Code Clone Configuration as a Multi-Objective Search Problem

Denis Sousa
State University of Ceará
Fortaleza, Brazil
denis.sousa@aluno.uece.br

Matheus Paixao
State University of Ceará
Fortaleza, Brazil
matheus.paixao@uece.br

Chaiyong Ragkhitwetsagul
Faculty of Information and Communication Technology, Mahidol University
Nakhon Pathom, Thailand
chaiyong.rag@mahidol.edu

Italo Uchoa
State University of Ceará
Fortaleza, Brazil
italo.uchoa@aluno.uece.br

ABSTRACT

Clone detection is an automated process for finding duplicated code within a project's code base or between online sources. Nowadays, the code cloning community advocates that developers must be aware of the clones they may have in their code bases. In modern clone detection, rank-based tools appear as the ones able to handle the large code corpora that are necessary to identify online clones. However, such tools are sensitive to their parameters, which directly affects their clone detection abilities. Moreover, existing parameter optimization approaches for clone detectors are not meant for rank-based tools. To overcome this issue and facilitate empirical studies of code clones, we introduce Multi-objective Code Clone Configuration, a new approach based on multi-objective optimization to search for an optimal set of parameters for a rank-based clone detection tool. In our empirical evaluation, we ran 3 baseline search algorithms and NSGA-II to assess their performance in this new optimization problem. Additionally, we compared the optimized configurations with the default one. Our results show that NSGA-II was the algorithm that achieved the best performance, finding better configurations than those of the baseline algorithms. Finally, the optimized configurations achieved improvements of 71.08% and 46.29% for our fitness functions.

KEYWORDS

Clone Detection, Search-based Software Engineering, Multi-objective Optimization

ACM Reference Format:
Denis Sousa, Matheus Paixao, Chaiyong Ragkhitwetsagul, and Italo Uchoa. 2024. Code Clone Configuration as a Multi-Objective Search Problem. In *Proceedings of the 18th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), October 24–25, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3674805.3690757

1 INTRODUCTION

It is common for developers to adopt code cloning rather than other code reuse strategies, to increase productivity during software development [19, 35, 37, 38]. By leveraging existing code snippets, developers insert code clones into their systems, thereby saving time that would otherwise be spent writing code from scratch. The benefits and drawbacks of code clones are still debatable in the software engineering community [8, 16]. Nonetheless, the community agrees that it is important to be aware of the clones in one's software projects to properly manage them (e.g., leaving as-is, refactoring, tracking, and removing) [19, 35, 37, 38].

In the past, code clones were mainly created by duplicating code from other parts within the same project or from other projects [19, 35, 37, 38]. Nonetheless, the way developers reuse code evolved after the rise of the internet. We currently have online code clones, which are code fragments copied from online platforms (e.g., Stack Overflow, GitHub, ChatGPT) to software projects and vice-versa [13, 22, 23, 29, 32, 48, 49]. Both types of clones are used for the same purpose: to obtain new functionality, perform a specific task, or fix a bug. The main difference between traditional clones and online clones is that the latter are challenging to locate and rectify due to the scale of the large code bases on the internet [30, 40].

Developers must exercise caution when reusing of online code. They can be dangerous to systems, leading to security breaches, API misuse, license violations, and other issues [2, 30, 53]. The Stack Overflow platform, one of the main sources of data for code reuse, allows users to find out-of-date answers accepted by the community [51]. Ragkhitwetsagul et al. [33] present a use case where a Stack Overflow code snippet remained marked as the correct answer for approximately two years despite being poorly optimized. The same code snippet was later found in 435 Java projects on GitHub. This highlights the importance of developers being aware of the implications of copying online code.

Previous studies have demonstrated that the performance of clone detection tools is heavily influenced by their parameter configurations [20, 31, 34]. Ragkhitwetsagul et al. [31] show that using the tools' default configuration may lead to sub-optimal performance. Thus, it is crucial for clone researchers and practitioners who rely on clone detection tools to regularly adjust their tools' parameter configurations in their empirical studies or usages. Unfortunately, finding an optimal configuration for code clone detectors is not simple. It depends on the dataset and the goal of finding the clones (e.g., finding literal copies, looking for alternative implementations). Thus, having an automated way to tune the code clone detector's configuration can save developers' and researchers' time.

Using search algorithms is a means to obtain the optimized configurations for clone detectors. In the work by Wang et al. [44], the authors demonstrate a single-objective search approach employing genetic algorithms to discover the best configurations for six clone detectors [1, 7, 11, 14, 15, 28]. The authors observed that adjusting configurations can significantly enhance the quality of detection. Similarly, Ragkhitwetsagul et al. [34] employs the same single-objective genetic algorithm to optimize the parameters of four clone detection tools. Nonetheless, the result from the study shows that the single objective optimization may lead to producing additional false positives and false negatives. In this context, it is important to reduce the number of false positives from clone detectors to avoid "static analysis fatigue" [36]. At the same time, if the optimized configurations only aim to provide high precision, it may also report only a few clones (i.e., low recall). Thus, it is necessary to strike a balance between the precision of the clone detector and the number of clones reported.

In this paper, we propose a multi-objective approach to the problem of configuring parameters for clone detectors, aiming to balance the precision and recall of the clone recommendations. The clone detector selected for this study is Siamese [30], as it is one of the few tools that is applicable for online code clone scenarios, and also possesses an extensive set of parameters. The Mean Overall Precision (MOP) and Mean Reciprocal Rank (MRR) metrics were adopted as fitness functions for optimization. MOP is a novel metric we created to evaluate precision by considering the ranking of clones, while MRR is a metric already used for clone detectors capable of making multiple recommendations [6, 12, 30]. Our study provides a comparison between baseline algorithms for parameter optimization and NSGA-II [9].

Our results show that NSGA-II achieves the best results. In addition, the configurations identified by NSGA-II outperform the default configuration by 71.08% and 46.29% for MOP and MRR, respectively. We make our replication package publicly available [41].

2 BACKGROUND AND RELATED WORK

Clone detection is the automated process of finding duplicate code in a given codebase or between two codebases. This practice is useful in various areas of software development, such as code refactoring [5], bug detection [21], code quality [26], and others. It is difficult for a clone detector to simultaneously achieve high precision, recall, and scalability when searching for code clones [29–31]. Thus, clone detectors are usually customizable, having multiple parameters for developers to tweak.

Code-to-code search tools are a type of code clone detection tool that, given a code query, reports a ranked list of clone candidates [17, 18, 24, 27, 30]. These tools are scalable and can search for clones in large-scale source code data. These rank-based clone detectors are useful for locating clones or recommending alternative implementations from online code bases such as Stack Overflow or GitHub. Code clone search usually returns a large number of results because of the large-scale search corpus. By having the ranked list of results, the true clones not only need to be included in the result but also be closest to the top-ranked result because developers may be able to only look at the top N results and ignore the rest.

As previously mentioned the work by Wang et al. [44] presents EvaClone, a framework capable of optimizing configurations of clone detection tools through a single-objective genetic algorithm. Ragkhitwetsagul et al. [34] present a replication of this study. However, in the replication, EvaClone provided undesirable results. A qualitative analysis showed that the tools optimized with EvaClone reported many false positives and false negatives.

Previous research efforts in clone parameter optimization are limited to optimizing agreement between tools. Considering a properly formatted oracle, such as Bellon et al. [3]'s beanchmark, one may find good results. However, when applied to a real-world system, such an approach yielded poor results for clone quality. Given the additional challenges of online clones, existing approaches for optimizing parameters are unfeasible for modern clone detectors.

3 MULTI-OBJECTIVE CODE CLONE CONFIGURATION

3.1 Fitness Functions

Given a certain code snippet as a query, rank-based clone detection tools output a list of snippets the tool believes to be clones. The tool may also choose not to suggest any snippets in case it believes there are no clones for the query. In this case, the tool's output consists of an empty list. By leveraging an **oracle** of labeled clones, one can assess the quality of a certain configuration.

Consider Q to be the set of all code snippets (queries) in the oracle. Each code snippet in the oracle is represented by $q \in Q$. In this context, $O(q)$ indicates the known clones for query q in the oracle. Consider S to be a rank-based clone detector. Given a certain query q submitted to S, $S(q)$ represents the list of snippets suggested by the tool, where $S(q)_n$ indicates the code snippet ranked at position n in the list. The goal of is that for every $q \in Q$, $S(q) = O(q)$. In this condition, not only all the snippets suggested by the tool were correct (precision) but also the tool retrieved all clones according to the oracle (recall). In rank-based clone detection, precision and recall cannot be computed according to standard metrics [31, 39, 40, 52]. Hence, to compute the precision and recall for rank-based clone detectors, we used proxy metrics inspired by classic quality indicators in recommendation systems [25].

To compute the precision of $S(q)$, we leverage *Precision@k*. Considering the first k suggestions for a query ($k \leq |S(q)|$), this metric counts the number of correct suggestions until the k_{th} rank averaged by k, as shown in Equation 1.

$$Precision@k(S(q)) = \frac{\sum_{n=1}^{k} x, \begin{cases} x = 1, & \text{if } S(q)_n \in O(q) \\ x = 0, & \text{if } S(q)_n \notin O(q) \end{cases}}{k} \quad (1)$$

Since the goal of a clone detector for a certain query is to identify all existing clones as precisely as possible, the tool's precision cannot be assessed by considering only a single value of k. Instead, given a list of suggestions, it is necessary to evaluate the *Precision@k* for the entire list. Hence, we introduce the *OverallPrecision* (OP) metric, depicted in Equation 2. Given $S(q)$ for a certain query q, the $OP(S(q))$ metric sums the *Precision@k* values where $1 \leq k \leq |S(q)|$, and averages over the number of suggestions in the list.

$$OP(S(q)) = \frac{\sum_{k=1}^{|S(q)|} Precision@k(S(q))}{|S(q)|} \quad (2)$$

The $OP(S(q))$ metric can be used to assess the overall precision of a single query. However, the oracle is composed of several queries. Consider Q_S to be the subset of queries for which clone detector S provided a non-empty list of suggestions so that $Q_S \in Q$. To assess the precision of a clone detector for a certain oracle, we employ the *MeanOverallPrecision* (MOP) metric, shown in Equation 3.

$$MOP(Q) = \frac{\sum_{q=1}^{|Q_S|} OP(S(q))}{|Q_S|} \quad (3)$$

Measuring recall in rank-based clone detection is also not straightforward. Not only it is necessary to assess to what extent the clones in the oracle were retrieved but also if the clones were retrieved within the top ranks in the list. For this, to the best of our knowledge, one needs to employ the *ReciprocalRank* (RR) indicator [43].

Consider $rank(S(q))$ to represent the rank position of the first correct clone suggested by tool S for query q. RR is computed according to Equation 4. As one can see, when a correct clone is suggested in the top ranks of the list, a higher value of RR is awarded. Differently, if a correct clone only appears at the end of the list, the metric yields a lower value. The worse value of $RR(S(q)) = 0$ occurs on two occasions: i) the tool does not retrieve a single correct clone from the oracle; and ii) the tool does not suggest any clones.

$$\begin{aligned} RR(q) &= \frac{1}{rank(S(q))}, && \text{if } |S(q)| > 0 \\ RR(q) &= 0, && \text{if } |S(q)| = \emptyset \end{aligned} \quad (4)$$

The RR metric can be used to assess the recommendation quality for a single query. To measure the recommendation quality for an entire oracle (Q), we employ the *MeanReciprocalRank* (MRR) metric, as depicted in Equation 5.

$$MRR(Q) = \frac{\sum_{q=1}^{|Q|} RR(q)}{|Q|} \quad (5)$$

3.2 Example of Fitness Computation

To demonstrate how the MOP and MRR metrics are computed, we will use two real examples from our oracle (see Section 4.2). Query q_1 is a code snippet for a `hashCode()` method implementation found in a StackOverflow post. The oracle indicates two snippets, s_1 and s_2, as clones of q_1, both found in NetBeans source code. Query q_2 is the method `getNoFocusBorder()` included in a StackOverflow post, for which our oracle indicates three existing clones in the source code of JHotDraw and NetBeans. The links and references for the clone pairs discussed in this section are available in our replication package [41]. Let's assume that, for q_1, we have the known clones in the oracle $O(q_1) = \{s_1, s_2\}$ and the suggestions made by a certain clone detector $S(q_1) = \{s_1, s_2\}$. For q_2, we have $O(q_2) = \{s_3, s_4, s_5\}$ and $S(q_2) = \{s_1, s_5, s_3\}$.

To compute $MOP(Q)$, one needs to first compute the values of $OP(q_1)$ and $OP(q_2)$. For q_1, the clone detector suggested two clones ($|S(q_1)| = 2$). Hence, we need to compute the $Precision@k$ for $k = 1$ and $k = 2$. For $k = 1$, we only look at the first suggestion made by the clone detector, which is a true clone. Thus, $Precisions@1(q_1) = \frac{1}{1} = 1$. For $k = 2$, we look at the first and second suggestions, which are both true clones. Thus, $Precisions@2(q_1) = \frac{1+1}{2} = 1$. As a result, $OP(q_1) = \frac{1+1}{2} = 1$.

For q_2, we compute $Precision@k$ for $k = \{1, 2, 3\}$. In this case, $Precision@1(q_2) = \frac{0}{1} = 0$ because s_1 is not a true clone. Next, $Precision@2(q_2) = \frac{0+1}{2} = 0.5$ and $Precision@3(q_2) = \frac{0+1+1}{3} = 0.66$. Hence, $OP(q_2) = \frac{0+0.5+0.66}{3} = 0.38$. Finally, $MOP(Q) = \frac{OP(q_1)+OP(q_2)}{2} = \frac{1+0.38}{2} = 0.69$.

To compute $MRR(Q)$, we need to compute the *ReciprocalRank* indicator for q_1 and q_2. Regarding $RR(q_1)$, we identify the rank of the first true clone suggested in $S(q_1)$. The first suggested true clone is s_1 at rank 1st. Hence, $RR(q_1) = \frac{1}{1} = 1$. In the case of $S(q_2)$, the first suggested true clone is s_5 at rank 2nd. Thus, $RR(q_2) = \frac{1}{2} = 0.5$. As a result, $MRR(Q) = \frac{RR(q_1)+RR(q_2)}{2} = \frac{1+0.5}{2} = 0.75$.

3.3 The MC3 Problem Formulation

Given a rank-based clone detector, the Multi-objective Code Clone Configuration (MC3) problem consists of searching for a parameter configuration that maximizes the *MeanOverallPrecision* and *MeanReciprocalRank* metrics, as defined in Section 3.1. Consider $\vec{x_S}$ the parameters that configure a clone detector S to be subjected to an oracle of queries Q. The MC3 problem is defined in Equation 6.

$$MC3 \Rightarrow \begin{cases} \max f_1(\vec{x_S}) = MOP(Q) \\ \max f_2(\vec{x_S}) = MRR(Q) \end{cases} \quad (6)$$

4 EMPIRICAL METHODOLOGY

In this paper, we set out to answer the following research questions. The rest of this section depicts our methodology including the clone detector, the oracle, and the search algorithms.

- *RQ1: What is the performance of search algorithms in the Multi-objective Code Clone Configuration problem?*
- *RQ2: How do the optimized configurations compare to the default configuration?*

4.1 Siamese

Siamese is a scalable rank-based code clone search tool that employs multiple techniques to improve the accuracy and scalability of its clone detection [30]. It relies on multiple code representation, query reduction (QR), and a customized ranking function. Siamese works in two phases: indexing and retrieval.

In the indexing phase, the corpus of source code that will be searched for clones is input into Siamese. Siamese works with four different code representations: r_0—original source code text, r_1—n-grams of code token with no renaming, r_2—n-grams of code token with the identifier, literal and type-token renaming, and r_3—n-grams of code token with all tokens renamed.

In the retrieval phase, a code snippet serves as a query to search for clones. The query is processed similarly to the indexing phase. After getting the query's four code representations, Siamese performs query reduction by removing irrelevant search tokens and

Table 1: Siamese parameters used in this study.

Parameter	Values
n-gram size (nGS)	{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}
QR threshold (qOr, qT1, qT2, qT3)	{2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
boosting (bOr, bT1, bT2, bT3)	{-1, 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
minCloneSize (mCS)	{6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
simThreshold (sTh)	{{10%,20%,30%,40%}, {20%,30%,40%,50%}, {30%,40%,50%,60%}, {40%,50%,60%,70%}, {50%,60%,70%,80%}, {60%,70%,80%,90%}, {10%,30%,40%,50%}, {20%,40%,60%,70%}, {30%,60%,80%,90%}}

n-grams. The *qr threshold* is used as a cut-off threshold for making such removal. The four representations after query reduction are used to retrieve similar code snippets. Finally, the similarity threshold is used to filter out the clones with lower similarity than the threshold. If there are no clones above the similarity threshold, Siamese does not make any recommendation.

Table 1 depicts Siamese's parameters used in our empirical study. Both the *qr threshold* and the *boosting* parameters are composed of four individual parameters, one for each code representation. Even though they are treated as separate parameters in the optimization process, these parameters have the same overall purpose and can assume the same values. We selected the same parameters used in Siamese's original publication [30] and also expanded by considering additional values for each parameter.

4.2 Clone Oracle

The oracle employed in our study is a manually annotated set of real-world online clone pairs between StackOverflow (SO) posts and open-source software projects written in Java [32]. The code snippets from SO come from accepted answers to Java-related questions. The Java open-source projects come from the Qualitas benchmark [42]. For each SO code snippet, the oracle indicates one or more snippets in the Qualitas' projects that have been validated as real clones. For each clone pair, the oracle indicates the start and end lines for both the SO post and the Qualitas code file.

In the original manual annotation [32], the clone pairs were categorized into different groups. For this study, we excluded the trivial clone pairs categorized in the BP (boiler-plate or IDE auto-generated), IN (inheritance/interface implementation) and NC (not clones) groups. For more details regarding the manual annotation and categories, we refer to the original publication [32]. This resulted in a total of 553 clone pairs between SO and Qualitas' projects, associating with 323 unique SO snippets.

To evaluate Siamese using the oracle, we had to download and preprocess the SO code snippets and the Qualitas projects listed in the oracle. The code snippets from SO were downloaded using StackExchange[1]. A SO post may contain several code blocks, where each code block may contain several lines of code. Hence, for each SO post listed in the oracle, we extracted the code snippet within the start and end line according to the oracle. In the cases where a single SO post contained more than one snippet in clone pairs, we separately extracted the code forming individual SO snippets.

[1]https://archive.org/details/stackexchange

The Java projects from the Qualitas benchmark were downloaded from the official website[2]. We downloaded the release version 20130901r, which is the same one used to create the oracle [32]. In total, the Qualitas benchmark used in this empirical study contains 111 projects, including 166,709 Java files and a total of 19,614,083 lines of code. The only preprocessing step performed in the Qualitas code was the removal of code comments and the application of pretty printing[3] to normalize the code structure.

In summary, our evaluation dataset consists of 323 snippets from SO with known clones among the Qualitas projects. Thus, each SO code snippet is considered a query to be searched in Siamese. In the context of this paper, to evaluate a certain parameters' configuration, we need to: i) index all Qualitas projects into Siamese; ii) use Siamese to search each of the 323 queries and store the suggestions; and iii) compute the MOP and MRR metrics for the entire oracle.

4.3 Search Algorithms

For this empirical evaluation, we selected one multi-objective algorithm and three baseline algorithms. The multi-objective algorithm selected for this paper was NSGA-II [9]. We believe such a classic algorithm fits the exploratory nature of this evaluation, which seeks to find out how search algorithms perform in the new problem of multi-objective code clone configuration. For the baseline algorithms, we employed Grid Search, Random Search, and Bayesian Search. These algorithms are commonly used in parameter optimization tasks, especially in the machine learning domain [47, 50].

For the Grid Search algorithm, a domain specialist must select a pre-defined set of values in the search space (a grid), which the algorithm will exhaustively explore. We use code clone detection literature and their prior knowledge to define parameters they believed would yield good results. We chose the n-gram size values of {4, 6, 8}, qr threshold values of {8, 10}, the boosting values of {-1, 10}, the minCloneSize values of {6, 10} and the two sets of simThreshold values of {{20%,40%,60%,80%}, {30%,50%,70%,90%}}. The Grid Search algorithm performs an exhaustive search within the reduced search space defined in the grid. Hence, its execution time and results are deterministic. For this reason, to ensure a fair comparison between algorithms, the Grid Search execution time was used as the stopping criterion for the other algorithms. In the computational infrastructure used to run our study, the average time to evaluate a single parameter's configuration was 66 seconds. The total number of parameter combinations in the grid is 3,072. Hence, the Grid Search algorithm took approximately 54 hours to execute.

For the Random Search, we generated parameters' configurations in a random fashion. For each parameter, we randomly sampled a value using a uniform distribution. New random configurations were generated and evaluated until the stopping criterion of 54 hours was reached.

Different than Grid and Random Search, the Bayesian Search is guided by a fitness function. Hence, we had to employ a weighted approach to combine MOP and MRR into a single metric. To enhance diversity, we used three different weighting strategies. For Bayesian-EQ, the weights given to MOP and MRR are equal. Bayesian-MOP favors the MOP metric, with a {0.7, 0.3} weight ratio between MOP

[2]http://qualitascorpus.com/download/
[3]https://astyle.sourceforge.net

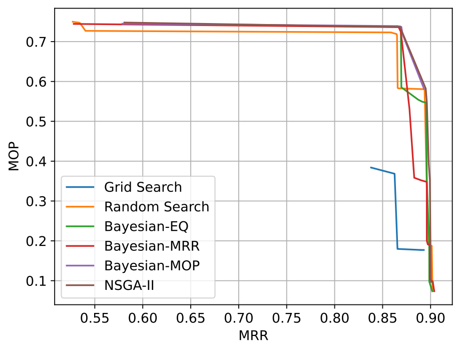

Figure 1: Pareto fronts obtained by the search algorithms.

and MRR, respectively. Similarly, Bayesian-MRR favors the MRR metric with a weight ratio of {0.3, 0.7} between MOP and MRR.

For NSGA-II, we used tournament selection, single-point crossover, uniform random mutation, and a population of size 10. We chose a small population size because of the time it takes to evaluate an individual's fitness. As previously mentioned, the average time to evaluate a single configuration (individual) is 66 seconds. In this scenario, a large population size would extrapolate the 54-hour time limit in a few generations. Thus, with the goal of better exploring the search space, we prioritized a larger number of generations over a large population.

Given the large amount of time to finish a single execution of a search algorithm, the non-deterministic algorithms included in this empirical evaluation (Random Search, Bayesian Search, and NSGA-II) were executed only once. We address this threat to the study's validity in Section 6.

After executing the search algorithms, we calculate the Pareto fronts of the algorithms. We also calculate the hypervolume (HV) from a reference point that dominates all other points, known as the *nadir* point, in order to compare the obtained Pareto fronts.. This guarantees that the extreme points of the Pareto front have a non-zero contribution to the total hypervolume [4]. For this study, using an offset of 0.01, the *nadir* reference point to compute hypervolume is $(MOP = 0.759, MRR = 0.913)$.

5 RESULTS

RQ1: What is the performance of search algorithms in the Multi-objective Code Clone Configuration problem?

Figure 1 presents the Pareto fronts obtained by the search algorithms included in this empirical evaluation. Overall, it is clear that the Grid Search algorithm achieved the worst results. All the points in its Pareto front are dominated by the Pareto fronts obtained by the other algorithms. This demonstrates the configurations designed by specialists were far from optimal, indicating the need for automated parameter optimization.

Random Search achieved much better results than Grid Search. When compared to the fitness-guided algorithms, the Pareto front obtained by Random Search is only slightly worse for the MOP metric and virtually the same for the MRR metric.

Based on a visual analysis, all the other algorithms achieved similar results. This is demonstrated by how their Pareto fronts are intertwined with each other. All the weighted Bayesian variations and NSGA-II achieved MOP and MRR values around 75% and 92%, respectively. To differentiate these algorithms, we need to look at their hypervolume.

Table 2 presents the hypervolume values for each search algorithm. In this context, the lower the hypervolume the better. As expected from the visual analysis, the hypervolumes computed for Grid and Random Search are the highest, indicating the worst-performing algorithms. The lowest hypervolume value is 0.01523, referring to NSGA-II, closely followed by Bayesian-EQ. With these results, among all the search algorithms included in our empirical evaluation, we can conclude that NSGA-II achieved the best performance for the problem of multi-objective code clone configuration.

Table 2: Hypervolumes of each search algorithm.

Search Algorithm	HV
Grid Search	0.330
Random Search	0.025
Bayesian-MRR	0.024
Bayesian-EQ	0.017
Bayesian-MOP	0.018
NSGA-II	0.015

Response to RQ1: *The NSGA-II multi-objective algorithm achieved the best performance for the problem of multi-objective code clone configuration.*

RQ2: How do the optimized configurations compare to the default configuration?

To answer RQ2, we compare Siamese's default configuration to optimized configurations found by NSGA-II, which was the best-performing search algorithm, as depicted in RQ1. For the optimized configurations, we selected the configurations in the Pareto front that achieved the highest MOP and MRR values. In addition, we also selected the configurations that strike the best balance between MOP and MRR. Table 3 presents the parameters' values for the default and optimized configurations followed by the MOP and MRR values they achieved.

As one can see from the table, the optimized configurations achieved higher values of MOP and MRR when compared to the default configuration. When comparing the optimized configurations with the highest MOP and MRR to the default, the optimized configurations show improvements of 71.08% and 46.29% for MOP and MRR, respectively. The configurations with the best balance between the fitness functions present better MOP (68.51% higher) and MRR (40.76% higher) compared to the default. When looking at the optimized configurations with the highest MOP and MRR, one can clearly see the trade-off between the fitness functions. This indicates how the objectives are conflicting, exacerbating the need for automated multi-objective optimization.

When analyzing the parameters' values, one can draw interesting observations. The *n-gram size* for all optimized configurations is 17, where the default configuration is 4. This may indicate that the clones between SO and Qualitas rarely contain added or removed tokens within the statements. Thus, the longer n-gram size of 17 can capture the sequence of clones better than the shorter one (i.e., 4). Considering the *qr threshold* for the original code representation (qOr), the values are vastly different between the configurations with the highest MOP and MRR values. While the configurations

Table 3: Siamese's default and optimized configurations found by NSGA-II followed by the MOP and MRR values.

Configuration	Parameters											MOP	MRR
	nGs	qOr	qT1	qT2	qT3	bOr	bT1	bT2	bT3	mCS	sTh		
Default	4	10	10	10	10	1	4	4	4	6	50%,60%,70%,80%	0.437	0.616
NSGA-II (MOP)	17	2	4	20	12	14	4	4	10	13	60%,70%,80%,90%	0.747	0.580
	17	2	4	20	12	20	4	4	10	13	60%,70%,80%,90%	0.747	0.580
NSGA-II (MRR)	17	12	12	20	12	14	4	4	10	7	10%,30%,40%,50%	0.096	0.902
	17	12	4	8	2	14	4	4	10	7	10%,30%,40%,50%	0.096	0.902
	17	12	4	20	12	14	4	4	10	7	10%,30%,40%,50%	0.096	0.902
	17	12	12	8	12	14	4	4	10	7	10%,30%,40%,50%	0.096	0.902
	17	12	4	8	6	14	4	4	10	7	10%,30%,40%,50%	0.096	0.902
NSGA-II (Balanced)	17	2	4	8	12	20	4	14	12	7	60%,70%,80%,90%	0.736	0.868
	17	2	4	20	2	20	4	14	12	7	60%,70%,80%,90%	0.736	0.868
	17	2	4	8	2	20	4	14	12	7	60%,70%,80%,90%	0.736	0.868
	17	2	12	20	2	20	4	14	12	7	60%,70%,80%,90%	0.736	0.868

with the highest MOP present $qOr = 2$, the ones with the highest MRR present $qOr = 12$. For the other *qr threshold* parameters (*qT1, qT2, qT3*), the values vary, even within configurations that achieved the same MOP and MRR values. This indicates that, for this oracle, the only *qr threshold* that influences the results is *qOr*. This observation complements the discovered large n-gram size of 17. It shows that the original code representation, i.e., the source code text, has the strongest effect on the clone search result.

One can see an interesting relationship between the *minCloneSize* and the *simThreshold* parameters. The default configurations establish a minimum clone size of 6 lines and a moderate similarity threshold. To achieve higher precision (MOP), the minimum clone size was raised to 13 with a higher similarity threshold. With this configuration, Siamese is more careful in its recommendations. To achieve high recall (MRR), the strategy is the opposite. While lowering the similarity threshold, the minimum clone size is raised to 7. This indicates that Siamese managed to achieve high recall looking for longer clones. Importantly, the balanced solutions achieved higher values of MOP and MRR while still employing stricter parameters, having higher clone sizes, and higher similarity thresholds.

Response to RQ2: *The optimized configurations achieved improvements of 71.08% and 46.29% for MOP and MRR, respectively. The balanced configurations achieved better MOP and MRR while employing stricter parameters.*

6 THREATS TO VALIDITY

Because of its preliminary and exploratory nature, our empirical study's validity is affected by some threats, which are discussed according to Wohlin et al. [46] guidelines on experimentation.

Conclusion and Internal threats: Due to resource and time constraints, we could only run each algorithm once. As a result, we were not able to perform statistical analyses to assess the algorithms' performance. To mitigate this threat, we performed a careful analysis of the results found during each algorithm's execution. *Construct threats:* We consider two threats to the construct validity. First, we chose Siamese, a state-of-the-art rank-based clone detector, as a clone search tool in this study. Second, our oracle is composed of real-world online clones from Stack Overflow and open-source projects. *External threats:* We employed a single clone detector and clone oracle. Thus, the results may not be generalized. The Qualitas corpus is over ten years old, so the code snippets in the Oracle may not represent modern Java applications. To improve the generalization, other rank-based clone detectors and additional oracles should be included in the study.

7 NEXT STEPS

In future work, we will add more multi-objective algorithms and increase the number of algorithm executions. We will also improve our infrastructure and parallelize Siamese's execution using surrogate models [10, 45] to reduce the time necessary to evaluate the parameter's configuration. Hence, we can execute the non-deterministic algorithms more times, allowing for statistical analyses. In addition, we will also conduct additional empirical studies with more clone detection tools and more oracles.

8 CONCLUSION

In this paper, we introduced emerging results of the Multi-objective Code Clone Configuration problem, where the parameters of a rank-based clone detector are optimized to balance precision and recall. We employed multi-objective and baseline search algorithms using a previously published oracle of real-world online clones between StackOverflow and open-source projects. We chose Siamese as our rank-based clone detector.

Our results showed that NSGA-II was the best-performing algorithm according to hypervolume. In addition, we also compared the NSGA-II optimized configurations to Siamese's default configuration. Our analyses demonstrate that the optimized configurations obtained better precision and recall, achieving 71.08% and 46.29% higher MOP and MRR, respectively. Our finding paves the way for better configurations of code clone detection tools in future empirical studies.

ACKNOWLEDGEMENTS

This work received partial funding from CNPq-Brazil, Universal grant 404406/2023-8, and support from CAPES - Funding Code 001.

REFERENCES

[1] 2015. Simian - Similarity Analyzer. http://www.harukizaemon.com/simian.

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2016. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–305.

[3] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.

[4] Nicola Beume, Boris Naujoks, and Michael Emmerich. 2007. SMS-EMOA: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research* 181, 3 (2007), 1653–1669.

[5] Magiel Bruntink, Arie Van Deursen, Remco Van Engelen, and Tom Tourwe. 2005. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering* 31, 10 (2005), 804–818.

[6] Muslim Chochlov, Gul Aftab Ahmed, James Vincent Patten, Guoxian Lu, Wei Hou, David Gregg, and Jim Buckley. 2022. Using a nearest-neighbour, BERT-based approach for scalable clone detection. In *ICSME '22*. IEEE, 582–591.

[7] James R Cordy and Chanchal K Roy. 2011. The NiCad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 219–220.

[8] KAPSER Cory. 2006. Cloning Considered Harmful'Considered Harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*.

[9] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[10] Katharina Eggensperger, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2015. Efficient benchmarking of hyperparameter optimizers via surrogates. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 29.

[11] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *CSMR '09*. IEEE, 219–228.

[12] Muhammad Hammad, Onder Babur, Hamid Abdul Basit, and Mark van den Brand. 2021. Clone-advisor: recommending code tokens and clone methods with deep learning and information retrieval. *PeerJ Computer Science* 7 (2021), e737.

[13] Muntasir Hoq, Yang Shi, Juho Leinonen, Damilola Babalola, Collin Lynch, Thomas Price, and Bita Akram. 2024. Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, Vol. 3487. ACM, 526–532.

[14] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2009. Clonedetective-a workbench for clone detection research. In *ICSE '09*. IEEE, 603–606.

[15] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[16] Cory J. Kapser and Michael W. Godfrey. 2008. Cloning considered harmful considered harmful: patterns of cloning in software. *Empirical Software Engineering* 13, 6 (Dec 2008), 645–692.

[17] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *ICSE '14*. 664–675.

[18] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY – A Code-to-Code Search Engine. In *ICSE '18*. 946–957.

[19] Rainer Koschke. 2007. Survey of research on software clones. *Duplication, Redundancy, and Similarity in Software - Dagstuhl Seminar 06301* (2007), 24.

[20] Jens Krinke and Chaiyong Ragkhitwetsagul. 2021. Code Similarity in Clone Detection. In *Code Clone Analysis*. Springer Singapore, 135–160.

[21] Jingyue Li and Michael D Ernst. 2012. CBCD: Cloned buggy code detector. In *ICSE '12*. 310–320.

[22] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *OOPSLA '17* 1 (Oct 2017), 1–28.

[23] Adriaan Lotter, Sherlock A Licorish, Bastin Tony Roy Savarimuthu, and Sarah Meldrum. 2018. Code reuse in stack overflow and popular open source java projects. In *ASWEC '18*. 141–150.

[24] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: code recommendation via structural code search. *OOPSLA '19* 3 (Oct 2019), 1–28.

[25] Christopher Manning, Raghavan Prabhakar, and Hinrich Schütze. 2009. *An Introduction to Information Retrieval*. Vol. 21. Cambridge University Press.

[26] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. 2002. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*. IEEE, 87–94.

[27] Jin-woo Park, Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. 2014. Surfacing code in the dark: an instant clone search approach. *Knowledge and Information Systems* 41, 3 (Dec 2014), 727–759.

[28] PMD. 2012. PMD's Copy/Paste Detector (CPD) 5.0. July 14 2012.

[29] Chaiyong Ragkhitwetsagul. 2018. *Code Similarity and Clone Search in Large-Scale Source Code Data*. Ph. D. Dissertation. University College London.

[30] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* 24, 4 (2019), 2236–2284.

[31] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. 2018. A comparison of code similarity analysers. *Empirical Software Engineering* 23, 4 (Aug 2018), 2464–2519.

[32] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. 2019. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering* 47, 3 (2019), 560–581.

[33] Chaiyong Ragkhitwetsagul and Matheus Paixao. 2022. Recommending Code Improvements Based on Stack Overflow Answer Edits. *arXiv preprint arXiv:2204.06773* (2022).

[34] Chaiyong Ragkhitwetsagul, Matheus Paixao, Manal Adham, Saheed Busari, Jens Krinke, and John H Drake. 2016. Searching for configurations in clone evaluation–a replication study. In *SSBSE '16*. 250–256.

[35] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (Jul 2013), 1165–1199.

[36] J. Regehr. 2010. Static Analysis Fatigue. https://blog.regehr.org/archives/259.

[37] Chanchal K. Roy and James R Cordy. 2007. *A Survey on Software Clone Detection Research*. Technical Report. 115 pages.

[38] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[39] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: detection of clones in the twilight zone. In *ESEC/FSE 2018*. 354–365.

[40] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *ICSE '16*. 1157–1168.

[41] Denis Sousa, Matheus Paixao, Ragkhitwetsagul Chaiyong, and Uchoa Italo. 2024. Replication Package for the paper: "Code Clone Configuration as a Multi-Objective Search Problem". https://zenodo.org/records/13694413

[42] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *APSEC '10*. 336–345.

[43] Ellen M Voorhees, Dawn M Tice, et al. 1999. The TREC-8 Question Answering Track Evaluation.. In *TREC*, Vol. 1999. 82.

[44] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for better configurations: a rigorous approach to clone evaluation. In *FSE '13*. 455–465.

[45] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2016. Two-stage transfer surrogate model for automatic hyperparameter optimization. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part I 16*. Springer, 199–214.

[46] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[47] Jia Wu, Xiu-Yun Chen, Hao Zhang, Li-Dong Xiong, Hang Lei, and Si-Hao Deng. 2019. Hyperparameter optimization for machine learning models based on Bayesian optimization. *Journal of Electronic Science and Technology* 17, 1 (2019), 26–40.

[48] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2019. How do developers utilize source code from stack overflow? *Empirical Software Engineering* 24 (2019), 637–673.

[49] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. 2017. Stack Overflow in Github: Any Snippets There?. In *MSR '17*.

[50] Tong Yu and Hong Zhu. 2020. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689* (2020).

[51] Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. 2021. Identifying versions of libraries used in stack overflow code snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 341–345.

[52] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *ICSE '19*, Vol. 2019-May. 783–794.

[53] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 886–896.