Heuristics for Planning with SAT

Jussi Rintanen

NICTA and the Australian National University
Canberra, Australia

Abstract. Generic SAT solvers have been very successful in solving hard combinatorial problems in various application areas, including AI planning. There is potential for improved performance by making the SAT solving process more application-specific. In this paper we propose a variable selection strategy for AI planning. The strategy is based on generic principles about properties of plans, and its performance with standard planning benchmarks often substantially improves on generic variable selection heuristics used in SAT solving, such as the VSIDS strategy. These improvements lift the efficiency of SAT based planning to the same level as best planners that use other search methods.

1 Introduction

Planning is one of several problems that have been successfully solved with SAT algorithms [1]. Most works have used a generic SAT solver, which recently use the conflict-directed clause learning (CDCL) algorithm and the VSIDS heuristic [2].

In this work we investigate SAT solving for planning with CDCL but with a heuristic that radically differs from VSIDS and is based on a simple property all solutions to a planning problem have to satisfy. The work is motivated by the need to better understand why SAT solvers are successful in solving AI planning and related problems, and by the need and opportunity to develop more powerful, problem-specific heuristics for SAT.

Our heuristic chooses action variables that contribute to the goals or (current) subgoals. The principle is extremely simple: *for a given (sub)goal, choose an action that achieves the (sub)goal and that can be taken at the earliest time in which the (sub)goal can become (and remain)* true. After choosing an action, its preconditions become new subgoals, for which supporting actions are found in the same way. This principle is easy to implement: start from a goal (or a subgoal), go backwards step by step until a time point in which the goal is *false*, and choose any of the actions that achieve the goal and are possible at that time point. If such an action existed in the plan already, perform the procedure recursively with the preconditions of the action as the subgoals.

The above principle is so simple that it is surprising that it alone, without any further heuristics, is, for satisfiable problems representing standard planning benchmark problems, very often far more effective than the VSIDS heuristic found in the best current SAT solvers. Furthermore, the new heuristic lifts the efficiency of SAT-based planning to the same level with the best existing planning algorithms that are based on other search methods, including heuristic state space search. This result is extremely surprising because the currently best state-space search planners, which have their origins in the work of Bonet and Geffner [3] more than ten years ago, use far more complex heuristics, and were in most cases developed with the very benchmark problems in mind which they are usually evaluated on. In contrast, our heuristic relies on a very natural and simple principle that represents common intuitions about planning.

We view the new heuristic as a step toward developing better SAT-based techniques for planning and other related problems such as model-checking and discrete-event systems diagnosis. More advanced heuristics for these applications are likely to also incorporate features from VSIDS, including the computation of *weights* of variables based on their occurrence in recently learned clauses.

As already mentioned, the new heuristic is better than VSIDS with satisfiable formulas representing standard planning benchmarks, and worse with unsatisfiable ones. Heuristics in SAT solving have two complementary roles. For satisfiable formulas, heuristics help in finding a satisfying assignment quickly, hopefully avoiding much of the brute-force search resulting from wrong choices of variables and truth-values. This is similar to heuristics in state-space search for planning which help in finding a path (a state sequence) to a goal state. For an unsatisfiable formula, heuristics help the solver to efficiently find a *refutation proof*. Our heuristic only indirectly addresses unsatisfiability proofs by avoiding actions that cannot possibly contribute to the refutation proof. Heuristics should more explicitly try to make finding the unsatisfiability proofs easier, but we did not have this as an objective for our new heuristics.

The two roles of heuristics in SAT solving are both important, but their importance depends on the type of planning being done. If the objective is to find an optimal plan, one has to *prove* that no better plans exist. For this one needs efficient detection of unsatisfiability. If the objective is to just find a plan, with no quality guarantees, the importance of efficient unsatisfiability proofs is much smaller. The experiments we will be presenting later address this duality. We leave the problem of improving the efficiency of unsatisfiability for future work.

In addition to improved efficiency of finding satisfying valuations, another benefit our variable selection heuristic has over VSIDS, from the point of view of planning research, is that it is easy to understand, and there are obvious avenues to improving the scheme, and obvious avenues for expressing follow-up questions about it, for example concerning the difficulty of finding unsatisfiability proofs. Addressing these things in the context of the VSIDS heuristics seems quite a bit trickier because of a lack of intuitive explanations of what VSIDS does in terms of search for plans.

The structure of the paper is as follows. Sect. 2 describes the background of the work in planning with SAT. In Sect. 3 we explain the CDCL algorithm. In Sect. 4 we present the variable selection scheme for planning, experimentally evaluate it in Sect. 5, and discuss related work in Sect. 6 before concluding the paper in Sect. 7.

2 Planning as Satisfiability

The classical planning problem involves finding a sequence of actions 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. In the simplest formalization of planning actions are pairs (p, e) where p and e are consistent

sets of propositional literals on the finite set A of *state variables*, respectively called *the precondition* and *the effects*. Actions described this way are often known as STRIPS actions for historical reasons. An action (p, e) is *executable* in a state s if $s \models p$. A state $s : A \to \{0, 1\}$ is a valuation of all state variables. For a given state s and an action (p, e) which is executable in s, the unique successor state $s' = exec_{(p,e)}(s)$ 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 solution to the planning problem with goal G (a set of literals) is a sequence of actions $o_1, \ldots, o_n$ such that $exec_{o_n}(exec_{o_{n-1}}(\cdots exec_{o_2}(exec_{o_1}(I))\cdots)) \models G$.

Kautz and Selman [1] proposed finding plans by reduction to SAT. The reduction is similar to the reduction of NP Turing machines to SAT used in Cook's proof of NP-hardness of SAT [4]. The reduction is parameterized by a horizon length $T \geq 0$. The value of each state variable $a \in A$ in each time point $t \in \{0, \ldots, T\}$ is represented by a propositional variable $a@t$. Additionally, it is often useful to have the actions represented as propositional variables, so for each action o and $t \in \{0, \ldots, T-1\}$ we similarly have a propositional variable $o@t$ indicating whether action o is taken at t.

In this paper, we use an encoding of planning which allows several actions to be taken at the same time point in parallel. The "parallelism" (partial ordering) is allowed when it is possible to totally order the actions to a sequential plan as defined above. There are different possibilities in defining this kind of parallel plans [5, 6].

To represent planning as a SAT problem, each action $o = (p, e)$ and time point $t \in \{0, \ldots, T-1\}$ is mapped to formulas $o@t \to \bigwedge_{l \in p} l@t$ and $o@t \to \bigwedge_{l \in e} l@(t+1)$.[1] These two formulas respectively correspond to the executability condition and the first part of the definition of successor states. The second part, about state variables that do not change, is encoded as follows, for the case in which in several actions can be taken in parallel. For each state variable $a \in A$ we have the formula

$$(\neg a@t \wedge a@(t+1)) \to (o_1^a@t \vee \cdots \vee o_n^a@t)$$

where $o_1^a, \ldots, o_n^a$ are all the actions which have a as an effect. Similarly we have for each $a \in A$ the formula

$$(a@t \wedge \neg a@(t+1)) \to (o_1^{\neg a}@t \vee \cdots \vee o_m^{\neg a}@t)$$

for explaining possible changes from true to false, where $o_1^{\neg a}, \ldots, o_m^{\neg a}$ are all the actions with $\neg a$ as an effect. These formulas (often called *the frame axioms*) allow to infer that a state variable does not change if none of the actions changing it is taken.

Finally, to rule out solutions that don't correspond to any plan because parallel actions cannot be serialized[2], further formulas are needed. In this work we have used the linear-size $\exists$-step semantics encoding of Rintanen et al. [5].

[1] For negative literals $l = \neg a$, $l@t$ means $\neg(a@t)$, and for positive literals $l = a$ it means $a@t$.

[2] For example, actions $(\{a\}, \{\neg b\})$ and $(\{b\}, \{\neg a\})$ cannot be made to a valid sequential plan, because taking either action first would falsify the precondition of the other.

There is one more component in efficient SAT encodings of planning, which is logically redundant but usually critical for efficiency: *invariants* (mutexes). Invariants $l \vee l'$ express dependencies between state variables. Many of the standard planning benchmarks represent multi-valued state variables in terms of several Boolean ones, and a typical invariant $\neg x_a \vee \neg x_b$ says that a multi-valued variable x can only have one of the values a and b. To compute these invariants, we used the algorithm of Rintanen [7] which is defined for the general (ground) PDDL language which includes STRIPS.

For a given set A of state variables, initial state I, set O of actions, goals G and horizon length T, we can compute (in linear time in the product of T and the sum of sizes of A, I, O and G) a formula Φ_T such that $\Phi_T \in$ SAT if and only if there is a plan with horizon $0, \ldots, T$. Φ_T includes the formulas described above, and for all $a \in A$ the unit clause $a@0$ if $I(a) = 1$ and $\neg a@0$ if $I(a) = 0$, and $l@T$ for all $l \in G$. These formulas are in CNF after trivial rewriting.

A planner can do the tests $\Phi_0 \in$ SAT, $\Phi_1 \in$ SAT, $\Phi_2 \in$ SAT, and so on, sequentially one by one, or it can make several of these tests in parallel (interleave them). For this we will later be using Algorithm B of Rintanen et al. [5] which allocates CPU time to different horizon lengths according to a decreasing geometric series, so that horizon length $t+1$ gets γ times the CPU the horizon length t gets, for some fixed $\gamma \in]0, 1]$. In general, the parallelized strategies can be orders of magnitudes faster than the sequential strategy because they do not need to complete the test $\Phi_t \in$ SAT (finding Φ_t unsatisfiable) before proceeding with the test $\Phi_{t+1} \in$ SAT. This explains why, in this setting, it is far more important to efficiently determine satisfiability than unsatisfiability.

3 The CDCL Algorithm for SAT

In this section we briefly describe the standard conflict-directed clause learning (CDCL) algorithm for the SAT problem. This algorithm is the basis of most of the currently leading SAT solvers in the zChaff family [2].

For a general overview of the CDCL algorithm see standard references [8, 9]. The main loop of the CDCL algorithm (see Fig. 1) chooses an unassigned variable, assigns a truth-value to it, and then performs unit propagation to extend the current valuation v with forced variable assignments that directly follow from the existing valuation by the unit resolution rule. If one of the clauses is falsified, a new clause which would have prevented the current valuation is learned. This new clause is a logical consequence of the original clause set. Then, some of the last assignments are undone, and the assign-infer-learn cycle is repeated. The procedure ends when the empty clause has been learned (no valuation can satisfy the clauses) or a satisfying valuation has been found.

The selection of the decision variable (line 5) and its value (line 6) can be arbitrary (without compromising the correctness of the algorithm), and can therefore be based on a heuristic. The heuristic is critical for the efficiency of the CDCL algorithm. On line 7 the standard unit propagation algorithm is run. It infers a forced assignment for a variable x if there is a clause $x \vee l_1 \vee \cdots \vee l_n$ or $\neg x \vee l_1 \vee \cdots \vee l_n$ and $v \models \neg l_1 \vee \cdots \neg \vee l_n$.

The inference of a new clause on line 10 is the key component of CDCL. The clause will prevent generating the same unsatisfy-

```
1:  procedure CDCL(C)
2:  Initialize v to satisfy all unit clauses in C;
3:  extend v by unit propagation with v and C;
4:  while C does not contain the empty clause do
5:      choose a variable a with v(a) unassigned;
6:      assign v(a) := 1 or v(a) := 0;
7:      extend v by unit propagation with v and C;
8:      if v falsifies a clause in C
9:      then
10:         infer a new clause c and add it to C;
11:         undo assignments until a so that c is not falsified;
12:     end if
13: end while
```

Fig. 1. Outline of the CDCL algorithm

ing assignment again, leading to traversing a different part of the search space.

4 A New Heuristic for Planning

The goal of the current work is to present a new way of choosing the decision variables (lines 5 and 6 in the CDCL procedure in Fig. 1) specific to planning. Generic CDCL solvers choose the decision variables based on the variables' weights calculated from their occurrence in recently learned conflict clauses. Our proposal only affects the variable selection part, and hence it doesn't affect the correctness or completeness of the underlying CDCL algorithm.

4.1 Variable Selection to Satisfy Goals and Subgoals

Our heuristic 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 another action (or they have to be *true* in the initial state.)

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 state of the SAT solver. The state of the solver is primarily characterized by A) the current set of learned clauses and B) the current (partial) valuation reflecting the decisions (variable assignments) and inferences (with unit propagation) made so far. We have restricted the variable selection to use only part B of the SAT solver state, the current partial valuation.

Our algorithm identifies one (sub)goal that is not at the current state of search supported (made true) by an action or the initial state. The search for such support proceeds from the goal literals G at the last time point T in the horizon.

The first step is to find the earliest time point at which a goal literal can become and remain *true*. This happens 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 it 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, and in case B we choose any action that could change the literal from *false* to *true* between t and $t+1$.[3] In case A we find support for the preconditions of the action in the same way. The first action found will be used as the next decision variable in the CDCL algorithm.

The computation is started from scratch at every step of the CDCL procedure because a particular support for a (sub)goal, for example the initial state, may become irrelevant because of a later decision, and a different support needs to be found.

When no (sub)goal without support is found, the current partial assignment represents a plan. The assignment can be made total by assigning the unassigned action variables to *false* and the unassigned fact variables the value they have in the closest preceding time point with a value (inertia).

The algorithm is given in Fig. 2. It takes a stack containing the top-level goals as input, and it returns the empty set or a singleton set $\{o@t\}$ for some action o and time t. Our operation for pushing elements in the stack marks them, and already marked elements are later ignored. This is to prevent subgoal literals occurring in the computation repeatedly. We define $\text{prec}((p,e)) = p$ and $\text{eff}((p,e)) = e$.

```
1:  procedure support(Stack, v)
2:  while Stack is non-empty do
3:      pop l@t from the Stack;
4:      t' := t − 1;
5:      found := 0;
6:      repeat
7:          if v(o@t') = 1 for some o ∈ O with l ∈ eff(o)
8:          then
9:              for all l' ∈ prec(o) do push l'@t' into the Stack;
10:             found := 1;
11:         else if v(l@t') = 0 then
12:             o := any o ∈ O such that l ∈ eff(o) and v(o@t') ≠ 0;
13:             return {o@t'};
14:         t' := t' − 1;
15:     until found = 1 or t' < 0;
16: end while
17: return ∅;
```

Fig. 2. Finding support for one unsupported subgoal

Example 1. We illustrate the search for an unsupported (sub)goal and the selection of an action with a problem instance with goals a and b and actions $X = (\{d\},\{a\})$, $Y = (\{e\},\{d\})$, and $Z = (\{\neg c\},\{b\})$.

variable	0	1	2	3	4	5	6
a	0	0			**0**		1
b	0	0	**0**	1			1
c	0	0					
d	0	0	**0**				
e	1						

The timed literals that are considered to be changing from *false* to *true* are shown in boldface. For goal a, the latest time at which a is *false* is 4.

Let's assume that $X@4$ is unassigned, and hence could make a *true* at 5, and we choose $X@4$ as a support. If this action was

[3] Such an action must exist because otherwise the literal's frame axiom would force the literal *false* also at $t+1$.

already in the partial plan, the precondition d of X would be the next unsupported (sub)goal, and it could be made *true* by Y at time 2. Hence Y would be the support of d in that case. The precondition e of Y is supported by the initial state and wouldn't need further support.

For the top-level goal b, assume that $Z@2$ is assigned *true* and hence explains the change of b from *false* to *true* between time points 2 and 3. Since Z's precondition $\neg c$ is satisfied by the initial state, again no further action is required.

4.2 Complexity of the Variable Selection Algorithm

If there are n state variables and the horizon length is T, there are nT variables that represent state variables at different time points. Because each literal is pushed into the stack at most once, the algorithm does the outermost iteration on line 2 for each goal or subgoal at most once, and hence at most nT times in total. The number of iterations of the inner loop starting on line 6 is consequently bounded by nT^2.

The actual runtime of the algorithm is usually much lower than the above upper bound. A better approximation for the number of iterations of the outer loop is the number of goals and the number of preconditions in the actions in the plan that is eventually found. In practice, the runtime of the CDCL algorithm with our heuristic is still strongly dominated by unit propagation, similarly to CDCL with VSIDS, and the heuristic takes somewhere between 5 and 30 percents of the total SAT solving time.

4.3 Integration in the CDCL Algorithm

Our variable selection scheme is embedded in the CDCL algorithm of Fig. 1 by replacing lines 5 and 6 by the code in Fig. 3. Note

```
1: empty Stack;
2: for all l ∈ G do push l@T into the Stack;
3: S := support(Stack, v);
4: if S = {o@t} for some o and t then v(o@t) := 1
5: else
6:     if there are unassigned a@t for a ∈ A and t ≥ 1
7:     then v(a@t) := v(a@(t − 1)) for one with minimal t
8:     else v(o@t) := 0 for any o ∈ O and t ≥ 0 with o@t unassigned;
```

Fig. 3. Variable selection replacing lines 5 and 6 in Fig. 1

that some actions are inferred by unit propagation on line 7 in the CDCL algorithm, and these actions are later handled indistinguishably from actions chosen by the heuristic.

The choice of o on line 12 of Fig. 2 and the ordering of the goal literals in the stack on line 2 of Fig. 3 are arbitrary, but for efficiency reasons they must be fixed. In particular, it is important that on line 12 of Fig. 2 the same action is chosen as long as the condition $v(o@t') \neq 0$ condition is satisfied. Intuitively, this is important to avoid losing focus in the CDCL search.

When $\text{support}(Stack, v) = \emptyset$, all goals and all action preconditions in the current plan are supported by another action or the initial state, and no further actions are needed. The valuation of the SAT solver is usually still partial, because many of the variables corresponding to actions that are not needed are still unassigned, and variables that do not change between two distant time points may be unassigned in some of the intermediate time points. To complete the assignment, we choose, starting from time point 1, unassigned fact variables and assign them the same value they have in the preceding time point (line 7 in Fig. 3). This also sets most of the unassigned action variables *false*. The remaining action variables (usually there are none left) are assigned *false* one by one (line 8 in Fig. 3.)

5 Evaluation

We will show that our variable selection heuristic beats VSIDS with *satisfiable* formulas, but not with *unsatisfiable* ones. Further, we will show that our heuristics lead to a planner that is competitive with one of the best planners that don't use SAT.

We used our own SAT solver which is based on CDCL, VSIDS [2], and the phase selection heuristic from RSAT [10]. We tried different clause learning schemes, and it seemed to us that for our problems the Last UIP scheme has a small edge over First UIP [9]. We use the former. We will comment on the relative efficiency of our SAT solver with respect to other solvers later.

As test material we chose 968 problem instances from the biennial international planning competitions from 1998 to 2008. Since our new heuristic is defined for the restricted STRIPS language only, we use all of the STRIPS problems from the planning competitions, except some from the first competition, nor an earlier variant of any benchmark domain that was used in two competitions.

As discussed in Sect. 2, our planner is based on the linear-size $\exists$-step semantics encoding of Rintanen et al. [5][4], and the algorithm B with parameter $\gamma = 0.9$. We considered the horizon length parameter $T \in \{0, 5, 10, 15, 20, \ldots\}$, and let the planner solve at most 18 SAT instances simultaneously. Most of the problem instances are solved with less than 500 MB of memory, but a couple of dozen required 2 GB or more, up to the 3.5 GB boundary where we could not allocate more memory. The memory restriction was dictated by the 32-bit Ubuntu Linux for which we compiled our programs. 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.

To test the performance of the heuristic for both satisfiable and unsatisfiable formulas, with emphasis on the unsatisfiable ones that are important for proofs of optimality of plans, we set up our planner to use the traditional sequential strategy which goes through horizon lengths 0, 1, 2, 3 and so on, until it finds a satisfiable formula. The results of this experiment are summarized in Fig. 4. The plot shows the number of problem instances that are solved (finding a plan) in n seconds or less when using VSIDS and when using the new heuristic (indicated by P). The solver with VSIDS solves about 15 per cent more instances when 300 seconds is spent solving each problem instance. When the optimality proof is required (to show that a plan has the optimal *parallel* length), usually almost

[4] Results for the standard parallel plans ($\forall$-step semantics) and for sequential plans are similar in terms of the improvement our new heuristic yields over VSIDS.

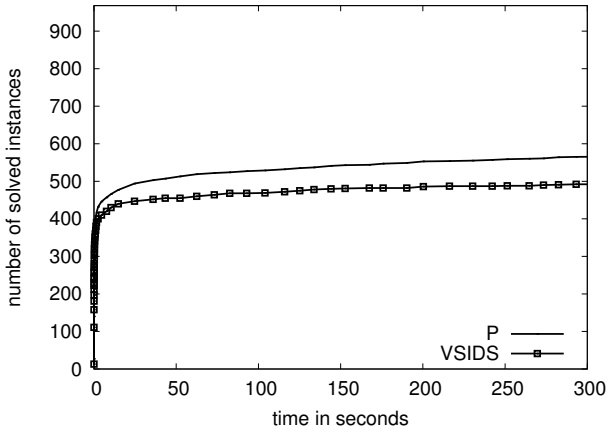

Fig. 4. Number of instances that can be solved in a given time with the sequential strategy

all of the effort is spent on the unsatisfiability tests for formulas right below the parallel length of the shortest parallel plan.

When optimality is not required, we use the planner with its default settings, as described earlier, interleaving the solving of several SAT instances for different plan lengths, thus avoiding the completion of the difficult unsatisfiability proofs. In this scenario any improvements in determining satisfiability, i.e. quickly finding a satisfying assignment corresponding to a plan, becomes much more important. The results of this experiment are given in Fig. 5. In this case the new heuristic has a clear advantage over VSIDS.

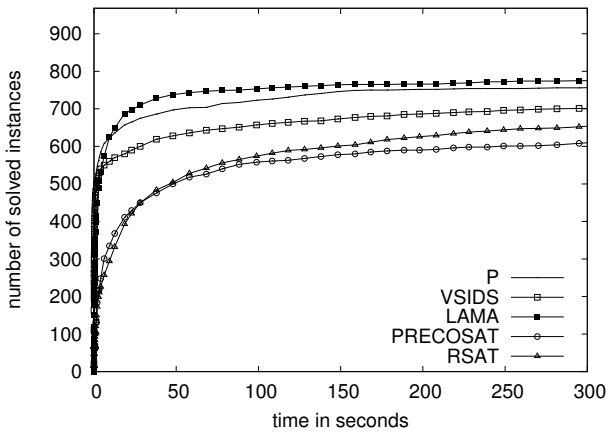

Fig. 5. Number of instances solved by different planners in a given time

The diagram also plots the runtimes of the 2008 International Planning Competition winner LAMA [11][5]. The performance is almost at the same level, in terms of the number of solved problems in a given time, although the sets of solved instances differ quite a bit.

[5] Following instructions from the authors of LAMA, we decreased the maximum invariant generation time to 60 seconds, to match our time out limit 300 seconds.

		VSIDS	P	LAMA	PRECO	RSAT
1998-GRIPPER	20	20	20	20	18	20
1998-MPRIME	20	16	18	20	13	13
1998-MYSTERY	19	16	17	19	16	17
2000-BLOCKS	102	71	85	51	52	64
2000-LOGISTICS	76	76	76	76	74	74
2002-DEPOTS	22	21	21	16	21	21
2002-DRIVERLOG	20	15	20	20	18	18
2002-FREECELL	20	4	5	18	5	4
2002-ZENO	20	18	20	20	17	19
2004-AIRPORT	50	40	42	37	22	26
2004-OPTICAL-TELEG	14	14	14	2	14	14
2004-PHILOSOPHERS	29	29	29	BUG	29	29
2004-PIPESWORLD-NOTANK	50	15	20	44	0	0
2004-PSR-SMALL	50	50	49	50	49	50
2004-SATELLITE	36	29	32	30	23	27
2006-PIPESWORLD	50	9	10	38	8	10
2006-ROVERS	40	40	40	40	25	33
2006-STORAGE	30	29	30	18	23	25
2006-TPP	30	26	26	30	30	29
2006-TRUCKS	30	19	29	8	18	19
2008-ELEVATORS	30	13	30	30	16	17
2008-OPENSTACKS	30	15	11	30	12	12
2008-PARCPRINTER	30	30	30	28	30	30
2008-PEGSOLITAIRE	30	25	21	29	30	28
2008-SCANALYZER	30	19	16	27	14	19
2008-SOKOBAN	30	2	4	18	2	2
2008-TRANSPORT	30	10	12	28	1	3
2008-WOODWORKING	30	30	30	28	30	30
total	968	701	757	775	610	653
average number of actions		82.51	61.22	67.13		

Table 1. Number of instances solved in 300 seconds by benchmark domain

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.) Some earlier planners have roughly the same performance as LAMA but do much worse with hard problems from the phase transition region or even with easy ones [12].

We also break down the results to different benchmark domains, showing the numbers of problem instances solved in 300 seconds by each planner in Table 1, and show the average number of actions for instances solved by both variants of our planner and LAMA. The numbers of actions in the plans LAMA produces are in general about the same, but for the blocks world problems LAMA's plans are substantially worse.

We also compared our results to the winners of the application/industrial category of the SAT competition in 2009 and 2007, Precosat and RSAT [10]. We translated all the test problems into DIMACS CNF for horizon lengths $0, 10, 20, \ldots, 100$ [6] and solved them with a 100 second time limit per instance[7], and then calculated the corresponding Algorithm B runtimes with $\gamma = 0.9$. The Precosat and RSAT runtimes exclude the construction of the CNF formulas and the writing and reading of DIMACS CNF files.

The main reason for the differences between our SAT solver and Precosat and RSAT is preprocessing: for many large SAT instances that we can solve almost immediately, RSAT and Precosat spend considerable time with the preprocessing before starting the search phase. We intend to later investigate better preprocessing strategies for SAT solvers in the planning context. Since the SAT instances for different horizon lengths of one planning instance are closely related, parts of the CNF preprocessing can be shared. This is one of the main avenues to improving preprocessing for planning with SAT.

Precosat is very good with the Peg Solitaire benchmark, presumably because of the preprocessor, but in general it seems that preprocessing in RSAT and Precosat starts to pay off only if there are ten minutes or more available to complete the planning task.

Many of the standard planning benchmarks lead to large SAT problems. The largest SAT problems Precosat solves (within the time bounds explained earlier) are instance 41 of Airport (417476 variables, 92.9 million clauses) and instance 26 of Trucks (926857 variables, 11.3 million clauses). Our planner solves instance 45 of Airport with a completed unsatisfiability test for horizon length 60 (582996 variables and 140.7 million clauses) and a plan for horizon length 70 (679216 variables, 164.1 million clauses). Our planner, with the new heuristic, also solves instance 34 of Satellite, with a plan found for horizon length 40 (8.5 million variables, 29.9 million clauses) backtrack-free in 20.01 seconds excluding translation into SAT and including effort to find unsatisfiability proofs for shorter horizon lengths. These are extreme cases. Most of the instances have less than 1 million variables and at most a couple of million clauses.

[6] We use the step 10 to reduce the time to perform the experiment. This affects the runtimes negligibly. For the blocks world problems we used horizon lengths up to 200.

[7] This is more than enough to determine the planners' runtimes up to time out limit 300 seconds.

6 Related Work

6.1 Earlier Planning Algorithms

Many earlier algorithms add actions to incomplete plans to support an existing goal or subgoal, for example the partial-order planning algorithms [13]. The LPG planner [14] does stochastic local search in the space of incomplete plans with parallel actions similar to the SAT-based approach. LPG's choice of actions to be added in the current incomplete plan is based on the impact of the action on violations of constraints describing the solutions.

Heuristic search [15] has long been an essential for problem solving in AI, but its use in planning was limited until the groundbreaking work of Bonet et al. [16]. Research quickly focused on state-space search guided by heuristics derived from declarative problem descriptions in a generic, problem-independent manner. A main emphasis in the research on classical planning has been in finding better heuristics, with very limited efforts to try something else than state-space search.

The Graphplan algorithm [17] uses backward search, constrained by the planning graph structure which represents approximate (upper bound) reachability information. The action selection of GraphPlan's search may resemble our action selection: given a subgoal l at time t, the choice of an action to reach l is restricted to actions in the planning graph at level $t - 1$. This same constraint on action selection shows up in any extraction of action sequences from exact distance information, for example in BDD-based planning [18] and corresponding model-checking methods. However, the data structures representing the distances (the planning graph or the BDDs) are not used as a heuristic as in our work: when the action choice for achieving l is not restricted by the contents of the planning graph, Graphplan will choose an arbitrary action with l as an effect. Another major difference is of course that our heuristic leverages on the search state of the CDCL algorithm (the learned clauses). This is the main reason why our heuristic, despite its extreme simplicity, is more informative than the more complex heuristics earlier used in AI planning.

6.2 Earlier Planners that Use SAT

The best known planner that uses SAT is BLACKBOX by Kautz and Selman [19]. Rintanen et al. [5] demonstrate that their $\forall$-step semantics encoding is often substantially faster than the BLACKBOX encoding, sometimes by a factor of 20 or more. Both encodings use the same definition of parallel plans. Runtime data in Sideris and Dimopoulos [20] indicates that newer planners in the BLACKBOX family implement slower encodings than BLACKBOX. For example, SATPLAN06 is often twice as slow as BLACKBOX. The only other encoding that is comparable to the $\forall$-step semantics encoding of Rintanen et al. in terms of efficiency and size is the recent factored encoding of Robinson et al. [21].

The more relaxed notion of parallel plans used in our planner, the $\exists$-step semantics [5, 22], allows shorter horizons and smaller formulas, and therefore leads to substantial efficiency improvements. This and parallelized search strategies [6] often mean further one, two or more orders of magnitudes of speed up over other SAT-based planners.

6.3 Domain-Specific Heuristics for SAT Solving

Not much is known about using problem specific heuristics in SAT solving or the workings of SAT solvers when solving planning problems. Beame et al. [8] demonstrate the utility of a problem-specific variable selection heuristic for a clause-learning algorithm solving a combinatorial problem (pebbling formulas.) They demonstrate an improvement in finding unsatisfiability proofs.

The decision variable heuristic proposed in this paper focuses on action variables, and only assigns fact variables at the last stages to complete the assignment. Theoretical results indicate that the efficiency of CDCL is decreased if variable assignments are restricted to a subset of the variables only, even if those variables are sufficient for determining satisfiability and unsatisfiability [23]. It is not clear to us what the practical or theoretical implications of these results are in our setting, more specifically because planning and state-space reachability problems are only a subset of all satisfiability problems. The experiments of Järvisalo and Junttila with instances from bounded-model checking style deadlock detection and LTL model-checking suggest that a CDCL solver with VSIDS restricted to a subset of decision variables fares worse than VSIDS without the restriction. In contrast, we have a heuristic that has such limitations, and the unlimited decision heuristic (VSIDS) with our test material fares in general worse. Of course, the results of Järvisalo and Junttila could be seen as saying that there are more effective variants of our heuristic which sometimes also choose fact variables as decision variables. Further, all known restrictions on SAT solving efficiency (in a given proof system) apply to unsatisfiability proofs only, which are not the focus of our work.

7 Conclusions and Future Work

The contribution of this paper is a simple yet powerful variable selection strategy for clause-learning SAT solvers that solve AI planning problems, as well as an empirical demonstration that the strategy outperforms VSIDS for standard planning benchmarks when no proofs of the optimality of the horizon length is required. A main additional benefit over VSIDS is that the variable selection strategy is understandable in terms of the planning problem, and that there are obvious and intuitive avenues for developing it further. The basic variable selection strategy is particularly promising because the features that makes it strong are largely complementary to the important features of VSIDS, making it possible to combine them, for example by adopting the weights of literals from VSIDS to the planning heuristic. This is a focus of future work, as is finding ways of doing the unsatisfiability proofs more efficiently.

Immediate improvement opportunities arise for example from the possibility of changing the order in which the top-level goals and the preconditions of an action are considered. Our initial experimentation in this area has demonstrated that the approach can be dramatically strengthened further, leading to planners that substantially outperform modern planners such as LAMA.

The main ideas in this work are quite general, and could be easily adapted to other applications of SAT and constraint-satisfaction, for example model-checking [24] and diagnosis [25], and of more expressive logics, such as QBF [26].

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. We thank Hector Geffner and Patrik Haslum for comments on earlier versions of this paper.

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. 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
3. Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence **129**(1-2) (2001) 5–33
4. Cook, S.A.: The complexity of theorem proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. (1971) 151–158
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. 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
7. Rintanen, J.: Regression for classical and nondeterministic planning. In Ghallab, M., Spyropoulos, C.D., Fakotakis, N., eds.: ECAI 2008. Proceedings of the 18th European Conference on Artificial Intelligence, IOS Press (2008) 568–571
8. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research **22** (2004) 319–351
9. Mitchell, D.G.: A SAT solver primer. EATCS Bulletin **85** (February 2005) 112–133
10. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., Sakallah, K.A., eds.: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007). Volume 4501 of Lecture Notes in Computer Science., Springer-Verlag (2007) 294–299
11. 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
12. Rintanen, J.: Phase transitions in classical planning: an experimental study. In Zilberstein, S., Koehler, J., Koenig, S., eds.: ICAPS 2004. Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling, AAAI Press (2004) 101–110
13. McAllester, D.A., Rosenblitt, D.: Systematic nonlinear planning. In: Proceedings of the 9th National Conference on Artificial Intelligence. Volume 2., AAAI Press / The MIT Press (1991) 634–639
14. Gerevini, A., Serina, I.: Planning as propositional CSP: from Walksat to local search techniques for action graphs. Constraints Journal **8** (2003) 389–413
15. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Company (1984)
16. Bonet, B., Loerincs, G., Geffner, H.: A robust and fast action selection mechanism for planning. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97), AAAI Press (1997) 714–719

17. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial Intelligence **90**(1-2) (1997) 281–300
18. Cimatti, A., Giunchiglia, E., Giunchiglia, F., Traverso, P.: Planning via model checking: a decision procedure for $\mathcal{AR}$. In Steel, S., Alami, R., eds.: Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97). Number 1348 in Lecture Notes in Computer Science, Springer-Verlag (1997) 130–142
19. Kautz, H., Selman, B.: Unifying SAT-based and graph-based planning. In Dean, T., ed.: Proceedings of the 16th International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers (1999) 318–325
20. Sideris, A., Dimopoulos, Y.: Constraint propagation in propositional planning. In: ICAPS 2010. Proceedings of the Twentieth International Conference on Automated Planning and Scheduling, AAAI Press (2010) 153–160
21. Robinson, N., Gretton, C., Pham, D.N., Sattar, A.: SAT-based parallel planning using a split representation of actions. In Gerevini, A., Howe, A., Cesta, A., Refanidis, I., eds.: ICAPS 2009. Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling, AAAI Press (2009) 281–288
22. 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
23. Järvisalo, M., Junttila, T.: Limitations of restricted branching in clause learning. Constraints Journal **14** (2009) 325–356
24. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In Cleaveland, W.R., ed.: Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99. Volume 1579 of Lecture Notes in Computer Science., Springer-Verlag (1999) 193–207
25. Grastien, A., Anbulagan, Rintanen, J., Kelareva, E.: Diagnosis of discrete-event systems using satisfiability algorithms. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07), AAAI Press (2007) 305–310
26. Rintanen, J.: Asymptotically optimal encodings of conformant planning in QBF. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07), AAAI Press (2007) 1045–1050