Evaluation Strategies for Planning as Satisfiability

Jussi Rintanen[1]

Abstract. We investigate different evaluation strategies for planning problems represented as constraint satisfaction or satisfiability problems. The standard evaluation strategy, evaluating the formulae by sequentially increasing the length one step at a time, guarantees that a plan corresponding to the first satisfiable formula is found first, yet this is often not the best possible strategy in terms of runtime. We present evaluation strategies based on parallel or interleaved evaluation of several formulae and show that with many problems this leads to substantially improved runtimes, sometimes several orders of magnitude. The cost of the improved runtimes is a possible decline in plan quality because an optimality guarantee of the standard evaluation strategy is lost.

1 INTRODUCTION

In this work we report new developments in satisfiability planning emerging from a paradigm shift in evaluation strategies together with important qualitative and quantitative improvements in satisfiability algorithms that have been taking place in the last years, leading to dramatically improved efficiency of satisfiability planning.

Earlier research on classical planning that split plan search into finding plans of given fixed lengths, for instance the Graphplan algorithm [1], planning as satisfiability [4], and related approaches [7, 6, 11, 10, 3], have without exception adopted a sequential strategy. This strategy starts with (parallel) plan length 1, and if no such plans exist, continues with length 2, length 3, and so on, until a plan is found. When every time step consists of exactly one operator, the standard sequential strategy is guaranteed to find a plan that is optimal with respect to plan length.

It seems that when we want to preserve this sequential optimality property, the sequential strategy cannot in general be substantially improved. For example, a strategy that increases the plan length by more than one until a satisfiable formula is found and then performs a binary search to find the shortest plan does not typically improve runtimes because the cost of evaluating the unsatisfiable formulae usually increases exponentially as the plan length increases.

However, when we want to find a plan of any quality, or when the sequential optimality criterion loses its meaning because one time step is allowed to contain several operators, we can use strategies that take the exponentially growing cost of the unsatisfiable formulae and the possibly much lower cost of the first satisfiable formulae into account. The evaluation costs typically follow the pattern in Figure 1 (except that sometimes the first satisfiable formulae are not cheaper than the preceding unsatisfiable ones, and the first satisfiable formula may not be the least expensive of the satisfiable ones.)

This suggests a strategy that interleaves the evaluation of several formulae. We want to detect the satisfiability of one of the formulae corresponding to a plan before having spent too much time determining the unsatisfiability of earlier formulae. A strategy implemented by our first algorithm distributes the computation to n concurrent processes and initially assigns the first n formulae to the n processes. Whenever a process finds its formula satisfiable, the computation is terminated. Whenever a process finds its formula unsatisfiable, the process is given the first unevaluated formula to evaluate. As is shown by Figure 1, this strategy can avoid completing the evaluation of many of the expensive unsatisfiable formulae, thereby saving a lot of computation effort.

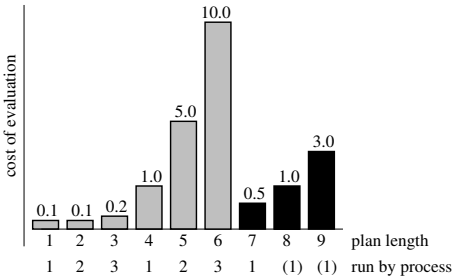

Figure 1. Evaluation cost of the unsatisfiable formulae for plan lengths 1 to 6 and the satisfiable formulae for plan length 7 and higher. With 3 processes, process 1 finds the first plan (satisfying assignment) after evaluating the formulae for plan lengths 1, 4 and 7 in 0.1+1+0.5 = 1.6 seconds. This is $3 \times 1.6 = 4.8$ seconds of total CPU time. The sequential strategy needs $0.1 + 0.1 + 0.2 + 1 + 5 + 10 + 0.5 = 16.9$ seconds. With 4 processes the plan would be found by process 3 in $0.2 + 0.5 = 0.7$ seconds of CPU time, which is $4 \times 0.7 = 2.8$ seconds of total CPU time.

It is surprising that these simple and – as it turns out – extremely effective strategies were not previously proposed for controlling a planner. The effectiveness of these strategies follows from the profile of runtimes for formulae for different parallel plan lengths. This profile, it seems, previously received little attention and the effectiveness of the strategies we present went unnoticed. The new evaluation strategies extend the applicability of a wide range of CSP and SAT based planning technologies to much more difficult problems. The disadvantage of the strategies is that the optimality of parallel plan lengths is lost, but, as optimal parallel plan length does not imply optimal (sequential) plan length, optimality is often not a reason for using the sequential strategy.

The structure of the paper is as follows. In Section 2 we formalize the above ideas in two algorithms that can be used as the high-level evaluation algorithm of a planner. Then we analytically investigate their computational properties in Section 3, and test them on commonly used planning benchmarks in Section 4. In Section 5 we discuss the meaning of the results for classical planning in general. Finally, in Section 6 we consider further research topics.

[1] Albert-Ludwigs-Universität Freiburg, Institut für Informatik, Georges-Köhler-Allee, 79110 Freiburg im Breisgau, Germany

2 EVALUATION ALGORITHMS

We propose two algorithms for evaluation of a sequence $\phi_0, \phi_1, \phi_2, \ldots$ of SAT/CSP problems that has the property that for some unknown j, all formulae ϕ_i with $i < j$ are unsatisfiable, and all formulae ϕ_k with $k \geq j$ are satisfiable. The goal is to find one of the first satisfiable formulae as fast as possible. An inherent property of the problem is that unsatisfiable (resp. satisfiable) formulae later in the sequence are in general more expensive to evaluate than earlier unsatisfiable (resp. satisfiable) formulae. The difficulty of the unsatisfiable formulae increases as i increases because the formulae become less constrained, contradictions are not found as quickly, and search trees grow exponentially. The exponential increase in difficulty of satisfiable formulae is less clear. For example, for the first satisfiable formula ϕ_s there may be few plans while for later formulae there may be many plans, and the formulae would be less constrained and easier to evaluate. However, as formula sizes increase, the possibility of in getting lost in parts of the search space that do not contain any solutions also increases. Therefore increase in plan length also later leads to an increase in difficulty.

The new algorithms are useful if a peak of difficult formulae precedes some of the easier satisfiable formulae. For example, when it is easier to find a plan of length n than to prove that no plans of length $n-1$ exists, and, if the first strongly constrained satisfiable formulae corresponding to the shortest plans are more difficult to evaluate than some of the later less constrained ones. The experiments show that with many problems one or both of these conditions hold.

We call the sequential evaluation algorithm Algorithm S. It first tests whether ϕ_0 is satisfiable, and if not, repeatedly proceeds with the next formula until a satisfiable formula is found. If the formulae ϕ_i encode the sets of plans of length i, then the sequential algorithm is guaranteed to find one of the shortest plans. As we will see, the sequential algorithm may have a much higher runtime than the non-sequential algorithms we propose. The new algorithms enable trading plan quality guarantees to better runtimes.

The first algorithm is based on dividing the evaluation tasks of different formulae to a fixed number n of different processes, each of which determines the satisfiability or unsatisfiability of one of the formulae, after which it is assigned a new formula to evaluate.

The second algorithm is based on evaluating an unbounded number of formulae in an interleaved manner. Most of the CPU is dedicated to evaluating the formulae ϕ_i with the smallest indices i, according to a parameter γ. This parameter plays a role similar to the n in the first algorithm. Because CPU time is a discrete resource, real implementations of this scheme are actively evaluating only a small finite number of formulae at any given moment.

2.1 Algorithm A: multiple processes

The first algorithm is based on parallel or interleaved evaluation of a fixed number n of formulae by n processes. As the special case $n = 1$ we have Algorithm S. Whenever a process finishes the evaluation of a formula, it is given the first unevaluated formula to evaluate. The algorithm is given in Figure 2.

There is a simple improvement to the algorithm: when formula ϕ_i is found unsatisfiable, the algorithm terminates the evaluation of all ϕ_j for $j < i$ because they must all be unsatisfiable. However, this modification does not usually have any effect because of the monotonically increasing evaluation cost of the unsatisfiable formulae: ϕ_j would already have been evaluated when ϕ_i with $i > j$ is found unsatisfiable. We ignore this improvement in the following.

PROCEDURE AlgorithmA(n)
P := $\{\phi_0, \ldots, \phi_{n-1}\}$;
uneval := n;
found := false;
REPEAT
  P' := P;
  FOR EACH $\phi \in P'$ *DO*
    continue evaluation of ϕ for ϵ seconds;
    IF evaluation of ϕ terminated *THEN*
      $P := P \cup \{\phi_{\text{uneval}}\} \backslash \{\phi\}$;
      uneval := uneval + 1;
      IF ϕ is satisfiable *THEN* found := true; *END IF*
    END IF
  END DO
UNTIL found

Figure 2. Algorithm A

2.2 Algorithm B: geometric division of CPU use

In Algorithm A the choice of n is determined by the (assumed) width and height of the peak preceding the first satisfiable formulae, and our experiments indicate that small differences in the choice of n may make a substantial difference in the runtimes. Our second algorithm addresses the difficulty of choosing the value n in Algorithm A. Our Algorithm B evaluates in an interleaved manner an unbounded number of formulae. The amount of CPU given to each formula depends on its index: if formula ϕ_k is given t seconds of CPU during a certain time interval, then a formula $\phi_i, i \geq k$ is given $\gamma^{i-k}t$ seconds. This means that every formula gets only slightly less CPU than its predecessor, and the choice of the exact value of the constant $\gamma \in]0, 1[$ is far less critical than the choice of n for Algorithm A.

Algorithm B is given in Figure 3. Variable t which is incrementally

PROCEDURE AlgorithmB(γ)
t := 0;
found := false;
FOR EACH $i \geq 0$ *DO* done[i] = false;
FOR EACH $i \geq 0$ *DO* time[i] = 0;
REPEAT
  $t := t + \delta$;
  FOR EACH $i \geq 0$ such that done[i] = false *DO*
    IF time[i] + $n\epsilon \leq t\gamma^i$ for some maximal $n \geq 1$ *THEN*
      continue evaluation of ϕ_i for $n\epsilon$ seconds;
      time[i] := time[i] + $n\epsilon$;
      IF evaluation of ϕ_i terminated *THEN* done[i] := true; *END IF*
      IF ϕ_i was found satisfiable *THEN* found := true; *END IF*
    END IF
  END DO
UNTIL found

Figure 3. Algorithm B

increased by δ characterizes the total CPU time $\frac{t}{1-\gamma}$ available so far. Because the evaluation of ϕ_i proceeds only if it has been evaluated for at most $t\gamma^i - \epsilon$ seconds, CPU is actually consumed less than $\frac{t}{1-\gamma}$, and there will be at time $\frac{t}{1-\gamma}$ only a finite number $j \leq \log_\gamma \frac{\epsilon}{t}$ of formulae for which evaluation has commenced.

In a practical implementation of the algorithm, the rate of increase δ of t is increased as the computation proceeds; otherwise the inner

foreach loop will later often be executed without evaluating any of the formulae further. We could choose δ for example so that the first unfinished formula ϕ_i is evaluated further at every iteration ($\delta = \frac{\epsilon}{\gamma^i}$).

3 PROPERTIES OF THE ALGORITHMS

In this section we analyze the properties of the algorithms.

Definition 1 (Speed-up) *The* speed-up *of an algorithm X (with respect to Algorithm S) is the ratio of the runtimes of Algorithm S and the Algorithm X.*

If the speed-up is greater than 1, then the algorithm is faster than Algorithm S.

In our analysis we assume that the constant ϵ in Algorithm A is infinitesimally small, and hence, after a process finishes with one formula, the evaluation of the next formula starts immediately, and the algorithm terminates immediately after a satisfiable formula is found.

If there is no peak, that is, the last unsatisfiable formulae are not more difficult than some of the first satisfiable ones, then Algorithm A with $n \geq 2$ may need n times more CPU than Algorithm S, because $n-1$ satisfiable formulae are evaluated unnecessarily. We formally establish worst-case bounds for Algorithm A.

Theorem 2 *The speed-up of Algorithm A with n processes is at least $\frac{1}{n}$. This lower bound is strict.*

Proof: The worst case $\frac{1}{n}$ can show up in the following situation. Assume the first satisfiable formula is evaluated in time t, the preceding unsatisfiable formulae are evaluated in time 0, and the following satisfiable formulae are evaluated in time $\geq t$. Then the total runtime of Algorithm A is tn, while the total runtime of Algorithm S is t.

Assume the runtimes (CPU time) of the formulae are $t_0, t_1, \ldots, t_s, \ldots$, and ϕ_s is the first satisfiable formula. The total runtime of Algorithm S is $\sum_{i=0}^{s} t_i$. This is also an upper bound on the CPU time consumed by Algorithm A on $\phi_0, \ldots, \phi_s$. Additionally, Algorithm A may spend CPU evaluating $\phi_{s+1}, \phi_{s+2}, \ldots$. The evaluation of these formulae starts at the same time or later than the evaluation of the first satisfiable formula ϕ_s. Because $n-1$ processes may spend all their time evaluating these formulae after the evaluation of ϕ_s has started, the total CPU time spent evaluating them may be at most $(n-1)t_s$. Hence Algorithm A spends CPU time at most

$$\sum_{i=0}^{s} t_i + (n-1)t_s$$

in comparison to

$$\sum_{i=0}^{s} t_i$$

with Algorithm S. The speed-up is therefore at least

$$\frac{\sum_{i=0}^{s} t_i}{\sum_{i=0}^{s} t_i + (n-1)t_s} = \frac{1}{1 + (n-1)\frac{t_s}{\sum_{i=0}^{s} t_i}} \geq \frac{1}{1+n-1} = \frac{1}{n}$$

□

In the other direction, there is no finite upper bound on the speed-up of Algorithm A in comparison to Algorithm S for any number of processes $n \geq 2$. Consider a problem instance with evaluation time t_0, t_1 and t_2 respectively for the first three formulae, the first two of which are unsatisfiable and the third satisfiable. Let $t_0 = t_2$ and $t_1 = ct_2$. The constant c could be arbitrarily high. Algorithm S runs in $(c+2)t_2$ time, while Algorithm A with $n = 2$ runs in $2t_2$ time. Hence the speed-up $\frac{c+2}{2}$ can be arbitrarily high.

Next we analyze the properties of Algorithm B assuming that the constants δ and ϵ are infinitesimally small, that is, the evaluation of all of the formulae ϕ_i proceeds continuously at rate γ^i.

The constants n and γ respectively for Algorithms A and B are roughly related by $\gamma = 1 - \frac{1}{n}$: of the CPU capacity $\frac{1}{n} = 1 - \gamma$ is spent evaluating the first unfinished formula, and the lower bound for Algorithm B is similarly related to the lower bound for Algorithm A. Algorithm S is the limit of Algorithm B when γ goes to 0.

Theorem 3 *The speed-up of Algorithm B is at least $1-\gamma$. This lower bound is strict.*

Proof: As with Algorithm A the worst case is achieved when all unsatisfiable formulae preceding the first satisfiable formula ϕ_s are evaluated and, additionally, the evaluation of many of the satisfiable ones has proceeded far. The disadvantage in comparison to Algorithm S is the unnecessary evaluation of many of the satisfiable formulae. Hence Algorithm B spends CPU time at most

$$\sum_{i=0}^{s} t_i + \sum_{i \geq 1} t_s \gamma^i = \sum_{i=0}^{s} t_i + \frac{1}{1-\gamma} t_s - t_s$$

in comparison to

$$\sum_{i=0}^{s} t_i$$

with Algorithm S. The speed-up is therefore at least

$$\begin{aligned} \frac{\sum_{i=0}^{s} t_i}{\sum_{i=0}^{s} t_i + \frac{1}{1-\gamma} t_s - t_s} &= \frac{1}{1 + \frac{\frac{1}{1-\gamma} t_s - t_s}{\sum_{i=0}^{s} t_i}} \geq \frac{1}{1 + \frac{\frac{1}{1-\gamma} t_s - t_s}{t_s}} \\ &= \frac{1}{1 + \frac{1}{1-\gamma} - 1} = 1 - \gamma. \end{aligned}$$

This lower bound is strict: if ϕ_i is satisfiable, evaluation times for $\phi_j, j < i$ are 0, and evaluation times for $\phi_i, i > 1$ are not lower than that of ϕ_1, then the speed-up is only $1 - \gamma$. □

4 EMPIRICAL EVALUATION

We illustrate properties of the new evaluation algorithms on a collection of problems from the AIPS planning competitions. Plans for most of these problems can be found in polynomial time by very simple domain-specific algorithms, and planners using heuristic search [2] have excelled on these problems, while they had been considered difficult for planners based on satisfiability testing or CSP techniques. On other types of problems satisfiability planning has had an advantage over heuristic planners.

Each problem instance is mapped to a sequence of propositional formulae according to a novel translation we have developed [9]. The new translation often improves the Graphplan-based translation of Kautz and Selman [5] by one or two orders of magnitude.

In running the experiments we use the Siege V4 SAT solver by Lawrence Ryan from the Simon Fraser University on a 3.6 GHz Intel Xeon computer to test the satisfiability of formulae. Then we compute from these runtimes of individual formulae the total runtime under algorithms A and B with different values for the parameters n and γ. Algorithm S is the special case $n = 1$ of Algorithm A. The constants ϵ and δ determining the granularity of CPU time division are set infinitesimally small. For each problem instance we produced

at least 5 or 10 formulae beyond the first (assumed) satisfiable formula, and the formulae after that are assumed to have infinite cost, as are formulae which take over 20 minutes to evaluate. The times do not include generation of the formulae.

The runtimes on a number of problems from the AIPS planning competitions of 1998, 2000 and 2002 are given in Table 1. For most benchmarks we give the runtimes of the most difficult problems, which in some cases are the last ones in the series. A big fraction of the runtimes not given are below one second for any evaluation strategy. Some of the benchmark series cannot be efficiently solved until the end, and we give data just some of the most difficult instances that can be solved. We discuss these benchmarks below.

The Movie, MPrime and Mystery benchmarks from the 1998 competition and Rovers from 2002 are very easy for every evaluation strategy (fraction of a second in most cases) but we cannot produce the biggest MPrime instance because of a memory restriction.

The Logistics (1998 and 2000) and Satellite (2002) series are solved completely. Proving inexistence of plans slightly shorter than the optimal plan length is in some cases difficult but the new evaluation algorithms handle this efficiently.

The Depots (2002) problems are also relatively easy but in contrast to most other benchmarks the new evaluation algorithms in some cases increase runtimes substantially.

The DriverLog and ZenoTravel (2002) problems are solved quickly except for some of the biggest instances. We cannot find satisfiable formulae for the last two ZenoTravel problems within our time limit, and finding plans for the preceding two instances of ZenoTravel and the second last of DriverLog is also slow. The last DriverLog instance cannot be solved with Siege V4 because of its restriction to 524288 propositions.

Blocks World (2000) is basically easy to solve for the newest generation of SAT solvers but the high plan lengths lead to very big formulae (size over 100 MB and over 524288 propositions), and we can solve only two thirds of the series.

Elevator (2000), Schedule (2000) and Gripper (1998) are a challenge because only very loose lower bounds on plan length are easy to prove. Finding plans corresponding to a given satisfiable formula is very easy (some seconds at most) but locating these formulae is very expensive. Increasing parameters n and γ improves runtimes.

The formulae generated for FreeCell (2002) are too big (hundreds of megabytes) for the current SAT solvers to solve them efficiently.

Because for most of these benchmarks there is a high number of easy satisfiable formulae corresponding to non-optimal plans, plans could also be found by evaluating only every fifth or tenth formula, thereby cutting the runtimes to a small fraction of the ones presented. However, we do not think that this is a good strategy in general.

All in all, it seems that a conservative use of the new algorithms (especially Algorithm B with $\gamma \in [0.7..0.9]$) leads to a general improvement in the runtimes in comparison to Algorithm S.

Plan quality sometimes decreases as total runtime decreases, but not always. The biggest decrease is on the biggest Logistics instances, with number of operators about 35 per cent higher for the more efficient evaluation strategies. In some cases longer parallel plan length yields slightly better plans in terms of the number of operators. Interestingly, for the Gripper problem the easy satisfiable formulae corresponding to plans much above optimum parallel length still yield very good plans with very few unnecessary operators.

5 DISCUSSION

For many of the above problems the only domain-independent planners that have earlier quickly solved them are based on heuristic search as proposed by Bonet and Geffner [2] and feature techniques specifically targeting these benchmarks. Our results now indicate that general-purpose techniques suffice and not even a planning-specific heuristic is needed. Sophisticated satisfiability algorithms easily find plans provided that the problem instance combined with a fixed plan length does not result in a too tightly constrained satisfiability problem. Such loosely constrained satisfiability problems can often be effectively found by simultaneously evaluating formulae corresponding to several plan lengths, a task for which we proposed the Algorithms A and B.

Until now, satisfiability planning had been considered a leading approach for solving different types of inherently difficult problems, for example difficult problems in the phase transition region [8] and optimal planning for problems that are easy to solve by heuristic search [2] when no optimality is required. Now our results show that, in fact, satisfiability planning is also rather strong in solving planning problems without the optimality requirement. The possible misperception of the strength of satisfiability planning has largely been a result of the use of the sequential strategy (Algorithm S). Of course, important factors in the positive results of the present paper (when compared to other types of algorithms) are also improvements in the translation of planning into the propositional logic and improvements in satisfiability algorithms in recent years, especially the emergence of very efficient satisfiability solvers based on sophisticated clause learning.

6 CONCLUSIONS

We have devised novel algorithms for managing the evaluation of a sequence of subproblems of finding plans of given lengths, as in satisfiability planning and the GraphPlan algorithm. The algorithms interleave the evaluation of several formulae, and can avoid completing the evaluation of some of the very difficult unsatisfiable formulae that would otherwise be responsible for very high runtimes. The cost of improved runtimes is that we no more have a guarantee that plans have optimal length.

The question arises whether the evaluation algorithms could be claimed to be optimal under a suitable criterion. This in an important topic for further research.

ACKNOWLEDGEMENTS

This work was partly funded by DFG grant RI 1177/2-1.

REFERENCES

[1] Avrim L. Blum and Merrick L. Furst, 'Fast planning through planning graph analysis', *Artificial Intelligence*, **90**(1-2), 281–300, (1997).
[2] Blai Bonet and Héctor Geffner, 'Planning as heuristic search', *Artificial Intelligence*, **129**(1-2), 5–33, (2001).
[3] Minh Binh Do and Subbarao Kambhampati, 'Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP', *Artificial Intelligence*, **132**(2), 151–182, (2001).
[4] Henry Kautz and Bart Selman, 'Pushing the envelope: planning, propositional logic, and stochastic search', in *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pp. 1194–1201, Menlo Park, California, (August 1996). AAAI Press.

	Algorithm A with n					Algorithm B with γ			
instance	1	2	4	8	16	0.500	0.750	0.875	0.938
log-39-0	-	-	54.2	8.7	5.4	136.4	17.2	9.5	10.1
log-39-1	-	564.9	84.2	15.6	5.3	86.2	11.6	7.8	8.9
log-40-0	1279.0	732.8	86.7	10.6	5.1	83.8	11.5	7.5	8.7
log-40-1	-	-	59.9	42.7	8.3	206.3	29.5	15.6	15.7
log-41-0	-	-	375.0	4.6	8.6	70.9	13.9	11.1	13.7
log-41-1	-	-	138.3	18.8	7.7	219.2	26.0	14.2	14.5
satell-18	0.2	0.2	0.2	0.3	0.6	0.2	0.3	0.4	0.8
satell-19	71.1	16.6	17.4	16.0	10.9	26.1	21.4	18.3	19.6
satell-20	-	-	67.1	42.4	29.6	124.0	80.1	79.3	70.1
depot-12-9876	148.1	295.2	2.2	2.4	4.6	3.2	2.9	3.9	6.0
depot-15-4534	63.8	124.3	245.4	486.6	972.2	124.6	246.1	489.1	975.1
depot-20-7615	51.2	100.9	3.1	3.0	5.9	6.8	4.5	5.4	8.1
depot-21-8715	0.3	0.4	0.7	1.3	2.6	0.5	0.9	1.7	3.0
depot-22-1817	174.9	346.6	689.8	1379.4	2758.7	347.3	692.1	1381.8	2761.2
driver-4-4-8	0.3	0.3	0.4	0.8	1.4	0.4	0.6	0.9	1.6
driver-5-5-10	805.4	426.6	393.7	146.6	249.3	754.0	304.0	284.4	376.4
driver-5-5-15	83.1	67.2	77.3	109.9	219.7	111.1	136.5	170.3	272.9
driver-5-5-20	667.1	301.7	50.9	99.3	198.4	103.8	92.7	134.1	230.3
driver-5-5-25	-	-	-	-	5445.3	$> 27h$	24641.5	10817.7	10851.0
zeno-5-10	0.4	0.6	0.7	1.0	1.6	0.7	1.2	1.8	2.6
zeno-5-15	166.2	101.1	121.3	3.4	6.7	27.2	7.4	6.9	9.6
zeno-5-15b	76.6	23.0	29.0	56.9	8.0	45.6	26.8	13.5	14.5
zeno-5-20	-	-	2273.2	4499.5	8999.0	9005.0	5338.7	6722.4	10927.4
zeno-5-20b	-	-	-	3399.9	6799.7	14173.6	5946.8	6374.2	9376.7
blocks-22-0	150.1	136.3	160.7	134.4	41.8	163.0	99.9	53.4	40.9
blocks-26-0	-	-	3970.2	2804.8	652.6	4100.6	1919.6	547.1	243.0
blocks-30-0	-	-	-	-	1519.5	22777.6	3573.0	1462.2	900.2
blocks-34-0	219.4	216.5	210.8	209.4	181.0	231.0	238.5	246.3	236.4
gripper-6	-	-	1196.6	40.2	1.3	28.6	4.7	2.3	2.3
gripper-7	-	-	-	181.6	15.5	1600.4	82.6	10.8	3.8
gripper-8	-	-	-	3077.5	73.9	9786.4	393.0	42.1	17.5
gripper-9	-	-	-	-	6089.0	$> 27h$	2999.7	117.9	26.6
gripper-10	-	-	-	-	-	$> 27h$	12027.4	183.3	34.7
gripper-11	-	-	-	-	-	$> 27h$	3712.5	55.1	9.4
gripper-12	-	-	-	-	-	$> 27h$	43813.2	198.9	19.4
gripper-13	-	-	-	-	-	$> 27h$	$> 27h$	761.4	119.6
sched-50-0	140.1	139.0	40.8	7.6	7.0	27.8	14.5	13.5	14.8
sched-50-1	-	-	-	-	3552.5	$> 27h$	4813.1	664.0	358.7
sched-50-2	-	132.9	53.9	28.4	18.4	104.3	35.1	27.5	32.4
sched-51-0	-	-	-	-	380.8	$> 27h$	2768.4	389.3	212.9
sched-51-1	-	-	-	775.1	350.4	30011.7	1033.0	209.6	144.5
sched-51-2	-	-	-	-	264.8	$> 27h$	4236.0	825.8	605.7
miconic/str-f32-p16-...-r0	-	-	-	4260.2	124.8	7903.6	593.1	151.9	110.4
miconic/str-f36-p18-...-r0	-	-	-	-	2193.8	$> 27h$	4773.9	457.8	211.0
miconic/str-f40-p20-...-r0	-	-	-	-	298.6	$> 27h$	1874.1	502.3	390.2
miconic/str-f44-p22-...-r0	-	-	-	-	-	$> 27h$	$> 27h$	6636.9	905.0
miconic/str-f48-p24-...-r0	-	-	-	-	-	$> 27h$	$> 27h$	12051.5	1729.7
miconic/str-f52-p26-...-r0	-	-	-	-	-	$> 27h$	$> 27h$	10108.8	2071.3
miconic/str-f56-p28-...-r0	-	-	-	-	-	$> 27h$	$> 27h$	88019.9	4796.1
miconic/str-f60-p30-...-r0	-	-	-	-	-	$> 27h$	$> 27h$	72980.6	5313.0
freecell4-4	0.1	0.2	0.2	0.2	0.3	0.2	0.2	0.3	0.4
freecell5-4	139.8	154.5	38.2	2.1	4.0	11.7	5.5	4.6	6.0
freecell6-4	499.5	455.2	554.2	120.7	78.9	629.2	159.9	112.7	134.2
freecell7-4	143.0	245.2	178.6	121.8	137.6	258.1	172.7	161.9	216.6

Table 1. Column $n = 1$ is Algorithm S. Dash indicates a missing upper bound on the runtime when some formulae were not evaluated in 20 minutes.

[5] Henry Kautz and Bart Selman, 'Unifying SAT-based and graph-based planning', in *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, ed., Thomas Dean, pp. 318–325. Morgan Kaufmann Publishers, (1999).

[6] Henry Kautz and Joachim Walser, 'State-space planning by integer optimization', in *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99) and the Eleventh Conference on Innovative Applications of Artificial Intelligence (IAAI-99)*, pp. 526–533. AAAI Press, (1999).

[7] Jussi Rintanen, 'A planning algorithm not based on directional search', in *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*, eds., A. G. Cohn, L. K. Schubert, and S. C. Shapiro, pp. 617–624. Morgan Kaufmann Publishers, (June 1998).

[8] Jussi Rintanen, 'Phase transitions in classical planning: an experimental study', in *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR 2004)*, eds., Didier Dubois and Christopher A. Welty. AAAI Press, (2004). to appear.

[9] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä, 'Parallel encodings of classical planning as satisfiability', Report 198, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, (2004).

[10] Peter van Beek and Xinguang Chen, 'CPlan: A constraint programming approach to planning', in *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99) and the Eleventh Conference on Innovative Applications of Artificial Intelligence (IAAI-99)*, pp. 585–590. AAAI Press, (1999).

[11] Steven A. Wolfman and Daniel S. Weld, 'The LPSAT engine & its application to resource planning', in *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, ed., Thomas Dean, volume I, pp. 310–315. Morgan Kaufmann Publishers, (1999).