Karlsruhe Institute of Technology

Incremental SAT Solving for SAT Based Planning

Master's Thesis of

Stephan Gocht

at the Department of Informatics
Institute of Theoretical Informatics, Algorithmics II

Advisor:	Dr. Tomáš Balyo
Second advisor:	Prof. Peter Sanders

August 2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 8th August 2017

. . . .

(Stephan Gocht)

Abstract

One of the most successful approaches to automated planning is the translation to propositional satisfiability (SAT). This thesis evaluates incremental SAT solving for 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, planning specific heuristics and advanced scheduling of makespans is not used, it is possible to outperform the state-of-the-art SAT based planning systems Madagascar and PDRPlan in the number of solved instances.

Acknowledgments

I thank Prof. Sanders for providing the resources to accomplish this work and Dr. Balyo for his expert advice and for encouraging me to publish my work. Finally, I thank the anonymous reviewers of my submitted papers, as their feedback helped me to improve my work.

Contents

List of Figures

List of Tables

1 Introduction

Planning is the task of finding a sequence of actions, i.e. a plan, that leads to the desired goal. To automate this process is a major aspect in artificial intelligence and there are numerous approaches to solve this problem. One of the most successful approaches is to use a satisfiability (SAT) solver to solve the planning problem after it has been transformed into a SAT formula. This method was introduced by Kautz and Selman [23] in 1992 and is still competitive.

The reasons for the success of SAT based planning are twofold: First, the performance of SAT solvers does increase every year, as can be seen in the yearly SAT competitions. And second, improvements to the method of SAT based planning itself have been made, such as new and more efficient encodings [6, 21, 40, 41].

A major obstacle in SAT based planning is that a SAT formula can only encode plans up to a fixed length, but the length of the searched plan is not known beforehand. One possible solution is to find upper bounds to the length of the plan. These upper bounds are usually too large to be practically useful but recent advances in this area are promising [1].

The classical solution is to use multiple calls to a SAT solver. This method can be improved by using an incremental SAT solver, which allows solving similar formulas and to carry over useful information between solve steps.

This thesis studies incremental SAT solving, for SAT based planning. A preview of the results can be found in Figure 1.1: The developed tool is competitive to state-of-the-art SAT based planners solely by using a recent SAT solver, as the non-incremental variant shows. By using incremental SAT solving the performance and number of solved instances can be improved further.

The thesis starts with Section 2 that provides an introduction to incremental SAT solving and SAT based planning. Section 3 discusses related approaches that try to keep

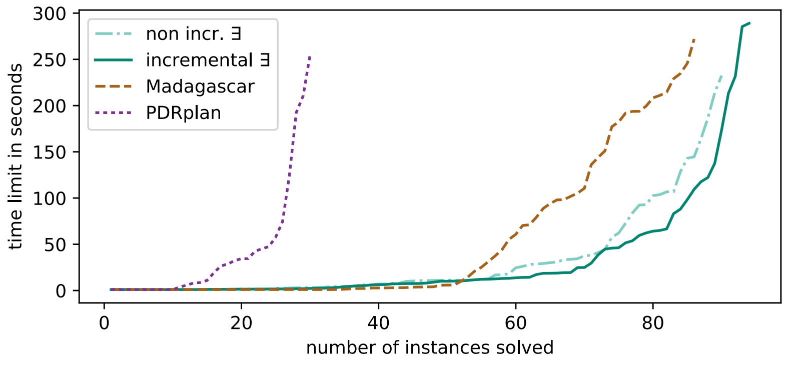


Figure 1.1: Cactusplot comparing the implemented tool to state-of-the-art SAT based planners Madagascar and PDRplan on IPC 2014 benchmarks.

information between multiple calls as well as related applications of incremental SAT solving.

Section 4 provides insights on incremental SAT solving and demonstrates its capabilities and pitfalls on a well studied theoretical problem. An especially interesting result of this section will be that incremental SAT solving is faster than solving the final instance directly. How to employ an incremental SAT solver for planning in different ways is described in Section 5 and contains a completely new approach that lead to a publication at the 27th International Conference on Automated Planning and Scheduling (ICAPS 2017) [18]. The different approaches are evaluated in Section 6: It is shown that the incremental approach improves the state-of-the-art. In addition, this section contains experiments to gather insights on which information is especially useful to carry over and analyzes the potential for further techniques. Finally, the thesis is concluded in Section 7.

2 Foundations

2.1 SAT Basics

The definitions of this section are based on [42]: A Boolean variable is either false (0) or true (1). A *literal* is either a Boolean variable or the negation ($\neg$) of a Boolean variable. A *clause* is a disjunction ($\vee$) of literals or the empty clause ($\bot$). Clause may be written as a set of literals, where the empty clause $\bot$ does not contain any literals. A formula in *conjunctive normal form* (CNF) is a conjunction ($\wedge$) of clauses and may be written as set of clauses. An example for a CNF formula is $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$. As the structure of a CNF is fixed it is clear that $\wedge$ is the top-level operator and the brackets may be left out, i.e. $x_1 \vee \neg x_2 \wedge x_2 \vee x_3$ is the same formula as above. Written as sets it is $\{\{x_1, \neg x_2\}, \{x_2, x_3\}\}$. The set of all variables in a literal, clause or formula X is denoted by $vars(X)$.

Not every Boolean formula is in CNF but in this thesis all formulas will be in CNF or it is trivial to transform them into CNF.

An assignment σ assigns each variable a truth value. Respectively a partial assignment only assigns some variables a truth value. An assignment can be generalized for CNF formulas. Using v to denote a variable, l a literal, C a clause and F a formula in CNF the generalization can be defined as follows: $\sigma(\neg v) := \neg\sigma(v), \sigma(\bot) = 0, \sigma(C) = \sigma(\bigvee_{l \in C} l) := \bigvee_{l \in C} \sigma(l)$ and $\sigma(F) = \sigma(\bigwedge_{C \in F} C) := \bigwedge_{C \in F} \sigma(C)$. An assignment σ is satisfying a formula F in CNF if $\sigma(F) = 1$. A formula is satisfiable if there is a satisfying assignment. Finding a satisfying assignment or showing that no such assignment exists is known as SAT solving.

For example $\sigma(x_1) = 0, \sigma(x_2) = 0, \sigma(x_3) = 1$ is a satisfying assignment of the formula $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$ and therefore the formula is satisfiable.

To refute a formula, i.e. to show a formula is unsatisfiable, it is not necessary to test all possible assignments. Instead, a proof system such as *resolution* can be used. Resolution consists of a single rule: For a variable x and clauses $(C \vee x), (\neg x \vee D)$ it is possible to derive the new clause containing all literals of C and D:

$$\frac{C \vee x \qquad \neg x \vee D}{C \vee D} \tag{2.1}$$

If a formula is unsatisfiable, then it is always possible to derive $\bot$ with this rule [42]. The sequence of derived clauses to show $\bot$ is called transcript.

Definition 1. *Given a CNF formula F, a clause C is a logical consequence of F, written $F \models C$ if and only if every satisfying assignment of F is also a satisfying assignment of C*

Note that if F is an unsatisfiable formula then $\bot$ is a logical consequence of F, i.e. $F \models \bot$. Sometimes we will talk about the conjunction of the negation of all literals of a clause C. This is the same as the negation of the clause by De Morgan's law: $\neg C = \neg(\bigvee_{l \in C} l) = \bigwedge_{l \in C} \neg l$.

Lemma 1. *$F \models C$ if and only if $F \wedge \neg C \models \bot$.*

Proof. " $\Rightarrow$ " $F \models C$ and let σ be a truth assignment such that $\sigma(F) = 1 \Rightarrow \sigma(C) = 1 \Rightarrow \exists l \in C : \sigma(l) = 1 \Rightarrow \sigma(F \wedge \neg C) = \sigma(F) \wedge \overset{l \in C}{\wedge} \neg\sigma(l) = 0$.

" $\Leftarrow$ " $F \not\models C$ then there is a truth assignment σ such that $\sigma(F) = 1$ and $\sigma(C) = 0 \Rightarrow \bigvee_{l \in C} \sigma(l) = 0 \Rightarrow \forall l \in C : \sigma(l) = 0 \Rightarrow 1 = \bigwedge_{l \in C} \sigma(\neg l) = \sigma(\bigwedge_{l \in C} \neg l) = \sigma(\neg C) \Rightarrow \sigma(F \wedge \neg C) = 1$. Therefore, σ is a satisfying assignment of $F \wedge \neg C$ and $F \wedge \neg C \not\models \bot$ □

2.2 Incremental SAT Solving

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* [14] has two methods that change the internal state of the solver. One adds a clause C and the other solves the formula with additional assumptions in form of a set of literals A:

$$\begin{gathered} add(C) \\ solve(assumptions = A) \end{gathered}$$

Note that we will add arbitrary formulas, but they will be transformable to CNF trivially. Thereafter, they can be added with multiple calls to *add*. 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 not a restriction. If we want to add a clause C we plan to remove later we add it with the negation of a fresh variable a, i.e. a variable not used before. This is called using an *activation literal*: Instead of adding C we add $(\neg a \vee C)$. If the clause needs to be active, a is added to the set of assumptions for the solve step. Otherwise, no assumption is added and the solver can always satisfy the clause by assigning false to a.

The solver is allowed to extend the set of clauses F in its internal state with a new clause C, if C is a consequence of F, i.e. $F \models C$. Such a clause will be called a *learned clause*. As clauses can not be removed, learned clauses will be valid for all subsequent calls of *solve*. This is important as a common technique is conflict driven clause learning (CDCL) [44], where new clauses are added based on conflicts, which arise for partial assignments during the *solve*-step. To retrieve the learned clauses there is a method *learned*$(\cdot)$:

$$learned(i) := \{C \mid C \text{ is a learned clause from the } i\text{-th call to solve}\}$$

Benefits from incremental SAT solving can arise from different information stored by the solver, such as:

- clauses learned from *conflict driven clause learning* (CDCL) [44]
- stored metrics to select the branching variable, i.e. choosing the next variable to assign a value to, such as the *variable activity* in the heuristic *variable state independent decaying sum* (VSIDS) [27]
- *watched literals* [27], an implementation technique for lazy clause evaluation

- *phase saving* in CDCL [33], i.e. storing the last partial assignment used to decide if true or false should be assigned to the branching variable

That it is sound to reuse learned clauses with the assumption based interface becomes clear when considering the following lemma:

Lemma 2. *Let F_1, F_2 be CNF formulas, C a clause and a fresh Boolean variable a, i.e. $a \notin vars(F_1), a \notin vars(F_2)$. It holds: If $F_1 \wedge (a \vee F_2) \models C$ and $a \notin C$ then $F_1 \models C$.*

Proof. $F_1 \wedge (a \vee F_2) \models C \Leftrightarrow F_1 \wedge (a \vee F_2) \wedge \neg C \models \perp$. If a is set to 1 then $(a \vee F_2)$ is satisfied but as the formula is unsatisfiable and a does not occur in F_1 nor in C it follows that $F_1 \wedge \neg C \models \perp \Leftrightarrow F_1 \models C$. □

From this lemma follows that every learned clause is either guarded by an activation literal or follows from clauses without an activation literal. For SAT practitioners this might not be surprising: the variable a is fresh and therefore might not be removed with resolution. Note however that this result is independent of the underlying solver and would still hold if the solver is not based on resolution.

2.3 Planning

Planning problems are described by a finite state space, actions that manipulate the states and an initial and a goal state. Automated planning is the process of finding a plan, i.e. a sequence of actions.

2.3.1 The SAS+ Formalism

There are different formalisms to describe a planning task such as STRIPS [16], ADL [31], PDDL [26] and SAS+ [5]. Based on the SAS+ formalism a planning task can be defined in the following way [6]:

A planning task is a tuple (X, O, s_I, s_G) where:

- $X = \{x_1, \ldots, x_n\}$ is a set of variables with finite domain $dom(x_i)$. A state assigns each variable with a value and a partial state is an assignment of values to a subset of all variables.
- O is the set of actions (or operators). An action A is a symbol that has a precondition $pre(A)$, which must hold before its execution, and an effect or postcondition $post(A)$, which holds after the execution of the action. $pre(A)$ and $post(A)$ are partial states. Applying an action A to s, i.e. $s' := \text{apply}(A, s)$, therefore requires that $pre(A) \subseteq s$ and $post(A) \subseteq s'$. Moreover, no variable but the ones defined in $post(A)$ may change.
- s_I is a state that describes the initial state
- s_G is a partial state that describes the goal

The planning problem is to find a sequence of actions that transform the initial state to a state that contains the goal state.

2.3.2 Planning as SAT

The basic idea of solving planning as SAT [23] 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 *makespan i*, i.e. a plan with i steps. 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 are also alternative ways of scheduling the makespan: For example using an exponential step size and only solving formulas $F_2, F_4, F_8, F_{16}, \ldots$ or solving different makespans in parallel and finish as soon as a solvable instance is found [37].

The variables of the plan encoding F_i are divided into $i + 1$ groups called *time points* with the same number of variables N, $x@t_j$ represents variable x 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 $(t_0t_1, t_1t_2, \ldots, t_{i-1}t_i)$ of consecutive time points

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. $\mathcal{T}(t_j, t_k)$ indicates that the transition clauses are applied from time point t_j to time point t_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:

$$F_i = \mathcal{I}(t_0) \wedge \left(\bigwedge_{k=0}^{i-1} \mathcal{U}(t_k) \wedge \mathcal{T}(t_k, t_{k+1}) \right) \wedge \mathcal{U}(t_i) \wedge \mathcal{G}(t_i)$$

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

$$F_i = \mathcal{I}(t_0) \wedge \left(\bigwedge_{k=0}^{i-1} \mathcal{T}'(t_k, t_{k+1}) \right) \wedge \mathcal{G}'(t_i)$$

where

$$\mathcal{T}'(t_j, t_k) := \mathcal{U}(t_j) \wedge \mathcal{T}(t_j, t_k) \tag{2.2}$$
$$\mathcal{G}'(t_k) := \mathcal{U}(t_k) \wedge \mathcal{G}(t_k) \tag{2.3}$$

This partitioning is natural and is used similarly in planning [45] and bounded model checking [15]. To my best knowledge all SAT encodings for planning can be expressed in terms of these four sets of clauses. However, this might not be true for future encodings, although it is hard to imagine something different. This thesis and the developed planning tool does only work with this abstraction as the presented principles are independent from the concrete encoding.

2.3.3 Example: Solving the SAT Representation

To get a better idea we will consider a small example that is not based on planning directly but uses the same concepts: There are two variables x_1, x_2 such that x_1 does change its state with every transition and x_2 is false unless itself or x_1 was true in the time point before. The initial state is $x_1 = 0$ and $x_2 = 0$, the goal state is $x_1 = 0$ and $x_2 = 1$. This can be described as:

$$\begin{aligned}
\mathcal{I}(t) &:= (\neg x_1@t) \wedge (\neg x_2@t) && (2.4)\\
\mathcal{T}(t,t') &:= (\neg x_1@t \vee \neg x_1@t') && (2.5)\\
&\wedge\ \ (x_1@t \vee \ \ x_1@t') && (2.6)\\
&\wedge\ \ (x_2@t \vee \ \ x_1@t \vee \neg x_2@t') && (2.7)\\
\mathcal{G}(t) &:= (\neg x_1@t \wedge \ \ x_2@t) && (2.8)
\end{aligned}$$

The task is to construct a formula F_i as described in the previous section and find the smallest makespan i for which the formula is satisfiable.

To solve this problem the first step is with one time point: $F_0 = \mathcal{I}(t_0) \wedge \mathcal{G}(t_0)$. This formula is obviously not satisfiable. The second step is $F_1 = \mathcal{I}(t_0) \wedge \mathcal{T}(t_0, t_1) \wedge \mathcal{G}(t_1)$. The goal can still not be reached as $\mathcal{I}(t_0) \wedge \mathcal{T}(t_0, t_1) \models x_1@t_1 \wedge \neg x_2@t_1$. The third and final step is $F_2 = \mathcal{I}(t_0) \wedge \mathcal{T}(t_0, t_1) \wedge \mathcal{T}(t_1, t_2) \wedge \mathcal{G}(t_2)$ which is satisfiable. $\mathcal{I}(t_0) \wedge \mathcal{T}(t_0, t_1) \wedge \mathcal{T}(t_1, t_2) \models \neg x_1@t_2$ and as we have seen before x_1 is true at t_1 so x_2 can be set to true at t_2. Note how it is intuitively possible to use information from earlier steps to solve later steps. Doing this within a SAT solver is the idea behind incremental SAT solving.

2.3.4 SAS+ in SAT Based Planning

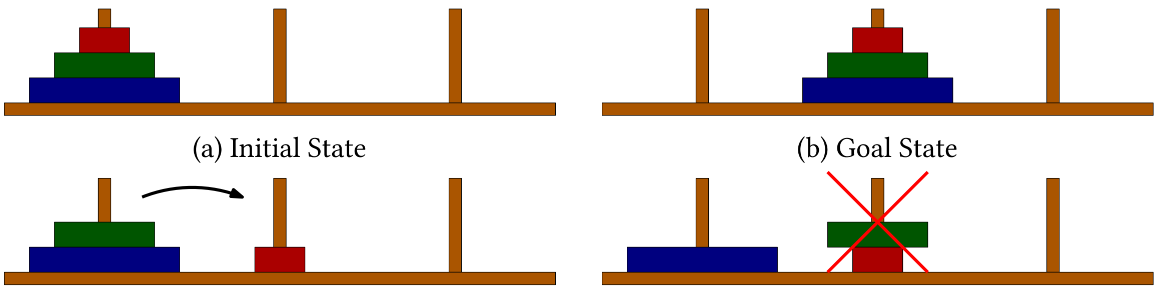
A possible encoding of the SAS+ formalism is as follows: For each variable $x \in X$ and each value $v \in dom(x)$, there is a boolean variable b_v^x which is only true if $x = v$. For each action $A \in O$ there is a boolean variable a_A indicating that action A is applied. The universal clauses are used to ensure that only one action is applied in each time point and each variable has only one value:

$$\mathcal{U}(t) := \text{exact-one}(\{a_A@t \mid A \in O\}) \wedge \bigwedge_{x \in X} \text{exact-one}(\{b_v^x@t \mid v \in dom(x)\}) \qquad (2.9)$$

The exact-one constraint ensures that exactly one literal of a given set of literals L is true. A simple encoding for this constraint is exact-one$(L) := (\bigvee_{l \in L} l) \wedge \bigwedge_{y,z \in L: y \neq z} (\neg y \vee \neg z)$.

The initial state and the goal state can be directly transformed into the according clauses:

$$\begin{aligned}
\mathcal{I}(t) &:= \bigwedge_{(x=v) \in s_I} b_v^x@t && (2.10)\\
\mathcal{G}(t) &:= \bigwedge_{(x=v) \in s_G} b_v^x@t && (2.11)
\end{aligned}$$



(a) Initial State (b) Goal State

(c) Possible First Move (d) Invalid State

Figure 2.1: Towers of Hanoi with 3 Discs

The transition clauses have two parts: The first part ensures that pre- and postcondition hold when an action is applied and the second part ensures that nothing changes, i.e. no variable is set if it was not set before, unless there is an action supporting this change.

$$\begin{aligned}\mathcal{T}(t,t') := \bigwedge_{A\in O} &\left[\left(\bigwedge_{(x=v)\in pre(A)} \neg a_A@t \vee b_v^x@t\right) \wedge \left(\bigwedge_{(x=v)\in post(A)} \neg a_A@t \vee b_v^x@t'\right)\right] \\ \wedge &\bigwedge_{x\in X, v\in dom(x)} \left(\neg b_v^x@t' \vee b_v^x@t \vee \bigvee_{\substack{A\in O:\\(x=v)\in post(A)}} a_A@t\right)\end{aligned} \tag{2.12}$$

The presented encoding of SAS+ to SAT is very straight forward (it is the SAS+ version of the direct encoding [23] and uses notation from the relaxed-relaxed exist encoding [6]). The encodings used for evaluation are actually more sophisticated. A very important technique is to allow the execution of multiple actions within one transition. This is possible if the actions do not interfere, known as forall-semantics (∀), or if there is an applicable order to all actions in one transition, known as exist- semantics (∃). The work of Rintanen, Heljanko, and Niemelä [40] contains a more detailed description of these approaches.

2.3.5 Example: Towers of Hanoi

The towers of Hanoi is a puzzle game. There is an arbitrary number of disks with different sizes and three sticks. The initial state is that all disks are on one stick and are ordered by size, with the smallest disk on top as in Figure 2.1a. The goal state is the same but with all disks moved to the second stick as in Figure 2.1b. It is only allowed to move one disk at a time and a larger disk may never lie upon a smaller disk. Such an invalid game state is shown in Figure 2.1d. A valid first move is shown in Figure 2.1c.

This puzzle with three disks can be encoded as SAS+ in the following way: There is a variable for each disk and one fore each stick. These variables encode which disk is placed on top of the disk or stick:

$$\begin{aligned} X := \{&\text{on-disk}_1, \text{on-disk}_2, \text{on-disk}_2,\\ &\text{on-stick}_1, \text{on-stick}_2, \text{on-stick}_3\} \end{aligned} \tag{2.13}$$

$$\begin{aligned} &\text{for } i \in \{1,2,3\} :\\ &\qquad \mathit{dom}(\text{on-stick}_i) := \{\text{disk}_1, \text{disk}_2, \text{disk}_3, \text{nothing}\} \end{aligned} \tag{2.14}$$

$$\begin{aligned} &\text{for } i \in \{1,2,3\} :\\ &\qquad \mathit{dom}(\text{on-disk}_i) := \{\text{disk}_k \mid 0 < k < i\} \cup \{\text{nothing}\} \end{aligned} \tag{2.15}$$

The subscript indicates the size of the disk, p.a. disk_1 is smaller than disk_3 and $\text{on-disk}_3 = \text{disk}_1$ encodes that disk_1 is on top of disk_3. The domains are designed such that states as shown in Figure 2.1d can not be represented. Initial and goal state are defined by:

$$\begin{aligned} I := \{&\text{on-stick}_1 = \text{disk}_3, \text{on-stick}_2 = \text{nothing}, \text{on-stick}_3 = \text{nothing},\\ &\text{on-disk}_3 = \text{disk}_2, \text{on-disk}_2 = \text{disk}_1, \text{on-disk}_3 = \text{nothing}\} \end{aligned} \tag{2.16}$$

$$\begin{aligned} G := \{&\text{on-stick}_1 = \text{nothing}, \text{on-stick}_2 = \text{disk}_3, \text{on-stick}_3 = \text{nothing}\\ &\text{on-disk}_3 = \text{disk}_2, \text{on-disk}_2 = \text{disk}_1, \text{on-disk}_3 = \text{nothing}\} \end{aligned} \tag{2.17}$$

Finally, there is an action for each move, i.e. for each possible combination of:

- a disk to move
- an origin, where the disk is currently on
- a destination where the disk will be moved to

More formally: The set of all actions $O := \{\text{move}(\text{disk}_i, \mathit{src}, \mathit{dst}) \mid \text{ for } \mathit{src}, \mathit{dst} \in X, i \in \{1,2,3\} \text{ such that } \text{disk}_i \in \mathit{dom}(\mathit{src}) \cap \mathit{dom}(\mathit{dst})\}$. With the pre- and postconditions defined as follows:

$$\mathit{pre}(\text{move}(\text{disk}_i, \mathit{src}, \mathit{dst})) := \{\text{on-disk}_i = \text{nothing}, \mathit{src} = \text{disk}_i, \mathit{dst} = \text{nothing}\} \tag{2.18}$$

$$\mathit{post}(\text{move}(\text{disk}_i, \mathit{src}, \mathit{dst})) := \{\mathit{src} = \text{nothing}, \mathit{dst} = \text{disk}_i\} \tag{2.19}$$

For example, there is one action $\text{move}(\text{disk}_1, \text{on-disk}_2, \text{on-stick}_2)$ for moving disk one from disk two to stick two as in Figure 2.1c. Note, that for this action it is not relevant where disk two is placed on, as long as nothing is placed on stick two. A solution is:
$\text{move}(\text{disk}_1, \text{on-disk}_2, \text{on-stick}_2)$, $\text{move}(\text{disk}_2, \text{on-disk}_3, \text{on-stick}_3)$,
$\text{move}(\text{disk}_1, \text{on-stick}_2, \text{on-disk}_2)$, $\text{move}(\text{disk}_3, \text{on-stick}_1, \text{on-stick}_2)$,
$\text{move}(\text{disk}_1, \text{on-disk}_2, \text{on-stick}_1)$, $\text{move}(\text{disk}_2, \text{on-stick}_3, \text{on-disk}_3)$,
$\text{move}(\text{disk}_1, \text{on-stick}_1, \text{on-disk}_2)$

Transforming this problem into SAT leads to the following clauses:

$$\begin{aligned}\mathcal{I}(t) :=& b^{\text{on-stick}_1}_{\text{disk}_3}@t \wedge b^{\text{on-stick}_2}_{\text{nothing}}@t \wedge b^{\text{on-stick}_3}_{\text{nothing}}@t \\ &\wedge b^{\text{on-disk}_3}_{\text{disk}_2}@t \wedge b^{\text{on-disk}_2}_{\text{disk}_1}@t \wedge b^{\text{on-disk}_1}_{\text{nothing}}@t\end{aligned} \tag{2.20}$$

$$\begin{aligned}\mathcal{G}(t) :=& b^{\text{on-stick}_1}_{\text{nothing}}@t \wedge b^{\text{on-stick}_2}_{\text{disk}_3}@t \wedge b^{\text{on-stick}_3}_{\text{nothing}}@t \\ &\wedge b^{\text{on-disk}_3}_{\text{disk}_2}@t \wedge b^{\text{on-disk}_2}_{\text{disk}_1}@t \wedge b^{\text{on-disk}_1}_{\text{nothing}}@t\end{aligned} \tag{2.21}$$

$$\mathcal{U}(t) := \text{exact-one}(\{a_A@t \mid A \in O\}) \wedge \bigwedge_{x \in X} \text{exact-one}(\{b^x_v@t \mid v \in dom(x)\}) \tag{2.22}$$

$$\begin{aligned}\mathcal{T}(t, t') := \bigwedge_{\substack{src, dst \in X, i \in \{1,2,3\}: \\ \text{disk}_i \in dom(src) \cap dom(dst)}} & \neg a_{\text{move}(\text{disk}_i, src, dst)}@t \vee b^{\text{on-disk}_i}_{\text{nothing}}@t \\ & \wedge \neg a_{\text{move}(\text{disk}_i, src, dst)}@t \vee b^{src}_{\text{disk}_i}@t \\ & \wedge \neg a_{\text{move}(\text{disk}_i, src, dst)}@t \vee b^{dst}_{\text{nothing}}@t \\ & \wedge \neg a_{\text{move}(\text{disk}_i, src, dst)}@t \vee b^{src}_{\text{nothing}}@t' \\ & \wedge \neg a_{\text{move}(\text{disk}_i, src, dst)}@t \vee b^{dst}_{\text{disk}_i}@t' \\ \wedge \bigwedge_{x \in X, v \in dom(x)} & (\neg b^x_v@t' \vee b^x_v@t \vee \bigvee_{\substack{A \in O: \\ (x=v) \in post(A)}} a_A@t)\end{aligned} \tag{2.23}$$

3 Related Work

3.1 Planning

3.1.1 Lemma Reusing

Lemma reusing [28] 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 the later introduced 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 Nabeshima et al. encountered is that reusing all learned clauses may be harmful. With the use of an incremental SAT solver both aspects are delegated to the SAT solver.

3.1.2 Single Call to SAT Solver

Another approach to retain learned clauses is to use a single call to a SAT solver [35]. To get a solution with the smallest makespan it is necessary to change the SAT solver such that it assigns the action variables in the right order. 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, which is not necessary when using incremental SAT solving. 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.

3.1.3 Incremental Solving for Refinement

Finally, incremental satisfiability modulo theories (SMT) solving was used for planning [11]. However, the focus of Dantam et al. is to add information about the physical feasibility of an action based on motion planning. The focus of this thesis is to preserve learned clauses while increasing the makespan.

3.2 Model Checking

The goal of model checking is to verify that property P holds in every state reachable from the initial state. This can be verified by checking whether the goal state defined by $\mathcal{G} := \neg P$ is reachable. As in planning, one outcome is that the goal state is reachable. In this case the path from initial to goal is a counterexample for P holding in every state. The other outcome is that the goal state is not reachable. This outcome is of great interest for model checking as it verifies that the property P does hold in every state. Therefore, it is necessary to either employ mechanisms to detect that the goal state is unreachable or to set an upper bound to the number of analyzed transitions. The latter case is called bounded model checking.

Incremental SAT solving is an established technique in SAT based model checking. It was first applied for temporal induction [14], which is similar to SAT based planning but allows detecting if the goal state is unreachable.

3.2.1 Property Directed Reachability

A more recent use of incremental SAT solving in model checking is for property directed reachability (PDR) known by the IC3 algorithm [9, 13].

The basic idea is to maintain a set of abstractions $A@t_i$, more precisely an over-approximation, for each time point that encodes states reachable at time point t_i in form of a CNF: If a state is reachable at t_i then it is represented by a satisfying assignment of $A@t_i$. To guarantee that $A@t_i$ is an over-approximation it is required that $A@t_0 := \mathcal{I}(t_0)$ and $A@t_i \wedge T(t_i, t_{i+1}) \models A@t_{i+1}$. If abstraction $A@t_i$ contains the goal state, i.e. $A@t_i \wedge G(t_i) \not\models \bot$, then it is either to weak and it is necessary to refine it, or it is possible to construct a solution. This is realized by a call to REFINE with the goal state. Note that states are simply represented by a set $s@t_i$ of literals at time point t_i. Adding the state to a formula simply means adding the conjunction of its literals and the negation is a clause containing the negation of all literals. If the goal state is not contained at t_i it is checked whether it is contained at t_{i+1}, i.e. the abstraction depth is increased.

Algorithm 1 Basic Idea of Refine.

```
 1: $A@t_i$, over-approximation for states reachable at time point $t_i$

    *This function tries to refine the abstraction $A@t_i$ such that the given assignment $s@t_i$
    does not satisfy the abstraction anymore, or returns an assignment for each time point,
    that shows that the given assignment is reachable.*
 2: **function** REFINE($s@t_i$)
 3:     **if** $i = 0$ **then return** $s@t_i$
 4:     **while** true **do**
 5:         **if** $A@t_{i-1} \wedge T(i-1, i) \models \neg s@t_i$ **then**
 6:             $A@t_i \leftarrow A@t_i \wedge \neg s@t_i$
 7:             **return** refined
 8:         **else**
 9:             $s@t_{i-1} \leftarrow$ partial assignment of variables at $t_{i-1}$
                            satisfying $A@t_{i-1} \wedge T(i-1, i) \wedge s@t_i$
10:             $r \leftarrow$ refine($s@t_{i-1}$)
11:             **if** r is not refined **then**
12:                 **return** $r \cup s@t_i$
```

Algorithm 1 describes the basic idea of the refinement step. However, it does not contain important optimizations, that are necessary for an efficient implementation. The basic optimizations are listed below. Please refer to [19] for an overview of more recent variations.

Subsumtion of Abstractions. If the transition allows to keep the state from one time point to another than all states reachable at t_i are reachable at t_{i+t} as well. Therefore, it is reasonable to enforce $A@t_i \models A@t_{i+1}$. This can be realized syntactically by making sure that if a clause C is contained in $A@t_i$ than it is also contained in $A@t_k$ for $0 \leq k < i$ as well (note that clauses restrict the reachable states).

Pushing Clauses Forward. Finding a good strengthening, i.e. clause, of the abstraction is very important. If $C@t_i \in A@t_i$ then it is possible to add $C@t_{i+1}$ to $A@t_{i+1}$ if $A@t_i \wedge T(i, i+1) \models C@t_{i+1}$. The clause is pushed forward.

Detecting Unreachability. A fix-point is reached if the goal state is not contained in $A@t_i$ or $A@t_{i+1}$ and both abstractions contain the same clauses. Checking that both abstractions contain the same clauses is a pure syntactic operation. Such a fix-point is reached eventually if the goal state is unreachable and if the optimizations subsumtion of abstractions and pushing clauses forward are used.

Generalization of the Strengthening. In Line 6 the over-approximation is strengthened to avoid that the same assignment can be generated again. It can happen that not all variables of $s@t_i$ are necessary to show that this state is unreachable. In this case, removing assignments of unnecessary variables before strengthening $A@t_i$ does not only ensure that $s@t_i$ is no longer part of reachable states as before but also that similar unreachable states are no longer part of the over-approximation.

Inductive Generalization of the Strengthening. This technique is used in combination with the subsumtion of abstractions. If $A@t_{i-1} \wedge \neg s@t_{i-1} \wedge T(i-1, i) \models \neg s@t_i$ and $A@t_0 \models \neg s@t_0$ then $\neg s@t_0$ is inductive relative to $A@t_{i-1}$ and can be added to $A@t_k$ for $0 \leq k \leq i$ [9].

Rescheduling of Counterexamples. If refine produces a counter example $s@t_i$, then it is known that it is a predecessor to the goal state. Therefore, it is also of interest whether $s@t_k$ for $k > i$ is feasible as well. This can be checked before the abstraction depth is increased and thereby allows to find paths to the goal state that are larger than the abstraction depth.

No Unrolling. The algorithm does only work with at most two time points in each call to the SAT solver. Therefore, it is not necessary to have different variables for different time points. Instead, variables can be shared and activation literals can be used to activate the clauses for the abstraction to be used.

3.2.2 Incremental Preprocessing

An important technique in SAT solving is to perform preprocessing such as subsumtion, variable elimination [12] and blocked clause elimination [22] to reduce the number of variables and clauses and thereby make the problem easier for a SAT solver. The problem in incremental SAT solving is that it is not generally known which clauses will be added and which assumptions will be used. Thereby, it could happen that clauses or literals are removed, which are necessary for later steps. Due to the strict structure of formulas in bounded model checking it is possible to use look-ahead to determine which variables will be used for assumptions and to connect clauses between steps. These variables will be excluded from simplification steps to guarantee equisatisfiability [25]. These results are directly transferable to SAT based planning.

4 Case Study: Refutation of the Pigeonhole Principle with Incremental SAT Solving

When using the sequential scheduling of makespans for SAT based planing, the most time is spent on refuting the formulas F_i one after the other, i.e. showing them unsatisfiable. The Pigeonhole Principle (PHP) is used to get a better insight on the capabilities and pitfalls of incremental SAT solvers in this scenario. The advantage of the PHP is that it allows to generate an arbitrary number of formulas that are hard to solve for SAT solvers.

4.1 Encodings of PHP

The pigeonhole principle PHP_h^p is the classical problem of fitting p pigeons into h holes, where each hole takes at most one pigeon. We will always use $h := p - 1$ so that the problem is unsolvable. For $n \in \mathbb{N}$ we use $\mathbb{G}_n := \{0, 1, \ldots, n-1\}$ as the set of all natural numbers smaller than n.

To encode PHP_h^p the variable $P_{i,j}$ for $i \in \mathbb{G}_p, j \in \mathbb{G}_h$ is used to describe that pigeon i is in hole j. The trivial SAT encoding consists of two different kinds of clauses:

$$\bigvee_{j \in \mathbb{G}_h} P_{i,j} \qquad \text{for } i \in \mathbb{G}_p \tag{4.1}$$

$$\neg P_{i,j} \vee \neg P_{k,j} \qquad \text{for } i, k \in \mathbb{G}_p : i \neq k; j \in \mathbb{G}_h \tag{4.2}$$

Equation (4.1) states that each pigeon has a hole and (4.2) that there is at most one pigeon in each hole. For these formulas it is known that resolution refutation is hard, i.e. the proof has exponential size [20].

There are more sophisticated approaches for the encoding of the at most one constraint (see [17] for an overview) and those encodings lead to improvements in solving times. If combined with rules to break symmetries, the encoding can be refuted trivially by state-of-the-art SAT solvers [24].

There is a 3-SAT variant of (4.1), i.e. each clause has at most three literals in it. $H_{i,j}$ for $i \in \mathbb{G}_p; j \in \mathbb{G}_{h+1}$ is a helper variable. Note that this encoding is still exponentially hard [2].

$$\neg H_{i,j} \vee P_{i,j} \vee H_{i,j+1} \qquad \text{for } i \in \mathbb{G}_p; j \in \mathbb{G}_h \tag{4.3}$$

$$H_{i,0} \wedge \neg H_{i,h} \qquad \text{for } i \in \mathbb{G}_p \tag{4.4}$$

While the presented formulas are exponentially hard for resolution, there are other proof systems such as extended resolution [46] that allow a polynomial refutation [10].

The rough idea of extended resolution is to add an auxiliary variable a and for two existing literals b, c add the clauses for $a \leftrightarrow b \wedge c$. It is possible to simulate extended resolution by adding the additional clauses manually.

The idea is to add clauses with the extended resolution that allows to reduce the problem of fitting n pigeons into $n-1$ holes to fitting $n-1$ pigeons into $n-2$ holes and so on, until 2 pigeons are tried to be fitted in 1 hole. The latter is obviously unsatisfiable (even for a SAT solver). Therefore, variables $P^n_{i,j}$ for $1 < n \leq p$ are used to encode that pigeon i fits in hole j, in the problem of fitting n pigeons to $n-1$ holes.

The clauses for encoding PHP^p_h with $h = p - 1$ for the simulated extended resolution are as in Equations 4.1 and 4.2, where the variables $P_{i,j}$ are replaced with $P^p_{i,j}$. And for $n \in \mathbb{N} : 1 < n < p$ the following clauses are added to simulate the extended resolution [10]:

$$\begin{array}{rl} P^n_{i,j} \vee \neg P^{n+1}_{i,j} & \neg P^n_{i,j} \vee P^{n+1}_{i,j} \vee P^{n+1}_{i,n-1} \\ P^n_{i,j} \vee \neg P^{n+1}_{i,n-1} \vee \neg P^{n+1}_{n,j} & \neg P^n_{i,j} \vee P^{n+1}_{i,j} \vee P^{n+1}_{n,j} \end{array} \tag{4.5}$$

Note that for $n \in \mathbb{N} : 1 < n \leq p$ it is possible to resolve clauses encoding PHP^n_{n-1} based on the variables $P^n_{i,j}$ in a few steps [10]:

$$\bigvee_{j \in \mathbb{G}_{n-1}} P^n_{i,j} \qquad \text{for } i \in \mathbb{G}_n \tag{4.6}$$

$$\neg P^n_{i,j} \vee \neg P^n_{k,j} \qquad \text{for } i, k \in \mathbb{G}_n : i \neq k; j \in \mathbb{G}_{n-1} \tag{4.7}$$

4.2 Studied Algorithms

4.2.1 Incremental Versions of the Pigeonhole Principle

The goal is to refute PHP^p_h, with $h := p - 1$ so that the formulas are unsatisfiable. Therefore, we fix the number of pigeons p and increase the number of holes h' with each successive call to the solve method until we reach the final number of holes h.

With the standard SAT encoding (4.1), (4.2) it is necessary to remove clauses (4.1) after each call, as they are only valid for the specific number of holes, they were added for. As mentioned earlier, it is possible to remove clauses by using activation literals. This leads to the first incremental implementation:

Algorithm 2 Naive Incremental Encoding
Make PHP incremental by fixing the number of pigeons and increasing the number of holes in each step. Deactivate invalid at-least-one constraints.

```
1: for $h' := 1;\ h' \leq h;\ h' := h' + 1$ do
2:     $add\left(\bigwedge_{i \in \mathbb{G}_p} (\neg a_{h'} \vee \bigvee_{j \in \mathbb{G}_{h'}} P_{i,j})\right)$          ▷ add at-least-one
3:     $add\left(\bigwedge_{i,k \in \mathbb{G}_p : i \neq k} (\neg P_{i,h'-1} \vee \neg P_{k,h'-1})\right)$          ▷ add at-most-one
4:     $solve(assumptions = \{a_{h'}\})$
```

Note that the literals $P_{i,j}$ occur only in clauses that also contain an activation literal. This is problematic, as clauses learned from a conflict in CDCL based SAT solving can be inferred with resolution, but all resolved clauses will contain the activation literal. Therefore, all learned clause are not useful in later steps as the activation literal is not assumed and they can be satisfied by setting the activation literal to true. It is still possible to get a benefit from incremental SAT solving: Usually, other information, beside learned clauses, is kept in successive incremental calls such as variable activity.

It is possible to overcome the described limitation by introducing helper variables $H_{i,j}$, that allow to extend the critical clause (4.1). $H_{i,j}$ encodes that pigeon i is in hole j or later, i.e. $H_{i,j} \rightarrow \exists j' \geq j : P_{i,j'}$. Note that this approach results in the 3SAT encoding of PHP^p_h.

Algorithm 3 Incremental 3SAT Encoding
Make PHP incremental by fixing the number of pigeons and increasing the number of holes using the 3SAT encoding to extend the at-least-one constraint.

```
1: $add\left(\bigwedge_{i \in \mathbb{G}_p} (H_{i,0} \wedge \neg H_{i,h})\right)$    ▷ upper and lower bound
2: for $h' := 1;\ h' \leq h;\ h' := h' + 1$ do
3:     $add\left(\bigwedge_{i \in \mathbb{G}_p} (\neg H_{i,h'-1} \vee P_{i,h'-1} \vee H_{i,h'})\right)$    ▷ extend at-least-one
4:     $add\left(\bigwedge_{i,k \in \mathbb{G}_p : i \neq k} (\neg P_{i,h'-1} \vee \neg P_{k,h'-1})\right)$    ▷ add at-most-one
5:     $solve(assumptions = \{\neg H_{i,h'} \mid i \in \mathbb{G}_p\})$
```

4.2.2 Guiding Refutation with Incremental SAT Solving

An intriguing fact is that adding the Equations in 4.5 simulating extended resolution is not helpful for SAT solvers [32]. However, for any n it should be trivial to resolve the clauses for PHP^n_{n-1} based on variables $P^n_{i,j}$ (Equations 4.6, 4.7) if the clauses for PHP^{n+1}_n based on $P^{n+1}_{i,j}$ have been learned already.

Algorithm 4 Extended Resolution Incrementally Guided
Make a solve step for each clause that is to be derived during extended resolution.

```
   ▷ Add the pigeonhole formulas and the clauses from extended resolution:
1: for Clause $C$ in all Equations from 4.2 to 4.5 do
2:     $add(C)$
   ▷ Guide the proof search using the incremental SAT solver:
3: for $n := p - 1;\ n > 1;\ n := n - 1$ do    ▷ increase proof depth
4:     for Clause $C$ in Equations 4.6 and 4.7 do
5:         $solve(assumptions = \neg C)$
6: $solve(assumptions = \emptyset)$
```

The idea of this algorithm is to assume the negation of a clause to be learned. Note that the negation of a clause is a conjunction of literals, which can be interpreted as a set of assumptions. Clauses learned during this process may be helpful in later steps.

As we will see later, this is working astonishingly well. However, it has the clear disadvantage that we have to know beforehand how the refutation should look like. This is fine for a well studied problem such as the pigeonhole problem but for other applications of incremental SAT solving it is not applicable.

Instead, a more general strategy called exhaustive incremental SAT solving is proposed: After the formula is shown unsatisfiable for a set of assumptions, every subset of the assumptions is tested for unsatisfiability. The idea is that this approach leads to learned clauses, which are more restrictive and therefore more helpful in later solve steps. In case of the pigeonhole principle, we already know that the formula is only unsatisfiable if the number of holes is smaller than the number of pigeons p, i.e. the current number of holes h' plus the number of unrestricted helper variables $H_{i,h'}$ needs to be smaller than the number of pigeons ($p - n + h' < p \Leftrightarrow h' < n$, where n is the number of restricted helper variables and $p - n$ the number of unrestricted helper variables).

Algorithm 5 Exhaustive Incremental 3SAT Encoding
Make a solve step for every subset of assumptions from Algorithm 3.

```
1: procedure MULTISOLVE(p, h')
2:     for $n := p$; $n > h'$; $n := n - 1$ do
3:                         ▷ note that the order is relevant as smaller $n$ imply larger ones
4:         for from $\{\neg H_{i,h'} \mid i \in \mathbb{G}_p\}$ choose n into $L$ do
5:             $solve(assumptions = L)$
```

The complete algorithm is the same as Algorithm 3, where the call to $solve(\cdot)$ is replaced with a call to MULTISOLVE($\cdot$). Note that although not every subset of assumptions is tested, this results in an exponential number of calls to solve.

4.3 Evaluation

For the evaluation the incremental approaches described in the previous section are compared to solving the different encodings of PHP^p_h with $h = p - 1$ directly. The different algorithms have been implemented and the implementation along with the experimental results are available at GitHub[1]. 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 900 seconds and resource limitation to 1 CPU core and 8 GB of memory. All times are wall clock time. The incremental version of glucose [3, 4] (a minisat [14] based solver) is used for the experiments — in the version submitted to the incremental track of the SAT competition 2016.

It turned out, that the solving time is very depended on clause and variable order. To counter this effect the calls to the SAT solver are randomized as described in Section 6.2.

[1]`https://github.com/StephanGocht/incphp`

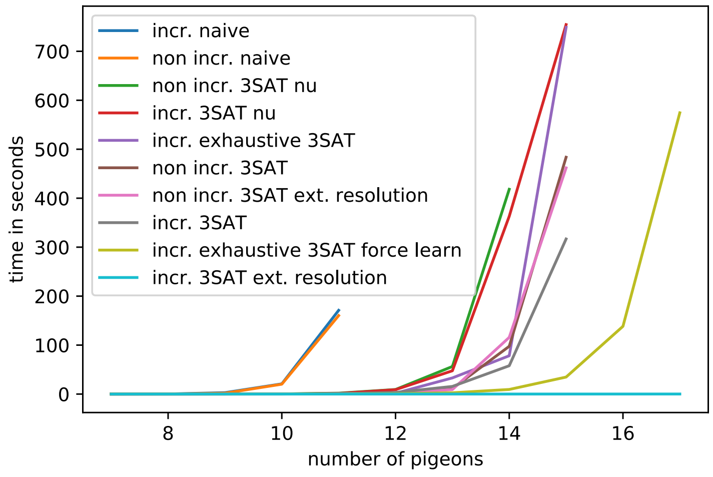
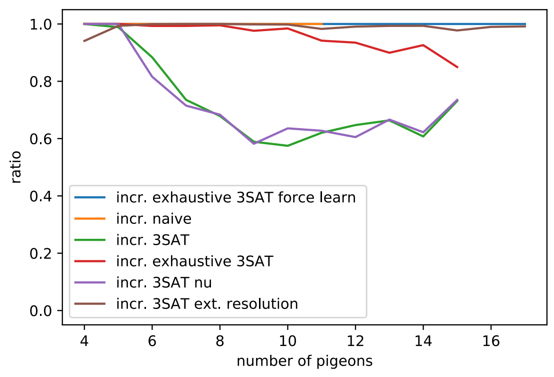


Figure 4.1: Runtime average for solving PHP^p_{p-1} with different encodings. The number of pigeons p is on the x-axis.

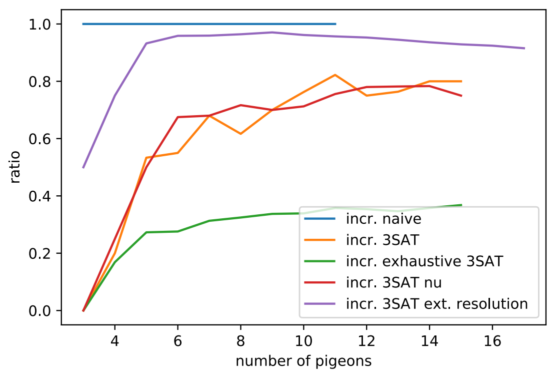
Every experiment was run 10 times and the plots show average values. Note that the standard deviation is rather high, for example for the runtime of the non incremental encoding it is as follows: 15 pigeons $\approx$ 141, 14 pigeons $\approx$ 88 and for 13 pigeons $\approx$ 4. This should be kept in mind, when interpreting the plots as they only provide a rough indication on the performance.

Nonetheless, we can make the following observations from Figure 4.1: The standard encoding is not very beneficial compared to the 3SAT encoding, no matter if it is used incrementally or not. The reason is that the helper variables lead to smaller clauses, which seem to be better for the SAT solver. Due to the exponential growth of the solving time the overhead for the naive incremental encoding is negligible. In Figure 4.3a we can see that the time needed for subsequent steps of the algorithm is increasing monotonously after an initial phase. This is a common behavior for other applications of incremental SAT solving as well, as the search space is usually increasing with an increasing number of incremental steps. In Figure 4.2a we can see, what was already stated earlier: Every learned clause contains the negation of the assumed literal. In Figure 4.2b we can see that the negation of the assumed literal is always learned as a clause, i.e. after calling $solver(a_{h'})$ there is a learned clause $\neg a_{h'}$.

For the incremental 3SAT encoding, let us have a look at Figure 4.3c first. In contrast to the behavior of the naive incremental encoding, the time needed for the last solve step is decreasing, when we reach a higher number of holes. This is uncommon, and an investigation of the phenomena showed, that the reason is that the final assumptions are actually added as clauses. If we remove the upper bound clauses of the form $\neg H_{i,h}$, we get a similar behavior as for the naive incremental approach as shown in Figure 4.3d. To investigate this further, consider the modified version of non-incremental 3SAT, which does not add the upper bound permanently, but assumes them before solving the instance. In Figure 4.1 these variants are suffixed with nu. The hit on runtime is significant and we can conclude, that for the tested solver it is beneficial to have unit clauses instead of assumptions. Overall, the performance of the incremental 3SAT encoding seems to be



(a) Ratio of learned clauses containing the negation of an assumed literal to the number of learned clauses.



(b) Ratio of calls to solve, where the negation of assumed literals was learned as a clause to the number of calls to solve.

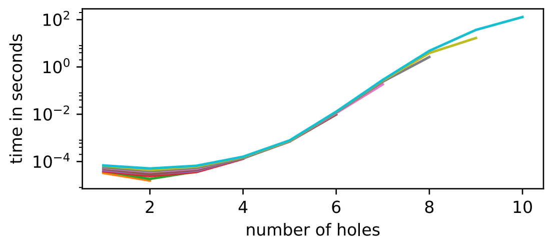
Figure 4.2: Analysis of learned clauses.

better than the non-incremental 3SAT encoding, although it is performing more work, as it is not only showing that n pigeons do not fit in $n-1$ holes, but also that they do not fit in a number of holes less than $n-1$.

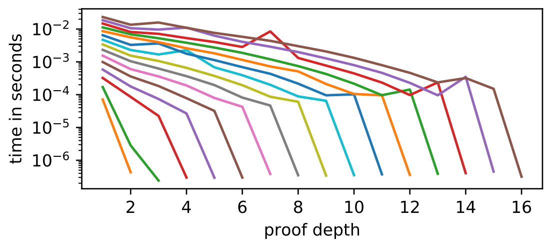
As already shown in earlier work, adding the clauses from extended resolution does not bring any benefit on its own. However, the incremental algorithm guides the SAT solver very well, so that it is able to refute the formula in no time (Figure 4.1). Even for 30 pigeons the refutation is taking less than two seconds. Figure 4.2b shows that most of the assumed clauses are actually learned and Figure 4.3b shows that the reduction to fewer pigeons and fewer holes becomes easier with increasing proof depth. This is not surprising as the number of clauses to infer decreases. Overall this shows that the extended resolution proof is followed, which explains the good performance, as the extended resolution proof is sub-exponential.

Finally, we have the exhaustive incremental 3SAT encoding. Its performance tends to be worse than the non-incremental 3SAT encoding (Figure 4.1). A view to Figure 4.2b shows, that in contrast to the incrementally guided extended resolution only a third of the assumptions are learned as a clause. However, the solver is intended to learn them all. To change this behavior it is possible to manually add the negation of an assumption as a clause after it is shown unsatisfiable, i.e. to force the learning of the clause.

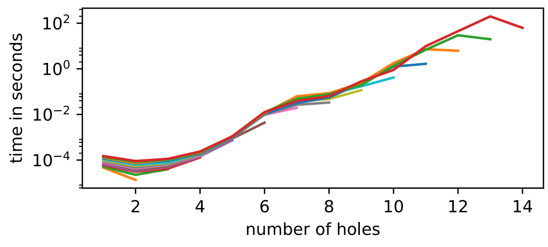
This approach outperforms the other algorithms, except the incremental extended resolution (Figure 4.1). An interesting property of this approach is that the time spent on solving an increasing number of holes is no longer growing monotonically but rather reaches a high, somewhere in the middle. The explanation is that the number of performed solve steps is largest if the number of holes is half of the number of pigeons. For the exhaustive incremental version without forced learning this behavior exists for small number of pigeons as well but is clearly disturbed as the number of missing clauses increases with higher number of pigeons (See Figure 4.3e and 4.3f).



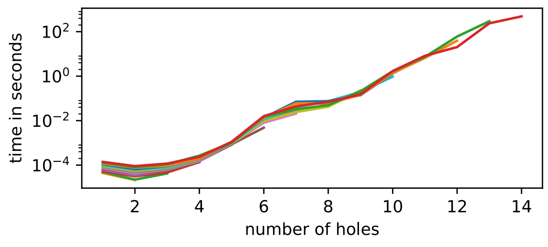
(a) Naive Incremental Encoding



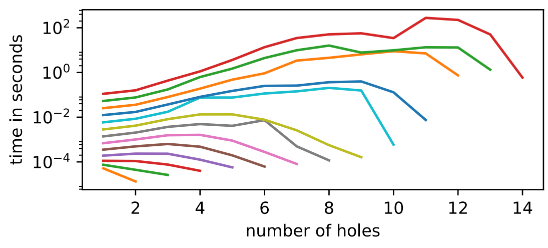
(b) Extended Resolution Incrementally Guided



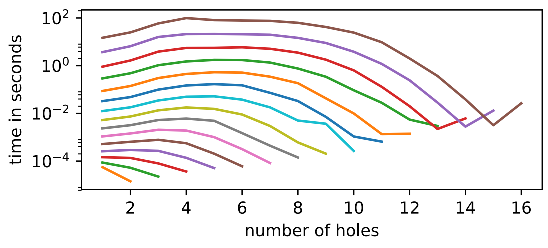
(c) Incremental 3SAT Encoding



(d) Incremental 3SAT Encoding without Upper Bound



(e) Exhaustive Incremental 3SAT Encoding



(f) Exhaustive Incremental 3SAT Encoding with Forced Learning

Figure 4.3: This figure shows the time needed for all solve steps per iteration of the outer loop of the listed algorithm. Each color in a plot shows the refutation of PHP^n_{n-1} for different n.

4.4 Conclusion

In this section we studied different algorithms using incremental SAT solving for the refutation of pigeonhole formulas. The results are:

- It is better to use unit clauses instead of assumptions, when applicable. This might be a solver specific issue as there are approaches to overcome this handicap with additional book keeping [29].
- Extending the formula incrementally can be faster than solving the formula directly.
- The negation of assumed literals are often learned as clauses and allow to guide the solver for the solution of a problem, if written down properly this is not surprising: if solving formula F with assumptions $\neg C$ is unsatisfiable $F \wedge \neg C \models \bot$ then $F \models C$.

- It is possible to exhaust the search space at earlier steps, which leads to a significant benefit for solving later steps. This might be transferable to planning but naive approaches have not been successful and are not contained within this thesis.

5 Incremental Planning Encodings

5.1 Single Ended Incremental Encoding

With non-incremental SAT solving the plan encodings are newly generated for each makespan 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 literal to the goal clauses. This allows activating only the latest goal clause and extend the formula by one transition in each step.

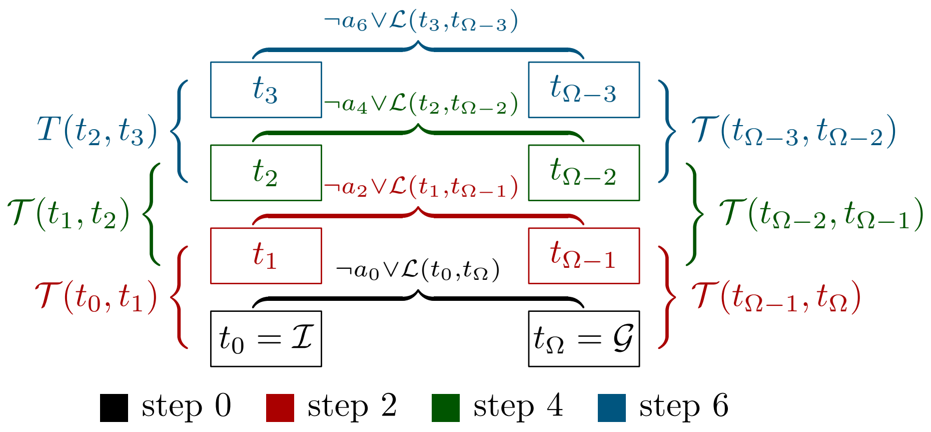
```
$step(0):$
    $add\left(\mathcal{I}(0) \land [\neg a_0 \lor \mathcal{G}'(0)]\right)$
    $solve(assumptions = \{a_0\})$
$step(k):$
    $add\left(\mathcal{T}'(k-1,k) \land [\neg a_k \lor \mathcal{G}'(k)]\right)$
    $solve(assumptions = \{a_k\})$
```

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.

5.2 Double Ended Incremental Encoding

To learn clauses from the goal state as well, 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: $\mathcal{L}(j,k) := \land_{l=1}^{N} v_l@j \Leftrightarrow v_l@k$.



$\neg a_6 \vee \mathcal{L}(t_3, t_{\Omega-3})$
t_3
$\neg a_4 \vee \mathcal{L}(t_2, t_{\Omega-2})$
$t_{\Omega-3}$
$\mathcal{T}(t_2, t_3)$
$\mathcal{T}(t_{\Omega-3}, t_{\Omega-2})$
t_2
$\neg a_2 \vee \mathcal{L}(t_1, t_{\Omega-1})$
$t_{\Omega-2}$
$\mathcal{T}(t_1, t_2)$
$\mathcal{T}(t_{\Omega-2}, t_{\Omega-1})$
t_1
$\neg a_0 \vee \mathcal{L}(t_0, t_\Omega)$
$t_{\Omega-1}$
$\mathcal{T}(t_0, t_1)$
$\mathcal{T}(t_{\Omega-1}, t_\Omega)$
$t_0 = \mathcal{I}$
$t_\Omega = \mathcal{G}$
step 0
step 2
step 4
step 6

Figure 5.1: Visualization of the double ended incremental encoding. For clear arrangement, link clauses are only shown for every other step. The colors show which clauses are new in the particular step. All clauses from previous steps are present as well.

Activation literals ensure that only the latest link is active.

```
step(0):
    add($\mathcal{I}(0) \wedge \mathcal{G}'(\Omega) \wedge [\neg a_0 \vee \mathcal{L}(0, \Omega)]$)
    solve(assumptions = $\{a_0\}$)
step(2k + 1):                                         add $t_{k+1}$
    add($\mathcal{T}'(k, k+1) \wedge [\neg a_{2k+1} \vee \mathcal{L}(k+1, \Omega - k)]$)
    solve(assumptions = $\{a_{2k+1}\}$)
step(2k):                                             add $t_{\Omega-k}$
    add($\mathcal{T}'(\Omega - k, \Omega - k + 1) \wedge [\neg a_{2k} \vee \mathcal{L}(k, \Omega - k)]$)
    solve(assumptions = $\{a_{2k}\}$)
```

Note that Ω is neither a precomputed number nor a fixed upper bound but a symbol which always represents the last time point and $\Omega - k$ 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 Ω. 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. This is visualized in Figure 5.1.

With this encoding the solver is able to learn new clauses based on both, 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 state and goal state instead of adding a new goal state.

5.3 Lemma Transformation

Instead of only reusing learned clauses from previous solve steps it is even possible to go one step further: If a learned clause is only based on transition clauses and not on initial or goal clauses than this clause can be transformed to other time points as well, if they have the same transition clauses. This idea of transforming learned clauses to newly added time points was already presented for SAT based bounded model checking with non-incremental SAT solving [43].

Definition 2. *For a natural number Δt, a clause C, a formula F and time points $t_0, t_1, \dots$ transform is defined as:*

$$\begin{aligned} \mathit{transform}(C, \Delta t) &:= \{x@(t_{i+\Delta t}) \mid x@t_i \in C\} \\ \mathit{transform}(F, \Delta t) &:= \{\mathit{transform}(C, \Delta t) \mid C \in F\} \end{aligned}$$

For example $\mathit{transform}(\mathcal{T}'(t_1, t_2), 1) = \mathcal{T}'(t_2, t_3)$. Adding these transformed clauses is sound in the following way:

Theorem 1.

$$\begin{aligned} &\bigwedge_{k \in \mathbb{G}_n} \mathcal{T}'(t_k, t_{k+1}) \models C \\ \Rightarrow &\bigwedge_{k \in \mathbb{G}_n} \mathcal{T}'(t_{k+l}, t_{k+1+l}) \models \mathit{transform}(C, l) \end{aligned}$$

Proof. Assume $\bigwedge_{k \in \mathbb{G}_n} \mathcal{T}'(t_k, t_{k+1}) \models C$ then it holds $\bigwedge_{k \in \mathbb{G}_n} \mathcal{T}'(t_k, t_{k+1}) \land \neg C \models \bot$ and there is a resolution refutation with transcript R. We construct R' by applying $\mathit{transform}(C, l)$ to every clause C in the transcript R. R' is now a transcript of a resolution refutation for $\bigwedge_{k \in \mathbb{G}_n} \mathcal{T}'(t_{k+l}, t_{k+1+l}) \land \neg \mathit{transform}(C, l) \models \bot$ and therefore $\bigwedge_{k \in \mathbb{G}_n} \mathcal{T}'(t_{k+l}, t_{k+1+l}) \land \models \mathit{transform}(C, l)$. □

This can be used to construct the following algorithm, assuming that the initial and goal clauses are only unit clauses.

```
$step(0):$
    $solve(assumptions = \mathcal{I}(t_0) \cup \mathcal{G}'(t_0))$
$step(k):$                                              add $t_k$
    $add(\mathcal{T}'(t_{k-1}, t_k))$
    for $i < k; C \in learned(i):$
        $\Delta t := k - i$
        $add(transform(C, \Delta t))$
    $solve(assumptions = \mathcal{I}(t_0) \cup \mathcal{G}'(t_k))$
```

This encoding is basically the single ended encoding with two differences: The initial clauses are not added but only assumed, and we have additional clauses based on the learned clauses of the previous steps. It is sound to add these transformed clauses because of Theorem 1.

If the initial or goal clauses are not unit clauses than they can be added with activation literals in which case it is safe to transform clauses that do not contain the activation literal by Lemma 2. A similar technique can be used for the double ended incremental encoding.

6 Evaluation

6.1 Benchmark Environment

All benchmark problems from the agile track of the 2014 International Planning Competition (IPC) [47] are used for evaluation: 280 problems are divided into 14 domains with 20 problems each. These benchmarks are chosen because it is the latest planning competition. The agile track is selected because the SAT based planner Madagascar participated in this track and is one of the most relevant competitors for our approach in SAT based planning. Two variants of Madagascar scored second and third place in this competition. The benchmark problems are provided in form of Planning Domain Definition Language (PDDL) files. PDDL is a standardized description for planning problems.

Different SAT based planners can be used to transform the planning problems into the SAT encoding. The used SAT encodings of Madagascar [40] ($\forall$, $\exists$) and Freelunch [6, 7] ($\exists^2$, reinforced) are represented in terms of initial clauses $\mathcal{I}$, transition clauses $\mathcal{T}$, universal clauses $\mathcal{U}$ and goal clauses $\mathcal{G}$ as described in Section 2.3.2.

The tool IncPlan, which was developed for this thesis, is used to solve the problems with the different encodings. To do so it relies on an arbitrary incremental SAT solver that supports the Re-entrant Incremental Satisfiability Application Program Interface (IPASIR). This interface was introduced in the 2015 SAT Race [8]. For the non-incremental encoding the provided feature to reset the SAT solver is used. The paper presenting the results of this thesis [18] used *COMiniSatPS 2Sun nopre* [30] as solver as it was the solver performing best on the planing problems, from solvers competing in the Incremental Library Track of the 2016 SAT Competition. However, to get deeper insights it was necessary to modify the solver, which was easier for glucose [3, 4] (a minisat [14] based solver) which is used for the experiments presented in this section.

Finally, the SAT based planner used for the encoding to SAT is used to construct the plan from the satisfying assignment of the SAT solver. A graphical overview of this system is shown in Figure 6.1.

The systems used to perform the evaluation have two Intel® Xeon® X5355 CPUs (4 cores with 2.66GHz) and 24GB 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. At most two instances run on each machine, each on its own CPU.

The rest of this chapter contains several scatter plots to compare different variants. They will all have a common structure: They 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 within the time limit 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. For easier reading of

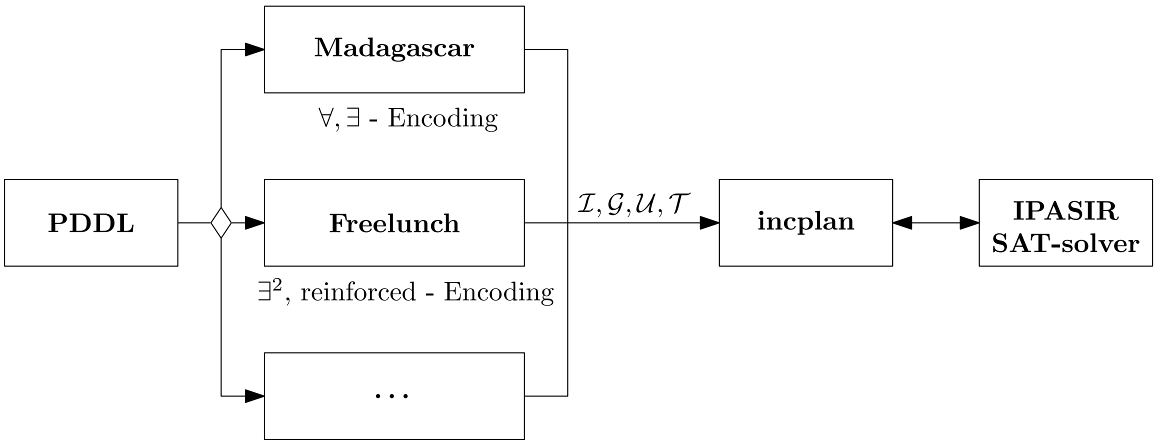

Madagascar
∀, ∃ - Encoding
PDDL
Freelunch
$\exists^2$, reinforced - Encoding
...
$\mathcal{I},\mathcal{G},\mathcal{U},\mathcal{T}$
incplan
IPASIR
SAT-solver

Figure 6.1: System Overview

the logarithmic scale there are multiple diagonal lines, indicating that one approach is n times faster than the other.

6.2 Randomizing Input through the Incremental SAT Interface

The order of clauses and naming of variables in a CNF can have an effect on the performance of the SAT solver, also the problem does not change when clauses are reorderd or variables are renamed. The evaluation takes place on very similar formulas and therefore it is important to counteract the effects of clause ordering and variable renaming.

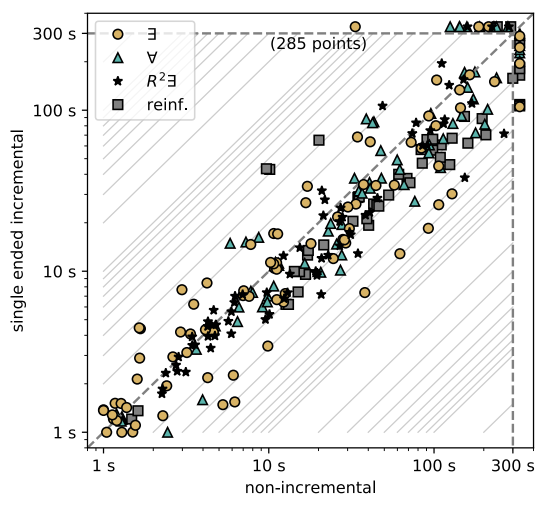
A common technique is to randomize the order of clauses, the order of literals within the clauses and the naming of the variables in a CNF. This technique is transferable to incremental SAT solving but it is important to keep in mind that clauses are added for a specific call to *solve*, for example if the calls to the interface are $add(C_1)$, $add(C_2)$, $solve(A_1)$, $add(C_3)$, $solve(A_2)$ then it is wrong to change the order such that C_1, C_2 are added after $solve(A_1)$ or C_3 is added before. On the other hand changing the order in which C_1, C_2 are added is perfectly fine. Additionally, it is important to keep track of renamed variables as they might be used later on in added clauses or assumptions. This leads the decoration of the incremental SAT interface described in Algorithm 6, which is used throughout the experiments.

6.3 Speedup due to Incremental SAT Solving

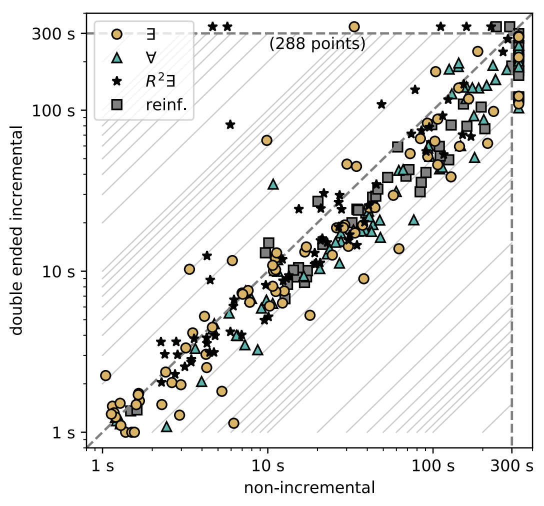
The main hypothesis behind this thesis is that incremental SAT solving is beneficial when applied to SAT based planning, where a series of similar formulas has to be solved. The evaluation clearly supports this hypothesis: Figure 6.2 shows that the incremental approaches are able to solve the given problems faster than the non-incremental approach. Additionally, the incremental approaches are able to solve more problems as can also be seen in Table 6.1. This is especially true for the double ended incremental approach.

Algorithm 6 Randomized Incremental Interface.

```
1: $F_{\text{new}} \leftarrow \emptyset$, set of clauses added since last call to *solve*
2: $V \leftarrow \emptyset$, set of all known variables
3: $r : V \longrightarrow \{x_1, x_2, \ldots, x_{|V|}\}$, one to one mapping representing the variable renaming

4: **procedure** $add_{rnd}$(C)
5:     $F_{\text{new}} \leftarrow F_{\text{new}} \cup C$

6: **procedure** $solve_{rnd}$(A)
7:     $V_{\text{new}} \leftarrow [vars(A) \cup vars(F_{\text{new}})] \setminus V$
8:     $r_{\text{new}} : V_{\text{new}} \longrightarrow \{x_{|V|+1}, x_{|V|+2}, \ldots, x_{|V|+|V_{\text{new}}|}\}$ is a random one to one mapping
9:
10:    $r \leftarrow r \cup r_{\text{new}}$
11:    $V \leftarrow V \cup V_{\text{new}}$
12:    $F_{\text{new}} \leftarrow r(F_{\text{new}})$
13:
14:    **for** $C \in F_{\text{new}}$ in random order **do**
15:        $add(C)$, where literals in $C$ are randomly ordered
16:    result $\leftarrow solve(assumptions = r(A))$
17:    $F_{\text{new}} \leftarrow \emptyset$
18:    **return** result

19: **procedure** $learned_{rnd}$(i)
20:    **return** $r^{-1}(learned(i))$
```



(a) Single Ended Incremental vs. Non-Incremental



(b) Double Ended Incremental vs. Non-Incremental

Figure 6.2: Comparison of Incremental and Non-Incremental Solving

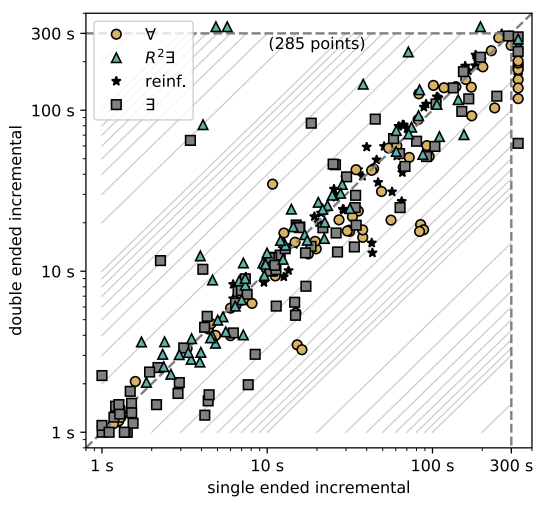
The benefit of incremental SAT solving in terms of runtime (Figure 6.2) is mostly independent from the used encoding, although the reinforced and the forall encoding seem to profit more from the incremental encodings than the exist and relaxed-relaxed-exist encoding. The same holds regarding the number of solved instances (Table 6.1): ∀ + 8, reinforced +7, ∃ + 4, $R^2∃ - 5$ instances solved.

Comparing the benefits in terms of runtime of the two incremental encodings in Figure 6.2a and 6.2b it seems that the double ended incremental encoding is even more beneficial than the single ended incremental encoding. The direct comparison in Figure 6.3a confirms this observation.

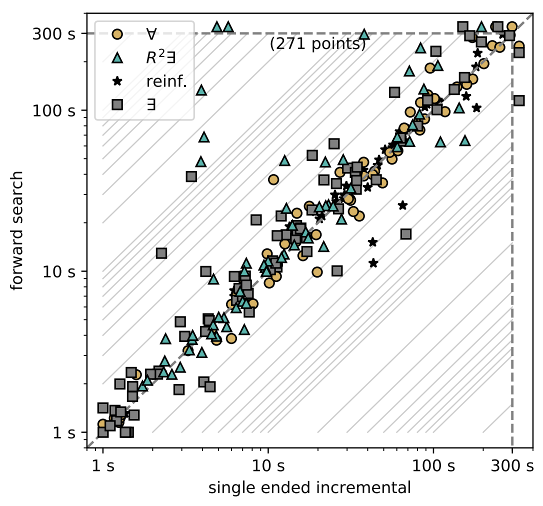
Note that there are 279 problems solved by both approaches, thereof 136 are solved faster by the single ended version. However, remember that the axis are logarithmic: Only 28 problems have a speedup of more than 1.3 compared to the double ended encoding. On the other hand the double ended approach has 74 problems that have a speedup of more than 1.3 compared to the single ended encoding. In other words: using the double ended encoding has a small overhead on some problems but leads to large performance gains in others.

A simple explanation for the overhead is that the chosen encoding with the link clauses causes the overhead. Consider the following to evaluate this: Use the double ended encoding but instead of adding transition clauses alternating to both sides, the transition clauses are only added to the side with the initial clauses. This variant will be called forward search, the reasons for the naming will be more clear in the next chapter.

The single ended incremental approach and forward search are conceptually identical and are only different in the way the goal clause is added: In the single ended incremental approach the goal clauses are directly assumed, while forward search assumes the goal clauses indirectly through the link clauses.



(a) Single vs. Double Ended Incremental

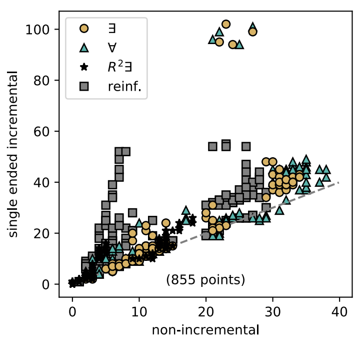


(b) Single Ended Incremental vs. Forward Search

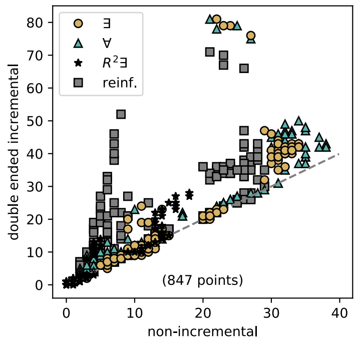
Figure 6.3: Comparison of Incremental Variants

The results of comparing the single ended approach to forward search can be found in Figure 6.3b. They show that this small difference in the encoding leads to quite some variance regarding the run time with a tendency in favor of the single ended approach. This result justifies the notion of an overhead during the comparison between the single and double ended encodings.

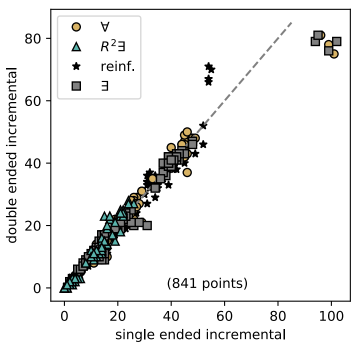
Figure 6.4 contains a comparison of the approaches for unsolved instances, but compares the largest makespan shown to be unsatisfiable instead of comparing the runtime. This sows that much larger makespans can be proven unsatisfiable with the incremental approaches. However, the double ended incremental encoding is not able to prove significant larger makespans than the single ended incremental encoding.



(a) Non-Incremental vs. Single Ended Incremental

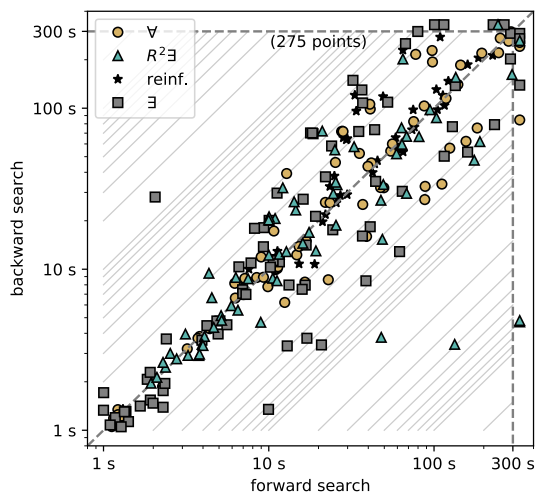


(b) Non-Incremental vs. Double Ended Incremental

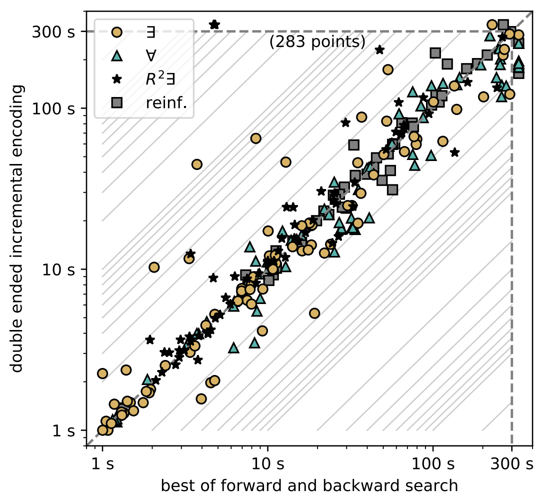


(c) Single Ended Incremental vs. Double Ended Incremental

Figure 6.4: Comparison on Reached Makespan for Unsolved Instances



(a) Forward vs. Backward Search



(b) Comparing the double ended incremental encoding (bidirectional search) to the best of forward and backward search.

Figure 6.5: Comparison of Forward, Backward and Bidirectional Search

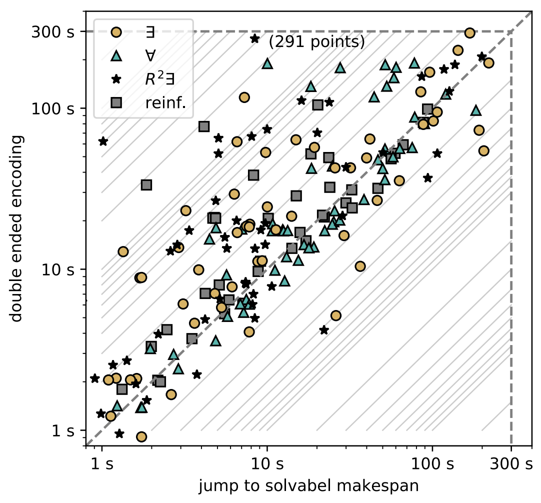
6.4 An Alternative View

The motivation for the double ended encoding was that clauses and other information derived from the goal clauses can not be carried over between solve steps. Another explanation for the performance gains compared to the single ended incremental approach can be made when viewing the problem from the perspective of classical, explicit state exploration search [34]: In a forward search the reachable states are explored starting from the initial state until the goal state is reached. Similarly, it is possible to do a backward search, that starts from the goal state and explores states that can reach the goal state until the initial state is reached. It is also possible to start from both sides and exploring both directions simultaneously. The search is complete when a state is found that is reachable from the initial state and from which the goal state is reachable. This approach is called bidirectional search. The advantage of bidirectional search in explicit state exploration is that it can lead to significant fewer states to be considered.

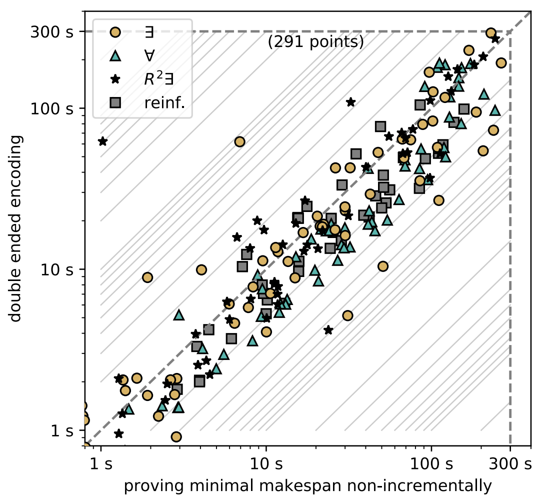
These concepts can be transferred to incremental SAT based planning with the double ended encoding: It is a forward search when transitions are only added to the side with initial clauses as the search is extended from the start. The other way round, when transitions are only added to the side with the goal clauses it is a backward search. And finally, adding transitions alternating to both sides as proposed is a bidirectional search.

Note that the notion of directional search is only reasonable if there is a basis to start from, a direction and especially information carried over between steps. Therefore, it can not be applied to non-incremental SAT based planning.

Figure 6.5a compares SAT based forward and backward search implemented in the double ended incremental encoding as described above. While the direction does not matter for most problems there are clearly problems that benefit from forward or from



(a) Comparing the double ended incremental encoding to only solving the first satisfiable makespan.



(b) Comparing the double ended incremental encoding to non-incrementally solving the last unsatisfiable and the first satisfiable makespan.

Figure 6.6: The figures show the potential of scheduling makespans. Only problems solved by both approaches are considered and only the time spent within the SAT solver is considered, especially not the encoding time.

backward search. However, it is usually not known beforehand which direction is more beneficial.

Nonetheless, this view raises the question whether the advantage of the double over the single ended encoding is only due to the fact that some problems are better solved in backward direction. After all, the double ended encoding is half backward search. Figure 6.5b compares the double ended incremental encoding to forward search or backward search, depending on which is faster. This comparison shows only a slight tendency in favor of the double ended encoding, which means that the advantage of the double over the single ended encoding is partially due to the fact that some problems are better solved in backward direction.

6.5 Potential of Scheduling Makespans

An important technique in SAT based planning is scheduling makespans as described in Section 2.3.2. To evaluate the potential of this technique within incremental SAT solving the following two questions are considered for non-incremental solving:

1. How long does it take to generate a solution, when the makespan is known?
2. How long does it take at least to prove the found makespan is minimal? To prove makespan i minimal it is necessary to show that F_i is solvable and that F_{i-1} is unsolvable. Note that the minimal makespan is already known for this experiment.

Figure 6.6 compares these times with the double ended incremental encoding, which infers the minimal makespan itself. Note that for the double ended incremental encoding the time for solving all makespans is considered while the time answering the questions does only consider the time for solving one, respectively two, makespans non-incrementally.

It is often much easier to find a solution if F_i is solvable, then showing previous formulas unsatisfiable. This is the main motivation behind scheduling makespans and is reflected in Figure 6.6a. Interestingly it is not always true: with the incremental approach it is sometimes faster to solve F_i for all makespans up to i than just solving F_i for the first solvable. Therefore, scheduling makespans will not always be beneficial when using the double ended incremental approach.

If we are interested in finding the minimal makespan as in Figure 6.6b then it is advisable to use the double ended incremental encoding without scheduling makespans, as this is faster than just solving the satisfiable and the last unsatisfiable instance. This also shows that it is easier to solve the formula F_i if previous instances have already been solved. The same effect was observable for incrementally solving the pigeonhole formulas in section 4.

6.6 Importance of Reused Information

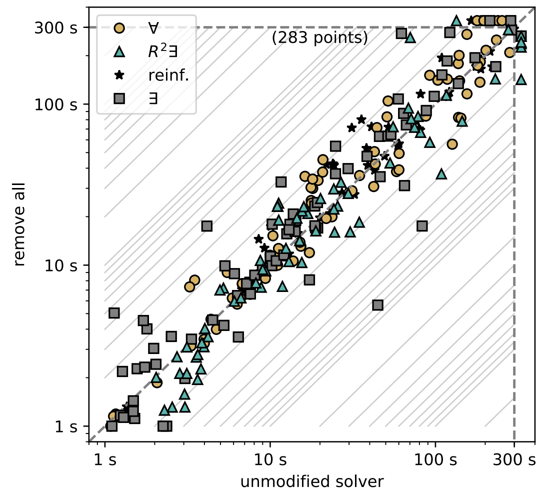
To evaluate which information is important to reuse in incremental planning the SAT solver has been modified to perform one or all of the following actions after each call to solve:

- remove all learned clauses
- reset variable activity
- reset phase

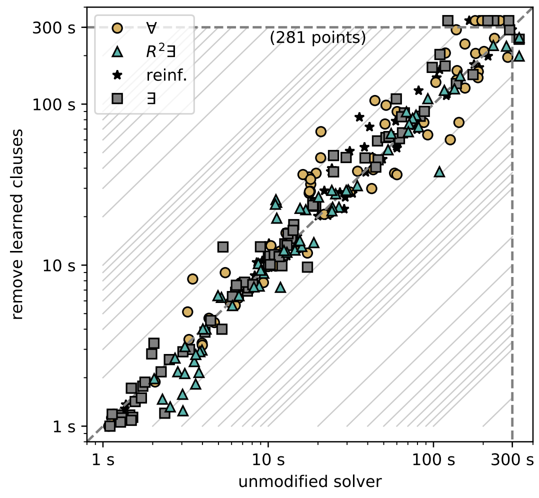
As can be seen in Figure 6.7a removing all the described information leads to a loose of performance for a lot of the problems, also some problems are solved faster. The reason for the outliers can be found in Figure 6.7d: resetting the variable activity introduces a lot of variance in the results, although there is no clear tendency. A similar result can be found for removing the phase in Figure 6.7c, but the effects are much smaller. Removing the learned clauses on the other hand is clearly disadvantageous (Figure 6.7b). However, the learned clauses are not important for all problems and removing them leads to small improvements in that case. This can be explained by the fact that overall fewer clauses need to be considered by the solver.

A very important result is that learned clauses are not the only reason for improved performance when using an incremental SAT solver: The effect of removing the learned clauses is not large enough. This can also be seen in Figure 6.8, where the non-incremental approach is compared with the double ended incremental approach with removing learned clauses.

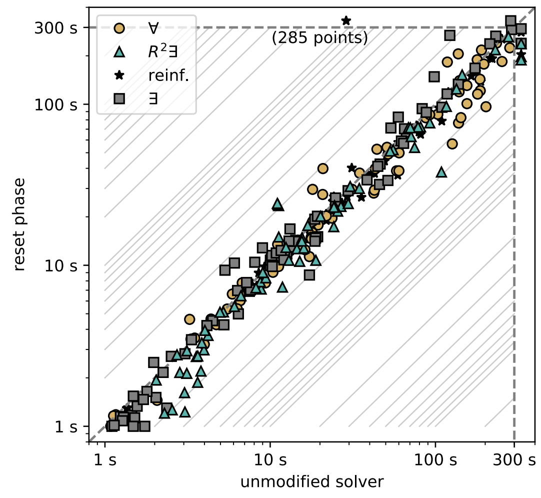
As phase saving and variable activity do not have a clear effect and therefore are ruled out as reasons for the speed up, it is left for future work to analyze the effect of other information reused by the solver. Possible reasons among others are watched literals and time saved by avoiding the input overhead of adding the same formula over and over again.



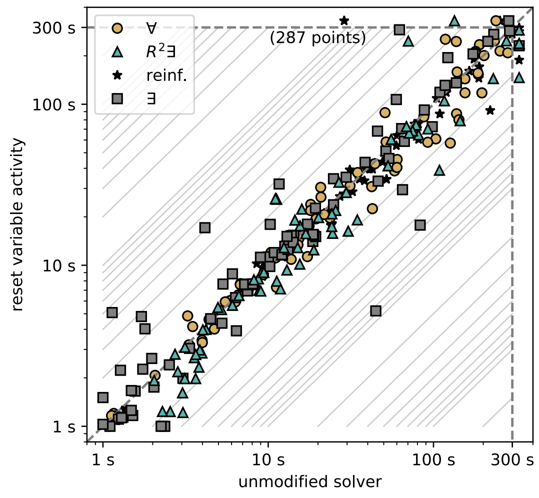
(a) Removing Learned Clauses, Resetting Phase and Variable Activity



(b) Removing Learned Clauses



(c) Resetting Phase



(d) Resetting Variable Activity

Figure 6.7: The figures show the effect of not reusing certain information when applying the double ended incremental encoding.

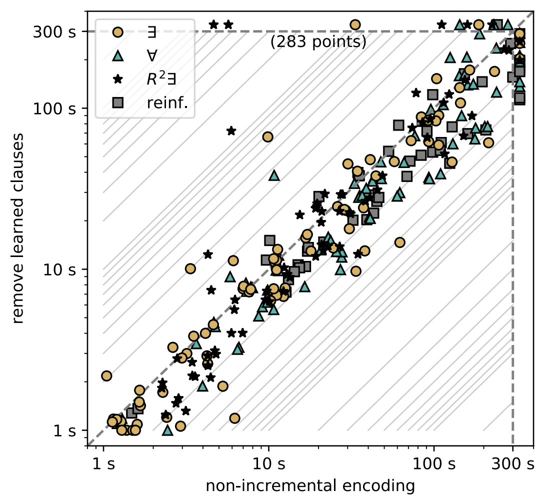


Figure 6.8: Double Ended Incremental vs. Non-Incremental with Removed Learned Clauses

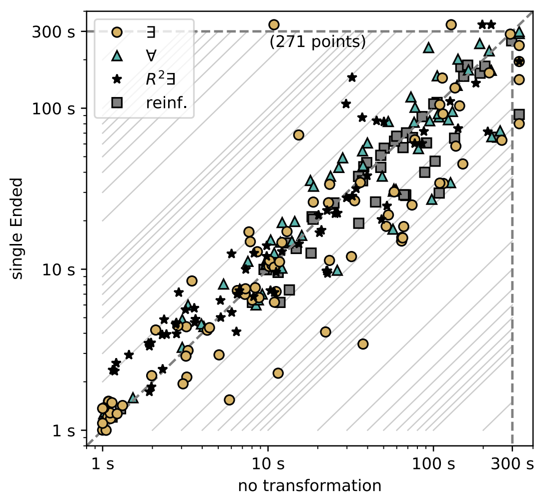


Figure 6.9: Overhead of assuming the initial clauses of the single ended encoding.

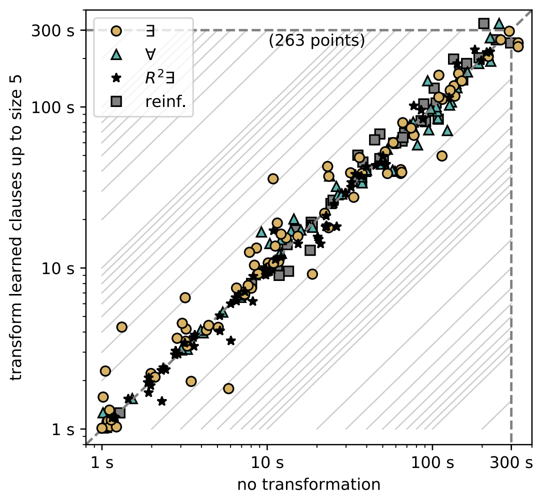
6.7 Lemma Transformation

This section evaluates the performance of transforming learned clauses to appended time points as described in section 5.3. The described algorithm requires that initial and goal clauses are all unit and only assumed. All problems do have the initial and goal clauses as unit clauses. However, as we have already seen in Section 4 for the pigeonhole formulas assuming unit clauses instead of adding them has a noticeable impact on the runtime. To suppress this effect the results for transforming learned clauses will be compared to a variant of the single ended encoding, where the initial clauses are only assumed. This variant is called *no transformation*. As for the pigeonhole formulas, this results in an increased solve time as can be seen in Figure 6.9.

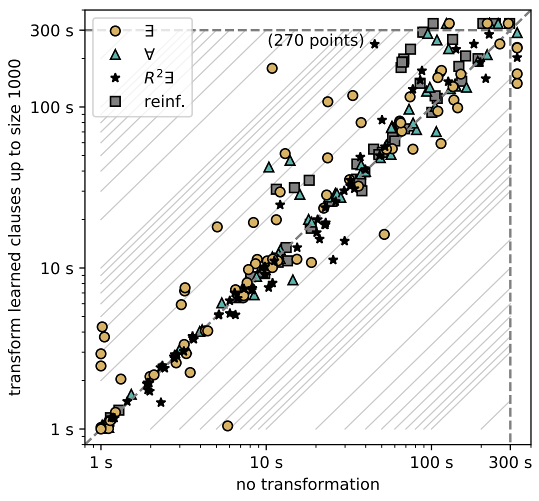
The size and number of learned clauses, which are transformed, may influence the results. Therefore, different limits have been tested: up to 5, 50 or 1000 literals per clause. Clauses, which are larger than the limit, have not been transformed.

There are some problems which benefit from transforming clauses with up to 50 literals (Figure 6.10c). Actually, most of these problems lay in only two domains: Thoughtful and CityCar as can be seen in Figure 6.10d. It is clearly not beneficial when too many clauses are added, as the overhead due to the additional causes is getting to high. This is reflected in Figure 6.10b where quite some problems perform worse than without transforming clauses. On the other hand there is also no clear trend, when too few clauses are transformed (Figure 6.10a).

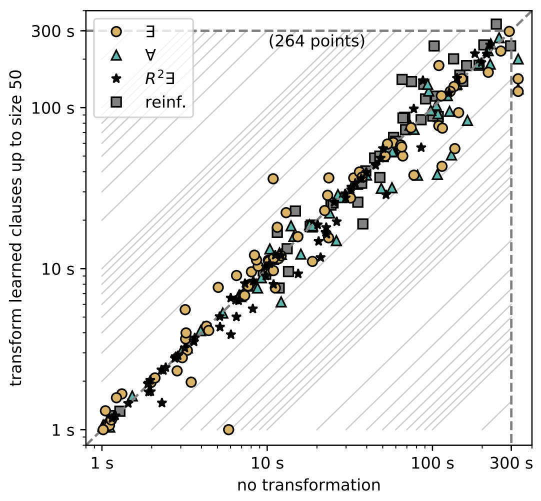
Concluding, it can be beneficial to transform learned clauses but the overhead is too large to justify the moderate improvements, that only occur in some domains. It might be possible to improve these results with careful domain depended tuning or by finding a feature for how many clauses to transform.



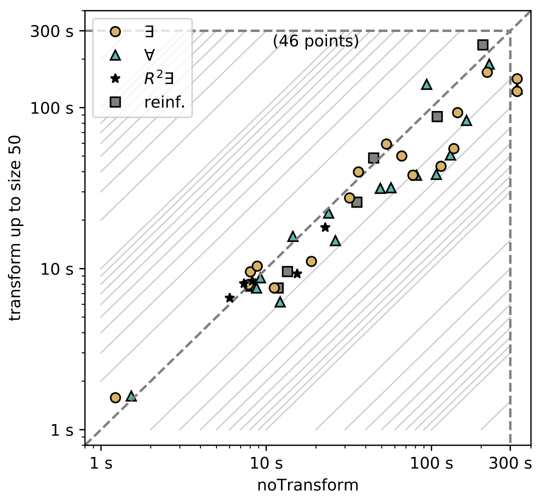
(a) Transforming Clauses With Up to 5 Literals



(b) Transforming Clauses With Up to 1000 Literals

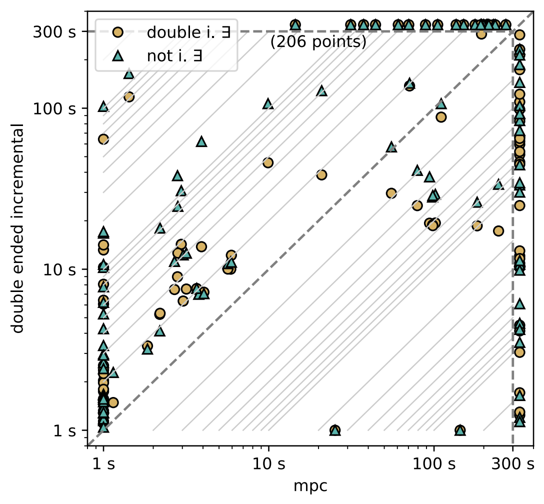


(c) Transforming Clauses With Up to 50 Literals

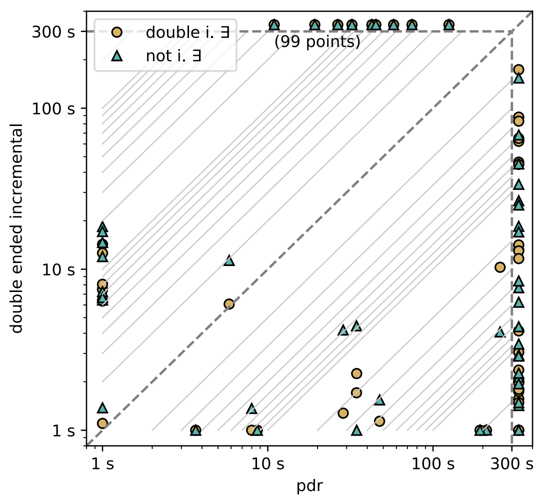


(d) Transforming Clauses With Up to 50 Literals Only Domains Thoughtful and CityCar

Figure 6.10: Comparing the transformation of learned clauses to not transforming any learned clauses.



(a) Comparison to MpC.



(b) Comparison to PDRplan.

Figure 6.11: Comparison to existing tools.

6.8 Comparison to State of the Art

This section compares the double ended incremental approach to the existing state-of-the-art planners Madagascar and PDRplan. Madagascar is a SAT based planner that competed in the last International Planning Competition 2014 and reached the second and third place in the agile track. Its variant MpC uses advanced scheduling of makespans [37], planning specific heuristics [38, 39] and an own SAT solver implementation optimized for planning [36]. PDRplan [45] is based on property directed reachability for planning but replaced the calls to an of the shelf SAT solver with planning specific methods. Note that PDRplan does not support all necessary features and could therefore be only compared on domains Transport, Thoughtful, Parking, Floortile, Childsnack and Barman.

The cactusplot in the introduction (Figure 1.1) compared the double ended incremental encoding to MpC and PDRplan and shows the potential of the double ended incremental approach. Especially, that more problems can be solved within the given time limit, as can be seen Table 6.1 as well.

However, Figure 6.11 shows that the double ended incremental encoding is quite complementary to the two tools it is compared to: Some problems are better solved by MpC or PDRplan while others are not solved by these two tools but are solved by the double ended incremental encoding. Note that this is not only due to the incremental encoding but the tendency is already present when using the non-incremental encoding. It is very domain dependent which solver is best. This can be seen in Table 6.1, that shows the number of solved instances for each domain.

For example the domains barman, ged and parking have been solved by MpC but not by the incremental approaches. The explanation can be found in Table 6.2: These domains require rather large makespans (more than 100), while domains solved by the double ended incremental ∃ encoding required a makespan of 20 at most. The simplest explanation

encoding	incremental	barman	cavediving	childsnack	citycar	floortile	ged	hiking	maintenance	parking	tetris	thoughtful	total
MpC		**3**	5	7	8	**20**	11	5	14	**4**	**4**	5	86
PDRplan		1	-	6	-	10	-	-	-	0	-	**13**	30
$\exists$	non-incremental	0	**7**	**19**	13	**20**	0	5	**20**	0	1	5	90
	single ended	0	**7**	17	14	**20**	0	6	**20**	0	2	5	91
	double ended	0	**7**	18	**19**	**20**	0	**7**	**20**	0	2	5	**94**
$\forall$	non-incremental	0	**7**	16	12	15	0	3	**20**	0	1	5	79
	single ended	0	**7**	18	13	8	0	4	**20**	0	2	5	77
	double ended	0	**7**	18	**19**	15	0	5	**20**	0	2	5	87
$R^2\exists$	non-incremental	0	**7**	**19**	1	15	0	1	**20**	0	1	5	69
	single ended	0	**7**	17	1	14	0	1	**20**	0	1	5	66
	double ended	0	**7**	17	0	15	0	1	18	0	1	5	64
reinf.	non-incremental	0	**7**	13	1	10	0	3	3	0	1	5	43
	single ended	0	**7**	16	2	8	0	4	3	0	3	5	48
	double ended	0	**7**	16	2	10	0	4	3	0	3	5	50

Table 6.1: Number of problems solved within 300 seconds by default Madagascar (MpC), PDRplan and the four tested encodings.

encoding		barman	cavediving	childsnack	citycar	floortile	ged	hiking	maintenance	parking	tetris	thoughtful
MpC	min	225	19	4	9	6	319	4	4	112	6	19
	max	2,559	19	9	319	27	2,559	19	4	225	9	27
incr. $\exists$	min	-	18	3	8	7	-	5	1	-	6	15
	max	-	18	3	14	20	-	8	1	-	7	19

Table 6.2: Minimal and maximal makespan of instances solved within 300 seconds by default Madagascar (MpC) and double ended incremental $\exists$ encoding.

is that MpC can reach those higher makespans due to its non-sequential scheduling of makespans. To evaluate this the non-incremental approach is used to directly solve the makespan known to be satisfiable by MpC. However, the used SAT solver did not find a solution within an extended time limit of 10 minutes for any of the problems in the domains barman, ged and parking. Therefore, scheduling of makespans alone is not sufficient to solve these instances.

7 Conclusion

This thesis investigated the use of incremental SAT solving for SAT based planning by evaluating different incremental encodings for planning and the pigeonhole principle. The latter was used as a case study to provide insights on incremental SAT solving on a series of unsatisfiable formulas. Some of these results can be transferred to planning as well.

The evaluation showed that incremental SAT solving is beneficial for SAT based planing. The presented approaches in combination with a state-of-the-art SAT solver can not only compete with state-of-the-art SAT based planning tools but also complements them by solving different problem instances.

The acceleration reached is partially but not solely caused by learned clauses. As phase saving, and variable activity for the VSIDS heuristic could be eliminated as reasons for the speed up it is necessary to investigate further techniques such as watched literals, which is left for future work.

The use of incremental SAT solving for SAT based planning opens the following research directions:

- When parallelizing the search for a plan with multiple SAT solvers it is possible to share clauses, even between solvers that try to solve different makespans, due to the use of activation literals.
- An incremental SAT solver allows to produce multiple solutions for the same formula, by adding a new clause that removes the previous solution. This can be utilized to generate different plans.

Clearly defined interfaces make it possible to combine the presented approaches with arbitrary SAT encodings for planning and with arbitrary SAT solvers that support the standardized IPASIR interface. This way the presented approaches will profit from future development in both areas.

Bibliography

[1] Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. “A State-Space Acyclicity Property for Exponentially Tighter Plan Length Bounds”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling*. 2017, pp. 2–10.

[2] Albert Atserias and Víctor Dalmau. “A combinatorial characterization of resolution width”. In: *Journal of Computer and System Sciences* 74.3 (2008), pp. 323–334.

[3] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. “Improving glucose for incremental SAT solving with assumptions: application to MUS extraction”. In: *International conference on theory and applications of satisfiability testing*. 2013, pp. 309–317.

[4] Gilles Audemard and Laurent Simon. “Glucose and Syrup in the SAT’16”. In: *Proceedings of SAT Competition 2016 - Solver and Benchmark Descriptions*. 2016, pp. 40–41.

[5] Christer Bäckström and Bernhard Nebel. “Complexity Results for SAS+ Planning”. In: *Computational Intelligence* 11 (1995), pp. 625–656.

[6] Toma Balyo. “Relaxing the relaxed exist-step parallel planning semantics”. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 2013, pp. 865–871.

[7] Tomáš Balyo, Roman Barták, and Otakar Trunda. “Reinforced Encoding for Planning as SAT”. In: *Acta Polytechnica CTU Proceedings*. Vol. 2. 2. Czech Technical University in Prague, 2015, pp. 1–7.

[8] Tomas Balyo et al. “SAT Race 2015”. In: *Artificial Intelligence* 241 (2016), pp. 45–65.

[9] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. 2011, pp. 70–87.

[10] Stephen A Cook. “A short proof of the pigeon hole principle using extended resolution”. In: *Acm Sigact News* 8.4 (1976), pp. 28–32.

[11] Neil T. Dantam et al. “Incremental Task and Motion Planning: A Constraint-Based Approach”. In: *Proceedings of Robotics: Science and Systems*. AnnArbor, Michigan, 2016.

[12] Niklas Eén and Armin Biere. “Effective Preprocessing in SAT Through Variable and Clause Elimination”. In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*. 2005, pp. 61–75.

[13] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. “Efficient implementation of property directed reachability”. In: *International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11, Austin, TX, USA, October 30 - November 02, 2011.* 2011, pp. 125–134.

[14] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers.* 2003, pp. 502–518.

[15] Niklas Eén and Niklas Sörensson. “Temporal induction by incremental SAT solving”. In: *Electronic Notes in Theoretical Computer Science* 89.4 (2003), pp. 543–560.

[16] Richard Fikes and Nils J. Nilsson. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artificial Intelligence* 2.3–4 (1971), pp. 189–208.

[17] Alan M Frisch and Paul A Giannaros. “SAT encodings of the at-most-k constraint. some old, some new, some fast, some slow”. In: *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation.* 2010.

[18] Stephan Gocht and Tomáš Balyo. “Accelerating SAT Based Planning with Incremental SAT Solving”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling.* 2017, pp. 135–139.

[19] Alberto Griggio and Marco Roveri. “Comparing Different Variants of the ic3 Algorithm for Hardware Model Checking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.6 (2016), pp. 1026–1039.

[20] Armin Haken. “The Intractability of Resolution”. In: *Theoretical Computer Science* 39 (1985), pp. 297–308.

[21] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. “A Novel Transition Based Encoding Scheme for Planning as Satisfiability”. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence.* 2010.

[22] Matti Järvisalo, Armin Biere, and Marijn Heule. “Blocked Clause Elimination”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings.* 2010, pp. 129–144.

[23] Henry Kautz and Bart Selman. “Planning as Satisfiability.” In: *ECAI ’92 Proceedings of the 10th European conference on Artificial intelligence.* Vol. 92. 1992, pp. 359–363.

[24] Will Klieber and Gihwon Kwon. “Efficient CNF encoding for selecting 1 from N objects”. In: *Proceedings of the International Workshop on Constraints in Formal Verification.* 2007.

[25] Stefan Kupferschmid et al. “Incremental preprocessing methods for use in BMC”. In: *Formal Methods in System Design* 39.2 (2011), pp. 185–204.

[26] Drew McDermott et al. *PDDL-the planning domain definition language.* 1998.

[27] Matthew W. Moskewicz et al. “Chaff: Engineering an Efficient SAT Solver”. In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001.* 2001, pp. 530–535.

[28] Hidetomo Nabeshima et al. “Lemma Reusing for SAT based Planning and Scheduling”. In: *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006.* 2006, pp. 103–113.

[29] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. “Ultimately Incremental SAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings.* 2014, pp. 206–218.

[30] Chanseok Oh. “COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS”. In: *Proceedings of SAT Competition 2016 - Solver and Benchmark Descriptions.* 2016, pp. 29–30.

[31] Edwin PD Pednault. “ADL and the state-transition model of action”. In: *Journal of logic and computation* 4.5 (1994), pp. 467–512.

[32] Justyna Petke. “SAT encodings of a classical problem: a case study”. In: *Bridging Constraint Satisfaction and Boolean Satisfiability.* 2015, pp. 89–98.

[33] Knot Pipatsrisawat and Adnan Darwiche. “A Lightweight Component Caching Scheme for Satisfiability Solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings.* 2007, pp. 294–299.

[34] Ira Pohl. “Bi-directional and heuristic search in path problems”. PhD thesis. Department of Computer Science, Stanford University, 1969.

[35] Katrina Ray and Matthew L Ginsberg. “The Complexity of Optimal Planning and a More Efficient Method for Finding Solutions.” In: *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008.* 2008, pp. 280–287.

[36] Jussi Rintanen. “Engineering Efficient Planners with SAT.” In: *ECAI’12 Proceedings of the 20th European Conference on Artificial Intelligence.* 2012, pp. 684–689.

[37] Jussi Rintanen. “Evaluation strategies for planning as satisfiability”. In: *Proceedings of the 16th European Conference on Artificial Intelligence.* 2004, pp. 682–686.

[38] Jussi Rintanen. “Planning as satisfiability: Heuristics”. In: *Artificial Intelligence* 193 (2012), pp. 45–86.

[39] Jussi Rintanen. “Planning with SAT, admissible heuristics and A*”. In: *Twenty-Second International Joint Conference on Artificial Intelligence, IJCAI 11.* 2011, pp. 2015–2020.

[40] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. “Planning as satisfiability: parallel plans and algorithms for plan search”. In: *Artificial Intelligence* 170.12 (2006), pp. 1031–1080.

[41] Nathan Robinson et al. “SAT-Based Parallel Planning Using a Split Representation of Actions”. In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009.* 2009.

[42] Peter H. Schmitt. “Formale Systeme. Winter 2013/2014”. lecture notes. URL: `http://i12www.ira.uka.de/~pschmitt/FormSys/FormSys1314/skriptum.pdf` (visited on 07/15/2017).

[43] Ofer Shtrichman. “Tuning SAT checkers for bounded model checking”. In: *International Conference on Computer Aided Verification.* 2000, pp. 480–494.

[44] João P. Marques Silva and Karem A. Sakallah. “GRASP - a new search algorithm for satisfiability”. In: *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design.* 1996, pp. 220–227.

[45] Martin Suda. “Property Directed Reachability for Automated Planning”. In: *Journal of Artificial Intelligence Research (JAIR)* 50 (2014), pp. 265–319.

[46] G Tseitin. “On the complexity ofderivation in propositional calculus”. In: *Studies in Constrained Mathematics and Mathematical Logic* (1968).

[47] Mauro Vallati et al. “The 2014 international planning competition: Progress and trends”. In: *AI Magazine* 36.3 (2015), pp. 90–98.