Singapore Management University

Institutional Knowledge at Singapore Management University

Dissertations and Theses Collection (Open Access)

Dissertations and Theses

1-2014

Generic Instance-Specific Automated Parameter Tuning Framework

Linda LINDAWATI
Singapore Management University, lindawati.2008@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/etd_coll

Part of the Operations Research, Systems Engineering and Industrial Engineering Commons, and the Theory and Algorithms Commons

Citation

LINDAWATI, Linda. Generic Instance-Specific Automated Parameter Tuning Framework. (2014). 1-159.
Available at: https://ink.library.smu.edu.sg/etd_coll/100

This PhD Dissertation is brought to you for free and open access by the Dissertations and Theses at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Dissertations and Theses Collection (Open Access) by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Generic Instance-Specific Automated Parameter Tuning Framework

Lindawati

Singapore Management University
2014

Generic Instance-Specific Automated Parameter Tuning Framework

by
Lindawati

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Hoong Chuin LAU (Supervisor/Chair)
Associate Professor of Information Systems
Singapore Management University

David LO
Assistant Professor of Information Systems
Singapore Management University

Feida ZHU
Assistant Professor of Information Systems
Singapore Management University

Roland YAP
Associate Professor of Computing
National University of Singapore

Singapore Management University
2014

Copyright (2014) Lindawati

Generic Instance-Specific Automated Parameter Tuning Framework

Lindawati

Abstract

Meta-heuristic algorithms play an important role in solving combinatorial optimization problems (COP) in many practical applications. The caveat is that the performance of these meta-heuristic algorithms is highly dependent on their parameter configuration which controls the algorithm behaviour. Selecting the best parameter configuration is often a difficult, tedious and unsatisfying task. This thesis studies the problem of automating the selection of good parameter configurations.

Existing approaches to address the challenges of parameter configuration can be classified into **one-size-fits-all** and **instance-specific** approaches. One-size-fits-all approaches focus on finding a single best parameter configuration for a set of problem instances, while instance-specific approaches attempt to find parameter configurations on a per instance-basis, based on identifying specific features of a specific problem. Both approaches have their strengths and limitations, yet neither offers a **generic** approach for finding instance-specific parameter configurations.

In this thesis, we take a middle ground hybrid approach, where our goal is to perform instance-specific tuning via clustering of problem instances using a problem-independent feature. Our approach is similar to ISAC [64], but instead of using problem-specific features, we propose a problem-independent feature from the local search trajectory.

We are primarily concerned with the tuning of target algorithms that are local-search based, where we make use of the local search trajectory as the feature, since they can be obtained from any given local-search based algorithm with a small additional computation budget. We show that there is a strong correlation between search trajectories and good parameter configurations, and hence clustering by search trajectories allow a configurator to find parameter configurations based on clusters rather than the entire set of training instances. We propose two **generic frameworks**: Clu-

PaTra and CluPaTra-II that cluster a set of instances using search trajectories before configuring the parameters for each cluster. In CluPaTra, we use a simple pair-wise sequence alignment technique, while in CluPaTra-II, we design two pattern mining techniques to extract compact features for clustering purposes. Using our approaches, we run extensive numerical experiments on three classical problems : Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP) and demonstrate encouraging results in both cluster quality and overall computational performance.

A second contribution of this thesis is the implementation of an automated parameter tuning system that comprises CluPaTra, CluPaTra-II, and other components required for automated tuning. More specifically, we develop AutoParTune, a web-based workbench that enables algorithm designers to perform automated parameter tuning with minimal effort. AutoParTune is constructed based on a three-tier architecture that integrates instance-specific parameter configuration with parameter search space reduction and global tuning. We implement two security techniques to prevent Internet attacks and design a communication schema to establish communication between components. We apply this workbench to tune two problems from industry: the Aircraft Spares Inventory Optimization Problem and the Theme Park Personalized Intelligent Route Guidance Problem. AutoParTune shows a better overall performance compared to the default configurations.

Finally, as a bridge for future works, we consider an extension of the above instance-specific tuning approach to tune population-based algorithms such as Genetic Algorithms. We introduce two preliminary ideas: PeTra and PaRG which are designed based on generic features pertaining to population dynamics in a Genetic Algorithm. Preliminary experiments with the Two-Population Genetic Algorithm have given promising results in terms of the overall computational performance.

In summary, we show in this thesis that our approach yields significant improvement in performance compared with the pure one-size-fits-all configurators on both classical and industry problems. We observe that our approach performs significantly

better or equal to several existing instance-specific configurators which use problem specific features. Based on these results, we claim that: (1) Methodologically dividing the instances into smaller clusters before tuning provides better parameter configurations; (2) The Search Trajectory is a suitable generic feature to cluster similar instances for tuning process; (3) Our web-based workbench provides an effective tool for tuning complex optimization problems; and (4) There are viable extensions for automated parameter tuning of population-based algorithms.

Contents

1 Introduction **1**

1.1 The Journey for a High-Performance Algorithm 1

1.2 Summary of Contribution 3

1.3 Thesis Outline 5

1.4 List of Publications 5

2 Background **7**

2.1 Combinatorial Optimization Problem 7

2.2 Algorithms for Solving COPs 9

2.2.1 Exact Algorithms 9

2.2.2 Non-Exact Algorithms 10

2.3 Meta-heuristic for Solving COPs 11

2.3.1 Local-search-based Algorithms 12

2.3.2 Population-based Algorithms 13

2.3.3 Challenge in Designing Meta-heuristic Algorithms for COPs . 15

2.4 Automated Parameter Tuning Problem 17

2.5 Literature Review on Automated Tuning 20

2.5.1 Classification of Current Approaches 21

2.5.2 Analyzing Different Approaches 22

2.5.3 Further Relevant Research 28

2.6 Chapter Summary 29

3 Instance-Specific Automated Parameter Tuning via Trajectory Clustering (CluPaTra) 31

3.1 Framework Overview 32

3.2 Feature Selection 33

3.2.1 Search Trajectory Definition 36

3.2.2 Representation of Search Trajectory 37

3.3 Similarity Calculation 39

3.3.1 Basic Sequence Alignment 40

3.3.2 Robust Sequence Alignment 41

3.4 Clustering Method 42

3.5 CluPaTra Instantiations 45

3.6 Empirical Experiment Result 45

3.6.1 Experiment Measurement 45

3.6.2 Target Problems and Algorithms 46

3.6.3 Experiment Setting and Setup 49

3.6.4 Verification of Similarity Preservation 50

3.6.5 Clustering Analyses 52

3.6.6 Computational Time 57

3.6.7 Performance Comparison 57

3.6.8 Comparison of Different Clustering Methods 60

3.7 Discussion 61

3.8 Chapter Summary 62

4 Pattern Mining Approaches for Instance-specific Automated Parameter Tuning 63

4.1 CluPaTra-II: Tuning Framework using Pattern Mining Approach . . . 64

4.2 SufTra: Pattern Mining via Suffix Tree 71

4.2.1 Sequence Hashing 72

4.2.2 Suffix Tree Construction 73

4.2.3 Features Retrieval 74

4.2.4 Instance-Feature Metric Calculation 75
4.3 FloTra: Graph Pattern Mining for Search Trajectory 75
4.3.1 Stage 1: Mining Flower Thorns and Petals 77
4.3.2 Stage 2: Mining Long Stem 78
4.3.3 Stage 3: Assembling the Flower 79
4.3.4 Stage 4: Instance-Feature Metric Calculation 79
4.4 Empirical Experiment Result 80
4.4.1 Cluster Analysis 80
4.4.2 Computational Time 82
4.4.3 Performance Comparison 83
4.5 Discussion 84
4.6 Chapter Summary 85

5 Web-based Automated Parameter Tuning Workbench 87
5.1 AutoParTune Overview 88
5.2 AutoParTune Components 91
5.2.1 Instance-Specific Tuning 91
5.2.2 Parameter Search Space Reduction 91
5.2.3 Global Tuning 94
5.3 AutoPartune Features 95
5.3.1 Security Issue 95
5.3.2 Integration Issue 96
5.4 Application Architecture 97
5.5 Empirical Experiment Result 99
5.5.1 Classical COPs 100
5.5.2 Aircraft Spares Inventory Optimization Problem 101
5.5.3 Theme Park Personalized Intelligent Route Guidance Problem 103
5.6 Discussion 110
5.7 Chapter Summary 111

6 Instance-Specific Tuning: Extension to Genetic Algorithms 112

6.1 PeTra: **P**opulation **E**volution **Tra**jectory Similarity 113

6.2 PaRG: **Pa**rent Inheritance **R**elationship similarity in **G**raph representation 115

6.2.1 Graph Transformation 116

6.2.2 Feature Extraction 118

6.3 Empirical Experiment Result 119

6.3.1 Target Problem and Algorithm 119

6.3.2 Experiment Setting and Setup 122

6.3.3 Performance Comparison 122

6.4 Discussion 122

6.5 Chapter Summary 123

7 Conclusions 125

7.1 Contributions 125

7.2 Future Directions 128

Appendix:

A Empirical Experiment Result 132

B Quick Start Guide for AutoParTune 136

List of Figures

1.1 Summary of PhD Contributions 3

2.1 Genetic Algorithm Cycles. 15

2.2 Tuning Scenario 17

3.1 CluPaTra Framework 33

3.2 CluPaTra Training Phase 34

3.3 CluPaTra Testing Phase 34

3.4 Example of Search Trajectory from the Traveling Salesman Problem (TSP) instance 36

3.5 Example of Direct Sequence Representation of Search Trajectory for the Travelling Salesman Problem (TSP) instance 37

3.6 Example of Transition Sequence for search trajectory of Traveling Salesman Problem (TSP) instance 40

3.7 Hierarchical Clustering Method: AGNES (AGglomerative NESting) . 43

3.8 Evaluation Graph for L-Method to Determine Number of Cluster . . . 44

3.9 Search Trajectories of three TSP instances using two random parameter configuration 53

3.10 Search Trajectory Similarity Score between two TSP and QAP instances and 10 other random instances using 5 Different Random Parameter Configurations 53

3.11 TSP Cluster Result Comparison 55

3.12 QAP Cluster Result Comparison 55

4.1 Similarity Patterns from three search trajectory sequences 65
4.2 Sequence and Graph Search Trajectories Representation for three Quadratic Assignment Problem (QAP) instances 67
4.3 Tuning Framework using Pattern Mining Approach 68
4.4 Tuning Framework using Pattern Mining Approach Training Phase . . 68
4.5 Tuning Framework using Pattern Mining Approach Testing Phase . . 69
4.6 CluPaTra-II Steps and Output 70
4.7 Example of Suffix Tree for a single string S_1 ($LMMNP$) and for two strings S_1=$LMMNP$ and S_2=$LMNMM$ 73
4.8 Flower Graph with stem, petals and thorns 76
4.9 DFS Path for a particular node in search trajectory graph 78
4.10 Create Flower Thorns and Petals Procedure using Suffix Tree 78
4.11 Example of frequent subgraph found by FloTra 79
4.12 Example of clusters' signature for each cluster generated using FloTra 82

5.1 Phases of Fact-RSM, Parameter Search Space Reduction Method using DoE methodology 92
5.2 2^k Full Factorial Design for Fact-RSM 93
5.3 AutoParTune Components Communication Schema 97
5.4 AutoParTune Design Architecture 99
5.5 AutoParTune Database Design 100
5.6 Tuning Scenario for Personalized Intelligent Route Guidance 107
5.7 Effect of Patron Preferences on Route Generated from Personalized Intelligent Route Guidance Algorithm 109

6.1 Genetic Algorithm Population Presentation. 115
6.2 Steps in PaRG: **Pa**rent Inheritance **R**elationship similarity in **G**raph representation. 116
6.3 Parent Inheritance Relationship Graph Representation. 117

6.4 Difference between the Search Trajectory Graph and Parent Inheritance Relationship Graph. 118
6.5 Two Populations Genetic Algorithm Procedure 121

List of Tables

2.1 Performance Comparison of Exact and Non-exact Algorithms 9

2.2 The Effect of Three Different Parameter Configurations on 4 QAP instances. 16

2.3 Classification of Current Approaches in Automated Parameter Tuning 22

3.1 Run Time for Random-3-SAT instances 35

3.2 Position Types Property of Search Trajectory 38

3.3 Example of Sequence Alignment from a pair of instances 41

3.4 Threshold Value for Robust Sequence Alignment 42

3.5 Four instantiations of CluPaTra 45

3.6 Parameters for ILS Algorithm for Traveling Salesman Problem (TSP) 47

3.7 Parameters for SA-TS Algorithm for Quadratic Assignment Problem (QAP) 48

3.8 Parameters for TS Algorithm for Set Covering Problem (SCP) 49

3.9 Examples of Clusters from Different Parameter Configurations 52

3.10 Similarity Score of Randomly Selected Instance Pairs for Instances' Similarity Preservation 54

3.11 CluPaTra's Cluster Quality Comparison for Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP) 56

3.12 CluPaTra's Computational Time 57

3.13 CluPaTra's Performance Comparison of Three Classical COPs 58

3.14 CluPaTra's Testing Instances Performance using Different Cluster's Parameter Configuration 59

3.15 Correlation between Cluster Quality and Overall Performance 60
3.16 CluPaTra's Performance Comparison using Different Clustering Methods 60

4.1 Average length of Stem, Thorn and Petal 77
4.2 CluPaTra-II with SufTra and FloTra Cluster Analyses Comparison . 81
4.3 CluPaTra-II with SufTra and FloTra Computational Time Comparison 83
4.4 CluPaTra-II with SufTra and FloTra Performance Result Comparison 83
4.5 CluPaTra-II with SufTra and FloTra Comparison in Two Groups of Search Trajectories 84

5.1 Five Tuning Strategies in AutoParTune 90
5.2 AutoParTune Components Input Output Standard 98
5.3 Search Trajectory Generator and Target Algorithm Standard for AutoParTune 99
5.4 AutoParTune Performance Result Comparison for Classical COPs . . 101
5.5 Parameters for SA on Aircraft Spares Inventory Optimization Problem 102
5.6 Aircraft Spares Inventory Optimization Problem Performance Result . 103
5.7 Parameter Configurations for Aircraft Spares Inventory Optimization Problem 103
5.8 Parameters for Heuristic Algorithm on Theme Park Personalized Intelligent Route Guidance Problem 106
5.9 Parameters Configurations for Theme Park Personalized Intelligent Route Guidance Problem 107
5.10 Theme Park Personalized Intelligent Route Guidance Algorithm Performance Result using Scenario 1 108
5.11 Routes from Default Configuration and AutoParTune Strategy 3 . . . 108
5.12 Theme Park Personalized Intelligent Route Guidance Algorithm Performance Result using Scenario 2 110

6.1 Parameters for Two Population Genetic Algorithm on Generalized Assignment Problem 122
6.2 PeTra and PaRG Performance Result 123

A.1 Performance Comparison on TSP 133
A.2 Performance Comparison on QAP 134
A.3 Performance Comparison on SCP 135

Acknowledgments

This thesis would not have been completed without the academic, spiritual and moral supports from many individuals.

First, I would like to thank my supervisor, Prof. Lau Hoong Chuin, for his guidance, patience, enthusiasm, motivation, and care about my Ph.D life. Thank you for introducing me to this interesting field of meta-heuristics research - which eventually draw my intention. Your passion for research in different areas is always amazed me.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. David Lo, Prof. Zhu Feida, and Prof. Roland Yap, for their valuable comments and insightful suggestions on my Ph.D thesis.

I am fortunate to be able to work with Prof. David Lo and Prof. Zhu Feida. I appreciate their encouragement, ideas and experience sharing in doing research. I have benefited intellectually from collaborating with them.

I would also like to express my appreciation to three other collaborators, Zhi Yuan, Aldy Gunawan, and Prof. Steve Kimbrough, for contributions to Chapter 4 and Chapter 5 of the thesis. I would also sincerely thanks all my SIS friends. Thanks for the friendship and memories.

Lastly, I would like to thanks two most important people in my life, my father, Madsalih Hamid and my mother, Sim Tiaw Nio. Papa, Mama, thanks for raising me up since I was a baby until now. Without your love and encouragement, I would not be able to be what I am today.

Chapter 1

Introduction

1.1 The Journey for a High-Performance Algorithm

In the last few decades, there has been a dramatic rise in designing various meta-heuristic algorithms to solve computationally difficult Combinatorial Optimization Problems (COPs) in many practical applications. Meta-heuristic algorithms, such as Iterated Local Search (ILS), Tabu Search (TS) and Simulated Annealing (SA), are basically high-level procedures that coordinate simple search methods and rules to find good (often optimal) approximate solutions [23]. Even though meta-heuristic algorithms do not guarantee global optimality, they generally provide good solutions within reasonable time [20].

Creating a simple meta-heuristic algorithm for a given COP is often easy. All one needs to do is to instantiate certain meta-heuristic components, set some parameters with (usually) default values, and run the algorithm on the given instances [47]. However, to design a high-performance meta-heuristic algorithm in general is difficult [60]. One fundamental aspect that affects the performance of meta-heuristic algorithm is its parameter configuration. For example, a Tabu Search algorithm will perform differently with different tabu lengths and a Genetic Algorithm will perform differently with different mutation rates. Previous studies have indicated that finding performance-optimizing parameter configurations of meta-heuristic algorithms often

requires considerable effort [17, 3, 7, 60]. In [3], it is stated that only ten percent of development time is spent on algorithm design and testing, while the rest is spent on fine-tuning the parameter settings which, in many cases, is performed manually and in an adhoc fashion by the algorithm designer.

Given a meta-heuristic algorithm to solve a given COP, it also has been observed that different problem instances require different parameter configurations in order for the algorithm to find good solutions [39, 90, 115]. With numerous parameter configurations and a large number of instances, finding an instance-specific automated tuning manually not only takes a lot of time and effort, due to the enormous parameter configurations space, but also requires substantial knowledge of the algorithm and the problem itself. This tedious labour-intensive work gives rise to the need for *automated parameter tuning*. Automated parameter tuning is useful in a variety of contexts, such as improving meta-heuristic performance and trading human time with machine time [58]. Furthermore, it has been shown that automated parameter tuning often leads to better performance compared to manual parameter tuning [60].

There are several existing works on automated parameter tuning (also called *automated algorithm configuration* or *automated parameter optimization*) which can be classified into two parts: *one-size-fits-all* and *instance-specific*. On one hand, one-size-fits-all approaches focus on finding the best parameter configuration for the entire set (or distribution) of problem instances [17, 60, 71, 7, 3, 58]. These approaches use average quality or other statistical measures to determine the best parameter configuration. One common shortcoming of these approaches is that the single configuration produced may not be effective for large and diverse instances [3].

On the other hand, instance-specific approaches attempt to select the best parameter configuration for given instances [90, 56, 57, 64] using problem-specific features. Unfortunately, finding features itself is often tedious and domain-specific, requiring re-examination of features for each new problem. As an example, Instance-Specific Algorithm Configuration (ISAC) [64], an instance-specific algorithm configuration for finding instance-specific parameter configuration for arbitrary algorithms, uses

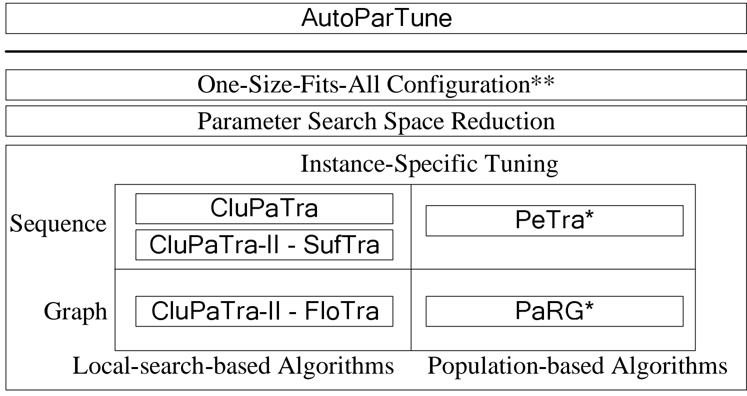


Figure 1.1: Summary of PhD Contributions

problem-specific features to identify characteristics of the problem instances. In [64], ISAC uses 24 features for Set Covering Problem [79] and 15 features for Satisfiability Problem [115].

To date, no single approach is generic enough to provide instance-specific parameter configurations. A one-size-fits-all approach is generic and may be applied to tune various applications in various COPs, but only provides a single best parameter configuration. On the other hand, an instance-specific approach tends to use problem-specific features that make the approach less general.

1.2 Summary of Contribution

In this thesis, we propose CluPaTra and CluPaTra-II which are **problem-independent** automated parameter tuning frameworks, based on clustering of instances using a new set of generic features extracted from the algorithm's search trajectory. The search Trajectory is defined as the path that a local search algorithm follows as it searches from an initial solution to its neighbour from one iteration to the next. We also implement a web-based workbench to enhance the usability of automated configurators. We demonstrate extensions of our approach on population-based algorithms. Thus, our major contributions, as illustrated in Fig. 1.1, are summarized as follows:

- **CluPaTra: Instance-specific Automated Parameter Tuning via Trajectory Clustering**

  We propose a **generic** instance specific automated parameter tuning framework by adopting a cluster-based approach for local-search-based algorithms. We also introduce the notion of an instance search trajectory as the problem-independent feature and represent it as a directed sequence. We apply a simple yet effective technique of sequence alignment to calculate a similarity score between a pair of instances based on its problem-independent feature.

- **CluPaTra-II: Pattern Mining Approaches for Instance-specific Automated Parameter Tuning**

  To boost CluPaTra's computational performance and improve the cluster quality and the quality of solutions, we introduce another technique for clustering instances. We model the features extraction in generic instance-specific parameter tuning as a frequent pattern mining problem and design two new algorithms: SufTra for search trajectory sequence and FloTra for search trajectory graph.

- **AutoParTune: Web-based Automated Parameter Tuning Workbench**

  To provide better decision support for tuning, we design a web-based workbench that integrates CluPaTra and CluPaTra-II. This workbench makes use of the parameter-space reduction method in [45] and the global (one-size-fits-all) parameter tuning in [60] and [19].

- **Instance-specific Tuning: Extension to Genetic Algorithms**

  We extend our methods for single point local search to population-based algorithms by introducing two new techniques: PeTra and PaRG, based on generic features pertaining to population dynamics.

1.3 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 briefly reviews the background knowledge related to automated parameter tuning problems including a formal definition of the problem and notations used in the rest of this thesis. In this chapter, we also review the recent trends of automated parameter tuning. Chapter 3 presents CluPaTra: our generic framework for instance specific automated parameter tuning that uses a simple yet effective model to calculate instance similarity using sequence alignment and introduces the problem independent feature based on the algorithm's search trajectory. Chapter 4 discusses the enhancement of CluPaTra by modeling the feature extraction as a pattern mining problem and proposes two new approaches, SufTra and FloTra. Chapter 5 then shows how our framework can be implemented onto a web-based automated parameter tuning workbench. In chapter 6, we present the extension of our approaches for population-based algorithms with two preliminary ideas based on population dynamics. Finally in chapter 7, we conclude our work and provide potential future directions.

1.4 List of Publications

The materials are presented in this thesis based on the following work conducted by the author together with her supervisors and colleagues.

Journal.

1. **Lindawati**, H.C. Lau, D. Lo, Clustering of Search Trajectory and Its Application to Parameter Tuning, *Journal of the Operational Research Society* (Special issue on Systems to build Systems), 64:1742-1752, 2013.

Conference Papers.

1. (FloTra) **Lindawati**, F. Zhu, H.C. Lau, FloTra: Flower-shape Trajectory Mining for Instance-specific Parameter Tuning, In *Proc. 10th Metaheuristics International Conference (MIC)*, Singapore, August 2013.

2. (SufTra) **Lindawati**, Z. Yuan, H.C. Lau, F. Zhu, Automated Parameter Tuning Framework for Heterogeneous and Large Instances: Case Study in Quadratic Assignment Problem, In *Proc. 7th Learning and Intelligent OptimizatioN Conference (LION)*, Catania, Italy, January 2013.

3. (CluPaTra) **Lindawati**, H.C. Lau, D. Lo, Instance-based Parameter Tuning via Search Trajectory Similarity Clustering, LNCS: In *Proc. 5th Learning and Intelligent Optimization (LION)*, pp. 131-145, 2011.

4. A. Gunawan, H.C. Lau, **Lindawati**, Fine-tuning Algorithm Parameters using the Design of Experiment Approach, LNCS: In *Proc. 5th Learning and Intelligent Optimization (LION)*, pp. 278-292, 2011.

5. S. Kimbrough, A. Kuo, H. C. Lau, **Lindawati**, D. H. Wood, On Using Genetic Algorithms to Support Post-Solution Deliberation in the Generalized Assignment Problem. In *Proc. 8th Metaheuristics International Conference (MIC)*, Hamburg, Germany, July 2009.

Workshop Paper.

1. **Lindawati**, H.C. Lau, F. Zhu, Instance-specific Parameter Tuning via Constraint-based Clustering, *ECAI Workshop on COmbining COnstraint solving with MIning and Learning (CoCoMile)*, 2012.

Chapter 2

Background

Automated parameter tuning is a rapidly evolving field that aims to overcome the limitations and difficulties associated with manual parameter tuning. Many approaches have been introduced to address this problem, including meta-heuristic and machine learning. The successful implementation of these tuning approaches for many Combinatorial Optimization Problems (COPs) emphasize its impact on meta-heuristic algorithm performance.

In this chapter, we provide background materials used for the rest of this thesis and discuss related works on automated parameter tuning. We start with a brief introduction of the Combinatorial Optimization Problem (COP) and its associated algorithms. We then provide a review of meta-heuristic algorithms for COP and the challenges in designing meta-heuristic algorithms. We continue to formally define (instance-specific) parameter tuning problems and introduce its notations. We discuss existing approaches for automated parameter tuning and finally provide a summary of the chapter.

2.1 Combinatorial Optimization Problem

Many problems, both theoretical (classic) and practical (real-life), focus on finding the *"best"* solution [89]. These problems can be categorized into two types: problems whose solutions are encoded with continuous variables, and problems whose solutions

are encoded with discrete variables. While both categories provide interesting materials for research, our research is oriented solely on the latter. Such problems are also called Combinatorial Optimization Problems (COPs). In the COPs, we are looking for an optimal solution from a finite solution set which is typically an integer number, a subset, a permutation or a graph structure. As in [20], COP is formally defined as follows:

Definition 1 (Combinatorial Optimization Problem [COP]) *Given a set of variables* $X = \{x_1, ..., x_n\}$*, a variable domain* D *for each variable* x *in* X*, constraints among variables, an* objective function f *to be minimized (or maximized) where* $f : D_1 \times ... \times D_n \rightarrow \mathbb{R}$*, a set of all possible feasible assignments* S=$\{s = \{(x_1, v_1), ..., (x_n, v_n)\} | v_i \in D_i$*, s satisfies all the constraints*$\}$*, the* COP *(S, f) is to find a solution* $s^* \in S$ *where* $f(s^*) \leq f(s) \forall_S \in S$.

S is also called a search (or solution) space or fitness landscape where each element in S can be seen as a candidate solution. s^* is called a globally optimal solution of (S, f) and the set of $S^* \subseteq S$ is called the set of globally optimal solutions. Finding a globally optimal solution to some problems may be challenging, but it is often possible to find a solution $\hat{s}$ which is relatively best in its *neighborhood* [89]. The neighborhood of solution s is a set of solutions that are "close" in some sense to solution s. The closeness is defined using a neighborhood structure such as $2 - change$ neighborhood structure where $N_2(s) = \{g : g \in S$ and g can be obtained from s by swapping two variables in X from $s\}$.The best solution $\hat{s}$ is called locally optimal which is defined as follows:

Definition 2 (Locally Optimal [LO]) *Given a COP* (S, f) *and neighborhood* N*, a feasible solution* $\hat{s} \in S$ *is called* locally optimal *with respect to* N *if* $f(\hat{s}) \leq f(g)$ *for all* $g \in N(\hat{s})$.

COPs have many application in real-life, e.g. the Traveling Salesman Problem (TSP) in VLSI chip fabrication and X-ray crystallography [62], the Quadratic Assignment Problem (QAP) in backboard wiring and campus planning [21], the Set Cover-

Table 2.1: Performance Comparison of Exact and Non-exact Algorithms

Algorithm	Best Found Objective Value	Time (sec)
Exact Algorithm (Branch and Bound)	1,150	2,947.32
Non-Exact Algorithm (Meta-heuristic)	1,252	1.90

ing Problem (SCP) in crew scheduling and railway application [25] and the Generalized Assignment Problem (GAP) in fixed-charge plant location models and resource scheduling [28]. With its many practical uses, it is important to have good solvers for COPs. Unfortunately, finding a good solver for several COPs is a real challenge due to the hardness of these COPs. Many important COPs are NP (Nondeterministic Polynomial time) complete [40] where a complete exhaustive enumeration solver might need exponential computation time in the worst-case [20], and this is the motivation for research in optimization algorithms.

2.2 Algorithms for Solving COPs

In general, algorithms for solving COPs are classified into two types: exact or non-exact algorithms [20]. Exact algorithms are guaranteed to find an optimal solution in bounded time, whereas non-exact algorithms sacrifice the guarantee of finding optimal solution and settle for obtaining good quality solutions in a significantly reduced amount of computational time. For many large instances of NP-complete problems, such as TSP where the largest instance solved is of size 85,900 [1] and QAP where the largest instance solved is of size 40 [22], exact algorithms require a lot of time to generate the optimal solution due to their very high computational time. Therefore, we turn to non-exact algorithms for a good enough solution within a reasonable computational time. An example of exact and non-exact algorithm performance for a QAP instance with the size of 15 are shown in Table 2.1.

2.2.1 Exact Algorithms

For some COPs, it is possible to design algorithms that are significantly faster than a traditional exhaustive search, although still not in polynomial time. This class of

algorithms are called exact (or complete) algorithms. They are complete in a sense that the existence of feasible and then optimal solution(s) can be determined with certainty once the algorithm has successfully terminated. Examples of exact algorithms are Branch and Bound (B&B) [110] and Constraint Programming (CP) [94].

The B&B algorithm searches the complete space of solutions for a given problem to find the best solution. However, explicit enumeration is normally impossible due to the exponentially increasing number of potential solutions. The use of bounds for the function to be optimized combined with the value of the current best solution enables the B&B algorithm to search parts of the solution space only implicitly.

CP works basically by stating the variables in the form of constraints. The constraints used in CP are of various kinds: those used in constraint satisfaction problems (e.g. "A or B is true"), those solved by the simplex algorithm (e.g. "x = 6"), and others. Constraints are usually embedded within a programming language or provided via separate software libraries.

Although exact algorithms are able to find optimal solutions, they are faced with computational bottlenecks especially for large instances of NP-complete problems. This often leads to computational times that are too high for practical purposes. If optimal solutions cannot be achieved, the other possibility is to trade optimality for efficiency. In other words, the guarantee of finding optimal solutions can be sacrificed for the sake of getting good solutions in polynomial time. A class of non-exact algorithms seeks to obtain this goal.

2.2.2 Non-Exact Algorithms

In non-exact algorithms, optimality is not guaranteed but good quality solutions can be found in polynomial time (either in the worst case or on average) [32]. In practical applications, we are often faced with extremely large instances with very limited time, for that near-optimal solutions are often good enough. Unlike exact algorithms, non-exact algorithms are unable to confirm the existence of an optimal solution once they have successfully terminated. We also cannot measure the absolute quality of the

found solutions with respect to the optimality. In the non existence of optimality, the goodness of found solutions are measured subjectively by the algorithm developers or users.

Non-exact algorithms can be divided in two: approximation and (meta-)heuristics algorithms. Approximation algorithms have a proven performance where the solution is within an approximation ratio of ε from an optimal solution. Examples of approximation algorithms are MST-Prim algorithm for 2-approximation TSP and greedy approximation algorithm for SCP [32].

Heuristic algorithms are defined as simple techniques which find good solutions at a reasonable computational cost (low-order polynomial time) [47]. They are usually based on the characteristics of the good solutions. The major limitation of heuristic algorithms is that they have the tendency to explore only a small search space and are easily trapped in a local optimal space. To solve these limitations, a more effective method can be implemented to guide the heuristic algorithm, in what are known as meta-heuristic algorithms.

Meta-heuristic algorithms are iterative generation processes which guide a basic heuristic methods by intelligently combining different concepts for exploring and exploiting the search space [20]. Most meta-heuristic algorithms have learning strategies that are used to structure information in order to find efficiently near-optimal solutions. Although meta-heuristic algorithms do not guarantee optimality, they provide good enough solutions in relatively short computational time. Examples of meta-heuristic algorithms are Tabu Search (TS) [42] and Genetic Algorithm (GA) [82].

2.3 Meta-heuristic for Solving COPs

Meta-heuristic algorithms have been introduced to solve many COPs such as Lin-Kernighan algorithm for TSP [52] and Robust Tabu Search algorithm for QAP in instances with essentially no strong structure [104] and Iterated Local Search for QAP in more structured instances [101]. Generally, meta-heuristic algorithms are classified

into two types: local-search-based and population-based. This classification is based on the number of solutions used and explored at the same time.

2.3.1 Local-search-based Algorithms

Local-search-based algorithms (or trajectory algorithms in [20]) are a class of algorithms that work on a single solution for each iteration. Local-search based algorithms start from an initial solution and move to a better solution in the search space by applying local changes, until a solution deemed optimal is found or a maximum time allowed is exceeded. These solutions' movements form a trajectory in the search space. Characteristics of the trajectory provide information about the behavior of the algorithm and its effectiveness with respect to the instance that is tackled [20]. The local-search-based algorithms used in this thesis are Simulated Annealing (SA) [68, 29], Tabu Search (ST) [42], and Iterated Local Search (ILS) [77].

Simulated Annealing (SA) is a probabilistic method proposed in [68, 29] which is known to be the oldest meta-heuristic algorithm and the first that has an explicit strategy to escape from local minima [20]. It is modeled after the physical process of annealing metals (cooling molten metal to solid minimal-energy state). SA allows worse moves (uphill moves) in order to escape from a local optimal using a certain probability. In the beginning of the search, the probability for uphill moves is high to allow search space exploration. The probability is slowly decreased to lead the search to a convergence (local optimal). SA works by first generating an initial solution and initializing a temperature parameter T. For each iteration, it randomly samples a solution s' based on the neighborhood structure of the current solution s and accept the new solution based on the objective function of s', s and a probability which is a function of T. The temperature T is decreased based on a cooling schedule. A slow cooling schedule guarantees the convergence to a global optimal. But for some of the COPs the cooling schedule is too slow [20].

Tabu Search (TS) is the most cited and used meta-heuristic algorithm for COPs [20]. TS works by maintaining a history list of forbidden moves (called tabu list)

[42]. Tabu list keeps the most recently visited solutions and forbids moves towards them to prevent endless cycling and forces the search to explore the search space by accepting even uphill moves. A set of aspiration criteria is defined to overwrite the tabu conditions where the selected solutions are better than the current best one. TS starts by generating an initial solution. At each iteration the best solution from the neighborhood, which is not in tabu list, is chosen as the new current solution. This solution is then added to the tabu list and one solution in the tabu list is removed in FIFO order. A tabu move is allowed if the aspiration criteria are met. The length of tabu list is controlled using the tabu list parameter.

Iterated Local Search (ILS) is a simple but powerful meta-heuristic algorithm [77]. It starts from an initial solution and applies a local search until it finds a local optimal. ILS then perturbs the solution and restarts the local search. There are four important components of ILS: initial solution generation, local search, perturbation and acceptance criteria. A good initial solution is very important so as to arrive at high-quality solutions as soon as possible. The standard way to generate the initial solution is either randomly or by greedy construction. The local search algorithm can be treated as a black box but ILS performance is quite sensitive to the choice of this local search. In practice, there are many different local-search-based algorithms that can be used as the local search component. The perturbation is very important to guide the search in ILS. Too small perturbations might not enable ILS to escape from the basins of attraction while too strong perturbations would make ILS similar to a random restart local search. The last component, acceptance criteria, is used to control the search balance between intensification and diversification. Intensification refers to the exploration of the accumulated search experience so far while diversification refers to the exploration of the search space [20].

2.3.2 Population-based Algorithms

In population-based algorithms, each iteration involves a set (i.e. a population) of solutions. Population-based algorithms provide a natural, intrinsic way to explore

the search space [20]. The most studied population-based algorithms for COPs are Evolutionary Computation (EC) [8].

Evolutionary Computation (EC) is inspired by nature's capability to evolve living beings well adapted to their environment. An EC uses operators called recombination or crossover to recombine two or more individuals to produce new individuals. Other operators in EC are mutation, inversion, and selection. Example of EV is Genetic Algorithm (GA) [82].

Genetic Algorithm is a meta-heuristic algorithm that moves from one population of chromosomes (e.g., strings of ones and zeros, or bits) to a new population by using ”natural selection” together with the genetics-inspired operators of crossover, mutation, and inversion [82]. Each solution is represented by a chromosome that consists genes (e.g., bits) as an instance of a particular allele (e.g., 0 or 1). Chromosome is an abstract representation of the possible solution. Each chromosome has a value corresponding to its fitness function, which evaluates how good the candidate solution is in terms of its objective value. The optimal solution is the one which maximizes (or minimizes) the fitness function. A set of reproduction operators is then applied directly on the chromosomes to perform mutations and recombinations.

As described in [98], the GA works as follows. It starts by generating an initial population of chromosomes. This first population must offer a wide diversity of genetic materials which is generally generated randomly. Then, the GA loops over an iteration process to make the population evolve as illustrated in Fig. 2.1. Each iteration in GA consists of the following:

1. **Selection;** where chromosomes in a population are selected as parents for reproduction process. The selection process is done randomly using a probability function depending on its relative fitness function. The good chromosomes are more often being selected than poor ones.

2. **Reproduction;** in which new offsprings are created by selected parent chromosomes using crossover mechanism. Crossover exchanges subparts (bits) of two selected chromosomes, mimicking biological recombination between two

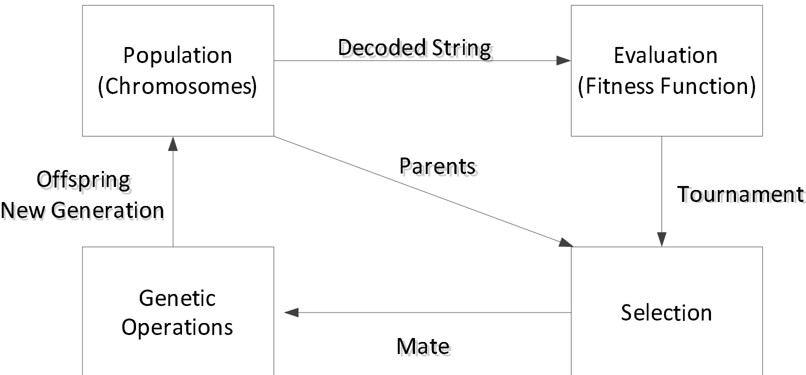


Figure 2.1: Genetic Algorithm Cycles.

single-chromosome organisms. To diversify the offsprings, mutation and inversion mechanisms may be applied after crossover. Mutation randomly changes the allele values of some locations in the chromosome while inversion reverses the order of a contiguous section of the chromosome, thus rearranging the order in which genes are constructed.

3. **Evaluation;** at this stage, the fitness function of the new offsprings is being evaluated.

4. **Replacement;** which is the last step where chromosomes from the old population is replaced by new offsprings according to the ”survival of the fittest” procedure.

2.3.3 Challenge in Designing Meta-heuristic Algorithms for COPs

Most often, even a quick-and-dirty implementation of a meta-heuristic is able to obtain fairly good results for solving COPs [16]. All one needs to do is to instantiate certain meta-heuristic components, set some parameters with (usually) default values, and run the algorithm on a set of COP instances [47]. But if state-of-the-art results are needed, some extra efforts are often necessary to optimize the meta-heuristic algorithm.

One factor that the meta-heuristic algorithm effectiveness hinges upon is its parameter configuration. Different COP's problems require different configurations so that meta-heuristic algorithms perform well. It also has been observed that different instances from certain COPs require different parameter configurations in order for

Table 2.2: The Effect of Three Different Parameter Configurations on 4 QAP instances.

Instances	Config. I	Config. II	Config. III
tai40a	1.4	**1.0**	2.0
tai60a	1.7	**1.6**	2.2
tai40b	9.0	9.0	**0.0**
tai60b	2.1	2.9	**0.3**

the algorithm to find good solutions (e.g. [39, 90, 115]). Table 2.2 provides an example of the performance from three different parameter configurations for 4 Quadratic Assignment Problem (QAP) instances as presented in [49]. Table 2.2 shows that the first 2 instances perform better in Configuration II while the rest perform better in Configuration III. The differences between the performances are significant.

Despite its importance, finding the optimal parameter configuration is often a difficult, tedious and unsatisfying challenge. Previous studies revealed that finding performance-optimizing parameter configuration of meta-heuristic algorithms often requires considerable efforts [17, 3, 7, 60]. In [3], it is also stated that only 10% of the time is spent on algorithm designing and testing; while the rest of the development time is spent on fine-tuning the parameter configurations. In many cases, this process is performed manually by the algorithm designer.

With the large parameter configuration space and the large number of instances, finding a configuration, especially instance-specific configuration, manually takes a lot of time and effort due to the enormous possible number of parameter configurations-instances matching. As an example, for a target algorithm with 4 parameters, where each parameter has 20 possible values, 100 instances, and assuming the time needed to run the target algorithm for each instances is 1 second, to manually try all parameter configuration will need approximately 185.2 days. Thus, a smart *automated parameter tuning* method is needed.

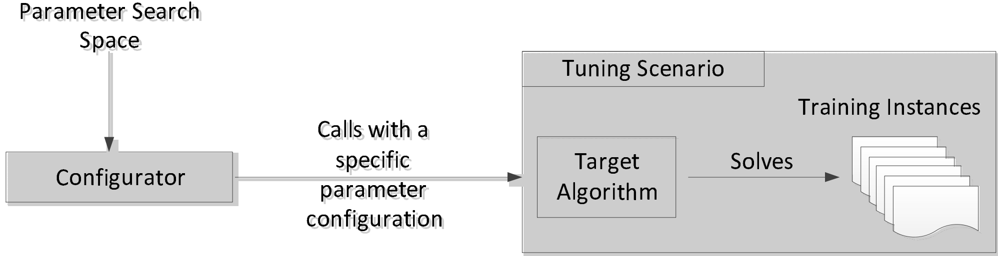


Figure 2.2: Tuning Scenario

2.4 Automated Parameter Tuning Problem

The automated parameter tuning problem is informally defined as: given a parametric algorithm with a set of parameter configurations and a set of training and testing instances, find a parameter configuration under which the algorithm achieves the best possible performance. As shown in [60], the tuning scenario is illustrated in Fig. 2.2. Automated parameter tuning is useful in a variety of contexts, such as improving meta heuristic performance and trades human time for machine time [58]. Furthermore, it has been shown that automated parameter tuning often leads to highly better performance compared to manual parameter tuning [60]. To avoid confusion between an algorithm whose performance is being optimized and an algorithm used to tune it, we refer to the former as the *target algorithm* and the latter as the *configurator*. Before we define the problem in greater detail, we introduce some notations and the performance metric.

Let:

- $\mathcal{A}$: be the target algorithm with n number of parameters to be tuned
- I: be the given set of (training and testing) problem instances
- x_i: be the parameter that can assume a value taken from a (real or integer value) interval $[a_i, b_i]$
- **x**: be the parameter configuration (i.e. a point in the parameter space)
- Θ: be the feasible region of all parameter configurations (parameter space)

- $Best$: be the best known value for each instance i in I. For benchmark instances with a known global optimal or best known value, we use it as OPT, while for new instances, we use the target algorithm's best found solution.

We measure the target algorithm performance based on the quality of the solutions. The performance metric is defined as follows:

Definition 3 (Performance Metric [$\mathcal{H}$]) *Let i be a problem instance, and $\mathcal{A}_{\boldsymbol{x}}(i)$ be the objective value of the corresponding solution for instance i obtained by a target algorithm $\mathcal{A}$ when executed under configuration* ***x****. Let $Best(i)$ denote the best known value for instance i. $\mathcal{H}_x(i)$ is formulated as:*

$$H_x(i) = \frac{(|Best(i) - A_{\boldsymbol{x}}(i)|)}{Best(i)} \tag{2.1}$$

Unlike standard optimization problems, function $\mathcal{H}$ is a meta-function on **x** and it is highly non-linear and very costly. Using the performance metric $\mathcal{H}$, we formally define the parameter tuning problem for the target algorithm that minimizes its objective value as follows.

Definition 4 (Parameter Tuning [PT]) *Given a set of instances I, a parameter configuration space Θ for a target algorithm $\mathcal{A}$ and a performance metric $\mathcal{H}$, the* PT *problem can be formulated as an optimization problem as follows:*

$$\boldsymbol{x}^* = argmin\, \mathcal{H}_{\boldsymbol{x}^*}(i) \tag{2.2}$$

$$subject\ to\ x^* \in \Theta \tag{2.3}$$

The central topic of this thesis revolves around the instance-specific parameter tuning problem. The purpose of tuning is not to find best configuration with good performance for all the problem instances, but to find the configuration that fits best

for each instance. Using the same notation as for the parameter tuning problem, we define the instance-specific parameter tuning problem as follows:

Definition 5 (Instance-Specific Parameter Tuning [ISPT]) *Given a set of instances I, a parameter configuration space Θ for a target algorithm $\mathcal{A}$ and a performance metric $\mathcal{H}$, the* ISPT *problem is to find a parameter configuration $x^* \in \Theta$ for each $i \in I$ such that $\mathcal{H}_{x^*}(i)$ is minimized over Θ.*

Instead of finding a parameter configuration for each instance, the ISPT problem can be approximated in a cluster-based manner in which problem instances are grouped into clusters and the parameter configuration is computed for each cluster [64]. We adopt this approach and focus on finding the best cluster and parameter configuration for each instance. We define the problem of cluster-based instance-specific parameter tuning as follows.

Definition 6 (Cluster-based Instance-Specific Parameter Tuning [C-ISPT]) *Given a set of instances I, a parameter configuration space Θ for a target algorithm $\mathcal{A}$, a performance metric $\mathcal{H}$, the* C-ISPT *problem is to find a clustering π of all instances of I and a parameter configuration $\boldsymbol{x} \in \Theta$ for each cluster of π such that (I) cluster quality of π is maximized; and (II) the average $\mathcal{H}_{x^*}(i)$ for each cluster is minimized over Θ.*

We measured the cluster quality using *extrinsic* method [51] for instances that has ground-truth clusters. *Extrinsic* methods compare the clusters against the known class labels or *ground-truth* clusters (i.e. the set of clusters which represents the ideal/optimal clustering). We define the cluster quality as follows:

Let this score be denoted as $\mathcal{Q}_c$, which is the average value of the *training clusters quality score* $\mathcal{Q}_{train}$ and *testing instances mapping score* $\mathcal{Q}_{test}$, defined as follows:

Let I (resp. I_t) be a set of training (resp. testing) instances, C be the set of clusters generated from the training phase and C_g be the *ground-truth* clusters. Each cluster in $c \in C$ has an associated *home* cluster $c_g \in C_g$ which contains the largest number of instances contained in c (ties broken arbitrarily).

Definition 7 (Training Clusters Quality Score $\mathcal{Q}_{train}$) *For each cluster $c \in C$, let $max(c)$ count the number of instances in the cluster which also belong together in the associated home cluster. $\mathcal{Q}_{train}$ is defined as the sum of $max(c)$ over all $c \in C$ divided by the number of instances in I.*

Definition 8 (Testing Instance Mapping Score $\mathcal{Q}_{test}$) *For each instance $i \in I_t$, we say that i is "matched" if it is mapped to a cluster $c \in C$ whose home cluster $c_g \in C_g$ also contains i. $\mathcal{Q}_{test}$ is defined as total number of such matches divided by the number of instances in I_t.*

2.5 Literature Review on Automated Tuning

The quest for finding a technique for smart automated parameter tuning started since the early 1990s. Some approaches are designed for a specific target algorithm on a specific problem such as the fine-tune technique for the corridor method on block relocation problem [26] and Tabu Search (TS) on the telecommunications network design problem [113]. In the corridor method on the block relocation problem, [26] tunes the parameters using Response Surface Methodology (RSM), a well-known technique in Design of Experimental (DoE) methodology. RSM represents the parameters as a planar model and uses the steepest ascent method to change the parameters to a promising range until it finds a local minimum. While in [113], they employ two standard statistical tests (Friedman's test and Wilcoxon's test for paired observations) to improve a specific TS algorithm in [112] for telecommunications network design. Although these two techniques are general and can conceptually be adapted on different target algorithms, applying it to target algorithms is not as simple.

Rather than focusing on a specific target algorithm, some approaches attempt to find the best configuration for generic target algorithms (such as [33, 17, 3, 58, 90, 64]). They can be used to fine tune target algorithms with categorical or numerical parameters. In this subsection, we discuss the current trends for these approaches.

2.5.1 Classification of Current Approaches

There exist different ways to classify and describe the current approaches in automated parameter tuning. Depending on the aspects in which they can be differentiated, several classifications are possible, each of them being the result of specific viewpoints. We briefly review two aspects to classify the current approaches.

One-Size-Fits-All vs. Instance-Specific

The most obvious way to categorize the current approaches is based on the strategy to handle diverse instances. There are one-size-fits-all approaches (or instance-oblivious in [64] or global tuner in [71]), like F-Race [17, 18] and ParamILS [61, 60], vs instance-specific approaches such as auto-WalkSAT [90] and ISAC [64].

One-size-fits-all approaches attempt to find the parameters resulting in the best average performance of a target algorithm on all training instances. They ignore the instance diversity and use a specific statistic measurement (such as mean or standard deviation) measured over the entire set of problem instances. This is the main drawback of one-size-fits-all approaches because not all instances yield to the same parameter configuration [107, 84]. This observation supports the No Free Lunch theorem [111] that states that no single algorithm can be expected to perform optimally over all instances.

With this observation, instance-specific tuning approaches attempt to generate parameter configuration for each instance by computing a set of features from the training instance set. The instance-specific tuning approaches usually assume the existence of a set of instance-specific features for different COPs, such as [64] uses 24 specific features from [79] for the Set Covering Problem.

Model-based vs Model-free

Another aspect that can be used for classifying current approaches is the existence of a statistical model. Approaches that use statistical models to guide the tuning process are called model-based approaches such as CALIBRA [3] and SMAC [58], while

Table 2.3: Classification of Current Approaches in Automated Parameter Tuning

Category		Common Technique	Example
One-size-fits-all	Model-Free	Machine Learning or Meta-Heuristic technique	F-Race [17, 18] and ParamILS [61, 60]
	Model-Based	Design of Experiment or statistic function	CALIBRA [3] and SMAC [58]
Instance-Specific	Model-Free	Clustering	ISAC [64]
	Model-Based	Regression and interpolation	Auto-WalkSAT [90] and HyDra [114]

other approaches that do not have a specific model for their tuning process are called model-free like ParamILS [61, 60] and GGA [7]. Some model-free approaches can handle a large number of numerical and even categorical parameters. Model-based approaches, on the other hand, offer statistical insights into the correlation of parameters with regard to algorithm performance.

Using the following two aspects: (1) the strategy to handle diverse instances and (2) the use of statistical models to explain the tuning process, we divide the current approaches into four groups as summarized in Table. 2.3.

2.5.2 Analyzing Different Approaches

In this subsection, we discuss recent and notable different approaches for each category. We start by introducing one-size-fits-all approaches with and without statistical models. We continue with the instance-specific approaches. We then give remarks for instance-specific approaches.

One-Size-Fits-All Model-Free

An early approach in this category is *F-Race*, proposed by [17]. F-Race is inspired by the AI machine learning literature for ”model selection through cross-validation”. F-Race works by empirically evaluating a set of candidate configurations and discarding bad configurations as soon as statistically sufficient evidence is gathered against them. When F-Race was first introduced in [17], candidate configurations were obtained by a full factorial design on parameter space which contains all combinations of values

for a set of discrete (or discretised) parameters. This severely limits the size of the configuration space such that F-Race can only be used to tune target algorithms with a small number of possible parameter configurations.

Two more recent variants of F-Race, Sampling F-Race and Iterative F-Race, have been introduced to address this limitation [10, 18]. Sampling F-Race randomly selects r number of samples and uses it as an initial set of configurations in a standard F-Race, while iterative F-Race uses an iteration procedure to refine its probabilistic model to a sample set of configurations. All three versions of F-Race assume that all parameters are numeric. Hence, F-Race can only be used to tune numerical parameters. The latest version of F-Race [19] overcomes this limitation by sampling categorical parameter values from discrete probability distributions. But the performance of F-Race is really dependent on the probability distributions used.

Other model-free approaches are ParamILS [61, 60] and GGA [7]. Both approaches apply meta-heuristic algorithms for tuning the parameters: ParamILS uses Iterated Local Search (ILS) while GGA uses Genetic Algorithm (GA). They also claim that they can be used to configure a very large number of parameters.

ParamILS utilizes ILS to explore the parameter configuration space in order to find a good parameter configuration for the given training instances. ParamILS has two different variants: BasicILS and FocusILS where the main difference is in the technique used to assess the performance of a configuration. BasicILS performs a fixed number of runs based on user defined values of the target algorithm using the same instances and pseudo random number seeds, while FocusILS evaluates configurations using few target algorithm runs and subsequently performs additional runs to obtain precise performance estimates for promising configurations. Using this technique, FocusILS is able to quickly focus on promising configurations rather than being trapped in evaluating poor configurations. To expedite its overall search process, ParamILS applies an adaptive capping technique to limit the time spent in evaluating poor configurations measured by the performance observed for the current incumbent.

ParamILS has been successfully applied to tune a broad range of high-performance

algorithms with a large number of parameters. Examples of the problems are the Satisfiability (SAT) Problem [61, 60], Mixed Integer Programming (MIP) Problem [60], Post Enrollment Course Timetabling problem for Track 2 of the International Timetabling Competition (ITC2007) [30], Planning System [37], Answer Set Programming (ASP) for homogeneous instances [41], etc. To date, ParamILS is the most powerful tuning algorithm and the only tuning algorithm that has been implemented for tuning a very large number of parameters such as CPLEX with 80 parameters [60].

However, we notice that ParamILS has two main limitations as follows. First, it can only be used to tune discrete parameters. For continuous parameters, a discretization mechanism should be performed beforehand. Second, ParamILS is dependent on the default (or initial) parameter configuration given by the user or from random initialization. ParamILS moves from the initial parameter by changing one parameter value at a time. If ParamILS is provided with a good default parameter, it gives a better performance.

GGA (or gender-based genetic algorithm) is a robust, parallel Genetic Algorithm to configure algorithms automatically. It uses the notion of gender separation (competitive and non-competitive population) to balance exploitation and exploration, and applies different selection pressure for these two populations. For competitive population, candidate configurations have to compete on a collection of training instances. The parameter configuration that yields best overall performance are then mated with candidates from the non-competitive population. The configuration with the poorest performance is removed. GGA also exploits the dependencies of parameters by applying a ”variable tree” structure which indirectly defines the cross-over operator.

GGA is claimed to be remarkably successful in tuning existing solvers, often outperforming ParamILS on some COPs [7], but GGA has only been implemented in a limited number of problems. One limitation of GGA is that it needs a very large tuning budget to avoid over-tuning where the configuration works on training instances but gives a bad result on testing instances [120].

Lau et. al. [71] proposed a Randomized Convex Search (RCS) which uses a

randomized scatter search technique. The underlying assumption of RCS is that the points lie inside the convex hull of a certain number of the best points (parameter configurations). RCS can be used to tune both discrete and continuous parameter values.

One-Size-Fits-All Model-Based

For model-based approaches, one idea is proposed by Coy et al. [33] using a procedure based on experimental design and gradient descent. They suggest that computing a good parameter set for few of instances and averaging all parameters results in parameters that would work well for the general case. It is used to tune two local search algorithms, for solving Vehicle Routing Problems, based on a variant of Lagrangian relaxation and an edge exchange procedure. The approach is reasonably effective in terms of solution quality. They also highlight that the response surface and average setting might not be appropriate if the class of problems is too broad. The approach, however, suffers once more parameters need to be set or if these parameters are not continuous.

Another model-based approach which was introduced is CALIBRA, proposed in [3], which combines statistical experimental design (design of experiment) and local search procedure. CALIBRA automatically calibrates parameter values from a given pre-defined range for each parameter. CALIBRA employs a full factorial 2^k design and a Taguchi fractional factorial design followed by a local search procedure to iteratively narrow down the range of parameter values until it converges to a ”local minimum”. Some notable limitations of CALIBRA are: (1) it only tunes up to 5 parameters; (2) if the given parameter value ranges are too small, CALIBRA is quickly trapped in a ”local minimum” of the configuration space and (3) it focuses on the main effects of parameters without exploiting the interaction effects between the parameters.

The most current model-based approach is SMAC, proposed in [58]. SMAC is an improved model-based technique which can tune multiple problem instances at a

time, which is an extension of their earlier work SPO+ [59] that can only deal with a single problem instance at a time. This line of work is based on the Sequential Parameter Optimization framework [12, 13]. It constructs predictive performance models to focus attention on promising regions of a design space. SMAC aims at tuning target algorithms with continuous and categorical parameters for sets of problem instances. The authors claims that SMAC can be used to configure a very large number of parameters. The performance of SMAC is highly depended on the accuracy of the performance model used to capture the interrelations of the parameters.

Instance-Specific Model-Free

There are not many approaches for instance specific approaches with model-free manner. One (and perhaps the only one) instance-specific with model-free manner is ISAC (Instance-specific algorithm configuration) proposed by [64]. ISAC is the first method that uses clustering to approximate instance-specific configurations. It extends the stochastic offline programming framework [79]. ISAC works by first running a clustering method, $g - means$, to cluster instances using problem specific features and then find a good parameter configuration for each cluster using a one-size-fits-all configurator.

It is interesting to note that ISAC does not make use of an explicit formulation (such as linear or Gaussian regression) that maps instances to clusters, which may be very hard if not impossible to derive. Instead, ISAC exploits the instance-features relationship that correlates with algorithm performance. Instances are clustered based on these features using predictive modeling. This form of clustering preserves rich features that represent the individual instances within it.

ISAC has been implemented in various problems problem such as Set Covering Problem (SCP) [64], Mixed Integer Programming (MIP) [64], Satisfiability (SAT) Problem [64, 70], Constraint Programming (CP) [70] and Black Box Optimization (BBO) problem [2]. In these problems, ISAC shows promising results which prove the effectiveness of clustering-treatment in solving parameter tuning problems [71, 79,

64].

One weakness of ISAC is that it uses problem-specific features and assumes that there exist a collection of problem-specific features for each instance that can be used to correctly identify its structure, and thus use to identify the sub-types of the problems. Hence, for new COPs or industrial cases without a known set of features, ISAC cannot be implemented.

Instance-Specific Model-Based

As in ISAC, the approaches in this category rely on instance-specific features. These approaches use regression or interpolation to fit a model that will determine the solver's strategy. Several approaches have been proposed using these techniques such as: auto-WalkSAT [90], empirical hardness model [56, 57], and Hydra [114].

Auto-WalkSAT calculates an estimation of the invariant ratio of a provided SAT instance and uses this value to set the noise value, or how frequently a random decision is made. It uses recursive bracketed search (golden section search) and parabolic interpolation to adaptively search the invariant ratio of the solution from the global minimum without exactly solving the satisfiability formula. These values can be used to guide a search for the minimum ratio which in turn leads to an estimated optimal noise level. It then return this estimated optimal noise level to the provided stochastic algorithm (target algorithm). Auto-WalkSAT is shown to be effective on four DIMACS benchmarks, but fails for those problems where there is no relationship between invariant ratio and optimal noise parameter.

Hutter et. al [56, 57] proposed an *empirical hardness model* to predict the runtime of search algorithms for hard combinatorial problems. This approach can handle both continuous and ordinal (but not categorical) parameters. The model predicts algorithm runtime for the problem instances at hand and then simply selects the configuration that minimizes the prediction by using the linear regression method, Bayesian linear regression. This Bayesian linear regression is used to learn mapping from features into a prediction of runtime. Based on this mapping for given instance features, a

parameter set that minimizes predicted runtime is searched for.

A recent approach in this category is Hydra. It works by combining automated algorithm configuration and portfolio-based algorithm selection. It automatically builds a set of solvers with complementary strengths by iteratively configuring new algorithms using regression to be used in its portfolio. To date, Hydra has only been applied to optimize the target algorithms runtime performance and not the quality of their solutions.

All the above approaches in instance-specific tuning using a model-based manner depend on accurately fitting a model from the features to a parameter. It is intractable and requires a lot of training data when the features and parameters have non-linear interactions. These approaches may need more tuning budget compared to ISAC.

Remarks on Instance-specific Approaches

While providing a significant quality improvement compared with one-size-fits-all approaches, these instance-specific approaches can only be used for problems that have a set of instance features. Unfortunately, finding instance-specific characteristics/features is nothing easy, which requires profound knowledge of the algorithm as well as the problem itself. Consequently, an interesting research problem is to develop a generic instance-specific automated parameter tuning scheme that is problem-independent and yet can perform as well as those exploiting problem-specific features.

2.5.3 Further Relevant Research

As described in [47], two problems closely related to automated parameter tuning problems are the algorithm selection problem and dynamic parameter adjustment. The goal in the algorithm selection problem is to correctly select an algorithm that yields the best performance for a particular instance. For example, [43] proposes to combine several algorithms into a portfolio, and run them in parallel or interleave them on a single processor. This approach is more robust than any of the individual solvers. Another well-known approach is SATzilla in [115] which uses an empirical hardness

model to select among their candidate solvers. The empirical hardness model is a predictor of an algorithm's runtime on a given problem instance based on the features of the instance and the algorithm's past performance.

Automated parameter configuration is mainly executed offline or before the actual target algorithm run. This contrasts with and complements the volume of works which seek to adaptively adjust the parameter configuration dynamically during search [14, 15]. For example, [14] applied reinforcement learning (*RL*) to adapt the diversification in a fast online manner to the characteristics of a task and of the local configuration. In a adaptive scenario, the parameter values are modified to respond the search algorithm's behavior during its execution.

All of the above approaches is done automatically without human interference. In a separate front, there are approaches which require direct collaboration with human to guide the tuning process [48, 4]. These approaches explore the human ability to the recognize target algorithm pattern and behavior to design a better target algorithm. For example, [48] visualized the local search algorithm's fitness landscape search trajectory that allows algorithm designers to investigate the fitness landscape structure of the target algorithm.

2.6 Chapter Summary

In this chapter, background materials for automated parameter tuning are discussed. We introduce notations and formal definitions for the automated parameter tuning problem. We categorize works related to automated parameter tuning into four groups based on two aspects: the strategy to handle diverse instances and the existence of statistical models to explain the tuning process, and review recent approaches in each category.

The existing approaches have shown significant improvements in the performance of target algorithms. Each approach also has its respective limitations; one-size-fits-all approaches find only a single configuration, which may not be effective on large

and diverse instances, while instance-specific approaches are less generic due to its instance-specific features.

Chapter 3

Instance-Specific Automated Parameter Tuning via Trajectory Clustering (CluPaTra)

In the previous chapter, we have formally defined the automated parameter tuning problem, which aims to find the parameter configuration to best optimize a target algorithm. We use a performance metric based on the percentage deviation of quality from the global optimum or best known solution to measure the target algorithm. We also introduce some terms and notations used in this thesis.

Several existing works for automated parameter tuning have been introduced in the literature. Some approaches return a "one-size-fits-all" parameter configuration for all instances. This is unsatisfactory because different instances may require the target algorithm to use very different parameter configurations in order to find good solutions. On a separate front, there have been approaches that perform instance-based automated tuning, but they are usually problem-specific due to their reliance on problem-specific features.

In this chapter, we propose CluPaTra, a **generic instance-specific parameter tuning framework** which automatically finds good parameter configurations by an instances clustering approach based on a problem-independent feature, search trajec-

tory. Search trajectory is defined as the path that a local search algorithm follows as it searches from an initial solution to its neighbor from one iteration to the next. The advantage of CluPaTra is the fact that the search trajectory is computed from a local-search based algorithm. Hence our approach is problem-independent and may conceptually be applied to any local search-based algorithm.

In this chapter we describe CluPaTra in greater detail. First we present the framework overview, followed by its three major components: feature selection, similarity calculation, and clustering method. We then describe CluPaTra's four instantiations. We show the results of our experiments on three COPs and then discuss the result. We conclude with a chapter summary.

3.1 Framework Overview

Rather than ambitiously attempting instance-specific tuning which we believe to be a computationally prohibitive and unachievable task in the near future because of the large parameter configuration space and large number of instances, CluPaTra adopts a cluster-based treatment. The result is a fine-grained tuning framework that does not produce a one-size-fits-all parameter configuration, but instance (or rather cluster)-based parameter configurations. Even though strictly speaking, our method is cluster-specific rather than instance-specific, it represents a big leap from one-size-fits-all schemes.

CluPaTra is designed as a generic (problem-independent) approach, based on **CLU**stering instances with similar **PA**tterns according to their search **TRA**jectories. We represent a search trajectory as a directed sequence and apply a well-studied sequence alignment technique to cluster instances based on the similarity of their respective search trajectories. We then tune each cluster to find a good parameter configuration for the respective cluster.

CluPaTra is illustrated in Fig. 3.1. It is divided into two phases: training and testing. The training phase starts with a clustering process and is followed by a tuning

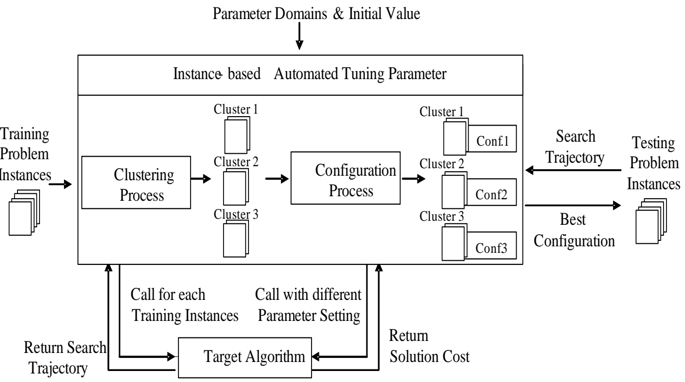


Figure 3.1: CluPaTra Framework

process. The clustering process is where we select and represent a generic feature, calculate similarities and perform clustering. Moving to the tuning process, we apply a tuning procedure to derive the best parameter configurations for each cluster.

In the testing phase, we match the search trajectory of the testing instance against the clusters using pair-wise sequence alignment to find the most similar cluster. We then return the parameter configuration found for that cluster (during the training phase) as the recommended parameter configuration for the respective testing instance. The steps involved in the training and testing phase are shown in Fig. 3.2 and Fig. 3.3 respectively.

In this thesis, we focus on the clustering process. For the tuning process, we use existing one-size-fits-all configurators such as CALIBRA [3], ParamILS [60] or F-Race [17]. The clustering process has three major components: (1) feature selection; (2) similarity calculation; and (3) clustering method. The component details are described as follows.

3.2 Feature Selection

Instance specific features that determine the intrinsic difficulty of an instance play an important role in the meta-heuristic algorithm's performance [78]. Consequently, there has been increasing interest in finding instances features that have impact on the difficulty, in terms of performance, of improving algorithm performance [6, 53, 54, 91, 93, 100, 102, 105, 106, 109, 115].

Procedure TrainingPhase
Inputs: A: Target Algorithm;
I: Training instances;
Θ: Parameter Configuration Space;
$\mathbf{x}_i$: Initial Sequence Configuration;
Outputs: C: A set of clusters;
$\mathbf{X}$: Parameter configurations for each cluster in C;
Method:
1: F = FeatureExtraction(A, I, $\mathbf{x}_i$);
2: S = SimilarityScore(I, F);
3: C = Cluster(I, S);
4: for all cluster in C do
5: $\mathbf{X}_i$ = configurator(A, C_i, Θ);
6: return C, $\mathbf{X}$;

Figure 3.2: CluPaTra Training Phase

Procedure TestingPhase
Inputs: A: Target Algorithm;
I_t: A set of Testing instance;
C: Set of clusters;
$\mathbf{X}$: Parameter configurations for each cluster in C;
$\mathbf{x}_i$: Initial Sequence Configuration;
Outputs: $BestConfig$: A recommended configuration;
Method:
1: for each instance in I_t do
2: $BestClust_i$ = Mapping(I_{ti}, A, x_i, C);
3: $BestConfig_i$ = $\mathbf{X}[BestClust_i]$;
4: return $BestConfig$;

Figure 3.3: CluPaTra Testing Phase

Various problem-specific features have been proposed for a wide range of Combinatorial Optimization Problems (COPs). Some notable features are flow dominance for Quadratic Assignment Problem (QAP) [53, 102, 105, 109] and population correlation structure and constraint slackness for the Knapsack Problem [54, 93]. The most straightforward features are those that are extracted from the problem or instance definition itself, such as number of variables and constraints, which can be derived to numerous candidate features using computational feature extraction processes [100]. Other non-straightforward features may require large scale experimental studies and are highly dependent on domain knowledge in a particular COP. Finding appropriate features takes tremendous human effort, and the features in most cases cannot be

Table 3.1: Run Time for Random-3-SAT instances

Instances	Percentage of Local Optima Found	Run Time
uf20-91/easy	0.11%	13.05
uf20-91/medium	0.13%	83.25
uf20-91/hard	0.16%	563.94
uf50-218/medium	47.29%	615.25
uf100-430/medium	43.89%	3,410.46
uf150-645/medium	41.95%	10,231.89

reused for another problem.

On a separate front, there have been approaches that attempted to find problem-independent features using correlation between objective function and search space (fitness landscape) [6, 55, 91, 106]. Problem-independent features can be used on different COPs, such as Traveling Salesman Problem (TSP) [91], Quadratic Assignment Problem (QAP) [6] and Knapsack Problem [106]. Examples of these features are: Fitness Distance Correlation (FDC) [91, 55] and ruggedness coefficient [6, 55]. In FDC, we test if there exists any correlation between delta fitness and distance from a solution to the nearest local optimum that is known priori. Unfortunately, to calculate the FDC, we need to find all the local optima. This means we need to explore the entire fitness landscape, which is time consuming and to some extent are impossible for certain instances [91]. To illustrate the amount of time needed to explore the fitness landscape, Table 3.1 shows the search cost of exhaustive renumeration of search space for Random-3-SAT instances [55]. Similarly, calculating the ruggedness coefficient also entails the exploration of the entire fitness landscape [6].

In attempting to utilize a problem-independent feature which can be more efficiently computed, we propose the use of the search trajectory, i.e. a solution path derived from one run of the target local-search algorithm. It is the proxy to fitness landscape that can be obtained with a small amount of additional computational time. In section 3.6, we demonstrate that the search trajectory indeed provides a good measure of the fitness landscape's similarity of instances.

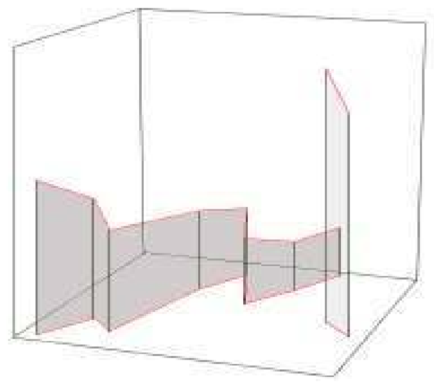


Figure 3.4: Example of Search Trajectory from the Traveling Salesman Problem (TSP) instance

3.2.1 Search Trajectory Definition

Search trajectory is defined as the path of solutions that a target algorithm $\mathcal{A}$ finds as it searches through the neighborhood search space. It can be derivied from any local search based algorithm without incurring much additional computation time. The search trajectory illustration for one Traveling Salesman Problem (TSP) instance is shown in Fig 3.4.

The xy plane represents the search space while z axis represent the objective value. Since it is not possible to provide a perfect 2-D layout for all the solutions such that the 2-D Euclidean layout distance preserves the Hamming distance for each pair of the solutions, we make use of the heuristics algorithm, namely the spring model [47], where it has been shown that the spring model can reduce the layout error by more than 83% (from 0.18 to 0.03) (see page 44 of [47]).

We propose search trajectory as a generic feature to define similarity between problem instances. The rationale of our feature is predicated on the relationship between fitness landscape and search trajectories [48], and the tight correlation between the fitness landscape and algorithm performance [92]. Whereas generating entire fitness landscape for each instance is time consuming and generally impractical, we propose to use search trajectory as a proxy for fitness landscape. Granted that different parameter configurations may produce (very) different search trajectories for a given instance, we claim that the *similarity* of search trajectories between instances is

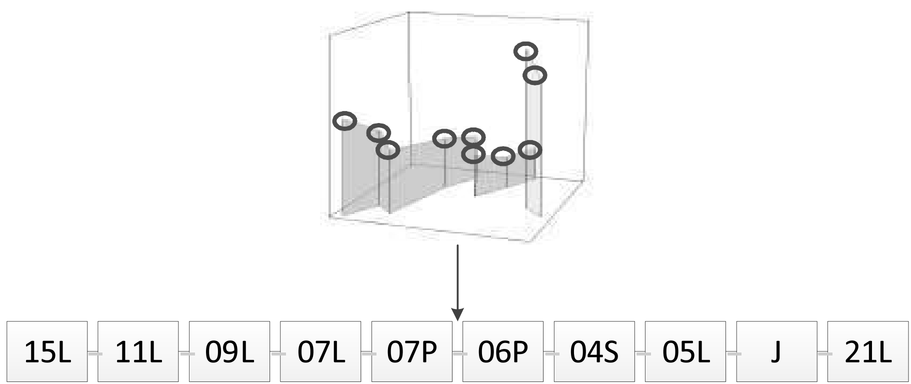


Figure 3.5: Example of Direct Sequence Representation of Search Trajectory for the Travelling Salesman Problem (TSP) instance

preserved across configurations.

Given a fixed local search algorithm $\mathcal{A}$, our bold conjecture is that instances with similar fitness landscapes have similar trajectory patterns under a fixed parameter setting; and that there exists a parameter setting that yields good solutions in instances with a similar fitness landscape. The latter claim has been observed in TSP and QAP instances [49].

Note that we are using search trajectory as a proxy for fitness landscape; granted however that the search trajectory will not adequately represent the entire fitness landscape. Our goal is to find similar behavioral patterns of the algorithm; not to measure the actual performance of the algorithm. To that extent, we claim that search trajectory (under a single suitably defined parameter configuration) is a sufficient proxy to measure similarity between instances.

3.2.2 Representation of Search Trajectory

Generally, we presented search trajectory as a directed sequence of symbols, each representing a solution along the search trajectory, as illustrated in Fig 3.5. Each symbol encodes a combination of two solution attributes: position type and its percentage deviation of quality from $Best$ (as defined in Definition 3).

Position type represents the local property of a solution with respect to its search neighborhood, and is defined based on the topology of the local neighborhood [55].

Table 3.2: Position Types Property of Search Trajectory

Position Type Label	Symbol	Objective Value larger	equal	smaller
SLMAX (strict local max)	A	No	No	Yes
LMAX (local max)	X	No	Yes	Yes
LEDGE	L	Yes	Yes	Yes
SLOPE	P	Yes	No	Yes
IPLat (interior plateau)	I	No	Yes	No
LMIN (local min)	M	Yes	Yes	No
SLMIN (strict local min)	S	Yes	No	No

'Yes' = present, 'No' = absent; referring to the presence of neighbors with larger, equal and smaller objective values

There are 7 position types determined by evaluating the solution objective value with its local direct neighbors' objective values - whether it is better, worse or equal. The 7 positions types are shown in Table 3.2. In the actual search trajectory, we only use either LMIN or LMAX (respectively SLMIN and SLMAX) depending on the target algorithm type (maximizing or minimizing).

The deviation of solution quality measures in a sense global property of a solution (since it is compared with $Best$). If the global optimum value is unknown, we use the best known value; granted the best known value is not the same as global optimal value. This provides a reasonably good upper bound (for a minimization problem); because our aim is to find similar patterns of the transition from one solution to the next; not to measure the actual absolute performance of the algorithm. The best known value suffices in providing a good proxy to the global optimal value for our purpose of representing the trajectory. We believe that the search trajectory using the best known value can be shifted (with a constant translation vector) to the real search trajectory using global optimum value.

Position type and percentage deviation of quality are combined into a symbol with the first two digits being the deviation of solution quality and the last digit being the position type. Note that the attributes are generic, which means they can be easily retrieved/computed from any local-search based algorithm albeit from different problems. Being mindful that some target algorithms may have cycles and (random)

restarts, we intentionally add two additional symbols: 'C' and 'J' in sequence representation. 'C' symbol is used when the target algorithm returns to a position that has been found previously. We do not record the cycle position and just use 'C' symbol to mark a cycle. The 'J' symbol is used when the local search is restarted.

An example of the sequence representing the search trajectory in Fig. 3.5 is *15L-11L-09L-07L-07P-06P-04S-05L-J-21L-19L*. Notice that after position 8, the target algorithm performs a random restart, hence we add a 'J' symbol after position 8.

In addition to the above representation (which we refer to as Exact Sequence), we also represent the search trajectory as a transition sequence. The transition sequence is made up of symbols that represent a transition (or movement) between two neighboring solutions in the search trajectory. The focus is not on solution position, but rather the movement along the search trajectory in order to detect trajectories that move in parallel but are not necessarily identical (their corresponding positions differ by a constant value). We use the transition sequence to capture similarity across different size instances. In the transition sequence, each symbol contains three parts:

1. the absolute difference in deviation between the first and second solutions
2. the position type of the first solution
3. the position type of the second solution

Similar to an exact sequence, a transition sequence may also have two additional symbols: 'C' and 'J'. These attributes are also generic and can be easily derivied from any exact sequence. An illustration of the transition sequence representing the search trajectory of the Traveling Salesman Problem (TSP) instance in Fig. 3.5 is shown in Fig. 3.6.

3.3 Similarity Calculation

Having represented trajectories as linear sequences, it is natural to apply pairwise sequence alignment to obtain the similarity score between a pair of trajectories. In

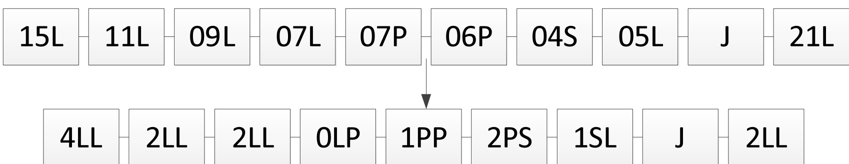


Figure 3.6: Example of Transition Sequence for search trajectory of Traveling Salesman Problem (TSP) instance

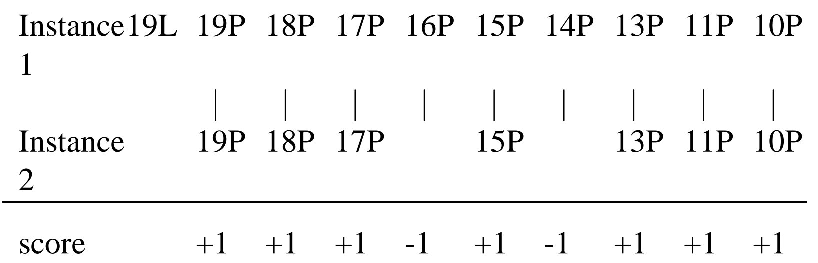
pairwise sequence alignment [51], the symbols of one sequence are matched with those of the other sequence while respecting the sequential order in two sequences. This also allows small gaps to occur if symbols do not match. Two or more search trajectories are similar if some fragments (several number of consecutive moves) of the path have identical solution attributes. The longer the fragments the more similar it is. In the following, we introduce our two techniques (basic and robust sequence alignment) for the search trajectory similarity calculation based on pairwise sequence alignment.

3.3.1 Basic Sequence Alignment

In basic sequence alignment, two symbols are matched if they have identical solution attributes. A standard sequence alignment method is applied to maximize the number of matched symbols between two sequences sequentially. A pair of matched symbols gives a positive score (+1), while a gap gives a negative score (-1). The similarity score is calculated as the sum of the scores of matched symbols (+1) and gaps in the alignment (-1). Noted that the sequence alignment is done after we have the whole search trajectory, thus there is no insertion, deletion or cost modification in the score calculation process. An example of sequence alignment for two search trajectories of Traveling Salesman Problem (TSP) instances is illustrated in Table 3.3.

There are two types of alignment strategies: local and global. In local alignment, sequences are aligned partially, whereas global alignment aligns the entire length of the sequences. Because search trajectory sequences have varying lengths, we find local alignment fits our needs. One well-known algorithm that performs such sequence alignment is the *Smith-Waterman algorithm* [51] that works by comparing all possible

Table 3.3: Example of Sequence Alignment from a pair of instances



Instance 1	19L	19P	18P	17P	16P	15P	14P	13P	11P	10P
		|	|	|	|	|	|	|	|	|
Instance 2		19P	18P	17P		15P		13P	11P	10P
score		+1	+1	+1	-1	+1	-1	+1	+1	+1

alignments regardless of their lengths, start and end positions. It then chooses the best alignment as the alignment that maximizes the similarity score, which is calculated as sum of the scores of matched symbols and gaps in the alignment. Note that the best alignment may start and end anywhere in the two sequences, so long as it produces the best similarity score. We adapt the Smith-Waterman algorithm and use the best similarity score for each pair of sequences. The final similarity score is normalized by dividing it with $\frac{1}{2} \times (|Sequence_1| + |Sequence_2|)$.

The sequence alignment algorithm is implemented using dynamic programming with time complexity $O(n^2)$ where n is the maximum sequence length. To cluster instances (see the subsection below), we need to compute similarity scores for all possible pairs of training instances. Hence, the total time complexity for sequence alignment is $O(m^2 \times n^2)$, where n is the maximum sequence length of the sequences and m is the number of instances in the training set.

3.3.2 Robust Sequence Alignment

In robust sequence alignment, we relax the matching criteria. Whereas in basic sequence alignment, two symbols are a match if and only if the two symbols are exactly identical, in robust sequence alignment, we consider partial matching where the symbols are identical but the deviation attribute is different in a certain threshold. This relaxed similarity calculation allows us to more robustly capture search trajectory similarity. Under robust sequence alignment, a match occurs if one of the following conditions is satisfied:

Table 3.4: Threshold Value for Robust Sequence Alignment

Threshold	Similarity Score
1	0.97
2	0.93
3	0.92
4	0.87
5	0.56

1. The two symbols are identical

2. The *position type* of the symbols is the same and the absolute difference in the *deviation* attribute of the two symbols is less than a certain threshold.

Robust sequence alignment requires us to make sure that the matched symbols are still very similar. Hence, the threshold should not be too far apart. We run a series of preliminary experiments to determine the threshold value. We calculate the average similarity between a pair of similar sequences using different thresholds as shown in Table. 3.4. We then set the threshold value to threshold values with the highest similarity score in our experiment.

We apply the same sequence alignment algorithm and score normalization techniques as in basic sequence alignment.

3.4 Clustering Method

Our goal in clustering is to group similar instances according to their search trajectory similarity. A typical clustering algorithm requires a distance measure between data points. For distance measure we use $\frac{1}{similarity\ score}$. After such measurement is made, a standard clustering algorithm could be deployed.

In instance-specific tuning process, we need a good and fast clustering method which can easily and automatically determine the optimum number of clusters without additional computation time. For this purpose, we compare two well-known clustering method: AGNES (AGglomerative NESting) [65] and k-medoids [51].

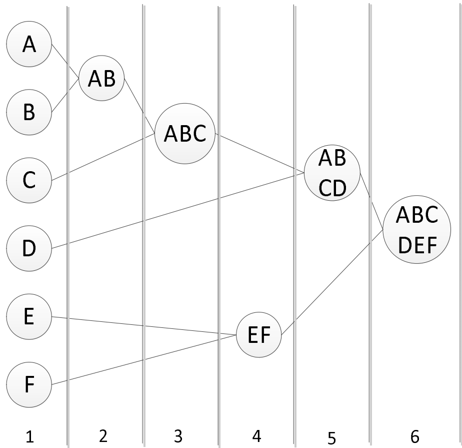


Figure 3.7: Hierarchical Clustering Method: AGNES (AGglomerative NESting)

AGNES works by placing each instance initially in a cluster of its own and iteratively merges two closest clusters (i.e., a pair of clusters with the smallest distance) resulting in a lesser number of clusters of larger sizes. The process is repeated until all nodes belong to the same cluster unless a termination condition applies. Examples of termination conditions are when a minimal number of cluster is reached or when the maximal inter-cluster distance goes below a certain value. AGNES can be computed in a linear computation time. AGNES is illustrated in Fig. 3.7.

To automatically determine the minimal number of clusters to be used, we apply the L method [95] that works using an evaluation graph where the *x*-axis is the number of clusters and the *y*-axis is the evaluation function value at *x* clusters. For the evaluation function value, we use average distance among all instances in two different clusters. The L method determines the number of clusters by fitting the evaluation graph into two lines that most closely fit the curve, as illustrated in Fig. 3.8. The method chooses the intersection point between those two lines as the optimum number of clusters. The intersect point is the point of maximum curvature of this graph which has minimum average distance (calculated using root mean square error) for both the left and right side of the intersect point. It is calculated using the following formula:

$$c^* = min\left[\frac{RMSE(L)}{n_L} + \frac{RMSE(R)}{n_R}\right] \tag{3.1}$$

where:

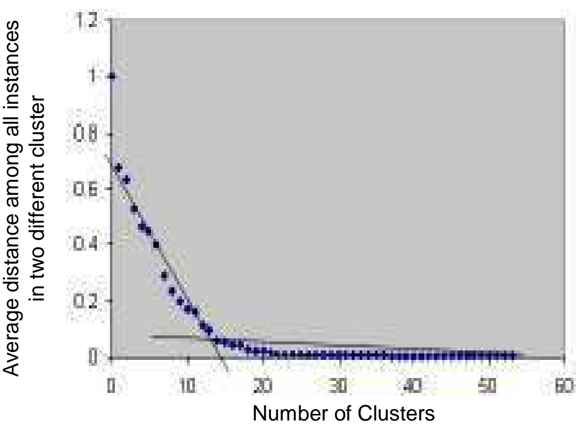


Figure 3.8: Evaluation Graph for L-Method to Determine Number of Cluster

Notation	Definition
$RMSE(L)$	root mean squared error of points in the left side of c
n_L	number of points in the left side of c
$RMSE(R)$	root mean squared error of points in the right side of c
n_R	number of points in the right side of c

To determine the optimum number of clusters, the L-method only requires AGNES algorithm to be run once because all the clusters generated by AGNES can be recorded in one run.

On the other hand, k-medoids is a partition-based clustering method that repeatedly breaks the data set up into k groups as an attempt to improve the clusters' evaluation function [51], which in this paper, is the average distance among all instances in two different cluster. It is a variant of the k-means method but it selects real data points as centers (medoids or exemplars) instead of imaginary points. The complexity of k-medoids is $O(k(n-k)^2)$ with k being the number of clusters and n being the number of instances.

In k-medoids, we may automatically determine the number of optimum clusters using statistical comparison methods on the cluster quality as in g-means, a variant of k-medoids, that is used in ISAC [64], an existing instance-specific parameter tuning. But the calculation may need some additional computation time.

Because AGNES with L-Method is easier to implement and has linear time com-

Table 3.5: Four instantiations of CluPaTra

Instantiation	Search Trajectory Representation	Similarity Calculation
Standard	Exact sequence	Basic Seq. Align.
Trans	Transition sequence	Basic Seq. Align.
Robust	Exact sequence	Robust Seq. Align.
Trans-Robust	Transition sequence	Robust Seq. Align.

plexity, we use AGNES with L-method as the clustering method. We provide a detailed comparison between AGNES and k-medoids in the Empirical Experiment Result section.

3.5 CluPaTra Instantiations

As described above, CluPaTra has two search trajectory representations, exact and transition sequence, and two similarity calculation techniques, basic and robust sequence alignment. We combine these techniques and construct four instantiations of CluPaTra. The terminology used is given in Table 3.5.

3.6 Empirical Experiment Result

We conduct a series of experiments to investigate CluPaTra performance. We start by describing our experiment measurement, target problems and algorithms and the experiment setting and setup. We then show the empirical result for: verification of similarity preservation, clustering analyses, computational time, performance comparison and different clustering method comparison.

3.6.1 Experiment Measurement

In this experiment, our objective is to investigate the CluPaTra performance on cluster quality as well as solution performance. For cluster quality, we use training and testing cluster quality in Definition. 7 and 8 respectively. For the solution performance, we

use the performance metric in Definition. 3.

3.6.2 Target Problems and Algorithms

To demonstrate the generic nature of our approaches, we experiment using three classical Combinatorial Optimization Problems (COPs): Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP). The details of these problems and their target algorithms are as follows.

Traveling Salesman Problem (TSP)

Given a list of cities and the distances between each pair of cities, the objective of Traveling Salesman Problem (TSP) is to find the shortest possible route that visits each city exactly once and returns to the origin city [74]. TSP is one of the NP-Complete problem [40]. It can be formally defined as follows.

Definition 9 (Traveling Salesman Problem [TSP]) *Given a complete weighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, with v being the set of cities and e being the weighted distance between two cities, the TSP objective is to find a closed tour s that visits each of the cities exactly once and minimizes the objective function* $\sum_{i=1}^{n-1} d_{s_i s_{i+1}} + d_{s_n s_1}$.

In our experiment, we use a well-known Iterated Local Search (ILS) algorithm as implemented in [49] as the target algorithm. We modify the code and extract 4 discrete parameters to be tuned as shown in Table 3.6. For all experiments, we fix the maximum number of iterations to 1000.

We apply our target algorithm to 70 benchmark instances extracted from TSPLib (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/). For best known values, we use the optimum/best values from TSPLib. Fifty six random instances are used as training instances and the remaining 14 instances as testing instances. The problem size (number of cities) varies from 51 to 3038.

Table 3.6: Parameters for ILS Algorithm for Traveling Salesman Problem (TSP)

Parameter	Description	Range
Pert	number of perturbations being done	[1,10]
n_improve	max non-improving moves	[1,10]
choice	perturbation strategy where: 3=3-opt change and 4=double-bridge move	[3,4]
acp	acceptance criteria strategy where: 0=accept only improving moves and 1=accept all moves	[0,1]

Quadratic Assignment Problem (QAP)

Quadratic Assignment Problem, or QAP in short, aims to assign n number of facilities to n number of locations with the goal of minimizing the sum of the distances and flows from every locations [73]. QAP is also an NP-Complete problem [40]. It can be formally defined as follows.

Definition 10 (Quadratic Assignment Problem [QAP]) *Given a $n \times n$ matrix of flow information between facilities A and $n \times n$ matrix of distance between locations B, the QAP objective is to find a permutation s=$\{1, 2, 3, ..n\}$ that minimizes the objective function $\sum_{i=1}^{n} \sum_{j=1}^{n} a_{s_i s_j} b_{ij}$.*

The target algorithm to solve QAP is hybrid Simulated Annealing and Tabu Search (SA-TS) algorithm (presented in [87]). It uses the Greedy Randomized Adaptive Search Procedure (GRASP) to obtain an initial solution, and then use a combined Simulated Annealing (SA) and Tabu Search (TS) algorithm to improve the solution. There are four parameters, real and integer values, to be tuned as described in Table 3.7. For all instances, we set the maximum number of iterations to 500.

We use two set of instances: (1) Set A: benchmark instances and (2) Set B: generated instances. In Set A, we use 50 benchmark instances from QAPLib (http://www.seas.upenn.edu/qaplib/), and randomly choose 40 instances for training and 10 for testing. The problem size (number of facilities) in Set A varies from 20 to 150. We use the optimum/best values from QAPLib for best known values. In Set B, we use two generators in [69] for single-objective QAP as in [88]. The first generator generates uniformly random instances where all flows and distances are integers

Table 3.7: Parameters for SA-TS Algorithm for Quadratic Assignment Problem (QAP)

Parameter	Description	Range
Temp	Initial temperature of SA	[100,5000]
Alpha	Cooling factor	[0.1,0.9]
Length	Length of tabu list	[1,10]
Pct	Percentage of non-improving iterations	[0.01,0.1]

sampled from uniform distributions. The second generator generates flow entries that are non-uniform random values, having the real-like structure and resemblance to the structure of QAP problems found in practical applications. We generate 500 instances with size from 10 to 150 from each generator and randomly choose 100 as training instances and 400 as testing instances.

Set Covering Problem (SCP)

Set Covering Problem (SCP) is an NP-Complete problem [40] that aims to find smallest number of sets from finite set X whose union still contains all elements in the family set of F [32]. It can be formally defined as follows.

Definition 11 (Set Covering Problem [SCP]) *Given a finite sets* S $=\{1, ..., n\}$ *of items, a family* F=$\{S_1, ..., S_m \subseteq S\}$ *of subsets of* S*, and a cost function* c=$F \rightarrow \mathbb{R}^+$*, the SCP objective is to find a subset* $C \subseteq F$ *such that* $S \subseteq \cup_{S_i \in C} S_i$ *and* $\sum_{S_i \in C} c(S_i)$ *is minimized.*

We use the tabu-search algorithm in [85] as the target algorithm with four parameters to be tuned as described in Table. 3.8.

We use two different instances set: (1) Set A: benchmark instances and (2) Set B: generated instances. For Set A, we use 50 benchmark instances from OR library (http://people.brunel.ac.uk/ mastjjb/jeb/orlib/scpinfo.html) and randomly pick 40 instances for training and 10 for testing. For Set B, we use 80 generated instances as used in [64], with 40 as training instances and 40 as testing instances.

Table 3.8: Parameters for TS Algorithm for Set Covering Problem (SCP)

Parameter	Description	Range
fTSLength	Tabu Length Factor	[1000, 10000]
iNonImprove	Non Improvement Moves	[5, 200]
iProbRandom	Probability of Random Moves	[1, 20]
iDeterministic	Stochastic Factor	[0, 1]

3.6.3 Experiment Setting and Setup

One-size-fits-all Configurator

In order to derive meaningful experimental comparison, we deliberately chose to use ParamILS [60] as our one-size-fits-all configurator. ParamILS is itself an iterated local search algorithm used for tuning discrete parameters. Since ParamILS works only with discrete parameters, we first discretize the values of the parameters if the target algorithm has parameters that assume continuous values. We discretize the continuous parameters to 20 possible values by simple enumeration from minimum to maximum value.

Validity and Statistical Significant Measurement

To ensure unbiased evaluation, we use a 5-fold cross-validation [51]. To do 5-fold cross validation, we randomly divide the instances into 5 random groups and use 4 groups as training instances and 1 group as testing instances. We repeat the process 5 times and take the average. We perform a statistical test to compare the significance of our result. We use a t-test [83]; and we consider p-values below 0.05 to be statistically significant (confidence level 5%).

Comparison Method

We compare our experiment results with the ISAC method, a similar clustering-approach that uses problem specific features [64]. Whereas ISAC requires problem-specific features, we select the standard deviation of the city distances, the variance of the normalized nearest neighbour distances and the coefficient of variation of the normalized nearest neighbour distances for TSP [99] and flow dominance and sparsity

of flow matrix for QAP [102]. We do not generate the clusters in SCP for ISAC but instead we use the clusters used previously by ISAC in [64].

Because our aim is to measure solution quality, we do not compare our approach with Hydra [114], another instance-specific configurator that seeks to optimize run time performance but not solution quality of the target algorithm.

Experimental Setup

All experiments are performed on a 1.7 GHz Pentium-4 machine running Windows XP. We measure runtime as the CPU time needed by this machine. As an input to the configurator, we set a cutoff time at 10-100 seconds per run for the target algorithms. For each cluster from our approaches, we allow each configuration process to execute the target algorithm for a maximum of two CPU hours and to call the target algorithm for a maximum of 25 x n times, where n is the number of instances in the cluster. To ensure fair comparison, we set the time budget for ISAC and ParamILS to be equal to the average total time needed to run a full process of CluPaTra. This time budget is the stopping condition for ISAC and ParamILS.

3.6.4 Verification of Similarity Preservation

Prior to presenting CluPaTra's performance, we provide a scientific argument for CluPaTra. In the following, we justify our claim, that the *similarity* of search trajectories between instances is preserved across configurations, by providing a series of experimental observations. For this purpose, we experiment on a small set of TSP and QAP instances (Set A).

First, we provide a visual intuition for similarity preservation across different parameter configurations. Fig. 3.9 shows the trajectories obtained by 10 consecutive moves of an Iterated Local Search (ILS) algorithm for three TSP benchmark instances, namely *a280*, *d198* and *berlin52* using two random parameter configurations, namely configuration I and configuration II.

The xy plane represents the search space while z axis represent the objective value.

To layout the moves into a 2-dimensional xy plane, we calculate the distance between two solutions (e.g., number of different cities in TSP) and apply *"the spring model"* [49]. *"The spring model"* provides a heuristic for good layout if and only if the Euclidean distance between 2 solutions in the xy plane is roughly proportional to their Hamming distance. In this example, we observe that for both configurations, *a280* and *d198*, exhibit very similar topology ((a) and (b), (d) and (e)), while *berlin52* has a different topology compared to the similarity of *a280* and *d198*.

Next, we provide a statistical verification of the notion of similarity preservation for the trajectories produced by the TSP and QAP target algorithms used in our experiments. For this purpose, we verify on random pairs of instances across different parameter configurations. First, we randomly select 2 source instances (namely, benchmark instances *a280, berlin52* for TSP and *chr20a, sko100b* for QAP); Next we select randomly 10 other destination TSP (resp. QAP) instances. We randomly generate 5 parameter configurations for each target algorithm, and record the search trajectory for each instance. To simplify the experiment, we take the first 300 solutions obtained from the target algorithm as the search trajectory samples and calculate its similarity scores.

For each source-destination pair in each configuration, we compute their similarity score (based on the Standard instantiation of CluPaTra). The results are presented in Fig. 3.10. Observe that most pairs of instances maintain their similarity across different parameter configurations as shown by the small scatter of similarity values in each column (with the exception of several instances in the a280 instance). The deviation, mean and coefficient of variance (CV) of similarity values for the different parameter configurations are given in Table 3.10. For most pairs, the CV value is low (especially for QAP pairs), which means that the similarity score across different parameter configurations do not differ substantially from one another.

Finally, we present examples of clusters based on three different parameter configurations for TSP and QAP generated using the Standard instantiation of CluPaTra. We use 10 instances for both TSP and QAP. The clusters are shown in Table 3.9. Most

Table 3.9: Examples of Clusters from Different Parameter Configurations

Parameter Config.	Cluster #	Instances
TSP		
1	1	a280, fl3795, d1655, ts225
1	2	berlin52, kroa150, krob100, prl152
1	3	lin105, ch150
2	1	a280, fl3795, ts225
2	2	berlin52, kroa150, krob100, prl152
2	3	lin105, ch150, d1655
3	1	a280, fl3795, d1655, ts225
3	2	berlin52, kroa150, krob100, prl152
3	3	lin105, ch150
QAP		
1	1	chr20a, chr22a, chr22b
1	2	sko100b, sko100e, sko90
1	3	nug28, nug30, tai30a, wil100
2	1	chr20a, chr22a, chr22b
2	2	sko100b, sko100e, sko90
2	3	nug28, nug30, tai30a, wil100
3	1	chr20a, chr22a, chr22b
3	2	sko100b, sko100e, sko90
3	3	nug28, nug30, tai30a, wil100

of the instances (except one instance of TSP, d1655) are clustered in the same groups regardless of the parameter configuration used.

Based on the above observations, we argue that even though a given instance may have different search trajectories under different configurations, the *similarity* between two instances is preserved across configurations. This similarity preservation property allows us to perform clustering of instances using an arbitrary parameter configuration.

3.6.5 Clustering Analyses

To investigate the quality of clusters generated from CluPaTra, we conduct series of experiments for TSP, QAP and SCP using its benchmark instances (set A for QAP and SCP) and compare the result with ISAC.

We compare an example of clusters generated by one of our approaches: CluPa-

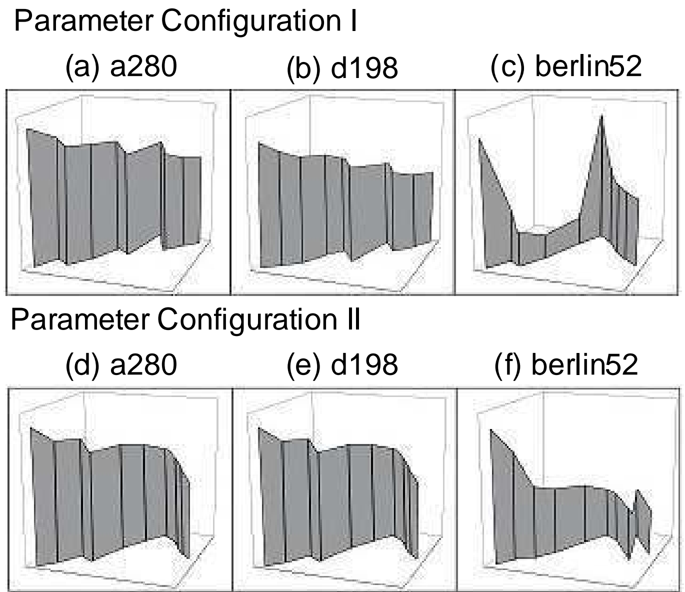


Figure 3.9: Search Trajectories of three TSP instances using two random parameter configuration

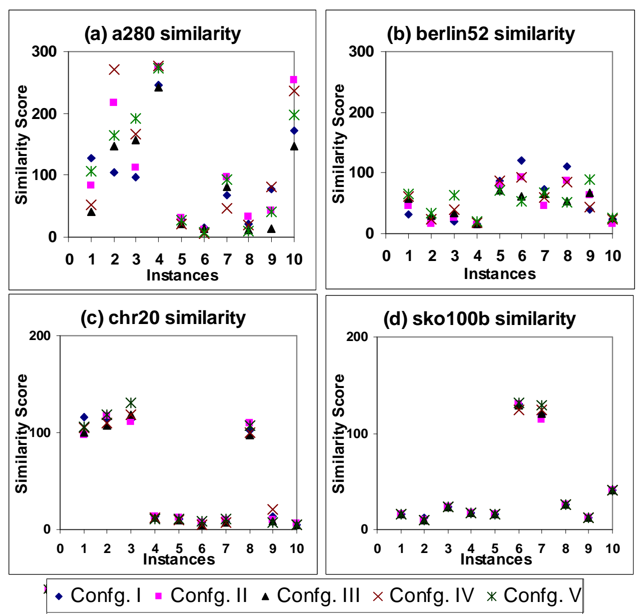


Figure 3.10: Search Trajectory Similarity Score between two TSP and QAP instances and 10 other random instances using 5 Different Random Parameter Configurations

Table 3.10: Similarity Score of Randomly Selected Instance Pairs for Instances' Similarity Preservation

Instances	σ	μ	c_v	σ	μ	c_v
I. TSP		a280			berlin52	
ch150	32.70	**82.20**	0.40	12.42	52.20	0.24
d1655	57.47	**181.20**	0.32	6.02	25.60	0.00
d657	35.31	**144.60**	0.24	15.54	36.20	0.43
fl3795	14.81	**262.00**	0.06	2.24	16.40	0.14
kroa150	4.12	25.80	0.16	6.73	**78.80**	0.09
krob100	3.58	11.00	0.33	24.33	**84.20**	0.29
lin105	18.18	**77.20**	0.24	9.35	62.40	0.15
pr152	7.78	18.80	0.41	22.38	**77.40**	0.29
rd100	25.32	50.80	0.50	17.85	**60.40**	0.30
ts225	39.55	**201.60**	0.20	3.88	22.40	0.17
II. QAP		chr20a			sko100b	
chr22a	6.49	**104.80**	0.06	0.00	16.00	0.00
chr22b	4.13	**113.40**	0.04	1.20	10.60	0.11
lipa50b	6.83	**118.40**	0.06	0.00	24.00	0.00
nug28	0.75	12.20	0.06	0.00	**18.00**	0.00
nug30	0.75	10.80	0.07	0.00	**16.00**	0.00
sko100e	1.60	6.80	0.24	2.87	**129.40**	0.02
sko90	1.60	8.80	0.18	5.04	**121.20**	0.04
ste36a	4.71	**103.20**	0.05	0.00	26.00	0.00
tai30a	4.71	12.20	0.39	0.00	**13.00**	0.00
wil100	0.40	5.20	0.08	0.00	**41.00**	0.00

σ=standard deviation; μ=mean; c_v=coefficient of variation; Boldface indicates the best similarity score mean.

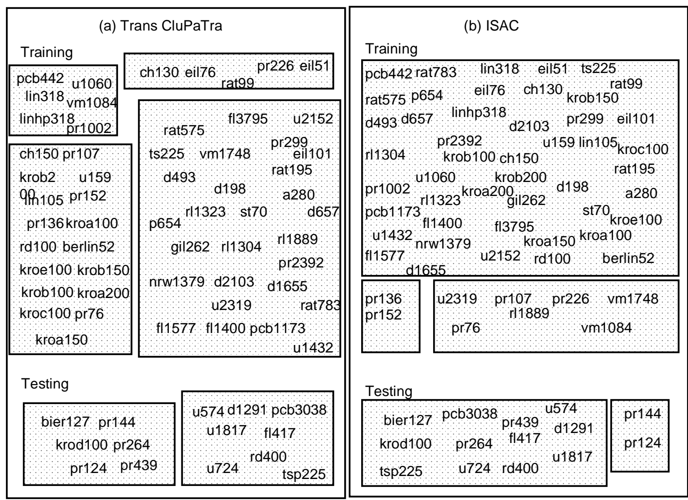


Figure 3.11: TSP Cluster Result Comparison

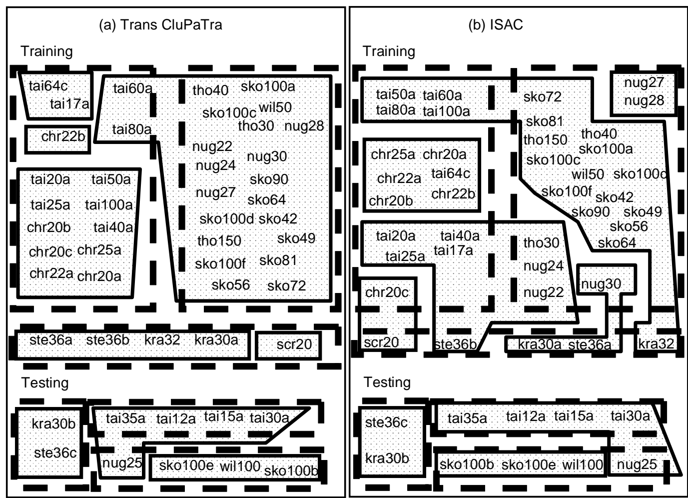


Figure 3.12: QAP Cluster Result Comparison

Tra (Trans Instantiation) and ISAC using one set of training and testing benchmark instances as reported in Fig. 3.11 for TSP and Fig. 3.12 for QAP.

For TSP, we observe that the CluPaTra (Trans Instantiation) method is able to capture the similarity of instances with differing sizes, which may have different search trajectory symbols but have similar transitions along the search trajectories. Because of the non-existence of *ground-truth* classification for TSP benchmark instances, we cannot compute the cluster qualities ($\mathcal{Q}_{train}$ and $\mathcal{Q}_{test}$) directly; instead it is inferred from the performance of the target algorithm which is described in the later subsection.

For QAP, we use the existing well-studied classification based on distance and flow metrics [105] as the *ground-truth* classification. It divides the instances into 5 groups:

Table 3.11: CluPaTra's Cluster Quality Comparison for Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP)

	QAP		SCP	
Technique	Training	Testing	Training	Testing
CluPaTra Standard	0.68	0.70	0.75	0.60
CluPaTra Trans	**0.85**	**0.90**	**0.82**	**0.65**
CluPaTra Robust	0.78	0.70	0.81	0.60
CluPaTra Trans-Robust	0.7	0.80	0.81	0.62
ISAC	0.80	0.80	-	-

Boldface indicates the best cluster quality.

(1) random and uniform distances and flows, (2) random flows on grids, (3) real-life problems, (4) characteristics of real-life problems and (5) non-uniform, random problems. Due to the limitation of the target algorithm (which is unable to solve group (4) and (5) problems), we only use instances from groups (1), (2) and (3). The clusters from CluPaTra and ISAC are shown in solid boxes while the *ground-truth* classification (for QAP only) are shown in dashed boxes. Notice that the clustering by CluPaTra (Trans Instantiation) is almost the same as the *ground-truth* classification.

We then compare the clusters generated for QAP and SCP by CluPaTra and ISAC and show the result in Table. 3.11. We do not compare our result on TSP because we do not have the *ground-truth* for those problems. We use the same ground-truth classification as above for QAP. For SCP, it has been shown in [38] that benchmark instances from OR library and [11] have very different FDC (Fitness Distance Correlation) scores. We consider those two sets of benchmark instances as the ground truth clusters. For SCP, we do not generate clusters for ISAC because we do not have features for SCP.

Our approaches construct better clusters compared to ISAC with respect to cluster quality metric ($\mathcal{Q}_{train}$ and $\mathcal{Q}_{test}$) as shown in Table 3.11. We observe that the cluster quality score for CluPaTra Trans is the highest compared to other approaches.

Table 3.12: CluPaTra's Computational Time

Technique	TSP Training	TSP Testing	QAP Training	QAP Testing	SCP Training	SCP Testing
CluPaTra Standard	5.58 s	**0.04** s	1,051 s	2,718 s	163 s	53 s
CluPaTra Trans	**5.46** s	0.05 s	**1,002** s	**2,547** s	**160** s	**48** s
CluPaTra Robust	6.02 s	0.07 s	1,984 s	3,157 s	198 s	80 s
CluPaTra Trans-Robust	6.05 s	0.08 s	2,012 s	3,254 s	205 s	89 s

Boldface indicates the fastest computation time.

3.6.6 Computational Time

The time needed (in seconds) for CluPaTra to form clusters in the clustering process is shown in Table. 3.12. For QAP and SCP, we used generated instances (set B) while for TSP we used benchmark instances.

The most time-consuming procedure in the training phase is calculating the similarity of trajectories. Evidently, different similarity calculation techniques require different computational budget for calculating the similarity. The most time-consuming procedure in the training phase is calculating the similarity of trajectories. Evidently, different similarity calculation techniques require different computational budget for calculating the similarity. In CluPaTra, the Robust sequence alignment technique takes almost four times longer than the basic sequence alignment. This happens because it requires more computation time to find partial-match symbols.

3.6.7 Performance Comparison

To evaluate the effectiveness of our approaches, we conduct experiments for TSP, QAP and SCP and compare its result against the result from vanilla one-size-fits-all configurator (ParamILS) and ISAC. For QAP and SCP, we use generated instances (set B) while for TSP we used benchmark instances. Table 3.13 shows the average performance result from 5-fold-cross-validation for TSP, QAP and SCP. Notice that CluPaTra Trans outperforms other methods in both training and testing instances.

We verify the effectiveness of our approaches in providing the best configuration

Table 3.13: CluPaTra's Performance Comparison of Three Classical COPs

Problem	Technique	Training	Testing
TSP	ParamILS	2.67	2.02
	CluPaTra Standard	2.22*	1.93*
	CluPaTra Trans	**2.01***	**1.72***
	CluPaTra Robust	2.10*	1.81*
	CluPaTra Trans-Robust	2.06*	1.93*
	ISAC	2.02	1.88
QAP	ParamILS	2.21	2.27
	ground-truth	1.93*	2.09*
	CluPaTra Standard	1.99*	2.19*
	CluPaTra Trans	**1.88***	**2.08***
	CluPaTra Robust	1.89*	2.10*
	CluPaTra Trans-Robust	1.90*	2.19*
	ISAC	1.98	2.15
SCP	ParamILS	1.53	0.82
	CluPaTra Standard	1.24*	0.81*
	CluPaTra Trans	0.78*	0.80*
	CluPaTra Robust	1.01*	0.98*
	CluPaTra Trans-Robust	**0.67***	0.78*
	ISAC	1.13*	**0.77***

* = statistically significant against ParamILS
Boldface indicates the fastest computation time.

for each testing instance by experiments using QAP benchmark instances (Set A) and generated the clusters using CluPaTra Trans. We run the target algorithm for all QAP testing instances in Fig. 3.12 using parameter configurations from each cluster and show the result in Table 3.14. From the table we observe that each testing instance, except for tai35a, has the best performance using parameter configurations from the most similar cluster.

To further investigate the effect of clustering in the overall performance result, we calculate the Pearson product-moment correlation coefficient for the testing instances in Table 3.14. We also calculate the Pearson correlation coefficient (and the p-value) for cluster number and the overall performance result for each testing instance and report the results in Table 3.15. From the table we observe that for each testing instance, except for nug25, there is a strong correlation between the cluster number and overall performance result. This may indicate that the clustering influence the overall performance result. Although there are other factors affecting the overall performance

Table 3.14: CluPaTra's Testing Instances Performance using Different Cluster's Parameter Configuration

Instance	Cluster	Parameter Configuration for each Cluster C#1	C#2	C#3	C#4	C#5	C#6
nug25	1	**0.48**	0.64	0.69	0.58	0.58	0.58
tai12a	1	**0**	**0**	**0**	**0**	**0**	2.80
tai15a	1	**0.19**	0.76	0.52	1.22	1.72	2.66
tai30a	1	**1.86**	2.81	2.20	2.57	3.03	2.65
tai35a	1	1.49	**1.38**	3.37	3.75	3.047	3.95
kra30b	2	**0.07**	**0.07**	0.97	**0.07**	1.88	1.18
ste36c	2	1.91	**1.71**	5.08	8.95	7.84	7.82
sko100b	3	0.69	1.22	**0.53**	1.16	1.31	1.29
sko100e	3	1.18	1.18	**1.10**	1.30	1.34	1.21
wil100	3	0.65	0.69	**0.63**	0.81	0.96	0.93

Parameter Configuration for:
C#1: Temp=4000, Alpha=0.9, Length=7,Pct=0.08
C#2: Temp=2000, Alpha=0.5, Length=7,Pct=0.09
C#3: Temp=3000, Alpha=0.3, Length=10,Pct=0.1
C#4: Temp=4000, Alpha=0.3, Length=10,Pct=0.07
C#5: Temp=100, Alpha=0.3, Length=10,Pct=0.03
C#6: Temp=5000, Alpha=0.1, Length=1,Pct=0.08

Boldface indicates the best performance result.

Table 3.15: Correlation between Cluster Quality and Overall Performance

Instance	Pearson's Coefficient	p-value
nug25	0.210	0.681
tai12a	0.654	0.133
tai15a	0.943***	0.001
tai30a	0.626	0.158
tai35a	0.838***	0.021
kra30b	0.709**	0.090
ste36c	0.874***	0.011
sko100b	0.620	0.164
sko100e	0.505	0.285
wil100	0.888***	0.008

***: p <0.05; **: p <0.1

Table 3.16: CluPaTra's Performance Comparison using Different Clustering Methods

	TSP		QAP		SCP	
Technique	Training	Testing	Training	Testing	Training	Testing
AGNES	2.01	**1.72**	**1.88**	**2.08**	**0.78**	**0.80**
k-medoids	**1.88**	1.90	2.08	2.16	0.99	**0.80**

Boldface indicates the best performance result.

result, such as the robustness of the global tuning and the stochastics of the target algorithm, we postulate that cluster quality significantly affects the overall performance result.

3.6.8 Comparison of Different Clustering Methods

To investigate the effect of different clustering methods in CluPaTra, we conduct an experiment using another well-known clustering method, k-medoids. We compare the performance result of using AGNES and k-medoids clustering methods on the Trans instantiation for TSP, QAP and SCP. For QAP and SCP, we use generated instances (set B) while for TSP we use benchmark instances. We set the k value to be equal to the AGNES cluster number. Table 3.16 shows that AGNES performs slightly better than k-medoids even though it is not statistically significant.

3.7 Discussion

The experimental results show that the instance-specific automated parameter tuning framework yields a significant improvement in performance compared with the pure one-size-fits-all configurator ParamILS. We also observe that all CluPaTra instantiations perform significantly superior to or equal to ISAC. Having more similar instances in smaller clusters will eventually guide the tuning process to find better parameter configurations for each cluster. Based on this result, we verify that dividing the instances into clusters using CluPaTra before running one-size-fits-all configurator provides a better parameter configuration for each instance and significantly improves the performance.

To represent the search trajectory, we need the best known/optimum solution value (OPT) for each instance. We use either (a) the known global optimal value, or (b) when the global optimal value is unknown, the best known value. For all TSP and several QAP benchmark instances, we use the known global optimal value from TSPLib and QAPLib respectively, while for other QAP benchmark instances, we use the best known value from QAPLib. For QAP and SCP generated instances, we use the best found values as the best known values. For generated instances, we use the best found solution. From the experiment result, we observe that our approaches using either known global optimal value or best known value are able to generate good clusters and hence improve the overall performance.

The effect of different clustering methods is also evaluated by comparing two well-studied clustering approaches, AGNES and k-medoids. The result shows that there is no significant difference; with these two clustering methods, this may indicate that the underlying clustering method does not have a substantial effect on CluPaTra.

Up to this stage, CluPaTra is bounded by limitations due to its reliance on sequence structure representation and sequence alignment to calculate similarity. Search trajectories naturally have cycles, and a sequence representation of the search trajectory does not record the cycles. Hence the sequence representation may reduce its granularity and remove some important information. Sequence alignment inherits a

computational bottleneck whose worst-case time complexity is $O(m^2 \times n^2)$ (where m is the number of instances in the training set and n is the maximum length of the sequences). This sequence alignment may not be scalable for large size of instances with long search trajectories.

3.8 Chapter Summary

In this chapter, we propose and discuss CluPaTra, generic instance-specific automated parameter tuning framework. We describe the framework overview and its three main components: feature selection, similarity calculation and clustering method. In feature selection, we present the notion of search trajectory as a problem-independent feature and represent in two variance: exact sequence and transition sequence representation. For similarity calculation, we used pairwise sequence alignment and implemented it in two variants: basic and robust sequence alignment. As a clustering method, we applied a well-known agglomerative hierarchical clustering, AGNES.

From a series of experiments on three classical COP: Travelling Salesmen Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP), CluPaTra shows a significant improvement compared to a vanilla one-size-fits-all approach, ParamILS. Compared with existing instance-specific tuning using problem-specific features, CluPaTra shows a significantly superior or equal result.

Chapter 4

Pattern Mining Approaches for Instance-specific Automated Parameter Tuning

In the previous chapter, we discuss CluPaTra, a generic instance-specific automated parameter tuning framework using search trajectory as its generic feature. It represents search trajectory as two simple directed sequences: exact and transition sequences. CluPaTra performs a sequence alignment method to calculate the similarity score for each pair of instances. Sequence alignment works by comparing all possible alignments regardless of their lengths, start and end positions, and then chooses the best alignment as the alignment that maximizes the similarity score, which is the sum of the scores for matched symbols and gaps in the alignment. After having the similarity score, CluPaTra then clusters the instances using agglomerative clustering method and tunes each cluster using an existing one-size-fits-all configurator. For testing instances, CluPaTra simply returns the most similar cluster's configuration as the testing instance's configuration.

However, the experimental results on three classical Combinatorial Optimization Problems (COPs) confirm that CluPaTra provides a promising improvement compared to existing tuning methods. Due to CluPaTra's reliance on sequence repre-

sentation and sequence alignment to calculate similarity, it inherits some structural issues and computational bottleneck. Due to these limitations, CluPaTra works only on short instances and when the number of instances is small.

To overcome the limitations of CluPaTra, we propose CluPaTra-II, a more complex approach by modeling the feature extraction as a pattern mining problem. For feature extraction and similarity calculation, we design two new pattern mining algorithms: (1) SufTra, **Suf**fix tree for sequential search **Tra**jectory pattern extraction, and (2) FloTra, **Flo**wer graph mining for graph search **Tra**jectory pattern extraction. SufTra is constructed for search trajectory sequence representation while FloTra for graph representation. These approaches provide efficient extraction of compact and discriminative features of search trajectory and are capable of retrieving similarity measures across multiple segments. Using a pattern mining model, features extracted using SufTra and FloTra can efficiently and effectively form better and tighter clusters and hence improve the overall performance.

In this chapter, we discuss these two approaches. We first elaborate CluPaTra-II, a pattern mining framework for automated parameter tuning. We then present our two novel pattern mining approaches: SufTra and FloTra. with SufTra as a pattern mining technique via suffix tree and FloTra for graph pattern mining for search trajectories. We then describe the experimental setting and result. Finally, we conclude by summarizing the chapter.

4.1 CluPaTra-II: Tuning Framework using Pattern Mining Approach

The CluPaTra dependency on sequential representation and sequence alignment to calculate similarity share the following limitations.

1. **Scalability.**

   Both sequence alignment techniques, basic and robust sequence alignment, are implemented using standard dynamic programming [51], with a complexity of

15L – 12L – 11L – 10L – 09L – 08L – 06L – 05L– …. – 04L – 04P – 04S – J – ..

14L – 13L – 11L – 10L – 09L – 08L – 06L – 05P – …. – 05L – 04L – 04S – J – ..

11L – 04L – 04P – 04S – 03S – J – 21L – 19L – 08L – 06L – 05L – … – J – 13L

Figure 4.1: Similarity Patterns from three search trajectory sequences

$O(n^2)$ where n is the maximum sequence length of the sequences. To cluster instances, we need to compute similarity scores for all possible pairs of training instances. Hence, the total time complexity for sequence alignment is $O(m^2 \times n^2)$, where m is the number of instances in the training set and n is the maximum length of the sequences. This poses a serious problem for instances with long search trajectories and when there is a large number of instances.

2. **Flexibility.**

   The process of sequence alignment is aligning a pair of sequence segments that gives the highest alignment score, when it is possible that the sequences, especially for long sequences, share similarities in more than one segment. Sequence alignment is not flexible enough to capture multiple-segment alignment with an acceptable time complexity.

   As an example, Fig 4.1 shows three search trajectory sequences. The boxes represent the similar patterns found in these three search trajectories. Using the CluPaTra similarity calculation method, we may conclude that instance 1 and 2 are similar and belong to the same cluster because the similarity score for instance (1) and (2) is 4 while the similarity score for other pair of instances is 3. If we examine the search trajectories further, we may discover other similar patterns and observe instance (1) and (3) actually share a higher number of similar patterns instead of instance (2) because instance (1) and (3) have matching symbols in two segments. Hence, instance (1) and (3) should belong to the same cluster, while (2) should be in a different cluster.

3. **Descriptiveness.**

   An important aspect of the fitness landscape that can be captured from the search trajectory is its local optima. A local optimum is an optimal (either maximal or minimal) solution with regards to its direct neighboring set of solutions [55]. It may or may not be the global optimum. General meta-heuristic algorithm should avoid being trapped in local optima. But a strong local optimum may force the search process to continuously return to this local optimum. Identifying the local optimal is quite essential to improve the performance of the target algorithm [88].

   Although local optima information can be extracted from the search trajectory, it can not be represented in sequence representation. When the target algorithm returns to a position that has been found previously, it only adds a 'C' symbol to the sequence but does not point the cycle solution. Hence, the use of sequence representation may result in a loss of the search trajectory structural pattern.

   For example, Fig. 4.2 shows the sequence and graph representation for three search trajectories of Quadratic Assignment Problem (QAP) instances. The three sequences have many similar subsequences (Fig. 4.2a) but the real search trajectories (as shown in Fig. 4.2b) are different; two search trajectories have a smoother search while the other one has many cycles.

As an attempt to answer these limitations, we propose a new tuning framework using a pattern mining approach which we refer as CluPaTra-II. Similar to CluPaTra, CluPaTra-II works in two phases: training and testing. The framework is illustrated in Fig. 4.3 and the steps involved in training and testing phases are shown in Fig. 4.4 and Fig. 4.5 respectively. The training phase works by first representing the search trajectory as a directed sequence (for SufTra) or graph (for FloTra). Instead of using the sequences or graphs directly to calculate the similarity score as in CluPaTra, we extract a set of compact features from search trajectories by using frequent pattern mining techniques.

(a) Sequence Representation of Search Trajectories

9L-8L-7L-7L-6M-9L-9L-9L-8L-8L-7L-6M-C-9L-8L-7L-7L-7L-7L-7L-7M-6M-C-6M-8L-8L-8L-C-...

9L-8L-7L-7L-7L-6L-6L-6M-6L-6M-C-6L-5L-5L-5L-5L-5L-5M-6L-5M-C-6L-5L-4M-6L-6L-7L-4M-...

9L-9L-8L-7L-7L-6L-6L-6L-6L-6L-5L-5L-5M-8L-7L-7L-5M-C-9L-7L-7L-7L-7L-6L-6M-5L-5L-5L-...

(b) Graph Representation of Search Trajectories

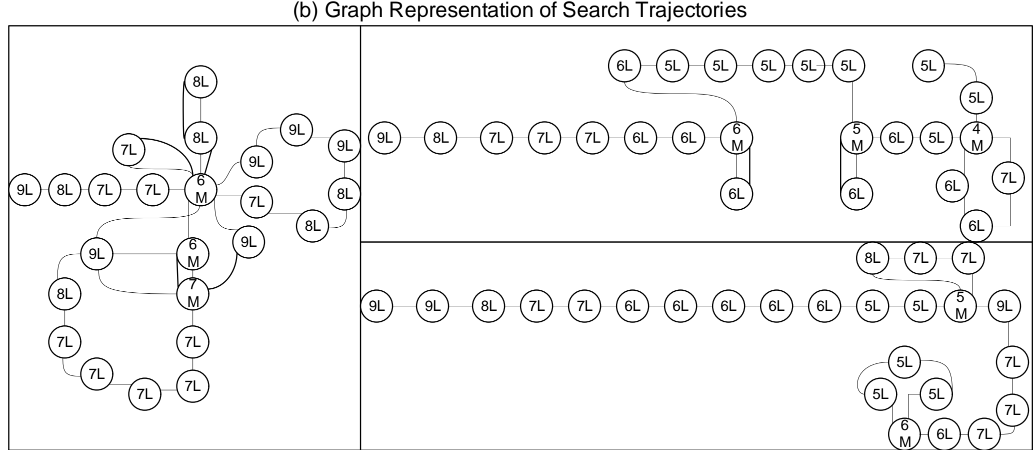


Figure 4.2: Sequence and Graph Search Trajectories Representation for three Quadratic Assignment Problem (QAP) instances

In frequent pattern mining, we find substructures (subsequences or subgraphs) that appear in a data set with a frequency of no less than a user-specified threshold (called $min_{support}$) [50]. In our setting, a set of items I is a set of search trajectories represented as a sequence s or a graph g. Since we want to find patterns from different search trajectories, we present the search trajectories in *vertical data format* (a set of sequences S or graphs G) and perform mining to find frequent patterns across search trajectories. We define the frequent pattern mining problem as follows.

Definition 12 (Frequent Pattern Mining [FPM]) *Given a set S of sequences or G graphs, a $min_{support}$ value and min_{size} value, the frequent pattern mining problem is to find all sub-sequences or sub-graphs of size at least min_{size} appearing in at least $min_{support}$ number of segments of S or graphs G.*

These sub-sequences or sub-graphs are used as distinctive features to describe the instances characteristics.

We construct two novel pattern mining approaches: SufTra and FloTra. SufTra utilizes the Suffix Tree structure to retrieve features in linear computational time,

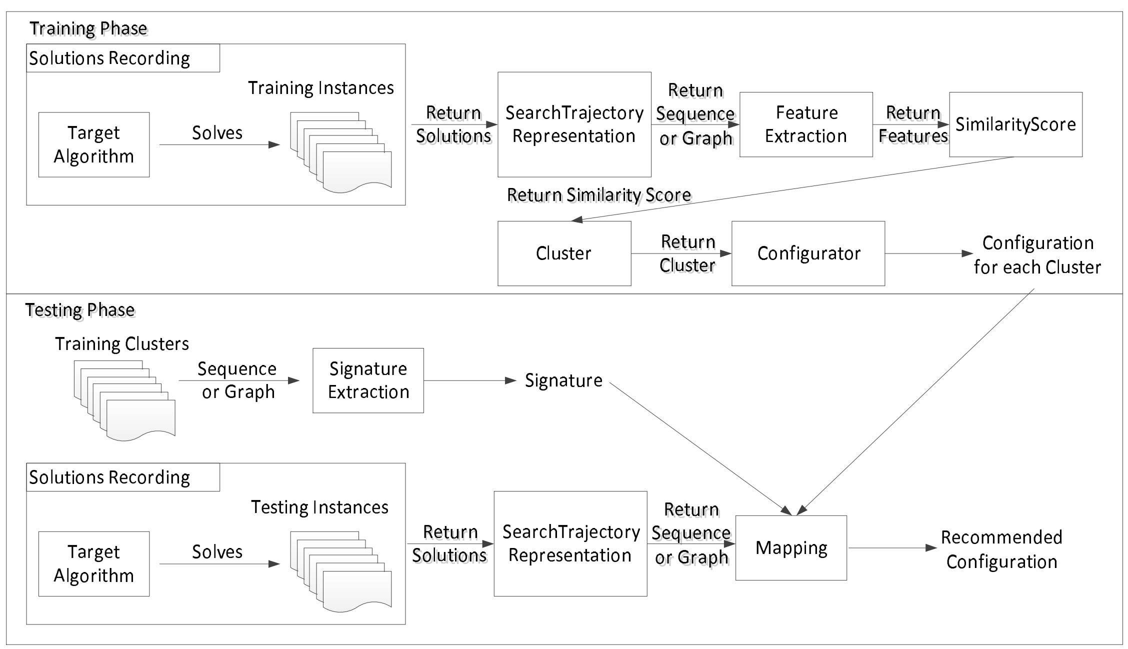
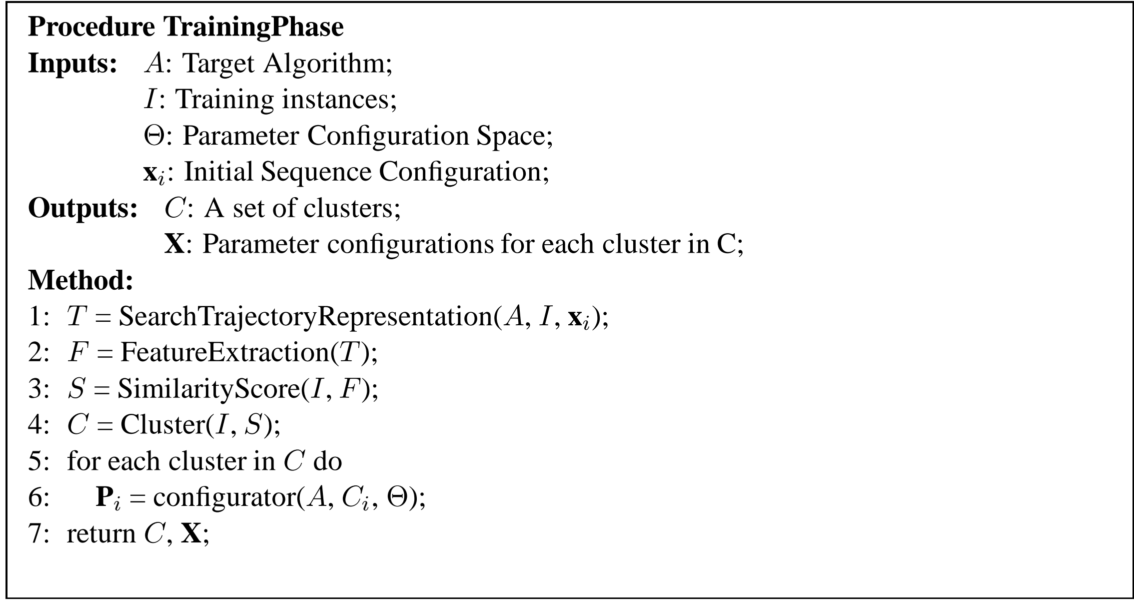


Figure 4.3: Tuning Framework using Pattern Mining Approach

whereas FloTra mines features from search trajectory graph. SufTra and FloTra extract the features and construct an instance-feature metric which correlates instances with each feature. SufTra and FloTra details are described separately in the following subsections.



Procedure TrainingPhase
Inputs: A: Target Algorithm;
I: Training instances;
Θ: Parameter Configuration Space;
$\mathbf{x}_i$: Initial Sequence Configuration;
Outputs: C: A set of clusters;
$\mathbf{X}$: Parameter configurations for each cluster in C;
Method:
1: T = SearchTrajectoryRepresentation(A, I, $\mathbf{x}_i$);
2: F = FeatureExtraction(T);
3: S = SimilarityScore(I, F);
4: C = Cluster(I, S);
5: for each cluster in C do
6: $\mathbf{P}_i$ = configurator(A, C_i, Θ);
7: return C, $\mathbf{X}$;

Figure 4.4: Tuning Framework using Pattern Mining Approach Training Phase

To calculate the similarity for each pair of instances from the instance-feature metric, we use cosine similarity, a widely-used similarity measure for comparing vectors

Procedure TestingPhase
Inputs: A: Target Algorithm;
I_t: A set of Testing instance;
C: Set of clusters;
X: Parameter configurations for each cluster in C;
$\mathbf{x}_i$: Initial Sequence Configuration;
Outputs: $BestConfig$: A recommended configuration;
Method:
1: for each cluster in C do
2: $SIGN_i$ = SignatureExtraction(C_i);
3: end for;
4: for each instance in I_t do
5: $BestClust_i$ = Mapping(I_{ti}, A, $SIGN$);
6: $BestConfig_i$ = $\mathbf{X}[BestClust_i]$;
7: return $BestConfig$;

Figure 4.5: Tuning Framework using Pattern Mining Approach Testing Phase

[46]. Cosine similarity is equal to 1 when the angle is 0, and it is less than 1 when the angle is of any other value. Cosine similarity is formulated as:

$$similarity = \frac{\sum_{i=0}^{n}(I_1(f_i) \times I_2(f_i))}{\sqrt{\sum_{i=0}^{n} I_1(f_i)^2} \times \sqrt{\sum_{i=0}^{n} I_2(f_i)^2}} \tag{4.1}$$

where $I_1(f_i)$ and $I_2(f_i)$ are the scores from instance-feature metric for feature i of Instance 1 and 2 respectively.

CluPaTra-II then clusters the instances by a well-known clustering approach, AGNES with L method. Detail description of AGNES and L method is provided in subsection 3.2. A tuning process is then performed to find the best parameter configuration for each cluster. An example that illustrates the steps in the algorithm is shown in Fig. 4.6.

For new testing instances, we improve the matching process by proposing a new classification method to map testing instances to clusters. This method enables us to generate more accurate mappings with shorter computation time. In the testing phase, we use the knowledge from the training phase to return instance-specific configuration(s) for testing instances. This phase is usually performed online. To achieve this, we design a new method for fast and accurate testing instance mapping. Our proposed method consists of two steps:

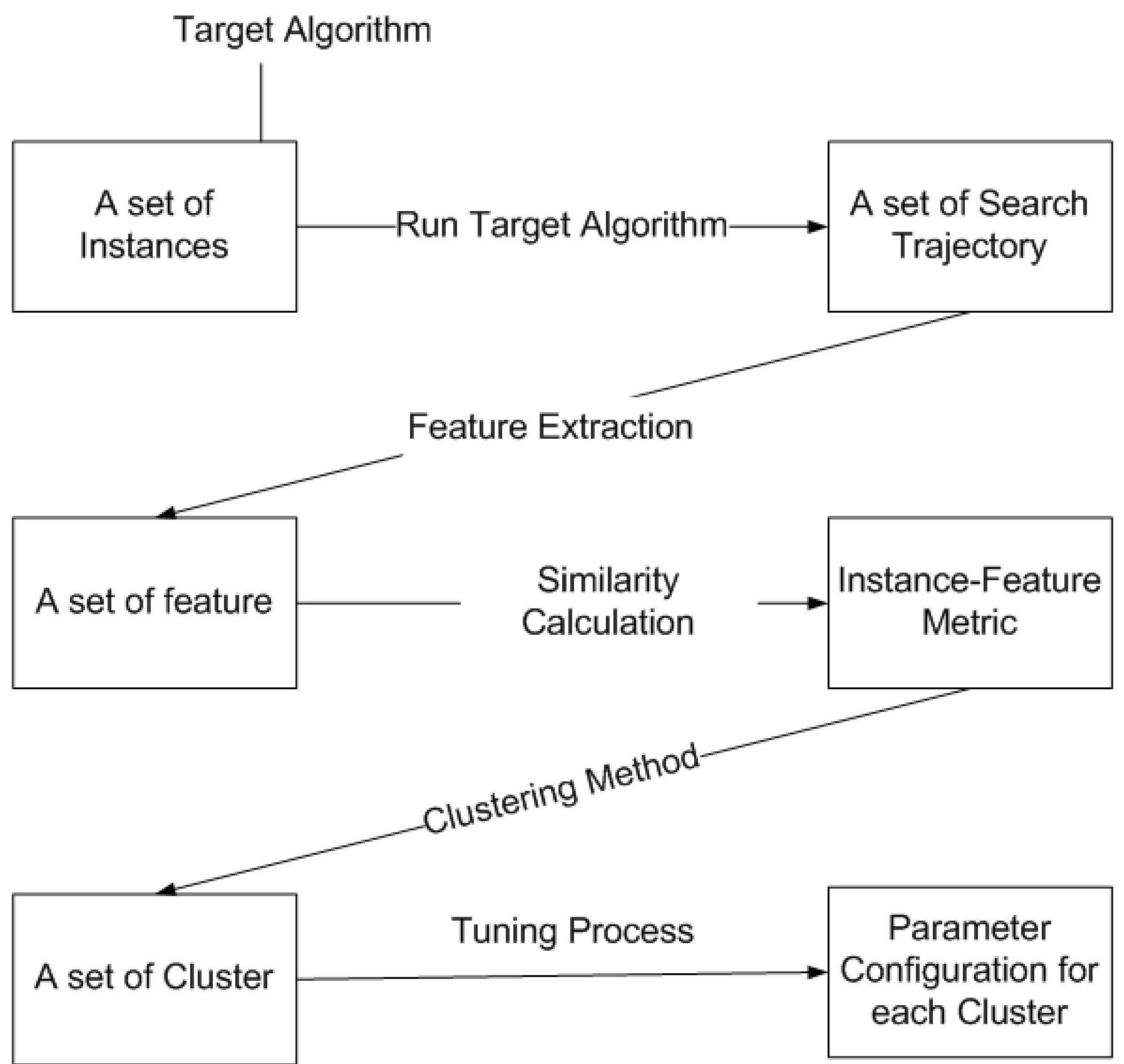


Figure 4.6: CluPaTra-II Steps and Output

1. Signature Construction. We construct the signatures for each cluster. This step is run once, and can be performed offline. The signature construction step is similar to feature extraction (in the training phase) but we need to run feature extraction for each cluster. We use these features as the cluster's signature.

2. Cluster Mapping. For an arbitrary testing instance, we match its search trajectory to the cluster's signature and return the parameter configuration from the best-matching cluster's as its parameter configuration. This step is performed online.

In the next subsections, we describe SufTra and FloTra in detail.

4.2 SufTra: Pattern Mining via Suffix Tree

SufTra utilizes suffix tree data structure to represent the search trajectories of a target algorithm. It extracts compact features from search trajectories for similarity calculation using the cosine similarity technique. SufTra addresses CluPaTra's limitations as follows:

1. Scalability: We propose a linear time algorithm for both Suffix Tree construction and traversal; and

2. Flexibility: We generate compact patterns from search trajectories and use them as features. The patterns may occur in multiple segments along the search trajectory, so suffix trees enable us to consider multiple-segment similarities to improve clustering accuracy.

In SufTra, we use the basic sequence representation of search trajectory as described in subsection 3.2. Here, we only explore one sequence representation. SufTra works in 4 stages: sequence hashing, suffix tree construction, features retrieval and instance-feature metric calculation. The details are as follows.

4.2.1 Sequence Hashing

In a search trajectory, several consecutive solutions may have similar solution properties before the final improvement to reach the local optimum (for example *04L-04L-04L-04L-04L-02P*). We therefore compress the search trajectory sequence to a *Hash String* by removing the consecutive repetition symbols and store the number of repetitions in a *Hash Table* to be used later in pair-wise similarity calculations. *Hash String* is the shorter version of the search trajectory after compressing all the repetition symbols. An example of *Hash String* from *04L-04L-04L-04L-04L-02P* the is *04L-02P*. If the sequence has a longer repetition, it should have a higher score because it contains more symbols. To store the number of repetition, we cannot simply encode it in the *Hash String* because it makes the symbol different if the repetition is different. Hence, we may lose some important features. To still include the repetition in the similarity score calculation and maintain the important feature, we use a *Hash Table* to store the repetition and calculate the repetition only to calculate the similarity score. In this example, the number of repetition of 04L is 5.

Removing consecutive repeated symbols gives us two advantages:

1. It offers **greater flexibility** for SufTra in capturing more varieties of similarity for symbol patterns between two instances. Two instances may share similar patterns (such as: *14L-5L*) but have different numbers of consecutive symbols, e.g., for $14L$ occurs 10 times in one instance and 5 times in another.

2. It **reduces computational cost** in constructing and traversing the suffix tree, since the time needed is decided by its length. Hash String is a more compact and shorter representation of the original search trajectory sequence.

After constructing Hash Table and removing repetitions, we convert the symbol for each solution to a single character and concatenate it into a string (Hash String).

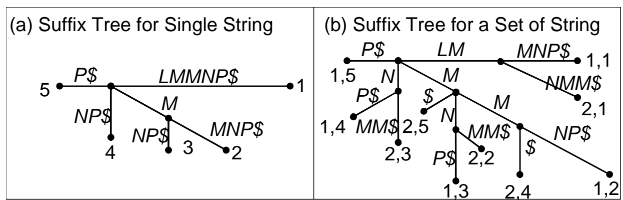


Figure 4.7: Example of Suffix Tree for a single string S_1 ($LMMNP$) and for two strings S_1=$LMMNP$ and S_2=$LMNMM$

4.2.2 Suffix Tree Construction

The search trajectory sequences found in the previous section is used to build a suffix tree. A suffix tree is a data structure that exposes the internal structure of a string for the particularly fast implementation of many important string operations. Suffix trees are used to solve exact and inexact matching problems in linear time and are widely used in substring problems [46]. The construction of a suffix tree proves to have a linear time complexity w.r.t. the input string length [46].

A suffix tree T for an m-character string S is a rooted directed tree having exactly m leaves numbered 1 to m. Each internal node, except for the root, has at least two children and each edge is labeled with a substring (including the empty substring) of S. No two edges out of a node has edge-labels beginning with the same character.

To represent suffixes of a set $\{S_1, S_2, ...S_n\}$ of strings, we use a $generalized$ suffix tree. A $generalized$ suffix tree is built by appending a different end of string marker (which is a symbol not used in any part of the string, such as *) to each string in the set, then concatenating all the strings together, and building a suffix tree for the concatenated string [46]. An example of a $generalized$ suffix tree for strings is $LMMNP$ and $LMNMM$ is $LMMNP * LMNMM*$. The time needed to build this suffix tree is proportional to the total length of all the strings. An example of a suffix tree for a single string S_1 and a set of string S_1 and S_2 is shown in Fig. 4.7.

In a suffix tree structure, we can easily retrieve matching substrings from a set of string by finding the branch that has leaves from corresponding strings. From our suffix tree example (Fig. 4.7b), branches with edge-label M, N, LM, MM, and MN have leaves from both string S_1 and S_2. These edge-labels represent the same substring

shared by S_1 and S_2. We use such common substrings to extract SufTra instance features.

We construct the suffix tree for the Hash Strings derived from search trajectories using the Ukkonen's algorithm [46]. We build a single $generalized$ suffix tree by concatenating all the Hash Strings together to cover all training instances. The length of the concatenate string is proportional to the sum of all the Hash String lengths. Ukkonen's algorithm works by first building an implicit suffix tree containing the first character of the string and then adding successive characters until the tree is complete. Details of Ukkonen's algorithm can be found in [46]. Our Ukkonen's algorithm implementation requires $O(n \times l)$, where n is the number of instances and l is the maximum length of the *Hash String*.

4.2.3 Features Retrieval

After constructing the suffix tree, we extract the frequent substrings. As described in Definition. 12, a substring is considered as frequent if it has a sufficient length and occurs in a significant number of strings [50]. The minimum number of length and occurrences is determined by $\min_{size}$ and $\min_{support}$.We apply a local search to provide sufficiently good values in reasonable times.

We use a first-improvement local search to move from initial values of $\min_{size}$ and $\min_{support}$ to their neighbors by changing either $\min_{size}$ or $\min_{support}$ at each move until the average distance among all instances in two different clusters are no longer improving. To find initial values of $\min_{size}$ and $\min_{support}$, we run a competition among 5 candidates, which are:

1. Lower bound of $\min_{size}$ and $\min_{support}$. We assume a good feature pattern should appear in more than one instance and contain more than one symbol, therefore, we set the lower bound value for both $\min_{size}$ and $\min_{support}$ to 2.

2. Upper bound of $\min_{size}$ and $\min_{support}$. To set $\min_{size}$ and the $\min_{support}$ upper bound, we observe the number of features extracted for different $\min_{size}$ and

$\min_{support}$ values. If $\min_{size}$ is more than 20% of maximum string length and $\min_{support}$ is more than 20% of the number of instances, most likely, we would not find any frequent substring. Therefore, we set the upper bound default value of $\min_{size}$ as 20% of maximum string length and the default value of $\min_{support}$ as 20% of the number of instances.

3. The middle value between the lower and upper bound.

4. First random value.

5. Second random value.

4.2.4 Instance-Feature Metric Calculation

After extracting the features, we calculate the instance's score for each feature and construct an instance-feature metric using the following rules:

1. if the instance does not contain the feature, the score is 0,

2. otherwise the score is calculated by summing up the number of repetitions for each symbol in the feature from the previously constructed *Hash Table*. A frequent substring may occur multiple times in one string. We calculate the score for each occurrence and choose the maximum score as the score for the instance-feature metric.

4.3 FloTra: Graph Pattern Mining for Search Trajectory

Representing the search trajectory with a sequence as in CluPaTra and SufTra suffers from the issue of descriptiveness due to their use of sequence representation model. CluPaTra and SufTra may oversimplify the search trajectory and lose finer granular details in some structural patterns.

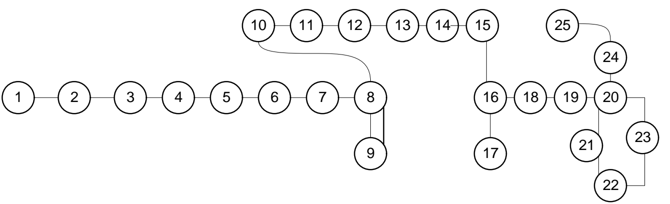


Figure 4.8: Flower Graph with stem, petals and thorns

To overcome this limitation, we introduce FloTra, a technique to uncover important patterns from search trajectory graph for generic instance-specific automated parameter tuning. FloTra constructs a graph representation of the search trajectory and conducts a graph pattern mining to discover specific and important patterns in the search trajectory. Using those patterns, FloTra then calculates instance-feature metric.

In FloTra, we represent a search trajectory as a graph. Each solution in graph representation is represented as symbol of two solution attributes: the position type and its performance metric as described in subsection 3.2. A node is a solution and an edge is the movement from one solution to another is presented as an edge as illustrated in Fig 4.8. A search trajectory graph is a special graph that has two distinctive structures: (1) a long skinny path representing solution movement from initial solution to end solution and (2) multiple short paths and loops representing the movement to or from local optima. The more loops in the graph, the stronger the local optima are.

The search trajectory graph can be considered as a flower-shape graph where the skinny long path is a stem and the short paths and loops are petals and thorns. In a flower-shape graph, we define the stem, thorns and petal as follows. Given a flower-shape graph, a stem is considered as a single long path from the initial node. An example of this stem is the path from the initial node (node 1) to the end leaf node (node 25) in Fig. 4.8. A petal is defined as a short path from any node along the stem that returns to the same node, while a thorn is a short path that does not return to the same node. To differentiate petals and thorns from stems, we assume that petal and thorn lengths should be shorter than stems. This is based on our observation of actual search trajectory graphs where we find that petals and thorns are shorter than stems.

Table 4.1: Average length of Stem, Thorn and Petal

Parts	Average Length
Stem	45
Petal	12
Thorn	6

The average length of stems and petals and thorns is shown in Table 4.1.

An example of a petal and thorn in Fig 4.8 is the path 8-9-8 and 17-18 respectively. To efficiently mine frequent patterns (subgraphs) from search trajectory graphs and calculate similarity scores for each pair of instances, we construct a feature extraction and similarity calculation method that exploits the graph distinctive structures.

The aim of FloTra is to find a set of frequent patterns (subgraphs) from a set of search trajectory graphs. As described in Definition. 12, FloTra has two parameters: $\min_{size}$ and $\min_{support}$. $\min_{size}$ determines the minimum subgraph length (which is translated to the minimum length of a stem and the maximum length of thorns and petals) whereas $\min_{support}$ determines the minimum number of graphs that contains a frequent subgraph. In this thesis, the values of $\min_{size}$ and $\min_{support}$ are fixed beforehand.

FloTra works in four stages. It first mines short frequent paths (thorns and petals) from all nodes, except the initial node. It then continues to mine long skinny paths (long stems) from the initial node. After having a set of thorns, petals and stems, FloTra then assembles the thorns, petals and stems together and extracts these as features. Finally, FloTra constructs the instance-feature metric. Details are as follows.

4.3.1 Stage 1: Mining Flower Thorns and Petals

To find petals and thorns, we only select nodes which are visited more than once in the search process. Hence, the number of edges must be greater than one. We first enumerate all the paths from the selected nodes using the Depth-First Search (DFS) algorithm [32]. One node may have several different DFS paths as shown in Fig 4.9.

For paths with length less than $\min_{size}$, we construct a Suffix Tree structure as

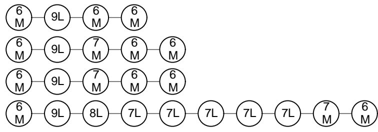


Figure 4.9: DFS Path for a particular node in search trajectory graph

in SufTra. This suffix tree is used to mine similar thorns and petals across different instances. To avoid redundancy, we only insert the same path once and we run a checking mechanism before inserting it. We then retrieve frequent substrings from different search trajectory graphs that occur more than $min_{support}$ as frequent patterns for flower thorns and petals. The details of this method are shown in Fig 4.10.

Procedure create_flower_thorn_petal
Inputs: G: Graph;
$min_{support}$: min support;
c_{size}: max cycle length;
Outputs: P_{freq}: a set of frequent flower thorn and petal;
Method:
1: Let $S = \emptyset$
2: For graph $g \in G$
3: Let n = node $\in g$ where edge > 2
4: For each $n \in g$
5: Let $P = generate_path_using_DFS(n)$;
6: For each path $p \in P$
7: if not $check_already_exists(p, S)$
8: $insert_to_suffixtree(p, S)$;
9: Let $P_{freq} = retrieve_frequent_substring(S, min_{support})$;
10: $sort(P_{freq})$
11: Output P_{freq};

Figure 4.10: Create Flower Thorns and Petals Procedure using Suffix Tree

4.3.2 Stage 2: Mining Long Stem

Aside from flower thorns and petals, another important structure that we want to retrieve is a long stem structure. The process is similar to stage 1. We first enumerate all paths from the initial node using a DFS algorithm [32]. For paths with lengths equal to or more than $\min_{size}$, we construct a Suffix Tree and find all frequent paths. We retrieve the frequent substrings from different search trajectory graphs that occur

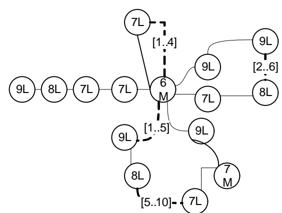


Figure 4.11: Example of frequent subgraph found by FloTra

more than $min_{support}$ as frequent patterns for long stem.

4.3.3 Stage 3: Assembling the Flower

At this stage, we assemble the flower thorns and petals from stage 1 with the long stem set from stage 2. For each long stem set that contains the node in the flower thorn and petal set, we attach the flower thorn and petal and consider it as a new candidate pattern. If the new candidate occurs no less than $min_{support}$ times, we accept it as a frequent pattern. Because frequent paths from both previous stages are generated from multiple segments in search trajectory, the assembling process may discover some gaps among those frequent paths. We allow these gaps and calculate the minimum number of gap and the maximum number of gap in between nodes as shown in Fig 4.11. The solid edge represents a direct path while the dashed edge represents a gap with the minimum and maximum number of nodes in between. After assembling the flower, we set all the found frequent pattern features if it occurs in at least $min_{support}$ number of graphs.

4.3.4 Stage 4: Instance-Feature Metric Calculation

After extracting the features, we calculate instance's score for each feature and construct an instance-feature metric by setting the score to 0 if the instance does not contain the feature, or otherwise to 1.

4.4 Empirical Experiment Result

We conduct a series of experiments to investigate the performance of CluPaTra-II with SufTra and FloTra. We apply CluPaTra-II for three Combinatorial Optimization Problems (COPs): Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP). As in CluPaTra, we use the Iterated Local Search (ILS) algorithm [49] for TSP, hybrid Simulated Annealing and Tabu Search (SA-TS) algorithm [87] for QAP and tabu-search algorithm [85] for SCP. These three algorithms have four parameters to tune. We use the same experiment measurement and setting as in CluPaTra. The details of the target problem and the algorithm, experimental measurement and setting have been described in Chapter 3 (section 3.6).

We compare the CluPaTra-II experiment result to CluPaTra-Tran, the most efficient (with respect to time and quality) instantiation of CluPaTra, and the ISAC result. To investigate the effectiveness of CluPaTra-II - FloTra in extracting features from the search trajectory graph, we also compare CluPaTra-II - FloTra with a well-known graph mining algorithm, gSpan [117]. We replace the FloTra feature extraction method with gSpan and compare the results.

4.4.1 Cluster Analysis

We first compare the clusters created from CluPaTra, ISAC, CluPaTra-II - SufTra, CluPaTra-II - FloTra and CluPaTra-II - gSpan for QAP and SCP. We use the same *ground truth* clusters as in CluPaTra. The cluster quality is shown in Table. 4.2. Notice that CluPaTra-II - FloTra has the highest cluster quality.

Next, we provide some insights on how CluPaTra-II generates a good feature from the problem instances. For this purpose, we investigate the signature features for each cluster.

We run CluPaTra and CluPaTra-II using FloTra to cluster 10 random instances of QAP from the ground-truth clusters [105] (random and uniform distances and flows; random flows on grids; and real-life problems). We then generate the features (sig-

Table 4.2: CluPaTra-II with SufTra and FloTra Cluster Analyses Comparison

	QAP		SCP	
Technique	Training	Testing	Training	Testing
CluPaTra-Tran	0.68	0.70	0.75	0.60
CluPaTra-II - SufTra	0.90	0.93	0.85	0.78
CluPaTra-II - FloTra	**0.95**	**0.96**	**0.86**	**0.79**
CluPaTra-II - gSpan	0.92	0.94	**0.86**	0.78
ISAC	0.80	0.80	-	-

Boldface indicates the best cluster quality.

natures) from each cluster. For CluPaTra, we generate the signatures using sequence alignment while for FloTra, we use the graph mining algorithm.

We illustrate the signatures in Fig. 4.12. In CluPaTra, cluster 1 has the smoothest search trajectory signature compared with the other two clusters. In cluster 1, the local search is able to guide the search towards a better solution without restarting which is shown by the signature that moves from position ledge (L) to position ledge (L) with lower $Best$ until it finds local minimum (P). The other two clusters have a more rough search trajectory that makes the search harder. It is often trapped in a bad local optimum (e.g.: $08P$ and $05P$). Apart from the $Best$ values, there seems to be no significant difference between the signatures of these two clusters.

On the other hand, in FloTra, each cluster has unique features. Cluster 1 has a long stem with a petal which indicates that the search landscape is smooth. Cluster 2 has a long stem with more thorns and petals - which indicates that the instances have more than one local optimum which the local search is able to escape from using restart. Cluster 3 has a lot of thorns and petals from one node which indicates that this node is a strong local optimum which trapped the local search.

Using this observation, we conclude that CluPaTra is only able to differentiate cluster 1 from cluster 2 and 3 and unable to differentiate clusters 2 and 3; while FloTra is able to capture different unique signatures for clusters 1, 2 and 3. These FloTra's signatures are also consistent with the observation in [105]. Using these abilities to capture better signatures, FloTra is able to create better (more similar and tighter)

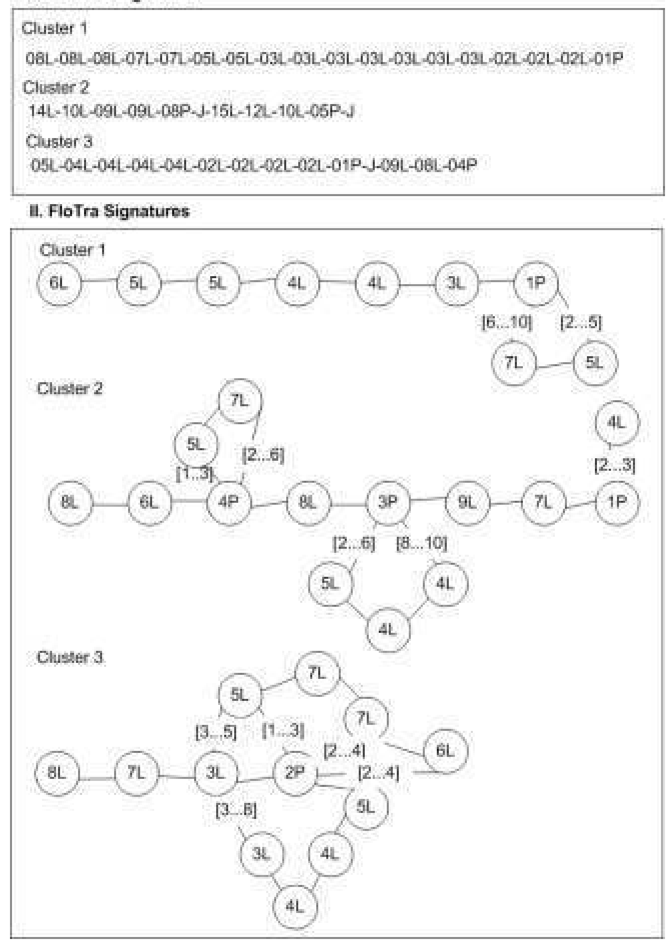


Figure 4.12: Example of clusters' signature for each cluster generated using FloTra

clusters and hence generate a more suitable parameter configuration for each cluster.

4.4.2 Computational Time

Next, we report the time needed (in seconds) for CluPaTra, CluPaTra-II - SufTra, CluPaTra-II - FloTra and CluPaTra-II - gSpan to form the clusters in training phase for TSP, QAP and SCP. For QAP and SCP, we use generated instances (Set B) while for TSP we use benchmark instances. Since we want to test the performance of SufTra and FloTra on long search trajectories and large sets of instances, we deliberately use generated instances for QAP (Set B) because the training and testing sets have large numbers of instances (100 instances for training and 400 instances for testing) with long search trajectories (average search trajectory = 15,536).

Table. 4.3 shows the result. SufTra is the fastest approach compared to other

Table 4.3: CluPaTra-II with SufTra and FloTra Computational Time Comparison

Technique	TSP Training	TSP Testing	QAP Training	QAP Testing	SCP Training	SCP Testing
CluPaTra-Trans	5.46 s	0.05 s	1,002 s	2,547 s	160 s	48 s
CluPaTra-II - SufTra	**3.01 s**	**0.02 s**	**56 s**	**146 s**	**15 s**	**8 s**
CluPaTra-II - FloTra	4.21 s	0.04 s	350 s	212 s	43 s	21 s
CluPaTra-II - gSpan	5.21 s	0.07 s	471 s	184 s	54 s	25 s

Boldface indicates the fastest approach.

Table 4.4: CluPaTra-II with SufTra and FloTra Performance Result Comparison

Technique	TSP Training	TSP Testing	QAP Training	QAP Testing	SCP Training	SCP Testing
CluPaTra-Trans	2.01	1.71*	1.87	2.08	0.78*	0.79
CluPaTra-II - SufTra	2.00	1.57*	0.83*	1.16*	0.35*	0.78
CluPaTra-II - FloTra	**1.98**	**1.25***	**0.78***	**1.07***	**0.27***	**0.52***
CluPaTra-II - gSpan	1.99	1.29*	0.80*	1.09*	0.31*	0.68*
ISAC	2.02	1.88	1.98	2.15	1.12	0.77

* = statistically significant against ISAC.
Boldface indicates the best performance result.

approaches, especially for QAP where SufTra is 18 times faster than CluPaTra.

4.4.3 Performance Comparison

Finally, we compare the target algorithm performance using parameter configurations from CluPaTra, ISAC, CluPaTra-II - SufTra, CluPaTra-II - FloTra and CluPaTra-II - gSpan. For QAP and SCP, we use generated instances (Set B) while for TSP we use benchmark instances. For the five instance-specific methods CluPaTra, ISAC, CluPaTra-II - SufTra, CluPaTra-II - FloTra and CluPaTra-II - gSpan, we use ParamILS [60] as a one-size-fits-all configurator. We measure the performance using performance metric as defined in Definition 3.

In Table. 4.4, we show the performance comparison results. Notice that CluPaTra-II - FloTra outperforms other methods in both training and testing instances.

Furthermore, depending on the structure of the search trajectory, the two meth-

Table 4.5: CluPaTra-II with SufTra and FloTra Comparison in Two Groups of Search Trajectories

Technique	Group without Cycles and Restarts			Group with Cycles and Restarts		
	Quality	Time (s)	Performance	Quality	Time (s)	Performance
CluPaTra-II - SufTra	**0.87**	**34**	**0.97**	0.86	**40**	0.94
CluPaTra-II - FloTra	0.91	163	1.03	**0.63**	182	**0.74**

Boldface indicates the best cluster's quality/time/performance result.

ods of CluPaTra-II may perform differently. To investigate the relative performance of these two methods, we run them under two different treatment groups of search trajectory structures. We retrieve the search trajectories from 20 QAP instances and transform them to sequences/graphs. For the first group, we remove all the cycles and restarts. For the second group, we retain them. Table. 4.5 shows the results of the two methods in terms of cluster quality, time and overall performance.

Notice here that SufTra performs slightly better than FloTra for the first group without cycles and restarts with much faster time. While in the group with cycles and restarts, the FloTra results are better. From these results, we claim that SufTra is best suited for search trajectories without (or with less) cycles and restarts, while FloTra is best for search trajectories with cycles and restarts.

4.5 Discussion

From the experimental results, we verify the performance of CluPaTra-II with SufTra and FloTra and observe a significant improvement in cluster quality, computational time and performance compared to its predecessor CluPaTra. CluPaTra-II with SufTra and FloTra also perform better than the existing instance-specific automated parameter tuning, ISAC.

On cluster quality, methods with the graph representation (CluPaTra-II with FloTra and gSpan) perform better than methods with sequence representation (CluPaTra and CluPaTra-II with SufTra). This implies that the graph gives a better represen-

tation of the search trajectory compared to the sequence and provides more reliable features. Hence, CluPaTra-II with FloTra and gSpan produce improved clusters compared to CluPaTra and CluPaTra-II - SufTra. We also notice that FloTra slightly outruns gSpan [117], a generic well-known graph mining method. This verifies that our graph mining approach, which is designed by considering specific search trajectory graph characteristics, is more suitable for search trajectory graph representation compared to the generic graph mining methods. Similar to cluster quality, regarding the performance result, CluPaTra-II - FloTra is also superior compared to CluPaTra and CluPaTra-II - SufTra. This further reinforces our claim that having more similar instances in smaller clusters eventually guides the tuning process to find better parameter configuration for each cluster.

Regarding computational time, CluPaTra-II - SufTra runs faster than other approaches especially for longer search trajectories and larger sets of instances, as in QAP. It is not surprising because CluPaTra-II - SufTra is naturally faster than any CluPaTra and CluPaTra-II - FloTra because it has a linear time complexity.

Based on these results, we claim that: (1) CluPaTra-II is a suitable approach for instance-specific configuration that significantly improves the performance with minor additional computational time; (2) CluPaTra-II - SufTra has overcome CluPaTra limitations in scalability and flexibility with a fast new efficient method for long search trajectories and large sets of instances, by producing better and tighter clusters faster; and (3) CluPaTra-II - FloTra overcomes the CluPaTra descriptiveness limitation by employing search trajectory graph representation to better identify instance features and produce better clusters compared to CluPaTra and CluPaTra-II - SufTra.

4.6 Chapter Summary

In this chapter, we discuss CluPaTra-II, a new tuning framework using a pattern mining technique as its feature extraction method. We introduce two new pattern mining techniques (SufTra and FloTra) for this purposes. SufTra extracts features from the

search trajectory sequence while FloTra, a more advanced technique, extracts features from the search trajectory graph. We then calculate similarity scores using cosine similarity calculation and cluster the instances using agglomerative clustering, AGNES. We then tune each cluster with a one-size-fits-all configurator. For the testing phase, we also construct a new mapping technique to find better clusters for each testing instance in less computational time.

We performed experiments on three COPs: Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP). From our experimental results, we verify that CluPaTra-II with SufTra and FloTra mines more suitable features with less computation time compared to CluPaTra. With better features, CluPaTra-II with SufTra and FloTra generate tighter clusters and thus result in improved performance.

Chapter 5

Web-based Automated Parameter Tuning Workbench

In the previous two chapters, we introduce CluPaTra and CluPaTra-II, frameworks for instance-specific automated parameter tuning. CluPaTra is the earlier version that relies on sequence alignment for similarity calculation, while CluPaTra-II overcomes CluPaTra's limitations on scalability, flexibility and descriptiveness by modeling the feature extraction mechanism as a pattern mining problem to capture compact and meaningful features from a search trajectory. In CluPaTra-II, we design two techniques for feature extraction: SufTra and FloTra. SufTra is a pattern mining technique which utilizes the Suffix Tree structure for search trajectory sequences while FloTra is a graph mining technique based on search trajectory graph characteristics. SufTra and FloTra extract meaningful features for tuning purposes.

In our empirical experiment result for three COPs: Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP), we show that CluPaTra and CluPaTra-II give encouraging improvements in cluster quality, computational time and solution performance. We also notice that for a large number of instances with long search trajectory, such as in QAP instances, CluPaTra-II - SufTra provides the fastest computational time compared to other approaches with comparable or even better performance.

To distribute and make this instance-specific tuning accessible, we design a web-based workbench for automated parameter tuning. We integrate CluPaTra and CluPaTra-II with parameter-space reduction method, Fact-RSM [45], and global (one-size-fits-all) parameter tuning, ParamILS [60] and F-Race [10], and construct AutoParTune. CluPaTra, CluPaTra-II and Fact-RSM are considered as preprocessing components for global parameter tuning.

In this chapter, we'll discuss AutoParTune in detail. We will begin with an overview of AutoParTune, and followed by the description of various AutoParTune components. We then discuss the major challenges for AutoParTune and the techniques to overcome these challenges. Next, we will describe the design architecture of AutoParTune. We then present the experimental results using AutoParTune in two industrial case studies. Finally, we provide a summary of this chapter.

5.1 AutoParTune Overview

It is stated that an ideal automated parameter tuning should have at least three characteristics: scalability, instance-specificity and problem structure exploration [71]. Scalability focuses on enabling the configurator to handle large parameter search spaces while instance-specificity focuses on producing different parameter configurations for different problem instances by exploring the problem structure (i.e. features) of the underlying problem instances.

Extending and implementing the work in [71], we design AutoParTune, a **web-based workbench** for automated parameter tuning, which is hosted in http://research.larc.smu.edu.sg/autopartune/index.aspx. AutoParTune consists of three components : instance-specific tuning, parameter search space reduction and global tuning.

- **Instance-Specific Tuning**

  In instance-specific tuning, instances are clustered according to a generic feature, search trajectory prior to the tuning process. This is an important pre-

processing step that provides a better parameter configuration for each instance while maintaining a minimum tuning time. To attain an instance-specific tuning component, we implement CluPaTra and CluPaTra-II.

- **Parameter Search Space Reduction**

  A large parameter search space uses a large amount of tuning time and sometimes misleads the tuning process. Reducing the parameter search space is a very critical preprocessing step that will reduce the overall tuning process, yet still provides a better parameter configuration. For parameter search space reduction, we apply the Fact-RSM technique, presented in [45], which is based on design of experiment (DoE), a well-established statistical approach that involves experiment designs for empirical modeling processes (see for example [83]).

- **Global Tuning**

  Global tuning is the kernel of AutoParTune. It produces the best parameter configuration for training and testing instances. As a global tuning component, we embed ParamILS [60] and iterated F-Race [10].

With two preprocessing components (instance-specific and parameter search space reduction) and global tuning component, AutoParTune is able to design five tuning strategies as described in Table. 5.1. Due to instance-specific tuning method limitations, which can only be implemented for local-search based target algorithms, Strategy 3, 4 and 5 in Table. 5.1 can only be used for local-search based target algorithms whereas Strategy 1 and 2 in Table. 5.1 can be used for a broader range of meta-heuristic target algorithms. Fact-RSM can only be applied for numerical parameters. Hence, Strategy 2, 4 and 5 in Table. 5.1 can only be used for numerical parameters.

AutoParTune is designed as a web-based workbench to address the needs for easy access to automated parameter tuning algorithms. Although there has been increasing interest for parameter tuning, an easy to use automated

Table 5.1: Five Tuning Strategies in AutoParTune

Strategy No.	Instance-specific (I)	Search Space Reduction (2)	Global Tuning (3)	Process Order
1	No	No	Yes	(3)
2+	No	Yes	Yes	(2)-(3)
3*	Yes	No	Yes	(1)-(3)
4*+	Yes	Yes	Yes	(1)-(2)-(3)
5*+	Yes	Yes	Yes	(2)-(1)-(3)

* = Only for local-search based target algorithm.
+ = Only for numerical parameters.

parameter tuning algorithm is not yet available. Existing approaches such as ParamILS (http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/) and CALIBRA (coruxa.epsig.uniovi.es/ãdenso/file_d.html) are publicly available, and these are usually compiled in an executable file (Windows or Linux compatible) along with how to use documentation. To use these executable files for tuning, we need to configure several settings by carefully reading their documentation. In AutoParTune, the tuning complexity setting is replaced by an easy-to-use and interactive web interface which makes it easier to understand and navigate.

One advantage of AutoParTune is that the tuning workload is shifted to the AutoParTune server. Users are only required to upload the necessary files and determine the tuning option in order for AutoParTune to run the tuning process on its server. This tuning process may require a lot of computational time and resources depending on the speed of the target algorithm. Once the tuning process is completed, an email message with the tuning result is sent to the requester. Hence, users of AutoParTune are freed from the complexity of the tuning process as well as the CPU and memory limitation of running the tuning process on their local machines which usually lacks the required computational power.

Another advantage of AutoParTune is its flexibility which allows for the addition of new techniques for its three components. New techniques for instance-specific, parameter search space reduction and global tuning can be added in AutoParTune without additional modifications. The new techniques just need to follow the AutoParTune format as described in Table. 5.2.

5.2 AutoParTune Components

AutoParTune has three components, namely: (1) instance-specific tuning, (2) parameter search space reduction; and (3) global tuning. The details of these components are discussed in the following subsections.

5.2.1 Instance-Specific Tuning

For instance-specific tuning, we implement CluPaTra and CluPaTra-II. CluPaTra and CluPaTra-II are premised on the assumption that an algorithm configuration is correlated with its fitness landscape, i.e. a configuration that performs well on a problem instance of a certain fitness landscape will also perform well on another instance with similar topology [92]. Furthermore, since the fitness landscape is difficult to compute, it can be approximated by a search trajectory [48, 49] which is deemed a probe through the landscape under a given algorithm configuration.

CluPaTra works by transforming the search trajectory as a directed sequence and uses sequence alignment to calculate similarity for each pair of instances. On the other hand CluPaTra-II is an extension of CluPaTra that overcomes three major limitations of CluPaTra: scalability, flexibility and descriptiveness. CluPaTra and CluPaTra-II are described in detail in Chapter 3 and 4 respectively.

Up to this stage, CluPaTra and CluPaTra-II can only be applied on local-search based target algorithms due to its reliance on search trajectory. A search trajectory generator is required to perform CluPaTra and CluPaTra-II.

5.2.2 Parameter Search Space Reduction

An often neglected preprocessing step in automated parameter tuning is to reduce the parameter space into a specific favorable parameter range. A good initial parameter range is able to *guide* the tuning process to provide a better parameter configuration with shorter computation time.

For the parameter search space reduction component, we apply the Fact-RSM [45]

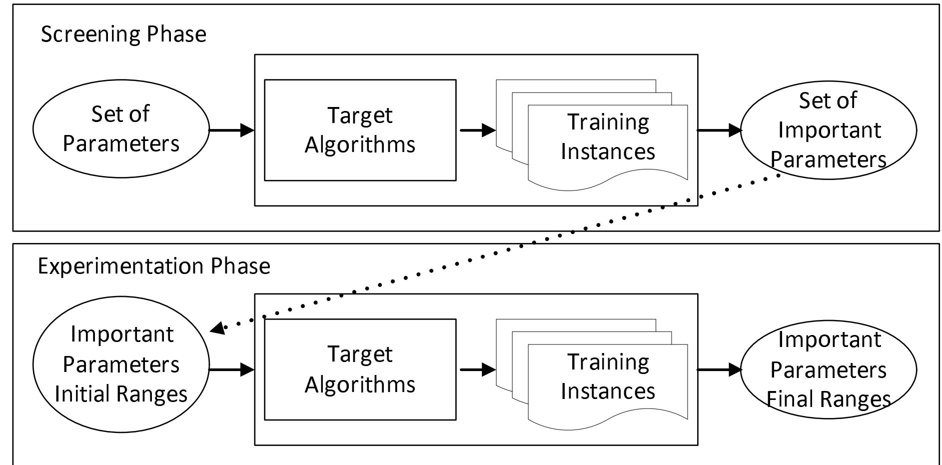


Figure 5.1: Phases of Fact-RSM, Parameter Search Space Reduction Method using DoE methodology

technique. Fact-RSM is a sequential experimental method for screening and reducing a parameter space for numerical parameters. Fact-RSM is based on the DoE (Design of Experiment) methodology as follows.

A full factorial experiment design is applied to first screen and rank the parameters. Parameters which are determined to be unimportant (i.e. the solution quality is insensitive to the values of these parameters) are set with constant values that reduce the parameter space to be explored. A first-order polynomial model based on RSM (Response Surface Methodology) is then built to define the promising initial range for the important parameter values. For statistical calculation, we use a well-known statistical software, **R** (http://www.r-project.org/).

Fact-RSM, as illustrated in Fig. 5.1, works in two phases: screening, and experimentation. The screening phase identifies the important parameters using 2^k full factorial design, while the experimentation uses RSM to locate ”promising” regions for important parameters. The details of the screening and experimentation phases are as follows.

Screening Phase

A screening process is conducted to determine which parameters are significantly more important to reduce the number of parameters under consideration. It applies a 2^k full factorial design which consists of k parameters, where each parameter x_i

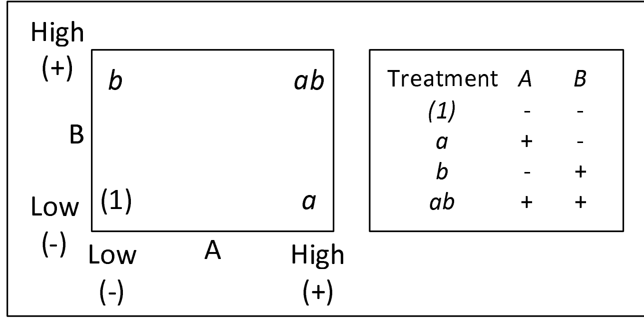


Figure 5.2: 2^k Full Factorial Design for Fact-RSM

only has two levels (a_i and b_i) with a_i as lower bound and b_i as upper bound.

As an example, consider if there are two parameters, A and B. Fig. 5.2 shows the 2^2 design with treatment combinations represented as the corners of the squares. The signs - and $+$ denote the values of a_i and b_i of each parameter x_i, respectively. Treatment combinations are the Cartesian product of the two parameters values (a_i and b_i). A treatment combination is represented by a series of lowercase letters. For example, the treatment combination a indicates that parameters A and B are set to b_A and a_B, respectively. To estimate the treatment combination, we run the target algorithm for n replicates for each treatment combination. A complete design requires (2 x 2 x ... x 2) x n = n x 2^k. For simplicity, we set n to 10.

Since the main focus in this phase is to determine the important parameters, the interactions between parameters are ignored. The importance of a particular parameter is defined by conducting a significance test on the parameter's main effect. A significant level is set to 5% ($\alpha = 5\%$). Parameters with $p - value \leq \alpha$ are significantly important. The important parameters are explored further in the next phase. On the other hand, unimportant parameters are set to a constant value by looking at the main effect value of the parameter. If the value is negative, it is set to its upper bound, if otherwise, to its lower bound.

Experimentation Phase

This phase aims to find and locate "promising" regions for important parameters by using the Response Surface Methodology (RSM). RSM is a model-based approach within DoE that can be used to quantify the importance of each parameter, support

interpolation of performance between parameter settings as well as extrapolation to previously-unseen regions of the parameter space [59]. RSM has been used in the parameter tuning scenario to finetune algorithm parameters [26, 27] and to identify ”promising” regions of a parameter search space [59].

The underlying assumption in this phase is that the region can be approximated by a planar model (the first-order model). Since we might be at a region on the response surface that is far from the optimum, we assume that there exists only a little curvature. Hence a planar model would be appropriate. The planar model of the parameters is approximated using the following function:

$$Y = \beta_0 + \beta_1 x_1 + ... + \beta_m x_m + \varepsilon \tag{5.1}$$

where $\beta_0, \beta_1, ..., \beta_m$ are parameter coefficients, $x_1, ..., x_m$ are parameters, and ε is error coefficient.

In order to move rapidly to the ”promising” regions, we apply the method of steepest descent (for a minimization problem). This method is a procedure for moving sequentially along the path of steepest descent, that is, in the direction of the maximum decrease in response Y. For example, if β_1 (coefficient of parameter 1) is the largest absolute coefficient value compared against other coefficient values, the step size of another parameter i is calculated by β_1/β_i.

This phase is terminated when the local optimum region is found. From a statistical point of view, the local optimum can be indicated by the existence of either interaction or curvature. Interaction is tested using analysis of variance (ANOVA) while curvature is tested using the $t - test$.

5.2.3 Global Tuning

As a global tuning component, we implement two efficient and well-established global tunings: ParamILS [60] and F-Race [19]. ParamILS [60] utilizes Iterated Local Search (ILS) to explore the parameter space in order to find a good parameter configuration

based on the given training instances. ParamILS has been very successfully applied to tune a broad range of high-performing algorithms for several hard combinatorial problems with a large number of parameters. ParamILS is itself an iterated local search algorithm used for tuning discrete parameters. Since ParamILS works only with discrete parameters, in AutoParTune, we first discretize the values of the parameters if the target algorithm has parameters that assume continuous values.

Iterated F-Race [19] is a racing algorithm for the task of automated parameter tuning for categorical and numerical parameters. Iterated F-Race is an extension of F-Race [17] which is based on a statistical approach for selecting the best parameter configurations using stochastic evaluations.

5.3 AutoPartune Features

AutoParTune is designed as a web-based workbench that integrates three different components of automated tuning to enable easy and flexible tuning. As a web-based workbench, AutoParTune users are able to perform a parameter tuning by uploading the necessary files, including the target algorithm (in Windows executables format) and selecting a tuning strategy from the five AutoParTune strategies. AutoParTune strategies are based on the three components (instance-specific tuning, parameter search space reduction, and global tuning) which are assumed to be independent components. To fully implement AutoParTune as a web-based workbench, a number of features are provided to make sure that AutoParTune is working in a web environment.

5.3.1 Security Issue

To protect AutoParTune against web attacks, we implement two security mechanisms that prevent automated-agent perpetrators and perform checks on the files uploaded for virus and malicious codes. The details of this are as follows.

1. Email Authentication Mechanism

The purpose of email authentication is to validate the user's email address and prevent automated-agent perpetrators. After the user uploads the necessary files, an email verification is sent to the user's email account. The user needs to verify the email by visiting the link attached with the email before continuing the tuning process. The tuning is run only after the verification is completed.

2. Antivirus Scanning Mechanism

   An antivirus scanning mechanism is implemented in AutoParTune to check if the uploaded files are clean from virus, malware or other malicious programs. Before starting the tuning process, AutoParTune executes antivirus scanning on uploaded files directory. This process is run automatically using AVG Antivirus (http://free.avg.com/ww-en/homepage) command-line interface. If one or more uploaded files are considered as suspicious by the scanner, the tuning process is stopped. The files will be deleted and the respective user will be notified by email.

5.3.2 Integration Issue

Each component in AutoParTune is assumed as an independent component that adds a specific feature to the existing workbench. The components are developed independently using different platforms and programming languages, such as: C#, C++ and Java. Integrating these components requires a common protocol for communicating with each other.

To integrate these components and maintain communication between components, AutoParTune designs a controller function, which is called the AutoParTune Controller. Each component in AutoParTune is compiled as a Windows command-line executable file with standard input and output formats to ensure the communication connection to AutoParTune Controller. Some additional text files such as configuration files may be required by these components. AutoParTune Controller runs each component by calling a command-line executable syntax using its input format. After

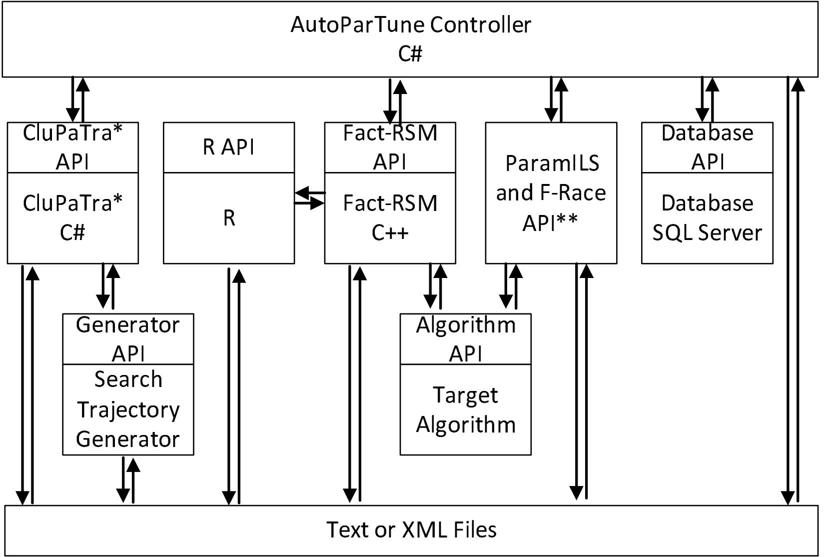


Figure 5.3: AutoParTune Components Communication Schema

the component execution is completed, AutoParTune Controller retrieves the result from the output command-line or text file. The detailed format for each component is shown in Table 5.2. The AutoParTune Controller decides which components to call based on the user's tuning strategy. It also retrieves and stores the tuning setting to a database.

For the target algorithm, the user needs to provide a Windows executable file with a standard input output format as described in Table 5.3. CluPaTra and CluPaTra-II need to call the search trajectory generator executable file to generate the search trajectory for each instance. On the other hand Fact-RSM, ParamILS and F-Race need to call the target algorithm executable file.

5.4 Application Architecture

We implement AutoParTune using a three-tier architecture, which is shown in Fig. 5.4. The presentation layer is hosted on Microsoft IIS server and contains AutoParTune web interface. The user interface is easily navigated with a step-by-step tuning upload process as described in the AutoParTune quick guide in Appendix A. The application layer contains the AutoParTune tuning logic as shown in Fig. 5.3.

Table 5.2: AutoParTune Components Input Output Standard

Component	Input and Output
CluPaTra and CluPaTra-II	**Input**: - trajectory generator executable - training instance file name - testing instance file name - random parameter configuration **Output**: instances' clusters file **Additional Files**: - training instance file - testing instance file - instance files
Fact-RSM	**Input**: - target algorithm executable - training instance list - parameter search space file name - training folder **Output**: new parameter search space file **Additional Files**: - training instance file - parameter search space file - instance files
R	**Output**: statistical result **Additional File**: R command file consists of data file name, anova test syntax, output file name
ParamILS and F-Race	**Input**: training folder **Output**: best tuning configuration in the last output line **Additional Files**: - scenario file for tuning setting - parameter search space file - instance files

Table 5.3: Search Trajectory Generator and Target Algorithm Standard for AutoParTune

Component	Input and Output
Search Trajectory Generator	**Input**: - Instance file name - seed - random parameter configuration **Output**: search trajectory file **Additional Files**: instance file
Target Algorithm	**Input**: - Instance file name - seed - random parameter configuration **Output**: best found objective value (displayed in the last line of the screen output) **Additional Files**: instance file

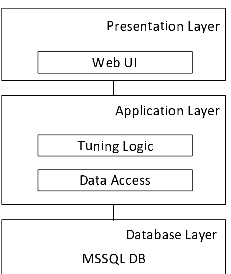


Figure 5.4: AutoParTune Design Architecture

The database layer is hosted on Microsoft SQL Server 2008. AutoParTune uses 4 tables to store tuning settings, training instances, testing instances and tuning results. The database design is shown in Fig. 5.5. For retrieving, inserting, updating and deleting the database, we use database Stored Procedures (SP). SP is a database subroutine that accesses a database system, which performs intermediate processing on the database server, without transmitting unnecessary data across the network. Using SP, AutoParTune can reduce the network usage between the user machine and server.

5.5 Empirical Experiment Result

To demonstrate the effectiveness of AutoParTune, we run a series of experiments on three Combinatorial Optimization Problems (COPs), namely: Traveling Salesman

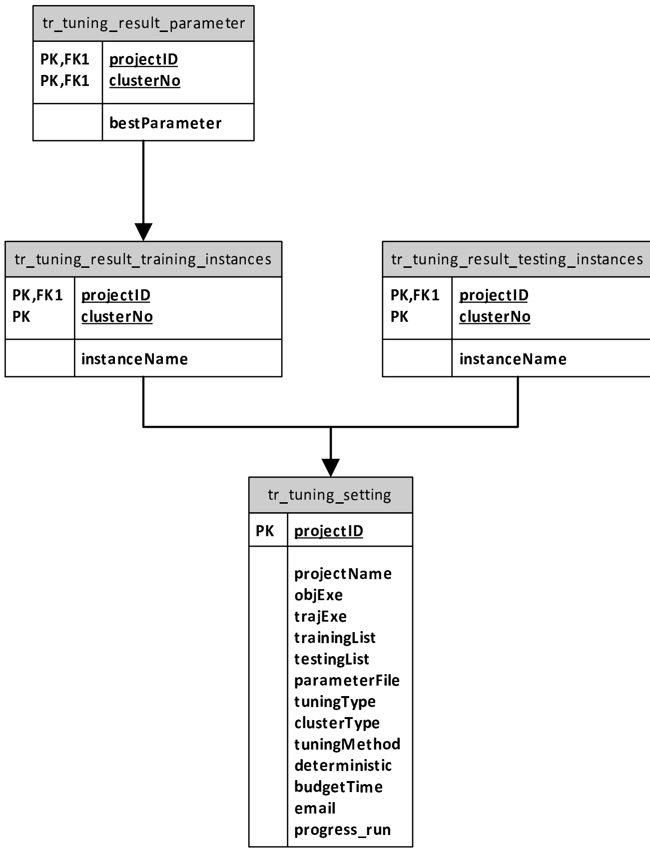


Figure 5.5: AutoParTune Database Design

Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP) and two industrial problems, namely: the aircraft spares inventory optimization problem and the theme park personalized intelligent route guidance problem.

We measure performance as the average of percentage deviation from the best found solution (Definition 3) and compare our experimental results with the best known values used by our industry partner. The cutoff time is set to 500 seconds per run and each configuration process is allowed to call the target algorithm for a maximum of 100 x n times, where n is the number of instances. To compare the significance of our result, we perform a t-test [83] and consider $p-values$ below 0.05 to be statistically significant ($\alpha \leq 5\%$).

5.5.1 Classical COPs

We compare the target algorithm performance using parameter configuration from AutoParTune strategy 2, 4 and 5. We do not conduct the experiment using strategy 1

Table 5.4: AutoParTune Performance Result Comparison for Classical COPs

Technique	TSP		QAP		SCP	
	Training	Testing	Training	Testing	Training	Testing
Strategy 2	2.07	1.37	0.87	1.23	0.76	0.78
Strategy 4	**1.88**	**1.13**	**0.71**	**1.06**	**0.22**	**0.49**
Strategy 5	1.91	1.24	0.81	1.13	0.42	0.45

Boldface indicates the best performance result.

(global tuner only) and 3 (instance-specific only) because the result is already shown in the respective chapter. We use the same experiment setting as in CluPaTra and CluPaTra-II. For QAP and SCP, we use generate instances (Set B) while for TSP we use benchmark instances. We measure the performance using performance metric as defined in Definition 3. In Table. 5.4, we show the performance comparison results. Notice that AutoParTune - Strategy 4 outperforms other methods in both training and testing instances.

5.5.2 Aircraft Spares Inventory Optimization Problem

We implement AutoParTune to tune an algorithm for an aircraft spares inventory optimization (minimization) problem of a large commercial aircraft maker based in Europe. Aircraft spares inventory optimization problem is a maintenance, repair and overhaul (MRO) operations problem faced by the aircraft total service support provider to meet target service levels with customers based on performance-based contracts. It is operated out of a network of airports. The problem objective is to determine the optimal inventory allocation strategy that can fulfill target services levels where optimality is defined in terms of minimal total life cycle costs for spares comprising inventory holding cost, part purchasing and repair cost, logistics delivery cost, while service levels are defined in terms of spares fill-rates.

This problem is solved using a *Simulated Annealing* (SA) algorithm [44], a local-search based algorithm which has 8 parameters that are used to control SA behavior as described in Table 5.5. The SA algorithm works as follows. It starts by creating one

Table 5.5: Parameters for SA on Aircraft Spares Inventory Optimization Problem

Parameter	Description	Range
maxSuccess	Maximum number of successes within one temperature	[100, 1000]
maxTries	Maximum number of tries within one temperature	[100, 1000]
maxComp	Maximum number of solutions generated	[1000, 50000]
maxConsReject	Maximum number of consecutive rejections	[100, 1000]
maxChangeG	Maximum change in a variable value when generating a new solution	[100, 1000]
maxTriesG	Number of tries to generate a feasible solution	[100, 1000]
coolingFactor	Factor to reduce the temperature by during each Temperature change	[0.5, 1]
oracleStrictness	A value to depict the strictness of the oracle function in accepting a new solution that has an objective value worse than the current one. A higher value would result in a higher rejection rate (e.g. a value of 100 would accept only better solutions)	[0, 100]

feasible initial solution. A new solution is generated by swapping n number of variable values where n is determined by $maxChangeG$ parameter. If the new solution is feasible, it computes the objective value and automatically accept it if the objective value is better than current best solution, if it is worse, it decides to accept or reject the new solution based on the $oracleStrictness$ parameter. It continues to generate a new solution until one of the termination criteria ($maxTriesG$, minimum Temperature, $maxConsReject$ or $maxTries$) is violated.

We apply our approaches on 50 synthetic instances based on real industrial instances. We randomly select 25 instances as training instances and the remaining 25 as testing instances. We compute the results using 5 strategies of AutoParTune on Table 5.6 and show the parameter configurations from AutoParTune on Table 5.7. To ease the experiment computation, we use CluPaTra-II - SufTra for instance-specific tuning and F-Race for global tuning component. We present the average percentage deviation value from the default (which is the best known value used by our industry partner).

The result shows that 5 strategies of AutoParTune give parameter configurations that generate solutions with lower objective values compared to the solutions from the default configuration (the percentage deviation values are negative). Most of the

Table 5.6: Aircraft Spares Inventory Optimization Problem Performance Result

Technique	Training	Testing
AutoParTune Strategy 1	-0.208	-0.375
AutoParTune Strategy 2	-0.569*	-0.471*
AutoParTune Strategy 3	-0.438	-0.557*
AutoParTune Strategy 4	**-0.898***	-0.634*
AutoParTune Strategy 5	-0.888*	**-0.676***

* = statistically significant against Default Configuration.
Boldface indicates the best performance result.

Table 5.7: Parameter Configurations for Aircraft Spares Inventory Optimization Problem

Parameter	Default	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
maxSuccess	100	200	500	900	300	400
maxTries	100	300	400	600	500	500
maxComp	1000	1500	5000	3000	2000	1000
maxConsReject	100	900	400	500	300	100
maxChangeG	100	300	400	500	700	100
maxTriesG	100	400	500	700	900	200
coolingFactor	0.95	0.55	0.70	0.50	0.90	0.80
oracleStrictness	30	20	70	60	10	90

results are statistically significant compared to the default configuration result. Notice that strategy 4 and 5 outperform other strategies and are statistically comparable to one another in that the percentage deviation values in strategy 4 are slightly better than those of strategy 5 in training instances and slightly worse in testing instances.

5.5.3 Theme Park Personalized Intelligent Route Guidance Problem

Our second industry problem is the theme park personalized intelligent route guidance problem that aims to provide a personalized route that maximizes the patron's experiences in the theme park for a given time constraint. The patron's experiences are measured by an utility function that factor in the patron's individual preferences as well as the statuses of current attractions such as service operation status and queue time. Hence, the objective of this problem is to maximize the utility function subject to:

1. A set of patron's attributes (attraction preferences, health issues and physical limitations).
2. A set of attraction's attributes (operation status, queue time, rank and accessibility).
3. A time duration.

This problem is solved using a heuristic algorithm which consists of 2 steps: utility mapping calculation and construction heuristic. Utility mapping calculation computes the temporal utility of each attraction (dynamic value versus time) based on patron's preferences. For each patron profile p, the utility U_{pit} of an attraction i in time duration t is a function of three subset factors, namely: critical subset (CS_{pit}), quantitative subset (QS_{pit}), and subjective subset (SS_{pi}), and could be expressed as:

$$U_{pit} = CS_{pit}[\alpha \times QS_{it} + (1 - \alpha) \times SS_{pi}] \tag{5.2}$$

where

Notation	Definition
CS_{pit}	Critical Subset Factor for attraction i and patron p on a specific time window t. This factor represents the attraction's restrictions, such as: opening hours, maximum weight, minimum height, and health restriction, that cannot be violated. The score is set to 1 if there is no violation, otherwise to 0.
QS_{it}	Quantitative Subset Factor which is a linear weighted sum of the attraction i factors: rank, service time, and queue time score in a specific time window t. It is calculated as: $QS_{it} = w_r rank_i + w_s servicetime_i + w_q queuetime_{it}$
SS_{pi}	Subjective Subset Factor which is a linear weighted sum of the factors: thrill, wet and dark suitability of patron p for attraction i. It is calculated as: $SS_{pi} = w_t thrill_{pi} + w_d dark_{pi} + w_w wet_{pi}$
$\alpha, w_r, w_s, w_q, w_t, w_d, w_w$	weight coefficient that is set between 0 and 1.

Using that utility score, a route which maximizes the overall utility is generated using the full-insertion construction heuristic. This heuristic inserts each unvisited attraction into the route at each possible location and then chooses the best insertion. For calculating the utility score, there are 7 weight coefficients which we consider as parameters that need to be set. We describe these parameters in Table 5.8.

To apply tuning on the theme park personalized intelligent route guidance problem, we designed two scenarios with two different data sets and tuned it separately. The first scenario focused on the tuning patrons subjective subset factor weights (w_t, w_d, w_w) while the second on quantitative subset factor weights (w_r, w_s, w_q). The scenarios are as follows.

Scenario 1: Patron's Subjective Subset Factor Weights

In this scenario, the tuning objective is to tune SS_{pi} weights such that the route, which consists of a set of attractions I, generated by personalized intelligent route guidance

Table 5.8: Parameters for Heuristic Algorithm on Theme Park Personalized Intelligent Route Guidance Problem

Parameter	Description	Range
α	weight coefficient for overall utility function	[0, 1]
w_r	weight coefficient for rank factor	[0, 1]
w_s	weight coefficient for service time factor	[0, 1]
w_q	weight coefficient for queue time factor	[0, 1]
w_t	weight coefficient for patron's thrill tolerance factor	[0, 1]
w_d	weight coefficient for patron's dark tolerance factor	[0, 1]
w_w	weight coefficient for patron's wet tolerance factor	[0, 1]

algorithm for a specific patron, satisfies the preferences of patron p. We assume that each patron decides to go to certain attractions based on the patrons own preferences (such as thrill, wet and dark preferences).

Given a set of patrons preferences P, the personalized intelligent route guidance algorithm generates the best route R_{algo} which consists of a set of attractions that match with preferences P. For each set of patron preferences P, there exists a set of patron "real" visited attractions R_{visit} as a "ground truth" set. The quality score is measured by comparing the set of attractions generated by the algorithm in R_{algo} with the "ground truth" set R_{visit}. It is calculated as the size of set intersection between a set of attractions generated by the algorithm and "ground truth" set ($|R_{algo} \cap R_{visit}|$). The route with a higher quality score is the better one.

We modify the basic tuning scenario in Fig. 2.2 to meet our needs and design a tuning scenario as illustrated in Fig. 5.6. The configurator calls the personalized intelligent route guidance algorithm (target algorithm) with a specific parameter configuration. The target algorithm generates a route for each patrons preference. "Quality Calculation method" compares the route with the "ground truth" and returns the quality score to the configurator. The configurator saves and examines the route quality for a given parameter configurator. The process continues until the configurator finds the best parameter configuration.

We apply AutoParTune on this tuning scenario to tune w_t, w_d, w_w. We set other parameter to a fixed value. We use the preferences of 48 real patrons and visited attractions gathered in a ground survey conducted on June 2012 at the largest theme

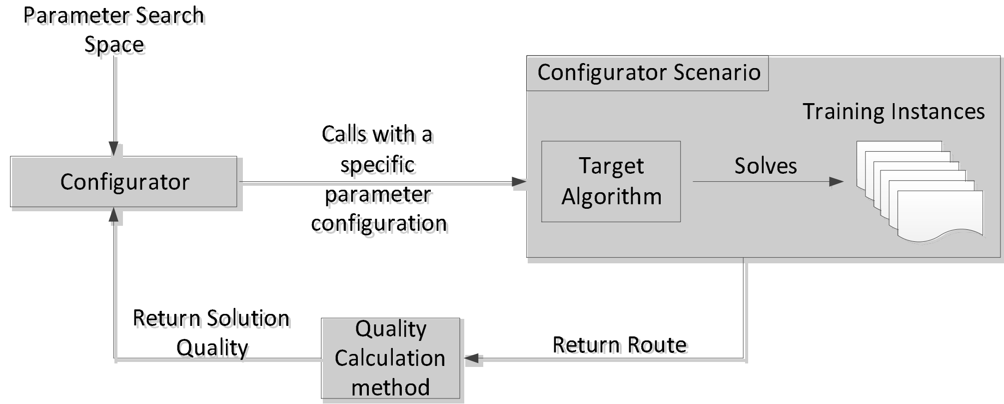


Figure 5.6: Tuning Scenario for Personalized Intelligent Route Guidance

Table 5.9: Parameters Configurations for Theme Park Personalized Intelligent Route Guidance Problem

Parameter	Default	Strategy 1	Strategy 3
α	0.1	0.2	0.1
w_r	0.2	0.4	0.5
w_s	0.2	0.5	0.6
w_q	0.2	0.6	0.4
w_t	0.2	0.4	0.6
w_d	0.2	0.6	0.5
w_w	0.2	0.3	0.5

park in Singapore as a set of preferences P and a set of visited attractions R_{visit}. We randomly select 24 instances as training instances and the remaining 24 as testing instances. To ensure a unbiased result, we use attraction rank, service time and queue time, as of June 2012.

We compute the results using 2 strategies of AutoParTune, namely: strategy 1 and 3, on Table 5.10. We use F-Race as global tuning component. We present the average size of intersection and compare it with the result of the default parameter value used by our industry partner. We show the default parameter configurations and parameter configurations from AutoParTune in Table 5.9 The results in Table 5.10 shows that 3 strategies of AutoParTune are superior to the default. Strategy 3 outperforms strategy 1 in training and testing instances.

To test the effectiveness of our tuned weights in matching the patron and attraction preferences, we run additional experiments for different preference factors. For thrill preference, we generate the patron's preferences with six different thrill values (1, 0.8, 0.6, 0.4, 0.2 and 0) and the other preference values are fixed to the same value. For

Table 5.10: Theme Park Personalized Intelligent Route Guidance Algorithm Performance Result using Scenario 1

Technique	Training	Testing
Default	0.760	0.834
AutoParTune Strategy 1	0.773	0.917*
AutoParTune Strategy 3	**0.818***	**0.919***

* = statistically significant against Default Configuration.
Boldface indicates the best performance result.

Table 5.11: Routes from Default Configuration and AutoParTune Strategy 3

Default Configuration*	AutoParTune Strategy 3*
CYLON (1)	Magic Potion Spin (0.1)
HUMAN (0.8)	Enchanted Airways (0.2)
Treasure Hunters (0)	Dino-Soarin (0.2)
Canopy Flyer (0.3)	Canopy Flyer (0.3)
Enchanted Airways (0.2)	Jurassic Park Rapids Adventure (0.5)

* = Attractions (Attraction's Thrill Factor).

this experiment, we do not compare the result to the "ground truth". We run the target algorithm 100 times for each parameter configuration. We assume if the patron's thrill preference is decreasing, the occurrences of an attraction with the highest thrill factor should also decrease.

Table 5.11 shows an example of the routes generated using the default configuration and the configuration from AutoParTune Strategy 3. In this example, we set patron's thrill preference to 0.2. Notice that Cylon that has thrill factor of 1, should not be included in the route because the patron's thrill preference is low. In the route from default configuration, Cylon is still appearing while in the route from AutoParTune Strategy 3 is not.

We then calculate the occurrences of an attraction with the highest thrill factor ($thrill_i$=1) and present the result in Fig. 5.7(Thrill Response Effect). The result from Strategy 1 and 3 configurations follow the natural assumption better than the result from default configurations which shows a static value for almost all preference values. We run the same treatment for dark and wet preferences and show the result in

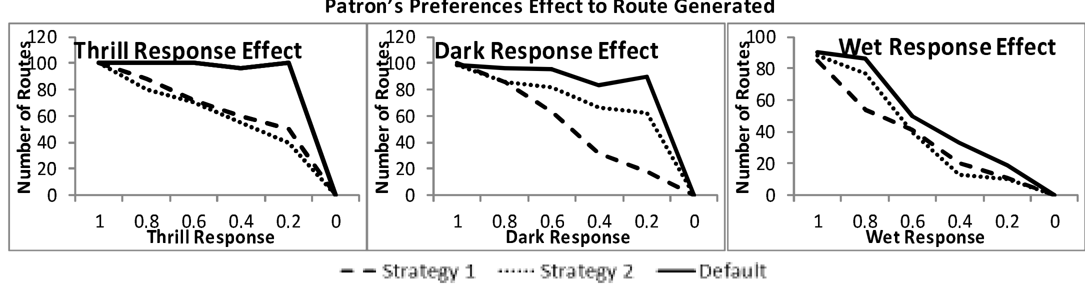


Figure 5.7: Effect of Patron Preferences on Route Generated from Personalized Intelligent Route Guidance Algorithm

Fig. 5.7. The obtained behavior is similar as for thrill preference.

Scenario 2: Qualitative Subset Factor Weights

Other than patron preference, queue time is another important factor in maximizing patron experience in the theme park. General survey results indicate a very high customer dissatisfaction with relation to long queue times [9]. Therefore apersonalized intelligent route guidance program should address this issue.

Taking the queue time into consideration, in this second tuning scenario, we set the tuning objective as tuning qualitative subset factor weights such that the route generated by the algorithm has the lowest overall queue time. We assume that the queue time for each attraction changes while attraction rank and service time are always fixed. Similar to the previous scenario, we use the tuning scenario as illustrated in Fig. 5.6 but for "Quality Calculation method" we simply calculate the overall queue time for the top 5 attractions and return it to the configurator. We only study the top ranked attractions because those attractions usually have a much higher queue time compared to the less popular attractions. We apply AutoParTune on this tuning scenario to tune w_r, w_s and w_q. For w_t, w_d and w_w, we use the best configuration from the previous scenario.

We use 50 generated preferences and randomly select 25 instances as training instances and the remaining 25 as testing instances. We compute the results using 2 strategies of AutoParTune, namely: strategy 1 and 3, on Table 5.12. We present the average of queue time and compare the result with the default parameter value used

Table 5.12: Theme Park Personalized Intelligent Route Guidance Algorithm Performance Result using Scenario 2

Technique	Training	Testing
Default	18.416	17.958
AutoParTune Strategy 1	16.416*	15.100
AutoParTune Strategy 3	**13.041***	**14.016**

* = statistically significant against Default Configuration.
Boldface indicates the best performance result.

by our industry partner. The result shows that solutions from AutoParTune strategies reduce the overall queue time by 2-5 minutes.

5.6 Discussion

In dealing with the complex optimization problem for industrial problems, we show that our approach provides better parameter configurations than the default manually tuned parameters. AutoParTune for non-local search based target algorithm (Theme Park Personalized Intelligent Route Guidance Problem) also shows a significant improvement compared to the default configuration. We claim that AutoParTune is sufficient for automatically tuning the parameters of a target meta-heuristic algorithm (local-search or non-local search based).

AutoParTune with preprocessing methods (strategy 2, 3, 4, 5 for Aircraft Spares Inventory Optimization Problem and strategy 2 for Theme Park Personalized Intelligent Route Guidance Problem) perform significantly better than AutoParTune with only global tuner (strategy 1). Based on this result, we verify that using the preprocessing method to guide the tuning process provides a better parameter configuration and significantly improves the overall performance.

Our experiments illustrate the practical impact of our proposed approach on tuning local search algorithms. As meta-heuristic algorithms are used designed for solving large complex optimization problems more than ever, our approach offers the ability to produce effective parameter settings automatically in a computationally efficient

manner, rather than relying on the tedious and mostly manual tuning.

5.7 Chapter Summary

AutoParTune, a web-based workbench for automated parameter tuning, is implemented to facilitate an easy and reliable tuning process for users. It combines two preprocessing processes with a global tuning component to provide a more effective and efficient automated tuning strategy. Two major challenges in implementing AutoParTune as a web-based workbench are security and integrity. We answer the security concerns by adding two security mechanisms: email authentication and antivirus scanning; whereas for integrity concerns, we develop the ”bridge” for each component to maintain the communication to each other. AutoParTune provides users with five tuning options.

We used AutoParTune on two industry problems and applied different AutoParTune strategies. The result shows encouraging superior performance as compared to the default parameter configuration used by our industry partner.

Chapter 6

Instance-Specific Tuning: Extension to Genetic Algorithms

In the previous chapters, we discussed two frameworks for instance-specific tuning, CluPaTra (Chapter 3), and CluPaTra-II (Chapter 4). These two frameworks use the local search trajectory as the generic feature for clustering. CluPaTra uses the pair-wise sequence alignment method to calculate similarity scores while CluPaTra-II models its feature extraction as a pattern mining problem and designs novel techniques to solve it. Both CluPaTra and CluPaTra-II show encouraging improvement when compared to one-size-fits-all and existing instance-specific configurators for three classical COPs: Travelling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP).

We also discuss CluPaTra and CluPaTra-II implementation on AutoParTune, a web-based workbench, in Chapter 5, which integrates CluPaTra and CluPaTra-II with Fact-RSM, a parameter search space reduction method, as well as ParamILS and F-Race as global tuning components. This workbench offers the user five combinations for performing tuning: (1) global tuning only; (2) parameter search space reduction and global tuning; (3) instance-specific tuning and global tuning; (4) instance-specific tuning, parameter search space reduction and global tuning; and (5) parameter search space reduction, instance-specific tuning and global tuning. We applied Au-

toParTune to tune two industrial study cases and presented significant improvements in the overall performance result compared to the result of a default configuration used by our industry partners.

Although CluPaTra and CluPaTra-II have shown promising results, there is still one apparent drawback due to their scope in local-search-based algorithms. As an attempt to extend these approaches to population-based algorithms, we investigate how to generate clusters from population-based algorithm using generic features pertaining to population dynamics. We propose in this chapter two unpublished preliminary ideas (PeTra and PaRG) for tuning a Genetic Algorithm (GA). PeTra is an extension of CluPaTra where we analyze similarity from GA's **P**opulation **E**volution **Tra**jectory and represent it as a directed sequence, whereas PaRG is an extension of CluPaTra-II - FloTra where we investigate GA's **Pa**rent Inheritance **R**elationship similarity in **G**raph representation.

We present the details of PeTra and PaRG. We then describe experimental results on tuning the Two Population Genetic Algorithm that is applied to solve the Generalized Assignment Problem (GAP). Finally, we conclude by summarizing the chapter.

6.1 PeTra: Population Evolution Trajectory Similarity

In PeTra, we focus on capturing population evolution movement from initial population to the next until it reaches its final population to analyze its evaluation leap. Evaluation leap has been used as a measurement for GA performance [103]. A generation is said to be an evaluation leap if the best solution produced at the generation is better than those in previous generations. We assume that similar instances will have similar evaluation leaps across their populations, such that clustering the instances based on its evaluation leaps will create a set of thigh clusters for the purpose of instance-specific tuning.

Following the work in CluPaTra, we transform the population evolution as a direct

sequence and use sequence alignment to calculate the similarity score for each pair of instances. We consider each population as a node and arrange it according to its generation order to form a directed path. Similar to search trajectory representation in CluPaTra (see chapter 3), each node in the directed sequence is a symbol based on two population properties: position type [55] and the percentage deviation of its quality from $Best$ (as defined in Definition 3). Unlike in search trajectory, where each node is a solution, in PeTra, we aggregate these position types and the percentage deviation of quality to represent a population *"snapshot"*.

For position type, we focus on capturing LOCAL MAXIMUM (or MINIMUM). Differing from CluPaTra, where we determine the position type based on the direct neighborhood solutions, a local maximum (or minimum) in PeTra is determined based on population topology by comparing each solution to others in the same population. We count the number of local maxima (or minima) in each population and normalize the value by scaling it between 0 and 1. We then categorize the value in three groups: HIGH (normalized value $\geq$ 0.7), MEDIUM (normalized value $\geq$ 0.4) and LOW (normalized value $<$ 0.4).

The percentage deviation of quality from each solution in a population is summarized with three values: minimum, maximum and average. The values are then compared with the $Best$ and categorized in three groups: HIGH (percentage deviation $\leq$ 5%), MEDIUM (percentage deviation $\leq$ 10%) and LOW (percentage deviation $>$ 10%).

These four properties (minimum, maximum, average, and local maximum) are combined and hashed into a unique symbol. The population representation process is illustrated in Fig. 6.1. Note that these population's properties are generic which can be easily retrieved or computed with little additional computation time from any Genetic Algorithm albeit for different problems.

After transforming the populations as a directed sequence, we follow the steps in CluPaTra framework. We calculate similarity using sequence alignment for each pair of population evolution trajectory sequences and cluster the instances using AGNES

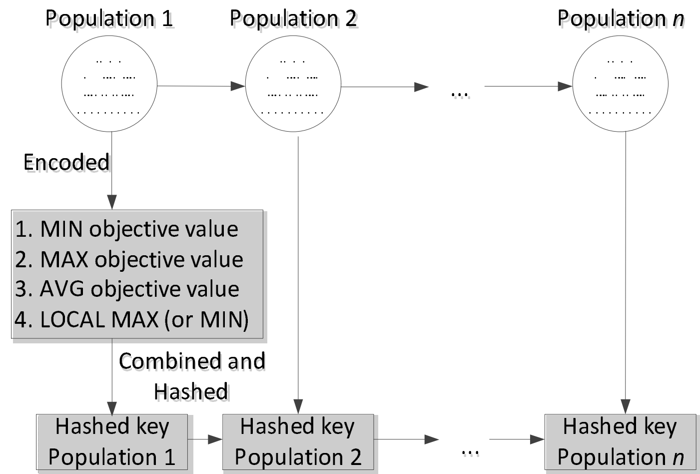


Figure 6.1: Genetic Algorithm Population Presentation.

with L method. We finally tune each cluster to find a good parameter configuration. For an arbitrary testing instance, we first map its population evolution trajectory to the closest cluster. The tuned parameter configuration for that cluster is then returned as the parameter configuration for this instance.

6.2 PaRG: Parent Inheritance Relationship similarity in Graph representation

In contrast with PeTra which investigates the population dynamic, PaRG focuses on GA's selection mechanism, an important operator in GA [82]. Selection mechanism chooses chromosomes from a population as parents using a certain selection criteria based on its fitness value. On average the better chromosomes are more likely to be selected than the poor ones. We explore the inheritance relationship between selected chromosomes (parents) and represent it as a graph. We extend the work on CluPaTra-II - FloTra (Chapter 4) by replacing the search trajectory graph with a parent inheritance relationship graph and running a pattern mining technique to retrieve a set of features. We calculate instance's similarity score using these features and implement AGNES with L method to cluster the instances based on its similarity score. Finally we tune the clusters using one-size-fits-all configurator. The steps of PaRG are shown in Fig. 6.2.

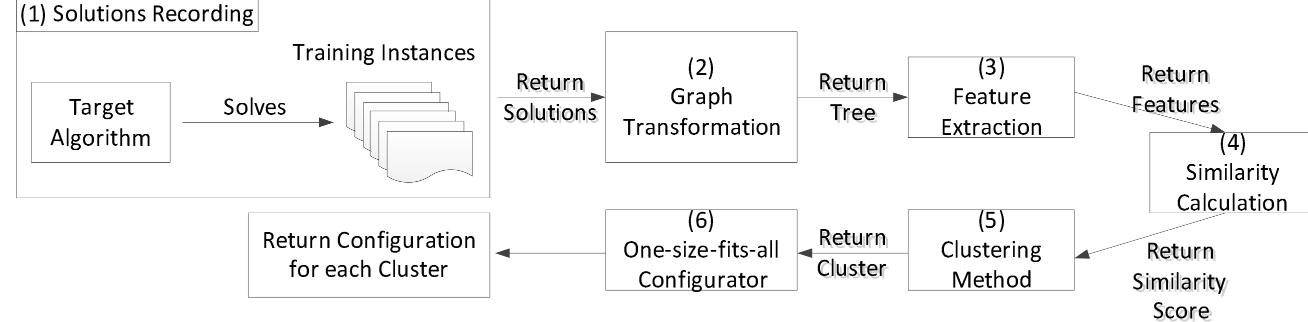


Figure 6.2: Steps in PaRG: **Pa**rent Inheritance **R**elationship similarity in **G**raph representation.

We follow the CluPaTra-II - FloTra framework and implement the same methods for step 4, 5 and 6. PaRG uses cosine similarity as the method for similarity calculation (step 4) and AGNES with L method as the clustering method (step 5). In comparison, for a one-size-fits-all configurator (step 6), PaRG uses existing approaches such as ParamILS, CALIBRA or F-Race. Details on step 2 (graph transformation) and 3 (feature extraction) are as follows.

6.2.1 Graph Transformation

The Parent Inheritance Relationship graph is defined as an inter-parent relationship where a node represents a parent chromosome and an edge represents the inheritance relationship between chromosomes. If a chromosome is a parent of another chromosome, we put an edge on the two chromosomes. A dense graph represents highly related parents where most of the parent chromosomes are the descendants of other parents in previous generations. This represents the existence of an elite group which consists of good solutions in the population. The elite group dominates other solutions in the selection mechanism and has a higher chance to carry over to subsequence generations. In contrast, a sparse graph behaves differently and it represents the case where most of the parent chromosomes do not have any relationship with each other, which indicates the non-existence of elite groups.

The rationale of our feature is predicted on the relationship between the elite population and GA's performance [34, 108]. Elitism, which is usually preserved using

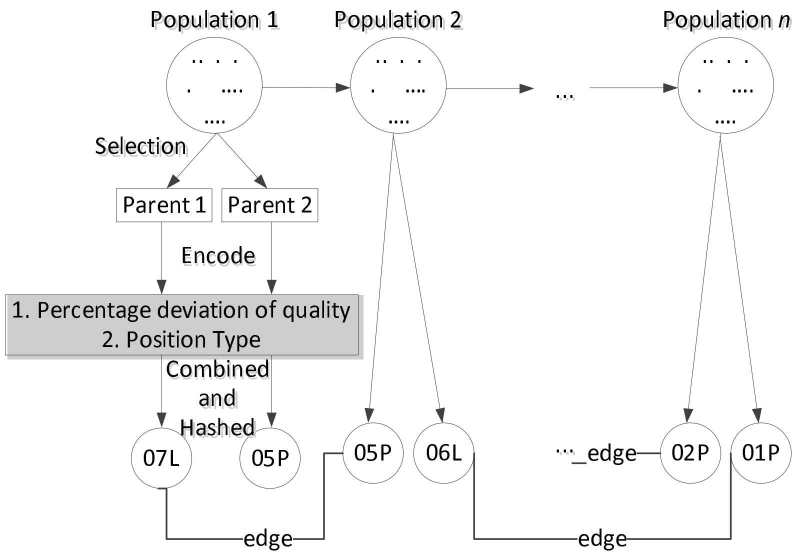


Figure 6.3: Parent Inheritance Relationship Graph Representation.

simple or complex elitism strategy, improves considerably the performance of GA in single or multi objective optimization problems [34]. Given a fixed GA, our conjecture is that similar instances will have similar elite groups under a fixed parameter setting; and that there exists a parameter setting that will yield good solutions in instances with a similar elite group.

We present the parent inheritance relationship graph as a undirected graph, where each node represents a parent chromosome and each edge represents a inheritance relationship between parents. As in a search trajectory graph, each node in the graph encodes a combination of two solution attributes: position type and the percentage deviation of its quality. In PaRG, we determine position type by evaluating the solution objective value with other solutions' objective values in the same population - whether it is better, worse or equal. The 7 positions types are shown in Table 3.2 (Chapter 3). The deviation of solution quality is calculated by comparing the solution's objective value with $Best$ (as defined in Definition 3). Position type and percentage deviation of quality are then combined and hashed into a symbol. The graph representation of a Parent Inheritance Relationship Graph is shown in Fig. 6.3.

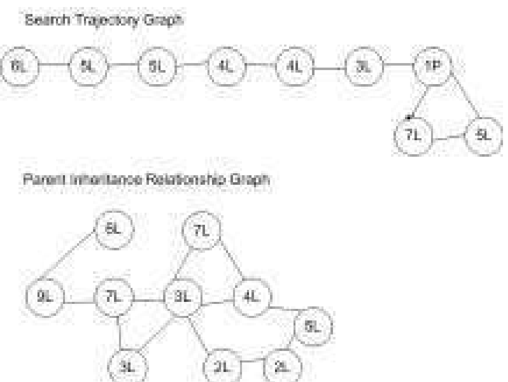


Figure 6.4: Difference between the Search Trajectory Graph and Parent Inheritance Relationship Graph.

6.2.2 Feature Extraction

After obtaining parent inheritance relationship graphs for each instance, we extract meaningful features using pattern mining techniques. The graph structure for the search trajectory graph and parent inheritance relationship graph are different, as illustrated in Fig. 6.4. The search trajectory graph has a long stem and several thorns and petals, while the parent inheritance relationship graph has a more complete graph.

Since the graph structures for the search trajectory graph and parent inheritance relationship graph are not the same, we cannot use similar pattern mining technique as in FloTra (Chapter 4) for PaRG. We turn to a well-established pattern mining technique for generic graph, gSpan [117].

gSpan (or graph-based substructure pattern mining) is a frequent pattern mining technique that uses theDepth First Search (DFS) algorithm to generate its subgraphs for mining in a large graph database. It also introduces a new lexicographic ordering system which is generated based on the DFS algorithm for efficient graph isomorphism tests. gSpan removes two most-time-and-memory-consuming tasks: candidate generations and false positive pruning. In candidate generations, a pattern mining algorithm creates size k frequent subgraphs and increases the size gradually by adding one node for each iteration. The generation of size $(k + 1)$ subgraph candidates from size k frequent subgraphs is complex and costly. On the other hand, false positive pruning is used to prune isomorphic candidates using a subgraph isomorphism test which is also very costly. gSpan replaces these two tasks by combining the growing

and checking of frequent subgraphs into one procedure, thus accelerating the mining process.

gSpan works as follows. It first creates a set of minimum spanning trees (dfs-tree) from the graphs using DFS which defines an order in which the edges are visited. gSpan then construct a canonical representation in that order, called graphs dfs-code. A growing technique is restricted by gSpan in two ways: first, a subgraph can only be extended at nodes that lie on the rightmost path of the dfs-tree; and secondly, subgraph generation is guided by occurrences in the appearance lists. gSpan computes the canonical (lexicographically smallest) dfs-code for each growing step in a series of permutations. The growing process stops either when the support of a subgraph is less than $minSup$ or its dfs-code is not a minimum code, which means this subgraph and all its descendants have been generated and discovered previously. For $minSup$ value, we need to set it beforehand.

6.3 Empirical Experiment Result

We perform the tuning of the Two Populations Genetic Algorithm for solving the Generalized Assignment Problem (GAP). We measure the performance as the average of the percentage deviation from the best found solution $Best$ (Definition 3) and compare our experiment results with default configurations and configurations from a one-size-fits-all configurator. The details of the experiment are as follows.

6.3.1 Target Problem and Algorithm

The Generalized Assignment Problem (GAP) is a widely-studied COP with many practical applications [66]. The GAP may be defined as follows. Given m agents (or processors) and n tasks (or jobs), the GAP aims at finding the maximum-profit assignment of each task to exactly one agent, subject to the capacity of each agent. It can be formulated as follows. Let $I = 1, ..., m$ be the set of agents and $J = 1, ..., n$ the set of jobs. A standard integer programming formulation for GAP is given in

expression (6.1) where p_{ij} is the profit from assigning job j to agent i, a_{ij} the resource required for processing job j by agent i, and b_i is the capacity of agent i. The decision variables x_{ij} are set to 1 if job j is assigned to agent i, and 0 otherwise. The constraints, including the integrality condition on the variables, state that each job is assigned to exactly one agent, and that the resource availabilities of agents do not exceed [66, 80].

$$\max z = \sum_{i \in I} \sum_{j \in J} p_{ij} x_{ij} \text{ s.t.} \sum_{i \in I} x_{ij} = 1 \quad \forall j \in J, \sum_{j \in J} a_{ij} x_{ij} \leq b_i \quad \forall_i \in I, x_{ij} \in \{0, 1\} \tag{6.1}$$

The GAP is known to be an NP-hard problem. Exact approaches to solve GAP include branch-and-price [96], and branch-and-bound [86] while heuristic approaches include tabu search [35], and path relinking with ejection chains [116]. Several Genetic Algorithms have been proposed to the GAP [81], from a GA with a problem specific heuristic operator involving two local improvement steps after the regular crossover [31] to a Guided GA that uses an extra weighting operation to identify which genes in a chromosome are more susceptible to being changed during crossover and mutation [72].

To solve the GAP problem, we construct the Two-Population Genetic Algorithm (FI2PopGA) [67]. In FI2PopGA, the population is divided into feasible and infeasible populations. The feasible population is a group of solutions that do not violate any constraints while the infeasible population is a group of solutions that violate at least one constraint. This approach arises from an intuitive idea that if one can separate the measuring of performance and feasibility, there may be a better chance to find optimal solutions that are located at the boundary from both the feasible and infeasible directions. FI2PopGA is an interesting target algorithm to tune because it has similar parameters as in any well-known GA algorithm with one categorical parameter (i.e.: FitnessMethod) which gives FI2PopGA an option to explore different calculation methods.

In FI2PopGA, feasible and infeasible populations are treated separately and dif-

Procedure for Two Populations Genetic Algorithm
Inputs: i: instance;
Method:
1: Initialize population for chromosomes (t)
2: Evaluate feasibility for each individual, separate it
into two groups (feasible and infeasible) (t)
3: Evaluate each chromosome in each population using fitness function (t)
4: Repeat until a stopping criteria is satisfied
4.1: Select parents from population depending on selection
and reproduction criteria (t+1)
4.2: Perform crossover on parents and create new offsprings (t+1)
4.3: Perform mutation on new population (t+1)
4.4: Evaluate feasibility for each chromosome and saperate it into
two groups (feasibile and infeasibile)(t+1)
4.5: Evaluate each chromosome using fitness function (t+1)
Output: s: solution;

Figure 6.5: Two Populations Genetic Algorithm Procedure

ferently. The fitness function for the feasible population is the value of their objective function; while for the infeasible population, the fitness function can be calculated from their distance to the boundary of the feasible region or from their penalty function. In the selection stage, solutions are compared only with other solutions in its own population. The selected chromosome mates with another chromosome in the same population to generate offsprings. FI2PopGA is outlined in Fig. 6.5.

The genetic operators we implement are standard: single-point crossover, uniform random mutation and tournament 2 selection. The single-point crossover uses one point to exchange part of the solution string from two parents. The uniform random mutation changes every number in the solution string with a given probability. The tournament 2 selection chooses two parents randomly and compares their fitness score, and the one with the higher fitness score gets to mate. The parameters to be tuned are described in Table. 6.1.

We apply our target algorithm to 100 generated instances and randomly choose 50 for training and the remaining for testing. The number of jobs and agents is set to 100. For best known values, we use the best found solution.

Table 6.1: Parameters for Two Population Genetic Algorithm on Generalized Assignment Problem

Parameter	Description	Range
numGeneration	number of generation	[100, 1000]
populationSize	population size	[100, 1000]
FitnessMethod	fitness calculation method	[0, 1]
MutationRate	probability to run mutation for new offsprings	[0, 1]
CrossOverPoint	probability of cross over point	[0, 100]
CandidateNum	number of generated candidate parents for tournament selection	[1, 10]

6.3.2 Experiment Setting and Setup

As in CluPaTra-II - FloTra, we use ParamILS [60] as our one-size-fits-all configurator. We discretize the continuous parameters to 20 possible values by simple enumeration from minimum to maximum value. All experiments are performed on a 3.30GHz Intel Core machine running Windows 7. Cutoff times are set to 500 seconds per run and each configuration process is allowed to call the target algorithm for a maximum of 100 x n times, where n is the number of instances. To compare the significance of our results, we perform a t-test [83]; and we consider p-values below 0.05 are taken as statistically significant ($\alpha \leq 5\%$).

6.3.3 Performance Comparison

We compare the target algorithm performance using parameter configurations from PeTra and PaRG. We measure the performance using performance metrics as defined in Definition 3. In Table. 6.2, we show the performance comparison results. Notice that our approach outperforms the default configuration and configurations from the one-size-fits-all configurator, ParamILS.

6.4 Discussion

The result of the experiments on the Two Population Genetic Algorithm for the Generalized Assignment Problem (GAP) verifies the performance of PeTra and PaRG. It shows an encouraging improvement in performance compared to that of the default

Table 6.2: PeTra and PaRG Performance Result

Technique	Training	Testing
Default	0.45	0.36
ParamILS	0.25*	0.14*
PeTra	0.13*	0.09*
PaRG	**0.10***	**0.08***

* = statistically significant against Default parameter configuration.
Boldface indicates the best performance result.

configuration and configurations from one-size-fits-all configurator, ParamILS. Based on this preliminary result, we verify that PeTra and PaRG are viable extensions of our instance-specific tuning approaches for population-based algorithms.

On the performance result, we notice that PaRG outperforms PeTra. This may indicate that the parent inheritance relationship graph describes GA's characteristics to be better than the population evolution trajectory. This may be caused by the population evolution trajectory oversimplifying the population dynamic due to its aggregation mechanism that replaces individual solution properties with its population summary statistics (minimum, maximum, average and local maximum). It will be of interest to implement other population properties to improve the performance of PeTra.

Based on these results, we claim that: (1) PeTra and PaRG are suitable extensions of instance-specific tuning for population-based algorithms; and (2) PaRG which uses the parent inheritance relationship graph is more superior to PeTra that uses population evolution trajectory.

6.5 Chapter Summary

In this chapter, we introduce ideas for extending instance-specific tuning to population-based algorithm. We study the interaction and population dynamic in Genetic Algorithm and propose two extensions: PeTra and PaRG. PeTra focuses on population evolution trajectories and extends the CluPaTra framework. On the other hand, PaRG explores the selection mechanism dynamic and constructs a parent in-

heritance relationship graph to represent it. PaRG extends the work in CluPaTra-II - FloTra and uses gSpan to extract compact features from the parent inheritance relationship graph.

We applied PeTra and PaRG in tuning the Two Population Genetic Algorithm that has 6 parameters. The result shows encouraging improvement from the default parameter configuration and vanilla global configurator, ParamILS.

Chapter 7

Conclusions

In the previous chapters, we have discussed our generic automated parameter tuning methodology and shown experimentally its significant improvement over the existing approaches. In this last chapter, we provide a summary of the main contributions of this thesis, and provide a few pointers for future directions.

7.1 Contributions

Although there has recently been keen research interest in automated parameter tuning, to date, there is no single approach that is clearly generic that provides instance-specific parameter configuration. One-size-fits-all approaches are generic and may be applied to tune various application in various COPs, but only provide one best parameter configuration for the entire set of problem instances. Instance-specific approaches on the other hand, tend to use problem-specific features that make the approaches less general. Thus, our major contributions are summarized as follows:

CluPaTra: Instance-specific Automated Parameter Tuning via Trajectory Clustering (Chapter 3).

- We have constructed a **generic** instance specific automated parameter tuning framework by first performing a clustering of problem instances, and tuning the target algorithms to derive the best parameter configurations for the respective

clusters. Subsequently, given an arbitrary instance, we map it to the closest cluster. The tuned parameter configuration for that cluster is returned as the parameter configuration for this instance.

- We have introduced the notion of an instances search trajectory as the problem-independent feature. Search trajectory is defined as the path that a local search algorithm follows as it searches from an initial solution to its neighbor from one iteration to the next. The advantage of our approach lies in the fact that the search trajectory may be computed from a local-search based algorithm. Hence our feature is problem-independent and may be conceptually retrieved from any local search-based algorithm.

- We have constructed a novel technique to extract problem-independent features and calculate similarity based on them using a well-known machine learning technique: sequence alignment. We have explored two different search trajectory sequence representations and two sequence alignment implementations.

- We applied CluPaTra on three classical COPs: Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Set Covering Problem (SCP) and showed significant improvement toward existing one-size-fits-all configurators.

CluPaTra-II: Pattern Mining Approaches for Instance-specific Automated Parameter Tuning (Chapter 4).

- We have overcome CluPaTra's limitations in scalability, flexibility and descriptiveness by constructing CluPaTra-II where we add a feature extraction step and replace the similarity calculation with a well-known method, cosine similarity.

- We have modeled feature extraction as a pattern mining problem and have designed two new data mining techniques to boost CluPaTra computational speed as well as to improve the cluster quality and the overall performance.

- We have presented SufTra, a pattern mining technique to extract patterns from sequence search trajectories based on Suffix Tree data structure. SufTra offers a linear time algorithm to exact meaningful features.

- We have designed FloTra, a graph mining technique for search trajectory graphs. FloTra offers a fast technique to extract compact features using specific characteristics of search trajectory graphs.

- We have applied CluPaTra-II on three classical COPs as in CluPaTra and showed improved results in terms of computation time, cluster quality and overall performance as compared to CluPaTra.

AutoParTune: Web-based Automated Parameter Tuning Workbench (Chapter 5).

- We have implemented our approaches for instance-specific automated parameter tuning in a web-based automated parameter tuning workbench that integrates our approaches with a method for parameter-space reduction and global (one-size-fits-all) parameter tuning.

- We have applied two basic security mechanisms to protect AutoParTune against Internet attacks from human and automated-agent perpetrators. We implement email Authentication Mechanism to prevent automated-agent perpetrators and anti-virus Scanning Mechanism to check files uploaded for malicious codes.

- We have designed component communication schema to enable communications between each of the components in AutoParTune.

- We have applied AutoParTune in two industrial cases: the Aircraft Spares Inventory Optimization Problem and the Theme Park Personalized Intelligent Route Guidance Problem and produced better overall performance results compared to the default configuration used by our industry partners.

Instance-specific Tuning: Extension to Genetic Algorithms (Chapter 6).

- We have extended our approaches for instance-specific automated parameter tuning to population-based algorithms. We analyze Genetic Algorithm population's dynamic and design two new approaches: PeTra and PaRG.
- We have presented PeTra, an extension of CluPaTra for Genetic Algorithm. PeTra is design to capture similarity from GA's Population Evolution Trajectory by representing it as a directed sequence and calculate the similarity using pairwise sequence alignment.
- We have introduced PaRG, an extension of CluPaTra-II - FloTra where we investigate GA's Parent Inheritance Relationship similarity in Graph representative. We implement a well-known pattern mining technique, gSpan, as the feature extraction method.
- We have implemented PeTra and PaRG on Generalized Assignment Problem (GAP) using Two-Population Genetic Algorithm and produced encouraging results compared to the default configuration and vanilla one-size-fits-all configurator.

7.2 Future Directions

There are a number of future directions that can be pursued to extend our work further, and these are summarized as follows.

First, we discuss AutoParTune's scalability. AutoParTune is designed as a web-based application that enables users to perform their tuning computation in the server. As all tuning processes, which are computationally time consuming, are done in the server, AutoParTune is not scalable for tuning large instances (which may require the target algorithm to run for a long time). Furthermore, the ability to handle multiple tuning processes concurrently poses a challenge in scalability for AutoParTune as well.

To overcome the challenge of scalability, two approaches can be considered, namely, process batch and peer-to-peer computing. First, AutoParTune may run the tuning process in batches, i.e. users may upload tuning tasks anytime, but AutoParTune will process them in batches periodically. A queuing system will drastically bring down computational load.

Second, AutoParTune may adopt a peer-to-peer (P2P) approach to distribute tuning tasks between the AutoParTune server and the user machine. The idea of P2P is to allow users to share resources, such as power, knowledge, disk storage and information, between computers [76]. P2P has been used mostly in large scale data and information sharing. Examples of well-known P2P applications are Napster and Oxford anti-cancer projects [76]. In P2P design, computers can act as both clients and servers, with the roles determined according to the requirements of the system at any particular given time. Using P2P techniques, one may distribute the balance workload between the AutoParTune server and user machine for certain tuning processes such as calling the target algorithm for different instances using different parameter configurations.

A second future research direction can be conceived to explore techniques for feature extraction in CluPaTra-II. In this thesis, we model instance-specific parameter tuning as a frequent pattern mining problem and construct a sequential pattern mining and a structural pattern mining algorithm to extract features from search trajectories. Other than sequential and structural pattern mining, search trajectory similarity may be computed using other methods such as time-series pattern mining [118, 75].

In time-series pattern mining, one may consider the search trajectory as time series data and represent each solution in the search trajectory as one data point. This technique is natural because the search trajectory has natural temporal ordering, where each solution is found in a different algorithm step (or iteration), which can also be constituted as a different time series. We then extract the features using a time-series pattern mining technique such as [97]. In [97], the time series data are clustered by constructing a spectra from the original time series data with the means adjusted to

zero and normalizing it by the differences with the largest peak (in terms of a search trajectory, we can associate the peak with a local optimal solution). They then apply a hierarchical clustering method to cluster the spectra.

One limitation in modeling instance-specific parameter tuning as a time series mining problem is that it can only work for sequence search trajectories because the time series data are assumed as a sequence of data points [5]. It may not be suitable for clustering search trajectories that have many cycles.

Other then time series pattern mining, search trajectory similarity may be computed using other data mining techniques such as correlation mining [51] and associative classification [51].

Third, in this thesis, we only use one single generic feature (search trajectory) to calculate similarity and cluster the instances. It will be of interest to investigate how different possible features (generic or problem-specific) such as Fitness Distance Correlation (FDC) and problem size, can be incorporated to improve the performance of the clustering and the overall tuning result. However, adding different features will increase the dimensionality of the data and make the clustering process more challenging [24].

The common approach to deal with high dimensional data is to transform it into lower dimensional data via Principal Component Analysis (PCA) [36]. PCA reduces high dimensional data into a few dimensions regardless of the nature of the original variables (i.e. ordinal, continues, categorical) [63]. Each dimension is called a Principal Component (PC) and represents a linear combination of the variables. The first PC accounts for as much variation in the data as possible. Each succeeding PC accounts for as much of the variation unaccounted for by preceding PCs as possible. PCs are orthogonal and guaranteed to be perfectly independent of each other. PCs are found by calculating the eigenvectors and eigenvalues of variable data. The eigenvector with the largest eigenvalue is the direction of greatest variation, the one with the second largest eigenvalue is the (orthogonal) direction with the next highest variation and so on. In our clustering context for instance-specific parameter tuning using different features,

we can apply PCA to reduce the dimensionality of the data set prior to clustering. The objective of using PCA prior to clustering is for the PC to extract the cluster structure in the data set as in [119] where they use the first two PCs to cluster the data using a variant of the hierarchical clustering.

Appendix A

Empirical Experiment Result

Table A.1: Performance Comparison on TSP

Approach	#Fold	Training	Testing
ParamILS	1	2.56	2.01
	2	2.86	2.11
	3	2.76	1.92
	4	2.55	1.93
	5	2.62	2.13
CluPaTra Standard	1	2.33	2.04
	2	2.18	2.01
	3	2.05	1.93
	4	2.14	1.91
	5	2.41	1.78
CluPaTra Trans	1	1.97	1.85
	2	2.18	1.77
	3	2.16	1.23
	4	1.97	2.03
	5	1.78	1.71
CluPaTra Robust	1	2.08	2.03
	2	2.11	1.71
	3	2.32	1.91
	4	2.11	1.58
	5	1.87	1.81
CluPaTra Trans-Robust	1	2.11	2.02
	2	1.98	2.09
	3	1.76	1.87
	4	2.45	1.67
	5	1.99	1.99
ISAC	1	2.23	2.09
	2	2.13	1.55
	3	1.95	1.23
	4	2.56	1.98
	5	1.23	2.53
CluPaTra-II - SufTra	1	1.98	1.87
	2	2.05	1.76
	3	2.13	1.31
	4	1.78	1.65
	5	2.05	1.25
CluPaTra-II - FloTra	1	1.87	1.23
	2	2.34	1.45
	3	2.01	1.42
	4	1.96	1.03
	5	1.74	1.13
CluPaTra-II - gSpan	1	1.87	1.45
	2	1.96	1.65
	3	2.21	1.04
	4	1.92	1.06
	5	1.98	1.24

Table A.2: Performance Comparison on QAP

Approach	#Fold	Training	Testing
ParamILS	1	2.03	2.48
	2	2.43	2.33
	3	2.55	2.01
	4	2.01	2.43
	5	2.05	2.12
CluPaTra Standard	1	1.92	2.13
	2	1.99	2.12
	3	1.98	2.17
	4	2.02	2.21
	5	2.04	2.32
CluPaTra Trans	1	1.87	2.13
	2	1.85	2.11
	3	1.96	2.04
	4	1.88	2.09
	5	1.84	2.04
CluPaTra Robust	1	1.89	2.14
	2	1.86	2.12
	3	2.01	2.11
	4	1.87	2.09
	5	1.82	2.06
CluPaTra Trans-Robust	1	1.91	2.19
	2	1.88	2.32
	3	1.97	1.99
	4	1.89	2.05
	5	1.87	2.39
ISAC	1	1.99	2.03
	2	2.1	2.54
	3	1.87	2.05
	4	2.04	2.01
	5	1.91	2.12
CluPaTra-II - SufTra	1	0.65	1.14
	2	1.02	1.43
	3	0.77	0.98
	4	0.82	1.23
	5	0.87	1.01
CluPaTra-II - FloTra	1	1.05	0.97
	2	0.65	1.15
	3	0.77	0.81
	4	0.59	1.19
	5	0.84	1.21
CluPaTra-II - gSpan	1	0.67	1.23
	2	0.87	0.93
	3	0.64	1.09
	4	1.09	1.04
	5	0.71	1.16

Table A.3: Performance Comparison on SCP

Approach	#Fold	Training	Testing
ParamILS	1	1.72	0.98
	2	1.67	0.88
	3	1.16	0.71
	4	1.54	0.65
	5	1.57	0.89
CluPaTra Standard	1	1.31	0.76
	2	1.22	0.56
	3	1.01	0.82
	4	1.23	0.93
	5	1.43	0.98
CluPaTra Trans	1	1.09	0.76
	2	0.77	0.77
	3	0.56	0.69
	4	0.91	0.81
	5	0.56	0.98
CluPaTra Robust	1	1.01	0.95
	2	1.04	1.18
	3	1.19	0.99
	4	0.91	0.89
	5	0.91	0.87
CluPaTra Trans-Robust	1	0.54	0.75
	2	0.55	0.88
	3	0.81	0.64
	4	0.81	0.76
	5	0.65	0.87
ISAC	1	1.18	0.55
	2	1.34	0.76
	3	1.22	0.55
	4	1.02	0.98
	5	0.91	1.02
CluPaTra-II - SufTra	1	0.54	0.88
	2	0.34	0.75
	3	0.33	0.65
	4	0.23	0.78
	5	0.31	0.81
CluPaTra-II - FloTra	1	0.23	0.45
	2	0.12	0.65
	3	0.56	0.23
	4	0.12	0.78
	5	0.34	0.48
CluPaTra-II - gSpan	1	0.36	0.77
	2	0.35	0.62
	3	0.23	0.78
	4	0.34	0.46
	5	0.28	0.79

Appendix B

Quick Start Guide for AutoParTune

Quick Start Guide for AutoParTune

1. Introduction

AutoParTune (Automated Parameter Tuning Framework) is a framework for generic automated parameter tuning for a given target algorithm (such as Tabu Search, Simulated Annealing, GRASP). This framework is consisted of several parts, namely: parameter space reduction, feature-based instances classification, and parameter tuning. The framework is outlined in the picture below.

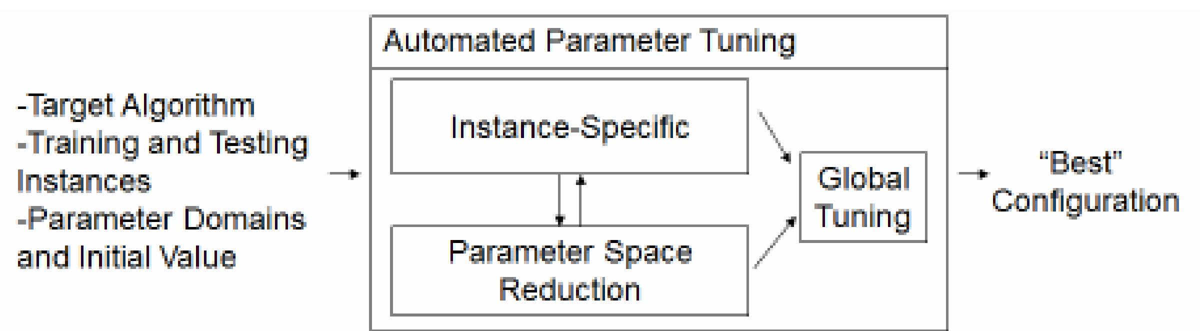


Figure 1. AutoParTune Framework

2. Input Files for AutoParTune

The user should provide:

- the target algorithm, which is compiled into Windows executable exe callable from the DOS command line.

  The target algorithm must be able to execute as follows:

  algo-executable –I instance_file –S seed params

  where:

  if target algorithm is a Stochastic Local Search then the code for the target algorithm need to be amended such that seed provides the value for the random seed used within the algorithm; else ignore the seed.

  params refers to the parameter values set for running the target algorithm

Example:

ils_tsp.exe –I kroa100.txt –S 2345 –P 10 –B 1

The output of the target algorithm is the best found objective value (displayed in the last line of the screen output when running the algorithm).

- a set of training and testing instances – one file for each instance
- a list of training and testing instances file.

  First line is a number of training or testing instances; and the rest are instance "file name" [tab] "best known value".

  Example:

  56

  a280.tspx 2579

  ch130.tspx 6110

- a txt file containing the parameter space.

  The parameter space file format is one parameter per line. Each line contains: parameter name, switch to pass the parameter to the algorithm, type of parameter (i=integer, r=real and c=categorical), minimum and maximum value for integer and real parameter or all possible parameter values for categorical parameter.

  Example:

PERTURBATION_STRENGTH	"-P "	r	[0.1, 10]
BETTER_ACCEPTANCE_CRITERIA	"-B "	i	[0, 1]
NON_IMPROVING_MOVES_TOLERANCE	"-N "	c	[1, 2, 3, 4]
OptChoose	"-O "	c	[3, 4]

- For instance-specific tuning (CluPaTra and SufTra), please provide trajectory generator target algorithm, which is compiled into Windows executable exe callable from the DOS command line (similar to "target algorithm").

  The trajectory generator should produce a [instance file name].result2.RunLog (example ch150.tspx.result1.RunLog) containing the instance's search trajectory obtained. The format of the file is:

Row	Field Name	Example
1	Instance's name	a280
2	Global optimal or best-found objective value	2579
3	Restart symbol	-
4*	Neighbor position, whether the solution has direct neighbor that has same, better or worse objective value. It is represented as 3 binary digits with 1 (yes) and 0 (no). - first digit for same objective value, second digit for better objective value and third digit for worse objective value.	1 0 1
5*	Objective value of the solution found	3334
6*	Solution found (sequence of nodes on the tour)	201,202,203,116,117,61,62,63,57,56,55,44,45,46,53,54,…
Last row	BF OV=best found objective value	BF OV=2911

*Rows 4-6 are repeated for each solution found by the target algorithm. Collectively, it represents the search trajectory.

3. Running AutoParTune

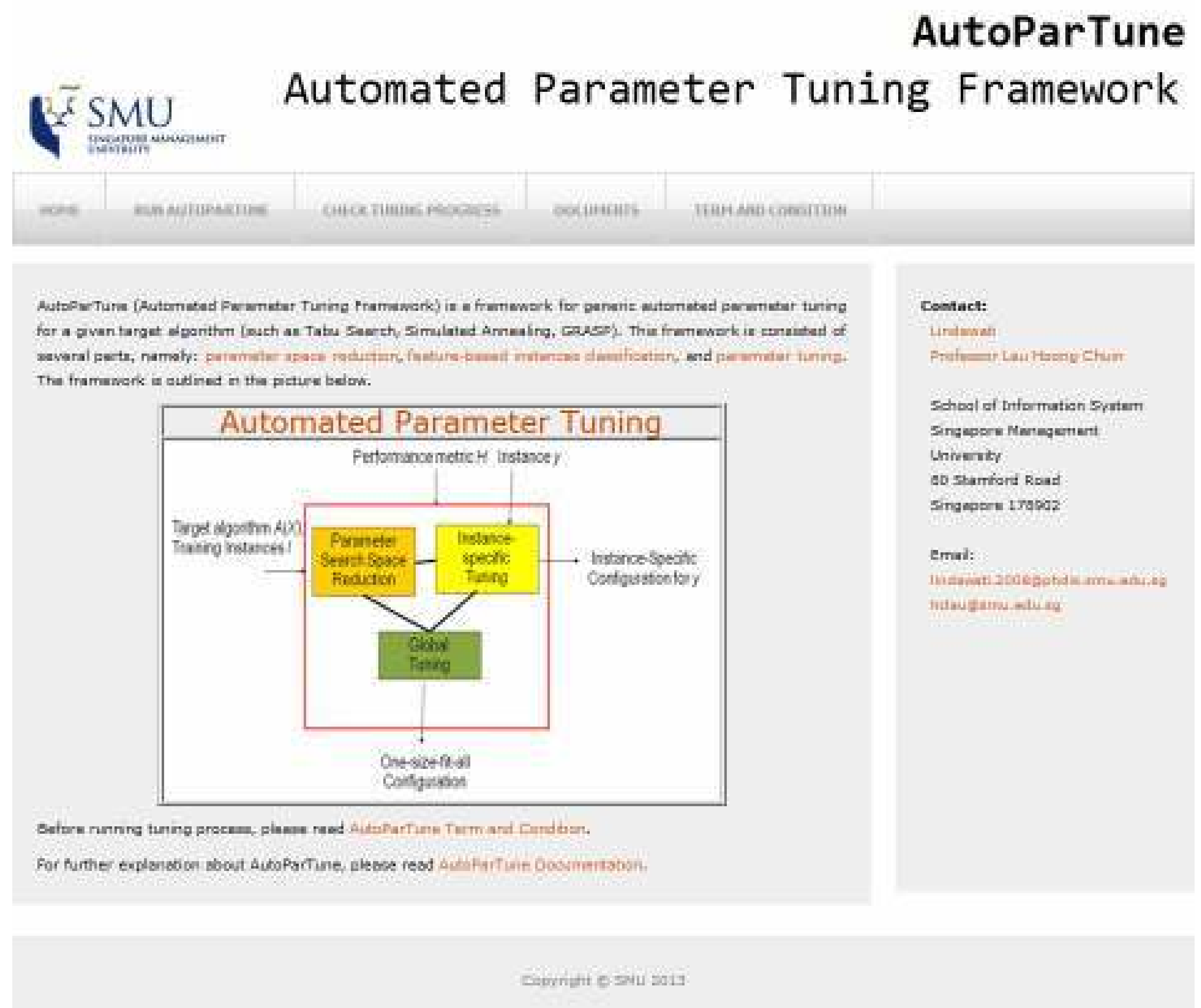


Figure 2. AutoParTune Home

Click “Run AutoParTune” to run parameter configuration.

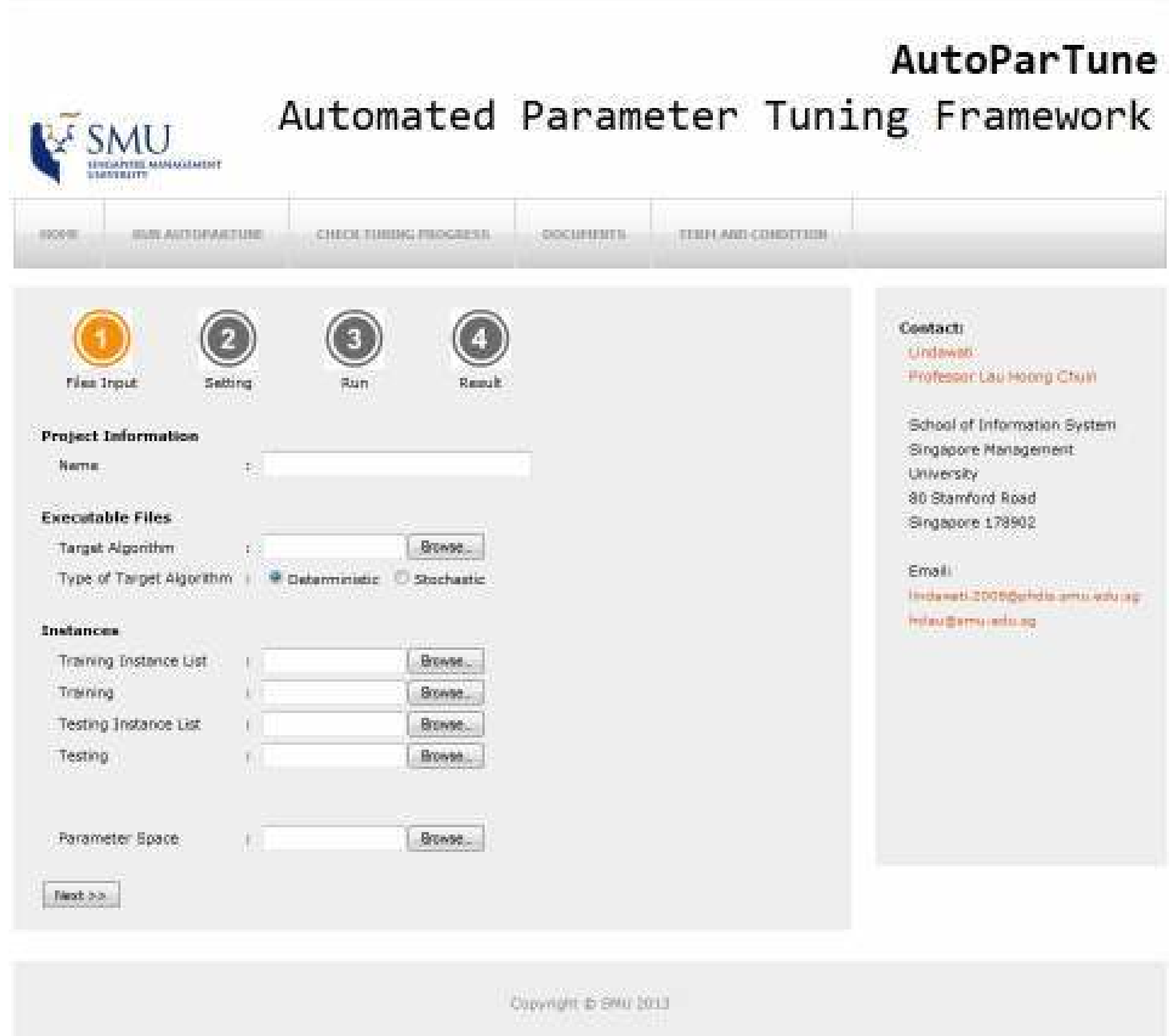


Figure 3. Run AutoParTune – File Input

Please input name of project, target algorithm (windows executable file), type of target algorithm, training instance list (txt file), training instance files (zip file), testing instance list (txt file), testing instance files (zip file), and parameter space (txt file).

Click “Next” to continue.

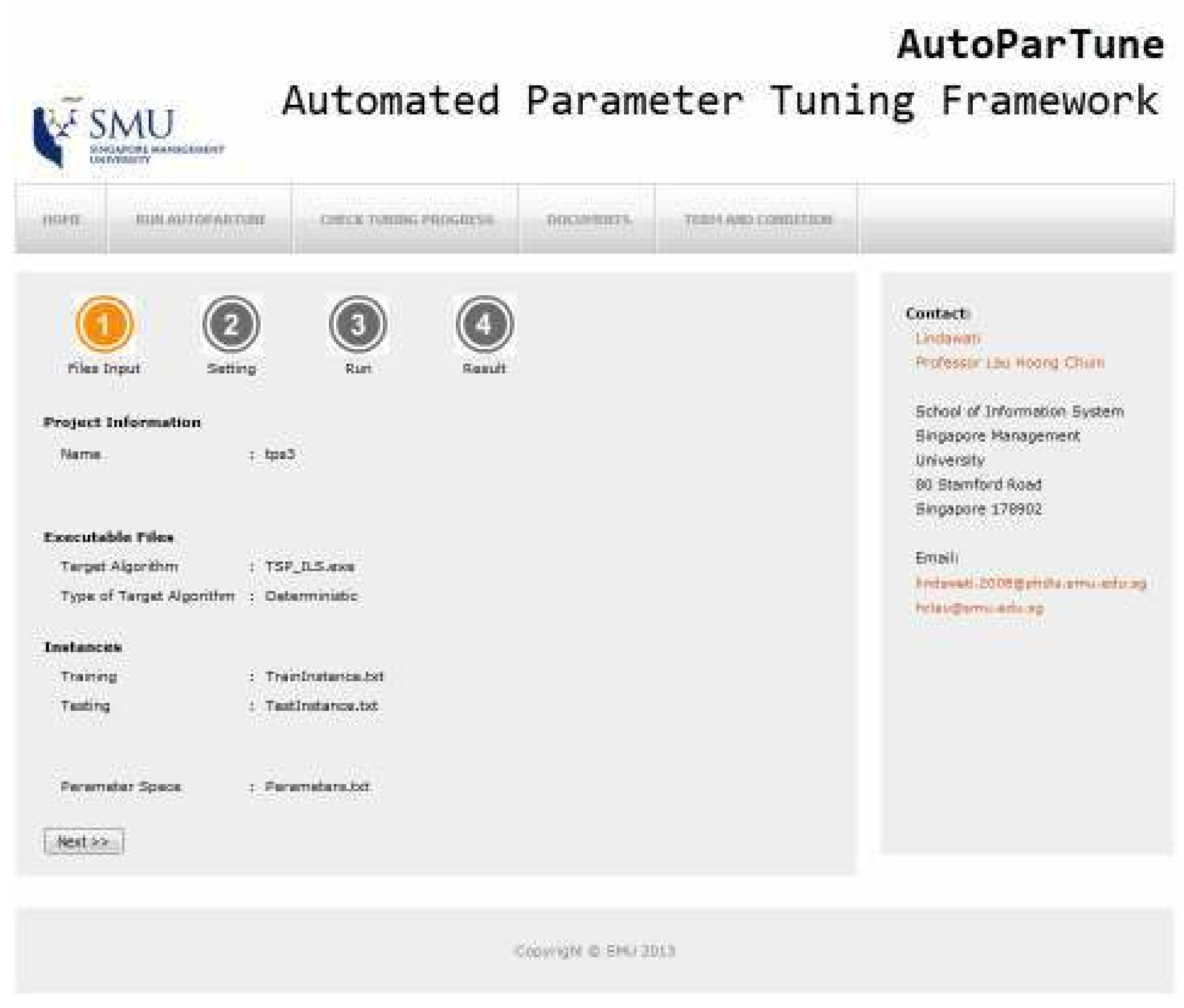


Figure 4. Run AutoParTune – File Input Confirmation

Click “Next” to continue.

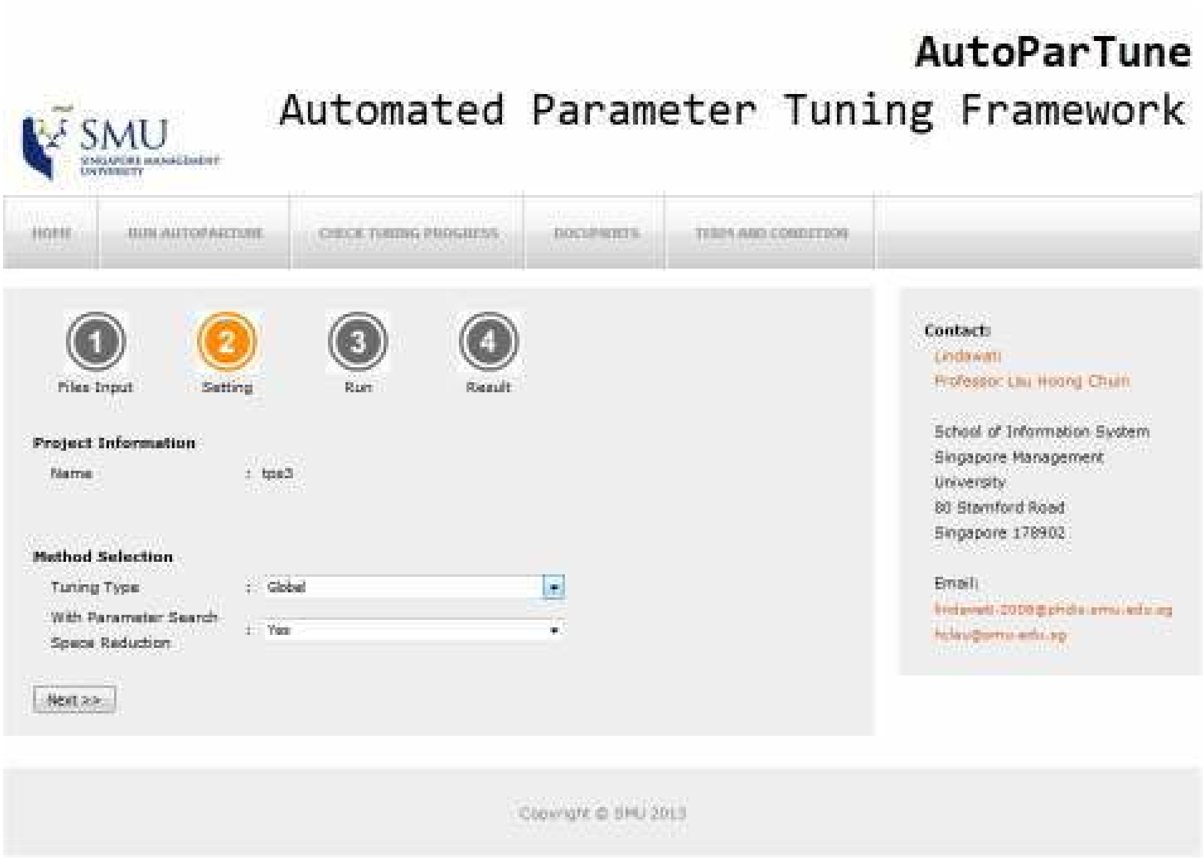


Figure 5. Run AutoParTune – Tuning Method

Choose the tuning type and whether the tuning with or without Parameter Search Space Reduction.

Click “Next” to continue.

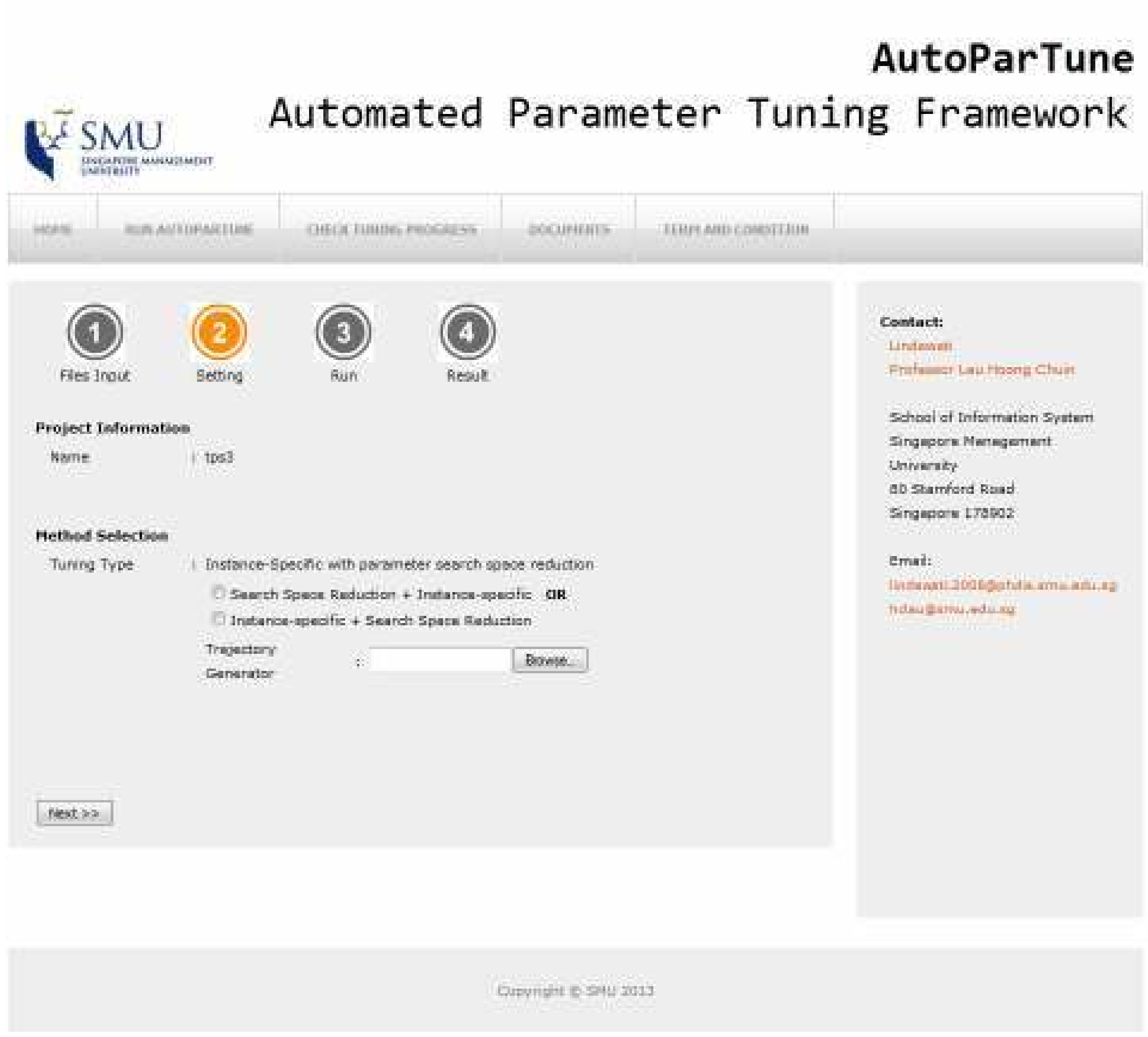


Figure 6. Run AutoParTune – Tuning Instance-specific Method

If Tuning Type is Instance-Specific, choose the instance-specific method, input trajectory generator (windows executable file).

Click “Next” to continue.

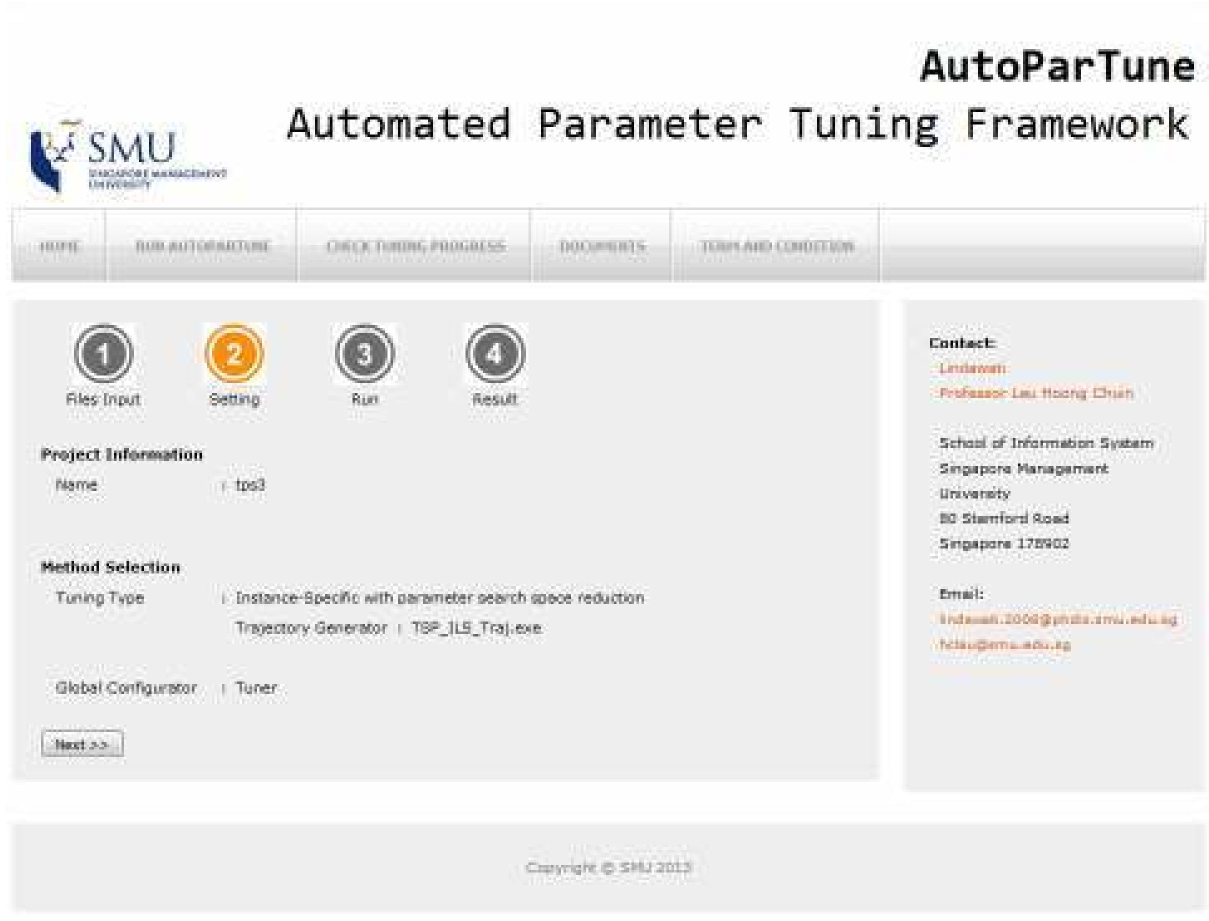


Figure 7. Run AutoParTune – Tuning Method Confirmation

Click "Next" to continue.

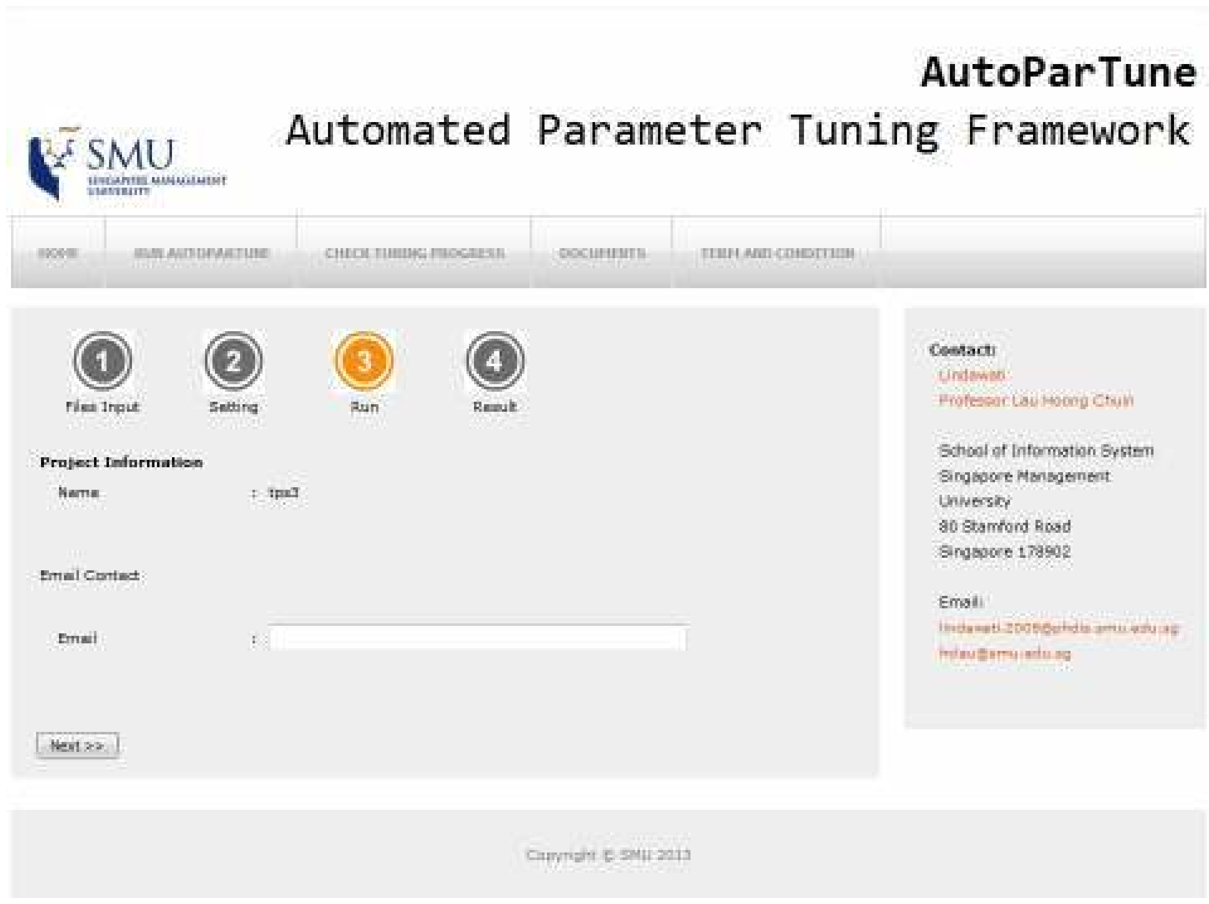


Figure 8. Run AutoParTune – Email Contact

Input the contact email address and click "Next" to finish the input process. Email verification will be sent to the email account. Please use the link in the email to start the tuning process. When the tuning process starts, an email notification will be sent.

Figure 9. Run AutoParTune – Email Verification

Figure 10. Run AutoParTune – Link from Email Verification

After the tuning process is done, an email with the result xml file will be sent to the email address.

To check the tuning progress, click menu “Check Tuning Progress”. Fill in the project id and click “Find Result”.

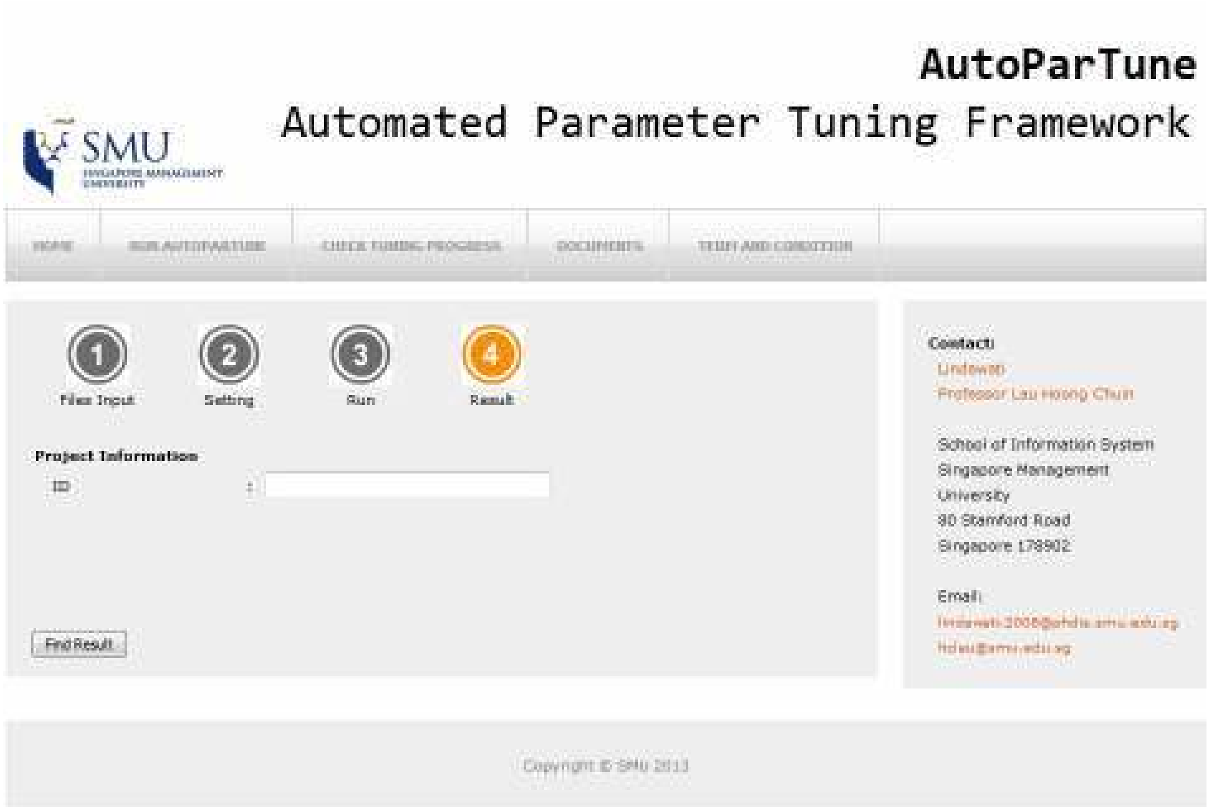


Figure 11. Check Tuning Progress

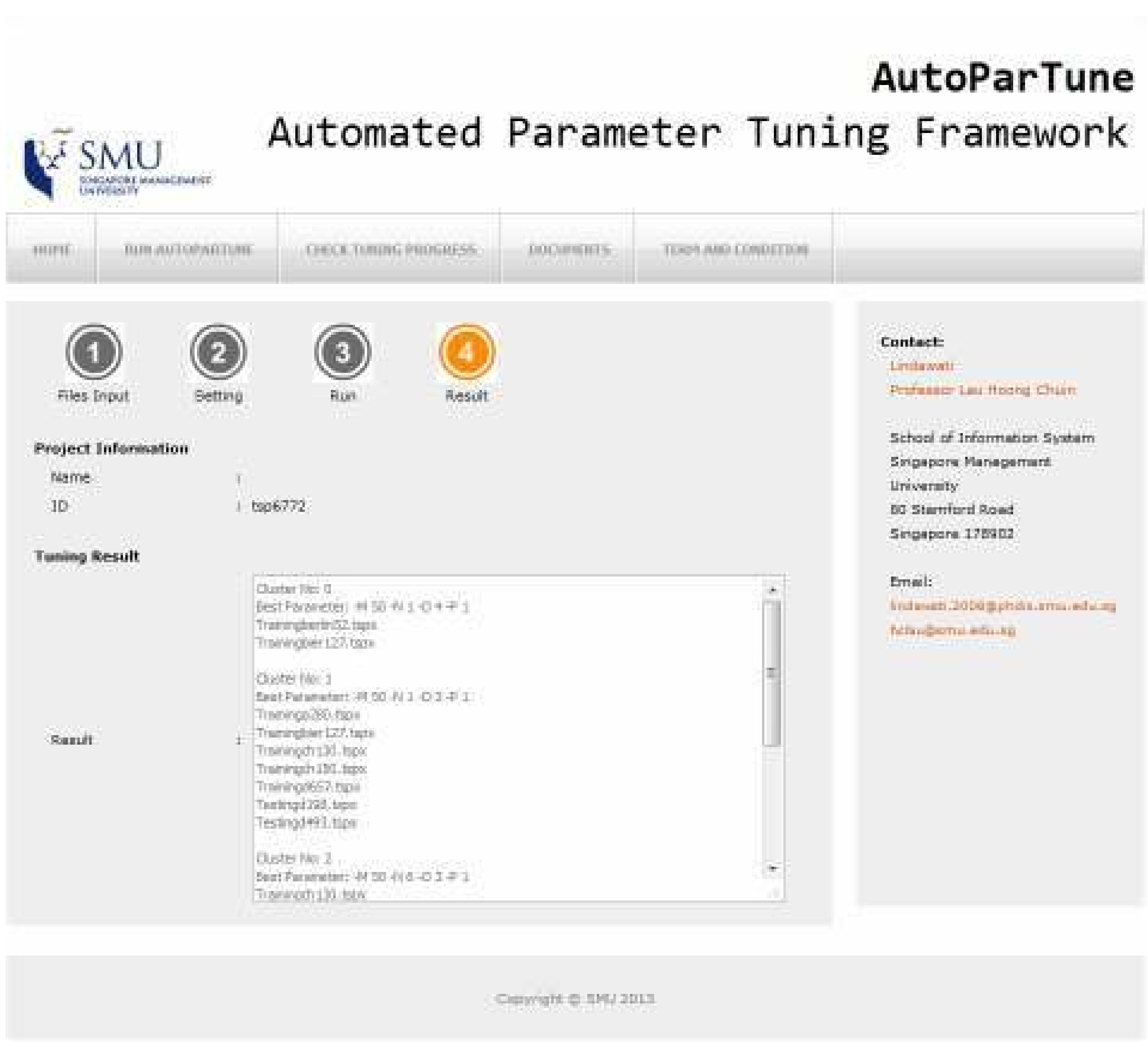


Figure 12. Tuning Result

Click menu “Documents” to view AutoParTune Documentation and related references.

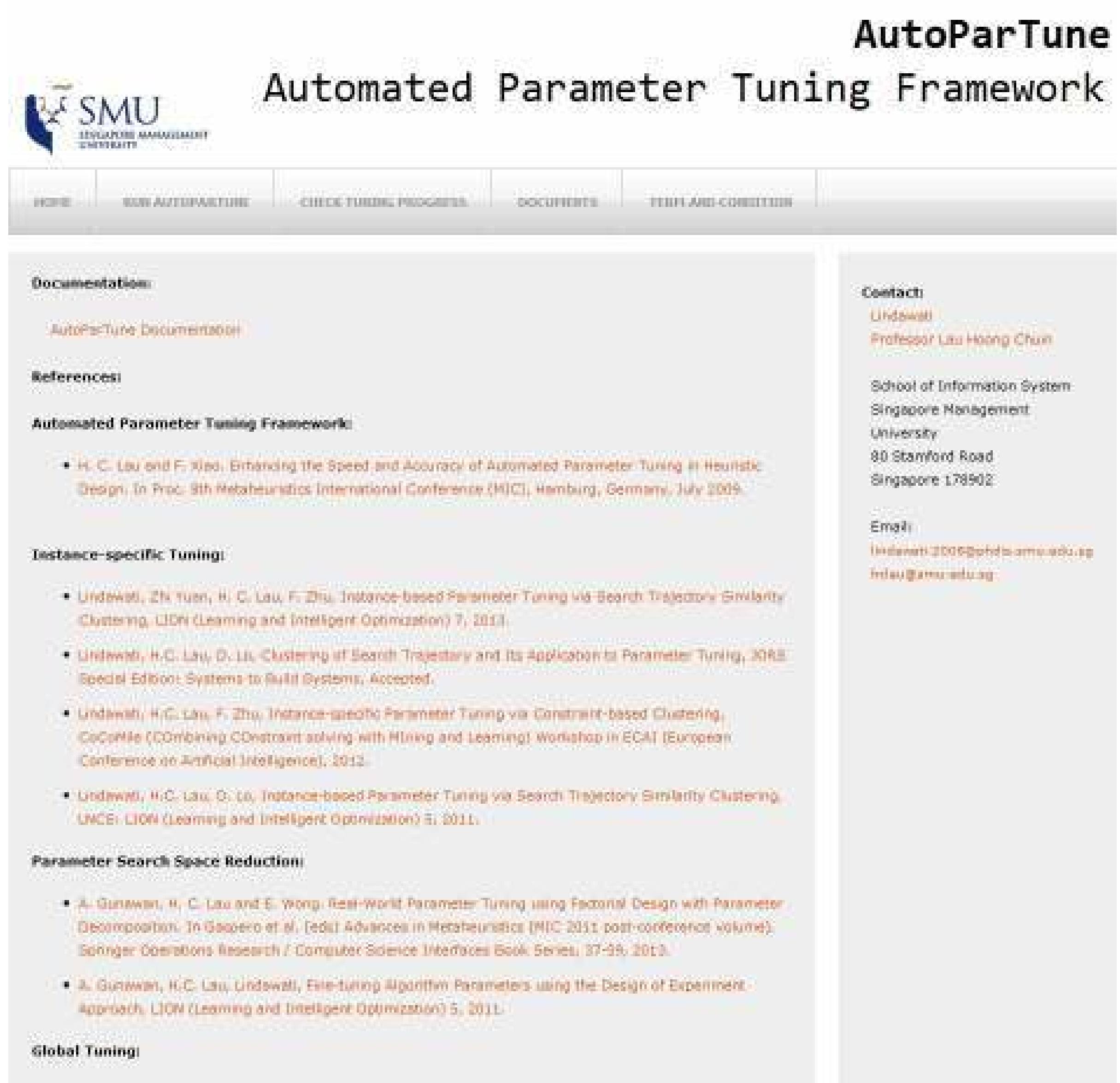


Figure 13. AutoParTune Documents

Click menu “Terms and Conditions” to open AutoParTune terms and conditions.

Figure 14. AutoParTune Terms and Conditions

Bibliography

[1] The traveling salesman problem. `http://www.tsp.gatech.edu/`. Accessed: 2013-06-11.

[2] T. Abell, Y. Malitsky, and K. Tierney. Features for exploiting black-box optimization problem structure. In *Proc. 7th Learning and Intelligent OptimizatioN Conference (LION). To appear.*, 2013.

[3] B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114, 2006.

[4] D. Anderson, E. Anderson, N. Lesh, J. Marks, B. Mirtich, D. Ratajczak, and K. Ryall. Human-guided simple search. In *Proc. 17th National Conference on Artificial Intelligence*, 2000.

[5] H. Andr-Jnsson and D.Z. Badal. Using signature files for querying time-series data. In *Proc. Principles of Data Mining and Knowledge Discovery*, pages 211–220, 1997.

[6] E. Angel and V. Zissimopoulos. On the hardness of the quadratic assignment problem with metaheuristics. *Journal of Heuristics*, 8(4):399–414, 2002.

[7] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proc. 15th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 142–157, 2009.

[8] T. Back, D.B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. IOP Publishing, Bristol, U.K., 1997.

[9] J. Baker and M. Cameron. The effects of the service environment on affect and consumer perception of waiting time: An integrative review and research propositions. *Journal of the Academy of Marketing Science*, 24:338–349, 1996.

[10] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *Proc. 4th International Workshop on Hybrid Metaheuristics*, page 108122. Springer Verlag, 2007.

[11] E. Balas and M.C. Carrera. A dynamic subgradient-based branch and bound procedure for set covering. *Operations Research*, 44:875–890, 1996.

[12] T. Bartz-Beielstein. Experimental research in evolutionary computation: The new experimentalism. *Natural Computing Series*, 2006.

[13] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *Proc. IEEE Congress on Evolutionary Computation*, pages 773–780. IEEE Press, 2005.

[14] R. Battiti, M. Brunato, and P. Campigotto. Learning while optimizing an unknown fitness surface. In *Proc. 2nd Learning and Intelligent OptimizatioN Conference (LION)*, pages 25–40, 2008.

[15] R. Battiti, M. Brunato, and F. Mascia. Reactive search and intelligent optimization. *Operations Research/Computer Science Interface*, 45, 2008.

[16] M. Birattari. *Tuning Metaheuristic: a Machine Learning Perspective*. IOS Press, 2005.

[17] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, pages 11–18, 2002.

[18] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. Automated algorithm tuning using f-races: Recent development. In *Proc. 8th Metaheuristics International Conference (MIC)*, 2009.

[19] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. *F-race and Iterated F-Race: An overview*. Springer Verlag, 2010.

[20] C. Blum and A. Roli. Metaheuristcs in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[21] R.E. Burkard. Quadratic assignment problems. *European Journal of Operational Research*, 18:283–289, 1984.

[22] R.E. Burkard, S.E. Karisch, and F. Rendl. A quadratic assignment problem library. *European Journal of Operational Research*, 55:115119, 1991.

[23] E. Burke and G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer, 2005.

[24] S. Girard C. Bouveyron and C. Schmid. High-dimensional data clustering. *Computational Statistics & Data Analysis*, 52(1):502–519, 2007.

[25] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47:730–743, 1999.

[26] M. Caserta and S. Voss. Corridor selection and fine tuning for the corridor method. In *Proc. 3rd Learning and Intelligent OptimizatioN Conference (LION)*, pages 163–175, 2009.

[27] M. Caserta and S. Voss. A math-heuristic algorithm for the dna sequencing problem. In *Proc. 4th Learning and Intelligent OptimizatioN Conference (LION)*, pages 25–36, 2010.

[28] D. G. Cattrysse and L.N.V. Wassenhove. A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, 60:260–272, 1992.

[29] V. Cerny. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimal Theory Application*, 45:41–51, 1985.

[30] M. Chiarandini, C. Fawcett, and H.H. Hoos. A modular multiphase heuristic solver for post enrolment course timetabling. In *Proc. International Conference on the Practice and Theory of Automated Timetabling (PATAT)*, 2008.

[31] P.C. Chu and J.E. Beasley. A genetic algorithm for the generalized assignment problem. *Computer Operational Research*, 24:17–23, 1997.

[32] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithm*. MIT Press, second edition edition, 2001.

[33] S.P. Coy, B.L. Golden, G.C. Runger, and E.A. Wasil. Using experimental design to find effective parameter setting for heuristics. *Journal of Heuristics*, 7(1):77–97, 2001.

[34] K. Deb and T. Goel. Controlled elitist non-dominated sorting genetic algorithms for better convergence. In E. Zitzler, L. Thiele, K. Deb, C.A. Coello Coello, and D. Corne, editors, *Evolutionary Multi-Criterion Optimization*, volume 1993 of *Lecture Notes in Computer Science*, pages 67–81. Springer Berlin Heidelberg, 2001.

[35] J.A. Diaz and E. Fernandez. A tabu search heuristic for the generalized assignment problem. *European Journal of Operational Research*, 132:22–38, 2001.

[36] C. Ding and X. He. K-means clustering via principal component analysis. In *Proc. 21th International Conference on Machine Learning*, 2004.

[37] C. Fawcett, M. Helmert, H.H. Hoos, E. Karpas, G. Rger, and J. Seipp. Fd-autotune: Domain-specific configuration using fast-downward. In *Proc. of International Conference on automated Planning and Scheduling - Planning and Learning (ICAPS-PAL)*, 2011.

[38] M. Finger, T. Stützle, and H. Lourenço. Exploiting fitness distance correlation of set covering problems. In *LNCS: Applications of Evolutionary Computing*, 2002.

[39] M. Gagliolo and J. Schmidhuber. Dynamic algorithm portfolio. In *Proc. 9th International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2006.

[40] M.R. Garey and D.S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. William H. Freeman, 1979.

[41] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M.T. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning*, pages 352–357, 2011.

[42] F. Glover and M. Laguna. *Tabu Search*. Springer, 1997.

[43] C. Gomes and B. Selman. Algorithm portfolio. *Artificial Intelligence*, 126:43–62, 2001.

[44] A. Gunawan, H. C. Lau, and E. Wong. Real-world parameter tuning using factorial design with parameter decomposition. In *Proc. 9th Metaheuristics International Conference (MIC)*, 2011.

[45] A. Gunawan, Hoong Chuin Lau, and Lindawati. Fine-tuning algorithm parameters using the design of experiments approach. In *Proc. 5nd Learning and Intelligent OptimizatioN Conference (LION)*, page 278292, 2011.

[46] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[47] S. Halim. *An Integrated White+Black Box Approach for Designing and Tuning Stochastic Local Search Algorithms*. Ph.D. thesis, National University of Singapore, Singapore, 2009.

[48] S. Halim, Y. Yap, and H.C. Lau. Viz: A visual analysis suite for explaining local search behavior. In *Proc. 19th ACM symposium on User Interface Software and Technology (UIST)*, pages 57–66, 2006.

[49] S. Halim, Y. Yap, and H.C. Lau. An integrated white+black box approach for designing and tuning stochastic local search. In *Proc. 13th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 332–347, 2007.

[50] J. Han, H. Cheng, and D. Xin. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.

[51] J. Han and M. Kamber. *Data Mining: Concept and Techniques, 2nd Edition*. Morgan Kaufman, San Francisco, 2006.

[52] K. Helsgaun. An effective implementation of the linkernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[53] W. Herroelen and A. Van Gils. On the use of flow dominance in complexity measures for facility layout problems. *International Journal of Production Research*, 23:97–108, 1985.

[54] R.R. Hill and C.H. Reilly. The effects of coefficient correlation structure in two-dimensional knapsack problems on solution procedure performance. *Management Science*, 46(2):302–317, 2000.

[55] H.H. Hoos and T. Stützle. *Stochastic Local Search: Foundation and Application*. Morgan Kaufman, San Francisco, 2004.

[56] F. Hutter and Y. Hamadi. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. In *Technical Report*. Microsoft Research, 2005.

[57] F. Hutter, Y. Hamadi, H.H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Proc. 18th International Conference of Principles and Practice of Constraint Programming (CP)*, pages 213–228, 2006.

[58] F. Hutter, H.H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. 5nd Learning and Intelligent OptimizatioN Conference (LION)*, 2011.

[59] F. Hutter, H.H. Hoos, K. Leyton-Brown, and K. Murphy. Time-bounded sequential parameter optimization. In *Proc. 4nd Learning and Intelligent OptimizatioN Conference (LION)*, 2010.

[60] F. Hutter, H.H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[61] F. Hutter, H.H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proc. 22nd AAAI Conference on Artificial Intelligence*, pages 1152–1157. AAAI Press, 2007.

[62] D. S. Johnson and L. A. McGeoch. The traveling salesman problem - a case study in local optimization. *Local Search in Combinatorial Optimization*, pages 215–310, 1997.

[63] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[64] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. Isac: Instance-specific algorithm configuration. In *Proc. 19th European Conference on Artificial Intelligence (ECAI)*, pages 751–756, 2010.

[65] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, 1990.

[66] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, 2004.

[67] S. Kimbrough, A. Kuo, H. C. Lau, Lindawati, and D. H. Wood. On using genetic algorithms to support post-solution deliberation in the generalized assignment problem. In *Proc. 8th Metaheuristics International Conference (MIC)*, 2009.

[68] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 4598:671–680, 1983.

[69] J. Knowles and D. Corne. Instance generators and test suites for the multiobjective quadratic assignment problem. In *Proc. Evolutionary Multi-Criterion Optimization*, 2003.

[70] C. Kroer and Y. Malitsky. Feature filtering for instance-specific algorithm configuration. In *Proc. 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 849–855, 2011.

[71] H.C. Lau and F. Xiao. Enhancing the speed and accuracy of automated parameter tuning in heuristic design. In *Proc. 8th Metaheuristics International Conference (MIC)*, 2009.

[72] T.L. Lau and E.P.K. Tsang. The guided genetic algorithm and its application to the generalised assignment problem. In *Proc. IEEE International Conference on Evolutionary Programming*, pages 135–155, 1995.

[73] E.L. Lawler. The quadratic assignment problem. *Management Science*, 9:586–599, 1963.

[74] E.L. Lawler, J.K. Lestra, A.H.G. Rinnooy Kan, and D.B. Shmoys (eds.). *The Traveling Salesman Problem: A guided Tour of Combinatorial Optimization*. John Wiley and Sons, 1983.

[75] T. Warren Liao. Clustering of time series dataa survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[76] A.W. Loo. *Peer-to-Peer computing: building supercomputers with web technologies*. Springer, 2007.

[77] H.R. Lourenço, O.C. Martin, and T. Stützle. Iterated local search. *Handbook of Metaheuristics, International Series in Operations Research & Management Science*, 57:320–353, 2003.

[78] W. Macready and D. Wolpert. What makes an optimization problem hard. *Complexity*, 5:40–46, 1996.

[79] Y. Malitsky and M. Sellmann. Stochastic offline programming. In *Proc. IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 351–371, 2009.

[80] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, New York, 1990.

[81] J. Martikainen and S.J. Ovaska. Hierarchical two-population genetic algorithm. *Proc. International Journal of Computational Intelligence Research (IJCIR)*, 2:367–380, 2006.

[82] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1999.

[83] D.C. Montgomery and G.C. Runger. *Applied Statistics and Probability for Engineers 2nd Edition*. John Wiley & Son, 1999.

[84] M. Muja and D.G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proc. International Conference on Computer Vision Theory and Applications (VISAPP)*, pages 331–340, 2009.

[85] N. Musliu. Local search algorithm for unicost set covering problem. In *LNCS: Advances in Applied Artificial Intelligence*, 2006.

[86] R.M. Nauss. Solving the generalized assignment problem: An optimizing and heuristic approach. *Informs Journal on Computing*, 15:249–266, 2003.

[87] K.M. Ng, A. Gunawan, and K.L. Poh. A hybrid algorithm for the quadratic assignment problem. In *Proc. International Conference on Scientific Computing*, pages 14–17, 2008.

[88] G. Ochoa, S. Verel, F. Daolio, and M. Tomassini. Clustering of local optima in combinatorial fitness landscape. In *Proc. 5nd Learning and Intelligent OptimizatioN Conference (LION)*, 2011.

[89] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Mineola, N.Y., 1998.

[90] D.J. Patterson and H. Kautz. Auto-walksat: A self-tuning implementation of walksat. *Electronic Notes in Discrete Mathematics*, 9:360–368, 2001.

[91] C.R. Reeves. Landscapes, operators and heuristic search. *Annals of Operations Research*, 86(1):473–490, 1999.

[92] C.R. Reeves. Landscapes, operators and heuristic search. *Annals of Operations Research*, 86(1):473–490, 1999.

[93] C.H. Reilly. Synthetic optimization problem generation: Show us the correlations! *INFORMS Journal on Computing*, 21:458–467, 2009.

[94] F. Rossi, P.V. Beek, and T. Walsh. *Handbook of Constraint Programming: Foundations of Artificial Intelligence*. Elsevier Science and Technology Books, 2006.

[95] S. Salvador and P. Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *Proc. 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 576–584, 2004.

[96] M. Savelsbergh. A branch-and-price algorithm for the generalized assignment problem. *Operation Research*, 45:831–841, 1997.

[97] C.T. Shaw and G. P. King. Using cluster analysis to classify time series. *Physica D: Nonlinear Phenomena*, 58(1):288–298, 1992.

[98] S.N. Sivanandam and S.N. Deepa. *Introduction to Genetic Algorithms*. Springer, 2008.

[99] K. Smith-Miles, J. Hemert, and X.Y. Lim. Understanding tsp difficulty by learning from evolved instances. In *Proc. 4th Learning and Intelligent OptimizatioN Conference (LION)*, pages 266–280, 2010.

[100] K. Smith-Miles and L. Lopes. Measuring instance difficulty for combinatorial optimization problems. *Computer and Operations Research*, 39:875–889, 2012.

[101] T. Stützle. Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519–1539, 2006.

[102] T. Stützle and S. Fernandes. New benchmark instances for the qap and the experimental analysis of algorithms. In *LNCS: Evolutionary Computation In Combinatorial Optimization*, 2004.

[103] K. Sugihara. Measures for performance evaluation of genetic algorithms. In *Proc. 3rd Joint Conference on Information Sciences (JCIS)*, 1997.

[104] E.D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[105] É.D. Taillard. Comparison of iterative searches for the quadratic assignment problem. *Location Science*, 3(2):87–105, 1995.

[106] J. Tavares, FB. Pereira, and E. Costa. Multidimensional knapsack problem: a fitness landscape analysis. *Proc. IEEE International Conference on Systems, Man, and Cybernetics*, 38(3):604–616, 2008.

[107] H. Terashima-Marín and P. Ross. Evolution of constraint satisfaction strategies in examination timetabling. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, pages 635–642, 1999.

[108] J. A. Vasconcelos, J. A. Ramrez, R. H. C. Takahashi, and R. R. Saldanha. Improvements in genetic algorithms. *Proc. IEEE Transactions aon Magnetics*, 37:3414–3417, 2001.

[109] T.E. Vollmann and E.S. Buffa. The facilities layout problem in perspective. *Management Science*, 12(10):450–468, 1966.

[110] W.L. Winston. *Operations Research: Applications and Algorithms*. Thomson Brooks/Cole, 2004.

[111] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation*, 1(1):67–82, 1997.

[112] J. Xu, S.Y. Chiu, and F. Glover. Probabilistic tabu search for telecommunications network design. *Combinatorial Optimization: Theory and Practice*, 1(1):69–94, 1996.

[113] J. Xu, S.Y. Chiu, and F. Glover. Fine-tuning a tabu search algorithm with statistical tests. *International Transactions in Operational Research*, 5(3):233–244, 1998.

[114] L. Xu, H.H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proc. Conference of the Association for the Advancement of Artificial Intelligence (AAAI-10)*, 2010.

[115] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[116] M. Yagiura, T. T. Ibaraki, and F. Glover. A path relinking approach with ejection chains for the generalized assignment problem. *European Journal of Operational Research*, 169:548–569, 2006.

[117] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proc. IEEE International Conference on Data Mining (ICDM)*, page 721723, 2002.

[118] J. Yang, W. Wang, and P. S. Yu. Mining asynchronous periodic patterns in time series data. *IEEE Transactions on Knowledge and Data Engineering*, 15:613–628, 2003.

[119] K.Y. Yeung and W.L. Ruzzo. Principal component analysis for clustering gene expression data. *Bioinformatics*, 17(9):763–774, 2001.

[120] Z. Yuan, K. Tierney, Lindawati, and H.C. Lau. Private Communication, March 2013.