Towards Clause-Learning State Space Search: Learning to Recognize Dead-Ends

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; Één 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 Mirkin 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 Schaeffler 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 (Kolobov, Mauzam, 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 $s \in \bar{S}$, are identified (become known) when all their descendants have been explored. We show how to refine $h^C$ on such components $\bar{S}$, adding new conjunctions into $C$ in a manner guaranteeing that, after the refinement, $h^C(s) = \infty$ for all $s \in \bar{S}$. 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 $\bar{S}$ may lead to recognizing also other dead-end states $s' \notin \bar{S}$. In our experiments, this happens at massive scale.\footnote{Note 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; Reinifeld and Marsland 1994; Barto, Bradtke, and Singh 1995; Bonet and Geffner 2006)).}

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 replaced by proof sketches. Full proofs are available in a TR (Steinmetz and Hoffmann 2015).

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 $\Pi = (F, A, I, G)$ 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 $\Pi = (F, A, I, G)$, as a transition system $\Theta^\Pi = (S, I, G, \delta)$, $S$ the set of states, i.e., $S = 2^F$. The transitions $\delta \subseteq S \times A \times S$ are the triples $(s, a, s')$, where $a$ is applicable to $s$, i.e., $pre(a) \subseteq s$, and $s' = (s \setminus del(a)) \cup add(a)$. $I$ is the task’s initial state, $\delta \subseteq S$ 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 $t \in \delta \subseteq S$; a plan for $I$ 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 $h : S \rightarrow \mathbb{N}_0^+ \cup \{\infty\}$. 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 unrecognized 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 \( \text{del}(a) \cap G = \emptyset \); otherwise, the regression is undefined and we write \( R(G, a) = \perp \). By \( A[G] \) we denote the set of actions where \( R(G, a) \neq \perp \). 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

\[
h^C(s, G) = \begin{cases} 0 & \text{if } G \subseteq s \\ 1 + \max_{a \in A[G]} h^C(s, R(G, a)) & \text{otherwise} \\ \end{cases}
\]

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 \( \Pi \). 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, \( t0, t1, t2 \) encoding the remaining fuel, and \( p1A, p1B, p1C, p1t, p2A, p2B, p2C, p2t \) encoding the positions of the two packages. There are actions to \text{drive} from \( X \) to \( Y \), given remaining fuel \( Z \), consuming 1 fuel unit; to \text{load} package \( X \) at \( Y \); and to \text{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 \( I = \{tA, f2, p1B, p2C\} \). The goal is \( G = \{p1C, p2B\} \).

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 \text{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 \( C = \{p \mid p \in F\} \). Given this, \( u^C(s) = \infty \) iff \( h^C(s) = \infty \). As regression over singleton subgoals ignores the delete lists, this is equivalent to the goal being (delete-)relaxation-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 \( s1 \) (\text{drive} to \( B \)) and \( s2 \) (\text{drive} to \( C \)) in Figure 1 (right). From these states, too, the goal is relaxed-reachable so \( u^C(s_1) = u^C(s_2) = \infty \). Say we expand \( s2 \) next, generating \( s3 \) (\text{load} \( p1 \)) and \( s4 \) (\text{drive} back to \( A \)). We get \( u^C(s_3) = 0 \) similarly as before. But we have \( u^C(s_4) = \infty \) because, in \( s4 \), there is no fuel left so the goal has now become relaxed-unreachable (in fact, \( s4 \) is trivial to recognize as a dead-end because it has no applicable actions). Search proceeds by expanding \( s3 \), generating a transition back to \( s1 \) (\text{unload} \( p1 \)), and generating \( s5 \) (\text{drive} back to \( A \)). Like \( s4 \), \( s5 \) is recognized to be a dead-end, \( u^C(s_5) = \infty \). Thus the descendants of \( s3 \) have been fully explored, and \( s3 \) now becomes a known, yet unrecognized, dead-end. In other words: search has encountered a conflict.

We call the learning process on \( s3 \), the aim being to analyze the conflict at \( s3 \), and refine \( C \) in a manner recognizing \( s3 \). 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 \( s3 \), 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...
from \( s_3 \).) In other words, adding \( c \) to \( C \) lets \( u^C \) recognize that a single fuel unit is not enough to solve \( s_3 \). For the same reasons, \( u^C(1, s_1) = \infty \), so that, as advertised, \( u^C \) now (1) refutes the search space below \( s_3 \). Observe that this will also (2) cause the search to backjump across \( s_1 \) to \( I \).²

We next call the clause learning on \( s_3 \). This is optional in theory, as the clauses we learn are weaker dead-end detectors than \( u^C \). They are much more efficiently testable though, and are crucial in practice. The clause learning process minimizes “the commitments made by the conflict state”. For \( s_3 \), this detects that the position of \( p_2 \) is irrelevant to the unsolvability of \( s_3 \), that having 0 fuel does not help, and that it does not help if \( p_1 \) is anywhere other than \( C \). We thus learn the clause \( \phi = tA ∨ tC ∨ f2 ∨ p_1 \), which any non-dead-end state must satisfy. Observe that \( s_1 \not| \phi \), so we can now recognize \( s_3 \) as a dead-end without having to invoke \( u^C(1, s_1) \).

The only open state left is \( s_2 \). Yet, re-evaluating \( u^C \) on \( s_2 \), we find that, now, \( u^C(2, s_2) = \infty \). This is due to very similar reasoning as on \( s_3 \). The new conjunction \( c = \{ tA, f1 \} \) is required to achieve the goal (regressing, now, from the other goal fact \( p_2 \)), yet is unreachable from \( s_2 \) because it requires the subgoal \( f2 \). In other words, (3) the knowledge learned on the previous branch, in the form of the reasoning encapsulated by the extended conjunctions set \( C = \{ p \mid p \in F \} \cup \{ c \} \), generalizes to the present branch. (Note that \( s_2 \models \phi \), so here dead-end detection using the clauses is strictly weaker than \( u^C \).)

With \( s_2 \) pruned, there are no more open nodes, and unsolvability is proved without ever exploring the option to drive to \( C \) first. We could at this point continue, running the learning process on the now known-recognized dead-end \( I \): 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.

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 \( s_1 \) 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 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 if 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 \( \text{Open} \) and \( \text{Closed} \) be the open and closed lists at some point during the execution of Algorithm 1. Let \( s \in \text{Closed} \) be a closed state, and let \( R[s] := \{ t \mid t \text{ reachable from } s \in \Theta^{|\text{Open} \cup \text{Closed}} \} \). We say that \( s \) is a known dead-end if \( R[s] \subseteq \text{Closed} \).

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 \( \text{R}[s] \not\subseteq \text{Closed} \).

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 \( s = s_0 \rightarrow s_1 \cdots \rightarrow s_n \in \text{Closed} \) is a plan for \( s \). Let \( i \) be the smallest index so that \( s_i \not\in \text{Closed} \). Algorithm 1 stops upon expanding a goal state, \( s_0 \not\in \text{Closed} \), so such \( i \) exists. Because \( s_0 \in \text{Closed} \), \( i > 0 \). But then, \( s_{i-1} \) has been expanded; and as \( s_i \) is not a dead-end state, \( u(s_i) \neq \infty \); so \( s_i \) necessarily is contained in \( \text{Open} \). Therefore, \( \text{R}[s] \not\subseteq \text{Closed} \) in contradiction. \( \square \)

Vice versa, if \( \text{R}[s] \not\subseteq \text{Closed} \), 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,
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 \( R[t] \subseteq R[s] \subseteq \text{Closed} \) 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. \( \square \)

Recconsider Figure 1 (right). After expansion of \( s_3 \), the call to CheckAndLearn\((s_3)\) constructs \( R[s_3] = \{s_3, s_1\} \), and finds that \( R[s_3] \subseteq \text{Closed} \). 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 \( t \in R[s] \), all states in \( R[s_1] \) are already recognized so the refinement step is skipped. The recursive calls on the parents of \( s_1 \), CheckAndLearn\((s_3)\) and CheckAndLearn\((\text{I})\), find that \( s_3 \) is already labeled, respectively that \( R[\text{I}] \not\subseteq \text{Closed} \), 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. 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. Observe that, as \( u \) will learn to refute the entire subtree below \( s \), \( u \) subsumes (and, typically, surpasses by far) the duplicate pruning afforded by a closed list on other search branches.

For DFS on the general/cyclic case, a similar effect arises (only) if \( u \) is transitive, i.e., if \( u(s) = \infty \) implies \( u(t) = \infty \) for all states \( t \) reachable from \( s \) (as is the case for \( u^C \)). Then, upon learning to refute \( s \), we can remove \( R[s] \) from the closed list without losing duplicate-pruning power.
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. \( u^{C,X}(s) = \infty \) for all \( t \in R[s] \). But how to find such \( X \)? Our key to answering this question are 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: \( R[s_3] = \{s_3, s_4\} \), and (*) is satisfied because the neighbor states \( s_5 \) and \( s_4 \) are already recognized by \( u^C \) (using the singleton conjuncts 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 \( u^C(a(t)) = \infty \). 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 \( h^*(s, C) \) 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.\(^5\) 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 \( t \in T \) 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 \( u^C(s, G) = \infty \) iff there exists \( c \in C \) such that:

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

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

**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 \( u^C(s, c) = \infty \).

---

\(^5\)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.

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

\[
\begin{align*}
\text{Procedure } \text{ExtractX}(G) \\
& x := 0; \\
& \text{/* Lemma 2 (ii) */}  \\
& \text{for every } t \in T \text{ do} \\
& \text{select } c_0 \in C \text{ s.t. } c_0 \subseteq G \text{ and } u^C(t, c_0) = \infty; \\
& x := x \cup c_0; \\
& \text{/* Lemma 2 (i) */}  \\
& \text{for every } s \in \hat{S} \text{ do} \\
& \text{if } x \subseteq s \text{ then} \\
& \text{select } p \in G \setminus s; x := x \cup \{p\}; \\
& \text{return } x.
\end{align*}
\]

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 \text{Refine} is on \( G := G \), 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 \text{ExtractX} procedure, i.e., if \( G \) is \( \hat{S} \)-unsolvable then so is the sub-conjunction \( x \subseteq G \) returned.

\text{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 \text{ExtractX}(G), we have that \( G = R(x, a) \), where \( x \) was constructed by a previous invocation of \text{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 \text{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 \( s \in \hat{S}, x \nsubseteq s \); 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 \( s = s_0 \rightarrow s_1 \cdots \rightarrow s_n \) from \( s \) to some state \( s_0 \) 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 contradiction 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 conjunction \( x \notin X \) 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 \texttt{ExtractX}, trying to select \( c_0 \) and \( p \) shared by many \( t \) and \( s \). Upon termination of \texttt{Refine}(\( g \)), 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 \texttt{Refine}(\( \{p_1 C, p_2 B\} \)). Calling \texttt{ExtractX}(\( \{p_1 C, p_2 B\} \)), \( c_0 = \{p_1 C\} \) is suitable for each of \( s_4 \) and \( s_5 \), and is not contained in \( s_3 \) nor \( s_1 \), so we may return \( x = \{p_1 C\} \). Back in \texttt{Refine}(\( \{p_1 C, p_2 B\} \)), we see that \( x \) may be achieved by \texttt{unloading} at \( C \), and we need to tackle the regressed subgoal through the recursive call \texttt{Refine}(\( \{C, p_1\} \)). \texttt{ExtractX} here returns \( x = \{tC\} \), and to exclude the \texttt{drive} from \( A \) to \( C \) we get the recursive call \texttt{Refine}(\( \{tA, f\} \)). In \texttt{ExtractX} now, the only choice of \( c_0 \) for each of \( s_4 \) and \( s_5 \) is \( \{f\} \). As \( f \) is contained in each of \( s_4 \) and \( s_5 \), we need to also add the other part of \( G \) into \( x \), ending up with \( x = \{G = \{tA, f\}\} \); exactly the one conjunction needed to render \( u^C(s_3) = u^{C\cup X}(s_3) = \infty \), as earlier explained. Indeed, the refinement process stops here, because the actions achieving \( x, \text{drive to } A \) from \( B \) or \( C \), both incur the regressed subgoal \( f^2 \), for which we have \( u^C(s_3, \{f\}) = u^C(s_1, \{f\}) = \infty \).

Clause Learning

Our clause learning method is based on a simple form of “state minimization”, inspired by Kolobov et al.’s (2012) work on SixthSense. Say we just evaluated \( u^C \) on \( s \) and found that \( u^C(s) = \infty \). Denote by \( \phi := \bigvee_{p \in F \setminus s} \) the disjunction of facts false in \( s \). Then \( \phi \) is a valid clause: for any state \( t \), if \( t \nvDash \phi \) then \( u^C(t) = \infty \). Per se, this clause is useless, as all states but \( s \) itself satisfy \( \phi \). This clause would 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 power as \( u^C \). 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 \( R[s] \) 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 \( s \). Furthermore, while \( u^C \) refutes \( R[s] \) after learning and thus subsumes closed-list duplicate pruning for \( R[s] \), 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 \( R[s] \).

Experiments

Our implementation is in FD (Helmert 2006). For \( u^C \), following 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 cylic problems, we use DFS based on Algorithm 1. We break ties (i.e. order children in the DFS) using \( h^{FF} \) (Hoffmann and Nebel 2001), which helps a bit mainly on solvable instances.

We always start with \( C = C^1 := \{\{p\} \mid p \in F\} \), where \( u^{C^1} \) emulates the standard \( h^1 \) heuristic. As learning too many conjunctions may slow down the search, we experiment with an input parameter \( \alpha \), stopping the learning when the number of counters reaches \( \alpha \) times the number of counters for \( C^1 \). This loosely follows a similar limit by Keyder et al. (2014). We experimented with \( \alpha \in \{1, 16, 32, 64, 128, \infty\} \). Our main interest will be the comparison \( \alpha = \infty \), unlimited learning, vs. \( \alpha = 1 \), the same search without learning. For \( \alpha = 1 \), \( u^C \) emulates \( h^1 \) throughout (which actually is redundant given \( h^{FF} \), but that does not affect our results here).

We focus on resource-constrained planning, where dead-ends abound as the goal must be achieved subject to a limited 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 \( h^{FF} \) and preferred operators, denoted “FD-\( h^{FF} \)”. 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.”
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 $C$, $u_C(I) = \infty$ 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 $h^s = \infty$ at the leaves, our refinement step exactly as stated here extracts the certificate. 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 $h^{max}$ to $h^m$. Proc. ICAPS’09, 354–357.


