Heuristic Planning with SAT: Beyond Uninformed Depth-First Search

Jussi Rintanen

NICTA and the Australian National University
Canberra, Australia

Abstract. Planning-specific heuristics for SAT have recently been shown to produce planners that match best earlier ones that use other search methods, including the until now dominant heuristic state-space search. The heuristics are simple and natural, and enforce pure depth-first search with backward chaining in the standard conflict-directed clause learning (CDCL) framework.
In this work we consider alternatives to pure depth-first search, and show that carefully chosen randomized search order, which is not strictly depth-first, allows to leverage the intrinsic strengths of CDCL better, and will lead to a planner that clearly outperforms existing planners.

1 Introduction

Translation into SAT, the satisfiability problem of the classical propositional logic, has been one of the main approaches to solving AI planning problems. The basic idea, first presented by Kautz and Selman [1], is to consider a bounded-horizon planning problem, to represent the values of state variables at every time point as propositional variables, and to represent the relation between two consecutive states as a propositional formula. This idea is essentially the same as in the simulation of nondeterministic polynomial-time Turing machines in Cook's proof of NP-hardness of SAT [2]. Kautz and Selman's idea, and Cook's even more so, was considered to be only of theoretical interest until 1996 when algorithms for SAT had developed far enough to make planning with SAT practical and even competitive with other search methods [3].

Recently, planning-specific improvements to generic SAT algorithms have been proposed. Specifically, the conflict-directed clause learning (CDCL) algorithm for SAT can be forced to do depth-first backward chaining search by a suitable variable selection scheme [4]. Although the idea is very simple and elegant, surprisingly it also results in dramatic improvements to SAT-based planning, and lifts its efficiency to the same level with the currently best algorithms for classical planning [4].

In this paper we propose an alternative search scheme which does not enforce a strict depth-first search. Algorithms for SAT have a great flexibility in choosing the decision variables, and the kind of fixed schemes as in the predecessor work, do not, for most applications, lead to the best possible performance (and for many applications would lead to a poor performance.) The technical challenge is increasing the flexibility in the decision variable selection in a way that actually improves performance.

Additionally, we propose heuristics to order goals and subgoals. The predecessor work [4] ordered them arbitrarily, determined by their order in the input file.

Our experiments show that with the new improvements our planner substantially outperforms all well-known classical planning systems, including LAMA, the winner of the last (2008) planning competition.

The structure of the paper is as follows. Section 2 explains the background of the work. In Section 3 we present the earlier variable selection scheme. Section 4 extends it by goal-ordering heuristics and by relaxing the search order. In Section 5 we experimentally evaluate the impact of the techniques. We conclude the paper in Section 6.

2 Preliminaries

The classical planning problem involves finding an action sequence from a given initial state to a goal state. The actions are deterministic, which means that an action and the current state determine the successor state uniquely. A state $s : A \to \{0, 1\}$ is a valuation of A, a finite set of *state variables*. In the simplest formalization of planning, actions are pairs (p, e) where p and e are consistent sets of propositional literals over A, respectively called *the precondition* and *the effects*. We define $\mathrm{prec}((p, e)) = p$. Actions of this form are known as STRIPS actions for historical reasons. An action (p, e) is *executable* in a state s if $s \models p$. For a given state s and an action (p, e) executable in s, the unique successor state $s' = \mathit{exec}_{(p,e)}(s)$ is determined by $s' \models e$ and $s'(a) = s(a)$ for all $a \in A$ such that a does not occur in e. This means that the effects are true in the successor state and all state variables not affected by the action retain their values. Given an initial state I, a plan to reach a goal G (a set of literals) is a sequence of actions $o_1, \ldots, o_n$ such that $\mathit{exec}_{o_n}(\mathit{exec}_{o_{n-1}}(\cdots \mathit{exec}_{o_2}(\mathit{exec}_{o_1}(I)) \cdots)) \models G$.

The basic idea in applying SAT to planning is, for a given set A of state variables, an initial state I, a set O of actions, goals G and a horizon length T, to construct a formula Φ_T such that $\Phi_T \in \mathrm{SAT}$ if and only if there is a plan with horizon $0, \ldots, T$. This formula is expressed in terms of propositional variables $a@0, \ldots, a@T$ for all $a \in A$ and $o@0, \ldots, o@T - 1$ for all $o \in O$. For a given $t \geq 0$, the valuation of $a_1@t, \ldots, a_n@t$, where $A = \{a_1, \ldots, a_n\}$, represents the state at time t. The valuation of all variables represents a state sequence so that the difference between two consecutive states corresponds to taking zero or more actions. This can be defined in several different ways [5]. For our purposes it is sufficient that the step-to-step change from state s to s' by a set X of actions satisfies the following three properties: *1)* $s \models p$ for all $(p, e) \in X$, *2)* $s' \models e$ for all $(p, e) \in X$, and *3)* $s' = \mathit{exec}_{o_n}(\mathit{exec}_{o_{n-1}}(\cdots \mathit{exec}_{o_2}(\mathit{exec}_{o_1}(s)) \cdots))$ for some ordering $o_1, \ldots, o_n$ of X. These conditions are satisfied by all main encodings of planning as SAT [4]. The only encoding not satisfying these conditions (part 1, specifically) is the relaxed $\exists$-step semantics encoding of Wehrle and Rintanen [6].

Given a translation into propositional logic, planning reduces to finding a horizon length T such that $\Phi_T \in \mathrm{SAT}$, and reading a plan from a satisfying assignment for Φ_T. To find such a T, early works sequentially tested Φ_1, Φ_2, and so on, until a satisfiable formula was found. More efficient algorithms exist [7, 8].

3 The Variable Selection Scheme

The conflict-directed clause learning (CDCL) algorithm is the basis of most of the currently best SAT solvers in the zChaff family [9]. Introductory presentations of CDCL algorithms exist [10, 11]. The algorithm repeatedly chooses a decision variable, assigns a truth-value to it, and performs inferences with the unit resolution rule, until a contradiction is obtained (the empty clause is derived, or, equivalently, the current valuation falsifies one of the input clauses or derived clauses.) The sequence of variable assignments that led to the contradiction is analyzed, and a clause preventing the repeated consideration of the same assignment sequence is derived and added to the clause set.

The earlier variable selection scheme for planning [4] performed a depth-first search by a stack-based algorithm, finding one action (decision variable) to be used in the CDCL algorithm as the next variable to which a value is assigned. In this section we present two technically simple extensions that allow more flexible traversal orders and the consideration of more than one candidate decision variable. In Section 4 we will utilize these extensions by proposing subgoal ordering heuristics and a more flexible decision variable selection scheme than the strict depth-first one used earlier.

The main challenge in defining a variable selection scheme is its integration in the overall SAT solving algorithm in a productive way. To achieve this, the variable selection depends not only on the initial state, the goals and the actions, represented by the input clauses, but also the current search state of the CDCL algorithm. The algorithm's execution state is characterized by 1) the current set of learned clauses and 2) the current (partial) valuation reflecting the decisions (variable assignments) and inferences (with unit propagation) made so far. Our variable selection scheme only uses part 2 of the execution state, the current partial valuation v.

The earlier variable selection scheme [4] is based on the following observation: each of the goal literals has to be made *true* by an action, and the precondition literals of each such action have to be made *true* by earlier actions (or, alternatively, these literals have to be *true* in the initial state.)

The first step in selecting a decision variable is finding the earliest time point at which a goal literal can become and remain *true*. This is by going backwards from the end of the horizon to a time point t' in which A) an action making the literal *true* is taken or B) the literal is *false* (and the literal is *true* or *unassigned* thereafter.) The third possibility is that the initial state at time point 0 is reached and the literal is *true* there, and hence nothing needs to be done. In case A we have an action already in the plan, and in case B we choose any action that can change the literal from *false* to *true* between t' and $t'+1$ and use it as a decision variable.[1] In case A we push the literals in the precondition into the stack and find supporting actions for them.

In the earlier work it was shown that finding just one action in a depth-first manner is sufficient for an impressive performance [4]. The new algorithm differs from the earlier algorithm in two respects. First, the depth-first search is not terminated after one action is found, but proceeds further (in Fig. 1 all possible candidate actions will be found.) Second, we replace the stack with a priority queue, which enables the use of a heuristic to impose different traversal orders. These two changes are technically trivial, and the challenge is to utilize them in a way that will actually lead to an improved performance.

The extension of the earlier algorithm [4] for computing a set of actions that support currently unsupported top-level goals or preconditions of actions in the current partial plan is given in Fig. 1. For negative literals $l = \neg a$, $l@t$ means $\neg(a@t)$, and for positive literals $l = a$ it means $a@t$. Similarly, we define the valuation $v(l@t)$ for negative literals $l = \neg a$ by $v(l@t) = 1 - v(a@t)$ whenever $v(a@t)$ is defined. For positive literals $l = a$ of course $v(l@t) = v(a@t)$.

```
 1: procedure support(G, O, T, v)
 2: empty the priority queue;
 3: for all l ∈ G do push l@T into the priority queue;
 4: X := ∅;
 5: while the priority queue is non-empty do
 6:    pop l@t from the priority queue;        (* Take one (sub)goal. *)
 7:    t' := t − 1;
 8:    found := 0;
 9:    repeat
10:       if v(o@t') = 1 for some o ∈ O with l ∈ eff(o)
11:       then                    (* The subgoal is already supported. *)
12:          for all l' ∈ prec(o) do push l'@t' into the priority queue;
13:          found := 1;
14:       else if v(l@t') = 0 then     (* Earliest time it can be true *)
15:          o := any o ∈ O such that l ∈ eff(o) and v(o@t') ≠ 0;
16:          X := X ∪ {o@t'};
17:          for all l' ∈ prec(o) do push l'@t' into the priority queue;
18:          found := 1;
19:       t' := t' − 1;
20:    until found = 1 or t' < 0;
21: end while
22: return X;
```

Fig. 1. Computation of supports for (sub)goals

The procedure in Fig. 1 is the main component of the variable selection scheme for CDCL given in Fig. 2, in which an action

```
1: S := support(G, O, T, v);
2: if S ≠ ∅ then v(o@t) := 1 for any o@t ∈ S; (* Found an action. *)
3: else
4:    if there are unassigned a@t for a ∈ A and t ∈ {1, ..., T}
5:    then v(a@t) := v(a@(t − 1)) for any a@t with minimal t
6:    else v(o@t) := 0 for any o ∈ O and t ≥ 0 with o@t unassigned;
```

Fig. 2. Variable selection for planning with the CDCL algorithm

is chosen as the next decision variable for the CDCL algorithm if one is available. If none is available, all goals and subgoals are already supported. The current valuation typically is still not complete, and it is completed by assigning unassigned fact variables

[1] Such an action must exist because otherwise the literal would have to be *false* also at $t'+1$.

the value they have in the predecessor state (line 5) and assigning unassigned action variables the value *false* (line 6). The code in Fig. 2 replaces VSIDS as the variable selection heuristic in the CDCL algorithm.

4 Heuristics for Variable Selection

The variable selection scheme, as described in Section 3, has already led to a planner that is very competitive with the best existing planners for the classical planning problem [4]. However, experience from SAT solvers and from the application of SAT solving to planning specifically [12] suggests that the fixed goal-orderings and the strict backward chaining depth-first search do not – although better than generic SAT-solvers [4] – ultimately represent the most efficient form of search in the CDCL context.

First, we will present a goal-ordering heuristic for controlling the priority queue. If only the first action found is returned, the traversal order in the algorithm in Fig. 1 directly determines the ordering in which variables are assigned in the CDCL algorithm.

Second, the search with strict backward chaining will be relaxed. Backward chaining means selecting an action with an effect x given a goal x, and taking the preconditions of the action as new goals, for which further actions are chosen. The search with backward chaining proceeds step by step toward earlier time points (until some form of backtracking will take place.) In the context of CDCL and other SAT algorithms, the search does not have to be directional in this way, and actions less directly supporting the current (sub)goals could be chosen, arbitrarily many time points earlier. The algorithm in Fig. 1 computes a complete set of candidate actions for supporting all goals and subgoals (as opposed to finding only one as in the predecessor work [4]), but randomly choosing one action from this set is not useful, and we need a more selective way of choosing a decision variable.

Next we will consider these two possible areas of improvement, and in each case propose a modification to the basic variable selection scheme which will be shown to lead to substantial performance improvements in Section 5.

4.1 Goal Ordering

We considered two measures according to which (sub)goals $l@t$ are ordered.

1. the maximal $t' < t$ such that $v(l@t') \neq 1$
2. the maximal $t' < t$ such that $v(l@t') = 0$

Above, $v(l@t') \neq 1$ includes the case that $v(l@t')$ is unassigned. In the first case, l gets a higher priority if it must have been made *true* earlier than other subgoals. The most likely plan involves making l first *true*, followed by making the other subgoals *true*. The second case looks at the time when the subgoals must have been *false* the last time. Empirically the best results were obtained with the first. Intuitively, this measure is a better indicator of the relative ordering of the actions establishing different preconditions of a given action.

A key property of these measures is that for every goal or subgoal $l@t$, the new subgoals $l_1@t-1, \ldots, l_n@t-1$ all have a higher priority than their parent $l@t$. This will still lead to depth-first search, but the ordering of the child nodes will be informed.

4.2 Computation of Several Actions

To achieve a less directional form of plan search with CDCL, we decided to compute some fixed number $N = |S|$ of actions (not only $N = 1$ as in [4]) and randomly choose one $o@t \in S$. In the algorithm in Fig. 1 this means adding a statement that returns S as soon as $|S| = N$. The initial experiments seemed very promising in solving some of the difficult problems much faster. However, the overall improvement was relatively small, and it very surprisingly peaked at $N = 2$.

What happened is the following. For a given top-level goal $l \in G$, several of the first actions that were chosen supported the goals. However, after everything needed to support l was included, the computation continued from the *next unsupported top-level goal*. So at the final stages of finding support for a top-level goal we would be, in many cases, instead selecting supporting actions for other top-level goals, distracting from finding support for l. With $N = 2$ the distraction is small enough to not outweigh the benefits of considering more than one action.

This analysis led us to a second variant, which proved to be very powerful. In this variant we record the time-stamp t of the first action found. Then we continue finding up to N actions, but *stop and exit* if the time-stamp of a would-be candidate action is $\geq t$. With this variant we obtained a substantial overall improvement with higher N. Later in the experiments we use $N = 10$ because the improvement leveled off at $N = 10$.

4.3 Discussion

The good performance of the fixed and uninformed variable selection [4] is due to its focus on a particular action sequence. Any diversion from a previously tried sequence is a consequence of the clauses learned with CDCL. This maximizes the utility of learned clauses, but also leads to the possibility of getting stuck in a part of the search space void of solutions. A remedy to this problem in current SAT solvers is restarts [9]. However, with deterministic search and without VSIDS-style variable (or action) weighting mechanism restarts make no difference. In SAT algorithms that preceded VSIDS, a small amount of randomization was used to avoid getting stuck [13]. However, too large diversion from the previous action sequences makes it impossible to benefit from the clauses learned with CDCL. Hence the key problem is finding a balance between focus to recently traversed parts of the search space and pursuing other possibilities.

The flexible depth-first style search from Section 4.2 provides an interesting balance between focus and variation. The candidate actions all contribute to one specific way of supporting the top-level goals, but because they often don't exactly correspond to an actual plan (except for at the very last stages of the search), varying the order in which they are considered seems to be an effective way of probing the "mistakes" they contain. An additional benefit seems to be that the non-linear ordering in which the candidate actions are used often leaves holes (missing actions) in the incomplete plan, which are immediately filled by unit propagation. For

this reason the number of decisions needed in the CDCL algorithm is sometimes much smaller.

5 Evaluation

Our base line in the evaluation is the backward chaining fixed variable-selection scheme introduced in the predecessor work [4]. This scheme was already shown to outperform the standard VSIDS heuristic, both our own implementation and current best implementations in generic SAT solvers, including Precosat and RSAT.

The test material was 968 problem instances from the international planning competitions from 1998 until 2008. Since our variable selection scheme is defined for the restricted STRIPS language only, we chose all the STRIPS problems except for some from the first competitions, nor did we choose benchmarks from an earlier competition if the same domain had been used in a later competition as well.

We used the most efficient known translation from planning into SAT, for the $\exists$-step semantics by Rintanen et al. [5], and solved the problems with the algorithm B of Rintanen et al. [5] with $B = 0.9$, testing horizon lengths $0, 5, 10, 15, \ldots$ and solving a maximum of 18 SAT problems simultaneously.

All the experiments were run in an Intel Xeon CPU E5405 at 2.00 GHz with a minimum of 4 GB of main memory and using only one CPU core. We ran our planner for all of the problem instances, giving a maximum of 300 seconds for each instance. The runtime includes all standard phases of a planner, starting from parsing the PDDL description of the benchmark and ending in outputting a plan. The different variants of the planner are the baseline fixed variant *base* from the earlier paper [4], *o* with the subgoal ordering from Section 4.1 but with only one action found and returned by the procedure call support(G, O, T, v), *m* with random choice from multiple candidate actions from Section 4.2, and *o+m* which combines the previous two. The randomization in *m* and *m+o* affects the runtimes, but not much: different complete runs of all 968 instances solved couple of instances more or less, depending on whether for some instances the runtime was slightly below or slightly above the 300 second time limit.

We also tested LAMA [14], the winner of the last (2008) planning competition, and ran it with its default settings, except for limiting its invariant computation to a maximum of 60 seconds according to Helmert's instructions, to adjust for the 300 second time limit we used. Due to a bug in one of its components, LAMA is not able to solve the first instance of OPTICAL-TELEGRAPH and the first 13 instances of PHILOSOPHERS (the rest take longer than 300 seconds.)

The results of the experiment are summarized in Table 1. The first column is the number of (solvable) problem instances in each domain. To get an idea of the differences in the runtime behavior of the different variants of the planner, we plotted a curve showing the number of problem instances solved (y axis) with a given timeout limit (x axis), shown in Fig. 3. Overall, the improvements of the new techniques over the baseline planner and LAMA are substantial, no matter which time out limit is considered.

Other well-performing planners in the planning competitions starting from 2000, including FF and YAHSP from the HSP family

domain		VSIDS	base	o	m	o+m	LAMA
1998-GRIPPER	20	20	20	20	20	20	20
1998-MPRIME	20	16	18	18	20	20	20
1998-MYSTERY	19	16	17	17	17	17	19
2000-BLOCKS	102	71	85	86	90	90	51
2000-LOGISTICS	76	76	76	76	76	76	76
2002-DEPOTS	22	21	21	22	22	22	16
2002-DRIVERLOG	20	15	20	20	19	19	20
2002-FREECELL	20	4	5	5	12	12	18
2002-ZENO	20	18	20	20	20	20	20
2004-AIRPORT	50	40	42	41	43	42	37
2004-OPTICAL-TELEG	14	14	14	14	14	14	2
2004-PHILOSOPHERS	29	29	29	29	29	29	BUG
2004-PIPESWORLD-NO	50	15	20	20	33	34	44
2004-PSR-SMALL	50	50	49	49	50	50	50
2004-SATELLITE	36	29	32	32	32	32	30
2006-PIPESWORLD	50	9	10	12	21	24	38
2006-ROVERS	40	40	40	40	39	39	40
2006-STORAGE	30	29	30	30	30	30	18
2006-TPP	30	26	26	28	30	30	30
2006-TRUCKS	30	19	29	29	30	30	8
2008-ELEVATORS	30	13	30	30	30	30	30
2008-OPENSTACKS	30	15	11	11	15	15	30
2008-PARCPRINTER	30	30	30	30	30	30	28
2008-PEGSOLITAIRE	30	25	21	27	23	30	29
2008-SCANALYZER	30	19	16	26	21	27	27
2008-SOKOBAN	30	2	4	4	5	5	18
2008-TRANSPORT	30	10	12	12	20	21	28
2008-WOODWORKING	30	30	30	30	30	30	28
total	968	701	757	778	821	838	775
time average		9.68	6.24	5.99	3.60	3.53	12.23
size average		81.53	60.68	60.40	64.61	64.33	66.64

Table 1. Number of problems solved in 300 seconds for each benchmark domain. Average solution times and numbers of actions for instances solved by all.

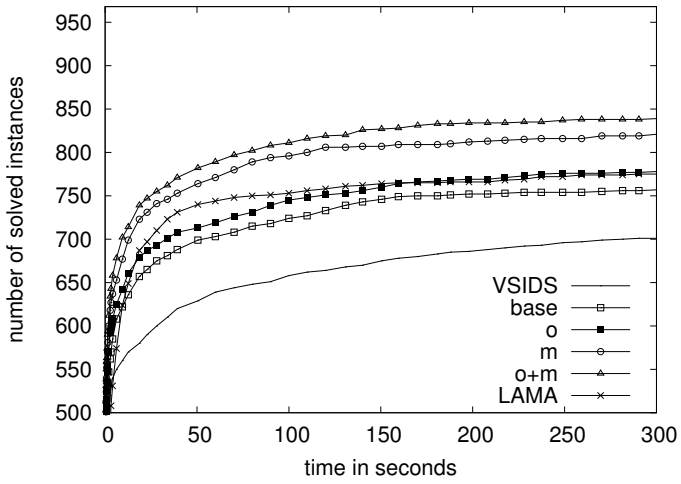

Fig. 3. Number of instances that are solved in a given time

of planners which use delete relaxation heuristics [15] and LPG-td [16], are overall very close to LAMA (within 1.5 per cent) in terms of number of solved problem instances. These planners solve respectively 786, 775[2] and 779 problem instances in 300 seconds. As an illustration of the overall performance difference, the number of problem instances FF solves in 30000 seconds equals the number for our planner with a 90 second time limit. This means that FF would have to become more than two orders of magnitude faster on average to match the performance of our planner.

6 Conclusions and Future Work

We have considered a number of goal orderings for a CDCL variable selection scheme for planning, and demonstrated substantial improvements in the performance of SAT solvers in solving standard benchmark problems.

A notable difference between our work and VSIDS [9] is that we are not using weights of decision variables obtained from conflicts as a part of variable selection. Such weights would be able to order the top-level goals and subgoals in the computation of actions, based on their role in conflicts. This, we believe, is the most promising area for future improvement in the implementations of our variable selection scheme.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Kautz, H., Selman, B.: Planning as satisfiability. In Neumann, B., ed.: Proceedings of the 10th European Conference on Artificial Intelligence, John Wiley & Sons (1992) 359–363
2. Cook, S.A.: The complexity of theorem proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. (1971) 151–158
3. Kautz, H., Selman, B.: Pushing the envelope: planning, propositional logic, and stochastic search. In: Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference, AAAI Press (August 1996) 1194–1201
4. Rintanen, J.: Heuristics for planning with SAT. In Cohen, D., ed.: Principles and Practice of Constraint Programming - CP 2010, 16th International Conference, CP 2010, St. Andrews, Scotland, September 2010, Proceedings. Number 6308 in Lecture Notes in Computer Science, Springer-Verlag (2010) 414–428
5. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artificial Intelligence **170**(12-13) (2006) 1031–1080
6. Wehrle, M., Rintanen, J.: Planning as satisfiability with relaxed $\exists$-step plans. In Orgun, M., Thornton, J., eds.: AI 2007 : Advances in Artificial Intelligence: 20th Australian Joint Conference on Artificial Intelligence, Surfers Paradise, Gold Coast, Australia, December 2-6, 2007, Proceedings. Number 4830 in Lecture Notes in Computer Science, Springer-Verlag (2007) 244–253
7. Rintanen, J.: Evaluation strategies for planning as satisfiability. In López de Mántaras, R., Saitta, L., eds.: ECAI 2004. Proceedings of the 16th European Conference on Artificial Intelligence, IOS Press (2004) 682–687
8. Rintanen, J.: Planning and SAT. In Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Number 185 in Frontiers in Artificial Intelligence and Applications. IOS Press (2009) 483–504
9. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01), ACM Press (2001) 530–535
10. Mitchell, D.G.: A SAT solver primer. EATCS Bulletin **85** (February 2005) 112–133
11. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research **22** (2004) 319–351
12. Rintanen, J.: A planning algorithm not based on directional search. In Cohn, A.G., Schubert, L.K., Shapiro, S.C., eds.: Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98), Morgan Kaufmann Publishers (1998) 617–624
13. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97), AAAI Press (1998) 431–437
14. Richter, S., Helmert, M., Westphal, M.: Landmarks revisited. In: Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08), AAAI Press (2008) 975–982
15. Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence **129**(1-2) (2001) 5–33
16. Gerevini, A., Serina, I.: Planning as propositional CSP: from Walksat to local search techniques for action graphs. Constraints Journal **8** (2003) 389–413

[2] YAHSP does not solve the 30 TPP problems because of a parser bug. Fixing this bug would probably lift YAHSP's number close to 805, making it the second fastest planner after ours.