Incremental Deterministic Planning

Ştefan ANDREI
National University of Singapore,
School of Computing,
E-mail: andrei@comp.nus.edu.sg

Wei-Ngan CHIN
National University of Singapore,
School of Computing,
E-mail: chinwn@comp.nus.edu.sg

Martin RINARD
MIT, Department of Electrical
Engineering and Computer Science,
E-mail: rinard@lcs.mit.edu

Abstract

We present a new planning algorithm that formulates the planning problem as a counting satisfiability problem in which the number of available solutions guides the planner deterministically to its goal. In comparison with existing planners, our approach eliminates backtracking and supports efficient incremental planners that add additional subformulas without the need to recompute solutions for previously provided subformulas. Our experimental results show that our approach is competitive with existing state-of-the-art planners that formulate the planning problem as a satisfiability problem, then solve the satisfiability problem using specialized off-the-shelf satisfiability solvers such as zChaff.

1 Background

Planning is one of the most studied problems in artificial intelligence. A *planning problem* is solved by a *planner*, which takes as input a description of an initial state, a goal, and the possible actions that can be performed in a given world. The output of the planner is a sequence of actions which, when executed in any world satisfying the initial state, will achieve the desired goal.

A planning problem can be viewed as a *reachability analysis* problem. Given a set $\mathcal{A}$ of actions, a state s is *reachable* from some initial state s_0 if there is a sequence of actions in $\mathcal{A}$ that defines a path from s_0 to s. *Reachability analysis* consists of analyzing which states can be reached from s_0 in some number of steps and how to reach them. Reachability can be computed exactly through a reachability tree or a reachability graph, but these data structures lead to an algorithm with exponential complexity.

This exponential complexity motivated the introduction of the *planning graph* [1], a data structure that provides an efficient way to estimate the set of propositions possibly reachable from an initial state under some set of actions. Planning graphs have polynomial size and can be built in polynomial time. But because the planning graph provides only an approximation of the reachability tree, algorithms that use the planning graph may need to backtrack to find a solution [2], which again leads to exponential time complexity in the worst case.

Researchers have also developed algorithms that formulate the planning problem as a propositional satisfiability problem [3–5], then use an off-the-shelf satisfiability solver to obtain a solution. Recent improvements in the performance of general purpose algorithms for propositional satisfiability provide the ability to scale up to relatively large planning problems [6, 7].

1.1 Our Approach

This paper presents a new planning algorithm. Like standard SAT-based algorithms, our technique is based on reducing the plan graph to a SAT problem. But instead of then using a standard SAT solver, we instead use a counting SAT solver, which provides the number of distinct satisfying assignments for the given propositional formula.

Our technique uses two key insights to completely eliminate the need for backtracking. The first insight is based on a property of the SAT encoding of the plan graph — specifically, it turns out that, at each choice point in the solution extraction algorithm, knowing the number of possible satisfying assignments for each choice makes it possible to find a distinguished choice that *dominates* all other choices in the sense that if *any* of the choices contains a solution to the planning problem, then the distinguished choice must also contain a solution. Obviously, a planner that always chooses the distinguished choice has no need to backtrack — it will always deterministically find a plan if such a plan exists.

The second insight is that the use of a counting SAT solver rather than a SAT solver provides the information required to find the distinguished choice. The end result is that we are able to reduce the planning problem to a SAT problem, then use a counting SAT solver to obtain a deterministic planner that 1) never backtracks, and 2) is guaranteed to find a solution to the planning problem if one exists.

Moreover, the use of counting SAT in this context provides an additional benefit. Counting SAT enables the use of efficient incremental algorithms. In general, the planner produces a related sequence of related SAT problems as it finds the plan. Instead of starting anew for each problem in the sequence, our incremental counting SAT solver can instead exploit the structure that the current and previous problems share to efficiently and incrementally update the solution to a previous problem to obtain a new solution to the current problem. Our results show that this approach can dramatically increase the efficiency of the solver.

We have implemented a planner based on this approach and used it to obtain solutions to a variety of planning problems. Our results show that this approach can produce a planner that executes more efficiently than existing SAT-based approaches (which backtrack at choice points).

1.2 Planning graph

Briefly, a planning graph is a directed layered graph, where arcs are permitted only from one layer to the next. Nodes in level 0 of the graph correspond to the set P_0 of propositions (or facts, literals, fluents) denoting the initial state of a planning problem. To search for a solution, the planning graph is iteratively expanded for the next level. Level 1 contains two layers: an action level A_1 (set of actions whose preconditions are nodes in P_0) and a proposition level P_1 (union of P_0 and effects of A_1). A *plan* Π with k levels is a sequence of sets of actions $\Pi =< \pi_1, \pi_2, ..., \pi_k >$, with $\pi_i \subseteq A_i$, $\forall\ i = 1, 2, ..., k$. Actions from π_i can be executed in parallel, in the sense that their execution order is immaterial. Two actions a_1 and a_2 are *dependent* if (i) a_1 deletes a precondition of a_2 or (ii) a_1 deletes an effect of a_2 or (iii) a_2 deletes a precondition of an effect of a_1. Two actions a_1 and a_2 from level A_i are *mutex* if either (i) a_1 and a_2 are dependant or (ii) if a precondition of a_1 is mutex with a precondition of a_2. Two propositions p_1 and p_2 from P_i are *mutex* (i.e., mutually exclusive) if every action in P_i that has p_1 as an effect is mutex with every action that produces p_2, and there is no action in A_i that produces both p_1 and p_2. In fact, when constructing the planning graph, propositions and actions monotonically increase from one level to the next, while mutex pairs monotonically decrease. This implies that it is likely for the solution to have more parallel actions close to the last generated level rather than the first levels.

1.3 Existing planners

Graphplan was the the first planner to solve planning problems using planning graph analysis [1]. Graphplan broke previous records in terms of raw planning speed, and has become a popular planning framework. Basically, the pseudocode of Graphplan [2] is:

```
for ($k = 0, 1, 2, ...$) {
  expand the planning graph up to level $k$;
  check whether the planning graph satisfies a necessary
    condition for plan existence for goal $g$;
  if it does, then call SolExtract($g, k$);
}
where SolExtract($g, k$) is given by:
{ if $k = 0$ return the solution;
  for (each proposition $p$ in $g$)
    nondeterministically choose an action from level $k-1$
      to achieve $p$
  if (any pair of chosen actions is mutex) do backtrack;
  $g' = \{$preconditions of chosen actions$\}$;
  recursive call SolExtract($g', k-1$);
}
```

Note that if there is a failure at level k during solution extraction, the algorithm will backtrack over the other subsets of $\mathcal{A}_{k+1}$. If level 0 is successfully reached, then the corresponding sequence is a solution plan.

SATPLAN [6, 7] uses a similar approach, but there are some differences as well (SATPLAN takes as input a set of axiom schemas, while Graphplan takes as input a set of STRIPS-like operators).

1.4 Our contribution

To summarize, the contributions of this paper are:

• we provide an automatic way to avoid backtracking while searching for a plan. This is based on considering a harder problem than SAT, namely the counting SAT problem;

• our approach considers one more piece of information, called the *increment*, so that we can deterministically choose the proper action for the plan extraction. In fact, we shall exploit the monotony of mutex actions from one level to the next. In other words, when searching for a solution plan, by doing less actions at the first levels and more actions at the last levels, we can ensure that a solution of the planning graph is actually a solution of the planning problem, too;

• the technique is applied incrementally, namely we consider the satisfiability of the new added subformulas without repeating the satisfiability of the former subformulas;

• the experimental results show that our counting SAT solver is comparable to the state-of-the-art SAT solvers that have been adopted by the existing planners.

1.5 Our running example

To illustrate our technique, we choose as running example the "dinner-date" example from [8]. Its specification can be made using STRIPS-like domains:

Initial state: (and (`garbage`) (`cleanHands`) (`quiet`))
Goal: (and (`dinner`) (`present`) (not (`garbage`)))
Actions:
`cook`: precondition (`cleanHands`)
    : effect (`dinner`)
`wrap`: precondition (`quiet`)
    : effect (`present`)

`carry`: precondition

: effect (and (not `garbage`)) (not (`cleanHands`)))

`dolly`: precondition

: effect (and (not `garbage`)) (not (`quiet`)))

In this example, it is supposed that someone has initially `cleanHands`, while the house has `garbage` and is `quiet`. There are four possible actions: `cook` (requires `cleanHands` and achieves `dinner`), `wrap` (requires `quiet` and produces `present`), `carry` (has no precondition, but it has two effects: removes `garbage`, and negates `cleanHands`), and `dolly` (with no precondition, but she may remove the `garbage` and negates `quiet`). The planning graph up to level 1 is given in Figure 1 [8]. Action names are surrounded by boxes and horizontal double lines demote maintainance actions. Thin, curved lines between actions and propositions at a single level denote mutex relations.

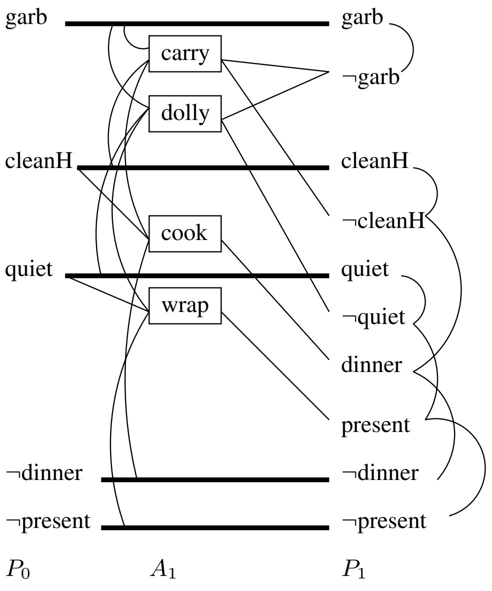

Figure 1. The dinner-date planning graph (levels 0, 1)

Obviously, we have two pairs of mutex actions: `carry` with `cook`, and `dolly` with `wrap`. Moreover, there exist two mutex pairs of propositions at level 1, namely ¬ `cleanH` (shortcut for ¬ `cleanHands`) with `dinner`, and `quiet` with `present`. It is easy to check that all propositions from the goal are possible at level 1, since they are not mutex with each other. Thus, there is a chance that a plan exists. Graphplan will consider two sets of actions: {`carry`, `cook`, `wrap`} and {`dolly`, `cook`, `wrap`}. Unfortunately, none of these sets of actions is a solution since `carry` is mutex with `cook` while `dolly` is mutex with `wrap`. Because the solution extraction fails, Graphplan will extend planning graph with one more level.

Next section shows a general technique to automatically decide whether it is necessary to extend the planning graph or to provide a solution plan. The idea is to construct a propositional formula F such that there exists a plan iff F is satisfiable.

2 A SAT encoding of the planning problem

The architecture of a typical planner has a *compiler* (which guesses a plan length, and generates a propositional formula), a *simplifier* (which shrinks the propositional formula by applying techniques such as unit clause propagation and pure literal elimination), a *solver* (which finds a truth assignment of the propositional formula), and a *decoder* (which gives a solution plan).

Let $\mathbb{LP}$ be the *propositional logic* over the finite set of *atomic formulae* (propositional variables) $V = \{X_1, ..., X_n\}$. A *literal* L is an atomic formula A (*positive* literal) or its negation $\neg A$ (*negative* literal), and denote $\mathcal{V}(L) = \mathcal{V}(\overline{L}) = A$. Any function $\mathcal{S} : V \to \{0,1\}$ is an *assignment* (or, *model*) and it can be uniquely extended in $\mathbb{LP}$ to F. Any propositional formulae $F \in \mathbb{LP}$ can be translated into the *conjunctive normal form* (CNF): $F = (L_{1,1} \vee ... \vee L_{1,n_1}) \wedge ... \wedge (L_{l,1} \vee ... \vee L_{l,n_l})$, where $L_{i,j}$ are literals and $l \geq 1$. We shall use the set representation $\{\{L_{1,1}, ..., L_{1,n_1}\}, ..., \{L_{l,1}, ..., L_{l,n_l}\}\}$ to denote F. Any finite disjunction of literals is a *clause* and a formula in CNF is called a *clausal formula*. A formula F is called *satisfiable* iff there exists a structure $\mathcal{S}$ for which $\mathcal{S}(F) = 1$. A formula F is called *unsatisfiable* (or *contradiction*) iff F is not satisfiable. We denote by $\Box$ the *empty clause* (i.e., the one without any literal). A *unit clause* has only one literal.

There are many ways of generating the CNF formula for a planning problem [2]. One way is to consider the following five types of formulas: 1) the initial state is encoded as the conjunctive of fluents that hold and the negation of those that do not hold; 2) the set of goal states is encoded as the conjunction of fluents that must hold in the last state; 3) the $A \Longrightarrow P, E$ *axioms*: given an applicable action, its preconditions must hold at that step, and its effects will hold at the next step; 4) the *explanatory frame axioms*: if a fluent changes, then one of the actions that have that fluent in its effects has been executed; 5) the *complete exclusion axioms*: only one action occurs at each step. These five sets of formulas encode the planning problem P having a goal with n actions to propositional satisfiability [5]; that is, the CNF formula is satisfiable iff there exists a solution plan of length n to P. Since only one action occurs at each level, this is also called *linear SAT-encoding*.

There exist variations of the above encoding, with the possibility of doing more than one action at each level, for example by considering *classical frame axioms* which state what fluents are left unchanged by a given action [9]. In this case, exclusion axioms are unnecessary since the classical frame axioms combined with $A \Longrightarrow P, E$ axioms ensure that any two actions occurring at time t lead to identical world-state at time $t + 1$.

Another variation refers to exclusion. Instead of complete exclusion axioms, only the conflict exclusion axioms [10] may be added (two actions conflict if one's precondition is inconsistent with the other's effect).

Another possible SAT encoding was shown in [4], i.e., the planning graph can be automatically converted into a

CNF formula by considering the initial state and goal encoding, mutex of conflicting actions, actions imply their preconditions, and each fact implies the disjunction of all operators (actions, facts) from the previous levels. Among all the existing SAT-encoding, it seems this latter one is the most efficient because of the CNF formula size. As mentioned in [4], this encoding is more efficient than Graphplan-based encodings when considering larger benchmark instances. However, this SAT-encoding is not complete, in the sense that not every solution of the SAT formula corresponds to a solution to a solution of the planning problem. Because of the lake of axioms of kind 'actions imply their effects', the SAT solutions may contain spurious actions. In fact, this matter was solved in [5], where the extraction function can simply delete actions from the solution those effects do not hold.

For efficiency reasons, we consider the SAT encoding from [4] and illustrate how it works for the dinner-date example [8]. Even if this SAT encoding is an incomplete one, there exist different ways to find the proper solution (details in Section 4). The fact that the planning graph having one level cannot lead to a solution can be automatically done by checking the satisfiability of a CNF formula which encodes the given problem. Let us denote by $V_1 = \{X_1,$ X_2, ..., $X_{14}\}$ the set of propositional variables which corresponds to `garbage`, `cleanHands`, `quiet`, `dinner`, `present` (from level 0), `carry`, `dolly`, `cook`, `wrap`, `garbage`, `cleanHands`, `quiet`, `dinner`, `present` (from level 1), respectively. The initial state corresponds to the formula: $\{ \{X_1\}, \{X_2\}, \{X_3\}, \{\overline{X_4}\}, \{\overline{X_5}\} \}$. The goal for the first level is: $\{ \{\overline{X_{10}}\}, \{X_{13}\}, \{X_{14}\} \}$. For example, by considering the 'actions imply their preconditions' axioms, we get the clauses $\{X_2, \overline{X_8}\}$, $\{X_3, \overline{X_9}\}$, by considering the mutex between actions axioms we get $\{\overline{X_6}, \overline{X_8}\}$, $\{\overline{X_7}, \overline{X_9}\}$, and by considering 'each fact implies the disjunction of all operators' axioms we get $\{X_6, X_7, X_{10}\}$, $\{X_8, \overline{X_{13}}\}$, and $\{X_9, \overline{X_{14}}\}$.

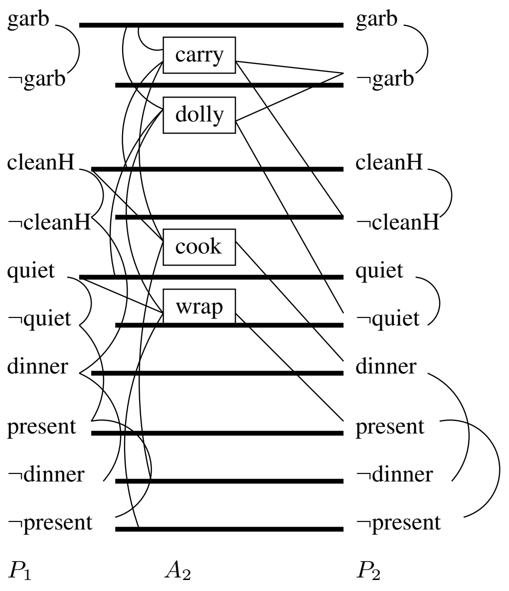

Figure 2. The dinner-date planning graph (level 2)

We denote by F_1 the CNF formula containing all the previous clauses. Since F_1 is unsatisfiable, there is no solution for the planning graph having only one level. Figure 2 shows the planning graph extended with one more level of actions and facts. In fact, we can do a binary search on instantiations of various sizes in order to find the smallest level for which a solution is found [4] (e.g. if the optimal plan length is 5 , the search can proceed for plans of length 2, 4 (no plan found), 8 (plan found), 6 (plan found) and finally 5.

Let us denote by X_{15}, X_{16}, ..., X_{23} the propositional variables which corresponds to `carry`, `dolly`, `cook`, `wrap`, `garbage`, `cleanHands`, `quiet`, `dinner`, `present`, all from level 2. The new set of variables is now $V_2 = V_1 \cup \{ X_{15}, X_{16}, ..., X_{23} \}$. Let us denote by F_2 the CNF formula which encode the planning graph up to level 2. Obviously, F_2 contains F_1, except the goal clauses, and the following new clauses $\{ \{\overline{X_{19}}\}, \{X_{22}\},$ $\{X_{23}\} \}$ (the goal for the second level), $\{X_{11}, \overline{X_{17}}\}$, $\{X_{12}, \overline{X_{18}}\}$ (the 'actions imply their preconditions' axioms), $\{\overline{X_{15}}, \overline{X_{17}}\}$, $\{\overline{X_{16}}, \overline{X_{18}}\}$ (the mutex between actions axioms), $\{\overline{X_{10}}, X_{15}, X_{16}, X_{19}\}$, $\{X_{13}, X_{17}, \overline{X_{22}}\}$, and $\{X_{14}, X_{18}, \overline{X_{23}}\}$ ('each fact implies the disjunction of all operators' axioms)

Since the old goal has been removed, we insert to F_2 the unit clauses which corresponds to the new goal: $\{\overline{X_{19}}\}$, $\{X_{22}\}$, $\{X_{23}\}$. This time F_2 is satisfiable, so we have to look for solutions by considering all the 12 possible disjunctions for the goal of level 2, namely $\{\texttt{carry}_2, \texttt{dolly}_2,$ $\neg\texttt{garb}_1 \} \times \{\texttt{cook}_2, \texttt{dinner}_1 \} \times \{\texttt{wrap}_2, \texttt{present}_1$ $\}$. Eventually, by running the procedure `solExtract()`, a solution will be found. Note that the solution extraction procedure may need to do backtracking in order to get the desired solution. The next section shows a different alternative by proposing a deterministic approach for solving the planning problem using counting satisfiability.

3 Incremental counting SAT

A counting problem P determines how many solutions exist, not just an answer "Yes/No" like a decision problem. The problem of counting the number of truth assignments (denoted by #SAT) was proved to be *#$\mathcal{P}$-complete* [11]. The *#$\mathcal{P}$-complete* problems are at least as hard as *$\mathcal{NP}$-complete* problems. In fact, the class #$\mathcal{P}$ includes $\mathcal{NP}$, and, in turn, is included in $\mathcal{PSPACE}$.

For a finite set A, $|A|$ denotes the number of elements of A. $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{N}_+$ denote the set of integers, positive integers, and the set of strict positive integers, respectively.

Notation 3.1 *Let $C_1, ..., C_s$ be clauses over V ($s \geq 1$). We denote:*
a) $m_V(C_1, ..., C_s) = |\{A \mid A \in V - \mathcal{V}(C_1 \cup ... \cup C_s)\}|$;
b) $dif_V(C_1, ..., C_s) = 0$ if ($\exists\ i, j \in \{1, ..., s\}$, $i \neq j$, such as $\exists\ L \in C_i$ and $\overline{L} \in C_j$) **or** *($\exists\ i \in \{1, ..., s\}$, such as $C_i = \square$); otherwise, $dif_V(C_1, ..., C_s) = 2^{m_V(C_1, ..., C_s)}$;*

c) the **determinant** *of the set of clauses* $\{C_1, ..., C_s\}$ *is* $det_V(C_1, ..., C_s) = 2^{|V|} - \sum_{j=1}^{s}(-1)^{j+1} \cdot \sum_{1 \leq i_1 < ... < i_j \leq s} dif_V(C_{i_1}, ..., C_{i_j})$; ■

In other words, the above notation says that $m_V(C_1, ..., C_s)$ denotes the number of atomic formulae from V which do not occur in $C_1 \cup ... \cup C_s$. The positive integer number $dif_V(C_1, ..., C_s)$ is 0 iff there is a literal in one of the argument's clause and its opposite in another clause **or** one of the clauses is the empty one. The integer $det_V(C_1, ..., C_s)$ is a sign-alternated sum of $dif_V()$. If $F = \{C_1, ..., C_s\}$, we may denote $m_V(C_1, ..., C_s)$ as $m_V(F)$, $dif_V(C_1, ..., C_s)$ as $dif_V(F)$ and $det_V(C_1, ..., C_s)$ as $det_V(F)$. In fact, the determinant of a clausal formula coincides with the number of truth assignments of that formula. Given $F \in \mathbb{LP}$ over V, there exist $det_V(F)$ truth assignments for F. Obviously, knowing $det_V(F)$, the SAT problem can be solved, too. That is, F is satisfiable iff $det_V(F) \neq 0$.

For example, let us consider $F = \{C_1, C_2, C_3\}$, where $V = \{p, q, r\}$ and $C_1 = \{p, \overline{q}\}$, $C_2 = \{q, \overline{r}\}$, $C_3 = \{p, r\}$. Then $m_V(C_1) = 1$, $m_V(C_1, C_2) = 0$, and so on. Thus, $dif_V(C_1) = 2$, $dif_V(C_1, C_2) = 0$, and so on. Therefore, $det_V(F) = 2^3 - (2^1 + 2^1 + 2^1) + 2^0 = 3$. It follows that F is satisfiable and has $det_V(F) = 3$ (distinct) truth assignments.

Incremental computation: Since $det_V(F)$ may contain an exponential number of $dif_V()$'s depending on the number of clauses of F, whenever a new clause C is added, it is better to compute *only* the $dif_V()$'s which contain C and not the whole $dif_V()$'s corresponding to $F \cup \{C\}$. Next, the increment of a given clausal formula F with an arbitrary clause C is defined.

Notation 3.2 *If* $F = \{C_1, ..., C_l\}$ *is an arbitrary clausal formula over* V *and* C *is an arbitrary clause over* V*, then* $inc_V(C, F) = \sum_{s=0}^{l}(-1)^{s+1} \cdot \sum_{1 \leq i_1 < ... < i_s \leq l} dif_V(C, C_{i_1}, ..., C_{i_s})$ *is called the* **increment** *of* F *with clause* C.

The increment is a negative integer representing the number of truth assignments which have to be added or subtracted from the previous value of the determinant.

In the following, the main result of this section is presented. It allows the computation of the determinant of a new clausal formula using the already computed determinant of the old clausal formula. Moreover, the incremental computation of the determinant of a formula containing new clauses is *optimal*. That is, no new $dif_V()$'s expressions are computed in the new incremental expression, that is $inc_V(C, F)$, except the ones which would have been created in the non-incremental approach.

Theorem 3.1 *Let* $F = \{C_1, ..., C_l\}$ *be a clausal formula over* V *and let* $F' = \{C_{l+1}, ..., C_{l+k}\}$, $k \geq 1$, *be a clausal formula over* V. *Then* $det_V(F \cup F') = det_V(F) + inc_V(C_{l+1}, F) + inc_V(C_{l+2}, F \cup \{C_{l+1}\}) + ... + inc_V(C_{l+k}, F \cup \{C_{l+1}\} \cup ... \cup \{C_{l+k-1}\})$.

Considering the notations from Theorem 3.1, we denote $inc_V(F', F) = inc_V(C_{l+1}, F) + inc_V(C_{l+2}, F \cup \{C_{l+1}\}) + ... + inc_V(C_{l+k}, F \cup \{C_{l+1}\} \cup ... \cup \{C_{l+k-1}\})$. The next corollar points out some situations when the computation of the determinant and the increment can be speed up.

Corollary 3.1 *Let* $F = \{C_1, ..., C_l\}$ *be a clausal formula over* V. *Then:*
a) if A *is a new atomic variable,* $A \notin V$, *then* $det_{V \cup \{A\}}(\{A\}, F) = det_{V \cup \{A\}}(\{\overline{A}\}, F) = det_V(F)$ *and* $inc_{V \cup \{A\}}(\{A\}, F) = inc_{V \cup \{A\}}(\{\overline{A}\}, F) = -det_V(F)$;
b) if $V' = \{X_1, ..., X_m\}$ *is a set of atomic variables,* $m \in \mathbb{N}_+$, $X_1, ..., X_m \notin V$, *then* $det_{V \cup V'}(F) = 2^m \cdot det_V(F)$ *and* $inc_{V \cup V'}(C, F) = 2^m \cdot inc_V(C, F)$.

4 Incremental deterministic planning

In the following, we shall describe our approach for the dinner-date example. By simply computing the determinant as shown in the previous section, we get $det_{V_1}(F_1) = 0$, and $det_{V_2}(F_2) = 172$. We immediately deduce that F_1 is unsatisfiable and F_2 is satisfiable. It implies there are no solutions at level 1 and there might be some solutions at level 2. To check that, one way is to verify which of the 12 candidate solutions are solutions of the planning problem, i.e., the elements of the cartesian product {`carry`$_2$, `dolly`$_2$, $\neg$`garb`$_1$ } × {`cook`$_2$, `dinner`$_1$ } × {`wrap`$_2$, `present`$_1$}. Since each of these cases refer to addition of exactly three unit clauses to F_2, the computation time of the corresponding determinants/increments can be done efficiently. These are:

- $F_2^{(1)} = \{\{X_{15}\}, \{X_{17}\}, \{X_{18}\}\} \cup F_2$, $inc_{V_2}(F_2^{(1)}) = -172$, so $det_{V_2}(F_2^{(1)}) = 0$;
- $F_2^{(2)} = \{\{X_{15}\}, \{X_{17}\}, \{X_{14}\}\} \cup F_2$, $inc_{V_2}(F_2^{(2)}) = -172$, so $det_{V_2}(F_2^{(2)}) = 0$;
- $F_2^{(3)} = \{\{X_{15}\}, \{X_{13}\}, \{X_{18}\}\} \cup F_2$, $inc_{V_2}(F_2^{(3)}) = -132$, so $det_{V_2}(F_2^{(3)}) = 40$;
- $F_2^{(4)} = \{\{X_{15}\}, \{X_{13}\}, \{X_{14}\}\} \cup F_2$, $inc_{V_2}(F_2^{(4)}) = -132$, so $det_{V_2}(F_2^{(4)}) = 40$;
- $F_2^{(5)} = \{\{X_{16}\}, \{X_{17}\}, \{X_{18}\}\} \cup F_2$, $inc_{V_2}(F_2^{(5)}) = -172$, so $det_{V_2}(F_2^{(5)}) = 0$;
- $F_2^{(6)} = \{\{X_{16}\}, \{X_{17}\}, \{X_{14}\}\} \cup F_2$, $inc_{V_2}(F_2^{(6)}) = -132$, so $det_{V_2}(F_2^{(6)}) = 40$;
- $F_2^{(7)} = \{\{X_{16}\}, \{X_{13}\}, \{X_{18}\}\} \cup F_2$, $inc_{V_2}(F_2^{(7)}) = -172$, so $det_{V_2}(F_2^{(7)}) = 0$;
- $F_2^{(8)} = \{\{X_{16}\}, \{X_{13}\}, \{X_{14}\}\} \cup F_2$, $inc_{V_2}(F_2^{(8)}) = -132$, so $det_{V_2}(F_2^{(8)}) = 40$;
- $F_2^{(9)} = \{\{\overline{X_{10}}\}, \{X_{17}\}, \{X_{18}\}\} \cup F_2$, $inc_{V_2}(F_2^{(9)}) = -144$, so $det_{V_2}(F_2^{(9)}) = 28$;
- $F_2^{(10)} = \{\{\overline{X_{10}}\}, \{X_{17}\}, \{X_{14}\}\} \cup F_2$, $inc_{V_2}(F_2^{(10)}) = -152$, so $det_{V_2}(F_2^{(10)}) = 20$;
- $F_2^{(11)} = \{\{\overline{X_{10}}\}, \{X_{13}\}, \{X_{18}\}\} \cup F_2$, $inc_{V_2}(F_2^{(11)}) = -152$, so $det_{V_2}(F_2^{(11)}) = 20$;

• $F_2^{(12)} = \{\{\overline{X_{10}}\}, \{X_{13}\}, \{X_{14}\}\} \cup F_2$, $inc_{V_2}(F_2^{(12)}) = -172$, so $det_{V_2}(F_2^{(12)}) = 0$;

This SAT encoding ensures that $F_2^{(1)}$, $F_2^{(2)}$, $F_2^{(5)}$, $F_2^{(7)}$ and $F_2^{(12)}$ do not correspond to solutions of the planning problem (because they are unsatisfiable formulas). However, the formulas $F_2^{(9)}$, $F_2^{(10)}$ and $F_2^{(11)}$ correspond to satisfiable SAT formulas, but they do not lead to solutions of the planning problem. Since a classical SAT solver can only answer with 'Yes/No' regarding the satisfiability of a given CNF formula, it is impossible to detect using a SAT solver that $F_2^{(9)}$, $F_2^{(10)}$ and $F_2^{(11)}$ cannot lead to solutions of the planning problem.

To the best of our knowledge, there are two traditional ways to eliminate the incorrect (spurious) solutions:

• consider a richer SAT encoding, by including the $A \Longrightarrow E$ axioms. However, this approach may lead to large SAT formulas;

• consider the above SAT encoding, but when the planning graph is created, special dummy "maintain" actions are also created [12]. Then, if a fact holds at time i, then it must either be added by an action at time $i-1$, or maintained by the corresponded maintain action at time $i-1$. This approach was successfully used in Blackbox [7].

Our alternative is different from these two traditional approaches. Instead, a counting SAT solver can use the number of truth assignments to decide which solution to select. For example, we shall use the information that the determinants of $F_2^{(9)}$, $F_2^{(10)}$ and $F_2^{(11)}$ are among the smallest out of 12 determinants. In this way, we are showing an efficient way to deterministically identify a solution plan using the determinants. So, coming back to our example, instead of doing 'generate and test' of all the 12 possible candidates, we consider first:

$inc_{V_2}(\{X_{15}\}, F_2) = -100$, so $det_{V_2}(\{\{X_{15}\}\} \cup F_2) = 72$;
$inc_{V_2}(\{X_{16}\}, F_2) = -100$, so $det_{V_2}(\{\{X_{16}\}\} \cup F_2) = 72$;
$inc_{V_2}(\{\overline{X_{10}}\}, F_2) = -112$, so $det_{V_2}(\{\{\overline{X_{10}}\}\} \cup F_2) = 60$;

In order to get the solution directly, we shall choose the cases having the maximum increment/determinant, e.g. either $\{X_{15}\}$ or $\{X_{16}\}$ which corresponds for $\texttt{carry}_2$ or $\texttt{dolly}_2$, respectively. Assuming that we choose $\texttt{carry}_2$, we compute the increments/determinants corresponding to $\texttt{cook}_2$ and $\texttt{dinner}_1$:

$inc_{V_2}(\{X_{17}\}, \{\{X_{15}\}\} \cup F_2) = -72$, so determinant is 0;
$inc_{V_2}(\{X_{13}\}, \{\{X_{15}\}\} \cup F_2) = 0$, so its determinant is 72;

According to the same strategy, we choose $\{A_{13}\}$ as the solution and continue with the third set of the cartesian product, namely $\{\texttt{wrap}_2, \texttt{present}_1\}$. Now, we continue with $inc_{V_2}(\{X_{18}\}, \{\{X_{13}\}, \{X_{15}\}\} \cup F_2) = -32$, so its determinant is 40; or $inc_{V_2}(\{X_{14}\}, \{\{X_{13}\}, \{X_{15}\}\} \cup F_2) = -32$, so its determinant is 40.

Since both determinants have the same value, it is likely that any of them may actually correspond to a solution of the planning problem. Because $\texttt{dinner}_1$ and $\texttt{present}_1$ are effects of $\texttt{cook}_1$ and $\texttt{wrap}_1$, we get either

$< \{\texttt{cook}_1\}, \{\texttt{carry}_2, \texttt{wrap}_2\} >$ or
$< \{\texttt{cook}_1, \texttt{wrap}_1\}, \{\texttt{carry}_2\} >$

as the first two solutions for the planning problem. Of course, if we have chosen initially $\texttt{dolly}_2$ as an action, by doing a similar strategy, we get the solutions:

$< \{\texttt{wrap}_1\}, \{\texttt{dolly}_2, \texttt{cook}_2\} >$
and
$< \{\texttt{cook}_1, \texttt{wrap}_1\}, \{\texttt{dolly}_2\} >$.

Next, we shall prove two results which allow us to choose deterministically a solution among the set of all solutions of the planning problem.

Lemma 4.1 *Let F_i be the CNF formula over V_i corresponding to level i of a planning graph, and let $f^{(i)}$ be a variable of the goal corresponding to the goal of level i, and $f^{(i)} \to \{a_1^{(i)}, ..., a_n^{(i)}, f^{(i-1)}\}$ be the implication of a fact of level i to actions of level i and a fact of level $i-1$. Then $det_{V_i}(F_i \cup \{a_k^{(i)}\}) > det_{V_i}(F_i \cup \{f^{(i-1)}\})$, $\forall\, k \in \overline{1,n}$.*

In fact, if we consider that actions $a_1^{(i)}$, ..., $a_n^{(i)}$ are all the actions related to facts $f^{(i)}$ and $f^{(i-1)}$, then the converse of Lemma 4.1 holds, too. In other words, Lemmas 4.1 and 4.2 ensure that when searching for a solution, it is necessary to consider the determinant having the maximum value among the actions of level i and facts from level $i-1$.

Lemma 4.2 *Let F_i be the CNF formula over V_i corresponding to level i of a planning graph, and let $\{a_1^{(i)}$, ..., $a_n^{(i)}\}$ be the set of actions of level i, and $f^{(i-1)}$ be a fact of level $i-1$ such that $f^{(i)} \to a_1^{(i)} \vee ... \vee a_n^{(i)} \vee f^{(i-1)}$ is a 'fact implies operators' axiom.*

If $\{a_j^{(i)}\} \cup F$ does not lead to a solution of the planning problem, for all $j \in \{1, ..., n\}$, then $f^{(i-1)} \cup F$ does not lead to a solution, too.

Here is our deterministic procedure $\texttt{detSolExtract}(g, k, F_k, S_k)$ for solution extraction, where g is the number of positive facts in the goal, k is the number of levels, F_k is the SAT-encoding corresponding to level k, and S_k is the space of solution plan, i.e., $S_k = \{op_1^{(1)}, ..., op_{k_1}^{(1)}\} \times ... \times \{op_1^{(g)}, ..., op_{k_g}^{(g)}\}$.

```
detSolExtract($g, k, F_k, S_k$)
  for ( $i = 0; i \le g; i++$ ) {
      let $k_j \in \{1, ..., k_i\}$ such that $inc_{V_k}(\{X_j^{(i)}\}, F'_k)$ has
      the maximum value, where $X_j^{(i)}$ is the propositional
      variable corresponding to $op_{k_j}^{(i)}$;
      $F'_k = F'_k \cup \{\{X_j^{(i)}\}\}$;
      choose $op_{k_j}^{(i)}$ as part of the solution plan;
  }
```

Note that procedure $\texttt{detSolExtract}()$ never backtracks. It will provide the plan $\Pi =< \pi_1, ..., \pi_k >$, according to the "as late as possible" strategy. The increment can decide the actions executed at a given step. In other words, when searching for a solution plan, our strategy will find a solution which corresponds to less actions at the first levels and more actions at the last levels.

5 Experimental results

Among the existing SAT-based planners, it seems that the SatPlan_2004, a planner developed by Henry Kautz and his team is the most efficient one. The SatPlan_2004 took first place for optimal deterministic planning [7]. This planner is based on one of the fastest SAT solvers, namely zChaff [13]. This section is devoted to two comparisons:

• the incremental approach versus the non-incremental approach and

• our counting SAT solver against some state-of-the-art SAT solvers (including zChaff). Our comparison uses randomly generated clausal formulae and some common planning examples.

The first experiment compares the incremental approach versus non-incremental approach. We consider the addition of two new clauses, i.e., C_{l+1} and C_{l+2}, to the clausal formula $F = \{C_1, ..., C_l\}$ over $V = \{X_1, ..., X_n\}$. Our testing instances refer to different values for (n, l), where n and l denote the number of variables and clauses, respectively. Let us denote by New, Old, Inc_1, and Inc_2 the time needed (in seconds) for computing the exact values of $det_V(F\cup \{C_{l+1}\}\cup \{C_{l+2}\})$, $det_V(F)$, $inc_V(C_{l+1}, F)$, and $inc_V(C_{l+2}, F\cup \{C_{l+1}\})$. Table 1 presents the results.

(n,l)	New	Old	Inc_1	Inc_2
$(10, 20)$	0.16	0.06	0.01	0.05
$(15, 25)$	0.37	0.13	0.11	0.21
$(20, 40)$	3.32	2.48	0.39	0.41
$(25, 45)$	2.18	1.50	0.16	0.71
$(30, 60)$	7.70	6.03	0.83	1.28
$(40, 75)$	11.64	8.77	1.42	2.19
$(50, 100)$	39.26	33.57	0.67	5.66
$(100, 200)$	2147	1992	144	30.48

Table 1. Incremental versus non-incremental

Definitely, once we know $det_V(F)$, it is more convenient to compute only $inc_V(C_{l+1}, F)$ and $inc_V(C_{l+2}, F\cup \{C_{l+1}\})$, rather than evaluating $det_V(F\cup \{C_{l+1}\}\cup \{C_{l+2}\})$. Actually, the time needed for computing $det_V(F\cup \{C_{l+1}\}\cup \{C_{l+2}\})$ is approximately equal to the time consumed by the computation of $det_V(F)$, $inc_V(C_{l+1}, F)$, and $inc_V(C_{l+2}, F\cup \{C_{l+1}\})$ together. For the first line of Table 1, we may see that the incremental computation is more efficient that non-incremental computation because $0.01+ 0.05$ is less than 0.16.

The second experiment is devoted to an approximate evaluation of the determinant of a clausal formula. Our counting SAT solver (called **CoSAT**) has two variations, namely a complete one (which returns the exact number of truth assignments) and an approximate one (which returns a lower bound for the number of truth assignments). The complete variation cannot work in a reasonable amount of time for large CNF formulas, while the approximate variation works for large CNF formulas. Table 2 shows the execution times of the approximated variation of CoSAT.

Existing tools (such as state of the art SAT solvers) for solving $\mathcal{NP}-$complete problems may offer fast answers and/or incomplete answers for particular subclasses of inputs. As mentioned in [13], GRASP [14] has been developed as a combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. zChaff [13] is based almost exclusively on the DP search algorithm. We run GRASP, zChaff and CoSAT on a Red Hat Linux 9.0 Pentium 4, 2.0 GHz processor, using 1 GB of memory. Table 2 presents the results of this experiment.

	GRASP	zChaff	CoSAT
(n,l)	Time	Time	Time
$(900, 2000)$	2.21	0.37	0.52
$(1000, 2200)$	2.82	0.45	0.60
$(1100, 2400)$	3.6	0.58	0.69
$(1200, 2600)$	4.42	0.71	0.70
$(1300, 2900)$	5.78	0.77	0.82
$(1500, 3500)$	8.74	1.63	1.34
$(1750, 4000)$	13.51	1.57	1.68
$(2000, 5000)$	20.34	2.31	2.99
$(2500, 6000)$	34.3	3.39	3.30
$(3000, 7000)$	53.32	5.07	4.43

Table 2. CoSAT against GRASP and zChaff

Table 2 demonstrates that our tool is comparable with existing state-of-the-art SAT solvers for our randomly generated planning problems. For example, CoSAT outperforms GRASP with a order of magnitude between 4 and 12. Moreover, CoSAT is comparable with zChaff, and tends to outperform it as the size of the input CNF formula increases. Note that CoSAT solves a harder problem than both SAT solvers. We also performed a preliminary study involving a set of planning problems. We compared the number of invocations of the SAT procedure by the classical planner. Assuming that the backtracking will explore the entire search tree (procedure `SolExtract`(g, k)), our approach (procedure `detSolExtract`(g, k)) can lead to faster SAT-based planners.

6 Related Work and Conclusions

The counting SAT problem has many applications in the areas of computer science, including artificial intelligence, real-time systems, and so on. As mentioned in [15], counting is often the most natural way of verifying equivalence between two theories. Moreover, it can provide degrees of how close is a theory by its approximation [16, 17]. Furthermore, counting can provide heuristics for guiding planning and search, where an estimation of the probability for a given search would help lead to a goal. The number of solutions found in a simplified version of the problem description can then serve as an estimation of this probability [18]. There exist complete (i.e. gives the total number of truth assignments) counting SAT solvers [19–21], but our tests showed that their execution times are still impractical for large benchmark planning problems (e.g. rocket, logistic, blockworld). Our use of approximated counting SAT solver is a step towards resolving this difficulty.

While there has been substantial progress for solving the

SAT problem, there are related problems where #SAT is more useful. A recent example is an application for improvement of propositional reasoning. In particular, while formula caching may have theoretical value in SAT solvers [22], component caching seems to be more promising when a #SAT solver is efficiently applied [23, 24].

This paper presents an efficient incremental #SAT-based planner as an alternative for existing SAT-based planners. The experimental results demonstrate that our approach is promising. The paper has laid down a set of underlying theories and techniques that form the basis for deterministic incremental planning.

References

[1] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995, pp. 1636–1642.

[2] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, Elsevier, 2004.

[3] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the Tenth European Conference on Artificial Intelligence*, 1992, pp. 359–363.

[4] ——, "Pushing the envelope: Planning, propositional logic, and stochastic search," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, 1996, pp. 1194–1201.

[5] H. Kautz, D. McAllester, and B. Selman, "Encoding plans in propositional logic," in *Proceedings of the Fifth International Conference on the Principle of Knowledge Representation and Reasoning*, 1996, pp. 374–384.

[6] H. Kautz and B. Selman, "Unifying SAT-based and graph-based planning," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999, pp. 318–325.

[7] H. Kautz, "SatPlan: Planning as satisfiability," in *International Planning Competition at the 14th International Conference on Automated Planning and Scheduling*, 2004. [Online]. Available: http://www.cs.washington.edu/homes/kautz/satplan/

[8] D. S. Weld, "Recent advances in AI planning," *AI Magazine*, vol. 20, pp. 93–123, 1999.

[9] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," pp. 26–45, 1987.

[10] M. D. Ernst, T. D. Millstein, and D. S. Weld, "Automatic SAT-compilation of planning problems," in *IJCAI-97, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997, pp. 1169–1176.

[11] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal on Computing*, vol. 8, pp. 410–421, 1979.

[12] H. Kautz, "Private communication," July 2005.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *DAC '01: Proceedings of the 38th conference on Design automation*. ACM Press, 2001, pp. 530–535.

[14] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506–521, 2000.

[15] R. Dechter and A. Itai, "Finding all solutions if you can find one," in *Workshop on Tractable Reasoning, AAAI'92*, 1992, pp. 35–39.

[16] R. Dechter and J. Pearl, "Structure identification in relational data," in *The Canadian Artificial Intelligence Conference*, 1992.

[17] B. Sellman and H. Kautz, "Knowledge compilation using Horn approximation," in *Proceedings of AAAI-91*, 1991.

[18] R. Dechter, "Network-based heuristics for constraint satisfaction problems," *Artificial Intelligence*, vol. 34, pp. 1–38, 1987.

[19] R. J. B. Jr. and J. D. Pehoushek, "Counting models using connected components." in *AAAI/IAAI*, 2000, pp. 157–162.

[20] S. Andrei, "Counting for satisfiability by inverting resolution," *Artificial Intelligence Review*, vol. 22, no. 4, pp. 339–366, 2004.

[21] T. Sang, P. Beame, and H. A. Kautz, "Heuristics for fast exact model counting." in *SAT*, 2005, pp. 226–240.

[22] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind, "Memoization and DPLL: Formula caching proof systems," in *Proceedings Eighteenth Annual IEEE Conference on Computational Complexity*. IEEE, 2003, pp. 225–236.

[23] F. Bacchus, S. Dalmao, and T. Pitassi, "DPLL with caching: A new algorithm for #SAT and Bayesian inference," in *Proceedings 44th Annual Symposium on Foundations of Computer Science*. IEEE, 2003.

[24] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi, "Combining component caching and clause learning for effective model counting," in *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.