. In this case, hashing blocks of code and storing them in the so-called "Clone Index" for the purpose of grouping similar files in one place and LSH will be used. The work [5] described the substeps of LSH, which will be used. This is shingling and generation of signatures (minhash) for given files, as well as grouping of similar ones. Below is a detailed description of each of them. A high-level workflow is presented in Figure 1.

Just as in the case of the LIIRD, the set of constant intermediate information will be called the index, which in Figure 1 is presented as LSH DS.[1]

[1] DS (Data Structures and Algorithms) – data structures and algorithms.

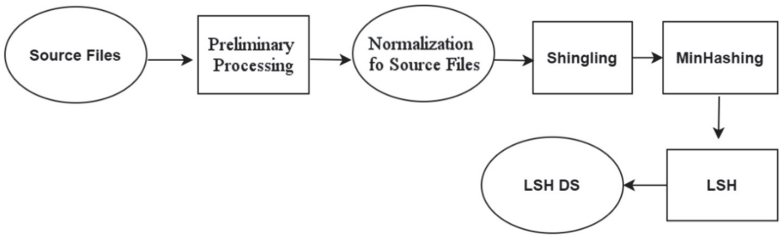


Figure 1 Sub-steps of the workflow of creating an index based on LSH.

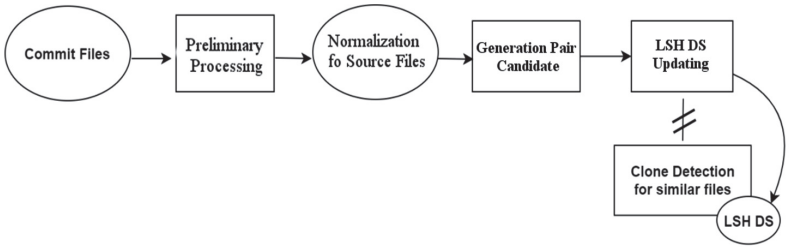


Figure 2 Sub-steps of the work process based on LSH.

3.1 Incremental Step-by-Step Workflow Based on LSH

The initiation of each incremental step, similar to the corresponding process of the LIIRD approach, occurs when the code base of the basic software project is updated to a new version. All normal preprocessing operations for this process remain unchanged. Another part of the pipeline is different from the LIIRD approach. More specifically, in the case of the detector, which will be based on the basis of LSH, identification is first carried out for the similarity of files in the existing codebase with the file that will be affected by the commit. For this, the LSH will be used, and the index will be queried based on the LSH. After that, each file falls into a hash segment with similar files. Likewise, the same approach as in the LIIRD detector is used for similar files. A hash value is generated and the same process is followed to detect repetition and redundancy in the program code. In Figure 2 presents a high-level overview, which is displayed by the substeps of this process. Commits in this case have the form of a JSON configuration file.

3.2 Approach Decomposition

Let's consider the decision regarding each separate sub-step of implementation and division of LSH in the context of a given extension. For this

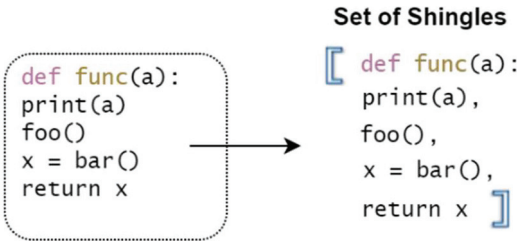


Figure 3 Converting a preprocessed file to a set of shingles.

extension, it is suggested to use a data set (datasketch) – which is presented by a third-party library, which includes a built-in implementation of MinHash LSH.

3.3 Shingling

This approach is covered in [5]. Shingles is the process of converting the original document into a set of shingles. In this case, the documents are the source files of the project to be analyzed. After the pre-processing step, a file is obtained which is transformed into a set of tiles under the action of conversion. That is, each line of the file is a separate tile. Figure 3 presented the result of this process, showing how the pre-processed file is converted into a set of tiles corresponding to the corresponding lines of this file.4

3.4 MinHashing

MinHashing is the next step in the process of the general LSH algorithm. That is, the purpose of this step is to eliminate the phenomenon of large sets of tiles, due to which there is an increase in the time required for the calculations of the similarity metric or the Jaccard coefficient. In this case, there is a need to use k-hash functions. Under their influence, hashing takes place for each individual file, each tile in a set of tiles. Next, the algorithm selects the lower value of the hash function for each of the k hash functions, and thanks to this, a signature is generated. Figure 4 presented the second substep of the process.

This transformation is combined with loss of information, due to the transformation of sets into fixed-length signatures. That is why it is not possible to use them to calculate the exact similarity. In [3], however, it was demonstrated that it is possible to obtain accurate estimates even in this case. It follows that the accuracy of the estimate depends on the length of the signatures, that is, it depends on the number of used hash functions.

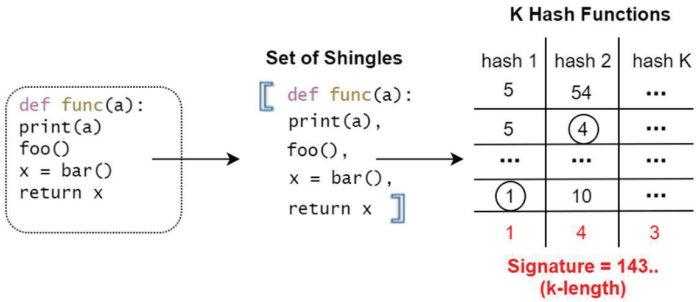


Figure 4 Signature generation using MinHashing.

The higher the number of hash functions, the better the score, but the process of calculating the score becomes more time-consuming.

You can actually set the estimation error coefficient by the following formula:

$$error = \frac{1}{\sqrt{k}},$$

where k – is the number of hash functions, that is, for example, a 6.25% probability of false-negative or false-positive activations for 256 hash functions will be obtained. The datasketch configuration parameter can be used to support changing the number of hash functions.

4 Locale-sensitive Hashing

Increasing the computation time required to compare each signature with any other signature is the final step of the LSH approach. As already mentioned in work [5], such a technique as banding is used (that is, banding – encirclement, binding, unification). The technique consists in the fact that all hash values for each tile from the set that make up the matrix are divided into b columns, each of which in turn consists of r rows. This is followed by consideration of document D1 to pair candidates with another document D2. At the same time, it is necessary that each row of the defined range D1 coincides with each row of the corresponding range in D2. If this happens, the documents will be considered pairs of candidates and fall into the same hash segment. However, you will have to choose the value for b and r yourself, in relation to the configurable LSH implementation. At the same time, you should always pay attention to how these values affect the similarity threshold. On the contrary, datasketch works, allowing you to determine the desired threshold, thanks to the automatic calculation of the correct values of these parameters.

4.1 The Results of the Experiment

To visually confirm the operation of the LIIRD and the use of its extension LSH, a demonstration of the results of the experiments is presented. All studies for the authenticity of the experiment were carried out on an experimental machine with the following configuration (although somewhat outdated): memory – 32 GB; processor – Intel Xeon E5-2650 v2 @2.6GHz. Despite the fact that the experiment was carried out on rather outdated equipment, as of today, the time spent turned out to be quite significant.

More specifically, to begin with, the results of experiments with the proposed LIIR detector will be considered. Next, we will present the results of experimental attempts to measure the effectiveness of the traditional SIG approach, which was mentioned in [4, 5], for detecting blocks with repetitions and redundancy in the program code. Comparison of the SAT (software analysis test) approach with the additional repeat and redundancy detector proposed in the study. And at the end, the results of the experiments are provided, thanks to which the effectiveness of the expansion, which is based on LSH, is studied. The obtained results are studied in comparison with the results of experiments with LIIRD.

4.2 Measurement of LIIRD Indicators

Table 2 contains the exact measurements that were used for the evaluation experiments of the proposed LIIRD. Five open source systems were selected for the study.

The table shows that compared to the initial estimates obtained with the CLOC[2] tool (this tool is capable of counting blank lines, comment lines, and

Table 2 Dimensions of LIIRD

Project	Previous Measurements Number of LOCs	Number of Lines (LOCs) Read in Commits	Processed	Index Creation Time (sec)	Average Step Time (sec)	Standard Deviation of the Incremental Step (sec)
Rippled	312.011	208.100	42	3.75	0.85	1.31
Kooboo	670.265	681.143	50	16.28	0.03	0.04
Tensorflow	3.194.893	3,814.652	45	65.89	4.29	3.32
OpenJDK-14	12.045.316	3,377.211	46	48.53	4.72	5.07
LinuxKernel	23.229.768	23.603.823	45	321.21	N/A	N/A

[2]Description and access to the tool at Github: https://github.com/AlDanial/cloc.

physical lines of source code in many programming languages), the LOC (Lines Of Code) for each project will range in within the margin of error of approximately 2.7%. Exceptions are Tensorflow and OpenJDK, for which a significant portion of the total code base is apparently made up of excluded directories and file extensions. Moreover, almost all of the fifty commits that were analyzed for each codebase were processed. A reminder here is that all omitted commits refer to commits that only affect files ignored by the implementation of the approach proposed in the study in the first place (e.g., text files).

The next two columns provide an overview of the average time that has passed since the index was created and the incremental steps (steps with increments) of the workflows to implement the LIIRD. The last column highlights the standard deviation for the incremental step time measurement.

The experimental software proposed in the study successfully completed the analysis of four software systems, but for Linux systems, it could not be done due to its complexity and size, so the table substitutes the term N/A instead of the values.

5 Creating an Index for LIIRD

5.1 Time Measurement for LIIRD

The indicators obtained when measuring the time for the index creation process by the LIIR detector proposed in the study are presented in Figure 5.

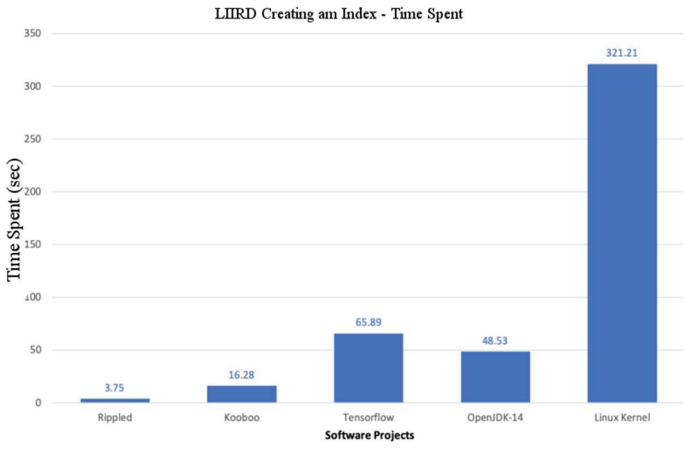


Figure 5 LIIRD – Time of execution of the index creation process.

The total time required to generate the index required by the LIIRD, as can be seen, will range from a few seconds for small systems (such as Rippled) to about five minutes for large systems (such as the Linux kernel), bearing in mind, that the time required for this step depends on the size of the software system. This is expected because the calculation time will be required more and it depends on the repetition indices (source code clones), which in systems with a lot of source files and LOC are quite large. However, the overall index creation time will fluctuate at a fairly low level. It is taken into account that such a time took exactly more, about five minutes for such a large system as the Linux kernel, which consists of more than 20 million LOCs. A comparison can be made with Table 1, which presents the data obtained in the course of Hammel's work [1]. As can be seen, the approach using MNIDP spends less time to create the index.

5.2 Memory Measurement for LIIRD

For the analysis of the five software systems used in this study, the amount of memory (Table 3) required for the proposed LIIR detector is used.

The figures presented are still largely in line with the memory requirements of the first systems, despite the fact that measurements were taken throughout the entire analysis (including index creation and incremental steps) of the proposed systems. Short-term spikes in memory usage, as will be highlighted in the implementation of the incremental step for LIIRD (Figure 6), were mostly caused by taking additional steps in a few seconds. It follows that when considering the overall memory usage that is required for a specific detector implementation, their appearance does not matter. Figure 6 shows the memory usage required by the LIIRD approach during each analysis. As you can see, even taking into account modern memory standards, the indicators for the first four systems are quite insignificant. However, memory usage rises to higher levels when dealing with a larger Linux kernel system.

Table 3 LIIRD – memory indicators

Project	Memory (MB)
Rippled	129
Kooboo	348
Tensorflow	1791
OpenJDK-14	1712
LinuxKernel	12500

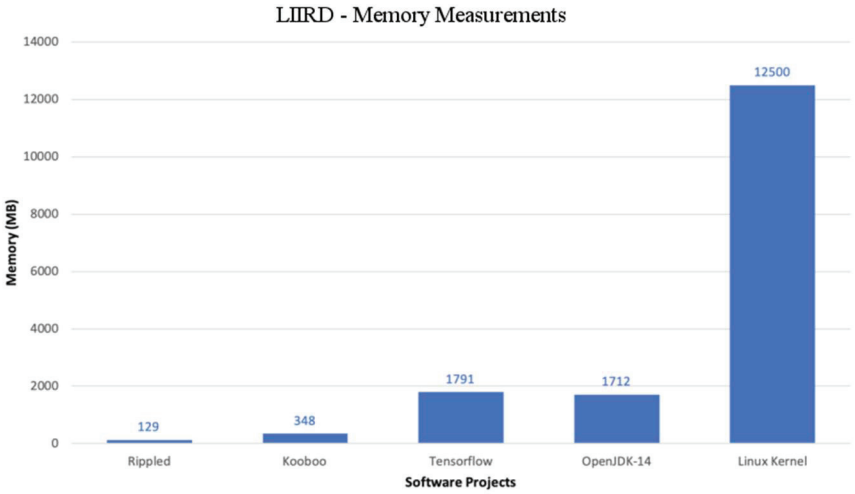


Figure 6 LIIRD – Cumulative memory requirements.

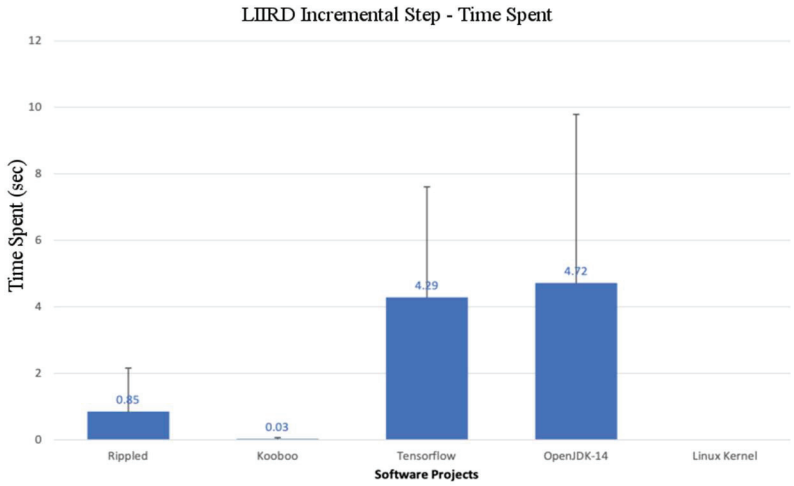


Figure 7 LIIRD – average execution time for an incremental step.

5.3 An Incremental Step for LIIRD

Changes in the time spent for the incremental step of the implementation of the LIIRD are presented in Figure 7. This process is started every time there are code changes in the form of a commit. The results obtained for Rippled, Tensorflow and OpenJDK systems give an idea that such a process does not take more than a few seconds on average.

Regarding the unusual deviation during the experiments for Kooboo, such low figures justify themselves in cases where the process of detecting blocks with repetitions and redundancy is not started at all. This happens if the files affected by the commit are not cloned and destroyed. This causes the most time-consuming parts of the detection process to fail. Finally, due to the size of the Linux kernel, the proposed setup could not cope with the memory load of this process.

Note that this occurred not because of the memory requirements for the incremental step per se, but because of the version control checking sub-process running in the application developed in the study. Because for large codebases, such a process would require a significant amount of memory, more than what was available on the experimental setup machine at the time. More generally, a large number of different factors affect the execution time required for an incremental step. Elements that can affect the results are understood as: the number of files that are included in the commit; type of changes (creation, update, deletion, renaming); length of analyzed files; similarity threshold in the case of LSH.

Additional experiments were conducted to gain a complete picture of the interaction of all these factors.

It is important to note that the overall performance of this step is more efficient compared to the index creation process. This is clearly visible in Figures 5 and 6 when comparing them. This is significant given that the process is performed every time new changes are made to the code.

5.4 Evaluation of LIIRD vs SIG

The results of our study analyzing the performance of the general SAT analysis (this tool is used by the SIG to perform quality analysis of a software project and measure various indicators such as its maintainability – and compare it to the approach proposed in the study) together with a sub-process built into the same tool and designed to detect blocks of repetition and redundancy in the program code are presented in Table 4.

More specifically, the SAT SIG tool was used in the analysis of the data set for the proposed study, which consists of five open source projects.

Given that SAT is a complex tool that includes many basic operations unrelated to duplicate code detection, it was decided to isolate the relevant parts and measure the fraction of total elapsed time. This fraction is directly related to the detection of blocks with repetitions and redundancy in the program code.

Table 4 SAT parameters and total detection time of LIIRD

Project	Total SAT Analysis Time	Clone Detection Time	Creation of the LIIRD Index and the Time of the Incremental Step
Rippled	4 min.	5.63 sec.	4.6 sec.
Kooboo	22 min.	397 sec.	16.31 sec.
Tensorflow	8 hours 30 min.	177.1 sec.	91.29 sec.
OpenJDK-14	N/A	N/A	56.34 sec.
LinuxKernel	N/A	N/A	>321.21 sec.

As already mentioned in point 4.2, the experimental software used in the study successfully completed the analysis of three software systems, but for the OpenJDK and Linux systems, it was not possible to do this due to their complexity and size, so the term N/A is substituted in the table instead of the values.

However, it can be observed that even if the time required to detect blocks of code with repetitions and redundancy is only a small fraction of the total analysis time, SAT will require a significantly larger amount of time to perform the analysis. It is especially visible that for Tensorflow the total analysis time was more than 8 hours. This number indicates a limitation of the number of checks that could be made to five. Regarding the time indicators for detecting clones, they vary from a few seconds to a couple of minutes for large and more complicated systems. It is necessary to pay attention to the fact that the time of detection of blocks with repetitions and redundancy depends not only on the size of the packet, in the context of SAT. Additional factors that affect these measurements are: the complexity of the code itself, the programming language in which the system is written. Accordingly, and to reduce the time of detecting blocks with code clones for Tensorflow compared to the smaller Kooboo system.

When repeatedly starting the clone detection process, individual measurements, which at first glance may seem quite small, quickly accumulate. The influence of the measured time, when the process of detecting repetitions and redundancy in the program code is repeated often, can be well understood from the presented diagrams in Figure 8.

In particular, the Tensorflow system was used as a parameter to demonstrate how the time required to detect clones will increase as the number of commits increases. As indicated, while the difference for a single commit does not appear to be large, the results look different for a large number of commits, such as 100 or 500. In the latter case, the entire analysis, for example, using a traditional SIG tool, would require approximately 1387

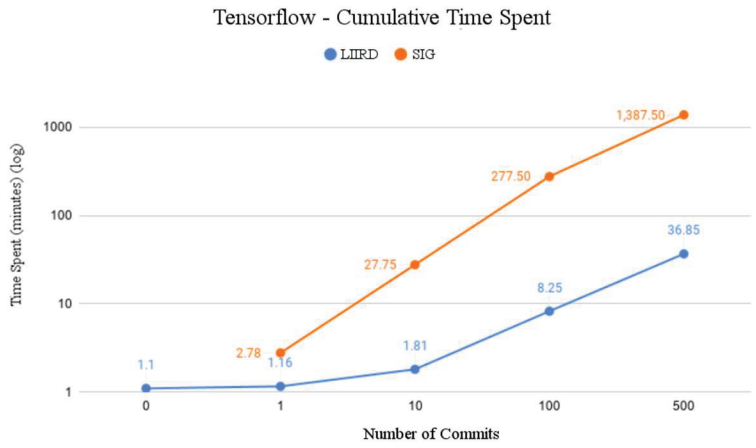


Figure 8 Cumulative comparison of the SIG detector with the LIIRD.

Table 5 Measurement of expansion based on LIIRD and LSH

	Index Creation Time (sec)		Average Incremental Step Time (sec)		Standard Deviation of Incremental Step (sec)	
Project	LIIRD	LSH	LIIRD	LSH	LIIRD	LSH
Rippled	3.75	10.42	0.85	0.82	1.31	1.35
Kooboo	16.28	37.82	0.03	0.25	0.04	0.04
Tensorflow	87	192.8	4.29	1.57	3.32	2.13
OpenJDK-14	51.62	139.87	4.72	4.26	5.07	6.34
LinuxKernel	321.21	954.56	N/A	4.38	N/A	10.23

minutes – around 11 p.m. On the other hand, it only takes about 37 minutes to perform the same procedure using a step-by-step approach.

5.5 Measurement of Expansion Based on LSH

In order to evaluate the implementation proposed in the study based on LSH, it is necessary to compare it with the results of experiments with the LIIR detector. To perform an individual comparison between them, individual measurements are made for two workflows for each of the two approaches presented in Tables 5 and 6. It should be noted that for this type of research, the same experimental procedures were used as those presented for measuring time, memory and an incremental step for LIIRD (see above). This means that the same number of LOCs and the same number of commits were processed for each of the systems.

Table 6 Measurement of expansion based on LIIRD and LSH

	Memory (MB)	
Project	LIIRD	LSH
Rippled	129	60
Kooboo	348	122
Tensorflow	1791	524
OpenJDK-14	1712	429
LinuxKernel	12500	2600

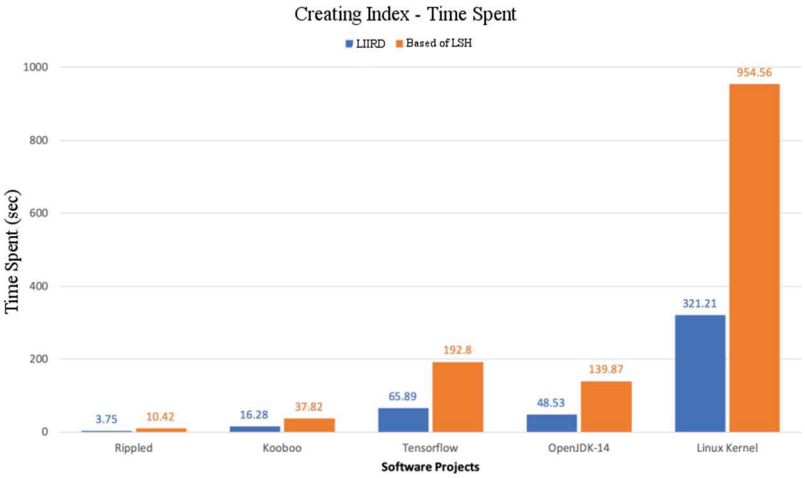


Figure 9 Execution time of the index creation process for two implementations.

As mentioned earlier, N/A values are provided to the LinuxKernel system, due to the inability to detect them using LSH.

6 Creating an Index for the LSH

6.1 Time Measurement for LSH

The resulting large time difference between the two implementations relates to the time required to create the index. It can be seen from Figure 9, that for all software systems in our information fund, the LIIRD approach is almost three times faster compared to the expansion based on LSH.

Reminder – 64 hash functions were used for the MinHashing process when implemented based on LSH. The number of hash functions was chosen so low precisely to create a threshold for how long it takes for this extension

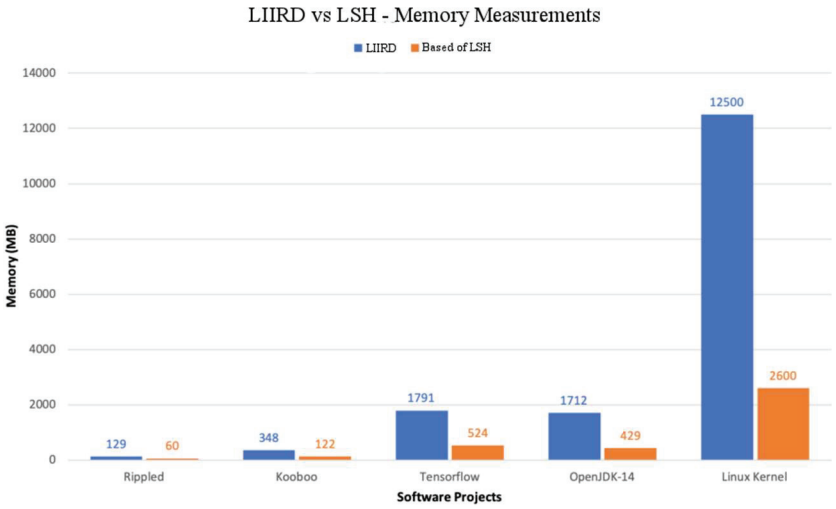


Figure 10 Memory requirements for two implementations.

to build an index, as this number results in a similarity error rate of 12,5%. Additional time costs, which subsequently lead to an increase in the time of creating an based on LSH implementation index, arise precisely because of an increase in the number of hash functions, and thus, a decrease in the number of errors.

6.2 Memory Measurement for LSH

The memory requirements of the based on LSH expansion, which are constructed together with the preliminary measurements carried out in the study for the LIIRD approach, are presented in Figure 10. During the research, it was found that the memory usage levels of the LIIRD approach are two to three times higher than those of the based on LSH approach. Especially in the case of the Linux kernel, there is a significant difference between the two approaches: approximately five times more memory will be required to complete LIIRD detection.

6.3 Incremental Step when Using LSH

Corresponding measurements regarding the execution of an incremental step for two implementations are presented in Figure 11. At first glance, the relevant data obtained look rather confusing. All because worse performance

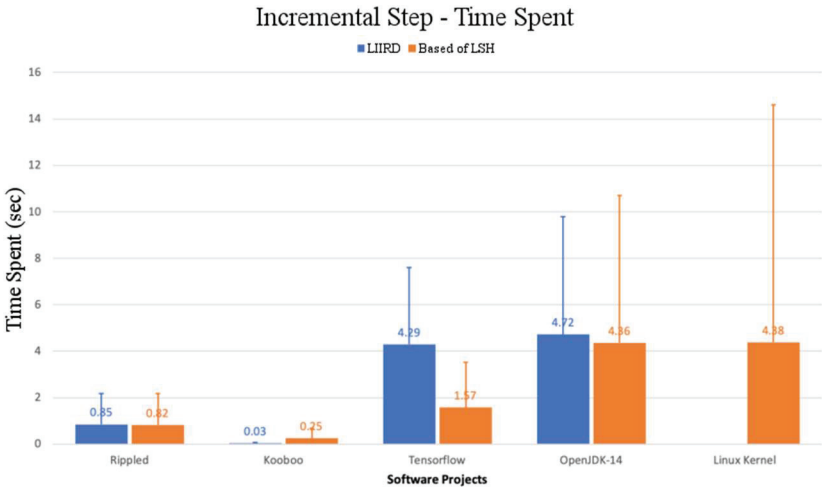


Figure 11 The execution time of the incremental steps process for two implementations.

was expected for all analyzed projects, compared to the LIIRD approach than the proposed based on LSH provisioning, taking into account the additional step of calculating the repeat index entries (clones) and redundancy on the fly.

However, upon further investigation, it became clear that in many cases the chosen similarity threshold of 20% was not low enough to identify a large number of files for similarity and was the reason why this was not reflected in our visualization. And as a result, many detections were missed, which led to a decrease in time, in the diagram presented of Figure 11. Upon further investigation into the reasons for such a large disparity between provisions, it turned out that the MinHash part of the overall LSH scheme is quite computationally heavy. In fact, approximately 37% of the time (Figure 12) when creating the index was spent on the MinHashing substep, that is, it follows from the raw data obtained by isolated measurements of the time of the index creation step for the implementation of LSH using the Tensorflow system. The remainder (namely 63%) was split between shingling and the process of banding (shown as LSH), with the former taking the most time.

6.4 Hash Function Variables

In work [5] it was mentioned that 64 hash functions for the MinHashing process were used to perform the tasks of our research with an implementation based on LSH. Also, since MinHashing is one of the most time-consuming

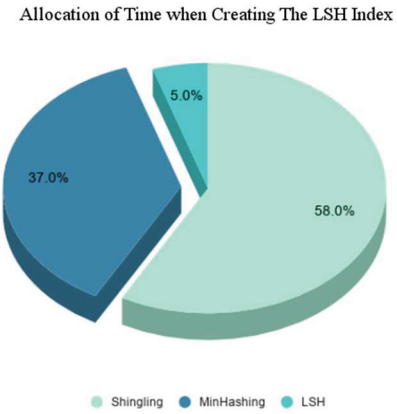


Figure 12 Sub-dimensions of creating an index for implementation based on LSH.

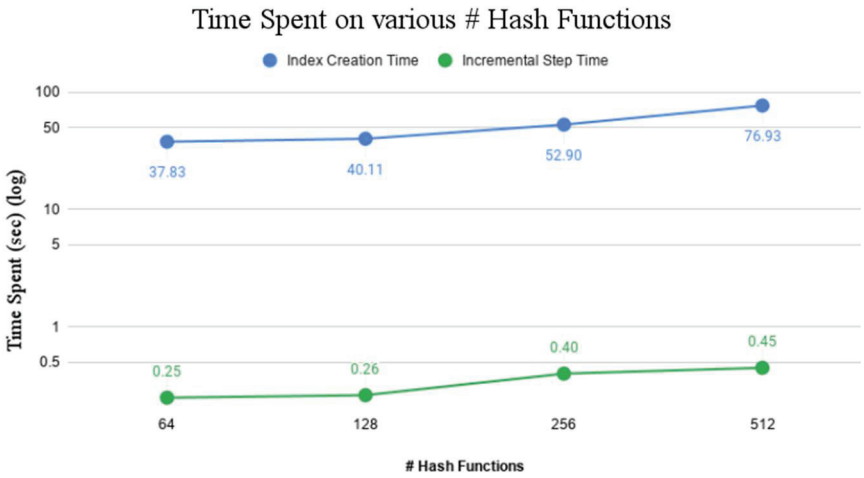


Figure 13 Execution time value for different number of hash functions in MinHashing process.

parts of the LSH implementation, it would be interesting to investigate how changing the number of hash functions will affect the incremental step and total index creation time.

The consequences of increasing the number of hash functions, correspondingly reducing the number of errors, the time required to create the corresponding index, and the time for processing the incremental step are presented in Figure 13. More specifically, the Kooboo software project was used and index creation times were measured for 64, 128, 256, and 512 hash

functions, which in turn would correspond to error rates of 12.5%, 8.8%, 6.25%, and 4.4% respectively.

From the diagram, it turns out that the important role, when processing two work processes, the required time depends on the number of hash functions. In particular, the index creation time practically doubled when comparing the measurement data of the lowest threshold of 64 and the highest of 514 hash functions. This behavior is consistent with measurements of the incremental step of the process, where an increase of a similar magnitude was observed

7 Conclusions

The results of the experiments conducted for the LIIRD and those presented in Hammel's work [1] do not fully agree. A significant difference was observed in the measures of time to create the index and the incremental step of the workflow. Of course, there were expectations that under the influence of the three main factors during the measurements, there will be minor deviations.

These factors were:

Experimental device. Unlike the hardware used in Hammel's original study [1], the hardware used in our experiments was much faster.

Normalization. Exclusion in the study of the proposed implementation of the stage of tokenization of the original detector, to take into account independence from the programming language, leads to the removal of additional overhead costs necessary for this process.

Storage in memory. Memory persistence was used in contrast to the original study where the measurements were of an implementation that stored intermediate information in a database. Taking into account the factors mentioned above, it was expected that the time required to create an index would be reduced.

Although the expectation is supported by the findings, the difference is much larger than these factors can justify. Specifically, the time required to create an index for a project consisting of approximately 40 million lines of code (LOC) was 7 hours and 4 minutes, according to initial research. However, when conducting our experiment, this indicator was already a little more than 5 minutes for LIIRD and about 15 minutes for the based on LSH approach, in relation to the Linux Kernel with half the size. These

measurements indicate processing time differences of about a factor of 42 and 14, respectively, assuming that Hummel's approach scales linearly. Based on this observation, it can be said that any further attempts to improve the creation of the index will be quite difficult, since it is happening much faster than expected.

Regarding the additional step, the results obtained from the experiment carried out in the work provide new insights compared to the original study. More specifically, the obtained measurements allow us to conclude that the size of the system affects the total time of the incremental step. At first, this is not logical, since the query and update of the hash index should result in similar constant time measurements. However, collisions may occur, which lead to additional overhead in the overall calculation in large systems where the index is filled with a very large number of records. In addition, individual measurements are affected by several factors, such as the number of files in the commit, the length of those files, and the type of changes they make. Finally, the initial study simulates commits by randomly deleting and re-adding source files. Because our study used actual commits, the results provide a better understanding of incremental step workflow behavior.

Based on the research, it has been found that using a language-independent repeat detector is appropriate for detecting certain types of blocks with repetitions and redundant code, but improvements due to the use of LCH will be quite difficult.

References

[1] Benjamin Hummel, ElmarJuergens, Lars Heinemann, and Michael Conradt. Indexbased code clone detection: incrementtal, distributed, scalable. In 2010 IEEE International Conference on Software Maintenance, pages 1–9. IEEE, 2010.

[2] Indyk Piotr, MotwaniRajeev. Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the thirtieth annual ACM symposium on Theory of computing, pages 604–613, 1998.

[3] LeskovecJure, RajaramanAnand, UllmanJeffrey David. Mining of Massive Datasets. Cambridge University Press, USA, 2nd edition, 2014. ISBN 1107077230.

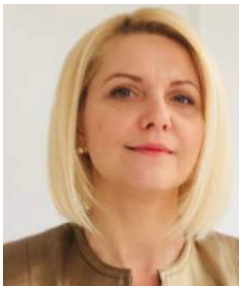
[4] Pravorska N.I., Barmak O.V., Medzatiy D.M., Shestakevych T.V. The process of detecting blocks with repetitions and redundancy when

using a language-independent incremental detector. KHNU Bulletin, Technical Sciences series, №3, 2021, pp. 39–45.

[5] Pravorska N.I., Bedratyuk L.P, Forkun Y.V. Yashina O.M. Language-independent detector for detection and elimination of repetitions and redundancies of the program code. Measuring and computing equipment in technological processes. – Khmelnytskyi, 2021. №1, pp. 56–61.

[6] ZhouWei, HuJiankun, WangSong. Enhanced locality-sensitive hashing for fingerprint forensics over large multi-sensor databases. IEEE Transactions on Big Data, 2017.

Biography



Nataliia I. Pravorska has the degree of Candidate of Pedagogical Sciences 2005, PhD in Pedagogy (theory and methods of informatics (computer science)) 2011, MS of Software Engineering 2021. She is a associate professor of the Department of Software Engineering at Khmelnytskyi National University (2004–). Research interests: C++ and Java programming, object-oriented programming, development of software products based on mathematical models, Internet of Things. Educational activity. Teaches disciplines: Applied information systems, Software design, Object-oriented programming, Basics of team software development, Java programming technologies, Software systems development methodologies and technologies.