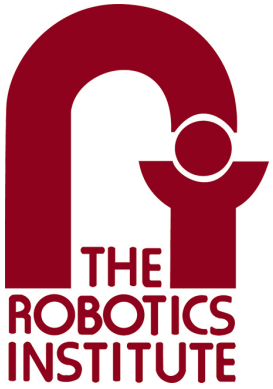
CMU-RI-TR-19-04

Combining Neighborhood-based Heuristics: A Framework and Pilot Study on Baselines

December 2018



Chung-Yao Chuang and Stephen F. Smith

The Robotics Institute
School of Computer Science
Carnegie Mellon University

Combining Neighborhood-based Heuristics: A Framework and Pilot Study on Baselines

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

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

***Abstract*—In this paper, we propose an algorithmic framework for combining multiple neighborhood-based heuristics. An algorithm derived from this framework will function by chaining the heuristics in a pipelined fashion. Conceptually, this framework is an algorithmic template that contains two pieces of changeable 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 will first offer a theoretical discussion on the design of the policy $\mathcal{L}$, and provide empirical evidence showing the effectiveness of the derived algorithm. We will then briefly touch on the issue of designing the policy $\mathcal{H}$ at the end of this paper, and demonstrate a simple pruning strategy and how it can contribute positively to the performance of the algorithm.**

I. Introduction

In most real-world situations, it is often the case that for a computationally challenging problem, there exists multiple heuristics, and it is generally the case that any such heuristic exploits only a limited number of problem characteristics among all the possible problem characteristics that we can think of. 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. In order to compensate for this, researchers have been investigating and experimenting with the idea of *combining multiple heuristics* (e.g. [1], [2], [3].)

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 facets similar to simpler and more-studied problems. For example, we can see that 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. In this case, it is very natural to attempt to reuse the heuristics that we developed for those more-studied problem domains.

In this paper, we propose a study on how to automatically combine multiple neighborhood-based heuristics. In the following, we will first review some background and describe the algorithmic framework that we intend to study.

A. Neighborhood-based Heuristics

The class of search heuristics considered in this study 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 $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ (where $\mathcal{S}$ is the solution space) 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 identified 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 $\mathbf{x}'$ is better than $\mathbf{x}$), 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:

Algorithm 1 A 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.

$\mathbf{x} \leftarrow$ initial solution.
while stopping criteria not met **do**
  $\ell \leftarrow$ a length chosen according to $\mathcal{L}$
  $\mathbf{x}_0 \leftarrow \mathbf{x}$
  for $i = 0$ to $(\ell - 1)$ **do**
    $h_{i+1} \leftarrow$ a heuristic chosen from H according to $\mathcal{H}$.
    $\mathbf{x}_{i+1} \leftarrow h_{i+1}(\mathbf{x}_i)$
    if $\mathbf{x}_{i+1}$ is better than $\mathbf{x}$ **then**
      break
    end if
  end for
  if the best among $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\ell$ is better than $\mathbf{x}$ **then**
    $\mathbf{x} \leftarrow$ the best among $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\ell$
  end if
end while

We can put a priority on the members of $\mathcal{N}(\mathbf{x})$, and examine them 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 $\mathcal{N}(\mathbf{x})$. The first-improvement version can be used for efficiency purpose, especially when the neighborhood size is huge. This also demonstrates another element mentioned above for specifying heuristics: we can alter the transition rule to specify a different heuristic.

As for the iterating conditions, there is a wide range of options as well. For example, we can specify a criterion based on time (e.g. the heuristic will run for 10 minutes), or we can specify it based on repetition (e.g. the heuristic will run for a single transition, 10 transitions, or until no further improvements.) And it is easy to come up with some other possibilities, such as iterating until the improvement is sufficiently small.

B. Subject of Study

In this section, we outline an algorithmic framework that we would like to study in this research. 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 section, the class of heuristics that we consider in this study are neighborhood-based heuristics. One characteristic of them 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 interact 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 a heuristic (chosen by certain policy) to the previous solution to generate the next solution. The algorithmic description of this idea is shown in Algorithm 1. 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

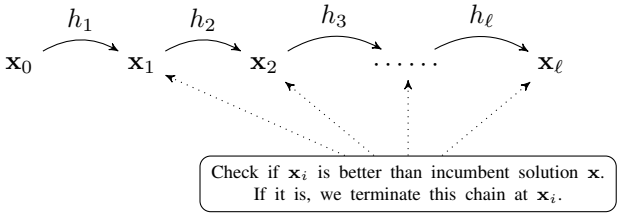


Fig. 1. The process of sampling a solution chain starts with a solution $\mathbf{x}_0$. Each subsequent step consists of selecting a heuristic h_{i+1} from the set of provided heuristics H and applying h_{i+1} to the previous solution $\mathbf{x}_i$ to get a new solution $\mathbf{x}_{i+1}$. If $\mathbf{x}_{i+1}$ 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. 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.

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 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. 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 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. Apparently, there are still two elements that we need to supply to make Algorithm 1 functional: the policy $\mathcal{L}$ for choosing the chain lengths and the policy $\mathcal{H}$ for choosing the heuristics. In this paper, we will first offer some theoretical discussion on the policy $\mathcal{L}$. To promote the generality of the result, we will center our discussion on a setting that contains two agnostic assumptions:

- 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 as a black-box) and a set of predefined heuristics.
- Secondly, we assume that we have no information about the amount of time allocated for running our algorithm.

For simplicity, this theoretic discussion on $\mathcal{L}$ will use a simple 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 can generalize to arbitrary $\mathcal{H}$. And at the end of this paper, we will also offer a preliminary empirical evidence of how choosing a more sophisticated $\mathcal{H}$ can have a positive impact.

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

briefly reviewing the field of hyper-heuristics and connect our algorithm to this field.

C. Hyper-heuristics

The term "hyper-heuristics" [4], [1] 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 which utilizes the lower-level components in combination to solve a given problem. The term "hyper-heuristics" was first introduced by Cowling et al. [5] 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. [6], [7].) In this place, 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 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. Offline learning, in contrast, takes place before the actual search starts. While most of the recent hyper-heuristics employ some forms of learning, in this paper, we will first look at the no-learning cases because it simplifies our discussion. Furthermore, as we will see in Section IV, a no-learning algorithm can still be pretty competitive when comparing to more elaborated, learning-based hyper-heuristics.

Hyper-heuristics without a learning mechanism are among the first hyper-heuristics being discussed. In their initial paper, Cowling et al. [5] 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 we laid out as Algorithm 1. 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 1 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 the following sections, we will first discuss the 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 will first define the scenario that we consider, which contains two agnostic assumptions. Based on that, Section III presents a strategy for choosing ℓ which has an asymptotic guarantee. Following that, Section IV shows the result of our preliminary experiments. And finally, we will close this paper with a brief look at a simple "learning" mechanism for choosing heuristics.

II. Agnostic Setting and Anytime Property

The scenario that we consider for our discussion on $\mathcal{L}$ contains two agnostic aspects. The first aspect is that we don't assume that we have detailed knowledge about the problem being solved except that we have access to the objective function (treating as a black-box) and a set of predefined heuristics. This aspect has been studied in previous hyper-heuristic research, often under the name of "cross-domain optimization" (e.g. [8], [9], [10], [11].) The main goal of this line of research is to design widely-applicable 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 reusing the higher-level policy. In this way, a well-designed hyper-heuristic has the potential to deal with the aforementioned agnostic assumption. In fact, this aspect has often been deemed as the de facto conception of selective hyper-heuristics.

The second agnostic aspect that we consider in this chapter is about the amount of computational resource that we have for

Algorithm 2 A Modified Version of Algorithm 1

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

solving a given problem. Most previous research (especially in hyper-heuristics) 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 hinders the development towards more general optimization methods because it 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 which performs well with a particular amount of computational resource will compare unfavorably if the resource is doubled or halved. For example, Remde et al. [12] compared several hyper-heuristics and noted that the best performing hyper-heuristics depend on the allotted CPU time. 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 solution *as soon and as often as possible*. If we look at Algorithm 1 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 1 into a procedure and rewriting it as Algorithm 2. 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, takes more time.

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 unsuitable to be applied because they assumes a fixed reward distribution[4].

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

[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.

Procedure 3 Search by Sampling a Solution Chain

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

Algorithm 4 A Modified Version of Algorithm 2

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

III. 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 4.

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

[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.

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.

A. 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 FindImprovement$(\mathbf{x}, \ell)$ will successfully return an improving solution. 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. *Given an incumbent solution $\mathbf{x}$, and let $q(\ell)$ be the probability that* FindImprovement$(\mathbf{x}, \ell)$ *returns an improving solution, then the fixed-length strategy $\mathcal{S}_{\ell^*} = (\ell^*, \ell^*, \ell^*, \ldots)$ where*

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

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

Proof. The key is to recognize that FindImprovement$(\mathbf{x}, \ell)$ is a Las Vegas algorithm if we set ℓ to ∞, i.e. whenever 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 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 [13]. □

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.)

B. Balanced Strategies for Unknown Distributions

In this subsection, we consider the situation in which $q(\ell)$ is unknown and describe a "balanced" strategy which 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:

$$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, 1, 2 \ldots$$

It can be defined recursively as

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

[5]Also note that it can be the case that 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.

This sequence was proposed by Luby et al. [13] and we will refer to it as Luby's sequence or Luby's strategy. Using the Luby's strategy in Algorithm 4, 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 Theorem 1 by 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 [13] 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 $m \log m$.

IV. Preliminary Experiments and Results

This section describes the setting of our preliminary experiments and results. We will first go over HyFlex, a software framework that the hyper-heuristics research community developed for testing hyper-heuristics. We then describe our algorithm implementation on HyFlex, which is based on Algorithm 4 and Luby's sequence. Implementing our algorithm on HyFlex enables us to compare our algorithm with other hyper-heuristics 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 participated in CHeSC were based on some sorts of learning during optimization, while on the contrary, our algorithm has no learning components. However, the results suggest that although we could not clinch top spots, our no-learning algorithm is nevertheless very competitive when comparing to most of the hyper-heuristics participated in CHeSC 2011.

[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 $q(\ell)$.

TABLE I
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

A. HyFlex and CHeSC 2011

HyFlex is a software framework that was created to aide the development and testing of hyper-heuristics [14]. It has a particular emphasis on 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[7]. 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.

Although the HyFlex's interface hides most details of these heuristics, it reveals a categorical description of them by classifying each heuristic as *mutation* (MU) which modifies a solution in some way with no guarantee of improvement, *ruin and recreate* (RR) heuristic which destructs a portion of the given complete solution and then reconstruct it in certain fashion, *hill climber* (HC) which performs a local search returning a solution that has the same or better quality than the original solution, or *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 climber heuristics. However, for this preliminary study, we will only use the default parameters and will not perform any further tuning.

HyFlex was used as the underlying framework for CHeSC. And for this competition, it provides six problem domains: one-dimensional bin packing (BP), personnel scheduling (PS), permutation flow shop problem (FS), the optimization version of boolean satisfiability problem (SAT), traveling salesman problem (TSP) and vehicle routing problem (VRP). Table I 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[8]. In the following, we will compare the results of our algorithm with the 20 contestants participated in the CHeSC.

B. Implementation Details

Basically, our algorithm implementation follows the structure of Algorithm 4. 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 have found a better solution to replace the incumbent solution, we will restart the Luby's sequence, i.e., jumping back to the beginning of the Luby's sequence.)

One augmentation that we have done for testing on CHeSC is that we 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[9] 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 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.

Another algorithmic modification that we have done is to allow the incumbent solution to be replaced by a solution of an 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

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

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

[9]For example, in our implementation, it is set to 15 milliseconds.

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, we found that for CHeSC, it is better to bound the Luby's sequence instead of growing arbitrarily. We suspect that this might be because some CHeSC domains have heuristics that are roughly equal to a random initialization of the solution. Once such a heuristic has been applied, the subsequent operations will typically have a much smaller chance of finding an improvement over the incumbent solution because this is equivalent to forfeiting the work that has been done to the point. So if such kind of heuristics is included in the domain, it is better to keep the sampling length short to minimize the occurrence of this kind of destruction and subsequent waste.

C. Preliminary Results

As mentioned previously, we use problem domains implemented in CHeSC as a benchmark and compare our results with the algorithms 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.

For our algorithm, we use the following configuration: We allow for an amplified heuristic to run for 15 milliseconds, and we bound the Luby's sequence at 32, i.e., when $\ell_i = 32$, the following sequence will be a repetition from the beginning of the sequence to ℓ_i. 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.

Table II shows the scores achieved by our algorithm, denoted as Luby-32, if competing with other contestants in CHeSC. As we can see, 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 participated in CHeSC 2011 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 an evidence that the principles that we relied on 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 III lists the scores and the rank of the algorithm if 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.

TABLE II
RANKING AND SCORES IF PARTICIPATED IN CHESC

Method	SAT	BP	PS	FS	TSP	VRP	Total
AdapHH	34.10	42.00	8.00	36.00	40.25	15.00	175.35
VNS-TW	33.60	2.00	37.50	33.00	18.25	5.00	129.35
ML	11.50	8.00	31.00	38.00	13.00	21.00	122.50
Luby-32	19.20	39.00	13.00	23.00	8.00	9.00	111.20
PHUNTER	8.50	2.00	11.50	7.00	25.25	33.00	87.25
EPH	0.00	6.00	10.50	17.00	33.25	12.00	78.75
HAHA	31.60	0.00	23.50	0.83	0.00	13.00	68.93
NAHH	11.50	17.00	2.00	21.00	12.00	5.00	68.50
ISEA	4.00	26.00	14.50	1.50	10.00	5.00	61.00
KSATS-HH	22.20	7.00	8.50	0.00	0.00	21.00	58.70
HAEA	0.00	1.00	1.00	5.33	11.00	26.00	44.33
ACO-HH	0.00	16.00	0.00	7.33	8.00	1.00	32.33
GenHive	0.00	10.00	6.00	5.00	3.00	6.00	30.00
SA-ILS	0.60	0.00	17.50	0.00	0.00	4.00	22.10
DynILS	0.00	9.00	0.00	0.00	11.00	0.00	20.00
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	2.50	0.00	2.00	0.00	4.50
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 III
USING LUBY'S SEQUENCE VS. USING $\mathcal{S}_1$

Method	SAT	BP	PS	FS	TSP	VRP	Total	Rank
Luby-32	19.20	39.00	13.00	23.00	8.00	9.00	111.20	4/21
Length 1	0.00	33.00	10.00	21.00	0.00	4.00	68.00	8/21

V. PRUNING HEURISTIC SET

As a prelude to our follow-up research, in this section we demonstrate how, by adding a simple mechanism that changes the policy for choosing heuristics, we are able to obtain a higher total score in CHeSC 2011 with the previous algorithm. This serves as a starting point for us to think about $\mathcal{H}$, the policy for choosing heuristics, and more importantly, how to design $\mathcal{H}$ in a more principled manner. Furthermore, how can we derive $\mathcal{H}$ automatically from prior experience.

Perhaps the simplest mechanism that we can add to our algorithm is to record 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 at the initial stage of a run. And after this initial stage, 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 redistribute 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 30 times in the initial stage.

As before, we performed our experiments on Amazon Web Service's EC2 c4.large virtual machines, and we set the

TABLE IV
PERFORMANCE OF ADDING A SIMPLE PRUNING MECHANISM

Method	SAT	BP	PS	FS	TSP	VRP	Total
AdapHH	34.10	42.00	8.00	33.00	40.25	14.00	171.35
Pruning	19.60	37.00	14.50	35.00	14.00	8.00	128.10
VNS-TW	33.60	2.00	37.50	29.00	18.25	5.00	125.35
ML	11.00	8.00	31.00	34.00	13.00	21.00	118.00
PHUNTER	8.00	2.00	11.50	6.50	24.25	32.00	84.25
EPH	0.00	6.00	10.50	17.00	33.25	12.00	78.75
HAHA	32.10	0.00	23.50	0.83	0.00	13.00	69.43
NAHH	11.50	17.00	1.00	21.00	11.00	6.00	67.50
KSATS-HH	22.35	7.00	8.50	0.00	0.00	22.00	59.85
ISEA	3.50	26.00	14.50	1.50	10.00	4.00	59.50
HAEA	0.00	1.00	1.00	5.33	10.00	27.00	44.33
ACO-HH	0.00	18.00	0.00	6.83	7.00	1.00	32.83
GenHive	0.00	10.00	6.00	5.00	2.00	6.00	29.00
SA-ILS	0.75	0.00	17.50	0.00	0.00	4.00	22.25
AVEG-Nep	10.50	0.00	0.00	0.00	0.00	9.00	19.50
DynILS	0.00	9.00	0.00	0.00	10.00	0.00	19.00
XCJ	3.50	10.00	0.00	0.00	0.00	5.00	18.50
GISS	0.75	0.00	8.00	0.00	0.00	6.00	14.75
SelfSearch	0.00	0.00	2.00	0.00	2.00	0.00	4.00
MCHH-S	3.75	0.00	0.00	0.00	0.00	0.00	3.75
Ant-Q	0.00	0.00	0.00	0.00	0.00	0.00	0.00

initial data-gathering stage to one minute. Table IV shows the scores obtained by our algorithm with the addition of the above pruning mechanism (denoted as Pruning in the table), comparing to the methods originally participated in CHeSC. As we can see, now we are able to obtain a higher total score and ranked second overall. This demonstrates the possibility that we may be able to deduce a better policy $\mathcal{H}$ from prior experience.

VI. FUTURE WORKS

In this paper, we described an architecture that allows us to chain multiple heuristics in a pipelined fashion. We laid out this architecture in the hope that based on this, we can further think about the issue of how to combine multiple neighborhood-based heuristics in a more principled manner. As a future work, we would like to continue on the exploration of how to design a policy for choosing heuristics, which corresponds to the $\mathcal{H}$ component in our framework. In this paper, we have looked at a baseline policy $\mathcal{H}_u$, that each time selects a heuristic uniformly at random from the set of heuristics. And in the last section, we have brief experimented with a simple "learning" policy that changes its behavior for choosing heuristics during a run. In the following work, we would like to study how to come up with a better policy, and furthermore, how to automate its construction.

REFERENCES

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

[2] C. P. Gomes and B. Selman, "Algorithm portfolio design: Theory vs. practice," in *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1997, pp. 190–197.

[3] M. Streeter, D. Golovin, and S. F. Smith, "Combining multiple heuristics online," in *In Proceedings of the Twenty-Second Conference on Artificial Intelligence*, 2007.

[4] K. Chakhlevitch and P. Cowling, "Hyperheuristics: Recent developments," in *Adaptive and Multilevel Metaheuristics*, C. Cotta, M. Sevaux, and K. Sörensen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 3–29.

[5] P. Cowling, G. Kendall, and E. Soubeiga, "A hyperheuristic approach to scheduling a sales summit," in *Practice and Theory of Automated Timetabling III: Third International Conference, PATAT 2000 Konstanz, Germany, August 16–18, 2000 Selected Papers*, E. Burke and W. Erben, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 176–190.

[6] E. K. Burke, M. R. Hyde, and G. Kendall, "Evolving bin packing heuristics with genetic programming," in *Parallel Problem Solving from Nature - PPSN IX: 9th International Conference, Reykjavik, Iceland, September 9-13, 2006, Proceedings*, T. P. Runarsson, H.-G. Beyer, E. Burke, J. J. Merelo-Guervós, L. D. Whitley, and X. Yao, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 860–869.

[7] N. B. Ho and J. C. Tay, "Evolving dispatching rules for solving the flexible job-shop problem," in *2005 IEEE Congress on Evolutionary Computation*, vol. 3, Sept 2005, pp. 2848–2855 Vol. 3.

[8] S. Asta and E. Özcan, "A tensor-based selection hyper-heuristic for cross-domain heuristic search," *Information Sciences*, vol. 299, pp. 412–432, 2015.

[9] P.-C. Hsiao, T.-C. Chiang, and L.-C. Fu, "A vns-based hyper-heuristic with adaptive computational budget of local search," in *2012 IEEE Congress on Evolutionary Computation*. IEEE, 2012, pp. 1–8.

[10] E. K. Burke, M. Gendreau, G. Ochoa, and J. D. Walker, "Adaptive iterated local search for cross-domain optimisation," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 1987–1994.

[11] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, B. McCollum, G. Ochoa, A. J. Parkes, and S. Petrovic, "The cross-domain heuristic search challenge – an international research competition," in *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, C. A. C. Coello, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 631–634.

[12] S. Remde, P. Cowling, K. Dahal, N. Colledge, and E. Selensky, "An empirical study of hyperheuristics for managing very large sets of low level heuristics," *Journal of the operational research society*, vol. 63, no. 3, pp. 392–405, 2012.

[13] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of las vegas algorithms," in *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*. IEEE, 1993, pp. 128–133.

[14] G. Ochoa, M. Hyde, T. Curtois, J. A. Vazquez-Rodriguez, J. Walker, M. Gendreau, G. Kendall, B. McCollum, A. J. Parkes, S. Petrovic, and E. K. Burke, "Hyflex: A benchmark framework for cross-domain heuristic search," in *Evolutionary Computation in Combinatorial Optimization: 12th European Conference, EvoCOP 2012, Málaga, Spain, April 11-13, 2012. Proceedings*, J.-K. Hao and M. Middendorf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 136–147.