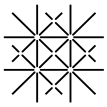

University
of Basel

Computing Abstract Plans for Counterexample-Guided Cartesian Abstraction Refinement

Bachelor thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence

Examiner: Dr. Gabriele Röger
Supervisor: Dr. Jendrik Seipp

Samuel von Allmen
samuel.vonallmen@stud.unibas.ch
2004-053-815

2019-05-10

Table of Contents

1 Introduction

Counterexample-guided abstraction refinement (CEGAR) is a way to incrementally compute abstractions of transition systems. It works by starting with a coarse abstraction (possibly one single abstract state containing all states of the original *concrete* system), and then repeating the following steps:

1. Find a solution in the abstraction, i.e., a series of transitions from the abstract state containing the concrete initial state to an abstract state containing a concrete goal state.
2. Apply the solution in the concrete state space. If it is applicable, we have solved the concrete problem.
3. Otherwise, find out where and why it fails.
4. Split the offending abstract state into two abstract states in a way that ensures that the same flaw cannot occur again, and recompute the possible transitions to and from these new abstract states.

As more states are split, and the abstraction grows in size, finding a solution for the abstract system becomes more and more costly. Because the abstraction grows incrementally, however, it is possible to maintain heuristic information about the abstract state space, allowing the use of informed search algorithms like A*.
As the quality of the heuristic is crucial to the performance of informed search, the method for maintaining the heuristic has a significant impact on the performance of the abstraction refinement as a whole. In this thesis, we investigate different methods for maintaining the value of the perfect heuristic $h^\star$ at all times and evaluate their performance.

2
Background

2.1 Definitions

The following definitions largely follow those in Seipp and Helmert [3]:

2.1.1 Planning Tasks and Transition Systems

Def. 1 (Planning tasks)**.** A *planning task* is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where:

- $\mathcal{V} = \langle v_0, \ldots, v_n \rangle$ is a finite sequence of *state variables*, each with an associated finite domain $\mathit{dom}(v_i)$.

  An *atom* is a pair $\langle v, d \rangle$ with $v \in \mathcal{V}$ and $d \in \mathit{dom}(v)$.

  A *partial state* s is an assignment that maps a subset $\mathit{vars}(s)$ of $\mathcal{V}$ to values in their respective domains. Alternatively, it can be interpreted as a set of atoms on different variables. $s[v] \in \mathit{dom}(v)$ is the value assigned to v by s. Partial states defined on all variables are called *states*. The set of all states of Π is called $\mathcal{S}(\Pi)$.

  The *update* of partial state s with partial state t, written $s \oplus t$, is the partial state with $\mathit{vars}(s \oplus t) = \mathit{vars}(s) \cup \mathit{vars}(t)$, and $(s \oplus t)[v] = \begin{cases} t[v], & \text{if } v \in \mathit{vars}(t) \\ s[v], & \text{if } v \in \mathit{vars}(s) \setminus \mathit{vars}(t) \end{cases}$

- $\mathcal{O}$ is a finite set of *operators*. Each operator o has a precondition $\mathit{pre}(o)$, an effect $\mathit{eff}(o)$ and a cost $\mathit{cost}(o) \in \mathbb{R}_0^+$. $\mathit{pre}(o)$ and $\mathit{eff}(o)$ are partial states. An operator $o \in \mathcal{O}$ is applicable in state s if $\mathit{pre}(o) \subseteq s$. Applying o to s results in $s[\![o]\!] = s \oplus \mathit{eff}(o)$. The partial state defined by $\mathit{pre}(o) \oplus \mathit{eff}(o)$ is a called the postcondition $\mathit{post}(o)$.

- $s_0 \in \mathcal{S}(\Pi)$ is the *initial state* and $s_\star$ is a partial state called the *goal*.

Def. 2 (Transition systems, regressions, plans and traces)**.** A *transition system*, also called a *state space*, is a tuple $\mathcal{T} = \langle S, \mathcal{L}, T, s_0, S_\star \rangle$, where:

- S is a finite set of *states*. $s_0 \in S$ is the *initial state* and $S_\star \subseteq S$ is the set of *goal states*.

- $\mathcal{L}$ is a finite set of *labels*.

- $T \subseteq S \times \mathcal{L} \times S$ is a set of *transitions* $s \xrightarrow{l} s'$ from $s \in S$ to $s' \in S$ with label $l \in \mathcal{L}$.

The *regression* of a set of states $S' \subseteq S$ with respect to a label $l \in \mathcal{L}$ is defined as $regr(S', l) = \{s \in S \mid s \xrightarrow{l} s' \in T \wedge s' \in S'\}$.
A sequence of transitions $\langle s^0 \xrightarrow{l_1} s^1, s^1 \xrightarrow{l_2} s^2, \ldots s^{k-1} \xrightarrow{l_k} s^k \rangle$ is called a *trace* from s^0 to s^k. A trace from s to a state $s_\star \in S_\star$ is called a *goal trace* for s. The empty sequence is considered a trace from s to itself for all states s. The sequence of labels $\langle l_1, l_2, \ldots, l_k \rangle$ of a goal trace for s is called a *plan* for s. Plans and goal traces for s_0 are also called plans and goal traces, respectively, for $\mathcal{T}$.

A planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ induces a transition system $\mathcal{T}$ with states $S = \mathcal{S}(\Pi)$, initial state s_0, goal states $S_\star = \{s \in \mathcal{S}(\Pi) \mid s_\star \subseteq s\}$ and transitions $T = \{s \xrightarrow{o} (s \oplus \mathit{eff}(o)) \mid s \in \mathcal{S}(\Pi), o \in \mathcal{O}, \mathit{pre}(o) \subseteq s\}$. Plans for $\mathcal{T}$ are also called plans for Π.

2.1.2 Optimal Planning

Weighting the transition system with a cost function $cost : \mathcal{L} \mapsto \mathbb{R}_0^+$, every plan $\langle l_1, l_2, \ldots, l_k \rangle$ has a *plan cost* of $\sum_{i=1}^{k} cost(l_i)$. A plan for Π (or its induced transition system $\mathcal{T}$) such that no other plan with a smaller plan cost exists, is called *optimal*. Optimal planning is the problem of finding such an optimal plan, or proving that no plan for Π exists at all.
A (weighted) transition system is a (weighted) directed graph (possibly with parallel edges). Optimal planning on such a system can therefore be performed by employing graph search algorithms like Dijkstra's algorithm (Dijkstra [1]) or A* (Hart et al. [2]).

2.1.3 Cartesian Abstractions

Def. 3 (Induced abstraction)**.** Let $\mathcal{T} = \langle S, \mathcal{L}, T, s_0, S_\star \rangle$ be a transition system induced by a planning task Π, and $\sim$ an equivalence relation on S. Its equivalence classes are then called *abstract states*, and the abstract state to which the concrete state s belongs is written $[s]_\sim$ or $[s]$, if clear from context.
$\sim$ then induces the *abstract transition system*, or *induced abstraction* $\mathcal{T}'$ with states $S' = \{[s] \mid s \in S\}$, transitions $T' = \{[s] \xrightarrow{l} [s'] \mid s \xrightarrow{l} s' \in T\}$, initial state $[s_0]$ and goal states $S'_\star = \{[s] \mid s \in S_\star\}$.

In the context of abstractions, the planning task Π on which the abstraction is based is also called the *concrete* task for the sake of clarity. Analogously, the states, transitions, traces and plans of Π and $\mathcal{T}$ are called *concrete* to distinguish them from abstract ones.
Every concrete plan for Π is also an abstract plan for $\mathcal{T}'$. This can easily be seen as for every transition $s \xrightarrow{l} s' \in T$ there is a corresponding transition $[s] \xrightarrow{l} [s'] \in T'$. Note that $[s]$ and $[s']$ might well be identical; the resulting transition $[s] \xrightarrow{l} [s]$ is called a *looping transition*. As a consequence, if there is no plan for $\mathcal{T}'$, the concrete task Π is proven to be unsolvable as well.

Def. 4 (Cartesian sets and Cartesian abstractions)**.** A set of states for a planning task with variables $\langle v_1, \ldots, v_n \rangle$ is called *Cartesian* if it is of the form $A_1 \times A_2 \times \cdots \times A_n$, where $A_i \subseteq dom(v_i)$ for all $1 \leq i \leq n$.
An abstraction is called *Cartesian* if all its abstract states are Cartesian sets.

For an abstract state $a = A_1 \times A_2 \times \cdots \times A_n$, we define $dom(v_i, a) = A_i$ as the set of values that variable v_i can have in a.

Def. 5 (Induced Cartesian sets)**.** Every partial state s of a planning task with variables $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ induces a Cartesian set, defined as

$$\mathit{Cartesian}(s) = A_1 \times \cdots \times A_n, \text{ where } A_i = \begin{cases} \{s[v_i]\}, & \text{if } v_i \in \mathit{vars}(s) \\ \mathit{dom}(v_i) & \text{otherwise} \end{cases}$$

Examples of Cartesian sets:

- The set of goal states of a planning task Π is $\mathit{Cartesian}(s_\star)$.
- The set of states where a particular operator o is applicable is $\mathit{Cartesian}(\mathit{pre}(o))$.
- The set $\mathcal{S}(\Pi)$ of all states is the Cartesian set $\mathit{dom}(v_1) \times \cdots \times \mathit{dom}(v_n)$. The abstraction having this set as its only abstract state is called the *trivial abstraction* of Π and is a Cartesian abstraction.

2.2 Counterexample-Guided Cartesian Abstraction Refinement

Algorithm 2.1 The main loop of the CEGAR algorithm. Given a planning task Π, it either returns a plan, proves that no plan exists (i.e., that Π is unsolvable), or returns an abstraction of Π.

```
1:  function CEGAR(Π)
2:      T' ←TRIVIALABSTRACTION(Π)
3:      T' ←INITIALREFINEMENT(T')
4:      while not TERMINATIONCONDITION() do
5:          τ' ←FINDOPTIMALTRACE(T')
6:          if τ' is "no trace" then
7:              return task is unsolvable
8:          φ ← FINDFLAW(τ')
9:          if φ is "no flaw" then
10:             return plan extracted from τ'
11:         T' ← REFINE(T', φ)
12:     return T'
```

The workings, implementation and proof of correctness of the Cartesian CEGAR algorithm are described in detail in Seipp and Helmert [3]. Following, we will give a short overview of the most important aspects for this thesis.

2.2.1 Main Loop

The main loop of the refinement algorithm is shown in Algorithm 2.1. At all points, $\mathcal{T}'$ is a Cartesian abstraction. In each iteration, a search is run in the abstraction to find an optimal goal trace τ' for $\mathcal{T}'$. Strategies for finding this trace will be discussed at length in the next chapter. If no such trace can be found, the abstraction, and therefore the concrete task, is proven to be unsolvable. Otherwise, we test whether the abstract goal trace is also a concrete goal trace by applying it, operator by operator, to the concrete system starting

with the concrete initial state s_0. If this works, we can extract a plan for Π from τ' and have thus solved Π. In most cases, however, the abstract plan will fail in the concrete system, producing a *flaw.* In this case, the abstraction is refined with regard to this flaw, splitting an abstract state into two new Cartesian sets such that the same flaw cannot occur again in subsequent iterations. Then, a new search is run on the refined abstraction, repeating this loop until the task is either solved, proven to be unsolvable, or a preset limit (e.g., with regards to time or memory) is reached. In the latter case, we return the abstraction in its current state, which can then serve as a basis for a heuristic for the concrete task.

2.2.2 Flaws and Refinement

A *flaw* is a pair $\langle s, c \rangle$ of a concrete state s and a Cartesian set $c \subseteq [s], c \neq \emptyset$ such that the abstract plan failed in the concrete system because $s \notin c$.
To find a flaw, we start with the initial concrete state $s = s_0$ and iteratively apply each operator o from the abstract goal trace τ' to s until we encounter one of the following situations:

1. o is not applicable in s. c is then the Cartesian set of concrete states from $[s]$ in which o is applicable, $[s] \cap \mathit{Cartesian}(\mathit{pre}(o))$.
2. o is applicable in s, but the resulting concrete state $s[\![o]\!]$ is not contained in the next abstract state in the goal trace. In this case, c is the regression of the desired abstract state with respect to o.
3. We have applied the last operator in the goal trace, but the resulting concrete state is not a goal state. c is then the set of goal states in $[s]$, $[s] \cap \mathit{Cartesian}(s_\star)$.

Once a flaw $\langle s, c \rangle$ has been found, $[s]$ is split into two new abstract states $a, b \subset [s]$ such that $s \in a$, $c \subseteq b$, $a \cap b = \emptyset$ and $a \cup b = [s]$.

2.2.3 Initial Refinement

To avoid possible complications that could arise from having to split (i.e., refine) abstract goal states, the abstraction is prepared by splitting the single abstract state of the trivial abstraction into several abstract states such that all of them either contain no concrete goal states at all, or contain only concrete goal states. If the transition system is induced by a planning task with a single (partial) goal state $s_\star$, this results in one single abstract goal state $\mathit{Cartesian}(s_\star)$.

2.3 Fast Downward

All work was implemented in the Fast Downward planning system[1]. The search component of the planner is implemented in C++.

[1] http://www.fast-downward.org

3
Finding abstract solutions

An important part for the performance of the CEGAR algorithm is the speed with which optimal goal traces, or *abstract solutions*, are found in the abstract transition system. The informed search algorithm A* (Hart et al. [2]) is guaranteed to find an optimal solution when provided with an admissible heuristic, i.e., a heuristic such that $h(s) \leq h^\star(s)$ for all states s (*abstract* states in this case, as we are performing a search on an abstract transition system), $h^\star(s)$ being the *perfect heuristic* for state s, telling us the cost of an optimal goal trace, also called the *goal distance*, for s.
As the abstraction increases in size by splitting abstract states into smaller ones, goal distances can only increase. The goal distance of the state being split is therefore an admissible heuristic for both states resulting from the split. For the states that are not being split, heuristic values that were admissible in previous iterations will stay admissible for all subsequent iterations.
Following, we will explain several algorithms that were evaluated in order to optimize the computation speed of the abstract search.

3.1 A* with g-based Heuristic (`gval-astar`)

After an A* search, all states s that have been visited by the search will have a g-value, $g(s)$, giving the cost of a path from s_0 to s. Subtracting this value from the path cost $C^\star$ of the found goal trace τ yields an admissible heuristic. We therefore set $h(s) \leftarrow max(C^\star - g(s), h(s))$ if $g(s) < \infty$.

Proof. Let $g^\star(s)$ be the cost of an *optimal* path from s_0 to s. The actual g-value cannot be lower than this value, so we have:

$$g(s) \geq g^\star(s).$$

Let us also assume that $g^\star(s) < \infty$ and $h^\star(s) < \infty$, as otherwise the state would be either unreachable from the initial state, or unsolvable and therefore not part of any goal trace.
As $h^\star(s)$ is the cost of an optimal goal trace for s, $g^\star(s) + h^\star(s)$ is the minimum cost of a path from s_0 over s to a goal. As τ is an optimal goal trace for s_0, we know that for all

states s the following must apply:

$$g^\star(s) + h^\star(s) \geq C^\star.$$

Combining these inequalities yields $g(s) + h^\star(s) \geq C^\star$. If $g(s) < \infty$, this results in $h^\star(s) \geq C^\star - g(s)$. □

This algorithm was already implemented in the Fast Downward planning system by Jendrik Seipp before the start of this thesis, and served as a baseline the following approaches were compared against. The original implementation and reasoning can be found at http://hg.fast-downward.org/file/7b26321cb582/src/search/cegar/abstract_search.cc, lines 49–83.

3.2 A* with Full Dijkstra (`full-astar`)

Algorithm 3.1 Dijkstra's algorithm computes an optimal path to a goal state $g \in G$ for each state. It returns a distance $d(s)$ and a transition $spt(s)$ for each state, such that iteratively following the transitions in spt, starting in s, will lead to a goal state on an optimal path with cost $d(s)$.

```
1:  function FindOptimalDistances()
2:      Q ← empty queue.
3:      d(s) ← ∞ for all states.
4:      for all g ∈ G do
5:          d(g) ← 0
6:          spt(g) ← none
7:          Q.push(⟨0, g⟩)
8:      while Q not empty do
9:          Q.pop(⟨d, s⟩) based on smallest d
10:         if d(s) < d then
11:             continue
12:         for all s' --o--> s do
13:             d' ← d(s) + cost(o)
14:             if d' < d(s') then
15:                 d(s') ← d'
16:                 spt(s') ← (s' --o--> s)
17:                 Q.push(⟨d', s'⟩)
18:     return d, spt
```

The most straightforward approach to maintaining $h^\star$ at all times was to run a modified version of the algorithm of Dijkstra [1] as detailed in Algorithm 3.1 after each refinement step, setting $h(s)$ to the value $d(s)$ returned by the algorithm. As Dijkstra's algorithm returns an optimal path for each state, we know that $d(s) = h^\star(s)$. While this computation is obviously very expensive, supplying A* with $h^\star$ did improve the performance over the admissible heuristic of the baseline algorithm.

3.3 SPT with Full Dijkstra (`full-spt`)

As Algorithm 3.1 simultaneously computes a *shortest-path tree*, or SPT, the task of finding a goal trace can be simplified to a simple tree traversal as detailed in Algorithm 3.2, further

Algorithm 3.2 Traversal of the shortest-path tree to find a goal trace

```
1: function FindTrace(spt)
2:     τ ← empty
3:     s ← s_0
4:     while s is not a goal state do
5:         (s -o-> s') ← spt(s)
6:         append s -o-> s' to τ
7:         s ← s'
8:     return τ
```

improving search performance over a perfectly-informed A* search.

3.4 SPT with Backward-expanded Orphaned List (`obw-spt`)

Algorithm 3.3 Orphaned list algorithm with backward expansion. Recomputes the goal distances and shortest-path tree after state s_{old} has been split into s_1 and s_2. For clarity of notation, the successor function $succ(s)$ is defined as the target state of the transition from the SPT: $spt(s) = s \xrightarrow{o} s' \Rightarrow succ(s) = s'$

```
1: function UpdateDistances(s_old, s_1, s_2, d, spt)
2:     Q ← empty queue
3:     mark s_1, s_2 as orphaned
4:     for all s with succ(s) = s_old do
5:         RecursiveInsert(s)
6:     d(s) ← ∞ for all states s that are orphaned.
7:     for all s is not orphaned do
8:         for all s' -o-> s with s' orphaned do
9:             if d(s) + cost(o) < d(s') then
10:                d(s') ← d(s) + cost(o)
11:                spt(s') ← (s' -o-> s)
12:    for all orphaned states s with d(s) < ∞ do
13:        Q.push(⟨d(s), s⟩)
14:    while Q not empty do
15:        ⟨d, s⟩ ← Q.pop() based on smallest d
16:        if d(s) < d then
17:            continue
18:        for all s' -o-> s with s' orphaned do
19:            d' ← d(s) + cost(o)
20:            if d' < d(s') then
21:                d(s') ← d'
22:                spt(s') ← (s' -o-> s)
23:                Q.push(⟨d', s'⟩)
24:    return d, spt
25: function RecursiveInsert(s)
26:    mark s as orphaned
27:    for all s' with succ(s') = s do
28:        RecursiveInsert(s')
```

Even though the transition system changes with each iteration of the refinement process, recomputation of distance and shortest-path information is not necessary for all states.

One possible approach is therefore to identify all states that can possibly need updating, marking them as *orphaned states*. Algorithm 3.3 achieves this by first marking as orphaned the new states s_1 and s_2 that did not exist in the previous iteration, plus all states where the shortest-path information leads to the state s_{old} that no longer exists. Then, all states where the shortest-path information leads to an orphaned state are recursively *inserted into the orphaned list*, i.e., marked as orphaned states (Line 5). For all other states, $d(s)$ as well as $spt(s)$ still correspond to an optimal goal trace in the new transition system. These states will not need to have any of their information updated.
After that, in a step called *initial expansion*, all orphaned states that have at least one transition to a non-orphaned state are updated (Lines 7-11). These are then used to seed the queue of a modified Dijkstra's algorithm that ignores non-orphaned states and updates all reachable orphaned states (Line 14ff). As a result, all states s now again have a value $d(s)$ that corresponds to the cost of an optimal goal trace from s, and a transition $spt(s)$ that corresponds to the first element of such a trace, if these properties were given for the transition system before the refinement of s_{old} into s_1 and s_2.

3.5 SPT with Forward-expanded Orphaned List (`ofw-spt`)

Algorithm 3.4 Detail of the orphaned list algorithm with forward expansion.

```
1: for all s' is orphaned do
2:     for all s' -o-> s with s not orphaned do
3:         if d(s) + cost(o) < d(s') then
4:             d(s') ← d(s) + cost(o)
5:             spt(s') ← (s' -o-> s)
```

A variant of the previous approach is to perform the initial expansion slightly differently. Instead of looping over all *non-orphaned* states to search for transitions $s' \xrightarrow{o} s$ from an orphaned to a non-orphaned state, we loop over all *orphaned* states. Lines 7-11 from Algorithm 3.3 are thus replaced with Algorithm 3.4. The result will be the same as in `obw-spt` considering the value of $d(s)$, but if multiple optimal goal traces exist for a state, $spt(s)$ now might indicate a different, but still optimal, goal trace.

3.6 SPT with Filtered Orphaned List (`filter-spt`)

Not all orphaned states need updating of their distance $d(s)$ as well as their shortest-path information $spt(s)$. If there is a non-orphaned state s' with $s \xrightarrow{o} s'$ and $d(s) = cost(o)+d(s')$, it is sufficient to *reconnect* s by setting $spt(s) = (s \xrightarrow{o} s')$ and leaving $d(s)$ unchanged. In that case, states s'' with $succ(s'') = s$ will not have to be changed at all, as their SPT then still leads to an optimal goal trace with cost $d(s'')$.
To further reduce the number of states that have to be recomputed, Algorithm 3.5 therefore filters possible orphaned states through a *candidate queue*, eliminating states that can be reconnected without changing $d(s)$. The candidate queue C is first seeded with the new states s_1 and s_2 as well as all states s with $succ(s) = s_{old}$. These are then tested for the

Algorithm 3.5 Filtered orphaned list algorithm with forward expansion. Screens candidate states for the possibility of reconnecting them to non-orphaned states.

```
1: function UpdateDistances(s_old, s_1, s_2)
2:     Q ← empty queue
3:     C ← empty queue
4:     C.push(⟨d(s_old), s_1⟩)
5:     C.push(⟨d(s_old), s_2⟩)
6:     for all s with succ(s) = s_old do
7:         C.push(⟨d(s), s⟩)
8:     while C not empty do
9:         ⟨d, s⟩ ← C.pop() by lowest d.
10:        for all s -o-> s' with s' not orphaned do
11:            if d = cost(o) + d(s') then
12:                spt(s) = (s -o-> s')
13:                mark s as reconnected
14:                break
15:        if s has not been reconnected then
16:            mark s as orphaned
17:            for all s' with succ(s') = s do
18:                C.push(s)
19:    d(s) ← ∞ for all orphaned states.
20:    for all s' is orphaned do
21:        for all s' -o-> s with s not orphaned do
22:            if d(s) + cost(o) < d(s') then
23:                d(s') ← d(s) + cost(o)
24:                spt(s') ← (s' -o-> s)
25:    for all orphaned states s with d(s) < ∞ do
26:        Q.push(⟨d(s), s⟩)
27:    while Q not empty do
28:        ⟨d, s⟩ ← Q.pop() based on smallest d
29:        if d(s) < d then
30:            continue
31:        for all s' -o-> s with s' orphaned do
32:            d' ← d(s) + cost(o)
33:            if d' < d(s') then
34:                d(s') ← d'
35:                spt(s') ← (s' -o-> s)
36:                Q.push(⟨d', s'⟩)
37:    return d, spt
38: function RecursiveInsert(s)
39:    mark s as orphaned
40:    for all s' with succ(s') = s do
41:        RecursiveInsert(s')
```

possibility of reconnection. Only if this fails, they are marked as orphaned, and all states whose SPT points to the newly orphaned state are inserted into the candidate queue.
After all candidates have been either reconnected or marked as orphans, the remaining orphans are processed as described in Section 3.5.
The candidate queue relies on the assumption that $succ(s') = s$ implies $d(s) < d(s')$, i.e., that $cost(o) > 0$ for all operators o. Then, Line 18 can only add candidates with a strictly larger d to the queue, and states can only be reconnected to states with a strictly smaller d, ensuring the eventual termination of the queue. If this assumption is violated, a state s could possibly be reconnected to another state s' that will later be added to the queue itself. When s' is then popped from the queue, it will again add s to the queue in Line 18, leading to an infinite loop. Therefore, planning tasks with *zero-cost operators*, i.e. o with $cost(o) = 0$, cannot be solved with this algorithm.

4

Experiments

The different algorithms were tested on a set of 1827 benchmark tasks[2], each until either the abstraction reached a size of 20000 states, the task was proven to be unsolvable, a memory limit of 3584 MiB, or a time limit of 30 minutes was reached. In this chapter, relative scatter plots will only display those tasks where both considered algorithms finished without reaching the time or memory limit.

4.1 A* Performance With Perfect Information

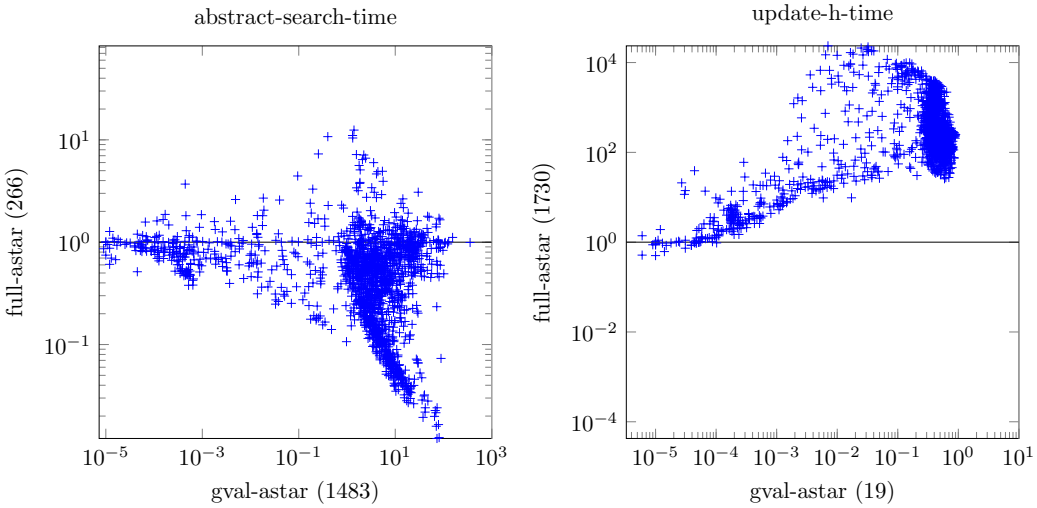

Figure 4.1: The x axis indicates the time (in seconds) it took the `gval-astar` algorithm to find an abstract plan (*abstract-search-time*), or to update the heuristic information (*update-h-time*), respectively. The y axis indicates the factor by which this time increased (upper half) or decreased (lower half) when utilizing the `full-astar` algorithm

When comparing `full-astar` with the baseline `gval-astar`, the search performance of the A* algorithm for finding abstract plans increased for 1483 of the tested tasks while

[2] https://bitbucket.org/aibasel/downward-benchmarks

decreasing only for 266 tasks (Fig. 4.1), showing the power of perfect information. However, as can be seen in the same figure, fully recomputing $h^\star$ for every state in every iteration was costly and in some cases took up to 10^4 times longer than in the baseline algorithm.

4.2 Tree traversal

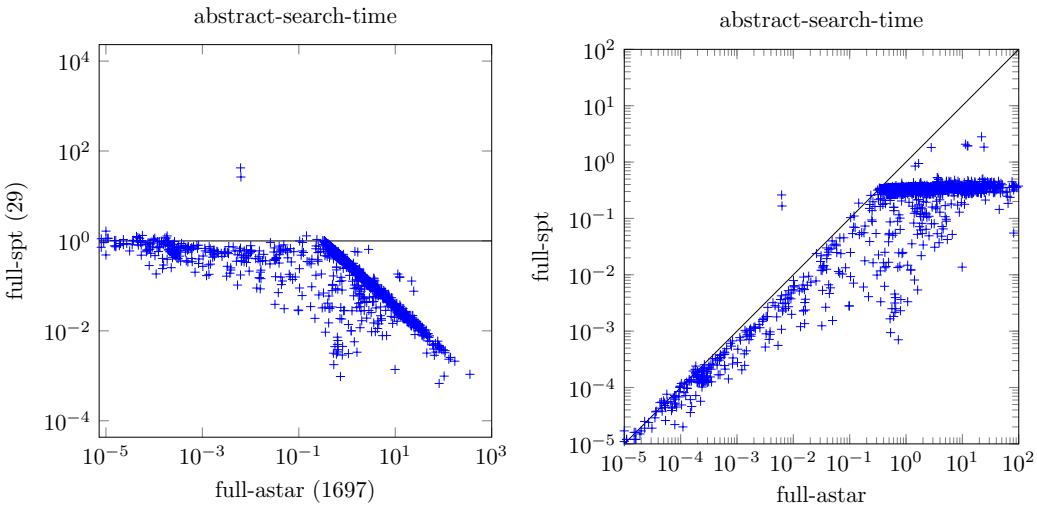

Figure 4.2: Comparison of the abstract search times for A* vs. SPT-traversal. The plot on the left is a relative scatter plot like those in Fig. 4.1, the one on the right plots the absolute search times of both algorithms.

While an A* informed search already performs better under perfect information, a significant further speedup was achieved by exploiting the shortest-path tree that was built alongside the heuristic and replacing the A* search with a simple tree traversal. Not only did this approach increase the search performance for virtually all tested tasks (1697 of 1726), it also effectively put a hard limit on search time (see Fig. 4.2), because the time for a tree traversal is bounded by the size of the abstraction, which, due to the experiment design, was effectively bounded at 20000 states.

4.3 Using an Orphaned List

Reducing the number of states that were actually recalculated by utilizing the orphaned list algorithms from Section 3.4 and Section 3.5 improved the performance of the heuristic-keeping for most tasks, as can be seen in Fig. 4.3, with the forward-expanded version significantly outperforming the backward-expanded version.

The filtered orphaned list from Algorithm 3.5 (`filter-spt`) was inapplicable for 350 of the tested tasks due to zero-cost operators (see Section 3.6). For the other tasks, it showed a slight, but inconsistent improvement over the forward-expanded orphaned list from Algorithm 3.4 (Fig. 4.4).

A final comparison of the most efficient algorithms `ofw-spt` and `filter-spt` against the baseline algorithm `gval-astar` as shown in Fig. 4.5 reveals that both algorithms

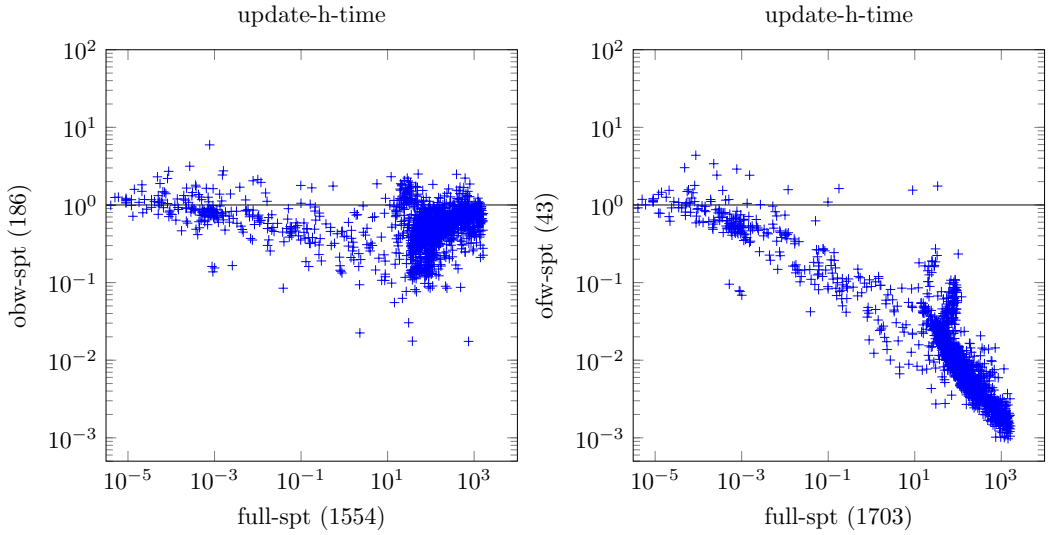

Figure 4.3: The x axis indicates the time needed for updating the h-values with a "naive" Dijkstra-algorithm (Algorithm 3.1), the y axis the factor by which this time changed when applying Algorithm 3.3 (left) or Algorithm 3.4 (right).

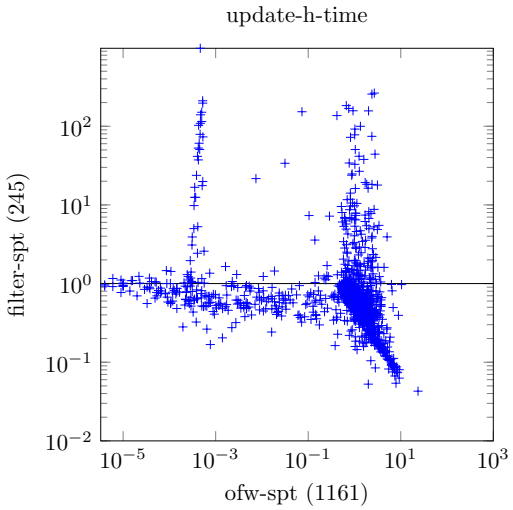

Figure 4.4: `filter-spt` shows inconsistent improvement over `ofw-spt`.

outperform the baseline algorithm in a significant portion of the compared tasks when considering the overall time for building the abstraction, which includes the time for finding an abstract trace as well as the time for updating the heuristic information, along with other components like the time for finding flaws, or refining the abstraction. Specifically, `ofw-spt` showed an improvement for 1370 of 1813 tasks (75.6%), while `filter-spt` outperformed `gval-astar` for 1069 of 1405 tasks (76.1%).

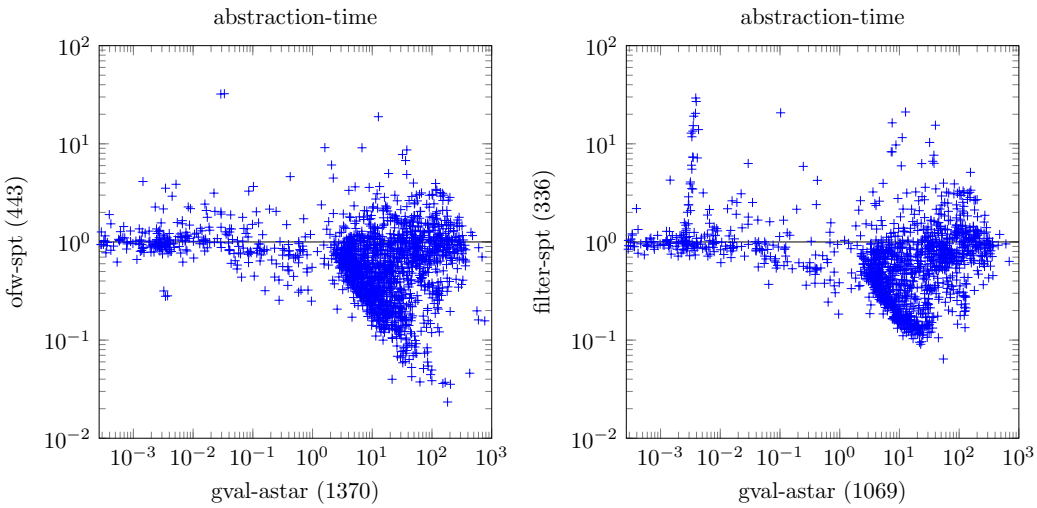

Figure 4.5: Comparing the full time for building the abstraction, comparing the baseline algorithm with `ofw-spt` as well as `filter-spt`.

5
Conclusion

We have shown that there is significant potential for improving the performance of CEGAR by maintaining shortest-path information at all times, if heuristic-keeping is performed by an efficient algorithm. Namely, the algorithms `ofw-spt` and `filter-spt`, described in Section 3.5 and Section 3.6, respectively, show an improvement over `gval-astar`, showing that the cost of maintaining optimal path information can be offset by replacing the abstract search with a simple tree traversal.
The following amendments to the algorithms presented in this thesis could possibly lead to further performance gains:

- Extend `filter-spt` to tasks featuring zero-cost operators. This could be achieved either by transforming the zero-cost operators into ε-cost operators, assigning a cost $\varepsilon > 0$ to them, or by adding an ancestor-check to the data structure containing the SPT, ensuring that no state is ever reconnected with one that is its descendant in the SPT.

- An option for searching the SPT in reverse, i.e. a function that directly provides $s \mapsto \{s' \mid succ(s') = s\}$ without the necessity to loop over all incoming transitions of s.

Bibliography

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

[2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC–4(2):100–107, 1968.

[3] Jendrik Seipp and Malte Helmert. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577, 2018.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Samuel von Allmen

Matriculation number — Matrikelnummer
2004-053-815

Title of work — Titel der Arbeit
Computing Abstract Plans for Counterexample-Guided Cartesian Abstraction Refinement

Type of work — Typ der Arbeit
Bachelor thesis

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 2019-05-10

Signature — Unterschrift