Combining Multiple Heuristics Online

Matthew Streeter Daniel Golovin Stephen F. Smith
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{matts,dgolovin,sfs}@cs.cmu.edu

Abstract

We present black-box techniques for learning how to interleave the execution of multiple heuristics in order to improve average-case performance. In our model, a user is given a set of heuristics whose only observable behavior is their running time. Each heuristic can compute a solution to any problem instance, but its running time varies across instances. The user solves each instance by interleaving runs of the heuristics according to a *task-switching schedule*. We present (*i*) exact and approximation algorithms for computing an optimal task-switching schedule offline, (*ii*) sample complexity bounds for learning a task-switching schedule from training data, and (*iii*) a no-regret strategy for selecting task-switching schedules online. We demonstrate the power of our results using data from recent solver competitions. We outline how to extend our results to the case in which the heuristics are randomized, and the user may periodically restart each heuristic with a fresh random seed.

Introduction

Many important computational problems seem unlikely to admit algorithms with provably good worst-case performance, yet must be solved as a matter of practical necessity. Examples of such problems include Boolean satisfiability, planning, integer programming, and numerous scheduling and resource allocation problems. In each of these problem domains, heuristics have been developed that perform much better in practice than a worst-case analysis would guarantee, and there is an active research community working to develop improved heuristics. Indeed, there are entire conferences devoted to the study of particular problem domains (e.g., Boolean satisfiability, planning), and annual solver competitions are held in order to assess the state of the art and to promote the development of better solvers.

Unfortunately, the behavior of a heuristic on a previously unseen problem instance is difficult to predict in advance. The running time of a heuristic may vary greatly across seemingly similar problem instances or, if the heuristic is randomized, across multiple runs on a single instance that use different random seeds (Gomes *et al.* 2000). For this reason, after running a heuristic unsuccessfully for some time one might decide either to restart the heuristic with a fresh random seed, or to suspend the execution of that heuristic and start running a different heuristic instead.

Previous work has shown that combining multiple heuristics into a *portfolio* can dramatically improve the mean running time (Huberman, Lukose, & Hogg 1997; Gomes & Selman 2001). A portfolio consists of a set of heuristics that are run in parallel (or interleaved on a single processor) according to some schedule, with all runs being terminated as soon as one of the heuristics returns a solution. The mean running time of a portfolio can be much better than that of any single heuristic, for example if each heuristic has a few rare instances on which its running time is very large.

Our paper presents new techniques for performing the schedule-selection step in algorithm portfolio design. In our model, we are given a pool of heuristics whose only observable behavior is their running time, and a sequence of problem instances to solve. Our goal is to perform schedule selection in a way that minimizes the total CPU time we spend on all instances. We consider the schedule selection problem in offline, learning-theoretic, and online settings, and provide new theoretical guarantees in each setting.

Our paper focuses on deterministic heuristics (so restarts play no role). However, we wish to emphasize that our results extend in interestings way to randomized heuristics, as we outline at the end of the paper.

Definitions and notation

In our model, we are given a set $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$ of deterministic heuristics, and a set $\{x_1, x_2, \ldots, x_n\}$ of instances of some decision problem. Heuristic h_j, when run on instance x_i, runs for $\tau_{i,j}$ time units before returning a (provably correct) "yes" or "no" answer, where $\tau_{i,j} \in \{1, 2, \ldots, B\} \cup \{\infty\}$. We assume that for each x_i, there is some h_j such that $\tau_{i,j} \leq B$. We may think of B as the maximum time we are willing to spend on any single heuristic when solving any particular instance. We solve each instance by interleaving the execution of the heuristics according to a *task-switching schedule*, stopping as soon as one of the heuristics returns an answer.

Definition (task-switching schedule). *A* task-switching schedule $S : \mathbb{Z}^+ \to \mathcal{H}$ *specifies, for each integer* $t \geq 0$*, the heuristic* $S(t)$ *to run from time* t *to time* $t + 1$.

For any task-switching schedule S, let $c_i(S)$ be the time required to solve x_i when interleaving the execution of the

heuristics according to S. Specifically, $c_i(S)$ is the smallest integer t such that, for some heuristic h_j, S devotes $\tau_{i,j}$ time steps to h_j during the interval $[0, t]$ (i.e., $\tau_{i,j} = |\{t' < t : S(t') = h_j\}|$ for some j). We denote by $\mathcal{S}_{ts}$ the set of all task-switching schedules. To make the meaning of some of our results more clear, we let c^* denote an upper bound on $c_i(S)$ for the instance sequence under consideration (clearly, $c^* \leq Bk$).

Summary and contributions

In this paper we consider the problem of selecting task-switching schedules in three different settings:

1. **Offline**. In the offline setting we are given the n by k matrix τ as input and wish to compute the task-switching schedule $S^* = \arg\min_{S \in \mathcal{S}_{ts}} \sum_{i=1}^{n} c_i(S)$. We show that a simple greedy algorithm gives a 4-approximation to the optimal task-switching schedule and that, for any $\alpha < 4$, computing an α-approximation is NP-hard. We also give exact and approximation algorithms based on shortest paths whose running time is exponential as a function of k (k is the number of heuristics) but is polynomial for any fixed k.
2. **Learning-theoretic.** In the learning-theoretic setting we draw training instances independently at random from a distribution, compute an optimal task-switching schedule for the training instances, and then use that schedule to solve additional test instances drawn from the same distribution. We give bounds on the number of instances required to learn a schedule that is *probably approximately correct*.
3. **Online.** In the online setting we are fed a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of instances one at a time and must obtain a solution to each instance before moving on to the next. We give no-regret schedule-selection strategies both for the *distributional* online setting (in which problem instances are drawn from a distribution) and the *adversarial* online setting (in which the matrix τ is secretly filled in by an adversary).

Experimentally, we use data from recent solver competitions to show that task-switching schedules have the potential to improve the performance of state-of-the-art solvers in several problem domains.

Related Work

Algorithm portfolios

Our work is closely related to, and shares the same goals as, previous work on *algorithm portfolios* (Huberman, Lukose, & Hogg 1997; Gomes & Selman 2001). An algorithm portfolio is a schedule for combining runs of various heuristics. The schedules considered in previous work simply run each heuristic in parallel at equal strength and assign each heuristic a fixed restart threshold. Gomes *et al.* (2001) have addressed the problem of constructing an optimal algorithm portfolio offline given knowledge of the run length distribution of each algorithm, under the assumption that each algorithm has the same run length distribution on all problem instances.

In this paper we consider a more powerful class of schedules that allow the proportion of CPU time allocated to each heuristic to change over time. This flexibility requires us to develop new algorithms even in the offline setting, and even in the special case when all heuristics are deterministic. Our paper also gives rigorous results for schedule selection in the learning-theoretic and online settings, which Gomes *et al.* (2001) identified as an important open problem.

Resource-sharing schedules

A recent paper by Sayag *et al.* (2006) considered the problem of selecting task-switching schedules and resource-sharing schedules, both in the offline and learning-theoretic settings. A *resource-sharing schedule* $S : \mathcal{H} \to [0, 1]$ specifies that all heuristics in $\mathcal{H}$ are to be run in parallel, with each $h \in \mathcal{H}$ receiving a proportion $S(h)$ of the CPU time. The primary contribution of their paper was an offline algorithm that computes an optimal resource-sharing schedule in $O(n^{k-1})$ time. They also discuss an $O(n^{k+1})$ algorithm for computing optimal task-switching schedules offline. As proved by Sayag *et al.* (Lemma 1), an optimal task-switching schedule always performs as good or better than an optimal resource-sharing schedule. Our main contributions relative to their paper are to give computational complexity results and a greedy approximation algorithm for computing an optimal task-switching schedule offline, and to give a no-regret strategy in the adversarial online setting. We also give an improved sample complexity bound in the learning-theoretic setting.

The Offline Setting

We first consider the problem of computing an optimal task-switching schedule in an offline setting. That is, given an n by k matrix τ as input, we wish to compute the schedule

$$S^* = \arg\min_{S \in \mathcal{S}_{ts}} \sum_{i=1}^{n} c_i(S) \, .$$

Computational complexity

When the number of heuristics k is unrestricted, the problem of computing an optimal task-switching schedule is NP-hard even to approximate. To see this, consider the special case $B = 1$ (so each $\tau_{i,j} \in \{1, \infty\}$). In this case, an optimal task-switching schedule can be represented simply as a permutation of the k heuristics. Viewing each heuristic h_j as the set of elements $\{x_i : \tau_{i,j} = 1\}$, our goal is to order these sets from left to right so as to minimize the sum, over all elements x_i, of the position of the leftmost set that contains x_i. This is exactly the *min-sum set cover* problem. For any $\alpha < 4$, achieving an approximation ratio of α for min-sum set cover is NP-hard (Feige, Lovász, & Tetali 2004). Thus we have the following theorem.

Theorem 1. *For any $\alpha < 4$, computing an α-approximation to the optimal task-switching schedule is NP-hard.*

An exact algorithm based on shortest paths

Although computing an optimal task-switching schedule is NP-hard in general, we might hope to find a polynomial

time algorithm when the number of heuristics k is small. In this section we show how to find an optimal task-switching schedule by computing a shortest path in a graph. The key is to establish a one-to-one correspondence between task-switching schedules and paths in a graph $G = \langle V, E \rangle$ such that the following property holds: for each instance x_i, we can define vertex weights $w_i : V \to \{0, 1\}$ such that for any task-switching schedule S, $c_i(S)$ equals the weight assigned by w_i to the path corresponding to S.

The vertices of our graph will be arranged in a k-dimensional grid, where each vertex has coordinates in $\{0, 1, \ldots, B\}^k$. The vertex with coordinates $\langle t_1, t_2, \ldots, t_k \rangle$ will correspond to the state of having run each heuristic h_j for t_j time units so far. There is a directed edge from u to v if v can be reached from u by advancing one unit along some axis. A task-switching schedule S corresponds to the path that starts at grid coordinates $\langle 0, 0, \ldots, 0 \rangle$ and, on step t of the path, advances along the axis corresponding to heuristic $S(t-1)$. The weights w_i are defined as follows: if vertex v has grid coordinates $\langle t_1, t_2, \ldots, t_k \rangle$ then $w_i(v) = 1$ if $t_j < \tau_{ij}$ for all j; otherwise $w_i(v) = 0$. In other words, w_i assigns weight 1 to vertices that correspond to states in which instance x_i has not yet been solved, and assigns weight 0 to all other vertices. It is easy to check that $c_i(S)$ is equal to the weight assigned by w_i to the path corresponding to S. It follows that the optimal task-switching schedule for instances $x_1, x_2, \ldots, x_n$ is the schedule that corresponds to the shortest path in G (from vertex $\langle 0, 0, \ldots, 0 \rangle$ to $\langle B, B, \ldots, B \rangle$) under the weight function $w = \sum_{i=1}^{n} w_i$ (see Figure 1).

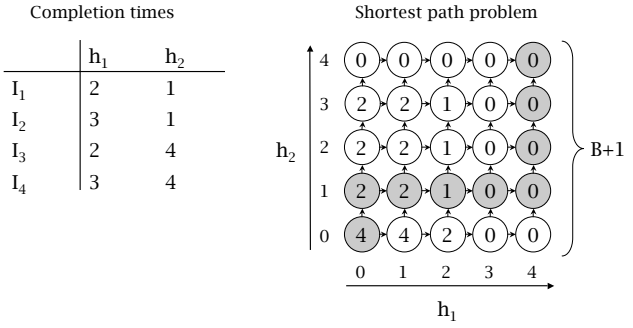

Figure 1: An example set of completion times and the corresponding shortest path problem (here $k = 2$ and $B = 4$). The optimal schedule (indicated by the shaded path) runs h_2 for one time unit, then runs h_1 for four time units, then runs h_2 for three additional time units.

The time required to find an optimal task-switching schedule using this algorithm is dominated by the time required to compute the vertex weights, which is $O(nk|V|) = O(nk(B+1)^k)$. We now outline two ways to reduce the time complexity, leaving the details as an exercise to the reader.

To obtain an α-approximation to the optimal task-switching schedule, we may restrict our attention to schedules that only suspend a heuristic when the time invested in that heuristic so far is a power of α. An optimal schedule within this restricted set can be found by solving a shortest path problem on an *edge*-weighted graph whose vertices form a k-dimensional grid with sides of length $\lceil \log_\alpha B \rceil + 1$, and the time complexity is $O(nk(\lceil \log_\alpha B \rceil + 1)^k)$.

If the number of problem instances n is less than B, an optimal schedule can be computed more efficiently by using the following fact (which can be proved using an interchange argument):

Fact 1. *If t is an integer such that $S^*(t) = h_j \neq S^*(t+1)$ and t_j is the number of time steps S^* devotes to h_j during the interval $[0, t]$, then $t_j = \tau_{i,j}$ for some i.*

By exploiting this fact we can reduce the time complexity to $O(kn^{k+1})$.[1]

Theorem 2. *An optimal task-switching schedule can be computed in time $O(nk \cdot \min\{B+1, n\}^k)$, and an α-approximation can be computed in time $O(nk \cdot \min\{\lceil \log_\alpha B \rceil + 1, n\}^k)$.*

A greedy approximation algorithm

In this section we give a greedy algorithm that runs in time $\mathrm{poly}(n, k)$ and produces a schedule whose total cost is at most 4 times optimal (by Theorem 1, achieving a better approximation ratio is NP-hard). Our algorithm and its analysis are generalizations of results of Feige *et al.* (2004).

The greedy algorithm can be described as following the rule "greedily maximize the number of instances solved per unit time". More specifically, the algorithm proceeds in a series of epochs. At the beginning of epoch z, R_z represents the set of instances that have not been solved by the schedule-so-far (initially $z = 1$ and R_1 contains all instances). The greedy algorithm then runs some heuristic h_j for (an additional) Δ_z time steps, where h_j and Δ_z are chosen so as to maximize the number of instances in R_z that are solved per unit time.

Greedy algorithm for constructing task-switching schedules:

1. Initialize $R_1 \leftarrow \{1, 2, \ldots, n\}$, $z \leftarrow 1$, $T \leftarrow 0$, and $t_j \leftarrow 0 \ \forall j, 1 \le j \le k$.
2. While $|R_z| > 0$:
   (a) For any integers j and t, let $X_j^z(t) = \{i \in R_z : \tau_{i,j} \le t\}$. Find the pair $(h_j, \Delta_z) \in \mathcal{H} \times \mathbb{Z}^+$ that maximizes
   $$\frac{|X_j^z(t_j + \Delta_z)|}{\Delta_z}.$$
   (b) Set $R_{z+1} \leftarrow R_z \setminus X_j^z(t_j + \Delta_j)$.
   (c) Set $S^G(t) \leftarrow h_j$ for all t, $T \le t < T + \Delta_z$.
   (d) Set $t_j \leftarrow t_j + \Delta_z$, $T \leftarrow T + \Delta_z$, and $z \leftarrow z + 1$.
3. Return the task-switching schedule S^G.

Theorem 3. *The greedy algorithm runs in time $O(nk \log n \cdot \min\{n, Bk\})$ and returns a schedule S^G that is a 4-approximation to the optimal task-switching schedule.*

[1] The $O(kn^{k+1})$ exact algorithm also appears as Theorem 12 of (Sayag, Fine, & Mansour 2006).

Proof. See Appendix A. □

Remark 1. The greedy algorithm just described can be modified to produce a schedule that requires only one run to be kept in memory at a time (i.e., instead of suspending a run we throw it away and may later start over from scratch) without degrading the worst-case approximation ratio. This modification is useful if it is costly to keep multiple runs in memory.

The Learning-Theoretic Setting

To apply the offline algorithms of the previous section in practice, we might collect a set of problem instances to use as training data, compute an (approximately) optimal schedule for the training instances, and then use this schedule to solve additional test instances. Under the assumption that the training and test instances are drawn (independently) from a fixed probability distribution, we would then like to know how many training instances are required so that the optimal schedule for the training instances will be near-optimal for the test instances with high probability.

For any task-switching schedule S, let $\bar{C}(S) = \frac{1}{m}\sum_{i=1}^{m} c_i(S)$ be the sample mean cost of S on the training instances, and let $C(S)$ be the (unknown) true mean. The following theorem gives a bound on the number of training instances required to ensure that, with high probability, the maximum difference between $C(S)$ and $\bar{C}(S)$ over all $S \in \mathcal{S}_{ts}$ is small. Our theorem improves upon the uniform convergence result of Sayag *et al.* (2006), whose sample complexity is $O\left(\frac{k^3B^3}{\epsilon^2}\ln\frac{k}{\delta}\right)$ (when comparing this bound to ours, recall that $c^* \leq Bk$).

Theorem 4. *Let $C(S)$ and $\bar{C}(S)$ be defined as above. For $m \geq m_0(\epsilon,\delta) = \frac{1}{2}\left(\frac{c^*}{\epsilon}\right)^2 \ln\left(\frac{2|V|}{\delta}\right)$, the inequality*

$$\max_{S\in\mathcal{S}_{ts}} \left|\bar{C}(S) - C(S)\right| \leq \epsilon$$

holds with probability at least $1-\delta$, where $|V| = (B+1)^k$.

Proof. See Appendix A. □

As a corollary of Theorem 4 we obtain a bound on the number of training instances required to compute a schedule that is probably approximately optimal. In particular, if $\bar{S}$ is an α-approximation to the optimal schedule for the training instances (i.e., $\bar{C}(\bar{S}) \leq \alpha \min_{S\in\mathcal{S}_{ts}} \bar{C}(S)$), and $S^* = \arg\min_{S\in\mathcal{S}_{ts}} C(S)$ is the true optimal schedule, Theorem 4 implies that $C(\bar{S}) \leq \bar{C}(\bar{S}) + \epsilon \leq \alpha\bar{C}(S^*) + \epsilon \leq \alpha C(S^*) + \epsilon(1+\alpha)$ holds with probability at least $1-\delta$ for $m \geq m_0(\epsilon,\delta)$.

The Online Setting

One weakness of Theorem 4 is that it assumes we can draw training (and test) instances independently at random from a fixed probability distribution. In practice, the distribution might change over time and successive instances might not be independent.

In this section we consider the problem of selecting task-switching schedules in an adversarial (worst-case) online setting. In this setting we are fed, one at a time, a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ of problem instances to solve. Prior to receiving instance x_i, we must select a task-switching schedule S_i. We then use S_i to solve x_i and incur cost $c_i(S_i)$. Our *regret* at the end of n rounds is equal to

$$\frac{1}{n}\left(\mathbb{E}\left[\sum_{i=1}^{n} C_i\right] - \min_{S\in\mathcal{S}_{ts}} \sum_{i=1}^{n} c_i(S)\right) \tag{1}$$

where C_i is the cost we incur on instance x_i,[2] and the expectation is over any random bits used by our schedule-selection strategy. That is, regret is the difference between our average cost and the average cost of the optimal schedule for the (unknown) set of n instances. A strategy's worst-case regret is the maximum value of (1) over all instance sequences of length n (i.e., over all possible n by k matrices τ). A *no-regret strategy* has worst-case regret that is $o(1)$ as a function of n.

Before describing our no-regret strategy, let us consider the special case in which each instance is drawn independently at random from a fixed distribution. In this case, Theorem 4 suggests a simple schedule-selection strategy: take the first $m = m_0(\epsilon,\delta)$ instances as training data, compute an optimal schedule for them, and use that schedule on the remaining $n-m$ instances. The regret of this strategy is at most $\frac{1}{n}\left(mBk + (n-m)(2\epsilon + \delta Bk)\right)$. By choosing ϵ and δ appropriately one can get regret $O(Bk(\frac{\ln(n|V|)}{n})^{1/3}) = o(1)$ in the distributional online setting. One can prove analogous guarantees if an α-approximation rather than an optimal schedule is computed for the training instances. Unfortunately, these strategies do not have good worst-case regret bounds, because in the worst case the optimal schedule for the first m instances might perform very poorly on the remaining $n-m$ instances.

To achieve good performance in the worst-case setting, we use a strategy based on the "label efficient forecaster" described by Cesa-Bianchi *et al.* (2005). When we receive instance x_i, with probability $\frac{m}{n}$ we *explore* by solving the instance with all k heuristics and adding it to our pool of training instances (initially the pool is empty). With probability $1-\frac{m}{n}$, we *exploit* by sampling a schedule from a distribution in which each schedule S is assigned probability proportional to $\exp(-\eta m_c \bar{C}(S))$, where $\bar{C}(S)$ is the average cost of schedule S on the training instances in our pool, m_c is the current number of training instances, and $\eta > 0$ is a learning rate parameter (we will describe how to sample from this distribution efficiently). Adapting Theorem 1 of Cesa-Bianchi *et al.* (2005) to our setting yields the following regret bound, which is $o(1)$ as a function of n.

Theorem 5. *The label-efficient forecaster with learning rate $\eta = (\frac{\ln N}{n\sqrt{2}})^{2/3}$ and exploration probability $\frac{m}{n} = \min\left\{1, \sqrt{\frac{\eta}{2}}\right\}$ has regret at most $2c^*\left(\frac{2\ln N}{n}\right)^{1/3}$, where $N \leq k^{Bk}$ is the number of task-switching schedules.*

To implement the label-efficient forecaster efficiently, we exploit the shortest path formulation introduced in the offline

[2] C_i might equal $c_i(S_i)$ or it might be larger, for example if we choose to solve x_i multiple times in order to gather training data.

Table 1: Results for the ICAPS 2006 optimal planning competition (cross-validation results are parenthesized).

Solver	Avg. CPU (s)	Num. solved
Greedy schedule (x-val)	358 (407)	98 (97)
Modified greedy (x-val)	476 (586)	96 (95)
SATPLAN	507	83
Maxplan	641	88
MIPS-BDD	946	54
CPT2	969	53
FDP	1079	46
Parallel schedule	1244	89
IPPLAN-1SC	1437	23

setting. Specifically, using the dynamic programming approach described by György *et al.* (2006), we can maintain the desired distribution over schedules (i.e., paths) implicitly by maintaining a weight for each edge in our graph, and the time required for an exploitation step is $O(|E|)$. To converge to an optimal schedule we must set $|E| = k(B+1)^k$, while to converge to an α-approximation we may set $|E| = k(\lceil \log_\alpha B \rceil + 1)^k$, as per the discussion leading up to Theorem 2. By using a "lazy" implementation of the exploitation steps, we can reduce the total decision-making time to $O(m|E|)$.

Unfortunately, the decision-making time required by the label-efficient forecaster is still exponential in k. As future work, we are developing a strategy based on the greedy approximation algorithm that makes decisions in time $\mathrm{poly}(n, k)$ and whose performance in the adversarial online setting asymptotically converges to that of the offline greedy algorithm.

Experimental Evaluation

Each year, various conferences hold solver competitions designed to assess the state of the art in some problem domain. In these competitions, each submitted solver is run on a sequence of problem instances, subject to some per-instance time limit. Solvers are awarded points based on the instances they solve, and prizes are awarded to the highest-scoring solvers. Most competitions award separate prizes for different categories of instances.

In this section we describe experiments performed using data from solver competitions held at the following four conferences: SAT 2005 (Boolean satisfiability), ICAPS 2006 (planning), CP 2006 (constraint solving), and IJCAR 2006 (theorem proving). For each competition, we constructed the table τ of solver completion times using data from the competition web site (we did not actually run any of the solvers). We present detailed results for the satisfiability and planning competitions, and summarize the results for the other competitions.

Results for the ICAPS 2006 planning competition

Six optimal planners[3] were entered in the ICAPS 2006 competition. Each was run on 240 instances, with a time limit of 30 minutes per instance. On 110 of the instances, at least one of the six planners was able to find a (provably) optimal plan. We used the greedy algorithm to construct an approximately optimal task-switching schedule, given as input the completion times of each of the six planners on each of these 110 instances. To address the possibility of overfitting, we repeated our experiments using leave-one-out cross-validation.[4] We also evaluated a modified greedy algorithm (described in Remark 1) that throws away all its work when switching between solvers (when executing the modified greedy schedule, only a single run needs to be kept in memory at a time).

Table 1 presents the results. As shown in the table, the greedy schedules outperform the naïve parallel schedule (which simply runs all six planners in parallel) as well as each of the six individual planners, both in terms of the lower bound on average CPU time and in terms of the number of instances solved within the 30 minute time limit. The lower bound was obtained by capping the CPU time for each instance at the timeout value (we do not know the time a solver truly takes on instances it did not solve within the 30 minute time limit). Figure 2 shows the task-switching schedule constructed by the offline greedy algorithm.

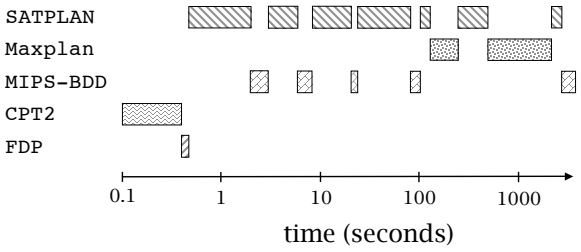

Figure 2: Greedy task-switching schedule for interleaving solvers from the ICAPS 2006 optimal planning competition (the solver IPPLAN-1SC is not shown because it did not appear in the schedule).

Results for the SAT 2005 competition

We performed an experiment similar to the one just described using the top two solvers from the random instance category of the SAT 2005 competition. After removing instances that neither of the top two solvers could solve, the number of remaining instances was 284. Table 2 summarizes our results. In this experiment it was practical to compute an optimal schedule using the shortest paths algorithm

[3] An optimal planner is one that finds a plan of provably minimum makespan.

[4] Leave-one-out cross-validation is performed as follows: for each instance, we remove that instance from the table and run the greedy algorithm on the remaining data to obtain a schedule to use in solving that instance.

Table 2: Results for the SAT 2005 competition, random category (cross-validation results are parenthesized).

Solver	Avg. CPU (s)	Num. solved
Optimal schedule	1173	261
Greedy schedule (x-val)	1173 (1259)	261 (260)
Parallel schedule	1325	257
`ranov`	2026	209
`kcnfs-2004`	2874	167

from Theorem 2. To our surprise, the optimal schedule's average CPU time was only about 0.1% better than that of the greedy schedule. The greedy schedule improves on the average CPU time of the faster of the two original solvers by a factor of 1.7 and solves significantly more instances within the 100 minute time limit. With only two available heuristics, the improvement relative to the naïve parallel schedule is smaller but still significant. We performed similar experiments using the top two solvers in the hand-crafted category, where we obtained a factor of 2 improvement in average CPU time relative to the faster of the two solvers. In the industrial category we obtained a factor of 1.2 improvement.

Summary

Table 3 summarizes our results on the four solver competitions we considered. The "speedup factor" listed in the second column is the ratio of the average CPU time for the best individual solver to that of the greedy algorithm (e.g., for the planning competition the speedup factor is $\frac{507}{358} \approx 1.4$). For each competition, we list the range of speedup factors we observed within each instance category (e.g., the SAT competition had separate categories for industrial, random, and hand-crafted instances).

Table 3: Summary of results for four solver competitions.

Solver competition	Speedup factor (range across categories)
SAT 2005	1.2–2.0
ICAPS 2006	1.4
CP 2006	1.0–2.0
IJCAR 2006	1.0–7.7

The Generalization to Restart Schedules

If $\mathcal{H}$ contains randomized heuristics, it may help to periodically restart them with a fresh random seed. In this setting, we outline how our results on task-switching schedules can be extended to a more powerful class of schedules which we call *restart schedules*.

Definition (restart schedule). *A* restart schedule $S : \mathbb{Z}^+ \rightarrow \mathcal{H} \times \{0, 1\}$ *specifies, for each integer* $t \geq 0$*, a pair* $S(t) = (h, r)$*; where* h *is the heuristic to run from time* t *to time* $t + 1$ *and* r *is a flag that, if set to 1, specifies that* h *should be restarted with a fresh random seed (at time* $t + 1$*).*

Our most exciting result is that a generalization of the greedy approximation algorithm for task-switching schedules achieves the optimal approximation ratio of 4 for the problem of computing restart schedules offline (by Theorem 1, no polynomial-time algorithm can achieve a better approximation ratio unless $P = NP$). Informally, we accomplish this generalization by changing the rule our algorithm follows to "greedily maximize the *expected* number of instances solved per unit time".

The remainder of our results build on the following fact: a restart schedule for a single randomized heuristic h can be represented as a task-switching schedule for the set of heuristics $\{h_1, h_2, \ldots, h_B\}$, where h_i runs h with restart threshold i. Using this observation plus some additional facts, we can prove analogues of Theorems 2 and 5 for restart schedules. In a companion paper (Streeter, Golovin, & Smith 2007), we present these results in more detail and evaluate them experimentally.

Conclusions

A task-switching schedule specifies how to interleave the execution of a set of deterministic heuristics, with the aim of improving the mean running time. We described how to select task-switching schedules in offline, learning-theoretic, and adversarial online settings. Experimentally, we showed that task-switching schedules have the potential to improve the performance of state-of-the-art solvers in several problem domains. We outlined how to extend our results to randomized heuristics, in which case we allow the heuristics to be periodically restarted with a fresh random seed.

Acknowledgment. This work was supported in part by NSF ITR grants CCR-0122581 and IIS-0121678 and by DARPA under Contract #FA8750-05-C-0033.

References

Cesa-Bianchi, N.; Lugosi, G.; and Stoltz, G. 2005. Minimizing regret with label efficient prediction. *IEEE Transactions on Information Theory* 51:2152–2162.

Feige, U.; Lovász, L.; and Tetali, P. 2004. Approximating min sum set cover. *Algorithmica* 40(4):219–234.

Gomes, C. P., and Selman, B. 2001. Algorithm portfolios. *Artificial Intelligence* 126:43–62.

Gomes, C.; Selman, B.; Crato, N.; and Kautz, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfications problems. *Journal of Automated Reasoning* 24(1/2):67–100.

György, A., and Ottucsák, G. 2006. Adaptive routing using expert advice. *The Computer Journal* 49(2):180–189.

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

Sayag, T.; Fine, S.; and Mansour, Y. 2006. Combining multiple heuristics. In *Proceedings of the* 23^{rd} *International*

Symposium on Theoretical Aspects of Computer Science, 242–253.

Streeter, M.; Golovin, D.; and Smith, S. F. 2007. Restart schedules for ensembles of problem instances. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*.

Appendix A: Proofs

Theorem 3. *The greedy algorithm runs in time $O(nk \log n \cdot \min\{n, Bk\})$ and returns a schedule S^G that is a 4-approximation to the optimal task-switching schedule.*

Proof. Let $GREEDY = \sum_{i=1}^{n} c_i(S^G)$, and let $OPT = \sum_{i=1}^{n} c_i(S^*)$, where S^* is an optimal task-switching schedule. Our goal is to show that $GREEDY \leq 4 \cdot OPT$.

On the z^{th} epoch, the greedy algorithm runs some heuristic h_j for an additional Δ_z time steps. At that point in the schedule, only the instances in R_z remain unsolved, so these Δ_z time steps contribute at most $|R_z|\Delta_z$ to $GREEDY$. Thus

$$GREEDY \leq \sum_{z \geq 1} |R_z| \Delta_z \, . \tag{2}$$

Let $Q_t = \{i : c_i(S^*) > t\}$ be the set of instances that remain unsolved by S^* at time t. Then

$$OPT = \sum_{t \geq 0} |Q_t| \, . \tag{3}$$

The following claim states the key property of the greedy algorithm.

Claim 1. For any epoch z and time t, $|Q_t| \geq |R_z| - ts_z$, where s_z is the maximum slope (i.e., the maximum value of $\frac{|X_j^z(t_j + \Delta_z)|}{\Delta_z}$) from the z^{th} epoch.

Proof (of Claim 1). If $|Q_t| < |R_z| - ts_z$ then, at time t, the schedule S^* has solved more than s_z instances from R_z per unit time. There must therefore be some heuristic h_j that, at some time $t' \leq t$, has solved more than s_z instances from R_z per unit time, so $\frac{|X_j^z(t')|}{t'} > s_z$. It must be that $t' > t_j$; otherwise $X_j^z(t') = 0$ (by definition, $X_j^z(t')$ is a subset of R_z, which only contains instances that remain *unsolved* after running h_j for t_j time steps). But by definition of s_z, for any heuristic h_j and time $t' \geq t_j$ we have $s_z \geq \frac{|X_j^z(t')|}{t' - t_j} \geq \frac{|X_j^z(t')|}{t'}$, a contradiction. □

We are now ready to prove that $GREEDY \leq 4 \cdot OPT$. Let T_z be the smallest integer t such that $|Q_t| \leq \frac{1}{2}|R_z|$. As illustrated in Figure 3,

$$OPT = \sum_{t \geq 0} |Q_t| \geq \sum_{z \geq 1} T_z \frac{|R_z| - |R_{z+1}|}{2} \, . \tag{4}$$

By Claim 1, $T_z \geq \frac{|R_z|}{2s_z}$. By inspection of the greedy algorithm, $|R_z| - |R_{z+1}| = s_z \Delta_z$. Thus using (4) we have $OPT \geq \sum_{z \geq 1} \frac{|R_z|}{2s_z} \frac{s_z \Delta_z}{2} \geq \frac{1}{4} GREEDY$, where the second inequality follows from (2).

We now analyze the running time. Because T increases by one each epoch and $|R_z|$ decreases by one each epoch, the number of epochs is at most $\min\{Bk, n\}$. By exploiting the fact that the pair (h_j, Δ_z) selected on the z^{th} epoch must have $t_j + \Delta_z = \tau_{i,j}$ for some i, each epoch can be implemented in $O(nk \log n)$ time.

□

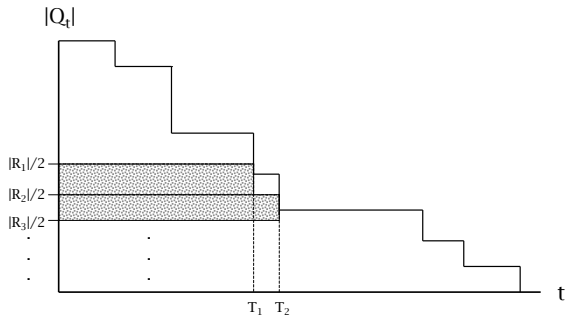

Figure 3: A proof of inequality (4). OPT equals the area under the curve, whereas $\sum_{z \geq 1} T_z \frac{|R_z| - |R_{z+1}|}{2}$ equals the sum of the areas of the shaded rectangles.

Theorem 4. *Let $C(S)$ and $\bar{C}(S)$ be defined as above. For $m \geq m_0(\epsilon, \delta) = \frac{1}{2}\left(\frac{c^*}{\epsilon}\right)^2 \ln\left(\frac{2|V|}{\delta}\right)$, the inequality*

$$\max_{S \in \mathcal{S}_{ts}} \left|\bar{C}(S) - C(S)\right| \leq \epsilon$$

holds with probability at least $1 - \delta$, where $|V| = (B+1)^k$.

Proof. As illustrated in Figure 1, each task-switching schedule corresponds to a path in a vertex-weighted graph. For any vertex v and training instance x_i, let $w_i(v)$ be the weight assigned to v on behalf of x_i (so $w_i(v) \in \{0, 1\}$); let $\bar{W}(v) = \frac{1}{m}\sum_{i=1}^{m} w_i(v)$ be the sample mean weight assigned to v, and let $W(v)$ be the true expected value. For any schedule S, let V_S be the set of vertices in the corresponding path. Then $\bar{C}(S) = \sum_{v \in V_S} \bar{W}(v)$ by construction while $C(S) = \sum_{v \in V_S} W(v)$ by linearity of expectation. Each schedule S corresponds to a path that passes through at most c^* vertices of non-zero weight. Thus, $\max_{v \in V} |\bar{W}(v) - W(v)| \leq \frac{\epsilon}{c^*}$ implies $\max_{S \in \mathcal{S}_{ts}} \left|\bar{C}(S) - C(S)\right| \leq \epsilon$.

For any vertex $v \in V$ the sample mean $\bar{W}(v)$ is the average of m independent identically distributed random variables, each of which has range $[0, 1]$. Thus by Hoeffding's inequality,

$$\mathbb{P}\left[|\bar{W}(v) - W(v)| \geq \frac{\epsilon}{c^*}\right] \leq 2\exp\left(-2m\left(\frac{\epsilon}{c^*}\right)^2\right) \, .$$

Setting the right hand side equal to $\frac{\delta}{|V|}$, solving for m, and using the union bound proves the theorem. □