On Portfolios for Backtracking Search in the Presence of Deadlines

Huayue Wu and Peter van Beek
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{hwu, vanbeek}@uwaterloo.ca

Abstract

Constraint satisfaction and propositional satisfiability problems are often solved using backtracking search. Previous studies have shown that portfolios of backtracking algorithms—a selection of one or more algorithms plus a schedule for executing the algorithms—can dramatically improve performance on some instances. In this paper, we consider a setting that often arises in practice where the instances to be solved arise over time, the instances all belong to some class of problem instances, and a limit or deadline is placed on the computational resources that the backtracking algorithm can consume in solving any instance. For such a scenario, we present a simple scheme for learning a good portfolio of backtracking algorithms from a small sample of instances. We demonstrate the effectiveness of our approach through an extensive empirical evaluation on a real-world instruction scheduling testbed.

1. Introduction

Constraint satisfaction and propositional satisfiability problems are often solved using backtracking search. In this paper, we consider a setting where the instances to be solved arise over time, the instances all belong to some class of problem instances, and a limit or deadline is placed on the computational resources that the backtracking algorithm can consume in solving any instance. Such a setting often arises in practice. For example, a common scenario in scheduling and rostering is that at regular intervals on the calendar a similar scheduling problem must be solved and a schedule is useful only if it is found within some deadline. For a further example, in our evaluation testbed of instruction scheduling, thousands of instances arise each time a compiler is invoked on some software project and a limit needs to be placed on the time given for solving each instance in order to keep the total compile time to an acceptable level.

Since the first formal statements of backtracking algorithms over 40 years ago [3, 7], many techniques for improving the efficiency of a backtracking search algorithm have been suggested and evaluated (see, e.g., [26]). Unfortunately, even with these improvements, any given backtracking algorithm can still be quite brittle, performing well on some instances but poorly on other seemingly similar instances. To reduce the brittleness or variability in performance of any single algorithm, portfolios of multiple algorithms have been proposed and shown to dramatically improve performance on some instances (e.g., [10, 14, 17, 23]).

Given a set of possible backtracking algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ and a time deadline d, a *portfolio* P for a single processor is a finite sequence of pairs,

$$P = [(\mathcal{A}_{k_1}, t_1), (\mathcal{A}_{k_2}, t_2), \dots, (\mathcal{A}_{k_m}, t_m)],$$

where each $\mathcal{A}_{k_i}$ is a backtracking algorithm, each t_i is a positive integer, and $\Sigma_{i=1}^{m} t_i = d$. To apply a portfolio to an instance, algorithm $\mathcal{A}_{k_1}$ is run for t_1 steps. If no solution is found within t_1 steps, algorithm $\mathcal{A}_{k_1}$ is terminated and algorithm $\mathcal{A}_{k_2}$ is run for t_2 steps, and so on until either a solution is found or the sequence is exhausted as the time deadline d has been reached. A *fixed cutoff* portfolio is a portfolio where all of the t_i's are equal. An *algorithm selection* portfolio is a portfolio where,

$$P = [(\mathcal{A}, d)];$$

i.e., a single algorithm is selected and run until either a solution is found or the deadline is reached. A *restart strategy* portfolio is a portfolio where,

$$P = [(\mathcal{A}, t_1), (\mathcal{A}, t_2), \dots, (\mathcal{A}, t_m)];$$

i.e., the same algorithm is continually restarted until either a solution is found or the deadline is reached [10, 14, 23]. Of course, for a restart strategy to make sense, $\mathcal{A}$ must be a non-deterministic algorithm. The usual method for randomizing a backtracking algorithm is to randomize the variable or value ordering heuristic (e.g., [11, 14]).

A portfolio can be either *instance-based* in that it is intended or tailored to be used on a specific instance, or it can be *class-based* in that the same portfolio is intended to be used on any instance from a problem class. The question that we address in this paper is, given a class of problem instances and a deadline d, can we learn a good class-based portfolio from a small representative sample of instances?

We present a simple scheme for learning a good portfolio of backtracking algorithms in the presence of deadlines. In contrast to previous work, where the differences in the possible backtracking algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ often involves the variable ordering heuristic, we create variability in solving performance by increasing levels of constraint propagation from light-weight to heavy-weight.

The learning is done in an offline manner from a training set and the portfolio that is learned is then used on unknown instances in the future when the application is deployed. The portfolios that we learn only allow the sequential application of algorithms rather than running algorithms concurrently and they do not allow the execution of an algorithm to be suspended and later resumed; i.e., in our portfolios an algorithm is terminated if it fails to find a solution within the specified number of time steps. Because of the presence of a deadline and because we focus on simpler portfolios, we can systematically and efficiently search for a portfolio that has a low cost on the training set. One key to the success of our approach is that we find a portfolio which is *robust* on the training set in that it gives the lowest cost and is furthest away from any portfolio which leads to poorer results.

We demonstrate the effectiveness of our approach through an extensive empirical evaluation on a real-world instruction scheduling testbed. The portfolio is learned from a small training set and then evaluated on a large test set to estimate its performance on unknown instances in the future. We show that on our testbed, the class-based portfolio that is learned can significantly outperform a restart strategy portfolio which uses an *oracle* to always select the best restart cutoff for each instance, and can approach the performance of an algorithm selection portfolio which uses an *oracle* to always select the best algorithm for each instance. In practice, any algorithm selection method at the instance-level cannot be perfect. Once an algorithm selection method makes even a very small percentage of mistakes, the portfolio learned by our methodology exceeds the performance of the algorithm selection method.

2. Related work

In this section, we discuss related work on portfolios. We categorize previous work into general portfolios, algorithm selection portfolios, and restart strategy portfolios. Our focus is on single processor portfolios of backtracking algorithms.

2.1. General portfolios

Huberman, Lukose, and Hogg [17] may have been the first to coin the term portfolios of algorithms, drawing an analogy to financial portfolios. They consider portfolios of multiple copies of a single backtracking algorithm with a randomized variable ordering and show that one can trade-off performance and risk on a given instance. Their work is important for introducing and giving a preliminary demonstration of the effectiveness of the portfolio approach.

Gomes and Selman [8, 9] perform an extensive empirical validation of the portfolio approach for portfolios of multiple backtracking algorithms that differ in their randomized variable ordering. Their work clearly demonstrates that a portfolio approach can give important performance gains in the (usual) case where no single backtracking algorithm dominates across all instances. Gomes and Selman provide several general guidelines for designing portfolios, but a more practical approach for doing so is left as an open question [9, p. 44].

Lagoudakis and Littman [19] present a method for constructing a portfolio of backtracking algorithms (each with a different variable ordering heuristic) using techniques from reinforcement learning. However, in an empirical evaluation, the performance of the learned portfolio was often not better than the best algorithms by themselves.

Finkelstein, Markovitch, and Rivlin [4] present a procedure for constructing class-based single processor portfolios. Their method requires a performance profile for each possible algorithm which specifies the probability that a solution will be found by the algorithm as a function of time. Given these performance profiles, the construction of a portfolio is formulated as an optimization problem and solved by branch-and-bound search. Their method is elegant but suffers from a high computational cost. As well, the method has only been evaluated under the assumption that the instances are homogeneous and it is unclear how their method would perform under heterogeneous instances. For example, in their experiments on Latin square completion, profiles were constructed from 50,000 instances and all instances were of the same size and contained the same number of pre-assigned squares. In contrast, our approach has a relatively low computational cost and we demonstrate its applicability in scenarios where the problem class contains heterogeneous instances.

Gagliolo and Schmidhuber [5] also consider class-based single processor portfolios. In their method, all algorithms are run in a time-sharing fashion and the allocation of time to each algorithm is dynamically updated as instances are solved. However, their proposal relies on estimates of the time still needed by a backtracking algorithm to complete—a difficult task as the performance of backtracking algorithms can be very unpredictable.

2.2. Algorithm selection portfolios

Lobjois and Lemaître [22] examine instance-based algorithm selection in the setting of branch-and-bound algorithms for optimization problems. Carchrae and Beck [2] argue for a "low-knowledge" approach that does not require expertise in the problem structure and in the algorithms in order to devise features for building predictive models. They show good success with features that are common to all optimization problems. The techniques developed in these papers are specific to optimization problems and do not carry over in any straight-forward manner to our context of constraint and propositional satisfiability.

Leyton-Brown et al. [20, 21] propose a methodology for instance-based algorithm selection in the setting of backtracking algorithms. Their methodology requires gathering performance profiles for each algorithm in the set of possible backtracking algorithms by applying the algorithm to a large collection of instances. Features of the problem structure and of algorithm performance are then identified and regression techniques are used to learn a function of the features to predict the running time (see also [16]). Once one can predict the running time of each algorithm on an instance, one can then choose the best algorithm for that instance. Guerri and Milano [13], in followup work, use decision trees to select between two algorithms and report a 90% selection accuracy.

In our work we show that the class-based general portfolio that is learned by our methodology can approach the performance of a *perfect* instance-based algorithm selection portfolio. And if an algorithm selection method makes even a small percentage of mistakes—as it would in practice—our learned portfolio would outperform the algorithm selection portfolio.

2.3. Restart strategy portfolios

Luby, Sinclair, and Zuckerman [23] examine restart strategies in the more general setting of Las Vegas algorithms. A Las Vegas algorithm is a randomized algorithm that always gives the correct answer when it terminates, however the running time of the algorithm varies from one run to another and can be modeled as a random variable. Let $f(t)$ be the probability that a backtracking algorithm $\mathcal{A}$ applied to instance x stops after taking exactly t steps; $f(t)$ is referred to as the runtime distribution of algorithm $\mathcal{A}$ on instance x. Luby, Sinclair, and Zuckerman show that, given full knowledge of the runtime distribution of an instance, the optimal restart strategy for that instance is given by $[(\mathcal{A}, t), (\mathcal{A}, t), \ldots]$, for some fixed cutoff t.

Of course, the runtime distribution of an instance is not known in practice. As a result, there have been various proposals for learning portfolios that may be sub-optimal but still have good performance (e.g., [6, 18, 25]). As well, there have been extensive empirical evaluations of restart strategy portfolios for backtracking algorithms with various randomized heuristics (e.g., [11, 12, 15]). Interestingly, Gomes and Selman [9] conclude from their experiments on general portfolios of backtracking algorithms that, if only a single processor is available, using a restart strategy is often the best portfolio.

However, in our work we show that in the presence of deadlines the class-based general portfolio that is learned by our methodology can significantly outperform the optimal instance-based restart strategy portfolio. In other words, one portfolio applied to every instance can perform better than the case where one is assumed to know the runtime distribution of each instance and is permitted to pick the optimal restart strategy portfolio for each instance.

3. Portfolio design

In this section, we present our simple methodology for learning a good portfolio in the presence of deadlines. We are considering a scenario where instances from a problem class are to be solved over time and we are to learn a good class-based portfolio in an offline manner.

Step 1: Construct possible backtracking algorithms. To begin, we must decide on the possible backtracking algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \ldots\}$ from which to construct a portfolio. In previous work, the possible backtracking algorithms often differ only in their variable ordering heuristic. However, weakening or changing the variable ordering heuristic sometimes leads to dominated algorithms (algorithms which have poorer performance across all or almost all instances) and does not always lead to an increase in variability of problem solving performance (where different algorithms perform well on different instances). This is particularly true in cases where the variable ordering has been carefully crafted, as is often done in real applications. It is the variability in algorithm performance across instances which a portfolio can take advantage of in order to improve on individual algorithms. In our work, we propose to create variability in performance by increasing the level of constraint propagation from light-weight to heavy-weight propagation. Constraint propagation is the active use of the constraints to prune parts of the search space (see, e.g., [1, 26]). The technique is general in the sense that any backtracking algorithm that incorporates a base level of constraint propagation (which all successful backtracking algorithms do) can easily be modified to incorporate a higher level of constraint propagation through the use of a technique called singleton consistency where the base constraint propagator is repeatedly called [1]. As well, constraint programming systems and libraries invariably have options for specifying the level of propagation enforced on individual constraints.

Step 2: Construct training and test set. We next construct (as in machine learning) a representative sample for learning the portfolio (a training set) and a representative sample for estimating how well the portfolio would work when deployed (a test set). Each possible backtracking algorithm is run on each instance in the training and test sets and the performance data is collected. The algorithms are run with the same deadline or limit on CPU time as will be used when the portfolio is deployed.

Step 3. Simulate possible portfolios on training set. We next systematically step through the possible portfolios at an appropriate level of abstraction and simulate each portfolio on the training set and record its performance. To reduce the computation, the portfolios that we learn are restricted to only allow the sequential application of algorithms rather than running algorithms concurrently and they do not allow the execution of an algorithm to be suspended and later resumed; i.e., in our portfolios an algorithm is terminated if it fails to find a solution within the specified number of time steps. As well, we further restrict the portfolios by mandating that the algorithms can appear in the portfolio only in order of increasing level of propagation. Because of the presence of a deadline and because we focus on simpler portfolios, this step can be done efficiently. For example, in our experiments, if the deadline was 10 minutes and we examined portfolios in units of 1 second, it took approximately 9.0 seconds of CPU time to examine all of the possible portfolios.

Note that in the methodology of Finkelstein, Markovitch, and Rivlin [4], this step would be omitted as the performance data from Step 2 would be summarized in a performance profile and used to select a best portfolio. However, this summary step, while perhaps more elegant than a brute-force approach, loses information and may not be applicable in the case of heterogeneous instances. To see this, suppose that two algorithms have identical performance profiles but (a) the algorithms' performance on instances is completely uncorrelated and (b) their performance is completely correlated. Finkelstein et al.'s approach would determine the same portfolio in both cases, whereas our approach would not.

As well, note that if our performance measure is the number of instances that are solved by a portfolio, restricting the portfolios in the way we have done provably has no affect. However, if our performance measure is the total runtime needed to apply the portfolio to each instance, restricting the portfolios may have an affect; i.e., there can exist a lower cost suspend-and-resume portfolio or a lower cost portfolio where the algorithms are applied in a different order. However, on our testbed, we found that the restrictions had only a minimal impact on the cost.

Step 4. Select best portfolio. We next select the best portfolio by choosing the portfolio that minimizes the performance measure on all of the instances in the training set and is furthest away in distance from any portfolio with poorer results. The aim of this criteria is to find a low-cost and robust portfolio and it is one of the keys to the success of our approach. More precisely, the lowest cost portfolios on the training set are bounded by a region (the coordinates of the region are the time limits allotted to each algorithm in the portfolio) and we are finding the centroid of the region. The centroid is easily determined given the performance of the possible portfolios recorded in Step 3.

Step 5. Evaluate selected portfolio on the test set. The performance of the portfolio on the test set gives an estimate for how well the portfolio would work when deployed and applied to unknown instances from the same problem class.

4. Experimental evaluation

In this section, we present the results of an empirical evaluation of our approach. We performed two sets of experiments. In the first set of experiments, we consider learning a good portfolio when the performance measure is the number of instances which are not solved. In the second set of experiments, we consider learning a good portfolio when the performance measure is the expected time to solve the instances. We begin by presenting the experimental setup that is in common to the two sets of experiments, followed by the results of the experiments themselves.

4.1. Experimental setup

We used instruction scheduling problems for multiple-issue pipelined processors in our experiments. Multiple-issue and pipelining are two techniques for performing instructions in parallel and processors which use these techniques are now standard in desktop and laptop machines. In such processors, there are multiple functional units and multiple instructions can be issued (begin execution) in each clock cycle. Examples of functional units include arithmetic-logic units (ALUs), floating-point units, memory or load/store units which perform address computations and accesses to the memory hierarchy, and branch units which execute branch and call instructions. On such processors, the order that the instructions are scheduled can significantly impact performance and instruction scheduling is one of the most important steps in improving the performance of object code produced by a compiler. The task is to find a minimal length schedule for a basic block—a straight-line sequence of code with a single entry point and a single exit point—subject to precedence, latency, and resource constraints.

We formulated a constraint programming model for the instruction scheduling problem and solved instances using backtracking search. In constraint programming, a problem

Table 1. Portfolios used in experimental evaluation.

$P_1(i) = \left[(\mathcal{A}_i^{rand}, t^*), \ldots, (\mathcal{A}_i^{rand}, t^*)\right]$, $i = 1, 2$, or 3.	Gold standard for restart strategies applied to algorithm $\mathcal{A}_i^{rand}$. For a given instance, choose the fixed cutoff $t^* \in \{1, 2, \ldots\}$ that is optimal for that instance and $\mathcal{A}_i^{rand}$ (i.e., the optimal fixed cutoff for the runtime distribution for the algorithm applied to the instance) and then apply the restart strategy to $\mathcal{A}_i^{rand}$ until a solution is found or the deadline d is reached.
$P_2(i) = \left[(\mathcal{A}_i^{rand}, d)\right]$, $i = 1, 2$, or 3.	Execute algorithm $\mathcal{A}_i^{rand}$ until a solution is found or the deadline d is reached.
$P_3(i) = \left[(\mathcal{A}_i^{det}, d)\right]$, $i = 1, 2$, or 3.	Execute algorithm $\mathcal{A}_i^{det}$ until a solution is found or the deadline d is reached.
$P_4 = \left[(\mathcal{A}^{det}, d)\right]$	Gold standard for algorithm selection at the instance level. For a given instance, choose the best deterministic algorithm $\mathcal{A}^{det} \in \{\mathcal{A}_1^{det}, \mathcal{A}_2^{det}, \mathcal{A}_3^{det}\}$ for that instance and execute the algorithm until a solution is found or the deadline d is reached.
$P_5 = \left[(\mathcal{A}_1^{det}, t_1), (\mathcal{A}_2^{det}, t_2), (\mathcal{A}_3^{det}, t_3)\right]$	Portfolio learned from the training set using the proposed methodology (see Section 3).

is modeled by specifying constraints on an acceptable solution, where a constraint is simply a relation among several unknowns or variables, each taking a value in a given domain. In our model, there is a variable for each instruction, the domains of the variables are the time cycles in which the instruction could be scheduled, and the constraints consist of linear inequalities, global cardinality constraints, and other specialized constraints. For example, the global cardinality constraints ensure that the number of instructions of each type issued at each clock cycle does not exceed the number of functional units of that type. The scheduler was able to solve almost all of the basic blocks that we found in practice, including basic blocks with up to 2600 instructions. We refer the reader to [24] for further details on the problem, the model, and the backtracking algorithm. The point we wish to emphasize here is that, prior to our examination of portfolios, considerable effort went into improving the constraint programming model, the backtracking algorithm, and the variable order heuristic to reduce the number of unsolved instances and to reduce the computation time.

For our experiments, we used scheduling instances that arise from the SPEC 2000 and MediaBench benchmark suites, two standard real-world benchmarks in compiler research. These benchmark suites consist of source code for software packages that are chosen to be representative of a variety of programming languages and types of applications. We had available to us a total of 6,377 hard scheduling instances from 28 different software packages. For our sample of instances, or training set, we used all 927 of the hard scheduling instances from the galgel, gap, mpeg, and jpeg software packages. We chose these four software packages for two reasons. First, the instances give approximately fifteen percent of the scheduling instances and, second, these software packages provide a good cross section of the data as they include both integer and floating point instructions as well as a variety of programming languages and types of applications. For our test set, we used the remaining 5,450 hard scheduling instances. Our training and test sets are heterogeneous as the instances arise from different software applications and different high-level programming languages, and the sizes of the instances range from as few as five instructions to as many as 2,598 instructions.

For our experiments, we used a deterministic backtracking algorithm capable of performing three levels of constraint propagation:

Level = 1	bounds consistency,
Level = 2	singleton bounds consistency, and
Level = 3	singleton bounds consistency to a depth of two.

We chose bounds consistency—instead of the more usual arc consistency—as in our problem it is equivalent but more efficient. In bounds consistency, one ensures that each upper and lower bound of the domain of a variable is consistent with each constraint. In singleton bounds consistency, one temporarily assigns a value to a variable and then performs bounds consistency. In singleton bounds consistency to a depth of two, one temporarily assigns a value to a variable and then performs singleton consistency. In each, if the value is found to be inconsistent it is not part of any solution and can be removed from the domain of the variable.

In order to study restart strategy portfolios, we randomized the backtracking algorithm by having the variable ordering heuristic randomly picked a variable from the top 5 variables (or fewer, if there were fewer variables left).

Table 2. Expected number of scheduling instances in the test set *not* solved within a deadline d—one second, ten seconds, one minute, ten minutes, one hour, and ten hours, respectively—for various portfolios of algorithms.

	Randomized						Deterministic				
d	$P_1(1)$	$P_1(2)$	$P_1(3)$	$P_2(1)$	$P_2(2)$	$P_2(3)$	$P_3(1)$	$P_3(2)$	$P_3(3)$	P_4	P_5
1s	158.6	120.7	823.5	223.5	124.3	823.6	201	128	824	64	74
10s	135.3	59.1	364.8	210.9	65.5	364.9	195	64	368	32	46
1m	124.6	35.0	158.7	201.9	42.5	158.8	188	40	161	18	30
10m	118.6	25.7	84.3	193.2	33.9	85.0	175	32	87	16	17
1h							171	31	53	15	15
10h							157	31	27	14	14

Table 3. Percentage increase in the expected number of scheduling instances in the test set *not* solved within a deadline d relative to the gold standard portfolio P_4, for various portfolios of algorithms.

	Randomized						Deterministic				
d	$P_1(1)$	$P_1(2)$	$P_1(3)$	$P_2(1)$	$P_2(2)$	$P_2(3)$	$P_3(1)$	$P_3(2)$	$P_3(3)$	P_4	P_5
1s	147.8%	88.6%	1186.7%	249.2%	94.2%	1186.9%	214.1%	100.0%	1187.5%	0%	15.6%
10s	322.8%	84.7%	1040.0%	559.1%	104.7%	1040.3%	509.4%	100.0%	1050.0%	0%	43.8%
1m	592.2%	94.4%	781.7%	1021.7%	136.1%	782.2%	944.4%	122.2%	794.4%	0%	66.7%
10m	641.3%	60.6%	426.9%	1107.5%	111.9%	431.3%	993.8%	100.0%	443.8%	0%	6.3%
1h							1040.0%	106.7%	253.3%	0%	0.0%
10h							1021.4%	121.4%	92.9%	0%	0.0%

This gave a total of six distinct backtracking algorithms $\mathcal{A}_i^s$, where $i = 1, 2, 3$ indicates the level of constraint propagation and $s \in \{det, rand\}$ indicates whether the algorithm is deterministic or has been randomized.

We ran each of the six algorithms on each instance in the training and test sets and recorded the performance data. If the algorithm was randomized, we collected 1000 samples of its runtime distribution by each time running the randomized backtracking algorithm on the instance with a different random seed and recording the amount of time taken in seconds. The samples are censored in that we ran the backtracking algorithm with a timeout mechanism; if the instance was not solved within the deadline, the backtracking algorithm was terminated and the maximum amount of time was recorded. The empirical runtime distributions and performance data were then used to learn and test various portfolios. The portfolios that we examined in the experiments are summarized in Table 1. When portfolios involved randomized algorithms, the statistics that we report are the average over 10,000 experiments.

All of the runtime experiments were performed on a cluster which consists of 768 machines running Linux, each with 4 GB of RAM and four 2.2 GHz processors.

4.2. Experiment 1

In our first set of experiments, we determined whether the portfolio learned using our methodology is effective at reducing the number of problems which are not solved in the presence of various deadlines. We used deadlines of one second, ten seconds, one minute, ten minutes, one hour, and ten hours, respectively. Table 2 summarizes the results. Table 3 presents the same information except now stated in terms of percentage change. Because of the high number of samples needed for the randomized algorithms to ensure significance (1000 samples) and the associated high computational cost, we did not collect runtime distributions for the larger timeouts (one hour and ten hours) and thus some of the entries for the portfolios of randomized algorithms were not determined and are left blank in the tables.

On this testbed, portfolio P_5—the portfolio learned by our approach—-performs well. It is interesting to note that

Table 4. Total time (sec.) for *all* of the scheduling instances in the test set, where each instance was either solved within the deadline d or the deadline was reached, for various portfolios of algorithms.

	Randomized			Deterministic				
d	$P_1(1)$	$P_1(2)$	$P_1(3)$	$P_3(1)$	$P_3(2)$	$P_3(3)$	P_4	P_5
1s	188.2	227.8	1,197.7	223.7	217.6	1,133.4	91.5	121.4
10s	1,455.9	922.1	5,598.5	2,003.7	968.8	5,594.5	481.7	712.2
1m	8,362.3	3,006.3	16,561.6	11,635.0	3,370.8	16,706.0	1,694.6	2,269.9
10m	73,723.3	17,845.9	72,956.9	109,493.6	21,483.0	73,323.5	10,524.8	16,458.3
1h				625,797.8	115,461.0	263,237.5	55,878.5	82,841.8
10h				5,848,373.4	1,119,861.0	1,329,250.2	510,022.7	565,111.7

Table 5. Percentage increase in the total time (sec.) for *all* of the scheduling instances in the test set relative to the gold standard portfolio P_4, for various portfolios of algorithms.

	Randomized			Deterministic				
d	$P_1(1)$	$P_1(2)$	$P_1(3)$	$P_3(1)$	$P_3(2)$	$P_3(3)$	P_4	P_5
1s	105.7%	149.0%	1,209.0%	144.5%	137.8%	1,138.7%	0.0%	32.7%
10s	202.2%	91.4%	1,062.2%	316.0%	101.1%	1,061.4%	0.0%	47.9%
1m	393.5%	77.4%	877.3%	586.6%	98.9%	885.8%	0.0%	33.9%
10m	600.5%	69.6%	593.2%	940.3%	104.1%	596.7%	0.0%	56.4%
1h				1,019.9%	106.6%	371.1%	0.0%	48.3%
10h				1,046.7%	119.6%	160.6%	0.0%	10.8%

the performance of the class-based portfolio P_5 is superior to the performance of the gold standard restart strategy portfolios $P_1(i)$, i = 1, 2, or 3, even though the gold standard restart strategies were computed by determining for each instance and algorithm the optimal fixed cutoff for the runtime distribution for the algorithm applied to the instance. As well, the performance of P_5 is superior to the portfolios which contain a single algorithm. Even though there is one dominant algorithm among the deterministic algorithms (algorithm $\mathcal{A}_2^{det}$ as used in portfolio $P_3(2)$) a portfolio of multiple algorithms can take advantage of the variability in performance of the algorithms across instances to improve the performance measure. Finally, it is interesting to note that the performance of the class-based portfolio P_5 approaches the performance of the gold standard algorithm selection portfolio which always selects the best algorithm for each instance. This is especially true for the larger deadlines. Of course, no algorithm selection method will be perfect. For example, Guerri and Milano [13] report a 90% selection accuracy when selecting between just two algorithms. We experimented with having the algorithm selection method make mistakes: with some small probability p the method does not select the best algorithm for an instance. Depending on the deadline, if the probability of a mistake is greater than from 1% to 6%, portfolio P_5 would outperform the algorithm selection portfolio.

4.3. Experiment 2

In our second set of experiments, we determined whether the portfolio learned using our methodology is effective at reducing the expected time to solve the instances in the presence of various deadlines. We used the same deadlines as in Experiment 1. Table 4 summarizes the results. Table 5 presents the same information except now stated in terms of percentage change.

On this testbed, portfolio P_5—the portfolio learned by our approach—-again performs well. The performance of the class-based portfolio P_5 is superior to the performance of the gold standard restart strategy portfolios, and superior to the portfolios which contain a single algorithm. Also, under this performance measure too, the performance of the class-based portfolio P_5 approaches the performance of the gold standard algorithm selection portfolio. We again experimented with having the algorithm selection method make mistakes: with some small probability p the method does not select the best algorithm for an instance. Depending on the deadline, if the probability of a mistake is greater

than from 3% to 8%, portfolio P_5 would outperform the algorithm selection portfolio.

5. Conclusions

We presented an approach for learning good class-based portfolios of backtracking algorithms in the commonly occurring scenario where instances from a problem class are to be solved over time and a deadline is placed on the computational resources that the backtracking algorithm can consume in solving any instance. Our approach has a relatively low computational cost and is applicable in scenarios where the problem class contains heterogeneous instances. We demonstrated the effectiveness of our approach through an extensive empirical evaluation on a real-world scheduling testbed. On our testbed, the portfolio that is learned by our methodology outperforms the best possible (but unobtainable in practice) restart strategy portfolio and approaches the performance of the best possible (but again unobtainable in practice) algorithm selection portfolio.

Acknowledgments.

This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET).

References

[1] C. Bessiere. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[2] T. Carchrae and J. C. Beck. Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21:372–387, 2005.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. ACM*, 5:394–397, 1962.

[4] L. Finkelstein, S. Markovitch, and E. Rivlin. Optimal schedules for parallelizing anytime algorithms: The case of shared resources. *J. of AI Research*, 19:73–138, 2003.

[5] M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Math. and AI*, 47:295–328, 2006.

[6] M. Gagliolo and J. Schmidhuber. Learning restarts strategies. In *Proc. of the 20th Int'l Joint Conf. on AI*, pages 792–797, Hyderabad, India, 2007.

[7] S. Golomb and L. Baumert. Backtrack programming. *J. ACM*, 12:516–524, 1965.

[8] C. Gomes and B. Selman. Algorithm portfolio design: Theory vs. practice. In *Proc. of the 13th Annual Conf. on Uncertainty in AI (UAI-97)*, pages 190–197, Providence, RI, 1997.

[9] C. Gomes and B. Selman. Algorithm portfolios. *Artif. Intell.*, 126:43–62, 2001.

[10] C. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Proc. of the Third Int'l Conf. on Principles and Practice of Constraint Programming*, pages 121–135, Linz, Austria, 1997.

[11] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. of Automated Reasoning*, 24:67–100, 2000.

[12] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of the Fifteenth National Conf. on AI*, pages 431–437, Madison, Wisconsin, 1998.

[13] A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proc. of the 16th European Conf. on AI*, pages 475–479, Valencia, Spain, 2004.

[14] W. D. Harvey. *Nonsystematic backtracking search*. PhD thesis, Stanford University, 1995.

[15] T. Hogg and C. P. Williams. Expected gains from parallelizing constraint solving for hard problems. In *Proc. of the Twelfth National Conf. on AI*, pages 331–336, Seattle, 1994.

[16] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and M. Chickering. A Bayesian approach to tackling hard computational problems. In *UAI-2001*, pages 235–244, 2001.

[17] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.

[18] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *Proc. of the Eighteenth National Conf. on AI*, pages 674–681, Edmonton, 2002.

[19] M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability, 2001.

[20] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proc. of the Eighth Int'l Conf. on Principles and Practice of Constraint Programming*, pages 556–572, Ithaca, New York, 2002.

[21] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Proc. of the Ninth Int'l Conf. on Principles and Practice of Constraint Programming*, pages 899–903, Kinsale, Ireland, 2003.

[22] L. Lobjois and M. Lemaitre. Branch and bound algorithm selection by performance prediction. In *Proc. of the Fifteenth National Conf. on AI*, pages 353–358, Madison, Wisconsin, 1998.

[23] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[24] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. In *Proc. of the 18th IEEE Int'l Conf. on Tools with AI*, pages 279–287, Washington, DC, 2006.

[25] Y. Ruan, E. Horvitz, and H. Kautz. Restart policies with dependence among runs: A dynamic programming approach. In *Proc. of the Eighth Int'l Conf. on Principles and Practice of Constraint Programming*, pages 573–586, Ithaca, New York, 2002.

[26] P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.