Statistical Heuristic Selection for Graph Coloring

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Andrew Adams Freer

2012
Abstract of the Thesis

Statistical Heuristic Selection for Graph Coloring

by

Andrew Adams Freer

Master of Science in Computer Science

University of California, Los Angeles, 2012

Professor Miodrag Potkonjak, Chair

Although a heuristic algorithm’s usefulness is grounded in its empirical performance on a set of problem instances, much of recent graph coloring heuristic development neglects statistical methodology in several important ways. First, heuristic parameters are often set in an ad-hoc, irreproducible fashion. Second, heuristic parameters are often tuned and evaluated on the same set of instances, causing over-tuning. Last, the common winner-take-all approach limits a heuristic’s application to a very specific set of graph instances.

To address the above issues, we employ machine learning techniques to perform instance-based algorithm selection from a set of diverse heuristic algorithms, including multiple parameterizations of the same algorithm. The implemented strategy improves on a winner-take-all strategy by over a color on average on a set of IID random graphs when allowed a maximum runtime of approximately 15 minutes, and in general achieves near-optimal heuristic selection on unseen graphs.
The thesis of Andrew Adams Freer is approved.

Adnan Youssef Darwiche

Glenn D. Reinman

Miodrag Potkonjak, Committee Chair

University of California, Los Angeles

2012
# Table of Contents

1 Introduction .................................................................................. 1
  1.1 Graph Coloring ........................................................................ 1
    1.1.1 Problem Definition ......................................................... 1
    1.1.2 Algorithms ..................................................................... 2
    1.1.3 Heuristic Algorithms ..................................................... 3
  1.2 Problem Formulation and Proposed Methodology .................... 5
    1.2.1 Lack of Statistical Analysis in Heuristic Research .............. 5
    1.2.2 Proposed Methodology .................................................. 7
    1.2.3 An Illustration with State-Of-The-Art Heuristic Algorithms . 11
  1.3 Related Work .......................................................................... 16
    1.3.1 Algorithm Portfolios ...................................................... 16
    1.3.2 Parameter Setting ........................................................ 17
    1.3.3 Hyper-heuristics ........................................................... 18
    1.3.4 Relation of Proposal to Previous Work ............................ 19

2 Design and Implementation ........................................................... 23
  2.1 Development Framework and Principles ................................... 23
    2.1.1 MySQL Database ......................................................... 23
    2.1.2 Java ............................................................................. 24
    2.1.3 Miscellaneous ............................................................. 26
  2.2 Graph Coloring Benchmarks and Materials ............................ 26
  2.3 Implemented Heuristic Algorithms ......................................... 27
    2.3.1 DSATUR: An Effective Constructive Heuristic ................. 27
# List of Figures

1.1 A proper vertex coloring of the Petersen graph with three colors, the minimum number possible [Wik12]. ............................ 2

1.2 The average number of colors over all benchmarks used by each algorithm, as the maximum allowed runtime is steadily increased from 30 seconds to 3600 seconds (one hour). The “Optimal Algorithm” denotes the selection of an algorithm with minimum average colors for each instance. ....................... 14

1.3 The average runtime (three trials) to solve the k-GCP instance on a value of k for the algorithm configurations under consideration on flat1000_50.0. 15

1.4 The success rate (three trials) to solve the k-GCP instance on a value of k for the algorithm configurations under consideration on flat1000_50.0. 16

2.1 DSATUR pseudocode ................................................. 28

2.2 Modified TABUCOL pseudocode ................................. 30

2.3 HCA pseudocode .................................................. 32

2.4 GPX pseudocode .................................................. 33

3.1 The average number of colors over all instances as the maximum runtime is steadily increased from 30 seconds to 3600 seconds (one hour). ............... 39

3.2 Figure 3.1, normalized to the performance of the “Best Algorithm” strategy. 40

3.3 The average number of colors of each algorithm over all instances. ............ 43

3.4 The fraction of problem instances in which the underlying machine learning algorithm selects a graph coloring algorithm to be applied to a problem instance. 44

3.5 Figure 3.2, with additional contour lines representing different training sample sizes. ................................................................. 45
List of Tables

1.1 Average solution colors (five trials) for several algorithm configurations with a time limit of one hour on current benchmark instances. Table 1.2 is a legend for the different algorithm configurations [x]. ............................................. 12

1.2 Legend for Table 1.1. [Algorithm(Parameter1=value1, . . .)] denotes an algorithm configuration. ................................................................. 13
I would like to thank my advisor, Professor Miodrag Potkonjak, for his invaluable insight, guidance, encouragement, and humor throughout my graduate career. I would also like to thank the members of the committee, Professor Adnan Youssef Darwiche and Associate Professor Glenn D. Reinman, for whom I have great respect. Lastly, I would like to thank Karen, my family, and my friends for their support and patience.
CHAPTER 1

Introduction

In this chapter we provide an introduction to the latest research in graph coloring, identify some open issues in graph coloring heuristic algorithm development, propose a strategy to address them, examine work that is related to this issue, and place the proposed approach in context of the related work.

1.1 Graph Coloring

In this section we provide an introduction to the latest research in graph coloring. This includes a formal definition of the problem, references to some current applications, and an overview of current state-of-the-art algorithms for graph coloring. An emphasis is placed on heuristic algorithms for graph coloring, since they are currently the most effective algorithms for large graphs.

1.1.1 Problem Definition

Graph coloring is a well-known combinatorial optimization problem in computer science and many related scientific and engineering fields. The most common variation is more explicitly called vertex coloring. Given a graph $G = (V, E)$, the goal is to assign a “color” to each vertex such that no vertex is connected to another vertex of the same color, using a minimum total number of distinct colors. Alternatively, this problem may be viewed as partitioning the vertexes $V$ into a minimum number of independent sets $k$.

Graph coloring is computationally difficult. A graph which can be legally colored with $k$ colors is known as $k$-colorable. The problem of determining if a graph is $k$-colorable ($k$-GCP
for short), is NP-complete for all $k \geq 3$. The minimum number of colors $k$ for which a graph $G$ is $k$-colorable is known as its chromatic number, $\chi(G)$. The problem of determining a graph’s chromatic number (GCP for short) is NP-hard.

There are many useful applications of graph coloring, including school and university timetabling [BN99, Car86, Lew08], sports scheduling [LT11], frequency assignment [AHK03, Val01], compiler register allocation [Cha04], and short-circuit testing [GJS76]. Typically, these applications are interested in solving the GCP as well as possible within a reasonable amount of time. This has led to a variety of methods, including exact, approximate, and heuristic algorithms.

![Figure 1.1: A proper vertex coloring of the Petersen graph with three colors, the minimum number possible [Wik12].](image)

### 1.1.2 Algorithms

Several exact algorithms have been developed for GCP, including [GM11, MMT11, MZ06, MT95, Bre79]; for a recent survey, refer to Malaguti and Toth [MT10]. However, the run time for all exact methods is exponential, and the required memory for some of them is exponential as well, so running these methods to completion quickly becomes intractable as the size of the graph becomes large. An often cited number for intractability is greater than 100 nodes [MT10, HPZ08], although the complete story is more complex, and depends on several other factors. More importantly, from an anytime algorithm perspective, none of them explore the search space as effectively as state-of-the-art heuristic methods, and as a result are typically outperformed in all published empirical results, as demonstrated by
computational results on larger graphs in the survey by Malaguti and Toth [MT10].

Approximation guarantees for GCP exist, but are generally too weak for practical use. For example, a trivial greedy algorithm colors each vertex with the smallest available non-conflicting color, yielding a $k$-coloring of $k \leq (\Delta(G) + 1)$, where $\Delta(G)$ denotes the maximum degree of any vertex $v$ in graph $G$. If we consider DIMACS benchmark graph DSJC1000.9 ($|V| = 1000$, $|E| = 898,898$, $\Delta(G) \approx 900$) [JT96], we note that $\chi(G) \leq 224$ [MT10], making $(\Delta(G) + 1)$ a very poor approximation. For more information about approximation algorithms for GCP, we refer the reader to a recent survey by Paschos [Pas03].

The weaknesses of the aforementioned exact and approximation algorithms have led to the development of a large variety of heuristic algorithms, to be discussed in the following section.

1.1.3 Heuristic Algorithms

The many heuristic graph coloring algorithms can be broken into a variety of different classifications, from which multiple elements are often successfully combined. For an overview of metaheuristics in combinatorial optimization in general by Blum et al. please see [BR03]. In addition, there are several helpful recent surveys on graph coloring heuristics [CDS05, GH06, MT10]. For context, we provide a brief overview of currently popular graph coloring heuristics below:

1. **Constructive algorithms** [Bre79, Lei79, JAM91, Hal93, CL96, KP98]. Constructive algorithms for graph coloring typically color vertexes one at a time in a greedy fashion. Once a vertex is colored it is never again considered. This allows efficient solution generation, but often results in poor solution quality. Some constructive algorithms include performance guarantees [Hal93]. Some recursively construct and remove entire color classes from the graph [Lei79, JAM91, KP98], optimizing certain heuristic properties of created color classes. Many are iterative, performing minor quality-preserving transformations in between runs [CL96], or introducing randomization during the process [JAM91, KP98]. Although these heuristics generate unsatisfactory solutions on
their own, they are often used to create good initial starting points for local search algorithms.

2. **Local search algorithms** [HW87, Mor96, HM03, BZ08, HPZ08]. Stochastic local search algorithms are currently one of the most popular class of techniques for graph coloring, partly because the rapid generation of neighboring solutions and partly because of their empirically-demonstrated ability to escape local optima. TABUCOL [HW87], a stochastic local search based on tabu search [GL97], is currently one of the most popular algorithms for graph coloring, and is often used as a subroutine of other more complex hybrid graph coloring heuristics. PARTIALCOL [BZ08], VNS [HM03], and VSS [HPZ08] are other competitive stochastic local search algorithms for graph coloring. These local search algorithms represent a variety of search spaces, search neighborhoods, and local-optima escaping techniques. The more complex methods employ several different search spaces (e.g. VSS [HPZ08]), search neighborhoods (e.g. VNS [HM03]), and reactive schemes for local-optima escaping strategies (e.g. variants of PARTIALCOL [BZ08]).

3. **Memetic algorithms** [GH99, GHZ08, MMT08, LH10]. Memetic algorithms are a variant of genetic/evolutionary algorithms, where random mutation is replaced by a local search algorithm. An effective crossover for graph coloring called Greedy Partition Crossover (GPX) [GH99] greedily constructs new colorings by combining color classes of two existing solutions. Variants of this crossover have been used to introduce diversity into stochastic local search algorithms, resulting in several competitive memetic algorithms for graph coloring which improve upon the effectiveness of their embedded local search algorithms under certain conditions. The first well-known memetic algorithm for graph coloring is HCA [GH99], which has been followed by several other successful variations (e.g. AMACOL [GHZ08], MMT [MMT08], and MACOL [LH10]).

4. **Other algorithms** [CH97, JC98]. There are numerous metaheuristic techniques for combinatorial optimization problems. Many have been successfully applied to the graph coloring problem, although different algorithms fall in and out of favor, and not
necessarily due to clear dominance of one over the other. Some examples of heuristic algorithms for graph coloring which differ significantly from algorithms described above include ant colony optimization [CH97] and squeaky wheel optimization [JC98], which can be considered to be variations of reinforcement learning.

1.2 Problem Formulation and Proposed Methodology

In this section we identify some open issues in graph coloring heuristic algorithm development stemming from a lack of rigorous statistical analysis. We propose a methodology to address some of these issues. Finally, we illustrate the issues on some common graph coloring benchmark problem instances with several state-of-the-art heuristic algorithms and demonstrate possible performance gains achieved with the new methodology.

1.2.1 Lack of Statistical Analysis in Heuristic Research

“Heuristic” is an overloaded term in computer science. Informally it means a “rule of thumb”, i.e. a technique that seems to work well in practice. In computer science it has several meanings. In heuristic search, for example, a heuristic is an estimate of the total cost from a current state to a goal state. Usually, an admissible heuristic is desired, which computes a provable lower bound on the total cost from a current state to a goal state. Note that admissible heuristics are more than just good guesses: they have provable theoretical properties.

In heuristic algorithms, although sometimes the heuristics involved have provable theoretical properties (such as approximation guarantees), typically the goal is to go beyond provable theoretical bounds and perform as well as possible in practice. To perform well in practice requires (i) a definition of “well” (i.e. an evaluation function) and (ii) the use of statistical analysis to determine what works well “in practice”. Although the definition of an evaluation function depends on the application, statistical analysis should always be a fundamental part of heuristic algorithm development.
Current graph coloring heuristic algorithm research has been neglecting statistical analysis in some important ways. Many of these issues are not limited to graph coloring, and apply to heuristic algorithm development in general.

1. **Ad-hoc parameter setting.** Most heuristic algorithms come with a variety of parameters which can dramatically impact the performance and characteristics of the algorithm. For example, memetic algorithms must choose how to initialize the population, the child generation scheme (which parents are selected, what crossover operation is used), the population update scheme (what is the static or dynamic size of the population, which parent is replaced by a child and when), when to terminate the [stochastic] embedded local search, and so forth. Any particular derived parameter setting may not transfer well to a different statistical pool of instances, and an ad-hoc parameter setting scheme may be difficult to replicate for a non-expert. Also, an ad-hoc parameter setting scheme is unlikely to bring a newly developed heuristic algorithm to its full potential.

2. **Tuning and evaluating an algorithm on the same instances.** Much of recent heuristic algorithm research in graph coloring can be classified as “horse race papers”, where the goal is to dominate other state-of-the-art algorithms on a set of common benchmarks. In this scenario, algorithm parameters are often tuned on the same set of common benchmarks on which they are evaluated, as this makes them more competitive. This leads to a well-known problem in machine learning known as “over-tuning”, where an algorithm is tuned to the point where it works well on a specific set of instances at the expense of performing poorly on other unseen instances. This problem can be statistically mitigated by including a sufficient number of diverse problem instances in the common set of benchmarks, but it is unclear if this property is valid for the current common set of graph coloring benchmarks. In addition, some research reports results where the algorithm under development is tuned for each individual problem instance, but does not include the tuning time in the reported runtime, which underestimates the amount of time needed to solve a new, unseen instance.
3. **Winner-take-all approach.** Given the variety of possible objective functions that a practical user of an algorithm may be concerned with (e.g. runtime/cost tradeoffs, minimized performance variability, various resource constraints), it is practically impossible for one algorithm to dominate another. Even with respect to a single objective function, it is difficult for any one relatively simple algorithm to dominate all other reported algorithms on a diverse set of graph instances. The issue here is that the winner-take-all approach may limit the application of a developed algorithm to a very specific set of graph instances.

In summary, the lack of statistical analysis can create misleading results, and has several detrimental impacts on heuristic algorithm development. Without proper parameter setting techniques, heuristic algorithms cannot reach their full potential, especially on unseen instances. Similarly, requiring a single heuristic algorithm to dominate all others on a diverse set of graph instances is unrealistic, and may hinder the development of useful heuristic algorithms that work extremely well in a specific setting, like a special purpose tool in your algorithmic toolbox.

### 1.2.2 Proposed Methodology

To address these issues, we propose the following methodology. We will borrow heavily from the machine learning paradigm. First, to address issues of over-tuning, we will train offline (e.g. “tune”) on problem instances that are randomly generated from a distribution of interest. We will then evaluate on a test set generated from the same distribution. Second, to address shortcomings of the “winner-take-all” approach, we will use machine learning techniques to perform algorithm selection for a problem instance from a set of diverse heuristic algorithms. Parameter setting will be addressed by selecting among multiple parameterizations of the same algorithm.
1.2.2.1 Evaluation Function

Our primary evaluation function will be minimizing the average solution colors (i.e. maximizing solution quality) given a maximum runtime, a primary concern of current graph coloring research [MT10]. We note that a different method for aggregating results between instances of varying chromatic number could be used instead of directly averaging the solution colors. The largest maximum runtime considered is one hour, which is similar to the experimental maximum runtime used in other current graph coloring research [BZ08, HPZ08, LTM12].

1.2.2.2 Algorithms

A small number of heuristic algorithms will be selected based on the following criteria:

1. **Competitiveness.** Some currently competitive graph coloring heuristic algorithms will be selected in order to obtain non-trivial experimental results. We note that including non-competitive solvers that are unlikely to be selected by the machine learning algorithm will result in wasted training time, and will have no positive impact on the performance of the algorithm selection strategy.

2. **Diversity.** Algorithms will be selected in order to maximize the diversity of the set of algorithms. Since the evaluation function involves varying maximum runtimes, algorithms that span the continuum of greedy to robust are desirable. Also, algorithms that explore different spaces or use radically different techniques are desirable.

3. **Simplicity.** While the above issues are of primary concern to the overall performance of the algorithm selection strategy, ease of development is also an important consideration. With that in mind, algorithms will be selected partially with respect to ease of implementation (or availability of source code). It should be noted that simplicity does not imply a lack of competitiveness.
1.2.2.3 Machine Learning Approach

Next, we consider the machine learning approach. The problem can be viewed as a classification problem with a generalized loss function, where the loss associated with predicting algorithm $a$ for problem instance $p$ is the achieved solution cost of $a$ on $p$, subject to the maximum runtime constraint. Unfortunately, most machine learning classifiers only consider the common 0/1 loss function, which does not account for ties, and more generally attempts to minimize the prediction error instead of the achieved solution cost as we desire. A simple alternative to 0/1 classification is to use a regression technique to predict the solution cost of each algorithm. Then, the algorithm with the minimum predicted solution cost is selected. Although any regression technique could be used in this strategy, this work uses a regression tree. A regression tree is a decision tree with regression models at its leaves.

1.2.2.4 Features

Discriminative features of problem instances must be identified in order to make accurate predictive models of algorithm performance. Some simple features for graph coloring include counts of the number of vertexes and edges. Given our interest in predicting the solution cost, these features are likely insufficient. An excellent feature would be an approximation for the chromatic number, but unfortunately no tight approximation for the chromatic number exists. However, we can efficiently compute an upper bound on the chromatic number using a constructive heuristic like DSATUR [Bre79]. In this work we will consider only “static features” of this nature which can be efficiently computed with negligible overhead.

1.2.2.5 Training Methodology

Last, but not least, we consider the training method. The primary concern of this work is on the performance achievable through algorithm selection, which does not include time and resources spent training the predictive models. Therefore, in this work a simple training strategy is adopted: each algorithm under consideration will be run on each training instance for the maximum runtime for some number of trials.
A mathematical proof based on statistics theory by Birattari [Bir09] shows that the variance of the estimator of an algorithm’s performance on a distribution of instances, when allowed $N$ total training trials, is minimized by performing one trial on $N$ instances, as opposed to any strategy that runs multiple trials on the same instance. Minimizing the variance of estimators corresponds to maximizing the accuracy of predictions. Thus, to maximize the predictive power of our models with respect to training time, each algorithm under consideration will be run on each training instance for the maximum runtime for only one trial.

Note that this is still a non-trivial amount of time, even with a small set of algorithms. Consider a scenario with 10 algorithms, 100 training instances, and a maximum runtime of one hour. This corresponds to a total training time of 1,000 hours, or roughly 42 days. While this can be addressed in part by parallel and distributed computation, clearly some more sophisticated strategy would be desirable, and is a promising area for future research.

1.2.2.6 Summary

In summary, the methodology taken in this paper involves the following:

1. Identification of a desired distribution of problem instances.

2. Identification of a desired evaluation function.

3. Selection of a set of algorithms, i.e. the “algorithm portfolio”.

4. Selection of a number of features of the problem instances.

5. Training predictive regression models of each algorithm’s performance on the problem instances. Training is performed offline and performs one trial per training instance in order to maximize the accuracy of predictions. The algorithm selection strategy consists of selecting the algorithm with the best predicted performance for a problem instance.
6. Evaluation of the algorithm selection strategy on a set of test problem instances. The predictive power of the trained models, and thus the optimality of the algorithm selection strategy, will be maximized if the set of test problem instances are randomly generated from the same distribution as the training problem instances.

1.2.3 An Illustration with State-Of-The-Art Heuristic Algorithms

Using code described later on, we illustrate the above mentioned issues on current graph coloring benchmark instances with several state-of-the-art heuristic algorithms. Please see Section 2.2 for more details about these benchmarks, Section 2.3 for more details about these algorithms and their implementations, and Section 2.5 for more details about the experimental platform.

Table 1.1 contains the results of running a series of GCP trials on a set of difficult graph coloring benchmark instances. Each trial is an hour long, on an Intel® Core™ i7 CPU 1.73 GHz with 4 GB RAM. Five trials were performed for each algorithm to reduce the statistical variance.

Three different heuristic algorithms were used. The first, TABUCOL [HW87], has been shown in a recent computational study [CS10] to still be a competitive algorithm, despite its relative simplicity and date of conception. The next algorithm, PARTIALCOL [BZ08], is a tabu search like TABUCOL, but uses a different neighborhood structure. Lastly, six configurations (i.e. parameterizations) of HCA [GH99] are included. HCA is a memetic algorithm that employs TABUCOL as a local improvement operator. The six configurations of HCA vary both the size of the population and the length of the local search improvement, and were selected strategically to maximize the diversity of the configurations’ performances. In summary, eight different algorithm configurations were selected in order to maximize performance diversity. Since all three of these algorithms are designed for $k$-GCP, an initial $k$ value is determined constructively using DSATUR [Bre79], after which the $k$-GCP algorithm is applied to decreasing values of $k$.

There are several interesting trends we can observe in this data which highlight the issues
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DSJC250.5</td>
<td>28.0</td>
<td>28.8</td>
<td>28.8</td>
<td>28.0</td>
<td>28.0</td>
<td>28.6</td>
<td>28.0</td>
<td>28.0</td>
</tr>
<tr>
<td>DSJC500.1</td>
<td>12.0</td>
<td>12.0</td>
<td>13.0</td>
<td>12.6</td>
<td>12.0</td>
<td>12.8</td>
<td>12.4</td>
<td>12.0</td>
</tr>
<tr>
<td>DSJC500.5</td>
<td>49.4</td>
<td>50.0</td>
<td>50.0</td>
<td>48.6</td>
<td>48.8</td>
<td>49.6</td>
<td>48.2</td>
<td>48.4</td>
</tr>
<tr>
<td>DSJC500.9</td>
<td>127.0</td>
<td>128.0</td>
<td>128.4</td>
<td>127.0</td>
<td>126.6</td>
<td>128.4</td>
<td>127.0</td>
<td>127.0</td>
</tr>
<tr>
<td>DSJC1000.1</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>20.6</td>
<td>21.0</td>
<td>20.8</td>
<td>20.0</td>
<td>21.0</td>
</tr>
<tr>
<td>DSJC1000.5</td>
<td>88.8</td>
<td>90.8</td>
<td>89.2</td>
<td>85.8</td>
<td>87.4</td>
<td>88.4</td>
<td>85.6</td>
<td>88.4</td>
</tr>
<tr>
<td>DSJC1000.9</td>
<td>226.8</td>
<td>228.8</td>
<td>232.2</td>
<td>227.2</td>
<td>228.0</td>
<td>231.2</td>
<td>228.6</td>
<td>230.6</td>
</tr>
<tr>
<td>DSJR500.1c</td>
<td>86.8</td>
<td>87.4</td>
<td>86.0</td>
<td>86.4</td>
<td>86.0</td>
<td>86.6</td>
<td>86.2</td>
<td>85.8</td>
</tr>
<tr>
<td>DSJR500.5</td>
<td>127.0</td>
<td>126.8</td>
<td>123.8</td>
<td>123.6</td>
<td>123.8</td>
<td>123.6</td>
<td>123.8</td>
<td>124.0</td>
</tr>
<tr>
<td>R250.5</td>
<td>66.0</td>
<td>66.2</td>
<td>66.0</td>
<td>65.8</td>
<td>65.4</td>
<td>66.0</td>
<td>65.8</td>
<td>65.6</td>
</tr>
<tr>
<td>R1000.1c</td>
<td>98.2</td>
<td>100.8</td>
<td>100.0</td>
<td>99.6</td>
<td>98.4</td>
<td>101.0</td>
<td>99.6</td>
<td>98.2</td>
</tr>
<tr>
<td>R1000.5</td>
<td>242.2</td>
<td>243.2</td>
<td>241.8</td>
<td>242.0</td>
<td>242.6</td>
<td>242.2</td>
<td>242.4</td>
<td>242.6</td>
</tr>
<tr>
<td>le450_15c</td>
<td>16.0</td>
<td>15.0</td>
<td>16.0</td>
<td>15.6</td>
<td>15.8</td>
<td>15.6</td>
<td>15.6</td>
<td>15.2</td>
</tr>
<tr>
<td>le450_15d</td>
<td>16.0</td>
<td>15.0</td>
<td>17.0</td>
<td>15.8</td>
<td>16.0</td>
<td>15.6</td>
<td>15.8</td>
<td>15.8</td>
</tr>
<tr>
<td>le450_25c</td>
<td>26.0</td>
<td>27.0</td>
<td>26.0</td>
<td>26.0</td>
<td>26.0</td>
<td>26.0</td>
<td>26.0</td>
<td>26.0</td>
</tr>
<tr>
<td>le450_25d</td>
<td>26.0</td>
<td>27.0</td>
<td>26.0</td>
<td>25.8</td>
<td>26.0</td>
<td>26.0</td>
<td>26.0</td>
<td>26.0</td>
</tr>
<tr>
<td>flat300_26_0</td>
<td>26.0</td>
<td>26.0</td>
<td>31.4</td>
<td>26.0</td>
<td>26.0</td>
<td>26.0</td>
<td>30.0</td>
<td>26.0</td>
</tr>
<tr>
<td>flat300_28_0</td>
<td>31.0</td>
<td>28.8</td>
<td>32.0</td>
<td>31.0</td>
<td>31.0</td>
<td>31.0</td>
<td>31.0</td>
<td>31.0</td>
</tr>
<tr>
<td>flat1000_50_0</td>
<td>69.8</td>
<td>50.0</td>
<td>87.6</td>
<td>69.0</td>
<td>76.0</td>
<td>84.8</td>
<td>70.4</td>
<td>81.0</td>
</tr>
<tr>
<td>flat1000_60_0</td>
<td>85.2</td>
<td>76.6</td>
<td>88.2</td>
<td>83.0</td>
<td>84.6</td>
<td>86.4</td>
<td>82.4</td>
<td>85.6</td>
</tr>
<tr>
<td>flat1000_76_0</td>
<td>88.4</td>
<td>89.8</td>
<td>88.4</td>
<td>84.6</td>
<td>86.6</td>
<td>87.0</td>
<td>84.6</td>
<td>87.6</td>
</tr>
<tr>
<td>C2000.5</td>
<td>165.6</td>
<td>166.4</td>
<td>164.8</td>
<td>161.8</td>
<td>165.6</td>
<td>164.6</td>
<td>165.8</td>
<td>167.2</td>
</tr>
<tr>
<td>C4000.5</td>
<td>313.2</td>
<td>312.8</td>
<td>336.8</td>
<td>320.2</td>
<td>317.0</td>
<td>349.4</td>
<td>324.6</td>
<td>318.6</td>
</tr>
<tr>
<td>latin_square_10</td>
<td><strong>102.8</strong></td>
<td>112.4</td>
<td>106.4</td>
<td>105.6</td>
<td>104.6</td>
<td>107.2</td>
<td>105.6</td>
<td>104.4</td>
</tr>
</tbody>
</table>

Table 1.1: Average solution colors (five trials) for several algorithm configurations with a time limit of one hour on current benchmark instances. Table 1.2 is a legend for the different algorithm configurations [x].
Table 1.2: Legend for Table 1.1. \([\text{Algorithm(Parameters=value, \ldots)}]\) denotes an algorithm configuration.

First, we note that no single configuration of HCA dominates another configuration of HCA. This is no surprise, as the original paper [GH99] tunes HCA individually for each instance in an ad-hoc fashion. A recent computational comparison [LTM12] sets the population size \(P\) to 10 and the local search iterations \(L\) to \(16|V|\) which seems to do fairly well, but clearly there is some additional benefit to be gained by using different parameter values on different instances, such as varying the size of the population.

Second, we note that a winner-take-all approach will not yield optimal results, and there are some graphs on which certain algorithm configurations clearly outperform others. Observing that some algorithm configurations may be slower but more robust than others, we plotted the effect that maximum runtime has on average colors used over all benchmarks in Figure 1.2. This figure also shows the average colors achieved by optimally selecting the correct algorithm to be used in advance. This figure demonstrates that the correct winner-take-all algorithm depends on your evaluation function. It also demonstrates possible performance gains achieved through instance-based algorithm selection.
Figure 1.2: The average number of colors over all benchmarks used by each algorithm, as the maximum allowed runtime is steadily increased from 30 seconds to 3600 seconds (one hour). The “Optimal Algorithm” denotes the selection of an algorithm with minimum average colors for each instance.

Lastly, we analyze an unexpected anomaly in Table 1.1 on graph flat1000_50_0. On this graph, PARTIALCOL outperforms both TABUCOL and HCA by at least 19 colors when the maximum allowed time is one hour, an outlier which strongly affects Figure 1.2. This is especially curious since the reported runtime on flat1000_50_0 with $k = 50$ for PARTIALCOL is 26 seconds and for TABUCOL is 426 seconds on a 2 GHz Pentium 4 with 512 MB of RAM, as reported by the original PARTIALCOL paper [BZ08]. We would expect this value for the minimum value of $k$ to reflect a somewhat substantial portion of the overall GCP runtime, so the fact that TABUCOL lags so far behind PARTIALCOL in this instance is surprising.

Figures 1.3 and 1.4 plot the average runtime and success rate of each algorithm for $k = [50, 90]$ averaged over three trials, with a maximum allowed runtime of 1800 seconds (30 minutes). The average runtime on flat1000_50_0 with $k = 50$ for PARTIALCOL is 14 seconds and for TABUCOL is 394 seconds, which corresponds to the aforementioned reported results. First, we note that HCA does not result in an improvement over TABUCOL in this instance, and the two configurations with the smallest local search improvement iterations fail at relatively large values of $k$. This is somewhat surprising, as most analysis of HCA indicates
that it greatly improves on the solution quality yielded by TABUCOL, given more time [GH99, LTM12]. Second, we observe that since the average runtime growth for TABUCOL is nearly constant for decreasing values of $k$, the amount of computation done on suboptimal values of $k$ dominates the overall runtime. What we can conclude from these two observations is the following: (i) it is unlikely that a particular algorithm is more robust than another algorithm on every graph in a diverse set of graph instances and (ii) even if one algorithm is more robust than another algorithm, it may require a significant amount of extra computation before it can catch up to the speedier, less robust algorithm. Again, some form of statistical analysis is necessary in order to make good heuristic algorithm choices.

Figure 1.3: The average runtime (three trials) to solve the $k$-GCP instance on a value of $k$ for the algorithm configurations under consideration on flat1000.50.0.
1.3 Related Work

In this section we examine related work. Several significant bodies of research are particularly relevant: algorithm portfolios, parameter setting, and hyper-heuristics. Lastly, we place the proposed approach in the context of the related work.

1.3.1 Algorithm Portfolios

The term “algorithm portfolios” initially referred to running several algorithms in parallel as opposed to just one algorithm [HLH97]. Running two instances of the same algorithm can convert the performance distribution of an algorithm with a heavy-tailed performance distribution, i.e. high risk (variance), into a new performance distribution with a reduced tail, i.e. low risk, and increased mean performance. Running instances of different algorithms in parallel can also reduce risk and increase mean performance: where when one does poorly, another does well. The price paid for this benefit is extra computation due to redundant work. Computation, or runtime, is not necessarily the sole measure of performance, but it is
trivially included in order to avoid infinite or intractable runtimes. However, depending on the original runtime distribution, a properly designed algorithm portfolio can still decrease the expected runtime. In summary, running several algorithms configurations in parallel can decrease the overall risk, increase the mean performance, and decrease the expected runtime.

Research into algorithm portfolios has since broadened in scope. In addition to running algorithms in parallel on different processors, algorithm portfolio research has investigated running algorithms interleaved on the same processor as well as employing various restart strategies [GS01]. Interleaving algorithms on the same processor, or in general a limited number of processors, highlights the issue of redundant computation associated with algorithm portfolios. This has led to a number of works concerned with instance-based and dynamic strategies for reducing the redundant computation involved [Ric76, LNA03a, GS06, SS08, XHH08].

1.3.2 Parameter Setting

Due to the large variety of parameters associated with many heuristic algorithms for combinatorial optimization, there exists a body of research associated with how to optimally set the parameters of an algorithm. This is occasionally referred to as “the parameter setting problem”. Approaches to the parameter setting problem include:

1. **Factorial experiment design.** Factorial experiments are designed to analyze the effect of factors on a response variable of interest. In the case of the parameter setting problems, parameters become factors, and the desired performance measure becomes the response variable. A full factorial experiment involves simulating the response variable for every combination of factor levels, which quickly becomes infeasible with a large number of factors, or for factors with a large number of possible values (e.g., discretized continuous factors). Therefore, for efficiency reasons, factorial experiments often include only some small number of levels for each factor (traditionally only two, the minimum and maximum). Also, some fraction of the experiments can be omitted under certain assumptions about the effects of various factors, such as considering only first order interactions among factors. Factorial experiment de-
sign has been successfully used on several occasions for the parameter setting problem [CGR01, AL06, RK07, RBS12].

2. **Local search methods.** Since the parameter space of an algorithm is a combinatorial problem, one approach to efficiently explore the space is local search. One example of a local search method for parameter setting is ParamILS, which changes one parameter at a time until it achieves a local optima [HHL09].

3. **Racing.** Birattari introduces a class of “racing algorithms” for tuning metaheuristics in [Bir09]. In short, a finite number of algorithm configurations are statistically evaluated on a series of randomly generated target problem instances. The key insight is this: once it becomes statistically significant that an algorithm is performing worse than its competitors, it is eliminated from further experimentation, freeing up valuable experimentation time which can be better used on more competitive configurations.

Most of the work in the parameter setting problem does not attempt to perform instance-based algorithm selection. One exception is a recent work by Ries et al. [RBS12], which combines a full factorial design, statistical analysis, and a fuzzy logic rule-based system to perform instance-specific parameter selection for the travelling salesman problem. The main shortcoming from a machine learning point of view is that the fuzzy logic system is configured manually, which is expensive, and has no statistical guarantees.

### 1.3.3 Hyper-heuristics

Hyper-heuristics are often described as “heuristics to choose heuristics”, but using statistical techniques (e.g. machine learning) to choose heuristics is not excluded [BKN03]. In contrast to algorithm portfolios, the focus of most research on hyper-heuristics is to reduce the development burden and domain-specific expertise required for creating high-performance heuristic algorithms [BKN03]. Several overviews for hyper-heuristics exist [BKN03, Ros05, BHK10].

A series of hyper-heuristic works related to timetabling (which is closely related to graph
coloring) have been performed. A small sampling includes [BKS03, BPQ06, BMM07, TK09]. In general, the focus of these works is on combining low-level heuristics such as constructive heuristics or local search neighborhood heuristics, as opposed to combining existing algorithms such as metaheuristics. Also, statistical analysis is not strongly emphasized, with the focus being more on “heuristics to choose heuristics” as opposed to the more general “methods to choose heuristics”.

1.3.4 Relation of Proposal to Previous Work

The work that has the most in common with this work belongs to research on algorithm portfolios. The aspect of algorithm portfolios related to scheduling is not as relevant, and we can view any algorithm portfolio schedule as a new algorithm, with its own new performance distribution. However, the body of work relating to algorithm selection is directly related to this work. Some notable recent work on algorithm selection includes the following:

1. A series of related works by Leyton-Brown and his coauthors [LNA03a, LNA03b, XHH08] involve the use of regression models to predict the expected performance of each algorithm in an “algorithm portfolio”. The considered applications are the winner determination problem (WDP) and the propositional satisfiability problem (SAT). The culmination is in the description of an algorithm submitted to the 2007 SAT competition, SATzilla [XHH08], which contains a variety of tricks to improve the accuracy of their regression models.

Our work differs in several ways. First, it differs in application, as the application under study in our work is graph coloring. Second, the primary focus in these related works has been on complete algorithms, not on incomplete heuristic algorithms. While Xu et al. [XHH08] eventually consider portfolios of both complete and incomplete algorithms, they do not consider portfolios containing only incomplete algorithms. Therefore, their regression models may allow effective selection between complete and incomplete algorithms on an instance, but not between different incomplete algorithms, which may be very difficult or impossible to predict. Our work considers incomplete
algorithms exclusively. Lastly, these related works also train offline and run each algorithm on each training instance. However, no mention is made of how many trials are run on each training instance, which neglects the observation made earlier that one trial per training instance makes optimal use of training time [Bir09].

2. A work by Streeter and Smith [SS08] integrates both scheduling and selection aspects of algorithm portfolios. It relies on two online approximation results. First, it uses an online greedy approximation algorithm to construct (nearly) four-approximations to the optimal schedule for a distribution of instances. Second, it uses results from the sleeping experts problem to extend the four-approximation to different subsets of the distribution of instances identified by boolean features. The applications under study are boolean satisfiability, zero-one integer programming, and A.I. planning. In contrast to this work, our work does not attempt to construct algorithm schedules, and training is not online. However, our work does not require the selection of boolean features, which essentially amounts to manually dividing the distribution of instances into a number of predetermined classes. Therefore, our work’s opportunity to discover and exploit various classes of instances is much more powerful and automatic.

3. Gagliolo and Schmidhuber [GS06] explore a dynamic, online approach to algorithm selection. Each algorithm in the portfolio is assigned a dynamic time share of the available processing resources, which is updated based on both static and dynamic features of the problem and the algorithms. The advantage of this dynamic approach is that it exploits more information than is present before the execution of any algorithm, and dynamic information about the algorithm state is likely to be one of the most informative predictors of its future performance (e.g. a lack of recent improvement could indicate cycling). This related work also implements a meta-selection between different dynamic time share allocators using a bandit-problem solver approach, but this approach is feature oblivious, and is based solely on past performance of the underlying dynamic time share allocators.

Dynamic, instance-based time share allocation to an algorithm portfolio can be seen
as a generalization (or superset) of all algorithm portfolio techniques discussed thus far. For instance, a dynamic instance-based time share allocator could always allocate 100% of available time to the predicted best algorithm. It also does not exclude the possibility of creating algorithm schedules as opposed to just running one algorithm. Lastly, it approaches training from an online perspective, which hopefully reduces the computational burden associated with offline training, or at a minimum will include it in the final evaluation of its performance.

The scope and goals of this related work are quite ambitious. Their results are applied to the winner determination problem (WDP) and the propositional satisfiability problem (SAT). While their problem is more challenging since the cost of training is essentially included in the evaluation, their results indicate that on a non-trivial amount of occasions their method performs worse than a uniform schedule (equal time allocated to each algorithm). It is not clear how successful their techniques for dynamic prediction of algorithm performance are, nor how successful their techniques for converting those predictions into effective schedules are. In addition, it is somewhat unclear why selection between multiple time allocators is desirable, unless no single time allocator effectively addresses the desired performance measure (a time allocator minimizing the expected runtime is curiously absent, even though this is the evaluation objective, and is mentioned in the work).

The vast majority of parameter setting work attempts to select the single best algorithm for a distribution of instances, and does not attempt to perform instance-based selection. Thus, our work stands apart. As mentioned previously, the one exception known to us [RBS12] involves a significant amount of manual analysis and configuration, and does not train on the same distribution on which it is tested. Our method for training does share the same quality and limitations of parameter setting research related to factorial experiment design. Our method for training borrows from the excellent research on racing algorithms [Bir09], but cannot easily make use of its primary advantage of dropping algorithms from consideration, as algorithms that perform worse on average may be the best performers on some unidentified subset of the problem distribution.
Our work has goals in common with the stated goals of hyper-heuristic research, i.e. general methods to choose among heuristics, reduced development time, and reduced domain expertise. However, as discussed in Section 1.3.3, the focus in hyper-heuristic research tends towards heuristic methods to select heuristics as opposed to statistical methods. Also, the selected heuristics in hyper-heuristics research tend to be low-level constructive or neighborhood move heuristics as opposed to existing competitive heuristic algorithms. Therefore, to our knowledge, our work has little in common with any previous work in hyper-heuristics research.
CHAPTER 2

Design and Implementation

We now present the design and implementation of the proposed methodology for algorithm selection for graph coloring heuristics. Implementation details include sections on the development framework and principles, publicly available graph coloring benchmarks and open source materials, and experimental platform details. Various implemented graph coloring heuristics are discussed in detail. A machine learning algorithm for predicting algorithm performance is selected and discussed. Finally, the different elements of selected experiments are identified and discussed.

2.1 Development Framework and Principles

The development framework and principles are based on good practices for experimental analysis of algorithms. For a good discussion of said practices by Johnson, the reader is referred to [Joh01]. Major elements of the development framework are discussed below.

2.1.1 MySQL Database

An open source MySQL transactional database (version 5.5.19) was used to store experimental results. This contributes to reproducibility, comparability, and efficient experimental design.

All implemented algorithms may be made deterministic via the setting of their (seeded) random number generator. For reproducibility, nearly all aspects of a given trial, e.g. a GCP trial, are stored in the database. This includes the name of the instance, a complete description of the algorithm and its configuration, the name of the random number generator
used, and random number generator seed. When accompanied with a tagged copy of the relevant source code and input files, this forms a completely reproducible description of all experimental data.

Although comparability is in part addressed by the reproducibility described above, it is enhanced by including machine-independent metrics such as iteration counts. A machine-independent operation count was defined for each implemented algorithm, and is stored alongside experimental runtime data.

Lastly, using a transactional database to store experimental results contributes greatly to efficient experimental design. First, any reasonably long-running experiment will almost certainly beg detailed analysis, not all of which can be planned for in advance. A transactional database trivially allows data to be collected just once, but analyzed many times. Second, storing intermediate results can allow multiple analyses of the same data. In this research, the primary interest is on the performance of algorithms on GCP given a maximum runtime. However, by recording statistics on each new minimum coloring achieved by a GCP algorithm, we can simulate the same GCP trial with smaller values of the maximum runtime with no additional experimentation. Lastly, some additional benefits include structured data and ACID properties that become increasingly important as experiments are parallelized.

2.1.2 Java

The project was implemented primarily in Java (JDK version 1.7.0_03). Java has a wide variety of advantages over C++, which is often the language of choice for developing algorithms and computationally intensive applications in general.

First, studies have generally found programmers to be more productive in Java, so it is especially attractive to researchers who want to make the best use of their time, and are planning to do a non-trivial amount of programming. In addition to automatic garbage collection, a major element of this success is Java’s fairly large and consistent set of APIs. Even basic APIs such as string manipulation and unit testing can give programmers an efficiency boost, and make them more likely to write effective and complete code on the first
iteration. For this project, Java had a major impact on the efficiency and correctness of the developed code.

Second, and most importantly for this project, Java has some very nice open-source libraries available in addition to the standard APIs. The JDBC API allowed easy access to the MySQL database. WEKA (Waikato Environment for Knowledge Analysis, version 3.6) [HFH09], an open source Java based library of modern machine learning algorithms, was employed to perform the machine learning experiments. To plot and analyze data, an open source Java based charting library named JFreeChart (version 1.0.14) was used. Although it did not end up being used in this work, JPPF (Java Parallel Processing Framework, version 3.0.1) [JPP12] is an open source Java based library for performing grid computing with relatively little setup; if time had permitted it would have been used to perform large experiments on a self-built cluster. Similar libraries are available for C++, but generally require a larger amount of setup and are not as well documented as Java (due to Java’s Javadocs).

Lastly, Java is platform independent. This allows virtually anyone to run the source code as is, regardless of their platform, without needing to tweak the source code in order to make it compile correctly, or worse, having to work out bugs which may not present themselves to different compilers (e.g. some C++ compilers initialize all uninitialized memory, some do not). In addition, it facilitates easy change of development environment or implementation of cluster computing solutions. While cluster computing fell out of the scope of this work due to time constraints, it was initially planned that experiments be run on a variety of available Windows and Linux based machines.

The primary concern of using Java over C++ is efficiency. The main assumption made here is that Java has a small constant overhead and that efficient implementations of algorithms will have a more important and significant effect on runtime. Without exception, experimental testing revealed comparable (or improved) runtimes with respect to reported runtimes on similar machine speeds and C++ implementations. Whether or not this justifies the perhaps perceived benefits of Java over C++ listed above is a subject of great debate among current programmers.
2.1.3 Miscellaneous

Runtime is a major element of this work. In order to minimize variance associated with runtime measurement, only time spent in user mode was recorded. This was accomplished via Java’s ThreadMXBean, which includes a function named getCurrentThreadUserTime. This function is not guaranteed to be available on all platforms supported by Java, but it was on all development machines. This factors out events such as operating system thread scheduling (more threads than processors), sleep mode, hibernate mode, etc. It also disregards time spent in kernel mode such as page faults, etc., which are often not the fault of the program in question and are instead due to the presence of other executing threads and processes in the context of the operating system. Since all algorithms under consideration are sequential and should spend approximately all of their time in user mode anyways, measuring runtime with user mode time is correct and desirable.

2.2 Graph Coloring Benchmarks and Materials

The currently used common graph coloring benchmarks come from the Second DIMACS Implementation Challenge [JT96]. This challenge defined a simple format for graph coloring problem graphs and solutions, and collected a diverse set of graphs to empirically evaluate the performance of graph coloring algorithms. Since then, the COLOR02/03/04 series was developed to “encourage research on computational methods for graph coloring problems, to evaluate alternative approaches using a common testbed, and to stimulate discussion on present and future directions in computational combinatorial optimization.” [JMT12]. The benchmarks from the COLOR02/03/04 series built off of the benchmarks from the Second DIMACS Implementation Challenge. All benchmarks are available for download off of their respective websites [Dis12, JMT12].

Several sets of open source materials for graph coloring, including both algorithms, instance generators, and interesting collections of problem instances, are publicly available.

Joseph Culberson has compiled a significant amount of quality materials for graph col-
oring together on his website [Cul12]. This includes a bibliography, algorithms for graph coloring, instance generators, and links to other useful graph coloring resources. We made use of his instance generators, which are responsible for a subset of the above mentioned benchmarks. We will describe the instance generators used in more detail later.

Some other interesting materials include an up-to-date bibliography on graph coloring [CG12] and open source algorithms available from a recent computational study of various graph coloring heuristics [CS10].

2.3 Implemented Heuristic Algorithms

A number of graph coloring heuristic algorithms were implemented in Java. They were selected according to the criteria outlined in Section 1.2.2.2. All are considered competitive in several recent surveys and computational comparison studies [MT10, CS10, LTM12]. The algorithms are diverse: two are local search solvers (TABUCOL [HW87] and PARTIALCOL [BZ08]) utilizing different problem spaces and neighborhoods, one is a memetic algorithm (HCA [GH99]) that keeps a population of solutions and generates new child solutions via an effective crossover. Diversity could be increased by implementing new algorithms with additional development cost and experimental effort. The selected algorithms are relatively simple to implement.

It should be noted that open source implementations for some of the following algorithms are available in C++ (see Section 2.2). For the purposes of our work, we found it simple and convenient to implement the algorithms ourselves in Java.

In the following sections we provide further details regarding the implemented algorithms relevant to this research.

2.3.1 DSATUR: An Effective Constructive Heuristic

DSATUR is a greedy constructive algorithm for GCP [Bre79]. It iteratively colors the vertex with maximal saturation degree, which is defined as the number of unique colors of colored
vertexes to which it is adjacent, initially zero for all vertexes. It remains a popular greedy
heuristic in current graph coloring research, and with slight modifications can be used to
create initial solutions for $k$-GCP as well, where $k$ is lower than the amount of colors used
by DSATUR. For example, the remaining uncolorable vertexes can be assigned randomly
or greedily to (conflicting) color classes. Pseudocode is displayed in Algorithm 2.1 below;
additional clarifications of the original published pseudocode in [Bre79] are italicized, and
although the lines to color the first vertex are not strictly necessary, they are included for
similarity.

```
Input: Graph $G = (V, E)$
Output: Solution $s = \{s_1, s_2, \ldots\}$, where elements $s_x \in [1, |V|]
1 Select an uncolored vertex $v_0$ according to the following order: maximum degree, 
   random.
2 Color $v_0$ with color 1 (the lowest available color).
3 while (uncolored vertexes remain) do
   4 Select an uncolored vertex $v$ according to the following order: maximum 
      saturation degree, maximum degree, random.
   5 Color $v$ with the lowest available color.
6 end
```

Figure 2.1: DSATUR pseudocode

DSATUR is used in several places throughout this work. It is used to generate a starting
$k$ for algorithms that are designed to work in decision mode, i.e. solve the $k$-GCP. It is also
used to create initial solutions for several algorithms which require an initial solution, or
solutions in the case of a population-based approach like HCA. Lastly, the number of colors
used by DSATUR is used as a predictive feature for the performance of algorithms for GCP
(see Section 2.6.3).

Different tie-breaking techniques for the vertex with maximum saturation degree are used
for DSATUR. The most common tie-breaking method is to choose in order of decreasing
degree, which corresponds to the original pseudocode [Bre79]. This is the method assumed
for the rest of this work. Some other common methods include choosing in order of increasing
degree, random tie-breaking, and in-order tie-breaking (dependent on the arbitrary vertex
ordering of the original input graph).
RLF [Lei79] is another popular greedy constructive algorithm for GCP. Many comparative studies of RLF and DSATUR show RLF to perform significantly better on several classes of graphs, but do not establish dominance [CS10]. However, RLF’s asymptotic computational dependence on the number of edges makes it more expensive to compute than DSATUR for large, dense graphs.

2.3.2 TABUCOL: A Tabu Search with Improper Solutions

TABUCOL [HW87] is a stochastic local search algorithm based on the tabu search paradigm [GL97]. The distinguishing feature of tabu search algorithms is a small memory (called the tabu list) which remembers recently performed neighborhood moves, and disallows them for a static or dynamic period of time (the tabu tenure). A tabu list provides a way to escape local optima, and is an alternative to methods such as random restarts and simulated annealing. In TABUCOL, the solution space consists of complete, improper graph colorings, where every vertex is assigned a color but some neighboring vertexes are conflicting (i.e. are assigned the same color). An allowed neighborhood move is a pair, (vertex, color), that assigns a color to a vertex.

HCA [GH99], a memetic algorithm for graph coloring, was created later and used TABUCOL as a subsystem for improving generated child solutions. It made some minor modifications that are generally considered to have been beneficial to TABUCOL in terms of robustness and performance, partly due to its impact on implementation efficiency [GH06]. Since this paper, most graph coloring research that refers to TABUCOL is actually referring to this implementation. The key differences from the original TABUCOL are the following:

1. Moves are only considered if the vertex is conflicting with at least one neighbor.
2. The tabu list maintains an entry for every possible move, rather than being a static size.
3. The tabu tenure \( t_l \) of a move is dynamically set according to the equation \( t_l = \text{Random}(A) + \alpha \times nb_{CFL} \). \( \text{Random}(A) \) returns a number uniformly selected from
$[0, A - 1]$. $nb_{CFL}$ denotes the number of conflicting vertexes in the current configuration. $A$ and $\alpha$ are thus parameters of the search and are typically set to 10 and 0.6 respectively.

4. An additional aspiration condition allows tabu moves that lead to a solution better than the best solution found so far.

For the remainder of this paper, TABUCOL shall refer to the modified version presented in the original HCA [GH99] paper, and in general, the only considered parameterization sets $A = 10$ and $\alpha = 0.6$. This is based on both reported results and experimental validation that demonstrates little profit in varying these parameters. The pseudocode is presented below in Figure 2.2.

---

**Input:** Graph $G = (V, E)$, Initial solution $s = \{s_1, s_2, \ldots\}$, where elements $s_x \in [1, |V|]$

**Output:** Last best solution $s^*$

1. $s^* \leftarrow s$
2. while !StopCondition() do
3. $\quad (v, i) \leftarrow \text{FindBestAllowedMove}()$
4. $\quad \text{SetTabu}((v, i))$
5. $\quad \text{PerformMove}((v, i))$
6. $\quad \text{if Cost}(s) \leq \text{Cost}(s^*) \text{ then} $
7. $\quad \quad s^* \leftarrow s$
8. $\quad \text{end}$
9. $\text{end}$

Figure 2.2: Modified TABUCOL pseudocode

### 2.3.3 PARTIALCOL: A Tabu Search with Incomplete Solutions

PARTIALCOL [BZ08] refers to a class of tabu search techniques for GCP based on (modified) TABUCOL. The search space is based on partial, proper solutions where only some vertexes are colored, but no conflicts exist. An allowed neighborhood move is a pair, (vertex, color), that assigns a color to a vertex. Since only proper solutions are allowed, any neighboring vertexes that are of the same color then become uncolored.
In addition to the differences in search space and neighborhood moves, three tabu tenure schemes are proposed: static, dynamic, and reactive. A static tabu tenure $tl$ corresponds to the equation $tl = TL$, where $TL$ is a constant. A dynamic tabu tenure is a function of the current solution; an example of a dynamic tabu tenure is $tl = \text{Random}(A) + \alpha \ast |O|$, which is the same as the tabu tenure considered in Section 2.3.2, where $nb_{CFL}$ has been replaced by $|O|$, the number of uncolored vertexes in the current solution. A reactive tabu tenure is a function of the entire search history, and is designed to make tabu search even more robust to local optima, or conversely more efficient within global optima.

The pseudocode for PARTIALCOL is therefore the same as the pseudocode for TABUCOL (see Figure 2.2), with a different neighborhood move structure and cost function.

In general, the original results for PARTIALCOL [BZ08] do not indicate any advantage to using the reactive tabu tenure. Thus, for the rest of this work, we consider only the dynamic tabu tenure analogous to that used in the modified TABUCOL. As with TABUCOL, we consider a single parameterization for PARTIALCOL, $A = 10$ and $\alpha = 0.6$.

2.3.4 HCA: A Memetic Algorithm

HCA [GH99], or “Hybrid Coloring Algorithm”, is a memetic algorithm. The pseudocode for HCA (Figure 2.3) can be interpreted as a basic description of any memetic algorithm.

HCA includes the original proposal for an effective crossover for graph coloring called Greedy Partition Crossover (GPX) [GH99] which greedily constructs new colorings by combining color classes of two existing solutions. By considering color classes independent of the color class labels, label symmetry is broken. The local search improvement algorithm used is the modified TABUCOL algorithm (see Section 2.3.2). If we assume the default settings for TABUCOL ($A = 10$, $\alpha = 0.6$), that leaves two additional parameters for HCA: $P$, the size of the population, and $L$, the maximum number of iterations allowed to TABUCOL during the child improvement phase. The pseudocode for both GPX and HCA are provided below in Figures 2.4 and 2.3 respectively.

Different values of $P$ and $L$ have a significant and complex impact on HCA’s performance.
Input: Graph $G = (V,E)$, integer $k$

Output: Last best solution $s^* = \{s^*_1, s^*_2, \ldots\}$, where elements $s^*_x \in [1, |V|]

1. $P \leftarrow \text{InitPopulation}(|P|)$
2. while !StopCondition() do
3.   $(s_1, s_2) \leftarrow \text{ChooseParents}(P)$
4.   $s \leftarrow \text{Crossover}(s_1, s_2)$
5.   $s \leftarrow \text{LocalSearch}(s, L)$
6.   $P \leftarrow \text{UpdatePopulation}(P, s)$
7.   if Cost$(s) \leq \text{Cost}(s^*)$ then
8.     $s^* \leftarrow s$
9.   end
10. end

Figure 2.3: HCA pseudocode

Larger values of $P$ tend to promote larger population diversity, and help avoid premature convergence [GH99]. In general, this translates to HCA being more robust, but slower with increasing values of $P$. Similarly, $L$ represents an explicit tradeoff between “intensification and diversification”, an important “exploration/exploitation” tradeoff which can dramatically affect the efficiency of metaheuristics [BR03]. The authors of the original HCA paper [GH99] also note that larger values of $L$ also promote population diversity by allowing child solutions to wander farther away from their parents. The end result is that different values of $P$ and $L$ are optimal for different instances and situations, which the authors of the original paper understood and exploited in their evaluation of HCA. In many situations, HCA was shown to be more efficient and robust than TABUCOL alone, depending on optimal parameter settings.

We later exploit this interesting $P$ and $L$ parameter space by including several configurations of HCA in our algorithm portfolio.

2.4 Machine Learning Algorithm

Next, we consider the machine learning approach. We are interested in a model that predicts the best algorithm for a problem instance based on some simple static features.
Input: Graph $G = (V, E)$, integer $k$, solutions $s_1 = \{V^1_1, \ldots, V^1_k\}$ and $s_2 = \{V^2_1, \ldots, V^2_k\}$. Each $V_i$ denotes a color class, i.e. a set of vertexes in the solution that are assigned the same color.

Output: The child solution $s = \{V_1, \ldots, V_k\}$

1. for $1 \leq l \leq k$ do
2.    if $l$ is odd then
3.        $A \leftarrow 1$
4.    else
5.        $A \leftarrow 2$
6.    end
7.    Choose $i$ such that $V^{A}_i$ has a maximum cardinality.
8.    $V_l \leftarrow V^{A}_i$
9.    Remove the vertexes of $V_l$ from $s_1$ and $s_2$.
10. end
11. Randomly assign colors to remaining uncolored vertexes of $s$.

Figure 2.4: GPX pseudocode

The problem could be viewed as a classification problem with a generalized loss function, where the loss associated with predicting algorithm $a$ for problem instance $p$ is the achieved solution quality of $a$ on $p$, subject to the maximum runtime constraint. Unfortunately, most machine learning classifiers only consider the common 0/1 loss function, which trivially does not account for ties, and more generally attempts to minimize the prediction error instead of the achieved solution quality as we desire. A simple alternative to 0/1 classification is to use a regression technique to predict the solution quality of each algorithm. Then, the algorithm with the minimum predicted solution quality is selected.

Any regression technique could be used in this strategy. Practical methods for selecting a particular machine learning technique for a problem typically involve a rough analysis of the data, some intuition regarding which machine learning techniques may be effective on the data, and a trial and error process, where ideally the final machine learning technique is selected with respect to a validation set.

We were fortunate in our research to identify an effective regression technique on our first try. We had some knowledge and intuition about the data to be modeled. We expect the solution quality to be a noisy function of the simple instance features, although this is
aided somewhat by including the solution quality of the DSATUR heuristic as a feature, as
will be discussed later. We generally expect some non-linear relationships between solution
quality and instance features. And we generally expect that, since we really only care
about the relative performance of each algorithm in order to perform algorithm selection,
our machine learning scheme might perform best if we use the same regression technique
for each algorithm, in order to minimize large prediction discrepancies (e.g. one regression
technique overfits, one underfits).

The regression technique used in this work is a regression tree named M5P [WW97],
a variant of M5 [Qui92]. A regression tree is a decision tree with regression models at its
leaves. An implementation of M5P is available through WEKA [HFH09], as mentioned in
Section 2.1.

Future work will explore various alternative machine learning techniques to perform al-
gorithm selection. Various alternatives to M5P and regression trees could be explored. Also,
although predictive models of an algorithm’s performance are extremely interesting for a
variety of algorithmic decisions, we expect a high statistical variance which makes accurate
prediction impossible. It seems logical that some kind of classification algorithm, which
has a much smaller space of predictions, will be able to perform much better in general. Of
course, such a classification algorithm will need to support a general loss function for reasons
discussed above.

2.5 Experimental Platform Details

All experiments were performed on an Intel® Core™ i7 CPU 1.73 GHz with 4 GB RAM.
The installed operating system was Windows 7 Professional, Service Pack 1, 64 bit. All
experiments were performed approximately between the months 2/2012 and 6/2012.
2.6 Selected Experiments

In this section, we select some experiments for the proposed algorithm selection methodology. This involves the general steps outlined in Section 1.2.2.6. Specifically, we select some problem instance distributions, evaluation functions, algorithms, and predictive features.

2.6.1 Common Selected Algorithms and Configurations

The following algorithms were selected for the algorithm portfolio in each experiment.

First, TABUCOL, with the default parameter values of $A = 10$ and $\alpha = 0.6$. Different values for the parameters were experimentally explored with no interesting improvements.

Next, PARTIALCOL, with the default parameter values of $A = 10$ and $\alpha = 0.6$. This is the only reported parameter setting for PARTIALCOL [BZ08], and it appears to be robust in practice.

Finally, six different configurations of HCA, with population sizes $P = \{5, 10\}$ and local search maximum iterations $L = \{10^3, 10^4, 10^5\}$. These six configurations were chosen to maximize the diversity per configuration. The spread of parameters roughly correlates to observed values in the original HCA paper [GH99] as well as a recent computational study [LTM12].

In total, eight algorithm configurations were selected.

2.6.2 Common Evaluation Function

As discussed in the initial proposal, our primary evaluation function will be minimizing the average solution colors (i.e., maximum solution quality) given a maximum runtime. Minimizing the solution colors appears to be a primary concern of current graph coloring research [MT10], so this evaluation function is reasonable, although a different method for aggregating results between instances of varying chromatic number could be used instead of directly averaging the solution colors. The largest maximum runtime considered is one hour, which is similar to the experimental maximum runtime used in other current graph coloring
Storage of intermediate GCP results (see Section 2.1), i.e. recording every time a solution with a new minimum color count is achieved, allows analysis of algorithm performance with all maximum runtime values up to one hour with a single GCP trial.

This evaluation function will be used in each of the following experiments.

2.6.3 Common Problem Instance Features

The following problem instance features were used in all of the following experiments:

1. The number of vertexes in the graph, \(|V|\).
2. The number of edges, \(|E|\).
3. The edge density, \(\frac{2|E|}{|V|(|V|-1)}\).
4. The average vertex degree, \(\mu_\delta = \frac{2|E|}{|V|}\).
5. The variance of the vertex degree, \(\sigma^2 = \sum_{v \in V} (\delta_v - \mu_\delta)^2\).
6. The standard deviation of the vertex degree, \(\sigma\).
7. The coefficient of variation of the vertex degree, \(\frac{\sigma}{\mu_\delta}\).
8. The number of colors used by DSATUR to color the problem instance, in one random trial.

Note that several features that can be mathematically derived from other features are included: a standard feature selection technique in machine learning. The inspiration to use the coefficient of variation is due to analysis performed in a recent computational comparison by Lewis et al. [LTM12].

The most conceptually interesting feature here is the output of the greedy constructive heuristic DSATUR. Intuitively, this should help predict the “empirical hardness” [XHH08] of the problem instance, and allow differential solution quality prediction in learned regression
functions. This is not a novel idea, as several other works involve the use of empirical algorithm performance on an instance as a measure of empirical hardness [LNS06, XHH08], or more generally the use of a limited “observation horizon” to enhance predictive power [KHR02].

Using greedy heuristics such as DSATUR and RLF as predictive features could result in significant prediction overhead that outweighs the benefits of algorithm selection, or at least should be included in the final evaluation. However, we note that the runtime of DSATUR and RLF is generally trivial, even on large graphs, which is due to their greedy formulations. On all considered problem instances in this research, up to $|V| = 2000$ and $|E| = 1,900,950$ (edge density of 0.95), DSATUR requires less than 0.2 seconds, and RLF requires less than 3.0 seconds (RLF is asymptotically sensitive to $|E|$ where DSATUR is not, which accounts for the efficiency gap). Also, we note that many graph coloring heuristic algorithms require an initial solution (or solutions) constructed by greedy heuristics like DSATUR and RLF, and therefore computation used to compute these features could be reused.

2.6.4 Random Graph Pool

The first considered problem instance distribution is generated using one of Joseph Culberson’s graph coloring problem instance generators [Cul12]. IID (independent and identically distributed) graphs with no hidden colorings are considered. In IID graphs, colloquially called “random graphs”, each pair of vertexes are considered in turn and assigned an edge with identical edge probability $p$. The only other required parameter is the number of vertexes in the graph, $|V|$. Instances were randomly generated with uniform random vertex counts $|V| = [500, 2000]$ and uniform random edge probabilities $p = [0.05, 0.95]$.

Experimental results and analysis for the proposed algorithm selection methodology are discussed in Section 3.1.
CHAPTER 3

Results and Analysis

In this chapter we present and analyze the experimental results of the proposed algorithm selection methodology on selected experiments outlined in Section 2.6. In all selected experiments, the algorithm methodology achieves near-optimal algorithm selection on previously unseen problem instances.

3.1 Random Graph Pool

First we present the results and analysis of the application of the proposed algorithm selection methodology on the random graph pool instance distribution discussed in Section 2.6.4.

All algorithms in the algorithm portfolio were run once with a maximum runtime of one hour. Trials were completed for a total of 300 problem instances. The total training time is therefore (300 instances * 8 algorithms * 1 hour) = 100 days. This training time was reduced through parallelization on a machine with eight cores (details in Section 2.5), resulting in an observed training time of 12.5 days.

To make the most of our limited data, the standard machine learning technique of cross-validation was used in order to experimentally evaluate the performance of our algorithm selection methodology. A 10-fold cross-validation was performed, and was repeated 100 times in order to reduce variance.

3.1.1 Results and Analysis

First, we analyze the algorithm selection strategy performance, referred to as “Predicted Algorithm” in the following figures, as the maximum runtime is increased from 30 seconds
Figure 3.1: The average number of colors over all instances as the maximum runtime is steadily increased from 30 seconds to 3600 seconds (one hour).

up to 3600 seconds (one hour). The standards for comparison are (i) the performance of a winner-take-all strategy, denoted by “Best Algorithm” in the following figures, and (ii) the performance of the optimal algorithm selection strategy performed by an oracle, denoted by “Optimal Algorithm” in the following figures.

Figure 3.1 displays the average performance (achieved minimum colors) with respect to the maximum runtime. Figure 3.2 displays the same graph normalized to the performance of the winner-take-all strategy, which better displays the reduced cost compared to the winner-take-all strategy.

Looking at Figure 3.1, it appears that even at one hour we have not yet converged to the best solution quality achievable by algorithms in the algorithm portfolio. This is likely due to the presence of large graphs in the distribution, with $|V| > 1000$.

Clearly both the optimal and predicted algorithm selection strategies are improving on
Figure 3.2: Figure 3.1, normalized to the performance of the “Best Algorithm” strategy.

In Figure 3.2, there are several interesting features. First, we note that the performance of the algorithm selection strategy almost always improves upon the performance of the “winner-take-all” strategy. The only exception is at very small runtimes where the largest variation in performance occurs. However, a simple cross-validation scheme that considers a winner-take-all strategy as an alternative to the selection strategy would resolve this negative performance.

Second, we note that the optimal algorithm selection strategy achieves significant performance gains with respect to the winner-take-all strategy, with a minimum average color improvement at (967 seconds, -1.337 average colors). The general “phase-transition” observed in Figure 3.2 can be explained by some simple intuition regarding our algorithm
portfolio. In our algorithm portfolio, there are several “speedy” heuristics which are very efficient at “easy” values of $k$, but not at hard values. Their efficiency either decreases significantly at hard values of $k$, or they run forever, cycling. In our algorithm portfolio, TABUCOL and PARTIALCOL fit this description. In contrast, there are several “robust” algorithms in our portfolio that are not very efficient at easy values of $k$, but their efficiency decreases more steadily as $k$ gets harder, allowing their performance to eventually surpass the performance of the speedy algorithms for hard values of $k$. In our algorithm portfolio, the various configurations of HCA fit this description, with larger $P$ and $L$ values generally resulting in more robust algorithms.

This dichotomy explains the “phase-transition” in the following way. When the maximum runtime is small, the algorithms only have time to explore easy values of $k$. Therefore, the speediest algorithm will dominate. Figure 3.3 (discussed below) indicates that the speediest algorithm in this scenario is PARTIALCOL. In contrast, when the maximum runtime is large, the most robust algorithm will dominate, since the runtime at which the best solution is achieved is not a part of the evaluation function. Figure 3.3 indicates that the most robust algorithm in this scenario is HCA, with $P = 5$ and $L = 10^4$. However, in between the two extremes of small and large maximum runtimes exists a transition region where some instances can be explored to easy values of $k$ and others can be explored to hard values of $k$. In this region, optimal algorithm selection should have the greatest opportunity for improvement over a winner-takes-all strategy.

Returning to Figure 3.2 we now consider how well our algorithm selection strategy performs in comparison to the optimal algorithm selection strategy. Throughout the entire considered maximum runtime domain, our algorithm selection strategy performs nearly as well as the optimal algorithm selection strategy, lagging in performance improvement by a small, relatively constant amount of 0.2 average colors. At the global minimum for both our algorithm selection and optimal algorithm selection (at approximately 967 seconds), optimal algorithm selection achieves approximately 1.337 average colors improvement and our algorithm selection achieves approximately 1.013 average colors improvement.

This is a fairly impressive result considering the small set of instance features identified.
and the use of a static algorithm selection strategy that does not react to any dynamic algorithm performance metrics, such as time since last improvement. The remaining gap between our algorithm selection strategy and the optimal selection strategy can be explained by several factors. First, the optimal selection strategy is computed from only one trial of each algorithm. This one trial is a weak statistical estimator for an algorithm’s expected performance. Additionally, in the context of the optimal algorithm selection strategy, this statistic is biased when conditioned on the information that it performed better than the other algorithms. So in general, the optimal algorithm selection strategy should perform somewhat worse in expectation. Second, since only a limited (and simple) set of features are supplied to the machine learning algorithm to perform algorithm selection, some of the performance gap is theoretically unachievable without additional features. Lastly, it is possible that an alternative machine learning approach would perform better.

In order to see the impact of each algorithm on the algorithm portfolio, two additional figures are supplied. Figure 3.3 displays the average performance of each individual algorithm with respect to the maximum runtime. Figure 3.4 displays the fraction of instances in which each algorithm in the algorithm portfolio is selected by the algorithm selection strategy, again with respect to the maximum runtime.

Figure 3.3 illustrates the evolution of the best algorithm in the winner-take-all strategy with respect to maximum runtime. As discussed above, there are several “speedy” algorithms. PARTIALCOL and TABUCOL, which are pure stochastic local search algorithms, fit this description, although on this instance distribution TABUCOL appears to be more robust on average. We also note the presence of “robust” algorithms, in this case the various parameterizations of HCA. Although we expect the most robust configurations of HCA to have the largest values of $P$ and $L$, it would appear that maximum runtime would have to significantly larger than one hour for this to be true on the given instance distribution.

Figure 3.4 is interesting for several reasons. First, we observe that our algorithm selection strategy uses a wide variety of algorithms to achieve its good performance. Although $HCA(P = 5, L = 10^4)$ dominates on the second half of the figure, it only accounts for roughly 60% of selected algorithms, and that percentage appears to be somewhat steady.
Figure 3.3: The average number of colors of each algorithm over all instances.

as the maximum runtime increases. This reflects positively on both the possible benefits of an algorithm selection strategy as well as the particular set of algorithms chosen for our algorithm portfolio.

The second interesting feature of Figure 3.4 begins with the observation that two algorithms in particular are rarely selected: HCA($P = 5$, $L = 10^5$) and HCA($P = 10$, $L = 10^5$). As discussed above, we predict these algorithms will become more important to the algorithm selection strategy as the maximum runtime increases beyond one hour, but in the context of this figure we are interested in why these two algorithms are occasionally selected near the minimum values of maximum runtime. This can be explained by two factors. First, at small values of maximum runtime, the initial $10^5$ local search iterations may well be all that is required for considered values of $k$. Second, our implemented algorithm selection strategy does not consider any tie-breaking strategies. In particular, two algorithms could have the same expected runtime on an instance, but one could have better average performance on the entire instance distribution, or one could have a very large variance. In this scenario, we
would like to select the algorithm with better average performance on the instance distribution in case the instance is atypical of the computed instance features. We would also like to select the algorithm that has smaller variance to reduce risk. Tie-breaking strategies for algorithm selection are therefore likely to be an important part of future algorithm selection strategies, especially when prediction uncertainty is high.

Lastly, we consider what fraction of the total training instances are necessary to achieve the reported performance, and also extrapolate what performance might be available in the limit of a larger number of training samples. To do so, we perform the same cross-validation strategy, but supply only a fraction of the training instances to the machine learning algorithm. The results are plotted in Figure 3.5, and are in the same normalized format as Figure 3.2.

In Figure 3.5, several contours are provided, corresponding to exponentially decreasing fractions of the total available training instances (in a 10-fold cross validation with 300
samples, the maximum available training instances is 270). The performance of the machine learning algorithm with few training samples again illustrates the need for a cross-validation strategy that at least considers the winner-take-all strategy. Interestingly, it appears that the final performance in the limit of large training sample size is achieved somewhere between 33 and 67 training samples. This is approximately one fifth of the total training instances available, which corresponds to a greatly decreased training burden of about 20 days instead of 100.

### 3.1.2 Summary

The results on the random graph pool are encouraging. Our algorithm selection methodology, although simple, achieves performance close to the optimal algorithm selection strategy. This performance gain turns out to be non-trivial (1.013 average colors when the maximum runtime is 967 seconds).
3.1.3 Future Analyses

Some future analyses we would like to perform on the collected data include the following.

We would like to analyze the empirical impact that a one trial per instance training strategy makes on achieved performance (recall Section 1.2.2.5, also see [Bir09]). Some variable number of trials, \( x \), could be run on the generated instances. Then, given a maximum total training time, the performance of the algorithm selection strategy using one trial per instance and the performance using \( x \) trials per instance could be compared.

We would also like to analyze how well the constructed predictive models apply to common graph coloring benchmarks. Since the benchmark instances and the training instances are not described by the same instance distribution, there are no statistical performance guarantees on the benchmark instances. Generally this is referred to as “transfer learning” and “inductive transfer” in machine learning research. Since we already have collected run-time data for the algorithm portfolio on the benchmarks for multiple trials (see Section 1.2.3), no further experimental data would be necessary.

Various feature selection strategies could be performed to improve the performance of the algorithm selection strategy. This would also allow more confident inclusion of a larger number of features. WEKA (see Section 2.1) includes several feature selection strategies to experiment with.

Other machine learning algorithm alternatives to using M5P as an underlying regression model could be explored. More generally, we would like to explore other machine learning methodologies for algorithm selection such as a classification approach that supports a generalized loss function. WEKA (see Section 2.1) includes a large variety of machine learning algorithms to experiment with.
CHAPTER 4

Conclusion and Future Work

In conclusion, our proposed algorithm selection methodology was very successful in the performed experiments, achieving near-optimal algorithm selection on a set of previously unseen problem instances.

This success was made possible by several primary factors. First and foremost was the existence of a large opportunity for improvement over a winner-take-all strategy on the experimental instance distribution. Second was the effectiveness of our relatively simple algorithm selection methodology. Last was the empirically determined statistical predictability of the underlying heuristic algorithms, which enabled intelligent and effective algorithm selection beyond a simple winner-take-all strategy.

In light of this, our proposed methodology successfully addressed the main issues in current graph coloring heuristic algorithm research outlined in Section 1.2.1, at least in part.

1. Ad-hoc parameter setting can be partially addressed by implementing our proposed methodology and including several diverse parameterizations of the algorithm in the algorithm portfolio. Unfortunately, the question of how to optimally identify these diverse parameterizations is not addressed by this research.

2. The over-tuning issue associated with tuning and evaluating an algorithm on the same set of instances is addressed in our proposed methodology by adhering to sound principles taken from the field of machine learning.

3. Our proposed algorithm selection methodology proved to be a successful alternative to the winner-take-all approach, as demonstrated by significant performance improvement on experimental problem instance distributions.
Although our proposed algorithm selection methodology was successful, there are many opportunities for future work, in addition to those outlined in Section 3.1.3.

Experiments with other instance distributions would be beneficial. In general, application of the proposed algorithm selection methodology to other applications that are limited by winner-take-all approaches and parameterization issues could be explored.

In this work, the application required serious computational resources due to the significantly long runtime of trials. This may be a limiting factor of some applications which require the running of algorithms for days or even weeks on a particular instance. One strategy of addressing this limitation would be the ability to generalize (make predictions from) observed algorithm performance to longer runtimes. Another strategy is to use online or dynamic methodologies similar to those proposed by Gagliolo and Schmidhuber in [GS06].

In a similar vein, training burden may be reduced by maximizing the information gained in training trials. As an example, racing algorithms [Bir09] stop running training trials for an algorithm once its performance becomes statistically worse than the best performing algorithm on an instance distribution. While we cannot directly use this trick, other methodologies for optimal training could be explored. Promising areas include the field of active learning (training on discriminative problem instances, see [Set09]) and confidence machines (determining the confidence of a prediction, i.e. the probability that it is accurate, see [PNV02]).

Lastly, future work in general will include searching for opportunities in applications requiring heuristic solutions where statistical modeling can be profitably applied.
References


52


