An Empirical Comparison of Code Similarity Algorithms

Jahnae Miller
Computer Science, NYUAD
jahnae@nyu.edu

Advised by: Yasir Zaki, Moumena Chaqfeh

ABSTRACT

As the usage of JavaScript libraries becomes more prevalent, so does the adoption of bad and careless practices, such as including multiple versions of the same library within the same document [16]. These bad practices negatively impact the page load times (PLTs) of the web pages. Thus, an intuitive way of reducing page load times is removing these duplicate scripts.

There are four main categories of duplicate detection techniques: metric-based techniques, token sequence-based techniques, tree-based techniques and PDG-based techniques. However, PDG-based and metric-based techniques have already been proven to be costly and not very accurate, respectively. Thus, suitable candidate algorithms must be found within the realm of tree-based and token-sequence based techniques.

In this paper, three algorithms are explored: two token-sequence based algorithms and one tree-based algorithm. These algorithms are compared and evaluated to find the best algorithm for use in a light-weight tool meant to help reduce page load times by cutting out unnecessary JavaScript.

KEYWORDS

clone detection, web optimization, javascript, similarity comparison, redundant code

Reference Format:
Jahnae Miller. 2020. An Empirical Comparison of Code Similarity Algorithms. In *NYUAD Capstone Project 2 Reports, Spring 2020, Abu Dhabi, UAE.* 8 pages.

1 INTRODUCTION

The web has become one the one of the major platforms for software applications [20], leading to the meteoric rise of JavaScript's popularity as a programming language. JavaScript is the most widely used language for client-side applications, and it has even made its way to the server-side [9]. To compliment its popularity, a massive number of JavaScript libraries and frameworks have been developed to perform a multitude of tasks and ease the burden of web development. In fact, 86.6% of the Alexa Top 75k websites used at least one well-known JavaScript library [16]. However, despite their usefulness, these libraries and frameworks contribute heavily to "web bloat".

"Web bloat" describes the trend in websites to increase in size and complexity over time [8]. These increases in size and complexity have a profoundly negative impact on the PLTs (Page Load Times), a key performance metric of websites. While network conditions and download speeds play a role in PLTs, a major part of the increase in PLTs comes from the evaluation of JavaScript and its blocking of HTML (HyperText Markup Language) parsing. Both JavaScript and HTML affect the DOM (Document Object Model), and when modifying shared resources, maintaining the correct order of execution is critical. To ensure the correct order of execution, JavaScript evaluation blocks HTML parsing, but this slows down the rendering of the page, thus increasing page load times. The speed of JavaScript evaluation depends on factors, such as CPU speed and memory, that are not easily or cheaply modified. Accordingly, special techniques must be introduced to reduce the page load times, especially in developing regions where page load times are cripplingly slow and can range from tens of seconds to a few minutes [23].

One of the obvious solutions is simply disabling or removing the JavaScript, especially the massive libraries that take much longer to download and parse, but JavaScript provides the bulk of functionality on the web pages. For example, the most widely used library of the Alexa Top 75k websites is jQuery [10]. jQuery simplifies and makes it much easier to implement and use the most widely used JavaScript features, such as HTML DOM Tree traversal and manipulation, and event handling. If a library like jQuery is removed, most of the page's functionality will be removed alongside it. Even further, some web pages and their content are generated entirely by JavaScript. Over 75.9% of the home pages of 6,085 popular websites use JavaScript dynamic generation techniques [22].

Not all the JavaScript that is included in web pages is strictly necessary, however. Some of these JavaScript libraries, often loaded from external domains and referred to as third-party libraries, are included in web pages to not simply make the use of JavaScript features easier and more accessible like jQuery does but to add features, such as advertising, tracking, social media, analytics and more. 43% of 4,517 sites include JavaScript files from at least three external domains [22]. Most of these modules do not add any functionality that actively benefits the users of the websites, and tracking may even be considered harmful.

Furthermore, the prevalence and wide-spread use of these libraries by web developers have led to the adoption of careless practices, such as including multiple versions, or even the same version, of a library in the same document [16]. Having multiple copies of the same version of a library does not add any additional functionality to a website. Instead, it can lead to security risks and non-deterministic behaviours because asynchronous loading makes it unclear which copy of the library will be used [16].

An intuitive way of speeding up PLTs is identifying and removing the duplicates of the JavaScript libraries. Exact duplicates, which can be assumed to be copies of the same version of the library or script, can be removed immediately. However, treating different versions of the library will require another solution. A website might include an older version of a library because it uses features which are removed from or not supported by newer versions of the library. However, it may also be the case that the other version is still present because of a development oversight. To this end, near duplicates should also be flagged and identified, so that they can be reviewed.

There exist a multitude of algorithms to identify similarities between documents. However, different algorithms will perform differently based on the type of document given and the parameters used to tune the algorithms. In this case, the algorithms will be applied to JavaScript source files. This paper explores three similarity detection algorithms: k-shingling, winnowing and abstract tree fingerprinting. Even further, this paper evaluates and compares their performance based on four metrics: run-time, accuracy, CPU usage and memory usage. Where possible, these metrics are also compared against a baseline: the Stanford MOSS (Measure of Software Similarity), a tool used for plagiarism detection. These comparisons are done in hopes of finding the best algorithm for use in a light-weight tool meant to help speed up PLTs by removing the unnecessary JavaScript that wastes CPU resources and blocks HTML rendering.

2 BACKGROUND AND RELATED WORK

First and foremost, Schleimer et al. posit that a duplicate detection algorithm should possess three main properties: white-space insensitivity, noise suppression and position independence. To ensure whitespace insensitivity, matches should not be affected by whitespace, punctuation, capitalization, etc. These can be easily treated by going over the data and removing the whitespace and punctuation, and then transforming all the text to either uppercase or lowercase. When matching software text, it is also desirable to remove sensitivity to variable names. For noise suppression, matches must be large enough to indicate duplication, and that the word is not a common word or idiom. Finally, to ensure position independence, changing the order of the contents of the document should not affect the measure of similarity. Also, removing part of the document should not affect the matches in the parts that remain, and adding to the document should not affect the matches in the original portion [19].

There exist four main categories of clone detection techniques, all possessing all or some of these properties: metrics-based techniques, token sequence based techniques, tree-based techniques and program dependency graph (PDG) based techniques [2].

2.1 Metric-based techniques

Merlo et al. compute different metrics for chunks of code and compare the metric vectors instead of comparing the code directly [12]. The metrics typically chosen are: the number of functions called, the ratio of input/output values to the number of functions called, the number of linearly independent paths through the program's source code, a modified function point metric, and a modified information flow quality metric. Distance, the Euclidean distance, for example, within a certain threshold between these vectors can hint at similar code. For web pages, in particular, one suggested metric is the number of the number of occurrences of each tag on the page [5]. For JavaScript files, programmer-defined functions can be checked for possible cloning by first checking the identifier used as their function names for equality, and then comparing metrics, such as the number of lines of code, number of effective lines of code (excluding blank lines and comments), and the number of commented lines of code [15]. Metric based typically methods perform poorly because of their sensitivity to minor edits, and, nowadays, very few programs use pure metric based methods [3].

2.2 Token sequence-based techniques

Token based matching is a historical approach to dealing with source code plagiarism detection. Source code is tokenized and then supplied to string matching algorithms. A fingerprint selection strategy can be used to lessen the number of tokens that are kept and compared. JPlag [18], a web-based code plagiarism detection tool, uses an algorithm based on the Greedy String Tiling algorithm. In the Greedy String Tiling algorithm, the two texts are first broken up into a sequence of tokens where the tokens are taken from a set of "significant keywords", such as keywords and built-in functions. A tile is a permanent and one-to-one association between a substring in one set with a matching substring in the other set. When a tile is formed, the substrings are marked and cannot be used in any other matches. The matches are assumed to be as long as possible, and a minimum match length can be used to set a threshold which matches must be longer than to be counted. The Greedy String Tiling has a worst case complexity of $O(n^3)$ [21].

2.3 Tree-based techniques

Tree-based methods aim to exploit the syntactic properties of programs. By only exploring the syntactic properties, this method is not affected by typical obfuscation methods, such as renaming variables. Tree-based methods consider the syntax tree obtained from parsing the source code. Abstracted syntax trees provide better recall. The edit distance of the trees can be computed, but hashing the trees and comparing the hashes is much more scalable. The syntax trees are typically hashed and put into buckets based on their hashes. To avoid doing a large number of comparisons, $O((nm)^2)$ for comparing nm sub-trees of m projects of size n, only hashes within the same buckets are compared [3]. A threshold can be specified and sub-trees with weights below that threshold can either be ignored completely [1] or, in solutions such as CloneDR, hashed to the same value [4]. However, hashing all sub-trees below the threshold to the same value leads to an increased number of false-positives.

2.4 PDG-based techniques

A Program Dependency Graph (PDG) can be used to represent the control and data flow dependencies within a function. Statements and control predicates are represented by the nodes in the graph. The edges incident to a node represent the data values that the nodes operations depend on and the control conditions the execution of the operations is dependent on [7]. Isomorphic subgraphs may be identified as clones. However, this is very costly, as finding isomorphisms between graphs is a NP-complete problem. Krinke [2, 14] uses approximate solutions to find these isomorphic graphs because of the computational complexity.

3 METHODOLOGY

Since finding isomorphic subgraphs is costly, and the ideal duplicate detection algorithm should be fast and lightweight, as well as, accurate, PDG-based techniques were not considered. Additionally, because of their known issues with accuracy, metric based techniques were not considered either. Two token-based methods were considered: k-shingling and winnowing, and the final duplicate detection algorithm considered was tree-based: abstract syntax tree fingerprinting.

For both of the token-based algorithms, extra steps were taken and the JavaScript files were first pre-processed to ensure whitespace insensitivity. In both cases, the whitespace and punctuation were removed and the data was all transformed to lower case. All identifiers were replaced with the letter 'X' and all string literals were replaced with the letter 'Y'.

3.1 K-Shingling

K-Shingling is a token-based document clone detection algorithm. It is a method of transforming documents into sets of tokens upon which set similarity comparisons can be performed. Firstly, the documents are divided into k-sized shingles (or k-grams). A shingle is a contiguous sub-sequence of characters or words in a document. The size of k should be chosen to reduce the odds of any given shingle appearing in a document [17]. A size of 3 was found to perform the best, in terms of both accuracy and run-time. Picking shingles of size k also ensures noise suppression, as no tokens less than size k will be considered.

After the shingles are formed, their SHA-1 hashes are computed. The choice of hash function can be changed to MD5 or a faster algorithm to speed up the comparisons, but it was found that SHA-1 performed just fine. Since we are dealing with sets, any repeated hash values are immediately discarded. This reduces the number of comparisons required to calculate the set similarity greatly, especially considering the repetitive vocabulary of source code. The Jaccard similarity index is used to represent the similarity between the sets of hashes obtained from each document. The Jaccard similarity index is given by: $\frac{A \cap B}{A \cup B}$. It is the ratio of the intersection of the sets to their union.

Intuitively, k-shingling scales much better than the abstract syntax tree fingerprinting algorithm because of its linear complexity in time and space. More importantly, this algorithm works on more than just source code. For the Abstract Syntax Tree representation to work, the document must be complete and syntactically correct, as well as, in a programming language that the parser producing the syntax tree understands [13].

3.2 Winnowing

Winnowing can be thought of as a relative of the k-shingling algorithm, but it is more robust. Once more, the documents are divided into sets of k-grams and the k-grams are hashed. Winnowing has extra steps to reduce the number of comparisons performed to calculate the similarity of the sets. The winnowing algorithm selects a specific subset of the total hashes of all the k-grams to represent the document. This subset is referred to as the "fingerprints" of the document.

To select the fingerprints, the winnowing algorithm employs a sliding window. In each window, the minimum hash value is chosen. The minimum is chosen in each window to keep the number of fingerprints chosen minimal. Since the window is a sliding window, the minimum in a window is likely to still be the minimum in adjacent windows, since the chance that a random number, w, is smaller than the minimum of a set of x random numbers is low [19].

Once the fingerprints of the document are chosen, they can be compared to the fingerprints of another document by once more using the Jaccard similarity index.

On the first run of the algorithm, winnowing will run slower than the simple k-shingling algorithm as the document is first hashed, then the sliding window is employed to pick the fingerprints. However, for repeated comparisons, given the fingerprints are stored somewhere, it will be faster, since there are typically much less fingerprints than there are shingles. It will also be faster than the abstract syntax tree fingerprinting considering the cost of traversing all of the sub-trees that will be found in the AST.

3.3 Abstract Syntax Tree Fingerprinting

Abstract syntax tree fingerprinting is a tree-based clone detection algorithm. Instead of preparing the document by getting rid of whitespace, punctuation, etc., the source code must be parsed to produce the abstract syntax tree. An abstract syntax tree represents the syntactic structure of a program, so it is intrinsically whitespace insensitive. It is also position independent as it looks at the syntactical structure of portions of the code, without regard to their location in the document.

After the abstract syntax tree is produced, the trees must be compared. Instead of taking the edit distance between the two trees, the sub-branches are instead hashed. Any branch with less than 5 nodes is ignored. The hash calculated is actually the hash of the concatenation of the node-type hash and the hash of the node-count vector. The node-type hash is simply the hash of the string representing the node-type, whether that is a "variable" node, "block statement" node, etc. The node count vector is the count of the types of nodes in the sub-tree. This set of hashes is saved as the set of fingerprints of the abstract syntax tree. These fingerprints are also compared using the Jaccard similarity index.

The abstract syntax tree will fundamentally run slower than the other algorithms explored because of the cost of traversing and hashing all of the sub-trees above the threshold. Constructing the abstract syntax tree is also much more expensive than constructing the k-shingles and k-words. While it may be more expensive, since the abstract syntax tree is a structural representation, it should be less sensitive to changes, such as reordering the sequence of statements.

While not as efficient, abstract syntax tree fingerprinting has more applications than the other methods. Since most refactoring tools are based on abstract syntax trees, direct clone removal is much easier when dealing with the abstract syntax tree representation. So, instead of completely removing duplicates files, the duplicate branches within the abstract syntax tree of the JavaScript file can be removed instead, and the code can be reconstructed without the duplicate branches.

As mentioned before, abstract syntax tree fingerprinting has one fatal flaw. Unlike the token-based techniques, which will work on any document in any language, abstract syntax tree fingerprinting will only work on languages for which the parser that is being used can understand. In this case, Esprima [6], a JavaScript parsing library was used to produce the abstract syntax trees. However, Esprima can only parse pure JavaScript, and it only has experimental support for JSX [11], a very popular syntax extension to JavaScript. Thus, Esprima could not parse all the files provided to it.

4 EVALUATION

This section details the results of exploring the different duplication detection algorithms. The three algorithms, k-shingling, winnowing and abstract syntax tree fingerprinting, were run on approximately 3,400 scripts from 100 websites. The websites were taken from the Alexa Top 10,000 internet sites. A man-in-the-middle proxy was used to intercept HTTP requests and save the files to a server. The Stanford MOSS was selected to be used as the baseline measure for accuracy and timing.

4.1 Stanford MOSS

The Stanford MOSS (Measure of Software Similarity), developed in 1994, is a web-based software system that is used to prevent plagiarism, typically in programming classes, by detecting the similarity of software programs. MOSS is specifically designed for software programs, and is compatible with a variety of languages, such as C++, Python, JavaScript, etc. MOSS works by using a more sophisticated form of the winnowing algorithm called "robust winnowing". In the typical

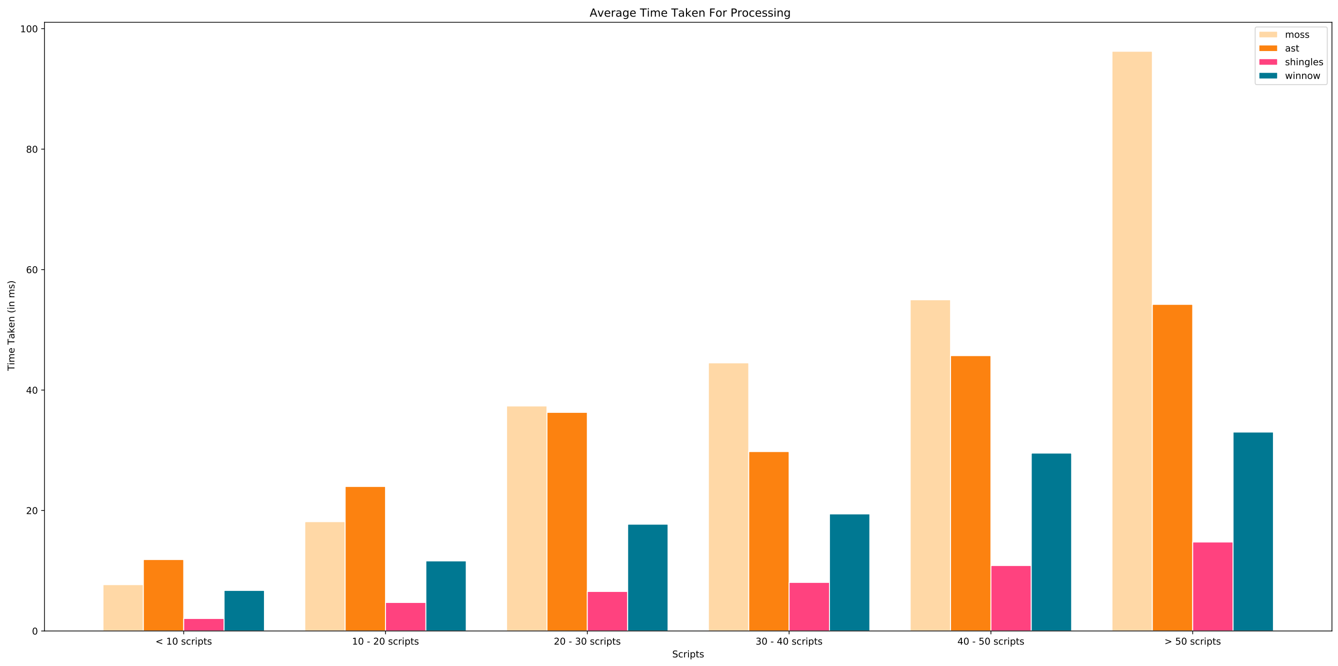

Figure 1: Bar-chart Comparing Algorithm Speeds

winnowing algorithm, the minimum is selected in each window, and if there is a tie, the right-most minimum is selected. In "robust winnowing", ties are broken by selecting the same hash as the window one position to the left, if possible. If it is not possible, the right-most minimal hash is selected [19]. MOSS is highly accurate, but, while the details of the implementation of winnowing algorithm are public, the tuning parameters are kept secret. Thus, MOSS's results are hard to reproduce. On the other hand, thanks to its availability as a free to use software and its reputation of being very accurate, MOSS is an excellent resource to test the accuracy of the proposed duplicate detection algorithms against as a baseline measure.

In response to a query, containing the files to be analyzed, sent by a user, the Stanford MOSS produces HTML pages where pairs of programs with similar code are listed. On these HTML pages, the number of lines of code in common between the pairs of files is reported, alongside the relative percentage of each file that the lines in common make up. To match the format of the MOSS scores, the duplicate detection algorithms were modified to not only give the similarity coefficient – they also provide the percentage of each file that the shingles and fingerprints in common make up. Code to record the time taken from the submission of the scripts until the final server response containing the results was injected into the MOSS submission script provided.

4.2 Run-time

Fig. 1 illustrates the average time taken by each algorithm to process and find the duplicates. The run-time was determined on an Intel dual-core processor architecture (2.7GHz) with 8 GB RAM running macOS Mojave. Once more than 20 scripts are being compared, all of the explored algorithms run faster than MOSS. This is because MOSS is an online service, so the Javacript files being compared must be uploaded to the server, and the user has to wait for a response. On the other hand, k-shingling, winnowing and abstract syntax tree fingerprinting were run on our local machine. For websites with over 50 scripts, MOSS took 100 ms, almost double the time that the second slowest algorithm took.

As expected, the abstract syntax tree fingerprinting had the worst run-time of the offline duplicate detection algorithms, with winnowing following after it, and shingling being the fastest. Winnowing is slower than shingling because, while shingling only hashes the tokens and compares the complete set, the winnowing algorithm goes over the token hashes and selects the fingerprints of the document before comparing the fingerprints.

4.3 Accuracy

As mentioned earlier, since the MOSS reports two separate scores and not just one similarity index, the duplicate detection algorithms also provide the percentage of each file that the shingles and fingerprints in common make up. Fig.2

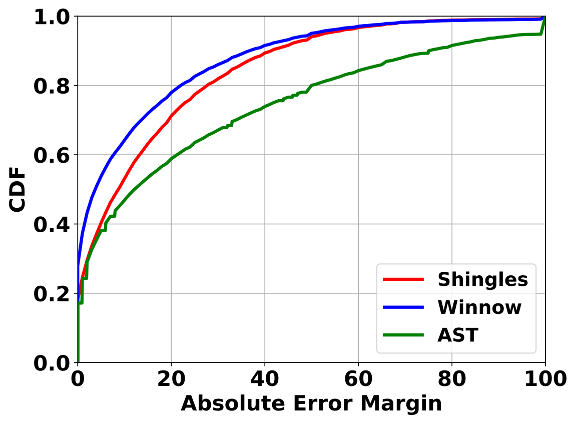

Figure 2: CDFs of Absolute Margin of Error

reports cumulative distribution function of the absolute margin of error of each algorithm when compared to MOSS's scores. The absolute margin of error is the absolute value of the algorithm's scores subtracted from MOSS's scores.

Winnowing has best accuracy – the median absolute error margin is about 5%. AST fingerprinting has the worst accuracy with a median of about 10%. Most of the difference in accuracy most likely stems from differences in the way MOSS prepares the data. Winnowing was expected to have the best performance because it is closest to the algorithm tht MOSS uses. However, exact replication of MOSS's performance is impossible, unless the algorithms used are the same with the same document preparation steps and tuning parameters, and MOSS's document preparation steps and tuning parameters are kept private. Shingling performed worse than winnowing, with a median of about 10%, but since it is faster, it might be worth sacrificing some of the accuracy for speed.

Abstract syntax tree fingerprinting has the worst scores because MOSS does not only check for structure and syntax, it also maintains some level of semantic information, such as language-specific keywords. From further examination of the tested JavaScript files, there are lines of code that are broken down into equal abstract syntax trees, but they are not MOSS matches. As hypothesized, the abstract syntax tree method does report many more false-positives than any other method.

One thing to note about the accuracy scores reported here is that the percentages between MOSS and the algorithms also differ because MOSS does not work with sets like the algorithms do. If a line shows up twice in a JavaScript files, MOSS reports it as a duplicate twice, maintaining the exact number of lines that are in the original document. If a shingle or word or sub-tree hash appears more than once, any occurrence after the first is discarded, leading to the overall document to be reduced in size, as the sets only contain unique hashes. Naturally, this leads to discrepancies in the percentage scores. However, the MOSS scores can still serve as a good guideline of general correctness.

4.4 CPU Usage

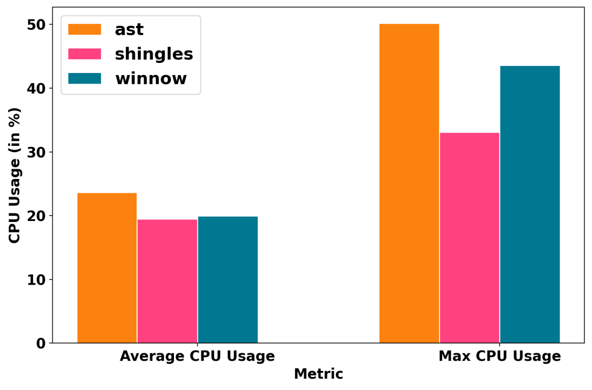

Figure 3: Bar-chart Comparing CPU Usage

CPU usage, charted in Fig. 3 is another important metric, especially if one of these algorithms is to be used in a tool whose job is simplifying JavaScript and HTML pages to speed up their load times. As expected from both the background research and the run-time graphs, abstract syntax tree sub-tree traversal and hashing is much more CPU intensive than just hashing a linear sliding window of tokens. The average CPU usage for abstract syntax tree fingerprinting is about 10% higher than it is for shingling. Winnowing, while working off the same principle of a linear sliding window of tokens, also does more computations by virtue of picking the minimal hash in each window after computing all the hashes of the tokens of the document. The average CPU usage of winnowing is only about 2% higher than that of shingling, but the maximum CPU usage is about 10% higher than the shingling algorithm's maximum CPU usage.

The CPU utilization could not be compared to MOSS's because MOSS runs on a private web server, so it was not possible to obtain this metric.

4.5 Memory Usage

Fig. 4 which charts the memory usage of all three algorithms reports some interesting and unexpected behaviour. Unlike in all the other graphs, the abstract syntax tree fingerprinting is not the worst performing algorithm. The high memory usage for winnowing may be due to keeping a large number of shingles in memory while picking the fingerprints. However, if one only examines the average memory usage, it is

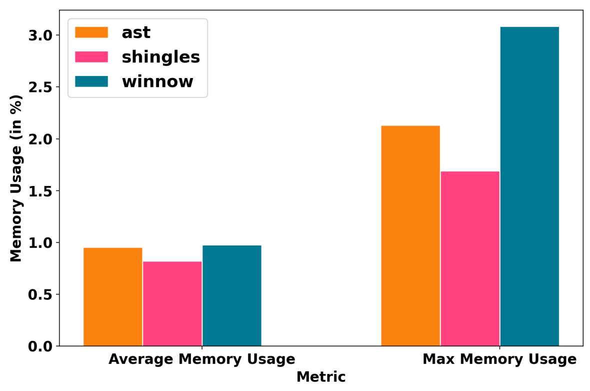

Figure 4: Bar-chart Comparing Memory Usage

equal to that of the abstract syntax tree's average memory usage. This may imply that the maximum memory usage reported for the winnowing algorithm may be an outlier. All of the algorithms only use about 1.0% of RAM on average, which means they are comparably light-weight and not very memory intensive.

The memory utilization could not be compared to MOSS's because MOSS runs on a private web server, so it was not possible to obtain this metric.

5 CONCLUSION

In this paper, three algorithms for document similarity detection were examined and compared using a dataset of 100 websites and about 3,400 scripts. These algorithms are k-shingling, winnowing and abstract tree fingerprinting. In k-shingling, the document is broken up into tokens, then those tokens are hashed and compared. In winnowing, instead of comparing all of the token hashes, a sliding window is used to select a set of hashes to represent the whole document as its fingerprints. Finally, abstract syntax tree fingerprinting considers the hashes of the abstract syntax tree of the source code. These algorithms were evaluated in four categories: run-time, accuracy, CPU usage and memory usage.

Overall, for general use, a clear winner cannot be decided. These algorithms each have their own strengths and weaknesses. However, if the algorithm is meant to be incorporated in any tools that want to remain light-weight and fast, abstract syntax tree fingerprinting is not a viable option because of its run-time and CPU usage. While shingling is faster and uses less memory than winnowing, there exists a trade-off as it is less accurate. The use of either shingling or winnowing is wholly dependent on whether a faster or more accurate solution is desired. Since winnowing typically takes double the time that shingling does, this is a very important consideration to make.

While abstract syntax trees may not be the best route to take when looking for fast and non-intensive duplicate detection, there are other uses. During this research, the idea of using abstract syntax trees to find and track dependencies between functions in JavaScript files was explored.

REFERENCES

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377.

[2] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33 (07 2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[3] Michel Chilowicz and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. 243–247. https://doi.org/10.1109/ICPC.2009.5090050

[4] CloneDR. [n.d.]. http://www.semanticdesigns.com/Products/Clone/

[5] Giuseppe Di Lucca, Massimiliano Di Penta, and Anna Fasolino. 2002. An Approach to Identify Duplicated Web Pages. *Proceedings - IEEE Computer Society's International Computer Software and Applications Conference*, 481–486. https://doi.org/10.1109/CMPSAC.2002.1045051

[6] Esprima. [n.d.]. https://esprima.org/

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[8] Mohammad Ghasemisharif, Peter Snyder, Andrius Aucinas, and Benjamin Livshits. 2018. SpeedReader: Reader Mode Made Fast and Private. arXiv:cs.IR/1811.03661

[9] Neline Ginkel, Willem Groef, Fabio Massacci, and Frank Piessens. 2019. A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries. *Security and Communication Networks* 2019 (05 2019), 1–21. https://doi.org/10.1155/2019/9629034

[10] jQuery. [n.d.]. https://jquery.com/

[11] JSX. [n.d.]. https://facebook.github.io/jsx/

[12] Kostas Kontogiannis, Renato De Mori, Ettore Merlo, M. Galler, and Morris Bernstein. 1996. Pattern Matching for Clone and Concept Detection. *Autom. Softw. Eng.* 3 (06 1996), 77–108. https://doi.org/10.1007/BF00126960

[13] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 253–262. https://doi.org/10.1109/WCRE.2006.18

[14] J. Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. 301–309.

[15] F. Lanubile and T. Mallardo. 2003. Finding function clones in Web applications. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* 379–386.

[16] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. *ArXiv* abs/1811.00918 (2017).

[17] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard Coefficient for Keywords Similarity.

[18] Lutz Prechelt and Guido Malpohl. 2003. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science* 8 (03 2003).

[19] Saul Schleimer, Daniel Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. *Proceedings of the ACM SIGMOD International Conference on Management of Data* 10 (04 2003). https://doi.org/10.1145/872757.872770

[20] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. 2008. Web Browser as an Application Platform. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*. 293–302.

[21] Michael Wise. 1993. String Similarity via Greedy String Tiling and Running KarpRabin Matching. *Unpublished Basser Department of Computer Science Report* (01 1993).

[22] Chuan Yue and Haining Wang. 2009. Characterizing Insecure Javascript Practices on the Web. In *Proceedings of the 18th International Conference on World Wide Web* (Madrid, Spain) *(WWW '09)*. ACM, New York, NY, USA, 961–970. https://doi.org/10.1145/1526709.1526838

[23] Yasir Zaki, Jay Chen, Thomas Pötsch, Talal Ahmad, and Lakshminarayanan Subramanian. 2014. Dissecting Web Latency in Ghana. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 241–248. https://doi.org/10.1145/2663716.2663748