Planning with Uncertainty: Symmetries, Policy Inference, and Solution Compression

Frederico Messa[a], André Grahl Pereira[a]

[a]*Institute of Informatics, Federal University of Rio Grande do Sul, Brazil*

Abstract

Fully-observable non-deterministic (FOND) planning is at the core of artificial intelligence planning with uncertainty. It models uncertainty through actions with non-deterministic effects. In this work, we present a collection of techniques that establish explicit best-first policy-space search as a method competitive with the state of the art for solving FOND planning tasks. We study how to define equivalence relations between policies, allowing part of the search space to be pruned. We show it is possible to use group theory techniques to effectively compute canonical symmetries between states. We also present two contributions that go beyond just policy-space search: we present a procedure that infers in polynomial time a solution policy function given just the specification of its domain set, and an integer-programming formulation procedure that, given a solution policy defined over complete states, yields a set of resource-efficient models that are capable of finding a partial-state policy that represents it unambiguously with the fewest partial states possible.

Keywords: FOND Planning, Heuristic Search, Pruning, Symmetries, Compression

 Accepted for publication in *Artificial Intelligence* (doi: `https://doi.org/10.1016/j.artint.2026.104574`).

Email addresses: `frederico.messa@inf.ufrgs.br` (Frederico Messa), `agpereira@inf.ufrgs.br` (André Grahl Pereira)

1. Introduction

The ability to plan is one of the defining characteristics of intelligence. It allows the judicious selection of actions to achieve a goal, especially in unfamiliar or complex situations, where blindly performing actions may lead to undesired outcomes, such as unrecoverable failures or unnecessarily high costs. Planning is particularly important in environments with uncertainty, where the agent either does not have full knowledge of the world or does not have full control over the future, and must plan in advance for all possible contingencies that might arise. In this work, we focus on fully-observable non-deterministic planning, a planning paradigm that models the uncertainty through actions with non-deterministic effects.

Fully-observable non-deterministic (FOND) planning is at the core of artificial intelligence planning with uncertainty. It assumes that the agent is capable of correctly enumerating all possible contingencies that might occur during the execution of his actions and has a plan for all of them. A successful plan for a FOND planning task is a policy describing, in advance, what actions the agent should take in each state he might reach so that he eventually reaches a goal state from the initial state. Although modeling uncertainty, concrete probabilities are not a concern in the context of FOND planning, and all possible outcomes of an action must be considered equally to ensure the plan provides flawless guarantees.

FOND planning has been used in a wide range of applications, including *qualitative numerical planning* (Bonet and Geffner, 2020), *hyperproperty verification* (Beutner and Finkbeiner, 2024), *multi-agent planning* (Muise et al., 2016), *contingent planning* (Albore et al., 2009), *resilient planning* (Aineto et al., 2023), *generalized planning* and *LTL synthesis* (Bonet et al., 2017; Camacho et al., 2018), and *dialogue-agent systems design* (Muise et al., 2019).

There are several works in the literature that present techniques for solving FOND planning tasks, from approaches that use classical planning techniques to first generate optimistic policies and iteratively refine them into flawless solutions (Kuter et al., 2008; Fu et al., 2011; Muise et al., 2012; Pereira et al., 2022; Muise et al., 2024) and approaches that incrementally solve the task for bigger and bigger portions of the state space until solving the original problem (Hansen and Zilberstein, 2001; Mattmüller et al., 2010), to approaches that compile FOND into some more general paradigm (Cimatti et al., 2003; Kissmann and Edelkamp, 2009; Ramirez and Sardina, 2014; Geffner and Geffner, 2018; Rodriguez et al., 2022; Yadav and Sardina,

2023). In this work, we focus on another direction for solving FOND planning tasks: policy-space explicit search (Messa and Pereira, 2023).

A* with Non-Determinism (AND* for short) (Messa and Pereira, 2023) "generalizes" A* (Hart et al., 1968) for FOND planning. It performs a best-first search in a constructive space of policies to find a solution policy for the FOND planning task. Starting with only the empty policy, which maps no state to no action, AND* uses a priority queue to keep track of a range of incomplete policies that are candidates to become a solution. It repeatedly chooses from the queue the most promising candidate policy until it finds one that is a solution. If the chosen policy is not yet a solution, AND* improves it to generate new candidate policies, which are then inserted in the queue. The search continues until a solution is found or the queue becomes empty. AND* is sound and complete, and has optimality guarantees when using an appropriate heuristic function to guide the search. In this work, we study the structure of FOND policies and policy-space search aiming to develop a collection of techniques to improve the performance of AND*.

We study policy and state equivalences in the context of policy-space search. With the knowledge of policy equivalences, we can make AND* prune part of the search space by expanding only one policy from each equivalence class. We show that state symmetries can be used to broaden the concepts of policy equivalences and that it is possible to improve the computation of state symmetries in the context of FOND planning, using group theory techniques.

We present two contributions whose applications go beyond AND* and policy-space search. The first is a procedure, which we call *the concretizer*, that, given a restriction specifying which parts of the state space the agent is allowed to reach, constructs—in time polynomial in the size of this portion of the state space—a solution policy that precisely "matches" it, if and only if one exists. We equip AND* with the concretizer to provide it an extra mechanism for finding solutions. This modification makes it capable of performing more aggressive prunings in its standard policy-space search, without becoming incomplete, as it is able to recover pruned solutions in certain circumstances. The concretizer has the potential to foster new planning approaches, since it allows changing the objective of the search.

The second is a compression technique, which we call *the compressor*, that, given a solution policy defined over complete states, finds—through the use of integer programming—a partial-state policy that unambiguously represents the same information using the minimum number of partial states

possible (what can take exponentially less space). We test the compression technique, showing that the compressor is time and memory-efficient for the benchmark tasks we use in this work, and that applying the compressor in the solutions returned by AND*, makes them have state-of-the-art compactness in the majority of the benchmark domains. The compressor is not limited to be used by AND*, as it can be used to compress complete-state solutions provided by any planner.

Finally, we also evaluate applying some satisficing search techniques from the literature to further improve the effectiveness of AND*, and compare it with other planners from the literature.

In sum, this work presents and studies a collection of techniques that, in unison, are able to establish explicit policy-space search, initially introduced by Messa and Pereira (2023), as a competitive method for solving FOND planning tasks. FOND planning heuristics and other complementary topics introduced by Messa and Pereira (2023) are not part of this work.

1.1. Outline

This work is structured as follows:

- We start by presenting a deeper **background** on FOND planning and policy-space search (Section 2);
- Then, we describe the **experimental settings**: computational environment, benchmarks, resource limits, etc (Section 3);
- Then, we present the concept of search-space equivalence pruning and evaluate some possible concepts of **equivalences between policies**. Here we also present **the concretizer** (Section 4);
- We then proceed by presenting concepts of **state symmetries** and we use them to strengthen the policy equivalences (Section 5);
- Then, we present the **solution compressor** and evaluate it. Here we also enable and evaluate some **satisficing features** (Section 6);
- Finally, we compare AND* with **state-of-the-art** FOND planners (Section 7) and conclude (Section 8).

A helpful glossary of the main symbols and functions used in this work is provided on the last page of this work (Appendix A).

2. Background on FOND Planning and Policy-Space Search

A FOND planning task Π, in STRIPS (Fikes and Nilsson, 1971) formalism, is a 4-tuple $\langle \mathcal{F}, \mathcal{I}, \gamma, \mathcal{A} \rangle$, with $\mathcal{F}$ a finite set of *facts* that can be either true or false, $\mathcal{I} \subseteq \mathcal{F}$ the facts that are true in *the initial state* $s_{\mathcal{I}}$, $\gamma \subseteq \mathcal{F}$ the facts that we want to be true, i.e. *the goal*, and $\mathcal{A}$ a finite set of *actions*. $\mathcal{F}[s]$ are the facts that are true in a state s, and a state s_* is a *goal state* iff $\mathcal{F}[s_*] \supseteq \gamma$. There are $2^{|\mathcal{F}|}$ states, one for each truth valuation. We call $\mathcal{S}$ the set of all possible states, and $\mathcal{S}_* \subseteq \mathcal{S}$ the set with all goal states.

Each action $a \in \mathcal{A}$ is a pair $\langle \texttt{pre}(a), \texttt{effs}(a) \rangle$, with $\texttt{pre}(a) \subseteq \mathcal{F}$ the precondition of a and $\texttt{effs}(a) = \{\texttt{eff}_1(a), \texttt{eff}_2(a), \ldots \texttt{eff}_{n \geq 1}(a)\}$ the list of possible effects of applying a. An action a can be applied to a state s iff $\mathcal{F}[s] \supseteq \texttt{pre}(a)$. Each $\texttt{eff}_i(a)$ of a is a pair $\langle \texttt{del}_i(a), \texttt{add}_i(a) \rangle$, with $\texttt{del}_i(a) \subseteq \mathcal{F}$, $\texttt{add}_i(a) \subseteq \mathcal{F}$ and $\texttt{del}_i(a) \cap \texttt{add}_i(a) = \emptyset$. The application of a to a given state s non-deterministically results in a state $s' \in \texttt{succs}(s, a) = \{\texttt{succ}_1(s, a), \texttt{succ}_2(s, a), \ldots, \texttt{succ}_n(s, a)\}$, with $\mathcal{F}[\texttt{succ}_i(s, a)] = (\mathcal{F}[s] \setminus \texttt{del}_i(a)) \cup \texttt{add}_i(a)$.

An action is said to be *fair* (Cimatti et al., 2003), if each possible outcome of an action will occur infinitely often if the action is applied infinitely often in a given state. Otherwise, it is said to be *adversarial*. For simplicity, in this work we assume that all actions are fair, as done in some works in the literature (Pereira et al., 2022; Messa and Pereira, 2023; Muise et al., 2024). Nevertheless, most of the contributions presented in this work can be adapted to work in adversarial environments as well. The worst-case complexity of FOND planning is EXPTIME-complete in both cases (Littman, 1997; Mattmüller et al., 2010).

A *transition* t is a triplet $\langle s, a, s' \rangle$ with s a state, a an action applicable to s, and $s' \in \texttt{succs}(s, a)$. The start of t is $\texttt{begin}(t) = s$ and the end of t is $\texttt{end}(t) = s'$. And the action of t is $\mathcal{A}[t] = a$.

A *trajectory* ω is a sequence of transitions $\langle t_1, t_2, \ldots, t_n \rangle$, with $\texttt{end}(t_i) = \texttt{begin}(t_{i+1})$ for each $i \in \{1, 2, \ldots, n-1\}$. For every pair of states s and s', if $n \geq 1$, $\texttt{connects}(\omega, s, s')$ is defined to be $\top$ if $s = \texttt{begin}(t_1)$ and $s' = \texttt{end}(t_n)$, otherwise $\bot$. If $n = 0$, then the trajectory is empty and $\texttt{connects}(\omega, s, s')$ is defined to be $\top$ if $s = s'$, otherwise $\bot$.

A *policy* π is a partial function mapping non-goal states to actions, with $\pi[s]$ applicable to s for each $s \in \texttt{dom}(\pi)$, where $\texttt{dom}(\pi)$ denotes the set of states mapped by π, called the *domain* of π. We denote $\pi|_X = \{s \mapsto \pi[s] : s \in \texttt{dom}(\pi) \cap X\}$ for $X \subseteq \mathcal{S}$.

A π-*trajectory* is a trajectory $\omega = \langle t_1, t_2, \ldots, t_n \rangle$, with $\mathcal{A}[t_i] = \pi[\texttt{begin}(t_i)]$

for each $i \in \{1, 2, \ldots, n\}$. Following the guidance of a policy π under a non-adversarial environment, one has a chance of reaching a state s' starting from a state s if and only if there is a π-trajectory ω with $\texttt{connects}(\omega, s, s') = \top$.

The *reachability set* of π is $\texttt{reach}(\pi) = \{s' : \exists s \in \texttt{dom}(\pi)$ and $\exists \pi$-trajectory ω such that $\texttt{connects}(\omega, s, s') = \top\} = \texttt{dom}(\pi) \cup \bigcup_{s \in \texttt{dom}(\pi)} \texttt{succs}(s, \pi[s])$. The reachability set lists all the states that one, following the guidance of the policy, might reach starting from *any* of the states mapped by the policy. We call $\texttt{reach}(\pi, s) = \{s' : \exists \pi$-trajectory ω such that $\texttt{connects}(\omega, s, s') = \top\}$.

The *escape set* of π is $\texttt{escape}(\pi) = \texttt{reach}(\pi) \setminus \texttt{dom}(\pi)$. It contains the *unmapped* states that one may reach starting from a state mapped by the policy. We call $\texttt{escape}(\pi, s) = \texttt{reach}(\pi, s) \cap \texttt{escape}(\pi)$.

A policy π is *closed* iff $\texttt{escape}(\pi) = \emptyset$ and *goal-closed* iff $\texttt{escape}(\pi) \subseteq \mathcal{S}_*$. A policy π is *proper* iff $\texttt{escape}(\pi, s) \neq \emptyset$ for each $s \in \texttt{dom}(\pi)$. We note that if π is proper, then each $\pi' \subseteq \pi$ is proper as well. A policy that is simultaneously goal-closed and proper is called *strong-cyclic*[1].

A *solution* for Π is a strong-cyclic policy π_* with $s_\mathcal{I} \in S_* \cup \texttt{dom}(\pi_*)$. In other words, a solution is a policy that under a non-adversarial environment safely guides one from $s_\mathcal{I}$ *eventually* to a goal state, by addressing all the possible outcomes of the taken actions, and ensuring that: (1) from every state mapped by the policy it is not possible to reach a state that is not handled by the policy unless it is a goal state (goal-closedness); (2) from every state mapped by the policy there is always a chance to reach a state not handled by the policy (properness + fairness); and (3) the initial state is either a goal state or is mapped by the policy. One following a solution policy may possibly visit some states an arbitrary number of times before eventually reaching the goal, due to possible cyclic π_*-trajectories. Yet, while it does not reach the goal, it is ensured to always reach only states mapped by the policy, from where there will be a new chance to reach the goal.

We define $\texttt{remain}(\pi) = (\texttt{escape}(\pi) \setminus \mathcal{S}_*) \cup (\{s_\mathcal{I}\} \setminus (\mathcal{S}_* \cup \texttt{dom}(\pi)))$. The `remain` set captures if a policy is goal-closed and deals with the initial state. Specifically, $\texttt{remain}(\pi) = \emptyset$ iff π is goal-closed and $s_\mathcal{I} \in S_* \cup \texttt{dom}(\pi_*)$. Consequently, a *proper* policy π is a solution if and only if $\texttt{remain}(\pi) = \emptyset$.

We remind that a simplified glossary of the symbols and functions we most use throughout this work is present on the last page of this work (in

[1]We note that "Strong-cyclic" is a terminology that comes from the literature (Cimatti et al., 2003). It does not imply the existence of cyclic trajectories, but allows.

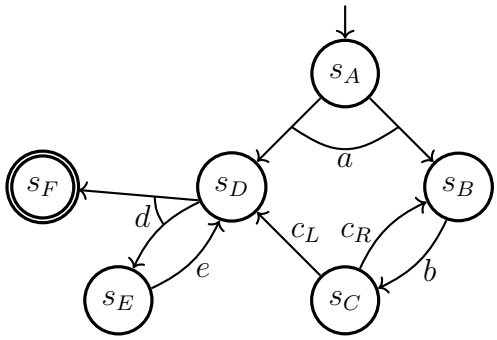

Figure 1: Example state space to illustrate the background concepts.

the Appendix). We now exemplify some of the presented concepts.

Example. Figure 1 depicts the state space of a FOND planning task. It has six states s_A, s_B, s_C, s_D, s_E, and s_F. The initial state is s_A, and s_F is the single goal state. There are six actions a, b, c_L, c_R, d, and e. The only non-deterministic actions are a — which can lead to either s_B or s_D from s_A — and d — which can lead to either s_E or s_F from s_D. We use it to exemplify some of the presented concepts regarding policies.

The policy $\pi_1 = \{s_D \mapsto d, s_E \mapsto e\}$ is a strong-cyclic policy, meaning that it is goal-closed and proper. It is goal-closed because $\texttt{escape}(\pi_1) = \{s_F\} \subseteq \mathcal{S}_*$. It is proper because for each $s \in \texttt{dom}(\pi_1) = \{s_D, s_E\}$, $\texttt{escape}(\pi_1, s) = \texttt{reach}(\pi_1, s) \cap \texttt{escape}(\pi_1) \neq \emptyset$, given that there is a π_1-trajectory leading s_D to s_F and another leading s_E to s_F. For instance, the π_1-trajectories $\langle\langle s_D, d, s_F\rangle\rangle$ and $\langle\langle s_E, e, s_D\rangle, \langle s_D, d, s_F\rangle\rangle$ respectively lead s_D and s_E to s_F. Although one may cycle an arbitrary number of times between s_D and s_E before reaching a solution if they follow π_1 actions starting from one of these states, the assumption of non-adversariality (fairness) ensures that the "incorrect" outcome of applying e will not occur forever and that the one leading them to s_F will eventually occur. Note that π_1 is not a solution because it does not cover the initial state (i.e. $s_A \notin \texttt{dom}(\pi_1)$). The only solution for this example task is $\pi_2 = \{s_A \mapsto a, s_B \mapsto b, s_C \mapsto c_L, s_D \mapsto d, s_E \mapsto e\}$. Note that π_2 is a superset of π_1.

The policy $\pi_3 = \{s_A \mapsto a, s_B \mapsto b, s_C \mapsto c_R, s_D \mapsto d, s_E \mapsto e\}$ is not a solution because it is not proper. It is not proper because $\texttt{escape}(\pi_3, s_B) = \emptyset$ and $\texttt{escape}(\pi_3, s_C) = \emptyset$. This occurs because $\texttt{succs}(s_B, \pi_3[s_B]) = \{s_C\}$ and $\texttt{succs}(s_C, \pi_3[s_C]) = \{s_B\}$. They form what we call a *deadlock*. Unlike the cycle between s_D and s_E, the cycle formed by s_B and s_C in π_3 is inescapable

because there is no outcome that leads outside it.

2.1. Searching for a Solution – The AND Algorithm*

Algorithm 1: AND*

```
1  π_I := ∅, Queue := {π_I}
2  while Queue ≠ ∅ :
3      Remove from Queue a policy π with least f(π)
4      if π is a solution :
5          return π
6      else:
7          for each π' ∈ successors(π) :
8              Insert π' in Queue
9  return ⊥ // unsolvable task
```

```
10 Method successors(π):
11     if remain(π) ≠ ∅ :
12         Select some state s from remain(π)
13         return {π ⊔ {s ↦ a} : a ∈ A applicable to s}
14     else:
15         return ∅
```

The A* with Non-Determinism (AND* for short) algorithm (Messa and Pereira, 2023) "generalizes" the A* algorithm (Hart et al., 1968) for FOND planning. It is a best-first heuristic search algorithm that uses a priority queue over policies to explore the *policy space* of Π and search for a solution policy. It starts with the empty policy $\emptyset$. At each iteration, it removes from the priority queue a policy π with lowest $f(\pi)$ value and returns it if it is a solution for Π. Otherwise, AND* expands π to generate successor[2] policies $\pi' = \pi \sqcup \{s \mapsto a\}$ for some state $s \in \texttt{remain}(\pi)$ and each action a

[2]We note that there are alternative ways of defining the successors of a given policy π. For instance, we could generate instead policies $\pi' \supseteq \pi$ with $\texttt{dom}(\pi') = \texttt{dom}(\pi) \sqcup \texttt{remain}(\pi)$ mapping *at once* every state in $\texttt{remain}(\pi)$ to some action. That would result in a "big-step" successor function. Messa and Pereira (2023) discuss "big-step" vs. "small-step".

applicable to s (given that there is one such state s, otherwise π is simply discarded). Each generated successor policy is inserted into the priority queue. If the priority queue becomes empty, then there is no solution for Π. Algorithm 1 shows the pseudocode for AND*.

AND* can be seen as a simple A* search performed in a different search space: the constructive policy space induced by the custom successor function and initial node AND* defines. AND* is sound and complete (Messa and Pereira, 2023).

2.1.1. Optimality and Heuristics

In FOND planning, one might want not just a solution policy but the most compact one. Alternatively, one might seek the solution with the minimal expected cost to reach the goal, after introducing a distribution of probabilities for the outcomes of the actions. In other words, there are multiple possible metrics for FOND planning that one might want to optimize. In principle, the $f(\pi)$ values that AND* uses to guide its search should evaluate how promising each candidate policy π is for becoming a good solution according to the metric we are trying to optimize. Formally, Messa and Pereira (2023) show that for any given metric φ, AND* always returns a solution π_* with minimal $\varphi(\pi_*)$ if the f function used is *admissible* and *goal-aware* in the context of the chosen metric[3]. They define admissibility and goal awareness for the f function as follows. The function f is admissible iff, for any given policy π that can become a solution through the addition of new mappings (i.e., with at least one solution policy $\pi_* \supseteq \pi$), $f(\pi) \leq \varphi(\pi_*)$ for any solution π_* that π can become. Moreover, f is goal-aware iff $f(\pi) = \varphi(\pi)$ whenever π is a solution.

Sometimes, it might also be useful to define g and h^* values for candidate policies. When aiming to find a solution policy with minimal domain size, we could say that the g-value (the "already paid cost") of a given policy π is $|\texttt{dom}(\pi)|$, and that the h^*-value (the "optimal remaining cost") of π is the minimum $|\texttt{dom}(\pi')|$ such that $\pi \sqcup \pi'$ is a solution for Π (or ∞ if no such π' exists). If we have a function h that estimates h^* in a never-pessimistic way (that is, $h(\pi) \leq h^*(\pi)$ for any policy π) and that never returns negative values, then a function $f(\pi) = |\texttt{dom}(\pi)| + h(\pi)$ is admissible and goal-aware

[3]Note that for some metrics, such as *maximizing* the domain size ($\varphi(\pi_*) = -|\texttt{dom}(\pi_*)|$), there exists no f function that simultaneously satisfies the properties of admissibility and goal-awareness.

for the domain size metric, and AND* using such a function only returns solution policies with minimal domain size.

Messa and Pereira (2023) present a somewhat complex function f called $\Delta_{\downarrow}$ that employs a classical planning heuristic as a subcomponent and exploits key structural characteristics of FOND planning to evaluate policies focusing on the metric of minimal domain size. If the given classical planning heuristic is admissible (in the classical sense of admissibility (Edelkamp and Schrödl, 2011, Section 1.6)), $\Delta_{\downarrow}$ becomes admissible and goals-aware for the domain size metric. In this work, we conduct most of our experiments using $\Delta_{\downarrow}$ equipped with the admissible classical planning heuristic $h^{\text{LM-Cut}}$ (Helmert and Domshlak, 2009) as AND*'s guiding function, because $h^{\text{LM-Cut}}$ was the best-performing admissible classical planning heuristic in the experiments of Messa and Pereira (2023), and $\Delta_{\downarrow}$ was the best-performing f function among the ones they presented. In the final sections, we switch to $\Delta_{\downarrow}$ equipped with the satisficing (non-admissible) classical planning heuristic h^{FF} (Hoffmann and Nebel, 2001) to maximize coverage, as preserving optimality is no longer of interest there. For more details on $\Delta_{\downarrow}$, including how it incorporates the use of classical planning heuristics to evaluate policies in the context of FOND planning, we refer readers to Messa and Pereira (2023).

3. Benchmarks and Experimental Settings

Throughout this work, we present techniques that can be used to improve AND*'s performance. We evaluate them through empirical experiments.

We use the same two benchmarks as Pereira et al. (2022) for our experiments. IPC-FOND, with 379 tasks over 12 FOND planning domains from the International Planning Competition (IPC), and NEW-FOND, introduced by Geffner and Geffner (2018), with 211 tasks over five FOND planning domains. The NEW-FOND benchmark was developed to mislead FOND planners. It has tasks with a large number of trajectories that lead to goal states but are not part of any solution.

As in Messa and Pereira (2023), we merge the *blocksworld-2* and *blocksworld-new* domains into a single domain called *blocksworld-advanced*, as they are actually equal domains. Moreover, we remove the tasks without solution, namely 25 tasks from the *first-responders* domain. These tasks are trivial because even the translator[4] we use to parse the tasks is able to rapidly detect

[4]We use PRP's (Muise et al., 2012) translator. It is based on the translator of Fast

	Domain	#
IPC-FOND	acrobatics	8
	beam-walk	11
	blocksworld-original	30
	blocksworld-advanced	55
	chain-of-rooms	10
	earth-observation	40
	elevators	15
	faults	55
	first-responders	75
	tireworld-triangle	40
	zenotravel	15
NEW-FOND	doors	15
	islands	60
	miner	51
	tireworld-spiky	11
	tireworld-truck	74

Table 1: Resulting benchmark domains.

their unsolvability. Table 1 shows the resulting 16 benchmark domains.

We run the experiments on an AMD Ryzen 9 3900X, using limits of 8 GB RAM and 30 minutes per task. We use greater g-value ($|\pi|$) as the AND* policy-selection tie-breaker, and we always select to map the most recently added state $s \in \texttt{remain}(\pi)$ when generating the successors of a policy π. Additionally, we make AND* not distinguish between two distinct goal states (i.e., from AND*'s perspective, all generated goal states are the same). This can be seen as a trivial application of the concept of state symmetries. The used code and data are publicly available on Zenodo (Messa and Pereira, 2026).

4. Policy Equivalence Pruning

An equivalence between policies is any reflexive, symmetric, and transitive binary relation between policies. It has no inherent meaning unless applied in some context. In this work, we study policy equivalences in the context of FOND planning and the search performed by AND*.

Downward (Helmert, 2006), and not just parse tasks written in the PDDL format (McDermott, 2000), but also makes some static analysis of the tasks. The time taken by the translator is counted for the time limit of the runs.

Let $\sim$ be a given equivalence relation between policies. We intend to modify AND* so that if it ever expands a given candidate policy π, then it will not later expand any other candidate policy π' that is considered equivalent to π under $\sim$, effectively pruning part of the search space that AND* has to explore.

In this work, we are particularly interested in equivalence relations that can enable as much pruning of the search space as possible, without compromising the soundness and completeness of AND*. In this section, we study two basic concepts of equivalence between policies.

4.1. AND* with Equivalence Pruning

In order to employ equivalence pruning, we need to store information about the policies that were already expanded. The naive way to do this would be to store each policy in a set, and then, when expanding a new policy, check if any policy in the set is equivalent to it. However, this would neither be time nor memory-efficient. Therefore, we focus on equivalence relations defined in the following way. Let `sign` be a function that maps policies to objects called *signatures*. Given such a function, we define an equivalence relation $\sim$ between policies by declaring $\pi_1 \sim \pi_2$ iff $\texttt{sign}(\pi_1) = \texttt{sign}(\pi_2)$. Note that this way the signature of a policy is the representative of its equivalence class.

Algorithm 2 shows the pseudocode for AND* with equivalence pruning. It uses a set called `Done` to store the signatures of the policies that were already expanded. At the removal of a new policy π, it checks if $\texttt{sign}(\pi)$ is in `Done`. If it is, then π is skipped. Otherwise, $\texttt{sign}(\pi)$ is added to `Done`, and π is normally expanded.

Note that Algorithm 1 is a special case of Algorithm 2 because they are equivalent when Algorithm 2's `sign` is the identity (i.e., $\texttt{sign}(\pi) = \pi$)[5].

4.2. Relation to A*

A* (Hart et al., 1968) is a best-first heuristic search algorithm that uses a priority queue to explore a search space. It starts with just the initial node in the queue. Each node is a pair $n = \langle s, n' \rangle$, with s a state and n' the parent node of n ($\perp$ if n is the initial node), which can be used to reconstruct the path from the initial state to s. A* repeatedly removes from the queue

[5]No policy π is generated twice by AND* (Messa and Pereira, 2023).

Algorithm 2: AND* with Equivalence Pruning

```
1  π_I := ∅, Queue := {π_I}
2  while Queue ≠ ∅ :
3      Remove from Queue some policy π with least f(π)
4      if π is a solution :
5          return π
6      else if sign(π) ∉ Done :
7          Insert sign(π) in Done
8          for each π' ∈ successors(π) :
9              Insert π' in Queue
10 return ⊥ // unsolvable task
```

the node n with the lowest $f(n)$ value, until it finds a goal state. When the heuristic function used to compute the f-values is admissible and goal-aware, A* always finds an optimal solution. As described in Subsection 2.1, AND* can be seen as A* performed in a policy space instead of state space. However, without the need for the duplicate detection described next.

In order to avoid expanding the same states multiple times without need, A* employs duplicate detection. It uses a map called `Closed` to store the states of the nodes that were already expanded. In this map, each state is mapped to the least g-value of any already expanded node containing it. Then, when expanding a node n, it checks if the state s of n is in `Closed`. If it is, it expands n only if $g(n) <$ `Closed`$[s]$ (in which case it updates `Closed`$[s]$ to $g(n)$). Otherwise, it skips n. A* with duplicate detection also ensures optimality. Further improvements can be done in case the heuristic is determined *"consistent"* (Edelkamp and Schrödl, 2011). In that case, A*'s `Closed` map can be replaced by simply a set of states, allowing for a stronger pruning, while still ensuring optimality.

The AND* algorithm is said to be a "generalization" of A* for FOND planning (Messa and Pereira, 2023) because it produces the exact same search as a state-space search A* (without duplicate detection) when the input FOND task is actually also a classical planning task (when all actions have only one outcome possible, and thus are deterministic), while it is additionally able to handle tasks with non-determinism. Then, our intent with the AND* algorithm with equivalence pruning is to design a best-effort generalization

of A* *with* duplicate detection, for FOND planning.

For simplicity, in this work we focus on the duplicate detection version that uses a set instead of a map. In this work, we do not study consistency for FOND planning. Our main goal is to evaluate different equivalence relations, while understanding their guarantees.

Hereafter, we call the `Closed` set "`Done`" in the context of AND*, to avoid confusion with the property of "closedness" of policies. Moreover, when we hereafter mention "AND*", we will be referring to Algorithm 2, not Algorithm 1, unless explicitly stated otherwise.

4.3. Which `sign` *to Use?*

Since using $\texttt{sign}(\pi) = \pi$ is the same as not having equivalence pruning, we would like to have a better concept of equivalence than identity. The AND*'s equivalence pruning analogous to looking at the state of an A* classical planning node (to decide whether to prune it or not) would be to use $\texttt{sign}(\pi) = \texttt{escape}(\pi)$. This is the case because when a policy π is simply a single trajectory starting from the initial state—as it is the case for every proper policy in the context of classical planning—$\texttt{escape}(\pi)$ is a singleton set containing *precisely* the state that it leads to.

Then, the key question is how AND* would behave, given an arbitrary FOND planning task, when $\texttt{sign} = \texttt{escape}$. This is the key question because when the input task has only deterministic actions, AND* with $\texttt{sign} = \texttt{escape}$ will search exactly like a state-space search A* with duplicate detection would in classical planning, since that task would also be a classical planning task. It is, however, not clear at the first glance how it would behave when the task *has* non-deterministic actions. Next, we present an example state space to analyze this possibility and show what can go wrong when AND* with $\texttt{sign} = \texttt{escape}$ receives FOND tasks that are not classical planning tasks.

The example state space, depicted in Figure 2, has seven states s_A, s_B, s_C, s_C, s_E, s_F, and s_X; and seven actions a, a_{bad}, b, c, d_L, d_R, and e.. The initial state is s_A, the single goal state is s_F, and s_X is a dead-end state. The only two non-deterministic actions are b (which can lead to either s_E, s_D, or s_C from s_B) and a_{bad} (which can lead to either s_B or s_X from s_a).

Let π_L and π_R be the two policies with domain $\{s_A, s_B, s_D\}$, but differing on which action apply to s_D. $\pi_L = \{s_A \mapsto a, s_B \mapsto b, s_D \mapsto d_L\}$ maps s_D to the left action that leads to s_E, while $\pi_R = \{s_A \mapsto a, s_B \mapsto b, s_D \mapsto d_R\}$ maps s_D to the right action that leads to s_C. Both π_L and π_R have $\{s_C, s_E\}$

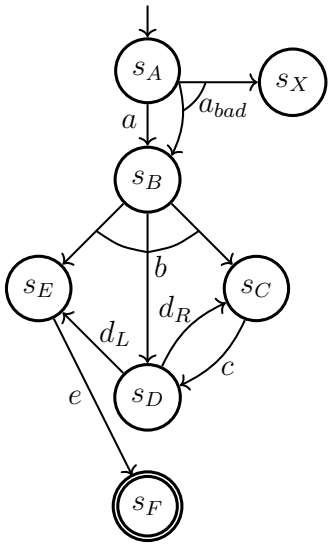

Figure 2: Example state space to show the incompleteness of AND* with equivalence pruning using $\texttt{sign}(\pi) = \texttt{escape}(\pi)$ or $\texttt{sign}(\pi) = \langle \texttt{dom}(\pi), \texttt{escape}(\pi) \rangle$.

escape states. Therefore, if $\texttt{sign} = \texttt{escape}$, we have $\pi_L \sim \pi_R$. But while π_L is two mappings from becoming a solution, π_R cannot become a solution since the only mapping possible for s_C – to the action c – will lead to a successor policy with a deadlock with s_D. Thus, if we set $\texttt{sign} = \texttt{escape}$, AND* can discard all solution-leading policies and incorrectly return $\bot$ for a task with solutions. Note that the same problem would have arisen even if we had set $\texttt{sign}(\pi) = \langle \texttt{dom}(\pi), \texttt{escape}(\pi) \rangle$ for each policy π since $\pi_L \sim \pi_R$ would still be the case.

4.4. The Concretizer

We just showed that AND* using $\texttt{sign}(\pi) = \texttt{escape}(\pi)$ or $\texttt{sign}(\pi) = \langle \texttt{dom}(\pi), \texttt{escape}(\pi) \rangle$ may fail to find a solution for solvable tasks. This happens because two policies π_1 and π_2 with the same signature do not necessarily have the same mappings – and in FOND planning, not only do the escape states matter, but also how we will act after we happen to cycle back to each domain state.

In this subsection, we present a procedure that effectively allows a sound and complete use of $\texttt{sign}(\pi) = \langle \texttt{dom}(\pi), \texttt{escape}(\pi) \rangle$ equivalence pruning. Given only the sets $\langle \texttt{dom}(\pi), \texttt{escape}(\pi) \rangle$ — i.e. without the mappings — this procedure produces a solution policy if the sets match the ones from a

solution policy, in time polynomial in $|\texttt{dom}(\pi)| + |\Pi|$,[6] where $|\Pi|$ is the size of the compact representation of the task, which is usually of the same order of $|\mathcal{F}| + \sum_{a \in \mathcal{A}} |\texttt{effs}(a)|$. Namely, we present a procedure (Algorithm 3), called *the concretizer*, that — given a domain set D and a escape set E — produces a *proper* policy π with $\texttt{dom}(\pi) = D$ and $\texttt{escape}(\pi) \subseteq E$ iff exists any, in $O(|D|^2 \cdot |D \cup E| \cdot |\Pi|)$ time[7]. Otherwise, it returns $\perp$.

Algorithm 3: Policy Concretizer

Input: $\langle D, E \rangle$ `// a "hollow" policy (without mappings) with` $D \cap E = \emptyset$
$R \coloneqq D \sqcup E$
$\pi \coloneqq \emptyset$
$D_{handled} \coloneqq \emptyset$
$D_{remaining} \coloneqq D$
while $D_{remaining} \neq \emptyset$ **:**
  if $\exists s \in D_{remaining}$ **and** $\exists a \in \mathcal{A}$ applicable to s **such that** $\texttt{succs}(s, a) \cap (E \sqcup D_{handled}) \neq \emptyset$ **and** $\texttt{succs}(s, a) \subseteq R$ **:**
    Choose such an s and a.
    $\pi \coloneqq \pi \sqcup \{s \mapsto a\}$
    $D_{handled} \coloneqq D_{handled} \sqcup \{s\}$
    $D_{remaining} \coloneqq D_{remaining} \setminus \{s\}$
  else:
    return $\perp$ `// unconcretizable hollow policy`
return π

The concretizer starts with an empty policy π and uses a pair of sets $D_{handled}$ and $D_{remaining}$ to keep track of which states of D it has already defined a mapping for. It repeatedly finds some state-action pair $\langle s, a \rangle$ with $s \in D_{remaining}$ such that, if we add $s \mapsto a$ to π, π remains proper, while completely disregarding mappings $s \mapsto a$ such that $\texttt{succs}(s, a) \not\subseteq E \cup D$. It ensures that the policy π remains proper, by ensuring that the mapping to

[6] For any policy π, $|\texttt{escape}(\pi)| \leq (\max_{a \in \mathcal{A}} |\texttt{effs}(a)|) \cdot |\texttt{dom}(\pi)|$, so we do not need to factor the size of `escape` here.

[7] With a more efficient implementation, the concretizer runs in $O(|D| \cdot |D \cup E| \cdot |\Pi|)$ time.

be added $s \mapsto a$ has $\texttt{succs}(s, a) \cap E \neq \emptyset$ or $\texttt{succs}(s, a) \cap D_{handled} \neq \emptyset$. That takes into account that $E \cap \texttt{dom}(\pi)$ is always $\emptyset$, and that, for each s' already in $D_{handled}$, we have $\texttt{reach}(\pi, s') \cap E \neq \emptyset$ inductively ensured. In the end, π will be a proper policy with $\texttt{dom}(\pi) = D$. Moreover, $\texttt{reach}(\pi)$ will be a subset of $D \cup E$ due to the filtering of mappings, meaning that $\texttt{escape}(\pi)$ will be a subset of E. We formally prove the soundness and completeness of the concretizer after exemplifying its execution.

4.4.1. Example

We use our running example of state space (Figure 2) to exemplify the concretizer execution. We choose $D = \{s_A, s_B, s_C, s_D, s_E\}$ and $E = \{s_F\}$ because there are some ways to go wrong when mapping states in D and we want to show that, by construction, the concretizer is hindered from including the incorrect mappings. Figure 3 illustrates every step of the execution example we will show. It displays in blue the states in $E \cup D_{handled}$, in black the states in $D_{remaining}$, and in red the states outside $E \cup D$. Mappings that were already added to π are also shown in blue. Transitions that lead to a blue state are highlighted in green.

Initially, $D_{handled} = \emptyset$. Therefore, in order for a mapping $s' \mapsto a'$ to be added to π, $\texttt{succs}(s', a')$ must include s_F, since $F = \{s_F\}$. The concretizer determines that $s_E \mapsto e$ is only the mapping that fulfills that requirement (Figure 3a). Moreover, $\texttt{succs}(s_E, e) \subseteq D \cup F$ is also true (the other requirement). Therefore, the concretizer adds $s_E \mapsto e$ to π. Now $D_{handled} = \{s_E\}$. Therefore, the requirement of $\texttt{succs}(s, a) \cap \{s_F\} \neq \emptyset$ is weakened to $\texttt{succs}(s', a') \cap \{s_E, s_F\} \neq \emptyset$, allowing candidate mappings $s_B \mapsto b$ and $s_D \mapsto d_L$ to be added (Figure 3b). Mapping s_D to d_L avoids the problem of deadlock that would happen if we map s_D to d_R. In order to show that the mapping s_D to d_R will never be an option, we make the concretizer choose not to add $s_D \mapsto d_L$ yet, but first $s_B \mapsto b$. Now, $\pi = \{s_B \mapsto b, s_E \mapsto e\}$, and the requirement is weakened to $\texttt{succs}(s', a') \cap \{s_B, s_E, s_F\} \neq \emptyset$. Three mappings meet the weakened requirement, $s_D \mapsto d_L$ (as it already met its stronger version), $s_A \mapsto a$, and $s_A \mapsto a_{bad}$ (Figure 3c). Another way the concretizer could go wrong was if it decided to add $s_A \mapsto a_{bad}$. However, this is the only mapping that does not meet the other requirement: $\texttt{succs}(s_A, a_{bad}) = \{s_B, s_X\} \not\subseteq (D \cup F) = \{s_A, s_B, s_C, s_D, s_E, s_F\}$. Therefore, $s_A \mapsto a_{bad}$ is discarded. The candidates are $s_D \mapsto d_L$ and $s_A \mapsto a$. Let's say the concretizer decides to delay adding $s_D \mapsto d_L$ once more, and adds $s_A \mapsto a$ to π. Now $\pi = \{s_A \mapsto a, s_B \mapsto b, s_E \mapsto e\}$, and the requirement is weakened

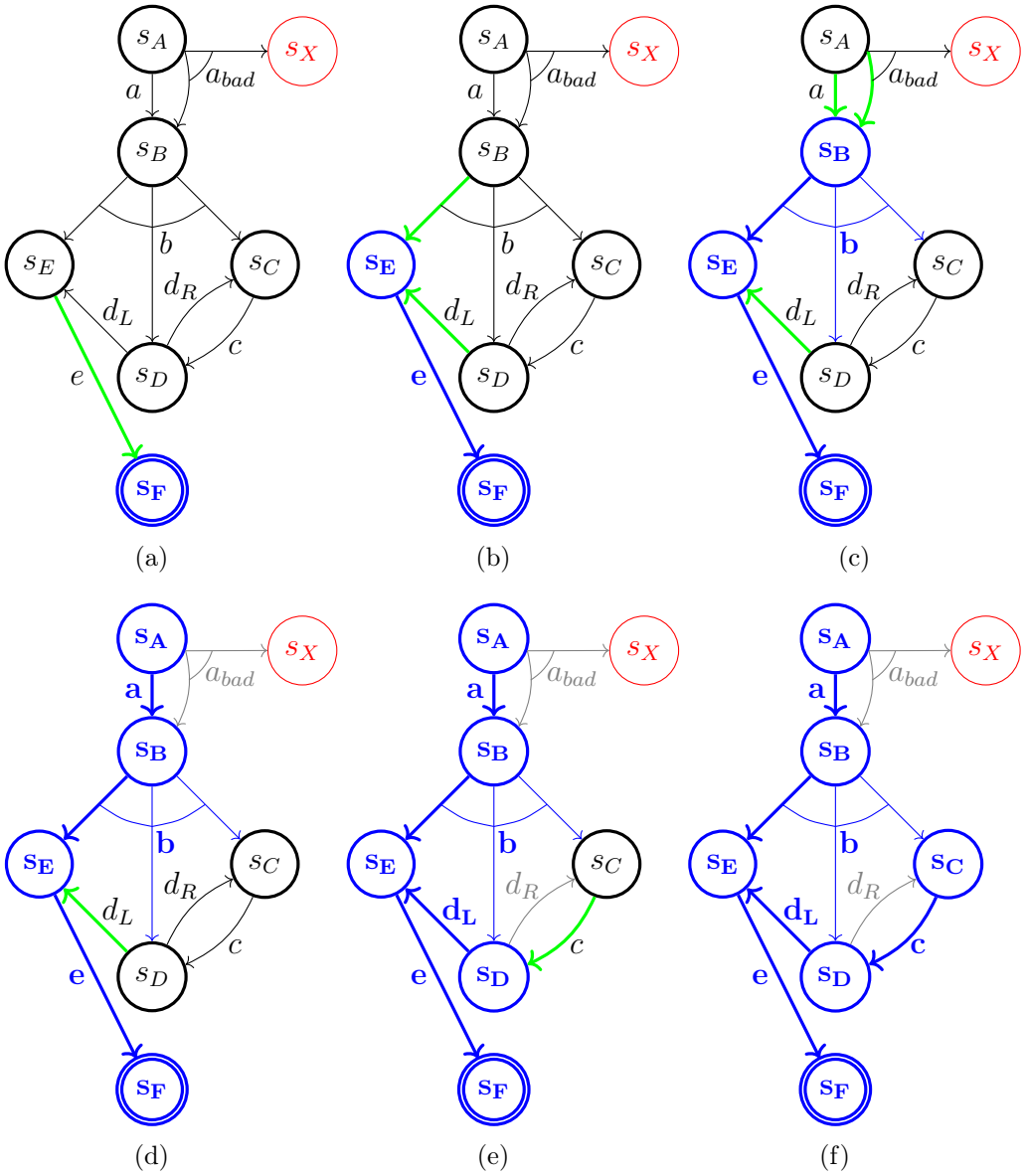

Figure 3: Illustration of the concretizer execution example.

to $\texttt{succs}(s', a') \cap \{s_A, s_B, s_E, s_F\} \neq \emptyset$. Now, the only mapping that meets both requirements is $s_D \mapsto d_L$ (Figure 3d). The mapping of $s_D \mapsto d_R$ that would construct an improper policy with the s_D and s_C deadlock was never available. Since $s_D \mapsto d_R$ leads only to s_C, it would only be available if we had handled s_C. Given that we could not, we have no option but to map s_D to another action, one that leads to some handled state. $s_D \mapsto d_L$ is added to π. Now, $\pi = \{s_A \mapsto a, s_B \mapsto b, s_D \mapsto d_L, s_E \mapsto e\}$. Finally, $s_C \mapsto c$ meets the requirements, as s_D is now in $\texttt{dom}(\pi) = D_{handled}$ (Figure 3e). The concretizer adds $s_C \mapsto c$ to π and returns because all states in D are mapped (Figure 3f). The returned policy $\pi = \{s_A \mapsto a, s_B \mapsto b, s_C \mapsto c, s_D \mapsto d_L, s_E \mapsto e\}$ is proper, has $\texttt{dom}(\pi) = D$, and $\texttt{escape}(\pi) \subseteq F$.

4.4.2. Theoretical Properties

We now prove the soundness and completeness of the concretizer.

Lemma 1. *The concretizer always terminates.*

Proof. At each iteration, either the concretizer ends returning $\perp$, or it decreases the size of $D_{remaining}$ by one. Thus, since the input sets are finite, the number of iterations is finite. Then, since each instruction takes a finite time, each execution of the concretizer must eventually terminate. □

Lemma 2. *If the concretizer returns a policy π, then it is proper, $\texttt{dom}(\pi) = D$, and $\texttt{escape}(\pi) \subseteq E$.*

Proof. The return must have occurred at Line 14. We show by induction on the iterations of the concretizer that π had $\texttt{dom}(\pi) \subseteq D$, $\texttt{reach}(\pi) \subseteq D \cup E$ and $\forall s \in \texttt{dom}(\pi), \texttt{reach}(\pi, s) \cap E \neq \emptyset$ (implying properness) throughout the entire execution. By proving that, we will achieve the desired conclusion, taking into account that when Line 14 is reached, $D_{remaining} = D \setminus \texttt{dom}(\pi) = \emptyset$, and therefore $\texttt{dom}(\pi) = D$.

- Before the first iteration, the hypothesis is trivially true since $\pi = \emptyset$.
- We show that if, at the beginning of an iteration, the hypothesis is true, then it must also be at the end of that iteration. Note that since $\perp$ was never returned, then the if-statement was satisfied at each iteration.
  - Given that $\texttt{dom}(\pi) \subseteq D$, then $\texttt{dom}(\pi \cup \{s \mapsto a\}) \subseteq D$ by choice of $s \in D_{remaining} = D \setminus \texttt{dom}(\pi)$.

– Given that $\texttt{reach}(\pi) \subseteq D \cup E$, then $\texttt{reach}(\pi \cup \{s \mapsto a\}) \subseteq D \cup E$ by choice of s and a with $\texttt{succs}(s, a) \subseteq D \cup E$.
– Given that each $s' \in \texttt{dom}(\pi)$ π-reaches a state in E, then each $s' \in \texttt{dom}(\pi \cup \{s \mapsto a\})$ $(\pi \cup \{s \mapsto a\})$-reaches a state in E by choice of s and a with $\texttt{succs}(s, a) \cap (E \cup \texttt{dom}(\pi)) \neq \emptyset$ (since $\texttt{dom}(\pi) = D_{handled}$). □

Lemma 3. *The concretizer is correct whenever there is no proper policy π' with $\texttt{dom}(\pi') = D$ and $\texttt{escape}(\pi') \subseteq E$.*

Proof. By contraposition, assume it is not correct. Then, for some input $\langle D, E \rangle$ for which there is no proper policy π' with $\texttt{dom}(\pi') = D$ and $\texttt{escape}(\pi') \subseteq E$, it returns something different from $\bot$ (otherwise it would be correct). Thus, it returns some policy π at Line 14. Then, by applying Lemma 2, we obtain a contradiction with our assumption of the nonexistence of a proper policy π' with $\texttt{dom}(\pi') = D$ and $\texttt{escape}(\pi') \subseteq E$. □

Lemma 4. *The concretizer is correct whenever there is some proper policy π' with $\texttt{dom}(\pi') = D$ and $\texttt{escape}(\pi') \subseteq E$.*

Proof. By contraposition, assume it is not correct. Then, for some input $\langle D, E \rangle$ for which there is some proper policy π' with $\texttt{dom}(\pi') = D$ and $\texttt{escape}(\pi') \subseteq E$, it returns something different from a policy with such properties. Then, by Lemma 2, it must have returned no policy at all. The only alternative is that it returned $\bot$ at Line 13. Then, at the beginning of some iteration it occurred that did not exist $s \in D_{remaining} = D \setminus \texttt{dom}(\pi)$ and $a \in \mathcal{A}$ with a applicable to s, $\texttt{succs}(s, a) \cap (E \cup \texttt{dom}(\pi)) \neq \emptyset$, and $\texttt{succs}(s, a) \subseteq D \cup E$. We show that that could not have occurred under the current assumptions. Let $\pi'' = \{s \mapsto \pi'[s] : s \in \texttt{dom}(\pi') \setminus \texttt{dom}(\pi)\} = \{s \mapsto \pi'[s] : s \in D \setminus \texttt{dom}(\pi)\}$. Since $\pi'' \subseteq \pi'$ and π' is proper, π'' is proper. Therefore $\texttt{escape}(\pi'') \neq \emptyset$ and $\exists s \in \texttt{dom}(\pi'')$ such that $\texttt{succs}(s, \pi[s]) \cap \texttt{escape}(\pi'') \neq \emptyset$. We choose such a state s and action $a = \pi[s]$, and we note that $\texttt{succs}(s, a) \subseteq D \cup E$ because $\texttt{reach}(\pi') \subseteq D \cup E$ and $\pi'[s] = \pi''[s] = a$ (by construction of π''). Moreover, since $\texttt{succs}(s, a) \cap \texttt{escape}(\pi'') \neq \emptyset$ and $\texttt{escape}(\pi'') = \texttt{reach}(\pi'') \setminus \texttt{dom}(\pi'') \subseteq \texttt{reach}(\pi') \setminus \texttt{dom}(\pi'') \subseteq (D \cup E) \setminus (D \setminus \texttt{dom}(\pi)) \subseteq E \cup \texttt{dom}(\pi)$, then it must be true that $\texttt{succs}(s, a) \cap (E \cup \texttt{dom}(\pi)) \neq \emptyset$. Therefore, there exists $s \in D \setminus \texttt{dom}(\pi)$ and $a \in \mathcal{A}$ applicable to s, with $\texttt{succs}(s, a) \cap (E \cup \texttt{dom}(\pi)) \neq \emptyset$ and $\texttt{succs}(s, a) \subseteq D \cup E$. Contradiction. □

Theorem 1. *The concretizer is sound and complete.*

Proof. By combination of Lemmas 1, 3, and 4. □

4.5. Domain-Escape Equivalence Pruning

The main consequence of the existence of the concretizer is that we do not need to find a solution π_* during a policy-space search. It suffices to find the domain set and a superset of the escape set of some solution π_*, if the second set is composed only of goal states. We prove that if we find such sets, we can call the concretizer to obtain a solution. First, it will not return $\bot$ due to the existence of π_*. Then, let π' be the proper policy it returns. Since the provided E set is composed only of goal states, π' is goal-closed. Finally, we know the returned policy handles the initial state because π_* does so, and they share the same domain set. Therefore, π' is a solution.

The following question is how we can find such sets. As one can expect, finding a valid pair $\langle D, E\rangle$ that satisfies the condition should be a hard task. While we could change the constructive search space to explicit search for those pairs, we decided to proceed with the same policy search space and use the `dom` and `escape` sets of candidate policies as potential guesses. Importantly, we prove a key result in this section: in order to make AND* sound and complete with domain-escape equivalence pruning, we only need to run the concretizer on $\langle \texttt{dom}(\pi), \texttt{escape}(\pi)\rangle$ for the visited policies π that have $\texttt{remain}(\pi) = \emptyset$ (Algorithm 4). As a bonus, if the concretizer does not return $\bot$, it is guaranteed in such cases that the policy it returned is a solution, so we do not really need to check it is a solution.

We revisit our running example from Figure 2 to show that, by using the concretizer as described in Algorithm 4, we find the solution even if we use $\texttt{sign}(\pi) = \langle \texttt{dom}(\pi), \texttt{escape}(\pi)\rangle$. We note the reader might find useful to follow Figure 4 while reading the following example. $\pi_{\mathcal{I}} = \emptyset$ is the initial policy. It has two successors $\pi_1 = \{s_A \mapsto a\}$ and $\pi_2 = \{s_A \mapsto a_{bad}\}$. π_2 leads to no successor because there is no applicable action for s_X and $\texttt{remain}(\pi_2) = \{s_X\}$. π_1 has a single successor $\pi_3 = \{s_A \mapsto a, s_B \mapsto b\}$. $\texttt{remain}(\pi_3) = \texttt{escape}(\pi_3) = \{s_C, s_D, s_E\}$. AND* arbitrarily chooses a state in $\texttt{remain}(\pi_3)$ to map. In order to reach the problematic situation we previously showed, let's say AND* chooses to map first s_D. π_3 generates two successors $\pi_L = \{s_A \mapsto a, s_B \mapsto b, s_D \mapsto d_L\}$ and $\pi_R = \{s_A \mapsto a, s_B \mapsto b, s_D \mapsto d_R\}$. π_L and π_R are the two policies we have brought attention to in previous discussions. Note that $\pi_L \sim \pi_R$, since they have the same `dom` and `escape`

Algorithm 4: AND* with equivalence pruning and the concretizer

```
1  π_I := ∅, Queue := {π_I}
2  while Queue ≠ ∅ :
3      Remove from Queue some policy π with least f(π)
4      if remain(π) = ∅ :
5          if π is a solution :
6              return π
7          if concretizer(dom(π), escape(π)) is a solution :
8              return concretizer(dom(π), escape(π))
9      if sign(π) ∉ Done :
10         Insert sign(π) in Done
11         for each π' ∈ successors(π) :
12             Insert π' in Queue
13 return ⊥ // unsolvable task
```

sets. While π_L can become a solution, π_R cannot because it will have a deadlock on s_D and s_C when we eventually map $s_C \in \texttt{remain}(\pi_R)$ to c. If we try to expand π_R before π_L, we will prune π_L. This was a critical problem before the concretizer. Now, it causes no harm. After expanding π_R, we generate either $\pi_6 = \{s_A \mapsto a, s_B \mapsto b, s_D \mapsto d_R, s_C \mapsto c\}$ or $\pi_7 = \{s_A \mapsto a, s_B \mapsto b, s_D \mapsto d_R, s_E \mapsto e\}$, since $\texttt{remain}(\pi_R) = \{s_C, s_E\}$. Regardless of what the successor of π_R is, the successor of its successor will be $\pi_8 = \{s_A \mapsto a, s_B \mapsto b, s_C \mapsto c, s_D \mapsto d_R, s_E \mapsto e\}$. π_8 is the first policy to have an empty `remain` set. It has an empty `remain` set because it is goal-closed (all states in $\texttt{escape}(\pi_8) = \{s_F\}$ are goal states), and it deals with the initial state ($s_A \in \texttt{dom}(\pi_8)$). The only reason it is not a solution is that it is not proper, as it has a deadlock on c and d. Since $\texttt{remain}(\pi_8) = \emptyset$, the concretizer will be called on $\langle \texttt{dom}(\pi_8), \texttt{escape}(\pi_8) \rangle = \langle \{s_A, s_B, s_C, s_D, s_E\}, \{s_F\} \rangle$, deducing the solution $\pi_* = \{s_A \mapsto a, s_B \mapsto b, s_C \mapsto c, s_D \mapsto d_L, s_E \mapsto e\}$[8], which π_L (the policy pruned by π_R) would lead to if not pruned. This is the main benefit of having the concretizer. If the only problem that makes a policy not a solution

[8]The process of deducting $\{s_A \mapsto a,\ s_B \mapsto b,\ s_C \mapsto c,\ s_D \mapsto d_L,\ s_E \mapsto e\}$ from $\langle \{s_A, s_B, s_C, s_D, s_E\}, \{s_F\} \rangle$ is displayed as an example earlier on this section.

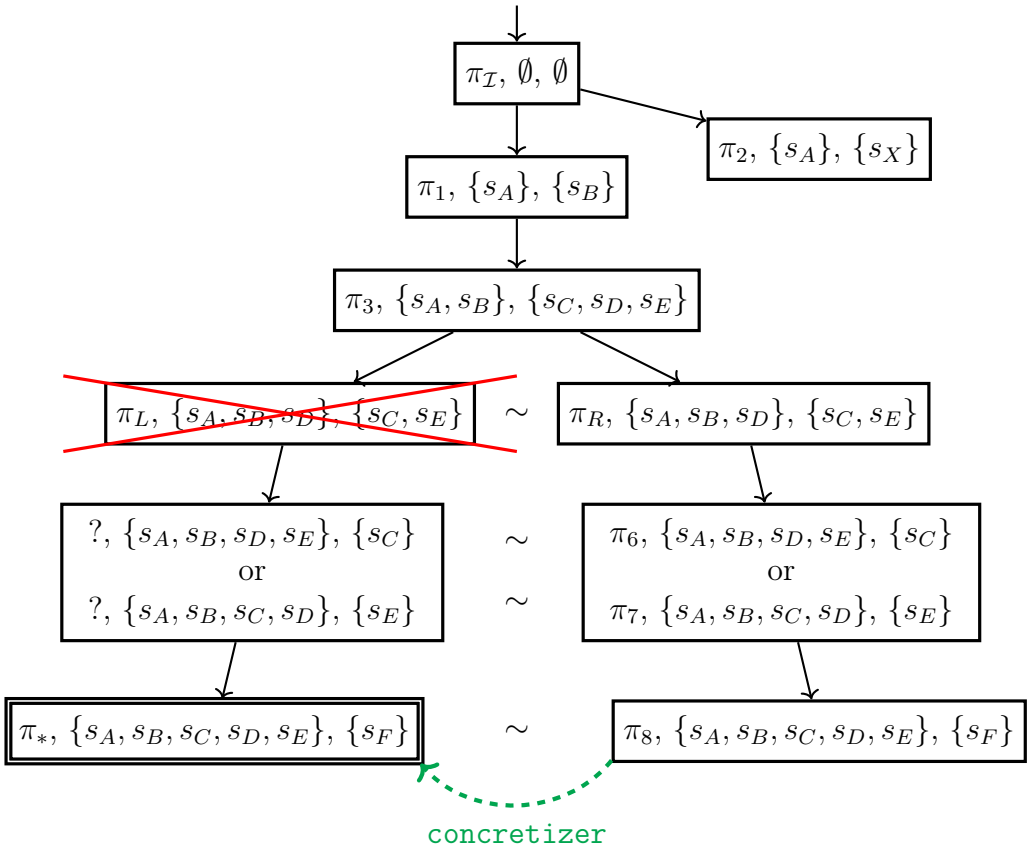

Figure 4: The policy space from the execution example that illustrates how the use of the concretizer fixes the issues of AND* using domain-escape.

is improperness, and its `dom` and `escape` sets are correct, the concretizer will fix the properness by finding the correct mappings. Figure 4 illustrates how using the concretizer fixes what previously was considered a mis-pruning. It displays the `dom` and `escape` set of each policy on the generated policy space.

4.5.1. Theoretical Properties

We now formally prove that AND* with the concretizer is sound and complete when using domain-escape equivalence pruning. We also prove that it returns solutions with minimal domain size when using the $\Delta_{\downarrow}(h^{\text{LM-Cut}})$ heuristic.

Arguing that AND* always either fails or returns solutions, (absence of false positives) is fairly straightforward, regardless of the equivalence relation

used. The difficult part is proving that it always returns a solution when the given task is solvable (absence of false negatives), despite the introduced pruning. One of the key ideas for proving the latter will be to show that if there exists a sequence of mappings that transforms one given policy π_1 into a solution, then the same sequence of mappings also transforms any equivalent policy $\pi_2 \equiv \pi_1$ into a policy where the concretizer is able to retrieve a solution. With that, we can show that whenever we prune a policy that would lead to a solution, there is another policy, with the same signature (and thus same size), that also leads to a solution and that was successfully expanded before.

Lemma 5. *Regardless of the choice of* `sign`*, AND* never returns false positives (i.e., a policy that is not a solution). Moreover, if* `sign` *can be computed in finite time, AND* always terminates.*

Proof. By design, AND* only has return statements that return solutions. Moreover, it always terminates because the number of successors of a policy is finitely bounded by $|\mathcal{A}|$, the maximum size a policy may have is $|S|$, and every successor of a policy has an incremented size. □

Definition 1. *Let a solution policy be optimal if it has minimal domain size among all solution policies. We call a policy π a "good candidate policy" iff there exists a policy π' subset of an optimal solution policy π_* (i.e. $\pi' \subseteq \pi_*$) with* $\texttt{dom}(\pi') = \texttt{dom}(\pi)$ *and* $\texttt{escape}(\pi') = \texttt{escape}(\pi)$*. In such a case, we denote $P(\pi) = \top$. Otherwise, we denote $P(\pi) = \bot$.*

Lemma 6. *For any given policy π, if $P(\pi) = \top$, then either Algorithm 4 returns a solution when evaluating π, or $\exists \pi' \in \texttt{successors}(\pi)$ such that $P(\pi') = \top$.*

Proof. Assume, for the sake of contradiction, that neither Algorithm 4 returns a solution when evaluating π, nor π has a successor π' with $P(\pi') = \top$. Let π_B be the policy subset of an optimal solution policy π_* with $\texttt{dom}(\pi_B) = \texttt{dom}(\pi)$ and $\texttt{escape}(\pi_B) = \texttt{escape}(\pi)$, that exists due to $P(\pi) = \top$. Firstly, since Algorithm 4 does not return a solution when evaluating π, then π is not a solution and either $\texttt{remain}(\pi) \neq \emptyset$ or the concretizer returns $\bot$ for $\langle \texttt{dom}(\pi), \texttt{escape}(\pi) \rangle$. However, the latter could not have happened because of the existence of π_B, which is proper as it is a subset of a solution, so $\texttt{remain}(\pi) \neq \emptyset$ must be the case. Then, $\exists s \in \texttt{remain}(\pi)$ such that for every action a applicable to s, $\pi \sqcup \{s \mapsto a\} \in \texttt{successors}(\pi)$. Note that such

state s must be in $\texttt{dom}(\pi_*)$, as it is in $\texttt{remain}(\pi_B) = \texttt{remain}(\pi)$ and $\pi_B \subseteq \pi_*$. Otherwise $s \in \texttt{remain}(\pi_*)$ and π_* would not be a solution. Let $a' = \pi_*[s]$ and let $\pi' = \pi \sqcup \{s \mapsto a'\}$. π' is in $\texttt{successors}(\pi)$. Moreover, let $\pi'_B = \pi_B \sqcup \{s \mapsto a'\} \subseteq \pi_*$. $\texttt{dom}(\pi'_B) = \texttt{dom}(\pi')$ and $\texttt{escape}(\pi'_B) = \texttt{escape}(\pi'_B)$. Therefore, $P(\pi') = \top$. Contradiction. □

Lemma 7. *If two policies π_1 and π_2 are equivalent according to domain-escape, then $P(\pi_1) = P(\pi_2)$ and they have the same domain size.*

Proof. Trivial, since P only depends on the domain and escape sets of a policy. □

Note 1. *We note that for two policies π_1 and π_2 with $\texttt{dom}(\pi_1) = \texttt{dom}(\pi_2)$:*

1. *If $\texttt{escape}(\pi_1) = \texttt{escape}(\pi_2)$, then $\Delta_\downarrow(\pi_1) = \Delta_\downarrow(\pi_2)$.*
2. *If $\texttt{escape}(\pi_1) \subseteq \texttt{escape}(\pi_2)$, then $\Delta_\downarrow(\pi_1) \leq \Delta_\downarrow(\pi_2)$.*

Trivially from the definition of $\Delta_\downarrow$ (Messa and Pereira, 2023).

Theorem 2. *AND* with $\texttt{sign}(\pi) = \langle \texttt{dom}(\pi), \texttt{escape}(\pi) \rangle$ using the concretizer is sound and complete. Moreover, it returns solutions with minimal domain size when guided by the $\Delta_\downarrow$ heuristic using $h^{\text{LM-Cut}}$.*

Proof. By Lemma 5, we know AND* without the concretizer always terminates and never returns false-positives. This does not change with the use of the concretizer, since the concretizer always terminates (Lemma 1) and since we still always check if a policy is a solution before returning it. Therefore, we only need to prove that it always returns a solution when the task is solvable to ensure its soundness and completeness. Assume, for the sake of contradiction, that, for some solvable task, it does not return a solution. Then, it must have returned precisely $\bot$ (Algorithm 4's Line 13), as it never returns false-positives. That means that every policy π that was inserted in `Queue` was removed from it. Consider the policy π with $P(\pi) = \top$ with maximal domain size that was removed from `Queue` (there is at least one since $P(\emptyset) = \top$, as $\emptyset$ is a subset of any solution). If there is more than one such policy, consider the first one that was removed. Let's look for the moment it was removed from `Queue`. By Lemma 7, it must also not have been pruned, since that would imply that there was another policy π' with $P(\pi') = \top$ and same domain size that was removed from `Queue` before π. We also know that AND* with the concretizer did not

return any solution when analyzing π, since $\bot$ was the return value. Then, each policy in $\texttt{successors}(\pi)$ was inserted in `Queue`. However, at least one $\pi' \in \texttt{successors}(\pi)$ must have $P(\pi') = \top$ (by Lemma 6). And it must have been removed from `Queue` at some point. However, such a π' has domain size greater than π's, contradicting the choice of π. This proves that AND* must return a solution when the task is solvable, and therefore it is sound and complete.

Now, let us prove the optimality. Assume, for the sake of contradiction, that there is a solution with domain size k, but it returns a suboptimal solution π'_* with $|\texttt{dom}(\pi'_*)| > k$. Let π' be the policy that AND* with the concretizer analyzed in order to return π'_*. Then, either $\pi' = \pi'_*$ or π' is a policy with $\texttt{dom}(\pi') = \texttt{dom}(\pi_*)$ and $\texttt{escape}(\pi') \supseteq \texttt{escape}(\pi_*)$. Then, we note that $\Delta_{\downarrow}(\pi') \geq |\texttt{dom}(\pi'_*)|$ due to the heuristic goal-awareness plus Note 1.2. Then, all policies π with $\Delta_{\downarrow}(\pi) \leq k < |\texttt{dom}(\pi'_*)| \leq \Delta_{\downarrow}(\pi')$ inserted in `Queue` must have been eventually removed from it, as of Algorithm 4's Line 3's best-first choices. Therefore, all policies π with $P(\pi) = \top$ inserted in `Queue` were eventually removed from it, because they must have $\Delta_{\downarrow}(\pi) \leq k$, by the heuristic admissibility, Note 1.1, and the fact that a policy π only has $P(\pi) = \top$ if it has the same domain and escape as a policy that is a subset of some optimal solution (by the definition of P). The remainder of the proof follows equal to the previous paragraph. We consider some policy π with maximal domain size that has $P(\pi) = \top$ and was once removed from `Queue`, and we reach a contradiction in the end. □

4.5.2. Empirical Evaluation

We empirically compare AND* using identity and domain-escape equivalence pruning (the latter with the concretizer). Table 2 shows the results through two subtables. The first subtable presents the ratio of tasks that were solved, i.e., the *coverage* (%S), the ratio of tasks that failed by the time limit (%T), and the ratio of tasks that failed by the memory limit (%M), for each domain and each approach. The second subtable presents the average execution time in seconds ($t(s)$), the average number of generated policies ($\#\pi$), and the average returned solution size ($|\pi_*|$) for each domain and each approach. The second subtable considers only the tasks solved by AND* using both identity and domain-escape equivalence pruning. The %∩S column shows for each domain the ratio of the tasks solved using both approaches. In both subtables, the first block of domains is the domains from the IPC-FOND benchmark, while the second block of domains is the domains from

	identity *(without the concretizer)*			domain-escape		
Domain	%S	%T	%M	%S	%T	%M
acrobatics	0.50	0	0.50	**1**	0	0
beam-walk	0.82	0.18	0	0.82	0.18	0
blocksworld-original	0.47	0.33	0.20	0.50	0.33	0.17
blocksworld-advanced	0.20	0.60	0.20	0.20	0.64	0.16
chain-of-rooms	0.30	0	0.70	0.30	0	0.70
earth-observation	0.20	0	0.80	0.20	0	0.80
elevators	0.47	0	0.53	0.47	0	0.53
faults	0.42	0	0.58	0.40	0	0.60
first-responders	0.44	0	0.56	0.48	0	0.52
tireworld-triangle	**1**	0	0	**1**	0	0
zenotravel	0.33	0.67	0	0.33	0.67	0
doors	**1**	0	0	0.87	0	0.13
islands	0.38	0	0.62	0.42	0	0.58
miner	0.10	0.02	0.88	0.10	0.02	0.88
tireworld-spiky	0.45	0	0.55	0.45	0	0.55
tireworld-truck	0.16	0	0.84	0.18	0	0.82
TOTAL	0.453	0.113	0.435	0.482	0.115	0.403

(a) Ratio of successful, timeout, and memory-out runs per configuration.

		identity *(without the concretizer)*			domain-escape		
Domain	%∩S	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$
acrobatics	0.50	0.32	39,409	14.00	0.04	197	14.00
beam-walk	0.82	0.38	454	453.22	0.22	454	453.22
blocksworld-original	0.47	82.59	408,803	12.79	81.46	226,549	12.79
blocksworld-advanced	0.20	0.41	8,444	9.64	0.39	4,552	9.64
chain-of-rooms	0.30	13.75	1,193,474	57.00	1.83	118,996	57.00
earth-observation	0.20	7.58	756,538	13.88	7.30	693,463	13.88
elevators	0.47	0.33	42,196	14.86	0.39	42,196	14.86
faults	0.40	3.16	329,743	15.36	2.28	238,657	15.36
first-responders	0.43	2.40	336,141	7.56	2.56	194,506	7.56
tireworld-triangle	**1**	27.46	485	244.00	26.26	485	244.00
zenotravel	0.33	2.33	1,808	14.80	2.33	1,808	14.80
doors	0.87	8.00	5,048	5,038.62	9.05	5,048	5,038.62
islands	0.38	1.38	271,729	4.65	1.50	271,515	4.65
miner	0.10	211.36	2,271,451	14.20	212.47	2,209,231	14.20
tireworld-spiky	0.45	45.99	1,268,072	25.00	49.41	1,268,072	25.00
tireworld-truck	0.16	1.18	282,463	13.00	1.51	268,504	13.00
TOTAL	0.442	4.18	68,857	29.94	3.20	37,209	29.94

(b) Average runtime, number of generated policies, and size of returned solutions, per configuration, over the intersection of solved tasks.

Table 2: Empirical results for AND* with and without domain-escape equivalence pruning.

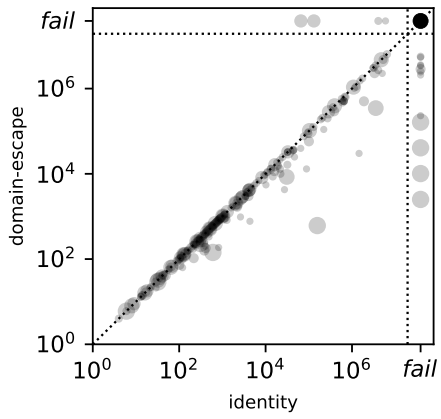

Figure 5: Per-task comparison on the number of generated policies between using domain-escape equivalence pruning or not.

the NEW-FOND benchmark. The final row aggregates the results from all domains. All domains have the same weight in the aggregation, independently of the number of (solved) tasks each has. We use arithmetic means to aggregate ratios and geometric means to aggregate other kinds of information to avoid giving more weight to hard domains.

The number of generated policies has decreased in several domains. Major decreases have occurred in five domains: *acrobatics* (99.5% reduction), *blocksworld-original* (44.6% reduction), *blocksworld-advanced* (46.1%), *chain-of-rooms* (90.0%), *faults* (27.6%) and *first-responders* (42.1%) – all from the IPC-FOND benchmark.

Figure 5 depicts the per-task differences in the number of generated policies between identity and domain-escape pruning. It has one mark per task, with proportionally bigger marks for tasks from domains with fewer tasks.

The reduction was often not enough to increase coverage. We verify that the average ratio of solved tasks (coverage) is similar for both approaches, with small changes in most domains. A major change appears only in one domain: *acrobatics* (from 0.50 to 1).

From now on, we assume that AND* always follows Algorithm 4.

4.6. Escape Equivalence Pruning

One can ask if the problem we previously showed regarding the use of `sign` = `escape` equivalence pruning still holds if we use the concretizer.

This is a very reasonable question since domain-escape equivalence pruning was not complete before and became complete with the use of the concretizer. Unfortunately, escape equivalence pruning is still sound but not complete, even if we use the concretizer. In this subsection, we show how one can cope with that, and effectively use escape equivalence pruning.

Figure 6 shows an example state space with five states: s_A – the initial state –, s_B, s_C, s_D, and s_E – the single goal state. We use it to show the incompleteness of escape equivalence pruning. The choices made by `successors` method regarding which unmapped state to map when expanding a policy determine the specific policy space AND* is searching on. We analyze a possible policy space for this state space. The initial search node is $\pi_{\mathcal{I}} = \emptyset$ and it has two successors: $\pi_1 = \{s_A \mapsto a_L\}$ and $\pi_2 = \{s_A \mapsto a_R\}$. We note that $\texttt{escape}(\pi_1) = \{s_B, s_D\}$ and $\texttt{escape}(\pi_2) = \{s_C, s_D\}$. Let's say AND* chooses to map s_B instead of s_D for π_1 and s_C instead of s_D for π_2. Then, the single successor of π_1 is $\pi_3 = \{s_A \mapsto a_L, s_B \mapsto b\}$ and the single successor of π_2 is $\pi_4 = \{s_A \mapsto a_R, s_C \mapsto c\}$. We note that $\texttt{escape}(\pi_3) = \{s_C, s_D\}$ and $\texttt{escape}(\pi_4) = \{s_B, s_D\}$. Both π_3 and π_4 are two steps away from becoming a solution policy. The critical problem is that both π_3 and π_4 will be pruned because $\pi_1 \sim \pi_4$ and $\pi_2 \sim \pi_3$ if AND* chooses to expand π_1 and π_2 before expanding either π_3 or π_4. This kind of "crossed-mutual" pruning, where the descendant of some policy is pruned by the ancestor of another policy and vice-versa, cannot happen when using previous equivalence pruning concepts because $\pi \sim \pi'$ implies/requires $|\texttt{dom}(\pi)| = |\texttt{dom}(\pi')|$ when `sign` is identity or domain-escape.[9]

Despite being incomplete, AND* always terminates and never returns false-positive answers when using escape equivalence pruning (Lemma 5). So, AND* always returns either $\perp$ or a solution. Never a non-solution policy π. That is the case because, by construction, it has no means of returning a policy that is not a solution, regardless of the choice of `sign`. We can use this fact to make AND* with escape equivalence pruning sound and complete in a straightforward way. If we run first an incomplete but sound algorithm with termination guarantees, and then – only if it fails – run a sound and complete algorithm, we create a hybrid algorithm that has the desired guarantees. This approach was taken by Hoffmann and Nebel (2001) to create the FF

[9] We note that even if we prevent "crossed-mutual" prunings from occurring, there are more complex examples where escape equivalence pruning is still incomplete.

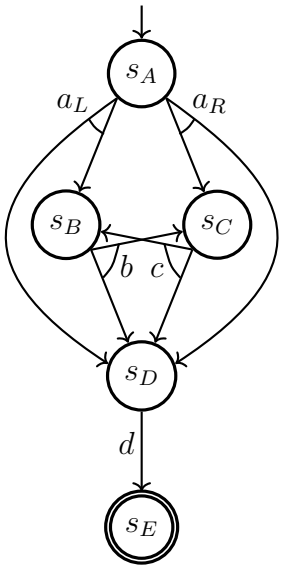

Figure 6: Example state space to show the incompleteness of AND* with escape equivalence pruning, even with the concretizer.

planning system. We follow the same approach. We set AND* with escape equivalence pruning as the first phase algorithm, and with domain-escape equivalence pruning as the second phase algorithm. *Optimality is lost here.*

Later in this work, we present other approaches that improve on AND* with escape equivalence pruning. These other approaches are also incomplete, so we always use them in the same way, making them complete by setting up a fallback to AND* with domain-escape equivalence pruning. Nevertheless, this second phase was never needed in any of our experiments. The hybrid algorithms either failed due to resource limits during the first phase or found a solution through the first phase.

We empirically compare AND* using escape equivalence pruning againt AND* using just domain-escape equivalence pruning (Table 3). We can check that there was an expressive increase in coverage in comparison to domain-escape equivalence pruning (from 0.482 to 0.651). Major gains have occurred in *chain-of-rooms* (from 0.30 to 1), *elevators* (from 0.47 to 0.80), *first-responders* (from 0.48 to 0.69), *islands* (from 0.42 to 0.63), *tireworld-spiky* (from 0.45 to 0.82), and *tireworld-truck* (from 0.18 to 0.46). There is a dramatic improvement in the number of generated policies in several domains. Decreases of at least 50% have occurred in *acrobatics*, *blocksworld-*

	domain-escape			escape		
Domain	%S	%T	%M	%S	%T	%M
acrobatics	**1**	0	0	**1**	0	0
beam-walk	0.82	0.18	0	0.82	0.18	0
blocksworld-original	0.50	0.33	0.17	0.53	0.47	0
blocksworld-advanced	0.20	0.64	0.16	0.20	0.71	0.09
chain-of-rooms	0.30	0	0.70	**1**	0	0
earth-observation	0.20	0	0.80	0.33	0.15	0.53
elevators	0.47	0	0.53	0.80	0	0.20
faults	0.40	0	0.60	0.56	0	0.44
first-responders	0.48	0	0.52	0.69	0.25	0.05
tireworld-triangle	**1**	0	0	**1**	0	0
zenotravel	0.33	0.67	0	0.33	0.67	0
doors	0.87	0	0.13	**1**	0	0
islands	0.42	0	0.58	0.63	0	0.37
miner	0.10	0.02	0.88	0.24	0.76	0
tireworld-spiky	0.45	0	0.55	0.82	0.18	0
tireworld-truck	0.18	0	0.82	0.46	0	0.54
TOTAL	0.482	0.115	0.403	0.651	0.211	0.138

(a) Ratio of successful, timeout, and memory-out runs per configuration.

		domain-escape			escape		
Domain	%∩S	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$
acrobatics	**1**	3.10	27,206	126.50	2.08	11,109	126.50
beam-walk	0.82	0.22	454	453.22	0.21	454	453.22
blocksworld-original	0.50	121.56	402,523	13.40	152.25	182,954	13.47
blocksworld-advanced	0.20	0.39	4,552	9.64	0.37	1,729	9.64
chain-of-rooms	0.30	1.83	118,996	57.00	0.40	828	57.00
earth-observation	0.20	7.30	693,463	13.88	0.08	3,725	14.00
elevators	0.47	0.39	42,196	14.86	0.07	383	14.86
faults	0.40	2.28	238,657	15.36	0.73	79,559	16.27
first-responders	0.48	4.96	487,417	8.03	8.47	18,001	8.08
tireworld-triangle	**1**	26.26	485	244.00	25.72	485	244.00
zenotravel	0.33	2.33	1,808	14.80	2.33	934	14.80
doors	0.87	9.05	5,048	5,038.62	7.86	5,048	5,038.62
islands	0.42	2.96	575,809	4.84	0.67	17,057	4.84
miner	0.10	212.47	2,209,231	14.20	214.63	166,141	14.20
tireworld-spiky	0.45	49.41	1,268,072	25.00	37.97	138,650	25.00
tireworld-truck	0.18	3.20	511,915	13.15	0.15	2,568	13.15
TOTAL	0.482	4.89	60,651	34.70	2.11	6,305	34.87

(b) Average runtime, number of generated policies, and size of returned solutions, per configuration, over the intersection of solved tasks.

Table 3: Empirical results for AND* using escape and domain-escape equivalence pruning.

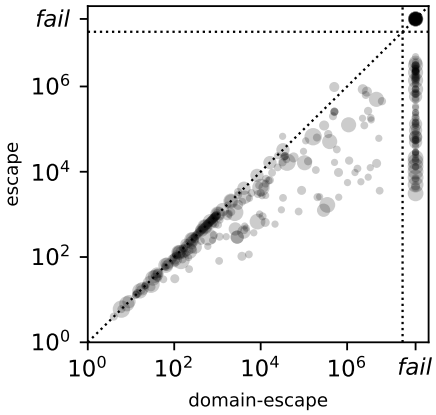

Figure 7: Per-task comparison on the number of generated policies between using domain-escape and escape equivalence pruning.

original, *blocksworld-advanced*, *chain-of-rooms*, *earth-observation*, *elevators*, *faults*, *first-responders*, *zenotravel*, *islands*, *miner*, *tireworld-spiky*, and *tireworld-truck* – all but three domains. Decreases of at least one order of magnitude have occurred in seven domains: *chain-of-rooms*, *earth-observation*, *elevators*, *first-responders*, *islands*, *miner*, and *tireworld-truck*. Figure 7 depicts, per task, the differences.

5. Improving Equivalence Pruning with State Symmetries

In this section, we focus on applying the knowledge of *states* that are equivalent to each other, to broaden the definition of policies that are equivalent to each other, consequently strengthening the policy equivalence pruning.

Two states are considered symmetrically equivalent to each other if they participate on "symmetrical" trajectories in the state space. For instance, in our example from Figure 6, s_B and s_C are symmetrically equivalent (or simply symmetric) to each other. We can use this knowledge to define a new signature function for policies.

Let `sign2` be a function with $\texttt{dom}(\texttt{sign2}) = \mathcal{S}$ that maps symmetric states to the same signature. Each signature defines an equivalence class of states. Using this `sign2` function, we can define a new escape equivalence definition

as $\texttt{sign}(\pi) = \{\!\{\texttt{sign2}(s) : s \in \texttt{escape}(\pi)\}\!\}$[10]. With this new definition, two policies are considered equivalent to each other iff they contain in their `escape` the same number of states from each equivalence class of symmetric states. However, two questions arise. How can we determine that two states are symmetric to each other? And what happens when we use this new definition of equivalence pruning with AND*?

Structural state symmetries (Pochter et al., 2011; Shleyfman et al., 2015) is a well-defined concept of state symmetries that uses the structure of the planning task to determine which states are symmetric to each other. Structural state symmetries capture part of all existing state symmetries, and Winterer et al. (2016) shows we can apply the concept of structural state symmetries for FOND planning. In this work, we use structural state symmetries to define the `sign2` function.

Regarding the second question, we evaluate the empirical impact of using the new definition of equivalence pruning with AND*, using two different techniques to find structural state symmetries: an approximated (greedy) approach and the perfect approach. The greedy approach comes from the literature. Pochter et al. (2011) and Winterer et al. (2016) compute structural state symmetries using a greedy hill-climbing procedure, respectively in the context of classical and FOND planning. The justification given for using an approximated approach is that computing all structural state symmetries (the perfect approach) is *intractable* (Winterer et al., 2016).

We show that we can *in practice* efficiently perfectly compute structural state symmetries for FOND planning tasks using group theory techniques from Jefferson et al. (2019), when using a slightly more restrictive definition of structural state symmetries.

5.1. Structural State Symmetries

Let $\Pi = \langle \mathcal{F}, \mathcal{I}, \gamma, \mathcal{A} \rangle$ be a FOND planning task. A structural symmetry for Π is[11] a permutation $\sigma : \mathcal{F} \sqcup \mathcal{A} \to \mathcal{F} \sqcup \mathcal{A}$, such that:

[10] The double curly brackets notation denotes a multiset: i.e., a collection of elements, that unlike a set, allows repeated elements, but like a set, does not have an order.

[11] We combine the definitions from Shleyfman et al. (2015) and Winterer et al. (2016). The former defines structural symmetries for classical planning tasks in STRIPS (Fikes and Nilsson, 1971), while the latter defines them for FOND planning tasks in SAS+ (Bäckström and Nebel, 1995). Since our work's running definition of FOND planning tasks uses the STRIPS formalism, our formal definition of structural symmetries is an

1. $\sigma = \sigma_{\mathcal{F}} \sqcup \sigma_{\mathcal{A}}$, where $\sigma_{\mathcal{F}} : \mathcal{F} \to \mathcal{F}$ is a permutation of facts and $\sigma_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ is a permutation of actions;
2. $\sigma(f) \in \gamma \Leftrightarrow f \in \gamma$; and
3. $\{\sigma(f) : f \in \texttt{pre}(a)\} = \texttt{pre}(\sigma(a))$ and $\{\{\sigma(f) : f \in e\} : e \in \texttt{effs}(a)\} = \{e : e \in \texttt{effs}(\sigma(a))\}$, for each $a \in \mathcal{A}$.

We overload σ, so that, for each state $s \in \mathcal{S}$, $\sigma(s)$ denotes the state $s' \in \mathcal{S}$ with $\mathcal{F}[s'] = \{\sigma(f) : f \in \mathcal{F}[s]\}$. We note that if σ is a structural symmetry, $\sigma(s)$ is a goal state if and only if s is a goal state. Moreover, an action $\sigma(a)$ is applicable to $\sigma(s)$ if and only if the action a is applicable to s. Finally, the set $\{s' : s' \in \texttt{succs}(\sigma(s), \sigma(a))\}$ is equal to the set $\{\sigma(s') : s' \in \texttt{succs}(s, a)\}$.

The definition of structural symmetries we use in this work is more restrictive. It has two extra conditions:

4. $P_{\mathcal{F}}(\sigma(f)) = P_{\mathcal{F}}(f)$, for each $f \in \mathcal{F}$, where $P_{\mathcal{F}} : \mathcal{F} \to \mathbb{Z}$ is a *given* partition of the facts; and
5. $P_{\mathcal{A}}(\sigma(a)) = P_{\mathcal{A}}(f)$, for each $a \in \mathcal{A}$, where $P_{\mathcal{A}} : \mathcal{A} \to \mathbb{Z}$ is a *given* partition of the actions.

The partition $P_{\mathcal{F}}$ comes from the fact we use a SAS+ (Bäckström and Nebel, 1995) formalism to represent the FOND planning tasks in our implementation. When using SAS+, facts are partitioned into variables such that, for each variable, exactly one fact of that variable is true at each state. This allows a more compact representation of actions and states (Helmert, 2009). With the first extra restriction, our definition of structural symmetries matches the definition from Winterer et al. (2016).

The second extra restriction is an optimization. We look only for structural symmetries that permute actions that were grounded from the same lifted action. This potentially loses symmetries between distinct lifted actions but allows a more focused computation of structural symmetries.

5.2. Computing Structural Symmetries

Let $Aut(\Pi)$ be the *permutation group* containing all structural symmetries for Π. The ideal objective would be to have a *canonical* `sign2` function: i.e.,

adaptation of theirs. Finally, since we use SAS+ in our implementation, we also talk about it.

a $\texttt{sign2}$ function such that $\texttt{sign2}(s) = \texttt{sign2}(s')$ iff there exists a structural symmetry $\sigma \in Aut(\Pi)$ with $s' = \sigma(s)$[12].

Let $Orb(s) = \{\sigma(s) : \sigma \in Aut(\Pi)\}$ be the set of states symmetric to a given state s. We could set $\texttt{sign2}(s) = Orb(s)$ for each state s, and obtain a canonical $\texttt{sign2}$ function. Moreover, if we want the $\texttt{sign2}$ function to have smaller signature objects, and map states to states instead of mapping states to sets of states, we could choose an arbitrary relation of total order $\prec$ over states, and then we could instead set $\texttt{sign2}(s) = \min Orb(s)$ w.r.t. $\prec$. The issue is that $Aut(\Pi)$ is usually too large even to be enumerated. In the worst case, the asymptotic size of $Aut(\Pi)$ is factorial to the size of facts and actions. Therefore, this direct approach cannot be used in practice. Nevertheless, the idea of finding the minimal symmetric state is still used in the techniques we present next.

5.2.1. Approximating a Canonical Function

Pochter et al. (2011) and Shleyfman et al. (2015) show we can compute a *generator* for $Aut(\Pi)$ using a *Problem Description Graph* (PDG) of the task. A generator of a permutation group $\mathcal{G}$ is a set of permutations g, such that for every permutation $\sigma \in \mathcal{G}$, there exists some sequence $\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ of permutations in g, possibly with repetitions, such that the result of their composition is σ. In other words, a generator of a permutation group $\mathcal{G}$ is a compact representation of $\mathcal{G}$.

Winterer et al. (2016) *approximate* the desired canonical function using a greedy hill-climbing procedure w.r.t. $\prec$ on the local state neighborhood $\mathcal{N}$ provided by the direct application of generator permutations – i.e. $\mathcal{N}(s) = \{\sigma(s) : \sigma \in g\}$, where g is the computed generator of $Aut(\Pi)$. Following their approach, we could define $\texttt{sign2}(s)$ to be the final state resulting from the hill-climbing procedure starting from s. If $s \sim s'$ for the $\texttt{sign2}$ defined from this greedy hill-climbing procedure, then $s \sim s'$ also holds for a canonical $\texttt{sign2}$, but the opposite is not always true.

We note that after we compute the generator g of $Aut(\Pi)$, we could, for the purposes of this work, ignore the actions part of each $\sigma \in g$. This is because the symmetries between actions are not necessary for the computation of $\texttt{sign2}$. Knowledge about the actions is only necessary at the first

[12]Note that if σ is a structural symmetry for a task Π, then σ^{-1} also is. So if exists a structural symmetry $\sigma \in Aut(\Pi)$ with $s' = \sigma(s)$, then exists a structural symmetry $\sigma' = \sigma^{-1} \in Aut(\Pi)$ with $s = \sigma'(s')$.

step, to ensure that $Aut(\Pi)$ contains only correct symmetry relations for the task, which requires associating the facts with the actions. In other words, once we have already computed the generator g of $Aut(\Pi)$, we can create a new generator $g' = \{\sigma_{\mathcal{F}} : \sigma \in g\}$, which induces a new $Aut(\Pi)$, simplified, that acts only on the facts, but retains the same symmetries between states. We make this remark because the definitions we use for presenting the next technique mostly assume that the permutation group acts only on the facts.

5.2.2. Perfectly Computing a Canonical Function

In this work, we apply a technique presented by Jefferson et al. (2019) that allows the computation of the canonical signature of a state by using a (compactly represented) *stabilizer chain* of $Aut(\Pi)$.

A fact f is said to be *point-wise stable* in a permutation group $\mathcal{G} \leq Aut(\Pi)$, where $\leq$ denotes that $\mathcal{G}$ is a *subgroup* of $Aut(\Pi)$, iff, for every structural symmetry $\sigma \in \mathcal{G}$, $\sigma(f) = f$.

A stabilizer chain of $Aut(\Pi)$ is a sequence $\langle \mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n \rangle$ of subgroups of $Aut(\Pi)$, such that:

1. $\mathcal{G}_0 = Aut(\Pi)$;
2. $\mathcal{G}_n$ is a group that contains only the identity structural symmetry (i.e., the structural symmetry σ that maps every fact to itself and every action to itself); and
3. for each $i \in \{1, 2, \ldots, n\}$:
   (a) $\mathcal{G}_i \leq \mathcal{G}_{i-1}$;
   (b) the set of facts that are point-wise stable in $\mathcal{G}_i$ is a strict superset of the set of facts that are point-wise stable in $\mathcal{G}_{i-1}$; and
   (c) there is no subgroup $\mathcal{G}'$ of $Aut(\Pi)$ such that $\mathcal{G}_i \leq \mathcal{G}' \leq \mathcal{G}_{i-1}$ that still stabilizes more facts than $\mathcal{G}_{i-1}$.

For every possible stabilizer chain of $Aut(\Pi)$, there is a total ordering of facts that compactly represents it unambiguously. Conversely, every total ordering of facts compactly represents some stabilizer chain of $Aut(\Pi)$. These total orderings of facts are called *bases*. Let $\mathcal{B} = \langle f_1, f_2, \ldots, f_{|\mathcal{F}|} \rangle$ be a base. The stabilizer chain that $\mathcal{B}$ represents is the $\langle \mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n \rangle$ stabilizer chain where, for each $i \in \{1, 2, \ldots, n\}$, $\mathcal{G}_i$ is the maximal subgroup of $\mathcal{G}_{i-1}$ that stabilizes the first fact of $\mathcal{B}$ that is not stabilized by $\mathcal{G}_{i-1}$.

Let $\mathcal{G}$ and $\mathcal{G}'$ be two permutation groups that appear consecutively in a stabilizer chain of $Aut(\Pi)$ and let f be the first fact in the base of the stabilizer

chain that is stabilized by $\mathcal{G}'$ but not by $\mathcal{G}$. The *cosets* of $\mathcal{G}'$ in $\mathcal{G}$ are the result of the partition of the permutation group in $\mathcal{G}$ into the unique *minimal set* of permutation groups $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m\}$, such that, for each coset $\mathcal{G}_i$, every pair of permutations $\sigma_1, \sigma_2 \in \mathcal{G}_i$ satisfies $\sigma_1(f) = \sigma_2(f)$. A *transversal* is a set of permutations $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$, with exactly one permutation from each coset. An *inverse transversal*, on the other hand, is a set of permutations $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$, such that $\{\sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_m^{-1}\}$ is a transversal.

Jefferson et al. (2019) show that if we have a generator g of a permutation group $\mathcal{G}$, a base $\mathcal{B}$ that represents a stabilizer chain $\langle \mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n \rangle$ of $\mathcal{G}$, a set of elements $\langle e_1, e_2, \ldots, e_n \rangle$ such that each e_i is the first element of $\mathcal{B}$ that $\mathcal{G}_i$ stabilizes but $\mathcal{G}_{i-1}$ does not, and a set of inverse transversals $\langle \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_n \rangle$ such that each $\mathcal{I}_i$ is the inverse transversal of $\mathcal{G}_i$ in $\mathcal{G}_{i-1}$, then we can use that to compute a *canonical image*[13] of any subset of elements w.r.t. $\mathcal{G}$. Applied to our context, if $\mathcal{G} = Aut(\Pi)$, we can compute the canonical signature of any given state, by looking at the canonical image of the set of facts that are true in that state.

Algorithm 5 shows the pseudocode of Jefferson et al. (2019)'s technique[14] applied to our context, where elements are facts, and subsets of elements are states. As far as we know, this is the first time their work is being applied to planning.

We show next that with their technique, we are able to compute a canonical signature of states almost as fast as we can compute a greedy signature using the hill-climbing procedure.

5.3. Empirical Evaluation

Table 4 shows, for each domain, the ratio of tasks with structural symmetries (besides the identity symmetry). For eight of the 16 domains, at least 80% of tasks in each domain present structural symmetries. And for ten, at least half the tasks present structural symmetries. We use Nauty (McKay and

[13] A canonical image of a subset of elements is a representative chosen so that two subsets have the same canonical image if and only if one can be transformed into the other by permutations from the group.

[14] Throughout their work, they present more than one technique to compute the canonical image of a subset of elements w.r.t. to a permutation group that acts on those elements. We follow their simplest presented technique (`FixedOrbit`). The impact of using the other presented techniques, which vary $\mathcal{B}$ throughout different runs, is an interesting question for future work.

Algorithm 5: Adapatation from one of Jefferson et al. (2019)'s algorithms to the context of FOND planning.

Task Constants: g, $\mathcal{B}$, $\langle f_1, f_2, \ldots, f_n \rangle$, $\langle \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_n \rangle$
Input: s `// a state`
SymmetricStates $\coloneqq \{s\}$
$M \coloneqq \emptyset$
for each $f \in \mathcal{B}$ **while** $|M| < |\mathcal{F}[s]|$ **:**
    if $\exists f_i \in \langle f_1, f_2, \ldots, f_n \rangle$ **such that** $f_i = f$ **:**
        NewSymmetricStates $\coloneqq \emptyset$
        for each $\sigma \in \mathcal{I}_i$ **:**
            for each $s' \in$ SymmetricStates **:**
                $X \coloneqq \{\sigma(f') : f' \in \mathcal{F}[s']\}$
                Let s'' be the state with $\mathcal{F}[s''] = X$.
                NewSymmetricStates $\coloneqq$ NewSymmetricStates $\cup \{s''\}$
        SymmetricStates $\coloneqq$ NewSymmetricStates
    SSWithF $\coloneqq \{s' : s' \in \text{SymmetricStates} \wedge f \in \mathcal{F}[s']\}$
    SSWithoutF $\coloneqq \{s' : s' \in \text{SymmetricStates} \wedge f \notin \mathcal{F}[s']\}$
    if SSWithF $\neq \emptyset$ **and** SSWithoutF $\neq \emptyset$ **:**
        SymmetricStates $\coloneqq$ SSWithF `// This was an insight from Jefferson et al. (2019). We can this way safely prune part of the symmetry search space, without losing the canonical image of the input, in case we are following the inverse transversals of a stabilizer chain.`
    if SSWithF $\neq \emptyset$ **:**
        $M \coloneqq M \cup \{f\}$
Let s' be the state with $\mathcal{F}[s'] = M$.
return s' `// Jefferson et al. (2019) prove that` M `is the minimal symmetric subset (minimal image) of the input subset (in this case,` $\mathcal{F}[s]$`) w.r.t the group represented by` g `and the ordering induced by` $\mathcal{B}$`, when the algorithm terminates. These minimal images become canonical representatives if` $\mathcal{B}$ `is fixed in all runs.`

	Domain	% tasks w/ sym.
IPC-FOND	acrobatics	none
	beam-walk	none
	chain-of-rooms	none
	tireworld-triangle	none
	earth-observation	just one
	blocksworld-original	50%
	blocksworld-advanced	55%
	elevators	80%
	faults	all but one
	first-responders	all but one
	zenotravel	all
NEW-FOND	doors	none
	tireworld-truck	89%
	tireworld-spiky	all but one
	islands	all but one
	miner	all

Table 4: Ratio of tasks with non-trivial symmetries for each domain in the benchmark.

Piperno, 2014) to compute automorphism group generators of PDGs, from which we can extract $Aut(\Pi)$ generators. We are able to compute a generator of $Aut(\Pi)$ in at most 1.75 seconds for all tasks that have symmetries. We set an overcautious time limit of five seconds for this computation. If Nauty fails to conclude its execution within this time limit, the use of structural symmetries is disabled. It only fails for tasks *p7* and *p8* of *acrobatics*, tasks *p6* to *p11* of *beam-walk*, and tasks *p11* to *p40* of *tireworld-triangle* – all of which likely do not have structural symmetries because, in these domains, the computation does not find symmetries for all tasks where it concludes. We use GAP (*Groups, Algorithms, and Programming* software) to compute the inverse transversals for a permutation group, given an arbtirary base and a generator. We also use GAP to order the facts by their *orbit sizes*[15] in $Aut(\Pi)$, descendingly, to obtain the base. For every task where computing $Aut(\Pi)$ within time limits was successful, we are able to compute the stabilizer chain of $Aut(\Pi)$ in at most five extra seconds.

We compare using greedy and canonical structural symmetries against not using state symmetries (Table 5). The results demonstrate that Jefferson et al. (2019)'s contributions allow a fast computation of canonical structural

[15]I.e., the number of different facts they can be mapped to by a symmetry $\sigma \in Aut(\Pi)$. This ordering was one of the suggestions made by Jefferson et al. (2019).

	none			greedy			canonical		
Domain	%S	%T	%M	%S	%T	%M	%S	%T	%M
acrobatics	**1**	0	0	**1**	0	0	**1**	0	0
beam-walk	0.82	0.18	0	0.82	0.18	0	0.82	0.18	0
blocksworld-original	0.53	0.47	0	0.53	0.47	0	0.53	0.47	0
blocksworld-advanced	0.20	0.71	0.09	0.20	0.71	0.09	0.20	0.71	0.09
chain-of-rooms	**1**	0	0	**1**	0	0	**1**	0	0
earth-observation	0.33	0.15	0.53	0.33	0.15	0.53	0.33	0.15	0.53
elevators	0.80	0	0.20	0.80	0	0.20	0.80	0	0.20
faults	0.56	0	0.44	**1**	0	0	**1**	0	0
first-responders	0.69	0.25	0.05	0.73	0.25	0.01	0.75	0.24	0.01
tireworld-triangle	**1**	0	0	**1**	0	0	**1**	0	0
zenotravel	0.33	0.67	0	0.33	0.67	0	0.33	0.67	0
doors	**1**	0	0	**1**	0	0	**1**	0	0
islands	0.63	0	0.37	0.80	0.13	0.07	0.80	0.13	0.07
miner	0.24	0.76	0	0.55	0.45	0	0.53	0.47	0
tireworld-spiky	0.82	0.18	0	**1**	0	0	**1**	0	0
tireworld-truck	0.46	0	0.54	0.57	0	0.43	0.57	0	0.43
TOTAL	0.651	0.211	0.138	0.729	0.188	0.083	0.728	0.189	0.083

(a) Ratio of successful, timeout, and memory-out runs per configuration.

		none			greedy			canonical		
Domain	%∩S	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$
acrobatics	**1**	2.08	11,109	126.50	3.36	11,109	126.50	3.36	11,109	126.50
beam-walk	0.82	0.21	454	453.22	2.45	454	453.22	2.45	454	453.22
blocksworld-original	0.53	229.56	270,882	14.12	229.84	270,709	14.12	229.86	270,708	14.12
blocksworld-advanced	0.20	0.37	1,729	9.64	0.37	1,730	9.64	0.49	1,730	9.64
chain-of-rooms	**1**	33.01	7,968	162.00	33.75	7,968	162.00	33.29	7,968	162.00
earth-observation	0.32	2.72	219,401	26.08	2.83	219,178	26.08	2.74	219,172	26.08
elevators	0.80	8.87	374,049	25.33	5.49	161,881	25.33	6.56	161,881	25.33
faults	0.56	4.80	534,692	19.13	0.06	273	19.13	0.73	273	19.13
first-responders	0.69	84.72	244,376	10.63	59.93	172,288	10.71	60.78	171,978	10.71
tireworld-triangle	**1**	25.72	485	244.00	29.46	485	244.00	30.31	485	244.00
zenotravel	0.33	2.33	934	14.80	1.93	689	14.80	2.72	689	14.80
doors	**1**	118.06	17,484	17,473.73	114.54	17,484	17,473.73	117.27	17,484	17,473.73
islands	0.63	22.38	373,141	5.74	3.23	21,403	5.74	8.66	19,214	5.74
miner	0.24	623.20	978,423	15.33	138.27	153,427	15.33	147.98	153,427	15.33
tireworld-spiky	0.82	380.18	918,175	25.00	1.70	3,798	25.00	2.41	3,798	25.00
tireworld-truck	0.46	27.17	488,319	13.94	0.27	4,454	13.94	0.80	4,454	13.94
TOTAL	0.651	15.60	44,784	45.43	5.81	10,012	45.45	8.38	9,944	45.45

(b) Average runtime, number of generated policies, and size of returned solutions, per configuration, over the intersection of solved tasks.

Table 5: Empirical results for AND* using canonical and greedy structural symmetries, and not using them.

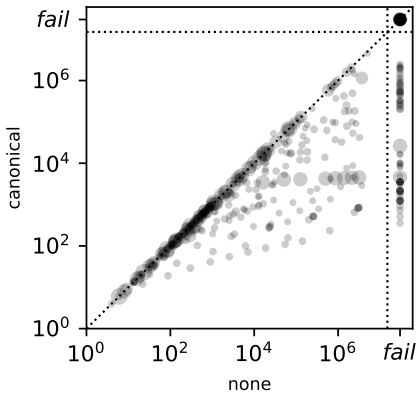

Figure 8: Per-task comparison on the number of generated policies between using canonical structural state symmetries and not using state symmetries.

symmetries. Computing the canonical structural symmetries takes only 45% more (aggregated) time per generated policy than computing the structural symmetries through hill-climbing for the solved tasks. On the other hand, the greedy structural symmetries provided by the hill-climbing procedure prove effective, as only 0.69% more policy generations are required to solve the solved tasks on average, a difference that comes almost exclusively from the domain of *islands*.

Both approaches improve AND*'s performance to a coverage near of 0.73, much superior to the coverage of 0.651 we had before using state symmetries. The number of generated policies reduces by at least 25% when using state symmetries on the domains of *elevators*, *faults*, *zenotravel*, *islands*, *miner*, *tireworld-spiky*, and *tireworld-truck*. The reduction is astonishing in *faults* (-99.95%), *tireworld-spiky* (-99.59%), and *tireworld-truck* (-99.09%). Figure 8 depicts the per-task differences in the number of generated policies between using canonical symmetries and not using structural symmetries. The gains in coverage occur in *faults*, *first-responders*, *islands*, *miner*, *tireworld-spiky*, and *tireworld-truck*. The major gains are in *faults* (from 0.56 to 1) and *miner* (from 0.24 to 0.55 and 0.53). No change occurs in domains without structural symmetries besides the increase in execution time due to the execution of Nauty. The returned solution size remains, in general, mostly unchanged. We decide to stick with canonical structural state symmetries for the remainder of this work.

6. Solution Compression

In this section, we introduce a procedure, called *the compressor*, that allows the compression of solution policies. The compressor receives as input a policy π defined over states, and returns a policy τ defined over partial states that is *optimal* (has the minimum number of partial states), such that τ maps each state in the $\texttt{dom}(\pi)$ to the same action it is mapped in π, and τ maps no state in the $\texttt{escape}(\pi)$. The central idea of the compressor is to use an integer program iteratively, which is fast in practice, to determine the partial states part of policy τ. The compressor is a general procedure and can compress solutions of any planner that finds policies defined over states.

6.1. Partial States and Partial Policies

A *partial state* is a *partial* function $p : \mathcal{F} \rightharpoonup \{\text{True}, \text{False}\}$. Partial states represent simple properties regarding the truth valuation of facts, allowing the concise representation of the set of states with such a property. We note that we can view a state s as a total function $s : \mathcal{F} \to \{\text{True}, \text{False}\}$ with $s[f] = \text{True}$ iff $f \in \mathcal{F}[s]$. We say a state s *holds* a property (partial state) p – denoted $s \models p$ – iff, for each $f \in \mathcal{F}$, $p[f]$ is undefined or $s[f] = p[f]$. The set of states represented by a partial state p is $\mathcal{S}[p] = \{s : s \in \mathcal{S} \ \wedge \ s \models p\}$. We can perceive the task goal condition and the action preconditions as partial states. A *partial policy* τ is a partial function mapping partial states to actions. For a given state s, we define $\tau[s] = \{\tau[p] : p \in \texttt{dom}(\tau) \wedge s \models p\}$. Note that $\tau[s]$ is a set of actions. We define $\texttt{decompress}(\tau) = \{s \mapsto a : s \in \mathcal{S} \ \wedge \ \tau[s] = \{a\}\}$. $\texttt{decompress}(\tau)$ is a partial function mapping states to actions, and it maps a state s iff $\tau[s]$ is a singleton. Therefore, there are two distinct ways for a state s to become unmapped on $\texttt{decompress}(\tau)$. Namely, if either $\tau[s] = \emptyset$ or $|\tau[s]| \geq 2$. We use $\texttt{ambg}(\tau) = \{s : s \in \mathcal{S} \ \wedge \ |\tau[s]| \geq 2\}$ to contain each state that becomes unmapped on $\texttt{decompress}(\tau)$ because there are multiple partial states in τ representing it, and they do not agree on the action. Let $\texttt{pruned}(\pi) = \pi|_{\texttt{reach}(\pi, s_\mathcal{I})}$. A partial policy τ_* is a *solution partial policy* for Π iff $\texttt{pruned}(\texttt{decompress}(\tau_*))$ is a solution policy for Π and $\texttt{reach}(\texttt{pruned}(\texttt{decompress}(\tau_*))) \cap \texttt{ambg}(\tau_*) = \emptyset$.[16]

[16] We ensure no state s with $|\tau_*[s]| \geq 2$ is reachable because there are multiple ways to deal with such states. Setting $\texttt{decompress}(\tau_*)[s] = \bot$ is just one possibility. Therefore, if we were to let $\texttt{reach}(\texttt{pruned}(\texttt{decompress}(\tau_*))) \cap \texttt{ambg}(\tau_*) \neq \emptyset$, one would need to be

6.2. The Compressor

Algorithm 6 displays the pseudocode for the compressor. The compressor allows the compression of any given π into a partial policy τ such that $\texttt{decompress}(\tau)|_{\texttt{reach}(\pi)} = \pi$ and $\texttt{reach}(\pi) \cap \texttt{ambg}(\tau) = \emptyset$, with minimal $|\texttt{dom}(\tau)|$. In other words, it produces a partial policy τ such that $\texttt{decompress}(\tau)$ agrees with π for each $s \in \texttt{dom}(\pi)$ and does not map any state $s \in \texttt{escape}(\pi)$. Additionally, it ensures $\tau[s] = \emptyset$ for each state $s \in \texttt{escape}(\pi)$.

Algorithm 6: Policy Compressor

Input: π
$\tau \coloneqq \emptyset$
for each $a \in \mathcal{A}$ **:**
  $X \coloneqq \{s : s \in \texttt{dom}(\pi) \ \wedge \ \pi[s] = a\}$
  $Y \coloneqq \{s : s \in \texttt{dom}(\pi) \ \wedge \ \pi[s] \neq a\} \sqcup \texttt{escape}(\pi)$
  Find some minimal-sized set Z of partial states such that
   $\forall s \in X : \exists p \in Z, \ (s \models p)$ and $\forall s \in Y : \neg\exists p \in Z, \ (s \models p)$
  Set $\tau[p] \coloneqq a$ for each $p \in Z$
return τ

In order for π to agree with $\texttt{decompress}(\tau)$ in the mapping of some state $s \in \texttt{dom}(\pi)$, $\tau[s]$ must be $\{\pi[s]\}$. Therefore, there must exist at least one partial state p with $s \models p$ mapped to $\pi[s]$ in τ (otherwise $\pi[s] \notin \tau[s]$), and there must exist no other partial state p' with $s \models p'$ mapped to other action besides $\pi[s]$ in τ (otherwise $\tau[s] \not\subseteq \{\pi[s]\}$). That is enough to ensure an agreement of π and $\texttt{decompress}(\tau)$ regarding states in $\texttt{dom}(\pi)$. We additionally enforce that no partial state in τ represents any state in $\texttt{escape}(\pi)$.

The compressor subdivides the original problem of deciding all the mappings of τ into independent subproblems of deciding all the mappings to each action. For each action, it finds the minimal set of partial states that covers all states that need to be mapped to that action without covering any other domain or escape state from π. The resulting partial policy τ has $\texttt{decompress}(\tau)|_{\texttt{reach}(\pi)} = \pi$ since $\texttt{decompress}(\tau)$ agrees with π for all states

informed about how to proceed at $\texttt{ambg}(\tau_*)$ states in order to use τ_* properly. Some works set an external rule that defines which action should be used at s if $|\tau_*[s]| \geq 2$. For instance, by presenting a total order for the mappings in τ_*.

in $\texttt{dom}(\pi)$ and has $\texttt{decompress}(\tau) = \bot$ for all states in $\texttt{escape}(\pi)$. Since we ensure $\tau[s] = \emptyset$ for each $s \in \texttt{escape}(\pi)$, we have $\texttt{reach}(\pi) \cap \texttt{ambg}(\tau) = \emptyset$. Therefore, the resulting partial policy τ is a solution partial policy when the input policy π is a solution policy.

6.3. Finding the Minimal-Sized Set of Partial States Z

To find the minimal set of partial states Z that fulfills the required conditions of representing each $s \in X$ but no $s \in Y$, we repeatedly try to solve an integer programming problem with an increasing number of partial states, starting from one.

The following integer program models the task of determining whether there is a set of k partial states representing each $s \in X$ but no $s \in Y$, or not.

$$\text{minimize} \quad \sum_{i=1}^{k} \sum_{f \in \mathcal{F}} \sum_{b \in \mathbb{B}} p_i[f \mapsto b]$$

$$\begin{aligned}
\text{subject to} \quad \sum_{i=1}^{k} p_i[s] &\geq 1 && \forall s \in X && (1)\\
\sum_{i=1}^{k} p_i[s] &= 0 && \forall s \in Y && (2)\\
p_i[f \mapsto \neg s[f]] &\leq 1 - p_i[s] && \forall i \in [1,k], \forall s \in X \sqcup Y, \forall f \in \mathcal{F} && (3)\\
\sum_{f \in \mathcal{F}} p_i[f \mapsto \neg s[f]] &\geq 1 - p_i[s] && \forall i \in [1,k], \forall s \in X \sqcup Y && (4)\\
p_i[f \mapsto \text{True}] + p_i[f \mapsto \text{False}] &\leq 1 && \forall i \in [1,k], \forall f \in \mathcal{F} && (5)\\
p_i[s] &\in \mathbb{B} && \forall i \in [1,k], \forall s \in X \sqcup Y &&\\
p_i[f \mapsto b] &\in \mathbb{B} && \forall i \in [1,k], \forall f \in \mathcal{F}, \forall b \in \mathbb{B}. &&
\end{aligned}$$

For each partial state p_i, with $i \in \{1, 2, \ldots, k\}$, we have a boolean variable $p_i[f \mapsto b]$ for each $f \in \mathcal{F}$ and $b \in \mathbb{B}$,[17] and a boolean variable $p_i[s]$ for each $s \in X \sqcup Y$. $p_i[f \mapsto b] = 1$ means $p_i[f] = b$ and $p_i[s] = 1$ means $s \models p_i$.

The first set of constraints ensures each $s \in X$ is represented by some partial state p_i. The second set of constraints ensures each $s \in Y$ is *not* represented by any partial state p_i. The third and fourth sets of constraints

[17]Since our implementation uses SAS+ FOND planning tasks instead of STRIPS FOND planning tasks the actual IPs we produce has variables $p_i[x \mapsto v]$, where x is a variable and $v \in \texttt{dom}(x)$. Despite we present the IPs for STRIPS FOND planning tasks, every aspect described here can be directly translated to an analogous aspect we would have in IPs for SAS+ FOND planning tasks.

ensure the semantics of $\models$. The third enforces p_i *not* to map any fact f differently from s if $s \models p_i$. While the fourth enforces p_i to map some fact f differently from s if $s \not\models p_i$. The fifth set of constraints ensures p_i maps each fact to at most one truth valuation.

Since we try to solve the problem first with $k = 1$, then with $k = 2$, and so on, until it is solvable (and certainly it will be if k reaches $|X|$), we do not need to model the optimization of k in the integer program. The optimization function minimizes instead the total number of facts mapped through the fixed k partial states. While we could instead minimize zero, in order to have an easier integer program, this provides a much more human-readable output and also provides heuristic guidance for the integer programming solver we use.

We now simplify this integer program. Firstly, we note that since every $p_i[s]$ must be zero if $s \in Y$, then we can fully remove the second set of constraints by replacing these variables with a constant zero in every other constraint they appear. This makes the third set of constraints trivial for $s \in Y$, so we make the third set of constraints exist only for $s \in X$.

Secondly, we note that the fourth set of constraints is irrelevant for $s \in X$. They hinder $\sum_{f \in \mathcal{F}} p_i[f \mapsto \neg s[f]] = 0$ from occurring simultaneously with $p_i[s] = 0$ for some $s \in X$ and some p_i. However, allowing that causes no problem. Whenever that situation is possible, considering the other constraints, changing $p_i[s]$ to 1 is also possible. Thus no extra solution is ever permitted by removing the constraints for $s \in X$ in the fourth set of constraints. Moreover, we do not use the values of variables $p_i[s]$ when extracting the set of partial states p_i, so they do not need to be precise. Therefore, we make the fourth set of constraints exist only for $s \in Y$.

Thirdly, the fifth set of constraints is irrelevant because it can only hurt to set both $p_i[f \mapsto \text{True}]$ and $p_i[f \mapsto \text{False}]$ for some fact f, since this makes p_i useless as it will not be able to represent any state in X. And since we only try solving with k partial states after failing to solve with $k - 1$ (or if $k - 1 = 0$), no solution has some useless p_i. So we remove these constraints.

The integer program resulting from these simplifications is shown below.

$$\text{minimize} \quad \sum_{i=1}^{k} \sum_{f \in \mathcal{F}} \sum_{b \in \mathbb{B}} p_i[f \mapsto b]$$

$$\begin{aligned}
\text{subject to} \quad \sum_{i=1}^{k} p_i[s] &\geq 1 && \forall s \in X \\
p_i[f \mapsto \neg s[f]] &\leq 1 - p_i[s] && \forall i \in [1,k], \forall s \in X, \forall f \in \mathcal{F} \\
\sum_{f \in \mathcal{F}} p_i[f \mapsto \neg s[f]] &\geq 1 && \forall i \in [1,k], \forall s \in Y \\
p_i[s] &\in \mathbb{B} && \forall i \in [1,k], \forall s \in X \\
p_i[f \mapsto b] &\in \mathbb{B} && \forall i \in [1,k], \forall f \in \mathcal{F}, \forall b \in \mathbb{B}.
\end{aligned}$$

Additionally, we note that we just need to create a variable $p_i[f \mapsto b]$ only if there is at least one state $s \in X$ with $s[f] = b$. Otherwise, this variable is useless, as it will always be zero in some solution.

The approach of first subdividing the original compression problem into a subproblem for each action and then trying to solve it with an increasing number k of partial states using this optimized IP formulation was critical to have a sufficiently fast and memory-efficient procedure to compress the solutions returned by AND*.

6.4. Empirical Evaluation

We toggle on some satisficing features to AND*, to further improve its coverage, and we test the compressor on the solution policies returned by the resulting new version of AND*.

*6.4.1. Enabling a Few Satisficing Features to AND**

We toggle on the *deadlock detection* presented by Messa and Pereira (2023), which prunes policies with deadlocks upon generation. This procedure potentially makes AND* miss more solutions, as it may prune a policy with deadlocks that would otherwise be fixed by the concretizer[18], but helps to focus the search on the parts of the policy space that are more likely to lead to a solution. We also change the evaluation of policies from $f(\pi) = \Delta_{\downarrow}(\pi)$ to $f(\pi) = 2\Delta_{\downarrow}(\pi) - |\texttt{dom}(\pi)|$. This is equivalent to making AND* a "2-Weighted

[18] AND* with deadlock detection and domain-escape pruning is incomplete even if we enable the concretizer.

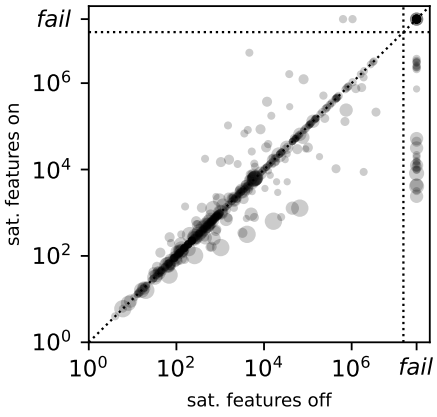

Figure 9: Per-task comparison on the number of generated policies before and after togging on the satisficing features.

AND*", inspired by Weighted A* (WA*) (Pohl, 1970). Finally, we change the classical planning heuristic used by $\Delta_{\downarrow}(\pi)$ from $h^{\text{LM-Cut}}$ (Helmert and Domshlak, 2009) to h^{FF} (Hoffmann and Nebel, 2001). Each of these three changes would alone make original AND* lose optimality, which is one of the reasons we did not apply them from the start (up to Section 4.5 we were still aiming for optimality). We also did not apply these changes earlier to make a clean comparison between the different pruning methods introduced, over a baseline policy-space search algorithm.

Table 6 shows the results of AND* with and without the satisficing features. We verify an increase in coverage from 0.728 to 0.900 after enabling the satisficing features. The highest increases occur in *blocksworld-original* (from 0.53 to 1), *blocksworld-advanced* (0.2 to 0.82), and *zenotravel* (0.33 to 0.87). Only in *islands* we observe a decrease in coverage (from 0.80 to 0.75).

Regarding the number of generated policies, the results are mixed. The average number of generated policies in the intersection of solved tasks substantially decreases in all but two IPC-FOND domains (all but *beam-walk* and *tireworld-triangle*), but increases in three NEW-FOND domains (*islands*, *miner* and *tireworld-spiky*). Figure 9 shows these mixed results, but also shows that for most of the tasks only AND* with the satisficing features solves within resource limits; it only needs to generate around 10,000 policies to solve them.

	sat. features off			sat. features on		
Domain	%S	%T	%M	%S	%T	%M
acrobatics	**1**	0	0	**1**	0	0
beam-walk	0.82	0.18	0	**1**	0	0
blocksworld-original	0.53	0.47	0	**1**	0	0
blocksworld-advanced	0.20	0.71	0.09	0.82	0.18	0
chain-of-rooms	**1**	0	0	**1**	0	0
earth-observation	0.33	0.15	0.53	0.45	0	0.55
elevators	0.80	0	0.20	0.93	0	0.07
faults	**1**	0	0	**1**	0	0
first-responders	0.75	0.24	0.01	0.99	0	0.01
tireworld-triangle	**1**	0	0	**1**	0	0
zenotravel	0.33	0.67	0	0.87	0.13	0
doors	**1**	0	0	**1**	0	0
islands	0.80	0.13	0.07	0.75	0	0.25
miner	0.53	0.47	0	0.59	0.18	0.24
tireworld-spiky	**1**	0	0	**1**	0	0
tireworld-truck	0.57	0	0.43	**1**	0	0
TOTAL	0.728	0.189	0.083	0.900	0.031	0.070

(a) Ratio of successful, timeout, and memory-out runs per configuration.

		sat. features off			sat. features on		
Domain	%∩S	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$	$t(s)$	$\#\pi$	$\lvert\pi_*\rvert$
acrobatics	**1**	3.36	11,109	126.50	1.44	318	126.50
beam-walk	0.82	2.45	454	453.22	3.86	454	453.22
blocksworld-original	0.53	229.86	270,708	14.12	0.91	482	15.44
blocksworld-advanced	0.20	0.49	1,730	9.64	0.20	220	9.91
chain-of-rooms	**1**	33.29	7,968	162.00	0.39	458	162.00
earth-observation	0.32	2.74	219,172	26.08	0.93	69,272	26.69
elevators	0.80	6.56	161,881	25.33	0.56	1,790	28.75
faults	**1**	0.87	1,050	24.31	0.70	123	23.00
first-responders	0.75	127.49	297,969	11.66	0.94	2,114	13.18
tireworld-triangle	**1**	30.31	485	244.00	5.99	485	244.00
zenotravel	0.33	2.72	689	14.80	0.89	174	15.60
doors	**1**	117.27	17,484	17,473.73	136.59	17,484	17,473.73
islands	0.75	31.10	109,284	6.11	54.24	352,310	6.11
miner	0.49	513.65	422,493	16.28	141.26	662,302	16.64
tireworld-spiky	**1**	4.75	5,913	25.73	1.37	6,712	25.73
tireworld-truck	0.57	1.25	5,618	14.64	0.79	3,000	14.55
TOTAL	0.723	11.13	13,869	46.96	2.34	2,439	48.16

(b) Average runtime, number of generated policies, and size of returned solutions, per configuration, over the intersection of solved tasks.

Table 6: Empirical results for AND* with and without the satisficing features.

6.4.2. Compression Results

We use IBM's ILOG CPLEX to solve integer programming problems.

Domain	%∩S	$\lvert\pi_*\rvert$	$\lvert\tau_*\rvert$
acrobatics	**1**	126.50	126.50
beam-walk	**1**	1,487.73	1,487.73
blocksworld-original	**1**	25.20	24.77
blocksworld-advanced	0.82	41.56	37.76
chain-of-rooms	**1**	162.00	162.00
earth-observation	0.45	34.17	28.78
elevators	0.93	33.79	29.71
faults	**1**	23.00	14.18
first-responders	0.99	16.84	15.42
tireworld-triangle	**1**	244.00	163.00
zenotravel	0.87	37.38	36.92
doors	**1**	17,473.73	18.00
islands	0.75	6.11	6.11
miner	0.59	16.97	16.97
tireworld-spiky	**1**	25.73	22.36
tireworld-truck	**1**	18.64	14.69
TOTAL	0.900	65.44	38.09

Table 7: Comparison on the size of the returned solutions before and after compression.

For each solved task, the compressor was able to compress the returned solution within the memory and time limits that remained after the execution performed by AND*. For the majority of the tasks, we are able to compress the solutions returned in less than 20 ms. Besides *p14* and *p15* of *doors*, and *p11* of *beam-walk*, all other solved tasks had their returned solutions compressed in at most 12 seconds. The most time-consuming compression (on *p15* of *doors*) took 55 seconds to complete. Only four domains have some task for which the returned solution took more than 250 ms to be compressed: *beam-walk*, *chain-of-rooms*, *doors*, and *tireworld-triangle*. The domain of *doors* is especially hard because, while the largest solution returned for some other domain has size 8,191 (from *p11* of *beam-walk*), the optimal solution size of *p15* of *doors* is 131,070 before compression.

Table 7 shows that the compressor reduces the size of the returned solution in 11 of the 16 domains. Decreases of at least 9% occur in eight domains: *blocksworld-advanced*, *earth-observation*, *elevators*, *faults*, *tireworld-triangle*, *doors*, *tireworld-spiky*, and *tireworld-truck*. In *doors*, the average size of returned solutions decreases by three orders of magnitude.

We note that if we were to run the compressor on the solution policies

returned by a configuration of AND* that returns solution policies with minimal domain size, the resulting compressed solutions would *not* necessarily be the most compact possible solution partial policies for the tasks. The compressor only ensures that the resulting compressed solution is the most compact solution partial policy that agrees with the input solution policy.

We also note that even though in some contexts performing data compression implies waiving human-readability, most of our compression gains came together with making solutions more human-readable. A clear example is *doors*, not only due to the dramatic decreases in the number of partial states, but also due to decreases in facts per mapping. Compressed solutions for *doors* had always either one or two facts per partial state, while complete states for *p15*, for example, have 34 facts each. We argue these decreases make the solution much easier to understand.

In the next section, we compare the after-compression size of AND* returned solutions with the size of solutions returned by two state-of-the-art FOND planners that natively reason over partial states.

7. Comparison with the State of the Art

In this section, we compare the best version of AND* (with escape equivalence pruning, structural symmetries, solution compression, and the satisficing features enabled in the previous section) with some FOND planners from the literature. We choose to compare AND* with PRP (Muise et al., 2012), FOND-SAT (Geffner and Geffner, 2018), IDFSP (Pereira et al., 2022), CFOND-ASP (Yadav and Sardina, 2023), and PR2 (Muise et al., 2024).

7.1. State-of-the-Art Planners

PRP (Muise et al., 2012) is a long-standing state-of-the-art satisficing FOND planner. PRP uses a classical planner to find trajectories to the goal. Starting by finding a trajectory from the initial state to the goal, it constructs a partial policy by repeatedly generalizing each new trajectory found using partial states. This process repeats until all the states reachable by the decompressed version of the current partial policy are covered, or until it concludes that one of them is a dead end. In the case of the latter, it learns a constraint that prevents the same error from occurring again after it restarts the search.

FOND-SAT (Geffner and Geffner, 2018) compiles FOND planning tasks into SAT instances. When trying to solve a FOND planning task, it con-

structs a series of SAT instances that encode requirements a partial policy (with an increasing number of partial states) would need to have in order to be a solution for the given task, until one of them is solved. From the solution of the solved SAT instance, it extracts a compact solution partial policy for the FOND planning task. FOND-SAT aims to have good scalability in tasks with high degrees of non-determinism due to the fact that different branching futures can be dealt with in conjunction with the SAT formulation.

IDFSP (Pereira et al., 2022) is a state-of-the-art satisficing FOND planner. IDFSP solves FOND planning tasks by performing an iterative depth-first *implicit* policy-space search. Bounded by an increasing search depth limit, it first tries to find a trajectory from the initial state to the goal. By doing so, it implicitly maps each state in the sequence to an action. Then, in a depth-first manner, it tries to find a goal trajectory for each other state reachable by this implicit policy. In case of a dead end or failure, it backtracks, removing mappings from the implicit policy. It uses different rules to cut short the exploration in some directions, taking into account the current search depth limit, the heuristic value of the states, and the previous explorations of those states. In case of ultimate failure, it restarts the search with an increased search depth limit.

CFOND-ASP (Yadav and Sardina, 2023), like FOND-SAT, compiles FOND planning tasks into another formalism. Specifically, it translates FOND planning tasks into Answer Set Programming (ASP) (Lifschitz, 2002) instances. Similar to FOND-SAT and PRP, CFOND-ASP generates a compact solution representation using partial states. A key distinguishing feature of CFOND-ASP is its novel ability to produce compact solutions that retain the best possible action for each complete state, unlike FOND-SAT, whose SAT-based formulation inherently trades state-specific decision precision for increased compactness.

PRP Rebooted (or PR2 for short) (Muise et al., 2024) is contemporary to this work. It makes several advancements over the original PRP, establishing itself as an unrivaled new state-of-the-art satisficing FOND planner. A key innovation in PR2 lies in its internal representation of the search states. It uses a pair of interconnected structures that efficiently link the explored state space with the incumbent partial policy. This dual-structure design accurately captures most of the inherent non-determinism required to handle dead ends and other critical aspects of FOND planning during the search process. Additionally, PR2 incorporates several clever engineering optimizations that exploit the structure of common FOND planning tasks to further

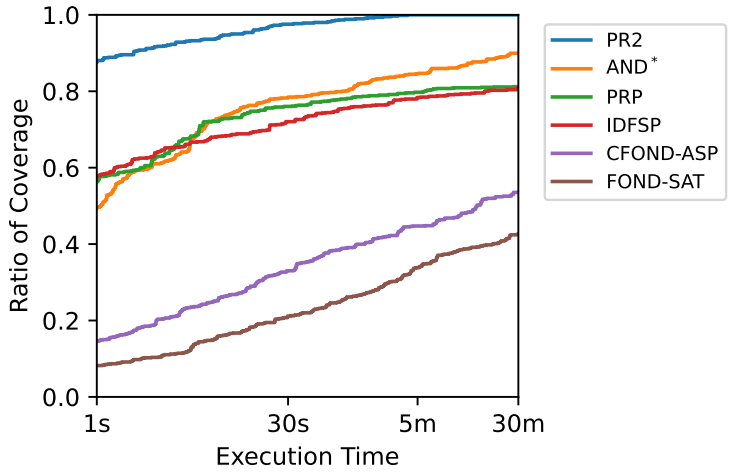

Figure 10: Aggregated ratio of solved tasks over time, for the different FOND planners.

enhance its performance.

We ran each of the FOND planners in their recommended settings. We compare them with the best version of AND*.

7.2. Coverage Results

Table 8 presents part of the results. From it, we can verify that AND* has the second best ratio of solved tasks (0.900), followed by PRP and IDFSP, which have similar ratios of solved tasks (0.812 and 0.808, respectively), then CFOND-ASP (0.535), and finally FOND-SAT (0.426). PR2 is the only planner that solves all tasks within the time and memory limits. Figure 10 shows that AND* has a slower start in comparison to PRP and IDFSP, and a better scalability with more resources.

We also verified that if we were to create a hybrid planner that can run any of the planners for a pre-determined amount of time, the combination that would be able to solve all the tasks allocating the least summed amount of pre-determined time (the *virtual best solver*) would be the one that runs AND* for 14.8s and PR2 for 27.1s, totaling 41.9s.

If we were to remove the option of using AND*, the virtual best solver would be to run *only* PR2 for 250.9s (that is the time it takes to solve the hardest task from *tireworld-triangle*). In other words, among the tested planners, AND* is, in some sense, the only one that still aggregates empirical value for solving this specific benchmark, after the introduction of PR2.

	FOND-SAT	CFOND-ASP	IDFSP	PRP	AND*	PR2
Domain	%S	%S	%S	%S	%S	%S
acrobatics	0.38	0.50	**1**	**1**	**1**	**1**
beam-walk	0.18	0.27	**1**	**1**	**1**	**1**
blocksworld-original	0.33	0.33	0.97	**1**	**1**	**1**
blocksworld-advanced	0.24	0.20	0.75	**1**	0.82	**1**
chain-of-rooms	0	0.10	**1**	**1**	**1**	**1**
earth-observation	0.12	0.17	0.65	**1**	0.45	**1**
elevators	0.47	0.47	0.53	**1**	0.93	**1**
faults	0.51	0.67	**1**	**1**	**1**	**1**
first-responders	0.59	0.55	0.69	**1**	0.99	**1**
tireworld-triangle	0.05	0.07	0.23	**1**	**1**	**1**
zenotravel	0.33	0.33	0.60	**1**	0.87	**1**
doors	0.67	**1**	0.93	0.80	**1**	**1**
islands	0.98	**1**	**1**	0.52	0.75	**1**
miner	0.94	0.98	**1**	0.27	0.59	**1**
tireworld-spiky	0.18	0.91	0.91	0.09	**1**	**1**
tireworld-truck	0.85	**1**	0.68	0.31	**1**	**1**
TOTAL	0.426	0.535	0.808	0.812	0.900	**1**

(a) Coverages per FOND planner.

		FOND-SAT		CFOND-ASP		IDFSP	PRP		AND*		PR2	
Domain	%∩S	$t(s)$	$\lvert\tau_*\rvert$	$t(s)$	$\lvert\tau_*\rvert$	$t(s)$	$t(s)$	$\lvert\tau_*\rvert$	$t(s)$	$\lvert\tau_*\rvert$	$t(s)$	$\lvert\tau_*\rvert$
acrobatics	0.38	8.09	8.33	1.07	8.33	0.02	2.55	8.33	0.04	8.33	0.03	8.33
beam-walk	0.18	2.85	11.00	0.31	11.00	0.01	0.14	11.00	0.04	11.00	0.04	11.00
blocksworld-original	0.33	27.76	10.10	9.85	10.50	0.06	0.01	10.70	0.41	10.70	0.01	10.70
blocksworld-advanced	0.20	27.04	8.82	8.57	8.82	0.06	0.01	9.82	0.21	9.55	0.01	9.82
chain-of-rooms	0	-	-	-	-	-	-	-	-	-	-	-
earth-observation	0.12	6.08	8.80	0.76	8.80	0.02	0.01	9.60	0.04	8.60	0.00	9.60
elevators	0.47	98.65	14.86	5.46	14.86	0.04	0.00	16.71	0.70	15.86	0.01	16.29
faults	0.51	256.62	10.25	13.25	11.25	0.07	0.01	10.25	0.67	10.25	0.01	10.25
first-responders	0.49	79.13	8.30	94.82	8.30	58.12	0.18	8.86	0.75	8.59	0.01	8.84
tireworld-triangle	0.05	18.90	11.00	2.34	11.00	0.03	0.01	16.00	0.05	11.00	0.00	11.00
zenotravel	0.33	205.84	14.80	639.85	14.80	0.51	0.02	21.60	0.90	15.40	0.01	21.60
doors	0.47	306.89	16.00	18.52	16.00	0.40	0.04	16.00	0.46	16.00	0.01	16.00
islands	0.52	11.25	5.84	1.20	5.84	0.06	49.16	5.84	1.67	5.84	0.00	5.84
miner	0.24	129.23	15.58	32.79	15.58	0.20	197.02	17.75	58.37	16.17	0.01	15.83
tireworld-spiky	0.09	419.76	22.00	34.75	22.00	0.07	20.04	22.00	0.07	22.00	0.01	22.00
tireworld-truck	0.26	123.07	12.11	2.43	12.11	32.15	170.05	23.05	0.48	12.32	1.34	15.84
TOTAL	0.290	48.43	11.23	7.50	11.33	0.15	0.21	12.81	0.35	11.47	0.01	12.06

(b) Average runtime and size of returned partial-state solutions, per FOND planner, over the intersection of solved tasks.

Table 8: Empirical results for the state-of-the-art FOND planners.

	FOND-SAT	CFOND-ASP	IDFSP	PRP	AND*	PR2
Benchmark	%S	%S	%S	%S	%S	%S
IPC-FOND	0.291	0.333	0.765	**1**	0.915	**1**
NEW-FOND	0.724	0.978	0.904	0.398	0.868	**1**

Table 9: Aggregated coverage per benchmark for the state-of-the-art FOND planners.

On the other hand, if we were to remove only the option of using PR2, the virtual best solver would be to run AND* for 14.8s, PRP for 17.7s, and IDFSP for 21.6s, totaling 54.1s, lower than the allocated time required when only the option of using AND* is removed. This indicates that AND* might introduce a distinguished set of strengths that better complement the capabilities of previous FOND planners.

PR2, AND*, and IDFSP are the only tested planners that achieve high coverage in both the NEW-FOND and the IPC-FOND benchmarks.

7.3. Compactness Results

We compare the (compressed) size of solutions returned by AND* with the size of solutions returned by PRP, FOND-SAT, CFOND-ASP, and PR2. We do not compare with IDFSP because it reasons over states and returns solutions over states, while the other four planners reason over partial states and return solutions over partial states. Table 10a presents the comparison with PRP. The nine domains in the table are the ones with at least one task — among the ones solved by both AND* and PRP — with different returned solution sizes. Each row presents the average returned solution size of AND* and PRP for the entire intersection of tasks solved by both of them. Table 10b, 10c, and 10d do the same, but respectively for PR2, FOND-SAT and CFOND-ASP. We note that we disregard the mapping of the goal condition to the no-op action, which some of these planners explicitly include in their solution partial policies.

We verify that there is only one domain where AND* returns on average bigger solutions than PRP: *blocksworld-advanced* (2.3% bigger). AND* returns solution partial policies on average at least 10% smaller for five domains: *earth-observation* (24.0% smaller), *elevators* (22.7%), *tireworld-triangle* (33.2%), *zenotravel* (20.0%), and *tireworld-truck* (42.7%). There are three other domains where AND* also returns smaller solutions on average: *blocksworld-original*, *first-responders*, and *miner*. For the remaining seven domains, AND* and PRP have tied in this metric.

Domain	%∩S	$\lvert\tau_*\rvert$		$\lvert\tau_*\rvert$
blocksworld-original	**1**	26.63	>	24.77
blocksworld-advanced	0.82	36.91	<	37.76
earth-observation	0.45	37.89	>	28.78
elevators	0.93	38.43	>	29.71
first-responders	0.99	16.81	>	15.42
tireworld-triangle	**1**	244.00	>	163.00
zenotravel	0.87	49.38	>	36.92
miner	0.24	17.75	>	16.17
tireworld-truck	0.31	22.39	>	12.83

(a) PRP vs. AND*

Domain	%∩S	$\lvert\tau_*\rvert$		$\lvert\tau_*\rvert$
blocksworld-original	**1**	26.63	>	24.77
blocksworld-advanced	0.82	36.91	<	37.76
earth-observation	0.45	37.17	>	28.78
elevators	0.93	37.64	>	29.71
first-responders	0.99	16.30	>	15.42
zenotravel	0.87	49.38	>	36.92
miner	0.59	16.87	<	16.97
tireworld-truck	**1**	19.31	>	14.69

(b) PR2 vs. AND*

Domain	%∩S	$\lvert\tau_*\rvert$		$\lvert\tau_*\rvert$
blocksworld-original	0.33	10.10	<	10.70
blocksworld-advanced	0.24	9.92	<	10.77
earth-observation	0.12	8.80	>	8.60
elevators	0.47	14.86	<	15.86
first-responders	0.59	8.98	<	9.39
zenotravel	0.33	14.80	<	15.40
miner	0.59	16.60	<	16.97
tireworld-truck	0.85	13.46	<	13.90

(c) FOND-SAT vs. AND*

Domain	%∩S	$\lvert\tau_*\rvert$		$\lvert\tau_*\rvert$
blocksworld-original	0.33	10.50	<	10.70
blocksworld-advanced	0.20	8.82	<	9.55
earth-observation	0.18	14.14	>	12.29
elevators	0.47	14.86	<	15.86
faults	0.67	12.76	>	11.76
first-responders	0.55	8.63	<	8.90
zenotravel	0.33	14.80	<	15.40
miner	0.59	16.60	<	16.97
tireworld-truck	**1**	14.26	<	14.69

(d) CFOND-ASP vs. AND*

Table 10: Comparison on the size of returned solutions, on the intersection of solved tasks, excluding the domains where both compared planners return solutions of the same size.

The comparison with PR2 shows similar results to the comparison with PRP. There are only two domains where AND* returns on average bigger solutions than PR2: *blocksworld-advanced* (2.3% bigger) and *miner* (0.6%). There are six domains where AND* returns smaller solutions on average: *blocksworld-original*, *earth-observation*, *elevators*, *first-responders*, *zenotravel*, and *tireworld-truck* — all but *blocksworld-original* and *first-responders* with at least 20% smaller solutions. For the remaining eight domains, AND* and PR2 have tied in this metric.

In comparison to FOND-SAT, the sizes of solutions returned by AND* are very comparable. The difference in the average returned solution size was never more than 10% for a domain. Nevertheless, for seven domains, FOND-SAT returned smaller solutions, while for only one, *earth-observation*, the opposite was observed. This is possible because, while FOND-SAT is expected, by design, to return compact solution partial policies, the SAT

formulation it uses has some restrictions that may make it miss the minimal-sized solution partial policy. For the remaining eight domains, AND* and FOND-SAT have tied in this metric.

The comparison with CFOND-ASP shows similar results to the comparison with FOND-SAT. The difference in the average returned solution size was usually less than 10%, except for *earth-observation*, where AND* returned on average 13.1% smaller solutions. AND* also returned smaller solutions on average for *faults*. On the other hand, it returned bigger solutions on average for seven domains. For the remaining seven domains, AND* and CFOND-ASP have tied in this metric.

8. Conclusion and Future Work

This work contributes to the field of fully-observable non-deterministic (FOND) planning by presenting new techniques and insights into policy-space search and solution representation. Central to our contributions is the study of policy equivalences and their impact on making policy-space search a more effective approach for solving FOND planning tasks.

Through an analysis of policy-space structures, we studied two distinct policy features – domain-escape and escape – that enable the derivation of policy equivalences with different guarantees and effectiveness. Using escape equivalence pruning substantially improved the performance of AND* (Messa and Pereira, 2023), the policy-space explicit search algorithm that we focus on in this work.

An additional contribution for the FOND planning community is the *concretizer*, a polynomial-time procedure that, given the states of a solution policy, deduces their correct mappings. In this work, the concretizer enables the sound and complete application of domain-escape pruning. Importantly, the concretizer illustrates that knowing the set of states that need to be mapped is sufficient to retrieve a solution policy, making it a potentially transformative tool for other FOND planners and probabilistic planning approaches.

We also showed that accounting for structural symmetries among states enables stronger policy pruning, and that using group theory techniques effectively allows a more rigorous computation of these symmetries. The efficient computation of canonical symmetries for FOND planning highlights the potential for using canonical symmetries in other planning paradigms.

We also introduce the *compressor*, a technique for reducing solution policies to their minimal unambiguous representation using partial states. On

our benchmark tasks, the compressor was computationally efficient and allowed AND* to achieve state-of-the-art solution compactness in most domains. Importantly, the compressor can compress decompressed solutions from any FOND planner, making it a useful tool for improving the compactness of solutions in the field.

In summary, this work presents a set of distinct new contributions for the FOND planning community. The new techniques make AND* competitive and provide a basis for designing more efficient algorithms and heuristics for FOND planning.

Future directions include developing heuristics tailored to policy-space search with equivalence pruning and exploring the use of partial states to compress information during search, possibly incorporating dead-end generalization techniques from PRP and PR2. Further research on the *concretizer* could unlock new possibilities, including its use in designing entirely new FOND algorithms. Specifically, we can leverage the ability to deduce solutions from domain sets D alone via concretizer calls on $\langle D, \mathcal{S}_* \rangle$, noting that we can efficiently check whether a state is in $\mathcal{S}_*$, despite its size.

Acknowledgments

We thank UFRGS, CNPq, CAPES, and FAPERGS for partially funding this research. The present work was carried out with the support of CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brazil. We acknowledge support from FAPERGS with project 21/2551-0000741-9. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. We thank the authors of Winterer et al. (2016) for making their code available.

References

Aineto, D., Gaudenzi, A., Gerevini, A., Rovetta, A., Scala, E., Serina, I., 2023. Action-failure resilient planning, in: Proceedings of the European Conference on Artificial Intelligence, IOS Press. pp. 44–51.

Albore, A., Palacios, H., Geffner, H., 2009. A translation-based approach to contingent planning., in: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1623–1628.

Bäckström, C., Nebel, B., 1995. Complexity results for SAS+ planning. Computational Intelligence 11, 625–655.

Beutner, R., Finkbeiner, B., 2024. Non-deterministic planning for hyperproperty verification, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 25–30.

Bonet, B., Geffner, H., 2020. Qualitative numeric planning: Reductions and complexity. Journal of Artificial Intelligence Research 69, 923–961.

Bonet, B., Giacomo, G.D., Geffner, H., Rubin, S., 2017. Generalized planning: Non-deterministic abstractions and trajectory constraints, in: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 873–879.

Camacho, A., Baier, J., Muise, C., McIlraith, S., 2018. Finite LTL synthesis as planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 29–38.

Cimatti, A., Pistore, M., Roveri, M., Traverso, P., 2003. Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence 147, 35–84.

Edelkamp, S., Schrödl, S., 2011. Heuristic Search: Theory and Applications. Elsevier.

Fikes, R.E., Nilsson, N.J., 1971. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial intelligence 2, 189–208.

Fu, J., Ng, V., Bastani, F., Yen, I.L., 2011. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems, in: Proceedings of the International Joint Conference On Artificial Intelligence, pp. 1949–1954.

Geffner, T., Geffner, H., 2018. Compact policies for fully observable nondeterministic planning as SAT, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 88–96.

Hansen, E.A., Zilberstein, S., 2001. LAO*: A heuristic search algorithm that finds solutions with loops. Artificial Intelligence 129, 35–62.

Hart, P.E., Nilsson, N.J., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost oaths. IEEE transactions on Systems Science and Cybernetics 4, 100–107.

Helmert, M., 2006. The Fast Downward planning system. Journal of Artificial Intelligence Research 26, 191–246.

Helmert, M., 2009. Concise finite-domain representations for PDDL planning tasks. Artificial Intelligence 173, 503–535.

Helmert, M., Domshlak, C., 2009. Landmarks, critical paths and abstractions: what's the difference anyway?, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 162–169.

Hoffmann, J., Nebel, B., 2001. The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14, 253–302.

Jefferson, C., Jonauskyte, E., Pfeiffer, M., Waldecker, R., 2019. Minimal and canonical images. Journal of Algebra 521, 481–506.

Kissmann, P., Edelkamp, S., 2009. Solving fully-observable non-deterministic planning problems via translation into a general game, in: Proceedings of the Annual German Conference on Artificial Intelligence, pp. 1–8.

Kuter, U., Nau, D., Reisner, E., Goldman, R.P., 2008. Using classical planners to solve nondeterministic planning problems, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 190–197.

Lifschitz, V., 2002. Answer set programming and plan generation. Artificial Intelligence 138, 39–54.

Littman, M.L., 1997. Probabilistic propositional planning: Representations and complexity. AAAI/IAAI , 748–754.

Mattmüller, R., Ortlieb, M., Helmert, M., Bercher, P., 2010. Pattern database heuristics for fully observable nondeterministic planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 105–112.

McDermott, D.M., 2000. The 1998 AI planning systems competition. AI magazine 21, 35–35.

McKay, B.D., Piperno, A., 2014. Practical graph isomorphism, II. Journal of Symbolic Computation 60, 94–112.

Messa, F., Pereira, A.G., 2023. A best-first search algorithm for fond planning and heuristic functions to optimize decompressed solution size, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 277–285.

Messa, F., Pereira, A.G., 2026. And-Star-Project. `https://doi.org/10.5281/zenodo.20027382`. doi:`10.5281/zenodo.20027382`.

Muise, C., Chakraborti, T., Agarwal, S., Bajgar, O., Chaudhary, A., Lastras-Montano, L.A., Ondrej, J., Vodolan, M., Wiecha, C., 2019. Planning for goal-oriented dialogue systems. arXiv preprint arXiv:1910.08137 .

Muise, C., McIlraith, S., Beck, C., 2012. Improved non-deterministic planning by exploiting state relevance, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 172–180.

Muise, C., McIlraith, S.A., Beck, J.C., 2024. PRP rebooted: Advancing the state of the art in FOND planning, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 20212–20221.

Muise, C.J., Felli, P., Miller, T., Pearce, A.R., Sonenberg, L., 2016. Planning for a single agent in a multi-agent environment using FOND., in: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 3206–3212.

Pereira, R.F., Pereira, A.G., Messa, F., De Giacomo, G., 2022. Iterative depth-first search for FOND planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 90–99.

Pochter, N., Zohar, A., Rosenschein, J., 2011. Exploiting problem symmetries in state-based planners, in: Proceedings of the AAAI conference on artificial intelligence, pp. 1004–1009.

Pohl, I., 1970. Heuristic search viewed as path finding in a graph. Artificial Intelligence 1, 193–204.

Ramirez, M., Sardina, S., 2014. Directed fixed-point regression-based planning for non-deterministic domains, in: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 235–243.

Rodriguez, I.D., Bonet, B., Sardina, S., Geffner, H., 2022. FOND planning with explicit fairness assumptions. Journal of Artificial Intelligence Research 74, 887–916.

Shleyfman, A., Katz, M., Helmert, M., Sievers, S., Wehrle, M., 2015. Heuristics and symmetries in classical planning, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 3371–3377.

Winterer, D., Wehrle, M., Katz, M., 2016. Structural symmetries for fully observable nondeterministic planning, in: Proceedings of the International Joint Conference On Artificial Intelligence, pp. 3293–3299.

Yadav, N., Sardina, S., 2023. A declarative approach to compact controllers for FOND planning via answer set programming, in: Proceedings of the European Conference on Artificial Intelligence, pp. 2818–2825.

Appendix A. Glossary of Symbols and Functions

$\mathcal{F}$	The set of facts, which might be either true or false in a state.
$\mathcal{F}[s]$	The set of facts that are true in the state s.
$\mathcal{S}$	The set of states.
$\mathcal{S}_*$	The set of *goal* states.
$\mathcal{A}$	The set of actions.
$\texttt{succs}(s, a)$	The set of all states that might be achieved if we apply the action a at the state s. Also called the successor states of s through a.
π	A partial function that maps non-goal states to actions. It is called a *policy*.
$\pi[s]$	The action that s is mapped to in π. $\pi[s] = \perp$ iff s is not mapped in π.
$\texttt{dom}(\pi)$	The set of states mapped in π — i.e., the domain of π.
$\texttt{reach}(\pi)$	The set of states reachable from some state in $\texttt{dom}(\pi)$ through the use of actions in π. This set includes the domain states themselves.
$\texttt{escape}(\pi)$	Equal to $\texttt{reach}(\pi) \setminus \texttt{dom}(\pi)$. A state in $\texttt{escape}(\pi)$ is called an escape state of π.
$\texttt{remain}(\pi)$	Equal to $\texttt{escape}(\pi) \setminus \mathcal{S}_*$ plus the initial state if it is non-goal and is not in the domain of the policy. A proper policy π is a solution iff $\texttt{remain}(\pi)$ is empty.
$\texttt{reach}(\pi, s)$	The set of states reachable from s through the use of actions in π. This set includes s itself.
$\texttt{escape}(\pi, s)$	Equal to $\texttt{reach}(\pi, s) \setminus \texttt{dom}(\pi)$. Also equal to $\texttt{reach}(\pi, s) \cap \texttt{escape}(\pi)$. A policy π is proper iff, for each domain state s, this set is *not* empty.
$\pi\vert_X$	Operation that returns the policy $\pi' = \{s \mapsto \pi[s] : s \in \texttt{dom}(\pi) \cap X\}$ for $X \subseteq \mathcal{S}$. In other words, it returns a copy of π pruned of states not in X.