Scalable Code Clone Detection and Search based on Adaptive Prefix Filtering

Manziba Akanda Nishi[a], Kostadin Damevski[a]

[a]*Department of Computer Science, Virginia Commonwealth University*

Abstract

Code clone detection is a well-known software engineering problem that aims to detect all the groups of code blocks or code fragments that are functionally equivalent in a code base. It has numerous and wide ranging important uses in areas such as software metrics, plagiarism detection, aspect mining, copyright infringement investigation, code compaction, virus detection, and detecting bugs. A scalable code clone detection technique, able to process large source code repositories, is crucial in the context of multi-project or Internet-scale code clone detection scenarios. In this paper, we focus on improving the scalability of code clone detection, relative to current state of the art techniques. Our adaptive prefix filtering technique improves the performance of code clone detection for many common execution parameters, when tested on common benchmarks. The experimental results exhibit improvements for commonly used similarity thresholds of between 40% and 80%, in the best case decreasing the execution time up to 11% and increasing the number of filtered candidates up to 63% .

Keywords: code clone detection, prefix filtering, software maintenance
2010 MSC: 00-01, 99-00

1. Introduction

Developers introduce clones in a code base mainly when reusing existing code blocks without significant alteration, or when certain code blocks are im-

Email addresses: nishima@mymail.vcu.edu (Manziba Akanda Nishi), kdamevski@vcu.edu (Kostadin Damevski)

plemented by developers following a common mental macro [1, 2, 3]. Researchers have shown that developers tend to perform software maintenance tasks more effectively when they have the results of code clone detection [4, 5]. Empirical studies have noted that code clones are widespread, and that a significant portion of source code (between 5% and 20%) is copied or modified from already implemented code fragments or blocks [1, 6, 7].

Performing code clone detection across numerous software repositories is a common use case. Specific applications for large scale code clone detection include querying library candidates [8], categorizing copyright infringement and license violations [9, 10], plagiarism detection [10, 9], finding product lines in reverse engineering [11, 10], tracing the origin of a component [12], searching for code blocks in large software repositories [13, 14], and spotting analogous applications in Android markets [15, 16]. However, most existing code clone detection techniques have difficulty scaling up to extremely large collections of source code [17, 18].

Among the tools aimed towards large scale code clone detection, a common limitation is in the complexity of differences among clones they can detect. For instance, scalable token based approaches [3, 19, 8] have difficulty detecting near miss (Type-3) code clones, which can occur more frequently than other types of clones [18, 20, 21]. Parallel and distributed clone detection techniques like D-CCFinder [22] can be more burdensome to manage, requiring specialized hardware or software support, while tree based code clone detection technique, such as Deckard [23], place higher demands on memory.

In this paper, we propose a token-based code clone detection technique aimed at scalability and detecting Type-3 clones, consisting of two main steps: filtering and verification. In the filtering step we aim to significantly reduce the number of code blocks for comparison, removing from consideration blocks that do not have any possibility of being code clones. In the verification step we determine whether candidate pairs that survived the filtering step are really code clones. This two step process greatly reduces the runtime of code clone detection, allowing the technique to scale up to very large corpora. This technique is based

on the *adaptive prefix filtering heuristic* [24], which is an extended version of *prefix filtering heuristic* [25] [26] previously applied towards code clone detection in SourcererCC [16]. To our knowledge, SourcererCC is the best scaling code clone detection tool able to detect Type-3 clones. In this paper, we demonstrate improvements in execution time relative to SourcererCC, while obtaining the same accuracy, for many common similarity thresholds, when evaluated on a large scale inter-project source code corpus [27].

A separate novel idea presented in this paper is the effective application of our technique to code clone search, without modification, in addition to code clone detection. Code clone search is a related problem to code clone detection where the user specifies a single code block (i.e. query block) to search for in a corpus of many code blocks. Once indexed, the corpus should be able to serve numerous such queries. Ours is among few techniques that can be applied to both of these problems. The contributions of this paper are the following:

- A novel code clone detection technique that can scale to very large scale source code repositories (or sets of repositories) with the ability to detect Type-1, Type-2, and Type-3 code clones, while maintaining high precision and recall.
- An extension of our proposed technique so that it can be effectively utilized for code clone search without modification.

We have organized the rest of this paper as follows. Section 2 describes the background and related work in both code clone detection and code clone search. Section 3 describes the adaptive prefix filtering heuristic, which we utilized in our code clone detection approach. Section 4 describes the design of a code clone detection system based on our technique, which includes several necessary optimization steps. Section 5 describes the experimental results evaluating our proposed approach, as well as a comparison to the recent code clone detection tool SourcererCC. Section 6 concludes the paper, summarizing it's contributions.

2. Background and Related Work

Based on the nature of the similarity between code blocks, the software engineering community has identified four types of code clones, by which code clone detection techniques can be organized. Syntactically equivalent code blocks are called Type-1 clones. Type-2 code clones are code blocks that are syntactically comparable but are slightly contrasting in terms of variable names, function names, or identifier names. If two code blocks contain statements that have been inserted, altered, expunged, there is a gap in statements, or statement order differs, then these are called Type-3 clones. Semantically equivalent code blocks are called Type-4 clones [16].

Scaling code clone detection to work across multiple repositories is a specific area of interest among researchers. Popular and notable examples of large-scale code clone detection include CCFinderX [3], which is one of the foremost token based code clone detection tools able to scale up to large repositories and detect Type-1 and Type-2 code clones. In [19], an inverted index-based approach was first proposed, detecting Type-1 and Type-2 clones. Deckard [23] is another tool that aims to scale to large source code repositories. It uses a tree-based data structure and detects clones by identifying similar subtrees and is able to detect up to Type-3 code clones. NiCad [28] is a scalable code clone detection tool that can detect Type-3 clones using a technique based on parsing, normalization and filtering. When compared using similar execution parameters, CCFinderX scales up to 100 million lines of code, Nicad scales up to 10 million lines of code, while Deckard scales up to 1 million lines of code [16].

Parallel, distributed or online (i.e. incremental) techniques add another dimension in examining scalable code clone detection technique. For instance, iClone [29] is the first incremental code clone detection technique that detects code clones in the current version of code repository by leveraging executions on previous versions of the same repository. It uses a suffix tree-based and token-based approach that can detect Type-1 and Type-2 code clones. A scalable distributed code clone detection tool named D-CCFinder was proposed in [22],

which can scale code clone detection in a distributed environment to detect Type-1 and Type-2 clones. In [17] a scalable code clone detection technique has been introduced where input files are partitioned into smaller subsets and a shuffling framework is utilized to allow the clone detection tool to execute on each of the subsets separately, enabling it to detect code clones in parallel.

Recently, the SourcererCC [16] code clone detection tool proposed a token-based prefix filtering code clone detection technique, which greatly reduces the number of candidate code clone pairs, enabling it to detect up to Type-3 clones in Internet-scale source code repositories. SourcererCC is the best scaling tool on a single machine that we are aware of. The approach described in this paper extends SourcererCC with an adaptive approach that allows for even greater gains in performance for large-scale source code datasets.

SourcererCC is based on two filtering heuristics, prefix filtering and token position filtering, which reduce the number of candidate pairs that require costly pairwise comparison of all of their tokens [25, 26]. These two filtering heuristics attempt to rapidly, with few token comparisons, detect pairs of code blocks that diverge very significantly from each other. To perform this task, a subset (or prefix) is isolated in each of the two code blocks, where if there are no matching tokens in the subsets then we can safely reject them as a candidate pair, without proceeding further, and without attempting to compare all of their tokens. On the other hand, a single matching token in the subset allows the pair to proceed to further scrutiny as a code clone.

Recently, an extremely scalable code clone detection tool VUDDY [30] has been presented. VUDDY's purpose is to detect vulnerable code clones for security improvement. While VUDDY has been shown to be significantly faster than SourcererCC it is designed to only detect Type-1 and Type-2 code clones. So far, the precision and recall of VUDDY has only been evaluated for relatively few instances of code with security vulnerabilities.

Wang et al. [24] recently proposed an additional filtering heuristic to those used in SourcererCC, called *adaptive prefix filtering*. This technique posits that deeper prefix lengths, which attempt more aggressive filtering at a higher perfor-

mance cost, can achieve good performance on some types of input. An adaptive prefix filtering technique attempts to estimate the right level of filtering for each candidate by optimizing the trade-off between the cost of deeper filtering with the benefit of reducing the number of candidates. This paper applies adaptive prefix filtering to code clone detection, and evaluates it's adequacy.

2.1. Code Clone Search

Code clone search is a related area of research where a single code block is supplied as a query to be matched in large source code corpus, retrieving a list of code clones. Code clone search requires the source code to be pre-indexed, and for the similarity threshold between the query block and it's clones to be specified at query time, instead of, at indexing time. This twist makes it challenging for typical code clone detection to be used without modification, and requires a more flexible data structure. In this section we describe some of the previous approaches directed towards code clone search, which are significantly fewer than those that target code clone detection. Our technique targets both code clone detection and code clone search.

Multidimensional indexing structures have previously been proposed for code clone search. The technique proposed by Lee et al. [31] captures semantic information via a characteristic vector containing the occurrence counter of each syntactic element. This technique also retrieves a ranked list of code clones for each query, similar to typical information retrieval techniques. Another code clone search approach that uses multi-level indexing to detect Type-1,Type-2, Type-3 code clone was proposed by Kelvanloo et al. [14].

SeClone is a technique that targets code clone search [32] using a complex workflow, which includes the creation of Java ASTs for each file, two types of indices (code pattern index and type usage index), a tailored search algorithm for code search, and a post-processing step that leverages clone pair clustering and grouping. Another complex technique Internet scale code clone search technique has been recently proposed, performing an abstracted code search for working code examples for reuse [33]. This technique uses an abstract representation of

code snippets (using p-strings[34]), which are searched using frequent itemset mining. The result set of working code example is ranked using relevance of code patterns, popularity of abstract solutions, and similarity of code snippets to the output code snippets [33].

A clone search technique proposed by Koschke et al. [35] detects clones between two different systems. The technique uses a suffix tree to represent the code base that is smaller between the two. Subsequently, every file of the other system is compared against the suffix tree. A hash-based technique is additionally used for reducing the number of file comparisons. The technique only detects Type-1 and Type-2 code clones.

All of the above approaches towards code clone search utilize complex data structures to represent code blocks and (or) leverage complicated indexing schemes. Different from code clone detection, code clone search techniques typically do not optimize the index building time, since it is considered to be a fixed cost. Our proposed approach applies to both code clone search and code clone detection. Unlike other approaches for code clone search, it uses simple token-based code block representation that is fast and scalable to build and also maintains the ability to produce reasonable detection accuracy for Type-1, Type-2 and Type-3 code clones.

3. Adaptive Prefix Filtering Technique

This paper centers on the combination of three filtering heuristics, two of which have been previously proposed for token-based code clone detection: *prefix filtering*, *token position filtering* and *adaptive prefix filtering*. Prefix filtering was proposed by researchers so that number of candidate pairs in token based code clone detection can be significantly reduced, improving performance and scalability. As a complement to prefix filtering, token position filtering utilizes the position of the tokens in the code block to further reduce the number of candidate pairs. The third heuristic, adaptive prefix filtering is an extended version of prefix filtering that looks for beneficial opportunities to filter candi-

date pairs even more aggressively, aimed at even further improving performance and scalability to large datasets. SourcererCC [16] has implemented the first two heuristics, while in our proposed code clone detection technique *we have applied all three heuristics*. In this section, we describe the context for token-based code clone detection, followed by an explanation of each of the three filtering heuristics. As a running example that helps clearly explain our approach we use 5 code blocks shown in Table 1.

```
//Code Block 1 (CB1)
public static int factorial(int result) {
  if(result <= 1) return 1;
  return result * factorial(result-1); }
```

```
//Code Block 2 (CB2)
public static int factorial(int n) {
  int result = 1;
  for(int i=1; i<=n; i++) {
    result = result * i;
  }
  return result; }
```

```
//Code Block 3 (CB3)
public static int factorial(int n) {
  if(n >= 0) {
    result[0] = 1;
    for(int i=1; i<=n; i++) {
      result[i] = i * result[i-1];
    }
    return result[n]; } }
```

```
//Code Block 4 (CB4)
public static void main(String[] args) {
  int result = 5;
  int factorial = result;
  for(int i=result-1; i>1; i--) {
    factorial = factorial * i; } }
```

```
//Code Block 5 (CB5)
public int factorial(int result) {
  if(result == 0) {
    return 1;
  } else {
    return result * factorial(result-1); } }
```

Table 1: Code blocks for running example.

For a token-based approach of code clone detection, at first, we must convert the source code into a set of tokens. To this end, we extract the set of string literals, keywords, and identifiers in each code block, removing all the special characters, operators and comments. Each of the extracted tokens can occur

Code Block	**Sorted Tokens**
CB_1	$if^{\{9\}}_{\{1,3\}}$ $static^{\{10\}}_{\{1,4\}}$ $public^{\{11\}}_{\{1,5\}}$ $return^{\{13\}}_{\{2,6\}}$ $factorial^{\{14\}}_{\{2,9\}}$ $1^{\{15\}}_{\{3,12\}}$ $int^{\{17\}}_{\{2,14\}}$ $result^{\{18\}}_{\{4,19\}}$
CB_2	$for^{\{8\}}_{\{1,3\}}$ $static^{\{10\}}_{\{1,4\}}$ $public^{\{11\}}_{\{1,5\}}$ $n^{\{12\}}_{\{2,6\}}$ $return^{\{13\}}_{\{1,6\}}$ $factorial^{\{14\}}_{\{1,9\}}$ $1^{\{15\}}_{\{2,12\}}$ $i^{\{16\}}_{\{4,14\}}$ $int^{\{17\}}_{\{4,14\}}$ $result^{\{18\}}_{\{4,19\}}$
CB_3	$0^{\{7\}}_{\{2,3\}}$ $for^{\{8\}}_{\{1,3\}}$ $if^{\{9\}}_{\{1,3\}}$ $static^{\{10\}}_{\{1,4\}}$ $public^{\{11\}}_{\{1,5\}}$ $n^{\{12\}}_{\{4,6\}}$ $return^{\{13\}}_{\{1,6\}}$ $factorial^{\{14\}}_{\{1,9\}}$ $1^{\{15\}}_{\{3,12\}}$ $i^{\{16\}}_{\{6,14\}}$ $int^{\{17\}}_{\{3,14\}}$ $result^{\{18\}}_{\{4,19\}}$
CB_4	$5^{\{1\}}_{\{1,1\}}$ $String^{\{2\}}_{\{1,1\}}$ $args^{\{3\}}_{\{1,1\}}$ $main^{\{5\}}_{\{1,1\}}$ $void^{\{6\}}_{\{1,1\}}$ $for^{\{8\}}_{\{1,3\}}$ $static^{\{10\}}_{\{1,4\}}$ $public^{\{11\}}_{\{1,5\}}$ $factorial^{\{14\}}_{\{3,9\}}$ $1^{\{15\}}_{\{2,12\}}$ $i^{\{16\}}_{\{4,14\}}$ $int^{\{17\}}_{\{3,14\}}$ $result^{\{18\}}_{\{3,19\}}$
CB_5	$else^{\{4\}}_{\{1,1\}}$ $0^{\{7\}}_{\{1,3\}}$ $if^{\{9\}}_{\{1,3\}}$ $public^{\{11\}}_{\{1,5\}}$ $return^{\{13\}}_{\{2,6\}}$ $factorial^{\{14\}}_{\{2,9\}}$ $1^{\{15\}}_{\{2,12\}}$ $int^{\{17\}}_{\{2,14\}}$ $result^{\{18\}}_{\{4,19\}}$

Table 2: Tokenized code blocks sorted based on global token ordering. Each token x is represented as $x^{\{m\}}_{\{l,g\}}$ where m is the global position (the position after sorting all the tokens of all the code blocks based on global frequency), l is the local frequency of the token in the code block, while g is the global frequency of token x in the corpus

several times in a code block so we annotate each token with it's local occurrence frequency. The sum of the local occurrence frequencies for a specific token across all the code blocks in the corpus is it's global occurrence frequency. We sort all the tokens, in each code block, based their global frequency in ascending order. When there is a tie, the tokens are arranged in alphabetical order. For instance, in Table 2 we show the sorted token-based representation of the code blocks from Table 1.

In token-based code clone detection techniques, two code blocks are classified as clones if they match a certain number of tokens. The required number of matched tokens depends on the user defined threshold value θ, which ranges between 1 and 10 (i.e. 10% to 100%), and the length of the code blocks. A similarity function (e.g. edit, hamming) measures the degree of similarity between two code blocks. In this paper we use the simple overlap similarity function that measures the number of tokens in the intersection between two code blocks, i.e. $\mathcal{O}(CB_1, CB_2) = |CB_1 \cap CB_2|$.

To determine whether code blocks CB_1 and CB_2 are code clones, we take the maximum value between the number of tokens in each of the code blocks

CB_1 and CB_2, $t = max\{length(CB_1), length(CB_2)\}$, and multiply it by the similarity threshold value, θ. A code clone detection technique has to examine at least $i = \lceil \theta|t| \rceil$ tokens to determine whether a pair of code blocks are code clones. There is a significant performance cost in performing this token comparison for a large source code repository containing numerous code blocks. The prefix filtering heuristic improves on this by first determining if a strict comparison is necessary, or if the pair of blocks can be filtered out. We describe this technique in more detail next.

3.1. Prefix Filtering

According to this heuristic, if the sorted tokens (as shown in Table 2) of a pair of code blocks *match at least one token in their prefix* then these blocks are a code clone candidate pair, which we verify whether are really code clone to each other in a subsequent verification step. On the other hand, if not a single token is matched, then we can discard them from further consideration.

The prefix of each code block is of size $|t| - \lceil \theta|t| \rceil + 1$ where $|t|$ is the total number of tokens in the corresponding code blocks. In Table 3 we show the prefixes of two code blocks from our running example, assuming a similarity threshold value θ=0.8. For CB_1, whose size is $|t| = 16$, Table 3 shows that it's prefix is computed as $|t| - \lceil \theta|t| \rceil + 1 = 16 - 0.8 * 16 + 1 = 4$. In the fourth column of this table all the extracted tokens within the prefix of CB_1 are shown. It is worth mentioning that although the prefix length is four we have extracted five tokens because this metric ignores repeated tokens in the last position. Since *return*(last token with 1-prefix length) has repeated twice in CB_1 it is similarly included in the prefix. Now, we only test whether at least one token is matched in the two prefixes to determine if they need further scrutiny or if they can be filtered out. We generalize this to Property 1 as follows.

Property 1: *If two code blocks CB_1 and CB_2 consist of t terms each, which follow an order O, and if $|CB_1 \cap CB_2| \geq i$, then the sub-block CB_{sb1} consisting of the first $t - i + 1$ terms of CB_1 and the sub-block CB_{sb2} consisting of first $t - i + 1$ terms of CB_2 must match at least one token.*

Code Block	Size of Code Block	1-Prefix Scheme	Tokens within Prefix
CB_1	$\|t\| = 16$	$\|t\| - \lceil\theta\|t\|\rceil + 1 = 16-0.8*16+1 = 4$	{if static public return return}
CB_2	$\|t\| = 21$	$\|t\| - \lceil\theta\|t\|\rceil + 1 = 21-0.8*21+1 = 5$	{for static public n n}

Table 3: Prefix filtering of CB_1 and CB_2

3.2. Token Position Filtering

For the second heuristic, token position filtering, we derive an upper bound by summing of the number of current matched tokens and minimum number of unseen tokens between two code blocks. If this upper bound is smaller than the needed threshold we can safely reject this code block pair.

For code blocks CB_1 and CB_2, if we examine the position of the first matching token *static*, we see that it is in position 2 for both code blocks. At position 2, the minimum number of unseen tokens is 14 for code block CB_1 and 19 for code block CB_2, since the total number of tokens is 16 and 21 for code blocks CB_1 and CB_2 respectively. We compute the upper bound between CB_1 and CB_2 as $\{1 + min(14, 19)\} = 15$, to communicate the number of matching tokens, in the best case scenario, given the position of the match in the *static* token. The upper bound in this case is lower than the required number of tokens (17) we needed to match, assuming the similarity threshold value of 0.8 (θ*$\{length(CB_1), length(CB_2)\}$=$0.8 * max(16, 21) = 17$). Therefore, we can reject this pair of code blocks without proceeding further. We derive property 2 for token position filtering, as follows.

Property 2: *Let blocks CB_1 and CB_2 be ordered and $\exists$ token t at index i in CB_1, such that CB_1 is divided in to two parts, where $CB_1(first) = CB_1[1...(i-1)]$ and $CB_1(second) = CB_1[i...(|CB_1|)]$. Now if $|CB_1| \cap |CB_2| \geq \theta * max(|CB_1|, |CB_2|)$, then $\forall\ t \in (CB_1 \cap CB_2)$, $|CB_1(first) \cap CB_2(first)|$ $+ min(|CB_1(second)|,$*
$|CB_2(second)|) \geq \theta * max(|CB_1|, |CB_2|)$.

3.3. Adaptive Prefix Filtering

Adaptive prefix filtering heuristic is the extended version of prefix filtering where rather than matching one token in the prefixes of code blocks, we match more, while deepening the size of the prefixes. For example, instead of matching only the token *static* in the prefix of CB_1 and CB_2 we can match multiple tokens. This variant defines an ℓ-prefix (instead of a 1-prefix scheme), where ℓ is the number of tokens we want to match. The size of the prefix (i.e. number of tokens within prefix) changes from $|t| - \lceil \theta|t| \rceil + 1$ to $|t| - \lceil \theta|t| \rceil + \ell$. Adaptive prefix filtering reduces the number of candidates more aggressively, at the cost of more comparisons in the filtering step. An advantageous value of ℓ can be selected based on the cost calculation framework for each of the code block, discussed in Section 4.2.

In Table 4 we have shown how the 1-prefix scheme in Table 3 can be prolonged to 2-prefix and 3-prefix schemes. For instance, for CB_1 in Table 4 the length of 2-prefix scheme is $|t| - \lceil \theta|t| \rceil + 2 = 16 - 0.8 * 16 + 2 = 5$ resulting in the shown set of tokens.

Code Block	**2-Prefix Scheme**	**Tokens within 2-Prefix Scheme**	**3-Prefix Scheme**	**Tokens within 3-Prefix Scheme**
CB_1	$\|t\| - \lceil \theta\|t\| \rceil + 2 = 16 - 0.8 * 16 + 2 = 5$	{if static public return return factorial factorial}	$\|t\| - \lceil \theta\|t\| \rceil + 3 = 16 - 0.8 * 16 + 3 = 6$	{if static public return return factorial factorial 1 1 1}
CB_2	$\|t\| - \lceil \theta\|t\| \rceil + 2 = 21 - 0.8 * 21 + 2 = 6$	{for static public n n return}	$\|t\| - \lceil \theta\|t\| \rceil + 3 = 21 - 0.8 * 21 + 3 = 7$	{for static public n n return factorial}

Table 4: Adaptive prefix filtering for CB_1 and CB_2

Here, *factorial* is the new token added in 2-prefix scheme which was not included in 1-prefix scheme. For the 2-prefix scheme of CB_1 and CB_2, the number of similar tokens between the prefixes is 3. These similar tokens are *static*, *public* and *return*. It is worth mentioning that although *return* has repeated two times in CB_1 it has occurred only once in CB_2 so we have to count it only once. According to adaptive prefix filtering, for CB_1 if we take 2-prefix scheme then

we will keep CB_1 and CB_2 as candidate pairs because at least ℓ=2 tokens are matched in the prefixes of CB_1 and CB_2. This is generalized in Property 3 for adaptive prefix filtering and Lemma 1, which assert that we can freely use an ℓ-prefix scheme in place of a 1-prefix scheme.

Property 3: *If blocks CB_1 and CB_2 consisting of t terms each, which follow an order O, and if $|CB_1 \cap CB_2| \geq i$, then the sub-block CB_{sb1} consisting of the first $t - i + \ell$ terms of CB_1 and the sub-block CB_{sb2} consisting of first $t - i + \ell$ terms of CB_2 will match at least ℓ tokens.*

Lemma 1: *For any code block pair (CB_1, CB_2) if $P_\ell(CB_1) \cap P_\ell(CB_2) < \ell$ then $|CB_1 \cap CB_2| < \lceil \theta|t| \rceil$ where $t = max\{length(CB_1), length(CB_2)\}$ and θ is user defined threshold value [24].*

Here $P_\ell(CB_1)$ and $P_\ell(CB_2)$ denote the ℓ-prefix set of CB_1 and CB_2 which are the subsets of CB_1 and CB_2 respectively, and where each subset consists of the first $|t| - \lceil \theta|t| \rceil + \ell$ elements.

4. System Design for Adaptive Prefix Filtering

In this section, we describe how a system that uses adaptive prefix filtering can be implemented in practice, which is crucial to making this technique useful. A *delta inverted index* presents an efficient data structure for clone candidate filtering, while *prefix cost calculation* is important in determining the size of the prefix ℓ that optimizes the tradeoff between greater reduction in candidate pairs and the added cost of deeper filtering. Both the data structure and cost calculation are crucial steps in implementing the adaptive prefix filtering heuristic in practice.

4.1. Delta Inverted Index

An inverted index data structure is commonly used to retrieve matching documents (i.e. code blocks) using a particular token as a query (implemented in popular tools such as Apache Lucene [36]). For prefix filtering and token position filtering only a single inverted index data structure is required. Instead

of creating index for each document inverted index is creates based on each token where it stores all the documents which contain that particular token. Thats why it is named as inverted index. However, adaptive prefix filtering requires a separate inverted index for each of the ℓ prefix schemes. A delta inverted index [24] overcomes the repetition that would occur if a simple set of inverted indices were used for adaptive prefix filtering. We describe this data structure.

The requirement for inverted list $I_\ell(e)$ is to store all the code blocks whose ℓ-prefix set contains the token e. Similarly, $I_{\ell+1}(e)$ stores all code blocks whose $\ell+1$ prefix set contains e, and $I_\ell(e) \subseteq I_{\ell+1}(e)$. A delta inverted index $\Delta I_{\ell+1}(e)$ data structure eliminates repetition by only storing the different code blocks between $I_\ell(e)$ and $I_{\ell+1}(e)$. At the outset, inverted index $\Delta I_1(e) = I_1(e)$, and as l increases we create delta inverted indexes $\Delta I_2(e)$, $\Delta I_3(e)$, ..., $\Delta I_t(e)$ for $I_1(e)$, $I_2(e)$, $I_3(e)$, ..., $I_t(e)$ where ($1 \leq \ell \leq$t-1).

The delta inverted index up to $\ell = 3$ is shown in Figure 1 for threshold value 0.8. Each of the three large boxes, ΔI_1, ΔI_2 and ΔI_3, represents a delta inverted index that can be queried separately, which contains a set of tokens mapped to their respective containing code blocks. ΔI_2 contains code blocks that are not present in I_1 but present in I_2. Index ΔI_3 contains code blocks that are not in I_1 and I_2 but are present in I_3. As an example, for token *return* inverted index $I_1(return)$ contains CB_1 and inverted index $I_2(return)$ contains CB_1, CB_2 and CB_5, while delta inverted index $\Delta I_2(return)$ contains CB_2 and CB_5.

4.2. Cost Calculation

In order to select the appropriate prefix length for each of the code blocks, we need to optimize the trade-off between filtering cost (i.e. the cost of looking up tokens deeper in the delta inverted index) and verification cost (i.e. the cost of determining if a pair of code blocks are actual clones by comparing all of the necessary tokens). The adaptive prefix filtering technique iteratively estimates the cost for ℓ-prefix scheme and $\ell+1$-prefix scheme, and if $\ell+1$-prefix

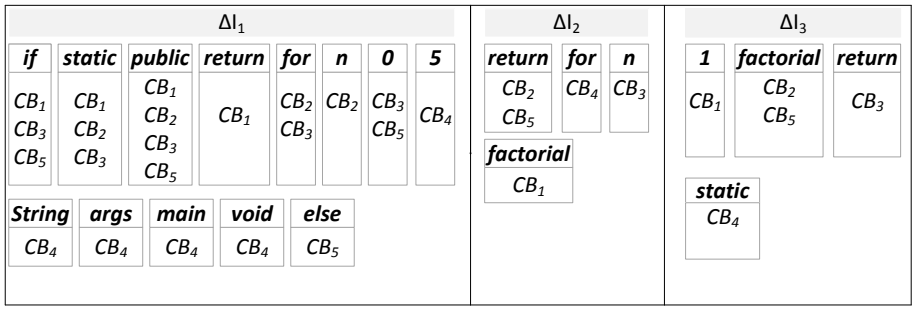

Figure 1: *Delta inverted index data structure for CB_1 to CB_5*

scheme's cost is greater than ℓ-prefix scheme cost then we select ℓ-prefix scheme for that particular code block. Otherwise we continue to compute the next prefix scheme's cost. Prior results include the fact that this technique selects a global minimum for the optimal prefix scheme [24]. Algorithm 1 lists steps of cost calculation which is integral part of adaptive prefix filtering technique. Without cost calculation this technique is not adaptive at all.

Suppose that for each code block $CB \in R$, where R is the repository of all code blocks, $F_\ell(CB)$ is the filtering cost and $V_\ell(CB)$ is verification cost. Therefore, for code block CB, we can derive following general equation for the cost:

$$Total\,Cost_\ell = F_\ell(CB) + V_\ell(CB) \tag{1}$$

For each code block $CB \in R$, it is necessary to query the inverted list of each token $e \in P_\ell(CB)$, where $P_\ell(CB)$ denotes the prefix set for the ℓ prefix scheme for code block CB. We introduce a new notation $\Phi_\ell(CB)$ to denote the set of all delta inverted indices for ℓ-prefix scheme used in the filtering step of code block CB such that $\Phi_\ell(CB) = \{\Delta I_i(e) | e \in P_\ell(CB), 1 \leq i \leq \ell\}$. Similarly, let $\Delta\Phi_\ell(CB)$ denote the set of additional delta inverted lists to be processed in ℓ prefix scheme comparing to $\ell - 1$ prefix scheme. Assume $C_\ell(CB)$ is the candidate set of code block CB, containing the code blocks that appear at least ℓ number of inverted lists of the elements in $P_\ell(CB)$.

Also, let $cost_v(CB)$ be the average cost of verifying the candidate CB. Us-

ing these abstractions, we can derive filter cost and verification cost using the following equations:

$$F_\ell = \sum_{e \in P_\ell(CB)} |I_\ell(e)| \tag{2}$$

$$V_\ell = cost_v(CB) \cdot |C_\ell(CB)| \tag{3}$$

As delta inverted index contains only the different code blocks in between ℓ-prefix scheme and $\ell+1$-prefix scheme, we do not need to calculate the filtering cost from scratch every time, converting the previous filter cost equation to the following variant.

$$F_\ell(CB) = F_{\ell-1}(CB) + \sum_{\Delta I(e) \in \Delta\Phi_\ell(CB)} |\Delta I(e)| \tag{4}$$

Prefix Scheme	Filter Cost
1-Prefix	$\|I_1(if)\|+ \|I_1(static)\|+ \|I_1(public)\|+ \|I_1(return)\|$ =3+3+4+1=11
2-Prefix	$\|I_1(if)\|+ \|I_1(static)\|+ \|I_1(public)\|+ \|I_1(return)\|+$ $\|I_1(factorial)\|+ \|\Delta I_2(if)\|+ \|\Delta I_2(static)\|+$ $\|\Delta I_2(public)\| + \|\Delta I_2(return)\|+ \|\Delta I_2(factorial)\|$ =11+2+1=14
3-Prefix	$\|I_1(if)\|+ \|I_1(static)\|+ \|I_1(public)\|+ \|I_1(return)\|+$ $\|I_1(factorial)\|+\|I_1(1)\| + \|\Delta I_2(if)\|+ \|\Delta I_2(static)\|+$ $\|\Delta I_2(public)\| + \|\Delta I_2(return)\|+ \|\Delta I_2(factorial)\|+$ $\|\Delta I_2(1)\|+\|\Delta I_3(if)\| + \|\Delta I_3(static)\|+ \|\Delta I_3(public)\|+$ $\|\Delta I_3(return)\| + \|\Delta I_3(factorial)\|+ \|\Delta I_3(1)\|$ =14+1+1+2+1=19

Table 5: Calculation of filter cost for CB_1

In Table 5 we show the calculation of filter cost for different prefix schemes for our running example, CB_1. For instance, to calculate the filter cost for prefix scheme $\ell = 2$ we can use filter cost from prefix scheme $\ell = 1$. So the filter cost for prefix scheme $\ell = 2$ (14) is the summation of filter cost of prefix scheme $\ell = 1$ (i.e. 11) and the size of the delta inverted index of the tokens: {*if*, *static*, *public*, *return*, *factorial*}.

Algorithm 1 Pseudo-code of the cost calculation for each code block.

Input: CB=code block represented as a bag-of-tokens with tokens sorted according to their global frequency, $\Delta\Phi_\ell$(CB)=set of additional delta inverted indices for ℓ-prefix scheme comparing to (ℓ-1)-prefix scheme of code block CB , $\Phi_\ell(CB)$=$\{\Delta I_i(e)|e \in P_\ell(CB),\ 1 \leq i \leq \ell\}$ where $\Delta I_i(e)$=delta inverted index of i-prefix scheme, n=user-defined maximum threshold scheme

Output: Total Cost of code block CB, prefix scheme ℓ with lowest cost for code block CB

Variables: T_ℓ=Total Cost of ℓ-prefix scheme; F_ℓ=Filter Cost of ℓ-prefix scheme; V_ℓ=Verification Cost of ℓ-prefix scheme; $C_\ell(CB)$=candidate set of ℓ-prefix scheme; $H[CB]$= Hashmap storing the number of processed lists that contain code block CB; S=union set of multiple $\Delta I_\ell(e)$; $cost_v(CB)$=average cost of verifying the candidate CB; $C^{=}_{\ell-1}(CB)$=set of code blocks that occur at least $(\ell-1)$ lists in $\Phi_{\ell-1}(CB)$; $C^{>}_{\ell-1}(CB)$=set of code blocks that appear in more than $(\ell-1)$ lists in $\Phi_{\ell-1}(CB)$;

function Cost_Calculation(CB,$\Delta\Phi_\ell$(CB))
  $H[CB] = 0$
  ℓ=1
  S=$\emptyset$
  while ($\ell <= n$) **do**
    $C^{=}_{\ell-1}(CB)$=$\emptyset$
    $C^{>}_{\ell-1}(CB)$=$\emptyset$
    $tokensToBeIndexed = |CB| - \lceil\theta|CB|\rceil + \ell$
    for each token e∈CB[1:$tokensToBeIndexed$] **do**
      F_ℓ=$F_\ell(CB) = F_{\ell-1}(CB) + \sum_{\Delta I(e)\in\Delta\Phi_\ell(CB)} |\Delta I(e)|$ ▷ calculation of filter cost
      if ($\ell == 1$) **then**
        $C_\ell(CB) = C_\ell(CB) \cup \Delta I_\ell(e)$
        for each code block $CB \in \Delta I_\ell(e)$ **do**
          $H[CB] = H[CB] + 1$
        end for
      else
        $S = S \cup \Delta I_\ell(e) \in \Delta\Phi_\ell(CB)$
        for each code block $CB \in S$ **do**
          if ($H[CB] > (\ell-1)$) **then**
            $C^{>}_{\ell-1}(CB)$= $C^{>}_{\ell-1}(CB) \cup CB$
          end if
          if ($H[CB] == (\ell-1)$) **then**
            $C^{=}_{\ell-1}(CB)$= $C^{=}_{\ell-1}(CB) \cup CB$
          end if
        end for
        for each $\Delta I_\ell(e) \in \Delta\Phi_\ell(CB)$ **do**
          for each code block $CB \in \Delta I_\ell(e)$ **do**
            $H[CB] = H[CB] + 1$
          end for
        end for
        $|C_{(\ell)}(CB)| = |C^{>}_{(\ell-1)}(CB)| + |C^{=}_{(\ell-1)}(CB) \cap \bigcup_{\Delta I(e)\in\Delta\Phi_\ell(CB)} \Delta I(e)|$
      end if
    end for
    V_ℓ=$cost_v(CB) \cdot |C_\ell(CB)|$ ▷ calculation of verification cost
    T_ℓ= F_ℓ + V_ℓ ▷ calculation of total cost
    if ($T_\ell > T_{\ell-1}$) **then return** $T_{\ell-1}$, $(\ell-1)$
    end if
    ℓ++
  end while
end function

The average cost of verifying a code block CB is $cost_v(CB)$. We compute this cost using s_u and s_l, the upper bound and lower bound of the sizes of all the code blocks within code repository R. We express this via following equation:

$$cost_v(CB) = |CB| + \frac{s_{|u|} + s_{|l|}}{2} \tag{5}$$

The average cost of verifying CB_1, $cost_v(CB_1)$=16+$\frac{16+28}{2}$=38. Here, 16 and 28 are the lower and upper bound of the size among the five code blocks in our running example.

To estimate the candidate set size of 1-prefix scheme, $C_1(CB)$, we simply calculate the number of code blocks which appear in at least one inverted list using the tokens within 1-prefix scheme. In Table 6, the first row shows that CB_1,CB_2,CB_3,CB_5 have appeared in at least one inverted index of tokens within 1-prefix scheme of CB_1, and, therefore, the candidate set size is 4. For computing the candidate set size $|C_{\ell+1}(CB)|$ of $(\ell+1)$-prefix scheme we can utilize the candidate set size of ℓ-prefix scheme $|C_\ell(CB)|$.

Those code blocks that appear in more than ℓ number of inverted lists of tokens within ℓ-prefix scheme, also appear in the candidate set of $(\ell+1)$-prefix scheme. All other code blocks which appear in at least ℓ number of inverted lists of tokens within ℓ-prefix scheme can also appear in the candidate set of $(\ell+1)$-prefix scheme if and only if these code blocks appear in the additional delta inverted lists of the $(\ell+1)$-prefix scheme. We take the summation of the size of these two sets where one set contains the intersection of the code blocks appearing both in the ℓ-number of inverted lists in ℓ-prefix scheme and the additional delta inverted lists of $(\ell+1)$ prefix scheme and the other set contains the code blocks appearing in more than ℓ number of inverted lists in ℓ-prefix scheme.

Let $C_\ell^{=}(CB)$ represents the set of code blocks that occur at least ℓ list in $\Phi_\ell(CB)$ and let $C_\ell^{>}(CB)$ represents the set of code blocks that appear in more than ℓ lists in $\Phi_\ell(CB)$. Using these, we can define the following candidate set

equation.

$$|C_{(\ell+1)}(CB)| = |C_{\ell}^{>}(CB)| + |C_{\ell}^{=}(CB) \cap \bigcup_{\Delta I(e) \in \Delta\Phi_{\ell+1}(CB)} \Delta I(e)| \qquad (6)$$

In Table 6 we show how we utilize the candidate set size $|C_1(CB_1)|$ of 1-prefix scheme to derive the candidate set size $|C_2(CB_1)|$ of 2-prefix scheme for code block CB_1. For estimating candidate set size $|C_2(CB_1)|$ first we have to calculate the number of code blocks which appear in more than one inverted lists of tokens within the 1-prefix scheme. In this case, we only need to consider those code blocks which appear in ΔI_2 which is the delta inverted list for 2-prefix scheme of CB_1. If we check all the tokens {*if, static, public, return, factorial*} within 2-prefix scheme of CB_1, we see only tokens {*return ,factorial*} are in CB_1,CB_2,CB_5 in the delta inverted list ΔI_2. Since CB_1,CB_2,CB_5 have appeared in at least 2 inverted lists of the tokens within 1-prefix scheme of CB_1, these code blocks are the elements of set $C_1^{>}(CB_1)$. Therefore code blocks CB_1,CB_2,CB_5 should be included in the candidate set $C_2(CB_1)$ of 2-prefix scheme of CB_1.

Those code blocks which appear only in one inverted list of the tokens of 1-prefix scheme, are the elements of set $C_1^{=}(CB_1)$. If those code blocks of $C_1^{=}(CB_1)$, also appear in the additional delta inverted list $\Delta\Phi_2(CB_1)$ of 2-prefix scheme then these code blocks are the elements of the candidate set $C_2(CB_1)$ of 2-prefix scheme. Therefore we have to take the intersection between two sets where one set, $C_1^{=}(CB_1)$, contains the code blocks appearing in one inverted list and the other set contains the code blocks appearing in the additional delta inverted list $\Delta\Phi_2(CB_1)$. For CB_1 there are no code blocks that appear only in one inverted list, and the intersection of these two sets is $\emptyset$.

Finally we take the summation of the size of those two sets. In Table 7, we show the calculation of verification cost and total cost for our running example. To calculate the verification cost of CB_1 we have multiplied the average verification cost by the candidate set size. As the 2-prefix scheme cost is smaller than the 3-prefix scheme cost, we stop here and take $\ell = 2$ as the preferred prefix scheme.

Prefix Scheme	Candidate Set Size
1-Prefix	$\{\|\Delta I_1(if)\|, \|\Delta I_1(static)\|, \|\Delta I_1(public)\|, \|\Delta I_1(return)\|\}=$ $\{\|(CB_1, CB_3, CB_5)\|,\|(CB_1, CB_2, CB_3)\|$ $\|(CB_1, CB_2, CB_3, CB_5)\|,\|(CB_1)\|\}=$ $\{\|(CB_1, CB_2, CB_3, CB_5)\|\ \}=4$
2-Prefix	$\|C_2(CB_1)\| = \|C_1^{>}(CB_1)\| + \|C_1^{=}(CB_1) \cap \bigcup_{\Delta I(e) \in \Delta\Phi_2(CB_1)} \Delta I(e)\|$ $= \|C_1^{>}(CB_1)\| + \|C_1^{=}(CB_1) \cap \{\Delta I_2(if) \cup \Delta I_2(static) \cup \Delta I_2(public) \cup$ $\Delta I_2(return) \cup \Delta I_2(factorial) \cup \Delta I_1(factorial)\}\|$ $=\|C_1^{>}(CB_1)\| + \|C_1^{=}(CB_1) \cap \{CB_2 \cup CB_5 \cup CB_1\}\|$ $=\|C_1^{>}(CB_1)\| + \|\emptyset \cap \{CB_2 \cup CB_5 \cup CB_1\}\|$ $=\|\{CB_1, CB_2, CB_5\}\| + \|\emptyset \cap \{CB_2 \cup CB_5 \cup CB_1\}\|$ $=3 + 0 = 3$
3-Prefix	$\|C_3(CB_1)\| = \|C_2^{>}(CB_1)\| + \|C_2^{=}(CB_1) \cap \bigcup_{\Delta I(e) \in \Delta\Phi_3(CB_1)} \Delta I(e)\|$ $= \|C_2^{>}(CB_1)\| + \|C_2^{=}(CB_1) \cap \{\Delta I_3(if) \cup \Delta I_3(static) \cup \Delta I_3(public) \cup$ $\Delta I_3(return) \cup \Delta I_3(factorial) \cup \Delta I_3(1) \cup \Delta I_2(1) \cup \Delta I_1(1)\}\|$ $=\|C_2^{>}(CB_1)\| + \|C_2^{=}(CB_1) \cap \{CB_4, CB_3, CB_2, CB_5, CB_1\}\|$ $=\|C_2^{>}(CB_1)\| + \|\emptyset \cap \{CB_4, CB_3, CB_2, CB_5, CB_1\}\|$ $=\|\{CB_1, CB_2, CB_3, CB_5\}\| + \|\emptyset \cap \{CB_4, CB_3, CB_2, CB_5, CB_1\}\|$ $=4 + 0 = 4$

Table 6: Calculation of candidate set size for CB_1

Prefix Scheme	Filter Cost	Verification Cost	Total Cost
1-Prefix	11	$cost_v(CB) \cdot \|C_1(CB)\|$ =38*4.0=152.0	163.0
2-Prefix	14	$cost_v(CB) \cdot \|C_2(CB)\|$ =38*3.0=114.0	128.0
3-Prefix	19	$cost_v(CB) \cdot \|C_3(CB)\|$ =38*4.0=152.0	171.0

Table 7: Calculation of total cost for CB_1.

4.3. Code Clone Search

The adaptive prefix filtering technique can also be utilized for code clone search, where a user-specified code block (i.e. the query) is matched in a corpus consisting of numerous code blocks. In code clone search, different from code clone detection, we do not pre-specify the similarity threshold value before the index is built from the corpus, as we would like for the same index to be able to serve different queries with different threshold values. Therefore, the index structure should be able to deal with any threshold value, $1 <= |s| <= 10$, where $|s|$ is the maximum threshold value the index could serve.

As a naive approach, we can build an index for all possible threshold value

from 1 to $|s|$ for each code block. However, this would take up huge space and be very time consuming.

Instead of building delta inverted indices for each threshold value from 1 to l, we can build delta inverted indices for the maximum threshold value $|s|$, i.e. for 100% similarity (threshold value 10). With this maximum threshold value we build delta inverted indices, ΔI_1, ΔI_2, ΔI_3 and so on, until we reach the maximum prefix scheme. For example for code block $CB1$ from Table 2, ΔI_1 contains the token *if*, ΔI_2 contains the token *static*, ΔI_3 contains the token *public* and so on. We continue to populate the delta inverted indices until ΔI_8, which contains the final token for this specific code block, *result*. We continue this process for all of the code blocks in the corpus. At retrieval time, we use this data structure as the means to answer code clone search queries with retrieval-time similarity thresholds. Apart from this modification to the data structure, the algorithm follows the same logic as in code clone detection.

5. Experimental Results

Our goal is to implement a code clone detection tool that can scale to mas-
450 sive inter or intra project source code repositories and overcome limitations in many existing tools, such as, unsustainable execution time, inadequate system memory, restrictions in inner data structures, and exhibiting errors due to their design not expecting a large input [16, 17, 37]. In evaluating our tool, we focused on answering the following set of research questions.

- **RQ 1:** *Does adaptive prefix filtering achieve better performance at scale than the best state of the art tool SourcererCC?*

  SourcererCC is a recent code clone detection tool aimed at scalability on a single machine [16]. The adaptive prefix filtering heuristic presented in this paper extends the filtering heuristics used by SourcererCC, so a comparisson between the two is both natural and necessary.

  Several code clone detection tools have been benchmarked in recent papers [38, 39]. Among all of the measured tools, four publicly available

tools achieve exceptional scalability and accuracy in code clone detection: CCFinderX [3], Deckard [23], iClones [29] and NiCad [40]. In turn, SourcererCC has been measured to outperform these four publicly available and popular tools [16].

Recently, several popular code clone detection tools were compared [30], including VUDDY [30], SourcererCC [16], CCFinderX [3] and Deckard [23]. SourcererCC outperformed CCFinderX [3] and Deckard [23]. VUDDY outperformed SourcererCC, however, it only detects Type-1 and Type-2 clones.

- **RQ 2:** *Does adaptive prefix filtering achieve reasonable accuracy in clone detection?*

  This research question aims to determine whether the adaptive prefix filtering heuristic can achieve reasonable precision and recall, and whether it can serve as a replacement to SorcererCC.

- **RQ 3:** *Does the application of adaptive prefix filtering to code clone search achieve acceptable query response time?*

  Searching for similar code fragments in very large scale source code repository within reasonable amount of time is a challenging problem. In using our technique for code clone search, we want to determine whether this application produces reasonable response time, which is key for it to be useful in practice.

5.1. Performance of Adaptive Prefix Filtering (RQ1)

In this section we discribe a comparison of the scalability of adaptive prefix filtering technique relative to SourcererCC [16]. To answer this research question we rely on publicly available large-scale evaluation datasets, which have recently become available. The IJaDataset 2.0 [27] is a large inter-project Java repository containing 25,000 open source projects with 3 million source files and 250MLOC. It is mined from SourceForge and Google Code [16]. To compare SourcererCC

with adaptive prefix filtering, both tools were benchmarked on a standard workstation with 3.50GHz quad-core i5 CPU, 32.0 GB of RAM memory, and 64-bit windows operating system. The execution of both tools was scripted to measure the run-time of both tools 5 times for each input and report the average. For adaptive prefix filtering, we used the 2-prefix scheme in constructing the delta inverted index. In order to measure the performance at different input sizes, we blindly selected a subset of the Java source files in IJADataset [27] ranging from 10,000 to 160,000 files, approximately from 1MLOC to 17MLOC, such that the selected files at smaller input sizes were also contained in the larger inputs. This ensures a stable measurement since code clone technique execution time may be reliant on clone density [16]. Both techniques consumed extreme amounts of time (>10 hours) at 160,000 files(approximately 17MLOC), and therefore we did not attempt using larger input sizes. And for same reason we limited similarity degree from 70% to 90% in comparison. But for smaller dataset(1MLOC) we have experimented on all similarity degrees(from 10% to 100%).

Figure 2 shows a comparison in execution time for adaptive prefix filtering technique and SourcererCC for threshold values 7, 8, and 9 (i.e. 70%-90% similarity). For threshold values 7 and 8 adaptive prefix filtering is showing an improvement relative to SourcererCC, but not at threshold value 9. As the the input size increases toward 160,000 files, the difference in execution time between SourcererCC and adaptive prefix filtering becomes wider for threshold values 7 and 8.

In Table 8 we show the execution times for SourcererCC and adaptive prefix filtering technique for a fixed input of 10,000 files(approximately 1MLOC), but at varying threshold values. We show both raw execution times as well as the percentage improvement of adaptive prefix filtering technique in the rightmost column. Adaptive prefix filtering performs better than SourcererCC for threshold values ranging from 4 to 8. For threshold value 1, 2, 3, 9 and 10 the execution time of adaptive prefix filtering technique is larger. We argue that the similarity degrees from 40% to 80% are more commonly used. Exact or near-exact similarity (i.e. 90% and 10%) do not make sense for detecting Type-2 and

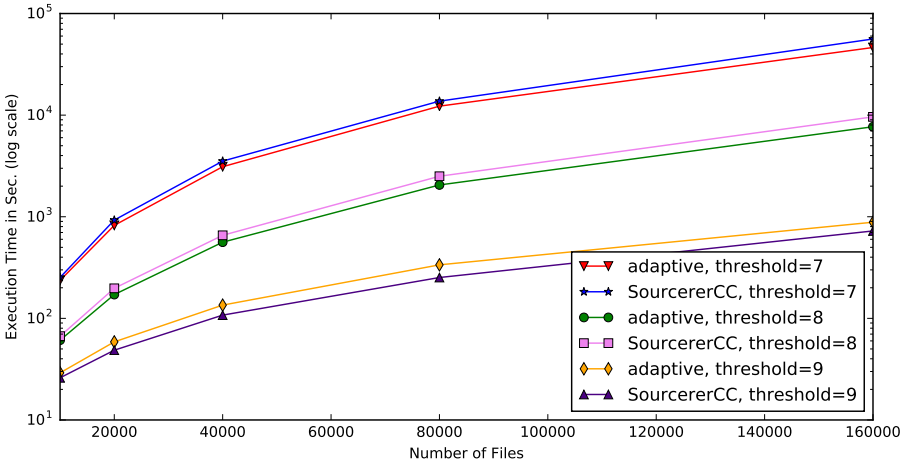

Figure 2: *Comparison of execution time for different input sizes between adaptive prefix filtering and SourcererCC.*

Threshold Value	Adaptive Prefix Filtering	SourcererCC	% Improvement
1	4982.23	4452.31	-10.64%
2	5357.54	5339.58	-0.34%
3	4499.14	4478.93	-0.45%
4	3001.19	3015.76	0.49%
5	1627.95	1652.18	1.49%
6	712.45	740.16	3.89%
7	222.68	249.27	11.94%
8	58.84	64.25	9.19%
9	27.25	23.99	-11.96%
10	23.25	21.89	-5.85%

Table 8: Comparison of execution time (in seconds) between adaptive prefix filtering and SourcererCC (10,000 files).

Type-3 code clones, while code clones detected with 10% to 30% similarity are likely to contain a very large number of false positives.

To better understand the effect of the more aggressive filtering performed by adaptive prefix filtering, and understand the rationale behind the performance numbers in Table 8, we compared the number of candidate pairs (in log scale) of SourcererCC and adaptive prefix filtering technique in Figure 3 for 10000 files. The candidate pairs are the number of remaining code clone candidates after the filtering that both techniques perform. A reduction in the number of candidates translates to improvement in execution time, after the penalty for the more sophisticated indexing structure and cost calculation of adaptive prefix filtering is factored in. We observe the strongest reduction in the percentage of candidate pairs in the similar thresholds of 4 to 9 where we observed improvements in execution time. Threshold value 9 has a large percentage reduction in candidate numbers, but the actual reduction numbers are very low[1] and insufficient for offsetting the penalty of more aggressive filtering. For threshold value 10 (100% similarity) the percentage decrease is 0. This is logical because with complete similarity adaptive prefix filtering reduces to regular prefix filtering as there is no room to extract a deeper prefix.

Memory Usage. Fitting with the limited memory budget available on commodity machines is an important factor in scalable code clone detection. For our adaptive prefix filtering technique we use a deeper prefix scheme, resulting in a larger index that will clearly consume more memory than SourcererCC. We measured the memory requirement for both SourcererCC and adaptive prefix filtering for 1 MLOC in the indexing and clone detection phases. While varying the threshold value, we measured how much memory is consumed in each iteration of the indexing process and recorded the maximal amount of memory consumed. The resulting memory requirement was 103 MB for threshold value 1 and 60 MB for threshold value 10, which is 60% and 20% higher relative to SourcererCC's maximal memory footprint in indexing. The memory

[1]Note that it is a log scale graph.

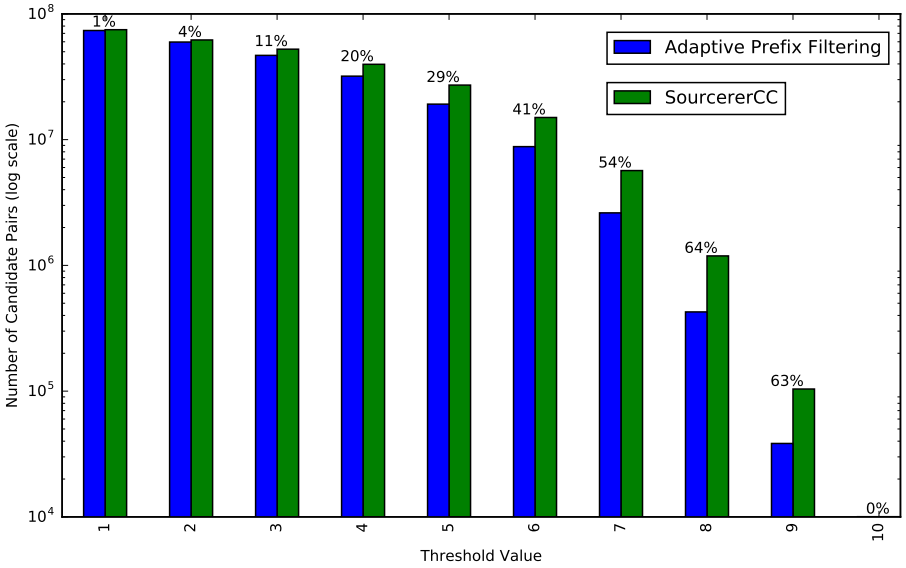

Figure 3: *Comparison of number of candidate pairs between adaptive prefix filtering and SourcererCC (10,000 files).*

usage is smaller for the larger threshold value because as the threshold value is increasing the prefix length is decreasing, so fewer tokens need to be stored in the index structure. In the subsequent clone detection phase we measured the memory requirement for each query block, again, selecting the maximum amount of memory required. We obtained 316 MB for adaptive prefix filtering and 204 MB for SourcererCC. In conclusion, although adaptive prefix filtering technique requires higher amount of memory compared to SourcererCC, the memory requirement of both techniques is relatively small and would not impact most deployments. Part of the reason for the small footprint is the efficient allocation of memory by Apache Lucene, which was used as to store the inverted index by both SourcererCC and our implementation.

5.2. Accuracy of Adaptive Prefix Filtering (RQ2)

We use precision and recall for measuring the accuracy of adaptive prefix filtering technique as these are the two most commonly used metrics used to

determine the quality of a code clone detection technique [41]. Measurement of clone recall and precision has been greatly aided by recent datasets and frameworks like BigCloneEval [42]. This framework can be used for the evaluation of code clone detection tools based on the BigCloneBench clone detection benchmark [43]. BigCloneBench contains a large set of known clones from the inter-project software repository IJaDataset 2.0 [21, 16], which we used in RQ1. Note that SourcererCC and adaptive prefix filtering produce the same clones as output due to the inherent similarity in the techniques.

Recall. For measuring recall of adaptive prefix filtering we use file level granularity, which means the clone pairs are actually pair of two Java source files that are detected as code clones to each other. The evaluation of SourcererCC was at the method level [16]. In measuring recall using BigCloneBench, Type-3 and Type-4 code clones are separated in four categories because it is difficult to separate Type-3 and Type-4 since there is no consent on the smallest similarity of Type-3 [21]. These four categories are: Very Strongly Type-3 (VST3) clones that has range of syntactical similarity from 90% to 100%, Strongly Type-3 (ST3) that has range of syntactical similarity from 70% to 90%, Moderately Type-3 (MT3) that has range of syntactical similarity from 50% to 70% and Weakly Type-3 (WT3/T4) that has range of syntactical similarity from 0% to 50%, which are often Type-4 code clones [16].

For this evaluation we used a similarity threshold of 70% in executing adaptive prefix filtering, which is the default setting for BigCloneEval. Our technique produced very high recall for Type-1 code clones (97%), and detected Type-2 code clones with a reasonable recall of 77%. For Type-3 clones, the recall decreases significantly, from 60% for the VST3 category, 26% for ST3, 16% for MT3 and less than 1% for WT3/T4. The Weakly Type-3/Type-4 clones have low syntactic similarity which makes it very hard for our technique to detect at the 70% threshold, so this result is not unexpected.

Precision. The adaptive prefix filtering technique detects the same code clones as SourcererCC, and therefore it's precision (and recall) will be same as SourcererCC. SourcererCC's precision was previously evaluated via a set of 390 clone

pairs, which were manually identified by several researchers with high mutual inter-agreement. Out of these 390 clone pairs, 355 were true positives while 35 were found false positives, resulting in a precision of 91% computed at method level granularity [16].

The value of the precision metric, unlike recall, is influenced by the number of false positives. For token based code clone detection techniques, a common source of false positive clones stems from the fact that these techniques commonly treat the input as a bag of words, ignoring the ordering of tokens in the input. To illustrate this point we show a false positive clone pair identified by our technique (and likely most other token based techniques), snippets of which are shown in Table 9. The pair of code blocks constituting the false positive example are: 1) a code block that implements password validation and 2) a code block that implements password encryption. Although these two code fragments have 70% similar tokens they are functionally dissimilar. In fact, the pair of code blocks have significant differences at the line level and their different purpose would be easily observed by a human. However, both of the code fragments are dealing with passwords there are significant number of similar tokens. Some of these are: *password*, *passwordInDb*, *MessageDigest*, *update*.

```
//Password Validation
byte[] digestInDb =
  new byte[pwdInDb.length - SALT_LENGTH];
System.arraycopy(pwdInDb, SALT_LENGTH,
     digestInDb, 0, digestInDb.length);
if (Arrays.equals(digest, digestInDb)) {
  return true;
} else {
  return false;
}
```

```
//Password Encryption
pwd = new byte[digest.length+SALT_LENGTH];
System.arraycopy(salt,0,pwd,0,SALT_LENGTH);
System.arraycopy(digest,0,pwd,SALT_LENGTH,
     digest.length);
return byteToHexString(pwd);
```

Table 9: Snippets of false positive clone pair identified by our technique.

5.3. Applicability Towards Code Clone Search (RQ3)

We showed that the adaptive prefix filtering technique can easily be extended so that it can support similarity search. In this section, we examine the practicality of code clone search based on this technique by measuring the performance of this variant. Our goal here is to show that the use of our technique for code clone search is practical, but not to show that our technique outperforms those that specialize solely on the code clone search problem. Table 10 shows the time it takes to construct the index, and, more importantly, the average query time for 1000 files randomly selected from IJADataset with similarity degree 80% (i.e. threshold value 8). By index time, we refer to the time required to build the special indexing structure given a maximum similarity threshold of 100%. After creating this index, we use a random selection of 1000 Java source files, from the files used to construct the index, performing a code clone search for each file. We report the average search time for different input sizes.

In examining the performance of our technique in Table 10, we observe subsecond response times for each queries, even at the larger corpus sizes. This is likely to be reasonable performance for many applications.

Number of Files	Similarity	Index Time (in sec.)	Query Time (in sec.)
10000	80%	70.52	0.046
20000	80%	160.12	0.095
40000	80%	351.29	0.177
80000	80%	736.93	0.368

Table 10: Performance of code clone search using adaptive prefix filtering.

6. Threats to Validity

- *Database Selection:* Our experimental evaluation is limited only to the Java programming language via the IJaDataset. A threat to validity is that our approach may not be be effective for other programming languages or other corpora. As a mitigating factor to this threat, we argue

that researchers have observed that programming languages possess common statistical traits in their term distribution relative to natural language [44]. Therefore, our technique is likely to exploit these similarity notions and be beneficial across different languages and code bases.

- *Hardware and Input Size Limitation*: We evaluated adaptive prefix filtering for code clone detection on a commodity workstation, and not on specialized hardware. Our implementation is also multithreaded, which improves performances on modern CPUs. It is unclear whether the performance improvements we observed translate to other hardware architectures.

  We also used a fixed time budget of roughly one day in executing our tool, which limited our largest input to 160,000 Java source file containing approximately 17MLOC. Another threat to validity is that we did not experiment with larger inputs, for which it is possible that our technique fails to perform well. As a mitigating factor, we note that our results span numerous (smaller) input sizes and the trend we observed is towards improved rather than degrading performance compared to SourcererCC as the input size increases.

7. Conclusions and Future Work

In this paper we described a novel code clone detection technique utilizing the adaptive prefix filtering heuristic [24]. Our proposed technique can outperform the most recent scalable code clone detection tool SourcererCC in terms of execution time within a certain range of similarity thresholds (between 40% to 80%). Since the technique directly extends SourcererCC, it produces the same output, duplicating the same high precision and recall as SourcererCC. We have evaluated our proposed code clone detection tool by randomly creating a subset of the IJaDataset 2.0 [27], a large code clone benchmark that contains 250MLOC and 25,000 open source Java systems.

Our experimental results indicate that our adaptive prefix filtering based code clone detection technique can be practically utilized in various code clone detection related applications that require large source code repository to be processed on a single machine. In our experiments we successfully performed code clone detection on a 17MLOC Java code base within a reasonable time window of several hours. Our paper also attempts to provide numerous examples and practical implementation advice for future applications of adaptive prefix filtering.

To the best of our knowledge our approach is also among few code clone detection techniques in the literature that can be additionally used for code clone search, which is the related problem of retrieving similar code blocks to a single code block issued as a query. Typically, code clone detection techniques make design decisions that allow them to only operate in batch mode, while code clone search requires the flexibility to answer numerous clone queries using a pre-built index. We show acceptable indexing and querying times for this application of our technique.

As future work, we intend to better evaluate adaptive prefix filtering across a variety of languages and applications. We also aim to attempt to parallelize the technique to distributed memory architectures, which would greatly improve its scalability and extend its applicability .

8. Acknowledgment

We acknowledge the support of the DARPA MUSE program under Air Force Research Lab contract no. FA8750-16-2-0288.

References

[1] C. K. Roy, J. R. Cordy, A survey on software clone detection research, Queens School of Computing TR 541 (115) (2007) 64–68.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Software Maintenance, 1998. Proceedings., International Conference on, IEEE, 1998, pp. 368–377.

[3] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670.

[4] D. Chatterji, J. C. Carver, N. A. Kraft, J. Harder, Effects of cloned code on software maintainability: A replicated developer study, in: Reverse Engineering (WCRE), 2013 20th Working Conference on, IEEE, 2013, pp. 112–121.

[5] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, N. A. Kraft, Measuring the efficacy of code clone information in a bug localization task: An empirical study, in: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on, IEEE, 2011, pp. 20–29.

[6] C. K. Roy, J. R. Cordy, An empirical study of function clones in open source software, in: Reverse Engineering, 2008. WCRE'08. 15th Working Conference on, IEEE, 2008, pp. 81–90.

[7] B. S. Baker, On finding duplication and near-duplication in large software systems, in: Reverse Engineering, 1995., Proceedings of 2nd Working Conference on, IEEE, 1995, pp. 86–95.

[8] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto, Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects, in: Reverse Engineering (WCRE), 2012 19th Working Conference on, IEEE, 2012, pp. 387–391.

[9] R. Koschke, Large-scale inter-system clone detection using suffix trees, in: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, 2012, pp. 309–318.

[10] D. M. German, M. Di Penta, Y.-G. Gueheneuc, G. Antoniol, Code siblings: Technical and legal implications of copying code between applications, in: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on, IEEE, 2009, pp. 81–90.

[11] A. Hemel, R. Koschke, Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices, in: Reverse Engineering (WCRE), 2012 19th Working Conference on, IEEE, 2012, pp. 357–366.

[12] J. Davies, D. M. German, M. W. Godfrey, A. Hindle, Software bertillonage: finding the provenance of an entity, in: Proceedings of the 8th working conference on mining software repositories, ACM, 2011, pp. 183–192.

[13] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, H. Iida, Shinobi: A real-time code clone detection tool for software maintenance, Nara Institute of Science and Technology (2008) 26.

[14] I. Keivanloo, J. Rilling, P. Charland, Internet-scale real-time code clone search via multi-level indexing, in: Reverse Engineering (WCRE), 2011 18th Working Conference on, IEEE, 2011, pp. 23–27.

[15] K. Chen, P. Liu, Y. Zhang, Achieving accuracy and scalability simultaneously in detecting application clones on android markets, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 175–186.

[16] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, SourcererCC: Scaling code clone detection to big-code, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 1157–1168.

[17] J. Svajlenko, I. Keivanloo, C. K. Roy, Scaling classical clone detection tools for ultra-large datasets: An exploratory study, in: Proceedings of the 7th International Workshop on Software Clones, IEEE Press, 2013, pp. 16–22.

[18] C. K. Roy, M. F. Zibran, R. Koschke, The vision of software clone management: Past, present, and future (keynote paper), in: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on, IEEE, 2014, pp. 18–33.

[19] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based code clone detection: incremental, distributed, scalable, in: Software Maintenance (ICSM), 2010 IEEE International Conference on, IEEE, 2010, pp. 1–9.

[20] C. K. Roy, J. R. Cordy, Near-miss function clones in open source software: an empirical study, Journal of Software: Evolution and Process 22 (3) (2010) 165–189.

[21] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, M. M. Mia, Towards a big data curated benchmark of inter-project code clones, in: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, 2014, pp. 476–480.

[22] S. Livieri, Y. Higo, M. Matushita, 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, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 106–115.

[23] L. Jiang, G. Misherghi, Z. Su, S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 96–105.

[24] J. Wang, G. Li, J. Feng, Can we beat the prefix filtering?: an adaptive framework for similarity join and search, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, 2012, pp. 85–96.

[25] S. Sarawagi, A. Kirpal, Efficient set joins on similarity predicates, in: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM, 2004, pp. 743–754.

[26] R. Vernica, M. J. Carey, C. Li, Efficient parallel set-similarity joins using mapreduce, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 495–506.

[27] Ambient Software Evoluton Group. IJaDataset 2.0., `http://secold.org/projects/seclone`, june 2017.

[28] J. R. Cordy, C. K. Roy, The nicad clone detector, in: Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, IEEE, 2011, pp. 219–220.

[29] N. Göde, R. Koschke, Incremental clone detection, in: Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on, IEEE, 2009, pp. 219–228.

[30] S. Kim, S. Woo, H. Lee, H. Oh, Vuddy: A scalable approach for vulnerable code clone discovery, in: Security and Privacy (SP), 2017 IEEE Symposium on. IEEE, 2017.

[31] M.-W. Lee, J.-W. Roh, S.-w. Hwang, S. Kim, Instant code clone search, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, ACM, New York, NY, USA, 2010, pp. 167–176.

[32] I. Keivanloo, J. Rilling, P. Charland, Seclone-a hybrid approach to internet-scale real-time code clone search, in: Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, IEEE, 2011, pp. 223–224.

[33] I. Keivanloo, J. Rilling, Y. Zou, Spotting working code examples, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 664–675.

[34] B. S. Baker, A theory of parameterized pattern matching: algorithms and applications, in: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, ACM, 1993, pp. 71–80.

[35] R. Koschke, Large-scale inter-system clone detection using suffix trees and hashing, Journal of Software: Evolution and Process 26 (8) (2014) 747–769.

[36] Apache Lucene, `https://lucene.apache.org/core/`, june 2017.

[37] J. Svajlenko, I. Keivanloo, C. K. Roy, Big data clone detection using classical detectors: an exploratory study, Journal of Software: Evolution and Process 27 (6) (2015) 430–464.

[38] J. Svajlenko, C. K. Roy, Evaluating clone detection tools with bigclonebench, in: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, 2015, pp. 131–140.

[39] J. Svajlenko, C. K. Roy, Evaluating modern clone detection tools, in: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, 2014, pp. 321–330.

[40] C. K. Roy, J. R. Cordy, Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, IEEE, 2008, pp. 172–181.

[41] A. Sheneamer, J. Kalita, A survey of software clone detection techniques, International Journal of Computer Applications (2016) 0975–8887.

[42] BigCloneEval, `https://github.com/jeffsvajlenko/BigCloneEval`, june 2017.

[43] BigCloneBench, `https://github.com/clonebench/BigCloneBench`, june 2017.

[44] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: Proceedings of the 34th International Conference on Software

Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 837–847.
URL http://dl.acm.org/citation.cfm?id=2337223.2337322