University of Basel

Pattern Selection using Counterexample-guided Abstraction Refinement

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
http://ai.cs.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Silvan Sievers

Alexander Rovner
alex.rovner@stud.unibas.ch
2015-050-289

12.08.2018

Acknowledgments

I would like to thank Prof. Dr. Malte Helmert for the opportunity to write this thesis as well as Dr. Silvan Sievers for supervising the project and providing valuable insight and feedback. I also want to thank Dr. Pascal Bercher for his input on the complexity of the plan interleaving problem. Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing center at University of Basel.

Abstract

Pattern Databases are a powerful class of abstraction heuristics which provide admissible path cost estimates by computing exact solution costs for all states of a smaller task. Said task is obtained by abstracting away variables of the original problem. Abstractions with few variables offer weak estimates, while introduction of additional variables is guaranteed to at least double the amount of memory needed for the pattern database. In this thesis, we present a class of algorithms based on counterexample-guided abstraction refinement (CEGAR), which exploit additivity relations of patterns to produce pattern collections from which we can derive heuristics that are both informative and computationally tractable. We show that our algorithms are competitive with already existing pattern generators by comparing their performance on a variety of planning tasks.

Table of Contents

1 Introduction

Consider a problem from the logistics domain (McDermott, 2000). Here the goal is to transport a number of packages by truck or plane between and inside cities. Solving such a task usually boils down to finding an appropriate set of travel instructions, also called a *plan*, whose execution will result in all packages being delivered to their destinations.

All actions, however, are associated with certain costs, as trucks and planes need to be refueled and workers must be paid. For this reason, the goal in *cost-optimal planning* is to find a plan that not only solves the given task, but also does so as cheaply as possible. In practice, the search for a solution implies the need to evaluate many different courses of action until the best plan has been found. In planning, a *state* is defined as a set of variables whose values describe the current situation (e.g. location of packages, trucks and airplanes). All possible states of a task describe a *state space*. For large problems, i.e. problems with many variables that can take on many different values, the state space is so big that it is not feasible to exhaustively evaluate all possibilies.

Heuristic search algorithms alleviate this problem by expanding the most promising states first. To this end, they use heuristic functions which estimate the cost of solving the given task from all states.

The main focus of this thesis are pattern database (PDB) heuristics (Culberson and Schaeffer, 1998) for domain-independent cost-optimal planning. These heuristics provide plan cost estimates for the original problem by computing the exact plan cost for subproblems. Each subproblem is induced by a *pattern*, which determines which variables from the original problem are to be considered part of the subtask and which are to be discarded.

Because PDBs must compute cost estimates for the entire state space of a subproblem, it is usually only feasible to generate PDBs for small subtasks, which lack many details of the original problem. Due to this lack of knowledge, resulting PDBs tend to greatly underestimate the real plan cost.

To address this issue, modern planners work with multiple different PDBs that are combined into a single informative heuristic using cost partitioning. For a detailed overview of various cost partitioning techniques, we refer to Seipp et al. (2017). In this thesis, we are interested in combining PDBs by adding together their individual estimates whenever possible. Since PDBs are generated from subtasks which, in turn, are induced by patterns, the choice of a good pattern collection is essential for the construction of an accurate heuristic.

For this purpose Haslum et al. (2007) introduced the *iPDB* algorithm, which starts with a set of small patterns with often uninformative corresponding PDB heuristics and uses *hillclimbing* to incrementally improve the pattern collection. Heuristics derived from this collection are combined in the *canonical heuristic function*, which identifies subsets of *additive* patterns, adds their corresponding PDB estimates and returns the largest estimate that resulted from the aforementioned summation over additive subsets. Because the nature of local search precludes some patterns from being reached by iPDB (namely patterns that are not immediate improvements), Pommerening et al. (2013) presented an algorithm that systematically generates all *interesting* patterns up to a certain size.

Counterexample-guided abstraction refinement (CEGAR) (Clarke et al., 2000) is a general technique used in model checking to refine abstractions only in necessary places. The idea is to check an abstraction against the *concrete planning problem* and indentify critical information that the abstraction is missing and only then refine the abstraction with said missing knowledge. In planning, CEGAR was impelemented for *Cartesian abstractions* by Seipp and Helmert (2013).

In this thesis, we apply the CEGAR principle to the generation of pattern collections. Specifically, we present two CEGAR algorithms in Chapter 3: $CEGAR^{fadd}$ generates pattern collections where all patterns are pairwise additive, so that their corresponding PDB estimates can always be added together. $CEGAR^{nadd}$ on the other hand, does not force additivity relations between patterns and relies on the canonical heuristic for combining resulting patterns.

Unlike Cartesian abstractions, PDBs do not allow for fine-grained refinement. A single pattern refinement step is guaranteed to at least double the size of the corresponding PDB. For this reason, CEGAR algorithms for PDBs can run into memory issues. In Chapter 4, we discuss three techniques for addressing this problem. In Chapter 5, we present the results of experimental evaluation of our algorithms and compare them to iPDB.

2 Background

In this section we present underlying concepts that are crucial for understanding the algorithms in Chapter 3.

2.1 Planning Tasks

Planning tasks discussed in this thesis will be expressed in the SAS$^+$ planning formalism (Bäckström and Nebel, 1995). A SAS$^+$ task Π is given by a 4-tuple $\langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ with the following components:

- $\mathcal{V}$: finite set of variables of the planning task. Each variable $v \in \mathcal{V}$ is associated with a finite *domain* $\mathcal{D}_v$. A *partial assignment* $s : \mathit{Vars}(s) \rightarrow \bigcup_{v \in \mathit{Vars}(s)} \mathcal{D}_v$ is a function that maps variables $v \in \mathit{Vars}(s) \subseteq \mathcal{V}$ to values of their respective domains. Furthermore, we define $s[v]$ as the value of variable $v \in \mathit{Vars}(s)$ in partial assignment s. Two partial assignments s, s' are said to *agree* iff $s[v] = s'[v]$ for all variables $v \in \mathit{Vars}(s) \cap \mathit{Vars}(s')$.
- $\mathcal{I}$: initial assignment of the planning task that describes initial values of all variables, i.e. $\mathit{Vars}(\mathcal{I}) = \mathcal{V}$
- $\mathcal{G}$: partial assignment that defines the goal of the planning task. Variables $v \in \mathit{Vars}(\mathcal{G})$ will be referred to as *goal variables* in cases where this distinction helps to clarify concepts.
- $\mathcal{A}$: set of actions where each action $a \in \mathcal{A}$ consists of:
  - partial assignment $\mathit{pre}(a)$ that defines *preconditions* of a.
  - *effects* $\mathit{eff}(a)$, which are also given as a partial assignment.
  - $\mathit{cost}(a) \in \mathbb{R}_0^+$: the cost of performing action a.

The SAS$^+$ planning task induces the state-space $\mathcal{S}(\Pi) = \langle S, A, \mathit{cost}, T, s_0, S_* \rangle$ with

- $S = \mathcal{D}_{v_1} \times \mathcal{D}_{v_2} \times ... \times \mathcal{D}_{v_n}$ with $v_i \in \mathcal{V}$: Set of all possible assignments s with $\mathit{Vars}(s) = \mathcal{V}$. These total assignments are called *states*.

- A: set of all actions as defined in Π.
- *cost*: cost of performing an action
- $T \subseteq S \times A \times S$: set of transitions of the form $s \xrightarrow{a} s'$ with $s, s' \in S$ and a is applicable in s. An action a is *applicable* in state s iff $pre(a)$ and s agree. The state s' with $s'[v] = s[v]$ for all $v \notin Vars(eff(a))$ that agrees with $eff(a)$ is called a *successor* of s.
- s_0: initial state. Corresponds to the initial variable assignment $\mathcal{I}$.
- $S_* \subseteq S$: set of goal states: states which agree with the goal assignment $\mathcal{G}$.

A sequence of transitions is called a *path*. The cost of a path is the sum of costs of its actions. A path which leads from the initial state to some goal state $s_* \in S_*$, is called a *plan*. In cost-optimal planning, the goal is to find a plan with minimal total cost.

2.2 Heuristic Functions

The problem of finding a plan for the given planning task Π, is equivalent to that of finding a path from the initial state to some goal state in $\mathcal{S}(\Pi)$. While it is possible to find the optimal plan by performing an exhaustive search over the entire state-space with, e.g. Dijkstra's Algorithm (Dijkstra, 1959), this approach is infeasible for large state-spaces.

Efficient search algorithms like A* (Hart et al., 1968) or IDA* (Korf, 1985) use *heuristic functions* to identify and expand only promising successor states. Let S be the set of states of $\mathcal{S}(\Pi)$ of some planning task Π, then a heuristic is a function $h : S \to \mathbb{R}_0^+$. Such a function provides the estimated cost of reaching a goal from any given state of the state-space. The perfect heuristic h^* is a function that always returns the exact plan cost for all states. A heuristic h that never overestimates the exact cost, i.e. $h(s) \leq h^*(s)$ for all s, is called *admissible*.

Admissible heuristics are an important class of heuristics in cost-optimal planning, since plans found by A* equipped with such a heuristic are guaranteed to be optimal. The amount of time and memory that is needed to find an optimal solution with A* search largely depends on the quality of the estimates generated by the heuristic. Given two admissible heuristics h_1 and h_2, we say that h_2 *dominates* h_1 if $h_1(s) \leq h_2(s)$ for all $s \in S$. This means that cost estimates provided by h_1 are never better than those returned by h_2.

In most cases, the more precise the estimator used, the fewer search node expansions will A* need to find a solution. Therefore, we are generally interested in finding the most dominant admissible heuristic that can still be realistically computed under the given time and memory constraints.

2.3 Pattern Database Heuristics

One way of obtaining admissible heuristics is to use pattern abstractions. Here, the goal is to reduce the original planning task to a smaller task by abstracting away variables of the original problem. Since the number of states in a state-space is given as

$$|S| = |\mathcal{D}_{v_1} \times \mathcal{D}_{v_2} \times ... \times \mathcal{D}_{v_n}| = |\mathcal{D}_{v_1}| \cdot |\mathcal{D}_{v_2}| \cdot ... \cdot |\mathcal{D}_{v_n}| \tag{2.1}$$

omitting even a single variable will result in an abstract task which induces a significantly smaller state-space. For example, removing only a single variable with a binary domain will lead to a state-space with 50% fewer states than the original.

The goal is to reduce the number of states to the point where it becomes feasible to compute the perfect heuristic for all states of the abstract task and store these values in a look-up table in memory. Said look-up tables are called *pattern databases (PDB)* and we can use their entries as path-cost estimates when solving the original problem.

Given a *concrete task* $\Pi = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ let $P \subseteq \mathcal{V}$ be a subset of its variables. The *abstract task* $\Pi^P = \langle \mathcal{V}^P, \mathcal{I}^P, \mathcal{G}^P, \mathcal{A}^P \rangle$ induced by P is constructed by setting $\mathcal{V}^P := P$ and discarding from $\mathcal{I}$, $\mathcal{G}$ as well as from *pre*(a) and *eff*(a) of all $a \in \mathcal{A}$ all variables that are not included in P. The set P which induces this projection from Π to Π^P is called a *pattern*.

A PDB for Π^P is constructed by traversing the entire *abstract state space* $\mathcal{S}(\Pi^P)$ backwards (i.e. starting with the goal states) using Dijkstra's Algorithm, which computes the optimal path-cost for all *abstract states* s from $\mathcal{S}(\Pi^P)$. The resulting PDB heuristic h^P is defined as the perfect heuristic h^* of Π^P.

In order to obtain a PDB heuristic value for some state $s \in S$ of the concrete task, a perfect hash function is used. This function maps the concrete state to the index under which the estimate for the corresponding abstract state is saved. For a detailed description of how pattern databases are implemented in practice, we refer to Sievers et al. (2012).

PDB heuristics are guaranteed to be admissible for the concrete task because pattern database abstractions preserve paths of the concrete state-space, i.e. transitions that exist in $\mathcal{S}(\Pi)$ also exist $\mathcal{S}(\Pi^P)$. However, due to some action preconditions being abstracted away, new transitions may become possible in the abstract task, thus lowering the optimal plan cost for some states in $\mathcal{S}(\Pi^P)$. Therefore, in summary, PDB estimates are always equal to or lower than the perfect heuristic of the concrete task.

2.3.1 Additivity of Patterns

Different patterns result in different abstractions, from which, in turn, different pattern databases are generated. Therefore, the choice of a pattern also influences the quality of the resulting heuristic. Small patterns result in heuristics that offer weak estimates, while large patterns frequently lead to pattern databases that are prohibitively expensive to compute and store in memory. We are interested in circumventing this dilemma by combining several small and therefore computationally inexpensive heuristic functions into a single admissible heuristic that dominates the individual functions it is composed of.

A possible approach is to take the maximum over all heuristics, i.e. given n admissible heuristics $h_1, h_2, ..., h_n$, we combine them as $h^{max}(s) := \max\{h_1(s), h_2(s), ..., h_n(s)\}$. Here, h^{max} clearly dominates the individual heuristics.

We can obtain an even stronger heuristic by adding the estimates of multiple heuristics. Addition, however, only results in an admissible heuristic if the individual heuristics are *additive* with each other. To define additivity for patterns, we first define what it means for two variables of a planning task to be *correlated*:

Let $\Pi = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ be a planning task. Two variables $v_i, v_j \in \mathcal{V}$ are correlated if one of the following two conditions holds for at least one action $a \in \mathcal{A}$:

1. $v_i \in \mathit{Vars}(\mathit{pre}(a))$ and $v_j \in \mathit{Vars}(\mathit{eff}(a))$ (*pre-eff correlation*)
2. $v_i \in \mathit{Vars}(\mathit{eff}(a))$ and $v_j \in \mathit{Vars}(\mathit{eff}(a))$ (*eff-eff correlation*)

Informally, v_i and v_j are correlated either if the action that changes v_j can only be executed when v_i has a certain value, or if v_i and v_j are both changed by the same action.

Now we define additivity of patterns as follows: Two disjoint patterns P_1 and P_2 are *additive* if no variable from P_1 is correlated with any of the variables in P_2.

Intuitively, two additive patterns describe two distinct subproblems of the original task. Additivity between two such patterns guarantees that no part of one such subproblem depends on any part of the other problem. Because these tasks are independent, we can solve them in any order and even arbitrarily *interleave* plans for each subtask.

2.3.2 Causal Graph

In this thesis, we visualize correlations between variables of a planning task in the *causal graph* (Knoblock, 1994) which we define for a task Π with the set of variables $\mathcal{V}$ as a graph $G = \langle E, V \rangle$. This graph holds a vertex for each variable $v \in \mathcal{V}$ and an edge between each pair of correlated variables. We define G to be *undirected* because correlations that we intend to visualize are bidirectional.

To illustrate this, we introduce the following task: Two robots r_g and r_p are tasked with collecting waste for recycling. Specifically, r_g must collect items g_1, g_2, g_3, which are made of glass and the goal of robot r_p is to collect plastic objects p_1, p_2 and p_3. These items are scattered in a 3×3 grid which represents rooms that the robots can move in. An item can either be located in one of the rooms, or be picked up and stored inside a robot. Robots can only collect items that are in the same room as them. The causal graph of this task is visualized in Figure 2.1. g_n and p_n for $n \in \{1, 2, 3\}$ are goal variables.

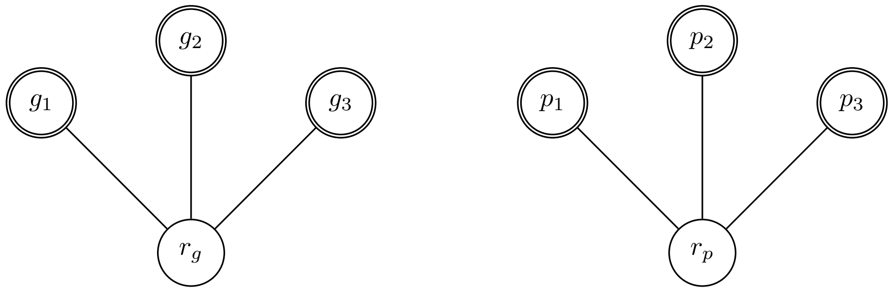

Figure 2.1: Causal graph of the recycling robots task.

2.3.3 Canonical Heuristic

Not all algorithms we present in this paper are guaranteed to output a pattern collection in which all patterns are additive. A more differentiated approach to deriving admissible heuristics from pattern collections is the *canonical heuristic function* (Haslum et al., 2007), which is defined as follows: Given a collection of patterns C and the set A of maximal additive subsets of C, the canonical heuristic h^C is the function

$$h^C(s) = \max_{S \in A} \sum_{P \in S} h^P(s) \tag{2.2}$$

For example, in the case $C = \{P_1, P_2, P_3\}$, where P_2 is additive with P_1 and P_3, which, however, are not additive with each other, we get $A = \{\{P_1, P_2\}, \{P_2, P_3\}\}$ as the set of maximal additive subsets and the canonical heuristic is

$$h^C(s) = \max\{\, h^{P_1}(s) + h^{P_2}(s),\ h^{P_2}(s) + h^{P_3}(s)\,\} \tag{2.3}$$

Intuitively, the canonical heuristic tries to add for as long as admissibility is guaranteed and takes the maximum of all heuristics that cannot be added. Because of this versatility, the canonical heuristic is used with the output of all pattern collection generation algorithms that are presented in this paper.

3 Constructing Pattern Collections with CEGAR

Counterexample Guided Abstraction Refinement, also abbreviated as CEGAR, is an iterative technique for generating abstraction heuristics which provide accurate path cost estimates for states of the original problem.

The idea of the algorithm is to start with a trivial, weak abstraction and solve the corresponding abstract planning task. Upon having obtained an optimal plan for the abstract task, the algorithm tests whether the sequence of actions from the plan can also be used to solve the concrete task. If the abstract plan is not applicable in the concrete case, then the algorithm will detect a *flaw*, which is a packet of information that states the reason why the abstract plan could not be executed in the concrete setting. Based on the information contained in the flaw, the algorithm is able to refine the abstraction such that said flaw does not occur again. After refinement, the algorithm repeats at the next iteration. This process ends if an abstraction is found whose plan could be applied on the original problem without flaws or some other implementation-specific termination condition is met. In the former case, CEGAR has managed to find an optimal solution for the concrete planning task, while in the latter case, the algorithm returns an admissible heuristic which can then be used by a search algorithm like A* for finding the plan.

3.1 CEGAR for Pattern Databases

The goal of CEGAR for PDBs is to generate a pattern collection whose corresponding canonical heuristic offers accurate plan cost estimates. Algorithm 1 illustrates the general framework of the CEGAR approach to pattern construction.

The main difference of this algorithm compared to the general idea described at the beginning of this chapter is that CEGAR for PDBs works with several abstractions as opposed to just a single one. This is due to the fact that the heuristic we are looking to construct is based on a collection of abstractions. Furthermore, the plan of each abstraction can fail to execute on the concrete task Π, which would give rise to a set of flaws, instead of a single one. With multiple flaws present, it is no longer unambigously clear which flaw should be repaired during a single iteration, which is why our algorithms can employ different *flaw selection* strategies.

Algorithm 1: General CEGAR algorithm for pattern collection generation

```
Input: concrete task $\Pi$
Output: pattern collection C

C $\leftarrow$ InitialCollection($\Pi$);
while $\neg$Terminate() do
    F $\leftarrow$ GetFlaws(C, $\Pi$);
    if $|F| = 0$ then
        ExtractPlan();
    else
        f $\leftarrow$ SelectFlaw(F);
        Refine($\Pi$, f);
    end
end
return C;
```

In this chapter, we introduce two CEGAR algorithms for pattern databases, which both share the same general structure as presented in Algorithm 1, but generate pattern collections with different properties.

$CEGAR^{fadd}$ (*fadd* stands for *forced additivity*) only outputs pattern collections where all patterns are pairwise additive. As such, this algorithm attempts to decompose the original planning task into a set of pairwise independent subtasks, which would allow the canonical heuristic to simply return the sum of all PDB estimates.

In order to ensure additivity for all patterns, it is often necessary to merge two non-additive patterns together. Because this merging threatens to dramatically increase PDB memory requirements, we introduce $CEGAR^{nadd}$ as an alternative to $CEGAR^{fadd}$.

This algorithm does not try to always produce a set of pairwise additive patterns. Instead, its goal is merely to ensure that all patterns are *disjoint*, i.e. a variable is not allowed to be included in more than one pattern at the same time. This requirement was put in place because, while the algorithm does not aggressively attempt to ensure that additivity holds between all patterns, we are still interested in obtaining a collection of patterns with additivity relations between as many patterns as possible, as this would yield a stronger canonical heuristic. Patterns with overlapping variables, however, are automatically not additive as each variable is always trivially correlated with itself.

Both algorithms are signaled to terminate by the Terminate function once at least one PDB has reached the maximum allowed size limit, or if an optimal concrete plan could be found (i.e. the list of flaws is empty).

In practice, the two algorithms differ in their respective implementations of GetFlaws and Refine functions. In the following sections, we present building blocks that $CEGAR^{nadd}$ and $CEGAR^{fadd}$ consist of.

3.2 The Initial Pattern Collection

The starting point of our CEGAR algorithms is described by an initial collection of patterns. In our framework, this is handled by the InitialCollection function. Because the choice of initial patterns has the potential to negatively influence the convergence of CEGAR, we focus on two general options which are known to be *unproblematic* and also carry the advantage of being easy to implement. In this thesis, the initial C holds either a single pattern, with one randomly selected goal variable, or a separate pattern for each goal variable of the planning task.

Generally, unproblematic initial collections are collections whose patterns only contain variables that are at least *indirectly correlated* with each other, i.e. for each pair of variables v_i, v_j of some pattern, the causal graph of the planning task must contain a path from v_i to v_j. It is easy to see that the two approaches we employ to generate the initial collection trivially fulfill this criterion, since all patterns contain only one variable. The reason for this requirement will be discussed in greater detail in Section 3.5 where we characterize the convergence of our algorithms.

3.3 Flaws

Understanding what a flaw is in context of pattern database heuristics is crucial for understanding how the CEGAR algorithm operates. For this reason this section discusses the two types of flaws that can arise. We also provide a detailed overview of the GetFlaws function whose task is to detect said flaws.

As previously established, the state space generated by a pattern abstraction P is obtained by joining all states with identical values for all variables $v \in P$. Differences in values of variables that are not in P are ignored. Since these variables are also removed from $pre(a)$ for all actions a of the planning task, it is possible for the abstract state-space to have transitions that do not exist in the concrete state-space. If the optimal plan for said abstract state-space involves any such transition, then it can not be applied to the concrete task. We refer to this kind of flaw as *precondition violation flaw* because it is caused by an action not being applicable.

A different problem arises when a pattern is missing a goal variable. In this case, it is possible to obtain a flawlessly applicable plan, which, however, does not lead to a concrete goal state. This type of flaw will be referred to as a *goal violation flaw.*

To illustrate these two cases on the recycling robots task from Section 2.3.2, we construct abstract tasks induced by the patterns $P_1 = \{p_1, p_2, p_3\}$ and $P_2 = \{r_p, p_1, p_2, p_3\}$. In the case of P_1, the abstract task is missing information about the position of the plastic collecting robot r_p. Thus preconditions that ensure that an item can be picked up only when it is in the same room as the robot, are also missing. Therefore the optimal plan of the abstract task will consist only of instructions for picking up each of the three plastic objects and no movement instructions for the robot. This plan will always fail on the concrete task if at least one item is not located in the same room as robot r_p, thus raising a precondition violation flaw.

The task induced by pattern P_2 has knowledge about the plastic collecting robot as well as all plastic waste, but contains no information about glass items or the robot that must collect them. This means that this subtask is concerned only with the collection of plastic. Here, the optimal plan contains both, the necessary movement and collection actions for robot r_p. For this reason, this sequence of operations can indeed be executed in the original problem, however, it leads to a state where all plastics have been collected and all glass objects are still scattered around the different rooms. Because this state does not agree with the goal state, this situation constitutes a goal violation flaw.

3.3.1 Solving Abstract Tasks

As shown in the above example, in order to find flaws of an abstraction P, one needs the corresponding abstract plan, which can be obtained by solving the abstract task. Because there are no conceptual differences between the concrete task Π and its abstracted version Π^P, we could employ the same technique, specifically the A* algorithm, for finding the abstract plan. This approach, however, comes with unnecessary overhead, as it requires maintaining a priority queue of yet unvisited search nodes, whose associated actions may or may not be part of the optimal plan. Algorithm 2 is a greedy algorithm which is guaranteed to return optimal abstract solutions.

Algorithm 2: FindPlan

```
Input: task $\Pi^P$ and its perfect heuristic $h^*_P$
Output: optimal plan $\tau$ of $\Pi^P$
$\tau \leftarrow \emptyset$;
s ← GetInitialState($\Pi^P$);
if $h^*_P(s) = \infty$ then
    return unsolvable;
end
while $h^*_P(s) \neq 0$ do
    A ← GetApplicableActions(s);
    â ← $\arg\min_{a \in A} h^*_P$(Successor(s, a));
    s ← Successor(s, â);
    $\tau \leftarrow \tau \cup \{$â$\}$;
end
return $\tau$;
```

The optimality guarantee of this algorithm stems from the fact that it uses the perfect heuristic h^*_P of Π^P. This means that FindPlan will always select the correct node for expansion.

This algorithm, however, does not work for tasks which have zero-cost operators, because those tasks may have a zero-cost optimal plan (i.e. $h^*_P(s) = 0$ for the initial state s). In this case, our CEGAR implementation falls back to the A* approach.

3.3.2 Abstract Plan Validation

An algorithm for validating abstract plans in the concrete state-space is an essential part of both flaw retrieval routines that will be introduced in the next two sections. In this section we describe the ApplySolution function, whose purpose is precisely the validation of the aforementioned abstract solutions. The function iterates over all actions from the given plan τ and verifies for each action a whether it can be applied to the current concrete state s.

This is accomplished by comparing variable values of s to the ones specified in $pre(a)$ and collecting those variables which do not satisfy the preconditions. If s agrees with $pre(a)$ then operator a is applicable and plan validation may continue, otherwise the algorithm aborts and returns the last state it could reach along with the list of variables that prevent action a from being applicable. This list is precisely the set of flaws of τ.

3.3.3 Flaw Retrieval for *CEGAR*^{fadd}

As already established in Section 2.3.1, a collection which consists only of additive heuristics describes a set of independent subtasks. Individually, solutions to these subtasks are unlikely to solve the original planning task. However, the additivity property makes it possible to arbitrarily combine these solutions. The GetFlaws function shown in Algorithm 3 starts with the initial state of the concrete state space $\mathcal{S}(\Pi)$ and iterates over all patterns from the given pattern collection. The function then tries to apply the plan of the corresponding abstract task to the concrete state s.

Algorithm 3: GetFlaws(C, Π) for *CEGAR*^{fadd}

```
Input: pattern collection C and task Π
Output: list of flaws F
F ← $\emptyset$;
s ← GetInitialState(Π);
for P ∈ C do
    $\tau$ ← FindPlan($\Pi^P$, $h^*_P$);
    (s, $F_{new}$) ← ApplySolution(s, $\tau$);
    F ← F ∪ $F_{new}$;
end
if F = $\emptyset$ ∧ ¬IsGoal(s) then
    F ← {RandomUnusedGoalVariable(Π)};
end
return F;
```

If the algorithm has processed all $P \in C$ and the list of flaws is empty, but s is not a goal state, then this means that the combination of all plans associated with already existing patterns was unable to satisfy some goal condition. From this we can conclude that there is a goal violation. As already mentioned, flaws of this type are caused by at least one goal variable not being included in any pattern $P \in C$. However, if *CEGAR*^{fadd} has indeed arrived at a goal state, then a combination of plans associated with patterns $P \in C$ constitutes a solution. In this case, the algorithm will return an empty flaw list that will prompt Algorithm 1 to extract the optimal plan with the ExtractPlan function and terminate.

3.3.4 Flaw Retrieval for *CEGAR*^{nadd}

If we construct patterns which are not necessarily pairwise additive, we can no longer arbitrarily interleave their respective plans. In this case, the GetFlaws function processes all abstract solutions in isolation, as shown in Algorithm 4.

Algorithm 4: GetFlaws(C, Π) for *CEGAR^{nadd}*

```
Input: pattern collection C and task Π
Output: list F of flaws
F ← ∅;
for P ∈ C do
    s ← GetInitialState(Π);
    $\tau$ ← FindPlan($\Pi^P$, $h^*_P$);
    (s, $F_{new}$) ← ApplySolution(s, $\tau$);
    if $F_{new} = \emptyset$ then
        if ¬IsGoal(s) then
            $F_{new}$ ← {RandomUnusedGoalVariable(Π)};
        else
            MarkSolution(P);
            return ∅;
        end
    end
    F ← F ∪ $F_{new}$;
end
return F;
```

The main difference of this version of GetFlaws to the one from Algorithm 3 is that all plans are applied to the concrete initial state rather than to the state returned by the previous ApplySolution call. Furthermore, since abstract plans can no longer be combined, a goal violation can occur on each iteration. The algorithm terminates prematurely if it finds a single plan that did not generate any flaws and led to a goal state. A plan like this describes a solution of the concrete task, which makes further iterations of the CEGAR loop (see Algorithm 1) unnecessary. The pattern that induced this plan is marked such that ExtractPlan is able to extract the correct solution.

3.4 Pattern Refinement

In Section 3.3, we established that flaws are generally caused by an abstraction not having information about either a variable for which some action of the concrete task has a precondition, or a variable for which there exists an assignment in $\mathcal{G}$ of the given planning task Π. For this reason, abstraction refinement can be viewed as the act of extending an abstract task with knowledge about the variable that caused the flaw. In the case of a PDB abstraction, this is done by adding the offending variable to the pattern.

Similarly to flaw retrieval, details of the refinement process vary between our two variations of CEGAR for PDBs.

3.4.1 Refinement in *CEGAR^{nadd}*

Given a collection C, a flaw f with the corresponding variable v_f the Refine function of *CEGAR^{nadd}* repairs said flaws according to the following rules: If f is a goal violation flaw, then pattern $\{v_f\}$ is added to the pattern collection (i.e. $C = C \cup \{\{v_f\}\}$). On the other hand, if f is a precondition violation flaw raised by the abstract plan of pattern P_f, then the refinement algorithm checks whether the variable v_f is not already part of some other pattern P. If it is, then the only way to repair the flaw while also preserving disjointedness of P and P_f is to merge the two patterns. If v_f is not included in any pattern of C yet, then it is added to P_f.

3.4.2 Refinement in *CEGAR*^{fadd}

The set of all pattern collections in which all patterns are pairwise additive is a subset of all pattern collections with disjunct patterns. Therefore *CEGAR*^{fadd} can be viewed as a special case of $CEGAR^{nadd}$. Despite this fact, the additivity restriction makes for a simpler refinement algorithm, where it is not necessary to differentiate between goal and constraint violations.

Here, the refinement function creates a separate pattern for the flaw variable v_f and then tests it for additivity with all other patterns that are already in the pattern collection. Patterns $P \in C$ which turn out to not be additive with $\{v_f\}$ are collected in the *merge list* M. Once all additivity checks are done, all patterns in M are merged into a single pattern, which is then added to C, replacing the patterns it was made of in the process.

3.5 Convergence

In an environment with limitless resources (i.e. time and memory) *CEGAR*^{fadd} is guaranteed to find an optimal solution to the concrete task in a finite number of steps for all tasks $\Pi = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$. Specifically, *CEGAR*^{fadd} will reach this solution in at most $n-m$ iterations, where $n = |\mathcal{V}|$ and m is the number of variables in the initial pattern collection. This property is a direct consequence of the refinement policy of the algorithm. Recall that refinement of a pattern in reaction to some flaw always implies the introduction of a new variable $v \in V$ to the pattern collection. With m variables being part of the collection from the beginning, only $n - m$ variables can still be added. Past this point, no new refinements can be made any more and we say that the algorithm has *converged*.

With knowledge about correlations between variables it is also possible to predict the exact pattern collection that the algorithm will converge to (provided it does not find a solution sooner). If the causal graph of the given concrete task is connected, i.e. there exists a path from each vertex to any other vertex, then the algorithm will converge to the collection $C = \{\mathcal{V}\}$, i.e. it will output the set of all variables as its sole pattern.

If the graph is disconnected, however, then this means that the concrete task can be decomposed into a set of pairwise independent subtasks. In this case, each maximal connected subgraph of the causal graph represents a separate pattern. Because subtasks induced by these patterns are independent, their respective plans can be combined into a single optimal solution of Π.

We illustrate this on the causal graph of the recycling robots task (Figure 2.1). From the graph it is apparent that the tasks of collecting glass and plastic items are indeed independent. Therefore *CEGAR*^{fadd} will converge to $C = \{\{r_g, g_1, g_2, g_3\}, \{r_p, p_1, p_2, p_3\}\}$.

Here, the optimal solutions of the two abstractions, when combined, represent a solution of the original task.

While $CEGAR^{nadd}$ converges to the same final collection, there are two key differences that one needs to be aware of. Firstly, $CEGAR^{nadd}$ is only guaranteed to converge in *at least* $n - m$ steps, but can also take longer, because flaw refinement of this algorithm does not necessarily imply introduction of new variables to the pattern collection. A flaw can also be caused by a pattern not having a variable that is already included in some other pattern. In this case, the refinement step only entails a merge of the two existing patterns. Secondly, even when $CEGAR^{nadd}$ reaches the point where further refinements are no longer possible, it is not guaranteed to find a plan that solves the concrete task. This is due to the fact that $CEGAR^{nadd}$ does not attempt to interleave or combine individual abstract plans, but instead validates them in isolation (see Algorithm 4).

Regardless of the chosen CEGAR algorithm, pattern collections that are returned on convergence are only guaranteed to have the form described above if the initial collection is unproblematic (i.e. it contains only patterns whose variables are at least indirectly correlated with one another).

For an example where a collection which does not fulfill this requirement leads to less-than-optimal results, consider the initial pattern collection $\tilde{C}_{init} = \{\{g_1, p_1\}\}$ of the recycling robots problem. Variables g_1 and p_1 are clearly uncorrelated with each other, but all other variables of the planning task are at least indirectly correlated with one of them. Therefore the algorithm will converge to $\tilde{C} = \{\mathcal{V}\}$, thus completely ignoring the convenient decomposition into subtasks that is possible in this case.

4 Merge Avoidance

Although frequent merging is often necessary to ensure that the pattern collection always contains only disjunct, additive patterns, it also represents a major problem for the algorithm. This is due to the fact that pattern database sizes scale with the number of variables in the pattern, as well as the domain sizes of said variables. Therefore, a merge of several patterns with small pattern databases has the potential to result in a pattern database whose space requirements greatly exceed the total available memory. For example, consider a pattern collection with three patterns, where each pattern has five variables with a domain size of five. Assuming that a single pattern database entry is a 4 byte integer and using Formula 2.1 we get $4\,\text{B} \times (5^5 + 5^5 + 5^5) = 37.5\,\text{kB}$ as the combined memory cost of all pattern databases. Merging the three patterns, however, would result in a single pattern database of size $4\,\text{B} \times 5^{15} \approx 122\,\text{GB}$. Altough merging cannot be eliminated entirely, we present three different techniques for alleviating this problem in the following sections.

4.1 Least-Common-First Flaw Selection

So far, the two CEGAR algorithms we introduced, randomly selected the next flaw for refinement. The list of flaws generated on each iteration, however, is allowed to store duplicates, i.e. a variable may be mentioned in the flaw list several times. This occurs when multiple abstract plans fail in the concrete state space because of the same variable not being present in the corresponding pattern. For this reason, random flaw selection is naturally biased towards choosing the flaw with the most common variable. Consider a pattern collection $C = \{P_1, P_2, P_3, P_4\}$ whose respective abstract plans failed with the following list of flaws:

Pattern	Flaw Variable
P_1	v_1
P_2	v_1
P_3	v_1
P_4	v_2

The first three abstract plans failed with a precondition violation because of variable v_1 and the abstract plan of pattern P_4 failed because of v_2. In this case, there is a 75%

chance that v_1 is chosen for refinement next. In the case of $CEGAR^{fadd}$, choosing to refine a flaw associated with v_1 would cause P_1, P_2 and P_3 to be merged with $\{v_1\}$. The goal of *least-common-first* (LCF) flaw selection is to delay pattern merges by selecting the flaw with a variable that appears in the given flaw list the fewest number of times. In the given example, LCF algorithm would choose v_2 for refinement, which, in the ideal case would only merge P_4 with $\{v_2\}$. The choice made by LCF, however, is not always guaranteed to be optimal. For example, it is possible that patterns P_1, P_2 and P_3 also also not additive with $\{v_2\}$ but just have not issued a corresponding flaw yet. In this situation, v_1 would have been a better choice. Still, when faced with the choice between the possibility of a merge and a guaranteed merge, LCF always chooses the former.

At a first glance, it might not be immediately clear why a different order of handling flaws might have an influence on the end-result, since the order of refinement does not influence convergence. The reason why LCF selection still makes a difference is due to the pattern size constraints that are imposed on the CEGAR algorithms in practice: Once at least one pattern has reached the allowed limit, the algorithms abort and return the pattern collection that could be generated up to that point. With LCF selection, the goal is to reach that limit by merging as little as possible.

4.2 Blacklisting

A more invasive way to avoid merging is to change CEGAR algorithm to completely ignore variables which threaten to cause many merges. We refer to this approach as *blacklisting.* This technique identifies a number of non-goal variables with the most correlations and puts them on a *blacklist.* Preconditions which involve blacklisted variables are always ignored during plan validation, i.e. validation continues even if the plan does not satisfy a precondition on one of the blacklisted variables. For this reason, a CEGAR algorithm will never add said variables to the pattern collection. Thus blacklisting effectively changes the pattern collection that CEGAR will to converge to.

We demonstrate this effect on a planning task for which correlations between variables are visualized in Figure 4.1. We established in Section 3.5 that CEGAR with this task would converge to $C = \{\mathcal{V}\}$, which is often too expensive to generate a PDB from. After blacklisting v_1, which is the variable with the most correlations, CEGAR will have the same convergence as the task described by Figure 4.2. In this case, the final output is $C = \{\{v_2, v_6, v_7\}, \{v_3, v_8, v_9\}, \{v_4, v_{10}, v_{11}\}, \{v_5, v_{12}, v_{13}\}\}$ where individual pattern databases are significantly easier to compute and store.

One of the drawbacks of blacklisting is that ignoring variables causes the resulting heuristic to further understimate the path cost for all states. In addition to that, that blacklisting severely restricts an algorithm's ability to find optimal concrete plans during refinement. This is due to the fact that an empty flaw list can now have two meanings where it previously only had one: In CEGAR without blacklisting, an empty list of flaws implies that one or multiple abstract plans constitute a concrete solution, since they could be flawlessly applied in the concrete state-space and led to a goal state.

In CEGAR with blacklisted variables, however, the absence of flaws could also mean

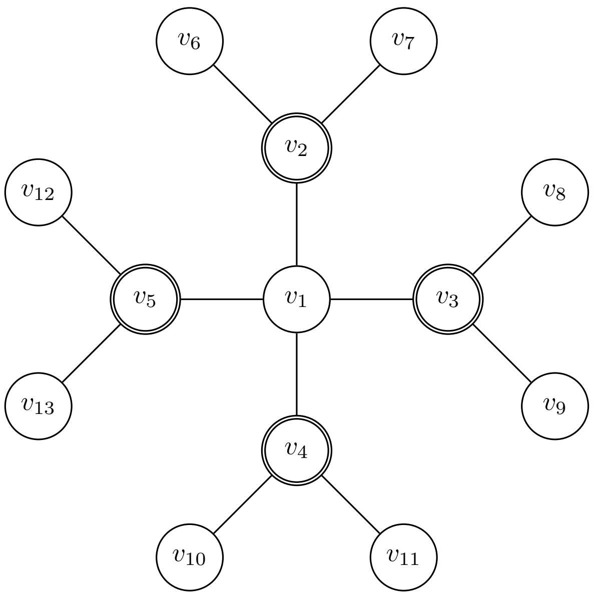

Figure 4.1: Causal graph of the blacklisting example task.

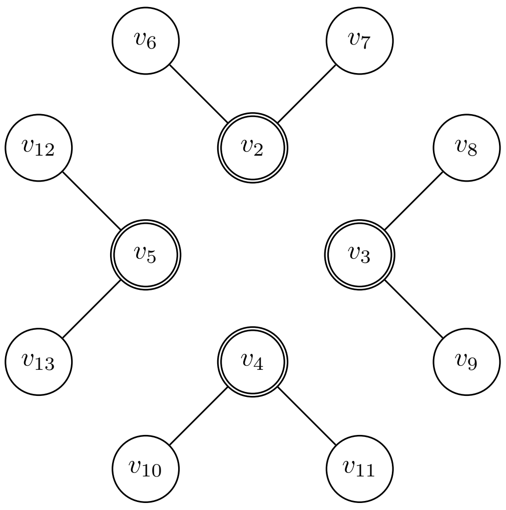

Figure 4.2: Causal graph of the blacklisting example task after blacklisting.

that there were in fact precondition violations during plan verification, which were, however, ignored because repairing them would have required the introduction of a blacklisted variable.

The question of whether the often more favorable convergence outweighs the disadvantages of this technique will be discussed in the next chapter.

4.3 Partial Additivity

We can obtain a different convergence for CEGAR by employing a stricter definition of correlations. Recall that correlations, as defined in Section 2.3.1 demand that two variables v_i, v_j are either changed by the same action (effect-effect correlation), or that the applicability of some action that changes v_i depends on v_j (precondition-effect correlation). We now restrict this definition by removing the latter condition, i.e. two variables are only correlated if both are assigned new values by the same action. Using restricted correlations results in a causal graph with lower density, since edges that were induced by pre-eff correlations do not exist under the strict definition. Because the graph has fewer edges, it is more likely that it will be disconnected, which would make CEGAR converge to multiple separate patterns describing discrete subtasks.

Additivity that is defined analogously to the definition in Section 2.3.1, but only for effect-effect correlations will be referred to as *partial additivity.*

$CEGAR^{padd}$ is a modified version of $CEGAR^{fadd}$ that aims to output a collection of pairwise partially additive patterns. While much of the algorithm remains unchanged between these two versions, there is a number of important differences:

Firstly, the sum of PDBs of two partially additive patterns P_1 and P_2 is guaranteed to be lower or equal to their union, i.e. $h^{P_1 \cup P_2}(s) \geq h^{P_1}(s) + h^{P_2}(s)$ for all states s. This is because there may be precondition-effect correlations between variables of P_1 and P_2. Operator preconditions that induce these correlations are known to the task $\Pi^{P_1 \cup P_2}$, but are abstracted away in tasks Π^{P_1} and Π^{P_2}. For this reason, actions that are not applicable in $\Pi^{P_1 \cup P_2}$ may be applicable in the individual tasks, thus creating 'shortcuts' in their corresponding state spaces, which can lead to lower plan cost estimates for some states.

Secondly, abstract plans of partially additive patterns can no longer be arbitrarily interleaved or combined into a single concrete plan the way it is possible with additivity that accounts for eff-eff as well as pre-eff correlations. This is because given two patterns P_1 and P_2, an action of the plan τ_{P_1} of P_1 may have a precondition on some variable $v \in P_2$. None of the actions of τ_{P_1} have an effect on v (as this would go against the definition of partial additivity), but the plan τ_{P_2} may have actions that change this variable. In this case, the applicability of actions of τ_{P_1} may depend on how many actions of plan τ_{P_2} were already executed. For example, variable v may initially have the value needed by actions from τ_{P_1} to be applicable, but the first action of τ_{P_2} might change this value, thus causing τ_{P_1} to fail.

Thus, it is possible that a certain order of actions from multiple plans constitutes a concrete solution while other combinations only lead to more flaws. Pascal Bercher conjectures that the problem of finding such an optimal interleaving is NP-hard due to a reduction from a proof in the context of partial order planning by Nebel and Bäckström (1994).[1]

Instead, $CEGAR^{padd}$ greedily looks for a working interleaving by iterating over all plans and executing them as far as possible (i.e. until the plan either fails with a flaw or finishes). If all plans could be fully executed, then $CEGAR^{padd}$ has successfully found a combination of plans that solves the concrete task. If some plans could not be finished because of a flaw, then the algorithm, once again, iterates over all plans and tries to apply them starting from the point where they last failed. This process repeats until either a combination is found or a full iteration over all plans passes without a single plan being able to execute one of its actions.

Furthermore, the $CEGAR^{padd}$ algorithm can encounter *irreparable flaws.* For an exam-

[1] Personal communication.

ple where this can happen, consider the following scenario: The plan of pattern P_1 fails with a precondition violation flaw because of some variable v_f. In both, $CEGAR^{nadd}$ and $CEGAR^{fadd}$, this implies that once this flaw has been chosen for refinement, variable v_f will be added to P_1 (possibly along with other variables). In $CEGAR^{padd}$, however, this is not guaranteed. If pattern $\{v_f\}$ turns out to be partially additive with P_1, then the two will never be joined together. Therefore P_1 will never receive variable v_f and will thus keep failing with the exact same flaw on each iteration, while v_f will reside in a separate pattern. In these situations, $CEGAR^{padd}$ keeps running for as long as it keeps finding flaws that can be fixed and returns its current pattern collection once only irreparable flaws are left.

Lastly, because of the latter problem, it is possible for a pattern collection to have patterns that do not contain a single goal variable (e.g. if v_f from the above example is a non-goal variable). Such a pattern models a subtask that does not have a goal. In this case, the h-values are always zero and thus do not contribute to the final estimate.

5 Evaluation

All techniques introduced in the previous two chapters were implemented in the Fast Downward planning system (Helmert, 2006). We tested various combinations of the three CEGAR algorithms on a benchmark set of 1667 tasks from planning domains of all International Planning Competitions (IPC). A CEGAR algorithm is fully defined by the following parameters:

- PDB size limit: once at least a single PDB has reached this limit, the algorithm will abort. Preliminary experiments showed that a size limit of 1 000 000 states for individual PDBs provides good results. Hence, all experiments presented in this chapter use this limit.
- Initial pattern collection, i.e. the collection returned by InitialCollection. As previously mentioned, our implementation accounts for two options here, namely a single pattern with a random goal variable, or a pattern for each goal variable of the planning task.
- Flaw selection strategy: We either select flaws randomly, or in accordance with the LCF strategy. Random selection is the default strategy that is used in all our experiments unless specified otherwise.
- Blacklisting: whether the algorithm should ignore a number of variables with many correlations. Blacklisting is deactivated per default. In experiments where blacklisting is used, the blacklist has a size of 20, meaning that the algorithm will try to blacklist *up to* 20 non-goal variables.

Furthermore, in all experiments, execution time (i.e. time for pattern collection generation and subsequent A* search) is limited to 30 minutes, while memory is limited to 2GB. In the following sections we show how our three CEGAR algorithms react to different choices for the last three paramters.

5.1 Initial Collections and Flaw Selection

First we show how the coverage (i.e. the number of solved tasks) of the three CEGAR algorithms differs for different initial collections. Table 5.1 shows that, despite being able to

	CEGAR^{fadd}	*CEGAR*^{padd}	*CEGAR*^{nadd}
random goal	735 (231)	736 (122)	**757** (153)
all goals	736 (229)	740 (118)	**790** (145)
max of both	750 (240)	724 (129)	**791** (158)
random goal & LCF	737 (230)	742 (122)	**757** (153)
all goals & LCF	739 (227)	740 (118)	**790** (143)
max of both	748 (239)	721 (129)	**793** (158)

Table 5.1: Coverage of CEGAR algorithms given different starting collections and flaw selection strategies. Numers in parentheses show how many tasks could be solved during pattern collection construction.

solve slightly more tasks during refinement (see numbers in parentheses), CEGAR algorithms that start with a single random goal variable in the initial pattern collection are generally worse than algorithms that start with the other option. While this difference is negligible in the case of *CEGAR*^{fadd} and *CEGAR*^{padd}, *CEGAR*^{nadd} is able to perform significantly better when starting with all goal variables. We believe that the reason why all goals are a better starting point is the slightly faster convergence, since if all goal variables are already present, goal violation flaws are no longer possible. Furthermore, starting with many patterns instead of just one allows CEGAR to refine many different subtasks. On the other hand, if an algorithm starts with only one goal variable, then it is possible that CEGAR will refine the single corresponding pattern until its respective PDB has more than a million entries. In this case, our algorithms would terminate before encountering a single goal violation. For this reason, they would never try to model a different subtask.

Our analysis of collection construction time as well as node expansions until the last f-layer (see Figures 5.1-5.6) shows that starting with all goal variables indeed leads to a better heuristic most of the time. This further supports our theory that CEGAR algorithms initialized this way are better because they have a wider pool of patterns available for refinement from the very beginning. The fact that algorithms initialized with a single pattern focus exclusively on that one pattern until a goal violation occurs, allows the algorithm to reach the maximum pattern size and terminate earlier. This explains often shorter runtime for algorithms that start with a single goal variable.

A lower coverage, of course, does not necessarily imply that the stronger algorithm could solve all tasks that were solved by the weaker algorithm. It is possible that the algorithm with lower coverage could find solutions for problem instances for which the stronger algorithm failed. For this reason, we take the maximum of canonical PDB heuristics that operate on pattern collections generated with different approaches. As can be seen in the third row of Table 5.1, the combination of *CEGAR*^{fadd} with different starting collections indeed has a higher coverage than both of them do individually. This is not the case for *CEGAR*^{padd}, where the added overhead of executing CEGAR twice causes more timeout errors.

Finally, our evaluation of LCF selection shows that this approach offers little to no improvements for all algorithms and starting collections. We believe that the reason for this is that CEGAR algorithms are able to reach flaws that cause many merges even though they are selected last by LCF.

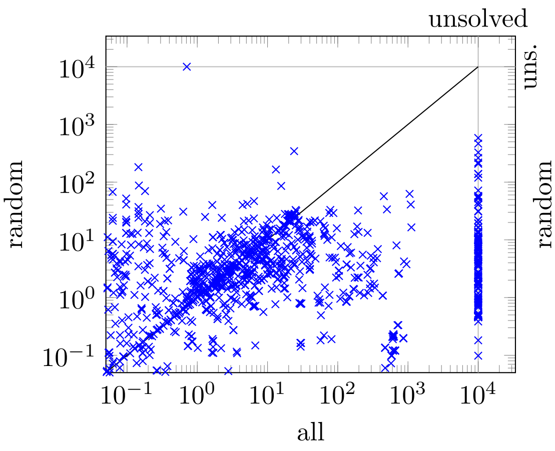

Figure 5.1: $CEGAR^{fadd}$ constr. time

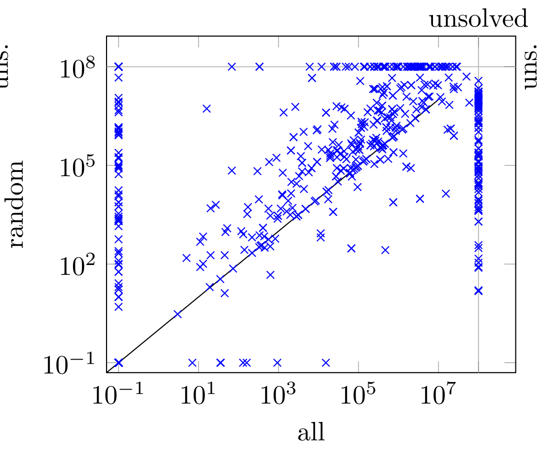

Figure 5.2: Expansions with $CEGAR^{fadd}$

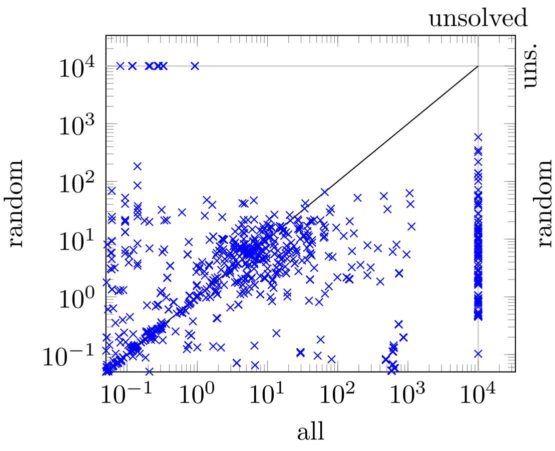

Figure 5.3: $CEGAR^{padd}$ constr. time

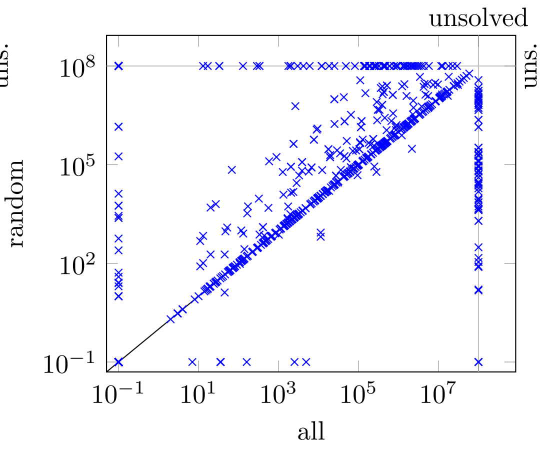

Figure 5.4: Expansions with $CEGAR^{padd}$

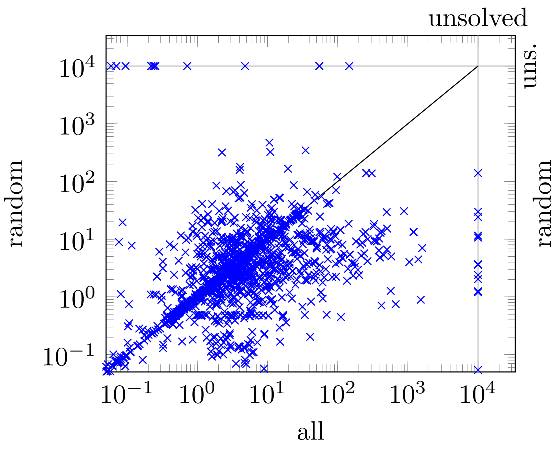

Figure 5.5: $CEGAR^{nadd}$ constr. time

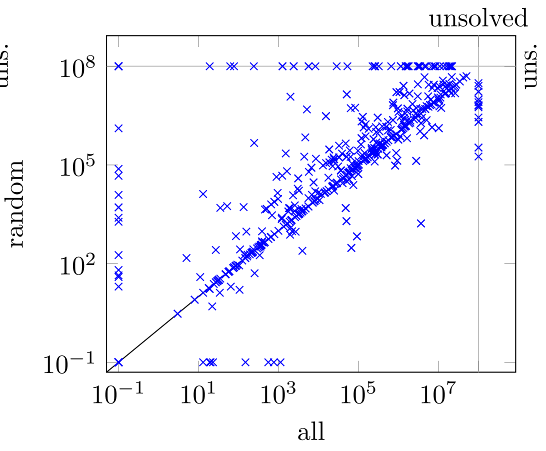

Figure 5.6: Expansions with $CEGAR^{nadd}$

	$CEGAR^{fadd}$	$CEGAR^{padd}$	$CEGAR^{nadd}$
no blacklisting	737 (230)	742 (122)	**790** (143)
blacklisting	743 (34)	**748** (36)	743 (4)
max of both	787 (230)	775 (119)	**811** (146)

Table 5.2: Coverage of CEGAR algorithms when blacklisting is employed.

5.2 Blacklisting

In the following experiment, we tested how blacklisting influences CEGAR performance (i.e. collection construction time, coverage and expansions until last f-layer). Here, $CEGAR^{fadd}$ and $CEGAR^{padd}$ are configured to start with one random goal variable, because this approach results in similar coverage while also being faster than starting with all goals. $CEGAR^{nadd}$, however, is initialized with a pattern for each goal variable, because here this initial collection leads to significantly better coverage. Additionally, all algorithms utilize LCF flaw selection. Table 5.2 shows our results.

As predicted, we observe that blacklisting severely restricts the ability of CEGAR to find concrete plans during refinement. Furthermore, we observe that blacklisting improves coverage in algorithms that have a greater tendency to merge patterns (namely $CEGAR^{fadd}$ and $CEGAR^{padd}$). This improvement does not extend to $CEGAR^{nadd}$, where blacklisting results in a disproportionate decrease in coverage. Here, the version without blacklisting usually results in better heuristic estimates which leads to fewer expansions (see Figure 5.12).

Taking the maximum of canonical heuristics that operate on pattern collections that were generated with and without blacklisting provides the greatest improvement in coverage as can be seen in the last row of Table 5.2, meaning that algorithms with and without blacklisting complement each other rather well.

A very prominent feature that is present in all scatter plots for expansions is the 'column' of points at the very left of each plot. These points represent planning tasks Π for which a CEGAR algorithm could converge to $C = \{\mathcal{V}\}$ (where $\mathcal{V}$ is the set of variables of Π), thus successfully computing the perfect heuristic for the concrete task. These are usually very small tasks with few variables. By blacklisting up to 20 variables in such tasks, the set of variables that can still be used for refinement is reduced to the point where no refinement is possible and CEGAR is forced to return the initial collection, which results in a weak heuristic. This is why blacklisting-based approaches can lead to orders of magnitude more search node expansions for simple problems.

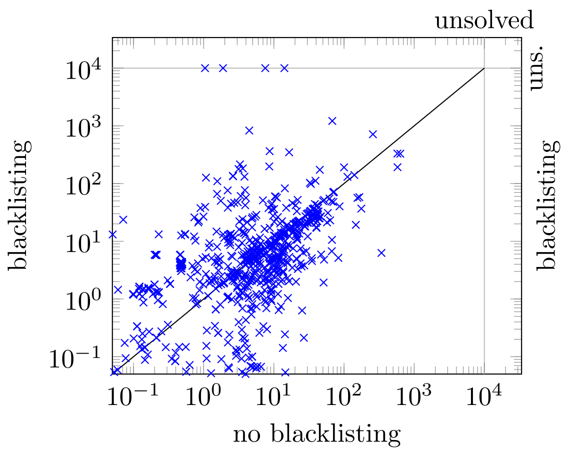

Figure 5.7: *CEGAR*^{fadd} constr. time

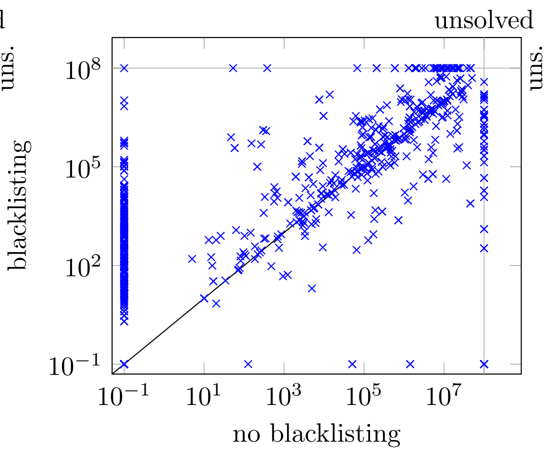

Figure 5.8: Expansions with *CEGAR*^{fadd}

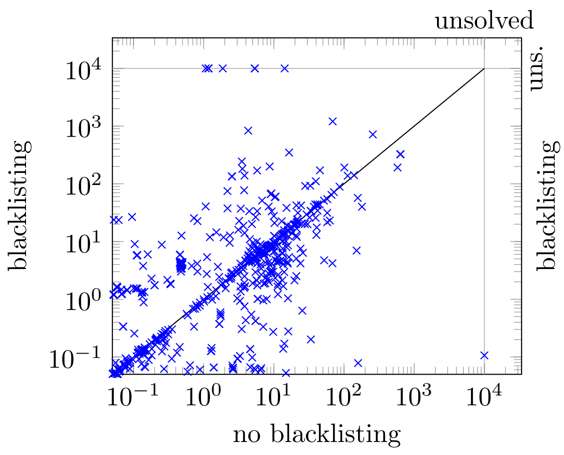

Figure 5.9: *CEGAR*^{padd} constr. time

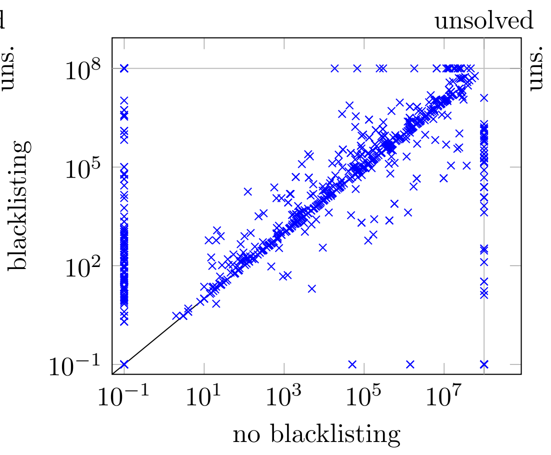

Figure 5.10: Expansions with *CEGAR*^{padd}

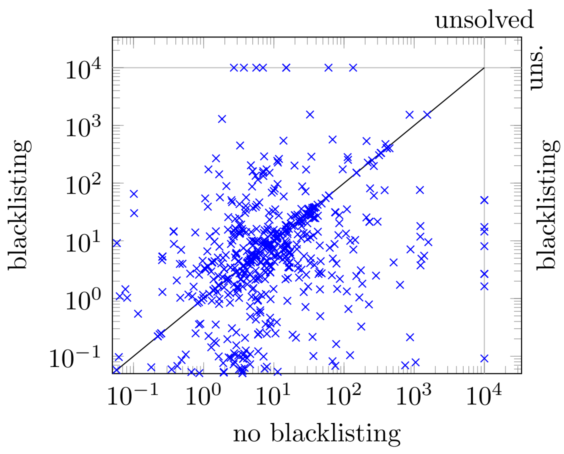

Figure 5.11: *CEGAR*^{nadd} constr. time

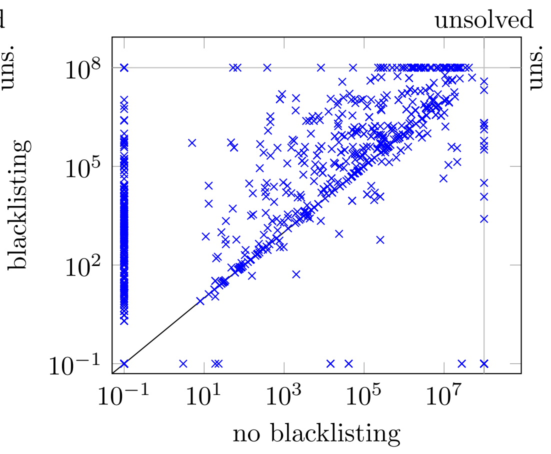

Figure 5.12: Expansions with *CEGAR*^{nadd}

5.3 iPDB

So far, $CEGAR^{nadd}$ has been the most powerful pattern collection generator discussed in this thesis. In this section we compare it to the iPDB algorithm (Haslum et al., 2007). The iPDB pattern collection generator approaches the problem of finding a set of good patterns by defining a search space where each state respresents a collection of patterns. The initial collection holds each goal variable of the planning task in a separate pattern. Starting with this state, the algorithm uses hillclimbing to visit neighboring states whose pattern collections result in a stronger canonical heuristic. Heuristic quality is evaluated empirically on a sample set of states of the given planning task.

Our experiments (Table 5.3) show that iPDB slightly outperforms $CEGAR^{nadd}$. A combination of all three CEGAR algorithms (set up in the same way as in the previous section but without blacklisting) comes out slightly on top, but combining $CEGAR^{nadd}$ with its blacklisted version provides better results (see Table 5.2).

By looking at the expansions comparison between iPDB and $CEGAR^{nadd}$ in Figure 5.13, we can observe that there are quite a few tasks for which $CEGAR^{nadd}$ generates the better heuristic. Indeed, the maximum of heuristics derived from collections generated by the two approaches results in the best coverage values, thus proving that $CEGAR^{nadd}$ and iPDB are complementary.

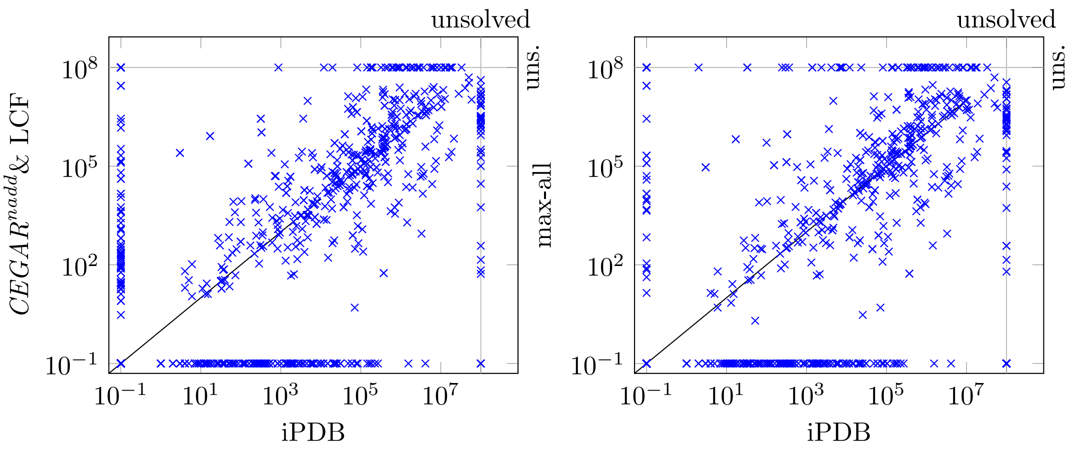

Figure 5.13: Expansions when deriving a heuristic from pattern collections generated by iPDB compared to $CEGAR^{nadd}$ (left) and in comparison to using the combination (maximum) of $CEGAR^{nadd}$, $CEGAR^{padd}$ and $CEGAR^{fadd}$ (right)

	$CEGAR^{nadd}$	max-all	iPDB	max(iPDB, $CEGAR^{nadd}$)
coverage	790 (143)	807 (253)	802	833 (148)

Table 5.3: iPDB in comparison as well as in combination with CEGAR. max-all describes the maximum of canonical heuristics of collections generated by $CEGAR^{nadd}$, $CEGAR^{padd}$ and $CEGAR^{fadd}$.

We believe that this complementarity can be explained by the fact that CEGAR algorithms can generate patterns that contain multiple goal variables. iPDB, on the other hand, is not capable of doing so, because this algorithm only extends patterns that already exist in the initial collection and never merges or introduces new patterns. Therefore, by using iPDB in combination with CEGAR we can obtain a more diverse pool of possible patterns.

6 Conclusion and Future Work

We have shown that the $CEGAR^{nadd}$ algorithm is comparable to iPDB in terms of coverage. However, despite the introduction of blacklisting and LCF, $CEGAR^{fadd}$ and $CEGAR^{padd}$ could not be made competitive with $CEGAR^{nadd}$.

Taking the maximum of canonical heuristics of pattern collections that were generated with and without blacklisting is an approach that leads to notable improvements in coverage for all algorithms. The need for multiple runs of a CEGAR algorithm, however, presents additional overhead. We believe that this problem can be alleviated by extending future CEGAR implementations with a classifier that decides on runtime whether blacklisting is appropriate for the given planning task. Furthermore, alternative blacklisting strategies, such as blacklisting based on variable domain size instead of the number of correlations, are also of interest.

While LCF flaw selection has only demonstrated marginal improvements over versions of CEGAR that select flaws randomly, it would be interesting to explore alternative flaw selection strategies here as well. A possible improvement over LCF could be a strategy that picks the next flaw based on available knowledge about correlations between variables.

All in all, we conclude that CEGAR is a promising approach to generating collections of patterns. We believe that the algorithms presented in this thesis can be further improved to derive even more informative heuristics.

Bibliography

Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

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

Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107, July 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, Sven Koenig, et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, volume 7, pages 1007–1012, 2007.

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

Craig A Knoblock. Automatically generating abstractions for planning. *Artificial intelligence*, 68(2):243–302, 1994.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97 – 109, 1985. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(85)90084-0. URL http://www.sciencedirect.com/science/article/pii/0004370285900840.

Drew M McDermott. The 1998 ai planning systems competition. *AI magazine*, 21(2):35, 2000.

Bernhard Nebel and Christer Bäckström. On the computational complexity of temporal projection, planning, and plan validation. *Artificial Intelligence*, 66(1):125–160, 1994. doi: 10.1016/0004-3702(94)90005-1.

Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the most out of pattern databases for classical planning. In *IJCAI*, pages 2357–2364, 2013.

Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement. In *ICAPS*, 2013.

Jendrik Seipp, Thomas Keller, and Malte Helmert. A comparison of cost partitioning algorithms for optimal classical planning. 2017.

Silvan Sievers, Manuela Ortlieb, and Malte Helmert. Efficient implementation of pattern database heuristics for classical planning. In *SOCS*, 2012.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Alexander Rovner

Matriculation number — Matrikelnummer
2015-050-289

Title of work — Titel der Arbeit
Pattern Selection using Counterexample-guided Abstraction Refinement

Type of work — Typ der Arbeit
Bachelor Thesis

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

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

Basel, 12.08.2018

Alex R

Signature — Unterschrift