Compressed Pattern Databases for Classical Planning

Bachelor thesis

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

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

Pascal Mafli
pascal.mafli@unibas.ch
06-065-965

November 24th 2020

Acknowledgments

I want to thank my supervisor Silvan Sievers for the time and effort helping me to finish this thesis. He was always very helpful and patient when problems occurred. Another thank you goes out to Malte Helmert for giving me the opportunity to do my bachelor thesis in the artificial intelligence research group. Last but not least my girlfriend Béatrice Gauvain for her moral support when the bugs kept me working for hours longer than anticipated.

Abstract

Pattern Databases are admissible abstraction heuristics for classical planning. In this thesis we are introducing the *Boosting* processes, which consists of enlarging the pattern of a Pattern Database P, calculating a more informed Pattern Database P' and then min-compress P' to the size of P resulting in a compressed and still admissible Pattern Database P'' . We design and implement two boosting algorithms, *Hillclimbing* and *Randomwalk*.
We combine pattern database heuristics using five different cost partitioning methods. The experiments compare computing cost partitionings over regular and boosted pattern databases.
The experiments, performed on IPC (optimal track) tasks, show promising results which increased the coverage (number of solved tasks) by 9 for canonical cost partitioning using our Randomwalk boosting variant.

Table of Contents

1 Introduction

In Artificial Intelligence, Automated Planning is one of the major research areas. The main goal of Automated Planning is to find sequences of actions (plans) in a state space which is described by a planning formalism. In the planner *Fast Downward* by Helmert [5], the used input planning formalism is the *Planning Domain Definition Language (PDDL)*, which needs to be translated into a *Simplified Action Structures (SAS$^+$)* task. This translation step could be skipped by directly providing a SAS$^+$ task.

To perform a search in such a state space we need an admissible heuristic. For this thesis we are looking at abstraction heuristics, called Pattern Databases. This approach was introduced into automated planning by Edelkamp [2]. Pattern Databases abstract the original planning task by using projections. To perform the projection only a subset of the variables of the planning task are used and form an abstract planning task. To get better heuristics, the pattern database heuristics are combined using cost partitioning methods, first introduced by Katz and Domshlak [7]. Another existing concept is Pattern Database compression, which is used to compress a larger PDB of size M into a smaller PDB of size N. The *min-compression* described by Helmert et al. [6] is used in this thesis. It is a lossy compression method, which combines several entries of the larger PDB into a single value in the smaller PDB.

The new concept introduced in this thesis is the process of *Boosting*. This process combines several steps, like increasing the pattern of a given Pattern Database to calculate a related, but bigger Pattern Database and then min-compress it to the original size.

One goal of this thesis is to design and evaluate such boosting algorithms. We implement several boosting variants in the Fast Downward planner. The boosting is then interleaved with already implemented and to-be-implemented cost partitioning methods. All implementations are compared using the A^* search algorithm and a large collection of planning competition benchmarks (IPC Optimal Tracks).

2 Background

In this section, we define the concepts and notation used throughout this thesis.

2.1 Planning

The definitions in this section follow the notation used in the Foundations of Artificial Intelligence lecture[1].

Definition 1. *A state space is a transition system, which is defined by a 6-tuple:*

$$\mathcal{S} = \langle S, L, c, T, s_0, S_* \rangle$$

where S is a finite set of states, L is a finite set of labels, $c : L \mapsto \mathbb{R}_0^+$ a cost function, $T \subseteq S \times L \times S$ a set of transitions, $s_0 \in S$ is the start state and $S_* \subseteq S$ is the set of goal states. A transition is a triple $t = \langle s, a, s' \rangle$, where $s, s' \in S$ and $a \in L$.

A Simplified Action Structures (SAS$^+$ task) is a common planning formalism introduced by Bäckström and Nebel [1] in 1995.

Definition 2. *A SAS$^+$ planning task is a 5-tuple:*

$$\Pi = \langle V, dom(v), I, G, A \rangle$$

where V is a finite set of state variables, $dom(v)$ is the non-empty and finite domain of each $v \in V$, I the initial state (total assignment), G a partial assignments, A a finite set of actions a. An action a has three components, $\text{pre}(a)$ *a set of preconditions (partial assignments),* $\text{eff}(a)$ *a set of effects (partial assignments) and $c(a) \in \mathbb{N}_0$.*

Another concept is the induced state space.

Definition 3. *A SAS$^+$ task Π induces the state space $\mathcal{S}(\Pi) = \langle S, A, c, T, s_0, S_* \rangle$, with:*
S total assignments of V according to dom, L the action defined in Π, action costs as defined in Π, there is at $= \langle s, a, s' \rangle \in T$ iff $\text{pre}(a)$ *complies with s, s' complies with* $\text{eff}(a)$ *for all variables in* eff, *s' complies (two states comply, if they agree on the variables they are defined on) with s for all other variables. Initial state $s_0 = I$, goal states $s \in S_*$ for state s iff G complies with s.*

[1] https://dmi.unibas.ch/de/studium/computer-science-informatik/lehrangebot-fs20/lecture-foundations-of-artificial-intelligence/

2.2 Pattern Databases

The definitions in this section are adapted from Sievers et al. [11].

The goal distance $h^*(s)$ of a state s is the cost of a shortest path from s to any goal state in the induced state space. If there is no path to the goal from a certain state s then $h^*(s) = \infty$

A pattern is a subset of the variables $P \subseteq V$, for planning we assume the variables are numbered ($V = \{v_1, \ldots, v_k\}$ and their corresponding domain $\mathcal{D}_{v_i}$ is abbreviated as $\mathcal{D}_i$).

An abstraction of an induced state space considers the states which agree on all variables in P as equivalent.

A Pattern Database (PDB) is defined by a pattern and based on an abstraction of the induced state space of the SAS$^+$ task Π. It is a look-up table containing all goal distances for all states in the abstract task and implemented as a one-dimensional array of size $N = \prod_{i=1}^{k} |\mathcal{D}_i|$. The loopup is performed by using a perfect hashing function from states to indices $\{0, \ldots, N-1\}$ (ranks). This function is called a ranking function and its inverse is called an unranking function.

The ranking function used is given by $rank(s) = \sum_{i=1}^{k} N_i s[v_i]$, where the coefficients are defined as $N_i = \prod_{j=1}^{i-1} |\mathcal{D}_j|$.

In this thesis compression refers to PDB compression, which is needed to reduce the entries of the goal distances values of a larger PDB. We are using *min-compression* as described in Helmert et al. [6]. The following description is based on that paper.

We need to compress a PDB of size M to a PDB of size N, where $M \geq N$. Both PDBs have goal distances arrays T_P with corresponding sizes. The compression can be performed by using a compression function $comp : \{0, \ldots, M-1\} \rightarrow \{0, \ldots, N-1\}$. The compressed goal distances table with N entries can be defined as $T_P^{comp}[i] = min_{r \in comp^{-1}(i)} T[r]$.

2.3 Cost Partitioning (CP)

So far we have only introduced single PDBs to use as a heuristic. Due to the loss of most of the information about a task Π, when using a projection, the heuristic is not ideal. This leads to the use of collections of PDBs used as a combined heuristic. The issue is how to combine heuristics in a way, such that they remain admissible. *Cost Partitioning* is a method that can be applied to derive admissible heuristics for state-space search problems in general [9].

Definition 4 (Seipp et al. [9])**.** *Let $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ be a tuple of admissible heuristics for a regular state space $\mathcal{S} = \langle S, A, c, T, s_0, S_* \rangle$. A cost partitioning over $\mathcal{H}$ is a tuple $\mathcal{C} = \langle c_1, \ldots, c_n \rangle$ of general cost functions whose sum is bounded by c: $\sum_{i=1}^{n} c_i(a) \leq c(a)$ for all $a \in A$. The cost-partitioned heuristic h^c is defined as $h^c(s) := \sum_{i=1}^{n} h_i^{c_i}(s)$.*

2.3.1 Canonical CP

The canonical heuristic allows to combine several pattern database heuristics. It was first introduced by Haslum et al. [4].

Definition 5 (Seipp et al. [9])**.** *Let $\mathcal{H}$ be a tuple of admissible heuristics for regular state space $\mathcal{S}$, and let MIS be a set of all maximal (w.r.t. set inclusion) subsets of $\mathcal{H}$ such that all heuristics in each* *subset are independent. The canonical heuristic in state $s \in S$ is*

$$h^{CAN}(s) = \max_{\sigma \in MIS} \sum_{h \in \sigma} h(s)$$

2.3.2 Greedy Zero-One CP

Zero-One cost partitioning (ZOCP) was introduced by Haslum et al. [3]. All of the costs of an action are assigned to only one of the heuristics in the cost-partitioning and 0 for all other heuristics. The variation of ZOCP used in Fast Downward is the Greedy Zero-One cost partitioning. For Greedy Zero-One CP and others we have to define $\mathcal{A}(h) = \{a \in A \mid a \text{ affects } h\}$ [9].

Definition 6 (Seipp et al. [9])**.** *Given a regular state space $\mathcal{S}$ and a set of admissible heuristics $\mathcal{H} = \{h_1, \ldots, h_n\}$ for $\mathcal{S}$, the set of orders of $\mathcal{H}$ ($\Omega(\mathcal{H})$) consists of all permutations of $\mathcal{H}$ i.e. all tuples of heuristics obtained by ordering $\mathcal{H}$ in any way. For a given order $\omega = \langle h_1, ..., h_n \rangle \in \Omega(\mathcal{H})$, the greedy zero-one cost partitioning is the tuple $\mathcal{C} = \langle c_1, \ldots, c_n \rangle$, where*

$$c_i(a) = \begin{cases} c(a) & \textit{if } a \in \mathcal{A}(h_i) \textit{ and } a \notin \bigcup_{j=1}^{i-1} \mathcal{A}(h_j) \\ 0 & \textit{otherwise} \end{cases}$$

for all $a \in A$. We write h_ω^{GZOCP} for the heuristic that is cost-partitioned by greedy zero-one cost partitioning for order ω.

2.3.3 Saturated CP

In contrast to Greedy Zero-One cost partitioning, Saturated Cost Partitioning (SCP), introduced by Seipp and Helmert [8], does not assign the full costs to the first heuristic affected by a certain action. The heuristic using greedy Zero-One cost partitioning may only need a fraction of those costs. SCP only assigns the costs used by the heuristic and offers the remaining costs to the next heuristic.

Definition 7 (Seipp et al. [9])**.** *Let $\mathcal{S}$ be a regular state space and $\mathcal{H}$ be a set of admissible heuristics. Given an order $\omega = \langle h_1, \ldots, h_n \rangle \in \Omega(\mathcal{H})$, the saturated cost partitioning $\mathcal{C} = \langle c_1, \ldots, c_n \rangle$ and the remaining cost functions $\langle \bar{c}_1, \ldots, \bar{c}_n \rangle$ are defined by:*

$$\begin{aligned} \bar{c}_0 &= c \\ c_i &= saturate(h_i, \bar{c}_{i-1}) \\ \bar{c}_i &= \bar{c}_{i-1} - c_i \end{aligned}$$

We write h_ω^{SCP} for the heuristic that is cost-partitioned by saturated cost partitioning by order ω.

The mentioned *saturate* function takes an abstraction heuristic h_i and calculates the saturated costs for all actions $a \in A$, which is equal to the maximum over all $h(s) - h(s')$ for all transitions $s \to s'$ in the abstract state space induced by a.

2.3.4 Uniform CP

Another cost partitioning method is Uniform Cost Partitioning (UCP). The idea is to assure admissibility by uniformly distributing the costs for each action a to all heuristics affected by action a.

Definition 8 (Seipp et al. [9]). *Given a regular state space $\mathcal{S}$ and a tuple of admissible heuristics $\mathcal{H} = \langle h_1, \dots, h_n \rangle$, the uniform cost partitioning is the tuple $\mathcal{C} = \langle c_1, \dots, c_n \rangle$, where for all $a \in A$*

$$c_i(a) = \begin{cases} \frac{c(a)}{|\{h \in \mathcal{H} | a \in \mathcal{A}(h)\}|} & \textit{if } a \in \mathcal{A}(h_i) \\ 0 & \textit{otherwise} \end{cases}$$

We write h^{UCP} for the heuristic that is cost-partitioned by uniform cost partitioning.

2.3.5 Opportunistic Uniform CP

Similar to ZOCP, UCP has the same issue with heuristics not using the full offered costs. To prevent this problem the Opportunistic Uniform Cost Partitioning (oUCP) uses the saturate function as used in SCP.

Definition 9 (Seipp et al. [9]). *Let $\mathcal{S}$ be a state-space and $\mathcal{H}$ be a set of admissible heuristics. Given an order $\sigma = \langle h_1, \dots, h_n \rangle \in \Omega(\mathcal{H})$, the opportunistic uniform cost partitioning $\mathcal{C} = \langle c_1, \dots, c_n \rangle$, the remaining cost functions $\langle \bar{c}_0, \dots, \bar{c}_n \rangle$ and the offered cost functions $\langle \widetilde{c}_1, \dots, \widetilde{c}_n \rangle$ are defined by*

$$\bar{c}_0 = c$$
$$\widetilde{c}_i(a) = \begin{cases} \frac{\bar{c}(a)_{i-1}}{|\{h \in \mathcal{H} | a \in \mathcal{A}(h)\}|} & \textit{if } a \in \mathcal{A}(h_i) \\ 0 & \textit{otherwise} \end{cases}$$
$$c_i = saturate(h_i, \widetilde{c}_i)$$
$$\bar{c}_i = \bar{c}_{i-1} - c_i$$

We write h^{OUCP}_{ω} for the heuristic that is cost-partitioned by opportunistic uniform cost partitioning for order ω.

3 Combining PDB Compression and Cost Partitioning

During the work on this bachelor thesis, there were several components to be implemented. For the implementation we used the Fast Downward planning system by Helmert [5], which combines a translator and a search component. Pattern Databases are part of heuristics and therefore used in the search component. This thesis builds on several pre-existing classes and cost-partitioning methods. We divide this chapter into two main parts. First the boosting of Pattern Databases and second the different cost partitioning methods.
In Fast Downward the general process of creating a cost partitioned Pattern Database collection is divided into two phases, the *Pattern Generation* and the *Pattern Database Collection Creation*. Our new boosting approach hooks into the *Pattern Database Collection Creation*.

3.1 Boosting

This part of the thesis describes the boosting process in detail and the algorithms chosen to perform the boosting. We define *Boosting* as a process which first increases the pattern of a given PDB, calculates a new, larger PDB and min-compress the larger PDB to be the same size as the initial PDB. Due to the min-compression, the PDB is at least as good or better than the initial PDB and still admissible.

Problem description: Given a Pattern Database P with the pattern p and the SAS$^+$ task Π with the state variables V. Boost P to a new PDB P' such that the heuristic values for P' are increased. To solve this problem we need several general components.

- Finding variables $v \in V$, where $v \notin p$ to improve the pattern p. For this task we have used the causal graph (CG) of Π to determine the causally relevant variables to the current pattern p by collecting all successors in the CG of the all variables in p. This is visualized in algorithm 1.
- Create a larger Pattern Database P' given the same parameters as for the creation of P, but change the pattern to p'. This step uses the efficient Pattern Database construction algorithm by Sievers et al. [11]. In Algorithms 4 and 5 this function is called *createPDB*.

Algorithm 1: findRelevantMissingVariables

```
Data: Pattern p, Task $\Pi$
$CG \longleftarrow getCausalGraph(\Pi)$;
$relevant\_vars \longleftarrow \emptyset$;
for $v \in p$ do
    $relevant\_vars \cup getSuccessors(CG, v)$;
end
$relevant\_vars \longleftarrow relevant\_vars \setminus p$;
return $relevant\_vars$;
```

- Min-Compression of P' given the original PDB P. We can compress P' with size M to a PDB P'' with size N, where N is equal to the size of the original PDB P. For the implementation we use a trick to decrease the difficulty of min-compressing the larger PDB P''. The PDB goal distances look-up table is constructed in the variable order provided by the pattern. For this reason we are prepending the variables for the construction of p' to the previous pattern p. Now the entries containing values of the newly added variables are next to each other and form a block of size $AddedSize = \prod_{i=1}^{k} |\mathcal{D}_i|$, where $k =$ *Index of the last additional variable*. Then we can compress all entries linear by determining AddedSize and use it to find the blocks distance entries which should be min-compressed to a new distance entry in P''. The result of this operation is a compressed Pattern Database P'', containing the original information of P and the compressed goal distance entries from P'. The pseudo code implementation is shown in Algorithm 2.

Algorithm 2: minCompress

```
Data: PatternDatabase $P$, PatternDatabase $P'$, Integer domainSizeDifference
$distances\_compressed \longleftarrow$ vector<integer>;
$distances\_large \longleftarrow P'.getDistances()$;
for $i \leftarrow 0$ to $distances\_large.size()$ by domainSizeDifference do
    $min\_distance \longleftarrow \infty$;
    for $j \leftarrow 0$ to domainSizeDifference by 1 do
        if $distances\_large[i + j] < min\_distance$ then
            $min\_distance \longleftarrow distances\_large[i + j]$;
        end
    end
    $distances\_compressed.push\_back(min\_distance)$;
end
return $P''$ (all properties from $P$, but $distances = distances\_compressed$);
```

- Compare P and P''. The difference between the two Pattern Databases are only their distance entries. In this step we are comparing all entries and count the amount of increased values in P''. Due to *min-compression*, we can guarantee the values would only be greater or equal to the original value. This results in an indicator $\frac{\text{improved\_entries}}{\text{total\_entries}}$. In other algorithms this function is called *comparePDBS* and used as an evaluation function.

Those building blocks are part of the concrete algorithms to boost a given Pattern Database. In the thesis we have designed two approaches. In this chapter we will introduce the method and in Chapter 4 we will test and discuss the results.

3.1.1 Hillclimbing algorithm

The first idea for a boosting algorithm was to use Hillclimbing for finding an improved Pattern Database P'. Hillclimbing is a local search algorithm, which searches in the neighbourhood of a certain configuration. All neighbours are evaluated and a set of suitable candidates is chosen to proceed into the next iteration. In our case we evaluate all neighbours with an additional causally relevant variable added to the pattern and chose only the best one as the new base for the next iteration. In the implementation we have added command line parameters to limit the iterations (*max_iterations*) and a minimum improvement percentage as an early break condition (*min_impr_compression*). The Hillclimbing algorithm is shown in Algorithm 4.

Algorithm 3: computeCandidates

```
Data: Pattern $p$, Task $\Pi$
$vars \longleftarrow findRelevantVariables(p, \Pi)$;
$candidates \longleftarrow \emptyset$;
for $v \in vars$ do
    $candidates \longleftarrow candidates \cup (p \cup \{v\})$;
end
return $candidates$
```

Algorithm 4: boostHillclimbing

```
Data: PatternDatabase $P$, Pattern $p$, Task $\Pi$, operator_costs
$min\_improvement \longleftarrow \langle$global value$\rangle$;
$max\_iterations \longleftarrow \langle$global value$\rangle$;
$iteration \longleftarrow 0$;
$best\_pdb \longleftarrow P$;
$best\_pattern \longleftarrow p$;
$best\_improvement \longleftarrow 0$;
$candidates \longleftarrow computeCandidates(p, \Pi)$;
while $iteration < max\_iterations$ do
    if $best\_improvement < min\_improvement$ AND $iteration > 0$ then
        return $best\_pdb$;
    end
    for $candidate \in candidates$ do
        $larger\_pdb \longleftarrow createPDB(\Pi, candidate, operator\_costs)$;
        $domain\_size \longleftarrow determineDomainSizeDifference(p, candidate, \Pi)$;
        $compressed\_pdb \longleftarrow minCompress(P, larger\_pdb, domain\_size)$;
        $improvement\_score \longleftarrow comparePDBS(P, compressed\_pdb)$;
        if $improvement\_score > best\_improvement$ then
            $best\_improvement \longleftarrow improvement\_score$;
            $best\_pdb \longleftarrow compressed\_pdb$;
            $best\_pattern \longleftarrow candidate$;
        end
    end
    $candidates \longleftarrow computeCandidates(best\_pattern, \Pi)$;
    $iteration \longleftarrow iteration + 1$;
end
return $best\_pdb$;
```

One of the issues with the Hillclimbing approach is the duration of the whole boosting process, even with a low $max\_iterations$ limit. This triggered the need for an additional approach. Which will be introduced in the next subsection.

3.1.2 Randomwalk algorithm

The main need for this approach was to make the boosting faster. The idea was to define a maximal amount of states for a single Pattern Database P and randomly add one of the possible variables to the current pattern p until we reach the defined limit. This limit is passed as a command line parameter. The intention was to reduce the time needed for the boosting process.
Compared to the Hillclimbing approach, the Randomwalk only calculates one Pattern Database during the boosting process instead of several smaller ones during the Hillclimbing. The Randomwalk is shown in Algorithm 5.

Algorithm 5: boostRandomwalk

```
Data: PatternDatabase $P$, Pattern $p$, Task $\Pi$, operator_costs
$num\_states \longleftarrow getSize(P)$;
$max\_states\_options \longleftarrow \langle$global value$\rangle$;
$relevant\_variables \longleftarrow findRelevantVariables(p, \Pi)$;
$new\_pattern \longleftarrow p$;
if $relevant\_variables = \emptyset$ then
    return $P$;
end
while $num\_states < max\_states\_options$ do
    $candidate \longleftarrow$ pick random $v \in relevant\_variables$;
    if $domainSize(candidate) * num\_states < max\_states\_options$ then
        $num\_states \longleftarrow num\_states * domainSize(candidate)$;
        $new\_pattern \longleftarrow new\_pattern \cup candidate$;
    end
    $relevant\_variables \longleftarrow relevant\_variables \setminus \{candidate\}$;
    if $relevant\_variables = \emptyset$ then
        break;
    end
end
$larger\_pdb \longleftarrow createPDB(\Pi, new\_pattern, operator\_costs)$;
$domain\_size \longleftarrow determineDomainSizeDifference(p, candidate, \Pi)$;
$compressed\_pdb \longleftarrow minCompress(P, larger\_pdb, domain\_size)$;
if $comparePDBS(P, compressed\_pdb) > 0$ then
    return $compressed\_pdb$;
else
    return $P$;
end
```

Both boosting algorithms are in a combined class and can be chosen by the *compr_algo* CLI parameter with either *HILLCLIMBING* or *RANDOMWALK* as values. In the following Cost Partitioning methods we ommit a concrete implementation and call the function $computeBoostedPDB$.

3.2 Cost Partitionings

Now we have the ability to boost a given Pattern Database P and get an improved or equal PDB P'. As mentioned in Chapter 2, a single PDB is not a really good heuristic and should be interleaved by using Cost Partitionings (CPs). During the thesis five different CPs were implemented or adapted to support the usage of compressed PDBs. A crucial implementation detail for all cost partitioning methods was to calculate the cost functions using the original PDB and not the boosted PDB.
From the five CPs two were already implemented in Fast Downward (Canonical CP and greedy Zero-One CP), the remaining three had do be implemented from scratch. The five CPs can be divided into three categories. The first category includes only the Canonical CP. Fast Downward already provides a collection of PDBs, when using the canonical heuristic. We only have to boost them if we want to, which is implemented by using the *compress_pdbs* flag. The straightforward boosting is shown in Algorithm 6. The PDB returned by the *computeBoostedPDB* function modifies the previous PDB in-place.

```
Algorithm 6: boostCANCollection
Data: PatternCollection $pc$, PatternDatabaseCollection $PDC$, Task $\Pi$
for $p, P \in pc, PDC$ do
    $p \longleftarrow computeBoostedPDB(p, P, \Pi)$;
end
```

The next category includes greedy Zero-One CP and Uniform CP. In both cases the computation of the Pattern Databases can only take place during computing the cost partitioning, because all PDBs have to be computed on different cost functions. The same cost function of a calculated PDB is then used during the boosting. Algorithm 7 describes the greedy Zero-One CP. The algorithm for UCP is not added, due to a lot similarities.

```
Algorithm 7: ZeroOnePDBs
Data: PatternCollection $pc$, Task $\Pi$, boolean $compress\_pdbs$
$remaining\_operator\_costs = vector$<$integer$>;
$pattern\_databases = vector$<$PatternDatabase$>;
for $a \in A$ (of $\Pi$) do
    $remaining\_operator\_costs.push\_back(a.getCosts())$;
end
for $p \in pc$ do
    $pdb \longleftarrow createPDB(\Pi, p, remaining\_operator\_costs)$;
    if $compress\_pdbs$ then
        $pdb \longleftarrow computeBoostedPDB(p, pdb, \Pi, remaining\_operator\_costs)$;
    end
    for $a \in A$ (of $\Pi$) do
        if $pdb.isOperatorRelevant(a)$ then
            $remaining\_operator\_costs[a] = 0$;
        end
    end
    $pattern\_databases.push\_back(pdb)$;
end
return $pattern\_database$
```

The last category consists of the two Cost Partitioning (SCP and OUCP) using the saturated cost function. In the thesis we use the *calculateUsedCost* function (Algorithm 8) as a part of the saturate function, which calculates the effectively needed costs by the transitions. For showing the CP and boosting overlap, SCP (Algorithm 9) was chosen for the report. For the *saturate* function, we need information about the actions and their transitions in the abstract state space. This transition system is only calculated on the fly during PDB construction. To implement this functionality, the transition system had to be calculated and stored for later use. Due to similarities, the oUCP algorithm is not listed in the thesis.

Algorithm 8: calculateUsedCosts

```
Data: PatternDatabase $P$, Task $\Pi$, boolean compress_pdbs
$used\_costs = vector<integer>$;
$distances \longleftarrow P.getDistances()$;
$operator\_transitions \longleftarrow P.getTransitions()$;
for $a \in A$ (of $\Pi$) do
    if $P.isOperatorRelevant(a)$ then
        $max \longleftarrow 0$;
        for Transition $s \rightarrow s' \in operator\_transitions[a]$ do
            $value \longleftarrow distances[s] - distances[s']$;
            $max \longleftarrow max(value, max)$;
        end
        $used\_costs[a] \longleftarrow max$;
    else
        continue
    end
end
return $used\_costs$;
```

Algorithm 9: SaturatedPDBs

```
Data: PatternCollection $pc$, Task $\Pi$, boolean compress_pdbs
$remaining\_operator\_costs = vector<integer>$;
for $a \in A$ of($\Pi$) do
    $remaining\_operator\_costs.push\_back(a.getCosts())$;
end
$pattern\_databases = vector<PatternDatabase>$;
for $p \in pc$ do
    $pdb \longleftarrow createPDB(\Pi, p, remaining\_operator\_costs)$;
    $used\_costs \longleftarrow calculateUsedCosts(p, \Pi)$;
    if compress then
        $pdb \longleftarrow computeBoostedPDB(p, pdb, \Pi, remaining\_operator\_costs)$;
    end
    $remaining\_operator\_costs \longleftarrow remaining\_operator\_costs - used\_costs$;
    $pattern\_databases.push\_back(pdb)$;
end
return $pattern\_database$
```

We additionally consider limiting the size of pattern databases. Which includes limits for a single pattern database and for collections. This is not shown in the the algorithms for clarity. General idea: Whenever a PDB has to be created, we make sure that it is not created if that would violate these limits. A comparision between a not limited and a limited approach will be discussed in Chapter 4.

4 Experiments

4.1 Setup

For the experimental part of the thesis, we used the Downward Lab testing environment [10] to define and run several configurations. The benchmark domains[2] used are a collection of domains from the International Planning Competition (optimal track). All tests were performed on the High Performance Computing cluster of sciCORE, the center of competence for scientific computing at the University of Basel. The CPU used for the tests was an Intel Xeon Silver 4114 2.2GHz (10 cores/20 threads). The tests had several parameters in common:

- Time limit: 30 minutes
- Memory limit: 3.5 GiB RAM
- Search algorithm: A*

For all five cost partitioning methods we ran configurations to compare the boosting methods (with different parameters) and to compare boosting to the not-boosted base case. Additionally for each cost partition methods we compared different pattern generation algorithms listed in Table 4.1.

Table 4.1: Used pattern generator methods.
Bold, used in experiments and discussed. Brackets, used but not discussed.

Pattern Generator	**CAN CP**	**gZOCP**	**SCP**	**UCP**	**oUCP**
Hillclimbing	**Yes**	No	**Yes**	**Yes**	**Yes**
Genetic	No	**Yes**	No	No	No
Systematic(2)	No	No	**Yes**	(Yes)	(Yes)
Systematic(3)	No	No	(Yes)	(Yes)	(Yes)

For Canonical CP and Greedy Zero-One CP, the pattern generators were fixed to those commonly used for them, which is Hillclimbing and Genetic respectively. For the remaining Cost Partitionings, we reduce the scope to the best of the three initially tested generators.
For the boosting methods we have different parameters to influence the behaviour.

[2] https://github.com/aibasel/downward-benchmarks

- Both boosting methods: Limited PDB/PDB collection size. Either unlimited or 2M PDB and 20M PDB collection size. As mentioned in Chapter 3.
- Hillclimbing (Both parameters are introduced in Section 3.1.1):
  - *minimum improvement*: Threshold to proceed with HC, rpossible range 0.0 to 1.0.
  - *maximum iterations*: Iterations for HC used range 1 to 3.

For the tables we use this syntax to abbreviate certain parameter combinations: RND or HC for the boosting method, if HC the number of max iterations is appended, e.g. HC3 is three iterations. Limited is indicated by a L, e.g. HC2L is HC 2 iterations and limited. Lastly, the minimum improvement is indicated by an underscore and the percentage HC1L_0.25 for 25 percent.
The metrics and their abbreviations used for comparisons are the following:

- Coverage (Cov): Number of solved tasks. Larger is better.
- Expansions until last jump (Exp): Expansions during search, exluding last f-layer. Smaller is better.
- Out Of Memory (OOM): Sum of tasks resulting in a search out of memory error
- Out of Time (OOT): Sum of task resulting in a search out of time error
- Total time (Time): Overall time (geometric mean), includes translation. Smaller is better.

4.2 Results

First we are looking at Canonical CP, the data is shown in Table 4.2. The first indication is, the baseline (only using Canonical CP, without any boosting) takes less time compared to all boosted approaches. This implies, the time spent for boosting is not amortized during the search. Another observation is a reduction in the *Expansions until last jump* attribute for the experiments using boosting. Figure 4.1 shows a scatter plot for expansions (Baseline versus RNDL). The last observation should be treated carefully due to different coverages and therefore inherently different expansion counts. For example, one solved problem in the baseline case could be the primary source of total expansions and skew the results. When looking at a combination of the coverage and the expansions, the boosted approaches with limited PDB size have lower expansions even with more solved problems. As an example we compare the baseline (cov 919, exp 1.02 billion) with Randomwalk (cov 928, exp 801 million).
For Canonical CP we can conclude, limiting the PDB/PDB collection size increases the coverage and is better than the not limited variants. It reduces the overall time significantly, this reduces the *Search Out of Time* as well and improves the coverage. In the limited case, we have a order (based on coverage) *Baseline* $< HC1L \leq HC3L < HC2 < RNDL$. The iteration count for the Hillclimbing without limit decrease the coverage due to the time constraint. This effect is also visible when using the limit of the PDB size, but it is not significant.
For CAN CP and Greedy Zero-One CP we performed an additional experiment to compare the influence of the threshold (*minimum improvement*, Section 3.1.1) on the result. Table 4.3 shows the data for Canonical CP. From our data we can not make a clear statement. One observeration is that a

Table 4.2: CAN CP - Unlimited PDB Size vs Limited PDB Size

	Baseline	HC1	HC1L	HC2	HC2L	HC3	HC3L	RND	RNDL
Cov	919	910	923	909	924	900	923	**928**	**928**
OOM	685	634	683	**616**	680	635	681	674	675
OOT	204	264	**202**	283	204	273	204	206	205
Exp	1021M	879M	918M	858M	915M	841M	915M	**802M**	802M
Time	**2.79**	4.67	3.37	5.07	3.39	5.22	3.42	5.73	5.60

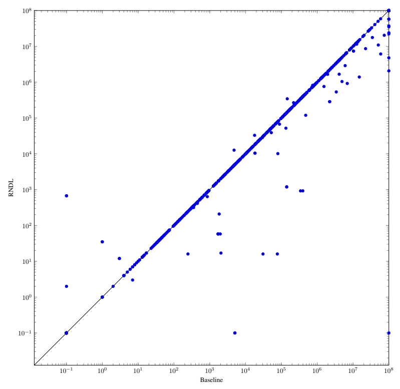

Figure 4.1: CAN CP - Expansions - x-axis=Baseline, y-axis=RNDL

Table 4.3: Canonical CP - minimum improvement comparison, the value after the underscore is the used percentage

	Baseline	HC1L_5	HC1L_10	HC1L_25	HC3L_5	HC3L_10	HC3L_25
Cov	919	923	923	923	923	**924**	**924**
OOM	685	**679**	680	682	680	680	680
OOT	204	206	205	**203**	205	204	204
Exp	1063M	954M	954M	954M	**952M**	**952M**	954M
Time	**3.30**	4.09	4.05	4.07	4.12	4.08	4.06

higher value results in only performing one iteration of the Hill Climbing. Additionally it seems only a few tasks profit from more than one iteration of the hill climbing, because it is too time consuming and results in less coverage. The data for Greedy Zero-One shows the same trend.

Randomwalk is as the name implies based on a random factor, for this reason we have performed one experiment for the CAN CP to test the consistency of the result. The same configuration was used as in the RND (limited) setup and performed ten times. The coverage average was 927.9 and a standard deviation of 0.31, which implies a stable result.

The second cost partitioning method we are looking at is Greedy Zero-One CP. For Greedy Zero-One CP and the remaining cost partitioning methods (UCP excluded), the order of the heuristics matters. In the thesis we have used only one order, the one provided by Fast Downward. The results are shown in Table 4.4. As in CAN CP, we have performed experiments using the algorithms with and without size restrictions. Like in the canonical case, the limit increased the coverage and decreased the overall time. The expansions are slightly higher, when comparing for example HC2 with HC2L. The higher expansions do not have an adverse effect in this case. Like before with CAN CP, the iterations of the unlimited HC approach decrease the coverage.

For Greedy Zero-One CP, the Hillclimbing boosting algorithm (using only one iteration) is better

Table 4.4: Greedy Zero-One CP - Unlimited PDB Size vs Limited PDB Size

	Baseline	HC1	HC1L	HC2	HC2L	HC3	HC3L	RND	RNDL
Cov	828	**837**	**837**	833	**837**	829	**837**	830	830
OOM	982	958	972	911	971	**891**	972	980	980
OOT	**0**	14	**0**	65	1	89	**0**	**0**	**0**
Exp	1915M	1644M	1662M	1595M	1651M	**1593M**	1651M	1785M	1785M
Time	**3.41**	5.05	4.60	5.73	4.74	6.25	4.78	5.90	5.87

Table 4.5: UCP- Hillclimbing pattern generation

	Baseline	HC1L	HC2L	HC3L	RNDL
Cov	**860**	858	858	858	851
OOM	738	723	722	722	**715**
OOT	**210**	227	228	228	242
Exp	1567M	1545M	1536M	**1535M**	1548M
Time	**3.41**	5.05	5.05	5.11	12.10

Table 4.6: oUCP- Hillclimbing pattern generation

	Baseline	HC1L	HC2	HC3L	RNDL
Cov	**817**	815	815	815	813
OOM	791	784	783	784	**766**
OOT	**194**	203	204	203	223
Exp	1492M	1469M	1461M	**1460M**	1471M
Time	**3.20**	4.60	4.60	4.66	11.36

than the Randomwalk and in summary the boosting is performing better than the baseline (genetic pattern generation). The bad performance for the Randomwalk was a bit surprising.

The data from CAN CP and Greedy Zero-One CP had shown the limited approach to be equal or better than the non-limited variants. Due to this finding the remaining three Cost Partitioning methods were only tested with a limit of 2M states for a PDB and 20M states for a PDB collection.

The next two Cost Partitionings UCP and oUCP will be discussed together. In both cases the boosting did not show a beneficiary effect and the coverage was reduced as shown in Tables 4.5 and 4.6. One impact was that boosting reduced the amount of amount of tasks with a *Search Out Of Memory* outcome, but increased the *Search Out Of Time*. The *Search Out Of Time* outcomes are a bit higher due to the time needed to perform the boosting. Hillclimbing was a bit better than Randomwalk, this is most likely due to the Randomwalk being more time consuming when the patterns get significantly bigger.

The last cost partitioning method is Saturated Cost Partitioning (SCP). Like in the last two cost partitioning methods, we have no commonly used pattern generation method. We tested three different ones (Hillclimbing, Systematic(2) and Systematic(3)) and picked the two different approaches for SCP. The results are shown in Tables 4.7 and 4.8. In summary, the experiments using a boosting algorithm have a better coverage and require more time. As in CAN CP and Greedy Zero-One CP, the iterations of the Hillclimbing boost seem not to have a significant effect. The expansions until last jump decrease significantely as shown in Figure 4.2 where the baseline (x-axis) is plotted against RNDL (y-axis) for the SYS(2) pattern generation. An interesting observation are the tasks with 0 expansions, which are mainly in the two *woodworking* domains.

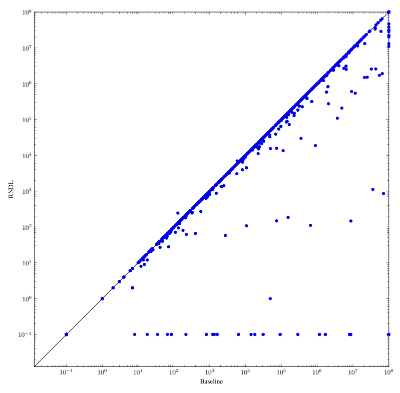

Figure 4.2: SCP - SYS(2) - Expansions - x-axis=Baseline, y-axis=RNDL

Table 4.7: SCP - Hillclimbing pattern generation

	Baseline	HC1L	HC2L	HC3L	RNDL
Cov	828	831	831	831	**835**
OOM	785	781	781	781	**772**
OOT	**195**	196	196	196	201
Exp	1127M	1010M	989M	986M	**756M**
Time	**3.12**	3.88	3.88	3.89	7.86

Table 4.8: SCP- Systematic(2) pattern generation

	Baseline	HC1L	HC2L	HC3L	RNDL
Cov	892	908	908	909	**915**
OOM	905	**845**	**845**	**845**	879
OOT	**11**	55	55	54	14
Exp	1757M	1503M	1483M	1467M	**1259M**
Time	**0.89**	1.81	1.83	1.86	8.75

A comparison between the two pattern generation methods (HC and SYS(2)) shows, that boosting a lot small patterns (as provided by SYS(2)) results in a bigger relative gain than boosting larger patterns (HC).

Overall most cost partionings yield better results when using Randomwalk as a boosting algorithm with limited Pattern Database sizes to limit the runtime. The best cost partitioning method was Canonical CP combined with limited Randomwalk. The mediocre performance of Saturated CP, Greedy Zero-One CP and UCP/oUCP was most likely due to only using one fixed order of the heuristics, which is a crucial aspect for those cost partitionings.

In our experiments for canonical CP, the domain *termes-opt18-strips* has shown a big improvement in terms of coverage (+3), this could be investigated to figure out the reason of the good effect. For Greedy Zero-One CP, the Hillclimbing increased the coverage of several domains by one. SCP using SYS(2) pattern generation had big impacts on the domains *termes-opt14-strips* (+4 coverage), *woodworking-opt08-strips* (+6 coverage) and *woodworking-opt14-strips* (+6 coverage). We do not have an explanation for those improvements other than the 0 expansions as seen in Figure 4.2.

5 Conclusion

In this thesis we have used Pattern Databases as a heuristic for an A^* search. We have introduced the process of *Boosting* and designed algorithms to perform this process. The resulting boosting algorithms were based on Hillclimbing and Randomwalk. The boosting was then combined and evaluated with cost partitioning methods like Canonical CP, Saturated CP and others.
Our experiments have shown really good results for the Canonical Cost Partitioning, where the coverage could be improved from 919 to 928 (+9 coverage). This result was achieved by using the Randomwalk based boosting algorithm with a limited Pattern Database size of 2M states and a collection size of 20M states.
The two Cost Partitioning methods UCP and oUCP were the only ones where the boosting did not show better results than the baseline. This was due to many tasks resulting in *Search Out Of Time*. In general time was a critical factor, which was the driving force for creating the Randomwalk variant for boosting. This is one topic which could be investigated further by trying to design faster algorithms for improving the boosting process.
There are several topics which could be further investigated. One really important topic would be the different orders for Greedy Zero-One CP, SCP, UCP and oUCP. We have only partitioned and boosted them in the order provided by Fast Downward. There we would need to find different methods to find good orders, boost and compare those orders. On top of finding a single order, several good orders could be combined.
In Chapter 4 we have mentioned some domains where the boosting had good results. Those domains could be analyzed and evaluated why the boosting is working particularly well there.

Bibliography

[1] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Comput. Intell.*, 11:625–656, 1995.

[2] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 84–90, 09 2001.

[3] Patrik Haslum, Blai Bonet, and Hector Geffner. New admissible heuristics for domain-independent planning. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1163–1168. AAAI Press / The MIT Press, 2005.

[4] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1007–1012. AAAI Press, 2007.

[5] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006.

[6] Malte Helmert, Nathan R. Sturtevant, and Ariel Felner. On variable dependencies and compressed pattern databases. In *Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS 2017, 16-17 June 2017, Pittsburgh, Pennsylvania, USA*, pages 129–133. AAAI Press, 2017.

[7] Michael Katz and Carmel Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, pages 174–181. AAAI, 2008.

[8] Jendrik Seipp and Malte Helmert. Diverse and additive cartesian abstraction heuristics. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. AAAI, 2014.

[9] Jendrik Seipp, Thomas Keller, and Malte Helmert. A comparison of cost partitioning algorithms for optimal classical planning. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, pages 259–268. AAAI Press, 2017.

[10] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

[11] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. Efficient implementation of pattern database heuristics for classical planning. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*, pages 105–111. AAAI Press, 2012.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Pascal Mafli

Matriculation number — Matrikelnummer
06-065-965

Title of work — Titel der Arbeit
Compressed Pattern Databases for Classical Planning

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, November 24th 2020

[illegible]

Signature — Unterschrift