Artificial Intelligence 245 (2017) 1–37

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

State space search nogood learning: Online refinement of critical-path dead-end detectors in planning

Marcel Steinmetz *, Jörg Hoffmann *

Saarland University, Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 2 August 2016
Received in revised form 6 December 2016
Accepted 12 December 2016
Available online 15 December 2016

Keywords:
Search
Heuristic search
Planning

ABSTRACT

Conflict-directed learning is ubiquitous in constraint satisfaction problems like SAT, but has been elusive for state space search on reachability problems like classical planning. Almost all existing approaches learn nogoods relative to a fixed solution-length bound, in which case planning/reachability reduces to a constraint satisfaction problem. Here we introduce an approach to learning more powerful nogoods, that are sound regardless of solution length, i.e., that identify *dead-end* states for which no solution exists.

The key technique we build on are *critical-path heuristics* h^C, relative to a set C of conjunctions. These *recognize* a dead-end state s, returning $h^C(s) = \infty$, if s has no solution even when allowing to break up conjunctive subgoals into the elements of C. Our key idea is to learn C during search. Whenever forward search has identified an *unrecognized* dead-end s, where $h^C(s) < \infty$, we analyze the situation at s, and add new conjunctions into C in a way guaranteeing to obtain $h^C(s) = \infty$. Thus we learn to recognize s, as well as similar dead-ends search may encounter in the future. We furthermore learn *clauses* ϕ where $s' \not\models \phi$ implies $h^C(s') = \infty$, to avoid the overhead of computing h^C on every search state. Arranging these techniques in a depth-first search, we obtain an algorithm approaching the elegance of nogood learning in constraint satisfaction, learning to refute search subtrees.

We run comprehensive experiments on solvable and unsolvable planning benchmarks. In cases where forward search can identify dead-ends, and where h^C dead-end detection is effective, our techniques reduce the depth-first search space size by several orders of magnitude, and often result in state-of-the-art performance.

1. Introduction

The ability to analyze conflicts, and to learn nogoods (or, dually, implied clauses) that avoid similar mistakes in the future, is a key algorithm ingredient in constraint satisfaction, including but not limited to SAT (e.g. [1–7]). For reachability problems in large transition systems, like goal reachability in classical planning which we consider here, progress in this direction has been more elusive. Existing techniques almost entirely pertain to *length-bounded reachability*, where the learned nogoods are valid only relative to a fixed bound on the length of the (remaining) solution path. In this setting, planning/reachability reduces to a constraint satisfaction problem. Via a SAT encoding (e.g. [8,9]), the respective nogood learning techniques apply unmodified. For state space search, conflicts in length-bounded search take the form of states unsolvable within the bound.

* Corresponding authors.
E-mail addresses: steinmetz@cs.uni-saarland.de (M. Steinmetz), hoffmann@cs.uni-saarland.de (J. Hoffmann).

http://dx.doi.org/10.1016/j.artint.2016.12.002
0004-3702/

This has been thoroughly investigated in the context of *Graphplan* [10–12], and more recently in *property-directed reachability (PDR)* [13,14] (which, as pointed out by Suda [14], can be viewed as an extension of Graphplan).

From the perspective of reachability testing, length-bounded reachability is a limitation, as one needs to iterate over different length bounds until some termination criterion applies. So, *do we actually need a length bound to be able to do conflict analysis and nogood learning?*

1.1. Nogood learning without a length bound

Research results in this direction are very sparse. For the identification of *dead-end* states in forward state space search – states from which no solution exists, arguably the most canonical form of "conflicts" in forward search – nogood learning techniques are unavailable.

In work done as part of the *Prodigy* system development in the late 80s/early 90s, sound action-pruning rules were learned by analyzing the structure of a backward search [15]. Kambhampati et al. thoroughly analyzed conflict-based learning, and its relation with CSP methods, in partial-order planning, i.e., a plan-space search framework [16,17]. Bhatnagar and Mostow [18] considered forward state space search nogood learning, yet their techniques are not sound (do not guarantee that pruned states actually are dead-ends). Kolobov et al.'s *SixthSense* technique [19] learns to detect dead-ends in probabilistic forward search planning, yet incorporates classical planning as a sub-procedure (SixthSense did, however, inspire part of our techniques, as we detail below). Value function refinement using Bellman updates (e.g. [20–23]) will eventually learn that a state is a dead-end, yet does not generalize that knowledge so is not a meaningful form of "nogood learning".

Nevertheless, the question of learning nogoods for dead-end detection is quite relevant. Such conflicts, though not as quintessential as in constraint satisfaction (including length-bounded reachability), are important in many applications. For example, bad decisions often lead to dead-ends in over-subscription planning (e.g. [24–26]), in planning with limited resources (e.g. [27–29]), and in single-agent puzzles like Sokoban (e.g. [30]) or Solitaire card games (e.g. [31]). In explicit-state model checking of safety properties (e.g. [32–34]), a dead-end is any state from which the error property cannot be reached – which one would expect to be the case for most states.

We introduce a method that learns sound and generalizable nogoods from dead-end states during forward state space search. To our knowledge, this is the first of its kind. Our work is placed in classical planning, where the state space is modeled in terms of finite-domain state variables, an initial state, deterministic transition rules (actions), and a conjunctive goal condition. But, in principle, the approach applies to reachability checking in other transition system models as well, so long as they are suitable for the design of so-called *critical-path heuristics*. We briefly discuss applications beyond classical planning at the end of the paper.

1.2. Dead-end detection methods in classical planning

We distinguish between (a) *dead-end detection* vs. (b) *proving unsolvability*. Both are closely related, but while (a) is dedicated to the detection of dead-ends during a forward state space search, (b) is dedicated to proving that the initial state is a dead-end. The difference lies in the attribution of computational resources: (a) will be done on every state during a search so should be fast, while (b) will be done just once so can use the entire computational resources available. At the algorithm design level, this means that techniques (a) will typically employ easy-to-test sufficient criteria, while (b) will explore a search space in some form. Forward state space search employing (a) is one method for doing (b), but is not limited to (b): dead-end detection can be useful also on solvable problems, as it allows to avoid fruitless parts of the search space. Our approach here falls into class (a).

Both (a) and (b) have been traditionally neglected in classical planning; (b) has even been completely ignored, the focus being exclusively on solvable problems. In particular, the benchmarks used in the *International Planning Competition (IPC)* are all solvable. First works designing techniques dedicated to (b) appeared in 2013 and 2014 [35,36]; (b) became the center of attention for the first time in 2016, with the inaugural *Unsolvability International Planning Competition (UIPC'16)*.[1]

That said, *heuristic functions* – estimators of goal distance – have been intensely investigated in classical planning (e.g. [39–45]), and most of these have the ability to detect dead-ends. They return heuristic value $h(s) = \infty$ if state s is unsolvable even in the relaxation underlying h. Yet this has traditionally been treated as a by-product of estimating goal distance. Hoffmann et al. [36] were the first to break with this tradition. They introduced the concept of *unsolvability heuristics*, reducing heuristic functions to dead-end detectors which either return ∞ ("dead-end") or 0 ("don't know"). They designed unsolvability heuristics based on state-space abstractions, specifically *merge-and-shrink* abstractions [46–48,45]. In UIPC'16, apart from merge-and-shrink abstractions [49], new unsolvability heuristics participated based on *pattern databases* [42,50], *potential heuristics* [51–53], and *critical-path heuristics* [39,54,55]. The latter is the approach we introduce here. All other dead-end detectors in UIPC'16 – all that exist at the time of writing – are statically fixed before search begins, i.e., they do not learn from search experience at all.

Apart from the mentioned unsolvability heuristics, UIPC'16 had many participants falling into class (b), including BDDs [56,57], partial delete-relaxation [58–60], and admissible pruning [61–63]. We will get back to this in the experiments.

[1] Some attention has been given to (a) in richer planning frameworks, in the aforementioned work by Kolobov et al. [19], as well as in work on the PRP system [37,38]. Both rely on expensive sub-procedures though, solving non-deterministic or classical planning problems as part of the dead-end detection.

1.3. Our approach

Critical-path heuristics lower-bound goal distance through the relaxing assumption that, to achieve a conjunctive subgoal G, it suffices to achieve the most costly *atomic* conjunction contained in G. In the original critical-path heuristics h^m, where m is a parameter, the atomic conjunctions are all conjunctions of size $\leq m$ [39]. This restriction was later on relaxed by Haslum [64]. As part of recent works on partial delete relaxation [65–68], the critical-path heuristic h^C was defined, whose parameter is a set C of atomic conjunctions that can be chosen freely.

It is well known that, for sufficiently large m, h^m delivers perfect goal distance estimates: simply set m to the number of state variables, reasoning over all relevant conjunctions. The latter is impractical of course, and indeed this guarantee is only of theoretical interest. Goal distance is preserved under the h^C relaxing assumption only if achieving the most costly atomic $c \subseteq G$ involves achieving the remaining subgoal $G \setminus c$ as a side effect. Practically feasible critical-path heuristics, in particular h^1, are known to typically be weak lower bounds (e.g. [40]).

However, matters change when we shift focus to dead-end detection rather than goal distance estimation. As a corollary of the above, *for appropriately chosen C, h^C detects all dead-ends*. This is a much more realistic ambition. First, it requires accuracy, not on all states, but only on the dead-ends. Second, it is quite natural for G to be unsolvable because some small $c \subseteq G$ is. For example, in resource-constrained planning, say our remaining fuel does not suffice to reach the 3 most distant locations, out of the 100 locations G requires us to visit.

In summary, h^C has the necessary power to refute arbitrary dead-end states in principle, and there is reason to believe it will be useful for that purpose in practice. Our idea is to use h^C for dead-end detection, learning the conjunctions C from the conflicts – the unrecognized dead-end states – encountered during search.

We denote the unsolvability-heuristic variant of h^C, that returns ∞ iff h^C does, by u^C. We initialize C, before search begins, to the set of singleton conjunctions. During search, components $\hat{S}$ of undetected dead-ends, where $u^C(s) = 0$ for all $s \in \hat{S}$, are identified (become *known*) when all their descendants have been explored. We show how to *refine* u^C on such components $\hat{S}$, adding new conjunctions into C in a manner guaranteeing that, after the refinement, $u^C(s) = \infty$ for all $s \in \hat{S}$.[2] The refined u^C may *generalize* to other dead-ends search may encounter in the future, i.e., refining u^C on $\hat{S}$ may result in detecting also other dead-end states $s' \notin \hat{S}$. As we show in our experiments, this can happen at massive scale.

The computation of u^C, like that of h^C, is low-order polynomial time in the number of atomic conjunctions $|C|$. Nevertheless, as C becomes larger, recomputing u^C on every search state may cause substantial runtime overhead. We tackle this with a form of *clause learning* inspired by Kolobov et al.'s [19] aforementioned SixthSense method. Whenever $u^C(s) = \infty$, we learn a minimal clause ϕ that any non-dead-end state must satisfy, specifically where $s' \not\models \phi$ implies $u^C(s') = \infty$. We do so by starting with the disjunction of facts p false in s, and iteratively removing p – adding p into s – while preserving $u^C(s) = \infty$; this method is inherited from SixthSense, which does essentially the same though based on h^2. Whenever we need to test whether a state s' is a dead-end, we first evaluate the clauses ϕ, and invoke the computation of $u^C(s')$ only in case s' satisfies all of these.

Arranging these techniques in a depth-first search, we obtain an algorithm approaching the elegance of nogood learning in constraint satisfaction: When a subtree is fully explored, the u^C-refinement and clause learning **(i) learns to refute that subtree**, **(ii) backjumps to the shallowest non-refuted ancestor**, and **(iii) generalizes to other similar search branches in the future**. Our experiments show that this can be quite powerful.

We provide a comprehensive evaluation with respect to the state of the art, for finding plans in solvable benchmarks with dead-ends, and for proving unsolvability in unsolvable benchmarks. We find that the success of our techniques depends on the extent of three structural properties of the input planning task: *conflict identification*, the ability of forward search to quickly find conflicts and thus enable the learning in the first place; *effective learning*, the ability to recognize dead-ends with small conjunction sets C; and *generalization*, the ability of u^C to detect states s' it was not refined on. On cases where the extent of all three properties is large, our techniques reduce depth-first search space size by orders of magnitude, and often result in performance competitive with, or surpassing, the state of the art. This is most pronounced on resource-constrained planning, specifically the benchmarks by Nakhost et al. [28] as well as over-constrained versions thereof. On competition benchmarks, this kind of structure is less frequent, but does appear in several domains from both the IPC and the UIPC.

For unsolvable benchmarks, we also evaluate the usefulness of the learned conjunction sets C as *unsolvability certificates* – a role they are suited to in principle, given they are efficiently verifiable (polynomial time in $|C|$), while potentially exponentially smaller than the state space itself.

Section 2 provides our basic notations and the necessary planning background. Section 3 provides an example walk-through to illustrate the workings of our techniques. Section 4 explains how we arrange forward search to identify, and learn from, dead-ends. Section 5 introduces two alternative methods for u^C refinement, applicable under different conditions. Section 6 discusses our clause learning technique. We describe our experiments in Section 7 and conclude in Section 8. Some proofs are moved out of the main text, and are available in Appendix A.

[2] Refinement methods adding new conjunctions into C were previously designed for the purpose of goal distance estimation via partial delete relaxation [66,69,67]. These methods can, in principle, be used in our context as well. Yet, as we will show, they are not well suited for dead-end detection in practice. Our new refinement methods, geared to that purpose, are typically superior.

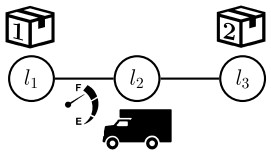

Fig. 1. Our illustrative running example.

2. Background

We consider the STRIPS framework for classical planning. A planning task is a quadruple $\Pi = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{F}$ is a set of propositions (*facts*), $\mathcal{I} \subseteq \mathcal{F}$ is the *initial state*, $\mathcal{G} \subseteq \mathcal{F}$ is the *goal*, and $\mathcal{A}$ is a set of *actions*. Associated with each action $a \in \mathcal{A}$ is a *precondition* $pre(a) \subseteq \mathcal{F}$, an *add list* $add(a) \subseteq \mathcal{F}$, and a *delete list* $del(a) \subseteq \mathcal{F}$. Since we are only concerned with goal reachability, which is independent of action costs, we assume unit action costs of 1 throughout.

A planning task is a compact representation of a transition system, its *state space* $\Theta^\Pi = \langle \mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{S}_\mathcal{G} \rangle$, where

- $\mathcal{S} = 2^\mathcal{F}$ is the set of all possible *states*. A *state* $s \in \mathcal{S}$ contains the facts considered to be true in s. All other facts are considered to be false.
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of *transitions*. There is a transition from state $s \in \mathcal{S}$ to state $s[[a]] \in \mathcal{S}$ via action $a \in \mathcal{A}$, denoted $s \xrightarrow{a} s[[a]]$, if a is *applicable* to s, i.e. $pre(a) \subseteq s$, and $s[[a]]$ is given by $(s \setminus del(a)) \cup add(a)$. If the action does not matter, we write transitions as $s \to s'$.
- $\mathcal{I}$ is Π's initial state.
- $\mathcal{S}_\mathcal{G} \subseteq \mathcal{S}$ is the set of *goal states*, i.e. all those $s \in \mathcal{S}$ where $\mathcal{G} \subseteq s$.

A *plan* for a state s is a path from s to some $t \in \mathcal{S}_\mathcal{G}$ in Θ^Π. A plan for Π is a plan for $\mathcal{I}$. If a state s does not have any plan, we call s a *dead-end*. If $\mathcal{I}$ is a dead-end, we say that Π is *unsolvable*. Deciding whether a plan for a state/for Π exists is **PSPACE**-complete [70].

Example 1. We will illustrate our techniques with a (simple) transportation planning task. Consider Fig. 1. There is one truck that should bring the two packages on the given map to their destinations, namely package 1 to l_3 and package 2 to l_1. The truck movements are subject to fuel consumption. There is no refueling so we need to make do with what's initially available.

For an initial fuel amount of 5, the problem can be modeled as the following STRIPS planning task $\Pi = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$. $\mathcal{F}$ consists of the facts $t(x)$ for $x \in \{l_1, l_2, l_3\}$ denoting the position of the truck; the location of the packages $p_1(x)$ and $p_2(x)$ for $x \in \{l_1, l_2, l_3, T\}$ (T denoting that the package has been loaded into the truck); and $f(x)$ for $x \in \{0, 1, \ldots, 5\}$ specifying the available amount of fuel. The initial state is $\mathcal{I} = \{t(l_2), p_1(l_1), p_2(l_3), f(5)\}$. The goal is $\mathcal{G} = \{p_1(l_3), p_2(l_1)\}$. There are actions $drive(x, y, z)$ to drive the truck from location x to location y, assuming that z fuel units are available and driving the truck consumes one fuel unit; actions $load(p, x)$ to load package p at location x; and actions $unload(p, x)$ to unload package p at location x. For example, $a = drive(l_1, l_2, 3)$ has precondition $pre(a) = \{t(l_1), f(3)\}$, add list $add(a) = \{t(l_2), f(2)\}$, and delete list $del(a) = \{t(l_1), f(3)\}$; and $a = unload(p_2, l_1)$ has precondition $pre(a) = \{p_2(T), t(l_1)\}$, add list $add(a) = \{p_2(l_1)\}$, and delete list $del(a) = \{p_2(T)\}$.

A plan for Π is to drive from l_2 to l_1, load p_1, drive to l_3, unload p_1 and load p_2, drive back to l_1, and unload p_2. This plan consumes all 5 fuel units. There is no plan that consumes less than 5 fuel units, so if we set the initial fuel amount to a value smaller than 5, then Π becomes unsolvable.

A *heuristic* is a function $h : \mathcal{S} \to \mathbb{N}_0^+ \cup \{\infty\}$. By h^*, we denote the *perfect heuristic*, which maps every s to the length of a shortest plan for s, or to ∞ if s is a dead-end.

Following Hoffmann et al. [36], we say that a heuristic u is an *unsolvability heuristic*, or *dead-end detector*, if u assigns each state to either ∞ or 0. The interpretation of $u(s) = \infty$ will be "dead-end", that of $u(s) = 0$ will be "don't know". We require u to be *sound*, i.e., whenever $u(s) = \infty$ then s is indeed a dead-end. In other words, there are no false positives. This is to preserve optimality/completeness when pruning states where $u(s) = \infty$. On the other hand, the dead-end detector may return $u(s) = 0$ although s is actually a dead-end, i.e., false negatives are possible. Following established terminology [71,72], we refer to dead-end states s where $u(s) = \infty$ as *recognized*, and to those where $u(s) = 0$ as *unrecognized*. The ideal dead-end detector would be the perfect one, denoted u^*, that recognizes all dead-ends, as this would allow us to prune all dead-ends during search. However, like h^*, computing u^* corresponds to solving the input planning task in the first place.

The family of *critical-path heuristics*, which underlie Graphplan [10] and were formally introduced by Haslum and Geffner [39], approximate the cost of achieving a fact conjunction, e.g. the goal, by the cost of achieving the most expensive *atomic conjunction* taken into account by the heuristic. In the original formulation, the critical-path heuristic h^m considered as atomic all fact conjunctions of size of at most m, $m \in \mathbb{N}$ being a parameter of the heuristic. This restriction was later on relaxed by Haslum [64], who designs a procedure approximating h^m, considering only part of the size-m subgoals. Most recently, Hoffmann and Fickert [73] defined the heuristic h^C, parameterized by an arbitrary set C of fact conjunctions.

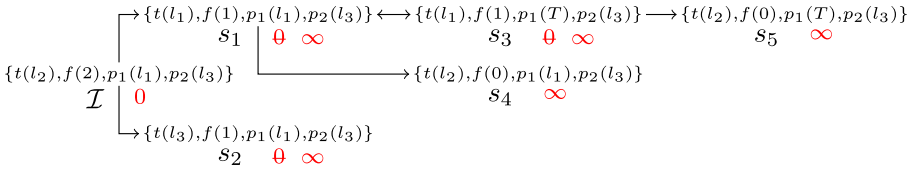

Fig. 2. The search space using our methods, on our running example (Fig. 1) with initial fuel amount 2. We annotate u^C values in red; crossed-out values become outdated after conflict-based u^C-refinement.

To formalize critical-path heuristics, we define the **regression** of a fact set G over an action a as $R(G,a) := (G \setminus add(a)) \cup pre(a)$ in case that $add(a) \cap G \neq \emptyset$ and $del(a) \cap G = \emptyset$; otherwise, the regression is undefined and we write $R(G,a) = \bot$. By $\mathcal{A}[G]$ we denote the set of actions where $R(G,a) \neq \bot$. We identify fact conjunctions with fact sets; let $C \subseteq 2^{\mathcal{F}}$ be any set of conjunctions. The generalized critical-path heuristic $h^C(s)$ is defined through $h^C(s) := h^C(s, \mathcal{G})$ where

$$h^C(s,G) = \begin{cases} 0 & G \subseteq s \\ 1 + \min_{a \in \mathcal{A}[G]} h^C(s, R(G,a)) & G \in C \\ \max_{G' \subseteq G, G' \in C} h^C(s,G') & \text{else} \end{cases} \quad (1)$$

Note here that we overload h^C to denote both, a function of state s in which case the estimated distance from s to the global goal $\mathcal{G}$ is returned, and a function of state s and *subgoal* G in which case the estimated distance from s to G is returned. We will use this notation convention throughout.

The definition of h^C distinguishes between three cases. If the subgoal is already true in the considered state s (top case), then its value is 0. If the subgoal G is not an atomic conjunction (bottom case), then its h^C value is estimated by the most expensive atomic subgoal that is a subset of G. Otherwise (middle case), G is an atomic subgoal that is not already true in the considered state. Then the h^C value is set to the cheapest possible way of achieving G, by minimizing over the actions that can be used to achieve G, and computing the resulting costs recursively for each.

Observe that, in the middle case, if there exists no action that can be used to achieve G, then the minimization evaluates to ∞, and thus the overall outcome value may be $h^C(s) = \infty$. Intuitively, this happens if s has no solution even when allowing to break up subgoals into the elements of C. As we are interested only in dead-end detection, not goal distance estimation, this is the main ability of h^C we are interested in. Consequently, most of the time we will consider not h^C but the *critical-path unsolvability heuristic*, denoted u^C, defined by $u^C(s) := \infty$ if $h^C(s) = \infty$, and $u^C(s) := 0$ otherwise.

One can compute h^C, solving Equation (1), in time polynomial in $|C|$ and the size of Π, using simple dynamic programming algorithms similar to those for h^m [39]. It is known that, in practice, h^m is reasonably fast to compute for $m = 1$, consumes substantial runtime for $m = 2$, and is typically infeasible for $m = 3$. The behavior is similar when using general conjunction sets C, in the sense that very large C can incur substantial runtime overhead. As hinted, we will use a clause-learning technique to alleviate this.

The conjunction set C is a very powerful algorithm parameter. The downside is, how to choose the value of that parameter? This question has been previously addressed only in the context of partial delete-relaxation heuristics [66,67,73], which extract delete-relaxed plans on top of h^C. All known methods learn C *offline*, prior to search, by iteratively refining a delete-relaxed plan for the initial state. Once this refinement process stops, the same set C is used throughout the search. Departing from this, here we learn C *online*, based on conflict analysis during search.

3. An illustrative example walkthrough

To provide the reader with an intuition before delving into the technical details, we next give an example walkthrough. We illustrate the overall search process, and how the learning **(i) refutes completed parts of the search**, **(ii) leads to backjumping**, and **(iii) generalizes to other similar search branches**.

Reconsider Example 1, and assume an initial fuel amount of 2. This is insufficient to transport the packages to their goal locations, so the planning task is unsolvable. However, to prove unsolvability, a standard state space search needs to explore all action sequences containing at most two *drive* actions. In particular, the search needs to explore two very similar main branches, driving first to l_1 vs. driving first to l_3. Using our methods, the learning on one of these branches immediately excludes the other branch.

Fig. 2 illustrates the search space for a depth-oriented search (with open & closed lists but expanding the deepest node first) using our methods. The set of atomic conjunctions C is initialized to the singleton conjunctions, i.e. single facts, $C = \{\{p\} \mid p \in \mathcal{F}\}$. In other words, with the initial conjunction set C, $u^C(s) = \infty$ iff $h^1(s) = \infty$. As regression over singleton subgoals ignores the delete lists, this is equivalent to the goal being delete-relaxed unreachable (relaxed-unreachable for short). In our example, this basically means to ignore fuel consumption so long as at least one fuel unit is left.

In the initial state $\mathcal{I}$, the goal is relaxed-reachable and we get $u^C(\mathcal{I}) = 0$. Thus, $\mathcal{I}$ is expanded: the truck can be driven to l_1 (resulting in state s_1) or to l_3 (resulting in state s_2). In both cases, the remaining fuel is decreased to 1. In particular, some fuel is still available, so the goal remains relaxed-reachable, $u^C(s_1) = u^C(s_2) = 0$. Say we expand s_1 next. Loading package p_1 results in state s_3, and since loading a package does not affect the fuel, we still have $u^C(s_3) = 0$. On the other hand,

driving the truck back to l_2 (state s_4) sets the available fuel to 0, and thus the goal becomes relaxed-unreachable from s_4, $u^C(s_4) = \infty$ (in fact, s_4 can be recognized trivially as a dead-end, since there are no actions).

Whenever we obtain $u^C(s) = \infty$ on some state s, in the present case $u^C(s_4) = \infty$, we call clause learning on s to identify a clause ϕ where, for all states s', $s' \not\models \phi$ implies $u^C(s') = \infty$. This is not mandatory in theory – the dead-end detection power of the learned clauses is dominated by that of u^C – but can be useful in practice, as it can reduce the number of calls to u^C and hence the runtime overhead. To identify the clause ϕ, we minimize the inverse state $\mathcal{F} \setminus s$ – the set of facts false in s – to obtain a minimal *reason* for $u^C(s) = \infty$. In the specific case of s_4, the learned clause is $\phi = t(l_3) \vee f(1) \vee f(2) \vee p_1(l_3)$. To understand that clause, observe that s_4 remains a dead-end regardless of the position of p_2; and, if p_1 is not at the goal location l_3, then it has to be transported to l_3 meaning that either the truck has to be at l_3, or there must be sufficient fuel to drive to l_3.

After the computation of ϕ has finished, search continues with the expansion of s_3, where p_1 can be unloaded (leading back to s_1), or the truck can be driven to l_2 (resulting in s_5). The latter state is a dead-end because we ran out of fuel. That will be recognized by u^C of course. Observe, however, that $s_5 \not\models \phi$, so we recognize s_5 to be a dead-end without even invoking the computation of $u^C(s_5)$.

Now the descendants of s_3 have been fully explored, so s_3 becomes a *known, yet unrecognized, dead-end*. In other words: search has encountered a conflict.

To learn from that conflict, we start a refinement of C, in a manner guaranteeing that u^C recognizes s_3 to be a dead-end after the refinement. In the present case, the refinement algorithm – precisely, "neighbors refinement" which will be explained in Section 5.2 – extends C by a single atomic conjunction, $c = \{t(l_2), f(1)\}$. The details of the refinement algorithm are technically involved, so we omit them here. For now, observe that adding c into C indeed suffices to obtain $u^C(s_3) = \infty$. This is because, to achieve the goal $p_1(l_3)$, we need to unload p_1 at l_3 which has $t(l_3)$ in its precondition. The $drive(l_2, l_3, z)$ actions achieving $t(l_3)$ have $f(z)$ in their precondition, which in s_3 is possible only for $z = 1$. However, $drive(l_2, l_3, 1)$ has the new atomic conjunction $c = \{t(l_2), f(1)\}$ in its precondition. Although each fact $t(l_2)$ and $f(1)$ is reachable from s_3 individually – drive to l_2 for $t(l_2)$, do nothing for $f(1)$ – they are not reachable in conjunction. The latter is recognized by u^C when including c into C, i.e., $u^C(s_3, c) = \infty$ for $C = \{\{p\} \mid p \in \mathcal{F}\} \cup \{c\}$. In summary, adding c to C lets u^C recognize that a single fuel unit is not enough to solve s_3, because there is insufficient fuel to drive to location l_3.

Having finished the u^C refinement on s_3, we call the clause learning on s_3. We learn the clause $t(l_2) \vee t(l_3) \vee f(2) \vee p_1(l_3)$. That clause does not hold in s_1, so we have shown (without invoking u^C on s_1) that $u^C(s_1) = \infty$. In particular, as advertised, (i) u^C now refutes the entire state space below the conflict node s_3, and (ii) search can backjump to the shallowest non-refuted ancestor, $\mathcal{I}$.[3]

Finally, s_2 is the only state left open. However, re-evaluating $u^C(s_2)$ before expanding s_2, we find that $u^C(s_2) = \infty$. Similarly as in s_3, getting the goal $p_2(l_3)$ requires to achieve the atomic conjunction $c = \{t(l_2), f(1)\}$ in the first place, yet that conjunction is unreachable from s_2. In other words, (iii) the knowledge learned on the previous search branch, in the form of the reasoning encapsulated by the extended conjunctions set $C = \{\{p\} \mid p \in \mathcal{F}\} \cup \{c\}$, generalizes to the present search branch.

With s_2 pruned, there are no more open nodes, and unsolvability is proved without ever exploring the option to drive to l_3 first. We could at this point continue, running the learning process on the now known-yet-unrecognized dead-end $\mathcal{I}$: if we keep running our search on an unsolvable task, then u^C eventually learns to refute the initial state itself.

We now explain these algorithms in detail. We cover the identification of conflicts during search, conflict analysis & refinement for u^C, and clause learning, in this order.

4. Forward search with conflict identification & learning

Assume some unsolvability heuristic u. We will henceforth refer to dead-end states s unrecognized by u, $u(s) = 0$, as *conflicts*. We wish to learn from conflicts, refining u, during search. For that purpose, we need to augment search to identify the conflicts in the first place; and we need to say where exactly to learn.

This is straightforward in principle, but subtleties arise from the need to do so efficiently. After any search step, how to navigate directly to the known conflict states? And actually, how to directly identify the components of such states, which may be (and is, for one of our u^C refinement methods) relevant to the learning?

We spell out these subtleties in what follows. We first (Section 4.1) design a generic extension to search algorithms using open & closed lists, like A*, greedy best-first search, etc., preserving their optimality and completeness guarantees. We then (Section 4.2) design a dedicated depth-first search variant, which has turned out to be most useful in our experiments as it identifies conflicts much more quickly, facilitating the learning process. Throughout this section, we consider an arbitrary unsolvability heuristic u, the only assumption being that there exists a refinement method which, given a conflict state s, refines u to recognize s.

[3] The latter would happen here anyway as s_1 has no open children, which furthermore (given the transition from s_3 back to its parent s_1) was necessary to identify the conflict at s_3. For an example with non-trivial backjumping, say we have packages $p_1, \ldots, p_n$ all initially at l_1 and with goal l_3, and one can unload a package only at its goal location. Then our method expands a single sequence of loading actions below s_1, learns the same conjunction $c = \{t(l_2), f(1)\}$ at the bottom, and backjumps all the way to $\mathcal{I}$. Similar situations can be constructed for non-symmetric packages.

Algorithm 1: Generic open & closed list based forward search algorithm, with conflict identification and learning.

```
Procedure GenericForwardSearch(Π)
    Open := {I}, Closed := ∅;
    while Open ≠ ∅ do
        select s ∈ Open;
        if G ⊆ s then
            return path from I to s;
        Closed := Closed ∪ {s};
        if u(s) = ∞ then
            continue;
        for all a ∈ A applicable to s do
            s' := s[[a]];
            if s' ∈ Closed then
                continue;
            if u(s') = ∞ then
                Closed := Closed ∪ {s'};
                continue;
            Open := Open ∪ {s'};
        CheckAndLearn(s);
    return unsolvable;
Procedure CheckAndLearn(s)
    /* loop detection                                                    */
    if s is labeled as dead-end then
        return;
    R[s] := {t | t reachable from s in Θ^search|_{Open∪Closed}};
    if R[s] ⊆ Closed then
        label s;
        /* refinement (conflict analysis)                                */
        refine u s.t. u(t) = ∞ for every t ∈ R[s];
        /* backward propagation                                          */
        for every parent t of s do
            CheckAndLearn(t);
```

4.1. Generic search algorithm

Algorithm 1 shows pseudo-code for our procedure. Consider first only the main loop, a generic search that can be instantiated into standard search algorithms in the obvious manner by suitable handling of the open and closed lists. The only difference to the standard algorithms then lies in the dead-end pruning, (a) via $u(s)$ at node expansion time, (b) via $u(s')$ at node generation time, and (c) via a call to the `CheckAndLearn()` procedure after state expansion. Of these, (a) and (b) are straightforward. A state is pruned, and considered closed, if it is detected to be a dead-end. Note that (a) makes sense despite the fact that s was already tested by (b) when it was first generated. This is because u may have been refined in the meantime, and may now recognize s to be a dead-end.

The conflict identification and learning process is organized by (c), the `CheckAndLearn()` procedure. Before explaining that procedure, we introduce some basic observations and terminology: we will capture the "knowledge" of the search in terms of a partial state-space graph. We require some notations identifying such graphs, and we require a simple comparison concept that will serve to identify the class of all possible search graphs consistent with the search knowledge.

Denote by $\Theta^\Pi = \langle \mathcal{S}^\Pi, \mathcal{T}^\Pi, \mathcal{I}^\Pi, \mathcal{S}^\Pi_\mathcal{G} \rangle$ the state space of our input task Π. We consider transition systems $\Theta = \langle \mathcal{S}, \mathcal{T}, \mathcal{I}^\Pi, \mathcal{S}_\mathcal{G} \rangle$ over subsets $\mathcal{S} \subseteq \mathcal{S}^\Pi$ of states, with $\mathcal{S}_\mathcal{G} = \mathcal{S}^\Pi_\mathcal{G} \cap \mathcal{S}$, and with potentially arbitrary transitions $\mathcal{T}$. Given such Θ, for a subset $\mathcal{S}' \subseteq \mathcal{S}$ of states, by $\Theta|_{\mathcal{S}'}$ we denote the subgraph of Θ induced by $\mathcal{S}'$. Given two such transition systems $\Theta^1 = \langle \mathcal{S}^1, \mathcal{T}^1, \mathcal{I}^\Pi, \mathcal{S}^1_\mathcal{G} \rangle$ and $\Theta^2 = \langle \mathcal{S}^2, \mathcal{T}^2, \mathcal{I}^\Pi, \mathcal{S}^2_\mathcal{G} \rangle$, and a state set $\mathcal{S}' \subseteq \mathcal{S}^1$ ($\mathcal{S}'$ will be the closed list below), we say that Θ^2 *coincides* with Θ^1 on $\mathcal{S}'$ if $\mathcal{S}' \subseteq \mathcal{S}^2$ and, for all $s \in \mathcal{S}'$, $s \to^2 s'$ if and only if $s' \in \mathcal{S}^1$ and $s \to^1 s'$.

Assume now any time point during search using Algorithm 1. We define a transition system reflecting the current search, namely, the transition system which is like $\Theta^\Pi|_{Open \cup Closed}$ except that closed states $s \in Closed$ that were pruned – that were detected to be dead-ends – do not have any outgoing transitions. We refer to this transition system as the *search graph*, and we denote it by Θ^{search}. We say that a state s is a *known dead-end* if, given the search graph – intuitively, given the search "knowledge" about Θ^Π so far – s must be a dead end. Formally, capturing the search knowledge as the set of transition systems Θ that coincide with Θ^{search} on *Closed*, s is a known dead-end if it is a dead-end in every such Θ. In other words, search knows s to be a dead-end if that is so in all state spaces indistinguishable from the present one given the search so far.

It is easy to see that the known dead-ends are exactly the states all of whose descendants in the search graph are already closed. That is, denoting

$$\mathcal{R}[s] := \{t \mid t \text{ reachable from } s \text{ in } \Theta^{\text{search}}|_{Open \cup Closed}\},$$

we have:

Proposition 1. *At any time point during the execution of Algorithm 1, the known dead-ends are exactly those states s where $\mathcal{R}[s] \subseteq Closed$.*

Proof. First, say that $\mathcal{R}[s] \subseteq Closed$. Consider any descendant state t of s in the current search graph. Then t is closed, either because it has already been expanded, or because it has been detected as a dead-end. In the former case, all outgoing transitions of t lead to states in *Closed*; in the latter case, t does not have any outgoing transitions. Let now Θ be a transition system over $\mathcal{S} \subseteq \mathcal{S}^{\Pi}$ as above, that coincides with Θ^{search} on *Closed*. Then all states reachable from s in Θ are contained in *Closed*. As $Closed \cap \mathcal{S}_{\mathcal{G}} = \emptyset$, s must be a dead-end in Θ, which is what we needed to prove.

Vice versa, if $\mathcal{R}[s] \not\subseteq Closed$, then some descendant t of s in the current search graph is still open. We can construct a counter-example Θ simply by extending $\Theta^{\text{search}}|_{Open \cup Closed}$ with a direct transition from t to some goal state. □

Given this, a naïve means to identify all known conflicts is to evaluate, after every state expansion and for every $s \in Closed$, whether $\mathcal{R}[s] \subseteq Closed$. But one can do much better than this, by a dead-end labeling procedure.

One might, at first sight, expect such a labeling procedure to be trivial, doing a simple bottom-up labeling following the reasoning that, if all direct successors of s are already known dead-ends, then s is a known dead-end as well. Such a simple procedure would, however, be incomplete, i.e., would in general not label all known dead-ends, due to cycles. If states s_1 and s_2 are dead-ends but have outgoing transitions to each other, then neither of the two will ever be labeled. Our labeling method, conducted as part of `CheckAndLearn()`, thus involves complete lookaheads on the current search graph, but on only those states that might actually have become a known dead-end given the last state expansion. Namely, a state t can only become a known dead-end after the expansion of a descendant s of t where $\mathcal{R}[s] \subseteq Closed$ after the expansion: otherwise, either the descendants of t have not changed at all, or t still has at least one open descendant.

Consider now the bottom part of Algorithm 1. In the top-level invocation of `CheckAndLearn(`s`)`, s cannot yet be labeled; the label check at the start of `CheckAndLearn()` is needed only for loop detection in recursive invocations, cf. below. The test on $\mathcal{R}[s]$ corresponds to Proposition 1. For $t \in \mathcal{R}[s]$, as t is reachable from s, we have $\mathcal{R}[t] \subseteq \mathcal{R}[s]$, and thus $\mathcal{R}[t] \subseteq Closed$, so all $t \in \mathcal{R}[s]$ are known dead-ends as well. Some t may be recognized already, $u(t) = \infty$, and thus (be dead-ends but) not be conflicts. If that is so for all $t \in \mathcal{R}[s]$, then the refinement step is void and can be skipped.

Backward propagation on the parents of s is needed to identify all dead-ends known at this time. Observe that the recursion will eventually reach all ancestors t of s, and thus all states t that might have become a known dead-end. Given the label check at the start of `CheckAndLearn()`, every state is labeled at most once and hence $|Closed|$ is an obvious upper bound on the number of recursive invocations, even if the state space contains cycles. Note that, in each recursive call, we can only label s itself, not all $t \in \mathcal{R}[s]$. This is because $\mathcal{R}[s]$ may contain ancestors t of s, and some other ancestor t' of s may be connected to s only via such t. In that case, t' would not be reached by the recursion.

In short, we label known dead-end states bottom-up along forward search transition paths, conducting a full lookahead of the current search graph in each. With the arguments outlined above (a full proof is available in Appendix A), this is sound and complete relative to the search knowledge:

Theorem 1. *At the start of the **while** loop in Algorithm 1, the labeled states are exactly the known dead-ends.*

Example 2. Reconsider the search space on our running example, depicted in Fig. 2. After expansion of s_3, the call to `CheckAndLearn(`s_3`)` constructs $\mathcal{R}[s_3] = \{s_3, s_1\}$, and finds that $\mathcal{R}[s_3] \subseteq Closed$. Thus s_3 is labeled, and u is refined to recognize s_3 and s_1. Backward propagation then calls `CheckAndLearn(`s_1`)`, the parent of s_3. As we have the special case of an ancestor $t \in \mathcal{R}[s]$, all states in $\mathcal{R}[s_1]$ are already recognized so the refinement step is skipped. The recursive calls on the parents of s_1, `CheckAndLearn(`s_3`)` and `CheckAndLearn(`$\mathcal{I}$`)`, find that s_3 is already labeled, respectively that $\mathcal{R}[\mathcal{I}] \not\subseteq Closed$, so the procedure terminates here.

It is worth noting that the search "knowledge" considered in the above is only the *explicit* knowledge, about states the search has already expanded or pruned. This disregards the *implicit* knowledge potentially present due to generalization: refining u on $\mathcal{R}[s]$ might recognize dead-ends $t' \notin \mathcal{R}[s]$. In particular, the search might have already visited t', $t' \in Open \cup Closed$, and then, via u, the search might actually already know that t' (and potentially some of its ancestors) are dead-ends.

One can capture this formally by denoting with $U := \{t' \mid t' \in \mathcal{S}, u(t') = \infty\}$ the currently recognized dead-end states; defining $\Theta^{\text{search}}[U]$ to be like Θ^{search} except that all $t' \in U$ have no outgoing transitions; and defining a state to be a *u-known dead end* if it is a dead-end in all Θ that coincide with $\Theta^{\text{search}}[U]$ on $Closed \cup (Open \cap U)$. The u-known dead-ends then are exactly those s where $\mathcal{R}[s] \subseteq Closed \cup U$. To find all these s during search – and thus immediately learn from all already identified dead-end states – after every state expansion, we would have to reevaluate u on the entire open and closed lists (plus backward propagation whenever a new dead-end is found). This would cause prohibitive overhead. Hence we stick to learning only on the known dead-ends, explicitly captured by the search.

An optimization we do apply in our instantiation of Algorithm 1 is to reevaluate the learned clauses (see Section 6) every time $\mathcal{R}[s]$ is computed: if a descendant of s is recognized by the clauses in the meantime, consider it closed. As clause evaluation is fast, this tends to pay off. In particular, it allows to not include detected dead-ends s into the closed list in the first place, as such s will be recognized by the clauses anyway.

Algorithm 1 has several desirable properties regardless of its concrete instantiation:

(1) **Preserving guarantees:** Instantiating the main loop to reflect any standard search algorithm, the optimality and/or completeness guarantees of that algorithm are preserved, as the only change is the pruning of dead-end states.
(2) **Unsolvability certificate:** Upon termination, we have $u(\mathcal{I}) = \infty$, due to the final call to `CheckAndLearn()`, doing backward propagation when all nodes are closed.
In case an unsolvability certificate is not required, the final call to `CheckAndLearn()` is redundant work. In our implementation, we provide an *early termination* option, which skips that step when the open list is already empty.
(3) **Bail-out:** Provided the unsolvability heuristic is *transitive*, search terminates without any further state expansions if an unsolvability certificate is already found. Here, we say that u is transitive if $u(s) = \infty$ implies $u(t) = \infty$ for all descendants t of s. This is a natural property for u to have – after all, proving s to be a dead end involves some form of reasoning about its descendants – and it does hold for the u^C unsolvability heuristic considered here. With transitive u, if $u(\mathcal{I}) = \infty$ then $u(s) = \infty$ for all s, hence no more states will be expanded.

We remark that the problem of labeling the known dead-end states relates closely to cost revision steps in cyclic AND-OR graphs, as done in AO* and other AND-OR search algorithms (e.g. [74–76]). It is equivalent to labeling solved nodes when viewing states as AND nodes, and viewing actions as trivial OR nodes with a single outgoing edge (action outcome). Effective methods for labeling solved nodes, without complete forward lookaheads, have been designed, yet these exploit non-trivial OR nodes (through, e.g., focusing on a current greedy policy [75]). It remains an open question whether such methods can be beneficial for our purposes. In any case, as we shall see next, in depth-first search – which turns out to be most useful in practice – the issue of forward lookaheads disappears.

4.2. Depth-first search

Depth-first search (DFS) is particularly well suited for our purposes, because it fully explores the descendants of a state before proceeding with anything else. In other words, DFS is geared at obtaining $\mathcal{R}[s] \subseteq \mathit{Closed}$ as quickly as possible. This is key to identifying conflicts quickly.

But what exactly does DFS look like in our context? The issue is that state spaces are, in general, cyclic, and nodes may have solutions via their parents. In our running example, s_3 has a transition to its parent s_1. A simple way to tackle this is what we will refer to as *depth-oriented search (DOS)* (as previously indicated in Section 3), instantiating Algorithm 1 with a depth-first search order, ordering the open list by decreasing distance from the initial state.

It turns out that one can do better though. We next design an elegant DFS variant of our approach, similar to backtracking in constraint satisfaction problems. Consider first, to get some intuitions, the acyclic case. This is restricted yet not entirely unrealistic: acyclic state spaces naturally occur, e.g., if every action consumes a non-0 amount of budget or resource. In DFS on an acyclic state space, state s becomes a known dead-end exactly the moment its subtree has been completed, i.e., when we backtrack out of s. Hence we can simply refine u at this point. As the same has previously been done on the children s' of s, we will have $u(s') = \infty$ for every such s', so the conflict component $\mathcal{R}[s]$ simplifies to just s. Overall, the complex `CheckAndLearn` procedure can be replaced by refining u on s at backtracking time. But then, we do not need the open and closed lists anymore, and can instead use a classical DFS.

In the cyclic case, matters are not that easy. But it turns out that one can obtain a valid DFS algorithm (which defaults to classical DFS in the acyclic case) from Tarjan's algorithm to compute maximal strongly connected components (SCCs) [77]. (Which has previously been put to use in certain dynamic programming algorithms for probabilistic planning [78,23].)

Algorithm 2 shows the pseudo-code. The key observation is that s becomes a known dead-end exactly at the moment when we have identified the maximal SCC $S \subseteq \mathcal{S}$ that contains s, i.e., once DFS backtracks out of the last state in S. This is simply because, with $\mathcal{R}[s] \subseteq \mathit{Closed}$, we must also have $\mathcal{R}[t] \subseteq \mathit{Closed}$ for any ancestor state t of s reachable from s. Thus, to get rid of the expensive `CheckAndLearn` procedure, DFS can use Tarjan's algorithm to identify the maximal SCCs, and refine u whenever a maximal SCC has been found. Henceforth, whenever we say "DFS", we mean DFS as per Algorithm 2.

Regarding the properties of DFS, obviously property (1 preserving guarantees) from above is not meaningful here, and DFS is complete but not optimal. DFS inherits properties (2 unsolvability certificate) and (3 bail-out). Like for Algorithm 1, we implemented a simple *early termination* option in case an unsolvability certificate is not desired. DFS furthermore has several desirable properties beyond (2) and (3):

(4) **Backjumping:** Due to the pruning test on $u(s) = \infty$ inside the state-expansion loop, DFS will backjump across predecessor states s that are now recognized dead-ends. For transitive unsolvability heuristics, the backjump will be across *all* recognized dead-ends on the current search path, as $u(s) = \infty$ implies $u(t) = \infty$ for all t below s.
(5) **Immediate u-known learning:** DFS guarantees to learn, before the next state expansion, on all dead-ends t' that are u-known but not known, and where $u(t') \neq \infty$ (there is still something to learn on t'). To see this, let t' be such a state.

Algorithm 2: Depth-first search (DFS), with conflict identification and learning following Tarjan's algorithm.

```
Global variables: N := 0;
                  stack := empty stack;
                  idx, lowlink functions S ↦ ℕ ∪ {∞},
                  initially idx(s) = lowlink(s) = ∞ for all s ∈ S;
Procedure DFS(s)
    if G ⊆ s then
        return true;
    if u(s) = ∞ then
        return false;
    idx(s) := N; lowlink(s) := N; N := N + 1; push s onto stack;
    for each a ∈ A applicable to s do
        s' := s[[a]];
        if idx(s') = ∞ then
            if DFS(s') then
                return true;
            else if u(s) = ∞ then
                return false;
            else
                lowlink(s) := min{lowlink(s), lowlink(s')}
        else if s' is on stack then
            lowlink(s) := min{lowlink(s), idx(s')}
    if idx(s) = lowlink(s) then
        R[s] := ∅;
        while s ∉ R[s] do
            t := stack.top(); stack.pop();
            R[s] := R[s] ∪ {t};
        refine u s.t. u(t) = ∞ for every t ∈ R[s];
    return false;
```

As t' is not a known dead-end, it must lie along the current search path $\vec{t}$. As $u(t') \neq \infty$ but t' is a u-known dead-end, t' must be an inner node along $\vec{t}$, and every leaf node s along $\vec{t}$ reachable from t' must satisfy $u(s) = \infty$. But then, search backtracks out of the SCC containing t' without any further state expansions, which is what we needed to show.

(6) **Duplicate pruning for free:** As u learns to refute the subtree below s, it subsumes the duplicate pruning that would be afforded by a closed list. Due to generalization, if will often surpass that pruning by far.

Compared to this, in the generic search of Algorithm 1, towards (4) one can test, at the start of the `CheckAndLearn()` procedure, whether $u(s) = \infty$. This leads to backjumping in depth-oriented search, and leads to aggressive pruning of search paths in other open list based searches like greedy best-first search. For (5), as discussed this does not hold for Algorithm 1 in general, as the new u-known states may lie on arbitrary search paths; it does hold for depth-oriented search though. Finally, (6) is specific to DFS, and cannot be exploited by Algorithm 1 regardless of the search order, as that algorithm needs to maintain a closed list anyway. Intuitively, depth-first search is closer to the structure of dead-end detection, and combines more gracefully with it than other search algorithms.

For both Algorithm 1 and Algorithm 2, in practice it is often useful to combine several dead-end detectors $\{u_1, \ldots, u_k\}$, instantiating u in the respective pseudo-code with $\max_i u_i$ to profit from complementary dead-end detection capabilities. The refinement step then in principle allows arbitrary combinations of refinements on the individual u_i. Here, we will empirically investigate the combination of u^C with the aforementioned dead-end detectors u based on merge-and-shrink abstraction [36,49] respectively potential heuristics [52,53]. As refinement methods for the latter are not available at the time of writing, we will refine u^C only. A subtlety that arises in this context regards the handling of dead-end states recognized by u but not by u^C. We now consider the refinement step in detail.

5. Conflict analysis & refinement for critical-path heuristics

We now tackle the refinement step in Algorithms 1 and 2, for the dead-end detector u^C. Given $\mathcal{R}[s]$ where all $t \in \mathcal{R}[s]$ are dead-ends, how to refine u^C on $\mathcal{R}[s]$ to recognize all these dead-ends?

Naturally, the refinement will add a set X of conjunctions into C. A suitable refinement is always possible, i.e., there exists X s.t. $u^{C \cup X}(s) = \infty$ for all $t \in \mathcal{R}[s]$. But how to find such X?

One possibility is to use known conjunction-learning methods from the literature [66,69,67], which iteratively remove conflicts in delete-relaxed plans for a given state s. These methods do guarantee to eventually recognize s if it is a dead-end. But they are not geared to this purpose, and as we shall see, are not effective in practice for that purpose. Here we introduce two methods specifically designed for dead-end detection: *path-cut refinement* and *neighbors refinement*.

The major difference between the two methods lies in their applicability. Neighbors refinement applies only to $\mathcal{R}[s]$ that satisfy what we call the u^C*-recognized neighbors* property. Consider the neighboring states t of $\mathcal{R}[s]$, i.e., those with

Algorithm 3: A single step of path-cut refinement on a state s. The initializing call to the algorithm is on $G := \mathcal{G}$ and $n := h^C(s)$. The algorithm assumes that $h^C(s) < h^*(s)$, and identifies a set X of conjunctions so that $h^C(s) < h^{C \cup X}(s)$. s, C, and X are global variables; $X := \emptyset$ is set initially.

```
Procedure PathCutRefine(G, n)
    if n = 0 then
        /* We know here that G ⊈ s */
        let p ∈ (G \ s); x := {p};
    else
        /* Select an atomic conjunction (invariant: h^C(s, G) ≥ n) */
        let c ∈ C s.t. c ⊆ G and h^C(s, c) ≥ n;
        x := c;
        if h^C(s, c) = n then
            /* Cut each path that achieves c */
            for every action a ∈ A[c] do
                if del(a) ∩ G ≠ ∅ then
                    let p ∈ del(a) ∩ G;
                    x := c ∪ {p};
                    /* ⇒ a is no longer an achiever of x */
                else
                    x' := PathCutRefine(R(G, a), n − 1);
                    x := x ∪ (x' \ pre(a));
                    /* ⇒ R(x, a) contains x' */
    X := X ∪ {x};
    return x;
```

an incoming transition from $\mathcal{R}[s]$. All these t must already be recognized as dead-ends. But recognized by which dead-end detector? We say that $\mathcal{R}[s]$ has u^C-recognized neighbors if all t are recognized by u^C, $u^C(t) = \infty$. This necessarily holds if u^C is the only dead-end detector used. But if u^C is combined with some other dead-end detector u, then some of the states t may be recognized only by u, not by u^C.

It turns out that the recognized neighbors property can be exploited for an especially effective refinement method, neighbors refinement. For the general case, we design the alternate path-cut refinement method.

Path-cut refinement (Section 5.1) learns conjunctions X cutting off the critical paths in a h^C computation reaching the goal. One such refinement step guarantees to strictly increase the value of h^C. To render h^C infinite as desired, we need to iterate these refinement steps, recomputing h^C in between iterations. Neighbors refinement (Section 5.2), in contrast, is a constructive method, identifying the new conjunctions X directly from those for the neighbor states, without necessitating any intermediate recomputations of u^C.

5.1. Path-cut refinement

Path-cut refinement assumes some arbitrary dead-end state s as input, and augments C to recognize s. To recognize all dead-ends within the component $\mathcal{R}[s]$, we run the method on s only. Due to the aforementioned transitivity property of u^C, this suffices to recognize all states in $\mathcal{R}[s]$.

The refinement is based on cutting off *critical paths*, i.e., the recursion paths in the definition of h^C (Equation (1)). The refinement is iterative, where each iteration identifies a set X of conjunctions adding which into C guarantees to strictly increase $h^C(s)$. Given this, the method really pertains to h^C rather than the simplified u^C, and it applies not only to dead-end states, but to any state s where $h^C(s) < h^*(s)$. Therefore, for the remainder of this subsection, we will talk about h^C, not u^C. At the end of the refinement on a dead-end state s, we will have $h^C(s) = u^C(s) = \infty$.

We consider now in detail a single refinement step (one iteration of the overall refinement). In what follows, like in Equation (1) we use $h^C(s, G)$ to denote the h^C value of subgoal fact set G, i.e., the approximated cost, given the h^C relaxation of achieving G from s. Correspondingly, we use $h^*(s, G)$ to denote the real cost of achieving G from s. The h^C recursion path on a current subgoal G is cut off by identifying a small conjunction $x \subseteq G$ that cannot be achieved with action sequences of length at most $h^C(s, G)$. The union of these x over all critical recursion paths yields the desired set X. Algorithm 3 shows the pseudo-code.

To understand Algorithm 3, consider the initializing call on $G = \mathcal{G}$ and $n = h^C(s)$. Our aim is to identify a (small) conjunction $x \subseteq \mathcal{G}$ that cannot be achieved from s by any action sequence of length at most $h^C(s, \mathcal{G})$. Towards finding such x, we start by selecting an arbitrary critical (maximum h^C value) atomic conjunction $c \in C$, $c \subseteq \mathcal{G}$. We initialize $x := c$. As $h^C(s, c) = n$, c is achieved, under the h^C approximation, by an action sequence of length n. However, as $n = h^C(s, \mathcal{G}) < h^*(s, \mathcal{G})$, we know that we can extend $x = c$ with additional facts $p \in \mathcal{G} \setminus c$ in a way excluding that case, i.e., making x achievable under h^C only by action sequences of length $> n$.

To find suitable facts p for extending x, we recursively consider the actions $a \in \mathcal{A}[c]$, i.e., the actions that can achieve c (that add part of c and delete none of it). For each of these, we augment x so that there is a conjunction $x' \subseteq R(x, a)$

that cannot be achieved with action sequences of length at most $n-1$. If a deletes part of G, we can tackle a simply by adding one such deleted fact p into x, effectively removing a from the set of achievers of x. For the remaining actions a, we recursively identify a suitable $x' \subseteq R(G,a)$. The latter is necessarily possible as we will always have $h^C(s,G) < h^*(s,G)$ (in particular, $G \not\subseteq s$ at the recursion termination $n=0$). We then extend x in a way ensuring that x' is contained in the regression $R(x,a)$, implying that x cannot be reached at time n.

Spelling out these arguments (see the proof in Appendix A), one obtains that `PathCutRefine` is correct:

Theorem 2. *Let C be any set of atomic conjunctions. Let s be a state with $h^C(s) < h^*(s)$. Then:*

(i) *The execution of* `PathCutRefine`$(\mathcal{G}, h^C(s))$ *is well defined, i.e., (a) in any call* `PathCutRefine`(G,n) *there exists $c \in C$ so that $c \subseteq G$ and $h^C(s,c) \geq n$; and (b) if $n=0$, then $G \not\subseteq s$.*
(ii) *If X is the set of conjunctions resulting from* `PathCutRefine`$(\mathcal{G}, h^C(s))$*, then $h^{C\cup X}(s) > h^C(s)$.*

As a single call to `PathCutRefine` only guarantees to increase $h^C(s)$ by at least 1, for dead-end refinement we need to iterate these calls, setting $C := C \cup X$ after each call, until $h^C(s) = \infty$ holds. This is guaranteed to eventually happen, simply because every iteration adds at least one new conjunction to C (otherwise, the value of h^C could not have increased), and the number of conjunctions is finite. In the worst case, C eventually contains all conjunctions, and $h^C(s) = \infty$ holds trivially.

Example 3. Consider again the search space of our running example in Fig. 2. After expanding s_3, all of its children are either closed or recognized under u^C. Thus, s_3 becomes a known, though unrecognized dead-end. At this point in time C consists only of the unit conjunctions, $C = \{\{p\} \mid p \in \mathcal{F}\}$, and hence $h^C = h^1$. Say that we now conduct path-cut refinement on s_3 to suitably extend C.

In the first call to `PathCutRefine`, we have $G_4 := \mathcal{G} = \{p_1(l_3), p_2(l_1)\}$ and $n = h^C(s) = h^1(s) = 4$. There is only one option for the selection of c, because $h^1(s, \{p_1(l_3)\}) = 3 < 4 = h^1(s, \{p_2(l_1)\})$. So we choose $c_4 = \{p_2(l_1)\}$ and initialize the conflict to $x_4 := c_4$. To see whether $p_2(l_1)$ can be reached with an action sequence of length no longer than 4, and thus to determine whether we have to augment x_4 by $p_1(l_3)$, we continue with the recursion on the only achiever of c_4, $unload(p_2, l_1)$. This yields a recursive call on $G_3 := R(G_4, unload(p_2,l_1)) = \{p_1(l_3), t(l_1), p_2(T)\}$.

In `PathCutRefine`$(G_3, 3)$, there are two options for choosing the conjunction c_3, namely $c_3 = \{p_1(l_3)\}$ and $c_3 = \{p_2(T)\}$. Say that we consider the critical path responsible for the top-level goal $p_2(l_1)$, i.e., we choose $c_3 = \{p_2(T)\}$, and we set $x_3 := c_3$. With the critical-path action $load(p_2, l_3)$, this yields a recursive call on $G_2 := R(G_3, load(p_2,l_3)) = \{p_1(l_3), t(l_1), t(l_3), p_2(l_3)\}$. From the options in that recursive call, say we consider again the critical path responsible for $p_2(l_1)$, setting $c_2 = \{t(l_3)\}$ and $x_2 := c_2$. With the critical-path action $drive(l_2, l_3, 1)$, this yields a recursive call on $G_1 := R(G_2, drive(l_2,l_3,1)) = \{p_1(l_3), t(l_1), p_2(l_3), t(l_2), f(1)\}$. Considering again the critical path responsible for $p_2(l_1)$, we select $c_1 = \{t(l_2)\}$ and set $x_1 := c_1$. Now, the only supporting action to be considered is $drive(l_1, l_2, 1)$ (higher fuel levels yield unreachable preconditions under the current h^C already). However, we do not require a recursive call over that action, as $drive(l_1, l_2, 1)$ deletes $t(l_1)$ and $f(1)$, which are both part of our subgoal G_1. Say we extend x_1 with the deleted fact $f(1)$.

The recursion now goes back up the recursion path over the levels $i \in \{2,3,4\}$, corresponding to the actions $drive(l_2,l_3,1)$, $load(p_2,l_3)$, $unload(p_2,l_1)$, with the current conjunctions $x_2 = \{t(l_3)\}$, $x_3 = \{p_2(T)\}$, $x_4 = \{p_2(l_1)\}$. At each step, we extend x_i with x_{i-1} minus the respective action's precondition. At $i=2$, $x_{i-1} = \{t(l_2), f(1)\}$; both are in the precondition of $drive(l_2,l_3,1)$, so x_2 remains the same. But then, x_3 and x_4 also remain the same in the remaining steps. Therefore, upon termination the only non-singleton conjunction in X is $\{t(l_2), f(1)\}$. Recall from our example walkthrough in Section 3 that this is exactly the single conjunction needed to render $u^{C\cup X}(s_3) = u^{C\cup X}(s_1) = \infty$.

5.2. Neighbors refinement

Neighbors refinement assumes as input a set $\hat{S}$ of dead-end states that satisfies the *u^C-recognized neighbors property*. Namely, we denote by $\hat{T}$ the *neighbors* of $\hat{S}$, i.e., the set of states $t \notin \hat{S}$ where there exists $s \in \hat{S}$ s.t. $s \to t$. The u^C-recognized neighbors property requires that $u^C(t) = \infty$ for all $t \in \hat{T}$.

In the context of Algorithms 1 and 2, we set $\hat{S} := \{s' \mid s' \in \mathcal{R}[s], u^C(s') = 0\}$. Provided that u^C is the only dead-end detector used, it is easy to see that the u^C-recognized neighbors property always holds at the refinement step on $\mathcal{R}[s]$: $\mathcal{R}[s]$ contains only closed states, so it contains all states s' reachable from s, up to the neighbor states $t \in \mathcal{R}[s] \setminus \hat{S}$ where $u^C(t) = \infty$.

For illustration, consider again our running example, specifically the search space in Fig. 2. At the refinement step on s_3, we have $\hat{S} = \{s' \mid s' \in \mathcal{R}[s_3], u^C(s') = 0\} = \{s_3, s_1\}$. The neighbor states are $\hat{T} = \{s_4, s_5\}$. The u^C-recognized neighbors property is satisfied: each of the neighbor states is already recognized by u^C, using the singleton conjunctions only, as there is no more fuel left.

We use $u^C(s,G)$ to denote the u^C value of subgoal fact set G. Our refinement method is based on what we refer to as the u^C *neighbors information*: the values $u^C(t,c)$ for all $t \in \hat{T}$ and $c \in C$. We compute this information once, at the

Algorithm 4: Neighbors refinement on a state s. The initializing call to the algorithm is on $G := \mathcal{G}$. The algorithm assumes that $\hat{S}$ is a set of dead-end states, that $\hat{T}$ are its neighbors, and that the u^C-recognized neighbors property holds. It assumes that the u^C neighbors information (the values $u^C(t, c)$ for all $t \in \hat{T}$ and $c \in C$) is available. It identifies a set X of conjunctions so that $u^{C \cup X}(s) = \infty$ for all $s \in \hat{S}$. $\hat{S}$, $\hat{T}$, C, and X are global variables; $X := \emptyset$ is set initially.

```
Procedure NeighborsRefine(G)
    x := Extract(G);
    X := X ∪ {x};
    for a ∈ A[x] where ex. s ∈ Ŝ s.t. u^C(s, R(x, a)) = 0 do
        if there is no x' ∈ X s.t. x' ⊆ R(x, a) then
            NeighborsRefine(R(x, a));

Procedure Extract(G)
    x := ∅;
    /* Lemma 2 (i)                                                    */
    for every t ∈ T̂ do
        select c0 ∈ C s.t. c0 ⊆ G and u^C(t, c0) = ∞;
        x := x ∪ c0;
    /* Lemma 2 (ii)                                                   */
    for every s ∈ Ŝ do
        if x ⊆ s then
            select p ∈ G \ s; x := x ∪ {p};
    return x;
```

start of the refinement procedure.[4] Thanks to that information, in contrast to path-cut refinement as well as all previous conjunction learning methods, we do not require any intermediate recomputation of u^C during the refinement. Instead, neighbors refinement uses the u^C neighbors information to directly pick suitable conjunctions x for the desired set X. The method is inspired by the following simple characterizing condition for u^C dead-end recognition:

Lemma 1. *Let C be any set of atomic conjunctions, let s be a state, and let $G \subseteq \mathcal{F}$. Then $u^C(s, G) = \infty$ if and only if there exists $c \in C$ such that:*

(i) $c \subseteq G$ *and* $c \not\subseteq s$*; and*
(ii) *for every* $a \in \mathcal{A}[c]$, $u^C(s, R(c, a)) = \infty$.

Proof. "⇒": By definition of u^C, there must be a conjunction $c \in C$ so that $c \subseteq G$ and $u^C(s, c) = \infty$. The latter directly implies that $c \not\subseteq s$, and that $u^C(s, R(c, a)) = \infty$ for every $a \in \mathcal{A}[c]$.

"⇐": As $c \subseteq G$, we have $u^C(s, G) \geq u^C(s, c)$. As $c \not\subseteq s$, we have $u^C(s, c) = \min_{a \in \mathcal{A}[G]} u^C(s, R(G, a))$. For every $a \in \mathcal{A}[G]$, $u^C(s, R(c, a)) = \infty$, so we have $u^C(s, c) = \infty$ as desired. □

Say now that s is any state in $\hat{S}$. We need to find X so that $u^{C \cup X}(s) = u^{C \cup X}(s, \mathcal{G}) = \infty$. Given Lemma 1, we can do so by (i) picking some conjunction c with $c \subseteq \mathcal{G}$ and $c \not\subseteq s$, and then, recursively in the same manner, (ii) picking for every possible supporter $a \in \mathcal{A}[c]$ an unreachable conjunction c' for $R(c, a)$. As s is a dead-end, and as u^C recognizes all dead-ends in the limit, Lemma 1 tells us that a suitable conjunction c exists at every recursion level. But the lemma does not tell us what that conjunction is. In particular, c must actually be unreachable, i.e., it must hold that $h^*(s, c) = \infty$. But, given s and any one conjunction c, this is the same as asking whether a plan for c exists, which is **PSPACE**-complete to decide.

Now, we already know that the states $s \in \hat{S}$ are dead-ends. Therefore, we can in principle use the full subgoals as our conjunctions, i.e., in (i) we can use $c := \mathcal{G}$ because we know that $h^*(s, \mathcal{G}) = \infty$, and in (ii) we can use $c' := R(c, a)$ because we know that $h^*(s, R(c, a)) = \infty$. However, this naïve solution is pointless. It effectively constructs a full regression search tree from $\mathcal{G}$, selecting conjunctions corresponding to the regressed search states. For the method to be practically useful, what we need to find are *small* unreachable subgoals. It turns out that one can exploit the u^C-recognized neighbors property to that end.

Pseudo-code for our neighbors refinement procedure is shown in Algorithm 4. The purpose of a call to `NeighborsRefine(G)`, invoking the refinement on a target subgoal G, is to include conjunctions into X making G *unreachable from* $\hat{S}$ *under* $u^{C \cup X}$, i.e., so that $u^{C \cup X}(s, G) = \infty$ for all $s \in \hat{S}$. For this to be possible, of course G must be *unreachable from* $\hat{S}$, i.e., $h^*(s, G) = \infty$ for every $s \in \hat{S}$. That prerequisite is obviously true at the top-level call, where $G = \mathcal{G}$, because the states $s \in \hat{S}$ are dead-ends. As we shall see below, the prerequisite is invariant over calls to `Extract(G)`, i.e., the returned x also is

[4] Alternatively, one can cache the u^C neighbors information during search prior to the refinement on $\hat{S}$. But that turns out to be detrimental. Intuitively, as new conjunctions are continually added to C, the cached u^C information is "outdated". Using up-to-date C yields more effective learning.

unreachable from $\hat{S}$. As, for an unreachable subgoal, all regressed subgoals also are unreachable, the prerequisite thus holds at every invocation of NeighborsRefine(G).

But how to identify the conjunctions X? To this end, the top-level procedure of NeighborsRefine(G) mirrors the structure of Lemma 1. Following Lemma 1 (i), it starts by calling Extract(G), which identifies a subgoal $x \subseteq G$ unreachable from $\hat{S}$. Following Lemma 1 (ii), the procedure then finds conjunctions making x, and thus the target subgoal G which contains x, unreachable from $\hat{S}$ under $u^{C \cup X}$. To this end, the refinement is called recursively on the regressed subgoals $R(x,a)$ for the actions a supporting x.

More precisely, a recursive call is needed only for those supporters a not dealt with by the previous conjunctions C, i.e., those where, on some state $s \in \hat{S}$, the regressed subgoal $R(x,a)$ is actually reachable under u^C. Furthermore, such a supporting action has already been dealt with by the new conjunctions X in case there is some $x' \subseteq R(x,a)$. That is so because the conjunctions X are constructed so that, upon termination, they are unreachable from $\hat{S}$ under $u^{C \cup X}$. (We get back to the latter below.)

Consider now the Extract(G) sub-procedure, called on a target subgoal G. The procedure assumes that (a) G is unreachable from $\hat{S}$. It identifies a smaller subgoal $x \subseteq G$, giving the guarantees that (b) x is still unreachable from $\hat{S}$, and (c) x is unreachable from the neighbor states $\hat{T}$ under u^C. To be precise, (c) means that, for every $t \in \hat{T}$, there exists $c_0 \in C$ such that $c_0 \subseteq x$ and $u^C(t, c_0) = \infty$. To guarantee (c), the sub-procedure relies on the u^C neighbors information.

The first loop in the sub-procedure is over the neighbor states $t \in \hat{T}$. Drawing on the u^C neighbors information, it selects for each of these a prior atomic conjunction, $c_0 \in C$, justifying that $u^C(t, G) = \infty$. Such a c_0 must always exist: For the top-level goal $G = \mathcal{G}$, by construction we have that $u^C(t, \mathcal{G}) = \infty$, so by the definition of u^C there exists $c_0 \subseteq \mathcal{G}$ with $u^C(t, c_0) = \infty$. For later invocations of Extract(G), we have that $G = R(x,a)$, where x was constructed by a previous invocation of Extract(G). By property (c) of that previous invocation, there exists $c'_0 \in C$ such that $x \supseteq c'_0$ and $u^C(t, c'_0) = \infty$. But then, in particular we have that $u^C(t, x) = \infty$. Given this, we must also have that $u^C(t, R(x,a)) = \infty$, i.e., $u^C(t, G) = \infty$. But then, as desired there exists a conjunction $c_0 \subseteq G$ with $u^C(t, c_0) = \infty$.

The loop over neighbor states accumulates all the c_0 into x. Subsequently, the sub-procedure loops over the dead-end states s. In case the current x is contained in s, it adds a fact $p \in G \setminus s$ into x. Such a p necessarily exists due to the assumed property (a), G is unreachable from $\hat{S}$. (In practice, to keep x small, we use simple greedy strategies in Extract, trying to select c_0 and p shared by many t and s.)

Observe that the returned x satisfies property (c) by construction. It remains to prove that x also satisfies property (b), i.e., that x is, again, unreachable from $\hat{S}$:

Lemma 2. *Let C be any set of atomic conjunctions. Let $\hat{S}$ be a set of dead-end states, and let $\hat{T}$ be its neighbors with the u^C-recognized neighbors property. Let $x \subseteq \mathcal{F}$. If*

(i) *for every $t \in \hat{T}$, there exists $c_0 \in C$ such that $c_0 \subseteq x$ and $u^C(t, c_0) = \infty$; and*
(ii) *for every $s \in \hat{S}$, $x \not\subseteq s$;*

then $h^*(s, x) = \infty$ for every $s \in \hat{S}$.

Proof. Assume for contradiction that there is a state $s \in \hat{S}$ where $h^*(s, x) < \infty$. Then there exists a transition path $s = s_0 \to s_1 \cdots \to s_n$ from s to some state s_n with $x \subseteq s_n$. Let i be the largest index such that $s_i \in \hat{S}$. Such i exists because $s_0 = s \in \hat{S}$, and $i < n$ because otherwise we get a contradiction to (ii). But then, $s_{i+1} \notin \hat{S}$, and thus $s_{i+1} \in \hat{T}$ by definition. By (i), there exists $c_0 \subseteq x$ such that $u^C(s_{i+1}, c_0) = \infty$. This implies that $h^*(s_{i+1}, c_0) = \infty$, which implies that $h^*(s_{i+1}, x) = \infty$. The latter is in contradiction to the selection of the path. The claim follows. □

Altogether, Algorithm 4 is correct in the following sense:

Theorem 3. *Let C be any set of atomic conjunctions. Let $\hat{S}$ be a set of dead-end states, and let $\hat{T}$ be its neighbors with the u^C-recognized neighbors property. Then:*

(i) *The execution of* NeighborsRefine($\mathcal{G}$) *is well defined, i.e., it is always possible to extract an x as specified.*
(ii) *The execution of* NeighborsRefine($\mathcal{G}$) *terminates.*
(iii) *Upon termination of* NeighborsRefine($\mathcal{G}$), *$u^{C \cup X}(s) = \infty$ for every $s \in \hat{S}$.*

The arguments for (i) were outlined above. (ii) holds because every recursive call adds a new conjunction $x \notin X$: before the recursive call to NeighborsRefine($R(x,a)$) in the top-level procedure, there is no $x' \in X$ s.t. $x' \subseteq R(x,a)$; but that condition holds for the x' constructed in that recursive call. The number of possible conjunctions is finite, so the recursion must eventually terminate.

Finally, (iii) is a corollary of the aforementioned property that, upon termination, every $x \in X$ is unreachable from $\hat{S}$ under $u^{C \cup X}$. To see that the latter property holds, assume to the contrary that $u^{C \cup X}(s, x) = 0$. Then $h^{C \cup X}(s, x) = n$ for some finite number n. Let a be a best achiever of x under $h^{C \cup X}$, i.e., $h^{C \cup X}(s, R(x,a)) = n - 1$. By construction, in the recursive

call that included x into X, either an $x' \subseteq R(x,a)$ was already present in X, or such an x' was included in the recursive call on $R(x,a)$. But then, $h^{C \cup X}(s,x') \leq n-1$. Iterating this argument, we obtain a conjunction $x_0 \in X$ where $h^{C \cup X}(s,x_0) = 0$, i.e., $x_0 \subseteq s$. Such x_0 are never included into X by construction, in contradiction, concluding the argument.

In short, all input states $s \in \hat{S}$ are refuted by $u^{C \cup X}$ upon termination, because `NeighborsRefine`($\mathcal{G}$) finds a conjunction refuting G itself (Lemma 1 (i)), then makes sure to find conjunctions refuting the regressed subgoal for every supporting action (Lemma 1 (ii)). A detailed proof of Theorem 3 is available in Appendix A.

Example 4. Consider again the search space of our running example in Fig. 2, and the refinement process on the states $\hat{S} = \mathcal{R}[s_3] = \{s_3, s_1\}$, where the truck has driven to l_1 and has a single fuel unit left. The recognized neighbor states are $\hat{T} = \{s_4, s_5\}$ where the truck has driven back to l_2 and has no fuel left.

We initialize $X = \emptyset$ and call `NeighborsRefine`($\{p_1(l_3), p_2(l_2)\}$). Consider the call to the sub-procedure, `Extract`($\{p_1(l_3), p_2(l_2)\}$). Here, we find that $c_0 = \{p_1(l_3)\}$ is suitable for each of s_4 and s_5: it is unreachable under u^C because there is no fuel left. Furthermore, $c_0 = \{p_1(l_3)\}$ is neither contained in s_3 nor in s_1. So we return $x = \{p_1(l_3)\}$. Note that this is not a new conjunction; it is already contained in C. The x extracted is guaranteed to not already be in X, but it may be an element of C. In other words, as the x in each recursive call, we may use a conjunction that was already atomic beforehand.

Back in `NeighborsRefine`($\{p_1(l_3), p_2(l_2)\}$), we see that x can be achieved by unloading at l_3. We need to tackle the regressed subgoal, through the recursive call `NeighborsRefine`($\{t(l_3), p_1(T)\}$). In that call, the extraction sub-procedure returns $x = \{t(l_3)\}$, which is suitable for the same reasons as above. Observe that, again, this conjunction was already atomic beforehand. We next need to exclude the drive action from l_2 to l_3, via the recursive call `NeighborsRefine`($\{t(l_2), f(1)\}$).

Consider finally the extraction sub-procedure in that recursive call, i.e., the call to `Extract`($\{t(l_2), f(1)\}$). For the $\hat{T}$ neighbor states, s_4 and s_5, where the truck is at l_2 but there is no more fuel, only the subgoal fact $f(1)$ is eligible. That is, we get $c_0 = \{f(1)\}$ for each of them. However, $f(1)$ is contained in the $\hat{S}$ states s_3 and s_4. So we need to add also the other subgoal fact into x, ending up with $x = G = \{t(l_2), f(1)\}$. Observe that this last x is the first non-atomic x extracted.

The refinement process stops here, because the actions achieving x, driving to l_2 from l_1, and driving to l_2 from l_3, both incur the regressed subgoal $f(2)$, for which we have $u^C(s_3, \{f(2)\}) = u^C(s_1, \{f(2)\}) = \infty$. Hence the set X returned contains, like for path-cut refinement above, just the one new conjunction $\{t(l_2), f(1)\}$, which is exactly what is needed to render $u^{C \cup X}(s_3) = u^{C \cup X}(s_1) = \infty$.

6. Clause learning

As previously discussed, the computation of u^C is low-order polynomial time in the number $|C|$ of atomic conjunctions, through simple dynamic programming techniques. Yet in practice it may incur a substantial runtime overhead for large C. We now introduce a *clause learning* technique to alleviate this. Essentially, we learn weaker nogoods in addition to u^C, that are easier to evaluate than u^C and serve as a filter in front of u^C.

The clause learning method is technically quite simple. It follows an earlier proposal by Kolobov et al. [19], in their SixthSense dead-end detection method for forward search probabilistic planning, where essentially the same technique was used for verifying and minimizing nogood candidates (the candidates having been previously derived from information obtained in a determinization, i.e., via classical planning).

Consider any state s where $u^C(s) = \infty$. Denote by $\phi := \bigvee_{p \in \mathcal{F} \setminus s} p$ the disjunction of facts false in s. Then ϕ is a *valid clause*: for any state s', if s' does not satisfy ϕ, written $s' \not\models \phi$ as usual, then $u^C(s') = \infty$; in particular, if s' does not satisfy ϕ then it is a dead-end. To see this, just note that, as $u^C(s) = \infty$, the goal is unreachable from s under u^C. To make the goal reachable under u^C, we need to make true at least one of the facts that are false in s.

The valid clause ϕ just defined is, per se, not useful as it generalizes to only those states subsumed by s, i.e., whose true facts are contained in those of s. This changes when *minimizing* ϕ, testing whether individual facts p can be removed while preserving validity. In other words, we aim at obtaining a minimal *reason* for $u^C(s) = \infty$ (similarly as done for failed "obligations" in property-directed reachability [14]). Our minimization method is straightforward, testing the facts $p \in \mathcal{F} \setminus s$ one by one.

We start with $s' := s$. We then loop over all $p \in \mathcal{F} \setminus s$. In each loop iteration, we test whether $u^C(s' \cup \{p\}) = \infty$; if so, we set $s' := s' \cup \{p\}$. Upon termination of the loop, s' is a set-inclusion maximal superset of s that preserves goal unreachability under u^C, i.e., where $u^C(s') = \infty$. We then obtain our clause as the disjunction of the remaining facts, $\phi := \bigvee_{p \in \mathcal{F} \setminus s'} p$. SixthSense does the same except for using h^2, rather than u^C.

Example 5. Consider again our running example, and the clause learning on state s_4 from Fig. 2, where the truck has moved from l_2 to l_1 and back to l_2, so that the packages are still in their initial locations but there is no more fuel left.

The minimization loop starts with $s' = s = \{t(l_2), f(0), p_1(l_1), p_2(l_3)\}$. Adding the other possible locations of p_2, i.e. the facts $p_2(l_1)$ and $p_2(l_2)$, the goal is still unreachable under u^C because we cannot achieve the goal $p_1(l_3)$. So we set $s' := s' \cup \{p_2(l_1), p_2(l_2)\}$. Considering now the other possible locations of p_1, i.e. the facts $p_1(l_2)$ and $p_1(l_3)$, $p_1(l_2)$ can be added as we still cannot reach $p_1(l_3)$. But $p_1(l_3)$ cannot be added as we would then have $\mathcal{G} \subseteq s'$. So we set $s' := s' \cup \{p_2(l_2)\}$. If we add any amount of fuel, the goal becomes reachable under u^C again (with singleton conjunctions only, fuel consumption is ignored), so neither $f(1)$ nor $f(2)$ can be added. Finally, considering the other possible locations of the truck, $t(l_1)$ and

$t(l_3)$, we can add $t(l_1)$ as the truck still cannot reach l_3. But we cannot add $t(l_3)$ as, then, in the extended s' the truck would be at l_2 and l_3 simultaneously, and able to transport p_1 from l_2 to l_3 without fuel consumption.

We end up with $s' = \{t(l_1), t(l_2), f(0), p_1(l_1), p_1(l_2), p_2(l_1), p_2(l_2), p_2(l_3)\}$. This yields the clause $\phi = t(l_3) \vee f(1) \vee f(2) \vee p_1(l_3)$ as previously advertised in our example walkthrough (Section 3).

Note that the clause we end up with depends on the ordering of facts in the minimization loop. If, for example, we test $t(l_3)$ at the very beginning of the loop, then it can be added: with p_1 being only at l_1, having t at both l_2 and l_3 still requires fuel to achieve $p_1(l_3)$. On the other hand, if we do add $t(l_3)$ into s', then later on we cannot add $p_1(l_2)$, nor $t(l_1)$. We use an arbitrary fact ordering in our implementation, i.e., we do not attempt to find clever orderings.

The recomputation of $u^C(s' \cup \{p\})$ for each fact p in the minimization loop can be optimized by doing it in an incremental manner. Omitting implementation details, we essentially store the dynamic programming table (an index from atomic conjunctions into $\{0, \infty\}$) of the previous iteration, identify the table cells changing from ∞ to 0 due to the inclusion of p, and propagate these changes. Nevertheless, the minimization sometimes incurs a significant computational overhead. To reduce that overhead, in our implementation we slightly diverge from the above. We test, in each iteration of the minimization loop, not individual facts p, but entire "state variables" v in the internal representation of the Fast Downward planning system [79,80], which our implementation is based on. A state variable v here corresponds to a set $\mathcal{F}(v)$ of facts exactly one of which is true in every state. In each loop iteration, we test whether the entire $\mathcal{F}(v)$ for some v can be included into s'. This yields weaker clauses, but takes less runtime.

During search, we maintain a conjunction Φ of clauses, starting with empty Φ and adding ϕ to Φ whenever a new clause ϕ is learned. Clearly, whenever $s' \not\models \Phi$, then $u^C(s') = \infty$. So Φ is a weaker dead-end detector than u^C; yet it can be evaluated more effectively. Our implementation does so through a counter-based scheme, where we loop over Fast Downward's state variables v just once, incrementing a counter for every clause ϕ that v's value in s' does not comply with. Then $s' \not\models \Phi$ iff one of these counters reaches the total number of state variables.

The question remains how to arrange the search: (1) When exactly to learn clauses, (2) when exactly to test them, and (3) when to make a full call to the unsolvability heuristic u^C? Regarding (1), a clear outcome from our preliminary experiments is that a clause should be learned every time we evaluate u^C on a state s and find that $u^C(s) = \infty$. Regarding (2) and (3), the obvious arrangement is to use the clauses as a filter, i.e., whenever a state s is scheduled for evaluation by u^C, first test whether $s \not\models \Phi$, and if so, return ∞ without invoking u^C.

Observe that, in particular, while we get duplicate pruning "for free" from the refutation knowledge accumulated in u^C (cf. property (6) in Section 4.2), in practice, computing u^C may be a lot more time-consuming than duplicate checking. That is dealt with by the clauses, which also subsume duplicate pruning and are similarly cheap to evaluate.

7. Experiments

We evaluate our techniques for three different purposes: finding plans in solvable planning tasks with dead-ends, proving unsolvability, generating unsolvability certificates. We next detail our experiment setup, then cover these different purposes in this order.

7.1. Experiment setup: algorithms and benchmarks

Variants of our technique We implemented all the described techniques in Fast Downward [79], short *FD*. In our preliminary experiments, we found that search algorithms other than depth-first search hardly ever benefited from conflict-driven learning, as they did not identify enough conflicts. This pertains especially to A*, which explores the search space in a breadth-oriented fashion, considering many options. In contrast, for the identification of conflicts, it is beneficial to search deeply not broadly, pushing the state at hand to either the goal or a dead-end situation.

Given this, we consider depth-first searches only. In particular, for solvable planning tasks, we consider *satisficing planning*, not giving a plan quality guarantee. We focus on DFS, i.e., our extension of Tarjan's algorithm (cf. Section 4.2), our most elegant and effective search algorithm. For ordering children nodes in DFS, we focus on an ordering by smaller value of the delete relaxation heuristic h^{FF} [41], which turns out to be beneficial for both, finding plans and (to a lesser degree) proving unsolvability.

We experiment with three different conjunction learning methods, namely path-cut refinement and neighbors refinement as introduced here, plus Keyder et al.'s [69,67] as a representative of prior conjunction-learning methods. Keyder et al.'s method, like Haslum's preceding one [66], is based on iterative removal of conflicts in a delete-relaxed plan, and we will refer to it as *conflict refinement*. We also run DFS without any refinement, as a direct comparison pointing out the impact of learning vs. an identical search without learning.

For finding plans and for proving unsolvability, we run DFS with early termination, stopping search as soon as there are no more open nodes. For generating unsolvability certificates, we run DFS without early termination, refining u^C until it refutes the initial state.

We also run offline learning variants, for proving unsolvability and for dead-end detection. In either case, we use path-cut refinement and conflict refinement, as neighbors refinement is not applicable in the offline context. For proving unsolvability, we simply refine u^C on the initial state until $u^C(I) = \infty$. For dead-end detection, we refine u^C on the initial state until a

size bound N is reached. Then we use the same set C for u^C dead-end detection throughout the search. This is inspired by previous work on partial delete relaxation heuristics [69,67,68], and like that work the size bound N is multiplicative: we stop when the number of "counters" – atomic entities in the implementation of u^C – reaches N times that when C contains the singleton conjunctions only.

To evaluate the complementarity of our method vs. strong competing methods, we design simple combinations with the two strongest alternate dead-end detection techniques, namely unsolvability heuristics u obtained from merge-and-shrink abstractions respectively potential heuristics (more on these below). The combination uses both dead-end detectors every time we test whether a state s is a dead-end. The only subtlety here is during refinement at s, where, because of the additional dead-end detector u, as already mentioned the u^C-recognized neighbors property may not hold. Distinguishing the two possible cases, (a) if s has u^C-recognized neighbors, then we use neighbors refinement which generally works best if applicable. Case (b), if s does not have u^C-recognized neighbors, then we can use path-cut refinement. Observe though that the latter will force u^C to recognize all the dead-ends below s already recognized by u. Therefore, we experiment with two combination methods, one using both (a) and (b), and one using only (b).

For ablation study purposes, we finally run variants of our strongest configuration (DFS with neighbors refinement), with vs. without the clause learning mechanism, and using depth-oriented search (open list preferring deepest states), short *DOS*, instead of Tarjan's algorithm.

State-of-the-art competing algorithms Apart from standard search algorithms and heuristic functions, the relevant techniques in our context are (a) dead-end detection, (b) admissible pruning techniques, and (c) other methods for proving unsolvability. For all of these, the state of the art at the time of writing is represented by the participants of the 2016 inaugural *Unsolvability International Planning Competition (UIPC'16)*. To provide a comprehensive picture, we include all UIPC'16 participants, save those vastly dominated in that competition. However, our interest is in understanding algorithm behavior, as opposed to running a systems competition. For systems composed of several distinct algorithm components, we therefore consider, not the systems, but their components. This pertains primarily to the winning system Aidos [53], an algorithm portfolio; for the other UIPC'16 participants, our modifications are minor.

In detail, we run the following algorithms. Throughout, we use the original planning task representation produced by the FD translator [80], without additional preprocessors. From category (a), we run the following algorithms. We run *merge-and-shrink (M&S)* abstractions, in the two most competitive unsolvability-heuristic variants of Hoffmann et al. [36]. One of the two M&S variants computes the perfect unsolvability heuristic u^*, the other imposes an abstraction size limit and yields an approximate dead-end detector. Very similar M&S heuristics were used in UIPC'16 [49], in combination with additional irrelevance pruning [63] and dominance pruning [62]; we separate out the latter components to keep things clean. We furthermore run the new *pattern database* unsolvability heuristic [50], using a more restricted class of abstractions based on projection (e.g. [42]), which participated in UIPC'16 on its own and as one component of Aidos. We run *potential heuristics* [51,52] for dead-end detection, another component of Aidos, and the one that contributed most to its overall performance. These potential heuristics use an LP solver to find an invariant proving that the goal is unreachable (because the current state's potential cannot be reduced to that of the goal states).

From category (b), we run *simulation-based dominance* pruning [62], used in the UIPC'16 M&S system [49], which finds a simulation relation over states and prunes dominated states during search. We run *strong stubborn sets (SSS)* pruning, used in two UIPC'16 entries [81,53], a long-standing partial-order reduction method (e.g. [82,61]) exploiting permutability. We run *irrelevance pruning* [63], also used in two UIPC'16 entries [57,49], which detects irrelevant operators (that cannot be part of an optimal solution) based on dominance analysis in a merge-and-shrink abstraction.

From category (c), we run *BDD-based symbolic search* (e.g. [83,84,63]), specifically the UIPC'16 Sympa system [57], separating out the irrelevance pruning (same as for M&S heuristics above). We run *property-directed reachability (PDR)* [13,14] as in UIPC'16 [85]. We run *resource-variable detection*, another component of Aidos, which performs domain analysis to identify a state variable encoding a consumed resource-budget, and which during search uses a *cartesian abstraction* [86,87] lower bound to prune against the remaining budget. We run *star-topology decoupled state space search* [88,89], a decomposition technique exploiting possible factorizations into star topologies. We separate the standard state space search and strong stubborn sets pruning components used as alternatives to star-topology decoupled state space search in the UIPC'16 system. We run partial delete-relaxation via *red–black planning* [58,60], in its most competitive configuration established after UIPC'16 [59]. This approach searches in a relaxation where "red" state variables (but not "black" ones) are delete-relaxed. If there is no relaxed plan, there cannot be a real plan either, so unsolvability of the input task can be proved this way. The relaxation is iteratively strengthened by painting one more variable black. The only difference between our version and the UIPC'16 one is the variable order in which that is done.

We do *not* run the UIPC'16 theorem proving approach [90], as this was vastly outperformed by the other competition entries. We do not run Haslum's UIPC'16 entry [91], which also performed badly, and is very similar to our configuration doing offline learning with conflict refinement. The main difference is that it uses a less effective representation, where h^C is computed in a compiled planning task Π^C whose size is worst-case exponential in $|C|$.

Benchmarks For unsolvable benchmarks, there is exactly one standard benchmark set at the time of writing, namely that of UIPC'16, which we use.

As solvable benchmarks, naturally we use the benchmark suites of the International Planning Competition (IPC). We consider all IPC editions, 1998–2014, specifically the STRIPS benchmarks for satisficing planning (where these distinctions are made). Yet more specifically, our learning techniques are interesting only in domains that actually contain conflicts, i.e., dead-ends unrecognized under h^1. Therefore, from IPC'98–IPC'08 we use the subset of domains, as determined in Hoffmann's [72,92] analyses, where that is the case. From IPC'11 and IPC'14, where a formal analysis has not yet been carried out, we use those domains where DFS with h^1 dead-end detection (with either of the two children ordering methods) identifies at least one conflict, i.e., where DFS backtracks out of a strongly connected component at least once on at least one instance.

In addition to the competition benchmarks just described, we also consider planning with resources (e.g. [93,94,27–29]), specifically so-called *resource-constrained planning* [28] where the goal must be achieved subject to a limited resource budget. We use the benchmarks by Nakhost et al. [28], which are *controlled* in that the minimum required budget $b_{\min}$ is known, and the actual budget is set to $W * b_{\min}$, where the *constrainedness level* W is a benchmark instance parameter. For constrainedness levels W much larger than 1, resources are plenty and a plan is typically easy to find; for constrainedness levels W much smaller than 1, resources are extremely scarce and unsolvability is typically easy to prove; for constrainedness levels W close to 1, resource-constrained planning is notoriously difficult. Intuitively, the constrainedness level controls the frequency and detection difficulty of dead-ends, which is of obvious interest to our purposes here. For the solvable case, we use the exact benchmark suites provided by Nakhost et al. For the unsolvable case, we use those same benchmarks but with $W \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$, not considered by Nakhost et al. (but previously considered by Hoffmann et al. [36]).

All experiments were run on a cluster of Intel E5-2660 machines running at 2.20 GHz, with runtime (memory) limits of 30 minutes (4 GB).

7.2. Dead-end detection in solvable planning tasks

We consider first the solvable case. We run the six variants of our technique described above: DFS without learning; DFS with learning using one of the three refinement methods; and the DFS combinations with dead-end detectors from UIPC'16, M&S respectively potential heuristics. We run these algorithms with h^{FF} children ordering, and compare them to DFS with arbitrary children ordering. We furthermore compare them with search algorithms using heuristic search as the baseline, specifically FD's lazy-greedy best-first search (GBFS) using h^{FF} with a dual open queue for preferred operators [79]. This is the canonical baseline algorithm for satisficing heuristic search planning, and yields competitive performance while being reasonably simple. We use M&S respectively potential heuristics for dead-end detection in that baseline search. We use simulation-based dominance, strong stubborn sets (SSS), respectively irrelevance pruning to prune the state space in that baseline search.

We finally run offline learning with a size bound x to generate static u^C dead-end detectors. We use these in both, DFS without learning for direct comparison to our techniques, and in the GBFS baseline search for direct comparison to the other static dead-end detectors. We experiment with $N = 2$ as that was the best size bound in prior work on partial delete relaxation heuristics, and we experiment with $N = 32$ as a larger yet still reasonable setting.

The main base algorithm our learning methods start from, DFS with h^{FF} children ordering, is quite different from the GBFS h^{FF} baseline. So we start with a brief experiment comparing these two, without learning. Consider Table 1. We see that the two base algorithms have complementary strengths in different domains. Sometimes the differences are drastic, most notably in Sokoban, Woodworking, as well as resource-constrained Rovers and TPP where DFS is much stronger; and Mprime, Mystery, NoMystery, and Thoughtful where GBFS is much stronger. In total, these differences cancel each other out though, and the two algorithms are on a similar level. This is remarkable in itself, seeing as GBFS is widely used in heuristic search planning while, to the authors' knowledge, DFS has not been used at all in this context yet. For our purposes here though, the main conclusion is that, overall, performance is comparable so we are not a priori much disadvantaging either side.

Table 1 also includes a variant of DFS disallowing backtracking. This serves to point out the cases where, with h^{FF} tie breaking, despite the presence of conflicts in principle, no learning will happen simply because a goal state is found without ever encountering a conflict. As the table shows, this happens to a surprising extent. In particular, this simplistic search procedure is an extremely effective solver for ParcPrinter, Pathways, and TPP, where it solves all instances solved by the common baseline (almost all, in case of Pathways), but within split seconds or a few seconds at most. We conclude from this that someone should put this algorithm into their portfolio at the next planning competition. That aside, we continue with our own research focus here, which is on those benchmarks where learning does indeed happen.

Table 2 shows coverage data. We will consider offline learning separately below. For the resource-constrained domains, as the constrainedness level has a large impact on performance for most algorithms, we show data for each level separately. Note here that IPC Mprime, Mystery, and NoMystery also are resource-constrained domains. However, their constrainedness levels are not known or only partially known, so we do not separate these. (We will however do so for the unsolvable resource-constrained benchmarks from UIPC'16 below.)

Consider first the different refinement variants within DFS h^{FF}, i.e., our neighbors refinement ("Nei") and path-cut refinement ("Pat") methods vs. the conflict refinement ("Con") from prior work. There are large performance differences in many domains, showing that the way we learn conjunctions is important. In particular, conflict refinement has the lead, and a marginal one at that, in only one case (IPC Trucks), showing that it is important to target the conjunction learning to

Table 1
Coverage results on solvable benchmarks, comparing the two base algorithms, as well as DFS without backtracking, counting as solved only those tasks where the first DFS search branch finds a goal state. Best results highlighted in **boldface**. Abbreviations: "GBFS" FD greedy best-first search baseline using h^{FF} and preferred operators; "DFS" depth-first search using Tarjan's algorithm for conflict identification, with children nodes ordered using h^{FF}.

Domain	#	GBFS h^{FF}	DFS h^{FF}	DFS h^{FF} w/o backtracking	
		Coverage	Coverage	Coverage	Average runtime
IPC Benchmarks					
Airport	50	30	**34**	14	.33
Childsnack	20	**1**	0	0	
Floortile	40	**10**	8	0	
Freecell	80	77	**78**	63	2.65
Mprime	35	**35**	27	26	2.14
Mystery	30	**20**	16	13	.37
NoMprime	35	**34**	25	23	1.56
NoMystery	50	**33**	22	15	.30
Openstacks	100	100	100	100	98.28
ParcPrinter	50	50	50	50	.04
Pathways	60	**46**	**46**	44	1.00
PegSol	50	**50**	**50**	3	.01
Pipesworld-Tankage	50	**40**	37	35	78.15
Sokoban	50	32	**40**	2	.11
Thoughtful	20	**13**	9	9	1.65
Tidybot	20	13	13	13	60.05
TPP	30	30	30	30	2.88
Trucks	30	**18**	17	5	.01
Woodworking	50	8	**40**	**40**	257.94
$\sum$	850	640	**642**	485	
Nakhost et al. [28] Benchmarks					
NoMystery	210	**63**	27	7	.17
Rovers	210	1	**8**	2	.38
TPP	30	5	**13**	3	2.71
$\sum$	450	**69**	48	12	
$\sum$ Total	1300	**709**	690	497	

dead-end detection. Path-cut refinement is best overall on the IPC, but neighbors refinement has a huge advantage in the resource-constrained benchmarks so has best overall coverage.

Consider now the effect of learning vs. no learning ("No"). On the resource-constrained domains, the improvement is consistently dramatic, with some minor exceptions in TPP. On the IPC benchmarks, the picture is much more mixed. On Airport, Freecell, PegSol, Pipesworld-Tankage, Sokoban, and Trucks, the learning has a detrimental effect. We will analyze in some detail below why that is so. On Tidybot and Woodworking, as well as ParcPrinter, Pathways, and TPP for the DFS h^{FF} variants, the learning has no impact at all. That is mostly because DFS does not identify many conflicts here, so the learning is seldom, or never, invoked (we also get back to this in more detail below). On the other domains, improvements are possible. These are most pronounced in Floortile for path-cut and conflict refinement; in Mystery, NoMprime, and NoMystery for neighbors refinement; and in ParcPrinter and Pathways for neighbors refinement in DFS without h^{FF} children ordering.

Overall, compared to the baselines without learning (including also GBFS), on the IPC the learning is detrimental, as the size of its losses outweighs that of its gains. On the resource-constrained domains, the picture is very different, with a substantial and consistent win over the baselines. The only exception is NoMystery with $W = 2.0$, where fuel is relatively plentiful and the baseline does not struggle with unrecognized dead-ends as much as it does with constrainedness levels closer to 1.0.

Ordering children nodes in DFS by h^{FF} ("DFS h^{FF}") outperforms arbitrary children ordering ("DFS") dramatically overall, and almost consistently across domains. Therefore, from now on, we consider h^{FF} children ordering exclusively. We will do so not only for the solvable case, but also for unsolvable benchmarks, where h^{FF} children ordering does not have as large an impact, but is never worse and sometimes helps (intuitively because DFS is drawn towards more relevant dead-end situations, learning more relevant knowledge). Henceforth, whenever we say "DFS" we mean "DFS with h^{FF} children ordering".

Consider next the UIPC'16 algorithms. On the IPC benchmarks, compared to their GBFS baseline, only strong stubborn sets pruning ("SSS") improves overall coverage, and only SSS and irrelevance pruning ("Irr") have higher overall coverage than our methods (DFS h^{FF} with neighbors refinement respectively path-cut refinement). The strengths of these two methods lie in being less risky, not deteriorating the baseline as much, especially in Openstacks and Pipesworld-Tankage; and in the improvements they yield in Airport, Sokoban, and (for irrelevance pruning) Woodworking. The strong cases for our methods are Floortile where DFS with path-cut refinement, and also with conflict refinement, vastly outperforms all competitors;

Table 2
Coverage results on solvable benchmarks, comparison to the state of the art. Best results highlighted in **boldface**. Abbreviations: "Base" baseline; "GBFS" FD's lazy-greedy dual-queue best-first search; "DFS" depth-first search using Tarjan's algorithm for conflict identification; "DFS h^{FF}" DFS ordering children by h^{FF} value; "DFS h^{FF} comb." combination (N/NP, see below) with another dead-end detector; "#" number of instances; "No" no learning; "Nei" neighbors refinement; "Pat" path-cut refinement; "Con" conflict refinement; "N" combination using neighbors refinement only; "NP" combination using neighbors refinement if applicable, else path-cut refinement; "MSa" approximate merge-and-shrink abstraction; "Pot" potential heuristic; "Sim" simulation dominance pruning; "SSS" strong stubborn sets pruning; "Irr" irrelevance pruning.

Domain (W)	#	Base	DFS		DFS h^{FF}				DFS h^{FF} comb.				UIPC'16: Base +				
		GBFS							MSa		Pot		Detection		Pruning		
		h^{FF}	No	Nei	No	Nei	Pat	Con	N	NP	N	NP	MSa	Pot	Sim	SSS	Irr
IPC Benchmarks																	
Airport	50	30	**35**	30	34	24	21	21	17	17	24	23	18	38	12	**35**	34
Childsnack	20	1	0	0	0	1	1	1	1	1	0	0	0	0	3	0	**4**
Floortile	40	10	6	0	8	4	**37**	27	4	4	4	4	15	15	8	12	13
Freecell	80	77	78	72	78	71	72	69	50	50	71	71	51	76	51	77	**80**
Mprime	35	**35**	18	19	27	28	27	27	17	17	27	30	18	30	25	30	34
Mystery	30	20	14	20	16	26	23	23	20	20	24	26	19	**28**	15	25	18
NoMprime	35	34	17	19	25	29	27	26	11	11	27	26	11	**34**	18	34	29
NoMystery	50	33	18	29	22	**38**	36	32	28	32	30	33	28	33	35	33	32
Openstacks	100	**100**	**100**	**100**	**100**	98	**100**	**100**	32	29	66	49	30	66	50	90	58
ParcPrinter	50	**50**	25	38	**50**	**50**	**50**	**50**	**50**	10	**50**	10	**50**	**50**	**50**	**50**	**50**
Pathways	60	**46**	8	10	**46**	**46**	**46**	**46**	**46**	**46**	**46**	**46**	**46**	**46**	47	44	44
PegSol	50	**50**	**50**	41	**50**	41	30	12	41	41	48	23	**50**	**50**	**50**	**50**	**50**
Pipesworld-Tankage	50	**40**	17	14	37	35	35	34	8	8	17	17	8	16	16	**40**	30
Sokoban	50	32	42	11	40	9	6	3	10	8	9	9	42	35	5	42	**48**
Thoughtful	20	13	5	1	9	9	9	9	5	5	**10**	7	5	7	8	7	8
Tidybot	20	**13**	8	8	**13**	**13**	**13**	**13**	0	0	**13**	**13**	0	12	0	12	0
TPP	30	**30**	24	23	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	20	**30**	25
Trucks	30	**18**	11	6	17	9	16	17	10	10	9	9	**18**	16	**18**	16	**18**
Woodworking	50	8	6	6	40	40	40	40	13	13	28	28	8	7	49	22	**50**
$\sum$ (IPC)	850	640	482	447	642	601	619	580	393	352	533	454	447	589	480	**649**	625
Nakhost et al. [28] Benchmarks																	
NoMystery(1.0)	30	2	0	10	1	12	7	4	19	7	3	7	20	0	**26**	1	8
NoMystery(1.1)	30	3	0	10	0	12	8	5	17	8	3	8	**20**	1	**20**	2	9
NoMystery(1.2)	30	5	0	7	4	14	13	9	18	10	7	10	21	1	**26**	2	16
NoMystery(1.3)	30	6	0	7	5	13	9	7	15	11	5	7	23	3	**27**	3	25
NoMystery(1.4)	30	12	0	5	3	17	13	9	11	16	8	10	24	5	**29**	10	25
NoMystery(1.5)	30	12	0	6	5	17	13	9	10	15	6	10	24	5	**29**	9	**29**
NoMystery(2.0)	30	23	0	8	9	19	18	14	12	17	11	15	28	7	**30**	16	**30**
$\sum$	210	63	0	53	27	104	81	57	102	84	43	67	160	22	**187**	43	142
Rovers(1.0)	30	0	0	19	0	**23**	9	8	7	17	**23**	**23**	6	0	15	0	0
Rovers(1.1)	30	0	0	16	0	**23**	12	13	7	19	22	21	5	0	15	0	2
Rovers(1.2)	30	0	0	19	0	**18**	7	9	5	19	**18**	**18**	1	0	14	0	2
Rovers(1.3)	30	0	0	17	0	**20**	7	6	8	17	**20**	**20**	2	0	16	0	1
Rovers(1.4)	30	0	0	19	1	**19**	8	7	8	18	**19**	**19**	4	0	16	0	3
Rovers(1.5)	30	0	0	19	1	**21**	12	8	10	19	**21**	**21**	5	0	15	0	6
Rovers(2.0)	30	1	0	17	6	21	16	15	12	18	21	21	11	0	**23**	0	10
$\sum$	210	1	0	126	8	**145**	71	66	57	127	144	143	34	0	114	0	24
TPP(1.0)	5	0	0	0	1	1	0	0	1	1	1	0	0	0	**2**	0	1
TPP(1.1)	5	0	0	0	0	2	0	0	1	2	0	0	1	0	**3**	0	3
TPP(1.2)	5	0	0	0	2	**3**	**3**	**3**	**3**	**3**	0	0	2	0	**3**	0	**3**
TPP(1.3)	5	2	0	0	2	**5**	3	3	4	4	0	0	4	0	4	2	4
TPP(1.4)	5	3	0	0	3	**5**	4	**5**	**5**	**5**	0	0	4	0	4	2	4
TPP(1.5)	5	0	0	0	**5**	**5**	**5**	**5**	**5**	**5**	0	0	**5**	0	**5**	0	2
$\sum$	30	5	0	0	13	**21**	15	16	19	20	1	0	16	0	**21**	4	17
$\sum$ Nakhost et al.	450	69	0	179	48	270	167	139	178	231	188	210	210	22	**322**	47	183
$\sum$ Total	1300	709	482	626	690	**871**	786	719	571	583	721	664	657	611	802	696	808

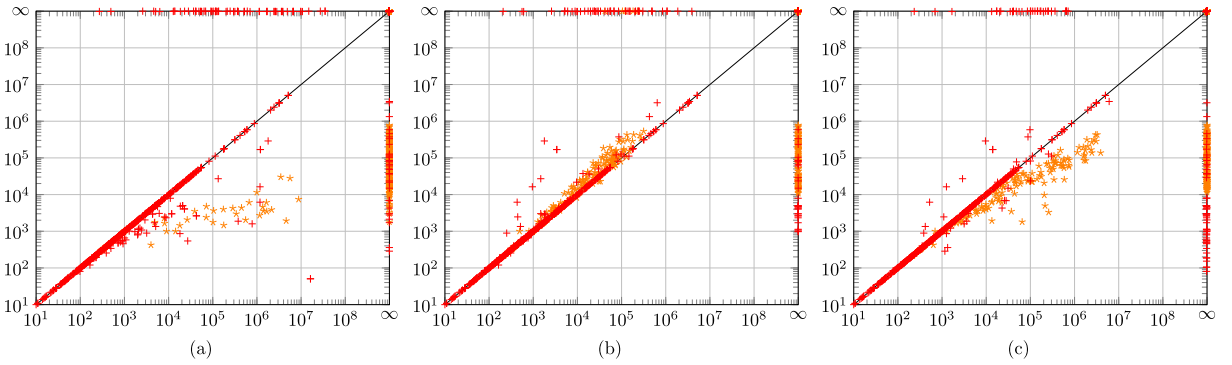

Fig. 3. DFS search space size (number of states visited) on solvable benchmarks, with different learning techniques. We compare neighbors refinement on the y-axis to (a) no learning, (b) path-cut refinement, respectively (c) conflict refinement on the x-axis. Unsolved benchmark instances are shown as ∞. Data points for IPC instances are shown in red, those for resource-constrained instances are shown in orange. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

NoMystery where DFS with neighbors refinement performs best; Openstacks, Tidybot, and TPP, where the learning methods incur less overhead than many of the UIPC'16 methods; as well as Woodworking where the DFS baseline is much better than the GBFS baseline, and there is no learning overhead.

On the resource-constrained benchmarks, the picture is again much clearer. Potential heuristics ("Pot") and SSS are basically useless. Irrelevance pruning, merge-and-shrink ("MSa"), and especially simulation-based dominance pruning ("Sim") excel in NoMystery. In the other two domains, our learning methods tend to be superior. In Rovers, DFS with neighbors refinement ("Nei") vastly outperforms all UIPC'16 competitors. Similarly in TPP, except for simulation-based dominance which does equally well in coverage (though typically a lot worse in runtime, on commonly solved instances). In terms of overall coverage, DFS with neighbors refinement is the clear winner. Other algorithms perform better in one part (IPC vs. Nakhost et al.) of the benchmark set, but DFS with neighbors refinement is most consistently good overall.

Consider finally the combinations of neighbors-refinement DFS with merge-and-shrink respectively potential heuristics. For merge-and-shrink, this basically does not work well here. The DFS h^{FF} component dominates the combined methods consistently, the only notable exception being NoMystery with small values of W where the "N" combination does better. Yet merge-and-shrink alone does still a lot better there so this is not a valuable result. There is no case where a combination outperforms both its components. (As we shall see below, matters are quite different on the unsolvable benchmarks.)

For the combination with potential heuristics, the picture is similar, though not quite as bleak. The "N" combination outperforms the DFS h^{FF} component on PegSol, where the potential heuristic prevents most (but not all) of the loss compared to the DFS baseline. The "N" combination does better than both of its components on Thoughtful (though by only 1 instance relative to the DFS component).

Fig. 3 provides a view on the search space sizes under our different DFS learning techniques, i.e., no learning vs. the three different refinement methods. We use neighbors refinement, the strongest method, as the comparison baseline.

Consider first the comparison to no learning, Fig. 3(a). On the IPC benchmarks, in line with the above, we see that the learning is risky, improving performance in some cases but deteriorating it in many others. On the resource-constrained benchmarks, on the other hand, the benefits of learning are dramatic. Most benchmark instances are not solved at all without learning. On those that are solved, we get search space reductions of several orders of magnitude. Observe that the *only* reason for this is generalization, i.e., refinements of u^C on states s leading to pruning on states other than s. Without generalization, the search spaces would be identical, including tie-breaking. Generalization is what lifts a hopeless planner (DFS with h^1 dead-end detection) to a planner competitive with the state of the art in resource-constrained planning.

In the comparison Fig. 3(b) between neighbors refinement and path-cut refinement, we see that the methods either perform very similarly, or are highly complementary (with one of the two methods failing to solve the task).

The comparison Fig. 3(c) between neighbors refinement and conflict refinement provides further evidence that tailoring the refinement method to dead-end detection is typically beneficial. In the vast majority of cases where learning takes place (where conflicts are identified by search), neighbors refinement leads to better generalization, and thus to smaller search spaces, than conflict refinement.

We close this subsection by a consideration of offline learning, a performance analysis, and ablation studies. Table 3 shows the data, covering these topics in parts (A), (B), and (C) respectively.

Consider first part (A) of the table, and within that part consider first the IPC benchmarks. With $N = 2$, though not with $N = 32$, the offline methods are, generally, superior on these benchmarks compared to the same refinement methods when used online. This is, however, simply because having a small size bound means to be less risky. The offline methods, especially $N = 2$, avoid the dramatic performance losses in Airport, PegSol, and Sokoban ($N = 32$ is more risky, e.g. in Openstacks). However, the risk reduction also comes with a benefits reduction ("no risk no fun") on those domains where the online variants excel, most notably Floortile and NoMystery. Indeed, the offline-learning DFS variants *never* beat the coverage of the DFS no-learning baseline ("DFS No"), whereas the online learning variants beat it in 6 domains.

Table 3
(A) Comparison between offline and online learning methods, (B) performance analysis, (C) ablation studies, on solvable benchmarks. Best results within each of (A) and (C) highlighted in **boldface**. Abbreviations: "All" results over all instances covered; "D.I., No" results over instances where at least one conflict is identified by search, and that are commonly covered by DFS Nei and DFS No; "$N = 2$" offline C-learning with size bound 2, then DFS without learning; "$N = 32$" like $N = 2$ but with size bound 32; "CI #" number of instances where at least one conflict is identified by search; "Slo N" slowdown relative to h^1, in terms of the geometric mean over the size increase factor N; "Red $\mathcal{S}$" geometric mean over the search space size reduction factor of DFS Nei relative to DFS No; "DOS" depth-oriented search, using an open list favoring deep states; "Cl" with clause learning; "NoCl" without clause learning. Other abbreviations as before.

Domain (W)	#	(A) Online vs. offline learning													(B) Analysis			(C) Ablation (Nei)			
		Coverage													DFS Nei			Coverage			
		Base	DFS				Offline, DFS				Offline, GBFS				All	CI, No					
		GBFS					$N = 2$		$N = 32$		$N = 2$		$N = 32$		CI	Slo	Pru	DFS		DOS	
		h^{FF}	No	Nei	Pat	Con	Pat	Con	Pat	Con	Pat	Con	Pat	Con	#	N	$\mathcal{S}$	Cl	NoCl	Cl	NoCl
IPC Benchmarks																					
Airport	50	30	34	24	21	21	33	34	27	33	29	**35**	27	33	10	12.7	1.2	24	24	24	24
Childsnack	20	**1**	0	**1**	**1**	**1**	0	0	0	0	**1**	0	**1**	0	1			**1**	0	**1**	**1**
Floortile	40	10	8	4	**37**	27	8	8	7	8	10	15	10	15	4	78.2	6.2	4	4	4	4
Freecell	80	77	**78**	71	72	69	**78**	**78**	74	**78**	77	**78**	75	77	8	2.9	1.0	71	71	**72**	**72**
Mprime	35	**35**	27	28	27	27	22	27	11	27	29	30	19	30	2	20.2	1.8	28	28	28	28
Mystery	30	20	16	**26**	23	23	15	16	13	15	17	21	14	21	13	5.1	316.1	**26**	**26**	25	25
NoMprime	35	**34**	25	29	27	26	20	25	13	25	27	**34**	15	34	6	2.2	1.1	29	29	29	29
NoMystery	50	33	22	**38**	36	32	22	22	20	22	31	28	26	28	23	4.2	180.8	**38**	37	36	37
Openstacks	100	**100**	**100**	98	**100**	**100**	99	68	93	53	**100**	68	92	53	0			98	**100**	99	99
ParcPrinter	50	**50**	**50**	**50**	**50**	**50**	48	**50**	45	**50**	47	**50**	46	**50**	0			50	50	50	50
Pathways	60	**46**	**46**	**46**	**46**	**46**	**46**	**46**	10	**46**	44	**46**	8	**46**	2	2.1	1.2	46	46	46	46
PegSol	50	**50**	**50**	41	30	12	**50**	**50**	**50**	**50**	49	**50**	49	**50**	38	29.7	1.7	41	**42**	37	40
Pipesworld-Tankage	50	**40**	37	35	35	34	36	37	30	37	39	**40**	32	39	1	24.6	1.0	35	36	38	**39**
Sokoban	50	32	40	9	6	3	38	40	22	40	30	**42**	26	**42**	7	59.8	2.8	9	9	**12**	**12**
Thoughtful	20	**13**	9	9	9	9	9	9	9	9	**13**	8	**13**	8	0			9	9	9	9
Tidybot	20	**13**	**13**	**13**	**13**	**13**	12	**13**	3	12	12	**13**	5	12	0			13	13	13	13
TPP	30	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	28	**30**	**30**	29	28	0			30	30	30	30
Trucks	30	**18**	17	9	16	17	17	17	14	16	**18**	**18**	16	15	4	63.1	2.6	9	9	9	9
Woodworking	50	8	**40**	**40**	**40**	**40**	38	**40**	31	**40**	8	8	4	8	0			**40**	**40**	33	33
$\sum$ (IPC)	850	640	**642**	601	619	580	621	610	502	589	611	614	507	589	119			601	**603**	595	600
Nakhost et al. [28] Benchmarks																					
NoMystery(1.0)	30	2	1	**12**	7	4	1	1	1	1	3	2	3	2	12	1.2	114.0	**12**	10	10	7
NoMystery(1.1)	30	3	0	**12**	8	5	0	0	0	0	7	3	6	3	12			**12**	11	12	11
NoMystery(1.2)	30	5	4	**14**	13	9	4	4	3	4	12	5	11	5	14	1.4	16.2	**14**	**14**	**14**	12
NoMystery(1.3)	30	6	5	13	9	7	5	5	5	5	**19**	6	17	6	12	3.3	79.5	**13**	11	12	11
NoMystery(1.4)	30	12	3	17	13	9	3	3	3	3	**23**	12	21	12	17	1.4	19.2	**17**	14	16	14
NoMystery(1.5)	30	12	5	17	13	9	5	5	5	5	**23**	12	20	12	16	1.5	17.8	**17**	15	16	14
NoMystery(2.0)	30	23	9	19	18	14	9	9	8	9	**29**	23	29	20	14	1.6	31.1	19	19	19	19
$\sum$	210	63	27	104	81	57	27	27	25	27	**116**	63	107	60	97			**104**	94	99	88

Table 3 (*continued*)

Domain (W)	#	(A) Online vs. offline learning												(B) Analysis			(C) Ablation (Nei)			
		Coverage												DFS Nei			Coverage			
		Base	DFS				Offline, DFS				Offline, GBFS			All	CI, No					
		GBFS					$N=2$		$N=32$		$N=2$		$N=32$	CI	Slo	Pru	DFS		DOS	
		h^{FF}	No	Nei	Pat	Con	Pat	Con	Pat	Con	Pat	Con	Pat	Con	#	N	$\mathcal{S}$	Cl	NoCl	Cl	NoCl
Rovers(1.0)	30	0	0	**23**	9	8	0	0	0	0	1	0	1	0	23			**23**	20	22	20
Rovers(1.1)	30	0	0	**23**	12	13	0	0	0	0	2	0	2	0	23			**23**	18	20	16
Rovers(1.2)	30	0	0	**18**	7	9	0	0	0	0	1	0	1	0	18			**18**	16	17	14
Rovers(1.3)	30	0	0	**20**	7	6	0	0	0	0	2	0	1	0	20			**20**	17	18	16
Rovers(1.4)	30	0	1	**19**	8	7	1	1	1	1	1	0	1	0	19	2.3	12.8	**19**	17	18	16
Rovers(1.5)	30	0	1	**21**	12	8	1	1	1	1	2	0	1	0	21	3.8	29.8	**21**	17	20	16
Rovers(2.0)	30	1	6	**21**	16	15	6	6	3	6	9	1	8	1	19	3.4	96.8	**21**	19	20	17
$\sum$	210	1	8	**145**	71	66	8	8	5	8	18	1	15	1	143			**145**	124	135	115
TPP(1.0)	5	0	**1**	**1**	0	0	**1**	**1**	0	0	0	0	0	0	1	54.8	115.1	**1**	0	0	0
TPP(1.1)	5	0	0	**2**	0	0	0	0	0	0	0	0	0	0	2			**2**	1	1	0
TPP(1.2)	5	0	2	**3**	**3**	**3**	2	2	1	0	2	0	2	0	3	1.5	111.4	3	3	3	3
TPP(1.3)	5	2	2	**5**	3	3	2	2	0	0	4	2	3	0	5	2.0	326.1	**5**	3	4	3
TPP(1.4)	5	3	3	**5**	4	**5**	3	3	2	0	4	3	3	0	4	7.1	31.1	5	5	5	5
TPP(1.5)	5	0	5	**5**	**5**	**5**	**5**	**5**	**5**	0	**5**	0	3	0	3	1.7	5.7	5	5	5	5
$\sum$	30	5	13	**21**	15	16	13	13	8	0	15	5	11	0	18			**21**	17	18	16
$\sum$ Nakhost et al.	450	69	48	**270**	167	139	48	48	38	35	149	69	133	61	258			**270**	235	252	219
$\sum$ Total	1300	709	690	**871**	786	719	669	658	540	624	760	683	640	650	377			**871**	838	847	819

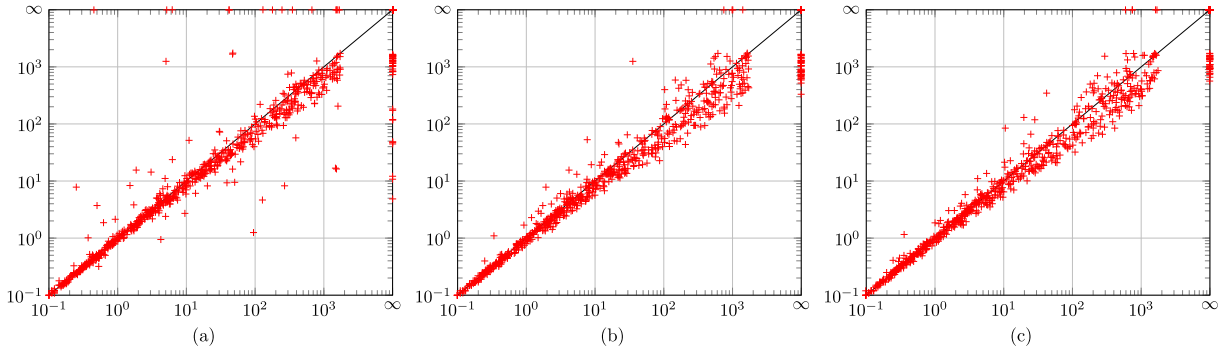

Fig. 4. Runtime comparisons. (a) DFS (y-axis) vs. DOS (x-axis). (b) DFS with clause learning (y-axis) vs. without clause learning (x-axis). (c) DOS with clause learning (y-axis) vs. without clause learning (x-axis).

On the resource-constrained benchmarks, matters are very clear-cut. The baselines are weak, and online learning improves them vastly. The same is not true for offline-learning DFS, which never improves over the baseline at all. Offline-learning GBFS is more successful, improving over the GBFS baseline in almost all cases with path-cut refinement. But it becomes competitive with online-learning DFS only in NoMystery for large values of W.

Consider now part (B) of Table 3, which shows data supporting a performance analysis with respect to the three prerequisites for online learning to work well: (1) conflict identification, i.e., the ability of forward search to find conflicts and thus enable the learning in the first place; (2) effective learning, i.e., the ability of recognizing dead-ends with small conjunction sets C; (3) strong generalization, i.e., the ability of u^C to detect states s' it was not trained on. Part (B) of Table 3 captures (1) in terms of the "CI #" data, the number of instances on which at least one conflict was identified, and hence some learning was done. It captures (2) in terms of the "Slo N" data, the size of the representation underlying the u^C computation, as a multiple of that for singleton conjunctions; i.e., the slowdown relative to h^1. It captures (3) in terms of the "Red $\mathcal{S}$" data, the search space size reduction factor relative to using only h^1 for dead-end detection.

On commonly solved instances, the slowdown and search reduction factors are directly comparable, and a performance advantage should be expected, roughly, when the former exceeds the latter. Indeed, this is a good indicator in our data here. The domains with large reduction yet small slowdown are IPC Mystery and NoMystery, as well as all resource-constrained domains, where indeed neighbors refinement vastly improves the coverage of DFS. Conversely, small reductions yet large slowdowns occur in Airport, Floortile, Freecell, Mprime, PegSol, Pipesworld-Tankage, Sokoban, and Trucks. Except for Mprime, these are precisely the cases where neighbors refinement is detrimental. In Mprime, neighbors refinement actually (slightly) improves coverage. Another unclear case is NoMprime, where neighbors refinement significantly improves coverage, yet the slowdown is larger than the reduction. In both these cases, there are only few commonly solved instances where at least one conflict is identified, which may contribute to the unclear picture. Similarly in Childsnack, where coverage is improved from 0 to 1 so there is no common instance basis to use for comparison. In all other domains – Openstacks, ParcPrinter, Pathways, Thoughtful, Tidybot, TPP, and Woodworking – the lack of advantages through learning are due to a lack of ability (1), with no or hardly any conflicts being identified by forward search.

Let us finally consider the ablation study, Table 3 part (C), fixing neighbors refinement but varying the search algorithm – DFS vs. DOS (depth-oriented search, cf. Section 4) – as well as switching clause learning on/off. (The data for DFS with clause learning is identical to that in column "DFS Nei"; we repeat it here for convenience.) On the IPC benchmarks, both DFS and clause learning, as opposed to DOS and no clause learning, have little impact on coverage. On the resource-constrained benchmarks though, both clearly and significantly improve coverage. Overall, they are useful algorithm improvements. Fig. 4 provides further evidence towards this through a per-instance runtime comparison.

7.3. Proving unsolvability

We now consider the unsolvable case. We again run the six variants of our technique (with early termination, i.e., not generating unsolvability certificates). We run offline learning without a size bound to prove unsolvability directly on the initial state; and we run offline learning with size bound N to generate static u^C dead-end detectors. We use $N \in \{2, 32\}$ like above, and we use the resulting static dead-end detectors in DFS without learning.

We run blind forward search as a simple reference baseline. We run search with h^1 as a canonical dead-end detector. We run all the competing algorithms described above, i.e., search with alternate dead-end detectors, search with admissible pruning techniques, as well as the other UIPC'16 techniques including BDD-based symbolic search etc. The admissible pruning techniques are run along with h^1, which is more competitive than blind search.

Table 4 shows coverage data (as before, we will consider offline learning separately below). Compared to DFS without learning, all refinement methods result in a performance boost on resource-constrained domains, where conflict learning is key, especially with constrainedness levels close to 1 where conflicts abound. On the non-resource UIPC'16 benchmarks

Table 4
Coverage results (number of instances proved unsolvable) on unsolvable benchmarks, comparison to the state of the art. Best results highlighted in **boldface**. Abbreviations: "Bli" blind search; "h^1" search with h^1 dead-end detection; "MSp" perfect merge-and-shrink abstraction; "PDB" pattern database heuristic; "BDD" BDD-based search in Sympa; "PDR" property-directed reachability; "Res" resource-variable detection; "Star" star-topology decoupled search; "RB" red–black state space search. Other abbreviations as before.

Domain (W)	#	Base		DFS				DFS combined with				UIPC'16 algorithms											
								MSa		Pot		Detection				Pruning h^1 and			Other				
		Bli	h^1	No	Nei	Pat	Con	N	NP	N	NP	MSp	MSa	PDB	Pot	Sim	SSS	Irr	BDD	PDR	Res	Sta	RB
UIPC'16 Non-Resource-Constrained Benchmarks																							
BagBarman	20	**12**	4	4	0	0	0	0	0	0	0	0	1	**12**	4	2	4	4	8	0	0	0	0
BagGripper	25	5	3	3	3	3	3	2	3	2	0	2	3	3	3	0	3	0	**7**	0	0	0	1
BagTransport	29	6	6	6	5	5	6	5	5	**24**	12	1	6	7	**24**	7	5	6	7	0	0	10	1
Bottleneck	50	18	40	42	18	26	28	18	18	**50**	36	10	40	38	**50**	0	40	0	0	42	0	0	**50**
CaveDiving	25	7	7	7	5	4	4	5	6	5	5	3	7	**8**	7	7	7	6	7	5	0	0	5
ChessBoard	23	5	5	5	2	1	1	2	2	**23**	5	2	5	5	**23**	5	6	4	9	1	0	0	4
Diagnosis	20	6	7	7	9	5	**12**	6	7	9	9	5	4	4	4	0	7	11	0	7	0	0	10
DocTransfer	20	5	7	6	4	5	8	8	9	4	3	5	10	12	7	**15**	6	11	10	0	0	0	3
PegSol	24	**24**	**24**	**24**	14	12	4	14	14	20	14	**24**	**24**	**24**	22	**24**	**24**	**24**	**24**	0	0	0	22
PegSolRow5	15	5	5	5	4	3	2	4	4	**15**	9	3	4	5	**15**	5	5	2	4	3	0	0	6
SlidingTiles	20	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	0	0	0	**10**
Tetris	20	5	5	5	5	5	5	5	5	**20**	0	5	5	10	**20**	0	5	5	5	0	0	0	0
$\sum$	291	108	123	124	79	79	83	79	83	182	103	70	119	138	**189**	75	122	83	91	58	0	10	112
UIPC'16 Resource-Constrained Benchmarks																							
NoMystery (0.4)	5	2	2	2	3	4	3	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	2	0	**5**	3	**5**	4	4
NoMystery (0.85)	5	0	0	0	**2**	**2**	**2**	**2**	**2**	0	**2**	**2**	**2**	**2**	0	**2**	0	0	**2**	1	**2**	**2**	**2**
NoMystery (0.99)	5	0	0	0	**2**	1	0	1	**2**	0	**2**	**2**	1	**2**	0	**2**	0	1	**2**	0	1	**2**	**2**
NoMystery (0.999)	5	0	0	0	**2**	0	0	1	**2**	0	**2**	**2**	1	**2**	0	**2**	0	1	**2**	0	1	**2**	**2**
Rovers (0.99)	20	4	7	7	12	11	8	10	12	12	12	15	9	12	6	11	7	7	13	10	14	8	**19**
TPP (0.5)	15	9	10	9	14	12	9	**15**	**15**	14	13	14	14	**15**	14	11	9	9	**15**	4	**15**	0	9
TPP (0.99)	15	6	6	6	5	3	3	9	6	5	5	10	9	10	4	6	5	6	9	1	**14**	0	6
$\sum$	70	21	25	24	40	33	25	43	44	36	41	50	41	48	29	39	23	25	48	19	**52**	18	44
$\sum$ UIPC'16	361	129	148	148	119	112	108	122	127	**218**	144	120	160	186	**218**	114	145	108	139	77	52	28	156
Nakhost et al. [28] Resource-Constrained Benchmarks																							
NoMystery (0.5)	30	14	25	25	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	29	**30**	25	12	**30**	29	**30**	**30**	27
NoMystery (0.6)	30	2	15	15	**30**	28	27	**30**	**30**	20	29	**30**	**30**	**30**	12	**30**	15	8	**30**	28	**30**	**30**	27
NoMystery (0.7)	30	0	5	7	29	23	21	29	**30**	5	26	**30**	29	**30**	2	28	6	6	**30**	22	28	**30**	29
NoMystery (0.8)	30	0	0	0	26	18	11	26	28	2	18	**30**	26	**30**	0	21	0	7	**30**	17	25	**30**	**30**
NoMystery (0.9)	30	0	0	0	14	10	7	24	25	1	14	29	24	**30**	0	13	0	3	25	8	18	29	29
$\sum$	150	16	45	47	129	109	96	139	143	58	117	149	139	**150**	43	122	46	36	145	104	131	149	142

(*continued on next page*)

Table 4 (*continued*)

Domain (W)	#	Base		DFS				DFS combined with				UIPC'16 algorithms											
								MSa		Pot		Detection				Pruning h^1 and			Other				
		Bli	h^1	No	Nei	Pat	Con	N	NP	N	NP	MSp	MSa	PDB	Pot	Sim	SSS	Irr	BDD	PDR	Res	Sta	RB
Rovers (0.5)	30	1	3	5	**30**	**30**	29	**30**	**30**	**30**	**30**	**30**	29	**30**	1	24	4	3	29	26	20	6	**30**
Rovers (0.6)	30	0	2	2	**30**	26	27	25	**30**	**30**	**30**	29	25	28	0	19	2	0	26	26	16	4	**30**
Rovers (0.7)	30	0	0	0	**30**	25	26	23	**30**	**30**	**30**	29	23	19	0	9	0	0	21	26	12	0	**30**
Rovers (0.8)	30	0	0	0	29	24	24	21	**30**	29	29	24	21	13	0	5	0	0	13	21	8	0	**30**
Rovers (0.9)	30	0	0	0	24	17	20	13	26	24	24	16	13	6	0	1	0	0	9	16	7	0	**30**
$\sum$	150	1	5	7	143	122	126	112	146	143	143	128	111	96	1	58	6	3	98	115	63	10	**150**
TPP (0.5)	5	2	4	4	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	2	3	**5**	0	**5**	0	1
TPP (0.6)	5	0	1	1	**5**	3	3	**5**	**5**	4	2	**5**	**5**	**5**	4	**5**	0	1	**5**	0	**5**	0	0
TPP (0.7)	5	0	0	0	2	2	0	3	**5**	1	1	**5**	3	**5**	1	4	0	0	3	0	**5**	0	0
TPP (0.8)	5	0	0	0	1	0	0	0	0	1	1	1	1	4	1	2	0	0	1	0	**5**	0	0
TPP (0.9)	5	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	**5**	0	0
$\sum$	25	2	5	5	13	10	8	13	15	11	9	16	14	20	11	16	2	4	14	0	**25**	0	1
$\sum$ Nakhost et al.	325	19	55	59	285	241	230	264	**304**	212	269	293	264	266	55	196	54	43	257	219	219	159	293
$\sum$ Total	686	148	203	207	404	353	338	386	431	430	413	413	424	**452**	273	310	199	151	396	296	271	187	449

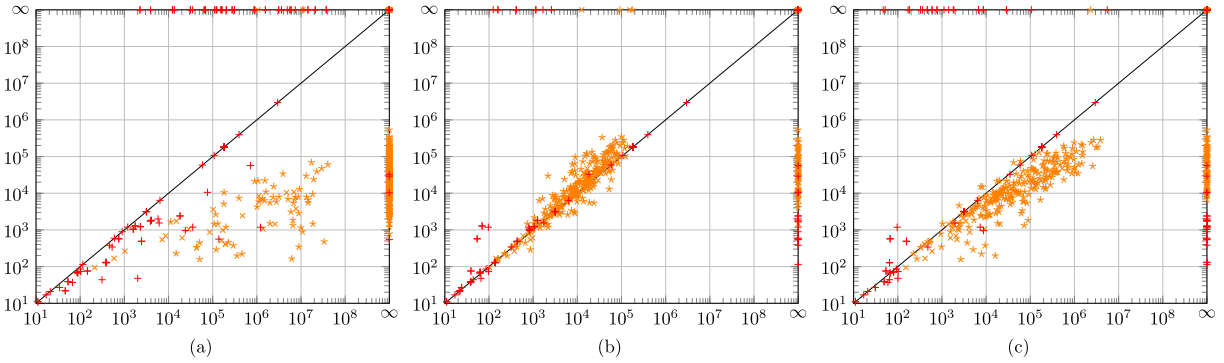

Fig. 5. DFS search space size (number of states visited) on unsolvable benchmarks, with different learning techniques. We compare neighbors refinement on the y-axis to (a) no learning, (b) path-cut refinement, respectively (c) conflict refinement on the x-axis. Unsolved benchmark instances are shown as ∞. Data points for non-resource UIPC'16 instances are shown in red, those for resource-constrained instances are shown in orange. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

though, this kind of learning simply does not work well. It helps only, for some configurations, in Diagnosis and DocTransfer. We will analyze the reasons below.

Compared to the baselines, DFS without learning has basically the same coverage as search with h^1, which makes sense as both use the same dead-end detector throughout. Blind search is consistently outclassed except in BagBarman where it is state of the art (the pattern database heuristic does not detect any dead-ends in this domain, so defaults to blind search).

Comparing the different refinement variants within our DFS framework, they behave similarly overall, though there are remarkable differences in individual domains, namely Bottleneck, Diagnosis, DocTransfer, and PegSol. On resource-constrained domains, neighbors refinement is clearly superior, followed by path-cut refinement and the conflict refinement from prior work.

Consider now the UIPC'16 algorithms. Potential heuristics clearly dominate on the non-resource UIPC'16 benchmarks, outclassing the competition (including our techniques), beat only on BagBarman, Diagnosis, and DocTransfer. On resource-constrained domains, on the other hand, potential heuristics are very weak. The next-best techniques from UIPC'16 are approximate merge-and-shrink, PDBs, and red–black state space search, with reasonable results on non-resource domains, and with strong results on resource-constrained ones. Strong stubborn sets pruning does not yield any benefits here. Dominance pruning and property-directed reachability work well on resource-constrained domains, yet still worse than the other competitors. The latter applies also to resource-variable detection. Irrelevance pruning is typically detrimental here, with benefits only in Diagnosis and DocTransfer.

Comparing the UIPC'16 algorithms to our techniques, there is no strong case for conflict learning on the non-resource domains. On the resource-constrained domains though, our techniques, especially neighbors refinement, are competitive. On the Nakhost et al. benchmarks, DFS with neighbors refinement is second in overall coverage only to approximate merge-and-shrink and red–black state space search, both completely unrelated algorithms. It beats approximate merge-and-shrink in Rovers, and it beats red–black state space search in TPP.

In difference to the solvable benchmarks discussed above, on the unsolvable benchmarks combinations of our learning methods with other dead-end detectors exhibit considerable synergy. This is most pronounced for the "NP" combination with merge-and-shrink, using neighbors refinement or path-cut refinement depending on the situation. This combination outperforms its components in each of the Nakhost et al. domains, and it has the best overall coverage on these domains, of all the algorithms tested here.

For the other combinations, the synergy is weaker. The only case where the combination outperforms its components is "N" with merge-and-shrink in UIPC'16 TPP with $W = 0.5$. Across domains, though, almost all combinations exhibit more consistent strength than their components. Indeed, all combinations except "N" with merge-and-shrink dominate their components in overall coverage.

Similarly to our discussion of solvable benchmarks above, we next provide a view of search space sizes under our different DFS learning techniques, with neighbors refinement as the comparison baseline. Fig. 5 gives the data.

The main conclusions are very similar to those we made for solvable planning tasks (cf. Fig. 3). The comparison to no learning in (a) shows that learning is often detrimental on the non-resource UIPC'16 benchmarks, yet has dramatic benefits on the resource-constrained ones. Remember that the only reason for this is generalization. Search space size reduction factors provide an impressive view on how dramatic the improvements are. Over those instances commonly solved by both configurations in Fig. 5(a), the minimum/geometric mean/maximum reduction factors on the Nakhost et al. domains are 6.7/436.5/37561.5 for NoMystery; 65.0/1286.6/69668.1 for Rovers; and 190.0/711.9/1900.5 for TPP. That is, we get reductions of 2–3 orders of magnitude on average, and even the minimum reductions are of 1–2 orders of magnitude.

The comparison between refinement methods in Fig. 5(b) and (c) yields, like on the solvable benchmarks, the conclusion that neighbors refinement and path-cut refinement either perform very similarly or are highly complementary, and that tailoring the refinement method to dead-end detection is typically beneficial.

Table 5
(A) Comparison between offline and online learning methods, (B) performance analysis, (C) ablation studies, on unsolvable benchmarks. Best results within each of (A) and (C) highlighted in **boldface**. Abbreviations: "$N = \infty$" offline C-learning without size bound, proving unsolvability on the initial state; "CI" instances where at least one conflict is identified by search *prior to exhausting the search space*. Other abbreviations as before (see captions of Tables 3 and 4).

Domain (W)	#	(A) Online vs. offline learning												(B) Analysis			(C) Ablation (Nei)			
		Coverage												DFS Nei			Coverage			
		Base		DFS				Offline						All	CI, No					
								$N = \infty$		$N = 2$		$N = 32$		CI	Slo	Pru	DFS		DOS	
		Bli	h^1	No	Nei	Pat	Con	Pat	Con	Pat	Con	Pat	Con	#	N	S	Cl	NoCl	Cl	NoCl
UIPC'16 Non-Resource-Constrained Benchmarks																				
BagBarman	20	**12**	4	4	0	0	0	0	0	4	4	0	4	0			0	0	0	0
BagGripper	25	**5**	3	3	3	3	3	0	0	2	2	0	0	0			3	3	3	3
BagTransport	29	6	6	6	5	5	6	**9**	4	5	6	3	6	0			5	5	5	5
Bottleneck	50	18	40	**42**	18	26	28	26	28	40	**42**	36	**42**	14	29.6	1.9	18	18	18	18
CaveDiving	25	**7**	**7**	**7**	5	4	4	3	1	**7**	**7**	6	**7**	5	23.5	8.3	5	6	6	**7**
ChessBoard	23	**5**	**5**	**5**	2	1	1	1	1	**5**	**5**	4	**5**	2	169.7	2.1	2	2	2	2
Diagnosis	20	6	7	7	9	5	**12**	4	**12**	5	7	5	7	9	2.9	10.4	9	9	9	9
DocTransfer	20	5	7	6	4	5	**8**	5	5	6	6	6	6	4	17.3	5.7	4	4	**5**	**5**
PegSol	24	**24**	**24**	**24**	14	12	4	14	4	**24**	**24**	**24**	**24**	14	123.7	1.8	14	14	14	14
PegSolRow5	15	**5**	**5**	**5**	4	3	2	4	3	**5**	**5**	4	**5**	2	43.5	4.0	4	4	4	4
SlidingTiles	20	**10**	**10**	**10**	**10**	**10**	**10**	0	0	**10**	**10**	**10**	**10**	0			10	10	10	10
Tetris	20	**5**	**5**	**5**	**5**	**5**	**5**	0	0	**5**	**5**	**5**	**5**	0			5	5	5	5
$\sum$	291	108	123	**124**	79	79	83	66	58	118	123	103	121				79	80	81	**82**
UIPC'16 Resource-Constrained Benchmarks																				
NoMystery (0.4)	5	2	2	2	3	4	3	**5**	4	2	2	2	2	3	1.6	76.0	3	3	3	3
NoMystery (0.85)	5	0	0	0	**2**	**2**	**2**	**2**	**2**	0	0	0	0	2			2	2	2	2
NoMystery (0.99)	5	0	0	0	**2**	1	0	0	1	0	0	0	0	2			2	2	2	2
NoMystery (0.999)	5	0	0	0	**2**	0	0	0	1	0	0	0	0	2			2	2	2	2
Rovers (0.99)	20	4	7	7	**12**	11	8	3	7	7	7	6	7	12	4.0	690.9	12	12	12	12
TPP (0.5)	15	9	10	9	**14**	12	9	12	9	10	9	10	9	14	4.6	24.4	14	14	14	14
TPP (0.99)	15	**6**	**6**	**6**	5	3	3	3	2	5	**6**	5	**6**	5	60.6	43.6	5	5	5	5
$\sum$	70	21	25	24	**40**	33	25	25	26	24	24	23	24				40	40	40	40
$\sum$ UIPC'16	361	129	**148**	**148**	119	112	108	91	84	142	147	126	145				119	120	121	**122**

Table 5 (*continued*)

Domain (W)	#	(A) Online vs. offline learning												(B) Analysis			(C) Ablation (Nei)			
		Coverage												DFS Nei			Coverage			
		Base		DFS				Offline						All	CI, No					
								$N=\infty$		$N=2$		$N=32$		CI	Slo	Pru	DFS		DOS	
		Bli	h^1	No	Nei	Pat	Con	Pat	Con	Pat	Con	Pat	Con	#	N	$\mathcal{S}$	Cl	NoCl	Cl	NoCl
Nakhost et al. [28] Resource-Constrained Benchmarks																				
NoMystery (0.5)	30	14	25	25	**30**	**30**	**30**	**30**	**30**	25	25	**30**	25	29	2.6	433.5	30	30	30	30
NoMystery (0.6)	30	2	15	15	**30**	**28**	27	**30**	28	17	15	23	15	30	3.0	355.6	30	30	30	30
NoMystery (0.7)	30	0	5	7	**29**	23	21	25	22	10	7	14	7	29	9.9	724.5	29	29	29	29
NoMystery (0.8)	30	0	0	0	**26**	18	11	21	10	0	0	7	0	26			**26**	24	25	25
NoMystery (0.9)	30	0	0	0	**14**	10	7	12	6	0	0	2	0	14			**14**	**14**	**14**	13
$\sum$	150	16	45	47	**129**	109	96	118	96	52	47	76	47				**129**	127	128	127
Rovers (0.5)	30	1	3	5	**30**	**30**	29	27	28	20	5	25	5	29	1.3	2460.2	30	30	30	30
Rovers (0.6)	30	0	2	2	**30**	26	27	25	26	16	2	23	2	29	1.2	254.5	30	30	30	30
Rovers (0.7)	30	0	0	0	**30**	25	26	22	26	10	0	14	0	30			30	30	30	30
Rovers (0.8)	30	0	0	0	**29**	24	24	13	17	4	0	10	0	29			**29**	27	28	27
Rovers (0.9)	30	0	0	0	**24**	17	20	8	9	1	0	6	0	24			24	24	24	24
$\sum$	150	1	5	7	**143**	122	126	95	106	51	7	78	7				**143**	141	142	141
TPP (0.5)	5	2	4	4	**5**	**5**	**5**	**5**	**5**	3	4	4	4	5	6.6	710.3	5	5	5	5
TPP (0.6)	5	0	1	1	**5**	3	3	3	2	1	1	2	1	5	29.5	718.6	5	5	5	5
TPP (0.7)	5	0	0	0	**2**	**2**	0	**2**	0	0	0	1	0	2			2	2	2	2
TPP (0.8)	5	0	0	0	**1**	0	0	**1**	0	0	0	0	0	1			1	1	1	1
TPP (0.9)	5	0	0	0	0	0	0	0	0	0	0	0	0	0			0	0	0	0
$\sum$	25	2	5	5	**13**	10	8	11	7	4	5	7	5				13	13	13	13
$\sum$ Nakhost et al.	325	19	55	59	**285**	241	230	224	209	107	59	161	59				**285**	281	283	281
$\sum$ Total	686	148	203	207	**404**	353	338	315	293	249	206	287	204				**404**	401	**404**	403

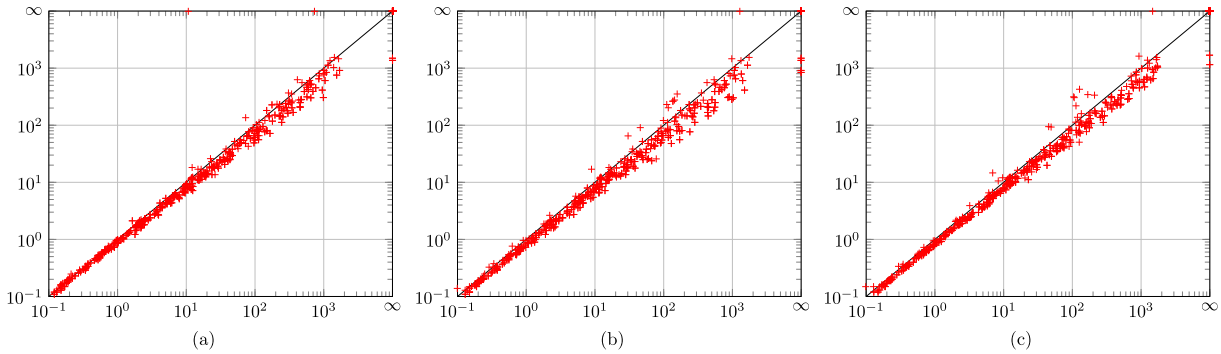

Fig. 6. Runtime comparisons. (a) DFS (y-axis) vs. DOS (x-axis). (b) DFS with clause learning (y-axis) vs. without clause learning (x-axis). (c) DOS with clause learning (y-axis) vs. without clause learning (x-axis).

Table 5 shows the data for (A) offline learning, (B) performance analysis, and (C) ablation studies. Consider first part (A) of the table. Regarding unbounded offline learning, where unsolvability is proved without search on the initial state, the data shows that this has very little merit compared to online learning. Each of path-cut refinement and conflict refinement is almost consistently dominated by the respective online learning method, the only noteworthy exceptions being BagTransport, PegSol, and NoMystery. Comparing to neighbors refinement which is not available in the offline context, the only strong cases for offline refinement are BagTransport for offline path-cut refinement, and Diagnosis for offline conflict refinement. In all cases, the online learning approaches are superior in overall coverage.

For bounded offline learning, i.e., static u^C dead-end detectors, the picture changes dramatically on the non-resource UIPC'16 benchmarks. This is, however, simply due to the size bound, which avoids the slowdown incurred by learning too many conjunctions – all the bounded offline learning configurations are dominated consistently here by DFS without any learning at all. Furthermore, on the resource-constrained benchmarks, the bounded offline learning configurations are dominated, typically outperformed, by their online learning counterparts. The single exception to the latter is the UIPC TPP domain with $W = 0.99$, where the static dead-end detectors do better. Overall, the picture is quite clearly in favor of search with online learning.

Consider now part (B) of Table 5, showing data assessing the dependency of our techniques on (1) conflict identification, (2) effective learning, and (3) strong generalization. Regarding (1), on the resource-constrained benchmarks, unsurprisingly, the "D.I. #" data shows that conflicts are identified on (almost) every instance. From the other benchmarks though, on half of the domains ability (1) is not given, i.e., no or not enough learning can take place. Namely, in all Bag* domains, in SlidingTiles, and in Tetris, no conflicts are identified at all; in PegSolRow5 it is almost that bad. In SlidingTiles and Tetris, this is simply because all actions are invertible, so the state space is strongly connected, and the first conflict is identified only after the entire state space is already explored. In the Bag* domains, in the rare cases where a conflict is identified, the learning is ineffective and prevents the search from terminating successfully.

Regarding (2) and (3), consider the "Slo N" and "Red $\mathcal{S}$" columns. We should expect good performance if the value for "Red $\mathcal{S}$" (search reduction factor) is larger than that for "Slo N" (per-node slowdown factor), on commonly solved instances. On the resource-constrained benchmarks, this is consistently the case. On the non-resource UIPC, the only domain where the reduction exceeds the slowdown is Diagnosis, precisely the only domain where neighbors refinement improves coverage relative to the baseline. In all other domains where ability (1) is given, the slowdown is much larger than the reduction, to a particularly striking extent in Bottleneck, ChessBoard, and PegSol, precisely the domains where neighbors refinement is most detrimental.

Let us finally consider the ablation study, Table 5 part (C). The different configuration settings have little impact on coverage here. DFS does a little worse than DOS on the non-resource UIPC, but a little better on the resource-constrained benchmarks; similarly for clause learning vs. no clause learning. As Fig. 6 shows, however, both DFS and clause learning are useful algorithm improvements, that rarely hurt runtime performance, while improving it in the most challenging cases.

7.4. Generating unsolvability certificates

Let us finally consider the generation of unsolvability certificates. An unsolvability certificate must (1) be verifiable in its size; must (2) be feasible to compute; and (3) is useful only if it is *compact*, i.e., loosely speaking, it is much smaller than the state space itself. Conjunction sets C where $u^C(I) = \infty$ qualify for (1). But how feasible is it to compute them, and how compact are they?

All our online learning variants guarantee to terminate with $u^C(I) = \infty$, provided the early termination option is switched off. In difference to the previous section, we now consider that setting. For comparison, we run the two unbounded offline learning variants. Table 6 shows the data.

We consider DFS with neighbors refinement, the overall most effective refinement method. Our main interest lies in comparing this online learning method to offline learning. We include both refinement methods applicable to the latter

Table 6
Results for generating unsolvability certificates C on unsolvable benchmarks, comparing online learning with DFS and neighbors refinement to offline learning with path-cut respectively conflict refinement. Best results among these three strategies highlighted in **boldface**. (A) coverage, i.e., number of instances for which a certificate was generated. (B) Geometric mean of certificate size C. (C) Compactness relative to state space size; an entry "$1:x$" means that, in the geometric mean, $|C|$ is x times smaller than $|S|$, an entry "$x:1$" means that, in the geometric mean, $|C|$ is x times larger than $|S|$. In PegSolRow5 there is no data as, on the commonly covered instances, $h^1(\mathcal{I}) = \infty$ so nothing needs to be done. Abbreviations: "w/o Cer" without certificate, i.e., DFS run with early termination option; "w/o Cer" with certificate i.e., DFS run without early termination option. Other abbreviations as before.

Domain (W)	#	(A) Coverage						(B) Geomean $\|C\|$			(C) Ratio $\|C\|/\|\mathcal{S}\|$		
		Base		DFS		Offline		Commonly covered			Commonly covered w/ Bli		
				w/o Cer	w/ Cer	$N = \infty$		DFS	Offline		DFS	Offline	
		Bli	h^1	Nei	Nei	Pat	Con	Nei	Pat	Con	Nei	Pat	Con
UIPC'16 Non-Resource-Constrained Benchmarks													
BagBarman	20	12	4	0	0	0	0						
BagGripper	25	5	3	3	0	0	0						
BagTransport	29	6	6	5	2	**9**	4	**509.0**	13156.0	3152.0	**1 : 12.6**	2.1 : 1	1 : 2.0
Bottleneck	50	18	40	18	18	26	**28**	829.7	144.5	**19.8**	1 : 24.0	1 : 137.6	**1 : 1003.2**
CaveDiving	25	7	7	5	**6**	3	1	66.0	**15.0**	85.0	1 : 1.5	**1 : 6.4**	1 : 1.1
ChessBoard	23	5	5	2	**2**	1	1	1463.0	9220.0	**691.0**	2.8 : 1	17.4 : 1	**1.3 : 1**
Diagnosis	20	6	7	9	8	4	**12**	**26.8**	392.5	1274.0	**1 : 11006.4**	1 : 751.3	1 : 231.5
DocTransfer	20	5	7	4	**5**	**5**	**5**	7417.0	515.0	**17.0**	1 : 1121.7	1 : 16154.9	**1 : 489404.4**
PegSol	24	24	24	14	**14**	**14**	4	**386.0**	901.6	747.9	**1.2 : 1**	2.8 : 1	2.3 : 1
PegSolRow5	15	5	5	4	**4**	**4**	3	–	–	–	–	–	–
SlidingTiles	20	10	10	10	0	0	0						
Tetris	20	5	5	5	0	0	0						
$\sum$	291	108	123	79	59	**66**	58						
UIPC'16 Resource-Constrained Benchmarks													
NoMystery (0.4)	5	2	2	3	3	**5**	4	**509.2**	737.7	598.8	**1 : 23218.1**	1 : 16176.1	1 : 13587.5
NoMystery (0.85)	5	0	0	2	2	2	2	**3534.4**	103531.7	9378.2			
NoMystery (0.99)	5	0	0	2	**2**	0	1						
NoMystery (0.999)	5	0	0	2	**2**	0	1						
Rovers (0.99)	20	4	7	12	**12**	3	7	**85.7**	3888.6	184.3	**1 : 9107.5**	1 : 211.6	1 : 4914.1
TPP (0.5)	15	9	10	14	**14**	12	9	416.3	**413.7**	819.9	**1 : 79.9**	1 : 77.1	1 : 40.7
TPP (0.99)	15	6	6	5	**5**	3	2	**2399.4**	4993.0	3411.4	**1 : 11.6**	1 : 5.6	1 : 8.2
$\sum$	70	21	25	40	**40**	25	26						
$\sum$ UIPC'16	361	129	148	119	**99**	91	84						

(*continued on next page*)

Table 6 (*continued*)

Domain (W)	#	(A) Coverage						(B) Geomean $\|C\|$			(C) Ratio $\|C\|/\|\mathcal{S}\|$		
		Base		DFS		Offline		Commonly covered			Commonly covered w/ Bli		
				w/o Cer	w/ Cer	$N=\infty$		DFS	Offline		DFS	Offline	
		Bli	h^1	Nei	Nei	Pat	Con	Nei	Pat	Con	Nei	Pat	Con
Nakhost et al. [28] Resource-Constrained Benchmarks													
NoMystery (0.5)	30	14	25	30	30	30	30	**216.1**	390.8	379.8	**1 : 25760.0**	1 : 22387.7	1 : 23410.2
NoMystery (0.6)	30	2	15	30	**30**	**30**	28	**516.9**	1304.3	939.8	**1 : 86197.8**	1 : 25389.4	1 : 53497.4
NoMystery (0.7)	30	0	5	29	**29**	25	22	**1180.6**	2957.6	2804.8			
NoMystery (0.8)	30	0	0	26	**26**	21	10	**2694.1**	5648.3	4260.0			
NoMystery (0.9)	30	0	0	14	**14**	12	6	**3262.3**	12782.0	10929.4			
$\sum$	150	16	45	129	**129**	118	96						
Rovers (0.5)	30	1	3	30	**30**	27	28	105.1	121.7	**90.9**	**1 : 596338.5**	1 : 67191.2	1 : 280623.0
Rovers (0.6)	30	0	2	30	**30**	25	26	211.3	181.4	**166.7**			
Rovers (0.7)	30	0	0	30	**30**	22	26	**406.1**	800.3	427.4			
Rovers (0.8)	30	0	0	29	**29**	13	17	588.4	460.9	**265.5**			
Rovers (0.9)	30	0	0	24	**24**	8	9	963.1	**502.9**	755.5			
$\sum$	150	1	5	143	**143**	95	106						
TPP (0.5)	5	2	4	5	5	5	5	**1914.8**	4023.9	2490.9	**1 : 1855.8**	1 : 249.8	1 : 680.2
TPP (0.6)	5	0	1	5	**5**	3	2	**2742.0**	3214.1	3270.9			
TPP (0.7)	5	0	0	2	**3**	2	0						
TPP (0.8)	5	0	0	1	**1**	**1**	0						
TPP (0.9)	5	0	0	0	0	0	0						
$\sum$	25	2	5	13	**14**	11	7						
$\sum$ Nakhost et al.	325	19	55	285	**286**	224	209						
[6pt] $\sum$ Total	686	148	203	404	**385**	315	293						

purpose. We include a comparison to DFS with (neighbors refinement and) early termination, not producing an unsolvability certificate, to assess the overhead incurred by the final refinement step when the search space is already empty. We include the baselines only for reference.

Consider first coverage, part (A) of the table, measuring how effective the three different strategies are at producing unsolvability certificates. Neighbors refinement is clearly best overall, and it consistently dominates on the resource-constrained benchmarks. On the non-resource benchmarks though, the offline methods are competitive, path-cut refinement even better overall, with substantial advantages in BagTransport, Bottleneck, and Diagnosis.

Observe that, with early termination, neighbors refinement "beats" the offline methods on the non-resource UIPC, while without early termination it does not. The advantage of neighbors refinement without early termination here mainly stems from the SlidingTiles and Tetris domains, which is simply due to the aforementioned fact that no learning takes place here (cf. Table 5). So the superiority of DFS with early termination here is one of search, not of learning. On the resource-constrained benchmarks, switching early termination off does not have any adverse impact on coverage.

Consider now part (B) of Table 6, giving a view of absolute certificate size (while (C) is relative to state space size). Online learning with neighbors refinement is superior on the resource-constrained benchmarks, except for Nakhost et al. Rovers (and one case of UIPC TPP), where the picture is more mixed. On the non-resource UIPC benchmarks, the methods are complementary, with differing strengths depending on the domain.

By definition, the relative performance of learning methods is the same in parts (C) and (B). What's remarkable in (C) is that *the certificates found often are extremely compact, several orders of magnitude smaller than the state space itself*. This is especially pronounced in the resource-constrained benchmarks, but also happens in some of the other domains, most notably in Bottleneck, Diagnosis, and DocTransfer.

8. Conclusion

Our work pioneers conflict-directed learning, of sound generalizable knowledge, from conflicts – dead-end states – in forward state space search. The basis are critical-path heuristic functions h^C, that allow to consider an arbitrary set C of atomic conjunctions, and that detect all dead-ends in the limit. Our key technical contributions are search methods identifying conflict states, and refinement methods extending C so that h^C recognizes these states. The resulting technique is, in our humble opinion, quite elegant, and suggests that the learning from "true" conflicts in state space search, not necessitating a solution length bound, is worth the community's attention.

Beauty contests aside, from a pragmatical point of view our techniques certainly do not, as they stand, deliver an empirical breakthrough. They require a rather specific kind of problem structure to work well, namely structure that allows for (1) quick conflict identification, (2) effective learning, and (3) strong generalization. This kind of problem structure is typical of resource-constrained planning, as far as reflected by the current benchmarks from that area. On other domains, as far as reflected by the competition benchmarks, this structure is scarce, though it does sometimes appear.

An interesting question in this context is the relation between requirement (1) vs. a plan length bound. The two requirements are correlated in that conflict identification will be easier on problems whose search paths are typically short. Furthermore, if a bound is available, then manifold alternate conflict analysis techniques can be used, simply via the correspondence to constraint satisfaction.

However, having "typically" short search paths is a much weaker assumption than having a globally valid length bound, in particular a bound that is known a-priori before search begins. In the non-resource domains where our techniques work well – Floortile, ParcPrinter, Pathways, Childsnack, Diagnosis, DocTransfer – it is completely unclear how an upper bound should be derived. Even in the resource-constrained benchmarks, this is not obvious: not all actions consume resources, so some reasoning over the possible non-consuming action subsequences would be required.

Regarding future work, ours is merely a first foray into forward search conflict-learning techniques, and lots more remains to be explored. We hope and expect our work to be the beginning of the story, not its end.

For conjunctions learning, important open questions pertain, e.g., to ranking criteria allowing a more informed choice of which new conjunctions to construct during refinement, as well as allowing to *forget* conjunctions learned previously in case they did not prove useful for the search.

For clause learning, exciting questions pertain to extending its, as yet, very limited role. Can we learn easily testable criteria that, in conjunction, are sufficient *and* necessary for $u^C = \infty$, thus matching the pruning power of u^C itself? Can such criteria form building blocks for later learning steps, like the clauses in SAT? Can we do some form of reasoning over clauses, deducing new knowledge from the old one, given the action specifications?

Critical-path heuristics are merely one means for dead-end detection, and an exciting big line of research is the design of refinement methods for other dead-end detectors. Can we refine merge-and-shrink unsolvability heuristics on the fly? What about potential heuristics? If so, how to make the most out of the combination of all three methods?

Last but not least, one thing we would particularly like to see is the export of this (kind of) technique to game-playing and model checking, where dead-end detection is at least as, probably more, important than in classical planning. For h^C refinement, this works "out of the box" modulo the applicability of Equation (1), i.e., the definition of critical-path heuristics. As is, this requires conjunctive subgoaling behavior. But more general logics (e.g. minimization to handle disjunctions) should be manageable.

Acknowledgements

This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1, "Critically Constrained Planning via Partial Delete Relaxation".

Appendix A. Proofs

Theorem 1. *At the start of the **while** loop in Algorithm 1, the labeled states are exactly the known dead-ends.*

Proof. Soundness, i.e., t labeled $\implies$ t is a known dead-end, follows immediately from construction because at the moment a state t is labeled we have $\mathcal{R}[t] \subseteq \mathit{Closed}$, and once that condition is true obviously it remains true for the remainder of the search.

Completeness, i.e., t is a known dead-end $\implies$ t labeled, can be proved by induction on the number of expansions. Assume that the claim holds before a state s is expanded; we need to show that, for any states t that were not known dead-ends before but are known dead-ends now, t will be labeled. Call such t *new-label states*. Clearly, any new-label state must be an ancestor of s. Therefore, a new-label state can exist only if, after the expansion, $\mathcal{R}[s] \subseteq \mathit{Closed}$: else, an open state is reachable from s, and transitively is reachable from all ancestors of s. In the case where $\mathcal{R}[s] \subseteq \mathit{Closed}$, s is labeled and so is every new-label state parent t of s, due to the recursive invocation on t. It remains to show that each new-label state t will indeed be reached, and thus labeled, during the reverse traversal of the search space induced by the recursive invocations of `CheckAndLearn`. This is a direct consequence of the following two observations: (1) each ancestor t of s has not been labeled dead end so far, and (2) for each new-label state t, the search space contains a path of new-label states $t = t_0, t_1, \ldots, t_n = s$. The first observation follows immediately from the algorithm: since t is an ancestor of s, t must have been expanded at some point (which means that t could not have been labeled dead end before its expansion), and t could not have been labeled dead end during any previous call to `CheckAndLearn` because at least one open state was reachable from t during any such call. The second observation follows immediately from $\mathcal{R}[t] \subseteq \mathit{Closed}$: (as above) t is an ancestor of s, i.e., the search space contains a path $t = t_0, t_1, \ldots, t_n = s$, and due to the transitivity of reachability, for every $1 \leq i < n$, $\mathcal{R}[t_i] \subseteq \mathit{Closed}$. Obviously, for every $1 \leq i \leq n$, s is also reachable from t_i, meaning that t_i must be a new-label state, too. Since `CheckAndLearn` will traverse at least one such path from t to s in reverse order, t will indeed be labeled eventually. □

Theorem 2. *Let C be any set of atomic conjunctions. Let s be a state with $h^C(s) < h^*(s)$. Then:*

(i) *The execution of* `PathCutRefine`$(\mathcal{G}, h^C(s))$ *is well defined, i.e., (a) in any call* `PathCutRefine`(G, n) *there exists $c \in C$ so that $c \subseteq G$ and $h^C(s, c) \geq n$; and (b) if $n = 0$, then $G \not\subseteq s$.*

(ii) *If X is the set of conjunctions resulting from* `PathCutRefine`$(\mathcal{G}, h^C(s))$*, then $h^{C \cup X}(s) > h^C(s)$.*

Proof.

(i) (a) follows directly from Equation (1). Initially, there must be some $c \in C$ so that $c \subseteq \mathcal{G}$ and $h^C(s, c) = h^C(s, \mathcal{G})$ (last case of Equation (1)). Consider a recursive call `PathCutRefine`(G, n). Let G', n' be the arguments of the call to `PathCutRefine` that caused the recursion, let a be the corresponding action, and let $c' \subseteq G'$ be the conjunction selected in `PathCutRefine`(G', n'). Due to the selection of c', we have $h^C(s, c') = h^C(s, G') = n' = n + 1$; and because $c' \subseteq G'$, we also have $R(c', a) \subseteq R(G', a) = G$. Hence, by definition of h^C, $h^C(s, G) \geq n$, i.e. there is a conjunction $c \in C$, $c \subseteq G$ so that $h^C(s, c) \geq n$.

For (b), assume for contradiction that there is a call `PathCutRefine`$(G, 0)$ where $G \subseteq s$. Let $a_n, \ldots, a_1$ be the actions that label the recursion path down to the call `PathCutRefine`$(G, 0)$. It is easy to show by induction that $\langle a_1, \ldots, a_n \rangle$ is actually a plan for s. However, n is exactly $h^C(s)$, which means that $h^C(s) = h^*(s)$, a contradiction to the assumption.

(ii) We show for every call `PathCutRefine`(G, n) and for the constructed conflict x that $h^{C \cup X}(s, x) > n$ when `PathCutRefine`(G, n) terminates. In other words, when `PathCutRefine`$(\mathcal{G}, h^C(s))$ terminates, then we have $h^{C \cup X}(s) > h^C(s)$, as desired. The proof is by induction on n. For $n = 0$, the conflict $x \subseteq G$ is chosen such that $x \not\subseteq s$. Hence, $h^{C \cup X}(s, x) > 0 = n$ due to Equation (1). For the induction step, $n > 0$, let x be the conflict that is constructed in the call `PathCutRefine`(G, n). Since $n > 0$, there must be an atomic conjunction $c \in C$ that is part of x, $c \subseteq x$, and so that $h^C(s, c) \geq n$. If $h^C(s, c) > n$, then clearly $h^C(s, x) > n$ and the claim follows. So, assume that $h^C(s, c) = n$, and let $a \in \mathcal{A}[x]$ be an arbitrary achiever of x (i.e., $R(x, a) \neq \bot$). In case $\mathcal{A}[x]$ is empty, it directly follows that $h^{C \cup X}(s, x) = \infty > n$ (Equation (1)). Otherwise, distinguish between the cases $a \in \mathcal{A}[c]$ and $a \notin \mathcal{A}[c]$. If $a \notin \mathcal{A}[c]$, then, since $c \subseteq x$ and $x \cap \mathit{del}(a) = \emptyset$, i.e. $c \cap \mathit{del}(a) = \emptyset$, we have that $\mathit{add}(a) \cap c = \emptyset$. Therefore, $c \subseteq R(x, a)$ and thus $h^{C \cup X}(s, R(x, a)) \geq n$. On the other hand, if $a \in \mathcal{A}[c]$, then we must have recurred on $G' = R(G, a)$ and $n' = n - 1$. If x' is the conflict constructed in this call, then we know by induction hypothesis that $h^{C \cup X}(s, x') > n'$. Because of the selection of x, we ensured that $x' \subseteq R(x, a)$, and as a consequence $h^{C \cup X}(s, R(x, a)) > n' = n - 1$. Since a was chosen arbitrarily, this shows that $h^{C \cup X}(s, x) > n$. □

Theorem 3. *Let C be any set of atomic conjunctions. Let $\hat{S}$ be a set of dead-end states, and let $\hat{T}$ be its neighbors with the u^C-recognized neighbors property. Then:*

(i) *The execution of* `NeighborsRefine`$(\mathcal{G})$ *is well defined, i.e., it is always possible to extract an x as specified.*
(ii) *The execution of* `NeighborsRefine`$(\mathcal{G})$ *terminates.*
(iii) *Upon termination of* `NeighborsRefine`$(\mathcal{G})$, $u^{C\cup X}(s) = \infty$ *for every $s \in \hat{S}$.*

Proof. (i) follows by induction on the recursion depth. For the induction beginning, note that the selection of $x \subseteq G = \mathcal{G}$ in `ExtractX` is well-defined because of the recognized neighbors property (*), and because $\hat{S}$ does not contain a goal state. For the induction step, assume for contradiction that `ExtractX` fails to select a conjunction $x \subseteq G$ that satisfies the condition of Lemma 2, where $G = R(x', a)$ is given as input, i.e., x' is chosen as in Lemma 2, and a is an action from $\mathcal{A}[x']$. In other words, (i) there is a state $s \in \hat{S}$ with $G \subseteq s$, or (ii) there is a state $t \in \hat{T}$ so that for all conjunctions $c_0 \in C$ with $c_0 \subseteq G$, $u^C(t, c_0) < \infty$. It cannot be (i) because otherwise, it follows from $a \in \mathcal{A}[x']$ and $R(x', a) = G \subseteq s$ that $s[[a]]$ is defined and $x' \subseteq s[[a]]$, i.e., $h^*(s, x') < \infty$. This is a contradiction to the selection of x'. For (ii), let $c'_0 \in C$, $c'_0 \subseteq x'$ be some conjunction with $u^C(t, c'_0) = \infty$. Such a conjunction must exist due to the selection of x'. Since $u^C(t, c'_0) = \infty$, it directly follows that $c'_0 \not\subseteq R(x', a)$. However, $c'_0 \subseteq x'$, so $c'_0 \subseteq add(a)$, and thus $a \in \mathcal{A}[c'_0]$. Now plugging in the definition of u^C, we get from $u^C(t, R(x', a)) < \infty$ and $R(c'_0, a) \subseteq R(x', a)$ that $u^C(t, R(c'_0, a)) < \infty$. In other words: $u^C(t, c'_0) < \infty$. This is clearly a contradiction to the selection of c'_0. We conclude that there must be a conjunction $x \subseteq G$ that satisfies the conditions of Lemma 2.

For (ii) note that in every single recursion, a new conjunction x is added to X. This is true because before going into recursion on some $R(x, a)$, we make sure that there does not exist $x' \in X$ so that $x' \subseteq R(x, a)$. Thus, regardless of the selection of the conflict $x' \subseteq R(x, a)$ in the corresponding call to `ExtractX`$(R(x, a))$, x' cannot be contained in X. After selecting the conflict x', it is added to X. So X is extended by a new conflict in each recursion. But since the overall number of conjunctions is bounded, it immediately follows that the number of recursions is bounded.

To show (iii), we make use of the observation $u^C(s, G) = \infty$ iff $h^C(s, G) = \infty$ for any set of facts $G \subseteq \mathcal{F}$. Let $s \in \hat{S}$ be arbitrary, and let $x \in X$ be a conjunction with minimal $h^{C\cup X}$-value, i.e., let $x \in X$ be so that for all $x' \in X$: $h^{C\cup X}(s, x) \leq h^{C\cup X}(s, x')$. Assume for contradiction that $h^{C\cup X}(s, x) < \infty$. Due to the construction of X, it must be $x \not\subseteq s$, meaning that there must be an action $a \in \mathcal{A}[x]$ with $h^{C\cup X}(s, R(x, a)) < h^{C\cup X}(s, x)$ (definition of $h^{C\cup X}$). However, the refinement algorithm ensures that X contains a conjunction $x' \subseteq R(x, a)$: in the call to `Refine` where x is added to X, the algorithm makes sure that for each action $a \in \mathcal{A}[x]$, either there is already a conjunction $x' \in X$ so that $x' \subseteq R(x, a)$, or it calls `Refine`$(R(x, a))$ which in turn adds a conjunction $x' \subseteq R(x, a)$ to X. But this is a contradiction to the $h^{C\cup X}$ minimality assumption: as there is a conjunction $x' \in X$ with $x' \subseteq R(x, a)$, it is $h^{C\cup X}(s, x') \leq h^{C\cup X}(s, R(x, a)) < h^{C\cup X}(s, x)$. This shows that $h^{C\cup X}(s, x) = \infty$ for every $x \in X$, and for every state $s \in \hat{S}$, and thus $u^C(s) = \infty$ for every $s \in \hat{S}$. □

References

[1] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, Artif. Intell. 41 (1990) 273–312.
[2] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Comput. Intell. 9 (1993) 268–299.
[3] J. Marques-Silva, K. Sakallah, GRASP: a search algorithm for propositional satisfiability, IEEE Trans. Comput. 48 (1999) 506–521.
[4] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings of the 38th Conference on Design Automation, DAC-01, IEEE Computer Society, Las Vegas, Nevada, USA, 2001.
[5] R. Dechter, D. Frost, Backjump-based backtracking for constraint satisfaction problems, Artif. Intell. 136 (2002) 147–188.
[6] N. Eén, N. Sörensson, An extensible sat-solver, in: Proceedings of the 6th International Conference Theory and Applications of Satisfiability Testing, SAT'03, 2003, pp. 502–518.
[7] P. Beame, H.A. Kautz, A. Sabharwal, Towards understanding and harnessing the potential of clause learning, J. Artif. Intell. Res. 22 (2004) 319–351.
[8] H. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: M. Pollack (Ed.), Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI'99, Morgan Kaufmann, Stockholm, Sweden, 1999, pp. 318–325.
[9] J. Rintanen, K. Heljanko, I. Niemelä, Planning as satisfiability: parallel plans and algorithms for plan search, Artif. Intell. 170 (2006) 1031–1080.
[10] A.L. Blum, M.L. Furst, Fast planning through planning graph analysis, Artif. Intell. 90 (1997) 279–298.
[11] D. Long, M. Fox, Efficient implementation of the plan graph in STAN, J. Artif. Intell. Res. 10 (1999) 87–115.
[12] S. Kambhampati, Planning graph as a (dynamic) CSP: exploiting EBL, DDB and other CSP search techniques in graphplan, J. Artif. Intell. Res. 12 (2000) 1–34.
[13] A.R. Bradley, Sat-based model checking without unrolling, in: Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'11, 2011, pp. 70–87.
[14] M. Suda, Property directed reachability for automated planning, J. Artif. Intell. Res. 50 (2014) 265–319.
[15] S. Minton, J.G. Carbonell, C.A. Knoblock, D. Kuokka, O. Etzioni, Y. Gil, Explanation-based learning: a problem solving perspective, Artif. Intell. 40 (1989) 63–118.
[16] S. Kambhampati, S. Katukam, Y. Qu, Failure driven dynamic search control for partial order planners: an explanation based approach, Artif. Intell. 88 (1996) 253–315.
[17] S. Kambhampati, On the relations between intelligent backtracking and failure-driven explanation-based learning in constraint satisfaction and planning, Artif. Intell. 105 (1998) 161–208.
[18] N. Bhatnagar, J. Mostow, On-line learning from search failures, Mach. Learn. 15 (1994) 69–117.
[19] A. Kolobov, Mausam, D.S. Weld, Discovering hidden structure in factored MDPs, Artif. Intell. 189 (2012) 19–47.
[20] R.E. Korf, Real-time heuristic search, Artif. Intell. 42 (1990) 189–211.
[21] A. Reinefeld, T.A. Marsland, Enhanced iterative-deepening search, IEEE Trans. Pattern Anal. Mach. Intell. 16 (1994) 701–710.
[22] A.G. Barto, S.J. Bradtke, S.P. Singh, Learning to act using real-time dynamic programming, Artif. Intell. 72 (1995) 81–138.

[23] B. Bonet, H. Geffner, Learning depth-first search: a unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs, in: D. Long, S. Smith (Eds.), Proceedings of the 16th International Conference on Automated Planning and Scheduling, ICAPS'06, Morgan Kaufmann, Ambleside, UK, 2006, pp. 142–151.
[24] D.E. Smith, Choosing objectives in over-subscription planning, in: S. Koenig, S. Zilberstein, J. Koehler (Eds.), Proceedings of the 14th International Conference on Automated Planning and Scheduling, ICAPS'04, Morgan Kaufmann, Whistler, Canada, 2004, pp. 393–401.
[25] A. Gerevini, P. Haslum, D. Long, A. Saetti, Y. Dimopoulos, Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners, Artif. Intell. 173 (2009) 619–668.
[26] C. Domshlak, V. Mirkis, Deterministic oversubscription planning as heuristic search: abstractions and reformulations, J. Artif. Intell. Res. 52 (2015) 97–169.
[27] P. Haslum, H. Geffner, Heuristic planning with time and resources, in: A. Cesta, D. Borrajo (Eds.), Proceedings of the 6th European Conference on Planning, ECP'01, Springer-Verlag, 2001, pp. 121–132.
[28] H. Nakhost, J. Hoffmann, M. Müller, Resource-constrained planning: a Monte Carlo random walk approach, in: B. Bonet, L. McCluskey, J.R. Silva, B. Williams (Eds.), Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS'12, AAAI Press, 2012, pp. 181–189.
[29] A.J. Coles, A. Coles, M. Fox, D. Long, A hybrid LP-RPG heuristic for modelling numeric resource flows in planning, J. Artif. Intell. Res. 46 (2013) 343–412.
[30] A. Junghanns, J. Schaeffer, Sokoban: evaluating standard single-agent search techniques in the presence of deadlock, in: Proceedings of the 12th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, 1998, pp. 1–15.
[31] R. Bjarnason, P. Tadepalli, A. Fern, Searching solitaire in real time, ICGA J. 30 (2007) 131–142.
[32] G. Behrmann, J. Bengtsson, A. David, K.G. Larsen, P. Pettersson, W. Yi, UPPAAL implementation secrets, in: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, 2002.
[33] G. Holzmann, The Spin Model Checker – Primer and Reference Manual, Addison–Wesley, 2004.
[34] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Directed explicit-state model checking in the validation of communication protocols, Int. J. Softw. Tools Technol. Transf. 5 (2004) 247–267.
[35] C. Bäckström, P. Jonsson, S. Ståhlberg, Fast detection of unsolvable planning instances using local consistency, in: M. Helmert, G. Röger (Eds.), Proceedings of the 6th Annual Symposium on Combinatorial Search, SOCS'13, AAAI Press, 2013, pp. 29–37.
[36] J. Hoffmann, P. Kissmann, Á. Torralba, "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability, in: T. Schaub (Ed.), Proceedings of the 21st European Conference on Artificial Intelligence, ECAI'14, IOS Press, Prague, Czech Republic, 2014.
[37] C.J. Muise, S.A. McIlraith, J.C. Beck, Improved non-deterministic planning by exploiting state relevance, in: B. Bonet, L. McCluskey, J.R. Silva, B. Williams (Eds.), Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS'12, AAAI Press, 2012.
[38] A. Camacho, C. Muise, S.A. McIlraith, From FOND to robust probabilistic planning: computing compact policies that bypass avoidable deadends, in: A. Coles, A. Coles, S. Edelkamp, D. Magazzeni, S. Sanner (Eds.), Proceedings of the 26th International Conference on Automated Planning and Scheduling, ICAPS'16, AAAI Press, 2016.
[39] P. Haslum, H. Geffner, Admissible heuristics for optimal planning, in: S. Chien, R. Kambhampati, C. Knoblock (Eds.), Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems, AIPS'00, AAAI Press, Menlo Park, Breckenridge, CO, 2000, pp. 140–149.
[40] B. Bonet, H. Geffner, Planning as heuristic search, Artif. Intell. 129 (2001) 5–33.
[41] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. 14 (2001) 253–302.
[42] S. Edelkamp, Planning with pattern databases, in: A. Cesta, D. Borrajo (Eds.), Proceedings of the 6th European Conference on Planning, ECP'01, Springer-Verlag, 2001, pp. 13–24.
[43] M. Helmert, C. Domshlak, Landmarks, critical paths and abstractions: what's the difference anyway?, in: A. Gerevini, A. Howe, A. Cesta, I. Refanidis (Eds.), Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS'09, AAAI Press, 2009, pp. 162–169.
[44] S. Richter, M. Westphal, The LAMA planner: guiding cost-based anytime planning with landmarks, J. Artif. Intell. Res. 39 (2010) 127–177.
[45] M. Helmert, P. Haslum, J. Hoffmann, R. Nissim, Merge & shrink abstraction: a method for generating lower bounds in factored state spaces, J. Assoc. Comput. Mach. 61 (2014).
[46] K. Dräger, B. Finkbeiner, A. Podelski, Directed model checking with distance-preserving abstractions, in: A. Valmari (Ed.), Proceedings of the 13th International SPIN Workshop, SPIN 2006, in: Lecture Notes in Computer Science, vol. 3925, Springer-Verlag, 2006, pp. 19–34.
[47] M. Helmert, P. Haslum, J. Hoffmann, Flexible abstraction heuristics for optimal sequential planning, in: M. Boddy, M. Fox, S. Thiebaux (Eds.), Proceedings of the 17th International Conference on Automated Planning and Scheduling, ICAPS'07, Morgan Kaufmann, Providence, Rhode Island, USA, 2007, pp. 176–183.
[48] K. Dräger, B. Finkbeiner, A. Podelski, Directed model checking with distance-preserving abstractions, Int. J. Softw. Tools Technol. Transf. 11 (2009) 27–37.
[49] Á. Torralba, J. Hoffmann, P. Kissmann, MS-Unsat and SimulationDominance: merge-and-shrink and dominance pruning for proving unsolvability, in: UIPC 2016 Planner Abstracts, 2016, pp. 12–15.
[50] F. Pommerening, J. Seipp, Fast downward dead-end pattern database, in: UIPC 2016 Planner Abstracts, 2016, p. 2.
[51] F. Pommerening, M. Helmert, G. Röger, J. Seipp, From non-negative to general operator cost partitioning, in: B. Bonet, S. Koenig (Eds.), Proceedings of the 29th AAAI Conference on Artificial Intelligence, AAAI'15, AAAI Press, 2015, pp. 3335–3341.
[52] J. Seipp, F. Pommerening, M. Helmert, New optimization functions for potential heuristics, in: R. Brafman, C. Domshlak, P. Haslum, S. Zilberstein (Eds.), Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS'15, AAAI Press, 2015, pp. 193–201.
[53] J. Seipp, F. Pommerening, S. Sievers, M. Wehrle, Fast downward Aidos, in: UIPC 2016 Planner Abstracts, 2016, pp. 28–38.
[54] M. Steinmetz, J. Hoffmann, Towards clause-learning state space search: learning to recognize dead-ends, in: D. Schuurmans, M. Wellman (Eds.), Proceedings of the 30th AAAI Conference on Artificial Intelligence, AAAI'16, AAAI Press, 2016.
[55] M. Steinmetz, J. Hoffmann, Clone: a critical-path driven clause learner, in: UIPC 2016 Planner Abstracts, 2016, pp. 24–27.
[56] Á. Torralba, V. Alcázar, Constrained symbolic search: on mutexes, BDD minimization and more, in: M. Helmert, G. Röger (Eds.), Proceedings of the 6th Annual Symposium on Combinatorial Search, SOCS'13, AAAI Press, 2013, pp. 175–183.
[57] Á. Torralba, Sympa: symbolic perimeter abstractions for proving unsolvability, in: UIPC 2016 Planner Abstracts, 2016, pp. 8–11.
[58] C. Domshlak, J. Hoffmann, M. Katz, Red–black planning: a new systematic approach to partial delete relaxation, Artif. Intell. 221 (2015) 73–114.
[59] D. Gnad, M. Steinmetz, M. Jany, J. Hoffmann, I. Serina, A. Gerevini, Partial delete relaxation, unchained: on intractable red–black planning and its applications, in: J. Baier, A. Botea (Eds.), Proceedings of the 9th Annual Symposium on Combinatorial Search, SOCS'16, AAAI Press, 2015.
[60] D. Gnad, M. Steinmetz, J. Hoffmann, Django: unchaining the power of red–black planning, in: UIPC 2016 Planner Abstracts, 2016, pp. 19–23.
[61] M. Wehrle, M. Helmert, Efficient stubborn sets: generalized algorithms and selection strategies, in: S. Chien, M. Do, A. Fern, W. Ruml (Eds.), Proceedings of the 24th International Conference on Automated Planning and Scheduling, ICAPS'14, AAAI Press, 2014.
[62] Á. Torralba, J. Hoffmann, Simulation-based admissible dominance pruning, in: Q. Yang (Ed.), Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI'15, AAAI Press/IJCAI, 2015, pp. 1689–1695.
[63] Á. Torralba, P. Kissmann, Focusing on what really matters: irrelevance pruning in merge-and-shrink, in: L. Lelis, R. Stern (Eds.), Proceedings of the 8th Annual Symposium on Combinatorial Search, SOCS'15, AAAI Press, 2015, pp. 122–130.
[64] P. Haslum, Improving heuristics through relaxed search – an analysis of TP4 and HSP*a in the 2004 planning competition, J. Artif. Intell. Res. 25 (2006) 233–267.

[65] P. Haslum, $h^m(P) = h^1(P^m)$: alternative characterisations of the generalisation from $h^{\max}$ to h^m, in: A. Gerevini, A. Howe, A. Cesta, I. Refanidis (Eds.), Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS'09, AAAI Press, 2009, pp. 354–357.
[66] P. Haslum, Incremental lower bounds for additive cost planning problems, in: B. Bonet, L. McCluskey, J.R. Silva, B. Williams (Eds.), Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS'12, AAAI Press, 2012, pp. 74–82.
[67] E. Keyder, J. Hoffmann, P. Haslum, Improving delete relaxation heuristics through explicitly represented conjunctions, J. Artif. Intell. Res. 50 (2014) 487–533.
[68] M. Fickert, J. Hoffmann, M. Steinmetz, Combining the delete relaxation with critical-path heuristics: a direct characterization, J. Artif. Intell. Res. 56 (2016) 269–327.
[69] E. Keyder, J. Hoffmann, P. Haslum, Semi-relaxed plan heuristics, in: B. Bonet, L. McCluskey, J.R. Silva, B. Williams (Eds.), Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS'12, AAAI Press, 2012, pp. 128–136.
[70] T. Bylander, The computational complexity of propositional STRIPS planning, Artif. Intell. 69 (1994) 165–204.
[71] J. Hoffmann, Local search topology in planning benchmarks: an empirical analysis, in: B. Nebel (Ed.), Proceedings of the 17th International Joint Conference on Artificial Intelligence, IJCAI'01, Morgan Kaufmann, Seattle, Washington, USA, 2001, pp. 453–458.
[72] J. Hoffmann, Where 'ignoring delete lists' works: local search topology in planning benchmarks, J. Artif. Intell. Res. 24 (2005) 685–758.
[73] J. Hoffmann, M. Fickert, Explicit conjunctions w/o compilation: computing $h^{\mathrm{FF}}(\Pi^C)$ in polynomial time, in: R. Brafman, C. Domshlak, P. Haslum, S. Zilberstein (Eds.), Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS'15, AAAI Press, 2015.
[74] N.J. Nilsson, Problem Solving Methods in Artificial Intelligence, McGraw–Hill, 1971.
[75] P. Jiménez, C. Torras, An efficient algorithm for searching implicit AND/OR graphs with cycles, Artif. Intell. 124 (1) (2000) 1–30.
[76] B. Bonet, H. Geffner, Labeled RTDP: improving the convergence of real-time dynamic programming, in: E. Giunchiglia, N. Muscettola, D. Nau (Eds.), Proceedings of the 13th International Conference on Automated Planning and Scheduling, ICAPS'03, Morgan Kaufmann, Trento, Italy, 2003, pp. 12–21.
[77] R.E. Tarjan, Depth first search and linear graph algorithms, SIAM J. Comput. 1 (1972) 146–160.
[78] B. Bonet, H. Geffner, Faster heuristic search algorithms for planning with uncertainty and full feedback, in: G. Gottlob (Ed.), Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI'03, Morgan Kaufmann, Acapulco, Mexico, 2003, pp. 1233–1238.
[79] M. Helmert, The Fast Downward planning system, J. Artif. Intell. Res. 26 (2006) 191–246.
[80] M. Helmert, Concise finite-domain representations for PDDL planning tasks, Artif. Intell. 173 (2009) 503–535.
[81] D. Gnad, Á. Torralba, J. Hoffmann, M. Wehrle, Decoupled search for proving unsolvability, in: UIPC 2016 Planner Abstracts, 2016, pp. 16–18.
[82] A. Valmari, Stubborn sets for reduced state space generation, in: Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, 1989, pp. 491–515.
[83] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. 35 (1986) 677–691.
[84] S. Edelkamp, M. Helmert, Exhibiting knowledge in planning problems to minimize state encoding length, in: S. Biundo, M. Fox (Eds.), Proceedings of the 5th European Conference on Planning, ECP'99, Springer-Verlag, 1999, pp. 135–147.
[85] T. Balyo, M. Suda, Reachlunch entering the Unsolvability IPC 2016, in: UIPC 2016 Planner Abstracts, 2016, pp. 3–5.
[86] J. Seipp, M. Helmert, Counterexample-guided Cartesian abstraction refinement, in: D. Borrajo, S. Fratini, S. Kambhampati, A. Oddi (Eds.), Proceedings of the 23rd International Conference on Automated Planning and Scheduling, ICAPS'13, AAAI Press, Rome, Italy, 2013, pp. 347–351.
[87] J. Seipp, M. Helmert, Diverse and additive cartesian abstraction heuristics, in: S. Chien, M. Do, A. Fern, W. Ruml (Eds.), Proceedings of the 24th International Conference on Automated Planning and Scheduling, ICAPS'14, AAAI Press, 2014.
[88] D. Gnad, J. Hoffmann, Beating LM-cut with h^{max} (sometimes): fork-decoupled state space search, in: R. Brafman, C. Domshlak, P. Haslum, S. Zilberstein (Eds.), Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS'15, AAAI Press, 2015.
[89] D. Gnad, J. Hoffmann, Red–black planning: a new tractability analysis and heuristic function, in: L. Lelis, R. Stern (Eds.), Proceedings of the 8th Annual Symposium on Combinatorial Search, SOCS'15, AAAI Press, 2015.
[90] K. Korovin, M. Suda, iProverPlan: a system description, in: UIPC 2016 Planner Abstracts, 2016, pp. 6–7.
[91] P. Haslum, Adapting h^{++} for proving plan non-existence, in: UIPC 2016 Planner Abstracts, 2016, p. 1.
[92] J. Hoffmann, Analyzing search topology without running any search: on the connection between causal graphs and h^+, J. Artif. Intell. Res. 41 (2011) 155–229.
[93] P. Laborie, M. Ghallab, Planning with sharable resource constraints, in: S. Mellish (Ed.), Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI'95, Morgan Kaufmann, Montreal, Canada, 1995, pp. 1643–1649.
[94] J. Koehler, Planning under resource constraints, in: H. Prade (Ed.), Proceedings of the 13th European Conference on Artificial Intelligence, ECAI'98, Wiley, Brighton, UK, 1998, pp. 489–493.