Thesis

Combining Multiple Heuristics: Studies on Neighborhood-base Heuristics and Sampling-based Heuristics

Chung-Yao Chuang

CMU-RI-TR-20-02

March, 2020

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Neighborhood-based Heuristics, Local Searches, Hyper-heuristics, Luby's Strategy, Combination of Heuristics, Heuristic Macros, Sampling-based Heuristics, Estimation of Distribution Algorithms

Abstract

This thesis centers on the topic of how to automatically combine multiple heuristics. For most computationally challenging problems, there exist multiple heuristics, and it is generally the case that any such heuristic exploits only a limited number of aspects among all the possible problem characteristics that we can think of, and by definition, is not infallible. Thus, if the situation encountered does not align well with the nature of the employed heuristic, the algorithm can progress very slowly or get trapped in a bad local optimum. In order to compensate for this, researchers have been investigating and experimenting with the idea of combining multiple heuristics. The development of this idea is also motivated by the fact that we have progressed to a point at which we started to consider more complex problems that have subproblems or facets similar to simpler and more-studied problems. In this case, it is very natural to attempt to reuse the heuristics that we developed for those more-studied problem domains. In this study, we intend to build on this approach of combining multiple heuristics. At the broadest level, we would like to explore possible ways to synthesize effective search processes for solving combinatorial optimization problems. The specific strategies we consider will be based on using existing heuristics as algorithmic components and combining them in an automatic fashion. Furthermore, this research will have an emphasis on creating and utilizing collaborations among heuristics as an underlying means. This leads to several research questions such as how to set up an environment so that collaborations among heuristics can emerge, how do we reuse the collaborated efforts, and how do we adjust the search process so that the benefit of collaboration can be amplified.

In this thesis, we develop two types of integration architectures, each of which is specific to a broad class of heuristics. The first part of this thesis focuses on studying possible ways of combining *neighborhood-based heuristics*, which operate based on the idea of iteratively searching for improvements in the neighborhood of the current solutions. We will first present a basic architecture that we use as a foundation for enabling cooperation among multiple neighborhood-based heuristics. The fundamental idea of this architecture is to chain multiple heuristics in a pipelined fashion so that we can utilize the interaction between heuristics. Based on that, we will proceed to examine some simple learning mechanisms that adjust the behavior of the search algorithm based on the collected data. Finally, we will explore how to learn more explicit collaboration patterns among the neighborhood-based heuristics, and we will evaluate the benefit of using these learned patterns in a more rigorous cross-validation assessment.

The second part of this thesis looks at how to combine multiple *sampling-based heuristics*, which compose a solution by sampling a probabilistic model that encodes the structures of potentially good candidate solutions. We will propose a method that uses a linear interpolation to combine multiple sampling-based heuristics. The weights associated with the participating heuristics are estimated automatically based on observed data and dynamically changed from iteration to iteration. Finally, by analyzing this approach, we will further distill a generalized framework for combining sampling-based heuristics.

Acknowledgments

First of all, I must thank my advisor, Stephen F. Smith, for his generous support and continuous guidance through this long journey. I really appreciate that he gave me the freedom to pursue the research topics that I was interested in, and provided valuable suggestions that really helped me shape my thesis.

I also would like to thank Dr. Gabriela Ochoa, Dr. Zachary Rubinstein and Prof. Katia Sycara for being in my thesis committee and providing me with critical comments and advice. I am especially grateful to Dr. Ochoa, for her suggestions on my initial thesis proposal. Her expertise in hyper-heuristics really helped me improve my ideas. In addition to these, I also want to express my gratitude to Prof. Ying-ping Chen, who mentored me when I was in National Chiao Tung University.

During my time in Pittsburgh, I have made a lot of friends—too many to name. I would like to express my gratitude to all of them here, for their companion and the fun time we spent together. I will always keep this cordial memory in my mind. Especially, I want to thank Dr. Ling-Wan Chen and (soon-to-be doctor) Ming Hsiao. It wouldn't be possible for me to complete my Ph.D. without you two. Also, I would like to thank my long-time office mate, Dr. Hsu-Chieh Hu, for all the academic discussions and daily chats that we had throughout the years. Finally, I would like to thank Ian Chen, Reni Liu, Frank Tseng, Tim Tai and Sam Chang, for the first two years of fun time that we spent together in Pittsburgh and for your companion to the island of Bali.

I also want to express my appreciation to all the encouragements that I got from my friends from Taiwan. In particular, I want to thank Jer-Min Hsiao, Chiio Tut, Tim Hsu, Bart Hsiao, Bill Tsao, Tsung-Han Tsai. I really miss the time that we spent together in college. In addition to that, I want to thank Cheng-Hsiu Chen and Jhih-Jie Jhan, for keeping me informed of what was happening in my lovely hometown.

Last, but certainly not least, I want to thank my family for supporting me and my decision to pursue a doctoral degree. I owe my greatest gratitude to you all, and thank you for all the things that you have done for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many computational problems that emerge from the real-world situations are NP-hard, and thus unlikely to admit efficient solving procedures from a worst-case point of view. Although these problems are widely believed to be intractable in the worst case, it is often possible to approach even very large instances of such problems with well-crafted *heuristics*. The artificial intelligence (AI) and operation research (OR) communities have achieved great success in designing and deploying heuristics. Many approaches in AI and OR rely on heuristic methods, in part or in whole, to achieve good performance.

Even though heuristic approaches typically do not have a worst-case guarantee, the resulting algorithms can nevertheless be very effective. This is because the design of these heuristics generally takes into account the particular structures of the problem domain, or special characteristics of the group of problem instances that will actually be encountered in the application.

It is also typical that for most computationally hard problems, there exist multiple heuristics. And it is generally the case that any such heuristic exploits only a limited number of aspects among all possible problem characteristics, either for the concern of program efficiency, or for the ease of implementation. As a result, if the situation encountered does not align well with the nature of the employed heuristic, the algorithm can progress very slowly or get trapped in a bad local optimum.

From this point of view, the effectiveness of a heuristic can be seen as being bounded to some particular subsets of situations. And depending on the situation, it might be the case that another heuristic, with more appropriate search bias, will progress more rapidly toward better solutions than the employed heuristic.

In order to compensate for such performance variations and make heuristic approaches more robust, AI and OR research communities have been investigating and experimenting with the idea of *combining multiple heuristics* (e.g. [14, 42, 96].) This idea can be traced back to the 1960s when Fisher and Thompson proposed the idea of combining multiple heuristic dispatching rules for production scheduling [37]. They showed that an unbiased random combination of these dispatching rules can outperform any of them taken independently. More recently, in the satisfiability domain, researchers have also taken this concept and developed successful *portfolio* approaches (e.g., [79, 105]) that utilize multiple preexisting algorithms.

The development of this concept is further motivated by the fact that the state of research has progressed to a point at which more complex problems are being considered that have facets similar to simpler and more-studied problems. In this case, it is very natural to attempt to *reuse* the heuristics developed for those more-studied problem domains. For example, we can see the line of research in routing domains moves from traveling salesman problem (TSP) to TSP with time windows, and then to vehicle routing problems and orienteering problems (also called the prize-collecting TSP.) Thus, it can be very beneficial if we are able to reuse some of the heuristics developed for TSP and for constrained scheduling.

Extending from this example, we can also see that many real-world problems are structured as several simpler subproblems coupled together. Thus, the research on how to combine multiple preexisting heuristics from different domains has a potentially broad range of applications.

1.1 Research Objectives

In this thesis, we intend to build on this view of combining multiple heuristics. At the broadest level, we would like to explore possible ways to synthesize effective search processes for solving combinatorial optimization problems. The specific strategies we consider will be based on using existing heuristics as algorithmic components and combining them in an automatic fashion. In this thesis, we place a particular focus on the automation aspect of the techniques that we develop. Many previous studies have utilized the idea of combining multiple heuristics. However, many of them use fixed, manual specifications to create the combinations. This can require significant trial-and-error and experimentation. For this thesis, we emphasize more on learning-based and data-driven approaches, and we believe that this perspective can potentially reduce the labor spent on figuring out the appropriate way to combine heuristics for solving different types of problems.

Methodologically, this research will emphasize the creation and the use of collaborations among heuristics as an underlying mechanism to assist problem solving. Generally speaking, in order for a behavior to be deemed a collaboration, we think the following two elements are essential:

1. Information passing or sharing must occur among the participating heuristics; and
2. Through this information passing or sharing, at least one of the participating heuristics will have a higher chance of achieving the intended goal, relative to the prospects that any of the constituent heuristics would have individually.

In short, we would like to create a synergy among heuristics so that better results can be achieved.

The above intention leads to several research questions such as how to setup an environment so that patterns of collaborations can emerge, how do we recognize those patterns, and once we have those patterns, how frequently should we use one pattern versus another to compose the entire search process. In this research, we intend to look into those questions and provide some possible approaches to address them.

In this thesis, we develop two types of integration architectures, each is specific to a broad class of heuristics. In the first part of this thesis, we will look at possible ways for combining *neighborhood-based heuristics*, which operate based on the idea of iteratively searching for improvements in the neighborhood of the current solution. In the second part of this thesis, we will study how to combine *sampling-based heuristics*, which compose a solution by sampling a probabilistic model that encodes the structures of potentially good candidate solutions. In both cases, we will develop methods that automatically combine the participating heuristics. And, although not explored in this thesis, these two types of heuristics can be further hybridized to create potentially more effective search procedures.

1.2 Road Map

In this chapter, we described the motivation behind this research and pointed out the objectives of this thesis. Our focus is on developing automated procedures for combining multiple heuristics and on the cooperative aspect of utilizing multiple heuristics.

In the following chapter, we will begin with a review on the background and establish some terminology that will be used throughout this thesis. We will also review some research areas related to this thesis, and contrast them with our intended research.

After the chapter on background reviews, we will begin with a study on combining multiple neighborhood-based heuristics. In Chapter 3, we first present a basic architecture that we use as a foundation for enabling cooperation among multiple neighborhood-based heuristics. The fundamental idea of this architecture is to chain multiple heuristics in a pipelined fashion so that we can utilize the interaction between heuristics. Based on that, in Chapter 4, we examine some simple learning mechanisms that adjust the behavior of the search algorithm based on the collected data. In Chapter 5, we explore how to learn more explicit collaboration patterns among the neighborhood-based heuristics. We will also discuss a setting that has a distributional assumption over the problem instances, and evaluate the learned policies with a more rigorous cross-validation assessment.

In Chapter 6, we switch to the topic of combining multiple sampling-based heuristics. We develop a method that uses a linear interpolation to combine multiple sampling-based heuristics. The weights associated with the participating heuristics are estimated automatically based on observed data and dynamically changed from iteration to iteration. Based on that, in Chapter 7, we further assess this dynamic adjustment approach by offering comparisons with other alternatives.

Finally, Chapter 8 concludes this thesis by summarizing its contents and discussing important directions for extensions.

Chapter 2

Background

In this chapter, we will provide some review on the background of this research and define two classes of heuristics that we consider in this thesis: *neighborhood-based heuristics* and *sampling-based heuristics*. We will first look at the neighborhood-based heuristics and point out why this class of heuristics are convenient to work with in a collaborative context. Following that, Section 2.3 and 2.4 review some research areas related to our study of combining neighborhood-based heuristics. Next, we briefly describe the sampling-based heuristics in Section 2.5, and characterize them as using handcrafted or estimated probabilistic models. We also review a field related to sampling-based heuristics called Estimation of Distribution Algorithms in Section 2.6. Finally, Section 2.7 summarizes this review.

2.1 Combinatorial Optimization Problems

A deterministic optimization problem can be defined as

$$\text{minimize } f(\mathbf{x}) \;\; \text{s.t. } \mathbf{x} \in \mathcal{X}, \mathcal{X} \subseteq \mathcal{S}$$

where f is a real-valued *objective function*, $\mathcal{S}$ is the *solution space* (or *search space*) that contains all the candidate solutions, and $\mathcal{X}$ is the set of *feasible solutions* of which $\mathbf{x}$ is a member. Note that we can focus on minimization without loss of generality because the maximization cases can be converted to minimization by negating the objective function, i.e., by minimizing $f'(\mathbf{x}) = -f(\mathbf{x})$. If the objective is to minimize, f is often called a *cost function*, and a solution $\mathbf{x}^* \in \mathcal{X}$ is an *optimal* one if

$$f(\mathbf{x}^*) \leq f(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}$$

that is, a feasible solution with the least cost.

If $\mathcal{S}$ is a finite or countably infinite set, a *combinatorial optimization problem* is defined. If $\mathcal{S} = \mathbb{R}^n$, we call it a *continuous optimization problem*. In this thesis, we focus on combinatorial optimization problems, though some of the techniques investigated here will also be applicable to continuous optimization.

2.2 Neighborhood-based Heuristics

In the first part of this thesis, we will investigate the task of combining *neighborhood-based heuristics*. This class of search heuristics has a common trait: Given a solution $\mathbf{x}$, the heuristic can modify a portion of $\mathbf{x}$ to produce another solution $\mathbf{x}'$. To represent the idea that a solution $\mathbf{x}'$ can be obtained by changing a portion of a given solution $\mathbf{x}$, we introduce a neighboring relation on the search space. That is, $\mathbf{x}'$ is a neighbor of $\mathbf{x}$ if $\mathbf{x}'$ can be obtained by modifying some parts of $\mathbf{x}$. We use $\mathbf{x} \rightsquigarrow \mathbf{x}'$ to denote that $\mathbf{x}'$ is a neighboring solution of $\mathbf{x}$, and $\mathcal{N}(\mathbf{x})$ to represent the neighborhood of $\mathbf{x}$, i.e., the set $\{\mathbf{x}' | \mathbf{x} \rightsquigarrow \mathbf{x}'\}$. Thus, a mapping $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ encodes the set of modifications under consideration and is called a *neighborhood function* or a *neighborhood structure*. In this way, we can characterize each heuristic by the sets of modifications it considers, and represent this characterization in a unified manner by the notion of neighborhood functions.

In general, we can specify a broad class of heuristics around the concept of neighborhood functions. We call them *neighborhood-based* search heuristics. To differentiate each member of this class, we identify three elements that together can be used to describe a heuristic in this class:

1. the neighborhood structure it considers,
2. the transition rule it uses, and
3. the iterating condition of the process.

To demonstrate this specification, consider a simple optimization procedure that starts from some initial solution $\mathbf{x}$ and evaluates a predefined set of modifications that it can perform on $\mathbf{x}$, which corresponds to a neighborhood structure $\mathcal{N}$. Of all the members of $\mathcal{N}(\mathbf{x})$, it chooses the solution $\mathbf{x}'$ with the best objective value (with the premise that $f(\mathbf{x}') < f(\mathbf{x})$), then switches to consider $\mathbf{x}'$ and the neighbors of $\mathbf{x}'$. This process iterates until there is no further improvement in the objective value.

Here, the neighborhood structure corresponds to the set of predefined modifications that this heuristic considers to perform on a solution. The transition rule is to switch to the best solution in the neighborhood if it is an improvement over the current solution. And the iterating condition is to iterate until no improvement is possible.

Note that changing the employed neighborhood structure will lead to a different heuristic. Thus, we can specify a range of heuristics by supplying different kinds of neighborhood structures.

The above procedure is often called a *best-improvement local search*. A related procedure called *first-improvement* local search can be obtained by changing the transition rule: We can put a priority on the members of $\mathcal{N}(\mathbf{x})$, and examine them sequentially based on this priority. If the neighbor $\mathbf{x}'$ being evaluated is better than $\mathbf{x}$, then we switch to $\mathbf{x}'$ without further considering other solutions in $\mathcal{N}(\mathbf{x})$. This can be used for efficiency purpose, especially when the neighborhood size is huge.

Note that since these two procedures only take transition to a better solution and will iterate until no further improvement is possible, the resulting solution from the process will be a *local optimum* with respect to the employed neighborhood structure $\mathcal{N}$, that is, a solution $\mathbf{x}^+$ with the property that $f(\mathbf{x}^+) \leq f(\mathbf{x}), \forall \mathbf{x} \in \mathcal{N}(\mathbf{x}^+)$.

To have a general sense of the variety of the heuristics in this class of neighborhood-based search heuristics, we provide the following categorization to each of the above three elements:

- The neighborhood structure can be either *static* or *dynamically-defined*.
- The transition rule can be either *deterministic* or *stochastic*.
- The iterating condition can be based on time, e.g., run it for 10 minutes, or based on repetition, e.g., run for a single transition/10 transitions/until no further improvement. . . etc.

Static neighborhood structures correspond to predefined modifications to solutions, and are usually memoryless. On the other hand, dynamically-defined neighborhood structures take historic data about the search process into account. For example, a Tabu Search [38, 39] keeps a record of recently visited solutions and avoids considering solutions similar to those solutions.

A deterministic transition rule specifies transitions without any randomness, and this is the case for most of the local searches. In contrast, a stochastic transition rule will choose a neighbor based on some probability distribution, either specified explicitly or implicitly.

Iterating conditions admit a much looser classification. Here, we distinguish them by whether they are based on time or based on repetition. However, we recognize that for iterating condition, there are other possibilities. For example, it can be that the heuristic will iterate until the improvement is sufficiently small. Another more sophisticated example is using *late acceptance* strategy [12] as a termination condition.

With this categorization, we can further compare and contrast the heuristics in this class. As we will see later, a lot of optimization procedures mentioned in this thesis are (or have components of) this kind of heuristic. For example, a perturbation operation in Iterated Local Search [64] (and see Section 2.4.2) can be described as using a static neighborhood structure, adopting a stochastic transition, and it only takes one transition. A Variable Neighborhood Descent [10, 46] (and see Section 2.4.3) can be described as using a dynamic neighborhood structure, adopting a deterministic transition rule, and iterating until no further improvements.

In summary, in this thesis, we will consider a range of heuristics, each of which can be thought of abstractly as a process that imposes a topology on top of the search space and performs a walk based on this topology. This unified view gives us a foundation to architect mechanisms that can foster and promote collaboration among these heuristics.

2.2.1 Comparison to Branching-based Search Heuristics

As a comparison to the neighborhood-based heuristics, in this section, we briefly describe another category of heuristics called *branching-based* heuristics, which are also popular in AI and OR applications.

With a branching-based heuristic, the search process is structured as a tree. In this tree, a node n is a child of another node m if we can extend the partial solution corresponding to m to get the partial solution corresponding to n, and a terminal node represents a complete solution. This implies that there is some decomposition of the problem so that a complete solution can be viewed as several "parts" assembled together. And each extension to the tree corresponds to adding one more part to a partial solution, resulting in one more node adding to the search tree.

Ideally, we would like to explore only a small portion of such a tree to get a high-quality or even optimal solution. This can be done by making intelligent choices on which branch should we follow. And the branching-based heuristics are techniques intended to manifest such intelligent choices. Popular methods in this category, such as A*, branch-and-bound, and beam search, implement priorities on which branch we should proceed to explore and/or whether we should ignore the exploration of certain branches.

2.2.2 Heuristics in Collaborative Settings

In this section, we discuss some of the advantages of neighborhood-based heuristics in collaborative settings, and we will also make some comparisons to branching-based heuristics to illustrate our perspectives.

First of all, the neighborhood-based paradigm provides a natural and unified interface for heuristics to collaborate with each other. That is, by sending and receiving *complete* solutions. By definition, a neighborhood-based heuristic can pick up a complete solution and modify it to generate a new solution. No complicated or heuristic-specific internal state needs to be sent and parsed by the receiving heuristic. Thus, it is straight-forward to chain the efforts of different heuristics.

In contrast, there is no unified mechanism that allows one branching-based heuristic to pick up the work done by other heuristics. The difficulty comes from the fact that a branching-based heuristic often maintains an internal state that records the progress of the search and uses it to direct future decisions. However, it is often nontrivial how to initialize this state based on the work done by other heuristics. In order to do this, one might have to consider different formulation to the decomposition, different ordering of the decomposition, and different kinds of information evaluated by each heuristic. In general, it is more difficult to devise such a mechanism.

Another advantage of a neighborhood-based framework is that it copes well with different formulation of the same problem. Given that a problem can often be formulated in different ways, it is possible to include switches from one formulation to another in the optimization process. Each formulation should have a set of neighborhood-based heuristics that is "natural" to that formulation, and they can work together the same way since a complete solution found in one formulation can be easily transformed to its equivalent in another formulation.

Of course, such a strategy will make a difference only when heuristics associated with different formulations behave differently. For example, [77] presents a study on the Circle Packing Problem (CPP). It is shown there that a stationary point for a nonlinear programming formulation of CPP in Cartesian coordinates is not necessarily a stationary point in polar coordinates. They used this observation to devise a method that systematically alternates between different formulations. The results were comparable to the best-known values, but it provides a substantial speedup to the alternative single formulation approach. A similar idea was presented in [50] for solving graph coloring problems. They also observed that switching between different formulations[1] has an advantage over single-formulation alternatives.

2.3 Algorithm Portfolios

Another field of research related to this thesis is the study of *algorithm portfolios* [42, 43]. This field also considers the question of how to utilize multiple heuristics[2]. Previous works from this field noted that for most computationally hard problems, there exists multiple solving approaches, and it is typical that none of them completely dominates all others across multiple problem instances. Based on this observation, the goal of algorithm portfolios is to use this kind of performance variation to create an ensemble such that the weakness of one algorithm is covered by the other algorithms. In other words, the idea is to accept that no single algorithm will offer the best performance on all instances and instead, we gather a set of complementary algorithms and devise a strategy for choosing

[1]They referred to different formulations as different "search spaces" in [50].

[2]Note that algorithm portfolio generally does not restrict the component heuristics to be neighborhood-based.

among them.

As noted by Streeter and Smith [97], the problem of designing algorithm portfolios has both a *prediction* aspect[3] and a *scheduling* aspect. It is perhaps most straight-forward to consider only the prediction aspect, in which we use a set of features of the target problem instance to attempt to predict which algorithm will yield the best performance, then simply run that algorithm exclusively. The prediction model is usually learned through some historic data about the past performance of each algorithm. For example, Leyton-Brown et al. [62] use a regression model to estimate how much time each algorithm needs for solving a target problem instance, and then run the algorithm with the shortest estimated duration. This kind of approaches has been successfully applied to satisfiability domains, resulting in systems such as SATzilla [105] and ArgoSmArT [79]. However, its main weakness is that there is no way to mitigate a poor selection, that is, the system cannot recover if the chosen algorithm exhibits poor performance on the given problem instance.

Alternatively, we can also consider the scheduling aspect of the algorithm portfolio: we can construct a schedule that specifies an ordering and time budget according to which we run all or a subset of the algorithms. For stochastic algorithms, we can further consider the questions of whether and when to restart the algorithms. For example, Gomes et al. [44] demonstrated that we can boost the performance of a heuristic solving method by randomizing the method's decision making heuristics and running this randomized version with an appropriate restart schedule.

The earliest works on this scheduling aspect measure the performance of a schedule in terms of its competitive ratio (e.g., the time required to solve a given problem instance using the schedule, divided by the time required by the optimal schedule for that instance). Results include the universal restart schedule of Luby et al. [69] and the schedule of Kao et al. [56] for allocating time among multiple deterministic algorithms subject to memory constraints.

To further utilize the capability of this scheduling aspect, one needs to solve the problem of how to compute schedules that perform well on average over a target distribution of problem instances. In reality, this distribution is usually approximated by a collection of problem instances, treated as training data. Independently, Petrik and Zilberstein [85]

[3]It was called *machine learning* aspect in [97]. However, we think the term "prediction" would be more accurate.

and Sayag et al. [92] addressed this problem for two classes of schedules: task-switching schedules and resource-sharing schedules. For each of these two classes of schedules, the problem of computing an optimal schedule is NP-hard. Streeter et al. [96] presented a polynomial-time 4-approximation algorithm for computing task-switching schedules.

Although research in algorithm portfolio has looked at scenarios comprising multiple heuristics, the setting they consider generally ignores the possibility of allowing component heuristics to interact with each other. That is, they assume that each component heuristic functions independently and there is no information passing or sharing among the component heuristics. In contrast, this thesis has a special focus on the collaboration among heuristics. We would like to look at scenarios in which there will be information passing between heuristics and study how synergy among heuristics can emerge, be extracted, and be utilized to improve the optimization process.

Another minor difference between the work in this thesis and algorithm portfolios is that the research on algorithm portfolio has been mostly focused on solving decision problems, such as satisfiability domains. In such cases, the performance of an algorithm is usually evaluated by the amount of time it needs to find a valid solution. On the other hand, in this thesis, we work primarily on combinatorial optimization problems, for which the natural evaluation criterion is the quality of the resulting solution. Although these two types of problems can usually be transformed from one to another, this transformation can be unnatural or not intuitive, and might not match well with the heuristics under consideration.

2.4 Extensions to Local Search

In Section 2.2, we mentioned a general and intuitive optimization strategy called local search. A local search takes an initial solution $\mathbf{x}$ and descends to a nearby local optimum $\mathbf{x}^+$ according to the provided neighborhood structure $\mathcal{N}$. However, the resulting local optimum is usually not a global minimum, and if we would like to improve upon this, we will need to resort to some other means.

In the following sections, we will review some previous works that are relevant to this proposal. One way to look at them is that these methods were created to address the

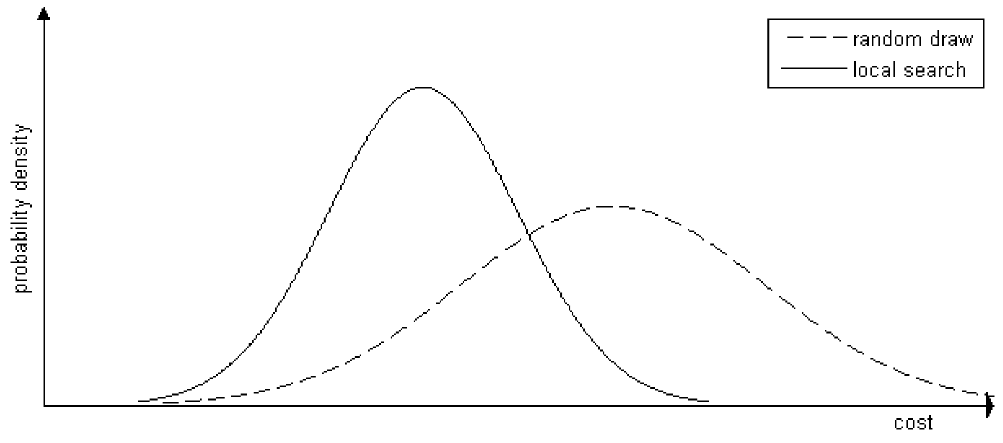


Figure 2.1: The distribution of costs (objective values) of solutions drawn randomly from the search space versus the distribution of costs obtained from running a local search with randomly selected initial solutions.

problem of how to continue the search after descending to (and hence being trapped in) a local optimum.

2.4.1 Random Restart

Perhaps the simplest approach to improve upon a cost found by local search is to run it again with some other starting point. If we draw the initial solutions randomly and independently from the search space, then every local optimum generated as the result of the local search will also be independent. Thus, we can reduce the chance of having only found a poor local optimum by repeating this “sampling” process.

To have a more “visual” understanding of the process, Figure 2.1 illustrates two distributions. Let the dashed line represent the distribution of objective values of solutions drawn randomly from the search space. Now, if we pass each random draw through a local search, the resulting distribution will normally look like the one depicted by the solid line. Note that the average cost shifts to a lower value and the distribution is now more concentrated. So even if we are in a situation where the local search output a poor local optimum, as illustrated in Figure 2.2, the subsequent run of the local search will have a high probability of finding a solution that surpass that local optimum because of this concentration and shifting.

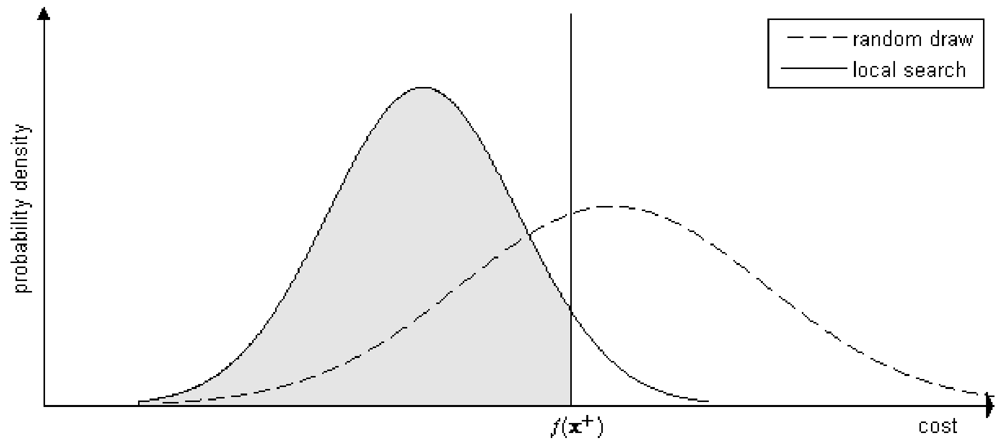


Figure 2.2: The benefit of random restart: if the local search outputs a poor local optimum, illustrated here as $\mathbf{x}^+$ with a cost value $f(\mathbf{x}^+)$, then restarting the local search from another initial point will have a large probability (shaded area) of finding an improvement over $f(\mathbf{x}^+)$.

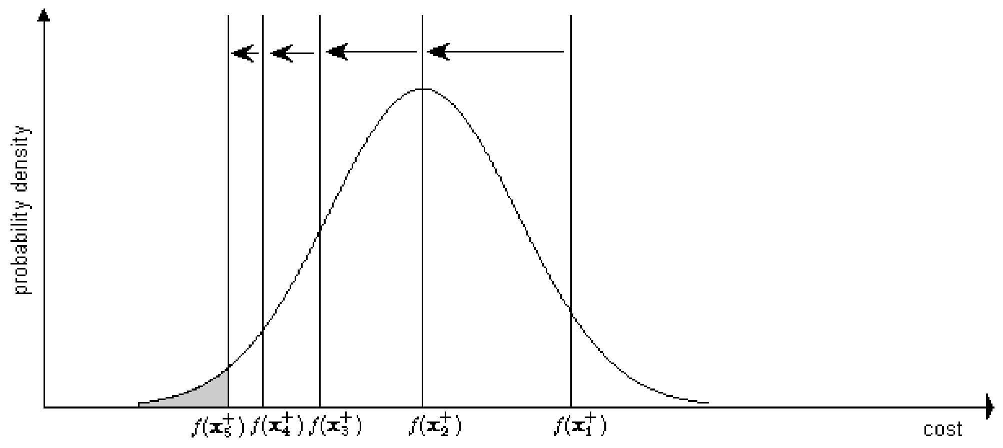


Figure 2.3: An illustration of the progression of repeated random restarts. The probability of finding yet another improvement diminishes rapidly.

As we repeat the process of this *random restart*, we will find increasingly better and better solutions, which corresponds to gradually moving the cost bar to the left, as shown in Figure 2.3. However, with each move, the probability of finding another improvement diminishes accordingly, and the rate of this decrement is usually rather fast. Thus, it may still be difficult for random restarts to find a solution that is significantly better than the mean cost of the distribution.

Furthermore, such a random restart approach might not scale well as the problem size grows. This is because the tail of the cost distribution tends to collapse as we consider larger and larger problem instances, i.e., the distribution becomes more concentrated. For example, [93] shows empirically that local search algorithms on large graph bisection problems lead to cost distributions that (i) have a mean that is a fixed percentage away from the optimum cost; (ii) are increasingly peaked around the mean cost as the size of the problem grows. This will make it very difficult for random restarts to find a solution whose cost is even a little bit percentage-wise lower than the typical cost.

2.4.2 Iterated Local Search

This section provides a short review of Iterated Local Search (ILS) [28, 63, 64]. ILS is a conceptually simple framework that has led to the development of many optimization procedures that exhibit the state-of-the-art performance for many combinatorial optimization problems (e.g. [45], [74], [90], [98].)

An ILS relies on two component heuristics: a *local search* and a *perturbation* heuristic. They are also called the *descent operation* and *perturbation operation* in the ILS framework.

The same as before, the local search of an ILS takes in a solution $\mathbf{x}$ and iteratively moves to a better neighbor according to its neighborhood function $\mathcal{N}_{\mathsf{L}}$. This process typically iterates until no further improvement is possible. On the other hand, a perturbation operation, equipped with its own neighborhood structure $\mathcal{N}_{\mathsf{P}}$, takes as input a solution $\mathbf{x}$ and randomly outputs a solution $\mathbf{x}'$ from $\mathcal{N}_{\mathsf{P}}(\mathbf{x})$. This process usually doesn't involve any checking on the quality of $\mathbf{x}'$ and is performed only once, i.e., no iteration is involved. That is, a perturbation can be seen as an operation that stochastically changes a portion of $\mathbf{x}$.

In short, instead of doing random restarts, an ILS tries to continue the search after descending to a local optimum $\mathbf{x}^+$ by first perturbing $\mathbf{x}^+$ to get a slightly modified solution $\mathbf{y}$, and then performing a local search from $\mathbf{y}$ to get a new local optimum $\mathbf{y}^+$. If the new local optimum $\mathbf{y}^+$ is better than $\mathbf{x}^+$, we switch to $\mathbf{y}^+$ and iterate from there. Otherwise, we return to $\mathbf{x}^+$. This process is listed in Algorithm 1.

To see why we may want to use an ILS instead of a random restart, consider the following assumption,

Common Parts Assumption: It is observed very often in combinatorial optimization

Algorithm 1 Basic Iterated Local Search (Basic ILS)

```
Require: objective function $f$, neighborhood structures $\mathcal{N}_\mathsf{L}$ & $\mathcal{N}_\mathsf{P}$
1: $\mathbf{x} \leftarrow$ initial solution
2: $\mathbf{x}^+ \leftarrow \mathsf{LocalSearch}(\mathbf{x}, \mathcal{N}_\mathsf{L})$
3: repeat
4:     $\mathbf{y} \leftarrow \mathsf{Perturbation}(\mathbf{x}^+, \mathcal{N}_\mathsf{P})$
5:     $\mathbf{y}^+ \leftarrow \mathsf{LocalSearch}(\mathbf{y}, \mathcal{N}_\mathsf{L})$
6:     $\mathbf{x}^+ \leftarrow \mathbf{y}^+$ if $f(\mathbf{y}^+) < f(\mathbf{x}^+)$
7: until termination criteria are met
```

that, for a locally optimal solution $\mathbf{x}^+$, there exists a better solution that shares common parts with $\mathbf{x}^+$.

This commonality may appear in the form that both solutions have the same assignment to a subset of variables, or it can be in the form that both solutions exhibit some similar structures. For example, in a scheduling problem, it can be that both solutions have the same precedence ordering over some subset of jobs.

This implies that a local optimum has the potential to provide components that can be used to construct an even better solution. However, it is usually unknown in advance which components are such. Since these components cannot be discerned easily or automatically, some trial-and-error or exploratory process is needed. The way ILS approaches this problem is by performing a perturbation followed by a local search. Conceptually, we may think of a perturbation as an operation that aims to change a portion of the solution which is not part of these common structures. As a follow-up, the local search tries to improve upon the perturbed solution and/or revert some part of the perturbation that did more harm than good.

Another view on the ILS can be described by the cost distributions. Let the dashed line in Figure 2.4 represent the cost distribution of solutions obtained from a local search with starting points drawn randomly from the whole search space. Assume that the current best solution is a local optimum $\mathbf{x}^+$, then ideally the cost distribution resulted from a perturbation on $\mathbf{x}^+$ followed by the local search will be of the form depicted by the solid line. Note that this second cost distribution has probability mass that is more concentrated around the cost of $\mathbf{x}^+$. This translates to a higher probability of finding an improvement (illustrated by the larger shaded area to the left of $f(\mathbf{x}^+)$ under the solid line) compared to

Algorithm 2 General Iterated Local Search (General ILS)

```
Require: objective function $f$, neighborhood structures $\mathcal{N}_\mathsf{L}$ & $\mathcal{N}_\mathsf{P}$
1: $\mathbf{x} \leftarrow$ initial solution
2: $\mathbf{x}^+ \leftarrow \mathsf{LocalSearch}(\mathbf{x}, \mathcal{N}_\mathsf{L})$
3: repeat
4:     $\mathbf{y} \leftarrow \mathsf{Perturbation}(\mathbf{x}^+, \mathcal{N}_\mathsf{P})$
5:     $\mathbf{y}^+ \leftarrow \mathsf{LocalSearch}(\mathbf{y}, \mathcal{N}_\mathsf{L})$
6:     $\mathbf{x}^+ \leftarrow \mathsf{AcceptanceCriterion}(\mathbf{x}^+, \mathbf{y}^+)$
7: until termination criteria are met
```

using a random restart (area to the left of $f(\mathbf{x}^+)$ under the dashed line.) This can explain why a well-designed ILS may be less likely to stagnate than a random restart approach especially when the search moves away from the mean cost of the local search distribution.

Note that the above describes the "ideal" situation for an ILS. In practice, depending on the perturbation used, the probability mass may not distribute equally on both sides of $f(\mathbf{x}^+)$ or it can even degenerate to a single spike at at the location of $f(\mathbf{x}^+)$ (this happens in the case in which no matter what modification the perturbation has done to $\mathbf{x}^+$, the following local search will systematically undo it.) In general, this view also relies the assumption that small changes on the solution tends to correspond to small changes on the objective value. Otherwise, there is no reason for the probability mass to be concentrated around $f(\mathbf{x}^+)$.

The above basic ILS can be further generalized. In practice, the conditional in line 6 of Algorithm 1 is often replaced by some other criterion for the purpose of greater performance. This modification usually aims to explore a broader range of the search space by accepting a $\mathbf{y}^+$ that is not necessarily better than $\mathbf{x}^+$. For example, acceptance criteria similar to Simulated Annealing [32, 59] can be used. We can abstract this extension into a function called AcceptanceCriterion. This generalized ILS is shown as Algorithm 2.

The performance of an ILS is crucially dependent on the interaction between its three components: LocalSearch $(\,\cdot\,, \mathcal{N}_\mathsf{L})$, Perturbation$(\,\cdot\,, \mathcal{N}_\mathsf{P})$, and AcceptanceCriterion$(\,\cdot\,,\,\cdot\,)$. For example, Lourenço and Zwijnenburg [65, 66] used ILS to tackle the job shop scheduling problem under the makespan criterion. They performed extensive experiments comparing different methods for generating initial solutions, various local searches, different perturbation heuristics and three acceptance criteria. They found that while the initial solution had only a limited influence, the other components turned out to be very important, and

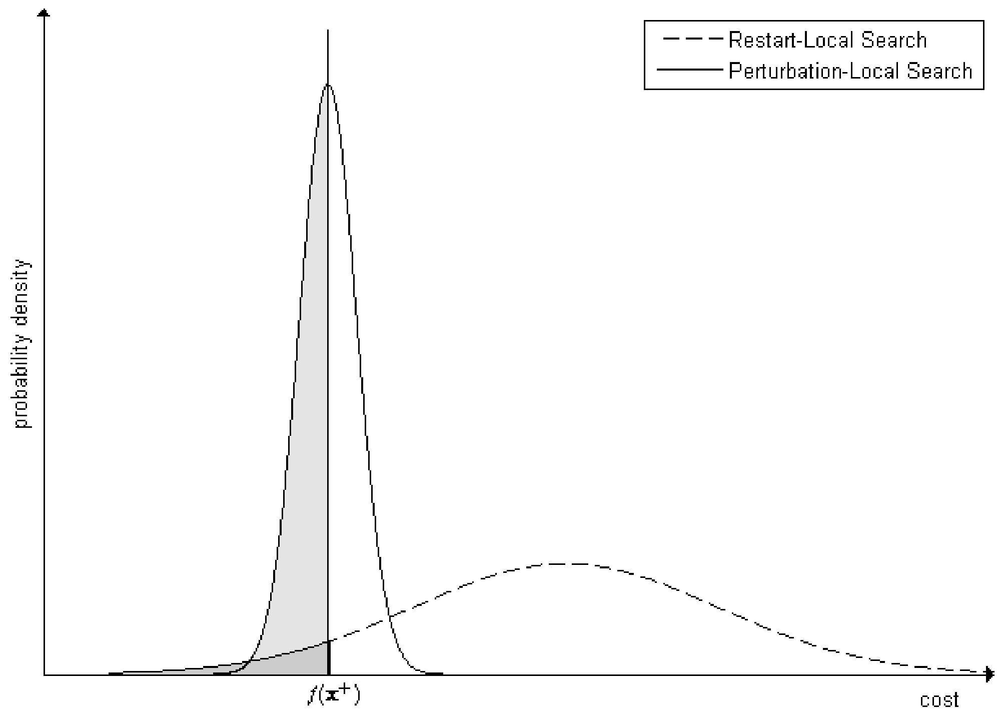


Figure 2.4: Randomly restarting a local search vs. performing a perturbation followed by a local search. The dashed line depicts the cost distribution of solutions obtained from a local search with initial points drawn randomly from the search space. The solid line corresponds to an ideal cost distribution of solutions obtained by first perturbing the current solution $\mathbf{x}^+$ and then applying a local search on the perturbed version. Ideally, with small perturbation, this distribution spreads narrowly around $f(\mathbf{x}^+)$. Shaded areas represent the probability of finding an improving solution over the current solution $\mathbf{x}^+$, i.e. a solution with a lower cost than $f(\mathbf{x}^+)$.

better results can be obtained by tuning the combination.

In general, in order for an ILS to efficiently explore alternative candidate solutions, the perturbation should offer changes that cannot be systematically undone by the following local search. Otherwise, it will fall back to the previous local optimum just visited. However, if the perturbation is too aggressive, the ILS will behave more like a random restart, and reduce the probability of finding a better solution. So the type and degree of the modification performed by the $\mathsf{Perturbation}(\,\cdot\,, \mathcal{N}_{\mathsf{P}})$ is considered very important for constructing an effective ILS.

Furthermore, choosing an appropriate perturbation heuristic is not an isolated decision.

An intelligent choice of $\mathsf{Perturbation}(\cdot, \mathcal{N}_\mathsf{P})$ will depend on the choice of the accompanying $\mathsf{LocalSearch}\ (\cdot, \mathcal{N}_\mathsf{L})$. And similarly, the choice of $\mathsf{AcceptanceCriterion}(\cdot, \cdot)$ will depend on both $\mathsf{Perturbation}(\cdot, \mathcal{N}_\mathsf{P})$ and $\mathsf{LocalSearch}\ (\cdot, \mathcal{N}_\mathsf{L})$. In [64], Lourenço et al. offered a loose guideline for such decisions:

- The perturbation should not be easily undone by the local search; if the local search has obvious shortcomings, a good perturbation should compensate for them.
- Large perturbations are only useful if the resulting new solutions can be accepted, which occurs only if the acceptance criterion is not too biased toward better solutions.

Given that there exists such relations between these algorithmic components, and for each component, there are usually quite a few options, a practitioner can spend a considerable amount of effort on finding a combination in which the chosen options interact effectively and generate positive behavior for the search. In this thesis, we consider automated procedures that can potentially reduce this inconvenience.

We are interested in developing automated methods that can extract patterns of beneficial interactions between heuristics. Moreover, our interest is not limited to finding just *one* such useful configuration of interaction, as in the case of ILS, but we would like to consider a set of potentially useful interaction patterns and mechanisms which utilize them.

Furthermore, we think it will be potentially helpful to consider other types of combination besides the perturbation—local search, as in the case of ILS. This can also be generalized to a combination of more than two heuristics, as we shall see in the later sections.

2.4.3 Variable Neighborhood Search

Variable Neighborhood Search (VNS) [47, 48] refers to a collection of methods that share a common paradigm of systematically switching between multiple neighborhood structures during the search. This section briefly reviews two such procedures.

Basic VNS [76] has an algorithmic structure similar to an ILS: it also alternates between $\mathsf{Perturbation}$ and $\mathsf{LocalSearch}$. In some sense, it is a generalization of the Basic ILS scheme. With Basic VNS, we augment the perturbation step so that it will now switch between different neighborhood structures in a predefined order.

This augmentation is created in attempt to address the situation in which a local opti-

Algorithm 3 Basic Variable Neighborhood Search (Basic VNS)

```
Require: objective function $f$, a neighborhood structure $\mathcal{N}_{\mathsf{L}}$ for local search, and a set of
    neighborhood structures $\{\mathcal{N}_{\mathsf{P}}^k\}_{k=1\ldots K}$ for perturbation.
 1: $\mathbf{x} \leftarrow$ initial solution
 2: $\mathbf{x}^+ \leftarrow \mathsf{LocalSearch}(\mathbf{x}, \mathcal{N}_{\mathsf{L}})$
 3: $k \leftarrow 1$
 4: repeat
 5:     $\mathbf{y} \leftarrow \mathsf{Perturbation}(\mathbf{x}^+, \mathcal{N}_{\mathsf{P}}^k)$
 6:     $\mathbf{y}^+ \leftarrow \mathsf{LocalSearch}(\mathbf{y}, \mathcal{N}_{\mathsf{L}})$
 7:     if $f(\mathbf{y}^+) < f(\mathbf{x}^+)$ then
 8:         $\mathbf{x}^+ \leftarrow \mathbf{y}^+$
 9:         $k \leftarrow 1$
10:     else
11:         $k \leftarrow k + 1$
12:         if $k > K$ then $k \leftarrow 1$ end if
13:     end if
14: until termination criteria are met
```

mum might *trap* a certain kind of perturbation operation, i.e., the perturbed solutions will constantly fall back to the originating local optimum after the ensuing descent operation. The principle behind the Basic VNS is that a local optimum may trap certain kinds of perturbation operations but probably not all kinds of perturbations. Thus, by systematically switching from one neighborhood structure to another, we decrease the risk of being trapped permanently. The Basic VNS is shown in Algorithm 3.

The idea of adopting multiple neighborhood structures can also be used in constructing descent operations. The Variable Neighborhood Descent (VND) [10, 46] presents one possibility to such an idea. Similar to the Basic VNS, a VND switches from one neighborhood structure to another in a predefined order. If the local search operating on the kth neighborhood structure does not find a better solution, it will advance to the $(k + 1)$-th neighborhood structure. On the other hand, once it has found an improved solution, it switches back to the first neighborhood structure. This process is illustrated in Algorithm 4.

The concept of the VND is based on the following two facts:

1. A local optimum with respect to one neighborhood structure is not necessarily a local optimum with respect to another neighborhood structure.
2. A global optimum is a local optimum with respect to all possible neighborhood structures.

Note that a VND will stop at a solution that is a local optimum with respect to all K neighborhood structures that we build into it, and hence we will be more likely to get a global optimum than using just a single neighborhood structure.

Algorithm 4 VND

```
Require: objective function $f$, and a set of neighborhood structures $\{\mathcal{N}_{\mathsf{L}}^{k}\}_{k=1\dots K}$.
1: $\mathbf{x} \leftarrow$ initial solution
2: while $k \leq K$ do
3:     $\mathbf{x}^{+} \leftarrow \mathsf{LocalSearch}(\mathbf{x}, \mathcal{N}_{\mathsf{L}}^{k})$
4:     if $f(\mathbf{x}^{+}) < f(\mathbf{x})$ then
5:         $\mathbf{x} \leftarrow \mathbf{x}^{+}$
6:         $k \leftarrow 1$
7:     else
8:         $k \leftarrow k + 1$
9:     end if
10: end while
```

One view on the VND is that it can be regarded as an *aggregated local search.* This view can be used to extend the Basic VNS. We can now replace the LocalSearch on line 6 of Algorithm 3 by a VND, taking a separate set of neighborhood structures $\{\mathcal{N}_{\mathsf{L}}^{j}\}_{j=1\dots J}$. This *General VNS* framework has led to some of the most successful applications in the VNS literature [2, 10, 18, 19, 46, 49, 88].

Compared to ILS (and also to the vanilla local search), VNS provides an extended idea that we can perform systematic switches among a set of predefined neighborhood structures. This allows more heuristics, defined as perturbations and local searches operating on different neighborhood structures, to interact within the search process. On one hand, these interactions can take the form of Perturbation—LocalSearch similar to what we discussed before on the ILS. On the other hand, it can happen during the iterations within a VND in which ideally, the employed neighborhood structures present complementary coverage of to different descent possibilities.

However, similar to the problem of finding an appropriate pair of perturbation and local search for an ILS, the construction of a VNS requires decisions on what neighborhood structures to be included in the algorithm. It can still be a considerable amount of effort to decide how to compose the set of neighborhood structures $\{\mathcal{N}_{\mathsf{P}}^{k}\}$ and/or $\{\mathcal{N}_{\mathsf{L}}^{j}\}$ for a VNS if done manually.

Furthermore, as we can see in the Basic VNS, the algorithm applies the set of perturbation heuristics in a round-robin fashion. However, if some perturbation heuristics are on average more effective than others, would it still be ideal to apply them with roughly equal opportunity? It is obvious that for most of the cases, maintaining a diverse set of heuristics is still preferable. But perhaps we can make the mechanism more flexible than a round robin. Later in this thesis, we will see that we can approach this problem by putting a distribution on the set of heuristics that we consider to apply, and choose which heuristic to apply next by sampling this distribution. Thus instead of a rigid round robin, we now have a mechanism that can encode our belief of how effective a heuristic (or a cooperation pattern of heuristics) might be for the current situation.

Also note that although VNS considers a wider range of interactions between component heuristics, these interactions still follow two fixed patterns:

1. the perturbation—local search interaction, and
2. the complementary coverages from different local search heuristics.

For this thesis, we would like to develop algorithms that generalize beyond this. We are particularly interested in mechanisms that allow useful interactions to emerge flexibly, and methods for mining and extracting the patterns of these interactions so that we can later use them for improving the optimization procedure.

2.5 Sampling-based Heuristics

In the second part of this thesis, we will switch to investigating the topic of combining multiple *sampling-based heuristics.* With this class of heuristics, the process of constructing a solution is based on sampling some explicit probabilistic model rather than imposing perturbations on an existing solution like the case of neighborhood-based heuristics. Such probabilistic models can be hand-crafted or estimated from some data source.

Hand-crafted probabilistic models have been used in previous search frameworks such as Bresina's HBSS [9] and Cicirello and Smith's VBSS [25]. In general, these frameworks use probabilistic models that are backed by some heuristic function. Such a heuristic function assigns a value to each possible candidate (partial) solution, which is then used to calculate the probability mass to be placed on selecting a particular (partial) solution.

The algorithm then iteratively samples the probabibilistic model to compose a solution.

As an example, in Chapter 6, we will develop our discussion using the Traveling Salesman Problem (TSP) as the demonstrative combinatorial optimization problem. A handcrafted probabilistic model for forming a solution to a TSP can be

$$P(w_i|w_{i-1}) = \frac{d(w_{i-1}, w_i)^{-10}}{\sum_{v \in \mathbf{V}} d(w_{i-1}, v)^{-10}}$$

where w_i is a possible city to be placed as the next city in the current partial tour, w_{i-1} is the city that we picked in the previous step, $d(u, v)$ is the distance between city u and city v, and $\mathbf{V}$ is the set of remaining cities to be picked. In this place, we use the distance as our heuristic function and create a probabilistic model accordingly. This basically expresses the simple heuristic that the shorter the link between two cities, the more likely that link will be adopted in the solution.

In addition to handcrafting probabilistic models, another approach is to use models estimated from some data source. This data source usually contains a group of good solutions so that the resulting model can potentially learn or encode their particular features. And sampling the learned model corresponds to a recombination of those features found in different good solutions. A representative framework of this category is that of Estimation of Distribution Algorithms, which we will briefly review in the next section.

2.6 Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) are a class of population-based stochastic search techniques that search the solution space by learning and sampling probabilistic models [60, 68, 83, 84]. To find an optimal solution, an EDA iteratively updates a population of candidate solutions and adjusts its search direction based on promising solutions found along the way. To perform the update, an EDA builds a probabilistic model based on the promising solutions in the current population and then samples the constructed model to generate new candidate solutions. These new solutions replace some existing solutions in the current population according to some replacement strategy, which aims to increase the average solution quality of the population.

The general procedure of an EDA is outlined in Algorithm 5. The initial population is

Algorithm 5 General Procedure of an EDA

```
Initialize a population $\mathbf{P}$ with a set of solutions.
Evaluate the solutions in $\mathbf{P}$.
$t \leftarrow 1$.
while the stopping criterion is not met do
    $M_t \leftarrow$ build a probabilistic model based on promising solutions in $\mathbf{P}$.
    $\mathbf{S}_t \leftarrow$ sample $M_t$ to generate new candidate solutions
    Evaluate the solutions in $\mathbf{S}_t$.
    Incorporate $\mathbf{S}_t$ into $\mathbf{P}$ using some replacement strategy.
    $t \leftarrow t + 1$.
end while
```

usually formed by sampling a uniform distribution over admissible solutions, but it could also be generated by using a distribution biased by some prior knowledge. The algorithm then iterates through steps of model building, sampling, evaluating and replacing solutions until some termination criterion is met. This could be, for example, when a solution of satisfactory quality has been found or when the algorithm has visited a maximum allowable number of solutions.

Historically, EDAs have grown out of the work on Evolutionary Algorithms [33], and are especially connected to Genetic Algorithms (GAs) [41, 52]. One may think of an EDA as a GA with traditional variation operators, such as mutation and crossover, being replaced by the process of building and sampling a probabilistic model. This replacement enables an EDA to use techniques from machine learning and statistics to automatically discover patterns exhibited in promising solutions. This information is summarized in the resulting model. Through sampling the learned model, these patterns can be effectively recombined to produce possibly better candidate solutions. Compared to information-blind crossover operators, this process is more likely to reproduce promising partial solutions and reduce the chance of disrupting good patterns.

In past studies, EDAs have been applied to a wide variety of academic and real-world optimization problems [60, 84], achieving competitive results in many scenarios: bioinformatics [3], chemical applications [73, 91], power systems [22, 57], and environmental resources [31, 82], to name a few. Most of these studies focus on domains in which a solution can be naturally represented as a fixed-length string with no ordering dependencies. However, many combinatorial optimization problems do not fit naturally into this kind of encoding. In the later chapter on combining multiple sampling-based heuristics, we will develop our method using sequencing and routing problems as the demonstrative problem domain. For this kind of problem, a solution can be more naturally represented as an ordering of items.

2.7 Summary

In this chapter, we have given introduction to the types of heuristics that we will use as elementary components in the studies described in the following chapters. We also reviewed some research areas related to this thesis and gave comparisons between them and our intended studies. In the following chapter, we begin with a basic architecture that serves as the foundation of our study on neighborhood-based heuristics. Then in Chapter 6, we will switch to the topic of combining multiple sampling-based heuristics.

Chapter 3

An Architecture for Combining Neighborhood-based Heuristics

In this chapter, we present a basic architecture that will serve as the underlying foundation for the first part of this thesis. It functions by passing the work of one heuristic to another in the form of a complete solution. This complete solution will be used by the receiving heuristic for initializing its own operation. As mentioned in the previous chapter, the class of heuristics that we consider in the first part of this thesis are neighborhood-based heuristics. One characteristic of the neighborhood-based heuristics is that they can pick up a complete solution and modify it to generate a new solution. Thus, we can design a high-level flow that chains the effort of multiple heuristics in a pipelined fashion. In this way, we can create an environment that allows heuristics to collaborate with each other.

Another view on this idea is that we are essentially structuring the exploration of the search space as a process that constructs many *solution chains*. Each solution chain is formed by successively applying one of a set of heuristics (chosen by certain policy) to the current solution to generate the next solution. The algorithmic description of this idea is shown in Algorithm 6. In this algorithm, we assume that we are given a set of heuristics H, a policy $\mathcal{L}$ for choosing the chain length, and a policy $\mathcal{H}$ for choosing among the given heuristics. Every iteration starts with picking a number ℓ based on the policy $\mathcal{L}$. This ℓ will bound the length of the next solution chain. The algorithm then goes on constructing a chain of solutions by applying a sequence of heuristics (selected according to policy $\mathcal{H}$) successively, as illustrated in Figure 3.1. If any solution encountered during this process is

Algorithm 6 A Basic Architecture for Heuristic Collaboration

```
Require: a set of heuristics $H$, a policy $\mathcal{L}$ for choosing lengths, and a policy $\mathcal{H}$ for choosing
    heuristics.
 1: $\mathbf{x} \leftarrow$ initial solution.
 2: while stopping criteria not met do
 3:     $\ell \leftarrow$ a length chosen according to $\mathcal{L}$
 4:     $\mathbf{x}_0 \leftarrow \mathbf{x}$
 5:     for $i = 0$ to $(\ell - 1)$ do
 6:         $h_{i+1} \leftarrow$ a heuristic chosen from $H$ according to $\mathcal{H}$.
 7:         $\mathbf{x}_{i+1} \leftarrow h_{i+1}(\mathbf{x}_i)$
 8:         if $\mathbf{x}_{i+1}$ is better than $\mathbf{x}$ then
 9:             break
10:         end if
11:     end for
12:     if the best among $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\ell$ is better than $\mathbf{x}$ then
13:         $\mathbf{x} \leftarrow$ the best among $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\ell$
14:     end if
15: end while
```

better than the incumbent solution $\mathbf{x}$, we break out of the inner loop and replace $\mathbf{x}$ with the better solution. Otherwise, this process repeats. Note that since in most cases, we have stochastic heuristics included in H, we can also think of this process as *sampling* a solution chain.

For this research, we mainly consider the situation where the set of heuristics H is predefined and is given to us from some external source. Thus, we can think of a *problem domain* as a combination of an optimization problem and a set of heuristics designed for that problem. Apparently, there are still two elements that we need to supply to make Algorithm 6 functional: the policy $\mathcal{L}$ for choosing the chain lengths and the policy $\mathcal{H}$ for choosing the heuristics. For this preliminary study, we will mainly focus on the policy $\mathcal{L}$. Specifically, we will center our discussion on a setting that makes two agnostic assumptions:

1. We assume that we do not have detailed knowledge about the problem domain being solved, except that we have access to the objective function (treating it as a black-box) and a set of predefined heuristics.

2. We assume that we have no information about the amount of time allocated for running our algorithm.

We develop our discussion based on these two assumptions and will come back to the

$$\begin{tikzcd}
\mathbf{x}_0 \arrow[r, bend left, "h_1"] & \mathbf{x}_1 \arrow[r, bend left, "h_2"] & \mathbf{x}_2 \arrow[r, bend left, "h_3"] & \cdots\cdots \arrow[r, bend left, "h_\ell"] & \mathbf{x}_\ell \\
& & & \boxed{\begin{array}{c}\text{Check if } \mathbf{x}_i \text{ is better than} \\ \text{incumbent solution } \mathbf{x}. \\ \text{If it is, we terminate this chain at } \mathbf{x}_i.\end{array}} \arrow[ull, dotted] \arrow[ul, dotted] \arrow[u, dotted] \arrow[ur, dotted] &
\end{tikzcd}$$

Figure 3.1: Sampling a solution chain. This process starts with a solution $\mathbf{x}_0$. Each following step consists of selecting a heuristic h_{i+1} from the set of provided heuristics H and applying h_{i+1} to the previous solution $\mathbf{x}_i$ to get a new solution $\mathbf{x}_{i+1}$. If $\mathbf{x}_{i+1}$ is better than the incumbent solution $\mathbf{x}$, we terminate this process and replace $\mathbf{x}$ with $\mathbf{x}_{i+1}$. Otherwise, we proceed until reaching the bound ℓ. Since in most cases, we have stochastic heuristics included in H, and hence the $\mathbf{x}_i$'s are generated stochastically, we can think of this process as *sampling* a solution chain.

issue of designing $\mathcal{H}$ in the following chapters. For simplicity, the following discussion will use a policy $\mathcal{H}_u$ that makes a uniformly random choice among the provided heuristics H each time it is consulted. We should emphasize that this particular policy is for illustrative purpose. The techniques presented in the following generalizes to arbitrary $\mathcal{H}$. And in the next chapter, we will start exploring mechanisms that automatically derive $\mathcal{H}$ policies from data.

Conceptually, we can see that this architecture is an algorithmic template that we can plug different policies into to make it behave differently. It also relates to a growing research field called *hyper-heuristics*. In the following section, we will start by briefly reviewing the field of hyper-heuristics and connecting our algorithm to this field.

3.1 Hyper-heuristics

The term "hyper-heuristics" [14, 16, 21] refers to a set of methods that aim at achieving the following objective: given a set of heuristics (or building blocks that can be used to construct heuristics), automatically produce an adequate combination that handles the given problem. This framework has a two-layer structure: At the lower level, there is a set of heuristics or building blocks for constructing heuristics, and the top level corresponds to a mechanism that utilizes the lower-level components in combination to solve a given problem. The term “hyper-heuristics” was first introduced by Cowling et al. [26] as “heuristics to choose heuristics.” Its definition has since been extended. Broadly speaking, hyper-heuristic approaches can be roughly categorized into two classes: methods that harness a pre-existing set of heuristics, and methods that generate new heuristics. In the first category, the hyper-heuristics are equipped with a set of predefined low-level heuristics, and the task is to decide which low-level heuristic to apply at a given point during the optimization. The second category corresponds to methods that build new heuristics from a set of components, usually via genetic programming (e.g. [11], [51], [58].) For our purposes here, we focus on the connection to the first category.

Most of the methods from the first category follow an iterative procedure: Given an initial solution (either generated randomly or heuristically), the hyper-heuristic loops through the steps of (i) selecting a heuristic from the set of provided heuristics (usually neighborhood-based heuristics), then (ii) applying the selected heuristic to the incumbent solution to generate a new solution, and finally (iii) deciding whether the new solution should be accepted as the new incumbent solution. This process iterates until the termination criteria are met.

Another dimension for classifying hyper-heuristics is based on the learning mechanisms they employ. A common classification is to categorize them as online learning, offline learning or no learning at all. An online learning hyper-heuristic adjusts itself based on the feedback received during the search process and dynamically biases the selection probabilities (e.g. [75], [72], [61]). Offline learning, in contrast, takes place before the actual search starts (e.g. [71], [4]). While most of the recent hyper-heuristics employ some forms of learning, in this chapter, we will focus on the no-learning cases because it simplifies our discussion. Furthermore, as we will see in Section 3.4, a no-learning algorithm can still be pretty competitive when comparing to more elaborated, learning-based hyper-heuristics.

Hyper-heuristics without a learning mechanism were among the first hyper-heuristics to appear. In their initial paper, Cowling et al. [26] proposed a no-learning hyper-heuristic called Simple Random which chooses a low-level heuristic uniformly randomly at each step. The authors experimented with two acceptance strategies: All Moves (AM) and Only Improving (OI). AM accepts new solutions regardless of their quality, while OI accepts only better solutions, and if the new solution is inferior than the incumbent, it is simply discarded.

Now consider the algorithm that was previously presented as Algorithm 6. If we use a simple policy $\mathcal{H}_u$ that selects a heuristic uniformly at random each time it is consulted, then it can be seen as an extension to the Simple Random with OI. And instead of a single-step heuristic application, in Algorithm 6 we consider the accumulated effect of applying multiple heuristics. More specifically, we consider the exploration of solutions within a chain of limited length. As described before, this solution chain is formed by successively applying a randomly chosen heuristic to the previous solution to generate the next solution. For this setup, the only unspecified element of the algorithm is how to make choices on the bounding length ℓ, and the algorithm can be viewed as a procedure which at each iteration of the inner loop, samples a solution that is at most ℓ operations (i.e. heuristic applications) away from the incumbent solution $\mathbf{x}$. In the following, we will often use the word "operation" to refer to the action of applying a heuristic.

In this chapter, we discuss policies for choosing ℓ. One can imagine that this choice can have a significant impact on the algorithm. For example, if the nearest improving solution is k operations away, then choosing an $\ell < k$ will not yield any improvement. On the other hand, if we choose an ℓ that is much larger than needed, we can waste a significant amount of time exploring unpromising regions. Furthermore, proper choice of ℓ may vary during the optimization process, possibly making rigid policies (e.g. always choose $\ell = 10$) inefficient or ineffective.

In the following section, we first define the scenario that we consider, with its two agnostic assumptions. Based on that, Section 3.3 presents a strategy for choosing ℓ which has an asymptotic guarantee. Following that, Section 3.4 shows the result of our preliminary experiments. We offer some further discussion of the results in Section 3.5. And finally, Section 3.6 summarizes this chapter.

3.2 An Agnostic Setting

To make our results as general as possible, we impose two kinds of agnostic requirements to our discussions. This setup will allow us to derive a widely applicable policy for the

$\mathcal{L}$ component, which we will then use as a foundation for further developing learning mechanisms on the $\mathcal{H}$ component.

The first agnostic requirement concerns the amount of information that we have on the problem domain. As mentioned before, we define the concept of a problem domain as a combination of an optimization problem and a set of heuristics designed for that problem. To promote generality, we will treat both of these as black-boxes. This setting has been studied in previous hyper-heuristic research, often under the name of *cross-domain optimization* (e.g. [4], [53], [15], [13].) The main goal of this line of research is to design optimization methods that can operate on different problem instances and domains. The two-layer structure of hyper-heuristics provides a logical separation between lower-level domain-specific heuristics and a higher-level control policy, which makes it possible to change the domain-specific components while retaining the higher-level policy. In this way, a hyper-heuristic has the potential to be widely-applicable.

The second agnostic assumption that we make reflects the amount of computational resource that we have for solving a given problem. Most previous research has assumed that we are given a fixed amount of computing power for solving the given problem, and this amount is known beforehand (e.g. the algorithm implementation will be run for 10 minutes on some specific machine.) However, we argue that this assumption essentially corresponds to a preference for methods that are tuned to a specific amount of computing power. It may very well be the case that a method that performs well with a particular amount of computational resource will compare unfavorably if the resource is doubled or halved. For example, [87] compared several hyper-heuristics and noted that the best performing hyper-heuristics depended on the allotted CPU time. Furthermore, this tuning can be specific to a particular problem domain, and might not be ideal if we change to a different problem domain. In the following theoretical discussion on $\mathcal{L}$, we consider the situation in which we do not know beforehand the amount of time allocated for running our algorithm. The algorithm has to function properly under the assumption that it does not know when will it be terminated externally, hopefully yielding a satisfactory result upon termination. In short, we would like the algorithm to have a good *anytime profile*[1].

Consequently, under this condition, we would like our algorithm to find an improving

[1]This can be of important concern when the search and execution are tightly coupled (e.g. dynamic vehicle routing problems.)

Algorithm 7 A Modified Version of Algorithm 6

```
Require: a policy $\mathcal{L}$ for choosing lengths
 1: $\mathbf{x} \leftarrow$ initial solution
 2: while stopping criteria not met do
 3:     $\ell \leftarrow$ a length chosen according to $\mathcal{L}$
 4:     if ($\mathbf{x}' \leftarrow$ FindImprovement($\mathbf{x}, \ell$)) $\neq$ null then
 5:         Replace $\mathbf{x}$ with $\mathbf{x}'$
 6:     end if
 7: end while
```

Procedure 8 Search by Sampling a Solution Chain

```
Require: a set of heuristics $H$ and policy $\mathcal{H}$ for selecting heuristics.
 1: procedure FindImprovement($\mathbf{x}, \ell$)
 2:     $\mathbf{x}_0 \leftarrow \mathbf{x}$
 3:     for $i = 0$ to $\ell - 1$ do
 4:         $h_{i+1} \leftarrow$ a heuristic chosen from $H$ according to $\mathcal{H}$
 5:         $\mathbf{x}_{i+1} \leftarrow h_{i+1}(\mathbf{x}_i)$
 6:         if $\mathbf{x}_{i+1}$ is better than $\mathbf{x}$ then
 7:             return $\mathbf{x}_{i+1}$
 8:         end if
 9:     end for
10:     return null
11: end procedure
```

solution *as soon and as often as possible.* If we look at Algorithm 6 from this perspective, the objective will be to find the next improvement using as few operations (heuristic applications) as possible, assuming each operation takes an equal amount of time[2]. However, the situation is actually more involved. It relates to the probability that the inner loop terminates with an improving solution, and also relates to the length of each inner loop execution. This can be better understood by folding line 4 through 11 of Algorithm 6 into a procedure and rewriting it as Algorithm 7. Ideally, we would like a call to FindImprovement to have a high probability of returning a better solution (equivalently, having a non-null return.) One way to increase this probability is to use a larger ℓ. However, a larger ℓ also corresponds to a higher expected number of operations per call to FindImprovement and on average, will take more time.

[2]Note that in reality, different heuristics take different amounts of time, but because they are selected randomly and independently, we can take the expectation on the execution time and make this simplification.

Also note that this probability depends on the incumbent solution $\mathbf{x}$. For example, for a local optimum $\mathbf{x}$, because there is no immediate neighbor that is better in quality[3], using $\ell = 1$ will have a zero probability of finding an improvement, while for other $\mathbf{x}$, there will be a non-zero probability for $\ell = 1$. Furthermore, every time we change to a new incumbent solution, this probability is likely also going to change, making techniques such as multi-armed bandit procedures unsuitable, since they assumes a fixed reward distribution[4].

3.3 Choosing Sampling Lengths

In this section, we discuss the policy for choosing the sampling lengths. As mentioned in the previous section, we would like to have a policy $\mathcal{L}$ so that we can find an improving solution as fast as possible. For this scenario, we can describe the task as follows: For each new incumbent solution $\mathbf{x}$, the policy $\mathcal{L}$ decides on an infinite sequence $\mathcal{S} = (\ell_1, \ell_2, \ell_3, \ldots)$ which represents the sampling lengths for the subsequent iterations. And we call FindImprovment according to this sequence until it yields an improving solution. This process is summarized as Algorithm 9.

Note that we can think of each of these infinite sequences as corresponding to a different *strategy* for exploring the search space. In the following, we will often use the term "strategy" to refer to such an infinite sequence, i.e. a sequence $(\ell_1, \ell_2, \ell_3, \ldots)$ where each $\ell_j \in \mathbb{Z}^+ \cup \{\infty\}$.

Now let $Y_{\mathcal{S}}(\mathbf{x})$ be the random variable representing the number of operations accumulated from successive calls to FindImprovement under strategy $\mathcal{S}$ until it returns an improvement over $\mathbf{x}$. Our objective can be stated as to minimize the expected value of $Y_{\mathcal{S}}(\mathbf{x})$ by choosing a good strategy $\mathcal{S}$. In the following, we will first investigate the situation in which for any ℓ, we have the knowledge of how probable it is that a call to FindImprovement$(\mathbf{x}, \ell)$ will return an improvement, and describe an optimal strategy under this condition. This will serve as the basis for further discussion on other strategies.

[3]Here, we assume that it is a local optimum with respect to *all* heuristics in the set H.

[4]Another reason that renders multi-armed bandit algorithms unsuitable is that the number of choices for ℓ can be large or potentially unbounded, which corresponds to having a large number of arms (if not infinite) and cannot be efficiently addressed by the conventional multi-armed bandit algorithms.

Algorithm 9 A Modified Version of Algorithm 7

```
1: $\mathbf{x} \leftarrow$ initial solution
2: while stopping criteria not met do
3:     Decide a strategy $\mathcal{S} = (\ell_1, \ell_2, \ell_3, \ldots)$
4:     for $j = 1$ to $\infty$ do
5:         if $(\mathbf{x}' \leftarrow \textsc{FindImprovement}(\mathbf{x}, \ell_j)) \neq$ null then
6:             Replace $\mathbf{x}$ with $\mathbf{x}'$
7:             break
8:         end if
9:     end for
10: end while
```

3.3.1 Optimal Strategy when Success Probabilities are Known

We start with the assumption that if we know, for any ℓ, the probability that a call to $\textsc{FindImprovement}(\mathbf{x}, \ell)$ will successfully return an improving solution, then this information will in theory enable us to construct an optimal strategy that achieves the minimal $\mathbb{E}[Y_{\mathcal{S}}(\mathbf{x})]$.

Theorem 1. *Assume that we are given an incumbent solution* $\mathbf{x}$*, and let* $q(\ell)$ *be the probability that* $\textsc{FindImprovement}(\mathbf{x}, \ell)$ *returns an improving solution. Then the fixed-length strategy* $\mathcal{S}_{\ell^*} = (\ell^*, \ell^*, \ell^*, \ldots)$ *where*

$$\ell^* = \operatorname*{arg\,min}_{\ell < \infty} \frac{1}{q(\ell)} \left(\ell - \sum_{\ell' < \ell} q(\ell') \right)$$

is an optimal strategy for minimizing $\mathbb{E}[Y_{\mathcal{S}}(\mathbf{x})]$.

Proof. The key is to recognize that $\textsc{FindImprovement}(\mathbf{x}, \ell)$ is a Las Vegas algorithm if we set ℓ to ∞, i.e. whenever $\textsc{FindImprovement}(\mathbf{x}, \infty)$ halts, it provides an improving solution. However, the number of operations it executed is a random variable[5]. Let $p(\ell)$ be the probability that $\textsc{FindImprovement}(\mathbf{x}, \infty)$ stops after applying exactly ℓ operations and note that $q(\ell)$ is the cumulative distribution function of p, we can prove the above theorem by reducing it to Theorem 3 of [69]. $\square$

[5]Also note that it can be the case that $\textsc{FindImprovement}(\mathbf{x}, \infty)$ will run forever. Either because $\mathbf{x}$ is a global optimum, or because the set of operations cannot provide enough variation to enable the algorithm to reach certain points in the search space. We ignore these situations for simplicity.

Of course, the assumption of having detailed knowledge of $q(\ell)$ is unrealistic. However, we present this theorem because we would like to provide guarantees on other strategies in terms of this theoretically optimal behavior. In the following, we will denote the above theoretically minimal $\mathbb{E}[Y_{\mathcal{S}}(\mathbf{x})]$ as $m_{\mathbf{x}}$ (or m when the binding to $\mathbf{x}$ is implicitly assumed.)

3.3.2 Balanced Strategies for Unknown Distributions

In this subsection, we consider the situation in which $q(\ell)$ is unknown and describe a "balanced" strategy that tries many different choices of ℓ and spends on each chain length a roughly equal amount of heuristic applications. Specifically, this strategy is an infinite sequence of the form:

$$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, 1, 2, 1, 1, 2, 4, 1, 1, 2 \dots$$

It can be defined recursively as

$$\ell_j = \begin{cases} 2^{k-1}, & \text{if } j = 2^k - 1; \\ \ell_{j-2^{k-1}+1}, & \text{if } 2^{k-1} \leq j \leq 2^k - 1. \end{cases}$$

This sequence was proposed by Luby et al. [69] and we will refer to it as Luby's sequence or Luby's strategy. Using Luby's strategy in Algorithm 9, we have the following theorem.

Theorem 2. *Given an incumbent solution* $\mathbf{x}$*, let* $m_{\mathbf{x}} = \min_{\mathcal{S}} \mathbb{E}[Y_{\mathcal{S}}(\mathbf{x})]$*. If we make calls to* FINDIMPROVEMENT$(\mathbf{x}, \ell)$ *using Luby's strategy* $\mathcal{S}_{Luby}$*, then we have an asymptotic bound on* $\mathbb{E}[Y_{\mathcal{S}_{Luby}}(\mathbf{x})]$ *as* $O(m_{\mathbf{x}} \log m_{\mathbf{x}})$*. Furthermore,* $O(m \log m)$ *is the best asymptotic guarantee we can hope to achieve if we assume an unknown* $q(\ell)$*.*

Proof. This theorem can be proved similar to the proof of Theorem 1 by first recognizing that FINDIMPROVEMENT$(\mathbf{x}, \infty)$ is a Las Vegas algorithm, and defining $p(\ell)$ accordingly. With this condition, we can invoke Theorem 5 and 7 of [69] to prove the above theorem. □

Note that this bound is provided without the assumption that we know $q(\ell)$, and hence can be applied easily. Furthermore, Luby's strategy is regarded as "universal" in the sense that it will eventually extend to long enough sampling lengths if shorter lengths are insufficient[6].

Two variations can be introduced while still retaining the asymptotic guarantee: First, we can change the geometric factor. For example, using 4 as the geometric factor will give the sequence: (1, 1, 1, 1, 4, 1, 1, 1, 1, 4, ...) The second variation is that we can scale the sequence by a constant s, i.e. $(s, s, 2s, s, s, 2s, 4s, s, s, \ldots)$. Adopting these two variations will change the leading constant but will not affect the term $m \log m$. That is, if we write the bound as $O(c \cdot m \log m)$ for some constant c, adopting any or both of these variations will change the constant c, but not the term $m \log m$.

3.3.3 Mixing Luby's Strategy with Other Strategies

For some scenarios, we might have the knowledge that a particular strategy performs well on a good portion of the problem instances but is not robust across all instances. For example, if we define each operation in FindImprovement to have the form of a perturbation followed by a local search, for certain domains, we will have a relatively high probability of finding an improving solution by applying just a few operations[7]. In this situation, a fixed-length strategy $\mathcal{S}_\ell = (\ell, \ell, \ell, \ldots)$ with a small ℓ can perform reasonably well. However, this strategy will fail catastrophically when the assumption is not met (e.g. when the nearest improvement is $\ell + 1$ operations away.)

To hedge against this situation, we propose the idea of mixing strategies without a guarantee with the Luby's strategy. For example, a mixture of $\mathcal{S}_1 = (1, 1, 1, \ldots)$ and $\mathcal{S}_{\text{Luby}}$ will be the sequence:

$$\underline{1}, 1, \underline{1}, 1, \underline{2}, 1, 1, \underline{1}, 1, \underline{1}, 1, \underline{2}, 1, 1, \underline{4}, 1, 1, 1, 1, \underline{1}, 1 \ldots$$

where the underlined elements are from the Luby's sequence while others are from $\mathcal{S}_1$. This sequence is constructed in a way such that at any point in the sequence, the sum of the numbers from $\mathcal{S}_1$ is roughly equal to the sum of the numbers from $\mathcal{S}_{\text{Luby}}$. Note

[6]Note that one can devise other universal strategies by growing the sampling length in certain ways, but as stated in Theorem 2, $O(m \log m)$ is the best asymptotic guarantee one can hope to achieve if we assume unknown $q(\ell)$.

[7]This is also one of the reasons for the popularity of iterated local search [63], which basically employs this kind of operation.

that we can similarly construct a mixture of unequal proportion. As long as we allocate a fixed proportion of computational budget (here, the execution of operations) to the Luby's strategy, we can still retain the asymptotic guarantee from the previous theorem, albeit with the constant term c changing accordingly (e.g. the sequence above will change the constant term to $2c$.)

3.4 Experiments

This section describes the setting and results of our preliminary experiments on using Luby's strategy as the $\mathcal{L}$ policy. We first go over HyFlex, a software framework that the hyper-heuristics research community developed for testing hyper-heuristics. We then describe our implementation of Algorithm 9 with Luby's strategy within HyFlex. Implementing our algorithm on HyFlex enables us to compare our algorithm with other hyper-heuristics that participated in the 2011 Cross-domain Heuristic Search Challenge (CHeSC), which was based on the HyFlex framework. We should point out upfront that most of the hyper-heuristics that participated in CHeSC were based on some sorts of learning during optimization, while on the contrary, our algorithm simply selects heuristics uniformly at random. However, the results suggest that although we could not clinch top spots, our no-learning algorithm is nevertheless very competitive when compared to most of the hyper-heuristics that participated in the CHeSC.

3.4.1 HyFlex and CHeSC 2011

HyFlex is a software framework that was created to aide in the development and testing of hyper-heuristics [80]. It was designed to emphasize the concept of cross domain optimization. To promote this concept, HyFlex features a common interface for dealing with different combinatorial optimization problems. This interface encapsulates problem specific details, such as solution representations and how each heuristic actually works, from the user of the framework[8]. It was expected that the designer of a hyper-heuristic will come up with some mechanism that utilizes these encapsulated heuristics based on the feedback of their performance during the search process.

[8]Note that is corresponds exactly to our first agnostic assumption described in Section 3.2.

Table 3.1: Categorical Summary of Heuristics in CHeSC 2011

	SAT	BP	PS	FS	TSP	VRP
MU	0–5	0,3,5	11	0–4	0–4	0,1,7
RR	6	1,2	5–7	5,6	5	2,3
HC	7,8	4,6	0–4	7–10	6–8	4,8,9
XO	9,10	7	8–10	11–14	9–12	5, 6

Although the HyFlex's interface hides most details of these heuristics, it reveals a categorical description of them by classifying each heuristic as a *mutation* (MU) which modifies a solution in some way with no guarantee of improvement, a *ruin and recreate* (RR) heuristic which destructs a portion of the given complete solution and then reconstruct it in certain fashion, a *hill climber* (HC) which performs a local search returning a solution that has the same or better quality than the original solution, or a *crossover* (XO) which creates a new solution by combining some parts from two given solutions. In addition, HyFlex also provides a parametric control over the *intensity* of the mutation and ruin and recreate heuristics, as well as the *depth of the search* in the hill climbing heuristics. However, for this preliminary study, we only use the default settings for these parameters and do not perform any further tuning.

HyFlex was used as the underlying framework for CHeSC. And for this competition, it provides six problem domains: personnel scheduling (PS), one-dimensional bin packing (BP), permutation flow shop problem (FS), the optimization version of boolean satisfiability problem (SAT), traveling salesman problem (TSP) and vehicle routing problem (VRP). Table 3.1 provides a categorical summary of the heuristics implemented for each problem domain. Each heuristic from a domain is identified by a unique ID number and classified to a category as described above. Besides these two pieces of information, the framework exposes no further details of the heuristic to the participants of the competition.

The ranking method used at CHeSC was inspired from the Formula 1 point scoring system. For each problem instance, the top eight contestants are determined by comparing the median objective values that the contestants achieved over 31 runs where each run lasts for 10 minutes on the organizer's benchmark machine. Each algorithm is then awarded a score according to its ranking. The winner receives 10 points, the runner up gets 8 and then 6, 5, 4, 3, 2 and 1, respectively. In the case of a tie, the corresponding points are added together and shared equally between each algorithm. In CHeSC, each domain contains 5 instances for the final scoring. The winner is the one which has the highest accumulated score from the 30 instances across 6 problem domains.

Twenty teams submitted their algorithms to the CHeSC. The results, along with the description of each algorithm, are available from the website of the competition[9]. In the following, we will compare the results of our algorithm with the 20 contestants that participated in the CHeSC.

3.4.2 Implementation Details

Basically, our algorithm implementation follows the structure of Algorithm 9. Inside the FindImprovement procedure, we use a uniformly random policy, $\mathcal{H}_u$, for choosing heuristics. And we use Luby's sequence $\mathcal{S}_{\text{Luby}}$ as the strategy for choosing the sampling lengths. (Note that every time we find a better solution to replace the current incumbent solution, we will restart the Luby's sequence, i.e., jump back to the beginning of the Luby's sequence.)

One augmentation that we have done for testing on CHeSC was to also consider an "amplified" version of each original heuristic provided by the HyFlex. Let $\mathbf{x}' = h(\mathbf{x})$ represent an application of some heuristic h to some solution $\mathbf{x}$, yielding another solution $\mathbf{x}'$. We can amplify this process by applying h multiple times. If an application yields an $\mathbf{x}'$ that is worse than $\mathbf{x}$, then it is simply discarded. Otherwise, we replace $\mathbf{x}$ with $\mathbf{x}'$. This process repeats for a period of time[10] and returns the final $\mathbf{x}$.

This particular design stems from our desire to promote collaboration among heuristics. Typically, it is very often the case that a stochastic heuristic will only modify a randomly-chosen small portion of the given solution, and this small portion might have very little relation to the modification done by the previous heuristic. Thus, consecutive heuristics might have very little chance to create a collaborated effort. For example, in the SAT domain, we can have the situation that one heuristic flips a variable and the next heuristic flips another variable that has no overlapping clauses with the previously flipped variable. Thus, the two operations are largely independent. Our idea of creating

[9]http://www.asap.cs.nott.ac.uk/external/chesc2011

[10]For example, in our implementation, it is set to 10 milliseconds.

an amplified heuristic out of a heuristic defined by HyFlex is to increase the chance of putting consecutive heuristics to work on related pieces of a solution. With this addition, we double the size of our heuristic set, H, with each original heuristic accompanied by its amplified version. (Also note that for this preliminary study, we do not use the crossover heuristics provided by the HyFlex, for they require two solutions as input, and thus do not fit naturally to our framework.)

Another algorithmic modification that we have made is to allow the incumbent solution to be replaced by a solution of equal quality. This is implemented as follows. When a chained exploration fails to discover an improving solution, we will examine the solution chain $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\ell$ to see if there is a solution $\mathbf{x}_i$ that has the same objective value as the incumbent solution. If multiple solutions were found, we pick the one that is furtherest down the chain, and replace the incumbent solution with this solution. We chose to do this to alleviate stagnation. Note that after performing such kind of replacement, we do not reinitialize the Luby's strategy as we do when we found an improving solution. (Another view of this is that we are treating the group of solutions that are of the same quality as an equivalence class.)

Finally, for implementation convenience, we bound the Luby's sequence at 2^{10} instead of growing arbitrarily. i.e., when $\ell_i = 2^{10}$, the following sequence will be a repetition from the beginning of the sequence to ℓ_i. In our experiments, this bound is rarely reached and should be long enough a chain length for most situations.

3.4.3 Results

As mentioned previously, we use problem domains implemented in CHeSC as a benchmark and compare our results with the algorithms that participated in that contest. For fair comparisons, the CHeSC organizer provides a calibration software that reports, for a user's machine, the time budget equivalent to 10 minutes on the organizer's machine that was used for holding the competition. We performed our experiments on Amazon Web Service's EC2 c4.large virtual machines, for which the calibration software reports a 346 seconds time budget. In accord with CHeSC, we performed 31 runs for each problem instance and took the median values.

Table 3.2 shows the scores achieved by our algorithm, denoted as **Luby**, if it was

Table 3.2: Ranking and Scores if participated in CHeSC

Method	SAT	BP	PS	FS	TSP	VRP	Total
AdapHH	33.60	40.00	8.00	36.00	40.25	15.00	172.85
VNS-TW	33.60	2.00	37.50	34.00	19.25	6.00	132.35
ML	10.50	8.00	31.00	39.00	13.00	21.00	122.50
Luby	23.60	43.00	12.50	12.00	4.00	9.00	104.10
PHUNTER	7.50	2.00	11.50	9.00	25.25	33.00	88.25
EPH	0.00	6.00	10.00	19.00	33.25	12.00	80.25
HAHA	31.60	0.00	24.00	1.50	0.00	13.00	70.10
NAHH	11.50	17.00	2.00	22.00	12.00	5.00	69.50
ISEA	3.50	24.00	14.50	1.50	11.00	4.00	58.50
KSATS-HH	21.85	7.00	8.00	0.00	0.00	21.00	57.85
HAEA	0.00	1.00	1.00	8.00	11.00	26.00	47.00
ACO-HH	0.00	16.00	0.00	8.00	8.00	1.00	33.00
GenHive	0.00	10.00	6.50	5.00	3.00	6.00	30.50
SA-ILS	0.25	0.00	17.50	0.00	0.00	4.00	21.75
DynILS	0.00	9.00	0.00	0.00	12.00	0.00	21.00
AVEG-Nep	10.50	0.00	0.00	0.00	0.00	8.00	18.50
XCJ	3.50	10.00	0.00	0.00	0.00	5.00	18.50
GISS	0.25	0.00	8.00	0.00	0.00	6.00	14.25
SelfSearch	0.00	0.00	3.00	0.00	3.00	0.00	6.00
MCHH-S	3.25	0.00	0.00	0.00	0.00	0.00	3.25
Ant-Q	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 3.3: Using Luby's Sequence vs. Using $\mathcal{S}_1$

Method	SAT	BP	PS	FS	TSP	VRP	Total	Rank
Luby	23.60	43.00	12.50	12.00	4.00	9.00	104.10	4/21
Length 1	0.00	33.00	10.00	21.00	0.00	4.00	68.00	8/21

competing with other contestants in CHeSC. As can be seen, it ranks 4th overall. Although this was not a spectacular performance, it is nevertheless very interesting. It is interesting because most of the other algorithms that participated in the CHeSC employ some sorts of learning or adaptation during the search. On the other hand, our algorithm has no such a learning component: $\mathcal{H}_u$ is just a uniformly random selection, and $\mathcal{S}_{\text{Luby}}$ is a fixed sequence. From our point of view, this can be seen as evidence that the principles underlying the approach are indeed effective.

To further evaluate the contributing factors of our algorithm, we also tested a version with the sampling length constantly setting to 1, i.e., using a strategy $\mathcal{S}_1 = (1, 1, 1, \ldots)$. This is equivalent to performing single-step explorations. Table 3.3 lists the scores and the rank of the algorithm if it had participated in CHeSC. From the result, we can see that using Luby's sequence has an advantage over the length-1 strategy. This can be seen as a further proof that Luby's strategy contributes positively to the algorithm.

3.5 Discussion

The results presented in the last section point out an interesting case: Our approach that uses a no-learning $\mathcal{H}_u$ policy ranked quite high in the CHeSC benchmark. The flip side of this observation is the question that why some of the learning-based approaches performed relatively poorly?

To have a discussion on this matter, we need to first have an (abstract) idea on what these learning mechanisms introduce into the search process. In essence, the actions of a learning mechanism in this context is to change the behavior of the search procedure based on some observed data. Doing this is equivalent to modifying the algorithm so that it will have a different kind of search bias. However, this learning process can go wrong in at least two ways. First note that both the quantity and quality of the collected data have a decisive impact on the outcome of the learning. For example, it is well-known that a good search algorithm should strike a balance between exploration and exploitation. If we learn from a too limited amount of data, the resulting policy can consequently be directed to focus on the limited number of patterns presented in the collected data, and thus put too much emphasis on exploitation (i.e. focusing on using only a small group of patterns.)

Secondly, we can face the problem of not knowing how much we should learn from the collected data. Ideally, we would like to learn prominent patterns in the data, which are more likely to be still applicable in the future, i.e. patterns that generalize. However, different learning models have different capacity for learning patterns. So if we adopt a learning model that is capable of memorizing more details than appropriate, it can be the case that it will encode overly specific behaviors into the resulting policy, i.e. the learning mechanism can result in an *overfitting* to the collected data. This can also lead to a situation similar to over emphasizing exploitation, but in this case, we are likely to use overly complicated patterns that don't generalize well.

Finally, using a more complicated learning mechanism will inevitably lead to more computation, which can take up a larger portion of the total time budget. This will leave a smaller time budget for actually executing the heuristics that perform the search in the solution space.

3.6 Summary

In this chapter, we described an architecture that allows us to chain multiple heuristics in a pipelined fashion. We laid out this architecture to provide a framework for thinking about the issue of how to combine multiple neighborhood-based heuristics in a more principled manner. Based on this architecture, we looked at the algorithmic component (denoted as $\mathcal{L}$) for choosing the lengths of the pipelines, and discussed a strategy for performing this task. In the next chapter, we will start to look at the $\mathcal{H}$ component in our architecture, which specifies how to choose heuristics. More specifically, we would like to explore how to derive useful $\mathcal{H}$ policies from prior experience. And based on our discussion in the last section, we will first start with simple learning mechanisms.

Chapter 4

Fitting Simple Policies for Choosing Heuristics with a Warm-up Period

In the previous chapter, we proposed a framework for combining multiple neighborhood-based heuristics. This framework, presented as Algorithm 6, has two policy components: $\mathcal{H}$ and $\mathcal{L}$, which specify how to pick heuristics for applications and how to to choose the length limit of a solution chain, respectively. With this framework, We demonstrated that by employing a good $\mathcal{L}$ policy, we can achieve a relatively good performance even in the case that we just pick heuristics uniformly at random. We also hypothesized that the relatively poor performance of some other "learning-based" approaches that participated in the CHeSC competition could be due to employing overly-complex learning models that are prone to introduce overfitting and thus making the overall optimization process unable to achieve a good balance between exploration and exploitation. Furthermore, complex learning mechanisms also require more computation and thus can take up a larger portion of our fixed computation budget.

In this chapter, we start to explore how to reasonably derive an $\mathcal{H}$ component that can enhance the performance of the overall search algorithm. Based on our previous hypothesis, we will restrict ourselves to simple learning mechanisms first, which are less prone to overfitting and thus are better at avoiding learning random patterns presented in the data.

We adopt the following procedure for collecting the data for learning: For each run, we will allocate a period of time at the beginning of the run as a warm-up period. In this warm-up period, we will execute the baseline algorithm, which is just the configuration that uses Luby's strategy for the $\mathcal{L}$ component and uniformly random selection as the $\mathcal{H}$ policy (exactly the same algorithm that we presented in the previous chapter.) We will collect relevant data during this warm-up period, and at the end of this warm-up period, we will feed the collected data into a learning mechanism to derive a learned $\mathcal{H}$ policy, which will then be used for the remaining of the run.

In the following sections, we will look at some simple learning mechanisms that instantiate $\mathcal{H}$ policies based on the collected data. As in Chapter 3, we will use the CHeSC benchmark as an assessment of the performance.

4.1 Pruning Heuristic Set

Perhaps the simplest mechanism that we can add to our algorithm is to keep an eye on whether a heuristic has ever participated in the creation of a solution chain that led to the discovery of an improving solution. This data gathering is done during the warm-up period of a run. Then after the warm-up period, we prune away the heuristics that have never showed up in such a record, reducing the size of the heuristic set H. This is equivalent to setting the probability of selecting those heuristics to zero and redistributing the probability mass equally to the remaining heuristics. Note that as a safeguard against making pruning based on insufficient information, we do not prune any heuristics that had not been applied for more than 15 times in the initial warm-up stage.

Table 4.1 shows the scores obtained by our algorithm with the addition of the above simple pruning mechanism (denoted as **Pruning** in the table), comparing again to the methods originally participated in the CHeSC. As before, our experiments are performed on Amazon Web Service's EC2 c4.large virtual machines, and we set the warm-up period to one minute. As can be seen, our procedure now obtains a higher total score and is ranked second overall according to the CHeSC scoring rules.

This demonstrates that depending on the problem domain or a particular instance of the problem domain, it might be the case that not all heuristics contribute positively to performance. It pays to be discretionary on which heuristics to apply and in this way, we can be more efficient about spending our search time budget.

Table 4.1: Performance of a Simple Pruning Mechanism

Method	SAT	BP	PS	FS	TSP	VRP	Total
AdapHH	34.25	40.00	8.00	35.00	40.25	14.00	171.50
Pruning	21.00	43.00	18.50	26.75	14.00	12.00	135.25
VNS-TW	34.25	2.00	37.50	30.00	18.25	6.00	128.00
ML	11.00	8.00	31.00	36.00	13.00	21.00	120.00
PHUNTER	7.50	2.00	11.50	7.00	24.25	33.00	85.25
EPH	0.00	6.00	9.00	18.00	33.25	12.00	78.25
HAHA	32.25	0.00	22.50	1.25	0.00	13.00	69.00
NAHH	11.50	17.00	1.00	21.00	11.00	5.00	66.50
KSATS-HH	22.00	7.00	7.00	0.00	0.00	21.00	57.00
ISEA	3.50	24.00	14.50	1.50	10.00	3.00	56.50
HAEA	0.00	1.00	1.00	5.75	10.00	25.00	42.75
ACO-HH	0.00	16.00	0.00	7.75	7.00	1.00	31.75
GenHive	0.00	10.00	6.00	5.00	2.00	6.00	29.00
SA-ILS	0.25	0.00	17.50	0.00	0.00	4.00	21.75
DynILS	0.00	9.00	0.00	0.00	10.00	0.00	19.00
AVEG-Nep	10.50	0.00	0.00	0.00	0.00	8.00	18.50
XCJ	3.50	10.00	0.00	0.00	0.00	5.00	18.50
GISS	0.25	0.00	8.00	0.00	0.00	6.00	14.25
SelfSearch	0.00	0.00	2.00	0.00	2.00	0.00	4.00
MCHH-S	3.25	0.00	0.00	0.00	0.00	0.00	3.25
Ant-Q	0.00	0.00	0.00	0.00	0.00	0.00	0.00

4.2 Learning the Frequency of Selection

A natural generalization of the above pruning mechanism is to use a degree of frequency instead of a crisp inclusion or exclusion. To implement this idea, we will first run the uniformly random selection for one minute as before. During this warm-up period, if a solution chain leads to an improving solution, we record the corresponding sequence of heuristics that has generated that solution chain. This process will give us a log of sequences and we can accordingly make an estimation of the frequency of applying each heuristic. That is, at the end of the warm-up period, we will construct a simple probabilistic model which is essentially a table describing the probability of applying each heuristic. And for the rest of the run, we will use this model for selecting heuristics instead of choosing heuristics uniformly at random.

In our implementation, in order to achieve a more accurate modeling, we also adopt the following extension: We split the sequence log into two collections. One collection contains the *singleton* sequences, i.e., the sequences of length 1, and the other contains all the other sequences that are of length 2 or longer. We then estimate two separate models based on these two collections. With this specialization, our policy for choosing heuristics can now be conditioned on the length of the heuristic chain. That is, when the length policy $\mathcal{L}$ suggests that the next heuristic chain should be of length 1 (i.e. $\ell = 1$), we will use the singleton model for choosing heuristics. On the other hand, if $\ell > 1$, we will use the model built on sequences of length 2 or more to select heuristics.

One pitfall of the above idea of estimating probabilistic models is that it only works well when we have collected a large enough set of sequences. If that is not the case, the model may be biased to some random fluctuations and may impact performance negatively. To alleviate this situation, in our experiments, we adopt a simple strategy: if the number of sequences is less than 10, we will use a fallback heuristic selection policy which for the first heuristic in the sequence, selects a random perturbation heuristic (in the HyFlex terminology, a MU or RR heuristic), and for the following heuristics in the sequence, it selects only the hill climber heuristics (also choosing them uniformly at random.) This fallback heuristic selection policy is based on the operations of the popular Iterated Local Search [64].

The same as previously, we perform the experiment on the CHeSC benchmark. Ta-

Table 4.2: Performance of Using Frequency Learning

Method	SAT	BP	PS	FS	TSP	VRP	Total
AdapHH	32.35	40.00	7.00	35.00	39.25	15.00	168.60
Frequency	25.95	43.00	28.50	26.50	21.00	10.00	154.95
VNS-TW	32.85	2.00	37.00	29.50	18.25	6.00	125.60
ML	11.00	8.00	29.50	36.00	13.00	21.00	118.50
PHUNTER	8.00	2.00	11.00	7.00	22.25	32.00	82.25
EPH	0.00	6.00	8.50	18.00	31.25	11.00	74.75
HAHA	29.85	0.00	21.50	1.50	0.00	13.00	65.85
NAHH	11.50	17.00	1.00	21.00	10.00	5.00	65.50
ISEA	3.50	24.00	14.00	1.50	9.00	5.00	57.00
KSATS-HH	21.20	7.00	6.00	0.00	0.00	21.00	55.20
HAEA	0.00	1.00	1.00	6.00	11.00	26.00	45.00
ACO-HH	0.00	16.00	0.00	8.00	6.00	1.00	31.00
GenHive	0.00	10.00	4.50	5.00	2.00	6.00	27.50
SA-ILS	0.60	0.00	15.50	0.00	0.00	4.00	20.10
DynILS	0.00	9.00	0.00	0.00	10.00	0.00	19.00
AVEG-Nep	10.50	0.00	0.00	0.00	0.00	8.00	18.50
XCJ	3.50	10.00	0.00	0.00	0.00	5.00	18.50
GISS	0.60	0.00	8.00	0.00	0.00	6.00	14.60
SelfSearch	0.00	0.00	2.00	0.00	2.00	0.00	4.00
MCHH-S	3.60	0.00	0.00	0.00	0.00	0.00	3.60
Ant-Q	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 4.3: The Contribution of the Fallback Mechanism

Method	SAT	BP	PS	FS	TSP	VRP	Total	Rank
Frequency	25.95	43.00	28.50	26.50	21.00	10.00	154.95	2/21
w/o Fallback	29.35	43.00	14.50	28.00	22.00	12.00	148.85	2/21
Fallback Only	11.00	33.00	23.50	18.50	27.60	6.00	119.60	4/21

ble 4.2 shows the performance of adopting the above idea of estimating the frequency of each heuristic (denoted as **Frequency** in the table). As we can see, it obtains a higher number of points than the previous **Pruning** strategy (154.95 vs. 135.25), showing the effectiveness of using the probabilistic models.

In addition to the above experiment, we also performed an auxiliary experiment that compares and evaluates the effect of incorporating the fallback strategy. The results are listed in Table 4.3. We can see that although the fallback strategy doesn't help in terms of the ranking, it does help the algorithm to achieve a higher score. Most notably, there was a good improvement in the PS domain. A post-experiment analysis reveals that the operations in the PS domain are generally very time-consuming, which leads to the situation that we often cannot collect enough sequences for learning. In this situation, adopting a fallback strategy seems to be a reasonable approach and compensates well for the performance.

4.3 Learning the Bigram Statistics

A natural question following the above experiment is whether we can gain more improvement by adopting a more elaborate probabilistic model. In this section, we experiment with a simple extension that models the conditional probability of selecting a heuristic based on the heuristic applied in the previous step. That is, using the notation presented in Figure 3.1, we want to model $P(h_i|h_{i-1})$. As before, we estimate the model based on the relative frequencies observed in the log of sequences that we collected during the first minute of the run. A maximum likelihood estimate can be specified as follows:

$$P(h_i|h_{i-1}) = \frac{C(h_{i-1}h_i)}{\sum_{h \in H} C(h_{i-1}h)}$$

where $C(h_{i-1}h_i)$ denotes the number of occurrences of a particular adjacent pattern of heuristics $h_{i-1}h_i$ observed in the sequence log. Such kind of adjacency modeling is often used in the tasks relating to natural language processing (NLP), and is referred to as the *bigram* model in the NLP field [70].

Table 4.4 shows the results of adopting such a model (denoted as **Bigram** in the table.) We can see that the scores are further improved and now our method places at the top spot compared to other algorithms participated in the CHeSC. This shows that modeling the sequential dependences can contribute positively and it can be beneficial to adopt such a consideration into the policy for choosing heuristics.

4.4 Summary

In Chapter 3, we experimented with a dispatching policy $\mathcal{H}_u$ that each time select a heuristic uniformly at random from the set of available heuristics. Using this uniformly random policy as a baseline, in this chapter, we have examined three simple learning mechanisms that changes its behavior for choosing heuristics based on the collected data. This data collection is done in a warm-up period of a run. Then after the warm-up period, the algorithm switches to a learned policy for choosing heuristics. We showed that even in the case of just using a simple learning mechanism, we can achieve a high rank in the CHeSC benchmark. Also note that, for the experiments, we didn't attempt an intensive tuning on the algorithm parameters such as the length of the warm-up period, the criterion for pruning, etc. The value of these parameters are mostly picked arbitrarily. So it's possible to get an even better result by changing the values of these parameters.

Reviewing the methods defined and tested in this chapter, we can see that the learning processes presented in this chapter are mostly based on (variations of) learning the *frequency* of applying each heuristic. However, besides the frequency, we may also wonder whether the *structure* can also be an important factor. As we saw in the last section, mod-

Table 4.4: Performance of Using Bigram Learning Mechanism

Method	SAT	BP	PS	FS	TSP	VRP	Total
Bigram	29.70	43.00	28.00	37.00	26.00	12.00	175.70
AdapHH	31.43	40.00	7.00	34.00	37.25	14.00	163.68
VNS-TW	31.93	2.00	37.00	29.00	18.25	6.00	124.18
ML	11.00	8.00	29.50	34.50	12.00	21.00	116.00
PHUNTER	8.00	2.00	11.00	6.00	22.25	32.00	81.25
EPH	0.00	6.00	8.50	16.00	31.25	11.00	72.75
HAHA	28.43	0.00	21.50	0.83	0.00	13.00	63.77
NAHH	11.50	17.00	1.00	19.50	9.00	5.00	63.00
ISEA	3.50	24.00	14.00	1.50	9.00	4.00	56.00
KSATS-HH	20.70	7.00	6.50	0.00	0.00	21.00	55.20
HAEA	0.00	1.00	1.00	5.33	10.00	26.00	43.33
ACO-HH	0.00	16.00	0.00	6.33	6.00	1.00	29.33
GenHive	0.00	10.00	4.50	5.00	2.00	6.00	27.50
SA-ILS	0.60	0.00	15.50	0.00	0.00	4.00	20.10
DynILS	0.00	9.00	0.00	0.00	10.00	0.00	19.00
AVEG-Nep	10.50	0.00	0.00	0.00	0.00	8.00	18.50
XCJ	3.50	10.00	0.00	0.00	0.00	5.00	18.50
GISS	0.60	0.00	8.00	0.00	0.00	6.00	14.60
SelfSearch	0.00	0.00	2.00	0.00	2.00	0.00	4.00
MCHH-S	3.60	0.00	0.00	0.00	0.00	0.00	3.60
Ant-Q	0.00	0.00	0.00	0.00	0.00	0.00	0.00

eling the sequential dependencies between heuristics can also be beneficial to the overall search performance. However, by using a bigram model, we can only capture the structural properties implicitly. In the next chapter, we will propose a method that explicitly captures the structures present in the sequences and thus learns a more comprehensive model that describes the collaboration patterns of the heuristics presented in the data.

Chapter 5

Learning Explicit Collaboration Patterns from Run Logs

In the previous chapters, we have developed and experimented with a high-level framework for combining multiple neighborhood-based heuristics. The fundamental idea of this framework is to chain multiple heuristics in a pipelined fashion so that we can utilize the interaction between heuristics. To derive a concrete algorithm from this framework, one needs to supply two user-defined components: 1) a policy $\mathcal{H}$ for selecting heuristics, and 2) a policy $\mathcal{L}$ for choosing the length of the pipeline that chains the selected heuristics. In Chapter 3, we offered a theoretical discussion on the design of the $\mathcal{L}$ component and described a policy that has an asymptotic guarantee. In Chapter 4, we experimented with some simple learning mechanisms that derive adjusted $\mathcal{H}$ policies from data collected during a warm-up period. Those learning mechanisms were mostly based on learning the frequency for applying each heuristic. In this chapter, we explore a way to more explicitly capture the collaboration patterns among heuristics, and investigate whether using an $\mathcal{H}$ policy that utilizes these collaboration patterns offers any advantage.

Our proposal is based on an idea similar to the previous chapter that we can first run the simple baseline algorithm presented in Chapter 3, and record the positive experiences encountered during its execution. By analyzing this record of positive experiences, we can potentially extract useful interaction patterns among the heuristics, and accordingly, use this knowledge to construct a better $\mathcal{H}$ policy.

In this chapter, in addition to the overall search performance, we are also interested in whether our policy construction procedure can produce policies that *generalize* well. This generalization ability is an indication on whether we are learning useful information or just merely capturing non-reusable patterns presented in the data. To reasonably assess this generalization aspect, we need to first lay out the underlying assumptions properly. So in the next section, we will begin with a description on our distributional assumption on the problem instances. Following that, Section 5.2 describes our policy construction algorithm. The experiments and results are then presented in Section 5.3, and Section 5.4 offers some discussion. Finally, Section 5.5 concludes this chapter.

5.1 Distributional Assumption

In this chapter, we investigate the issue of how to construct a policy for choosing how to apply a given set of heuristics. Previously, we have experimented with a baseline policy $\mathcal{H}_u$, which simply selects a heuristic uniformly randomly from the set of heuristics each time it is consulted. Using $\mathcal{H}_u$ as a baseline, in this work, we would like to study how to come up with a better policy, and more importantly, how to automate its construction.

However, in order to proceed, we need to define what it means to be a better policy. Apparently, if we don't limit the scope of applicability, it can be challenging to give a definition that is both reasonable and easy to work with. It can be a daunting task to search for a "universally good" policy because this definition amounts to enumerating "all possible problem domains[1]," which is itself abstract to begin with. So instead, we will handle the construction of policies in a per-domain fashion.

More importantly, we will proceed with a distributional assumption that we are given a set of problem instances for training, and the future problem instances will be from the same distribution that we drew the training set from. To elaborate more on this, we implicitly assume that the future problem instances will have similar characteristics as those in the training set. Hence, there is a reason to hope for the possibility that a policy derived from the training set may generalize well to the future instances.

[1]As mentioned previously, we define a problem domain as a set of heuristics together with a mechanism to evaluate the quality of a solution. These objects are treated as black-boxes and we assume no detailed knowledge was revealed about the inner work of these objects. Note that with this definition, with a different set of heuristics, we will have a different problem domain, even if the underlying combinatorial optimization problem is in fact the same.

To state more explicitly, we define our task as follows: With a fixed set of heuristics, construct a dispatching policy $\mathcal{H}$ based on a set of problem instances drawn from a target distribution $\mathcal{D}$ so that the algorithm using $\mathcal{H}$ will have a good *expected* performance over the future instances drawn from $\mathcal{D}$. With this notion of expected performance over a target problem instance distribution, we can formally compare two policies and make statistical statements about our observations. Empirically, this setup also allows us to use cross-validation to assess the effectiveness of our policy construction procedure, which we introduce in the following section.

Note that since we intend to evaluate the generalization ability with the above setting, using the CHeSC evaluation protocol as we did in the previous chapters becomes inappropriate. This is because 1) in the CHeSC evaluation protocol, each run is considered independent, and there is no notion of the "future problem instances." 2) CHeSC benchmark adopted problem instances that tend to be eclectic, and this property does not fit well with our distributional assumption. Thus, for the experiments, we will instead use a setup that allows us to assess performance using cross-validations. However, we will keep using the HyFlex API, which we review in Section 5.3.1, as the experimentation platform to take advantage of its collection of implemented heuristics.

5.2 Automated Policy Construction

Our idea for an automated policy construction procedure is that it will take the set of training problem instances and perform the following operation: For each training problem instance, it will attempt to solve it using the configuration that employs $\mathcal{H}_u$ as heuristic selection policy and Luby's strategy as length selection policy (basically, the same baseline algorithm that we developed in Chapter 3.) During a run, if the solver created a solution chain that led to an improving solution (i.e. a solution that is better than the previous best solution), we will record the corresponding sequence of heuristics that generated that solution chain. This process will give us a log of sequences. And because these sequences are the ones that had led to improving solutions, we believe it is possible to construct a better policy through analysis of this log.

For example, suppose that by running the baseline algorithm, we collected a small log

```
10
6
10
8
0  8
1  7
1  10  0  7
0  8
0  0  10  8
0  7
1  10
7
1  8
0  7
```

(a) A Sample Log of Sequences

```
10
6
10
8
[0  8]
1  7
[1  10]  [0  7]
[0  8]
0  0  10  8
[0  7]
[1  10]
7
1  8
[0  7]
```

(b) Segmented Version

Pattern	Probability
0	0.1
1	0.1
6	0.05
7	0.1
8	0.15
10	0.15
[0 8]	0.1
[1 10]	0.1
[0 7]	0.15

(c) A Probabilistic Model

Figure 5.1: On the left, we list a sample log of sequences. Each number number in this log maps to a heuristic from a problem domain. In this place, the problem domain has 11 heuristics and they are indexed from 0 to 10. Each line of this log represents a sequence of heuristics that had created an improving solution. In the middle, some of the common patterns are segmented out. And on the right, we have a model estimated from the segmented sequences.

like the one shown in the Figure 5.1a, in which each number maps to a heuristic from a problem domain. In this example, the problem domain has 11 heuristics and they are indexed from 0 to 10. Each line of this log represents a sequence of heuristics that had created an improving solution. By inspecting this log, we can observe some interesting patterns that seem to occur more frequently than others. Our hypothesis is that each of these patterns corresponds to an effective processing flow, and more importantly, it can be thought of as a "heuristic macro" representing a collaboration of the participating heuristics. If we can segment them out, like what are shown in Figure 5.1b, then by simple counting, we can estimate a probabilistic model that encodes these structures, such as the one shown in Figure 5.1c. The idea is that by sampling this model instead of sampling uniformly randomly like $\mathcal{H}_u$, we will be able to reuse those patterns and compose new sequences of heuristics with these "heuristic macros" embedded as sequence components. This can potentially improve the efficiency of the search[2].

However, it is not immediately obvious how to come up with such a segmentation without manual intervention. As mentioned above, our goal is a fully automated procedure for constructing policies, so solving this issue is necessary. Our idea for proceeding is to recognize that if someone handed us a probabilistic model, then we would be able to divide a sequence into its most probable segmentation using dynamic programming. On the other hand, once we have such a segmentation for every sequence in the log, we can estimate a probabilistic model by simple counting. These two steps seem to form an Expectation-Maximization (EM) loop. So we think it is possible to iteratively build a model based on an EM procedure.

The dynamic programming procedure for segmenting the sequences is listed in Algorithm 10. It takes as input a model $\mathcal{M}$ of the same form as the one shown in Figure 5.1c and divides the given sequence $\mathbf{s} = s_0 s_2 \dots s_{n-1}$ into parts so that the resulting parts in combination has the highest probability according to $\mathcal{M}$. Note that in Algorithm 10, we index the elements of a sequence starting from 0 instead of 1, and and we use $\mathcal{M}[\ \cdot\ ]$ to denote a query to the probability table of the model $\mathcal{M}$. For example, $\mathcal{M}[\texttt{0 8}]$ will yield

[2]Also note that although there were 11 heuristics, not all of them are included in the model. This is because some of the heuristics were never part of any sequences that had led to an improving solution and hence, are being left out of the model. This can be seen as a way to prune the ineffective heuristics, an idea that we also experimented with in the Chapter 4.

Algorithm 10 Segmenting a Sequence Given a Model

```
Input: a sequence $\mathbf{s}$ and a model $\mathcal{M}$.
 1: $n \leftarrow$ the length of sequence $\mathbf{s}$
 2: $k \leftarrow$ the length of the longest pattern in model $\mathcal{M}$
 3: $\mathbf{p} \leftarrow$ an array of length $n$ (for keeping probabilities)
 4: $\mathbf{b} \leftarrow$ an array of length $n$ (for keeping backpointers)
 5: for $j = 0$ to $n - 1$ do
 6:     $\mathbf{p}[j] \leftarrow \mathcal{M}[s_0 s_1 \cdots s_j]$
 7:     $\mathbf{b}[j] \leftarrow 0$
 8:     for $i = \max(j - k + 1, 1)$ to $j$ do
 9:         if $\mathcal{M}[s_i s_{i+1} \cdots s_j] \neq 0$ then
10:             if $\mathbf{p}[j] < \mathbf{p}[i-1] \times \mathcal{M}[s_i s_{i+1} \cdots s_j]$ then
11:                 $\mathbf{p}[j] \leftarrow \mathbf{p}[i-1] \times \mathcal{M}[s_i s_{i+1} \cdots s_j]$
12:                 $\mathbf{b}\,[j] \leftarrow i$
13:             end if
14:         end if
15:     end for
16: end for
17: $\mathbf{d} \leftarrow$ a stack (for recording the segments of $\mathbf{s}$)
18: $j \leftarrow n - 1$
19: while $j \geq 0$ do
20:     $i \leftarrow \mathbf{b}\,[j]$
21:     Push $s_i s_{i+1} \cdots s_j$ as a unit to $\mathbf{d}$
22:     $j = i - 1$
23: end while
24: return $\mathbf{d}$
```

0.1 by the model shown in Figure 5.1c.

To explain the core idea of this algorithm, note that for a partial sequence $s_0 s_1 \cdots s_j$, if we consider a tail part $s_i s_{i+1} \cdots s_j$ as a unit, then the optimal probability of the model $\mathcal{M}$ generating the partial sequence $s_0 s_1 \cdots s_j$ (with the tail part $s_i s_{i+1} \cdots s_j$ as a unit) is the probability of the most probable segmentation of $s_0 s_1 \cdots s_{i-1}$ times $\mathcal{M}[s_i s_{i+1} \cdots s_j]$. With this observation, we can recursively define the optimal segmentation given a model and use a dynamic programming approach to solve for it.

Equipped with the above algorithm for segmenting sequences, we can now construct an EM procedure as follows: First, we initialize a model by collecting all the sub-sequences appearing in the sequence log as patterns[3], with the premise that the number of appearances is higher than certain threshold θ.[4] Each pattern's initial probability is set to be proportional to the number of appearances in the log. With this initial model, we then run Algorithm 10 to obtain a segmentation for each sequence in the log. Based on the segmented version of the log, we re-estimate a model by counting the frequency of each pattern. Finally, we proceed to the next iteration by performing a segmentation using the new model. This process iterates until we get a re-estimated model that is identical to the old one.

5.3 Experiments and Results

This section describes the implementation details and the results of an experimental analysis of our policy construction algorithm. We will first provide a brief review on HyFlex, a software framework upon which we built our programs. We then describe the implementation details of our approach. Following that, we will show the results of experiments and compare the proposed approach to two alternatives.

[3]We can also set a bound on the length of the patterns for efficiency purposes.

[4]Note that we do not impose this threshold restriction on sub-sequences of length 1. Otherwise, it may result in an error when we apply the initial model to the task of segmenting sequences, i.e., the situation that the model doesn't contain a heuristic that appears in the sequence.

5.3.1 HyFlex and Its Extensions

HyFlex is a software framework that was developed to facilitate the research of hyper-heuristics [80]. The benefit of using HyFlex is that it offers a common interface for dealing with different combinatorial optimization problems. This interface encapsulates problem specific details, such as solution representations and how each heuristic actually works, from the user of the framework. Thus, it provides a convenient platform on which we can experiment with our ideas.

The initial HyFlex software package has four problem domains built into it: maximum satisfiability, one-dimensional bin packing, permutation flow shop and personnel scheduling. Later, for the purpose of the CHeSC competition, two more problem domains (Traveling Salesman Problem and Vehicle Routing Problem) are added into the standard package. Each of these problem domains has an implementation consisting of a set of problem specific heuristics ready to be called from a unified interface. Note that HyFlex also offers a parametric control over some tunable aspects of the defined heuristics. However, for simplicity, we will only use the default parameters and will not perform any further tuning on them.

Since its initial release, HyFlex has being extended, e.g., [1, 104]. In this work, we also use an extension [1] that offers an implementation of the quadratic assignment problem.

As mentioned in Section 5.1, our approach makes a distributional assumption about the problem instances from a domain. However, the default collections of problem instances in both the original HyFlex and Adriaensen et al.'s extension [1] don't seem to follow this assumption: For each domain, they tend to put together problem instances that are characteristically dissimilar to each other, which violates our distributional assumption. To create our target setting, we modified their code so that we can load problem instances from sources that follow more closely to our distributional assumption.

Specifically, we will test our approach with three problem domains: permutation flow shop problem, 1-D bin packing and quadratic assignment problem (they will be denoted as FS, BP, and QAP, respectively.) The descriptions of these problem domains and the implementation details of their accompanying heuristics can be found in [54], [103] and [1]. For each problem domain, we will run experiments on multiple sets of problem instances where each set follows a particular distribution. Table 5.1 shows the name of each instance

Table 5.1: Problem Domains & Sets of Instances

Domain	Instance Sets	Source
FS	100x10, 100x20, 200x10, 200x20, 500x20	[99]
BP	testdual4, testdual7, testdual8, testdual11	[17]
QAP	tai45e, tai75e	[30]

set and the source where we obtained them.

5.3.2 Implementation Details

Our implementation starts with an initial stage that collects a log of sequences. For an instance set, we will run the baseline algorithm from Chapter 3 on each of the training problem instances 31 times (with different random seeds) to collect sequences of heuristics that lead to improving solutions. In our experiments, each run lasts for 3 minutes on an Amazon Web Service's EC2 c4.large virtual machine. The collected sequences[5] are then fed into a model construction procedure to produce a probabilistic model that will later be used as the policy for selecting heuristics.

In order to achieve a more accurate modeling, we further adopt the following extension to the model construction procedure described in the previous section: We split the sequence log into two collections. One collection contains the *singleton* sequences, i.e., the sequences of length 1, and the other contains all the other sequences, which are of length 2 or longer. We then build two separate models based on these two collections. With this specialization, our policy for choosing heuristics can now be conditioned on the length of the heuristic chain. That is, when the length policy $\mathcal{L}$ suggests that the next heuristic chain should be of length 1 (i.e. $\ell = 1$), we will use the singleton model for choosing heuristics. On the other hand, if $\ell > 1$, we will use the model built on sequences of length 2 or more to select among heuristics and heuristic macros.

Note that there is a parameter θ that needs to be set for the model construction procedure. It specifies the minimal number of appearances in the log in order for a subsequence to be considered as a pattern in the initial model. In order to account for the

[5]Note that this collection contains all sequences from all runs on all training problem instances from the instance set.

variations of the number of sequences in the log (denoted as N), we use the following formula,

$$\theta = \max(3, N \times \rho)$$

where we use $\rho = 0.01$ for all our experiments.

Once we have constructed a model, we will then use it as the policy for choosing heuristics (i.e. the $\mathcal{H}$ component) and plug it into Algorithm 6. As for the length policy (i.e. the $\mathcal{L}$ component), we will continue to use the Luby's sequence in the following experiments.

5.3.3 Empirical Results

With our distributional assumption, we conducted experiments on each instance set separately, and in order to assess the generalization aspect, the problem instances were used in a leave-one-out fashion, e.g., if we want to assess the performance of the proposed method on the first problem instance, we will use the second to the tenth instance (assuming there are ten instances) as the training problem instances to collect sequences.

With a collection of sequences, we then built a policy $\mathcal{H}$ by performing the aforementioned policy construction procedure. With the resulting policy $\mathcal{H}$, along side with Luby's strategy as the $\mathcal{L}$ component, we constructed an algorithm out of the template of Algorithm 6. The resulting algorithm was then tested for 31 runs, each run lasting for 30 seconds. Note that the discrepancy between the amount of time allocated for the training runs and the amount of time allocated for the testing runs is because 1) we would like to collect a sufficient number of sequences, and 2) we would like to obtain patterns that may only show up in the later stage of a run. Basically, our objective is that through these experiments, we would like to see whether the proposed approach offers any speed-up over the baseline approach, i.e., whether the algorithm with a learned policy can find substantially better solution compared to the baseline algorithm at an earlier stage. In this way, we can measure whether a constructed policy contains information that is useful for enhancing the search ability of the algorithm.

As a further comparison, we also tested a configuration that uses what we call "plain" models. This kind of model only considers individual heuristics and does not extend to patterns of length 2 or more. Basically, it is just a frequency estimate on each of the individual heuristics. Note that for this configuration, we also used the singleton sequence specialization mentioned previously, so that we can compare meaningfully on whether there is an advantage from building a more elaborated model.

To provide a fair comparison, we also account for the effect of the initial solution by synchronizing the three approaches to start with the same initial solution. That is, for the i-th run of all three approaches, the same initial solution is used so that none will have the advantage of starting from a better solution. This setting also allows us to use paired t-test and Wilcoxon signed-rank test to statistically evaluate the results.

The results of the experiments on the FS instance sets are shown in Table 5.2. For each target problem instance, we compare the results of (1) the proposed approach, denoted as Macro, (2) the baseline policy (i.e. using uniformly random selection), denoted as Baseline, and (3) the configuration that uses plain models, denoted as Plain. Each number listed in these tables represents the Averaged Relatived Percentage Deviation (ARPD) over 31 runs:

$$\text{ARPD} = \frac{1}{31}\sum_{i=1}^{31}\frac{\text{objval}_i - \text{bestknown}}{\text{bestknown}} \times 100$$

where objval_i is the final objective value obtained from run i, and bestknown represents the objective value of the current best-known solution.

To ensure a sound analysis, we also performed statistical tests to see if there is a significant difference between the results of different methods. The statistical tests are arranged as follows: If the pairwise differences between the results of two approaches are distributed normally (as certified by a normality test with p-value > 0.1), then we use a paired t-test. Otherwise, we use the Wilcoxon signed-rank test. The results of the tests are also shown in the tables: we use a $^+$ symbol to denote that the result is significantly different (as determined by a p-value < 0.1) from the result of the baseline, and a * symbol to represent that there is a significant difference (also thresholded by p-value < 0.1) between the result of the proposed approach and the result of the approach that uses plain models.

As shown in Table 5.2, for the FS domain, the proposed approach seems to have an advantage over the baseline method, as supported by the generally better ARPD values. Furthermore, the significance of this advantage appears to increase as the problem becomes larger (i.e., increasing the number of jobs) or harder (i.e., increasing the number of machines

Table 5.2: Results on Taillard's FS Instance Sets

(a) 100x10

Instance	Baseline	Plain	Macro
01	0.240398	0.182255^{+}	**0.135294**$^{+*}$
02	0.278014	0.230975^{+}	**0.223738**$^{+}$
03	0.053991	**0.051717**	0.052854
04	0.866018	0.748280^{+}	**0.735446**$^{+}$
05	0.741693	0.613062^{+}	**0.604802**$^{+}$
06	0.124701	0.108277	**0.094286**$^{+}$
07	0.132030	0.096861	**0.089366**$^{+}$
08	0.642060	0.522607^{+}	**0.487575**$^{+}$
09	0.410437	0.319229^{+}	**0.262086**$^{+*}$
10	0.285328	0.129695^{+}	**0.093270**$^{+}$

(b) 100x20

Instance	Baseline	Plain	Macro
01	2.640147	2.245373^{+}	**2.102340**$^{+*}$
02	2.160972	1.742551^{+}	**1.671075**$^{+}$
03	1.870361	1.556576^{+}	**1.482503**$^{+}$
04	1.849860	1.549355^{+}	**1.515908**$^{+}$
05	2.125333	1.913311^{+}	**1.731942**$^{+*}$
06	2.196326	2.010807^{+}	**1.958598**$^{+}$
07	2.155341	1.783766^{+}	**1.758548**$^{+}$
08	2.436111	2.265775^{+}	**2.055122**$^{+*}$
09	2.163732	1.847063^{+}	**1.791544**$^{+}$
10	1.829996	1.564772^{+}	**1.459484**$^{+}$

(c) 200x10

Instance	Baseline	Plain	Macro
01	0.192740	0.176406	**0.159775**$^{+}$
02	0.585139	0.437084^{+}	**0.405688**$^{+}$
03	0.525427	0.378343^{+}	**0.367415**$^{+}$
04	0.038808	0.035549	**0.034364**
05	0.140386	0.123527	**0.120768**$^{+}$
06	0.362899	0.264523^{+}	**0.237040**$^{+}$
07	0.315031	0.276693	**0.242218**$^{+}$
08	0.382106	0.290713^{+}	**0.251030**$^{+}$
09	0.249399	0.207678^{+}	**0.188826**$^{+}$
10	0.339654	0.348417	**0.218478**$^{+*}$

(d) 200x20

Instance	Baseline	Plain	Macro
01	1.986774	1.773545^{+}	**1.688830**$^{+*}$
02	2.606157	2.311593^{+}	**2.160424**$^{+*}$
03	2.470325	2.252717^{+}	**2.130902**$^{+*}$
04	2.240183	2.047636^{+}	**1.790716**$^{+*}$
05	1.760885	1.355761^{+}	**1.261786**$^{+*}$
06	2.212979	1.956958^{+}	**1.847276**$^{+*}$
07	2.034019	1.660041^{+}	**1.541913**$^{+*}$
08	2.131184	1.978347^{+}	**1.707110**$^{+*}$
09	2.499481	2.294842^{+}	**2.167735**$^{+*}$
10	2.302188	2.008985^{+}	**1.914965**$^{+*}$

(e) 500x20

Instance	Baseline	Plain	Macro
01	1.602195	1.398687^{+}	**1.295943**$^{+*}$
02	1.650002	1.440909^{+}	**1.282538**$^{+*}$
03	1.494799	1.286604^{+}	**1.239142**$^{+}$
04	1.278690	1.094087^{+}	**1.052021**$^{+}$
05	1.185883	0.975802^{+}	**0.890545**$^{+*}$
06	1.218708	1.020362^{+}	**0.873065**$^{+*}$
07	1.045157	0.888445^{+}	**0.808010**$^{+*}$
08	1.385299	1.258866^{+}	**1.140813**$^{+*}$
09	1.521296	1.334855^{+}	**1.229168**$^{+*}$
10	1.126722	0.990774^{+}	**0.883844**$^{+*}$

Table 5.3: Results on Taillard's QAP Instance Sets

(a) tai45e

Instance	Baseline	Plain	Macro
01	6.580404	3.086954^{+}	**1.595798**$^{+}$
02	5.339964	5.189194	**3.429459**
03	7.010209	2.943039^{+}	**1.300211**$^{+*}$
04	6.868685	5.049172	**2.592011**$^{+*}$
05	3.449316	3.267493	**2.063026**
06	6.064243	2.135901^{+}	**1.969049**$^{+}$
07	4.913718	2.534868^{+}	**1.453027**$^{+}$
08	8.285509	2.772993^{+}	**2.239460**$^{+}$
09	6.608827	3.470362^{+}	**1.746826**$^{+}$
10	4.267595	3.492093	**1.391387**$^{+*}$

(b) tai75e

Instance	Baseline	Plain	Macro
01	19.109029	15.763880^{+}	**11.862988**$^{+*}$
02	20.694830	18.774176^{+}	**14.406250**$^{+*}$
03	18.232165	14.948470^{+}	**13.158483**$^{+}$
04	17.519799	15.020942	**9.616173**$^{+*}$
05	18.173579	14.491593^{+}	**12.517651**$^{+}$
06	19.500507	16.992233^{+}	**12.266506**$^{+*}$
07	18.260845	14.754167^{+}	**10.938063**$^{+*}$
08	22.025588	17.240534^{+}	**14.025829**$^{+*}$
09	16.053041	13.710570^{+}	**11.782226**$^{+}$
10	16.564989	11.761283^{+}	**9.898807**$^{+}$

from 10 to 20.) As for the comparison between the proposed approach and the approach that uses plain models, we can observe a similar trend: the advantage also seems to increase as the problem becomes larger or harder. Furthermore, although the plain approach gives a better ARPD on one of the 50 instances, the difference for this instance is not statistically significant.

The results of the experiments on the QAP domain are shown in Table 5.3. These results are also represented in ARPDs. In the QAP domain, we observe the same phenomenon as in the FS domain: overall, the proposed approach always gives a better result, and for the larger problem instances (i.e., instances in tai75e) the differences between the proposed approach and the plain approach are generally significant.

Table 5.4 shows the results of the experiments on the BP domain. In this case, problem instances from all four instance sets are of the same size: they are all one-dimensional bin packing problems containing 500 pieces. The difference lies in how the sizes of the pieces are distributed. As described in [17], these instance sets were created by drawing pieces from different pairs of Gaussian distributions. Note that for this domain, the results are shown in objective values instead of ARPDs because we cannot find a published source of best-known values.

As shown in the tables, both the plain approach and the proposed approach are better than the baseline approach with statistical significance. Comparing the proposed approach with the plain approach, we can see that for most of the problem instances, the proposed

Table 5.4: Results on Burke et al.'s BP Instance Sets

(a) testdual4

Instance	Baseline	Plain	Macro
00	0.089207	0.069133^{+}	**0.068139**$^{+*}$
01	0.091488	0.070113^{+}	**0.069213**$^{+*}$
02	0.093708	0.072988^{+}	**0.071486**$^{+*}$
03	0.090633	0.070524^{+}	**0.069611**$^{+*}$
04	0.089375	0.069370^{+}	**0.068516**$^{+*}$
05	0.092950	0.072249^{+}	**0.071397**$^{+*}$
06	0.094640	0.072804^{+}	**0.071660**$^{+*}$
07	0.097589	0.077429^{+}	**0.076264**$^{+*}$
08	0.090515	0.071128^{+}	**0.070128**$^{+*}$
09	0.089263	0.069609^{+}	**0.068092**$^{+*}$

(b) testdual7

Instance	Baseline	Plain	Macro
00	0.028948	0.022164^{+}	**0.020675**$^{+*}$
01	0.028869	0.022314^{+}	**0.020459**$^{+*}$
02	0.029878	0.022866^{+}	**0.021361**$^{+*}$
03	0.029103	0.021700^{+}	**0.020662**$^{+*}$
04	0.029866	0.022316^{+}	**0.021595**$^{+}$
05	0.030726	0.023608^{+}	**0.021421**$^{+*}$
06	0.029945	0.022211^{+}	**0.021128**$^{+*}$
07	0.032096	0.024515^{+}	**0.022694**$^{+*}$
08	0.030181	0.022947^{+}	**0.021105**$^{+*}$
09	0.030498	0.024017^{+}	**0.022329**$^{+*}$

(c) testdual8

Instance	Baseline	Plain	Macro
00	0.097940	0.073250^{+}	**0.072815**$^{+}$
01	0.095974	0.072190^{+}	**0.070945**$^{+*}$
02	0.096373	0.073214^{+}	**0.072088**$^{+*}$
03	0.094882	0.070756^{+}	**0.070599**$^{+}$
04	0.094714	0.069339^{+}	**0.069100**$^{+}$
05	0.099978	0.073889^{+}	**0.073201**$^{+}$
06	0.094166	0.070222^{+}	**0.069058**$^{+*}$
07	0.093902	0.071990^{+}	**0.070911**$^{+*}$
08	0.095113	0.068894^{+}	**0.068639**$^{+}$
09	0.097508	0.071883^{+}	**0.071384**$^{+}$

(d) testdual11

Instance	Baseline	Plain	Macro
00	0.036997	0.030537^{+}	**0.027129**$^{+*}$
01	0.034632	0.027890^{+}	**0.025346**$^{+*}$
02	0.035710	0.028108^{+}	**0.025693**$^{+*}$
03	0.035079	0.028418^{+}	**0.025837**$^{+*}$
04	0.036217	0.028120^{+}	**0.027041**$^{+*}$
05	0.035668	0.027843^{+}	**0.025871**$^{+*}$
06	0.036229	0.028726^{+}	**0.025967**$^{+*}$
07	0.037245	0.028298^{+}	**0.027081**$^{+*}$
08	0.034261	0.027077^{+}	**0.024494**$^{+*}$
09	0.035490	0.028428^{+}	**0.026406**$^{+*}$

approach offered statistically better results. However, for the tesetdual8 instance set, the significance seems to drop. We suspect that this is because the problem instances from this instance set are relatively easier to optimize because the pieces are drawn from a pair of Gaussian distributions that have means and standard deviations of (50, 10) and (35, 5). This makes the overall distribution of the 500 pieces look more unimodal than that of the problem instances from the other instance sets.

5.4 Discussion

In the previous section, we evaluated whether a learned $\mathcal{H}$ policy offers any speed-up over baseline methods, and furthermore, whether explicitly expressing the potentially useful collaboration patterns (i.e., what we called *heuristic macros*) offers any advantages. Note that the aim of our experiments was not to solve the combinatorial optimization problems to their state-of-the-art objective values since the proposed technique was fundamentally constrained by the set of heuristics provided within HyFlex. Rather our intent is simply to test whether the proposed learning method can be effective in the proposed setting. However, we would like to point out that by adding the state-of-the-art neighborhood-based heuristics into the algorithm (i.e., by adding them into the set of heuristics H), we can incorporate their power and perhaps achieve an improvement over the state of the art.

Also note that the above experiments were conducted with a cross-validation setup in which a learned policy was evaluated using problem instance that it has never been trained on. This ensures that the improvement that we observed was due to having learned generalizable patterns. On the other hand, if we test the learned policy on problem instances that it has been trained on, we would not be able to make such an assertion.

5.5 Conclusions

In this chapter, we developed a technique that allows us to automate the task of building the policies for choosing heuristics. This technique distills potentially useful patterns of interactions among heuristics, which are represented as a set of *heuristic macros* (i.e. concatenations of several heuristics), along with an estimate for the frequency of using each heuristic macro. The empirical results on three problem domains have shown that the proposed approach is effective and has an advantage over the baseline methods.

Chapter 6

A Study on Combining Multiple Sampling-based Heuristics

In this chapter, we present a study on combining multiple *sampling-based heuristics*. The main characteristic of this class of heuristics is that the process of forming a new candidate solution is based on sampling some explicit probabilistic model that represents our tendency to explore certain regions of the search space. In general, with a sampling-based heuristic, the process of forming a new candidate solution is multi-step: At each step, we will sample the heuristic's probabilistic model to obtain a segment of the solution, and this sampling can be conditioned on other segments obtained in prior steps. The steps proceed until we obtain a full solution. And for the purpose of this research, we can further categorize the sampling-based heuristics by whether they use handcrafted probabilistic models or probabilistic models that were estimated from some data source. In this chapter, we will develop a general technique that is capable of combining both heuristics that rely on handcrafted probabilistic models and heuristics that use estimated probabilistic models.

To briefly recap, the idea of using handcrafted probabilistic models has been considered before in other search frameworks such as Bresina's HBSS [9] and Cicirello and Smith's VBSS [25]. These frameworks use probabilistic models that are backed by some handcrafted heuristic functions. To sample a solution segment, such a heuristic function assigns a value to each possible candidate for that segment, and then we transform that value to be the probability associated with selecting that particular candidate.

As for heuristics that use estimated probabilistic models, in this chapter, we will use the

Estimation of Distribution Algorithms (EDAs) framework [60, 68, 83, 84] as our template for creating these kind of heuristics. In general, an EDA maintains a population of solutions, which is then used as the set of data points on which to estimate the probabilistic model. To explore the search space, an EDA samples the resulting probabilistic model to generate new candidate solutions, and subsequently updates the population with the new solutions. A more detailed review can be found in Section 2.6.

In this chapter, we will develop our techniques using sequencing and routing problems as the demonstrative problem domain. To provide some context, we begin with a review on previous works of EDAs that targeted at this domain.

6.1 Background

As mentioned in Section 2.6, most of the EDA studies have focused on domains in which a solution can be naturally represented as a fixed-length string with no ordering dependencies (that is, identical to the conventional representations used by the Genetic Algorithm, e.g. bit-string encoding of solutions.) However, many interesting and important combinatorial optimization problems require other types of solution representations. The domain of sequencing and routing provides canonical examples of such problems. In this domain, the objective is to search for an optimal ordering of a set of items or to search for an optimized sequence for performing a given set of operations. One classical example of this kind of problem is the Traveling Salesman Problem (TSP). The objective of the TSP is to find the shortest route for a traveling salesman who is on the mission to visit every city on a given list precisely once and then return to the initial city. The problem is equivalent to finding the Hamiltonian cycle that has the smallest cost in a complete weighted graph. The TSP is celebrated because many scientific and engineering problems can be formulated as TSPs and it has long been used to study sequencing and routing problems. In this chapter, we will use the TSP as our model problem for illustration and evaluation purposes.

Some previous works can be found in the literature that deal with sequencing and routing problems by means of EDAs. However, most of these approaches are direct adaptations of EDAs designed for discrete or continuous problems that have no ordering properties. Earliest attempts [89] applied discrete EDAs, such as Univariate Marginal Distribution Algorithms (UMDA) [78], Mutual Information Maximization for Input Clustering (MIMIC) [27] and Estimation of Bayesian Network Algorithm (EBNA) [35], as if a solution has no sequential dependencies. The obvious drawback is that the information of relative ordering among items is not explicitly considered in the constructed models. This deficiency may be the cause of low success rate in finding the global optimum as reported in [89, 100, 101].

Adaptation of continuous EDAs [67, 89] has also been explored. Most of the research in this direction uses the random keys representation [5]. With this scheme, a solution is represented as a real-valued vector, in which the i-th item is associated with the i-th value in the vector. To convert a real-valued vector into an ordering of the items, one sorts the items according to their associated values. In this way, some of the information about the relative ordering of the items can be encoded in the probabilistic model. Nevertheless, with this type of construct, an algorithm has to search for solutions in a largely redundant real-valued space. This inefficiency is reflected in their relatively inferior performance in the review by [20].

The limitations of these direct adaptations of EDAs designed for problems without ordering properties has encouraged the EDA community to invent other approaches that specifically target sequencing problems [7, 8, 100, 101]. More relevant to this research is the work done by Tsutsui et al. [101]. They proposed an approach called Edge Histogram Based Sampling Algorithm (EHBSA) [100], which constructs an edge histogram matrix by counting the number of occurrences that item i and item j appear consecutively in the sequences. For TSP, this is how many times the link between the i-th and j-th city is observed in the promising solutions. Based on these statistics, a probabilistic model is estimated that gives conditional probability of the next item given the previous one. This approach is equivalent to estimating a bigram model from the current population of good solutions. In this chapter, we will look at sampling-based heuristics that estimate and sample generalized n-gram models.

Although this generalization seems straightforward, as we will show empirically, the naive approach of increasing the order of the model (e.g., using trigram instead of bigram) does not work. Instead, we will develop a collaboration-based approach that utilizes multiple probabilistic models of different orders. The specific technique is to combine multiple models in the form of a linear interpolation (in which each model has an associated weight), and use a holdout set to estimate the weight associated with each model. In this way, the search can gradually shift the emphasis from a low-order model to higher order ones as longer patterns emerge in the population. Furthermore, as shown in Section 6.5, this technique is also general enough to incorporate heuristics with handcrafted probabilistic models.

In the next section, we will describe the formulation of the n-gram models. After that, Section 6.3 introduces our sampling-based heuristics derived from the EDA framework that use n-gram models for guiding the search process. It also discusses the difficulty encountered when moving from the bigram model to higher-order models. In Section 6.4, we present a method that is able to combine multiple models of different orders, and thus provides a smooth transition from a lower-order model to higher-order ones. Section 6.5 further demonstrates how a heuristic with a handcrafted probabilistic model can be easily incorporated using the same technique. In Section 6.6, we will give some performance comparisons showing the advantage of using a combination of multiple models compared to a single model approach and also more traditional evolutionary algorithms. Finally, Section 6.8 summarizes this chapter.

6.2 Modeling Sequences with n-gram Statistics

An n-gram is a pattern of n consecutive items, which is usually a segment from a longer sequence. Such a construct is often used in the tasks of modeling statistical properties of sequences, especially in the field of natural language processing (NLP). For example, a classic task in NLP is to predict the next word given the previous words. Such a task can be stated as attempting to estimate the conditional probability of observing some item w_i as the next item given the history of items seen so far. The n-gram approach to this estimate is to make a Markov assumption that only prior local context—the last few items—affects the next item. More formally, we are interested in estimating

$$P(W_i = w_i | W_{i-n+1} = w_{i-n+1}, \ldots, W_{i-1} = w_{i-1})$$

where the sequence $w_1, w_2, \cdots$ is some instantiation of a sequence of random variables $W_1, W_2 \cdots$. In the following, we will use $P(w_i|w_{i-n+1} \cdots w_{i-1})$ as a shorthand for this probability function.

The obvious first answer to the above formulation is to suggest using a *maximum likelihood estimate* (MLE):

$$P_{\text{MLE}}(w_i|w_{i-n+1}\cdots w_{i-1}) = \frac{C(w_{i-n+1}\cdots w_{i-1}w_i)}{\sum_{v\in\mathbf{V}} C(w_{i-n+1}\cdots w_{i-1}v)}$$

where $C(w_{i-n+1}\cdots w_{i-1}w_i)$ is the frequency of a certain n-gram in the training samples, and $\mathbf{V}$ is the set of possible items. However, a drawback is that MLE assigns a zero probability to unseen events, which effectively zeros out the probability of sequences with component n-grams that just happened not appearing in the training samples. For our scenario, this creates a risk of arbitrarily discarding some portion of the unexplored search space. Thus, we need a more suitable estimator that takes previously unseen patterns into consideration.

A simple solution to this problem is to smooth the distribution with some *pseudocount* κ:

$$P_{\kappa}(w_i|w_{i-n+1}\cdots w_{i-1}) = \frac{C(w_{i-n+1}\cdots w_{i-1}w_i)+\kappa}{\sum_{v\in\mathbf{V}} \left(C(w_{i-n+1}\cdots w_{i-1}v)+\kappa\right)}$$

where κ is usually set to a value smaller than 1. In this work, we use this simple method to allocate probability mass for unobserved events, though more sophisticated estimators are possible for this task.

6.3 Heuristics That Use Estimated n-gram Models

As mentioned previously, in this study, we will use the EDA framework to derive our sampling-based heuristics. The main idea is that by following the EDA framework, we can derive heuristics that use *estimated* probabilistic models, as opposed to handcrafted probabilistic models. Specifically, we will work on sampling-based heuristics that use estimated n-gram models.

To briefly review the operations performed by an EDA: At each iteration, we start with a set of promising solutions. The algorithm then constructs a probabilistic model based on statistics gathered from those solutions. Once a model is obtained, a number of new solutions will be generated by sampling the model to replace solutions in the current pop-

Algorithm 11 General Procedure of an EDA

```
Initialize a population $\mathbf{P}$ with a set of solutions.
Evaluate the solutions in $\mathbf{P}$.
$t \leftarrow 1$.
while the stopping criterion is not met do
    $M_t \leftarrow$ build a probabilistic model based on promising solutions in $\mathbf{P}$.
    $\mathbf{S}_t \leftarrow$ sample $M_t$ to generate new candidate solutions
    Evaluate the solutions in $\mathbf{S}_t$.
    Incorporate $\mathbf{S}_t$ into $\mathbf{P}$ using some replacement strategy.
    $t \leftarrow t + 1$.
end while
```

ulation according to some replacement strategy. This replacement strategy is designed to guide the growth of the average solution quality of the population. This general procedure of EDA is summarized in Algorithm 11.

In this work, instead of generating an entire solution anew, we first take an existing solution from the current population and randomly choose two points on the solution string to extract a subsequence from that solution. This segment will then be taken as the first part of the new solution and serve as the "history" on which the further sampling is based. This kind of *partial sampling* technique has been used by previous researchers such as [23, 24] and [101], achieving better usage of diversity and resulting in significant improvement in performance. For our purpose, this has an additional benefit of providing a convenient basis to initialize the sampling from the n-gram models.

Once we have the first part of the new solution, the remainder of the solution is generated by repeatedly sampling the n-gram model, with previous $n-1$ items as the history. In order to produce a valid solution, the set of possible items $\mathbf{V}$ may be varied as the sampling goes on. For example, when dealing with the TSPs, the set of possible next cities $\mathbf{V}$ has to be altered to exclude cities that have already been included in the constructed partial solution.

As for the replacement strategy, we compare the quality of a newly generated solution with the solution from which we took its segment for initialization. The better one of the two is selected and placed into the next population. This strategy has a good effect in preserving the diversity of the population and thus helps avoiding premature convergence. In this work, we use replacement as the sole means for selecting promising solutions, i.e.

better solutions are preserved under the replacement process. This is similar to the mechanism used by the evolution strategies [6], in which every solution in the current population is seen as a potentially good solution because they have survived previous replacement competitions.

To summarize the overall flow of the algorithm: At each iteration, we slide a window of size n through each solution in the current population to obtain the frequency counts of n-gram patterns. These statistics are then used for estimating an n-gram model in the form of a conditional distribution. To generate a new solution, we use partial sampling on an existing solution in the current population. Each solution in the current population is visited once for this sampling. Following each partial sampling, a replacement competition is held between the new solution and the solution from which that new solution's starting segment was extracted.

As a first step, we examine the performance of using a bigram model

$$P_{\text{2G}}(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i) + \kappa}{\sum_{v\in\mathbf{V}}\left(C(w_{i-1}v) + \kappa\right)}$$

for solving a 48-city TSP instance, `gr48`, taken from TSPLIB[1] [86]. Let ℓ denote the problem size. In this experiment, the population size N is set to 5ℓ, the pseudocount is set to $\kappa = 0.01$, and the termination criterion is when either the optimal tour is found or when the algorithm reaches 50ℓ iterations. We ran the algorithm 30 times to observe the average performance. The result of the experiment is presented in Table 6.1. It shows the success rate in finding the optimal tour and the average number of objective function evaluations used by the algorithm among the successful runs. Here, we use the number of times that an algorithm invokes the objective function to evaluate a solution as a measure of how much computational resource was consumed by the algorithm. This measure is commonly used in research concerning GAs and EDAs, in which the implicit assumption is that in order to generalize the results, evaluating the objective function should be treated as the most resource intensive bottleneck because it can correspond to some complex computation such as running a simulation.

From Table 6.1, we can see that the bigram approach gives a pretty decent performance.

[1]`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

Table 6.1: Observations when solving `gr48`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	277024	34535.4
3G	9/30	224640	118490.2
2G $\xrightarrow{\text{800 iter.}}$ 3G	30/30	240032	20753.3

It shows a high success rate in finding the global optimum, and a reasonable usage of function evaluations.

A tempting thought to proceed is to increase the order of the model. For example, instead of using bigram, we could use a trigram model

$$P_{3G}(w_i|w_{i-2}w_{i-1}) = \frac{C(w_{i-2}w_{i-1}w_i) + \kappa}{\sum_{v\in\mathbf{V}} (C(w_{i-2}w_{i-1}v) + \kappa)}$$

for learning the patterns. However, as shown in Table 6.1, this results in a significant drop in success rate. Our explanation is that at early stage of a run, there are not so many long patterns that are of good quality. If we attempt to use a higher-order model to learn longer patterns when there are none, we will end up encoding mediocre patterns into the model. Thus, a better way to proceed may be to use a low-order model like bigram model at the beginning of a run and switch to higher-order ones after longer patterns have emerged in the population. This will correspond to a form of *collaboration* among heuristics. More specifically, the heuristics are exchanging information using the population of solutions as a shared memory.

To provide some empirical support for this conjecture, we performed another experiment with the following modification to the algorithm: it starts out with using a bigram model, then it switches to using trigram model after 800 iterations. As shown in the third row of Table 6.1, the success rate returns to the same level as using the bigram model and it shows some improvement on function evaluations over the bigram approach. It seems that we can use this technique to gradually move to higher-order models.

However, choosing an appropriate schedule to make such switches is a nontrivial task. To address this issue and avoid having to choose a fixed switching point to elevate to a higher-order model, we propose an approach that estimates multiple n-gram models of different orders and combines those models into one composite model. The method automatically calibrates the degree of emphasis placed on each n-gram model. The detail of our formulation is presented in the next section.

6.4 Combining Multiple Models with Linear Interpolation

Specifically, we formulate the synthesis of multiple models as a linear combination of distributions

$$P(w_i|\mathbf{h}_i) = \sum_j \lambda_j P_j(w_i|\mathbf{h}_i) \tag{6.1}$$

where $\mathbf{h}_i$ represents the history of items we have seen so far, j is the index to a particular model, and λ_j is the weight associated with the j-th model such that $\lambda_j > 0$ and $\sum_j \lambda_j = 1$. A combination of bigram and trigram model will be

$$P_{\text{2G+3G}}(w_i|\mathbf{h}_i) = \lambda_{\text{2G}} P_{\text{2G}}(w_i|w_{i-1}) + \lambda_{\text{3G}} P_{\text{3G}}(w_i|w_{i-2}w_{i-1})$$

Assuming that we have K models that we want to combine together, for this formulation, we have to determine K weights, $\lambda_1, \lambda_2, \ldots, \lambda_K$, one associated with each model. In order to do this, we reserve a portion of the population for the task of estimating appropriate values for those λ_j's. Suppose that there are M items in such a holdout set for which we can give conditional probabilities. For each model P_j, we create a probability stream $\mathbf{p}_j = (p_{j1}, p_{j2}, \ldots, p_{jM})$ where p_{ji} is the probability of item w_i predicted by the model P_j, i.e., $p_{ji} = P_j(w_i|\mathbf{h}_i)$. These K probability streams (each of length M) are then used as the input to Algorithm 12 [55]. The resulting λ_j's are the best fit with respect to those input probability streams, i.e., these are the weights which optimize the average likelihood with respect to this holdout set. For simplicity, we use $\max_j |\lambda_j^{(t)} - \lambda_j^{(t-1)}| < 0.001$ as the condition to terminate Algorithm 12. In all our experiments, the invocations of Algorithm 12 terminated within just a few iterations (usually less than 10), and didn't seem to vary as we incorporate more models. Thus, the computation requirement for finding the weights in this way only grows linearly with M and similarly, linear with K as we need to prepare the probability stream of each model. Note that M is usually small (a tenth of the population

Algorithm 12 Estimating the Weights λ_j's

```
Input: A set of probability streams $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_K\}$,
  such that each $\mathbf{p}_j = (p_{j1}, p_{j2}, \ldots, p_{jM})$ is of length $M$.

  Initialize $\mathbf{\Lambda}^{(0)} = \{\lambda_1^{(0)}, \lambda_2^{(0)}, \ldots, \lambda_K^{(0)}\}$ s.t. each $\lambda_j^{(0)} > 0$ and $\sum_{j=1}^{K} \lambda_j^{(0)} = 1$
  repeat from $t = 0$
      For $j = 1 \ldots K$, update $\lambda_j$ using $\lambda_j^{(t+1)} = \frac{1}{M} \sum_{i=1}^{M} \frac{\lambda_j^{(t)} p_{ji}}{\sum_{k=1}^{K} \lambda_k^{(t)} p_{ki}}$
      $t = t + 1$
  until the difference between $\mathbf{\Lambda}^{(t)}$ and $\mathbf{\Lambda}^{(t-1)}$ is small
  return $\mathbf{\Lambda}^{(t)}$
```

Table 6.2: Parameter Settings

(a) For problem size $\ell < 70$

Parameters	Value
population size	$N = 5\ell$
pseudocount	$\kappa = 0.01$
size of holdout set	$R = \lfloor \frac{N}{10} \rfloor$
max iterations	50ℓ

(b) For problem size $\ell >= 70$

Parameters	Value
population size	$N = 5\ell$
pseudocount	$\kappa = 0.001$
size of holdout set	$R = \lfloor \frac{N}{10} \rfloor$
max iterations	80ℓ

size in our experiments), so this extra computation is quite manageable.

To illustrate the search behavior, Fig. 6.1 shows the variation of weights in a typical run that uses a combination of the bigram and trigram models. The weight for the bigram model starts out with a high value and gradually decreases. On the other hand, the weight of the trigram model will begin to dominate in later part of the search, meaning that we do more and more sampling with the trigram model. Based on this self-adaptive behavior, which adjusts the weights automatically, we call our proposal the "evolving mixture."

To evaluate the effectiveness of our proposal, we performed a set of experiments on ten TSP instances from TSPLIB. In the following, we will denote the problem size (the number of cities) as ℓ. The parameters to the algorithms are listed in Table 6.2. As before, the termination criterion is when either the optimal tour is found or when an algorithm reaches maximal number of iterations. Again, we ran each algorithm included for comparison for 30 times to give the average performance. The outcomes are presented in Table 6.3. In each table, we listed the success rate of each algorithm and its average usage of function evaluations among the successful runs. Note that the trailing number in each instance's name represents the number of cities in that instance.

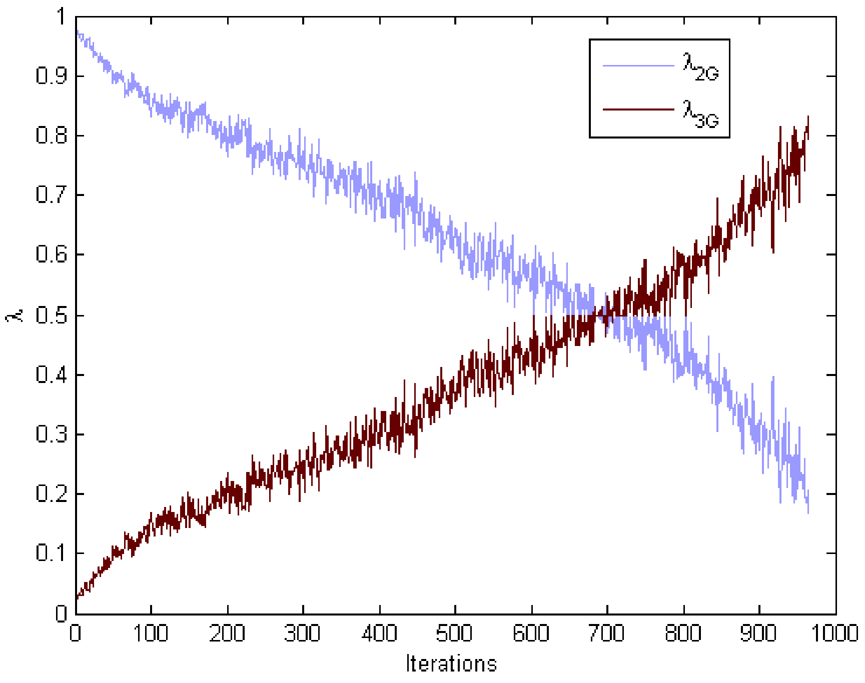


Figure 6.1: Typical variation of weights associated with bigram and trigram model. This illustration is from a run on `gr48` with population size $N = 450$ and 30% of the population as holdout set.

The result of using the evolving mixture to combine bigram and trigram is listed in the third row of each table. Comparing to the trigram approach (second row of the table), it gives a significantly better success rate, which is at the same level of the bigram approach. On the other hand, when compared with the bigram approach, the evolving mixture approach generally uses less function evaluations on average.

6.5 Incorporating Heuristics That Use Handcrafted Probabilistic Models

The previous experiments have demonstrated that using the evolving mixture to combine two n-gram models delivered a good result. It shows some improvement over the bigram

Table 6.3: Experiment Results

(a) `ulysses16`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	5461.3	902.4
3G	23/30	7207.0	10552.4
2G+3G	30/30	5040.0	1250.3
2G+3G+DH	30/30	4597.3	973.0
DH	30/30	17725.3	12070.3

(b) `gr24`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	18436.0	2356.7
3G	25/30	16574.4	11117.6
2G+3G	30/30	16492.0	2325.1
2G+3G+DH	30/30	11432.0	1717.7
DH	30/30	13340.0	6265.2

(c) `bay29`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	36332.0	3914.3
3G	22/30	40336.4	43365.6
2G+3G	30/30	31218.5	3409.1
2G+3G+DH	29/30	22785.0	3251.0
DH	19/30	107063.4	44051.6

(d) `att48`

Method	Success Rate	# of Evaluations	
		mean	std
2G	27/30	298053.3	54469.6
3G	10/30	196176.0	127370.7
2G+3G	26/30	215021.5	18175.5
2G+3G+DH	30/30	111544.0	10272.6
DH	28/30	265637.1	90849.8

(e) `gr48`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	277024.0	34535.4
3G	9/30	224640.0	118490.2
2G+3G	30/30	230048.0	23985.8
2G+3G+DH	30/30	154984.0	20243.5
DH	0/30	N/A	N/A

(f) `eil51`

Method	Success Rate	# of Evaluations	
		mean	std
2G	19/30	493572.6	73781.0
3G	2/30	199665.0	39308.1
2G+3G	21/30	391182.1	90013.1
2G+3G+DH	29/30	217031.4	52819.1
DH	8/30	448513.1	157609.4

(g) `berlin52`

Method	Success Rate	# of Evaluations	
		mean	std
2G	29/30	264276.6	74339.8
3G	9/30	320348.9	237082.3
2G+3G	29/30	217171.7	26798.3
2G+3G+DH	30/30	113906.0	9854.6
DH	30/30	180882.0	77478.7

(h) `st70`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	1426938.3	201534.3
3G	12/30	554983.3	69040.0
2G+3G	29/30	930444.8	76827.8
2G+3G+DH	30/30	298841.7	16886.3
DH	0/30	N/A	N/A

(i) `kroA100`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	3484900.0	208750.7
3G	7/30	2068357.1	233884.2
2G+3G	30/30	2773483.3	107248.1
2G+3G+DH	30/30	650783.3	46476.1
DH	0/30	N/A	N/A

(j) `lin105`

Method	Success Rate	# of Evaluations	
		mean	std
2G	30/30	3772142.5	230364.5
3G	3/30	1984325.0	154667.8
2G+3G	28/30	2844131.3	119353.2
2G+3G+DH	30/30	572197.5	19629.8
DH	0/30	N/A	N/A

approach in the usage of function evaluations. And it does not suffer low success rate as the trigram approach does. As we will show in this section, this technique is actually more versatile than what we have just demonstrated in the previous section.

In its formulation of Eq. (6.1), we did not put a restriction on the type of the models that can be included. It does not have to be an n-gram model for the evolving mixture to work. The only constraint is that the model takes the form of a compatible probability distribution so that we can evaluate its corresponding estimate of $P(w_i|\mathbf{h}_i)$. Thus, we can even incorporate sampling-based heuristics that use handcrafted probabilistic models without a change in the overall process.

For example, if we want to incorporate a distance-based heuristic for TSP into the search mechanism, we could do so by crafting an "artificial distribution"

$$P_{\text{DH}}(w_i|w_{i-1}) = \frac{d(w_{i-1}, w_i)^{-10}}{\sum_{v \in \mathbf{V}} d(w_{i-1}, v)^{-10}}$$

where $d(u, v)$ is the distance between city u and city v. This formula assigns a probability mass to each remaining city according to the distance between that city and the previous one in the partial tour constructed so far. This basically says that the shorter the link between two cities, the more likely that link will be adopted in the solution.

To see the effect of incorporating such a heuristic, we performed experiments on the same set of TSP instances as previous section. The results are presented in the fourth rows of Table 6.3. It can be observed that the performance improves significantly. Comparing to using only n-gram models, it uses far less function evaluations, especially for larger instances, while retaining a high success rate.

For completeness, we also include the results of using solely the distance-based heuristic for updating the population. For more than half of the tested instances, this heuristic alone does not provide a satisfying success rate. It seems that the effectiveness of relying solely on this distance-based heuristic will depend on the property of that particular TSP instance. Nevertheless, when combined with bigram and trigram model, this tends to give a great performance boost to the algorithm.

The results also yield some interesting insight into how the algorithm utilizes the provided heuristics. For example, Figure 6.2 shows the shifts of weights among three distribu-

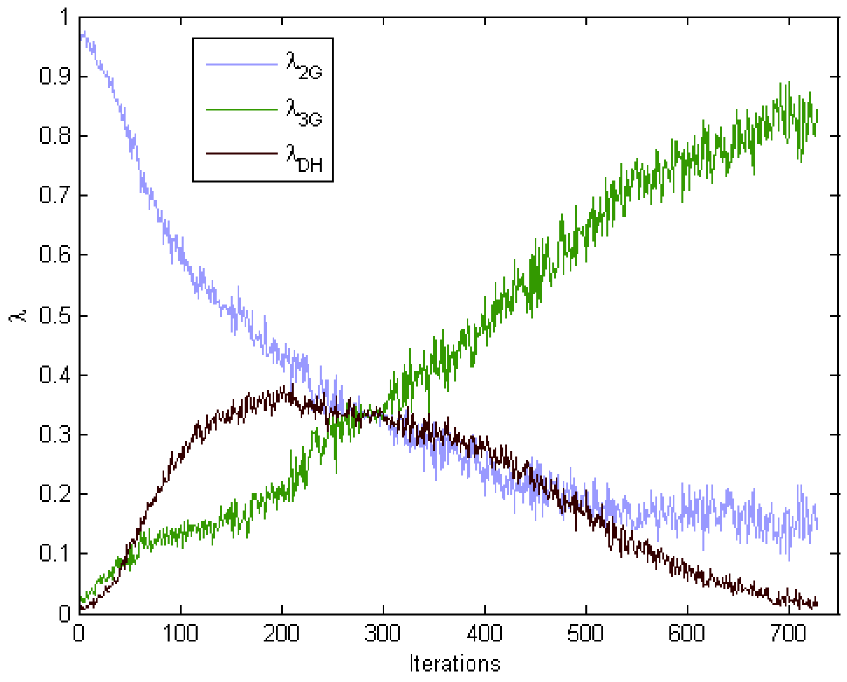


Figure 6.2: Typical variation of weights from a run using a combination of bigram, trigram models and distance heuristics. This illustration was generated from a run on `gr48` with population size 450 and using 30% of the population as the holdout set.

tions when solving `gr48`. It illustrates how the evolving mixture adjusts the emphasis on different n-gram models and the distance heuristic at different stages of the process. This adjustment is dynamically determined based on the promising solutions in the holdout set. We believe this dynamic behavior can lead to a more flexible search strategy and easier integration of multiple heuristics. This can also be extended beyond solving TSPs, to other types of problems as well, where we have different sampling-based heuristics available for guiding the search.

6.6 Performance Comparison with Other Approaches

This section offers some performance comparisons between our method and other population-based approaches. We are especially interested in comparing our results to Tsutsui's EHBSA [100]. Note that in the review by [20], a variant of EHBSA, EHBSA-WT, gave the best empirical performance on TSPs comparing to 13 other EDAs for permutation-based problems. In addition to EHBSA, we also include several well-known classical GAs for sequencing problems: OX [81], eER [95] and PMX [40]. These are traditional GAs with crossover operators tailored for manipulating permutations of items.

To have a meaningful comparsion with EHBSA-WT[2] we adopt the parameter settings used in [101] for running our algorithms. Parameters specific to our approach (pseudocount and the size of the holdout set) are still set according to Table 6.2. The results of our methods along with the data taken from [101] are listed in Table 6.4. From these results, several observations can be made. The first thing to note is that the traditional GAs are not very good at finding the global optima. Except for the smallest problem tested here, the traditional GAs' success rates are significantly lower than EHBSA-WT and our approaches. For `gr24`, eER seems to gain a decent increase in success rate when the population size N is increased from 60 to 240. However, the other two traditional GAs were still having trouble in obtaining high success rate, even when the population size was increased to fourfold. This inadequacy is more pronounced when we move to larger problems. We can see that for `gr48` and `pr76`, the success rates of OX, eER and PMX are not ideal even

[2]More specifically, we compare our approach to the results of configuration EHBSA-WT2, which shows a more stable performance overall. Also it is more similar to our proposal in the way of doing partial sampling.

when population size is increased to eight times larger. On the other hand, EHBSA-WT and our methods delivered pretty high success rates in all three problems, while using a fairly small population size.

To further compare our methods with EHBSA-WT, we should turn our attention to the differences between them in the number of function evaluations used. As we can see in Table 6.4, our approaches perform better than EHBSA-WT, both in the average number of function evaluations used and in having smaller standard deviations. The difference is especially prominent when we compare it with the combination of bigram, trigram and distance-based heuristic (2G+3G+DH, the last row of each table.) We can see that this combination spent only about half of the function evaluations required by EHBSA-WT, and it also has much smaller standard deviations. We think this result indicates that our proposal offers an improvement over the traditional GAs and EHBSA.

6.7 A Framework for Combining Multiple Sampling-based Heuristics

In this chapter, we demonstrated a method that combines multiple sampling-based heuristics. Algorithmically, there are two interesting pieces that are crucial for the operations of this approach. We will discuss them in the following, and based on the discussion, we further distill a framework for combining sampling-based heuristics.

The first core algorithmic component is that our method maintains a population of good solutions. This population provides an important function: It serves as a communication medium among heuristics. To see how this works, first note that each solution being kept in the population contains some good solution segments discovered by some heuristics, and subsequently, these good solution segments can be learned by other heuristics that estimate their probabilistic models based on the population. In this way, we allow knowledge that originally resides in one heuristic to be propagated to other heuristics. For instance, in our TSP case, some solution segments discovered by sampling the distance-based heuristic can be subsequently learned by the bigram and trigram heuristics if a solution bearing these segments is kept in the population.

The second core algorithmic component that supports our approach is the adoption of

Table 6.4: Performance Comparisons

(a) Solving `gr24`

Method	N	Success Rate	# of Evaluations mean	# of Evaluations std
OX	60	0/10	N/A	N/A
OX	240	4/10	34140	3793
eER	60	1/10	4738	0
eER	240	10/10	13394	1726
PMX	60	0/10	N/A	N/A
PMX	240	2/10	23191	1798
EHBSA-WT	60	10/10	10713	7419
2G+3G	60	10/10	9522	1739
2G+3G+DH	60	10/10	5982	983

(b) Solving `gr48`

Method	N	Success Rate	# of Evaluations mean	# of Evaluations std
OX	120	0/10	N/A	N/A
OX	960	2/10	287852	6706
eER	120	0/10	N/A	N/A
eER	960	5/10	166286	4932
PMX	120	0/10	N/A	N/A
PMX	960	0/10	N/A	N/A
EHBSA-WT	120	10/10	144032	29115
2G+3G	120	10/10	131724	16748
2G+3G+DH	120	10/10	83508	17105

(c) Solving `pr76`

Method	N	Success Rate	# of Evaluations mean	# of Evaluations std
OX	120	0/10	N/A	N/A
OX	960	0/10	N/A	N/A
eER	120	0/10	N/A	N/A
eER	960	3/10	394887	22321
PMX	120	0/10	N/A	N/A
PMX	960	0/10	N/A	N/A
EHBSA-WT	120	9/10	457147	65821
2G+3G	120	10/10	405660	54893
2G+3G+DH	120	10/10	195960	28123

Algorithm 13 A Framework for Combining Sampling-based Heuristics

```
Require: a set of heuristics $H$, a policy $\mathcal{P}$ for maintaining a population of good solutions,
    and a policy $\mathcal{S}$ for selecting which heuristic to sample.
 1: $\mathbf{P} \leftarrow$ initial population of solutions.
 2: while stopping criteria not met do
 3:     $\mathbf{x} \leftarrow \phi$ or some initial partial solution.
 4:     while $\mathbf{x}$ is not a complete solution do
 5:         $h \leftarrow$ a heuristic selected from $H$ according to $\mathcal{S}$.
 6:         $y \leftarrow$ a solution segment obtained by sampling $h$ conditioned on $\mathbf{x}$.
 7:         $\mathbf{x} \leftarrow \mathbf{x} \cup \{y\}$
 8:     end while
 9:     Update $\mathbf{P}$ with $\mathbf{x}$ using policy $\mathcal{P}$.
10:     Update the parameters of policy $\mathcal{S}$. (Optional)
11: end while
```

a parameterizable policy for selecting heuristics. This allows us to adjust the policy for selecting heuristics based on a reserved portion of the population. It also relates to the notion described above that the population can be seen as a storage for keeping what have been discovered so that we can adjust algorithm's behavior based on that.

Now having identified the above two algorithmic components, we can further formulate a framework for combining multiple sampling-based heuristics that provides a generalization over the approach described earlier in this chapter. As shown in Algorithm 13, this framework takes as input a set of sampling-based heuristics H, a policy $\mathcal{P}$ for maintaining a population, and a policy $\mathcal{S}$ for selecting which heuristic from H to sample. For each iteration, we will construct a new candidate solution $\mathbf{x}$ by sampling a series of heuristics selected by the policy $\mathcal{S}$. This new candidate solution is then incorporated into the population according to the policy $\mathcal{P}$. For example, in our approach described earlier in this chapter, the policy $\mathcal{P}$ corresponds to the rule that the new candidate solution will compete with its parent solution, and if it has a better objective value, then it will replace the parent solution's spot in the population. Finally, we can also update the parameters of the policy $\mathcal{S}$ to adjust its selection behavior. This can happen at each iteration or at some interval like our approach described earlier, which only updates once per $|\mathbf{P}|$ iterations. Also note that we mark line 10 of Algorithm 13 as optional. In general, the policy $\mathcal{S}$ can be a fixed selection policy that provides no adjustable parameters. In that case, the behavior of $\mathcal{S}$ is fixed throughout the run, as opposed to our approach described earlier, which dynamically adjusting $\mathcal{S}$'s behavior as the population updates with new solutions.

6.8 Summary

In this chapter, we examined a way of using multiple sampling-based heuristics, and based on that, extracted a generalized framework. We begun with looking at sampling-based heuristics that use estimated n-gram models. In this case, in order to provide a smooth transition from lower-order models to higher-order ones, we proposed using a linear interpolation for combining multiple probabilistic models. The weights associated with those models are estimated automatically from a reserved portion of the population of good solutions. Then we also showed that this method can as well be used to incorporate sampling-based heuristics that use handcrafted probabilistic models.

Qualitative inspection of the experimental results obtained also provides some support for our intuition that different heuristics may be more suitable than others at different stages of a run. For example, Figure 6.2 shows the shifts of weights among three distributions in a run for solving `gr48`. It illustrates how the proposed technique adjusts the emphasis toward using different heuristics at different stages of the process, and this adjustment is done dynamically based on the good solutions retained in the holdout set. As a further illustration, Figure 6.3 shows the weight shifts when solving `st70`. Compared to Figure 6.2, we can observe that a different degree of utilization of the distance heuristic was adopted in this case. We believe this discrepancy was due to the condition that the search bias provided by the distance heuristic was more appropriate in one case than in the other. Since this kind of knowledge of the degree of appropriateness of the search bias may be unavailable for most cases, we argue that our approach of automatically adjusting the emphasis on each heuristic can potentially bring convenience to the process. In the next chapter, we will explore this aspect in more detail through a series of experiments.

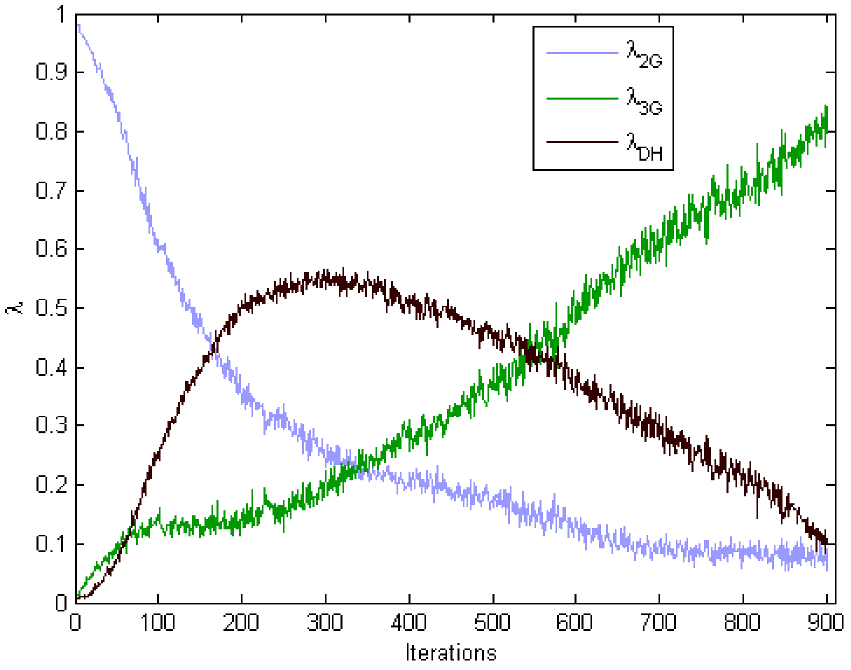


Figure 6.3: Typical variation of weights with P_{2G}, P_{3G} and P_{DH}. This illustration was generated from a run on `st70` with population size 500 and using 30% of the population as the holdout set.

Chapter 7

A Deeper Look at the Dynamic Combinations of Sampling-based Heuristics

In the previous chapter, we looked at the results of using a linear interpolation to combine multiple sampling-based heuristics. Compared to using only a single sampling-based heuristic (which uses a bigram, trigram, or handcrafted model), this approach demonstrated a significantly better performance. Now, having established that combining multiple sampling-based heuristics is worthwhile, we turn attention to examination of our design decision of adopting an automated procedure for estimating the weight associated with each sampling-based heuristics.

This design decision is worth further investigation for a couple of reasons:

1. An obvious alternative is to manually assign weights to the constituent heuristics and hold them fixed throughout the run. And although we demonstrated the convenience of adopting an automated procedure for estimating the weights of the mixture, we didn't offer a comparison to this simple alternative.
2. In order to automatically estimate the weights assigned to the heuristics, we need to reserve a portion of the population of solutions for the calibration procedure. This may cause the overall algorithm to require more computational resources, and we provided no justification as to whether this is actually a worthwhile tradeoff to

make.

Finally, since our approach re-estimates the weights at each iteration, we are interested in whether this dynamic adjustment will compare favorably to a static weight assignment[1]. For example, given that we formulate the combination as

$$P_{\text{2G+3G+DH}}(w_i|\mathbf{h}_i) = \lambda_{\text{2G}} P_{\text{2G}}(w_i|w_{i-1}) + \lambda_{\text{3G}} P_{\text{3G}}(w_i|w_{i-2}w_{i-1}) + \lambda_{\text{DH}} P_{\text{DH}}(w_i|w_{i-1})$$

we would like to know the advantage of using a holdout set to re-estimate the weights for each iteration, compared to using a static combination, which fixes the λ's to some values throughout a run. Obviously, using holdout sets alleviates the users from having to decide the values of the weights. However, one may wonder whether it is the case that merely setting the weights to some fixed values will provide an adequate performance in most cases[2]. Thus, in this chapter, we investigate how easy or hard is it to pick an adequate set of parameters for a fixed mixture, and how the performance of this approach compares to our approach of using dynamic adjustment.

7.1 Comparisons Arbitrarily Picking Static Weights

To have a preliminary idea of the level of difficulty of picking a good set of weights, as a first step, we simulate the situation of having no knowledge or experience on how to set the parameters of the mixture. So for static combination, we randomly choose values for the λ's and use these fixed values throughout a run. This experiment was repeated 100 times on `kroA100` and `lin105`, each time with a different set of λ values.

The results are listed in Table 7.1, along with the results of using dynamic adjustment (also repeated 100 times.) As shown in the table, our approach outperforms this static combination approach. It uses a lower number of function evaluations on average, and the hypothesis testing also indicates the significance of this observation (μ_{fix} and μ_{dyn} denote the average number of function evaluations spent by the static combination and dynamic adjustment approaches, respectively.) This fact, along with high standard deviations ob-

[1]In terms of the framework we laid out as Algorithm 13, using a static weight assignment corresponds to skipping the update at line 10 of Algorithm 13.

[2]This assumption relates to the widespread success of Ant Colony Optimizations (ACOs), which fix the parameters of combination. We will offer some discussion on this in a later section.

Table 7.1: Static Combination vs. Dynamic Adjustment on $(\lambda_{2G}, \lambda_{3G}, \lambda_{DH})$-Mixture

(a) Solving `kroA100`

$(\lambda_{2G}, \lambda_{3G}, \lambda_{DH})$	Success Rate	# of Evaluations mean (μ)	std (σ)
Random & Fixed	100/100	1091580.0	404080.7
Dynamic	100/100	656555.0	46400.7

$*$ t-test on null hypothesis: $\mu_{\text{fix}} \le \mu_{\text{dyn}}$ using $\alpha = 0.05$
$\Rightarrow$ reject null hypothesis, $p = 1.2\text{E-}18$

(b) Solving `lin105`

$(\lambda_{2G}, \lambda_{3G}, \lambda_{DH})$	Success Rate	# of Evaluations mean (μ)	std (σ)
Random & Fixed	99/100	1019210.6	463858.2
Dynamic	100/100	573525.7	21389.2

$*$ t-test on null hypothesis: $\mu_{\text{fix}} \le \mu_{\text{dyn}}$ using $\alpha = 0.05$
$\Rightarrow$ reject null hypothesis, $p = 5.5\text{E-}16$

Table 7.2: Proportions of Static Combination Runs Thresholded by Function Evaluations

TSP	$> \mu_{\text{dyn}}$	$> \mu_{\text{dyn}} + \sigma_{\text{dyn}}$	$> \mu_{\text{dyn}} + 2\sigma_{\text{dyn}}$
`kroA100`	91/100	88/100	81/100
`lin105`	89/100	86/100	85/100

served in Table 7.1, suggests that we can easily get significantly inferior performance in terms of function evaluations if we don't choose the λ's carefully for a static combination.

To see how frequently this can happen, Table 7.2 shows how many times out of the 100 runs that the number of function evaluations spent by a static combination run is higher than the average number of function evaluations spent by the dynamic adjustment. The numbers are listed in the "$> \mu_{\text{dyn}}$" column. In the same way, we also list the number of runs exceeding more than one standard deviation "$> \mu_{\text{dyn}} + \sigma_{\text{dyn}}$" and more than two standard deviations "$> \mu_{\text{dyn}} + 2\sigma_{\text{dyn}}$". From this table, we can see that only about ten percents of the runs that use static weights achieved an average level of performance of the dynamic adjustment. And most of the runs that use static weights are distributed at least two standard deviations away from the average performance of dynamic adjustment. This shows that the range of appropriate weights can be quite small, and this also reinforces our belief that using an estimation procedure for calibrating the weights associated with each heuristic can be a lot easier than manually picking an appropriate set of weights.

7.2 Comparisons to Tuned Static Weights

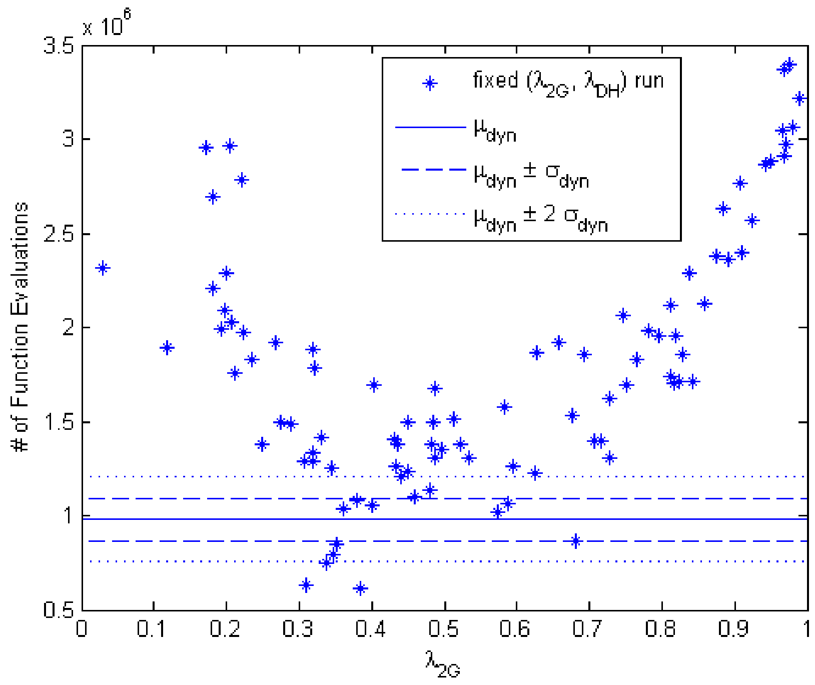
In this section, we compare our dynamic adjustment approach to tuned static weights. Obviously, tuning λ's for the mixture of three heuristics considered above is not very easy, considering we have to tweak three values simultaneously. To further our discussion, we simplify the mixture to

$$P_{\text{2G+DH}}(w_i|\mathbf{h}_i) = \lambda_{\text{2G}} P_{\text{2G}}(w_i|w_{i-1}) + \lambda_{\text{DH}} P_{\text{DH}}(w_i|w_{i-1})$$

to provide a basis for more systematic testing, despite the fact that this form of the mixture loses some advantages provided by the trigram model. In this case, we only have to vary one of the parameters, say λ_{2G}, and the other will be $1 - \lambda_{\text{2G}}$, since they have to sum to one.

To gain some initial perspective, we first repeated the experiment of randomly sampling weights as done above for this simplified form. Figure 7.1 shows the plots of λ_{2G} versus the number of function evaluations used, from the successful runs. As can be seen from the graphs, the better range of the parameters is between 0.3 and 0.7. Based on this, we

(a) Solving `kroA100`



(b) Solving `lin105`

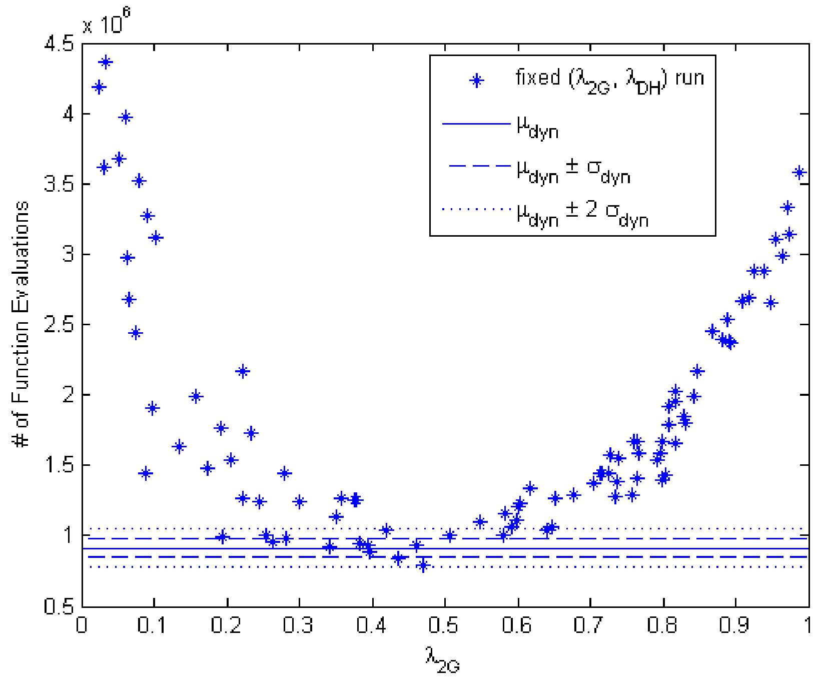


Figure 7.1: Static Combination vs. Dynamic Adjustment on ($\lambda_{2G}, \lambda_{DH}$)-Mixture

Table 7.3: Parameter Sweep on Static $(\lambda_{\text{2G}}, \lambda_{\text{DH}})$-Mixture vs. Dynamic Adjustment

(a) Solving `kroA100`

$(\lambda_{\text{2G}}, \lambda_{\text{DH}})$	Success Rate	# of Evaluations	
		mean (μ)	std (σ)
(0.3, 0.7)	30/30	1503733.3	363151.3
(0.4, 0.6)	30/30	1360650.0	337795.9
(0.5, 0.5)	30/30	1165400.0	210292.8
(0.6, 0.4)	30/30	1389116.6	246623.9
(0.7, 0.3)	30/30	1432833.3	213297.1
Dynamic	30/30	981233.3	113005.4

(b) Solving `lin105`

$(\lambda_{\text{2G}}, \lambda_{\text{DH}})$	Success Rate	# of Evaluations	
		mean	std
(0.3, 0.7)	30/30	1123027.5	210353.0
(0.4, 0.6)	30/30	994367.5	161215.8
(0.5, 0.5)	30/30	993142.5	135707.8
(0.6, 0.4)	30/30	1146705.0	125444.0
(0.7, 0.3)	30/30	1310610.0	131794.1
Dynamic	30/30	913955.0	69474.3

systematically varied the λ_{2G} from 0.3 to 0.7 (and hence the λ_{DH} from 0.7 to 0.3.) The results are listed in Table 7.3 . It can be observed that even when comparing to the best choice of λ's among these configurations, our approach of using dynamic adjustment still compares favorably to the static combination. This provides an evidence that although the dynamic adjustment approach requires a portion of the population of solutions reserved for the purpose of estimating the weights associated with the heuristics, it does not incur an excess overhead (or in the case of these problems, none at all) in terms of the number of objective function evaluations[3] compared to using static weight assignments. Furthermore, the dynamic adjustment approach shows a much smaller standard deviation comparing to

[3]As mentioned before, we use the number of objective function evaluations as the measure of how much computational resources was used by the algorithm. This measure is commonly used in research concerning evolutionary algorithms, in which the implicit assumption is that in order to generalize the results, evaluating the object function should be treated as the bottleneck because it can involve some complex computation such as running a simulation.

all $(\lambda_{2G}, \lambda_{DH})$ configurations of the static combination. We think this smaller standard deviation shows that using a reserved set for re-estimating the weights at each iteration offers a more stable performance and a much better average behavior.

Note that although for both `kroA100` and `lin105`, $(\lambda_{2G}, \lambda_{DH}) = (0.5, 0.5)$ seems to be the best choice for a static combination, we think that there may be some situations in which (0.5, 0.5) is not ideal. For example, it might be the case that the distance heuristic is less informative for a particular problem instance. This can happen for TSPs when the distances between cities are more uniform, thus the distance heuristic will not provide as effective a guidance. Taking this to an extreme, we can consider the case where the provided heuristic model is largely misleading. In this case, the sensible action would be to set the corresponding λ to zero or near zero. However, there might be no easy way to assess the quality of an externally supplied heuristic model except by doing some preliminary trial runs. The problem gets even more complicated when we move to adopting more than two sampling-based heuristics. For example, consider the case of combining bigram, trigram and distance heuristics that we experimented with in the previous chapter. In this case, setting a set of equal weights to these three heuristics will not be ideal since at the beginning of a run, trigram model will not be very effective (as we have shown in the previous chapter.) Thus, we believe our method offers some advantage in this aspect.

7.3 Contrast to Ant Colony Optimization: Difference and Implications for Combining Heuristics

Readers who are familiar with the Ant Colony Optimization (ACO, [29]) might relate ACO to our approach in the aspect that both combine multiple sources of information. To briefly recap, the canonical ACO uses the following form for constructing a model for sampling new solutions:

$$p_{xy} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{y' \in \{\text{allowed } y'\}}(\tau_{xy'}^{\alpha})(\eta_{xy'}^{\beta})}$$

where p_{xy} stands for the probability of choosing y as the next state given the previous state is x, τ_{xy} represents the amount of "pheromone" deposited in the transition from state x to state y, and η_{xy} represents the desirability of state transition of x to y (for TSP, this usually corresponds to the inverse of the distance between x and y, and hence it is roughly equivalent to our distance-based heuristic). For the above form, α and β are the parameters that control the relative influences of τ_{xy} and η_{xy} respectively.

Comparing to our approach, one should first notice the difference in the form of the combination: we use a linear interpolation to combine components while ACO uses a product of component values (which is then converted to probability using a normalization term.) The benefit of adopting our linear form is that we then have a procedure for estimating the weights associated with heuristics automatically. On the other hand, ACO's product form makes it difficult to adopt a similar estimation procedure for tuning its parameters (i.e. α and β in the above form.) Thus, ACO usually holds those parameters constant throughout a run.

As mentioned in the previous sections, we think that using a dynamic adjustment procedure can offer several advantages in terms of convenience and robustness. Furthermore, in order to have a good performance, ACO requires the user to manually pick a good set of parameters, which might not be an easy task if we relate that to our earlier experiments.

Furthermore, because our approach automatically adjusts the weight associated with each component heuristic, in the case that we have an ACO with properly tuned parameters, we can just incorporate the ACO into our mixture without any change in the overall algorithm.

7.4 Conclusions

In this chapter, we performed a set of experiments to evaluate the benefits of adopting our dynamic approach to adjust the weights associated with various component heuristics in the case of sampling-based search. Specifically, we compared this approach to a couple of schemes for establishing a good set of static weight assignments. From the results of the experiments, we believe that this approach offers several advantages in terms of convenience and robustness.

In terms of convenience, the experiments showed that adopting our automated procedure for estimating the mixture weights does not increase the number of objective function evaluations. And we also gave a preliminary assessment on the likelihood of manually picking a good set of weights.

In terms of robustness, we showed that using the dynamic adjustment of weights gives a more consistent behavior: Comparing to runs that use static weight assignments, we observed that adopting dynamic weight adjustments generally results in a much smaller standard deviation in the number of objective function evaluations used.

As future work, we would also like to apply our approach to other sequencing and routing problems. Problems such as the Generalized Traveling Salesman Problem [94] and the Sequential Ordering Problem [34] may be good extensions to this work. We are also very interested in problems like the Orienteering Problem [102] and the TSP with profit [36], for which the length of solution representation is not fixed. For some heuristic approaches such as traditional EAs, it is less convenient to work with variable length representation. However, because our approach composes a new solution sequentially, it may have an advantage in these kinds of problems.

Chapter 8

Conclusions

In this thesis, we investigated the subject of how to combine multiple heuristics. The setting that we considered is to use existing heuristics as algorithmic components and combine them in an automatic fashion. Particular emphasis was given to the collaborative aspect of utilizing multiple heuristics and showed how this can be achieved for the cases that we studied. In the sections below, we summarize the principal contributions of this thesis toward mechanisms for collaborative problem solving with multiple heuristics. We start with the case of combining multiple neighborhood-based heuristics and then follow on to the case of combining multiple sampling-based heuristics.

8.1 Combining Multiple Neighborhood-based Heuristics

In Chapter 3, we described an architecture that allows us to chain multiple heuristics in a pipelined fashion. Schematically, this architecture is an algorithmic framework that has two user-defined policy components. To derive a concrete algorithm from this framework, one needs to plug in 1) a policy $\mathcal{H}$ for selecting heuristics, and 2) a policy $\mathcal{L}$ for choosing the length of the pipeline that chains the selected heuristics.

We laid out this framework in order to further discuss the subject of how to combine multiple neighborhood-based heuristics in a more principled manner: For the $\mathcal{L}$ component, we first offered a theoretical discussion on its design in Chapter 3 and described a policy based on Luby's sequence that has an asymptotic guarantee. This policy is then used as the default $\mathcal{L}$ policy for the rest of this research.

For the $\mathcal{H}$ component, we first looked at a baseline policy $\mathcal{H}_u$, that each time selects a heuristic uniformly at random from the set of heuristics. Based on that, in Chapter 4, we experimented with three simple learning mechanisms that change the search algorithm's behavior for choosing heuristics during a run. We first showed that even in the case of just using a simple pruning strategy, we can achieve a relatively high rank in the CHeSC benchmark that is used in the hyper-heuristics research community. Following that, we demonstrated that by learning the frequency of applying each heuristic, we can further boost the performance, and our method of learning the bigram statistics, i.e. modeling $P(h_i|h_{i-1})$, was shown to rank better in the CHeSC benchmark than any of the original competitors.

In Chapter 5, we further developed a policy construction procedure which can produce policies that have explicit collaboration patterns embedded in them. In addition to that, in Chapter 5, we also adopted a setting that has a distributional assumption over the problem instances, which enabled us to evaluate the learned policies with a more rigorous cross-validation assessment. As future works, we would like to study other forms for expressing the potentially useful collaboration patterns among heuristics, and more importantly, how to distill these patterns automatically from past experience.

Another direction for future research is to develop a way of using the baseline scheme presented in Chapter 3 as an analysis tool for gaining information about the optimization problem. For example, using the idea that we can run the baseline algorithm and collect a log of sequences that are chains of heuristics which had led to improving solutions, we can get some idea about the characteristics of the landscape of the objective function: We can look at the proportion of perturbation heuristics in the log to have a coarse understanding of the roughness of the landscape (and if we have more detailed information about how much perturbation each heuristic induces, we can gain even more information.) On the other hand, we can also look at the lengths of the sequences to determine whether good solutions are clustered in the landscape.

Finally, we are also curious about whether we can extend the proposed framework to a population-based search scheme. For example, what are the criteria for effective use of crossover operators and how to carry the anytime property that we desire to a population-based algorithm.

8.2 Combining Multiple Sampling-based Heuristics

In the second part of this thesis, we studied the problem of combining multiple sampling-based heuristics. In this case, each component heuristic uses a probabilistic model for sampling candidate solutions. As mentioned in Section 2.5, the underlying probabilistic models can be either handcrafted or estimated from some data source.

In Chapter 6, we proposed a method for combining heuristics of this type. We begun by looking at sampling-based heuristics that use estimated n-gram models. In this case, in order to provide a smooth transition from lower-order model to higher-order ones, we proposed using a linear interpolation for combining multiple probabilistic models. The weights associated with those models are estimated automatically from a reserved portion of the population of good solutions, and this weight estimation is done repeatedly from iteration to iteration to provide a dynamic adjustment. Furthermore, we also showed that this method can as well be used for incorporating sampling-based heuristics that use handcrafted probabilistic models. The experiments demonstrated that using the proposed integration mechanism, we can achieve better results than are possible using any individual component heuristic exclusively. Furthermore, using this mechanism, we obtained a more stable performance as evidenced by the generally much smaller standard deviations.

Two distinguishing characteristics of the proposed technique are that 1) the weights associated with each sampling-based heuristics are automatically estimated based on a reserved portion of the population of good solutions (instead of manually assigned), and 2) these weights are re-estimatd at each iteration to provide a dynamic adjustment. In Chapter 7, we looked further into these characteristics by offering comparisons to other alternatives. From the results of the experiments, we believe that our approach offers several advantages in terms of convenience and robustness. In terms of convenience, our experiments showed that adopting our automated procedure for estimating the mixture weights does not increase the number of objective function evaluations. And we also gave an assessment of the likelihood of manually picking a good set of weights. In terms of robustness, we showed that using the dynamic adjustment of weights gives a more consistent behavior, as evidenced by generally much smaller standard deviations.

Based on an analysis that identifies two crucial algorithmic components of the proposed method, i.e. maintaining a population of solutions and adopting a parameterizable policy for selecting heuristics, we also formulated a more general framework for combining sampling-based heuristics in Section 6.7. This framework can serve as a skeleton for creating other methods that combine multiple sampling-based heuristics.

As a future research direction, we are especially interested in the possibility that we can plug in additional heuristics to enhance the search mechanism. Of course, this idea itself is nothing new. Previous works such as ACOs have demonstrated the possible benefits of adopting this concept. However, here we presented a method that allows us to automatically determine the parameters associated with the heuristic incorporation. We think this can be very useful for certain types of scheduling problems for which multiple "dispatching rules" have been designed and studied.

Another potential future project is to study the effect of integrating local search into the mechanism. In practice, excellent results have been obtained with local search algorithms for a wide range of problems. We would like to see if there is some complementary effect from hybriding our technique with local search methods. We would also like to apply our approach to other sequencing and routing problems. Problems such as Generalized Traveling Salesman Problem [94] and Sequential Ordering Problem [34] may be good extensions to this work. Furthermore, we are very interested in problems like the Orienteering Problem [102] and the TSP with profit [36], for which the length of solution representation is not fixed. For some heuristic approaches such as traditional evolutionary algorithms, it is less convenient to work with variable length representations. However, because our approach can build up a solution sequentially, it may have an advantage in such kind of problems.

References

[1] Steven Adriaensen, Gabriela Ochoa, and Ann Nowé. A benchmark set extension and comparative study for the HyFlex framework. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pages 784–791. IEEE, 2015. 5.3.1

[2] Dario J Aloise, Daniel Aloise, Caroline TM Rocha, Celso C Ribeiro, José C Ribeiro Filho, and Luiz SS Moura. Scheduling workover rigs for onshore oil production. *Discrete Applied Mathematics*, 154(5):695–702, 2006. 2.4.3

[3] Rubén Armañanzas, Iñaki Inza, Roberto Santana, Yvan Saeys, Jose Luis Flores, Jose Antonio Lozano, Yves Van de Peer, Rosa Blanco, Víctor Robles, Concha Bielza, et al. A review of estimation of distribution algorithms in bioinformatics. *BioData mining*, 1(6):1–12, 2008. 2.6

[4] Shahriar Asta and Ender Özcan. A tensor-based selection hyper-heuristic for cross-domain heuristic search. *Information Sciences*, 299:412–432, 2015. 3.1, 3.2

[5] James C. Bean. Genetic algorithms and random keys for sequencing and optimization. *INFORMS Journal on Computing*, 6(2):154–160, 1994. 6.1

[6] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002. 6.3

[7] Peter A. N. Bosman and Dirk Thierens. Crossing the road to efficient IDEAs for permutation problems. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2001*, pages 219–226. Morgan Kaufmann, 2001. 6.1

[8] Peter A. N. Bosman and Dirk Thierens. Permutation optimization by iterated estimation of random keys marginal product factorizations. In *Parallel Problem Solving from Nature—PPSN VII*, pages 331–340. Springer, 2002. 6.1

[9] John L Bresina. Heuristic-biased stochastic sampling. In *AAAI/IAAI, Vol. 1*, pages 271–278, 1996. 2.5, 6

[10] Jack Brimberg, Pierre Hansen, Nenad Mladenović, and Eric D Taillard. Improvements and comparison of heuristics for solving the uncapacitated multisource weber problem. *Operations Research*, 48(3):444–460, 2000. 2.2, 2.4.3, 2.4.3

[11] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In Thomas Philip Runarsson, Hans-Georg Beyer, Edmund Burke, Juan J. Merelo-Guervós, L. Darrell Whitley, and Xin Yao, editors, *Parallel Problem Solving from Nature - PPSN IX: 9th International Conference, Reykjavik, Iceland, September 9-13, 2006, Proceedings*, pages 860–869. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 3.1

[12] Edmund K Burke and Yuri Bykov. The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78, 2017. 2.2

[13] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Barry McCollum, Gabriela Ochoa, Andrew J. Parkes, and Sanja Petrovic. The cross-domain heuristic search challenge – an international research competition. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, pages 631–634. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 3.2

[14] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *the Journal of the Operational Research Society*, 64, 2013. 1, 3.1

[15] Edmund K. Burke, Michel Gendreau, Gabriela Ochoa, and James D. Walker. Adaptive iterated local search for cross-domain optimisation. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1987–1994, New York, NY, USA, 2011. ACM. 3.2

[16] Edmund K. Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. A classification of hyper-heuristic approaches. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 449–468. Springer US, Boston, MA, 2010. 3.1

[17] Edmund K Burke, Matthew R Hyde, and Graham Kendall. Providing a memory mechanism to enhance the evolutionary design of heuristics. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010. 5.1, 5.3.3

[18] Suzana A Canuto, Mauricio GC Resende, and Celso C Ribeiro. Local search with perturbations for the prize-collecting steiner tree problem in graphs. *Networks*, 38(1):50–58, 2001. 2.4.3

[19] Gilles Caporossi, Dragoš Cvetković, Ivan Gutman, and Pierre Hansen. Variable neighborhood search for extremal graphs. 2. finding graphs with extremal energy. *Journal of Chemical Information and Computer Sciences*, 39(6):984–996, 1999. 2.4.3

[20] Josu Ceberio, Ekhine Irurozki, Alexander Mendiburu, and Jose A. Lozano. A review on estimation of distribution algorithms in permutation-based combinatorial optimization problems. *Progress in Artificial Intelligence*, 1(1):103–117, 2012. 6.1, 6.6

[21] Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics: Recent developments. In Carlos Cotta, Marc Sevaux, and Kenneth Sörensen, editors, *Adaptive and Multilevel Metaheuristics*, pages 3–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 3.1

[22] Chao-Hong Chen and Ying-ping Chen. Real-coded ECGA for economic dispatch. In *Proceedings of the 9th GECCO*, pages 1920–1927. ACM, 2007. 2.6

[23] Chung-Yao Chuang and Ying-ping Chen. On the effectiveness of distributions estimated by probabilistic model building. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 391–398. ACM, 2008. 6.3

[24] Chung-Yao Chuang and Ying-ping Chen. Sensibility of linkage information and effectiveness of estimated distributions. *Evolutionary Computation*, 18(4):547–579, 2010. 6.3

[25] Vincent A Cicirello and Stephen F Smith. Enhancing stochastic search performance by value-biased randomization of heuristics. *Journal of Heuristics*, 11(1):5–34, 2005. 2.5, 6

[26] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach

to scheduling a sales summit. In Edmund Burke and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III: Third International Conference, PATAT 2000 Konstanz, Germany, August 16–18, 2000 Selected Papers*, pages 176–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 3.1

[27] Jeremy S. De Bonet, Charles L. Isbell Jr., and Paul Viola. MIMIC: Finding optima by estimating probability densities. In *Advances in neural information processing systems*, pages 424–430. The MIT Press, 1996. 6.1

[28] Matthijs den Besten, Thomas Stützle, and Marco Dorigo. Design of iterated local search algorithms. In Egbert J. W. Boers, editor, *Applications of Evolutionary Computing: EvoWorkshops 2001: EvoCOP, EvoFlight, EvoIASP, EvoLearn, and EvoSTIM Como, Italy, April 18–20, 2001 Proceedings*, pages 441–451, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 2.4.2

[29] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997. 7.3

[30] Zvi Drezner, Peter M Hahn, and Éeric D Taillard. Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for metaheuristic methods. *Annals of Operations research*, 139(1):65–94, 2005. 5.1

[31] Els I. Ducheyne, Bernard De Baets, and Robert De Wulf. Probabilistic models for linkage learning in forest management. In *Knowledge incorporation in evolutionary computation*, pages 177–194. Springer, 2005. 2.6

[32] RW Eglese. Simulated annealing: a tool for operational research. *European journal of operational research*, 46(3):271–281, 1990. 2.4.2

[33] Agosten E. Eiben and James E. Smith. *Introduction to evolutionary computing.* Natural Computing Series. Springer Berlin, 2003. 2.6

[34] LF Escudero. An inexact algorithm for the sequential ordering problem. *European Journal of Operational Research*, 37(2):236–249, 1988. 7.4, 8.2

[35] Ramon Etxeberria and Pedro Larranaga. Global optimization using bayesian networks. In *Proceedings of the Second Symposium on Artificial Intelligence (CIMAF-*

99), pages 332–339, 1999. 6.1

[36] Dominique Feillet, Pierre Dejax, and Michel Gendreau. Traveling salesman problems with profits. *Transportation science*, 39(2):188–205, 2005. 7.4, 8.2

[37] Henry Fisher and Gerald L Thompson. Probabilistic learning combinations of local job-shop scheduling rules. *Industrial scheduling*, 3(2):225–251, 1963. 1

[38] Fred Glover. Tabu searchpart i. *ORSA Journal on computing*, 1(3):190–206, 1989. 2.2

[39] Fred Glover. Tabu searchpart ii. *ORSA Journal on computing*, 2(1):4–32, 1990. 2.2

[40] D. E. Goldberg and R. Lingle. Alleles, loci and the traveling salesman problem. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 154–159, 1985. 6.6

[41] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison Wesley, 1989. 2.6

[42] Carla P. Gomes and Bart Selman. Algorithm portfolio design: Theory vs. practice. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, UAI'97, pages 190–197, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. 1, 2.3

[43] Carla P Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001. 2.3

[44] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence. 2.3

[45] A Grosso, F Della Croce, and R Tadei. An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. *Operations Research Letters*, 32(1):68 – 72, 2004. 2.4.2

[46] Pierre Hansen and Nenad Mladenović. J-means: a new local search heuristic for minimum sum of squares clustering. *Pattern recognition*, 34(2):405–413, 2001. 2.2,

2.4.3, 2.4.3

[47] Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez. Variable neighborhood search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 61–86. Springer US, Boston, MA, 2010. 2.4.3

[48] Pierre Hansen, Nenad Mladenović, and José A Moreno Pérez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010. 2.4.3

[49] Pierre Hansen, Nenad Mladenović, and Dragan Urošević. Variable neighborhood search and local branching. *Computers & Operations Research*, 33(10):3034–3045, 2006. 2.4.3

[50] Alain Hertz, Matthieu Plumettaz, and Nicolas Zufferey. Variable space search for graph coloring. *Discrete Applied Mathematics*, 156(13):2551 – 2560, 2008. 2.2.2, 1

[51] Nhu Binh Ho and Joe Cing Tay. Evolving dispatching rules for solving the flexible job-shop problem. In *2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2848–2855 Vol. 3, Sept 2005. 3.1

[52] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975. 2.6

[53] Ping-Che Hsiao, Tsung-Che Chiang, and Li-Chen Fu. A vns-based hyper-heuristic with adaptive computational budget of local search. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012. 3.2

[54] Matthew Hyde, Gabriela Ochoa, T Curtois, and JA Vázquez-Rodríguez. A HyFlex module for the one dimensional bin-packing problem. Technical report, School of Computer Science, University of Nottingham, 2009. 5.3.1

[55] Frederick Jelinek and Robert L. Mercer. Interpolated estimation of markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice*, 1980. 6.4

[56] Ming-Yang Kao, Yuan Ma, Michael Sipser, and Yiqun Yin. Optimal constructions of hybrid algorithms. In *Proceedings of the Fifth Annual ACM-SIAM Symposium*

on Discrete Algorithms, SODA '94, pages 372–381, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics. 2.3

[57] Yuji Katsumata and Takao Terano. Cabling and scheduling for electric power plant operation via tabu-BOA algorithm. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation, CEC-2004*, volume 2, pages 1675–1682. IEEE, 2004. 2.6

[58] R. E. Keller and R. Poli. Linear genetic programming of parsimonious metaheuristics. In *2007 IEEE Congress on Evolutionary Computation*, pages 4508–4515, Sept 2007. 3.1

[59] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. 2.4.2

[60] Pedro Larrañaga and Jose A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation.* Kluwer Academic Publishers, Norwell, MA, USA, 2001. 2.6, 2.6, 6

[61] Andreas Lehrbaum and Nysret Musliu. A new hyperheuristic algorithm for cross-domain search problems. In *International Conference on Learning and Intelligent Optimization*, pages 437–442. Springer, 2012. 3.1

[62] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. Boosting as a metaphor for algorithm design. In *Principles and Practice of Constraint Programming–CP 2003*, pages 899–903. Springer, 2003. 2.3

[63] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated local search. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 320–353. Springer US, Boston, MA, 2003. 2.4.2, 7

[64] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated local search: Framework and applications. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 363–397. Springer US, Boston, MA, 2010. 2.2, 2.4.2, 2.4.2, 4.2

[65] Helena Ramalhinho Lourenço. Job-shop scheduling: Computational study of local search and large-step optimization methods. *European Journal of Operational Research*, 83(2):347 – 364, 1995. 2.4.2

[66] Helena Ramalhinho Lourenço and Michiel Zwijnenburg. Combining the large-step optimization with tabu-search: Application to the job-shop scheduling problem. In *Meta-Heuristics: Theory & Applications*, pages 219–236. Springer, 1996. 2.4.2

[67] J. A. Lozano and A. Mendiburu. Solving job scheduling with estimation of distribution algorithms. In *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*, pages 231–242. Kluwer Academic Publishers, Norwell, MA, USA, 2002. 6.1

[68] Jose A. Lozano, Pedro Larrañaga, Iñaki Inza, and Endika Bengoetxea. *Towards a New Evolutionary Computation: Advances on Estimation of Distribution Algorithms.* Studies in Fuzziness and Soft Computing. Springer-Verlag New York, 2006. 2.6, 6

[69] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, pages 128–133. IEEE, 1993. 2.3, 3.3.1, 3.3.2, 3.3.2

[70] Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999. 4.3

[71] Franco Mascia and Thomas Stützle. A non-adaptive stochastic local search algorithm for the chesc 2011 competition. In *International Conference on Learning and Intelligent Optimization*, pages 101–114. Springer, 2012. 3.1

[72] David Meignan. An evolutionary programming hyper-heuristic with co-evolution for chesc11. In *The 53rd Annual Conference of the UK Operational Research Society (OR53)*, 2011. 3.1

[73] Alexander Mendiburu, José Miguel-Alonso, Jose Antonio Lozano, Miren Ostra, and Carlos Ubide. Parallel EDAs to create multivariate calibration models for quantitative chemical applications. *Journal of Parallel and Distributed Computing*, 66(8):1002–1013, 2006. 2.6

[74] Peter Merz and Jutta Huhse. An iterated local search approach for finding provably good solutions for very large tsp instances. In Günter Rudolph, Thomas Jansen, Nicola Beume, Simon Lucas, and Carlo Poloni, editors, *Parallel Problem Solving from Nature – PPSN X: 10th International Conference, Dortmund, Germany, September 13-17, 2008. Proceedings*, pages 929–939. Springer Berlin Heidelberg, Berlin, Heidel-

berg, 2008. 2.4.2

[75] Mustafa Mısır, Katja Verbeeck, Patrick De Causmaecker, and Greet Vanden Berghe. An intelligent hyper-heuristic framework for chesc 2011. In *International Conference on Learning and Intelligent Optimization*, pages 461–466. Springer, 2012. 3.1

[76] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997. 2.4.3

[77] Nenad Mladenović, Frank Plastria, and Dragan Urošević. Reformulation descent applied to circle packing problems. *Computers & Operations Research*, 32(9):2419 – 2434, 2005. 2.2.2

[78] Heinz Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997. 6.1

[79] Mladen Nikolić, Filip Marić, and Predrag Janičić. Instance-based selection of policies for sat solvers. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 326–340, Berlin, Heidelberg, 2009. Springer-Verlag. 1, 2.3

[80] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A. Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J. Parkes, Sanja Petrovic, and Edmund K. Burke. Hyflex: A benchmark framework for cross-domain heuristic search. In Jin-Kao Hao and Martin Middendorf, editors, *Evolutionary Computation in Combinatorial Optimization: 12th European Conference, EvoCOP 2012, Málaga, Spain, April 11-13, 2012. Proceedings*, pages 136–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 3.4.1, 5.3.1

[81] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 224–230, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc. 6.6

[82] Ralph J. Olsson, Zoran Kapelan, and D. Savic. The design and rehabilitation of water distribution systems using the hierarchical bayesian optimisation algorithm. In *Proceedings of the World Environmental and Water Resources Congress*, 2007. 2.6

[83] Martin Pelikan, David E. Goldberg, and Fernando G. Lobo. A survey of optimization by building and using probabilistic models. *Computational optimization and applications*, 21(1):5–20, 2002. 2.6, 6

[84] Martin Pelikan, Kumara Sastry, and Erick Cantú-Paz. *Scalable optimization via probabilistic modeling: From algorithms to applications*, volume 33 of *Studies in Computational Intelligence*. Springer, 2006. 2.6, 2.6, 6

[85] Marek Petrik and Shlomo Zilberstein. Learning parallel portfolios of algorithms. *Annals of Mathematics and Artificial Intelligence*, 48(1):85–106, 2006. 2.3

[86] Gerhard Reinelt. TSPLIB–a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991. 6.3

[87] Stephen Remde, Peter Cowling, Keshav Dahal, Nic Colledge, and Evgeny Selensky. An empirical study of hyperheuristics for managing very large sets of low level heuristics. *Journal of the operational research society*, 63(3):392–405, 2012. 3.2

[88] Celso C Ribeiro and Maurıcio C Souza. Variable neighborhood search for the degree-constrained minimum spanning tree problem. *Discrete Applied Mathematics*, 118(1):43–54, 2002. 2.4.3

[89] V. Robles, P. de Miguel, and P. Larranaga. Solving the traveling salesman problem with EDAs. In *Estimation of Distribution Algorithms*, pages 211–229. Springer, 2002. 6.1

[90] Rubén Ruiz and Thomas Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033 – 2049, 2007. 2.4.2

[91] Kumara Sastry. Efficient cluster optimization using extended compact genetic algorithm with seeded population. In *Proceedings of the Optimization by Building and Using Probabilistic Models Workshop at the Genetic and Evolutionary Computation Conference (GECCO-2001 OBUPM)*, 2001. 2.6

[92] Tzur Sayag, Shai Fine, and Yishay Mansour. Combining multiple heuristics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 242–253. Springer, 2006. 2.3

[93] Georg R Schreiber and Olivier C Martin. Cut size statistics of graph bisection heuristics. *SIAM Journal on Optimization*, 10(1):231–251, 1999. 2.4.1

[94] SS Srivastava, Santosh Kumar, RC Garg, and P Sen. Generalized traveling salesman problem through n sets of nodes. *CORS Journal*, 7:97–101, 1969. 7.4, 8.2

[95] T. Starkweather, S. Mcdaniel, D. Whitley, K. Mathias, and C. Whitley. A comparison of genetic sequencing operators. In *Proceedings of the fourth International Conference on Genetic Algorithms*, 1991. 6.6

[96] Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *In Proceedings of the Twenty-Second Conference on Artificial Intelligence*, pages 1197–1203. AAAI Press, 2007. 1, 2.3

[97] Matthew Streeter and Stephen F. Smith. New techniques for algorithm portfolio design. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, pages 519–527, Arlington, Virginia, United States, 2008. AUAI Press. 2.3, 3

[98] Thomas Stützle. Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519 – 1539, 2006. 2.4.2

[99] Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993. 5.1

[100] Shigeyoshi Tsutsui. Probabilistic model-building genetic algorithms in permutation representation domain using edge histogram. In *Parallel Problem Solving from Nature—PPSN VII*, pages 224–233. Springer, 2002. 6.1, 6.6

[101] Shigeyoshi Tsutsui, Martin Pelikan, and David E. Goldberg. Using edge histogram models to solve permutation problems with probabilistic model-building genetic algorithms. Technical Report 2003022, 2003. 6.1, 6.3, 6.6

[102] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209(1):1–10, 2011. 7.4, 8.2

[103] José Antonio Vázquez-Rodrıguez, Gabriela Ochoa, Tim Curtois, and Matthew Hyde. A hyflex module for the permutation flow shop problem. Technical report, School of

Computer Science, University of Nottingham, 2009. 5.3.1

[104] James D Walker, Gabriela Ochoa, Michel Gendreau, and Edmund K Burke. Vehicle routing and adaptive iterated local search within the HyFlex hyper-heuristic framework. In *Learning and Intelligent Optimization*, pages 265–276. Springer, 2012. 5.3.1

[105] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1):565–606, June 2008. 1, 2.3