Online Selection, Adaptation, and Hybridization of Algorithms

(Thesis proposal)

Matthew Streeter

November 19, 2006

Abstract

We propose to develop black-box techniques for improving the performance of algorithms by adapting them to the sequence of problem instances on which they are run. In particular, we propose to develop effective strategies for solving the following four problems:

1. *Combining multiple heuristics online.* Given k (randomized) algorithms, we seek to learn online a policy for interleaving and restarting them in order to solve a sequence of instances as quickly as possible.
2. *Online reduction from optimization to decision.* Given a sequence of instances of an optimization problem and an oracle for the corresponding decision problem, we seek to find a provably (approximately) optimal solution to each instance while minimizing the CPU time consumed by all oracle calls.
3. *Online tuning of branch and bound algorithms.* Given a branch and bound algorithm with a number of parameters (e.g., choices of relaxations, constraint propagators), we seek to determine near-optimal parameter settings online.
4. *The max k-armed bandit problem*. In this variant of the classical k-armed bandit problem one seeks to maximize the highest payoff received on any single pull, rather than the total payoff. A strategy for solving the max k-armed bandit problem can be used to allocate trials among multi-start optimization heuristics.

For each problem, we prove the existence of a no-regret strategy. We show experimentally that our strategy for combining multiple heuristics online has the potential to improve the performance of state-of-the-art SAT solvers and A.I. planners, and that our max k-armed bandit strategy effectively allocates trials among multi-start heuristics for the resource-constrained project scheduling problem.

This proposal is based on part on joint work with Stephen Smith [36, 37] and Daniel Golovin.

1 Introduction

In practice, a single piece of code is often run on a large sequence of problem instances, with no information retained from one instance to the next. For example, an operating system's sorting routine may be invoked on thousands or millions of lists over the lifetime of a system. The total CPU time required to sort the lists depends on (a) the compiled code (i.e., choice of sorting algorithm, pivoting rules, and various implementation and compilation details), (b) the sequence of lists, and (c) various details of the hardware and software environment. Because (b) and (c) are unknown in advance, compiled code is judged according to available theory (i.e., worst-case and average-case analysis) as well as performance on benchmark instances intended to be representative of the real world.

The premise of this thesis is that the performance of algorithms can be improved by adapting them online to the sequence of instances on which they are run, and furthermore that this can be accomplished using *black box* strategies (i.e., strategies that observe an algorithm's external behavior but not its internal state). These strategies may select among algorithms, tune the parameters or components of an individual algorithm, or compose multiple algorithms into a hybrid. We refer to the high-level problem such strategies solve as *online algorithm adaptation*.

Our goal is to identify and solve special cases of the online algorithm adaptation problem that are *tractable* (i.e., we can prove the existence of a no-regret strategy) and *practical* (i.e., we can demonstrate performance gains on real-world sequences of problem instances). To focus our efforts, we formulate four problems: *combining multiple heuristics online*, *online reduction from optimization to decision*, *online tuning of branch and bound algorithms*, and *the max k-armed bandit problem*. For each problem, we seek to develop a strategy with good theoretical guarantees (i.e., average regret approaching zero as the number of problem instances approaches infinity) as well as good performance in practice on instance sequences of reasonable length. Each of these problems is designed to have the property that an effective strategy could be used (or has already been used) to improve state-of-the-art algorithms in some domain.

1.1 Problem catalog

1.1.1 Combining multiple heuristics online

Suppose we are given a set $\mathcal{H}$ of heuristics to use in solving a sequence of instances of some decision problem. Each heuristic is able to compute a solution to any problem instance, but the heuristic's running time varies from instance to instance. We solve each instance by interleaving the execution of the heuristics according to a *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 $t + 1$.

In the online version of the problem, we are fed problem instances one at a time and must (adaptively) select a task-switching schedule to use in solving each instance. Building on existing results in online learning, we give a no-regret strategy for selecting task-switching schedules so as to minimize total CPU time.

As an extension, we allow $\mathcal{H}$ to contain randomized heuristics, and allow for schedules that periodically restart each heuristic using a fresh random seed. In the special case when $|\mathcal{H}| = 1$, our results give a no-regret strategy for the widely-studied problem of selecting restart schedules for a single Las Vegas algorithm.

1.1.2 Online reduction from optimization to decision

Suppose we wish to solve a sequence of optimization problems, each of which entails minimizing a cost function with range $\mathcal{B} \equiv \{1, 2, 3, \ldots, B\}$. Each problem is to be solved by making a series of calls to an oracle that answers queries of the form "Does there exist a solution with cost $\leq q$?", and the goal is to minimize the total CPU time consumed by all oracle calls.

A *reduction policy* can be represented as a binary search tree with key set $\mathcal{B}$. Internal nodes of the tree specify the next query point to submit to the oracle, and leaf nodes specify optimal values of the cost function. We give a no-regret strategy for selecting reduction policies in an online setting.

An effective strategy for selecting reduction policies can potentially be used to boost the performance of optimal planners such as SATPLAN, which find a minimum-length solution to a planning problem by solving a sequence of boolean satisfiability problems (i.e., using a SAT solver as a decision oracle). Such a strategy can also be used to discover deepening policies for A^*-style tree search.

1.1.3 Online tuning of branch and bound algorithms

Branch and bound algorithms are ubiquitous in operations research and constraint programming. The performance a of branch and bound algorithm depends critically on the choice of a number of components of the algorithm, including variable selection heuristics, value ordering heuristics, relaxations, and constraint propagators. When running a branch and bound algorithm on a sequence of problem instances, we would like to adaptively select choices of components that allow us to solve the sequence of instances as quickly as possible. We allow for different components to be selected for use in different levels of the branch and bound search tree.

We give a no-regret strategy for selecting relaxation (constraint propogation) policies in an online setting, in which we are fed an arbitrary sequence of optimization problems and must adaptively select a policy for deciding which relaxation(s) (constraint propagator(s)) to run at each level of the search tree.

1.1.4 The max k-armed bandit problem

In the max k-armed bandit problem, one is faced with a set of k slot machines, each having an arm that, when pulled, yields a payoff drawn from a fixed (but unknown) distribution. The goal is to allocate trials to the arms so as to maximize the maximum payoff obtained over a series of n trials.

Our motivation for studying the problem is to boost the performance of *multi-start* heuristics, which obtain a solution to an optimization problem by performing a number of independent runs of a randomized heuristic and returning the best solution obtained. Multi-start heuristics are widely used in practice, and represent the state of the art in a number of domains [6, 10, 15]. A max k-armed bandit strategy can be used to distribute trials among multi-start heuristics or to tune the parameters of a single multi-start heuristic. Previous work has demonstrated the effectiveness of such an approach on the RCPSP/max, a difficult real-world scheduling problem [9, 11, 37].

1.2 Summary of results

Our results for the max k-armed bandit problem are based on two conference papers [36, 37]. We first define a natural notion of regret and show that a no-regret strategy does not exist in the case of arbitrary payoff distributions over a bounded interval. We then define a strategy called Threshold Ascent which is designed to work well when the payoff distributions have certain characteristics which we expect to be present in cases of practical interest. We demonstrate the effectiveness of Threshold Ascent by using it to select among multi-start heuristics for a difficult real-world scheduling problem. Motivated by a result from extreme value theory, we then prove the existence of a no-regret strategy for the special case in which payoffs are drawn from *generalized extreme value* distributions.

For the remaining problems, our results are more preliminary. For the problems of selecting task-switching schedules, restart schedules, relaxation policies, and constraint propagation policies, we prove the existence of a no-regret strategy by reducing the problem to *online shortest paths* [2, 19]. For the problem of selecting reduction policies, we develop a no-regret strategy based on the widely-used multiplicative weight update method.

1.3 Proposed work

We discuss specific extensions to our work on the max k-armed bandit problem in §6.6. We are interested in developing additional applications of the max k-armed bandit problem, in integrating the parametric and non-parametric approaches to the problem that we have developed, and in extending our results to an online setting in which a sequence of max k-armed bandit problems must be solved.

For the remaining online problems, our stated goal is to develop *effective* strategies. Thus, in addition to having worst-case average regret that tends to zero as the number of problem instances approaches infinity, we would like our strategies to show good performance in practice on real-world instance sequences of reasonable size. In some cases we may be able to obtain improved regret bounds (and presumably improved performance in practice) by exploiting the structure of the problem. In other cases we may be able to discover ad-hoc techniques that improve performance on instance sequences encountered in practice without affecting worst-case performance. In §7 we identify other potential extensions to this work.

2 Background and Terminology

This section introduces terminology and concepts that will be used in the discussion of the three problems presented in §§3-5.

Suppose we are given a sequence $\mathcal{I} = \langle I_1, I_2, \dots I_n \rangle$ of problem instances (e.g., a sequence of lists to sort, a sequence of integer programs to solve). To solve each instance I_j, we must choose a *policy* π from a set Π of possible policies (e.g., a set of pivoting rules to use in a sorting algorithm, a set of branching rules to use in an ILP solver). Running policy π on instance I_j returns an observation $o(\pi, I_j)$ and incurs a cost $c(\pi, I_j)$. Our goal is to (adaptively) select a sequence of policies $\pi_1, \pi_2, \dots, \pi_T$ such that the total cost $\sum_{j=1}^{n} c(\pi_j, I_j)$ incurred over all instances is as small as possible.

A *strategy* $\mathcal{S}$ is a function that takes as input (a) the history

$$h_{j-1}^{\mathcal{S}} = \langle (\pi_1, o_1), (\pi_2, o_2), \dots, (\pi_{j-1}, o_{j-1}) \rangle$$

of policies and observations resulting from using the strategy $\mathcal{S}$ on the first $j-1$ instances and (b) an infinite sequence of random bits, and returns as output a policy $\mathcal{S}(h_{j-1}^{\mathcal{S}}) \in \Pi$ to use to solve the next instance I_j.

Consider some fixed strategy $\mathcal{S}$ and sequence $\mathcal{I}$, and let c_j denote the cost that $\mathcal{S}$ incurs on the j^{th}

instance. The *regret* of $\mathcal{S}$ on instance sequence $\mathcal{I}$ is equal to

$$\frac{1}{n}\left(\mathbb{E}\left[\sum_{j=1}^{n} c_j\right] - \min_{\pi\in\Pi}\sum_{j=1}^{n} c(\pi, I_j)\right) \tag{1}$$

where the expectation is over the random bits used by $\mathcal{S}$. In other words, regret measures the difference between the (expected) cost incurred by following strategy $\mathcal{S}$ and the cost incurred by the minimum-cost policy *for the instance sequence* $\mathcal{I}$. The worst-case regret of a strategy is the maximum value of (1) over all instance sequences of length n. A *no-regret* strategy is one for the worst-case regret is $o(1)$ as a function of n. A no-regret strategy has the property that, as n gets large, the performance of the strategy will be almost as good as that of the best fixed policy even if the instance sequence is chosen by an adversary who knows the strategy.

2.1 Online shortest paths

Several of our no-regret strategies are based on reductions to the online shortest paths problem. The online shortest paths problem can be described as the following adversarial game. Given is a graph $G = \langle V, E\rangle$, and two distinguished vertices $s, t \in V$. At the beginning of the game, an adversary selects edge weights $w_j : E \to [0, 1]$ for each round j, $1 \le j \le n$. At the beginning of round j, the player selects a path π_j from s to t and then incurs cost $\sum_{e\in\pi_j} w_j(e)$. In the variant of the problem we will consider, the player then observes the value of $w_j(e)$ for each $e \in \pi_j$ but does not observe the weights assigned to any other edges.

In terms of the framework just described, the problem instances I_j correspond to the weights w_j; the policy space Π is the set of paths from s to t; the cost $c(\pi_j, I_j)$ is the weight assigned to the path π_j; and the observation o_j is a list of weights for each of the edges in π_j.

3 Combining Multiple Heuristics Online

In many cases of practical interest, different algorithms have complementary strengths and weaknesses. For example, in the domain of boolean satisfiability, local search algorithms such as WalkSAT [35] are able to quickly find solutions to satisfiable formulae but are ineffective on unsatisfiable formulae, while chronological backtracking algorithms such as zChaff [26] are able to prove that a formula is unsatisfiable but are (often) relatively slow at discovering solutions to satisfiable formulae. In such domains, good performance has been achieved by using *algorithm portfolios*. An algorithm portfolio is a policy for parallelizing and restarting a set of k (randomized) algorithms, with the aim of solving problem instance(s) more quickly than would be possible using any of the k algorithms in isolation [18, 20].

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_k$ be a set of available heuristics for use in solving a sequence of instances of some decision problem. We first consider the case in which all heuristics in $\mathcal{H}$ are deterministic. In this case, we solve each instance by interleaving the execution of the heuristics according to a *task-switching schedule*. A task-switching schedule $S : \mathbb{Z}^+ \to \mathcal{H}$ specifies, for each integer $t \ge 0$, the heuristic $S(t)$ to run from time t to time $t + 1$.

As a generalization, we allow the heuristics in $\mathcal{H}$ to be randomized, and allow for schedules that periodically restart each heuristic with a fresh random seed. A *restart schedule* $S : \mathbb{Z}^+ \to \mathcal{H} \times \{0, 1\}$ specifies, for each time $t \ge 0$, a pair $S(t) = (h, r)$; if $r = 0$ then the run of h is continued for one additional time step (as in a task-switching schedule), while if $r = 1$, h is restarted with a fresh random seed and then run for one time step.

We propose to develop practical strategies for selecting task-switching (restart) schedules in an online setting. As a first step, we prove the existence of a no-regret strategy by reducing each of the two problems to *online shortest paths* (see 2.1).

3.1 Related work

The term *algorithm portfolio* was introduced by Huberman et al. [20], who found that performing two parallel runs of a chronological backtracking algorithm for graph coloring yielded lower expected solution time than performing a single run. The algorithm portfolios considered in previous work can be represented as tuples of the form $\langle \mathcal{P}, t \rangle$, where $\mathcal{P} \subseteq \mathcal{H}$ is the set of algorithms to run in parallel (each at equal strength), and $t : \mathcal{P} \to \mathbb{Z}$ is a set of *restart thresholds* (i.e., each heuristic $h_i \in \mathcal{P}$ is to be restart every $t(h_i)$ time steps). Gomes et al. [18] have addressed the problem of constructing an optimal algorithm portfolio 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 the case where all heuristics in $\mathcal{H}$ are deterministic (so that restart thresholds play no role), task-switching schedules generalize the algorithm portfolios considered in previous work in that they do not require each algorithm being run to receive an equal proportion of CPU time and do not require the proportion of CPU time allocated to each algorithm to be constant over time. In the case where $\mathcal{H}$ contains randomized heuristics, restart schedules allow for a non-constant sequence of restart thresholds, which can be beneficial in the case where different instances have different run length distributions. Thus, in addition to learning policies in a more challenging setting (online vs. offline), we consider a richer policy space than has been considered in previous work.

3.2 Selecting task-switching schedules online

We now introduce some additional notation. For any heuristic h and instance I, let $\tau(h, I)$ denote the time that h must run before returning a solution to I. We assume $\tau(h, I) \in \{1, 2, \ldots, B\}$ for some known B. Abusing notation, for any task-switching schedule S and instance I we denote by $\tau(S, I)$ the time required to solve instance I using task-switching schedule S (i.e., $\tau(S, I)$ is the smallest t such that for some h, S has run h for $\tau(h, I)$ time units during the interval $[0, t]$).

3.2.1 Solving the offline problem

We first consider the offline problem of computing an optimal task-switching schedule for the instance sequence $I_1, I_2, \ldots, I_n$ (i.e., the schedule S that minimizes $\sum_{j=1}^{n} \tau(S, I_j)$), given knowledge of $\tau(h_i, I_j)$ for all i and j. 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 I_j, we can define vertex weights $w_j : V \to \{0, 1\}$ such that for any task-switching schedule S, $\tau(S, I_j)$ equals the weight assigned by w_j to the path corresponding to S.

The vertices our graph will be arranged in a k-dimensional grid with sides of length $B + 1$. The vertex with grid coordinates $\langle t_1, t_2, \ldots, t_k \rangle$ will correspond to the state of having run each heuristic h_i for t_i 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_j are defined as follows: $w_j(v) = 1$ if there is a path from vertex v to the vertex with grid coordinates $\langle \tau(h_1, I_j) -$

$1, \tau(h_2, I_j)-1, \ldots, \tau(h_k, I_j)-1\rangle$; otherwise $w_j(v) = 0$. In other words, w_j assigns weight 1 to vertices that correspond to states in which instance I_j has not yet been solved, and assigns weight 0 to all other vertices. It is easy to check that $\tau(S, I_j)$ is equal to the weight assigned by w_j to the path corresponding to S.

It follows that the optimal task-switching schedule for instances $I_1, I_2, \ldots, I_n$ is the schedule that corresponds to the shortest path in G under the weight function $w = \sum_{j=1}^{n} w_j$. Figure 1 gives an example set of completion times and the corresponding shortest paths problem.

Completion times

	h_1	h_2
I_1	2	1
I_2	3	1
I_3	2	4
I_4	3	4

Shortest path problem

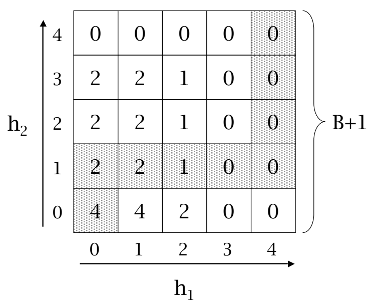

Figure 1: An example set of completion times and the corresponding shortest path problem.

The time required to find an optimal task-switching schedule using this algorithm is $O(nB^k)$. 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 $(\log_\alpha B) + 1$, and the time complexity is $O(n(\log_\alpha B)^k)$.

The time complexity of the algorithms discussed so far is exponential in k. In §3.3 we discuss a polynomial-time approximation algorithm that can be used as an alternative when k is large.

3.2.2 Solving the online problem

As shown in the previous section, the problem of selecting task-switching schedules online corresponds to an online shortest paths problem in a k-dimensional grid with sides of length L (where $L = B + 1$ if we are looking for an exact solution and $L = (\log_\alpha B) + 1$ if we are looking for an α-approximation). The feedback received in this particular online shortest path problem is somewhat unusual, however, due to the special structure of the weight functions. To see this, suppose that after selecting some schedule (path) we solve the instance in t time units. If v is the t^{th} vertex on the selected path then it must be that all predecessors of v have cost 1 while all successors of v have cost 0. In particular, note that the weights assigned to the vertices on our selected path are always revealed to us, and that the exact set of vertex weights that are revealed to us depends on both the schedule we select and the completion times of each heuristic.

György et al. considered the variant of the online shortest path problem discussed in 2.1, in which at the end of each round the player learns the weights assigned to each edge in the player's selected path. Adapting Theorem 1 of their paper to our setting gives an $O\left(Bk\sqrt{\frac{k(\ln k)L^k}{n}}\right)$ regret bound. This bound is exponential in k. To improve the dependence on k, we exploit the fact that it is possible to reveal the weights assigned to *all* vertices simply by solving each problem instance with each heuristic $h \in \mathcal{H}$ (at a cost of at most Bk).

Such a setting has been considered by Cesa-Bianchi et al. [8] under the heading of “label efficient” prediction. Adapting Theorem 1 of their paper to our setting yields an $O\left(Bk(\frac{Lk\ln k}{n})^{\frac{1}{3}}\right)$ regret bound. Given n and k, one may use whichever strategy has the lower regret bound. It is also possible to hybridize the two strategies to obtain regret bounds that improve slightly over the minimum of the two bounds just stated.

3.2.3 Preliminary results

In this section, we present experiments designed to demonstrate the power of combining algorithms using a task-switching schedule, as opposed to running them individually or in parallel. Toward this end, we will solve the *offline* problem of determining an optimal task-switching schedule given knowledge of the amount of time each algorithm requires on each instance. Of course, this offline solution is not of practical use by itself. However, our hope is that a no-regret algorithm will perform almost as well as the best fixed offline schedule for instance sequences of reasonable size.

Every year in the annual SAT competition, each submitted SAT solver is run a sequence of boolean formulae. Each solver is allowed 20 minutes per formula, and the goal is to solve as many formulae as possible. Prizes are awarded to the solvers that perform best on industrial, random, and hand-crafted formulae.

For each of the three instance categories, we computed an optimal task-switching schedules for solvers that won first prize in the satisfiable and unsatisfiable subsets.[1] Within each category, we filtered out instances that neither of the solvers could solve. If one of the two solvers did not solve an instance, we artificially set its completion time to the timeout value of 20 minutes.

Table 1 displays the CPU time and number of instances solved within the 20 minute time limit for the two solvers in each category, a schedule that simply runs these solvers in parallel, and the optimal task-switching schedule. In all three categories, the optimal task-switching schedule solves more instances within the 20 minute time limit than either of the original solvers or the naïve parallel schedule. In the random and hand-crafted instance categories, the optimal task-switching schedule outperforms either of the two individual solvers by a substantial margin, while in the industrial category the optimal task-switching schedule offers a slight improvement over the best single solver.

We performed similar experiments with prize-winning solvers in the random and industrial categories. For the random formulae, the policy of running the solvers *kcnfs-2004* and *ranov* in parallel performed substantially better than any of the individual solvers in isolation both in terms of CPU time and number of instances solved. For the industrial formulae, the optimal task-switching schedule performed only slightly better than the best individual solver.

3.3 Handling a large number of heuristics

The strategies discussed so far require decision-making time exponential in the number of heuristics k, and it is natural to wonder if some other approach not based on shortest paths could perform substantially better when k is large.[2]

To answer this question, we first consider the complexity of the offline version of the problem. In the special case where each $\tau(h_i, I_j)$ is either 0 or ∞, we may think of each heuristic h_i as covering the set of instances for which $\tau(h_i, I_j) = 0$, and the offline problem is equivalent to set covering under the “min sum”

[1] In the industrial category the solver $SatELiteGTI$ won the prize for both the satisfiable and unsatisfiable subsets, so we instead combined it with one of the second-place solvers.

[2] The results presented in this section are based on unpublished joint work with Daniel Golovin.

Table 1: Results of interleaving prize-winning solvers from the SAT 2005 competition.

Category (#Instances)	**Solver**	**Avg. CPU time (s)**	**Num. solved**
Hand-crafted (384)	Optimal schedule	115	367
	Parallel schedule	173	352
	Vallst	240	324
	SatELiteGTI	300	318
Random (238)	Optimal schedule	256	216
	Parallel schedule	307	214
	ranov	393	178
	kcnfs-2004	611	140
Industrial (423)	Optimal schedule	200	406
	SatELiteGTI	215	402
	MiniSAT 1.13	260	376
	Parallel schedule	312	380

objective function. This variant of set cover was introduced by Feige & Tetali [14], who showed that a simple greedy algorithm gives a 4-approximation and that obtaining a $4 - \epsilon$ approximation is NP-hard for any $\epsilon > 0$. It can be shown that a generalization of this greedy algorithm also gives a 4-approximation for computing task-switching and schedules. These observations are summarized in the following theorem.

Theorem 1. *A greedy algorithm obtains a 4-approximation to the optimal task-switching schedule; and for any $\epsilon > 0$, obtaining a $4 - \epsilon$ approximation to the optimal task-switching schedule is NP-hard.*

The fact that the offline problem is hard to approximate can be used to show that any strategy that makes decisions in time polynomial in the number of instances and heuristics cannot be a no-regret strategy (modulo complexity-theoretic assumptions). In spite of this negative result, we have found that the greedy algorithm performs very well in practice, and we suspect that it can be used as the basis of an online strategy that has low regret in practice.

3.3.1 Preliminary results

We evaluate our greedy algorithm by using to find task-switching schedules for interleaving runs of optimal planners entered in the 2006 International Planning Competition.

Briefly, *STRIPS planning* [32] is the task of finding a sequence of actions (a plan) that lead from an initial state to a goal state. Each state is represented by a conjunction of literals. Each action has certain preconditions (literals that must be present in the description of a state in order for it to be possible to apply the action at that state) and effects (literals that become true or false as a result of executing the action). *Optimal planning* is the task of finding a plan with a minimum number of actions.

Six optimal planners were entered in the 2006 International Planning Competition. Each planner 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. As in our experiments with SAT solvers, we evaluated our algorithm by using it to solve the offline problem of constructing an approximately optimal task-switching schedule, given as input the completion times of each of the six planners on each of 110 instances. Table 2 presents the results. As shown in the table, the greedy schedule outperforms the both the

Table 2: Performance of eight solvers on 110 problem instances from the 2006 International Planning Competition.

Solver	**Avg. CPU time (s)**	**Num. solved**
Greedy schedule	358	98
Satplan	507	83
Maxplan	641	88
Mips-BDD	946	54
CPT2	969	53
FDP	1079	46
Parallel schedule	1244	89
IPPLAN-1SC	1437	23

naïve parallel schedule and each of the six planners, both in terms of average CPU time and the number of instances solved within the 30 minute time limit. Figure 2 shows the task-switching schedule constructed by the greedy algorithm.

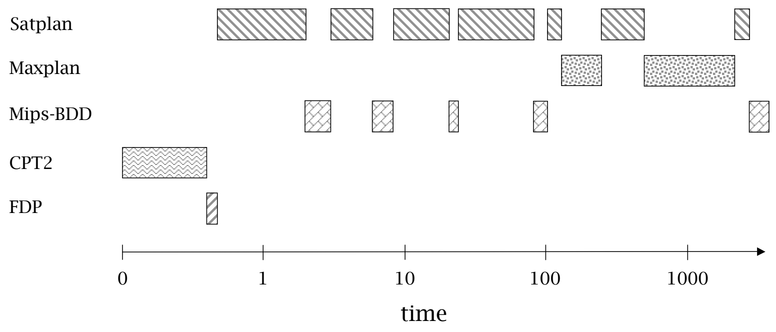

Figure 2: Greedy task-switching schedule for interleaving solvers from the 2006 International Planning Competition. The solver IPPLAN-1SC is not shown because it did not appear in the schedule.

3.4 Selecting restart schedules online

In this section we consider the problem of selecting restart schedules in the adversarial online setting discussed in §2. For simplicity of exposition, we will consider the case when $\mathcal{H}$ consists of a single randomized algorithm (call it $\mathcal{A}$). In this case a restart schedule can be represented as a sequence of thresholds

$$\langle t_1, t_2, \ldots \rangle$$

whose meaning is "run $\mathcal{A}$ for t_1 time steps; if it does not terminate then restart and run for t_2 time steps, ...".

Recall that in our worst-case model, the sequence of run length distributions we encounter is selected adversarially, and the only feedback we receive after solving instance I_j using restart policy π_j is the CPU time that was required. In particular, we never learn the actual run length distributions of any instance that we solve. It is perhaps surprising that despite this limited feedback, it is still possible to give a no-regret strategy.

It is worth noting that if $T = 1$ (or if the run length distribution of $\mathcal{A}$ is the same for all problem instances), the optimal restart policy is a *uniform* policy of the form $\langle t^*, t^*, \ldots, t^* \rangle$, for some appropriately chosen t^* [25], and thus we could restrict our search to policies of this form. If the run length distribution differs from instance to instance, however, one can construct examples where the best uniform policy is suboptimal by an arbitrarily large factor.

3.4.1 Related work

A 1998 paper by Gomes et al. [17] drew attention to a surprising attribute of popular chronological backtracking algorithms: when run on real-world problem instances, they often exhibit a *heavy-tailed* run length distribution, meaning that the probability that the run has failed to terminate after t time steps exhibits only a power-law decay. Simply restarting the algorithm after a fixed amount of CPU time has elapsed forces the tail of the run length distribution to decay exponentially, and can dramatically reduce the mean run length. Gomes et al. reported order-of-magnitude reductions in mean run length for a then state-of-the-art SAT solver [16, 17]. Subsequently, restart policies have received much attention in the A.I. community. Today, state-of-the-art SAT solvers such as MiniSat [13] make use of randomization and restart and perform poorly if restarts are turned off.

Previous work on restart policies has either sought to obtain fixed restart policies that work well in a wide variety of practical settings, or has assumed knowledge of the run length distribution(s) of the algorithm in question on each problem instance [30, 33], or has attempted to predict the future behavior of a Las Vegas algorithm based on observed variables that describe its execution state [22]. Our proposed work is novel in that, despite the fact that we treat the Las Vegas algorithm as a black box whose only observable behavior is its running time, we are able to guarantee good performance on a (sufficiently long) sequence of instances *even if the run length distributions are chosen adversarially*.

3.4.2 Reduction to selecting task-switching schedules online

Our strategy for selecting restart schedules can be expressed in a straightforward way in terms of our strategy for selecting task-switching schedules. Given some Las Vegas algorithm $\mathcal{A}$, let h_t be a heuristic that runs $\mathcal{A}$ with restart threshold t. It is easy to see that any restart schedule for $\mathcal{A}$ can be represented as a task switching schedule for the set of heuristics $\mathcal{H} = \{h_1, h_2, \ldots, h_B\}$, where B is a bound on the amount of time the restart schedule is allowed to run. Furthermore, the optimal task-switching schedule for the set $\mathcal{H}_\alpha = \{h_1, h_\alpha, h_{\alpha^2}, h_{\alpha^3}, \ldots, h_B\}$ is a α-approximation to the optimal restart schedule for $\mathcal{A}$. Combining this observation with our earlier α-approximation for computing task-switching schedules, we get that an α^2-approximation to the optimal task-switching schedule can be computed in time $O((\log_\alpha B)^{\log_\alpha B}) = O(B^{\log_\alpha \log_\alpha B})$.[3]

An an alternative to the approach based on online shortest paths, a generalization of the greedy algorithm discussed in Theorem 1 provides a 4-approximation to the optimum restart schedule, and could be used as the basis of an online strategy. We expect the approximation ratio of the greedy algorithm to be much better than 4 in practice.

[3]We do not expect this time complexity to be prohibitive for the values of B that will arise in our applications. Also, as shown in previous work (e.g., [19, 21]), we do not necessarily need to solve the shortest path problem on every trial.

4 Online Reduction from Optimization to Decision

Suppose we desire to solve a sequence $\mathcal{I} = \langle I_1, I_2, \ldots, I_n \rangle$ of problem instances, where each instance $I = (f, \mathcal{F})$ is the minimization problem

$$OPT(I) = \min_{x \in \mathcal{F}} f(x)$$

where f is the cost function to be minimized and $\mathcal{F}$ is the feasible set. We assume that each f has range $\mathcal{B} \equiv \{1, 2, \ldots, B\}$. Each instance is to be solved by making a sequence of calls to a decision oracle $\mathcal{O}$, where $\mathcal{O}(I, q)$ returns "yes" if there exists an $x \in \mathcal{F}$ such that $f(x) \leq q$ and returns "no" otherwise. Our goal is to find a *provably* optimal solution to each problem instance while minimizing the total CPU time consumed by all oracle calls.

A *reduction policy* can be represented as a binary search tree with key set $\mathcal{B}$. Internal nodes of the tree specify the next query point to submit to the oracle, and leaf nodes specify optimal values of the cost function. Figure 3 illustrates the *ramp-up*, *ramp-down*, and *binary search* policies for $B = 4$.

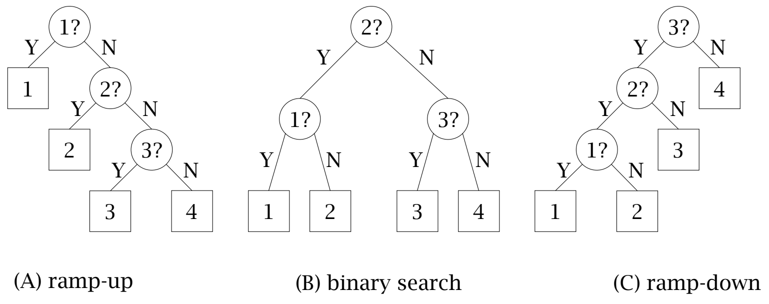

Figure 3: The ramp-up, binary search, and ramp-down policies (for $B = 4$).

Each oracle query $\mathcal{O}(I, q)$ requires CPU time $\tau(I, q)$. The cost $c(\pi, I)$ of solving instance I using policy π is the sum of the CPU times for each query. Note that, depending on the cost structure, an apparently naïve strategy such as ramp-up may outperform binary search. The problem of selecting reduction policies online generalizes the *tree update problem* [21], which corresponds to the special case when $\tau(I, q) = 1 \ \forall I, q$.

4.1 A no-regret strategy

To obtain a no-regret strategy for the problem of learning reduction policies, we first express the CPU time required by any tree on any particular instance as a linear function of an appropriate set of variables. Specifically, we represent each tree T by an indicator vector $x(T)$ of length B^2 with components

$$x(T)_{q,o} = \begin{cases} 1 & \text{if non-leaf node } q \text{ lies along the path from the root to leaf } o \text{ in } T \\ 0 & \text{otherwise} \end{cases}$$

for all $q, o \in \{1, 2, \ldots, B\}$. The cost vector c^j corresponding to the j^{th} instance has components

$$c^j_{q,o} = \begin{cases} \tau(I_j, q) & \text{if } OPT(I_j) = o \\ 0 & \text{otherwise.} \end{cases}$$

It is easy to check that the CPU time required by tree T to solve instance I_j is precisely $c^j \cdot x(T)$. The cost of any tree over the sequence of n instances is therefore a function of the B^2 components of the vector $\sum_{j=1}^{n} c_j$. Given the values of these B^2 components, the optimal tree can be computed in $O(B^3)$ time using dynamic programming.

To obtain a no-regret strategy, we use the *multiplicative weight update method*. Briefly, the idea of the multiplicative weight update method is that on round j, we select tree T with probability proportional to

$$\exp\left(-\alpha x(T) \cdot \sum_{i=1}^{j-1} \hat{c}^i\right)$$

where $\hat{c}^i$ is an unbiased estimate of c^i for $1 \leq i \leq j-1$, and α is a parameter. To obtain an intuition for why this method should quickly converge to the optimal tree, note that the quantity $x(T) \cdot \sum_{i=1}^{j-1} \hat{c}^i$ is an unbiased estimate of the CPU time that tree T *would have consumed* if run on the first $j-1$ instances, and that the probability assigned to T is decreasing exponentially as a function of this quantity. For a more detailed discussion of the multiplicative weight update method, see [1] and references therein.

Using dynamic programming, it is possible to sample from this distribution $O(B^3)$ time, and to compute the marginal probabilities required to maintain unbiased estimates. A analysis along the lines of the proof of Theorem 1 of [19] then shows that the regret is at most $B\sqrt{\frac{2B \ln B}{n}}$.

4.2 Finding approximately optimal solutions

To minimize the time required to find an *approximately* optimal solution, we simply work with appropriately pruned trees. For example if we require a 2-approximation, a subtree covering the leaves $\{3, 4, 5, 6\}$ can be replaced with a single leaf node labeled with the range "$[3, 6]$", and all of our results still hold. Both additive and multiplicative approximations can be obtained in this way.

4.3 Applications

4.3.1 Improving the performance of optimal planners

As mentioned in §3.3.1, STRIPS planning is the task of finding a sequence of actions (a plan) that lead from an initial state to a goal state, and optimal planning is the task of finding a plan with provably minimum length.

The current state-of-the-art optimal STRIPS planners find a minimum-length plan by constructing a sequence of boolean formulae and determining whether each of the formulae is satisfiable using a SAT solver. We denote the sequence of formulae by $\langle \sigma_{k_1}, \sigma_{k_2}, \ldots \rangle$. Formula σ_{k_i} is satisfiable if and only if a plan with k_i or fewer actions exists.

SATPLAN [23] finds a minimum-length plan using the ramp-up policy. That is, it solves the sequence of formulae $\langle \sigma_1, \sigma_2, \ldots \rangle$, stopping when it first encounters a satisfiable formula. In 2004, SATPLAN won first place for optimal deterministic planning at the International Planning Competition. In 2006, SATPLAN tied for first place with MaxPlan [38], another SAT-based planner. One of the key differences between MaxPlan and SATPLAN is that MaxPlan uses the ramp-down policy in place of ramp-up. It is our hope that by learning a reduction policy online, we can boost the performance of these planners.

4.3.2 Learning deepening policies for tree search

In *tree search*, one seeks to find a minimum-weight path from the root node of an edge-weighted tree to one of a set of "goal" nodes. *Iterative deepening* is a popular strategy for solving such problems. In the simplest case when each edge has a cost of 1, iterative deepening simply performs a series of depth-first searches with a depth limit of d for $d = 0, 1, 2, \ldots$ until a goal node is first discovered. In other words, iterative deepening employs the ramp-up policy. Given a sequence of tree search problems, one may be able to learn better deepening policies online.

5 Online Tuning of Branch and Bound Algorithms

Branch and bound algorithms are ubiquitous in operations research and constraint programming. The performance of a branch and bound algorithm is strongly influenced by the choice of each of a number of subroutines (e.g., variable selection heuristics, value ordering heuristics, relaxation(s), constraint propagator(s)). In general it is difficult to determine in advance which choices of subroutines will work well on a particular problem instance.

Given a set $\mathcal{R}$ of available relaxations, a *relaxation policy* $\pi : \mathbb{Z}^+ \to 2^{\mathcal{R}}$ specifies, for each level d of the search tree, the subset $\pi(d) \subseteq \mathcal{R}$ of relaxations that are to be run on tree nodes at level d. A *constraint propagation policy* is defined analogously. In this section we consider the problem of selecting relaxation (constraint propagation) policies in the online setting described in $\S 2$.

5.1 Background: branch and bound

Branch and bound algorithms are used to solve minimization problems of the form

$$z = \min_{x \in \mathcal{F}} f(x)$$

where $\mathcal{F} \subseteq \mathcal{D} = \times_{i=1}^{d} \mathcal{D}_i$ is the *feasible set* and $f : \mathcal{F} \to \mathbb{R}$ is the *cost function*. We denote the i^{th} component of x by x_i (so $x = \langle x_1, x_2, \ldots, x_d \rangle$, where $x_i \in \mathcal{D}_i \; \forall i$).

A branch and bound algorithm solves an optimization problem recursively by breaking the problem into subproblems (branching) and solving relaxed versions of each subproblem to quickly identify subproblems that can be ignored (bounding). More formally, *branching* breaks a problem with feasible set $\mathcal{F}$ into subproblems with feasible sets $\mathcal{F} \cap \mathcal{P}_1, \mathcal{F} \cap \mathcal{P}_2, \ldots, \mathcal{F} \cap \mathcal{P}_b$, where $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_b$ are disjoint subsets of $\mathcal{D}$ called *partial assignments*. Given a subproblem with feasible set $\mathcal{F} \cap \mathcal{P}$, *bounding* solves the minimization problem $\min_{x \in \mathcal{F}_{relax} \cap \mathcal{P}} f(x)$, where $\mathcal{F}_{relax} \supseteq \mathcal{F}$. Letting x^* denote the optimal solution to the relaxed subproblem, if it happens that $x^* \in \mathcal{F}$ then $f(x^*)$ provides an upper bound on z. If $f(x^*)$ is greater than or equal to a known upper bound on z, the subproblem can be ignored or *pruned*.

Pseudocode for a generic branch and bound algorithm is given below (the code does not include constraint propagation, which we discuss later). To determine z we initialize $\bar{z} \leftarrow \infty$ and then call BranchAndBound($\emptyset$). When the procedure terminates, $z = \bar{z}$.

```
Procedure **BranchAndBound**($\mathcal{P}$):

  1. $x^* \leftarrow relax(\mathcal{P})$. If $f(x^*) \geq \bar{z}$ then return; otherwise if $x^* \in \mathcal{F}$,
     set $\bar{z} \leftarrow \min\{\bar{z}, f(x^*)\}$ and return.
  2. $i \leftarrow variable\_select(\mathcal{P})$
  3. $\langle v_1, v_2, \ldots, v_{|\mathcal{D}_i|} \rangle \leftarrow value\_order(i, \mathcal{P})$
  4. For $j$ from 1 to $|\mathcal{D}_i|$:
       • Call BranchAndBound($\mathcal{P} \cap \{x \in \mathcal{D} : \ x_i = v_j\}$)
```

Example. An *0/1 integer linear program* is an optimization problem of the form

$$\min_{x \in \mathcal{F}} c \cdot x$$

where $\mathcal{F} = \{x \in \{0,1\}^d \mid Ax \leq b\}$ is the subset of $\{0,1\}^d$ that satisfies the system of linear inequalities $Ax \leq b$. Branch and bound algorithms for this problem use linear programming relaxations (i.e., $\mathcal{F}_{relax} = \{x \in [0,1]^d \mid Ax \leq b\}$).

5.2 Selecting relaxation policies online

It is not necessary to solve the relaxation at every node of the branch and bound search tree. In fact, line 1 in the pseudocode above could be replaced with the line "If $\mathcal{P}$ is a singleton, let x^* be the one element in $\mathcal{P}$ and, if $x^* \in \mathcal{F}$, set $\bar{z} \leftarrow \min\{\bar{z}, f(x^*)\}$". This approach could be called "solving the problem by branching". Although solving the problem by branching is extremely inefficient in general, it may be effective at highly-constrained nodes of the tree where solving the relaxation consumes more time than solving the problem by brute force.

In some cases, we may have more than one relaxation at our disposal. For example, both linear programming and semidefinite programming relaxations have been developed for the Max-SAT problem. Semidefinite relaxations provide a tighter bound, but are more expensive to compute. It may be best to run the the semidefinite relaxation near the root of the search tree, and switch to the linear programming relaxation closer to the leaves.

In this section we give a no-regret strategy for selecting relaxation policies online. We note that in many branch and bound algorithms, the variable selection heuristic makes use of information obtained by solving the relaxation (e.g., this is true of branch and bound algorithms for integer linear programming). Our results will apply to the more restricted setting in which the relaxation is used only to determine whether or not to abandon a given partial assignment.

We now introduce some additional terminology. First, we think of the partial assignments as nodes in a tree being searched by the branch and bound algorithm. For any tree node $\mathcal{P}$, let $\bar{z}(\mathcal{P})$ be the value of the upper bound when node $\mathcal{P}$ is encountered. Note that $\bar{z}(\mathcal{P})$ is independent of the relaxation policy; otherwise the relaxation would be invalid! Call a node *prunable* if $relax(\mathcal{P}) \geq \bar{z}(\mathcal{P})$.

As an example, consider the branch and bound search tree depicted in Figure 4, where prunable nodes are indicated by an X. Call this problem instance I. Suppose that $\mathcal{R}$ contains a single relaxation, solving this relaxation always requires 1 time step, and that the other steps required to process a search tree node (e.g., variable selection, value ordering) require a total of 1 time step. The relaxation policy $\pi_{always}(d) = \{R\}$, which applies the relaxation at every internal node in the search tree, has cost $c(\pi_{always}, I) = 12$, because it must spend 2 time steps on each of 5 internal nodes and 1 time step on each of 2 leaf nodes. The relaxation policy $\pi_{never}(d) = \emptyset$ has cost $c(\pi_{never}, I) = 15$ because it must spend 1 time step on each of the 15 nodes in the search tree.

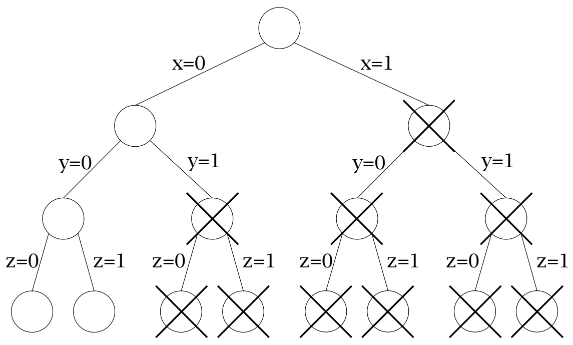

Figure 4: A branch and bound search tree. Prunable nodes are indicated by an X.

5.2.1 Reduction to online shortest paths

In this section we show that the problem of learning relaxation policies can be reduced to an online shortest paths problem (see §2.1). For simplicity we will assume $|\mathcal{R}| = 1$; the generalization to $|\mathcal{R}| > 1$ will be straightforward.

Each vertex will have coordinates $\langle d, i\rangle$, where d represents the current depth in the search tree and i represents the largest depth $< d$ at which the relaxation was applied (i.e., i is the largest value $< d$ such that $R \in \pi(d)$, where R is the single element of $\mathcal{R}$). Each vertex $\langle d, i\rangle$ has outgoing edges to vertices at coordinates $\langle d+1, d\rangle$ and $\langle d+1, i\rangle$.

To assign costs to the edges in this graph, let $nodes(d, i)$ denote the set of nodes that would be encountered at level d given that the relaxation was last applied at level i (i.e., $nodes(d, i)$ is the set of nodes at level d whose level i ancestor is not prunable). The cost assigned to the edge from $\langle d, i\rangle$ to $\langle d+1, i\rangle$ is equal to the sum of the processing costs of each node in $nodes(d, i)$, whereas the cost assigned to the edge from $\langle d, i\rangle$ to $\langle d+1, d\rangle$ is equal to the sum of the processing costs of each node in $nodes(d, i)$ plus the cost of solving the relaxation at each node in $nodes(d, i)$.

A relaxation policy π corresponds to a path from s to t which, at each vertex $\langle d, i\rangle$, takes the edge to $\langle d+1, d\rangle$ if $\pi(d) = \{R\}$ and takes the edge to $\langle d+1, i\rangle$ if $\pi(d) = \emptyset$. By construction, the cost of this path is equal to $c(\pi, I)$.

Figure 5 illustrates the graph that corresponds to the search tree depicted in Figure 4. The policy $\pi_{always}(d) = \{R\}$ corresponds to diagonally upward path from s to t, with cost 12, while the policy $\pi_{never}(d) = \emptyset$ corresponds to the horizontal path from s to t, with cost 15. The shortest path from s to t has cost 11, and corresponds to the policy of only solving the relaxation at level 1 of the search tree.

If $k = |\mathcal{R}| > 1$, each vertex will have a label of the form $\langle d, d_1, d_2, \ldots, d_k\rangle$, where d is the current depth in the search tree, and d_i is the last depth at which relaxation i has been applied. The number of vertices is of course exponential in k, and we expect our approach to be effective when k is relatively small.

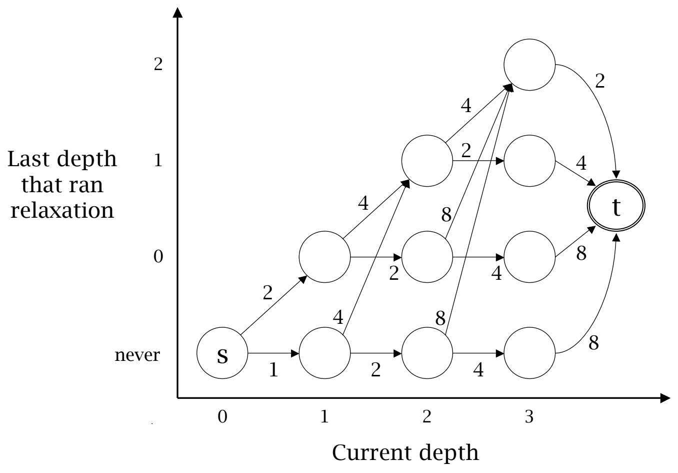

Figure 5: Shortest path problem corresponding to the tree in Figure 4.

5.3 Selecting constraint propagation policies online

Constraint propagation is a key component of branch and bound algorithms. Briefly, a constraint propagator p_i is a function that takes as input a partial assignment $\mathcal{P}$ and upper bound $\bar{z}$, and returns a set $\mathcal{C}_i = p(\mathcal{P}, \bar{z})$ such that $\{x \in \mathcal{F} \cap \mathcal{P} : f(x) < \bar{z}\} \subseteq \mathcal{C}_i$ (we assume the propagator has access to the feasible set $\mathcal{F}$ through a global variable). If $\{p_1, p_2, \ldots, p_k\}$ is the set of propagators being used, the relaxation is solved on $\mathcal{F}_{relax} \cap \mathcal{P} \cap \mathcal{C}_1 \cap C_2 \cap \ldots \cap C_k$, rather than $\mathcal{F}_{relax} \cap \mathcal{P}$. An as example, branch and bound algorithms for integer linear programming generate linear inequalities called cutting planes that must be satisfied by any optimal solution (each C_i is the set of points that satisfy one or more of these linear inequalities).

In many domains in which branch and bound algorithms are used, a variety of constraint propagators are available. The propagators differ in the amount of CPU time they require as well as the number and quality of constraints they are able to infer. The time required by a branch and bound algorithm to solve an instance can be dramatically affected by the choice of constraint propagator(s) to run at each search tree node, but in general there is no systematic way to make this choice.

Under suitable assumptions, our results on selecting relaxation policies carry over to the case of selecting constraint propagation policies. As in the problem of learning relaxation policies, our results will hold only when the constraint propagation policy has no effect on the variable selection or value ordering heuristics. The other assumption we require is that the time required to compute the set returned by each propagator p at some node at level d in the search tree is a function of the vector $\langle d_1, d_2, \ldots, d_k \rangle$, where d_i is the largest depth $< d$ at which propagator p_i was last run. Under this assumption (which does not seem too restrictive), the reduction described in the previous section gives a no-regret strategy for selecting constraint propagation policies online.

6 The Max k-Armed Bandit Problem

In the classical k-armed bandit problem [3] one is faced with a set of k slot machines, each having an arm that, when pulled, yields a payoff drawn from a fixed (but unknown) distribution. The goal is to allocate trials to the arms so as to maximize the cumulative payoff obtained over a series of n trials. Solving the problem entails striking a balance between exploration (determining which arm yields the highest mean payoff) and exploitation (repeatedly pulling this arm).

In the *max* variant of the k-armed bandit problem, the goal is to maximize the *maximum* (rather than cumulative) payoff. This version of the problem is designed to model tradeoffs between exploration and exploitation that are faced when using randomized algorithms to solve optimization problems. The problem was introduced by Cicirello & Smith [9, 11], who showed that a strategy for the max k-armed bandit was effective at selecting among randomized heuristics for solving a set of benchmark scheduling problems.

6.1 Applications

6.1.1 Selecting among heuristics

Suppose we are given k randomized heuristics to use in solving an optimization problem. Each heuristic, when run, produces a solution with a certain numerical quality (higher quality being better). Given a budget of n runs, we would like to allocate computation time to the k heuristics so as to maximize the quality of the best solution obtained from any of the n runs (e.g., the maximum number of clauses satisfied by any sampled variable assignment, the minimum makespan of any sampled schedule). Cicirello and Smith [9, 11] obtain good performance on the resource-constrained project scheduling problem with maximal time lags (RCPSP/max) by using an algorithm for the max k-armed bandit problem to select among five randomized priority dispatching rules.

6.1.2 Tuning the parameters of multi-start heuristics

A *multi-start heuristic* solves an optimization problem by repeatedly sampling solutions from a fixed probability distribution, and returning the best solution obtained. Typically, the probability distribution is governed by a large number of numeric parameters which are set to instance-specific values according to a heuristic rule. Examples of multi-start heuristics include *heuristic-biased stochastic sampling* [6], *value-biased stochastic sampling* [10], and the widely-used *greedy randomized adaptive search procedure* (GRASP) family of heuristics [4, 7, 15, 29].

In *adaptive multi-start*, the parameters of the probability distribution are modified based on feedback from earlier samples [5, 28, 31]. A strategy for solving the max k-armed bandit problem can be used in a natural way to convert a multi-start heuristic into an adaptive multi-start heuristic. Given an initial vector of parameters controlling the distribution from which solutions are sampled, we define k "neighboring" probability distributions with slightly altered parameter vectors, and use a max k-armed bandit strategy to select among them. This process can be applied iteratively until a locally optimal parameter vector is obtained.

6.2 Definitions and notation

An instance $I = \langle G_1, G_2, \ldots, G_k \rangle$ of the max k-armed bandit problem is k-tuple of probability distributions, each thought of as an arm on a slot machine. The i^{th} arm, when pulled, returns a payoff drawn independently at random from distribution G_i. The goal is to maximize the highest single payoff received over a sequence of n pulls.

A *strategy* $\mathcal{S}$ for solving the max k-armed bandit problem is a function that takes as input the history

$$h^{\mathcal{S}}_{j-1} = \langle (i_1, r_1), (i_2, r_2), \ldots, (i_{j-1}, r_{j-1}) \rangle$$

of arms pulled and payoffs received by using $\mathcal{S}$ for the first $j-1$ pulls, and produces as output the index $\mathcal{S}(h_{j-1}^{\mathcal{S}}) \in \{1, 2, \dots, k\}$ of the arm to pull next.

The *regret* of strategy $\mathcal{S}$ on instance I after n pulls is defined by

$$regret(I, \mathcal{S}, n) = \max_{1 \le i \le k} \mathbb{E}[M_n^i(I)] - \mathbb{E}[\mathcal{S}_n(I)] \tag{2}$$

where the random variable $M_n^i(I)$ is the maximum payoff obtained by pulling the i^{th} arm n times, and $\mathcal{S}_n(I)$ is the maximum payoff obtained by running $\mathcal{S}$ on I for n pulls. Equivalently, regret is equal to the expected difference between the maximum payoff obtained using S and the maximum payoff that would have been obtained using the best *pure* strategy (where a pure strategy is one that pull the same arm for all n pulls). Note that in contrast to the classical k-armed bandit problem (where the goal is to maximize cumulative payoff), the strategy that maximizes $\mathcal{S}_n(I)$ may not be a pure strategy, and it is possible to have $regret(I, \mathcal{S}, n) < 0$.

The worst-case regret of strategy S is the maximum value of (2) as a function of k and n. We say that S is a no-regret strategy if, for any fixed k, the worst-case regret is $o(1)$ as a function of n. Note that this is stronger than simply requiring the regret to be $o(1)$ as a function of n for any fixed k-tuple of distributions. Indeed, as long as payoffs are bounded then simply sampling the arms in round-robin order meets the latter requirement, but (as discussed in the next section) round-robin sampling is not a no-regret strategy in general.

6.3 Payoffs drawn from arbitrary distributions

Ideally, we would like to come up with a no-regret strategy for the max k-armed bandit problem that requires as few distributional assumptions as possible. As a first step, it seems reasonable to require that all payoffs come from a bounded interval (this will be true in all the applications we intend to consider). In fact, a no-regret strategy does not exist even under the stronger assumption that all payoffs are either 0 or 1. To see this, imagine that we randomly select one of the k arms to be "good"; this arm returns a payoff of 1 with probability $\frac{1}{n}$ and returns 0 otherwise. All other arms always return payoff 0. It can be shown that on instances drawn from this distribution, any strategy has expected regret at least $1 - \frac{1}{k} - \frac{1}{e}$.

6.4 Threshold Ascent

In this section we present a strategy for the max k-armed bandit problem called *Threshold Ascent* that is designed to work well on the type of payoff distributions we expect to encounter in practice, although (as just discussed) it cannot be expected to perform well if the payoff distributions are chosen adversarially. Roughly speaking, Threshold Ascent should perform well when the following two criteria are satisfied.

1. There is a (relatively low) threshold $t_{critical}$ such that, for all $t > t_{critical}$, the arm that is most likely to yield a payoff $> t$ is the same as the arm most likely to yield a payoff $> t_{critical}$. Call this arm i^*.
2. As t increases beyond $t_{critical}$, there is a growing gap between the probability that arm i^* yields a payoff $> t$ and the corresponding probability for other arms. Specifically, if we let $p_i(t)$ denote the probability that the i^{th} arm returns a payoff $> t$, the ratio $\frac{p_{i^*}(t)}{p_i(t)}$ should increase as a function of t for $t > t_{critical}$, for any $i \neq i^*$.

Figure 6 illustrates a set of two payoff distributions that satisfy these assumptions.

Threshold Ascent takes as input a strategy $\mathcal{S}$ for the classical k-armed bandit problem and an integer m. It maintains a variable threshold T, initially set to 0. $\mathcal{S}$ is used as a heuristic to obtain payoffs $> T$. Once m above-threshold payoffs have been obtained, T is incremented and the process continues.

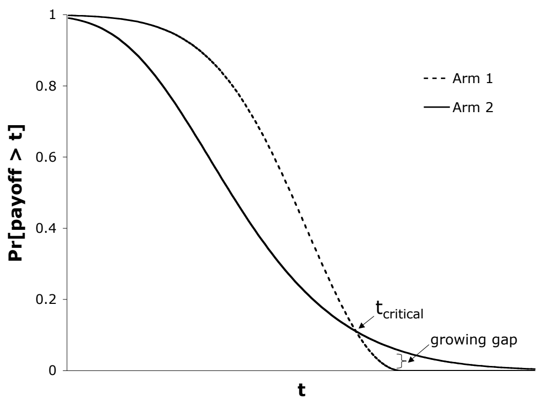

Figure 6: A max k-armed bandit instance on which Threshold Ascent should perform well.

The code for Threshold Ascent is given below. For simplicity, we assume that all payoffs are non-negative integers.

```
Procedure **ThresholdAscent**($\mathcal{S}$, $m$):
  1. Initialize $T \leftarrow 0$, $h \leftarrow \langle\rangle$.
  2. Do:
       (a) While $|\{(i,r) \in h : r > T\}| \geq m$ do:
                 $T \leftarrow T + 1$
       (b) $\hat{i} \leftarrow \mathcal{S}(h')$, where the history $h'$ is obtained by first copying
           $h$ and then replacing each pair $(i,r)$ with $(i,r')$, where $r' =$
           $0$ if $r \leq T$ and $r' = 1$ otherwise.
       (c) Pull arm $\hat{i}$, receive payoff $r$, and append $(\hat{i}, r)$ onto $h$.
```

The parameter m controls the tradeoff between exploration and exploitation. To understand this tradeoff, it is helpful to consider two extreme cases.

Case $m = 1$. When $m = 1$, Threshold Ascent passes as an argument to $\mathcal{S}$ a history in which all arm pulls have returned a payoff of 0. For reasonable $\mathcal{S}$, this results in the arms being sampled in a round-robin fashion.

Case $m = \infty$. When $m = \infty$, Threshold Ascent is equivalent to running $\mathcal{S}$ on a k-armed bandit instance where payoffs $> T$ are mapped to 1 and payoffs $\leq T$ are mapped to 0.

6.4.1 Experimental evaluation

Following Cicirello and Smith [9, 11], we evaluate our strategy for the max k-armed bandit problem by using it to select among randomized priority dispatching rules for the resource-constrained project scheduling

Table 3: Performance of eight max k-armed bandit strategies on 169 RCPSP/max instances.

Strategy	Σ **Regret**	$\mathbb{P}[\text{Regret} = 0]$
Threshold Ascent	188	0.722
Round-robin sampling	345	0.556
LPF	355	0.675
MTS	402	0.657
QD-BEACON	609	0.538
RSM	2130	0.166
LST	3199	0.095
MST	4509	0.107

problem with maximal time lags (RCPSP/max). We consider the five randomized priority dispatching rules in the set $\mathcal{H} = \{LPF, LST, MST, MTS, RSM\}$. See Cicirello and Smith [9, 11] for a complete description of these heuristics.

Briefly, in the RCPSP/max one must assign start times to each of a number of activities in such a way that certain temporal and resource constraints are satisfied. Such an assignment of start times is called a *feasible schedule*. The goal is to find a feasible schedule whose makespan is as small as possible, where makespan is defined as the maximum completion time of any activity. For a more complete description, see [27].

We evaluate our approach on a set $\mathcal{I}$ of 169 RCPSP/max instances from the ProGen/max library [34]. For each instance $I \in \mathcal{I}$, we ran each dispatching rule $h \in \mathcal{H}$ 10,000 times, storing the results in a file. Using this data, we created a set $\mathcal{K}$ of 169 five-armed bandit problems (each of the five heuristics $h \in \mathcal{H}$ represents an arm). After the data were collected, makespans were converted to payoffs by multiplying each makespan by -1 and scaling them to lie in the interval $[0, 1]$.

For each instance $K \in \mathcal{K}$, we ran three max k-armed bandit algorithms, each with a budget of $n = 10,000$ pulls: Threshold Ascent, the QD-BEACON algorithm of Cicirello and Smith [11], and an algorithm that simply sampled the arms in a round-robin fashion. When running Threshold Ascent we set $m = 100$ and for the subroutine $\mathcal{S}$ we used an interval estimation strategy based on Chernoff bounds. For each instance $K \in \mathcal{K}$, the regret of a strategy is the difference between the minimum makespan (which corresponds to the maximum payoff) sampled by the algorithm and the minimum makespan sampled by any of the five heuristics (on any of the $10,000$ stored runs of each of the five heuristics).

Table 3 summarizes our results. Of the eight max k-armed bandit strategies we evaluated (Threshold Ascent, QD-BEACON, round-robin sampling, and the five pure strategies), Threshold Ascent has the least regret (summed over all instances) and achieves zero regret on the largest number of instances.

6.5 Payoffs drawn from generalized extreme value distributions

In the previous section we presented a strategy that was designed to work well when the payoff distributions have the loosely-defined property illustrated in Figure 6. In this section we will make a stronger assumption motivated by a result in *extreme value theory*: we will assume each payoff distribution is a *generalized extreme value* (GEV) distribution. This assumption will be sufficient to allow us to develop a no-regret strategy.

Definition (GEV distribution). *A random variable Z has a* generalized extreme value *(GEV) distribution if*

$$\mathbb{P}[Z \leq z] = \exp\left(-\left(1 + \frac{\xi(z-\mu)}{\sigma}\right)^{-\frac{1}{\xi}}\right)$$

for some constants μ, $\sigma > 0$, and ξ.

The assumption that payoffs are drawn from a GEV is justified by the Extremal Types Theorem [12], which singles out the GEV as the limiting distribution of the maximum of a large number of independent identically distributed (i.i.d.) random variables. Roughly speaking, one can think of the Extremal Types Theorem as an analogue of the Central Limit Theorem. Just as the Central Limit Theorem states that the sum of a large number of i.i.d. random variables converges in distribution to a Gaussian, the Extremal Types Theorem states that the maximum of a large number of i.i.d. random variables converges in distribution to a GEV. A formal statement of the Extremal Types Theorem is given below.

Definition (convergence in shape). *A sequence $\langle f_1, f_2, \ldots \rangle$ of functions with common domain X and codomain Y* converges in shape *to $f : X \rightarrow Y$ if, for some sequences of constants $\langle a_1, a_2, \ldots \rangle$ and $\langle b_1, b_2, \ldots \rangle$,*

$$\lim_{n\rightarrow\infty} \max_{x \in X} \left| \frac{f_n(x) - a_n}{b_n} - f(x) \right| = 0 \, .$$

The Extremal Types Theorem. *Let Z be a random variable with cumulative distribution function G. Let $M^i = \max(Z_1, Z_2, \ldots Z_i)$ be the maximum of i independent realizations of Z, and let G_i be its cumulative distribution function (i.e., $G_i(z) = \mathbb{P}[M_i \leq z]$). If the sequence $\langle G_1, G_2 \ldots \rangle$ converges in shape to G^* and G^* is non-degenerate (i.e., not a point mass) then G^* is a GEV distribution.*

There are two arguments for studying the case where each payoff distribution is a GEV. First, in practice the distribution of payoffs returned by a strong heuristic may be approximately GEV even if the conditions of the Extremal Type Theorem are not formally satisfied [9]. Second, a max k-armed bandit instance with non-GEV payoff distributions starts to look like an instance with GEV payoff distributions if we take a coarse-grained view of the problem. In particular, if we sample a non-GEV payoff distribution in blocks of size b (i.e., a single arm pull samples the distribution b times and returns the maximum), the distribution of the block maxima converges in shape to a GEV as $b \rightarrow \infty$.

In previous work [36], we developed a no-regret strategy whose performance is summarized by the following theorem.

Theorem 2. *Fix an integer k, and let $\mathcal{I}_k^{GEV}$ be the set of max k-armed bandit instances whose arms all return payoffs drawn from GEV distributions. There exists a strategy $\mathcal{S}$ such that for any integer n,*

$$\max_{I \in \mathcal{I}_k^{GEV}} regret(I, \mathcal{S}, n) = O\left(\ln(nk)\ln(n)^2 \sqrt[3]{\frac{k}{n}}\right) = o(1) \, .$$

6.6 Proposed extensions

6.6.1 Heuristics based on block maxima

Threshold Ascent applies a classical k-armed bandit strategy $\mathcal{S}$ to a transformed history of payoffs, where payoffs $> T$ are mapped to 1 and payoffs $\leq T$ are mapped to zero. It may also be effective to apply $\mathcal{S}$ to a transformed history of payoffs where the maximum of b consecutive payoffs is mapped to a single "block maximum" payoff. By ramping up b over time, we ensure convergence to the desired arm.

6.6.2 Integrating parametric and non-parametric approaches

The advantage of Threshold Ascent is that it is designed to work well given only very weak distributional assumptions. The price for this generality is that it is unable to notice and exploit trends in the tails of the payoff distributions. In contrast, a strategy that assumes payoffs are drawn from GEV distributions is able to quickly make predictions about portions of the upper tail for which there is no observed data. We would like to develop a strategy that combines the advantages of a parametric fit (which makes more rapid extrapolation possible) with the robustness of Threshold Ascent.

6.6.3 Solving a sequence of max k-armed bandit instances

A drawback of selecting among heuristics using a max k-armed bandit approach is that it can only be done when a large number of runs of k multi-start heuristics are to be performed on a single problem instance. A more common situation would be for a smaller number of runs (e.g., 10-100 runs) to be performed on each instance in a sequence of instances.

To model this scenario, suppose that we are allowed n pulls on each of a sequence $\mathcal{I} = \langle I_1, I_2, \ldots, I_T \rangle$ of instances of the max k-armed bandit problem, and our goal is to maximize the sum of the maximum payoffs received on each instance. Note that while we assume that payoffs are drawn independently at random from fixed distributions within each instance, the sequence of instances may be determined adversarially. If $n = 1$, this is simply the "nonstochastic multiarmed bandit problem" studied by Auer et al. [1]. If n is very large relative to T, the problem effectively reduces to the max k-armed bandit problem. When n and T are of the same order of magnitude, however, strategies that combine attributes of max k-armed bandit strategies with attributes of strategies for the nonstochastic multiarmed bandit problem may be effective.

7 Extensions

7.1 Exploiting instance-specific features and expert advice

We have considered the setting in which a strategy $\mathcal{S}$ must solve a sequence of instances $\mathcal{I} = \langle I_1, I_2, \ldots, I_n \rangle$ by selecting a policy $\mathcal{S}(h^{\mathcal{S}}_{j-1}) \in \Pi$ to use to solve each instance I_j based only on the history vector $h^{\mathcal{S}}_{j-1} = \langle (\pi_1, o_1), (\pi_2, o_2), \ldots, (\pi_{j-1}, o_{j-1}) \rangle$ of policies and observations on the first $j-1$ instances. In practice, there are often a number of quickly-computable features that differentiate one problem instance from another. Instances with certain features may be solved more quickly by one policy, while instances with different features may be solved more quickly be another. We can imagine a setting in which $\mathcal{S}$ is supplied a vector f_j of one or more features along with an augmented history vector $h^{\mathcal{S}}_{j-1} = \langle (\pi_1, f_1, c_1), (\pi_1, f_2, c_2), \ldots, (\pi_{j-1}, f_{j-1}, c_{j-1}) \rangle$. On a related note, we may have access to one more *experts* that make predictions about the behavior of policies on particular instances, perhaps changing their predictions while the policy is running. For example, in the problem of *selecting restart schedules*, we may have access to a number of experts that predict how long a particular run of a Las Vegas algorithm will continue, given a description of its execution state (e.g., [22]). Adaptive game playing with expert advice has been well-studied from a theoretical point of view (e.g., [1]), and we expect that adapting these results to our setting should give significant performance improvements.

7.2 Learning how to solve subproblems in divide-and-conquer

It is natural to consider using our results on solving sequences of problems to try to solve the *subproblems* created by divide-and-conquer algorithms more efficiently as well. For example, given access to a set $\mathcal{H}$ of divide-and-conquer sorting algorithms we can define a space of policies $\pi : \mathbb{Z} \rightarrow \mathcal{H}$, where $\pi(d)$ specifies the algorithm to run on subproblems at depth d in the recursion tree (or perhaps subproblems of size d). Work by Lagoudakis and Littman [24] using a reinforcement learning approach to this task demonstrated good performance on sorting tasks.

We have seen examples of such policy spaces in the problem of *selecting relaxation policies* discussed in §5.2. There, we were able to take advantage of the fact that relaxations are well-behaved (i.e., the state of the search tree at depth d is completely described by the greatest depth $i < d$ at which the relaxation was applied) in order to guarantee low regret with respect to a globally optimal policy. In general, if the decisions made by a policy at one level of the recursion tree are allowed have an arbitrary effect on the result of later computations we do not expect to be able to provide such strong guarantees. However, even a naïve approach (e.g., running a no-regret algorithm at each level of the recursion tree) may prove of practical value in this setting.

References

[1] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[2] Baruch Awerbuch and Robert Kleinberg. Adaptive routing with end-to-end feedback: Distributed learning and geometric approaches. In *Proceedings of the* 36^{th} *ACM Symposium on Theory of Computing*, pages 45–53, 2004.

[3] Donald. A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, London, 1986.

[4] S. Binato, G. C. de Oliveira, and J. L. de Araujo. A greedy randomized adaptive search procedure for transmission expansion planning. *IEEE Transactions on Power Systems*, 16(2):247–253, 2001.

[5] Kenneth D. Boese, Andrew B. Kahng, and Sudhukar Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, 16:101–113, 1994.

[6] J. L. Bresina. Heuristic-biased stochastic sampling. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 271–278, 1996.

[7] Mauro Brunato and Roberto Battiti. A multistart randomized greedy algorithm for traffic grooming on mesh logical topologies. In *Proceedings of the Sixth IFIP ONDM*, pages 417–430, 2002.

[8] Nicolò Cesa-Bianchi, Gábor Lugosi, and Gilles Stoltz. Minimizing regret with label efficient prediction. *IEEE Transactions on Information Theory*, 51:2152–2162, 2005.

[9] Vincent A. Cicirello and Stephen F. Smith. Heuristic selection for stochastic search optimization: Modeling solution quality by extreme value theory. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 197–211, 2004.

[10] Vincent A. Cicirello and Stephen F. Smith. Enhancing stochastic search performance by value-biased randomization of heuristics. *Journal of Heuristics*, 11(1):5–34, 2005.

[11] Vincent A. Cicirello and Stephen F. Smith. The max k-armed bandit: A new model of exploration applied to search heuristic selection. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1355–1361, 2005.

[12] Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer-Verlag, London, 2001.

[13] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the* 6^{th} *International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

[14] Uriel Feige and Prasad Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.

[15] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[16] Carla Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfications problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[17] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 431–437, 1998.

[18] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.

[19] András György, Tamás Linder, and György Ottucsák. The shortest path problem under partial monitoring. In *Proceedings of the* 19^{th} *Annual Conference on Learning Theory*, pages 468–482, 2006.

[20] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.

[21] Adam Kalai and Santosh Vempala. Efficient algorithms for the online decision problem. In *Proceedings of the Sixteenth Conference on Computational Learning Theory*, pages 26–40, 2003.

[22] Henry Kautz, Yongshao Ruan, and Eric Horvitz. Dynamic restarts. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 674–681, 2002.

[23] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 318–325, 1999.

[24] Michail G. Lagoudakis and Michael L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518, San Francisco, CA, 2000. Morgan Kaufmann.

[25] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[26] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the* 38^{th} *Conference on Design Automation*, pages 530–535, 2001.

[27] Klaus Neumann, Christoph Schwindt, and Jürgen Zimmerman. *Project Scheduling with Time Windows and Scarce Resources*. Springer-Verlag, 2002.

[28] Marcelo Prais and Celso C. Ribeiro. Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS Journal on Computing*, 12(3):164–176, 2000.

[29] Alexander J. Robertson. A set of greedy randomized adaptive local search procedure (grasp) implementations for the multidimensional assignment problem. *Computational Optimization and Applications*, 19(2):145–164, 2001.

[30] Yongshao Ruan, Eric Horvitz, and Henry 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, 2002.

[31] Wheeler Ruml. Incomplete tree search using adaptive probing. In *Proceedings of the Seventeenth International Joint Conference on Artificial Inteligence (IJCAI-01)*, pages 235–241, 2001.

[32] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Inc., 2003.

[33] Tuomas Sandholm. Terminating decision algorithms optimally. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 950–955, 2003.

[34] C. Schwindt. Generation of resource–constrained project scheduling problems with minimal and maximal time lags. Technical Report WIOR-489, Universität Karlsruhe, 1996.

[35] B. Selman, H. Kautz, and B. Cohen. Noise strategies for local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 337–343, 1994.

[36] Matthew J. Streeter and Stephen F. Smith. An asymptotically optimal algorithm for the max k-armed bandit problem. In *Proceedings of the Twenty-First National Conference on Artificial lntelligence*, pages 135–142, 2006.

[37] Matthew J. Streeter and Stephen F. Smith. A simple distribution-free approach to the max k-armed bandit problem. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming*, 2006.

[38] Zhao Xing, Yixin Chen, and Weixiong Zhang. MaxPlan: Optimal planning by decomposed satisfiability and backward reduction. In *Proceedings of the Fifteenth International Planning Competition, International Conference on Automated Planning and Scheduling*, pages 53–56, 2006.