

Implementing Symbolic Pattern Databases for Planning

Matthew Fahrni

April 4, 2023

Contents

1	Introduction	2
2	Background	2
2.1	Planning	2
2.2	Heuristic	2
2.3	Pattern Database	3
2.4	Symbolic Data Structures	3
2.4.1	Binary Decision Diagram	3
2.4.2	Algebraic Decision Diagram	3
3	Implementing Symbolic Pattern Databases using Binary Decision Diagrams	5
3.1	Introducing Symbolic Variables of State Space	5
3.2	Building the Transition Relations	6
3.3	Backward search for computation of Pattern Database	6
4	Results	8
4.1	Technical Setup	8
4.2	Comparison Using Greedy Pattern Generator	8
4.3	Comparison Using Hillclimbing Pattern Collection Generator	9
4.4	Comparison Using Systematic Pattern Collection Generator	9
5	Discussion	16
5.1	Improvements and Extensions	16
6	Conclusion	16
7	Acknowledgements	16

1 Introduction

Planning is a field of Artificial Intelligence. Planners are used to find a sequence of actions, to get from the initial state to a goal state. Many planning algorithms use heuristics, which allow the planner to focus on more promising paths. Pattern database heuristics allow us to construct such a heuristic, by solving a simplified version of the problem, and saving the associated costs in a pattern database. These pattern databases can be computed and stored by using symbolic data structures.

In this paper we will look at how pattern databases using symbolic data structures using binary decision diagrams and algebraic decision diagrams can be implemented. We will extend fast downward (Helmert [2006]) with it, and compare the performance of this implementation with the already implemented explicit pattern database.

2 Background

2.1 Planning

A SAS+ planning task is a 4-tuple $\Pi = \langle V, I, G, A \rangle$ where

- $V = \{v_1, \dots, v_n\}$ is a finite set of state variables, where the domain $dom(v_i)$ of every state variable $v_i \in V$ is finite. A fact is an assignment of a state variable ($v_i \mapsto d_i$) with $d_i \in dom(v_i)$. A partial state is a consistent set of facts. A state is a consistent set of facts, in which every state variable $v_i \in V$ is assigned a value of its domain $dom(v_i)$. We denote the set of all possible states S .
- I is the initial state. It is represented as a state.
- G is the goal. It is represented as a partial state. We say that a state s is a goal state, if s is consistent with G .
- A is a finite set of actions, where every $a \in A$ consists of the preconditions, effects and cost of action $a := \langle prec, eff, cost \rangle$.
 - The preconditions of an action $prec(a)$ are represented as a partial state. We say that action a is applicable on a state s if $prec(a) \subseteq s$.
 - The effect of an action $eff(a)$ is a partial state. It defines what the resulting state s' is, if the action is applied on state s .
 - The cost of an action $cost(a) \in \mathbb{R}_0^+$ shows, how expensive it is to apply the action.

We say that s' is successor of s and s is predecessor of s' , if applying the action a on s results in the state s' .

Planning tasks allow us to represent complex problems efficiently. The purpose is to find a plan. A plan is a sequence of actions $\pi = (a_1, \dots, a_n)$ to get from the initial state I to a goal state. A **optimal** plan π is a plan, which finds the cheapest sequence of actions to get to a goal state.

2.2 Heuristic

A heuristic is a function $h(s)$ which takes a state, and estimates the cost to get to a goal state. They allow the search algorithm to focus on more promising paths, thus reducing the search time and the memory usage. The perfect heuristic $h^*(s)$ returns the cost of the cheapest path beginning from that state. We call a heuristic

- **safe** if $h^*(s) = \infty \forall s \in S$ with $h(s) = \infty$
- **goal-aware** if $h(s) = 0 \forall s \in G$
- **admissible** if $h(s) \leq h^*(s) \forall s \in S$
- **consistent** if $h(s) \leq cost(a) + h(s') \forall (s, a, s') \in T$

2.3 Pattern Database

A pattern P is a subset of the state variables of a planning task $P \subseteq V$. Using the pattern, we can define an abstract planning task $\Pi^P = \langle P, I^P, G^P, A^P \rangle$ by only keeping the facts in the original planning task, which contain the variables present in the pattern. As the abstract planning task is smaller than the original, it is possible to fully explore it, and save the perfect heuristic values in a pattern database. The perfect heuristics of one or multiple abstract states can be used as an abstraction heuristic for the original planning task, as it is a **safe**, **goal-aware**, **admissible** and **consistent** heuristic.

2.4 Symbolic Data Structures

2.4.1 Binary Decision Diagram

A binary decision diagram (BDD) is a rooted, directed, acyclic graph (Torralba Arias de Reyna [2015]). Every nonterminal node in a BDD is assigned a variable x_i , as well as two arcs which point to child nodes, which refer to a \top or \perp assignment of said variable. There are two terminal nodes, which represent \top or \perp respectively.

An **Ordered** Binary Decision Diagram (OBDD) is a BDD, in which the Nodes representing variables always appear in a predefined order from root to sink, i.e. if the variable ordering (x_0, \dots, x_n) is given, then on any path from the root to the terminal node, the node x_i appears before node x_j , for all $i < j$.

A **Reduced** Ordered Binary Decision Diagram (ROBDD) is a OBDD, on which the following two reduction rules are applied:

- **isomorphic** Reduction: If the subgraphs rooted at two arbitrary nodes are the same, then all incoming arcs can be redirected to the same node, and the other node can be deleted. An example can be seen by comparing the figures 1a and 1b.
- **Shannon** Reduction: If two arcs beginning from the same parent node point to the same child node, then the incoming arcs of the parent node can be redirected to the child node. The parent node can then be deleted. An example can be seen by comparing the figures 1b and 1c.

From this point onward, whenever we talk about BDD, we are actually referring to ROBDD. ROBDD are more useful than BDD, as they are a canonical, if used with a fixed variable ordering, which allows for faster computation of equality of different ROBDDs.

In a mathematical sense, a BDD with n variables represents a function, for which holds, that $f(x) \in \{0, 1\} \forall x \in \{0, 1\}^n$. As there exists an injective mapping $v_i : \{0, 1\}^{\lceil \log_2(n_i) \rceil} \rightarrow \{0, 1, \dots, n_i\} := X_i \forall n_i \in \mathbb{N}_0$ and therefore an injective mapping $v : (v_1, \dots, v_m) \rightarrow (X_1 \times \dots \times X_m) := \mathbf{X} \forall m \in \mathbb{N}_0$ a single BDD with equal or less than $\lceil \log_2(n_1) \rceil + \dots + \lceil \log_2(n_m) \rceil$ BDD variables can be used to represent subsets. In the following chapters, we will refer to these BDDs using their respective characteristic function:

$$f_X(x) = \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{if } x \notin X \end{cases} \quad \forall x \in \mathbf{X}$$

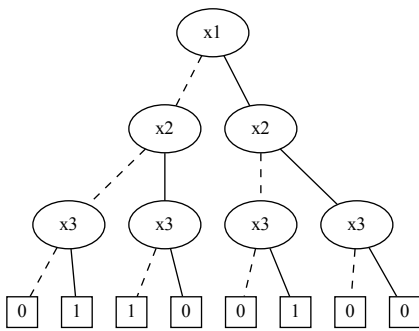
It is also important to note, that the size of a ROBDD is affected by variable ordering given (See Figure 1d and 1c).

2.4.2 Algebraic Decision Diagram

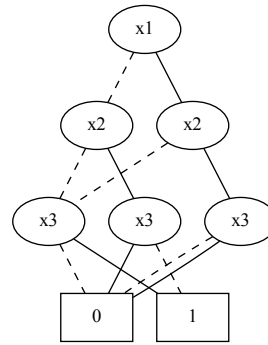
An Algebraic Decision Diagram (ADD) is closely related to a BDD. Similarly to BDDs, every nonterminal Node in a ADD is assigned a variable x_i , as well as two child nodes, which refer to a \top or \perp assignment of said variable. However in ADDs, there can exist an arbitrary finite number of terminal nodes.

ADD can be used to represent the function

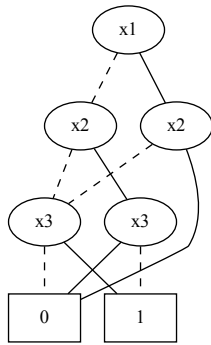
$$f(x) = \begin{cases} c_1 & \text{if } x \in X_1 \\ \dots & \\ c_n & \text{if } x \in X_n \\ 0 & \text{otherwise} \end{cases} \quad \text{with } X_i \cap X_j = \emptyset \forall i, j \in \{1, \dots, n\}, i \neq j, c_k \in \mathbb{R}_0^+$$



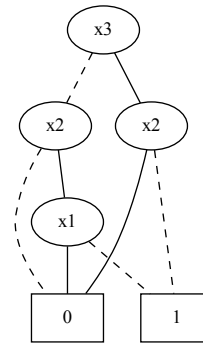
(a) OBDD with ordering (x_1, x_2, x_3)



(b) OBDD with isomorphic reduction



(c) ROBDD



(d) ROBDD with ordering (x_3, x_2, x_1)

Figure 1: BDDs of function $f(x_1, x_2, x_3) = 1$ if $(x_1, x_2, x_3) \in \{(0, 0, 1), (0, 1, 0), (1, 0, 1)\}$ and 0 otherwise

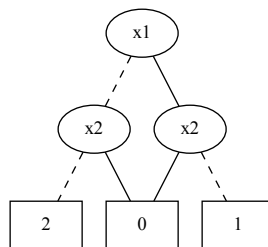


Figure 2: ADD representing function $f(x_1, x_2) = \begin{cases} 2 & \text{if } (x_1, x_2) \in \{(0, 0)\} \\ 1 & \text{if } (x_1, x_2) \in \{(1, 0)\} \\ 0 & \text{otherwise} \end{cases}$

A **Reduced Ordered** Algebraic Decision Diagram, is an ADD, for which the same rules of Ordering and Reduction hold as for the BDD.

3 Implementing Symbolic Pattern Databases using Binary Decision Diagrams

3.1 Introducing Symbolic Variables of State Space

As we have shown in a previous chapter we can represent any subset of a finite set using a BDD using them to represent their characteristic functions. In this chapter we will define the needed BDDs and functionalities of these BDDs such that we can calculate the Pattern Database.

For all following functions and operators we assume that $S' = S$, $s \in S$, $s' \in S'$, $s = \{(v_1 \mapsto d_1), \dots, (v_n \mapsto d_n)\}$, $s_n = \{(v'_1 \mapsto d'_1), \dots, (v'_n \mapsto d'_n)\}$ and $x \in \text{dom}(v_i)$.

Definition 3.1

$$f[v_i \mapsto x](s, s') := \begin{cases} 1 & \text{if } (v_i \mapsto x) \in s \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

This characteristic function is independent of s' and defines a subset $(S[v_i \mapsto x] \times S') \subseteq (S \times S')$. If a given state pair (s, s') is contained in this subset, it implies that $(v_i \mapsto x) \in s$.

Definition 3.2

$$f[v'_i \mapsto x](s, s') := \begin{cases} 1 & \text{if } (v'_i \mapsto x) \in s' \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

This characterises a subset $(S \times S'[v'_i \mapsto x]) \subseteq (S \times S')$. If a given state pair (s, s') is contained in this subset, it implies that $(v'_i \mapsto x) \in s'$.

Definition 3.3

$$f[v_i = v'_i](s, s') := \begin{cases} 1 & \text{if } v_i = v'_i \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

This characteristic function represents a subset $(S \times S')[v_i = v'_i] \subseteq (S \times S')$, which includes all state pairs (s, s') in which the assignment of the corresponding variable is the same.

Definition 3.4

$$f(s, s') +' g(s, s') = \begin{cases} 0 & \text{if } f(s, s') = g(s, s') = 0 \\ 1 & \text{otherwise} \end{cases} \quad (4)$$

If two characteristic functions f, g represent subsets of the same set $X_f, X_g \subseteq (S \times S')$, then $f +' g$ represents a set with $X_{f+'g} = X_f \cup X_g$.

Definition 3.5

$$f(s, s') * g(s, s') = \begin{cases} 1 & \text{if } f(s, s') = g(s, s') = 1 \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

If two c.functions f, g represent subsets of the same set $X_f, X_g \subseteq (S \times S')$, then $f +' g$ represents a set with $X_{f*g} = X_f \cap X_g$.

Definition 3.6

$$F_{\exists s}(f(s, s')) = \begin{cases} 1 & \text{if } \exists x' \in S' \text{ s.t. } f(s, x') = 1 \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

If a characteristic function f represents a subset $X_f \subseteq (S \times S')$ then $F_{\exists v}(f)$ defines a set, which includes $(\{s\} \times S')$ if $(\{s\} \times S') \cap (S \times S')_f \neq \emptyset$. We can understand it, as creating a superset of $(S \times S')_f$, which is independent of s' Torralba Arias de Reyna [2015].

Definition 3.7

$$F_{v \leftrightarrow v'}(f(s, s')) = f(s', s) \quad (7)$$

If $(s, s') \in X_f$ then $(s', s) \in X_{F_{v \leftrightarrow v'}(f)}$.

3.2 Building the Transition Relations

The actions $a \in A$ of the Planning task are implemented by representing them as transition relations $TR_a \subseteq (S \times S')$ and using a BDD to represent their c. function (8).

$$T_a(s, s') = \begin{cases} 1 & \text{if } (s, s') \in TR_a \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

For simplicity reasons, we assume that the we do not use patterns. If a pattern is used, we can simply ignore variables, which are not present in the pattern, thus building our TRs over a subset $(S^P \times S'^P)$, where $S'^P = S^P$ and $S^P \subseteq S$.

$$T_a(s, s') = \left(\prod_i f[v_i \mapsto x_i] \right) * \left(\prod_j f[v'_j \mapsto x'_j] \right) * \left(\prod_k f[v_k = v'_k] \right)$$

$$\text{with } i \in \{i | (v_i \rightarrow x_i) \in prec(a)\},$$

$$j \in \{j | (v'_j \rightarrow x'_j) \in eff(a)\},$$

$$k \in \{k | (v'_k \rightarrow x'_k) \notin eff(a)\}$$

The first product operator calculates a characteristic function of the preconditions of the action $a \in A$. This is done by creating a characteristic function for every assignment in $prec(a)$ (1). This can be understood, as a subset of $(S_{prec(a)} \times S') \subseteq (S \times S')$, which contains every variable pair $(s, s') \in (S_{prec(a)} \times S')$, for which the preconditions are satisfied in v , and all v' are allowed.

The second product operator does the same, but instead over the preconditions of a $eff(a)$ (2). Therefore we get a subset $(S \times S_{eff(a)}) \subseteq (S \times S')$, which contains every state pair $(s, s') \in (S \times S_{eff(a)})$, for which the s' is a possible successor state, and all s are allowed.

The third product operator is needed to ensure, that variables which are not present in $eff(a)$ do not change their value. This is achieved by calculating the characteristic function of the biimplications (3) for every variable, which is not assigned in $eff(a)$. This means that we create a subset $(S_{biimp(a)} \times S'_{biimp(a)}) \subseteq (S \times S')$, which contains all state pairs (s, s') , in which the corresponding assignments are the same $v_i = v'_i$, as long as they are not present in $eff(a)$.

The resulting Transition Relation is a subset $TR_a \subseteq (S \times S')$. If for given a variable assignment (s, s') it holds, that $T_a(v, v') = 1$, then it means that the state s' can be reached applying the corresponding action a on state s . As the function T_a does not include the cost of the action $cost(a)$, it must be kept track of separately.

Since these transition relations are used only to calculate the heuristic value of a state, it is not necessary to distinguish between different operators of the same cost. This allows us to merge multiple T_{a_i} into a singular transition relation T^c .

3.3 Backward search for computation of Pattern Database

Backward search for the computation of a Pattern Database is based on the Dijkstra Algorithm. It is performed by representing the sets of states as a single BDD and operating with it, using the former calculated Transition Relations representing sets of operations.

Dijkstra using symbolic data structures (see Algorithm 1) begins by calculating the corresponding BDD of all goal states. Afterwards, we can apply every 0-cost operation at once, using the corresponding transition relation. We can add this new BDD, which represents all the states with a heuristic

value of 0, to a priorityqueue, which contains BDDs as well as its heuristic value. The priorityqueue is ordered by minimal h-value.

While the priority-queue pq is not empty, we take out the BDD with the smallest h-value. If there are multiple BDDs with the same h-value, we can merge these BDDs, such that we don't have to apply the same operation multiple times (lines 5-8). This newly calculated BDD is added to the pattern database vector $[(f^h, h)]$ with the corresponding h-value. For every Transition Relation with non-zero cost, we apply it to the BDD 2, as well as 0-cost transition relation afterward. It is important to apply the 0-cost transition relations repeatedly, until no more states are expanded, as it is possible, that the amount of represented states can increase using multiple 0-cost operations in succession. If the newly calculated BDD represents more sets of states as the former BDD, we add it to the priority-queue pq with their corresponding h-value.

Algorithm 1 Backward search for computation of Pattern Database

Input: goal states as BDD: f_{goal}
Transition Relations with cost vector: $((T_1, c_1), \dots, (T_n, c_n))$
optional zero-cost Transition Relation: T^0
Output: pattern database vector $\langle \text{BDD, heuristic-value} \rangle$: $((f_1, h_1), \dots, (f_n, h_n))$

- 1: $reached \leftarrow preimage(f_{goal}, T^0)$
- 2: $pq.push(reached, 0)$
- 3: **while** pq is not empty **do**
- 4: $(reached, h_i) \leftarrow pq.pop$
- 5: **while** $pq.top.h = h_i$ **do**
- 6: $reached \leftarrow reached +' pq.top.f$
- 7: $pq.pop$
- 8: **end while**
- 9: $reached \leftarrow preimage(T^0, reached)$
- 10: $pdb.add(reached, h_i)$
- 11: **for all** $(T_i, c_i) \in ((T_1, c_1), \dots, (T_n, c_n))$ **do**
- 12: $new_states = preimage(reached, T^c)$
- 13: **if** $reached \neq new_states$ **then**
- 14: $pq.push(new_states, c + h_i)$
- 15: **end if**
- 16: **end for**
- 17: **end while**

The preimage algorithm 2 is used to calculate the predecessors of a given set of states. We begin by creating a copy of the BDD, which encodes all the currently reached states. We then swap the primed and unprimed variables, leading to a BDD, which is independent of the unprimed variables. By taking the product with the Transition Relation of the action, we get a BDD, which includes all the possible state pairs (s, s') , where s is a possible predecessor state, when action a is applied. As we only want to keep track of the predecessor states, and do not care, what the successor is, we apply a Existential Quantification over all variables s . We then take the sum of the resulting BDD and the input BDD.

Algorithm 2 preimage computation

Input:
BDD of successor states: f_{succ}
BDD of action: T_a
Output:
BDD of predecessor states BDD_{pred}
h-value of predecessor states $cost(pred)$

- 1: $temp \leftarrow f_{succ}$
- 2: $temp \leftarrow F_{v \leftrightarrow v'}(temp)$
- 3: $temp \leftarrow temp * T_a$
- 4: $temp \leftarrow F_{\exists v}(temp)$
- 5: **return** $(temp +' f_{succ})$

Merging of multiple BDDs into a single ADD is used, such that a single ADD can be used to represent all states with their corresponding heuristic value. This is useful, as it alleviates the need to evaluate a state in multiple BDDs to get its respective heuristic value.

Before we look at how we can achieve this, we must first define, how we convert a pair (f^h, h) to an ADD. We convert the BDD, by replacing the terminal node \top with h , and the terminal node \perp with ∞ . This conversion is denoted by $convert(f^h, h)$ in the merging algorithm 3. This converted ADD represents the function:

$$f^{ADD}(s, s') = \begin{cases} cost & \text{if } f^h(s, s') = 1 \\ \infty & \text{otherwise} \end{cases} \quad \forall x$$

For the actual merging algorithm (see Algorithm 3), we initialize an ADD with constant value ∞ , i.e. an ADD which represents the function $f(x) = \infty \forall x$. Then, for every pair of (f_i, h_i) inside of our vector, we convert it to an ADD, as represented by the aforementioned function, and take the minimal terminal value of the former ADD. We repeat this process for every $(f^h, h) \in [(f^h, h)]$, thus getting an ADD which can be used, to access all the heuristic values.

Algorithm 3 merging

Input:

pattern database as list: $((f_1, h_1) \dots (f_n, h_n))$

Output:

pattern database as ADD: f^{ADD}

- 1: $f^{ADD} \leftarrow ADD(\infty)$
 - 2: **for all** $(f_i, h_i) \in \{(f_1, h_1) \dots (f_n, h_n)\}$ **do**
 - 3: $temp \leftarrow convert(f_i, h_i)$
 - 4: $f^{ADD} \leftarrow min(temp, f^{ADD})$
 - 5: **end for**
-

4 Results

4.1 Technical Setup

For the experiments, the symbolic pattern database was implemented into Fast Downward, which is a domain-independent classical planning system (Helmert [2006]) with the integration of PDBs (Sievers et al. [2021]). The symbolic data structures were implemented using the **Colorado University Decision Diagram Package**. The used search algorithm is A*. The patterns were generated using different pattern generators, namely the greedy pattern generator, the hillclimbing pattern collection generator, and the systematic pattern collection generator. The experiments were conducted on the sciCORE scientific computing center at University of Basel and evaluated using LAB (Seipp et al. [2017]).

The following three sub chapters compare different metrics when using the implemented symbolic datastructure to compute and represent the PDB, as opposed to the explicit representation. All runs have a overall time limit of 30 minutes.

4.2 Comparison Using Greedy Pattern Generator

In this chapter we will compare the performance when the patterns are created using the greedy pattern generator. The amount of represented abstract states is limited to 1000000.

In the summary (see Table 1) we can see that the coverage is the same, which means that the symbolic representation cannot solve more tasks.

The computation time of the PDB is generally better when using the symbolic approach.

The PDB size compares the number of abstract states represented in explicit algorithm compared to the amount of nodes in the ADD, which represents the same PDB. It must be noted, that a direct comparison is not as useful, as a node needs additional memory. However the needed space is

	symbolic	explicit
coverage	791	791
PDB computation time - Geometric mean	0.10	0.31
PDB size - Geometric mean	246.59	275418.36
search time - Geometric mean	0.57	0.48

Table 1: Summary of results when using the greedy pattern generator

	symbolic	explicit
coverage	873	915
sum CPDB size - Geometric mean	677.77	4182.60
average PDB size - Geometric mean	75.17	294.20
total CPDB computation time - Geometric mean	20.94	1.59
number of patterns - Geometric mean	8.84	8.84
search time - Geometric mean	0.39	0.19

Table 2: Summary of results when using the hillclimbing pattern collection generator

proportional to the amount of nodes, and we can see in the following graphs, that the needed nodes is not proportional to the amount of represented abstract states.

The time for the actual search increases, when the symbolic PDB is used. This is most likely because to get the heuristic value of a state in the ADD, the state must first be converted, to get the corresponding heuristic value. In the scatterplot comparing the number of nodes to the number of abstract states (see Figure 3) it can be seen, that the amount of needed nodes is generally not proportional to the amount of abstract states, but instead increases slower. There are some outliers where more than 10000 abstract states can be represented with one node. While this may seem really efficient, it must be noted that this also implies that the heuristic value of all states is the same, which means that the generated pattern is bad.

In the scatterplot (see Figure 4), we can see that the computation time of the PDB is generally better. However we can also see that there is a lot of variance between the two approaches. Especially for domains, where the actions have different costs, the computation time can increase vastly, because we cannot merge all actions into one transition relation and thus have to compute the preimage multiple times.

4.3 Comparison Using Hillclimbing Pattern Collection Generator

In this chapter we compare the symbolic PDBs with the explicit PDBs using the hillclimbing pattern collection generator. To ensure that the same patterns for both types of PDBs are generated, the amount of generated patterns has been limited to 1000. In the summary (see Table 2) we can see that the coverage is worse when using symbolic PDBs. While the amount of nodes needed to represent the same amount of abstract states is lower, when using symbolic PDBs, it is not as extensive as with the pattern generated with the greedy algorithm. The search time is also worse with the symbolic PDBs, because of the aforementioned reason.

In the scatterplot (see Figure 5) the same trend can be seen as before. The number of nodes does not increase proportionally with the amount of represented abstract states, but less. In the scatterplot (see Figure 6) it can be seen, that computing time of symbolic PDBs is generally worse. However in in unit-cost domains, this difference is not as extensive. This is because the actions must first be transformed into their respective transition relations.

4.4 Comparison Using Systematic Pattern Collection Generator

For the last experiment, the systematic pattern collection generator is used, with the pattern size being limited to 2. In the summary table (see Table 3) we can see, that the symbolic PDBs are outperformed by the explicit PDBs, as the coverage is higher, the search and computation time is higher, and the PDB size is not low enough. In the scatterplot (see Figure 7) we can see, that the amount of nodes needed to represent the PDB is about proportional to the amount of abstract states it represents.

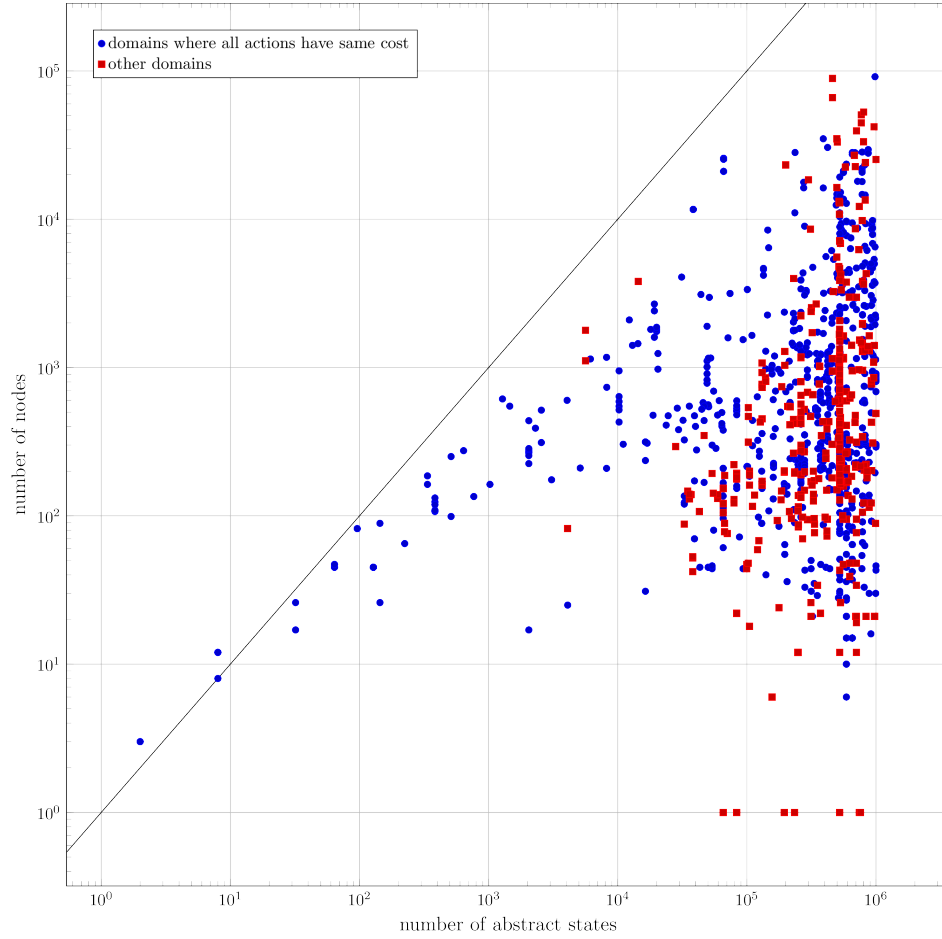


Figure 3: Scatterplot comparing the number of nodes of the ADD with the number of states it represents with pattern generated using greedy pattern generator

	symbolic	explicit
coverage	754	787
average PDB size - Geometric mean	9.68	19.53
sum CPDB size - Geometric mean	460.47	901.25
total CPDB computation time - Geometric mean	0.40	0.12
number of patterns - Geometric mean	44.89	44.89
search time - Geometric mean	1.64	0.49

Table 3: Summary of results when using the systematic pattern collection generator

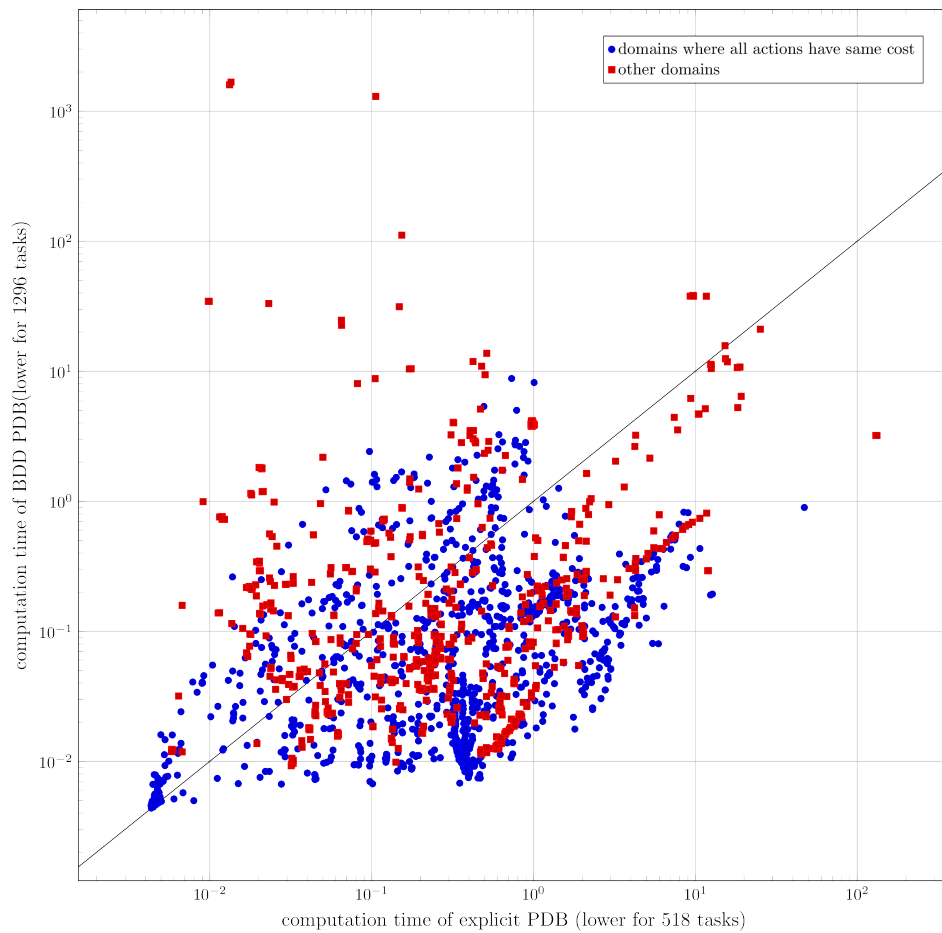


Figure 4: Scatterplot comparing the computation time of explicit PDB with symbolic PDB with pattern generated using greedy pattern generator

This is because we have limited the pattern size to 2, and the symbolic PDBs are better, when bigger patterns are used.

In the scatterplot comparing the computation time (see Figure 8) we can see, that the computation time of the PDBs is worse when using symbolic PDBs.

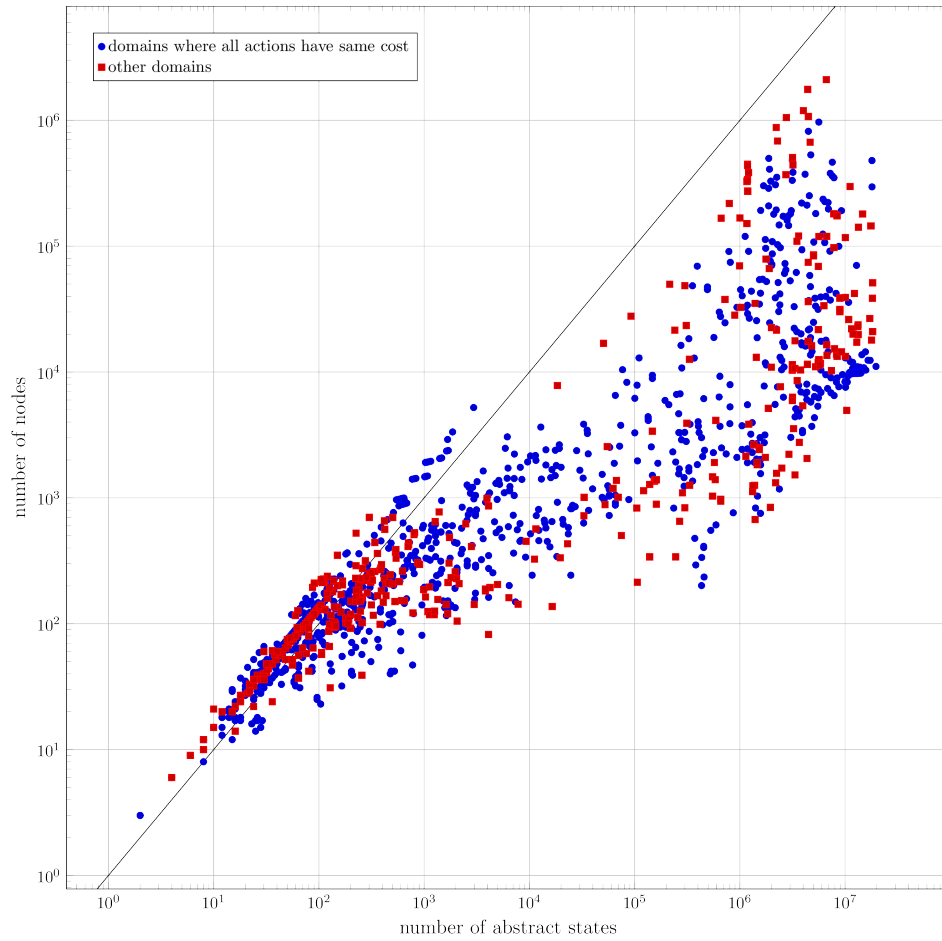


Figure 5: Scatterplot comparing the number of nodes of the ADD with the number of states it represents with patterns generated using hillclimbing pattern collection generator

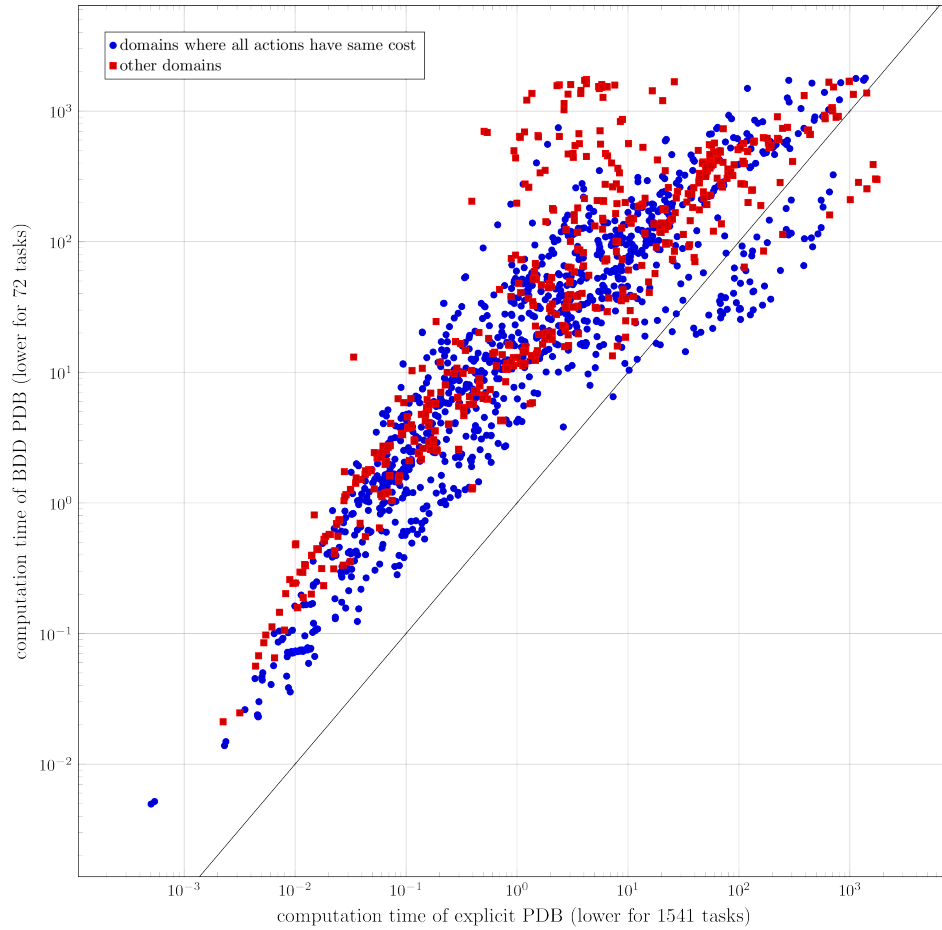


Figure 6: Scatterplot comparing the computation time of explicit PDB with symbolic PDB with patterns generated using hillclimbing pattern collection generator

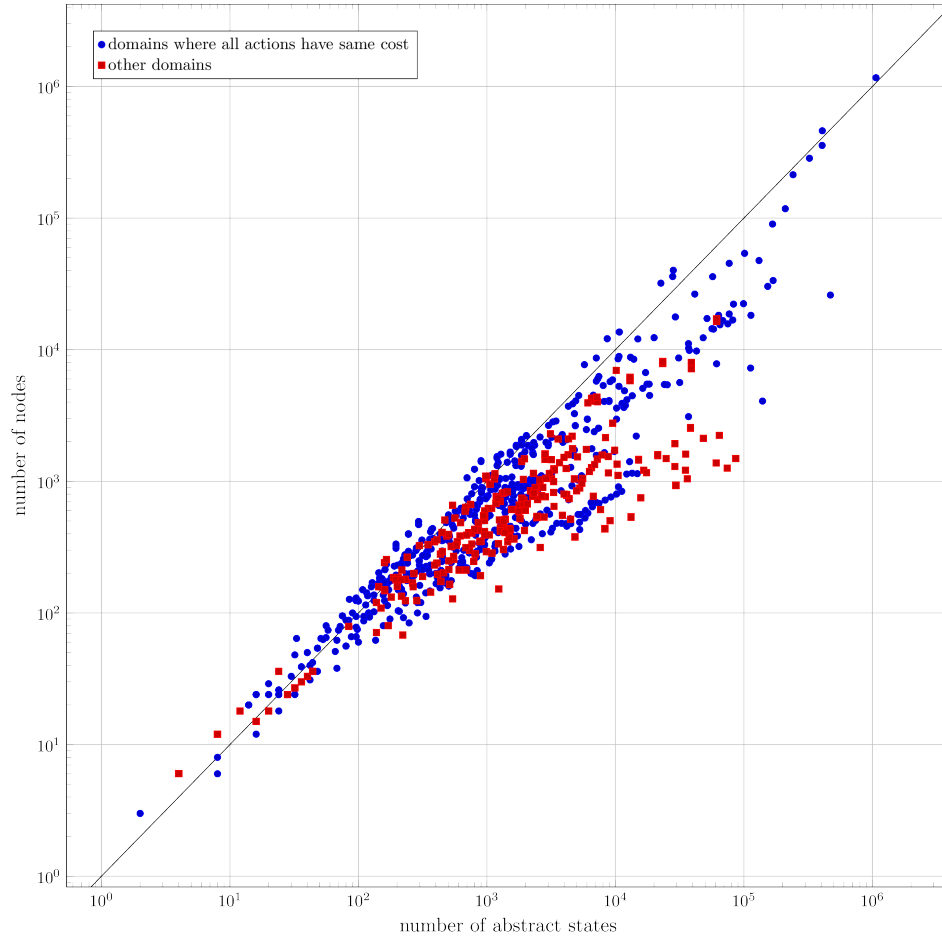


Figure 7: Scatterplot comparing the number of nodes of the ADD with the number of states it represents with pattern generated using systematic pattern collection generator

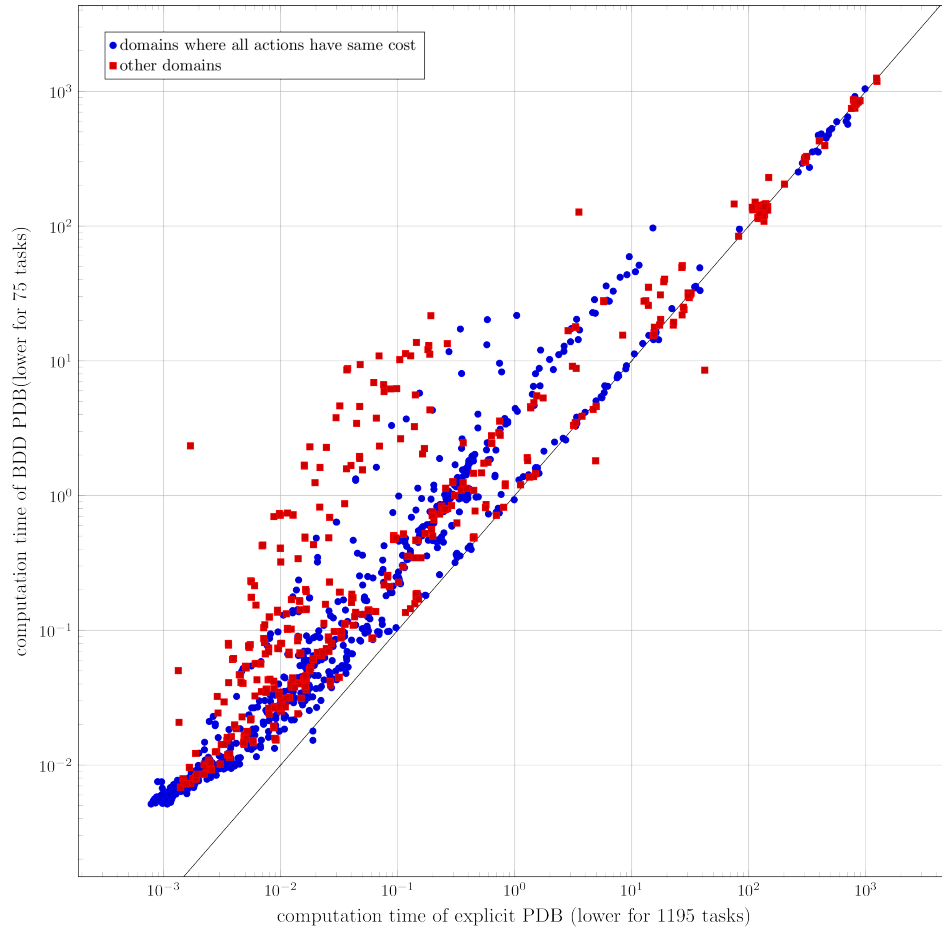


Figure 8: Scatterplot comparing the computation time of explicit PDB with symbolic PDB with patterns generated using systematic pattern collection generator

5 Discussion

5.1 Improvements and Extensions

While the symbolic pattern database is functional, there are lots of improvements and extensions, which can be added. Most importantly the current implementation using BDDs and ADDs is poorly optimized. For example there are functionalities which are implemented subsequently instead of "interweaved", such as the conversion from BDD to ADD, which can be made in the same loop instead of subsequently. Also the variables in the respective BDDs and ADDs are not reordered as mentioned in 2.4.1.

For possible extensions the most obvious one is implementing symbolic search. This would also allow for some data structures to be shared between the symbolic PDB and the search algorithm, which would improve runtime and memory usage. It would also be useful to implement other symbolic data structures, such as zero-suppressed decision diagrams, or edge-valued multi-valued decision diagrams. However, to implement such data structures it might be required to use other planning tasks to utilize their full potential.

6 Conclusion

The implementation of symbolic pattern databases using BDDs and ADDs do not outperform the currently implemented explicit pattern databases, but it is better for in some instances, especially for big patterns and domains with unit-cost actions. During the implementation of symbolic pattern databases I realized, that the current implementation of PDBs is already extremely well optimized. To get a better representation of the usability of symbolic pattern databases, a lot of optimization needs to be done.

While the my implementation of symbolic PDBs is not very well optimized, working on this I have learned a lot about planning in general, particularly about pattern databases, as well symbolic data structures and their usability.

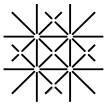
7 Acknowledgements

I would like to thank my supervisor Silvan Sievers, as he has helped me in many aspects of this work, and giving me the guidance needed to tackle such a project.

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

References

- Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26, 2006. doi: 10.1613/jair.1705.
- Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. <https://doi.org/10.5281/zenodo.790461>, 2017.
- Silvan Sievers, Manuela Ortlieb, and Malte Helmert. Efficient implementation of pattern database heuristics for classical planning. *Proceedings of the International Symposium on Combinatorial Search*, 3(1):49–56, 2021. doi: 10.1609/socs.v3i1.18237.
- David Speck, Florian Geißer, and Robert Mattmüller. Symbolic planning with edge-valued multi-valued decision diagrams. 2018.
- Álvaro Torralba Arias de Reyna. Symbolic search and abstraction heuristics for cost-optimal planning. 2015.



Erklärung zur wissenschaftlichen Redlichkeit und Veröffentlichung der Arbeit (beinhaltet Erklärung zu Plagiat und Betrug)


Titel der Arbeit:

Name Beurteiler*in: Dr. Gabriele Röger

Name Student*in: Matthew Fahrni

Matrikelnummer: 17-056-672

Mit meiner Unterschrift 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.

Ort, Datum: Liesberg, 05.04.23 Student*in: 

Wird diese Arbeit veröffentlicht?

Nein

Ja. Mit meiner Unterschrift bestätige ich, dass ich mit einer Veröffentlichung der Arbeit (print/digital) in der Bibliothek, auf der Forschungsdatenbank der Universität Basel und/oder auf dem Dokumentenserver des Departements / des Fachbereichs einverstanden bin. Ebenso bin ich mit dem bibliographischen Nachweis im Katalog SLSP (Swiss Library Service Platform) einverstanden. (nicht Zutreffendes streichen)

Veröffentlichung ab: _____

Ort, Datum: _____ Student*in: _____

Ort, Datum: _____ Beurteiler*in: _____

Diese Erklärung ist in die Bachelor-, resp. Masterarbeit einzufügen.