Parameter-less Late Acceptance Hill-climbing: Foundations & Applications

Mosab Bazargani
School of Electronic Engineering and Computer Science
Queen Mary University of London

Submitted for the degree of
Doctor of Philosophy

2021

Abstract

Stochastic Local Search (SLS) methods have been used to solve complex hard combinatorial problems in a number of fields. Their judicious use of randomization, arguably, simplifies their design to achieve robust algorithm behaviour in domains where little is known. This feature makes them a general purpose approach for tackling complex problems. However, their performance, usually, depends on a number of parameters that should be specified by the user. Most of these parameters are search-algorithm related and have little to do with the user's problem.

This thesis presents search techniques for combinatorial problems that have fewer parameters while delivering good anytime performance. Their parameters are set automatically by the algorithm itself in an intelligent way, while making sure that they use the entire given time budget to explore the search space with a high probability of avoiding the stagnation in a single basin of attraction. These algorithms are suitable for general practitioners in industry that do not have deep insight into search methodologies and their parameter tuning. Note that, to all intents and purposes, in real-world search problems the aim is to find a good enough quality solution in a pre-defined time.

In order to achieve this, we use a technique that was originally introduced for automating population sizing in evolutionary algorithms. In an intelligent way, we adapted it to a particular one-point stochastic local search algorithm, namely Late Acceptance Hill-Climbing (LAHC), to eliminate the need to manually specify the value of the sole parameter of this algorithm. We then develop a mathematically sound dynamic cutoff time strategy that is able to reliably detect the stagnation point for these search algorithms. We evaluated the suitability and scalability of the proposed methods on a range of classical combinatorial optimization problems as well as a real-world software engineering problem.

I dedicate this work to the most beautiful expressions of my life;
"my mother and my dad".

Acknowledgements

First and foremost, I would like to thank my supervisor Professor Edmund K. Burke for accepting me as his student and his enthusiastic encouragement. He gave me the freedom to explore and pursue the research avenues that I really love, and at the same time he constantly provided me feedback, and guided me in my journey. I was fortunate to learn from him, not only about scientific methods, but also about my role as a researcher and a scholar. I could not have imagined having a better supervisor.

I am very grateful to Fernando G. Lobo for his guidance and friendship, as well as for accepting my several visits to him at University of Algarve. With him, I have learn many things concerning research; he taught me to write better, taught me to explain things better, and taught me that hard work pays off. I also enjoyed our conversions about philosophy, social science, and the future of democracy. I am also thankful to his wife, Paula, and his kids, Gil and Ana, for inviting me several times in their house.

I am also very thankful to Jun Chen for being there for me at the most difficult time of my PhD. His support and advise during difficult days are very precious to me. He taught me how to focus on my goals, in any organization, without being diverted into unnecessary challenges. I also learn from him the importance of hunting academic funds. I feel very fortunate to have had the opportunity to work with him.

I am similarly grateful to Fabrizio Smeraldi who I got to know early in my journey. Thank you for the friendship, the support and advice you gave me, our conversation, and afternoon coffee. I am happy that we continue this all virtually throughout COVID-19 pandemic.

I also want to thank Arman Khouzani, for offering me his couch to sleep, right before my stage one, when I didn't have a place to stay. Thank you for the friendship, the advice, and our conversation. I learn a lot from you

as a demonstrator and, later on, as the teaching fellow. Thank you for all this.

For more than two years, I had the honour to be the EECS Chief PhD representative and closely work with a group of representatives from different research groups and centres as well as interacting with a number of EECS admin staff. I would like to thank all of them, especially, Alan, Edward, Laura, Nicole, Keith, Melissa, Hayley, Julie, Lowri, Yue, Dorothee. Thank you for all the support you gave me in this journey.

I would like to thank all the current and former Operational Research lab members with whom I had the pleasure to work and interact. I am very grateful to all my colleagues and friends, Ali, Farid, Una, Elham, Maryam, Louise, and Stathis. Thank you all for the company and friendship, for all the good times we shared, for all the funny moments, and for being an inspiration to me.

Kind gratitude to my beloved family, my mother who always wanted me to complete my post-graduate study, my father and two sisters, Mena and Marve, who taught me to be myself. They taught me to never give up in achieving my ultimate goals and helped me pursue an academic degree. They are the spiritual support of my life. I owe a big thank you to you. A very special thanks goes to Houman Samim, my brother-in-law, who has always been there for me.

I thank EU Cost Action CA15140 for funding my Short-Term Scientific Mission to visit the University of Algarve in 2017 and 2019. The work in this thesis was partially sponsored by EPSRC grant EP/J017515/1.

Declarations

Parts of this thesis have appeared in the several publications which have been subject to peer review; one of them is still under review. These publications are listed below:

Chapter 3 is based on the two following papers:

- Bazargani, M., Lobo, F. G. (2017). Parameter-less late acceptance hill-climbing. *ACM SIGEVO Genetic and Evolutionary Computation Conference (GECCO'17)*, ACM Press, pp. 219–226.
  doi: 10.1145/3071178.3071225.
- Bazargani, M., Drake, J. H., Burke, E. K. (2018). Late acceptance hill climbing for constrained covering arrays. *21st European Conference on the Applications of Evolutionary Computation (EvoApplications)*, In *Lecture Notes in Computer Science*, Springer, pp. 778–793. **Best Paper Award Nomination**.
  doi: 10.1007/978-3-319-77538-8_52.

Chapter 4 is based on:

- Lobo, F. G., Bazargani, M., Burke, E. K. (2020). A cutoff time strategy based on the coupon collector's problem. *European Journal of Operational Research*. Elsevier. pp. 101–114.
  doi: 10.1016/j.ejor.2020.03.027.

Chapter 5 is based on:

- Bazargani, M., Lobo, F. G., Burke, E. K. (2020). Parameter-less late acceptance hill-climbing: an anytime performance local search algorithm. [under review].

I would like to express here my thanks to all co-authors.

Licence

Contents

List of Figures

List of Tables

List of Abbreviations

AMS	Adaptive Multi-Start heuristic
ARPD	Average Relative Percentage Deviation
ATSP	Asymmetric Travelling Salesman Problem
CA	Covering Arrays
CALA	Covering Arrays by Late Acceptance
CASA	Covering Arrays by Simulated Annealing
CATS	Covering Array by Tabu Search
CCA	Constrained Covering Arrays
CCP	Coupon Collector's Problem
CIT	Combinatorial Interaction Testing
CMA-ES	Covariance Matrix Adaptation Evolutionary Strategy
EA	Evolutionary Algorithm
ES	Evolutionary Strategy
FDA	Fitness-distance Analysis
FE	Function Evaluations
GA	Genetic Algorithm
GDA	Great Deluge Algorithm
GOMEA	Gene-pool Optimal Mixing EA
GRASP	Greedy Adaptive Search Procedure heuristic
hBOA	Hierarchical Bayesian Optimization Algorithm
HC	Hill Climbing
II	Iterative Improvement

IIL	Inner Iteration Limit
ILS	Iterated Local Search
I_{max}	Maximum number of improving moves
I_{move}	Average number of improving moves
irace	Iterated Racing
LAHC	Late Acceptance Hill-Climbing
MKP	Multidimensional Knapsack Problem
mLAHC	Multistart LAHC
NEH	Algorithm of Nawaz, Enscore & Ham (1983)
OF	Overhead Factors
OR	Operational Research
PFSP	Permutation Flowshop Scheduling Problem
pLAHC	Parameter-less Late Acceptance Hill-Climbing
pLAHC-s	Parameter-less LAHC with seeding
QAP	Quadratic Assignment Problem
QAPLIB	Quadratic Assignment Problem Library
SA	Simulated Annealing
SAT	Boolean Satisfiability Problem
SBSE	Search Based Software Engineering
SIL	Successful Iterations List
SLS	Stochastic Local Search
SPO	Sequential Parameter Optimization
TA	Threshold Accepting
TS	Tabu Search
TSP	Travelling Salesman Problem
TSPLIB	Travelling Salesman Problem Library
VLSI	Very Large-Scale Integration
VRF	PFSP benchmark of Vallada, Ruiz & Framinan (2015)

Science is nothing but perception.

Plato

Chapter 1

Introduction

This thesis is about one-point stochastic local search algorithms inspired from Darwin's theory of evolution. These are heuristic methods for solving complex and computationally hard problems — often combinatorial problems. They are suitable for problems that can be formulated as a search problem and aim to find a solution, with a maximum objective function value, among a set of candidate solutions. They move from one solution to another in the space of candidate solutions, induced by a small change to the current solution, until an optimum solution is found in a given time budget. They have been successfully applied in a number of different fields, including operational research, engineering, medicine, finance, and even arts. Notwithstanding, the performance of these algorithms, to a very great degree, relies on the tuning of their parameter(s) value(s). It often requires a good level of expertise and knowledge about these algorithms to tune them in a way that makes them work well in practice. Moreover, tuning is a quite time-consuming exercise. Therefore, it is no surprise that most general practitioners in industry have no choice other than to undertake ad-hoc experimentation with a variety of parameter settings. Consequently, search methodologies as implemented today —albeit robust— are not often easy-to-use and require a lot of parameter tuning. Note that, to all intents and purposes, in real-world applications the aim is to find a good enough quality solution in a pre-defined time.

The results of parameter tuning also depend on the particular computational infrastructure in which the search algorithm is launched for a given time budget. The computational resources are gradually improving; thereby,

the time required to execute a single iteration of a search algorithm is also being gradually reduced. This impacts on previously tuned parameters when their computational infrastructure is changed. So, it is instrumental to have search methodologies that use all of the given time budget in an intelligent manner to yield a quality solution regardless of the used computational infrastructure.

Today's complex real-world problems have parameters themselves. For example, the number of drivers working in an online shopping store. These problem-related parameters can often be amended by the stakeholders if that can lead to a better quality solution. The conglomeration of problem-related and search-algorithm-related parameters creates even further complexity in addressing real-world problems, something that should be avoided. One way to achieve this is by reducing the parameters of search algorithms.

1.1 Thesis Objectives

The goal of this thesis is to design search techniques that have fewer parameters for combinatorial problems while delivering good *anytime performance*. To achieve this goal, the work carried out herein is broken down to the following four objectives:

- Introduce the parameter-less Late Acceptance Hill-Climbing (pLAHC) by eliminating the sole parameter of the late acceptance hill-climbing, making a simple algorithm even simpler to apply in practice.
- Investigate the functionality of pLAHC in a real-world problem by adapting it to an existing complex search framework.
- Design a mathematically sound dynamic cutoff time strategy for one-point stochastic local search algorithms that accepts worsening moves, applied to combinatorial problems.
- Finally, incorporate the proposed cutoff time strategy in pLAHC.

1.2 Introduced Algorithms

In order to fulfil the objectives of this thesis, the following algorithms are introduced throughout this thesis:

- **parameter-less Late Acceptance Hill-Climbing (pLAHC)** embeds restart mechanism in the LAHC. It starts by running LAHC with a small history list length. Thereafter, whenever —with a high probability— it is in a local optimum, it triggers a new run of LAHC with exponentially increasing history list length.

- **parameter-less Late Acceptance Hill-Climbing with seeding (pLAHC-s)** has the exact same principle as pLAHC, except when a new LAHC is trigged, it uses previously obtained knowledge to set the initial solution and the history list.

- **multistart Late Acceptance Hill-Climbing (mLAHC)** restarts LAHC with a fixed history list length.

- **Coupon Collector's Problem (CCP) cutoff time** employs a classical problem in probability theory to introduce a new cutoff time. This mathematically sound cutoff time is used as a restating mechanism in above mentioned algorithms.

1.3 Main Contributions

The main contributions of this thesis are:

- Introduce search approaches that have fewer parameters while delivering a quality solution (Bazargani & Lobo 2017, Bazargani, Drake & Burke 2018, Lobo, Bazargani & Burke 2020).

- Propose an automated strategy to eliminate the only parameter of the Late Acceptance Hill-Climbing (LAHC) algorithm (Bazargani & Lobo 2017), a general purpose metaheuristic. It successfully adapts a technique, that was previously introduced for automating population sizing in evolutionary algorithms, to avoid the need to tune the value of the parameter of LAHC. Source code for this work is made available online at `https://github.com/mbazargani/pLAHC`.

- A case study of the application of pLAHC to a real-world problem (Bazargani, Drake & Burke 2018). It replaces simulated annealing used in a well-known existing search framework with LAHC and uses the structure of the search framework to automatically decide when to

trigger a new run of LAHC with exponentially increasing history list length.

- Introduce a mathematically sound dynamic cutoff time strategy that is able to reliably detect the stagnation point for one-point stochastic local search algorithms applied to combinatorial problems (Lobo, Bazargani & Burke 2020). The strategy is derived from the Coupon Collector's Problem, and is scalable based on the employed perturbation operator and its induced neighbourhood size, as well as from feedback from the search.

- Evaluate the proposed cutoff time on a range of common benchmark sets derived from three classical NP-hard problems, namely the Travelling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP), and the Permutation Flowshop Scheduling Problem (PFSP), using two different perturbation operators (Lobo, Bazargani & Burke 2020). A complete set of the results of these experiments is published in a 79-page PDF document as supplementary material available at `https://bit.ly/3foVjud`. Furthermore, the same information is also available in a companion website at `https://mbazargani.github.io/CCPcutoffTime/`, which has several useful features such as searching and sorting, as well as allowing the results to be exported in a variety of formats such as JSON, XML, CSV, and SQL.

- For the first time, using a construction heuristic (that provides a very good solution) to create an initial solution for LAHC while making sure that it still accepts worsening moves (Lobo, Bazargani & Burke 2020).

- Improve the best-known solutions of 71 PFSP instances (out of 480) (Lobo, Bazargani & Burke 2020), taken from a newly introduced hard benchmark by Vallada, Ruiz & Framinan (2015) —known as VRF benchmark. The solutions to these instances and their makespan costs (C_{max}) can be obtained from `https://mbazargani.github.io/CCPcutoffTime/PFSP/new-vfr-best-knowns.txt`.

- Introduce an *anytime performance* local search algorithm that has no parameters while delivering a quality solution, making it a suitable algorithm for general practitioners who do not have deep insight into

the search methodologies. This research work is submitted and is under review.

- Study the applications of the search methodologies to the search based software engineering problems (Bazargani, Drake & Burke 2018, Guizzo, Bazargani, Paixao & Drake 2017).

1.4 Thesis structure

The thesis is composed of six chapters. Chapter 1 introduces the motivation behind this research work, and states the main objectives. Following that, it enumerates the contributions to the state-of-the-art, and outlines the contents of the remaining chapters.

Chapter 2 reviews the fundamental concepts and definitions of stochastic local search algorithms designed for combinatorial problems. It starts by explaining the importance of these algorithms and introducing combinatorial problems. The main components of most on-point stochastic local search algorithms are also given in this chapter. Thereafter, it presents a range of well-known stochastic local search methods, their most relevant variants and components. A few key application areas are highlighted and some examples are cited to demonstrate the diversity of real-world problems where these algorithms have been successfully applied. It also describes the LAHC algorithm and provides a brief analysis and applications of it. Hyper-heuristics are explained next, since some SLS algorithms, introduced in this chapter, are employed as the acceptance mechanism in selection hyper-heuristics in a variety of application areas. The chapter ends by presenting some theoretical aspects of search methodologies, namely cutoff time strategies, the parameter tuning, and the search space.

Chapter 3 first explains the parameter-less search scheme that was used in the evolutionary algorithms. Then, it establishes a connection between the history list length of LAHC and the related problem of population sizing in evolutionary algorithms. Based on this analysis, it introduces the parameter-less LAHC algorithm (pLAHC) that has no parameter. A refined version of pLAHC that uses seeded restarts is also investigated in this chapter. The validity of the method is shown with a number of computational experiments applying it to several TSP benchmark instances. The last part of the chapter focuses on a real-world application of pLAHC by us-

ing a problem that is taken from the search based software engineering field, i.e., combinatorial interaction testing problem. The combinatorial interaction testing is a cost-effective black-box sampling technique for discovering interaction faults in highly configurable systems. For illustration purpose and a fare analysis of the proposed approach, the pLAHC is adapted within a well-known existing search framework for this problem. The problem and its benchmark, the search framework, and background material are all presented in this part. The example is used as a case study but similar design principles may be applied to other kinds of real-world problems as well.

Chapter 4 explains the pitfalls of the stagnation in search algorithms. The Coupon Collector's Problem and how it can be used to design a cutoff strategy for stochastic local search is presented here. Although the proposed method is generic, the suitability and scalability of the method, in this chapter, is illustrated with the LAHC algorithm on a comprehensive set of benchmark instances of three well-known combinatorial optimization problems: the Travelling Salesman Problem, the Quadratic Assignment Problem, and the Permutation Flowshop Scheduling Problem. It also provides a discussion about two criticisms that can be made to the proposed technique.

Chapter 5 discusses the tradeoff between search time and solution quality, the importance of anytime performance, and the benefits of parameterless search. It presents a refinement of pLAHC that uses a restart strategy based on the dynamic cutoff time introduced in Chapter 4. The resulting algorithm has no parameters and a comprehensive series of experiments shows that it provides good anytime performance, competitive to state-of-the-art ones.

Chapter 6 summarizes and concludes the major contributions of the thesis, and gives suggestions for future research.

Evolution continually innovates, but at each level it conserves the elements that are recombined to yield the innovations.

John H. Holland

Chapter 2

Stochastic Local Search

A good number of local shops in the United Kingdom have started home delivery services after the breakout of COVID-19. In a nutshell, every day, they collect several customers' orders and then deliver them to their home addresses, preferably in one go in a day. It is cost-effective for a shop and its customers to carry out deliveries by choosing a shortest path (tour) that goes through all addresses starting from the shop and ending there. Google Maps Platform provides the Distance Matrix API that calculates travel distance and time for a matrix of origins and destinations. This API allows us to measure the distance between every two locations. Now, the foremost question that remains to be answered is *in what order customers should be visited to minimize the cost?* The cost can be the elapsed time, the total distance, or a form of blended function of both time and distance. To explain the computational complexity of this problem, let us consider an example. Imagine a shop has 20 deliveries for a particular day. That makes 20! different possible tours (candidate solutions) for that day to be considered to find the best solution. If computing the cost of each solution takes one nanosecond[1], then our shop needs to wait about 2815 days to find the best (optimal) solution for that day.

Of course, in practice, it is not viable to examine all possible candidate solutions. An attractive approach is to start from a somewhat arbitrary tour (solution), then, move from one solution to another by iteratively performing

[1] $10^{-9} second$. Note that a processor which works at 1GHz has one billion clock cycles per second. In this example, we assume that computing the cost of a solution takes one clock cycle. In reality, it takes much more than one clock cycle; however, we simplified it for the sake of clarity.

small changes on a given tour, with a goal of improving the cost of it. This type of approach is known as a Stochastic Local Search (SLS) algorithm. Such approaches are often the methods of choice for solving combinatorial problems like the one illustrated above. It may be noted that the above mentioned problem can be seen as a variation of the Travelling Salesman Problem (TSP), which will be more formally introduced in the next section.

This chapter is devoted to fundamental concepts and definitions of SLS algorithms designed for combinatorial problems. Some theoretical aspects of SLS that are a prerequisite to understand later chapters are also covered here. It is organized as follows. It starts with an introduction to combinatorial problems and presents the travelling salesman problem for an illustrative purpose. Other combinatorial problems used in this work will be explained in the chapters to come. Section 2.2 introduces a range of SLS methods, their most relevant variants, components, and presents some interesting applications of them to real-world problems in the literature. This is followed by an introduction to the Late Acceptance Hill-Climbing algorithm, a brief analysis and applications of it. Hyper-heuristics are not the subject of this thesis. However, since we have mentioned some applications of hyper-heuristics based local search algorithms (introduced in this chapter), they are briefly explained in Section 2.3. Moreover, hyper-heuristics feature in the section on future research (Section 6.2.) Thereafter, there are three sections that consider more theoretical aspects of search algorithms. Section 2.4 discusses some related work on cutoff time strategies and search stagnation. Section 2.5 briefly presents parameter tuning in SLS methods. Search space and its impact on behaviour of SLS is discussed in Section 2.6. Finally, a brief summary of this chapter is given in Section 2.7.

2.1 Combinatorial Problems

Combinatorial problems have important applications in several areas, such as artificial intelligence, operational research, machine learning, finance, bioinformatics, and software engineering. In a nutshell, a combinatorial problem has a finite set of *candidate solutions*, each of which consists of discrete objects, and the intention is to find the quality ones among them. Some of the well-known combinatorial problems are planning, scheduling, vehicle routing, timetabling, hardware design, and resource allocation.

Combinatorial problems can be classified as *decision problems* or *optimization problems*. In decision problems, the objective is to find a solution or a set of solutions to a given problem instance composed of a set of logical conditions (Hoos & Stützle 2005, p. 15). The aim in this type of problem is to find a feasible solution(s) or to come to a decision that no solution exists to satisfy all the conditions. A well-known combinatorial decision problem is the Boolean Satisfiability Problem (SAT) (Selman, Levesque & Mitchell 1992). In combinatorial optimization problems, the solutions are evaluated by an *objective function* to find those with optimal objective function values, a.k.a. *solution quality* (Hoos & Stützle 2005, p. 15). The goal here would be to *minimize* or *maximize* solution quality. TSP is an example of a minimization problem (see Section 2.1.1), and the Linear Ordering Problem (Schiavinotto & Stützle 2004) is a maximization problem. In cases where the objective function amounts to a measure of the cost of a solution, then it is a *cost function* and typically the aim is to minimize it.

Many combinatorial optimization problems can be also seen as an extension of decision problems, since they have *constraints*. The vast majority of real-world combinatorial optimization problems have this characteristic. In this type of problem, candidate solutions must satisfy a set of constraints to become *feasible* solutions; so then, optimal solution(s) is (are) identified, based on their objective function values, among feasible solutions.

2.1.1 A Prototypical Combinatorial Problem: TSP

In the following, we introduce a conceptually-simple combinatorial optimization problem, i.e., the Travelling Salesman Problem (TSP), which will be utilized throughout this chapter for illustrating features of the SLS approaches. Other combinatorial problems used in this study will be presented in the next two chapters where they have been first employed.

The TSP is probably the most well-studied classical NP-hard combinatorial optimization problem. It has several real-world applications, such as logistics, printed circuit board assembly, and the manufacture of microchips, to name a few. The problem is defined as follows. Given a set of cities and the distance between them, the goal is to find a shortest closed tour (roundtrip) while visiting each city only once. Usually, the term TSP refers to the symmetric TSP, where the distance from city i to city j is the same as the distance from city j to city i. In the Asymmetric TSP (ATSP) in-

stances, these two distances are not the same. In this research work we only use symmetric TSP instances. A graphic representation of a TSP instance is given in Figure 2.1.

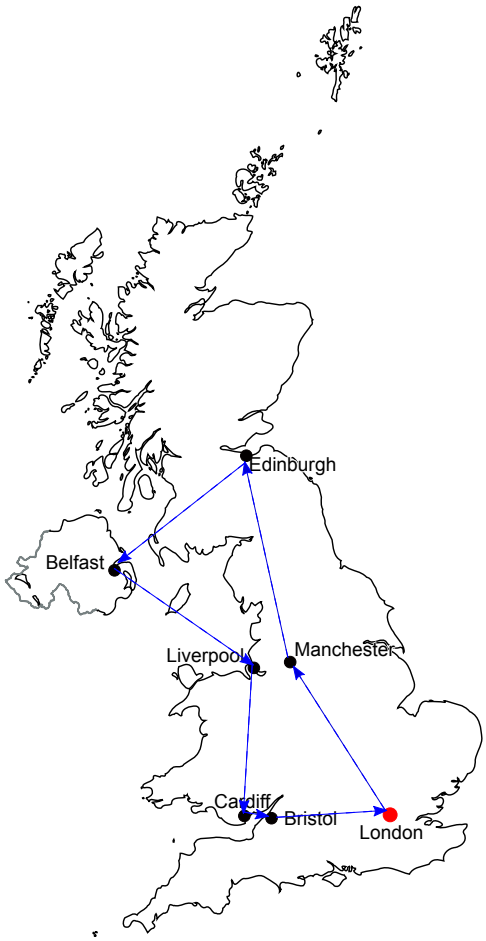


Figure 2.1: An example of Travelling Salesman Problem that contains of seven UK cities. A solution to this particular problem is indicated by the dashed line and arrows.

The most well-known TSP benchmark library is TSPLIB, introduced by Reinelt (1991), which is available at `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`. In this benchmark, the number of cities is indicated in the instance name. For example, `rat783` is an instance with 783 cities. TSPLIB contains 111 instances and Table 2.1 shows the names and sizes (number of cities) of seven of them.

A considerable amount of literature has been published on applying dif-

Table 2.1: TSP benchmark instances taken from the TSPLIB repository.

Dataset	Size
rat783	783
u1060	1060
fl1400	1400
u1817	1817
d2103	2103
pcb3038	3038
fl3795	3795

ferent search techniques to TSP. The interested reader is referred to surveys that provide a detailed description of some of these studies, e.g., Cook, Applegate, Bixby & Chvátal (2011), Gutin & Punnen (2007), Larrañaga, Kuijpers, Murga, Inza & Dizdarevic (1999), Jünger, Reinelt & Rinaldi (1995), Laporte (1992).

2.2 SLS Algorithms

There are different *exact methods* which can produce the globally *optimal (best) solution* for a given instance of a combinatorial problem, e.g., exhaustive search (Burke & Kendall 2014, p. 7), constraint programming (Jaffar & Lassez 1987), linear programming (Dantzig 1963), backtracking (Knuth 2019, subsection 7.2.2), and branch-and-bound (Land & Doig 1960) methods. These are acceptable strategies for small problems (possibly with a few constraints); however, they are intractable as problems become larger. The types of 21^{st} Century combinatorial problems that often occur in the real world tend to grow very large and very quickly with many constraints (see the example that was given at the beginning of this chapter.) Moreover, although there are methods to generate all of the possibilities in some combinatorial universe (Knuth 2014, subsection 7.2.1), it is not always the case, even in small problems (Burke & Kendall 2014, p. 7). Given this situation, the intention is to design *search algorithms* that go through the search space in an "intelligent" way to avoid sampling unpromising points; and, for a given problem, provide "a" solution —rather than "the" solution— which is good enough, cheap enough, and fast enough. These search algorithms are known as *Stochastic Local Search* (Hoos & Stützle 2005) and are also often called *metaheuristics* (Chopard & Tomassini 2018, Hoos & Stützle 2015).

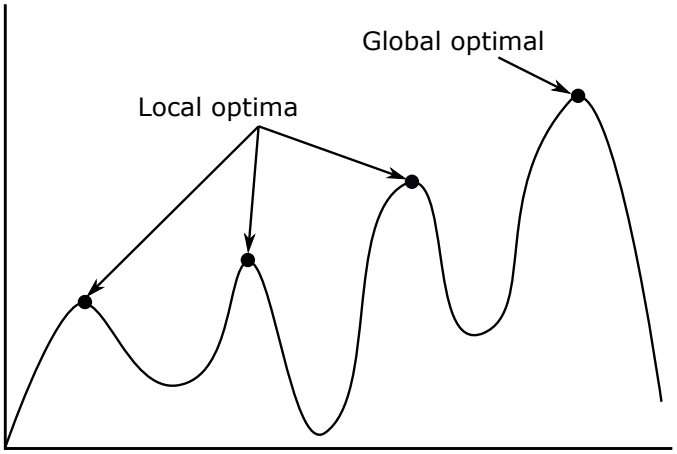


Figure 2.2: An illustration of global optimal solution and local optimum solutions in a maximization problem.

They do not guarantee to find the global optimal solution, but a promising locally *optimum solution*. Figure 2.2 illustrates the difference between a globally optimal solution and locally optimum ones.

The design of SLS algorithms mostly intends to be easy to implement and simple to understand in order to be successfully employed in tackling computationally complex problems. There are two types of SLS algorithms, i.e., one-point (a.k.a. single-solution) and population-based. *One-point SLS* algorithms maintain only one solution at each *iteration* (a.k.a. *search step*); Hill Climbing (HC) (Chopard & Tomassini 2018, p. 32) and Simulated Annealing (SA) (Kirkpatrick, Gelatt & Vecchi 1983) are examples of this kind of SLS. *Population-based SLS* methods, on the other hand, manipulate sets of candidate solutions, known as a *population*, at each *generation*; Genetic Algorithm (GA) (Goldberg 1989), and Evolutionary Strategy (ES) (Beyer & Schwefel 2002) are two well-known algorithms of this type of SLS. The fundamental idea behind these methods can be outlined as follows. They iteratively generate and evaluate a set of candidate solutions (or a population of candidate solutions) and in each iteration (generation in the case of population-based SLS), based on some criteria, they either *accept* or *reject* the candidate solution(s). The search stops when the specified *stopping criteria* are satisfied.

One-point SLS algorithms are quite popular in the field of Operational Research which is the focus of this research work; so that, we only discuss

one-point SLS in this chapter. One of their major advantages is their simplicity of use and ease of implementation. The main components of most one-point SLS are introduced in the following:

- *Solution representation*: in combinatorial problems, it is often the *permutation* of n objects. For example, in the TSP instance given in Figure 2.1, it is a permutation of 7 cities.

- *Initialization*: this specifies the starting point of the search process in the search space. It is usually generated uniformly at random (u.a.r. for short). However, there are *construction heuristics* that build an initial solution from scratch. Take the TSP instance given in Figure 2.1 as an example. One way to generate an initial solution is to start from an empty tour with only London in it and then gradually add the remaining 6 cities into it until there is nothing left to add. Each time we add the city that is the closest one to the last in the tour. The idea behind the construction heuristic is that instead of starting the search process from a random place in the search space, it is beneficial to start it from an area that has already shown a promising performance. They are often a single-pass approach and have a fast computational performance (Burke & Kendall 2014, p. 9).

- *Neighbourhood structure*: this is related to a *move* operator (Hoos & Stützle 2005, p. 44) and induces the set of *neighbours* of the current solution —as potential replacements. Different move operators for combinatorial problems are introduced in the literature which form the basis for many successful applications of SLS; however, there is no principled way of choosing it since it strongly depends on the problem at hand.

  One of the most widely used move operator is the *2-exchange move*. In TSP, it operates as follows: two non-adjacent edges are removed from a tour creating two sub-tours and subsequently one of the sub-tours is reconnected in reverse order. Figure 2.3 illustrates two tours that are neighbours under the 2-exchange move.

  It also worth mentioning that a finite sequence of successors of the current position, generated by the move operator, are around the *search trajectory* of the problem at hand. It also determines the neighbourhood size of the current solution, and obviously a large neighbourhood offers more choices around the search trajectory.

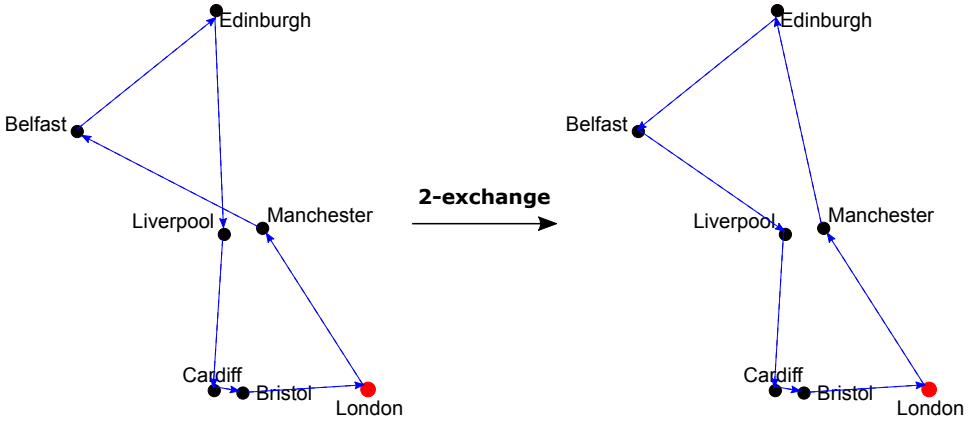


Figure 2.3: A 2-exchange move for the TSP instance of seven UK cities.

- *Evaluation functions*: this measures the quality of candidate solutions and guides the search process. The term "*objective functions*" is also used. There is often no distinction made between an objective function and an evaluation function in the literature. However, some researchers in the SLS community distinguish between these two terms (Hoos & Stützle 2005, p. 46). In their distinction, a combinatorial optimization problem includes an objective function that needs to be optimized, while a SLS algorithm for addressing this problem might make use of an evaluation function.

  An evaluation function somehow depends on search space, neighbourhood, and constraints. In the case of the latter, combinatorial problems usually have constraints that need to be satisfied and are normally formulated in an objective function. There are two classes of constraints: namely, *hard constraints* and *soft constraints*. A hard constraint must be satisfied, and violation of it leads to an *infeasible* candidate solution. On the other hand, a soft constraint is a condition that we would like to satisfy, but at the same time, it is not mandatory. Some applications of search approaches simply ignore infeasible candidate solutions throughout the search process and do not include them in the evaluation function. However, in some other approaches, a very high penalty value is considered for violation of hard constraints. As for soft constraints, they are mostly included in the evaluation function by a summation of the penalty values for them.

- *Stopping criterion*: this decides when to terminate the search process. It

is also known as *termination condition* or *cutoff time*. They are generally classified into two types: static and dynamic (Hoos & Stützle 2005, p. 194). The static stopping conditions do not rely on the state of the search, e.g., total CPU time elapsed or the total number of function evaluations. The dynamic stopping conditions depend on the search state, e.g., objective function value of the current solution or the total number of iterations without improving the current solution. All existing stopping conditions in the literature are empirical; however, recently a new *cutoff* time based on a solid mathematical foundation is introduced by Lobo, Bazargani & Burke (2020) and it is presented in Chapter 4 of this thesis.

After introducing the main components of most one-point SLS, in the following section we are going to give examples of six widely used one-point SLS algorithms, i.e., Iterative Improvement, Simulated Annealing, Tabu Search, Threshold Accepting, Great Deluge Algorithm, and Iterated Local Search. Thereafter, we will explain in detail the Late Acceptance Hill-Climbing which is the pillar of this research work.

Iterative Improvement

Iterative Improvement (II) is the most basic and simplest SLS algorithms. It starts from an initial solution in the search space (usually generated u.a.r.), and then iteratively tries to improve the quality of the current solution with respect to the evaluation function of the given problem. Iterative improvement is also known as *iterative descent* or *hill-climbing*. The term hill-climbing is traditionally used for applications of iterative improvement to maximization problems.

There are different variations of iterative improvement based on the way they choose the next point to explore in the search space. Four well-known approaches are *iterative best improvement*, *iterative first improvement*, *randomized iterative improvement*, and *probabilistic iterative improvement* (Chopard & Tomassini 2018, p. 32). All these methods check neighbours of the current solution in a systematic order. The iterative best improvement (a.k.a. *steepest-ascent hill-climbing*) chooses the best neighbour of the current solution as the next search point to move to, while the iterative first improvement (a.k.a. *first-ascent hill-climbing*) chooses the first

improving neighbour as the next point (see Algorithm 2.1). Note that these two approaches are normally terminated when none of the neighbours of the current solution brings about an improvement with respect to the evaluation function. Obviously, these two approaches are greedy and often get stuck in a local optimum very quickly. To overcome this problem, randomized iterative improvement and probabilistic iterative improvement are introduced to occasionally accept candidate solutions with a solution quality worse than the current one, known as *worsening moves*.

Algorithm 2.1: Iterative first improvement.

Output: A solution to a given problem.

```
1 Produce an initial solution s;                    // Usually u.a.r.
2 Calculate its cost function value C(s);
3 while s is not a local optimum do
                              // Checks neighbours of the current
                              // solution in a systematic order.
4   Choose a neighbour s' of s such that C(s') < C(s);
5   s = s';
6 return s;
```

The key idea behind the randomized iterative improvement is to occasionally, based on a given probability, move to random neighbouring candidate solutions regardless of the solution quality. Unlike randomized iterative improvement, probabilistic iterative improvement does not accept worsening moves without considering the amount of deterioration in the evaluation function value. It uses a probability of acceptance that depends on the change of the evaluation function value incurred (Hoos & Stützle 2015). The simulated annealing, discussed next, is a special case of a probabilistic iterative move. In both algorithms, if the given probability is so set to 0%, then they effectively turn into an iterative first improvement algorithm. If the given probability is 100% in randomized iterative improvement or infinity in probabilistic iterative improvement, then they degenerate to a uniform *random walk*.

Simulated Annealing

Simulated Annealing (SA) was introduced by Kirkpatrick, Gelatt & Vecchi

(1983). The idea behind it is inspired by the physical process of metallurgy, and its terminology is taken from that field. SA always accepts *improving moves*; however, based on the parameter T, a.k.a. *temperature*, it decides to either accept or reject worsening moves. The temperature T is initially set to a high value and then progressively decreased during the search process. At high temperatures, worsening moves are more likely to be accepted, and this helps to avoid getting stuck in a local optimum. However, as the search progresses —it intends to converge towards a local optimum— the temperature is gradually decreased and as a consequence, the probability of accepting worsening moves tends to zero.

The acceptance probability, for minimization problems, is often based on the *Metropolis criterion* (Aarts, Korst & van Laarhoven 2003, p. 97) and defined as $p = \exp((C - C')/T)$, with C and C' denoting the evaluation function values of the current and candidate solutions. The fundamental aspect of SA is the fact that it progressively lowers the temperature during the search. The modification of temperature T is managed by a *cooling schedule*, also called an *annealing schedule*, and can be defined in different ways. One of the widely used cooling schedules that has shown to be quite efficient in many problems is a geometric cooling schedule (Hoos & Stützle 2005, p. 77). It requires two parameters, temperature length, k, and cooling rate, α, which must be between 0 and 1. It starts from an initial given temperature (T_0), and then, every k iterations, it updates the temperature as $T = \alpha \times T$. The standard steps of a Simulated Annealing —for a minimization problem— with a geometric cooling schedule is given in Algorithm 2.2. Note that the initial temperature of SA and the other two parameters are empirical and problem dependent.

SA is one of the most studied SLS algorithms and is well-known in the field of *Artificial Intelligence* (Russell & Norvig 2016, p. 125). In a very detailed study, Johnson, Aragon, McGeoch & Schevon (1989, 1991) concluded that one of the main reasons that SA is effective is due to accepting worsening moves.

There are many successful applications of SA for real-world combinatorial problems. It has been widely studied within the Operational Research literature and has been applied to problems in a variety of applications, including, exam timetabling (Thompson & Dowsland 1996, 1998, Burke, Bykov, Newall & Petrovic 2003, Burke, Eckersley, McCollum, Petrovic &

Algorithm 2.2: Simulated annealing with a geometric cooling schedule.

Input : Initial temperature, T_0;
Input : Cooling rate, α; `// Must be between 0 and 1`
Input : Temperature length, k.
Output: A solution to a given problem.

Produce an initial solution s; `// Usually u.a.r.`
Calculate its cost function value $C(s)$;
while *termination criterion not satisfied* **do**
    Choose a neighbour s' of s;
    if $C(s') \leq C(s)$ **then**
        $s = s'$;
        $i = i + 1$;
    else
        if *Uniform(0,1)* $< \exp((C(s) - C(s'))/T)$ **then**
            `// Metropolis condition`
            $s = s'$;
            $i = i + 1$;
    if $i == k$ **then**
        $T = \alpha \times T$; `// Cooling schedule`
        $i = 0$;
return s;

Qu 2003, Leite, Melício & Rosa 2019), job shop scheduling (Seçkiner & Kurt 2007, Ying & Lin 2020, Li, Wang, Gao, Song & Li 2020), crew scheduling (Emden-Weinert & Proksch 1999, Hanafi & Kozan 2014), vehicle routing (Breedam 1995, Chiang & Russell 1996, Wang, Mu, Zhao & Sutherland 2015), large scale aircraft trajectory planning (Chaimatanan, Delahaye & Mongeau 2014, Islami, Chaimatanan & Delahaye 2017), and packing problems (Rao & Iyengar 1994, Burke, Kendall & Whitwell 2009), to name a few. Furthermore, SA is a popular choice for optimization problems in other scientific areas. For example, in electronic engineering, it has been widely used for placement and routing problems of VLSI design (Sechen 1988, Anand, Saravanasankar & Subbaraj 2011, Shunmugathammal, Columbus & Anand 2019), and also some optimization related problems in renewable energy sources (Katsigiannis, Georgilakis & Karapidakis 2012, Garlík & Křivan 2013, Zhang, Maleki, Rosen & Liu 2018, Zhang, Wu, Maleki & Zhang 2018). In medicine, it has applications in medical image processing (Bick & Giger

1997, Sharma, Ray, Sharma, Shukla, Aggarwal & Pradhan 2009, Tubic, Zaccarin, Beaulieu & Pouliot 2001, Matsopoulos, Mouravliansky, Delibasis & Nikita 1999), the interpretation of the acid-base balance of blood in the umbilical cord of newborn infants (Garibaldi & Ifeachor 1999), and plan dosages in radiotherapy treatment (Webb 1989, Jacob, Raben, Sarkar, Grimm & Simpson 2008, Kubicky, Yeh, Lessard, Joe, Speight, Pouliot & Hsu 2008, Tinkle, Weinberg, Chen, Littell, Cunha, Sethi, Chan & Hsu 2015). In architecture, it has been used for multiple project scheduling (Chen & Shahandashti 2009), cost modelling (Alwisy, Bouferguene & Al-Hussein 2018), process modelling (Chan, Kwong & Luo 2009), and building modelling from point clouds (Yang, Xu & Dong 2013, Chen, Koc, Shi & Soibelman 2018). Examples of other fields include chemistry (Andrade, Nascimento, Mundim, Sobrinho & Malbouisson 2008, Ghosh, Sharma & Chaudhury 2020), economics (Chen & Yeh 2001, Gilli & Schumann 2011), and astronomy (Habib, Vernin, Benkhaldoun & Lanteri 2006, Chira & Plionis 2019). The above mentioned applications represent just the tip of the iceberg of many SA applications existing in the literature. A complete review of them is beyond the scope of this thesis.

Tabu Search

Tabu Search (TS) was proposed by Glover (1989). It is significantly different from search approaches that have been introduced so far, because it makes a systematic and direct use of memory to guide the search process and escape from local optima. It uses a short-term memory to keep track of those candidate solutions that have been recently visited in a *tabu list* and forbids the search to revisit them for some time. A parameter known as the *tabu tenure* indicates tabu list length and determines the duration (number of iterations) for which these restrictions should apply to candidate solutions. The core concept underlying TS is to prevent the search from immediately revisiting a point in the search space that has been previously explored (Chopard & Tomassini 2018, p. 43) and also to avoid cycling (Hoos & Stützle 2005, p. 79). In the literature, there are many successful applications of TS (Gendreau & Potvin 2013, Glover & Laguna 1997); however, its performance strongly depends on the tabu tenure setting (Hoos & Stützle 2015).

TS typically uses an iterative best improvement strategy to select the

best neighbour from *admissible neighbours* in each iteration. Once in a local optimum, this can lead to accepting a worsening move (Gendreau & Potvin 2013, p. 246). Since memorizing complete visited solutions is overwhelming, TS usually associates a tabu status with specific solution components. For example, in a UK road trip example if the last change in the current solution is visiting Manchester after London, then instead of keeping the whole solution in the tabu list, we only store the tabu position in the permutation. In this case, assuming the first position in the perturbation belongs to London, it will be stored in the tabu list. This means that for a tabu tenure iterations, we cannot replace Manchester in position two of the perturbation with any other city.

Tabus are sometimes quite restrictive, in the sense that they may avoid attractive moves or cycling even when there is no danger of it (Gendreau & Potvin 2013). To overcome this issue, TS usually employs different mechanisms to revoke tabus. These mechanisms are known as *aspiration criteria*. The most commonly used aspiration criterion is that a tabu move is allowed if its resulting candidate solution is better than the best solution so far. A general algorithm outline for TS is shown in Algorithm 2.3.

Algorithm 2.3: A general template for Tabu Search.

Input : Tabu tenure.
Output: A solution to a given problem.

```
1 Produce an initial solution s;                    // Usually u.a.r.
2 Calculate its cost function value C(s);
3 while termination criterion not satisfied do
4 |   Determine set N of admissible neighbours of s;
5 |   Choose a best improving solution s' in N;
6 |   Update tabu list based on s';
7 |   if C(s') < C(s) then
8 |   |   s = s';
9 return s;
```

Over the last three decades, since TS was proposed, hundreds of papers presenting applications of it to various combinatorial problems have appeared in the literature. TS has many applications in Operational Research. In several cases, its applications are among the most effective to tackle difficult real-world problems. For example, there are applications of it

in the nurse rousting (Burke, Causmaecker & Berghe 1999, 2004, Václavík, Šůcha & Hanzálek 2016), timetabling (Burke, Kendall & Soubeiga 2003, Amaral & Pais 2016), graph optimization (Wu, Wang & Glover 2020, Pastore, Martínez-Gavara, Napoletano, Festa & Martí 2020), assignment (Demeester, Souffriau, Causmaecker & Berghe 2010, Liu, Zhang, Zhang, Kurniawan, Juhana & Ai 2020), vehicle routing (Diabat, Abdallah & Le 2014, Alinaghian, Tirkolaee, Dezaki, Hejazi & Ding 2020), and aviation (Soykan & Rabadi 2016, Çiftçi & Özkır 2020). TS also has applications in other scientific areas, e.g., telecommunications (Xu, Chiu & Glover 1999, Lee & Kang 2000), earth science and agriculture (Zagré, Marcotte, Gamache & Guibault 2018, Kong, Kuriyan, Shah & Guo 2019), and chemistry (Cheng & Fournier 2004, Lin, Chavali, Camarda & Miller 2005).

Threshold Accepting

Threshold Accepting (TA) is a local search method that accepts worsening moves. It was first introduced by Dueck & Scheuer (1990) at IBM scientific centre in Heidelberg. On the other side of the Atlantic, a quite identical approach, around the same time, was suggested by Moscato & Fontanari (1990), where they called it a "deterministic update in SA". Both publications argued that the stochastic way of accepting worsening moves used in the SA algorithm does not play a major role in the success of the search process, but only accepting worsening moves in this algorithm does so. Therefore, they introduced a simpler approach that does not require the computation of a cooling schedule. In a theoretical study, Althöfer & Koschnick (1991) proved that TA has convergence properties that are similar to those of SA.

TA starts the search process from an initial solution (usually generated u.a.r.) Thereafter, it always accepts improving moves. In the case of the worsening moves, it only accepts them if they are not worse than a particular *threshold*, T. In other words, it accepts a candidate solution if $C' - C < T$, with C and C' denoting the evaluation function values of the current and candidate solutions. Note that the initial threshold value must be bigger than zero. The threshold value is updated over time and decreases to zero; so, towards the end of the search process, TA turns into an iterative improvement algorithm. The pseudocode of the TA is given in Algorithm 2.4.

Algorithm 2.4: A general template for Threshold Accepting.

Input : T; `// Threshold value that must be bigger than 0`
Input : I; `// Non improving iterations before updating` T
Input : ϵ; `// A parameter value for updating` T
Output: A solution to a given problem.

```
1  Produce an initial solution s;                          // Usually u.a.r.
2  Calculate its cost function value C(s);
3  i = 0;
4  while termination criterion not satisfied do
5      Construct a candidate solution s' ∈ N(s);
6      Calculate its cost function value C(s');
7      Δ = C(s') − C(s);
8      if Δ < T then
9          s = s';
10         i = 0;
11     else
12         i = i + 1;
13     if i > I then
14         T = T − ε;
15         if T < 0 then
16             T = 0;
17 return s;
```

In analogy to SA, the parameter T of the TA might be interpreted as a deterministic cooling schedule. An adaptive approach to update the threshold value was also introduced by Hu, Kahng & Tsao (1995). However, it is not clear when to update the threshold value and how much it should be decreased. Although TA refined the SA acceptance procedure to make it a simpler algorithm, it still involves a few parameters whose values are deduced empirically (Burke, Bykov, Newall & Petrovic 2003). This technique is not widely applied in comparison to SA and TS. Some of the applications of TA are as follows: vehicle routing (Tarantilis, Kiranoudis & Vassiliadis 2002, 2004), scheduling (Lee, Vassiliadis & Park 2004, Marimuthu, Ponnambalam & Jawahar 2009), assignment (Nissen & Paul 1995, Leutner, Gschwind, Liermann, Schwarz, Gemmecker & Kessler 1998), aircraft landing (Liu 2010), evaluation of the discrepancy of a set of points (Winker & Fang 1997), classification with feature selection (Ravi & Zimmermann 2000), and economics

(Winker 2000).

Great Deluge Algorithm

The Great Deluge Algorithm (GDA) is another generic one-point SLS algorithm that was introduced by Dueck (1993). The algorithm is inspired by the idea of a Great Deluge event (a.k.a. great flood). In the Great Deluge analogy, a person climbing a hill moves around, in any direction, as the water level rises to avoid getting his/her feet wet. Finally, the person "gets wet feet" when a local optimum is reached. The water level is known as the *upper limit* in the GDA.

The outline of the GDA, based on what Dueck (1993) suggested, is given in Algorithm 2.5. The algorithm starts from an initial solution that is usually generated u.a.r. Thereafter, it always accepts any move (candidate solution, s') that is not worse than the upper limit, B. Note that the initial upper limit value must be bigger than the cost of the initial solution. The upper limit value is updated over time and decrease whenever a candidate solution is accepted by a decay rate of ϵ. The algorithm terminates when stopping criterion is satisfied.

Algorithm 2.5: A general template for Great Deluge Algorithm.

Input : B; `// Initial upper limit, must be bigger than 0`
Input : ϵ; `// A parameter value for updating` B
Output: A solution to a given problem.

```
1 Produce an initial solution s;                    // Usually u.a.r.
2 Calculate its cost function value C(s);
3 while termination criterion not satisfied do
4 |   Construct a candidate solution s' ∈ N(s);
5 |   if C(s') < B then
6 |   |   s = s';
7 |   |   B = B − ε;
8 return s;
```

Burke, Bykov, Newall & Petrovic (2003) extended GDA to prevent a premature convergence by accepting all the candidate solutions which are better than the current one. This encourages current solutions to return into the feasible region. In their approach, they also initialize the upper

limit, B, with the initial cost function value. This prevents sharp descents and idle steps in the beginning of the search process. Hence, the extended GDA has only one parameter, ϵ, which is the decay rate of the upper limit, B, at each iteration. They explained that it is relatively easy to specify the only parameter of the extended version of the GDA (Burke, Bykov, Newall & Petrovic 2003, 2004).

The Flex-Deluge algorithm is another extension of the GDA that is introduced by Burke & Bykov (2006, 2016). Accepting worsening moves in this version depends on a "flexibility" coefficient, k_f ($0 \leq k_f \leq 1$), which is formulated in the two following acceptance rules: 1) $C(s') \leq C(s) + k_f \times (B - C(s))$, when $C(s) < B$; 2) $C(s') \leq C(s)$, when $C(s) \geq B$. According to the first part of this formulation, the increase in the accepted penalty for the candidate solution should not be greater than the difference between $C(s)$ and B multiplied by k_f. This modification is based on the observation that the quality of results can be improved by slowing down the acceptance of worsening moves (a.k.a. "the uphill motion" in the minimization) of a prospective search (Burke & Bykov 2016). Note that slowing down the acceptance of worsening moves increases diversification in the search process which eventually increases the possibility of escaping from local optima. The second part of the above mentioned formulation, which is the same as the extended version of the GDA, makes sure that once the upper limit surpasses the current solution cost value, any candidate solution with a solution cost better than that of the current one is accepted regardless of the upper limit. Obviously, the Flex-Deluge algorithm degenerates to an iterative improvement by setting $k_f = 0$, and converts to the original GDA by $k_f = 1$. Burke & Bykov (2016) successfully applied this method to university exam timetabling and showed that their approach improves the solution quality of 20 instances (out of 28) compared to previously published methodologies.

The GDA has been applied to a number of real-world problems. It has been particularly investigated and adapted for timetabling problems (Bykov 2003, Burke, Bykov, Newall & Petrovic 2004, 2003, Burke & Bykov 2006, 2016, Mohmad Kahar & Kendall 2015, Landa-Silva & Obit 2009, McMullan 2007, Abdullah, Turabieh, McCollum & McMullan 2010, McCollum, McMullan, Parkes, Burke & Abdullah 2009). There are also applications of it in other scheduling problems (McMullan & McCollum 2007, Abdullah, Aickelin, Burke, Din & Qu 2007, Eng, Muhammed, Mohamed & Hasan

2020), vehicle routing (Yassen, Ayob, Nazri & Sabar 2017), attribute reduction (Mafarja & Abdullah 2014), engineering design (Dhouib 2010), training fuzzy cognitive maps (Baykasoglu, Durmusoglu & Kaplanoglu 2011), preventive maintenance optimization (Nahas, Khatab, Ait-Kadi & Nourelfath 2008), channel assignment in cellular communication (Kendall & Mohamad 2004), facility layout (Nourelfath, Nahas & Montreuil 2007), and patient-admission (Kifah & Abdullah 2015).

Several multiobjective formulations of GDA have also been explored (Petrovic & Bykov 2003, Bykov 2003, Acan & Ünveren 2020).

Iterated Local Search

The search methodologies that we have discussed so far are classified as simple SLS (Hoos & Stützle 2015). Iterated Local Search (ILS) (Lourenço, Martin & Stützle 2019), in contrast, is a hybrid SLS, since it combines different types of search steps (Hoos & Stützle 2005, p. 85). The general idea behind ILS is as follows. It starts the search from an initial solution, and then applies a local search algorithm to it to obtain a locally optimal solution. Thereafter, it iteratively carries out the following three steps until it satisfies the termination criterion. First, a *perturbation operator* is applied to the current solution, s, which yields a modified candidate solution, s'. This step aims to change the search area from the current basin of attraction that has been already explored to a new one. Second, a local search is performed on s', the resulting local optimum is saved then again in s'. In the last step, an acceptance criterion is used to decide from which of the two local optima, s or s', the search process is continued. A pseudo-code of a high-level outline of ILS is given in Algorithm 2.6.

The local search algorithm often used in ILS is based on iterative first improvement which stops in a local optimum. ILS has a simple principle. It first carries out *intensification* by means of a local search algorithm, and then, once in a local optimum, it undertakes *diversification* by using a perturbation operator to escape the local optimum. Note that in the literature, the terms perturbation operator and move operator are sometimes used interchangeably. However, in the context of ILS and some other SLS, a perturbation operator refers to the partial destruction of the given solution in a random way and not necessarily guided by an evaluation func-

Algorithm 2.6: A high-level outline of Iterated Local Search.

Input : Perturbation operator parameter(s).
Output: A solution to a given problem.

```
1 Produce an initial solution s;                    // Usually u.a.r.
2 Perform a local search on s;
3 while termination criterion not satisfied do
4 |   s' = Apply a perturbation operator to s, resulting in s';
5 |   Perform a local search on s';
6 |   Based on acceptance criterion, either continue with s or accept
  |     s = s';
7 return s;
```

tion (Lourenço, Martin & Stützle 2019). It is reported in the literature that the strength of the used perturbation operator has an extensive influence on the performance of ILS (Hoos & Stützle 2005, p. 87).

A weak perturbation does not usually change the basin of attraction in the search process, and leads the subsequent local search phase to fall back into the local optimum just visited. On the other hand, a too strong perturbation resembles a random restart of the search process, and thereby, it converts ILS to a multistart hill-climbing. Although, there are reports of successful applications of multistart hill-climbing in the literature (Martí, Aceves, León, Moreno-Vega & Duarte 2018, Lobo & Bazargani 2015, Boese, Kahng & Muddu 1994), perturbation of ILS aims to benefit further by employing obtained knowledge, from previously visited promising results, for a guided restart.

ILS has been applied to a variety of benchmark problems, including TSP, QAP, and PFSP. It has been also successfully applied to a number of real-world combinatorial problems. Some of the applications of ILS are vehicle routing (Penna, Subramanian & Ochi 2011, Silva, Subramanian & Ochi 2015), scheduling (Maenhout & Vanhoucke 2010, Hachemi, Gendreau & Rousseau 2013, Soria-Alcaraz, Özcan, Swan, Kendall & Carpio 2016, Song, Liu, Tang, Peng & Chen 2018), image processing (Cordón, Damas & Bardinet 2003, Cordón & Damas 2006), design of water distribution networks (Corte & Sörensen 2016, Martinho, Melo & Sörensen 2020), car sequencing (Cordeau, Laporte & Pasin 2008, Ribeiro, Aloise, Noronha, Rocha & Urrutia 2008), and many others. Interested readers can find some of these

applications in (Lourenço, Martin & Stützle 2019).

2.2.1 Late Acceptance Hill-Climbing (LAHC)

The Late Acceptance Hill-Climbing (LAHC) was introduced by Burke & Bykov (2017, 2008). It was shown to be competitive (and often superior) to other algorithms of its kind, such as SA, TA, and the GDA, when applied to Travelling Salesman and Exam Timetabling benchmark problems.

One of the major advantages of LAHC compared with other algorithms of its kind, such as SA, TA, and GDA, is its simplicity of use. LAHC has a single parameter, the *history length*, whose meaning appears to be well understood and directly related to runtime execution and solution cost (Burke & Bykov 2017, Bazargani & Lobo 2017). Specifically, the longer the history length is, the longer it takes to reach a good quality solution; but at the same time a better solution quality can be expected at the end of the search process. Figure 2.4 shows the above mentioned tradeoff for the `u1817` instance of TSP. For the other instances, the pattern observed is similar.

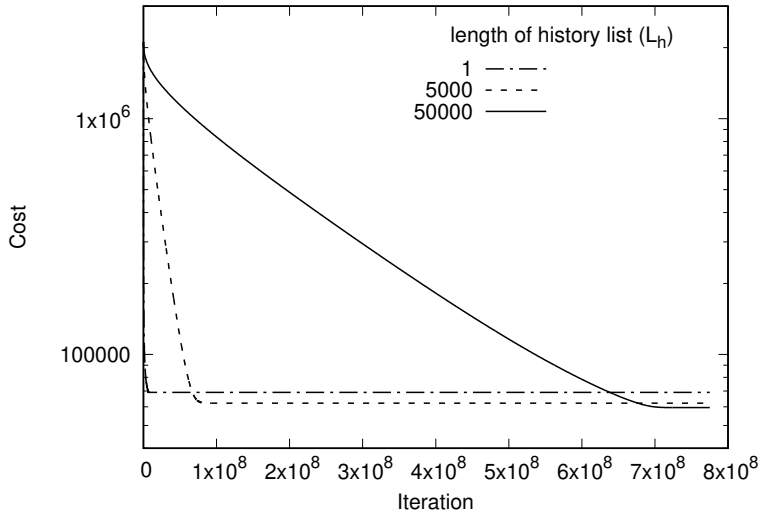


Figure 2.4: Tradeoff between solution quality and number of iterations, for the `u1817` instance taken from TSPLIB. Plot obtained with averaged data collected from 100 independent runs. Solution quality (cost) is shown in log scale.

Similarly to the above mentioned algorithms, the LAHC algorithm can also accept worsening moves. However, unlike SA, TA, and GDA, it does

not require a cooling schedule. Instead, the algorithm uses a list to *memorize* previous values of the current solution's cost. The size of the list, denoted as L_h, is the sole parameter of the algorithm. The basic idea of LAHC is that a candidate solution is compared with a solution which was current several iterations before, more precisely, L_h iterations before. This contrasts to iterative improvement algorithms (traditional hillclimber) or a (1+1)-*Evolutionary Algorithm* (EA) where the candidate solution is compared with the current solution of the immediate previous iteration. It is straightforward to see that LAHC degenerates into an iterative first improvement by setting $L_h = 1$.

Note that the list contains the solution cost function values only, not the solutions themselves. Again, this contrasts to other search algorithms that memorize previously visited solutions, such as Tabu Search and all sorts of evolutionary algorithms, where the list is in effect a population of visited solutions.

The idea of *late acceptance* can be implemented with minor variations and extensions. Building upon the work of Burke & Bykov (2017), we settle on their final version whose pseudocode for a minimization problem is reproduced as Algorithm 2.7.

In LAHC, an initial current solution is generated, usually u.a.r., and all the L_h elements of the history list are initialized with the same value: the cost of the current solution. The algorithm combines the late acceptance idea with the greedy rule of always accepting non-worsening moves as done with SA, i.e., it accepts a candidate solution if its cost is better than the cost value stored in the history list L_h iterations before or if it is not worse than the current solution of the immediate previous iteration. Another extension proposed by Burke & Bykov (2017) is to update the history list with better values only and exclude any updating with worse values. The updates on the list are constant-time operations due to a circular list implementation (Line 9 of Algorithm 2.7); therefore, not requiring actual removal and element shifting.

The stopping criterion is not specified in the pseudocode shown in Algorithm 2.7. In the context of search, the stopping criterion is usually not considered to be a parameter, but rather a decision that the user of the algorithm has to make to decide when the algorithm should stop.

It is also worth mentioning that LAHC has been successfully used in

Algorithm 2.7: Late Acceptance Hill-Climbing (LAHC).

Input : The history length, L_h.
Output: A solution to a given problem.

```
1  Produce an initial solution s                          // Usually u.a.r.
2  Calculate its cost function value C(s)
3  forall k ∈ {0 ... L_h−1} do
4      f_k = C(s)                                  // Initialize history list
5  I = 0                                                  // Iteration counter
6  while stopping criterion is not fulfilled do
7      Construct a candidate solution s' ∈ N(s)
8      Calculate its cost function value C(s')
9      v = I mod L_h                                      // Virtual beginning
10     if C(s') < f_v or C(s') ≤ C(s) then
11         s = s'                                          // Accept candidate
12     if C(s) < f_v then
13         f_v = C(s)                              // Update the fitness array
14     I = I + 1;
15 return s
```

several competitions (Burke & Bykov 2017). In 2011, it was the winner of the International Optimisation Competition organized by SolveIT Software Pty Ltd, an Australian based company. The goal of the competition was to develop an application to solve the largest constrained Magic Square problem within one minute of run time. It has also been incorporated in real-world software systems, an example being the constraint satisfaction solver OptaPlanner, an open source project by Red Hat (http://www.optaplanner.org/).

A Brief Analysis of LAHC

In spite of its simplicity, there are several facts that may go unnoticed regarding the behaviour of the LAHC algorithm. Herein, we reason about the pseudocode shown before as Algorithm 2.7 and highlight statements that are true regardless of the problem instance in which the algorithm is applied.

1. **The history list values monotonically decrease.** The history list values are initialized in lines 3-4 with $C(s)$, the cost value of the

initial solution s. Subsequently the sole place where the history list is updated is in line 13, and that line of the code is only executed if the cost of the current solution $C(s)$ strictly improves upon the value f_v. Thus, the history list values either decrease or remain the same as times goes by.

2. **Acceptance criteria.** A candidate solution s' is accepted if and only if at least one of the following three conditions occur (see line 10 of the pseudocode):

   (i) $C(s') < f_v$. We call this the *late acceptance* condition.

   (ii) $C(s') < C(s)$. We call this the *hill-climbing* condition.

   (iii) $C(s') = C(s)$. We call this the *plateau* condition.

   The last two conditions are mutually exclusive, but any other two are not.

3. f_v **is an upper bound for the cost value of the solution which was current** L_h **iterations before.** Note that it is possible that a candidate solution is not accepted, yet the history list value f_v still gets updated. The position v changes at each iteration, and as long as $C(s) < f_v$, f_v is updated. In other words, the value f_v is guaranteed to be less than, or equal to, the cost value of the solution which was current L_h iterations before.

The three items listed above are important to understand the behaviour of LAHC. In particular, the *state of the algorithm* is given by the current solution, its cost value, and the history list values. There are two important remarks to be made:

- The state of the algorithm changes on a plateau move (see item 2 (iii) above), that is, the current solution changes, even though no solution cost improvement results from such a state change.

- The state of the algorithm can change even without accepting a candidate solution, as shown above in item 3. As a corollary of that same item, if the cost of the current solution does not change for L_h consecutive iterations, then all history list values are less than, or equal to the cost of the current solution.

Applications of LAHC

LAHC has been successfully applied as a move acceptance mechanism within methodologies for a wide variety of problem domains. Burke & Bykov (2008, 2017) originally demonstrated its effectiveness using the classic travelling salesman problem and exam timetabling benchmarks. Their work on exam timetabling was extended by Ozcan, Bykov, Birben & Burke (2009), who used the Late Acceptance as a move acceptance mechanism in selection hyper-heuristics (see Section 2.3 for details on hyper-heuristics). Demeester, Bilgin, Causmaecker & Berghe (2011) also used a tournament-based hyper-heuristic approach consisting of (among other things) the Late Acceptance strategy as a move acceptance criterion. They made improvements on the best-known results, for seven out of thirteen instances, of the Toronto examination timetabling benchmarks (Carter, Laporte & Lee 1996). Ahmed, Mumford & Kheiri (2019) used LAHC along with SA and GDA as the non-deterministic move acceptance methods in their hyper-heuristic approach to tackle an urban transit route design problem. An application of a hyper-heuristic based local search that uses LAHC with other four local search algorithms, including SA, ILS, GDA, and iterative improvement, was introduced by Turky, Sabar, Dunstall & Song (2018) to tackle the bin packing problem and the Google machine reassignment problem. A choice function - Late Acceptance strategy hyper-heuristic was shown to be the best of nine selection hyper-heuristics for the multidimensional Knapsack problem by Drake, Özcan & Burke (2016). A Late Acceptance selection hyper-heuristic was used by Abdulaziz, Elnahas, Daffalla, Noureldien, Kheiri & Ozcan (2018) for wind farm layout optimisation, and by Kheiri & Özcan (2013) for the magic squares problem.

Vancroonenburg & Wauters (2013) extended the LAHC algorithm to multi-objective optimization. Zhou & Kang (2018) incorporated the LAHC in a multi-objective hybrid imperialist competitive algorithm to minimize the line efficiency and the weighted total relevant costs per unit in the multirobot cooperative assembly line balancing problems.

The local search algorithm that was developed by Wauters, Toffolo, Christiaens & Malderen (2015) and Toffolo, Christiaens, Malderen, Wauters & Berghe (2018) used the LAHC algorithm as the local search component of the Iterated Local Search algorithm. It was the winner of the VeRoLog

Challenge 2014 (Heid, Hasle, & Vigo 2014). The same approach was used by Dang & Causmaecker (2016) to develop a systematic method to characterize each neighbourhood's behaviours, and using cluster analysis to form similar groups of neighbourhoods. Turky, Sabar, Sattar & Song (2016) proposed a parallel LAHC algorithm for the Google Machine Reassignment problem that was introduced in ROADEF/EURO challenge 2012. The approach outperforms or at least was comparable to the state-of-the-art methods from the literature. LAHC was also used for the first time in genetic programming problems by McDermott & Nicolau (2017).

Other application areas include lock scheduling (Verstichel & Berghe 2009), high-school timetabling (Fonseca, Santos & Carrano 2016), vehicle routing (Sartori & Buriol 2018), a job scheduling strategy to improve big data processing in geo-distributed contexts (Cavallo, Modica, Polito & Tomarchio 2017*a*), multi-job Hadoop scheduling (Cavallo, Modica, Polito & Tomarchio 2017*b*), general lot sizing and scheduling with constraints (Goerler, Lalla-Ruiz & Voß 2020), identifying critical nodes of weighted graphs (Zhou, Wang, Jin & Fu 2021), the constrained covering arrays problem (Bazargani, Drake & Burke 2018), patient admission scheduling problem (Bolaji, Bamigbola & Shola 2018), 2D and 3D strip packing problems (Wauters, Verstichel & Berghe 2013), the travelling purchaser problem (Goerler, Schulte & Voß 2013), optimizing office-space allocation (Bolaji, Michael & Shola 2018), and finding fatigue damage in railway bridges (Frøseth & Rönnquist 2019).

2.3 Hyper-heuristics

Hyper-heuristics are not the direct subject of this thesis. However, since we have discussed some literature on the applications of hyper-heuristic based local search using SLS algorithms in this thesis, this section briefly dicsusses the area. Moreover, there is significant potential for the work described in this thesis to impact upon future research in hyper-heuristics. This is briefly discussed in Section 6.2.

Hyper-heuristics are another class of effective search approaches that have been applied to a wide variety of complex real-world problems. The main difference between metaheuristics and hyper-heuristics is that while most applications of metaheuristics are designed and tuned to a specific

problem, a hyper-heuristic is "an automated methodology for selecting or generating heuristics to solve hard computational search problems" (Burke, Curtois, Hyde, Kendall, Ochoa, Petrovic, Vazquez-Rodriguez & Gendreau 2010). While most implementations of metaheuristics explore a search space of solutions to a given problem, hyper-heuristics search through a search space of heuristics or heuristic components. Of course, there is nothing to prevent the use of a metaheuristic as a hyper-heuristic (Bilgin, Demeester, Misir, Vancroonenburg & Berghe 2011, Grobler, Engelbrecht, Kendall & Yadavalli 2015, Damaševičius & Woźniak 2017).

Figure 2.5 depicts an overview of the hyper-heuristic categorisation given by Burke, Curtois, Hyde, Kendall, Ochoa, Petrovic, Vazquez-Rodriguez & Gendreau (2010). Based on the nature of the heuristic search space and the source of feedback during learning, Burke, Curtois, Hyde, Kendall, Ochoa, Petrovic, Vazquez-Rodriguez & Gendreau (2010) proposed a classification for hyper-heuristics containing two classes, namely *heuristic selection* and *heuristic generation*. Heuristic selection includes methodologies that decide which heuristic from a set of existing heuristics should be applied at any time in the search process. There are several techniques that can be used for making such a decision, e.g., choice function, multi-armed bandit, and uniform random. A comprehensive survey on selection hyper-heuristics is recently published by Drake, Kheiri, Özcan & Burke (2020). The second category of hyper-heuristic contains those methodologies that create new heuristics from a set of components of existing heuristics. Many examples of this type of hyper-heuristic use genetic programming (Burke, Kendall & Whitwell 2009), a branch of evolutionary algorithms that has been also used for automatically generating computer programs (Koza 1994). Burke, Curtois, Hyde, Kendall, Ochoa, Petrovic, Vazquez-Rodriguez & Gendreau (2010) also divided methodologies within each class of hyper-heuristic to construction and perturbation. Construction approaches start a search process with an empty solution, and their goal is to gradually build a complete solution. Conversely, perturbation approaches start with a complete solution, and their aim is to iteratively improve the current solution.

Many hyper-heuristics are considered to be learning algorithms when they use some feedback information from the search process. Hyper-heuristics have two ways of learing, online learning and offline learning. The former takes place during the search process when a hyper-heuristic uses feedback

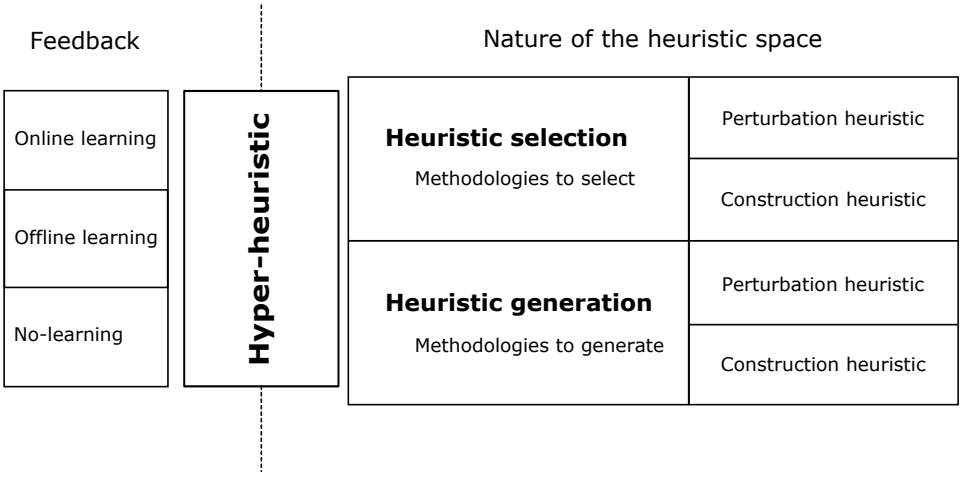


Figure 2.5: Classification of hyper-heuristic based on the nature of the heuristic search space and the source of feedback during learning. The figure is reproduced and quoted from Burke, Curtois, Hyde, Kendall, Ochoa, Petrovic, Vazquez-Rodriguez & Gendreau (2010).

it receives to constantly adapt itself throughout the search process. The latter, offline learning hyper-heuristics, gather knowledge in the form of rules or programs from a set of training instances, and then use this information to generalize a method for solving unseen instances. There are also some non-learning hyper-heuristics that do not use feedback from the search process.

2.4 Cutoff Time Strategies

It is very important to decide an appropriate moment to restart an SLS algorithm, i.e., to determine when it is stagnant. Terminating too early is detrimental because those last few changes in the current solution — upon which the search stagnates— are usually the most predominant ones in the sense that they require more effort to be discovered. Thus, restarting the search before carrying out a sufficient enough number of iterations is a barrier to utilize the stretch capacity of the algorithm. In this case, even a restarting mechanism might not be able to improve the solution cost because the algorithm ends up restarting before running those crucial iterations, every single time. Similarly, letting the algorithm run when it is stagnant is obviously a burden and a waste of time.

It is worth mentioning that the cutoff time decision still comes into play when the algorithm is given a fixed time budget to operate, because even in those cases it can be beneficial to restart the algorithm, as opposed to letting it run when it is stagnant.

In the current practice reported in the literature, the cutoff time is often determined based on empirical grounds and after a considerable amount of tuning effort. We should stress, however, that systematic tuning of the cutoff time is extremely difficult and unlikely to be successful. Any fixed cutoff time derived from previous experimental runs on different problem instances is often not adequate, not only because it requires prior experimental runs but also because the optimal cutoff time varies across problem instances and problem classes. Although several computational methods for offline tuning of metaheuristics have been gaining popularity, such as Iterated Racing (irace) (López-Ibáñez, Dubois-Lacoste, Cáceres, Birattari & Stützle 2016) and Sequential Parameter Optimization (SPO) (Bartz-Beielstein, Lasarczyk & Preuss 2005), they are computationally demanding. Tuning has its strengths in the absence of a sound method to specify a parameter value for an algorithm, but if a sound method exists there is no reason not to use it.

An alternative and more practical approach to tackle the cutoff time problem is to use a dynamic strategy that bases its decision on the search progress. A common approach is to cutoff the search once a certain amount of time θ has elapsed without improvement on the solution cost. Such time θ is usually measured in the number of iteration steps, and typically depends on properties of the problem instance, most notably, instance size (Hoos & Stützle 2005). For example, Hoos & Stützle (2005) report a dynamic cutoff time strategy for iterated local search applied to TSP that uses $\theta = m$ with m being the number of cities of the TSP instance. Another common approach is to determine the cutoff time once the algorithm does not improve the solution cost for a sufficiently long period, specified as a fraction of the total search time elapsed so far, and allowing the algorithm to perform a certain minimum number of iterations (Burke & Bykov 2017).

There is no guarantee, however, that these mechanisms are effective because nothing prevents the cutoff time from being too short or too long for the given problem instance, resulting either in not fulfilling the stretch capacity of the algorithm or a waste of computational resources.

2.5 Parameter Tuning

Most SLS algorithms have parameters that are also known as *guiding parameters* (Chopard & Tomassini 2018, p. 66). The performance of SLS significantly depends on the choice of appropriate values for those parameters, e.g., initial temperature, cooling schedule, and temperature length of SA, tabu tenure of TS, as well as, history list length of LAHC. They are problem dependent (in most cases, instance dependent) and hence there is no principled way to satisfactorily set their values for all problems. They are usually set based on some rules-of-thumb. Note that we do not use parameter tuning in this research work; on the contrary, we try to design approaches that have fewer parameters. That being said, it is absolutely important to understand parameter tuning and its drawbacks since they are the incentive behind this research work.

The main challenge in parameter tuning for SLS is how to find the optimal choices for those parameters. This is an optimization problem within the search process of the original problem (Kramer 2017, p. 21). Besides, the tuning task of some parameters has a dynamic nature, since the optimal choice varies throughout the search process. An adaptive perturbation for ILS is an example of this (Lourenço, Martin & Stützle 2019, p. 141). Another major drawback of tuning is that it imposes a significant increase in the total amount of time required for solving a problem. Moreover, because of the tuning issue of SLS, end users need to have a good enough knowledge of these methods to implement them successfully in practice, something that many users do not have. Ideally, the fewer parameters an algorithm has, the easier it should be to use from a practical point of view.

There are a number of research papers on parameter tuning of SLS algorithms in the literature. Eiben, Hinterding & Michalewicz (1999) presented a survey on controlling values of various parameters of an evolutionary algorithm, and Eiben & Smit (2011) introduced a conceptual framework for parameter tuning. Moreover, there have been some theoretical analysis on this subject in the literature (Doerr & Doerr 2015, Doerr & Wagner 2018, Doerr & Doerr 2019, Dang & Doerr 2019).

In recent years, several computational methods for offline tuning of SLS have been introduced in the literature and have gained popularity, e.g., Iterated Racing (irace) (López-Ibáñez, Dubois-Lacoste, Cáceres, Birattari

& Stützle 2016) and Sequential Parameter Optimization (SPO) (Bartz-Beielstein, Lasarczyk & Preuss 2005). These approaches, in an automatic way, try to find the most appropriate settings of an algorithm given a set of instances of a problem. Tuning is useful for algorithms that have several parameters whose interaction is not well understood by a practitioner (or even by an expert). However, if a search algorithm can be designed in a way that is minimalist in terms of parameters, and moreover those parameters are well understood in terms of their effect in the algorithms' performance, then a rational strategy for automating the setting of those parameters values may be obtained.

2.6 Search Space Structure

The behaviour and performance of SLS algorithms also depends on *search space structure*. This is known as the *fitness landscape* in the context of evolutionary algorithms (a.k.a. population-based SLS). Understanding search space structure helps in studying the behaviour of SLS and by that means improving applications of SLS methods or designing new ones. The deep analysis of search space structure is beyond the scope of this study and here we only briefly introduce some main structural aspects of the spaces being searched by SLS algorithms. There are some detailed studies in the literature on this subject that the interested reader can consult, e.g., chapter 5 of Hoos & Stützle (2005) and Reeves (2013).

A neighbourhood structure of a combinatorial problem samples points from its search space and together with the evaluation function captures the *search landscape* of a given problem. One should note that the concept of the search landscape abstracts from details of the actual search process. In the literature, *fitness-distance analysis* (FDA), introduced by Jones & Forrest (1995), is widely used to analysis and characterise search landscapes. It computes the correlation between the evaluation function value of candidate solutions and their distance to the closest optimum solution. Divorced from details of the search space structure, three concepts of search landscapes, i.e., *distribution of local optima*, *landscape ruggedness*, and *plateaus*, have influenced this research work (to be presented in the later chapters), in both aspects: designing proposed approaches as well as analysing their behaviour.

The distribution of optimum solutions across the search landscape has

an impact on designing a successful SLS application. If solutions are evenly distributed through the entire landscape, then the starting point of the search is not as influential as when they are not distributed evenly (Hoos & Stützle 2005, p. 209). However, if solutions are clustered in different regions of the landscape and far from each other, then the restarting mechanism used in some SLS is beneficial (Schiavinotto & Stützle 2004).

Unlike some problems with *smooth landscape*, intuitively, most hard combinatorial problems have a *rugged landscape* (Hoos & Stützle 2005, p. 228). In these problems, the evaluation function value of a candidate solution is weakly correlated with its direct neighbours. It is clear that the concept of landscape ruggedness is related to the number of local optima. Intuitively, problems with smooth landscape have fewer local optima, and they allow iterative improvement algorithms to explore larger parts of their search space and thus results in high quality solutions. On the other hand, problems with a rugged landscapes are intuitively harder to search for SLS algorithms (Reeves 2013), and accepting worsening moves is an essential for providing a quality solution in these problems.

Regions of neighbouring candidate solutions with identical evaluation function values are known as plateaus (Whitley, Sutton & Howe 2008). There are two different types of plateaus with respect to the existence of exits to a better quality solution, i.e., *open plateaus* and *closed plateaus*. In open plateaus, there exists solutions in the plateau that have neighbours with a better solution quality. In contrast, closed plateaus do not have access to solution with better solution quality than the current solution. Once search algorithms are in a plateau then there is no difference between SLS algorithms and the random walk. Understanding plateaus and also methods to circumvent them are essential for finding efficient methods for hard combinatorial problems (Reeves 2013). There are some methods in the literature to tackle this problem (Frank, Cheeseman & Stutz 1997).

2.7 Summary

At the beginning of this chapter, by giving an example related to one of the consequences of the COVID-19 crisis, we explained the importance of *Stochastic Local Search* (SLS) algorithms in our daily life. Then we briefly introduced *combinatorial problems* and distinguished between two main types

of them, i.e., *decision* and *optimization* problems. To better explain some of the features of SLS algorithms, we introduced a prototypical combinatorial optimization problem: Travelling Salesman Problem (TSP). We explained that there are two types of search methodologies based on the number of solutions that they maintain in each *search step*, namely *one-point* SLS and *population-based* SLS. Since this research work is about a one-point SLS algorithm, in this chapter we only focus on this type. Next, we discussed various properties of search methodologies and highlighted main components of SLS, i.e., *solution representation*, *initialization*, *neighbourhood structure*, *evaluation function*, and *stopping criterion*.

Building on the aforementioned search paradigms and properties, we introduced and discussed a number of simple one-point SLS strategies such as two variants of *Iterative Improvement*. Those are *iterative first improvement* and *iterative best improvement* that form the basis of many complex SLS methods. Overall, the main drawback with simple iterative improvement algorithms is that they get quickly stuck in local optima and they do not have any mechanism to escape it. To overcome this problem, many SLS algorithms accept *worsening moves*. Two variations of iterative improvements that accept worsening moves, i.e., *randomized iterative improvement* and *probabilistic iterative improvement*, were introduced. Thereafter, five well-studied SLS algorithms that use different techniques to accept worsening moves were explained. The *Simulated Annealing* (SA) algorithm employs a *cooling schedule* for accepting worsening moves, while *Tabu Search* (TS) utilizes memory to guide the search process and accept worsening moves. Unlike SA that accepts worsening moves based on a stochastic mechanism, *Threshold Accepting* (TA) and *Great Deluge Algorithm* (GDA) employ a deterministic method (deterministic annealing) for this purpose. Furthermore, *Iterative Local Search* (ILS) uses a local search algorithm and once it gets stuck in a local optimum, it applies a *perturbation operator* to start the search from hopefully another *basin of attraction*. We then introduced the *Late Acceptance Hill-Climbing* (LAHC) and provide a brief analysis and applications of this algorithm, since it is the main focus of this research work. We next explained *hyper-heuristics* and their variations and explained that the *selection hyper-heuristics* use local search algorithms as their acceptance mechanism.

Finally, three theoretical aspects of SLS algorithms, *cutoff time strate-*

gies, *parameter tuning*, and *search space structure*, were explained. We discussed search stagnation in the context of SLS algorithms and described some cutoff time strategies that are used in the literature. In this research work, we do not use parameter tuning. In contrast, we try to design methods that have fewer parameters and are suitable for general practitioners. However, understanding them highlights the importance of this research work. We finished this chapter by discussing search space features and properties. We explained that the structure of a search space has an important impact on the behaviour and performance of SLS algorithms.

Science may be described as the art of systematic over-simplification.

Karl Popper

Chapter 3

Parameter-less Late Acceptance Hill Climbing

As we already explained in Subsection 2.2.1, LAHC has a single parameter, the history list length, whose meaning appears to be well understood and directly related to runtime execution and solution quality. Because of that, Burke & Bykov (2017) argue that it is simpler to use compared with the aforementioned algorithms. In this chapter, we go a step further and simplify, even more, the application of LAHC by eliminating that sole parameter. The new algorithm is called *parameter-less Late Acceptance Hill-Climbing* (pLAHC). The term *parameter-less* used in this algorithm is taken from the parameter-less search technique introduced in Evolutionary Algorithms (EAs), since we use a similar technique in pLAHC. In the past, the parameter-less search technique has been successfully used in automating the population size parameter in a variety of Evolutionary Algorithm (EAs). The validity of the method is shown with computational experiments on a number of instances of the Travelling Salesman Problem (TSP). We evaluate pLAHC application to a real-world Search Based Software Engineering (SBSE) problem, namely Combinatorial Interaction Testing (CIT). CIT is a cost-effective black-box sampling techniques for discovering interaction faults in highly configurable systems. For illustration purpose we will be using a well-known existing framework and will replace a SA used in this framework with pLAHC.

This chapter is based on two papers by the author, i.e., Bazargani & Lobo (2017) and Bazargani, Drake & Burke (2018). It starts by introducing

the parameter-less search scheme and establishing a connection between the history list length of LAHC and the related problem of population sizing in EAs. Building on that, Section 3.2 presents a parameter-less version of the LAHC algorithm. The validity of the method is shown with a number of computational experiments applying it to TSP benchmark instances that were also used in the work of Burke & Bykov (2017). Section 3.3 presents a refinement of the technique proposed in Section 3.2 with the purpose of speeding up the search for good solutions. In Section 3.4, we show how the proposed approach can be transferred to a real-world problem, i.e., CIT. The example is used as a case study but similar design principles may be applied to other kind of real-world problems as well. This problem, background material, the benchmark used, and obtained results are all explained in this section.

3.1 Parameter-less Search Scheme

Results reported by Burke & Bykov (2017) suggest that the history length parameter has a more meaningful interpretation from the user point of view, as it is directly related to solution quality and execution time. Specifically, the longer the history length is, the slower the algorithm becomes in reaching a good solution quality. Also, the longer the history length is, the better chance the algorithm has of reaching a better quality solution given enough time do so. The tradeoff is a logical one; to reach a better solution quality we should expect to pay a price in terms of algorithm runtime.

This tradeoff is akin to what is observed with respect to population sizing in EAs. In general, a large population has a better chance, given sufficient execution time, to reach a better quality solution than the same EA with a smaller population (Harik, Cantú-Paz, Goldberg & Miller 1999). The problem, however, is that an EA with a large population converges slower than the same EA with a smaller population. To explore this tradeoff, Harik & Lobo (1999) proposed the *parameter-less* genetic algorithm (GA), a technique that automates population sizing in an EA. The method establishes a race among exponentially sized populations, which evolve in an alternate fashion, in an attempt to reach an adequate population size. In this technique, smaller populations are preferred by allocating them more runtime (fitness evaluations) than to larger populations. However, as time goes by

the small sized populations can be eliminated when there is evidence that they will not be competitive with a larger population in terms of solution quality delivered.

The parameter-less search scheme or a form of it has been incorporated in different EAs. Smorodkina & Tauritz (2007) limited the number of parallel population racing to two, while the number of parallel population racing in the parameter-less GA is unlimited. A similar technique is used by Auger & Hansen (2005) in the context of the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES). It has also been successfully used with model-based EAs such as the Hierarchical Bayesian Optimization Algorithm (hBOA) (Pelikan & Lin 2004), and with the Gene-pool Optimal Mixing EA (GOMEA) family of algorithms (Luong, Poutré & Bosman 2015, Bosman, Luong & Thierens 2016, den Besten, Thierens & Bosman 2016).

Given that the population size parameter of an EA appears to have a similar effect (and involve similar tradeoffs) to the history length parameter of LAHC, it is likely that the methods used to automate population sizing in EAs can also be used to automate the history length parameter of the LAHC algorithm. It is precisely this observation that led us to conduct the research presented herein.

3.2 LAHC with Exponentially Increasing History List Length

To eliminate the need to specify the history list length parameter, we propose an automated restart strategy for the LAHC algorithm with an exponentially increasing history length. The strategy follows closely what was done in Auger & Hansen (2005). Initially, the history list length can be a very small number, for example $L_h = 1$. The restart is fired whenever the LAHC with a fixed history list length L_h stops as shown by Line 6 of Algorithm 2.7. Any restarting (stopping) criterion can be used by LAHC but here we follow exactly what was suggested in Burke & Bykov (2017), i.e., the algorithm halts when the number of consecutive non-improving (idle) iterations reaches 2% over the total number of iterations, and at least 100,000 iterations are performed to avoid early termination.

The overall stopping criterion for the restart strategy has to be specified in some other way. Criteria that come to mind could be for example a

total maximum number of iterations, a total maximum execution time, a certain solution quality reached, or a certain number of restarts without an improvement in the current solution at the end of each LAHC execution, to name a few. With respect to the rate of increase for the history list length we use a factor of 2, i.e., the history list length doubles on each restart. We call the resulting algorithm *parameter-less LAHC* (pLAHC).

Another form of growth schedule could be specified; for example, linear instead of exponential. However, we settle on exponential with a factor of 2 because it seems to be the most logical thing to do, following previous research by others (Harik & Lobo 1999, Auger & Hansen 2005, Pelikan & Lin 2004, Luong, Poutré & Bosman 2015, Bosman, Luong & Thierens 2016, den Besten, Thierens & Bosman 2016). Smaller factors than 2 make the search progress slow and it takes longer time to reach to the adequate history list length. In contrast, search using factors bigger than 2 are more probable to suppress the adequate history list length.

3.2.1 Experiments with LAHC alone

We start by presenting a series of experiments of the LAHC algorithm with fixed history list length to confirm the existence of the tradeoff observed by Burke & Bykov (2017). We do it for several instances of TSP that are shown in Table 2.1 and are taken from the well-known TSPLIB repository. We use the exact same instances and the exact same TSP perturbation operator used in Burke & Bykov (2017), namely the 2-exchange move where a tour is randomly divided into two parts and subsequently reconnected in reverse order, as depicted in Figure 2.3.

For every instance, LAHC is run for 100 independent times and the results are averaged. As we explained in Section 3.2, the algorithm halts when the number of consecutive non-improving (idle) iterations reaches 2% over the total number of iterations, and at least 100,000 iterations are performed. We use 3 settings for the history list length parameter (1, 5000, and 50000), replicating the experiments conducted in Burke & Bykov (2017). Table 3.1 shows the results obtained. The Cost column gives the best ever solution quality reached by the algorithm by the end of the run, averaged over the 100 runs. The Iterations column gives the number of iterations executed until the end of the run, averaged over the 100 runs.

The results confirm the observations made by Burke & Bykov. The

Table 3.1: Results for seven TSP instances produced by LAHC using three different history list lengths, $L_h \in \{1, 5000, 50000\}$. The results are averaged over 100 independent runs, and match very well with those reported by Burke & Bykov.

Dataset	$L_h = 1$		$L_h = 5000$		$L_h = 50000$	
	Cost	Iterations	Cost	Iterations	Cost	Iterations
rat783	10808	774187	9354	28375627	9105	258717906
u1060	264264	2177387	235426	43375332	229013	389063282
fl1400	22483	4055588	20732	57679069	20426	492849859
u1817	69056	8384640	62175	90558784	59502	750645922
d2103	98643	12444579	89579	112400626	86370	877424569
pcb3038	160238	27520354	150439	176549830	144211	1342738864
fl3795	33159	63452752	31147	264306010	30170	1816215196

longer the history list length, the slower the algorithm is but a better solution quality is reached.

3.2.2 Experiments with Automated Restarts

We now present experiments of the parameter-less implementation of LAHC as described at the beginning of Section 3.2, i.e., with automated restarts doubling the history length L_h on each restart; initially $L_h = 1$. We run the algorithm on the same TSP instances described earlier. Again, 100 independent runs are executed. The stopping criterion for each run is reaching the target solution quality C_x obtained by the regular LAHC with history length x averaged over 100 independent runs. The results are summarized in Table 3.2. As an example, the entry in Table 3.2 corresponding to the instance `rat783` and C_1 has the value 1116839. This value is the number of iterations needed by pLAHC to reach the solution quality of 10808 (which can be read from Table 3.1, `rat783` with $L_h = 1$) on all of the 100 independent runs.

As expected, pLAHC is slower than a tuned LAHC (see the Overhead Factors (OF) in Table 3.2); however, pLAHC needs no tuning. More importantly, pLAHC tries to escape local optima by restarting with a larger history list length when its current history list length appears to be insufficient. LAHC, on the other hand, needs to have L_h properly set; not doing so makes it unable to improve beyond a certain solution quality. For example, if LAHC is allowed to run for more iterations (ignoring the 2% idle iteration condition) it would not improve the solution quality that much more. Take, for example, the `rat783` instance. If we run LAHC on it, with $L_h = 1$, for

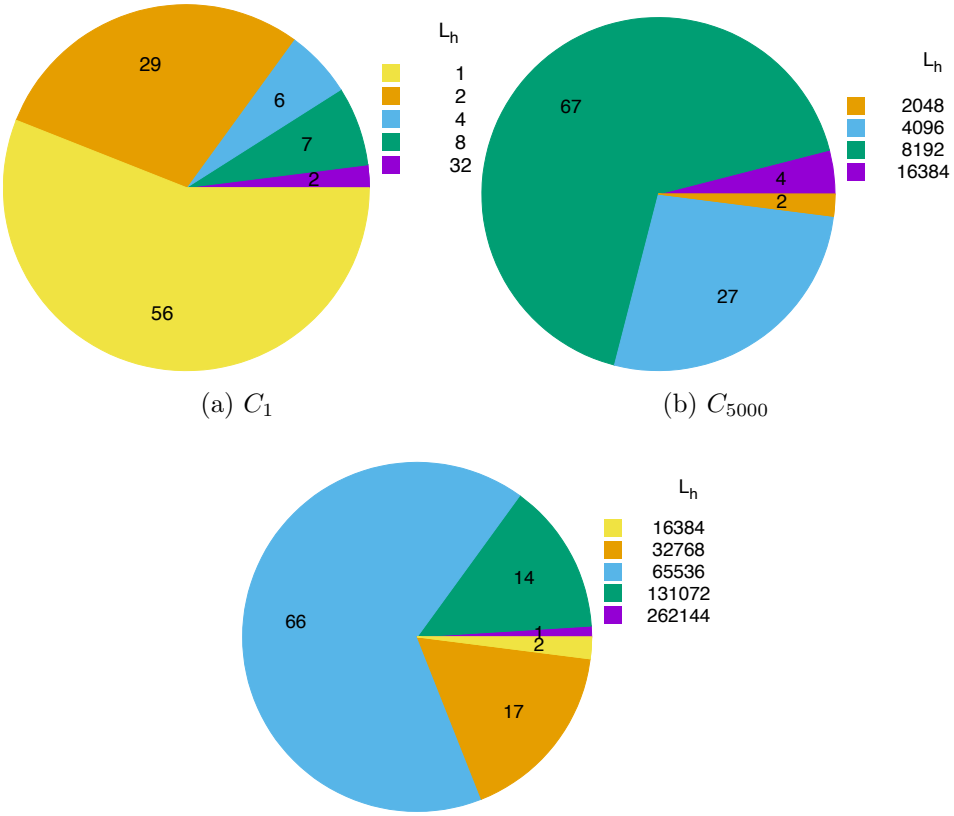
Table 3.2: Number of iterations needed by pLAHC to reach at least the same solution quality as that obtained by LAHC. OF stands for overhead factor, i.e. how much slower pLAHC is compared with LAHC. The results are averaged over 100 independent runs.

Dataset	Iterations needed by pLAHC to reach quality C_x					
	C_1	OF	C_{5000}	OF	C_{50000}	OF
rat783	1116839	1.44	102151765	3.60	710536424	2.75
u1060	3352705	1.54	149197449	3.44	1108126907	2.85
fl1400	6267468	1.55	187044984	3.24	1042146093	2.11
u1817	13169262	1.57	352278209	3.89	2207268273	2.94
d2103	19253166	1.55	468100823	4.16	2572121231	2.93
pcb3038	42243384	1.53	804089579	4.55	4116303379	3.07
fl3795	104336892	1.64	1409994608	4.33	6302508196	3.47

710536424 iterations (those needed by pLAHC to reach a solution quality $C_{50000} = 9105$) the average solution quality over 100 independent runs is only 9933, nowhere close to the 9105 value. As a matter of fact, none of those 100 runs reaches the 9105 value. A similar behaviour occurs with the other instances.

Our experiments collect other relevant information concerning the pLAHC runs. Due to space restrictions we focus our analysis on a specific instance, `u1817`, which is median in terms of dimensionality (the number of cities). Figure 3.1 shows the distribution of the required history list length needed by pLAHC to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the `u1817` instance. In Figure 3.1a, we can observe that the majority of the runs (56%) reach the target solution quality not needing any restarts, i.e., with $L_h = 1$. This result is consistent with what one would expect because we are only running pLAHC until it reaches the target quality C_1 (obtained by LAHC with $L_h = 1$). Similarly, in Figure 3.1b we observe that most of the pLAHC runs reach the target solution quality C_{5000} (obtained by LAHC with $L_h = 5000$) either with $L_h = 4096$ (27%) or $L_h = 8192$ (67%). Again, this is consistent with what is expected because these are the values closest to 5000. The same expected behaviour occurs in Figure 3.1c for the C_{50000} case, with most pLAHC runs needing to reach a history list length of 65536, close to 50000.

These results suggest that pLAHC is capable, without any tuning, to automatically discover an appropriate history list length required to reach a certain target solution quality.



(a) C_1 (b) C_{5000}

(c) C_{50000}

Figure 3.1: Distribution of the required history length needed by pLAHC to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the `u1817` instance.

3.3 Speeding up with Seeded Restarts

Here we explore a refinement of the pLAHC algorithm presented in the previous section. The idea is to use information collected from the execution of a LAHC run to seed the next history list upon a restart. In other words, as opposed to having the history list initialized with the solution quality of a randomly generated solution, we are going to explore an alternative mechanism to avoid starting the search from scratch. We note that the idea of seeding the start of the search in the context of parameter-less search algorithms has been suggested before (see Holdener 2008).

The first idea that comes to mind is to use the current solution of the LAHC run that has just expired, and use its cost function value to initialize

the history list of the new LAHC run. We tested this idea but the results were not good. The reason is rather obvious. If the history list is filled up with the cost value of the current solution, there is very little chance that the delayed acceptance criteria will be successful because the current solution is already very good (a local optimum) and the history list has no diversity; the combination of these two factors leads to having most perturbations rejected.

The idea behind *late accepting* is to use a delayed comparison (with a solution which was current several iterations ago) to escape from a local optima more easily. It is therefore desirable that the history list cannot be composed of extremely good and identical values; it needs to have diversity. Note that no diversity with a low quality value would not be a problem because in that case most perturbations on a current solution would be accepted (but that corresponds to the standard initialization of LAHC, i.e., to start the search from scratch, which is what we are trying to avoid.)

The above argument suggests that to be successful, a seeding strategy needs to initialize the history list with good, yet diverse, cost function values of previously visited solutions. The very next thing that comes to mind is to fill the history list with the cost function values of past successful iterations. By past successful iterations, we mean those iterations that yielded solutions that improve upon the best cost-function-value ever found so far. To do so, we use another list to memorize those best-so-far values. We call this list the *successful iterations list* (SIL).

pLAHC is now augmented with a successful iterations list. The list is shared among the LAHC restarts, and gets updated whenever there is an improvement on the best-so-far solution quality value, regardless of which LAHC produces it (Line 32 of Algorithm 3.1). Once a current LAHC expires, the next LAHC restarts with its history list initialized with values taken from the end of successful iterations list. Lines 8-12 of Algorithm 3.1 denote this initialization. Note that in case the length of the SIL is lower than the length of the newly created LAHC history list, we keep looping through SIL until the history list is filled up (in that case, we will have duplicate values.) In order to fulfil this functionality, we need to fetch the length of SIL in each restart, since it is a dynamic list. Line 10 of Algorithm 3.1 grants this, where $SIL.size()$ returns the size of SIL. The history list is then sorted with larger values at the virtual beginning and smaller values at the virtual end

Algorithm 3.1: Parameter-less Late Acceptance Hill-Climbing with Seeding (pLAHC-s).

Output: A solution to a given problem.

```
Produce an initial solution s                                   // Usually u.a.r
Calculate its cost function value C(s)
SIL = []                                  // Successful Iteration List (SIL)
Append C(s) to SIL
L_h = 1
best = s
while stopping criterion is not true do
    v = 0
    while v < L_h do                          // History list initialization
        w = SIL.size() − 1 − (v mod SIL.size())
        f_v = SIL_w
        v = v + 1
    sort (f)                                  // Sort in descending order
    I = 0                                                 // Iteration counter
    I_idle = 0                                          // Idle iteration counter
    do until (I > 100000) and (I_idle > I × .02)
                                              // Stopping criterion of LAHC,
        // originally introduced by Burke & Bykov (2017)
        Construct a candidate solution s'
        Calculate its cost function value C(s')
        if C(s') ≥ C(s) then
            I_idle = I_idle + 1
        else
            I_idle = 0                                          // Reset counter
        v = I mod L_h                                       // Virtual beginning
        if C(s') < f_v or C(s') ≤ C(s) then
            s = s'                                           // Accept candidate
        else
            s = s                                            // Reject candidate
        if C(s) < f_v then
            f_v = C(s)                          // Update the fitness array
        I = I + 1
        if C(s) < SIL_{SIL.size()−1} then                          // Update SIL
            Append C(s) to SIL
            best = s
    L_h = 2 × L_h
    s = best
return s
```

of the list.

This way, instead of restarting the search from a uniformly randomly generated solution, LAHC restarts from a good solution found previously and hopefully has sufficient diversity of good solution quality values in its history list right from the beginning. As we shall see, the combination of these two factors gives enough flexibility for LAHC to take advantage of the late acceptance idea without needing to start anew on every restart. We refer to this refinement of pLAHC as *parameter-less LAHC with seeding* (pLAHC-s). Pseudocode for it is presented in Algorithm 3.1, with lines in red colour indicating changes with respect to LAHC (Algorithm 2.7) to design pLAHC-s.

3.3.1 Experiments with pLAHC-s

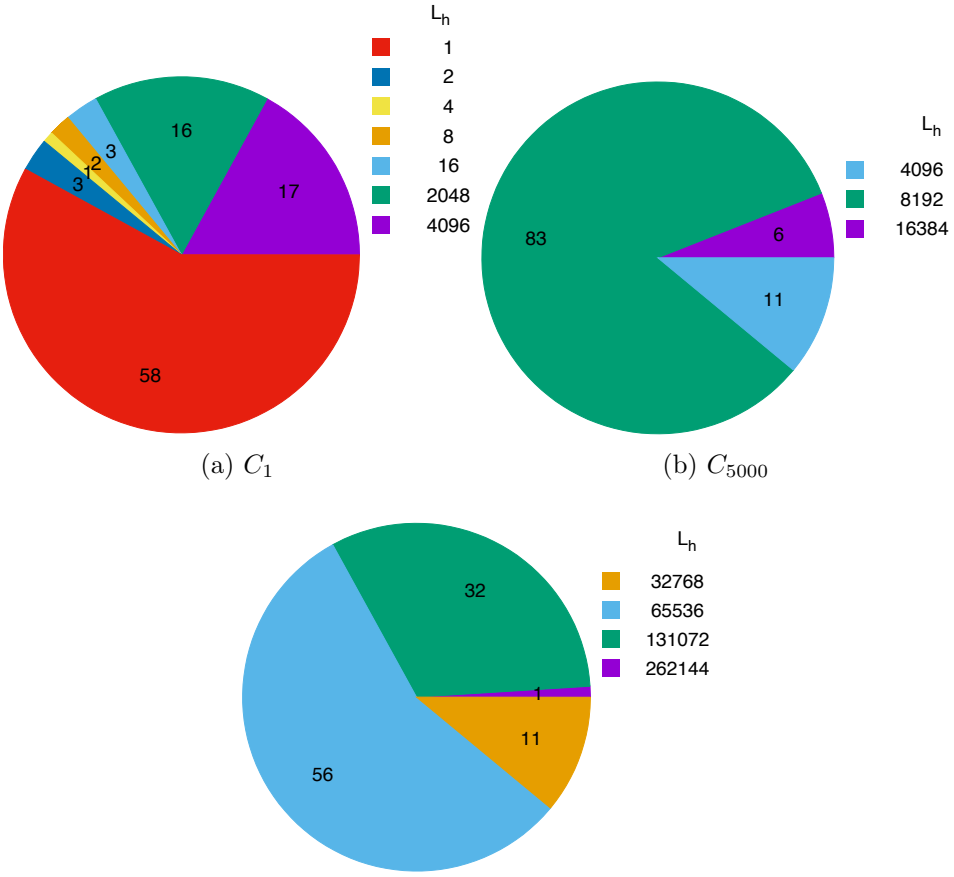
We repeat the experiments described in Section 3.2.2, this time with pLAHC-s instead of pLAHC. The results are summarized in Table 3.3. It is straightforward to observe that seeding the history list speeds up the search considerably. The overhead factors of pLAHC-s with respect to LAHC are substantially lower than those obtained with pLAHC, especially for the target solution quality C_{5000} and C_{50000}. These are the cases where LAHC requires long history length values, i.e., where restarts are an absolute must.

Table 3.3: Number of iterations needed by pLAHC-s to reach at least the same solution quality as that obtained by LAHC. OF stands for overhead factor, i.e., how much slower pLAHC-s is compared with LAHC. The results are averaged over 100 independent runs.

Dataset	Iterations needed by pLAHC-s to reach quality C_x					
	C_1	OF	C_{5000}	OF	C_{50000}	OF
rat783	771194	1.00	56110157	1.98	670283547	2.59
u1060	2225981	1.02	80241114	1.85	993939929	2.55
fl1400	6620509	1.63	136349568	2.36	987121266	2.00
u1817	13083187	1.56	130230000	1.44	1977770681	2.63
d2103	19857202	1.60	175890001	1.56	1718555845	1.96
pcb3038	48826067	1.77	163564379	0.93	2851442286	2.12
fl3795	128646608	2.03	360440032	1.36	3844065351	2.12

Figure 3.2 is the equivalent of Figure 3.1 for the pLAHC-s case. It shows the distribution of the required history length needed by pLAHC-s to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the `u1817` instance. Again, the results suggest that pLAHC-s is capable, without any

tuning, to automatically discover an appropriate history list length required to reach a certain target solution quality.



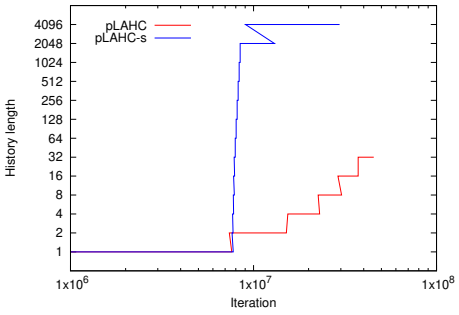
(a) C_1 (b) C_{5000}

(c) C_{50000}

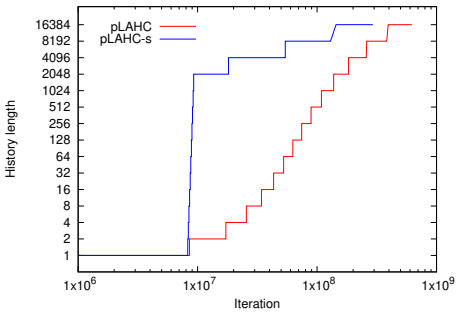
Figure 3.2: Distribution of the required history length needed by pLAHC-s to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the `u1817` instance.

Figure 3.3 illustrates the dynamics of restarts as the search progresses through time, pLAHC and pLAHC-s vis-a-vis, for the `u1817` instance. Figure 3.3a is for the target solution quality C_1, Figure 3.3b for target quality C_{5000}, and Figure 3.3c for target quality C_{50000}. The data used in the various plots was collected from the runs described earlier. For any given point in time (iteration number on the horizontal axis), we obtain the history list length (averaged over 100 independent runs) at that given point in time, for both pLAHC and pLAHC-s. It is notable that pLAHC-s quickly eliminates small-sized history lengths compared to pLAHC. The results agree with our

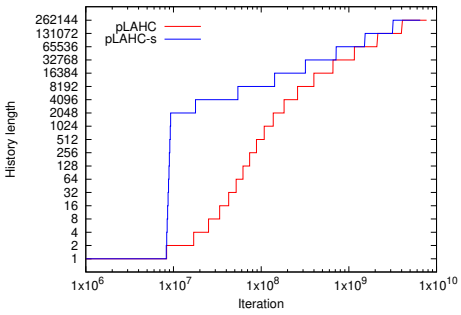
intuition; it is the quick elimination of small-sized history lengths that allows pLAHC-s to achieve a lower overhead factor compared to pLAHC.



(a) C_1

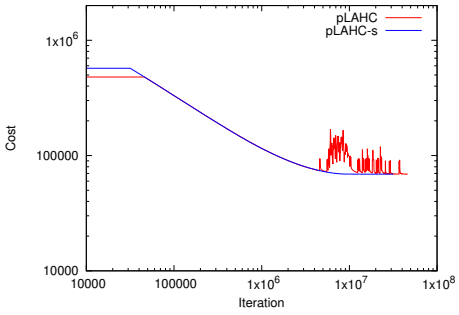


(b) C_{5000}

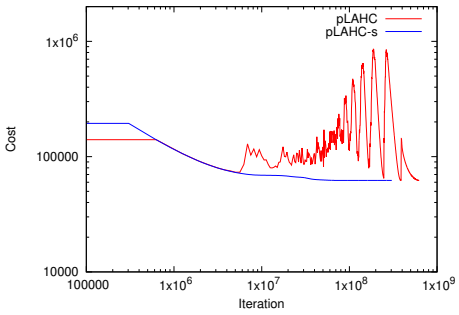


(c) C_{50000}

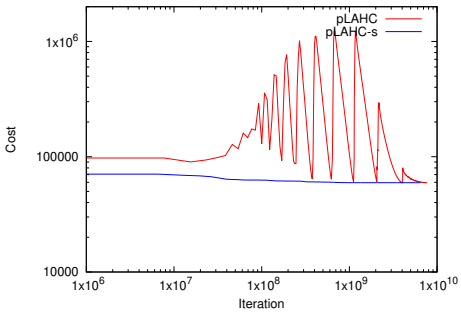
Figure 3.3: Average history list length through time for the `u1817` instance, pLAHC and pLAHC-s vis-a-vis. Data collected from the average of 100 independent runs.



(a) C_1



(b) C_{5000}



(c) C_{50000}

Figure 3.4: Average current solution cost through time for the `u1817` instance, pLAHC and pLAHC-s vis-a-vis. Data collected from the average of 100 independent runs.

Figure 3.4 also shows interesting dynamics of pLAHC and pLAHC-s

on the u1817 instance, this time showing the cost of the current solution through time, averaged over 100 independent runs. Note how the cost oscillates in the case of pLAHC but not in the case of pLAHC-s. The oscillation observed with the pLAHC case is due to restarting every single LAHC from scratch. This doesn't occur with pLAHC-s, precisely because of the seeding mechanism. Overall, the plots shown in Figure 3.3 and Figure 3.4 help us to understand in a clear way why the seeding mechanism is beneficial within the parameter-less restart strategy.

Although the plots in Figures 3.1, 3.2, 3.3, and 3.4, only concern the u1817 instance, we did analyze what happens with the other instances and the pattern observed is similar to the u1817 case.

The source code in C++ for the implementation of LAHC, pLAHC, and pLAHC-s, described in this chapter, along with the input files necessary to replicate all experiments presented herein are available online at https://github.com/mbazargani/pLAHC.

3.4 From Theory to Practice: A Case Study using a Software Engineering Problem

During this chapter, so far, we have introduced the pLAHC algorithm and shown its performance on several instances of TSP, a well-known academic problem. In the remainder of this chapter, we aim to show how pLAHC can be transferred to a real-world problem. For illustration purposes, we will be using a well-known Search Based Software Engineering (SBSE) problem, namely, Combinatorial Interaction Testing (CIT), which is a cost-effective black-box sampling technique for discovering interaction faults in highly configurable systems.

Before going further in this section, let us first briefly explain CIT in the context of software engineering. In recent years, instead of producing an entire application from scratch, modern software development often produces components of related products, where some components are integrated from existing applications. This provides reusable components that help developers to produce new products more quickly, offering a wider choice of features to both developers and users. As a result, newly produced software represents highly-configurable systems, which can add or remove features from the core set of software functionality in a flexible manner.

Such highly-configurable systems are more difficult to validate than traditional software with comparable scale and complexity. These systems raise the issue of interaction faults, since faults in a system may be triggered by interactions between features of different components. In the literature, it has been shown that it is generally impractical to test all possible configurations by validating one combination in a single run of a system (Kuhn, Wallace & Gallo 2004). Instead of doing so, testers need a technique to judiciously sample some combinations for validating the system. Empirical studies suggest that combinations of relatively few features actually cause triggering failures (Kuhn, Wallace & Gallo 2004). This finding has significant implications for testing, since testing combinations of all parameters is no longer required. This technique is known as Combinatorial Interaction Testing (CIT).

There are several techniques for CIT; one of the most established is Covering Arrays by Simulated Annealing (CASA) introduced by Garvin, Cohen & Dwyer (2011). CASA is a three-nested-layer search framework using SA in its most inner layer. Here we replace SA in CASA with pLAHC, proposing a modified framework, Covering Arrays by Late Acceptance (CALA). The result of this work was first published in Bazargani, Drake & Burke (2018), and in order to be better embraced into the structure of this chapter, it is presented in this single section.

In the next subsection, we present constrained CIT Problem and related work. Thereafter, Subsection 3.4.2 briefly explains the three-layer search framework of CASA and presents the proposed modified framework, CALA. The benchmarks used, experimental settings and results are given in Subsection 3.4.3.

3.4.1 Constrained Combinatorial Interaction Testing Problem

In the literature, a t-way interaction test suite covers a set of t combinatorial features and is known as the strength of combinatorial interaction testing. The central problem of CIT is to construct Covering Arrays (CA) with a minimum number of rows. A CA contains N rows with k columns, where each column represents a feature of the system. Each column can only contain valid values of the corresponding feature. In CA, for any choice of t columns, all combinations of t features (all sets of t-way interactions) should

appear in at least one row. Consequently, the t-way interactions are said to be covered. The aim is to cover all possible sets of t-way interactions in a minimum number of rows N. In the literature, the notation of covering arrays is typically presented as $CA(t, v_1^{k_1} v_2^{k_2} \dots v_m^{k_m})$, where t is the strength of the array, the sum of $k_1, k_2 \dots k_m$, is the number of features, and v_i denotes the number of values that each of the k_i feature(s) can take.

Empirical studies reveal that most failures are triggered by interaction between only two features (2-way) of a system, and that no failure was recorded to be triggered with t greater than 6 features (6-way) (Kuhn, Wallace & Gallo 2004, Nie & Leung 2011). Most real-world systems also have constraints, where some values of different features cannot appear together. CAs supporting constraints are referred to as Constrained Covering Arrays (CCA) (Cohen, Dwyer & Shi 2008). In this context, the goal of the combinatorial interaction problem is to find a minimum number of rows that cover all valid t-way combinations of features' values of a system, also known as tuples, with respect to its constraints. The presence of constraints in CIT is a major impediment to building an optimized CCA (Garvin, Cohen & Dwyer 2011, Galinier, Kpodjedo & Antoniol 2017).

Let us explain CIT with an example. Assume that in a simple system, there are three features, X, Y, and Z, where they can respectively have the following values 2, 3, and 3. This system has also two constraints as follows: 1) $X \neq Y$ which donates that X and Y cannot have the same value, and 2) $\neg(X = 1 \wedge Z = 2)$ indicates that when X is 1, Z cannot be 2 and vice versa. For a 2-way interaction, this system is presented as $CA(2, 2^2, 3^1)$. There are only 6 feasible covering arrays out of 12 total solutions. Those 6 CCA are: $(1,2,1), (1,3,1), (2,1,1), (2,1,2), (2,3,1), (2,3,2)$. However, for 2-way interactions, all 2-way tuples can be covered in 5 CCA as follows: $(1,2,1), (1,3,1), (2,1,1), (2,1,2), (2,3,2)$. Obviously, we eliminated $(2,3,1)$, since three tuples in this array is covered by others. Thus the objective is to find a minimum number of arrays that can cover all possible t-way tuples.

CIT has proven to be useful when testing software product lines, operating systems, development environments, and many other systems that are typically governed by a large configuration, parameters, and feature spaces (Nie & Leung 2011).

Heuristic Approaches to the CIT Problem

Heuristic search methods have been successfully applied to the constrained CIT problem to construct CCA. More often than not, those methods use an off-the-shelf satisfiability solver (a.k.a. SAT solver) to check whether or not the constructed rows of a CCA satisfy the constraints. In the literature, heuristic techniques devoted to CCA can be classified into two categories (Nie & Leung 2011), one-test-at-a-time (e.g., Cohen, Dalal, Kajla & Patton 1994) and in-parameter-order (e.g., Yu, Lei, Nourozborazjany, Kacker & Kuhn 2013).

A number of different metaheuristic methods have been proposed for constructing and improving CCAs. Many of these methods first construct a valid CCA with N rows for a CIT problem instance, using one of the one-test-at-a-time methods. They then iteratively reduce the number of arrays of the initial CCA, using a metaheuristic search algorithm to make the new CCA feasible with respect to the given constraints (Bryce & Colbourn 2007). This process continues until a given stopping criterion is met, such as a particular number of iterations or a given time budget. The most well-established software application among this kind of framework is the Covering Arrays by Simulated Annealing (CASA) tool (Garvin, Cohen & Dwyer 2011). CASA is based on a nested three-layer search framework. An outer search layer, resembling binary search, selects a target value for N, with an inner layer based on SA used to attempt to cover all tuples within a CCA of that size. As CASA is the basis for much of our experimentation in our study in this section, it is discussed in detail in Subsection 3.4.2.

In addition to CASA, two other frameworks of this nature have recently shown promising results for the CIT. Lin, Luo, Cai, Su, Hao & Zhang (2015) presented a 'Two-mode meta-heuristic framework for Constrained Covering Arrays' (TCA), which uses a mixture of random walk and Tabu Search. In each iteration of TCA, one uncovered valid t-tuple is inserted into a randomly selected array (row) modifying only one cell. The modification only happens if that cell has not been changed during the last T iterations, where T is the length of tabu tenure (see Section 2.2 for details on Tabu Search). More recently, Galinier, Kpodjedo & Antoniol (2017) proposed Covering Array by Tabu Search (CATS). Unlike other methods that are restricted to feasible areas of the search space, CATS extends the search process to allow

infeasible solutions, using an objective function that balances between the number of constraint violations and the number of uncovered valid tuples. Like TCA, it also puts restrictions on modifying a recently changed cell in a CCA for a number of iterations based on the length of the tabu tenure.

There has recently also been work applying hyper-heuristics to the CIT problem. Jia, Cohen, Harman & Petke (2015) reported that their hyper-heuristic approach outperforms CASA, employing a Simulated Annealing-based hyper-heuristic and six low-level heuristics. Zamli, Alkazemi & Kendall (2016) proposed a high-level hyper-heuristic (HHH) to tackle CIT, using Tabu Search as a high-level metaheuristic operating over four different low-level metaheuristics.

Although here we have provided a review of some of the best-known methods for combinatorial interaction testing, we refer the interested reader to the following recent surveys on the topic by Khalsa & Labiche (2014) and Nie & Leung (2011) for a detailed review of these and other approaches.

3.4.2 Covering Arrays by Simulated Annealing (CASA)

In 2003, Cohen, Colbourn & Ling (2003) first introduced a two-layer framework for the constrained CIT problem, based on an outer search layer, and an inner search layer. The proposed framework works iteratively, performing outer search and inner search in each iteration. The outer search, resembling binary search, decides a target value for the size of CCA (N) to search for, within a particular range. The inner search layer, aided by SA, then attempts to fit all valid t-way interactions within a covering array of that size. Using a covering array initialised by AETG (Cohen, Dalal, Kajla & Patton 1994), a simple mutation operator then generates a new solution, with SA used to decide whether to accept the new solution. The mutation operator replaces the value of a randomly chosen feature from a random row of a CCA with another valid value for that feature. If the inner search fails, the outer search decreases the upper bound on the range of current CCA sizes to the current CCA size and defines a target value for N within the new range. In the case that the inner layer is successful, the lower bound on the range of CCA sizes is increased to the current CCA size and again another value for N is taken from the new range.

Extending this existing framework, Garvin, Cohen & Dwyer (2009, 2011) introduced the three-layer Covering Array by Simulated Annealing (CASA)

framework. The features of this three-layer search algorithm for the constrained CIT problem are shown in Figure 3.5. The three layers are referred to as *outermost search*, *binary search*, and *inner search*. Binary search and inner search are called iteratively in a similar manner to the previous version, to build a CCA. Once binary search is terminated it sends the constructed CCA to the outermost-search layer. Then that layer will again call binary search for further search. The outermost search halts the search process once the binary search in cooperation with inner search cannot find a smaller size CCA.

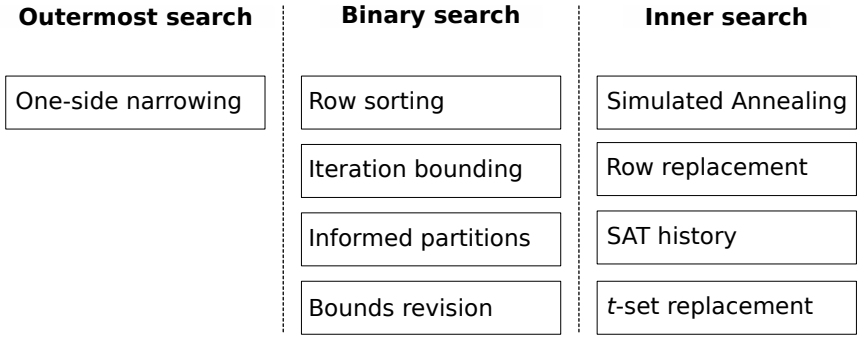
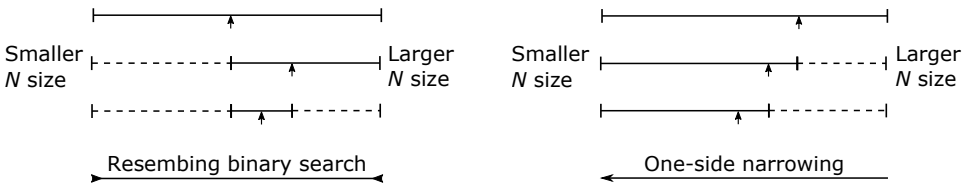


Figure 3.5: Features of each layer of the three-layer search framework of CASA.

The three-layer search starts from the outermost-search layer with a given problem, lower and upper bounds on the range of CCA sizes, and Inner Iteration Limit (IIL). The IIL is the stopping criterion for SA-based inner-search layer below. This layer is complementary to the binary-search layer. The binary search supposes that the inner search can determine whether or not it is possible to generate a CCA for a given size N. Based on the result that binary search receives from the inner-search layer, it eliminates a range of sizes of N and will never again revisit that range. As the inner search uses a stochastic algorithm, it cannot assure that finding a CCA of that size is possible. As a result, following the termination of the binary-search layer, the outermost-search layer does *one-side narrowing* of the range of CCA sizes (N) from the upper bound, and then calls the binary-search layer again. Figure 3.6 illustrates the difference between the binary search and the outermost search layers. This way the soundness of using binary search is somehow guaranteed, as it has a chance of revisiting those ranges that were eliminated earlier in the previous call of the binary-search layer.



(a) Binary search layer (b) Outermost search layer

Figure 3.6: Binary search vs outermost search (one-side narrowing). Dashes partitions are the eliminated subrange of N. In a and b, the subrange progresses from top to bottom.

The binary-search layer has four features, namely *row sorting*, *iteration bounding*, *informed partitioned*, and *bounds revision*. This layer receives three parameters from the outermost-search layer, i.e., upper and lower bounds as well as IIL. Based on the feedback it receives from the inner-search layer, it modifies IIL (iteration bounding) and the upper and lower bounds (bounds revision). Iteration bounding doubles the inner iteration limit under certain conditions. The binary-search layer estimates the best choice for the next value of N, based on the number of iterations taken to find a feasible CCA for the current N value (informed partitions). When the inner-search layer successfully finds a feasible solution for a given N, some rows of the current solution will need to be removed as the next value of N will be smaller. When doing so, the binary-search layer keeps the rows containing less frequent t-sets for the next call of the inner-search layer, as they are more difficult to make than t-sets that are already duplicated multiple times within a solution.

The inner-search layer receives a CCA from the binary-search layer. If it receives a valid CCA, it immediately returns to the binary-search layer and receives a smaller CCA. Once it receives an invalid CCA, it attempts to transform it into a valid CCA. The inner-search layer uses three different strategies to modify and create rows, using SA to decide whether to accept newly generated solutions. In each iteration, it writes a missing t-set to a randomly chosen row (*t-set replacement*). It then sends the modified row to a SAT solver to check feasibility. It accepts the modified row if it is feasible, otherwise it tries again to modify a randomly chosen row with a t-set replacement strategy. If for 32 consecutive attempts it fails to make a feasible t-set replacement, then row replacement is performed. The *row replacement* strategy randomly picks a row and replaces it with a entirely

new generated row. Before using the newly generated row, it will be sent to a SAT solver to ensure it is feasible. If it is not feasible, values of that row will keep being perturbed until it becomes a feasible solution. During this process, the algorithm remembers infeasible values that are rejected and does not try them again (*SAT history*).

SA accepts worsening moves with probability $p = \exp((C - C')/T)$, with C and C' denoting the cost function values of current and candidate solutions, respectively, and T is the temperature (see Section 2.2 for more details on SA). In CASA, the cost function is the number of non-covered tuples. The initial solution is randomly generated. The starting temperature is 0.5, and is updated in each iteration using a cooling rate of 0.0001% (Garvin, Cohen & Dwyer 2011). SA halts once it finds a CCA for a given number of rows N, or when it exceeds the IIL as defined by the binary-search layer.

Proposed Modifications: CALA

Herein we propose the use of pLAHC in the context of the constrained CIT problem. To provide a fair comparison between SA and pLAHC, we use the CASA framework as the basis of our work. We replace SA in that framework with the pLAHC search technique, calling this modified approach Covering Arrays by Late Acceptance (CALA). Note that the three-layer CASA framework was designed and tested specifically using SA. Replacing SA in CASA with pLAHC will provide us an insight whether or not pLAHC is an effective alternative to SA within this framework.

To implement CALA we added the history list of LAHC to the inner-search layer. Each time that the inner-search layer is called, the list will be initialised with the number of non-covered tuples (since it is the objective function of the inner search) in the CCA received from the binary-search layer. CALA modifies the current solution using the same strategies as CASA. The initial length of the history list is set to 32. This is based on what we explained earlier in Section 3.2 that the initial history list length can be very small. We double the length of the history list as suggested in pLAHC (see Section 3.2) whenever the maximum number of iterations are doubled by iteration bounding. The idea behind doubling the maximum number of iterations (IIL) is that the inner-search requires more time to find a CCA. And again, the idea behind doubling the length of history list in

pLAHC is to allow LAHC to accept a greater number of worsening moves, in order to explore the search space more widely.

To provide a simple baseline for comparison, we also replaced SA with standard Hill-Climbing (HC) and a non-deterministic naïve move acceptance (naïve) (Burke, Curtois, Hyde, Kendall, Ochoa, Petrovic, Vazquez-Rodriguez & Gendreau 2010). The results of HC will provide evidence as to whether accepting worsening moves is necessary to improve performance in the three-layer search framework of CASA. This has not been reported in the literature so far. The non-deterministic naïve move acceptance accepts all improving moves, and worsening solutions with a given fixed probability. This will give some indication as to whether it is the presence of non-improving moves that improves performance, or the adaptive mechanism that controls such moves that is required.

3.4.3 Experimentation

This section describes the experiments performed to evaluate the performance of pLAHC within the three-layer search framework of CASA. Since the presence of constraints increases the difficulty of constructing an optimized CCA and better reflects problems found in the real world, we performed all our experiments on constrained problem instances. We use two well-known benchmark suites:

- **[Real-2]** (Cohen, Dwyer & Shi 2008) contains five constrained 2-way real problem instances. Apache is a web server application, Bugzilla is a web-based bug tracking system, GCC is a compiler system from the GNU project, Spin-S and Spin-V are two components for model simulation and model verification.

- **[Syn-C2]** (Garvin, Cohen & Dwyer 2011) contains 30 constrained 2-way problem instances. This benchmark suite was randomly generated, based on the structure of the five real-world problem instances in [Real-2].

To implement the modifications outlined in subsection above, we used the freely available C++ code of CASA from `https://cse.unl.edu/~citportal/`. In our experiments, we applied four different acceptance methods to each problem instance using eight different inner iteration limit (IIL) of 256,

512, 1024, 2048, 4096, 8192, 16384, and 32768, as it is suggested by Garvin, Cohen & Dwyer (2011). For each IIL, 100 independent runs were executed. In the case of non-deterministic naïve move acceptance experiments, for each IIL, we used 9 different probabilities for accepting worsening moves, i.e., 10%, 20%, ..., 90%. In this case, 100 independent runs were executed for each probability.

In the following, we present and analyse the results obtained using four different acceptance methods within the CASA framework. We report the best results as is the practice in the CIT literature (Galinier, Kpodjedo & Antoniol 2017, Garvin, Cohen & Dwyer 2009). For each instance, results are obtained from 100 independent runs of 8 different inner iteration limits per algorithm.

Table 3.4 compares the best results obtained using SA, pLAHC, HC and the best of the nine variants of naïve hill climbing, from the eight IIL values tested for each. The first column lists the name of each problem instance. The first five problem instances are real problem instances from the [Real-2] benchmark, and the remaining 30 instances are the synthetic problem instances from the [Syn-C2] benchmark. Problem instances where CALA found a smaller CCA than CASA are shown in gray. The number of unconstrained and constrained parameters of these problem instances are presented in the second and third columns respectively. Figure 3.7 presents box plots for a selection of instances, showing the performance of methods over all 100 runs for those instances.

Despite the fact that CASA was designed and tuned to use SA, pLAHC performs very well in general in terms of CCA size as shown in Table 3.4. CALA is able to outperform CASA in terms of CCA size obtained in 14 of the 35 problem instances tested, matching the performance of CASA in another 20 problem instances. CASA yields a better result than CALA in only one problem instance, instance 30 of [Syn-C2]. However, Figure 3.7e shows that there is little difference in performance between CASA and CALA on that problem instance. Note that simple HC is able to find the same results as CASA and CALA in 14 problem instances (2 of [Real-2] and 12 of [Syn-C2]), suggesting that an advanced high-level search method is not required for all instances. Despite this, inferior performance in the remaining instances indicates that accepting worsening moves during the search is required to find very high-quality CCA. Of the 14 instances where HC achieved the same

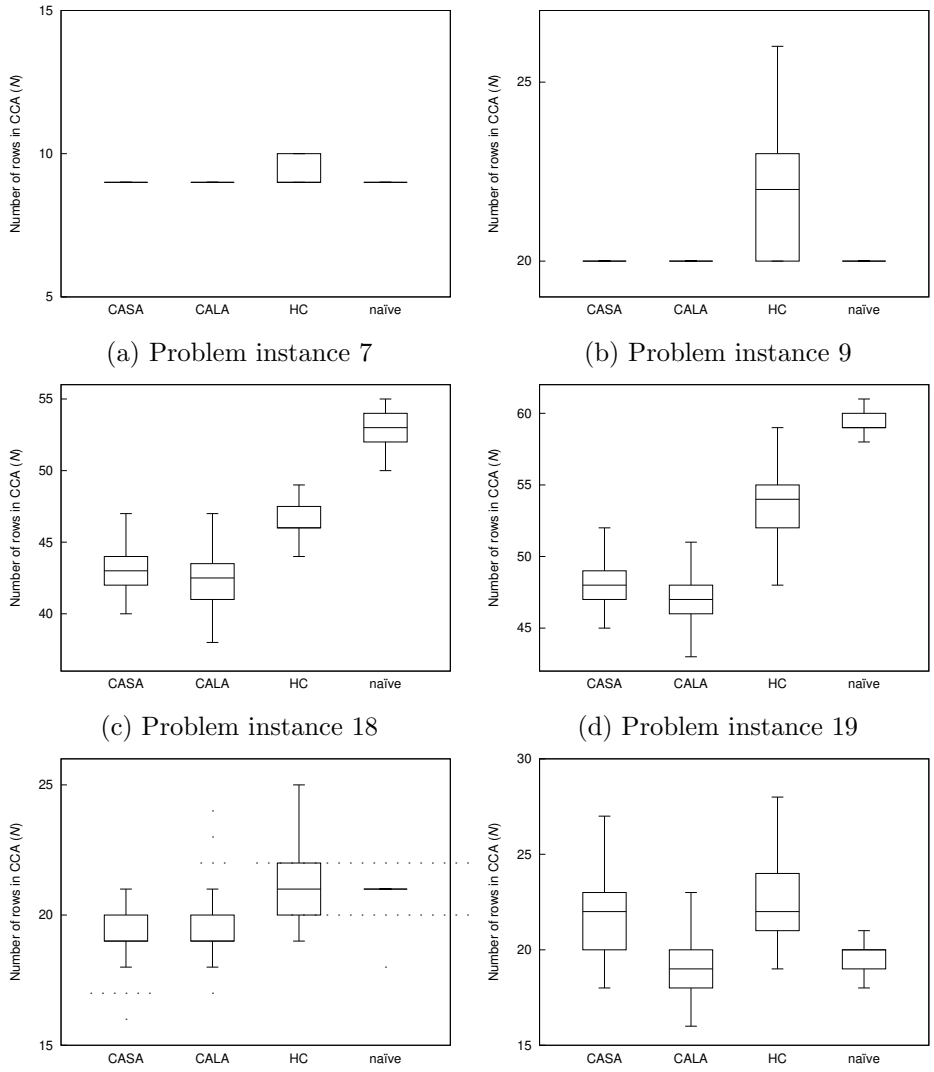
Table 3.4: Best solutions (size of CCA, N) produced by CASA, CALA, Hill-Climbing (HC) and non-deterministic naïve move acceptance implemented within the three-layer search framework of CASA over 100 independent runs. Problem instances where CALA found a smaller CCA than CASA are shown in gray. Lowest size of CCA in each row are shown in boldface.

Name	Model	Constraints	CASA	CALA	HC	naïve
Apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	**30**	**30**	31	33
Bugzilla	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	**16**	**16**	**16**	**16**
GCC	$2^{189}3^{10}$	$2^{37}3^{3}$	18	**16**	19	18
SPIN-S	$2^{13}4^{5}$	2^{13}	**19**	**19**	**19**	20
SPIN-V	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	33	**32**	34	37
1	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	37	**36**	39	46
2	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	**30**	**30**	**30**	31
3	$2^{27}4^{2}$	$2^{9}3^{1}$	**18**	**18**	**18**	**18**
4	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	**20**	**20**	**20**	**20**
5	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	44	**43**	48	56
6	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	**24**	**24**	**24**	**24**
7	$2^{29}3^{1}$	$2^{13}3^{2}$	**9**	**9**	**9**	**9**
8	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	38	**37**	42	48
9	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	**20**	**20**	**20**	**20**
10	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	41	**38**	45	53
11	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	40	**39**	43	51
12	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	**36**	**36**	41	45
13	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	**36**	**36**	**36**	37
14	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	**36**	**36**	38	40
15	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	**30**	**30**	**30**	31
16	$2^{81}3^{3}4^{2}6^{1}$	$2^{30}3^{4}$	**24**	**24**	**24**	**24**
17	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$	**36**	**36**	40	44
18	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$	40	**38**	43	50
19	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$	45	**42**	48	57
20	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$	**51**	**51**	53	66
21	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$	**36**	**36**	37	39
22	$2^{72}3^{4}4^{1}6^{2}$	$2^{31}3^{4}$	**36**	**36**	**36**	**36**
23	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$	**12**	**12**	**12**	**12**
24	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$	40	**39**	43	51
25	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{3}4^{1}$	46	**45**	49	59
26	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$	29	**27**	31	35
27	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$	**36**	**36**	**36**	**36**
28	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$	49	**47**	51	62
29	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	26	**25**	28	30
30	$2^{73}3^{3}4^{3}$	$2^{20}3^{2}$	**16**	17	19	18

results as CASA and CALA, simple naïve acceptance was also able to obtain the same results for 10 of them. This highlights two things. Firstly, for those 14 instances where naïve acceptance matches CASA and CALA, it might be that performance is determined by other parts of the framework than the inner-search layer that are not varied within our experiments. Secondly, if it is the case that accepting non-improving moves is necessary for good performance, as we have supposed based on the performance of HC, clearly an intelligent mechanism to manage such moves is required.

Although Table 3.4 and Figure 3.7 provide an overview of the performance of each acceptance method, due to the nature of the termination criteria of the CASA framework the computational effort to generate these results can differ. Table 3.5 reports the number of function evaluations executed (FE) and inner iteration limit (IIL) used by the the acceptance methods which obtain the results given in Table 3.4. The probability that was used by the non-deterministic naïve move acceptance to obtain the best results for each instance is also reported (p_n). Note that in Table 3.5, we give the results with the smallest size CCA executing the smallest number of FEs. Thus, in a few cases, it is possible that a smaller IIL is able to generate the same "best" size CCA. However, the IIL of the acceptance criterion using the smallest number of FEs is given. Figure 3.8 shows the average of the 35 best CCAs found by each acceptance method, for each of the eight IIL values tested.

Here we observe that increasing the IIL when using SA is not leading to improved performance, with the majority of best results using a limit of 256 iterations. Although we tested the iteration limit for CASA at eight different levels (from 256 through to 32768), improvement in solution quality was only observed in a handful of cases. This is in contrast to CALA, where a higher iteration limit can lead to improved performance. Note that in general a higher iteration limit corresponds to a longer list length, which has previously been shown to improve the performance of LAHC in other problem domains. This trend is clearly visible in Figure. 3.8, where we plot the best sizes of CCA using eight different levels of IIL averaged over all 35 problem instances. For ten of the eleven problem instances that CASA and CALA report the same solution quality from the same IIL, it is also worth highlighting that CALA usually performs fewer FEs on average than CASA (i.e., [Syn-C2] instances 2, 3, 4, 6, 9, 15, 16, 22, 23, 27). For HC, most of



(a) Problem instance 7

(b) Problem instance 9

(c) Problem instance 18

(d) Problem instance 19

(e) Problem instance 30

(f) GCC

Figure 3.7: Sizes of CCA obtained over 100 independent runs for 6 different problem instances. In (e) we also show the statistical outliers. In problem instance 30, only in one run —a statistical outlier— CASA reports a smaller CCA than CALA.

Table 3.5: Inner iteration limit (IIL), and average function evaluations (FE) used by acceptance methods obtaining the best results for each instance over 100 independent runs. p_n reports the probability that was used by the best-performing non-deterministic naïve move acceptance. The problem instances where CALA found a smaller CCA than CASA are shown in gray.

Name	CASA		CALA		HC		naïve		
	FE	IIL	FE	IIL	FE	IIL	FE	IIL	p_n
Apache	12428	256	42984	1024	21318	1024	3217527	32768	0.1
Bugzilla	6906	256	5014	256	3042	256	14601	256	0.2
GCC	23966	256	170960	4096	19933	256	47475	256	0.1
SPIN-S	31709	256	28599	256	5609	256	263872	8192	0.1
SPIN-V	85587	1024	249668	4096	440130	8192	652949	2048	0.1
1	271544	256	705174	2048	93010	4096	387637	2048	0.1
2	12800	256	9515	256	7497	512	135987	1024	0.1
3	3971	256	2530	256	2550	256	2914	256	0.1
4	18606	256	7382	256	96610	8192	125470	2048	0.1
5	337674	2048	1725605	32768	211295	4096	1446352	8192	0.1
6	9279	256	9138	256	3868	256	321926	4096	0.1
7	2239	256	3723	256	2302	256	5494	256	0.1
8	262800	256	276528	4096	48474	2048	720739	2048	0.1
9	10502	256	9009	256	8851	256	24964	256	0.1
10	1773201	16384	728932	8192	14899	256	341910	1024	0.1
11	299857	256	408347	8192	18350	512	83994	256	0.1
12	78917	256	134122	512	83982	4096	166717	256	0.1
13	25669	512	14098	256	7728	512	97738	512	0.1
14	44803	256	56625	2048	303042	16384	2115549	16384	0.1
15	19422	256	6979	256	162134	8192	349896	4096	0.1
16	8815	256	8445	256	3632	256	53157	512	0.1
17	183186	512	150178	2048	21319	1024	393618	2048	0.1
18	204717	256	921257	16384	223842	8192	692631	4096	0.1
19	215703	512	1418309	32768	825576	32768	2126520	16384	0.1
20	990948	512	1826568	32768	1369898	32768	569307	1024	0.1
21	44435	256	26059	512	33339	2048	95927	512	0.1
22	5836	256	3431	256	2818	256	21103	256	0.2
23	9171	256	7767	256	6068	256	12667	256	0.5
24	572118	2048	516839	8192	622386	32768	1468696	8192	0.1
25	600257	512	1685187	32768	27570	256	1971717	8192	0.1
26	67397	256	212219	4096	10488	512	46200	256	0.1
27	11015	256	7809	256	4698	256	39820	256	0.1
28	525787	256	2672647	32768	133453	4096	1817860	8192	0.1
29	104441	2048	59401	256	378675	16384	123522	256	0.1
30	488065	16384	26133	1024	11888	512	38258	512	0.1

the best solutions reported have an inner iteration limit (IIL) of 256, the smallest iteration limit used. This is perhaps to be expected, since HC does not have a strategy to escape from a local optimum and can quickly arrive at sub-optimal solutions.

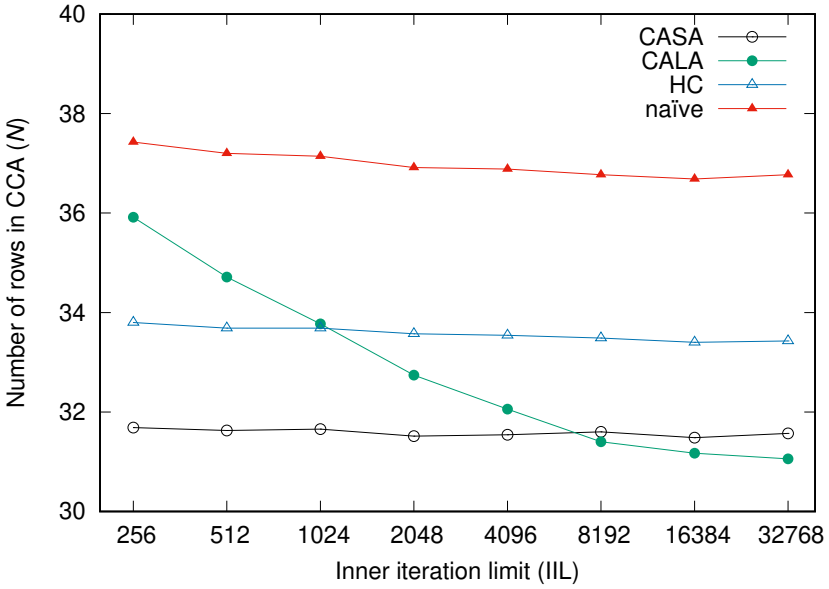


Figure 3.8: Best sizes of CCA using eight different levels of IIL averaged over all 35 problem instances.

For the naïve approach, the best results are almost always obtained with $p_n = 0.1$, i.e., worsening moves are accepted 10% of the time. Again this might be expected, as increasing the value of p_n simply leads the search to behave as a random walk. What is not shown in Table 3.5 are the cases where other values of p_n achieve the same best CCA size, albeit using a greater number of FEs. As mentioned above, it appears that some instances are easier to solve than others within the overall CASA framework. For a number of instances (Bugzilla and [Syn-C2] instances 3, 4, 6, 7, 9, 16, 22, 23 and 27), the same best CCA size is found using all four acceptance methods. Figures 3.7a and 3.7b show two of these instances. In the case of instance 3 and 23 from [Syn-C2], the same best results are obtained by all inner iteration limits (256, ..., 32768) of all naïve probabilities (10%, ..., 90%). As discussed above, it seems that in the case of these instances the search process is driven by other mechanisms within the framework.

3.5 Summary

This chapter proposed an automated strategy to eliminate the sole parameter of the LAHC algorithm, a general purpose metaheuristic introduced by Burke & Bykov (2017). LAHC belongs to the class of local search methods, is simple to implement, and easy to use in practice, requiring the tuning of a single parameter whose effect on the search is well understood (see Subsection 2.2.1). Building on techniques that have been successfully used in parameter-less evolutionary algorithms, we developed a parameter-less version of LAHC, i.e., pLAHC, and then refined it even further with seeded restarts, i.e., pLAHC-s. The resulting parameter-less algorithm is of course slower than a LAHC fine-tuned with an appropriate history length value, but has the advantage of not requiring any tuning (which in practice is time consuming, and needs to be redone for different problems, and even for different instances across the same problem.) Experiments on several TSP benchmark instances show that the parameter-less LAHC is effective.

Afterwards, in order to show that the proposed approach has a broad applicability, we applied the pLAHC algorithm to a real-world problem taken from the Search Based Software Engineering field. We employed the constrained CIT problem using the well-established CASA framework. CASA was specifically designed and tuned to use Simulated Annealing. In the proposed modified framework, CALA, Simulated Annealing is replaced within CASA by pLAHC. Using the structure of the CASA framework, the history list length of LAHC is set and controlled automatically. Although CASA was originally designed and tuned to use Simulated Annealing, pLAHC shows good performance compared to Simulated Annealing in the CASA framework. In 14 of the 35 benchmark problem instances tested, CALA outperforms CASA, with CASA only outperforming CALA in one instance.

It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong.

Richard P. Feynman

Chapter 4

Theory Driven by Practice: A Cutoff Time Strategy for SLS

The performance of pLAHC significantly relies on when the LAHC restart is fired. Throughout the course of an optimization run, the probability of yielding further improvement becomes smaller as the search proceeds, and eventually the search stagnates. Under such a state, letting the algorithm continue to run is a waste of time as there is little hope that subsequent improvement can be made. The ability to detect the stagnation point is therefore of prime importance. If such a point can be detected reliably, then it is possible to make better use of the computing resources, perhaps restarting the algorithm at the stagnation point, either with the same or with a different parameter configuration.

This chapter proposes a new *cutoff time* strategy. It presents a method that is able to reliably detect the stagnation point for one-point stochastic local search algorithms applied to combinatorial optimization problems. The strategy is derived from the Coupon Collector's Problem (CCP), and is scalable based on the employed perturbation operator and its induced neighbourhood size, as well as from feedback from the search. The suitability and scalability of the method is illustrated with the Late Acceptance Hill-Climbing algorithm on a comprehensive set of benchmark instances of three well-known combinatorial optimization problems: the Travelling Salesman Problem, the Quadratic Assignment Problem, and the Permutation

Flowshop Scheduling Problem.

The Coupon Collector's Problem is a well-known problem from probability theory, and has been used to analyse the behaviour of randomized algorithms (Motwani & Raghavan 1995). However, in this chapter, we use it to introduced a new dynamic cutoff time strategy. In the CCP there are n coupon types and at each trial a coupon is chosen uniformly at random with replacement. The CCP answers the following question: how many trials need to be collect to get at least one copy of each type of coupon? In the proposed cutoff time, the size of the neighbourhood equated the number of coupons, n, in CCP formulation. Thus, the CCP calculates the number of ideal moves in which, with a high probability, we can conclude that the search is stagnant. Note that the perturbation operator used in a SLS algorithm induces a neighbourhood of a given size.

This work was first published in Lobo, Bazargani & Burke (2020); it has a companion website available at `https://mbazargani.github.io/CCPcutoffTime/` and also a 79-page PDF document as supplementary material available at `https://bit.ly/3foVjud`. The chapter is organized as follows. The next section explains the motivation behind this work. Thereafter, Section 4.2 reviews the coupon collector's problem and how it can be used to design a cutoff strategy for stochastic local search. Sections 4.4 and 4.5 present the experimental setup and the analysis of the results obtained, respectively. In Section 4.6, we discuss two criticisms that can be made to the proposed technique. Finally, we summarize and present the major conclusions of this chapter.

4.1 Motivation

As we explained in Section 2.2, one-point SLS algorithms maintain a current solution which is perturbed until a specified stopping criterion is reached. At each iteration, the current solution is modified yielding a candidate solution. This can be accepted, becoming the new current solution for the next iteration, or rejected, in which case the current solution remains unchanged. One-point SLS algorithms differ among each other in large part by their acceptance criteria and in the way they use (or not) additional memory to help guide the algorithm. The current solution together with the additional memory define *the state of the SLS algorithm* at any given point in time.

During the course of running such an algorithm, the current solution changes quite often during the initial phase of the search. Then, as the search progresses, those changes become less frequent, and eventually the search stagnates. Under such a state, there is little chance that better solutions are produced, and it may be beneficial to either stop or restart the algorithm. Indeed, it is widely recognized that SLS algorithms often benefit from restarts (Hoos & Stützle 2005, p. 192). By doing so, the algorithm has a chance to explore new regions of the search space that can potentially lead to other good solution(s), as opposed to always staying in the same region, which could be already close to being fully explored and with little hope of yielding further improvement. In some sense, the restarting mechanism deals with the dilemma of exploration versus exploitation. By restarting, the algorithm can explore an unexplored region of the search space; by not restarting, the algorithm continues to exploit the region that it has already explored. Another way to look at this problem is to recognize that at any given point in time during the run, the SLS algorithm needs to make a decision: either continue its normal mode of operation, or perform a restart from a different position in the search space. A crucial aspect is to decide the proper time to do the restart, the so-called cutoff time.

For algorithms that fully explore the neighbourhood of a solution, the cutoff time decision can be easily made. This is the case with iterative first improvement or iterative best improvement algorithms (see Section 2.2): having explored the entire neighbourhood of a current solution without improvement, we know that the algorithm has reached a local optimum under the given neighbourhood and the best it can do is to continue to explore (i.e., restart) from a different initial solution. The new initial solution is not necessarily selected uniformly at random from the search space. Often, the new initial solution is obtained by making a larger perturbation of the current solution, which, by being larger than usual, may allow the search algorithm to escape that local optimum without needing to do a complete restart. This is the basic idea behind iterated local search (Stützle & Ruiz 2018), a quite effective method in practice.

There are, however, other popular and effective SLS algorithms that do not systematically explore the entire neighbourhood of a solution, e.g., Simulating Annealing, Great Deluge Algorithm, Flex-Deluge Algorithm, Threshold Accepting, and Late Acceptance Hill-Climbing, to name a few. The

proper estimation of the cutoff time of this type of SLS algorithm is non-trivial and is crucial.

This chapter proposes a method to make the restart decision based on theoretical grounds as opposed to an empirical rule-of-thumb. The method is based on results from the coupon collector's problem, a well-known problem from the probability theory. The coupon collector's problem has been used to analyse the behaviour of randomized algorithms (Motwani & Raghavan 1995). Herein we use it to design a scalable approach for choosing an appropriate cutoff time for one-point randomized SLS algorithms that can accept worsening moves. The proposed approach does not impose a computational expense on the search algorithm.

We illustrate the proposed cutoff time strategy with the LAHC algorithm. We test the proposed method on a range of common benchmark sets derived from three classical NP-hard problems, namely the Travelling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP), and the Permutation Flowshop Scheduling Problem (PFSP), using two different perturbation operators. Our experiments show that the proposed approach can firmly decide the cutoff time based on the problem instance size and perturbation operator employed. It also shows that the current solution obtained at the cutoff time is at a local optimum with high confidence.

4.2 The Coupon Collector's Problem

We present the coupon collector's problem (CCP) as described by Motwani & Raghavan (1995). In the coupon collector's problem, there are n coupon types and at each trial a coupon is chosen at random. Each coupon type is equally likely to be drawn. We are interested in the expected number of trials needed to collect at least one copy of each type of coupon. This problem is equivalent to an occupancy problem in which m balls are randomly distributed in n bins and we are interested in the expected number of trials to get at least one ball in every bin. Let X be a random variable denoting the number of trials needed to collect at least one copy of each type of coupon. The expected value of X is $E[X] = nH_n$, where $H_n = \sum_{i=1}^{n} 1/i$ is the nth Harmonic number which is asymptotically equal to $\ln n + O(1)$. A good approximation is given by $E[X] = n \ln n + \gamma n$, where $\gamma \approx 0.5777216$ is the Euler-Mascheroni constant.

Regarding the distribution of X, it can be shown that it is sharply concentrated around its mean value (Motwani & Raghavan 1995, p. 58). Asymptotically, the probability that X deviates from $n \ln n$ by a quantity cn, for any real constant c, is given by

$$\lim_{n \to \infty} Pr[n \ln n - cn \leq X \leq n \ln n + cn] = e^{-e^{-c}} - e^{-e^{c}} , \qquad (4.1)$$

which quickly approaches 1 as c increases. For the non-asymptotic case, for a constant $\beta > 1$ an upper tail bound is given by

$$Pr[X > \beta n \ln n] \leq n^{-\beta+1} . \qquad (4.2)$$

We shall use this last result to propose a cutoff time for one-point SLS algorithms that accept worsening moves.

4.2.1 Cutoff Time Strategy Based on the Coupon Collector's Problem

When a one-point randomized SLS algorithm is used to solve a combinatorial optimization problem, the current solution s is perturbed by a problem specific operator for that problem yielding a solution s'. The operator induces a neighbourhood $N(s)$ of a given size. For example, a common operator for the TSP is the 2-exchange move which removes two non-adjacent edges of a tour and reconnects the sub-tours by reversing one of them (see Fig. 2.3). In a problem with m cities, there are $m(m-3)/2$ possible 2-exchange moves and this is the size of the 2-exchange TSP neighbourhood. Obviously, different operators yield different neighbourhood sizes.

The state of a one-point randomized SLS algorithm can be described by the current solution that it maintains and by other information that it requires for its operation.

We now address the following question: how can we conclude that the search is stagnant? The answer to this question is simple: when there is strong evidence that subsequent operator moves will not change the *state of the algorithm*. Note that the state of a one-point randomized SLS algorithm can be described by the current solution that it maintains and by other information that it requires for its operation. The next question is: how can we detect such evidence? Again the answer is simple: when the

neighbourhood of the current solution has been (with high probability) fully explored without resulting in a change in the state of the algorithm. It turns out that the coupon collector's problem allows us to detect that evidence by equating the size of the neighbourhood to the number of coupons, and recognizing that the possible moves on a solution are equally likely.

Inequality 4.2 tells us that the right probability that the number of trials needed to fully explore a neighbourhood of size $|N(s)|$ is greater than $\beta\ |N(s)| \ln |N(s)|$ is at most $|N(s)|^{-\beta+1}$. By raising β this probability can be made arbitrarily small.

Let $p = 1 - |N(s)|^{-\beta+1}$ be the confidence that the neighbourhood of size $|N(s)|$ of a solution is fully explored upon $\beta\ |N(s)| \ln |N(s)|$ trials. Solving the previous equation for β gives us the desired value for a given confidence p.

$$\beta = 1 - \frac{\ln(1-p)}{\ln |N(s)|}\,. \tag{4.3}$$

The number of iterations needed to cutoff the algorithm would then be equal to $\theta = \beta\ |N(s)| \ln |N(s)|$ trials without changing the state of the algorithm, with β obtained from Equation 4.3 for a given confidence level p. We shall now illustrate how this idea can be used within the LAHC algorithm.

4.3 Stopping LAHC Using the Results of the Coupon Collector's Problem

Following what was presented in Subsection 4.2.1, we can use the coupon collector's problem to decide the cutoff time for the LAHC. There is, however, a subtlety: the cutoff time θ may never be reached because the state of the algorithm can change indefinitely due to plateau moves. As such, rather than counting the number of consecutive iteration steps where the state of the algorithm does not change, we count the number of consecutive iteration steps where neither the current solution has improved nor the history list was updated.

This number is used to decide at any given point in time whether the algorithm should stop. We can do so using a certain confidence that the entire neighbourhood of the current solution has been explored without affecting the state of the algorithm, plateau moves aside. As an example, for a neighbourhood size $|N(s)| = 1000$ and a confidence level $p = 0.95$, we

can use Equation 4.3 to obtain $\beta = 1.4337$ and the cutoff time would be $\theta = \beta\ |N(s)| \ln |N(s)| = 9903$ iterations.

Due to the specificities of LAHC, an extra condition needs to be fullfilled: $\theta \geq L_h$. Not fulfilling this condition would imply that the history list could still be updated in subsequent iterations had the algorithm not stopped. This extra condition is a direct consequence of the corollary stated at the end of Subsection 2.2.1. The cutoff time θ for LAHC is therefore given by the maximum of L_h and $\beta\ |N(s)| \ln |N(s)|$. We shall now present experiments to validate this strategy.

4.4 Experimental Setup

This and the following section present an extensive experimental evaluation of the proposed cutoff time strategy in order to validate its suitability and scalability. In this section, we describe the goals of the experiments, the benchmark problems, and the experimental setup. In the subsequent section, the results will be presented and discussed.

The cutoff strategy is illustrated with the LAHC algorithm. The experiments were programmed in C++ and executed using High Performance Computing provided by Queen Mary University of London. One should note that the aim of this work is not to introduce a new SLS algorithm or a new operator that performs better on the benchmark problems used in this study, but rather to test the new cutoff time based on CCP. The overall goals of our experiments are:

- To compare it with another dynamic cutoff time approach used by Burke & Bykov (2008, 2017), which is a common strategy of stopping the algorithm when the solution cost does not improve for a sufficiently long period of time: In their approach, the cutoff time is determined when the number of consecutive non-improving (idle) iterations over the total number of iterations done from the beginning of the search reaches a specified threshold (2% in their case) and at least a minimum number of iterations (100,000 in their case) are performed to avoid early termination. As an example, if the search has produced one million iterations from the beginning and during the last 20,000 iterations, there was no improvement, then the search is stopped.

- To investigate the scalability of the proposed approach based on the instance size: To accomplish this aim, besides applying it to several TSP instances ranging from 783 to 3795 cities, we also apply it to all QAP instances of QAPLIB, which have much smaller sizes (ranging from 10 to 256). We use the 2-exchange move as a perturbation operator for both problems. This operator induces a neighbourhood of size $m(m-3)/2$ for the case of TSP, and $\binom{m}{2} = m(m-1)/2$ for the case of QAP, with m being the number of cities/facilities of the TSP/QAP instance.

- To investigate the scalability of the proposed approach based on the perturbation operator, we also applied it to all Taillard (1993) instances of PFSP using a different perturbation operator, i.e., the insertion move. This operator induces a different neighbourhood size with respect to the 2-exchange move used for the TSP and QAP instances, namely $(m-1)^2$, with m being the number of jobs.

- To study the solidity of the method, we check all neighbours of the current solution at the cutoff time. Our experimental results confirm our expectation drawn from the CCP that at the cutoff time, the current solution is at local optimum with high probability.

Although we will only explicitly compare the CCP method with the 2% strategy, we will be implicitly comparing it with any fixed-based percentage strategy. The experiments that we are about to present show that a cutoff time based on a fixed-percentage of total time without improvement (no matter what that value is, 2% or any other) does not scale well across different problem classes, across problem instances within each class, and even across parameter settings for a given problem instance, as we shall see in Subsection 4.5.2.

4.4.1 Benchmark Problems

We are using three well-known combinatorial optimization problems to evaluate the performance of the proposed cutoff time. We perform experiments on 30 TSP instances taken from the well-known TSPLIB repository. Among the 30 instances are the 7 instances used by Burke & Bykov (2017). Subsection 2.1.1 provides details on TSP and TSPLIB. Remember that the number

of cities is indicated in the instance name; for example, rat783 is an instance with 783 cities.

The remaining of this subsection explains the other two problems and datasets used in our experiments.

Quadratic Assignment Problem (QAP)

The QAP is another well-studied NP-hard optimization problem (Sahni & Gonzalez 1976). Many real-world problems can be formulated as QAPs, namely campus layout (Dickey & Hopkins 1972), scheduling (Geoffrion & Graves 1976), and designing typewriter layout (Zhai, Hunter & Smith 2002), to name a few. It has been shown in the literature that they are one of the hardest combinatorial optimization problems (Hoos & Stützle 2005). To date, instances of size bigger than 30 cannot generally be solved using exact methods. The QAP can be described as the problem of assigning a set of m facilities to a set of m locations, with given flows between facilities and given distances between locations. The QAP aims to assign facilities to locations in such a way that every facility is assigned to exactly one location, every location is assigned to exactly one facility, and the sum of the products flow×distance is minimized. This can be formally stated as searching for a permutation ϕ that minimizes the function f defined as follows:

$$f(\phi) = \sum_{i=1}^{m} \sum_{j=1}^{m} a_{ij} b_{\phi(i)\phi(j)} \tag{4.4}$$

where ϕ is a permutation of the set of integers $\{1, \ldots, m\}$ denoting an assignment of the m facilities to the m locations. $f(\phi)$ denotes the cost of permutation ϕ. A flow of value a_{ij} has to go from facility i to facility j, $\phi(i)$ gives the location of facility i in ϕ. Thus, $b_{\phi(i)\phi(j)}$ is the cost of assigning facility i to location $\phi(i)$ and facility j to location $\phi(j)$, simultaneously. In this study, we use all QAP instances of the QAPLIB repository accessible at `http://anjos.mgi.polymtl.ca/qaplib/`. See Burkard, Karisch & Rendl (1997) for detailed information on QAPLIB. The instance sizes of QAPLIB varies from 10 to 256, and is indicated in the instance name. For example, `tai256c` has 256 facilities and locations. The QAPLIB repository contains 136 instances that can be broadly classified into four types as follows:

1. Real-world instances taken from practical applications of QAP.
2. Unstructured instances that are generated randomly according to a uniform distribution. These are among the hardest QAP instances to solve using exact methods.
3. Randomly generated instances with structure resembling a distribution found in real-world problems.
4. Instances with distance matrix based on the Manhattan distance on a grid.

Permutation Flowshop Scheduling Problem (PFSP)

The PFSP is one of the most thoroughly studied class of scheduling problems (Framinan, Leisten & García 2014). There are different variants of it. In the literature, the most commonly studied variant is the minimization of the total completion time of the last job (Fernandez-Viagas, Ruiz & Framinan 2017), i.e., the makespan C_{max}, and that is the variant that we use in this chapter. Formally, a PFSP instance is given by a set of n independent jobs $\{J_1, \ldots, J_n\}$, where each job J_j must be sequentially processed on a set of m machines $\{M_1, \ldots, M_m\}$. Each job J_j requires a given fixed non-negative processing time p_{ij} on each machine M_i. In the PFSP, all n jobs are to be processed on the m machines in the same machine order starting from M_1 and finishing on M_m. The objective of PFSP-Cmax is to find a job sequence ϕ that minimizes the makespan. For our experiments, we use the well-known standard benchmark set of Taillard (1993). It is composed of instances with number of jobs in $\{20, 50, 100\}$ for a number of machines in $\{5, 10, 20\}$, instances with 200 jobs for 10 and 20 machines, and instances with 500 jobs for 20 machines. For each combination of number of jobs and number of machines, Taillard provides 10 instances, so, in total the set contains 120 instances. They are accessible from Taillard's website `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html`. In the tables containing the experimental results of PFSP, the instances of one particular size are labeled with $n \times m$, with n being the number of jobs and m the number of machines. We also

carried out experiments with a more recent benchmark introduced by Vallada, Ruiz & Framinan (2015), a.k.a. VRF, which contains instances with up to 800 jobs and 60 machines.

4.4.2 Neighbourhood Structures

The perturbation operators that we employ are used quite often with search algorithms for these problems. For TSP and QAP, we use the 2-exchange move. In TSP it operates as follows: two non-adjacent edges are removed from a tour creating two sub-tours and subsequently one of the sub-tours is reconnected in reverse order. In QAP, it swaps the locations of two randomly selected facilities. As explained earlier in Subsection 4.2.1, the neighbourhood size of this operator is $m(m-3)/2$ for TSP and $\binom{m}{2} = m(m-1)/2$ for QAP, with m being the number of cities/facilities. The 2-exchange move for QAP is sometimes called the swap or interchange move. For PFSP, we use the insertion move as a perturbation operator, which consists of randomly picking job J_i at position i of the permutation and inserting it at position $j \neq i$, i.e., J_i can be inserted before or after position j. This operator is sometimes called the shift operator. The size of the insertion neighbourhood is $(m-1)^2$, with m being the number of jobs.

4.4.3 Algorithm Setup

We compare two cutoff strategies: the 2% of total search time without improvement in solution cost, with a minimum of 100,000 iterations, as undertaken previously in Burke & Bykov (2017), and the coupon collector's problem based strategy with a confidence $p = 0.95$ of having explored the entire neighbourhood of the current solution without changing the state of the algorithm, plateau moves aside, as explained earlier. Henceforth, we refer to these two strategies as the 2% strategy and the CCP strategy.

We note that the confidence p chosen for the CCP calculation has a clear mathematical meaning. We choose $p = 0.95$ in our experiments but, of course, other value could be used. The important thing is that for a given p the theory says that at the cutoff time the solution should be, on expectation, at a local optimum with a probability of at least p. Obviously, a better guarantee of stopping at a local optimum can be obtained by raising p closer (but never reaching) 100% at the expense of waiting a little longer. Note

that the distribution of the number of trials needed to collect all coupons on the CCP is sharply concentrated around its mean value. Take the example described earlier in Section 4.3. If the size of the neighbourhood is 1000 and $p = 0.95$, the resulting $\beta = 1.4337$. If instead we choose $p = 0.99$ for the same neighbourhood size we would get $\beta = 1.66666$ (just a small factor increase). A choice of $p = 0.9999$ would give $\beta = 2.33333$ (still a reasonable factor). Of course we cannot set $p = 1.0$ because that implies $\beta = \infty$.

For every problem instance, we carry out experiments with three history lengths, $L_h \in \{1, 5000, 50000\}$. These are the lengths used by Burke & Bykov (2017) in their original paper on LAHC. For every instance, the LAHC with a given history length was run 100 independent times, for both cutoff strategies.

To make the comparison as fair as possible, each of the 100 independent runs eventually stops (as specified by the cutoff strategy, either the 2% or the CCP strategy) and at that point in time we record several pieces of information that are tracked by our computer simulations. Subsequently, the run is resumed and continues until the other cutoff strategy tells the algorithm to stop. This way, the behaviour of the LAHC algorithm is exactly the same in both cases up to the point where one of them stops as dictated by the cutoff strategy; the only difference being that one of them will run for a longer period of time. By doing so we remove external factors that could blur the comparison between the two cutoff strategies.

When the algorithm stops, we record the total number of iterations performed by the algorithm and the best solution cost achieved up to that point in time. At the time the algorithm stops, we also visit all the neighbours of the current solution and count how many of those neighbours are better and how many have an equal cost to that of the current solution.

Initialization

The initial candidate solutions for TSP and QAP are generated uniformly at random. For these two problems, all the L_h elements of the history list of LAHC are initialized with the cost value of the initial solution.

For the PFSP, the well-known construction heuristic NEH (Nawaz, Enscore & Ham 1983) is used to generate the initial solution. Most recent metaheuristic algorithms for PFSP rely on NEH (algorithm of Nawaz, Enscore

& Ham) initialization (Ruiz & Maroto 2005, Ruiz & Stützle 2007, Taillard 1990). NEH can be described by the following three steps: 1) sort the m jobs by descending order of the sums of processing times on the machines; 2) take the first two jobs and schedule them (from two possible schedules) to minimize the partial makespan; 3) take the remaining jobs one at the time and find the best schedule position for them in the sequence of jobs that are already scheduled to minimize the partial makespan. It has been shown in the literature that the NEH heuristic has the best performance for a wide variety of problem instances (Taillard 1990, Ruiz & Stützle 2007). NEH has a complexity of $O(m^3r)$, which can be problematic for large problem instances. However, Taillard reduced its complexity to $O(m^2r)$ by using a new data structure (see Taillard (1990) for details). We use Taillard's method in our implementation of NEH. Since the initial solution generated by NEH is already very good, initializing the history list of LAHC with the NEH solution cost is not a good idea because LAHC would degenerate quickly into a simple hill-climbing algorithm. Note that LAHC benefits from accepting worsening moves. Thus initializing the history list with very good cost values holds back LAHC from accepting worsening moves right from the start. In order to avoid this, we initialize all the L_h elements of the history list of LAHC with a cost value of a randomly generated solution. This complies with Burke & Bykov (2017) proposition on initializing the history list of LAHC. Although in the original paper of LAHC, they initialized the elements of the history list with the cost of the initial solution, they also noted that the list can be initialized with other values, if necessary.

4.5 Experimental Results

The LAHC algorithm was applied to 30 TSP instances taken from TSPLIB, to all 136 QAP instances contained in QAPLIB, to all 120 PFSP Taillard's instances, and to all 480 PFSP instances of the more recent benchmark introduced by Vallada, Ruiz & Framinan (2015). In this chapter, due to space restrictions, we present here a subset of the experiments. For TSP, we show the results of 7 instances, the exact same ones used by Burke & Bykov (2017) in their study. For QAP, we present 17 Taillard instances of size larger than 30. These are among the most difficult QAPLIB instances (Ochoa & Herrmann 2018). For PFSP, we report the results of 12 instances,

one for each combination of number of jobs and number of machines found in Taillard's benchmark. Notwithstanding, results for all problem instances are available in a companion website of this work at `https://mbazargani.github.io/CCPcutoffTime/` and also as a 79-page PDF document at `https://bit.ly/3foVjud`.

4.5.1 Comparison of the CCP and the 2% Cutoff time Strategy

Table 4.1 shows the results obtained for the three problem classes and for the three settings of the history length $L_h \in \{1, 5000, 50000\}$. Recall that 100 independent runs were made for each combination of problem instance and L_h value. Table 4.1 reports how many of those runs reached a better, equal, or worse solution cost (labeled as $<$, $=$, or $>$), when using the CCP as opposed to the 2% cutoff strategy. It also reports on the number of runs for which the CCP strategy took less or more iterations to stop than the 2% strategy (labeled as $<$ or $>$). As an example, consider the first TSP instance `rat783` with $L_h = 5000$. The table entries say that in 32 out of 100 runs, the CCP strategy lead to a better solution cost than that provided by the 2% strategy, and in the remaining 68 runs the solution cost obtained at the cutoff time was the same for both strategies. Regarding the number of iterations, the CCP strategy allowed the algorithm to run for a longer period of time on all of the 100 runs.

For completeness, Tables 4.2, 4.3, and 4.4, show the Average Relative Percentage Deviation ($ARPD$) from the optimal or best-known solutions, and the number of iterations spent, averaged over the 100 independent runs for the TSP, QAP, and PFSP instances, respectively. The $\overline{RPD}$ is calculated as:

$$\sum_{i=1}^{r}((Sol_i - Best_{sol})/Best_{sol})/r, \tag{4.5}$$

where Sol_i is the solution quality obtained in the i^{th} run, and $Best_{sol}$ is either the optimal solution or the best-known solution for each given instance. The Wilcoxon signed-rank test is used to assess if the differences obtained in solution qualities and number of iterations are statistically significant. We considered the results to be statistically significant for a p-value

Table 4.1: Summary of results obtained for the three problem classes and for the three settings of the history length $L_h \in \{1, 5000, 50000\}$. 100 independent runs were made for each combination of problem instance and L_h value. The table reports how many of those runs the CCP strategy reached a better, equal, or worse solution cost, than the 2% cutoff strategy (labeled as $<$, $=$, or $>$, under the Cost columns). It also reports on the number of runs for which the CCP strategy took less or more iterations to stop than the 2% strategy (labeled as $<$ or $>$, under Iterations). The CCP cutoff time is calculated using a confidence level $p = 0.95$.

TSP

Dataset	$L_h = 1$					$L_h = 5000$					$L_h = 50000$				
	Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)		Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)		Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)	
	$<$	$=$	$>$	$<$	$>$	$<$	$=$	$>$	$<$	$>$	$<$	$=$	$>$	$<$	$>$
rat783	100	0	0	0	100	32	68	0	0	100	0	100	0	100	0
u1060	100	0	0	0	100	67	33	0	0	100	4	96	0	0	100
fl1400	100	0	0	0	100	99	1	0	0	100	15	85	0	0	100
u1817	100	0	0	0	100	96	4	0	0	100	27	73	0	0	100
d2103	100	0	0	0	100	86	14	0	0	100	7	93	0	0	100
pcb3038	100	0	0	0	100	78	22	0	0	100	3	97	0	0	100
fl3795	100	0	0	0	100	100	0	0	0	100	85	15	0	0	100

QAP

Dataset	$L_h = 1$					$L_h = 5000$					$L_h = 50000$				
	Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)		Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)		Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)	
	$<$	$=$	$>$	$<$	$>$	$<$	$=$	$>$	$<$	$>$	$<$	$=$	$>$	$<$	$>$
tai30a	0	100	0	100	0	0	100	0	99	1	0	100	0	100	0
tai30b	0	100	0	100	0	0	100	0	100	0	0	100	0	100	0
tai35a	0	100	0	100	0	0	100	0	100	0	0	100	0	100	0
tai35b	0	100	0	100	0	0	100	0	100	0	0	100	0	100	0
tai40a	0	100	0	100	0	0	100	0	99	1	0	100	0	100	0
tai40b	0	99	1	100	0	0	100	0	100	0	0	100	0	100	0
tai50a	0	100	0	100	0	0	100	0	72	28	0	100	0	100	0
tai50b	0	100	0	100	0	0	100	0	100	0	0	100	0	100	0
tai60a	0	100	0	100	0	0	100	0	1	99	0	100	0	100	0
tai60b	0	100	0	100	0	0	100	0	100	0	0	100	0	100	0
tai64c	0	100	0	100	0	10	90	0	0	100	0	100	0	0	100
tai80a	0	100	0	98	2	1	99	0	0	100	0	100	0	100	0
tai80b	0	100	0	99	1	0	100	0	0	100	0	100	0	100	0
tai100a	2	98	0	46	54	1	99	0	0	100	0	100	0	100	0
tai100b	5	95	0	18	82	0	100	0	0	100	0	100	0	100	0
tai150b	95	5	0	0	100	0	100	0	0	100	0	100	0	100	0
tai256c	85	15	0	0	100	81	19	0	0	100	0	100	0	0	100

PFSP

Dataset	$L_h = 1$					$L_h = 5000$					$L_h = 50000$				
	Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)		Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)		Cost (CCP vs. 2%)			Iterations (CCP vs. 2%)	
	$<$	$=$	$>$	$<$	$>$	$<$	$=$	$>$	$<$	$>$	$<$	$=$	$>$	$<$	$>$
tai001 – 020×05	0	33	67	100	0	9	87	4	14	86	0	100	0	0	100
tai011 – 020×10	0	84	16	100	0	2	98	0	0	100	0	100	0	1	99
tai021 – 020×20	0	91	9	100	0	2	98	0	0	100	0	100	0	1	99
tai031 – 050×05	0	86	14	100	0	13	87	0	0	100	0	100	0	0	100
tai041 – 050×10	1	52	47	98	2	31	69	0	0	100	0	95	5	100	0
tai051 – 050×20	2	75	23	93	7	33	67	0	0	100	0	94	6	100	0
tai061 – 100×05	0	100	0	0	100	20	80	0	0	100	0	100	0	0	100
tai071 – 100×10	46	54	0	0	100	60	40	0	0	100	1	99	0	24	76
tai081 – 100×20	80	20	0	0	100	72	28	0	0	100	0	75	25	100	0
tai091 – 200×10	71	29	0	0	100	69	31	0	0	100	48	52	0	0	100
tai101 – 200×20	100	0	0	0	100	95	5	0	0	100	25	75	0	0	100
tai111 – 500×20	100	0	0	0	100	100	0	0	0	100	83	17	0	0	100

< 0.05, and typeset those in boldface in the tables. Some of the $\overline{RPD}$ values displayed in Tables 4.2, 4.3, and 4.4 are identical when rounded to two decimal places, but there is still a statistical significance difference between them.

Obviously, the strategy that allows the algorithm to run longer can never reach a worse solution cost than the other. We shall now discuss the obtained results separately for each problem class.

TSP

For the TSP instances, it can be observed that the CCP strategy allowed the algorithm to run longer except in one case, the `rat783` instance with $L_h = 50000$. It can also be observed that the solution cost obtained at the CCP cutoff time was never worse than the solution cost obtained at the 2% cutoff strategy. For $L_h = 1$ it was always better, and for $L_h \in \{5000, 50000\}$ it was either better or equal. In the sole case where the 2% strategy ran for a longer period of time (`rat783` with $L_h = 50000$), the CCP strategy never got to a worse solution cost.

Overall, the results obtained with the TSP instances show that the 2% strategy was stopping the runs prematurely in several cases, not allowing the algorithm to reach its full capacity.

QAP

Regarding the QAP instances, it can be observed that the 2% strategy often allows the algorithm to run longer than the CCP strategy (sometimes by a factor of more than 10). However, running it longer did not result in a improved solution cost except in one single run out of the 100 independent runs performed on the `tai40b` instance with $L_h = 1$. In contrast, for the cases where the CCP strategy ran longer, the solution cost was often better. This is especially clear for the `tai150b` and `tai256c` instances.

The 2% strategy reveals a drawback here. Recall that this strategy always performs a minimum number of iterations, specified by a parameter, which was set here to be 100,000 iterations. This explains why the 2% strategy ran much longer than the CCP strategy on the smaller sized instances

Table 4.2: Results obtained by LAHC on seven TSP instances with both stopping criteria, CCP and 2% of total search time, using $L_h \in \{1, 5000, 50000\}$. The CCP cutoff time is calculated using a confidence level $p = 0.95$. The results are averaged over 100 independent runs. Entries in boldface are statistically significant with a p-value < 0.05 according to the Wilcoxon signed-rank test.

Dataset	Stopping Criterion	$L_h = 1$		$L_h = 5000$		$L_h = 50000$	
		$\overline{RPD}$	Iters. (10^3)	$\overline{RPD}$	Iters. (10^3)	$\overline{RPD}$	Iters. (10^3)
rat783	2%	0.24	**774**	0.06	**28195**	0.03	259162
	ccp	**0.13**	8295	**0.06**	33005	0.03	**258750**
u1060	2%	0.17	**2252**	0.05	**43576**	0.02	**388940**
	ccp	**0.14**	17617	**0.05**	54757	0.02	390672
fl1400	2%	0.12	**3954**	0.03	**57279**	0.01	**491071**
	ccp	**0.08**	50783	**0.03**	92089	**0.01**	500390
u1817	2%	0.21	**8297**	0.09	**92454**	0.04	**752937**
	ccp	**0.17**	73853	**0.09**	146712	**0.04**	774101
d2103	2%	0.23	**12318**	0.11	**111923**	0.07	**879397**
	ccp	**0.20**	80799	**0.11**	164602	**0.07**	903109
pcb3038	2%	0.16	**28181**	0.09	**175972**	0.05	**1340996**
	ccp	**0.15**	143671	**0.09**	273101	0.05	1400232
fl3795	2%	0.15	**61248**	0.08	**266016**	0.05	**1812087**
	ccp	**0.12**	523161	**0.08**	678023	**0.05**	2142779

with $L_h = 1$. Indeed, one can observe that for the smaller list sizes it almost always stopped immediately after completing the 100,000 iterations.

We should highlight that the value of 100,000 was not tuned in this case; we simply use the same value that Burke & Bykov (2017) suggested in their original LAHC work. Obviously, the 100,000 value was tuned and was appropriate for the TSP instances, but not so for the QAP case.

As in the TSP case, the CCP strategy stops the algorithm at an appropriate moment, neither too soon nor too late.

PFSP

We shall now discuss the results obtained with the PFSP instances. Similarly to the QAP case, the 2% strategy let the algorithm run for a much longer period of time (sometimes by a factor of more than 20) when applied to the smaller sized instances with $L_h = 1$, again due to the minimum number of 100,000 iterations it requires before stopping. In this case, however, a different scenario emerged compared to the QAP case. While in the QAP case, running it longer was not beneficial, in this case it was. Thus, it seems

Table 4.3: Results obtained by LAHC on 17 of Taillard's instances of QAP of size larger than 30 taken from QAPLIB with both stopping criteria, CCP and 2% of total search time, using $L_h \in \{1, 5000, 50000\}$. The CCP cutoff time is calculated using a confidence level $p = 0.95$. The results are averaged over 100 independent runs. Entries in boldface are statistically significant with a p-value < 0.05 according to the Wilcoxon signed-rank test.

Dataset	Stopping Criterion	$L_h = 1$		$L_h = 5000$		$L_h = 50000$	
		$\overline{RPD}$	Iters. (10^3)	$\overline{RPD}$	Iters. (10^3)	$\overline{RPD}$	Iters. (10^3)
tai30a	2%	0.05	100	0.03	335	0.02	3443
	ccp	0.05	**6**	0.03	**333**	0.02	**3424**
tai30b	2%	0.13	100	0.03	427	0.03	4435
	ccp	0.13	**7**	0.03	**424**	0.03	**4396**
tai35a	2%	0.05	100	0.03	406	0.02	4121
	ccp	0.05	**9**	0.03	**404**	0.02	**4088**
tai35b	2%	0.08	100	0.02	526	0.02	5395
	ccp	0.08	**10**	0.02	**521**	0.02	**5337**
tai40a	2%	0.05	100	0.03	480	0.02	5004
	ccp	0.05	**13**	0.03	**478**	0.02	**4954**
tai40b	2%	0.10	100	0.02	640	0.02	6538
	ccp	0.10	**13**	0.02	**634**	0.02	**6458**
tai50a	2%	0.05	100	0.03	645	0.03	6621
	ccp	0.05	**21**	0.03	**645**	0.03	**6539**
tai50b	2%	0.07	100	0.01	860	0.00	8709
	ccp	0.07	**23**	0.01	**855**	0.00	**8585**
tai60a	2%	0.04	100	0.03	**809**	0.03	8292
	ccp	0.04	**33**	0.03	811	0.03	**8176**
tai60b	2%	0.06	100	0.01	1102	0.00	11147
	ccp	0.06	**35**	0.01	**1099**	0.00	**10974**
tai64c	2%	0.01	100	0.00	**132**	0.00	**1429**
	ccp	0.01	**27**	**0.00**	158	0.00	1485
tai80a	2%	0.04	100	0.03	**1142**	0.02	11874
	ccp	0.04	**64**	0.03	1155	0.02	**11687**
tai80b	2%	0.05	100	0.01	**1592**	0.01	16081
	ccp	0.05	**69**	0.01	1595	0.01	**15810**
tai100a	2%	0.04	100	0.03	**1534**	0.02	15754
	ccp	0.04	104	0.03	1563	0.02	**15496**
tai100b	2%	0.05	**100**	0.00	**2144**	0.00	21482
	ccp	**0.05**	115	0.00	2158	0.00	**21109**
tai150b	2%	0.03	**105**	0.01	**3576**	0.00	35435
	ccp	**0.03**	318	0.01	3642	0.00	**34864**
tai256c	2%	0.01	**100**	0.00	**789**	0.00	**8187**
	ccp	**0.00**	608	**0.00**	1358	0.00	8466

that CCP strategy makes an immature call to stop the algorithm for the smaller sized instances (the first six small instances from `tai001–020×05` to `tai051–050×20`) when using the history list length $L_h = 1$.

There is an explanation for this behaviour which is related to the existence of a sequence of plateau moves in many of these PFSP instances that

Table 4.4: Results obtained by LAHC on 12 of Taillard's PFSP instances with both stopping criteria, CCP and 2% of total search time, using $L_h \in \{1, 5000, 50000\}$. The CCP cutoff time is calculated using a confidence level $p = 0.95$. The results are averaged over 100 independent runs. Entries in boldface are statistically significant with a p-value < 0.05 according to the Wilcoxon signed-rank test.

Dataset	NEH	Stopping	$L_h = 1$		$L_h = 5000$		$L_h = 50000$	
	RPD	Criterion	$\overline{RPD}$	Iters. (10^3)	$\overline{RPD}$	Iters. (10^3)	$\overline{RPD}$	Iters. (10^3)
tai001 – 020×05	0.01	2%	**0.00**	100	0.00	**115**	0.00	**1403**
		ccp	0.00	**4**	0.00	120	0.00	1524
tai011 – 020×10	0.06	2%	**0.01**	100	0.01	**209**	0.00	**2171**
		ccp	0.01	**5**	0.00	211	0.00	2189
tai021 – 020×20	0.05	2%	**0.02**	100	0.01	**196**	0.00	**2109**
		ccp	0.02	**5**	0.01	200	0.00	2130
tai031 – 050×05	0.00	2%	**0.00**	100	0.00	**147**	0.00	**1610**
		ccp	0.00	**28**	**0.00**	182	0.00	1657
tai041 – 050×10	0.06	2%	**0.03**	100	0.03	**386**	**0.02**	4819
		ccp	0.04	**40**	**0.03**	438	0.02	**4705**
tai051 – 050×20	0.07	2%	**0.05**	100	0.04	**536**	**0.04**	5819
		ccp	0.05	**58**	**0.04**	579	0.04	**5722**
tai061 – 100×05	0.00	2%	0.00	**100**	0.00	**180**	0.00	**1951**
		ccp	0.00	133	**0.00**	317	0.00	2039
tai071 – 100×10	0.02	2%	0.01	**100**	0.01	**514**	0.00	**5773**
		ccp	**0.01**	221	**0.00**	746	0.00	5780
tai081 – 100×20	0.08	2%	0.05	**100**	0.05	**884**	**0.03**	10785
		ccp	**0.04**	335	**0.04**	1178	0.03	**10360**
tai091 – 200×10	0.01	2%	0.00	**100**	0.01	**399**	0.01	**4551**
		ccp	**0.00**	788	**0.00**	1548	**0.00**	6022
tai101 – 200×20	0.05	2%	0.03	**100**	0.03	**1239**	0.02	**13931**
		ccp	**0.02**	1390	**0.02**	2520	**0.02**	14549
tai111 – 500×20	0.03	2%	0.03	**100**	0.02	**2172**	0.01	**24049**
		ccp	**0.01**	12574	**0.01**	12210	**0.01**	32159

lead to an improved solution cost. The reader may wonder why the same behaviour was not observed in the QAP experiments. Indeed, there are also plateau moves in some of the QAP instances that we tested, but it turned out that the sequence of such moves did not allow the algorithm to improve any further within the minimum 100,000 iterations. As explained earlier in the paper, a plateau move changes the state of the LAHC algorithm but the CCP strategy ignores such moves because it could lead the algorithm to never halt. We shall address this issue in Section 4.6.

Regarding the experiments with $L_h \in \{5000, 50000\}$, the CCP strategy almost always got to an equal or better solution cost than the 2% strategy.

To the best of our knowledge, it is the first time that a construction

solution (in this case, obtained by NEH) is used in the LAHC and the history list is initialised with the cost of a random solution. It is stated in the literature that NEH provides a very good solution and afterwards the mean improvement is less than 1% (Taillard 1990). In our experiments on over 120 of Taillard's instances of PFSP with $L_h = 50000$, using the average of 100 independent runs, we improved the NEH solutions by 3.1%, and considering the best out of 100 runs, the percentage is up to 3.5%.

Although in this work we do not aim to come with a method to reach a better solution cost, it is worth mentioning that in 82% of QAPLIB instances (111 out of 136) the LAHC with history length of 50000 using the CCP cutoff time reached the best-known solutions. For PFSP, this rate was 43% for Taillard's instances (51 out of 120), and 41% for VRF's instances (196 out of 480). As a matter of fact, for VRF's instances we improved the best-known solutions on 15% of the instances (71 out of 480). These results are typeset in boldface and colored blue in the supplementary material.

Our reference for the previous best solutions for the VRF benchmark are combined from the online materials of Vallada, Ruiz & Framinan (2015) and the recent paper from Pagnozzi & Stützle (2019).

4.5.2 Percentage of the Total Search Time for the CCP Cut-off Strategy

It is interesting to look at the CCP strategy through the lens of Burke & Bykov's dynamic strategy of a certain percentage of total time without improvement. To do so, we computed, for the CCP strategy, the amount of time —percentage-wise in terms of total search time— that it was idle before deciding to halt the algorithm. The results are shown in Table 4.5. Note that by definition, producing a similar table for the 2% strategy would yield 2% for all entries, except for those cases that correspond to stopping precisely when 100,000 iterations have been reached.

For all problem instances with $L_h = 1$, the CCP strategy performed much more than 2% of the total search time without improvement before it reached the cutoff time. However, on 13 out of the 17 QAP instances, and on 6 out of the 12 PFSP instances , the algorithm ran for a shorter period of time than with the 2% strategy, since the latter one takes effect after a minimum of 100,000 iterations have elapsed (see number of iterations used by these two stopping criterion when $L_h = 1$ in Tables 4.3 and 4.4).

Table 4.5: The CCP cutoff time for TSP, QAP, and PFSP instances with confidence level $p = 0.95$, as a percentage of the total search time. Results are the average over 100 independent runs.

TSP

Dataset	$L_h = 1$	$L_h = 5000$	$L_h = 50000$
rat783	58.45%	14.47%	1.84%
u1060	53.10%	16.65%	2.33%
fl1400	33.96%	18.04%	3.28%
u1817	40.56%	19.73%	3.69%
d2103	49.19%	23.74%	4.30%
pcb3038	59.34%	31.04%	6.04%
fl3795	27.27%	20.38%	6.34%

QAP

Dataset	$L_h = 1$	$L_h = 5000$	$L_h = 50000$
tai30a	67.97%	1.52%	1.48%
tai30b	59.28%	1.18%	1.14%
tai35a	64.81%	1.40%	1.23%
tai35b	58.77%	1.07%	0.94%
tai40a	62.72%	1.59%	1.02%
tai40b	58.47%	1.19%	0.78%
tai50a	61.28%	1.93%	0.77%
tai50b	56.16%	1.45%	0.58%
tai60a	58.10%	2.30%	0.61%
tai60b	53.52%	1.69%	0.46%
tai64c	79.91%	13.60%	3.38%
tai80a	57.23%	3.04%	0.43%
tai80b	51.98%	2.19%	0.32%
tai100a	56.25%	3.66%	0.37%
tai100b	50.62%	2.64%	0.27%
tai150b	44.45%	3.78%	0.39%
tai256c	72.77%	32.37%	5.17%

PFSP

Dataset	$L_h = 1$	$L_h = 5000$	$L_h = 50000$
tai001 – 020×05	85.42%	4.29%	3.29%
tai011 – 020×10	64.39%	2.38%	2.30%
tai021 – 020×20	70.50%	2.53%	2.37%
tai031 – 050×05	93.29%	14.36%	3.03%
tai041 – 050×10	74.61%	6.12%	1.07%
tai051 – 050×20	50.75%	4.49%	0.88%
tai061 – 100×05	90.40%	37.74%	5.86%
tai071 – 100×10	60.00%	16.38%	2.07%
tai081 – 100×20	41.33%	10.32%	1.16%
tai091 – 200×10	72.31%	37.75%	9.21%
tai101 – 200×20	42.29%	22.11%	3.71%
tai111 – 500×20	34.89%	35.07%	12.10%

What stands out from Table 4.5 is that the cutoff time specified by the

CCP strategy corresponds to having different percentages of consecutive "idle time" over total running time, and it does so automatically depending on a problem instance basis without the need of any tuning. Another important observation that can be made is that no fixed percentage value is ideal for all problem classes and problem instances. If instead of 2% we chose another value, say 3%, the resulting cutoff point would still be inadequate.

4.5.3 How Good is the Current Solution at the Cutoff Point?

To further validate this work, we performed an analysis of the neighbourhood of the current solution at the cutoff point, for both strategies. Since our experiments used a confidence value $p = 0.95$ in Equation 4.3 to determine the number of iterations required to visit the entire neighbourhood of a solution without changing the state of the algorithm, plateau moves aside, we should expect the current solution to be a local optimum with a probability of at least 95% at the cutoff point determined by the CPP strategy. Regarding the 2% strategy no such guarantee can be given.

Table 4.6: Analysis over the neighbours of the current solution obtained by LAHC on TSP instances, for both cutoff strategies. I_{move} denotes the average number of improving moves over 100 independent runs at the cutoff time. I_{max} denotes the maximum number of improving moves in a single run out of 100 independent runs. *Local Optimum* denotes the percentage of runs where the current solution at the cutoff point was at a local optimum. Entries in boldface are statistically significant with a p-value < 0.05 according to the Wilcoxon signed-rank test.

Dataset	Stopping Criterion	$L_h = 1$			$L_h = 5000$			$L_h = 50000$		
		I_{move}	I_{max}	Local Optimum	I_{move}	I_{max}	Local Optimum	I_{move}	I_{max}	Local Optimum
rat783	2%	124.10	1582	0%	0.30	2	76%	0.00	0	100%
	ccp	**0.00**	0	100%	**0.00**	0	100%	0.00	0	100%
u1060	2%	54.10	294	0%	0.66	5	61%	0.00	0	100%
	ccp	**0.00**	0	100%	**0.02**	1	98%	0.01	1	99%
fl1400	2%	54.13	251	0%	1.47	5	34%	0.06	1	94%
	ccp	**0.02**	1	98%	**0.03**	1	97%	0.03	1	97%
u1817	2%	43.86	194	0%	1.29	5	36%	0.05	1	95%
	ccp	**0.01**	1	99%	**0.01**	1	99%	0.04	1	96%
d2103	2%	39.83	196	0%	1.45	7	31%	0.01	1	99%
	ccp	**0.00**	0	100%	**0.00**	0	100%	0.00	0	100%
pcb3038	2%	28.50	110	0%	1.87	11	30%	0.01	1	99%
	ccp	**0.00**	0	100%	**0.00**	0	100%	0.00	0	100%
fl3795	2%	22.23	76	0%	3.16	13	15%	0.42	3	68%
	ccp	**0.07**	2	94%	**0.06**	2	95%	**0.03**	1	97%

To verify this, we enumerated the entire neighbourhood of the current solution at the cutoff time and counted how many of those neighbours were better than the current solution, i.e., how many immediate improving moves could be made upon the current solution. Tables 4.6, 4.7, and 4.8 show the results obtained for the TSP, QAP, and PSFP instances, respectively. The results were collected from the 100 independent runs conducted for each instance for both cutoff strategies. The column labeled I_{move} refers to the average number of improving moves at the cutoff time, over the 100 independent runs. The column labeled I_{max} refers to the maximum number of improving moves observed in a single run, and the column labeled *Local Optimum* refers to the percentage of runs that the current solution was at a local optimum at the cutoff time.

The results confirm, once again, that the 2% strategy often stops the search prematurely, with the current solution not being a local optimum at the cutoff time. In contrast, the CCP strategy stops the algorithm at the correct moment with the current solution at the cutoff time being at a local optimum with at least 95% probability on all instances except in 2 out of a total of 108 cases: TSP instance `fl3795` with $L_h = 1$, and PFSP instance `tai041-050x10` with $L_h = 5000$, where the observed probabilities were 94% and 92%, respectively, and not that far from 95%.

On the TSP instances, it is clear that the 2% strategy stops the search prematurely, especially for the smaller list sizes $L_h = 1$ and $L_h = 5000$.

On the QAP instances, the CCP strategy always reached a local optimum at the cutoff time, except in `tai40b` with $L_h = 1$ where it was reached in 99% of the runs. Note that on the small instances with $L_h = 1$, as shown in Tables 4.1 and 4.3, the 2% strategy executed many more iterations than the CCP strategy, but yet, the CCP strategy fairly accurately made a correct decision about the cutoff time. For the larger history lengths of 5000 and 50000, not even in a single table entry, the CCP failed to deliver a 100% local optimum rate. This rate varies from 13% to 100% for the 2% strategy.

On the PFSP instances, the local optimum rate reached by the 2% strategy varies from 10% to 100%. For the CCP strategy, it was always at least 95%, except in one case where it was 92%. This is normal as we are running a stochastic algorithm, and even in that one case the CCP strategy reached a better C_{max} value than the 2% strategy.

This analysis confirms once again how well and how consistently the

Table 4.7: Analysis over the neighbours of the current solution obtained by LAHC on QAP instances, for both cutoff strategies. I_{move} denotes the average number of improving moves over 100 independent runs at the cutoff time. I_{max} denotes the maximum number of improving moves in a single run out of 100 independent runs. *Local Optimum* denotes the percentage of runs where the current solution at the cutoff point was at a local optimum. Entries in boldface are statistically significant with a p-value < 0.05 according to the Wilcoxon signed-rank test.

Dataset	Stopping Criterion	$L_h = 1$			$L_h = 5000$			$L_h = 50000$		
		I_{move}	I_{max}	Local Optimum	I_{move}	I_{max}	Local Optimum	I_{move}	I_{max}	Local Optimum
tai30a	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai30b	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai35a	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai35b	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai40a	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai40b	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.01	1	99%	0.00	0	100%	0.00	0	100%
tai50a	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai50b	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai60a	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai60b	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai64c	2%	0.00	0	100%	0.46	4	72%	0.00	0	100%
	ccp	0.00	0	100%	**0.00**	0	100%	0.00	0	100%
tai80a	2%	0.00	0	100%	0.01	1	99%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai80b	2%	0.00	0	100%	0.00	0	100%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai100a	2%	0.04	2	98%	0.01	1	99%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai100b	2%	0.05	1	95%	0.00	0	100%	0.00	0	100%
	ccp	**0.00**	0	100%	0.00	0	100%	0.00	0	100%
tai150b	2%	6.71	21	5%	0.00	0	100%	0.00	0	100%
	ccp	**0.00**	0	100%	0.00	0	100%	0.00	0	100%
tai256c	2%	2.29	10	15%	2.74	18	13%	0.02	1	98%
	ccp	**0.00**	0	100%	**0.00**	0	100%	0.00	0	100%

CCP strategy can make an instance-based decision about the cutoff time.

Table 4.8: Analysis over the neighbours of the current solution obtained by LAHC on PFSP instances, for both cutoff strategies. I_{move} denotes the average number of improving moves over 100 independent runs at the cutoff time. I_{max} denotes the maximum number of improving moves in a single run out of 100 independent runs. *Local Optimum* denotes the percentage of runs where the current solution at the cutoff point was at a local optimum. Entries in boldface are statistically significant with a p-value < 0.05 according to the Wilcoxon signed-rank test.

Dataset	Stopping Criterion	$L_h = 1$			$L_h = 5000$			$L_h = 50000$		
		I_{move}	I_{max}	Local Optimum	I_{move}	I_{max}	Local Optimum	I_{move}	I_{max}	Local Optimum
tai001 – 020×05	2%	**0.00**	0	100%	0.01	1	99%	0.00	0	100%
	ccp	0.06	2	95%	0.01	1	99%	0.00	0	100%
tai011 – 020×10	2%	0.00	0	100%	0.01	1	99%	0.00	0	100%
	ccp	0.03	1	97%	0.01	1	99%	0.00	0	100%
tai021 – 020×20	2%	0.00	0	100%	0.04	2	97%	0.00	0	100%
	ccp	0.00	0	100%	0.00	0	100%	0.00	0	100%
tai031 – 050×05	2%	0.01	1	99%	0.11	3	94%	0.00	0	100%
	ccp	0.00	0	100%	**0.00**	0	100%	0.00	0	100%
tai041 – 050×10	2%	0.19	19	99%	0.05	2	96%	0.01	1	99%
	ccp	0.04	3	98%	0.16	8	92%	0.03	3	99%
tai051 – 050×20	2%	0.07	2	95%	0.14	4	92%	0.00	0	100%
	ccp	0.05	2	97%	**0.00**	0	100%	0.00	0	100%
tai061 – 100×05	2%	0.00	0	100%	0.52	6	62%	0.01	1	99%
	ccp	0.00	0	100%	**0.00**	0	100%	0.00	0	100%
tai071 – 100×10	2%	0.18	5	89%	0.09	2	92%	0.01	1	99%
	ccp	**0.02**	1	98%	0.02	1	98%	0.00	0	100%
tai081 – 100×20	2%	0.61	14	76%	0.52	8	79%	0.00	0	100%
	ccp	**0.05**	2	96%	**0.01**	1	99%	0.00	0	100%
tai091 – 200×10	2%	0.85	12	71%	0.80	16	65%	0.05	1	95%
	ccp	**0.02**	1	98%	**0.00**	0	100%	0.01	1	99%
tai101 – 200×20	2%	6.03	143	50%	1.20	21	70%	0.01	1	99%
	ccp	**0.04**	1	96%	**0.01**	1	99%	0.01	1	99%
tai111 – 500×20	2%	66.14	3556	10%	11.74	808	44%	0.22	9	89%
	ccp	**0.01**	1	99%	**0.02**	1	98%	**0.00**	0	100%

4.6 Discussion

Before finishing this chapter we would like to present and discuss two criticisms that can be made of the CCP strategy.

Not handling plateau moves. In the LAHC, plateau moves change the state of the algorithm because the resulting solution is accepted, but the CCP method considers them as an idle move, not reseting the idle iteration count. As explained earlier in this chapter, this decision is needed because not doing so would imply that the CCP cutoff time

could potentially never be reached. Note that if there is a plateau move from solution A to solution B, then A and B are neighbours, and it would be very likely that a subsequent plateau move would occur from B to A within the cutoff time period. This situation could continue indefinitely, and that would be even more likely in the presence of multiple plateau moves from a solution.

While this can be seen as a limitation of the method, we note that the same occurs with other commonly used cutoff methods, such as the 2% strategy, or any other fixed-percentage strategy.

With the CCP method, we can be confident that the current solution at the cutoff time is a local optimum with high probability, something that cannot be stated with a fixed-percentage based strategy. Furthermore, we can say that it is unlikely that the algorithm improves its solution cost if we let it run beyond the cutoff time, unless there is a sequence of plateau moves that can lead the algorithm to escape from that local optimum.

On performing a systematic enumeration of the neighbourhood.

A systematic enumeration of the neighbourhood refers to visit neighbours of the current solution without replacement. Thus, one could say that the cutoff time derived by the CCP could be avoided if the perturbations on the current solution were done in a systematic manner avoiding visiting the same neighbour more than once. This would result in a cutoff time of $|N(s)|$ instead of $\beta\ |N(s)| \ln |N(s)|$.

We note, however, that such an approach would be costly because each time the current solution changes, a random order of the neighbourhood of that solution would need to be computed. Such a computation needs $O(|N(s)|)$ time and would have to be carried out very often through the entire search. Data collected from our experiments shows that the number of times the state of LAHC changes (plateau moves aside) is much larger than the overhead factor of $\beta \ln |N(s)|$ (see Table 4.9 for TSP results), which implies that a systematic visit of the neighbourhood would not compensate. This is observed with all tested history list sizes and is aggravated for the larger ones. Note, for example, that for $L_h = 50000$, very often more than 50% of the iterations result in a state change. A similar behaviour is observed for

Table 4.9: For each TSP instance, and for $L_h \in \{1, 5000, 50000\}$, the table displays two values ($idle_r$ and $idle_p$) collected from the experiments that used the CCP cutoff time. $idle_r$ stands for the number of times that the state of LAHC changes, plateau moves aside, and is shown in thousands of iterations rounded to the nearest integer. $idle_p$ shows that same number percentage-wise in terms of the total number of iterations done during the entire run. The results are averaged over 100 independent runs. For each instance, we also show the value of $\beta \ln |N(s)|$ (using $p = 0.95$), the overhead factor of the CCP calculation with respect to visiting each solution in the neighbourhood exactly once.

Dataset	$\beta \ln \lvert N(s) \rvert$	$L_h = 1$		$L_h = 5000$		$L_h = 50000$	
		$idle_r$ (10^3)	$idle_p$	$idle_r$ (10^3)	$idle_p$	$idle_r$ (10^3)	$idle_p$
rat783	15.63	4	0.05%	16117	48.87%	161686	62.49%
u1060	16.23	6	0.03%	24198	44.30%	240926	61.68%
fl1400	16.79	9	0.02%	31318	34.41%	309054	61.78%
u1817	17.31	11	0.02%	44514	30.78%	446753	57.74%
d2103	17.60	13	0.02%	52834	32.27%	528565	58.54%
pcb3038	18.34	19	0.01%	79761	29.28%	800315	57.16%
fl3795	18.78	27	0.01%	103946	15.67%	1007428	47.27%

the QAP and PFSP experiments, whose results are available in the supplementary material.

The above argument gives concrete evidence that generating a random order of the neighbourhood each time the state of the algorithm changes, incurs a prohibitive computational cost. The only other reasonable alternative would be to undertake a systematic exploration of the neighbourhood using always the same fixed ordering as suggested by Connolly (1990). Such a strategy avoids the need to generate the neighbourhood anew each time the state of the algorithm changes, but it is questionable whether using a fixed ordering is always better than a random ordering. Connolly (1990) obtained good results for simulated annealing applied to the QAP, but also acknowledged that "the 'natural' ordering may cause the search to perform badly for other problems", and then continues saying "random search-without-replacement techniques could probably achieve the same effect, but at the expense of slightly more computational effort".

We should also note that, for certain algorithms such as LAHC, if a fixed ordering is used then we could question if we are still dealing with the SLS algorithm: the only randomized step would be in the genera-

tion of the initial solution (and even that not always —for example in PFSP we use NEH for the initial solution which is deterministic) and in the generation of the fixed ordering which would have to be generated randomly once and used for the entire run. Although technically such an algorithm can still be labelled as the SLS algorithm, very little about it is stochastic: given the initial solution and the fixed random ordering, everything else is deterministic.

We should also highlight that the way in which one-point randomized SLS algorithms accept worsening moves, that is presented in this chapter (and the entire of this research work), is the same as much of the literature, i.e., neighbours are sampled uniformly and independently at random from the entire neighbourhood, not in a systematic manner to avoid duplicate neighbours. The reason is most likely due to the imposed overhead needed for generating a random neighbourhood each time the algorithm changes its state. Nonetheless, a systematic exploration of the neighbourhood using a fixed ordering is also a viable alternative of performing local search, and in that case the CCP strategy obviously does not apply. Another viable alternative is to use a mixed strategy where most of the search uses a random exploration of the neighbourhood, and when a significant part of it has been covered, the algorithm could switch to a systematic exploration using a fixed ordering.

4.7 Summary

This chapter presented a cutoff time strategy based on the coupon collector's problem for one-point stochastic local search algorithms. The technique was illustrated with the LAHC algorithm on several instances of three classical combinatorial optimization problems: TSP, QAP, and PFSP.

The results show that the proposed strategy is a sound method to stop a one-point SLS algorithm that accepts worsening moves, and compares favorably with the 2% —or for that matter, any fixed percentage— of total time without improvement strategy. The CCP strategy decides to stop the algorithm at an adequate moment, neither too soon nor too late, and with a high probability that the current solution at the cutoff point is at a local optimum. Letting the algorithm run beyond that time could only improve

the solution cost if a sequence of plateau moves escapes from that local optimum.

Another advantage of the CCP strategy is that it requires no tuning and adapts itself depending on the neighbourhood size induced by the employed perturbation operator, and by the feedback of the search itself.

For this type of algorithm, the literature suggests three usual variants for the stopping condition: (1) the algorithm stops after reaching some target solution (or target value of the cost function), (2) the algorithm stops after a certain amount of time (or number of iterations), and (3) the algorithm stops when no further improvement seems possible.

The first variant is inapplicable when the target solution is unknown, and among the two other variants the third one is often the more appropriate one. The 2% strategy was an attempt to detect a reasonable cutoff point, the point beyond which no improvement seems possible. Although more appropriate than a static strategy consisting of a fixed amount of time (number of iterations) without improvement, the 2% strategy still makes incorrect decisions many times, as the results presented in this chapter illustrate.

The ideal cutoff time is problem (and instance) dependent, and the CCP strategy presented in this paper is a reliable method to detect it. Another important feature of the CCP strategy is the simplicity of its application. The same cannot be stated for a fixed percentage value based strategy which requires the specification of a percentage value and a minimum number of iterations before the percentage calculation takes place, both of which require previous tuning to work reasonably well, and even then it will make mistakes because no fixed percentage value is ideal for all cases.

Letting the search run when no further improvement seems possible is a waste of time, and could be spent more effectively by restarting the algorithm, possibly with a larger history length. This observation has been made before (Burke & Bykov 2017) and motivated the development of the parameter-less LAHC (Bazargani & Lobo 2017). In that work, the authors proposed an automated way of increasing the history list length at the cutoff point determined by the 2% strategy. The parameter-less LAHC could obviously benefit from using the CCP rather than the 2% strategy, something that we are currently addressing and will report in the near future.

The strategy presented here goes beyond the LAHC algorithm, and is applicable to other one-point randomized search algorithms. The method

is based on solid mathematical foundations and that is the major difference from other strategies found in the literature which often rely on an experimental rule-of-thumb. The mathematics behind the coupon collector's problem have been used for analysing randomized algorithms. Herein, we used it not for the analysis but for designing a rational way for deciding the cutoff time of SLS algorithms. We believe this is a good example of using theory to help design an algorithmic strategy in practice.

Nature is an endless combination
and repetition of very few laws.
She hums the old well-known air
through innumerable variations.

Ralph W. Emerson

Chapter 5

Revisiting pLAHC with more Suitable Cutoff Time

The pit fall of the parameter-less Late Acceptance Hill-Climbing (pLAHC) algorithm, introduced in Chapter 3, is that it uses an empirical restart strategy as opposed to a mathematically sound dynamic restart strategy. Now, since we succeeded in introducing a dynamic approach in the previous chapter that can scale based on employed perturbation operator and its induced neighbourhood size, as well as from feedback from the search, we are going to revisit pLAHC. This chapter aims to broader the scalability of pLAHC by incorporating the mathematically sound cutoff time to decide when a given LAHC execution should be restarted with an increased history length.

Note that, to all intents and purposes, in real-world problems the aim is to find a good enough quality solution in a pre-defined time. That being so, we would like the search algorithm to use all of its given time budget in an intelligent manner to yield as good a solution as it can. This means that the algorithm should not prematurely converge. Moreover, because the time budget can change for different reasons (even in the same system for the same instance running in different times), ideally the algorithm should provide us a good solution quality at any given time of the search, i.e., it should deliver good *anytime performance*. This is the overall motivation behind the pLAHC algorithm described in this chapter.

We present a comprehensive series of experiments which show that pLAHC, using the new cutoff strategy, provides good anytime performance on benchmark instances of three well-known combinatorial optimization problems:

the Travelling Salesman Problem, the Quadratic Assignment Problem, and the Permutation Flowshop Scheduling Problem. Our results confirm that the enhanced version of pLAHC works well not only on the TSP instances originally tested, but also on other real-world combinatorial optimization problems. We used Average Relative Percentage Deviation ($ARPD$) to report solution qualities which makes it possible to compare our results with those reported by state-of-the-art approaches. Although in TSP instances we do not reach to the best-known solutions due to the limitation of the TSP 2-exchange move operator, for some instances of QAP and PFSP, we obtain best known solutions reported in the literature. In addition to the specification of a perturbation operator suitable for a given problem, the only other decision that the user needs to make is to define the time budget, i.e., a certain period of time that the algorithm should run.

This chapter is organized as follows. Before introducing the modified pLAHC, we first, in the next section, discuss the tradeoff between search time and solution quality, the importance of anytime performance, and the benefits of parameter-less search for general practitioners in industry. Section 5.2 briefly reviews the parameter-less LAHC algorithm and a refinement of pLAHC and then describes the incorporation of the new restart strategy, based on the Coupon Collector's Problem, in pLAHC. Sections 5.3 and 5.4 present the experimental setup and the results obtained, respectively.

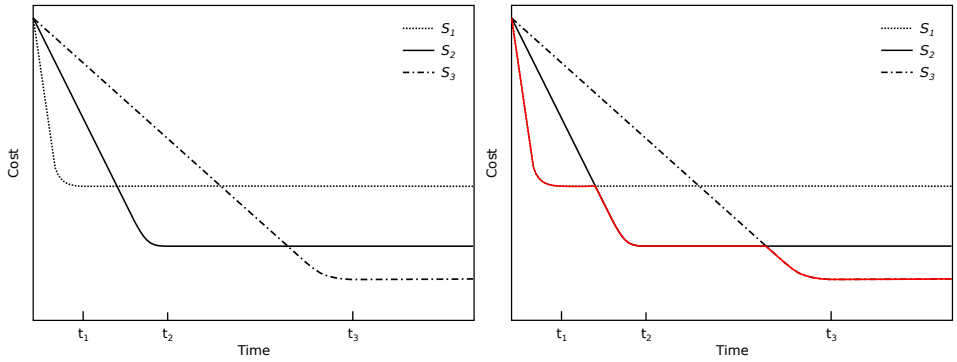
5.1 Anytime Performance and the Benefits of Parameter-less Search

Search algorithm design often represents an explicit trade-off between the time spent for the search and the quality of the obtained solution. This is a lucid proposition: the aim is to develop a search methodology where a longer search increases the chance of finding a better solution as it gets the opportunity to explore a larger part of the search space (Burke, Bykov, Newall & Petrovic 2003). With this in mind, we consider search algorithms that perform in a way that: 1) do not converge too quickly before using the entire given time budget; and, 2) do not need more time budget than the given one to utilize its stretch capacity. Configuring a search algorithm to effectively balance this trade-off is non-trivial, even for experienced practitioners.

Figure 5.1a depicts a typical scenario of the above mentioned trade-off,

where we are trying to minimize a cost function. In the figure, S_1, S_2, and S_3, represent three different parameter configurations of an optimization algorithm applied to a given problem instance. For a time budget t_2, the best parameter configuration would be S_2. Note that by using configuration S_1, the algorithm would quickly converge at t_1, and thereafter, from t_1 onwards, it would be stagnated. On the other hand, by using parameter configuration S_3, the algorithm would prematurely stop at t_2 before utilizing its entire capacity.

In the context of anytime performance, a search algorithm should exhibit good performance regardless of the total allocated time budget given to it, or even in cases where a pre-defined time budget is not specified. Under this scenario, its performance should ideally approximate to the one obtainable by the best configuration at any point in time, shown in red in Figure 5.1b.



(a) C_1 (b) C_1

Figure 5.1: Typical scenario of trade-off between cost and time in the search algorithms using different parameter configurations.

Trial-and-error experiments have been extensively reported in the literature for tuning parameters of search algorithms (some of these methods are introduced in Subsection 2.6.) These methods tune parameters of search algorithms for a given fixed time-budget, using certain sets of given instances of a problem. Therefore, upon changing the time budget, they need to be rerun to update those parameters' values in order to maintain the best performance of the employed search algorithm. This is quite unsustainable in some Operational Research (OR) complex uncertain environments such as airport operations (Weiszer, Burke & Chen 2020, Androutsopoulos, Manousakis & Madas 2020, Atkin, Burke, Greenwood & Reeson 2007).

Moreover, these methods use sets of given instances as an input for tuning parameters of algorithms with a presumption that those tunings would sustain beyond those instances. It is noteworthy that tuning of SLS algorithms applied to real-world complex problems, to a great extent, is instance dependent. Furthermore, for a large range of real-world optimization problems, the general practitioner just wants to press a button and get a result after an adequate amount of time. The period of time can also change for different reasons. Sometimes, the user wants a quick result, while in some other cases the user might want to give more time to the search algorithm to get to a better quality solution.

We need to mention that for a given time budget, all tuning approaches rely on the particular computational infrastructure, in which, the search algorithm is launched. The computational infrastructure could be affected by different factors, e.g., the computer hardware, the operating system, and the compiler. Each of these factors also depends on several other factors and they are constantly changing. For instance, the power of computer hardware to a great degree depends on processor speed, hard disk speed, and the amount of RAM. Since these computational resources are gradually improving, the time required to execute a single iteration is also being gradually reduced. Consequently, this strikes at previously tuned parameters. Besides, in the era of the Internet of Things, a lot of companies and organizations are using cloud computers as an outsourced computing power to shrink the size of their organization and furthermore reduce their expenses. This imposes even more complexity to calculate the real processing time of search algorithms, and incontrovertibly to tune their parameters. Tuning is useful for algorithms that have several parameters whose interaction is not well understood by a practitioner (or even by an expert). However, if a search algorithm can be designed in a way that is minimalist in terms of parameters, and moreover those parameters are well understood in terms of their effect in the algorithms' performance, then a rational strategy for automating the setting of those parameters values may be obtained.

5.2 Parameter-less LAHC Using CCP Cutoff Time

In most SLS algorithms, when the search is at a local optimum, there is little chance to escape from it. There are successful approaches that use

some sort of restart mechanism to overcome this barrier, e.g., Iterated Local Search (Stützle & Ruiz 2018), Adaptive Multi-Start (AMS) heuristic (Boese, Kahng & Muddu 1994), and Greedy Adaptive Search Procedure (GRASP) heuristic (Feo & Resende 1995), to name a few. These methods somewhat restart the search from somewhere else in the search space once a region has been extensively explored with the hope to find a better quality solution.

The pLAHC algorithm not only benefits from the restart mechanism, it also enlarges the area of the search space that it can explore in each restart by using a history length twice as large as in the previous restart. The restart decision can be easily made for those algorithms that systematically explore the entire neighbourhood of a solution. This is, however, non-trivial in pLAHC and other search algorithms that accept worsening moves on a current solution. Ideally, the restart should be performed at the point where the search stagnates, but that is not easy to detect.

In the original work of pLAHC, the dynamic cutoff time that was suggested by Burke & Bykov (2017) was used do decide when to restart LAHC with a larger history length. In that approach, the LAHC algorithm restarts when the number of consecutive non-improving (idle) iterations reaches 2% over the total number of iterations and, at least, 100,000 iterations are performed to avoid early termination. Although, the approach was effective on applying pLAHC to several TSP instances, the cutoff time strategy utilized was based on empirical grounds. In Chapter 4, we have done a very through investigation on this and introduced a new dynamic cutoff time strategy that is able to reliably detect the stagnation point (point of convergence) for SLS algorithms that accept worsening moves, applied to combinatorial optimization problems. This new approach is derived from the Coupon Collector's Problem (CCP), and is based on theoretical grounds as opposed to an empirical rule-of-thumb. It can be scaled based on the employed perturbation operator and its induced neighbourhood size, as well as from feedback from the search. In this chapter, we replace the 2% cutoff time, that was used in the original pLAHC algorithm, with this new cutoff time strategy. The pseudocode of the pLAHC algorithm using the CCP cutoff time is given in Algorithm 5.1, with lines in red colour indicating changes with respect to Algorithm 2.7 (i.e., LAHC).

The CCP cutoff time is calculated in Lines 1 and 2. Line 1 calculates β as it is given in Equation 4.3, and based on that, the number of iterations

Algorithm 5.1: Parameter-less Late Acceptance Hill-Climbing (pLAHC) using the CCP cutoff time.

Input : Confidence level for the CCP cutoff strategy: p
Output: A solution to a given problem.

```
1  β = 1 − ln(1 − p)/ln(|N(s)|)
2  θ = ⌈β ∗ |N(s)| ∗ ln(|N(s)|)⌉                          // CCP cutoff time
3  L_h = 1
4  best = ∅
5  while stopping criterion is not fulfilled do
6      Produce an initial solution s
7      Calculate its cost function value C(s)
8      forall k ∈ {0 ... L_h−1} do
9          f_k = C(s)                                     // Initial history list
10     θ = max (θ, L_h)
11     v = 0
12     I = 0                                              // Iteration counter
13     I_idle = 0                                         // Idle iteration counter
14     do until (I_idle < θ)
15         Construct a candidate solution s'
16         Calculate its cost function value C(s')
17         if C(s') ≥ C(s) then
18             I_idle = I_idle + 1
19         else
20             I_idle = 0                                 // Reset idle counter
21         v = I mod L_h                                  // Virtual beginning
22         if C(s') < f_v or C(s') ≤ C(s) then
23             s = s'                                     // Accept candidate
24         if C(s) < f_v then
25             f_v = C(s)                                 // Update history list
26         I = I + 1
27         if C(s) < C(best) then
28             best = s                                   // Update best so far
29     L_h = 2 × L_h
30 return best
```

needed to restart the algorithm, θ, is calculated in Line 2. Line 3 of the algorithm keeps the history list length of the current run of LAHC, and it is doubled in each restart as it is shown in Line 29. Line 4 keeps the best-ever-solution that the algorithms found so far throughout all restarts of LAHC, and it is updated in Lines 27 and 28. Line 10 makes sure that the algorithm checks all elements of a history list once before restarting it.

5.3 Experimental Setup

This section describes the goals of the experiments, the benchmark problems, and the experimental setup. The experimental results and its analysis are deferred to the following section.

The experiments were programmed in C++ and executed using High Performance Computing (HPC) provided by Queen Mary University of London. The overall goals of our experiments are:

- To study the effectiveness of pLAHC using the newly introduced dynamic-cutoff-time in Chapter 4 across different problem classes, across problem instances within each class, different perturbation operators, and even different overall stopping criteria. To carry out this, besides applying pLAHC to the same TSP instances used in Chapter 3, we also applied it to several instances of the Quadratic Assignment Problem (QAP) and the Permutation Flowshop Scheduling Problem (PFSP). (Details of these three problems are given in Subsections 2.1.1, 4.4.1.) We do not compare the obtained results with the original work of pLAHC, since, as we presented in Table 4.5, no fixed percentage value of consecutive "idle time" over total running time (that is what was used in the original work) is adequate for all problem classes and problem instances. Moreover, the restarting mechanism used in the original work requires at least 100,000 iterations before considering any restart of LAHC, which is a lot more than necessary for instances with small sizes (see Tables 4.3 and 4.4).

- To show that pLAHC can reach to any solution cost obtained by the regular LAHC with a fixed history length. This validates our early assertion that pLAHC does not require any tuning or preprocessing to reach to a given solution cost. To do so, we used the solution cost obtained by the regular LAHC with a fixed history length as the stopping criterion in pLAHC.

- To investigate the anytime performance behaviour of pLAHC. Once again, it has been shown in the literature that SLS can benefit from restarts. Building on this, we used a multistart LAHC with a fixed history length to obtain the total number of iterations in 50 independent restarts. We used this number as the overall stopping criterion

(maximum number of iterations) for pLAHC. Our experimental results confirm that pLAHC without requiring any tuning can obtain similar results to the ones obtained by the multistart LAHC with a fixed history length. In some cases it even reaches to a better cost.

- To investigate the efficacy of seeding within pLAHC beyond the TSP instances that were originally studied in Chapter 3. In their work, they concluded that parameter-less LAHC with seeding (pLAHC-s) speeds up the search on several instances of TSP. Note that pLAHC-s uses knowledge obtained in previous runs to initialize a new run of LAHC (see Section 3.3 and Algorithm 3.1). In this chapter, by applying pLAHC-s to several instances of QAP and PFSP, we aim to study, if pLAHC-s maintains its advantages for other combinatorial problems.

5.3.1 Algorithm Setup

For each problem, we employed a perturbation operator that is frequently used in the literature for that problem. Similar to Chapter 4, for TSP and QAP, we used the 2-exchange move, and for PFSP, an insertion move is employed.

The initialization of pLAHC is identical when applied to TSP and QAP instances: the initial candidate solution, in each restart, is generated uniformly at random. Then, all the L_h elements of the history list of pLAHC are initialized with the cost value of the initial solution. For PFSP, we use the well-known construction heuristic NEH (Nawaz, Enscore & Ham 1983) to generate the initial solution. Details of the NEH algorithm is given in Subsection 4.4.3. In our experiments, we used the NEH solution as the initial solution in each restart of pLAHC. However, since the initial solution constructed by NEH is already a very good solution (Taillard 1990), in each restart of pLAHC, we initialize all the L_h elements of the history list with a cost value of a randomly generated solution. Note that, as explained in Subsection 2.2.1, LAHC benefits from accepting worsening moves and initializing the history list with a value that is already good forestalls LAHC to leverage its design.

We used a confidence value $p = 0.95$ for the CCP cutoff time (to decide when a given LAHC execution, within either pLAHC or pLAHC-s, should be restarted). This indicates that LAHC restarts with a history list length

twice as large when, with 95% confidence, the entire neighbourhood of the current solution has been explored without resulting in a change in the state of the algorithm, plateau moves aside (as it is explained in Chapter 4).

5.4 Experimental Results

This section presents a set of computer experiments to address the goals listed at the beginning of Section 5.3.

5.4.1 Ability of pLAHC to Find a Solution with a Given Cost

We start by conducting experiments that show that pLAHC is able to reach a solution cost reached by LAHC with a given fixed history length, and that it does so without the user needing to specify any parameter, showing that pLAHC requires no tuning. To accomplish this, for every problem instance, LAHC is run for 100 independent times using three history lengths $L_h \in \{1, 5000, 50000\}$. The stopping criterion for each of these runs is dictated by the CCP cutoff time, as it is known that from that point on nearly no further improvement can be expected as described earlier in Subsection 4.2.1. In some sense, the solution cost C_x obtained by each of these independent runs is close to the best possible cost attainable when using a given fixed history length x. In other words, if the target is to reach a solution cost C_x, then a history length $L_h = x$ is a tuned parameter value to reach that target solution cost C_x.

The history lengths that we used are the same as those reported by Burke and Bykov in the experimental section of the original paper on LAHC (Burke & Bykov 2017). Then, for each combination of problem instance and history length we computed the average of the best-ever-solution cost reached by the end of each of the 100 independent runs. One should note that the current solution is not always the best-ever-solution, since the algorithm accepts worsening moves. We used the obtained average cost, which we denote as C_x (obtained by the regular LAHC with history length x) as the stopping criterion for pLAHC. In other words, pLAHC runs until it reaches a solution cost C_x and then stops. As an example, C_{5000} for the TSP instance `rat783` refers to the average of the best-ever-solution cost obtained on 100 independent runs of LAHC with history length 5000.

The results are summarized in Table 5.1, where C_x are shown as Average Relative Percentage Deviation ($ARPD$) from the optimal or best-known solutions for the TSP, QAP, and PFSP instances. The $ARPD$ is calculated as described in Equation 4.5. The Overhead Factor (OF) indicates how much slower (or faster) pLAHC is compared with LAHC. The OF is calculated by dividing the average number of iterations of pLAHC with the average number of iterations of LAHC. Table 5.1 presents the $ARPD$ of each instance using the three different history lengths. We provide these to give the reader an idea of how far the obtained results are from the optimal or best-known results.

The first thing that stands out from Table 5.1 is that $C_1 \geq C_{5000} \geq C_{50000}$ across all problem instances of the different problem classes. This confirms that the longer the history length is, the better solution cost can be expected to be at the end of the run.

We highlight that a given C_x value is close to the best possible cost attainable by LAHC when using history length x. In other words, a history length $L_h = x$ is a tuned parameter value for LAHC if the goal of the experiment is to reach a solution cost C_x: a smaller history length may be unable to reach solution cost C_x at all, while a larger history length should allow LAHC to reach a solution cost C_x but it will take more time to do so, when compared with the configuration that uses $L_h = x$.

Note how pLAHC is always able to reach the target solution cost C_x, and does so without needing any tuning. Indeed, pLAHC is capable of discovering an appropriate history length required to reach a certain solution quality obtained by a tuned LAHC. These results show that pLAHC, in a simple and effective way, can escape local optima by restarting LAHC with a larger history length when its current history length appears to be insufficient.

As expected, pLAHC, in most cases, is slower than a tuned LAHC (see the overhead factors in Table 5.1), but that is a reasonable price to pay for relieving a user to set the L_h parameter, and eliminate the risk of setting it with an inadequate value. Bear in mind that running a regular LAHC with a fixed (and potentially inappropriate) history length value, might hinder the possibility of the algorithm to reach the target solution cost no matter how much time is given to the algorithm. It is also notable that sometimes pLAHC is even faster than a tuned LAHC. The history list length that

Table 5.1: Overhead Factor (OF) of pLAHC on TSP, QAP, and PFSP instances to reach at least the same solution quality (C_x) as that obtained by LAHC with a fixed history length. C_x for $x \in \{1, 5000, 50000\}$ are shown as Average Relative Percentage Deviation ($ARPD$). The results are averaged over 100 independent runs.

TSP

Dataset	C_1		C_{5000}		C_{50000}	
	$ARPD$	OF	$ARPD$	OF	$ARPD$	OF
rat783	0.13	1.18	0.06	5.61	0.03	3.44
u1060	0.14	1.03	0.05	6.10	0.02	3.74
fl1400	0.08	0.76	0.03	7.80	0.01	3.69
u1817	0.17	0.86	0.09	7.95	0.04	4.30
d2103	0.20	1.12	0.11	7.68	0.07	4.27
pcb3038	0.15	1.02	0.09	8.07	0.05	4.26
fl3795	0.12	0.97	0.08	7.71	0.05	5.52

QAP

Dataset	C_1		C_{5000}		C_{50000}	
	$ARPD$	OF	$ARPD$	OF	$ARPD$	OF
tai30a	0.05	1.17	0.03	1.67	0.02	1.06
tai30b	0.13	0.82	0.03	0.11	0.03	0.03
tai35a	0.05	1.14	0.03	2.16	0.02	1.48
tai35b	0.08	1.38	0.02	0.32	0.02	0.04
tai40a	0.05	1.04	0.03	1.89	0.02	2.29
tai40b	0.10	1.03	0.02	0.77	0.02	0.24
tai50a	0.05	1.13	0.03	2.49	0.03	1.46
tai50b	0.07	0.81	0.01	0.53	0.00	0.12
tai60a	0.04	1.56	0.03	2.00	0.03	1.31
tai60b	0.06	1.30	0.01	0.76	0.00	0.31
tai64c	0.01	0.47	0.00	0.62	0.00	0.66
tai80a	0.04	1.09	0.03	1.71	0.02	1.80
tai80b	0.05	0.91	0.01	1.76	0.01	1.32
tai100a	0.04	1.14	0.03	2.51	0.02	2.09
tai100b	0.05	1.01	0.00	2.37	0.00	0.82
tai150b	0.03	0.68	0.01	2.65	0.00	1.41
tai256c	0.00	0.96	0.00	2.60	0.00	1.23

PFSP

Dataset	C_1		C_{5000}		C_{50000}	
	$ARPD$	OF	$ARPD$	OF	$ARPD$	OF
tai001 – 020×05	0.00	7.37	0.00	0.22	0.00	0.02
tai011 – 020×10	0.01	1.27	0.00	1.17	0.00	0.76
tai021 – 020×20	0.02	0.93	0.01	0.90	0.00	1.24
tai031 – 050×05	0.00	0.06	0.00	0.82	0.00	0.09
tai041 – 050×10	0.04	1.67	0.03	0.27	0.02	0.73
tai051 – 050×20	0.05	1.15	0.04	1.96	0.04	2.51
tai061 – 100×05	0.00	0.12	0.00	0.05	0.00	0.01
tai071 – 100×10	0.01	0.90	0.00	0.30	0.00	0.05
tai081 – 100×20	0.04	1.11	0.04	1.02	0.03	1.54
tai091 – 200×10	0.00	0.40	0.00	0.07	0.00	0.02
tai101 – 200×20	0.02	1.35	0.02	1.20	0.02	0.36
tai111 – 500×20	0.01	1.60	0.01	4.52	0.01	3.36

obtained the best-ever-solutions are not shown in Table 5.1 since it are discrete numbers and cannot be averaged over 100 independent runs.

The overhead factors obtained for the TSP instances are mostly larger than those obtained for QAP and PFSP. We hypothesize that this is due to the search space structure of these problems. It has been shown in the literature that, as opposed to what happens with TSP instances, local optima in QAP and PFSP instances are quite far from each other (Tayarani-N. & Prügel-Bennett 2015, Hernando, Daolio, Veerapen & Ochoa 2017). Thus, the implicit restarting mechanism incorporated in pLAHC provides it with the opportunity to find a given solution cost faster.

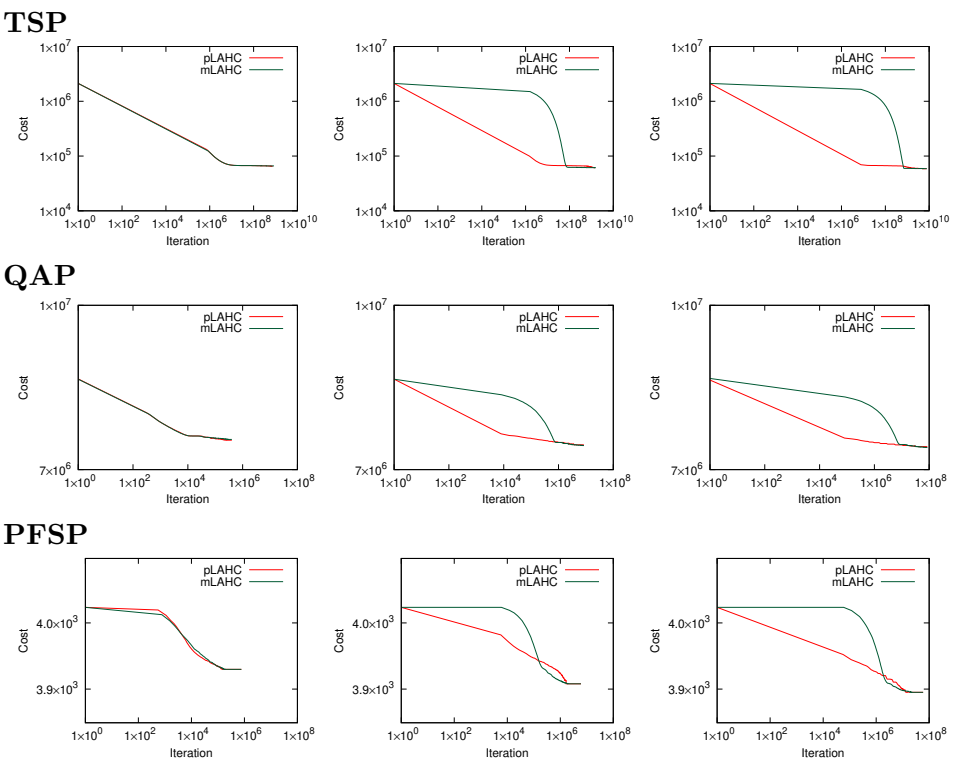
As a final observation, consider the results obtained for the QAP instance `tai100b` and the PFSP instance `tai011` – `020×10`, where the best-known or optimal solution for those instances was reached by pLAHC. It would be extremely unlikely that a practitioner (even an expert one) would "guess" the proper value of L_h needed to reach that best-known or optimal solution, while pLAHC does not have this problem.

5.4.2 Anytime Performance

This subsection presents a different set of experiments to illustrate the anytime performance of pLAHC. The pLAHC has a restarting mechanism in its mode of operation because it runs LAHC with exponentially increasing history list lengths. This gives it an advantage over a regular LAHC, even if both algorithms are given the same time budget. To make a more fair comparison, we should compare pLAHC with a multistart version of the regular LAHC. By doing so, both algorithms can benefit from restarts, the differences being that the LAHC restarts with the same history list length.

We employed a multistart LAHC (which we refer as mLAHC) that restarts LAHC for 50 times using the same fixed history length. Similarly to the previous experiments, we used $L_h \in \{1, 5000, 50000\}$, and a confidence $p = 0.95$ for the CCP cutoff time. For each problem instance and history length, mLAHC is run for 100 independent times, and the average of total number of iterations is computed. We refer to the resulting average number of iterations as I_x when mLAHC uses $L_h = x$, and use that as the stopping criterion for pLAHC. Thus, the two algorithms are given the same time budget. However, pLAHC does not execute LAHC as many times as mLAHC due to increasing the history list length in each restart.

Figure 5.2 depicts log-log plots of the average current solution cost through time, obtained by mLAHC and pLAHC for three instances, one for each problem set. A similar behaviour is observed for the other instances. By the end of a given maximum number of iterations (I_x), pLAHC obtained a similar cost as mLAHC in all instances of the three different problem classes. For both pLAHC and mLAHC, the cost value decreases through time. However, pLAHC delivers a better anytime performance, and it does so without having any parameter.



(a) I_1 (b) I_{5000} (c) I_{50000}

Figure 5.2: Solution cost through time obtained by pLAHC and mLAHC, for the TSP `u1817` instance, QAP `tai60a` instance, and PFSP `tai051–050×20`. Subfigures (a), (b), (c), correspond to different maximum number of iterations allowed for the algorithms to run (I_1, I_{5000}, I_{50000}), with I_x being the number of iterations taken by mLAHC with a fixed history length $L_h = x$ and 50 restarts. The data is collected from the average of 100 independent runs. Cost and iteration are shown in log scale.

For the stopping criterion of I_1 maximum number of iterations, mLAHC does 50 restarts using $L_h = 1$, while pLAHC does fewer restarts. Note

that, the last restart of LAHC within the pLAHC scheme presumably halts abruptly before utilizing its stretch capacity, when it hits the given maximum number of iterations. Bazargani & Lobo (2017) explained that there is no meaningful difference between solutions obtained by very small history lengths. However, this changes when the difference between history lengths is large. Therefore, it is not surprising that mLAHC and pLAHC have a similar behaviour when I_1 maximum iterations is used as stopping criterion.

For the stopping criterion of I_{5000} and I_{50000} maximum number of iterations, mLAHC is able to compete with pLAHC towards the end of the given time budget, but not at the beginning. Overall these results indicate that pLAHC is a better alternative in terms of anytime performance, with the added benefit of being easier to use since it has no parameters to set.

5.4.3 Does Seeding always Speeds up pLAHC?

Herein, we repeat the experiments done in Subsection 5.4.1, this time, by means of pLAHC-s. In Chapter 3, it was shown empirically that seeding could speed up the pLAHC algorithm when applied to TSP instances. In this chapter, although we use a different cutoff time, we still observe the same behaviour from pLAHC-s applied to TSP instances (see Table 5.2). However, this behaviour does not hold for the other two problem classes, i.e., QAP and PFSP.

Let us, again, briefly review the difference between pLAHC and pLAHC-s. Similarly to pLAHC, once LAHC with a fixed history length stagnates, it restarts LAHC by doubling its history length. However, instead of starting the search process from scratch, it uses the best solution obtained in the previous run as the initial solution of the new one, and it also initializes the history list with the cost values collected from previous successful iterations. In case the history list length is larger than the total number of previous successful iterations, then there will be duplicate cost values in the initial history list.

We argue that the seeding mechanism is advantageous on fitness landscapes where local optima are relatively close to each other. The advantage, compared with the regular pLAHC, comes from not needing to restart the search from scratch every time the history list length doubles its size. By keeping the best solution from the previous LAHC run and by initializing the history list with cost values from previously found successful iterations,

Table 5.2: Number of iterations needed by pLAHC and pLAHC-s to reach to at least the same solution quality as that obtained by LAHC with a fixed history length (C_x). Results are shown in a form of overhead factor (OF), i.e. how much slower pLAHC is compared with LAHC. The results are averaged over 100 independent runs. Entries in boldface are statistically significant with a *p-value* < 0.05 according to the Wilcoxon signed-rank test.

TSP

Dataset	pLAHC (OF)			pLAHC-s (OF)		
	C_1	C_{5000}	C_{50000}	C_1	C_{5000}	C_{50000}
rat783	**1.18**	5.61	3.44	2.69	**4.07**	3.06
u1060	1.03	6.10	3.74	1.88	**4.57**	**3.21**
fl1400	**0.76**	7.80	3.69	2.72	**6.23**	**2.75**
u1817	**0.86**	7.95	4.30	1.88	**5.26**	**3.42**
d2103	**1.12**	7.68	4.27	2.04	**5.47**	**3.29**
pcb3038	**1.02**	8.07	4.26	3.26	**5.87**	**3.13**
fl3795	**0.97**	7.71	5.52	2.03	**5.00**	**3.68**

QAP

Dataset	pLAHC (OF)			pLAHC-s (OF)		
	C_1	C_{5000}	C_{50000}	C_1	C_{5000}	C_{50000}
tai30a	**1.17**	1.67	1.06	2.26	0.96	**0.32**
tai30b	**0.82**	**0.11**	**0.03**	3.16	0.26	0.03
tai35a	**1.14**	2.16	1.48	2.26	**1.16**	**0.37**
tai35b	**1.38**	**0.32**	**0.04**	3.26	0.58	0.05
tai40a	**1.04**	1.89	2.29	2.13	**1.04**	**0.60**
tai40b	**1.03**	0.77	0.24	3.12	0.61	0.11
tai50a	**1.13**	2.49	1.46	1.95	2.04	0.76
tai50b	**0.81**	**0.53**	0.12	2.42	0.69	0.16
tai60a	**1.56**	2.00	1.31	2.43	2.05	0.75
tai60b	**1.30**	**0.76**	0.31	3.87	1.29	0.36
tai64c	0.47	**0.62**	0.66	1.18	0.87	**0.24**
tai80a	**1.09**	1.71	1.80	1.98	1.80	1.04
tai80b	**0.91**	1.76	1.32	2.85	1.23	0.78
tai100a	**1.14**	2.51	2.09	1.77	2.45	1.20
tai100b	**1.01**	2.37	0.82	2.75	**1.84**	0.99
tai150b	**0.68**	2.65	1.41	2.83	3.13	1.96
tai256c	**0.96**	2.60	1.23	1.85	2.90	1.29

PFSP

Dataset	pLAHC (OF)			pLAHC-s (OF)		
	C_1	C_{5000}	C_{50000}	C_1	C_{5000}	C_{50000}
tai001 – 020×05	7.37	0.22	0.02	**1.92**	0.07	**0.01**
tai011 – 020×10	**1.27**	1.17	0.76	2.40	**0.36**	**0.27**
tai021 – 020×20	**0.93**	**0.90**	1.24	2.82	3.06	**0.34**
tai031 – 050×05	0.06	**0.82**	**0.09**	0.13	2.55	0.32
tai041 – 050×10	**1.67**	0.27	0.73	2.60	0.33	**0.11**
tai051 – 050×20	**1.15**	1.96	2.51	1.82	**0.91**	**0.46**
tai061 – 100×05	0.12	0.05	0.01	0.10	0.04	0.01
tai071 – 100×10	0.90	0.30	0.05	1.78	0.45	0.08
tai081 – 100×20	1.11	1.02	1.54	1.22	0.88	**0.70**
tai091 – 200×10	0.40	0.07	0.02	0.56	0.08	0.02
tai101 – 200×20	1.35	1.20	0.36	1.37	0.99	0.30
tai111 – 500×20	1.60	4.52	3.36	1.91	3.81	**2.01**

the seeding mechanism can more easily jump to a different local optimum. This seems to be the case with many TSP instances, where local optima are concentrated in a small region of the landscape (see Hoos & Stützle 2005, p. 220).

However, if local optima are located far from each other on the fitness landscape, then, the seeding mechanism is too exploitative and does not speed up pLAHC to reach to a given cost value. This seems to be the case in most Taillard's instances of QAP used in this work. They are real-life like instances that have their local optima, under the 2-exchange neighbourhood operator, distributed across the entire fitness landscape (Freisleben & Merz 2000, Hoos & Stützle 2005, p. 481). In the case of PFSP, the fitness landscape induced by the insertion neighbourhood is smooth (Hernando, Daolio, Veerapen & Ochoa 2017). It has a high degree of neutrality (Marmion, Dhaenens, Jourdan, Liefooghe & Verel 2011) with local optima being spread across the landscape. Both algorithms, i.e., pLAHC and pLAHC-s, employed the construction heuristic, NEH, to generate the initial solution. This means that they start the search process from the exact same basin in the fitness landscape. To work well under this scenario, pLAHC-s requires a longer history list length with more copies of poor cost values to be able to move out from one local optimum to another one with the hope of getting to a better quality solution.

5.5 Summary

This chapter presents a modification of pLAHC, that was introduced in Chapter 3, to make it good anytime performance. It aims at delivering a quality solution comparable to what can be obtained by the best configuration of the LAHC algorithm at any point in time. The pLAHC algorithm, used in this chapter, was originally introduced to eliminate the only parameter of the LAHC algorithm. It was successfully applied in the past on several TSP instances. However, due to the employed restart mechanism, it could not be as effective when applied to other problem classes.

In this chapter, we employed a newly introduced dynamic cutoff time strategy, i.e., CCP cutoff time, to decide when to restart each LAHC execution within pLAHC. As a result, the new pLAHC becomes scalable based on the employed perturbation operator, the size of neighbourhood, as well as from feedback received from the search. It is a ready-for-use heuristic ap-

proach that any general practitioner can easily apply to solve combinatorial optimisation problems without needing to have a deep knowledge on their internal mechanisms. More importantly, the resulting algorithm delivers good anytime performance. We carried out a set of experiments on benchmark instances of TSP, QAP, and PFSP. We employed Average Relative Percentage Deviation to report the obtained solution qualities which makes it possible to compare our results to those reported by the state-of-the-art. In some instances of QAP and PFSP, pLAHC obtained the best-known solutions reported in the literature (note that entries of Table 5.2 that report zero ARPD, pLAHC obtained the best-known solution.) We also investigated if the seeding mechanism can always speed up pLAHC. It definitely can on fitness landscapes where local optima tend to be clustered in a particular region of the search space. However, if local optima are spread over the search space, then seeding turns out to be detrimental compared to the non-seeding version of pLAHC.

The important thing is not to stop questioning. Curiosity has its own reason for existing.

Albert Einstein

Chapter 6

Conclusions and Future Work

This final chapter summarizes the work contained in this dissertation, presents the major contributions, and highlights a number of topics that deserve further exploration.

6.1 Summary of Contributions

This dissertation started by giving an example of application of SLS algorithms to a real-world problem to demonstrate the importance and impact of this class of algorithms on day-to-day life. We explained that these algorithms are suitable for addressing complex and large scale combinatorial problems. Combinatorial problems involve finding a solution from a finite set of discrete objects that satisfies certain conditions. The travelling salesman problem is a prototypical example of a combinatorial optimization problem. In this type of problem, going through all feasible solutions to find the best one, assuming it is possible to generate them all, is not usually a viable option, since there are too many of them to be processed. SLS are successful and popular search algorithms for this type of problem. They go through the search space in an intelligent way, by avoiding unpromising points, to provide a promising locally optimum solution. We provided some fundamental concepts and definitions of SLS algorithms, particularly, one-point SLS algorithms that represent the main focus of this thesis. Iterative Improvement, Simulated Annealing, Tabu Search, Threshold Accepting, the Great

Deluge Algorithm, and Iterated Local Search are among the widely used one-point SLS algorithms. They were introduced, discussed, and some of their key application areas were cited to demonstrate the type of real-world problems where they have been successfully applied.

We then meticulously presented the LAHC algorithm, a successful one-point SLS (Franzin & Stützle 2018), that was recently introduced by Burke & Bykov (2008, 2017). This method has only one parameter, the history list length, whose meaning is quite straightforward: the longer the history list length is, the better quality solution is expected to be obtained at the end of the search process; of course, at the expense of time. The LAHC is simpler than other SLS of its kind, such as SA,TS, TA, GDA, and ILS. It also benefits from adapting some features of other SLS algorithms.

Similar to TS, LAHC uses a list to memorise some knowledge obtained during previous iterations for future utilisation. However, instead of keeping a solution itself (or a form of it) in the list as is done by TS, it only memorises the solution quality value. This makes the LAHC a memory affordable approach with a history list containing values that are simple to interpret. Furthermore, in TS, the tabu tenure that indicates tabu list length does not have the same interpretation as the history list length in LAHC. The longer the tabu list length is, the more restrictive the TS becomes (Gendreau & Potvin 2013), and that is why different mechanisms, such as aspiration criteria, are introduced to revoke tabus. However, this is not the case for the history list length of LAHC. Similar to SA, TA, and GDA, LAHC accepts worsening moves, but it neither requires a deterministic cooling schedule (i.e., TA and GDA), nor probabilistic cooling schedule (i.e., SA). Instead, it uses an adaptive approach that makes such decisions based on the cost of the current solution or the cost of the solution from some iterations before. This approach is adaptive to the search space, since the history list get updated as the search proceeds. In the absence of the cooling schedule, the tuning of the only parameter of LAHC is easier than SA, TS, TA, and GDA.

One of the major contributions of this thesis was to further simplify LAHC with removing the necessity to tune its only parameter. This makes LAHC a suitable and easy-to-use search methodology for general practitioners who do not have a deep insight into the parameter tuning of SLS algorithms. It has been shown in the literature that SLS can benefit from restarting the search from somewhere else in the search space (Hoos & Stützle

2005, Martí, Aceves, León, Moreno-Vega & Duarte 2018). Another contribution of this thesis was to incorporate a successful restart mechanism in the LAHC algorithm. It is quite straightforward for algorithms such as ILS to make a restart decision; however, it is non-trivial for those SLS algorithms that accept worsening moves. In ILS, whenever the Iterative Improvement algorithm used within ILS is in the local optimum, then it restarts the algorithm. This is not the case in LAHC, since it accepts worsening moves and therefore there is no way to know that we are in the local optimum. Another major contribution of this thesis was to propose a new dynamic cutoff time strategy that is able to reliably detect the stagnation point for this type of one-point SLS algorithms. In examining these contributions we studied their application to a real-world problem. We also showed when these contributions come together, they can make an anytime performance local search algorithm. This will be discussed in more detail in the following paragraphs.

One of the known pitfalls of applications of SLS is that users of these algorithms usually have to do a lot of parameter tuning in order to succeed with them (Smit & Eiben 2009), where most of tuning is based on some experimental rule-of-thumbs (Smit & Eiben 2010). The first step of our journey in this thesis was to introduce a search algorithm that is easy-to-use, by general practitioners. We proposed an automated strategy which does not require parameter tuning. The goal was to eliminate the sole parameter of the LAHC algorithm. To do so, we used a technique that was originally introduced for automating population sizing in evolutionary algorithms (Harik & Lobo 1999), and adapted it, for eliminating the need to manually specify the value of the history list length of LAHC. We called this new approach the parameter-less LAHC algorithm (pLAHC). The strategy follows closely the restart mechanism that was used in Auger & Hansen (2005). The pLAHC starts the search with executing a LAHC with a small history list length and thereafter, whenever the LAHC is in the local optimum, it restarts it with a new history list length which is double the previous one. It should be noted that in each restart of pLAHC, with doubling the history list length, the algorithm expands the search area of the search space with the hope of finding a better quality solution. The pLAHC contains a restart mechanism and does not have any parameter to tune. It is slower than LAHC with a fine-tuned history list length; however, it does not require any tuning effort.

The validity of the method was shown with computational experiments on a number of instances of the Travelling Salesman Problem (Bazargani & Lobo 2017).

We also study the application of pLAHC on a real-world search based software engineering problem, known as the Combinatorial Interaction Testing (CIT) problem. CIT is known as a black-box sampling technique for discovering faults in highly configurable systems. In order to have a fare comparison with Simulated Annealing, we employed the constrained CIT problem using the well-established CASA framework introduced by Garvin, Cohen & Dwyer (2011). CASA is a three-nested-layer search framework that uses SA in its most inner layer. We successfully adapted pLAHC to the CASA framework and replaced SA by LAHC. We called the new framework Covering Array with Late Acceptance (CALA). We also used the CASA's design for triggering a restart of LAHC with doubling its history list length. Although CASA was originally designed and tuned to use SA, pLAHC showed good performance compared to SA in the CASA framework. CALA outperformed CASA in 14 of the 35 benchmark problem instances, and CASA outperformed CALA only in a single instance (Bazargani, Drake & Burke 2018).

Up to this point, we had successfully designed an approach that utilised a restart mechanism and did not have any parameter to tune. We had also tried two different types of cutoff time for restarting LAHC, namely dynamic and static cutoff times. Bazargani & Lobo (2017) used a dynamic cutoff time (i.e., the 2% strategy) that was originally introduced by Burke & Bykov (2008), and Bazargani, Drake & Burke (2018) employed a static cutoff time which was the maximum number of iterations before triggering a restart. One should note that, quoting from Burke & Bykov (2012):

> "[...] the most effective approach is to terminate the search at the moment of convergence. If we terminate it before the convergence then we do not employ the full power of the method. If we let the search run after the convergence, then we just waste computing time (this time could be spent more effectively with a larger [history list length])."

In the above quote, the *point of convergence* (a.k.a. the moment of convergence) refers to the point beyond which no improvement seems possible.

The exact same analogy applies to the restarting point of the search process in pLAHC. In our experiments, we had observed that although the 2% strategy is more appropriate than a static strategy consisting of a fixed amount of time (number of iterations) with or without improvement, it still makes incorrect decisions many times, as the results presented in Table 4.5. Burke & Bykov (2012) elaborate on the issue of detecting the point of convergence and the then state of the literature on this subject, which we quote again:

> "[. . .] the maximum effectiveness can be achieved when the search is stopped exactly at the point of the intersection of the time-cost curve and the envelope of all these curves. However, the form of the envelope is problem dependent and we currently have no reliable method to detect these points during the search."

While implementing pLAHC in practice, we also realized that the cutoff time strategies often used in the literature for one-point SLS algorithms for combinatorial problems, are mostly based on some empirical studies as opposed to theoretical grounds. Thus, the next obvious step in our journey, to introduce an anytime performance local search, was to design a reliable dynamic cutoff time strategy for one-point SLS algorithms that accepts worsening moves.

Recognizing this gap between the theory and practice of cutoff time strategies led us toward a practice-driven theory approach for this problem. So that, we introduced a new dynamic cutoff time strategy that is mathematically sound and derived from the Coupon Collector's Problem (CCP). CCP is a well-known problem from probability theory (Motwani & Raghavan 1995). It has been used to analyse the behaviour of SLS algorithms (Doerr 2011, Jansen 2013). In this thesis, we used it to design a scalable approach for choosing an appropriate cutoff time for one-point SLS algorithms that can accept worsening moves. The proposed approach does not impose a computational expense on the search algorithm, and does not have any parameter. It is scalable based on the employed perturbation operator and its induced neighbourhood size, as well as from feedback from the search. The suitability and scalability of the method was illustrated with the Late Acceptance Hill-Climbing algorithm on a comprehensive set of benchmark instances of three well-known combinatorial optimization problems: the Travelling Salesman Problem, the Quadratic Assignment Problem, and

the Permutation Flowshop Scheduling Problem. Although we illustrated the CCP cutoff time in LAHC, it is rather a generic cutoff time, and its functionality is not limited to LAHC. We also improved the best-known solutions of the 71 instances (out of 480) of Vallada, Ruiz & Framinan's instances for Permutation Flowshop Scheduling Problem (Lobo, Bazargani & Burke 2020).

Finally, we had all the components to fulfil the overachieving goal statement of this thesis, which was, to design search techniques for combinatorial problems that have fewer parameters while delivering good *anytime performance*. We incorporated the proposed cutoff time strategy in the pLAHC algorithm, as a restart mechanism, to decide when a given LAHC execution should be restarted with an increased history list length. The resulting algorithm is an anytime performance local search algorithm. It has no parameters and delivers good anytime performance, making it a suitable algorithm for the general practitioner. We presented a comprehensive series of experiments which show that pLAHC provides good anytime performance, competitive to state-of-the-art, on benchmark instances of three well-known combinatorial optimization problems: the Travelling Salesman Problem, the Quadratic Assignment Problem, and the Permutation Flow- shop Scheduling Problem.

6.2 Future Research

Following the work presented in this thesis, a number of topics deserve further investigation; some of them are outlined in the following.

Expand pLAHC into combinatorial problems with constraints.
We have shown that there is strong evidence that pLAHC is working well in benchmark problems. We also showed how successfully pLAHC can be applied to a real-world problem taken from the search based software engineering field. However, the application spectrum is wide, and can be applied to a wide range of real-world problems. As we mentioned in Chapter 2, real-world combinatorial problems often have some constraints. Their constraints are either blended in the objective function or taken into account separately. Although the LAHC can handle the former, it is important to investigate how to incorporate the latter in LAHC and pLAHC. This is another step to

widen the applicability of pLAHC (introduced in Chapter 3) for general practitioners who are dealing with this kind of problem. This can be done with using the data structure of the history list of LAHC to memorise more information about each solution (in this case number of constraints that have been violated) than just its cost value. This approach does not introduce any new parameters and it is easy to implement.

Integeration of CCP cutoff time in other one-point SLS. We illustrated the suitability and scalability of the CCP cutoff time strategy with LAHC. However, in Chapter 4, we explained that it "goes beyond the LAHC algorithm, and is applicable to other one-point randomized search algorithms." Similar to LAHC, having a reliable and mathematically sound cutoff time for other one-point SLS is necessary to avoid premature stopping decision of the search algorithm or even running it after the point of convergence. Thus, the integration of CCP cutoff time into other one-point SLS approaches is an extension to Chapter 4. One should note that the state of the algorithm that is used in the CCP cutoff time strategy differs from one algorithm to another. For example in LAHC, the state of the algorithm is given by the current solution and the history list. So that, the adaptation of the CCP cutoff time strategy to other one-point SLS is another important future research avenue.

Extend CCP cutoff time to hyper-heuristics. In our experiments in Chapters 4 and 5, we used one move operator for each problem class. However, in complex problems, the functionality of one operator is often limited; thereby, several move operators are normally employed to better explore the search space. This represents selection hyper-heuristics which were briefly discussed in Section 2.3. As presented in the different subsections of Chapter 2, there are many applications of one-point SLS in the literature that use hyper-heuristics. Therefore, it is a promising avenue of research to extend the CCP cutoff time introduced in Chapter 4 to hyper-heuristics. Note that, in selection hyper-heuristics, each move operator induces its neighbourhood size that should be used in CCP cutoff time. Moreover, neighbours induced by different move operators have overlap. We need to take into account

all the above facts when investigating this research topic.

Parallelization. The pLAHC algorithm introduced in this thesis is run in a sequential fashion. We explained in Chapter 5 that it is slower than a tuned LAHC. With today's advanced technology in computational power and cloud computing, it is important to investigate how to parallelize pLAH to speed up execution time. As we explained in Chapter 5, pLAHC is a memory-affordable algorithm that adapts itself based on the search space and the feedback from the search. The parallelization of pLAHC makes it a time-affordable algorithm for a general practitioner who wants to speed up the search process without impacting the search outcome. It can be done by running several copies of LAHC with different history list length in parallel. There are guidelines for carrying out efficient parallelization of evolutionary algorithms (Sudholt 2015). This is an extension to Chapter 5 that requires a thorough investigation.

Bibliography

Aarts, E. H. L., Korst, J. H. M. & van Laarhoven, P. J. M. (2003), Simulated annealing, *in* E. H. L. Aarts & J. K. Lenstra, eds, 'Local Search in Combinatorial Optimization', 2 edn, Princeton University Press, chapter 4, pp. 91–120.

Abdulaziz, H., Elnahas, A., Daffalla, A., Noureldien, Y., Kheiri, A. & Ozcan, E. (2018), Late acceptance selection hyper-heuristic for wind farm layout optimisation problem, *in* '2018 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)', IEEE.

Abdullah, S., Aickelin, U., Burke, E., Din, A. M. & Qu, R. (2007), Investigating a hybrid metaheuristic for job shop rescheduling, *in* '3rd Australian Conference on Progress in Artificial Life (ACAL 2007)', Vol. 4828 of *Lecture Notes in Computer Science*, Springer, pp. 357–368.

Abdullah, S., Turabieh, H., McCollum, B. & McMullan, P. (2010), 'A hybrid metaheuristic approach to the university course timetabling problem', *Journal of Heuristics* **18**(1), 1–23.

Acan, A. & Ünveren, A. (2020), 'Multiobjective great deluge algorithm with two-stage archive support', *Engineering Applications of Artificial Intelligence* **87**, 103239.

Ahmed, L., Mumford, C. & Kheiri, A. (2019), 'Solving urban transit route design problem using selection hyper-heuristics', *European Journal of Operational Research* **274**(2), 545–559.

Alinaghian, M., Tirkolaee, E. B., Dezaki, Z. K., Hejazi, S. R. & Ding, W. (2020), 'An augmented tabu search algorithm for the green inventory-routing problem with time windows', *Swarm and Evolutionary Computation* p. 100802.

Althöfer, I. & Koschnick, K.-U. (1991), 'On the convergence of "threshold accepting"', *Applied Mathematics & Optimization* **24**(1), 183–195.

Alwisy, A., Bouferguene, A. & Al-Hussein, M. (2018), 'Framework for target cost modelling in construction projects', *International Journal of Construction Management* **20**(2), 89–104.

Amaral, P. & Pais, T. C. (2016), 'Compromise ratio with weighting functions in a tabu search multi-criteria approach to examination timetabling', *Computers & Operations Research* **72**, 160–174.

Anand, S., Saravanasankar, S. & Subbaraj, P. (2011), 'Customized simulated annealing based decision algorithms for combinatorial optimization in VLSI floorplanning problem', *Computational Optimization and Applications* **52**(3), 667–689.

Andrade, M. D. D., Nascimento, M. A. C., Mundim, K. C., Sobrinho, A. M. C. & Malbouisson, L. A. C. (2008), 'Atomic basis sets optimization using the generalized simulated annealing approach: New basis sets for the first row elements', *International Journal of Quantum Chemistry* **108**(13), 2486–2498.

Androutsopoulos, K. N., Manousakis, E. G. & Madas, M. A. (2020), 'Modeling and solving a bi-objective airport slot scheduling problem', *European Journal of Operational Research* **284**(1), 135–151.

Atkin, J. A. D., Burke, E. K., Greenwood, J. S. & Reeson, D. (2007), 'Hybrid metaheuristics to aid runway scheduling at london heathrow airport', *Transportation Science* **41**(1), 90–106.

Auger, A. & Hansen, N. (2005), A restart CMA evolution strategy with increasing population size, *in* 'IEEE Congress on Evolutionary Computation (CEC 2005)', IEEE.

Bartz-Beielstein, T., Lasarczyk, C. & Preuss, M. (2005), Sequential parameter optimization, *in* 'IEEE Congress on Evolutionary Computation (CEC 2005)', IEEE.

Baykasoglu, A., Durmusoglu, Z. D. & Kaplanoglu, V. (2011), 'Training fuzzy cognitive maps via extended great deluge algorithm with applications', *Computers in Industry* **62**(2), 187–195.

Bazargani, M., Drake, J. H. & Burke, E. K. (2018), Late acceptance hill climbing for constrained covering arrays, *in* 'Applications of Evolutionary Computation (EvoApplications 2018)', Springer, pp. 778–793.

Bazargani, M. & Lobo, F. G. (2017), Parameter-less late acceptance hill-climbing, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'17)', ACM.

Beyer, H.-G. & Schwefel, H.-P. (2002), 'Evolution strategies — a comprehensive introduction', *Natural Computing* **1**(1), 3–52.

Bick, U. & Giger, M. L. (1997), 'Method and system for detection of lesions in medical images'.
URL: *https://patents.google.com/patent/US6185320B1/en*

Bilgin, B., Demeester, P., Misir, M., Vancroonenburg, W. & Berghe, G. V. (2011), 'One hyper-heuristic approach to two timetabling problems in health care', *Journal of Heuristics* **18**(3), 401–434.

Boese, K. D., Kahng, A. B. & Muddu, S. (1994), 'A new adaptive multi-start technique for combinatorial global optimizations', *Operations Research Letters* **16**(2), 101–113.

Bolaji, A. L., Bamigbola, A. F. & Shola, P. B. (2018), 'Late acceptance hill climbing algorithm for solving patient admission scheduling problem', *Knowledge-Based Systems* **145**, 197–206.

Bolaji, A. L., Michael, I. & Shola, P. B. (2018), Adaptation of late acceptance hill climbing algorithm for optimizing the office-space allocation problem, *in* 'Hybrid Metaheuristics', Springer, pp. 180–190.

Bosman, P. A., Luong, N. H. & Thierens, D. (2016), Expanding from discrete cartesian to permutation gene-pool optimal mixing evolutionary algorithms, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'16)', ACM Press.

Breedam, A. V. (1995), 'Improvement heuristics for the vehicle routing problem based on simulated annealing', *European Journal of Operational Research* **86**(3), 480–490.

Bryce, R. C. & Colbourn, C. J. (2007), One-test-at-a-time heuristic search for interaction test suites, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)', ACM Press.

Burkard, R. E., Karisch, S. E. & Rendl, F. (1997), 'QAPLIB — a quadratic assignment problem library', *Journal of Global Optimization* **10**(4), 391–403.

Burke, E., Causmaecker, P. D. & Berghe, G. V. (1999), A hybrid tabu search algorithm for the nurse rostering problem, *in* 'Second Asia-Pacific Conference on Simulated Evolution and Learning (SEAL'98)', Lecture Notes in Computer Science, Springer, pp. 187–194.

Burke, E. K. & Bykov, Y. (2006), Solving exam timetabling problems with the flex-deluge algorithm, *in* 'Proceedings of the 5th International Conference on the Practice and Theory of Automating Timetabling (PATAT 2006)'.

Burke, E. K. & Bykov, Y. (2008), A late acceptance strategy in hill-climbing for exam timetabling problems, *in* 'Proceedings of the 7th International Conference on the Practice and Theory of Automating Timetabling (PATAT 2008)'.

Burke, E. K. & Bykov, Y. (2012), The late acceptance hill-climbing heuristic, Technical Report CSM-192, Department of Computing Science and Mathematics, University of Stirling.

Burke, E. K. & Bykov, Y. (2016), 'An adaptive flex-deluge approach to university exam timetabling', *INFORMS Journal on Computing* **28**(4), 781–794.

Burke, E. K. & Bykov, Y. (2017), 'The late acceptance hill-climbing heuristic', *European Journal of Operational Research* **258**(1), 70–78.

Burke, E. K., Bykov, Y., Newall, J. & Petrovic, S. (2003), 'A time-predefined approach to course timetabling', *Yugoslav Journal of Operations Research* **13**(2), 139–151.

Burke, E. K., Bykov, Y., Newall, J. & Petrovic, S. (2004), 'A time-predefined local search approach to exam timetabling problems', *IIE Transactions* **36**(6), 509–528.

Burke, E. K., Causmaecker, P. D. & Berghe, G. V. (2004), Novel meta-heuristic approaches to nurse rostering problems in Belgian hospitals, *in* J. Y.-T. Leung, ed., 'Handbook of Scheduling: Algorithms, Models, and Performance Analysis', CRC Press, chapter 44.

Burke, E. K., Curtois, T., Hyde, M., Kendall, G., Ochoa, G., Petrovic, S., Vazquez-Rodriguez, J. A. & Gendreau, M. (2010), Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms, *in* 'IEEE Congress on Evolutionary Computation (CEC 2010)', IEEE.

Burke, E. K., Eckersley, A., McCollum, B., Petrovic, S. & Qu, R. (2003), Using simulated annealing to study behaviour of various exam timetabling data sets, *in* '5th Meta-heuristics International Conference (MIC03)', Kyoto, Japan.

Burke, E. K. & Kendall, G., eds (2014), *Search Methodologies*, Springer.

Burke, E. K., Kendall, G. & Soubeiga, E. (2003), 'A tabu-search hyper-heuristic for timetabling and rostering', *Journal of Heuristics* **9**(6), 451–470.

Burke, E. K., Kendall, G. & Whitwell, G. (2009), 'A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock-cutting problem', *INFORMS Journal on Computing* **21**(3), 505–516.

Bykov, Y. (2003), Time-predefined and trajectory based search: Singleand multiobjective approaches to exam timetabling, PhD thesis, The University of Nottingham, Nottingham, UK.

Carter, M. W., Laporte, G. & Lee, S. Y. (1996), 'Examination timetabling: Algorithmic strategies and applications', *Journal of the Operational Research Society* **47**(3), 373–383.

Cavallo, M., Modica, G. D., Polito, C. & Tomarchio, O. (2017*a*), A LAHC-based job scheduling strategy to improve big data processing in geo-distributed contexts, *in* '2nd International Conference on Internet of Things, Big Data and Security', SCITEPRESS - Science and Technology Publications.

Cavallo, M., Modica, G. D., Polito, C. & Tomarchio, O. (2017*b*), Multi-job hadoop scheduling to process geo-distributed big data, *in* 'IEEE Symposium on Computers and Communications (ISCC)', IEEE, pp. 1175–1181.

Chaimatanan, S., Delahaye, D. & Mongeau, M. (2014), 'A hybrid metaheuristic optimization algorithm for strategic planning of 4D aircraft trajectories at the continental scale', *IEEE Computational Intelligence Magazine* **9**(4), 46–61.

Chan, K., Kwong, C. & Luo, X. (2009), 'Improved orthogonal array based simulated annealing for design optimization', *Expert Systems with Applications* **36**(4), 7379–7389.

Chen, M., Koc, E., Shi, Z. & Soibelman, L. (2018), 'Proactive 2D model-based scan planning for existing buildings', *Automation in Construction* **93**, 165–177.

Chen, P.-H. & Shahandashti, S. M. (2009), 'Hybrid of genetic algorithm and simulated annealing for multiple project scheduling with multiple resource constraints', *Automation in Construction* **18**(4), 434–443.

Chen, S.-H. & Yeh, C.-H. (2001), 'Evolving traders and the business school with genetic programming: A new architecture of the agent-based artificial stock market', *Journal of Economic Dynamics and Control* **25**(3-4), 363–393.

Cheng, J. & Fournier, R. (2004), 'Structural optimization of atomic clusters by tabu search in descriptor space', *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)* **112**(1), 7–15.

Chiang, W.-C. & Russell, R. A. (1996), 'Simulated annealing metaheuristics for the vehicle routing problem with time windows', *Annals of Operations Research* **63**(1), 3–27.

Chira, M. & Plionis, M. (2019), 'A simulated annealing algorithm to quantify patterns in astronomical data', *Monthly Notices of the Royal Astronomical Society* **490**(4), 5904–5920.

Chopard, B. & Tomassini, M. (2018), *An Introduction to Metaheuristics for Optimization*, Springer.

Çiftçi, M. E. & Özkır, V. (2020), 'Optimising flight connection times in airline bank structure through simulated annealing and tabu search algorithms', *Journal of Air Transport Management* **87**, 101858.

Cohen, D. M., Dalal, S. R., Kajla, A. & Patton, G. C. (1994), The automatic efficient test generator (AETG) system, *in* '5th International Symposium on Software Reliability Engineering (ISSRE 1994)', IEEE, pp. 303–309.

Cohen, M., Colbourn, C. & Ling, A. (2003), Augmenting simulated annealing to build interaction test suites, *in* '14th International Symposium on Software Reliability Engineering (ISSRE 2003)', IEEE.

Cohen, M., Dwyer, M. & Shi, J. (2008), 'Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach', *IEEE Transactions on Software Engineering* **34**(5), 633–650.

Connolly, D. T. (1990), 'An improved annealing scheme for the QAP', *European Journal of Operational Research* **46**(1), 93–100.

Cook, W. J., Applegate, D. L., Bixby, R. E. & Chvátal, V. (2011), *The Traveling Salesman Problem*, Princeton University Press.

Cordeau, J.-F., Laporte, G. & Pasin, F. (2008), 'Iterated tabu search for the car sequencing problem', *European Journal of Operational Research* **191**(3), 945–956.

Cordón, O. & Damas, S. (2006), 'Image registration with iterated local search', *Journal of Heuristics* **12**(1-2), 73–94.

Cordón, O., Damas, S. & Bardinet, E. (2003), 2D image registration with iterated local search, *in* 'Advances in Soft Computing', Springer, pp. 233–242.

Corte, A. D. & Sörensen, K. (2016), 'An iterated local search algorithm for water distribution network design optimization', *Networks* **67**(3), 187–198.

Damaševičius, R. & Woźniak, M. (2017), State flipping based hyper-heuristic for hybridization of nature inspired algorithms, *in* 'Artificial Intelligence and Soft Computing', Springer, pp. 337–346.

Dang, N. & Doerr, C. (2019), Hyper-parameter tuning for the (1+(λ, λ)) GA, *in* 'Proceedings of the 2015 on Genetic and Evolutionary Computation Conference (GECCO'19)', ACM.

Dang, N. T. T. & Causmaecker, P. D. (2016), Characterization of neighborhood behaviours in a multi-neighborhood local search algorithm, *in* P. Festa, M. Sellmann & J. Vanschoren, eds, 'International Conference on Learning and Intelligent Optimization (LION'2016)', Springer, Cham, pp. 234–239.

Dantzig, G. (1963), *Linear Programming and Extensions*, RAND Corporation.

Demeester, P., Bilgin, B., Causmaecker, P. D. & Berghe, G. V. (2011), 'A hyperheuristic approach to examination timetabling problems: benchmarks and a new problem from practice', *Journal of Scheduling* **15**(1), 83–103.

Demeester, P., Souffriau, W., Causmaecker, P. D. & Berghe, G. V. (2010), 'A hybrid tabu search algorithm for automatically assigning patients to beds', *Artificial Intelligence in Medicine* **48**(1), 61–70.

den Besten, W., Thierens, D. & Bosman, P. A. N. (2016), The multiple insertion pyramid: A fast parameter-less population scheme, *in* 'Parallel Problem Solving from Nature (PPSN XIV)', Springer, pp. 48–58.

Dhouib, S. (2010), A multi start great deluge metaheuristic for engineering design problems, *in* 'ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010', IEEE.

Diabat, A., Abdallah, T. & Le, T. (2014), 'A hybrid tabu search based heuristic for the periodic distribution inventory problem with perishable goods', *Annals of Operations Research* **242**(2), 373–398.

Dickey, J. W. & Hopkins, J. W. (1972), 'Campus building arrangement using topaz', *Transportation Research* **6**(1), 59–68.

Doerr, B. (2011), Analyzing randomized search heuristics: Tools from probability theory, *in* 'Series on Theoretical Computer Science', WORLD SCIENTIFIC, pp. 1–20.

Doerr, B. & Doerr, C. (2015), Optimal parameter choices through self-adjustment, *in* 'Proceedings of the 2015 on Genetic and Evolutionary Computation Conference (GECCO'15)', ACM.

Doerr, B. & Doerr, C. (2019), Theory of parameter control for discrete black-box optimization: Provable performance gains through dynamic parameter choices, *in* 'Natural Computing Series', Springer, pp. 271–321.

Doerr, C. & Wagner, M. (2018), Simple on-the-fly parameter selection mechanisms for two classical discrete black-box optimization benchmark problems, *in* 'Proceedings of the 2015 on Genetic and Evolutionary Computation Conference (GECCO'18)', ACM.

Drake, J. H., Kheiri, A., Özcan, E. & Burke, E. K. (2020), 'Recent advances in selection hyper-heuristics', *European Journal of Operational Research* **285**(2), 405–428.

Drake, J. H., Özcan, E. & Burke, E. K. (2016), 'A case study of controlling crossover in a selection hyper-heuristic framework using the multidimensional knapsack problem', *Evolutionary Computation* **24**(1), 113–141.

Dueck, G. (1993), 'New optimization heuristics: The Great Deluge algorithm and the record-to-record travel', *Journal of Computational Physics* **104**(1), 86–92.

Dueck, G. & Scheuer, T. (1990), 'Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing', *Journal of Computational Physics* **90**(1), 161–175.

Eiben, A., Hinterding, R. & Michalewicz, Z. (1999), 'Parameter control in evolutionary algorithms', *IEEE Transactions on Evolutionary Computation* **3**(2), 124–141.

Eiben, A. & Smit, S. (2011), 'Parameter tuning for configuring and analyzing evolutionary algorithms', *Swarm and Evolutionary Computation* **1**(1), 19–31.

Emden-Weinert, T. & Proksch, M. (1999), 'Best practice simulated annealing for the airline crew scheduling problem', *Journal of Heuristics* **5**(4), 419–436.

Eng, K., Muhammed, A., Mohamed, M. A. & Hasan, S. (2020), 'A hybrid heuristic of variable neighbourhood descent and great deluge algorithm for efficient task scheduling in grid computing', *European Journal of Operational Research* **284**(1), 75–86.

Feo, T. A. & Resende, M. G. C. (1995), 'Greedy randomized adaptive search procedures', *Journal of Global Optimization* **6**(2), 109–133.

Fernandez-Viagas, V., Ruiz, R. & Framinan, J. M. (2017), 'A new vision of approximate methods for the permutation flowshop to minimise makespan: State-of-the-art and computational evaluation', *European Journal of Operational Research* **257**(3), 707–721.

Fonseca, G. H. G., Santos, H. G. & Carrano, E. G. (2016), 'Late acceptance hill-climbing for high school timetabling', *Journal of Scheduling* **19**(4), 453–465.

Framinan, J. M., Leisten, R. & García, R. R. (2014), *Manufacturing Scheduling Systems*, Springer.

Frank, J., Cheeseman, P. & Stutz, J. (1997), 'When gravity fails: Local search topology', *Journal of Artificial Intelligence Research* **7**, 249–281.

Franzin, A. & Stützle, T. (2018), Comparison of acceptance criteria in randomized local searches, *in* 'Lecture Notes in Computer Science', Springer, pp. 16–29.

Freisleben, B. & Merz, P. (2000), 'Fitness landscape analysis and memetic algorithms for the quadratic assignment problem', *IEEE Transactions on Evolutionary Computation* **4**(4), 337–352.

Frøseth, G. T. & Rönnquist, A. (2019), 'Finding the train composition causing greatest fatigue damage in railway bridges by late acceptance hill climbing', *Engineering Structures* **196**, 109342.

Galinier, P., Kpodjedo, S. & Antoniol, G. (2017), A penalty-based tabu search for constrained covering arrays, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'17)', ACM Press.

Garibaldi, J. M. & Ifeachor, E. C. (1999), 'Application of simulated annealing fuzzy model tuning to umbilical cord acid-base interpretation', *IEEE Transactions on Fuzzy Systems* **7**(1), 72–84.

Garlík, B. & Křivan, M. (2013), 'Renewable energy unit commitment, with different acceptance of balanced power, solved by simulated annealing', *Energy and Buildings* **67**, 392–402.

Garvin, B. J., Cohen, M. B. & Dwyer, M. B. (2009), An improved meta-heuristic search for constrained interaction testing, *in* '2009 1st International Symposium on Search Based Software Engineering', IEEE.

Garvin, B. J., Cohen, M. B. & Dwyer, M. B. (2011), 'Evaluating improvements to a meta-heuristic search for constrained interaction testing', *Empirical Software Engineering* **16**(1), 61–102.

Gendreau, M. & Potvin, J.-Y. (2013), Tabu search, *in* E. K. Burke & G. Kendall, eds, 'Search Methodologies', 2 edn, Springer, chapter 9, pp. 243–263.

Geoffrion, A. M. & Graves, G. W. (1976), 'Scheduling parallel production lines with changeover costs: Practical application of a quadratic assignment/LPApproach', *Operations Research* **24**(4), 595–610.

Ghosh, K., Sharma, R. & Chaudhury, P. (2020), 'Structure elucidation and construction of isomerisation pathways in small to moderate-sized (6–27) MgO nanoclusters: an adaptive mutation simulated annealing based analysis with quantum chemical calculations', *Physical Chemistry Chemical Physics* **22**(17), 9616–9629.

Gilli, M. & Schumann, E. (2011), 'Heuristic optimisation in financial modelling', *Annals of Operations Research* **193**(1), 129–158.

Glover, F. (1989), 'Tabu search — Part I', *ORSA Journal on Computing* **1**(3), 190–206.

Glover, F. W. & Laguna, M. (1997), *Tabu Search*, Kluwer Academic Publishers.

Goerler, A., Lalla-Ruiz, E. & Voß, S. (2020), 'Late acceptance hill-climbing matheuristic for the general lot sizing and scheduling problem with rich constraints', *Algorithms* **13**(6), 138.

Goerler, A., Schulte, F. & Voß, S. (2013), An application of late acceptance hill-climbing to the traveling purchaser problem, *in* 'Lecture Notes in Computer Science', Springer, pp. 173–183.

Goldberg, D. (1989), *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, Reading, Mass.

Grobler, J., Engelbrecht, A. P., Kendall, G. & Yadavalli, V. (2015), 'Heuristic space diversity control for improved meta-hyper-heuristic performance', *Information Sciences* **300**, 49–62.

Guizzo, G., Bazargani, M., Paixao, M. & Drake, J. H. (2017), A hyper-heuristic for multi-objective integration and test ordering in google guava, *in* 'Search Based Software Engineering', Springer, pp. 168–174.

Gutin, G. & Punnen, A. P., eds (2007), *The Traveling Salesman Problem and Its Variations*, Springer.

Habib, A., Vernin, J., Benkhaldoun, Z. & Lanteri, H. (2006), 'Single star scidar: atmospheric parameters profiling using the simulated annealing algorithm', *Monthly Notices of the Royal Astronomical Society* **368**(3), 1456–1462.

Hachemi, N. E., Gendreau, M. & Rousseau, L.-M. (2013), 'A heuristic to solve the synchronized log-truck scheduling problem', *Computers & Operations Research* **40**(3), 666–673.

Hanafi, R. & Kozan, E. (2014), 'A hybrid constructive heuristic and simulated annealing for railway crew scheduling', *Computers & Industrial Engineering* **70**, 11–19.

Harik, G., Cantú-Paz, E., Goldberg, D. E. & Miller, B. L. (1999), 'The gambler's ruin problem, genetic algorithms, and the sizing of populations', *Evolutionary Computation* **7**(3), 231–253.

Harik, G. R. & Lobo, F. G. (1999), A parameter-less genetic algorithm, *in* 'Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation (GECCO'99)', Vol. 1, Morgan Kaufmann Publishers, pp. 258–265.

Heid, W., Hasle, G., & Vigo, D. (2014), VeRoLog solver challenge 2014 – VSC2014 problem description, *in* 'VeRoLog (EURO Working Group on Vehicle Routing and Logistics Optimization) and PTV Group', pp. 1–6.

Hernando, L., Daolio, F., Veerapen, N. & Ochoa, G. (2017), Local optima networks of the permutation flowshop scheduling problem: Makespan vs. total flow time, *in* 'IEEE Congress on Evolutionary Computation (CEC 2017)', IEEE.

Holdener, E. (2008), The art of parameterless evolutionary algorithms, PhD thesis, Missouri University of Science and Technology, Rolla, MO, USA.

Hoos, H. H. & Stützle, T. (2005), *Stochastic Local Search: Foundations & Applications*, Elsevier / Morgan Kaufmann.

Hoos, H. H. & Stützle, T. (2015), Stochastic local search algorithms: An overview, *in* 'Springer Handbook of Computational Intelligence', Springer, pp. 1085–1105.

Hu, T. C., Kahng, A. B. & Tsao, C.-W. A. (1995), 'Old bachelor acceptance: A new class of non-monotone threshold accepting methods', *ORSA Journal on Computing* **7**(4), 417–425.

Islami, A., Chaimatanan, S. & Delahaye, D. (2017), Large-scale 4D trajectory planning, *in* 'Air Traffic Management and Systems II', Vol. 420 of *Lecture Notes in Electrical Engineering*, Springer, pp. 27–47.

Jacob, D., Raben, A., Sarkar, A., Grimm, J. & Simpson, L. (2008), 'Anatomy-based inverse planning simulated annealing optimization in high-dose-rate prostate brachytherapy: Significant dosimetric advantage over other optimization techniques', *International Journal of Radiation Oncology∗Biology∗Physics* **72**(3), 820–827.

Jaffar, J. & Lassez, J.-L. (1987), Constraint logic programming, *in* 'Proceedings of the 14th Symposium on Principles of Programming Languages (POPL'87)', ACM Press.

Jansen, T. (2013), *Analyzing Evolutionary Algorithms*, Springer.

Jia, Y., Cohen, M. B., Harman, M. & Petke, J. (2015), Learning combinatorial interaction test generation strategies using hyperheuristic search, *in* 'IEEE/ACM International Conference on Software Engineering', IEEE.

Johnson, D. S., Aragon, C. R., McGeoch, L. A. & Schevon, C. (1989), 'Optimization by simulated annealing: An experimental evaluation; Part I, graph partitioning', *Operations Research* **37**(6), 865–892.

Johnson, D. S., Aragon, C. R., McGeoch, L. A. & Schevon, C. (1991), 'Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning', *Operations Research* **39**(3), 378–406.

Jones, T. & Forrest, S. (1995), Fitness distance correlation as a measure of problem difficulty for genetic algorithms, *in* 'Proceedings of the 6th International Conference on Genetic Algorithms', Morgan Kaufmann Publishers, p. 184–192.

Jünger, M., Reinelt, G. & Rinaldi, G. (1995), The traveling salesman problem, *in* 'Handbooks in Operations Research and Management Science', Elsevier, pp. 225–330.

Katsigiannis, Y. A., Georgilakis, P. S. & Karapidakis, E. S. (2012), 'Hybrid simulated annealing–tabu search method for optimal sizing of autonomous power systems with renewables', *IEEE Transactions on Sustainable Energy* **3**(3), 330–338.

Kendall, G. & Mohamad, M. (2004), Channel assignment in cellular communication using a great deluge hyper-heuristic, *in* '12th IEEE International Conference on Networks (ICON 2004)', IEEE.

Khalsa, S. K. & Labiche, Y. (2014), An orchestrated survey of available algorithms and tools for combinatorial testing, *in* 'IEEE 25th International Symposium on Software Reliability Engineering (ISSRE 2014)', IEEE.

Kheiri, A. & Özcan, E. (2013), 'Constructing constrained-version of magic squares using selection hyper-heuristics', *The Computer Journal* **57**(3), 469–479.

Kifah, S. & Abdullah, S. (2015), 'An adaptive non-linear great deluge algorithm for the patient-admission problem', *Information Sciences* **295**, 573–585.

Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P. (1983), 'Optimization by simulated annealing', *Science* **220**(4598), 671–680.

Knuth, D. (2014), *The Art of Computer Programming, Volume 4A, The: Combinatorial Algorithms, Part 1*, Addison-Wesley, Boston, MA.

Knuth, D. E. (2019), *The Art of Computer Programming, Volume 4, Fascicle 5: Mathematical Preliminaries Redux; Introduction to Backtracking; Dancing Links*, Addison-Wesley, Boston, MA.

Kong, Q., Kuriyan, K., Shah, N. & Guo, M. (2019), 'Development of a responsive optimisation framework for decision-making in precision agriculture', *Computers & Chemical Engineering* **131**, 106585.

Koza, J. R. (1994), 'Genetic programming as a means for programming computers by natural selection', *Statistics and Computing* **4**(2).

Kramer, O. (2017), *Genetic Algorithm Essentials*, Springer.

Kubicky, C. D., Yeh, B. M., Lessard, E., Joe, B. N., Speight, J. L., Pouliot, J. & Hsu, I.-C. (2008), 'Inverse planning simulated annealing for magnetic resonance imaging-based intracavitary high-dose-rate brachytherapy for cervical cancer', *Brachytherapy* **7**(3), 242–247.

Kuhn, D., Wallace, D. & Gallo, A. (2004), 'Software fault interactions and implications for software testing', *IEEE Transactions on Software Engineering* **30**(6), 418–421.

Land, A. H. & Doig, A. G. (1960), 'An automatic method of solving discrete programming problems', *Econometrica* **28**(3), 497.

Landa-Silva, D. & Obit, J. H. (2009), Evolutionary non-linear great deluge for university course timetabling, *in* '4th International Conference on Hybrid Artificial Intelligence Systems (HAIS 2009)', Lecture Notes in Computer Science, Springer, pp. 269–276.

Laporte, G. (1992), 'The traveling salesman problem: An overview of exact and approximate algorithms', *European Journal of Operational Research* **59**(2), 231–247.

Larrañaga, P., Kuijpers, C., Murga, R., Inza, I. & Dizdarevic, S. (1999), 'Genetic algorithms for the travelling salesman problem: A review of representations and operators', *Artificial Intelligence Review* **13**(2), 129–170.

Lee, C. & Kang, H. (2000), 'Cell planning with capacity expansion in mobile communications: a tabu search approach', *IEEE Transactions on Vehicular Technology* **49**(5), 1678–1691.

Lee, D. S., Vassiliadis, V. S. & Park, J. M. (2004), 'A novel threshold accepting meta-heuristic for the job-shop scheduling problem', *Computers & Operations Research* **31**(13), 2199–2213.

Leite, N., Melício, F. & Rosa, A. C. (2019), 'A fast simulated annealing algorithm for the examination timetabling problem', *Expert Systems with Applications* **122**, 137–151.

Leutner, M., Gschwind, R. M., Liermann, J., Schwarz, C., Gemmecker, G. & Kessler, H. (1998), 'Automated backbone assignment of labeled proteins using the threshold accepting algorithm', *Journal of Biomolecular NMR* **11**(1), 31–43.

Li, Y., Wang, C., Gao, L., Song, Y. & Li, X. (2020), 'An improved simulated annealing algorithm based on residual network for permutation flow shop scheduling', *Complex & Intelligent Systems* .

Lin, B., Chavali, S., Camarda, K. & Miller, D. (2005), 'Computer-aided molecular design using tabu search', *Computers & Chemical Engineering* **29**(2), 337–347.

Lin, J., Luo, C., Cai, S., Su, K., Hao, D. & Zhang, L. (2015), TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (t), *in* '30th IEEE/ACM International Conference on Automated Software Engineering (ASE)', IEEE.

Liu, H., Zhang, J., Zhang, X., Kurniawan, A., Juhana, T. & Ai, B. (2020), 'Tabu-search-based pilot assignment for cell-free massive MIMO systems', *IEEE Transactions on Vehicular Technology* **69**(2), 2286–2290.

Liu, Y.-H. (2010), 'A genetic local search algorithm with a threshold accepting mechanism for solving the runway dependent aircraft landing problem', *Optimization Letters* **5**(2), 229–245.

Lobo, F. G. & Bazargani, M. (2015), When hillclimbers beat genetic algorithms in multimodal optimization, *in* 'Proceedings of the Companion Publication of the Genetic and Evolutionary Computation Conference (GECCO'15)', ACM Press.

Lobo, F. G., Bazargani, M. & Burke, E. K. (2020), 'A cutoff time strategy based on the coupon collector's problem', *European Journal of Operational Research* **286**(1), 101–114.

López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M. & Stützle, T. (2016), 'The irace package: Iterated racing for automatic algorithm configuration', *Operations Research Perspectives* **3**, 43–58.

Lourenço, H. R., Martin, O. C. & Stützle, T. (2019), Iterated local search: Framework and applications, *in* M. Gendreau & J.-Y. Potvin, eds, 'Handbook of Metaheuristics', 3 edn, Springer, chapter 5, pp. 129–168.

Luong, N. H., Poutré, H. L. & Bosman, P. A. (2015), Exploiting linkage information and problem-specific knowledge in evolutionary distribution network expansion planning, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'15)', ACM Press.

Maenhout, B. & Vanhoucke, M. (2010), 'A hybrid scatter search heuristic for personalized crew rostering in the airline industry', *European Journal of Operational Research* **206**(1), 155–167.

Mafarja, M. & Abdullah, S. (2014), Fuzzy modified great deluge algorithm for attribute reduction, *in* 'Recent Advances on Soft Computing and Data Mining', Vol. 287 of *Advances in Intelligent Systems and Computing*, Springer, pp. 195–203.

Marimuthu, S., Ponnambalam, S. & Jawahar, N. (2009), 'Threshold accepting and ant-colony optimization algorithms for scheduling m-machine flow shops with lot streaming', *Journal of Materials Processing Technology* **209**(2), 1026–1041.

Marmion, M.-E., Dhaenens, C., Jourdan, L., Liefooghe, A. & Verel, S. (2011), On the neutrality of flowshop scheduling fitness landscapes, *in* 'Lecture Notes in Computer Science', Springer, pp. 238–252.

Martí, R., Aceves, R., León, M. T., Moreno-Vega, J. M. & Duarte, A. (2018), Intelligent multi-start methods, *in* M. Gendreau & J.-Y. Potvin, eds, 'Handbook of Metaheuristics', 3 edn, Springer, chapter 7, pp. 221–243.

Martinho, W. C. S., Melo, R. A. & Sörensen, K. (2020), An enhanced simulation-based iterated local search metaheuristic for gravity fed water distribution network design optimization, Technical Report 2009.01197, arXiv.

Matsopoulos, G. K., Mouravliansky, N. A., Delibasis, K. K. & Nikita, K. S. (1999), 'Automatic retinal image registration scheme using global optimization techniques', *IEEE Transactions on Information Technology in Biomedicine* **3**(1), 47–60.

McCollum, B., McMullan, P., Parkes, A. J., Burke, E. K. & Abdullah, S. (2009), An extended great deluge approach to the examinationtimetabling problem, *in* '4th Multidisciplinary International Scheduling Conference – Tools & Applications (MISTA 2009)'.

McDermott, J. & Nicolau, M. (2017), Late-acceptance hill-climbing with a grammatical program representation, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO'17)', ACM.

McMullan, P. (2007), An extended implementation of the great deluge algorithm for course timetabling, *in* '7th International Conference on Computational Science (ICCS 2007)', Vol. 4487 of *Lecture Notes in Computer Science*, Springer, pp. 538–545.

McMullan, P. & McCollum, B. (2007), Dynamic job scheduling on the grid environment using the great deluge algorithm, *in* '9th International Conference on Parallel Computing Technologies (PaCT 2007)', Lecture Notes in Computer Science, Springer, pp. 283–292.

Mohmad Kahar, M. N. & Kendall, G. (2015), 'A great deluge algorithm for a real-world examination timetabling problem', *Journal of the Operational Research Society* **66**(1), 116–133.

Moscato, P. & Fontanari, J. (1990), 'Stochastic versus deterministic update in simulated annealing', *Physics Letters A* **146**(4), 204–208.

Motwani, R. & Raghavan, P. (1995), *Randomized Algorithms*, Cambridge University Press, New York, NY, USA.

Nahas, N., Khatab, A., Ait-Kadi, D. & Nourelfath, M. (2008), 'Extended great deluge algorithm for the imperfect preventive maintenance optimization of multi-state systems', *Reliability Engineering & System Safety* **93**(11), 1658–1672.

Nawaz, M., Enscore, E. E. & Ham, I. (1983), 'A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem', *Omega* **11**(1), 91–95.

Nie, C. & Leung, H. (2011), 'A survey of combinatorial testing', *ACM Computing Surveys* **43**(2), 1–29.

Nissen, V. & Paul, H. (1995), 'A modification of threshold accepting and its application to the quadratic assignment problem', *OR Spektrum* **17**(2-3), 205–210.

Nourelfath, M., Nahas, N. & Montreuil, B. (2007), 'Coupling ant colony optimization and the extended great deluge algorithm for the discrete facility layout problem', *Engineering Optimization* **39**(8), 953–968.

Ochoa, G. & Herrmann, S. (2018), Perturbation strength and the global structure of QAP fitness landscapes, *in* 'Parallel Problem Solving from Nature (PPSN XV)', Springer, pp. 245–256.

Ozcan, E., Bykov, Y., Birben, M. & Burke, E. K. (2009), Examination timetabling using late acceptance hyper-heuristics, *in* '2009 IEEE Congress on Evolutionary Computation', IEEE.

Pagnozzi, F. & Stützle, T. (2019), 'Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems', *European Journal of Operational Research* **276**(2), 409–421.

Pastore, T., Martínez-Gavara, A., Napoletano, A., Festa, P. & Martí, R. (2020), 'Tabu search for min-max edge crossing in graphs', *Computers & Operations Research* **114**, 104830.

Pelikan, M. & Lin, T.-K. (2004), Parameter-less hierarchical BOA, *in* 'Genetic and Evolutionary Computation (GECCO'04)', Springer, pp. 24–35.

Penna, P. H. V., Subramanian, A. & Ochi, L. S. (2011), 'An iterated local search heuristic for the heterogeneous fleet vehicle routing problem', *Journal of Heuristics* **19**(2), 201–232.

Petrovic, S. & Bykov, Y. (2003), A multiobjective optimisation technique for exam timetabling based on trajectories, *in* '4th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2002)', Vol. 2740 of *Lecture Notes in Computer Science*, Springer, pp. 181–194.

Rao, R. & Iyengar, S. (1994), 'Bin-packing by simulated annealing', *Computers & Mathematics with Applications* **27**(5), 71–82.

Ravi, V. & Zimmermann, H.-J. (2000), 'Fuzzy rule based classification with FeatureSelector and modified threshold accepting', *European Journal of Operational Research* **123**(1), 16–28.

Reeves, C. R. (2013), Fitness landscapes, *in* E. K. Burke & G. Kendall, eds, 'Search Methodologies', 2 edn, Springer, chapter 22, pp. 681–705.

Reinelt, G. (1991), 'TSPLIB — a traveling salesman problem library', *ORSA Journal on Computing* **3**(4), 376–384.

Ribeiro, C. C., Aloise, D., Noronha, T. F., Rocha, C. & Urrutia, S. (2008), 'A hybrid heuristic for a multi-objective real-life car sequencing problem with painting and assembly line constraints', *European Journal of Operational Research* **191**(3), 981–992.

Ruiz, R. & Maroto, C. (2005), 'A comprehensive review and evaluation of permutation flowshop heuristics', *European Journal of Operational Research* **165**(2), 479–494.

Ruiz, R. & Stützle, T. (2007), 'A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem', *European Journal of Operational Research* **177**(3), 2033–2049.

Russell, S. & Norvig, P. (2016), *Artificial Intelligence: A Modern Approach, Global Edition*, Pearson Higher Education.

Sahni, S. & Gonzalez, T. (1976), 'P-complete approximation problems', *Journal of the ACM (JACM)* **23**(3), 555–565.

Sartori, C. S. & Buriol, L. S. (2018), A matheuristic approach to the pickup and delivery problem with time windows, *in* 'Lecture Notes in Computer Science', Springer, pp. 253–267.

Schiavinotto, T. & Stützle, T. (2004), 'The linear ordering problem: Instances, search space analysis and algorithms', *Journal of Mathematical Modelling and Algorithms* **3**(4), 367–402.

Sechen, C. (1988), *VLSI Placement and Global Routing Using Simulated Annealing*, Springer.

Seçkiner, S. U. & Kurt, M. (2007), 'A simulated annealing approach to the solution of job rotation scheduling problems', *Applied Mathematics and Computation* **188**(1), 31–45.

Selman, B., Levesque, H. J. & Mitchell, D. G. (1992), A new method for solving hard satisfiability problems, *in* W. R. Swartout, ed., 'Proceedings of the 10th National Conference on Artificial Intelligence', AAAI Press / The MIT Press, pp. 440–446.

Sharma, N., Ray, A. K., Sharma, S., Shukla, K., Aggarwal, L. M. & Pradhan, S. (2009), 'Segmentation of medical images using simulated annealing based fuzzy c means algorithm', *International Journal of Biomedical Engineering and Technology* **2**(3), 260.

Shunmugathammal, M., Columbus, C. C. & Anand, S. (2019), 'A novel b∗tree crossover-based simulated annealing algorithm for combinatorial optimization in VLSI fixed-outline floorplans', *Circuits, Systems, and Signal Processing* **39**(2), 900–918.

Silva, M. M., Subramanian, A. & Ochi, L. S. (2015), 'An iterated local search heuristic for the split delivery vehicle routing problem', *Computers & Operations Research* **53**, 234–249.

Smit, S. & Eiben, A. (2009), Comparing parameter tuning methods for evolutionary algorithms, *in* 'IEEE Congress on Evolutionary Computation (CEC 2009)', IEEE.

Smit, S. K. & Eiben, A. E. (2010), Parameter tuning of evolutionary algorithms: Generalist vs. specialist, *in* 'European Conference on the Applications of Evolutionary Computation (EvoApplications 2010)', Vol. 6024 of *Lecture Notes in Computer Science*, Springer, pp. 542–551.

Smorodkina, E. & Tauritz, D. (2007), Greedy population sizing for evolutionary algorithms, *in* 'IEEE Congress on Evolutionary Computation (CEC 2007)', IEEE.

Song, T., Liu, S., Tang, X., Peng, X. & Chen, M. (2018), 'An iterated local search algorithm for the university course timetabling problem', *Applied Soft Computing* **68**, 597–608.

Soria-Alcaraz, J. A., Özcan, E., Swan, J., Kendall, G. & Carpio, M. (2016), 'Iterated local search using an add and delete hyper-heuristic for university course timetabling', *Applied Soft Computing* **40**, 581–593.

Soykan, B. & Rabadi, G. (2016), A tabu search algorithm for the multiple runway aircraft scheduling problem, *in* 'International Series in Operations Research & Management Science', Springer, pp. 165–186.

Stützle, T. & Ruiz, R. (2018), Iterated local search, *in* 'Handbook of Heuristics', Springer, pp. 579–605.

Sudholt, D. (2015), Parallel evolutionary algorithms, *in* 'Handbook of Computational Intelligence', Springer, pp. 929–959.

Taillard, E. (1990), 'Some efficient heuristic methods for the flow shop sequencing problem', *European Journal of Operational Research* **47**(1), 65–74.

Taillard, E. (1993), 'Benchmarks for basic scheduling problems', *European Journal of Operational Research* **64**(2), 278–285.

Tarantilis, C., Kiranoudis, C. & Vassiliadis, V. (2002), 'A backtracking adaptive threshold accepting algorithm for the vehicle routing problem', *Systems Analysis Modelling Simulation* **42**(5), 631–664.

Tarantilis, C., Kiranoudis, C. & Vassiliadis, V. (2004), 'A threshold accepting metaheuristic for the heterogeneous fixed fleet vehicle routing problem', *European Journal of Operational Research* **152**(1), 148–158.

Tayarani-N., M.-H. & Prügel-Bennett, A. (2015), 'Quadratic assignment problem: a landscape analysis', *Evolutionary Intelligence* **8**(4), 165–184.

Thompson, J. M. & Dowsland, K. A. (1996), 'Variants of simulated annealing for the examination timetabling problem', *Annals of Operations Research* **63**(1), 105–128.

Thompson, J. M. & Dowsland, K. A. (1998), 'A robust simulated annealing based examination timetabling system', *Computers & Operations Research* **25**(7-8), 637–648.

Tinkle, C. L., Weinberg, V., Chen, L.-M., Littell, R., Cunha, J. A. M., Sethi, R. A., Chan, J. K. & Hsu, I.-C. (2015), 'Inverse planned high-dose-rate brachytherapy for locoregionally advanced cervical cancer: 4-year outcomes', *International Journal of Radiation Oncology∗Biology∗Physics* **92**(5), 1093–1100.

Toffolo, T. A., Christiaens, J., Malderen, S. V., Wauters, T. & Berghe, G. V. (2018), 'Stochastic local search with learning automaton for the swap-body vehicle routing problem', *Computers & Operations Research* **89**, 68–81.

Tubic, D., Zaccarin, A., Beaulieu, L. & Pouliot, J. (2001), 'Automated seed detection and three-dimensional reconstruction. II. reconstruction of permanent prostate implants using simulated annealing', *Medical Physics* **28**(11), 2272–2279.

Turky, A., Sabar, N. R., Dunstall, S. & Song, A. (2018), Hyper-heuristic based local search for combinatorial optimisation problems, *in* 'AI 2018: Advances in Artificial Intelligence', Springer, pp. 312–317.

Turky, A., Sabar, N. R., Sattar, A. & Song, A. (2016), Parallel late acceptance hill-climbing algorithm for the google machine reassignment problem, *in* 'AI 2016: Advances in Artificial Intelligence', Springer, pp. 163–174.

Václavík, R., Šůcha, P. & Hanzálek, Z. (2016), 'Roster evaluation based on classifiers for the nurse rostering problem', *Journal of Heuristics* **22**(5), 667–697.

Vallada, E., Ruiz, R. & Framinan, J. M. (2015), 'New hard benchmark for flowshop scheduling problems minimising makespan', *European Journal of Operational Research* **240**(3), 666–677.

Vancroonenburg, W. & Wauters, T. (2013), Extending the late acceptance metaheuristic formulti-objective optimization, *in* '6th Multidisciplinary International Scheduling Conference (MISTA 2013)'.

Verstichel, J. & Berghe, G. V. (2009), A late acceptance algorithm for the lock scheduling problem, *in* 'Logistik Management', Physica-Verlag HD, pp. 457–478.

Wang, C., Mu, D., Zhao, F. & Sutherland, J. W. (2015), 'A parallel simulated annealing method for the vehicle routing problem with simultaneous pickup–delivery and time windows', *Computers & Industrial Engineering* **83**, 111–122.

Wauters, T., Toffolo, T., Christiaens, J. & Malderen, S. V. (2015), The winning approach for the VeRoLog solver challenge 2014: the swap-body vehicle routing problem, *in* '29th Belgian Conference on Operations Research (OR)'.

Wauters, T., Verstichel, J. & Berghe, G. V. (2013), 'An effective shaking procedure for 2D and 3D strip packing problems', *Computers & Operations Research* **40**(11), 2662–2669.

Webb, S. (1989), 'Optimisation of conformal radiotherapy dose distribution by simulated annealing', *Physics in Medicine and Biology* **34**(10), 1349–1370.

Weiszer, M., Burke, E. K. & Chen, J. (2020), 'Multi-objective routing and scheduling for airport ground movement', *Transportation Research Part C: Emerging Technologies* **119**, 102734.

Whitley, D., Sutton, A. M. & Howe, A. E. (2008), Understanding elementary landscapes, *in* 'Proceedings of the Gnetic and Evolutionary Computation Conference (GECCO'08)', ACM Press.

Winker, P. (2000), *Optimization Heuristics in Econometrics : Applications of Threshold Accepting*, John Wiley & Sons.

Winker, P. & Fang, K.-T. (1997), 'Application of threshold-accepting to the evaluation of the discrepancy of a set of points', *SIAM Journal on Numerical Analysis* **34**(5), 2028–2042.

Wu, Q., Wang, Y. & Glover, F. (2020), 'Advanced tabu search algorithms for bipartite boolean quadratic programs guided by strategic oscillation and path relinking', *INFORMS Journal on Computing* **32**(1), 74–89.

Xu, J., Chiu, S. Y. & Glover, F. (1999), 'Optimizing a ring-based private line telecommunication network using tabu search', *Management Science* **45**(3), 330–345.

Yang, B., Xu, W. & Dong, Z. (2013), 'Automated extraction of building outlines from airborne laser scanning point clouds', *IEEE Geoscience and Remote Sensing Letters* **10**(6), 1399–1403.

Yassen, E. T., Ayob, M., Nazri, M. Z. A. & Sabar, N. R. (2017), 'An adaptive hybrid algorithm for vehicle routing problems with time windows', *Computers & Industrial Engineering* **113**, 382–391.

Ying, K.-C. & Lin, S.-W. (2020), 'Solving no-wait job-shop scheduling problems using a multi-start simulated annealing with bi-directional shift timetabling algorithm', *Computers & Industrial Engineering* **146**, 106615.

Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R. N. & Kuhn, D. R. (2013), An efficient algorithm for constraint handling in combinatorial test generation, *in* 'IEEE Sixth International Conference on Software Testing, Verification and Validation', IEEE.

Zagré, G. E., Marcotte, D., Gamache, M. & Guibault, F. (2018), 'New tabu algorithm for positioning mining drillholes with blocks uncertainty', *Natural Resources Research* **28**(3), 609–629.

Zamli, K. Z., Alkazemi, B. Y. & Kendall, G. (2016), 'A tabu search hyper-heuristic strategy for t-way test suite generation', *Applied Soft Computing* **44**, 57–74.

Zhai, S., Hunter, M. & Smith, B. A. (2002), 'Performance optimization of virtual keyboards', *Human–Computer Interaction* **17**(2-3), 229–269.

Zhang, G., Wu, B., Maleki, A. & Zhang, W. (2018), 'Simulated annealing-chaotic search algorithm based optimization of reverse osmosis hybrid desalination system driven by wind and solar energies', *Solar Energy* **173**, 964–975.

Zhang, W., Maleki, A., Rosen, M. A. & Liu, J. (2018), 'Optimization with a simulated annealing algorithm of a hybrid system for renewable energy including battery and hydrogen storage', *Energy* **163**, 191–207.

Zhou, B.-h. & Kang, X.-y. (2018), 'A multiobjective hybrid imperialist competitive algorithm for multirobot cooperative assembly line balancing problems with energy awareness', *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* **233**(9), 2991–3003.

Zhou, Y., Wang, Z., Jin, Y. & Fu, Z.-H. (2021), 'Late acceptance-based heuristic algorithms for identifying critical nodes of weighted graphs', *Knowledge-Based Systems* **211**, 106562.