Plan-A: A Cost Optimal Planner Based on SAT-Constrained Optimization [*]

Yixin Chen
Department of Computer Science
Washington University in St. Louis
St. Louis, MO 63130, USA
chen@cse.wustl.edu

Qiang Lv
Department of Computer Science
University of Science and Technology of China
Hefei, Anhui, China
rczx@mail.ustc.edu.cn

Ruoyun Huang
Department of Computer Science
Washington University in St. Louis
St. Louis, MO 63130, USA
rh11@cse.wustl.edu

Abstract

Planning as satisfiability, represented by SATPlan, can find plans with the shortest makespan for classical planning. However, it has been deemed a limitation of SAT-based approaches for delivering optimality regarding metrics other than the makespan.

Our Plan-A solver uses the SAT-based framework to find plans with minimal total action costs. The idea is to optimally solve SAT instances based on certain optimality metrics. Bounding and pruning techniques are proposed and integrated with the standard DPLL algorithm to reduce the search cost.

Introduction

Planning as satisfiability, represented by Blackbox (Kautz & Selman 1996), SATPlan (Kautz 2004; 2006), and Max-Plan (Chen, Zhao, & Zhang 2007), is a well-known approach for optimal classical planning.

Given a planning problem, SATPlan encodes the problem into a SAT formulation with a fixed "makespan" k and checks the satisfiability using a SAT solver. If the SAT model is satisfiable, SATPlan returns a solution; otherwise it increases k by 1 and repeats. Most SAT based planners follow this framework and generate plans that are optimal in terms of the makespan.

However, in many planning applications, optimal plans that minimize the action costs are needed. In such problems, each action has a nonnegative cost and the goal is to find a plan with minimum total cost of actions. It is deemed a major limitation of the SATPlan framework that it is difficult to optimize other plan metrics, such as the total action costs.

SATPlan$^{\prec}$ (Giunchiglia & Maratea 2007), a recent variation of SATPlan, makes a significant step towards using SATPlan for cost-optimal planning. It can generate plans with minimum number of actions for a given makespan. This is achieved by solving the SAT instances using OPT-SAT (Giunchiglia & Maratea 2006), a tool for solving SAT-constrained optimization problems. Unlike the traditional DPLL style algorithms, OPTSAT does not follow a branch and bound scheme but solves the optimization problem by imposing a partial ordering on the literals to branch on. Given a SAT model Π and a subset S of the variables in Π, OPTSAT can return an optimal solution that minimizes the number of variables in S that are assigned to $true$.

We propose another alternative approach to optimize the action costs. This algorithm, called Plan-A, is based on DPLL-OPT, a novel algorithm we propose for finding solutions to SAT instances that minimize an objective function. Instead of seeking for any feasible solution, DPLL-OPT completely searches the space of a SAT instance to minimize the given objective function. DPLL-OPT follows the DPLL framework. However, when a satisfiable solution is found, DPLL-OPT does not quit but instead adds a blocking clause to the clause database to prevent the search to return the same solution again. It then performs backtracking to search for better solutions.

Since exhaustive search of the variable space is expensive, we propose to integrate some pruning techniques in the standard DPLL algorithm to significantly reduce the time complexity. We show that the pruning techniques are effective and DPLL-OPT requires only moderate overhead to find the optimal solution.

Based on the DPLL-OPT algorithm, Plan-A operates in a similar way as SATPlan that keeps increasing the makespan. For a fixed makespan, Plan-A uses DPLL-OPT to either decide the instance is unsatisfiable or finds a plan that minimizes the objective function. Like SATPlan$^{\prec}$, Plan-A can only guarantee optimality for a given makespan. However, through extensive experimentation, we empirically found that in most (over 94%) planning problems, the cost optimal solutions also have the shortest makespan, which implies that the optimal solution returned from the first feasible makespan has a high probability to be the global optimal solution. Further, the pruning techniques employed by DPLL-OPT allows SAT instances for longer makespans to be solved much faster, using the incumbent solution to prune the space. Therefore, Plan-A has an efficient anytime version that keeps increasing the makespan and reports better solutions if found.

[*]This work is supported by a Microsoft Research New Faulty Fellowship and NSF Grant IIS-0713109.

Algorithm 1: Plan-A()

```
Input: a STRIPS planning problem $P$
Output: a solution plan with minimum action-cost
1 set an initial value of $k$ ;
2 while $k \leq MaxLayer$ do
3     encode $P$ with $k$ layers into a SAT instance $S$ ;
4     call DPLL-OPT($S$);
5     if a solution is found then
6         update the incumbent solution;
7         update the incumbent objective value ;
8     $k = k + 1$ ;
```

According to the experimental results, we see that Plan-A is much faster than SATPlan$^{\prec}$. Comparing to SATPlan and other suboptimal planners, Plan-A obtains solutions of better quality, generally with longer solution time.

Overview of Plan-A

We give the overall process of Plan-A in Algorithm 1, which follows the structure of SATPlan.

Plan-A first uses a reachability analysis to find the minimum number of layers k needed for any solution plan (Line 1). The main while loop begins with encoding a STRIPS planning problem P of k layers to a SAT instance S (Line 3), then calls the procedure DPLL-OPT developed below to solve S (Line 4). When a solution is found at a layer k (Line 5), we may keep increasing k to find better solutions.

There is no theoretical guarantee of optimality. However, as we show in the experimental results on all STRIPS domains in the recent International Planning Competitions (IPCs), for over 94% of the problems, the optimal solution is found at the first feasible layer. For all problems we have tested, the optimal solution can be found within three layers beyond the first feasible layer. Therefore, once the first feasible layer is reached, we may keep searching for a limited number of layers to see if there are any better solutions.

The key part of Plan-A is the DPLL-OPT algorithm that finds solutions optimal in terms of the total action cost, which we discuss in the next section. We remark that the problem is different from the MAX-SAT problem whose goal is to minimize the number of unsatisfied clauses, and different from weighted MAX-SAT which assigns weights to unsatisfied clauses.

Optimization with SAT Constraints

Most modern SAT solvers are based on the DPLL (Davis, Logemann, & Loveland 1962) framework, along with other techniques such as clause learning (Marques-Silva & Sakallah 1996) and boolean constraint propagation (Moskewicz *et al.* 2001). We first briefly review the DPLL algorithm and then propose our new algorithm that can optimize an objective function subject to the SAT constraints.

The DPLL algorithm

There are many variations of the DPLL algorithm. Here, we largely adopt the implementation used in the MiniSat solver (Eén & Sörensson 2004). The main procedure of the DPLL algorithm is shown in Algorithm 2. DPLL() is a conflict-driven procedure that detects and resolves conflicts until a satisfiable solution is found. After some initialization, the main loop starts by calling propagate() (Eén & Sörensson 2004), which propagates the first literal p in the propagation queue and returns a conflict if there is any (Line 3). If no conflict occurs and all literals are assigned, a solution is found.

Algorithm 2: DPLL(S)

```
Input: SAT problem $S$
Output: a satistiable solution
1  initialize the solver;
2  while $true$ do
3      $conflict \leftarrow$ propagate();
4      if $conflict$ then
5          $learnt \leftarrow$ analyze($conflict$);
6          add $learnt$ to the clause database;
7          if top-level conflict found then
8              return UNSAT;
9          else
10             backtrack();
11     else
12         if all variables are assigned then
13             return SAT;
14         else
15             decide();
```

If no conflict occurs but there are unassigned literals, it calls decide() to select an unassigned variable p, assign it to be $true$, and insert it into the propagation queue. Consequently, those clauses related to variable p will also be propagated, leading to more variable assignments. Each literal has a decision level. Those newly assigned literals have the same decision level as p. Starting from zero, the decision level is increased by one each time decide() is called.

Once a conflict occurs, the procedure analyze() analyzes the conflict to get a learnt clause (Line 5) and adds the learnt clause to the clause database (Line 6). The learnt clauses help enhance the effectiveness of constraint propagation. We refer to (Eén & Sörensson 2004) for details of generating the learnt clause. After the learnt clause is added, it calls backtrack() to cancel the assignments that result in the conflict (Line 10). The backtrack() procedure keeps undoing the variable assignments with a decreasing decision level until exactly one of the variables in the learnt clause becomes unassigned (they are all $false$ when a conflict occurs).

The DPLL-OPT algorithm

Unlike the standard DPLL algorithm which stops whenever a solution is found, DPLL-OPT searches the whole space to solve an optimization problem defined as follows.

Given a SAT instance encoding a planning problem, suppose it has n variables $(x_1, x_2, \ldots, x_n)$, DPLL-OPT mini-

Algorithm 3: DPLL-OPT(S)

```
Input: SAT problem $S$
Output: a solution with minimum $cost$
1  initialize the solver;
2  $\tau \leftarrow \infty$ ;
3  $num \leftarrow 0$
4  while $true$ do
5      $conflict \leftarrow$ propagate();
6      if $conflict$ then
7          $learnt \leftarrow$ analyze($conflict$);
8          if $conflict$ is of top-level then
9              return $num > 0$ ? SAT:UNSAT;
10         else
11             add $learnt$ to the clause database;
12             backtrack();
13     else
14         let $cost(V)$ be the cost of the current partial
           assignment $V$;
15         if $cost(V) \geq \tau$ then
16             pruning($V$);
17         else
18             if all variables are assigned then
19                 add_blocking_clause($V$);
20             else
21                 decide();
```

mizes the objective function:

$$cost(V) = \sum_{i=1}^{n} c_i v_i,$$

in which $v_i = 1$ if variable x_i is *true* and $v_i = 0$ if x_i is *false*.

The SAT encoding we use is based on SATPlan (Kautz 2006), which has three different types of variables, including those for facts, actions and dummy actions. We assign each action variable a non-negative cost, and all the other variables a zero cost. Therefore, the cost of each variable $c_i, i = 1, \cdots, n$ is defined as:

$$c_i \begin{cases} \geq 0, & x_i \text{ corresponds to an action} \\ = 0, & x_i \text{ corresponds to a dummy action or a fact} \end{cases}$$

More generally, for a partial assignment V where some of the variables are $free$, we can still calculate $cost(V)$ by excluding the unassigned variables in the summation.

Algorithm 3 illustrates the DPLL-OPT algorithm, which largely follows the DPLL framework.

We initialize to infinity τ, the upper bound of the minimum objective value. There are two major changes. First, to enable continued search after satisfiable solutions are found, we do not terminate the search but instead add a blocking clause which prevents the search to yield previously found solutions (Line 19). Second, we employ a pruning strategy that prunes the search tree whenever the cost of the current partial assignment is greater than τ (Line 16).

Algorithm 4: add_blocking_clause(V)

```
Input: a solution $V$
1  $num$++;
2  $\tau \leftarrow cost(V)$;
3  update the incumbent solution;
4  create a blocking clause $M$;
5  add $M$ to the clause database;
6  $learnt \leftarrow$ analyze($M$);
7  add $learnt$ to the clause database;
8  backtrack();
```

Blocking clauses Each time a solution is found, we add a corresponding blocking clause to the clause database. By representing the negation of a satisfying solution, the blocking clause guarantees that the visited solution will not be found again. According to our experiments, there are usually a large quantity (up to hundreds of thousands) of satisfiable solutions to each individual SAT instance. As a result, we need to prune the search tree.

Given a SAT problem with n variables $(x_1, x_2, ..., x_n)$, suppose we have a valid solution $V = (v_1, v_2, \ldots, v_n), v_i \in \{true, false\}$, we synthesize a blocking clause M as:

$$M = \bigvee_{i=1}^{n} literal(V, i),$$

where $literal(V, i) = \begin{cases} \overline{x_i}, & v_i = true \\ x_i, & v_i = false \end{cases}$.

The blocking clause ensures that any future solution found by the search will differ from the current solution by at least one variable. For example, if a solution to a 5-variable SAT problem is $(x_1, x_2, x_3, x_4, x_5) = (true, false, true, true, false)$, then we add the blocking clause $\overline{x_1} \vee x_2 \vee \overline{x_3} \vee \overline{x_4} \vee x_5$.

Algorithm 4 gives details of add_block_clause(), which is invoked when the solver finds a new solution. It starts by increasing the counter of the number of solutions num (Line 1) and updating the minimum cost τ with the cost of the new solution (Line 2). Then, a solution plan will be recorded (Line 3). After that, we create a blocking clause (Line 4) and add it to the clause database (Line 5) to ensure that the solver will never generate the same solution later. The clause will provide an additional constraint that enhances constraint propagation. Using the blocking clause, a learnt clause will be produced by analyze() (Line 6) and added to the clause database (Line 7). We can do this because the current solution violates the blocking clause and thus can be treated as a conflict. Finally, the procedure will undo assignments until precisely one of the literals of the learnt clause becomes unassigned (Line 8).

Pruning clauses Once a solution is found, we update the threshold τ with its cost value. In the following search, if the cost of the current partial assignment V exceeds τ ($cost(V) > \tau$), we consider it a deadend since no further assignments could make the cost any lower. In this case, we call pruning() to stop propagating and backtrack. This pruning technique is crucial for speeding up the search. Without

Algorithm 5: pruning(V)

```
Input: A partial assignment $V$
1 create a pruning clause $M$;
2 $learnt \leftarrow$ analyze($M$);
3 add clause $learnt$ to the clause database;
4 backtrack();
```

this pruning, we found that finding all the solutions for a give SAT instance is prohibitively expensive.

Given a SAT instance of n variables $(x_1, x_2, ..., x_n)$, suppose the current assignment is $V = (v_1, v_2, \dots, v_n), v_i \in \{true, false, free\}$ and it turns out that $cost(V) \geq \tau$, We synthesize a pruning clause M as:

$$M = \bigvee_{i=1}^{n} literal(V, i),$$

where

$$literal(V,i) = \begin{cases} \overline{x_i} & v_i = true \\ x_i & v_i = false \\ false & v_i = free \end{cases}$$

For example, suppose we have an assignment: $(x_1, x_2, x_3, x_4, x_5) = (true, false, free, false, true)$, the pruning clause will be $\overline{x_1} \vee x_2 \vee x_4 \vee \overline{x_5}$.

Algorithm 5 gives the details of the prune() procedure. It starts by creating a pruning clause given the partial assignment V (Line 1). Then, it generates a learnt clause from the pruning clause (Line 2) and adds it to the clause database (Line 3). Finally, we do backtracking to undo the assignments (Line 4), since it is considered a deadend due to the upper bound τ.

References

Chen, Y.; Zhao, X.; and Zhang, W. 2007. Long distance mutual exclusion for propositional planning. In *Proceeding of International Joint Conference on Artificial Intelligence*, 1840–1845.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. In *Communications of the ACM*, volume 5.

Eén, N., and Sörensson, N. 2004. An Extensible Sat-solver. *Theory and Applications of Satisfiability Testing* 502–518.

Giunchiglia, E., and Maratea, M. 2006. optsat: A Tool for Solving SAT Related Optimization Problems. In *JELIA*, volume 4160, 485–489.

Giunchiglia, E., and Maratea, M. 2007. SAT-Based planning with minimal-$\sharp$actions plans and ”soft” goals. In *Artificial Intelligence and Human-Oriented Computing*, volume 4733, 422–433.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of 13th National Conference on Artificial Intelligence(AAAI-96)*, 1194–1201. AAAI.

Kautz, H. 2004. SATPLAN04: Planning as satisfiability. In *Proceedings of IPC4, ICAPS*.

Kautz, H. 2006. SatPlan: Planning as Satisfiability. In *Booklet of the 5th International Planning Competition*.

Marques-Silva, J. P., and Sakallah, K. A. 1996. GRASP-A New Search Algorithm for Satisfiability. In *ICCAD. IEEE Computer Society Press*.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineerying an Efficient SAT Solver. In *Proc. of the 38^{th} Design Automation Conference*.