Karlsruhe Institute of Technology

PASAR
Planning as Satisfiability with Abstraction Refinement

Master Thesis

Karlsruhe Institute of Technology
Department of Informatics
Institute of Theoretical Informatics,
Algorithmics II

Nils Froleyks

January 30, 2020

Supervisors:	Prof. Dr. Peter Sanders
	Dr. Tomáš Balyo
	M.Sc. Dominik Schreiber

Abstract

We present a new algorithm for the satisficing (non-optimal) classical planning problem. The presented planner participated in the Sparkle Planning Challenge 2019 and won a considerable share (22.13%) of the first prize.

We combine SAT-based planning with forward state space search using the principles of counterexample-guided abstraction refinement (CEGAR). A state-of-the-art SAT solver is used to solve an abstraction of the planning task at hand. As an abstraction, we use an incomplete encoding where interference between actions is ignored. With a graph-theoretic test we determine whether the solution found by the SAT solver can be directly transformed into a plan. If it can be transformed, we realize an encoding allowing the application of many actions in parallel (exist-step semantics) in a very compact manner, without imposing a fixed order on the actions. If the transformation is not possible, we use the solution to define a heuristic function that is used during a state space search. If the search fails to find a plan within a very short timeout, the abstraction is refined.

To increase the synergy between SAT solving and forward search, both can learn new actions and add them to the problem. This allows the SAT solver to make big jumps in the search space, while the forward search benefits from the SAT solver's ability to solve combinatorially hard problems.

Using benchmark domains from recent international planning competitions, we compare our approach with various state-of-the-art planners from the fields of SAT-based planning and heuristic search. On almost all tested domains we can match the performance of the best tested solvers and on some domains we outperform the entire competition.

Zusammenfassung

PASAR ist ein neuer Algorithmus für das nicht optimale klassische Planungsproblem. Bei der Sparkle Planning Challenge 2019 hat PASAR einen nennenswerten Anteil (22.13%) des ersten Platzes erlangt.

Durch gegenbeispielgetriebene Abstraktionsverfeinerung werden Aussagenlogik basierte Planung und Vorwärtssuche im Zustandsraum kombiniert. Ein SAT-Solver wird eingesetzt um eine Abstraktion des Planungsproblems zu lösen. Als Abstraktion wird eine unvollständige Kodierung verwendet, bei der Konflikte zwischen Aktionen zunächst nicht beachtet werden. Durch einen graphentheoretischen Test kann bestimmt werden, ob die Lösung, die der SAT-Solver gefunden hat, direkt in einen Plan umgewandelt werden kann. Wenn dieser Schritt erfolgreich ist, konnte $\exists$-step Semantik mit einer sehr kompakten Kodierung umgesetzt werden, ohne den Aktionen dabei eine festgelegte Reihenfolge aufzuerlegen. Wenn nicht verwenden wir die Lösung dazu eine heuristische Funktion zu definieren. Diese wird während einer anschließenden Zustandsraumsuche verwendet, um den voraussichtlichen Nutzen einer Aktion zu bestimmen. Hat die Suche innerhalb eines sehr geringen Zeitrahmens keinen Plan gefunden, wird die Abstraktion verfeinert.

Um die Zusammenarbeit der Suche und des SAT-Solvers zu stärken, können beide neue Aktionen lernen und dem Planungsproblem hinzufügen. Dem SAT-Solver wird es dadurch möglich große Sprünge im Suchraum zu machen, während die Zustandsraumsuche von der Fähigkeit des SAT-Solver profitiert, kombinatorisch schwere Probleme zu lösen.

Wir vergleichen PASAR mit anderen aktuellen *Planern*, sowohl SAT basierte als auch heuristische Suchen. Der Vergleich zeigt, dass PASAR auf fast allen Testdomänen die Leistung der besten getesteten Planer erreicht und sie in manchen Fällen übertrifft.

Acknowledgments

First, I would like to thank Prof. Peter Sanders for providing the funds that enabled me to visit the Symposium on Combinatorial Search in the years 2017 and 2019 to present my work. In addition, he provided the computing power that made the evaluation of our work possible.

Together with my supervisors Dr. Tomáš Balyo and Dominik Schreiber, I went through a strenuous but very productive research phase that led to our publication [FBS19] at SoCS 2019. Together we developed interesting and successful solutions to the challenges we encountered during the design of the planner presented in this thesis.

I would also like to thank Marvin Williams, who worked on a similar topic during this time. Way more than a rubber duck; our exchanges on topics ranging from the finer points of SAT encodings to efficient C++ helped me a lot.

Teile dieser Arbeit wurde bereits auf der SoCS 2019 veröffentlicht [FBS19].

Karlsruhe, den 30. Januar 2020

Nils Froleyks

Contents

1 Introduction

The construction of technical systems that can perform tasks autonomously has been a popular field of research for a long time. Difficulties arise when the tasks to be performed or the environments in which they must be carried out are too diverse to anticipate all eventualities. These environments can be physical or purely virtual. Automated planning – the problem of finding a sequence of actions that can be performed to achieve a given goal – is one aspect of designing such systems. The problem of automated planning is very generic and universal, therefore a broad range of application domains are interested in the development of automated planners, including robotics for industrial applications [RP12], spacecraft control [Fuk+97] and transportation & logistics [Flo+13].

Despite the research that has been conducted since the seventies, computing power is still a limiting factor in many applications. Current research focuses on finding ways to make the planning process as efficient as possible.

Newly developed planners are evaluated in competitions such as the International Planning Competition (`IPC`) or the Sparkle Planning Challenge. The latter aims to create the best possible portfolio of current planners each year. An earlier version of the planner presented in this thesis participated in 2019 and contributed 22.13% to the final portfolio.[1] [2]

The most common approach to planning is to define a graph that represents all states the environment can be in, and which connects two states when the actor can perform an action to transition between them. Then some form of heuristic search is used to find a path to a state that fulfills the goal. If a path is found, it can be interpreted as a plan for the original task.

An alternative approach is to use Boolean variables and the basic logical operators: $\vee, \wedge, \neg$ to encode the planning problem. Then, one of the highly efficient SAT solvers developed in recent years is used to find a satisfying assignment to the variables. This assignment can be decoded into a plan to reach the goal.

The performance of these general approaches is highly dependent on the *planning domain* which they are applied to. While current sophisticated heuristic searches tend to solve more instances overall, there are a number of domains where SAT-based

[1] `http://ada.liacs.nl/events/sparkle-planning-19/results.html`

[2] Despite not supporting conditional effects and therefore failing on a third of the tested instances by default.

approaches dominate. This is one of the reasons for the popularity of portfolios in planning competitions. In this work we will go beyond a simple portfolio and combine the two approaches in such a way that they can benefit from each other's strengths.

Planning Domains The diversity in the application of planners is also reflected in the benchmark domains commonly used in planning competitions. A great variety of tasks has representative planning domains; from planning how a robot bartender should prepare cocktails and snacks for children with gluten intolerance, over choosing a route for a hike, to editing the human genome.

Throughout this thesis we will use a planning task from the *Trucking* domain as an example. In this domain, a fleet of trucks must deliver packages between cities.

Example 1. Trucking. In our example there are two trucks (T_1, T_2) and two packages (P_1, P_2). T_1 is at city A and T_2 at city B. The packages are also at city B. They must be delivered to city C. There are roads connecting cities (A, B) and (B, C). The example is illustrated in figure 1.1. The trucks can pickup packages, move between the cities and drop the packages again.

It would be easy enough to describe the road network as a graph and write a program to solve the problem. We add two restrictions: the trucks can only transport one package at a time, and we only have one-way streets connecting the cities. So T_2 for example may only move to city C. Now we would need to implement considerable changes to our program. If we want to solve problems with more cities and more trucks, we might need to find a faster algorithm and so on. Instead of writing a specialized program, we can encode our trucking problem as a *planning* task and then solve it with an automatic planner.

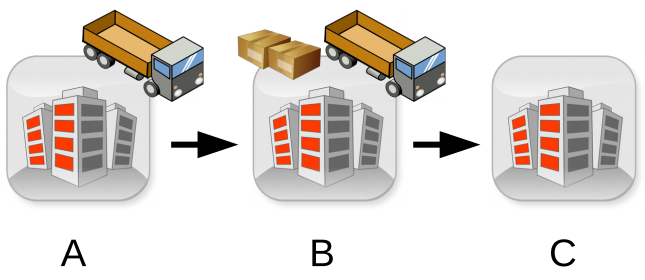


Figure 1.1: An Instance of the trucking problem.

1.1 Methods and Results

The planner we designed is a hybrid of a SAT-based planning approach and a forward state space search.

The problem of solving the planning task within a limited number of steps is encoded into a SAT formula. Interference between actions is not taken into account by the encoding. Therefore, any number of actions can be executed in the same step as long as each action is applicable. A SAT solver is used to solve the formula. If it succeeds within its time limit, we will proceed to the next step. If this is not the case, the formula for a higher number of steps is generated and the SAT solver is called again.

Since the used encoding is incomplete, the applied actions cannot necessarily be ordered into a sequential plan. Whether a step can be ordered is determined by a simple graph-theoretic test: for each step a graph of the actions is constructed and their interferences are inserted as edges. A topological ordering of the graph is a valid application order for the actions.

If all steps can be ordered, we found a valid plan. If not, a greedy best-first search (GBFS) is started to find a path from the initial state to a goal state. The heuristic function is based on the distance to the states which would be visited if the steps could be executed without interference. If no plan is found within a very short time limit, clauses are added to the formula to prevent the observed interferences from occurring again. The SAT solver is called again and the cycle repeats.

A number of improvements to this initial approach have been identified and implemented:

Refinement Strategy The clauses to prevent interferences are added to all steps. Therefore, the SAT solver might be forced to change a solution to a step that previously could be ordered. We investigate different ways to generate clauses to limit this effect.

Sparsification The SAT solver may use unnecessary actions to reach the goal. Those actions can cause interferences and thus prevent the ordering of a step. Another related problem is that the heuristic function used by the GBFS is *over-defined*: the heuristic is based on the distance to complete states, while only a small subset of the state is relevant for the plan. A *sparsification* routine is executed to reduce the number of actions wherever possible and to identify the relevant subsets of the states.

Same Makespan Limit The complexity of some planning tasks arises solely from interference between actions. Ignoring interference makes the planning task trivial. We recognize this and trigger a fallback mechanism that adds more interference to the encoding where necessary.

Interleaving Search For some planning tasks not a single formula can be solved. In these cases we essentially spend all of our time computing a heuristic function. These instances are not necessarily hard, on the contrary, many of them can be solved in a fraction of a second by a simple forward search. We have added the option to split the computation time between the SAT solver and a simple GBFS that uses the distance to the goal as heuristic until the first formula is solved.

With above improvements, our planner is very competitive. Thorough evaluation on the `IPC` domains of the last years shows that PASAR can compete with SAT-based planners (Madagascar [Rin14]) and heuristic searches (Fast Downward [Hel06]). On many domains it matches the performance of the best tested solver and on some domains it even outperforms its competition. Furthermore, the strength of PASAR seems to differ from current heuristic search algorithms. Therefore, it might be advantageous to include PASAR in a portfolio of state-of-the-art solvers.

1.2 Structure of the Thesis

In chapter 2 we introduce the classical planning problem and define our notation for planning tasks. In addition, we introduce basic planning algorithms and the concept of counterexample-guided abstraction refinement. In chapter 3 we give a brief overview of previous work on planning algorithms in general and the use of CEGAR in planning in particular. Afterwards we will introduce our planner PASAR in chapter 4. In chapter 5 we will evaluate the effect of different aspects of our algorithm and compare it with other well known planners. Finally, we conclude this work in chapter 6 and give some ideas for future work on the topic.

2 Preliminaries

This chapter introduces the problem of classical planning and defines our formalism for planning tasks. Planning as a state space search is introduced and the basics of SAT-based planning are explained. Finally, we give a short introduction to counterexample-guided abstraction refinement.

2.1 Classical Planning

There are different forms of the planning problem. We will deal with classical planning as described by Ghallab, Nau, and Traverso [GNT16] and presented below.

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 described by a set of finite-domain variables and is:

Fully-observable The value of every variable is known.

Deterministic The exact effects of executing an action are known.

Static The world only changes as a result of the actions in our plan.

We assume that actions are instantaneous, and we therefore only need to deal with their sequencing. Actions have preconditions that determine the states of the world in which they can be applied and effects that determine how the world is changed after an action is executed.

We use multi-valued planning tasks [Hel06] based on SAS+ [BN95], instead of the classical STRIPS formalism [FN71] using propositional logic. Note that we deviate slightly from multi-valued planning tasks by not distinguishing between *prevail*-conditions and preconditions. Besides that, we do not handle conditional effects at the moment.

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

- $X = \{x_1, \ldots, x_n\}$ is a finite set of state variables with finite domains $\mathrm{dom}(x_i)$ for $i = 1, \ldots, n$. A (partial) *state* is a set of assignments of the form $x_i = v$ where $x_i \in X$ and $v \in \mathrm{dom}(x_i)$ and each variable is assigned at most once. In a *complete* state each variable is assigned exactly one value. E is the set of all states and $S \subseteq E$ is the set of all complete states.
    - A state s *fulfills* a state p iff $p \subseteq s$.

- An assignment $x_i = v$ is called *inconsistent* with a state s iff $x_i = v' \in s$ and $v \neq v'$. Two states are called inconsistent if one of their assignments is inconsistent with the other state.
- Given two states $s, e \in E$, we can get a new state by *applying* e to s. The resulting state s' is the union of the state s and e, except for the assignments in s that are inconsistent with e.

$$s' = s \oplus e := s \cup e \setminus \{x_i = v \in s \mid x_i = v \text{ is inconsistent with } e\}$$

- $O \subseteq E \times E$ is a finite set of actions. Each action $a \in O$ is a pair of states $(\text{pre}(a), \text{eff}(a))$ where $\text{pre}(a)$ is the set of preconditions of a and $\text{eff}(a)$ is the set of effects of a.
- $s_I \in S$ is the (complete) *initial state.*
- $s_G \in E$ is a partial state and every state that fulfills s_G is a *goal state.*

This allows the execution of actions and plans to be defined as follows:

- An action a is *applicable* in state s if $\text{pre}(a)$ is fulfilled in s.
- $\text{execute}(a, s) := s \oplus \text{eff}(a)$ is defined if action a is applicable in state s.
- 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 $s_G \subseteq \text{execute}(a_k, \text{execute}(a_{k-1} \ldots \text{execute}(a_2, \text{execute}(a_1, s_I)) \ldots))$.

As we deviate from the classic STRIPS formalism, we show that the problems we are dealing with are still just as hard. More precisely, we prove that the word problem for the language $\textsc{PlanMin} = \{\langle \Pi, N\rangle \in \Psi^* \mid \text{A plan of length } N \text{ or less exists for the planning task } \Pi.\}$ is PSPACE-complete in Appendix A.

Example 2. Trucking as planning. To describe our trucking problem as a planning task, we first have to describe the world completely with variables.

- x^{T_1} and x^{T_2} are the locations of the trucks with $\text{dom}(x^{T_1}) = \text{dom}(x^{T_2}) = \{L_A, L_B, L_C\}$.
- x^{P_1} and x^{P_2} are the locations of the packages with $\text{dom}(x^{P_1}) = \text{dom}(x^{P_2}) = \{L_A, L_B, L_C, T_1, T_2\}$.
- x^{e_1} and x^{e_2} are Boolean variables that indicate whether the trucks are empty or not.

Additionally, we have the constant Boolean variables $x^{r_{(u,v)}}$ for $u, v \in \{A, B, C\}$ to describe the roads the trucks can drive through. Constant means that they cannot change their value during the planning process (they are not affected by any action).

We can use the variables to describe the initial state
$s_I = \{x^{T_1} = L_A, x^{T_2} = L_B, x^{P_1} = L_B, x^{P_2} = L_B, x^{e_1} = \texttt{true}, x^{e_2} = \texttt{true}\} \cup R$.

In R all *road*-variables are set to `false` except $x^{r^{(A,B)}}$ and $x^{r^{(B,C)}}$ which are set to `true`.

Note that every variable is assigned a value and s_I is therefore a complete state. The goal is described by the partial state $s_G = \{x^{P_1} = L_C, x^{P_2} = L_C\}$.

O consists of three kinds of actions. For each $i, j \in \{1, 2\}$ and $\ell, \ell' \in \{A, B, C\}$ we have:

- pickup(T_i, P_j, L_ℓ) $= (\{x^{T_i} = L_\ell, x^{P_j} = L_\ell, x^{e_i} = \texttt{true}\},$ $\{x^{P_j} = T_i, x^{e_i} = \texttt{false}\})$
- move$(T_i, L_\ell, L_{\ell'})$ $= (\{x^{T_i} = L_\ell, x^{r(\ell,\ell')} = \texttt{true}\},$ $\{x^{T_i} = L_{\ell'}\})$
- drop(T_i, P_j, L_ℓ) $= (\{x^{T_i} = L_\ell, x^{P_j} = T_i\},$ $\{x^{P_j} = L_\ell, x^{e_i} = \texttt{true}\})$

A possible plan for this planning task consists of the following seven actions:
$P = \langle$ pickup(T_2, P_1, B), move(T_2, B, C), drop(T_2, P_1, C), move(T_1, A, B),
pickup(T_1, P_2, B), move(T_1, B, C), drop(T_1, P_2, C) $\rangle$

2.2 Planning as State Space Search

The simplest approach to planning is a forward state space search. We can represent the state space of a planning task as a directed graph $G = (V, E)$. The nodes are all complete states ($V = S$) and we have an edge from state u to state v, if an action can be executed in u that leads to v: $E = \{(u, v) \mid u, v \in V,\ \exists a \in O : \text{pre}(a) \in u \text{ and } u \oplus \text{eff}(a) = v\}$.

The edges are labeled with the corresponding action. A path from the initial state to a goal state can be interpreted as a plan. Note that multiple actions can connect two nodes.

Although easy to define, the graph can be huge even for a simple planning problem. Consider trucking example 2 again. If we were dealing with 15 cities and 10 trucks, each delivering 5 packages, the size of S would already exceed the highest estimates for the number of particles in the observable universe.[1]

In addition to the large search space and the possibly long plans, we may have to deal with high branching factors. Depending on the domain, some instances from the *IPC*-set may exceed a branching factor of 18 000 , i.e. up to 18 thousand actions may be applicable in a single state.

A number of different graph search algorithms have been explored for planning. For a detailed introduction we refer to the work of Ghallab, Nau, and Traverso [GNT16]. For non-optimal planning, greedy best-first search (GBFS) is the most common approach [GNT16]. GBFS is a depth-first search, which explores the best successor of the current node according to some *heuristic function* next. Pseudo-code for a GBFS in the context of planning and a discussion of some important implementation details are given in section 5.1.2.

[1]$\sim 15^{10} \times (15 + 10)^{50} \times 2^{10} > 10^{80}$ (`https://www.popularmechanics.com/space/a27259`)

2.3 Planning as Satisfiability

Besides state space search, SAT-based planning algorithms have proven to be very successful. In the following we will introduce the basics.

2.3.1 Boolean Satisfiability Problem

The Boolean Satisfiability Problem (SAT) is the problem of deciding whether a given Boolean formula can be satisfied with a truth assignment to the variables. It was the first problem to be proven to be NP-complete by Cook [Coo71] and Levin.

A Boolean variable can take one of two values: `true` or `false`. A *literal* is a Boolean variable a or its negation: $\overline{a}$. A *clause* is the disjunction (`OR`) of a finite number of literals, e.g., $(a \lor \overline{b} \lor c)$. A Boolean formula (in conjunctive normal form) is the conjunction (`AND`) of a finite number of clauses, e.g., $(a \lor \overline{b} \lor c) \land (\overline{a} \lor c)$.

Given an assignment of truth values to the variables; a formula is satisfied if all of its clauses are satisfied. A clause is satisfied if at least one of its literals is satisfied. A positive literal (a) is satisfied if the value of its variable is `true` and a negative literal is satisfied if the value is `false`. A satisfying assignment is called a *model* of the formula.

SAT solvers determine the satisfiability of a formula and return a model if possible.

2.3.2 Incremental SAT Solving

In a lot of applications (including planning), a SAT solver will solve not just a single formula, but a sequence of incrementally generated SAT formulas.

The idea of *incremental* SAT solving is to add the ability to expand a formula that the SAT solver has already tried to solve. The advantage over just solving a new formula every time is that the SAT solver can reuse what has already been learned in previous runs.

In incremental SAT solving, clauses can be added to a formula but never removed. For this reason *assumptions* are added. A set of assumptions A consists only of literals. When the SAT solver is called, it tries to find a model that satisfies the formula, while all assumptions in A hold. Unlike clauses, assumptions can be added and removed at will. Note that we can only assume literals, not clauses. However, this is no limitation. If we want to add a clause C that we plan to remove later, we add it to the formula with a new *toggle* variable: $(T \lor C)$. Additionally, we add the literal $\overline{T}$ to the assumptions. If we want to remove the clause C, we simply remove the literal $\overline{T}$ from the set of assumptions. The SAT solver can satisfy the clause by assigning `true` to the variable T.

2.3.3 Encoding Planning as SAT

The basic idea of solving planning as SAT [KS92] is to encode the planning problem Π up to a certain number of *steps* N as a Boolean formula F_i in such a way that F_i is satisfiable if and only if there is a plan with i *steps* or less. Additionally, a valid plan must be constructible from a model of F_i.

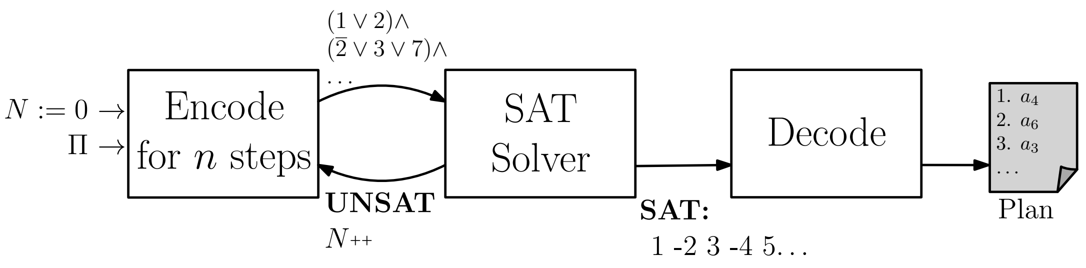


Figure 2.1: Planning as Satisfiability.

In a standard SAT encoding we have state and action variables for each timestep $t = 1, \dots, N$. For each action $a_i \in O$ a Boolean variable a_i^t is introduced, which indicates the execution of a_i in timestep t. For each state variable $x_i \in X$ and each value $v_j \in \text{dom}(x_i)$ we introduce $s_{i,j}^t$. In each timestep one of the $s_{i,j}^t$ will be set per variable to encode its value (*one-hot encoding*). To reduce the number of variables needed, we only introduce one variable s_i^t if $|\text{dom}(x_i)| = 2$.

The following constraints are encoded:

(i) The values of state variables in the first step are consistent with the initial state.

$$\forall x_i \in X : \bigwedge_{v_j \in \text{dom}(x_i)} \begin{cases} s_{i,j}^0 & \text{if } x_i = v_j \in s_I \\ \overline{s_{i,j}^0} & \text{otherwise} \end{cases}$$

(ii) The values of the state variables in the last step are consistent with the goal conditions. If an incremental SAT solver is used, the literals are added to the assumptions, not the formula, so that they can be removed if the makespan is increased.

$$\forall x_i = v_j \in s_G : \quad s_{i,j}^N$$

(iii) In each step t each state variable assumes exactly one value from its respective domain. For variables with domain size two, the clauses are not necessary.

$$\forall x_i \in X : \bigvee\nolimits_{v_j \in \text{dom}(x_i)} \quad s_{i,j}^t$$
$$\forall x_i \in X : \bigwedge\nolimits_{v_j, v_k \in \text{dom}(x_i)} \quad \overline{s_{i,j}^t} \vee \overline{s_{i,k}^t}$$

(iv) When an action is executed in step t, its preconditions are satisfied in step t and its effects are fulfilled in step $t+1$.

$$\begin{aligned}\forall a_k \in O, \forall x_i &= v_j \in \operatorname{pre}(a) &: a_k^t \Rightarrow s_{i,j}^t \\ \forall a_k \in O, \forall x_i &= v_j \in \operatorname{eff}(a) &: a_k^t \Rightarrow s_{i,j}^{t+1}\end{aligned}$$

(v) If the value of a state variable changes between steps t and $t+1$, then there must be an action in step t causing this change by one of its effects (frame axioms). We introduce the *value support* for this: the value support $\mathrm{V_{Supp}}$ maps each assignment to the set of actions that have this assignment as an effect: $a \in \mathrm{V_{Supp}}(v = x) \Leftrightarrow (v = x) \in \operatorname{eff}(a)$.

$$\forall x_i \in X, \forall v_j \in \operatorname{dom}(x_i) : s_{i,j}^t \Rightarrow s_{i,j}^{t+1} \bigvee_{a_k \in \mathrm{V_{Supp}}(x_i = v_j)} a_k^t$$

(vi) At most one action is executed in each step t.

$$\bigwedge_{\substack{a_i, a_j \in O, \\ i \neq j}} \overline{a_i^t} \vee \overline{a_j^t}$$

Essentially we are solving PLANMIN instances encoded as SAT until we find one that is satisfiable. Note that we create variables and clauses for each $t \in [1, \ldots, N]$. In the PLANMIN instance N is encoded logarithmically, so our SAT encoding is exponential. Since SAT is in NP and PLANMIN is PSPACE-hard, we cannot expect to do better. An encoding that could be computed in polynomial time would imply PSPACE = NP.

The presented encoding is called *sequential* since only one action may be executed in each step and the actions are therefore in a fixed order. We can relax constraint (vi) to allow multiple actions to be executed in *parallel*.

2.3.4 Parallel Plans

A parallel plan is a sequence of action sets, one for each step. It can be advantageous to allow the execution of several actions in one step, since longer plans can be found by solving smaller formulas. A parallel plan is only used to represent one or more sequential plans and does not suggest that actions can actually be physically executed in parallel. The length of a parallel plan is called *makespan*.

We must ensure that the actions executed in each step can be ordered into at least one sequential plan. If the clauses encoding constraint (vi) are simply omitted, this is no longer the case. A satisfiable formula does not even imply the existence of a plan. Consider the planning problem described in example 2. Encoding it using only the constraints (i) to (v) for two steps yields a satisfiable formula. By decoding the model, we obtain the actions performed in each step:

1 {`pickup`(T_2, P_1, L_B), `pickup`(T_2, P_2, L_B), `move`(T_2, L_B, L_C)}

2 {`drop`(T_2, P_1, L_C), `drop`(T_2, P_2, L_C)}

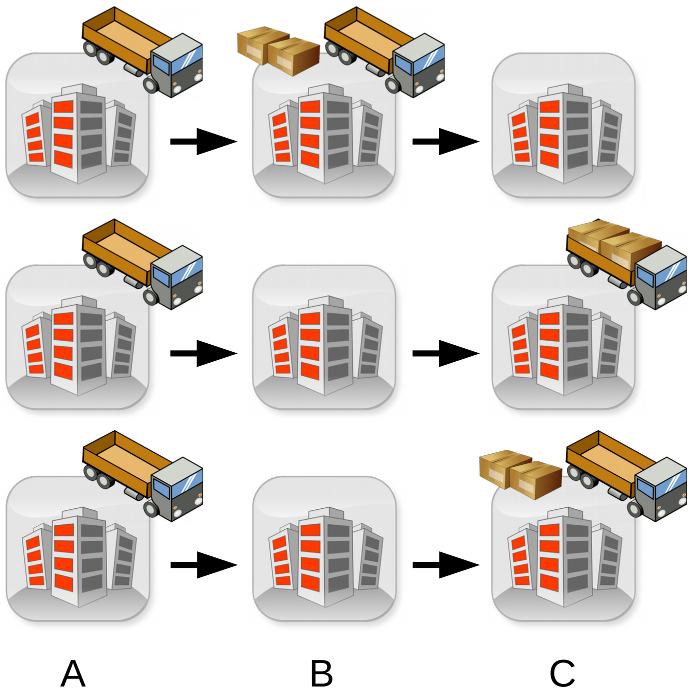


Figure 2.2: An invalid parallel plan: a truck can only load one package at a time.

The state after executing the first step cannot be reached (see figure 2.2), because there are several packages in the same truck. The two `pickup` actions are applicable in the initial state, however, there is no way to arrange the actions into a sequential plan without one destroying the preconditions of the other.

We introduce two semantics for parallel plans based on [RHN06]: a parallel plan satisfies the ∀-*step* semantics if *every* possible ordering of the actions within each step results in a valid plan.

We can still achieve higher parallelism. Consider step 1 from the invalid plan example but with only one of the `pickup` actions: only one of the two possible orderings results in a valid plan, because `pickup`(T, P, L_B) requires the location of the truck to be L_B and `move`(T, L_B, L_C) changes the location to L_C.

The even more general ∃-*step* semantics also allow parallel execution of actions as long as there is *any* ordering of action within each step that leads to a valid plan.

2.3.5 Makespan Scheduling

Up to this point, we have suggested that in order to find a plan, one checks all formulas $F_0, F_1, F_2, \dots$ until a formula can be solved. This is called *sequential scheduling* and is indeed very common: "for instance the GraphPlan algorithm [Blum and Furst 1997] and planning as satisfiability [Kautz and Selman 1996] and related approaches [Rintanen 1998; Kautz and Walser 1999; Wolfman and Weld 1999; van Beek and Chen 1999; Do and Kambhampati 2001], have without exception adopted a sequential strategy for plan search" – Rintanen, Heljanko, and Niemelä [RHN06].

Sequential scheduling always finds the plan with the minimum makespan. However, this generally does not correspond to any practically interesting optimality criterion, especially in the case of parallel plans.[2] When it comes to finding any plan, we want to select a makespan for which the formula can be solved in the shortest possible time. An important observation in this context is that "the satisfiability tests for the last unsatisfiable formulae are often much more expensive than for the first satisfiable formulae" [RHN06]. The makespan for the first satisfiable formula is not a function of the problem size, but varies greatly between different domains and cannot be calculated efficiently. However, selecting a makespan that is too high will result in a large formula and increase the probability that the SAT solver gets lost in a part of the search space that does not contain a solution. Different scheduling algorithms that decide how much time the SAT solver spends on each formula have been studied. See Rintanen, Heljanko, and Niemelä [RHN06] for more details.

2.4 Counterexample-Guided Abstraction Refinement

Counterexample-guided abstraction refinement (CEGAR) was first proposed by Clarke et al. [Cla+00] for the purpose of model checking, i.e. to check whether a model of a system meets a given specification. The idea is to start with a coarse abstraction of the model and refine it iteratively only where necessary to combat the state explosion problem for larger applications.

In the context of model checking, CEGAR starts by automatically generating an initial abstraction that has the following property: the abstraction fulfilling the specification implies that the original model also meets the specification.

If the abstraction does not match the specification, an *abstract counterexample* is generated. The next step is to check whether the abstract counterexample corresponds to at least one actual counterexample. If this is the case, the counterexample is returned showing that the model does not meet the specification. If no actual counterexample

[2]This is the opinion of the author. In the `IPC-4` and `IPC-5`, a plan was considered optimal if it fulfilled $\forall$-step semantics with a minimal makespan.

can be identified, the abstract counterexample is called *spurious*. In this case the abstraction is refined to eliminate the spurious counterexample.

As both automated planning and model checking have a combinatorial search space, CEGAR has also been adopted in planning approaches. To avoid confusion, we will explain some technical terms we use when talking about CEGAR in connection with planning. When using model checking e.g. for the application of software verification, the goal is to find a *counterexample* e.g. a sequence of instructions that leads to an index out-of-bounds exception, or to prove that no such sequence exists. For planning, the equivalent of a counterexample is a *plan* i.e. a sequence of actions leading to a goal state. Proving that there is no plan may be interesting but is rarely the focus of building a planner. Instead of abstract counterexamples, we will talk about abstract plans and use the term counterexample only in the context of refining an abstraction.

3 Related Work

This chapter presents an overview of both forward search and SAT-based planners and an attempt to combine them. In addition, we discuss the previous use of CEGAR in connection with planning.

Heuristic Search Currently the most successful planners use some kind of heuristic search. An early example of such a planner is *Fast-Forward* [Hof01]. Fast-Forward introduced the *relaxed graphplan heuristic* (delete relaxation). This heuristic estimates the cost to reach the goal from a visited state by constructing a relaxed plan from the visited state to the goal. In the relaxed plan, the negative or delete effects of actions are ignored.[1] The performance of this heuristic and thus also of Fast-Forward depends strongly on the interaction of the subgoals and thus the planning domain.

One of the more recent and very successful planners is *Fast Downward* [Hel06]. Variations of Fast Downward performed very well in recent planning competitions, winning the Satisficing and Cost-bounded track of the IPC 2018 [Sei18] and dominating the leaderboard (depending on the test set[2]) in the Sparkle Planning Challenge 2019 [Sei19].

The success of the Fast Downward planning system is the result of several factors. Basic operations are implemented very efficiently, such as the *successor generation*, which allow a forward search to quickly generate the set of applicable actions.

But most important for the performance of Fast Downward is the *causal graph heuristic*. To compute it efficiently, the critical information of the planning task is compiled into *domain transition graphs* and a *causal graph* in a preprocessing or *knowledge compilation* step. The computation of the heuristic is strongly connected to *hierarchy*, which is often encountered in planning problems. Indeed, the computation requires that the causal graph is acyclic, which corresponds to a strict hierarchical structure in the planning task. The computations mentioned all rely on first translating a planning task into a multi-valued representation, which makes many of the implicit constraints of a propositional planning task explicit.

[1]The equivalent in our formalism would be that variables can assume multiple values at once and that they retain all the values assigned to them.

[2]`http://ada.liacs.nl/events/sparkle-planning-19/leaderboard.html`

SAT-based Planning SAT-based planning follows a fundamentally different approach than heuristic search-based planning. Kautz and Selman [KS92] introduced the idea of encoding a planning task into a sequence of Boolean formulas and then solving it with a SAT solver. Since then a lot of work has been put into improving this method. For example, many increasingly compact and efficient encodings have been developed: Kautz and Selman [KS92] originally proposed a sequential encoding. This was replaced by parallel plans without action interference, later formalized as $\forall$-*step* semantics [RHN06]. For the fourth and fifth international planning competition, the objective of the newly introduced optimal track was to find parallel plans that meet $\forall$-*step* semantics and have a minimal number of steps.

Rintanen, Heljanko, and Niemelä [RHN06] proposed to relax the constraints on parallelism and introduced $\exists$-*step* semantics to reduce the number of SAT solver calls necessary to find a plan. $\exists$-*step* semantics allow the parallel execution of actions if they *can* be ordered into a sequential plan. Wehrle and Rintanen [WR07b] later expanded on that idea by relaxing $\exists$-*step* semantics, allowing actions to be executed in a step where they are initially not applicable. Balyo [Bal13] further relaxed the constraints by allowing effects of the actions applied in one step to cancel each other out.

Another approach to encoding a planning task builds on the work of Blum and Furst [BF97]. They introduced planning graphs in which both assignments and actions are represented by nodes in alternating layers. Kautz and Selman [KS99] were the first to combine the potential of SAT solvers with graph-based planning in their solver *Blackbox*. It constructs the planning graph for a number of layers, encodes it into a Boolean formula and uses a SAT solver to solve it. If the SAT solver dismisses the formula as unsolvable, the number of layers in the planning graph is increased.

Besides enhanced encodings, different SAT solver schedules [RHN06] and using incremental SAT solving [GB17] had an impact on the performance of SAT-based planning.

Besides his theoretical work, Rintanen also implemented *Madagascar* [Rin14]. This SAT-based planner uses a non-incremental SAT solver that was specifically written for this application. On the one hand, it has the advantage of using a heuristic that is specialized for planning [Rin12] and other specializations. On the other hand it lacks some important features that have been developed for SAT solvers in recent years.

Modern SAT solvers are based on the method of Conflict-Driven Clause Learning (CDCL). For a good introduction we refer to Biere et al. [Bie+09]. The core principle is to use a specialised heuristic to choose a variable and assign a truth value to it. This assignment is propagated through all clauses. The process is repeated until either a model or a *conflict* is found. Learned conflicts are remembered for the rest of the solving procedure. The popular SAT solvers *Glucose* [AS09], *MiniSAT* [ES03], Lingeling [Bie17] and *PicoSAT* [Bie08] are all based on CDCL. The internal SAT solver Madagascar uses is also based on CDCL.

Unifying SAT-based planning and forward search The only planner that explicitly combines SAT-based planning and forward search is *CO-PLAN* [RGP08]. It deals with optimal planning and was designed after the sixth international planning competition introduced action costs for the optimal track. It attempts to build on the success of the descendants of Blackbox, that dominated the optimal track in previous competitions.

CO-PLAN starts in the same way Blackbox does: it constructs the planning graph, encodes it into a Boolean formula and solves it with a SAT solver to get a step-optimal parallel plan. It then uses the cost of this plan to bound a complete forward search in the state space.

CEGAR in planning Counterexample-guided abstraction refinement (CEGAR) was originally introduced by Clarke et al. [Cla+00] for the purpose of model checking. It has been applied to optimal classical planning by Seipp [Sei12]. They proposed an online heuristic that uses CEGAR to improve its accuracy during search.

The paper "Counterexample-guided Planning" by Chatterjee et al. [Cha+05] used CEGAR for planning in adversarial and probabilistic environments with perfect information. This model differs form classical planning both in the problems that can be described and the approaches to solving it.

To the best of our knowledge, we are the first to introduce the CEGAR paradigm to SAT-based planning.

4 Our Planner: PASAR

PASAR is a hybrid between a SAT-based planning algorithm and a forward state space search. We try to combine the strength of both approaches. A graph-based search can quickly explore large parts of the state space and is able to find very long plans. Whereas a sophisticated SAT solver can far surpass it in solving combinatorially hard problems that are smaller in size.

We use a SAT solver to find a model for an *abstraction* of the planning task. Such a solution cannot necessarily be decoded into a plan. Instead, we use it to define a heuristic function that is used during a state space search. If the search does not find a plan, we use the information collected to refine our abstraction.

To increase synergy, both the search and the SAT solver can learn new actions and add them to the problem.

The outline of the algorithm is depicted in figure 4.1. In the following we will discuss the different components.

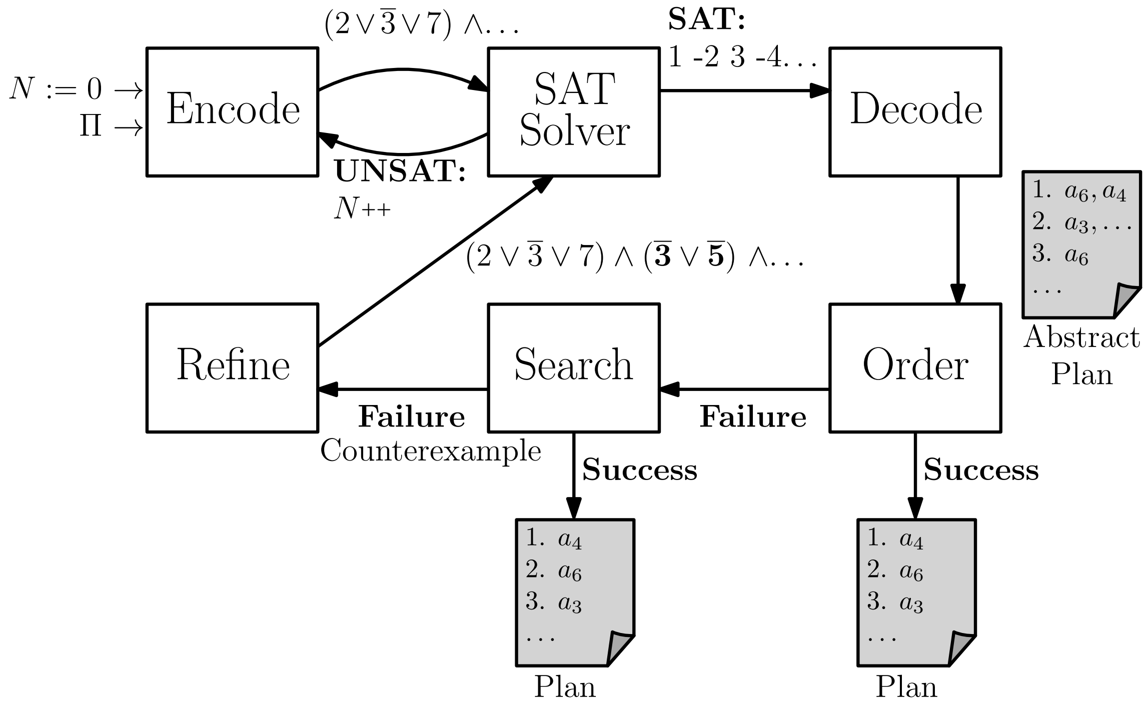


Figure 4.1: PASAR.

4.1 Encoding

To obtain the encoding for our abstraction, we make two changes to the standard encoding presented in the introduction. First, we omit the *at-most-one*-constraint (vi) for the actions.

Therefore, any set of actions may be executed in one step as long as their preconditions hold and the effects are fulfilled in the next state. As we explained in section 2.3.4, a model of the SAT formula cannot be decoded into a valid plan in all cases. The encoding is therefore incomplete.

Second, we want to use incremental SAT solving to allow the SAT solver to use actions learned during the planning process. For new actions, we will need to add clauses to ensure that their preconditions are fulfilled and that their effects hold (iv) – we can simply add these. Additionally, we have to modify the *frame axioms* (v). Otherwise, new actions could not change the value of a variable because they were not part of the originally computed *value support* for that assignment. To make the frame-clauses for each assignment $x_i = v_j$ removable, we add a new variable $T^0_{i,j}$ during initial creation. We use the same variable $T^0_{i,j}$ for each timestep t. The literal $\overline{T^0_{i,j}}$ is added to the set of assumptions A that are activated before starting the SAT solver.

$$s^t_{i,j} \Rightarrow s^{t+1}_{i,j} \bigvee_{a_k \in \mathrm{V_{Supp}}(x_i = v_j)} a^t_k \vee T^0_{i,j}$$

If the value support for an assignment $x_i = v_j$ is updated, we remove the literal $\overline{T^k_{i,j}}$ for the highest possible k from A and add the literal $T^k_{i,j}$. Then the frame-clause with the new value support is added, using a new variable $T^{k+1}_{i,j}$. The literal $\overline{T^{k+1}_{i,j}}$ is added to A to activate the clause.

Many abstractions can be used in our planner as long as they can be refined by simply adding additional clauses. We will focus on the clauses dealing with action interference, as they generally contribute most to the size of the formula – even when a linear size encoding is used. We will realize ∃-step semantics by the way in which we *order* abstract plans. Furthermore, we will do so without imposing a fixed order on the actions, as is common when encoding ∃-step semantics. Rintanen, Heljanko, and Niemelä [RHN06] presented two encodings without a fixed order. However, they deemed them impractical because of their size and did not present experimental results for them.

4.2 SAT Solver

We use IPASIR, a generic interface for incremental SAT solving [Bal+16]. A number of popular SAT solvers support IPASIR; we tested *Glucose* [AS09], *MiniSAT* [ES03], Lingeling [Bie17] and *PicoSAT* [Bie08].

Makespan scheduling We discussed makespan scheduling before in section 2.3.5. The same principles apply even if we only solve an abstraction. We use a constant timeout for each formula and increase our makespan N exponentially similar to Rintanen [Rin14]: $N := \max(N + 1, \gamma N)$, where $\gamma > 1$ is a constant that controls the exponential makespan-increase.

Instead of a timeout, it is also possible to limit the number of conflicts (CDCL[1]) that may occur. This is not supported in the IPASIR interface and can currently only be used with *Glucose* and *MiniSAT*. For testing purposes this option is preferred, because a timeout based on wall time can lead to a higher variance in runtimes.

After some (very high) makespan is reached, we use an infinite timeout. If the SAT solver is complete, this guarantees the completeness of PASAR, since the abstraction eventually approaches a standard SAT encoding.

4.3 Decoding SAT Models

Unlike traditional SAT-based planning, a model for a formula cannot be decoded into an actual plan. Instead, it is decoded into an *abstract plan* consisting of a sequence of *states* and intermediate *action sets*. See figure 4.2 for an illustration.

The action sets will be *ordered* into a sequential plan where possible and the states from the abstract plan will be used to define a heuristic for the *search* phase.

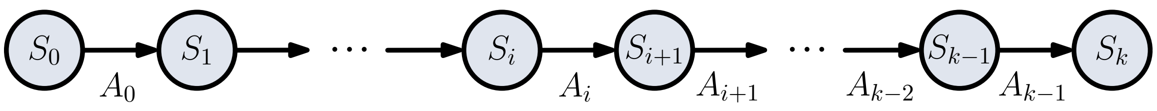


Figure 4.2: An abstract plan. S_0 is the initial state and S_k is a goal state. Each action in A_i is applicable in state S_i and executing all actions in A_i results in the state S_{i+1}.

Sparsification This step is not strictly necessary, but can be advantageous as a preprocessing step for the following phases of our algorithm.

The states in our abstract plan are complete. We want to identify the subsets of the states that are actually relevant for reaching the goal. Additionally, a SAT solver may be inclined to set a lot of action variables to `true` to satisfy the formula. Therefore, an abstract plan may contain a lot of actions that are not necessary to reach the goal.

[1]Conflict-Driven Clause Learning is a technique used in SAT solvers. For an introduction to the topic we refer to Biere et al. [Bie+09]. For our purposes, it is sufficient to understand the number of conflicts as a metric similar to explored nodes in a graph search.

```
Procedure Sparsification($\Pi = (X, O, s_I, s_G), P = \langle S_0, A_0, \ldots, A_{k-1}, S_k \rangle$)
    $S_k = s_G$
    for $i = (k-1), \ldots, 0$ do
        $r = S_{i+1}$
        for $a \in a_i$ do
            if $\text{eff}(a) \cap r \neq \emptyset$ then
                $r = r \setminus \text{eff}(a)$
                $S_i = S_i \cup \text{pre}(a)$
            else
                $A_i = A_i \setminus a$
            end
        end
        $S_i = S_i \cup r$
    end
```

Algorithm 1: Abstract plan sparsification routine.

A *sparsification* routine is executed to identify the relevant subsets of the states and action sets. Algorithm 1 is illustrated in figure 4.3.

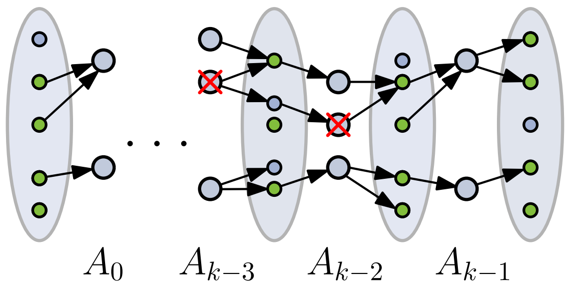


Figure 4.3: Sparsification of an abstract plan: The states are represented as ellipses. The relevant assignments are colored green. The actions between the states are pointed at by their precondition and point at their effects. We iterate over the actions from top to bottom.

Starting with the final state S_k, we move backwards along the abstract plan and keep only the assignments and actions that are relevant for the plan. For the final state s_k the relevant assignments are s_G. We then iterate over the actions leading to this state and consider their effects: if the action has a relevant effect, we keep the action and remove its effects from the set of relevant assignments `r` that have not yet been fulfilled. The subset of relevant assignments for the next (towards the initial) state consists of the relevant assignments that could not be fulfilled in this step and the preconditions of the actions we retained. On a conceptual level, this procedure is similar to the backwards search that is used to extract a plan from a planning graph [BF97].

The result of the sparsification depends on the order in which we iterate over the action sets. To get a better result with fewer actions in the action sets or fewer assignments in the states, a problem similar to finding a *hitting set* can be solved here. We have left this to future work.

It is not entirely obvious that the relevant subsets of states we assign in line 13 are actually subsets of the state. For the preconditions we add, we can rely on the clauses in our encoding that enforce the fulfillment of preconditions. For an assignment in the set of unfulfilled preconditions `r` we know that it is necessary either for the goal or for the application of a following action and that no subsequent action has it as an effect; thus, it must already be part of the state because the frame axioms are encoded.

Step Reduction After the sparsification, we check if each set of actions A_i is necessary to reach the goal. More specifically, we try to remove A_i and s_{i+1} from the abstract plan and execute the remaining plan. All subsequent steps where an action in an action set is no longer applicable are removed in the same way. If all goals are still satisfied in the end, the shortened abstract plan is kept and we try to remove the next step. Conceptually, we contract the action sets into single actions and then execute a greedy action elimination algorithm [NM10].

4.4 Ordering Action Sets

The action sets in the abstract plan already constitute a valid parallel plan, if and only if the action set in each step can be ordered into a sequential plan to get from one (partial) state to the next. Checking whether an action set can be ordered can be done efficiently with a simple graph-theoretic test: an action set can be ordered if and only if the *disabling graph* for the actions is cycle-free. The disabling graph[2] G for a set of actions A is defined as:

$$G = (A, \{(a_1, a_2) \in A^2 \mid a_1 \neq a_2, \mathrm{pre}(a_1) \text{ is inconsistent with } \mathrm{eff}(a_2)\})$$

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. Actions that induce a cycle in the disabling graph are called *interfering*. If an action set contains no interfering actions, any topological ordering is a valid plan to get from one step to the next.

After an abstract plan is found, we construct the disabling graph for the action set in each step and try to find a topological ordering by using a DFS that tracks the

[2]This definition differs slightly from the definition introduced by Rintanen, Heljanko, and Niemelä [RHN06]. First, this work does not deal with conditional effects and disabling graphs are only used for actions that are applicable in the same state. Second, the edges are inverted because we came up with the idea independently and the authors do not want to get confused while writing.

departure time of the nodes [Tar76]. If all steps can be ordered, ∃-step semantics are fulfilled and we can order the action sets into a sequential plan.

Learning actions Even if not all action sets can be ordered, we can still benefit from those that can. If we succeed in ordering an action set, the action sequence is contracted into a single action and added to the planning task. The action sequence is saved as a witness for the correctness of the added action.

If at any point a plan for the actual planning task is found, the actions that were not part of the original planning task are recursively replaced by their witnesses until we return a plan that contains only original actions.

Adding learned actions can be advantageous because they directly transition from one partial state in the abstract plan to another. For the SAT solver, they have the additional advantage that the same original action can be applied several times in one step. They can also be used by the search algorithm in the next phase.

4.5 Forward Search

This phase is optional. We can switch off the search completely and are left with a CEGAR approach to SAT-based planning – we call this configuration *pure PASAR*. It simply extends the formula if an action set cannot be ordered. Eventually, the abstraction is *refined* enough that one of the abstract plans can be ordered completely.

In the search phase, a greedy best-first search is started on the space of complete states. The goal is to find a plan from one partial state to the next where the plan proposed by the SAT solver cannot be ordered. But the search is not forced to visit a state that may actually be unreachable. If the search reaches a specified time limit or exceeds a specified number of nodes it is allowed to explore, it returns unsuccessfully and the abstraction is refined. More details on the implementation of our GBFS can be found in section 5.1.2.

Heuristic Function To guide the search along a similar path through the search space as suggested by the abstract plan, we use a heuristic function based on *guide states*. The guide states $H = s_0, \ldots, s_k$ are the partial states of the abstract plan.

To estimate the *gain* of executing an action a in the current state s we iterate over the guide states and the effects of the action. An effect is considered beneficial if it establishes a new assignment that is part of the guide state, and detrimental if it destroys an already fulfilled assignment that is part of the guide state. For more details, see section 5.1.2.

To encourage the search to move forward, we only consider guide states that follow the latest visited guide state in the abstract plan. In addition, we assign more weight to the guide states closer to the goal. How strongly the search is encouraged to focus on the guide states that are closer to the goal is controlled by a constant $\gamma \geq 1$.

$$\text{gain}(a, s, H) = \sum_{i=lastVisited+1}^{k} \gamma^i \sum_{x=v \in \text{eff}(a)} \begin{cases} 1, & \text{if } x = v \notin s, x = v \in s_i \\ -1, & \text{if } \exists u \in \text{dom}(x) : v \neq u, x = u \in s, x = u \in s_i \\ 0, & \text{otherwise} \end{cases}$$

If we did not execute the sparsification routine before, the gain function would mainly compute the distance to parts of the guide states that are not actually relevant to the plan, and it would be less likely that the search would actually visit a state that fulfills a guide state.

The search can use the learned actions that have been added to the problem and is encouraged by the heuristic to do so: learned actions have a high heuristic value because they directly transition a complete state that fulfills a guide state to a complete state that fulfills the next one in the current abstract plan. Previously learned actions are usually still useful to reach the goal.

In addition, the search can learn actions: if a guide state is visited and the path from the last guide state visited is sufficiently long, the plan from one guide state to the next is contracted into a single action and added to the planning task. Those plans can be thousands of actions long and allow the SAT solver to make jumps in the search space that it otherwise could not.

4.6 Refining Abstractions

If the search does not find a plan within the time limit, the abstraction is *refined.* The action sets that we could not order or skip during the search show that the abstract plan is *spurious* and serve as a *counterexample.* For each step t, clauses are added to the formula to prevent interfering actions from being used in the same action set of the next abstract plan the SAT solver finds.

Many encodings have been proposed for SAT-based planning that differ only in the way they deal with action interference. All of them could be used to refine our abstraction. We have implemented three. For all of them we iterate over the action sets in the counterexample and construct the (cyclic) disabling graph. Different clauses are added depending on the chosen *refinement strategy*:

Foreach The standard quadratic-size encoding to deal with interference. Each edge (a_1, a_2) in the disabling graph is added as a clause $\overline{a_1^t} \vee \overline{a_2^t}$. Clauses for cycles of length two are only added once.

Cycle Break This refinement strategy identifies a subset of the clauses `Foreach` adds that is sufficient to prevent all cycles. Each action in the disabling graph is assigned a rank. The rank is assigned by the same algorithm based on DFS departure times we used when ordering action sets in section 4.4. We then iterate over all edges (a_1, a_2) and generate clauses that prevent the execution of the actions in the same step if $\text{rank}(a_1) < \text{rank}(a_2)$. Essentially, we add clauses for all back edges that are defined by the spanning tree constructed by the DFS. Figure 4.4 illustrates the process.

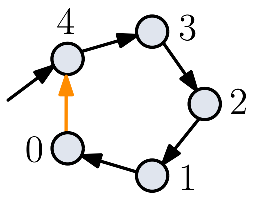


Figure 4.4: Illustration of the `Cycle Break` refinement strategy. The rank assigned to each node is given. Clauses are generated for the highlighted edge only.

Cordless Cycles This encoding has not been used as a *normal* encoding before. It would implement ∃-step semantics without imposing a fixed order on the actions, but the number of clauses can be exponential in the number of actions. The advantage over asymptotically smaller encodings [RHN06] is that no auxiliary variables are necessary. Since we do not need to generate clauses for all actions, but only for the (normally small) action sets, the asymptotically exponential size is not important. Given a disabling graph $G = (V, E)$ we identify all sets of nodes $C \subseteq V$ that induce a cordless cycle in G. A cord is an edge that connects two non-neighbouring nodes of a cycle. For each step the clause $\bigvee_{a_i \in C} \overline{a_i^t}$ is added, ensuring that at least one action in each cycle must not be executed. Figure 4.5 shows that an exponential number of clauses may be necessary.

It is sufficient to add clauses for cordless cycles only: consider a cycle with a cord. The cord skips a number of nodes of the cycle. Replacing them with the cord results in a smaller cycle. Recursive repetition yields a cordless cycle whose nodes are a subset of the original cycle. Therefore, the clause for the original cycle would in no way restrict the solution.

The advantage of this refinement strategy over the `Foreach` strategy and to a lesser extend the `Cycle Break` strategy is that clauses added to prevent a flaw in one step cannot force the SAT solver to change an action set that can be ordered in another step.

Example 3. Trucking with PASAR. Let us return one last time to the trucking problem presented in example 1. The encoding can be solved for makespan two. Decoding it gives us the action sets of the abstract plan:

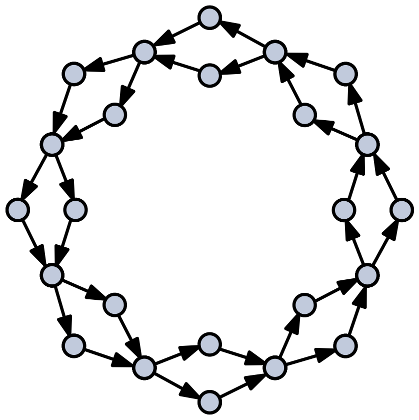


Figure 4.5: Graph with an exponential number of cordless cycles.

1 {`pickup(`T_2, P_1, L_B`)`, `pickup(`T_2, P_2, L_B`)`, `move(`T_2, L_B, L_C`)`}

2 {`drop(`T_2, P_1, L_C`)`, `drop(`T_2, P_2, L_C`)`}

Figure 4.6 is an attempt to depict the states in the abstract plan after the execution of the sparsification routine. The sparsification does not remove any action.

Next we try to order the action sets. The disabling graph for the first action set is shown in figure 4.7. It cannot be ordered because the two pickup actions interfere with each other. The disabling graph for the second action set does not contain any edges.

Since we have not found a plan yet, the search is started. For this example we assume that it failed and did not visit a guide state. The next step is to refine the abstraction. The first action set is the counterexample. Different clauses are added depending on the chosen refinement strategy. The `Cordless Cycles` strategy would identify the cycle of length two involving the two pickup actions and add a clause to ensure that at most one of the actions is executed in the same step. In this example, `Cycle Break` would generate the same clauses.

The SAT solver determines that with this clause added for each step, the formula for makespan two can no longer be solved. Increasing the makespan to three yields the following abstract plan. In this abstract plan all action sets can be ordered and PASAR returns a valid plan.

1 {`pickup(`T_2, P_1, L_B`)`, `move(`T_2, L_B, L_C`)`, `move(`T_1, L_A, L_B`)`}

2 {`drop(`T_2, P_1, L_C`)`, `pickup(`T_1, P_2, L_B`)`, `move(`T_1, L_B, L_C`)`}

3 {`drop(`T_1, P_2, L_C`)`}

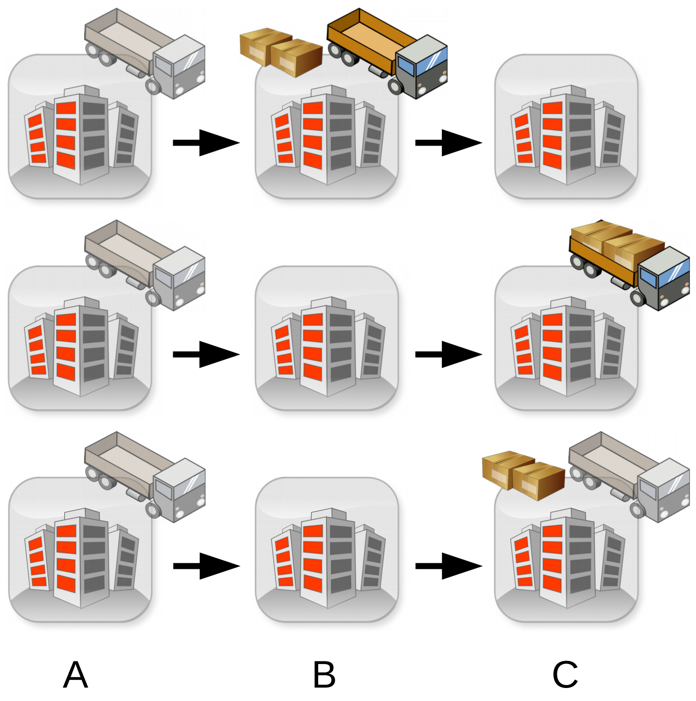


Figure 4.6: In the final state only the box positions (the goal) are relevant. The two drop actions in the second action set satisfy both. Their preconditions are the location of the boxes in the truck and the position of the truck itself. In the first step the relevant assignments are the box and truck positions being L_B and the truck being empty.

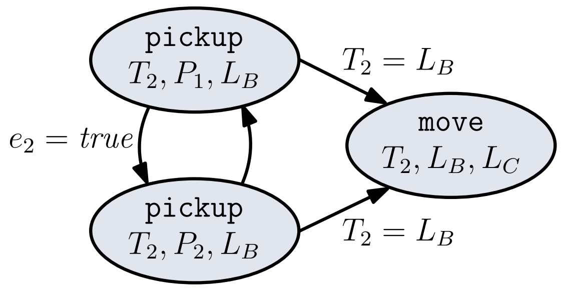


Figure 4.7: Disabling graph for the first action set. The edges are labeled with the precondition that is disabled.

4.7 Additional Features

To build a competitive planner, we have implemented some additional features that significantly improve performance.

4.7.1 Same Makespan Limit

For some planning tasks PASAR solves and refines the abstraction again and again without having to increase the makespan. This indicates that the abstraction does not represent enough of the complexity of the original problem. In order to limit the number of refinement iterations in such cases, we introduce a fallback mechanism that is triggered as soon as a certain number of refinement steps are executed without increasing the makespan. In such a case, we apply our refinement strategy to all actions, thus switching directly to the final SAT encoding instead of slowly approaching it.

4.7.2 Interleaving Search

For some planning tasks not even the first abstraction can be solved. In these cases we essentially spend our entire computation time calculating a heuristic. We have added the option to run a greedy best first search with the goal of the planning task as the only guide state before the SAT solver finds an initial solution. The runtime is spent alternating between the SAT solver and the search until one succeeds.

5 Experimental Evaluation

In this chapter we will first describe some implementation details. We continue by evaluating the various features of our planner and tune their parameters. Finally, we compare the final configuration with other well known planners.

The source code for our planner is available at GitHub.[1]

5.1 Implementation Details

Our algorithm is implemented using `C++` and compiled with `g++ v.7.4.0` using full optimization flags (`-O3`).

The speed of the code that generates encodings, decodes SAT models or performs similar tasks is not critical to the performance of our planner. Relevant are the performance of the used SAT solver, the *grounding* procedure and the speed of the forward search. Out of these, we only implemented the search ourselves.

5.1.1 Grounding

Planning tasks are usually defined in the human-readable *Planning Domain Definition Language* (PDDL). *Grounding* refers to the process of translating a planning task defined in a high-level language like PDDL to an expanded representation better suited for most planning techniques. We use Fast Downward [Hel06] to ground PDDL problem files into multi-valued planning tasks and then make some simplifications to obtain our internal representation. The explicit mutex information[2], sometimes generated by Fast Downward is encoded for the SAT solver.

Since we use a smaller timeout for our experiments than the 30 minutes Fast Downward was tuned for, the *invariant generation* is limited to 10 seconds. This is suggested by the authors of Fast Downward.[3]

[1]`https://github.com/Froleyks/pasar.git`
[2]`http://www.fast-downward.org/TranslatorOutputFormat#mutex`
[3]`http://www.fast-downward.org/IpcPlanners`

```
Procedure GBFS($\Pi = (X, O, s_I, s_G)$)
    plan = $\langle\rangle$
    visited = $\{s_I\}$   // Hash Set
    $A$ = getApplicableActions($s_I$)
    $s = s_I$
    while $s_G \not\subseteq s$ do
        sort($A, s$)   // sort actions by their gain
        for $a \in A$ do
            $s'$ = apply($a, s$)
            if $s' \notin$ visited then
                plan.append($a$)
                visited.insert($s'$)
                $A$ = updateApplicableActions($A, a, s'$)
                $s = s'$
                break
            end
        end
        if no action was appended to the plan then
            $a$ = plan.pop()
            $s$ = apply($a^{-1}, s$)   // undo changes caused by $a$
            $A$ = updateApplicableActions($A, a^{-1}, s'$)
        end
    end
    return plan
```

Algorithm 2: Greedy Best First Search (GBFS).

5.1.2 Greedy Best-First Search

Implementing a forward search in the context of planning presents a number of challenges. This section describes how we address them, without being specific to PASAR. First, we explain why we use *gain* values instead of a heuristic value for states. Then we explain how we implement state expansion (successor generation) and finally we describe how our hash function is defined. The pseudo-code for a GBFS in the context of planning is specified in algorithm 2.

Heuristic Since we only compare direct successors of a state, we can use a *gain*-function to calculate how beneficial it is to execute an applicable action a in the current state s. We do this instead of the more common approach of computing heuristic values for states, because this way only the effects of a have to be considered. The number of effects of an action is typically much smaller than the number of state variables. An effect is considered beneficial if it establishes a new assignment that is part of the goal, and detrimental if it destroys an already fulfilled assignment.

$$\text{gain}(a, s, s_G) = \sum_{x=v\in \text{eff}(a)} \begin{cases} 1, & \text{if } x = v \notin s, x = v \in s_G \\ -1, & \text{if } \exists u \in \text{dom}(x) : v \neq u, x = u \in s, x = u \in s_G \\ 0, & \text{otherwise} \end{cases}$$

Note that we compute the gain of all actions and sort them before we check if the state that is reached by it was visited before. This check is realized with a *hash set* and although some work has been put into optimizing this step, the computation of the gain-function is typically still much cheaper than the table lookup.

Tie Breaking To break ties, we add pseudo-random noise to the gain-values. This prevents a bias from developing in the search trajectory based on the order of the actions in memory. The effect of adding noise is explicitly evaluated in section 5.3.3.

State Expansion An important implementation detail is the `updateApplicableActions` routine in line 13. The easiest way to implement it is to iterate over all actions in O and check if they are applicable. This would take quite a long time, as the number of actions tends to be high[4] and the routine has to be executed for each visited state. A more efficient way is to precompute the *action support*.

The action support A_{Supp} maps each assignment to the set of actions that have this assignment as a precondition: $a \in \text{A}_{\text{Supp}}(v = x) \Leftrightarrow (v = x) \in \text{pre}(a)$.

With the action support, we do not have to check for each action in O if it is applicable. An action a' that is applicable in the new state s' was already applicable in s or all missing preconditions were fulfilled by the effects of a. Therefore, only the previously applicable actions and the action supports of each assignment in $\text{eff}(a)$ need to be checked.

Hash Function To ensure that the search does not explore the same state twice, a hash set is used to store each visited state (see line 10 in algorithm 2).

Most planning tasks have a lot of Boolean state variables and a few variables with larger domains, which are referred to as multi-value variables in the following. For reasons of space and access speed it is important to store the values of the variables in a compact way. To store Boolean variables we use single bits (`std::vector<bool>` does that in our implementation of `C++`) and for multi-valued variables we use a small data type that fits the domain size.

The hash function is defined on this representation of a state. For the Boolean variables `std::hash` is used, since it has an efficient specialization for `std::vector<bool>`.

[4]The maximum in the *IPC*-set exceeds one million and the average is approximately 28K.

For the multi-valued variables Zobrist hashing [Zob70] is used. The results are combined using an exclusive `or` operation (`XOR`).

The compact representation is not optimal for every operation we want to perform. For example, computing heuristic values requires random access to the values of a few variables. Random read access to `std::vector<bool>` can be very slow. Therefore, we also maintain an expanded representation in which each Boolean variable is stored as a byte.

5.2 Experimental Setup

Our experiments are based on the agile track of the *International Planning Competition*. The runtime is limited to 300 seconds and the memory is limited to 15 GB. The quality of the discovered plans is ignored. Instead, we will focus on the number of instances solved and the *score* for each planner. The *score*[5] of a configuration or planner on a task solved in T seconds is defined as:

$$\text{score}(T) = \begin{cases} 1, & \text{if } T \leq 1 \\ 1 - \frac{\log(T)}{\log(300)}, & \text{if } 1 < T \leq 300 \end{cases}$$

The score for a task that is not solved within the time limit is 0. The score of a planner for a domain or an entire test set is the sum of the scores for each task.

5.2.1 Environment

The tests are run on an AMD EPYC 7551P 32-Core processor. 256 GB of DDR3 RAM are available. The computer runs Ubuntu 18.04.3 LTS with the Linux kernel 4.15.0-72-generic.

Up to 16 instances of our planner are run in parallel. To limit memory and runtime, we use the *runlim* tool.[6]

The same test setup is used for the other planners we tested, with exception of *Fast Downward*. Since runlim does not work properly with Fast Downward, we used the command line options provided by the planner to limit memory.

5.2.2 Test Instances

We use two test sets for our experiments. The *IPC*-set consists of 571 planning tasks from the satisficing and optimal tracks of the International Planning Competitions

[5]This metric is used to rank the planners competing in the agile track of the IPC `https://ipc2018-classical.bitbucket.io/#tracks`

[6]`http://fmv.jku.at/runlim/`

2014 and 2018. We do not include any of the benchmarks that have conditional effects, as our planner is not yet equipped to deal with them. We use this test set for an internal comparison of different configurations of PASAR.

In addition, we use the *sparkle*-set for a final comparison of PASAR with other planners. It consists of the 70 benchmarks without conditional effects used in the *Sparkle Planning Challenge* 2019. As there is no overlap in the domains used in the two test sets, we avoid an unfair advantage for our planner when comparing it to the competition. That being said, the parameters of our planner are tuned to the time and memory limits used in the experiment, while the other solvers do not have this potential advantage.

The exact domains used and the number of planning tasks in each domain can be found in the Appendix C. Unless otherwise specified, the *IPC*-set is used for the experiments.

Instead of repeating the grounding step for each experiment, we grounded each planning task only once and saved the result. The grounding was performed with the same time and memory limits as described above. The time needed for grounding a planning task is added to the total runtime of our planner whenever we do a comparison to other planners.

5.2.3 Variance

We did not run all our experiments several times to measure the variance. Our algorithm is deterministic if none of the timeouts based on wall time is used, so we do not expect any significant variance. Also, the test sets are quite large.

Nevertheless, we ran our final configuration five times on the IPC-set (sparkle-set) and measured the standard deviation of the runtime on each instance. The average standard deviation over the test set was 0.16 (0.73) seconds and the maximum was measured at 4.92 (4.67) seconds for an instance with an average runtime of 106 (180) seconds.

5.3 Parameter Evaluation

PASAR exposes a lot of parameters that influence its performance.

We tried automatic parameter tuning with `ParamILS` [Hut+09] without success. We used version 2.3.8, running 16 diversified instances of `FocusedILS` in parallel for 10 days. All configurations found during this time performed significantly worse than the default configuration we started with. This may be because the parameter space is too large, the test instances are ill-suited for this purpose or simply because of the limited time available.

We will introduce the different features of PASAR one by one and evaluate their influence on the performance of our planner. As a baseline, we start with standard SAT encodings and compare them with a basic version of our algorithm without forward search. We then evaluate the search on its own and as a part of PASAR. Finally, we evaluate the contribution of the different features to the performance of our final configuration.

5.3.1 Basic Encodings

First, we evaluate standard encodings and the different SAT solvers that we can use to establish a baseline for our experiments.

SAT Solvers We start with a standard SAT encoding that implements $\forall$-step semantics for our multi-valued formalism as a baseline. PASAR supports any SAT solver that implements the `IPASIR` interface. We compare the two that support limiting the number of conflicts, namely `Glucose` [AS09] and `MiniSAT` [ES03]. The number of conflicts per makespan is limited to 20000. The makespan for the first formula we solve is 5, and we increase the makespan by a factor of 1.2. See section 4.2 for more details.

The results are shown in figure 5.1 using a *Cactus*-plot. It shows how many planning tasks are solved by each SAT solver over time. Details on this type of plot can be found in [BDG17]. In addition, the legend shows the score of each solver.

`Glucose` will be used for the next experiments, as it clearly outperformed its predecessor `MiniSAT`.

SAT solvers will be used slightly differently in our algorithm. The formulas will contain fewer clauses of size two (for which many SAT solvers are optimized) and more emphasis will be placed on solving a formula that is only slightly different from the last one solved. We reevaluate the SAT solver for the use with different versions of PASAR in Appendix B. There we also test the two SAT solvers `Lingeling` and `PicoSAT`. The results do not change significantly and `Glucose` still performs very well.

Refinement Strategy Every refinement strategy introduced in section 4.6 induces a normal encoding when being applied to the set of all actions. The `Foreach` strategy induces the same well known encoding we used in the last experiment, while the other two are more novel. Figure 5.2 shows how many instances are solved with each of them over time.

While the `Cycle Break` encoding initially outperforms the `Cordless Cycle` encoding, the latter ultimately solves more instances within the time limit and achieves a higher score. This can be attributed to the higher parallelism the encoding allows.

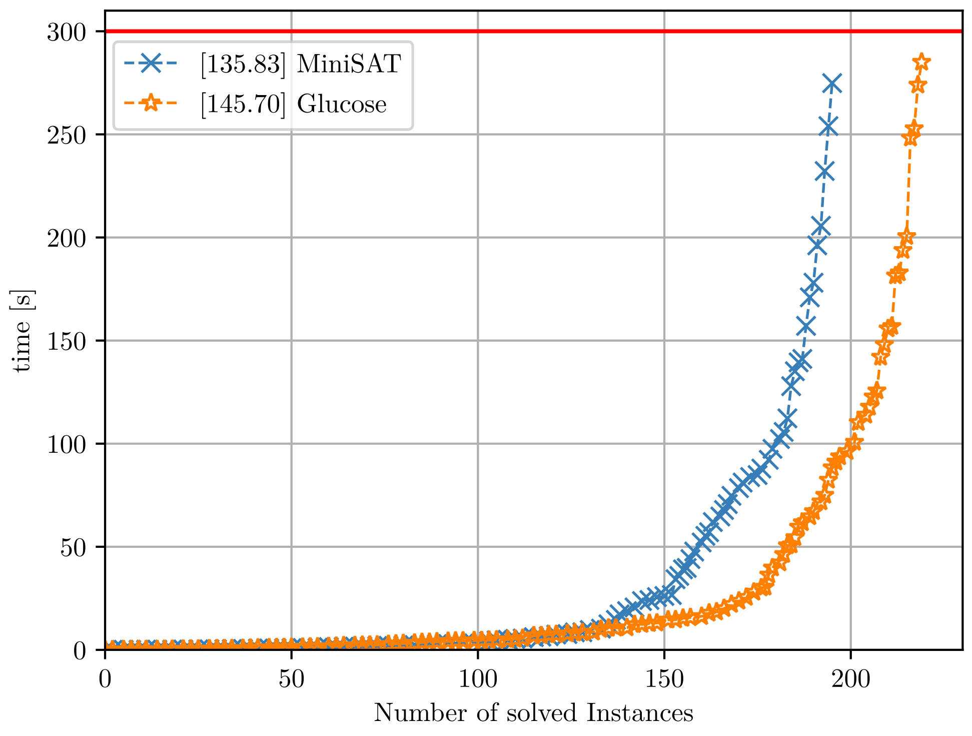


Figure 5.1: Comparison of SAT solvers on a classical ∀-encoding for planning tasks.

On easier planning tasks, proportionally more time is spent on proving that the formulas are unsatisfiable for the lower makespans. Proving that a formula is not satisfiable is generally easier if the formula is more constrained. This may explain the initial advantage of `Cycle Break`.

Figures 5.3a and 5.3b compare the two encodings with the `Foreach` encoding using *scatter plots*. This type of plot is used to compare the performance of two algorithms on the same instance. Marks in the margins represent instances that can be solved by one algorithm, but not by the other. For more information we refer to [BDG17].

Since `Cycle Break` generates a subset of the clauses generated by `Foreach`, it is not surprising that their performance is similar on many instances. However, in some cases `Cycle Break` has a clear advantage due to the higher parallelism and smaller formulas. `Cordless Cycle` does not perform as well as `Foreach` on a number of instances. This can again be explained by the fact that it takes longer to dismiss formulas for lower makespans because they are less constrained. Within the time limit `Cordless Cycle` still solves many more instances.

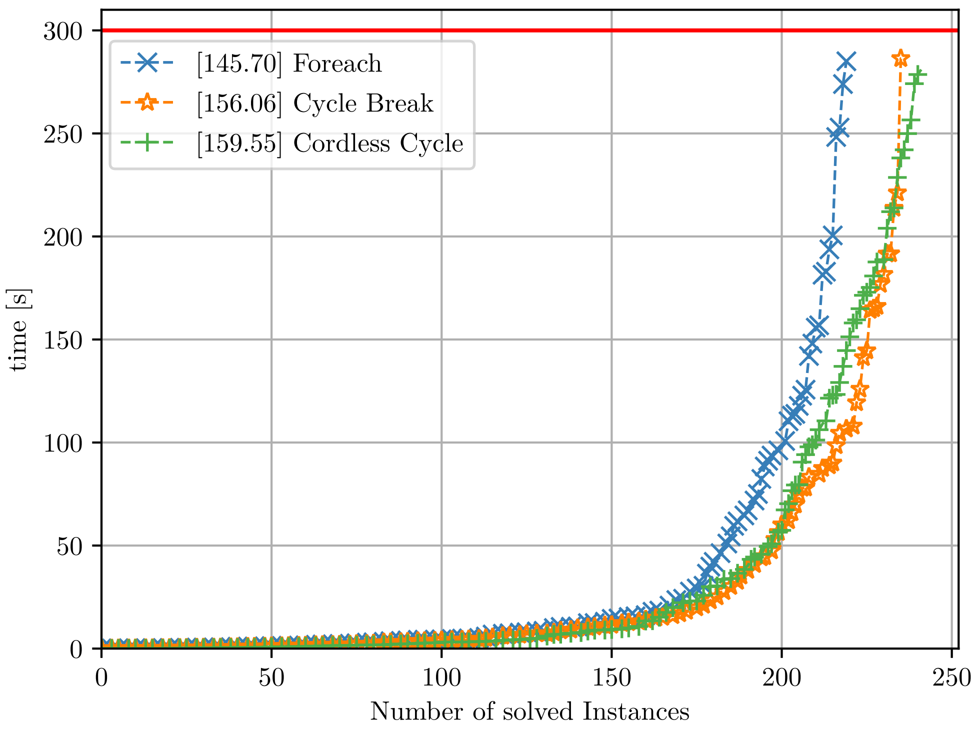
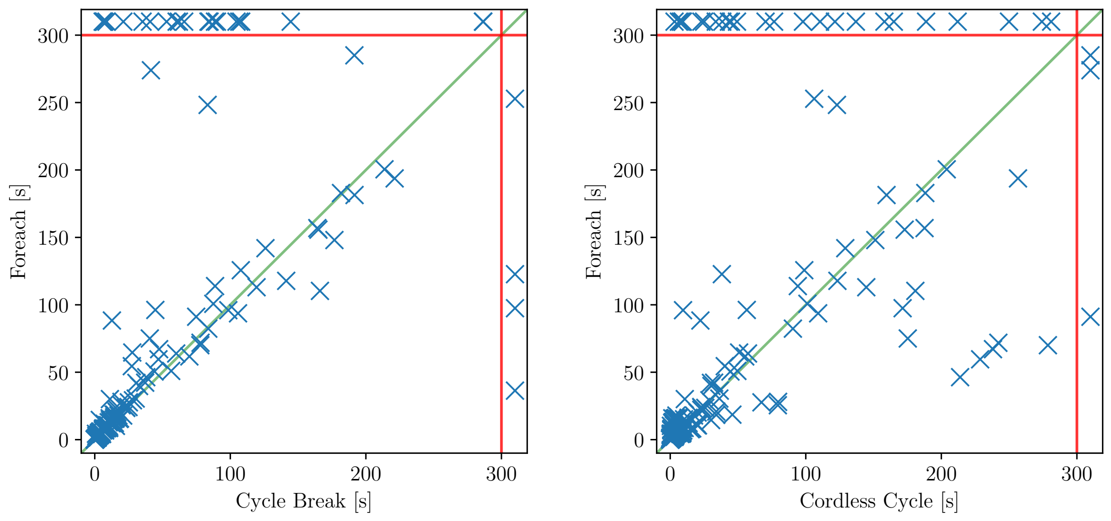


Figure 5.2: Comparison of the encodings induced by each refinement strategy.



(a) `Cycle Break` encoding.

(b) `Cordless Cycle` encoding.

Figure 5.3: Comparison of encodings to the `Foreach` encoding.

5.3.2 SAT-Based Planning with CEGAR

The most basic version of our algorithm is a CEGAR approach to SAT-based planning, which we have introduced as *pure PASAR* in section 4.5. Pure PASAR simply finds a model for an encoding without action interference, tries to order the action sets and adds interferences to the encoding where it failed to order the actions.

Sparsification Before we compare the different refinement strategies, we evaluate the effect of *sparsification* on the performance of pure PASAR. The results are shown in figure 5.4. Since the states of the abstract plan are not used in the current configuration, the only effect is that actions could be removed from the action sets, making it more likely that they can be ordered.

A few instances are solved more slowly or not at all. This can be attributed to the fact that the performance of SAT solvers is influenced by the order of clauses. This effect is relevant enough that benchmark formulas in SAT competitions are shuffled before solving [LS04].

In general, sparsification is beneficial; increasing the score from 127.21 to 140.83. We will enable it in the following experiments.

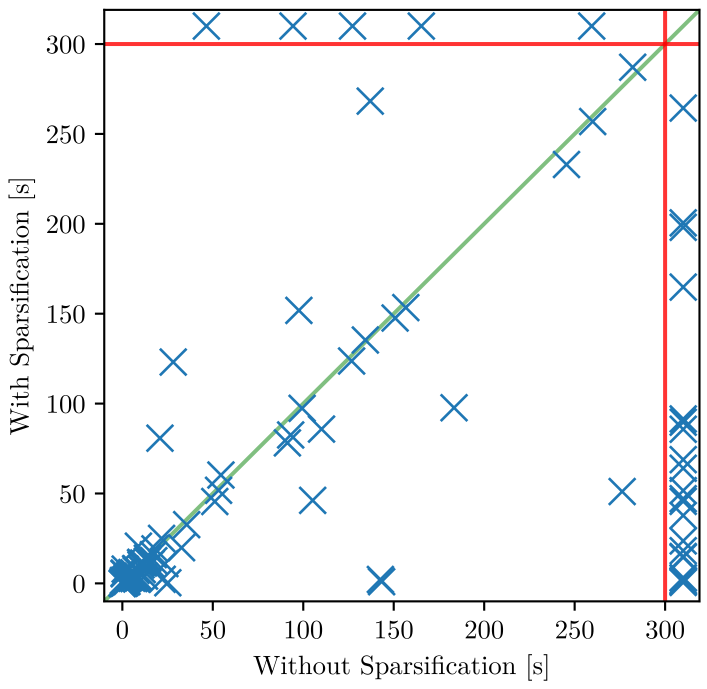


Figure 5.4: Comparison of pure PASAR using the `Foreach` refinement strategy with and without sparsification.

Refinement Strategy Compared to normal SAT-based planning, the formulas that have to be solved during the CEGAR approach are much smaller, especially in comparison to encodings that allow similar parallelism. Table 5.1 compares the number of clauses generated to avoid interference by the normal encoding with the number of clauses added to the formula by pure PASAR for each refinement strategy.

A comparison of the normal encodings shows that `Foreach` always generates the most clauses and `Cycle Break` the least. The differences in the number of clauses added are not as significant. Whenever an action set can be ordered, ∃-step semantics are fulfilled and the refinement strategy is not used. If the ordering of an action set fails, all refinement strategies must prevent the interfering actions from being used again in the same step. The only difference between them is the effect they have on the other actions in the set and in other sets. `Cycle Break` and `Cordless Cycle` still perform significantly better than `Foreach`.

Interesting are the domains *floortile, ged, snake* and *visitall.* For planning tasks from these domains, the first abstract plan can always be ordered, so no additional clauses are added. In general, the number of clauses added is small. For `Cycle Break` and `Cordless Cycle`, the maximum percentage of clauses added per domain is less than 1.5%. The maximum is reached on *openstacks*.

However, when it comes to the total number of instances solved, pure PASAR performs significantly worse than the normal encodings. Details are given in table 5.2. One reason for this is the added overhead of refining the formula and restarting the SAT solver. Another important factor is that the formula is less constrained. In many cases, this leads to the SAT solver taking longer to dismiss a formula that is unsolvable for a particular makespan.

Same Makespan Limit To recover some of the lost instances, we introduced the same-makespan-limit in section 4.7.1. For these tests the same-makespan-limit is set to 10. The results are shown in figure 5.5.

With the same-makespan-limit enabled `Cycle Break` outperforms the `Cordless Cycle` refinement strategy. One reason for this could be that the computation of the potentially exponential number of clauses to be added for the fallback can take a significant amount of time. In figure 5.6 the results of combining the Cordless Cycles refinement strategy with different fallback mechanisms are shown.

The combination of `Cordless Cycle` with `Cycle Break` performs the best out of these options but `Cycle Break` on its own still reaches a higher score. We will stick to `Cycle Break` as the refinement strategy for the rest of our experiments.

In table 5.3 different values for the same-makespan-limit are compared. While the differences are not that significant, we will stick to a same-makespan-limit of 10.

	Foreach		Cycle Break		Cordless Cycle	
Domain	Clauses	Added	Clauses	Added	Clauses	Added
OPT						
childsnack (15)	410 769	7 303	402 789	1 802	402 807	1 933
data-network (18)	528 273	22	462 185	21	480 199	20
floortile (20)	21 650	**0**	8 099	**0**	17 392	**0**
ged (20)	331 319	**0**	251 313	**0**	305 207	**0**
hiking (8)	445 581	1 169	276 259	1 099	366 806	1 249
openstacks (4)	446 036	6 509	425 604	6 251	435 812	6 509
organic-synth. (6)	2 055 660	1 908	1 739 213	1 796	1 770 619	1 840
petri-net-alig. (1)	64 213	119	30 842	32	30 850	32
snake (3)	22 395 449	**0**	22 219 787	**0**	22 306 698	**0**
termes (1)	97 978	130	50 000	324	93 414	120
tetris (12)	5 303 957	850	3 141 504	472	4 213 040	746
transport (13)	1 372 481	25	1 034 763	11	1 134 861	10
visitall (17)	79 693	**0**	78 799	**0**	79 239	**0**
SAT						
childsnack (5)	790 562	11 549	777 696	1 902	777 717	1 086
floortile (20)	37 588	**0**	14 196	**0**	31 076	**0**
hiking (3)	14 496 650	4 409	8 754 254	5 191	11 719 410	2 154
organic-synth. (6)	2 158 055	7 711	1 693 523	7 015	1 736 098	9 008
tetris (2)	4 338 438	886	2 575 573	1 146	3 434 294	890
thoughtful (4)	342 199	276	293 930	556	320 478	176
Average (173)	2 932 450	2 256	2 327 912	1 454	2 613 475	1 356

Table 5.1: Comparison of the number of clauses generated by the normal encoding to prevent action interference to the number of clauses added by pure PASAR for the different refinement strategies. Only instances solved using every refinement strategy are considered. The number of solved instances per domain is given in parentheses. All values are the arithmetic mean over all solved instances in the domain. The domains are split in test instances from the optimal and the satisficing tracks. See C.1 for more details.

Configuration	Solved	Score
PASAR Foreach	188	140.82
PASAR Cycle Break	197	145.82
PASAR Cordless Cycle	197	147.12
Foreach	219	144.70
Cycle Break	236	155.05
Cordless Cycle	**241**	**158.55**

Table 5.2: Comparison of pure PASAR to the normal encoding for each refinement strategy. The table is sorted by the number of solved instances. According to the score the normal `Foreach` encoding performs worse than the other two refinement strategies when used with PASAR.

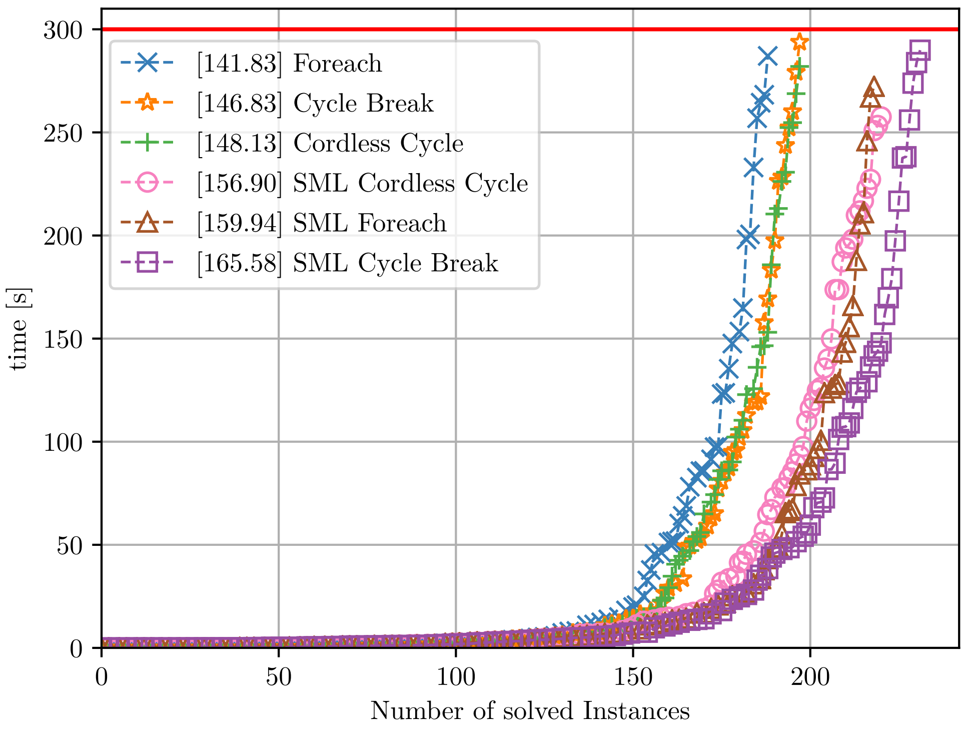


Figure 5.5: Effects of enabling the same-makespan-limit for each refinement strategy.

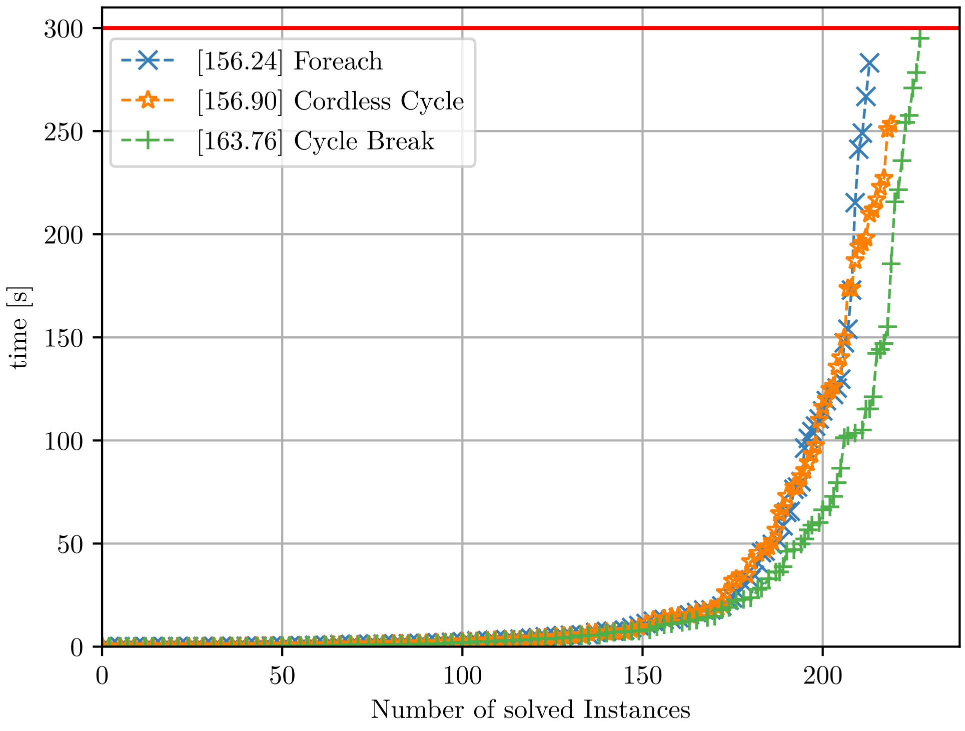


Figure 5.6: Combining `Cordless Cycle` with different fallback mechanisms after the same-makespan-limit is reached.

SML	Solved	Score
5	226	163.19
9	226	162.97
10	**231**	**164.58**
11	229	163.04
15	224	163.19
30	222	161.59

Table 5.3: Comparison of different values for the same-makespan-limit, using pure PASAR with the `Cycle Break` refinement strategy.

5.3.3 Search

Next, we will evaluate the forward search on its own. Implementation details are explained in section 5.1.2. The results of the experiment are shown in figure 5.7. A sobering observation is that even a simple DFS outperforms the best configuration of pure PASAR that we have found. Besides that, the benefit of random tie breaking is astounding.

In PASAR the search will be used differently; it will only be allowed to explore a few nodes before it is restarted. Therefore, it will not be able to close many states. The experiment shows that this does not affect the performance too much, which is not surprising since the number of states the search can close is dwarfed by the size of the search space.

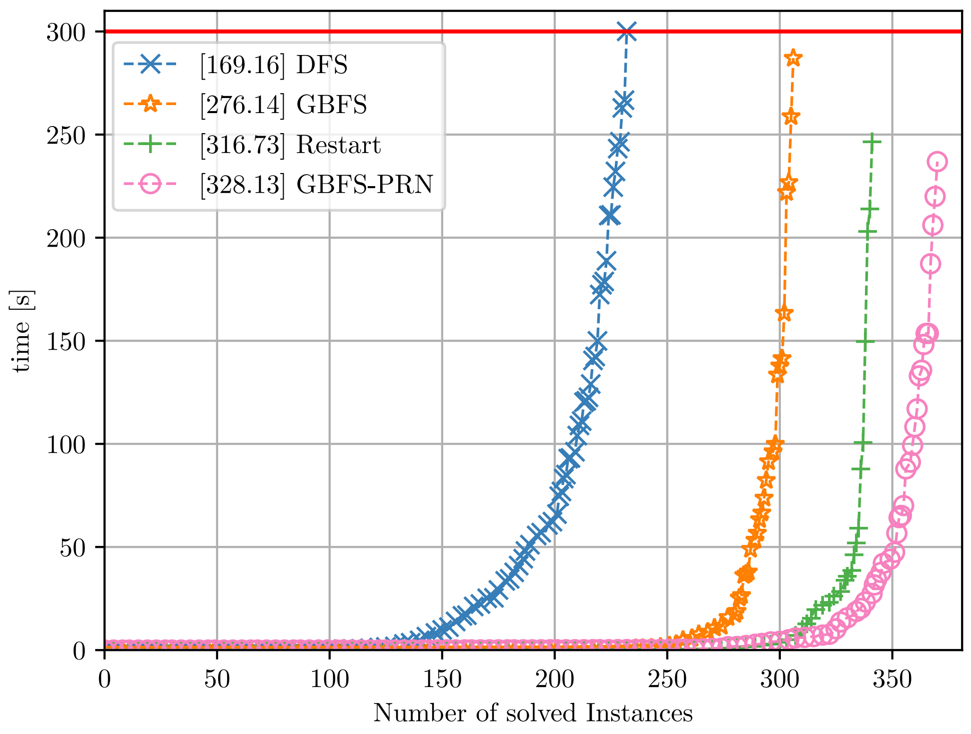


Figure 5.7: Comparison of basic forward search algorithms. *DFS* is an unguided depth first search. *GBFS* is a greedy best-first search that uses the distance to the goal as a heuristic function. *GBFS-PRN* is the same algorithm with the addition of a pseudo-random noise to break ties. *Restart* does the same as *GBFS-PRN*, but the search restarts (after clearing the visited table) from the initial state after exploring 20000 states.

5.3.4 PASAR

Combining the SAT-based aspects and the forward search of PASAR introduces a number of new parameters. Of particular interest are the parameters that control how the resources are shared between the SAT solver and the forward search.

For the following experiments the search is enabled and the number of nodes it is allowed to explore is limited to 20000. The algorithm now completely follows the outline shown in figure 4.1.

Sparsification Now that the search is enabled, the sparsification does not only help with ordering action sets, but also influences the heuristic function that the search uses. We evaluate the effects of sparsification once again in figure 5.8. With sparsification enabled, 23 additional instances are solved. The few lost instances are randomly distributed over the domains and can be attributed to the influence the order of clauses has on SAT solvers.

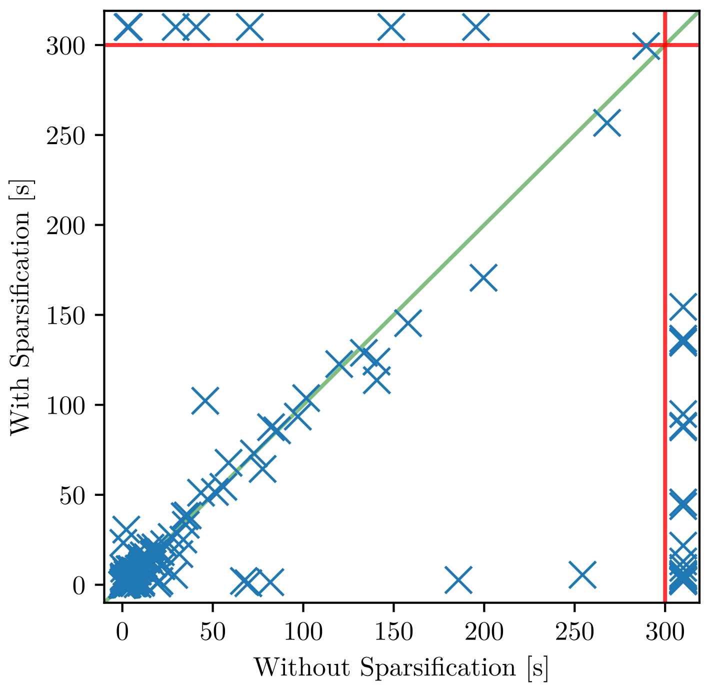


Figure 5.8: Evaluating the effect of sparsification on full PASAR.

Interleaving Search Even with the sparsification, PASAR can only achieve a score of 246.69, which is significantly less than the score of 327.13 the greedy best-first search achieves on its own. In table 5.4 we investigate the connection between solving the first abstraction of a planning task and solving the complete problem.

For a few domains the first abstraction is easy to solve, but solving the planning task is still hard (SAT-data-network, SAT-hiking, SAT-barman, OPT-barman). This indicates that the abstraction for these planning tasks does not sufficiently reflect the actual difficulty of the problem. In contrast, we have the domains SAT-visitall, SAT-transport and especially snake from both tracks; for these planning tasks, even the first abstraction is rarely solved. The GBFS on the other hand solves them almost instantly. This suggests a fairly long minimal plan (and abstract plan) length and the existence of many valid plans for the planning tasks. Although we use exponential makespan scheduling, SAT solvers are not equipped to deal with formulas that exceed a certain size. In these cases, PASAR essentially spends its entire runtime on computing a heuristic that is never used. To address this problem we have introduced *interleaving search* in section 4.7.2. Enabling it increases the performance considerably (see figure 5.9).

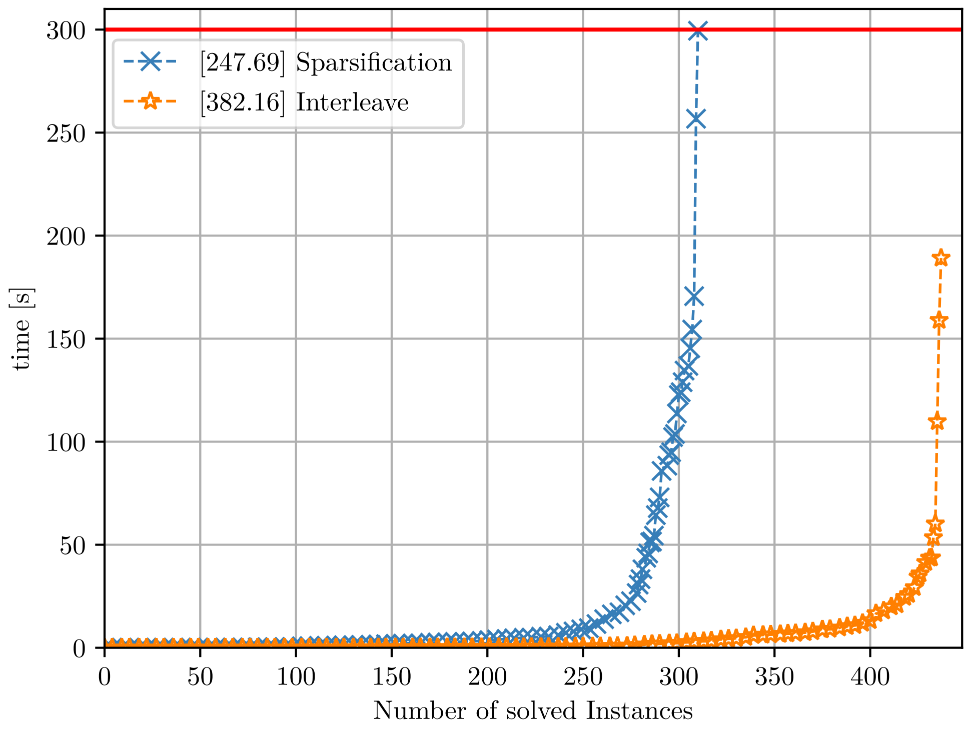


Figure 5.9: Evaluation of the effect of interleaving search.

Domain	abst.	PASAR	GBFS	$t_{max}[s]$
OPT				
barman (14)	**14**	0	2	148.19
childsnack (20)	**20**	**20**	0	—
data-network (20)	**20**	**20**	**20**	1.88
floortile (20)	**20**	**20**	0	—
ged (20)	**20**	**20**	**20**	0.02
hiking (20)	**20**	16	**20**	3.97
openstacks (20)	**20**	**20**	**20**	0.13
organic-synth. (20)	**20**	9	11	108.30
petri-net-alig. (20)	18	17	**20**	43.05
snake (20)	4	4	**20**	**0.13**
termes (20)	11	5	9	206.12
tetris (17)	**17**	**17**	**17**	0.69
tidybot (20)	**20**	19	13	0.32
transport (20)	17	14	**20**	0.18
visitall (20)	17	17	**20**	0.01
SAT				
barman (20)	**20**	0	0	—
childsnack (20)	**20**	19	0	—
data-network (20)	**20**	3	15	219.99
floortile (20)	**20**	**20**	0	—
ged (20)	0	0	10	187.34
hiking (20)	17	3	**20**	135.72
openstacks (20)	**20**	**20**	**20**	0.44
organic-synth. (20)	17	2	5	42.07
snake (20)	0	0	**20**	**0.51**
termes (20)	2	0	3	70.08
tetris (20)	**20**	**20**	**20**	0.43
thoughtful (20)	5	5	5	236.89
transport (20)	0	0	**20**	**8.11**
visitall (20)	0	0	**20**	**0.25**
Sum (571)	419	310	370	—

Table 5.4: The number of instances per domain is given in parentheses after the domain name. The first column shows the number of planning tasks for which PASAR can solve the first abstraction and the next column lists how many planning tasks are actually solved. GBFS gives the number of instances the greedy-best first search can solve on its own, and the last column shows the maximum time the GBFS took to solve an instance for each domain. In this column values are highlighted which are considered to be particularly low compared to the effort required to solve the first abstraction. In the other columns, an entry is marked if it has reached the maximum possible value.

Search Limit The *search limit* is the number of nodes the search is allowed to explore before it returns unsuccessfully and the abstraction is refined. Using too low a search limit prevents the search from closing states and requires that the abstraction is refined enough to guide the search almost perfectly. It also limits the length of the plans that can be found. However, learned actions can be used by the search to extend its reach. Choosing a search limit that is too high slows down the refinement of the abstraction, and in the worst case, turns PASAR into a not particularly sophisticated forward search.

Different search limits are compared in table B.1. Allowing the search to explore up to 4 million nodes has a negative impact on the performance, both in terms of solved instances and the score the planner achieves. A less extreme increase of the search limit allows PASAR to solve some more instances but also reduces the score. In figure 5.10, configuration **A** which solves the most instances is compared to configuration **B** which achieves the best score. For a number of instances, the time to solve does not change. These are solved within a few refinement steps. The instances that require more refinement steps are solved much more slowly. We will stick to a lower search limit of 20000 for our final configuration to preserve what makes PASAR novel.

	Search Limit	Solved	Score
	4000000	427	362.47
A	800000	**444**	369.56
	600000	443	370.23
	400000	443	372.75
	100	424	376.33
B	20000	437	**381.15**

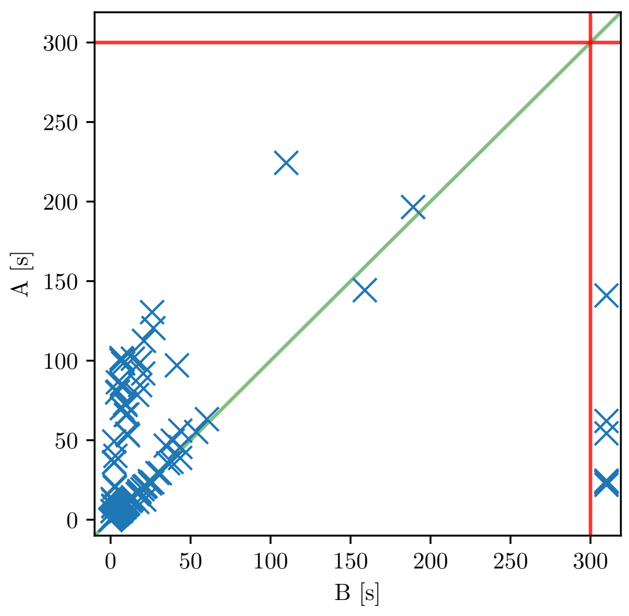


Figure 5.10 & Table 5.5: Comparison of different search limits.

Contribution We give a final overview of the contribution of each feature to the performance of PASAR in figure 5.11.

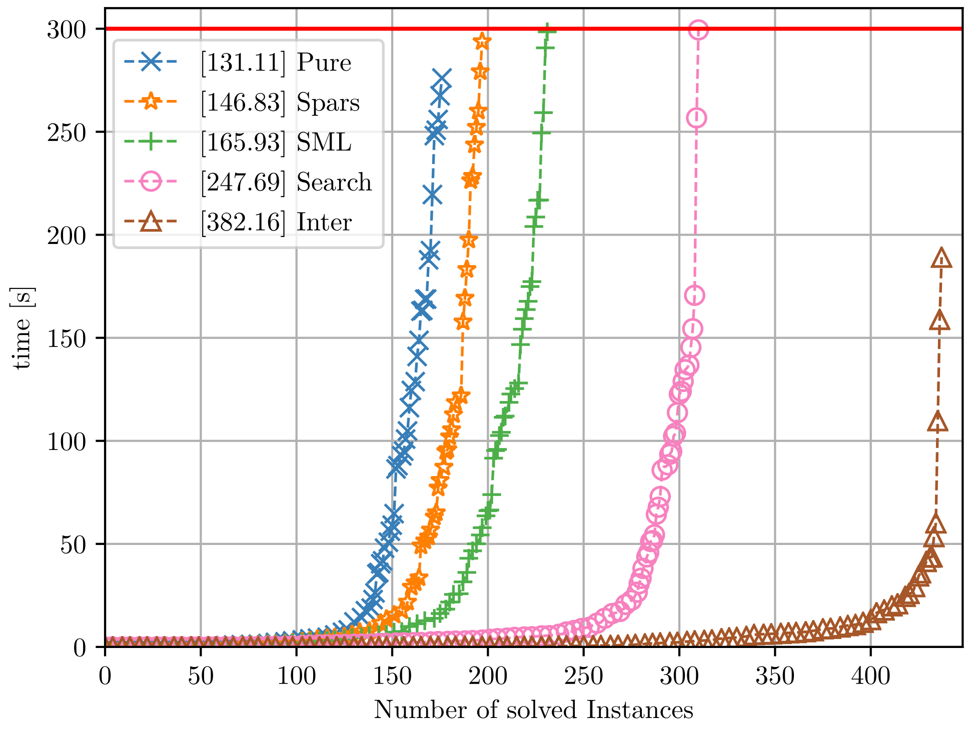


Figure 5.11: The features of our final configuration of PASAR are enabled one after the other. *Pure* uses the `Cycle Break` refinement strategy and no additional features. Then, the sparsification is enabled and then the fallback after reaching the same-makespan-limit. *Search* starts the search after an abstraction has been solved. *Inter* additionally starts a search before the first abstraction has been solved.

5.4 Comparison

In this section we will compare the performance of our final configuration to other well known planners.
We include

- *Fast-Forward* **FF** [Hof01], which dominated in the beginning of planning competitions,
- *Fast Downward* **FD** [Hel06] in its LAMA 2011 configuration, which we patched to return after finding a plan instead of optimizing it, and
- *Madagascar* [Rin14] a SAT-based planner which uses its own integrated SAT solver, both in its default configuration with ∃-step semantics (**M-E**) and a configuration utilizing ∀-step semantics (**M-FE**).

Grounding In the previous experiments we ignored the time needed to ground the planning tasks to a multi-valued representation. In the following experiments, the grounding is performed during the measured runtime.

5.4.1 Tuning Set

The number of instances the planners have solved over time is presented in figure 5.12 using a cactus plot, and the number of solved instances per domain is given in table 5.6. Regarding the number of instances solved, Fast Downward is the only competition. For the first 10 seconds it has a very small advantage, after that PASAR is significantly faster. This also results in a higher score for PASAR. Regarding the number of solved instances, Fast Downward can catch up within the time limit.

A look at the performance for each domain (especially the harder ones from the satisficing track) shows that we accomplished our goal of combining the strength of SAT-based and forward search based planning algorithms on most domains. On *floortile* and *childsnack*, Madagascar has a clear advantage over the forward searches and PASAR can easily match it. On the domains *visitall*, *openstacks*, *petri-net-alignment*, and *data-network* Fast Downward outperforms the SAT-based approach. PASAR can match it as well.

PASAR outperforms both approaches on the domains *tetris*, *hiking* and especially *transport* and *snake*. There are also domains where PASAR does not perform well; especially *barman*. On *ged* and, to a lesser extend, on *organic-synthesis*, every planner tested except for Fast-Forward solves more instances.

Overall, PASAR solves nearly as many instances as Fast Downward and solves more domains completely. It also achieved the highest score of all tested planners.

Figure 5.13 shows a scatter plot that directly compares the performance of Fast Downward and PASAR on the same instance. It suggests that it would be beneficial to combine the two in a portfolio. The simplest portfolio possible, running each planner for half of the time available, can solve 93% of the planning tasks in the *IPC*-set.

Domain	FF	M-FE	M-E	FD	PASAR
OPT					
barman (14)	6	**14**	**14**	**14**	0
childsnack (20)	3	**20**	15	15	**20**
data-network (20)	**20**	**20**	**20**	**20**	**20**
floortile (20)	0	**20**	**20**	8	**20**
ged (20)	**20**	**20**	**20**	**20**	**20**
hiking (20)	14	10	9	**20**	**20**
openstacks (20)	3	**20**	**20**	**20**	**20**
organic-synth. (20)	0	12	12	**17**	10
petri-net-alig. (20)	0	1	14	**20**	19
snake (20)	10	15	15	13	**20**
termes (20)	7	0	1	**18**	5
tetris (17)	0	15	**17**	**17**	**17**
tidybot (20)	19	16	**20**	19	**20**
transport (20)	**20**	**20**	**20**	**20**	**20**
visitall (20)	18	**20**	**20**	**20**	**20**
SAT					
barman (20)	0	0	4	**19**	0
childsnack (20)	0	17	7	5	**19**
data-network (20)	4	4	4	11	**12**
floortile (20)	1	**20**	**20**	2	**20**
ged (20)	4	14	12	**20**	4
hiking (20)	6	4	6	16	**20**
openstacks (20)	0	2	0	**20**	**20**
organic-synth. (20)	0	7	7	9	4
snake (20)	3	7	7	3	**20**
termes (20)	0	0	0	**13**	2
tetris (20)	5	5	8	14	**20**
thoughtful (20)	12	5	5	**15**	5
transport (20)	0	0	0	10	**20**
visitall (20)	0	0	0	**20**	**20**
Sum (571)	175	308	317	**438**	437
Score	141.6	225.76	236.42	309.63	**314.36**

Table 5.6: Number of solved instances for each domain in the *IPC*-set.

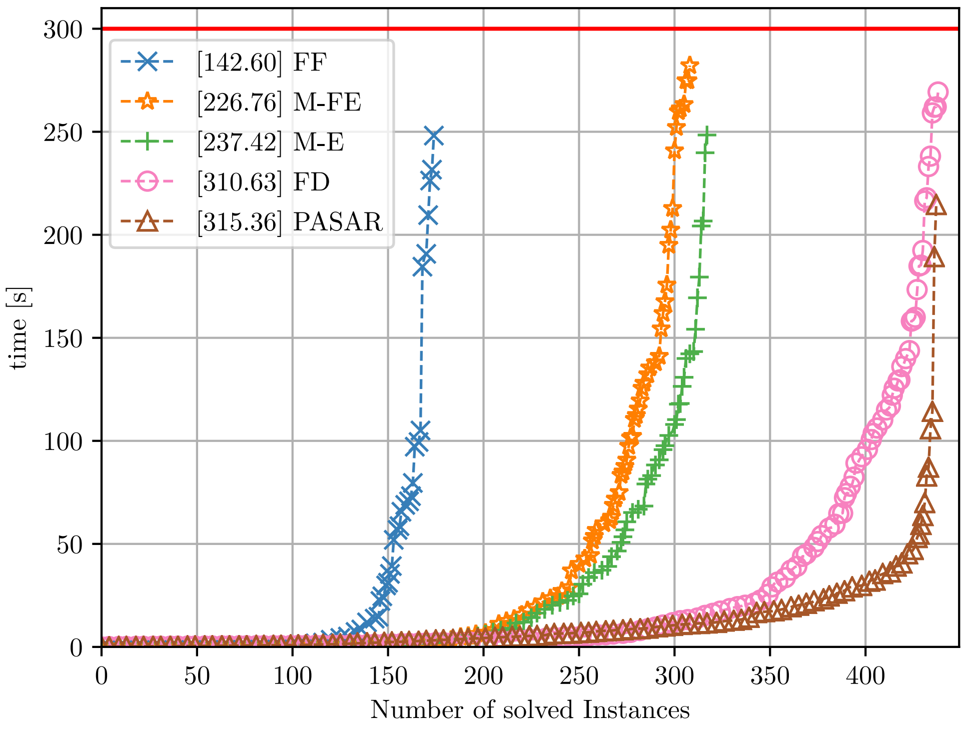


Figure 5.12: Comparison of the planners on the *IPC*-set.

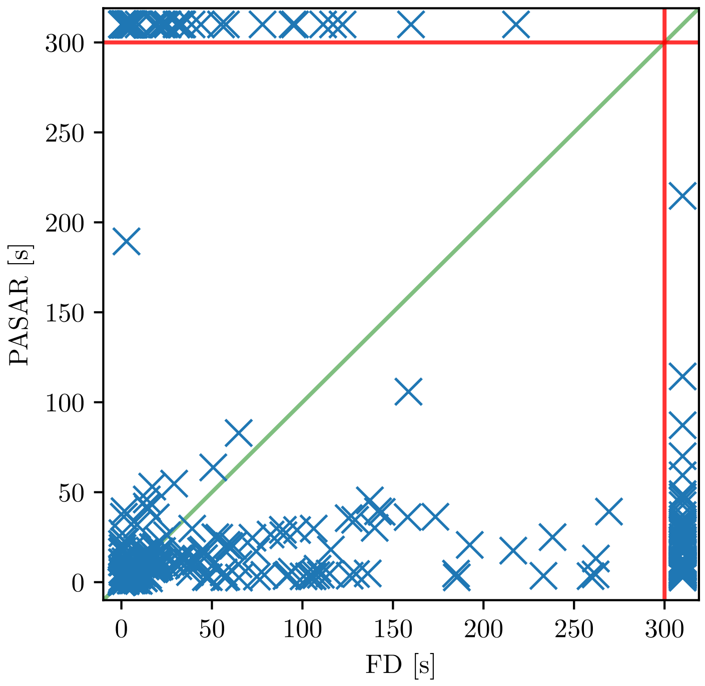


Figure 5.13: Comparison between Fast Downward and PASAR.

5.4.2 Validation Set

In the previous experiment, PASAR may have had an unfair advantage because we used the *IPC*-set to tune our planner. Therefore we repeat the experiment on the *sparkle*-set. The results are presented in figure 5.14 and table 5.7.

Fast Downward failed to ground 7 out of the 10 planning tasks in the *ChairGame* domain within the time limit. These instances are therefore lost to PASAR and Fast Downward itself. Fast-Forward only runs out of memory grounding 3 of the planning tasks. Solving the ground representation seems to be easy.

The planning tasks in *Pipegrid* are combinatorially complex and not too large. Therefore, the SAT-based approaches perform well. In contrast, PASAR is not able to solve a single abstraction for the tasks in *Parking* and the interleaving search with its basic heuristic is not successful either, while Fast Downward solves all instances.

The *pizza* domain seems to be hard for all tested planners. PASAR solves the two smallest instances. When solving the other planning tasks in the domain, the same-makespan-limit is reached and generating all interference clauses is sufficient to exhaust the available memory. Overall, PASAR again achieves the highest score.

Domain	M-FE	M-E	FD	FF	PASAR
Agricola (10)	0	1	5	0	**6**
ChairGame (10)	3	3	3	**7**	3
Parking (10)	2	3	**10**	3	0
Pipegrid (10)	9	**10**	0	0	**10**
Termes (10)	0	0	**5**	0	1
UTC-distribution (10)	**10**	**10**	**10**	**10**	**10**
pizza (10)	0	0	0	0	**2**
Sum (70)	24	27	**33**	20	32
Score	14.51	16	16.16	16.23	**17.62**

Table 5.7: Number of solved instances for each domain in the *IPC*-set.

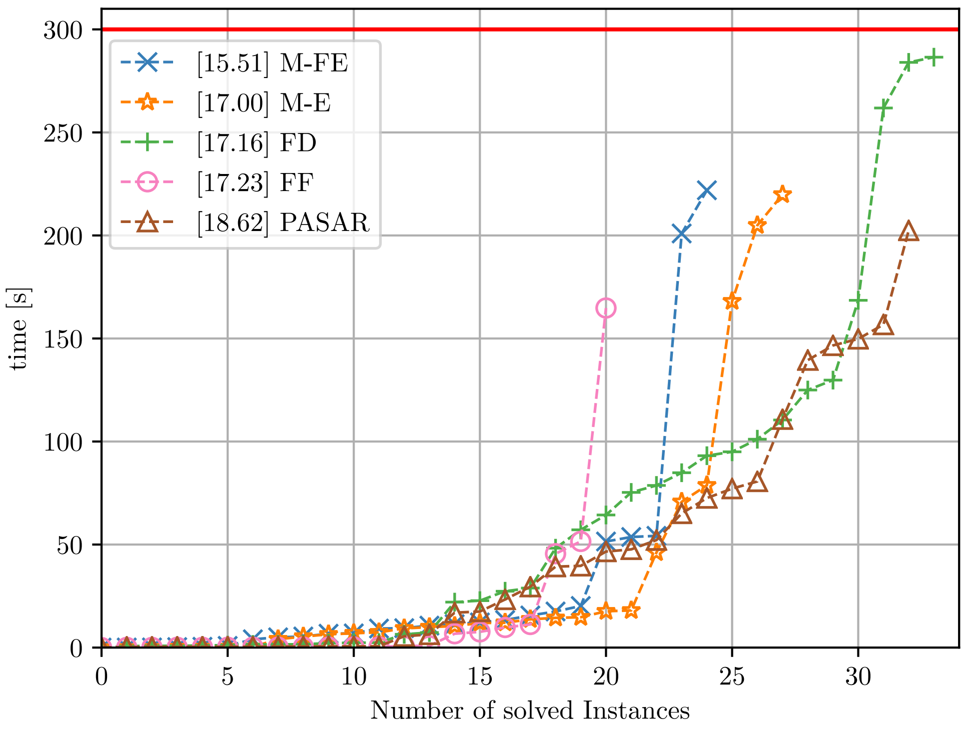


Figure 5.14: Comparison of the planners on the *sparkle*-set.

6 Conclusion

We conclude the thesis with a brief summary of our results and give an overview of possible future work to extend PASAR.

6.1 Summary

In this thesis we built a hybrid planner that combines SAT-based planning and searching in the state space in a novel way. We demonstrated how incremental SAT solvers can be used interactively in planning. A number of challenges in our theoretical approach have been identified and addressed.

In addition, we have introduced two new SAT encodings for the classical planning problem which deal with action interference in a novel way.

Thorough evaluation of PASAR on a broad range of commonly used benchmark domains has been conducted. It shows that PASAR can compete with SAT-based planners and heuristic searches. On many domains it matches the performance of the best tested solvers and on some domains it even outperforms the competition.

A previous version of this planner competed in the Sparkle Planning Challenge 2019. Although it had a significant contribution to the final portfolio, it did not perform well on its own. Our evaluation suggests that we mended this shortcoming, while retaining what made the previous version novel and relevant for use in conjunction with other planners.

6.2 Future Work

In the following an outlook on possible future work is provided.

Conditional Effects The integration of conditional effects into our planning approach would make PASAR useful for a wider range of realistic planning domains. The biggest challenge is that the ordering of action sets can no longer be done with a simple graph-theoretic test. However, since the action sets are usually small, an exhaustive search could be used to replace it.

Normalizing Abstract Plans Besides ∀-step and ∃-step semantics, Rintanen, Heljanko, and Niemelä [RHN06] introduced *process* semantics. A parallel plan fulfills process semantics if every action is applied in the earliest possible step.[1] We can transform an abstract plan into one fulfilling process semantics to normalize it. The hope is that this will make sections of the abstract plan more similar to others and thus increases the use of learned actions.

Abstractions Our abstraction is very simple. We have focused exclusively on action interference.

The most successful SAT-encodings for planning are *relaxed* [WR07a]. They allow the execution of actions in a step even if their preconditions are not fulfilled after executing the last step. Even further relaxed encodings [Bal13] additionally allow the effects of actions to be negated in the same step. Their advantage is that more actions can be executed in the same step. All encodings that use this idea impose a fixed order on the actions that is independent of the step. While this reduces parallelism, it is necessary to obtain an efficient encoding. With our CEGAR based approach we do not need such a fixed order.

Even coarse abstractions could be useful. Allowing variable to have multiple values at once is something akin to the concept of *delete relaxation* [Hof01] in our formalism. It would allow solving abstractions of planning tasks that are currently too difficult.

[1]In addition, the plan is required to meet ∀-step semantics. We will ignore that for our purposes.

A Complexity of Planning

The word problem for the language PlanMin =
$\{\langle \Pi, N\rangle \in \Psi^* \mid$ A plan of length N or less exists for the planning task Π.$\}$
is PSPACE-complete.

Proof. We need to prove that PlanMin is in PSPACE and that PlanMin is PSPACE-hard. We will show that PlanMin $\in$ NPSPACE by describing a non-deterministic Turing machine that can solve PlanMin-instances with polynomial space complexity. Since NPSPACE = PSPACE [Sav70], PlanMin is also in PSPACE.

Let the planning task $\Pi = (X, O, s_I, s_G)$ and $N \in \mathbb{N}$ be an instance of PlanMin. The initial configuration of the Turing machine encodes the PlanMin-Instance in a suitable encoding. The Turing machine will mainly edit the counter for N and the encoding of s_I, referred to as s in the following. This encoding could be realized with a binary number for each variable, which represents the index of its value in its domain. Note that every variable is assigned a value in s_I.

The Turing machine repeats the following program:

(i) If the current state s fulfills s_G, accept the input.
(ii) If N equals zero, reject the input, otherwise decrement the counter by one.
(iii) Non-deterministically choose an action $a \in O$.
(iv) If a is not applicable in the current state s, reject the input.
(v) Change the current state $s = s \oplus \text{eff}(a)$.

The Turing machine can non-deterministically choose a sequence of applicable actions to reach the goal before the counter reaches zero, exactly if a plan of length N or less exists. All operations can be implemented with a constant factor of additional space.

To prove that PlanMin is PSPACE-hard, we show that for each $L \in \text{PSPACE} \subseteq 2^{\Sigma^*}$ a function $f_L : \Sigma^* \to \Psi^*$ exists with:

- $f_L(w) \in$ PlanMin $\Leftrightarrow w \in L$,
- f_L can be computed in polynomial time.

Without loss of generality let $T_L = (Q, q_0, q_{YES}, \Gamma, \perp, \Sigma, \delta)$ be a Turing machine, which decides L with polynomial space complexity. Q is a finite set of states, $q_0 \in Q$ the initial state, $q_{YES}, \in Q$ the only states to signal acceptance of the input, Γ a finite set of tape symbols, $\perp \in \Gamma$ the blank symbol, $\Sigma \subseteq \Gamma \setminus \{\perp\}$ a set of input symbols, and

$\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ the transition function. Let S_{T_L} be a polynomial; $S_{T_L}(|w|)$ is an upper bound on the number of cells T_L will visit with input w. $|w|$ denotes the length of the word w.

For a fixed language $L \in$ PSPACE, the function f_L constructs an instance of the PlanMin word problem $\langle \Pi = (X, O, s_I, s_G), N \rangle$ from an input w. The construction is independent of the given input w, except for the initial state s_I and $M := S_{T_L}(|w|)$ the maximum space used by the Turing machine. First we define X. We have one variable for each cell that may be visited by the head of the Turing machine: $t_1, \ldots, t_M$ and $\mathrm{dom}(t_i) = \Gamma$. Additionally we have a variable for the position of the head $h, \mathrm{dom}(h) = \{1, \ldots, M\}$ and one for the current state of the Turing Machine $q, \mathrm{dom}(q) = Q$. In the initial state s_I every tape variable is set to $\perp$ except for those corresponding to the initial input w of the Turing machine. In addition $q = q_0$ and $h = 1$ are set. For the goal we have $s_G = \{q = q_{YES}\}$.

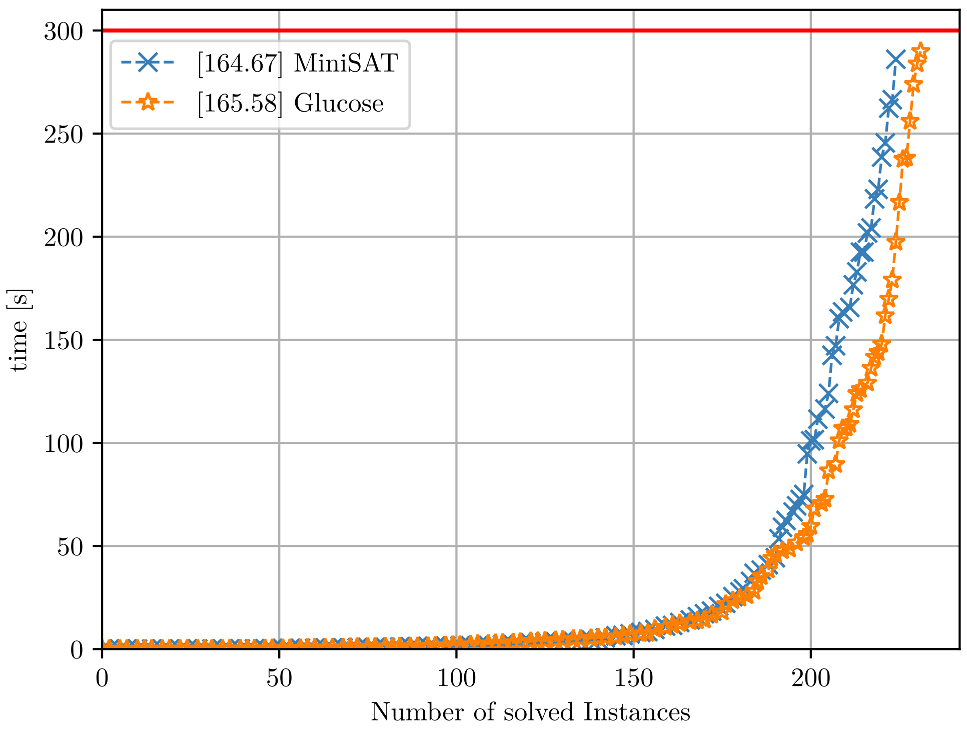
For $\delta(q_i, a) = (q_j, b, d)$ and $k \in \{1, \ldots, M\}$ we add an action $(\{q = q_i, t_k = a, h = k\}$, $\{q = q_j, t_k = b, h = k + d\})$ to O. The number of operators is therefore in $\mathcal{O}(M)$, since the number of transitions is constant for a fixed Turing machine T_L.

We set $N := \Pi_{x \in X} |\mathrm{dom}(x)| = |\Gamma|^M \cdot M \cdot |Q|$. This is an upper bound on the number of configurations the Turing machine can be in and on the number of complete states that are induced by the planning task. A plan with a length exceeding N must visit one state twice and therefore have a loop which can be removed to get a shorter plan. Since N is encoded logarithmically, the computation can be done polynomially in M.

$S_{T_L}(|w|)$ can be evaluated polynomially in $|w|$ and the rest of the construction can be implemented polynomially in M. Since M is bound polynomially in $|w|$, f_L can be computed in polynomial time.

A solution to the PlanMin-Instance directly induces a path through the configuration graph of the Turing machine from the initial to a final configuration. Conversely, if there is such a path in the configuration graph a plan can be found. It is interesting to note that for a deterministic Turing machine only one action will be applicable in each state during the planning process. $\square$

B Reevaluating SAT Solvers



SAT solver	Solved	Score
MiniSAT	224	163.67
Glucose	**231**	**164.57**

Figure B.1 & Table B.1: Comparison of SAT solvers for the use with pure PASAR. The conflicts per makespan are limited to 20000.

The results shown in table B.2 are not directly applicable because a wall-time-based time out is used to abort a makespan instead of limiting the conflicts. This is done to allow the comparison to all supported SAT solvers.

SAT solver	Solved	Score
PicoSAT	438	376.87
Lingeling	441	372.01
MiniSAT	443	**382.61**
Glucose	**444**	381.38

Table B.2: Comparison of SAT solvers for the use with the final configuration. A time out of 15 seconds per makespan is used instead of limiting the conflicts.

C Test Sets

The following domains have been used in the test set:

C.1 IPC-Set

IPC 2014 Optimal Track
`https://helios.hud.ac.uk/scommv/IPC-14/repository/benchmarksV1.1.zip`

domain	tasks
barman	14
childsnack	20
floortile	20
ged	20
hiking	20
openstacks	20
tetris	17
tidybot	20
transport	20
visitall	20

IPC 2014 Satisficing Track
`https://helios.hud.ac.uk/scommv/IPC-14/repository/benchmarksV1.1.zip`

domain	tasks
barman	20
childsnack	20
ged	20
openstacks	20
tetris	20
thoughtful	20
transport	20
visitall	20

IPC 2018 Optimal Track
`https://bitbucket.org/ipc2018-classical/domains/src/default/opt`

domain	tasks
data-network	20
organic-synthesis-split	20
petri-net-alignment	20
snake	20
termes	20

IPC 2018 Satisficing Track
`https://bitbucket.org/ipc2018-classical/domains/src/default/sat`

domain	tasks
data-network	20
organic-synthesis-split	20
snake	20
termes	20

C.2 Sparkle-Set

IPC 2018 Satisficing Track
`http://ada.liacs.nl/events/sparkle-planning-19/documents/benchmark/sparkle_planning_challenge_2019_testing_set.zip`

domain	tasks
Agricola	10
ChairGame	10
Parking	10
Pipegrid	10
Termes	10
UTC-distribution	10
pizza	10

Bibliography

[AS09] Gilles Audemard and Laurent Simon. “Predicting learnt clauses quality in modern SAT solvers”. In: *Twenty-first International Joint Conference on Artificial Intelligence*. 2009, pp. 1–6.

[Bal+16] Tomas Balyo et al. “SAT Race 2015”. In: *Artificial Intelligence* 241 (2016), pp. 45–65. DOI: 10.1016/j.artint.2016.08.007.

[Bal13] Tomas Balyo. “Relaxing the Relaxed Exist-Step Parallel Planning Semantics”. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE Computer Society, 2013, pp. 865–871. DOI: 10.1109/ICTAI.2013.131.

[BDG17] Martin Brain, James H. Davenport, and Alberto Griggio. “Benchmarking Solvers, SAT-style”. In: *Proceedings of the 2nd International Workshop on Satisfiability Checking and Symbolic Computation 2017*. 2017, pp. 1–15. URL: http://ceur-ws.org/Vol-1974/RP3.pdf.

[BF97] Avrim Blum and Merrick L. Furst. “Fast Planning Through Planning Graph Analysis”. In: *Artificial Intelligence* 90.1-2 (1997), pp. 281–300.

[Bie+09] Armin Biere et al. “Conflict-driven clause learning sat solvers”. In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009), pp. 131–153.

[Bie08] Armin Biere. “PicoSAT essentials”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 75–97.

[Bie17] Armin Biere. “CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017”. In: *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Marijn Heule, and Matti Järvisalo. Vol. B-2017-1. Department of Computer Science Series of Publications B. University of Helsinki, 2017, pp. 14–15.

[BN95] Christer Bäckström and Bernhard Nebel. “Complexity Results for SAS+ Planning”. In: *Computational Intelligence* 11 (1995), pp. 625–656.

[Cha+05] Krishnendu Chatterjee et al. “Counterexample-guided Planning”. In: *Twenty-First Conference on Uncertainty in Artificial Intelligence*. UAI’05. Edinburgh, Scotland: AUAI Press, 2005, pp. 104–111. ISBN: 0-9749039-1-4. URL: http://dl.acm.org/citation.cfm?id=3020336.3020349.

[Cla+00] Edmund Clarke et al. “Counterexample-guided abstraction refinement”. In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 154–169.

[Coo71] Stephen A. Cook. “The Complexity of Theorem-proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: `10.1145/800157.805047`. URL: `http://doi.acm.org/10.1145/800157.805047`.

[ES03] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *SAT*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518. ISBN: 3-540-20851-8.

[FBS19] Nils Froleyks, Tomas Balyo, and Dominik Schreiber. “PASAR—Planning as Satisfiability with Abstraction Refinement”. In: *Twelfth Annual Symposium on Combinatorial Search*. 2019, pp. 70–78.

[Flo+13] José Florez et al. “Combining linear programming and automated planning to solve intermodal transportation problems”. In: *European Journal of Operational Research* 227 (May 2013), pp. 216–226. DOI: `10.1016/j.ejor.2012.12.018`.

[FN71] 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.

[Fuk+97] Alex Fukunaga et al. “ASPEN: A framework for automated planning and scheduling of spacecraft control and operations”. In: *Proc. International Symposium on AI, Robotics and Automation in Space*. 1997, pp. 181–187.

[GB17] Stephan Gocht and Tomáš Balyo. “Accelerating SAT based planning with incremental SAT solving”. In: *Twenty-Seventh International Conference on Automated Planning and Scheduling*. 2017, pp. 1–5.

[GNT16] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016, pp. 30–36.

[Hel06] Malte Helmert. “The Fast Downward Planning System”. In: *Journal of Artificial Intelligence Research (JAIR)* 26 (2006), pp. 191–246.

[Hof01] Jörg Hoffmann. “FF: The fast-forward planning system”. In: *AI magazine* 22.3 (2001), pp. 57–57.

[Hut+09] Frank Hutter et al. “ParamILS: An Automatic Algorithm Configuration Framework”. In: *Journal of Artificial Intelligence Research* 36 (Oct. 2009), pp. 267–306.

[KS92] Henry A. Kautz and Bart Selman. “Planning as Satisfiability”. In: *Tenth AAAI Conference on Artificial Intelligence.* 1992, pp. 359–363.

[KS99] Henry Kautz and Bart Selman. “Unifying SAT-based and graph-based planning”. In: *IJCAI.* Vol. 99. 1999, pp. 318–325.

[LS04] Daniel Le Berre and Laurent Simon. “The Essentials of the SAT 2003 Competition”. In: *Theory and Applications of Satisfiability Testing.* Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 452–467. ISBN: 978-3-540-24605-3.

[NM10] Hootan Nakhost and Martin Müller. “Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement.” In: *ICAPS 2010 - Proceedings of the 20th International Conference on Automated Planning and Scheduling.* Jan. 2010, pp. 121–128.

[RGP08] Nathan Robinson, Charles Gretton, and Duc-Nghia Pham. “Co-plan: Combining SAT-based planning with forward-search”. In: *Proceedings of the 6th International Planning Competition.* 2008, pp. 1–2.

[RHN06] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. “Planning as satisfiability: parallel plans and algorithms for plan search”. In: *Artificial Intelligence* 170.12-13 (2006), pp. 1031–1080. DOI: `10.1016/j.artint.2006.08.002`.

[Rin12] Jussi Rintanen. “Planning as satisfiability: Heuristics”. In: *Artificial Intelligence* 193 (2012), pp. 45–86. DOI: `10.1016/j.artint.2012.08.001`.

[Rin14] Jussi Rintanen. “Madagascar: Scalable planning with SAT”. In: *Proceedings of the 8th International Planning Competition (IPC-2014)* 21 (2014), pp. 1–5.

[RP12] Purushothaman Raja and Sivagurunathan Pugazhenthi. “Optimal path planning of mobile robots: A review”. In: *International journal of physical sciences* 7.9 (2012), pp. 1314–1320.

[Sav70] Walter J. Savitch. “Relationships between nondeterministic and deterministic tape complexities”. In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/S0022-0000(70)80006-X`. URL: `http://www.sciencedirect.com/science/article/pii/S002200007080006X`.

[Sei12] Jendrik Seipp. “Counterexample-guided abstraction refinement for classical planning”. MA thesis. 2012, pp. 1–55.

[Sei18] Jendrik Seipp. “Fast downward remix”. In: *Ninth International Planning Competition Booklet (IPC 2018)* (2018), pp. 67–69.

[Sei19] Jendrik Seipp. "Planner Description Kronk". 2019. URL: `http://ada.liacs.nl/events/sparkle-planning-19/documents/solver_description/seipp-sparkle2019.pdf`.

[Tar76] Robert Endre Tarjan. "Edge-disjoint spanning trees and depth-first search". In: *Acta Informatica* 6.2 (June 1976), pp. 171–185. ISSN: 1432-0525. DOI: `10.1007/BF00268499`. URL: `https://doi.org/10.1007/BF00268499`.

[WR07a] Martin Wehrle and Jussi Rintanen. "Planning as Satisfiability with Relaxed Exist-Step Plans". In: *Australian Conference on Artificial Intelligence*. Ed. by Mehmet A. Orgun and John Thornton. Vol. 4830. Lecture Notes in Computer Science. Springer, 2007, pp. 244–253. ISBN: 978-3-540-76926-2.

[WR07b] Martin Wehrle and Jussi Rintanen. "Planning as Satisfiability with Relaxed exist-Step Plans". In: *AI 2007: Advances in Artificial Intelligence*. Ed. by Mehmet A. Orgun and John Thornton. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 244–253. ISBN: 978-3-540-76928-6.

[Zob70] Albert L Zobrist. "A new hashing method with application for game playing". In: *ICCA journal* 13.2 (1970), pp. 69–73.