Classical Planning with Avoid Conditions

Marcel Steinmetz[1], **Jörg Hoffmann**[1], **Alisa Kovtunova**[2], **Stefan Borgwardt**[2]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] Institute of Theoretical Computer Science, Technische Universität Dresden, Germany
lastname@cs.uni-saarland.de, firstname.lastname@tu-dresden.de

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 trajectory constraints. Here we focus on the simpler case where the constraint is a non-temporal formula φ – 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 φ, and we show how to tailor abstractions to recognize that avoiding φ 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 φ. The results show that our techniques are often superior.

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 trajectory constraints*. Work since then has devised compilations back into classical tasks (Edelkamp 2006; Baier and McIlraith 2006; De Giacomo, De Masellis, and Montali 2014; Torres and Baier 2015; Bonassi et al. 2021), compilations into SAT (Mattmüller and Rintanen 2007), and approaches handling soft-goal plan preferences effectively (Baier, Bacchus, and McIlraith 2007, 2009).

Here we focus on the special case where the state trajectory constraint is a state formula φ that must remain false in all states along the plan. We refer to such constraints as *avoid conditions*. This special case is relevant, for example, to express dangerous or undesirable situations, such as risky states in a deterministic approximation of a probabilistic planning application. While the avoidance of such situations could in principle be enforced via additional action preconditions, it is typically easier and much more natural to model them directly as conditions to avoid. Moreover, avoid conditions can be used as a tool to specify domain-specific control knowledge, e. g., characterizing states the user knows to be dead ends. The question then becomes how to make use of the provided knowledge in the best possible way.

A straightforward way to handle avoid conditions is the compilation into preconditions. However, 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 φ can no longer be avoided.

Our learning method is based on *traps* (Lipovetzky, Muise, and Geffner 2016). Traps are goal-free sets of states that, once entered, cannot be left again. Represented compactly, traps yield an effective method to recognize dead-end states during forward search. By initiating trap refinements from unrecognized dead ends encountered in search, one can incrementally extend the representation as a form of nogood learning (Steinmetz and Hoffmann 2017). We extend the trap definition to take into account φ. We show how these changes fit into the trap learning approach, and introduce an adaption of *Trapper* (Lipovetzky, Muise, and Geffner 2016) as an alternative trap generation method.

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' that *implies* φ, i.e., where $s' \models \varphi$ 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, 2018). We show how one can test $A' \Rightarrow \varphi$ for Cartesian states. We modify the CEGAR process to incorporate φ as an additional source of counter-examples and, therewith, of refinement steps.

We run experiments on satisficing planning, optimal plan-

ning, and proving unsolvability, evaluating compilations, traps, and abstraction. We do so on (1) reformulated standard benchmarks that incorporate aspects (more) naturally formulated as avoid conditions; and (2) benchmarks involving road maps, where we systematically impose avoid conditions of the form "do not use particular combinations of road segments". The results show that our new methods can be superior, in particular for proving unsolvability.

Technical details, including full proofs and detailed benchmark descriptions, are provided in online appendix (Steinmetz et al. 2022).

Preliminaries

We consider classical planning tasks in FDR notation (Bäckström and Nebel 1995; Helmert 2006). An **FDR planning task** is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$. $\mathcal{V}$ is a set of **variables**, each $v \in \mathcal{V}$ has a finite domain $\mathcal{D}_v$. A **fact** is a variable assignment $p = \langle v, d \rangle$ for $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$. The **initial state** $\mathcal{I}$ is a complete assignment to $\mathcal{V}$. The **goal** $\mathcal{G}$ is a partial assignment to $\mathcal{V}$. For a partial variable assignment P, $\mathcal{V}(P) \subseteq \mathcal{V}$ denotes the set of variables v for which $P(v)$ is defined. For two partial variable assignments P_1 and P_2, we denote by $P_1[P_2]$ the update of P_1 by P_2, i.e., $P_1[P_2](v) = P_2(v)$ for $v \in \mathcal{V}(P_2)$, and $P_1[P_2](v) = P_1(v)$ for $v \in \mathcal{V}(P_1) \setminus \mathcal{V}(P_2)$. $\mathcal{A}$ is the set of **actions**. Each action $a \in \mathcal{A}$ 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** $\mathcal{S}$ of Π are all complete variable assignments. A state $s_\mathcal{G}$ is called **goal state** if $s_\mathcal{G}(v) = \mathcal{G}(v)$ for all $v \in \mathcal{V}(\mathcal{G})$. An action a is **applicable** in a state s if $s(v) = pre_a(v)$ for all $v \in \mathcal{V}(pre_a)$. The application results in the state $s[\![a]\!] = s[\mathit{eff}_a]$. These definitions are extended to sequences of actions in a straightforward manner. A sequence of actions π is a **plan** for s if π is applicable in s and $s[\![\pi]\!]$ is a goal state. 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 Π is a plan for $\mathcal{I}$. Π is called **unsolvable** if $\mathcal{I}$ is a dead end.

We denote Boolean formulae over facts by ϕ. We consistently use ψ to denote conjunctions of facts. Ψ and Φ denote negation-free formulae in disjunctive normal form (DNF). Partial variable assignments, conjunctions of facts, and sets of facts are used interchangeably. Ψ and Φ are treated like sets of conjunctions. We denote by $s \models \phi$ that state s satisfies ϕ. By $[\phi] \subseteq \mathcal{S}$ we denote the set of all states that satisfy ϕ. For action a, $\mathit{Progress}(\phi, a)$ denotes the **progression** of ϕ by a, and states the condition that holds after applying a to any $s \in [\phi]$. The computation of $\mathit{Progress}(\phi, a)$ for general ϕ is shown by Rintanen (2008). For a conjunction ψ, $\mathit{Progress}(\psi, a) = (\psi \cup pre_a)[\mathit{eff}_a]$ if $\psi(v) = pre_a(v)$ for all $v \in \mathcal{V}(\psi) \cap \mathcal{V}(pre_a)$, and $\mathit{Progress}(\psi, a) = \bot$ otherwise.

An **avoid condition** φ is an arbitrary Boolean formula. A plan $a_1, \ldots, a_n$ for Π is called **φ-compliant** if $\mathcal{I} \notin [\varphi]$, and it holds for all $1 \leq i \leq n$ that $\mathcal{I}[\![a_1, \ldots, a_i]\!] \notin [\varphi]$. An **optimal φ-compliant plan** is a φ-compliant plan with minimal action cost. We say that a state s is **φ-unsolvable** if there is no φ-compliant plan for s.

Compilations

Compiling avoid conditions into the planning task is straightforward in principle, but the naïve method is not very effective. 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. They all operate at the PDDL input level.

Conditions Compilation The first compilation ensures φ-compliance by conjoining $\neg\varphi$ to the preconditions of all actions and the goal. We denote by $\Pi^{\neg\varphi}$ the resulting FDR planning task. Trivially, the plans of $\Pi^{\neg\varphi}$ are the φ-compliant plans of Π.

LTL Compilation Our second method uses existing tools for compiling temporal formulas into planning tasks (Edelkamp 2006; Baier and McIlraith 2006). The compilation proceeds in two steps: 1) building an automaton representation of the formula, and 2) encoding this automaton into the planning task via additional variables and actions. The LTL formula in our case is $G\neg\varphi$ (*always* not φ), which translates into an automaton with exactly two locations. The initial location is accepting and has a self-loop conditioned by $\neg\varphi$. The other location is not accepting, and is reached from the initial location if φ is satisfied. We denote by Π^{LTL} the result of encoding this automaton into Π. Π^{LTL} enforces an update of the automaton location in between applications of actions from Π. The automaton "blocks" as soon as it leaves its accepting state. Discarding the automaton-related actions, the plans of Π^{LTL} are exactly the φ-compliant plans of Π. Moreover, plan optimality is not affected provided that the newly introduced actions have 0 cost.

Axiom Compilation Our last compilation exploits *derived predicates*, aka *axioms* (Hoffmann and Edelkamp 2005). Axioms are defined by rules of the form $p \leftarrow \phi_p$. The fact p must not be affected by any action, i.e., its truth value must be exclusively 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 ϕ_p. To enforce $\neg\varphi$ with axioms, we introduce a rule $\mathsf{(avoid)} \leftarrow \varphi$, and conjoin $\neg\mathsf{(avoid)}$ to the precondition of every action and to the goal. We denote by $\Pi^{\mathcal{X}}$ the resulting FDR task with axioms.

Avoid Prediction

In theory, the compilations induce a blow-up that is polynomial in the original task and φ. However, PDDL grounding typically involves formula normalization, which if not done carefully can actually result in an exponential overhead depending on φ. We now turn to our advanced techniques that handle φ without the detour via compilation. The base algorithm is forward search on Π. φ-compliance is assured by pruning all states that satisfy φ. We enhance this basic pruning condition by **φ-predictors**, functions $u : \mathcal{S} \mapsto \{0, \infty\}$ that may identify φ-unsolvable states, $u(s) = \infty$, where φ is not satisfied directly. Provided that $u(s) = \infty$ indeed only holds if s is φ-unsolvable, these states can be pruned in addition without affecting the search's completeness (returning a φ-compliant plan if one exists) and optimality (returning an

optimal φ-compliant plan) properties. In what follows, we devise two such predictors u from well-known techniques.

Avoid-Prediction via Traps

There is an obvious methodological relation between φ-prediction and dead-end detection. Both methods attempt to prove the absence of goal-leading paths. Yet, while the latter requires universal absence, the former makes exceptions according to φ. This section shows how to incorporate this exception into *traps* (Lipovetzky, Muise, and Geffner 2016).

Background: Traps

A **trap** is a set of states $T \subset \mathcal{S}$ that **(T1)** does not contain goal states, and **(T2)** is closed under transitions. The conditions guarantee that all states in T are dead ends. Lipovetzky, Muise, and Geffner (2016) considered DNF formulae Ψ, for which $T = [\Psi]$ satisfies **(T1)** and **(T2)** iff it holds for every $\psi \in \Psi$ that **(t1)** $\psi \wedge \mathcal{G} \Rightarrow \bot$, i.e., ψ and $\mathcal{G}$ disagree on some variable; and **(t2)** for every action a, $\mathit{Progress}(\psi, a) \Rightarrow \Psi$, i.e., $\psi' \subseteq \mathit{Progress}(\psi, a)$ for some $\psi' \in \Psi$ or else $\mathit{Progress}(\psi, a) = \bot$. *Trapper* uses these syntactic conditions to build Ψ from conjunctions of a fixed size k, called k-trap. Algorithm 1 sketches the procedure, which, starting from all size-k conjunctions, iteratively removes conjunctions from Ψ until **(t1)** and **(t2)** are satisfied.

Algorithm 1: k-trap computation. Adaptions to φ in blue.

1 $\Psi \leftarrow \{\psi \mid |\psi| = k\} \cup \Phi$;
2 **foreach** $\psi \in \Psi$ *s.t.* $\psi \wedge \mathcal{G} \not\Rightarrow \bot$ / $\psi \wedge \mathcal{G} \not\Rightarrow \Phi$ **do** $\Psi \leftarrow \Psi \setminus \{\psi\}$;
3 **while** *there are* $\psi \in \Psi$ *and* $a \in \mathcal{A}$ *s.t.* $\mathit{Progress}(\psi, a) \not\Rightarrow \Psi$ *and* $\psi \wedge \mathit{pre}_a \not\Rightarrow \Phi$ **do**
4 $\quad \Psi \leftarrow \Psi \setminus \{\psi\}$

k-traps are limited in practice to small k. Steinmetz and Hoffmann (2017) presented a method to construct Ψ without bounding conjunction size. They interleave search's exploration with Ψ refinements. Search starts with the empty trap, $\Psi := \bot$, and terminates as soon as a goal state is found. Ψ is updated by *learning* from dead ends $\hat{S}$ that were explored in search. Such $\hat{S}$ is identified whenever all non-$\hat{S}$ successors of the states in $\hat{S}$ were pruned due to Ψ. The refinement computes a new trap $\hat{\Psi} := \Psi \vee \bigvee_{s \in \hat{S}} \psi_s$, where $\psi_s \subseteq s$ for every $s \in \hat{S}$. This is possible because, as per the identification of $\hat{S}$, $\hat{\Psi}$ is a trap if $\psi_s = s$. Every state newly represented by $\hat{\Psi}$ besides those in $\hat{S}$ can lead to additional pruning in the remainder of the search. To achieve this *generalization*, Steinmetz and Hoffmann use the greedy procedure sketched in Algorithm 2, which adds facts from s to ψ_s only as necessary to prevent the violation of **(t1)** (lines 1 – 3) and **(t2)** (lines 4 – 5).

Tailoring To Avoid Condition

We call a set of states $T \subseteq \mathcal{S}$ a **φ-trap** if **(T1$^\varphi$)** every goal state in T satisfies φ, and **(T2$^\varphi$)** every transition that leaves

Algorithm 2: Computation of ψ_s for the trap update $\hat{\Psi}$ as discussed in the text. Adaptions to φ in blue.

1 **foreach** $s \in \hat{S}$ **do**
2 $\quad$ select $v \in \mathcal{V}(\mathcal{G})$ s.t. $s(v) \neq \mathcal{G}(v)$;
3 $\quad \psi_s \leftarrow \langle v, s(v) \rangle$;
4 **while** *there are* $s \in \hat{S}$ *and* $a \in \mathcal{A}$ *s.t.* $\mathit{Progress}(\psi_s, a) \not\Rightarrow \hat{\Psi}$ *and* $\psi_s \wedge \mathit{pre}_a \not\Rightarrow \Phi$ **do**
5 $\quad \psi_s \leftarrow \psi_s \wedge \langle v, s(v) \rangle$ for some $v \notin \mathcal{V}(\psi_s)$;

T either originates in a state that satisfies φ, or goes into one that does. Every state that satisfies φ is φ-unsolvable by definition. **(T2$^\varphi$)** additionally ensures that leaving T is possible only by making φ true. In summary, it is not possible to reach the goal from any state in T without satisfying φ:

Theorem 1. *If T is a φ-trap, then every state in T is φ-unsolvable.*

To operationalize on this notion, we extend the construction methods from above. This is possible, in principle, because deciding whether $[\Psi]$ satisfies **(T1$^\varphi$)** and **(T2$^\varphi$)** can still be decomposed into individual conditions on each ψ:

Theorem 2. *Let Ψ be a DNF formula over facts without negation. $[\Psi]$ is a φ-trap if it holds for all $\psi \in \Psi$ that **(t1$^\varphi$)** $(\psi \wedge \mathcal{G}) \Rightarrow \varphi$, and **(t2$^\varphi$)** it holds for all $a \in \mathcal{A}$ that $\mathit{Progress}(\psi \wedge \neg\varphi, a) \Rightarrow (\Psi \vee \varphi)$.*

The proof is straightforward and provided in the appendix. Unfortunately, the appearance of φ significantly increases the complexity of verifying the conditions. Both conditions involve deciding propositional formula tautology, which without assumptions on φ is **coNP**-complete.

Testing **(t1$^\varphi$)** and **(t2$^\varphi$)** could be cast into appropriate calls to an off-the-shelf SAT solver. This is however bound to generate a large overhead, given the overall number of such tests required. Instead, we reformulate **(t1$^\varphi$)** and **(t2$^\varphi$)** into simple trap membership tests. Consider the DNF transformation Φ of φ. Plugging Φ into **(t1$^\varphi$)** gives $(\psi \wedge \mathcal{G}) \Rightarrow \Phi$, which boils down to finding a member $\psi' \in \Phi$ with $\psi' \subseteq (\psi \cup \mathcal{G})$, as above. To reformulate **(t2$^\varphi$)**, we split it into two cases: φ must be true before or it must be true after the application of a. Formally, **(t2$^\varphi$a)** $\mathit{Progress}(\psi, a) \Rightarrow (\Psi \vee \varphi)$ or **(t2$^\varphi$b)** $(\psi \wedge \mathit{pre}_a) \Rightarrow \varphi$. **(t2$^\varphi$a)** is clearly a sufficient condition of **(t2$^\varphi$)**. Moreover, if **(t2$^\varphi$b)** is satisfied, then $\mathit{Progress}(\psi \wedge \neg\varphi, a) = \bot$, i. e., **(t2$^\varphi$)** is also satisfied. By replacing φ by Φ, both conditions can again be verified via trap membership tests. We need to insert two remarks. **(t2$^\varphi$a)** and **(t2$^\varphi$b)** are sufficient but not necessary, as removing $\neg\varphi$ entirely from the progression comes with a loss of information about a's application context that can be necessary to conclude **(t2$^\varphi$)**. Secondly, the DNF transformation may come at the cost of an exponential blow-up in formula size.

Nevertheless, the Φ approach offers another advantage: it straightforwardly integrates into the trap construction methods from before. The blue parts in Algorithm 1 and 2 show the necessary changes. In both methods, we make sure that Φ is included in Ψ at all time. This is

possible because $[\Phi]$ itself trivially constitutes a φ-trap. Since $\mathit{Progress}(\psi, a) \Rightarrow (\Psi \vee \Phi)$ then becomes equivalent to $\mathit{Progress}(\psi, a) \Rightarrow \Psi$, **(t2$^\varphi$a)** is handled out-of-the-box. **(t2$^\varphi$b)** maps into additional loop conditions. **(t1$^\varphi$)** replaces **(t1)** in the k-trap construction procedure. This is not necessary in Algorithm 2 because there, by design, $\hat{S}$ does not contain goal states. By definition, Algorithm 1 guarantees that **(t1$^\varphi$)** and **(t2$^\varphi$)** are satisfied upon termination. Hence, Theorem 2 directly yields the desired result:

Theorem 3. *If Ψ is a k-φ-trap, then $[\Psi]$ is a φ-trap.*

Algorithm 2 also guarantees that **(t1$^\varphi$)** and **(t2$^\varphi$)** hold upon termination. Yet, termination here is guaranteed only for $\hat{S}$ such that $([\Psi] \cup \hat{S})$ remains a φ-trap. That this is always the case can be shown via the exact same arguments as already given by Steinmetz and Hoffmann (2017).

Theorem 4. *Every refinement of the φ-trap learning procedure terminates with a φ-trap $\hat{\Psi}$.*

Abstraction for Avoid-Prediction

We recall Cartesian abstractions and show how to tailor them to the identification of φ-unsolvable states.

Background: Cartesian Abstractions

An **abstraction** for Π is an equivalence relation $\sim$ between the states $\mathcal{S}$. The **abstract states** $\mathcal{S}^\sim$ 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 $\Theta^\sim = \langle \mathcal{S}^\sim, \mathcal{T}^\sim, s^\sim_\mathcal{I}, \mathcal{S}^\sim_\mathcal{G} \rangle$ with **abstract initial state** $s^\sim_\mathcal{I} = [\mathcal{I}]$ and **abstract goal states** $\mathcal{S}^\sim_\mathcal{G} = \{[s] \mid s \in \mathcal{S}, \mathcal{G} \subseteq s\}$. The **abstract transitions** are given by $\mathcal{T}^\sim = \{\langle [s], a, [s[\![a]\!]] \rangle \mid s \in \mathcal{S}, a \in \mathcal{A} \text{ applicable in } s\}$.

Let the variables of Π be $\mathcal{V} = \{v_1, \ldots, v_N\}$. **Cartesian abstractions** (Seipp and Helmert 2018) are abstractions whose abstract states are of the form $A_1 \times A_2 \times \cdots \times A_N$, where $A_i \subseteq \mathcal{D}_{v_i}$ for all i.

This structure makes Cartesian abstractions particularly suitable for a **counter-example guided abstraction 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 $[s_0] = s^\sim_\mathcal{I}$ to some abstract goal state $[s_n] \in \mathcal{S}^\sim_\mathcal{G}$. If no such path exists, then Π must be unsolvable, and the refinement terminates. Otherwise, the corresponding concrete path $\underline{s_0}, a_1, \underline{s_1}, a_2, \ldots$ is computed by applying the actions successively, starting from $\underline{s_0} = \mathcal{I}$. The computation is stopped when one of the following conditions is satisfied:

(C1) Action a_i is not applicable in $\underline{s_{i-1}}$.

(C2) Concrete and abstract state do not match: $[\underline{s_i}] \neq [s_i]$.

(C3) $\underline{s_n}$ does not satisfy the goal.

If not stopped, we have found a plan for Π 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 ($\uplus$ denotes disjoint set union):

(C1) $[s_{i-1}]$ is split into $[t_1] \uplus [t_2]$ such that $\underline{s_{i-1}} \in [t_2]$ and $[t_2]$ has no abstract transition via a_i.

(C2) $[s_{i-1}]$ is split into $[t_1] \uplus [t_2]$ such that $\underline{s_{i-1}} \in [t_2]$ and $[t_2]$ no longer has an abstract transition to $[s_i]$ via a_i.

(C3) $[s_n]$ is split into $[t_1] \uplus [t_2]$ such that $\underline{s_n} \in [t_2]$ and $[t_2]$ 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 is updated by "rewiring" transitions to $[t_1]$ and $[t_2]$. For full details, we refer to the work by Seipp and Helmert (2018). Once the abstract state space has been updated, a new abstract solution is extracted, and the whole process starts anew.

Tailoring to Avoid Conditions

An abstract state $[s]$ **implies** φ, written $[s] \Rightarrow \varphi$, if $s' \models \varphi$ holds for all represented concrete states $s' \in [s]$. Since the abstract state space is path-preserving, its analysis with respect to this property yields information for φ-prediction:

Theorem 5. *Let $[t]$ be any abstract state. If every path from $[t]$ to any abstract goal state visits some $[s]$ s.t. $[s] \Rightarrow \varphi$, then every state represented by $[t]$ is φ-unsolvable.*

Intuitively, the φ-unsolvable abstract states are exactly the abstract dead ends after pruning all $[s] \Rightarrow \varphi$. We next show how to use this observation in Cartesian abstractions.

Implication Test Unfortunately, deciding whether $[s] \Rightarrow \varphi$ for Cartesian abstractions is **coNP**-hard in general. Suppose all variables are Boolean. Then, for the full Cartesian product $[s] = \{\top, \bot\}^N$, $[s] \Rightarrow \varphi$ holds exactly if φ is a tautology, deciding which is known to be **coNP**-complete. Yet despite the worst-case complexity, the implication check was usually not the bottleneck in our experiments. Our implementation runs a simple backtracking search for a state $t \in [s]$ such that $t \models \neg\varphi$, as depicted in Algorithm 3. We assume a positive DNF representation Φ of φ. This allows us to easily identify situations where branching is not required (line 3).

Algorithm 3: *Contains*(i, Φ): Tests whether $\exists t \in [s] = A_1 \times \cdots \times A_N$ s.t. $t \models \neg\Phi$. Initially, $i = 0$.

```
1 if Φ = ∅ then return true ;
2 if i = N + 1 then return false ;
3 if ∃d_i ∈ A_i: ⟨v_i, d_i⟩ ∉ ψ for all ψ ∈ Φ then
4     return Contains(i + 1, {ψ ∈ Φ | v_i ∉ V(ψ)})
5 else
6     foreach d_i ∈ A_i do
7         if Contains(i + 1, {ψ ∈ Φ | v_i ∉ V(ψ),
              or ψ(v_i) = d_i} then return true ;
8     return false;
```

CEGAR We propose two CEGAR variants that incorporate φ. Abstract goal paths are generally restricted such that

$[s_i] \not\Rightarrow \varphi$ holds at all times. If such a path does not exist, then $\mathcal{I}$ must be φ-unsolvable, and we can terminate. Otherwise the refinement proceeds as follows.

Integrated: The first variant introduces an additional error condition into the original analysis procedure:

(C4) The concrete state $\underline{s_i}$ satisfies φ.

Assume (C4) is satisfied. Since $[s_i] \not\Rightarrow \varphi$ by assumption, $[s_i]$ must hence contain states that satisfy φ as well as ones that do not. We split $[s_i]$ such that $[\underline{s_i}] \Rightarrow \varphi$ holds after the refinement. In line with the previous error conditions, this is sufficient to ensure that the same concrete path cannot be subject to any future refinement iteration.

Concretely, $[s_i]$ is split into abstract states $[t_1] \uplus \cdots \uplus [t_k]$ such that $[t_k] \Rightarrow \varphi$, and $\underline{s_i} \in [t_k]$. Contrary to the original error conditions, a split into exactly two abstract states ($k = 2$) is not possible in general. To illustrate this, let x and y be two binary variables, and $\varphi = (x = 1 \wedge y = 1)$, and consider the abstract state $[s_i] = (\{0, 1\} \times \{0, 1\})$. $[t_k] \Rightarrow \varphi$ can only be satisfied for $[t_k] = (\{1\} \times \{1\})$. However, every possible split of $[s_i]$ into this $[t_k]$ requires $k \geq 3$ abstract states.

We use the following procedure to find $[t_1] \uplus \cdots \uplus [t_k]$. We start with $[t] = [s_i]$ and $j = 1$. We continue to split $[t]$ into abstract states $[t_j]$ and $[t']$ with $s_i \in [t']$ until $[t] \Rightarrow \varphi$ is satisfied. $[t']$ replaces $[t]$ in the next round, $j + 1$. Let v be any variable whose value set A in $[t]$ is not a singleton. Such a variable must exist. Otherwise $[t] = \{s_i\}$, therefore $[t] \Rightarrow \varphi$ as per assumption (C4), and we would have terminated the refinement already. Given such v, $[t]$ is split by dividing A into $(A \setminus \{\underline{s_i}(v)\})$ and $\{\underline{s_i}(v)\}$ respectively. This method guarantees that $[t]$ will eventually only contain $\underline{s_i}$. Termination with the desired result hence follows. Since we kept splitting abstract states into pairs, the abstract state space can be updated via the same efficient methods as before.

Detached: To prioritize refinements based on φ, our second CEGAR variant checks, prior to the original analysis steps, whether any abstract state $[s_i]$ along the considered abstract goal path contains a concrete state $s \models \varphi$. If so, then $[s_i]$ is split following the procedure just described. Afterwards, we directly proceed to the next refinement iteration, skipping the original conditions altogether. Since the φ error condition is tested here before constructing the concrete path, the state $s \in [s_i]$ with $s \models \varphi$ must be searched *actively*. This is computationally more expensive than the simple check in (C4). To find s, we follow a backtracking search similar to the one shown in Algorithm 3.

We close this section with the remark that the original conditions (C1) – (C3) still play a central role for identifying φ-unsolvable states. Consider the example in Figure 1. As the (spurious) path $[s_0], a_3, [s_1]$ shows, paths in the abstraction can simply bypass φ even if the concrete paths cannot. Note that this abstract path violates (C1). The corresponding refinement will split $[s_0]$ by dividing the values of x into $\{0\}$ and $\{1\}$. After the refinement, every abstract goal path from $[s_0]$ needs to go through $[s_2]$. Hence, since $[s_2] \Rightarrow \varphi$, via the refinement due to (C1), the abstraction becomes able to prove that no φ-compliant plan exists.

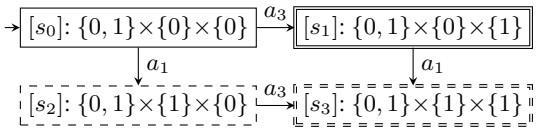

Figure 1: Example abstract state space. The planning task consists of binary variables x, y, z, initially all 0, goal z=1, and three actions with *pre*/*eff*: (a_1) y=0/y=1; (a_2) y=1/x=1; and (a_3) x=1/z=1. The abstract states are depicted in terms of $A_x \times A_y \times A_z$. The avoid condition is $\varphi = (y{=}1)$. Abstract states that imply φ have dashed borders. Goal states have double borders.

Experiments

We implemented all described methods in Fast Downward (FD) (Helmert 2006). The avoid condition is specified as an additional input file in the full PDDL *condition* syntax. The compilations and DNF transformations are implemented as part of FD's translator component. Source code and benchmarks are available online[1]. The experiments were run on machines with Intel Xeon E5-2660 @ 2.20GHz CPUs, and 30 minutes time and 4 GB memory cutoffs.

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 LM-cut (Helmert 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 φ-predictors: "prune-φ" no prediction, only prune by φ; "k-trap" k-φ-traps; "Ŝ-trap" φ-trap learning; "aOri" Cartesian abstraction constructed via the *original* CEGAR approach; "aInt" our *integrated* CEGAR variant; respectively "aDet" the *detached* variant. For the k-φ-traps, we experimented with $k \in \{2, 3, 4, 5\}$. To terminate CEGAR, we enforced an upper limit N on the number of abstract states, $N \in \{25k, 50k, 100k, 150k, 200k, 300k\}$. In addition, we ran a φ-trap learning variant "Ŝ-k-trap" that uses the k-φ-trap with $k = 2$ as kick-start. For the compilations $\Pi^{\neg\varphi}$ and Π^{LTL}, we also considered traps and Cartesian abstractions for pruning dead-ends (not for $\Pi^{\mathcal{X}}$ as neither of them supports axioms).

Benchmark Design

Benchmarks with avoid conditions already appeared in IPC 2006 (Dimopoulos et al. 2006), encoded via state trajectory constraints. But hard constraints appeared only in benchmarks of the temporal track, which makes them unsuited for our experiments. Instead we created a new benchmark set, including solvable as well as unsolvable instances. By synthetically increasing the impact of φ, the unsolvable part pinpoints and evaluates the capabilities of the different pre-

[1] https://fai.cs.uni-saarland.de/steinmetz/planning_with_avoid_conditions.zip

		Coverage														Search Reduction Factors (left: geometric mean, right: max)									
		Compil.				φ-Prediction										φ-Prediction vs. prune-φ									
		$\neg\varphi$	LTL	$\mathcal{X}$	prune-φ	k-trap		Ŝ-trap		aOri		aInt		aDet		k-trap		Ŝ-k-trap		aOri		aInt		aDet	
Domain	#					2	3		$k2$	25k	100k	25k	100k	25k	100k	k=3		k=2		100k		100k		100k	
Satisficing																									
CaveDiving-REDONE	20	0	**7**	**7**	**7**	**7**	**7**	6	6	**7**	**7**	**7**	**7**	0	0	1.0	1.0	1.4	1.5	1.6	3.1	1.0	1.1		
Fridge-REDONE	24	1	6	20	**21**	**21**	11	**21**	**21**	**21**	**21**	**21**	**21**	13	11	1	1	1	1	1	1	1	1	1	1
Miconic-REDONE	178	0	25	20	117	117	117	**120**	**120**	117	117	111	111	72	67	1.6	415	2.5	415	1	1	1.3	145	1.3	145
Nurikabe-REDONE	20	0	2	**12**	11	11	8	11	11	9	9	7	6	4	0	1.4	3.7	2.1	86.1	1	1	1.0	1.1		
Openstacks-REDONE	60	0	4	12	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	8	7	1	1	1	1	1	1	1	1	1	1
Trucks-REDONE	30	0	7	8	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	15	15	1	1	1.2	1.7	1.1	2.2	1.0	1.2	1.0	1.1
Driverlog-ROAD	21	8	4	9	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	6	5	1.4	23.6	1.0	1.3	1	1	1.0	1.0	1	1
Rovers-ROAD	64	4	2	7	12	14	**17**	12	14	12	12	12	12	12	12	4.4	310.3	3.8	310.3	1	1	1.4	75.0	1	1
TPP-ROAD	40	4	4	8	9	9	7	**10**	**10**	9	8	**10**	9	9	8	3.0	258.2	2.4	258.2	1	1	1.3	9.9	1.9	258.2
Transport-ROAD	116	26	32	49	52	58	**64**	47	55	50	50	52	52	23	14	5.5	76.0K	3.5	76.0K	1	1	1.3	6.2	1.4	412.5
$\sum$	573	43	93	152	271	**279**	273	269	**279**	267	266	262	260	162	139	2.0	76.0K	2.2	76.0K	1.0	3.1	1.2	145	1.2	412.5
Optimal																									
CaveDiving-REDONE	20	0	**7**		**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	0	0	1.0	1.0	1.0	1.0	1.3	2.0	1.0	1.0		
Fridge-REDONE	24	1	6		**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	2	2	1.0	1.0	1	1	1	1	1.0	1.0	1	1
Miconic-REDONE	178	0	25		68	68	68	**70**	**70**	62	61	62	62	28	23	1.7	415	1.9	415	1	1	1.3	145	1.4	145
Nurikabe-REDONE	20	0	2		**10**	**10**	8	**10**	**10**	9	9	7	6	3	0	1.4	1.9	1.4	2.1	1	1	1.0	1.3		
Openstacks-REDONE	60	0	4		**25**	**25**	**25**	**25**	**25**	**25**	24	**25**	**25**	20	19	1	1	1	1	1	1	1	1	1	1
Trucks-REDONE	30	0	10		**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	8	8	1	1	1.1	1.2	1.0	1.1	1.0	1.1	1.0	1.0
Driverlog-ROAD	21	7	12		**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	8	7	1.7	9.0	1.1	1.5	1	1	1.0	1.0	1.1	1.5
Rovers-ROAD	64	**3**	**3**		**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	1.5	2.5	1.5	2.5	1	1	1.2	1.4	1	1
TPP-ROAD	40	**0**	**0**		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**										
Transport-ROAD	116	23	41		45	**49**	**49**	45	**49**	45	45	46	46	16	7	2.7	1.4K	2.0	1.4K	1	1	1.2	3.2	1.2	19.1
$\sum$	573	34	110		192	196	194	194	**198**	185	183	184	183	88	69	1.6	1.4K	1.5	1.4K	1.0	2.0	1.2	145	1.2	145
Unsolvability																									
CaveDiving-REDONE	20	0	4	7	7	**20**	16	7	**20**	7	7	7	7	16	16	17.4K	29.5K	17.4K	29.5K	1.1	1.2	17.4K	29.5K	17.4K	29.5K
Miconic-REDONE	28	0	0	0	18	**19**	18	16	16	18	18	16	16	16	16	1.6	1.7	10.0	15.4	1	1	15.0	46.6K	17.3	46.6K
Driverlog-ROAD	22	8	4	8	8	9	**12**	5	8	8	8	8	8	8	8	0.45M	36.4M	0.45M	36.4M	1	1	52.2K	36.4M	0.45M	36.4M
Rovers-ROAD	64	0	0	0	0	16	**22**	14	21	0	0	2	2	1	1										
TPP-ROAD	40	4	0	4	4	8	8	6	**10**	4	4	5	5	4	4	315.7	0.45M	3.2K	0.45M	1	1	0.10M	0.45M	0.10M	0.45M
Transport-ROAD	116	25	32	46	46	53	61	59	**64**	46	45	48	46	55	56	163.1	28.7M	65.7	28.7M	1	1	35.0	46.8K	913.4	28.7M
$\sum$	290	37	40	65	83	125	137	107	**139**	83	82	86	84	100	101	196.4	36.4M	215.9	36.4M	1.0	1.2	154.4	36.4M	1.3K	36.4M

Table 1: Left half: coverage results, best in **bold**. Results for the compilations are shown for the base configurations only. The configuration names are described in the text. Right half: ratio of states visited by prune-φ versus states visited with a φ-predictor on top (K for thousand, M for million). Larger values indicate more pruning. For each method per-domain geometric mean and maximum values are shown. Values across different configurations are not directly comparable.

diction methods to generalize from φ. We design two categories of benchmarks.

REDONE. Several well-known benchmarks feature 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 actions descriptions in an equivalence-preserving manner. In summary, we use CaveDiving (IPC14), hiring a diver may preclude hiring other divers; Fridge, constraints on fridge components; Miconic, elevator moves are restricted by constraints on boarded passengers of numerous kinds; Nurikabe (IPC18), illegal groupings of board cells; Openstacks (IPC08), production and delivery must follow a particular order; Trucks (IPC06), relationship between the occupancy of and legal accesses to truck storage areas. An explicit avoid condition is a natural model for all of these, and partly actually more natural than the original PDDL. Additionally to the existing (solvable) instances, we created unsolvable instances for CaveDiving by introducing a cycle in the divers' preclude relationships[2]; and solvable and unsolvable Miconic instances in which passengers must be served one at a time to not eventually violate φ. In the remaining domains, creating instances unsolvable due to φ is either difficult (Nurikabe), or not possible at all (Fridge, Openstacks, Trucks).

ROAD. Our second category encompasses a set of *controlled* benchmarks, in which the avoid condition is generated in a systematic fashion. We adapted standard transportation-like benchmarks without dead ends: Driverlog, Rovers, TPP, and Transport. We add avoid conditions that forbid the usage of certain combinations of con-

[2]The CaveDiving instances of the UIPC instead make restrictions to the available resources. Diver preclude relationships are not used at all.

nections in the road-map graphs. The avoid conditions are computed individually per base instance. The computation is parameterized in the size n of combinations to be added to φ. Given a base instance, φ is built by: 1) computing a φ-compliant plan (for the current φ), and 2) extending φ by a random selection of n connections from this plan. This is repeated until no φ-compliant plan is left. The result is added to the unsolvable part. To obtain a solvable instance, we drop the last entry of φ, skipping an instance if this would make the condition empty. The φ-compliant plans were computed by domain-specific solvers. We considered $n = 2$ and $n = 3$, and aborted the generation of the conditions for an instance after 10 minutes. Note that the avoid conditions here are DNF formulas.

Results for 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 on top of the compilations turned out to be detrimental in all cases, so we omit these results.

Both $\Pi^{\neg\varphi}$ and Π^{LTL} cause a significant overhead in grounding for almost all domains. This was to be expected for the ROAD part, as grounding in both compilations requires the conversion of the CNF $\neg\varphi$ back into DNF, which with the standard FD translator method is exponential in the size of φ. That said, the results are not much better on the REDONE benchmarks either. This is because, after the elimination of existential quantifiers, the avoid conditions there turn into big disjunctions too. The results for $\Pi^{\neg\varphi}$ are significantly worse than for Π^{LTL} because the former needs to do the DNF conversion for every action, while the automaton construction in Π^{LTL} requires this only once.

The axiom compilation $\Pi^{\mathcal{X}}$ was designed to avoid these problems ($\Pi^{\mathcal{X}}$ is missing in the optimal part since axioms are not supported by the optimal planner configuration). Grounding was indeed much less of an issue, with the exception of Miconic, whose complex avoid condition caused problems to FD's axiom normalization. Nevertheless, planning performance does not benefit from having the avoid condition encoded directly in the model. $\Pi^{\mathcal{X}}$ is dominated almost universally by prune-φ.

Results for φ-Prediction

For the φ-prediction methods, Table 1 also shows search space size reduction statistics. For space reasons, we included a subset of the configurations only. Specifically, we excluded k-trap with $k \geq 4$, whose construction overhead caused a significant drop in performance in all domains but Transport. Differences between the abstraction configurations for $N \geq 150$k diminishes, as the refinements for 150k already exceeded the resource limits in many cases.

The results in the optimal and satisficing parts are similar. However, differences between the φ-predictors tend to be larger in the satisficing part, where the base planner configuration is able to solve more instances. In general, the impact of φ-prediction varies between the different domains. It turns out that Fridge, Openstacks, and Trucks actually do not contain φ-unsolvable states besides ones that already satisfy φ. In these domains, φ-prediction becomes pointless, while still adding an overhead. Search could be reduced in Trucks due to dead ends, which can be identified by the φ-predictors as a byproduct. Overall, the performance of the φ-prediction configurations was worse than the baseline prune-φ only if too many resources were dedicated to the φ-predictor construction. In particular, the smallest k-trap configuration performed as well, or better than prune-φ in all domains. Of the remaining three REDONE domains, φ-prediction could increase coverage in just Miconic, yet search reduction can be observed in all three. By design, reasoning over φ is central in the ROAD benchmarks and in the unsolvability part. Here, φ-prediction turned out advantageous throughout, and improvements over the prune-φ configuration were largest.

aDet performs significantly worse than the other two abstraction variants in the solvable part, but has an edge for proving unsolvability. aDet is more prone to generating unreachable abstract states, which resulted in longer abstraction construction times. Moreover, in the solvable part, aOri and aInt sometimes found concrete solutions, and therefore could terminate the refinement before the state limit was reached (the former more so than the latter, the former not checking φ-compliance for the refinements). This did not happen in aDet at all. Vice versa, aDet was slightly better in proving the initial state φ-unsolvable, causing early termination there. Taking into account φ for the refinement has proved necessary for φ-prediction. aOri could identify additional φ-unsolvable states in (almost) no domain, while aInt achieved notable search reduction in all domains, with the aforementioned exceptions. The φ implication checks can slow down the abstraction construction though. This was a particular issue in Miconic and Nurikabe.

In the solvable part, the impact of $\hat{\text{S}}$-trap was limited, only few sets $\hat{S}$ could be identified to start trap learning. k-trap was able to identify additional φ-unsolvable states almost throughout. Both construction methods showed different strengths in different domains. This has been exploited effectively by $\hat{\text{S}}$-k-trap, even surpassing the performance of the individual methods in some domains. Compared to the abstractions, the trap configurations offered a better trade-off between φ-prediction and overhead.

Conclusion

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

While our benchmarks are mostly designed having in mind a human modeler who specifies the avoid condition, 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 φ.

Acknowledgments

This work was funded by DFG Grant 389792660 as part of TRR 248 (CPEC, https://perspicuous-computing.science).

References

Bacchus, F.; and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2): 123–191.

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS^+ Planning. *Computational Intelligence*, 11(4): 625–655.

Baier, J. A.; Bacchus, F.; and McIlraith, S. A. 2007. A Heuristic Search Approach to Planning with Temporally Extended Preferences. In Veloso, M. M., ed., *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 1808–1815. Hyderabad, India: Morgan Kaufmann.

Baier, J. A.; Bacchus, F.; and McIlraith, S. A. 2009. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 173(5-6): 593–618.

Baier, J. A.; and McIlraith, S. A. 2006. Planning with first-order temporally extended goals using heuristic search. In Gil, Y.; and Mooney, R. J., eds., *Proceedings of the 21st National Conference of the American Association for Artificial Intelligence (AAAI'06)*, 788–795. Boston, Massachusetts, USA: AAAI Press.

Bonassi, L.; Gerevini, A. E.; Percassi, F.; and Scala, E. 2021. On Planning with Qualitative State-Trajectory Constraints in PDDL3 by Compiling them Away. In Biundo, S.; Do, M.; Goldman, R.; Katz, M.; Yang, Q.; and Zhuo, H. H., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, 46–50. AAAI Press.

De Giacomo, G.; De Masellis, R.; and Montali, M. 2014. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In Brodley, C. E.; and Stone, P., eds., *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, 1027–1033. Austin, Texas, USA: AAAI Press.

Dimopoulos, Y.; Gerevini, A.; Haslum, P.; and Saetti, A. 2006. The benchmark domains of the deterministic part of IPC-5. In *IPC 2006 planner abstracts*, 14–19.

Doherty, P.; and Kvarnström, J. 2001. TALplanner: A Temporal Logic Based Planner. *The AI Magazine*, 22(3): 95–102.

Edelkamp, S. 2001. Planning with Pattern Databases. In Cesta, A.; and Borrajo, D., eds., *Proceedings of the 6th European Conference on Planning (ECP'01)*, 13–24. Springer-Verlag.

Edelkamp, S. 2006. On the Compilation of Plan Constraints and Preferences. In Long, D.; and Smith, S., eds., *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS'06)*, 374–377. Ambleside, UK: AAAI Press.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6): 619–668.

Haslum, P.; and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In Chien, S.; Kambhampati, R.; and Knoblock, C., eds., *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS'00)*, 140–149. Breckenridge, CO: AAAI Press, Menlo Park.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the Association for Computing Machinery*, 61(3): 16:1–16:63.

Hoffmann, J.; and Edelkamp, S. 2005. The Deterministic Part of IPC-4: An Overview. *Journal of Artificial Intelligence Research*, 24: 519–579.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Lipovetzky, N.; Muise, C. J.; and Geffner, H. 2016. Traps, Invariants, and Dead-Ends. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS'16)*, 211–215. AAAI Press.

Mattmüller, R.; and Rintanen, J. 2007. Planning for Temporally Extended Goals as Propositional Satisfiability. In Veloso, M. M., ed., *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 1966.

Rintanen, J. 2008. Regression for Classical and Nondeterministic Planning. In Ghallab, M., ed., *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, 568–572. Patras, Greece: Wiley.

Seipp, J.; and Helmert, M. 2013. Counterexample-guided Cartesian Abstraction Refinement. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, 347–351. Rome, Italy: AAAI Press.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62: 535–577.

Steinmetz, M.; and Hoffmann, J. 2017. Search and Learn: On Dead-End Detectors, the Traps they Set, and Trap Learning. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4398–4404. AAAI Press/IJCAI.

Steinmetz, M.; Hoffmann, J.; Kovtunova, A.; and Borgwardt, S. 2022. Classical Planning with Avoid Conditions: Technical Report. https://fai.cs.uni-saarland.de/steinmetz/aaai22_tr.pdf.

Torres, J.; and Baier, J. A. 2015. Polynomial-time reformulations of LTL temporally extended goals into final-state goals. In Yang, Q., ed., *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, 1696–1703. AAAI Press/IJCAI.

Classical Planning with Avoid Conditions: Appendix

#3477

Proofs: φ-Traps

Some of our claims use *Progress*(ϕ, a) for general fact formulae ϕ. In our proofs, we use the following *semantic* definition:

$$[\textit{Progress}(\phi, a)] := \{s[\![a]\!] \mid s \in \mathcal{S}, \textit{pre}_a \subseteq s, s \models \phi\} \quad (1)$$

We write $s \models \textit{Progress}(\phi, a)$ if $s \in [\textit{Progress}(\phi, a)]$. Progression can be defined *syntactically* along the lines of Rintanen's (2008) *regression* definition.

Proof of Theorem 2

Let Ψ be some DNF formula of facts without negation. We need to show that the set of states $[\Psi]$ represented by Ψ is a φ-trap if every element $\psi \in \Psi$ satisfies

(t1′) $\forall s \in \mathcal{S}$, if $s \models \psi \wedge \mathcal{G}$ then $s \models \varphi$; and

(t2′) $\forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}$, if $s \models \textit{Progress}(\psi \wedge \neg\varphi, a)$ then $s \models (\Psi \vee \varphi)$.

Assume that both conditions are satisfied for all elements of Ψ. Let $s \in [\Psi]$ be any state that satisfies Ψ, and let $\psi \in \Psi$ be some conjunction for which $s \models \psi$. If s is a goal state, then $s \models \psi \wedge \mathcal{G}$, i.e., $s \models \varphi$ as per (t1′). Therefore, $[\Psi]$ must satisfy (T1′). If $s \models \varphi$, s cannot possibly violate (T2′). Assume that $s \not\models \varphi$, i.e., $s \models \neg\varphi$. Let a be any action that is applicable in s. Plugging $\phi = \psi \wedge \neg\varphi$ into (1) gives $s[\![a]\!] \models \textit{Progress}(\psi \wedge \neg\varphi, a)$. As per assumption (t2′), hence $s[\![a]\!] \in [\Psi]$ or $s[\![a]\!] \models \varphi$. In either case, (T2′) is satisfied.

Reformulation of (t2′) into (t2′a) and (t2′b)

Theorem 6. *Let Ψ be some DNF formula of facts without negation. Let $\psi \in \Psi$ be arbitrary. It holds that ψ satisfies (t2′) if, for every action $a \in \mathcal{A}$, one of*

(t2′a) *$\forall s \in \mathcal{S}$, if $s \models (\psi \wedge \textit{pre}_a)$ then $s \models \varphi$, or*

(t2′b) *$\forall s \in \mathcal{S}$, if $s \models \textit{Progress}(\psi, a)$ then $s \models (\Psi \vee \varphi)$*

is satisfied.

Proof. Let $\psi \in \Psi$ and $a \in \mathcal{A}$ be arbitrary. If (t2′a) is satisfied for ψ and a, then there is no state s that satisfies ψ and $\textit{pre}_a$, yet not φ. The set $[\textit{Progress}(\psi \wedge \neg\varphi, a)]$ becomes empty, the premise of (t2′) for a is always false. Or in other words, (t2′) is satisfied for ψ and a. For the other condition, (t2′b), note that $[\textit{Progress}(\psi \wedge \neg\varphi, a)] \subseteq [\textit{Progress}(\psi, a)]$. If for every $s \in [\textit{Progress}(\psi, a)]$ it holds that $s \models (\Psi \vee \varphi)$, then so does every $s \in [\textit{Progress}(\psi \wedge \neg\varphi, a)]$. Thus, if (t2′b) is satisfied for ψ and a, then (t2′) is satisfied for ψ an a. We conclude that, if ψ satisfies (t2′a) or (t2′b) for every $a \in \mathcal{A}$, then ψ also satisfies (t2′).

□

Theorem 6 shows that (t2′a) and (t2′b) are sufficient to guarantee (t2′). Replacing (t2′) in any φ-trap generation method preserves the correctness of the construction.

Algorithm 3: k-φ-trap computation

1: $\Psi \leftarrow \Phi \vee \bigvee_{\psi, |\psi| = k} \psi$
2: **while** there is $\psi \in \Psi$ s.t. $\psi \not\Rightarrow \Phi$ and $\psi \Rightarrow \mathcal{G}$ **do**
3: $\quad \Psi \leftarrow \Psi \setminus \{\psi\}$
4: **while** there is $\psi \in \Psi$ and $a \in \mathcal{A}$ s.t. $\psi \wedge \textit{pre}_a \not\Rightarrow \Phi$ and $\textit{Progress}(\psi, a) \not\Rightarrow \Psi$ **do**
5: $\quad \Psi \leftarrow \Psi \setminus \{\psi\}$
6: **return** Ψ

Proof of Theorem 3

The computation of k-φ-traps is sketched in Algorithm 3.

Note that Φ remains part of Ψ throughout. The loop condition in line 2 makes sure that line 3 does not remove any $\psi \in \Phi$. The second loop cannot remove any $\psi \in \Phi$ either since $\psi \wedge \textit{pre}_a \Rightarrow \Phi$ for such ψ and all $a \in \mathcal{A}$. Therefore, $\Psi \Rightarrow \Phi$ is satisfied throughout the construction. The second condition of the loop in line 4 then boils down to (t2′b). The method is guaranteed to terminate since every loop iteration removes an element from Ψ, of which there are only finitely many. For the resulting Ψ, lines 2–3 have made sure that (t1′) is satisfied for all $\psi \in \Psi$. Lines 4–5 guarantee that (t2′a) or (t2′b) is satisfied for all $\psi \in \Psi$ and $a \in \mathcal{A}$. With Theorem 6, all $\psi \in \Psi$ hence satisfy (t2′). Theorem 2 concludes the proof.

Proof of Theorem 4

The adapted trap refinement procedure is depicted in Algorithm 4. We assume that Ψ is initialized to Φ. As refinements do not remove elements from Ψ, Φ remains a part of Ψ at all

Algorithm 4: Computation of ψ_s for the φ-trap update $\Psi' := \Psi \vee \bigvee_{s\in\hat{S}} \psi_s$. 2 – 3 ensure that Ψ' satisfies (t1′); 4 – 5 take care of (t2′a) and (t2′b).

1: $\psi_s \leftarrow \top$ for all $s \in \hat{S}$
2: **for all** ψ_s s.t. $(\psi_s \wedge \mathcal{G}) \not\Rightarrow \Phi$ **do**
3: $\quad \psi_s \leftarrow \psi_s \wedge \langle v, s(v)\rangle$ for some $v \in \mathcal{V}(\mathcal{G})$ s.t. $s(v) \neq \mathcal{G}(v)$
4: **while** there are ψ_s and a s.t. $(\psi_s \wedge \mathit{pre}_a) \not\Rightarrow \Phi$ and $\mathit{Progress}(\psi_s, a) \not\Rightarrow \Psi'$ **do**
5: $\quad \psi_s \leftarrow \psi_s \wedge \langle v, s(v)\rangle$ for some $v \notin \mathcal{V}(\psi_s)$

time. In particular, the second part of condition of the loop in line 4 becomes (t2′b).

We first show that the φ-trap refinement procedures terminates with a φ-trap Ψ' if the set of states $\hat{S}$ used for the refinement is such that

(i) $\Psi \vee \bigvee_{s\in\hat{S}} s$ is a φ-trap, and

(ii) $\hat{S}$ does not contain a goal state.

Condition (ii) is necessary to guarantee that such variable as in line 3 exists. By adding the corresponding fact to ψ_s, $\psi_s \wedge \mathcal{G}$ can no longer be satisfied. Hence, each ψ_s can be processed by the loop only once, the loop must terminate. After the loop terminated, the termination condition guarantees that every ψ_s at this point satisfies (t1′). The second loop only adds facts to the conjunctions. These modifications cannot make (t1′) become false again. Hence, all ψ_s satisfy (t1′) after termination.

Note that $(\Psi \vee \bigvee_{s\in\hat{S}} s)$ implies Ψ' during the entire refinement call, because each $s \in \hat{S}$ implies ψ_s at all time. Hence, if some ψ_s and action $a \in \mathcal{A}$ exist for which the condition in line 4 is satisfied, it must hold that $\psi_s \subset s$ as per condition (i). A variable as required in line 5 exists. The same argument shows that the loop must terminate eventually, namely when $\psi_s = s$ for all $s \in \hat{S}$ at the latest. The loop termination condition ensures that all ψ_s satisfy (t2′a) and (t2′b) for all $a \in \mathcal{A}$. Theorem 6 says that then all ψ_s satisfy (t2′).

The previous elements in $\psi \in \Psi$ are not affected. As per assumption (i), they (still) must satisfy (t1′). Exchanging s by ψ_s in (i) cannot make (t2′) become false either, as Φ remains part of Ψ', and membership in $(\Psi \vee \bigvee_{s\in\hat{S}})$ implies membership in Ψ'. In conclusion, every element in Ψ' satisfies (t1′) and (t2′). By Theorem 2, Ψ' constitutes a φ-trap.

We finally need to show that every $\hat{S}$ identified during search satisfies (i) and (ii). Steinmetz and Hoffmann's (2017) identification of $\hat{S}$ is left unchanged, i.e., search calls a trap refinement once states $\hat{S}$ have been explored, whose successors leaving $\hat{S}$ were pruned due to the current Ψ. Since search terminates upon finding a (non-pruned) goal state, all goal states in $\hat{S}$ must satisfy Ψ. Given that Ψ is a φ-trap — which holds initially, and is guaranteed by the refinements, as shown above — all goal states in $\hat{S}$ must hence satisfy φ. With the constraints on the successor, (i) is hence guaranteed by the identification method. $\hat{S}$ must satisfy (ii) because every goal state in $\hat{S}$ must be identified by Ψ. However, states that satisfied Ψ are not explored by search, so cannot appear in $\hat{S}$.

The REDONE Benchmarks

CaveDiving The goal requires to hire every initially available diver eventually. There is an (acyclic) preclude relationship between the divers, according to which hiring certain divers will no longer be possible depending on which divers have been hired before. Our avoid condition reformulation models this relation explicitly by ensuring that the diver that has been hired last is not precluded by any diver that has already been hired. To generate the unsolvable instances, we extended the standard IPC instances by adding a cycle to the preclude relationships.

Fridge Fridges must be repaired by swapping compressors. The action conditions ensure that (1) compressors cannot be removed if not all screws have been unfastened; (2) a new compressor cannot be attached to a fridge if the old one has not been removed; (3) compressors cannot be attached if some of their screws are fastened; and (4) a fridge can only be started if all screws of the new compressor are fastened. All four conditions translate directly into φ, satisfied if (1/3) a compressor is moved while some screw is still fastened; (2) a fridge contains multiple compressors; and (4) a fridge is running whose compressor has some unfastened screws.

Miconic An elevator benchmark with complex constraints on the passengers. The elevator can move up, down, and stop at the current floor. Stopping at a floor departs all passengers in the elevator whose destination is that floor, and boards all passengers which are waiting at that floor. Our reformulation allows to board and depart each passenger individually. The avoid condition ensures that the elevator does not move before having boarded/departed all passenger as just described. Additionally, the avoid condition takes the role of enforcing the passenger constraints, removing action preconditions accordingly: (1) the elevator may only move up (down) if no passenger is boarded that wants to go down (up); (2) passengers of conflict class A must not be boarded at the same time as passengers of the conflict class B; (3) some passengers must never be alone in the elevator; (4) the elevator must not stop at a floor if a passenger is boarded which should not have access to this floor; (5) some passengers have a "non-stop" requirement that once boarded disallow the elevator to stop at any floor other than the passenger's destination; (6) VIP passengers must be served before all others. To encode these constraints into the avoid condition, we added additional facts representing whether the elevator has stopped, moved up, or moved down.

We generated additional Miconic instances of the following kind. We split n passengers and m floors into half. One half of the passengers is assigned to conflict class A, and the other to conflict class B. The class A passengers are placed in the bottom half of the floors, the B passengers at the upper part. The goal is to swap their places. To not satisfy φ eventually, each stop of the elevator must flip the class of

the passengers in the elevator (departing A and board B, or vice versa depart B and board A). An instance is made unsolvable by introducing an additional A passenger, whose initial position and goal make this flip impossible. We generated 28 solvable and 28 unsolvable instances of this form, scaling passengers n from 12 to 36 and floors m from 16 to 28 with increments of 4.

Nurikabe Nurikabe distributes numbers on a 2D grid. The goal is to find for every numbered cell, a contiguous region surrounding this cell that contains exactly so many cells as given by the number. Different regions must be disconnected. The latter constraint can be trivially encoded as an avoid condition: adjacent cells must not be assigned to different regions.

Openstacks One needs to create products to serve orders. A product must not be made until all orders are started that include this product. Vice versa, an order must not be shipped before all its products were made. Both constraints are naturally expressed as an avoid condition.

Trucks Each truck has a limited storage area, with positions ordered from front to back. A position can only be accessed if no position closer to the front is occupied. We add facts for each truck that represent which of the truck's storage area positions has been accessed last. The constraint is replaced by the avoid condition that checks whether some position in front of the one accessed last is occupied.

References

Rintanen, J. 2008. Regression for Classical and Nondeterministic Planning. In Ghallab, M., ed., *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, 568–572. Patras, Greece: Wiley.

Steinmetz, M.; and Hoffmann, J. 2017. Search and Learn: On Dead-End Detectors, the Traps they Set, and Trap Learning. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4398–4404. AAAI Press/IJCAI.