Computational Intelligence, Volume 21, Number 4, 2005

APPLYING MACHINE LEARNING TO LOW-KNOWLEDGE CONTROL OF OPTIMIZATION ALGORITHMS

TOM CARCHRAE

Cork Constraint Computation Center, Department of Computer Science, University College Cork, Ireland

J. CHRISTOPHER BECK

Toronto Intelligent Decision Engineering Lab, Department of Mechanical & Industrial Engineering, University of Toronto, Canada

This paper addresses the question of allocating computational resources among a set of algorithms to achieve the best performance on scheduling problems. Our primary motivation in addressing this problem is to reduce the expertise needed to apply optimization technology. Therefore, we investigate algorithm control techniques that make decisions based only on observations of the improvement in solution quality achieved by each algorithm. We call our approach "low knowledge" since it does not rely on complex prediction models, either of the problem domain or of algorithm behavior. We show that a low-knowledge approach results in a system that achieves significantly better performance than all of the pure algorithms without requiring additional human expertise. Furthermore the low-knowledge approach achieves performance equivalent to a perfect high-knowledge classification approach.

Key words: algorithm selection, algorithm control, machine learning, classifiers, scheduling.

1. INTRODUCTION

Despite both commercial and academic success of optimization technology, using it still requires significant expertise. For nontrivial applications the quality of a system still has much to do with the experience and capability of the person that implemented it (Le Pape et al. 2002). In this paper, we investigate algorithm control techniques aimed at achieving strong scheduling performance using off-the-shelf algorithms without requiring significant human expertise. This is done through the application of machine learning techniques to low-knowledge algorithm control. Rather than building knowledge-intensive models relating algorithm performance to problem features, we base the control decisions on the changes in solution quality over time.

Given a time limit T to find the best solution possible to a problem instance, we investigate two control paradigms, shown in Figure 1. The first, *predictive*, paradigm runs a set of algorithms during a prediction phase and chooses one to run for the remainder of T. Based on these short runs during the prediction phase, we decide which algorithm to continue running with the remaining time. We start by comparing some simple rules, such as choosing the algorithm with the minimum cost found during the prediction phase. We then look at a Bayesian classifier, which attempts to correlate the algorithm performance trends from the prediction phase with the best performing algorithm for that problem instance. Once it is trained, the Bayesian classifier is used to predict the algorithm that is expected to be the best performer on new problem instances based only on performance in the prediction phase.

In the second, *switching*, paradigm control decisions allocate computational resources to each algorithm over a series of iterations such that the total runtime is T. In an iteration, we run each algorithm, one after another, passing the best-known solution from one algorithm to the next. Thus, rather than make a single decision about which algorithm to select for the remaining runtime, we revisit our choice over and over. We apply a reinforcement learning approach to allocate more runtime to algorithms that perform well.

We use a low-knowledge design in all of the algorithms explored here. We explain this idea in more detail below, however it is important to stress why we make the distinction

© 2005 Blackwell Publishing, 350 Main Street, Malden, MA 02148, USA, and 9600 Garsington Road, Oxford OX4 2DQ, UK.

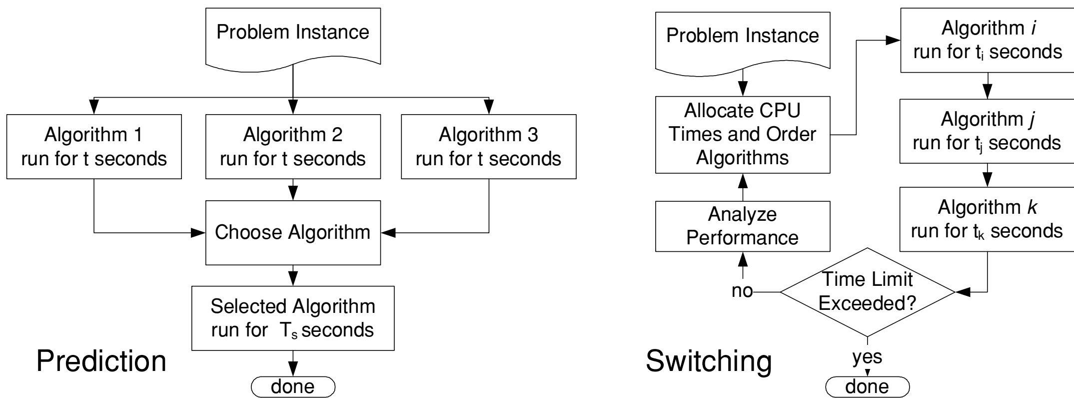

FIGURE 1. The two control methods presented in this paper. *prediction* makes a single decision based on performance observations over short runtimes of a set of algorithms, while *switching* allocates computational resources to each algorithm over a series of iterations.

between high- and low-knowledge approaches. As far as we are aware, all previous work on algorithm control has required building predictive models based on a detailed feature analysis of either problem structure or algorithm performance, or both. Such techniques require that the implementer has significant knowledge about the algorithm and problem features to pose the algorithm control problem. Even techniques that perform automated feature selection (Boyan and Moore 2000; Leyton-Brown, Nudelman, and Shoham 2002) require an original set of problem features from which to select. While existing methods have demonstrated strong performance on the algorithm selection problem, they do not tackle the issue of the external knowledge required to apply them to a new problem. This issue must be addressed to make optimization technology easier to use.

The contributions of this paper are the introduction of a low-knowledge approach to algorithm control and the demonstration that such an approach can achieve performance significantly better than the best pure algorithm and as good as a perfect high-knowledge approach.

2. THE ALGORITHM SELECTION PROBLEM

The algorithm selection problem is to choose the algorithm (or algorithm parameters) that leads to a high level of performance on a given problem instance (Rice 1976). From its original definition, this problem has been addressed by what we term “high-knowledge” approaches: detailed models of both problem features and algorithm performance are developed and correlated in a nontrivial model-building phase. The underlying justification, of course, is that algorithms that perform well on a problem instance with a particular cluster of features are likely to perform well on others instances with similar feature clusters.

Unfortunately, prediction models are typically limited to the problem classes and algorithms for which they were developed. For example, Leyton-Brown et al. (2002) developed strong selection techniques for the winner determination problem for combinatorial auctions. They take into account 35 problem features and their interactions based on three problem representations. Boyan and Moore (2000) developed STAGE, which attempts to correlate problem features with search performance to build a new objective function that improves

search. Lagoudakis and Littman (2000) use a Markov decision process with reinforcement learning applied to algorithm control. While this work takes a low-knowledge view of algorithm behavior it still a high-knowledge approach as it requires the determination of useful problem features and a suitable learning model. Other work applying machine learning techniques to algorithm generation (Minton 1996) and algorithm parameterization (Horvitz et al. 2001; Kautz et al. 2002; Ruan, Horvitz, and Kautz 2002) is also knowledge intensive, developing models specialized for particular problems and/or search algorithms and algorithm components.

2.1. Low-Knowledge Algorithm Selection

Our motivation for addressing the algorithm selection problem and its more general form of algorithm control is to lessen the knowledge (i.e., the expertise) necessary to use optimization technology. While existing algorithm control techniques have shown impressive results, their knowledge-intensive nature means that domain and algorithm expertise is necessary to develop the models. The requirement for expertise has not, therefore, been reduced; it has been shifted from algorithm building to predictive model building.

It could still be argued that the expertise is reduced if the predictive model can be applied to different types of problems. Unfortunately, the performance of a predictive model tends to be inversely related to its generality. While models accounting for over 99% of the variance in search cost exist, they are not only algorithm and problem specific, but problem instance specific as well (Watson 2003). The model-building *approach* is general, but the requirement for expertise remains: in-depth knowledge of the domain and different problem representations is necessary to identify a set of features that may be predictive of algorithm performance.

The distinction between low- and high-knowledge (or knowledge-intensive) approaches focuses on the number, specificity, and computational complexity of the measurements that need to be made of a problem instance. A low-knowledge approach has very few, inexpensive metrics, applicable to a wide range of algorithms and problem types. A high-knowledge approach has more metrics, which are more expensive to compute, and more specific to particular problems and algorithms. This distinction is independent of the approach taken to build a predictive model. In particular, sophisticated model-building techniques (e.g., based on machine learning) are consistent with low-knowledge approaches provided the observations on which they are built do not require significant expertise to identify. The extreme low-knowledge approach pursued in this paper, requires *no* knowledge of problem features.

Given our motivation, our approach is to investigate algorithm control techniques to reduce the engineering effort required to apply existing optimization techniques to a new problem. In our experience, the most significant component of such an engineering effort is the development of problem specific knowledge. An inexperienced engineer must first develop an understanding of the "difficult" parts of a given problem and devise ways to address them. It is typically through experimentation and trial and error that the challenging problem characteristics are revealed. In contrast, an expert engineer already has a foundation of knowledge and the difficult characteristics of a problem are sometimes readily apparent or can be understood with minimal experimentation.

Automated, high-knowledge approaches to algorithm control seek to reproduce the knowledge of an expert engineer but, as argued above, still require external knowledge to identify likely problem features. Our approach seeks to achieve good algorithm performance without such knowledge. Our goal is reduced engineering effort; our means is low-knowledge algorithm control.

2.2. Pure Algorithm Assumptions

We make some basic assumptions about the pure algorithms for which our techniques are appropriate. First, after a brief start-up time (at most, a few seconds) an algorithm is always able to return the best, complete solution it has found so far. Secondly, we require that an algorithm is able to take an external solution and search for solutions that are better. If an algorithm has not found a better solution, when asked for its best solution, the algorithm returns the external solution.

We believe these assumptions are very general: the anytime nature and being able to improve a solution. In fact, the algorithms we use in our experiments are quite diverse: two use only the upper bound of the best-known solution, the other modifies a copy of the solution by making moves.

3. PROBLEM DOMAIN: SCHEDULING

To introduce the idea of algorithm selection, we describe the problem domain, job shop scheduling, the scenario where these problems occur, and three algorithms that are used to solve them.

3.1. Scenario

A problem instance is presented to a scheduling system and that system has a fixed CPU time of T s to return a solution. We assume that the system designer has been given a learning set of problem instances at implementation time and that these instances are representative of the problems that will be later presented. We also assume that there exists a set of algorithms, A, that are applicable to the given problem.

3.2. Problem Instances

The job shop scheduling problem involves scheduling n jobs across m machines. Each job j consists of m ordered operations, such that each operation is scheduled one after another. For each job, every operation requires a unique machine for a specified duration, and the sequence of these requirements differs among jobs. We look at the objective of minimizing makespan: the total time required from the start of the earliest scheduled operation to the finish of the latest scheduled operation.

Three sets of 20×20 job shop scheduling problems are used. A total of 100 problem instances in each set were generated and 60 problems per set were arbitrarily identified as the learning set. The rest were placed in the test set.

The difference among the three problem sets is the way in which the activity durations are generated.

In the random (Rand) set, durations are drawn randomly with uniform probability from the interval [1, 99].

The machine-correlated (MC) set has activity durations drawn randomly from a normal distribution. The mean and standard deviation are the same for the activities on the same machine but different on different machines. The durations are, therefore, MC.

In the job-correlated (JC) set the durations are also drawn randomly from a normal distribution. The means and standard deviations are the same for activities in the same job but independent across jobs. Analogously to the MC set, these problems are JC.

These problem structures have been studied for flow-shop scheduling (Watson et al. 2002). They were chosen based on the intuition that the different structures may differentially favor one pure algorithm and therefore the algorithms would exhibit different relative performance on the different sets.

3.3. Algorithms

Three pure algorithms are taken from the scheduling literature and implemented in C++ using ILOG Scheduler 5.3, a constraint programming library. These algorithms were chosen from a set of eight algorithms because they have generally comparable behavior on the learning set. The other techniques performed much worse (sometimes by an order of magnitude) on every problem. The three algorithms are as follows:

1. tabu-tsab: a sophisticated tabu search due to Nowicki and Smutnicki (1996). The neighborhood is based on swapping pairs of adjacent activities on a subset of a randomly selected critical path. An important aspect of tabu-tsab is the use of an evolving set of the five best solutions found. Search returns to one of these solutions and moves in a different direction after a fixed number (1000 in our experiments) of iterations without improvement.
2. texture: a constructive search technique using texture-based heuristics (Beck and Fox 2000), strong constraint propagation (Nuijten 1994; Laborie 2003), and bounded chronological backtracking. The texture-based heuristic identifies a resource and time point with maximum competition among the activities and chooses a pair of unordered activities, branching on the two possible orders. The heuristic is randomized by specifying that the resource and time point is chosen with uniform probability from the top 10% most critical resources and time points. The bound on backtracks follows the pattern of $BT = \{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, 1, 1, 2, 4, 8, 16, \ldots\}$. Formally, $BT_i = \lceil 2^{k-2} \rceil$ where $k = n - (m(m+1)/2) + 1$, $m = \lfloor (\sqrt{8n+1} - 1)/2 \rfloor$ and $n = i + 2$. This was inspired by the optimal, zero-knowledge pattern of Luby, Sinclair, and Zuckerman (1993), but is more aggressive in how it increases the bound (Wu and van Beek 2003).
3. settimes: a constructive search technique using the SetTimes heuristic (Scheduler 2001), the same propagation as texture, and slice-based search (Beck and Perron 2000), a type of discrepancy-based search. The heuristic chronologically builds a schedule by examining all activities with minimal start time, breaking ties with minimal end time, and then breaking further ties arbitrarily. The discrepancy bound follows the pattern: $2, 4, 6, 8, \ldots$.

We assume that these pure algorithms are able to find a sequence of increasingly good solutions. As we are minimizing a cost function, an initial solution (with a very high cost) is always easily available. Each algorithm successively finds better solutions as it progresses either through local search or branch-and-bound, terminating when it has either proved optimality or exceeded a time limit. At any given time, each algorithm can return the best complete solution that it has found so far.

4. EXPERIMENTS WITH PURE ALGORITHMS

Our primary evaluation criterion is mean relative error (MRE), a measure of the mean extent to which an algorithm finds solutions worse than the best-known solutions. MRE is defined as follows:

$$\mathrm{MRE}(a, K, R) = \frac{1}{|R|} \frac{1}{|K|} \sum_{r \in R, k \in K} \frac{c(a, k, r) - c^*(k)}{c^*(k)}, \tag{1}$$

TABLE 1. Mean Fraction of Problems in Each Learning Problem Set for Which the Best Solution Was Found by Each Algorithm (MFB) and Its Mean Relative Error (MRE)

	MC		Rand		JC		All	
	MFB	MRE	MFB	MRE	MFB	MRE	MFB	MRE
tabu-tsab	0.39833	**0.00740***	0.19333	0.01425	0.2	0.01308	0.26389	0.01158
texture	0.03833	0.02027	0.06	0.01608	0.87333	**0.00197***	0.32389	0.01277
settimes	0.03333	0.04243†	0	0.04210†	0.38333	0.01152	0.13889	0.03202†

'*' indicates significantly better (i.e., lower) MRE than all other pure techniques, while '†' indicates significantly *worse* performance than all other pure techniques.

where R is a set of independent runs with different random seeds, K is a set of problem instances, $c(a, k, r)$ is the lowest cost solution found by algorithm a on problem instance k during run r, and $c^*(k)$ is the lowest cost solution known for problem instance k. For the learning set, $c^*(k)$ is the best performance of the pure algorithms. For the test set, $c^*(k)$ is the best performance of the pure algorithms and the switching techniques, which outperform the pure algorithms in some cases. Due to the stochastic nature of the algorithms, we run each algorithm 10 times on every problem instance.

In Table 1, we also show the mean fraction of problems in each set for which the algorithm found the best-known solution, known as mean fraction best (MFB).[1] MFB is defined as

$$\text{MFB}(a, K, R) = \frac{1}{|R|} \sum_{r \in R} \frac{|best(a, K, r)|}{|K|}, \tag{2}$$

where $best(a, K, r)$ is the set of solutions for $k \in K$ during run r where $c^*(k) = c(a, k, r)$.

The pure algorithms were run for $T = 1200$ CPU seconds on the learning set of problem instances. Table 1 displays the mean fraction of problems in each subset and overall for which each algorithm found the best solution (MFB) and the MRE. Across all problems, the difference in MRE performance between texture and tabu-tsab is not statistically significant.[2] However, there are significant differences among the problems sets: while tabu-tsab dominates in the MC problem set, the results are more uniform for the Rand problem set and texture is superior in the JC set.

These results demonstrate the need to solve an algorithm selection problem as no one pure algorithm dominates. In practice, a common approach is to simply choose the pure algorithm that is best on average on the learning set (e.g., the algorithm with the lowest MRE) and run that algorithm for all subsequent problem instances. Clearly, with no dominant algorithm, such an off-line algorithm selection approach will not result in the best performance possible on each problem instance. It is unclear, however, that an on-line approach (either high knowledge or low knowledge) will be better as some on-line computational time must be spent in selecting the algorithm.

[1] MFB reports the MFB of 10 runs on the same instance. In some cases the best solution will not be found in all runs. This results in lower MFB scores than if we had experimented with a single run of each algorithm.

[2] All statistical results in this paper are measured using a randomized paired-t test (Cohen 1995) and a significance level of $p \leq 0.005$.

5. PREDICTION

On-line algorithm selection chooses an algorithm to run only after the problem instance has been presented to the system. In this case, the time to make the selection must be taken into account. To quantify this, let t_p represent the prediction time and t_r the subsequent time allocated to run the chosen pure technique. It is required that the total runtime $T = t_p + t_r$. In the prediction techniques investigated here, during the prediction phase each pure algorithm, $a \in A$, is run for a fixed number of CPU seconds, t, on the problem instance. We require that $t_p = |A| \times t$. The quality of solutions found for each run is used to select the algorithm that will achieve the best performance given the time remaining. We assume that when a pure algorithm has been selected it does not have to restart: it can continue the search from where it left off in the prediction phase. The total runtime for the selected algorithm is $T_s = t_r + t$.

5.1. Simple Prediction Rules

In Beck and Freuder (2004), three variations of simple prediction rules were investigated on the same problem sets used in this paper. The rules considered the cost of solutions found, *pcost*, the slope of improvements, *pslope*, and an extrapolation of improvements, *pextrap*. The minimum, median and mean of these attributes was examined during the prediction phase. The result of the experiments was that only by choosing the minimum of *pcost* was a performance achieved that was significantly better than the best pure algorithm. For this reason, the only simple rule we consider here is *pcost_min*.

5.2. Bayesian Classifiers

A Bayesian classifier is constructed for each pure algorithm given a predefined prediction time, t_p, for the purpose of predicting the final performance of that algorithm at time T_s. In each classifier, for every 10 s of prediction time, a feature variable is created with possible values of “best” or “behind” for each pure algorithm. The value is assigned depending on whether the algorithm has found the best solution out of all the algorithms, $a \in A$, at that time point. More than one algorithm may have found the best solution at a particular time. Each classifier has a class variable representing the final performance of the algorithm (i.e., at time T_s). It also has the value “best” or “behind.” For the learning phase, feature and class values are input. For the test phase, only feature values are given and the classifier predicts the final performance of each algorithm.

For example, let $T = 1200$, $t_p = 90$, and $|A| = 3$, as shown in Figure 2. In this example, we input nine feature variables: three for each algorithm at 10, 20, and 30 s and the classifiers predict the final performance for each algorithm at $T_s = 1140$ s. The classifiers are used on the learning set to train a hybrid algorithm that runs each pure algorithm for 30 s, uses the classifiers to predict which of the algorithms will be best at 1140 s, and then runs the selected algorithm for the remaining 1110 s, continuing the search from where it left off.

We repeat this procedure, learning a new set of Bayesian classifiers for values of $t_p \in \{30, 60, 90, \dots, 1200\}$. This results in an assessment of hybrid algorithm performance for each prediction time. The prediction time with the best performance is t^*. The Bayesian classifiers for $t_p = t^*$ are then used on the test set.

In all cases, ties among the pure algorithms are broken by selecting the algorithm with the best mean solution quality on the learning set at time T. The Bayesian classifiers are learned using WinMine 2.0 (Chickering 2002) with default parameters. We refer to this techniques as *Bayes*.

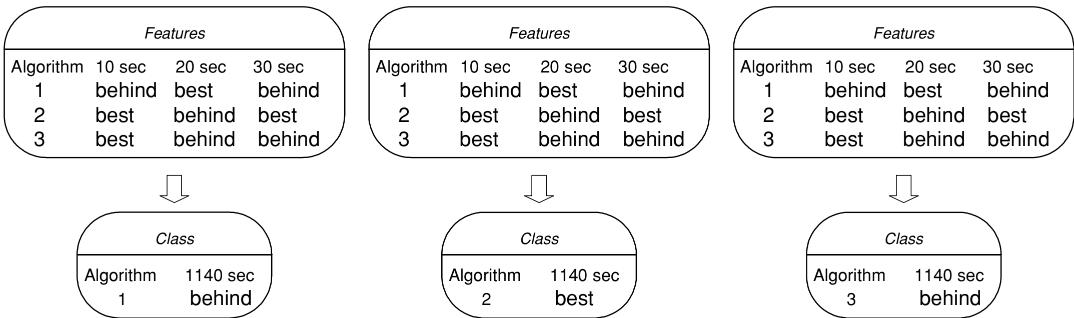

FIGURE 2. Three Bayesian classifiers are used together to predict the best algorithm at time $T_s = 1140$. In this example the prediction time is $t_p = 90$ s which means each algorithm is run for 30 s during the prediction phase. The performance trends of these 30 s runs are input as features to all of the classifiers, each of which predict the final performance of a specific algorithm.

5.3. Perfect Knowledge Classifiers

We can perform an a posteriori analysis of our data to compare our low-knowledge approach with the best performance that can be achieved with a high-knowledge classifier which seeks to select a pure algorithm for a given problem instance. The *best-subset* classifier chooses the pure algorithm that is best, on average, for the problem set to which the test instance belongs. This is the best result we can achieve with a high-knowledge classifier that infallibly and with zero CPU time can categorize instances into the three problem sets. The *best-instance* classifier makes a stronger assumption: that, with zero CPU time, the high-knowledge classifier can always choose the best pure algorithm for a given instance. No technique based on choosing a pure algorithm for a given instance can perform better.

5.4. Experiments: Prediction Control

For these experiments, once again the overall time limit, T, is 1200 CPU seconds. Each pure algorithm is run for T s with results being logged whenever a better solution is found. This design lets us process the results to examine the effect of different settings for the prediction time, t_p, and different values for $T \leq 1200$. As noted, the number of algorithms, $|A|$, is 3.

Recall that the learning set is used to build a set of Bayesian classifiers for each possible prediction time and identify the one with the smallest MRE. The best t_p for the Bayesian classifiers was found to be 270 s, meaning that each pure algorithm is run for 90 s and then the Bayesian classifiers are used to select the pure algorithm to be run for the subsequent 930 s.

Again we use the measure of MRE to compare algorithm performance, this time on a test set of new problem instances. Recall that MRE is calculated by comparing how well an algorithm did on a problem instance k against the best-known solution, $c^*(k)$. The best-known solution is not necessarily the optimal solution, but rather the best solution found during our experiments. While no prediction technique can perform better than the best pure technique, some of the switching techniques, which we will present in Section 6.1, do perform better than the best pure technique on the test problem set. This has the effect of increasing the MRE of the pure algorithms on the test set, as we are now finding better solutions with our

TABLE 2. The Performance of Each Technique on the Test Set

	MC		Rand		JC		All	
	MFB	MRE	MFB	MRE	MFB	MRE	MFB	MRE
tabu-tsab	0.0075	0.01985	0.005	0.02115	0.175	0.01574	0.0625	0.01891
texture	0.0175	0.02919	0.0225	0.02056	0.7225	0.00894	0.2542	0.01956
settimes	0	0.04602	0	0.05185	0.325	0.01684	0.10833	0.03823
Bayes	0.0175	0.01918	0.015	0.01863	0.625	0.00737	0.21917	0.01506
pcost_min	0.015	0.02038	0.0075	0.02110	0.6675	0.00729	0.23	0.01626
best-subset	0.0075	0.01985	0.0225	0.02056	0.7225	0.00894	0.25	0.01645
best-instance	0.025	0.01482‡	0.0275	0.01498‡	0.7225	0.00491‡	0.25833	0.01157‡
rl-double	0.0525	**0.0141‡**	0.0575	0.0142	0.7575	**0.0042***	0.28917	**0.0108‡**
rl-static	0.015	0.0243†	0.0075	0.0265	0.76	0.0049	0.26083	0.0185

'*' indicates significantly lower mean relative error (MRE) than all pure techniques, '‡' indicates significantly lower MRE than all other control algorithms, and '†' indicates significantly *worse* performance than the best pure technique. MFB (mean fraction best) is the mean fraction of problems in each set for which the algorithm found the best-known solution.

hybrid approaches. Similarly, the mean fraction of best solutions found decreases for the pure algorithms.

Table 2 presents the results of the experiment and the statistical significance of the prediction and perfect classifier techniques compared with the pure algorithms and with all prediction algorithms on the test set.[3] Only the *best-instance* perfect classifier finds an MRE significantly different (lower) than all other prediction techniques. In other words, there are no significant differences among the two on-line prediction variations, and the *best-subset* classifier. This demonstrates that even a simple rule can be applied in a low-knowledge context to achieve competitive performance.

When we compare the results of the pure algorithms in Table 2 against Table 1, we see a similar ranking of MRE performance, despite the MRE values increasing due to the better values of $c^*(k)$ for the reason mentioned above. However, when we look at the mean fraction of best solutions (MFB) found for an instance, the performance varies significantly. This is due to the switching approaches finding better solutions to problem instances, indicating that the hybrid approach is often superior for these problem instances.

To address the issue of different time limits, we performed a series of experiments using the same design as above with $T \in \{120, 240, \ldots, 1200\}$. For each time limit, the learning set was used to create a set of Bayesian classifiers and to identify the optimal prediction time, t^* for both *Bayes* and *pcost_min*. Table 3 presents the results. The "Best Pure" column is the MRE of the best pure technique on the test set for each time limit. For $T \in \{120, \ldots, 960\}$, texture achieves the lowest MRE of all pure techniques. For $T \geq 1080$, tabu-tsab is best. The MRE of both *Bayes* and *pcost_min* are not significantly better than the best pure algorithm for any time limit and they are both worse than the best pure algorithm except for $T = 120$. Though not shown in Table 3, *best-subset* is significantly better than all prediction techniques (but not the switching techniques) at all time limits.

We conclude that the results for $T = 1200$ are applicable to other choices for an overall time limit. The prediction approaches are no worse than the best pure technique except for

[3] The results for the switching algorithms presented Section 6.1 are also shown in the table. These will be discussed below.

TABLE 3. The Mean Relative Error (MRE) of Each Algorithm for Different Time Limits

Time	Algorithm							
Limit	Best Pure	Bayes	pcost_min	rl-double	rl-static	nl-double	nl-static	no-com
120	0.0387	0.0406†	0.0409†	0.0356	0.0340	0.0364	0.0347	0.0560†
240	0.0311	0.0298	0.0315	0.0270	**0.0246***	0.0278	0.0270	0.0440†
360	0.0277	0.0258	0.0266	**0.0216***	**0.0211***	0.0232	0.0238	0.0387†
480	0.0256	0.0232	0.0238	**0.0191***	**0.0197***	**0.0203***	0.0225	0.0353†
600	0.0241	0.0208	0.0216	**0.0168‡**	0.0190	**0.0188***	0.0220	0.0330†
720	0.0225	0.0196	0.0203	**0.0147‡**	0.0187	**0.0174***	0.0218	0.0311†
840	0.0215	0.0181	0.0193	**0.0134‡**	0.0186	**0.0158***	0.0217	0.0298†
960	0.0207	0.0169	0.0181	**0.0125‡**	0.0186	**0.0146***	0.0217†	0.0288†
1080	0.0198	0.0158	0.0171	**0.0116‡**	0.0186	**0.0139***	0.0217†	0.0277†
1200	0.0189	0.0151	0.0163	**0.0108‡**	0.0185	**0.0132***	0.0217†	0.0268†

'*' indicates an MRE significantly lower than all pure techniques at the same time limit, '‡' signifies an MRE significantly lower than all other tested algorithms at the same time limit, while '†' indicates significantly worse than the best pure technique.

very low time limits. The *pcost_min* technique does not perform significantly worse than *Bayes*, demonstrating that even a simple rule can be applied in a low-knowledge context to achieve competitive performance.

5.5. Summary: Prediction Control

Our results demonstrate that a prediction-based low-knowledge algorithm selection technique can perform as well as choosing the best pure algorithm (on average) based on a learning set and as well as a reasonable, though idealized, high-knowledge algorithm selection technique that is able to infallibly classify problem instances into different underlying sets. Performance is not as good as the optimal high-knowledge approach that is able to infallibly choose the pure algorithm that will lead to the best solution for each problem instance.

Even as simple a rule as choosing the algorithm that returned the best solution in the prediction phase achieves this level of performance. A more sophisticated prediction technique, based on learning Bayesian classifiers performs no better than the simple rule. While these experiments do not allow us to make general conclusions about the use of machine learning techniques (or even Bayesian classifiers) within a predictive control paradigm, it does suggest that our simple learning model provides insufficient information for the learning mechanism. Given that our goal is to minimize the expertise necessary in choosing, building more complicated learning models to use machine learning techniques as part of the predictive paradigm does not appear to be a promising direction.

6. CONTINUOUS CONTROL

Our second approach to algorithm selection generalizes the single decision of the predictive paradigm to a control structure where the selection decision can be made multiple times and, more generally, the decision is not one of selection but of allocation of computational resources to the pure algorithms. There are a number of ways that this paradigm

could be instantiated. In this paper, we investigate running a series of iterations where the control problem consists of deciding what portion of the iteration time is given to each pure algorithm.

6.1. Algorithm Switching

The switching paradigm allocates runtime to the pure algorithms during search. In contrast to predictive selection, the allocation decision is made multiple times. N iterations are performed such that each iteration, i, has a time limit of t_i CPU seconds and $\sum_{1 \leq i \leq N} t_i = T$. During each iteration, the runtime of each pure algorithm, $a \in A$, is determined using a weight, $w_i(a)$, which is learned as the search progresses. The weights for an iteration are normalized to sum to 1, and therefore, $w_i(a)$ corresponds to the fraction of time t_i allocated to algorithm a. For example, if algorithm a has weight $w_i(a) = 0.2$ and $t_i = 60$ s, then a is run for 12 s during iteration i. All weights are initialized to $1/|A|$.

Weights are updated after each iteration by considering the current weight and the performance of each algorithm during the last iteration. Performance is measured in cost improvement per second, which allows us to compare algorithms despite the differing run times. We normalize the cost improvement per second to sum to 1 producing a performance value, $p_i(a)$. The weights are then adjusted using a standard reinforcement learning formula: $w_{i+1}(a) = (1 - \alpha) \times w_i(a) + \alpha \times p_i(a)$. The α value controls the influence of the previous weight and the performance value. We show a diagram of the switching approach in Figure 3.

6.1.1. Sharing Solutions. Unlike the predictive paradigm, switching shares information among algorithms. Each invocation of an algorithm begins with the best solution found so far. However, different pure algorithms are able to exploit this information differently. The constructive algorithms, texture and settimes, only make use of the bound on the solution quality, searching for a solution that is strictly better than the best solution found so far. Each iteration of tabu-tsab, however, must begin with an initial solution. The best solution found so far is used. Recall that at any given time, each algorithm can return the best complete solution that it has found so far. If, in a given time interval, no better solution has been found, the solution found in the previous time interval is returned.

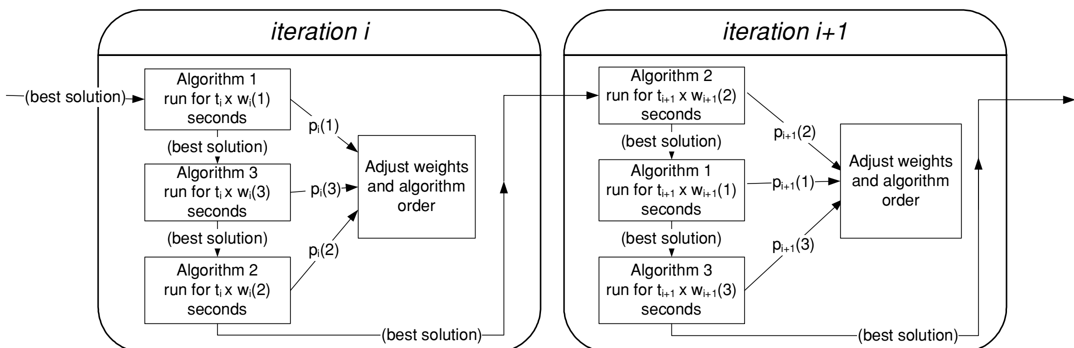

FIGURE 3. Algorithm switching passes the best solution found from one algorithm to the next. Each algorithm is run for a proportion of the iteration time t_i based on the algorithm's weight value $w_i(a)$. At the end of an iteration, the performance, $p_i(a)$ of each algorithm a is used to adjust the weight values used in the next iteration. Here we show two iterations.

6.1.2. Iteration Times. Two switching variations are used here. In *rl-static*, the length of each iteration does not vary: $t_i = 60$, for all i. For *rl-double* the time for the first two iterations is fixed, $t_1, t_2 = 60$, and subsequently $t_{i+1} = 2t_i$. The motivation for increasing iteration times comes from the observation that the algorithms take more time to find better solutions as we minimize the cost.

6.1.3. Algorithm Order. We switch between algorithms $|A| - 1$ times during an iteration, running each algorithm exactly once. For each iteration, the order in which the algorithms are run is randomly generated with uniform probability. Learning a good order of algorithms is an interesting area for future work.

6.1.4. Learning Rate. Similarly, no weight updating is done until after the second iteration, as it is very easy to find large improvements in solution quality in the first iteration. In both *rl-static* and *rl-double*, $\alpha = 0.5$. Experimenting with different values of α and t_i is an interesting direction we intend to pursue in our future work.

6.1.5. Learning Set. One distinct advantage of the switching approach is that it does not rely on a learning set. All learning is done on-line, that is only after the problem instance has been presented. This means the system does not require any off-line configuration and so the switching technique is not dependent on the learning set being representative of the real problems that will be encountered.

However, a learning set may be useful for tuning the switching parameters such as the learning rate α, the duration of iteration times, and the ordering of algorithms. Perhaps the most useful benefit from a learning set is the ability to filter out algorithms that are not competitive at all with a particular problem class.[4]

6.2. Experiments: Algorithm Switching

Table 2 displays the results on the three problem sets averaged over 10 independent runs, with $T = 1200$. We use the same method to compute MRE as described in Section 3.3. The two reinforcement learning algorithms achieve strong performance. The better one, *rl-double*, performs significantly better than all other algorithms based on MRE and is able to find the best-known solution for over half the instances. One advantage of the switching paradigm is that it can use different pure algorithms in different parts of the search, and therefore, can achieve better performance on a problem instance than the best pure technique. In contrast, predictive selection techniques cannot perform better than the best pure technique. In fact, *rl-double* finds a better solution than any pure technique in 45 of the test problems. The *rl-static* technique performs significantly worse than *rl-double*. The doubling of the time in the iterations appears to be an important part of the performance of *rl-double*. Nonetheless, *rl-static* is able to achieve performance that is not significantly worse than the best pure technique. Furthermore, it finds a better solution than any pure technique in 16 of the problems.

As with the prediction techniques, we also varied the overall time limit. Table 3 presents the results. The *rl-static* approach achieves significantly lower MRE than all pure techniques for three time limits: $T \in \{240, 360, 480\}$. Finally, *rl-double* achieves MRE significantly lower than all pure techniques for $T \geq 360$ and comparable performance for $T \leq 240$. For

[4]As mentioned in Section 3.3, we performed this filtering by hand but this could easily be automated.

TABLE 4. The Mean Relative Error (MRE) for Different Overall Time Limits

Time Limit	Algorithm		
	rl-double	best-subset	best-instance
120	0.0356	0.0366	**0.0313**‡
240	0.0270	0.0291	**0.0242**‡
360	**0.0216***	0.0259	0.0205
480	**0.0191***	0.0234	0.0184
600	**0.0168***	0.0219	0.0170
720	**0.0147***	0.0203	0.0156
840	**0.0134***	0.0190	0.0144
960	**0.0125***	0.0180	0.0133
1080	**0.0116***	0.0171	0.0122
1200	**0.0108***	0.0164	0.0115

'*' indicates a significantly lower MRE when comparing *rl-double* against *best-subset* while '‡' signifies the same thing comparing *best-instance* against *rl-double*.

$T \geq 600$, *rl-double* achieves significantly better performance than all other algorithms in Table 3.

To assess reinforcement learning, we run *rl-static* and *rl-double* with $\alpha = 0$, which disables reinforcement learning by ignoring the current performance result and keeps the weights constant. Each pure algorithm receives an equal portion of the iteration time, regardless of previous performance. We call these "no-learning" variations, *nl-static* and *nl-double*. Results, also in Table 3, demonstrate that reinforcement learning has a strong impact. Although the significant difference is not marked in the table, *rl-static* is significantly better than *nl-static* for all T, and *rl-double* is significantly better than *nl-double* at all time limits except $T = 120$. Interestingly, *nl-double* not only achieves significantly better performance than the pure techniques on seven of the 10 time limits but also achieves a significantly lower MRE than all algorithms except *rl-double* for $T = 840$ and $T = 960$.

Finally, to investigate the impact of communication among the algorithms, the *no-com* technique is *nl-double* without sharing of the best-known solution among the algorithms; each algorithm is allowed to continue from where it left off in the previous iteration.[5] *no-com* performs worse than any other technique at all time limits; communication between algorithms is clearly an important factor for performance.

Turning to the comparison of the switching algorithms to the perfect knowledge classifiers, in Table 4 we present a very promising result. The *rl-double* technique achieves a lower MRE than *best-subset* for $T \geq 360$. Furthermore, *rl-double* performs as well as *best-instance* for all time limits $T \geq 360$. In other words, without complex model building our best low-knowledge approach achieves performance that is as good as the best possible high-knowledge predictive classification approach.

6.3. Algorithm Switching Summary

The experiments in this section show that the algorithm switching approach to algorithm control can achieve strong performance. Not only is the strongest variation (*rl-double*)

[5] Another way to see this is that *no-com* runs each pure technique for $\frac{T}{|A|}$ s and returns the best solution found.

significantly better than the best pure technique, it achieves performance that is equivalent to the optimal high-knowledge predictive approach. There are three main components of *rl-double*: communication of the best solution among the pure algorithms, the doubling of the iteration time, and the reinforcement learning. Comparisons with variations that selectively remove each of these components shows that they all contribute to the overall performance.

The fact that we have created a technique that achieves significantly better performance than the best pure technique, and performance equivalent to an optimal high-knowledge classifier, is not the main conclusion to be drawn from these experiments. Rather, the importance of these results stems from the fact that the technique does not depend on expertise, either in terms of development of new pure scheduling algorithms, or in terms of development of a detailed domain and algorithm model. A low-knowledge approach is able to achieve better performance while *reducing* the required expertise.

7. FUTURE WORK

We have examined a number of algorithm control approaches on a single problem type. While there is no reason to expect job-shop scheduling problems (JSPs) to be so unlike other optimization problems that similar results would not be observed, it is necessary to evaluate low-knowledge approaches on other optimization problems. In addition, it may be more relevant to apply algorithm control to problems with particular characteristics. A real system is typically faced with a series of related problems: a scheduling problem gradually changes as new orders arrive and existing orders are completed. As the problem changes, algorithm control techniques may have the flexibility to appropriately change the manner in which underlying pure algorithms are applied.

We have examined a very simple version of reinforcement learning and the best control technique, *rl-double*, made no use of the off-line learning set. Two obvious uses for the off-line problems are as a basis for setting the α value and as a basis for iteration-specific weights. If it is true that different pure algorithms are more appropriate for different phases of the search, we could use the learning set to weight each algorithm in each iteration. The issue of the ordering of the pure algorithms within an iteration is also an interesting area for future work. It is clear from the result of *nl-double* that the simple act of switching among a set of algorithms brings problem solving power. Understanding the reasons behind this may help to further leverage such simple techniques.

Finally, we intend to investigate the scaling of these techniques as the number of pure algorithms increases. It seems likely that the prediction-based techniques would suffer: prediction time will need to be longer and/or the time given to each pure technique will be reduced. These implications will both lead to poorer performance as less time is left for the chosen algorithm and the algorithm selection will be less accurate. In contrast, switching with reinforcement learning, especially if off-line learning is used to set iteration-specific weights, may be able to isolate the subset of pure algorithms that are most useful.

8. CONCLUSION

We have shown that low-knowledge metrics of pure algorithm behavior can be used to form a system that consistently and significantly outperforms the best pure algorithm. Machine learning techniques play an important role in this performance, however, even a simple-minded approach that evenly distributes increasing runtime among the pure algorithms performs very well.

Our motivation for investigating low-knowledge algorithm control was to reduce the expertise necessary to exploit optimization technology. Therefore, the strong performance of our techniques should be evaluated not simply from the perspective of an increment in solution quality, but from the perspective that this increment has been achieved without additional expertise. We neither invented a new pure algorithm nor developed a detailed model of algorithm performance and problem instance features. Therefore, we believe low-knowledge approaches are an important area of study in their own right.

ACKNOWLEDGMENT

This work has received support from Science Foundation Ireland (grant 00/PI.1/C075), Irish Research Council for Science, Engineering, and Technology (grant SC/2003/82), and ILOG, SA.

REFERENCES

BECK, J. C., and M. S. FOX. 2000. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. Artificial Intelligence, **117**(1):31–81.

BECK, J. C., and L. PERRON. 2000. Discrepancy-bounded depth first search. *In* Proceedings of the Second International Workshop on Integration of AI and OR Technologies for Combinatorial Optimization Problems (CPAIOR'00), Paderborn, Germany.

BECK, J., and E. FREUDER. 2004. Simple rules for low-knowledge algorithm selection. *In* Proceedings of the First International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR'04), pp. 50–64, Nice, France.

BOYAN, J., and A. MOORE. 2000. Learning evaluation functions to improve optimization by local search. Journal of Machine Learning Research, **1**:77–112. http://citeseer.ist.psu.edu/boyan00learning.html

CHICKERING, D. M. 2002. The WinMine toolkit, Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA.

COHEN, P. R. 1995. Empirical Methods for Artificial Intelligence. MIT Press, Cambridge, MA.

HORVITZ, E., Y. RUAN, C. GOMES, H. KAUTZ, B. SELMAN, and M. CHICKERING. 2001. A bayesian approach to tacking hard computational problems. *In* Proceedings of the Seventeenth Conference on uncertainty and Artificial Intelligence (UAI-2001), pp. 235–244, Seattle, WA.

KAUTZ, H., E. HORVITZ, Y. RUAN, C. GOMES, and B. SELMAN. 2002. Dynamic restart policies. *In* Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02), pp. 674–681, Edmonton, Alberta, Canada.

LABORIE, P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. Artificial Intelligence, **143**:151–188.

LAGOUDAKIS, M. G., and M. L. LITTMAN. 2000. Algorithm selection using reinforcement learning. *In* Proceedings of the Seventeenth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA, pp. 511–518. citeseer.nj.nec.com/lagoudakis00algorithm.html

LE PAPE, C., L. PERRON, J. RÉGIN, and P. SHAW. 2002. Robust and parallel solving of a network design problem. *In* Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP02), pp. 633–648, Ithaca, NY.

LEYTON-BROWN, K., E. NUDELMAN, and Y. SHOHAM. 2002. 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 (CP02), pp. 556–572, Ithaca, NY.

LUBY, M., A. SINCLAIR, and D. ZUCKERMAN. 1993. Optimal speedup of Las Vegas algorithms. Information Processing Letters, **47**:173–180.

MINTON, S. 1996. Automatically configuring constraint satisfaction programs: A case study. Constraints, **1**(1,2):7–43.

NOWICKI, E., and C. SMUTNICKI. 1996. A fast taboo search algorithm for the job shop problem. Management Science, **42**(6):797–813.

NUIJTEN, W. P. M. 1994. Time and resource constrained scheduling: A constraint satisfaction approach, PhD Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology.

RICE, J. 1976. The algorithm selection problem. Advances in Computers, **15**:65–118.

RUAN, Y., E. HORVITZ, and H. KAUTZ. 2002. Restart policies with dependence among runs: A dynamic programming approach. *In* Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-2002), Springer-Verlag, Berlin, pp. 573–586, Ithaca, NY.

SCHEDULER. 2001. ILOG Scheduler 5.2 User's Manual and Reference Manual, ILOG, SA.

WATSON, J.-P. 2003. Empirical Modeling and Analysis of Local Search Algorithms for the Job-Shop Scheduling Problem, PhD Thesis, Department of Computer Science, Colorado State University.

WATSON, J.-P., L. BARBULESCU, L. WHITLEY, and A. HOWE. 2002. Contrasting structured and random permutation flow-shop scheduling problems: Search-space topology and algorithm performance. INFORMS Journal on Computing, **14**(2):98–123.

WU, H., and P. VAN BEEK. 2003. Restart strategies: Analysis and simulation. *In* Ninth International Conference on Principles and Practice of Constraint Programming, p. 1001, Kinsale, Ireland.