Search and Learn: On Dead-End Detectors, the Traps they Set, and Trap Learning

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

Abstract
A key technique for proving unsolvability in classical planning are dead-end detectors $\Delta$: effectively testable criteria sufficient for unsolvability, pruning (some) unsolvable states during search. Related to this, a recent proposal is the identification of traps prior to search, compact representations of non-goal state sets $T$ that cannot be escaped. Here, we create new synergy across these ideas. We define a generalized concept of traps, relative to a given dead-end detector $\Delta$, where $T$ can be escaped, but only into dead-end states detected by $\Delta$. We show how to learn compact representations of such $T$ during search, extending the reach of $\Delta$. Our experiments show that this can be quite beneficial. It improves coverage for many unsolvable benchmark planning domains and dead-end detectors $\Delta$, in particular on resource-constrained domains where it outperforms the state of the art.

1 Introduction
Classical planning is concerned with the analysis of goal reachability in large state spaces, compactly described in terms of planning tasks specifying a vector of state variables, an initial state, a set of actions, and a goal condition. Planning research has traditionally been concerned with solvable tasks, reflected for example in the benchmarks used in the International Planning Competition (IPC) up to the year 2014 [Bacchus, 2001; Long and Fox, 2003; Hoffmann and Edelkamp, 2005; Gerevini et al., 2009; Coles et al., 2012]. However, proving planning tasks unsolvable is also quite relevant in practice. Unsolvable tasks occur, for example, in over-subscription planning [Smith, 2004; Gerevini et al., 2009; Domshlak and Miris, 2015] and in directed model checking [Edelkamp et al., 2004; Kupferschmid et al., 2006; 2008]. Furthermore, even solvable planning tasks often contain unsolvable – dead-end – states, for example when dealing with limited resources [Laborie and Ghallab, 1995; Nakhost et al., 2012; Coles et al., 2013].

Research in classical planning has recently seen a surge of techniques addressing these issues, designing effective techniques for proving unsolvability. After initial works [Bäckström et al., 2013; Hoffmann et al., 2014], a wealth of techniques participated in the inaugural Unsolvability International Planning Competition (UIPC'16) (e. g., [Torralba and Alcázar, 2013; Domshlak et al., 2015; Torralba et al., 2016; Pommerening and Seipp, 2016; Seipp et al., 2016; Steinmetz and Hoffmann, 2016a; Torralba, 2016; Gnäd et al., 2016]). One major strand of these works designs what we will refer to as dead-end detectors $\Delta$: effectively testable criteria sufficient for unsolvability, designed to be called on every state during search, serving to prune those dead-end states detected. Such $\Delta$ were designed based on suitable variants of heuristic functions, namely pattern databases [Edelkamp, 2001], merge-and-shrink heuristics [Helmert et al., 2014; Hoffmann et al., 2014], potential heuristics [Pommerening et al., 2015], and critical-path heuristics [Haslum and Geffner, 2000; Steinmetz and Hoffmann, 2016b; 2017]. These detect a state $s$ to be a dead-end if $s$ is unsolvable in the approximation underlying the heuristic function.

A recent related proposal is the identification of traps [Lipovetzky et al., 2016]: compact representations of non-goal state sets $T$ that cannot be escaped, i.e., where from any state $s \in T$, all states $s'$ reachable from $s$ are also contained in $T$. Such traps can be identified through an offline analysis, prior to search. Here we extend the trap idea in two ways:

(i) We observe that traps can be combined for synergistic effect with arbitrary dead-end detectors $\Delta$.

(ii) We observe that traps can be learned online during search, from the dead-end states encountered.

By (i), the trap $\Theta$ extends the reach of $\Delta$, avoiding “the traps set for the search by $\Delta$”. By (ii), this is done dynamically from information that becomes available during search.

Notably, our technique can also be run without any other dead-end detector $\Delta$ (technically: a trivial $\Delta$ not detecting any dead-ends). In this case, (i) is mute, and (ii) turns our technique into an online-learning variant of the original traps proposal [Lipovetzky et al., 2016].

In the ability to learn sound and generalizable knowledge – “nogoods” – about dead-ends during search, our work is rivaled only by recent methods for the online refinement of a critical-path heuristic dead-end detector $\Delta^C$ [Steinmetz and Hoffmann, 2016b; 2017]. In the ability to exploit synergy with another dead-end detector $\Delta$, our technique is unique.
in the following sense: if $s$ is a state all of whose successor states $s'$ are detected to be dead-ends by $\Delta$, then we can learn to detect $s$ without having to detect also the states $s'$. This is in contrast to all other known dead-end detectors: when learning to detect $s$, these necessarily – and redundantly with the given $\Delta$ – also learn to detect all $s'$. The latter is because all known dead-end detectors are transitive, i.e., when they detect a state $s$, they also detect all states reachable from $s$. Transitivity is a natural property, as, after all, dead-end detectors need to reason about all possible descendant states; for dead-end detectors based on a heuristic function, transitivity follows from consistency. Steinmetz and Hoffmann [2017] explore combinations of $\Delta^C$-learning with other dead-end detectors $\delta$, yet find that these suffer from having to learn to subsume $\delta$. Our notion of $\Delta$-traps does not have that issue, and is empirically synergistic with several $\delta$.

We implemented our techniques in combination with essentially all known dead-end detectors, in particular those run in UIUC’16. We also enhanced the UIUC’16 winner, the Aidos portfolio [Seipp et al., 2016], in this manner. Our experiments show that online $\Delta$-trap learning can be quite beneficial. It is competitive on its own, run without any other dead-end detector. Combined with a variety of previous dead-end detectors $\Delta$, it improves coverage for many unsolvable benchmark planning domains, in particular on resource-constrained domains where it outperforms the state of the art.

2 Background

We use the finite-domain representation (FDR) framework. A planning task is a tuple $\Pi = (V, A, \Delta, G)$. $V$ is a set of state variables, each $v \in V$ associated with a finite domain $D(v)$. $\Delta$ is a set of actions $a$, each a pair $(pre_a, ef_a)$ of partial assignments to $V$. The initial state $I$ is a complete assignment to $V$, the goal is a partial assignment to $V$. A state $s$ is a complete assignment to $V$. An action $a$ is applicable in $s$ if $pre_a \subseteq s$, and applying such $a$ results in the state $s[a]$ overwriting $s$ with $ef_a$ where $ef_a$ is defined. A plan for $s$ is an action sequence $\pi$ whose iterative application leads to $s_G$ where $G \subseteq s_G$; $s$ is a dead-end if no such $\pi$ exists. For a partial variable assignment $t$, we denote by $V(t)$ the set of variables $v$ for which $t(v)$ is defined. If $t(v)$ is not defined, we also write $t(v) = \perp$. For a subset of variables $V \subseteq V$, the projection of $t$ onto $V$ is denoted $t|_V$.

We denote the set of all states in $\Pi$ by $S$. A heuristic function is a function $h : S \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$. The return value $\infty$ is used to indicate dead-ends. That ability has traditionally been treated as a by-product of goal distance estimation, but the aforementioned recent works have designed heuristic function variants dedicated to dead-end detection.

Another recent proposal for dead-end detection are traps [Lipovetzky et al., 2016]. A trap is a set $T$ of non-goal states that is invariant, i.e., once we are in $T$, we can never leave it again. Formally, for every state $s \in T$ it must hold $G \not\subseteq s$, and for all actions $a$ applicable to $s$ that $s[a] \in T$. The idea is to identify a compact representation $\Theta$ of such a $T$ offline, prior to search, and to use $\Theta$ to detect and prune dead-ends during search. This compact representation is determined from partial states, partial variable assignments $t$ of size up to $k$, where $k$ is a parameter. The states induced by such $\Theta$ are given by $T^\Theta := \{s \in S \mid \exists t \in \Theta : t \subseteq s\}$. Verifying whether $T^\Theta$ is a trap can be done equivalently on $\Theta$ through progression over partial states.

Thus, $a$ is applicable to a partial state $t$ if $pre_a(t(v)) \subseteq t$, and if applicable, define the progression of $t$ over $a$ as the partial variable assignment $t[a]$ where

\[
\begin{align*}
&\quad\text{if } \text{eff}_a(v) \neq \perp, \\
&\quad\text{if } \text{eff}_a(v) = \perp \\
&\quad\text{if } \text{eff}_a(v) = \perp \text{ and } t(v) \neq \perp, \\
&\quad\text{if } \text{eff}_a(v) = \perp \text{ and } \text{pre}_a(v) \neq \perp, \\
&\quad\text{otherwise}
\end{align*}
\]

In words, $t$ is extended by $pre_a$ and the resulting (partial) variable assignment is overwritten by $\text{eff}_a$. By definition, $t[a] \subseteq s[a]$ for the application of $a$ in any state $s$ where $a$ is applicable and $t \subseteq s$. It is easy to show that $T^\Theta$ constitutes a trap if and only if (a) every $t \in \Theta$ disagrees with the goal on some $v$, and (b) $\Theta$ is closed under progression, i.e., for all $t \in \Theta$ and for all actions $a$ applicable to $t$, there is $t' \in \Theta$ so that $t' \subseteq t[a]$.

3 Dead-End Detectors and the Traps they Set

We show that trap identification can be combined with arbitrary dead-end detectors. To this end, we consider a generic notion of dead-end detectors, and we introduce an accordingly modified notion of traps.

A dead-end detector is a function $\Delta : S \rightarrow \{0, \infty\}$ where $\Delta(s) = \infty$ only if $s$ is a dead-end state. Like for heuristic functions, the intention is to call $\Delta$ on every state during search, so $\Delta$ will typically be effectively computable. As a baseline, we will use the native dead-end detector, denoted $\Delta^0$, which returns $0$ for all states (i.e., does not recognize any dead-end). More elaborate known dead-end detectors we consider are $\Delta^{DBR}$ (pattern databases [Seipp et al., 2016]), $\Delta^{MS}$ (merge-and-shrink [Toralba et al., 2016]), $\Delta^{Po}$ (potential heuristics [Seipp et al., 2016]), and $\Delta^C$ (critical-path heuristics [Haslum and Geffner, 2000; Steinmetz and Hoffmann, 2016a]). To consider combinations of two (or more) dead-end detectors $\Delta_i$, we use addition, which dominates each $\Delta_i$, returning $\infty$ whenever $\Delta_i$ does.

Given an arbitrary dead-end detector $\Delta$, a $\Delta$-trap is a set $T \subseteq S$ of states such that, for all $s \in T$, (a) $G \not\subseteq s$, and (b) for every action $a$ applicable to $s$, either $s[a] \in T$ or $\Delta(s[a]) = \infty$. In other words, a $\Delta$-trap is a set of non-goal states whose only escape routes lead into dead-ends detected by $\Delta$. Intuitively, such $T$ is a “trap set for the search by $\Delta$”, in that, starting from $T$, $\Delta$ will eventually prune every search path; yet $\Delta$ doesn’t explicitly indicate this, so we will have to search through the entirety of $T$ before finding out.

For the trivial dead-end detector $\Delta = \Delta^0$, the additional condition $\Delta^0(s[a]) = \infty$ is never satisfied. Thus, $\Delta$-traps generalize the original traps (the special case of $\Delta^0$-traps). An important difference between $\Delta$-traps and original traps,
as we move away from $\Delta^0$ and use more informed $\Delta$, is transitivity. While $\Delta^0$-traps $T$, by definition, have the property that for every $s \in T$ and every transition $s \rightarrow s'$, it must be $s' \in T$, this is no longer so for $\Delta$-traps in general: those $s'$ where $\Delta(s') = \infty$ no longer need to be contained in $T$. As previously discussed, this is key to synergy, as it allows $T$ to be complementary to $\Delta$, instead of forcing $T$ to subsume $\Delta$.

4 Compact $\Delta$-Trap Representations

To make use of $\Delta$-traps in search, the idea is to identify compact representations $\Theta$ whose computation does not require the enumeration of $T^\Theta$. This $\Theta$ can then be used for dead-end detection through the dead-end detector $\Delta^\Theta$ where $\Delta^\Theta(s) = \infty$ if $t \subseteq s$ for some $t \in \Theta$, and $\Delta^\Theta(s) = 0$ else.

For the characterization of such $\Theta$, we require the dead-end detector $\Delta$ to be partial-state compatible. This is, it must be possible to evaluate $\Delta$ on every partial-state $t$ efficiently, where $\Delta(t) = \infty$ if and only if $\Delta(s) = \infty$ for all states $s \in S$ so that $t \subseteq s$. Note that in principle every dead-end detector can be evaluated on partial states, since $\Delta(t)$ can be computed trivially by enumerating the states $s$ that satisfy $t$, and evaluating $\Delta$ for every one of them. But this is not in general efficient as the number of states $s$ is exponential in the number of variables where $t$ is undefined. Fortunately, many dead-end detectors natively support the evaluation of partial states, and for those that don’t, there usually exist sufficient conditions for $\Delta(t) = \infty$ that can be tested efficiently. We give more details on that in the experiments section.

The original trap conditions are easily generalized:

**Theorem 1** $T^\Theta$ is a $\Delta$-trap if (C1) all $t \in \Theta$ disagree with $G$ on some variable $v$, and (C2) for all actions $a$ that are applicable to $t$, there is either $t' \in \Theta$ with $t' \subseteq t[a]$ or $\Delta(t[a]) = \infty$.

**Proof.** Assume the contrary. It follows immediately from (C1) that $T^\Theta$ cannot contain a goal state. Hence, there must be a state $s \in T^\Theta$ and an action $a$ that is applicable to $s$ so that $s[a] \not\in T^\Theta$ and $\Delta(s[a]) < \infty$. Let $t \in \Theta$ be so that $t \subseteq s$. Note that because $\preceq_a \subseteq s$, it immediately follows that $\preceq_{a \sqcup (\langle a \rangle \subseteq t)}$ and thus $a$ is applicable to $t$. Now, due to (C2), either there exists $t' \in \Theta$ with $t' \subseteq t[a]$, or $\Delta(t[a]) = \infty$. Since $t[a] \subseteq s[a]$, both cases end up in a contradiction. □

5 Offline Computation

The algorithm for computing $\Delta^0$-traps proposed by Lipovetzky et al. [2016] can be easily adapted to support the generation of $\Delta$-traps for arbitrary partial-state compatible dead-end detectors $\Delta$. The procedure is depicted in Figure 1. It identifies a subset of the given partial state candidates $\mathcal{C}$, guaranteeing that the result satisfies the conditions of Theorem 1. Originally, all partial states of size up to $k$ were considered in $\mathcal{C}$, where $k$ was a parameter. However, this is not required for the correctness of the algorithm; $\mathcal{C}$ may be chosen arbitrarily.

Finding the desired subset of partial states corresponds to propagating markings through an AND/OR-graph whose AND-nodes correspond to actions, whose OR-nodes correspond to the selection of partial states in consideration, and whose edges correspond to progression over those partial states. The partial states not marked during this procedure give the resulting $\Theta$. The propagation starts with the OR-nodes, so partial states, that do not disagree with the goal on any variable, i.e., those satisfying (C1). An AND-node is marked when all its successors are marked, and thus the corresponding progression would not be covered by the resulting trap. An OR-node is marked when at least one of its successors is marked, i.e., when there is an action whose progression would lead out of the trap, violating (C2).

The main difference to Lipovetzky et al.’s algorithm is that an action $a$ applicable to $t$ is ignored if the progression of $t$ over $a$ is already detected by $\Delta$ as dead-end. The resulting AND/OR-graph may hence contain fewer edges, so fewer partial states may be touched during marking propagation and hence removed from $\mathcal{C}$, resulting in larger traps $T^\Theta$.

The procedure guarantees to find a maximal trap:

**Theorem 2** For the result $\Theta$ of ComputeTrap($\mathcal{C}, \Delta$), it holds

(i) $\Theta$ satisfies (C1) and (C2) of Theorem 1, and

(ii) if $\Theta'$ satisfies (C1) and (C2) and $\Theta' \subseteq \mathcal{C}$, then $\Theta' \subseteq \Theta$.

Claim (i) is a simple extension of the argument by Lipovetzky et al. [2016]. Claim (ii) holds because the procedure does not unnecessarily remove any partial states.

6 Online Learning

One major drawback of the algorithm from the previous section is that the computation of $\Theta$ requires an a-priori choice of partial state candidates $\mathcal{C}$. The only known method is the enumeration of all partial states of size up to $k$. This is feasible only for small $k$. On the other hand, many of those partial states might actually not be relevant for the resulting trap representation, and by imposing a size bound on the partial states, we might be missing the ones that actually matter.

Inspired by the online-learning paradigm by Steinmetz and Hoffmann [2017], in this section we present a method to choose the partial states dynamically during search. However, instead of computing $\mathcal{C}$ fed into Algorithm 1, we compute a $\Delta$-trap representation $\Theta$ directly. The general idea is to
run search with dead-end detector $\Delta + \Delta^\Theta$, starting with the trivial initialization $\Theta = \emptyset$. As search keeps progressing, $\Delta$-traps $T$ will become known that are not yet represented by $\Theta$. Whenever this happens, we compute a generalization $\Theta'$ of $\Theta$, guaranteeing that $\Theta'$ still satisfies (C1) and (C2) of Theorem 1, and, additionally, (C3) $\Theta'$ is weaker than $\Theta$, i.e., every state represented by $\Theta$ is also represented by $\Theta'$, and (C4) the states in $T$ are all covered by $\Theta'$. Conditions (C1) and (C2) ensure that $T^\Theta$ still constitutes a $\Delta$-trap; (C3) and (C4) ensure progress in that $\Theta'$ recognizes strictly more dead-ends than $\Theta$. When the computation of $\Theta'$ is finished, we replace $\Theta$ by $\Theta'$ and continue with search. This refinement has the potential to generalize to states outside of $T$, and in particular to states not visited by search so far, so may lead to less search in the future.

6.1 Identifying $\Delta$-Traps in Search

Before we go into details on the generalization step, let us briefly discuss how the states required for that step are identified in search. Following the notation of Steinmetz and Hoffmann [2017], a dead-end $s$ becomes known in search as soon as every successor of $s$ either has been visited by search and is not a goal state, or is identified by $\Delta + \Delta^\Theta$ as dead-end for the current $\Theta$. In other words, $s$ becomes a known dead-end as soon as it is possible to prove $s$ to be a dead-end by combining $\Delta + \Delta^\Theta$ with the knowledge about the state space that the search has provided so far.

Steinmetz and Hoffmann have shown how to extend search algorithms for the purpose of identifying known dead-ends. This particularly becomes easy in depth-oriented search algorithms, where a dead-end $s$ becomes known as soon as search backtracks out of the maximal strongly-connected component (SCC) $S$ which contains $s$. Moreover, before this actually happens, every successor of every state in $S$ that is not contained in $S$ itself must have been identified as known dead-end already due to the depth-oriented nature of the exploration. Hence, if we guarantee that $\Theta$ is updated whenever a known dead-end is found, then before search backtracks out of $S$, $\Delta + \Delta^\Theta$ recognizes every such successor state, and thus $S \cup T^\Theta$ constitutes a $\Delta$-trap. Note that in particular before a task is proven to be unsolvable, search has to backtrack out of the maximal SCC containing the initial state. If $\Theta$ is generalized on this SCC as well, then the resulting trap represents every reachable state not recognized as a dead-end by $\Delta$.

6.2 Generalizing $\Delta$-Traps

The search methods just described deliver, for the purpose of any one generalization step, a set $S$ of states where $S \cup T^\Theta$ constitutes a $\Delta$-trap, and $S \subseteq T^\Theta$. Our aim in the generalization step is to compute a $\Theta'$ that compactly represents $S \cup T^\Theta$, i.e., that satisfies (C1) – (C4), and that may generalize to states outside $S$.

A $\Theta'$ that satisfies (C1) – (C4) can trivially be computed by just adding the variable assignments from $S$ to $\Theta$. That every partial state in $\Theta \cup S$ disagrees with the goal on some variable follows from the invariant that $\Theta$ satisfies (C1), and the assumption that $S$ does not contain a goal state. That (C2) is satisfied follows from the invariant that $\Theta$ satisfies (C2), and from the way how $S$ is selected: every transition going out of the states in $S$ ends in a dead-end recognized by $\Delta$, or a state that is represented by $\Theta$. (C3) and (C4) are satisfied by construction. However, in this computation of $\Theta'$, $T^\Theta$ would merely be an extension of the previous trap $T^\Theta$ to the states in $S$. In particular, $\Theta'$ does not generalize to states that have not yet been visited by search so far.

In order to obtain $\Theta'$ that may generalize to other states than $S$, the idea is compute partial states from $S$ by removing variable assignments not relevant w.r.t. (C1) and (C2). The pseudocode of this generalization procedure is shown in Figure 2. For each state in $S$, a subset of variables is computed so that the extension of $\Theta$ by the projections of the states in $S$ on their respective variable subset still satisfies (C1) and (C2). The smaller those variable subsets are, the more states are represented by the resulting set of partial states.

To ensure that $\Theta'$ satisfies condition (C1), for every state $s \in S$, the corresponding variable subset is initialized to one of the variables where $s$ disagrees with the goal. Such a variable must exist because $S$ does not contain a goal state by assumption. Condition (C2) is ensured by iteratively re-adding variables to the variable subsets of the states. As argued above, $\Theta \cup S$ satisfies (C2), and hence this happens when $V_s = V$ for all $s \in S$ at the latest. Note that regardless which state is selected inside the while loop, there must be always a variable $v \in V$ that is not contained in $V_s$. For any state $s$ whose variable subset $V_s$ contains all variables, it holds that $s|_{V_s} = s$, and hence $s[v] \in S$ or $(\Delta + \Delta^\Theta)(s[v]) = \infty$, due to the assumption how $S$ is selected. Since $s|_{V_s} \subseteq s'$ for every state $s' \in S$, this however means that $s$ cannot be the state violating (C2), i.e., such states $s$ cannot be chosen.

Hence, the execution of TrapGeneralization($\Theta$, $S$) is well-defined, and terminates with $\Theta'$ that satisfies (C1) and (C2). $\Theta'$ satisfies (C3) because it is a superset of $\Theta$. $\Theta'$ satisfies (C4) because it contains the projection of $s$ onto $V_s$ for every state $s \in S$. Finally, $\Theta'$ may generalize to states outside of $S$ as soon as at least one $V_s$ does not contain all variables.

Theorem 3 The execution of TrapGeneralization($\Theta$, $S$) is well-defined, and terminates with $\Theta'$ satisfying (C1) – (C4).

3When extending $\Theta$ by new partial states, it may become possible to also minimize the existing partial states in $\Theta$. Thus, it could makes sense to apply the procedure of Figure 2 to all partial states $\Theta \cup S$, instead of just $S$. Yet in our experiments this turned out to be detrimental.
7 Experiments

Our implementation is in FD [Helmert, 2006]. We use the UIPC’16 benchmarks; the part of Hoffmann et al. [2014] unsolvable benchmark collection that is not used in UIPC’16; and unsolvable versions of the resource-constrained benchmarks by Nakhost et al. [2012], obtained by scaling resource constrainedness within \{0.5, 0.6, \ldots, 0.9\}. All experiments were run on a cluster of Intel Xeon E5-2650v3 machines, with runtime (memory) limits of 30 minutes (4 GB).

We experiment with seven dead-end detectors \(\Delta\) for \(\Delta\)-trap detection, mostly taken from the UIPC’16 participants:

- \(\Delta^0\) is included as a baseline, showing the impact of dead-end detection by traps alone.
- Critical-path heuristic \(h^m\) for \(m = 1\) and \(m = 2\) [Haslum and Geffner, 2000]. For partial states \(t\), we approximate the value of \(\Delta^m(t)\) through \(h^m(t^+)\), where \(t^+\) contains \(v = t(v)\) if \(t(v)\) is defined, and contains \(v = d\) for every \(d \in D(v)\) if \(t(v)\) is not defined. Obviously, \(h^m(t^+) = \infty\) implies \(\Delta^m(t) = \infty\), so this can be used as a sufficient condition.
- The two most competitive unsolvability merge-and-shrink (M&S) abstractions [Hoffmann et al., 2014; Torralba et al., 2016]: MSp which computes the perfect dead-end detector \(\Delta^*\), recognizing all dead-ends; and MSa which imposes a bound on the abstraction size, and hence approximates \(\Delta^*\). We include MSp for reference only, and do not use it for computing \(\Delta\)-traps. For a partial state \(t\), \(\Delta^{\text{MSa}}(t)\) is computed by finding all abstract states of the states represented by \(t\) (this can be done effectively given the cascading tables representation of MSa). Then, \(\Delta^{\text{MSa}}(t) = \infty\) iff every such abstract state is a dead-end in the abstract state space.
- The dead-end PDB heuristic from the UIPEC’16 winner Aidos [Seipp et al., 2016]. We approximate \(\Delta^{\text{PDB}}(t)\) by checking whether the PDB heuristic contains a pattern \(V\) so that \(V \subseteq \forall(t)\) and \(t_{\forall(t)}\) is recognized as dead-end in the respective abstraction.
- The operator-counting heuristic \(\text{Seq}\) that is obtained from state-equation constraints [Pommerening et al., 2014]. For a state \(s\), \(\Delta^{\text{Seq}}(s) = \infty\) if the corresponding LP does not have a solution. To approximate \(\Delta^{\text{Seq}}(t)\), we set the the lower- and upper-bounds of the constraints in the corresponding LP so that these constitute lower-, respectively upper-bounds for every state that is represented by \(t\). Hence, if this LP does not have a solution, the LP corresponding to every \(s\) with \(t \subseteq s\) cannot have one either, i.e., \(\Delta^{\text{Seq}}(t) = \infty\).
- The dead-end potential heuristic \(\text{Pot}\) from Aidos, also an operator-counting heuristic. Our approximation of \(\Delta^{\text{Pot}}(t)\) follows the same idea as for \(\Delta^{\text{Seq}}(t)\).

We evaluate the benefit of each of these dead-end detectors \(\Delta\) for two different purposes: (1) offline identification of \(\Delta\)-traps \(\Theta\), with subsequent search using only \(\Delta^\Theta\) for dead-end detection; and (2) online learning of a \(\Delta\)-trap \(\Theta\), during search, using \(\Delta + \Delta^\Theta\) for dead-end detection (vs. \(\Delta\) alone). We also experiment with a variant of the Aidos portfolio in which we added \(\Delta\)-trap online learning to each of its components. In this context, we disable two of Aidos’ techniques, partial-order reduction and resource variable detection, neither of which is compatible with the \(\Delta\)-trap learning algorithm. All configurations, apart from the original Aidos version, run depth-oriented search with duplicate detection.

Figure 3(a) shows the coverage results. The modified Aidos configuration is indicated by “+”. The version that participated in UIPC’16 is shown on the right-hand side of the table, together with MSp and the only other online dead-end learning technique \(\Delta^C\) [Steinmetz and Hoffmann, 2016a]. The results for \(\Delta\)-trap online learning are shown in the middle part of the table (“–” shows the results for \(\Delta\) alone, “\(\Theta\)” for \(\Delta + \Delta^\Theta\)). The left part shows the results for dead-end detection by \(\Delta^0\), for offline computed \(\Delta^0\)-traps \(\Theta\). The partial state candidates for the offline \(\Delta\)-trap computation are chosen to all partial states of size up to \(k = 1\), respectively \(k = 2\). Remember that \(\Delta^0\)-traps correspond exactly to the traps as originally proposed by Lipovetzky et al. [2016], and that offline \(\Delta^0\)-traps are computed according to exactly that proposal. Figure 3(b) shows the effect of the different \(\Delta\) on offline computed traps for \(k = 2\). The results for \(k = 1\) look similar, and are left out for space reasons.

Consider first offline construction. Partial states of size 1 are not enough to compute a non-empty \(\Delta^0\)-trap in any domain but DocTransfer, effectively turning the respective configuration into blind search. The computed \(\Delta^0\)-traps for \(k = 2\) help in Bottleneck, DocTransfer, and most notably in Mystery for which the initial state was already represented by the trap in every single task. As indicated by Figure 3(b), the different dead-end detectors \(\Delta\) have complementary effects on trap generation. Any one \(\Delta\) helps in some domain, yet is detrimental in others. The latter is due to the additional overhead resulting from the evaluation of such \(\Delta\) during trap construction. At the same time, \(\Delta\) does not always help to improve the computed trap. An extreme example for that is MSa, which vastly increases coverage in the resource-constrained domains, but worsens the results on all other domains considerably. Overall, dead-end detection through an offline computed \(\Delta\)-trap alone cannot compete with current state-of-the-art methods.

The potential of \(\Delta\)-traps really becomes alive, however, in online learning. State-of-the-art performance is achieved in many domains even for \(\Theta\) using \(\Delta^0\), i.e., without any additional dead-end detector. This vanilla configuration outperforms, in particular, the only other dead-end learning configuration \(\Delta^C\) in its prime discipline, the resource-constrained domains. Coverage in the latter can be increased even further through combination with the PDB heuristic, pushing the respective \(\Delta\)-trap learning configuration to perfect coverage in NoMystery and Rovers. In TPP, \(\Delta\)-trap learning is inferior only to Aidos’ resource-variable identification component.

\(\Delta\)-trap learning improves overall coverage on the resource-constrained domains for every tested \(\Delta\) except for MSa. On the other domains, the overall picture is not as consistently good. The number of domains where \(\Delta\)-trap learning positively (negatively) affects coverage are 9 (7) for \(\Delta^1\); 7 (5)
for $\Delta^2$: 3 (10) for MSa; 7 (7) for PDB; 6 (4) for Seq; 5 (4) for Pot; and 7 (5) for Aidos$^1$. Synergistic effects, where the combination of $\Delta$ with $\Delta$-trap learning solves instances not solved by either of the component techniques alone, occur for PDBs in the resource-constrained domains, as well as for $\Delta_1$, Seq, and Pot in Diagnosis and DocTransfer.

Figure 3(c) compares for every benchmark instance the search and runtime reduction factors that result from enabling trap online learning on the $\Delta^2$-baseline. Per-instance based comparison of search reduction factors (x-axis) vs. runtime reduction factors (y-axis) for UIPC (+) and resource-constrained domains (x).

**8 Conclusion**

Dead-end detection is an important technique in planning. One major limitation of previous dead-end detectors is transitivity, which entails that, while we can of course use as many dead-end detectors as we like, each of those is doomed to ignore the information provided by its peers. We introduce $\Delta$-traps as a remedy, generalizing the previous trap idea to allow for synergy with a complementary dead-end detector $\Delta$. We furthermore introduce methods allowing to learn the trap online, instead of a static offline analysis. Our experiments show that both contributions can be quite beneficial.

For future work, an aspect to look at is the generalization step in online learning, which might be optimizable for computational cost and generalization power. Beyond this, and beyond just trap learning, major questions are whether one can usefully exploit search knowledge falling short of fully identified dead-end components, and whether one can design learning methods targeted at resource limits specifically.

**Acknowledgments**

This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1, “Critically Constrained Planning via Partial Delete Relaxation”, as well as by the German Federal Ministry of Education and Research (BMBF) for the Center for IT-Security, Privacy and Accountability (CISPA, grant no. 16KIS0656). Thanks to Malte Helmert for pointing out that dead-end detectors based on consistent heuristics are transitive.
References


