ALBERT-LUDWIGS-UNIVERSITÄT
FREIBURG
INSTITUT FÜR INFORMATIK

Arbeitsgruppe
Grundlagen der Künstlichen Intelligenz

Prof. Dr. Bernhard Nebel

Counterexample-guided Abstraction Refinement for Classical Planning

Master's Thesis

Jendrik Seipp

Advisor: Prof. Dr. Malte Helmert (Universität Basel)

December 2012

Acknowledgments

I would like to thank Prof. Dr. Nebel for supervising this thesis and Malte Helmert for his outstanding support and mentoring.

Erklärung (Declaration)

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

(I hereby declare that I wrote this thesis on my own, only making use of the sources mentioned and marking all parts taken from published works. Additionally, I confirm that this thesis has not been prepared for a different exam, neither in parts nor in its entirety.)

Freiburg im Breisgau, Dezember 2012 Jendrik Seipp

Abstract

Counterexample-guided abstraction refinement (CEGAR) is a methodological framework for incrementally computing abstractions of transition systems.

We propose a CEGAR algorithm for computing abstraction heuristics for optimal classical planning. Starting from a coarse abstraction of the planning task, we iteratively compute an optimal abstract solution, check if and why it fails for the concrete planning task and refine the abstraction so that the same failure cannot occur in future iterations.

A key ingredient of our approach is a novel class of abstractions for classical planning tasks that admits efficient and very fine-grained refinement. Our implementation performs tens of thousands of refinement steps in a few minutes and produces heuristics that are often significantly more accurate than *pattern database* heuristics of the same size.

Our second contribution is the application of those heuristics for heuristic improvement during search. We show how refining the abstraction online when the search errs reduces the number of expanded nodes.

Zusammenfassung

Counterexample-guided abstraction refinement (CEGAR) ist eine Methode aus dem Bereich des Model Checking, mit der inkrementell Abstraktionen von Transitionssystemen berechnet werden können.

In dieser Arbeit schlagen wir einen CEGAR Algorithmus für das optimale Lösen von klassischen Handlungsplanungsproblemen vor. Mit einer groben Abstraktion des Problems beginnend, finden wir iterativ eine optimale abstrakte Lösung, überprüfen ob und warum sie für das konkrete Planungsproblem scheitert und verfeinern die Abstraktion derart, dass derselbe Fehler in nachfolgenden Iterationen nicht mehr auftritt.

Ein wichtiger Teil unseres Ansatzes ist eine neuartige Klasse von Abstraktionen für klassische Planungsprobleme, die effiziente und sehr feinkörnige Verfeinerungen erlaubt. Unsere Implementation macht Zehntausende Verfeinerungen in wenigen Minuten und erzeugt Heuristiken, die häufig deutlich genauer sind als *Pattern-Database-Heuristiken* der gleichen Größe.

Außerdem zeigen wir, wie wir diese neuen Heuristiken auch noch während der Suche im Zustandsraum verfeinern und so die Anzahl der expandierten Zustände verringern können.

Contents

Chapter 1

Introduction

Classical planning is the problem of finding plans in a fully observable world with a single initial state, completely deterministic actions and a set of goal states. It has been shown that this problem is PSPACE-complete (Bylander 1994) and thus we cannot hope for polynomial algorithms that solve it optimally. When researchers first started working on classical planning, the predominant technique was to search for a plan in the space of all possible plans. Today however, traversing the space of world states is the prevailing method for solving planning problems. This had not been feasible earlier because the number of possible states can grow exorbitantly large even for small planning tasks and searching for a plan in such a system does not yield solutions in a reasonable amount of time if the search does not have a proper guidance. Recent research overcame this obstacle with the introduction of planning *heuristics* for state-space search. The search can navigate through the state space efficiently by querying such a heuristic for the estimated goal distance of potential next visited states and choose one that appears to be closest to the goal.

Heuristics for classical planning can be divided into four categories: Delete relaxation heuristics, critical path heuristics, landmark heuristics and abstraction heuristics. In this work we focus on the latter and create a new class of abstractions for classical planning by using counterexample-guided abstraction refinement (CEGAR).

CEGAR is an established technique for model checking in large systems (Clarke, Grumberg, and Peled 2000; Clarke et al. 2000). The idea is to start from a coarse (i. e., small and inaccurate) abstraction, then iteratively improve (refine) the abstraction in only the necessary places. In model checking, this means that we search for error traces (behaviors that violate the system property we want to verify) in the abstract system, test if these error traces generalize to the actual system (called the *concrete system*), and only if not, refine the abstraction in such a way that this particular error trace is no longer an error trace of the abstraction.

1.1 Related work

Despite the similarity between model checking and planning, counterexample-guided abstraction refinement has not been thoroughly explored by the planning community. The work that comes closest to ours (Chatterjee et al. 2005) contains no experimental evaluation or indication that the proposed algorithm has been implemented. The algorithm is based on blind search, and we believe it is very unlikely to deliver competitive performance. Moreover, the paper has several critical technical errors which make the main contribution (Algorithm 1) unsound.

The purpose of CEGAR approaches in model checking is usually to prove the absence of an error trace. In this work, we use CEGAR to derive heuristics for optimal state-space search, and hence our CEGAR procedure does not necessarily have to completely solve the problem: abstraction refinement can be interrupted at any time to derive an admissible search heuristic. The application of CEGAR to finding optimal solutions is (to the best of our knowledge) novel.

Haslum (2012) introduces an algorithm for finding lower bounds on the solution cost of a planning task by iteratively "derelaxing" its delete relaxation. Keyder, Hoffmann, and Haslum (2012) apply this idea to build a strong satisficing planning system based on the FF heuristic. Our approach is similar in spirit, but technically very different from Haslum's because it is based on homomorphic abstraction rather than delete relaxation. As a consequence, our method performs shortest-path computations in abstract state spaces represented as explicit graphs in order to find abstract solutions, while Haslum's approach exploits structural properties of delete-free planning tasks.

A key component of our approach is a new class of abstractions for classical planning, called *Cartesian abstractions*, which allow efficient and very fine-grained refinement. Cartesian abstractions are a proper generalization of the abstractions that underlie pattern database heuristics (Culberson and Schaeffer 1998; Edelkamp 2001) in the sense that every heuristic that is compactly representable as a pattern database is compactly representable as a Cartesian abstraction, while the opposite is not true.

1.2 Outline

This work is organized as follows. After some preliminaries and a formal definition of Cartesian abstractions we present the CEGAR algorithm for optimal planning and illustrate it with an example. In Chapter 4 we go over some of the algorithmic decisions we made and the data structures we used to implement the algorithm within the Fast Downward planning system. We compare the obtained heuristic with other abstraction-based planning approaches in Chapter 5. Afterwards, we evaluate the effectiveness of using the maximum estimate of multiple abstractions as a heuristic. In the last chapter we show how our CEGAR approach can be used for improving the heuristic online during search.

Chapter 2

Terms and definitions

Throughout this work we will use a toy planning problem as our running example for many definitions and algorithms. The task is taken from the Gripper domain (McDermott 2000) in which a robot has to transport balls from room A to room B. In our example task the robot has a single gripper G and there is only one ball. The robot can pick up and drop the ball and move between the two rooms. Initially, the robot is in room A, so obviously the shortest solution for this problem is to let the robot pick up the ball, move to room B and drop the ball there.

2.1 Planning tasks and transition systems

We use a SAS$^+$-like (Bäckström and Nebel 1995) finite-domain representation to formalize planning tasks and note that planning tasks specified in PDDL can be converted to such a representation automatically (Helmert 2009).

Definition 2.1. ***Planning tasks.***
*A **planning task** is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ where:*

- *$\mathcal{V}$ is a finite set of **state variables**, each with an associated finite domain $\mathcal{D}(v)$.*

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

  *A **partial state** is a function s defined on some subset of $\mathcal{V}$. We denote this subset by $\mathcal{V}_s$. For all $v \in \mathcal{V}_s$, we must have $s(v) \in \mathcal{D}(v)$. Where notationally* *convenient, we treat partial states as sets of atoms. Partial states defined on all variables are called **states**, and $\mathcal{S}(\Pi)$ is the set of all states of Π.*

*The **update** of partial state s with partial state t, $s \oplus t$, is the partial state defined on $\mathcal{V}_s \cup \mathcal{V}_t$ which agrees with t on all $v \in \mathcal{V}_t$ and with s on all $v \in \mathcal{V}_s \setminus \mathcal{V}_t$.*

- *$\mathcal{O}$ is a finite set of **operators**. Each operator o is given by a **precondition** pre_o and **effect** eff_o, which are partial states, and a **cost** $cost_o \in \mathbb{N}_0$. The **postcondition** $post_o$ of an operator is defined as $pre_o \oplus eff_o$.*
- *We call s_0 the **initial state** and the partial state $s_\star$ the **goal**.*

In this notation our example can be written as the SAS$^+$ task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ with $\mathcal{V} = \{rob, ball\}$, $\mathcal{D}(rob) = \{A, B\}$, $\mathcal{D}(ball) = \{A, B, G\}$, $\mathcal{O} = \{$move-A-B, move-B-A, pick-in-A, pick-in-B, drop-in-A, drop-in-B$\}$, $s_0(rob) = A$, $s_0(ball) = A$ and $s_\star(ball) = B$. We demonstrate the definition of SAS$^+$ operators using the example of operator $o =$ pick-in-A. For o we have $pre_o(rob) = A$, $pre_o(ball) = A$, $eff_o(ball) = G$ and thus $post_o(rob) = A$ and $post_o(ball) = G$.

The notion of transition systems is central for assigning semantics to planning tasks:

Definition 2.2. ***Transition systems and plans.***
*A **transition system** $\mathcal{T} = \langle S, L, T, s_0, S_\star \rangle$ consists of a finite set of **states** S, a finite set of **transition labels** L, a set of labelled **transitions** $T \subseteq S \times L \times S$, an **initial state** s_0 and a set of **goal states** $S_\star \subseteq S$. Each label $l \in L$ has an associated **cost** $cost_l$.*

*A path from s_0 to any $s_\star \in S_\star$ following the labelled transitions T is a **plan** for $\mathcal{T}$. A plan is **optimal** if the sum of costs of the labels along the path is minimal.*

A planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ induces a transition system with states $\mathcal{S}(\Pi)$, labels $\mathcal{O}$, initial state s_0, goal states $\{s \in \mathcal{S}(\Pi) \mid s_\star \subseteq s\}$ and transitions $\{\langle s, o, s \oplus eff_o \rangle \mid s \in \mathcal{S}(\Pi), o \in \mathcal{O}, pre_o \subseteq s\}$. Optimal planning is the problem of finding an optimal plan in the transition system induced by a planning task or proving that no plan exists.

The transition system induced by the Gripper example is shown in Figure 2.1. It uses a pair notation for states in which the pair $\langle d_1, d_2 \rangle$ stands for the state s with $s(rob) = d_1$ and $s(ball) = d_2$.

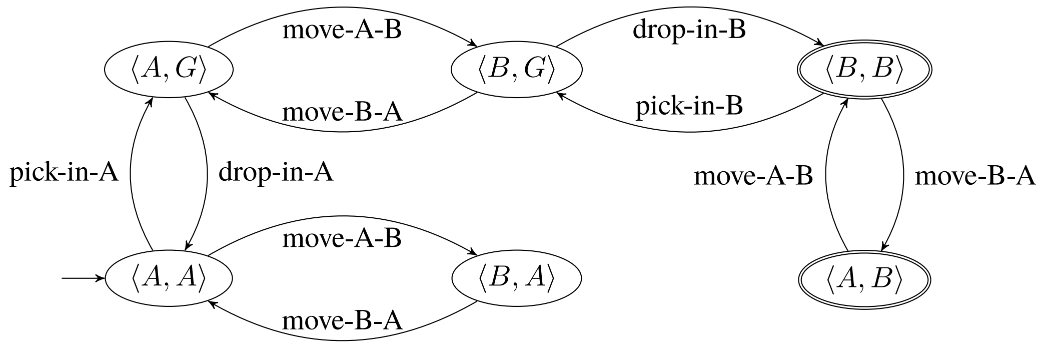


Figure 2.1: Transition system for the example Gripper task with a single ball, a gripper G and two rooms A and B. The pair $\langle d_1, d_2 \rangle$ stands for the state s with $s(\mathit{rob}) = d_1$ and $s(\mathit{ball}) = d_2$.

For notational convenience we introduce a more general form of this representation: we assume that the n state variables are (arbitrarily) numbered $v_1, \dots, v_n$ and write $\langle d_1, \dots, d_n \rangle$ to denote the state s with $s(v_i) = d_i$ for all $1 \leq i \leq n$.

2.2 Cartesian abstractions

Abstracting a planning task means losing some distinctions between states to obtain a more “coarse-grained”, and hence smaller, transition system. For our purposes it is convenient to use a definition based on equivalence relations:

Definition 2.3. ***Abstractions.***
Let Π be a planning task inducing the transition system $\langle S, L, T, s_0, S_\star \rangle$.

*An **abstraction relation** $\sim$ for Π is an equivalence relation on S. Its equivalence classes are called **abstract states**. We write $[s]_\sim$ for the equivalence class to which s belongs. The function mapping s to $[s]_\sim$ is called the **abstraction function**. We omit the subscript $\sim$ where clear from context.*

*The **abstract transition system** induced by $\sim$ is the transition system $\mathcal{T}$ with states $\{[s] \mid s \in S\}$, labels L, transitions $\{\langle [s], l, [s'] \rangle \mid \langle s, l, s' \rangle \in T\}$, initial state $[s_0]$ and goal states $\{[s_\star] \mid s_\star \in S_\star\}$.*

Abstraction preserves paths in the transition system and can therefore be used to define admissible and consistent heuristics for planning. Specifically, $h^\sim(s)$ is de-

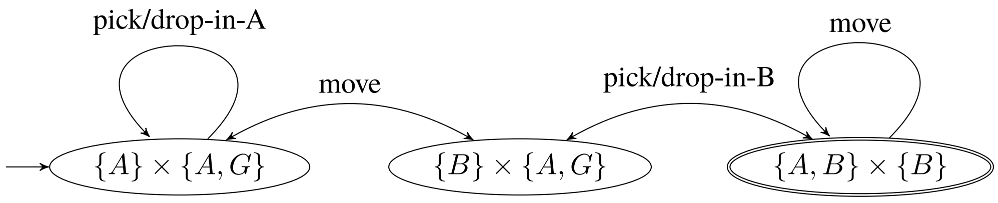


Figure 2.2: Example abstraction of the Gripper example. In the left and center state we know where the robot is, but not whether its gripper holds the ball, whereas in the abstract goal state we ignore the position of the robot.

fined as the cost of an optimal plan starting from $[s]$ in the abstract transition system. Practically useful abstractions should be efficiently computable and give rise to informative heuristics. These are conflicting objectives.

We want to construct compact and informative abstractions through an iterative refinement process. Choosing a suitable class of abstractions is critical for this. For example, *pattern databases* (Edelkamp 2001) do not allow fine-grained refinement steps, as every refinement at least doubles the number of abstract states. *Merge-and-shrink abstractions* (Helmert, Haslum, and Hoffmann 2007) do not maintain efficiently usable representations of the preimage of an abstract state, which makes their refinement complicated and expensive.

Because of these and other shortcomings, we introduce a new class of abstractions for planning tasks that is particularly suitable for abstraction refinement.

Definition 2.4. ***Cartesian sets and Cartesian abstractions.***
A set of states is called ***Cartesian*** *if it is of the form $A_1 \times A_2 \times ... \times A_n$, where $A_i \subseteq \mathcal{D}(v_i)$ for all $1 \leq i \leq n$.*

An abstraction is called ***Cartesian*** *if all its abstract states are Cartesian sets.*

Definition 2.5. ***Abstract domains.***
We define $\mathcal{D}_{[s]}(v) \subseteq \mathcal{D}(v)$ as the set of values a variable $v \in \mathcal{V}$ can have in the abstract state $[s]$.

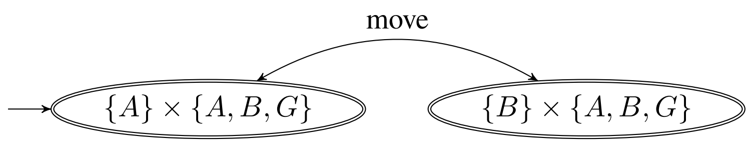
Figure 2.2 shows an example Cartesian abstraction for the gripper problem. The Cartesian sets $\{A\} \times \{A, G\}$, $\{B\} \times \{A, G\}$ and $\{A, B\} \times \{B\}$ are the states in the abstract transition system. The heuristic $h^{\sim}$ assigns them the distance to the abstract goal: 2, 1 and 0.

The name "Cartesian abstraction" was coined in the model-checking literature by Ball, Podelski, and Rajamani (2001) for a concept essentially equivalent to Definition 2.4. (Direct comparisons are difficult due to different state models.) Although Cartesian abstractions are a special case of merge-and-shrink abstractions, they still form a fairly general class since they include the concept of *domain abstraction* (Hernádvölgyi and Holte 2000) as a special case. Similarly to Cartesian abstractions, domain abstractions partition each variable's domain into sets of values that are considered equal. While in this abstraction class the same domain partitioning is made for all abstract states, the more general Cartesian abstractions allow segmenting the domains for each state individually.

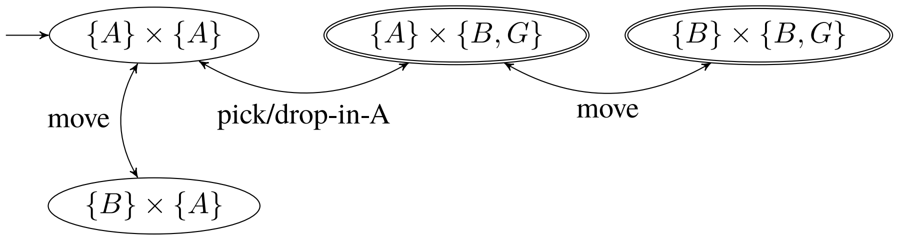
In turn, domain abstractions are more general than pattern databases, because every domain abstraction that either distinguishes between all or none of the values a variable can have for all variables in the task *is* a pattern database.

Unlike the two classes they subsume, general Cartesian abstractions can have very different levels of granularity in different parts of the abstract state space. One abstract state might correspond to a single concrete state, while another abstract state corresponds to half of the states of the task.

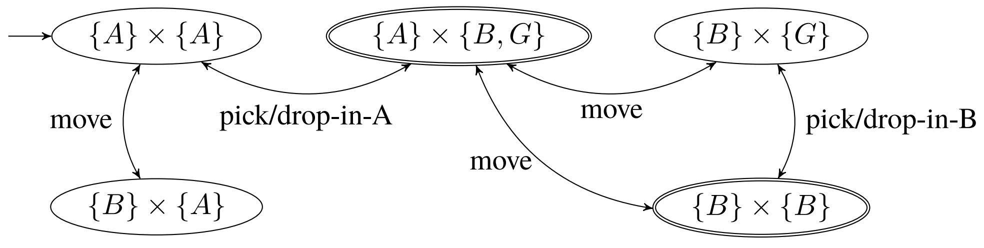
We illustrate the relationships between the different classes of abstractions with example abstractions of our Gripper task. Figure 2.3a shows the abstract transition system induced by the pattern database $\{rob\}$. Clearly, this abstraction is also a domain, Cartesian and merge-and-shrink abstraction. When we additionally partition the domain for variable *ball* into the groups $\{A\}$ and $\{B, G\}$, the abstraction is not a pattern database anymore (Figure 2.3b). The most specific formalism it adheres to now is domain abstraction. A further split of state $\{B\} \times \{B, G\}$ into the two states $\{B\} \times \{B\}$ and $\{B\} \times \{G\}$ yields the Cartesian abstraction in Figure 2.3c. Since not all domains are split equally for all states, it is not a domain abstraction anymore. Combining the states $\{A\} \times \{B, G\}$ and $\{B\} \times \{A\}$ results in the system in Figure 2.3d. This abstraction is not Cartesian anymore, but can be expressed in the merge-and-shrink formalism.



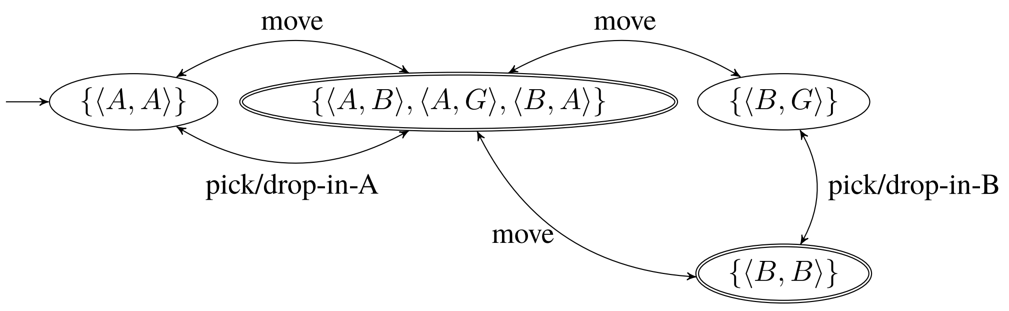
(a) Pattern database.



(b) Domain abstraction.



(c) Cartesian abstraction.


move
move
$\{\langle A, A\rangle\}$
$\{\langle A, B\rangle, \langle A, G\rangle, \langle B, A\rangle\}$
$\{\langle B, G\rangle\}$
pick/drop-in-A
pick/drop-in-B
move
$\{\langle B, B\rangle\}$

(d) Merge-and-shrink abstraction.

Figure 2.3: Example abstractions of the Gripper task for different classes of abstractions. The captions report the most specific class each abstraction belongs to. Self-loops are omitted to avoid clutter.

Theorem 2.6. ***Properties of Cartesian sets.***
The following properties make Cartesian sets interesting for CEGAR in planning:

- *(P1) The set of goal states of a planning task is Cartesian.*
- *(P2) The set of states where operator o is applicable is Cartesian.*
- *(P3) The intersection of Cartesian sets is Cartesian.*
- *(P4) The regression of operator o over a Cartesian set is a Cartesian set.*

Proof: Let $n = |\mathcal{V}|$.

- (P1) The set of goal states can be written as
  $A_1 \times A_2 \times ... \times A_n$ with $A_i = \{s_\star(v_i)\}$ if $s_\star(v_i)$ defined, else $A_i = \mathcal{D}(v_i)$.
- (P2) The set of states where operator o is applicable can be written as $\varphi = A_1 \times A_2 \times ... \times A_n$ with $A_i = \{pre_o(v_i)\}$ if $pre_o(v_i)$ defined, else $A_i = \mathcal{D}(v_i)$.
- (P3) Given two Cartesian sets X and Y with $X = A_1 \times A_2 \times ... \times A_n$ and $Y = B_1 \times B_2 \times ... \times B_n$ the intersection $X \cap Y$ can be written as $(A_1 \cap B_1) \times (A_2 \cap B_2) \times ... \times (A_n \cap B_n)$. Since each individual intersection $(A_i \cap B_i) \subseteq \mathcal{D}(v_i)$ for all $1 \leq i \leq n$, we have that $X \cap Y$ is Cartesian.
- (P4) The procedure for regressing an abstract state over an operator is presented in Algorithm 3.3 on page 18. It shows that the regression is Cartesian.

■

Moreover, the respective computational problems (intersecting two Cartesian sets, computing the regression, etc.) can all be performed in $O(n)$ time, where n is the number of atoms of the planning task. We will show a suitable representation for Cartesian sets guaranteeing this assessment in Chapter 4. The following theorem states two additional runtime guarantees.

Theorem 2.7. ***Runtimes of operations on Cartesian sets.***
If n is the number of atoms in the planning task, the following functions are computable in $O(n)$ for Cartesian abstractions $\sim$:

- *(R1) Given $s \in S$, compute $[s]_\sim$ and $h^\sim(s)$ (after abstract goal distances have been precomputed).*
- *(R2) Given Cartesian sets A and B and operator o, decide if $\langle A, o, B\rangle$ is an abstract transition.*

Proof:

- (R1) We present the function that computes the abstract state corresponding to a given concrete state in Algorithm 4.1 on page 25. Since we traverse the refinement hierarchy from the root node and move down one layer with each iteration of the algorithm, the algorithm runs in time $O(j \cdot k)$ if j is the height of the tree and each iteration runs in time $O(k)$. The number of atoms in the planning task n is an upper bound on j because in the worst case, we split off a single atom of the same state repeatedly. In that case GetWantedAtoms returns a singleton and thus we have that $O(k) = O(1)$. Since we can only split off at most n atoms it follows from $j \leq n$ that the algorithm runs in $O(n)$. (We note that in our implementation we ensure that at each node in the refinement hierarchy only the value of a single variable has to be taken into account for determining the appropriate child node by adding additional "helper" nodes to the tree if more than one atom is split off an abstract state.)
- (R2) Algorithm 3.6 on page 22 shows the general procedure for checking if a transition exists between two states. The runtime of the third case in the loop dominates the runtimes of the other two because it involves the intersection of domains instead of a simple membership test. One intersection for variable v runs in time $O(|\mathcal{D}(v)|)$. In the worst case an intersection is performed for each variable $v \in \mathcal{V}$ thus the whole algorithm has the asymptotic runtime $O(\sum_{i=1}^{|\mathcal{V}|} |\mathcal{D}(v_i)|) = O(n)$.

■

Chapter 3

Abstraction refinement algorithm

We now describe our abstraction refinement algorithm (Alg. 3.1). At every time, the algorithm maintains a Cartesian abstraction $\mathcal{T}'$, which it represents as an explicit graph. Initially, $\mathcal{T}'$ is the trivial abstraction with only one abstract state. The algorithm iteratively refines the abstraction until a termination criterion is satisfied (usually a time or memory limit). At this point, $\mathcal{T}'$ can be used to derive an admissible heuristic for state-space search algorithms.

Each iteration of the refinement loop first computes an optimal solution for the current abstraction, which is returned as a *trace* τ' (i. e., as an interleaved sequence of abstract states and operators $\langle [s'_0], o_1, \dots, [s'_{n-1}], o_n, [s_n] \rangle$ that form a minimal-cost goal path in the transition system). If no such trace exists (τ' is undefined), the abstract task is unsolvable, and hence the concrete task is also unsolvable: we are done.

Algorithm 3.1 Refinement loop.

```
$\mathcal{T}' \leftarrow$ TrivialAbstraction()
while not Terminate() do
    $\tau' \leftarrow$ FindOptimalTrace($\mathcal{T}'$)
    if $\tau'$ is undefined then
        return task is unsolvable
    $(s, \varphi) \leftarrow$ FindFlaw($\tau'$)
    if $\varphi$ is undefined (there is no flaw in $\tau'$) then
        return plan extracted from $\tau'$
    $\mathcal{T}' \leftarrow$ Refine($\mathcal{T}', s, \varphi$)
return $\mathcal{T}'$
```

Algorithm 3.2 Return the first "flawed" concrete state in the concrete trace of the abstract solution τ' and the flaw that occurred in it. In the case of diverging abstract and concrete paths we consider the last state on the common path flawed.

```
function FindFlaw($\tau'$)
    $\langle [s'_0], o_1, \dots, [s'_{n-1}], o_n, [s_n] \rangle \leftarrow \tau'$
    for all $i \in \{0, \dots, n\}$ do
        if $[s_i] \neq [s'_i]$ then
            $\varphi \leftarrow$ Regress($[s'_i], o_i$)
            return $(s_{i-1}, \varphi)$
        else if $i = n$ then
            if $s_n$ is a goal state then
                return (undefined, undefined)
            else
                $\varphi \leftarrow A_1 \times A_2 \times ... \times A_{|\mathcal{V}|}$ with $A_i = \begin{cases} \{s_\star(v_i)\} & \text{if } s_\star(v_i) \text{ defined} \\ \mathcal{D}(v_i) & \text{else} \end{cases}$
                return $(s_n, \varphi)$
        else if $o_{i+1}$ is not applicable in $s_i$ then
            $\varphi \leftarrow A_1 \times A_2 \times ... \times A_{|\mathcal{V}|}$ with $A_i = \begin{cases} \{pre_{o_{i+1}}(v_i)\} & \text{if } pre_{o_{i+1}}(v_i) \text{ defined} \\ \mathcal{D}(v_i) & \text{else} \end{cases}$
            return $(s_i, \varphi)$
        else
            $s_{i+1} \leftarrow$ Apply($s_i, o_{i+1}$)
```

Algorithm 3.3 Regress the abstract state $[s'] \in S'$ over operator $o \in \mathcal{O}$.

```
function Regress($[s'] \in S', o \in \mathcal{O}$)
    $[r'] \leftarrow$ new abstract state
    for all $v \in \mathcal{V}$ do
        if $post_o(v)$ undefined then
            $\mathcal{D}_{[r']}(v) \leftarrow \mathcal{D}_{[s']}(v)$
        else if $post_o(v) \notin \mathcal{D}_{[s']}(v)$ then
            return $\emptyset$
        else if $pre_o(v)$ defined then
            $\mathcal{D}_{[r']}(v) \leftarrow \{pre_o(v)\}$
        else if $eff_o(v)$ defined then
            $\mathcal{D}_{[r']}(v) \leftarrow \mathcal{D}(v)$
    return $[r']$
```

Otherwise, we attempt to convert τ' into a *concrete* trace in the FINDFLAW procedure shown in Algorithm 3.2. This procedure starts from the initial state of the concrete task and iteratively applies the next operator in τ' to construct a sequence of concrete states $s_0, \dots, s_n$ until one of the following *flaws* is encountered:

1. Concrete state s_i does not fit the abstract state $[s'_i]$ in τ', i. e., $[s_i] \neq [s'_i]$: the concrete and abstract traces diverge. This can happen because abstract transition systems are not necessarily deterministic: the same state can have multiple outgoing arcs with the same label.

2. The concrete trace has been completed, but s_n is not a goal state.

3. Operator o_{i+1} is not applicable in concrete state s_i.

The check for diverging traces is made first in order to ensure that the other two checks always operate on the correct concrete states.

If none of these conditions occurs, we have found an optimal solution for the concrete task and can terminate. Otherwise, FINDFLAW returns the first "flawed" state $s \in S$ and the flaw φ that occurred in it. In the case of diverging abstract and concrete paths we consider the last concrete state on the common path flawed. Properties P1, P2 and P4 from Theorem 2.6 entail that φ is a Cartesian set. It indicates the hypothetical state we would have wanted to land in.

Afterwards, we proceed by refining the abstraction so that φ cannot arise in future iterations by splitting $[s]$ into two abstract states $[t']$ and $[u']$ in a way suited for φ. The corresponding REFINE function is shown in Algorithm 3.4. It calls the method GETPOSSIBLESPLITS (Alg. 3.5) to determine possible splits of state $[s]$ for flaw φ. A split on variable v is feasible if its value in s is not in the set of "wanted" values $\mathcal{D}_\varphi(v)$. In that case we have to separate $s(v) \in$ *unwanted* from *wanted* by splitting the abstract domain $\mathcal{D}_{[s]}(v)$ appropriately. We are free to choose the partition of the remaining values *unwanted* $\setminus \{s(v)\}$ and put them into the same partition as $s(v)$. This likely results in a greater increase of the average h value because the state containing the *wanted* values is probably closer to the goal.

Since sometimes there may be multiple variables with feasible splits we call the function CHOOSEVARIABLE to make a decision. For now we assume that the choice is made randomly but compare different selection methods in Chapter 4.

Algorithm 3.4 Refine the abstraction by splitting the abstract state $[s]$ into two new states.

```
function REFINE($\mathcal{T}', s \in S, \varphi$)
   $\langle S', L', T', [s_0], S'_\star \rangle \leftarrow \mathcal{T}'$
   candidates $\leftarrow$ GETPOSSIBLESPLITS($s, \varphi$)
   $\langle v, \mathit{wanted}, \mathit{unwanted} \rangle \leftarrow$ CHOOSEVARIABLE(candidates)
   $[t'], [u'] \leftarrow$ new abstract states
   $S' \leftarrow S' \cup \{[t'], [u']\}$
   $\mathcal{D}_{[t']}(v) \leftarrow \mathit{unwanted}$
   $\mathcal{D}_{[u']}(v) \leftarrow \mathit{wanted}$
   for all $v_i \in \mathcal{V}, v_i \neq v$ do
      $\mathcal{D}_{[t']}(v_i) \leftarrow \mathcal{D}_{[s]}(v_i)$
      $\mathcal{D}_{[u']}(v_i) \leftarrow \mathcal{D}_{[s]}(v_i)$
   for all $([r'], o, [s]) \in T', [r'] \neq [s]$ do
      CHECKANDADDARC($[r'], o, [t']$)
      CHECKANDADDARC($[r'], o, [u']$)
      REMOVEARC($[r'], o, [s]$)
   for all $([s], o, [w']) \in T', [w'] \neq [s]$ do
      CHECKANDADDARC($[t'], o, [w']$)
      CHECKANDADDARC($[u'], o, [w']$)
      REMOVEARC($[s], o, [w']$)
   for all $([s], o, [s]) \in T'$ do
      CHECKANDADDARC($[t'], o, [u']$)
      CHECKANDADDARC($[u'], o, [t']$)
      CHECKANDADDARC($[t'], o, [t']$)
      CHECKANDADDARC($[u'], o, [u']$)
      REMOVEARC($[s], o, [s]$)
   $S' \leftarrow S' \setminus \{[s]\}$
   return $\mathcal{T}'$

procedure CHECKANDADDARC($[a'] \in S', o \in \mathcal{O}, [b'] \in S'$)
   if CHECKTRANSITION($[a'], o, [b']$) then
      ADDARC($[a'], o, [b']$)

procedure ADDARC($[a'] \in S', o \in \mathcal{O}, [b'] \in S'$)
   $T \leftarrow T \cup \{([a'], o, [b'])\}$

procedure REMOVEARC($[a'] \in S', o \in \mathcal{O}, [b'] \in S'$)
   $T \leftarrow T \setminus \{([a'], o, [b'])\}$
```

Algorithm 3.5 Return a list of variables together with domain partitionings that satisfy the requirements of a split.

```
function GetPossibleSplits($s \in S, \varphi$)
    candidates $\leftarrow \emptyset$
    for all $v \in \mathcal{V}$ do
        $\textit{wanted} \leftarrow \mathcal{D}_{[s]}(v) \cap \mathcal{D}_{\varphi}(v)$
        $\textit{unwanted} \leftarrow \mathcal{D}_{[s]}(v) \setminus \mathcal{D}_{\varphi}(v)$
        if $s(v) \in \textit{unwanted}$ then
            candidates $\leftarrow$ candidates $\cup \{\langle v, \textit{wanted}, \textit{unwanted} \rangle\}$
    return candidates
```

Regardless of the chosen variable the effect of a split can be described as follows using the variable names in the FindFlaw function. In the case of violated preconditions (3.), we split $[s_i]$ into $[t']$ and $[u']$ in such a way that $s_i \in [t']$ and o_{i+1} is inapplicable in all states in $[t']$. In the case of violated goals (2.), we split $[s_n]$ into $[t']$ and $[u']$ in such a way that $s_n \in [t']$ and $[t']$ contains no goal states. Finally, in the case of diverging traces (1.), we split $[s_{i-1}]$ into $[t']$ and $[u']$ in such a way that $s_{i-1} \in [t']$ and applying o_i to any state in $[t']$ cannot lead to a state in $[s_i']$. Performing this split involves computing the regression of $[s_i']$ over the operator o_i as shown in Algorithm 3.3. The Regress function calculates for an abstract state $[s'] \in S'$ and an operator $o \in \mathcal{O}$ the Cartesian set of states in which applying o leads into $[s']$. If the application of o can never take us into state $[s']$ the empty set is returned. Since we only call this algorithm for states $[s']$ that have been reached with o this case never occurs during the refinement.

Once a suitable split $\langle v, \textit{wanted}, \textit{unwanted} \rangle$ has been chosen, Refine updates the abstract transition system by replacing the state $[s]$ that was split with the two new abstract states $[t']$ and $[u']$. They inherit the abstract domains for all variables except v from $[s]$ and we assign $\mathcal{D}_{[u']}(v)$ the set of "wanted" atoms and $\mathcal{D}_{[t']}(v)$ the set of remaining atoms $\mathcal{D}_{[s]}(v) \setminus \textit{wanted} = \textit{unwanted}$. The last job of the Refine function is "rewiring" the new states. Here we need to decide for each incoming and outgoing transition of $[s]$ whether a corresponding transition needs to be connected to $[t']$, to $[u']$, or both. This check is done by Algorithm 3.6 which runs in time $O(n)$ where n is the number of atoms in the planning task as stated in Theorem 2.7.

Algorithm 3.6 Return true iff there should be an abstract transition labeled o between states $[s']$ and $[t']$.

```
function CheckTransition($[s'] \in S', o \in \mathcal{O}, [t'] \in S'$)
    for all $v \in \mathcal{V}$ do
        if $pre_o(v)$ defined and $pre_o(v) \notin \mathcal{D}_{[s']}(v)$ then
            return false
        if $post_o(v)$ defined and $post_o(v) \notin \mathcal{D}_{[t']}(v)$ then
            return false
        if $post_o(v)$ undefined and $\mathcal{D}_{[s']}(v) \cap \mathcal{D}_{[t']}(v) = \emptyset$ then
            return false
    return true
```

3.1 Example

Figure 3.1a shows the initial abstraction for the running example. The empty abstract solution $\langle\rangle$, which is an optimal plan for this abstraction since the abstract initial state is also an abstract goal state, does not solve Π because s_0 does not satisfy the goal. The GetPossibleSplits function returns the single split candidate $\langle \textit{ball}, \{B\}, \{A, G\}\rangle$ and Refine uses it to split $\mathcal{D}_{[s_0]}(\textit{ball})$ into the two partitions $\{A, G\}$ and $\{B\}$ leading to the finer abstraction in Figure 3.1b.

The plan $\langle$drop-in-B$\rangle$ does not solve Π because two preconditions are violated in s_0: *ball* $= G$ and *rob* $= B$. We assume that Refine performs a split based on variable *rob* (a split based on *ball* is also possible), leading to Figure 3.1c.

A further refinement step yields the system in Figure 3.1d with the abstract solution $\langle$move-A-B, drop-in-B$\rangle$. The first operator is applicable in s_0 and takes us into state s_1 with $s_1(\textit{rob}) = B$ and $s_1(\textit{ball}) = A$, but the second abstract state $[s_1'] = \{B\} \times \{A\}$ of the trace does not abstract s_1: the abstract and concrete paths diverge. Regression from $[s_1']$ for move-A-B yields the intermediate state $\varphi = \{A\} \times \{G\}$, and hence Refine must refine the abstract initial state $[s_0]$ in a way that separates φ from the concrete state s_0. The result of this refinement is shown in Figure 3.1e.

The solution for this abstraction is also a valid concrete solution, so we stop refining.

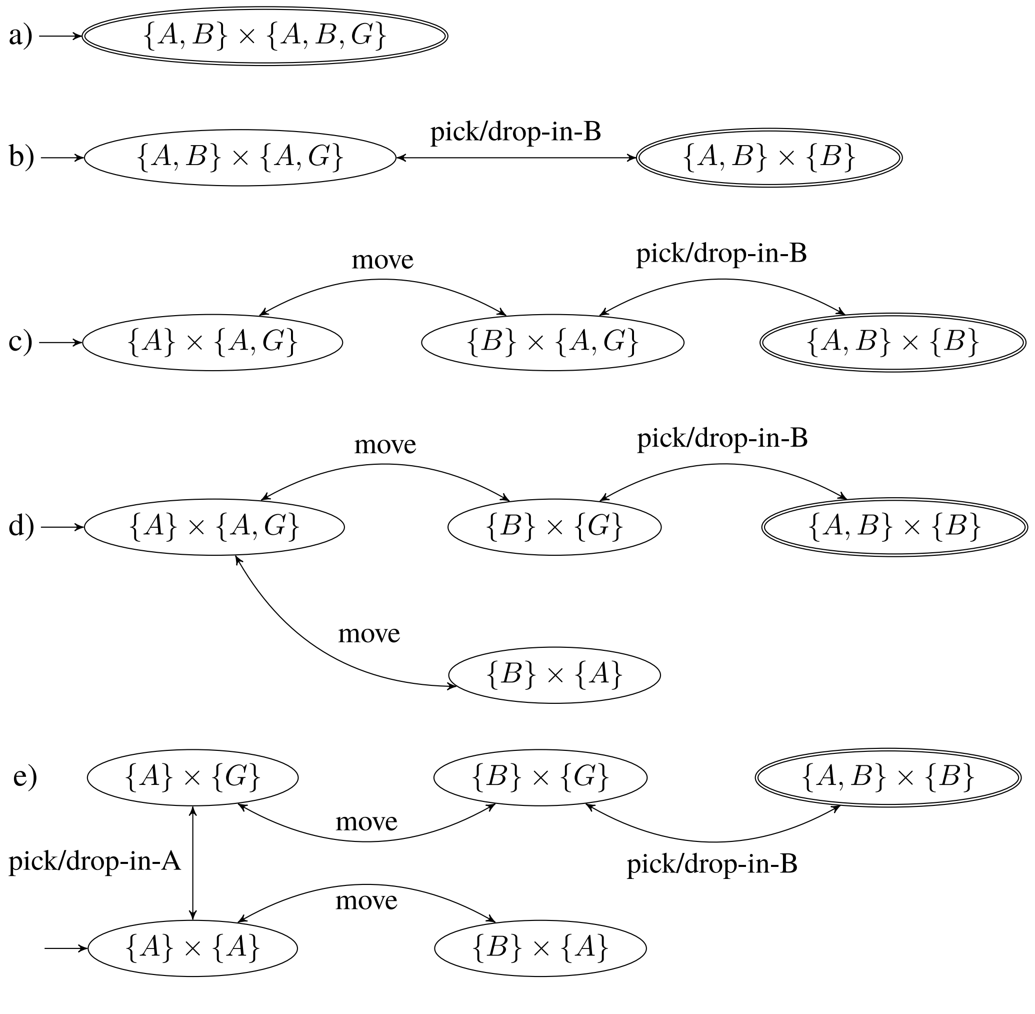


Figure 3.1: Refining the example abstraction. Self-loops are omitted to avoid clutter.

Chapter 4

Implementation

We implemented the abstraction refinement algorithm in the Fast Downward planning system (Helmert 2006) which consists of a variety of state-of-the-art heuristics and search algorithms. This not only allowed us to reuse many components that are necessary for a full planning algorithm, but also simplified comparing our new method to existing ideas.

4.1 Refinement hierarchy

After the abstraction has been computed, the A* search function traverses the state space and queries the heuristic for estimates for the evaluated states $s \in S$. It is critical that the abstraction calculates the corresponding abstract state $[s]$ and returns the precomputed abstract goal distance very fast. To this end, we store a *refinement hierarchy* of refined states instead of discarding them from memory when they are split. This hierarchy is a binary tree of abstract states with the single state in the trivial abstraction at its root. States that have been split have the two resulting new states as their child nodes. The leaves of the refinement tree are the states in the current abstraction. Figure 4.1 shows the refinement hierarchy for the final abstraction of the Gripper example from Figure 3.1e.

Additionally to the child nodes we also save the variable and the "wanted" atoms that each state has been split for and make this information available with the functions GetChildren, GetVariable and GetWantedAtoms that all take an abstract state

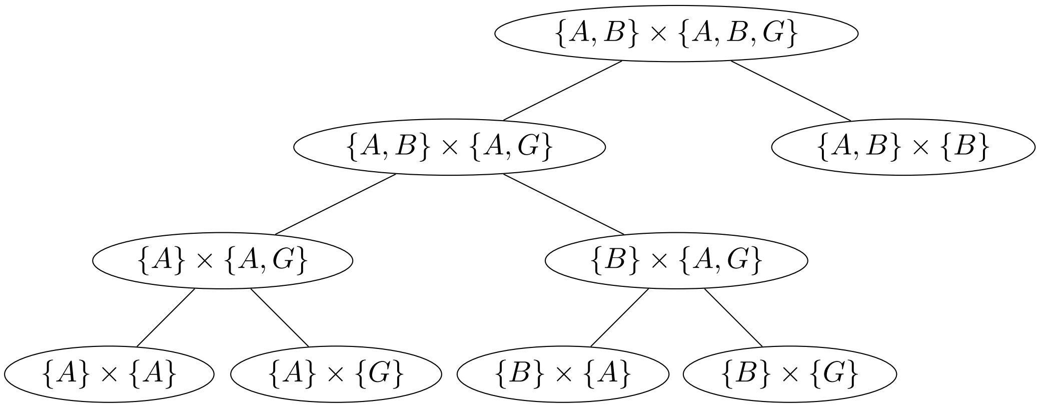


Figure 4.1: Refinement hierarchy for the final abstraction of the Gripper example from Figure 3.1e.

Algorithm 4.1 Return the equivalence class $[s]$ (i. e., the abstract state) for the concrete state s by traversing the refinement hierarchy.

```
function GetAbstractState($s \in S$)
    $[s'] \leftarrow$ root of the refinement hierarchy
    while GetChildren($[s']$) is defined do
        $[t'], [u'] \leftarrow$ GetChildren($[s']$)
        $v \leftarrow$ GetVariable($[s']$)
        if $s(v) \in$ GetWantedAtoms($[s']$) then
            $[s'] \leftarrow [u']$
        else
            $[s'] \leftarrow [t']$
    return $[s']$
```

as their single argument.

The proof of Theorem 2.7 shows how the refinement hierarchy allows for a fast computation of $[s]$ given s with Algorithm 4.1. Since the operation of retrieving the precomputed h value of an abstract state runs in time $O(1)$, the total time for computing $h(s)$ is linear in the number of atoms in the planning task.

4.2 Code optimizations

In this chapter we highlight some of the changes we made to our implementation after we had implemented the *original* version as described above. To evaluate their effectiveness we let each revision of the code build an abstraction consisting of at most 10 000 states within a time limit of five minutes for a number of benchmark domains from previous IPC challenges. Tables 4.1 and 4.2 report the time and memory consumption of each code version relative to the original implementation.

In these and all other tables containing relative data, we obtain the reported values in the following way: For each task we gather the values for the attribute in question (e. g. refinement time) from all configurations and calculate the ratio of each value to a reference value. This can be the value yielded by the first configuration or, e. g., the lowest value resulting from any configuration. We omit tasks for which at least one configuration has hit a time or memory limit and thus has no value for the problem or produced a value equal to zero. This is done in order to ensure that we compare all configurations on the same set of tasks. The numbers in brackets behind the domain names state how many tasks of each domain are taken into consideration for the comparison. For each domain we list the geometric mean of all ratios and report the geometric mean of the domain ratios in the last row.

Reuse solutions The refinement loop (Alg. 3.1) repeatedly finds and discards abstract solutions. An important insight is that we can save a lot of time by reusing the last found solution if it was not violated by the refinement. This is the case when the solution is still a valid path in the abstract system after we substitute the split state $[s]$ with one of its child states $[u']$ or $[t']$. Table 4.1 shows that we save 20% of the time when we use this shortcut compared to the original implementation.

A* search Our original implementation used Dijkstra's algorithm for finding shortest paths in the abstract transition system. The FindOptimalTrace function can be made significantly faster, however, if we use A* with the heuristic of the last iteration, i. e., the goal distance each node had before the refinement. This heuristic is admissible because a refinement can only increase the distances to goal nodes. Using this guidance during search reduces the times needed to build the abstraction by a factor of 4 on average across all domains. Some domains benefit even more from this change and the new runtime is only a tenth of the old one.

Refinement speedup	**Original**	**Reuse solutions**	**A* search**	**Loops vector**	**Single bitset**	**Transition check**	**Cascaded refinements**	**Keep transitions**	**Hash table**	h **updates**
airport (14)	1.00	1.33	4.35	3.60	5.00	11.68	22.13	25.68	**35.46**	29.14
blocks (26)	1.00	1.40	4.34	5.07	6.27	6.91	14.52	24.49	31.42	**32.18**
depot (5)	1.00	1.27	9.35	9.53	10.60	14.47	21.22	27.82	33.61	**34.17**
driverlog (14)	1.00	1.14	10.03	12.20	12.14	14.55	16.35	15.93	17.66	**20.17**
elevators-08 (30)	1.00	1.41	**2.00**	1.90	1.96	1.86	1.32	0.92	1.20	1.22
freecell (8)	1.00	1.13	7.56	8.24	10.60	13.76	13.14	23.90	30.85	**32.96**
grid (4)	1.00	1.41	7.82	16.47	13.29	24.81	25.22	35.95	**42.21**	38.47
gripper (19)	1.00	1.25	7.22	5.11	6.35	7.21	20.91	17.83	19.29	**21.07**
logistics-00 (25)	1.00	1.27	7.66	10.10	9.75	13.33	18.92	17.97	19.19	**19.92**
logistics-98 (6)	1.00	1.16	12.28	14.89	14.63	19.00	22.04	25.27	28.37	**28.87**
miconic (132)	1.00	1.13	5.60	6.02	6.76	7.66	8.19	6.81	8.74	**11.11**
mprime (13)	1.00	1.03	0.56	1.12	1.13	1.83	1.67	6.11	**10.19**	9.97
mystery (6)	1.00	1.04	2.05	1.92	1.76	3.10	3.23	4.37	5.73	**6.60**
openstacks-06 (14)	1.00	1.21	6.53	7.12	9.00	9.88	18.66	23.50	28.24	**30.15**
openstacks-08 (29)	1.00	1.42	1.24	1.18	1.54	1.80	1.17	1.82	**2.38**	2.21
parcprinter-08 (23)	1.00	1.32	7.36	11.89	13.42	17.62	20.51	18.04	20.94	**24.99**
pathways (9)	1.00	1.25	9.05	10.50	11.59	24.62	19.83	23.29	28.05	**30.45**
pegsol-08 (29)	1.00	1.38	1.51	1.78	2.03	**2.22**	1.94	1.34	1.82	1.80
pipesworld-nt (5)	1.00	1.07	3.49	3.95	4.08	6.98	9.83	14.86	**18.78**	18.69
pipesworld-t (6)	1.00	1.26	4.88	4.85	6.36	8.73	7.10	14.53	15.19	**17.81**
psr-small (27)	1.00	1.24	2.49	2.91	3.68	3.60	5.06	5.04	**6.16**	6.09
rovers (13)	1.00	1.32	13.66	15.77	16.93	24.04	23.43	22.06	24.23	**24.95**
satellite (7)	1.00	1.23	7.01	6.07	7.49	11.60	10.35	13.73	17.20	**17.61**
scanalyzer-08 (8)	1.00	1.33	4.33	5.11	5.14	6.13	4.66	6.22	6.94	**8.33**
sokoban-08 (30)	1.00	1.43	2.87	2.80	3.33	3.56	**5.61**	4.29	5.34	5.30
tpp (9)	1.00	1.30	13.08	19.21	22.80	32.06	29.67	28.91	31.21	**32.81**
transport-08 (15)	1.00	1.17	4.45	4.79	4.93	**6.00**	4.44	3.55	4.56	4.49
trucks (9)	1.00	1.30	11.67	16.55	16.79	23.00	40.28	34.36	40.85	**42.31**
wood-08 (12)	1.00	1.17	4.82	6.65	7.02	9.26	8.83	8.94	**11.81**	11.17
zenotravel (11)	1.00	1.19	10.24	12.91	12.08	17.73	17.86	19.93	24.02	**27.53**
Geom. mean (558)	1.00	1.25	5.05	5.85	6.45	8.68	9.83	11.34	13.86	**14.44**

Table 4.1: Relative refinement speedup of the individual code revisions compared to the time taken by the original implementation. Each number reports the geometric mean of all ratios for a given domain and the last row states their geometric mean. Best values are highlighted in bold.

Memory (relative)	**Original**	**Reuse solutions**	**A^* search**	**Loops vector**	**Single bitset**	**Transition check**	**Cascaded refinements**	**Keep transitions**	**Hash table**	h **updates**
airport (12)	1.00	1.00	1.03	1.02	0.28	0.26	**0.11**	0.23	0.23	0.24
blocks (28)	1.00	1.01	1.44	1.34	1.07	1.20	**0.63**	0.83	0.83	0.83
depot (6)	1.00	1.05	1.17	1.15	0.92	0.87	**0.58**	0.98	0.98	0.98
driverlog (14)	1.00	1.01	1.00	0.77	0.56	**0.55**	**0.55**	0.82	0.82	0.83
elevators-08 (30)	1.00	0.98	0.99	1.06	**0.95**	0.99	1.23	2.48	2.48	2.51
freecell (8)	1.00	1.06	1.09	1.05	**0.89**	0.94	0.98	2.16	2.16	2.12
grid (4)	1.00	1.02	0.87	0.73	0.58	0.45	**0.41**	0.55	0.56	0.56
gripper (20)	1.00	1.08	1.17	1.42	0.88	0.82	**0.26**	0.28	0.29	0.28
logistics-00 (26)	1.00	1.01	1.09	0.69	0.46	0.42	**0.41**	0.49	0.49	0.48
logistics-98 (6)	1.00	1.00	1.06	0.71	0.53	0.52	**0.50**	0.70	0.70	0.73
miconic (135)	1.00	1.00	1.00	1.01	0.48	**0.46**	0.47	0.76	0.76	0.76
mprime (13)	1.00	1.01	2.00	0.95	**0.89**	1.08	1.16	1.03	1.03	1.07
mystery (9)	1.00	1.04	1.29	1.19	1.01	1.10	**0.99**	1.21	1.21	1.22
openstacks-06 (14)	1.00	1.01	1.09	1.06	0.52	0.57	**0.32**	0.50	0.50	0.49
openstacks-08 (29)	1.00	1.04	1.14	1.14	0.68	**0.64**	0.75	1.20	1.20	1.27
parcprinter-08 (25)	1.00	0.95	0.91	0.68	0.27	0.27	**0.23**	0.30	0.31	0.31
pathways (10)	1.00	0.98	1.00	0.82	0.39	**0.35**	0.41	0.57	0.57	0.58
pegsol-08 (30)	1.00	0.99	1.10	1.06	**0.50**	**0.50**	0.55	1.00	1.00	0.98
pipesworld-nt (5)	1.00	0.98	1.39	1.37	0.94	0.87	**0.67**	0.84	0.84	0.88
pipesworld-t (6)	1.00	0.98	1.06	1.07	**0.80**	0.83	0.88	1.41	1.41	1.39
psr-small (31)	1.00	1.02	1.04	1.01	**0.39**	0.45	**0.39**	0.53	0.53	0.54
rovers (13)	1.00	1.01	0.99	0.72	**0.34**	**0.34**	0.38	0.53	0.54	0.55
satellite (7)	1.00	0.97	0.99	0.92	0.65	**0.64**	0.67	1.03	1.02	1.02
scanalyzer-08 (5)	1.00	0.96	1.02	0.97	**0.82**	0.84	0.89	1.75	1.75	1.76
sokoban-08 (30)	1.00	1.00	0.94	0.94	0.29	0.29	**0.24**	0.38	0.39	0.39
tpp (10)	1.00	0.99	1.00	0.71	0.26	**0.25**	0.27	0.33	0.33	0.33
transport-08 (16)	1.00	1.02	1.02	0.97	**0.79**	0.81	0.82	1.55	1.56	1.56
trucks (9)	1.00	1.00	1.02	0.76	0.57	0.56	**0.34**	0.72	0.72	0.73
wood-08 (12)	1.00	1.01	0.99	0.79	**0.52**	0.58	0.55	0.82	0.82	0.83
zenotravel (11)	1.00	1.00	0.97	0.70	**0.59**	0.60	0.66	1.09	1.09	1.09
Geom. mean (574)	1.00	1.01	1.08	0.94	0.58	0.58	**0.51**	0.76	0.77	0.77

Table 4.2: Relative amount of memory the individual code revisions need for building the abstraction compared to the amount used by the original implementation. Each number reports the geometric mean of all ratios for a given domain and the last row states their geometric mean. Best values are highlighted in bold.

Loops vector In the original implementation we used two vectors of incoming and outgoing transitions for each abstract state to represent the transitions T. We recognized however, that introducing a separate vector for self-loops, i. e., transitions of the form $([s'], o, [s'])$ for an abstract state $[s']$ and an operator $o \in \mathcal{O}$, further reduces the time needed to construct an abstraction. This is expected, because we can now easily avoid the overhead of following self-loops in the FindOptimalTrace function. Additionally, this change also saves a significant amount of memory. While we have to store two pointers for each "normal" transition, one for the operator and one for the destination state, we only need a single operator pointer for each self-loop. This results in a significant drop of memory usage for some domains as shown in Table 4.2.

Single bitset A simple way to save the abstract domains $\mathcal{D}_{[s']}(v)$, i. e., the sets of values each variable v can have in an abstract state $[s']$, is to use a vector of sets of integers. While we could have used the `set` data structure from the C++ standard library, we used the more space-efficient `dynamic_bitset` type from the Boost library to represent a set of integers in our original implementation. A further optimization is to combine the dynamic bitsets in a single one to allow for aligning the sets of values for variables with small domains (e. g. booleans) much more compactly. Internally dynamic bitsets are stored as vectors of words and each vector induces a certain amount of memory overhead. With this change we only need to store a single vector for each state, instead of one for every state plus one for every variable. Table 4.2 shows that this optimization is responsible for the biggest part of the memory savings introduced by all changes. It uses only $2/3$ of the memory the previous revision needs on average over all domains and as little as $1/3$ for some domains. As expected, the Airport domain with many boolean variables benefits the most from this replacement of data structures.

Informed transition check In Chapter 3 we have introduced the CheckTransition function (Alg. 3.6). Since this algorithm is called very often for each refinement it is crucial to make it as fast as possible. While it is perfectly suited for deciding if there should be an abstract transition between two states for an operator in general, we can leverage that we already have more information in our specific situation during a refinement because we only have to take into account the variable we refine on.

Algorithm 4.2 The refinement in progress splits state $[s]$ for variable $v \in \mathcal{V}$. This pseudocode adds new transitions for the old arc $([r'], o, [s])$ from $[r']$ to the new states $[t']$ and $[u']$ where necessary.

```
procedure UpdateIncomingArc(($[r'] \in S', o \in \mathcal{O}, [s] \in S'$), $v \in V$)
    $[t'], [u'] \leftarrow$ GetChildren($[s]$)
    if $post_o(v)$ undefined then
        if $|\mathcal{D}_{[r']}(v) \cap \mathcal{D}_{[t']}(v)| \neq \emptyset$ then
            AddArc($[r'], o, [t']$)
        if $|\mathcal{D}_{[r']}(v) \cap \mathcal{D}_{[u']}(v)| \neq \emptyset$ then
            AddArc($[r'], o, [u']$)
    else if $post_o(v) \in \mathcal{D}_{[u']}(v)$ then
        AddArc($[r'], o, [u']$)
    else
        AddArc($[r'], o, [t']$)
```

Algorithm 4.3 The refinement in progress splits state $[s]$ for variable $v \in \mathcal{V}$. This pseudocode adds new transitions for the old arc $([s], o, [w'])$ from the new states $[t']$ and $[u']$ to $[w']$ where necessary.

```
procedure UpdateOutgoingArc(($[s] \in S', o \in \mathcal{O}, [w'] \in S'$), $v \in V$)
    $[t'], [u'] \leftarrow$ GetChildren($[s]$)
    if $post_o(v)$ undefined then
        if $|\mathcal{D}_{[t']}(v) \cap \mathcal{D}_{[w']}(v)| \neq \emptyset$ then
            AddArc($[t'], o, [w']$)
        if $|\mathcal{D}_{[u']}(v) \cap \mathcal{D}_{[w']}(v)| \neq \emptyset$ then
            AddArc($[u'], o, [w']$)
    else if $pre_o(v)$ undefined then
        AddArc($[t'], o, [w']$)
        AddArc($[u'], o, [w']$)
    else if $pre_o(v) \in \mathcal{D}_{[u']}(v)$ then
        AddArc($[u'], o, [w']$)
    else
        AddArc($[t'], o, [w']$)
```

Algorithm 4.4 The refinement in progress splits state $[s]$ for variable v. This pseudocode adds new loops for and transitions between the new states $[t']$ and $[u']$ for the old loop $([s], o, [s])$ where necessary.

```
procedure UpdateLoop($([s] \in S', o \in \mathcal{O}), v \in V$)
    $[t'], [u'] \leftarrow$ GetChildren($[s]$)
    if $pre_o(v)$ undefined then
        if $post_o(v)$ undefined then
            AddLoop($[t'], o$)
            AddLoop($[u'], o$)
        else if $post_o(v) \in \mathcal{D}_{[u']}(v)$ then
            AddArc($[t'], o, [u']$)
            AddLoop($[u'], o$)
        else
            AddLoop($[t'], o$)
            AddArc($[u'], o, [t']$)
    else if $pre_o(v) \in \mathcal{D}_{[u']}(v)$ then
        if $post_o(v) \in \mathcal{D}_{[u']}(v)$ then
            AddLoop($[u'], o$)
        else
            AddArc($[u'], o, [t']$)
    else if $post_o(v) \in \mathcal{D}_{[u']}(v)$ then
        AddArc($[t'], o, [u']$)
    else
        AddLoop($[t'], o$)

procedure AddLoop($[a'] \in S', o \in \mathcal{O}$)
    $T \leftarrow T \cup \{([a'], o, [a'])\}$
```

For this purpose we split up the code for updating the transition system and use a specialized procedure for updating each of the three transition types: UpdateIncomingArc (Figure 4.2), UpdateOutgoingArc (Figure 4.3) and UpdateLoop (Figure 4.4). They require a lot more code to handle all the possible cases, but lead to a 20% reduction of execution time on average while some domains even only require half the time after the specialization.

Cascaded refinements We noted above that we reuse previous solutions if they remain valid after the refinement, i. e., the solution path is still a legal path through the abstract transition system after we substitute the split state $[s]$ with one of its child states $[t']$ or $[u']$. When we revisited our implementation we noticed that this approach was a good start, but that the idea could be improved. To this end, we let our algorithm recognize if a solution remains valid already during the refinement, and split the previous state in the solution in the same way as we split $[s]$ if it does. With this pattern we work our way backwards from the state in which the flaw occurred towards the abstract initial state, until a refinement breaks the solution. This saves us some time, because the refinement would have had to be made in the next round anyway.

Keep transitions Since we store the incoming and outgoing transitions of each state as an unordered vector, it is cheap to insert, but costly to remove a transition, because the time it takes to find a specific one is linear in the number of transitions. We tried to keep the vectors sorted, but experienced longer processing times instead of the desired speedup. Therefore, we employ a common technique and never delete invalid transitions, but only check whether they are still valid when accessing them. This results in a 1.15-fold speedup over the previous version of the code on average across all domains, while the amount of memory needed for building the abstraction rises by 49%. We accept the higher memory usage because the change allows us to build abstractions for domains with many transitions much faster, resulting in a big increase in the number of finished abstractions.

Intersection hash table The functions FindOptimalTrace and Refine are responsible for most of the processing time in the refinement loop (Alg. 3.1). While the time spent looking for optimal solutions increases with more abstract states inevitably, we can make the REFINE function faster by caching the results of its most expensive operation, the intersection of abstract domains. For this purpose we save whether the sets $\mathcal{D}_{[s']}(v)$ and $\mathcal{D}_{[t']}(v)$ intersect for two states $[s']$, $[t']$ and variable v in a hash table. This memoization saves us about $1/5$ of the time across all domains compared to the previous version of the code while the peak memory usage remains almost constant.

Heuristic updates As noted above we use A* search for finding the next abstract solution. When extracting a solution we update the goal distances of the states on the solution path, but leave the goal distances of other nodes untouched, because a complete backwards search from the goal node with Dijkstra's algorithm is too expensive to be performed after each refinement. Consequently, we allow specifying how often the goal distances of all states should be updated. While all previous code versions did an update every 3000 steps, experiments showed that updating the heuristic values every 1000 steps yields the best ratio of extra processing time versus informedness during search. Table 4.1 shows that this change only has a small impact on the time needed to build the abstraction. However, the effect is more noticeable once the abstractions get bigger and the searches need better heuristic estimates for a fast execution. Regardless of the parameter setting we update all abstract goal distances once again after the refinement finishes.

Summary In total the code optimizations decrease the time it takes to build an abstraction with 10 000 states by a factor of about 14 across all domains and around 33 on domains that benefit the most from the changes. These numbers would be even higher if we included more complex tasks in the comparison which are not accounted for due to the small time limit of 5 minutes for the refinement. Table 4.3 demonstrates the effectiveness of the changes convincingly: the original implementation manages to build abstractions for 13 of the 30 tasks in the TPP domain whereas the final version finishes them for the whole domain within the 5 minute time limit.

	Original	**Reuse solutions**	**A* search**	**Loops vector**	**Single bitset**	**Transition check**	**Cascaded refinements**	**Keep transitions**	**Hash table**	**h updates**
#01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
#02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
#03	**0.00**	**0.00**	0.01	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
#04	0.03	0.02	0.02	0.02	0.01	**0.00**	**0.00**	**0.00**	**0.00**	0.01
#05	36.63	25.62	2.90	2.73	**1.88**	2.05	2.06	2.58	2.43	2.17
#06	63.12	52.49	3.90	3.45	2.55	**2.08**	2.43	2.64	2.47	2.40
#07	73.06	41.20	4.89	3.91	3.12	2.64	**2.53**	2.78	2.76	2.58
#08	71.01	53.23	4.54	3.93	3.02	**2.49**	2.55	3.66	3.71	3.03
#09	90.95	82.64	8.08	4.72	4.05	3.03	2.88	2.76	**2.46**	2.58
#10	102.74	86.05	8.58	5.85	4.74	2.58	**2.39**	2.79	2.68	2.87
#11	126.35	123.09	17.58	7.06	6.73	4.63	4.39	5.18	4.47	**3.96**
#12	148.89	114.47	9.29	5.82	5.79	3.41	3.38	3.41	**3.08**	3.10
#13	299.70	195.21	19.90	10.82	11.24	5.10	10.74	4.75	4.03	**3.88**
#14	–	265.92	33.66	11.16	12.37	5.70	5.34	6.38	5.81	**4.43**
#15	–	–	25.43	15.05	14.69	6.09	8.94	5.71	4.98	**4.32**
#16	–	–	58.54	25.90	30.09	7.10	6.68	7.18	6.98	**5.64**
#17	–	–	77.51	27.96	37.66	8.95	8.44	9.41	7.56	**6.20**
#18	–	–	107.87	63.93	77.63	9.54	12.09	9.90	7.67	**6.56**
#19	–	–	90.18	45.86	59.03	8.84	10.25	9.76	8.40	**7.06**
#20	–	–	82.87	55.14	59.36	8.93	10.42	8.80	**6.69**	6.87
#21	–	–	157.41	62.04	83.85	12.04	13.29	10.29	8.47	**7.21**
#22	–	–	276.97	97.66	119.26	28.87	35.30	11.49	9.35	**7.98**
#23	–	–	259.46	118.25	137.26	17.20	18.66	13.81	10.16	**9.07**
#24	–	–	172.49	85.88	117.29	17.64	18.22	12.97	10.57	**9.19**
#25	–	–	291.03	126.71	159.28	35.91	43.56	13.36	**11.11**	12.21
#26	–	–	–	265.98	–	30.71	48.43	20.31	**15.38**	21.38
#27	–	–	–	170.05	214.58	34.88	55.55	17.03	14.56	**14.05**
#28	–	–	–	189.22	272.91	34.08	52.03	27.10	22.09	**16.06**
#29	–	–	–	–	–	53.44	55.67	28.06	21.97	**19.18**
#30	–	–	–	–	–	82.04	87.76	33.93	**20.78**	22.45

Table 4.3: Time in seconds for building an abstraction with at most 10 000 states in the TPP domain for the different code versions. Best values are highlighted in bold for non-trivial tasks.

4.3 Random starts

Due to the fact that the algorithm always starts finding and breaking the next solution from the abstract initial state, the heuristic tends to have a good estimate of the solution cost for s_0 and states close to it, whereas it might not be as informed for states closer to the goal. This let us to try finding arbitrary concrete states with a random walk from the initial state and starting the search from there. Experiments showed however, that always starting from the initial state leads to better heuristic estimates than alternatingly starting from a random and the initial state or always starting from random states.

4.4 Variable selection strategies

As noted in Chapter 3 sometimes there may be multiple variables that we could potentially use for splitting an abstract state. In this case our algorithm calls the function ChooseVariable and passes it the abstract state $[s]$ that will be split and a list of m distinct candidate variables $\{v_i \mid 1 \leq i \leq m\}$ adhering to the task's arbitrary variable ordering. The domain partitions each variable's abstract domain should be split into are not taken into account for the decision. We implemented the following selection methods:

- **Random** selects an arbitrary variable.
- **First** selects v_1.
- **Goal** and **no-goal** select variable v_i with the smallest index i for which $s_\star(v_i)$ is defined/undefined. If there is no such variable, the methods selects a variable v_i randomly.
- **Min-** and **max-constrained** select the variable v_i with the largest/smallest set of remaining values, i. e., $\arg\max/\min_{v_i} |\mathcal{D}_{[s]}(v_i)|$.
- **Min-** and **max-refined** select the variable that has been refined the least/most in $[s]$, i. e. $\arg\max/\min_{v_i} \frac{|\mathcal{D}_{[s]}(v_i)|}{|\mathcal{D}(v_i)|}$.

- **Min-** and **max-predecessors** make the decision based on an ordering of the variables in the task's causal graph (Helmert 2004). We create this ordering by iteratively appending the variable with the smallest number of incoming edges to the ordering and deleting it and its adjacent edges from the causal graph. We break ties by selecting nodes with higher numbers of outgoing edges before nodes with less successors. The strategies min- and max-predecessors use this ordering by selecting the variable v_i that occurs first/last in the list.

In order to avoid overfitting our implementation for a single ChooseVariable method, we conducted the experiments judging our code base with the *random* strategy. Although regardless of which split is chosen, the refinement loop will advance, the choice has a noticeable influence on the quality of the resulting abstraction. Table 4.4 shows that the resulting values for $h(s_0)$ are quite different for the individual ChooseVariable strategies. From the results it becomes clear that any principled way of selecting a variable is better than a random approach while the *max-refined* strategy yields the best estimates on average. Apart from the *random* and *min-refined* policies each strategy has a better estimate of the solution cost than the others on at least one domain, so it seems that some planning domains favor a specific selection method as shown in Figure 4.2. The plots exhibit the typical appearance of the graphs for some domains where one strategy significantly dominates the others.

Interestingly, the inherent symmetry in the Gripper domain (McDermott 2000) entails that the choice of the variable does not influence $h(s_0)$ there. However, the resulting abstractions are nonetheless different as can be seen in Table 4.5 showing the relative number of expansions. Even on domains for which the different selection methods had very similar $h(s_0)$ values, the number of expanded nodes during search varies greatly in between the individual strategies. Again, *max-refined* yields the best values overall.

$h(0)$ **relative**	**rand**	**first**	**goal**	**no goal**	**min con**	**max con**	**min ref**	**max ref**	**min pre**	**max pre**
airport (33)	1.18	1.38	1.83	1.05	2.11	1.33	1.29	1.71	1.31	**2.30**
blocks (35)	1.03	1.04	1.19	1.02	**1.22**	1.04	1.03	1.10	1.17	1.08
depot (21)	1.08	1.04	1.32	1.07	**1.36**	1.03	1.05	1.30	1.27	1.24
driverlog (20)	1.12	**1.27**	1.25	1.11	1.08	1.24	1.06	1.21	1.21	1.15
elevators-08 (30)	1.18	**2.54**	2.03	1.55	1.97	1.07	1.27	2.05	2.21	1.76
freecell (78)	1.07	1.07	1.14	1.10	1.08	**1.22**	1.09	1.17	1.19	1.08
grid (5)	1.32	1.31	1.21	1.31	1.18	1.38	1.18	**1.45**	1.31	1.01
gripper (20)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
logistics-00 (28)	1.12	1.05	1.17	1.11	1.16	1.05	1.04	1.17	1.05	**1.25**
logistics-98 (35)	1.19	1.09	**2.00**	1.14	1.94	1.09	1.08	1.98	1.09	1.89
miconic (150)	1.02	**1.03**	1.02	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**
mprime (32)	1.08	1.20	1.15	1.16	1.15	1.05	1.11	**1.31**	1.05	1.17
mystery (23)	1.09	1.26	1.20	1.15	1.21	1.00	1.14	**1.33**	1.14	1.17
openstacks-06 (30)	1.08	**1.20**	1.18	1.04	**1.20**	1.07	1.19	1.19	1.12	1.12
openstacks-08 (30)	1.16	1.20	1.25	1.18	**1.26**	1.20	1.21	1.19	1.00	**1.26**
parcprinter-08 (30)	1.16	1.54	1.49	1.23	1.54	1.52	1.52	1.55	**1.61**	1.35
pathways (30)	1.06	1.16	1.09	**1.18**	1.16	1.16	1.15	1.16	1.13	1.13
pegsol-08 (30)	1.01	1.02	1.02	1.16	**1.23**	1.02	1.02	1.21	1.03	1.12
pipesworld-nt (50)	1.10	1.18	**1.20**	1.08	1.14	1.19	1.18	1.19	1.06	1.13
pipesworld-t (35)	1.08	1.18	1.20	1.12	1.19	1.14	1.14	**1.27**	1.14	1.16
psr-small (50)	1.02	1.02	**1.03**	1.01	1.02	1.01	1.00	**1.03**	1.02	**1.03**
rovers (35)	1.33	1.12	1.23	1.09	1.12	1.32	1.23	1.13	1.04	**1.66**
satellite (34)	1.03	1.33	1.25	1.36	1.28	**1.56**	1.34	1.32	1.21	1.43
scanalyzer-08 (30)	1.20	1.01	1.01	1.20	1.03	1.22	1.04	1.02	1.06	**1.29**
sokoban-08 (30)	1.10	1.04	1.11	1.04	1.03	**1.26**	1.11	1.03	1.14	1.11
tpp (30)	1.14	1.04	1.19	1.03	1.10	**1.27**	1.09	1.19	1.09	1.24
transport-08 (30)	1.06	2.18	1.45	1.77	1.58	1.04	1.27	1.98	**2.19**	1.35
trucks (30)	1.04	1.02	**1.05**	1.02	1.02	**1.05**	1.04	1.03	1.03	**1.05**
wood-08 (30)	1.10	1.25	1.27	1.10	1.20	1.17	1.24	1.27	**1.31**	1.05
zenotravel (20)	1.07	1.09	1.23	1.03	1.14	1.11	1.08	1.22	1.09	**1.28**
Geom. mean (1064)	1.10	1.20	1.24	1.14	1.23	1.15	1.14	**1.27**	1.18	1.24

Table 4.4: $h(s_0)$ of each strategy relative to the lowest $h(s_0)$ of all strategies. Each number is the geometric mean of all such measures for a given domain. In turn the last row reports the geometric mean over all domain aggregates.

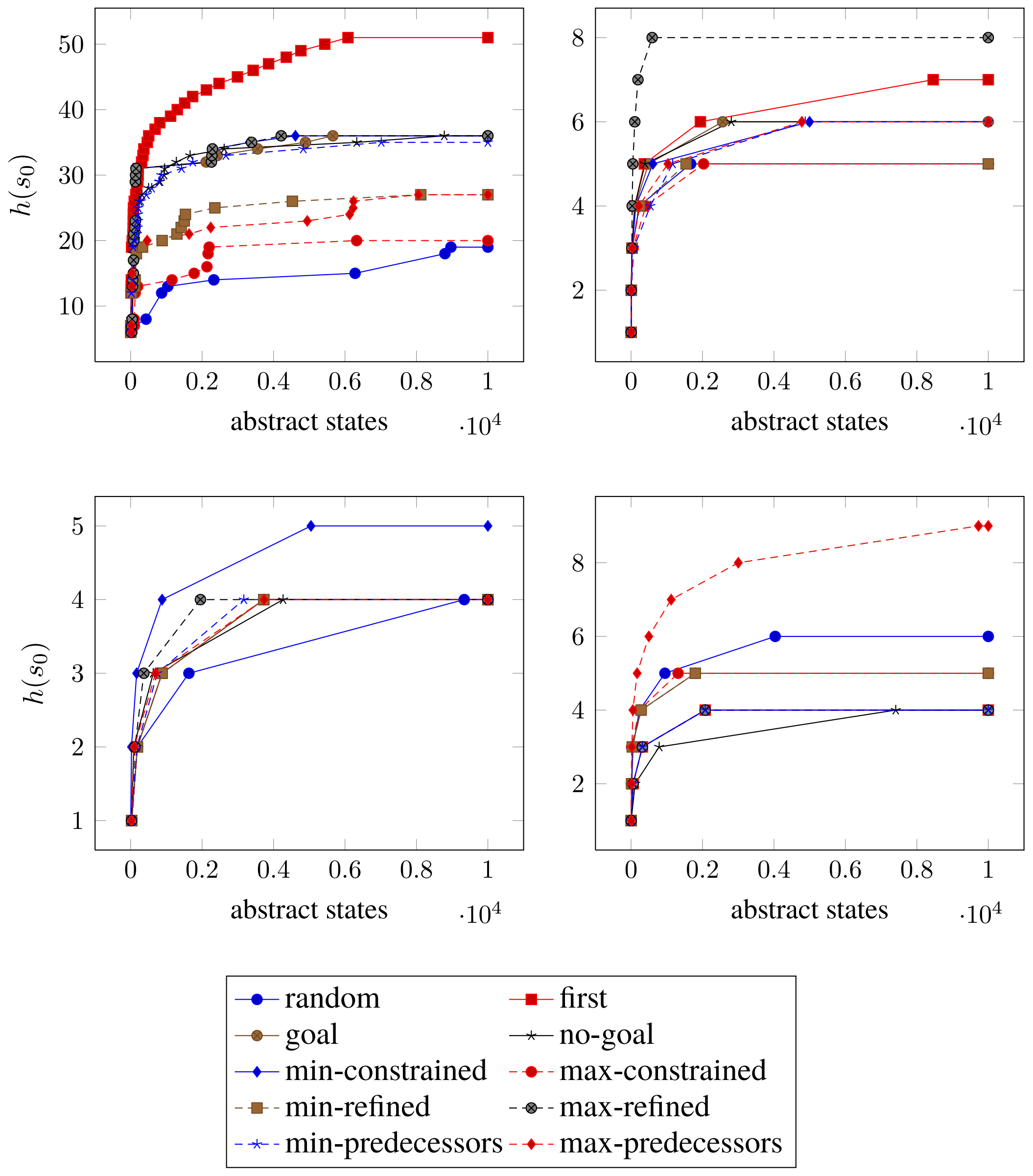


Figure 4.2: $h(s_0)$ subject to an increasing number of abstract states for different ChooseVariable strategies on four example tasks. From left to right and top to bottom the tasks are elevators-08 #23, mprime #24, pegsol-08 #12 and rovers #38.

Relative expansions	**rand**	**first**	**goal**	**no goal**	**min con**	**max con**	**min ref**	**max ref**	**min pre**	**max pre**
airport (19)	2.15	2.19	**1.20**	2.28	1.28	2.16	2.29	1.67	2.10	1.33
blocks (18)	1.93	2.05	1.40	2.01	**1.27**	2.01	2.20	1.69	1.50	1.60
depot (4)	1.55	1.51	**1.03**	1.51	1.35	1.58	1.54	1.09	1.07	1.10
driverlog (9)	5.89	**1.80**	2.12	4.72	5.57	5.18	2.71	1.98	4.58	2.31
elevators-08 (11)	19.69	**1.46**	6.58	13.18	8.27	23.36	18.78	7.03	6.87	4.57
freecell (15)	1.63	1.67	1.69	1.90	1.73	**1.37**	1.87	1.39	1.41	1.42
grid (2)	4.64	3.81	2.66	6.18	9.86	3.57	11.46	**1.00**	5.80	11.35
gripper (7)	1.21	1.29	1.28	1.22	1.28	1.28	1.28	1.29	1.28	**1.00**
logistics-00 (11)	2.63	2.33	2.05	2.92	2.17	2.33	2.83	1.81	2.33	**1.38**
logistics-98 (2)	485.7	1132.0	2.95	809.8	2.45	1170.3	789.2	2.84	1132.0	**1.00**
miconic (50)	1.43	1.40	1.47	1.41	1.40	**1.32**	1.38	1.40	1.40	1.34
mprime (21)	14.82	3.97	3.32	10.89	3.72	31.34	14.76	**1.15**	8.52	6.38
mystery (16)	7.33	2.25	2.83	5.29	4.16	23.12	6.64	**1.36**	6.82	3.05
openstacks-06 (7)	3.89	3.89	2.20	2.32	3.98	2.31	2.80	3.89	2.99	**1.38**
openstacks-08 (18)	1.04	1.04	1.07	1.05	1.05	1.04	1.04	**1.02**	1.06	1.03
parcprinter-08 (11)	2.46	2.23	2.55	1.79	2.28	2.32	2.31	2.24	2.22	**1.50**
pathways (4)	5.40	2.71	5.42	**1.78**	2.71	4.69	2.71	2.71	3.27	4.55
pegsol-08 (27)	2.01	1.63	1.63	1.48	**1.15**	1.63	1.63	1.29	1.84	1.68
pipesworld-nt (14)	1.73	1.42	1.51	1.92	**1.41**	**1.41**	1.42	1.42	2.07	2.42
pipesworld-t (11)	6.46	3.38	3.14	4.49	**1.89**	4.49	3.21	2.41	2.74	2.27
psr-small (49)	1.94	1.63	1.48	1.90	1.65	1.89	2.12	1.51	**1.42**	1.56
rovers (6)	1.92	1.93	1.79	1.87	1.64	1.88	2.12	1.74	1.87	**1.62**
satellite (6)	2.77	1.32	1.86	2.38	1.53	1.36	**1.30**	1.34	1.59	2.88
scanalyzer-08 (12)	1.07	1.05	1.05	1.07	1.05	1.05	1.05	1.05	1.05	**1.00**
sokoban-08 (20)	1.19	1.29	1.19	1.29	1.24	1.10	**1.09**	1.28	1.12	1.19
tpp (6)	2.74	**1.28**	2.49	3.10	**1.28**	**1.28**	**1.28**	**1.28**	**1.28**	2.24
transport-08 (11)	2.34	**1.36**	1.85	1.89	2.17	2.63	1.94	1.52	1.38	2.30
trucks (6)	2.39	5.79	4.53	2.87	5.31	5.09	5.01	3.82	6.23	**1.34**
wood-08 (7)	7.37	11.28	10.27	9.42	5.07	11.68	9.09	9.70	**1.83**	8.30
zenotravel (8)	3.21	2.77	1.83	4.67	2.61	2.72	2.94	1.97	2.73	**1.13**
Geom. mean (408)	3.39	2.55	2.13	3.16	2.21	3.43	3.16	**1.82**	2.75	1.97

Table 4.5: Number of expansions of each strategy relative to the lowest number of expansions needed by any of the strategies. Each number is the geometric mean of all such measures for a given domain. In turn the last row reports the geometric mean over all domain aggregates.

Chapter 5

Comparison to similar heuristics

In this chapter we compare CEGAR abstractions with state-of-the-art abstraction heuristics already implemented in the Fast Downward planning framework: h^{iPDB} (Haslum et al. 2007; Sievers, Ortlieb, and Helmert 2012) and the two $h^{\mathrm{m\&s}}$ configurations of IPC 2011 (Nissim, Hoffmann, and Helmert 2011). For this comparison we let our h^{CEGAR} heuristic use *max-refined*, the ChooseVariable strategy with the least number of expansions in the experiment that compares the different selection methods (Table 4.5). We apply a time limit of 30 minutes and a memory limit of 2 GB and let h^{CEGAR} refine for at most 15 minutes.

Table 5.1 shows the number of solved instances for a number of IPC domains. While the total coverage of h^{CEGAR} is not as high as for h^{iPDB} and $h_2^{\mathrm{m\&s}}$, we solve much more tasks than $h_1^{\mathrm{m\&s}}$ and the h^0 (blind) baseline. We remark that h^{CEGAR} is much less optimized than the other abstraction heuristics, some of which have been polished for years. Nevertheless, h^{CEGAR} outperforms them on some domains. In a direct comparison we solve more tasks than h^{iPDB}, $h_1^{\mathrm{m\&s}}$ and $h_2^{\mathrm{m\&s}}$ on 5, 9 and 7 domains. While h^{CEGAR} is never the single worst performer on any domain, the other heuristics often perform even worse than h^0. Only one task is solved by h^0 but not by h^{CEGAR}, while the other heuristics fail to solve 30, 68, 40 tasks solved by h^0. We note that the Mystery domain includes 11 unsolvable tasks. The heuristics h^0, h^{CEGAR}, h^{iPDB}, $h_1^{\mathrm{m\&s}}$ and $h_2^{\mathrm{m\&s}}$ can prove for 3, 7, 6, 3 and 5 tasks that there is no plan for these problems and we add the numbers to the total coverage. Overall, the coverage results show that h^{CEGAR} is more robust than the other approaches.

Coverage	h^0	h^{CEGAR}	h^{iPDB}	$h_1^{\text{m\&s}}$	$h_2^{\text{m\&s}}$
airport (50)	19	19 (13)	20	**22**	15
blocks (35)	18	18 (11)	**28**	**28**	20
depot (22)	4	4 (2)	**7**	**7**	6
driverlog (20)	7	10 (6)	**13**	12	12
elevators-08 (30)	11	16 (2)	**20**	1	12
freecell (80)	14	15 (6)	**20**	16	3
grid (5)	1	2 (1)	**3**	2	**3**
gripper (20)	7	7 (4)	7	7	**20**
logistics-00 (28)	10	14 (10)	**20**	16	**20**
logistics-98 (35)	2	3 (2)	4	4	**5**
miconic (150)	50	55 (40)	45	50	**74**
mprime (35)	19	**27** (23)	22	23	11
mystery (30)	18	**24** (15)	22	19	12
openstacks-08 (30)	**19**	18 (9)	**19**	8	**19**
openstacks (30)	7	7 (5)	7	7	7
parcprinter-08 (30)	10	11 (9)	11	15	**17**
pathways (30)	4	4 (4)	4	4	4
pegsol-08 (30)	27	27 (8)	3	2	**29**
pipesworld-nt (50)	14	15 (8)	**16**	15	8
pipesworld-t (50)	10	12 (5)	**16**	**16**	7
psr-small (50)	49	49 (46)	49	**50**	49
rovers (40)	5	6 (4)	7	6	**8**
satellite (36)	4	6 (4)	6	6	**7**
scanalyzer-08 (30)	12	12 (6)	**13**	6	12
sokoban-08 (30)	19	19 (4)	**28**	3	23
tpp (30)	5	6 (5)	6	6	**7**
transport-08 (30)	11	11 (6)	11	11	11
trucks (30)	6	7 (4)	**8**	6	**8**
woodworking-08 (30)	7	8 (7)	6	**14**	9
zenotravel (20)	8	9 (8)	9	9	**11**
Sum (1116)	397	441 (277)	**450**	391	449

Table 5.1: Number of solved tasks by domain. For h^{CEGAR}, tasks solved during refinement are shown in brackets. Best values are highlighted in bold.

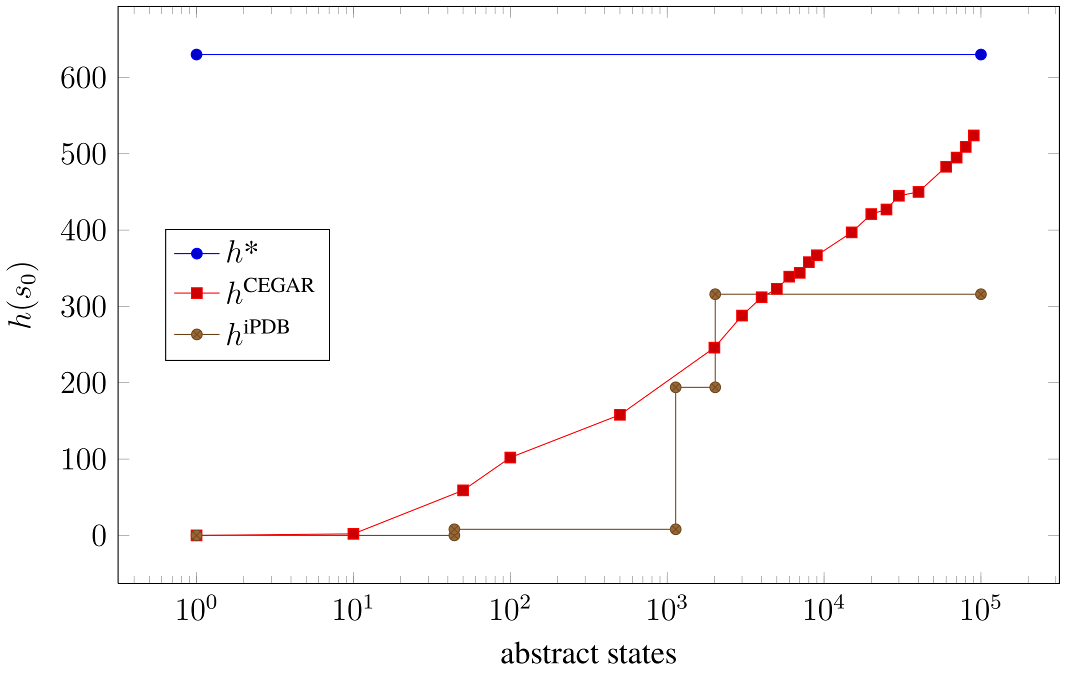


Figure 5.1: Initial state heuristic values for transport-08 #23.

Although h^{CEGAR} typically uses far fewer abstract states, its initial plan cost estimates are often best among all approaches. On commonly solved tasks the estimates are 38%, 134% and 21% higher than those of h^{iPDB}, $h_1^{\text{m\&s}}$ and $h_2^{\text{m\&s}}$ on average when examining the geometric mean of all mean domain ratios. Figure 5.1 shows how the cost estimate for s_0 grows with the number of abstract states on an example task. The h^{CEGAR} estimates are generally higher than those of h^{iPDB} and grow much more smoothly towards the perfect estimate. This behavior can be observed in many domains. Figure 5.2 shows a comparison of initial state estimates made by h^{CEGAR} and h^{iPDB} for a subset of domains.

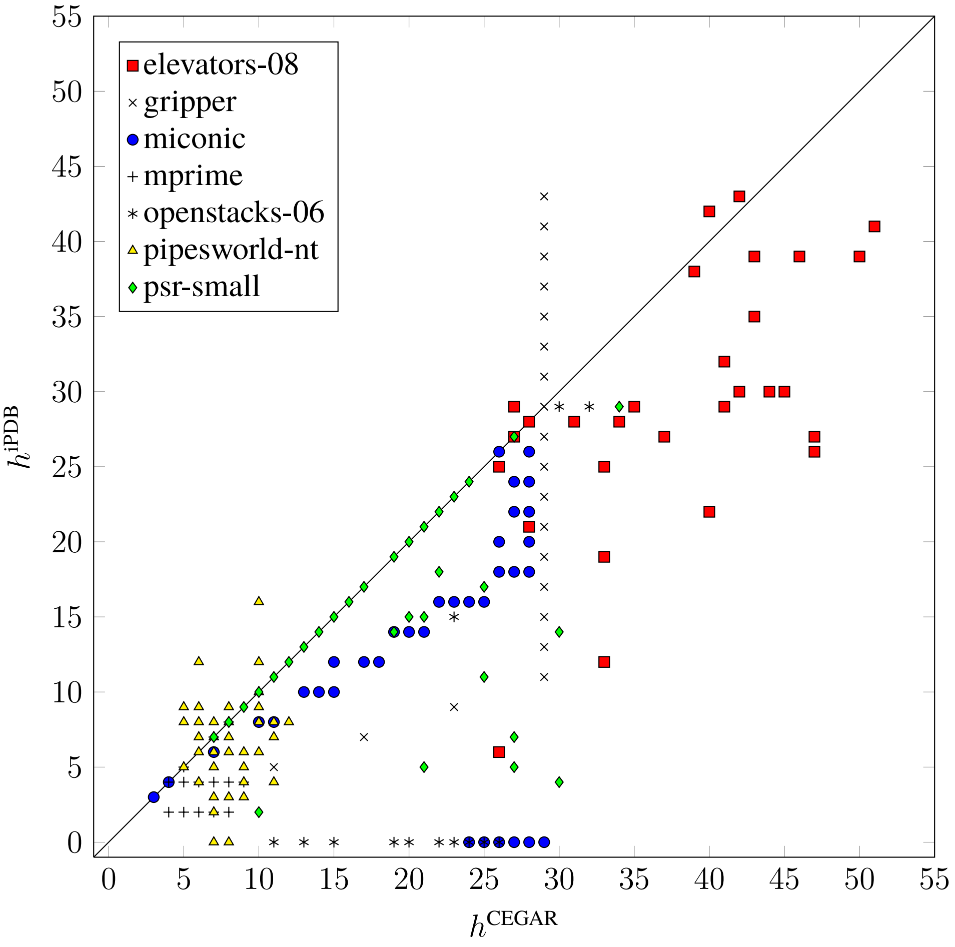


Figure 5.2: $h(s_0)$ for a subset of the domains where h^{CEGAR} has a better estimate of the plan cost than h^{iPDB}.

Chapter 6

Multiple abstractions

The observation from Section 4.4 that typically each domain prefers a specific flaw selection strategy suggests combining multiple abstractions found by different methods in a suitable way. We chose to calculate multiple abstractions and use the maximum of their heuristic estimates for each state during search. In order to evaluate such an approach we order the strategies ascendingly by their respective number of relative expansions in the last experiment (Table 4.5) and form 6 combinations of the best 1, 2, 3, 4, 5 and 6 methods. Afterwards we let each strategy in the combination build an abstraction containing at most 10 000 states and assign a time limit of 30 minutes and a memory limit of 2 GB for the refinement plus the search. Table 6.1 shows the number of solved tasks for each of the resulting heuristics. The biggest improvement results from the combination of the *max-refined* and *max-predecessors* strategies, but the addition of more methods increases the number of solved tasks even more.

As expected, the number of expanded nodes during search continuously decreases when the number of employed strategies increases. Table 6.2 confirms that adding the *max-predecessors* strategy achieves the biggest drop in the number of expansions (41%) while the additional four selection methods need 25% less expansions than the pair of methods.

Coverage	**max-ref**	**max-ref max-pre**	**max-ref max-pre goal**	**max-ref max-pre goal min-con**	**max-ref max-pre goal min-con first**	**max-ref max-pre goal min-con first min-pre**
airport (50)	19	19	19	**21**	**21**	**21**
blocks (35)	18	18	18	18	18	18
depot (22)	4	4	4	4	4	4
driverlog (20)	10	10	10	10	10	10
elevators-08 (30)	16	16	16	16	**19**	**19**
freecell (80)	15	15	15	15	15	15
grid (5)	2	2	2	2	2	2
gripper (20)	7	7	7	7	7	7
logistics-00 (28)	12	**14**	**14**	**14**	**14**	**14**
logistics-98 (35)	3	3	3	3	3	3
miconic (150)	50	**53**	**53**	**53**	**53**	**53**
mprime (35)	**27**	26	26	26	26	25
mystery (30)	**24**	**24**	23	23	22	23
openstacks-06 (30)	7	7	7	7	7	7
openstacks-08 (30)	**19**	**19**	**19**	**19**	**19**	18
parcprinter-08 (30)	11	11	11	11	11	11
pathways (30)	4	4	4	4	4	4
pegsol-08 (30)	27	27	27	27	27	27
pipesworld-nt (50)	15	15	15	15	15	15
pipesworld-t (50)	12	12	12	12	12	12
psr-small (50)	49	49	49	49	49	49
rovers (40)	6	**7**	**7**	**7**	**7**	**7**
satellite (36)	6	6	6	6	6	6
scanalyzer-08 (30)	12	12	12	12	12	12
sokoban-08 (30)	20	**21**	**21**	**21**	20	20
tpp (30)	6	6	6	6	6	6
transport-08 (30)	11	11	11	11	11	11
trucks (30)	6	6	6	6	6	6
wood-08 (30)	7	8	8	8	8	**9**
zenotravel (20)	8	8	8	8	8	8
Sum (1116)	433	440	439	441	**442**	**442**

Table 6.1: Number of solved tasks per domain for different combinations of ChooseVariable strategies. Best values are highlighted in bold.

Expansions relative to max-ref	**max-ref**	**max-ref max-pre**	**max-ref max-pre goal**	**max-ref max-pre goal min-con**	**max-ref max-pre goal min-con first**	**max-ref max-pre goal min-con first min-pre**
airport (19)	1.00	0.74	0.60	0.55	0.55	**0.54**
blocks (18)	1.00	0.69	0.60	0.54	0.53	**0.52**
depot (4)	1.00	0.84	0.78	0.77	0.77	**0.74**
driverlog (10)	1.00	0.45	0.40	0.33	0.34	**0.29**
elevators-08 (16)	1.00	0.41	0.38	0.36	**0.13**	**0.13**
freecell (15)	1.00	0.67	0.60	0.55	0.55	**0.51**
grid (2)	1.00	0.85	0.85	0.83	**0.80**	**0.80**
gripper (7)	1.00	0.68	**0.67**	**0.67**	**0.67**	**0.67**
logistics-00 (12)	1.00	0.34	0.34	0.34	**0.33**	**0.33**
logistics-98 (3)	1.00	0.34	**0.31**	**0.31**	**0.31**	**0.31**
miconic (50)	1.00	**0.58**	**0.58**	**0.58**	**0.58**	**0.58**
mprime (25)	1.00	0.80	0.73	0.72	**0.66**	**0.66**
mystery (17)	1.00	0.82	0.73	0.64	**0.49**	**0.49**
openstacks-06 (7)	1.00	0.41	0.41	**0.39**	**0.39**	**0.39**
openstacks-08 (18)	1.00	0.98	0.98	**0.97**	**0.97**	**0.97**
parcprinter-08 (11)	1.00	0.41	0.41	0.41	0.41	**0.38**
pathways (4)	1.00	0.57	0.55	0.55	0.55	**0.33**
pegsol-08 (27)	1.00	0.91	0.79	0.68	0.68	**0.67**
pipesworld-nt (15)	1.00	0.73	0.57	0.57	0.57	**0.50**
pipesworld-t (12)	1.00	0.46	0.41	0.31	0.31	**0.28**
psr-small (49)	1.00	0.80	0.67	0.61	0.61	**0.54**
rovers (6)	1.00	0.48	0.47	**0.46**	**0.46**	**0.46**
satellite (6)	1.00	0.73	0.68	0.65	0.63	**0.47**
scanalyzer-08 (12)	1.00	0.92	**0.91**	**0.91**	**0.91**	**0.91**
sokoban-08 (20)	1.00	0.92	0.88	0.87	0.87	**0.78**
tpp (6)	1.00	**0.69**	**0.69**	**0.69**	**0.69**	**0.69**
transport-08 (11)	1.00	0.79	0.70	0.67	0.54	**0.52**
trucks (6)	1.00	0.15	**0.14**	**0.14**	**0.14**	**0.14**
wood-08 (7)	1.00	0.42	0.33	0.20	0.20	**0.06**
zenotravel (8)	1.00	0.49	0.47	0.47	**0.46**	**0.46**
Geom. mean (423)	1.00	0.59	0.55	0.51	0.48	**0.44**

Table 6.2: Number of expansions for different combinations of ChooseVariable strategies relative to the number of expansions for the single strategy. Each number is the geometric mean of all such measures for a given domain. In turn the last row reports the geometric mean over all domain aggregates. Best values are highlighted in bold.

Chapter 7

Online learning

As we stated in the introduction, we use A* search with an admissible heuristic for finding optimal plans. Normally, the search function is strictly separated from the heuristic and the latter is used as a black box that returns estimates for given states. In this chapter we demonstrate how we can lift this restriction and actively improve the heuristic during search. We accomplish this by letting the A* algorithm detect when the heuristic makes an error and fix it before we continue searching.

A* maintains a list of states that may be visited next while traversing the state space. For each state s in the so called *open list* the algorithm records a value $f(s) = g(s) + h(s)$ which is the sum of the incumbent cost of reaching this state from s_0, $g(s)$ and the heuristic estimate of its goal distance $h(s)$. Initially, this list contains only the initial state s_0 with $f(s_0) = g(s_0) + h(s_0) = h(s_0)$. Until the list is empty, in each iteration A* removes the state s with the lowest value for $f(s)$ from the list and *expands* it, i. e.. adds all states that can be reached from s by applying a single operator to the open list. The algorithm finishes when a goal state is expanded or when there are no more new states to visit.

An error is made during the expansion of state s if $h(s) < h(t) + cost_o$ for all states t that can be reached by applying a single operator o in s. This follows from the admissibility of the heuristic. If we could already prove that the cost to reach the goal from t is at least $h(t)$, we know that the distance from s to the goal must be at least $h(t)$ plus the cost from s to t, $cost_o$, if going over t is the cheapest path from s.

In case we detect that the heuristic value for state s is too low, we refine the abstraction until its value rises by repeatedly finding and breaking abstract solutions that start from $[s]$.

Analogously to the offline refinement, we allow limiting the number of abstract states in the abstraction. In order to evaluate the effectiveness of online refinements, we compare five different h^{CEGAR} configurations using the *max-refined* selection strategy with at most 10 000 abstract states. The heuristics differ only in the way they divide the number of refinements between the off- and online phase. Table 7.1 shows the number of expansions each configuration uses relative to the one with the lowest number of expanded nodes. According to the results, the usefulness of online refinements varies greatly in between different domains. Only a few domains prefer either only off- or online refinements. In most domains a mix of the two kinds leads to the fewest expansions. Overall, splitting most states offline and using some additional states online seems to be the best strategy. However, a smaller number of expansions does not indicate that more tasks will be solved since all five configurations have roughly the same coverage.

Raising the maximum total number of states in the abstraction to 20 000 gives a clearer picture. As shown in Table 7.2 splitting all states already offline results in the largest number of problems solved and it is clearly not a good strategy to only refine online. This is confirmed when we examine the relative number of expansions in Table 7.3. Again, creating the largest amount of states during offline refinement with a small number of online refinements yields the smallest relative number of expanded nodes across all domains.

Expansions (relative)	**10K-0K**	**8K-2K**	**5K-5K**	**2K-8K**	**0K-10K**
airport (19)	1.07	**1.02**	1.10	1.42	1.58
blocks (18)	1.32	1.19	1.09	**1.08**	1.96
depot (4)	1.19	1.13	1.09	**1.07**	1.34
driverlog (9)	2.16	1.75	1.71	1.96	**1.30**
elevators-08 (12)	1.07	**1.04**	1.05	**1.04**	1.20
freecell (15)	1.28	1.25	**1.10**	1.17	1.18
grid (2)	1.16	1.43	**1.07**	1.56	1.22
gripper (7)	1.13	**1.00**	1.12	1.17	2.57
logistics-00 (12)	1.65	**1.15**	1.70	2.12	4.00
logistics-98 (2)	1.28	1.07	1.23	**1.02**	1.06
miconic (50)	1.18	**1.05**	1.20	1.65	3.93
mprime (22)	3.40	**1.71**	2.26	1.88	1.84
mystery (17)	1.79	1.42	**1.36**	1.47	1.63
openstacks-06 (7)	1.30	**1.08**	2.30	5.58	6.71
openstacks-08 (19)	1.08	1.06	1.05	1.03	**1.02**
parcprinter-08 (10)	1.09	**1.04**	1.10	2.67	5.81
pathways (4)	1.15	1.12	**1.08**	1.09	1.67
pegsol-08 (27)	1.17	1.09	**1.06**	1.12	1.30
pipesworld-nt (14)	1.59	1.39	1.31	**1.30**	1.33
pipesworld-t (11)	1.84	1.67	1.43	1.36	**1.31**
psr-small (49)	1.28	**1.08**	1.14	1.64	3.08
rovers (6)	1.14	1.11	1.10	**1.04**	2.37
satellite (6)	1.27	**1.12**	**1.12**	1.61	4.36
scanalyzer-08 (12)	**1.01**	**1.01**	**1.01**	**1.01**	1.24
sokoban-08 (19)	**1.04**	**1.04**	1.07	1.16	1.46
tpp (6)	**1.04**	1.06	1.14	1.17	2.02
transport-08 (11)	1.12	**1.01**	1.14	1.17	1.59
trucks (6)	**1.15**	1.35	1.94	2.99	4.50
wood-08 (7)	1.99	**1.36**	2.63	2.66	5.31
zenotravel (8)	1.85	**1.73**	2.64	2.26	2.77
Geom. mean (411)	**1.07**	1.55	1.85	2.31	4.05

Table 7.1: Number of expanded nodes relative to the configuration with the least number of expansions. The header shows the maximum number of off- and online refinements for the individual configurations.

Coverage	**20K-0K**	**16K-4K**	**12K-8K**	**4K-16K**	**0K-20K**
airport (50)	19	19	19	19	19
blocks (35)	18	18	18	18	18
depot (22)	4	4	4	4	4
driverlog (20)	10	10	10	10	10
elevators-08 (30)	**13**	**13**	12	12	12
freecell (80)	15	15	15	15	15
grid (5)	2	2	2	2	2
gripper (20)	7	7	7	7	7
logistics-00 (28)	13	13	13	13	13
logistics-98 (35)	2	2	2	2	2
miconic (150)	52	52	52	52	52
mprime (35)	**23**	**23**	**23**	**23**	22
mystery (30)	19	19	19	19	**20**
openstacks-06 (30)	7	7	7	7	7
openstacks-08 (30)	19	19	19	19	19
parcprinter-08 (30)	**11**	**11**	**11**	**11**	10
pathways (30)	4	4	4	4	4
pegsol-08 (30)	27	27	27	27	27
pipesworld-nt (50)	14	14	14	14	14
pipesworld-t (50)	11	11	11	11	11
psr-small (50)	49	49	49	49	49
rovers (40)	6	6	6	6	6
satellite (36)	**6**	**6**	**6**	5	**6**
scanalyzer-08 (30)	12	12	12	12	12
sokoban-08 (30)	**21**	20	**21**	**21**	19
tpp (30)	6	6	6	6	6
transport-08 (30)	11	11	11	11	11
trucks (30)	6	6	6	6	6
wood-08 (30)	**10**	**10**	**10**	**10**	7
zenotravel (20)	8	8	8	8	8
Sum (1116)	**425**	424	424	423	418

Table 7.2: Number of solved tasks for different h^{CEGAR} configurations differing only in the maximum number of states during off- and online refinement.

Expansions (relative)	**20K-0K**	**16K-4K**	**12K-8K**	**4K-16K**	**0K-20K**
airport (19)	1.11	**1.03**	1.05	1.25	1.68
blocks (18)	1.33	**1.06**	1.11	1.15	2.18
depot (4)	1.15	1.10	1.06	**1.02**	1.33
driverlog (10)	2.49	**1.60**	1.67	1.85	1.75
elevators-08 (12)	1.08	1.05	**1.04**	1.06	1.22
freecell (15)	1.71	1.54	**1.17**	1.66	1.77
grid (2)	**1.17**	1.41	1.45	1.42	1.22
gripper (7)	1.03	**1.00**	1.36	1.51	3.45
logistics-00 (13)	**1.31**	2.10	3.01	3.68	6.75
logistics-98 (2)	**1.00**	7.75	14.41	15.96	17.97
miconic (52)	**1.16**	1.27	1.36	2.10	5.40
mprime (22)	2.67	1.49	**1.44**	1.91	1.67
mystery (17)	1.84	1.63	1.57	**1.49**	1.73
openstacks-06 (7)	1.36	1.13	**1.00**	5.34	6.99
openstacks-08 (19)	1.10	1.08	1.07	1.05	**1.03**
parcprinter-08 (10)	1.14	1.11	**1.09**	1.55	6.11
pathways (4)	**1.06**	1.26	1.93	2.82	4.75
pegsol-08 (27)	1.17	**1.06**	1.13	1.15	1.41
pipesworld-nt (14)	2.04	1.49	**1.31**	1.46	1.67
pipesworld-t (11)	1.93	1.92	**1.38**	1.89	1.93
psr-small (49)	1.41	**1.13**	1.20	1.24	3.47
rovers (6)	1.23	**1.05**	**1.05**	1.13	2.37
satellite (5)	1.45	1.33	1.21	**1.00**	6.42
scanalyzer-08 (12)	1.02	**1.01**	1.02	**1.01**	1.26
sokoban-08 (19)	**1.02**	1.05	1.04	1.10	1.47
tpp (6)	1.11	**1.01**	1.02	1.47	2.62
transport-08 (11)	1.14	1.04	**1.02**	1.17	1.65
trucks (6)	1.16	**1.07**	1.08	2.62	3.53
wood-08 (7)	2.33	**1.81**	1.84	2.96	4.70
zenotravel (8)	1.81	**1.55**	1.68	2.49	2.80
Geom. mean (414)	1.36	**1.34**	1.38	1.73	2.60

Table 7.3: Number of expanded nodes relative to the configuration with the minimum number of expansions. The h^{CEGAR} configurations differ only in the maximum number of states created during off- and online refinement.

Chapter 8

Conclusion

We introduced a CEGAR approach for classical planning and showed that it delivers promising performance. We believe that further performance improvements are possible through more space-efficient abstraction representations and speed optimizations in the refinement loop, which will enable larger abstractions to be generated in reasonable time. One way of trying to achieve this is to break not one but all optimal solutions in one iteration. This should shift a big proportion of the time needed to build the abstraction from looking for abstract solutions to actually refining the abstraction.

We showed that using the maximum of the heuristic estimates made by multiple CEGAR abstractions can lead to more informed heuristics. Another promising direction is the exploitation of additive CEGAR abstractions, borrowing one of the major strengths of the h^{iPDB} approach.

Additionally, we demonstrated how we can refine the abstraction online and in this way improve the heuristic while searching.

All in all, we believe that Cartesian abstraction and counterexample-guided abstraction refinement are useful concepts that can contribute to the further development of strong abstraction heuristics for automated planning.

Bibliography

Bäckström, Christer and Bernhard Nebel (1995). “Complexity Results for SAS$^+$ Planning”. In: *Computational Intelligence* 11.4, pp. 625–655.

Ball, Thomas, Andreas Podelski, and Sriram K. Rajamani (2001). “Boolean and Cartesian Abstraction for Model Checking C Programs”. In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, pp. 268–283.

Bylander, Tom (1994). “The Computational Complexity of Propositional STRIPS Planning”. In: *Artificial Intelligence* 69.1–2, pp. 165–204.

Chatterjee, Krishnendu, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar (2005). “Counterexample-guided Planning”. In: *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI 2005)*, pp. 104–111.

Clarke, Edmund M., Orna Grumberg, and Doron A. Peled (2000). *Model Checking*. The MIT Press.

Clarke, Edmund M., Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith (2000). “Counterexample-Guided Abstraction Refinement”. In: *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*. Ed. by E. Allen Emerson and A. Prasad Sistla, pp. 154–169.

Culberson, Joseph C. and Jonathan Schaeffer (1998). “Pattern Databases”. In: *Computational Intelligence* 14.3, pp. 318–334.

Edelkamp, Stefan (2001). “Planning with Pattern Databases”. In: *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*. Ed. by Amedeo Cesta and Daniel Borrajo. Toledo, Spain, pp. 13–24.

Haslum, Patrik (2012). “Incremental Lower Bounds for Additive Cost Planning Problems”. In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. AAAI Press, pp. 74–82.

Haslum, Patrik, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig (2007). "Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning". In: *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*. AAAI Press, pp. 1007–1012.

Helmert, Malte (2004). "A Planning Heuristic Based on Causal Graph Analysis". In: *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*. Ed. by Shlomo Zilberstein, Jana Koehler, and Sven Koenig. AAAI Press, pp. 161–170.

— (2006). "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research* 26, pp. 191–246.

— (2009). "Concise Finite-Domain Representations for PDDL Planning Tasks". In: *Artificial Intelligence* 173, pp. 503–535.

Helmert, Malte, Patrik Haslum, and Jörg Hoffmann (2007). "Flexible Abstraction Heuristics for Optimal Sequential Planning". In: *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*. Ed. by Mark Boddy, Maria Fox, and Sylvie Thiébaux. AAAI Press, pp. 176–183.

Hernádvölgyi, István T. and Robert C. Holte (2000). "Experiments with Automatically Created Memory-Based Heuristics". In: *Proceedings of the 4th International Symposium on Abstraction, Reformulation and Approximation (SARA 2000)*. Ed. by Berthe Y. Choueiry and Toby Walsh. Vol. 1864. Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 281–290.

Keyder, Emil, Jörg Hoffmann, and Patrik Haslum (2012). "Semi-Relaxed Plan Heuristics". In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. AAAI Press, pp. 128–136.

McDermott, Drew (2000). "The 1998 AI Planning Systems Competition". In: *AI Magazine* 21.2, pp. 35–55.

Nissim, Raz, Jörg Hoffmann, and Malte Helmert (2011). "Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning". In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*. Ed. by Toby Walsh, pp. 1983–1990.

Sievers, Silvan, Manuela Ortlieb, and Malte Helmert (2012). "Efficient Implementation of Pattern Database Heuristics for Classical Planning". In: *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SOCS 2012)*. Ed. by Daniel

Borrajo, Ariel Felner, Richard Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan Sturtevant. AAAI Press, pp. 105–111.