Abstract

It is often natural in planning to specify conditions that should be avoided, characterizing dangerous or highly undesirable behavior. PDDL3 supports this with temporal-logic state constraints. Here we focus on the simpler case where the constraint is a non-temporal formula $\phi$ – the avoid condition – that must be false throughout the plan. We design techniques tackling such avoid conditions effectively. We show how to learn from search experience which states necessarily lead into $\phi$, and we show how to tailor abstractions to recognize that avoiding $\phi$ will not be possible starting from a given state. We run a large-scale experiment, comparing our techniques against compilation methods and against simple state pruning using $\phi$. The results show that our techniques are often superior.

1 Introduction

It is often natural in planning to specify conditions that should be avoided. Work along these lines has so far focused on temporal-logic formulas that must be true in the state sequence induced by the plan. One prominent early approach used such formulas as control knowledge for effective hand-tailored planning [Bacchus and Kabanza, 2000; Doherty and Kvarnström, 2001]. The PDDL3 language [Gerevini et al., 2009] features temporal formulas (among others) in the role of state constraints. Work since then has devised compilation techniques [Edelkamp, 2006; Baier and McIlraith, 2006; De Giacomo et al., 2014; Torres and Baier, 2015], and investigated how to effectively deal with (soft-goal) temporal plan preferences [Baier et al., 2007; Baier et al., 2009].

Here we focus on the simpler case where the state constraint is a non-temporal formula $\phi$ that must be false throughout the plan. We refer to such constraints as avoid conditions. This special case is relevant as avoid conditions naturally characterize dangerous or highly undesirable situations. For example, an avoid condition can capture states the user knows to be dead-ends; or risky states in a deterministic approximation of a probabilistic planning application.

Avoid conditions can be trivially compiled into preconditions, but this incurs a large overhead and is, as our experiments illustrate, often not effective. Our contribution consists in advanced algorithmic methods. Apart from several compilation techniques, we adapt prior work in classical planning to design a method learning from the avoid condition during search, and a method using abstraction to predict states starting from which $\phi$ cannot be avoided.

Our learning method is based on so-called trap learning. Traps [Lipovetzky et al., 2016] are sets of states from which there is no path to the goal. One can generalize from traps—and thus prune future search states—by minimizing the states to retain only the core reason why an escape is not possible. By seeding traps with dead-end states encountered during forward search, one can incrementally extend the traps as a form of nogood learning [Steinmetz and Hoffmann, 2017a]. Observing that traps correspond to DNF formulas, we transform $\phi$ to DNF (using the Fast Downward pre-process [Helmert, 2009]) and use it as the seed for trap learning.

Our abstraction method is a form of state abstraction, a wide-spread method used to design heuristic functions in planning [Edelkamp, 2001; Helmert et al., 2014; Seipp and Helmert, 2018]. Abstract state spaces group concrete states $s$ into block states $A$. Observe that, given such an abstract state space, we know that $s \in A$ can be pruned if all paths from $A$ to a goal block traverse a block $A'$ where $s' \models \phi$ for all $s' \in A'$. In other words, given an abstraction, we can predict that every plan for a state $s$ will necessarily traverse the avoid condition. The question remains how to tailor abstractions for this purpose. To this end, we leverage so-called Cartesian abstractions and their associated counter-example guided abstraction refinement (CEGAR) process [Seipp and Helmert, 2013; Seipp and Helmert, 2018]. We modify the CEGAR process to incorporate $\phi$ as an additional source of counter-examples and, therewith, of refinement steps.

We run a large-scale experiment on satisficing planning, optimal planning, and proving unsolvability, evaluating compilations, learning, and abstraction. We do so on (a) reformulated standard benchmarks that incorporate aspects (more) naturally formulated as avoid conditions; (b) benchmarks involving road maps (or similar), where we systematically impose avoid conditions of the form “do not use particular combinations of road segments”; and (c) a small collection of benchmarks where we generated avoid conditions automatically using trap learning (which is a bit artificial and merely serves as a showcase). The results show that our new methods can be superior, in particular for proving unsolvability.
2 Preliminaries

We consider classical planning tasks in FDR notation [Bäckström and Nebel, 1995; Helmer, 2006]. A planning task is a tuple \( T = (V, A, I, G) \). \( V \) is a set of variables, each \( v \in V \) has a finite domain \( D_v \). A fact is variable assignment \( p = (v, d) \) for \( v \in V \) and \( d \in D_v \). The initial state \( I \) is a complete assignment of \( V \). The goal \( G \) is partial assignment of \( V \). For a variable \( v \in V \), the set of variables for which \( P(v) \) is defined. For \( v \in (V \setminus P(v)) \), we write \( P(v) = \perp \). \( A \) is a set of actions. Each action \( a \in A \) has a precondition \( pre_a \) and an effect \( eff_a \), both partial variable assignments, and a non-negative cost \( c_a \in \mathbb{R}^+_0 \). The states \( S \) of \( T \) are all complete variable assignments. An action \( a \) is applicable in a state \( s \) if \( s(v) = pre_a(v) \) for all \( v \in V(pre_a) \). The result is given by \( s[a] \) where \( s[a](v) = eff_a(v) \) for all \( v \in V(eff_a) \), and \( s[a](v) = s(v) \) for the other variables. These definitions are extended to sequences of actions in an obvious manner. A plan for \( s \) is a sequence of actions \( \pi \) that is applicable in \( s \) and \( s[\pi](v) = G(v) \) for all \( v \in V(G) \). An optimal plan is a plan with minimal summed up action cost. \( s \) is called a dead end if there is no plan for \( s \). A plan for \( I \) is a plan for \( T \). \( T \) is called unsolvable if \( I \) is a dead end.

An avoid condition \( \phi \) for \( T \) is an arbitrary propositional formula over \( T \)'s facts. A plan \( a_1, \ldots, a_n \) for \( T \) is called \( \phi \)-compliant if \( I \not\models \phi \), and it holds for all \( 1 \leq i \leq n \) that \( I[a_1, \ldots, a_i] \not\models \phi \). An optimal \( \phi \)-compliant plan is a \( \phi \)-compliant plan with minimal action cost. We say that a state \( s \) is \( \phi \)-unsolvable if there is no \( \phi \)-compliant plan for \( s \).

3 Compilations

Compiling avoid conditions into the planning task is straightforward in principle, but the naive method is very ineffective so it is worth thinking this through more carefully. Furthermore, compilations for temporal plan constraints are well known and we address a special case here. Hence we evaluate three compilation methods in our experiments. All these compilations operate at the PDDL input level.

Conditions Compilation The first, and most straightforward, compilation ensures \( \phi \)-compliance by conjoining \( \neg \phi \) to the preconditions of all actions and the goal. We denote by \( \Pi^{\neg \phi} \) the resulting FDR planning task. Trivially, the plans of \( \Pi^{\neg \phi} \) are the \( \phi \)-compliant plans of \( T \).

LTL Compilation Our second method uses existing tools for compiling temporal formulas into planning tasks [Edelkamp, 2006; Baier and McIraith, 2006]. This usually works in two steps: (1) building an automaton representation of the formula, and (2) encoding this automaton into the planning task via additional state variables and actions. For our simple LTL formula \( G(\neg \phi) \) (always not \( \phi \)), the automaton representation will always consist of exactly two locations. The initial location is accepting and has a self-loop conditioned by \( \neg \phi \). The other location is not accepting, and is reached from the initial location if \( \phi \) is satisfied. We denote by \( \Pi^{LTL} \) the compilation of this automaton into \( T \). \( \Pi^{LTL} \) enforces an update of the automaton location in between applications of actions from \( T \). The automaton “blocks” as soon as it leaves its accepting state. Discarding the automaton-related actions, the plans of \( \Pi^{LTL} \) are exactly the \( \phi \)-compliant plans of \( T \). Moreover, plan optimality is not affected provided the newly introduced actions have 0 cost.

Axiom Compilation Both \( \Pi^{\neg \phi} \) and \( \Pi^{LTL} \) suffer from the use of \( \neg \phi \), which causes a blow-up in FDR’s translator if \( \phi \) is a DNF (because \( \neg \phi \) is a CNF which the translator naively transforms into a DNF). This motivates our last compilation, which employs derived predicates, aka axioms [Hoffmann and Edelkamp, 2005], to avoid that problem.

Axioms are defined by rules of the form \( p \leftarrow \psi_p \). The fact \( p \) must not be affected by any action, i.e., its truth value must be completely determined by the axioms. In the simple (non-recursive) form of axioms that we need for our compilation, \( p \) is true in a state iff the state satisfies one of its associated rule conditions \( \psi_p \). To enforce \( \neg \phi \) with axioms, we introduce a rule (avoid) \( \leftarrow \phi \), and conjoin \( \neg \text{(avoid)} \) to the preconditions of every action and to the goal. We denote by \( \Pi^{A} \) the resulting FDR task with axioms.

4 Trap Learning

The basic algorithm we assume for our new advanced techniques is forward search on \( T \), while pruning all states that satisfy \( \phi \). On top of this simple baseline, in what follows we introduce methods that can identify additional \( \phi \)-unsolvable states: trap learning in this section, abstraction in the next. Both methods preserve completeness (returning a \( \phi \)-compliant plan if one exists) and optimality (returning an optimal \( \phi \)-compliant plan).

4.1 Background: Traps

Traps have been proposed in classical planning for dead end detection [Lipovetzky et al., 2016]. Formally, a trap is a set of states \( T \subset S \) that (1) is goal-disjoint, and (2) is closed under transitions. Both conditions together ensure that the states in \( T \) are dead ends. Steinmetz and Hoffmann [2017a] have proposed an algorithm that incrementally builds a trap \( T_\Psi \) during a depth-first search from experience made in that search. At the same time, \( T_\Psi \) is used for pruning dead ends during the search. \( T_\Psi \) is characterized through a DNF formula over facts \( \Psi \), i.e., \( T_\Psi = \{ s \in S | s \models \Psi \} \). The procedure starts with the empty trap: \( \Psi = \perp \). \( \Psi \) is updated whenever the depth-first search backtracks out of a state \( s \) without having found the goal. At this point, all children of \( s \) must already be represented by \( \Psi \). Therefore, \( T_\Psi \cup \{ s \} \) still satisfies the trap definition. The trap refinement then aims to find a suitable extension \( \Psi' := \Psi \lor \psi_s \) to cover also \( s \). Every new state represented by \( \Psi' \) in addition to \( s \) may reduce work in the remainder of the search. This generalization is achieved by starting with the conjunction \( \psi_s := \psi_s \), and incrementally removing as many facts from \( \psi_s \) as possible, while \( T_{\Psi'} \) still satisfies the trap conditions. The latter can be tested efficiently based on the DNF structure of the formula.

4.2 Tailoring To Avoid Condition

We call a set of states \( T \subset S \) a \( \phi \)-trap if (1) every goal state in \( T \) satisfies \( \phi \); and (2) every transition that leaves \( T \) either originates in a state satisfying \( \phi \), or goes into one
that does. These conditions are weaker than their counterparts in the original trap definition. However, they are strong enough to guarantee that once a \( \phi \)-trap has ever been entered, it is no longer possible to reach the goal without satisfying \( \phi \) beforehand. In other words, \( \phi \)-traps imply \( \phi \)-unsolvability.

To compute and to use \( \phi \)-traps, we modify the learning algorithm from above accordingly. The only change required to make this work is changing the initialization of \( \Psi \) to \( \phi \). To match the expected trap format, this requires to bring \( \phi \) into DNF. Since every trap refinement only makes the trap larger, \( \phi \) implies \( \Psi \) at any point in time. So pruning by \( \Psi \) alone already guarantees \( \phi \)-compliance. The initialization is valid since \( \Psi = \phi \) trivially satisfies the \( \phi \)-trap definition. The original refinement method does not require changes. The initialization guarantees the prerequisites, conditions (1) and (2), in every refinement. (1) holds because whenever a goal state is found, it either satisfies \( \phi \) (no need to refine \( \Psi \)), or a \( \phi \)-compliant plan has been found (search terminates). (2) is retained since states can only be pruned if they satisfy \( \Psi \).

While we have sketched only depth-first search here, trap learning is not limited to that. The learning algorithm can be integrated into other search algorithms (like A* and greedy best-first search) as well [Steinmetz and Hoffmann, 2017b]. Their modifications work directly for \( \phi \)-trap learning.

## 5 Abstraction for Avoid-Prediction

We recall Cartesian abstractions and show how to tailor them to the identification of \( \phi \)-unsolvable states.

### 5.1 Background: Cartesian Abstractions

An abstraction for \( \Pi \) is an equivalence relation \( \sim \) between the states \( S \). The abstract states \( S' \) of \( \sim \) are given by its equivalence classes. For state \( s \), we denote by \( [s]_\sim \) the equivalence class that contains \( s \), and omit \( \sim \) if it is clear from the context. The abstract state space associated with \( \sim \) is the transition system \( (S', \mathcal{T}', s'_0, \mathcal{G}'_0) \) with abstract initial state \( s'_0 = [s_0]_\sim \) and abstract goal states \( \mathcal{G}'_0 = \{[s] | s \in S, G \subseteq s \} \). The abstract transitions are given by \( \mathcal{T}' = \{([s], a, [s']) | s \in S, a \in A \text{ applicable in } s \} \).

Let the variables of \( \Pi \) be \( \mathcal{V} = \{v_1, \ldots, v_N\} \). Cartesian abstractions [Seipp and Helmart, 2018] are abstractions whose abstract states are of the form \( A_1 \times A_2 \times \cdots \times A_N \), where \( A_i \subseteq D_i \) for all \( i \).

This structure makes Cartesian abstractions particularly suited for a counter-example guided refinement loop (CEGAR): The construction starts with the trivial abstraction that contains just a single abstract state. One then iteratively splits an abstract state into two until the abstraction provides enough information, or some size limit is reached. Each refinement step starts with the extraction of an abstract solution, i.e., an abstract path \( [s_0], a_1, [s_1], \ldots, a_n, [s_n] \) from the abstract initial state \( [s_0] = s'_0 \) to some abstract goal state \( [s_n] \in \mathcal{G}'_0 \). If no such path exists, then \( \Pi \) must be unsolvable, and the refinement terminates. Otherwise, the corresponding concrete path \( s_0, a_1, s_1, a_2, \ldots \) is computed by applying the actions successively, starting from \( s_0 = I \). The computation is stopped when one of the following conditions is satisfied:

\[ (C1) \text{ Concrete and abstract state do not match: } [s'_i] \neq [s_i]. \]
\[ (C2) \text{ Action } a_i \text{ is not applicable in } s'_{i-1}. \]
\[ (C3) \text{ } s'_{i} \text{ does not satisfy the goal.} \]

If not stopped, we have found a plan for \( \Pi \) and the refinement terminates. Otherwise, the violated condition is used to split an abstract state, guaranteeing that the same error cannot occur in future iterations (\( \cup \) denotes disjoint union):

\[ (C1) \ [s_i-1] \text{ is split into } [t_1] \cup [t_2] \text{ such that } s'_{i-1} \in [t_2] \text{ and } [t_2] \text{ no longer has an abstract transition to } [s_i] \text{ via } a_i. \]
\[ (C2) \ [s_i-1] \text{ is split into } [t_1] \cup [t_2] \text{ such that } s'_{i-1} \in [t_2] \text{ and } [t_2] \text{ has no abstract transition via } a_i. \]
\[ (C3) \ s_n \text{ is split into } [t_1] \cup [t_2] \text{ such that } s'_{n} \in [t_2] \text{ and } [t_2] \text{ is no longer an abstract goal state.} \]

The selection of \( [t_1] \) and \( [t_2] \) is done via simple syntactic checks. During the entire construction, a full representation of the abstract state space is maintained. After each split, this representation can be updated efficiently by “rewiring” transitions to \( [t_1] \) and \( [t_2] \). For full details, we refer to the work by Seipp and Helmart [2018]. Once the abstract state space has been updated, a new abstract solution is extracted, and the whole process starts anew.

### 5.2 Tailoring to Avoid Conditions

Abstractions can be used to identify states where \( \phi \) can no longer be avoided. Say that an abstract state \([s] \) implies \( \phi \), written \([s] \Rightarrow \phi \), if all represented concrete states \( s' \in [s] \) satisfy \( \phi \). If every goal path of an abstract state \([t]\) contains some \([s] \Rightarrow \phi \), then we know that there can be no \( \phi \)-compliant plan for any represented concrete state \( t \) either. Hence, \( t \) is \( \phi \)-unsolvable. But how to check whether \([s] \Rightarrow \phi \) without enumerating all represented states? Can we design abstractions specifically for the purpose of identifying as many such \([t]\) as possible within the abstraction size limits? Here, we answer these questions for Cartesian abstractions.

CEGAR

We gear the CEGAR approach towards the identification of \( \phi \)-unsolvable states. The main procedure stays the same. To start a refinement, we change the solution extraction to consider only abstract goal paths \([s_0], a_1, [s_1], \ldots, a_n, [s_n] \) such that \([s_i] \not\Rightarrow \phi \) holds at all time. If such a path does not exist, then the abstraction already proves that \( \mathcal{I} \) is \( \phi \)-unsolvable. We stop immediately. To enable refinements based on \( \phi \), we introduce the additional error condition

\[ (C4) \text{ The concrete state } s'_i \text{ satisfies } \phi. \]
We use the following procedure to find \( [t_1] \cup \ldots \cup [t_k] \). Let \( v_1, \ldots, v_n \) be the variables whose sets \( A_j \) in \( [s] \) contain more than one value. We start with \( [t] = [s] \) and \( j = 1 \). Then \( [t] \) is split into two abstract states \( [t_j] \) and \( [t'] \) by dividing \( A_j \) into \( A_j \setminus \{ s_j(v_j) \} \) and \( \{ s_j(v_j) \} \). The other variable sets are left unchanged. If after the split \( [t'] \Rightarrow \emptyset \) holds, then we are done. Otherwise \( [t'] \) becomes the new \( [t] \), and we continue with the next variable \( v_{j+1} \). After the \( m \)-th iteration, \( [t'] \) will only contain \( s_j' \), so \( [t'] \Rightarrow \emptyset \) holds trivially. Hence, the procedure terminates eventually with the desired split \( [t_1], \ldots, [t_k] \).

While these changes fit well into the existing CEGAR approach, the overall construction might have a strong bias towards violations of (C1) to (C3). Especially in the early refinement iterations, when the abstraction is very coarse, the three base conditions can be expected to be the main reason for the refinements. Thus, the actual impact of our modifications could be comparatively low. To enforce \( t \) more strictly, we also tested the following more rigorous alternative. Before even starting to evaluate the error conditions, we check whether the solution path contains an abstract state \( [s] \) with \( t \in [s] \) and \( t \vdash \phi \). If this is the case, we refine \( [s] \) as above. And only otherwise, we continue with the analysis according to (C1) to (C3). A disadvantage of this variant is that the states \( t \vdash \phi \) have to be searched actively. This is computationally more expensive than the simple check in (C4).

Finally note that it is not possible to get rid of (C1)–(C3), and only use \( \phi \) for the refinements. As the path \( [s_0], a_3, [s_1] \) in the example in Figure 1 shows, goal paths in the abstraction can bypass \( \phi \). In the example, (C2) is violated. The corresponding refinement splits \( [s_0] \), separating the values of \( x \) into \( \{ 0 \} \) and \( \{ 1 \} \). This suffices to make all abstract goal paths pass through \( [s_2]\), proving that no \( \phi \)-compliant plan exists.

\[
\begin{align*}
[s_0]: \{ 0, 1 \} \times \{ 0 \} \times \{ 0 \} & \xrightarrow{a_3} [s_1]: \{ 0, 1 \} \times \{ 0 \} \times \{ 1 \} \\
[s_2]: \{ 0 \} \times \{ 0 \} \times \{ 0 \} & \xrightarrow{a_1} [s_3]: \{ 0 \} \times \{ 1 \} \times \{ 1 \}.
\end{align*}
\]

Figure 1: Illustration of an abstract state space. Self loops are omitted. The planning task consists of binary variable \( x, y, z \), goal \( z = 1 \), and three actions with \( \text{pre/eff: } y = 0/0 \) \( y = 1/x = 1 \) \( (a_1) \); \( y = 1/x = 1 \) \( (a_2) \); and \( x = 1/z = 1 \) \( (a_3) \). The abstract states are depicted in terms of \( A_x \times A_y \times A_z \). The avoid condition is \( \phi = (y = 1) \). Abstract states satisfying \( \phi \) have dashed borders, goal states have double borders.

**Implication Check**

The test \( [s] \Rightarrow \emptyset \) builds the basis for pruning as well as abstraction construction. Unfortunately, deciding this for Cartesian states turns out to be NP-hard. This is easy to see in the case where all state variables are Boolean, and \( [s] \) is the full Cartesian product. Then \( [s] \Rightarrow \emptyset \) holds exactly if \( \neg \phi \) is a tautology, deciding which is known to be NP-complete. Despite the worst-case complexity, the implication check was often not the bottleneck in our experiments. In the implementation, we check \( [s] \Rightarrow \emptyset \) by running a backtracking search for a state \( t \in [s] \) with \( t \vdash \neg \phi \). However, we do not enumerate the states \( t \) exhaustively, but select variable values specifically to invalidate \( \phi \). For \( \phi \) in DNF, this can be done by maintaining the subset of conjunctions that can still be satisfied, and incrementally choosing variable assignments making this subset smaller. A similar procedure can be used to find a state \( t' \in [s] \) with \( t' \vdash \phi \), as required for the refinement.

## 6 Experiments

We implemented all described methods in Fast Downward (FD) [Hellmert, 2006]. The avoid condition is specified as an additional input file in the full PDDL condition syntax. The compilations are implemented as part of FD’s translator component. All DNF conversions are done as part of the standard FD preprocessing. The source code and benchmarks will be made publicly available. The experiments were run on machines with an Intel Xeon E5-2650v3 processor, and cutoffs of 30 minutes and 4 GB memory.

We conducted experiments in optimal and satisficing planning, as well as proving unsolvability. For each category, we chose a canonical base planner configuration: optimal planning via \( A^* \) search with LMcut [Hellmert and Domshlak, 2009]; satisficing planning via greedy best-first search with two open lists and preferred operators using \( h_{FF} \) [Hoffmann and Nebel, 2001]; and proving unsolvability via depth-first search with \( h_{max} \) [Haslum and Geffner, 2000] for dead-end detection. We extended these base configurations by the following prediction methods: “-” no prediction, only prune by \( \phi \); “trap” \( \phi \)-trap learning; “A” Cartesian abstraction constructed via the original CEGAR approach; “PA” via the CEGAR approach with the additional (C4) check; and “SPA” using the CEGAR variant with the strict \( \phi \) refinement check. We experimented with abstraction size limits of \( N \in \{ 10k, 20k, 40k, 80k, 160k \} \) abstract states. We also tested trap learning and Cartesian abstractions for pruning dead-ends in the \( \Pi^0 \) and \( \Pi^{IL} \) compilations (not \( \Pi^N \) because neither of them supports axioms). We next describe our benchmarks, then discuss the results.

### 6.1 Benchmark Design

Benchmarks with avoid conditions already appeared in IPC 2006 [Dimopoulos et al., 2006], encoded via state constraints. But hard state constraints were only used in benchmarks of the temporal track, and the constraints themselves heavily relied on temporal operators, which makes them unsuited for our experiments. Instead we created a new benchmark set, including solvable as well as unsolvable instances. We designed three categories of benchmarks.

**The “\( \omega \)-\( \Phi \)” benchmarks.** Several well-known benchmarks actually already use avoid conditions, not modeled explicitly but instead encoded into complex precondition and/or effect-condition formulas. We have identified 6 such domains, and manually separated the avoid condition from the action descriptions in an equivalence-preserving manner. The domains and avoid conditions are: CaveDiving (IPC14), mutual exclusion relationship between some divers; Fridge, constraints on fridge components; Miconic, complex relationship between passengers allowed to be in the elevator simultaneously, legal elevator moves are restricted by boarded pas-
The “(k)n” benchmarks. These benchmarks add avoid conditions systematically to some standard benchmarks. The avoid conditions we added enforce an upper limit k on the number of occurrences of n selected events in any plan. Specifically, we considered “road avoidance” in Storage, Transport, and Trucks, forcing each vehicle to not traverse ≥ k of n selected connections in the road-map graph; and “same-achiever avoidance” in Rovers and Satellite, forcing each rover/satellite to not achieve ≥ k of n selected goals.

We generated the benchmark instances as follows. The size of the avoid conditions scales with \(k^n\), so to keep the size under control we fixed k to 2 throughout. The n road-map connections/goals are selected arbitrarily. For each basic instance, we determined the smallest value of n, denoted \(n_{\infty}\), for which no \(\phi\)-compliant plan exists. Where such an \(n_{\infty}\) was found, we added the instance with avoid condition for \(n_{\infty} - 1\) to the solvable benchmark set, and for \(n_{\infty}\) to the unsolvable benchmark set. If \(n_{\infty} = 2\) was already unsolvable, we only added the instance for n = 1 to the unsolvable set.

Note that these avoid conditions are DNF formulas. We acknowledge that this creates a bias in our benchmark set to DNF avoid conditions. It appears natural though for an avoid condition to take the form of a list of bad things that should not happen, which is a DNF if each “bad thing” is characterized conjunctively just like preconditions and the goal.

The “-T” benchmarks. Finally, we designed a small set of benchmarks using trap learning as an avoid-condition generator. We considered unsolvable resource-constrained bench-
marks [Nakhost et al., 2012], where trap learning empirically works best [Steinmetz and Hoffmann, 2017a]. For each benchmark instance, we use trap learning to compute a complete trap, i.e., a DNF $\Psi_\infty$ that proves the instance unsolvable. For generating the avoid condition, we then select the first 20% of the conjunctions added to $\Psi_\infty$.

The advantage of this scheme is that it allows systematic benchmark generation; the disadvantage is that it is somewhat artificial, as the generated avoid conditions presumably are quite different from what a human user would specify. Our results on these benchmarks should thus be interpreted with care, and are included merely as a showcase.

6.2 Results using Compilations
Consider Table 1. For the compilations, the results in the different categories (satisficing, optimal, and unsolvability) are qualitatively similar. Using additional dead-end detectors (trap learning/abstraction) on top of the compilations turned out to be detrimental in all cases, so we omit these results.

Both $\Pi^{\phi}$ and $\Pi^{\phi}_{\text{TTL}}$ cause a significant overhead in grounding for almost all domains. This was to be expected for the $\binom{n}{k}$ and $\text{T}$ part, as grounding in both compilations requires the conversion of the CNF $\neg\phi$ back into DNF, which with the standard FD translator method is exponential in the size of $\phi$. That said, the results are not much better on the $\Phi$ benchmarks either. This is because, after the elimination of existential quantifiers, the avoid conditions then turn into big disjunctions too (reinforcing our view that DNF appears to be a natural form of avoid condition). The results for $\Pi^{\phi}$ are significantly worse than for $\Pi^{\phi}_{\text{TTL}}$ because the former needs to do the DNF conversion for every action, while the automaton construction in $\Pi^{\phi}$ requires this only once.

The axiom compilation $\Pi^\lambda$ is designed to avoid these problems ($\Pi^\lambda$ is missing in the optimal part since axioms are not supported by the optimal planner configuration). Nevertheless, planning performance does not benefit from having the avoid condition encoded directly in the model. $\Pi^\lambda$ is dominated almost universally by the simple $\phi$-pruning baseline (pruning states that satisfy $\phi$).

6.3 Results using Prediction Methods
For the $\phi$-prediction methods, Table 1 also shows search space size reduction statistics. We selected abstraction size limits of $20k$ and $160k$ whose results are representative.

The trade-off between overhead and benefit generally becomes better the more the solution space is constrained. In satisficing planning, coverage could be improved over the base configuration in two domains. While search effort could be reduced in other domains as well, this reduction does not outweigh the incurred overhead. Hence our prediction methods lag behind in terms of total coverage here. In optimal planning, coverage improvements are still limited to the same domains. However, as the overall search becomes more expensive, investing time into the predictor computation becomes (relatively) less of an issue. The impact of the prediction methods becomes clearest in proving unsolvability, where pruning is most important. Here coverage results are in favor of the prediction configurations in all but one domain.

For both satisficing and optimal planning, the impact of $\phi$-prediction in terms of search reduction highly varies between domains. In the $\Phi$ and $\binom{n}{k}$ benchmarks, there are many domains where the additional pruning has (almost) no effect. The reason lies in the structure of these domains. In Openstacks-$\Phi$, avoiding $\phi$ is always possible if it is not satisfied already. In the Nurikabe-$\Phi$ instances, it is almost not possible to make an illegal group assignment. Similarly, in most of the $\binom{n}{k}$ domains, there usually exist enough alternatives to get around the avoid condition. In Rovers-$\Phi$ and Satellite-$\Phi$, all goals can usually be achieved by all the agents so that restrictions on which agent is used for which goal are not important. In the other $\binom{n}{k}$ domains, the road network is often highly connected, always leaving open an alternative route.

The positive examples are CaveDiving-$\Phi$, Miconic-$\Phi$, and Trucks in both variations. In these domains, wrong decisions early on can make $\phi$ unavoidable. For example, in CaveDiving-$\Phi$, different divers must assist each other at different stages. Starting with a wrong diver can make this impossible. In Miconic-$\Phi$, boarding passengers in the wrong order can make it impossible to move the elevator later on without violating some of the constraints.

In the unsolvable benchmarks, reasoning over $\phi$ is most important, and the potential of $\phi$-prediction can be best seen. Comparing $\phi$-trap learning vs. the abstractions, each approach shows good results in some domains. In the $\binom{n}{k}$ domains, $\phi$-trap learning causes the larger overhead, while the abstractions provide better predictions. The superior results of $\phi$-trap learning in the $\text{T}$ benchmarks must be treated with caution due to the benchmark design, as the avoid conditions are generated by a trap-learning process in the first place.

The results for unsolvable benchmarks also clearly show that our modifications of CEGAR have the intended effect. Both PA and SPA were often able to prove the initial state unsolvable directly (no search needed). The $\phi$ implication checks can slow down the abstraction construction though. This is visible in the cases where $\phi$-prediction was not that useful. Between the two CEGAR variants PA and SPA, there is no clear winner. Prioritizing refinements based on $\phi$ works better in some domains, focusing on abstract-transition flaws in others. Overall SPA provides slightly better pruning capabilities, but is also often more expensive in the construction.

7 Conclusion
State constraints are a natural modeling construct in planning, and has so far been considered mostly in temporal form. Here we consider the non-temporal special case of avoid conditions $\phi$ that must be false throughout the plan. We have designed advanced methods predicting states unsolvable due to $\phi$, and our experiments show that they pay off.

While our benchmarks are mostly designed having in mind a human modeller who specifies the avoid condition $\phi$, an interesting avenue for future research is to instead leverage this modeling construct to connect to offline domain analyses. Under-approximations of unsafe or dangerous regions of states naturally form avoid conditions. It may then make sense to consider non-deterministic or probabilistic planning, and to directly handle BDD representations of $\phi$.  