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

Introduction

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?". 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. 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., one plus the largest k that elicited a "no" response) 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 and MaxPlan. 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).

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. On this instance, the ramp-up query strategy

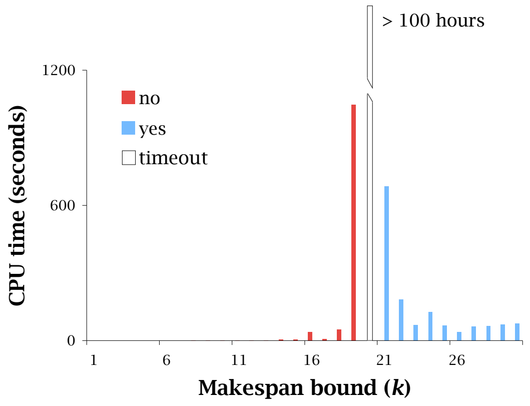

Figure 1: CPU time required by `siege` as a function of k (instance `p17` from the `pathways` domain).

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. 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. For planning problems where provably optimal plans are currently out of reach, obtaining provably approximately optimal plans quickly is a natural goal.

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 found that Algorithm B could yield dramatic performance improvements relative to the ramp-up strategy.

Preliminaries

Let $\tau(k)$ denote the time required by the decision procedure on input k. 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 Figure 1), and our query strategy was designed to take advantage of such behavior. More precisely, our query strategy is 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). 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 $k > x$.

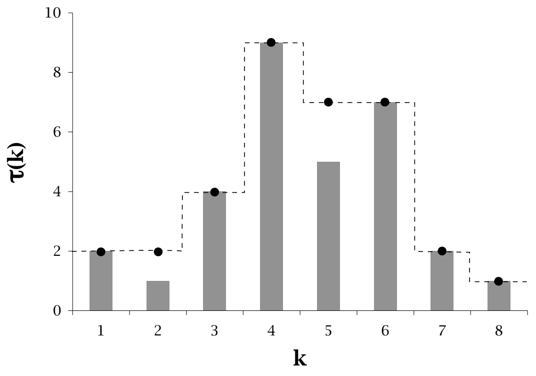

Figure 2: A function τ (gray bars) and its hull (dots).

Definition (stretch). *The* stretch *of τ is the quantity*

$$\Delta = \max_k \frac{\mathrm{hull}^\tau(k)}{\tau(k)} .$$

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

A New Query Strategy

We now describe a new query strategy S_2 designed to work well when Δ is low. 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).

Pseudocode for S_2 follows. Here U denotes an initial upper bound obtained using heuristics.

```
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 $t_u \leftarrow \max\{t_u, k\}$.
```

We now analyze S_2. 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 $\tau(k) > \frac{T}{\Delta}$ (by definition of stretch). Now consider some arbitrary k. Once $T \geq \Delta\tau(k)$ 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. Furthermore, there can be at most $O(\log U)$ queries in between updates to T (because the number of eligible k-values decreases exponentially). It follows that for any k, we have to wait $O(\Delta\tau(k) \cdot \log U)$ time before $k \notin [l, u-1]$. This implies the following theorem.

Theorem 1. *Suppose there exists a query strategy that obtains a lower bound l and upper bound u after running for a total of T time steps. Then, after running for $T \cdot O(\Delta \log U)$ time steps, S_2 will obtain a lower bound l' and an upper bound u' such that $l' \geq l$ and $u' \leq u$.*

In particular, Theorem 1 implies that if some query strategy achieves $u \leq \alpha l$ in time T (for some desired approximation ratio α), then S_2 will achieve a ratio as good or better in time $T \cdot O(\Delta \log U)$. This result is tight in that any query strategy S' must run for time $T \cdot \Omega(\Delta \log U)$ in order to make the same guarantee (we omit the proof of this fact).

Experimental Evaluation

In this section we evaluate S_2 experimentally by using it to create modified versions of state-of-the-art solvers in two domains: job shop scheduling and STRIPS planning.

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 use the algorithm of Brucker et al. (1994), henceforth referred to as `Brucker`.

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”). A query strategy can be used in conjunction with this decision procedure to solve 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 with a one hour time limit per instance, and recorded the upper and lower bounds obtained. On 50 of the benchmark instances, both query strategies found a (provably) optimal solution within the time limit. Table 1 presents the results for the remaining instances. 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.

Table 1: Performance of two query strategies on benchmark instances from the OR library.

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]

STRIPS Planning

Our STRIPS planning experiments were run on the benchmarks from the 2006 International Planning Competition. In what follows we summarize our results for the 30 benchmark instances from the `pathways` domain. More detailed results can be found in the full version of our paper.

We compared S_2 to the ramp-up query strategy, as used by SatPlan. On these 30 instances, S_2 always obtained upper bounds that are as good or better than those obtained by the ramp-up strategy. The lower bounds obtained by S_2 were only slightly worse, differing by at most two parallel steps from the lower bound obtained by the ramp-up strategy. S_2 always found a feasible plan, and for 26 out of the 30 instances, it found a plan whose makespan is (provably) at most 1.5 times optimal. In contrast, the ramp-up strategy did not even find a feasible plan for 21 of the 30 instances.

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 did 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. Like S_2, S_g always obtained 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.

References

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.

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 (ECAI’2004)*, 682–687.