Incremental Search for Counterexample-Guided Cartesian Abstraction Refinement

Jendrik Seipp, Samuel von Allmen, Malte Helmert
University of Basel
Basel, Switzerland
{jendrik.seipp, malte.helmert}@unibas.ch, samuel.vonallmen@stud.unibas.ch

Abstract

Counterexample-guided Cartesian abstraction refinement has been shown to yield informative heuristics for optimal classical planning. The algorithm iteratively finds an abstract solution and uses it to decide how to refine the abstraction. Since the abstraction grows in each step, finding solutions is the main bottleneck of the refinement loop. We cast the refinements as an incremental search problem and show that this drastically reduces the time for computing abstractions.

Introduction

The most common method for solving optimal classical planning tasks is to use A* (Hart, Nilsson, and Raphael 1968) with an admissible heuristic (Pearl 1984). One way of obtaining such a heuristic is counterexample-guided abstraction refinement (CEGAR) for Cartesian abstractions (Clarke et al. 2003; Seipp and Helmert 2013; 2018). Algorithm 1 shows pseudo-code for the CEGAR algorithm. It starts with an initial coarse abstraction of the original task. For this work, we assume that the initial abstraction separates goal states from non-goal states (i.e., a state is a concrete goal state iff its abstract image is an abstract goal state) but is otherwise trivial. In a SAS$^+$ planning task (Bäckström and Nebel 1995), a Cartesian abstraction of this form is easy to construct and requires at most $n+1$ abstract states for a planning task with n state variables (Seipp and Helmert 2018).

Next, CEGAR iteratively searches for a minimum-cost abstract solution, finds out where it fails for the concrete task (e.g., due to a violated precondition) and refines the abstraction by splitting a single abstract state into two abstract states in such a way that the same failure cannot happen in the next iteration. If there is no abstract solution, we have shown unsolvability and can stop early. Similarly, we stop refining if the abstract solution also works for the concrete task: in this situation, an optimal plan has been found. Otherwise, we stop the refinement process after hitting a time or memory limit and use the resulting heuristic for an A* search.

Figures 1a and 1c illustrate how the REFINE function splits an abstract state (v in Figure 1a) into two new states (v_1 and v_2 in Figure 1c): we replace v by v_1 and v_2 and rewire all incoming and outgoing transitions of v into incoming and outgoing transitions of v_1 and/or v_2. (For self-loops on v, this means that we obtain self-loops in v_1 and/or v_2 and/or transitions between the two new states.)

Algorithm 1 CEGAR refinement loop that receives a classical planning task Π and returns an abstraction $\mathcal{T}$ of the state space of Π (Algorithm 1 by Seipp and Helmert 2018).

```
1: function CEGAR($\Pi$)
2:     $\mathcal{T} \leftarrow$ INITIALABSTRACTION($\Pi$)
3:     while not TERMINATE() do
4:         $\pi \leftarrow$ FINDSHORTESTPATH($\mathcal{T}$)
5:         if $\pi$ is "no path" then
6:             return task is unsolvable
7:         $\varphi \leftarrow$ FINDFLAW($\Pi$, $\pi$)
8:         if $\varphi$ is "no flaw" then
9:             return plan extracted from $\pi$
10:        $\mathcal{T} \leftarrow$ REFINE($\Pi$, $\mathcal{T}$, $\varphi$)
11:    return $\mathcal{T}$
```

To find abstract solutions in CEGAR, we can use any optimal algorithm, such as Dijkstra's algorithm (1959). However, because refinements can only increase goal distances, we can use the goal distances from previous iterations of the refinement loop as a heuristic for an A* search. Even this informed search, however, suffers from the fact that finding abstract solutions takes longer as the abstraction grows. In a typical scenario, we perform on the order of 10^5 refinement steps before terminating. This makes the abstract searches the main bottleneck of the refinement loop.

It is wasteful that each A* search starts from scratch, even though the abstraction only changes locally. We would like to reuse information from previous searches and only update it locally during a refinement. Fortunately, we can cast the problem of maintaining all shortest paths in the abstraction as a *dynamic shortest path* problem, which has been studied extensively in the literature (e.g., Ramalingam and Reps 1996; Frigioni, Marchetti-Spaccamela, and Nanni 2000; Koenig, Likhachev, and Furcy 2004). We show that each refinement step can be viewed in such a way that the critical step is the removal of a set of transitions, which allows us to apply a specialized dynamic shortest path algorithm by Frigioni, Marchetti-Spaccamela, and Nanni (2000).

Experimentally, we demonstrate that incremental shortest path computation speeds up the refinement loop drastically. For many tasks we observe a speedup factor of over 1000 for computing the abstract solutions, making the runtime for the shortest path computations negligible, even for large abstractions.

Background

We want to maintain a *shortest path* from every state to a fixed goal state in a *weighted transition system* that is refined iteratively. Formally, a *transition system* $\mathcal{T}$ is a directed, labeled graph with a finite set of *states* $S(\mathcal{T})$, a finite set of *labels* $L(\mathcal{T})$, a set $T(\mathcal{T})$ of labeled *transitions* $s \xrightarrow{\ell} s'$ with $s, s' \in S(\mathcal{T})$ and $\ell \in L(\mathcal{T})$, an *initial state* $s_0(\mathcal{T}) \in S(\mathcal{T})$, and a set $S_\star(\mathcal{T}) \subseteq S(\mathcal{T})$ of *goal states*.

We combine a transition system $\mathcal{T}$ and a positive *cost function cost* : $L(\mathcal{T}) \mapsto \mathbb{R}_{>0}$ to obtain a *weighted* transition system. (Our implementation supports zero-cost actions by assigning a sufficiently small positive cost ϵ to them.)

A *path* in a (weighted) transition system $\mathcal{T}$ from $s \in S(\mathcal{T})$ to $s' \in S(\mathcal{T})$ is a sequence π of transitions from $T(\mathcal{T})$ such that $\pi = \langle s^0 \xrightarrow{\ell_1} s^1, s^1 \xrightarrow{\ell_2} s^2, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n \rangle$, where $s^0 = s$ and $s^n = s'$. The empty path $\langle\rangle$ is a path from s to s for all states s. The *cost* of a path is the sum of its label weights. A path from s to s' is optimal if there is no path from s to s' with lower cost.

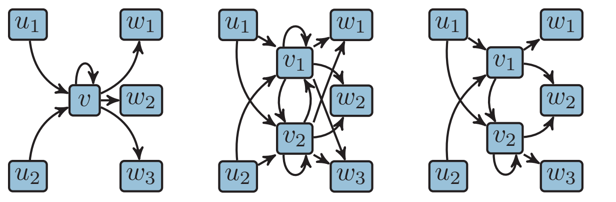
A *goal path* for $s \in S(\mathcal{T})$ is a path from s to any goal state $s' \in S_\star(\mathcal{T})$. Goal paths for the initial state are also called *solutions*. We call optimal goal paths *shortest paths*. A state is *solvable* if it has a goal path.

For each solvable state s we choose a shortest path π and define *parent*$(s) = \langle \ell, s' \rangle$ if π starts with $s \xrightarrow{\ell} s'$. If s is unsolvable or π is empty (because s is a goal state), we set *parent*(s) = *none*. We call the structure defined by the parent pointers the *shortest path tree*. For traversing this tree downwards, we define *children*$(s) = \{\langle \ell, s' \rangle \mid$ *parent*$(s') = \langle \ell, s \rangle\}$. Finally, we define $h(s)$ as the cost of π and $h(s) = \infty$ for unsolvable states s.

An *abstraction* (Helmert, Haslum, and Hoffmann 2007) is an equivalence relation over the set of concrete factored states. It is Cartesian if all equivalence classes are Cartesian sets. Each such set represents one abstract state.

Incremental Search

Instead of starting each abstract search from scratch in the refinement loop, we cast refining the abstraction and finding abstract solutions as a *dynamic shortest path* problem, also known as *incremental search*. Solving dynamic shortest path problems involves repeatedly finding shortest paths in a dynamically changing transition system. Between two searches, transition weights can increase or decrease, or transitions can be added or removed. The simplest method for solving a dynamic shortest path problem is to always run A* or Dijkstra's algorithm from scratch. When the transition system changes completely between two searches, this is actually often the preferable method. When changes are small,



(a) Before splitting v. (b) Copy v. (c) Prune transitions.

Figure 1: Splitting v into v_1 and v_2 in two steps. The first step copies v and its adjacent transitions while the second step removes spurious transitions.

however, specialized dynamic shortest path algorithms can be more efficient.

Dynamic shortest path algorithms usually assume a fixed set of states. In each round of the CEGAR refinement loop, however, a state is replaced by two new states. Therefore, it is not obvious how to treat finding abstract solutions in CEGAR as a dynamic shortest path problem.

To address this point, we conceptually divide a refinement into two steps. The first step introduces an additional state but otherwise maintains all shortest paths and goal distances. The second step removes some transitions. Only this step requires recomputing shortest paths, and because the only modifications are edge removals, we can use specialized dynamic shortest path algorithms tailored towards this case.

We illustrate the two-step refinement process in Figure 1: in the the first step from Figure 1a to Figure 1b we replace v by v_1 and v_2, but copy all incoming and outgoing transitions from v to both v_1 and v_2. For each self-loop at v this implies adding four transitions: from v_1 and v_2 to v_1 and v_2. It is easy to see that this change does not affect any shortest paths: for any state u with *parent*$(u) = \langle \ell, v \rangle$ we can now set either *parent*$(u) = \langle \ell, v_1 \rangle$ or *parent*$(u) = \langle \ell, v_2 \rangle$ without changing $h(u)$. Furthermore, v_1 and v_2 can inherit the shortest path of v, that is, we can set *parent*(v_1) = *parent*(v_2) = *parent*(v) and therefore $h(v_1) = h(v_2) = h(v)$. The second step from Figure 1b to Figure 1c removes all transitions that have no counterpart in the concrete transition system.

The dynamic shortest path algorithm INCREASE by Frigioni, Marchetti-Spaccamela, and Nanni (2000) handles transition removals (and weight increases) efficiently. Whenever some transitions are removed from transition system $\mathcal{T}$, it updates the shortest path information in time $O(p\, d \log n)$, where p is the number of states s for which the update changes *parent*(s), d is the maximum degree of $\mathcal{T}$, and $n = |S(\mathcal{T})|$. (The authors prove tighter bounds for graphs with special structures.) We can bound $p \leq M\, d$, where M is the number of states s for which $h(s)$ increases in this refinement step, because the parent of a state can only change if the goal distance of the previous parent increases. Moreover, if $\mathcal{T}$ is deterministic, d is bounded by the number of planning task actions, $|L(\mathcal{T})|$. So in this case we obtain an overall bound of $O(M|L(\mathcal{T})|^2 \log n)$, which is polynomial in the number of abstract states whose heuristic value improves and the (compact) description size of the planning task. In contrast, A* runtime grows with the *total* number of

Algorithm 2 Incrementally update all shortest paths and goal distances in abstract transition system $\mathcal{T}$ with transitions T after v has been split into v_1 and v_2.

```
1: function ComputeDirtyStates($v_1$)
2:     $D \leftarrow \emptyset$
3:     $C \leftarrow$ empty priority queue
4:     $C$.push($\langle h(v_1), v_1 \rangle$)
5:     while $C$ is not empty do
6:         $\langle h(s), s \rangle \leftarrow C$.popmin()
7:         if $\exists s \xrightarrow{\ell} s' \in T$ with $s' \notin D$
8:                 and $h(s) = cost(\ell) + h(s')$ then
9:             $parent(s) \leftarrow \langle \ell, s' \rangle$
10:        else
11:            $D \leftarrow D \cup \{s\}$
12:            for all $\langle \ell, s' \rangle \in children(s)$ do
13:                $C$.push($\langle h(s'), s' \rangle$)
14:    return $D$

15: procedure Incremental($T$, $v$, $v_1$, $v_2$)
16:    $h(v_1) \leftarrow h(v)$
17:    $h(v_2) \leftarrow h(v)$
18:    $parent(v_1) \leftarrow none$
19:    $parent(v_2) \leftarrow parent(v)$
20:    for all $\langle \ell, u \rangle \in children(v)$ do
21:        if $\exists u \xrightarrow{\ell'} v_2 \in T$ with $cost(\ell') = cost(\ell)$ then
22:            $parent(u) \leftarrow \langle \ell', v_2 \rangle$
23:        else
24:            $parent(u) \leftarrow \langle \ell, v_1 \rangle$
25:    $D \leftarrow$ ComputeDirtyStates($v_1$)
26:    $Q \leftarrow$ empty priority queue
27:    for all $s \in D$ do
28:        $h(s) \leftarrow \infty$
29:        for all $s \xrightarrow{\ell} s' \in T$ do
30:            if $cost(\ell) + h(s') < h(s)$ then
31:                $h(s) \leftarrow cost(\ell) + h(s')$
32:                $parent(s) \leftarrow \langle \ell, s' \rangle$
33:        if $h(s) < \infty$ then
34:            $Q$.push($\langle h(s), s \rangle$)
35:    while $Q$ is not empty do
36:        $\langle h(s), s \rangle \leftarrow Q$.popmin()
37:        for all $s' \xrightarrow{\ell} s \in T$ with $s' \in D$ do
38:            if $h(s) + cost(\ell) < h(s')$ then
39:                $h(s') \leftarrow h(s) + cost(\ell)$
40:                $parent(s') \leftarrow \langle \ell, s \rangle$
41:                $Q$.update($\langle h(s'), s' \rangle$)
```

abstract states, whether or not their heuristic value needs to be updated.

Algorithm 2 shows pseudo-code of the INCREASE procedure, adapted to our setting and notation. The INCREMENTAL procedure is called after splitting state v into two new states v_1 and v_2. It uses the insight that either v_1 or v_2 inherits the shortest path from v. To see this, assume that $\langle \ell, w \rangle$ is the parent of v. Because we have an induced abstraction, at least one of the two new states must have a transition via ℓ to w. We also know that at most one such transition exists, because the way we define parent edges coincides with the way we extract abstract solutions and we decide which state to split based on a flaw of the abstract solution. Thus, we know that either $v_1 \xrightarrow{\ell} w$ or $v_2 \xrightarrow{\ell} w$ exists and we assume the latter case in the pseudo-code.

The INCREMENTAL procedure starts by assigning the goal distance $h(v)$ to the two new states v_1 and v_2 (lines 16–17). Since v_2 inherits the shortest path from v, no further changes are needed for v_2. In contrast, $h(v)$ is only a lower bound for $h(v_1)$ and therefore $h(v_1)$ might increase later.

For each child u of v, we set the parent of u to v_2 whenever this is possible and to v_1 otherwise (lines 20–24). Here we use the fact that we can also select a transition to v_2 with a different label than the one stored as the parent of u if it has the same cost.

We set the parent of v_2 to *parent*(v) in line 19. The parent of v_1 is now undefined (line 18), and this is the reason why we may need to recompute something: the shortest path tree has become disconnected. Note that only states that are descendants of v_1 (including v_1 itself) may possibly need recomputation. We call states that need recomputation *dirty* and all other states *settled*.

We compute the set of dirty states with the COMPUTE-DIRTYSTATES function, which is based on the following observation: if the heuristic value of a state s does not change, then neither does the heuristic value of any of its descendants. The function potentially marks all descendants in the shortest path tree below v_1. Instead of marking all descendants as dirty, however, we only mark those states as dirty that cannot be reconnected (in the shortest path tree) to settled states at no extra cost. The algorithm uses the fact that all actions have a positive cost and therefore parents always have a strictly lower goal distance than their children.

After computing the set of dirty states, we perform a Dijkstra-like exploration to recompute all goal distances and shortest paths as follows (lines 26–41). The "initial state" of the search is a virtual state that represents all settled states. It is expanded first, with a cost of 0. Its outgoing transitions are all transitions (in the backward graph) that go from a settled state s to a dirty state s' with label ℓ, and the cost of the transition is $h(s) + cost(\ell)$. (Note that $h(s)$ for settled states is known.) After this initialization, we proceed with a normal Dijkstra search, but only consider transitions that lead from dirty to dirty states. (Note that every state we process has a parent because we only process solvable abstract states, and we never need to split the abstract goal state, which is the only solvable abstract state without a parent.)

Experiments

We implemented all algorithms in the Fast Downward planning system (Helmert 2006) and used the Downward Lab toolkit (Seipp et al. 2017) for running experiments on Intel Xeon Silver 4114 processors. Our benchmark set consists of all 1827 tasks without conditional effects from the optimal sequential tracks of the International Planning Competitions 1998–2018. We limit time by 30 minutes and memory by 3.5 GiB. All benchmarks, code and experiment data have

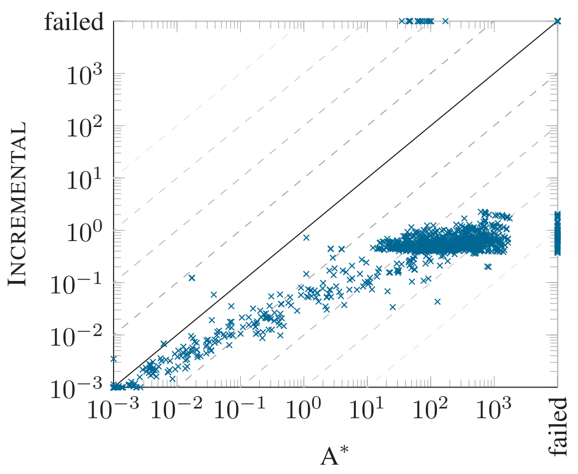


Figure 2: Time in seconds for finding abstract solutions in the refinement loop when computing a Cartesian abstraction with at most 10^5 states using either A* or INCREMENTAL.

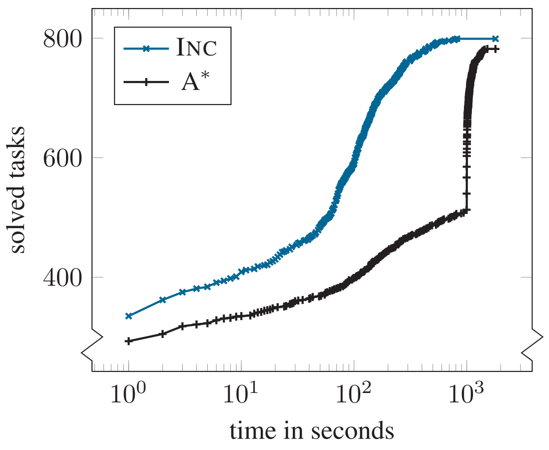


Figure 3: Number of solved tasks over time when using either INCREMENTAL or A* for finding abstract solutions. In both cases we stop refining and start the search after 10^3 seconds or when we approach the memory limit.

been published online (Seipp 2019; Seipp, von Allmen, and Helmert 2020a; 2020b).

In the first experiment, we build a Cartesian abstraction with at most 10^5 states using either A* or INCREMENTAL for computing shortest paths. Figure 2 compares the time for finding abstract solutions by the two approaches, where each data point is the sum of runtimes of all (up to 10^5) abstract searches for a given planning task. While the A* variant needs 1000 seconds and more to compute the abstract solutions, INCREMENTAL uses at most 2 seconds for all abstract searches. Also the relative runtimes differ drastically: with A* we spend 69% of the refinement loop runtime on abstract searches on average. This number decreases to 6% if we compute shortest paths incrementally.

The second experiment imposes no fixed limit on the number of abstract states. Instead, we stop refining the abstraction when we reach a time limit of 1000 seconds or approach the memory limit. At this point, we release the memory for the transitions (which account for the bulk of the CEGAR memory usage) and begin the A* search in the concrete state space with the resulting heuristic. Figure 3 shows the number of solved tasks over time by the two variants. For all evaluated time points, the INCREMENTAL variant solves more tasks than its A* counterpart. Overall, INCREMENTAL results in a coverage of 799 tasks compared to 782 with A*. INCREMENTAL solves as many tasks within 1 second as A* solves within 10 seconds, as many tasks within 10 seconds as A* solves within 121 seconds, and as many tasks within 100 seconds as A* solves within 1004 seconds. The A* version uses the full 1000 seconds to refine the abstraction for 814 tasks, approaches the memory limit for 576 tasks and finds a concrete solution for 419 tasks. In contrast, INCREMENTAL almost never reaches the 1000 second limit (2 tasks) and always either approaches the memory limit (1305 tasks) or finds a concrete solution (498 tasks) during the refinement loop.

Related Work

Cartesian abstractions also lend themselves well to online refinement during the search (Eifler and Fickert 2018). Whenever the search detects that the abstraction heuristic violates the Bellman equation (1957), we can refine the abstract states in the Cartesian abstraction that cause the error. In contrast to CEGAR, this approach involves no search for abstract solutions and therefore incremental search methods are not needed for online refinement.

The literature contains many algorithms for solving dynamic shortest path problems. The ones most closely related to Algorithm 2 are DynamicSWSF-FP (Ramalingam and Reps 1996) and Incremental A*, also known as LPA* (Koenig, Likhachev, and Furcy 2004). Like Algorithm 2, DynamicSWSF-FP maintains all shortest paths in a changing transition system, but it also supports decreasing transition costs. Incremental A* is a combination of DynamicSWSF-FP and A*: it reuses information from prior searches and prunes the search space with a heuristic.

Incremental search algorithms have been used for computing heuristics in automated planning by Liu, Koenig, and Furcy (2002), who extend DynamicSWSF-FP to speed up the computation of the h^{add} heuristic (Bonet and Geffner 2001). They exploit that consecutive heuristic evaluations often consider states that only differ in few state variables, so that h^{add} values of many state variables may remain unchanged when moving from one state to the next.

Conclusions

We showed that each refinement in the CEGAR loop for Cartesian abstractions can be viewed as a two-step process such that the first step maintains all shortest paths and the second step removes a set of transitions. This view allows us to use an incremental search algorithm to efficiently maintain all shortest paths. The new algorithm drastically reduces the time for finding abstract solutions and yields stronger heuristics in less time.

Acknowledgments

We have received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639).

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4):625–655.

Bellman, R. E. 1957. *Dynamic Programming*. Princeton University Press.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50(5):752–794.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Eifler, R., and Fickert, M. 2018. Online refinement of Cartesian abstraction heuristics. In Bulitko, V., and Storandt, S., eds., *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018)*, 46–54. AAAI Press.

Frigioni, D.; Marchetti-Spaccamela, A.; and Nanni, U. 2000. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms* 34(2):251–281.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

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

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong planning A*. *Artificial Intelligence* 155(1–2):93–146.

Liu, Y.; Koenig, S.; and Furcy, D. 2002. Speeding up the calculation of heuristics for heuristic search-based planning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI 2002)*, 484–491. AAAI Press.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Ramalingam, G., and Reps, T. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21(2):267–305.

Seipp, J., and Helmert, M. 2013. Counterexample-guided Cartesian abstraction refinement. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 347–351. AAAI Press.

Seipp, J., and Helmert, M. 2018. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research* 62:535–577.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Seipp, J.; von Allmen, S.; and Helmert, M. 2020a. Code from Seipp et al., ICAPS 2020. https://doi.org/10.5281/zenodo.3636137.

Seipp, J.; von Allmen, S.; and Helmert, M. 2020b. Experiment data from Seipp et al., ICAPS 2020. https://doi.org/10.5281/zenodo.3636139.

Seipp, J. 2019. STRIPS PDDL benchmarks from sequential optimization tracks of IPC 1998–2018. https://doi.org/10.5281/zenodo.2616479.