University of Basel

Generation of Domain Abstractions using Counterexample-Guided Abstraction Refinement

Bachelor's thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Clemens Büchner

Raphael Kreft
r.kreft@unibas.ch
2019-058-148

June 30, 2022

Acknowledgments

First of all I want to thank Prof. Malte Helmert for the opportunity to work on such an interesting Bachelor's Thesis in his research group. Another big thank you goes to Clemens Büchner for supervising the project. I really appreciated our weekly meetings that were always a source of valuable feedback and great conversations. Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing center at the University of Basel.

Abstract

In optimal classical planning, informed search algorithms like A* need admissible heuristics to find optimal solutions. Counterexample-guided abstraction refinement (CEGAR) is a method used to generate abstractions that yield suitable abstraction heuristics iteratively. In this thesis, we propose a class of CEGAR algorithms for the generation of domain abstractions, which are a class of abstractions that rank in between projections and Cartesian abstractions regarding the grade of refinement they allow. As no known algorithm constructs domain abstractions, we show that our algorithm is competitive with CEGAR algorithms that generate one projection or Cartesian abstraction.

Table of Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

2 Background **3**

2.1 Planning Tasks, State spaces and Plans . . . 3

2.2 Heuristic Functions and Informed Search . . . 4

2.3 Abstractions . . . 5

2.4 Domain Abstractions . . . 6

2.5 Counterexample-Guided Abstraction Refinement . . . 7

3 Constructing Domain Abstractions with CEGAR **8**

3.1 Initial Abstraction Selection . . . 9

3.2 Retrieve Flaws . . . 10

3.2.1 Find a plan in the abstract state space . . . 10

3.2.2 Find a flaw . . . 11

3.3 Refinement of the Abstraction . . . 11

3.3.1 Splitmethod . . . 12

3.3.1.1 Single Value Split . . . 12

3.3.1.2 Uniform Random Split . . . 12

3.3.2 Which fact pairs to consider for refinement . . . 12

3.3.2.1 Split along one flaw assignment . . . 13

3.3.2.2 Split along all flaw assignments . . . 13

3.4 Obtain the heuristic values . . . 14

3.4.1 Precalculation . . . 14

3.4.2 On demand computation . . . 14

4 Benchmarks and Evaluation **15**

4.1 Implementation details . . . 15

4.2 Setup . . . 16

4.3 Results . . . 17

4.3.1 Number of States . . . 18

4.3.2 Splitmethod . . . 19
4.3.3 Which fact pairs to consider for refinement . . . 20
4.3.4 Initial Abstraction Selection . . . 22
4.3.5 Obtain the heuristic values . . . 24
4.4 Best Configurations . . . 25

5 Comparison with CEGAR for Pattern Databases and Cartesian Abstractions 28
5.1 Projections, Pattern Databases and Cartesian Abstractions . . . 28
5.1.1 Projections and Pattern Database Heuristics . . . 28
5.1.2 Cartesian Abstractions . . . 29
5.2 The competing algorithms . . . 29
5.2.1 Projection . . . 29
5.2.2 Cartesian Abstraction . . . 30
5.2.3 Domain Abstraction . . . 30
5.2.4 Explicit transition systems . . . 30
5.3 Results . . . 31
5.3.1 Conclusion . . . 33

6 Conclusion and Future Work 34

Bibliography 36

Appendix A Appendix 37
A.1 Commands executed in the comparison benchmarks . . . 37

1 Introduction

Planning is a field in Computer Science that deals with finding plans in different problem settings. Consider, for example, the logistics task of delivering packages and letters from their start location to the desired recipients. This problem boils down to finding a sequence of actions, also called a plan, which must be performed so that, in the end, every package is at the correct location. Actions in this domain could be: "load package A into truck X" or "drive truck B to City Y". As every action is associated with costs (e.g. fuel or worker's salary), we are interested in finding a plan that minimises the overall costs. Finding such a cost-optimal plan is the goal of *cost-optimal planning*.

In planning, a state is a set of variable values describing a situation within the problem considered. All states that can exist in a problem form the so-called state space. When state spaces get huge, it is infeasible to find the cheapest plan by brute force. For this matter, a better approach is to use informed search algorithms like A* (Hart et al., 1968). Informed search algorithms use a *heuristic* to guide the search in the most promising directions first. A heuristic function takes a concrete state as input and estimates the cost of the cheapest path from this state to the closest goal state.

There are different ways one can develop a heuristic: one possibility is to use expert knowledge about the specific problem to construct a heuristic manually. As this method can be very time-intensive and, in some instances, not feasible, automated planning deals with how to create heuristics automatically for domain-independent planning.

One approach in automated planning is to construct and apply *abstraction heuristics*. Abstraction heuristics use *abstractions* of the original problem to compute heuristic values. The idea is to estimate solution costs by considering a minor planning task derived from the original problem. To do so, we use an abstraction function that determines which states of the original plan's state space are to be distinguished. The state space induced by this function has fewer states, called abstract states, and all affected transitions are rewired accordingly. As a result, an exhaustive search in the induced abstract state space becomes feasible. Given a state of the original problem, the heuristic value corresponds to the cost

of an optimal solution starting from this state in the abstract state space.

Abstraction heuristics gain much attention in the planning community and show good performance across various planning problems. While abstraction classes as *projections*, that are the abstractions underlying *pattern databases(PDB)* (Culberson and Schaeffer, 1998), *cartesian abstractions* (Seipp and Helmert, 2013) or *merge-and-shrink abstractions* have been thoroughly studied in the planning context, there yet exists little work on domain abstractions (Hernádvölgyi and Holte, 2000). In this thesis, we aim to close this gap by introducing an algorithm that automatically creates suitable domain abstractions for use in abstraction heuristics. The algorithm presented follows the counterexample-guided abstraction refinement(CEGAR) (Clarke et al., 2000) principle, which has also been used in the context of PDB's by Rovner et al. (2019) and with Cartesian Abstractions by Seipp and Helmert (2013). Using CEGAR for generating domain abstraction is motivated by the fact that CEGAR performed well for the two mentioned abstraction classes.
This thesis presents and evaluates an algorithm that generates domain abstractions following the CEGAR principle. After explaining the necessary background in Chapter 2, we present the actual algorithm and its configurations in Chapter 3. After the theoretical comparison of the different algorithm configurations, Chapter 4 will evaluate all versions of our algorithm using a benchmark set consisting of 1827 planning tasks from 65 problem domains. The best performing algorithm for constructing domain abstractions moves on to Chapter 5, where we compare it to related algorithms that also use the CEGAR principle to construct abstractions. These related algorithms and abstraction classes will be formally introduced and compared using the same benchmarks. The final chapter will draw a conclusion and outline future work.

2

Background

Before this thesis's central part, we will introduce the terms and concepts that are needed to understand this contribution. After formally introducing planning tasks and heuristic functions, we explain the concept of abstractions and how they are used in abstraction heuristics. Finally, we wrap up this chapter by shedding light on one specific class of abstractions called *domain abstractions*, which is the class of abstractions we want to construct with the algorithm presented in this thesis.

2.1 Planning Tasks, State spaces and Plans

We consider classical planning in the SAS^+ formalism (Bäckström and Nebel, 1995), which is a declarative description of planning tasks that uses variables with finite non-empty domains. This formalism is also internally used by the planning system Fast Downward (Helmert, 2006), which we used to implement and test the algorithms in this thesis.

Definition 1 (Planning Task) In the SAS^+ formalism a planning task Π is a 4-tuple $\langle V, s_0, G, A \rangle$ that consists of the following components:

- A finite set of *state variables* V. Each variable $v \in V$ is associated with a finite domain D_v. A *fact-pair* $\langle v, d \rangle$ assigns a variable $v \in V$ to a value d of its domain D_v. A *variable assignment* is a set of fact-pairs, at maximum one for each variable. A state is a set of fact-pairs which contains one $\langle v, d \rangle$ over all variables $v \in V$. We further use the notation $s(v) = d$ to denote that the variable v is assigned to d in state s.
- A state s_0 which is called the *initial state*.
- A variable assignment G which denotes the *goal conditions*. A state s is called a *goal state* if $G \subseteq s$.
- A finite set of *actions* A, where each action $a \in A$ is associated with two variable assignments, namely the *effects* $\mathit{eff}(a)$, the *preconditions* $\mathit{pre}(a)$ and a non negative cost $\mathit{cost}(a) \in \mathbb{R}_0^+$. An action a is *applicable* in a state s iff s satisfies the preconditions

of a, i.e. $pre(a) \subseteq s$. The application of an action leads to a successor state s' with $s'(v) = d\ \forall$ fact-pairs $\langle v, d\rangle \in eff(a)$ and $s'(v) = s(v)$ otherwise.

Definition 2 (State Space) The state space or transition system S induced by Π is a 6-tuple $\mathcal{S}(\Pi) = \langle S, A, cost, \mathcal{T}, s_0, S*\rangle$ with the following components:

- A finite set of all possible *states* $S = D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$.
- The finite set of *actions* A.
- The *cost* function $cost : A \to \mathbb{R}_0^+$, as defined in Π.
- A finite set of (state) *transitions* $T \subseteq S \times A \times S$, where each transition $t \in T$ has the form $s \xrightarrow{a} s'$ where $s, s' \in S$, action $a \in A$ applicable in s and s' is the state that is the result of the application of a in s.
- The *initial state* s_0, as defined in Π
- A finite set of *goal states* $S_* = \{s \in S | s \subseteq G\}$ which consists of all states that agree with the goal-variable-assignment G from Π.

A state space can be best imagined as a labelled directed graph where the states correspond to vertices and the transitions correspond to arcs. As the explicit representation of such a graph is often infeasible in terms of size and thus storage requirements, planning formalisms are used as compact declarative descriptions of the planning task. As seen in definition 2, we can derive the explicit transition system from Π.
Given a state space S, a path π is a series of transitions $t_0, \ldots, t_n$ such that $s_i \to_a s_i{+}1 = t_i \in \mathcal{T}$ for all $0 \leq i \leq n$, where the state where t_i ends is the same as the state where t_{i+1} starts. A path π that starts at the initial state s_0 and ends at a goal-state $s \in S_*$ is called a plan or solution. The cost of a plan π is equal the sum of action costs $cost(\pi) = \sum_{\langle s_i, a_i, s'_i\rangle \in \pi} cost(a_i)$. In optimal classical planning the goal is to find a plan that minimises the cost among all possible plans. Such a plan is then also called an *optimal plan* or solution for S .

2.2 Heuristic Functions and Informed Search

Since a planning task Π is a compact description of an equivalent explicit representation of the same task, namely $S(\Pi)$, the problem of finding a plan for Π is equivalent to finding a plan in $S(\Pi)$. As state spaces usually get very large for most interesting planning tasks, blind search algorithms such as "Dijkstra's Algorithm" (Dijkstra, 1959), that exhaustively explore the state space, are not feasible to use.
A common approach for solving large instances of planning tasks is to use informed search algorithms that use additional information to guide the search. The premise here is to expand the most promising states first. One class of informed search algorithms are *heuristic search algorithms*.
Heuristic search algorithms use a heuristic function or heuristic $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$, that maps every state $s \in S$ to a non-negative number or infinity. This number $h(s)$ estimates the cost of π_s, where π_s denotes an optimal plan starting in state s or should be to denote that

there is no path from this state to any goal. In the following, we describe some important concepts related to heuristic functions.
The *perfect heuristic* $h^*(s),\ s \in S$ returns not an estimate, but the *exact cost* of the optimal-plan starting from s, or infinity if no plan exists. The perfect heuristic is often not feasible to compute and thus often a theoretical construct that is used to describe properties we want heuristics to have:
A heuristic h is admissible iff $h(s) \leq h^*(s)$ for all $s \in S$. Such a heuristic never overestimates the exact optimal plan cost. Admissible heuristics are critical in cost-optimal planning as heuristic search algorithms like A* (Hart et al., 1968) need admissible heuristics such that the plans they find are guaranteed to be optimal.
The more informative the heuristic, the better a heuristic search algorithm using it performs. This intuitively makes sense and means that in addition to being admissible, the heuristic values $h(s)$ should be as close to the perfect heuristic values $h^*(s)$.

One approach for the construction of a heuristic is to create it domain-specific. That means that an expert analyses the problem domain and creates a hand-crafted heuristic for the specific problem domain. In classical planning, we are interested in automatically creating good heuristics based on the description of the planning task and regardless of the problem domain. One such approach will be discussed in the following.

2.3 Abstractions

Besides critical paths, landmarks and delete relaxations, current heuristic estimators for domain-independent classical planning are often based on abstractions. A state space abstraction drops distinctions between certain states while preserving the overall state space behaviour as good as possible. An abstraction of a state-space $\mathcal{S}$ is defined by an abstraction function α that determines which states of the original state space $\mathcal{S}$ are distinguished in the abstraction. The abstraction function α maps states from the original state space to states of the abstract state space $\mathcal{S}^\alpha$, also called *abstract states*.

Definition 3 (Induced abstraction) Let S be a state space and let $\alpha : S \rightarrow S^\alpha$ be a surjective function. The abstraction of state space $\mathcal{S}$ that is induced by α, denoted as $\mathcal{S}^\alpha$ is $\mathcal{S}^\alpha = \langle S^\alpha, A, cost, \mathcal{T}^\alpha, s_0^\alpha, S_*^\alpha \rangle$ with:

- $\mathcal{T}^\alpha = \{\langle \alpha(s), a, \alpha(s') \rangle | \langle s, a, s' \rangle \in T\}$
- $s_0^\alpha = \alpha(s_0)$
- $S_*^\alpha = \{\alpha(s) | s \in S_*\}$

The idea of abstraction heuristics is to use solution costs in $\mathcal{S}^\alpha$ as estimates for the concrete solution costs in $\mathcal{S}$. The abstraction-induced state space should have a feasible size to be solved by an exhaustive search algorithm like "Dijkstra's Algorithm" (Dijkstra, 1959). It is crucial to note that every state-space abstraction is a homomorphism, meaning they are structure-preserving. Consequently, the shortest path between two states in S is at least

as large as the shortest path between their corresponding abstract images in S^α. This circumstance makes for the fact that the abstract goal distances used as heuristic values for search in S are admissible as for example proofed by Hernádvölgyi and Holte (1999). This property makes abstraction heuristics reasonable for search algorithms such as A*, which need admissible heuristics to be optimal.

The choice of a good abstraction function α is crucial for the quality of abstraction heuristics. Every α leads to an admissible and consistent heuristic but most lead to low performing ones. Another important aspect is the size of S^α. On the one hand, an abstraction α that induces an abstract state space S^α that is too large makes it infeasible to compute the abstract solution costs. On the other hand, an abstraction that is too small may not be informative enough and lead to a longer runtime of the search algorithm.

2.4 Domain Abstractions

Apart from the well-known abstraction class of projections (Culberson and Schaeffer, 1998), which are the basis of pattern database heuristics, there also exist other abstraction classes, such as cartesian abstractions (Seipp and Helmert, 2013), that also gain attention in the planning community. We will get to know both of these and their application for abstraction heuristics in Chapter 5. For now, we introduce the abstraction class of *domain abstractions* that was originally introduced by Hernádvölgyi and Holte (2000).

Definition 4 (Domain abstraction) Consider a planning task Π with variables $V = \{v_1, \ldots, v_n\}$. A domain abstraction α is induced by equivalence relations $\sim_i$, one for each variable. Each $\sim_i$ denotes the equivalence relation for variable v_i and is defined on its domain D_i. The equivalence classes of $\sim_i$ are called *groups* of the domain D_i. The number of groups in D_i will be denoted as $|\sim_i|$ where $1 \leq |\sim_i| \leq |D_i|$. We give each group a unique number between 0 and $|\sim_i| - 1$. Each $\sim_i$ induces a surjective function $\alpha_i : D_i \to \{1, \ldots, |\sim_i|\}$ that maps the domain of variable v_i to the set of group numbers. The number of elements in domain D_i that are in the same group and share the same group number g, is denoted as size of group g: n^i_g. The abstraction α is defined as $\alpha(s) = \alpha(s')$ iff $\alpha_i(s(v_i)) = \alpha_i(s'(v_i))\ \ \forall i = 1 \ldots n$.

The abstraction induced state space $\mathcal{S}^\alpha$ has states s^α so that for every fact pair $v_i \to x, x \in D_i$ of state s, s^α has $v_i \to \sim_i (x)$.

Here an example given a planning task Π with one variable v_1 and a domain abstraction $\sim$. Domain abstraction $\sim$ has $\sim_1 = \{(0,0),(1,1),(1,0),(0,1),(2,2)\}$:

$$D_1 = \{\underbrace{0,1}_{1}, \underbrace{2}_{2}\}$$

Above, one can depict that the domain abstraction for domain D_1, induces two equivalence classes and hence groups. Giving these groups a number we say, that value 0 and 1 are in group 0, where value 2 is in group 1. Given an original state $s = \langle 1 \rangle)$ the abstract state

$\alpha(s)$ would be $s' = \alpha(s) = \langle 0 \rangle$).

The example makes clear that domain abstractions simplify the original problem by reducing each variable's domain size. Therefore multiple values from an original variable domain D_i are forming groups.

2.5 Counterexample-Guided Abstraction Refinement

In classical planning, we want to obtain good abstractions independent from the problem domain. One algorithm framework for the construction of abstractions is the Counterexample-guided abstraction refinement(CEGAR) principle, which has its roots in model checking for large systems (Clarke et al., 2000). It has emerged in classical planning, where it is leveraged to construct abstractions automatically. Two already existing practical applications are the construction of abstraction heuristics on the basis of projections by Rovner et al. (2019) and Cartesian abstractions by Seipp and Helmert (2013), both explained in further detail in Chapter 5.
The general idea is to start with a coarse abstraction α that is iteratively improved in sequential refinement steps. The goal of CEGAR is to only refine an abstraction in necessary places. In the context of planning, this means that given a task $S(\Pi)$, we first compute a plan that solves the induced abstract task S^α. The obtained plan is then tested to be applicable in the original state space $S(\Pi)$. If not, the information about why the plan is not applicable, called *flaw*, is used to refine the abstraction. The refinement makes sure that the same flaw will not occur again in future iterations. This process is repeated until a solution for the original task is found, or some termination criteria are met.

As CEGAR is more a principle than a detailed algorithm, it can be used to design algorithms that can construct abstractions based on different abstraction classes. In planning, CEGAR does not have to solve the problem completely, e.g. find a plan that is entirely applicable in the concrete task. It instead can be interrupted at any time to return abstractions that yield good heuristics.

The choice of the abstraction class is an essential consideration when using CEGAR. Projections do not allow for fine-grained refinement steps, which is why Pattern Database Heuristics (Culberson and Schaeffer, 1998) such as the previously mentioned ones by Rovner et al. (2019) combine multiple projections. Domain Abstractions allow for a more fine-grained refinement, which is why we focus on constructing one domain abstraction in this thesis.
As CEGAR shows diminishing returns, it is beneficial nonetheless to use CEGAR for the construction of multiple abstractions and combine them admissibly in a canonical heuristic.

3 Constructing Domain Abstractions with CEGAR

Domain abstractions were originally introduced by Hernádvölgyi and Holte (2000). In their experiments, they state that the abstractions are "generated randomly" but give no details about what this means or how their abstractions look. We are also not aware of any other work that generates domain abstractions. To fill this gap, this chapter introduces our attempt for the creation of one single domain abstraction. Chapter 4 will then evaluate the performance of different algorithms-configurations and compare the best of them with related work in Chapter 5.
In this chapter, we introduce an algorithm that, given a planning task Π, constructs one single domain abstraction $\sim$ by leveraging the CEGAR principle explained in section 2. We first show how the general algorithm for constructing one domain abstraction works. Next, we introduce different configurations of the same algorithm and compare them on the theoretical level. Furthermore, we introduce how heuristic values are obtained from the generated domain abstraction.
We leverage the CEGAR principle to construct a single domain abstraction $\sim$ for a given planning task Π. The goal is to construct $\sim$ such that the abstraction heuristic it induces yields accurate heuristic values for the search in $S(\Pi)$. Algorithm 1 depicts the high-level algorithm, which follows the CEGAR principle, to construct one single domain abstraction. Given the concrete planning task Π as input, the algorithm first creates an initial abstraction $\sim$. Based on this initial abstraction created by *getInitialDomainAbstraction*, we retrieve exactly one flaw f by obtaining an optimal plan for the abstraction π and then calling the *findFlaw* function on it. The flaw contains the information on why the solution found in the state space induced by the abstraction is not a solution of the original task.
In the next step, the function *refineAbstraction* uses the flaw and the current abstraction to refine the abstraction based on the flaw. The returned abstraction is the refined abstraction which is then used in the next iteration, which starts again by trying to find a flaw in the now refined abstraction.
This loop continues until one of two situations comes up:

1. No flaw is found by *findFlaw*. This means that the plan that solved the abstract task also solved the original task. This plan gets extracted using the *extractSolution* function and returned, as no further search is needed.

Algorithm 1 General CEGAR algorithm to construct a domain abstraction. Given a planning task returns a plan, proves that no plan exists or returns an abstraction of the task.

```
    Input: Planning Task Π
    Output: Domain-Abstraction ∼
  ∼ ← getInitialDomainAbstraction(Π)
  while ¬(terminate()) do
      π ← findOptimalAbstractPlan(∼)
      if t is "no trace" then
          return task is unsolvable
      end if
      f ← findFlaw(∼, t)
      if f is "no flaw" then
          return extractSolution(π)
      end if
      ∼← refineAbstraction(f, ∼)
  end while
  return ∼
```

2. Other criteria for termination are met. These conditions are checked inside the *terminate* function, which returns a boolean flag, indicating whether another refinement iteration will be started. Some examples for terminating include time limits or a maximum number of states in the abstraction-induced state space.

In both cases mentioned, the algorithm will break out of the loop and return the refined final abstraction $\sim$. After the construction of $\sim$, the abstraction will be used to obtain heuristic values that are then used for search. We discuss different ways to obtain heuristic values from $\sim$ in Section 3.4.
In the following sections, we discuss the building blocks of this framework in more detail. Section 3.1 discusses how to initialise the abstraction. Section 3.2 describes how flaws are obtained based on the current abstraction. Finally, Section 3.3 will deliberate how the abstraction is refined based on a flaw.

3.1 Initial Abstraction Selection

Our algorithm starts with creating an initial abstraction via the *getInitialDomainAbstraction* function. During the work on this thesis, multiple ways of creating initial abstractions were tested. The tested approaches include 1. the separation of all goal facts in different groups, and 2. starting with an abstraction that does not distinguish any state (i.e. $|\sim_i| = 1$ for all $i \in \{1, \ldots, n\}$) and is thus the most trivial abstraction possible. The state-space induced by the latter abstraction has just one state, which is the initial and goal state simultaneously. Finally, we decided to implement both approaches in our algorithm to be able to evaluate the effects of both. In the following we list some reasons for that we think starting with a coarse abstraction the favourable option:

1. The main reason for our choice is that the split of all goal facts might not be necessary to get an abstraction with no flaw. For example, this happens when two goalconditions

are symmetrical: Achieving one of the goal conditions always achieves the other one as well. In contrast to splitting off goal facts, starting from the one-state abstraction does not introduce any bias to the abstraction or its refinement. It allows for the refinement to make choices independently.

2. Consider the first iteration of CEGAR, where the optimal trace is the empty trace which is not a plan for the original task: The flaw, in this case, contains all goal facts that were not achieved. These will be split anyway.

Another point worth mentioning is that the creation of the initial abstraction is a tradeoff between time and memory: On one hand, splitting goal facts might be faster in many problems and avoids the effort to do so within the CEGAR loop. On the other hand, this approach might require more memory as the abstraction might get ample from the beginning. In the end, the performance strongly depends on the run's time and state limits, as discussed in Chapter 4.

3.2 Retrieve Flaws

The choices the refinement makes are purely based on flaws. From Chapter 2 we already know what flaws are. In the following section we will explain how our algorithm retrieves flaws that are then later used to refine the abstraction.
The first step in an iteration of the refinement loop is to use the current abstraction $\sim$ and the given original task Π to retrieve a flaw or find that no flaw exists. Therefore we first discuss how the functions $findOptimalAbstractPlan$ and $findFlaw$ work, discover the formal definition of flaws and which different types are differentiated. All algorithm configurations described in this thesis share the same behaviour regarding these two functions.

3.2.1 Find a plan in the abstract state space

Given the current abstraction, $\sim$ and the planning task Π the function *findOptimalAbstractPlan* performs an exhaustive blind search, Uniform Cost Search to be exact, in the state space induced by the abstraction to find a path which is an optimal plan for it. It makes sense to mention that because a blind search is used, abstractions that get too large can lead to a very long runtime of the refinement loop. More on that in Chapter 4.
The return value of *findOptimalAbstractPlan* can differ based on whether the abstract state space is solvable. When the abstraction-induced state space is unsolvable, the original state space is neither. The reason is that every plan in the original plan is also a plan in the abstraction, as described in Section 2.3. In that case, the search algorithm will fail to find a plan and return "no trace". This will lead the main algorithm to return "task is unsolvable". When there exists a solution, the exhaustive blind search will find it and return the according plan. This plan might be empty when the initial abstract state is already a goal state of the abstraction. For example, this is always the case for the optimal abstract plan for the most coarse abstraction, described in Section 3.1.

3.2.2 Find a flaw

When a plan has been found for the abstract state space, it is passed on to the *findFlaw* function. This function takes the plan $\pi = \langle a_0, \ldots, a_n \rangle$ as well as the original task Π and iterates over π. Starting at the initial state $s_0 \in S(\Pi)$, the function first tries to apply the first action a_0 by checking whether it is applicable in the initial state: $pre(a_0) \subseteq s_0$. If applicable, action a_0 is applied and leads to a successor state. Following the same scheme, each action a_i is checked to be applicable in s_i and gets applied in that case.
When an action a_i is not applicable in s_i, we call this a *precondition flaw*. The flaw f contains the missed facts, e.g. the variable assignments that state s missed so that a would be applicable in a state s. Hence, $f = pre(a) \setminus \{s\}$.
When all actions $a \in \pi$ have been successfully applied, we check whether the state s where we ended up is a goal state. When this is not the case, we have a *goal flaw*. That means that the state s we ended up with is not a goal state since $s \supseteq G$. The goal flaw f contains the variable assignments of s that do not match the ones that would have been needed for s to be a goal state: $f = G \setminus \{s\}$.
When no flaw could have been found, the optimal plan π for the abstraction induced state space, found by *findOptimalAbstractPlan* is also a plan of the state space/task. In that case, "no flaw" is returned, and the main algorithm will extract the plan and return it.

3.3 Refinement of the Abstraction

If the task is solvable and a flaw has been found, the goal is to improve the current abstraction based on the found flaw f and the current abstraction $\sim$, where the goal is that the same flaw will not occur again once the abstraction is refined. In our algorithmic framework, the routine *refineAbstraction* will take care of that. As previously mentioned, we present different options for refining the abstraction. The general approach considers the flaw f and the current domain abstraction $\sim$ to process and return a new, refined domain abstraction. There are two parts where the refinement, given a flaw f and a domain abstraction $\sim$ differs:

1. Which and how many fact pairs of the flaw are used for refinement. At minimum one fact pair must be used. Here we differentiate between *all-assignment split* and *single assignment split*.

2. How the refinement based on one assignment $v_i \rightarrow x_i \in f$ works. Here we differentiate *SingleValueSplit* and *UniformRandomSplit*

In the following, we will explain the two categories and their options in detail.
Recall that a flaw, whether precondition or goal flaw, is a partial variable assignment that disagrees with the state s in that the flaw occurred. Because we consider planning tasks in the SAS+ formalism, each flaw can contain a maximum of one missed assignment per variable.
When refining the abstraction, our goal is to ensure that the same flaw cannot occur again. This means we need to move at least one of the variable assignments $v_i \rightarrow x_i \in f$ in another equivalence class. In our algorithm, we will always move the assignments into a new

equivalence class, never one that already exists. This means we continually increase the domain of α_i by one for every refinement step on variable v_i. Nevertheless, the "regrouping" of the domain values in existing equivalence classes is an exciting idea for future research. Thus, regarding $\sim_i$, $s(v_i)$ will no longer be in the same equivalence class as $v_i \rightarrow x_i \in f$. Consequently, action a will no longer be applicable in the abstract state that corresponds to s. Thus the same flaw will not happen again. Since we now know the split operation's requirement, we present different strategies on how to choose the facts to split. The resulting domain abstraction will have changed local equivalence relations $\sim_i$ for every fact pair in the flaw we choose to split along.

3.3.1 Splitmethod

In this section, we discuss the different approaches to splitting along one fact pair of the flaw f. There exist two approaches: The *Single Value Split* as well as the *Uniform Random Split*. How to choose the facts we want to split along will be discussed in Subsection 3.3.2.

3.3.1.1 Single Value Split

Given one $v_i \rightarrow x_i \in f$ and the current domain abstraction $\sim$ the *Single Value Split* technique does the following: in the old abstraction $\sim_i$, x_i has equivalence class number n_i and the total number of equivalence classes in $| \sim_i | = k$. Now we introduce a new equivalence class for x_i with number $k + 1$, where only x_i is included. Formally that means $\sim_i^{new} = (\sim_i \cup (x_i, x_i)) \setminus \{(v, w) | (v, w) \in \sim_i, v = x_i \vee w = x_i\}$. Note that the old equivalence class k, x_i has been in, will not be empty after this operation. The reason is that, when equivalence class k would have had just x_i included, $v_i \rightarrow x_i \notin f$. The abstract plan could only apply actions that are applicable if $v_i \rightarrow x_i$, hence this cannot be a flaw.

3.3.1.2 Uniform Random Split

Given $v_i \rightarrow x_i \in f$ and the current domain abstraction $\sim$ the *Uniform Random Split* technique is doing the following: In the old abstraction $\sim_i$, x_i has equivalence class number v, so $\alpha_i(x_i) = v$, and the total number of equivalence classes is $| \sim_i | = k$. Just as in the single value split we introduce a new equivalence class, which has number $k + 1$. What we do now additionally is randomly choosing $(n_v^i - 1)/2$ other values of the equivalence class k and also move them into the new equivalence class $k+1$. Here $(n_v^i - 1)$ denotes the number of values in equivalence class v after removing x_i from it.

3.3.2 Which fact pairs to consider for refinement

The second category where the algorithms can be configured differently is the choice of which fact pairs $v_i \rightarrow x_i \in f$ are used for refinement. In our algorithm, one can choose between using all $v_i \rightarrow x_i \in f$ or select one of them. For the selection of one fact pair, one of three strategies can be chosen.

3.3.2.1 Split along one flaw assignment

Splitting along just one $v_i \rightarrow x_i \in f$ allows for more controlled refinement of the abstraction, as every refinement is necessary to ensure that the same flaw will not occur again in future iterations. When a flaw assigns multiple variables and thus contains multiple fact pairs, we must pick one. This is where we can apply one of three strategies:

Random pick The first and most straightforward strategy is to choose one variable assignment of the flaw uniformly at random.

Pick least refined domain This method involves the rating of each $v_i \rightarrow x_i \in f$ according to how refined the domain of the according variable v_i is at the moment. Based on the old abstraction $\sim$, we choose the local equivalence relation $\sim_i$, which is the least refined so far, e.g. has the minimal number of equivalence classes. Formally we choose the domain of variable k so that $argmin_i | \sim_i | = k$. When multiple variable assignments have the same rating, we pick one uniformly at random.

Pick maximal refined domain As in the previous approach, each $v_i \rightarrow x_i \in f$ gets rated. This time we rate each fast pair according to how many new abstract states the refinement of $\sim_i$ would introduce. Based on the old abstraction $\sim$, we choose the local equivalence relation $\sim_i$ for that the refinement would introduce a minimum new number of new abstract states, which is equivalent to choosing $\sim_i$ that has the maximum number of groups and thus has been refined most often. Formally we choose k so that $argmax_i | \sim_i | = k$. Note that the size of a domain plays no role but rather the number of groups. When multiple variable assignments have the same rating, one is uniformly random chosen out of it.

3.3.2.2 Split along all flaw assignments

If we split along all $v_i \rightarrow x_i \in f$, this means we apply the procedure explained in Section 3.3.1 to all $\sim_i$ where $v_i \rightarrow x_i \in f$. Splitting all fact pairs is strictly not necessary for correctness but is an intuitive technique which proves to be rather successful, as we will see in the next chapter.

When splitting all facts, the resulting domain abstraction can grow significantly in size. When a maximum number of states is given, this can lead to missed refinement potential, as in this case, the refinement is aborted, and the old abstraction is returned. This is the same problem that arises when splitting all goal facts as described in Section 3.1. Furthermore, we risk increasing the abstraction size significantly while having the risk of making decisions that are just side effects of other flaws.

To mitigate this problem, we check whether the abstraction would get too many states after refinement. When it does, we switch to the single flaw refinement described above and check whether the abstraction gets too big this time.

3.4 Obtain the heuristic values

Once the algorithm for the construction of one domain abstraction is finished and returned the final domain abstraction $\sim$, the goal is to obtain heuristic values $h(s)$ for all $s \in S$ where S is the set of all states in $\mathcal{S}(\Pi)$. As we have just one domain abstraction, there is just one heuristic function based on the abstraction $\sim$ we must consider.
Given a state $s \in S$, we first get the according abstract state for s, denoted as $\alpha(s)$. The heuristic value that is returned by an abstraction heuristic corresponds to $cost(\pi)$ where π is an optimal plan starting from $\alpha(s)$ in the abstraction induced state space S^α. There are two ways to obtain the abstract goal distances:

3.4.1 Precalculation

The first and, at the same time, most common approach to obtain heuristic values is the precomputation of all abstract goal distances. For every abstract state of the abstraction induced state space S^α, the cost of the optimal plan to the closest goal state is computed and stored in a lookup table in memory. This happens before the actual search starts. This is done via a backward search using "Dijkstra's Algorithm" (Dijkstra, 1959) starting from all goal states of the abstraction. The result is a lookup table containing the optimal path's cost to the closest abstract goal state for every abstract state, or ∞ if no goal state is reachable from the given abstract state.

3.4.2 On demand computation

Or also called on the fly is the second approach and needs no precomputation but obtains abstract goal distances on the fly. Every time we need $cost(\pi)$ for an abstract state s_α we have not yet calculated, we start a forward search using Uniform Cost Search in the abstract state space starting from s_α. The result is then stored in a lookup table to prevent double calculation of the same value, which is unnecessary.
To make the access of the lookup table efficient, an index for every abstract state is needed. In fact, the index is also used throughout the implementation; everywhere, identifying abstract states is necessary. For example, the duplicate checking in the blind search algorithms used by CEGAR and the precomputation need to identify the states that have already been expanded.
For identifying abstract states in a domain abstraction, we introduce a perfect hash function that returns a unique hash-value for every abstract state. This hash-function, inspired by the perfect hash function used for PDB heuristics, will be shown in the next section.

4

Benchmarks and Evaluation

Before we compare our algorithms to other abstraction heuristics, this section compares different configurations of our algorithm. After we mention some implementation details that are worth noting, we explain the benchmark setup in Section 4.2 and follow up with the presentation and evaluation of the benchmark results in Section 4.3. All algorithms in this section were implemented and benchmarked in the domain-independent classical planning system Fast Downward (Helmert, 2006).

4.1 Implementation details

A perfect hash function is a hash function that maps distinct elements of some set X to a set of integers with no collisions. In our implementation an efficient way to identify abstract states is vital and needed in different places e.g. the data-structures used for search in the abstraction or for efficiently accessing the lookup-table of heuristic values. We introduce a perfect hash function for domain abstractions that is inspired by perfect hash functions for pattern databases.

For domain abstractions, our perfect hash function maps each abstract state s_α of an abstract state space $\mathcal{S}^\alpha$ to an integer in the range from 0 to $|S^\alpha|$, which denotes the number of states in $\mathcal{S}^\alpha$.

As defined in Chapter 2, an abstract state s_α of an abstract state space $\mathcal{S}^\alpha$, that is induced by a domain abstraction $\sim$ is represented as a set of fact pairs that map each variable v_i to a group of $\sim_i$, denoted as g . This number is also denoted as abstract variable assignment or $s_\alpha(i) = g_i$ where $g_i \in \{0, \dots | \sim_i | - 1\}$.

Given an abstract state s_α, the hash is calculated in the following way:

- Let $\sim = \langle \sim_1, \dots, \sim_k \rangle$ be a domain abstraction for a planning task with k variables one for each variable.
- for each $i = 1 \dots k$ we pre-compute $N_i = \prod_{j=1}^{i-1} | \sim_j |$.

$$hash(s_\alpha) = \sum_{i=1}^{k} N_i s_\alpha(i)$$

This operation is very fast and can be completed in $O(k)$, where V is the number of variables in the planning task.
Another specificity of the Implementation we want to mention is the handling of situations, when the refinement algorithm does find a a plan for the original task. In such a case our algorithm depicted in Algorithm 1 aborts the refinement and returns the plan. In our implementation, we do not abort the refinement but log when a solution was found during refinement, as the logging is sufficient for performance comparison. Because of the abstraction size limit we explain in Section 4.3 such a scenario is unlikely to happen and thus not relevant for our analysis.
The last specificity of our algorithm comes to play when it uses all fact pairs of a flaw for refinement. Normally the algorithm aborts the refinement when an abstraction is too large after a refinement step and uses the previous abstraction. When using all fact pairs of a flaw for refinement, the algorithm instead tries to split using one fact-pair before it aborts. This makes sure that the size limit is maximum exploited.

4.2 Setup

To get a meaningful result for algorithm performance, we use a set of 1827 tasks from 65 different domains of International Planning Competitions(1998-2018). For each run, the algorithms have an overall time limit of 30 minutes for abstraction construction and search. Furthermore, we set a memory limit of 2 GB. For the comparison of our algorithms we are mostly interested in the following criteria, that we can extract from our experiments:

Coverage number of tasks solved by the algorithm. This is a good metric for the overall performance of the algorithm. We want algorithm to solve as many tasks as possible.

Total time time required to solve a task(heuristic construction and search). The less time an algorithm needs to solve the tasks, the better. This is also a very good metric for the overall performance of the algorithm that considers the balance of search- and heuristic construction.

Initial heuristic value the initial heuristic value $h(s_0)$. It gives us an idea of how good an abstraction images the original task. Usually we want this value to be as high as possible, since for abstraction heuristics it holds that $h(s_0) \leq h*(s_0)$.

Expansions until the last jump the number of expansions until the last layer of an A* search. This gives us an idea of how good the heuristic guides the search algorithm. The fewer expansions the more informative is the heuristic.

Search time time required to solve a task(just search)

Precomputation time the time required to pre-compute the heuristic values for the abstraction. For algorithms that obtain the heuristic values on the fly this value will

be zero. From this metric we can see when the precomputation of heuristic values is feasible or rather an obstacle.

As we will see in our experiments, these metrics can show a high pairwise correlation. For example expansions until last jump and search time are usually strongly coupled.

4.3 Results

Our analysis has two goals: first, we want to get an overview how each adjustable parameter of our algorithm tends to influence the performance. We do a parameter-wise evaluation as the number of adjustable parameters in our algorithm would lead to a number of overall configurations infeasible to compare to each other. This is also the reason why results for each parameter do not include the full range of all possible configurations. Instead, the data that a benchmark plot is based on consists of data from algorithm configurations that we think are representative. Hence there could exist a configuration that performs very good, but we did not find it. Of course the configurations used are explained in necessary places. Generally, our analysis considers a large number of experiments, which makes us confident that our results give a good direction of which configurations work well. Secondly, we will consider the two algorithm configurations that performed best in our experiments and compare them in detail in Section 4.4. This completes the more general overview over the parameters by showing two configurations performing well.
Finally,, it is crucial to note that the benchmark results for "total time", "expansions until the last jump", and "initial heuristic value" in contrast to "coverage" are just calculated based on the planning tasks that all of the algorithm configurations could solve. Possible implications of that are explained in the subsections when needed. Anyway, this restriction does not exist for the pairwise comparison of two configurations for ex shown in figure 4.4 as each data-point shown represents one task.
Recall the strategies for different parameter choices explained in more detail in Chapter 3:

Domain Abstraction size limit : once the domain abstraction we construct reaches this limit, the algorithm will abort the abstraction construction. The limit is given in terms of the number of states in the abstraction-induced State Space. It is important to note that the size limit is not automatically the size of the abstraction in the end. It is rather a good proxy for it when the step size of size limit values is chosen appropriately.

How is one Fact splitted : This option determines how the domain of a variable gets refined based on a fact pair. Namely we can choose between *Uniform Random Split* and *Single Value Split*.

How many Facts are split : Given a flaw, we either choose to split according to just one fact pair of the flaw or to all of them. When choosing to split just one fact, there are three different methods. As the method we refer to as *minStatesGain* performs best, we used this one for all benchmarks where applicable.

Initial Abstraction selection : determines the domain abstraction that the refinement starts with. This corresponds to the domain abstraction returned by the method

getInitialDomainAbstraction. We choose the most coarse abstraction possible or split as many goal facts as possible.

How are the heuristic values obtained : whether the abstract goal distances and thus the heuristic values are calculated on-demand or are pre-computed in an exhaustive backward search.

The following sections will consider each of the adjustable parameters one by one to compare the performance influence of each regarding the metrics described in Section 4.2.

4.3.1 Number of States

Early experiments during the work on this thesis showed that implementing a time limit of even a few seconds, yielded poor results with coverage of 300 up to 400. As it turned out the reason was that even within this short time, the refinement loop generated an abstraction, that was so fine, that obtaining the heuristic values took a very long time. This is why we decided to implement a size limit for abstractions in terms of maximum abstract states a range of 256 - 10000. Between a limit of 256 and 2048, the values we tested are $256 = 2^8, 2^9, \dots 2^{11}$ and after that $3000, 4000, \dots 10000$. This enables us to have a more detailed insight in the lower range, which is important as in this range the performance of our abstraction heuristics varies the most. The following plot shows the coverage of different algorithm configurations for a range of maximal states between 256 and 16000. For each number of maximal states we included the coverage of many different configurations to make the impression more general.

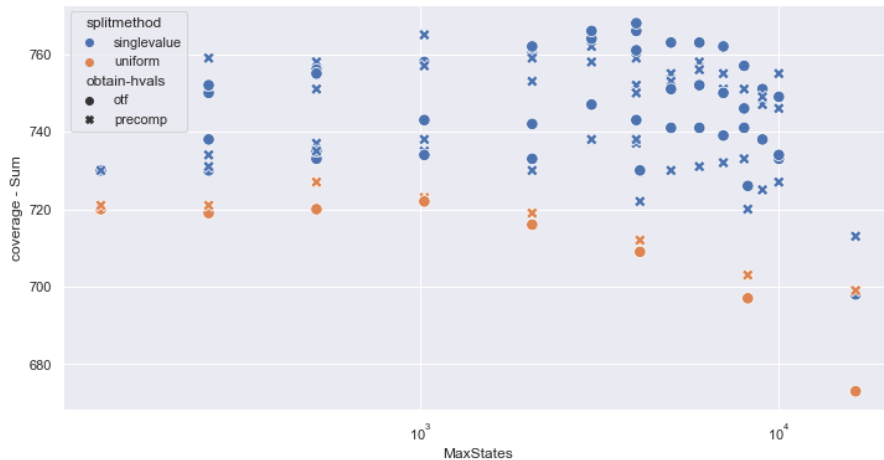


Figure 4.1: Coverage of different algorithms by varying abstraction size limits.

From Figure 4.1 we depict that the highest coverage is achieved with between 1000 - 6000 states where 1024 and 4000 size limit yield the best performing configurations overall. The figure also shows that different other parameter choices for example which split-method we use, yield its best results at different abstraction size limits. Nevertheless it is clear that the best coverage for all configurations can be achieved when choosing a limit between 1000 - 6000 states. In the following we explain why this is the case by looking at the expansions

until last jump and the total time of two algorithm-configurations that are equal except for the size limit:

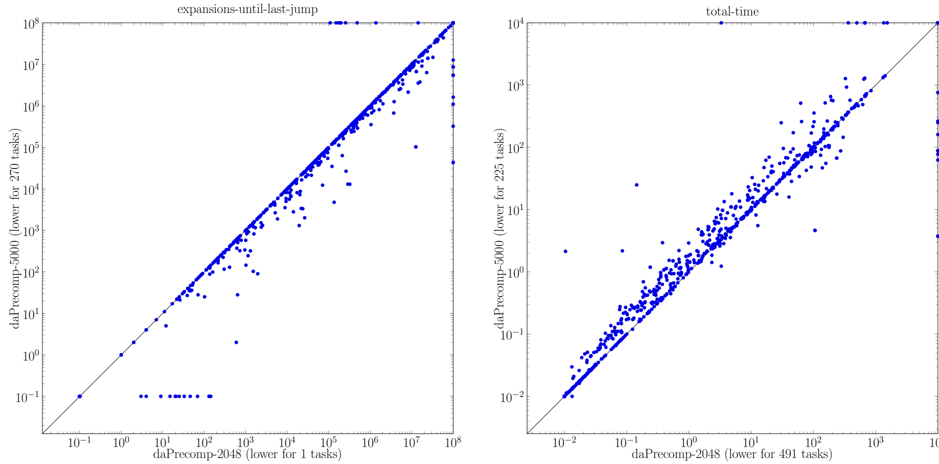


Figure 4.2: Comparison by Left: expansions until last jump, Right: total time

Consider the plots in Figure 4.2. They show that with increasing abstraction size, the total time needed to solve the tasks increases, while the informativeness of the heuristic increases with abstraction size. This indicates a trade-off between informativeness of an abstraction and the time needed to solve a task. Our analysis showed that the refinement loop for the abstraction sizes we use is very fast, thus not the limiting factor in terms of computational effort. Rather, the main time of solving a task is spent at obtaining the heuristic values. On the one hand larger abstractions make exhaustive search more time intensive the time of obtaining heuristic values increases with abstraction size. On the other hand a larger and thus more refined abstraction makes the heuristic more informative, which leads to less expansions and thus a shorter search. Our experiments show that a good balance regarding this trade-off is reached in a limit between 1000 - 6000 states, which shows best performance for this parameter.

4.3.2 Splitmethod

In this subsection, we compare algorithm configurations by the choices one can make on how to split the domain of a variable based on a fact pair $v \to x \in f$. How both approaches work is described in Section 3.3.1.

Recall the information shown in figure 4.1. Now the distinction of the configurations becomes important: orange coloured dots denote algorithm configurations that use *Uniform Random Split*, where the blue dots show configurations using the *Single Value Split* method. As we can clearly see, the coverage of the algorithms using *Uniform Random Split*, is constantly lower across the whole range of size limits. This difference is also visible in terms of time and expansions:

Consider figure 4.3.2 where we compare two configurations that have the same configuration except from the split-method they use. It becomes apparent, that in most tasks the configuration using *Single Value Split* needs less time and less expansions. We can derive that

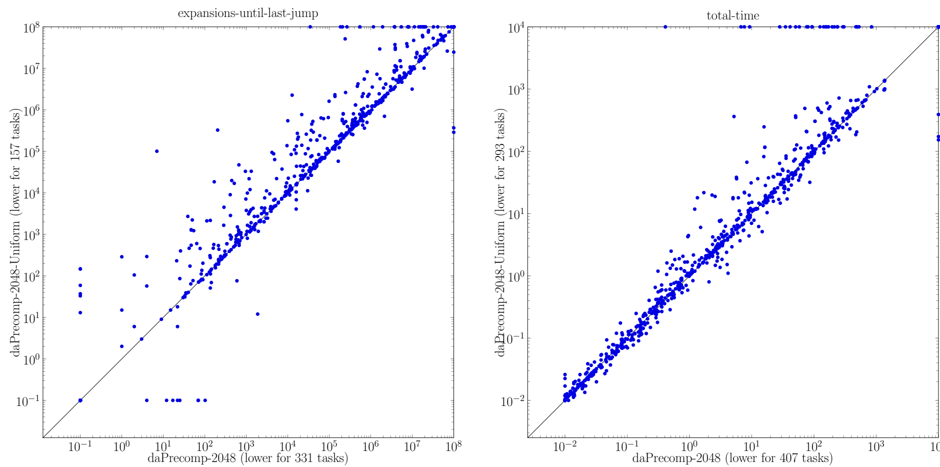


Figure 4.3: Comparison of split methods by Left: expansions until last jump, Right: total time

overall, configurations using *Single Value Split* are superior in terms of time and informativeness of the heuristic. This observation supports the data we have about the coverage of both methods. One reason for the better performance of using *Single Value Split* is that it just modifies the affected variable domain of v in necessary places, while *Uniform Random Split* randomly modifies it in other places which seems to be not a good commitment to make. To be clear, there are problem domains where *Uniform Random Split* performs better, either due to the randomness or the problem structure, but as we are interested in the overall behaviour of the algorithms for this parameter, we can conclude that *Single Value Split* tends to perform much better. In all of our experiments, we just had one configuration using *Uniform Random Split* that performed better than any of the configurations using *Single Value Split*.

4.3.3 Which fact pairs to consider for refinement

This subsection describes how the choice of parameter 3 that determines how many facts of a flaw are used in refining the domain abstraction influences the performance. The first option is to choose one fact-pair of a flaw and the second to choose all fact-pairs. When selecting the first option, one can further decide how to pick the one fact-pair when there are more than one.

We distinguish *maxRefinedDomain*, *minRefinedDomain* and *random*. The best-performing method will be compared to the configuration where all fact-pairs are used for refinement. The analysis of figures 4.3.3 and 4.3.3 tells us, that the method *maxRefinedDomain* performs best regarding coverage in most tasks. The same result when looking at expansions until last jump and the time. One reason for this could be that by refining a domain that has already the highest number of equivalence classes, the number of abstract states does not increase that much. This leads to more refinement steps and thus a more informative heuristic.

Next we compare the *maxRefinedDomain* method that picks one fact to split with configurations that split all facts. Please note that the following figures do not include the experimental data from algorithms that use *Uniform Random Split* but otherwise a bunch

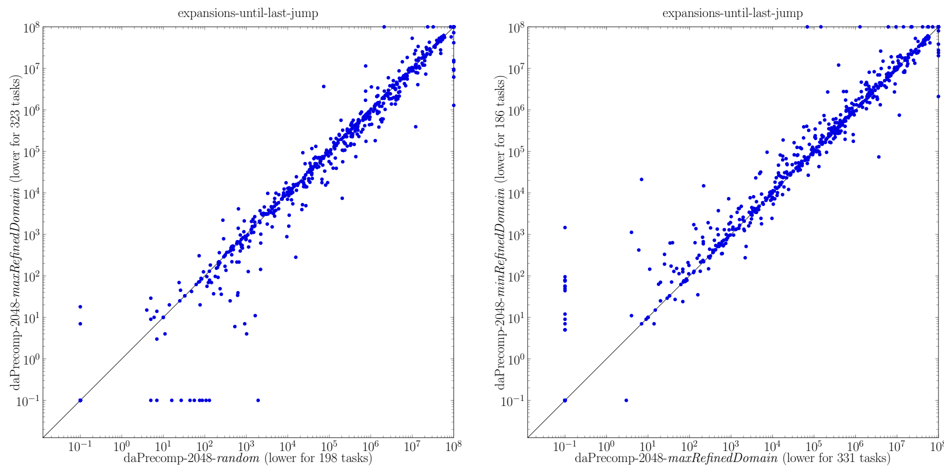


Figure 4.4: Compare *maxRefinedDomain* with *random*(left) and *minRefinedDomain*(right) by expansions until last jump

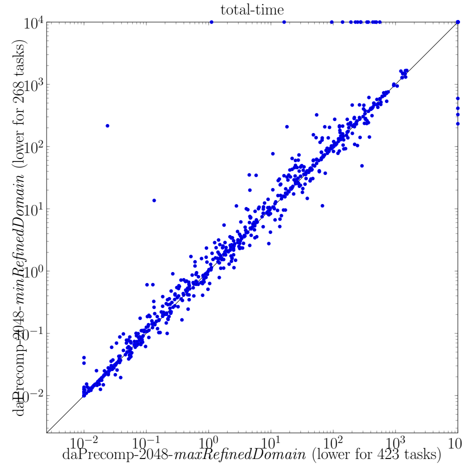


Figure 4.5: Compare *maxRefinedDomain* with *minRefinedDomain*(right) by total time

of configurations for each abstract size limit. Figure 4.6 shows the coverage of a lot of different algorithm configurations. Algorithm configurations that split just one fact are denoted in orange, whereas algorithms that split all facts are painted in blue.

It is clear to see, that all almost configurations we tested in our experiments perform better by a coverage of about 20, when they split all facts. Figure 4.7 shows correlating results, as algorithms that spit all facts consistently show less overall expansions, which means that the heuristic quality is higher. We can conclude that fixing all facts in CEGAR for domain abstractions seems like a good idea and generally leads to a more informative heuristic that yields better coverage. This is also supported by the comparison of expansions and time of two algorithm configurations, where one of them splits all facts, while the other uses the *maxRefinedDomain* method, as denoted in figure 4.3.3.

One possible reason why splitting all facts is performing better, might be that the assignments in a precondition or goal flaw are semantically connected as they were missing all together when trying to apply and action in the abstract state space. Also the refinement

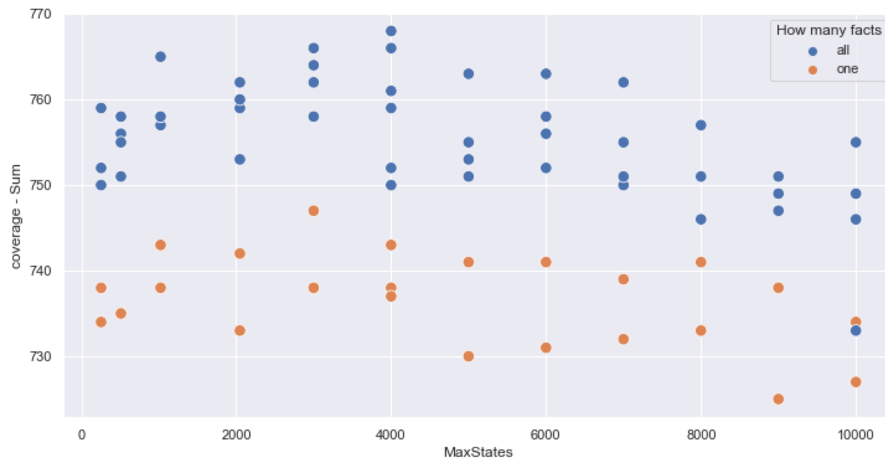


Figure 4.6: Coverage of algorithms splitting one fact vs splitting all facts

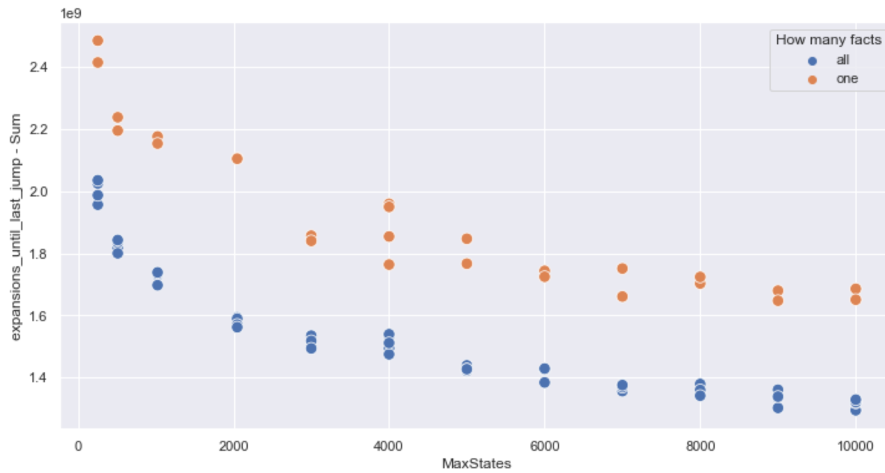


Figure 4.7: Expansions until last jump of algorithms categorised by how many facts are splitted and how heuristic values are obtained

tends to refine the abstraction more in one iteration. It is more efficient to split all fact pairs instead of one since the effort to find an optimal plan in the abstraction needs to be done just once to refine more domains instead of just one. To sum up, splitting all facts of a flaw yields tends to have better overall performance than splitting just one fact.

4.3.4 Initial Abstraction Selection

Next, we explore the effect of the initial abstraction choice on the performance. Here the algorithm allows the choice between the most coarse abstraction and an abstraction that has as many goal facts as possible split.

We start with a view of the coverage data, depicted in Figure 4.9.This figure does not contain any data of configurations that use *Uniform Random Split* or split just one fact. Up to a maximum abstraction size of 2000 states, algorithms that use an initial abstraction with goals split are superior. After that, most configurations that use a coarse initial abstraction

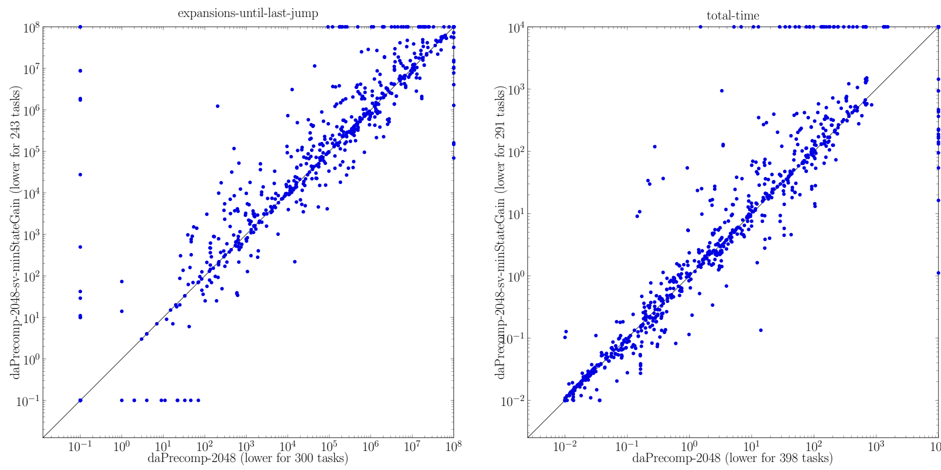


Figure 4.8: Comparison with 2048 state limit by Left: expansions until last jump, Right: total time

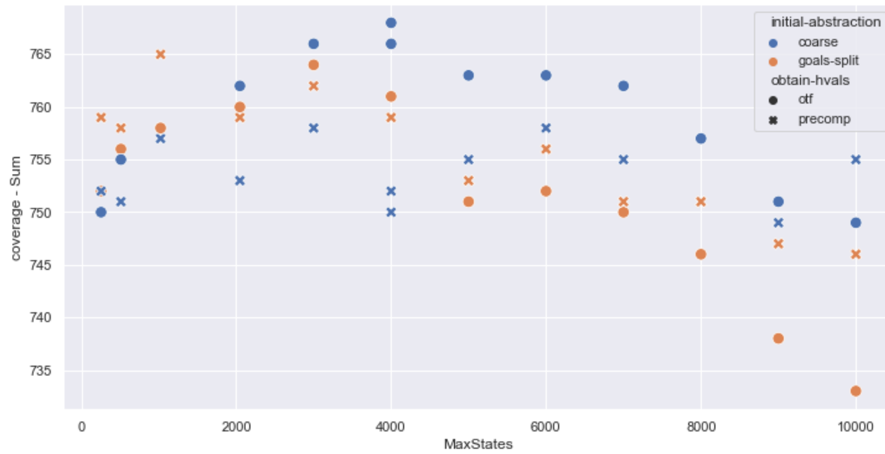


Figure 4.9: Coverage of algorithms categorised by what initial abstraction is used

have better coverage. This observation suggests the conclusion that splitting goals initially especially makes sense when having harsh space limits. In contrast, the decision to split goal facts initially loses its advantage when abstractions can get larger. We think that in this case, the configurations that did not split goals initially have more freedom where to refine the abstraction. In contrast, the other configurations that split goal facts initially will reach the space limit faster and probably miss essential refinement steps concerning action preconditions rather than the goal. When just focusing on configurations that precompute heuristic values, we observe that configurations with an abstraction size limit up to 5000 states have the highest coverage when using the initial abstraction with goal facts split.

The observation regarding coverage correlates with the geometric mean of total time (Figure 4.10). We depict that configurations, where the other parameters are fixed and use the initial abstraction with as many goals as possible split need more time to solve the tasks. Our experiments also showed that the expansions until the last jump are usually a little lower for abstractions that do an initial goal split, but very close. Combined with the

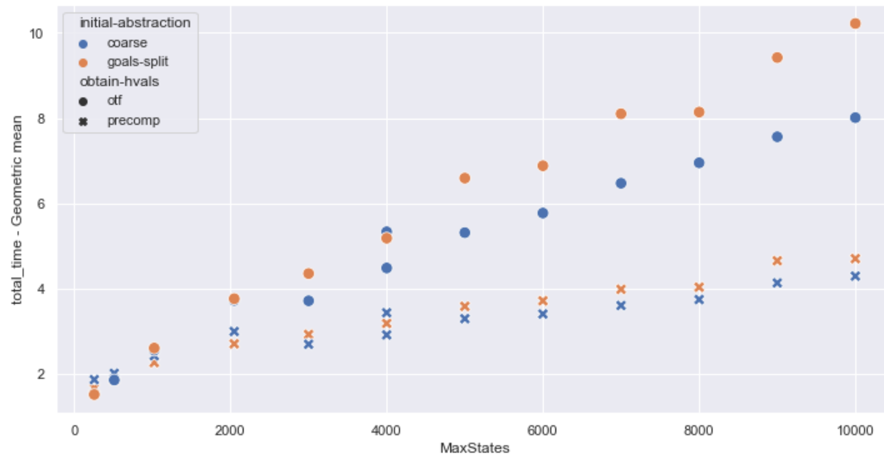


Figure 4.10: Total time of algorithms categorised by what initial abstraction is used

observation that the search time is lower for coarse abstraction configurations, we can derive that obtaining the heuristic values takes longer for configurations that use initial goal split abstractions. This suggests a trade-off between the accuracy of the heuristic and the time to get the heuristic values. Higher heuristic values suggest longer paths in the abstraction. This lead to a higher runtime of exhaustive search algorithms operating in that abstraction as they must consider more search nodes. Another reason could be that splitting goal facts builds an abstraction with longer paths but misses information in the important places.
Overall we can see that algorithm configurations that use the most coarse abstraction tend to have a better performance in terms of time and coverage. While this is the case especially for abstraction size limits of 4000 or more, the initial goal split seems to work well for lower size limits.

4.3.5 Obtain the heuristic values

The last parameter one can adjust is how to obtain heuristic values after the construction of the abstraction is finished. This has less to do with the actual algorithm for the construction of domain abstractions. Nevertheless different configurations seem to yield differences in performance regarding this parameter.
Here one can choose between the precomputation of heuristic values or to obtain them on demand. In practice algorithms that use abstraction heuristics such as pattern databases usually precompute the heuristic values as it has shown to be more efficient.
Regarding the coverage, we can see that, most of the best performing configurations when we take the coverage as measure, obtain their heuristic values on demand. Our guess is, that this has something to do with the previously mentioned trade-off between accuracy and the time needed to obtain heuristic values. This is supported by the observation that precomputation methods with a low number of states perform very good which indicates that the precomputation of the heuristic values seems to be the bottleneck for higher size limits. Further looking at the configurations that use initial goal split we see, that the performance of configurations using precomputation is much more stable around a coverage of 760.

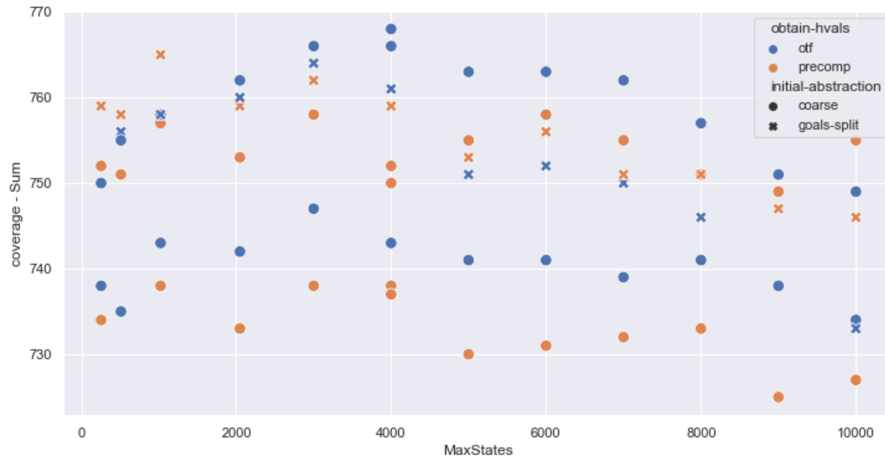


Figure 4.11: Coverage of algorithms categorised by how heuristic values are obtained

Overall we see that the trade-off for configurations obtaining heuristic values on demand, is perfect around a size limit of 4000 states whereas the performance of the precomputation configurations is best for around 1000 states and tends to decrease afterwards.
Taking the total time into account (Figure 4.10), we can see that the time for obtaining heuristic values on demand grows much faster. Regarding the coverage in the higher size limit range, these results are somewhat counter-intuitive as the time is generally lower for the configurations that use precomputation. Our best guess is that the precomputation of heuristic values took very long for the most challenging instances, such that the according algorithms did not get the chance to run the search for long. Hence they have less chance to find a solution.
Another reason for this behaviour is that the implementation of the precomputation is not efficient enough, so having all h-values during a search does not compensate the early start advantage of the on-demand computation.
Since the method how we obtain heuristic values does not affect the quality of a heuristic we omit the according comparison in this section.
In summary, even though configurations that obtain heuristic values on demand tend to better when it comes to coverage, we expect this to change once the precomputation is implemented more efficiently, regarding the much better total time needed. Nevertheless configurations in the lower abstraction limit range are also well performing and definitely the better choice when tasks need to be solved fast.

4.4 Best Configurations

After we have evaluated the performance of our algorithm according to each each parameter separately, we now want to show the two best performing configurations, namely DA^{OTF} and $DA^{precomp}$. The configuration of both can be depicted from table **??**.
As we can see, these are two mostly different configurations. The reason why they are so different is, that configurations that pre-compute their heuristic values seem to work best when goals are initially split. This agrees with the analysis of this parameter in Section

	DA^{OTF}	$DA^{precomp}$
Abstractionsize limit	4000	1024
Split method	*Single Value Split*	*Single Value Split*
Initial abstraction	most-corse	goals splitted
How many facts to split	all	all
Obtain heuristic values	on demand	precomputed

Table 4.1: Details of best-performing configurations

4.3.4 as abstractions using initial goal split tend to perform better when having just a low abstraction size limit as it is the case here. This is a very good example that a general performance tendency of a parameter might not hold in general for all possible configuration combinations. In general we can observe that the configurations achieving a good balance between informativeness of the heuristic and time needed to obtain the heuristic values tend to have the best performance. We can use the performance analysis for each parameter as orientation for a configuration and we will get good results with a high probability, but this depends also on the exact setting for each parameter as $DA^{precomp}$ shows.
After this remark on general best configurations, we will compare the performance of $DA^{precomp}$ and DA^{OTF} in detail.

	DA^{OTF}	$DA^{precomp}$
Coverage	**768**	765
Total Time(geometric mean)	5.33	**2.26**

Table 4.2: Coverage and geometric mean of total time for best configurations

A view on table 4.2, shows that DA^{OTF} reaches a little-bit more coverage at the cost of worse mean time.

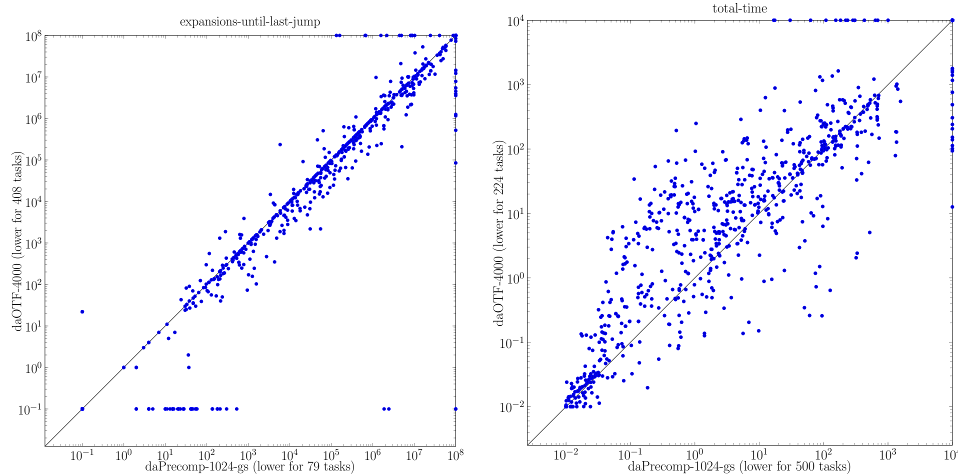


Figure 4.12: Comparison by Left: expansions until last jump, Right: total time

Our analysis of the figures 4.4 and 4.4 further shows that even though the expansions until last jump are lower for DA^{OTF} , the search time is higher. This shows that the heuristic is more informative than $DA^{precomp}$ but needs more time to obtain its heuristic values.

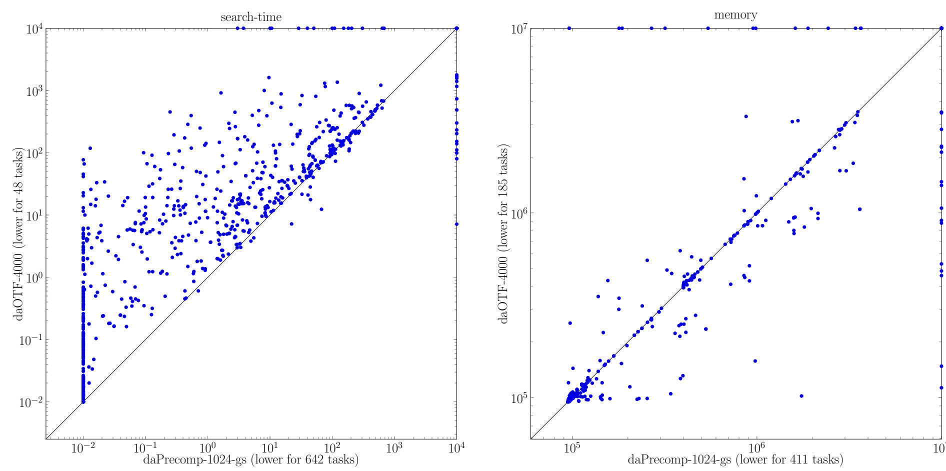


Figure 4.13: Comparison by Left: search time, Right: memory

Thus also the total time to solve a task is better for $DA^{precomp}$ because it over weighs the benefit of a more informative heuristic. Nevertheless, both algorithms were able to achieve a comparable coverage, where the one algorithm achieves this by searching very fast and the other by having a more informed heuristic with a good balance between the time needed to obtain heuristic values.

5

Comparison with CEGAR for Pattern Databases and Cartesian Abstractions

In the last two chapters, we discussed and evaluated a new method to create domain abstractions for abstraction heuristics. This chapter will compare the best-performing configurations of our algorithm to other methods that create abstraction heuristics based on other abstraction classes.
As mentioned in Chapter 3, we are unaware of any algorithm that constructs domain abstractions. Even the paper that introduced domain abstractions (Hernádvölgyi and Holte, 2000) just generated them randomly and did not specify how. This is why we compare our algorithm with abstraction heuristics that work with other abstraction classes, namely *projections* and *Cartesian abstractions*. The abstraction heuristics we compare our work with also use the CEGAR principle to construct the abstractions. First, we will compare against the PDB^{nadd} and PDB^{add} algorithms that correspond to the algorithms by Rovner et al. (2019), which are Pattern Database Heuristics that use multiple projections. Secondly, we will compare with an abstraction heuristic based on one single Cartesian abstraction introduced by Seipp and Helmert (2013).
Section 5.1 will introduce the underlying theory behind the algorithms we compare with our work. Next, Section 5.2 will present the algorithms and configurations we use in our benchmarks. Finally, we will discuss the benchmark results in Section 5.3.

5.1 Projections, Pattern Databases and Cartesian Abstractions

Projections and Cartesian abstractions are closely related to domain abstractions. While domain abstractions are a proper generalisation of projections, Cartesian abstractions are a proper generalisation of domain abstractions. In the following, we briefly introduce both abstraction classes and compare them to domain abstractions on a theoretical level.

5.1.1 Projections and Pattern Database Heuristics

Pattern database heuristics, are abstraction heuristics, that are induced by the abstraction class of *projections* (Culberson and Schaeffer, 1998). Given a planning task Π with variables

V, a projection $\alpha_P : S \mapsto S^\alpha$ is induced by a subset of the variables $P \subseteq V$ which is also called a *pattern*. Two states $s, s' \in S$ are mapped to the same abstract state $s^\alpha \in S^\alpha$ if they agree on the values of all variables in P. Formally this means that $\alpha_P(s) = \alpha_P(s')$ iff $s(p) = s'(p)$ for all $p \in P$.
In a projection, a variable either is irrelevant or gets considered completely. It is easy to see, that domain abstractions are a proper generalisation of projections, where projection is a domain abstraction $\sim = \{\sim_1, \ldots, \sim_n\}$ with $\mid \sim_i \mid = 1$ for $p \notin P$ and $\mid \sim_i \mid = |D_i|$ in case $p \in P$.
Pattern Database heuristics compute abstract goal distances in S^α and store them into a lookup table in memory, also called a *pattern database*. Like in other abstraction heuristics, these are then used to estimate goal distances in S.

5.1.2 Cartesian Abstractions

Cartesian Abstractions for planning (Seipp and Helmert, 2013) were first explicitly introduced with the idea of using the CEGAR principle and are a proper generalisation of domain abstractions. They are more general than domain abstractions and allow for fine-grained abstraction refinement in the places where needed. They achieve this by allowing to refine domains in a specific part of the planning task while other areas stay untouched. Each abstract state can define individual domains for the variables in the planning task. This allows for a very flexible refinement of the abstraction.
To explain this more formally we refer to the definition by Seipp and Helmert (2013): we are given a planning task Π with variables $V = \langle v_1, \ldots, v_n \rangle$ and their associated domains $D_1, \ldots, D_n$. In a planning task a set of states is called Cartesian if it has the form $C_1 \times \cdots \times C_n$ where $C_i \subseteq D_i$ for all $1 \leq i \leq n$. An abstraction is called Cartesian if all its abstract states are Cartesian sets. For an abstract state $s_\alpha = C_1 \times \cdots \times C_n$ we define $D_{i,s_\alpha} = C_i \subseteq D_i$ as the set of variables that variable v_i can have in abstract state s_α .
We can see that every abstract state of the abstraction-induced state space can define individual domains for each variable. This makes Cartesian abstractions a proper generalisation of domain abstractions, as domain abstractions are a particular case of Cartesian abstractions, where the domain of each variable v_i is equal in all abstract states.

5.2 The competing algorithms

This chapter will present the four algorithms that will be compared using the A* search algorithm in Section 5.3. All algorithms construct abstraction heuristics using the CEGAR principle, making the benchmarks more comparable. The exact commands and parameters that were used to run these algorithms can be found in the appendix.

5.2.1 Projection

The first two algorithms that participate in the comparison are the algorithms PDB^{nadd} and PDB^{add} that correspond to the according algorithms by Rovner et al. (2019), both creating a collection of multiple patterns. These patterns are then combined into one admissible

abstraction heuristic. The difference between the two algorithms is that PDB^{add} constructs additive patterns, whereas the patterns that PDB^{nadd} constructs are not additive. Two patterns P_1 and P_2 are disjoint when P_1 does not contain any variable of P_2. Two disjoint patterns are additive when no variable in P_1 is correlated with any variable in P_2. They never occur together in the effects of any action a, and neither does one occur in the preconditions and the other in the effects of the same action. To have an up-to-date comparison, we chose the best-performing configurations of these algorithms.
Furthermore, we will compare with a third algorithm $PDB^{singlePattern}$ or PDB^{SP} for short, that constructs just one pattern using the CEGAR algorithm by Rovner et al. (2019).
We compare domain abstractions with different pattern database heuristics that first use one abstraction and second use multiple abstractions. This is because the grade of refinement of a projection is a shortcoming and is why PDBs in practice often use multiple abstractions. As we want a complete overview of CEGAR performance for different abstraction classes, we also include a configuration that uses one projection.

5.2.2 Cartesian Abstraction

We use the algorithm from the work where Cartesian abstractions were introduced (Seipp and Helmert, 2013). As for PDBs, the algorithm uses the CEGAR principle to construct one or multiple cartesian abstractions. We will use a configuration where just one Cartesian abstraction is constructed since we also construct one domain abstraction. Furthermore, a domain abstraction has no benefit regarding the refinement grade, as was the case for Pattern Databases. From now on, we will denote this algorithm as $CARTESIAN^{singleAbstraction}$ or short $CARTESIAN^{SA}$.

5.2.3 Domain Abstraction

The configurations of our algorithm we selected to compete are the ones we found best-performing in our evaluation in Chapter 4. The first configuration will be denoted as DA^{OTF} and is the configuration that achieves a maximum coverage and obtains the heuristic values on the fly. Since we do not just compare the coverage but also the total time, we also selected a configuration with high coverage and an excellent performance regarding total time and expansions. This configuration will be denoted as $DA^{precomp}$ and precomputes the heuristic values. Both configurations are described in detail in Section 4.4.

5.2.4 Explicit transition systems

A domain abstraction or, more specifically, its equivalence relations apply to the whole state space equally. Hence there is no need to store an explicit representation of the abstract state space during refinement or precomputation. We instead implemented a black box interface that, based on an action or abstract state, computes the predecessors or successors of the given state. On the one hand, this makes the implementation easier and saves memory. On the other hand, an explicitly stored transition system could save computation time when implemented efficiently. Despite the refinement loop being very fast, even without an explicit

transition system, it would make sense for the precomputation of heuristic values. This is currently the main bottleneck of those configurations of our algorithm that precompute the heuristic values. A more efficient precomputation would allow a higher abstraction-size limit, which might lead to better performance.
In contrast, constructing a Cartesian abstraction using CEGAR requires an explicit transition system as every abstract state has its own domains for each variable.

5.3 Results

For the comparison of the algorithms presented above, we are using the same Benchmark setup as described in Chapter 4. In this section, we first compare the methods that construct just one abstraction. Secondly, we compare those with the methods that construct multiple abstractions.

	$CARTESIAN^{SA}$	DA^{OTF}	$DA^{precomp}$	PDB^{SP}
coverage	791	765	764	761

Table 5.1: Covergae of all algorithms compared

Table 5.3 shows that the overall performance of algorithms using one abstraction is comparable. As we can see, $CARTESIAN^{SA}$ has the best coverage, whereas the other algorithms have lower coverage by about 20 and just apart by max 5. For a more detailed analysis, we will look at the pairwise comparison of the algorithms.

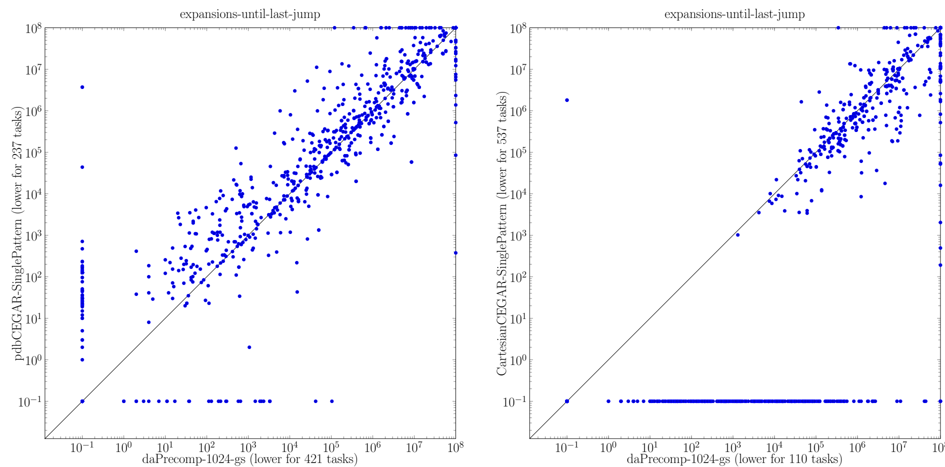


Figure 5.1: Comparison of $DA^{precomp}$ with $CARTESIAN^{SA}$ and PDB^{SP} by expansions until last jump

Consider Figures 5.3 and 5.3. Combined with the coverage data, our analysis shows increasing coverage with increasing abstraction class complexity while the time needed to solve a task decreases. One exception is DA^{OTF}, where the time is higher because it does not precompute the heuristic values as the other algorithms do. It makes sense that the more complex abstraction classes as CEGAR can do a more specified refinement that just refines parts of the abstraction that are necessary and not the domain of all variables as domain

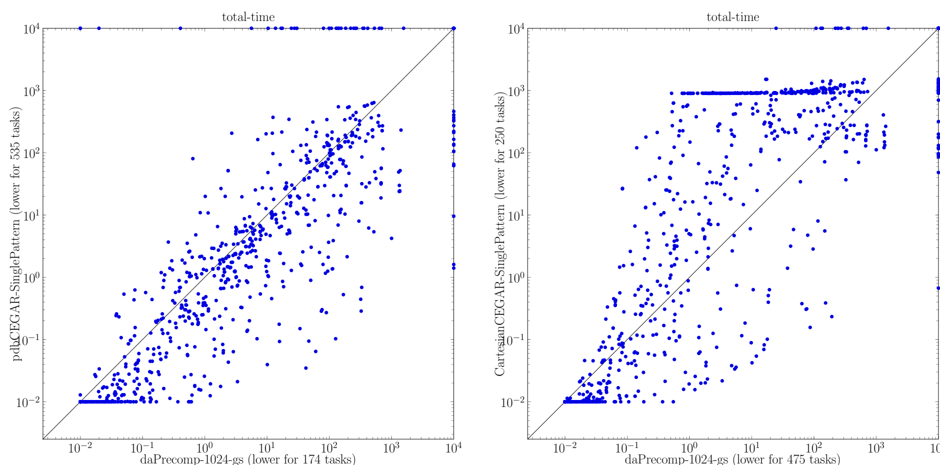


Figure 5.2: Comparison of $DA^{precomp}$ with $CARTESIAN^{SA}$ (right) and PDB^{SP} (left) by total time

abstractions do. In the same fashion, domain abstractions allow for a finer abstraction refinement than pattern databases, where every refinement step at least doubles the abstraction size. Consequently, more complex abstractions have a more efficient and accurate refinement that leads to more tasks solved and to more informative heuristics, as we can see in the plots denoting the expansions until the last jump, which are a good proxy for the informativeness of a heuristic.

Regarding the time, more complexity usually means more time spent on the refinement, which affects $CARTESIAN^{singleAbstraction}$ the most.

We also see that the algorithms $DA^{precomp}$ and PDB^{SP} are much more similar regarding the time they need to solve a task, search time and coverage. Generally, this is a lower improvement than we expected. However, we think that a more efficient implementation of the precomputation of heuristic values in $DA^{precomp}$ might lead to a more significant gap between the two algorithms. Another explanation for the segregation of $CARTESIAN^{singleAbstraction}$ from the other two could be the difference of the abstraction classes. Domain Abstractions as well as Pattern Databases make global decisions whereas Cartesian abstractions can focus their specificity on the parts of the State Space it matters the most. Overall we conclude that the performance of CEGAR algorithms for constructing domain abstractions is located in the area between Cartesian abstractions and projections, which is what we expected.

Now that we compared the algorithms that just construct one abstraction, we will look at the two algorithms that use the CEGAR principle to construct multiple projections.

	PDB^{add}	PDB^{nadd}
coverage	862	900

Table 5.2: Covergae of all algorithms compared

Table 5.3 clearly shows that the algorithms PDB^{add} and PDB^{nadd} have significantly higher coverage than the other algorithms. When we additionally consider the results of Seipp and Helmert (2013), that use multiple cartesian abstractions, we can conclude that constructing

multiple abstractions with CEGAR seems to work significantly better than just using one abstraction. We think this is the case because multiple distinct abstractions for the same task can be more informative than one big abstraction that has the same overall number of states. Moreover, the search in multiple small abstractions is often speedy, while searching in an abstraction gets more complicated with increasing abstraction size over linearly. Because domain abstractions, regarding their refinement grade lay in between projections and Cartesian abstractions we expect the construction of multiple domain abstractions to yield a comparable improvement in performance, especially regarding the coverage.

5.3.1 Conclusion

From the results, we can conclude that multiple abstractions benefit the performance of the abstraction heuristics. As Cartesian abstractions and Projections show better performance when constructing multiple abstractions, we think this would also benefit domain abstractions. When comparing the algorithms for all three abstraction classes that use one abstraction, we see that with increasing refinement grade that the abstraction class allows, the coverage increases. At the same time, the total-time increases with abstraction complexity. Overall our algorithm for domain abstractions performs as we expected when compared to other CEGAR algorithms that construct other abstraction classes where its performance and behaviour regarding time, coverage and informativeness of the heuristic lies between CEGAR for projections and Cartesian abstractions. We conclude that CEGAR is a descent algorithm for the construction of domain abstractions and has the potential when we would use multiple abstractions.

6 Conclusion and Future Work

This chapter summarises this thesis and briefly recalls what we presented. Furthermore, we give an outlook on future work and present ideas on how the presented algorithms can be improved or expanded.

Our analysis in Chapter 4 showed that for most unsolved tasks, the algorithm runs out of memory. Analysis of hard problem instances showed that the A* search and not the abstraction construction is the reason for that. One way to resolve this problem is to make the heuristic more informative, guiding the search algorithm better towards the goal.

As we saw in Chapter 5, constructing multiple abstractions with CEGAR yielded far better results in terms of coverage. Thus we think the construction of multiple domain abstractions would improve performance significantly. Moreover, other interesting ideas exist to improve the existing algorithm for constructing just one abstraction. One idea is to evaluate the effect of changing the number of facts in a flaw used for refinement. We only distinguish between choosing just one fact or all of them. We want to make it possible to split an arbitrary portion of the facts. Furthermore, expanding the amalgam of methods to pick a fact pair of a flaw for refinement could be interesting.

Another recently presented approach that could improve the construction of domain abstractions is to find not just one optimal plan in the abstraction and fix the flaw that occurs for it but to find all existing optimal plans and fix the flaw of each one. (Speck and Seipp, 2022). This approach showed to be successful for Cartesian abstractions, and we expect it would also benefit the construction of domain abstractions using CEGAR.

As we have seen in Chapter 4, the coverage of methods using on-demand calculation of heuristic values tends to be higher. We were surprised by this result as the state-of-the-art abstraction heuristics all precompute the heuristic values. We suspect that a more efficient implementation of the precomputation, namely using an explicit transition system, could yield significantly better performance for those configurations.

We have shown a CEGAR algorithm that constructs domain abstractions and yields promising results comparable with related algorithms constructing one projection or Cartesian abstraction. However, our algorithm could not match the coverage of techniques that construct a diverse set of abstractions and combine them by using cost-partitioning methods. As is the case for pattern databases and Cartesian abstractions, we believe that switching from

the construction of one to multiple domain abstractions would be highly beneficial.
Overall, we conclude that CEGAR is a promising approach to constructing domain abstractions and believe that the algorithm presented in this thesis can be extended to yield better coverage and more informative heuristics.

Bibliography

Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11(4):625–655, 1995.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

Joseph C. Culberson and Jonathan Schaeffer. Pattern Databases. *Computational Intelligence*, 14(3):318–334, 1998.

Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

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

István T. Hernádvölgyi and Robert C. Holte. PSVN: A Vector Representation for Production Systems. Technical Report TR-99-04, School of Infomation Technology and Engineering, University of Ottawa, 1999.

István T Hernádvölgyi and Robert C Holte. Experiments with Automatically Created Memory-Based Heuristics. In *Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation*, pages 281–290. Springer, 2000.

Alexander Rovner, Silvan Sievers, and Malte Helmert. Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*, pages 362–367, 2019.

Jendrik Seipp and Malte Helmert. Counterexample-Guided Cartesian Abstraction Refinement. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, pages 347–351, 2013.

David Speck and Jendrik Seipp. New Refinement Strategies For Cartesian Abstractions. In *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling*, pages "348–352", 2022.

A
Appendix

A.1 Commands executed in the comparison benchmarks

In the following one can depict the exact strings that were used to run the algorithms based on projections. The first two using multiple abstractions and the third one using just one:

PDB^{nadd} :

```
astar(cpdbs(multiple_cegar(max_pdb_size=1000000,
    max_collection_size=10000000,pattern_generation_max_time=
    infinity ,total_max_time=100,stagnation_limit=20,
    blacklist_trigger_percentage=0.75,
    enable_blacklist_on_stagnation=true ,random_seed=2018,
    verbosity=normal, use_wildcard_plans=false)),verbosity=silent
    )
```

PDB^{add} :

```
astar(cpdbs(disjoint_cegar(use_wildcard_plans=true ,max_time=100,
    max_pdb_size=1000000,max_collection_size=10000000,random_seed
    =2018,verbosity=normal)),verbosity=normal)
```

$PDB^{singlePattern}$:

```
astar(pdb(cegar_pattern(max_pdb_size=1000000,max_time=100,
    use_wildcard_plans=true , verbosity=normal, random_seed=2018)))
```

For the algorithm based on Cartesian abstractions we chose the following configuration to run:

$CARTESIAN^{singleAbstraction}$:

```
astar(cegar(subtasks=[original()],max_transitions=infinity ,
    max_time=900))
```

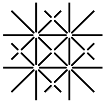
To run the algorithm based on domain abstractions we used the following strings to configure the search. One for the best version using precomputation of heuristic values and the best configuration taht obtains heuristic values on demand:

DA^{otf} :

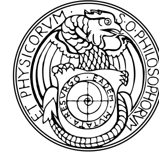
```
astar(domain_abstraction(precalculation=false, max_states=4000,
    initial_goal_split=false))
```

$DA^{precomp}$:

```
astar(domain_abstraction(precalculation=true, max_states=1024,
    initial_goal_split=true))
```


University of Basel

Faculty of Science



Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Generation of Domain Abstractions using Counterexample-Guided Abstraction Refinement

Name Assesor: Prof. Malte Helmert

Name Student: Raphael Kreft

Matriculation No.: 2019-058-148

With my signature I 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.

Place, Date: Lörrach, 30.06.2022 Student: R. Kreft

Will this work be published?

☐ No

☐ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: __________

Place, Date: Lörrach, 30.06.2022 Student: R. Kreft

Place, Date: __________ Assessor: __________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .