PASAR – Planning as Satisfiability with Abstraction Refinement

Nils Froleyks, Tomáš Balyo, Dominik Schreiber
Karlsruhe Institute of Technology
Karlsruhe, Germany
n.froleyks@gmail.com, {tomas.balyo,dominik.schreiber}@kit.edu

Abstract

One of the classical approaches to automated planning is the reduction to propositional satisfiability (SAT). In this paper, we present a further improvement to SAT-based planning by introducing a new algorithm named PASAR based on the principles of counterexample guided abstraction refinement (CEGAR). As an abstraction of the original problem, we use a simplified encoding where interference between actions is generally allowed. Abstract plans are converted into actual plans where possible or otherwise used as a counterexample to refine the abstraction. Using benchmark domains from recent International Planning Competitions, we compare our approach to different state-of-the-art planners and find that, in particular, combining PASAR with forward state-space search techniques leads to promising results.

Introduction

Planning is the problem of finding a sequence of actions – a plan – that transforms the world from some initial state to a goal state. The world is fully-observable (the entire world state is known), deterministic (the exact effects of executing an action are known) and static (only the agent we make the plan for changes the world). The number of the possible states of the world as well as the number of possible actions is finite, though possibly very large. We will assume that the actions are instantaneous and therefore we only need to deal with their sequencing. Actions have preconditions, which specify in which states of the world they can be applied as well as effects, which dictate how the world will be changed after the action is executed.

Satisfiability (SAT) based planning is one of the well established approaches to automated planning. It is based on the idea of encoding a planning problem instance into a sequence of SAT formulas and then use a SAT solver to solve them. The method was first introduced by (Kautz and Selman 1992) and is still very popular and competitive. Two reasons for this popularity are the rapidly increasing power of SAT solvers, which are becoming more efficient year by year, and various improvements that have been made to the method since its introduction. Some examples of these improvements are new compact and efficient encodings (Huang, Chen, and Zhang 2010; Rintanen, Heljanko, and Niemelä 2006; Robinson et al. 2009; Balyo 2013), better ways of scheduling the SAT solvers (Rintanen, Heljanko, and Niemelä 2006), specialized SAT solving heuristics for planning problems (Rintanen, Heljanko, and Niemelä 2006), and – most recently – using incremental SAT solving (Gocht and Balyo 2017).

In this paper we introduce a new algorithm named PASAR for SAT-based planning by utilizing the counterexample guided abstraction refinement technique (Clarke et al. 2000). The basic idea of PASAR is that we encode a relaxation of the planning problem instance to SAT, i.e., we leave out some of the clauses, thus obtaining an abstraction of the proper encoding. Then we find a solution to the SAT formula, which can be decoded into a sequence of actions P'. If P' is a valid plan, the procedure terminates. Otherwise, we attempt to transform the abstract plan into a valid plan using various techniques from conventional planning as well as a number of custom optimizations. If we fail to compute a valid plan, P' constitutes a counterexample to our abstract encoding and we need to refine it. We refine our abstraction by adding additional clauses (derived from P') to our formula. We repeat the cycle of solving our formula and refining our abstraction until we arrive at a valid plan.

The experimental results based on benchmarks from the recent International Planning Competitions show that PASAR solves a broad set of instances, performing better than state-of-the-art SAT-based planning in various cases. We also combine PASAR with forward state-space search techniques and find that such a combined approach significantly increases the amount of instances solved by our planning system.

Preliminaries

We give the basic definitions of the formalisms and techniques used in the paper.

Planning

In the introduction we briefly described what planning is, in this section we give the formal definitions. We will use the

Figure 1: The initial state (left) and the goal state (right) for the Trucking planning domain.

multivalued SAS+ formalism (Bäckström and Nebel 1995) instead of the classical STRIPS formalism (Fikes and Nilsson 1971) based on propositional logic.

A planning task Π in the SAS+ formalism is defined as a tuple $\Pi = (X, O, s_I, s_G)$ where

- $X = \{x_1, \ldots, x_n\}$ is a set of multivalued variables with finite domains $\mathrm{dom}(x_i)$.
- O is a set of actions (or operators). Each action $a \in O$ is a tuple $(\mathrm{pre}(a), \mathrm{eff}(a))$ where $\mathrm{pre}(a)$ is the set of preconditions of a and $\mathrm{eff}(a)$ is the set of effects of a. Each precondition and each effect is of the form $x_i = v$ where $x_i \in X$ and $v \in \mathrm{dom}(x_i)$.
- A *state* is a set of assignments to the state variables. Each state variable has exactly one value assigned from its respective domain.

  Let S be the set of all states. $s_I \in S$ is the initial state.

  We identify sets of states that fulfill *partial assignments*. In a partial assignment not all variables are assigned a value. s_G is a partial assignment and a state $s \in S$ is in the set of goal state if and only if $s_G \subseteq s$.

An action a is *applicable* in a given state s if $\mathrm{pre}(a) \subseteq s$. By $s' = \mathrm{apply}(a, s)$ we denote the state after executing the action a in the state s, where a is applicable in s. All the assignments in s' are the same as in s except for the assignments in $\mathrm{eff}(a)$ which replace the corresponding (same variable) assignments in s.

We say that two actions a_i and a_j are *interfering* if and only if $\mathrm{pre}(a_i)$ is inconsistent with $\mathrm{eff}(a_j)$ or $\mathrm{pre}(a_j)$ is inconsistent with $\mathrm{eff}(a_i)$.

A *plan* P of *length* k for a given planning task Π is a sequence of actions $P = \langle a_1, \ldots, a_k \rangle$ such that $s_G \subseteq \mathrm{apply}(a_k, \mathrm{apply}(a_{k-1} \ldots \mathrm{apply}(a_2, \mathrm{apply}(a_1, s_I)) \ldots))$.

Example 1 *Trucking Domain. A truck is moving between 3 locations L_A, L_B, and L_C starting at L_A. Two packages are located at L_A and L_B, which can be picked up or dropped by the truck at any of the given locations. The goal is to deliver both packages to location L_C. See Figure 1 for an illustration of the planning task. We model the problem using three state variables: the location of the truck x^T, $\mathrm{dom}(x^T) = (L_A, L_B, L_C)$, and the location of the packages x^{P_1} and x^{P_2}, $\mathrm{dom}(x^{P_1}) = \mathrm{dom}(x^{P_2}) = (L_A, L_B, L_C, T)$. We have three kinds of actions: move (*m*), pickUp (*pu*), and drop (*d*). For each $j, k \in \{A, B, C\}$ and $i \in \{1, 2\}$*

- $\mathrm{m}(L_j, L_k) = (\{x^T = L_j\}, \{x^T = L_k\})$
- $\mathrm{pu}(P_i, L_j) = (\{x^T = L_j, x^{P_i} = L_j\}, \{x^{P_i} = T\})$
- $\mathrm{d}(P_i, L_j) = (\{x^T = L_j, x^{P_i} = T\}, \{x^{P_i} = L_j\})$

A possible plan for this planning task consists of the following six actions $P = \langle\ \mathrm{pu}(P_1, L_A), \mathrm{m}(L_A, L_B), \mathrm{pu}(P_2, L_B), \mathrm{m}(L_B, L_C), \mathrm{d}(P_1, L_C), \mathrm{d}(P_2, L_C)\ \rangle$.

SAT Solving

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

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 (Eén and Sörensson 2003) has two methods. One adds a clause C and the other solves the formula with additional assumptions in form of a set of literals A:

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

The method $solve$ determines the satisfiability of the conjunction of all previously added clauses under the condition that all literals in A are true.

SAT-Based Planning

The basic idea of solving planning as SAT (Kautz and Selman 1992) is to encode the planning problem up to a certain number of *steps* as a Boolean formula F_i such that F_i is satisfiable if and only if there is a plan with i steps or less. Thereby, a *step* commonly denotes a set of actions that can be executed *in parallel*, i.e. simultaneously, in some consistent way. Additionally, a valid plan must be constructible from a satisfying assignment of F_i. To find a plan, it is common to check the plan encodings $F_0, F_1, F_2, \ldots$ until the first satisfiable formula is found (this approach is called *sequential scheduling*). The following constraints are encoded:

0. The values of state variables in the first step are consistent with the initial state.
1. The values of state variables in the final step are consistent with the goal conditions.
2. At each step, each state variable assumes exactly one value from its respective domain.
3. If an action is executed at step t, then its preconditions are satisfied at step t and its effects are valid at step $t + 1$.
4. If a value of a state variable changes between steps t and $t + 1$, then there must be an action at step t causing this change by one of its effects.
5. Pairs of interfering actions are forbidden to be executed simultaneously in one step.

For instance, the simplest approach to encode rule 5 is to introduce clauses of the form $(\neg do_{a_1}^{(t)} \vee \neg do_{a_2}^{(t)})$ for each pair of actions (a_1, a_2) and the corresponding Boolean variables

```
Procedure PASAR(Π = (X, O, s_0, s_G))
    k := 1;
    F := AbstractEncode(Π);
    while true do
        F_k := Instantiate(F, k);
        assignment := Solve(F_k);
        if assignment = UNSAT then
            k := IncreaseMakespan(k);
        else
            ρ := ExtractAbstractPlan(assignment);
            (π, F) := RepairPlan(F, ρ);
            if π ≠ FAILURE then
                return π;
            end
        end
    end
```

Algorithm 1: PASAR planning procedure.

```
Procedure RepairPlan(F, ⟨s_0, A_0, s_1, A_1, ..., s_k⟩)
    π := ⟨⟩;
    for i = 0, ..., k − 1 do
        if InterferenceGraph(A_i) is cyclic then
            A'_i := Replan(s_i, s_{i+1});
            if A'_i = FAILURE then
                F := RefineAbstraction(F, A_i);
                return FAILURE;
            else
                π = π ∘ ValidOrdering(A'_i);
            end
        else
            π = π ∘ ValidOrdering(A_i);
        end
    end
    return (π, F);
```

Algorithm 2: Procedure of converting an abstract plan into an actual plan.

$(do_{a_1}^{(t)}, do_{a_2}^{(t)})$ which indicate their execution at step t. This leads to a purely sequential encoding without any actions being executed in parallel.

A straight forward improvement of this encoding is the *foreach-step* semantics (Rintanen, Heljanko, and Niemelä 2006) where parallel actions are allowed as long as they are not interfering, i.e., as long as *each* possible action ordering within each step results in a valid plan. The even more general *exists-step* semantics also allows for the parallel execution of interfering actions as long as there *exists* some action ordering within each step resulting in a valid plan (Rintanen, Heljanko, and Niemelä 2006).

Counterexample Guided Abstraction Refinement

Counterexample Guided Abstraction Refinement (CEGAR) has first been proposed in (Clarke et al. 2000) for the purpose of improved model checking techniques. The general idea is to maintain an abstraction of the actual problem which relaxes some of the system's properties, simplifying the search for a solution. If a state considered during search is found to be invalid regarding the actual problem, it is considered as a counterexample that can be used to refine the abstraction.

As automated planning features a combinatorial search space just like model checking, CEGAR has been adopted into planning approaches as well. As such, the paradigm has been used as the basis for forward search heuristics (Seipp and Helmert 2013) as well as for probabilistic planning (Chatterjee et al. 2005). To our best knowledge, we are the first to introduce the CEGAR paradigm to SAT-based planning.

CEGAR for SAT-based Planning

Our CEGAR-based planning procedure is described in Algorithm 1. We begin to encode the given planning problem into a set of propositional logic rules F that can be instantiated into an actual formula F_k for any number of steps k (makespan). Note that we do not add clauses from rule (5), thus allowing any number of parallel actions as long as the preconditions and effects of each applied action are met.

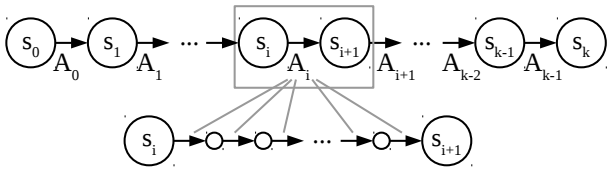

Figure 2: Illustration of an abstract plan with action sets $\langle A_0, A_1, \ldots, A_{k-1} \rangle$ (top) and the identification of a valid arrangement of actions in A_i to reach s_{i+1} from s_i (bottom).

For increasing values of k, we attempt to solve the corresponding formula until the solver reports satisfiability. Now, a sequence ρ of action sets A_i and intermediate states s_i can be extracted from the satisfying assignment. We call ρ an *abstract plan*. Next, we call the sub-procedure RepairPlan(F, ρ) as described in Algorithm 2 to check the validity of ρ and, if necessary, try to repair invalid steps in it.

The structure of an abstract plan is illustrated in Figure 2. For each adjacent pair of states (s_i, s_{i+1}) that the solver found, we check if the set of executed actions at step i is consistent. We describe the dependencies between actions in A_i with a simple *interference graph* where nodes are actions and a directed edge (a_1, a_2) indicates that a_1 requires a precondition that is removed by a_2 (in other words: a_1 must be applied *before* a_2). It follows that there is a valid ordering of actions in A_i if and only if the interference graph is cycle-free. If there is a valid ordering, we can find it by applying a topological sort on the graph. If the graph does contain a cycle, then the involved nodes correspond to the set of conflicting actions.

If the interference graph of A_i contains a cycle, we may attempt to *replan* this invalid step by starting a separate planning procedure where s_i is the initial state and s_{i+1} is the fully defined goal state. We employ a greedy best-first search approach driven by a Hamming distance heuristic with ties broken randomly. Note that this planning task is supposed to be comparably easy if the plan is reparable, so we fix

a certain small time frame to solve this planning problem. If no solution is found, we consider the set of interfering actions as a counterexample to a valid plan. We add non-interference clauses from rule (5) for the concerned set of actions to the abstract formula F and report that the previously found abstract plan cannot be repaired. The procedure is then restarted for the same makespan k, but with a refined formula F_k. Note that even if the best-first search misses a solution the solver does not become incomplete. Every plan will be found eventually, albeit at a higher makespan.

There are two major reasons for specifically omitting the non-interference clauses from the initial encoding. Firstly, from a pragmatic point of view, non-interference clauses are the most expensive part of a naïve SAT encoding of a planning problem, leading either to a very large number of clauses or to a significant number of additional variables. Secondly, the extent to which non-interference clauses are added to the encoding essentially characterizes the action semantics that is realized in the encoding. As we begin without any non-interference clauses, we realize a form of action parallelism that is equivalent to *exists-step* semantics until we find some invalid, irreparable plan. Afterwards, restrictions are added gradually until a plan is found, resulting in *foreach-step* semantics in the worst case where non-interference clauses are added for each pair of conflicting actions.

Example 2 *Applying PASAR to Example 1. The initial solving attempt may return a plan* $\langle A_1, A_2, A_3\rangle = \langle\{\mathrm{pu}(P_1, L_A), \mathrm{m}(L_A, L_B)\}, \{\mathrm{pu}(P_2, L_B), \mathrm{m}(L_B, L_C)\}, \{\mathrm{d}(P_1, L_C), \mathrm{d}(P_2, L_C)\}\rangle$ *along with the intermediate states* $\langle s_0, s_1, s_2, s_3\rangle$*. It has three steps with two actions each. Next, we analyze the first set of actions* A_0 *by building its interference graph (Figure 3) and we find that the ordering as indicated above leads to the intended result without any inconsistencies. We proceed to analyzing* A_1 *and then* A_2 *in a similar way and thus find that the found plan with established action orderings is valid. The plan can be output as a solution.*

As a nontrivial example, consider an adjusted version of Example 1 where the truck may only contain a single package at any given point in time. We introduce an additional state variable, l*,* $\mathrm{dom}(l) = (\mathrm{empty}, \mathrm{full})$*, that signalizes whether the truck has some package loaded. We add* $\{l = \mathrm{empty}\}$ *to* $\mathrm{Pre}(\mathrm{pu}(P_i, L_j))$ *and to* $\mathrm{Eff}(\mathrm{d}(P_i, L_j))$*, and we add* $\{l = \mathrm{full}\}$ *to* $\mathrm{Eff}(\mathrm{pu}(P_i, L_j))$*. Additionally, we introduce another package* P_3 *(at* L_A*) which needs to be transported to* L_C*.*

Running the proposed algorithm on this task, a minimum makespan plan would be
$\langle\{\mathrm{pu}(P_1, L_A), \mathrm{pu}(P_3, L_A), \mathrm{m}(L_A, L_B)\}, \{\mathrm{m}(L_B, L_C)\},$
$\{\mathrm{d}(P_1, L_C), \mathrm{d}(P_3, L_C), \mathrm{m}(L_C, L_B)\},$
$\{\mathrm{pu}(P_2, L_B), \mathrm{m}(L_B, L_C)\}, \{\mathrm{d}(P_2, L_C)\}\rangle$.
In this invalid plan, the truck loads two packages at once in the very first step. Analyzing A_0 *leads to the insight that the dependencies of the actions* $\{\mathrm{pu}(P_1, L_A), \mathrm{pu}(P_3, L_A)\}$ *are cyclic (Figure 3); the first action will always remove a requirement for the second action. In order to correct the plan, a state-space search attempting to reach* s_1 *from* s_0 *is initiated. It fails because it is impossible to have both packages inside the truck at the same time. Thereafter, we add a non-interference clause between* $\mathrm{pu}(P_1, L_A)$ *and* $\mathrm{pu}(P_3, L_A)$ *and restart the procedure. In the next iteration, no more invalid plans of this kind will occur, and a valid plan will be found similar to the previous example.*

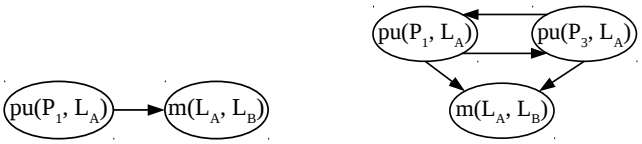

Figure 3: Interference graph of $\{\mathrm{pu}(P_1, L_A), \mathrm{m}(L_A, L_B)\}$ (left) and $\{\mathrm{pu}(P_1, L_A), \mathrm{pu}(P_3, L_A), \mathrm{m}(L_A, L_B)\}$ (right).

Makespan and Refinement Scheduling

In SAT-based planning, it is important by what amount the makespan k is increased when the current makespan turns out to be unsolvable (Rintanen 2004). We update $k := \max(k+1, 1.2k)$ each time IncreaseMakespan(k) is called, ensuring that the the amount of considered steps grows exponentially until some initial abstraction is found.

Similarly, if an abstract plan is found, a high amount of iterations may be necessary until all required refinements have been added to the formula such that either a valid plan can be constructed or the current makespan can be identified as overall unsatisfiable. In certain planning instances, it may even be necessary to add each of the up to $\mathcal{O}(|O|^2)$ clauses between pairs of actions in order to find a valid plan. To limit the number of refinement iterations in such cases, we introduce a fallback mechanism that triggers as soon as a certain large amount of refinements has been added during the same makespan, as such a behavior indicates stagnation in our planning process. In such a case, we add all remaining non-interference clauses between actions that are still missing, thus directly establishing the standard foreach-step semantics rather than slowly converging towards it.

Improvement of Abstract Plans

We present a number of techniques to optimize the process of repairing an abstract plan: State sparsification, step reduction, caching of partial plans, and step skipping.

State Sparsification As described in Algorithm 1, a forward search procedure is initiated whenever some step i is found to be invalid in an abstract plan, attempting to find a valid plan between s_i and s_{i+1}. Contrary to usual planning problems, the goal s_{i+1} is a fully defined state, i.e. each variable is assigned some value. This can be problematic if some of the differences between s_i and s_{i+1} are not essential for the remainder of the plan, and if the action that causes these changes turns out to be invalid with respect to other actions. Essentially, the planning problem is overdefined which makes it harder to find a valid plan.

To counteract this issue, we identify the partial assignment a_i that is relevant to the abstract plan for each step s_i. When a forward search is initiated from state s_i the goal is to

fulfill a_{i+1}. Note that we only start a forward search from a state if a valid plan to reach it from the initial state is known. Therefore the search always starts from a fully defined state and all applicable actions are known.

To find the relevant partial assignments, we execute a plan sparsification procedure directly after finding an abstract plan. Beginning from the final state s_k, we go backwards along the abstract plan and only keep the state assignments and actions which are essential to satisfying each of the goals. At the final step k, we keep each goal assignment s_G and each action in A_{k-1} that causes one of these assignments. At a previous step $i < k$, we keep the set of assignments in s_i that are required for the essential actions from A_i, and we keep the actions in A_{i-1} that are required to produce one of the essential assignments. On a conceptual level, this procedure is similar to the backwards search that is commonly employed to extract a plan from a planning graph (Blum and Furst 1997).

Using the sparsified abstract plan, the actual goals of individual forward search sub-procedures can be expressed in a more concise way. As a result, more focused state space search can be employed.

Step Reduction After state sparsification, each set of actions A_i is analyzed as to whether it is actually required in order to achieve the goal. More specifically, we try to remove A_i and s_{i+1} from the abstract plan and execute the remaining plan. Any successive steps where some actions are not applicable any more are removed in the same way. If all goals are left satisfied in the end, then the shortened abstract plan is still valid and can be used for the upcoming replanning phase. This technique reduces the amount of necessary replanning procedures and can also reduce the makespan of the plans that are returned in the end.

Caching of Partial Plans In some cases, our procedure will partially repair an abstract plan before failing at a certain step i because the actions in A_i are interfering in a non-trivial way. The found plan prefix $\langle A_0, \ldots, A_{i-1} \rangle$ is then discarded and the search for a new abstract plan begins. In order to profit from such partial plans even in later iterations, we cache the plan prefix by remembering its last valid state s_i and the sequence of action sets $\langle A_0, \ldots, A_{i-1} \rangle$ that leads to it. If we encounter state s_i in another abstract plan at some later point, we can directly reuse the known plan prefix instead of performing individual action orderings and replannings. In our implementation, we use the sparsified representation of s_i to achieve a higher number of matches than if the whole state was remembered.

We proceed in a similar way for plan suffixes: In addition to repairing an abstract plan from start to goal, we can also traverse it backwards starting from the goal and build a valid plan suffix until some step j features an invalid set of actions A_j. Note however that, unlike in a plan prefix, we do not initiate a replanning procedure: When fixing the plan from the start, we maintain the state that is reached after executing the current plan prefix and we can determine which actions are applicable. During backwards traversal we do not have a fully defined state but partial assignments as a result of state sparsification. We then cache the earliest valid state

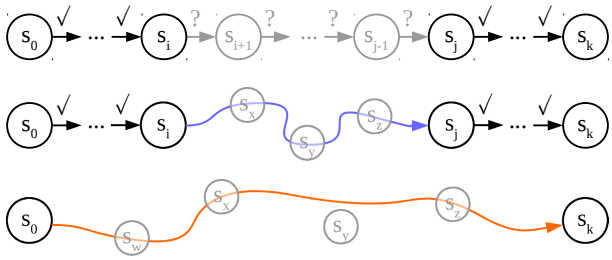

Figure 4: Illustration of step skipping: Partially repaired abstract plan (top), forward search by local step skipping (mid) and by global step skipping (bottom). States in gray represent intermediate states from the abstract plan that are used as a guidance for the forward search heuristics.

s_{j+1} together with the plan suffix $A_{j+1}, \ldots, A_{k-1}$. When encountering s_{j+1} later on, a shortcut to a goal state can be taken.

Step Skipping As illustrated in Figure 4, assume that for some plan repair process we already found a plan prefix $\langle A_0, \ldots, A_{i-1} \rangle$ from initial state s_0 as well as a plan suffix $\langle A_{j+1}, \ldots, A_{k-1} \rangle$ to a goal state s_k, but that both of these partial plans cannot be extended any further because their respective adjacent steps i and j feature invalid action sets. One of the following options applies:

1. For each $l \in \{i, i+1, \ldots, j-1\}$, some plan from s_l to s_{l+1} can be found.
2. For some $l \in \{i, i+1, \ldots, j-1\}$, s_{l+1} is unreachable from s_l. However, a plan between s_i and s_j can be found.
3. s_j is overall unreachable from s_i.

In case of option 1, the basic procedure as described in Algorithm 1 will fully repair the plan if the admitted time limit per replanning is large enough. This is not the case if option 2 or 3 applies, as a local dead-end prevents the abstract plan from being fully repaired.

In order to treat these cases accordingly and make the best use of the found abstract plan, we introduce two stages of *step skipping*. In this technique, we want to allow deviations from the originally found sequence of states, but still utilize them as an overall guideline.

In *local step skipping*, we employ a forward search planning procedure from s_i to s_j, using $\{s_{i+1}, \ldots, s_{j-1}\}$ as a guidance in the form of a heuristic. In *global step skipping*, we employ the same procedure from s_0 to s_k, again using the states in between as a guidance.

For global step skipping, the exact heuristic $h(s)$ we use to estimate the remaining distance from a state s to the goal is defined as

$$h(s) = \sum_{x=1}^{k} d_M(s, s_x) \cdot c^{x-1} \qquad (1)$$

for a constant $1 < c < 2$. For local step skipping, the same heuristic applies with an adjusted range for x. The function $d_M(s, s')$ evaluates the Hamming distance between two states: it is equal to the amount of variable assignments that

are different between s and s'. In this case, we calculate the distance from the current state s to each of the guiding states $\{s_1, \ldots, s_k\}$. The factor c^{x-1} serves as a weighting: similarity to one of the guiding states is more valuable the closer it is to the goal.

We modify the procedure from Algorithm 2 as follows: Instead of the original procedure to repair the plan (line 5), we attempt replanning with local step skipping for a certain time frame. If this also fails, we employ global step skipping for a certain time frame. Only if this fails as well, we refine the abstraction and return "failure" for this abstract plan.

Implementation Details

We provide some essential details regarding the implementation of our approach.

We use the grounding of PDDL problem files into SAS+ provided by Fast Downward (Helmert 2006) to translate problem instances into a ground representation that is well-suited for the encoding into propositional logic.

We utilize incremental SAT solving for our planning procedure, i.e., only one single CNF formula is maintained and extended over the course of the entire algorithm. For each call of Instantiate(F, k) (Algorithm 1, line 5), the abstract clauses F are added as necessary until all constraints for steps $0, 1, \ldots, k$ are encoded. The problem's goal is encoded as a set of assumptions that is considered only for the upcoming Solve(F_k) call and dropped afterwards. This way, clauses never need to be removed from the formula even when the makespan is extended.

We use IPASIR, a generic interface for incremental SAT solving (Balyo et al. 2016). As IPASIR can be used together with any popular SAT solver, we chose Glucose (Audemard and Simon 2009), a highly popular and award-winning SAT solver, as our solving backend.

The source code of our approach is available at https://github.com/froleyks/pasar.

Experimental Evaluation

In the following, we explain our experimental setup. All experiments were conducted on a AMD Epyc 7551P with 64 virtual cores clocked at 2.0-3.0 GHz and 256GB DDR4 RAM, running Ubuntu 18.04.

We used the benchmarks from the satisficing and optimal tracks of the International Planning Competitions (IPC) 2014 and 2018. We did not include any of the benchmarks that include conditional effects, as our encoding is not yet equipped to deal with these. We admit up to five minutes of run time per instance. Up to 16 instances are executed in parallel, but we ensure that this is done only for instances of the same domain and the same planning approach, such that out-of-memory issues do not interfere between different domains or competitors.

The following variants of PASAR are evaluated:

- **p1**: The action sets A_i are checked for a valid ordering in an abstract plan; no replanning is done.
- **p2**: If necessary, do forward search replanning between neighbored states with state sparsification, step reduction, and caching of partial plans.
- **p3**: Like p2, but with local step skipping instead of replanning between neighbored states.
- **p4**: Like p3, but with global step skipping as an additional replanning procedure if local step skipping fails.
- **p5**: A hybrid variant where after a global time limit (200s) forward search on the entire problem is employed if not a single abstract plan has been found yet (Hamming distance heuristic, ties broken randomly).

The motivation for the configuration p5 is that we would like to investigate whether SAT-based planning and conventional greedy best-first planning can complement each another in a beneficial way in our case. More specifically, instances where no abstract plan can be found at all are evidently difficult for a SAT solver, so we want to explore how many of such instances are solvable with simple search space techniques instead.

As a comparison, we include the SAT planner Madagascar (Rintanen 2013) which uses its own integrated SAT solver and makes use of various optimizations, both in its default configuration with exists-step semantics (**MpC**) and foreach-step semantics (**Ma**). As an approach that is more similar to ours, we also compare PASAR to Incplan (Gocht and Balyo 2017) where the encodings of Madagascar are used but a general-purpose incremental SAT solver is employed (**inca** and **ince** with foreach- and exists-step semantics respectively) . As a reference, we also include the state-of-the-art planning system Fast Downward (Helmert 2006) with the LAMA 2011 configuration (**LAMA**).

The domain-specific results of the evaluations are given in Tab. 1, and Figure 5 visualizes the relative performances between p[i] and p[i+1]. On some domains such as childsnack, floortile or tetris, we can see that even the most basic approach p1 is able to effectively find valid plans by not encoding any non-interference between actions and only adding such clauses where necessary. Employing replanning procedures between neighbored states in the abstract plan (p2) does not lead to overall improvements over p1. However, extending these replanning procedures by using step skipping (p3) yields drastic improvements in particular for the openstacks domain. These results hardly improve by including global step skipping (p4); it seems that in the most cases where a plan is reparable within reasonable time constraints, local step skipping is already able to find it. Last but not least, combining our procedure with a greedy state-space search as a fallback option (p5) leads to surprising improvements on multiple domains such as petri-net-alignment or visitall, overall resulting in more solved instances than any of the tested Madagascar variants. We can see here that the capabilities of our partially SAT-based planning approach and of "textbook forward search" are orthogonal up to a certain degree, and that a combination of both is appealing in order to resolve a broad set of planning instances.

By looking at the scatter plots in Figure 5 we gain some insight into the run times of PASAR versions p1 – p5. In most cases (except for p1 vs p2 – the introduction of replanning by forward search) the runtimes of the five versions are very similar on the easy instances (solved in under two minutes). The differences manifest in the top right corners

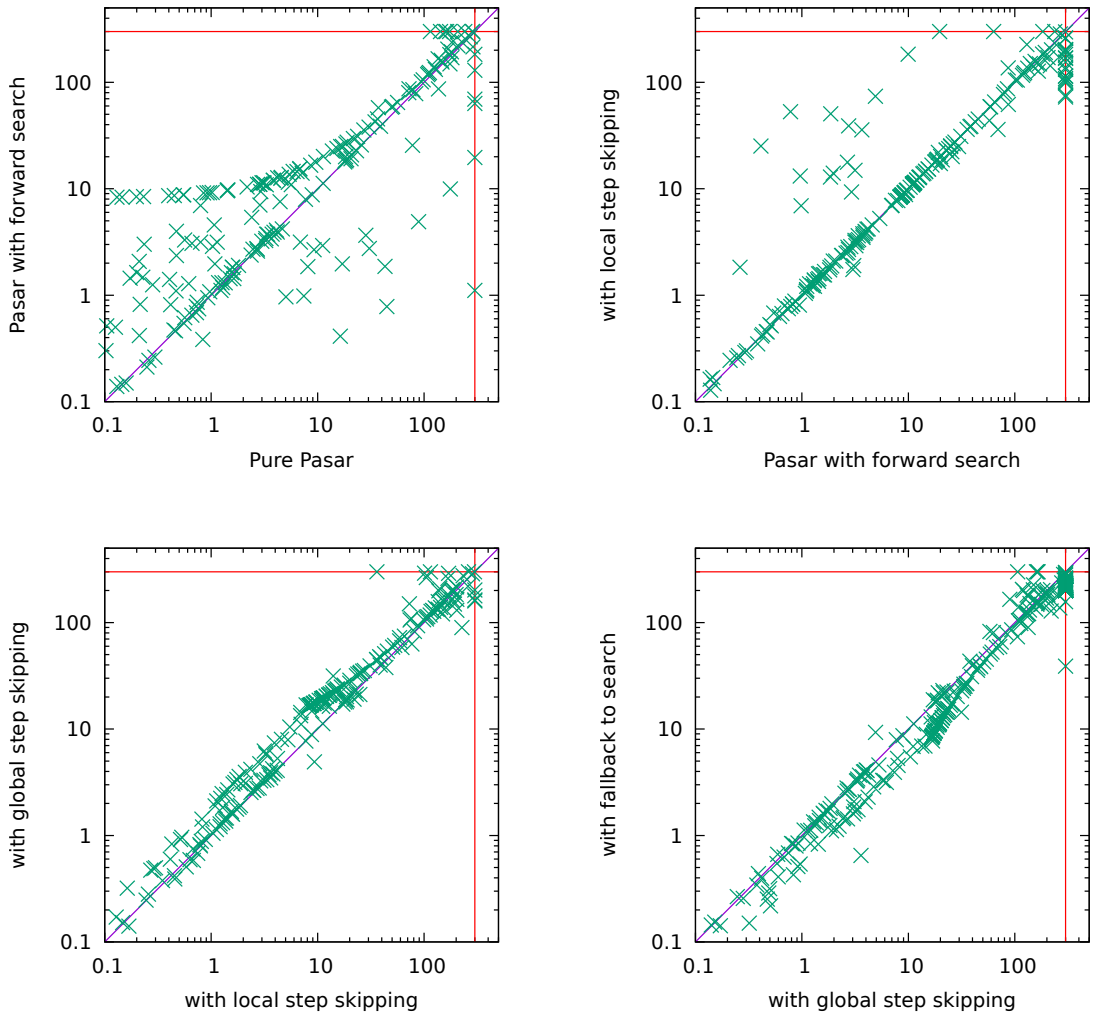

Figure 5: Scatter plots between the run times of p[i] and p[i+1]. All measures are in seconds. Timeouts are plotted at $t = 300s$.

(the harder instances) and in the number of solved instances. From that we can conclude that the versions are somewhat orthogonal on the harder instances and similar on easy ones.

As for the comparison to the competitors (see Table 1), in many domains we match the performance of LAMA, but there are domains such as opt-barman, sat-ged, or sat-termes where the SAT-based planners do not stand a chance. The Madagascar planners (Ma, MpC) perform very well on domains such as opt-barman and opt-tidybot, but the results do not transfer to inca or ince (that use the same encoding as Ma and MpC with a standard incremental SAT solver), which indicates that it is the special features (related to heuristic search) of Madagascar's custom SAT solver that solve these problems. Overall, we can see that our best non-hybrid PASAR version p4 can outperform inca and ince, which represent the state-of-the-art planning as satisfiability when using off-the-shelf SAT solvers. If we include a forward search with the trivial Hamming distance heuristic in PASAR (version p5), we can outperform both Madagascar versions. We still have a lot of work to do to match the overall performance of LAMA; however, in some domains such as childsnack, floortile, and snake we can already solve a lot more instances.

Conclusion

In this paper, we have presented a novel SAT-based planning approach that utilizes the paradigm of counterexample guided abstraction refinement. We have argued that omitting non-interference clauses between actions is an appropriate abstraction for the case of SAT-based planning, and that we achieve a solving procedure with a small amount of clauses and a significant amount of actions executed in parallel. We have presented an approach to transform an abstract found plan into an actual plan, which makes use both of conventional planning techniques and of novel optimization strategies tailored to our specific use case. In thorough

Table 1: Number of solved instances per domain and competitor. Numbers in brackets indicate for how many instances any abstract plan has been found. The row "Wins" indicates in how many domains the respective approach was one of the best approaches regarding solved instances.

Domain	LAMA	Ma	MpC	inca	ince	p1	p2	p3	p4	p5
opt-barman	8	**14**	**14**	0	1	2(14)	3(14)	3(14)	3(14)	3(14)
sat-barman	**19**	0	4	0	0	0(20)	0(20)	0(20)	0(20)	0(20)
opt-childsnack	11	**20**	15	**20**	**20**	**20**(20)	**20**(20)	**20**(20)	**20**(20)	**20**(20)
sat-childsnack	5	16	7	**20**	15	**20**(20)	**20**(20)	**20**(20)	**20**(20)	**20**(20)
opt-data-network	**20**	**20**	**20**	**20**	**20**	**20**(20)	**20**(20)	**20**(20)	**20**(20)	**20**(20)
sat-data-network	**11**	4	4	2	4	2(20)	2(20)	2(20)	2(19)	2(20)
opt-floortile	8	**20**	**20**	17	**20**	**20**(20)	**20**(20)	**20**(20)	**20**(20)	**20**(20)
sat-floortile	2	**20**	**20**	15	**20**	**20**(20)	**20**(20)	**20**(20)	**20**(20)	**20**(20)
opt-ged	**20**	**20**	**20**	**20**	**20**	**20**(20)	**20**(20)	**20**(20)	**20**(20)	**20**(20)
sat-ged	**20**	13	12	0	0	0(0)	0(0)	0(0)	0(0)	0(0)
opt-hiking	**20**	10	9	19	18	16(20)	18(20)	16(20)	16(20)	16(20)
sat-hiking	**15**	4	6	7	8	4(10)	5(10)	5(9)	5(9)	6(9)
opt-openstacks	**20**	**20**	**20**	0	0	4(20)	0(20)	18(20)	18(20)	16(20)
sat-openstacks	**17**	2	0	0	0	0(2)	0(11)	0(0)	0(1)	4(4)
opt-organic-synthesis-split	**16**	12	12	11	11	9(15)	9(15)	9(14)	10(14)	9(14)
sat-organic-synthesis-split	6	**7**	**7**	3	4	4(9)	3(9)	3(9)	3(10)	3(6)
opt-petri-net-alignment	**20**	1	14	0	15	7(7)	10(10)	7(7)	10(10)	16(15)
opt-snake	12	15	15	5	4	4(4)	4(4)	3(3)	3(3)	**20**(3)
sat-snake	3	7	7	0	0	1(1)	0(0)	1(1)	1(1)	**17**(0)
opt-termes	**18**	0	1	2	5	2(12)	3(12)	2(12)	2(12)	2(12)
sat-termes	**13**	0	0	0	0	0(3)	0(3)	0(2)	0(3)	0(3)
opt-tetris	**17**	15	**17**	12	**17**	**17**(17)	**17**(17)	**17**(17)	**17**(17)	**17**(17)
sat-tetris	9	5	8	3	6	**11**(20)	10(20)	**11**(20)	10(20)	**11**(20)
sat-thoughtful	**15**	5	5	5	5	5(5)	5(5)	5(5)	5(5)	5(5)
opt-tidybot	15	16	**20**	1	2	0(9)	0(10)	0(4)	0(3)	1(5)
opt-transport	**20**	**20**	**20**	18	**20**	13(14)	14(15)	14(16)	13(16)	14(15)
sat-transport	**8**	0	0	0	0	0(0)	0(0)	0(0)	0(0)	0(0)
opt-visitall	**20**	**20**	**20**	6	6	11(11)	9(9)	10(10)	11(11)	**20**(8)
sat-visitall	19	0	0	0	0	0(0)	0(0)	0(0)	0(0)	**20**(0)
Total Solved	407	306	317	206	241	232	232	246	249	322
Wins	18	10	11	4	7	8	7	8	7	12

evaluations, we showed that our SAT-based approach is able to solve a considerable set of IPC benchmark instances, performing better than state-of-the-art SAT-based planning on multiple domains. We have found that employing simple forward search techniques in the case where our abstraction fails to effectively approximate the original problem enables us to solve even more instances, indicating that a combination of these techniques is highly attractive for efficient automated planning.

Future Work

For future work, we would like to integrate conditional effects into our planning approach in order to make PASAR useful for a wider set of realistic planning instances. Furthermore, we will consider different approaches to parallelize our approach in an effective way. Last but not least, we will explore more relaxed abstractions for our approach, enabling it to find an initial abstract plan in more cases than before.

References

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Twenty-first International Joint Conference on Artificial Intelligence*.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:625–656.

Balyo, T.; Biere, A.; Iser, M.; and Sinz, C. 2016. SAT Race 2015. *Artificial Intelligence* 241:45–65.

Balyo, T. 2013. Relaxing the relaxed exist-step parallel planning semantics. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, 865–871. IEEE Computer Society.

Blum, A., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Chatterjee, K.; Henzinger, T. A.; Jhala, R.; and Majumdar, R. 2005. Counterexample-guided planning. In *Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI'05, 104–111. Arlington, Virginia, United States: AUAI Press.

Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, 154–169. Springer.

Eén, N., and Sörensson, N. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science - BMC'2003, First International Workshop on Bounded Model Checking* 89(4):543–560.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3/4):189–208.

Gocht, S., and Balyo, T. 2017. Accelerating SAT based planning with incremental SAT solving. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Huang, R.; Chen, Y.; and Zhang, W. 2010. A novel transition based encoding scheme for planning as satisfiability. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Tenth AAAI Conference on Artificial Intelligence*, 359–363.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12-13):1031–1080.

Rintanen, J. 2004. Evaluation strategies for planning as satisfiability. In *ECAI*, volume 16, 682.

Rintanen, J. 2013. Planning as satisfiability: state of the art. https://users.aalto.fi/rintanj1/satplan.html.

Robinson, N.; Gretton, C.; Pham, D. N.; and Sattar, A. 2009. SAT-Based parallel planning using a split representation of actions. In *Nineteenth International Conference on Automated Planning and Scheduling, ICAPS 2009*. AAAI Press.

Seipp, J., and Helmert, M. 2013. Counterexample-guided cartesian abstraction refinement. In *Twenty-Third International Conference on Automated Planning and Scheduling*.