Surpassing Threshold Barriers: Evaluating the Efficacy of Nature-Inspired Algorithms in Detecting Applied Refactorings

Iman Hemati Moghadam
iman.hematimoghadam@utwente.nl
University of Twente
Enschede, The Netherlands

Matthias Sleurink
matthias.sleurink@gmail.com
University of Twente
Enschede, The Netherlands

Vadim Zaytsev
vadim@grammarware.net
University of Twente
Enschede, The Netherlands

ABSTRACT

Refactoring is a pervasive activity in software development, and identifying refactorings applied to a program is crucial to understand its evolution. Currently, automated tool support for identifying applied refactorings is available for different programming languages (e.g., Kotlin, Java, C++, Go, JavaScript and Python). However, majority of the proposed approaches relies on similarity thresholds, where choosing proper thresholds that work in all projects, if not impossible, remains to be challenging. To overcome such limitations, in this paper we propose a search-based algorithm implemented on top of `RefDetect` to mitigate its dependency on similarity thresholds. In this proposed approach, we pay less attention to choosing a proper threshold value and instead employ a nature-inspired algorithm called the *Andean Condor Algorithm* (ACA) to eliminate refactoring instances that have been erroneously identified due to an employed lower similarity threshold value.

The performance of the proposed approach was evaluated on 513 commits of 185 open-source Java applications and compared with an existing NSGA-based multi-objective approach, a greedy algorithm, and the original version of `RefDetect` and also `RefactoringMiner`, a state-of-the-art refactoring detection tool that operates without relying on thresholds. The obtained results show the effectiveness of the employed search-based algorithm, where it outperformed the other search-based approaches, particularly outperforming the NSGA-based approach with a notable 17% improvement in F-score. The proposed approach also obtained a slightly better F-score compared with the other two tools not based on computational search and uncovered 238 true refactorings not detected by those tools.

KEYWORDS

Automated refactoring mining, Andean condor algorithm, Java

ACM Reference Format:
Iman Hemati Moghadam, Matthias Sleurink, and Vadim Zaytsev. 2024. Surpassing Threshold Barriers: Evaluating the Efficacy of Nature-Inspired Algorithms in Detecting Applied Refactorings. In *Proceedings of the 10th International Conference on Computer Technology Applications (ICCTA 2024).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Refactoring, the process of changing the internal structure of the program without changing its behaviour, is known as an essential activity in software development [1]. If refactorings are applied correctly, it will facilitate the program's comprehension and its future adaption. Identifying refactorings applied to a program is also a known important activity in software development [2]. Knowledge about applied refactorings can help developers to better understand their program's evolution, which in turn facilitates the program's maintenance activities. This knowledge can also assist developers of refactoring tools to better understand how refactorings are applied in real-world scenarios, which in turn can result in designing refactoring tools that better match the developers' needs.

Currently, automated tool support for identifying applied refactorings is available for different programming languages including Java [3–6], Python [7, 8], Kotlin [9], Go [10], C [4, 6], and JavaScript [6]. Among the existing tools, `RefactoringMiner` [3] exhibits the best accuracy in terms of *precision* compared to other refactoring detection tools. `RefactoringMiner`'s superiority arises from its unique characteristic of not requiring any similarity thresholds for identifying refactorings [3]. `RefactoringMiner` employs a replacement technique and uses some predefined heuristics to match statements in the method body, without requiring user-specified thresholds. However, in recent studies [4, 6], it was observed that in cases where no replacement or heuristic is defined for a particular situation in the code, there exists a possibility that `RefactoringMiner` cannot detect the applied refactorings. However, defining all possible replacement types and heuristics even within a specific programming language (e.g. Java) is challenging.

Silva et al. [6] and Hemati Moghadam et al. [4] have recently introduced language-neutral techniques not restricted to the specification of programming languages. However, their proposed approaches rely on similarity thresholds to identify refactorings. As acknowledged by both papers [4, 6], chosen thresholds may negatively affect the accuracy of the approach especially when the project has undergone a significant number of refactoring as well as non-refactoring changes. While employing strong thresholds can prevent the detection of some valid refactorings, employing weak thresholds may result in false positive cases. To overcome such limitations, we propose a search-based algorithm built upon `RefDetect` to mitigate its dependency on similarity thresholds.

In the proposed approach, we pay less attention to choosing a proper threshold value and employ a low similarity threshold value which allows identifying refactorings that their corresponding entities changed significantly. Additionally, we employ a search-based algorithm to eliminate refactoring instances that have been erroneously identified due to the employed threshold. The presented search-based algorithm is an adaption of the recently introduced Andean Condor Algorithm [11] which is inspired by the natural food-search behaviour of Andean condors. In this study, we mimic

ICCTA 2024, 15–17 May, 2024, Vienna, Austria

the search behaviour of the Andean condor in its search for food to identify valid refactorings from a pool of candidate refactorings, in which the possibility of false positive cases can be high.

It is worth mentioning that while presenting the refactoring identification process as an optimisation problem has been previously investigated [12–18], no prior research applies Andean Condor Algorithm (ACA) for identifying applied refactorings. The most similar approach to ours is one presented by Fadhel et al. [14], who proposed a mono-objective genetic algorithm to identify the applied refactorings. The authors later extended their approach by incorporating a new objective in the fitness function and used the non-dominated sorting genetic algorithm (NSGA-II) to find the best tradeoff between objectives [16]. The study revealed the strength of the new algorithm, where NSGA-II produces refactorings with higher correctness compared to the mono-objective algorithm [16].

To assess the effectiveness of our approach, we conducted experiments on 513 commits of 185 open-source Java applications and compared its performance with the approach introduced by Kessentini et al. [16], and a greedy algorithm. We also compared the performance of our approach against `RefactoringMiner` [3], and `RefDetect` [4] which are considered state-of-the-art refactoring detection tools in Java. The results of our approach, including the detected refactorings and the software applications used in the experiments are provided as an online appendix [19].

In summary, our contributions are as follows:

(1) We propose a novel approach that employs the Andean Condor Algorithm to diminish the reliance of RefDetect on the selection of suitable similarity thresholds for refactoring identification.
(2) We identify and address shortcomings in the existing Andean Condor Algorithm, and enhance its optimisation.
(3) We compared our approach against a greedy algorithm, an existing multi-objective genetic algorithm and two non-metaheuristic tools – `RefDetect` and `RefactoringMiner`, providing a comparison of their respective performance and effectiveness.

Following this part, we discuss our search-based approach in Section 2, present experiment results in Section 3, and provide a survey of related work in Section 4. We discuss threats to the validity in Section 5 and conclude with future work in Section 6.

2 PROPOSED APPROACH

In this section, we begin by providing a general overview of the Andean Condor Algorithm, highlighting its key attributes, and discussing the adapted version for identifying refactorings.

2.1 Andean Condor Algorithm

The Andean Condor Algorithm (ACA) is a nature-inspired swarm intelligence algorithm developed based on the food-search behaviour of male Andean condors [11]. The behaviour of this bird varies depending on the seasons of the year. While in the autumn and winter seasons, the birds will stay close to their nest, in the spring and summer months, they fly further to find food [20]. The fundamental concept upon which the ACA is developed is the ability to adapt the search behaviour depending on the conditions that are defined by the direction of changes (improvement or reduction) applied to the fitness value. In its original study, the ACA exhibited superiority over BAT [21], MBO [22] and PSO [23] algorithms, and demonstrated a convergence pattern that adeptly avoids local optimums [11].

The ACA is a population-based algorithm where each solution in the population is represented as a condor. Each condor can either have *Intensification* or *Exploration* status, and the *movement* pattern of the condors is determined according to their respective statuses. Whilst the specifics of intensification and exploration need to be customised to suit the specific problem being investigated, the overall idea is that, in intensification, small(er) changes are made to a condor to generate or move to a neighbouring solution (local search), and in exploration, large(r) changes are made to a condor to explore new regions of the search space (global search).

Deciding on the status of condors, between intensification or exploration, is of crucial importance. To decide on the number of condors that will perform intensification or exploration, the algorithm employs two parameters: the *Distribution Parameter* (DP) which determines the percentage of condors that engage in the exploration process (e.g., 60%), and the *Percentage of Change* (PC) which determines the magnitude of change in DP in each iteration of the algorithm (e.g., 10%) [11]. In the first iteration of the algorithm, the solutions in the initial population are *sorted* from the best to the worst based on their fitness and then the potion of *lower-ranked solutions*, determined by DP, are engaged in the *exploration* process, and the remaining *high-quality solutions* will participate in the *intensification* process. Subsequently, at the end of each iteration of the algorithm, the status of the condors will be updated according to the direction of changes (improvement or reduction) made to the average fitness value.

As depicted in Algorithm 1, at the end of each iteration of the algorithm, the average fitness value of the resulting solutions is calculated (Line 1), and the value of DP is updated depending on whether the new average fitness value shows an improvement or reduction compared to the average fitness value of the previous population (Lines 2–12). The ACA will dedicate more condors to the exploration process if there is no improvement in the average fitness value (Lines 2–6 and 13). Consequently, it results in a reduction in the intensification rate (Line 14). In this case, the algorithm changes the status of the worst condors with an intensification status to the exploration status (Lines 19–21). This can be advantageous, as it allows the lower-performing condors, with an intensification status, to be moved further away in search of better solutions while encouraging the high-performing condors to intensify their search further. Nevertheless, what happens if the high-performing condors are trapped in local optimums or if the lower-performing condors are already in the area of improvement through local search but are forced to move away from it? Furthermore, the algorithm makes an erroneous decision if the reduction in the fitness value resulted from the exploratory condors, and while some improvements in the fitness value resulted from condors involved in the intensification process, this improvement is less than the negative value caused by the exploration. In subsection 2.3, we will discuss how the proposed ACA is adapted to prevent such issues and ensure effective optimisation.

Algorithm 1 Calculate and Update States

Inputs: Andean Conder Population (ACP), Population Size (N), Population Average Fitness (AF), Distribution Parameter (DP), Percentage of Change (PC)
Outputs: New Average Fitness (AF_{new}), Updated Andean Conder Population, and Updated Distribution Parameter (DP)

- Calculate the average fitness of the newly generated population.
1: $AF_{new} \leftarrow \frac{1}{N} \sum_{i=1}^{N} fitness(ACP_i)$
- Compare the new average fitness with the old one and adjust DP accordingly. A higher average fitness value indicates better outcomes.
2: **if** $(AF_{new} \leq AF)$ **then**
3: $DP \leftarrow DP + PC$ ▷ The exploration rate ↑, and the intensification rate ↓
4: **if** DP > 1 **then**
5: $DP \leftarrow 1$
6: **end if**
7: **else**
8: $DP \leftarrow DP - PC$ ▷ The exploration rate ↓, and the intensification rate ↑
9: **if** DP < 0 **then**
10: $DP \leftarrow 0$
11: **end if**
12: **end if**
- Calculate Quantity of Exploration (QE) and Quantity of Intensification (QI)
13: $QE \leftarrow N * DP$
14: $QI \leftarrow N - QE$
- Sort the new population from best to worst.
15: $ACP \leftarrow Sort(ACP)$
- Update the status of condors based on the QI and the QE.
16: **for** $(i \leftarrow 1; i \leq QI; i++)$ **do**
17: $ACP_i \leftarrow$ intensification status;
18: **end for**
19: **for** $(i \leftarrow QI + 1; i \leq QI + QE; i++)$ **do**
20: $ACP_i \leftarrow$ exploration status;
21: **end for**
22: **return** AF_{new}, DP, ACP

In contrast, when the average fitness increases, the exploration rate will be decreased (Lines 7–12) and the status of the best exploratory condors will be updated to intensification (Lines 16–18). This can be beneficial as the well-performing condors that successfully explored new solutions gain an opportunity to be improved through local search. However, what happens if an exploring condor is moved into an area that other condors with an intensification status are already in? In this case, the intensification effort might be useless. In subsection 2.3, we will discuss how the proposed ACA is adapted to prevent such issues.

2.2 Identfying Applied Refactoring using ACA

Our proposed approach, which shall be referred to as ACA throughout the rest of the paper, will be discussed in this section. The proposed algorithm as depicted in Fig. 1 is composed of two phases: Refactoring Identification and Refactoring Validation.

While the first phase is the same as the one implemented in `RefDetect`, the second phase is an extension to `RefDetect`. To recall the point mentioned earlier, as a low similarity threshold is used to identify refactorings, the likelihood of false positive refactorings depending on the project being investigated may be high. Therefore, as a follow-up to the first phase, the second phase uses the Andean Condor Algorithm to identify the valid refactorings from a pool of candidate refactorings detected in the first phase.

Phase i: Refactoring Identification: The aim of this phase is to identify the initial list of candidate refactorings. This phase consists of four steps, which are discussed briefly due to page limitations, and directing readers to Hemati Moghadam et al. [4] for full details.

Table 1: Parameter Initialisation: Values and Domains

Parameter	Domain	Initial value
Entity similarity threshold	$[0, 1] \in \mathbb{R}^+$	0.3
Distribution Parameter (DP)	$[0, 1] \in \mathbb{R}^+$	0.5
Percentage of Change (PC)	$[0, 1] \in \mathbb{R}^+$	0.1
Population Size (N)		50
Quantity of Exploration (QE)	$[0, N] \in \mathbb{N}^+$	QE = N * DP
Quantity of Intensification (QI)	$[0, N] \in \mathbb{N}^+$	QI = N - QE
Termination Criteria: iterations without fitness improvement		3

(1) **Initialise Parameters**: as the first step, as depicted in Fig. 1, the parameters of the algorithm including entity similarity threshold are initialised. Entity similarity threshold shows the least similarity that two entities (i.e., classes, fields, or methods) must have to identify them as the same. As shown in Table 1, the threshold for entity similarity used in this paper is **0.3**, which means if two entities have a similarity of more than 0.3 they are deemed to be the same. This value is more than twice as low as RefDetect's original threshold [4].

(2) **Inconsistency Detection**: as the second step, the two input program versions are compared with each other, and their inconsistency in terms of entities deleted from the previous version and those introduced in the new version are determined. The algorithm compares entities of the same type (e.g., class, field, etc.) and identifies discrepancies if an entity exists in one version but does not have a corresponding one in another version. Entities of the same type are compared based on their signatures. For instance, methods are compared based on their names, input parameters, and return types [4].

(3) **Entity Matching**: the goal of this step is to establish associations between inconsistent entities identified in the previous step. Indeed, entities differing between the two versions are compared for similarity, and those exceeding the predefined threshold are considered to be the same [4]. Note that as a low similarity threshold value is used, each entity can be matched with more than one entity in its corresponding program. Not all matches are correct, and depending on the number of changes applied to the program, the number of mismatches can vary.

(4) **Refactoring Identification**: in this step, candidate refactorings are identified by applying some predefined rules to the entities that have been matched in the previous step. For instance, a Move Method refactoring is detected if a method in the first version is matched with a method in the second version, but their classes differ. A Move and Rename Method refactoring is detected if the name of the methods is also different [4].

Phase ii: Refactoring Validation: The result of the first phase of the algorithm is a set of candidate refactorings. However, as a low similarity threshold value is used, the possibility of false positive cases is high. Hence, it is the goal of the second phase of the algorithm to eliminate erroneously detected refactorings and to achieve this goal, a search-based algorithm based on the Andean Condor Algorithm is employed. This phase consists of five steps:

(5) **Initialise Parameters**: the parameters of the ACA including the distribution parameter (DP), the percentage of change (PC), the population size (N), and termination conditions are initialised.

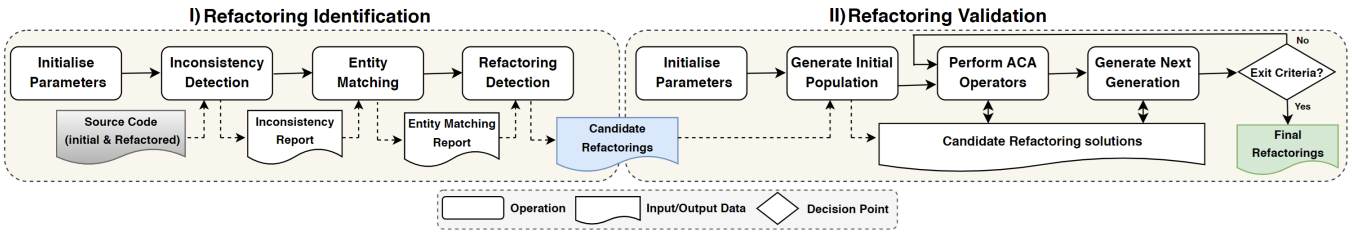

Figure 1: RefDetect Workflow: Steps in Refactoring Detection

The quantity of exploration (QE) and the quantity of intensification (QI) are also calculated as presented in Table 1.

(6) **Generate Initial Population**: in this step, the initial population is constructed, where each solution in the population comprises a sequence of refactorings *randomly* selected from the feasible search space (i.e., list of candidate refactorings detected in the first phase). Subsequently, each solution's fitness value is measured and the solutions are sorted according to their fitness from best to worst. A detailed description of the implemented approach is provided in subsection 2.3.

Steps 7–9 are repeated sequentially until termination:

(7) **Perform ACA Operators**: the search space is explored using intensification and exploration operators for better solutions in terms of the employed fitness function and simultaneously promoting solutions diversity. The operators are elaborately described in subsection 2.3.

(8) **Generate Next Generation**: the resulting solutions are sorted based on their fitness, and formed the next generation by replacing the previous population. The parameters of the algorithm including DP, QE and QI, are updated according to the direction of change (improvement or reduction) made to the average fitness value. The approach employed to update the quantity of intensification and exploration is discussed in subsection 2.3.

(9) **Termination**: if the termination conditions are fulfilled, the algorithm is terminated and the best solution according to the fitness value is returned. Otherwise, proceed to step 7. In the implemented approach the termination criteria is defined as the number of conservative iterations without improving the average fitness value and the fitness value of the best solution found so far. As shown in Table 1, if no improvement is observed after *three* conservative iterations, the algorithm terminates and the best solution across all iterations is returned.

2.3 Elaborating on the Phase II: ACA in Detail

This section describes in detail the proposed Andean Condor Algorithm employed for eliminating erroneously detected refactorings resulting from using a low similarity threshold.

Generating Initial Population: Initially, as discussed above, the initial population is constructed. Each solution in the population comprises a sequence of refactorings *randomly* selected from the feasible search space which is indeed the list of potential candidate refactorings. The number of refactorings each solution includes is an important factor that may differ for each application. Ideally, when the changes are all caused by refactorings, a value equal to the number of candidate refactorings is preferable, and in a scenario where changes are resulted due to non-refactoring changes, a lower value is preferable. However, as we have no prior knowledge about the type of changes, we employ a formula to estimate the size of solutions depending on the number of candidate refactorings. We ensure the size of the solution should be at least half the number of candidate refactorings and exhibits a rising trend as the number of candidate refactorings increases. It is worth mentioning that the size of solutions is not fixed and changes as the algorithm progresses.

Measuring Solution Fitness through Simulation: Following the generation of the initial population, the fitness value of each solution is measured. For each solution, its refactorings are sequentially applied to the initial version of the program, and their effect on reducing dissimilarity between the initial and final versions of the program is measured. In this way, the fitness function assigns a higher value to solutions that achieve a higher degree of similarity between the resulting program and the final version of the program.

One point to consider is that the similarity between two entities (e.g., classes or methods) is measured based on their name and type, as well as their relationships with the other entities (e.g., belonging to the same class/package, invoking/accessing by similar fields/methods, etc.), and the programming instructions (e.g., while loop, if-else statements etc.) are not taken into account. This method of measuring fitness is adapted due to the manner in which information is extracted from the source code by `RefDetect`. In fact, `RefDetect` avoids extracting language-specific details, and only extracts information found in all object-oriented programming languages. Taking methods as an example, `RefDetect` only extracts information about the signature of the method as well as fields and methods accessed by the method and it does not extract any details about programming language-specific statements (e.g., if, switch, for loop, return, try-catch, etc.) used in the method body [4].

Adapting Intensification and Exploration Rates: Upon assessing the degree of suitability of the initial solutions, they are *sorted* according to their fitness from best to worst. Subsequently, a portion of the *lower-ranked* solutions, determined by the DP, are selected to engage in the exploration process, and the remaining *high-quality* solutions take part in the intensification process. On the first iteration, as detailed in Table 1, we set the value of DP to 0.5 [11], which means 50% of solutions involved in the intensification process and the remaining involved in the exploration process. However, with the beginning of the second iteration, the intensification and exploration rates are updated according to how the average fitness value is changed compared to the previous population's average fitness value (refer to Algorithm 1 for details).

As explained in subsection 2.1, in the original version of ACA more condors will be dedicated to the exploration process if no improvement or a reduction in the average fitness value is observed. In contrast, if the new average fitness is better than the previous one, more condors will be dedicated to the intensification process (refer to Algorithm 1 for more) [11]. The idea is that when no more improvement is found, the worst intensifying condors can be employed for exploratory purposes, to find new areas in the search space for improvement (global search). However, when the condors find improvement, it means that their overall *location* is a favourable one, and they should search that area more (local search) [11]. Nevertheless, it is possible that the reduction in the average fitness resulted from the exploratory condors, and while some improvements in the fitness resulted from condors involved in the intensification process, this improvement is less than the negative value caused by the exploration. In the worst case, improvement may have resulted from the lower-performing intensifying condors, and these condors may be already in the area of improvement through local search but are forced to move away from it.

To address this issue, our proposed solution is that if the condors involved in the intensification process result in better solutions, then more focus should be directed towards the intensification, and if more improvement is found by exploration, then the condors are better served by doing more exploration in the next iteration. Importantly, at the end of each iteration, the solutions are sorted based on their fitness and if an exploratory solution shows a substantial fitness improvement, it will be shifted towards the beginning of the list of solutions, where it is more probable that its state will be modified to intensification, and the search for this solution will be directed towards a localised improvement. Subsequent sections detail the intensification and exploration operators.

Proposed Intensification Operator: Empirical studies reveal that refactorings are often applied in batches and they are not truly independent [24–26]. Two types of relationships may exist between refactorings: dependency and conflict. A *dependency* occurs if the execution of one refactoring is dependent on the execution of another refactoring, and a *conflict* arises if the execution of one refactoring prevents the execution of another one [27]. While in practice conflict between refactorings is not as frequent as dependency exists between them, in our study, as a low similarity threshold is used, the conflict between candidate refactorings can be substantial.

We use dependency and conflict relationships between candidate refactorings as a basis for implementing the intensification. In the intensification process, if a refactoring, such as ref_i within a solution, depends on a refactoring that is not currently a part of the solution, or if a refactoring not included in the solution, is dependent on ref_i, they will be added to the solution if they improve the fitness value. In addition, between two refactorings with a conflict, the one with a lower fitness value will be removed from the solution. To measure the effect of each refactoring on fitness, the refactoring and its dependents included in the solution are sequentially applied to the initial version of the program, and their effect on reducing dissimilarity is measured. Note that removing a refactoring eliminates its dependent refactorings as they will no longer be applicable. For instance, removing an Extract Class refactoring also eliminates its dependent refactorings, such as those moving methods and fields to the class created by Extract Class refactoring.

Proposed Exploration Operator: The exploration operator is designed with the primary objective of increasing the chance of finding an optimal solution and escaping from the local optima which may result due to the proposed intensification operator. To reach these goals, two types of exploration are implemented. Initially, to explore the search space, for each solution with an exploration status, one refactoring from the list of candidate refactorings will be *randomly* selected and added to the solution. In the exploration process, we prevent duplicate refactorings to be included in the solution. In addition, if the newly added refactoring results in two similar solutions in the population, it will not be accepted.

However, applying this operator alone may suffer the drawback of slow convergence, especially when the investigated program has undergone significant refactoring and non-refactoring changes, leading to a plethora of candidate refactorings. Moreover, in the preliminary experiments carried out in this study, we observed that the application of the aforementioned intensification and exploration operators can potentially lead to good solutions in the population containing different correctly detected refactorings, *especially when the program being investigated has undergone significant changes.*

While increasing the algorithm's iteration count or choosing a large solution size can mitigate the aforementioned problem, these approaches have their drawbacks. Worth mentioning that the original version of the Andean condor algorithm [11] cannot explore the search space by combining different solutions, as the crossover does in other metaheuristic approaches such as the genetic algorithm. To alleviate the aforementioned issues, we introduce a second exploration operator that occurs with a *certain probability*. The proposed operator allows *weaker* solutions with an exploration status to be combined with solutions with an intensification status, *randomly* selected from the population. The resulting solution will replace its weaker parent in the next generation.

We acknowledge that expanding the proposed operator to all exploratory solutions, and not limiting that to weaker exploratory solutions, may speed up the process of finding out an optimal solution. However, our primary goal is to adhere to the fundamental principles of the original Andean condor algorithm, except when it proves impractical in our context. The Andean condor algorithm is inspired by the movement pattern of the Andean condor when it searches for food. Hence, it is probable that condors with fewer food resources (lower fitness value) follow other condors with greater food resources (higher fitness value). However, it seems less likely that condors with moderate food resources follow this behaviour, as it is more reasonable for them to prefer to explore their immediate search space for new sources of food instead of moving far from their current location. Therefore, the proposed exploration operator is consistent with the movement pattern of the condor when it searches for food [20]. The second point is that if the new solution's fitness improves, it will be shifted towards the beginning of the list of solutions, where it is more probable that in the next iteration, its state will change to intensification, and the search for this solution will be directed towards localised improvement. This may help to escape from the local optima that may exist in the existing solutions with an intensification status.

3 EVALUATION

To assess the value of our approach (ACA), we conducted a set of experiments based on a dataset containing 513 commits of 185 Java applications. The dataset was originally created by Tsantalis et al. [3] and has been subsequently extended by Hemati Moghadam et al. [4]. To validate the efficiency of our proposed approach, initially, we compare ACA with an existing search-based algorithm introduced by Kessentini et. al [16]. The aim of this experiment is to show the distinctive features employed in both search-based approaches and provide an understanding of their strengths and weaknesses. We also compared ACA with a Greedy Algorithm in order to determine to what extent our approach is capable of outperforming a simpler heuristic-based algorithm. Additionally, we compared ACA with two existing state-of-the-art refactoring detection tools: `RefDetect` [4] and `RefactoringMiner` [3]. While `RefDetect` relies on similarity thresholds to identify applied refactorings [4], `RefactoringMiner` does not require any similarity thresholds to operate [3]. Overall, we designed our experiments with the aim of addressing the three following research questions:

- RQ1: Is ACA effective in identifying applied refactorings compared to an existing multi-objective search-based approach?
- RQ2: Is ACA effective in identifying applied refactorings compared to `RefactoringMiner`, and `RefDetect`, which are not based on metaheuristic search?
- RQ3: How does ACA perform in terms of execution time compared to the other approaches we are comparing it with?

To answer RQ1 and RQ2, the effectiveness of the employed approaches is evaluated using precision, recall, and F-score.

3.1 Approach Comparison for RQ1

To address RQ1, the efficiency of ACA is evaluated through a comparative analysis with a search-based method proposed by Kessentini et al. [16]. We begin by describing the employed approach [16], and then discuss the evaluation setup. Subsequently, we present the results and discuss the strengths and weaknesses of both approaches.

3.1.1 ***Algorithm Overview.*** In their recent research, Kessentini et al. [16] introduced a novel search-based algorithm capable of detecting and ordering refactorings applied to Java applications. What makes their approach uniquely significant is its independence from any reliance on a similarity threshold to identify refactorings.

The proposed approach is divided into two phases: *refactoring detection* and *refactoring refining* [16]. In the detection phase, all refactorings applicable to the initial version of the investigated program are identified. As an example, a method in a class can be moved to all other classes in the program using some candidate Move Method refactorings, provided that the pre-conditions of the refactoring are satisfied. For example, the Move Method refactoring requires as a pre-condition that no method with a similar signature is defined in the target class. However, this technique hugely increases the number of candidate refactorings and makes the space of possible refactoring sequences too large including a large number of false positive cases and not efficient to be explored exhaustively [16]. To overcome this issue, Kessentini et al. employed a search-based technique in the refining phase to determine the right refactorings among candidates.

The proposed refining phase is an adaptation of the Non-dominated Sorting Genetic Algorithm (NSGA-II) introduced by Deb et al. [28]. The proposed approach takes as input the list of candidate refactorings proposed in the first phase and produces as output the best refactoring sequence that represents a good compromise between the two conflicting objectives: *(i)* maximising the similarity between the initial and final versions of the program and *(ii)* minimising the number of refactorings in the solution [16]. In the proposed approach, each solution is represented as a sequence of refactorings.

While the population size (number of solutions) is a constant value and is specified at the beginning of the algorithm (e.g., 500), the length of each solution (number of refactorings) is a variable value. The solution size can be determined at random or by a person familiar with the system.[1] In the solution creation process, for each solution depends on its size a number of refactorings are *randomly* selected from the list of candidate refactorings. During the solution creation process in order to prevent the creation of non-performing refactorings, some pre-conditions are also examined. For instance, the pull-up refactoring requires as a pre-condition that the original class has a superclass or to be created before the pull-up refactoring can be applied. After generating the initial solutions, they are evolved over a series of iterations using the *crossover* and *mutation* operators, and the best solution in terms of the employed fitness function will be selected at the end of the algorithm (i.e., 100,000 solution evaluations).

In the proposed approach, a single-point crossover and a substitution technique as the mutation operator are used to generate new solutions. In the proposed crossover operator, for every pair of selected solutions (referred to as parents), a random point within the two solutions is identified and two new solutions are generated from their respective parents split at the identified crossover point. As mentioned, a substitution technique as the mutation operator is also used to replace some random number of refactorings in the solution with new refactorings. The mutation operator can modify the controlling elements of the selected refactorings (e.g., altering the method that is to be moved in a Move Method refactoring operation), but cannot replace the selected refactoring type with a new one (e.g., substituting a Move Method refactoring with an Extract Class refactoring). The crossover and mutation only occur with certain probabilities. Moreover, duplicated refactorings resulting from these operators are deleted and the pre-conditions of the refactorings must be met. At the end of each iteration, the worst solutions in the population will be replaced by the new solutions resulting from genetic operators, namely crossover and mutation.

3.1.2 ***Evaluation Setup.*** To investigate RQ1, ACA is compared with the search-based approach proposed by Kessentini et al. [16], referred to as the NSGA approach throughout the rest of the paper. We also compared ACA with a Greedy Algorithm (GR). The implemented GR starts with an empty list of refactoring and on each iteration randomly selects a refactoring from the list of candidate refactorings and includes it in the solution if the application of refactoring results in an *improvement* in the fitness. The algorithm terminates when no further fitness-enhancing refactoring is found.[2]

[1] Maximum 600 refactorings per solution [16].
[2] We used a GR algorithm, selecting the best fitness-enhancing refactoring per iteration, but simulating all refactorings each time led to impractical execution time.

We conducted the experiments on 513 commits of 185 open-source Java applications sourced from the datasets provided by Tsantalis et al. [3] and Hemati Moghadam et al. [4]. Worth mentioning that due to the stochastic nature of search-based approaches, we ran ACA, NSGA and GR *three* times each for each commit and presented the ones with median F-score results in the paper.

We ran NSGA according to the configuration specified by Kessentini et al. in their original paper (i.e., "crossover probability = 0.6; mutation probability = 0.4 where the probability of gene modification is 0.2; stopping criterion = 100,000 evaluations" [16]). However, in two cases, we made our assumptions as values provided in the paper were tailored for studied applications and not given as general values. We set the population size (number of solutions) to 500 and determined the solution size (number of refactorings) to a random number, ranging from half to twice the true refactorings included in the dataset for the application under investigation.

In the original version of NSGA [16], the search space comprises all refactorings applicable to the initial version of the program, regardless of whether the classes are changed or not. However, our pilot experiments have shown that apart from making the process very time-consuming, it also leads to weak accuracy. To ensure a fair comparison with ACA, we restricted the search space to classes which are changed between two versions of the program. As a result, *the search spaces that* ACA *and* NSGA *operate within are nearly identical*. Furthermore, in our pilot experiments with NSGA, we observed having no threshold to match entities, as proposed by Kessentini et al. [16], resulting in a *very weak* accuracy. To illustrate the issue, assume a situation that which a method named *foo* is deleted from a class and a new method called *bar* is created in that class. However, as no threshold is used the algorithm can incorrectly associate *foo* with *bar* and identifies an incorrect refactoring which is the renaming of the method *foo* to *bar*. Thus, to prevent obvious incorrect candidate refactorings, we selected a very weak threshold (i.e., 0.15) which is half than one used by ACA. A discussion of how this threshold affects the results is provided in the next section.

3.1.3 ***Results for RQ1***. Table 2 shows a comparison of the performance of ACA with the other approaches namely NSGA and GR. In total, 5,098 true refactorings of 23 different refactoring types were included in the dataset. As shown in Table 2, ACA outperforms the other two approaches, and achieving a 17% and 5% improvement in F-score when compared with NSGA and GR respectively. The Wilcoxon signed rank test (with $p < 0.05$) reveals that for all 23 refactoring types, ACA performs better than both NSGA (F-score: p-value = 0.00001), and GR (F-score: p-value = 0.00148). However, to gain further insight into factors that affect the performance of the employed approaches, we examined the results, as detailed below.

With the employed GR, an essential aspect needs to be brought to attention. As described earlier, in each iteration, GR adds a new refactoring in the solution if its application improves fitness. Obviously, if the refactoring is dependent on any other refactorings which have not yet been included in the solution, the refactoring will not be accepted as it is inapplicable. Therefore, GR adheres to the dependency between refactorings. However, the strength of ACA is that it comprehensively evaluates the combined effects of dependent refactorings. In fact, between two mutually exclusive refactorings which both have a positive impact on fitness, ACA gives more chance to the one that more refactorings are dependent on, while GR accepts the first selection it makes. Therefore, if the false positive refactoring is selected by GR, it will result in a reduction in both recall and precision.

Table 2: Comparison of ACA with NSGA and GR

	#TP	Precision	Recall	F-score
ACA	4,494	**0.87**	**0.80**	**0.83**
GR	4,260	0.81	0.76	0.78
NSGA	3,454	0.83	0.58	0.66

With the employed NSGA, we found that the *employed fitness function* and *utilised pre-conditions* may have a negative impact on the recall of NSGA. We observed that these factors can also have a positive effect on the precision of NSGA. However, due to the higher magnitude of their negative impact compared to their positive impact, it results in a reduction in the accuracy of the algorithm.

NSGA evaluates solutions based on two objectives: *(i)* maximising the similarity between the initial and final versions of the investigated program and *(ii)* minimising the number of refactorings in the solution. Since the NSGA generates multiple results, commonly referred to as the Pareto front, the algorithm employs a *knee point* strategy to effectively balance both objectives (i and ii) and select the final solution from the list of candidates [16]. However, there exist both benefits and drawbacks to the employed techniques. In the experiments, we frequently observed NSGA showing weak recall in cases where a large number of refactorings, with few non-refactoring changes, are applied to the program. For instance, while ACA demonstrated high recall for the java-algorithms and eucalyptus, NSGA only identified half of the actual refactorings. Upon closer inspection, we found that although NSGA identified almost all refactorings in some Pareto front solutions, its emphasis on solution size led it to select a solution that balanced both objectives, sometimes ignoring true refactorings especially those with low impact on similarity. The aforementioned aspects, however, help NSGA to achieve a high precision. Indeed, NSGA prioritises solutions with a few candidate refactorings and gives precedence to those with a higher impact on similarity, avoiding large solutions containing false positive cases with minor impact on similarity as observed in facebook-buck, hibernate-orm and Signal-Android. Worth mentioning selecting an appropriate solution size, as demonstrated by Kessentini et al. [16], can improve the accuracy of the algorithm. However, determining appropriate value can be challenging. Note that selecting an appropriate solution size is also an important factor in ACA. However, as the size of the solution is not considered as an objective in the fitness function, ACA achieved positive results more often than negative outcomes compared to NSGA.

In the experiments, we also observed situations where due to the employed pre-conditions, true refactorings included in the initial solutions were rejected by both ACA and NSGA. For instance, in bitcoinj, a method is extracted from methods in different classes using Extract and Move Method refactorings. However, among the detected refactorings only one can be applied as after applying the first refactoring, the subsequent ones are rejected because a similar method already exists in the class. While it is common practice to extract a method from different classes, relaxing this precondition might result in false positive cases. Changing the precondition to allow replacing a method only if the extracted

methods are identical, could be a potential solution. However, this is not always feasible, as the lines deleted from places the method extracted might be different. Overall, relaxing the pre-conditions can be a double-edged sword, as it might solve one problem but introduce new issues.

In our analysis, we also observed that while the employ thresholds help both ACA and *especially* NSGA to identify more valid refactorings, there are still valid refactorings not detected as the similarity between their corresponding entities were lower than the employed thresholds. In the majority of these cases, the refactored entities are changed significantly by *non-refactoring changes*, and only an extremely small threshold can help to match entities and identify the applied refactorings. However, using an extremely small threshold may result in a reduction in the precision of the algorithm as observed in the Move and Rename Class/Field/Method refactorings discussed in RQ2.

Findings: Incorporating interdependence between candidate refactorings during validation enables ACA to apply dependent refactorings simultaneously, and reduces the risk of accepting potentially erroneous and mutually exclusive changes. The utilised pre-conditions and thresholds generally enhance the accuracy of both ACA and NSGA, though negative impacts were also noted. Minimizing the number of refactorings in the solution, integrated into the fitness function employed by NSGA, proves effective in filtering false positive refactorings, but it results in a reduction in recall if the program undergoes few non-refactoring changes.

3.2 Approach Comparison for RQ2

To address RQ2, the efficiency of ACA is compared with `RefDetect` [4] and `RefactoringMiner` [3]. We used the configurations mentioned in their respective published papers to run each of these tools. We conducted experiments on a dataset with 513 commits from 185 Java applications. Both `RefactoringMiner` and `RefDetect` were previously evaluated on this dataset.

3.2.1 Results for RQ2. Table 3 presents a summary of the number of valid refactorings detected by ACA, as well as those that were not detected. Overall, ACA accurately detected 4,494 refactorings which are 88% of the total number of refactorings in the dataset. In fact, ACA detects 4% and 8% more valid refactorings in comparison to `RefDetect`, and `RefactoringMiner`, respectively.

A closer look reveals that out of all refactorings present in the dataset, 238 refactorings were only detected by ACA, and 187 refactorings were only detected by both ACA and `RefactoringMiner`, but not detected by `RefDetect`. Indeed, *employing a low similarity threshold value enables the* ACA *to identify 425 valid refactorings not detected by* `RefDetect`. `RefDetect` failed to detect these refactorings as significant changes that were applied to the refactored entities, leading to their similarity being lower than the thresholds used by `RefDetect`. However, as illustrated in Table 3, more than 600 valid refactorings were not detected by ACA, but detected *at least* by one of the other four tools. Remarkably, half of these refactorings, as shown in Table 3, are detected by `RefDetect`. Within this set of 302 undetected refactorings, some of them have arisen due to the limitations of the implemented pre-conditions, while others are because of the inefficacy of the implemented intensification and exploration operators.

To further discuss the factors that affect the efficiency of ACA, we focus on comparing ACA with the other two tools. The accuracy of each tool in detecting specific types of refactorings is presented in Table 4. The table presents the results for precision, and recall for each refactoring type. Overall, 23 refactoring types including composite ones (denoted by an asterisk) are supported in this study. As highlighted in Table 4 (shaded in grey), while ACA achieved the best *average* value in terms of recall for all class/field/method level refactorings, it achieved the weakest *average* precision value for class/method level refactorings. It is worth mentioning that ACA improved the average recall (+4%) compare to `RefDetect`, but by sacrificing some precision (−4%). Overall, across 23 refactoring types, ACA gained an *average* F-score of 83%, surpassing `RefDetect` and `RefactoringMiner` by 1% and 2%, respectively. However, we found no significant difference in the ranks of the F-scores being compared using the Friedman test. We found several reasons that can negatively affect the accuracy of ACA, but due to page limitations, we focus on briefly discussing the most important ones.

In our analysis, we found out that the main reason for the majority of *false positives* in Move and Rename Class/Field/Method refactorings is the employ threshold. In the majority of these cases, the original entity (class/field/method) is deleted from the program, but a new entity of the same type but a different name is created in the refactored program. However, as there are some similarities between these entities, such as both the deleted and new fields being invoked by some similar methods, which surpass the employed threshold, and the absence of dependencies and conflicts with other refactored entities, the algorithm incorrectly identifies them as matched and identifies an incorrect refactoring.

As presented in Table 4, in the seven refactoring types (i.e., 2, 3, 4, 8, 14, 20 and 23), ACA has a less favourable recall than `RefDetect`. This reduction partly happens as a result of the employed *simulation function*. As described earlier, the impact of each refactoring on fitness is measured by applying it to the program. Refactoring can be applied to the program if its pre-conditions have been validated. However, our observations, especially in composite refactorings, reveal instances where two entities (classes/fields/methods) with similar names were (renamed and) moved to a similar destination. In this case, while both refactorings were valid, only one of them was accepted. In fact, after applying the first refactoring, the second one will be rejected due to its precondition which forbids overwriting an existing entity in the target class.

We also observed a limitation in the intensification operator. We observed that while conflict (due to a low threshold) and dependency between candidate refactorings occur frequently, there were applications where no relationships exist between candidate refactorings. In such cases, the intensification operator has no impact on the algorithm convergence. This shows the importance of exploring a more comprehensive operator incorporating additional factors.

Findings: The employed low threshold value enables ACA to identify 425 refactorings not detected by `RefDetect`. However, we observed instances where the threshold either exceeded the requirement for valid refactoring or fell short, resulting in false positive cases. We also observed applications where no interdependency exists between refactorings, resulting in the intensification operator having no impact on the algorithm convergence.

Table 3: Comparison of ACA Performance in Detecting Refactorings

Total ACA Detected Refs.	ACA-Only Detected Refs.	Exclusively ACA & RefactoringMiner Detected Refs.	ACA Not Detected Refs.	RefDetect-Detected, ACA-Missed Refs.
4,494	238	187	615	302

Table 4: The Precision (P) and Recall (R) Results

		ACA		RefDetect		RefactoringMiner	
Refactoring Type	**#TP**	**P**	**R**	**P**	**R**	**P**	**R**
1. Rename Class	**54**	**0.89**	**0.89**	**0.94**	**0.81**	**1**	**0.82**
2. Move Class	**1052**	**1**	**0.96**	**1**	**0.98**	**1**	**0.98**
3. Move & Rename Class*	**23**	**0.55**	**0.74**	**0.79**	**0.83**	**0.95**	**0.83**
4. Extract Superclass	**32**	**1**	**0.88**	**1**	**0.91**	**1**	**0.97**
5 Extract Subclass	**5**	**1**	**0.80**	**1**	**0.60**	**1**	**0.60**
6. Extract Class	**91**	**0.84**	**0.84**	**0.78**	**0.76**	**0.91**	**0.23**
7. Extract Interface	**23**	**1**	**0.74**	**1**	**0.65**	**1**	**0.65**
Class-Level Refactorings	**1,280**	**0.90**	**0.84**	**0.93**	**0.80**	**0.98**	**0.73**
8. Rename Field	**126**	**0.85**	**0.75**	**0.87**	**0.78**	**0.98**	**0.71**
9. Move & Rename Field*	**13**	**0.45**	**0.38**	**0.36**	**0.29**	**1**	**0.38**
10. Move Field	**193**	**0.9**	**0.93**	**0.96**	**0.74**	**0.90**	**0.90**
11. Push Down Field	**27**	**0.96**	**0.89**	**1**	**0.89**	**1**	**0.85**
12. Pull Up Field	**132**	**1**	**0.87**	**1**	**0.85**	**1**	**0.92**
Field-Level Refactorings	**491**	**0.84**	**0.77**	**0.84**	**0.71**	**0.98**	**0.76**
13. Rename Method	**363**	**0.85**	**0.79**	**0.94**	**0.71**	**0.97**	**0.74**
14. Move & Rename Method*	**47**	**0.65**	**0.36**	**0.83**	**0.85**	**1**	**0.51**
15. Move Method	**194**	**0.81**	**0.88**	**0.89**	**0.88**	**0.97**	**0.86**
16. Push Down Method	**37**	**1**	**0.73**	**1**	**0.65**	**1**	**0.84**
17. Pull Up Method	**292**	**0.99**	**0.92**	**1**	**0.87**	**1**	**0.93**
18. Extract & Move Method*	**177**	**0.78**	**0.62**	**0.9**	**0.45**	**0.99**	**0.41**
19. Extract Method	**719**	**0.96**	**0.89**	**0.98**	**0.84**	**0.99**	**0.89**
20. Inline Method	**175**	**0.84**	**0.84**	**0.96**	**0.91**	**0.98**	**0.55**
21. Move & Inline Method*	**54**	**0.78**	**0.83**	**0.73**	**0.35**	**1**	**0.22**
22. Change Method Parameters	**1,238**	**0.98**	**0.92**	**0.96**	**0.92**	**0.96**	**0.87**
23. Move&Change Method Par's*	**31**	**0.87**	**0.87**	**0.91**	**0.94**	**0.91**	**0.68**
Method-Level Refactorings	**3,327**	**0.87**	**0.79**	**0.92**	**0.77**	**0.98**	**0.69**
All Refactoring Types	**5,098**	**0.87**	**0.80**	**0.91**	**0.76**	**0.98**	**0.72**

3.3 Results for RQ3

RQ3 relates to the performance of our approach in terms of execution time. Table 5 shows details of the time used by the five employed approaches to find refactorings applied in 513 commits. The most noteworthy observation is that `RefactoringMiner` outperforms the other approaches, with a total execution time of around 11 minutes. On the other hand, `NSGA` shows the weakest results, with a total time of 16 hours. Among the employed search-based approaches, `ACA` exhibits better results with 36 minutes in total. Indeed, `ACA` was able to find a solution in an average of 4.2 seconds, which is an acceptable time, and notably 66% of commits were processed by `ACA` in less than 1 second.

To gain further insight into possible performance bottlenecks of `ACA`, we closely reviewed 11 commits with an execution time higher than 1 minute. We observed that in these commits, several classes were removed from the program, and many new classes were introduced in the program. These changes resulted in significant differences between the two versions of the programs, and using a weak similarity threshold resulted in inconsistent entities being matched with more than one entity. Consequently, the algorithm spent more time identifying the final refactoring candidates.

Table 5: Execution Time Comparison

	Min. (s)	Max. (s)	Avg. (s)	Median (s)	Total (min)
`ACA`	0.1	291	4.2	0.5	36
`RefDetect`	0.06	78	1.5	0.4	12
`RefactorMiner`	**0.002**	**58**	**1.3**	**0.1**	**11**
`GR`	0.1	2054	12.9	0.5	110
`NSGA`	0.03	3367	113	7.9	966

This circumstance becomes even more notable in the amount of time expended by `GR`. While `GR` exhibits good performance in 67% of commits, its total time was almost *three* times more than `ACA`. In fact, in a significant majority of commits (67%), the number of non-refactoring changes is low and also there is no significant dependency between the applied refactorings. Therefore, `ACA` does not show any advantage over `GR`. However, with an increase in the number of non-refactoring changes or with an increase in the dependency between refactorings, the `ACA` shows its superiority by producing more highly accurate results at an acceptable time.

Findings: On average, `ACA` achieves an acceptable performance of 4.2 seconds, which is three times longer than that of `RefactoringMiner` and `RefDetect`, but three and 26 times faster than `GR` and `NSGA` respectively. While 66% of commits were processed by `ACA` in less than 1 second, we observed the amount of non-refactoring changes can negatively affect the performance of `ACA`, where it takes around 5 minutes to process cordova-plugin.

4 RELATED WORK

Using the differencing algorithms to identify changes applied between two versions of a program is the most employed technique for detecting refactoring. In this category, Xing and Stroulia develop `UMLDiff` capable of detecting refactorings in Java applications [5]. Their approach to refactoring detection was notably novel as they extracted UML models from the source code, and the refactoring detection process was defined on the extracted models, allowing the extension of the approach to support new programming languages. Silva et al. [6] and Hemati Moghadam et al. [4] also propose language-neutral refactoring detection tools capable of detecting refactorings in multiple programming languages including Java and C. Both these tools employed an intermediate representation of the source code and implemented the refactoring detection process based on this representation of the source code. All these three tools also share a similarity in their reliance on thresholds to match entities in different versions of programs. While their employed threshold values enable them to uncover refactorings not detected by other tools, it also results in lower precision levels. In the context of threshold-related challenges, the only tool unaffected by these limitations is `RefactoringMiner`. `RefactoringMiner` employs techniques such as argumentisation and replacement to match statements in two versions of the program. For instance, two method invocations with different names are considered identical if they have a similar receiver and a similar list of parameters [3]. Yet, defining all possible rules proves challenging and any omissions negatively impact the tool's recall [4, 6].

Search-based strategies offer an alternative approach to detect applied refactorings, primarily utilising pre- and post-conditions to guide the search and prioritising candidate refactorings based on minimising dissimilarity between programs. While early studies employed graph-based algorithms [12, 13], recent studies favour the genetic algorithm, where the fitness function exhibits significant variations among different approaches. For instance, while Mahouachi et al. [18] emphasised reducing metrics differences between two versions of the program, Fadhel et al. [14] focused on minimising structural discrepancies, and Kessentini et al. [29] examined both structural and textual similarities. Our approach closely aligns with Kessentini et al. [16]. However, while both approaches aim to minimise differences between programs, our approach incorporates interdependency between refactorings in the fitness function, whereas Kessentini et al. emphasise minimising the number of refactorings in solutions.

5 THREATS TO VALIDITY

The first threat to this study lies in the way we compare our approach with NSGA. As mentioned the implementation of NSGA is not publicly available, and we implemented this approach ourselves. While we follow the instructions provided by Kessentini et al. [16], there is a possibility that certain aspects might have been overlooked during implementation. Another concern is related to running each search-based algorithm three times on every commit and presenting the results with the median F-score in the paper. However, running each algorithm extensively, for instance, 30 times, may change the median result. The reason for selecting this strategy is to reduce the possibility of inaccurately validating newly introduced refactorings, consequently minimizing the potential for experimenter bias. To mitigate experimenter bias each new refactoring was validated by the first two authors of this study. However, the completeness of the employed dataset is not guaranteed, since there may be true positive refactorings not included in the dataset, and not detected by the employed search-based approaches.

Another concern is related to the introduced *intensification* and *exploration* operators. While the proposed operations were designed to address shortcomings in the existing Andean Condor Algorithm and enhance its optimisation, a more profound comprehension of potential weaknesses of the algorithm may lead to the development of more effective operators. Additionally, the parameters of the ACA are determined through a process of trial and error, and may not necessarily represent the optimal choices. Moreover, predefined pre-conditions for identifying refactorings, while beneficial, pose a risk to validity by potentially overlooking some valid cases.

6 CONCLUSIONS AND FUTURE WORK

In this work, we have presented ACA, a search-based refactoring detection algorithm built upon RefDetect to reduce its dependency on similarity thresholds. ACA mimicked the search behaviour of the Andean condor in its search for food to identify applied refactorings. To better align the algorithm with the goals of our study, we customised optimisation operators presented in the original version of the algorithm. ACA was compared with a multi-objective algorithm, a greedy algorithm and two other tools not rely on metaheuristic search. The results showed the superiority of ACA compared to the search-based algorithms and also achieved a slightly better F-score compared to RefDetect and RefactoringMiner. We reported cases in which ACA fails to detect the refactorings, and refinement is required to improve the precision. These concerns outline the roadmap for our possible future work.

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[2] Q. D. Soetens, R. Robbes, and S. Demeyer, "Changes as First-Class Citizens: A Research Perspective on Modern Software Tooling," *ACM Comput. Surv.*, vol. 50, no. 2, 2017.
[3] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, 2020.
[4] I. Hemati Moghadam, M. Ó. Cinnéide, F. Zarepour, and M. A. Jahanmir, "RefDetect: A Multi-Language Refactoring Detection Tool based on String Alignment," *IEEE Access*, vol. 9, 2021.
[5] Z. Xing and E. Stroulia, "Differencing Logical UML Models," *J. Autom. Softw. Eng.*, vol. 14, no. 2, 2007.
[6] D. Silva, J. Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A Multi-language Refactoring Detection Tool," *Trans. Softw. Eng.*, vol. 47, no. 12, 2020.
[7] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering Repetitive Code Changes in Python ML Systems," in *Proc. Int. Conf. Softw. Eng.* IEEE/ACM, 2022.
[8] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza, "PyRef: Refactoring Detection in Python Projects," in *Proc. Int. Conf. on Source Code Analysis and Manipulation.* IEEE, 2021.
[9] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin, "RefactorInsight: Enhancing IDE Representation of Changes in Git with Refactorings Information," in *Proc. Int. Conf. on Autom. Softw. Eng.* IEEE, 2021.
[10] R. Brito and M. T. Valente, "RefDiff4Go: Detecting Refactorings in Go," in *Proc. Brazilian Symp. on Software Components, Architectures, and Reuse.* ACM, 2020.
[11] B. Almonacid and R. Soto, "Andean Condor Algorithm for Cell Formation Problems," *Natural Computing*, vol. 18, 2019.
[12] J. Pérez and Y. Crespo, "Exploring a Method to Detect Behaviour-Preserving Evolution using Graph Transformation," in *Proc. Int. ERCIM Workshop on Software Evolution*, 2007.
[13] S. Hayashi, Y. Tsuda, and M. Saeki, "Detecting Occurrences of Refactoring with Heuristic Search," in *Proc. Asia-Pacific Softw. Eng. Conf.* IEEE, 2008.
[14] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer, "Search-based Detection of High-Level Model Changes," in *Proc. Int. Conf. on Softw. Maintenance*, 2012.
[15] I. Hemati Moghadam and M. Ó Cinnéide, "Automated Refactoring Using Design Differencing," in *Proc. Eur. Conf. on Softw. Maintenance and Reeng.* IEEE, 2012.
[16] M. Kessentini, U. Mansoor, M. Wimmer, A. Ouni, and K. Deb, "Search-based Detection of Model Level Changes," *Empirical Softw. Eng.*, vol. 22, no. 2, 2017.
[17] W. Kessentini, H. Sahraoui, and M. Wimmer, "Automated Metamodel/Model Co-Evolution: A Search-based Approach," *Inf. Softw. Technol.*, vol. 106, 2019.
[18] R. Mahouachi, M. Kessentini, and M. Ó. Cinnéide, "Search-based Refactoring Detection using Software Metrics Variation," in *Proc. Int. Symp. on Search Based Softw. Eng.* Springer, 2013.
[19] "Dataset," https://doi.org/10.5281/zenodo.10971389.
[20] E. F. Pavez, "Patrón de movimiento de dos cóndores andinos vultur gryphus (aves: Cathartidae) en los andes centrales de chile y argentina," *Boletín Chileno de Ornitología*, vol. 20, no. 1-2, 2014.
[21] X.-S. Yang, "A New Metaheuristic Bat-Inspired Algorithm," in *Nature inspired cooperative strategies for optimization.* Springer, 2010.
[22] E. Duman, M. Uysal, and A. F. Alkaya, "Migrating Birds Optimization: A New Metaheuristic Approach and Its Performance on Quadratic Assignment Problem," *Information Sciences*, vol. 217, 2012.
[23] J. Kennedy and R. Eberhart, "Particle Swarm Optimization," in *Proc. Int. Conf. on Neural Networks*, vol. 4. IEEE, 1995.
[24] T. Ferreira, J. Ivers, J. J. Yackley, M. Kessentini, I. Ozkaya, and K. Gaaloul, "Dependent or Not: Detecting and Understanding Collections of Refactorings," *IEEE Trans. Softw. Eng.*, vol. 49, 2023.
[25] I. Hemati Moghadam and M. Ó Cinnéide, "Resolving Conflict and Dependency in Refactoring to a Desired Design," *e-Inform. Softw. Eng. J.*, vol. 9, no. 1, 2015.
[26] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, 2012.
[27] H. Liu, G. Li, Z. Y. Ma, and W. Z. Shao, "Conflict-Aware Schedule of Software Refactorings," *IET Software*, vol. 2, no. 5, 2008.
[28] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, 2002.
[29] M. Kessentini and H. Wang, "Detecting Refactorings among Multiple Web Service Releases: A Heuristic-Based Approach," in *Proc. Int. Conf. on Web Services*, 2017.