University of Basel

1.6 Bit PDBs

Bachelor thesis

Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch/

Examiner: Examiner Prof. Dr. Malte Helmert
Supervisor: Supervisor Florian Pommerening

Sebastian Lukas
sebastian.lukas@stud.unibas.ch
17-050-691

17.10.2025

Acknowledgments

I would like to thank Prof. Dr. Malte Helmert for giving me the opportunity to write my Bachelor's thesis. I would also like to thank my supervisor Florian Pommerening for supervising this thesis, for his patience and his excellent advice and encouragement.

Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing center at University of Basel.

Abstract

In this thesis we describe a method to compress the heuristic values in a pattern database (PDB) down to 1.6 bit per state or 5 states per byte. The memory space taken up by the pattern database can be a limiting factor for PDB based heuristics as the full PDB needs to be kept in the memory during the search. This limits the memory space available for the search itself. We compare how heuristics using compressed and uncompressed pattern databases perform in order to investigate whether the memory freed up by the 1.6-Bit compression improves search coverage.

Table of Contents

1 Introduction

Automated Planning is a branch of Artificial Intelligence that involves choosing and ordering actions to achieve a specific goal [6]. A planning task consists of an initial state, a set of operators and at least one goal state. A solution to a planning task is a sequence of operators that lead to a goal state. This sequence of operators is called a plan.
A *heuristic search* has the goal of finding a solution to a planning task with the help of a heuristic function. This function assigns a heuristic value to each state in the search space to estimate the cost of reaching a goal state from that given state.

Pattern database heuristics [4] are abstraction heuristics for estimating the cost of the optimal plan from a given state in the planning task. They work by transforming the planning task into a smaller, simplified planning task and storing the optimal path cost for every state in a *pattern database* (PDB). The heuristic value of a state is the optimal path cost of the corresponding simplified state. *Canonical Heuristics* [7] combine multiple pattern databases to estimate plan costs.
The memory needed for a pattern database can be a limiting factor as it limits the size of the patterns that can be used. Additionally, the full pattern database needs to be kept in memory during the search of the planning task, reducing the available memory space for the search itself.

In this thesis, our goal is to implement a compressed *1.6-Bit pattern database* [1] which uses less memory than a regular PDB to store heuristic values of the same number of states. We then use the compressed PDBs in pattern database heuristics and canonical heuristics.

In the following chapters, we first describe abstraction heuristics, pattern databases and the 1.6-Bit compression, as well as PDB heuristics and canonical heuristics. We also discuss the implementation of the compressed pattern databases. Then we compare the compressed and uncompressed pattern databases, both in PDB heuristics and in canonical heuristics.

2 Background

2.1 Planning Tasks

A SAS$^+$ planning task [2] is a tuple $\Pi = \langle V, O, I, G \rangle$ which contains:

- A finite set of state variables V. Each variable $v \in V$ has a finite domain $dom(v)$. A partial state s assigns a value to some state variables $\text{Vars}(s) \subseteq V$. A state is a partial state where each state variable is assigned a value.
  A partial state s is consistent with partial state s' if all state variables that are assigned in both s and s' have the same value.
- A set of operators O. An operator consists of:
  - pre(o): a partial state denoting the precondition
  - eff(o): a partial state denoting the effect of the operator
  - c(o): the cost of the operator. If all operations in Π have the same cost of 1, the planning task has unit cost.

  In order for an operator $o \in O$ to be applicable in a state s, s must be consistent with pre(o). When the operator is applied, the state variables of the new state s' are assigned the same values as in eff(o) for all variables $v \in \text{Vars}(\text{eff}(o))$ and are kept the same for all other variables. If there is no assignment in eff(o), then the value in s' is the same as in s.
- An initial state I which is the starting configuration of the state variables.
- A goal G. A goal is a partial state and each state consistent with G is a goal state.

A solution to a planning task is also called a plan π from I. A plan is a sequence of operators $o_1, ..., o_n$ where o_1 is applicable a given state s and every operator o_i in the sequence is applicable in the state resulting from applying every previous operator $o_1, ..., o_{i-1}$ in sequence in s. The state resulting from applying all operators of π in sequence is consistent with G. An optimal plan π^* for a state s minimizes the total cost of the operators.
For example, in a planning task for solving an 8-tile puzzle (arrange eight tiles in a 3x3 grid in the right order) V would contain 9 state variables for the 8 tiles and the empty

tile, representing the tiles position on the grid. The operators in O are for moving a single tile. pre(o) assigns the position of the tile to be moved t and the position of the empty tile to the position t is to be moved, eff(o) assigning new positions to the moved tile and the empty tile, where their positions are swapped. The cost is always $c = 1$ (unit cost). In this representation the acts of moving tile 7 up from position 3 and moving it up from position 6 are considered two different operators with different preconditions and effects. I is the initial configuration of the puzzle and G assigns the target position for each tile.

2.2 State space

A state space is a tuple $\mathbb{S} = \langle S, L, \text{cost}, T, s_I, S_G \rangle$ which contains:

- A set of states S.
- A set of labels L.
- A cost function cost: $L \mapsto \mathbb{N}$, which assigns to each action $\ell \in L$ a nonnegative cost.
- A set of transitions $T \subseteq S \times L \times S$ containing tuples $\langle s, \ell, s' \rangle$.
- An initial state $s_I \in S$.
- A set of goal states $S_G \subseteq S$

A state space represents a directed graph, where the nodes are the states and the edges are the transitions. A solution of the state space is a path from the initial state to a goal state. The path can be represented by a sequence of transitions. If the path is optimal (it minimizes its cost), then is also an optimal solution to the state space.

A planning task $\Pi = \langle V, O, I, G \rangle$ induces a state space $\mathbb{S}(\Pi) = \langle S^\Pi, L^\Pi, \text{cost}^\Pi, T^\Pi, s_I^\Pi, S_G^\Pi \rangle$, where:

- S^Π is the set of all possible total assignments of the state variables V.
- $L^\Pi = O$.
- cost^Π with $\text{cost}^\Pi(o) = \text{c}(o)$.
- $T^\Pi = \{\langle s, o, s' \rangle | s, s' \in S^\Pi, o \in L^\Pi, o \text{ applicable in } s, s' \text{ successor of } s \text{ with } o\}$.
- The initial state $s_I^\Pi = I$.
- S_G^Π is the set of all states that are consistent with G.

A solution $\pi = o_1, ..., o_n$ for I to the planning task Π corresponds to a path in $\mathbb{S}(\Pi)$ from the initial state to a goal state where each operator o_i is replaced by a corresponding transition $t_i = \langle s, o, s' \rangle$ where s is the state resulting in applying all previous operators $o_1, ..., o_{i-1}$ in the sequence in I and s' is the state resulting from applying o_i in s. An optimal solution to Π is also a shortest path in $\mathbb{S}(\Pi)$.

In our 8-tile puzzle from before, $\mathbb{S}$ looks as follows: S contains all possible configurations of the tiles (there are $9! = 362'880$ configurations). L contains the operators from before. s_I is the initial configuration of the tiles and S_G contains a single goal state where each tile is at the right location.

2.3 Heuristic

A heuristic function $h : S \to \mathbb{N} \cup \{\infty\}$ for a state space $\mathbb{S} = \langle S, L, T, s_I, S_G \rangle$ assigns a nonnegative number or infinity to each state. It is used to estimate the cost of the optimal plan from state s to the nearest goal state. A heuristic function h^* that returns the optimal plan cost for each state is called optimal. If there is no solution for a state s, then $h^*(s) = \infty$. A heuristic is called admissible, if $h(s) \leq h^*(s)$ for all states $s \in S$.
A heuristic is called consistent, if $h(s) \leq h(s') + \text{cost}(\ell)$ for all transitions $s \xrightarrow{\ell} s'$ (the heuristic value of s is never larger than that of a successor state s' plus the transition cost from s to s').

2.4 Abstraction

An abstraction function $\alpha : S \to S'$ maps states of a state space $\mathbb{S} = \langle S, L, \text{cost}, T, s_I, S_G \rangle$ to a different set of states S'. The induced abstract state space $\mathbb{S}^\alpha = \langle S', L, \text{cost}, T', s'_I, S'_G \rangle$ contains:

- $T' = \{\langle \alpha(s), \ell, \alpha(s') \rangle \mid \langle s, o, s' \rangle \in T\}$
- $s'_I = \alpha(s_I)$
- $S'_G = \{\alpha(s) | s \in S_G\}$

Abstraction can remove the distinctions between certain states by mapping them onto the same abstract state but preserves the state space behavior as well as possible. The resulting abstract state space $\mathbb{S}^\alpha$ is usually smaller than $\mathbb{S}$.

An abstraction heuristic uses the optimal solution cost in the abstract state space to estimate the concrete solution cost ($h^\alpha(s) = h^*_{\mathbb{S}^\alpha}(\alpha(s))$). If the abstract state space is smaller than the concrete state space, searching for the optimal solution in the abstract state space and calculating $h^\alpha(s)$ is easier than calculating $h^*(s)$.
All actions and costs in $\mathbb{S}$ are also in $\mathbb{S}^\alpha$ and for each transition $t_i = \langle s, \ell, s' \rangle \in T$ there is a corresponding transition $t'_i = \langle \alpha(s), \ell, \alpha(s') \rangle \in T'$. Thus each path (or sequence of transitions t_i) in $\mathbb{S}$ has a corresponding path in $\mathbb{S}^\alpha$ where each transition $t_i = \langle s, \ell, s' \rangle$ is replaced by its corresponding transition $t'_i = \langle \alpha(s), \ell, \alpha(s') \rangle$. Since all pairs of t_i and t'_i share the same labels ℓ and costs $\text{cost}(\ell)$, the total costs of the corresponding paths remain the same. In particular, for the optimal path in $\mathbb{S}$ from each state s to a goal state $s_G \in S_G$ and with total cost $h^*(s)$, there exists a corresponding sequence of transitions in $\mathbb{S}^\alpha$ from $\alpha(s)$ to $\alpha(s_G) \in S'_G$ with the same cost. These paths are solutions in the abstract state space and because the abstraction heuristic $h^\alpha(s)$ takes the optimal solution cost in the abstract

state space, $h^\alpha(s) \leq h^*(s)$. This means that the abstraction heuristic is admissible. Abstraction heuristics are also consistent:

- Consider a transition $t : s \xrightarrow{\ell} s'$.
- If there is no solution for $\alpha(s')$ in the abstract state space, then $h^\alpha(s') = \infty$ and $h^\alpha(s) \leq h^\alpha(s') + \text{cost}(\ell) = \infty$.
- If there exists a solution for $\alpha(s')$, then there also exists an optimal path $\pi^*_{S^\alpha}$ for $\alpha(s')$ with a total cost of $h^\alpha(s') = h^*_{\mathbb{S}^\alpha}(\alpha(s'))$.
  The sequence $\pi_{t'}$ for defined as $\alpha(s) = (t', \pi^*_{S^\alpha}$ for $\alpha(s'))$ with the abstract transition $t' : \alpha(s) \xrightarrow{\ell} \alpha(s')$ is an abstract solution for $\alpha(s)$ with a cost of $h^\alpha(s') + \text{cost}(\ell)$. If $\pi_{t'}$ for $\alpha(s)$ is an optimal solution for $\alpha(s)$, then $h^\alpha(s) = h^*_{\mathbb{S}^\alpha}(\alpha(s)) = \text{cost}(\pi_{t'}$ for $\alpha(s)) = h^\alpha(s') + \text{cost}(\ell)$. If $\pi_{t'}$ for $\alpha(s)$ is not an optimal solution, there exists a path from $\alpha(s)$ with a lower total cost and $h^\alpha(s) < \text{cost}(\pi_{t'}(\alpha(s))) = h^\alpha(s') + \text{cost}(a)$.

2.5 Pattern Databases

A pattern is a subset of state variables $P \subseteq V$ of a planning task $\Pi = \langle V, O, I, G \rangle$. The induced state space $\mathbb{S}(\Pi)$ can be abstracted by the abstraction function α^P that maps all concrete states in S that only differ in the state variables not in P onto the same abstract state. If $P \subset V$, then the abstract state space $\mathbb{S}^P(\Pi)$ is smaller than $\mathbb{S}(\Pi)$.

An example pattern for a 8-tile puzzle could be one where only tiles 3, 6, 7, 8, 9 and the empty tile are considered. The resulting abstract state space only has 60'480 possible states.

A pattern database (PDB) [4] stores the cost of the optimal plan for each state in $\mathbb{S}^P(\Pi)$ in a lookup table. A perfect hash function is used to map each state in the pattern space to a position in the lookup table [5].
Since the induced state space of a planning task Π tends to be too large to compute a PDB, the pattern database is constructed for an abstract state space of a more manageable size. The values from the PDB are used as an abstraction heuristic h^P.

2.5.1 Canonical Pattern Databases

A pattern database for a single pattern of 6 state variables with 10 possible values each needs to store the heuristic values of 10^6 possible states. To store three pattern databases for patterns of 2 state variables and 10 possible values each, only a total of $3 \cdot 10^2$ heuristic values need to be stored instead. Combining multiple smaller pattern databases in a single heuristic function allows the use of more state variables within the same memory limits as a single PDB heuristic. However, some accuracy is lost.

A set of disjoint patterns $A = \{P_1, ..., P_k\}$ is called additive, if no operator has an effect on variables of two different patterns P_i and P_j. A set containing only a single pattern is

additive. A is called a maximal additive subset of C if $A \subseteq C$ and there is no other additive subset B of C to which A is also a subset ($A \subset B \subseteq C$).
For a set of additive patterns A, each transition $s \xrightarrow{\ell} s'$ affects only a single pattern P_i and thus the sum of heuristic values of the other patterns in A remains the same between the states s and s': $\Sigma_{j \neq i} h^{P_j}(s) = \Sigma_{j \neq i} h^{P_j}(s')$. h^{P_i} is consistent and thus $h^{P_i}(s) \leq h^{P_i}(s') + \text{cost}(\ell)$ holds. Therefore, $h^{P_i}(s) + \Sigma_{j \neq i} h^{P_j}(s) \leq h^{P_i}(s') + \text{cost}(a) + \Sigma_{j \neq i} h^{P_j}(s) = h^{P_i}(s') + \text{cost}(\ell) + \Sigma_{j \neq i} h^{P_j}(s')$ also holds and thus, the additive heuristic $h^A_{additive} = \Sigma_{P \in A} h^P$ is also consistent.
For unit cost problems, h^* is equal to the lowest number of operators needed to form a solution in the state space $\mathbb{S}(\Pi)$ and there exists an optimal plan π with a sequence of operators of length h^*. For each pattern $P_i \in A$ there is a path π_{P_i} with length $h^{P_i}(s) \leq ||\pi_{P_i}|| < h^*(s)$ for any state s (where $||\pi_{P_i}||$ is equal to the number of operators in π_{P_i}) and which contains only the operators in π that affect the state variables in P_i. There are no operators affecting the variables of any two different patterns P_i and P_j. Because of this, the sequences π_{P_i} and π_{P_j} share no operators and all operators in the pattern sequences are also present in π. There can be operators in π that are neither in π_{P_i} nor in π_{P_j} if those operators affect state variables not present in either pattern. The sum of the length of the pattern sequences $||\pi_{P_i}|| + ||\pi_{P_j}||$ can't be larger than the length of π for any pair of additive patterns and from this follows $h^A_{additive}(s) = \Sigma^k_{i=1} h^{P_i}(s) = \Sigma^k_{i=1} ||\pi_{P_i}|| \leq h^*(s)$. Thus, $h^A_{additive}$ is admissible.

An admissible heuristic function h dominates another admissible heuristic function h' when $h(s) \geq h'(s)$ for all states $s \in S$. Due to the admissibility of h, $h'(s) \leq h(s) \leq h^*(s)$ holds. The difference between the heuristic values $|h(s) - h^*(s)|$ is smaller than or equal to that between $|h'(s) - h^*(s)|$ for any given state s. Therefore, h is a closer estimate of h^* than h'. The heuristic function $h(s) = \max(h^{P_1}(s), h^{P_2}(s))$, given two admissible PDB heuristic functions h^{P_1} and h^{P_2}, is also admissible and clearly dominates h^{P_1} and h^{P_2}. Likewise, the additive heuristic $h^A_{additive}$ dominates the PDB heuristics of each pattern $P_i \in A$.
Given a collection of patterns $C = \{P_1, ..., P_k\}$ and $MAS(C)$ being the collection of all maximal additive subsets of C, the canonical heuristic function of C is defined as

$$h^C_{canonical}(s) = \max_{A \in MAS(C)} \sum_{P \in A} h^P(s) = \max_{A \in MAS(C)} h^A_{additive}(s).$$

The canonical heuristic is admissible and consistent and dominates the additive heuristics $h^A_{additive}$ for every additive subset $A \in MAS(C)$ [7]. $h^C_{canonical}$ dominates $h = \max_{P \in C} h^P$ because $h^A_{additive}$ dominates h^P for each pattern $P \in A$. Thus, $h^C_{canonical}$ is the more accurate estimate of h^*, without sacrificing admissibility.

2.5.1.1 Constructing The Pattern Collection

The quality of the canonical heuristic function h^C depends on the pattern collection C. However, the number of possible pattern collections is too large to search for the optimal pattern collection. A local search for a good pattern collection candidate can still be performed [7].

A hill climbing algorithm can be used to search for a locally optimal solution. It works by starting from an initial pattern collection $C_{current} = C_0$ and then repeatedly choosing the best 'neighbor' of $C_{current}$ as the new $C_{current}$. Once there is no better neighbor than $C_{current}$, the algorithm stops as it has found a local optimum. A minimum improvement ϵ can be used to reduce the time needed to construct the pattern, at the cost of quality.
The initial pattern collection is a collection of patterns containing only a single goal variable $C_0 = \{\{v\} | v \in \text{vars}(G)\}$. A neighbor C' of a pattern collection C can be constructed by taking a single pattern $P_i \in C$ and a variable v not in P_i and adding a new pattern $P_{new} = P_i \cup \{v\}$ to the collection. Additionally, we only consider neighbors where constructing the PDBs would not exceed a given memory limit. The quality of the new neighbor is defined by the number of states in the state space, where the heuristic is improved ($h^{C'}(s) > h^C(s)$). Because the number of possible states in a state space is likely too large, only a random sample of states is checked. In every iteration of the hill climbing algorithm, the neighbor with the highest number of improved states is selected as the new current collection $C_{current}$.

2.5.2 1.6-Bit Pattern Databases

For certain planning tasks, we can compress the pattern databases [1].
In a unit cost planning task with an undirected search space the difference of the heuristic values $h^P(s)$ and $h^P(s')$ between a state s and a successor state s' is limited to $-1 \leq h^P(s) - h^P(s') \leq 1$. The PDB heuristic h^P is consistent and thus $h^P(s) \leq h^P(s') + \text{cost}(\ell)$ holds. Additionally, because the search space is undirected, if there is a transition $t = \langle s, \ell, s' \rangle$ there must also be a transition $t_{reverse} = \langle s', \ell', s \rangle$ and because of the unit cost constraint, $\text{cost}(a) = \text{cost}(\ell') = 1$. From transition t we get $h^P(s) \leq h^P(s') + 1$ and thus $h^P(s) - h^P(s') \leq 1$ and from $t_{reverse}$ we get $h^P(s') \leq h^P(s) + 1$ and thus $h^P(s') - h^P(s) \leq 1$. Therefore $h^P(s) - h^P(s') \geq -1$ holds.
If we know the heuristic value $h^P(s)$ of state s and the modulo three compressed heuristic value $h^P(s')\%3$ of a successor state s', the concrete heuristic value of s', $h^P(s')$, can be reconstructed. $h^P(s) - h^P(s')$ has three possible results: -1, 0 or 1. Thus, $h^P(s')$ given $h^P(s)$ also has three possible values: $h^P(s) - 1$, $h^P(s)$ or $h^P(s) + 1$. These three values modulo three are always different.

In a straight-forward implementation of PDB heuristics, the heuristic value for each state in a pattern database is stored as a 4-byte integer. For pattern databases with many states, the memory required to store such a lookup table can be limiting.
Modulo three compressed heuristic values $h^c(s)$ can have three possible values. m values from $\{0, 1, 2\}$ can be encoded in a single number between 0 and $3^m - 1$. Each value $h^c(s)$ is assigned an index j with $n = \Sigma_{j=0}^{m-1} h_j^c \cdot 3^j$ being the encoded value of the m compressed heuristic values and h_j^c being the modulo three compressed heuristic value $h^c(s)$ encoded in

n with index j. The value of $h^c(s)$ can be extracted from n with its index j:

$$\begin{aligned}\lfloor\frac{n}{3^j}\rfloor\%3 = \lfloor\Sigma_{k=0}^{m-1}h_k^c \cdot 3^k/3^j\rfloor\%3 = \lfloor\Sigma_{k=0}^{m-1}h_k^c \cdot 3^{k-j}\rfloor\%3 \\ = \lfloor\Sigma_{k=j+1}^{m-1}h_k^c \cdot 3^{k-j} + h_j^c + \Sigma_{k=0}^{j-1}h_k^c \cdot 3^{k-j}\rfloor\%3 \\ \text{with } h_k^c \cdot 3^{k-j} \in \mathbb{N}\forall k \geq j\end{aligned}$$

Because $\Sigma_{k=j+1}^{m-1}h_k^c \cdot 3^{k-j}$ and h_j^c are natural numbers, it does not matter, whether they are added before or after rounding down.

$$\lfloor\frac{n}{3^j}\rfloor\%3 = (\Sigma_{k=j+1}^{m-1}h_k^c \cdot 3^{k-j} + h_j^c + \lfloor\Sigma_{k=0}^{j-1}h_k^c \cdot 3^{k-j}\rfloor)\%3$$

The sum $\Sigma_{k=0}^{j-1}h_k^c \cdot 3^{k-j}$ is always positive and smaller than 1. This is because $\Sigma_{k=0}^{j-1}h_k^c \cdot 3^{k-j} \leq \Sigma_{k=0}^{j-1}2 \cdot 3^{k-j} = \frac{2}{3} + \frac{2}{9} + ... + \frac{2}{3^j} < 1$. Thus $\lfloor\Sigma_{k=0}^{j-1}h_k^c \cdot 3^{k-j}\rfloor = 0$.
Furthermore, $\Sigma_{k=0}^{j-1}h_k^c \cdot 3^{k-j} = 3 \cdot \Sigma_{k=0}^{j-1}h_k^c \cdot 3^{k-j-1}$ is divisible by three and thus:

$$\lfloor\frac{n}{3^j}\rfloor\%3 = (3 \cdot \Sigma_{k=j+1}^{m-1}h_k^c \cdot 3^{k-j-1} + h_j^c)\%3 = h_j^c\%3 = h_j^c$$

The largest base three number that fits in a byte ($2^8 = 256$ different values) is $3^5 = 243$. Using the method above, five compressed heuristic values can be encoded in one byte (which means that one heuristic value takes up $8/5 = 1.6$ bit of storage space).
In a pattern database a rank-function is used to assign each state s a unique index i. This index is used to find the location of the integer that stores the heuristic value of s in the PDB. In our compressed version of the PDB however, a single byte stores the compressed heuristic values of five states. To access h^c of s, both the location of the relevant byte i' as well as the index j of h^c within the byte are needed. i' and j can both be computed from i with $i' = \lfloor i/5\rfloor$ and $j = i\%5$.
In order to reconstruct the heuristic value for a state from h^c, the heuristic value of the predecessor state needs to be known.

3 Implementation

The Fast Downward planning system supports heuristics based on pattern databases as well as canonical pattern databases. Fast Downward also supports the hill climbing algorithm for constructing a pattern collection. We implemented 1.6 Bit compressed versions of these heuristics based on the uncompressed implementations.

3.1 PDB Heuristic

The heuristic based on a single PDB is created by first creating an uncompressed PDB based on a given pattern. In a second step, a compressed PDB is created with the uncompressed PDB as its input. The heuristic values stored in the PDB are stored in their modulo three compressed form as described in Chapter 2.5.2. The concrete heuristic value of the initial state gets stored separately. Once the compressed PDB is constructed, the uncompressed PDB can be discarded.
The PDB heuristic can only reconstruct the heuristic value of a state, if the concrete heuristic value of its predecessor state is known. During the search, the heuristic value of each already visited abstract state is stored for this information to be available. An alternative approach where the heuristic value is reconstructed from the value of the initial state and the path used to reach a state would also be possible, but retrieving a heuristic value with this method would be slower.

A pattern can for example be generated using a greedy pattern generator. It works by gradually adding variables to the pattern until either all variables are used in the pattern or until a given size limit for the PDB is reached. The order by which the variables are added to the pattern prioritizes goal variables.

3.2 Canonical PDB Heuristic

The heuristic based on multiple PDBs is created by first creating the full collection of uncompressed PDBs and then compressing each PDB individually. The pattern collection

used by the heuristic is created using a hill climbing algorithm.
The concrete heuristic values of each visited abstract state needs to be stored for each PDB, so that the values of the predecessor states can be retrieved.

4 Experiments

In order to evaluate the implementation of the 1.6-Bit pattern databases for the PDB heuristic and canonical heuristic, we conducted a series of experiment. In the following, the setup of these experiments will be described and then the results will be discussed.

4.1 Setup

There are two main questions we want to answer with our experiments: First, are there advantages in using the compressed PDBs for a PDB heuristic or canonical heuristic? Second, are bigger PDBs / PDB collections in a canonical heuristic better or worse suited for compression (or does PDB size not matter at all)?

All calculations for the experiments were performed on the sciCORE cluster of the University of Basel on the Fast Downward planner [8] using the Downward Lab package [9]. Each task ran on a single core of a 2 x 10 Core Intel Xeon Silver 4114 2.2 GHz Processor. For each task there was a memory limit of 3947 MiB and a time limit of 30 minutes.
A collection of ICP benchmark instances was used[1]. Only domains with an undirected search space were considered. We used a Python script to check for each operator o in the search space if there are inverse operators o_i^{-1} that can reverse all effects of o for each reachable state from which o can be applied. Mutex information was used to constrain the reachable states. If each operator in the search space is invertible in this way, the search space is considered undirected.
The domains used were blocks, driverlog, elevators, gripper, logistics, termes and transport.

To compare the compressed and uncompressed PDB heuristics, we ran a search with an A* algorithm without pruning, both with a compressed and an uncompressed PDB heuristic. To create the pattern, we used a greedy pattern generator with a maximum of 1,000,000 states for the PDB.

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

To compare the compressed and uncompressed canonical heuristics, a search with an A* algorithm was performed. Both the compressed and uncompressed heuristics were created multiple times with different numbers of maximum states for individual PDBs and for the collection as a whole. The PDB limits used for the experiment were 2000, 20,000, 200,000 and 2,000,000 states. The limit of states for the entire PDB collection was always chosen to be 10 times higher than the limit for an individual pattern. In order to get the pattern collection, a hill climbing algorithm was used with a minimum improvement ϵ of 1, 1000 random samples used to estimate the improvement and a time limit of 15 minutes (or half the total time limit for the task).

4.2 Results

For the experiments with the PDB heuristics and canonical heuristics we measured the following metrics:

- **search time:** measures how long it takes to search the search space for a solution. This metric does not include the time it takes to construct the PDB. We expect that the compressed and uncompressed pattern databases to be similarly fast.
- **peak memory:** measures the peak memory usage during the PDB construction and search. If the peak memory usage ever exceeds the memory limit for the task, an out-of-memory error will occur.
- **PDB time:** measures how long it takes to construct the PDB or PDB collection.
- **PDB memory:** estimates how much memory is needed to store the PDB or PDB collection. The estimate is dependent on the number of states in the PDB as well as the number of cached values at the end of the search.
- **PDB memory usage:** measures how much of the total memory used during the search is taken up by the PDB or PDB collection.
- **PDB percentage evaluated:** measures the percentage of the PDB or PDB collection that was accessed during the search. This metric was only measured for the compressed PDB implementations.
- **evaluation speed:** measures how many state evaluations can be performed on average per second during the search.
- **coverage:** measures how many tasks can be successfully solved. Specifically, we want to know if the compression has an effect on coverage.
- **search-out-of-memory error (memory error):** tracks when the search fails due to insufficient memory. Since the 1.6-Bit compression frees up memory for the search, we expect to see less out-of-memory errors in the compressed heuristics than in their uncompressed versions.

- **search-out-of-time error (time error):** tracks when the search fails due to running out of time.

For the arithmetic mean values of a metric we only consider all tasks that were solved with all heuristics of the experiment. Tasks that were only solved with some heuristics but not others are ignored.

4.2.1 PDB Heuristic

The heuristic values of the compressed PDB heuristic are the same as those of the uncompressed PDB heuristic and thus the resulting plans will also be identical. Solving a task with the compressed heuristic is on average 7.41% slower (see Table 4.1). Constructing the PDB and retrieving heuristic values are both slower with a compressed pattern database than with an uncompressed one. However, the PDB construction time is always very low, never exceeding one second. Retrieving heuristic values from a compressed PDB is on average 8.54% slower than from an uncompressed PDB, which likely accounts for the main time loss for the compressed PDB heuristic.

	$h^{PDB}_{compressed}$	h^{PDB}
search time (s)	24.41	**22.60**
peak memory (kb)	324,754	**324,259**
PDB time (s)	0.19	**0.00**
PDB memory (kb)	**410**	1,964
evaluation speed (evals / s)	425,347	**465,060**
coverage (# of tasks)	135	135
time error (# of tasks)	0	0
memory (# of tasks)	143	143

Table 4.1: PDB heuristics. All metrics except coverage, time error and memory error are averaged using arithmetic mean.

The compressed PDB usually uses less memory than the uncompressed PDB (see Table 4.2). Because the caching required by the compressed pattern database takes up space dependent on the number of abstract states that were evaluated, the compression rate is not uniform. If enough abstract states are evaluated, the memory requirements for the caching become larger than the memory reduction from the compression, meaning that the compressed PDB heuristic needs more memory than the uncompressed heuristic. In this experiment, this happened 10 times (see Table 4.4).
On average the compressed pattern database takes up 79.1% less memory space than an uncompressed PDB.

The coverage is the same between the compressed and uncompressed PDB heuristics (see Table 4.1). For both heuristics there are no out-of-time errors and the same number of out-of-memory errors. As can be seen in Table 4.3 The memory space taken up by the PDB relative to the total memory used is on average 0.6% for 1.6-Bit compressed PDBs and 7.1% for uncompressed PDBs. The highest measured memory usage was 1.4% for compressed

PDB memory (kb)	$h^{PDB}_{compressed}$	h^{PDB}	**difference (%)**
blocks (21)	**183**	1870	90.2
driverlog (11)	**315**	1547	79.6
elevators-opt08-strips (20)	**219**	1769	87.6
elevators-opt11-strips (16)	**226**	2013	88.8
gripper (8)	**510**	1691	69.8
logistics00 (16)	**723**	2557	71.7
logistics98 (3)	**185**	2953	93.7
termes-opt18-strips (12)	**1223**	2611	53.2
transport-opt08-strips (12)	**153**	1340	88.6
transport-opt11-strips (8)	**411**	1916	78.5
transport-opt14-strips (8)	**362**	1332	72.8
Arithmetic mean (135)	**410**	1964	79.1

Table 4.2: PDB memory. Values are averaged over each domain using arithmetic mean.

PDB memory usage	$h^{PDB}_{compressed}$	h^{PDB}
blocks (21)	**0.006**	0.100
driverlog (11)	**0.005**	0.073
elevators-opt08-strips (20)	**0.005**	0.068
elevators-opt11-strips (16)	**0.005**	0.073
gripper (8)	**0.010**	0.056
logistics00 (16)	**0.004**	0.061
logistics98 (3)	**0.005**	0.106
termes-opt18-strips (12)	**0.003**	0.023
transport-opt08-strips (12)	**0.006**	0.075
transport-opt11-strips (8)	**0.007**	0.080
transport-opt14-strips (8)	**0.006**	0.062
Arithmetic mean (135)	**0.006**	0.071

Table 4.3: PDB memory space in relation to the peak memory. Values are averaged for each domain using arithmetic mean.

and 19.7% for uncompressed PDBs. This means that the memory saved by the compression has only a little effect on total memory usage.
The 1.6-Bit compressed heuristic shows no clear advantage in task coverage and because it performs slightly slower than the uncompressed PDB heuristic, it is generally better to use the uncompressed heuristic.

4.2.2 Canonical Heuristic

The following shorthands will be used to differentiate the searches with or without compression and with the different pattern / collection size limits: a search without compression is h, whereas a search with compression is h^c. The PDB state limit is specified in the subscript. The search with compression and a PDB limit of 20,000 states is therefore h^c_{20k}.

Independent of the pattern size, the uncompressed canonical heuristic search is usually faster than the compressed one (see Table 4.5). The PDB construction time is very similar between the compressed and uncompressed PDBs, which suggests that the time needed for

Task	**PDB memory (b)**		**% evaluated**
	$h^{PDB}_{compressed}$	h^{PDB}	
driverlog - p08	1481868	1119848	0.637
elevators-opt08-strips - p15	262812	262248	0.476
elevators-opt08-strips - p16	465201	334188	0.671
elevators-opt11-strips - p16	262812	262248	0.476
logistics00 - probLOGISTICS-7-1	4131984	4000104	0.491
termes-opt18-strips - p17	3786050	2097256	0.878
transport-opt08 - p14	642684	420008	0.740
transport-opt11 - p18	642684	420008	0.740
transport-opt14 - p08	1571484	937128	0.813
transport-opt14 - p14	157226	131176	0.574

Table 4.4: PDB percentage evaluated for exceptions where the compressed PDB needed more memory space than the uncompressed PDB

the compression step is negligible compared to the time needed to construct the pattern collection and the uncompressed PDBs. The search time for the compressed PDB collection is between 19% (for 2000 PDB states) and 42% (for 2,000,000 PDB states) slower compared to the uncompressed collection. The evaluation speed is between 21.7% and 23.7% slower for the compressed PDB collections. It is unclear how the discrepancy between search time and evaluation speed comes to be.

	h^c_{2k}	h_{2k}	h^c_{20k}	h_{20k}	h^c_{200k}	h_{200k}	h^c_{2000k}	h_{2000k}
search time (s)	23.07	**19.37**	14.07	**10.92**	12.08	**8.80**	9.47	**6.65**
peak memory (Mb)	170	170	139	139	120	**119**	144	**143**
PDB time (s)	62.03	62.07	92.62	92.87	118.19	118.44	118.97	119.08
PDB memory (kb)	**26.7**	27.1	**124**	212	**723**	1696	**4301**	15,030
evaluation speed (1000 evals / s)	310	**396**	290	**372**	249	**326**	236	**308**
coverage	173	173	178	178	182	182	184	184
time error	15	**14**	14	**13**	16	**13**	9	**5**
memory error	**90**	91	**86**	87	**80**	83	**85**	89

Table 4.5: Canonical PDBs heuristics. All metrics except coverage, out-of-time and out-of-memory are averaged using arithmetic mean.

The peak memory usage is roughly the same for the compressed and uncompressed heuristic searches. The memory needed for the PDB collection depends on the size of the collection as well as the number of abstract states evaluated. Like with the PDB heuristics, it is possible that the compressed collection needs more memory than the uncompressed collection for certain tasks (see Table 4.6).
The compressed PDB collections are on average between 1.6% (for 2,000 state PDBs) and 71.4% (for 2,000,000 state PDBs) smaller in terms of memory. The bigger PDB collections have significantly better compression than smaller collections. This suggests that larger patterns or pattern collections are better suited for 1.6-Bit compression.

The reason for this is the caching of all already evaluated abstract states of the PDB collection. As can be seen in Table 4.8, in the larger PDB collections a smaller percentage of the abstract state space is evaluated, which means that less values need to be stored relative to the collection size. In the blocks and gripper domains, the higher PDB / collection size limit does not result in a lower percentage of evaluated abstract states. This is because the pattern collections for the tasks in these domains remain small regardless of the size limit. The gripper domain is especially ill suited for the 1.6-Bit compression (see Table 4.9. In all successful searches in this domain, the entire PDB collection is evaluated, resulting in the worst case for the compression. The PDB collections for the gripper domain on average use up 68.8% more memory than the uncompressed collections.

PDB memory (kb)	h^c_{2k}	h_{2k}	h^c_{20k}	h_{20k}	h^c_{200k}	h_{200k}	h^c_{2000k}	h_{2000k}
blocks (28)	**2.3**	2.8	**4**	9.6	**4**	9.6	**4**	9.6
driverlog (13)	47.7	**42.6**	**201**	323	**770**	2,135	**5,090**	18,870
elevators-opt08-strips (30)	**38.6**	48.9	**184**	453	**928**	3,295	**5,983**	23,226
elevators-opt11-strips (20)	**29.3**	43.9	**129**	369	**682**	2,519	**4,281**	16,919
gripper (8)	2.9	**1.7**	2.9	**1.7**	2.9	**1.7**	2.9	**1.7**
logistics00 (22)	**26**	41.9	**123**	407	**803**	3,058	**7,593**	3,0185
logistics98 (5)	**29**	43	**118**	341	**684**	2,532	**9,039**	35,735
termes-opt18-strips (12)	35	**19**	132	**79**	**1,475**	1,687	**7,859**	16,528
transport-opt08-strips (14)	16	**13**	**63**	66	**221**	535	**1,536**	5,665
transport-opt11-strips (11)	25	**18**	176	**128**	**683**	1,000	**2,232**	8,060
transport-opt14-strips (10)	41	**23**	231	**157**	**1,696**	1,882	**3,691**	10,130
Arithmetic mean (173)	**27**	27	**124**	212	**723**	1,696	**4,301**	15,030

Table 4.6: PDB memory averaged per domain using arithmetic mean.

The compressed and uncompressed heuristics perform the same in terms of coverage (see Table 4.5). In nine cases across all PDB sizes, the compressed heuristic did not run out of memory and instead ran out of time (see Tables 4.10 and 4.11). Despite the reduced memory errors, the 1.6-Bit compression shows no clear improvement to the uncompressed canonical heuristic.

As can be seen in Table 4.7, on average the PDB collections do not make up much of the total memory used. Even with our highest PDB size limit of 2,000,000 states, the collections only take up an average of 4.1% for the compressed and 15.1% for the uncompressed PDB collections. Any memory freed up by the compression is therefore limited.

PDB memory usage	h^c_{2k}	h_{2k}	h^c_{20k}	h_{20k}	h^c_{200k}	h_{200k}	h^c_{2000k}	h_{2000k}
blocks (28)	0.000	0.000	**0.000**	0.001	**0.000**	0.001	**0.000**	0.001
driverlog (13)	0.002	0.002	**0.007**	0.016	**0.024**	0.081	**0.050**	0.182
elevators-opt08-strips (30)	**0.002**	0.003	**0.010**	0.027	**0.024**	0.087	**0.041**	0.162
elevators-opt11-strips (20)	**0.002**	0.004	**0.009**	0.026	**0.020**	0.076	**0.042**	0.167
gripper (8)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
logistics00 (22)	**0.002**	0.003	**0.008**	0.027	**0.032**	0.123	**0.058**	0.230
logistics98 (5)	**0.002**	0.003	**0.007**	0.023	**0.016**	0.060	**0.043**	0.170
termes-opt18-strips (12)	0.001	**0.000**	0.001	0.001	**0.017**	0.038	**0.058**	0.181
transport-opt08-strips (14)	0.001	0.001	**0.002**	0.004	**0.011**	0.033	**0.042**	0.150
transport-opt11-strips (11)	0.001	0.001	**0.003**	0.005	**0.016**	0.043	**0.059**	0.213
transport-opt14-strips (10)	0.001	0.001	**0.002**	0.005	**0.016**	0.044	**0.058**	0.210
Arithmetic mean (173)	**0.001**	0.002	**0.005**	0.012	**0.016**	0.053	**0.041**	0.151

Table 4.7: PDB collection memory space in relation to the peak memory. Values are averaged for each domain using arithmetic mean.

PDB percentage evaluated	h^c_{2k}	h^c_{20k}	h^c_{200k}	h^c_{2000k}
blocks (28)	0.850	0.850	0.850	0.850
driverlog (13)	0.430	0.187	0.114	0.073
elevators-opt08-strips (30)	0.258	0.063	0.013	0.003
elevators-opt11-strips (20)	0.217	0.046	0.009	0.002
gripper (8)	1.000	1.000	1.000	1.000
logistics00 (22)	0.154	0.035	0.025	0.022
logistics98 (5)	0.170	0.034	0.007	0.001
termes-opt18-strips (12)	0.859	0.660	0.363	0.129
transport-opt08-strips (14)	0.502	0.272	0.056	0.024
transport-opt11-strips (11)	0.764	0.416	0.095	0.011
transport-opt14-strips (10)	0.791	0.488	0.171	0.078
Arithmetic mean (173)	0.545	0.368	0.246	0.199

Table 4.8: PDB percentage evaluated

gripper	h^c_{2k}		h^c_{20k}		h^c_{200k}		h^c_{2000k}	
	size	accessed	size	accessed	size	accessed	size	accessed
prob01.pddl	30	1.000	30	1.000	30	1.000	30	1.000
prob02.pddl	36	1.000	36	1.000	36	1.000	36	1.000
prob03.pddl	42	1.000	42	1.000	42	1.000	42	1.000
prob04.pddl	54	1.000	54	1.000	54	1.000	54	1.000
prob05.pddl	54	1.000	54	1.000	54	1.000	54	1.000
prob06.pddl	60	1.000	60	1.000	60	1.000	60	1.000
prob07.pddl	72	1.000	72	1.000	72	1.000	72	1.000
prob08.pddl	72	1.000	72	1.000	72	1.000	72	1.000
prob09.pddl	78	None	78	None	78	None	78	None
prob10.pddl	90	None	90	None	90	None	90	None
prob11.pddl	90	None	90	None	90	None	90	None
prob12.pddl	96	None	96	None	96	None	96	None
prob13.pddl	102	None	102	None	102	None	102	None
prob14.pddl	108	None	108	None	108	None	108	None
prob15.pddl	114	None	114	None	114	None	114	None
prob16.pddl	120	None	120	None	120	None	120	None
prob17.pddl	126	None	126	None	126	None	126	None
prob18.pddl	132	None	132	None	132	None	132	None
prob19.pddl	144	None	144	None	144	None	144	None
prob20.pddl	144	None	144	None	144	None	144	None

Table 4.9: PDB collection size and percentage of the collection that was accessed during the search for the gripper domain. As can be seen, the PDB collections for this domain are very small regardless of the size limit. In all successful searches, the entire PDB collection space was accessed.

memory errors	h^c_{2k}	h_{2k}	h^c_{20k}	h_{20k}	h^c_{200k}	h_{200k}	h^c_{2000k}	h_{2000k}
blocks	7	7	7	7	7	7	7	7
driverlog	4	4	3	3	**1**	2	**4**	5
elevators-opt08-strips	0	0	0	0	0	0	0	0
elevators-opt11-strips	0	0	0	0	0	0	0	0
gripper	12	12	12	12	12	12	12	12
logistics00	0	0	0	0	0	0	**0**	1
logistics98	**24**	25	**26**	27	**26**	28	**26**	28
termes-opt18-strips	8	8	8	8	8	8	7	7
transport-opt08-strips	16	16	15	15	11	11	15	15
transport-opt11-strips	9	9	6	6	6	6	5	5
transport-opt14-strips	10	10	9	9	9	9	9	9
Sum	**90**	91	**86**	87	**80**	83	**85**	89

Table 4.10: out-of-memory errors per domain

time errors	h^c_{2k}	h_{2k}	h^c_{20k}	h_{20k}	h^c_{200k}	h_{200k}	h^c_{2000k}	h_{2000k}
blocks	0	0	0	0	0	0	0	0
driverlog	3	3	4	4	5	**4**	3	**2**
elevators-opt08-strips	0	0	0	0	0	0	0	0
elevators-opt11-strips	0	0	0	0	0	0	0	0
gripper	0	0	0	0	0	0	0	0
logistics00	6	6	6	6	3	3	3	**2**
logistics98	6	**5**	4	**3**	4	**2**	3	**1**
termes-opt18-strips	0	0	0	0	0	0	0	0
transport-opt08-strips	0	0	0	0	4	4	0	0
transport-opt11-strips	0	0	0	0	0	0	0	0
transport-opt14-strips	0	0	0	0	0	0	0	0
Sum	15	**14**	14	**13**	16	**13**	9	**5**

Table 4.11: out-of-time errors per domain

5 Conclusion

In this thesis we introduced 1.6-Bit compression for pattern databases. Our goal was to find out how the memory freed up by the compression affects the search. We looked at two heuristics that use pattern databases: PDB heuristics that use a single PDB and canonical heuristics use a PDB collection of one or more PDBs. For those heuristics we compared the effects on the search of compressed and uncompressed pattern databases.
1.6-Bit compression has multiple limitations. The compression only works on domains with unit cost, as well as an undirected search space. This restriction is not shared by regular pattern databases. Additionally, there is expected to be some time loss due to the compression when creating the PDB and decompression needed when retrieving the heuristic values.

Versions of the heuristics using 1.6-Bit compressed pattern databases were implemented in the Fast Downward planning system [8] and tested on the sciCORE cluster using the Downward Lab package [9].
In the first experiment we compared searches of PDB heuristics using compressed and uncompressed PDBs. We found that the search with a compressed pattern database took on average 8% longer. The time needed to compress the pattern database was negligible, but retrieving a heuristic value was slower for the compressed heuristic, resulting in 8.45% less evaluations per second. On average, the compressed PDBs required only 20.1% of the memory space as the corresponding uncompressed PDBs. Because only the compressed PDB heuristic needs to store values of already evaluated states, the compression rate depends on the number of evaluated abstract states in the PDB. In ten tasks of the experiment the compressed PDBs needed more memory than the uncompressed PDBs, but for most tasks there were significant reductions in memory space needed. However, the memory needed to store the PDB is relatively small, on average 7.1% for uncompressed PDBs. There were no differences in coverage or in the occurence of out-of-memory errors. Based on these results we suggest that PDB heuristics without compression are generally the better choice due to their faster performance and identical coverage.
In the second experiment we compared searches with canonical heuristics using collections of compressed and uncompressed pattern databases. We also ran the searches with different PDB and collection size limits. We found that the compressed PDB collections led to a

between 19% and 42% higher search time. The time needed to compress the PDB collection is negligible, but the slower retrieval of heuristic values seems to have an effect on the search time. The compressed PDB collections were on average smaller than the uncompressed collections, up to 71.4% in the case of the 2,000,000 PDB state limit. There were less out-of-memory errors, with the greatest reduction of 5 (or 4.5%) occuring at the greatest PDB size limit of 2,000,000 states. The coverage remained the same for compressed and uncompressed PDB collections in all PDB size limits. The total memory usage during the search is much higher than the memory needed to store the PDB collections. With the 2,000,000 PDB state limit and without compression, the collections make up only 15.1% of the total memory usage on average. Freeing up memory during the search with 1.6-Bit compression only has only very small advantages and comes at the cost of speed.
A higher PDB and collection size limit tends to increase the relative effect of the compression on the performance and the occurrence of memory errors. The largest size limit led to the greatest relative search time increase, the best compression rate and the highest relative memory error reduction between the compressed and uncompressed heuristics. For the 2,000 PDB state limit, the compression had very little effect. We conclude that larger patterns and pattern collections are more suited for 1.6-Bit compression.

There are multiple improvements that could be made to our implementation of canonical heuristics and PDB heuristics with compressed PDB collections in future works. Compressing the pattern databases immediately after construction and before the next PDB of the collection is constructed could reduce the memory space needed for constructing the PDB collection for a canonical heuristic. This would allow for the creation of larger collections while staying within the same memory space limits, potentially improving the heuristic.
Cooperman and Finkelstein's [3] modulo three breadth-first search in the pattern space would allow the construction of the compressed PDB without first constructing an uncompressed PDB and then compressing it. This method could reduce the memory usage during PDB construction, which would be helpful for constructing larger pattern databases within the same memory limits.
An alternative way of reconstructing the heuristic values of the compressed PDB exists, where the path is stored instead of the heuristic values of all visited states. This method would likely use up less memory but retrieving the heuristic values would be slower. This implementation might be useful for scenarios where memory is much more important than speed.

Bibliography

[1] Teresa Breyer and Richard Korf. 1.6-bit pattern databases. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 39–44, 2010.

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

[3] Gene Cooperman and Larry Finkelstein. New methods for using Cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37-38:95–118, 1992.

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

[5] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 13–24, 2002.

[6] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning*, chapter 1. Morgan Kaufmann, 2004.

[7] Patrik Haslum, Adi Botea, Malte Helmert, 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*, pages 1007–1012, 01 2007.

[8] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

[9] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab, 2017.