CLone: A Critical-Path Driven Clause Learner

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

Abstract

In this paper, we introduce a planner that, similar to CDCL (conflict-driven clause learning) SAT solvers, learns from making wrong decisions in a way guaranteeing to preclude these mistakes in the future. The planner we describe, named CLone, conducts a depth-first forward search in the state space of the problem. To identify dead ends early on, and thus to reduce search effort, we make use of the critical path heuristic h^C. h^C is defined relative to a given set C of fact conjunctions. During search, we identify unrecognized dead ends, i. e., dead ends s with $h^C(s) < \infty$, and we use them to update C in order to recognize s, and possibly also dead ends similar to s. As the evaluation of h^C is getting computationally more expensive the more conjunctions are added to C, we further maintain and continuously update a set of clauses Δ with the property that if $s \not\models \phi$ for some $\phi \in \Delta$ then $h^C(s) = \infty$. Although these clauses are subsumed by h^C by definition, they have a tremendous impact in practice. For a more detailed explanation of this approach, we refer the reader to (Steinmetz and Hoffmann 2016).

Introduction

CLone determines the (un-)solvability of a problem by running a depth-first like search in the state space. When only running search in the state space (i. e. without using additional state reduction techniques), one has to construct the entire state space to show the unsolvability of a problem. To reduce the number of states that actually have to be touched by the search, and thus to avoid building the whole state space, we rely on heuristic functions h, i. e., estimations of the distance from a given state to the goal. Contrary to finding solutions to solvable problems, however, we are not interested in the actual goal distance estimations given by the heuristics: if a problem is unsolvable, then the search has to explore all states s with $h(s) < \infty$ – where $h(s) = \infty$ means that s is even unsolvable in the relaxation underlying h. This led to the definition of *unsolvability heuristics*, heuristics that either return ∞ ("dead-end") or 0 ("don't know") (Bäckström *et al.* 2013; Hoffmann *et al.* 2014). Concrete unsolvability heuristics have been designed based on state-space abstractions, specifically projections (pattern databases (Edelkamp 2001)) and merge-and-shrink abstractions (Helmert *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.

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 *et al.* 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. 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 $\hat{S}$ of unrecognized dead-ends, where $h^C(s) < \infty$ for all $s \in \hat{S}$, are identified (become *known*) when all their descendants have been explored. We *refine* h^C on such components $\hat{S}$, adding new conjunctions into C in a manner guaranteeing that, after the refinement, $h^C(s) = \infty$ for all $s \in \hat{S}$. 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 $\hat{S}$ may lead to recognizing also other dead-end states $s' \notin \hat{S}$. It is known that computing critical-path heuristics over large sets C is (polynomial-time yet) computationally expensive. Computing h^C on all search states often results in prohibitive runtime overhead. We tackle this with a form of *clause learning*. For a dead-end state s where h^C evaluates to ∞, we extract a clause ϕ that guarantees for all states s' with $s' \not\models \phi$ that $h^C(s') = \infty$. When testing whether a new state s' is a dead-end, we first evaluate the clauses ϕ, and invoke the computation of $h^C(s')$ only in case s' satisfies all clauses $\phi \in \Delta$.

The resulting algorithm approaches the elegance of clause learning in SAT (e. g. (Marques-Silva and Sakallah 1999; Moskewicz *et al.* 2001; Eén and Sörensson 2003)): 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.

For full details on the techniques used in this planner, we refer the reader to (Steinmetz and Hoffmann 2016).

Background

We consider planning tasks $\Pi = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ in STRIPS encoding. $\mathcal{F}$ gives a set of **facts**; $\mathcal{A}$ a set of **actions**; $\mathcal{I} \subseteq \mathcal{F}$ is the **initial state**; and $\mathcal{G} \subseteq \mathcal{F}$ the **goal**. Each $a \in \mathcal{A}$ has a **precondition** $pre(a) \subseteq \mathcal{F}$, an **add list** $add(a) \subseteq \mathcal{F}$, and a **delete list** $del(a) \subseteq \mathcal{F}$. Action costs are irrelevant with respect to the solvability of planning tasks, so we assume unit cost 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 $c \subseteq \mathcal{F}$. 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). There is a transition from state s to $s[[a]]$ via action a if a is **applicable** to s, i. e., $pre(a) \subseteq s$, and $s[[a]] := (s \setminus del(a)) \cup add(a)$. **Goal states** are all states s where $G \subseteq s$. A **dead-end** is a state for which no path to a goal state exists. Viewing the state space of Π, denoted Θ^Π, as a directed graph over states, given a subset $\mathcal{S}'$ of states, by $\Theta^\Pi|_{\mathcal{S}'}$ we denote the subgraph induced by $\mathcal{S}'$. If there is a path in $\Theta^\Pi|_{\mathcal{S}'}$ from s to t, then we say that t is **reachable** from s in $\Theta^\Pi|_{\mathcal{S}'}$.

A **heuristic** is a function h mapping states to natural numbers or ∞. 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, it suffices to achieve the most costly subgoal (sub-conjunction). The family is parameterized by the set of atomic subgoals considered. Formally, for a fact set G and action a, define the **regression** of G over a as $R(G, a) := (G \setminus add(a)) \cup pre(a)$ in case that $add(a) \cap G \neq \emptyset$ and $del(a) \cap G = \emptyset$; otherwise, the regression is undefined and we write $R(G, a) = \bot$. By $\mathcal{A}[G]$ we denote the set of actions where $R(G, a) \neq \bot$. Let C be any set of conjunctions. The generalized critical-path heuristic (Hoffmann and Fickert 2015) $h^C(s)$ is defined through $h^C(s) := h^C(s, \mathcal{G})$ where

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

A well known property of critical path heuristics is that they are *admissible*, i. e., that they always underestimate the real goal distance. In other words, if $h^C(s) = \infty$, then, as desired, s is indeed a dead end, and s can be disregarded in search without loosing completeness. Note that $h^C(s) = \infty$ occurs (only) due to empty minimization in the middle case of Equation 1, i. e., if every possibility to achieve the global goal $\mathcal{G}$ incurs at least one atomic subgoal not supported by any action.

Similarly as for h^m, h^C can be computed in time polynomial in $|C|$ and the size of Π. 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.

Algorithm 1: CLone

```
1  Procedure DFS(Π)
2      Open := empty stack; push I to Open;
3      Closed := ∅;
4      while Open is not empty do
5          s ← Open.top();
6          Open.pop();
7          if s ∈ Closed then
8              continue;
9          if h^C(s) = ∞ then
10             Backtrack(s);
11             continue;
12         if G ⊆ s then
13             return solvable;
14         Closed := Closed ∪ {s};
15         for all a ∈ A applicable to s do
16             push s[[a]] to Open;
17         CheckAndLearn(s);
18     return unsolvable;
19 Procedure CheckAndLearn(s)
20     R[s] := {t | t reachable from s in Θ^Π|_{Open∪Closed}};
21     if R[s] ⊆ Closed then
22         refine C s.t. h^C(t) = ∞ for every t ∈ R[s];
23         Backtrack(s);
24 Procedure Backtrack(s)
25     label s;
26     for every unlabeled parent t of s do
27         CheckAndLearn(t);
```

Search, Fail, Refine & Repeat

CLone runs search in the state space of the problem using h^C as a dead end identifier. During search, CLone keeps track of expanded states to identify yet unrecognized dead ends. Whenever an unrecognized dead end s is found, the set C is extended by new atomic conjunction, guaranteeing that $h^C(s) = \infty$ after the refinement. In order to avoid as many of the rather expensive computations of h^C as possible, CLone learns clauses as sufficient conditions to $h^C(s) = \infty$ each time a state is found where $h^C(s)$ has been evaluated to ∞. The clauses are used to filter states before h^C is evaluated.

Identifying Failures in Search

Consider Algorithm 1. At the heart of CLone, it performs a depth-first forward search in the state space of the problem, while maintaining a closed list for duplicate checking. h^C is used as an *efficient* method to identify dead ends, eliminating necessity of exploring any of the state's successors.

To identify also dead ends not (yet) recognized by h^C, CLone analyzes the search space after each state expansion. The corresponding code, function `CheckAndLearn` in Algorithm 1, performs a full lookahead search in the current search space ($\Theta^\Pi|_{Open \cup Closed}$), looking for states that have

not been expanded so far. Intuitively, a state s is a **known dead-end** if the search has already proved that s is a dead end, meaning that all states t reachable from s have already been explored and no such state t is a goal state, i. e., $\mathcal{R}[s] \subseteq \mathit{Closed}$. 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.*

Vice versa, if $\mathcal{R}[s] \not\subseteq \mathit{Closed}$, then some descendants of s have not yet been explored, so the search does not know whether or not s is a dead-end.

Once a known dead-end s is found, C is refined in a way guaranteeing that $h^C(s) = \infty$ afterwards. As we know already that s is a dead end, forcing that $h^C(s) = \infty$ seems to be redundant. However, the reason of the refinement of C is not actually to have h^C recognize s as dead end, but rather the hope that dead ends similar to s will be recognized as well due to this very refinement.

To guarantee that all known dead-ends are found, and thus to learn as much as possible, $\mathcal{R}[t] \subseteq \mathit{Closed}$ has to be checked for each state $t \in \mathit{Closed}$ after every state expansion. Naively checking this property for each state $t \in \mathit{Closed}$ after every expansions is clearly infeasible. Instead, CLone uses the observation that the property $\mathcal{R}[t] \subseteq \mathit{Closed}$ can only change for ancestors t of the state s that was expanded last. To find all these states, CLone checks this condition on the parents of s, and recursively continues on those parents satisfying the condition.

Although CLone could in principle also run any other Closed-list based search algorithm, the key advantage of DFS in our setting is that it focuses on completely exploring subtrees, and hence it is able to identify unrecognized dead-ends quickly.

Failure Analysis & h^C Refinement

Once identified a known though unrecognized dead end s, we have to find conjunctions X that, when added to C, guarantee that $h^{C \cup X}(s) = \infty$. To find X, CLone makes use of the specific context in which C is going to be refined. Observe that whenever $\texttt{CheckAndLearn}(s)$ calls the refinement of C, it holds: *(*) For every transition $t \to t'$ where $t \in \mathcal{R}[s]$, either $t' \in \mathcal{R}[s]$ or $u^C(t') = \infty$.* We will refer to this by the **recognized neighbors** property. This is because $\mathcal{R}[s]$ contains only closed states, so it contains all states t reachable from s except for those where $h^C(t) = \infty$.

Algorithm 2 shows the overall refinement process. We use $\hat{S} := \mathcal{R}[s]$ to denote the component on which C is refined, and $\hat{T} := \{t' \mid t \xrightarrow{a} t', t \in \hat{S}, t' \notin \hat{S}\}$ to denote recognized neighbors. As shown at the top of Algorithm 2, CLone computes X by recursively adding an unreachable subgoal $x \subseteq R(G, a)$ for each $a \in \mathcal{A}[G]$ to X, corresponding to the middle case of Equation 1, and then continuing on $G = x$ until either $h^C(s, G)$ is already ∞ for every $s \in \hat{S}$, or $a[G] = \emptyset$. Note that determining whether some $x \subseteq R(G, a)$ is unreachable from a state s is PSPACE-complete in general. To still find such an x efficiently, CLone uses the recognized neighbors property:

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

```
1  Procedure Refine(G)
2      x := ExtractX(G);
3      X := X ∪ {x};
4      for a ∈ A[x] where ex. s ∈ Ŝ s.t. h^C(s, R(x,a)) < ∞
         do
5          if there is no x' ∈ X s.t. x' ⊆ R(x,a) then
6              Refine(R(x,a));
7  Procedure ExtractX(G)
8      x := ∅;
       /* Lemma 2 (ii)                                   */
9      while ∃t ∈ T̂ so that h^C(t,x) < ∞ do
10         c0 := ∅; n0 := 0;
11         for each c ∈ C where c ⊆ G do
12             n := |{t ∈ T̂ | h^C(t,x) < ∞, h^C(t,c) = ∞}|;
13             if n ≥ n0 or (n = n0 and |c \ x| < |c0 \ x|)
                 then
14                 c0 := c; n0 := n;
15         x := x ∪ c0;
       /* Lemma 2 (i)                                    */
16     for every s ∈ Ŝ do
17         if x ⊆ s then
18             select p ∈ G \ s; x := x ∪ {p};
19     return x;
```

Lemma 2 (Steinmetz and Hoffmann 2016)**.** *If $x \subseteq G$ satisfies*

(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 $h^C(t, c) = \infty$;

then x is unreachable from every $s \in \hat{S}$.

To ensure (ii) of Lemma 2 in the computation of x, CLone's procedure $\texttt{ExtractX}(G)$ tries to greedily construct a minimal conjunction that covers all recognized neighbors. It does so by merging an atomic conjunction $c_0 \in C, c_0 \subseteq G$ into x that covers as many recognized neighbors as possible, while being minimal in size, as long as the condition (ii) is not satisfied. 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, to ensure Lemma 2 (i). Putting things together, we get the desired result:

Theorem 2 (Steinmetz and Hoffmann 2016)**.** *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, $h^{C \cup X}(s) = \infty$ for every $s \in \hat{S}$.

Clause Learning

As pointed out, the clauses we learn do not have the same pruning power as h^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 h^C,

and we learn a new clause every time h^C is evaluated and returns ∞.

Different from the clause learning approach presented in our prior work (Steinmetz and Hoffmann 2016), CLone computes the clauses directly from the structure underlying h^C. Say h^C has been evaluated on s to ∞. CLone constructs a clause ϕ so that $s' \not\models \phi$ implies $h^C(s') = \infty$ following Equation 1. In detail, ϕ is set to a disjunction of atomic conjunctions so that (1) for each atomic conjunction c taking part in ϕ: $h^C(s, c) = \infty$; (2) for each $c \in \phi$ and for each $a \in \mathcal{A}[c]$, there must be some $c' \in \phi$ with $c' \subseteq R(c, a)$ (cf. middle case of Equation 1); and (3) there must be an atomic conjunction $c \in \phi$ so that $c \subseteq \mathcal{G}$ (cf. last case of Equation 1). In this way, CLone ensures that ϕ is self contained, i. e., that for each atomic conjunction $c \in \phi$, ϕ covers all possible ways of achieving c. In other words, if a state s' does not satisfy ϕ, i. e., $c \not\subseteq s'$ for every $c \in \phi$, then we obtain $h^C(s') = \infty$ as a direct consequence.

Implementation

CLone is implemented on top of Fast Downward (Helmert 2006), extended by the h^2 preprocessor (Alcázar and Torralba 2015). For h^C, following Hoffmann and Fickert (2015), we use counters over pairs (c, a) where $c \in C$, $a \in \mathcal{A}[c]$, and $R(c, a)$ does not contain a fact mutex. The depth-first search of CLone breaks ties (order of children) randomly .

Given a problem in form of a PDDL domain and a PDDL problem file, Fast Downward first compiles these files into an FDR planning task. All methods (and in particular the search), except of the computation and refinement of h^C, operate directly on this FDR encoding. Only the computation and refinement of h^C pretend to have a STRIPS encoding of the problem by considering variable value pairs as facts, and threating the actions accordingly.

CLone initializes C to the set of all unit conjunctions, i. e., $C = \{\{p\} \mid p \in \mathcal{F}\}$. Additionally, if the causal graph of the FDR task contains more than one maximal SCC, CLone adds the conjunctions of facts to C corresponding to the variable value assignments of all pairs of variables that are part of the root SCC. The intuition behind this is that the root component of a problem's causal graph is usually a central part of the problem structure, and having the pairs of the values of the corresponding variables often helps the refinement algorithm to find smaller sets X.

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".

References

Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. *ICAPS'15*.

Christer Bäckström, Peter Jonsson, and Simon Ståhlberg. Fast detection of unsolvable planning instances using local consistency. *SOCS'13*, 29–37.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):279–298, 1997.

Stefan Edelkamp. Planning with pattern databases. *ECP'01*, 13–24.

Niklas Eén and Niklas Sörensson. An extensible sat-solver. *SAT'03*, 502–518.

Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. *AIPS'00*, 140–149.

Patrik Haslum. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to h^m. *ICAPS'09*, 354–357.

Patrik Haslum. Incremental lower bounds for additive cost planning problems. *ICAPS'12*, 74–82.

Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the Association for Computing Machinery*, 61(3), 2014.

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

Jörg Hoffmann and Maximilian Fickert. Explicit conjunctions w/o compilation: Computing $h^{FF}(\pi^c)$ in polynomial time. *ICAPS'15*.

Jörg Hoffmann, Peter Kissmann, and Álvaro Torralba. "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability. *ECAI'14*.

Emil Keyder, Jörg Hoffmann, and Patrik Haslum. Improving delete relaxation heuristics through explicitly represented conjunctions. *Journal of Artificial Intelligence Research*, 50:487–533, 2014.

Joao Marques-Silva and Karem Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *DAC-01*.

Marcel Steinmetz and Jörg Hoffmann. Towards clause-learning state space search: Learning to recognize dead-ends. *AAAI'16*.

Álvaro Torralba and Vidal Alcázar. Constrained symbolic search: On mutexes, BDD minimization and more. *SOCS'13*, 175–183.