Proceedings of the Twelfth International
Symposium on Combinatorial Search (SoCS 2019)

Learning and Utilizing Interaction Patterns among Neighborhood-Based Heuristics

Chung-Yao Chuang
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
cychuang@cmu.edu

Stephen F. Smith
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
sfs@cs.cmu.edu

Abstract

This paper proposes a method for learning and utilizing potentially useful interaction patterns among neighborhood-based heuristics. It is built upon a previously proposed framework designed for facilitating the task of combining multiple neighborhood-based heuristics. Basically, an algorithm derived from this framework will operate by chaining the heuristics in a pipelined fashion. Conceptually, we can view this framework as an algorithmic template that contains two user-defined components: 1) the policy $\mathcal{H}$ for selecting heuristics, and 2) the policy $\mathcal{L}$ for choosing the length of the pipeline that chains the selected heuristics. In this paper, we develop a method that automatically derives a policy $\mathcal{H}$ by analyzing the experience collected from running a baseline algorithm. This analysis will distill potentially useful patterns of interactions among heuristics, and give an estimate for the frequency of using each pattern. The empirical results on three problem domains shows the effectiveness of the proposed approach.

1 Introduction

For many combinatorial optimization problems, a popular choice for heuristically finding a high quality solution is to use a local search procedure that combines multiple neighborhood-based operators, e.g. (Lourenço, Martin, and Stützle 2010; Hansen et al. 2019; Nowicki and Smutnicki 2005). It has been shown that this approach can be scalable and highly effective, and it has produced the best-known solutions for a number of problem domains. However, despite this approach's popularity, how to actually structure the application of constituent neighborhood-based heuristics or operators remains largely an engineering art. Frequently, they are put together in a very simplistic way: follow a fixed, pre-specified order of application. As such, an effort to generalize beyond this simplistic integration can potentially yield a more versatile technique.

In a recent work, Chuang and Smith (2018) have conceived a generalized 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 better utilize interactions 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 their report, Chuang and Smith (2018) offered a theoretical discussion on the design of the $\mathcal{L}$ component and described a policy that has an asymptotic guarantee. They also empirically showed that this policy can achieve promising results. Based on their work, in this paper, we further explore how to design a good $\mathcal{H}$ component.

Our proposal is based on the idea that we can first run the simple baseline algorithm proposed by Chuang and Smith (2018), and record the positive experiences encountered during the run. 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 good policy $\mathcal{H}$.

In the following, we will first provide some background and briefly describe the framework proposed by Chuang and Smith, as well as its connection to a field called hyper-heuristics (Burke et al. 2013).

1.1 Neighborhood-based Heuristics

In this paper, we investigate the task of combining multiple *neighborhood-based heuristics*. The key property of this class of heuristics that allows us to think about the possibility of combining them is the following: Given a solution $\mathbf{x}$ as input, a neighborhood-based heuristic will perform modifications on $\mathbf{x}$ to produce another solution $\mathbf{x}'$.

In general, the basic operations performed by this class of heuristics is to modify the current solution $\mathbf{x}$ slightly or to replace a portion of $\mathbf{x}$ with new values. These operations can be iterated multiple times to arrive at a more distant solution. To encapsulate this idea that we can obtain another solution by slightly altering the current solution, we can introduce a neighboring relation on the search space, i.e., $\mathbf{x}'$ is a neighbor of $\mathbf{x}$ if $\mathbf{x}'$ can be obtained from $\mathbf{x}$ by applying one of the modifications considered by the heuristic. We use $\mathbf{x} \rightsquigarrow \mathbf{x}'$ to denote such a neighboring relation. Based on this, we can further define the *neighborhood* of $\mathbf{x}$ as $\mathcal{N}(\mathbf{x}) = \{\mathbf{x}' | \mathbf{x} \rightsquigarrow \mathbf{x}'\}$. Alternatively, we can think of the

Algorithm 1 Basic Architecture for Combining Heuristics

```
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
```

mapping $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ (where $\mathcal{S}$ is the solution space) as a representation that encodes the set of modifications that the heuristic considers, and we call such an $\mathcal{N}$ a *neighborhood function* or a *neighborhood structure*.

Using the above concept, we can come up with a wide range of heuristics based on different neighborhood functions. And in addition to the neighborhood function, there are two more criteria for specifying a neighborhood-based heuristic: 1) the transition rule, and 2) the iterating condition of the process. For example, we can define a simple local search by making the following specification: The transition rule is to switch to the best solution in the neighborhood (on the premise that it is an improvement over the current solution.) And the iterating condition is to iterate until there is no further improvement transitions possible.

Based on the above three criteria (i.e. the neighborhood function, the transition rule, and the iterating condition), we can specify a class of heuristics that we characterize as the neighborhood-based heuristics. With this in mind, in the following, we briefly review Chuang and Smith's framework for combining multiple neighborhood-based heuristics.

1.2 Algorithmic Framework

As mentioned previously, the key attribute of neighborhood-based heuristics, which enables us to think about combining them, is that a neighborhood-based heuristic can pick up a complete solution and modify it to generate a new solution. Using this property, we can architect a procedure that chains the operations of multiple heuristics in a pipelined fashion. Chuang and Smith (2018) utilized this idea and outlined a simple algorithmic framework. Basically, an algorithm derived from this framework will operate 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. Thus, we can accumulate the modifications from multiple heuristics and explore the search space more diversely.

The description of this framework is shown in Algorithm 1. In this framework, we assume that we are given a set of heuristics H, along with two user-defined components: 1) a policy $\mathcal{L}$ for choosing the chain length, and 2) a policy $\mathcal{H}$ for choosing among the given heuristics. Following that, each iteration begins with deciding a positive integer ℓ according to the policy $\mathcal{L}$. This ℓ will bound the length of the next chain of heuristic applications. 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 1. If any solution encountered during this process is 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.

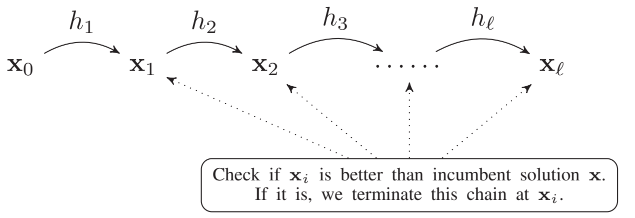
h_1
h_2
h_3
h_ℓ
$\mathbf{x}_0$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_\ell$
Check if $\mathbf{x}_i$ is better than incumbent solution $\mathbf{x}$.
If it is, we terminate this chain at $\mathbf{x}_i$.

Figure 1: The inner loop of Algorithm 1. It starts with a solution $\mathbf{x}_0$. Each subsequent 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.

Note that in most of the scenarios, we will have stochastic heuristics included in H (for example, heuristics that provide random perturbations in order to escape local optima), so even applying the same sequence of heuristics can potentially yield different solutions. Furthermore, since those solutions are generated stochastically, we can also think of the process depicted in Figure 1 as *sampling* a solution chain.

Following (Chuang and Smith 2018), in this work, we consider the situation that 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. With this setting, we are left with two user-defined components to specify to make Algorithm 1 functional: the policy $\mathcal{L}$ for choosing the chain lengths and the policy $\mathcal{H}$ for choosing the heuristics.

In their report, Chuang and Smith described a policy for the $\mathcal{L}$ component. This policy uses Luby's sequence (Luby, Sinclair, and Zuckerman 1993) for deciding the chain lengths. The idea is that we can pick ℓ sequentially according to the Luby's sequence, and jump back to the beginning of the Luby's sequence every time we replace the incumbent solution with a new best solution. The benefit of using this policy is that it offers a theoretic guarantee that it is asymptotically optimal in an agnostic scenario. In their experiments, they also showed that using this policy for the $\mathcal{L}$ component can yield a competitive result, even when coupled with a simple $\mathcal{H}$ component that selects heuristics uniformly at random. In this work, we will build upon their results and discuss how to construct a good $\mathcal{H}$ component from experience.

1.3 Hyper-heuristics

Conceptually, the above framework also relates to a research field called *hyper-heuristics* (Chakhlevitch and Cowling 2008; Burke et al. 2010; 2013). The term "hyper-heuristics" was first introduced as "heuristics to choose heuristics." However, this term was later also adopted by the genetic programming community to similarly refer to methods that assembles new search heuristics from a set of primitive operations. For the purpose of this paper, we will only make connections to the original conception of hyper-heuristics, which architecturally follow a two-level structure: At the lower level, there is a set of heuristics[1], and the top level corresponds to a mechanism that iteratively dispatches the lower-level heuristics in order to improve the quality of the current solution.

Operationally, most hyper-heuristics adhere to the following procedure: Given an initial solution (either generated randomly or heuristically), the hyper-heuristic goes through the steps of (1) selecting a heuristic from the set of provided heuristics, and (2) applying selected heuristic to the incumbent solution to generate a new solution, then finally (3) deciding whether the new solution should be accepted as the new incumbent solution. This process iterates until the termination conditions are met.

Note that the above description partially resembles the framework that we listed as Algorithm 1. The key difference is that in addition to having a component for selecting among heuristics, Algorithm 1 also has a component for choosing the length of the pipeline that chains the selected heuristics. In this way, we explicitly express the idea that we are structuring the exploration of the search space as a process that constructs many solution chains.

Also note that most of the recent hyper-heuristics use some sorts of learning mechanisms to boost their performance. One way to make a classification is to distinguish them by whether they use online learning or offline learning. An online learning hyper-heuristic adjusts itself based on the feedback received during the search process and dynamically adjusts its behavior. Offline learning, in contrast, takes place before the actual search starts. In this paper, we discuss the issue of how to construct a good $\mathcal{H}$ component for the framework that we described in the previous section. And the specific approach that we look into can be seen as an offline learning technique.

In the following section, we will first introduce a distributional assumption under which we developed our techniques. Based on this assumption, Section 3 will describe our proposal for an automated policy construction procedure. After that, in Section 4, we will show the results of the experiments, followed by a discussion in Section 5. Finally, Section 6 concludes this paper.

2 Distributional Assumption

In this work, we investigate the issue of how to design a policy for choosing among the given heuristics for application. In the algorithmic template that we described in the previous section, this component was denoted as $\mathcal{H}$. In their previous work, Chuang and Smith (2018) 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. 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[2]," 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 from which we drew the training set. 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.

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 $\mathcal{D}$. And 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 will introduce in the following section.

3 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 Chuang and Smith (2018) experimented in their previous report.) 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

[1]In most cases, the set of heuristics are what we have characterized as the neighborhood-based heuristics.

[2]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.

```
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 2: 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.

sequences. And because these sequences are the ones that had led to improving solutions, we think it is possible to construct a better policy through analyzing this log.

For example, suppose that by running the baseline algorithm, we collected a small log like the one shown in the Figure 2a, in which each number 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. 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 2b, then by a simple counting, we can estimate a probabilistic model that encodes these structures, such as the one shown in Figure 2c. 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[3].

However, it is not immediately obvious how to come up with such a segmentation without a manual intervention. As mentioned above, our goal is a fully automated procedure for constructing policies, so solving this issue is inevitable. Our idea to proceed is to recognize that if someone handed us a probabilistic model, then we will be able to divide a sequence into its most probable segmentation using a 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 a 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 2. It will take as input a model $\mathcal{M}$ of the same form as the one shown in Figure 2c and divide the given sequence $\mathrm{s} = s_0 s_2 \ldots s_{n-1}$ into parts so that the resulting parts in combination has the highest probability according to $\mathcal{M}$. Note that in Algorithm 2, 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 0.1 by the model shown in Figure 2c.

To briefly 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 subsequences appearing in the sequence log as patterns[4], with the premise that the number of appearances is higher than certain threshold θ.[5] Each pattern's initial probability is set

[3]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, being left out of the model. This can be seen as a way to prune the ineffective heuristics, an idea that Chuang and Smith (2018) also experimented in their previous report.

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

[5]Note that we do not impose this threshold restriction on sub-

Algorithm 2 Segmenting a Sequence Given a Model

```
Require: a model $\mathcal{M}$ for segmenting input sequence $\mathbf{s}$.
 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 $\mathbf{d}$
```

to be proportional to the number of appearances in the log. With this initial model, we then run Algorithm 2 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.

4 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 the experiments and compare the proposed approach to two alternatives.

4.1 HyFlex and Its Extensions

HyFlex is a software framework that was developed to facilitate the research of hyper-heuristics (Ochoa et al. 2012). 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. Each of these problem domains has an implementation 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., (Walker et al. 2012; Adriaensen, Ochoa, and Nowé 2015). In this work, we also use an extension (Adriaensen, Ochoa, and Nowé 2015) that offers an implementation of the quadratic assignment problem.

As mentioned in Section 2, 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 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 (Hyde et al. 2009), (Vázquez-Rodrıguez et al. 2009) and (Adriaensen, Ochoa, and Nowé 2015). For each problem domain, we will run experiments on multiple sets of problem instances where each set follows a particular distribution. Table 1 shows the name of each instance set and the source where we obtained them.

4.2 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[6] from (Chuang and Smith 2018) on each of the training problem instances for 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[7] 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 will split the sequence log into two collections. One collection contains the

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.

[6]That is, an algorithm that uses $\mathcal{H}_u$ as the heuristic selection policy and Luby's strategy as the length selection policy.

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

Table 1: Problem Domains & Sets of Instances

Domain	Instance Sets	Source
FS	100x10, 100x20, 200x10, 200x20, 500x20	(Taillard 1993)
BP	testdual4, testdual7, testdual8, testdual11	(Burke, Hyde, and Kendall 2010)
QAP	tai45e, tai75e	(Drezner, Hahn, and Taillard 2005)

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 sub-sequence to be considered as a pattern in the initial model. In order to account for the 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 $\rho = 0.02$ for FS and QAP, and 0.05 for BP.

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 1. As for the length policy (i.e. the $\mathcal{L}$ component), we will continue to use the Luby's sequence in the following experiments.

4.3 Empirical Results

With our distributional assumption, we conducted our experiments on each instance set separately, and 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. For each training problem instance, we performed 31 runs of the baseline algorithm from (Chuang and Smith 2018), each run lasted for 3 minutes on an Amazon Web Service's EC2 c4.large virtual machine, 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 1. The resulting algorithm was then tested for 31 runs, each run lasted 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. Most importantly, our objective for these experiments is to see if the proposed approach offers any speed-up over the baseline approach, i.e., whether the algorithm with a learned policy can find a solution with a better quality than the solution found by the baseline algorithm using the same amount of running time.

As a further comparison, we also tested a configuration that uses "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 based on the sequence log. 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 accounted 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, they use the same initial solution so that no one will have an advantage over the others by starting with 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 2. For each target problem instance, we compare the results of the proposed approach, denoted as Macro, to the results of Chuang and Smith's baseline, and to the configuration that uses plain models. 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.

In order to have 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 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.

Table 2: Results on Taillard's FS Instance Sets

(a) 100x10

Instance	Baseline	Plain	Macro
01	0.175546	**0.136412**$^{+}$	0.151506
02	0.255700	**0.227356**$^{+}$	**0.227356**$^{+}$
03	0.052854	0.052854	**0.051149**
04	0.816914	0.748280^{+}	**0.715916**$^{+}$
05	0.595951	0.578250	**0.519835**$^{+}$
06	0.100369	0.103411	**0.094286**
07	0.137219	0.090519^{+}	**0.088212**$^{+}$
08	0.552470	0.521458	**0.483555**
09	0.342306	**0.244504**$^{+}$	0.248350^{+}
10	0.081680	**0.056293**	0.059604

(b) 100x20

Instance	Baseline	Plain	Macro
01	2.518959	2.313510^{+}	**2.193361**$^{+}$
02	2.077496	1.880807^{+}	**1.700813**$^{+*}$
03	1.798859	1.584868^{+}	**1.547831**$^{+}$
04	1.676452	1.610073	**1.513850**$^{+*}$
05	2.118181	1.932214^{+}	**1.749824**$^{+*}$
06	2.109649	2.013341^{+}	**1.976339**$^{+}$
07	2.011755	1.980876	**1.768327**$^{+*}$
08	2.464333	2.269303^{+}	**2.173048**$^{+}$
09	2.172472	2.009510^{+}	**1.857345**$^{+*}$
10	1.772840	1.624435^{+}	**1.545218**$^{+}$

(c) 200x10

Instance	Baseline	Plain	Macro
01	0.192443	0.161557^{+}	**0.146114**$^{+}$
02	0.536814	0.428466^{+}	**0.403534**$^{+}$
03	0.442729	0.336994^{+}	**0.198770**$^{+*}$
04	0.039696	**0.034364**	0.035549
05	0.131803	0.123527	**0.120462**
06	0.362275	0.262337^{+}	**0.202999**$^{+}$
07	0.329891	0.244595^{+}	**0.234193**$^{+}$
08	0.398942	0.249526^{+}	**0.198719**$^{+*}$
09	0.250326	0.226220	**0.199024**$^{+}$
10	0.365037	0.316084	**0.252323**$^{+}$

(d) 200x20

Instance	Baseline	Plain	Macro
01	1.964587	1.732052^{+}	**1.679033**$^{+}$
02	2.517759	2.220027^{+}	**2.143435**$^{+}$
03	2.456314	2.197243^{+}	**2.143484**$^{+}$
04	2.174379	1.959230^{+}	**1.883413**$^{+}$
05	1.650006	1.395013^{+}	**1.273533**$^{+*}$
06	2.181518	1.955804^{+}	**1.848142**$^{+*}$
07	1.832974	1.575704^{+}	**1.524023**$^{+}$
08	2.117523	1.852263^{+}	**1.656164**$^{+*}$
09	2.391685	2.177823^{+}	**2.133725**$^{+}$
10	2.327908	**1.932683**$^{+}$	1.995268^{+}

(e) 500x20

Instance	Baseline	Plain	Macro
01	1.546738	1.316615^{+}	**1.262767**$^{+}$
02	1.577993	1.350654^{+}	**1.275848**$^{+*}$
03	1.439142	1.304096^{+}	**1.232292**$^{+*}$
04	1.212116	1.068116^{+}	**0.976545**$^{+*}$
05	1.120960	0.953753^{+}	**0.862126**$^{+*}$
06	1.225531	1.027550^{+}	**0.900477**$^{+*}$
07	0.996627	0.832581^{+}	**0.793341**$^{+}$
08	1.402910	1.204819^{+}	**1.103648**$^{+*}$
09	1.447488	1.245790^{+}	**1.169378**$^{+*}$
10	1.102093	0.930908^{+}	**0.892013**$^{+}$

Table 3: Results on Taillard's QAP Instance Sets

(a) tai45e

Instance	Baseline	Plain	Macro
01	5.984746	2.152215^{+}	**1.568631**$^{+}$
02	7.928935	4.937160^{+}	**4.789766**$^{+}$
03	6.414315	2.647260^{+}	**1.449401**$^{+}$
04	5.754245	**4.290159**	4.562749
05	3.781032	2.235093^{+}	**1.513122**$^{+}$
06	3.814179	2.779892	**2.564253**
07	5.798393	2.476576^{+}	**1.651908**$^{+}$
08	5.807830	4.365716	**3.557542**$^{+}$
09	6.214821	4.796009	**2.096192**$^{+*}$
10	4.259030	3.583970	**0.910981**$^{+*}$

(b) tai75e

Instance	Baseline	Plain	Macro
01	19.041342	16.238578^{+}	**12.578151**$^{+*}$
02	19.721550	19.836789	**13.911793**$^{+*}$
03	17.149603	16.152279	**11.497947**$^{+*}$
04	16.066372	14.313309	**10.116981**$^{+*}$
05	20.014321	15.954773^{+}	**12.920251**$^{+*}$
06	21.050871	19.556098	**11.310140**$^{+*}$
07	18.675130	14.813618^{+}	**11.363583**$^{+*}$
08	21.185602	18.548156^{+}	**14.333423**$^{+*}$
09	16.415657	13.739131^{+}	**10.255259**$^{+*}$
10	16.646816	12.334527^{+}	**10.240208**$^{+*}$

Furthermore, the significance of this advantage seems to increase as the problem becomes larger (i.e., increasing the number of jobs) or harder (i.e., increasing the number of machines 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 sometimes gives a better ARPD, the differences are not statistically significant.

The results of the experiments on the QAP domain are shown in Table 3. These results are also represented in ARPDs. In the QAP domain, we also observe the same phenomenon as in the FS domain: overall, the proposed approach usually gives a better result. Furthermore, for the larger problem instances (i.e., instances in tai75e) the differences between the proposed approach and the plain approach are generally significant.

Table 4 shows the results of the experiments on the BP domain. In this place, problem instances from all four instance sets are of the same sizes: 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 (Burke, Hyde, and Kendall 2010), 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 approach offered statistically better results. However, for the testdual8 instance set, the significance seems to drop. We suspect that it was 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 Discussion

In this paper, we are mainly interested in whether a learned $\mathcal{H}$ policy offers any speed-up over baseline methods, and furthermore, whether extending to chunks of heuristics (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, but rather 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., adding them into the set of heuristics H), we can incorporate their power as well to further boost the results.

Although the scenario that we considered in this paper (i.e, explicit distributional assumption and learning from a log of sequences) has had little discussion in the previous hyper-heuristic research, here we would like to offer an indirect connection. Note that Chuang and Smith (2018) experimented with a pruning strategy that eliminates a heuristic from further applications if after a period of time of running the algorithm, we never had observed that heuristic in any chain that had led to an improving solution. It was shown that by using this strategy, we can obtain a competitive result against most of the earlier hyper-heuristics. To relate to this pruning strategy, note that the "plain" model that we introduced in the experimental section can be seen as its generalization, and we speculate that they will offer comparable performance under the same settings.

6 Conclusions

In this paper, 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), and it will also give 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.

Table 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**$^{+*}$

References

Adriaensen, S.; Ochoa, G.; and Nowé, A. 2015. A benchmark set extension and comparative study for the HyFlex framework. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, 784–791. IEEE.

Burke, E. K.; Hyde, M.; Kendall, G.; Ochoa, G.; Özcan, E.; and Woodward, J. R. 2010. A classification of hyper-heuristic approaches. In Gendreau, M., and Potvin, J.-Y., eds., *Handbook of Metaheuristics*. Boston, MA: Springer US. 449–468.

Burke, E. K.; Gendreau, M.; Hyde, M.; Kendall, G.; Ochoa, G.; Ozcan, E.; and Qu, R. 2013. Hyper-heuristics: A survey of the state of the art. *the Journal of the Operational Research Society* 64.

Burke, E. K.; Hyde, M. R.; and Kendall, G. 2010. Providing a memory mechanism to enhance the evolutionary design of heuristics. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, 1–8. IEEE.

Chakhlevitch, K., and Cowling, P. 2008. Hyperheuristics: Recent developments. In Cotta, C.; Sevaux, M.; and Sörensen, K., eds., *Adaptive and Multilevel Metaheuristics*. Berlin, Heidelberg: Springer Berlin Heidelberg. 3–29.

Chuang, C.-Y., and Smith, S. F. 2018. Combining neighborhood-based heuristics: A framework and pilot study on baselines. Technical Report CMU-RI-TR-19-04, Carnegie Mellon University.

Drezner, Z.; Hahn, P. M.; and Taillard, É. D. 2005. Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. *Annals of Operations research* 139(1):65–94.

Hansen, P.; Mladenović, N.; Brimberg, J.; and Pérez, J. A. M. 2019. Variable neighborhood search. In *Handbook of metaheuristics*. Springer. 57–97.

Hyde, M.; Ochoa, G.; Curtois, T.; and Vázquez-Rodríguez, J. 2009. A HyFlex module for the one dimensional bin-packing problem. Technical report, School of Computer Science, University of Nottingham.

Lourenço, H. R.; Martin, O. C.; and Stützle, T. 2010. Iterated local search: Framework and applications. In *Handbook of metaheuristics*. Springer. 363–397.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of Las Vegas algorithms. In *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, 128–133. IEEE.

Nowicki, E., and Smutnicki, C. 2005. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling* 8(2):145–159.

Ochoa, G.; Hyde, M.; Curtois, T.; Vazquez-Rodriguez, J. A.; Walker, J.; Gendreau, M.; Kendall, G.; McCollum, B.; Parkes, A. J.; Petrovic, S.; and Burke, E. K. 2012. HyFlex: A benchmark framework for cross-domain heuristic search. In *Evolutionary Computation in Combinatorial Optimization: 12th European Conference, EvoCOP 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg. 136–147.

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

Vázquez-Rodrıguez, J. A.; Ochoa, G.; Curtois, T.; and Hyde, M. 2009. A HyFlex module for the permutation flow shop problem. Technical report, School of Computer Science, University of Nottingham.

Walker, J. D.; Ochoa, G.; Gendreau, M.; and Burke, E. K. 2012. Vehicle routing and adaptive iterated local search within the HyFlex hyper-heuristic framework. In *Learning and Intelligent Optimization*. Springer. 265–276.