A Planning Algorithm not based on Directional Search

Jussi Rintanen
Universität Ulm
Fakultät für Informatik
Albert-Einstein-Allee
D-89069 Ulm, Germany

Abstract

The initiative in STRIPS planning has recently been taken by work on propositional satisfiability. Best current planners, like Graphplan, and earlier planners originating in the partial-order or refinement planning community have proved in many cases to be inferior to general-purpose satisfiability algorithms in solving planning problems. However, no explanation of the success of programs like Walksat or relsat in planning has been offered. In this paper we discuss a simple planning algorithm that reconstructs the planner in the background of the SAT/CSP approach.

1 INTRODUCTION

Many of the recent interesting results in AI planning did not originate in traditional planning research, but in work on algorithms for checking the satisfiability of propositional formulae. STRIPS planning problems have been used as benchmarks to test SAT algorithms based on greedy local search [Kautz and Selman, 1992; Kautz and Selman, 1996], and new developments [Bayardo, Jr. and Schrag, 1997] of the well-known Davis-Putnam procedure. As a contribution to planning research, these SAT algorithms have proved to be in many cases orders of magnitude faster planners [Kautz and Selman, 1996] than algorithms specifically designed for planning, thereby pointing out the possibility of dramatic improvements and new interesting lines of research.

Planning by satisfiability was first investigated by Kautz and Selman [1992] by means of a stochastic search algorithm. In this approach, problem instances are translated to sets of propositional formulae that include frame axioms, formulae describing the effects the operators have, and facts that hold in the initial and goal state. The satisfiability algorithm finds an assignment of truth-values to the propositions, and a plan is obtained from the propositions that correspond to operator applications. Interestingly, there are several ways to express the frame axioms and to make the set of formulae more concise, which has led to a thread of research on the quantitative properties of different encodings [Kautz and Selman, 1996; Kautz *et al.*, 1996; Ernst *et al.*, 1997].

Parallel to advances in solving satisfiability problems by stochastic search, improvements to the well-known Davis-Putnam procedure [Davis *et al.*, 1962] have been discovered. State-of-the-art implementations include Crawford's [1996] tableau and Freeman's [1995] Posit. Techniques that have made it possible to solve planning problems of the same difficulty with the Davis-Putnam procedure as with *Walksat* [Kautz and Selman, 1996] include lookback techniques from constraint satisfaction research [Bayardo, Jr. and Schrag, 1997] and heuristics based on unit resolution [Li and Anbulagan, 1997].

In this paper, we directly apply techniques from satisfiability algorithms to STRIPS planning, bypassing the route via translations to SAT. This paper, however, is not about satisfiability testing, as the algorithm we present can be and should be completely understood in terms of planning per se. It is important to devise this algorithm for several reasons. The functioning of SAT algorithms when they are solving planning problems has not been analyzed in earlier research, which has kept the reason for the success of these algorithms from being widely known. For example, the role of various details of problem encodings, like the use of invariants, has been given only a quantitative explanation as a factor that improves the runtimes of SAT algorithms. Furthermore, it is important to identify the boundary between satisfiability testing and planning. We believe that there are a lot of techniques that can improve the performance of planning algorithms, but that are not likely to be discovered by research on propositional satisfiability. To find the boundary between the two fields and to identify lines of research relevant to planning, it is necessary to understand what a SAT algorithm is able to do when it is trying to find

a plan.

2 WORKINGS OF A SAT-BASED PLANNER

In this section we illustrate what takes place when satisfiability algorithms like the Davis-Putnam procedure solve a planning problem represented as propositional formulae. The representation of planning as satisfiability problems [Kautz and Selman, 1992; Kautz *et al.*, 1996] consists of the frame axioms, axioms describing the preconditions and effects of operators, and formulae describing the initial state and the goal. We take as an example the axiomatization from [Kautz *et al.*, 1996] with explanatory frame axioms [Haas, 1987; Schubert, 1990] and parallel operations.

For a fluent l that can be made true by operators $o_1, \ldots, o_n$, explanatory frame axioms are

$$(\overline{l^t} \wedge l^{t+1}) \rightarrow (o_1^t \vee \cdots \vee o_n^t) \qquad (1)$$

where $t \in \{0, \ldots, f\}$ is an integer referring to a point of time and f is the plan length. An operator application implies its preconditions and postconditions, that is, if the preconditions of o are $l_1, \ldots, l_n$ and the postconditions $l'_1, \ldots, l'_{n'}$, then we have the axiom

$$o^t \rightarrow (l_1^t \wedge \cdots \wedge l_n^t \wedge l_1'^{t+1} \wedge \cdots \wedge l_{n'}'^{+1}). \qquad (2)$$

Fluents $l_1, \ldots, l_n$ true at the initial state are encoded simply as the unit clauses $l_1^0, \ldots, l_n^0$, and the goal fluents $g_1, \ldots, g_m$ as the unit clauses $g_1^f, \ldots, g_m^f$. Finally, there are axioms that prevent the simultaneous application of operators that are mutually dependent.

Most of the inferences in the Davis-Putnam procedure when it is solving a planning problem are based on unit resolution and unit subsumption. From a unit clause o^t the preconditions at t and postconditions at $t+1$ are obtained. An important pattern of inference is started by unit clauses $\neg o^t$. First, clauses from axioms in Eq. 2 are deleted by unit subsumption. Second, and more importantly, unit resolution together with frame axioms (Eq. 1) may force the truth-value of a fluent to persist across a step of time or to force the application of an operator. If $\neg o^t$ was the last literal in the consequent we have $l^t \vee \overline{l^{t+1}}$, that is, if the fluent l is false at t then it is false also at $t+1$. Depending on the presence of $\overline{l^t}$ or l^{t+1}, this may yield $\overline{l^{t+1}}$ or l^t. If o^t was next to the last literal in the consequent and we already had $\overline{l^t}$ and l^{t+1}, unit resolution forces the application of the remaining operator o' at t. Similarly, a unit clause representing a fluent together with a frame axiom 1 may propagate its truth-value one step backward or forward or force an operator application.

The above inference patterns are very powerful in making forward inferences from the initial state. As the initial state in STRIPS planning determines the truth-values of all fluents, unit resolution directly lets us conclude which operators are not applicable at time 0 because their preconditions are false. For many fluents this – like shown above – allows to infer their persistence, which in turn yields the inapplicability of other operators at time 1, and so on.

After unit simplifications, the Davis-Putnam procedure does a case analysis on a literal l, in one case adding l to the clause set and in the other $\overline{l}$, in both cases enabling further inferences by unit simplifications.

The algorithms by Kautz and Selman [1992; 1996] based on greedy local search work quite unlike the Davis-Putnam procedure. These algorithms start from a randomly chosen truth-assignment that is repeatedly modified by reversing truth-values of literals so that more clauses are satisfied. However, flips roughly corresponding to unit simplifications are likely to be made by *Walksat* as they increase the number of satisfied clauses.

3 THE ALGORITHM

In this section we give a new algorithm for STRIPS planning. The algorithm is similar in structure to the Davis-Putnam procedure for propositional satisfiability. Each recursive call consists of applying a set of efficient inference rules, followed by a case analysis. The main procedure of the algorithm, given in Figure 3 uses a plan length as a parameter, and it is iteratively called for all plan lengths starting from 0 until a solution is found. Currently we employ no method for detecting insolvability, so the planner runs forever on problem instances that have no solution. The proofs of soundness and completeness of the algorithm are straightforward.

A distinguishing characteristic of the algorithm is that – unlike most earlier planning algorithms – the search does not start from the goal state and proceed towards the initial state. Instead, operator applications at any points of time may be chosen for the case analysis that makes the search tree branch. In one of the subtrees the operator is applied, and in the other it is not. This is in strong contrast with Graphplan [Blum and Furst, 1995] that generates a branch for every minimal set of operators that produces the current subgoal, and with algorithms like SNLP [McAllester and Rosenblitt, 1991] that branch on possible ways of extending a partially ordered plan so that a threat is removed or an operation in the incomplete plan has its precondition fulfilled.

An inference technique from satisfiability testing we have found useful is *failed literal detection*, proposed by Freeman [1995] and investigated by Li and Anbulagan [1997]. We discuss it in the section on invariants where we show that it enables backward inferences from the goal state.

```
procedure plan()
   allocate space for local arrays op' and prop';
   if applyrules() then return false; end if
   if op[t,o] = Unknown for no t,o then return true; end if
failedLiteralDetection:
   foreach operator o and time t such that op[t,o] = Unknown do
      op' := op; prop' := prop;
      op[t,o] := True;
      if applyrules() then (* Contradiction was derived. *)
         op := op'; prop := prop';
         op[t,o] := False;
         if applyrules() then return false;
      else
         if score for op,prop improves the previous best score
         then t1 := t; o1 := o; p1 := 1; end if
         op := op'; prop := prop';
         op[t,o] := False;
         if applyrules() then (* Contradiction was derived. *)
            op := op'; prop := prop';
            op[t,o] := True;
            applyrules();
         else
            if score for op,prop improves the previous best score
            then t1 := t; o1 := o; p1 := 0; end if
            op := op'; prop := prop';
         end if
      end if
   end foreach
   if operator applications were derived
   then goto failedLiteralDetection end if;
   op' := op; prop' := prop;
   if p1 = 1 then op[t1,o1] := True else op[t1,o1] := False; end if
   if plan() then return true; end if
   op := op'; prop := prop';
   if p1 = 1 then op[t1,o1] := False else op[t1,o1] := True; end if
   return plan();
end
```

Figure 1: The planning algorithm

This technique is also the basis for selecting an operator for branching. Failed literal detection proceeds by attempting to derive literals by proof-by-contradiction: assume the literal is true, and if contradiction can be derived by unit simplifications, conclude the negation of the literal, otherwise estimate the usefulness of branching on the literal in terms of the number of short clauses it yielded. In our case, we assume that a certain operator is or is not applied at a certain moment of time, and for branching we select the operator (in)application that reduces the number of unknown operation applications and proposition values most.

The main procedure of the algorithm is given in Figure 3. Before calling it, the elements of arrays `op` and `prop` are initialized to Unknown, `prop[0,p]` for all p is assigned the truth-value of proposition p in the initial state, and similarly for `prop[len,p]` and the goal state, where `len` is the plan length currently considered. If the procedure returns true, a plan can be read from the array `op`.

The function `applyrules` performs the following inferences and returns *true* if a contradiction was detected, that is, an assignment of True to an element of `op` or `prop` that had value False was attempted, or vice versa.

1. If operator o is assigned true at t, then make the preconditions of o true at t and the postconditions true at $t+1$.
2. If all operators with l in the postcondition are assigned false at t, then do the following. If l is false at t then make l false at $t+1$. If l is true at $t+1$ then make l true at t.
3. If operator o is assigned true at t, assign false at t to all operators that make the precondition of o false, and assign false at t to all operators the precondition of which is made false by o.

The second rule encodes the assumption that only operations change the value of propositions. The third rule makes the planner a partial-order planner: operators that can be executed in any order without affecting the outcome may be executed simultaneously.

Additional rules are used for speeding up inferences – especially to reduce the need for failed literal detection which is expensive – even though they are not required for the correctness of the algorithm. For example, if a proposition changes truth-value at time t and all but one operator making the change have been chosen to be not applied, the remaining operator has to be applied. And if the precondition is false at t or a postcondition is false at $t+1$, then the operator is not applied at t. It is easy to verify that these inferences correspond to the ones sanctioned by unit resolution in planners based on SAT, for example with the encoding in [Kautz *et al.*, 1996] with parallel operations and explanatory frame axioms.

Note that like the Davis-Putnam procedure, as long as case analysis is not needed, the algorithm runs in polynomial time.

As an alternative to doing failed literal detection and branching on operator applications, it is also possible to do it on propositions, or both operator applications and propositions. What is the best alternative depends on the properties of the class of planning problems to be solved. Doing failed literal detection on both operator applications and propositions of course prunes the search space at least as much as the other alternatives, but it is more expensive.

4 INVARIANTS

The basic algorithm given in the previous section sanctions most of the desired inferences from the initial state. For example in the blocks world domain, if there is a stack of n

blocks on top of block A, we can infer that after n moves A is still in its initial position. We would like to make similar inferences starting from the goal state. For example, if A is on top of B and B is on top of C at time t, then B is on top of C at $t-1$. To make this inference, we need to show that it is not possible to move B on top of C at $t-1$. This inference uses failed literal detection. Assume that B is moved on top of C at $t-1$. The precondition of this operation is that B and C are clear at $t-1$. The moving of A on B cannot be done in parallel with the move of B on C because the former makes the precondition of the latter false. Hence A cannot be moved on B at $t-1$. Therefore A is on B at $t-1$. But then B cannot be clear at $t-1$, which contradicts the precondition of moving B on C.

What is missing in the algorithm in the previous section is the last step that whenever A is on B, B cannot be clear. This is a fact that is not explicit in the facts that A is on B and B is on C. The goal state and many intermediate states are incomplete in that they do not specify the truth-values of all propositions. To extend the state descriptions we need *invariants* derived from the operators and the initial state[1]. In the above example the relevant invariant says that either B is not clear or A is not on B. Most of the work on SAT-based planning has used this kind of invariants to speed up plan search [Kautz and Selman, 1992; Kautz and Selman, 1996]. Invariants (usually incompletely) characterize the set of states reachable from the initial state.

The invariants used in the algorithm consist of 2-literal clauses, which is the relevant form of invariants in most of the benchmark domains, like the blocks world, the rocket domain, and the logistics domain. In the blocks world the only invariants that cannot be represented as 2-literal clauses are those that state that the *on* relation is acyclic and if a block is not clear then some block has to be on top of it. Invariants are incorporated in the algorithm as the following rule: if $l \vee m$ is an invariant and l becomes False at t, then make m True at t.

4.1 COMPUTATION OF INVARIANTS

We have devised an algorithm for computing 2-literal invariants. Its structure is similar to the invariant algorithms in computer-aided verification [Bensalem *et al.*, 1996]. The algorithm starts with a candidate invariant Q', consisting of all the 2-literal clauses true in the initial state, and weakening it repeatedly by removing clauses that are made false by an operator application. This is formalized as the computation of fixpoints of monotonic functions R_O. The definition of R_O for a set of operators $O = \{o_1, \ldots, o_n\}$ is as follows.

$$
\begin{aligned}
R_O(V) &= F_{o_1}(F_{o_2}(\cdots F_{o_n}(V)\cdots)) \\
F_{\langle p,e\rangle}(V) &= \begin{cases} V, \text{ if } V \cup p \models \bot, \text{ and otherwise} \\ \{a \vee b \in V | \neg a \notin e \text{ or } b \in U(V,p,e), \\ \qquad \neg b \notin e \text{ or } a \in U(V,p,e)\} \end{cases} \\
U(V,p,e) &= \{l \in L | V \cup p \models l\} \backslash \{\bar{l} | l \in e\} \cup e
\end{aligned}
$$

The function F_o takes a set of clauses, and deletes the ones the truth of which the operator o does not preserve. The function U performs an update. It computes the set of literals $U(V,p,e)$ that are true in all states that result from changing the literals in e true in states that satisfy V and p, where V is a set of 2-literal clauses. Because R_O is monotonic and the universe is finite, there is a fixpoint of R_0 that is obtained by a finite number $n \geq 0$ of iterations as $R_O^n(V_0) = R_O^{n+1}(V_0)$. Here $V_0 = \{a \vee b | a \in I, b \in L\}$ consists of the 2-literal clauses true in the initial state I and L is the set of all literals.

The algorithm does not compute all 2-literal invariants. This is because a 2-literal representation $R_O^i(V_0)$ of a subset of reachable states is not accurate ($s \models R_O^i(V_0)$ does not guarantee that s is reachable), and F_o considers applications of o in unreachable states that result in unreachable states where some 2-literal invariants are violated. In many problem domains, for example in the ones mentioned in this paper, this does not happen.

Lemma 4.1 *For any set of operators $O = \{o_1, \ldots, o_n\}$, the functions F_{o_i}, and consequently the function R_O, are monotonic.*

Lemma 4.2 *Let p and e be sets of literals, V be a set of 2-literal clauses, s be a model such that $s \models V \cup p$, and s' a model obtained from s by setting the literals in e true. Then $s' \models U(V,p,e)$.*

Lemma 4.3 *For all states s and sets of 2-literal clauses V such that $s \models V$ and s' is a successor of s (under the application of an operator in O), $s' \models R_O(V)$.*

Proof: Now $s \models p$ and $s' = s \backslash \{\bar{l} | l \in e\} \cup e$ for some operator $\langle p, e\rangle$. Let $a \vee b$ be any member of $R_O(V)$. Because $R_O(V) \subseteq V$ and $s \models V$, $s \models a \vee b$. Assume $\neg a \notin e$ and $\neg b \notin e$. Clearly $s' \models a \vee b$. So assume $\neg a \in e$ or $\neg b \in e$. Because of symmetry it suffices to consider the case $\neg a \in e$. Because $a \vee b \in R_O(V)$, for all $\langle p', e'\rangle \in O$ such that $V \cup p' \not\models \bot$, $b \in U(V,p',e')$. This holds also for $p' = p, e' = e$. By Lemma 4.2 $s' \models U(V,p,e)$, and hence $s' \models b$. Therefore $s' \models R_O(V)$. □

Theorem 4.4 *Let V be a fixpoint of R_O such that $V \subseteq V_0$. For all states s reachable from the initial state I with applications of operators in O, $s \models V$.*

[1] The mutual exclusion relations on literals in Graphplan [Blum and Furst, 1995] are a superset of the 2-literal invariants.

```
                0123456                    012345
          atR2JFK T FFFFF          loadR1R2JFK  FFFFF
          atR1JFK T FFFFF       loadR1R2London FFF FF
           inR1R2 F  F  F        loadR1R2Paris FFFFFF
       atR2London FF T FF          loadR2R1JFK  FFFFF
       atR1London FF T FF       loadR2R1London FFF FF
        atR2Paris FFFFF T        loadR2R1Paris FFFFFF
        atR1Paris FFFFF T        unloadR1R2JFK FFFFFF
           inR2R1 F  F  F    unloadR1R2London FF FFF
           fuelR1 TT   FF     unloadR1R2Paris FFFFF
    connJFKLondon TTTTTTT       unloadR2R1JFK FFFFFF
     connJFKParis FFFFFFF    unloadR2R1London FF FFF
   connLondonJFK FFFFFFF     unloadR2R1Paris FFFFF
connLondonParis TTTTTTT     moveR1JFKLondon F FFFF
     connParisJFK FFFFFFF     moveR1JFKParis FFFFFF
connParisLondon FFFFFFF     moveR1LondonJFK FFFFFF
           fuelR2 TT   FF   moveR1LondonParis FFFF F
                              moveR1ParisJFK FFFFFF
                            moveR1ParisLondon FFFFFF
                              moveR2JFKLondon F FFFF
                              moveR2JFKParis FFFFFF
                             moveR2LondonJFK FFFFFF
                            moveR2LondonParis FFFF F
                              moveR2ParisJFK FFFFFF
                            moveR2ParisLondon FFFFFF
```

Figure 2: An intermediate stage in finding a plan

Proof: Because by Lemma 4.1 R_O is monotonic, there is a fixpoint V of R_O such that $V \subseteq V_0$. Because s is reachable from I, there is a finite sequence $o_1, o_2, \ldots, o_m$ of operators in O and states $s_0, s_1, \ldots, s_m$ such that $I = s_0$, $s = s_m$ and s_i is obtained from s_{i-1} by the application of o_i. Because $I \models V_0$, by repeated applications of Lemma 4.3 $s_i \models R_O^i(V_0)$ for all $i \in \{0, \ldots, m\}$. Because $V \subseteq R_O^i(V_0)$, $s_i \models V$ for all $i \in \{0, ..., m\}$. Hence $s = s_m \models V$. □

The contents of planning graphs in GraphPlan resemble the intermediate stages in the computation of invariants in our algorithm.

5 AN EXAMPLE

Figure 5 illustrates the solution of a problem from the rocket domain of [Blum and Furst, 1995]. Initially two rockets are at JFK. In the goal state the rockets are in Paris. Flights are possible only from JFK to London and from London to Paris. The algorithm detects the inexistence of plans of lengths 1 to 5 without search only by using the easy polynomial-time inferences. For plan length 6, when the algorithm has reached its first branching point, the state and the known operations are as shown in Figure 5. The remaining possibilities represent the two possible plans, load one of the rockets into the other, fly to London (consuming the fuel from the rocket), unload, load the rocket with no fuel into the one with fuel, fly to Paris, and unload again.

Once the planner decides (arbitrarily) to try to load R1 into R2 at JFK at time 0, the rest of the operations are forced and no contradictions are obtained: R2 has to be flown to London, R1 has to be unloaded from R2, R2 has to be loaded into R1, R1 has to be flown to Paris, and finally R2 has to be unloaded from R1. On the same problem Graphplan generates a search tree with several nodes, because it does not infer anything from the goal state.

6 EXPERIMENTAL RESULTS AND RELATED WORK

We have implemented the algorithm in C, and evaluated it on a number of examples. Tables 1 and 2 lists the runtimes (in seconds) of our planner (the R column), Blum and Furst's *Graphplan* [1995], and the satisfiability programs *Walksat* [Kautz and Selman, 1996], *satz* [Li and Anbulagan, 1997], and *relsat* [Bayardo, Jr. and Schrag, 1997] on a number of benchmark problems from [Kautz and Selman, 1996]. All the SAT programs were run with problem encodings devised by Kautz and Selman. We ran the benchmarks using Li and Anbulagan's *satz* as they do not give runtimes for these benchmarks in their paper. The runs on *satz* and on our planner were on a Sun Ultra 2 workstation. Kautz and Selman ran Graphplan and Walksat on SGI Challenger, and Bayardo and Schrag ran their program on SPARC-10. Blank means that we found no runtime in the papers mentioned. Whenever a runtime was reported for several SAT encodings, we give here the lowest time. For our runs we give the total time taken by our planner, and in parentheses the time taken by the last stage, that is, the time it takes to find a shortest plan if it is known what the length of this plan is. The times in parentheses for the satisfiability algorithms are for the last stage as given in the respective papers. The times for identifying lengths of shortest plans are not given in the satisfiability papers presumably because the algorithms by Kautz and Selman – who started the work on planning by satisfiability – are not capable of determining the inexistence of plans of certain length, that is the unsatisfiability of a set of clauses, with certainty.

The runtimes of our program for the blocks world problems are worse than those by the satisfiability algorithms. Unlike in the other benchmarks, the constraints on the candidate plans at the last stages are very loose, and our reliance on inferences based on failed literal detection does not pay off. The solution of these benchmarks heavily relies on good branching heuristics, which in the case of the satisfiability algorithms are more successful. The more compact encoding of these problems by Kautz and Selman is not the cause of this difference. For example *satz* generates a search tree with only 6 choice nodes for bw-large.c. If we hardwire our program to make the same choices, the solution is found immediately without search. The parallel versions of these blocks world problems are more constrained and solutions are found quickly, for example, solving bw-large.d even with standard parallelism without post-serializability [Dimopoulos *et al.*, 1997] takes 100 seconds. The number of

Table 1: Runtimes of some benchmark problems

problem	R	BF95	KS96
rocket.ext.a	3.8 (1.4)	520	(0.1)
rocket.ext.b	2.7 (0.6)	2337	(0.2)
bw-large.b	74.1 (33.0)	27115	(22)
bw-large.c	7144 (2350)	$>$ 10 h	(564)
bw-large.d	$>$ 10 h	$>$ 10 h	(937)
logistics.a	2.2 (1.4)	6743	(2.7)
logistics.b	91.4 (1.9)	2893	(1.6)
logistics.c	871.2 (3.7)	$>$ 10 h	(1.9)

Table 2: Runtimes of some benchmark problems

problem	R	LA97	BS97
rocket.ext.a	3.8 (1.4)	(0.07)	
rocket.ext.b	2.7 (0.6)	(0.06)	
bw-large.b	74.1 (33.0)	(0.3)	
bw-large.c	7144 (2350)	(1.6)	(11.9)
bw-large.d	$>$ 10 h	(218.9)	(813.3)
logistics.a	2.2 (1.4)	(4.2)	(4.1)
logistics.b	91.4 (1.9)	(0.8)	(16.6)
logistics.c	871.2 (3.7)	(324.2)	(90.3)

choice nodes in the search tree is 5.

Table 4 lists runtimes on a number of benchmarks from Dimopoulos et al. [1997]. Dimopoulos et al. encode planning problems as nonmonotonic propositional logic programs, and find plans by using a program (smodels [Niemelä and Simons, 1996]) that computes the stable models of these programs. The runtimes for Graphplan are by Dimopoulos et al. and all runs by them were on Sparc Ultra. The times in the DNK97 column are for finding a shortest plan when the length of that plan is given as input. This corresponds to the time in our column in parentheses. Like above, our time outside the parentheses is the total runtime. The runtimes in Tables 1 and 2 are not directly comparable to the those in Table 4. The blocks world runtimes are for a parallel encoding, and further, in all the benchmarks, it is not required that parallel operations can be executed in all possible orders, but that there is at least one order in which they can be executed [Dimopoulos *et al.*, 1997]. For example loading or unloading a truck can be performed in parallel with moving it. With this relaxation shortest plans are shorter, the constraints on possible plans are tighter, and search space gets pruned dramatically. Graphplan runtimes are with standard parallelism.

Table 3: Sizes of the search trees

problem	choice nodes			plan	
	total	last	failed	length	ops
rocketext.a	31	7	3	7	34
rocketext.b	26	4	0	7	30
bw-large.b	2	2	0	18	18
bw-large.c	1573	789	785	28	28
bw-large.d					
logistics.a	14	13	0	11	54
logistics.b	438	25	1	13	47
logistics.c	3119	22	1	13	65

Table 4: Runtimes of some benchmark problems

problem	R	BF95	DNK97
bw-large.c	5.9 (5.5)	1830	(190)
bw-large.d	20.4 (19.7)	219600	(157)
bw-large.e	30.4 (29.3)		(365)
logistics.a	0.2 (0.1)	6743	
logistics.b	1.2 (0.2)	2893	
logistics.c	2.7 (0.3)	$>$ 10 h	(18)
train.a	22.7 (21.3)		(647)
train.b	100.1 (96.2)		(1261)
train.c	685.0 (109.2)		(5989)

Table 5: Sizes of the search trees

problem	choice nodes			plan	
	total	last	failed	length	ops
bw-large.c	8	8	0	6	21
bw-large.d	10	10	0	6	32
bw-large.e	10	10	0	7	37
logistics.a	13	13	0	7	68
logistics.b	30	21	0	8	56
logistics.c	39	18	0	8	68
train.a	54	54	45	8	39
train.b	53	53	43	7	34
train.c	410	65	52	8	42

Our runtimes do not include the expansion of operator schemata to sets of propositional operators nor the computation of invariants. These computations take between a second and a minute (bw-large.e) for the problems mentioned. The program performing this is written in Standard ML. An efficient implementation in C would run much faster. Our runtimes include, unlike the runtimes for satisfiability algorithms, a preprocessing stage a major part of which includes the precomputation of pairs of operators that interfere and therefore cannot be applied in parallel.

Tables 3 and 5 list statistics on the sizes of search trees and solutions with our algorithm. The first column is the total number of choice nodes in the search trees, that is, nodes

that correspond to case analyses on operations or propositions. The second column is the number of choice nodes at the last stage, and the third column is the number of failed choices at the last stage (at earlier stages all choices fail.) The third column being zero (or small in comparison to the second column) indicates that the heuristics based on failed literal detection was successful in choosing choice points for case analysis. The fourth column gives the length of the plan found, and the fifth the number of operations in the plan. Plan lengths for the other planning and satisfiability algorithms are the same except with the sequential version of blocks worlds problems, where our plans are twice the length of plans obtained with the satisfiability algorithms. This difference is due to the encoding by Kautz and Selman's, that perform a pickup and a putdown operation at the same point of time. The numbers of operations for different algorithms in many cases differ, as the solutions usually are not unique.

An interesting observation is that the solution of many of the problems requires hardly any search. For example on the parallel version of bw-large.e the algorithm detects the inexistence of plans of length 6 and below simply on the basis of what can be inferred from the initial and goal states[2]. All this computation is polynomial time on the size of the planning problem. On plans of length 7 the search tree contains 10 choice nodes, all of which have only one child, that is, the choice was successful. On train.b, on the other hand, finding the plan involves search through several dozens of nodes, but all the failed branches are of height 1, that is, a wrong operator is chosen for application but the error is soon detected by failed literal detection without generating more nodes to the search tree. Search trees of this structure are of polynomial size, and therefore represent polynomial time computation.

Graphplan [Blum and Furst, 1995] is the state-of-the-art STRIPS planner. The main differences between Graphplan and the rest of the work discussed here are that Graphplan is based on backward search from the goal state, and that the mutual exclusion relations that serve the same purpose as invariants are only used for abandoning subgoals not reachable from the initial state, and not for making inferences about the state of affairs at a particular moment of time. These differences seem to be the decisive factor that on many benchmarks makes Graphplan fare worse than the other programs. Note that the information in Graphplan's planning graph until "leveling off" [Blum and Furst, 1995] is stronger than the 2-literal invariants. However, most of this information is inferred by unit simplifications in SAT algorithms and the inference rules in our algorithm. Graphplan employs memoization to avoid repeatedly making the same mistakes, but this does not give an advantage over the rest of the approaches that do not have memoization (excluding *relsat*) on the kind of problems considered here.

[2] Also Kautz and Selman's satplan [1996] sometimes finds out during problem instance generation that there is no solution of a certain length, simply by unit resolution and unit subsumption.

7 CONCLUSIONS

An important research topic is the identification of better heuristics in choosing branching operators. Our current implementation simply chooses the operator (in)application that maximizes the number of propositions or operators that are assigned a truth-value. This is roughly what *satz* does, with the exception that we have nothing that corresponds to counting the number of short clauses, that is, no estimation on the possibilities to use the polynomial time inferences at the next steps is done. In many cases when the constraints imposed by the initial and goal states are not very tight our heuristic chooses an operation that is able to force the longest sequence of (nonsensical) operations which often does not contribute to bridging the gap between the initial state and the goal. Interestingly, when using the algorithm to find a long solution to a problem that has a short solution, this behavior often very quickly and successfully produces the desired result.

In addition to improved branching heuristics, there are prospects of speeding up the algorithm by a more aggressive use of invariants. In addition to 2-literal invariants, some problem domains would benefit from n-literal invariants for $n \geq 3$. Our invariant algorithm can easily be generalized to the $n \geq 3$ case, but it is not clear how it can be made efficient. Also, our algorithm does not take advantage of the fact that usually the set of ground operators is highly structured, and it is often possible to compute the invariants more efficiently directly from a schematic representation of the operators. The representation of a set of candidate invariants in the description of our invariant algorithm consumes a lot of memory (our implementation uses a slightly better representation), and therefore a more concise representation should be used instead, for example one based on binary decision diagrams [Bryant, 1992].

Acknowledgements

We thank Chu Min Li for kindly providing us copies of satz and related satisfiability programs, and Harald Rueß for comments and advice. Financial support from Suomalainen tiedeakatemia and Suomen Kulttuurirahasto is gratefully acknowledged.

References

Bayardo, Jr., R. J. and R. C. Schrag: Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 203–208, 1997.

Bensalem, S., Y. Lakhnech, and H. Saidi: Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, USA, July 1996. Springer Verlag.

Blum, A. L. and M. L. Furst: Fast planning through planning graph analysis. In C. S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1636–1642. Morgan Kaufmann Publishers, 1995.

Bryant, R. E.: Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

Crawford, J. M. and L. D. Auton: Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81:31–57, 1996.

Davis, M., G. Logemann, and D. Loveland: A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

Dimopoulos, Y., B. Nebel, and J. Köhler: Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the European Conference on Planning (ECP-97)*. Springer-Verlag, 1997.

Ernst, M., T. Millstein, and D. S. Weld: Tradeoffs in automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.

Freeman, J. W.: *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

Haas, A. R.: The case for domain-specific frame axioms. In F. M. Brown, editor, *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, pages 343–348. Morgan Kaufmann Publishers, 1987.

Kautz, H. and B. Selman: Planning as satisfiability. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, Wien, Austria, 1992. John Wiley & Sons.

Kautz, H. and B. Selman: Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eight Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, California, August 1996. AAAI Press / The MIT Press.

Kautz, H., D. McAllester, and B. Selman: Encoding plans in propositional logic. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 374–385. Morgan Kaufmann Publishers, 1996.

Li, C. M. and Anbulagan: Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, August 1997.

McAllester, D. A. and D. Rosenblitt: Systematic nonlinear planning. In T. L. Dean and K. McKeown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 634–639, Anaheim, California, 1991. The MIT Press.

Niemelä, I. and P. Simons: Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

Schubert, L.: Monotonic solution of the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In H. E. Kyburg, R. P. Loui, and G. N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–68. Kluwer, Boston, 1990.