Towards Clause-Learning State Space Search: Learning to Recognize Dead-Ends
(Technical Report)

Marcel Steinmetz and Jörg Hoffmann
Saarland University
Saarbrücken, Germany
{steinmetz,hoffmann}@cs.uni-saarland.de

Abstract
We introduce a state space search method that identifies dead-end states, analyzes the reasons for failure, and learns to avoid similar mistakes in the future. Our work is placed in classical planning. The key technique 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. Starting from a simple initial $C$, we augment search to identify unrecognized dead-ends $s$, where $h^C(s) < \infty$. We design methods analyzing the situation at such $s$, adding new conjunctions into $C$ to obtain $h^C(s) = \infty$, thus learning to recognize $s$ as well as similar dead-ends search may encounter in the future. We furthermore learn clauses $\phi$ where $s' \not\models \phi$ implies $h^C(s') = \infty$, to avoid the prohibitive 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 clause learning in SAT, learning to refute search subtrees. Our experiments show that this can be quite powerful. On problems where dead-ends abound, the learning reliably reduces the search space by several orders of magnitude.

Introduction
The ability to analyze conflicts, and to learn clauses that avoid similar mistakes in the future, is a key ingredient to the success of SAT solvers (e. g. (Marques-Silva and Sakallah 1999; Moskewicz et al. 2001; Eén and Sörensson 2003)). To date, there has been no comparable framework for state space search. Part of the reason of course is that conflicts, quintessential in constraint reasoning, play a much less prevalent role in transition systems. Nevertheless, defining a “conflict” to be a dead-end state -- a state not part of any solution -- conflicts are ubiquitous in many applications. For example, bad decisions often lead to dead-ends in oversubscription planning (e. g. (Smith 2004; Gerevini et al. 2009; Domshlak and Mirkis 2015)), in planning with limited resources (e. g. (Haslum and Geffner 2001; Nakhost, Hoffmann, and Müller 2012; Coles et al. 2013)), and in single-agent puzzles like Sokoban (e. g. (Junghanns and Schaeffer 1998)) or Solitaire card games (e. g. (Bjarnason, Tadepalli, and Fern 2007)). In explicit-state model checking of safety properties (e. g. (Behrmann et al. 2002; Holzmann 2004; Edelkamp, Lluch-Lafuente, and Leue 2004)), a dead-end is any state from which the error property cannot be reached.

We introduce a state space search method that, at a high level, shares many features with clause learning in SAT. Our work is placed in classical planning, but in principle the approach applies to reachability checking in other transition system models as well. It requires a state-variable based representation, with transition rule models suitable for critical-path heuristics. We briefly discuss this at the end of the paper. The paper is aimed at being accessible to researchers not only from planning, but also from related areas.

A dead-end in planning is a state from which the goal cannot be reached. Dead-end detection has been given some attention in probabilistic and non-deterministic planning (Kolobo, Mausam, and Weld 2012; Muise, McIlraith, and Beck 2012), where computationally expensive methods (e. g. using classical planning as a sub-procedure) may pay off. But very little has been done about dead-end detection in classical planning. Heuristic functions have been intensely investigated, and most of them have the ability to recognize dead-end states $s$, returning heuristic value $h(s) = \infty$ if $s$ is unsolvable even in the relaxation underlying $h$. But this has been treated as a by-product of estimating goal distance.

Recent work (Bäckström, Jonsson, and Ståhlberg 2013; Hoffmann, Kissmann, and Torralba 2014) has started to break with this tradition, introducing the concept of unsolvability heuristics, dedicated to dead-end detection. An unsolvability heuristic returns either $\infty$ (“dead-end”) or $0$ (“don’t know”), and serves as an efficiently testable sufficient criterion for unsolvability. Concrete unsolvability heuristics have been designed based on state-space abstractions, specifically projections (pattern databases (Edelkamp 2001)) and merge-and-shrink abstractions (Helmer et al. 2014). The empirical results are impressive, especially for merge-and-shrink which convincingly beats state-of-the-art BDD-based planning techniques (Torralba and Alcázar 2013) on a suite of unsolvable benchmark tasks. Yet, comparing these techniques to conflict detection methods in other areas, they are quite limited in that they are completely disconnected from the actual search, establishing the unsolvability heuristics once and for all in a pre-process. Can we instead refine the unsolvability heuristic during search, learning from the dead-ends encountered?
Recent research on classical planning heuristics has laid the basis for answering this question in the affirmative, through critical-path heuristics $h^C$ relative to a set $C$ of conjunctions that can be chosen freely.

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 $h^m$ (Haslum and Geffner 2000), the atomic conjunctions are all conjunctions of size $\leq m$, where $m$ is a parameter. As part of recent works (Haslum 2009; 2012; Keyder, Hoffmann, and Haslum 2014), this was extended to arbitrary sets $C$ of atomic conjunctions. Following Hoffmann and Fickert (2015), we denote the generalized heuristic with $h^C$. A well-known and simple result is that, for sufficiently large $m$, $h^m$ delivers perfect goal distance estimates (simply set $m$ to the number of state variables). As a corollary, for appropriately chosen $C$, $h^C$ recognizes all dead-ends. Our idea thus is to refine $C$ during search, based on the dead-ends encountered.

We start with a simple initialization of $C$, to the set of singleton conjunctions. During search, components $S$ of un-recognized dead-ends, where $h^C(s) < \infty$ for all $s \in S$, are identified (become known) when all their descendants have been explored. We show how to refine $h^C$ on such components $S$, adding new conjunctions into $C$ in a manner guaranteeing that, after the refinement, $h^C(s) = \infty$ for all $s \in S$. The refined $h^C$ has the power to generalize to other dead-ends search may encounter in the future, i.e., refining $h^C$ on $S$ may lead to recognizing also other dead-end states $s' \notin S$. In our experiments, this happens at massive scale.1

It is known that computing critical-path heuristics over large sets $C$ is (polynomial-time yet) computationally expensive. Recomputing $h^C$ on all search states often results in prohibitive runtime overhead. We tackle this with a form of clause learning inspired by Kolobov et al.’s (2012) SixthSense. For a dead-end state $s$ on which $h^C$ was just refined, we learn a minimal clause $\phi$ by starting with the disjunction of facts $p$ false in $s$, and iteratively removing $p$ while preserving $h^C(s) = \infty$. When testing whether a new state $s'$ is a dead-end, we first evaluate the clauses $\phi$, and invoke the computation of $h^C(s')$ only in case $s'$ satisfies all $\phi$.

Arranging these techniques in a depth-first search, we obtain an algorithm approaching the elegance of clause learning in SAT: When a subtree is fully explored, the $h^C$-refinement and clause learning (1) learns to refute that subtree, (2) enables backjumping to the shallowest non-refuted ancestor, and (3) generalizes to other similar search branches in the future. Our experiments show that this can be quite powerful. On planning with limited resources, relative to the same search but without learning, our technique reliably reduces the search space by several orders of magnitude.

Some proofs are moved out of the main text. They are available in an appendix.

---

1Note that this ability to generalize is a major difference to well-explored methods refining a value function based on Bellman value updates during search (e.g. (Korf 1990; Reinefeld and Marsland 1994; Barto, Bradtke, and Singh 1995; Bonet and Geffner 2006)).

**Background**

We use the STRIPS framework for classical planning, where state variables (facts) are Boolean, and action preconditions/effects are conjunctions of literals (only positive ones, for preconditions). We formulate this in the common fact-set based fashion. A planning task $\Pi = \langle F, A, I, G \rangle$ consists of a set of facts $F$, a set of actions $A$, an initial state $I \subseteq F$, and a goal $G \subseteq F$. Each $a \in A$ has a precondition $pre(a) \subseteq F$, an add list $add(a) \subseteq F$, and a delete list $del(a) \subseteq F$. (Action costs are irrelevant to dead-end detection, so we assume unit costs throughout.)

In action preconditions and the goal, the fact set is interpreted as a conjunction; we will use the same convention for the conjunctions in the set $C$, i.e., the $c \in C$ are fact sets $c \subseteq F$. The add and delete lists are just instructions which facts to make true respectively false. A state $s$, in particular the initial state $I$, is a set of facts, namely those true in $s$ (the other facts are assumed to be false). This leads to the following definition of the state space of a task $\Pi = \langle F, A, I, G \rangle$, as a transition system $\Theta^\Pi = \langle S, T, I, S_G \rangle$. $S$ is the set of states, i.e., $S = 2^F$. The transitions $T \subseteq S \times A \times S$ are the triples $(s, a, s'[a])$ where $a$ is applicable to $s$, i.e., $pre(a) \subseteq s$, and $s'[a] := (s \setminus del(a)) \cup add(a)$. $I$ is the task’s initial state, $S_G \subseteq S$ is the set of goal states, i.e., those $s \in S$ where $G \subseteq s$. A plan for state $s$ is a transition path from $s$ to some $t \in S_G$; a plan for $\Pi$ is a plan for $I$. A dead-end is a state for which no plan exists.

Viewing $\Theta^\Pi$ as a directed graph over states, given a subset $S' \subseteq S$ of states, by $\Theta^\Pi|_{S'}$ we denote the subgraph induced by $S'$. If there is a path in $\Theta^\Pi|_{S'}$ from $s$ to $t$, then we say that $t$ is reachable from $s$ in $\Theta^\Pi|_{S'}$.

A heuristic is a function $h : S \rightarrow N^+_0 \cup \{\infty\}$. Following Hoffmann et al. (2014), we define an unsolvability heuristic, also dead-end detector, as a function $u : S \rightarrow \{0, \infty\}$. The interpretation of $u(s) = \infty$ will be “dead-end”, that of $u(s) = 0$ will be “don’t know”. We require that $u(s) = \infty$ only if $s$ really is a dead-end: States flagged as dead-ends will be pruned by the search, so the dead-end detector must be sound (no false positives). The dead-end detector may, on the other hand, return $u(s) = 0$ even though $s$ is actually a dead-end (false negatives possible). This is necessarily so: obtaining a perfect dead-end detector (one that returns $u(s) = \infty$ if and only if $s$ is a dead-end) incurs solving the input planning task in the first place. Our central idea in this work is to refine an (initially simple) dead-end detector during search, in a manner recognizing more dead-ends. Namely, we say that a dead-end $s$ is recognized if $u(s) = \infty$, and that $s$ is unrecongnized otherwise.

The family of critical-path heuristics, which underly Graphplan (Blum and Furst 1997) and were formally introduced by Haslum and Geffner (2000), estimate goal distance through the relaxation assuming that, from any goal set of facts (interpreted as a fact conjunction that needs to be achieved), it suffices to achieve the most costly subgoal (sub-conjunction): intuitively, the most costly atomic subgoal, left intact by the underlying relaxation. The family is parameterized by the set of atomic subgoals considered. The traditional formulation uses all subgoals of size at most $m$,
where $m \in \mathbb{N}$ is the parameter and the heuristic function is denoted $h^m$. As recently pointed out by Hoffmann and Fickert (2015) though, there is no need to restrict the atomic subgoals in this manner. One can use an arbitrary set $C$ of fact conjunctions as the atomic subgoals.

Formally, for a fact set $G$ and action $a$, define the regression of $G$ over $a$ as $R(G, a) := (G \setminus \text{add}(a)) \cup \text{pre}(a)$ in case that $\text{add}(a) \cap G \neq \emptyset$ and $\text{del}(a) \cap G \neq \emptyset$; otherwise, the regression is undefined and we write $R(G, a) = G$. By $A(G)$ we denote the set of actions where $R(G, a) \neq G$. Let $C$ be any set of conjunctions. The generalized critical-path heuristic $h^C(s)$ is defined through $h^C(s) := h^C(s, G)$ where

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

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 $G$ 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.

Intuitively, Equation 1 says that, if a subgoal $G$ is already true then its estimated cost is 0 (top case); if a subgoal is atomic then we need to support it with the best possible action, whose cost is computed recursively (middle case); if a subgoal is not atomic then we estimate its cost through that of the most costly atomic subgoal (bottom case). It is easy to see that $h^C$ generalizes $h^m$, as the special case where $C$ consists of all conjunctions of size $\leq m$.

As we are interested only in dead-end detection, not goal distance estimation, 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. Note that $u^C(s) = \infty$ occurs (only) due to empty minimization in the middle case of Equation 1, i.e., if every possibility to achieve the global goal $G$ incurs at least one atomic subgoal not supported by any action.

Similarly as for $h^m$, $u^C$ can be computed (solving Equation 1) in time polynomial in $|C|$ and the size of II. It is known that, in practice, $h^m$ is reasonably fast to compute for $m = 1$, consumes substantial runtime for $m = 2$, and is mostly infeasible for $m = 3$. The behavior is similar when using arbitrary conjunction sets $C$, in the sense that large $C$ causes similar issues as $h^m$ for $m > 1$. As hinted, we will use a clause-learning technique to alleviate this.

For appropriately chosen $m$, $h^m$ returns the exact goal distance, and therefore, for appropriately chosen $C$, $h^C$ recognizes all dead-ends. But how to choose $C$? This question has been previously addressed only in the context of partial delete-relaxation heuristics (Haslum 2012; Keyder, Hoffmann, and Haslum 2014; Hoffmann and Fickert 2015), which are basically built on top of $h^C$. All known methods learn $C$ prior to search, by iteratively refining a relaxed plan for the initial state. Once this refinement process stops, the same set $C$ is then used throughout the search. This makes sense for goal distance estimation, as modifying $C$ during search would yield a highly volatile, potentially detrimental, heuristic. When using $C$ for dead-end detection, this difficulty disappears. Consequently, we learn $C$ based on information that becomes available during search.

An Illustrative Example

We give an example walkthrough to illustrate the overall search process, and how the learning (1) refutes completed parts of the search, (2) leads to backjumping, and (3) generalizes to other similar search branches. This can be shown in a simple transportation task with fuel consumption. Consider Figure 1 (left). The planning task $\Pi = \{F, A, I, G\}$ has facts $tA, tB, tC$ encoding the truck position, $f0, f1, f2$ encoding the remaining fuel, and $pA, pB, pC, p2B, p2C, p2d$ encoding the positions of the two packages. There are actions to drive from $X$ to $Y$, given remaining fuel $Z$, consuming 1 fuel unit; to load package $X$ at $Y$; and to unload package $X$ at $Y$.

These actions have the obvious preconditions and add/delete lists, e.g., $\text{drive}(A, B, 2)$ has precondition $\{tA, f2\}$, add list $\{tB, f1\}$ and delete list $\{tA, f2\}$. The initial state is $I = \{tA, f2, pB, p2B\}$. The goal is $G = \{pC, p2B\}$.

The task is unsolvable because we do not have sufficient fuel. To determine this result, 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 $B$ vs. driving first to $C$. Using our methods, the learning on one of these branches immediately excludes the other branch.

Say we run a depth-first search. Our set $C$ of conjunctions is initialized to the singletons $C = \{p\} | p \in F$. Given this, $u^C(s) = \infty$ if $h^C(s) = \infty$. As regression over singleton subgoals ignores the delete lists, this is equivalent to the goal being (delete-)relaxed-reachable. Simplifying a bit, the reader may think of this as “ignoring fuel consumption” in what follows. From $I$, the goal is relaxed-reachable, and we get $u^C(I) = 0$. So $I$ is not detected to be a dead-end, and we expand it, generating $s_1$ (drive to B) and $s_2$ (drive to C) in Figure 1 (right). From these states, too, the goal is relaxed-reachable so $u^C(s_1) = u^C(s_2) = 0$. Say we expand $s_2$ next, generating $s_3$ (load $p_1$) and $s_4$ (drive back to $A$). We get $u^C(s_3) = 0$ similarly as before. But we have $u^C(s_4) = \infty$ because, in $s_4$, there is no fuel left so the goal has now become relaxed-unreachable (in fact, $s_4$ is trivial to recognize as a dead-end because it has no applicable actions). Search proceeds by expanding $s_3$, generating a transition back to $s_1$ (unload $p_1$), and generating $s_5$ (drive back to $A$). Like $s_4$, $s_5$ is recognized to be a dead-end, $u^C(s_5) = \infty$. Thus the descendants of $s_3$ have been fully explored, and $s_3$ now becomes a known, yet unrecognized, dead-end. In other words: search has encountered a conflict.

We call the learning process on $s_3$, the aim being to analyze the conflict at $s_3$, and refine $C$ in a manner recognizing $s_3$. This process, as we will show later on when explaining the technical details, ends up selecting the single conjunction $c = \{tA, f1\}$. We set $C := C \cup \{c\}$, thus refining our dead-end detector $u^C$. This yields $u^C(s_3) = \infty$: On the one hand, $c$ is required to achieve the goal (regressing from the goal fact $p1C$ yields the subgoal $tC$, regressing from which yields the subgoal $c$). On the other hand, $u^C(s_3, c) = \infty$, i.e., $c$ is detected by $u^C$ to be unreachable from $s_3$, because regressing from $c$ yields the subgoal $f2$. (When using singleton conjunctions only, this is overlooked because each element of $c$, i.e. $tA$ respectively $f1$ on its own, is reachable.
Our method applies to search algorithms using open & closed lists (A*, greedy best-first search, ...). Depth-first search, which we use in practice, is a special case with particular properties, discussed at the end of this section.

Identifying Conflicts

Our method applies to search algorithms using open & closed lists (A*, greedy best-first search, ...). Depth-first search, which we use in practice, is a special case with particular properties, discussed at the end of this section.

Consider the top half of Algorithm 1, a generic forward search using dead-end pruning at node generation time. We assume here some unsolvability heuristic u that can be refined on dead-ends. The question we tackle is, how to identify the conflicts in the first place? In a complete manner, guaranteeing to identify all known dead ends, i.e., all states the search has already proved to be dead-ends?

A simple attempt towards answering these questions is, “if all successors of s are already known to be dead-ends, then s is known to be a dead-end as well”. This would lead to a simple bottom-up dead-end labeling approach. However, this is incomplete, due to cycles: if states s1 and s2 are dead-ends but have outgoing transitions to each other, then neither of the two will ever be labeled. Our labeling method thus involves a complete lookahead to currently reached states.

Let us spell this out in detail. First, when is a dead-end "known" to the search? By definition, state s is a dead-end iff no state t reachable from s is a goal state. Intuitively, the search "knows" this is so, i.e. the search has proved this already, i.e. all these states t have already been explored. We thus define "known" dead-ends as follows:

Definition 1. Let Open and Closed be the open and closed lists at some point during the execution of Algorithm 1. Let s ∈ Closed be a closed state, and let R[s] := {t | t reachable from s in Θ[Open ∪ Closed]}. We say that s is a known dead-end if R[s] ⊆ Closed.

We apply this definition to closed states only because, if s itself is still open, then trivially its descendants have not yet been explored and R[s] ⊈ Closed.

It is easy to see that the concept of "known dead-end" does capture exactly our intentions:

Proposition 1. Let s be a known dead-end during the execution of Algorithm 1. Then s is a dead-end.

Proof. Assume to the contrary that s = s0 → s1 → · · · → sn ∈ S is a plan for s. Let i be the smallest index so that si ∈ Closed. Algorithm 1 stops upon expanding a goal state, so such i exists. Because sj ∈ Closed, j > 0. But then, si−1 has been expanded; and as si is not a dead-end state, u(sj) ≡ ∞; so si necessarily is contained in Open. Therefore, R[s] ⊈ Closed in contradiction.

Vice versa, if R[s] ⊈ Closed, then some descendants of s have not yet been explored, so the search does not know whether or not s is a dead-end.

So how to identify the known dead-ends during search? One could simply re-evaluate Definition 1 on every closed state after every state expansion. As one would expect, this can be done much more effectively. Consider the bottom part of Algorithm 1, i.e., the CheckAndLearn(s) procedure.

We maintain state labels (Boolean flags) indicating the known dead-ends. At first, no state is labeled. In the top-level invocation of CheckAndLearn(s), s cannot yet be labeled.
Algorithm 1: Generic forward search algorithm with dead-end identification and learning.

Procedure ForwardSearch(\(\Pi\))
\[\text{Open} := \{I\}, \text{Closed} := \emptyset;\]
while \(\text{Open} \neq \emptyset\) do
\[\text{select } s \in \text{Open};\]
if \(I \subseteq s\) then
\[\text{return path from } I \text{ to } s;\]
\(\text{Closed} := \text{Closed} \cup \{s\};\)
for all \(a \in A\) applicable to \(s\) do
\[s^\prime := s[a];\]
if \(s^\prime \in \text{Closed}\) then continue;
if \(u(s^\prime) = \infty\) then continue;
\(\text{Open} := \text{Open} \cup \{s^\prime\};\)
CheckAndLearn(\(s\));
return unsolvable;

Procedure CheckAndLearn(\(s\))
\/* loop detection */
if \(s\) is labeled as dead end then
\(\text{return};\)
\(\mathcal{R}[s] := \{t \mid t \text{ reachable from } s \text{ in } \mathcal{H}\}_{\text{Open} \cup \text{Closed}};\)
if \(\mathcal{R}[s] \subseteq \text{Closed}\) then
label \(s;\)
\/* refinement (conflict analysis) */
refine \(u\) s.t. \(u(t) = \infty\) for every \(t \in \mathcal{R}[s];\)
\/* backward propagation */
for every parent \(t\) of \(s\) do
\[\text{CheckAndLearn}(t);\]

as \(s\) was just expanded and only closed states are labeled. The label check at the start of CheckAndLearn(\(s\)) is needed only for loop detection in recursive invocations cf. below.

The definition of, and check on, \(\mathcal{R}[s]\) correspond to Definition 1. For \(t \in \mathcal{R}[s]\), as \(t\) is reachable from \(s\) we have \(\mathcal{R}[t] \subseteq \mathcal{R}[s]\) and thus \(\mathcal{R}[t] \subseteq \text{Closed}\). If the latter was true already prior to expansion of \(s\), then \(t\) is already labeled, else \(t\) is now a new known dead-end (and will be labeled in the recursion, see below). Some \(t\) may be recognized already, \(u(t) = \infty\). If that is so for all \(t \in \mathcal{R}[s]\), then there is nothing to do and the refinement step is skipped. The refinement is applied to known, yet unrecognized, dead-ends.

The recursion, backward propagation on the parents of \(s\), is needed to identify all dead-ends known at this time. Observe here that the ancestors of \(s\) are exactly those states \(t\) whose reachability information may have changed when expanding \(s\).3 As \(s\) was open beforehand, any ancestor \(t\) is not yet labeled: it had the open descendant \(s\) until just now. A change to \(t\)’s label may be required only if \(s\) was newly labeled. Hence the recursive calls are only needed for such \(s\), and \(\text{Closed}\) is an obvious upper bound on the number of recursive invocations, even if the state space contains cycles.

In short, we label known dead-end states bottom-up along forward search transition paths, applying a full lookahead on the current search space in each. This is sound and complete relative to the dead-end information available during search:

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

**Proof (sketch).** Soundness, i.e., \(t\) labeled \(\Rightarrow\) \(t\) is a known dead-end, holds because \(\mathcal{R}[t] \subseteq \mathcal{R}[s] \subseteq \text{Closed}\) at the time of labeling. Completeness, i.e., \(t\) is a known dead-end \(\Rightarrow\) \(t\) labeled, holds because the recursive invocations of CheckAndLearn(\(t\)) will reach all relevant states.

Reconsider Figure 1 (right). After expansion of \(s_3\), the call to CheckAndLearn(\(s_3\)) constructs \(\mathcal{R}[s_3] = \{s_3, s_1\}\), and finds that \(\mathcal{R}[s_3] \subseteq \text{Closed}\). 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 \(\mathcal{R}[s]\) are already recognized so the refinement step is skipped. The recursive calls on the parents of \(s_1\), CheckAndLearn(\(s_3\)) and CheckAndLearn(\(I\)), find that \(s_3\) is already labeled, respectively that \(\mathcal{R}[I] \subseteq \text{Closed}\), so the procedure terminates here.

Note in this example that, even though we run a depth-first search (DFS), we require the open and closed lists. Otherwise, we couldn’t prove \(s_3\) to be a dead-end: \(s_3\) has a transition to its parent \(s_1\), so it may have a solution via \(s_1\). Excluding that possibility requires the open and closed lists, keeping track of the search space as a whole.4 Therefore, the “depth-first” search herein uses Algorithm 1, ordering the open list by decreasing distance from the initial state.

The key advantage of DFS in our setting is that, through fully exploring the search space below a state, it quickly identifies dead-ends. Experimenting with other search algorithms, in many cases few dead-ends became known during search, so not enough information was learned.

DFS is particularly elegant on acyclic state spaces, where matters become easier and more similar to backtracking in constraint-solving problems like SAT (whose search spaces are acyclic by definition). Acyclic state spaces naturally occur, e.g., if every action consumes some 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 \(s\). Thus, instead of the complex CheckAndLearn procedure required in the general case, we can simply refine \(u\) on \(s\) at this point. In particular, we don’t need a closed list, and can use a classical DFS.

With respect to the use of a closed list for duplicate pruning, observe that, as \(u\) learns to refute \(\mathcal{R}[s]\) – in the DFS acyclic case, exactly the subtree below \(s\) or \(u\) subsumes the duplicate pruning afforded by a closed list. It will, typically, surpass this pruning by far due to generalization. We thus get duplicate pruning “for free” in the DFS acyclic case, and in the general case we can remove \(\mathcal{R}[s]\) from the closed list without losing duplicate-pruning power.

3Note the special case of ancestors \(t\) contained in \(\mathcal{R}[s]\). These are exactly those \(t \in \mathcal{R}[s]\) where \(\mathcal{R}[t] \not\subseteq \text{Closed}\) before expanding \(s\), but \(\mathcal{R}[t] \subseteq \text{Closed}\) afterwards. Such \(t\) will be labeled in the recursion. We cannot label them immediately (along with \(s\) itself) as some other ancestor of \(s\) may be connected to \(s\) only via such \(t\).

4It may be worth considering functions \(u\) disallowing a state to be solved via its parent, thus detecting dead-ends not at a global level but at the scope of a state’s position in the search. It remains a research question how such \(u\) can actually be obtained.
Conflict Analysis & Refinement for \( u^C \)

We now tackle the refinement step in Algorithm 1, for the dead-end detector \( u = u^C \). Given \( R[s] \) where all \( t \in R[s] \) are dead-ends, how to refine \( u^C \) to recognize all \( t \in R[s] \)? The answer is rather technical, and the reader not interested in details may skip forward to the next section.

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 \( t \in R[s] \). But how to find such \( X \)? Our key to answering this question is the specific circumstances guaranteed by the CheckAndLearn\((s)\) procedure, namely what we will refer to as the recognized neighbors property: (*) For every transition \( t \rightarrow t' \) where \( t \in R[s] \), either \( t' \in R[s] \) or \( u^C(t') = \infty \). This is because \( R[s] \) contains only closed states, so it contains all states \( t \) reachable from \( s \) except for those where \( u^C(t) = \infty \). For illustration, consider Figure 1: \( R[s_3] = \{s_3, s_5\} \), and (*) is satisfied because the neighbor states \( s_5 \) and \( s_4 \) are already recognized by \( u^C \) (using the singleton conjunctions only).

Let \( \hat{S} \) be any set of dead-ends with the recognized neighbors property, i.e., for every transition \( s \rightarrow t \) where \( s \in \hat{S} \), either (a) \( t \in \hat{S} \) or (b) \( u^C(t) = \infty \). We denote the set of states \( t \) with (b), the neighbors, by \( \hat{T} \) (e.g. \( \hat{T} = \{s_4, s_5\} \) for \( \hat{S} = R[s_3] \)). Similarly as in Equation 1, we use \( u^C(s, G) \) to denote the \( u^C \)-value of subgoal fact set \( G \). We use \( h^*(s, G) \) to denote the exact cost of achieving \( G \) from \( s \).

Our refinement method assumes as input the \( u^C \) information for \( t \in T \), i.e., the values \( u^C(t, c) \) for all \( t \in T \) and \( c \in C \). We compute this at the start of the refinement procedure.\(^3\) Thanks to this information, in contrast to known \( C \)-refinement methods like Haslum’s (2012), we do not require any intermediate recomputation of \( u^C \) during the refinement. Instead, our method (Algorithm 2) uses the \( u^C \) information for \( t \in \hat{T} \) to directly pick suitable conjunctions \( x \) for the desired set \( X \). The method is based on the following characterizing condition for \( u^C \) dead-end recognition:

**Lemma 1.** Let \( s \) be a state and let \( G \subseteq F \). Then \( u^C(s, G) = \infty \) if and only if there exists \( c \in C \) such that:

(i) \( c \in G \) and \( c \not\subseteq s \); and

(ii) for every \( a \in A[c], u^C(s, R(c, a)) = \infty \).

**Proof.** \( \Rightarrow \) By definition of \( u^C \), there must be a conjunction \( c \in C \) so that \( c \subseteq G \) and \( u^C(s, c) = \infty \). This in turn implies that \( c \not\subseteq s \), and that \( u^C(s, R(c, a)) = \infty \) for every \( a \in A[c] \).

\( \Leftarrow \) As \( c \subseteq G \), \( u^C(s, G) \geq u^C(s, c) \). As \( c \not\subseteq s \), \( u^C(s, c) = \min_{a \in A[G]} u^C(s, R(c, a)) \). For every \( a \in A[G], u^C(s, R(c, a)) = \infty \), so \( u^C(s, c) = \infty \).

Given this, to obtain \( u^{C \cup X}(s) = u^{C \cup X}(s, G) = \infty \) for \( s \in \hat{S} \), we can pick some conjunction \( c \subseteq G \) but \( c \not\subseteq s \) (Lemma 1 (i)), and, recursively, pick an unreachable conjunction \( c' \subseteq R(c, a) \) for each supporting action \( a \in A[c] \) (Lemma 1 (ii)). For that to be possible, of course, \( c \) must actually be unreachable, i.e., it must hold that \( h^*(s, c) = \infty \).

\(^3\)One could cache this information during search, 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.

**Algorithm 2:** Refining \( C \) for \( \hat{S} \) with recognized neighbors \( T, C \) and \( X \) are global variables.

**Procedure Refine(G)**

\[
\begin{align*}
x & := \text{ExtractX}(G); \\
X & := X \cup \{x\}; \\
& \text{for} \ a \in A[x] \ \text{where} \ ex \ s.t. u^C(s, R(x, a)) = 0 \ \text{do} \\
& \quad \text{if there is no} \ x' \in X \ s.t. x' \subseteq R(x, a) \ \text{then} \\
& \quad \quad \text{Refine}(R(x, a)); \\
& \end{align*}
\]

**Procedure ExtractX(G)**

\[
\begin{align*}
x & := \emptyset; \\
& /* \text{Lemma 2 (i)} \ */ \\
& \text{for every} \ t \in \hat{T} \ \text{do} \\
& \quad \text{select} \ c_0 \in C \ \text{s.t.} \ c_0 \subseteq G \ \text{and} \ u^C(t, c_0) = \infty; \\
& \quad x := x \cup c_0; \\
& /* \text{Lemma 2 (i)} \ */ \\
& \text{for every} \ s \in \hat{S} \ \text{do} \\
& \quad \text{if} \ x \subseteq s \ \text{then} \\
& \quad \quad \text{select} \ p \in G \ \text{s.t.} \ x := x \cup \{p\}; \\
& \end{align*}
\]

return \( x \);

But this is PSPACE-complete to decide. As we already know that the states \( s \in \hat{S} \) are dead-ends, for (i) we can in principle use \( c := G \), and for (ii) we can in principle use \( c' := R(c, a) \). But this trivial solution would effectively construct a full regression search tree from \( G \), selecting conjunctions corresponding to the regressed states. We instead need to find small subgoals that are already unreachable. This is where the recognized neighbors property helps us.

Consider Algorithm 2. The top-level call of \texttt{Refine} is on \( G := \hat{G} \), with the global variable \( X \) initialized to \( \emptyset \). The procedure mirrors the structure of Lemma 1, selecting first an unreachable conjunction \( x \) for the top-level goal, then doing the same recursively for the regressed subgoals. The invariant required for this to work is that \( G \) is \( \hat{S} \)-unsolvable, i.e., \( h^*(s, G) = \infty \) for all \( s \in \hat{S} \). This is true at the top level where \( G = \hat{G} \), and is satisfied provided that the same invariant holds for the \texttt{ExtractX} procedure, i.e., if \( G \) is \( \hat{S} \)-unsolvable then so is the sub-conjunction \( x \subseteq G \) returned.

\texttt{ExtractX}(G) first loops over all neighbor states \( t \), and selects \( c_0 \in C \) justifying that \( u^C(t, G) = \infty \). Observe that such \( c_0 \) always exists: For the top-level goal \( G = \hat{G} \), we know by construction that \( u^C(t, G) = \infty \), so by the definition of \( u^C \) there exists \( c_0 \subseteq G \) with \( u^C(t, c_0) = \infty \). For later invocations of \texttt{ExtractX}(G), we have that \( G = R(x, a) \), where \( x \) was constructed by a previous invocation of \texttt{ExtractX}(G). By that construction, there exists \( c'_0 \in C \) such that \( x \subseteq c'_0 \) and \( u^C(t, c'_0) = \infty \). Thus \( u^C(t, x) = \infty \), so \( u^C(t, G) = u^C(t, R(x, a)) = \infty \) and we can pick \( c_0 \subseteq R(x, a) = G \) with \( u^C(t, c_0) = \infty \) as desired.

The \texttt{ExtractX}(G) procedure accumulates the \( c_0 \), across the neighbor states \( t \), into \( x \). If the resulting \( x \) is not contained in any \( s \in \hat{S} \) then we are done, otherwise for each affected \( s \) we add a fact \( p \in G \setminus s \) into \( x \). Such \( p \) must exist because \( G \) is \( \hat{S} \)-unsolvable by the invariant. That invariant is preserved, i.e., \( x \) itself is, again, \( \hat{S} \)-unsolvable:
Lemma 2. Let $\hat{S}$ and $\hat{T}$ be as above, and let $x \subseteq F$. If
(i) for every $s \in \hat{S}$, $x \not\subseteq s$; and
(ii) for every $t \in \hat{T}$, there exists $c \in C$ such that $c \subseteq x$ and
$u^C(t, c) = \infty$;
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
$s = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ 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 (i). But then, $s_{i+1} \notin \hat{S}$, and thus
$s_{i+1} \in \hat{T}$ by definition. By (ii), there exists $c \subseteq x$ such
that $u^C(s_{i+1}, c) = \infty$. This implies that $h^*(s_{i+1}, c) = \infty$,
which implies that $h^*(s_{i+1}, x) = \infty$. The latter is in contra-
diction to the selection of the path. The claim follows.

Theorem 2. Algorithm 2 is correct:
(i) The execution is well defined, i.e., it is always possible
to extract a conflict $x$ as specified.
(ii) The algorithm terminates.
(iii) Upon termination, $u^{C \cup X}(s) = \infty$ for every $s \in \hat{S}$.

Proof (sketch). (i) holds with Lemma 2 and the arguments
sketched above. (ii) holds as every iteration adds a new con-
junction $x \not\in X$ and the number of possible conjunctions is
finite. (iii) follows from construction and Lemma 1.

In practice, to keep $x$ small, we use simple greedy strategies
in ExtractX, trying to select $c_0$ and $p$ shared by many $t$ and
$s$. Upon termination of Refine($\mathcal{G}$), we set $C := C \cup X$.

Consider again Figure 1, and the refinement process
on $\hat{S} = \mathcal{R}[s_3] = \{s_3, s_1\}$, with neighbor states $\hat{T} = \{s_4, s_5\}$. We
initialize $X = \emptyset$ and call Refine($\{p_1C, p_2B\}$). Calling
ExtractX($\{p_1C, p_2B\}$), $c_0 = \{p_1C\}$ is suitable for each
of $s_4$ and $s_5$, and is not contained in $s_3$ or $s_1$, so we may
return $x = \{p_1C\}$. Back in Refine($\{p_1C, p_2B\}$), we see
that $x$ may be achieved by unloading at $C$, and we need to
tackle the regressed subgoal through the recursive call
Refine($\{tC, p_1t\}$). ExtractX here returns $x = \{tC\}$, and
to exclude the drive from $A$ to $C$ we get the recursive call
Refine($\{tA, f_1\}$). In ExtractX now, the only choice of $c_0$
for each of $s_4$ and $s_5$ is $\{f_1\}$. As $f_1$ is contained in each of
$s_3$ and $s_4$, we need to also add the other part of $G$ into
$x$, ending up with $G = \{tA, f_1\}$: exactly the one con-
junction needed to render $u^{C \cup X}(s_3) = u^{C \cup X}(s_1) = \infty$, as
earlier explained. Indeed, the refinement process stops
here, because the actions achieving $x$, drive to $A$ from $B$ or
$C$, both incur the regressed subgoal $f_2$, for which we have
$u^C(s_3, \{f_2\}) = u^C(s_1, \{f_2\}) = \infty$.

It is instructive to consider the special case where $\hat{S}$ has no
neighbors at all, i.e., $\hat{T} = \emptyset$ and all transitions on $\hat{S}$ remain
inside $\hat{S}$. Then ExtractX($\mathcal{G}$) simply collects $p \in G \setminus s$
for each $s$. This is sound only because $\hat{S}$ has no neighbors;
otherwise, it would not be sound otherwise. Imagine, e.g., $\hat{S} = \{s\}$
where $s = \{p\}$, the goal is $\mathcal{G} = \{p, q\}$, and the only action
adds $q$ but deletes $p$. Then $\mathcal{G}$ is not reachable from $s$, but
$x = \{q\}$ is reachable from $s$, violating our invariant. This is
not a counter-example to Lemma 2 because, in this situation,
s has the neighbor $t = \{q\}$; as $h^C(t, \mathcal{G}) = \infty$ must hold,
either $c = \{p\}$ or $c = \{p, q\}$ must be contained in $C$, so
we would collect $x = \{p, q\}$ not $x = \{q\}$. Intuitively, given
$\hat{S}$ without neighbors, it suffices to collect $p \in G \setminus s$ into $x$
because necessarily such $x$ cannot be reached within $\hat{S}$, and
$\hat{S}$ contains everything that can be reached.

Clause Learning
Our clause learning method is based on a simple form of
“state minimization”, inspired by Kobolov et al. (2012)
work on SixthSense. Say we just evaluated $u^C$ on $s$ and
found that $u^C(s) = \infty$. Denote by $\phi := \bigvee_{p \in \mathcal{F} \setminus p}$ the dis-
junction of facts false in $s$. Then $\phi$ is a valid clause: for
any state $t$, if $t \not\models \phi$ then $u^C(t) = \infty$. Per se, this clause
is useless, as all states but $s$ itself satisfy $\phi$. This changes
when minimizing $\phi$, testing whether individual facts $p$
can be removed. For such a test, we set $s' := s \cup \{p\}$ and check
whether $u^C(s') = \infty$ (this is done incrementally, starting
from the computation of $u^C(s)$). If yes, $p$ can be removed.
Greedily iterating such removals, we obtain a minimal valid
clause. (Intuitively, a minimal reason for the conflict in $s$.)

As pointed out, the clauses do not have the same pruning
due to $u^C$. Yet they have a dramatic runtime advantage,
which is key to applying learning and pruning liberally.
We always evaluate the clauses prior to evaluating $u^C$. We
learn a new clause every time $u^C$ is evaluated and returns $\infty$.
We re-evaluate the states in $\mathcal{R}[s]$ during CheckAndLearn($s$),
closing those where $u^C = \infty$ (dead-ends not recognized
when first generated, but recognized now). Observe that,
given this, the call of CheckAndLearn($s$) directly forces a
jump back to the shallowest non-pruned ancestor of $s$. Fur-
thermore, while $u^C$ refutes $\mathcal{R}[s]$ after learning and thus
subsumes closed-list duplicate pruning for $\mathcal{R}[s]$, this property
is mute in practice as computing $u^C$ is way more time-
consuming than duplicate checking. That is not so for the
clauses, which also subsume duplicate pruning for $\mathcal{R}[s]$.

Experiments
Our implementation is in FD (Helmert 2006). For $u^C$, fol-
lowing Hoffmann and Fickert (2015), we use counters over
pairs $(c, a)$ where $c \in C$, $a \in A(c)$, and $R(c, a)$ does not contain
a fact mutex. As suitable for cyclic problems, we use DFS based
on Algorithm 1. We break ties (i.e. order chil-
dren in the DFS) using
by definition. By (ii), there exists $c \subseteq x$ such
that $u^C(s_{i+1}, c) = \infty$. This implies that $h^*(s_{i+1}, c) = \infty$,
which implies that $h^*(s_{i+1}, x) = \infty$. The latter is in contra-
diction to the selection of the path. The claim follows.

resource budget. We use the benchmarks by Nakhost et al. (2012), which are especially suited as they are controlled: the minimum required budget $b_{\text{min}}$ is known, and the actual budget is set to $W+b_{\text{min}}$. The parameter $W$ allows to control the frequency of dead-ends, and, therewith, empirical hardness. Values of $W$ close to 1.0 are notoriously difficult. In difference to Nakhost et al., like Hoffmann et al. (2014) we also consider values $W < 1$ where the tasks are unsolvable.

We use a cluster of Intel E5-2660 machines running at 2.20 GHz, with runtime (memory) limits of 30 minutes (4 GB). As a standard satisficing planner, we run FD’s greedy best-first dual-queue search with $h^\text{FF}$ and prefer operators, denoted “FD-$h^\text{FF}$”. We compare to blind breadth-first search, “Blind”, as a simple canonical method for proving unsolvability. We compare to Hoffmann et al.’s (2014) two most competitive configurations of merge-and-shrink (M&S) unsolvability heuristics, “Own-A” and “N100K M100K”, denoted here “OA” respectively “NM” for brevity. These represent the state of the art for proving unsolvability in planning, in that, on Hoffmann et al.’s benchmark suite, they dominate and outperform all other approaches, including state-of-the-art BDD-based planners (Torralba and Alcázar 2013). We run them as dead-end detectors in FD-$h^\text{FF}$ to obtain variants competitive also for satisficing planning.

We finally experiment with variants using both M&S and $u^C$ for dead-end detection. The two techniques are orthogonal, except that the recognized neighbors property requires the neighbors to be recognized by $u^C$. If dead-ends are also pruned by some other technique, our refinement method cannot analyze the conflict. We hence switch M&S pruning on only once the $\alpha$ limit has stopped the $u^C$ refinements. We use only NM, as OA M&S guarantees to recognize all dead-ends and a combination with learning would be void.

Figure 2 (left) gives coverage data. Compared to the state of the art, our approach (“DFS-CL”) easily outperforms the standard planner FD-$h^\text{FF}$. It is vastly superior in Rovers, and generally for budgets close to, or below, the minimum needed. The stronger planners using FD-$h^\text{FF}$ with M&S dead-end detection (not run in any prior work) are better than DFS-CL in NoMystery, worse in Rovers, and about on par in TPP. The combination with M&S (shown for $\alpha = 32$ and $\alpha = 128$, which yield best coverage here) typically performed as well as the corresponding base configurations, and sometimes outperforms both. We consider these to be very reasonable results for a completely new technique.

The really exciting news, however, comes not from comparing our approach to unrelated algorithms, but from comparing $\alpha = \infty$ vs. $\alpha = 1$. The former outperforms the latter dramatically. Observe that the only reason for this is generalization, i.e., refinements of $u^C$ on $\mathcal{R}[s]$ leading to pruning on states outside $\mathcal{R}[s]$. Without generalization, the search spaces for $\alpha = \infty$ and $\alpha = 1$ would be identical (including tie breaking). But that is far from being so. Generalization occurs at massive scale. It lifts a hopeless planner (DFS with $h^3$ dead-end detection) to a planner competitive with the state of the art in resource-constrained planning. Figure 2 (right) compares the search space sizes directly. On instances solved by both, the reduction factor min/geometric mean/maximum is: NoMystery 7.5/412.0/18117.9; Rovers 58.9/681.3/70401.5; TPP 1/54.4/1584.3. The only cases with no reduction are 6 TPP instances with $W \geq 1.2$.

We also experimented with the remainder of Hoffmann et al.’s (2014) unsolvable benchmarks (Bottleneck, Mystery, UnsPegsol, and UnsTiles), and the IPC benchmarks Airport, FreeCell, Mprime, and Mystery. In Mystery, performance is similar to the above. Elsewhere, often the search space is reduced substantially but this is outweighed by the runtime overhead. Interestingly, the DFS search itself sometimes has a strong advantage relative to FD-$h^\text{FF}$, with reductions up to 3 (2) orders of magnitude in Airport (FreeCell).

Conclusion

Our work pioneers dead-end learning in state-space search classical planning. Assembling pieces from prior work on critical-path heuristics and nogood-learning, and contributing methods for refining the dead-end detection during search, we obtain a form of state space search that approaches the elegance of clause learning in SAT. This opens a range of interesting research questions.

Can the role of clause learning, as opposed to $u^C$ refinement, become more prominent? 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, which as of now happens only for...
the growing set $C$ in $u^C$? Can we learn to refute solutions via a parent and thus allow to use classical DFS in general?

An exciting possibility is the use of $C$ as an unsolvability certificate, a concept much sought after in state space search. The certificate is efficiently verifiable because, given $C$, $u^C(C) = \infty$ can be asserted in polynomial time. The certificate is as large as the state space at worst, and will typically be exponentially smaller. From a state space with a certificate is as large as the state space at worst, and will be manageable. From a state space pruned using other methods, different refinement methods are needed.

Last not least, we believe the approach has great potential for game-playing and model checking problems where detecting dead-ends is crucial. 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 subgoal behavior. But more general logics (e.g. minimization to handle disjunctions) should be manageable.

Acknowledgments. This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1.

References


Haslum, P. 2009. $h^m(P) = h^l(P^m)$: Alternative characterisations of the generalisation from $h_{max}^l$ to $h_{max}^m$. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS’09), 354–357. AAAI Press.


of the 21st European Conference on Artificial Intelligence (ECAI’14). Prague, Czech Republic: IOS Press.


Proofs

**Theorem 1.** At the start of the while loop 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 \( \mathcal{R}[t] \subseteq \text{Closed} \), 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, \( \mathcal{R}[s] \subseteq \text{Closed} \) else, an open state is reachable from \( s \), and transitively is reachable from all ancestors of \( s \). In the case where \( \mathcal{R}[s] \subseteq \text{Closed} \), \( s \) 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 \( \mathcal{R}[t] \subseteq \text{Closed} \) (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 \( 1 \leq i < n \), \( \mathcal{R}[t_i] \subseteq \text{Closed} \). Obviously, for every \( 1 \leq i < n \), \( s \) 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.** Algorithm 2 is correct:

(i) The execution is well defined, i.e., it is always possible to extract a conflict \( x \) as specified.

(ii) The algorithm terminates.

(iii) Upon termination, \( u^{\mathcal{C}_\mathcal{L}}(\hat{X}) = \infty \) 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 \( x \subseteq \mathcal{G} \) 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 \( c_0 \subseteq C \) 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 \( c'_0 \nsubseteq R(x', a) \). However, \( c'_0 \subseteq x' \), so \( c'_0 \subseteq \text{add}(a) \), and thus \( a \in A[c'_0] \). 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 \( \text{ExtractX}(R(x, a)) \), \( x' \) 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 \( G \subseteq F \). Let \( s \in \hat{S} \) be arbitrary, and let \( x \in X \) be a conjunction with minimal \( h^\text{CUJX} \) value, i.e., let \( x \in X \) be so that for all \( x' \in X \): \( h^\text{CUJX}(s, x) \leq h^\text{CUJX}(s, x') \). Assume for contradiction that \( h^\text{CUJX}(s, x) < \infty \). Due to the construction of \( X \), it must be \( x \nsubseteq s \), meaning that there must be an action \( a \in A[x] \) with \( h^\text{CUJX}(s, R(x, a)) < h^\text{CUJX}(s, x) \) (definition of \( h^\text{CUJX} \)). However, the refinement algorithm ensures that \( X \) contains a conjunction \( x' \subseteq R(x, a) \): in the call to \( \text{Refine} \) where \( x \) is added to \( X \), the algorithm makes sure that for each action \( a \in A[x] \), either there is already a conjunction \( x' \in X \) so that \( x' \subseteq R(x, a) \), or it calls \( \text{Refine}(R(x, a)) \) which in turn adds a conjunction \( x' \subseteq R(x, a) \) to \( X \). But this is a contradiction to the \( h^\text{CUJX} \) minimality assumption: as there is a conjunction \( x' \in X \) with \( x' \subseteq R(x, a) \), it is \( h^\text{CUJX}(s, x') \leq h^\text{CUJX}(s, R(x, a)) < h^\text{CUJX}(s, x) \). This shows that \( h^\text{CUJX}(s, x) = \infty \) for every \( x \in X \), and for every state \( s \in \hat{S} \), and thus \( u^C(s) = \infty \) for every \( s \in \hat{S} \). \(\square\)