Using Decision Procedures Efficiently for Optimization

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

Abstract

Optimization problems are often solved by making repeated calls to a decision procedure that answers questions of the form "Does there exist a solution with cost at most k?". In general the time required by the decision procedure varies widely as a function of k, so it is natural to seek a query strategy that minimizes the time required to find an (approximately) optimal solution. We present a simple query strategy with attractive theoretical guarantees. In case the same decision procedure is used for multiple optimization problems, we discuss how to tailor the query strategy to the sequence of problems encountered. We demonstrate the power of our query strategy by using it to create (*i*) a modified version of SatPlan that finds provably approximately optimal plans quickly and (*ii*) a modified branch and bound algorithm for job shop scheduling that yields improved upper and lower bounds.

Introduction

Optimization problems are often solved using an algorithm for the corresponding decision problem as a subroutine. Each query to the decision procedure can be represented as a pair $\langle k, t\rangle$, where t is a bound on the CPU time the decision procedure may consume in answering the question "Does there exist a solution with cost at most k?". The result of a query is either a (provably correct) "yes" or "no" answer or a timeout. A *query strategy* is a rule for determining the next query $\langle k, t\rangle$ as a function of the responses to previous queries.

The performance of a query strategy can be measured in several ways. Given a fixed query strategy and a fixed minimization problem, let $l(T)$ denote the lower bound (i.e., the largest k that elicited a "no" response plus one) obtained by running the query strategy for a total of T time units; and let $u(T)$ be the corresponding upper bound. A natural goal is for $u(T)$ to decrease as quickly as possible. Alternatively, we might want to achieve $u(T) \leq \alpha l(T)$ in the minimum possible time for some desired approximation ratio $\alpha \geq 1$.

In this paper we study the problem of designing query strategies. Our goal is to devise strategies that do well with respect to natural performance criteria such as the ones just described, when applied to decision procedures whose behavior (i.e., how the required CPU time varies as a function of k) is typical of the procedures used in practice.

Motivations

The two winners from the optimal track of last year's International Planning Competition were SatPlan (Kautz, Selman, & Hoffmann 2006) and MaxPlan (Xing, Chen, & Zhang 2006). Both planners find a minimum-makespan plan by making a series of calls to a SAT solver, where each call determines whether there exists a feasible plan of makespan $\leq k$ (where the value of k varies across calls). One of the differences between the two planners is that SatPlan uses the "ramp-up" query strategy (in which the i^{th} query is $\langle i, \infty\rangle$), whereas MaxPlan uses the "ramp-down" strategy (in which the i^{th} query is $\langle U - i, \infty\rangle$, where U is an upper bound obtained using heuristics).

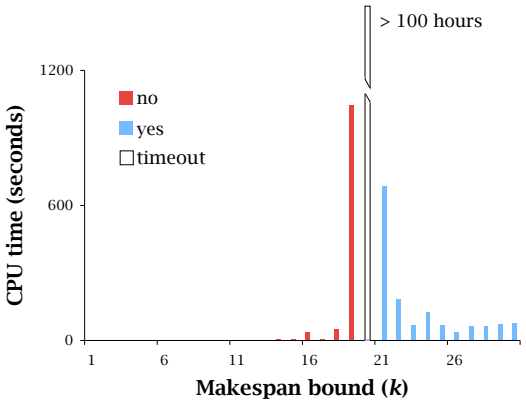

Figure 1: Behavior of the SAT solver `siege` running on formulae generated by SatPlan to solve instance `p17` from the `pathways` domain of the 2006 International Planning Competition.

To appreciate the importance of query strategies, consider Figure 1, which shows the CPU time required by `siege` (the SAT solver used by SatPlan) as a function of the makespan bound k, on a benchmark instance from the competition. For most values of k the solver terminates in under one minute; for $k = 19$ and $k = 21$ the solver requires

10-20 minutes; and for $k = 20$ the solver was run unsuccessfully for over 100 hours. Because only the queries with $k \geq 21$ return a "yes" answer, the ramp-up query strategy (used by SatPlan) does not find a feasible plan after running for 100 hours, while the ramp-down strategy returns a feasible plan but does not yield any non-trivial lower bounds on the optimum makespan. In this example, the time required by any query strategy to obtain a *provably optimal* plan is dominated by the time required to run the decision procedure with input $k = 20$. On the other hand, executing the queries $\langle 18, \infty \rangle$ and $\langle 23, \infty \rangle$ takes less than two minutes and yields a plan whose makespan is provably at most $\frac{23}{18+1} \approx 1.21$ times optimal. Thus, the choice of query strategy has a dramatic effect on the time required to obtain a *provably approximately optimal* solution. For planning problems where provably optimal plans are currently out of reach, obtaining provably approximately optimal plans quickly is a natural goal.

Summary of results

In this paper we consider the problem of devising query strategies in two settings. In the *single-instance* setting, we are confronted with a single optimization problem, and wish to obtain an (approximately) optimal solution as quickly as possible. In this setting we provide a simple query strategy S_2, and analyze its performance in terms of a parameter that captures the unpredictability of the decision procedure's behavior. We then show that our performance guarantee is optimal up to constant factors.

In the *multiple-instance* setting, we use the same decision procedure to solve a number of optimization problems, and our goal is to learn from experience in order to improve performance. In this setting, we prove that computing an optimal query strategy is NP-hard, and discuss how algorithms from machine learning theory can be used to select query strategies online.

In the experimental section of our paper, we demonstrate that query strategy S_2 can be used to create improved versions of state-of-the-art algorithms for planning and job shop scheduling. In the course of the latter experiments we develop a simple method for applying query strategies to branch and bound algorithms, which seems likely to be useful in other domains besides job shop scheduling.

Related work

The ramp-up strategy was used in the original GraphPlan algorithm (Blum & Furst 1997), and is conceptually similar to iterative deepening (Korf 1985).

Alternatives to the ramp-up strategy were investigated by Rintanen (2004), who proposed two algorithms. Algorithm A runs the decision procedure on the first n decision problems in parallel, each at equal strength, where n is a parameter. Algorithm B runs the decision procedure on all decision problems simultaneously, with the i^{th} problem receiving a fraction of the CPU time proportional to γ^i, where $\gamma \in (0, 1)$ is a parameter. Rintanen showed that Algorithm B could yield dramatic performance improvements relative to the ramp-up strategy.

Our query strategy S_2 exploits binary search and is quite different from the three strategies just discussed. In the experimental section of our paper, we compare S_2 to the ramp-up strategy and to a geometric strategy based on Algorithm B.

Preliminaries

In this paper we are interested in solving minimization problems of the form

$$OPT = \min_{x \in \mathcal{X}} c(x)$$

where $\mathcal{X}$ is an arbitrary set and $c : \mathcal{X} \to \mathbb{Z}_+$ is a function assigning a positive integer cost to each $x \in \mathcal{X}$. We will solve such a minimization problem by making a series of calls to a decision procedure that, given as input an integer k, determines whether there exists an $x \in \mathcal{X}$ with $c(x) \leq k$. When given input k, the decision procedure runs for $\tau(k)$ time units before returning a (provably correct) "yes" or "no" answer. Thus from our point of view, a minimization problem is completely specified by the integer OPT and the function τ.

Definition (instance). *An* instance *of a minimization problem is a pair $\langle OPT, \tau \rangle$, where OPT is the smallest input for which the decision procedure answers "yes" and $\tau(k)$ is the CPU time required by the decision procedure when it is run with input k.*

A *query* is a pair $\langle k, t \rangle$. To execute this query, one runs the decision procedure with input k subject to a time limit t. Executing query $q = \langle k, t \rangle$ on instance $I = \langle OPT, \tau \rangle$ requires CPU time $\min\{t, \tau(k)\}$ and elicits the response

$$\text{response}(I, q) = \begin{cases} \text{yes} & \text{if } t \geq \tau(k) \text{ and } k \geq OPT \\ \text{no} & \text{if } t \geq \tau(k) \text{ and } k < OPT \\ \text{timeout} & \text{if } t < \tau(k) \ . \end{cases}$$

Definition (query strategy). *A* query strategy *S is a function that takes as input the sequence $\langle r_1, r_2, \ldots, r_i \rangle$ of responses to the first i queries, and returns as output a new query $\langle k, t \rangle$.*

When executing queries according to some query strategy, we maintain upper and lower bounds on OPT. Initially $l = 1$ and $u = \infty$. If query $\langle k, t \rangle$ elicits a "no" response we set $l \leftarrow k + 1$; and if it elicits a "yes" reponse we set $u \leftarrow k$. We use $l(S, I, T)$ and $u(S, I, T)$, respectively, to denote the lower (resp. upper) bound obtained by spending a total of T time steps executing queries on instance I according to strategy S.

Performance of query strategies

In the single-instance setting, we will evaluate a query strategy according to the following competitive ratio.

Definition (α competitive ratio). *The* α competitive ratio *of a query strategy S on instance I is defined by*

$$\text{ratio}(S, I, \alpha) = \frac{T^S}{T^*}$$

where

$$T^S = \min \left\{ T \geq 0 : \frac{u(S, I, T)}{l(S, I, T)} \leq \alpha \right\}$$

is the time the query strategy requires to obtain a solution whose cost is (provably) at most αOPT and

$$T^* = \min\left\{\tau(k_1) + \tau(k_2) : \frac{k_2}{\alpha} \leq k_1 + 1 \leq OPT \leq k_2\right\}$$

is the minimum time required by a pair of queries that provides a solution whose cost is (provably) at most αOPT.

Behavior of τ

The performance of our query strategies will depend on the behavior of the function τ. For most decision procedures used in practice, we expect $\tau(k)$ to be an increasing function for $k \leq OPT$ and a decreasing function for $k \geq OPT$ (e.g., see the behavior of `siege` illustrated in Figure 1), and our query strategies are designed to take advantage of this behavior. More specifically, our query strategies are designed to work well when τ is close to its *hull*.

Definition (hull). *The* hull *of τ is the function*

$$\mathrm{hull}^\tau(k) = \min\left\{\max_{k_0 \leq k} \tau(k_0), \max_{k_1 \geq k} \tau(k_1)\right\}.$$

Figure 2 gives an example of a function τ (gray bars) and its hull (dots). Note that the region under the curve $\mathrm{hull}^\tau(k)$ is *not* (in general) the convex hull of the points $(k, \tau(k))$. Also note that the functions τ and hull^τ are identical if τ is monotonically increasing (or monotonically decreasing), or if there exists an x such that τ is monotonically increasing for $k \leq x$ and monotonically decreasing for $k > x$.

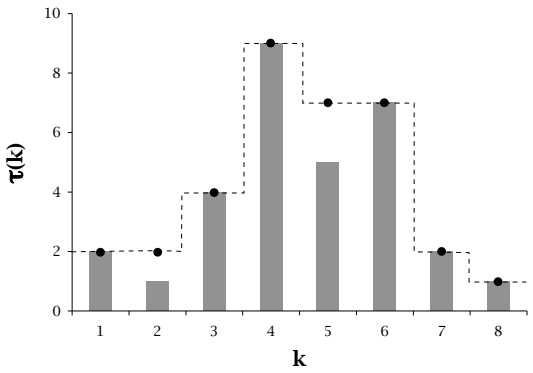

Figure 2: A function τ (gray bars) and its hull (dots).

We measure the discrepancy between τ and its hull in terms of the *stretch* of an instance.

Definition (stretch). *The* stretch *of an instance $I = \langle OPT, \tau\rangle$ is defined by*

$$\mathrm{stretch}(I) = \max_k \frac{\mathrm{hull}^\tau(k)}{\tau(k)}$$

The instance depicted in Figure 2 has a stretch of 2 because $\tau(2) = 1$ while $\mathrm{hull}^\tau(2) = 2$.

The Single-Instance Setting

We first consider the case in which we wish to design a query strategy for use in solving a single instance $I = \langle OPT, \tau\rangle$ (where OPT and τ are of course unknown to us). Our goal is to devise a query strategy that minimizes the value of $\mathrm{ratio}(S, I, \alpha)$ for the worst case instance I, for some fixed $\alpha \geq 1$. Although one might expect our query strategies to take α as a parameter, we will find it more natural to design a query strategy that works well *simultaneously* for all α. We assume $OPT \in \{1, 2, \ldots, U\}$ for some known upper bound U. For simplicity, we also assume $\tau(k) \geq 1$ for all k.

Arbitrary instances

In the case where the function τ is arbitrary, the following simple query strategy S_1 achieves an α competitive ratio that is optimal (to within constant factors). S_1 and its analysis are similar to those of Algorithm A of Rintanen (2004). We do not advocate the use of S_1. Rather, its analysis indicates the limits imposed by making no assumptions about τ.

Query strategy S_1:
1. Initialize $T \leftarrow 1$, $l \leftarrow 1$, and $u \leftarrow U$.
2. While $l < u$:
   (a) For each $k \in \{l, l+1, \ldots, u-1\}$, execute the query $\langle k, T\rangle$, and update l and u appropriately (if the response is "yes" then set $u \leftarrow k$, and if the response is "no" then set $l \leftarrow k+1$).
   (b) Set $T \leftarrow 2T$.

The analysis of S_1 is straightforward. Consider some fixed k, and let $T_k = 2^{\lceil \log_2 \tau(k)\rceil}$ be the smallest power of two that is $\geq \tau(k)$. After we run S_1 for time $U + 2U + 4U + \ldots + T_k U \leq 2T_k U \leq 4\tau(k)U$, we will either have $l > k$ or $u \leq k$. Because this holds for all k, it follows that $\mathrm{ratio}(S_1, I, \alpha) \leq 4U$.

To obtain a matching lower bound, suppose that $\tau(k) = 1$ if $k = OPT - 1$ or $k = OPT$, and $\tau(k) = \infty$ otherwise. For any query strategy S, there is some choice of OPT that forces S to consume time at least $\frac{U}{2}$ before executing a successful query, which implies $\mathrm{ratio}(S, I, \alpha) \geq \frac{U}{4}$. These observations are summarized in the following theorem.

Theorem 1. *For any instance I and any $\alpha \geq 1$, $\mathrm{ratio}(S_1, I, \alpha) = O(U)$. Furthermore, for any strategy S and $\alpha \geq 1$, there exists an instance I such that $\mathrm{ratio}(S, I, \alpha) = \Omega(U)$.*

Instances with low stretch

In practice we do not expect τ to be as pathological as the function used to prove the lower bound in Theorem 1. Indeed, as already mentioned, in practice we expect instances to have low stretch, whereas the instance used to prove the lower bound has infinite stretch. We now describe a query strategy S_2 whose competitive ratio is $O(\mathrm{stretch}(I) \cdot \log U)$, a dramatic improvement over Theorem 1 for instances with low stretch.

Like S_1, strategy S_2 maintains an interval $[l, u]$ that is guaranteed to contain OPT, and maintains a value T that is periodically doubled. S_2 also maintains a "timeout interval"

$[t_l, t_u]$ with the property that the queries $\langle t_l, T\rangle$ and $\langle t_u, T\rangle$ have both been executed and returned a timeout response.

Each query executed by S_2 is of the form $\langle k, T\rangle$, where $k \in [l, u-1]$ but $k \notin [t_l, t_u]$. We say that such a k value is *eligible*. The queries are selected in such a way that the number of eligible k values decreases exponentially. Once there are no eligible k values, T is doubled and $[t_l, t_u]$ is reset to the empty interval (so each $k \in [l, u-1]$ becomes eligible again).

Query strategy S_2:

1. Initialize $T \leftarrow 2$, $l \leftarrow 1$, $u \leftarrow U$, $t_l \leftarrow \infty$, and $t_u \leftarrow -\infty$.
2. While $l < u$:
   (a) If $[l, u-1] \subseteq [t_l, t_u]$ then set $T \leftarrow 2T$, set $t_l \leftarrow \infty$, and set $t_u \leftarrow -\infty$.
   (b) Let $u' = u - 1$. Define

$$k = \begin{cases} \left\lfloor \frac{l+u'}{2} \right\rfloor & \text{if } [l, u'] \text{ and } [t_l, t_u] \text{ are disjoint or } t_l = \infty \\ \left\lfloor \frac{l+t_l-1}{2} \right\rfloor & \text{if } [l, u'] \text{ and } [t_l, t_u] \text{ intersect and } t_l - l > u' - t_u \\ \left\lfloor \frac{t_u+1+u'}{2} \right\rfloor & \text{otherwise.} \end{cases}$$

   (c) Execute the query $\langle k, T\rangle$. If the result is "yes" set $u \leftarrow k$; if the result is "no" set $l \leftarrow k+1$; and if the result is "timeout" set $t_l \leftarrow \min\{t_l, k\}$ and set $t_u \leftarrow \max\{t_u, k\}$.

To analyze S_2, we first bound the number of queries that can be executed in between updates to T. As already mentioned, the k value defined in step 2(b) belongs to the interval $[l, u-1]$ but not to the interval $[t_l, t_u]$. By examining each case, we find that the number of k values that have this property goes down by a factor of at least $\frac{1}{4}$ every query, except for the very first query that causes a timeout. It follows that the number of queries in between updates to T is $O(\log U)$.

To complete the analysis, first note that whenever $t_l \neq \infty$ and $t_u \neq -\infty$, it holds that $\tau(t_l) > T$ and $\tau(t_u) > T$. For any $k \in [t_l, t_u]$, this implies $\mathrm{hull}^\tau(k) > T$ (by definition of hull) and thus $\tau(k) > \frac{T}{\mathrm{stretch}(I)}$ (by definition of stretch). Now consider some arbitrary k. Once $T \geq \mathrm{stretch}(I) \cdot \tau(k)$ it cannot be that $k \in [t_l, t_u]$, so we must have $k \notin [l, u-1]$ before T can be doubled again. Because there can be at most $O(\log U)$ queries in between updates to T, it follows that we have to wait $O(\mathrm{stretch}(I) \cdot \tau(k) \cdot \log U)$ time before $k \notin [l, u-1]$. Because this holds for all k, it follows that $\mathrm{ratio}(S_1, I, \alpha) = O(\mathrm{stretch}(I) \cdot \log U)$.

We now use a simple information-theoretic argument to prove a matching lower bound. Fix some query strategy S. Let $\tau(k) = 1$ for all k (clearly, $\mathrm{stretch}(I) = 1$). Assume without loss of generality that S only executes queries of the form $\langle k, 1\rangle$. For each $OPT \in \{1, 2, \ldots, U\}$, S must elicit a unique sequence of "yes" or "no" answers, one of which must have length $\geq \lfloor \log_2 U \rfloor$. Thus for some choice of OPT, $\mathrm{ratio}(S, I, 1) \geq \frac{\lfloor \log_2 U \rfloor}{2} = \Omega(\mathrm{stretch}(I) \cdot \log U)$. Thus we have proved the following theorem.

Theorem 2. *For any function τ and any $\alpha \geq 1$, $\mathrm{ratio}(S_2, \tau, \alpha) = O(\mathrm{stretch}(I) \cdot \log U)$. Furthermore, for any strategy S and $\alpha \geq 1$, there exists an instance I such that $\mathrm{ratio}(S, I, \alpha) = \Omega(\mathrm{stretch}(I) \cdot \log U)$.*

Generalizing S_2

Although the performance of query strategy S_2 (as summarized in Theorem 2) is optimal to within constant factors, in practice one might want to adjust the behavior of S_2 so as to obtain better performance on a particular set of optimization problems. Toward this end, we generalize S_2 by introducing three parameters: β controls the value of k; γ controls the rate at which the time limit T is increased; and ρ controls the balance between the time the strategy spends working to improve its lower bound versus the time it spends working to improve the upper bound. Each parameter takes on a value between 0 and 1. The parameters were chosen so as to include several natural query strategies in the parameter space. The original strategy S_2 is recovered by setting $\beta = \gamma = \rho = \frac{1}{2}$. When $\beta = \gamma = 0$ and $\rho = 0$, S_3 is equivalent to the ramp-up query strategy (in which the i^{th} query is $\langle i, \infty\rangle$). When $\beta = \gamma = 0$ and $\rho = 1$, S_3 is equivalent to the ramp-down query strategy (in which the i^{th} query is $\langle U - i, \infty\rangle$).

The analysis of S_3 follows along exactly the same lines as that of S_2. Retracing the argument leading up to Theorem 2 and working out the appropriate constant factors yields the following theorem, which shows that the class $S_3(\beta, \gamma, \rho)$ includes a wide variety of query strategies with performance guarantees similar to that of S_2 (note that the theorem provides no guarantees when $\beta = 0$ or $\gamma = 0$, as in the ramp-up and ramp-down strategies).

Theorem 3. *Let $S = S_3(\beta, \gamma, \rho)$, where $0 < \beta \leq \frac{1}{2}$, $0 < \gamma < 1$, and $0 < \rho < 1$. Then for any instance I and any $\alpha \geq 1$, $\mathrm{ratio}(S, I, \alpha) = O(\frac{1}{\beta\gamma} \cdot \mathrm{stretch}(I) \cdot \log U)$.*

Query strategy $S_3(\beta, \gamma, \rho)$:

1. Initialize $T \leftarrow \frac{1}{\gamma}$, $l \leftarrow 1$, $u \leftarrow U$, $t_l \leftarrow \infty$, and $t_u \leftarrow -\infty$.
2. While $l < u$:
   (a) If $[l, u-1] \subseteq [t_l, t_u]$ then set $T \leftarrow \frac{T}{\gamma}$, set $t_l \leftarrow \infty$, and set $t_u \leftarrow -\infty$.
   (b) Let $u' = u - 1$. If $[l, u']$ and $[t_l, t_u]$ are disjoint (or $t_l = \infty$) then define

$$k = \begin{cases} \lfloor (1-\beta)l + \beta u' \rfloor & \text{if } (1-\rho)l > \rho(U - u') \\ \lfloor \beta l + (1-\beta)u' \rfloor & \text{otherwise;} \end{cases}$$

   else define

$$k = \begin{cases} \lfloor (1-\beta)l + \beta(t_l - 1) \rfloor & \text{if } (1-\rho)(t_l - l) > \rho(u' - t_u) \\ \lfloor (1-\beta)u' + \beta(t_u + 1) \rfloor & \text{otherwise.} \end{cases}$$

   (c) Execute the query $\langle k, T\rangle$. If the result is "yes" set $u \leftarrow k$; if the result is "no" set $l \leftarrow k+1$; and if the result is "timeout" set $t_l \leftarrow \min\{t_l, k\}$ and set $t_u \leftarrow \max\{t_u, k\}$.

The Multiple-Instance Setting

We now consider the case in which the same decision procedure is used to solve a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of instances of some optimization problem. In this case, it is natural to attempt to learn something about the instance sequence and select query strategies accordingly.

Let $\mathcal{S}$ be some set of query strategies, and for any $S \in \mathcal{S}$, let $c_i(S)$ denote the CPU time required to obtain an acceptable solution to instance $x_i = \langle OPT_i, \tau_i \rangle$ using query strategy S (e.g., $c_i(S)$ could be the time required to obtain a solution whose cost is provably at most α times optimal). We consider the problem of selecting query strategies in two settings: offline and online.

Computing an optimal query strategy offline

In the offline setting we are given as input the values of $\tau_i(k)$ for all i and k, and wish to compute the query strategy

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

This offline optimization problem arises in practice when the instances $\langle x_1, x_2, \ldots, x_n \rangle$ have been collected for use as training data, and we wish to compute the strategy S^* that performs optimally on the training data.

Unfortunately, if $\mathcal{S}$ contains all possible query strategies then computing S^* is NP-hard. To see this, suppose that our goal is to obtain an approximation ratio $\alpha = U-1$. To obtain this ratio, we simply need to execute a single query that returns a non-timeout response. Consider the special case that $\tau_i(k) \in \{1, \infty\}$ for all i and k, and without loss of generality consider only query strategies that issue queries of the form $\langle k, 1 \rangle$. For our purposes, such a query strategy is just a permutation of the k values in the set $\{1, 2, \ldots, U\}$. For each k, let $A_k = \{x_i : \tau_i(k) = 1\}$. To find an optimal query strategy, we must order the sets $A_1, A_2, \ldots, A_U$ from left to right so as to minimize the sum, over all instances x_i, of the position of the leftmost set that contains x_i. This is exactly *min-sum set cover*. For any $\epsilon > 0$, obtaining a $4 - \epsilon$ approximation to min-sum set cover is NP-hard (Feige, Lovász, & Tetali 2004). Thus we have the following theorem.

Theorem 4. *For any $\epsilon > 0$, obtaining a $4-\epsilon$ approximation to the optimal query strategy is NP-hard.*

Certain special cases of the offline problem are tractable. For example, suppose all queries take the same time, say $\tau_i(k) = t$ for all i and k. In this case we need only consider queries of the form $\langle k, t \rangle$, and any such query elicits a non-timeout response. A query strategy can then be specified as a binary search tree over the key set $\{1, 2, \ldots, U\}$. The optimal query strategy is simply the optimum binary search tree for the access sequence $\langle OPT_1, OPT_2, \ldots, OPT_n \rangle$, which can be computed in $O(U^2)$ time using dynamic programming (Knuth 1971). Similarly, if we consider arbitrary τ_i but restrict ourselves to queries of the form $\langle k, \infty \rangle$ (so that again all queries succeed), dynamic programming can be used to compute an optimal query strategy. Finally, the offline problem is tractable if $\mathcal{S}$ is small enough for us to search through it by brute force. Based on the results of the previous section, a natural choice would be for $\mathcal{S}$ to include $S_3(\beta, \gamma, \rho)$ for various values of the three parameters.

Selecting query strategies online

We now consider the problem of selecting query strategies in an online setting, assuming that $|\mathcal{S}|$ is small enough that we would not mind using $O(|\mathcal{S}|)$ time or space for decision-making. In the online 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 query strategy $S_i \in \mathcal{S}$. 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

$$\frac{1}{n} \cdot \left(\mathbb{E}\left[\sum_{i=1}^{n} c_i(S_i) \right] - \min_{S \in \mathcal{S}} \sum_{i=1}^{n} c_i(S) \right) \qquad (1)$$

where the expectation is over any random bits used by our strategy-selection algorithm. An algorithm's worst-case regret is the maximum value of (1) over all instance sequences of length n. A *no-regret algorithm* has worst-case regret that is $o(1)$ as a function of n.

We now describe how two existing algorithms can be applied to the problem of selecting query strategies. Let M be an upper bound on $c_i(S)$, and let T be an upper bound on $\tau_i(k)$. Viewing our online problem as an instance of the "nonstochastic multiarmed bandit problem" and using the **Exp3** algorithm of Auer et al. (2002) yields regret $O\left(M\sqrt{\frac{1}{n}|\mathcal{S}|}\right) = o(1)$. The second algorithm makes use of the fact that on any particular instance x_i, we can obtain enough information to determine the value of $c_i(S)$ for all $S \in \mathcal{S}$ by executing the query $\langle k, T \rangle$ for each $k \in \{1, 2, \ldots, U\}$. This requires CPU time at most TU. We can then use the "label-efficient forecaster" of Cesa-Bianchi et al. (2005) to select query strategies. Theorem 1 of that paper shows that the regret is at most $M\left(\ln \frac{|\mathcal{S}|}{\eta} + n\frac{\eta}{2\varepsilon}\right) + \varepsilon n T U$, where η and ε are parameters. Optimizing η and ε yields regret $O\left(M\left(\frac{TU \ln |\mathcal{S}|}{Mn}\right)^{\frac{1}{3}}\right) = o(1)$. Given n as input, one can choose whichever of the two algorithms yields the smaller regret bound.

Experimental Evaluation

In this section we evaluate query strategy S_2 experimentally by using it to create modified versions of state-of-the-art solvers in two domains: STRIPS planning and job shop scheduling. In both of these domains, we found that the number of standard benchmark instances was too small for the online algorithms discussed in the previous section to be effective. Accordingly, our experimental evaluation focuses on the techniques developed for the single-instance setting.

Planning

The planners entered in the 2006 International Planning Competition were divided into two categories: *optimal* planners always return a plan of provably minimum makespan, whereas *satisficing* planners simply return a feasible plan

quickly. In this section we pursue a different goal: obtaining a *provably* near-optimal plan as quickly as possible.

As already mentioned, SatPlan finds a minimum-makespan plan by making a sequence of calls to a SAT solver that answers questions of the form "Does there exist a plan of makespan $\leq k$?". The original version of SatPlan tries k values in an increasing sequence starting from $k = 1$, stopping as soon as it obtains a "yes" answer. We compare the original version to a modified version that instead uses query strategy S_2. When using S_2 we do not share any work (e.g., intermediate result files) among queries with the same k value, although doing so could improve performance.

We ran each of these two versions of SatPlan on benchmark instances from the 2006 International Planning Competition, with a one hour time limit per instance, and recorded the upper and lower bounds we obtained. To obtain an initial upper bound, we ran the satisficing planner `SGPlan` (Hsu *et al.* 2006) with a one minute time limit. We chose `SGPlan` because it won first prize in the *satisficing planning* track of last year's competition. If `SGPlan` found a feasible plan within the one minute time limit, we used the number of actions in that plan as an upper bound on the optimum makespan; otherwise we artificially set the upper bound to 100.

Table 1 presents our results for 30 instances from the `pathways` domain. Numbers in bold indicate an upper or lower bound obtained by one query strategy that was strictly better than the bound obtained by any other query strategy. Not surprisingly, S_2 always obtains upper bounds that are as good or better than those obtained by the ramp-up strategy. Interestingly, the lower bounds obtained by S_2 are only slightly worse, differing by at most two parallel steps from the lower bound obtained by the ramp-up strategy. Examining the ratio of the upper and lower bounds obtained by S_2, we see that for 26 out of the 30 instances it finds a plan whose makespan is (provably) at most 1.5 times optimal, and for all but one instance it obtains a plan whose makespan is at most two times optimal. In contrast, the ramp-up strategy does not find a feasible plan for 21 of the 30 instances. Thus on the `pathways` domain, the modified version of SatPlan using query strategy S_2 gives behavior that is in many ways better than that of the original.

To better understand the performance of S_2, we also compared it to a geometric query strategy S_g inspired by Algorithm B of Rintanen (2004). This query strategy behaves as follows. It initializes T to 1. If l and u are the initial lower and upper bounds, it then executes the queries $\langle k, T\gamma^{k-l}\rangle$ for each $k = \{l, l+1, \ldots, u-1\}$, where $\gamma \in (0,1)$ is a parameter. It then updates l and u, doubles T, and repeats. Based on the results of Rintanen (2004) we set $\gamma = 0.8$. We do not compare to Rintanen's Algorithm B directly because it requires many runs of the SAT solver to be performed in parallel, which requires an impractically large amount of memory for some benchmark instances.

The results for S_g are shown in the second column of Table 1. Like S_2, S_g always obtains upper bounds that are as good or better than those of the ramp-up strategy. Compared to S_2, S_g generally obtains slightly better lower bounds and slightly worse upper bounds.

Table 1: Performance of two query strategies on benchmark instances from the `pathways` domain of the 2006 International Planning Competition. Bold numbers indicate the (strictly) best upper/lower bound we obtained.

Inst.	**SatPlan (S_2)** [lower,upper]	**SatPlan (S_g)** [lower,upper]	**SatPlan (orig.)** [lower,upper]
p01	[5,5]	[5,5]	[5,5]
p02	[7,7]	[7,7]	[7,7]
p03	[8,8]	[8,8]	[8,8]
p04	[8,8]	[8,8]	[8,8]
p05	[9,9]	[9,9]	[9,9]
p06	[12,12]	[12,12]	[12,12]
p07	[13,13]	[13,13]	[13,13]
p08	[15,17]	[16,17]	[16,∞]
p09	[15,17]	[15,17]	[15,∞]
p10	[15,15]	[15,15]	[15,15]
p11	[16,17]	[16,17]	[16,∞]
p12	[16,19]	[17,19]	[17,∞]
p13	[16,18]	[17,18]	[17,∞]
p14	[14,20]	[15,**19**]	[15,∞]
p15	[18,18]	[18,18]	[18,18]
p16	[17,**21**]	[19,22]	[19,∞]
p17	[19,**21**]	[20,22]	[20,∞]
p18	[19,**22**]	[19,23]	[19,∞]
p19	[17,**22**]	[18,24]	[18,∞]
p20	[17,28]	[18,**27**]	[**19**,∞]
p21	[20,25]	[21,25]	[**22**,∞]
p22	[17,**23**]	[18,26]	[**19**,∞]
p23	[17,25]	[17,25]	[**18**,∞]
p24	[21,**27**]	[21,28]	[**22**,∞]
p25	[20,**27**]	[20,∞]	[**21**,∞]
p26	[19,**27**]	[20,31]	[**21**,∞]
p27	[19,34]	[20,**31**]	[20,∞]
p28	[19,**27**]	[20,∞]	[**21**,∞]
p29	[**19**,29]	[18,29]	[18,∞]
p30	[20,**60**]	[21,∞]	[21,∞]

Similar tables for the remaining six problem domains are available online at `http://www.cs.cmu.edu/~matts/icaps07/appendixA.pdf`. For the `storage`, `rovers`, and `trucks` domains, our results are similar to the ones presented in Table 1: S_2 achieved significantly better upper bounds than ramp-up and slightly worse lower bounds, while S_g achieved slightly better lower bounds than S_2 and slightly worse upper bounds. For the `openstacks`, `TPP`, and `pipesworld` domains, our results were qualitatively different: most instances in these domains were either easy enough that all three query strategies found a provably optimal plan, or so difficult that no strategy found a feasible plan, with the ramp-up strategy yielding the best lower bounds.

To gain more insight into these results, we plotted the function $\tau(k)$ for various instances. Broadly speaking, we encountered two types of behavior: either $\tau(k)$ increased as a function of k for $k < OPT$ but decreased as a function of k for $k \geq OPT$, or $\tau(k)$ increased as a func-

tion of k for all k. Figure 3 (A) and (B) give prototypical examples of these two behaviors. The gross behavior of τ on a particular instance was largely determined by the problem domain. For instances from the `pathways`, `storage`, `trucks`, and `rovers` domains τ tended to be increasing-then-decreasing, while for instances from the `TPP` and `pipesworld` domain τ tended to be monotonically increasing, explaining the qualitative difference between our results in these two sets of domains. For most instances in the `openstacks` domain we found no k values that elicited a "yes" answer in reasonable time; hence we cannot characterize the typical behavior of τ.

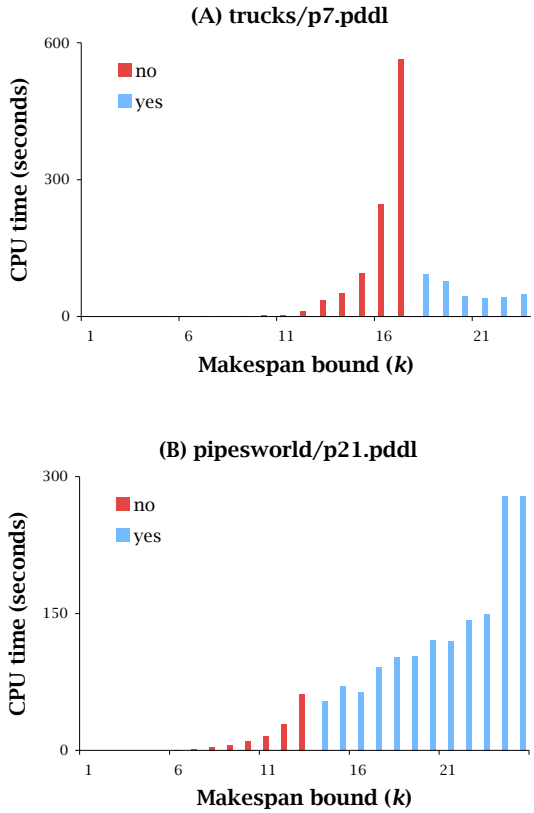

Figure 3: Behavior of the SAT solver `siege` running on formulae generated by SatPlan to solve (A) instance `p7` from the `trucks` domain and (B) instance `p21` from the `pipesworld` domain of the 2006 International Planning Competition.

Job shop scheduling

In this section, we use query strategy S_2 to create a modified version of a branch and bound algorithm for job shop scheduling. We chose the algorithm of Brucker et al. (1994) (henceforth referred to as `Brucker`) because it is one of the state-of-the-art branch and bound algorithms for job shop scheduling, and because code for it is freely available online.

Given a branch and bound algorithm, one can always create a decision procedure that answers the question "Does there exist a solution with cost at most k?" as follows: initialize the global upper bound to $k+1$, and run the algorithm until either a solution with cost $\leq k$ is discovered (in which case the result of the query is "yes") or the algorithm terminates without finding such a solution (in which case the result is "no"). Note that the decision procedure returns the correct answer independent of whether $k+1$ is a valid upper bound. A query strategy can be used in conjunction with this decision procedure to find optimal or approximately optimal solutions to the original minimization problem.

We evaluate two versions of `Brucker`: the original and a modified version that uses S_2. We ran both versions on the instances in the OR library (Beasley 1990) with a one hour time limit per instance, and recorded the upper and lower bounds obtained. We do not evaluate the ramp-up strategy or S_g in this context, because they were not intended to work well on problems such as job shop scheduling, where the number of possible k values is very large.

On 50 of the benchmark instances, both query strategies found a (provably) optimal solution within the time limit. Table 2 presents the results for the remaining instances. As in Table 1, bold numbers indicate an upper or lower bound that was strictly better than the one obtained by the competing algorithm. With the exception of just one instance (`la25`), the modified algorithm using query strategy S_2 obtains better lower bounds than the original branch and bound algorithm. This is not surprising, because the lower bound obtained by running the original branch and bound algorithm is simply the value obtained by solving the relaxed subproblem at the root node of the search tree, and is not updated as the search progresses. What is more surprising is that the upper bounds obtained by S_2 are also, in the majority of cases, substantially better than those obtained by the original algorithm. This indicates that the speculative upper bounds created by S_2's queries are effective in pruning away irrelevant regions of the search space and forcing the branch and bound algorithm to find low-cost schedules more quickly. These results are especially promising given that the technique used to obtain them is domain-independent and could be applied to other branch and bound algorithms. In related work, Streeter & Smith (2006) improved the performance of `Brucker` by using an iterated local search algorithm for job shop scheduling to obtain *valid* upper bounds and also to refine the branch ordering heuristic.

To better understand these results, we manually examined the function $\tau(k)$ for a number instances from the OR library. In all cases, we found that $\tau(k)$ increased smoothly up to a point and then rapidly decreased in a jagged fashion. Figure 4 illustrates this behavior. The smooth increase of $\tau(k)$ as a function of k for $k < OPT$ reflects the fact that proving that no schedule of makespan $\leq k$ exists becomes more difficult as k gets closer to OPT. The jaggedness of $\tau(k)$ for $k \geq OPT$ can be seen as an interaction between two factors: for $k \geq OPT$, increasing k leads to less pruning (increasing $\tau(k)$) but also to a weaker termination criterion (reducing it). In spite of this, the curve has low stretch overall, and thus its shape can be exploited by query strategies such as S_2.

Table 2: Performance of two query strategies on benchmark instances from the OR library. Bold numbers indicate the (strictly) best upper/lower bound we obtained.

Instance	**Brucker (S_2)** [lower,upper]	**Brucker (original)** [lower,upper]
abz7	[650,**712**]	[650,726]
abz8	[**622**,**725**]	[597,767]
abz9	[**644**,**728**]	[616,820]
ft20	[**1165**,**1165**]	[1164,1179]
la21	[**1038**,1070]	[995,**1057**]
la25	[971,979]	[**977**,**977**]
la26	[1218,1227]	[1218,**1218**]
la27	[1235,1270]	[1235,1270]
la28	[1216,**1221**]	[1216,1273]
la29	[**1118**,1228]	[1114,**1202**]
la38	[**1176**,1232]	[1077,**1228**]
la40	[**1211**,1243]	[1170,**1226**]
swv01	[**1391**,**1531**]	[1366,1588]
swv02	[1475,**1479**]	[1475,1719]
swv03	[**1373**,1629]	[1328,**1617**]
swv04	[**1410**,**1632**]	[1393,1734]
swv05	[**1414**,**1554**]	[1411,1733]
swv06	[**1572**,**1943**]	[1513,2043]
swv07	[**1432**,**1877**]	[1394,1932]
swv08	[**1614**,**2120**]	[1586,2307]
swv09	[1594,**1899**]	[1594,2013]
swv10	[**1603**,**2096**]	[1560,2104]
swv11	[2983,**3407**]	[2983,3731]
swv12	[**2971**,**3455**]	[2955,3565]
swv13	[3104,**3503**]	[3104,3893]
swv14	[2968,**3350**]	[2968,3487]
swv15	[2885,**3279**]	[2885,3583]
yn1	[**813**,**987**]	[763,992]
yn2	[**835**,**1004**]	[795,1037]
yn3	[**812**,**982**]	[793,1013]
yn4	[**899**,**1158**]	[871,1178]

Conclusions

Optimization problems are often solved using an algorithm for the corresponding decision problem as a subroutine. In this paper, we considered the problem of choosing which queries to submit to the decision procedure so as to obtain an (approximately) optimal solution as quickly as possible. Our main contribution was a new query strategy S_2 that has attractive theoretical guarantees and appears to perform well in practice. Experimentally, we showed that S_2 can be used to create improved versions of state-of-the-art algorithms for planning and job shop scheduling. An interesting direction for future work would be to apply S_2 in other domains where branch and bound algorithms work well, for example integer programming or resource-constrained project scheduling.

References

Auer, P.; Cesa-Bianchi, N.; Freund, Y.; and Schapire, R. E. 2002. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing* 32(1):48–77.

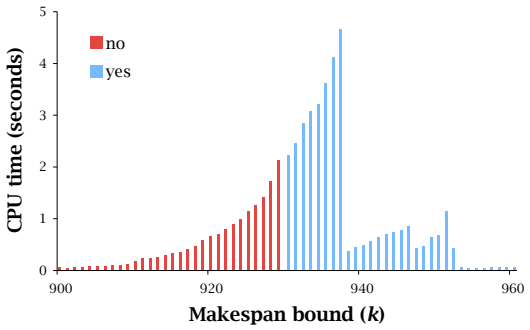

Figure 4: Behavior of `Brucker` running on OR library instance `ft10`.

Beasley, J. E. 1990. OR-library: Distributing test problems by electronic mail. *Journal of the Operational Research Society* 41(11):1069–1072.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Brucker, P.; Jurisch, B.; and Sievers, B. 1994. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics* 49(1-3):107–127.

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.

Hsu, C.-W.; Wah, B. W.; Huang, R.; and Chen, Y. 2006. New features in SGPlan for handling preferences and constraints in PDDL3.0. In *Proceedings of the Fifth International Planning Competition*, 39–42.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SATPLAN: Planning as satisfiability. In *Proceedings of the Fifth International Planning Competition*.

Knuth, D. E. 1971. Optimum binary search trees. *Acta Informatica* 1:14–25.

Korf, R. E. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Rintanen, J. 2004. Evaluation strategies for planning as satisfiability. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence*, 682–687.

Streeter, M. J., and Smith, S. F. 2006. Exploiting the power of local search in a branch and bound algorithm for job shop scheduling. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, 324–332.

Xing, Z.; Chen, Y.; and Zhang, W. 2006. MaxPlan: Optimal planning by decomposed satisfiability and backward reduction. In *Proceedings of the Fifth International Planning Competition*, 53–56.