International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

PORTFOLIOS WITH DEADLINES FOR BACKTRACKING SEARCH

HUAYUE WU

School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
hwu@cs.uwaterloo.ca

PETER VAN BEEK

School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
vanbeek@cs.uwaterloo.ca

Backtracking search is often the method of choice for solving constraint satisfaction and propositional satisfiability problems. 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 can be consumed 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 using two testbeds: real-world instruction scheduling problems and the widely used quasigroup completion problems.

Keywords: Constraint satisfaction; satisfiability; backtracking search; portfolios, restarts.

1. Introduction

Constraint programming is a methodology for solving difficult combinatorial problems such as scheduling, sequencing, and planning. The basic notion of a constraint programming approach is a constraint satisfaction problem (CSP), which is defined by a set of variables, a domain of values for each variable, and a set of constraints over the variables. The propositional satisfiability problem is a CSP where the domains of the variables are the Boolean values and the constraints are Boolean formulas.

In this paper, we consider a setting where the CSP 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 can be consumed 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 to keep the total compile time to an acceptable level.

To solve a CSP is to find an assignment of values to the variables so that all constraints are satisfied. In practice, constraint satisfaction and propositional satisfiability problems are often solved using backtracking search. Since the first formal statements of backtracking algorithms over 40 years ago [5,10], many techniques for improving the efficiency of a backtracking search algorithm have been suggested and evaluated (see, e.g., [36]). 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., [15,19,22,30]).

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

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

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,

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

i.e., the same algorithm is continually restarted until either a solution is found or the deadline is reached [15,19,30]. Of course, for a restart strategy to make sense, $\mathcal{A}$ must be a nondeterministic algorithm. The usual method for randomizing a backtracking algorithm is to randomize the variable or value ordering heuristic (e.g., [16,19]).

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 $\{\mathcal{A}_1, \mathcal{A}_2, \ldots\}$ 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 the widely used quasigroup completion problem testbed [13] and on a real-world instruction scheduling testbed [31]. The portfolio is learned from a small training set and then evaluated on a test set to estimate its performance on unknown instances in the future. We show that on both of our testbeds, 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. Background

In this section, we briefly review the needed background from constraint programming.

In a constraint programming approach, a problem is modeled in terms of variables, values, and constraints. Such a model is often called a constraint satisfaction problem (CSP) or a CSP model.

Definition 2.1. A *constraint satisfaction problem* (CSP) consists of a set of n *variables*, $\{x_1, \ldots, x_n\}$; a finite domain $dom(x_i)$ of possible *values* for each variable x_i, $1 \leq i \leq n$; and a collection of r *constraints*, $\{C_1, \ldots, C_r\}$. Each constraint C_i, $1 \leq i \leq r$, is a constraint over some set of variables, denoted by $vars(C_i)$, that specifies the allowed combinations of values for the variables in $vars(C_i)$. Given a constraint C, the notation $t \in C$ denotes a tuple t—an assignment of a value to each of the variables in $vars(C)$—that satisfies the constraint C. The notation $t[x]$ denotes the value assigned to variable x by the tuple t. A *solution* to a CSP is an assignment of a value to each variable that satisfies all of the constraints.

The propositional satisfiability problem is a CSP where the domains of the variables are the Boolean values and the constraints are Boolean formulas. Here we assume that the formula is in conjunctive normal form (CNF). A formula is in CNF if it is a conjunction of clauses, where each clause is a disjunction of literals and a literal is a Boolean variable or the negation of a Boolean variable.

CSPs are often solved using a backtracking algorithm. At every stage of the backtracking search, there is some current partial solution that the algorithm attempts to extend to a full solution by assigning a value to an uninstantiated variable. One of the keys behind the success of constraint programming is the idea of constraint propagation. During the backtracking search when a variable is assigned a value, the constraints are used to reduce the domains of the uninstantiated variables by ensuring that the values in their domains are consistent with the constraints.

One form of constraint propagation that is applicable when the domains of the variables are integers is called bounds consistency propagation. The minimum and maximum values in the domain $dom(x)$ of a variable x are denoted by $\min(dom(x))$ and $\max(dom(x))$, and the interval notation $[a, b]$ is used as a shorthand for the set of values $\{a, a+1, \ldots, b\}$.

Definition 2.2. Given a constraint C, a value $a \in dom(x)$ for a variable $x \in vars(C)$ is said to have a *support* in C iff there exists a $t \in C$ such that $a = t[x]$ and $t[y] \in [\min(dom(y)), \max(dom(y)]$, for every $y \in vars(C)$. A constraint C is said to be *bounds consistent* iff for each $x \in vars(C)$, each of the values $\min(dom(x))$ and $\max(dom(x))$ has a support in C.

A CSP can be made bounds consistent by repeatedly removing unsupported values from the domains of its variables.

Example 2.1. Consider the CSP model of a small scheduling problem which has variables A, …, E, each with domain $\{1, \ldots, 6\}$, and the constraints,

$$\begin{array}{lll} C_1\text{:}\ \ \mathrm{D} \geq \mathrm{A} + 3, & C_3\text{:}\ \ \mathrm{E} \geq \mathrm{C} + 3, & \\ C_2\text{:}\ \ \mathrm{D} \geq \mathrm{B} + 3, & C_4\text{:}\ \ \mathrm{E} \geq \mathrm{D} + 1, & C_5\text{:}\ \ \text{all-different(A, B, C, D, E)}, \end{array}$$

where constraint C_5 enforces that its arguments are pair-wise different. The constraints are not bounds consistent. For example, the minimum value 1 in the domain of D does not have a support in C_1 as there is no corresponding value for A that satisfies the constraint. Enforcing bounds consistency using constraints C_1 through C_4 reduces the domains as follows: $dom(\mathrm{A}) = \{1, 2\}$, $dom(\mathrm{B}) = \{1, 2\}$, $dom(\mathrm{C}) = \{1, 2, 3\}$, $dom(\mathrm{D}) = \{4, 5\}$, and $dom(\mathrm{E}) = \{5, 6\}$. Subsequently enforcing bounds consistency using constraint C_5 further reduces the domain of C to be $dom(\mathrm{C}) = \{3\}$. Now constraint C_3 is no longer bounds consistent. Re-establishing bounds consistency causes $dom(\mathrm{E}) = \{6\}$.

A second form of constraint propagation that is applicable to propositional satisfiability problems in conjunctive normal form is called unit propagation.

Example 2.2. Consider the small propositional satisfiability instance which has Boolean variables x_1, x_2, and x_3, and the three clauses (constraints),

$$C_1\text{:}\ \ x_1, \qquad C_2\text{:}\ \ \neg x_1 \vee \neg x_2, \qquad C_3\text{:}\ \ x_2 \vee \neg x_3$$

Clause C_1 is called a unit clause as it consists of a single literal. The unit clause C_1 forces variable x_1 to be true. As a result, clause C_2 can be simplified to be just $\neg x_2$. The simplified clause C_2 can now be propagated to clause C_3 which is simplified to be just $\neg x_3$.

Given a base level of constraint propagation, such as bounds consistency or unit propagation, a general technique called singleton consistency exists for enforcing higher levels of constraint propagation [6,32]. The idea is to, in turn, assign a value to a variable and test whether the assignment is consistent by performing constraint propagation on the subproblem. If the assignment is not consistent, the value cannot be part of any solution and can be removed. To define this more formally, we first introduce some notation. Given a CSP P and an assignment $x = a$, where $a \in dom(x)$, the CSP induced by the assignment, denoted $P|_{x=a}$, is obtained from P by reducing $dom(x)$ to be the singleton set $\{a\}$. A CSP is said to be bounds inconsistent if a domain of a variable is empty after enforcing bounds consistency; i.e., enforcing bounds consistency detects that the CSP has no solution.

Definition 2.3. A CSP P is said to be *singleton bounds consistent* iff for each x and each $a \in dom(x)$, $P|_{x=\min(dom(x))}$ is not bounds inconsistent and $P|_{x=\max(dom(x))}$ is not bounds inconsistent.

A CSP can be made singleton bounds consistent by assigning each value in turn and testing whether the CSP induced by the assignment is bounds inconsistent. If the CSP is bounds inconsistent, the value can be removed from the domain of the variable.

Example 2.3. Consider the CSP P for a small graph coloring problem which has three variables x_1, x_2, and x_3, each with domain {red, green}, and the constraints $x_1 \neq x_2$, $x_1 \neq x_3$, and $x_2 \neq x_3$. The CSP is bounds consistent; i.e., enforcing bounds consistency does not change the domains. However, the CSP is not singleton bounds consistent. Consider the assignment $x_1 = \text{red}$. Enforcing bounds consistency on $P|_{x_1=\text{red}}$ results in x_2 and x_3 having empty domains indicating the induced CSP is bounds inconsistent. Thus, the color red can be removed from the domain of x_1. In a similar manner, the color green can be removed and we have discovered that the original CSP does not have a solution.

The above defined singleton consistency using bounds consistency as the base level of constraint propagation. Other singleton consistencies can similarly be defined by replacing bounds consistency by some other form of constraint propagation [3]. In particular, in our experiments we make use of singleton bounds consistency to a depth of two. A CSP is singleton bounds consistent to a depth of two if each minimum and maximum value in the domain of a variable is not singleton bounds inconsistent; i.e., we test each value in turn and check that no domain of a variable is empty after enforcing singleton bounds consistency.

3. 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.

3.1. *General portfolios*

Huberman, Lukose, and Hogg [22] 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 tradeoff 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 [12,14] 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 [14].

Lagoudakis and Littman [24] 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 [7] 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 preassigned 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 [8] 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.

3.2. *Algorithm selection portfolios*

Lobjois and Lemaître [29] examine instance-based algorithm selection in the setting of branch-and-bound algorithms for optimization problems. Carchrae and Beck [4] 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. [25,26,39] 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 [21]). 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 [18], 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 can significantly outperform the algorithm selection portfolio.

3.3. *Restart strategy portfolios*

Luby, Sinclair, and Zuckerman [30] 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., [9,23,33,35,38]). For example, Streeter, Golovin, and Smith [35] present an online technique for constructing a single restart strategy for an ensemble of instances; i.e., a class-based restart strategy. However, we show that our methodology constructs portfolios that significantly outperform the single restart strategy that would be constructed by Streeter et al.'s method. We do so indirectly by showing in our experiments that our portfolios outperform optimal instance-based restart strategies. Then, since optimal instance-based restart strategies significantly outperform class-based restart strategies, our claim follows.

As well, there have been extensive empirical evaluations of restart strategy portfolios for backtracking algorithms with various randomized heuristics (e.g., [16,17,20]). Interestingly, Gomes and Selman [14] 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.

4. 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 heuristic 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. Of course, just as weakening the variable ordering heuristic can lead to dominated algorithms, increasing the level of constraint propagation can also do so. However, we found that on the benchmark instances that we experimented with no algorithm dominated; each algorithm was best over a significantly-sized set of instances.

The technique of creating variability by increasing levels of constraint propagation 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 (see Section 2). 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.

Example 4.1. To illustrate the methodology, suppose that in Step 1 we have decided on constructing a portfolio of three algorithms $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$, where the algorithms are ordered by increasing level of constraint propagation. Table 1 shows performance data for

Table 1. Example data for a training set and test set for three algorithms $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$, where the deadline is 600s.

training set				test set			
instance	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	instance	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$
train1	1	5	31	test1	1	7	40
train2	91	1	1	test2	1	1	50
train3	1	188	600	test3	1	1	7
train4	600	10	73	test4	1	3	14
train5	1	6	25	test5	600	600	600
train6	1	1	15	test6	600	1	5
train7	600	600	2	test7	1	10	67
train8	1	14	97	test8	1	443	600

hypothetical training and test sets when the algorithms are run on the instances with the deadline or limit on CPU time of 600 seconds (Step 2 of the methodology). Each entry in the table is the result of applying an algorithm on an instance: the number of seconds taken to solve the instance or the deadline, if no solution was found within the allotted time.

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. This last restriction may not be needed in every domain; the goal of the restriction is simply to reduce the number of possible portfolios that need to be examined. We also remark that another ordering, such as by decreasing level of propagation, could be a better restriction to impose and that deciding this would require some experimentation when applying the method to a new problem class.

Because of the presence of a deadline and because we focus on simpler portfolios, systematically stepping through the possible portfolios 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 [7], 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.

Example 4.2. Continuing with Example 4.1, suppose that in Step 3 we decide to examine portfolios in units of 5 seconds. We next simulate the possible portfolios of the form ($\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$), where the total time allocated to the three algorithms is equal to the deadline of 600 seconds; i.e., (0, 0, 600), (0, 5, 595), (0, 10, 590), . . . , (600, 0, 0).

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.

Example 4.3. Continuing with Example 4.2, suppose that in Step 4 we wish to learn a good portfolio when the performance measure—the value that we wish to minimize—is the number of instances which are not solved. There are many portfolios which minimize the number of instances in the training set that are not solved. For example, the portfolio (10, 585, 5) solves all of the instances in the training set; i.e., run algorithm $\mathcal{A}_1$ for 10 seconds or until a solution is found, if no solution is found run algorithm $\mathcal{A}_2$ for 585 seconds, and so on. However, this portfolio is not robust as it is close in distance to a portfolio, (10, 590, 0), with poorer results. The distance is measured by the number of times that one can move ± 5 (the level of abstraction we have chosen for our portfolios) from one of the entries in the portfolio to another, while ensuring that no entry becomes less than zero or greater than the deadline. The best portfolio, the one that is furthest away from any portfolio with poorer results, is (5, 190, 405). When this portfolio is applied to the test set (Step 5), the number of instances in the test set not solved is one.

Alternatively, suppose that in Step 4 the performance measure is the time to solve all of the instances. The (unique) portfolio that minimizes the time to solve all of the instances in the training set is (5, 10, 585). When this portfolio is applied to the test set, the total time for all instances in the test set is 612 seconds. It turns out that this is optimal as no other portfolio does better on the test set.

5. Experimental evaluation

In this section, we present the results of an empirical evaluation of our approach. In our empirical evaluation we used two testbeds: quasigroup completion problems and real-world instruction scheduling problems. For each testbed, 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. For each testbed, 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.

Table 2. 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 4).

Let $\mathcal{A}_i^s$ be a backtracking algorithm, where $i = 1, 2, \ldots$ indicates the level of constraint propagation used by the algorithm and $s \in \{det, rand\}$ indicates whether the algorithm is the deterministic or the randomized version of the algorithm. As explained in more detail below when we describe the experiments in each of the testbeds, we randomized a backtracking algorithm by randomizing the variable ordering heuristic. The notation for specifying the portfolios that we examined in the experiments is summarized in Table 2.

In all of our experiments, if the algorithm was randomized, we collected 1000 samples of its runtime distribution on each instance in the testbed 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. When a portfolio included randomized algorithms, the statistics that we report are the average over 10,000 trials.

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.

5.1. *Quasigroup Completion*

We first report on our experiments using quasigroup completion problems [13]. A Latin square is an $n \times n$ table filled with n different symbols such that each symbol occurs exactly once in each row and once in each column. In the quasigroup completion problem, we are given a partially filled $n \times n$ table and asked whether the remaining entries can be filled in such that the table is a Latin square. The quasigroup completion problem is known to be NP-complete.

For our experiments, we used 1000 randomly generated balanced quasigroup completion instances with $n = 31$ and each instance had 391 unfilled entries. The instances were generated using LSENCODE [11] and were encoded as a propositional satisfiability problem in conjunctive normal form. The parameters were chosen to give difficult instances and there are both satisfiable and unsatisfiable instances. The sizes of the instances range from 1,691 to 2,104 variables and from 11,757 to 16,354 clauses. We used 10-fold cross-validation to divide the data into training set and test set (see, e.g., [37]). In 10-fold cross-validation, the data set (the 1000 instances) is partitioned into 10 equal-sized subsamples. Then, for each $k = 1, \ldots, 10$, we reserve the kth subsample to be used as a test set to evaluate the learned portfolio and the other $k - 1$ subsamples are used as the training set from which to learn the portfolio. The average of the 10 test results are then reported.

For our experiments, we used two separate deterministic backtracking algorithms $\mathcal{A}_i^{det}$, $i = 1$ or 2, capable of performing distinct levels of constraint propagation:

i = 1	SATZ solver [27,28], performs unit propagation, and
i = 2	2CLS+EQ solver [1,2], performs extended binary clause reasoning.

Unit propagation is standard in satisfiability solvers (see Section 2). In extended binary clause reasoning, at each node in the search tree the backtracking algorithm also performs reasoning (resolution) on binary clauses (clauses with two literals) rather than just unit clauses.

To study restart strategy portfolios, the backtracking algorithms were randomized by having the variable ordering heuristic score the variable choices and then randomly pick a variable from among all the variables that are within a factor of 0.4 of the best scoring variable. This gave a total of four distinct backtracking algorithms $\mathcal{A}_i^s$, where $i = 1$ or 2 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 four algorithms on each quasigroup instance in the training and test sets and recorded the performance data.

Table 3. Expected number of quasigroup 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	$P_1(1)$	$P_1(2)$	$P_2(1)$	$P_2(2)$	$P_3(1)$	$P_3(2)$	P_4	P_5
1s	74.7	83.8	78.6	84.4	76.3	84.8	68.1	70.5
10s	39.5	74.9	52.9	77.3	49.0	77.3	39.8	41.5
1m	27.1	53.7	36.1	60.7	32.4	56.9	22.6	24.7
10m	25.7	21.1	28.3	31.7	27.4	29.9	16.5	17.8
1h					26.9	17.6	14.5	14.9
10h					26.9	12.6	12.3	12.6

Table 4. Percentage increase in the expected number of quasigroup 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	$P_1(1)$	$P_1(2)$	$P_2(1)$	$P_2(2)$	$P_3(1)$	$P_3(2)$	P_4	P_5
1s	9.7%	23.1%	15.4%	23.9%	12.0%	24.5%	0%	3.5%
10s	−0.8%	88.2%	32.9%	94.2%	23.1%	94.2%	0%	4.3%
1m	19.9%	137.6%	59.7%	168.6%	43.4%	151.8%	0%	9.3%
10m	55.8%	27.9%	71.5%	92.1%	66.1%	81.2%	0%	7.9%
1h					85.5%	21.4%	0%	2.8%
10h					118.7%	2.4%	0%	2.4%

Quasigroup completion: Experiment 1

In our first set of experiments for quasigroup instances, 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 3 summarizes the results. Table 4 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 the performance of the class-based portfolio P_5 is superior or close in performance to the gold standard restart strategy portfolios $P_1(i)$, i = 1 or 2, 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 just the individual algorithms, $P_3(i)$, i = 1 or 2.

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 P_4 which perfectly selects the best algorithm for each instance. Of course, no algorithm selection method will be perfect in practice. Xu et al. [39] report a 62% selection accuracy when selecting between three backtracking solvers on quasigroup instances. 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. The results are quantified in Figure 1(a). Depending on the deadline, if the probability of a mistake is greater than from 3% to 10%, portfolio P_5 would outperform the algorithm selection portfolio. As one example, consider the case where the deadline is 10m. If the probability of a mistake is zero, portfolio P_5 gives an increase of 7.9% in the expected number of instances not solved. However, once the probability of a mistake is greater than 0.06, portfolio P_5 outperforms the algorithm selection portfolio until when the probability of a mistake is 0.20 (a reasonable expectation in practice) portfolio P_5 gives a 16.4% improvement. As well, we note that the algorithm selection method, as exemplified by Xu et al.'s [39] approach, is difficult to apply in practice as it requires the user to develop an empirical hardness model—a task that requires expertise—whereas our approach is light weight.

Quasigroup completion: Experiment 2

In our second set of experiments for quasigroup instances, 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 5 summarizes the results. Table 6 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 approaches the performance of the gold standard restart strategy portfolios, and is 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. The results are quantified in Figure 1(b). Depending on the deadline, if the probability of a mistake is greater than from 3% to 34%, portfolio P_5 would outperform the gold standard algorithm selection portfolio P_4. As one example, if the deadline is 10m, P_5 is 18.5% slower when the probability of a mistake p is zero, roughly equal in performance when $p = 0.11$ and 11.2% faster when $p = 0.20$ (a reasonable expectation in practice).

5.2. *Instruction Scheduling*

We next report on our experiments using instruction scheduling problems for multiple-issue pipelined processors. 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,

Table 5. Total time (sec.) for *all* of the quasigroup 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	$P_1(1)$	$P_1(2)$	$P_3(1)$	$P_3(2)$	P_4	P_5
1s	83.2	85.8	82.8	86.3	75.7	81.6
10s	538.3	794.2	610.6	810.8	524.3	593.8
1m	2,050.2	3,926.1	2,463.8	4,069.5	1,890.8	2,275.1
10m	16,040.6	20,245.7	17,836.6	24,412.6	11,509.3	13,636.9
1h			99,074.1	88,104.5	57,624.8	64,322.8
10h			970,634.1	529,334.7	477,404.2	491,088.2

Table 6. Percentage increase in the total time (sec.) for *all* of the quasigroup instances in the test set relative to the gold standard portfolio P_4, for various portfolios of algorithms.

	Randomized		Deterministic			
d	$P_1(1)$	$P_1(2)$	$P_3(1)$	$P_3(2)$	P_4	P_5
1s	9.9%	13.3%	9.4%	14.0%	0%	7.8%
10s	2.7%	51.5%	16.5%	54.6%	0%	13.3%
1m	8.4%	107.6%	30.3%	115.2%	0%	20.3%
10m	39.4%	75.9%	55.0%	112.1%	0%	18.5%
1h			71.9%	52.9%	0%	11.6%
10h			103.3%	10.9%	0%	2.9%

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. The instruction scheduling problem is known to be NP-complete.

We formulated a constraint programming model for the instruction scheduling problem and solved instances using backtracking search. In constraint programming, a problem 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 (see Section 2). 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 [31] 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 ordering 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 $\mathcal{A}_i^{det}$ capable of performing three levels of constraint propagation:

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

See Section 2 for an explanation and examples of the three levels of constraint propagation. We chose bounds consistency—instead of the more usual arc consistency—as in our problem it is equivalent but more efficient (see [34]).

To study restart strategy portfolios, the backtracking algorithms were randomized by having the variable ordering heuristic randomly pick a variable from the top five variables (or fewer, if there were fewer variables left). 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 instruction scheduling instance in the training and test sets and recorded the performance data.

Instruction scheduling: Experiment 1

In our first set of experiments for instruction scheduling instances, 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 the same deadlines as in our experiments with quasigroup instances. Table 7 summarizes the results. Table 8 presents the same information except now stated in terms of percentage change. As in our experiments with quasigroup instances, because of the high computational cost, we did not collect runtime distributions for the larger timeouts 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.

Table 7. 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 8. 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	148%	89%	1187%	249%	94%	1187%	214%	100%	1188%	0%	16%
10s	323%	85%	1040%	559%	105%	1040%	509%	100%	1050%	0%	44%
1m	592%	94%	782%	1022%	136%	782%	944%	122%	794%	0%	67%
10m	641%	61%	427%	1108%	112%	431%	994%	100%	444%	0%	6%
1h							1040%	107%	253%	0%	0%
10h							1021%	121%	93%	0%	0%

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, the class-based portfolio P_5 approaches the performance of the gold standard algorithm selection portfolio P_4 which perfectly selects the best algorithm for each instance. This is especially true for the larger deadlines. However, it is clear that no algorithm selection method will be perfect. Guerri and Milano [18] report a 90% selection accuracy when selecting between just two algorithms when solving a constraint satisfaction problem for combinatorial auctions. 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. The results are quantified in Figure 1(c). Depending on the deadline, if the probability of a mistake is greater than from 1% to 8%, portfolio P_5 would outperform the gold standard algorithm selection portfolio P_4. As one example, consider the case where the deadline is 10m. If the probability of a mistake is zero, portfolio P_5 gives an increase of 6% in the expected number of instances not solved. However, once the probability of a mistake is greater than zero, portfolio P_5 outperforms the algorithm selection portfolio until when the probability of a mistake is 0.20 (a reasonable expectation in practice) portfolio P_5 gives a 58% improvement.

Table 9. 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 10. 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%

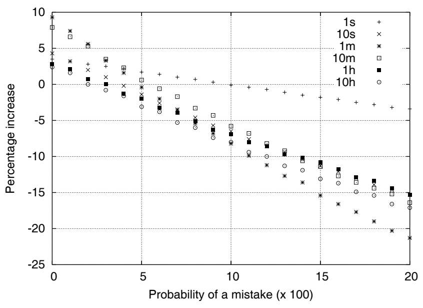
Instruction scheduling: Experiment 2

In our second set of experiments for instruction scheduling instances, 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 9 summarizes the results. Table 10 presents the same information except now stated in terms of percentage change.

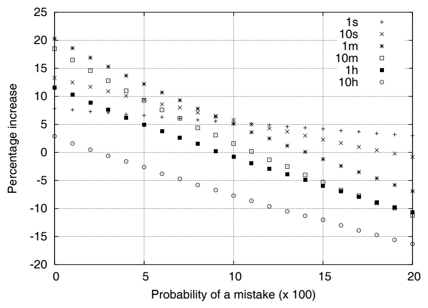
On this testbed, portfolio P_5—the portfolio learned by our approach—-again performs well. The performance of the class-based portfolio P_5 is superior to the performance of the gold standard restart strategy portfolios, and superior to the portfolios which contain a single algorithm. Also, under this performance measure too, the performance of the class-based portfolio P_5 approaches the performance of the gold standard algorithm selection portfolio P_4. 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. The results are quantified in Figure 1(d). Depending on the deadline, if the probability of a mistake is greater than from 2% to 7%, portfolio P_5 would outperform the gold standard algorithm selection portfolio. As one example, if the deadline is 10m, P_5 is 56.4% slower when the probability of a mistake p is zero, roughly equal in performance when $p = 0.07$ and 41.0% faster when $p = 0.20$ (a reasonable expectation in practice).

6. Conclusions

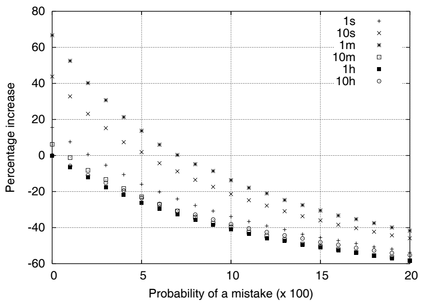
We presented an approach for learning good class-based portfolios of backtracking algorithms in the commonly occurring scenario where instances from a problem class are to be solved over time and a deadline is placed on the computational resources that the back-



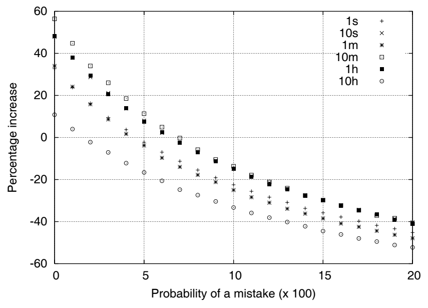
(a)



(b)



(c)



(d)

Figure 1. Comparison of portfolio P_5—the portfolio learned by our approach—with portfolio P_4—the algorithm selection portfolio—when the algorithm selection method makes a mistake with probability p: (a) percentage increase in the expected number of quasigroup instances in the test set *not* solved; (b) percentage increase in the total time for all of the quasigroup instances in the test set; (c) percentage increase in the expected number of scheduling instances in the test set *not* solved; (d) percentage increase in the total time for all of the scheduling instances in the test set.

tracking 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 quasigroup completion problems and a real-world scheduling testbed. On our testbeds, 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).

Bibliography

1. F. Bacchus. 2clseq: A DPLL solver employing extensive binary clause reasoning, 2002. Available from http://www.cs.toronto.edu/~fbacchus/2clseq.html.
2. F. Bacchus. Enhancing Davis Putman with extended binary clause reasoning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 613–619, Edmonton, 2002.
3. C. Bessiere. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.
4. T. Carchrae and J. C. Beck. Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21:372–387, 2005.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. ACM*, 5:394–397, 1962.
6. R. Debruyne and C. Bessiere. Domain filtering consistencies. *J. Artificial Intelligence Research*, 14:205–230, 2001.
7. L. Finkelstein, S. Markovitch, and E. Rivlin. Optimal schedules for parallelizing anytime algorithms: The case of shared resources. *J. of Artificial Intelligence Research*, 19:73–138, 2003.
8. M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47:295–328, 2006.
9. M. Gagliolo and J. Schmidhuber. Learning restarts strategies. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 792–797, Hyderabad, India, 2007.
10. S. Golomb and L. Baumert. Backtrack programming. *J. ACM*, 12:516–524, 1965.
11. C. Gomes. lsencode quasigroup instance generator, version 1.1. Available from http//www.cs.cornell.edu/gomes/new-demos.htm.
12. C. Gomes and B. Selman. Algorithm portfolio design: Theory vs. practice. In *Proceedings of the 13th Annual Conference on Uncertainty in Artificial Intelligence*, pages 190–197, Providence, Rhode Island, 1997.
13. C. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 221–226, Providence, Rhode Island, 1997.
14. C. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.
15. C. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 121–135, Linz, Austria, 1997.
16. 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.
17. C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 431–437, Madison, Wisconsin, 1998.
18. A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 475–479, Valencia, Spain, 2004.
19. W. D. Harvey. *Nonsystematic backtracking search*. PhD thesis, Stanford University, 1995.
20. T. Hogg and C. P. Williams. Expected gains from parallelizing constraint solving for hard problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 331–336, Seattle, 1994.
21. E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and M. Chickering. A Bayesian approach to tackling hard computational problems. In *Proceedings of the 17th Annual Conference on Uncertainty in Artificial Intelligence*, pages 235–244, Seattle, 2001.
22. B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.
23. H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *Proceed-*

ings of the Eighteenth National Conference on Artificial Intelligence, pages 674–681, Edmonton, 2002.

24. M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.
25. 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 *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 556–572, Ithaca, New York, 2002.
26. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 899–903, Kinsale, Ireland, 2003.
27. C.-M. Li and A. Anbulagan. Satz satisfiability solver, version 215.2. Available from http//www.laria.u-picardie.fr/~cli.
28. C.-M. Li and A. Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 341–355, Linz, Austria, 1997.
29. L. Lobjois and M. Lemaitre. Branch and bound algorithm selection by performance prediction. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 353–358, Madison, Wisconsin, 1998.
30. M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.
31. A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, pages 279–287, Washington, DC, 2006.
32. P. Martin and D. B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings of the Fifth International IPCO Conference*, pages 389–403, Vancouver, 1996.
33. Y. Ruan, E. Horvitz, and H. Kautz. Restart policies with dependence among runs: A dynamic programming approach. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 573–586, Ithaca, New York, 2002.
34. C. Schulte and P. J. Stuckey. When do bounds and domain propagation lead to the same search space. In *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, pages 115–126, Firenze, Italy, 2001.
35. M. Streeter, D. Golovin, and S. F. Smith. Restart schedules for ensembles of problem instances. In *Proceedings of the 22nd Conference on Artificial Intelligence*, pages 1204–1210, Vancouver, 2007.
36. P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.
37. I. H. Witten and E. Frank. *Data Mining*. Morgan Kaufmann, 2000.
38. H. Wu and P. van Beek. On universal restart strategies for backtracking search. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 681–695, Providence, Rhode Island, 2007.
39. L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 712–727, Providence, Rhode Island, 2007.