Accelerating SAT Based Planning with Incremental SAT Solving

Stephan Gocht
Karlsruhe Institute of Technology
Karlsruhe, Germany
stephan.gocht@student.kit.edu

Tomáš Balyo
Karlsruhe Institute of Technology
Karlsruhe, Germany
tomas.balyo@kit.edu

Abstract

One of the most successful approaches to automated planning is the translation to propositional satisfiability (SAT). We employ incremental SAT solving to increase the capabilities of several modern encodings for SAT based planning. Experiments based on benchmarks from the 2014 International Planning Competition show that an incremental approach significantly outperforms non incremental solving. Although we are using sequential scheduling of makespans, we can outperform the state-of-the-art SAT based planning system Madagascar in the number of solved instances.

Introduction

One of the most successful approaches to automated planning is encoding the planning problem into satisfiability (SAT) formulas and then use a SAT solver to solve them. The method was first introduced by Kautz and Selman (**?**) and is still very popular and competitive. This is partly due to the power of SAT solvers, which are getting more efficient year by year. Since then many new improvements have been made to the method, such as new compact and efficient encodings (**?**; **?**; **?**; **?**).

In this paper we apply incremental SAT solving in two different ways to state-of-the-art encodings of SAT based planning. The advantage of incremental SAT solving is that the SAT formula can be modified and solved again, while reusing information from previous solving steps.

We will use sequential scheduling and a SAT solver, which is not optimized for planning. There exist better ways of scheduling the makespans (**?**) and other improvements such as modifying the SAT solver's heuristics to be more suitable for solving planning problems (**?**). These features are not implemented within the current version of our planner but are planned as future work.

The experimental results (for a preview see Figure 1) are based on benchmarks from the International Planning Competition (IPC) 2014 and confirm previous work (**?**; **?**) on the reuse of learned clauses in planning: The incremental version is a significant improvement over the non incremental version and our approach can even outperform the state-of-the-art SAT-based planner Madagascar.

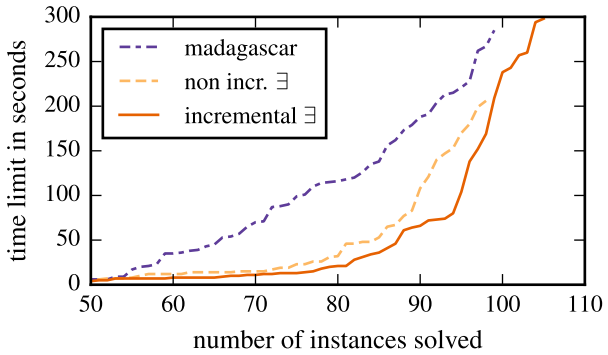


Figure 1: Comparison of Madagascar and our approach using the exists-step encoding on IPC 2014 benchmarks.

Preliminaries

Incremental SAT Solving

A *clause* is a disjunction (OR) of literals, a *literal* is a Boolean variable or its negation and a *Boolean variable* is variable with two possible values (True and False). A *conjunctive normal form* (CNF) *formula* is a conjunction (AND) of clauses. A CNF formula is satisfiable if there is an assignment of truth values to its variables that satisfies at least one literal in each clause of the formula.

The idea of incremental SAT solving is to utilize the effort already spent on a formula to solve a slightly changed but similar formula. The assumption based interface (**?**) has two methods. One adds a clause C and the other solves the formula with additional assumptions in form of a set of literals A:

$$add(C)$$
$$solve(assumptions = A)$$

Note that we will add arbitrary formulas, but they will be transformable to CNF trivially. The method $solve$ determines the satisfiability of the conjunction of all previously added clauses under the condition that all literals in A are true. Note that it is only possible to extend the formula, not to remove parts of the formula. However, this is no restriction. If we want to add a clause C we plan to remove later

we add it with an activation literal: Instead of adding C we add $(a \vee C)$. If the clause needs to be active, $\neg a$ is added to the set of assumptions for the solve step. Otherwise, no assumption is added and the solver can always satisfy the formula by assigning True to a.

SAT-Based Planning

A planning problem is to find a plan – a sequence of actions, that transforms the initial state into a goal state. The basic idea of solving planning as SAT (**?**) is to express whether a plan of length i exists as a Boolean formula F_i such that if F_i is satisfiable then there is a plan of length i. Additionally, a valid plan must be constructible from a satisfying assignment of F_i. To find a plan the plan encodings $F_0, F_1, \ldots$ are checked until the first satisfiable formula is found, which is called sequential scheduling.

There also exist more advanced algorithms to schedule the solving of plan encodings with different makespans (**?**). Since our goal is only to check the effect of incremental SAT solving, we will use the basic sequential scheduling algorithm.

Representation of the Plan Encoding

To apply incremental SAT solving it is necessary to break the plan encoding down to its essential parts. While an arbitrary encoding does not necessarily have this structure, all existing encodings already use this structure or are easily expressed within the presented terms.

The variables of the plan encoding F_i are divided into $i+1$ groups called *time points* with the same number of variables N, $v_k@j$ represents variable k at time point t_j. The clauses of F_i are divided into four groups:

- initial clauses $\mathcal{I}$: satisfied in the initial state t_0
- goal clauses $\mathcal{G}$: satisfied in the goal state t_i
- universal clauses $\mathcal{U}$: satisfied at every time point t_j
- transition clauses $\mathcal{T}$: satisfied at each pair of consecutive time points $(t_0t_1, t_1t_2, \ldots, t_{i-1}t_i)$

The clauses of $\mathcal{I}, \mathcal{G}, \mathcal{U}$ operate on the variables of one time point and $\mathcal{T}$ operates on the variables of two time points. $\mathcal{T}(j,k)$ indicates that the transition clauses are applied from time point j to time point k and similarly for $\mathcal{I}, \mathcal{G}, \mathcal{U}$. The plan encoding F_i for makespan i can be constructed from these clause sets:

$$F_i = \mathcal{I}(0) \wedge \left(\bigwedge_{k=0}^{i-1} \mathcal{U}(k) \wedge \mathcal{T}(k, k+1) \right) \wedge \mathcal{U}(i) \wedge \mathcal{G}(i)$$

As $\mathcal{U}$ is never used alone we can simplify F_i to

$$F_i = \mathcal{I}(0) \wedge \left(\bigwedge_{k=0}^{i-1} \mathcal{T}'(k, k+1) \right) \wedge \mathcal{G}'(i)$$

where

$$\mathcal{T}'(j,k) := \mathcal{U}(j) \wedge \mathcal{T}(j,k)$$
$$\mathcal{G}'(k) := \mathcal{U}(k) \wedge \mathcal{G}(k)$$

Appending Time Points Incrementally

With sequential scheduling of makespans the plan encodings are newly generated in every step and the SAT solver does not learn anything from previous attempts. With an incremental SAT solver it is possible to append a new time point in each step. The trivial way is to add an activation variable to the goal clauses. This allows activating only the latest goal clause and extend the formula by one transition in each step.

$$\begin{aligned} step(0):\ & add\big(\mathcal{I}(0) \wedge [a_0 \vee \mathcal{G}'(0)]\big) \\ & solve(assumptions = \{\neg a_0\}) \\ step(k):\ & add\big(\mathcal{T}'(k-1,k) \wedge [a_k \vee \mathcal{G}'(k)]\big) \\ & solve(assumptions = \{\neg a_k\}) \end{aligned}$$

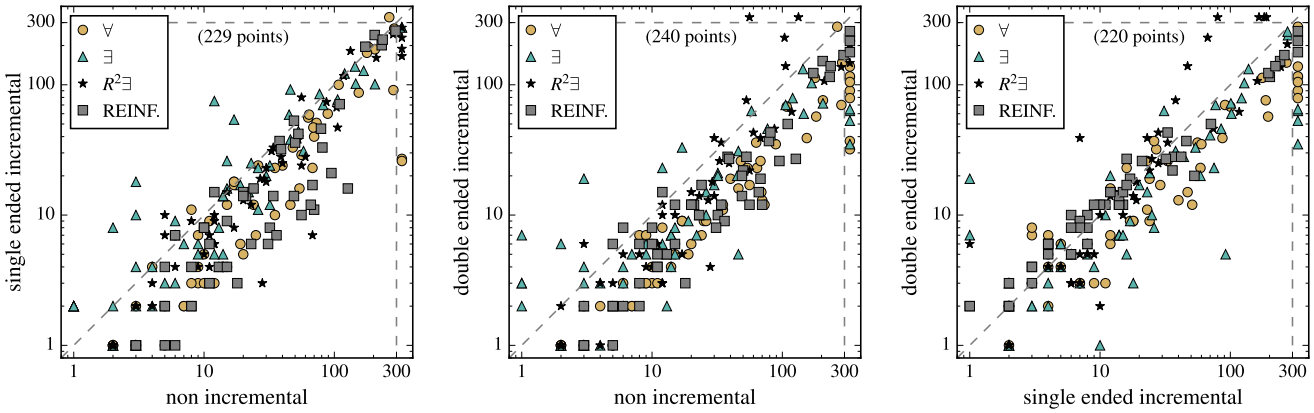
We will call this approach *single ended incremental encoding* as it can be understood as a single stack: New time points are pushed to the top. The bottom of the stack contains the first time point with the initial clauses and the goal clauses are only applied to the time point at the top. Intermediate time points are linked together with transition clauses.

This solution still has one disadvantage: The solver will not be able to apply clauses learned from the goal clauses in future steps as the goal clauses will not be activated. To avoid this problem two stacks can be used instead of one, which will be called *double ended incremental encoding*. One stack contains the time point with initial clauses at the bottom, the other contains the time point with the goal clauses at the bottom. New time points are pushed alternating to both stacks. The time points at the top of both stacks are linked together with link clauses, such that they represent the same time point, i.e. each variable has the same value in both time points: $\mathcal{L}(j,k) := \wedge_{l=1}^{N} v_l@j \Leftrightarrow v_l@k$. Activation variables ensure that only the latest link is active.

$$\begin{aligned} & step(0): \\ & \quad add\big(\mathcal{I}(0) \wedge \mathcal{G}'(n) \wedge [a_0 \vee \mathcal{L}(0,n)]\big) \\ & \quad solve(assumptions = \{\neg a_0\}) \\ & step(2k+1): && \text{add } t_{k+1} \\ & \quad add\big(\mathcal{T}'(k,k+1) \wedge [a_{2k+1} \vee \mathcal{L}(k+1, n-k)]\big) \\ & \quad solve(assumptions = \{\neg a_{2k+1}\}) \\ & step(2k): && \text{add } t_{n-k} \\ & \quad add\big(\mathcal{T}'(n-k, n-k+1) \wedge [a_{2k} \vee \mathcal{L}(k, n-k)]\big) \\ & \quad solve(assumptions = \{\neg a_{2k}\}) \end{aligned}$$

Note that n is neither a precomputed number nor a fixed upper bound but a symbol which always represents the last time point and $n-k$ is the k^{th} time point before the last. In step zero there is no transition between the first time point 0 and the last time point n. Therefore, both time points are the same. In step one there is one transition between the first and the last time point. In step two there are two transitions in between and so on.

With this encoding the solver is able to perform unit propagation and resolve new learned clauses with initial and goal state. The motivation is that connecting initial and goal state with too few transitions causes the conflict. Therefore, we add new transitions in between initial and goal state instead of adding a new goal state.



(a) Non Incremental vs Single Ended Incremental
Excluding Encoding Time

(b) Non Incremental vs Double Ended Incremental
Excluding Encoding Time

(c) Single Ended Incremental vs Double Ended Incremental
Excluding Encoding Time

Figure 2: The Scatter Plots show a data point for each problem. Trivial and difficult problems are not shown, i.e. problems solved by none of the approaches or solved within one second by both. If a problem is only solved by one approach but not the other it is plotted behind the 300 second mark. One configuration is faster than the other if the problem is plotted on the opposing side.

Experiments

The introduced approaches are implemented and available at GitHub[1] with detailed instructions to reproduce the experiments. The experiments were run on a computer with two Intel® Xeon® E5-2683 v4 CPUs (32 cores with 2.10GHz) and 512 GB of memory. Each instance solving a benchmark had a runtime limit of 300 seconds and resource limitation to 1 CPU core and 8 GB of memory. The benchmark problems are from the Agile Track of the 2014 International Planning Competition (IPC) (**?**): 280 problems are divided into 14 domains with 20 problems each. Three of the domains (openstacks, transport and visitall) do not occur in Table 1 because none of the benchmarked approaches was able to solve any problem from those domains within the time limit of 300 seconds.

Communication with the SAT solver is implemented via the IPASIR interface introduced in the 2015 SAT Race (**?**) which means that any SAT solver supporting IPASIR can be used in our planner. To select the incremental SAT solver for our experiments we evaluated all solvers from the Incremental Library Track of the 2016 SAT Competition. We chose *COMiniSatPS 2Sun nopre* (**?**) since it was able to solve the highest number of instances within the time limit. As SAT solvers are changing rapidly we decided to diversify our experiments with different encodings and select a single solver.

In our experiments we used two encodings from the solver Madagascar (**?**) – the forall-step (∀) and exists-step (∃) and two encodings from freelunch – reinforced (reinf.) (**?**) and relaxed relaxed exists-step ($R^2\exists$) (**?**).

Each sub figure in Figure 2 shows a logarithmic scatter plot comparing two different approaches and Table 1 summarizes the number of solved instances in each domain.

All encodings clearly profit from the single ended incremental encoding (Figure 2a). The double ended incremental encoding leads to further improvement (Figure 2b). However, there are some problems where the single ended incremental encoding is still better (see Figure 2c).

The difference between the Madagascar version MpC and double ended incremental ∃ encoding (Figure 1) is not solely caused by the incremental approach, but already present in the non incremental version as can be seen in Figure 3 and Table 1. The chosen SAT solver seems to be better than MpCs internal SAT solver. This is also true, if we change Madagascar to use the VSIDS heuristic instead of the planing specific heuristic.

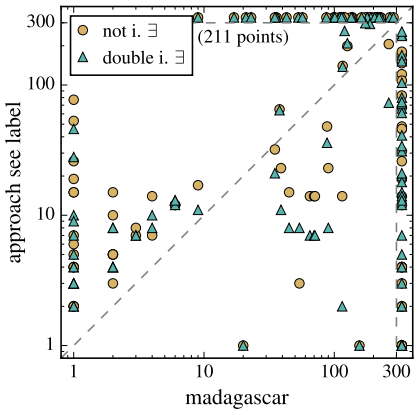


Figure 3: Comparison to Madagascar (MpC)
Including Encoding Time

[1] https://github.com/StephanGocht/incplan

encoding	incremental	barman	cavediving	childsnack	citycar	floortile	ged	hiking	maintenance	parking	tetris	thoughtful	total
MpC	-	**4**	5	7	9	**20**	**12**	6	14	**9**	**8**	**5**	99
$\forall$	non incremental	0	**7**	18	16	9	0	6	**20**	0	3	**5**	84
	single ended	0	**7**	**20**	15	9	0	6	**20**	0	3	**5**	85
	double ended	0	**7**	**20**	16	15	0	6	**20**	0	3	**5**	92
$\exists$	non incremental	0	**7**	18	16	**20**	0	**8**	**20**	0	4	**5**	98
	single ended	0	**7**	19	17	**20**	0	**8**	**20**	0	4	**5**	100
	double ended	0	**7**	**20**	**18**	**20**	0	**8**	**20**	0	7	**5**	**105**
$R^2\exists$	non incremental	0	**7**	19	1	17	0	3	**20**	0	7	**5**	79
	single ended	0	**7**	**20**	1	19	0	3	**20**	0	**8**	**5**	83
	double ended	0	**7**	19	1	19	0	2	**20**	0	6	**5**	79
reinf.	non incremental	0	**7**	18	2	13	0	5	3	0	3	**5**	56
	single ended	0	**7**	18	2	13	0	5	3	0	3	**5**	56
	double ended	0	**7**	18	2	15	0	5	3	0	4	**5**	59

Table 1: Number of problems solved within 300 seconds by default Madagascar (MpC) and the four tested encodings.

We believe that MpC is able to solve problems in the domains barman, ged and parking (see Table 1) due to the difference in scheduling makespans: Our incremental approach uses only sequential scheduling and the achieved makespans are rather small. The maximal makespan of a solution from the double ended incremental $\exists$ encoding is 18. In contrast, MpCs maximal makespan is 529 and the makespans are above 90 for problems only solved by MpC. Therefore, applying advanced scheduling methods is a necessary future work.

Related Work

Incremental SAT solving is an established technique in SAT based model checking. It was first applied for temporal induction (**?**), which is similar to SAT based planning but allows detecting if the goal state is unreachable. Recently it is used for the IC3 algorithm (**?**) and its variants (**?**) to refine an abstraction of reachable states.

Lemma reusing (**?**) is the foundation for reuse of learned clauses in the context of planning. The idea is to extract learned clauses, when the SAT encoding for makespan n is unsatisfiable and add them to the SAT encoding for makespan $n + 1$. This is comparable to our Single Ended Incremental approach but does not need activation literals. Instead, there are limitations on the learned clauses and the encoding. This is to ensure the reusability of learned clauses. A problem **?** encountered is that reusing all learned clauses may be harmful. With the use of an incremental SAT solver both aspects are delegate to the SAT solver.

Another approach to retain learned clauses is to use a single call to a SAT solver (**?**). To get a solution with the smallest makespan it is necessary to change the SAT solver such that it assigns the activation variables first. Encoding all possible makespans into one encoding is usually not feasible due to memory constraints. Therefore, the approach requires an upper bound to the makespan, in contrast to our approach. The disadvantage is that additional SAT solver calls are necessary if no plan is found within the provided upper bound, in which case no information is reused.

Finally, incremental satisfiability modulo theories (SMT) solving was used for planning (**?**). However, the focus of **?** is to add information about the physical feasibility of an action based on motion planning. Our focus is to preserve learned clauses while increasing the makespan.

Conclusion

In this work we implemented and evaluated incremental SAT-based planning. The experiments clearly show that this approach is very beneficial. The time needed to find a plan is reduced and the number of solved instances increased across different encodings. Additionally, the advantage can be increased further if the incremental solver is able to learn clauses based on the goal clauses.

We were able to increase the number of solved instances compared to the state-of-the-art SAT based planner Madagascar, but we did not outperform it in all domains. To combine Madagascars more advanced scheduling techniques and planning heuristics with incremental SAT solving is left for future work.

Acknowledgments. This work was partially supported by DFG grants SA 933/11-1.

References