Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

2024

Towards Algorithm Selection for Efficient Search-Based Software Engineering

Niranjana Deshpande
nd7896@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Deshpande, Niranjana, "Towards Algorithm Selection for Efficient Search-Based Software Engineering" (2024). Thesis. Rochester Institute of Technology. Accessed from

This Dissertation is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Towards Algorithm Selection for Efficient Search-Based Software Engineering

by

Niranjana Deshpande

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York

Towards Algorithm Selection for Efficient Search-Based Software Engineering

by

Niranjana Deshpande

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Naveen Sharma Date
Dissertation Advisor

Dr. Ernest Fokoué Date
Dissertation Committee Member

Dr. Qi Yu Date
Dissertation Committee Member

Dr. Mohamed Wiem Mkaouer Date
Dissertation Committee Member

Dr. Ricardo R. Figueroa Date
Dissertation Defense Chairperson

Certified by:

Dr. Pengcheng Shi Date
Ph.D. Program Director, Computing and Information Sciences

©[2024] [Niranjana Deshpande]
All rights reserved.

Towards Algorithm Selection for Efficient Search-Based Software Engineering

by

Niranjana Deshpande

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in
Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

In Search-Based Software Engineering (SBSE), metaheuristic search algorithms are used to construct software by combining components that fulfill diverse user and business requirements. These search algorithms typically employ randomization to accurately and efficiently evaluate numerous component combinations using limited computational resources. As a result, several search algorithms have been proposed to address SBSE problems, each with varying solution quality guarantees and computational resource usage. Recent research has demonstrated that search algorithms exhibit complementary behavior, that is, different search algorithms outperform each other on specific problem instances in terms of computational resource usage and solution quality. Problematically, current SBSE approaches do not leverage the complementary performance property and use a single search algorithm to solve all instances of a problem resulting in several inefficient or ineffective solutions. To address these limitations, we leverage a set of complementary search algorithms to fulfill complex user requirements and computational constraints for two SBSE applications.

Our research vision is to assist practitioners in building and maintaining effective and efficient software by providing automated, fine-grained search algorithm recommendations when employing SBSE techniques. To accomplish our vision, we study the impact of search algorithm selection on web service composition and third-party software library migration at the method level. Each of these SBSE problems has diverse requirements: search algorithms for web service composition need to be executed frequently to fulfill user-specified non-functional requirements, whereas library migration recommendations are used by developers to maintain software and fulfill functional requirements. We assess the relative strengths of 12 search algorithms on 6,144 web service composition and 7,200 third-party software library migration instances with multiple conflicting requirements. To accurately model variability in search algorithm performance, we leverage machine learning techniques to recommend algorithms for various SBSE instances. Our experimental evaluations demonstrate that our approach solves hard SBSE problems by fulfilling user requirements while minimizing computational resource usage. Additionally, we propose a transfer learning-based approach to predict algorithm performance across SBSE problems and mitigate challenges arising from a lack of training data. We conclude by outlining future research directions to build a generalizable algorithm selection framework for SBSE.

Acknowledgments

I am extremely fortunate to have had people who have supported and encouraged me through my Ph.D. Having had no prior exposure to research, I can honestly say that everything I know I have learned from my wonderful mentors and colleagues at RIT.

I want to begin by thanking my advisor, Dr. Naveen Sharma, for his guidance and unwavering support. Despite encountering failure and long periods of uncertainty, he always believed I would succeed.

This dissertation would not be possible without the sage counsel of my committee members Dr. Ernest Fokoué, Dr. Qi Yu and Dr. Mohamed Wiem Mkaouer. I am lucky to have collaborated with Dr. Qi Yu, whose insightful evaluations, incisiveness and persistence I hope to emulate in all my work. I am thankful to have worked with Dr. Mohamed Wiem Mkaouer, who has taught me the importance of scientific rigor, given me thoughtful feedback and offered advice on a variety of topics when I needed it. I am immensely fortunate to have Dr. Ernest Fokoué as a mentor. He has generously shared his wisdom on research, his infectious joy for statistical machine learning and boosted my confidence. In addition to my committee members, I thank Dr. Pengcheng Shi for his direct and practical advice as a result of which I have succeeded. I am sincerely grateful to Dr. Daniel E. Krutz for giving me detailed feedback on scientific writing, teaching me how to promote my work and encouraging a positive mindset through tough times. I also thank Dr. Ali Ouni, for having guided me through my work on API migration and for his thorough feedback. I have gained an important perspective on translating research to local communities by working with Dr. M Ann Howard and Dr. Ammina Kothari. I am grateful to Dr. Yin Pan, Kenn Martinez, Sudhir Khazanchi, Eric Mansfield, and Sandra Cirel Delaus for helping me become an effective teacher.

I would be remiss if I didn't mention my amazing colleagues, Dr. Nuthan Munaiah and Dr. Anthony Peruma, who continue to mentor me in various aspects of research thinking and career advancement. I'm also grateful to Dr. Danielle Gonzalez, Dr. Eman Abdullah AlOmar, Erika Mesh, and Dr. Joanna Cecilia da Silva Santos for their critiques on my presentations, advice on research, and navigating academia as a graduate student. I thank Milind Srivastava and Rijul Magu for helping me clarify my thinking, technical or otherwise, and for their encouragement.

All of the experiments in this dissertation were conducted using RIT's research computing cluster, and I am grateful to Sidney Pendelberry and the entire RC staff for their help and guidance. Additionally, I am grateful to Charles Gruener for patiently resolving issues on my local machine on multiple occasions. I would also like to thank our wonderful administrative staff: Lorrie Jo for her advice on key course requirements and life as an international student; Min-Hong and Siyuan for helping me navigate key milestones and the graduation process; Bridgette, Dawn, Denise and Marie for helping me get paid, coordinating meetings between busy committee members and assisting me with administrative problems.

I want to express my heartfelt appreciation for my friends: Abhishikta, Anvi, Bardh, Carlos, Carola, Cyril, Dishant, Ekta, Hitesh, Hrushi, Jordan, Karla, Katelynn, Khushbu, Navya, Nishi, Prakhar, Prashant, Renee, Rose, Sadie, Saurabh, Sharwari, Tappan, Vaishnavi, Viktoria, and Yiming. I sincerely apologize if I have unintentionally neglected to mention anyone, but please know that I am grateful for your presence and support.

I am truly grateful to Dr. Prahlad Vattipalle for encouraging me to pursue my ambitions, Dr. Megha Budruk for offering me advice on navigating academia, Dr. Pranjali Deshpande for encouraging me to pursue a Ph.D., and Anant Pathak for supporting me through various challenges over several years.

I am immensely thankful to Madhura and Varkki Chacko for opening their home to me and supporting my journey in more ways than I can count.

To my parents, Madhumati and Tushar, and my sister Tanaya, for their steadfast support and good humor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Billions of people use software applications daily and expect them to be robust and reliable. Software systems are increasingly constructed by combining numerous *components* in various configurations to fulfill complex functional (*e.g.,* payment processing, email etc.) and non-functional (*e.g.,* response time, availability etc.) requirements. Furthermore, software components and user requirements evolve frequently, necessitating periodic maintenance and re-construction at scale. As a result, software engineering (SE) practitioners frequently perform tedious and error-prone maintenance activities by assessing large numbers of component combinations. In fact, recent studies have found that developers spend up to 60% of their time understanding existing code, and up to 67% of a project's allocated budget maintaining pre-existing code [9]. Due to diverse requirements, numerous component combinations, and frequent software evolution, there is a need for automated tools to assist practitioners in engineering and maintaining software.

To address the challenges mentioned above, the Search-Based Software Engineering (SBSE) paradigm, heuristic *search* algorithms are used to accurately and efficiently solve complex software engineering problems. These search algorithms are computationally efficient because they use random exploration to evaluate numerous component combinations. As a result, SBSE techniques are widely used to find high-quality solutions using relatively fewer computational resources such as time and memory. For example, heuristic algorithms [10, 44, 64, 77, 118] have been applied to SE problems such as test-case generation [15], project scheduling [119], path planning in robots [122], web service composition [81] etc.Each heuristic algorithm provides near-optimal solution quality guarantees and uses variable computational resources to find suitable solutions. Existing work in SBSE typically uses a *single*, pre-determined heuristic algorithm to solve all instances of a specific SE problem.

Recent works [75, 76, 78, 79, 94] have demonstrated that heuristic algorithms exhibit *complementary performance* across different problem instances, that is, different algorithms outperform each other on specific instances of a problem. As a result, selecting one algorithm from a *complementary* set for specific problem instances reduces computational resource usage while delivering high-quality solutions. Problematically, current SBSE approaches do not consider this variability in heuristic algorithm performance and use a single algorithm for all problem instances leading to several inefficient solutions in terms of computational resource usage or solution quality. To address this limitation, we leverage the *complementary performance* of heuristic algorithms and demonstrate the benefits of selecting different algorithms for specific problem instances for web service composition and third-party library migration at the method-level (API migration). Our goal is to *assist practitioners in building and maintaining effective and efficient software by providing automated, fine-grained search algorithm recommendations when employing SBSE techniques.* By working towards this vision, we make the following contributions:

- **Self-Adaptive Service-Oriented Systems:** We propose a new adaptation mechanism that leverages complementary heuristic algorithm performance to efficiently construct applications. Service-oriented systems are constructed by combining various *services* that use web protocols (*e.g.,* HTTPS, REST etc.) to communicate. During service composition, numerous service combinations are evaluated to find solutions that fulfill diverse non-functional Quality of Service (QoS) requirements. However, unanticipated changes in service and application behavior such as new functionality and QoS fluctuations necessitate periodic re-composition. To address these challenges, heuristic algorithms are used to efficiently evaluate solutions to fulfill QoS requirements while reducing computational resource usage. In Chapters 4 and 5, we demonstrate the benefits of using a set of complementary heuristic algorithms in R-CASS, our self-adaptive framework for service-oriented systems. R-CASS profiles and dynamically selects heuristic algorithms from a complementary set for service composition instances. In this work, we take the first steps towards proposing a new adaptation mechanism for self-adaptive systems using algorithm selection. Specifically, an algorithm selection-based adaptation mechanism will help software systems self-optimize in uncertain environments.
- **Third-Party Software Library Migration at the Method-Level (API Migration):** In our work on API migration, we leverage heuristic algorithms to recommend software library methods with high precision. Software developers frequently use third-party software libraries to build reliable software and mitigate re-implementation efforts. These libraries are periodically updated to provide critical updates such as bug fixes, added features and functionality, and so on. During API migration, developers manually replace methods from older

libraries with their newer, updated counterparts by referring to source and target library application programming interfaces (APIs) and their related documentation. As a result, API migration is a tedious and error-prone process due to differences in library design, naming conventions, and large numbers of method mappings.

To address these challenges, we leverage heuristic algorithms to recommend source-target method mappings with high precision. In Chapter 6, we formulate API migration as a combinatorial optimization problem and leverage heuristic algorithms to recommend method mappings with high precision and recall. Compared to existing API migration approaches, our search-based approach reduces the requirement for expensive training routines and can generalize to different programming languages and frameworks. We build on our work in Chapter 7 and demonstrate that using an algorithm selection approach reduces the time required for recommendation while selecting mappings with high precision.

- **Algorithm Selection for Search-Based Software Engineering:** In SBSE literature, numerous heuristic algorithms [8,10,35,44] have been leveraged to address complex problems arising in SE. As a result, it may be difficult for practitioners to determine which algorithm to use for specific problems. We contribute to SBSE research by demonstrating the benefits of using a set of complementary algorithms to solve complex SE problems. We note that this set of complementary algorithms need not be the 'best' algorithms from the literature, but should use diverse search mechanisms so that they outperform each other on problem instances with different characteristics [139].

- **Application Areas for Algorithm Selection:** In our work, we apply algorithm selection for web service composition and API migration. To the best of our knowledge, the benefits of algorithm selection have not been demonstrated for these two SE problems. As a result, we contribute to the field of algorithm selection by discussing our approach and the challenges we faced when applying selection techniques to self-adaptive service composition and API migration. Our experimental evaluations illustrate the potential benefits of using algorithm selection for other SBSE problems in the future.

Through the tools proposed in this dissertation, we hope to make it easier for practitioners to:

- Identify and leverage a set of complementary heuristic algorithms that can be used together.

- Automate the process of selecting an appropriate algorithm from our complementary set for specific SE problem instances.

- Assist SBSE practitioners in constructing and maintaining robust, reliable software and reduce operational costs by optimizing computational resource usage.

1.1 Overview

We provide a brief overview of subsequent chapters in this dissertation.Figure 1.1 depicts our overall algorithm selection-based approach for web service composition and API migration. We gather requirements from software engineering practitioners for service composition and API migration. Each problem instance is associated with a schematic that consists of specific, configurable components. We model the performance of a set of complementary algorithms ('portfolio') using machine learning (ML) techniques trained on previously solved instances and select the algorithm that fulfills user requirements and minimizes computational resource usage.

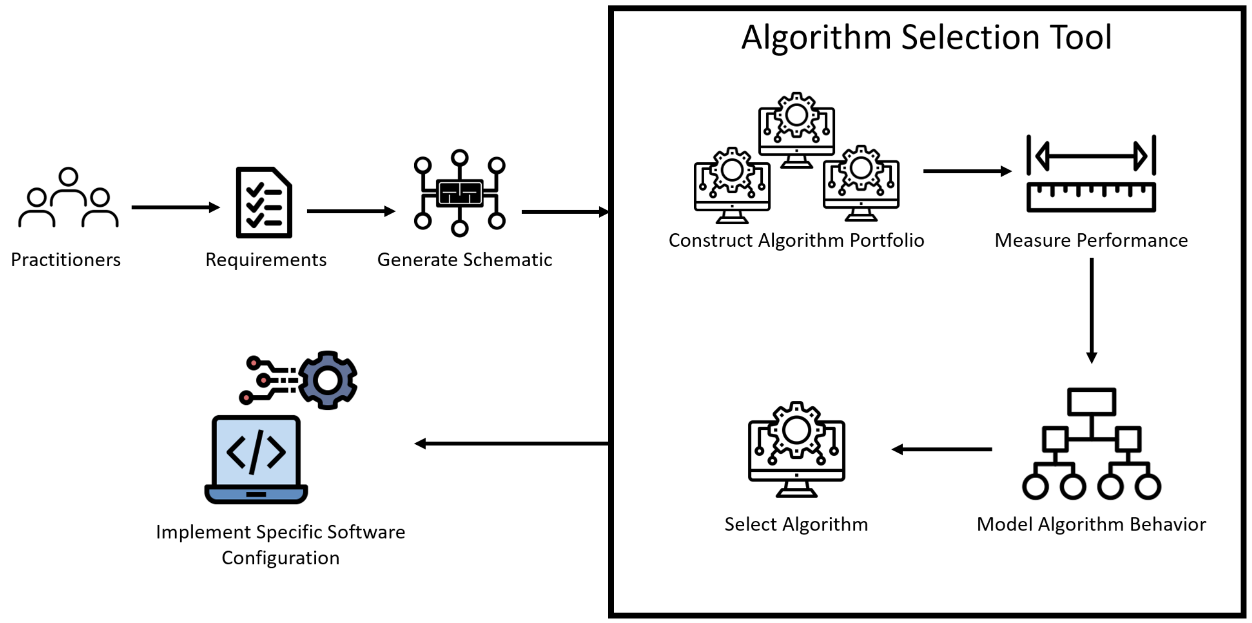


Figure 1.1: An overview of our algorithm selection approach for search-based software engineering problems.

In Chapter 4, we study the variability of heuristic algorithms for numerous service-oriented systems tasks. We propose the R-CASS framework that leverages classifiers and contextual multi-armed bandits to select an algorithm from a complementary set of service composition algorithms. Our experimental evaluations demonstrate that our approach reduces computational resource usage significantly while fulfilling user requirements. However, frequent fluctuations in web service attributes and changes to service-oriented systems applications pose key challenges when collecting training data for classifiers. We address this challenge in Chapter 5 by leveraging a transfer learning technique [132, 142] to 'transfer' knowledge of algorithm performance across domains. Specifically, we train our classifiers on extensive data from a different source domain and predict algorithm performance in the service-oriented systems domain. We assess R-CASS performance using the following research questions:

RQ 1 - Variability in Service Composition Algorithms
What behavior do Multi-Constrained Shortest Path (MCSP), Ant Colony System (ACS), Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) exhibit when solving composition tasks in terms of delivered solution quality, time and memory resources used?

RQ 2 - Assessing Model Accuracy
Which classification and regression models can best model Multi-Constrained Shortest Path (MCSP), Ant Colony System (ACS), Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) performance on composition tasks?

RQ 3 - R-CASS Efficacy
How effectively does a classifier-selected composition algorithm meet Quality of Service (QoS) requirements while minimizing computational resource usage as compared to a statically chosen composition algorithm?

RQ 4 - Evaluating Online Learning
Does online learning improve composition algorithm selection to reduce computational resource usage while meeting QoS constraints? If so, how do the three exploration strategies impact composition algorithm selection?

RQ 5 - Overhead
What is the overhead associated with selecting an algorithm for each composition task at runtime?

RQ 6 - Scope for Transfer Learning
How effectively can we transfer knowledge about Multi-Constrained Shortest Path (MCSP), Ant Colony System (ACS), Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) performance from TSP to service-oriented systems instances?

We also evaluate the benefits of heuristic algorithm selection for third-party library migration at the method level (API migration). In contrast to service-oriented systems, recommended APIs must fulfill specific functional requirements so that the underlying software functionality is not negatively impacted. As API migration has not previously been studied as a search-based problem, we formulate API migration as a combinatorial optimization problem in Chapter 6. In the subsequent Chapter 7, we analyze the complementary strengths of heuristic algorithms for API migration and the benefits of using an algorithm selection approach. We evaluate the efficacy of our approach using the following research questions:

RQ 7 - Search-Based API Migration
How accurately can our single-objective genetic algorithm (GA) approach recommend source-target method mappings?

RQ 8 - Comparing Multi-Objective Search
How effectively do UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, and MOEAD recommend source-target library method mappings for various migration rules?

RQ 9 - Assessing Similarity Schemes
What is the impact of using different similarity score schemes (method signature, documentation, and co-occurrence probabilities) when recommending source-target method mappings?

RQ 10 - Complementary Performance on API Migration Instances
How does UNSGAIII, RNSGAII, NSGAII, AGEMOEA and SMSEMOA performance vary across migration instances with different sizes and fitness distributions?

RQ 11 - Efficacy of Algorithm Selection for API Migration
What is the impact of selecting a different metaheuristic algorithm for specific migration instances?

1.2 Contributions

In this section, we enlist the papers published as part of this dissertation and release all datasets on GitHub. Each dataset was generated using considerable computational resources from RIT‘s research computing (RC) cluster. To motivate future research in algorithm selection-based approaches for software engineering, we make our datasets publicly available.

1.2.1 Datasets and Tools

1. **A dataset of service-oriented systems tasks**: In our work on self-adaptive service-oriented systems, we created a dataset of 6,144 service composition tasks and executed 4 popular service composition algorithms on each task. We sampled the popular WS-DREAM [148] dataset for response time and throughput attributes. Prior to this work, a similarly large

dataset of service composition tasks did not exist. All trained classifiers, contextual bandits code, and labeled datasets are publicly available at **https://github.com/niranjanadeshpande/contextual-bandits-composition-algorithm-selection**.

2. **Code for evaluating metaheuristic algorithms on API migration:** When leveraging single and multi-objective algorithms for API migration, we randomly generate migration instances containing valid and invalid mappings for each migration rule. We evaluated 8 algorithms using the PyMOO and MOEA frameworks on 9 migration rules and generated detailed results on individual algorithm performance. All code and results for this work are publicly available at **http://bit.ly/MOO-api-migration**.

3. **A dataset of API migration instances:** For our work on algorithm selection for API migration, we generate a dataset of 7,200 tasks and characterize each instance using 228 features. We record the performance of multiple evolutionary algorithms on each API migration task and generate a labeled dataset. The instance generation code, feature computation, and the labeled dataset are available at **https://github.com/niranjanadeshpande/algorithm-selection-api-migration**.

4. **Our dataset of Traveling Salesman Problem (TSP) instances for transfer learning:** Our experiments on using transfer learning for service-oriented systems required the creation of 27,267 TSP instances using the RUE and Netgen generation regimes. Additionally, we applied a TSP clustering algorithm on each instance and collected execution data for 4 algorithms on each instance. As a result, significant effort and computational resources were used to generate and validate our TSP dataset over multiple months. We note that this dataset was created using the same encoding and fitness functions used in service-oriented systems work. As a result, our dataset can be used by the service-oriented systems and algorithm selection community to evaluate new approaches. In the interest of reproducibility, we make our datasets and code available at **https://github.com/niranjanadeshpande/tsp-soa-transfer-learning**.

1.2.2 Publications

We have published 8 notable articles as part of this dissertation. With one exception (**contributes to the general area of self-adaptive systems*), each publication is directly related to our work on using algorithm selection to engineer self-adaptive service-oriented systems and API migration. We list the articles and abstracts as follows:

1. Search-Based Third-Party Library Migration at the Method-Level, In Applications of Evolutionary Computation, EvoApplications 2022, **CORE Rank B**

   In software development, third-party libraries are commonly used to reduce implementation efforts and errors, while delivering high-quality, reliable, and secure software. To support software evolution, newer libraries are continuously released to offer added features, and critical updates such as bug and vulnerability fixes. As a result, old (source) libraries and their methods must be replaced with their newer, updated counterparts (target libraries) during the library migration process. This is a time-consuming and error-prone process as developers must analyze both the source and target library's Application Programming Interface (API) documentation and implementation to replace every source API with target API(s). Recent studies do not provide generalizable guidelines on how each source API can be replaced with one or multiple target library APIs. To address this limitation, our work leverages evolutionary search algorithms to recommend APIs by (1) formulating API migration as a combinatorial optimization problem, and (2) using genetic algorithms (GA) to recommend suitable APIs during migration based on the method signature and documentation similarity, and co-occurrence. We conduct an empirical study on 9 popular library migrations from 57,447 open-source Java projects and demonstrate that GA can recommend multiple APIs for replacement with up to 100% precision for certain library migrations.

2. Third-Party Software Library Migration At The Method-Level Using Multi-Objective Evolutionary Search, Swarm and Evolutionary Computation, 2024, **Impact Factor 10**

   In this work, we address the limitations of our single-objective GA approach when recommending multiple APIs in many-to-many method mappings. In particular, we formulate library migration at the method level as a multi-objective combinatorial optimization problem and examine the performance of 7 multi-objective evolutionary algorithms: UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, and MOEAD. Our results demonstrate that UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA and MOEAD outperform GA top achieve 90%, 89%, 94%, 90%, 91%, 94%, and 71% precision on average, and 83%, 23%, 58%, 63%, 58%, 60% and 17% average recall respectively. All code and results are publicly available at: http://bit.ly/MOO-api-migration.

3. * Addressing Tactic Volatility In Self-Adaptive Systems Using Evolved Recurrent Neural Networks And Uncertainty Reduction Tactics, In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '22), **CORE Rank A**

Self-adaptive systems frequently use tactics to perform adaptations. Tactic examples include the implementation of additional security measures when an intrusion is detected, or activating a cooling mechanism when temperature thresholds are surpassed. Tactic volatility occurs in real-world systems and is defined as variable behavior in the attributes of a tactic, such as its latency or cost. A system's inability to effectively account for tactic volatility adversely impacts its efficiency and resiliency against the dynamics of real-world environments. To enable systems' efficiency against tactic volatility, we propose a Tactic Volatility Aware (TVA-E) process utilizing evolved Recurrent Neural Networks (eRNN) to provide accurate tactic predictions. TVA-E is also the first known process to take advantage of uncertainty reduction tactics to provide additional information to the decision-making process and reduce uncertainty. TVA-E easily integrates into popular adaptation processes enabling it to immediately benefit a large number of existing self-adaptive systems. Simulations using 52,106 tactic records demonstrate that: I) eRNN is an effective prediction mechanism, II) TVA-E represents an improvement over existing state-of-the-art processes in accounting for tactic volatility, and III) Uncertainty reduction tactics are beneficial in accounting for tactic volatility. The developed dataset and tool can be found at https://tacticvolatility.github.io/

Although we do not present this publication in our dissertation, this work mitigates challenges that may be encountered when deploying our algorithm selection-based approach at runtime.

4. Composition Algorithm Adaptation In Service-Oriented Systems, In Software Architecture, ECSA 2020, Communications in Computer and Information Science, **CORE Rank A**

   In service composition, complex applications are built by combining web services to fulfill user Quality of Service (QoS) and business requirements. To meet these requirements, applications are composed by evaluating all possible web service combinations using search algorithms. These algorithms must be accurate and inexpensive to evaluate numerous service combinations and services' fluctuating QoS attributes while meeting the constraints of limited computational resources. Recent research has shown that different search algorithms can outperform others on specific instances of a problem domain, in terms of solution quality and computational resource usage. Problematically, current service composition approaches ignore this property, leading to inefficient compositions. In this paper, we study the variability of 3 service composition algorithms: Genetic Algorithm (GA), Ant Colony System (ACS) and Multi-Constrained Shortest Paths (MCSP) on 105 composition requests using 8

QoS metrics sampled from the QWS dataset [5]. Our results demonstrate the complementary performance of MCSP, GA and ACS.

5. R-CASS: Using Algorithm Selection for Self-Adaptive Service Oriented Systems, 2021 IEEE International Conference on Web Services (ICWS), **CORE Rank A**

   We build on our work [56] and leverage the complementary strengths of MCSP, GA, PSO and ACS to adapt service-oriented systems at runtime. More specifically, we propose a self-adaptive framework, R-CASS, based on a composition algorithm selection framework to select one heuristic algorithm from a set for each composition task at runtime. We train random forest classifiers to predict algorithm performance using 408 service-oriented task features. Our evaluations on 6,144 composition requests demonstrate that R-CASS leads to more efficient compositions, reducing composition time by 55.1% and memory by 37.5%.

6. Online Learning Using Incomplete Execution Data for Self-Adaptive Service-Oriented Systems, 2022 IEEE International Conference on Web Services (ICWS), **CORE Rank A**

   Our previous work leverages pre-trained classifiers that become inaccurate at runtime due to changes in service QoS attributes and application evolution. As a result, unsuitable algorithms may be selected for service-oriented systems tasks that use excessive computational resource usage or do not fulfill user requirements. To address these limitations, we propose *online* composition algorithm selection using contextual multi-armed bandits that learn to select an algorithm for each composition task at runtime. Our evaluations demonstrate the benefits of online learning by reducing time and memory usage by up to 54.2% and 15.5% while fulfilling QoS requirements, compared to using a single composition algorithm for all tasks.

7. Algorithm Selection Using Transfer Learning, In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '21), **CORE Rank A**

   Per-instance algorithm selection has been shown to achieve state-of-the-art performance in solving Traveling Salesman Problems (TSP). Current approaches select different algorithms from a complementary set for each TSP instance leading to a significant reduction in computational time usage. In this work, we highlight how recent algorithm selection techniques apply to service composition, which is commonly posed as a TSP problem. However, unanticipated changes in the service composition environment pose a key challenge when collecting training data for all algorithms on unseen tasks. To address this problem, we propose the use of transfer learning techniques to improve classification accuracy in dynamic settings such as service composition.

8. Towards Algorithm Selection for Search-Based Software Engineering, ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023), **Doctoral Symposium, CORE RANK A***

   In this work, we summarize our results and contributions in a doctoral symposium paper that was presented at the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering in 2023. We share our findings with the software engineering community and elicit feedback to inform future research.

A summary of subsequent chapters is as follows: in Chapter 2, we list all the algorithms used in this study and provide a brief description of their mechanics. Additionally, we discuss the challenges associated with web service composition and API migration using motivating examples. Chapter 3 discusses current approaches that address web service composition and API migration, and their limitations. In Chapters 4 and 5, we leverage supervised learning and demonstrate the benefits of algorithm selection on a large dataset of 6,144 service composition tasks. We further study the impact of algorithm selection for SBSE on API migration in Chapters 6 and 7. Finally, we conclude in 8 and outline directions for future work.

Chapter 2

Background

In this dissertation, we examine the impact of using algorithm selection techniques [120] to assist software engineering (SE) practitioners in building effective and efficient software. As a result, our work leverages insights from SE, optimization, and machine learning (ML) to predict metaheuristic algorithm performance on SE problems. In this chapter, we discuss common terms and concepts used when discussing algorithm selection, API migration and service composition.

2.1 Background on Algorithm Selection

In this dissertation, we evaluate single and multi-objective *metaheuristic* algorithms on software engineering problems. The metaheuristic algorithms (metaheuristics or heuristics) used in our studies are inspired by natural processes *e.g.,* evolutionary algorithms that use mutation operators, swarm algorithms based on the flocking behavior of birds etc.We use the terms *metaheuristics* or *heuristics* interchangeably to refer to the set of algorithms used in our studies. Additionally, we assess the variability in algorithm behavior across diverse SE instances and demonstrate that heuristic algorithms exhibit complementary performance.

2.1.1 Complementary Performance

Heuristic algorithm selection techniques leverage the *complementary performance* property to identify suitable algorithms for specific problem instances. In a *complementary* set of algorithms, different algorithms outperform each other on specific instances of a problem as a result of their unique search mechanisms. Different algorithms use specific mechanisms to *search* the *space* of possible solutions. As a result, some solutions may be easier to find for specific algorithms in a given search space *e.g.,* a genetic algorithm may converge to a local optimum faster than an ant colony system. Recent studies [75, 78] have demonstrated that selecting one algorithm from a complementary set reduces computational resource usage while delivering high-quality solutions for TSP problems. In our work, we leverage the strengths of multiple *complementary* heuristics to find solutions in a wide range of search spaces for each of two SE problems. Our set of heuristics is static [78], that is, we determine a set of algorithms that do not change when we select one for each problem instance. Based on the findings presented by Xu *et al.* [139], we select algorithms that use comparatively different search mechanisms, rather than using the 'best' algorithms that achieve high performance on a majority of problem instances.

2.1.2 Fitness Landscape and Encoding

When using heuristic algorithms, fitness functions are commonly used to compare different solutions by assigning numeric values [95] that indicate solution quality. Each possible solution is assigned a fitness score that is used to compare it to a neighborhood of n other solutions. Additionally, each solution is represented using an *encoding* function *e.g.,* bit-string encoding is used to recommend method mappings during API migration. Fitness *landscapes* are generated by assigning fitness scores to each possible solution using common encoding and fitness function. Fitness landscapes can be used to visualize and compare metaheuristic performance for different problems and to understand the features of a specific search space. In fact, Malan *et al.* [94] recommend studying the characteristics of a problem function using fitness landscapes to gain insight into how search algorithms may perform across problem instances. In our work, we adopt various features from the algorithm selection literature [75,94] to characterize the fitness landscape when predicting algorithm performance. We discuss specific features and how they are calculated in subsequent chapters.

2.1.3 Single-Objective Algorithms

We enlist the single-objective algorithms used in our work and provide a brief description for ease of reference as follows:

- **Multi-Constrained Shortest Path (MCSP):** We study the MCSP algorithm proposed by Yu *et al.* [143] for service-oriented systems. MCSP exhaustively evaluates all possible solutions and always returns the optimal result. As a result, it can become computationally expensive on large problem instances.

- **Genetic Algorithm (GA):** The genetic algorithm [36] is motivated by the crossover and mutation operators from evolutionary processes. It uses these operators to recombine high-quality solutions at every iteration until a termination condition is met. GA is used for web service composition as well as API migration.

- **Ant Colony System (ACS):** The ant colony system algorithm was proposed by Dorigo *et al.* [57] and is inspired by the foraging behavior of ants. ACS periodically explores different solutions and assigns higher fitness to promising solutions.
- **Particle Swarm Optimization (PSO):** The particle swarm optimization algorithm was proposed by Kennedy *et al.* [74] inspired by the flocking behavior of birds.

2.1.4 Multi-Objective Algorithms

In contrast to single-objective algorithms, multi-objective algorithms optimize two or more objectives simultaneously. Multi-objective algorithms search for solutions that fulfill multiple objectives resulting in potentially longer search times. In our work, we evaluate 7 popular multi-objective algorithms for API migration:

- **Non-Sorting Genetic Algorithm (NSGAII):** NSGAII [51] is a popular search algorithm that leverages techniques such as non-dominated sorting and crowding distance to efficiently select and evolve candidate solutions. In this work, we choose NSGAII because it has been shown to perform well on problems with fewer objectives.
- **Indicator-Based Evolutionary Algorithm (IBEA):** We select IBEA [151] because it has been demonstrated to work well on problems with fewer objectives and is a relatively well-documented algorithm that allows us to study API migration-specific challenges and characteristics. Furthermore, we use the hypervolume metric to identify diverse solutions because we are interested in "knee points" where both conflicting objectives are balanced.
- **Reference-Based Non-Sorting Genetic Algorithm (RNSGAII):** We select RNSGAII [52] due to its ability to generate multiple solutions around user-specified reference points. Specifically, RNSGAII generates a set of solutions around a specified optimal point and uses Euclidean distance to evaluate generated solutions.
- **Unified Non-Sorting Genetic Algorithm (UNSGAIII):** The Unified NSGAIII algorithm [126] is a unified optimization approach that modifies NSGAIII for two and mono-objective problems. We evaluate this approach due to the performance of NSGAIII on many-objective problems and improvements made to the tournament selection procedure resulting in better performance on two objective problems.

- **Adaptive Geometry Estimation based Multi-Objective Evolutionary Algorithm (AGEMOEA):** The AGEMOEA algorithm [113] generalizes to different Pareto fronts and has been shown to outperform state-of-the-art algorithms such as NSGAIII, GrEA, MOEA/D, and AR-MOEA. We use this algorithm for our work on API migration and select this algorithm because the geometry of our Pareto front is unknown.
- **S-Metric Selection Evolutionary Multi-Objective Optimization Algorithm (SMSEMOA):** The SMSEMOA algorithm [24] aims explicitly to keep solutions that maximize hypervolume. At every iteration, SMSEMOA discards solutions with the least contribution to the dominated hypervolume. Since SMSEMOA relies on hypervolume maximization, we can evaluate the efficacy of using this indicator for our problem.
- **Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D):** The MOEAD algorithm [146] uses decomposition strategies to search for solutions fulfilling multi-objective constraints. We select MOEAD because it uses a different search strategy that divides the search space into subproblems and because it has demonstrated desirable convergence properties on multi-objective problems.

2.2 Self-Adaptive Service-Oriented Systems

Software is frequently constructed using a popular paradigm known as Service-Oriented Architecture (SOA) [29, 81]. SOA applications are built by combining different components that are deployed as internet communicable services, resulting in efficient and effective application development. During service composition, heuristic algorithms evaluate various service combinations to find solutions that fulfill complex user Quality of Service (QoS) requirements. These requirements consist of multiple non-functional objectives (*e.g.,* response time, throughput, and so on) that must be optimized resulting in an NP-Hard problem [17, 47]. Additionally, frequent fluctuations in candidate service QoS attributes and application evolution necessitate periodic re-composition using limited computational resources at runtime. To address these challenges, heuristic algorithms accurately and efficiently evaluate candidate service combinations. In the following sections, we discuss the QoS model for service composition, how we evaluate solution quality, and the challenges faced during service composition.

2.2.1 QoS Model for Service Composition

In service-oriented systems, an application is depicted as a Directed Acyclic Graph (DAG) [82, 143, 147]. As shown in Figure 2.1, different nodes or *abstract services* (in orange) in a DAG correspond to

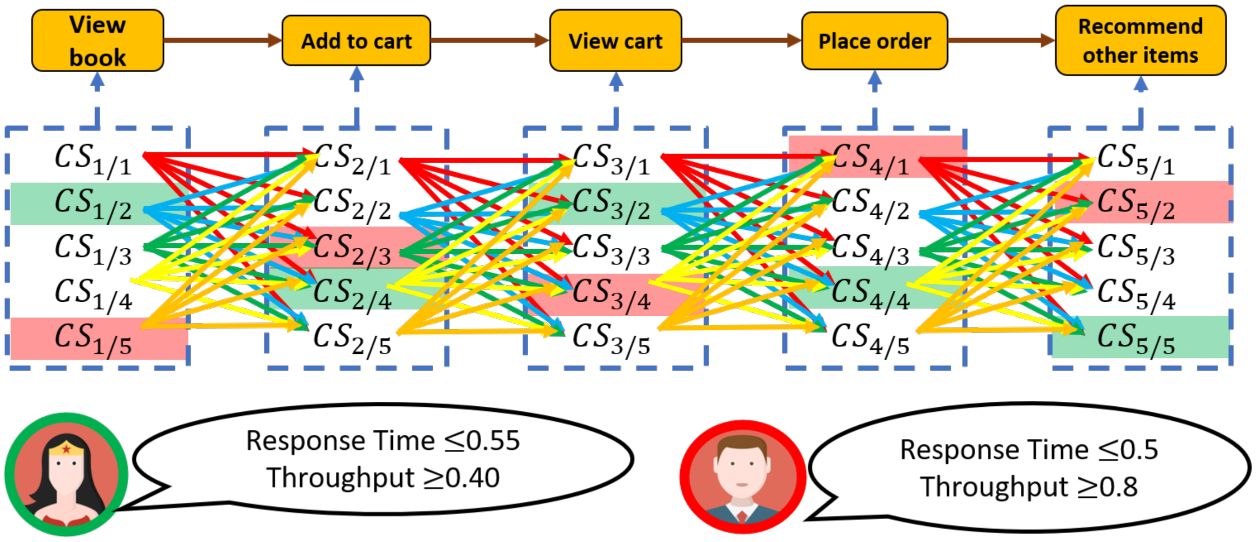

View book
Add to cart
View cart
Place order
Recommend other items
$CS_{1/1}$ $CS_{1/2}$ $CS_{1/3}$ $CS_{1/4}$ $CS_{1/5}$
$CS_{2/1}$ $CS_{2/2}$ $CS_{2/3}$ $CS_{2/4}$ $CS_{2/5}$
$CS_{3/1}$ $CS_{3/2}$ $CS_{3/3}$ $CS_{3/4}$ $CS_{3/5}$
$CS_{4/1}$ $CS_{4/2}$ $CS_{4/3}$ $CS_{4/4}$ $CS_{4/5}$
$CS_{5/1}$ $CS_{5/2}$ $CS_{5/3}$ $CS_{5/4}$ $CS_{5/5}$
Response Time ≤0.55 Throughput ≥0.40
Response Time ≤0.5 Throughput ≥0.8

Figure 2.1: We illustrate the challenges associated with service composition using a book buying software as an example. Heuristic algorithms are used to *compose* applications by evaluating various candidate service combinations to fulfill diverse user QoS requirements. The number of possible candidate service combinations increases exponentially with the number of abstract and candidate services. Moreover, fluctuations in candidate service QoS, and application evolution necessitate frequent re-selection of candidate services at runtime.

the available functionality *e.g.,* View Cart, Place Order etc.These *abstract services* are connected by edges that denote control dependencies and dictate the order of individual service execution. During service composition, a *concrete* service CS_{ny} is selected for each n^{th} abstract service AS_n. Concrete services for an application are selected to satisfy users' multi-objective QoS requirements. Each user provides their QoS needs $\vec{q}_{req} \leftarrow (q_1, q_2, ...q_z)$ and weights $\vec{w}_{req} \leftarrow (w_1, w_2, ...w_z)$ prioritizing z QoS attributes. So, a concrete service CS_{ny} selected for each abstract service AS_n has z QoS attributes. An application's QoS is an aggregate of each of the selected concrete services' QoS. For example, the response time of a composed application CA, is calculated by aggregating the response time of all selected concrete services $Q_{rt}(CA) = \sum_{i=1}^{n} Q_{rt}(CS_{iy})$. In this work, we consider response time and throughput as our QoS attributes, notating their objective functions in Table 2.1. QoS attribute values are normalized because they are each measured in different units and scales.

2.2.2 Measuring solution quality using the L_p function

Utility functions are commonly used to assess the quality of composed solutions and used within composition algorithms to guide concrete service selection. We use the L_p utility [3, 147] to weigh each edge in a DAG and select services connected by high utility edges. The L_p utility is a measure

Table 2.1: Aggregate QoS attributes describe whether an attribute should be minimized or maximized by a composition algorithm.

Metric	**Aggregate**	**Objective**
Response Time	$q_{rt}(CA) = \sum_{i=1}^{n} q_{rt}(CS_{iy})$	Minimized
Throughput	$q_{th}(CA) = \sum_{i=1}^{n} q_{th}(CS_{iy})$	Maximized

of the difference between the ideal QoS values and the current aggregate QoS values. It is defined as:

$$L_p(f(\vec{q})) = \left[\sum_{i=1}^{k} w_i | \frac{f_i^o - f_i(\vec{q})}{f_i^o} |^p + \sum_{i=k+1}^{z} w_i | \frac{f_i^o - f_i(\vec{q})}{f_i(\vec{q})} |^p \right]^{\frac{1}{p}} 1 \leq p \leq \infty$$

Here, objective functions $f_i(\vec{q})(1 \leq i \leq k)$ are the aggregate QoS (here, response time) to be minimized and $f_i(\vec{q})(k+1 \leq i \leq z)$ are the aggregate QoS (throughput) to be maximized. In this work, $z = 2$ because we consider two QoS attributes as shown in Table 2.1. Each QoS attribute in $\vec{q} \leftarrow (q_{rt}, q_{th})$ has a corresponding weight w_i, while p is set to 2. Note that, f_i^o is the ideal aggregate QoS for attribute i that is identified independently and used in the L_p metric to guide composition algorithms to a good result [3, 147]. As the L_p function measures distance from the ideal solution, we select a solution with the highest $1/L_p$ value. We note that any other utility function can be used in place of the L_p function depending on application requirements.

2.2.3 Challenges associated with service composition

To illustrate the challenges associated with multi-objective QoS driven service composition, we adopt the example of a book buying software as a motivating scenario. As shown in Figure 2.1, our book buying software implements certain functionalities (*e.g.,* view book, add to cart) and available candidate services. Users can request a software package with different Service Level Agreements (SLAs). The composing system must respond to diverse user requests - for example, user Bob may request faster response times and more throughput as compared to user Alice. As a result, different concrete services are selected (highlighted in red and green) for different users. Additionally, the composing system must not only fulfill user QoS requirements, but it also must account for computational resource usage. It is common for software applications to be deployed on Infrastructure as an Service (IaaS) instances, where computational resource usage directly impacts operational costs, and by extension, cost to users. For example, if composing an application requires several hours for completion, this results in significant system downtime and increased operational costs.

At runtime, selected concrete services may evolve to have different QoS and functional attributes. As a consequence, new service selections will have to be made using a composition algorithm to fulfill user requirements. Composition algorithms re-select concrete services for the entire DAG to meet QoS requirements, while also being computationally feasible. Because of large candidate service pools, some composition algorithms may become infeasible due to the computational resources required to compute an optimal solution. In such scenarios, approximate algorithms that use heuristics are preferred. However, these algorithms demonstrate variable behavior for different composition tasks depending on features such as size of DAG and features of the search space [75, 76]. So, using a *single* approximate algorithm for all composition instances leads to inefficient compositions, resulting in failure to meet application requirements and increased operational costs. Composing applications is an NP-Hard, combinatorial optimization problem [17,47], and the added consideration of computational resource usage makes composition at runtime more challenging. In this work, we address these challenges and demonstrate the positive effects of selecting different algorithms for different composition tasks for efficient runtime composition using the R-CASS framework.

2.3 Third-Party Software Library Migration at the Method-Level

We also demonstrate the benefits of algorithm selection on API migration, a complex software engineering activity with strict functional requirements. Third-party software libraries are routinely used to reduce implementation effort, reduce the number of bugs in code, and avoid redundancy [28] during development. Over time, these software libraries are updated to provide new features, functionality, and critical updates such as bug fixes [54]. As a result, developers routinely replace older, deprecated libraries with newer, updated libraries that are consistently maintained and updated. More specifically, developers replace each application programming interface (API) call that exposes a specific library function or procedure. This process of removing older (source) libraries' APIs and dependencies from source code and replacing them with APIs or methods from newer (target) libraries is known as library migration [129, 130].

2.3.1 Background

We begin by describing common terms and definitions used when refering to API migration as follows:

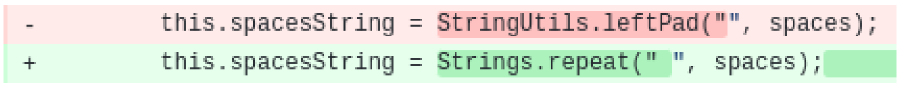
- **Library:** The term library refers to a software library containing a collection of objects and functions that can be accessed using an Application Programming Interface (API) [130].
- **Library Migration:** During library migration, we replace an older source library with a newer, updated target library by replacing all of its functional dependencies. [110, 129].
- **Migration Rule (Rule):** A migration rule, written as $source \rightarrow target$, specifies which $target$ library replaces a $source$ library's API calls. For example, the migration rule specifies that suitable replacements from $slf4j$ must be found for all methods and objects belonging to the source library $commons - logging$ [12].
- **Migration Mapping:** A migration mapping refers to a specific set of source methods that are replaced using one or many methods from the target library [13]. This is also referred to as a source-target method mapping or method mapping.

2.3.2 Challenges associated with API migration

To improve software functionality and reliability, developers manually examine API documentation to identify suitable source-target method mappings [11, 129] that do not alter underlying software functionality. However, due to differences in naming conventions and library designs, method mappings can have different cardinalities *e.g.,* multiple target library methods may be needed to replace one source method. To address these challenges, developers manually evaluate numerous source-target method mappings to identify correct replacements. However, evaluating all possible mappings is challenging because the number of mappings increases exponentially with the number of source and target library methods under consideration resulting in a *combinatorial explosion.* Manually evaluating and verifying all source-target method mapping combinations is tedious and requires developers to read the documentation of both source and target libraries. Additionally, developers must consider differences in library designs and naming conventions when evaluating potential source-target method mappings.

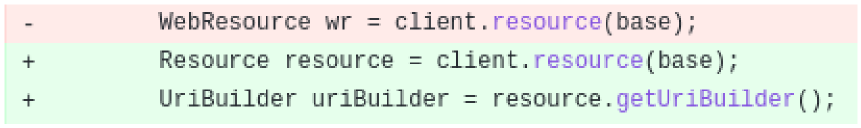
Figures 2.2, 2.4, 2.5 and 2.3 illustrate how one or more *source* (*e.g., commons-lang*) library methods are replaced by one or more newer, updated counterparts from a *target* (*e.g., guava*) library. These figures depict method mappings of different *cardinalities* between two popular Java libraries that were extracted from GitHub [1]. Differences in mapping cardinalities and library designs pose key challenges to library migration [14]. For example, in Figure 2.4, two source library methods are replaced with one target library method *i.e.,* it is a *many-to-one* method mapping. The number of

[1] `http://migrationlab.net`



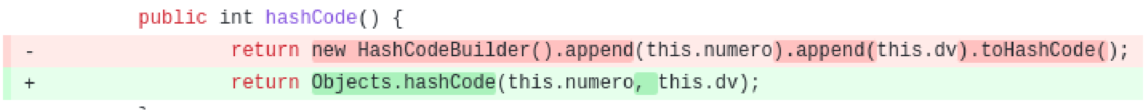
```
-         this.spacesString = StringUtils.leftPad("", spaces);
+         this.spacesString = Strings.repeat(" ", spaces);
```

Figure 2.2: An example one-to-one method mapping from the rule *commons-lang* → *guava*. The '-' sign on the left (highlighted in red) denotes source library methods that have been removed while the '+' sign denotes target library methods (also highlighted in green) that have been added.



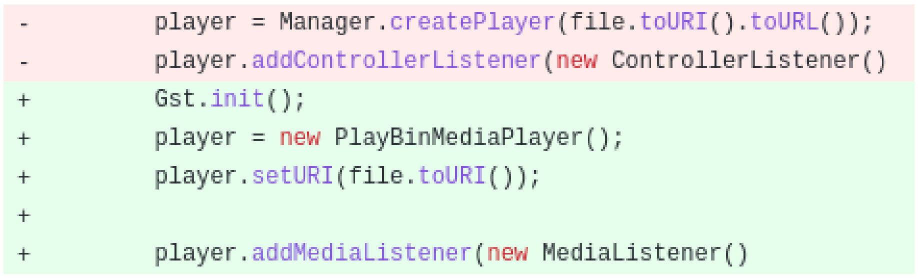
```
-         WebResource wr = client.resource(base);
+         Resource resource = client.resource(base);
+         UriBuilder uriBuilder = resource.getUriBuilder();
```

Figure 2.3: An example one-to-many method mapping from the rule *jersey-client* → *wink-client*. The '-' sign on the left (highlighted in red) denotes source library methods that have been removed while the '+' sign denotes target library methods (also highlighted in green) that have been added.



```
          public int hashCode() {
-                 return new HashCodeBuilder().append(this.numero).append(this.dv).toHashCode();
+                 return Objects.hashCode(this.numero, this.dv);
```

Figure 2.4: An example of a many-to-one method mapping from the rule *commons-lang* → *guava* where multiple source library methods are replaced by one target library method.The '-' sign on the left (highlighted in red) denotes source library methods that have been removed while the '+' sign denotes target library methods (also highlighted in green) that have been added.



```
-         player = Manager.createPlayer(file.toURI().toURL());
-         player.addControllerListener(new ControllerListener()
+         Gst.init();
+         player = new PlayBinMediaPlayer();
+         player.setURI(file.toURI());
+
+         player.addMediaListener(new MediaListener()
```

Figure 2.5: A many-to-many method mapping for the rule *jmf* → *gstreamer-java*. The '-' sign on the left (highlighted in red) denotes source library methods that have been removed while the '+' sign denotes target library methods (also highlighted in green) that have been added.

possible method mappings to be evaluated further increases when the number of target methods required is unknown. We further discuss each mapping example as follows:

- *One-to-one* mapping: Figure 2.2 depicts an example of a one-to-one method mapping mined from a project on Github when migrating from *commons-lang* → *guava*. In this example, the `leftPad()` method is replaced with exactly one method from the target library `repeat()`.

Here, the method names do not share any similarities due to differences in naming conventions. Due to these differences, it is difficult to correctly identify this mapping and there is a high likelihood of selecting incorrect replacements. Both these methods require the same number and type of inputs, but their names are different.[2]

- *Many-to-one* mapping: Figure 2.4 illustrates an example of a many-to-one method mapping where two source library methods are replaced with one target library method. More specifically, the `append()` and `toHashCode()` methods are replaced with the `hashCode()` method. This example illustrates how differences in library design impact mapping cardinality: only one target library method is needed to provide the same functionality as two source library methods. In contrast to the example above, the `toHashCode()` and `hashCode()` methods are easier to identify as they have similar method names. However, this mapping requires the addition of a "helper" method `append()` that is difficult to identify due to differences in source (`toHashCode()`) and target (`append()`) method names and functionalities. [3]
- *One-to-many* mapping: In Figure 2.3, one source library method (`resource()`) is replaced with two methods from the target library (`resource(), getUriBuilder()`). The `getUriBuilder()` method is difficult to identify unless co-occurrence probability is used because it is named very differently from the `resource()` method.[4]
- *Many-to-many* mapping: Figure 2.5 depicts an example of a *many-to-many* mapping where two source library methods are replaced with multiple target library methods. Specifically, the `createPlayer()` and `addControllerListener()` methods are replaced with `init(), PlayBinMediaPlayer(), setURI() and addMediaListener()` methods. Source and target methods are difficult to identify in this example, as each source and target method has completely different input parameter types and return types[5].

Due to the challenges discussed above, there is a need for automated and accurate third-party library migration tools to evaluate numerous source-target method mappings and recommend suitable method(s) for replacement. *In contrast to service composition, API migration has not previously been studied as a search-based problem.* We formulate library migration as a combinatorial optimization problem and assess the performance of single and multi-objective evolutionary algorithms in identifying suitable method mappings. Our goal is to correctly identify source-target method mappings of different cardinalities (*e.g.,* many-to-many, one-to-many etc.) by balancing the

[2]`https://github.com/querydsl/codegen/commit/1b777cdaa71f93cd2da5419bd09e25a4e379d240`
[3]`https://github.com/insula/opes/commit/5d3e9ac9cb9675d5de0840cfa3d42be81044efa1`
[4]`https://github.com/syphr42/libmythtv-java/commit/6e1cd7c06f3fa655be941e2b876ccef69253f4ee`
[5]`https://github.com/syphr42/libmythtv-java/commit/24544532420d9c1bf1b0cf2ad5aaad7ea585cd60`

similarity or *fitness* of selected method mappings and the number of recommended target library methods.

Chapter 3

Related Work

3.1 Algorithm Selection

Several works have leveraged algorithm selection techniques for a wide variety of computational problems [25,27,32,75,78,99,128,135,145]. Bezzera *et al.* [25] propose the AutoMOEA+ framework to design multi-objective algorithms by recombining algorithm components to solve different theoretical problems. Multiple works [27, 78, 99, 128] have demonstrated the benefits of using algorithm selection for theoretical problems such as traveling salesman problems, satisfiability problems, integer linear programming and so on. Other studies [79, 92, 123] combine hyperparameter and algorithm selection to recommend tuned search algorithms and machine learning pipelines.

Current literature addresses a wide range of computational problems using numerous heuristic algorithms with variable computational resource requirements and solution quality guarantees [26, 35, 141]. As a result, it can be difficult for practitioners to determine which algorithm to use when solving specific software engineering problems. Typically, software engineering practitioners select a *single* heuristic algorithm to solve all instances of a problem [6, 15, 111] that can result in solutions that do not fulfill requirements or use excessive computational resources [56]. To address this limitation, we leverage algorithm selection techniques for software engineering problems. More specifically, we demonstrate the benefits of using algorithm selection for web service composition and third-party software library migration at the method level. To the best of our knowledge, algorithm selection has not been studied before for either SBSE problem.

3.2 Web Services Composition

Numerous self-adaptive service composition algorithms and platforms have been proposed in the literature [4,7,10,17,19,20,38,63,81,82,83,125,133]. A substantial amount of work [7,10,18,19,36, 38,43,72,83,147] proposes new composition algorithms, either exhaustive or inexact. While inexact composition algorithms compute approximate solutions and are comparatively less expensive to use, they only provide near-optimal solution guarantees [36,72,147]. So, there may be some instances on which inexact algorithms do not converge or for which they use excessive computational resources. This leads to several undesirable, inefficient compositions.

Platforms such as those proposed in [17,34,37,63] identify individual problematic concrete services and replace them. Each approach is able to pin-point under-performing services and replace them individually using a pre-determined composition algorithm. This is done by using certain local constraints over every abstract service to replace under-performing services, which has the effect of reducing the search space for the composition algorithm used. Such approaches need carefully constructed constraints, or otherwise they lead to sub-optimal solutions. R-CASS is complementary to these platforms, because it can incorporate different graphs and constraints when profiling different composition algorithms. An algorithm is selected for a particular composition task based on its characteristics such as search space size, QoS constraints, node and utility distribution. By leveraging this information, we are able to obtain better results than a statically chosen algorithm used to solve all composition tasks. Thus, our work is fundamentally different from but complements current approaches by modeling different algorithm behaviors and selecting one for different composition tasks of varied search spaces.

Finally, the approach presented in [131] recommends algorithms for a set of batched user requests using only the most recent execution data. They use more resources for priority workflows and fewer for those that have not been executed frequently. Such an approach leads to inaccurate compositions for less frequently executed graphs. Moreover, they only consider one metaheuristic algorithm, which uses the same search mechanisms and demonstrates similar behavior across composition instances; thus they do not consider complementary performance of algorithms. Additionally, they do not leverage historic execution data that directly impacts which composition algorithms perform well for different instances of composition tasks. In our work, R-CASS is able to leverage historical execution data and model various algorithms' behaviors on various composition instances to obtain accurate predictions. Furthermore, our approach has the advantage of adapting efficiently at runtime in response to unanticipated changes in the composition environment.

Our work differs from pre-existing works in that R-CASS considers individual strengths of algorithms and selects a specific algorithm to solve a composition instance based on features such as graph size and search landscape properties. Furthermore, to avoid composition failure across tasks, R-CASS uses two strategies (1) By using a *set* of algorithms demonstrating complementary performance [75, 76], R-CASS is able to select other algorithms from the set that can compute a solution when one fails. (2) The set of algorithms under consideration include an exact algorithm that R-CASS selects when inexact ones are predicted to fail. Thus, in this work, we propose a novel adaptation mechanism to select one algorithm per composition task, rather than proposing a new composition algorithm. Our goal here is to build a self-optimizing system that can reconfigure itself to fulfill QoS requirements and minimize computational resource usage.

3.3 Third-Party Software Library Migration at the Method-Level

Evolutionary algorithms have been leveraged to address various challenges in software engineering [96] such as software development estimation [97], structure testing [33], workflow selection [91] etc. However, current library migration approaches do not evaluate search algorithms to recommend method mappings. In this section, we discuss the differences between our approach and related library migration techniques. We also assess various quality indicators proposed in the literature and describe the indicators used in this work.

3.3.1 Library-Level Approaches

Most library migration approaches [110, 129, 130] recommend which target library should be used to replace each source library, that is, they only recommend libraries and not methods. Recent work [66, 67] leverages multiple metrics to rank candidate target libraries, while other techniques [6, 62, 100] recommend specific types of libraries, *e.g.,* for Android, data science or python. Similarly, Nafi *et al.* [102] recommend libraries across 4 different programming languages using cosine similarities obtained using word vectors that are calculated from mined developer discussions. Other related work [134, 137, 140] recommends different versions of the same library. Nguyen *et al.* [105] use recurrent neural networks to update library versions and their dependencies, whereas Rubei *et al.* [121] create a migration graph to recommend which library version should be used for replacement. Our work can be used in conjunction with these approaches as we propose fine-grained recommendations by identifying replacement methods from a pre-specified target library.

3.3.2 Programming Language or Framework Specific Approaches

Numerous approaches [31, 71, 107, 112, 114] address library migration at the method-level, but can only be used for specific frameworks or languages. A recent work [109] proposes extensions to specific tools (refaster and ErrorProne) to automate library migration. Both tools are used in conjunction to analyze and refactor compilable Java code, and are not easily generalizable to all languages. Phan *et al.* [114] generate APIs in C# to replace Java methods using the Word2Vec model and create a transformation matrix that can be re-used to recommend API sequences. Similarly, Nguyen *et al.* [107] train an API2VEC model to recommend migrations from Java to C# by analyzing source and target language's usage contexts. However, such an approach does not consider differences in library designs that could lead to variations in surrounding contexts. Moreover, different developers can use different "helper" methods to achieve similar functionality, resulting in very different contexts for analogous methods. In this work, we propose an approach that can be used for any programming language or framework provided method and documentation similarities can be defined. Furthermore, we do not require historical data to identify method replacements, which is useful when recommending newer libraries for which sufficient data is unavailable.

3.3.3 Method-Level Library Migration Approaches

Existing method-level library migration approaches [61,103,124] leverage the "wisdom of the crowd" to identify suitable methods by mining API migration patterns from existing repositories or forums such as StackOverflow. Ramos *et al.* [117] and Nadi *et al.* [101] recommend API mappings for libraries used in data science applications by mining patterns of common migrations from GitHub. A related approach [89] models API usage using knowledge graphs and library documentation. Harman *et al.* [15] modify library source code to improve comprehension and maintainability. While this approach does not directly recommend methods during library migration, the authors use genetic improvement to modify the source code without impacting API definition.

Multiple works recommend methods during library migration [21, 40, 49, 150] using deep learning techniques that require extensive training data and computational power. Specifically, Collie *et al.* [49] use sophisticated program synthesis techniques on intermediate code representations to provide API recommendations. Similarly, Chen *et al.* [40] use unsupervised deep learning to embed usage semantics and infer likely mappings. These approaches only work for specific languages (compilable, Java) and require a lot of training data. Alrubaye *et al.* [12] propose a relatively more generalizable approach that uses a decision tree model to recommend one-to-one method mappings. However, such an approach cannot be used if there are newer mappings that are not present in the training dataset or libraries for which training data is unavailable [106]. To address these limitations, we leverage multi-objective search algorithms to identify suitable mappings. Our approach uses three different types of similarity schemes (method and documentation similarity, and co-occurrence probability) to recommend method mappings even when library data may not be available. Furthermore, we show that our approach can recommend different mapping cardinalities with high precision and recall. Additionally, our approach can be used for different programming languages and frameworks, as long as similarity scores can be defined.

3.3.4 Quality Indicators Assessment

Several quality indicators [8,86,87,88,108] have been proposed in the evolutionary search literature to evaluate the convergence, spread, uniformity and cardinality achieved by algorithms on different problems. We briefly summarize how each of these properties relates to the key characteristics of the API migration problem and the quality indicators we use in this study:

- In this work, our goal is to mitigate the time and effort required to evaluate numerous source-target method mappings during library migration. Since developers expend considerable effort verifying each recommended mapping, we want to minimize the number of recommended source-target method mappings without impacting precision and recall. As a result, it is not useful to evaluate the cardinality of generated solution sets for this problem.

- To achieve high precision and recall when recommending method mappings, it is desirable that algorithms converge to the best possible solutions after examining a wide range of mappings. Thus, we evaluate whether each evolutionary algorithm converges to recommend fewer but correct mappings.

- In this work, we evaluate the convergence property of each algorithm by comparing generated solutions with a "groundtruth" derived from the dataset used in our experiments. This groundtruth consists of a set of correct mappings that have been used before by other developers when migrating between software libraries. We measure how close the generated solutions are to the groundtruth using the mean Euclidean distance of all the solutions in the final iteration. Note that a similar formulation is used in IGD+ and GD+, except we measure distance from the groundtruth (pareto) point to the solution set rather than from the pareto front. Moreover, since the groundtruth is a single point of reference, we do not have a reference *s*et for comparison. As a result, indicators that measure the properties of a pareto front (specifically, uniformity and spread properties) are not well suited to this problem.

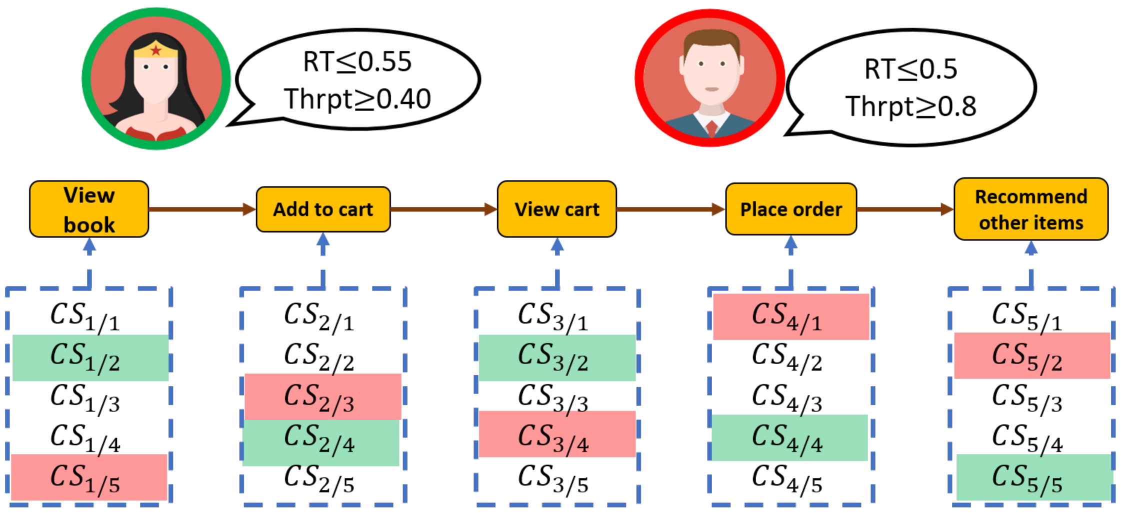
- In practice, developers do not know beforehand what the best method mappings are, that is, they do not have access to “groundtruth” mappings. As a result, indicators (such as Euclidean distance) that evaluate distance from the pareto front cannot be used. Moreover, as libraries evolve, developers may use different method mappings to achieve the same functionality, thereby changing the pareto fronts. However, to reduce time and effort required during the verification of mappings, we are interested in “knee points” that can balance the number of mappings while also being accurate. As a result, the hypervolume indicator is of interest.

Chapter 4

A Self-Adaptive Framework for Service-Oriented Systems Using Algorithm Selection

4.1 Motivation

Figure 4.1 illustrates the impact of variable algorithm behavior on service-oriented systems using the book buying software example introduced in Section 2. We observe two users, Alice and Bob, with specific response time (RT) and throughput ($Thrpt$) requirements, and examine three heuristic composition algorithms (H_a, H_b, H_c) to select specific candidate services for Alice (highlighted in green) and Bob (highlighted in red). Specifically, we demonstrate that leveraging *complementary performance* of H_a and H_b results in a significant reduction in time and memory resource usage while fulfilling user QoS requirements. We observe that between H_a and H_b, only H_b can fulfill Bob's QoS requirements. In contrast, both H_a and H_b select candidate services that fulfill Alice's QoS requirements. A suitable composition algorithm must be accurate and inexpensive. Service composition algorithms frequently re-select candidate services in response to QoS fluctuations, changes to the application DAG (*e.g.,* added features), and updated user requirements. Additionally, it is common for software applications to be deployed on Infrastructure as a Service (IaaS) instances, where computational resource usage directly impacts operational costs, and by extension, cost to users. For example, if composing an application requires several hours for completion, this results in significant system downtime and increased operational costs. As a result, H_a


RT≤0.55
Thrpt≥0.40
RT≤0.5
Thrpt≥0.8
View book
Add to cart
View cart
Place order
Recommend other items
$CS_{1/1}$ $CS_{1/2}$ $CS_{1/3}$ $CS_{1/4}$ $CS_{1/5}$
$CS_{2/1}$ $CS_{2/2}$ $CS_{2/3}$ $CS_{2/4}$ $CS_{2/5}$
$CS_{3/1}$ $CS_{3/2}$ $CS_{3/3}$ $CS_{3/4}$ $CS_{3/5}$
$CS_{4/1}$ $CS_{4/2}$ $CS_{4/3}$ $CS_{4/4}$ $CS_{4/5}$
$CS_{5/1}$ $CS_{5/2}$ $CS_{5/3}$ $CS_{5/4}$ $CS_{5/5}$

Algorithm	QoS 1 (RT)	QoS 2 (Thrpt)	Time (s)	Memory (kB)
H_a	0.51	0.60	25	120,000
H_b	0.40	0.95	32	150,000
H_c	?	?	?	?

Figure 4.1: This figure illustrates how different heuristic composition algorithms outperform each other on specific composition tasks. In this example, three algorithms H_a, H_b, H_c can be used to select candidate services to fulfill Alice and Bob's response time (RT) and throughput (Thrpt) requirements.

is better suited to fulfill Alice's QoS requirements because it uses relatively fewer time and memory resources compared to H_b. From So, selecting different algorithms for specific composition tasks results in fewer resources being used while also fulfilling QoS requirements.

Figure 4.2 provides an architectural overview of our approach. The *Composition Engine* is a central component of our approach that monitors every composed application, records existing service selections, detects QoS violations and selects an algorithm for composition. To accurately select one composition algorithm from a set of possible algorithms, we collect execution data from previously executed compositions and record delivered solution utility, time, and memory usage for m composition algorithms. This historical data is stored in the *Composition Execution Data Storage* and is used to train a *Composition Algorithm Selector* that models the performance of each algorithm at runtime (detailed further in the next section). In this Chapter, we first demonstrate the complementary performance of four popular composition algorithms and leverage *classifiers* to select one algorithm for each user.

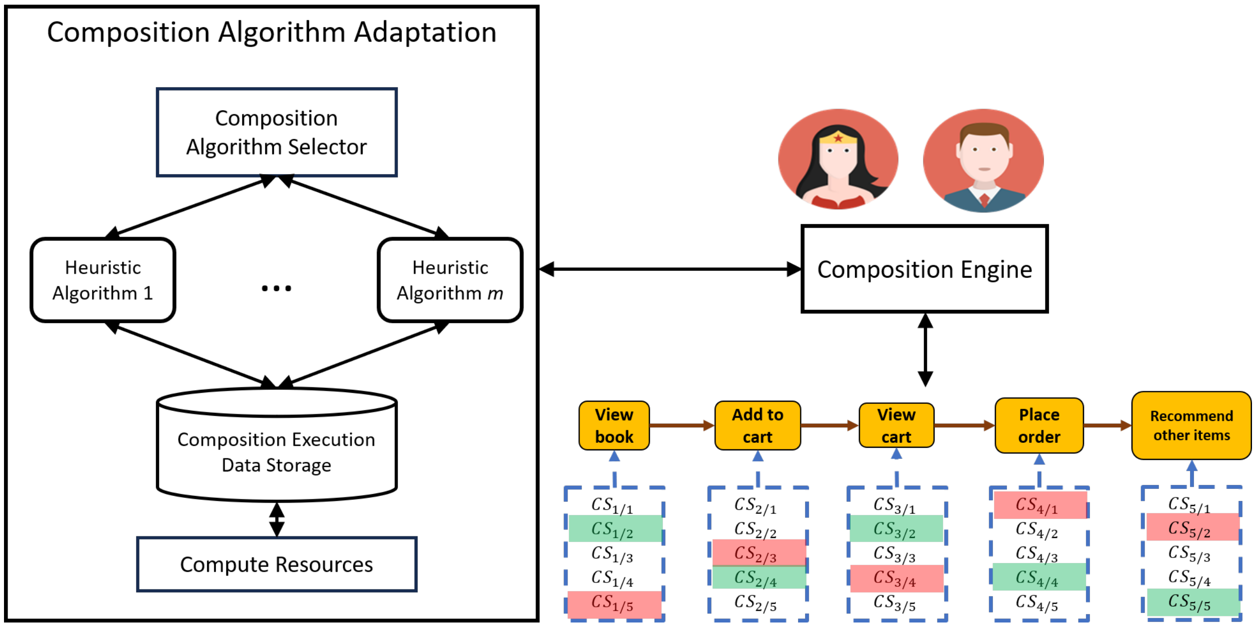

Composition Algorithm Adaptation
Composition Algorithm Selector
Heuristic Algorithm 1
...
Heuristic Algorithm m
Composition Execution Data Storage
Compute Resources
Composition Engine
View book
Add to cart
View cart
Place order
Recommend other items
$CS_{1/1}$ $CS_{1/2}$ $CS_{1/3}$ $CS_{1/4}$ $CS_{1/5}$
$CS_{2/1}$ $CS_{2/2}$ $CS_{2/3}$ $CS_{2/4}$ $CS_{2/5}$
$CS_{3/1}$ $CS_{3/2}$ $CS_{3/3}$ $CS_{3/4}$ $CS_{3/5}$
$CS_{4/1}$ $CS_{4/2}$ $CS_{4/3}$ $CS_{4/4}$ $CS_{4/5}$
$CS_{5/1}$ $CS_{5/2}$ $CS_{5/3}$ $CS_{5/4}$ $CS_{5/5}$

Figure 4.2: This figure presents an architectural overview of our approach. We select one of m heuristic algorithms to fulfill each user request based on their past performance.

When using pre-trained classifiers as *Composition Algorithm Selectors*, only one algorithm from a complementary set is selected for execution on the current user request. As a result, the performance of other unselected algorithms remains unobserved leading to an *incomplete data setting*. For example, in Figure 4.1, the performance of algorithm H_c is not observed because it is not selected for Alice or Bob. So, our training data does not contain information about H_c for these newer user requests. Consequently, classifiers become inaccurate over time as a result of *incomplete data*. In addition to evaluating classifiers as algorithm selectors, we assess contextual multi-armed bandits to *accurately* select algorithms in an *incomplete* data setting for *online* composition algorithm selection.

4.2 R-CASS Description

To facilitate algorithm selection in service-oriented systems, we propose R-CASS, a framework that leverages *composition algorithm selection* at runtime for self-adaptive service composition. Our goal is to predict a suitable algorithm for fulfilling each composition request, such that the selected algorithm fulfills QoS requirements while minimizing the time and memory resources used for composition. To achieve our goal, we propose a novel adaptation mechanism that uses a set of algorithms, each with its unique strengths, and selects one algorithm per composition task to meet QoS needs and optimize composition costs. In this section, we outline our proposed system, R-CASS.

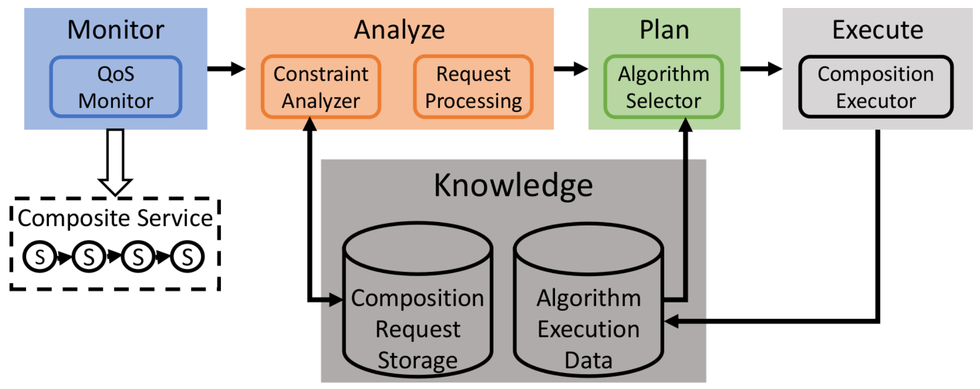


Figure 4.3: We provide a reference architecture for R-CASS to facilitate algorithm selection at runtime. Using the popular MAPE-K feedback loop, service compositions are monitored at runtime and data from recent executions is collected for accurate predictions.

4.2.1 R-CASS using the MAPE-K architecture

To deploy R-CASS in a service composition environment, we integrate different components of R-CASS into a feedback loop that can monitor applications and trigger adaptation at runtime. Figure 4.3 provides a reference architecture for implementing our approach using a feedback loop. In particular, we use the MAPE-K loop [70], a popular, influential reference architecture. Figure 4.3 shows how R-CASS is realized using a MAPE-K loop [70]. Every component of the R-CASS architecture is divided into one of four MAPE phases - **M**onitor, **A**nalyze, **P**lan and **E**xecute. All collected information about recent QoS violations, composition requests, and algorithm executions is stored in the **K**nowledge base. Such a design allows for a clear separation of concerns so that R-CASS can monitor composed applications for changes and adapt to them using the *Composition Algorithm Selector*. Once an algorithm is executed, data about its performance is also gathered and used to train the *Selector*. The integration of R-CASS with the phases of MAPE-K is as follows:

1. **Monitor**: In this phase, R-CASS uses the *QoS Monitor* to monitor and record the QoS of selected concrete services corresponding to each abstract service. If the aggregate QoS fails to meet business or user needs, R-CASS needs to adapt to (re-)compose a new solution by first selecting a composition algorithm and executing it to select new candidate services. R-CASS's **Monitor** phase records all QoS information at each timestep and passes it to the **Analyze** phase of the MAPE-K loop.

2. **Analyze**: In the **Analyze** phase, information from the *QoS Monitor* is used to decide if re-composition is needed. The *Constraint Analyzer* component takes as input the recorded

QoS information for the composed application and references it with the requested QoS parameters. If the requested solution quality is not met, it prepares a (re-)composition request. This composition request contains information about the dependency graph, QoS requirements $\vec{q}_{req}$, QoS weights $\vec{w}_{req}$, and QoS distribution features (described in Section 4.3) which is then passed to the **Plan** by the *Request Processing* component to select an algorithm. In this work, we re-compose by selecting concrete services for a DAG according to some user-provided constraints over the entire application. This ensures that a globally optimal solution is found. However, in certain business applications, nested DAGs or sub-workflows are commonly used [1,2], which means that an abstract task may need to be recomposed using its own dependency graph and constraints. In that case, R-CASS can be used to re-compose specific sub-workflows or DAGs as needed.

3. **Plan**: In the **Plan** phase, R-CASS *plans* for adaptation by selecting a composition algorithm once the **Analyze** phase triggers re-composition. The *Request Processing* component passes as input recorded features about the composition task and environment that are used to query the algorithm performance models. These models are trained to select an algorithm for each composition request using execution data from previous composition requests. In this work, we use classifiers to model composition algorithm performance and act as selectors. Ultimately, the **Plan** component uses predictions from the *Algorithm Selector* to solve the optimization problem presented in the next subsection and select a suitable composition algorithm.

4. **Execute**: The *Composition Executor* component uses the composition algorithm selected in the **Plan** phase and executes it. Execution data from the composition algorithm along with the composition request is stored in the *Algorithm Execution Data Storage* and *Composition Request Data Storage* components of the **Knowledge Base**. This data is used to train the *Algorithm Selector* for future executions.

4.3 Modeling Algorithm Performance

We begin by describing the features used to characterize a composition task/instance. Next, we describe how classifiers and contextual multi-armed bandits are trained using historical execution data to select algorithms at runtime. Finally, we describe the set of algorithms we use to demonstrate the positive effects of composition algorithm selection.

4.3.1 Features for Composition Algorithm Selection

We begin by characterizing service composition requests that serve as inputs to the classifiers. Each composition request is characterized using features that provide essential context to predict the most suitable composition algorithm. In addition to the base set of features [56], we expand our feature set to include information about node utility and distribution. All features are treated as continuous variables. Our set of basic features are as follows:

1. **Dependency graph characteristics**: The Directed Acyclic Graph (DAG) is a functional description of the application and is characterized by the number of abstract and concrete services as features. Existing works [10, 143] have demonstrated that general flow structures can be reduced to sequential flow graphs, therefore in our work, we focus on sequential flow graphs to test our approach.
2. **Requested QoS**: A user's requested solution quality is characterized using 2 QoS attributes, that are used to compute the requested solution utility. Weights for each QoS attribute are also provided to prioritize different QoS attributes using the L_p function. In this work, we weigh both QoS attributes equally so that both attributes are equally preferred for exposition. We note that our framework is flexible and allows a practitioner to customize the QoS attribute weights for their application.
3. **Computational resources**: Time (seconds) and memory (kilobytes) used by each algorithm are recorded, with the goal of minimizing resource usage.

To accurately predict computational resource usage and characterize the utility distribution in a DAG, we adapt additional node distribution features from [75]. These features characterize the search space an algorithm must explore and its complexity.

1. **Characterizing the adjacency matrix**: In a DAG, each concrete service for an abstract service is connected to the next abstract service's concrete services - thus creating an adjacency matrix. We characterize this matrix by measuring the mean values, variation coefficient, and skew between each adjacent pool of concrete services for each of the two QoS attributes.
2. **Centroids of the QoS attributes**: We calculate the centroids of both QoS attributes for a DAG's entire concrete service pool.
3. We also record the **bottleneck** between two nodes; this translates to the worst QoS values between two abstract nodes. These features, in addition to the adjacency matrix features,

provide insights about the search space in which the composition algorithms search for solutions [76].

4.4 Using classifiers as algorithm selectors

Classifiers are supervised learners that learn how to differentiate samples from multiple classes using labels assigned to each training datum. Using these labels, certain decision boundaries are learned on training data that can be used to similarly assign labels to unlabeled test data. In our work, we consider supervised learning because it incorporates human inputs in the form of labels and learns decision boundaries according to certain preferences. For R-CASS, we have 4 labels corresponding to one of 4 composition algorithms explained further in this section. To train classifiers, a previously executed service composition request is labeled with the algorithm that best solves it, as discussed in Section 4.4.1. The algorithm that can fulfill QoS requirements while using the least amount of time and memory, is the most suitable for a particular composition task.

4.4.1 Classification Optimization Problem

We describe the optimization problem that balances solution quality and computational resource usage when using R-CASS. We assume that there is a set of composition algorithm candidates that outperform each other on different composition instances, thus demonstrating complementary performance [75,76]. This set is denoted as $CASet = \{CASet_0, CASet_1, \ldots CASet_j\}$, wherein each composition algorithm has different solution guarantees and uses varied computational resources in terms of time $time_j$ and memory mem_j. Some algorithms may compose a solution more efficiently than others for a particular dependency graph and set of user QoS constraints. Our goal is to select an algorithm that fulfills QoS requirements while minimizing computational resource usage. Therefore, we model composition algorithm selection as a dynamic decision problem.

$$\begin{aligned}
&\text{argmin}_{j \epsilon CASet}\ \{\min(time_j), \min(mem_j)\}\\
&\textit{such that } U_j \geq \textit{ReqUtil}\\
&\textit{where } U_j = LpFunc(\vec{q}_j, \vec{w}_{req})\\
&\textit{and } \textit{ReqUtil} = LpFunc(\vec{q}_{req}, \vec{w}_{req})
\end{aligned} \tag{4.1}$$

This optimization problem is solved for each composition request req that is associated with an abstract service dependency graph or DAG for the requested functionality. The dependency graph consists of a set of abstract services $AS = \{AS_0, AS_1, \ldots AS_n\}$. Each abstract service AS_n is associated with a pool of concrete services of varying QoS values $CS_n = \{CS_{n0}, CS_{n1}, \ldots CS_{ny}\}$. We obtain the requested solution utility *ReqUtil* and composed utility delivered by a composition algorithm U_j using the L_p function described in Chapter 2. The best suited algorithm adaptation $j\epsilon CASet$ fulfills *ReqUtil* while using the least amount of $time_j$ and mem_j. We note here that fulfilling the QoS requirements is a *necessary* but not sufficient condition for selection.

4.5 Leveraging contextual multi-armed bandits for algorithm selection

To mitigate the risk of predicting inaccurate algorithms using pre-trained classifiers, we propose *online* composition algorithm selection and leverage incomplete data using a powerful reinforcement learning technique – contextual multi-armed bandits [53, 84]. Contextual bandits ① periodically execute different composition algorithms to collect additional *incomplete* execution data, and ② leverages this data to accurately model algorithm behavior on various tasks.

Contextual bandits learn to select the optimal *action* for the current *context* of the environment by balancing *exploration* and *exploitation* [84]. In this work, *actions* correspond to each service composition algorithm in the set $CASet$ and the *context* corresponds to the task for which an algorithm must be selected *e.g.,* select suitable candidate services to fulfill Bob's *RT* and *Thrpt* requirements of 0.5 seconds and 80%. When using contextual bandits approaches, balancing *exploration* and *exploitation* is crucial as it impacts the overall quality of current and future compositions. For example, a *purely exploitative* strategy can select Ex for both Alice and Bob leading to excessive computational resource usage. In contrast, *pure exploration* would randomly select algorithms that can lead to sub-optimal solutions *e.g.,* using H_b for Bob will lead to a QoS violation. In this work, we evaluate three different strategies to balance *exploration* and *exploitation* for online composition algorithm selection: greedy, ϵ-greedy, and upper confidence bound (UCB).

Algorithm 1 describes how the $\epsilon-$greedy strategy is applied to online composition algorithm selection using incomplete execution data. Each composition algorithm $CAlgo_j$ is associated with a regression model m_j that is trained to predict its performance using data from previous executions $T(CAlgo_j)$. Each model can be re-trained separately at runtime as an algorithm $CAlgo_j$ is executed more frequently, which allows us to leverage *incomplete* execution data leading to more accurate modeling and predictions over time. Contextual bandits strategies query these regression models and select a composition algorithm CA_{select} based on the predicted behavior of algorithms and the

Algorithm 1: Contextual bandits for algorithm selection using $\epsilon-$greedy

```
Input: Training data $T$, Exploration parameter $\varepsilon$, Set of composition algorithms $CASet$
Output: Selected composition algorithm, $CA_{select}$
$cost, m = \{\}, \{\}$
//Train a regression model for each algorithm
for each $CAlgo_j$ in $CASet$ do
    $T(CAlgo_j) = \{T_i$ for $i$ in $|T_j| \mid T_j \epsilon CAlgo_j\}$
    $m_j = m_j(T(CAlgo_j))$
end
//Predict composition algorithm performance
$\Psi = \Phi(CT_t)$
$\lambda = \text{argmin}_{j \epsilon CASet} m_j(\Psi)$ //Pure greedy selection
$\delta = \text{random}()$
if $\delta > \varepsilon$ then
    $CA_{select} = \lambda(CT_t)$
else
    $CA_{select} = random(CASet)$
end
return $CA_{select}$
```

exploration-exploitation strategy used. The current composition task CT_t is characterized by its QoS, node, and utility distribution features Ψ. These features are extracted using a feature-value mapping function Φ and used as an input to query each regression model m_j when selecting an algorithm for a composition task. The following contextual bandits strategies are evaluated in this work:

- *Greedy:* In Algorithm 1, the greedy approach [53] corresponds to always using $argmin_{j \epsilon CASet} m_j(\Psi)$, and setting the exploration parameter ϵ to 0. This leads to *greedy* selections that have the least predicted cost. However, a purely *exploitative* approach such as this can result in one algorithm being executed most of the time leading to inaccurate models of other composition algorithms and ultimately unsuitable algorithm selections on future tasks.
- *$\epsilon-$ Greedy:* The $\epsilon-$greedy strategy [60], shown in Algorithm 1 randomly selects a composition algorithm ϵ% of the time, and greedily selects an algorithm with the least cost $cost_j$ $(1-\epsilon)$% of the time. A larger ϵ value leads to more exploration and can result in more accurate modeling of all algorithms in $CASet$ as more execution data is collected.
- *Upper Confidence Bound (UCB):* The UCB strategy [84] constructs an upper bound on algorithm performance by adding a second term to a regression model's prediction. This additional term

incorporates the standard deviation σ_j of predicted $cost_j$ (Equation 4) and is multiplied by a discount factor γ_j and algorithm's execution *probability.* If a particular composition algorithm is selected frequently, its cost will increase resulting in other composition algorithms being explored. By including this term, UCB selects algorithms that (a) it is confident will compute an optimal solution, or (b) are under-explored.

Our approach can be deployed as a runtime adaptation mechanism for service-oriented systems using the influential MAPE-K [56] feedback loop architecture described in [56]. Contextual bandits can be used to select suitable composition algorithms and re-select candidate services for each task to fulfill QoS requirements while minimizing resource usage.

4.5.1 Bandits Optimization Problem

Our goal is to select a composition algorithm for each task such that it fulfills QoS requirements and minimizes time and memory usage. We formalize the optimization problem that contextual bandits in an *incomplete data* setting to select an algorithm $CASet_j$.

$$\text{argmin}_{j\epsilon|CASet|}\ cost_j(time_j, mem_j, U_j) \tag{4.2}$$

$$where\ CASet = \{CASet_0, CASet_1, \dots CASet_j\} \tag{4.3}$$

$$where\ cost(time_j, mem_j, U_j) = time_j + mem_j + penalty_j \tag{4.4}$$

$$where\ U_j = 1/L_p(\vec{q}_j, \vec{w}_{req}), ReqUtil = 1/L_p(\vec{q}_{req}, \vec{w}_{req}) \tag{4.5}$$

$$penalty_j = \begin{cases} 3 \text{ if } U_j < ReqUtil \\ 0 \text{ otherwise} \end{cases} \tag{4.6}$$

As shown in Equations 2–6, our approach optimizes the overall *cost* of using a composition algorithm for a specific task by taking into consideration delivered solution quality, time and memory resources used during composition. In this work, the cost of using a composition algorithm j is formulated as $cost_j$, and denotes a trade-off between $penalty_j$, composition $time_j$ and memory mem_j. Compositions that do not fulfill user QoS requirements $ReqUtil$ are strongly penalized with a penalty term of 3 that has a significantly larger value than the normalized values of $time_j$ and mem_j combined. A high penalty discourages sub-optimal solutions and leads to composition algorithm selections that fulfill QoS requirements and minimize computational resource usage. The formulation of this optimization problem is crucial because it determines how accurately contextual bandits approaches learn to model and select composition algorithms over time.

4.6 Evaluated Composition Algorithms

To illustrate the benefits of adapting composition algorithms at runtime, we implement and execute four popular service composition algorithms [72] on composition instances. We evaluate exhaustive and inexact composition algorithms and select one algorithm that fulfills QoS requirements while minimizing computational resource usage. We note that while we selected these algorithms for exposition, R-CASS can easily be extended to include additional composition algorithms. The algorithms we implement in this study are:

- **Multi-Constrained Shortest Path (MCSP)**: MCSP is an exhaustive algorithm proposed in [143]. It evaluates all possible candidate solutions and provides the highest utility solution. This approach is comparatively time and memory intensive, especially as the search space size increases.
- **Ant Colony Optimization (ACS)**: ACS is a meta-heuristic algorithm inspired by the behavior of ants originally proposed in [57, 147]. It uses multiple agents to find a solution by exploring graph edges with higher utility. The hyperparameter settings are: $\alpha = 2, \beta = 8, t_0 = 10$.
- **Genetic Algorithm (GA)**: GA is a search heuristic inspired by the theory of natural evolution. We modify the formulation in [36] and use the same hyperparameter settings.
- **Particle Swarm Optimization (PSO)**: Particle Swarm Optimization is a metaheuristic inspired by the flocking behavior of birds [74]. It computes *velocities* for possible solutions and evaluates solution fitness based on utility. We use the formulation presented in [48, 90].

We select these composition algorithms because they each behave differently, allowing us to examine why some algorithms are selected for certain composition tasks. We selected ACS, GA and PSO as they are the most widely-studied service composition algorithms [72] and MCSP as our exact baseline algorithm. By doing so, we leverage existing literature to establish ground truth and correctness of our initial results. Following existing guidance [36, 115, 122], we set 200 as the maximum number of iterations and as the population size for all metaheuristic algorithms. If a solution fulfilling user QoS constraints is found within these iterations, the algorithm is stopped. Alternatively, if a solution is not found, the algorithm is said to have not converged. Such a formulation is useful for a service composition tool because algorithms must finish running using finite resources to be feasible for an application.

4.7 Methodology

4.7.1 Dataset Generation

We set up experiments on 6,144 composition requests on DAGs of a wide range of sizes [45]. The number of abstract services vary from 5 to 40, and can be one of 5, 10, 15, 20, 25, 30, 35, 40 while the number of concrete services per abstract service is one of 5, 10, 15, 20. Using these values, we generate combinations of a different number of abstract and concrete services. This setup gives us a wide range of search space sizes [45] from a possible 3125 to $1.1e^{52}$ combinations, in which case exact optimization becomes expensive. Composition algorithms search through this space of possible solutions to generate one meeting user requirements. We use a real-world dataset, WS-DREAM [148] and randomly sample it for response time and throughput values of concrete services. We use three sets of constraints to signify different classes of users; with relative solution qualities of 0.85, 0.9 and 0.95. Solution qualities are relative to the best possible solution obtainable for a graph at a given time.

We vary the QoS for a pool of concrete services available for a graph at every time step, in two ways - (a) As data for the same set of concrete services is available at a certain time step, we observe changes in the QoS values for concrete services and recompose (b) If a concrete service fails or is unavailable at a given time step, it is replaced by sampling for another concrete service. In this manner, we evaluate R-CASS's performance in an online setting for 6,144 composition requests, spread across 64 time slices.

4.7.2 Comparative Approaches

With R-CASS, our goal is to select the right algorithm for each composition task such that it fulfills QoS needs while minimizing computational resource usage. So, we propose an efficient adaptation mechanism to leverage the strengths of different composition algorithms, rather than presenting a new composition algorithm. In our experiments, we validate the algorithm selection mechanism by comparing four different composition algorithms. An added advantage of our approach is that more algorithms can be included in practice as needed.

Because R-CASS is the first work of its kind to exploit variability in service composition algorithm performance, there are no other comparative works in existing literature. Furthermore, since our approach uses multiple composition algorithms each with their own strengths, it is not fair to compare any one pre-existing composition algorithm not included in the algorithm portfolio. So, we adopt the notion of single and virtual best solvers common in the algorithm selection literature from [75, 76] to provide upper and lower bounds on R-CASS's performance. They are as follows:

- **Single Best Solver (SBS)**: Our first baseline is the single best solver that shows the best performance for the majority of composition instances. Adapting composition algorithms using algorithm selectors is useful only if they outperform the single best solver. In this work, the single best solver is MCSP, because fulfilling QoS requirements is our first priority. This is because it is not useful to deliver sub-optimal solutions using fewer computational resources.
- **Virtual Best Solver (VBS)**: This is the perfect selector, which selects the best possible algorithm for each composition instance, rather than one for the entire dataset. It is the ideal result, and provides an upper bound for our algorithm selectors. Due to imperfect learning, the performance of selectors will fall short of the virtual best selector. In this study, the labels assigned for training supervised learning models are the best possible selections.

4.7.3 Evaluation metrics to assess R-CASS performance

We define the metrics used to evaluate R-CASS performance in this section. We use accuracy, precision, recall and the F1-score to determine the best classifier. Then, R-CASS uses the best classifier to evaluate how much time and memory is saved using algorithm adaptation.

1. **Accuracy:** Accuracy is simply the ratio of correct predictions to the total number of predictions.
2. **Precision:** Precision identifies which proportion of classified instances was correctly classified.
3. **Recall:** Recall identifies which proportion of actual positives was identified correctly.
4. **F1-Score:** F1-score gives a clearer picture of a classifiers ability to predict true positives by combining precision and recall. For imbalanced datasets, it is a more accurate metric of classifier performance.
5. **Cost:** We use cost when leveraging contextual multi-armed bandits. Cost is defined as the sum of differences between requested and delivered solution quality, time and memory resources used for composition. Each algorithm has a cost associated with it, and our goal is to select one with the least cost. Note that these differences are normalized since they have different units.

6. **Time and memory saved:** These are differences between time and memory used by the composition algorithm selected by our approach and the static SBS approach. Time is measured in seconds (s) and memory in kiloBytes (kB).

7. **Running overhead:** This is a measure of the time (s) taken to adapt by selecting a composition algorithm.

When reporting overall results we use the Wilcoxon test to measure statistical significance.

4.8 Experimental Evaluation

R-CASS selects one algorithm from a complementary *set*, such that it best meets user QoS requirements and minimizes computational resource usage in terms of time and memory. We achieve this goal by leveraging the unique strengths of multiple composition algorithms and select one for each composition task. To gauge R-CASS's ability to do so using classifiers and contextual multi-armed bandits, we formulate the following research questions:

- **RQ1**: What behavior do Multi-Constrained Shortest Path (MCSP), Ant Colony System (ACS), Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) exhibit when solving composition tasks in terms of delivered solution quality, time and memory resources used? *We demonstrate that MCSP, ACS, GA and PSO all solve different tasks better than each other in varied search spaces.*

- **RQ2**: Which classification and regression models can best model Multi-Constrained Shortest Path (MCSP), Ant Colony System (ACS), Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) performance on composition tasks? *We evaluate the ability of several classifiers and regressors to model composition algorithm performance. The best metrics are achieved by a random forest model.*

- **RQ3**: How effectively does a classifier-selected composition algorithm meet Quality of Service (QoS) requirements while minimizing computational resource usage as compared to a statically chosen composition algorithm? *Using a trained random forest model for classification, R-CASS uses 55.1% less time and 37.5% less memory for composition on average as compared to the SBS.*

- **RQ4**: Does online learning improve composition algorithm selection to reduce computational resource usage while meeting QoS constraints? If so, how do the three exploration strategies

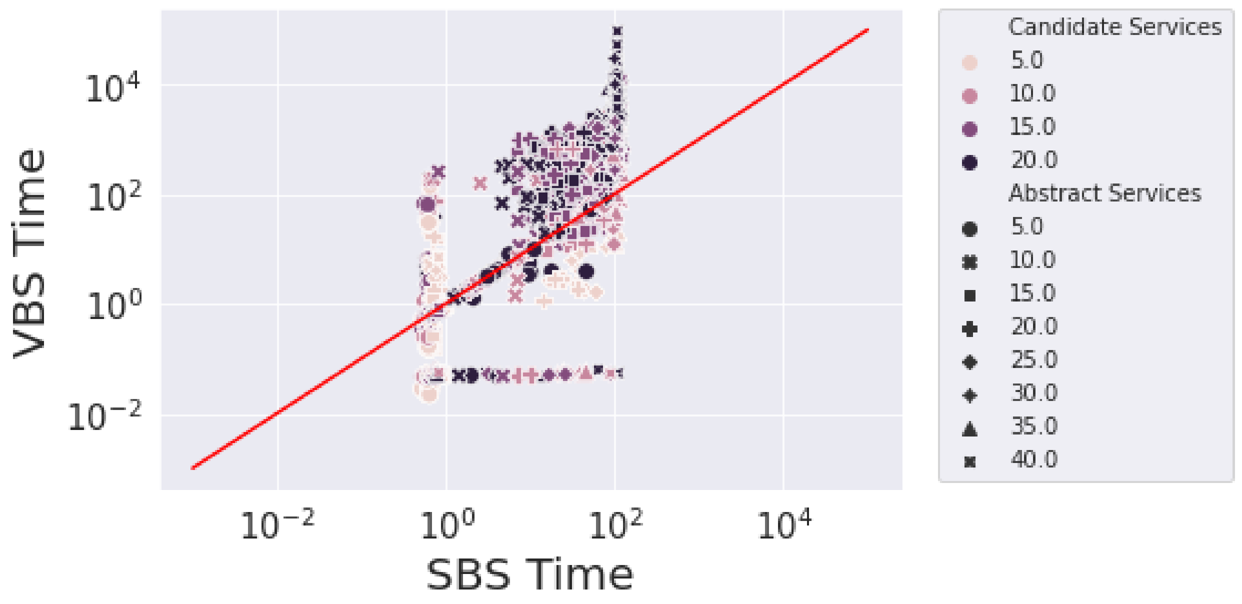


Figure 4.4: The gap between composition time using SBS and VBS selections is considerable. VBS delivers compositions faster with more points closer to the diagonal, than the SBS which is much slower.

impact composition algorithm selection? *We find that re-training regression models on incomplete data leads to diverse composition algorithm selections that use fewer computational resources. Additionally, we demonstrate the benefits of balancing exploration and exploitation to select the right composition algorithm for each task.*

- **RQ5**: What is the overhead associated with selecting an algorithm for each composition task at runtime? *We demonstrate that R-CASS has minimal overhead and is suitable for runtime use.*

4.8.1 RQ 1: Establishing Complementary Algorithm Performance

We pose this question to analyze how each composition algorithm's search efficiency changes with respect to different composition task features (presented in Section 4.3) and determine the potential for algorithm selection. To answer this research question, we first execute all algorithms on all composition requests and collect execution data. Next, for every composition instance, we identify which algorithms generated a solution meeting user constraints. Finally, from this subset, we determine which algorithm used the least amount of time and/or memory to generate such a solution. Every composition instance is labeled with the best algorithm and these labels are used to train classifiers. These individual labels are deemed the best possible selections for our study and

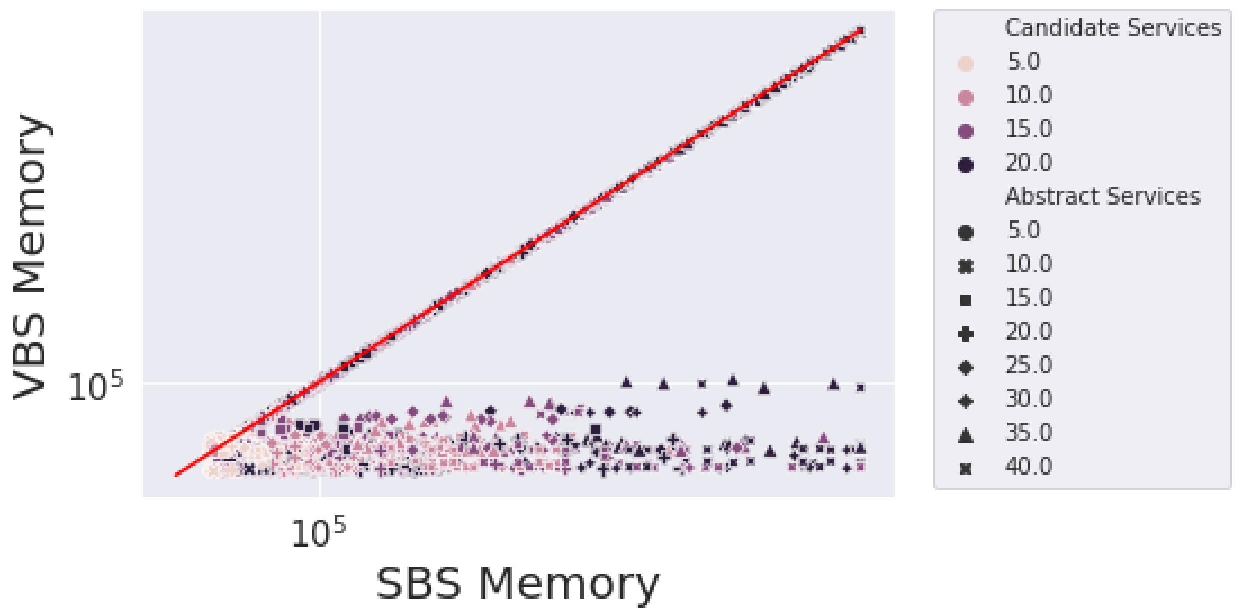


Figure 4.5: The gap between SBS and VBS memory usage demonstrates the potential for saving memory. VBS uses less memory than SBS for approximately 30% of composition instances.

thus constitute the virtual best solver (VBS). The distribution of algorithms is given in Table 4.1. As seen from Table 4.1, ACS is able to find solutions faster than MCSP, GA and PSO for 894 instances, while MCSP, GA and PSO are competitive with each other. There are approximately 1300 instances where only MCSP meets user constraints, as ACS, GA and PSO did not converge. To avoid selections that do not meet QoS requirements, MCSP is selected as the SBS.

Table 4.1: Algorithm label distribution for 6,144 composition instances. Each instance is labeled with the algorithm that fulfills constraints while minimizing time and memory usage.

Algorithm	**Labels**	**Algorithm**	**Labels**
MCSP	1805	GA	2057
ACS	894	PSO	1388

Figures 4.4 and 4.5 demonstrate the potential for algorithm selection to minimize excess computational resource usage. In these figures, the axes are visualized on a log-scale. We observe that the VBS takes less time for most composition instances and has points on the diagonal as compared to the baseline, which is desirable. The SBS demonstrates a larger spread of the resource distribution, demonstrating the potential for algorithm selection to bridge this performance gap. For a more detailed evaluation, we perform feature selection using Logistic Regression with lasso regularization and study the coefficients for selected features to analyze algorithm behavior. Out of 407 features,

280 are selected. To draw our conclusions, we use the mean, median and standard deviation of these feature values for composition instances labeled MCSP, ACS, GA and PSO. We observe

- The median number of candidate services for MCSP, ACS, GA and PSO selections were 10, 15, 10 and 10 respectively. All three algorithms were selected across graphs of varying abstract services. Interestingly, MCSP is selected for a large number of graphs with a higher number of abstract and candidate services because the others did not converge.
- Mean response times for each abstract service were highest for instances solved by MCSP followed by PSO and GA. Median values for these features was also highest for ACS in general. The standard deviation was highest for GA and PSO values for graphs that have ≥ 15 abstract services.
- For MCSP instances, mean and standard deviation values of variation coefficients of response times were highest. Skew values for response times across abstract services were all negative with a higher standard deviation observed for MCSP, GA, PSO and ACS instances, in that order.
- These findings indicate that MCSP is selected for instances with larger graphs with difficult search spaces, as characterized by the worst response time and throughput values. ACS is selected for smaller (≤ 10 nodes) graphs with relatively complex search spaces. Finally, GA and PSO are selected for similar instances, with GA selected for larger graphs. These have a higher mean throughput and lower mean response time, leading to a quicker search.

4.8.2 RQ 2: Prediction Accuracy using Classification and Regression Techniques

Assessing classifier performance:

Having demonstrated the potential for improvement using composition algorithm adaptation, we evaluate classifiers as algorithm selectors to adapt composition algorithms dynamically. We consider Multi-Layer Perceptron (MLP), Random Forests, Decision Trees, SVM with rbf and sigmoid kernels, Logistic Regression, Naive Bayes, k-Nearest Neighbors (kNN) and Quadratic Discriminant Analysis (QDA). We select these because each of these learns a different type of decision boundary for classification [65], allowing us to test different ways of learning. To improve classification performance, we also include features characterizing the search space, explained earlier in ***RQ1***. These features are used because they can be computed fast at runtime. The dataset was split 70/30

Table 4.2: By comparing classifier accuracies, we demonstrate MLP's ability in selecting composition algorithms to minimize computational resource usage.

Classifier	**Accuracy**	**F1-score**	**Precision**	**Recall**
Random Forest	**0.7446**	**0.7441**	**0.7436**	**0.7446**
SVM - rbf	0.7185	0.7185	0.7179	0.7185
MLP	0.6963	0.6965	0.6995	0.6963
Decision Tree	0.6518	0.6537	0.6559	0.6518
kNN	0.6328	0.6328	0.6367	0.6328
Logistic Regression	0.6269	0.6190	0.6173	0.6269
SVM Sigmoid	0.6382	0.6367	0.6379	0.6382
Naive Bayes	0.4175	0.3661	0.4572	0.4175
QDA	0.5580	0.5531	0.6036	0.5580

and 5 fold cross-validation was repeated 10 times to report the metrics shown in Table 4.2. From Table 4.2 we see that random forest has the best performance across accuracy, F1-score, precision and recall as compared to the others. As seen from Figure 4.6, the random forest classifier can correctly identify most algorithm instances, shown in the high accuracy across the diagonal.

Assessing regressor performance:

For our experiments using contextual multi-armed bandits, we leverage regression models to predict the cost of using each algorithm as described in Section 4.5.1. Exploration strategies leverage the predicted cost of using each algorithm to select a specific composition algorithm for each task. For example, if the normalized cost of using MCSP is 2 and GA is 2.7 for a specific composition task, a greedy strategy would select MCSP.

Our experiments demonstrate that Random Forest (RF) outperforms other regression techniques when modeling composition algorithm performance. We evaluate 6 different regression modeling techniques to predict the cost of using each algorithm on different tasks: Support Vector Machines (SVM) with the rbf and sigmoid kernels, Gaussian Processes (GP) with rbf kernel, Multi-Layered Perceptron (MLP), Lasso Regression (LASSO) and Random Forests (RF). Each regression model is trained to predict the $cost_j$ of every composition algorithm (MCSP, ACS, GA and PSO) based on the features of the current composition task, Ψ. We use a 70/30 train-test split and repeat 5 fold cross validation 10 times to report average results in Tables 4.3 and 4.4. We also use mutual information regression to eliminate features with importances < 0.1, and retain 271 features. To evaluate regression model performance, we use R^2, Mean Absolute Error (MnAE) and Median

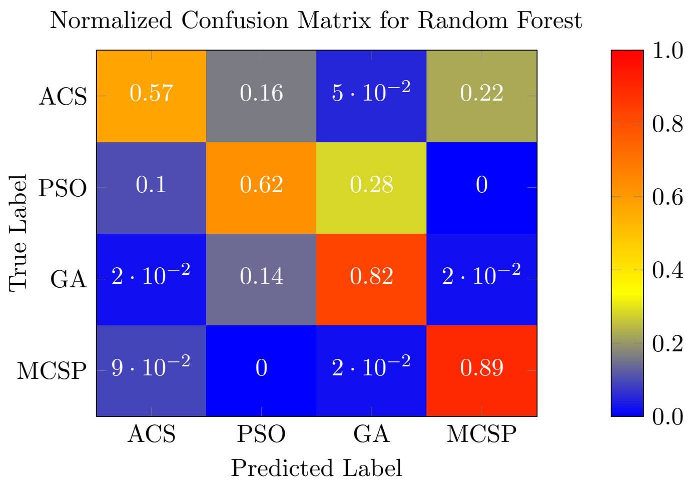


Figure 4.6: Confusion matrix for Random Forest. High classification accuracy is observed across the diagonal, which is necessary to (a) prevent utility loss and (b) minimize computational resource usage.

Table 4.3: Average R^2, MnAE and MdAE performance of each regressor on all 4 composition algorithms. We find that GP and RF demonstrate the best performance.

Model	R^2	**MnAE**	**MdAE**
SVM-RBF	0.75	0.27	0.18
SVM-SIG	0.52	0.32	0.22
GP	0.78	0.24	0.19
MLP	0.7	0.33	0.22
LASSO	0.57	0.37	0.27
RF	0.78	0.24	0.16

Absolute Error (MdAE) to select the technique with the best performance. The R^2 score measures what proportion of variance of the *cost* can be explained using the feature set, and must be maximized whereas MnAE and MdAE must be minimized.

From Table 4.3, we observe that RF and GP have the best R^2 and MnAE scores. Moreover, from Table 4.4 we observe that RF has lower error when predicting PSO cost compared to GP. Specifically, we observe that RF has 18% lower MnAE and 32.3% lower MdAE than GP. As contextual bandits selects the composition algorithm with the least predicted cost, it requires accurate predictions for all composition algorithms. So, we select RF as our regression model.

Table 4.4: Gaussian Processes (GP) and Random Forest (RF) performance for each of four composition algorithm. RF demonstrates the best performance when predicting PSO cost.

CAlgo	**GP** R^2	**GP MnAE**	**GP MdAE**	**RF** R^2	**RF MnAE**	**RF MdAE**
MCSP	0.87	0.11	0.01	0.82	0.13	0.02
GA	0.86	0.27	0.23	0.86	0.28	0.24
ACS	0.87	0.21	0.16	0.78	0.22	0.16
PSO	0.53	0.39	0.34	0.65	0.32	0.23
AVG	0.78	0.24	0.18	0.78	0.24	0.16

Table 4.5: Comparing SBS, R-CASS and VBS performance distribution in terms of utility, time and memory used.

Method	**Mean Utility**	**Mean Time (s)**	**Mean Memory (kB)**	**Median Utility**	**Median Time (s)**	**Median Memory (kB)**
VBS	9.36	616.83	115325.69	1.87	0.78	69329.6
R-CASS	8.39	350.89	91635.89	1.90	2.09	69654.0
SBS	7.80	780.67	146625.04	2.43	44.96	90902.0

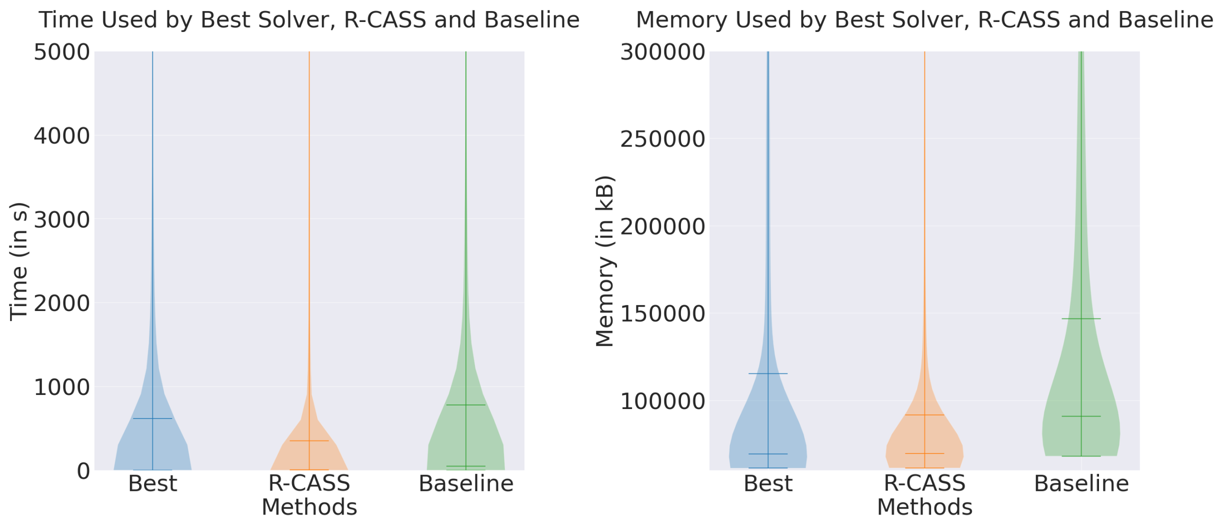


Figure 4.7: Time and memory taken by composition algorithms selected by the VBS, R-CASS and SBS. R-CASS selections use less time and memory than the SBS (Comparisons of R-CASS, VBS versus the SBS are statistically significant ($p < 0.05$))

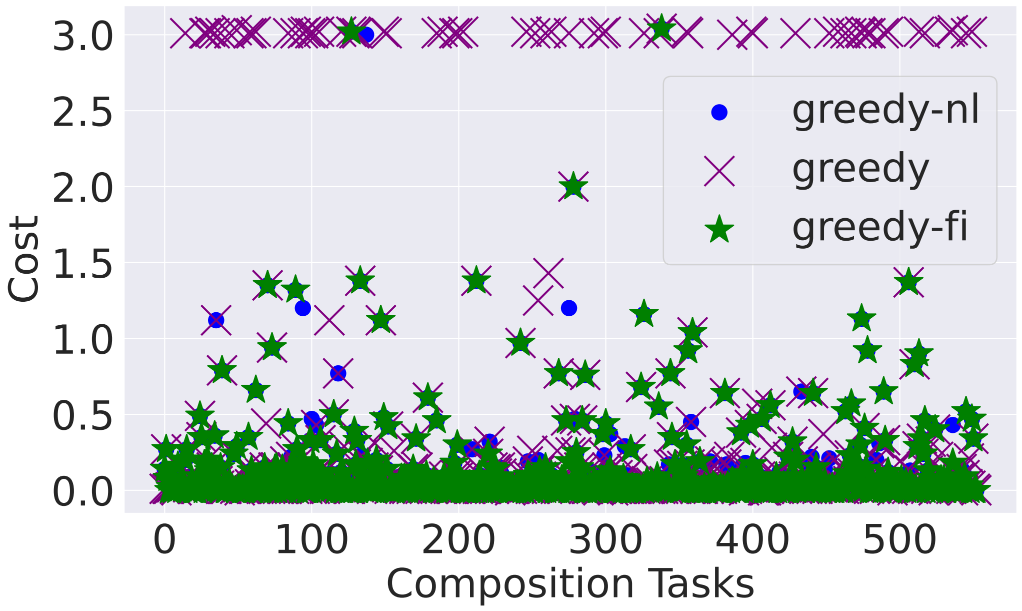


Figure 4.8: Each point shows the cost incurred on the verification dataset by greedy-nl, greedy and greedy-fi. In general, we observe that greedy selections outperform greedy-nl.

4.8.3 RQ 3: Evaluating Classifiers for Self-Adaptive Service-Oriented Systems

We use the best classifier as the *Algorithm Selector* in R-CASS and compare its performance with that of a statically selected algorithm denoted by the SBS. Figure 4.7 and Table 4.5 demonstrate that the time taken and memory used by an R-CASS selected algorithm is much less than that used by the SBS. On average, R-CASS takes 429.78 seconds (55%) less and 54989.15 kiloBytes (37.5%) less to generate solutions meeting constraints as compared to the SBS. The median time saved is 42.87 seconds and median memory saved is 21248 kiloBytes when compared to the SBS. Thus R-CASS is demonstrably more efficient than a statically chosen composition algorithm because of its algorithm adaptation mechanism.

4.8.4 RQ 4: Evaluating Multi-Armed Bandits for Online Service Composition Algorithm Selection

We demonstrate that leveraging incomplete data using online learning helps to improve regression model accuracy. Specifically, we evaluate three versions of the greedy strategy: ① greedy approach without retraining on online composition algorithm data (greedy-nl), ② a greedy algorithm that is trained using all algorithms' execution data leading to a full information setting (greedy-fi), and ③ a greedy approach that is periodically retrained using execution data for the selected composition algorithm only *i.e.,* using *incomplete* data. Furthermore, we evaluate these strategies by using a

10-80-10% dataset split as train, online and verification datasets. This corresponds to an extreme setting where there is very little training data [53]. Figure 4.8 demonstrates the cost incurred by each approach and we report the median time and memory for each greedy version. Greedy, greedy-nl and greedy-fi selections use 34.79, 40.5 and 7.04 seconds for composition respectively. Similarly, greedy, greedy-nl and greedy-fi selections use 78,632 kB, 83,932 kB and 71,408 kB respectively. Greedy-nl favors MCSP, as there is rarely a penalty associated with using it, leading to high time and memory consumption. Similarly, greedy tends to favor MCSP, but also learns to explore ACS, GA and PSO. In general, we observe that retraining regression models to *learn* from incomplete algorithm execution data leads to improved selections that (a) select diverse algorithms, and (b) use fewer computational resources.

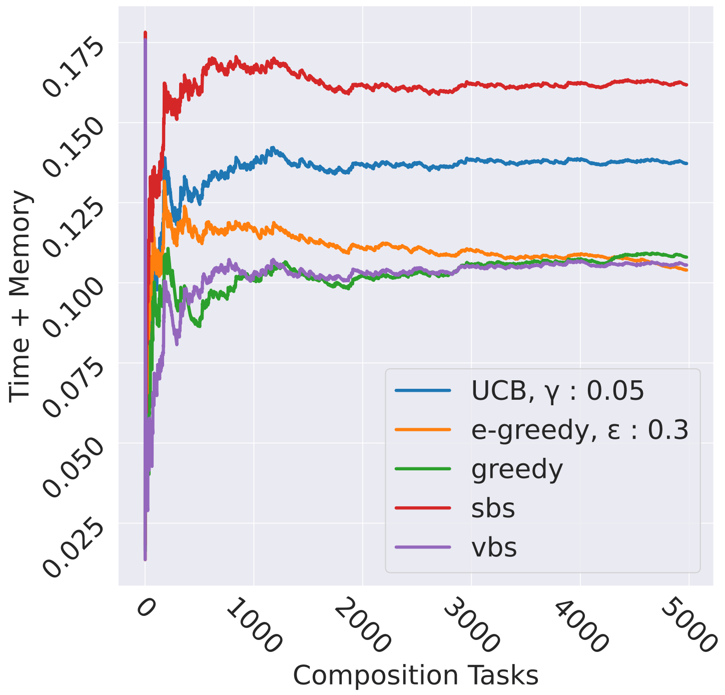


Figure 4.9: Greedy and $\epsilon-$greedy learn to select suitable composition algorithms over time. Each curve represents the normalized sum of time and memory for different contextual bandits exploration strategies on the online dataset.

Figure 4.9 and Table 4.5 demonstrate the positive impact of using explicit exploration to reducing composition time and memory usage. Each regression model m_j is initiated on the training dataset and re-trained for every 10 new datapoints observed during the online exploration phase, to leverage incomplete data. After the online phase, each model is greedily tested without exploration on the verification dataset to evaluate the impact of exploration. Figure 4.9 displays the cumulative sum of time and memory usage as a function of the number of composition tasks handled, obtained by selecting different composition algorithms. We carefully selected $\epsilon = 0.3$ and $\gamma = 0.05$ to reduce the number of incorrect selections. From Table 4.5, $\epsilon-$greedy demonstrates the highest time and memory resource savings, using 54.2% fewer time and 15.5% fewer memory resources compared to the SBS, with 10.3% incorrect selections not meeting QoS requirements. From Figure 4.9, we observe that $\epsilon-$greedy is closest to the VBS and outperforms greedy after $\sim$4200 composition tasks have been handled. We also observe that while UCB is relatively better than the SBS, UCB often selects MCSP because its performance is associated with a low standard deviation as it does not incur a penalty. Because MCSP exhaustively searches for and computes a solution with the highest utility, the mean time (MnT) and memory (MnM) is higher for UCB. Thus, we conclude that using explicit exploration (ϵ-greedy) leads to considerable time and memory savings.

4.8.5 RQ 5: Overhead

Classifier overhead:

Finally, we examine the time and data overhead imposed by the R-CASS decision making mechanism used to select algorithms at runtime. At runtime, R-CASS selects a composition algorithm for each task. As seen from Figure 4.10, the time required for selection is of the order of 10^{-3} seconds, which is negligible (1%) especially when compared to the actual time taken for composition. The second overhead pertains to the classifier's training data requirement. In R-CASS, training data is used to predict and model algorithm behavior, which is a challenging task. The availability of execution data enables more accurate and automatic selection, leading to significant time and memory savings. Additionally, this data overhead can be reduced by leveraging readily available execution data from existing systems to train the classifier. Thus, we conclude that R-CASS is suitable for runtime deployment due to its low overhead.

Contextual bandits overhead:

Our results demonstrate that there are small time and data overheads associated with using our approach for online composition algorithm selection. The mean time overhead associated with predicting composition algorithm performance is 0.07% when compared to executing composition algorithms. Additionally, the mean time overhead for periodically retraining regression models on online datapoints is 2.28% of the mean execution time. We note that retraining only occurs every 10 datapoints, so this overhead applies only at those timesteps. Furthermore, our experiments demonstrate the effectiveness of our approach in an extreme environment where insufficient training data [53] is available.

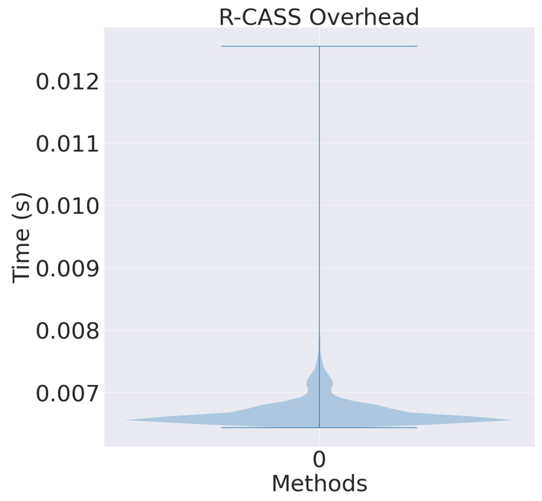


Figure 4.10: Time taken by R-CASS to select a composition algorithm. The y-axis scale demonstrates that a negligible amount of time is required to select an algorithm.

4.9 Limitations

- In this study, we compute 408 features relating to DAG characteristics and QoS distribution for each candidate service pool. However, these features can be expensive to calculate at runtime resulting in an increase in composition time and memory. We can address this challenge by considering feature-free techniques such as convolutional neural networks (CNNs), graph neural networks (GNNs) *etc.* to learn accurate DAG representations. These techniques can reduce prediction and composition time, but will require significant training data.

- When leveraging classifiers, we utilize a considerable portion of our dataset (70%) for training. Similarly, when using contextual multi-armed bandits, we use 10% to train our regression models and 80% data to explore different composition algorithms. Each of these techniques assumes that (a) a considerable amount training data is available and, (b) the training dataset is representative of composition tasks encountered at runtime. Due to QoS fluctuations, changes in application DAGs and user requirements, it is difficult to anticipate how composition tasks vary at runtime. As a result, the training data becomes unrepresentative of the tasks encountered at runtime and classifiers in particular can become inaccurate. Additionally, random exploration can result in the selection of unsuitable composition algorithms being selected which negatively impacts solution quality and excessive computational resource usage. To address this limitation, we examine the use of transfer learning techniques to reduce the training data requirement and accurately select composition algorithms for specific service-oriented systems tasks.

- Compared to classifiers, our contextual multi-armed bandits approach selects composition algorithms that use 17.5% more memory. Additionally, compared to the VBS, there is considerable room for improvement in terms of time and memory resource savings. In the future, we plan to explore other bandits strategies in addition to more powerful modeling techniques to further improve composition algorithm selection accuracy.
- In this work, we considered four popular service composition algorithms that assume that the candidate service pool does not change during composition, that is, candidate service pools are static. However, service QoS attributes vary frequently during a composition algorithm's execution, that is, candidate service QoS changes while a composition algorithm executes resulting in unsuitable candidate service selections. In the future, we plan to construct a set of composition algorithms that consider changes to candidate service pools during service composition.

4.10 Chapter Summary

In this chapter, we demonstrated the benefits of our proposed algorithm adaptation mechanism for service-oriented systems that leverages algorithm selection in a self-adaptive framework R-CASS for runtime compositions. Our work is fundamentally different from but complements existing service composition approaches, in that we select one composition algorithm from a portfolio of complementary algorithms, rather than proposing another composition algorithm. In particular, we leverage classifiers and contextual multi-armed bandits to select a different algorithm from a *complementary set* for each composition task.

Our evaluations demonstrate that when using a classifier as our selector, we reduce time usage by 55.1% and memory usage by 37.5% to deliver solutions fulfilling QoS requirements when compared to a static approach that uses a single composition algorithm for all composition requests. Moreover, when using contextual bandits for online composition algorithm selection, our approach reduces composition time and memory usage by 54.2% and 15.5% while fulfilling QoS requirements. In particular, our bandits approach requires less training data and takes advantage of online learning to leverage *incomplete* data generated at runtime.

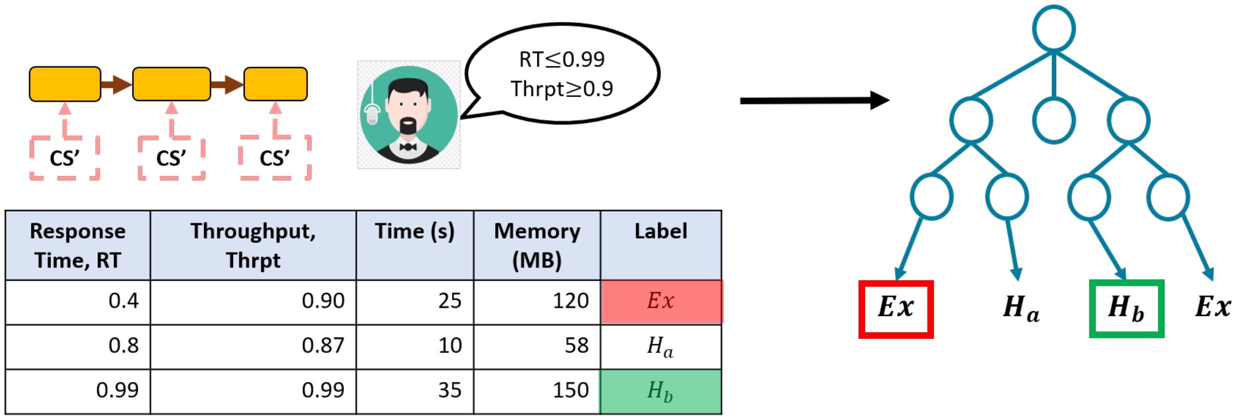
Chapter 5

Leveraging Transfer Learning for Algorithm Selection in Service-Oriented Systems

5.1 Motivation

In the previous chapter, we demonstrated the benefits of using classifiers and contextual multi-armed bandits to select heuristic service composition algorithms for SOA instances. We created a dataset of 6,144 service-oriented instances and leveraged it for training. During deployment, our training dataset corresponds to execution data collected from algorithms executed on previously seen instances. So, classifiers can be re-trained at runtime as more training instances become available. However, when deploying a pre-trained model to select algorithms at runtime, we can only select and execute one algorithm at a time. As a result, we have *incomplete* data where information about only one algorithm is available. Since this can negatively impact algorithm selection accuracy, we leverage contextual multi-armed bandits to periodically explore the use of other algorithms so that we can collect diverse execution data and more accurately model algorithm performance at runtime. In our experiments, we observed that the ϵ-greedy exploration strategy performs well and uses a random forest regression model to predict algorithm performance.

Figure 5.1 depicts additional challenges associated with using pre-trained models to predict algorithm performance. In this figure, a new user, Charlie, has strict response time and throughput requirements on a completely new composition task with different candidate services. We compute a set of features to characterize the composition task/instance and query our pre-trained random forest model, which selects the exact algorithm Ex based on the training data it has seen before. In contrast, when using random exploration in a contextual bandits strategy, the algorithm H_a may be selected to fulfill the user's requirements. However, in this scenario, only H_b can fulfill Charlie's requirements. In this example, it is difficult to anticipate changes to the composition task DAG, candidate service QoS and user requirements. In particular, the composition tasks at runtime may be very different from the training dataset resulting in incorrect algorithm selections. As a result, our pre-trained models may become inaccurate leading to solutions that do not fulfill user requirements or use excessive computational resources. So, the right algorithm H_b may not be selected as our final algorithm resulting in Charlie's QoS requirements not being fulfilled.



Response Time, RT	Throughput, Thrpt	Time (s)	Memory (MB)	Label
0.4	0.90	25	120	Ex
0.8	0.87	10	58	H_a
0.99	0.99	35	150	H_b

Figure 5.1: We illustrate the challenges associated with using classifiers trained on SOA instances. When selecting an algorithm for a new user Charlie with strict response time and throughput requirements, the random forest classifier would prefer the exact algorithm Ex based on the instances it has seen before. Once selected, we would have execution data only about Ex, when the right algorithm to select would have been H_b. When using contextual bandits, exploration may result in a different algorithm being chosen but result in inaccurate models.

To address the above-mentioned limitations, we leverage graph transfer learning to transfer knowledge about algorithm performance across clustered traveling salesman problem (CTSP) and service-oriented systems (SOA) instances. Figure 5.2 depicts how service-oriented systems tasks are commonly formulated as CTSP instances [10, 116, 147]. Each 'cluster' in a CTSP instance corresponds to an abstract service that must be visited by selecting one 'city' or 'candidate' service. In our CTSP and SOA tasks, the goal is to maximize the L_p utility and fulfill certain quality constraints. In this study, we use the GraphCL approach [142] to transfer knowledge of algorithm performance from CTSP to SOA instances. We execute MCSP, GA, PSO and ACS on widely available CTSP instances and leverage deep graph neural networks (DGNNs) to learn robust representations. Additionally, by using DGNNs to characterize graph instances, we reduce the time required to perform expensive feature computation routines used in the previous chapter. Crucially, recent studies [94, 95] have demonstrated that heuristic algorithm behavior is impacted by the choice of the encoding function used to characterize solutions. That is, heuristic algorithm performance changes based on the encoding function used. Our hypothesis is: *assuming the fitness function, problem encoding, and search algorithms used for CTSP and SOA instances are the same, we can transfer knowledge about algorithm performance from CTSP to SOA*. To achieve our goal, we adopt a popular transfer learning technique, GraphCL [142], to more accurately recommend heuristic algorithms across CTSP and SOA instances.

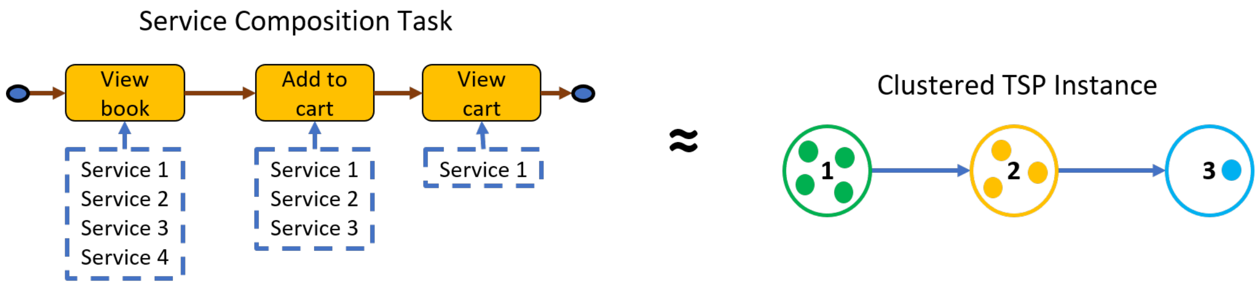


Figure 5.2: We highlight the potential similarities between service-oriented systems tasks and clustered traveling salesman problem (CTSP) instances. Service-oriented DAGs are commonly formulated as TSP instances where one candidate service must be selected for each abstract service.

5.1.1 Implications of Our Approach

Our goal is to improve algorithm selection accuracy on SOA instances by modeling heuristic algorithm performance on CTSP instances and 'transferring' knowledge of their performance. To achieve this goal, we leverage deep graph neural networks (DGNNs or GNNs) that learn robust graph representations during training and reduce the computational resources required to compute a feature set that characterizes CTSP/SOA instances. Additionally, since we cannot anticipate how SOA instances evolve at runtime, we reduce the need to collect expensive training data using SOA instances by leveraging commonly available CTSP instances. Furthermore, we evaluate GraphCL performance on a novel application and examine if we can accurately model algorithm performance by creating similar fitness landscapes. As a result, our work aims to train a generalizable algorithm selector for service-oriented systems.

5.2 Methodology

5.2.1 Applying GraphCL

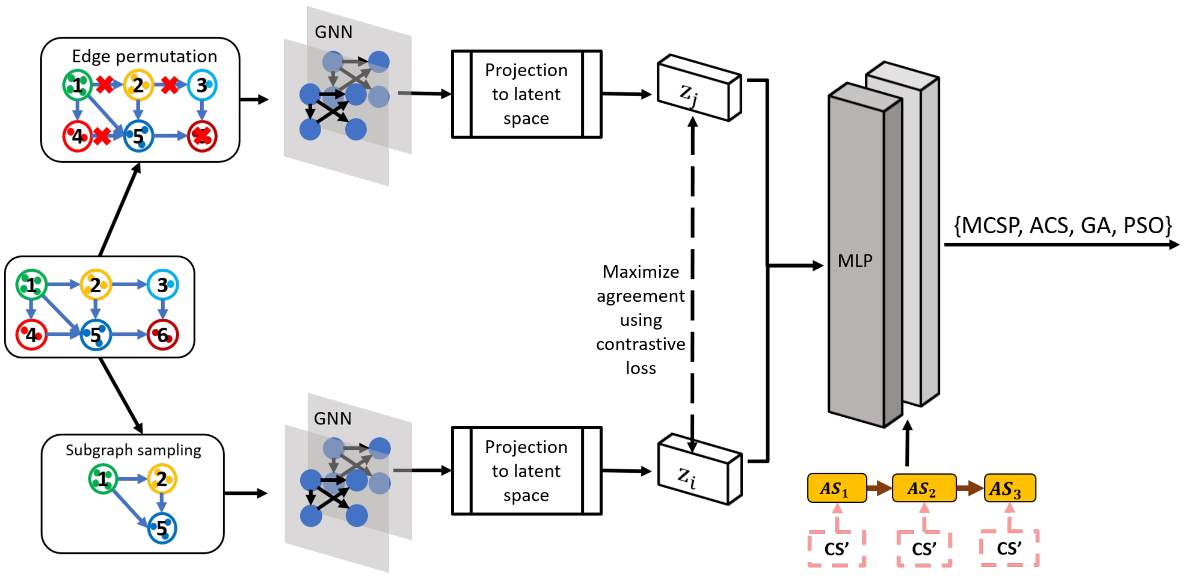


Figure 5.3: We adopt the GraphCL approach to transfer knowledge of algorithm behavior across CTSP and SOA instances. Specifically, we use the graph isomorphism network (GIN) to learn graph representations and use contrastive loss to reconcile augmentations with the original graphs.

Figure 5.3 depicts our overall approach that leverages deep graph neural networks (GNNs), graph augmentations, and contrastive loss. Specifically, we adopt the GraphCL approach proposed by You *et al.* [142]. For each clustered traveling salesman problem (CTSP) instance in our dataset, we generate one of three possible augmentations. We then pass the original and augmented graph pair through a GNN model and obtain a numerical representation or embedding for each graph. This representation is then passed through a *projection head* that reduces the dimensions of the embedding, that is, it *projects* the representations of both graphs to a common *latent* space. We then apply contrastive loss to make the augmented view and original graph agree by maximizing temperature-scaled cross-entropy loss between different graph representations. The full derivation can be found in [142]. Once the pre-training routine is completed, we introduce a linear layer to predict one of four classes corresponding to MCSP, GA, ACS and PSO. Finally, we finetune the entire network using SOA instances using cross-entropy loss.

In this work, we leverage the graph isomorphism network (GIN) model to learn graph representations. The GIN model [138] has been shown to outperform other architectures on multiple prediction tasks, including unsupervised transfer learning [142]. Specifically, GIN aggregates feature information from each node's neighbors to better model the relationships between multiple sets of nodes in a graph resulting in more accurate predictions. We use a GIN model with 5 layers, 300 hidden units in each layer and global maximum pooling. Figure 5.4 illustrates the augmentations we test in our study. By augmenting TSP instances, we introduce perturbations or noise in TSP instances. Our goal is to learn more robust representations of graphs in scenarios where complete information about a graph may be unavailable such as missing edges or node attributes. The augmented views are made to agree with the original TSP instances using contrastive loss. Additionally, our goal is to introduce 'small' perturbations that do not negatively impact the meaning of the graph *e.g.,* dropping 50% of the total cities in a graph consisting of 1000 cities results in a completely new TSP instance and alters graph connectivity and the underlying search space. We generate augmented views of approximately 20% of the TSP instances used during training. These augmentations are as follows:

- **Subgraph augmentation:** The intuition behind subgraph augmentations is that a local graph substructure contains the semantics of the entire graph [142]. To generate a subgraph, a random walk is performed by randomly selecting one 'city' from each 'cluster'. The number of clusters for which a city is selected is also random, that is, the path length is randomly determined and smaller than the path of the overall TSP instance.
- **Edge dropping:** A certain portion of the edges in the graph are dropped or added. This makes the learned graphs robust to changes in connectivity and is analogous to instances where certain web services are degraded or removed from consideration due to QoS fluctuations. These edges are dropped using a uniform IID distribution [142].
- **Node masking:** In this augmentation, attributes of nodes are hidden or masked. The goal is to prompt the GNN model to recover masked attributes based on other context information, *i.e.,* attributes that are not hidden. This corresponds to the case where QoS attribute estimates may not be available or accurate for specific web services but their performance in a composition can be inferred based on other neighbors. For example, the response time value for a specific candidate service can be estimated based on the L_p utility connecting two web services.

5.2.2 Dataset generation, baselines, metrics, and training splits

To evaluate the benefits of using graph contrastive learning for cross-domain algorithm selection, we generate a dataset of 27,267 clustered traveling salesperson problem (CTSP) tasks. We generate

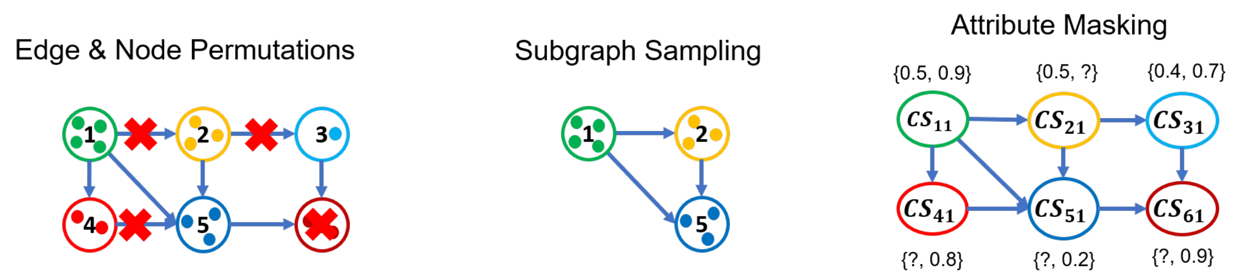


Figure 5.4: We evaluate the benefits of using different types of augmentations on our TSP dataset.

CTSP instances that contain {5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 75, 100} 'cities' per cluster, and {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500} total 'cities' per instance. Each 'city' has two attributes that are generated using a random uniform Euclidean distance (RUE) or Netgen scheme in [75]. We use the Netgen library[1] in R to generate each CTSP instance. A clustering algorithm is applied to group cities that are 'close' to each other by comparing their Euclidean distances $\sqrt{x^2 + y^2}$. These parameters result in a wide range of search spaces ranging from $5e^{20}$ to $4e^{117}$ possible combinations of visiting one 'city' in each cluster. Additionally, we study the impact of applying subgraph augmentation, edge permutations and attribute masking for learning robust graph representations. These augmentations can reduce the search space size considerably, so we aim to improve generalizability by using comparatively larger CTSP graphs on which to assess our approach.

We execute Multi-Constrained Shortest Path (MCSP), Genetic Algorithm (GA), Ant Colony System (ACS) and Particle Swarm Optimization (PSO) on each of these 27,267 CTSP instances to evaluate the variability of algorithm behavior. Each CTSP instance is labeled following the routine presented in chapter 4, where an algorithm that fulfills quality requirements while using the least amount of time and memory resource usage is considered the final label. Approximately 10% of CTSP instances from our dataset are invalid because a majority of 'cities' were assigned to a single cluster that could be solved relatively easily using a deterministic algorithm. As a result, these instances are not meaningful when studying our approach. Our final CTSP dataset consists of 24,681 CTSP instances. From Table 5.1, we observe that 12,016 instances are best solved by PSO, and the number of instances solved by MCSP and GA are relatively similar. We note that 4,853 instances are only solvable by MCSP, meaning that only MCSP can fulfill solution quality constraints. Although ACS fulfills solution quality requirements, it is selected for very few instances. This is because the ACS algorithm computes a large 'pheromone' matrix using relatively more computational resources resulting in it being selected for fewer instances.

[1]https://rdrr.io/cran/netgen/

Table 5.1: Algorithm label distribution for our dataset of 24,681 CTSP instances. Each instance is labeled with the algorithm that fulfills quality constraints while minimizing time and memory usage.

Algorithm	**Labels**	**Algorithm**	**Labels**
MCSP	7447 (4862)	GA	7668
ACS	136	PSO	12016

From the label distribution in Table 5.1, we observe that MCSP, GA, PSO and ACS demonstrate complementary behavior on our CTSP instances. So, we use our dataset of 24,681 instances and adopt the protocol based on using contrastive learning as specified in [142]. In other terms, we assess the impact of using unsupervised contrastive loss for learning graph representations, so the results reported in this study do not use the labels associated with CTSP instances during training. However, the labeling procedure helps us establish the scope for algorithm selection and determine whether our algorithm portfolio exhibits complementary performance on our CTSP instances. We plan to investigate other supervised approaches in the future.

Once our GNN model has been trained on our CTSP instances, we use our dataset of 6,144 service-oriented systems instances for finetuning. Specifically, we add a linear layer that learns to predict algorithm labels for our SOA instances using `BCE Logits` loss. We use 60% of our SOA dataset for finetuning and 40% to evaluate the impact of using transfer learning, that is, we report accuracy, precision and recall on 40% of our SOA dataset after finetuning.

5.3 Results

We evaluate the efficacy of our transfer learning-based approach using the following research question:

- **RQ6:** How effectively can we transfer knowledge about Multi-Constrained Shortest Path (MCSP), Ant Colony System (ACS), Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) performance from TSP to service-oriented systems instances?

Table 5.2 depicts the precision, recall and accuracy achieved using four GraphCL models. We observe that using augmentations improves accuracy, that is, subgraph, edge permutations and node masking improve accuracy by 14%, 14% and 4% respectively compared to using no augmentations in the 'None' model. Due to computational resource limitations, each model is trained using a

Table 5.2: We compare the accuracies achieved using different augmentations.

Augmentation	**Accuracy**	**Precision**	**Recall**
Subgraph	0.37	0.37	0.37
Edge Permutation	0.37	0.37	0.37
Node masking	0.27	0.27	0.27
None	0.23	0.23	0.23

subset of 5000 CTSP instances, with the learning rate set to 0.001. We train each model over 10 epochs and finetune the model over another 10 epochs. Notably, we ran each model for 100 epochs but observed that loss did not increase considerably over more iterations, that is, it stabilizes over 10 epochs. Each GNN model has 5 layers, and each layer has 300 units. We use maximum pooling and a dense layer as our projection head. We calculate the contrastive loss on the output of the dense layer and backpropagate it to train the DGNN. Furthermore, we set the augmentation ratio to 20% for each model, that is, augmentation is applied to 20% of the dataset. We note that the accuracy numbers are relatively poor, *e.g.,* our classifiers achieved 74% accuracy. This could be because the network architecture is too big compared to our dataset of sequential graphs. We did not observe significant changes in performance when learning rates were set to 0.001 and 0.01.

5.4 Future Work

In this work, we present our preliminary results when applying transfer learning for cross-domain heuristic algorithm prediction. We observe that adding in the subgraph and edge permutation augmentations improved prediction accuracy by 14% compared to using no augmentations. However, we note that the overall accuracy numbers are low, and outline future work to improve accuracy:

- In the future, we will dedicate significant effort to tuning our hyperparameters. In particular, we will test different learning rates, and weight decay and dropout values during training using CTSP instances. Additionally, we will also experiment with different augmentation ratios to evaluate if selection performance can be further improved.
- In the future, we will use more CTSP instances to train our GNN models. These instances are much larger than the SOA instances studied in this work. As a result, higher augmentation ratios may be more useful in improving selection accuracy. However, this computation is time consuming *e.g.,* applying subgraph augmentation with a 20% probability for our entire dataset would require approximately 9 days with 1 GPU and 8 CPU cores/workers. We will also investigate techniques to optimize augmentation generation and reduce training time.

- In this work, we adopt the architecture proposed in [142] that is used to predict the properties of large biochemical molecules. Consequently, this architecture may be too large for our CTSP and SOA instances which are limited to a few thousand 'cities' in clustered graphs. In the future, we will perform an architecture search to explore the use of different architectures with different numbers of layers, hidden units etc.

5.5 Chapter Summary

In this chapter, we demonstrated the impact of using deep graph neural networks (DGNNs) to model graph representations and reduce computationally expensive feature computational routines. We adopt a popular graph transfer learning technique, GraphCL [142], and transfer knowledge about heuristic algorithm performance from clustered traveling salesman problem (CTSP) instances to service-oriented systems tasks. We create a dataset of 24,681 CTSP instances and assess the scope for algorithm selection by executing MCSP, GA, PSO and ACS on each instance. Using the GraphCL approach, we train our DGNN using unsupervised contrastive loss and finetune our models on service-oriented systems instances. Our preliminary evaluations demonstrate that using subgraph augmentations, edge permutations and node masking improves algorithm selection accuracy by 14%, 14% and 4% compared to using no augmentations. We will continue to work on refining our approach in the future by performing architecture search, hyperparameter tuning and testing different augmentation rates.

Chapter 6

Search-Based Third-Party Library Migration at the Method-Level

In our work on self-adaptive service-oriented systems, we demonstrated that selecting one algorithm from a complementary set for specific service composition tasks results in significantly reduced time and memory resource usage while fulfilling solution quality requirements. *Given the success of algorithm selection for service composition, would other software engineering problems benefit from metaheuristic algorithm selection?*

In Chapters 6 and 7, we study the benefits of metaheuristic algorithms for third-party software library migration (API migration). Third-party software libraries are routinely used to reduce implementation effort, reduce the number of bugs in code, and avoid redundancy [28] during development. Over time, these software libraries are constantly updated to provide new features, functionality, and critical updates such as bug fixes [54]. As a result, developers routinely replace older, deprecated libraries with newer, updated libraries that are consistently maintained and updated. More specifically, developers replace each application programming interface (API) call that exposes a specific library function or procedure. This process of removing older (source) libraries' APIs and dependencies from source code and replacing them with APIs or methods from newer (target) libraries is known as API migration [129, 130].

In contrast to service composition, API migration emphasizes functional correctness. That is, service composition must fulfill non-functional QoS requirements whereas API migration must ensure that recommended APIs are functionally correct and do not negatively impact the underlying software functionality. So, our goal is to propose fewer but correct API mappings to reduce the burden on developers to manually verify mappings. Additionally, in Chapter 4, we began by demonstrating that service composition algorithms exhibit complementary behavior. While our approach was fundamentally different from current service composition work, it is founded on existing studies that have evaluated several metaheuristic algorithms for service composition. In contrast, metaheuristic algorithms have not been studied before for API migration. As a result, API migration provides a unique opportunity to study metaheuristic algorithm performance on a novel problem with unique requirements. So, in Chapter 6 we first evaluate various metaheuristic algorithms for API migration, and assess the benefits of algorithm selection in Chapter 7.

6.1 Introduction

We discussed how API migration is a challenging and complex task in Chapter 2 as a result of the large number of potential source-target API mappings to be evaluated. Additionally, other factors such as mapping cardinalities, differences in method names, return types, input parameters, and API documentation (if it exists) pose key challenges to search-based API migration. Current library migration approaches [12, 13, 41, 49, 68, 121, 136] rely on training data derived from previous migrations for specific programming languages and frameworks, or are designed to address one-to-one method mappings, so they cannot accurately recommend *one-to-many, many-to-one* and *many-to-many* mappings. As a result, these approaches cannot be used for different mapping cardinalities (*e.g.,* many-to-many mappings), newer libraries, or migrations between different languages [80].

Table 6.1: The 9 popular migration rules used in this study.

Source → Target
commons-logging → slf4j
slf4j → log4j
easymock → mockito
google-collect → guava
gson → jackson
testng → junit
json → gson
commons-lang → slf4j
json-simple → gson

In this chapter, we present our work on leveraging single and multi-objective metaheuristic algorithms for API migration. In particular, our GA approach recommends method mappings by constraining the number of recommended methods and maximizing method mapping similarity scores [55]. However, our single-objective approach achieves low precision and recall for many-to-many method mappings because it cannot accurately optimize the number of recommended target methods. To address this limitation, we leverage multi-objective optimization [42, 46, 58, 69, 73, 93, 98, 127, 144] to accurately recommend different mapping cardinalities during library migration. Our multi-objective approach explicitly minimizes the number of recommended method mappings and simultaneously maximizes the similarity of recommended methods from the replacement or *target* library. By formulating method-level migration as a multi-objective optimization problem, we achieve high precision and recall values while evaluating numerous combinations of target library methods to replace one or more methods from the original *source* library.

In contrast to existing techniques that require extensive training routines [12, 31, 107, 114] or specific mapping cardinalities [12], our approach can accurately recommend method mappings for newer software libraries, different programming languages and mapping cardinalities. We demonstrate the benefits of our approach using a single-objective genetic algorithm (GA) and 7 popular multi-objective evolutionary algorithms, namely, Unified Non-Dominated Sorting Algorithm (UNSGAIII) [113], Reference-Point Based Non-Dominated Sorting Algorithm (RNSGAII) [52], Adaptive Geometry Estimation Based Multi-Objective Evolutionary Algorithm (AGEMOEA) [113], S-Metric Selection Evolutionary Multiobjective Optimization Algorithm (SMSEMOA) [24], Non-Dominated Sorting Genetic Algorithm (NSGAII) [51], Indicator-Based Evolutionary Algorithm (IBEA) [151], and Multi-Objective Evolutionary Algorithm using Decomposition (MOEAD) [146]. In particular, our multi-objective approach explicitly balances two objectives: maximizing the fitness score and minimizing the number of all selected source-target method mappings. We evaluate each algorithm on a popular dataset consisting of 57,447 migrations from 9 Java library pairs or *migration rules* . Additionally, we examine the effectiveness of using three different similarity scores when evaluating source-target method mappings: Co-Occurrence score (CO), Method and Documentation Similarity (MS+DS), and a combination of all three (ALL). Our experimental evaluations demonstrate that UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD, and GA achieve an average precision of 90%, 89%, 94%, 90%, 91%, 94%, 71% and 61%, respectively, and average recall values of 83%, 23%, 58%, 63%, 58%, 60%, 17% and 16% respectively. Additionally, in the interest of reproducibility, we make all our code and results publicly available at: http://bit.ly/MOO-api-migration.

6.2 Methodology

We formulate API migration as a combinatorial optimization problem, specifically, as a *knapsack problem* [50]. Each migration rule is modeled as a knapsack containing all possible source-target method mappings, where each method mapping is an *item* with a specific *profit* and *weight*. An item's *profit* corresponds to its fitness score, and each item's *weight* is set to 1. Additionally, each knapsack is associated with a *capacity* constraint that dictates the *maximum* number of *items* that can be selected for a specific knapsack. During library migration, we want to recommend fewer, but accurate source-target method mappings so that developers can incorporate replacements into the source code. However, developers do not know beforehand which target methods should be used for replacement or how many will be required. These constraints inform the optimization objectives when using single and multi-objective metaheuristic algorithms.

6.2.1 Leveraging Single-Objective Optimization

When leveraging a single-objective algorithm, we set the *capacity* value to be equal to the number of source methods that need to be replaced. In this scenario, the capacity parameter is critical because it controls the number of target methods that can be recommended for selection, thus impacting performance. Without a constraint on the number of recommended methods, a knapsack's profit will be maximized by setting capacity to a high value, resulting in more target method recommendations; however, the time spent by developers to examine all recommended mappings increases considerably as the number of recommended target methods increases. Thus, to ensure that accurate source-target mappings are recommended while minimizing the number of recommendations, a penalty is imposed on solutions that exceed the capacity constraint by recommending too many target methods. If the number of recommended methods exceeds capacity, a penalty equal to the difference between the number of recommended methods and knapsack capacity is applied.

6.2.2 Using Multi-Objective Optimization

In contrast to the single-objective setting, where we impost a *capacity constraint*, we explicitly minimize the number of selected method mappings in our multi-objective formulation. In other terms, the maximum capacity value in the multi-objective setting is equal to all possible items in the knapsack. Our goal is to maximize the cumulative *fitness* of selected mappings (knapsack profit), while minimizing the *number* of selected mappings (knapsack weight). Accurately selecting an appropriate number of method mappings is necessary to reduce the effort expended by developers in manually verifying each recommendation and identifying newer mappings when unsuitable methods are suggested.

A fundamental requirement when using multi-objective optimization algorithms is that at least one optimization objective must conflict with at least one other objective, that is, achieving good results in one objective will result in poor values for another objective. In API migration, we consider the number of selected method mappings and the cumulative fitness score of selected mappings as our conflicting objectives. Our fitness score formulation results in positive values in the $[0, 1]$ range resulting in larger cumulative fitness scores as more method mappings are selected. So, the maximum cumulative fitness is achieved when all method mappings in a migration rule are selected. However, it is undesirable to recommend all mappings, as developers have to manually validate each selection resulting in a time-consuming and error-prone library migration process [13,49]. To achieve our goal of recommending the exact number of correct method mappings, each multi-objective algorithm must minimize the number of source-target method mappings while also maximizing the cumulative fitness score of all selected mappings.

6.2.3 Solution Representation

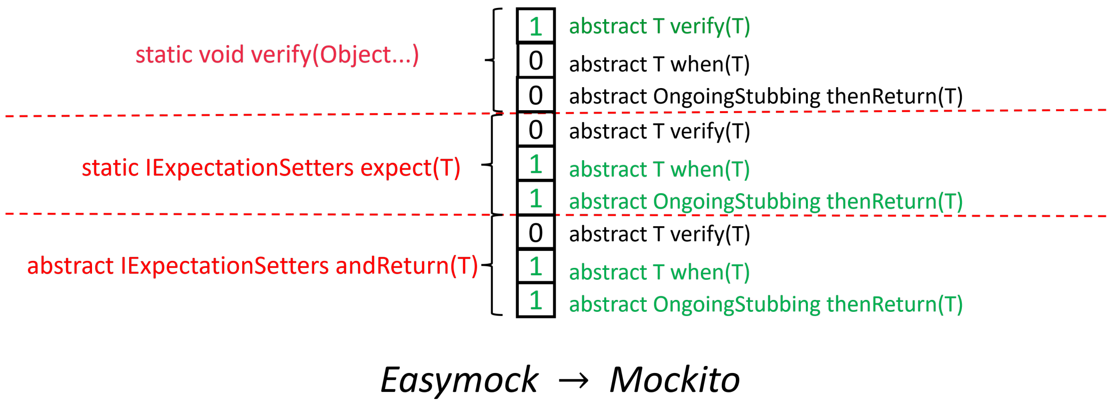

static void verify(Object...)
static IExpectationSetters expect(T)
abstract IExpectationSetters andReturn(T)
1 abstract T verify(T)
0 abstract T when(T)
0 abstract OngoingStubbing thenReturn(T)
0 abstract T verify(T)
1 abstract T when(T)
1 abstract OngoingStubbing thenReturn(T)
0 abstract T verify(T)
1 abstract T when(T)
1 abstract OngoingStubbing thenReturn(T)
Easymock → Mockito

Figure 6.1: In this study, source-target method mappings are represented using a chromosome bit string. Each $m = 3$ source method (on the left, in red) can be mapped to n target methods (on the right, in green). The first source method from *easymock*, *static void verify(Object...)*, is a one-to-one mapping and is replaced by *abstract T verify()* from *mockito*. A selected method mapping's corresponding gene value is represented as 1 in the bitstring. The second and third source methods are mapped to multiple target library methods *i.e.,* they are many-to-many mappings. Both *static IExpectationSetters expect(T)* and *abstract IExpectationSetters andReturn(T)* belong to the same code block and are replaced by *public abstract when(T)* and *public abstract OngoingStubbing thenReturn(T)*. During the search, these source methods should be individually mapped to both target methods.

We encode potential source-target method mappings for each migration rule as a chromosome containing individual genes. A chromosome represents all possible combinations of m source and n target methods, where each individual gene corresponds to a specific source-target method mapping. Figure 6.1 depicts a simplified example of migration between the *easymock* and *mockito* libraries, where each of $m = 3$ source methods (left, red) can be mapped to $n = 3$ target methods (green, right). In reality, the *easymock* and *mockito* libraries contain 2,211 and 2,265 methods respectively, so the number of potential method mappings is considerably higher, especially when considering *one-to-many* or *many-to-many* method mappings. Figure 6.1 depicts a chromosome consisting of $m \times n = 9$ genes, each corresponding to a potential source-target method mapping. If a specific source-target method mapping is selected, its corresponding gene value is set to 1, with the others set to 0.

- **One-to-One Mapping:** In Figure 6.1, the source-target method pair *static void verify(Object...)* $\rightarrow$ *abstract T verify(T)* is a one-to-one mapping, where one source method is replaced using one target method. When selected by an algorithm, the corresponding gene (at index 0) is marked as 1. Other potential mappings *static void verify(Object...)* $\rightarrow$ *abstract T when(T)*, and *static void verify(Object...)* $\rightarrow$ *abstract Ongoing Stubbing thenReturn(T)* are not selected. So, the genes in indices 1 and 2 representing these mappings are marked 0.
- **Many-to-Many Mapping**: We consider a many-to-many method mapping, where two source methods *static IExpectationSetters expect(T)* and *abstract IExpectationSetters andReturn(T)* need to be replaced by multiple target methods. In this example, the *static IExpectationSetters expect(T)* method is replaced using the *abstract T when(T)* and *abstract OngoingStubbing thenReturn(T)* methods. So, indices 4 and 5 representing these mappings in the chromosome bitstring are assigned a value of 1. Similarly, *abstract IExpectationSetters andReturn(T)* is replaced by *public abstract T when(T)* and *abstract OngoingStubbing thenReturn(T)*, so gene indices 7 and 8 are also assigned a value of 1. In this manner, each gene represents a unique source-target method mapping that can be selected by a multi-objective algorithm. During the search process, all gene values are concatenated and represented as a 0/1 bitstring for computational efficiency.

6.2.4 Fitness function

In this work, we use three different similarity schemes to denote the fitness of each source-target method mapping. Evolutionary algorithms use this notion of fitness to identify and select suitable source-target method mappings, for example, a method mapping with a higher fitness score is more likely to be correct and will thus be recommended. Therefore, correctly formulating the fitness function is crucial when using our approach. In this study, we use the measures described in [55,104] to calculate fitness using three components: method similarity, documentation similarity, and co-occurrence probability.

- **Method similarity:** Intuitively, source and target methods with equivalent functionality will have similar method names, return types, and input arguments. We use the formula proposed by Nguyen *et al.* [104] to calculate source and target method signature similarity. Let s and t be the source and target methods, each with a list of input arguments ip_s and ip_t, and return types r_s and r_t, respectively. Assuming that the source method name is represented as n_s and the target method name as n_t, the method similarity is calculated as:

$$\begin{aligned} Sim(s,t) =& 0.5 * sqSim(n_s, n_t) + 0.25 * sqSim(ip_s, ip_t) \\ &+ 0.25 * strSim(r_s, r_t) \end{aligned}$$

Here, the $sqSim$ function leverages the longest common sub-sequence algorithm to calculate the similarity between two word sequences, while the $strSim$ function calculates token-level similarity. We use 0.5, 0.25, and 0.25 as the weights for each component of the equation following guidance from existing literature [104]. Each weight represents the relative importance of a component, with method name similarity being assigned the highest importance. As an example, consider two methods: *void deleteDirectory(File)*, and *void deleteRecursively(File)*. Method similarity for this source-target method pair is calculated as follows:

$$\begin{aligned} Sim(s,t) &= 0.5 * sqSim(deleteDirectory, deleteRecursively) \\ &+ 0.25 * sqSim(File, File) + 0.25 * strSim(void, void) \\ &= 0.25 * (1/2) + 0.25 * (1/1) + 0.25 * (1/1) \\ &= 0.625 \end{aligned}$$

In the case of a *one-to-many* or *many-to-many* method mapping, all similarities are calculated separately for each possible pair of source and target methods. For example, consider a *one-to-many* method mapping from Figure 6.1, where the method *static IExpectationSetters expect(T)* is replaced by two target library methods *abstract T when(T)* and *abstract OngoingStubbing thenReturn(T)*. For such a *one-to-many* method mapping, method similarity would be calculated for two possible source-target method pairs: *abstract T when(T)* and *static IExpectationSetters expect(T), and static IExpectationSetters expect(T)* and *abstract OngoingStubbing thenReturn(T)*. Both target library methods would need to be selected independently of each other based on similarity scores.

- **Documentation similarity:** Intuitively, two methods that offer similar functionality will also have similar documentation. In this work, we calculate the similarity between the documentation for each source and target method pair using word vector embeddings obtained from a state-of-the-art neural network - the Universal Sentence Encoder (USE) [39]. The USE network is trained on a wide range of natural language datasets to generate sentence-level embeddings that can be used to calculate semantic similarity. Consider that s and t are our source and target methods respectively, whose documentation is an input to the USE network. We obtain the source and target documentation embeddings W_s and W_t and calculate cosine documentation similarity as follows:

$$SimDoc(s,t) = \frac{(W_s \cdot W_t)}{(||W_s|| \times ||W_t||)}$$

- **Co-occurrence probability:** In addition to method and documentation similarity, we leverage "wisdom of the crowd" knowledge from migrations performed by other developers in previously seen *migration diffs*. A source code *migration diff* contains all lines of removed and added code that we use to identify which source methods were replaced by which target methods. These code diffs can be identified using modern version control systems such as GitHub. In other terms, co-occurrence probability $CoOc(s,t)$ is how frequently a source and target method pair have been observed together so that higher fitness can be assigned to those method mappings that have been found in pre-existing programs. The co-occurrence probability is calculated by dividing the count for each source-target method pair by the maximum co-occurrence count for all method pairs.

$$coOc(s,t) = \frac{count(s,t)}{max_{s \epsilon L_s, t \epsilon L_t}(count(s,t))}$$

Here, L_s represents the source library and L_t represents the target library.

Next, we calculate an aggregate fitness score by combining co-occurrence, method, and documentation similarity scores for each source-target method pair s, t. We note that each of the three similarities is calculated as a normalized score with values in the $[0,1]$ range to give all similarities equal weight. Additionally, we divide the aggregate $(CoOc(s,t) + SimDoc(s,t) + Sim(s,t))$ score by the total number of source-target method mappings under consideration for each migration rule to ensure a fair chance of selection for every method mapping. So, our final fitness score also lies in the $[0,1]$ range. With m as the number of source methods and n as the number of target methods, our final fitness score calculation is:

$$fitness(s,t) = \frac{(coOc(s,t) + Sim(s,t) + SimDoc(s,t))}{(3 \times m \times n)}$$

6.2.5 Re-combination operators

Evolutionary algorithms use crossover and mutation genetic recombination operators to generate diverse solutions. These operators produce different combinations of source-target method map-

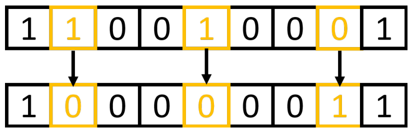


Figure 6.2: When applied to a bitstring, the mutation operator 'flips' bit values *e.g.,* bits with a value of 0 are changed to 1 at indices 1,4, and 7. Similarly, bits with a value of 1 are 'flipped' to 1. In our study, each bit corresponds to a specific source-target method mapping that may (1) or may not (0) be selected.

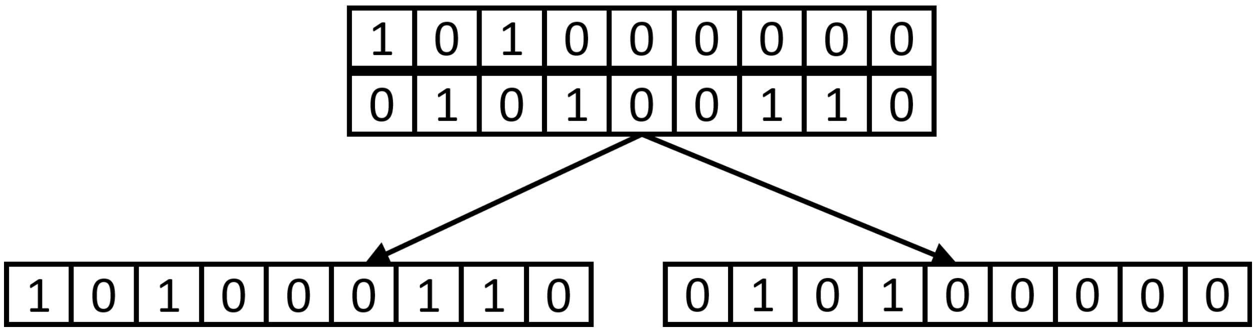


Figure 6.3: The crossover operator generates *offspring* chromosomes from two *parent* chromosomes. Each parent chromosome's bit string is divided at a pre-determined crossover point and recombined to generate new chromosomes. Crossover ensures that different solutions are explored periodically, and solutions with high fitness scores are retained.

pings at every iteration and retain method mappings with the highest fitness scores. These retained *parent* method mappings are used to generate more *offspring* by applying crossover and mutation until a termination condition is reached. They are described as follows:

- *Crossover:* Figure 6.3 depicts the crossover operator applied to two chromosomes represented as bit strings [51]. First, each chromosome is divided into two substrings at a determined crossover point. Then, each substring is recombined with the other chromosome's half to generate two offspring solutions. For example, in Figure 6.3, the first chromosome is divided into two substrings "10100" and "0000" and recombined with the second chromosome's substrings "01010" and "0110" to generate two offsprings "101000110" and "010100000". The crossover rate $P_c\%$ determines how frequently this operator is applied to a population of solutions.

- *Mutation:* Figure 6.2 illustrates how the mutation operator is applied to a bitstring. The mutation operator simply 'flips' the bit it wants to mutate *i.e.,* if the bit to be mutated has a value of 1, then it is mutated to 0 and vice versa. This operator is also associated with a mutation rate $P_m\%$ *i.e.,* it flips only $P_m\%$ of the population.

We note that there are no infeasible solutions for the method-level software library migration problem when using crossover and mutation operators. For each specific migration rule, all possible source target method mappings are represented as a bitstring that can either be selected (denoted by a 1) or not (denoted by a 0). At maximum, an algorithm can select all available target library methods resulting in a bitstring of 1s. While such a solution is undesirable, it is not infeasible as we do not impose any constraints on the selection.

6.2.6 Termination Condition and Parameters

Evolutionary algorithms iteratively generate solutions with higher fitness scores using mutation and crossover operators until a termination or stopping criterion is reached. In our experiments, we evaluate each generated solution by comparing it to a "groundtruth" obtained from the dataset used for our experiments. This "groundtruth" consists of a set of manually-validated and correct method mappings for each migration rule. However, developers do not have access to the set of correct source-target method mappings beforehand, so we cannot use a stopping criterion that measures solution quality. Hence, we set our termination criteria to the maximum number of function evaluations.

6.2.7 Algorithms Used for Search-Based API Migration

In this study, we examine one single-objective algorithm and 7 popular multi-objective algorithms that use different search and selection strategies [30, 59, 85, 149] to find solutions that balance both our objectives. As our work is the first to evaluate metaheuristic algorithms for third-party library migration at the method level, we select popular and well-understood algorithms to evaluate the impact of using evolutionary strategies for API migration. In particular, we consider GA as our single-objective algorithm, and UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, and MOEAD as our multi-objective evolutionary algorithms that are known to perform well with a smaller number of objectives. We discuss the rationale for selecting these algorithms as follows:

- **GA:** GA is a popular single-objective search algorithm that iteratively evolves a set of candidate solutions until a termination criteria is met using the *crossover* and *mutation* operators. We choose GA because it is a demonstrably efficient and accurate algorithm.
- **NSGAII:** NSGAII [51] is a widely regarded and popular search algorithm that leverages techniques such as non-dominated sorting and crowding distance to efficiently select and evolve candidate solutions. In this work, we choose NSGAII because it has been shown to perform well on problems with fewer objectives.

- **IBEA:** We select IBEA [151] because it has been demonstrated to work well on problems with fewer objectives and is a relatively well-documented algorithm that allows us to study API migration-specific challenges and characteristics. Furthermore, we use the hypervolume metric to identify diverse solutions because we are interested in "knee points" where both conflicting objectives are balanced.
- **MOEAD:** The MOEAD algorithm [146] uses decomposition strategies to search for solutions fulfilling multi-objective constraints. We select MOEAD because it uses a different search strategy that divides the search space into subproblems and because it has demonstrated desirable convergence properties on multi-objective problems.
- **UNSGAIII:** The Unified NSGAIII algorithm [126] is a unified optimization approach that modifies NSGAIII for two and mono-objective problems. We evaluate this approach due to the performance of NSGAIII on many-objective problems and improvements made to the tournament selection procedure resulting in better performance on two objective problems.
- **R-NSGAII:** We select RNSGAII [52] due to its ability to generate multiple solutions around user-specified reference points. Specifically, RNSGAII generates a set of solutions around a specified optimal point and uses Euclidean distance to evaluate generated solutions.
- **AGEMOEA:** The AGEMOEA algorithm [113] generalizes to different Pareto fronts and has been shown to outperform state-of-the-art algorithms such as NSGAIII, GrEA, MOEA/D, and AR-MOEA. We choose this algorithm because our Pareto front consists of only one "point" corresponding to our groundtruth, so its shape is unknown. Since AGEMOEA can adapt to different Pareto front shapes, we select it as one of the algorithms used in our study.
- **SMSEMOA:** The SMSEMOA algorithm [24] aims explicitly to keep solutions that maximize hypervolume. At every iteration, SMSEMOA discards solutions with the least contribution to the dominated hypervolume. Since SMSEMOA relies on hypervolume maximization, we can evaluate the efficacy of using this indicator for our problem.

6.2.8 Experimental Settings

Dataset

To evaluate the performance of multi-objective search for API migration, we use a popular dataset containing a set of manually curated mappings belonging to 9 popular library migrations collected from 57,447 open-source Java projects [12]. Details of the source and target libraries for each migration rule used in our study can be found in Appendix A. We use this dataset because it contains verified and *correct* mappings that can serve as our "groundtruth" when evaluating evolutionary algorithm performance. When conducting experiments, we generate source and target method mappings by randomly sampling from this dataset [12] to also include scenarios of incorrect method mappings that test our approach. This setting also allows us to examine algorithm behavior across a wide range of API migration scenarios containing different numbers of potential mappings with varying mapping cardinalities. We use algorithm implementations provided in the MOEA and PyMOO frameworks, as they are popular tools for implementing and evaluating various multi-objective evolutionary algorithms. We run and average the results of each search algorithm 30 times for statistical significance [16].

Algorithm Settings

We execute GA, NSGAII, IBEA, and MOEAD using the MOEA framework[1], and UNSGAIII, RNSGAII, AGEMOEA, and SMSEMOA using the PyMoo framework[2]. To ensure consistency, we specify the population size as 250 and the number of function evaluations for all algorithms as 100,000 to avoid premature convergence and ensure that diverse solutions are explored. The initial population is randomly generated for all algorithms. In our multi-objective algorithms, we set the number of selected mappings as an explicit objective that needs to be minimized, so that only method mappings with high similarity or fitness scores are selected. In contrast, GA uses an explicit constraint to limit the number of selected method mappings. We tune the crossover and mutation rates using a tree parzen estimator (TPE) included in PyMOO using the Optuna library. For algorithms executed using the MOEA framework, we conduct a grid search to determine the mutation and crossover rates. We select final parameter values using precision and recall and note that algorithm performance does not vary considerably with changes to parameters. So, we set the crossover rate to 1.0 and the mutation rate to 0.1.

When comparing our multi-objective algorithms, we follow guidance from [8, 86, 87, 88, 108] and select indicators that can incorporate developer (decision maker) preferences when recommending source-target method mappings. As our goal is to accurately identify the appropriate number of source-target method mappings, we carefully select Euclidean distance (ED) [8] and Hypervolume (HV) [87] [88] to compare multi-objective algorithm performance. When calculating these indicators for multi-objective algorithms, we specify *maximum* and *minimum* bounds for each objective against which hypervolume is calculated. To determine hypervolume, we compute the theoretical

[1]http://moeaframework.org/javadoc/index.html
[2]https://pymoo.org/algorithms/list.html

maximum and minimum values for fitness and the number of selected mappings by aggregating all fitness values and counting the total number of possible mappings for each migration rule. We calculate Euclidean distance (ED) of each solution with respect to our "groundtruth" which corresponds to the correct source-target method mappings from our dataset. Our "groundtruth" objectives are set to the aggregate fitness score of each correct method mapping and the number of verified source-target method mappings to be selected.

6.2.9 Comparative Approaches/Baselines

Random Search: A random search algorithm randomly selects potential method-mappings. We compare GA against random search to determine whether it converges to a solution, or randomly searches through our dataset. GA must outperform random search to be considered useful.

Hill-Climbing: Similar to GA, hill-climbing is an iterative algorithm that starts with a random solution and improves on the current solution by making small modifications. If hill-climbing outperforms GA, then the search space is relatively simple and does not necessitate the use of GA.

6.2.10 Metrics Used

In this work, we assess the benefits of using multi-objective optimization to select source-target method mappings during library migration. We compare our algorithms approach with a current tree-based classifier approach RAPIM [12], and random search and hill-climbing for our single-objective genetic algorithm (GA). To evaluate the efficacy of our approach, we use two sets of metrics: precision and recall for all algorithms and RAPIM; and Euclidean Distance (ED) and Hypervolume (HV) when evaluating multi-objective algorithms [8, 86, 87, 88, 108].

- **Precision:** Precision is the ratio of correctly selected mappings to all selected mappings.
- **Recall:** This is the ratio of correctly identified mappings to the number of correct mappings for each migration rule.
- **Hypervolume (HV):** The hypervolume metric evaluates how much of the objective space is covered by generated solutions and requires the specification of minimum and maximum points or bounds. To evaluate our algorithms, we set the minimum and maximum fitness as 0 and 1 respectively. Similarly, we set the minimum and maximum number of selected methods as 0. As a result, our ideal point is $[0, 0]$ and the reference point is $[1, 1]$ assuming the

minimization of both objectives. We use this indicator because we are interested in solutions that balance both our objectives: the fitness and number of recommended source-target method mappings.

- **Euclidean Distance (ED):** This indicator is calculated as the distance of each solution from the groundtruth based on each objective value. The ED metric evaluates solution quality by measuring how close generated solutions are to the desired point as specified by the groundtruth. When calculating ED, we normalize both fitness and the number of selected methods to be within [0,1] and report the mean values.

6.3 Results

Our goal is to evaluate the efficacy of using a search-based approach to to recommend methods during software library migration. The following research questions drive our experimentation:

- **RQ7**: How accurately can our single-objective genetic algorithm (GA) approach recommend source-target method mappings? *We compare GA performance against random search and hill-climbing to demonstrate the effectiveness of our single-objective approach during API migration.*
- **RQ8**: How effectively do UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, and MOEAD recommend source-target library method mappings for various migration rules? *We find that our multi-objective algorithms tend to outperform GA. Furthermore, we compare our multi-objective algorithms using hypervolume and Euclidean distance. We also plot solution sets generated by each algorithm to gain insight into their performance.*
- **RQ9**: What is the impact of using different similarity score schemes (method signature, documentation, and co-occurrence probabilities) when recommending source-target method mappings? *We compare precision and recall achieved by each algorithm using three similarity schemes: co-occurrence probability (CO), method and documentation similarity (MS+DS), and a combination of MS+DS and CO (ALL).*

6.3.1 RQ 7: GA Effectiveness Compared To Random Search And Hill Climbing

Table 6.2 shows the precision and recall obtained using GA, random search (RS) and hill-climbing (HC) averaged over 30 runs for different migration rules. We execute GA, RS and HC 30 times and use the Wilcoxon signed rank test ($p<0.05$) and observe that our results for each algorithm pair are statistically significant. In general, we observe that GA has a higher precision than both random search and hill-climbing for all migration rules, indicating that it can identify correct source-target method mappings. We observe that GA achieves the highest values of precision and recall compared to both baselines for rules $google - collect \rightarrow guava$ and $json \rightarrow gson$. These libraries contain a majority of one-to-one mappings and well-defined similarity scores resulting in good performance. While GA also achieves perfect precision when migrating between $commons - lang \rightarrow slf4j - api$ and $json - simple \rightarrow gson$, it recommends mappings extremely conservatively leading to poor recall. Similarly, GA achieves high precision and poor recall when migrating between $logging \rightarrow slf4j$, $slf4j - api \rightarrow log4j$, $gson \rightarrow jackson$ because these libraries contain a majority of *many-to-one, many-to-many and one-to-many* mappings that makes the search process difficult. Overall, the worst performance is observed for the rules $easymock \rightarrow mockito$ and $testng \rightarrow junit$ where the search space is complex due to multiple local optima and close to zero similarity scores. We discuss the different factors impacting GA performance in greater detail as follows:

The effect of capacity:

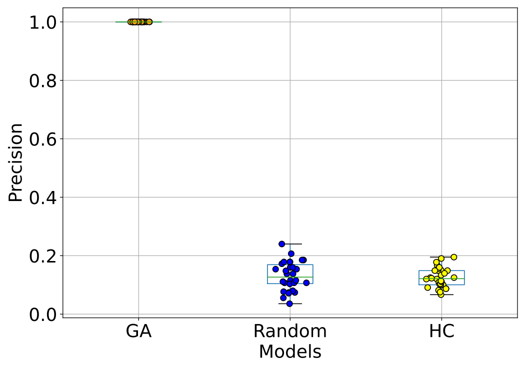
The capacity value is critical to GA performance as it determines which and how many target methods are recommended. However, the value of capacity is difficult to determine exactly because we cannot know how many mappings exist between two libraries unless all possible combinations are explored. Due to these reasons, we use a case study to conduct an experiment to determine the effect of different capacity values on GA performance. In this regard, we use the rule $commons - lang \rightarrow$

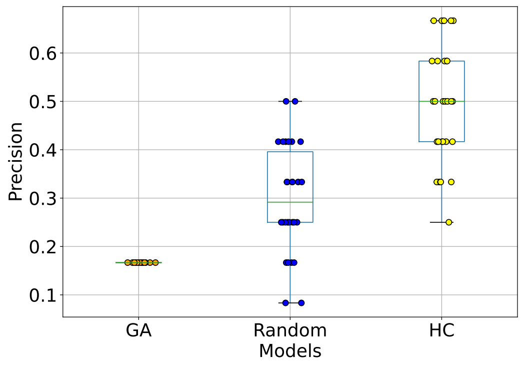
Table 6.2: Precision and recall values for GA, random search and hill-climbing obtained using 100 individuals and 50000 function evaluations.

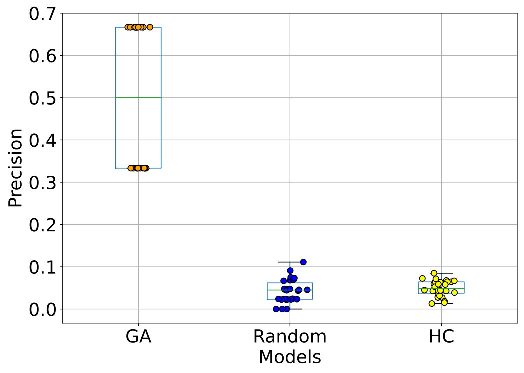
	Precision			Recall		
Migration Rule (↓)	**GA**	**RS**	**HC**	**GA**	**RS**	**HC**
logging → slf4j	**0.7166**	0.3148	0.3039	0.0855	0.3621	**0.4950**
slf4j-api → log4j	**0.4639**	0.2019	0.1944	0.1216	0.3807	**0.4912**
easymock → mockito	**0.0331**	0.0339	0.0331	0.0622	0.4177	**0.5000**
google-collect → guava	**0.4444**	0.0685	0.0721	**0.6666**	0.2444	0.4777
gson → jackson	**0.5000**	0.0478	0.0486	0.2500	0.3444	**0.5444**
testng → junit	**0.1417**	0.0769	0.0831	0.0500	0.3615	**0.5038**
json → gson	**0.6666**	0.4500	0.2152	**0.6666**	0.4388	0.5333
commons-lang → slf4j-api	**1.0000**	0.1281	0.1252	0.1666	0.2944	**0.4944**
json-simple → gson	**1.0000**	0.2541	0.2216	0.1666	0.3055	**0.4925**

$slf4j - api$ as a case study and evaluate the effect of increasing capacity on precision and recall in Figure 6.5. We observe that precision values are highest for low capacity values, in fact, we obtain perfect precision up to a capacity value of 4. This is expected because GA recommends methods conservatively leading to almost no false positives. As the capacity values increase, more correct and incorrect source-target methods are recommended. This increases the likelihood of selecting wrong mappings resulting in more false positives and lower precision values. Moreover, as capacity increases more correct methods are also identified leading to fewer false negatives thereby increasing recall. From the graph, we observe that a balance between precision and recall is achieved when the capacity is 12, which is also the number of correct source-target mappings that exist in the ground truth. That is, GA balances precision and recall when the capacity is set to the number of source methods to be replaced (which we know beforehand). Thus, in subsequent experiments, we set the capacity for each migration rule to be equal to the number of source methods to be replaced.

Next, we study the impact of capacity on different migration rules. When the capacity value is well-specified, GA achieves high precision and recall as demonstrated in Table 6.2 for the rules $google - collect \rightarrow guava$ and $json \rightarrow gson$. This also indicates that the source and target libraries involved in these migrations deliver the same functionality and thus have many similar methods leading to several correct potential mappings *i.e.,* they have equivalent functionality. This is also demonstrated in Figure 6.9 where GA consistently achieves relatively higher recall scores as the capacity value is equal to the number of mappings that must be found. When the capacity value is relatively high, GA evolves solutions containing a large number of mappings which increases the probability of finding the wrong mappings (more false positives), resulting in low precision. This is reflected in GA performance for rules $logging \rightarrow slf4j$, $slf4j - api \rightarrow log4j$, $gson \rightarrow jackson$ where it obtains high values of precision but does not select all correct mappings. When the capacity value is relatively low, GA becomes *picky*, and selects very few source-target method mappings. This increases precision because selected methods are correct, but also results in poor recall because a large number of correct mappings are not selected, resulting in low recall *e.g.,* $commons - lang \rightarrow slf4j - api$ and $json - simple \rightarrow gson$. This is illustrated further in Figure 6.9 where GA consistently achieves perfect precision and poor recall because only a few methods are correctly recommended compared to the actual number of correct mappings that can be found. In an extreme scenario, we also observe that GA is extremely picky when recommending mappings between $easymock \rightarrow mockito$ because it must evaluate a large number of target methods to recommend very few mappings (1% of all potential mappings). In this scenario, GA is unable to find a solution and instead randomly selects target methods to fulfill the capacity constraints.







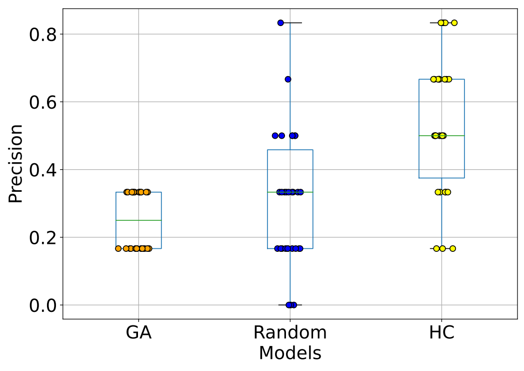
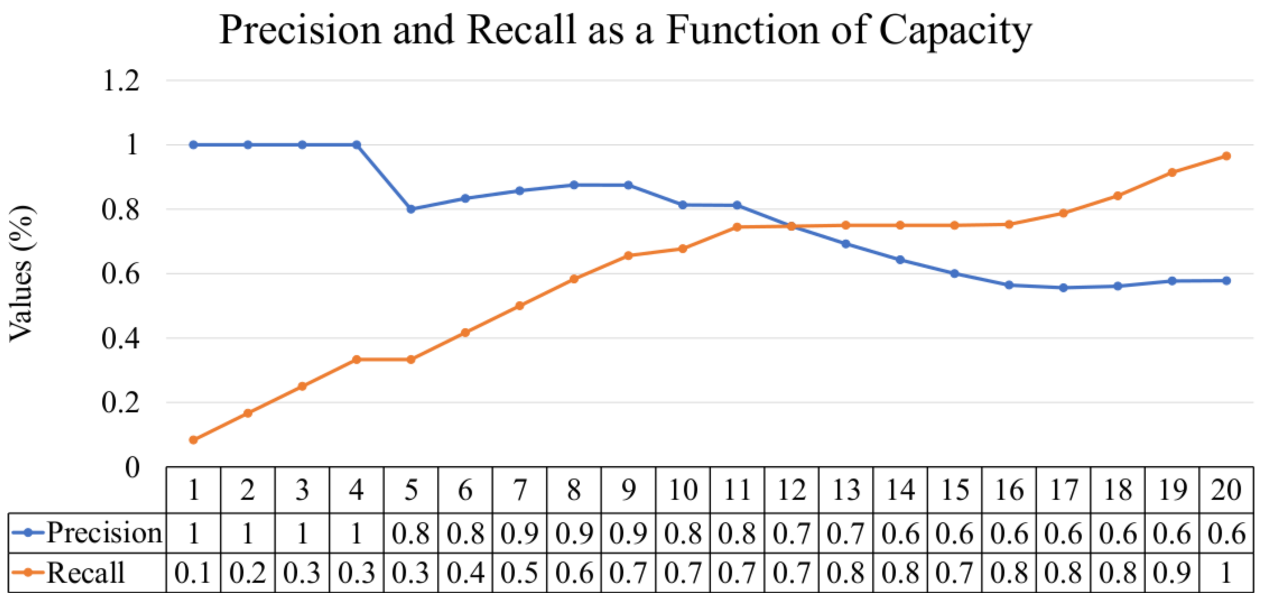


Figure 6.4: Precision and recall values over 30 runs demonstrate that GA outperforms random search and hill-climbing to recommend API mappings with high precision.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Precision	1	1	1	1	0.8	0.8	0.9	0.9	0.9	0.8	0.8	0.7	0.7	0.6	0.6	0.6	0.6	0.6	0.6	0.6
Recall	0.1	0.2	0.3	0.3	0.3	0.4	0.5	0.6	0.7	0.7	0.7	0.7	0.8	0.8	0.7	0.8	0.8	0.8	0.9	1

Figure 6.5: This graph demonstrates the effect of increasing the capacity values for the rule $commons - lang \rightarrow slf4j - api$. As the capacity increases from 1 to 20, we observe a decline in precision values and an increase in recall values. This is because more source-target mappings are recommended for higher capacities.

The effect of various cardinalities:

In general, it is difficult to find replacement methods for *many-to-one, one-to-many and many-to-many* mappings. It is relatively easier to recommend for one-to-one mappings, because only one target method (usually with the highest similarity score) needs to be recommended for each source method. Thus, the search process is simplified because it must only find m target methods with the highest similarity. This is reflected in both the precision and recall values when migrating between $google-collect \rightarrow guava$, $testng \rightarrow junit$ where the majority of mappings in these datasets are one-to-one, so the number of recommended methods is close to the number of actually correct source-target method mappings that exist. This leads to fewer false positives, thus resulting in high values of precision. Furthermore, GA exhibits perfect precision but poor recall when migrating between $commons - lang \rightarrow slf4j - api$ and $json - simple \rightarrow gson$ because it selects very few methods. These selected methods are correct and therefore lead to almost no false positives, however, GA cannot accurately identify all correct mappings and thus returns a high number of false negatives that lead to a low recall. This occurs because of a majority of one-to-many, many-to-one or many-to-many mappings where several methods are 'helper' functions that do not contain the main programming logic but are used to perform repetitive tasks *e.g.,* basic error handling or typecasting using the `toString()` method. As a result, there are many correct source-target mappings with similar fitness scores and GA is unable to differentiate between them, resulting in poor recall.

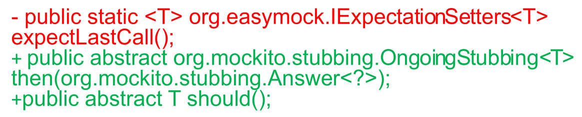
The effect of libraries' characteristics:

Additionally, we find that two characteristics impact search performance: the number of methods in source and target libraries (dataset size) and the number of potential source-target mappings with a zero similarity score. The migrations $logging \rightarrow slf4j$ and $testng \rightarrow junit$ contain various methods that are close in signature and documentation, *i.e.,* they contain the *least* number of source-target mappings with zero similarity. This results in a difficult search space with multiple *local optima*, thus making it hard for the fitness to discriminate wrong mappings. In contrast, the migration rules $easymock \rightarrow mockito$, $google - collect \rightarrow guava$, $gson \rightarrow jackson$, $json \rightarrow gson$, and $commons - lang \rightarrow slf4j - api$ contain a large number of source-target mappings that have a fitness of zero (distant method signatures and different documentations). That is, most of the correct source-target mappings have a non-zero similarity score. Moreover, the recall value for the rule $gson \rightarrow jackson$ is relatively higher compared to $easymock \rightarrow mockito$ and $testng \rightarrow junit$. This because the dataset for this rule contains the largest number of source-target methods with zero similarity, leading to fewer local optima.

The effect of different similarity scores:

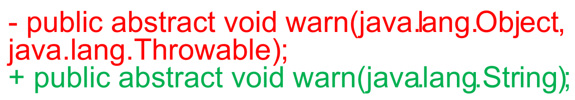
In general, each scoring measure (MS, DS and CO) captures a different dimension that can be useful for selecting correct source-target mappings. However, there are some exceptions: for example, some libraries do not have proper documentation for each method. In these scenarios, documentation similarity (DS) is zero and not only it does not provide the intended result, but it also biases the search: in our approach, if the documentation similarity is calculated as zero, it may drive the overall fitness closer to zero and adversely impact search performance. Thus, search algorithms may not select correct mappings due to poor documentation scores.

Combining the MS, DS and CO results in better recommendations as compared to using them separately, especially if one of them cannot be properly calculated (*e.g.,* in case of poor documentation). As a result, during our manual validation, we found that GA can also discover other mappings that are correct but have not been seen previously (absent from ground truth). For example, in Figure 6.6b, the correct mapping from `public abstract void warn(java.lang.Object, java.lang.Throwable)` to `public abstract void warn(java.lang.String)` has been returned by our algorithm, while being absent from the existing dataset. Another example in Figure 6.6a shows how the methods `public static <T> org.easymock.IExpectationSetters<T> expectLastCall()` and `public abstract T should()` have signature and documentation similarities as zero, but a non-zero co-occurrence score, thus leading to it being correctly recommended.



```
- public static <T> org.easymock.IExpectationSetters<T>
expectLastCall();
+ public abstract org.mockito.stubbing.OngoingStubbing<T>
then(org.mockito.stubbing.Answer<?>);
+public abstract T should();
```

(a) A one-to-many mapping. The method similarity score for *expectLastCall()* to *should()* is zero as their signatures are zero, as is the documentation score. However, as this mapping has been used before in our dataset, it has a co-occurrence of 1.



```
- public abstract void warn(java.lang.Object,
java.lang.Throwable);
+ public abstract void warn(java.lang.String);
```

(b) A discovered one-to-one mapping selected by GA. In this case, GA selects this mapping because the method similarity before normalization is calculated as 0.5 due to the same return type and method name.

Figure 6.6: Examples of correct mappings found by our algorithm.

6.3.2 RQ 8: Evaluating Multi-Objective Search

In subsequent research questions, we compare multiple multi-objective algorithms, so we determine the statistical significance of our results using the Kruskal-Wallis test ($p<0.05$) as it allows us to compare multiple groups at once. A p-value < 0.05 indicates that there is a significant difference in the values of the indicators of our algorithms. Results are statistically significant unless noted otherwise.

Table 6.3: Using the CO scheme, UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD achieve higher precision compared to our single-objective GA approach. We use the Kruskal-Wallis test to evaluate statistical significance.

Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**NSGAII**	**IBEA**	**MOEAD**	**GA**
$logging \rightarrow slf4j$	0.96	0.98	**1**	0.96	0.95	0.98	0.76	0.7
$slf4j-api \rightarrow log4j$	1	1	1	1	1	1	0.87	0.95
$easymock \rightarrow mockito$	0.94	0.92	0.96	0.92	0.95	**0.97**	0.87	0.85
$google-collect \rightarrow guava$	**1**	**1**	**1**	**1**	0.95	0.98	0.58	0.12
$gson \rightarrow jackson$	**1**	**1**	**1**	**1**	**1**	**1**	0.88	0.64
$testng \rightarrow junit$	0.56	0.48	0.68	0.52	0.62	**0.74**	0.32	0.29
$json \rightarrow gson$	0.67	0.67	0.83	0.66	0.75	**0.85**	0.36	0.3
$commons-lang \rightarrow slf4j-api$	**1**	**1**	**1**	**1**	**1**	**1**	0.9	0.68
$json-simple \rightarrow gson$	**1**	**1**	**1**	**1**	**1**	0.95	0.88	**1**
Average	0.90	0.89	0.94	0.90	0.91	0.94	0.71	0.61

Tables 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 present precision and recall values achieved by UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD and the single-objective genetic algorithm (GA) [55]. We evaluate each algorithm using three similarity schemes: co-occurrence probability (CO), a combination of method and documentation similarity (MS+DS), and an ag-

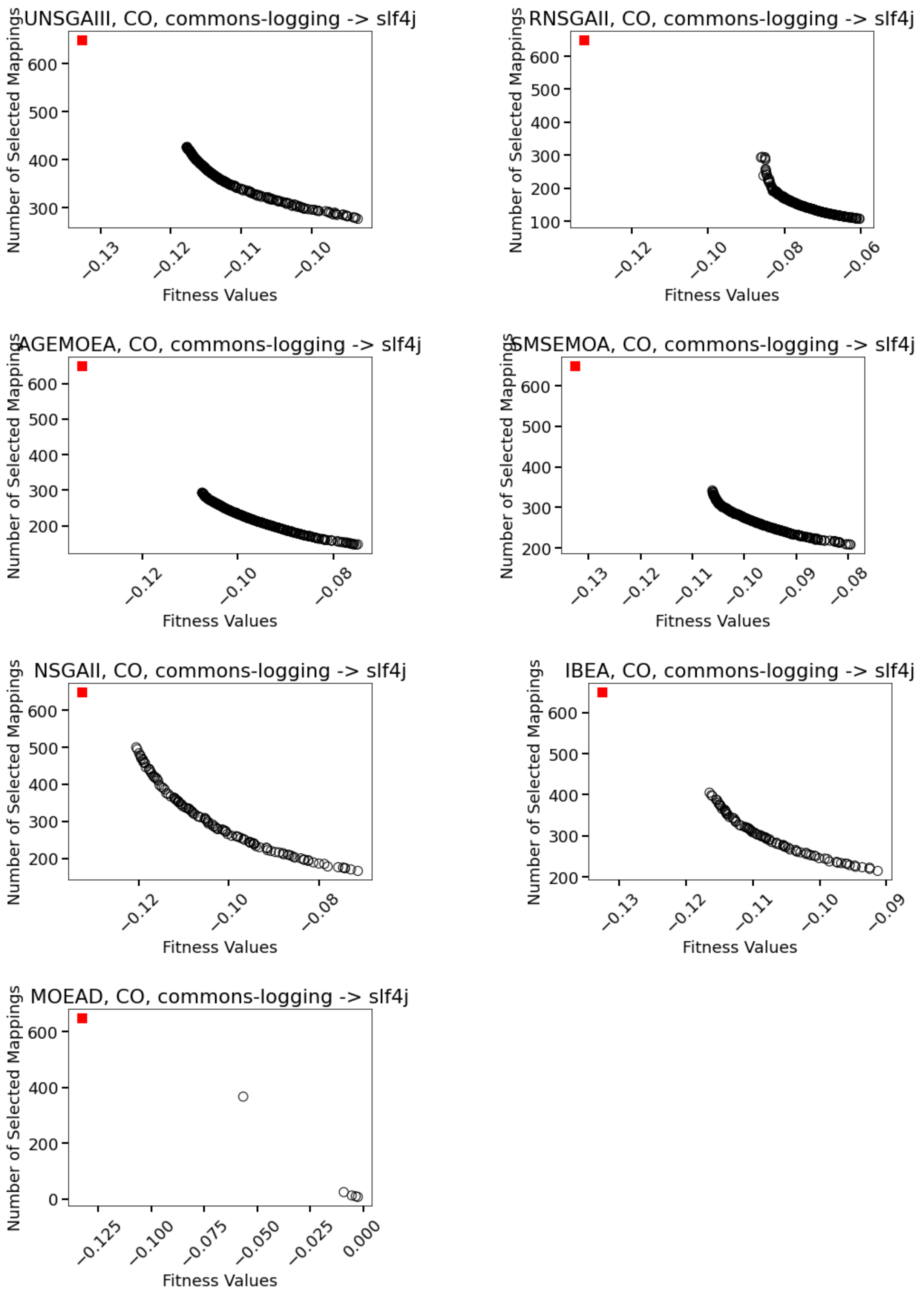


Figure 6.7: We visualize solutions closest to the mean Euclidean distance value for each algorithm for the $logging \rightarrow slf4j$ migration rule. The overall best-performing algorithm is UNSGAIII. Note that in these plots we depict both objectives as minimization for readability, hence the fitness values are negatives as they must be maximized.

Table 6.4: Using the ALL scheme, UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD achieve higher precision compared to our single-objective GA approach. We use the Kruskal-Wallis test to evaluate statistical significance.

Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**NSGAII**	**IBEA**	**MOEAD**	**GA**
$logging \rightarrow slf4j$	0.75	0.8	**0.81**	0.77	0.80	**0.81**	0.75	0.68
$slf4j - api \rightarrow log4j$	0.32	0.47	0.21	0.19	0.27	0.24	**0.65**	0.49
$easymock \rightarrow mockito$	0.91	0.91	0.93	0.9	0.93	**0.94**	0.87	0.85
$google - collect \rightarrow guava$	0.24	0.46	0.28	0.27	0.30	0.28	**0.48**	0.11
$gson \rightarrow jackson$	0.59	0.68	0.58	0.57	0.59	0.58	**0.76**	0.53
$testng \rightarrow junit$	0.36	0.39	**0.43**	0.38	0.40	0.41	0.32	0.28
$json \rightarrow gson$	0.36	**0.41**	0.42	0.37	0.39	0.40	0.32	0.3
$commons - lang \rightarrow slf4j - api$	0.8	**0.95**	0.82	0.79	0.82	0.81	0.87	0.66
$json - simple \rightarrow gson$	0.48	0.81	0.57	0.54	0.74	0.74	**0.86**	0.58
Average	0.53	**0.65**	0.56	0.53	0.58	0.58	**0.65**	0.50

Table 6.5: Using the MS+DS scheme, UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD achieve higher precision compared to our single-objective GA approach. We use the Kruskal-Wallis test to evaluate statistical significance.

Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**NSGAII**	**IBEA**	**MOEAD**	**GA**
$logging \rightarrow slf4j$	0.76	**0.86**	0.8	0.81	0.79	0.81	0.74	0.69
$slf4j - api \rightarrow log4j$	0.25	0.53	0.18	0.17	0.25	0.23	**0.62**	0.46
$easymock \rightarrow mockito$	0.92	0.92	0.91	0.91	0.92	**0.94**	0.87	0.86
$google - collect \rightarrow guava$	0.23	**0.52**	0.29	0.28	0.3	0.28	0.47	0.12
$gson \rightarrow jackson$	0.59	0.55	0.56	0.56	0.55	0.54	**0.7**	0.55
$testng \rightarrow junit$	0.38	**0.43**	0.39	0.39	0.39	0.41	0.32	0.29
$json \rightarrow gson$	0.38	**0.41**	**0.41**	0.38	0.38	0.39	0.32	0.29
$commons - lang \rightarrow slf4j - api$	0.7	0.7	0.79	0.8	0.75	0.76	**0.82**	0.65
$json - simple \rightarrow gson$	0.42	0.75	0.54	0.52	0.7	0.71	0.84	**1**
Average	0.51	**0.63**	0.54	0.54	0.56	0.56	**0.63**	0.55

Table 6.6: Using the CO scheme, UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD achieve higher recall compared to our single-objective GA approach. We use the Kruskal-Wallis test to evaluate statistical significance.

Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**NSGAII**	**IBEA**	**MOEAD**	**GA**
$logging \rightarrow slf4j$	**0.52**	0.23	0.34	0.4	0.43	0.43	0.06	0.14
$slf4j - api \rightarrow log4j$	**0.99**	0.04	0.54	0.78	0.55	0.53	0.14	0.19
$easymock \rightarrow mockito$	**0.5**	0.36	0.41	0.44	0.42	0.4	0.12	0.26
$google - collect \rightarrow guava$	**1**	0.26	0.91	0.92	0.77	0.92	0.11	0.12
$gson \rightarrow jackson$	**1**	0.01	0.52	0.57	0.52	0.52	0.08	0.05
$testng \rightarrow junit$	0.68	0.43	0.68	0.6	0.66	**0.72**	0.1	0.26
$json \rightarrow gson$	0.77	0.53	0.69	0.73	0.73	**0.79**	0.1	0.22
$commons - lang \rightarrow slf4j - api$	**1**	0.03	0.55	0.62	0.58	0.58	0.07	0.04
$json - simple \rightarrow gson$	**1**	0.21	0.6	0.62	0.6	0.5	0.27	0.15
Average	**0.83**	0.23	0.58	0.63	0.58	0.6	0.17	0.16

gregate of co-occurrence, method and documentation similarities (ALL). From Table 6.3 and 6.6,

Table 6.7: Using the ALL scheme, UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD achieve higher recall compared to our single-objective GA approach. We use the Kruskal-Wallis test to evaluate statistical significance.

Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**NSGAII**	**IBEA**	**MOEAD**	**GA**
$logging \to slf4j$	**0.56**	0.3	0.45	0.43	0.44	0.44	0.08	0.14
$slf4j - api \to log4j$	**0.56**	0.13	0.21	0.18	0.24	0.18	0.11	0.1
$easymock \to mockito$	**0.52**	0.38	0.43	0.45	0.44	0.43	0.14	0.26
$google - collect \to guava$	**0.61**	0.51	0.6	0.6	0.59	0.6	0.14	0.1
$gson \to jackson$	**0.52**	0.12	0.33	0.31	0.32	0.3	0.08	0.04
$testng \to junit$	**0.61**	0.47	0.59	0.58	0.56	0.57	0.14	0.25
$json \to gson$	**0.49**	0.37	0.4	0.43	0.42	0.41	0.1	0.22
$commons - lang \to slf4j - api$	**0.69**	0.16	0.42	0.43	0.44	0.43	0.07	0.04
$json - simple \to gson$	**1**	0.65	0.78	0.8	0.68	0.66	0.31	0.15
Average	**0.62**	0.34	0.47	0.47	0.46	0.45	0.13	0.14

Table 6.8: Using the MS+DS scheme, UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, MOEAD achieve higher recall compared to our single-objective GA approach. We use the Kruskal-Wallis test to evaluate statistical significance.

Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**NSGAII**	**IBEA**	**MOEAD**	**GA**
$logging \to slf4j$	**0.58**	0.25	0.46	0.45	0.43	0.43	0.08	0.14
$slf4j - api \to log4j$	**0.48**	0.11	0.17	0.15	0.17	0.15	0.11	0.09
$easymock \to mockito$	**0.53**	0.33	0.43	0.43	0.43	0.42	0.14	0.27
$google - collect \to guava$	**0.61**	0.35	0.59	0.6	0.56	0.59	0.15	0.1
$gson \to jackson$	**0.52**	0.1	0.32	0.35	0.27	0.27	0.07	0.04
$testng \to junit$	**0.65**	0.46	0.55	0.58	0.56	0.57	0.14	0.25
$json \to gson$	**0.48**	0.27	0.39	0.4	0.41	0.4	0.1	0.22
$commons - lang \to slf4j - api$	**0.58**	0.09	0.28	0.26	0.3	0.28	0.06	0.04
$json - simple \to gson$	**0.8**	0.61	0.74	0.76	0.62	0.59	0.3	0.15
Average	**0.58**	0.29	0.44	0.44	0.42	0.41	0.13	0.14

we observe that all 7 evolutionary algorithms achieve high precision and recall when using the CO scheme. In fact, for $logging \to slf4j$, $slf4j - api \to log4j$, $google - collect \to guava$, $json \to gson$, $commons - lang \to slf4j - api$, atleast one algorithm achieves 100% precision. Additionally, for $easymock \to mockito$ while 100% precision is not achieved, 6 out of 7 algorithms achieve precision above 90% precision. We observe that precision and recall values are highest when using the CO scheme because it provides an ideal scenario where prior knowledge about the preferred source-target method mappings is available in the form of probabilities.

While each of the 7 algorithms achieves high precision and recall, we observe that UNSGAII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, and IBEA outperform our single-objective GA approach on all migration rules. This is because GA uses a pre-defined *capacity* constraint to control the number of method mappings that can be selected. This capacity constraint is equal to the number of source methods which results in GA selecting fewer method mappings leading to better recall. However, since it difficult to know how many target methods will be required for each source method, GA performance varies based on mapping cardinalities *e.g.,* in the case of one-to-many mappings. Our multi-objective approach addresses this limitation by modeling the number of selected target methods as an objective. As a result, our multi-objective approach achieves higher precision and recall compared to GA, and it is also easier to use.

We observe that GA achieves high precision on $json - simple \rightarrow gson$ but suffers from poor recall. In this case, GA achieves high precision but low recall due to the under-selection of correct method mappings. Note that using a pre-defined capacity constraint could be more useful when considering only *one-to-one* method mappings, leading to more precise selections since the number of target methods to be selected is known beforehand. Unfortunately, the capacity constraint tends to be poorly defined for *one-to-many*, *many-to-one*, and *many-to-many* mappings, so GA selects fewer methods leading to poor recall.

Additionally, we compare the performance of multi-objective search with the RAPIM approach [12]. Figure 6.8 plots the best precision and recall values achieved by our approach and RAPIM. We observe that our multi-objective approach achieves higher precision and recall than RAPIM. Similar to the GA approach, RAPIM performs well on one-to-one mappings and does not generalize well to other mapping cardinalities resulting in poor precision and recall across different migration rules. Thus, we conclude that our approach outperform current state-of-the-art API migration technique RAPIM [12] and our single-objective GA approach.

Tables 6.3 and 6.6 demonstrate the differences in UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, and MOEAD performance using the CO scheme. In general, we observe that all algorithms achieve high precision on most migration rules. From Table 6.6, we observe that UNSGAIII achieves the highest or close to the highest recall value on all migration rules. These results suggest that UNSGAIII can correctly identify true and false positives and negatives, making it a powerful algorithm for API migration. In general, we observe that MOEAD demonstrates comparatively poor performance, in particular for $testng \rightarrow junit$ and $json \rightarrow gson$ for precision and all other rules for recall. We examine this further in Figure 6.7 by plotting the "best" pareto front achieved by each of the 7 algorithms used in this study. The "best" pareto front is the one that achieves the lowest mean Euclidean distance from the groundtruth. We observe that MOEAD generates fewer solutions that are also further away from the groundtruth, suggesting that MOEAD is more selective when identifying method mappings compared to the other algorithms and resulting in poor precision and recall. We also observe that 6 out of 7 algorithms generate similar pareto fronts, and UNSGAIII and NSGAII generate more balanced solutions that minimize the number of selected mappings while maximizing fitness.

From Table 6.9, we observe that UNSGAIII and IBEA achieve the lowest ED values across all migration rules, while RNSGAII achieves high hypervolume values when using the CO similarity scheme. This difference between ED values obtained by UNSGAIII and the other algorithms is most pronounced when using the CO scheme. However, as seen from Tables 6.10 and 6.11, UNSGAIII is able to find solutions with lower ED values when using the ALL and MS+DS schemes, but it is outperformed by RNSGAII, AGEMOEA and IBEA on multiple migration rules. Moreover, UNSGAIII suffers from relatively low hypervolume across all three similarity schemes, although the magnitude of all indicators is similar. We examine this further in Figure 6.9, where we plot HV and ED values against the number of function evaluations. We observe that UNSGAIII converges to a good HV value after $\sim$ 60000 function evaluations and converges to a very low ED value as shown in Table 6.9. In contrast, for RNSGAII HV and ED values slightly increase with the number of function evaluations from 0.56 to 0.68. In contrast, UNSGAIII ED drastically decreases from $\sim$0.65 to $\sim$0.03, in addition to high precision and recall values. We observe similar trends for different migration rules and provide all plots and results in our replication package.

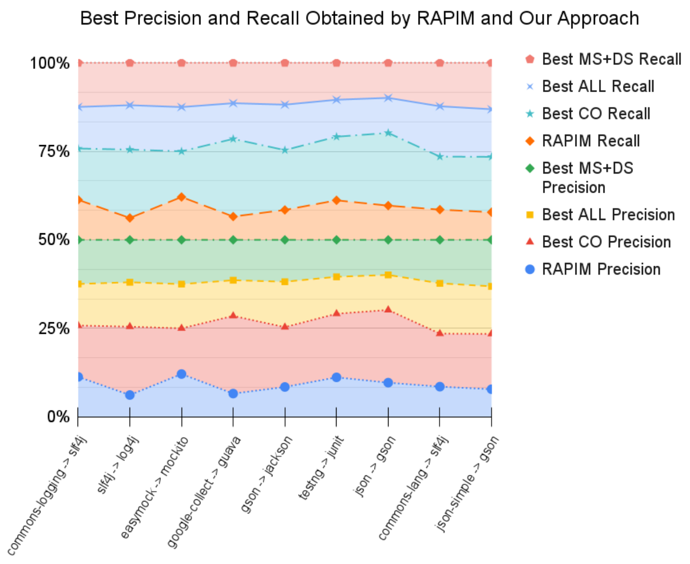


Figure 6.8: Our multi-objective approach outperforms the RAPIM approach presented in [12] using all three schemes in terms of precision and recall. We note that the RAPIM approach is trained for one-to-one method mappings, and uses 8 features derived from different components of each of three schemes (*e.g.,* return type description, method signature similarity etc.).

Table 6.9: Hypervolume and ED values achieved by UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA, and MOEAD using the CO scheme. We use the Kruskal-Wallis test to evaluate statistical significance.

	UNSGAIII		**RNSGAII**		**AGEMOEA**		**SMSEMOA**		**NSGAII**		**IBEA**		**MOEAD**	
Migration Rule (↓)	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**
$logging \rightarrow slf4j$	0.43	0.62	0.68	0.57	0.54	0.67	0.48	0.62	0.45	0.71	0.44	0.67	1.11	0.28
$slf4j - api \rightarrow log4j$	0.06	0.66	1.02	0.39	0.42	0.83	0.16	0.8	0.39	0.83	0.42	0.83	0.91	0.45
$easymock \rightarrow mockito$	0.55	0.42	0.72	0.39	0.6	0.47	0.58	0.42	0.59	0.54	0.59	0.53	1.13	0.27
$google - collect \rightarrow guava$	0.03	0.91	0.72	0.91	0.06	0.93	0.04	0.92	0.18	0.93	0.06	0.91	0.89	0.39
$gson \rightarrow jackson$	0.01	0.51	1.1	0.19	0.38	0.73	0.33	0.69	0.4	0.78	0.39	0.75	1.01	0.37
$testng \rightarrow junit$	0.17	0.57	0.36	0.5	0.14	0.65	0.22	0.55	0.18	0.69	0.13	0.69	0.9	0.2
$json \rightarrow gson$	0.18	0.58	0.39	0.52	0.21	0.66	0.19	0.57	0.2	0.72	0.14	0.71	0.92	0.24
$commons - lang \rightarrow slf4j - api$	0.01	0.39	1.13	0.3	0.46	0.64	0.39	0.59	0.44	0.68	0.44	0.69	1.07	0.35
$json - simple \rightarrow gson$	0.27	0.85	0.75	0.85	0.31	0.89	0.29	0.89	0.33	0.88	0.44	0.89	0.68	0.72

Table 6.10: Hypervolume and ED values achieved by UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA and MOEAD using the ALL scheme. We use the Kruskal-Wallis test to evaluate statistical significance.

	UNSGAIII		**RNSGAII**		**AGEMOEA**		**SMSEMOA**		**NSGAII**		**IBEA**		**MOEAD**	
Migration Rule (↓)	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**
$logging \rightarrow slf4j$	0.28	0.49	0.52	0.46	0.33	0.55	0.33	0.5	0.34	0.58	0.34	0.57	0.89	0.41
$slf4j - api \rightarrow log4j$	0.58	0.46	0.35	0.64	0.48	0.75	0.48	0.74	0.48	0.75	0.44	0.73	0.35	0.57
$easymock \rightarrow mockito$	0.51	0.4	0.68	0.38	0.56	0.43	0.54	0.39	0.55	0.49	0.56	0.49	1.06	0.35
$google - collect \rightarrow guava$	0.62	0.74	0.26	0.73	0.55	0.81	0.56	0.78	0.48	0.86	0.55	0.82	0.26	0.48
$gson \rightarrow jackson$	0.34	0.55	0.53	0.62	0.29	0.75	0.28	0.73	0.34	0.78	0.32	0.75	0.65	0.44
$testng \rightarrow junit$	0.24	0.4	0.08	0.38	0.2	0.43	0.23	0.4	0.22	0.49	0.22	0.49	0.35	0.35
$json \rightarrow gson$	0.3	0.54	0.13	0.47	0.29	0.59	0.27	0.51	0.31	0.65	0.34	0.63	0.41	0.34
$commons - lang \rightarrow slf4j - api$	0.13	0.44	0.74	0.59	0.34	0.7	0.31	0.67	0.34	0.71	0.33	0.68	0.91	0.37
$json - simple \rightarrow gson$	0.29	0.61	0.2	0.77	0.25	0.82	0.25	0.78	0.25	0.82	0.26	0.82	0.41	0.63

Table 6.11: Hypervolume and ED values achieved by UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA and MOEAD using the MS+DS scheme. We use the Kruskal-Wallis test to evaluate statistical significance.

	UNSGAIII		**RNSGAII**		**AGEMOEA**		**SMSEMOA**		**NSGAII**		**IBEA**		**MOEAD**	
Migration Rule (↓)	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**	**ED**	**HV**
$logging \rightarrow slf4j$	0.29	0.47	0.54	0.45	0.33	0.52	0.34	0.48	0.36	0.56	0.36	0.53	0.88	0.42
$slf4j - api \rightarrow log4j$	0.67	0.41	0.35	0.63	0.52	0.75	0.5	0.72	0.47	0.76	0.43	0.72	0.34	0.58
$easymock \rightarrow mockito$	0.52	0.38	0.69	0.37	0.57	0.42	0.56	0.39	0.57	0.48	0.57	0.47	1.05	0.36
$google - collect \rightarrow guava$	0.59	0.75	0.11	0.7	0.49	0.82	0.52	0.77	0.41	0.86	0.5	0.82	0.26	0.5
$gson \rightarrow jackson$	0.37	0.57	0.52	0.58	0.29	0.73	0.27	0.69	0.35	0.78	0.32	0.74	0.66	0.45
$testng \rightarrow junit$	0.24	0.39	0.06	0.38	0.2	0.42	0.2	0.39	0.21	0.48	0.21	0.47	0.34	0.36
$json \rightarrow gson$	0.3	0.51	0.12	0.46	0.27	0.57	0.26	0.5	0.29	0.63	0.32	0.61	0.39	0.35
$commons - lang \rightarrow slf4j - api$	0.3	0.54	0.72	0.61	0.42	0.77	0.4	0.75	0.42	0.79	0.41	0.74	0.82	0.4
$json - simple \rightarrow gson$	0.43	0.6	0.11	0.71	0.33	0.8	0.36	0.78	0.27	0.8	0.28	0.8	0.34	0.58

6.3.3 RQ 9: Assessing Similarity Schemes

Tables 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10 and 6.11 depict the impact of using different similarity schemes to recommend method mappings using multi-objective algorithms. All algorithms achieve

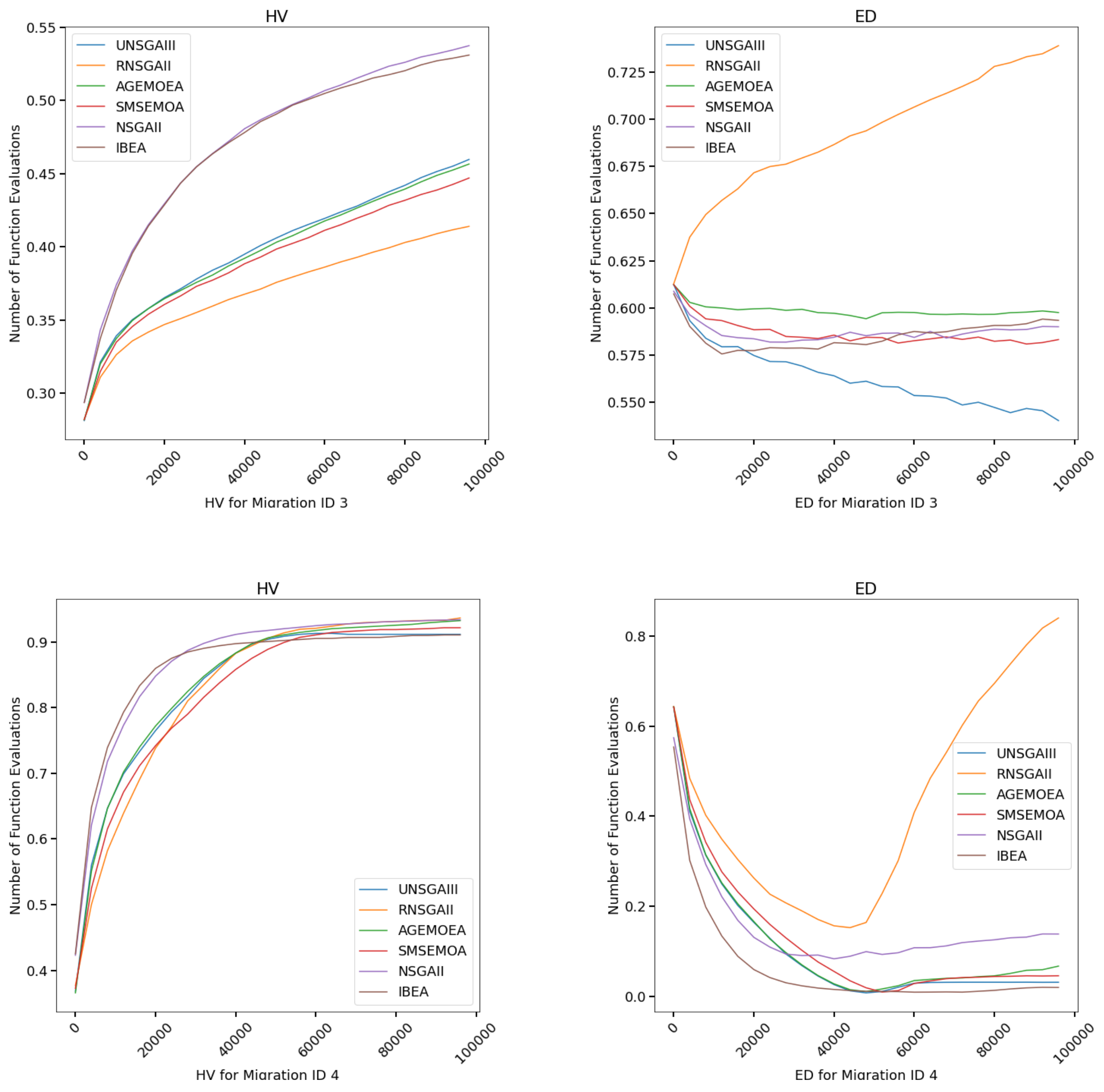


Figure 6.9: In general, we observe that ED values increase and HV values decrease as the number of function evaluations increases when using UNSGAIII for rules *easymock* → *mockito* and *google* − *collect* → *guava*. In contrast, the ED values for RNSGAII increase for *easymock* → *mockito*, although the change is not considerably higher. Refer to our replication package for more plots: http://bit.ly/MOO-api-migration.

the best performance in terms of precision and recall when using the CO scheme compared to using ALL and MS+DS. However, co-occurrence probabilities are difficult to compute particularly when the source and target libraries are new and fewer projects have migrated between them resulting in a lack of usage data. To examine if method mappings can be recommended in the absence of CO information, we evaluate an alternative similarity scheme consisting of combined method (MS) and documentation (DS) similarities. We also evaluate the ALL similarity scheme that combines CO, MS, and DS to examine if MS+DS information can improve source-target method

```
-  public java.lang.String toString();
+ public com.google.gson.JsonObject();
```

Figure 6.10: An example method mapping selected due to a high MS+DS score for the rule $json - simple \rightarrow gson$. The recommended target method is incorrect, as a String object and a JsonObject are incompatible. The '-' sign on the left denotes source library methods (in red) that have been removed while the '+' sign denotes target library methods (in green) that have been added.

```
-  public org.apache.log.Logger getLogger();
+ public static org.slf4j.Logger
getLogger(java.lang.Class);
```

Figure 6.11: An example mapping recommended by NSGAII for $logging \rightarrow slf4j$. While the source method does not accept any parameters, the target method accepts an argument of the type *java.lang.Class*. The method similarity score for this mapping is high, which leads to its recommendation, however, the co-occurrence score is 0.The '-' sign on the left denotes source library methods (in red) that have been removed while the '+' sign denotes target library methods (in green) that have been added.

```
-  public static <T> org.easymock.Capture<T> newCapture();
-  public static <T> org.easymock.IExpectationSetters<T> expect(T);
-  public abstract org.easymock.IExpectationSetters<T> andReturn(T);
+ public abstract <T> T given(T);
+ public abstract org.mockito.BDDMockito$BDDStubber willReturn(java.lang.Object);
```

```
-  public void assertEquals(java.util.Set<?>, java.util.Set<?>);
+ public static void assertEquals(java.lang.String, float, float, float);
```

Figure 6.12: An example mapping recommended for $testng \rightarrow junit$. Both source and target methods have similar definitions, apart from the return types and arguments, leading to a high MS+DS score. While this mapping does not exist in our groundtruth, the target method can be used as a replacement. The '-' sign on the left denotes source library methods (in red) that have been removed while the '+' sign denotes target library methods (in green) that have been added.

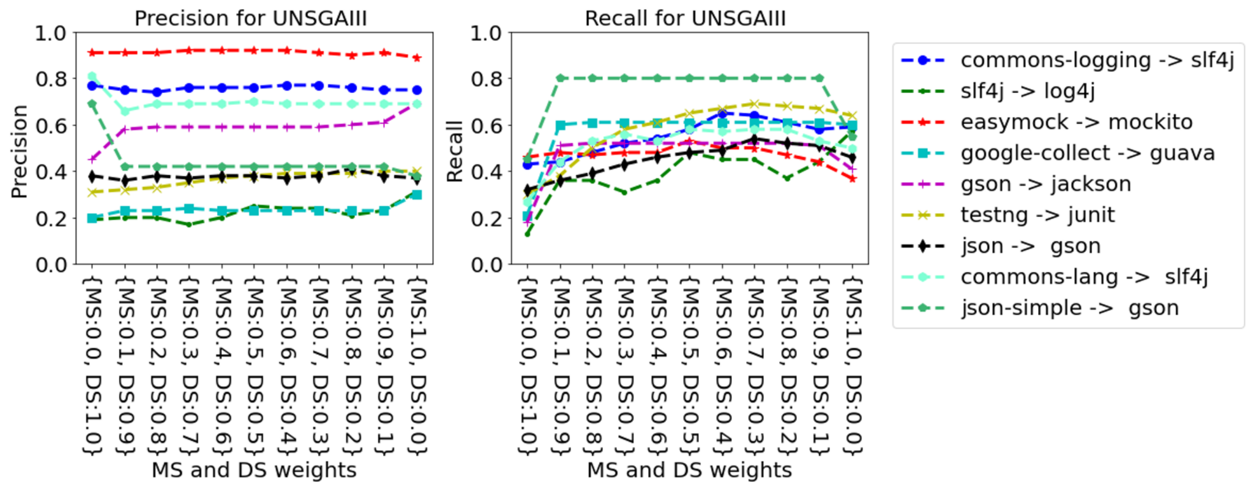


Figure 6.13: This figure illustrates the impact of using different weights for method (MS) and documentation (DS) similarities on UNSGAIII and IBEA precision and recall. We observe that each algorithm achieves high precision and recall for different weight combinations when considering method and documentation similarities.

recommendations.

From Tables 6.5 and 6.8, we observe that leveraging method and documentation similarity in the MS+DS scheme is useful, and can lead to good precision and recall when recommending source-target method mappings. However, all 7 algorithms do not achieve the highest precision and recall values when using the MS+DS scheme. This is because multi-objective algorithms may discover newer, *unobserved* mappings that are not assigned high fitness scores when using the CO scheme compared to the MS+DS scheme *e.g.,* as depicted in Figure 6.12. The CO scheme is derived from our ground truth, which only consists of mappings from our dataset that have been observed before. Moreover, the MS+DS scheme can assign higher fitness scores to similarly defined methods as a result of method and documentation similarity calculations leading to more mapping selections compared to the CO scheme. Figures 6.11 and 6.12 demonstrate that similarities in library designs and method naming conventions can lead to many more (correct) target methods being recommended that are absent in the ground truth leading to poor precision and recall. Additionally, using method similarity can be a source of noise because several source-target method mappings are assigned low non-zero fitness leading to random selections or false positives that negatively impact precision, such as in Figure 6.10. Method and documentation similarity can also be difficult to use for *many-to-many* mappings. Figure 6.12 depicts an example many-to-many mapping that consists of “helper” method calls that may not have equivalent replacements across both source and target libraries resulting in zero or close to zero similarity scores in the MS+DS scheme. As a result, search algorithms may not select these mappings. In such scenarios, only the CO scheme would accurately indicate whether specific "helper" methods are appropriate replacements. Thus, we make two observations: (a) method signature similarity (MS) results in accurate recommendations when two libraries have similar design conventions, and (b) using a combination of CO and MS+DS mappings can result in the discovery of source-target method mappings that have not been observed in past migrations.

In our previous experiments, we assigned equal weights to both method and documentation similarity. However, when source and target libraries are poorly documented or have different designs, variable weights should be assigned to documentation and method similarities to maximize available information about possible method mappings. From Figure 6.13, we observe that the highest values of precision and recall are obtained using different weight combinations for each migration rule *e.g.,* UNSGAIII achieves the highest recall using the {MS: 0.1, DS: 0.9} weight scheme for $json - simple \rightarrow gson$ (a 28% increase from Table 6.5), and achieves high recall when using the scheme {MS: 0.5, DS: 0.5} for $slf4j - api \rightarrow log4j$. We study the impact of different weight combinations for each migration rule as follows:

- When considering rules $logging \rightarrow slf4j$, $google - collect \rightarrow guava$ and $testng \rightarrow junit$, we observe that the precision curve does not vary a lot when considering different MS and DS weighing schemes. However, the recall curve improves considerably as method similarity is assigned a higher weight. These results suggest that assigning a higher weight to method signature similarity reduces the number of false negatives and that methods from the source and target libraries are similarly defined.
- $slf4j - api \rightarrow log4j$: While precision values are low when using UNSGAIII and MS+DS, both precision and recall trends increase as method similarities are assigned more weight. We also note that the median fitness score is closer to 0 for this rule suggesting that the mappings in our dataset do not have similar naming conventions or lack documentation.
- $easymock \rightarrow mockito$: While precision values are close to 100%, recall values are closer to 50% suggesting that there are a high number of false negatives. These libraries have similar naming conventions and designs, so the mean and median similarity scores are very close (within 1%) resulting in high precision but poor ability to identify negatives.
- $gson \rightarrow jackson$: For this migration rule, precision improves by approximately 10% when considering only method similarity, while recall decreases proportionately. So, documentation similarity results in more incorrect method mappings being chosen as compared to other rules.

- $json \rightarrow gson$: Similar to other migration rules recall steadily improves with an increase in MS weights. Moreover, the recall curve illustrates that DS similarity scores are important to correctly identify false negatives. So, for this migration rule, both MS and DS are required to accurately identify method mappings.

- For the rules $commons - lang \rightarrow slf4j - api$ and $json - simple \rightarrow gson$, assigning higher weights to documentation similarity results in fewer false positives leading to improved precision. Moreover, we observe that poor precision and recall are achieved when considering only method similarity, suggesting that mappings from these libraries do not have similar naming conventions.

We observe that differences in algorithm performance when using the MS+DS scheme are a result of dissimilar library designs, naming conventions, and documentation. While the MS+DS scheme achieves good results in terms of precision and recall, MS and DS weights must be carefully selected based on distribution statistics *e.g.,* if the mean documentation similarity is 0 and the standard deviation is low, then DS should be assigned a lower weight as compared to MS. We also note that the MS+DS scheme can discover method mappings that have not been used before, so it can be use in conjunction with the CO scheme. We summarize the strengths and weaknesses of using three similarity schemes in Table 6.12.

6.4 Limitations

- In our experiments, the best precision and recall values are generally observed when using the CO scheme, which relies on co-occurrence probabilities calculated from data generated by the developer community. This is similar to using training data to recommend suitable method mappings. Additionally, the USE network is not trained to handle a mix of code and natural language. As a result, we cannot include code examples that may be part of the documentation when calculating the DS score. To address these issues, in the future we will explore the use of large language models (LLMs) to recommend method mappings.

- In this work, we formulated API migration as an unconstrained, multi-objective search problem instead of a constrained single-objective problem [56]. However, single-objective algorithms are easier to use for libraries with more one-to-one method mappings because constraints may be easy to specify. In contrast, for many-to-many, one-to-many, or many-to-one mapping cardinalities, it is difficult to define constraint values for single-objective algorithms as we do not know beforehand how many target methods are required for replacement. So,

Table 6.12: A summary of the strengths and weaknesses of using each similarity scheme.

Scheme	Strengths	Weaknesses
CO	Can identify highly “dissimilar” mappings	Calculating CO requires historical data, so it may not work for newer libraries
	Can incorporate subtle best practices and mappings using “wisdom of the crowd”	Expensive to calculate since it requires train- ing data
MS+DS	Can recommend mappings when data is un- available *e.g.,* for newer libraries, and across different languages and platforms	Only works when both source and target library designs and naming conventions are similar
	Effective when library designs are similar and documentation is well-defined	Can result in lower similarity scores if libraries and methods do not have proper documentation
ALL	Incorporates MS, DS, and CO so it can still find mappings if one of the components is missing	Achieves lower precision and recall than CO. It may need trial and error to tune the weight for each component

our multi-objective approach should be used by developers to obtain recommendations on a wider range of mappings, particularly when mapping cardinalities are unknown.

- In this study, we found that search algorithms can also discover novel method mappings with high fitness scores that have not been observed before. In such cases, precision and recall values are low because our ground truth does not contain these mappings, leading to a large number of false positives and negatives. Additionally, these new mappings may need to be manually validated by developers, which results in a significant overhead. We note that such validation is necessary for real-world scenarios when leveraging API recommendation tools. In other terms, the precision and recall values achieved by each algorithm in our experimental evaluations may improve further after manual validation. Our experiments provide a baseline for algorithm performance based on the CO scheme and may underestimate the performance of our approach. However, this limitation will not exist in practice since any recommended source-target method mappings will be manually validated by developers before incorporating them into existing software. Also, developers may not intend to change the behavior of the system when migrating between libraries, and therefore, functional testing can be used to detect any regressions introduced by newly introduced methods.

- In this work, we have specified both minimum and maximum bounds for hypervolume calculations. Incorrect specifications of these points can lead to erroneous hypervolume values and, thus, inaccurate conclusions. We have mitigated this threat by deriving the minimum and maximum for each objective from all generated Pareto fronts. Moreover, we also evaluate algorithms by calculating Euclidean distance from our groundtruth derived from manually validated method mappings. However, despite these precautions, due to factors such as newly found method mappings, these calculations could contain errors that are not possible to know beforehand. In practice, when using our approach, developers will be able to calculate hypervolume, but will not be able to calculate Euclidean distance as a groundtruth is unavailable.

6.5 Chapter Summary

We formulate third-party library migration at the method-level as a combinatorial optimization problem. We evaluated a single-objective genetic algorithm (GA) and 7 popular multi-objective algorithms - UNSGAIII, RNSGAII, AGEMOEA, SMSEMOA, NSGAII, IBEA and MOEAD - to recommend source-target method mappings across 9 library pairs. Our experiments demonstrate that (a) GA outperforms random search and hill-climbing, and (b) multi-objective optimization further improves recommendation quality to outperform RAPIM and GA in terms of precision and recall. Moreover, we examine the benefits of using three different similarity schemes - co-occurence probability, method and documentation similarity, and a combination of all three. Additionally, we make our code and results publicly available at: http://bit.ly/MOO-api-migration.

Chapter 7

Algorithm Selection for Search-Based Third-Party Library Migration at the Method-Level

Having demonstrated the positive impact of using multi-objective metaheuristic algorithms in Chapter 6, we now examine if these algorithms exhibit complementary performance on API migration instances. To reduce developer effort, our primary goal is to ensure that our approach recommends target APIs with high precision. Based on our results from the previous chapter, we observe that all algorithms do not uniformly achieve the highest precision, recall, HV and ED when recommending for all migration rules. That is, metaheuristic performance varies by migration rules and their underlying fitness distribution. In this chapter, we explore metaheuristic variability further and demonstrate the potential benefits of using an algorithm selection approach for API migration. To the best of our knowledge, the impact of algorithm selection for API migration has not been studied before.

7.1 Methodology

In this study, we identify the variability in algorithm behavior on different migration instances and demonstrate the *complementary performance* of search algorithms for API migration. To achieve our goal, we generate a dataset of 7,200 API migration instances and examine the variability of 5 metaheuristic algorithms. We generate a labeled dataset that is used to train classifiers to predict a suitable algorithm for each migration instance.

7.1.1 Dataset generation

Our previous experiments in Chapter 6 compared the performance of various metaheuristic algorithms on *one* set (instance) of randomly sampled method mappings for each migration rule. Using a single set of mappings for each migration rule allowed us to compare algorithms fairly without introducing variations in the underlying fitness distribution or *search space*. However, in this study, our goal is to explicitly study how algorithm performance varies for diverse search spaces. As a result, we randomly sample multiple sets of method mappings for each each. Specifically, we generate 400 random instances for each migration rule with varying numbers and cardinalities of method mappings. Furthermore, our sampling procedure includes both correct and incorrect method mappings so that we can test the precision and recall of each algorithm. The correctness of a mapping is determined by checking whether a specific source-target method pair exists in our manually curated dataset of Java mappings. In other terms, a source and target method pair is considered correct if it co-exists in our dataset of mappings scraped from $\tilde{5}7,000$ projects. Since the CO scheme also calculates fitness based on the co-occurrence of source-target method pairs, all algorithms achieve the highest precision, recall, ED, and HV when using the CO scheme. So, we use the CO scheme to study the variability in algorithm performance because it provides the most accurate assessment. We also examine the impact of algorithm selection using the MS+DS scheme (method and documentation similarity). This configuration results in a dataset of $400 \times 9 \times 2 = 7,200$ different migration instances with varied search landscapes.

7.1.2 Metrics Used

In addition to the metrics used to study different metaheuristic algorithms in the previous chapter, we record the execution time (in seconds) for each algorithm. We present a brief overview of all the metrics used in this study as follows:

- **Runtime**: In search-based API migration, our goal is to build automated tools that recommend fewer, and correct API mappings when performing software library migration. Developer tools are frequently incorporated in integrated development environments (IDEs) to facilitate real-time recommendations. As a result, we aim to recommend mappings accurately and efficiently so that developers can incorporate suggested code changes during programming. So, we record the time in seconds required by each algorithm to recommend suitable mappings.

- **Precision**: Precision measures the ratio of correctly identified method mappings to the number of recommended mappings (true and false positives).
- **Recall**: Recall measures the ratio of correct (true positives) and incorrect (false negatives) mappings that were accurately identified.
- **Euclidean Distance (ED)**: The Euclidean distance indicator calculates the distance of a solution from the 'groundtruth'. The 'groundtruth' is calculated by referencing our dataset of $\tilde{5}$7,000 manually curated mappings. That is, if a source-target method pair exists in our dataset of Java mappings, it is correct, otherwise it is considered incorrect. As a result, this metric generally has higher values when using the co-occurrence (CO) scheme since similarity is calculated using co-occurrence in our dataset.
- **Hypervolume (HV)**: The hypervolume indicator evaluates the quality of the entire set of generated solutions. It is calculated using the best and worst possible values for each objective (here, the number and fitness of recommended mappings). Similar to ED calculations, the best and worst objective values are also informed by our $\tilde{5}$7,000 Java method mappings.

7.2 Algorithms Used

Based on our results from Chapter 6, we study the variability of UNSGAIII, RNSGAII, NSGAII, AGEMOEA and SMSEMOA on the instances from our dataset. MOEAD is excluded from this study due to its relatively poor precision for API migration as noted in Chapter 6. Furthermore, we leverage the PyMOO framework that contains optimized implementations of these algorithms for accurate comparison. So, IBEA is also excluded as it is not implemented in PyMOO, and using a different framework could lead to an unfair comparison particularly when considering runtime.

7.2.1 Features considered

We begin by characterizing each API migration instance that serve as inputs to our classifiers that predict a suitable algorithm. Each migration instance is characterized using a pre-determined set of features that provide essential context. We adapt features from our previous work on service composition in Chapter 4 to API migration and include information about the fitness distribution for each API migration instance. The features are:

1. **Characterizing the size of each migration instance**: We record the total number of source and target methods considered in each migration instance. As each migration instance

consists of randomly generated source-target mappings, we study how algorithm performance changes with search space size.

2. **Computational resources**: We record the execution time (in seconds) for each algorithm are recorded, with the goal of providing accurate and efficient API mappings to developers.

Additionally, to accurately predict computational resource usage and characterize the fitness distribution for each migration instance, we adapt node characteristic features from [75]. These features characterize the search space an algorithm must explore and its complexity.

1. **Characterizing each source method's fitness distribution**: For each source library method that needs to be replaced, we characterize the fitness distribution of potential target methods that can be used for replacement. More specifically, we measure the mean, variation coefficient and skew fitness value for each potential target library method that can be used for replacement. For example, consider we need to replace 5 source methods using one of 5 target methods for each source method. So, for each source method we will record the mean, variation and skew for the 5 target methods that can potentially be used for replacement.

2. **Centroids of fitness distribution**: In addition to the characteristics above, we also calculate the centroid of the fitness distribution for all target methods being considered to replace each source method.

3. We also record the **worst** fitness value for each set of target methods that can be used to replace different source methods. These features, in addition to the ones above, provide insights about the search space in which different metaheuristic algorithms search for solutions [76].

7.2.2 Algorithm Selection Optimization Function

Next, we formulate the optimization problem governing the precision and runtime trade-off when leveraging algorithm selection for API migration. We assume that there is a pre-determined set of *complementary* metaheuristic algorithms. That is, different algorithms in our set outperform each other on specific migration instances [75, 76]. A suitable algorithm is selected from a subset denoted as $CASet = \{CASet_0, CASet_1, \ldots CASet_j\}$, wherein each metaheuristic algorithm computes solutions with high precision using variable time $time_j$ resources. Each algorithm in $CASet$ must achieve high precision, that is, we only consider those algorithms that achieve precision within 5% of the best precision achieved by all algorithms in $ASet$. We use 5% as a constraint because all algorithms achieve similarly high precision values. In fact, even our SBS (UNSGAIII) achieves up to 4% less precision on average compared to the best algorithm. By doing so, we ensure that all selected algorithms generate high-quality solutions while also considering the execution time of different algorithms. Our goal is to select an algorithm that recommends mappings with high precision while minimizing execution time.

$$\text{argmin}_{j \epsilon CASet} \ \min(time_j) \tag{7.1}$$
$$where\ CASet = \{algo\ \epsilon\ ASet \mid |prec_{algo} - M| \leq 0.05M\} \tag{7.2}$$
$$where\ M = \max(\{prec_k \mid k \epsilon ASet\}) \tag{7.3}$$
$$and\ ASet = \{\text{UNSGAIII, RNSGAII, NSGAII, AGEMOEA, SMSEMOA}\} \tag{7.4}$$

This optimization problem is solved for each migration instance that is associated with the fitness features that we described above. We record the precision achieved and time $time_j$ required by each algorithm to generate high-quality solutions. The best suited algorithm $j \epsilon CASet$ achieves high precision $\mid prec_{algo} \ - \ M \mid \ \leq 0.05M$ while using the least amount of $time_j$. We note here that achieving high precision is a *necessary* but not sufficient condition to be considered for selection.

7.2.3 Classifiers Evaluated

In this work, we leverage classifiers to predict a suitable algorithm for each migration instance. We use classification techniques to learn decision boundaries for migration instances and correctly select algorithms for similar migration instances. Each migration instance is assigned a label corresponding to different algorithms used in this study. We compare the precision achieved and time used by each algorithm and use the algorithm selection optimization function to guide our labeling process. A classification approach allows us to incorporate human input and learn decision boundaries according to specific preferences. We train each classifier using a subset of our dataset, corresponding to previously solved migration instances. The algorithm that can recommend method mappings with high precision while using the least amount of time is the most suitable algorithm for a specific migration instance.

We use features similar to those in Chapter 4 to characterize each migration instance and study the variability of algorithm performance on 7,200 API migration instances. We focus on features that characterize the search space to improve accuracy as described earlier. So, we evaluate the performance of classifiers on our expanded dataset and demonstrate the benefits of using these features when profiling algorithm performance.

7.2.4 Comparative Approaches

We compare our algorithm selection-based API migration approach to two baselines from the algorithm selection literature described in Chapter 4. We restate these baselines for ease of reference as follows:

- **Single Best Selector (SBS):** The single best selector corresponds to using a single algorithm for all migration instances. For any algorithm selection approach to be considered useful, the SBS must be outperformed. In this study, we set the SBS to UNSGAIII as it achieves high precision and recall compared to other algorithms.
- **Virtual Best Selector (VBS):** The virtual best selector corresponds to a perfect selector that always selects the 'right' algorithm for each problem instance. In API migration, the 'right' algorithm recommends mappings with high precision and minimizes runtime. The VBS corresponds to the labels in our API migration dataset.

7.3 Results

Our goal is to demonstrate select a suitable algorithm from a complementary set for each API migration instance such that we minimize runtime while recommending source-target method mappings. We evaluate the efficacy of our approach using two research questions:

- **RQ 10:** How does UNSGAIII, RNSGAII, NSGAII, AGEMOEA and SMSEMOA performance vary across migration instances with different sizes and fitness distributions? *We demonstrate that UNSGAIII, RNSGAII, NSGAII, AGEMOEA and SMSEMOA exhibit complementary performance on various API migration instances.*
- **RQ 11:** What is the impact of selecting a different metaheuristic algorithm for specific migration instances? *By selecting different algorithms for specific API migration instances, we recommend method mappings with high precision and reduce time usage by 12.5%.*

7.3.1 RQ 10: Variability of Search Algorithms for API Migration

Table 7.1 depicts the mean precision, recall, ED, HV achieved by each of the algorithms used in our study. Additionally, we also record the execution time of each algorithm. We report the mean metrics for all algorithms using CO and MS+DS schemes. We observe that all algorithms

Table 7.1: Comparing the mean metrics achieved by UNSGAIII, RNSGAII, NSGAII, AGEMOEA, SMSEMOA using the CO and MS+DS similarity schemes. All algorithms achieve relatively hgh precision and similar ED and HV values. RNSGAII achieves comparatively higher ED and HV values, and we observe that there is variability in runtime performance.

Algorithm	**Precision**	**Recall**	**Runtime (s)**	**ED**	**HV**
UNSGAIII	0.85	0.91	513.22	0.12	0.42
RNSGAII	0.80	0.48	500.13	0.31	0.66
NSGAII	0.80	0.61	494.35	0.18	0.56
AGEMOEA	0.80	0.63	540.25	0.16	0.52
SMSEMOA	0.81	0.62	557.79	0.20	0.52

achieve similarly high levels of precision and HV, and low ED values across all migration rules. RNSGAII is an exception because it achieves high precision and high ED, with relatively poor recall. Additionally, we observe differences in algorithm runtime despite similarities in precision and some recall values.

Based on these observations, we generate labels for our dataset using the optimization function discussed in Section 7.2.2. Table 7.3 depicts the label distribution of our dataset. For each migration instance in our dataset, we first generate a list of algorithms that recommend method mappings within 5% of the best possible precision. We then label the instance with the algorithm that recommends mappings using the least amount of time. From Table 7.3, we observe that NSGAII and UNSGAIII are the most frequently selected algorithms, followed by RNSGAII. However, AGEMOEA and SMSEMOA are selected for considerably fewer instances. In fact, SMSEMOA and AGEMOEA are selected for only 3.4% and 6.3% of the instances in our dataset. As a result, we have an imbalanced dataset that may lead to inaccurate selections for classes that have fewer datapoints. We describe the classifiers used and training routine used to train algorithm selectors in the next subsection.

We study algorithm performance further using our labeled dataset and the feature set outlined in 7.2.1. In particular, we analyze features of migration instances labeled for each algorithm and make the following observations:

- We observe that UNSGAIII is selected uniformly for all migration rules, and NSGAII is selected more for $slf4j{-}api \rightarrow log4j$, $google{-}collect \rightarrow guava$, $commons{-}lang \rightarrow slf4j{-}api$ and $json - simple \rightarrow gson$. In contrast, RNSGAII is rarely used for $google - collect \rightarrow guava$ and AGEMOEA is selected at a relatively lower rate at $slf4j{-}api \rightarrow log4j$, $google{-}collect \rightarrow$

$guava$ and $json - simple \rightarrow gson$.

- We also note that AGEMOEA is usually selected for those migration instances with a larger number of method mappings, whereas SMSEMOA is selected for smaller search spaces. In general, the instances that SMSEMOA solves efficiently have the highest fitness values with low variation coefficient and skew. Additionally, we also note that UNSGAIII is frequently selected for those migration instances with a smaller number of mappings. Additionally, NSGAII and RNSGAII are selected uniformly across different search space sizes.

- We observed that UNSGAIII and AGEMOEA have the lowest mean fitness compared to RNSGAII, SMSEMOA and NSGAII. Additionally, instances labeled for UNSGAIII have the second lowest mean variation coefficient suggesting that the fitness values for these instances do not vary as much as for the other algorithms. That is, the fitness landscape is comparatively more uniform for those instances labeled as UNSGAIII. In contrast, AGEMOEA instances have the highest variation coefficient values, followed by RNSGAII and NSGAII.

- The mean fitness skew for AGEMOEA is highest, followed by RNSGAII, then NSGAII and UNSGAIII with similar values, and SMSEMOA has the lowest skew fitness. This suggests that AGEMOEA fitness landscapes are relatively more complex compared to SMSEMOA, and UNSGAIII.

Table 7.2 and Figures 7.1 and 7.2 illustrate the potential for using an algorithm selection approach. Figure 7.2 depicts the precision achieved when using the SBS and VBS on a log scale. We observe that all points are close to the diagonal meaning that both approaches achieve high precision and there is very little variation precision achieved by the SBS and VBS. In contrast, we observe from Figure 7.1 that a considerable portion of migration instances (in the bottom left corner) solved by the SBS use more time resources compared to the VBS selections as they are positioned further away from the diagonal. As a result, we can reduce time resource usage with little or no precision loss using an algorithm selection approach.

Based on our feature analysis and observations about SBS and VBS performance, we conclude that UNSGAIII, RNSGAII, NSGAII, AGEMOEA and SMSEMOA exhibit *complementary performance*. That is, different algorithms should be selected for specific API migration instances. In the next subsection, we evaluate the benefits of an algorithm selection approach for API migration.

Table 7.2: Comparing SBS, algorithm selection and VBS precision, recall, ED, HV and runtime using the CO similarity scheme. This table presents the mean values for each metric. All approaches achieve similarly high precision and there is a difference of approximately 25 seconds between SBS and VBS performance.

Method	Precision	Recall	Runtime (s)	ED	HV
VBS	0.87	0.71	489.89	0.20	0.59
AlgoSelect	0.86	0.73	495.37	0.19	0.56
SBS	0.85	0.91	515.29	0.12	0.42

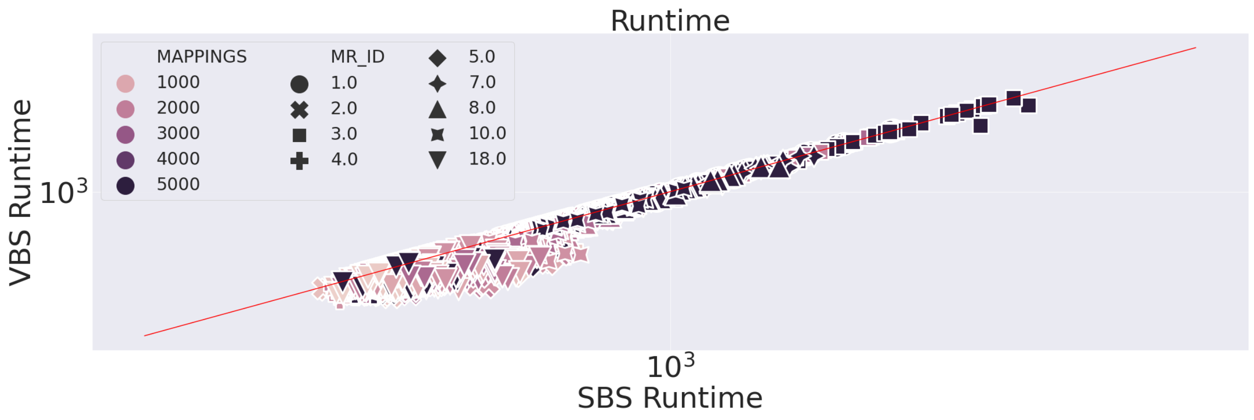


Figure 7.1: The gap between execution time using SBS and VBS selections is considerable. VBS can recommend source-target method mappings fasteras compared to SBS.

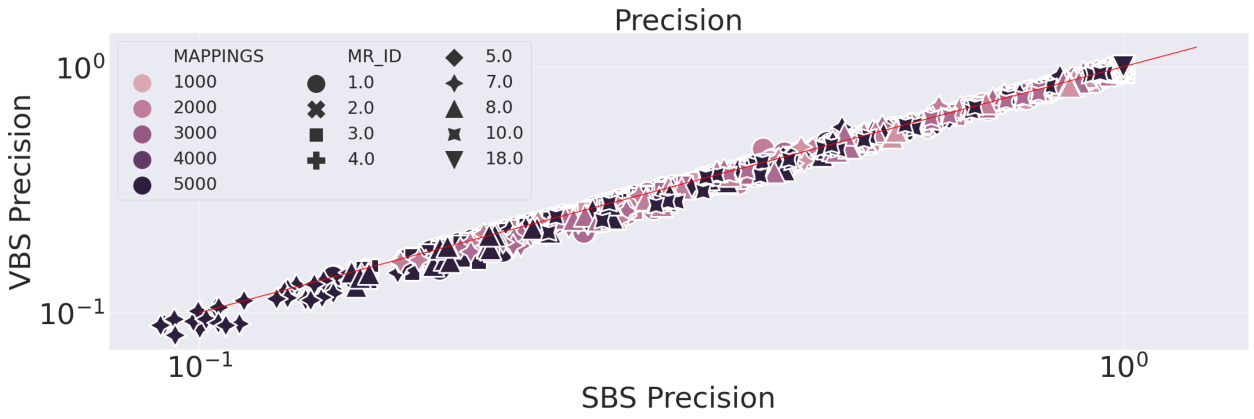


Figure 7.2: The gap in precision achieved by the SBS and VBS is negligible. Both the SBS and VBS can recommend source-target method mappings with the high precision

Table 7.3: Algorithm label distribution for 7,200 API migration instances. Each instance is labeled with the algorithm that recommends method mappings with high precision while minimizing time usage.

Algorithm	**Labels**
NSGAII	2330
UNSGAIII	2286
RNSGAII	1885
AGEMOEA	456
SMSEMOA	243

7.3.2 RQ 11: Algorithm Selection for API Migration

Table 7.4 illustrates that a random forest classifier achieves the highest accuracy and F1-score when predicting search algorithms for migration instances. We repeat 5 fold cross-validation 10 times and report the average. Additionally, we use an 80/20 train-test split and perform a grid search to determine the best set of hyperparameters for each classifier. From Table 7.4, we observe that random forest and decision tree achieve the highest accuracy, followed by k nearest neighbors. These findings suggest that the decision boundaries between classes are highly non-linear, indicating a complex relationship between our features and labels.

Figure 7.3 depicts the efficacy of our trained random forest classifier when selecting an algorithm for each migration instance. We observe that a majority of UNSGAIII and NSGAII instances are correctly identified. However, AGEMOEA, RNSGAII and SMSEMOA instances are frequently misclassified. We tested a one-vs-one training routine to assess if it improves random forest accuracy particularly for classifying AGEMOEA and SNSEMOA instances. However, we did not observe a significant increase in accuracy, precision, recall or F1-score values.

Figures 7.5, 7.4 and 7.6 illustrate the loss of precision when using our random forest classifier and the time used by SBS, random forest selected algorithms and the VBS to find suitable method mappings. We evaluate the time required by each approach in Figure 7.4 where the SBS and VBS plots have noticeably different shapes, highlighting the potential for improvement. From Table 7.5, we note that the mean computation time is reduced by 19.9 seconds and the median time by 20.2 seconds without considerable loss in precision. In fact, based on the plot in From Figure 7.5, we note negligible precision loss when using our algorithm selection approach. We also evaluate the time reduced using algorithm selection on test instances that were not labeled as UNSGAIII. Such an analysis allows us to assess if more time is saved on instances best solved by other algorithms that are not the SBS. Figure 7.6 illustrates the benefits of using algorithm selection on instances

Table 7.4: We compare classifier accuracies and observe that a random forest classifier selects algorithms that recommend mappings with high precision and minimize time.

Classifier	**Accuracy**	**F1-score**	**Precision**	**Recall**
Random Forest	0.61	0.6	0.6	0.61
SVM - rbf	0.5	0.48	0.48	0.5
MLP	0.51	0.48	0.48	0.51
Decision Tree	0.58	0.57	0.57	0.58
kNN	0.53	0.51	0.52	0.53
Logistic Regression	0.49	0.47	0.46	0.49
SVM Sigmoid	0.45	0.43	0.44	0.45
Naive Bayes	0.16	0.16	0.29	0.16
QDA	0.4	0.37	0.44	0.4

Table 7.5: Using a random forest classifier to select evolutionary algorithms reduces recommendation time while suggesting mappings with high precision

Method	**Mean Precision**	**Mean Time(s)**	**Median Precision**	**Median Time (s)**
VBS	0.87	489.89	1.0	457.65
Algo Select	0.86	495.37	1.0	462.66
SBS	0.85	515.29	1.0	482

best solved by algorithms other than the SBS. On non-SBS instances, the mean time reduced is 61.95 seconds and the median time reduced is 55.48 seconds. In other terms, time savings are considerably higher when an algorithm other than the SBS needs to be selected for API migration instances. So, we conclude that algorithm selection can reduce mean time usage by up to 12.5% for all API migration instances while achieving high precision.

7.4 Discussion and Future Work

- In this work, we generate 7,200 migration instances by randomly sampling a manually curated dataset of Java mappings for 9 migration rules. However, the range of search spaces evaluated in our API migration work is not as extensive as our service composition dataset. As a result, the benefits of our approach have not been tested on other programming languages (*e.g.,* python) and migration rules with more source and target library methods. In the future, we will evaluate our approach for other programming languages and frameworks to generate varied search spaces with more mappings and migration rules.

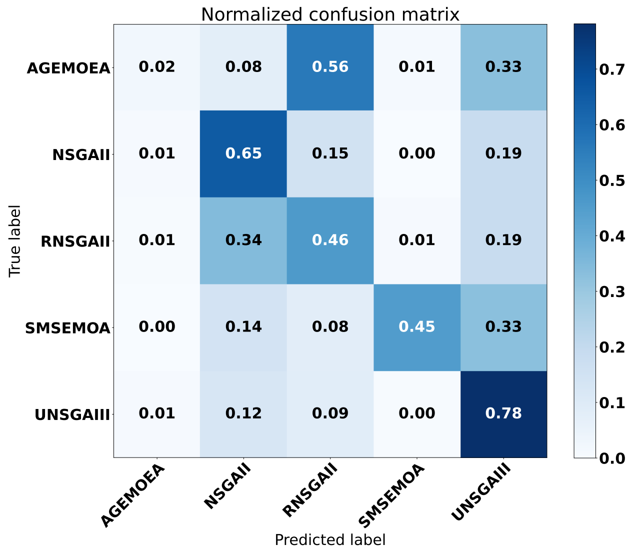


Figure 7.3: A random forest classifier achieves 61% accuracy. However, it frequently mis-classifies AGEMOEA and RNSGAII instances.

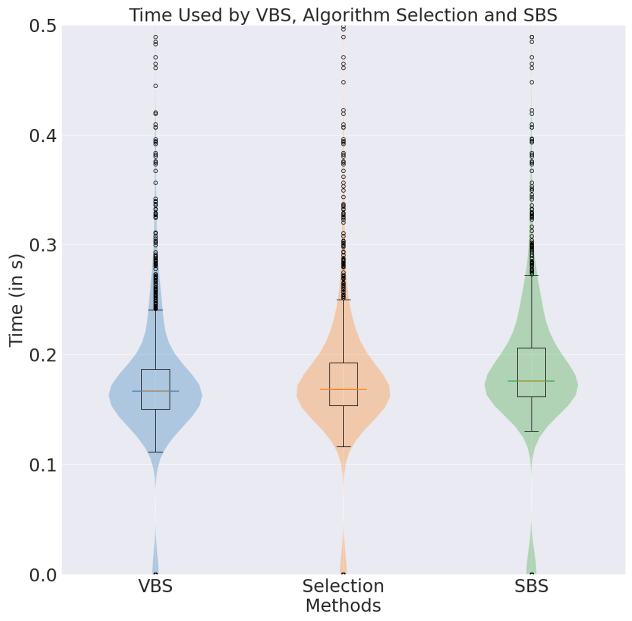


Figure 7.4: Both VBS and algorithm selection approaches reduce the time taken by each algorithm to recommend API mappings. Note that this graph is generated on our test set, which is 20% of our training data.

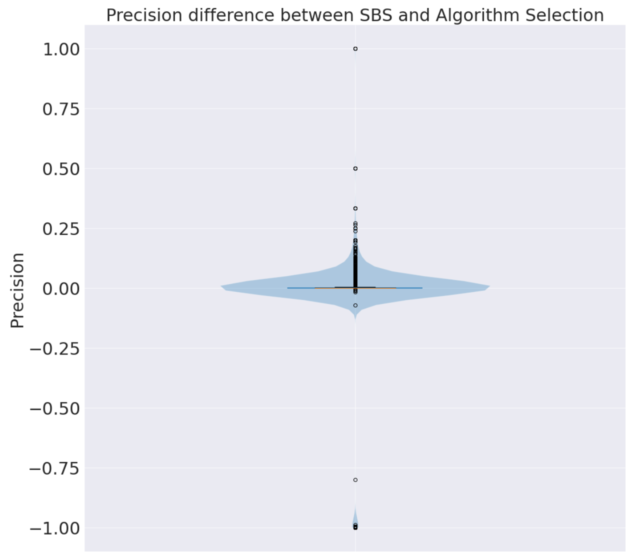


Figure 7.5: Mean precision loss when using algorithm selection is 0%. This figure depicts the distribution of precision loss over our test instances, which is minimal.

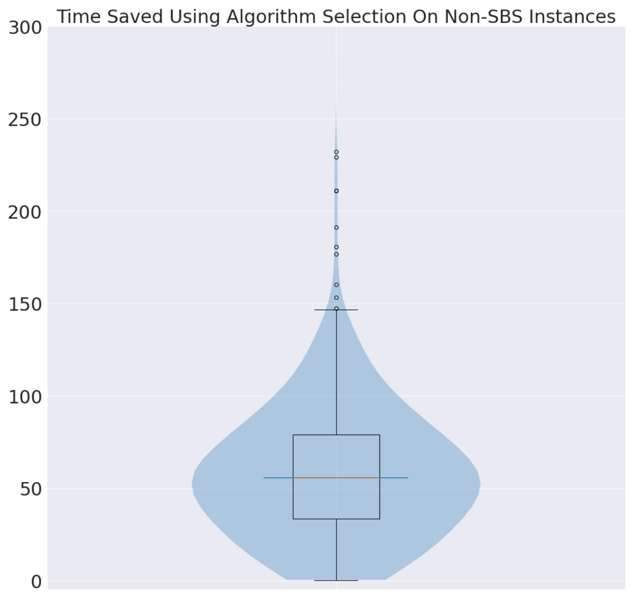


Figure 7.6: 52% of our test instances use the same algorithm as the SBS. On the other 42% instances that use a different algorithm, the mean time saved is considerably higher at 61 seconds.

- We leverage evolutionary algorithms to recommend *one-to-one, one-to-many, many-to-one* and *many-to-many* method mappings. More specifically, we recommend replacements for individual API calls and do not consider additional changes that may need to be made to the surrounding code *e.g.,* the addition of 'helper' methods to process the output. In the future, we plan to leverage Large Language Models (LLMs) to expand the scope of our approach and recommend changes to the entire source code snippet during library migration.
- We observed that ED and HV indicators were correlated to the precision achieved by each evolutionary algorithm. As a result, we consider precision and runtime when recommending suitable algorithms for API migration instances. However, there may be other indicators that can be used to assess the quality of generated solutions without referencing the 'groundtruth'. In the future, we plan on including indicators to measure convergence and solution quality when recommending algorithms for API migration.

7.5 Chapter Summary

In this chapter, we demonstrated that UNSGAIII, RNSGAII, NSGAII, AGEMOEA, and SMSEMOA exhibit *complementary performance* and are preferred for specific API migration instances. More specifically, each algorithm achieves high precision and requires variable time resources to recommend source-target method mappings. We generate a dataset of 7,200 API migration instances by randomly sampling a popular dataset consisting of manually curated mappings from 9 popular library migrations collected from 57,447 open-source Java projects [12]. Our experimental evaluations demonstrate that our approach reduces time usage by up to 12.5% while recommending method mappings with high precision.

Chapter 8

Summary

8.1 Research Contributions

Search-based software engineering (SBSE) commonly employs heuristic algorithms to accurately and efficiently evaluate numerous software configurations and find solutions that fulfill user requirements. Assessing different configurations is challenging because solutions must balance multiple conflicting functional (*e.g.,* accuracy) and non-functional (*e.g.,* execution time) objectives. To address these challenges, current approaches have leveraged various heuristic algorithms to solve diverse software engineering problems such as test case generation, web service composition, evaluating architectural configurations etc. Each heuristic algorithm provides near-optimal solution quality guarantees and uses variable amounts of computational resources to find suitable solutions. As a result, it is difficult to determine which heuristic algorithm to use for a software engineering problem.

Recent work in algorithm selection has demonstrated that different algorithms outperform each other on specific instances of a problem, that is, heuristic algorithms exhibit complementary performance in terms of solution quality and computational resource usage. However, current SBSE approaches do not leverage this complementary performance property, and use a *single* heuristic algorithm to solve all instances of a problem resulting in several inefficient solutions in terms of computational resource usage and solution quality. In this thesis, our *goal is to assist practitioners in building and maintaining effective and efficient software by providing automated, fine-grained search algorithm recommendations when employing SBSE techniques.* We demonstrate that by selecting a different algorithm from a complementary set for each problem instance, our algorithm selection-based approach can considerably reduce computational resource usage while delivering high-quality solutions.

We demonstrate the benefits of leveraging algorithm selection for two software engineering problems: web service composition and third-party software library migration. Our work on web service composition builds on existing work to propose an algorithm selection framework, R-CASS, to fulfill non-functional QoS requirements by re-selecting web services at runtime. To evaluate the efficacy of our approach, we created a dataset of 6,144 service composition instances and demonstrated the complementary performance of 4 popular service composition algorithms. We leverage classifiers and contextual multi-armed bandits to predict algorithm performance with up to 74% accuracy. Our experimental evaluations demonstrate that our approach reduces composition time and memory by up to 55% and 37.5% while fulfilling QoS requirements.

In contrast to service composition, heuristic algorithms have not been used to address API migration before. As a result, our first study on API migration analyzes the impact of using evolutionary algorithms for API migration on a popular dataset of 57,447 manually curated method mappings from 9 popular Java library pairs on GitHub. Our experimental evaluations show that leveraging multi-objective algorithms is beneficial in recommending APIs with up to 94% precision and 83% recall. Our subsequent study examines the variability of 5 multi-objective evolutionary algorithms on 7,200 randomly sampled API migration instances. We find that heuristic algorithms vary in terms of execution time and recommend API mappings with high precision. Our approach uses classifiers to select different evolutionary algorithms for specific API migration instances and reduces recommendation time by up to 12.5% while achieving high recommendation precision. We summarize the contributions of our work as follows:

- **Contribution to SBSE:** We developed fine-grained algorithm recommendation tools for two search-based software engineering applications. Specifically, we leverage algorithm selection to recommend different algorithms for specific problem instances. Our approach reduces computational resource usage while delivering high-quality solutions.
- **Contribution to Algorithm Selection:** Our work also contributes to algorithm selection and evolutionary computation literature. We study the variability of algorithm performance and detail the challenges of leveraging evolutionary algorithms for two SBSE applications.
- **Contribution to Service-Oriented Systems:** Our work on self-adaptive service-oriented systems leverages algorithm selection as an adaptation tactic to efficiently fulfill user requirements at runtime. Using our algorithm selection-based approach, we are able to reduce

composition time and memory significantly while fulfilling QoS requirements. Additionally, we also make our dataset publicly available to promote further research in this area.

- **Contribution to API Migration:** Our first contribution to API migration is devising a generalizable evolutionary algorithm-based approach to recommend method mappings with high precision. Notably, our approach can recommend algorithms across platforms and programming languages, and does not require extensive training routines. Our second contribution is to study the impact of selecting different algorithms for various API migration instances. In particular, we leverage classifiers to select algorithms for a large dataset of 7,200 migration instances. By harnessing the complementary strengths of different algorithms, our approach recommends method mappings with high precision and reduces execution time by up to 12.5%.

8.2 Future Work

Numerous heuristic approaches with variable solution quality guarantees have been proposed in the SBSE literature. Coupled with the variability in heuristic algorithm performance, determining the right heuristic algorithms for a specific SE problem is a challenge for software practitioners. Our work aims to mitigate this challenge using algorithm selection to assist practitioners in constructing efficient and effective software systems with multiple conflicting requirements. We note that determining the right heuristic algorithms for SBSE problems is a challenging, interdisciplinary problem that spans software engineering, optimization and AI. Although our work has taken the first steps to address this challenge, we hope that future work continues to build on our research. Some promising directions for future research include but are not limited to:

- **Algorithm Selection as an Adaptation Tactic:** In R-CASS, we dynamically adapt service-oriented applications by using different algorithms to select web services at runtime. In the future, we plan to evaluate the impact of using our framework for other cyber-physical systems *e.g.,* in robot coordination. A significant challenge when adopting heuristic algorithm selection is verifying and validating that recommended algorithms always find suitable configurations, particularly for critical applications. However, the inherent stochasticity of heuristic algorithms may make it difficult to prove theoretical bounds. In the future, we plan to study verification techniques for algorithm selection as an adaptation strategy.

- **Combining Heuristic Search with LLMs:** Large Language Models (LLMs) are powerful tools that can process text, code and image data for varied natural language processing tasks. More specifically, these tools can be used to understand code-related tasks such as API migration. Since LLMs are trained on large datasets of previously seen migrations, they can be used to accurately recommend target APIs across programming languages and frameworks. We note that our work on API migration was conceptualized before LLMs were widely available. So, in the future, we plan to explore how our heuristic algorithm-based approach can benefit from using LLMs particularly for API migration.

- **Automated Algorithm Design:** Heuristic algorithm performance can vary considerably due to specific hyperparameter values [22]. In our work on web service composition, we did not have to perform hyperparameter tuning because previous work had already tuned each composition algorithm used in our study. However, for API migration, we expended considerable effort to tune each evolutionary algorithm. However, predicting the best hyperparameters for each instance is an open research challenge. Recent work [22, 23] has proposed the use of data-driven techniques to derive algorithm performance guarantees for problem domains. In the future, we plan to leverage this work for the joint selection of algorithms and their hyperparameters to ensure the best search performance.

Bibliography

[1] Google cloud.

[2] Netflix technology blog.

[3] *Basic Concepts, Evolutionary Algorithms for Solving Multi-Objective Problems: Second Edition*, pages 1–60. Springer US, Boston, MA, 2007.

[4] H. Al-Helal and R. Gamble. Introducing replaceability into web service composition. *IEEE Transactions on Services Computing*, 7(2):198–209, April 2014.

[5] E. Al-Masri and Q. H. Mahmoud. Qos-based discovery and ranking of web services. In *2007 16th International Conference on Computer Communications and Networks*, pages 529–534, Aug 2007.

[6] Richardson Alexandre, Ali Ouni, Mohamed Aymen Saied, Salah Bouktif, and Mohamed Wiem Mkaouer. On the identification of third-party library usage patterns for android applications. EASE '22, page 255–259, New York, NY, USA, 2022. Association for Computing Machinery.

[7] N. Ali and C. Solis. Self-adaptation to mobile resources in service oriented architecture. In *2015 IEEE International Conference on Mobile Services*, pages 407–414, June 2015.

[8] Shaukat Ali, Paolo Arcaini, Dipesh Pradhan, Safdar Aqeel Safdar, and Tao Yue. Quality indicators in search-based software engineering: An empirical evaluation. *ACM Trans. Softw. Eng. Methodol.*, 29(2), mar 2020.

[9] Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. *Mining and Managing Big Data Refactoring for Design Improvement: Are We There Yet?* 02 2021.

[10] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th International Conference*

on World Wide Web, WWW '09, page 881–890, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *CASCON*, pages 60–71, 2018.

[12] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the API level. *Applied Soft Computing*, 90:106140, 2020.

[13] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party Java library migration at the method level. In *2019 International Conference on Program Comprehension.*

[14] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source Java libraries. In *Software Engineering for Variability Intensive Systems*, pages 295–320. Auerbach Publications, 2019.

[15] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. Software testing research challenges: An industrial perspective. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2023.

[16] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery.

[17] D. Ardagna, L. Baresi, S. Comai, M. Comuzzi, and B. Pernici. A service-based framework for flexible business processes. *IEEE Software*, 28(2):61–67, 2011.

[18] D. Ardagna and B. Pernici. Global and local qos constraints guarantee in web service selection. In *IEEE International Conference on Web Services (ICWS'05)*, page 806, 2005.

[19] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, 2007.

[20] Danilo Ardagna and Raffaela Mirandola. Per-flow optimal service selection for web services based processes. *Journal of Systems and Software*, 83(8):1512 – 1523, 2010. Performance Evaluation and Optimization of Ubiquitous Computing and Networked Systems.

[21] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning, 2023.

[22] Maria-Florina Balcan. Data-driven algorithm design. *CoRR*, abs/2011.07177, 2020.

[23] Maria-Florina Balcan, Dan DeBlasio, Travis Dick, Carl Kingsford, Tuomas Sandholm, and Ellen Vitercik. How much data is sufficient to learn high-performing algorithms? generalization guarantees for data-driven algorithm design. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 919–932, New York, NY, USA, 2021. Association for Computing Machinery.

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

[25] Leonardo C. T. Bezerra, Manuel López-Ibáñez, and Thomas Stützle. Automatically Designing State-of-the-Art Multi- and Many-Objective Evolutionary Algorithms. *Evolutionary Computation*, 28(2):195–226, 06 2020.

[26] Ying Bi, Bing Xue, Pablo Mesejo, Stefano Cagnoni, and Mengjie Zhang. A survey on evolutionary computation for computer vision and image analysis: Past, present, and future trends. *IEEE Transactions on Evolutionary Computation*, 27(1):5–25, 2022.

[27] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[28] Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-Oriented programming systems, languages, and applications*, pages 506–507, 2006.

[29] Athman Bouguettaya, Munindar Singh, Michael Huhns, Quan Z. Sheng, Hai Dong, Qi Yu, Azadeh Ghari Neiat, Sajib Mistry, Boualem Benatallah, Brahim Medjahed, Mourad Ouzzani, Fabio Casati, Xumin Liu, Hongbing Wang, Dimitrios Georgakopoulos, Liang Chen, Surya Nepal, Zaki Malik, Abdelkarim Erradi, Yan Wang, Brian Blake, Schahram Dustdar, Frank Leymann, and Michael Papazoglou. A service computing manifesto: The next 10 years. *Commun. ACM*, 60(4):64–72, March 2017.

[30] Dimo Brockhoff. Gecco 2018 tutorial on evolutionary multiobjective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, page 349–372, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Nghi Bui. Towards zero knowledge learning for cross language API mappings. In *International Conference on Software Engineering: Companion Proceedings*, 2019.

[32] Thomas H. W. Bäck, Anna V. Kononova, Bas van Stein, Hao Wang, Kirill A. Antonov, Roman T. Kalkreuth, Jacob de Nobel, Diederick Vermetten, Roy de Winter, and Furong Ye. Evolutionary Algorithms for Parameter Optimization—Thirty Years Later. *Evolutionary Computation*, 31(2):81–122, 06 2023.

[33] Xingjuan Cai, Shaojin Geng, Di Wu, and Jinjun Chen. Unified integration of many-objective optimization algorithm based on temporary offspring for software defects prediction. *Swarm and Evolutionary Computation*, 63:100871, 2021.

[34] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, May 2011.

[35] Felipe Campelo and Claus Aranha. Lessons from the Evolutionary Computation Bestiary. *Artificial Life*, 29(4):421–432, 11 2023.

[36] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, page 1069–1075, New York, NY, USA, 2005. Association for Computing Machinery.

[37] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. Moses: A platform for experimenting with qos-driven self-adaptation policies for service oriented systems. In Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, editors, *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 409–433, Cham, 2017. Springer International Publishing.

[38] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. *Towards Self-adaptation for Dependable Service-Oriented Systems*, pages 24–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[39] Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder, 2018.

[40] Chunyang Chen. SimilarAPI: Mining analogical APIs for library migration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 37–40, 2020.

[41] Chunyang Chen. Similarapi: mining analogical APIs for library migration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 37–40, 2020.

[42] Jixiang Chen, Fu Luo, Genghui Li, and Zhenkun Wang. Batch bayesian optimization with adaptive batch acquisition functions via multi-objective optimization. *Swarm and Evolutionary Computation*, 79:101293, 2023.

[43] N. Chen, N. Cardozo, and S. Clarke. Goal-driven service composition in mobile and pervasive computing. *IEEE Transactions on Services Computing*, 11(1):49–62, 2018.

[44] Tao Chen, Miqing Li, Ke Li, and Kalyanmoy Deb. Search-based software engineering for self-adaptive systems: Survey, disappointments, suggestions and opportunities, 2020.

[45] Tao Chen, Miqing Li, and Xin Yao. On the effects of seeding strategies: A case for search-based multi-objective service composition. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1419–1426, New York, NY, USA, 2018. Association for Computing Machinery.

[46] Denis D. Chesalin, Eugene A. Kulikov, Igor A. Yaroshevich, Eugene G. Maksimov, Alla A. Selishcheva, and Roman Y. Pishchalnikov. Differential evolution reveals the effect of polar and nonpolar solvents on carotenoids: A case study of astaxanthin optical response modeling. *Swarm and Evolutionary Computation*, 75:101210, 2022.

[47] J. Cho, H. Ko, and I. Ko. Adaptive service selection according to the service density in multiple qos aspects. *IEEE Transactions on Services Computing*, 9(6):883–894, Nov 2016.

[48] Maurice Clerc. *Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem*, pages 219–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[49] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F.P. O'Boyle. M3: Semantic API migrations. In *International Conference on Automated Software Engineering (ASE)*, 2020.

[50] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[51] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[52] Kalyanmoy Deb and J. Sundar. Reference point based multi-objective optimization using evolutionary algorithms. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, page 635–642, New York, NY, USA, 2006. Association for Computing Machinery.

[53] Hans Degroote, Patrick De Causmaecker, Bernd Bischl, and Lars Kotthoff. A regression-based methodology for online algorithm selection. In Vadim Bulitko and Sabine Storandt, editors, *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, pages 37–45. AAAI Press, 2018.

[54] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017.

[55] Niranjana Deshpande, Mohamed Wiem Mkaouer, Ali Ouni, and Naveen Sharma. Search-based third-party library migration at the method-level. In Juan Luis Jiménez Laredo, J. Ignacio Hidalgo, and Kehinde Oluwatoyin Babaagba, editors, *Applications of Evolutionary Computation*, pages 173–190, Cham, 2022. Springer International Publishing.

[56] Niranjana Deshpande and Naveen Sharma. Composition algorithm adaptation in service oriented systems. In *Software Architecture*, 2020.

[57] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization.* Bradford Company, USA, 2004.

[58] Maxim A. Dulebenets. A diffused memetic optimizer for reactive berth allocation and scheduling at marine container terminals in response to disruptions. *Swarm and Evolutionary Computation*, 80:101334, 2023.

[59] Michael T. M. Emmerich and André H. Deutz. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural computing*, 17(3):585–609, 2018. 30174562[pmid].

[60] Dylan Foster, Alekh Agarwal, Miroslav Dudik, Haipeng Luo, and Robert Schapire. Practical contextual bandits with regression oracles. ICML, 2018.

[61] Akalanka Galappaththi and Sarah Nadi. A data set of generalizable python code change patterns, 2023.

[62] Yaroslav Golubev, Egor Bogomolov, Egor Bulychev, and Timofey Bryksin. So much in so little: Creating lightweight embeddings of python libraries, 2022.

[63] Hassan Gomaa, Koji Hashimoto, Minseong Kim, Sam Malek, and Daniel A. Menascé. Software adaptation patterns for service-oriented architectures. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 462–469, New York, NY, USA, 2010. ACM.

[64] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE '07)*, pages 342–357, 2007.

[65] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer series in statistics. Springer, 2009.

[66] Hao He, Yulin Xu, Xiao Cheng, Guangtai Liang, and Minghui Zhou. Migrationadvisor: Recommending library migrations from large-scale open-source data. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 9–12, 2021.

[67] Hao He, Yulin Xu, Yixiao Ma, Yifei Xu, Guangtai Liang, and Minghui Zhou. A multi-metric ranking approach for library migration recommendations. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 72–83, 2021.

[68] Hao He, Yulin Xu, Yixiao Ma, Yifei Xu, Guangtai Liang, and Minghui Zhou. A multi-metric ranking approach for library migration recommendations. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 72–83. IEEE, 2021.

[69] Essam H. Houssein, Ahmed G. Gad, Yaser M. Wazery, and Ponnuthurai Nagaratnam Suganthan. Task scheduling in cloud computing based on meta-heuristics: Review, taxonomy, open challenges, and future trends. *Swarm and Evolutionary Computation*, 62:100841, 2021.

[70] IBM. An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[71] Mohayeminul Islam, Ajay Kumar Jha, and Sarah Nadi. Pymigbench and pymigtax: A benchmark and taxonomy for python library migration, 2022.

[72] C. Jatoth, G. Gangadharan, and R. Buyya. Computational intelligence based qos-aware web service composition: A systematic literature review. *IEEE Transactions on Services Computing*, 10(03):475–492, jul 2017.

[73] Yaochu Jin, Handing Wang, Tinkle Chugh, Dan Guo, and Kaisa Miettinen. Data-driven evolutionary optimization: An overview and case studies. *IEEE Transactions on Evolutionary Computation*, 23(3):442–458, 2019.

[74] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.

[75] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation*, 27(1):3–45, 2019. PMID: 30475672.

[76] Pascal Kerschke, Lars Kotthoff, Jakob Bossek, Holger H. Hoos, and Heike Trautmann. Leveraging tsp solver complementarity through machine learning. *Evolutionary Computation*, 26(4):597–620, 2018. PMID: 28836836.

[77] Tariq King, Alain Ramirez, Cruz Rodolfo, and Peter Clarke. An integrated self-testing framework for autonomic computing systems. *Journal of Computers*, 2, 11 2007.

[78] Lars Kotthoff. *Algorithm Selection for Combinatorial Search Problems: A Survey*, pages 149–190. Springer International Publishing, Cham, 2016.

[79] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. *Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA*, pages 81–95. Springer International Publishing, Cham, 2019.

[80] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Softw. Engg.*, 2018.

[81] S. Kumar, R. Bahsoon, T. Chen, K. Li, and R. Buyya. Multi-tenant cloud service composition using evolutionary optimization. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 972–979, 2018.

[82] Satish Kumar, Tao Chen, Rami Bahsoon, and Rajkumar Buyya. DATESSO: self-adapting service composition with debt-aware two levels constraint reasoning. *CoRR*, abs/2003.14377, 2020.

[83] T. Laleh, J. Paquet, S. Mokhov, and Y. Yan. Constraint adaptation in web service composition. In *2017 IEEE International Conference on Services Computing (SCC)*, pages 156–163, 2017.

[84] Tor Lattimore and Csaba Szepesvári. *Bandit Algorithms.* Cambridge University Press, 2020.

[85] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. Many-objective evolutionary algorithms: A survey. *ACM Comput. Surv.*, 48(1), sep 2015.

[86] Miqing Li, Tao Chen, and Xin Yao. A critical review of: "a practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering": Essay on quality indicator selection for sbse. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, page 17–20, New York, NY, USA, 2018. Association for Computing Machinery.

[87] Miqing Li, Tao Chen, and Xin Yao. How to evaluate solutions in pareto-based search-based software engineering? a critical review and methodological guidance. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[88] Miqing Li and Xin Yao. Quality evaluation of solution sets in multiobjective optimisation: A survey. *ACM Comput. Surv.*, 52(2), mar 2019.

[89] Mingwei Liu, Yanjun Yang, Yiling Lou, Xin Peng, Zhong Zhou, Xueying Du, and Tianyong Yang. Recommending analogical apis via knowledge graph embedding, 2023.

[90] Simone Ludwig. Applying particle swarm optimization to quality-of-service-driven web service composition. *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, pages 613–620, 03 2012.

[91] Simone A. Ludwig. Memetic algorithms applied to the optimization of workflow compositions. *Swarm and Evolutionary Computation*, 10:31–40, 2013.

[92] Gang Luo. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5, 2016.

[93] Zhongqiang Ma, Guohua Wu, Ponnuthurai Nagaratnam Suganthan, Aijuan Song, and Qizhang Luo. Performance assessment and exhaustive listing of 500+ nature-inspired metaheuristic algorithms. *Swarm and Evolutionary Computation*, 77:101248, 2023.

[94] Katherine M. Malan and Andries P. Engelbrecht. A survey of techniques for characterising fitness landscapes and some possible ways forward. *Information Sciences*, 241:148–163, 2013.

[95] Katherine M. Malan and Andries P. Engelbrecht. *Fitness Landscape Analysis for Metaheuristic Performance Prediction*, pages 103–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[96] Ruchika Malhotra, Megha Khanna, and Rajeev R. Raje. On the application of search-based techniques for software engineering predictive modeling: A systematic review and future directions. *Swarm and Evolutionary Computation*, 32:85–109, 2017.

[97] Chengying Mao, Lichuan Xiao, Xinxin Yu, and Jinfu Chen. Adapting ant colony optimization to generate test data for software structural testing. *Swarm and Evolutionary Computation*, 20:23–36, 2015.

[98] Setyo Tri Windras Mara, Ruhul Sarker, Daryl Essam, and Saber Elsayed. Solving electric vehicle–drone routing problem using memetic algorithm. *Swarm and Evolutionary Computation*, 79:101295, 2023.

[99] Kazem Meidani, Seyedali Mirjalili, and Amir Barati Farimani. Online metaheuristic algorithm selection. *Expert Systems with Applications*, 201:117058, 2022.

[100] Sarah Nadi and Nourhan Sakr. Selecting third-party libraries: The data scientist's perspective. *Empirical Softw. Engg.*, 28(1), jan 2023.

[101] Sarah Nadi and Nourhan Sakr. Selecting third-party libraries: the data scientist's perspective. *Empirical Software Engineering*, 28(1):15, 2023.

[102] Kawser Wazed Nafi, Muhammad Asaduzzaman, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Mining software information sites to recommend cross-language analogical libraries. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 913–924, 2022.

[103] Daye Nam, Brad Myers, Bogdan Vasilescu, and Vincent Hellendoorn. Improving api knowledge discovery with ml: A case study of comparable api methods. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1890–1906, 2023.

[104] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[105] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, Claudio Di Sipio, and Davide Di Ruscio. Deeplib: Machine translation techniques to recommend upgrades for third-party libraries. *Expert Systems with Applications*, 202:117267, 2022.

[106] Phuong T. Nguyen, Riccardo Rubei, Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Massimiliano Di Penta. Dealing with popularity bias in recommender systems for third-party libraries: How far are we?, 2023.

[107] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. Exploring API embedding for API usages and applications. In *International Conference on Software Engineering*, 2017.

[108] Eneko Osaba, Esther Villar-Rodriguez, Javier Del Ser, Antonio J. Nebro, Daniel Molina, Antonio LaTorre, Ponnuthurai N. Suganthan, Carlos A. Coello Coello, and Francisco Herrera. A tutorial on the design, experimentation and application of metaheuristic algorithms to real-world optimization problems. *Swarm and Evolutionary Computation*, 64:100888, 2021.

[109] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. Towards automated library migrations with error prone and refaster. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1598–1606, New York, NY, USA, 2022. Association for Computing Machinery.

[110] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. German, and Katsuro Inoue. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 2017.

[111] Ipek Ozkaya. A watershed moment for search-based software engineering. *IEEE Software*, 38(4):3–6, 2021.

[112] Rahul Pandita, Raoul Praful Jetley, Sithu D Sudarsan, and Laurie Williams. Discovering likely mappings between APIs using text mining. In *2015 International Working Conference on Source Code Analysis and Manipulation*, 2015.

[113] Annibale Panichella. An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 595–603, New York, NY, USA, 2019. Association for Computing Machinery.

[114] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. Statistical migration of API usages. In *International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[115] Rahul Putha, Luca Quadrifoglio, and Emily Zechman. Comparing ant colony optimization and genetic algorithm approaches for solving traffic signal coordination under oversaturation conditions. *Computer-Aided Civil and Infrastructure Engineering*, 27(1):14–28.

[116] Yutao Qi, Xiaodong Li, Jusheng Yu, and Qiguang Miao. User-preference based decomposition in MOEA/D without using an ideal point. *Swarm and Evolutionary Computation*, 44:597–611, 2019.

[117] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. Melt: Mining effective lightweight transformations from pull requests, 2023.

[118] Aurora Ramírez, José Raúl Romero, and Sebastián Ventura. A survey of many-objective optimisation in search-based software engineering. *Journal of Systems and Software*, 149:382–395, 2019.

[119] Allan Vinicius Rezende, Leila Silva, André Britto, and Rodrigo Amaral. Software project scheduling problem in the context of search-based software engineering: A systematic review. *J. Syst. Softw.*, 155(C):43–56, sep 2019.

[120] John R. Rice. The algorithm selection problem**this work was partially supported by the national science foundation through grant gp-32940x. this chapter was presented as the george e. forsythe memorial lecture at the computer science conference, february 19, 1975, washington, d. c. volume 15 of *Advances in Computers*, pages 65–118. Elsevier, 1976.

[121] Riccardo Rubei, Davide Di Ruscio, Claudio Di Sipio, Juri Di Rocco, and Phuong T. Nguyen. Providing upgrade plans for third-party libraries: A recommender system using migration graphs, 2022.

[122] N. B. Sariff and N. Buniyamin. Comparative study of genetic algorithm and ant colony optimization algorithm performances for robot path planning in global static environments of different complexities. In *2009 IEEE International Symposium on Computational Intelligence in Robotics and Automation - (CIRA)*, pages 132–137, 2009.

[123] Dimitrios Sarigiannis, Thomas Parnell, and Haralampos Pozidis. Weighted sampling for combined model selection and hyperparameter tuning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):5595–5603, Apr. 2020.

[124] Paul Schmiedmayer, Andreas Bauer, and Bernd Bruegge. Reducing the impact of breaking changes to web service clients during web api evolution. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 1–11, 2023.

[125] D. Schuller, M. Siebenhaar, R. Hans, O. Wenge, R. Steinmetz, and S. Schulte. Towards heuristic optimization of complex service-based workflows for stochastic qos attributes. In *2014 IEEE International Conference on Web Services*, pages 361–368, 2014.

[126] Haitham Seada and Kalyanmoy Deb. A unified evolutionary optimization procedure for single, multiple, and many objectives. *IEEE Transactions on Evolutionary Computation*, 20(3):358–369, 2016.

[127] Praveen Prakash Singh, Soumyabrata Das, Fushuan Wen, Ivo Palu, Asheesh K. Singh, and Padmanabh Thakur. Multi-objective planning of electric vehicles charging in distribution system considering priority-based vehicle-to-grid scheduling. *Swarm and Evolutionary Computation*, 77:101234, 2023.

[128] David Speck, André Biedenkapp, Frank Hutter, Robert Mattmüller, and Marius Lindauer. Learning heuristic selection with dynamic algorithm configuration. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):597–605, May 2021.

[129] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *J. Softw. Evol. Process*, 26(11):1030–1052, 2014.

[130] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Mining library migration graphs. In *19th Working Conference on Reverse Engineering*, 2012.

[131] Immanuel Trummer and Boi Faltings. Dynamically selecting composition algorithms for economical composition as a service. In *ICSOC*, 2011.

[132] Jindong Wang and Yiqiang Chen. *Introduction*, pages 3–38. Springer Nature Singapore, Singapore, 2023.

[133] L. Wang and Q. Li. A multiagent-based framework for self-adaptive software with search-based optimization. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 621–625, Oct 2016.

[134] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA: a hybrid approach to identify framework evolution. In *2010 International Conference on Software Engineering.*

[135] Xingyu Wu, Yan Zhong, Jibin Wu, Bingbing Jiang, and Kay Chen Tan. Large language model-enhanced algorithm selection: Towards comprehensive algorithm representation, 2024.

[136] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. API method recommendation via explicit matching of functionality verb phrases. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1015–1026, 2020.

[137] Zhenchang Xing and Eleni Stroulia. API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33, 2007.

[138] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2018.

[139] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 228–241, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[140] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of API migration edits. In *International Conference on Program Comprehension*, 2019.

[141] Bing Xue, Mengjie Zhang, Will N Browne, and Xin Yao. A survey on evolutionary computation approaches to feature selection. *IEEE Transactions on evolutionary computation*, 20(4):606–626, 2015.

[142] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. *CoRR*, abs/2010.13902, 2020.

[143] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1(1), May 2007.

[144] Xiaoning Zang, Li Jiang, Changyong Liang, Junfeng Dong, Wenxing Lu, and Nenad Mladenovic. Optimization approaches for the urban delivery problem with trucks and drones. *Swarm and Evolutionary Computation*, 75:101147, 2022.

[145] Jifan Zhang, Shuai Shao, Saurabh Verma, and Robert Nowak. Algorithm selection for deep active learning with imbalanced datasets, 2023.

[146] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

[147] Wei Zhang, Carl K Chang, Taiming Feng, and Hsin-yi Jiang. Qos-based dynamic web service composition with ant colony optimization. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 493–502. IEEE, 2010.

[148] Z. Zheng, Y. Zhang, and M. R. Lyu. Investigating qos of real-world web services. *IEEE Transactions on Services Computing*, 7(1):32–39, 2014.

[149] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32–49, 2011.

[150] Bingzhe Zhou, Xinying Wang, Shengbin Xu, Yuan Yao, Minxue Pan, Feng Xu, and Xiaoxing Ma. Hybrid api migration: A marriage of small api mapping models and large language models. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, Internetware '23, page 12–21, New York, NY, USA, 2023. Association for Computing Machinery.

[151] Eckart Zitzler and Simon Künzli. Indicator-based selection in multiobjective search. In Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, pages 832–842, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Appendices

Appendix A

Appendices

A.1 Details of Source and Target Library Pairs

Table A.1: This table depicts the number of source and target library method mappings available for each of the 9 migration rules used in this study. The number of source-target method mappings to be evaluated grows exponentially by m^n, where m is the number of source and n is the number of target library methods. For our experiments, we randomly sample source and target library methods to create a dataset with correct and incorrect mappings.

Source → Target	**Source library methods**	**Target library methods**
commons-logging → slf4j	260	586
slf4j-api → log4j	586	2021
easymock → mockito	2211	2265
google-collect → guava	2746	1952
gson → jackson	877	2328
testng → junit	3678	1429
json → gson	332	877
commons-lang → slf4j-api	2211	586
json-simple → gson	94	2328

Table A.2: Precision and recall obtained using the CO scheme when tuning hyperparameters. We leverage a grid-search for tuning algorithms from the MOEA framework.

0cm

	Precision			**Recall**		
Migration Rule (↓)	**NSGAII**	**IBEA**	**MOEAD**	**NSGAII**	**IBEA**	**MOEAD**
$logging \rightarrow slf4j$	0.71	0.71	0.7	0.5	0.5	0.49
$slf4j - api \rightarrow log4j$	0.51	0.53	0.46	0.6	0.6	0.54
$easymock \rightarrow mockito$	0.86	0.87	0.86	0.5	0.5	0.49
$google - collect \rightarrow guava$	0.14	0.15	0.13	0.66	0.67	0.58
$gson \rightarrow jackson$	0.59	0.6	0.55	0.54	0.54	0.5
$testng \rightarrow junit$	0.31	0.31	0.29	0.51	0.51	0.5
$json \rightarrow gson$	0.33	0.33	0.31	0.53	0.54	0.51
$commons - lang \rightarrow slf4j - api$	0.71	0.72	0.67	0.53	0.54	0.5
$json - simple \rightarrow gson$	0.47	0.49	0.4	0.68	0.71	0.6

Table A.3: Precision and recall obtained using the MS+DS scheme when tuning hyperparameters. We leverage a grid-search for tuning algorithms from the MOEA framework.

0cm

	Precision			**Recall**		
Migration Rule (↓)	**NSGAII**	**IBEA**	**MOEAD**	**NSGAII**	**IBEA**	**MOEAD**
$logging \rightarrow slf4j$	0.7	0.7	0.69	0.5	0.5	0.49
$slf4j - api \rightarrow log4j$	0.38	0.38	0.38	0.47	0.48	0.47
$easymock \rightarrow mockito$	0.86	0.86	0.86	0.5	0.5	0.49
$google - collect \rightarrow guava$	0.12	0.12	0.11	0.56	0.57	0.53
$gson \rightarrow jackson$	0.52	0.52	0.52	0.49	0.49	0.48
$testng \rightarrow junit$	0.3	0.3	0.29	0.51	0.51	0.5
$json \rightarrow gson$	0.3	0.31	0.3	0.5	0.5	0.49
$commons - lang \rightarrow slf4j - api$	0.64	0.64	0.64	0.49	0.49	0.48
$json - simple \rightarrow gson$	0.37	0.39	0.35	0.61	0.62	0.56

A.2 Additional Results from Tuning Experiments

Tables A.2, A.3 and A.4 depict the best precision and recall values obtained by NSGAII, IBEA, and MOEAD using a grid search. We evaluate 25 combinations of the crossover and mutation rates for each similarity scheme (CO, MS+DS, ALL). Additionally, we define 5000 evaluations as the termination condition for our hyperparameter tuning experiments. In particular, we select different rates belonging to [0.2,0.4,0.6,0.8,1.0]. Our evaluations demonstrate that precision and recall do

Table A.4: Precision and recall obtained using the ALL scheme when tuning hyperparameters. We leverage a grid-search for tuning algorithms from the MOEA framework.

0cm

	Precision			**Recall**		
Migration Rule (↓)	**NSGAII**	**IBEA**	**MOEAD**	**NSGAII**	**IBEA**	**MOEAD**
$logging \rightarrow slf4j$	0.7	0.7	0.69	0.5	0.5	0.49
$slf4j - api \rightarrow log4j$	0.38	0.38	0.38	0.48	0.48	0.47
$easymock \rightarrow mockito$	0.86	0.86	0.86	0.5	0.5	0.49
$google - collect \rightarrow guava$	0.12	0.12	0.11	0.56	0.57	0.53
$gson \rightarrow jackson$	0.53	0.53	0.52	0.5	0.5	0.48
$testng \rightarrow junit$	0.3	0.3	0.29	0.51	0.51	0.5
$json \rightarrow gson$	0.31	0.31	0.3	0.5	0.5	0.49
$commons - lang \rightarrow slf4j - api$	0.66	0.67	0.65	0.5	0.5	0.48
$json - simple \rightarrow gson$	0.41	0.43	0.37	0.67	0.68	0.58

not vary significantly for different hyperparameter values, and remain stable. We observe that increases in precision values result in a decrease in recall and vice versa. This behavior is expected as recall values are higher when more method mappings are selected because more false positives are identified leading to a simultaneous decrease in precision.

For UNSGAIII, RNSGAII, AGEMOEA, and SMSEMOA, we determine the impact of different mutation and crossover rates using a tree parzen estimator (TPE) provided by the PyMOO library. Similar to our tuning experiments with algorithms implemented with the MOEA framework, we set the maximum number of function evaluations to 5000. We use the TPE estimator because it is a popular Bayesian optimization technique that is more efficient than a grid search. Tables A.5, A.6 and A.7 depict the best precision and recall obtained after hyperparameter tuning for all three similarity schemes. We note that there are some changes in precision and recall, however, the average precision and recall are similar across all migration rules. We also note that the relative performance of all algorithms does not change considerably. Based on these observations, we conclude that using the default values for mutation and crossover rates is a good starting point. Moreover, in practice, the groundtruth bitstring required for computing precision, recall, and Euclidean distance is not available or known beforehand. Thus, to provide a fair assessment of our approach to practitioners and based on observations from our tuning experiments, we set the mutation rate to 0.1 and the crossover rate to 1.0 for all algorithms.

Table A.5: Precision and recall obtained using the CO scheme. We tune the crossover and mutation rate using a tree parzen estimator (TPE) provided by the PyMOO framework.

	Precision				**Recall**			
Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**
$logging \rightarrow slf4j$	0.99	0.99	1.00	0.99	0.58	0.20	0.37	0.39
$slf4j - api \rightarrow log4j$	1.00	1.00	1.00	1.00	1.00	0.11	0.60	0.77
$easymock \dot{\rightarrow} mockito$	0.95	0.95	0.95	0.94	0.48	0.32	0.40	0.42
$google - collect \rightarrow guava$	1.00	0.99	1.00	1.00	1.00	0.14	0.90	0.92
$gson \rightarrow jackson$	1.00	0.99	1.00	1.00	1.00	0.10	0.54	0.55
$testng \rightarrow junit$	0.64	0.62	0.64	0.58	0.75	0.44	0.64	0.64
$json \rightarrow gson$	0.78	0.82	0.81	0.73	0.85	0.49	0.70	0.72
$commons - lang \rightarrow slf4j - api$	1.00	1.00	1.00	1.00	1.00	0.12	0.52	0.60
$json - simple \rightarrow gson$	1.00	1.00	1.00	1.00	1.00	0.22	0.60	0.61

Table A.6: Precision and recall obtained using the MS+DS scheme. We tune the crossover and mutation rate using a tree parzen estimator (TPE) provided by the PyMOO framework.

0cm

	Precision				**Recall**			
Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**
$logging \rightarrow slf4j$	0.76	0.84	0.80	0.81	0.60	0.28	0.45	0.44
$slf4j - api \rightarrow log4j$	0.22	0.43	0.19	0.18	0.41	0.13	0.18	0.17
$easymock \dot{\rightarrow} mockito$	0.91	0.92	0.92	0.92	0.51	0.34	0.43	0.44
$google - collect \rightarrow guava$	0.23	0.52	0.29	0.28	0.61	0.36	0.59	0.60
$gson \rightarrow jackson$	0.59	0.55	0.56	0.57	0.52	0.10	0.32	0.36
$testng \rightarrow junit$	0.38	0.42	0.39	0.39	0.65	0.45	0.55	0.56
$json \rightarrow gson$	0.38	0.41	0.39	0.38	0.48	0.28	0.38	0.41
$commons - lang \rightarrow slf4j - api$	0.69	0.71	0.76	0.74	0.57	0.09	0.30	0.32
$json - simple \rightarrow gson$	0.42	0.76	0.55	0.51	0.80	0.60	0.74	0.76

Table A.7: Precision and recall obtained using the ALL scheme. We tune the crossover and mutation rate using a tree parzen estimator (TPE) provided by the PyMOO framework.

0cm

	Precision				**Recall**			
Migration (↓)	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**	**UNSGAIII**	**RNSGAII**	**AGEMOEA**	**SMSEMOA**
$logging \rightarrow slf4j$	0.76	0.84	0.81	0.81	0.60	0.27	0.45	0.44
$slf4j - api \rightarrow log4j$	0.32	0.41	0.24	0.20	0.66	0.13	0.26	0.20
$easymock \dot{\rightarrow} mockito$	0.92	0.92	0.92	0.92	0.52	0.34	0.43	0.44
$google - collect \rightarrow guava$	0.27	0.59	0.29	0.27	0.77	0.40	0.60	0.60
$gson \rightarrow jackson$	0.63	0.68	0.59	0.60	0.63	0.12	0.33	0.36
$testng \rightarrow junit$	0.38	0.42	0.40	0.39	0.65	0.45	0.55	0.56
$json \rightarrow gson$	0.38	0.42	0.41	0.39	0.49	0.29	0.40	0.43
$commons - lang \rightarrow slf4j - api$	0.82	0.96	0.82	0.81	0.85	0.13	0.43	0.45
$json - simple \rightarrow gson$	0.48	0.72	0.57	0.55	1.00	0.71	0.77	0.79

A.3 Additional Results regarding Statistical Significance

In our work, we used the Kruskal-Wallis test to evaluate the statistical significance of precision, recall, HV, and ED values achieved by all 7 algorithms. All values achieved using CO, MS+DS and ALL schemes are statistically significant *i.e.,* p-values were < 0.05. These results denote that at least one algorithm's distribution is different from the other algorithms, that is, the Kruskal-Wallis test does not tell us which algorithm pairs have different distributions. As a result, we apply the Dunn test for a posthoc analysis to determine which algorithm pairs are statistically different. In Tables A.8, A.10 and A.9, we list the algorithm pairs that have a p-value > 0.05 for each of the three similarity schemes for every migration rule. The listed algorithm pairs have similar distributions that are not statistically significant.

Table A.8: Posthoc test results from the Dunn test to examine the statistical significance of precision, recall, HV, and ED values when using the CO scheme. Algorithm pairs with p-values¿0.05 are listed and denote those distributions that are not statistically significant.

Migration Rule	Precision	Recall	HV	ED
logging $\rightarrow$ *slf4j*	NSGAII & IBEA, NSGAII & MOEAD, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA	NSGAII & IBEA, NSGAII & MOEAD, IBEA & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA	IBEA & UNSGAIII, IBEA & SMSEMOA, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA	NSGAII & IBEA, NSGAII & SMSEMOA, IBEA & UNSGAIII, IBEA & SMSEMOA, MOEAD & RNSGAII, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA
slf4j − *api* $\rightarrow$ *log4j*	NSGAII & IBEA, NSGAII & MOEAD, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & MOEAD, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII	NSGAII & IBEA, NSGAII & AGEMOEA, IBEA & SMSEMOA, MOEAD & UNSGAIII, UNSGAIII & RNSGAII, RNSGAII & SMSEMOA	NSGAII & IBEA, NSGAII & AGEMOEA, IBEA & AGEMOEA, MOEAD & RNSGAII, UNSGAIII & SMSEMOA
easymock $\rightarrow$ *mockito*	NSGAII & IBEA, NSGAII & MOEAD, MOEAD & RNSGAII, AGEMOEA & SMSEMOA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & MOEAD, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA,AGEMOEA & SMSEMOA	NSGAII & IBEA, IBEA & UNSGAIII, MOEAD & RNSGAII, UNSGAIII & AGEMOEA, RNSGAII & SMSEMOA, AGEMOEA & SMSEMOA	NSGAII & IBEA, NSGAII & AGEMOEA, NSGAII & SMSEMOA, IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & RNSGAII, AGEMOEA & SMSEMOA

google − *collect* → *guava*	NSGAII & IBEA, NSGAII & MOEAD, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA	NSGAII & IBEA, NSGAII & MOEAD, IBEA & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & IBEA, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA	NSGAII & RNSGAII, NSGAII & AGEMOEA, IBEA & UNSGAIII, RNSGAII & AGEMOEA	NSGAII & RNSGAII, NSGAII & AGEMOEA, IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, AGEMOEA & SMSEMOA
gson → *jackson*	NSGAII & IBEA, NSGAII & MOEAD, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA	NSGAII & IBEA, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & SMSEMOA	NSGAII & IBEA, IBEA & AGEMOEA, MOEAD & UNSGAIII, UNSGAIII & RNSGAII, RNSGAII & SMSEMOA, AGEMOEA & SMSEMOA	NSGAII & IBEA, NSGAII & AGEMOEA, IBEA & AGEMOEA, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, AGEMOEA & SMSEMOA
testng → *junit*	NSGAII & IBEA, NSGAII & MOEAD, MOEAD & RNSGAII, AGEMOEA & SMSEMOA	NSGAII & IBEA, NSGAII & MOEAD, IBEA & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, SMSEMOA & UNSGAIII	NSGAII & IBEA, MOEAD & RNSGAII, UNSGAIII & AGEMOEA, RNSGAII & SMSEMOA	NSGAII & RNSGAII, NSGAII & AGEMOEA, NSGAII & SMSEMOA, IBEA & UNSGAIII, MOEAD & RNSGAII, AGEMOEA & SMSEMOA
json → *gson*	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, MOEAD & NSGAII, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & NSGAII, UNSGAIII & SMSEMOA, RNSGAII & IBEA, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, SMSEMOA & UNSGAIII	NSGAII & IBEA, IBEA & NSGAII, IBEA & UNSGAIII, MOEAD & RNSGAII, UNSGAIII & IBEA, UNSGAIII & AGEMOEA, RNSGAII & MOEAD, RNSGAII & SMSEMOA, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	NSGAII & AGEMOEA, NSGAII & SMSEMOA, IBEA & UNSGAIII, MOEAD & RNSGAII, UNSGAIII & IBEA, RNSGAII & MOEAD, AGEMOEA & NSGAII, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & AGEMOEA

commons−lang → slf4j − api	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, MOEAD & NSGAII, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, IBEA & NSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & AGEMOEA, NSGAII & SMSEMOA, IBEA & NSGAII, IBEA & AGEMOEA, MOEAD & UNSGAIII, UNSGAIII & MOEAD, UNSGAIII & RNSGAII, RNSGAII & UNSGAIII, RNSGAII & SMSEMOA, AGEMOEA & NSGAII, AGEMOEA & IBEA, SMSEMOA & NSGAII, SMSEMOA & RNSGAII	NSGAII & IBEA, NSGAII & SMSEMOA, IBEA & NSGAII, IBEA & SMSEMOA, MOEAD & RNSGAII, RNSGAII & MOEAD, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, SMSEMOA & NSGAII, SMSEMOA & IBEA
json − simple → gson	NSGAII & IBEA, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, UNSGAIII & SMSEMOA, RNSGAII & NSGAII, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & UNSGAIII, IBEA & RNSGAII, IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & UNSGAIII, UNSGAIII & NSGAII, UNSGAIII & MOEAD, RNSGAII & IBEA, RNSGAII & AGEMOEA, RNSGAII & SMSEMOA, AGEMOEA & IBEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & IBEA, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	NSGAII & AGEMOEA, NSGAII & SMSEMOA, IBEA & MOEAD, MOEAD & IBEA, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, AGEMOEA & NSGAII, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA

Table A.9: Posthoc test results from the Dunn test to examine the statistical significance of precision, recall, HV, and ED values when using the ALL scheme. Algorithm pairs with p-values¿0.05 are listed and denote those distributions that are not statistically significant.

Migration Rule	Precision	Recall	HV	ED
$logging \rightarrow slf4j$	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, MOEAD & NSGAII, MOEAD & RNSGAII, UNSGAIII & AGEMOEA, RNSGAII & MOEAD, RNSGAII & SMSEMOA, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, UNSGAIII & AGEMOEA, RNSGAII & NSGAII, RNSGAII & SMSEMOA, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & RNSGAII, UNSGAIII & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, RNSGAII & UNSGAIII, AGEMOEA & IBEA, SMSEMOA & IBEA, SMSEMOA & UNSGAIII	NSGAII & IBEA, NSGAII & SMSEMOA, IBEA & NSGAII, MOEAD & RNSGAII, RNSGAII & MOEAD, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & AGEMOEA
$slf4j - api \rightarrow log4j$	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & NSGAII, UNSGAIII & AGEMOEA, RNSGAII & IBEA, RNSGAII & SMSEMOA, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, UNSGAIII & AGEMOEA, RNSGAII & NSGAII, RNSGAII & SMSEMOA, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	NSGAII & SMSEMOA, IBEA & RNSGAII, MOEAD & UNSGAIII, MOEAD & RNSGAII, UNSGAIII & MOEAD, RNSGAII & IBEA, RNSGAII & MOEAD, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & SMSEMOA, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & RNSGAII, UNSGAIII & AGEMOEA, RNSGAII & IBEA, RNSGAII & MOEAD, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & AGEMOEA

$easymock \rightarrow mockito$	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, MOEAD & NSGAII, MOEAD & RNSGAII, RNSGAII & MOEAD, AGEMOEA & SMSEMOA, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, UNSGAIII & SMSEMOA, RNSGAII & NSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, IBEA & NSGAII, MOEAD & RNSGAII, UNSGAIII & AGEMOEA, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, RNSGAII & SMSEMOA, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & AGEMOEA, NSGAII & SMSEMOA, IBEA & NSGAII, IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & RNSGAII, RNSGAII & MOEAD, AGEMOEA & NSGAII, AGEMOEA & IBEA, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & IBEA, SMSEMOA & AGEMOEA
$google - collect \rightarrow guava$	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & NSGAII, UNSGAIII & SMSEMOA, RNSGAII & IBEA, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & NSGAII, UNSGAIII & SMSEMOA, RNSGAII & IBEA, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & AGEMOEA, IBEA & SMSEMOA, UNSGAIII & RNSGAII, RNSGAII & UNSGAIII, AGEMOEA & NSGAII, AGEMOEA & SMSEMOA, SMSEMOA & IBEA, SMSEMOA & AGEMOEA	NSGAII & MOEAD, NSGAII & AGEMOEA, IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & NSGAII, MOEAD & RNSGAII, RNSGAII & MOEAD, AGEMOEA & NSGAII, AGEMOEA & IBEA, AGEMOEA & SMSEMOA, SMSEMOA & IBEA, SMSEMOA & AGEMOEA

gson → *jackson*	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & NSGAII, RNSGAII & IBEA, AGEMOEA & SMSEMOA, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, RNSGAII & NSGAII, AGEMOEA & SMSEMOA, SMSEMOA & AGEMOEA	IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & UNSGAIII, UNSGAIII & MOEAD, UNSGAIII & RNSGAII, RNSGAII & UNSGAIII, AGEMOEA & IBEA, AGEMOEA & SMSEMOA, SMSEMOA & IBEA, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & UNSGAIII, IBEA & NSGAII, IBEA & AGEMOEA, MOEAD & RNSGAII, UNSGAIII & NSGAII, UNSGAIII & RNSGAII, RNSGAII & MOEAD, RNSGAII & UNSGAIII, AGEMOEA & IBEA, AGEMOEA & SMSEMOA, SMSEMOA & AGEMOEA
testng → *junit*	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, MOEAD & NSGAII, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & NSGAII, UNSGAIII & SMSEMOA, RNSGAII & IBEA, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, IBEA & NSGAII, MOEAD & RNSGAII, UNSGAIII & AGEMOEA, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & SMSEMOA, IBEA & NSGAII, IBEA & SMSEMOA, MOEAD & UNSGAIII, UNSGAIII & MOEAD, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & IBEA, SMSEMOA & AGEMOEA
json → *gson*	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, MOEAD & NSGAII, MOEAD & RNSGAII, UNSGAIII & SMSEMOA, RNSGAII & MOEAD, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, UNSGAIII & SMSEMOA, RNSGAII & NSGAII, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, IBEA & NSGAII, IBEA & UNSGAIII, MOEAD & RNSGAII, UNSGAIII & IBEA, UNSGAIII & AGEMOEA, RNSGAII & MOEAD, RNSGAII & SMSEMOA, AGEMOEA & UNSGAIII, AGEMOEA & SMSEMOA, SMSEMOA & RNSGAII, SMSEMOA & AGEMOEA	NSGAII & AGEMOEA, NSGAII & SMSEMOA, IBEA & UNSGAIII, MOEAD & UNSGAIII, UNSGAIII & IBEA, UNSGAIII & MOEAD, AGEMOEA & NSGAII, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & AGEMOEA

$commons-lang \rightarrow slf4j-api$	NSGAII & IBEA, NSGAII & MOEAD, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & NSGAII, UNSGAIII & SMSEMOA, RNSGAII & IBEA, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, UNSGAIII & SMSEMOA, RNSGAII & NSGAII, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & SMSEMOA, IBEA & NSGAII, IBEA & RNSGAII, MOEAD & UNSGAIII, MOEAD & RNSGAII, UNSGAIII & MOEAD, RNSGAII & IBEA, RNSGAII & MOEAD, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & AGEMOEA, IBEA & NSGAII, IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & RNSGAII, RNSGAII & MOEAD, AGEMOEA & NSGAII, AGEMOEA & IBEA, AGEMOEA & SMSEMOA, SMSEMOA & IBEA, SMSEMOA & AGEMOEA
$json-simple \rightarrow gson$	NSGAII & IBEA, IBEA & NSGAII, UNSGAIII & SMSEMOA, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & RNSGAII, IBEA & NSGAII, IBEA & MOEAD, MOEAD & IBEA, UNSGAIII & SMSEMOA, RNSGAII & NSGAII, RNSGAII & AGEMOEA, AGEMOEA & RNSGAII, AGEMOEA & SMSEMOA, SMSEMOA & UNSGAIII, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & SMSEMOA, IBEA & NSGAII, IBEA & SMSEMOA, MOEAD & UNSGAIII, UNSGAIII & MOEAD, AGEMOEA & SMSEMOA, SMSEMOA & NSGAII, SMSEMOA & IBEA, SMSEMOA & AGEMOEA	NSGAII & IBEA, NSGAII & AGEMOEA, IBEA & NSGAII, IBEA & AGEMOEA, IBEA & SMSEMOA, MOEAD & UNSGAIII, UNSGAIII & MOEAD, AGEMOEA & NSGAII, AGEMOEA & IBEA, AGEMOEA & SMSEMOA, SMSEMOA & IBEA, SMSEMOA & AGEMOEA

Table A.10: Posthoc test results from the Dunn test to examine the statistical significance of precision, recall, HV, and ED values when using the MS+DS scheme. Algorithm pairs with p-values¿0.05 are listed and denote those distributions that are not statistically significant.

Migration Rule	Precision	Recall	HV	ED

logging → *slf4j*	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , MOEAD & NSGAII , MOEAD & RNSGAII , UNSGAIII & AGEMOEA , RNSGAII & MOEAD , RNSGAII & SMSEMOA , AGEMOEA & UNSGAIII , AGEMOEA & SMSEMOA , SMSEMOA & RNSGAII , SMSEMOA & AGEMOEA	NSGAII & IBEA , IBEA & NSGAII , AGEMOEA & SMSEMOA , SMSEMOA & AGEMOEA	NSGAII & IBEA , IBEA & NSGAII , IBEA & AGEMOEA , MOEAD & RNSGAII , UNSGAIII & RNSGAII , UNSGAIII & SMSEMOA , RNSGAII & MOEAD , RNSGAII & UNSGAIII , AGEMOEA & IBEA , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII	NSGAII & IBEA , NSGAII & SMSEMOA , IBEA & NSGAII , IBEA & SMSEMOA , MOEAD & RNSGAII , RNSGAII & MOEAD , AGEMOEA & SMSEMOA , SMSEMOA & NSGAII , SMSEMOA & IBEA , SMSEMOA & AGEMOEA
slf4j − *api* → *log4j*	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , IBEA & RNSGAII , MOEAD & NSGAII , UNSGAIII & AGEMOEA , RNSGAII & IBEA , RNSGAII & SMSEMOA , AGEMOEA & UNSGAIII , AGEMOEA & SMSEMOA , SMSEMOA & RNSGAII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & RNSGAII , IBEA & NSGAII , IBEA & MOEAD , MOEAD & IBEA , UNSGAIII & AGEMOEA , RNSGAII & NSGAII , RNSGAII & SMSEMOA , AGEMOEA & UNSGAIII , AGEMOEA & SMSEMOA , SMSEMOA & RNSGAII , SMSEMOA & AGEMOEA	NSGAII & AGEMOEA , IBEA & AGEMOEA , IBEA & SMSEMOA , MOEAD & UNSGAIII , MOEAD & RNSGAII , UNSGAIII & MOEAD , RNSGAII & MOEAD , RNSGAII & SMSEMOA , AGEMOEA & NSGAII , AGEMOEA & IBEA , SMSEMOA & IBEA , SMSEMOA & RNSGAII	NSGAII & IBEA , NSGAII & SMSEMOA , IBEA & NSGAII , IBEA & RNSGAII , MOEAD & RNSGAII , UNSGAIII & AGEMOEA , RNSGAII & IBEA , RNSGAII & MOEAD , AGEMOEA & UNSGAIII , AGEMOEA & SMSEMOA , SMSEMOA & NSGAII , SMSEMOA & AGEMOEA

easymock → mockito	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , MOEAD & NSGAII , MOEAD & RNSGAII , UNSGAIII & AGEMOEA , RNSGAII & MOEAD , RNSGAII & SMSEMOA , AGEMOEA & UNSGAIII , AGEMOEA & SMSEMOA , SMSEMOA & RNSGAII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & RNSGAII , IBEA & NSGAII , IBEA & MOEAD , MOEAD & IBEA , UNSGAIII & SMSEMOA , RNSGAII & NSGAII , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , IBEA & NSGAII , IBEA & AGEMOEA , MOEAD & RNSGAII , UNSGAIII & RNSGAII , UNSGAIII & SMSEMOA , RNSGAII & MOEAD , RNSGAII & UNSGAIII , AGEMOEA & IBEA , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & AGEMOEA , NSGAII & SMSEMOA , IBEA & NSGAII , IBEA & AGEMOEA , IBEA & SMSEMOA , MOEAD & RNSGAII , RNSGAII & MOEAD , AGEMOEA & NSGAII , AGEMOEA & IBEA , AGEMOEA & SMSEMOA , SMSEMOA & NSGAII , SMSEMOA & IBEA , SMSEMOA & AGEMOEA
google − collect → guava	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , IBEA & RNSGAII , MOEAD & NSGAII , UNSGAIII & SMSEMOA , RNSGAII & IBEA , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , IBEA & RNSGAII , MOEAD & NSGAII , UNSGAIII & SMSEMOA , RNSGAII & IBEA , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	IBEA & AGEMOEA , MOEAD & RNSGAII , UNSGAIII & RNSGAII , UNSGAIII & SMSEMOA , RNSGAII & MOEAD , RNSGAII & UNSGAIII , AGEMOEA & IBEA , SMSEMOA & UNSGAIII	NSGAII & MOEAD , NSGAII & AGEMOEA , IBEA & AGEMOEA , IBEA & SMSEMOA , MOEAD & NSGAII , MOEAD & RNSGAII , UNSGAIII & SMSEMOA , RNSGAII & MOEAD , AGEMOEA & NSGAII , AGEMOEA & IBEA , SMSEMOA & IBEA , SMSEMOA & UNSGAIII

gson → *jackson*	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , IBEA & RNSGAII , MOEAD & NSGAII , UNSGAIII & SMSEMOA , RNSGAII & IBEA , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , IBEA & NSGAII , UNSGAIII & SMSEMOA , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	IBEA & AGEMOEA , MOEAD & UNSGAIII , UNSGAIII & MOEAD , UNSGAIII & RNSGAII , RNSGAII & UNSGAIII , RNSGAII & SMSEMOA , AGEMOEA & IBEA , SMSEMOA & RNSGAII	NSGAII & IBEA , NSGAII & UNSGAIII , IBEA & NSGAII , IBEA & AGEMOEA , MOEAD & RNSGAII , UNSGAIII & NSGAII , UNSGAIII & RNSGAII , RNSGAII & MOEAD , RNSGAII & UNSGAIII , AGEMOEA & IBEA , AGEMOEA & SMSEMOA , SMSEMOA & AGEMOEA
testng → *junit*	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , MOEAD & NSGAII , MOEAD & RNSGAII , UNSGAIII & AGEMOEA , RNSGAII & MOEAD , RNSGAII & SMSEMOA , AGEMOEA & UNSGAIII , AGEMOEA & SMSEMOA , SMSEMOA & RNSGAII , SMSEMOA & AGEMOEA	NSGAII & IBEA , IBEA & NSGAII , UNSGAIII & SMSEMOA , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , IBEA & NSGAII , IBEA & AGEMOEA , MOEAD & RNSGAII , UNSGAIII & RNSGAII , UNSGAIII & SMSEMOA , RNSGAII & MOEAD , RNSGAII & UNSGAIII , AGEMOEA & IBEA , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & AGEMOEA , NSGAII & SMSEMOA , IBEA & NSGAII , IBEA & AGEMOEA , IBEA & SMSEMOA , MOEAD & UNSGAIII , UNSGAIII & MOEAD , AGEMOEA & NSGAII , AGEMOEA & IBEA , AGEMOEA & SMSEMOA , SMSEMOA & NSGAII , SMSEMOA & IBEA , SMSEMOA & AGEMOEA

$json \rightarrow gson$	NSGAII & IBEA , NSGAII & MOEAD , IBEA & NSGAII , MOEAD & NSGAII , MOEAD & RNSGAII , UNSGAIII & AGEMOEA , RNSGAII & MOEAD , RNSGAII & SMSEMOA , AGEMOEA & UNSGAIII , AGEMOEA & SMSEMOA , SMSEMOA & RNSGAII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & RNSGAII , IBEA & NSGAII , IBEA & MOEAD , MOEAD & IBEA , UNSGAIII & SMSEMOA , RNSGAII & NSGAII , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , IBEA & NSGAII , IBEA & AGEMOEA , MOEAD & RNSGAII , UNSGAIII & AGEMOEA , UNSGAIII & SMSEMOA , RNSGAII & MOEAD , RNSGAII & SMSEMOA , AGEMOEA & IBEA , AGEMOEA & UNSGAIII , SMSEMOA & UNSGAIII , SMSEMOA & RNSGAII	NSGAII & UNSGAIII , NSGAII & AGEMOEA , IBEA & MOEAD , IBEA & UNSGAIII , MOEAD & IBEA , UNSGAIII & NSGAII , UNSGAIII & IBEA , AGEMOEA & NSGAII , AGEMOEA & SMSEMOA , SMSEMOA & AGEMOEA
$commons-lang \rightarrow slf4j-api$	NSGAII & IBEA , NSGAII & RNSGAII , IBEA & NSGAII , IBEA & MOEAD , MOEAD & IBEA , UNSGAIII & SMSEMOA , RNSGAII & NSGAII , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & RNSGAII , IBEA & NSGAII , IBEA & MOEAD , MOEAD & IBEA , UNSGAIII & SMSEMOA , RNSGAII & NSGAII , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & AGEMOEA , IBEA & RNSGAII , IBEA & SMSEMOA , MOEAD & UNSGAIII , UNSGAIII & MOEAD , UNSGAIII & RNSGAII , RNSGAII & IBEA , RNSGAII & UNSGAIII , AGEMOEA & NSGAII , AGEMOEA & SMSEMOA , SMSEMOA & IBEA , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & AGEMOEA , NSGAII & SMSEMOA , IBEA & NSGAII , IBEA & AGEMOEA , IBEA & SMSEMOA , MOEAD & RNSGAII , RNSGAII & MOEAD , AGEMOEA & NSGAII , AGEMOEA & IBEA , SMSEMOA & NSGAII , SMSEMOA & IBEA

json − *simple* → *gson*	NSGAII & IBEA , IBEA & NSGAII , UNSGAIII & SMSEMOA , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & RNSGAII , IBEA & NSGAII , IBEA & MOEAD , MOEAD & IBEA , UNSGAIII & SMSEMOA , RNSGAII & NSGAII , RNSGAII & AGEMOEA , AGEMOEA & RNSGAII , AGEMOEA & SMSEMOA , SMSEMOA & UNSGAIII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & AGEMOEA , IBEA & NSGAII , MOEAD & UNSGAIII , UNSGAIII & MOEAD , UNSGAIII & RNSGAII , RNSGAII & UNSGAIII , RNSGAII & SMSEMOA , AGEMOEA & NSGAII , AGEMOEA & SMSEMOA , SMSEMOA & RNSGAII , SMSEMOA & AGEMOEA	NSGAII & IBEA , NSGAII & RNSGAII , IBEA & NSGAII , MOEAD & AGEMOEA , UNSGAIII & SMSEMOA , RNSGAII & NSGAII , AGEMOEA & MOEAD , SMSEMOA & UNSGAIII