Karlsruhe Institute of Technology

Calculating sets of plans with Decision Diagrams

Masterthesis of

Jean-Pierre von der Heydt

At the Department of Informatics
Institute of Theoretical Informatics, Algorithmics II

Reviewer:	Prof. Dr. Peter Sanders
Advisors:	Tomáš Balyo
	Dominik Schreiber

1 July 2022 – 1 February 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read and observed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT).*

Karlsruhe, 1.2.2023

. . . .

(Jean-Pierre von der Heydt)

Abstract

The field of automated planning is concerned with the automatic generation of plans for general problem formulations. Given an initial state and a goal, the objective is to find a plan – a sequence of operators – which translates the initial state into the goal. Often, not only a single plan is desired but a collection of multiple plans, for example when some constraints are difficult to formulate mathematically or when looking for particularly robust plans. Since the solution set can grow exponentially, it is infeasible to construct every plan explicitly.

Our aim is to represent the set of solutions as Binary Decision Diagrams (BDDs) or Sentential Decision Diagrams (SDDs). Both Decision Diagrams (DDs) have the advantage that they can represent formulas of propositional logic in a space efficient manner while still supporting many efficient queries. We propose different methods to encode the planning problem as a logic formula and explore which methods are best suited to build a DD from it. Our approach is able to answer queries on solution spaces that are too big for other currently existing planners; this includes counting the number of plans with certain properties and sampling uniform distributed plans. Our approach dominates for particularly large solutions sets of easier problems, while existing planners perform better on small to moderate ones.

Zusammenfassung

Das Forschungsfeld Automated Planning befasst sich mit dem automatischen Generieren von Plänen zur Lösung allgemeiner Probleme. Gegeben einen Startzustand und ein Ziel, wird nach einem Plan – einer Abfolge von Operatoren – gesucht, der den Startzustand in das Ziel übersetzt. Häufig ist es hilfreich, nicht nur einen einzigen Plan, sondern eine Sammlung von mehreren Plänen zu finden. Dies ist zum Beispiel der Fall, wenn einige Randbedingungen mathematisch schwer zu formulieren sind oder wenn nach besonders robusten Plänen gesucht wird. Da die Lösungsmenge exponentiell anwachsen kann, ist es nicht immer möglich, jeden Plan explizit zu konstruieren.

Unser Ziel ist es daher, die Lösungsmenge als Binary Decision Diagrams (BDDs) oder Sentential Decision Diagrams (SDDs) darzustellen. Beide Entscheidungsdiagramme (DD) haben den Vorteil, dass sie aussagenlogische Formeln platzsparend darstellen können und gleichzeitig viele effiziente Anfragen unterstützen. Wir stellen verschiedene Methoden zur Kodierung von Planungsproblemen durch eine logische Formel vor und untersuchen, welche dieser Methoden zur Konstruktion eines DD am besten geeignet sind. Unser Ansatz ist in der Lage, Anfragen auf Lösungsräumen zu beantworten, die kein anderer der derzeit existierende Planer beantworten kann; dazu gehören das Zählen von Plänen mit bestimmten Eigenschaften oder das Ziehen von Plänen mit gleichverteilter Wahrscheinlichkeit. Unser Ansatz dominiert auf besonders großen Lösungsräumen von einfacheren Problemen, während bestehende Planer auf kleinen und mittleren Lösungsmengen effizienter sind.

Contents

1 Introduction

This chapter introduces the key concept of using Decision Diagrams (DDs) to represent sets of plans. In Section 1.1 we motivate our approach and show how other research areas can benefit from it. In Section 1.2 our contributions are summarized. Section 1.3 gives an overview of the structure of the thesis.

1.1 Motivation

Planning problems are solved by a plan – a sequence of operators – that translate an initial state into a desired goal. In most cases, multiple different plans exist. To find these plans, we use Binary Decision Diagrams (BDDs) or Sentential Decision Diagrams (SDDs). Suffice it to say that DDs are data structures that represent formulas of propositional logic and all their solutions. A more extensive explanation of DDs is given in Section 2.2 and Section 2.3. By encoding a planning problem as a logic formula and using a DD to represent it, all solutions to the planning problem can be retrieved.

Finding a complete representation of the solution space is often better than obtaining a single plan. While some problems only require a single plan for a valid solution, there are many cases where multiple plans are desired. One major benefit of having access to multiple solutions is the possibility to choose between alternative plans. Some constraints are hard to formulate in the language of computers. For example, it is easy to formulate the placement of power poles as a planning problem, where the total distance of cables has to be minimized and some connectivity property has to be achieved. But it is hard to calculate the populations approval of this placement. In order to circumvent this problem, multiple optimal power pole placements can be generated and the residents are able to choose the most desireable placement. This popular application is mentioned by multiple researchers regarding the k-shortest path problem, including [Epp98].

Another advantage of this approach is that it provides a powerful tool for analyzing planning domains. Having a DD that represents all plans, makes it possible to identify operators and conditions that are always necessary to reach the goal of a planning problem. This can help identify structures that make a planning problem hard. It is also possible to analyze how sensitive a problem is to variations. By adding or lifting different restrictions on planning problems, DDs can be used to compare the change in the solution spaces and determine which parts of plans are robust to variations. In biological sequence alignment such solutions are more desired [Epp98].

While having a better understanding of planning domains is interesting in its own way, it can also help to construct heuristics. Planning pattern databases [Ede14] construct heuristics for planning problems by first abstracting to a simpler problem with less states and solving it completely. Knowledge on optimal solutions in the simplified problem is used to guide the search solving the original problem. In a similar way a DD can be constructed for a simpler planning problem and queries to the DD can be used as a basis for a heuristic to the original problem.

DDs offer the possibility to choose uniformly distributed solutions without the need to explicitly generate them all. This is especially useful if the solution set is too large to store explicitly. An interesting application of this is to randomly generate instances for puzzles, like Sudoku, Nurikabe or Sokoban. By formulating a planning problem, such that valid plans correspond to solvable puzzle instances, a DD can be used to generate random puzzles. Selecting random solutions sets are also interesting for machine learning approaches. These random set can be used as training sets for machine learning approaches to planning [TTTX18].

Multiple research areas, like Top-*k* planning [RSU14], Top-*q* planning [KSU20] and diverse planning [KS20] aim to find sets of plans and exploit their more complete description of the solution space.

1.2 Contributions

This thesis analyzes the potential and boundaries of DDs to represent all solutions of planning problems. To the best of our knowledge, there are no prior attempts to use DDs in this way. Although other works have explored BDDs to represent sets of planning states, their approach does not offer the same possibilities of queries.

We discuss different ways to encode the planning problem as a logic formula and show how these encodings affect the performance of DDs. A particular focus is placed on how an efficient construction of DDs is achieved with knowledge of the underlying planning problem. We identified the most appropriate encodings and orders for the construction of planning DDs and show that they are a significant improvement over general DD compilers.

We implemented all the algorithms in a planner called PLANDD which is also able to solve the Top-*k* or Top-*q* planning problem. Our planner is able to successfully generate a DD for up to 176 optimal unit-cost planning problems from the benchmarks of the International Planning Competition, making it possible to represent solution spaces with billions of plans and more with a small amount of memory needed. No other planner is able to find solutions of this size. Even state of the art Top-*k* planners have difficulties to compute solution spaces containing more than a million plans. Especially in the area of Top-*q* planning, we can represent several domains particularly well compared to existing planners. Nevertheless, it is difficult for our approach to scale to harder problems. Our approach is unable to create DDs for most harder planning problems and the existing Top-*K* planners outperform our approach on smaller solution spaces.

To complement the construction of DDs, we have implemented several proofs of concept that show how information about planning problems can be queried from a DD, once it has been created. This includes counting the number of plans, choosing uniformly distributed plans and finding the most common operators. These queries take almost no time, once the actual DD is constructed.

1.3 Structure of the Thesis

This thesis is structured as follows. In Chapter 2 the basics of Binary Decision Diagrams, Sentential Decision Diagrams and automated planning are explained. Chapter 3 presents two areas of automated planning that are most closely related to the approach of using DDs to represent sets of plans. We describe the algorithms and heuristic we developed in Chapter 4. This consists of encoding the planning problem in Section 4.1, ordering the encoding in

Section 4.2 and actually building the DD in Section 4.3. The performance of these algorithms is evaluated in Chapter 5. At last, we conclude the results of the thesis in Chapter 6 and point out possibilities for future work.

2 Preliminaries

In this chapter, we introduce some important concepts that we use throughout the work. We start with a brief overview of propositional logic. This is necessary to define BDDs and SDDs. We explain the main differences between BDDs and SDDs and point out their important algorithmic properties. At the end of this chapter we give a short introduction to automated planning.

2.1 Propositional Logic

A formula of propositional logic consist of variables $x_1, x_2 \dots$, to which the values True or False can be assigned. Variables can be combined by the *conjunction* $\land$, *disjunction* $\lor$, *implication* $\Rightarrow$, *equivalence* $\Leftrightarrow$ or *negation* $\neg$ operators, which are interpreted in their usual sense. We write $f_{x=v}, v \in \{\text{True}, \text{False}\}$ for the formula that is obtained from a formula f by assigning x the value v and partially evaluating f. The value of a formula can be evaluated by assigning True or False to every variable. Variables in a formula f can be existentially quantified by the following equation $\exists x.f = f_{x=\text{True}} \lor f_{x=\text{False}}$. A formula is satisfiable if an assignment of truth values exists, such that the formula evaluates to True.

The problem of determining whether a given formula of propositional logic is satisfiable is known as the SAT problem. Formulas of propositional logic are often presented in conjunctive normal form (CNF). CNFs consist of a set of clauses that are conjoined with each other. Each clause consists of variables and negated variables that build a disjunction. An example for a CNF would be $(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_3)$. Every logic formula can be converted into a CNF (with possibly more variables) that is satisfiable iff the original formula is satisfiable. Of course SAT is one of the well known NP-complete problems [Coo71].

2.2 Binary Decision Diagrams

First described in [Bry86], Binary Decision diagrams (BDDs) are directed acyclic graphs (DAGs) that can be used to represent formulas of propositional logic. The graph structure allows to reuse parts the formula in an efficient manner, making BDDs a compact description of logic formulas in some cases.

Definition 2.1 (Binary Decision Diagram):
A BDD is a rooted directed acyclic graph with the following properties.

- The graph contains exactly two nodes with no outgoing edge, labeled with 1 and 0. These nodes are called *terminal nodes* and represent True and False.
- Every other node has an assigned variable and exactly two outgoing edges. These edges are labeled with 1 and 0, which means assigning the value True or False to the variable. Nodes with outgoing edges are called *internal nodes*.
- On every path from the root to a terminal node, a variable may appear at most once.

An example of such a graph is given in Figure 2.1. Given an assignment of variables, a formula that is represented by the graph can be evaluated by the following method: A path is followed in the BDD starting from the root node and ending at a terminal node. At an internal node for variable v the edge with the value of v in the assignment is followed. If the terminal node 1 or 0 is reached, the assignment evaluates to True or False respectively.

A formula F can be converted into a BDD denoted by $G(F)$ with a recursive procedure. A random variable x from F is picked and the two formulas $F_{x=\text{True}}$ and $F_{x=\text{False}}$ are calculated. The BDD is then build from a new internal node, labeled with x, where the 1 edge points to $G(F_{x=\text{True}})$ and the 0 edge points to $G(F_{x=\text{False}})$. If F is constant True or False, $G(F)$ becomes the respective terminal node.

Ordered BDDs Definition 2.1 does not guarantee that a BDD is unique. It is possible for two different BDDs to represent the same propositional formula. This is mainly due to the arbitrary order in which the variables can occur in the DAG. The following definition ensures this uniqueness property.

Definition 2.2 (Orderd BDD and Reduced Ordered BDD):
A binary decision diagram is called *ordered* (OBDD) if a total order of the variables exists that the DAG respects. This means if x is ordered before y, then all nodes labeled with x occur before nodes labeled with y in paths from the root to a terminal node. An OBDD is called *reduced* (ROBDD) if

- for every pair of internal nodes, the subgraphs rooted at these nodes are not isomorphic.
- for every internal node, the outgoing edges do not point to the same node.

If two ROBDDs represent the same logic formula and use the same ordering of variables, then their Graphs will be isomorphic [Bry86].

An OBDD can be converted to a ROBDD by identifying pairs of internal nodes u, v with $u \neq v$, where both 1 edges point to the same node and both 0 egdes point to the same node or both edges of u point to v. If these nodes are merged in the OBDD until no other pairs with this property can be found, the resulting OBDD satisfies Definition 2.2 for an ROBDD. In most cases however, ROBDDs are constructed directly, without creating OBDDs as an intermediate product. Because this thesis only deals with ROBDD we will just call them BDDs from now on.

Definition 2.1 requires a BDD to have a single root node but BDDs with multiple root nodes are also possible. In this case each root node represents a logic formula. The different BDDs have to share the same variable order and by doing so they are also able to share their internal nodes.

Variable Ordering Definition 2.2 introduces an ordering of variables for BDDs. This ordering is not only important for the uniqueness property, but also for the size of the BDD. Figure 2.1 gives an example of a good and a bad variable ordering for a BDD. There exists families of BDDs over n variables with $\mathcal{O}(n)$ nodes for some ordering and $\mathcal{O}(2^n)$ nodes for another ordering [Bry86]. Finding a good ordering is not easy and in some cases not achievable [Bry86]. It is a known NP-hard problem to determine a variable order that results in a BDD with minimal size [BW96] and for certain formulas there exist no ordering where the BDD has polynomial size [Bry86].

In practice, BDDs are very sensitive to variable orders. There exist many algorithms that try to approximate good variable orderings. We propose some own ideas on how to formulate variable orders for BDDs in Section 4.2.

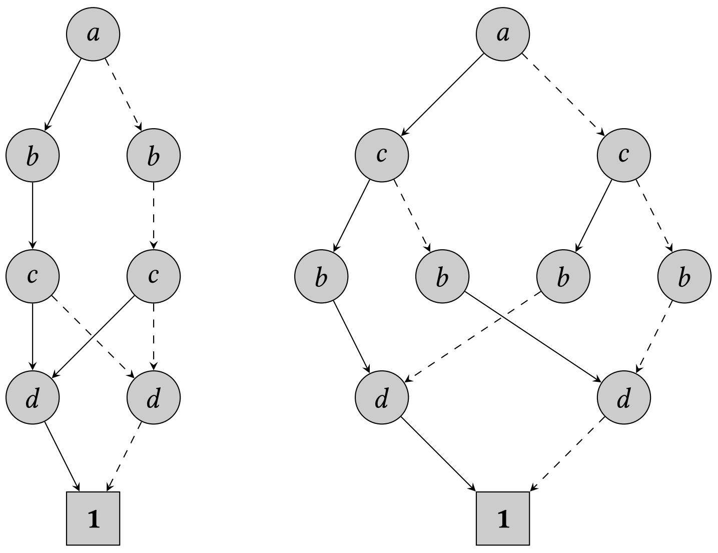

Figure 2.1: Example of a BDD with a good (left) and a bad (right) variable ordering for the formula $(a \Leftrightarrow b) \wedge (c \Leftrightarrow d)$. Left variable order is $[a, b, c, d]$, right variable order is $[a, c, b, d]$. Edges with 0 label use a continuous line and edges with a 1 label use a dashed line. Edges to the 0 terminal node are omitted for clarity.

Algorithms BDDs are built bottom up, using procedures that combine two BDDs under a given logic operator. Such procedures exist for every possible logic operator $(\wedge, \vee, \neg, \ldots)$ and their runtime is bounded by the product of the size of the two participating BDDs. For example, the runtime to conjoin a BDD with n_1 and a BDD with n_2 nodes takes $\mathcal{O}(n_1 \cdot n_2)$ time. Although all combine procedures have polynomial runtime, the construction time can still blow up exponentially if multiple procedures are executed consecutively.

A BDD allows many queries on the underlying logic formula to be answered in polynomial time. This includes determining if the formula is satisfiable in constant time. Finding a satisfying assignment is possible in linear time in the number of variables. Other possible queries include counting the number of satisfying assignments or picking at random one satisfying assignment in time linear in the number of BDD nodes. This does not contradict the NP-completeness of SAT, since the construction of a BDD from a given logic formula may take exponential time. We discuss these queries and their applications to automated planning in Section 4.4.

2.3 Sentential Decision Diagrams

Sentential decision diagrams (SDDs) [Dar11] are a recent generalization of BDDs. They retain most properties of BDDs but can have exponentially less nodes in some cases [Bov16].

They key difference between BDDs and SDDs is, that SDDs can branch other multiple logic cases, while BDDs always branch over two cases. In each layer of a BDD a variable is selected, and the children nodes are determined by branching over this variable. For SDDs multiple variable can be selected over which the branching occurs. This results in more than two children nodes. The set of variables used for the branching is formally known as a compressed partition.

Definition 2.3 (Compressed Partition):
Let f be a function of propositional logic. The partition of variables over f into sets X and Y is *compressed*, if f can be decomposed as

$$f = [p_1(X) \wedge s_1(Y)] \vee \cdots \vee [p_n(X) \wedge s_n(Y)],$$

where the p_i's are logic formulas over X, called *primes*, and the s_i's are logic formulas over Y, called *subs*, and

- for every assignment over X exactly one prime is True and every prime is satisfiable,
- all subs represent a different logic formula.

For every partition this decomposition exists and is unique. Such a partition can be found by reducing f with every possible assignment of the variables in X, each assignment corresponding to a prime. This results in $2^{|X|}$ formulas over the variables of Y, these are the subs. By grouping subs that are equivalent and disjoining their primes, a compressed partition is obtained. For example, the formula $f = (a \wedge b) \vee (b \wedge c) \vee (c \wedge d)$ can be split over the variables $\{a, b\}$ and $\{c, d\}$, resulting in

$$\begin{aligned} f &= (a \wedge b) \vee (b \wedge c) \vee (c \wedge d) \\ &= [(a \wedge b) \wedge \textsc{True}] \vee [(\neg a \wedge b) \wedge c] \vee [(a \wedge \neg b) \wedge (c \wedge d)] \vee [(\neg a \wedge \neg b) \wedge (c \wedge d)] \\ &= [\underbrace{(a \wedge b)}_{\text{prime}} \wedge \underbrace{\textsc{True}}_{\text{sub}}] \vee [\underbrace{(\neg a \wedge b)}_{\text{prime}} \wedge \underbrace{c}_{\text{sub}}] \vee [\underbrace{\neg b}_{\text{prime}} \wedge \underbrace{(c \wedge d)}_{\text{sub}}] \end{aligned}$$

BDDs only select a single variable for the set X, resulting in two branches but SDDs can branch over all primes for a given compressed partition. Instead of a variable order, SDDs use a v-Tree. A v-Tree is a rooted binary tree, where every leaf is labeled with a variable from the logic formula. The v-Tree determines the order of partitions of variables that is used during construction. Every internal node corresponds to a partition, where X contains all leafs in the left subtree and Y contains all leafs in the right subtree. A v-Tree can be obtained from a variable order $O = [v, O']$ denoted by $T(O)$ by constructing a new root note with the left child being the leaf v and the right child being the root node of $T(O')$. The other direction is not possible, since v-Trees offer more degrees of freedom.

To represent a SDD as a graph, the nodes are distinguished between decision nodes and pairs. Decision nodes correspond to compressed partitions and have one outgoing edge for each pair of prime and sub. The pairs have one pointer for the prime and one for the sub, they point to a SDD that represents the logic formula of the prime or sub. A graphical depiction of such an SDD and its v-Tree is given in Figure 2.2.

Most of the algorithms discussed in Chapter 4 can be applied to both BDDs and SDDs. Both DDs have a similar interface and support the same set of relevant queries. There are a few exceptions regarding the variable order and v-Trees, that we will mention if they become relevant.

2.4 Automated Planning

We follow the definition of [BN95] for planning problems as Extended Simplified Action Structures (SAS+).

Definition 2.4 (SAS+ planning problem):
A SAS+ planning problem is a tuple $\Pi = (V, O, I, G)$. The components are defined as follows:

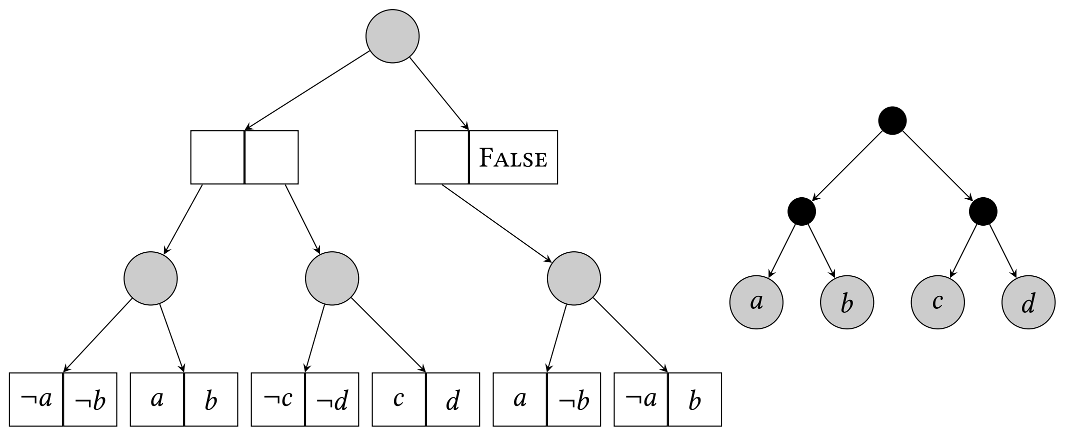

Figure 2.2: Left is an example of a SDD for the formula $(a \Leftrightarrow b) \land (c \Leftrightarrow d)$. The formula can be rewritten as $[(a \land b) \lor (\neg a \land \neg b)] \land [(c \land d) \lor (\neg c \land \neg d)]$. Decision nodes are represented by gray circles and pairs are depicted by two boxes. On the righthand side, the v-tree partitions first $\{a, b, c, d\}$ into $\{a, b\}$ and $\{c, d\}$ then $\{a, b\}$ into $\{a\}$ and $\{b\}$ and $\{c, d\}$ into $\{c\}$ and $\{d\}$

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of *state variables*. Each variable $v \in V$ has its own *domain size* d_v and *domain* $D_v = \{1, 2, \ldots, d_v\}$. An Assignment s that assigns every variable in $v \in V$ a value $s(v)$ is called a *state*, where $s(v)$ is the value of the variable v in s. The set of all possible states is denoted as S.
- $O = \{o_1, o_2, \ldots, o_m\}$ is a set of *operators*. An Operator $o \in O$ consist of *preconditions* and *effects* $o = (\mathrm{pre}_o, \mathrm{eff}_o)$. Both preconditions and effects are partial assignments over V.
- $I \in S$ is the *initial state* of the planning problem.
- G is a partial variable assignment that represents a *goal*.

For a partial variable assignment r we write r_* for the preimage of r, that means the set of variables that r is defined on. Like the goal, a partial variable assignment represents a set of states $\{s \mid s \in S : \forall v \in r_* : s(v) = r(v)\}$ and we use these two interchangeably.

An operator o is applicable in state s if $\forall v \in \mathrm{pre}_{o*} : s(v) = \mathrm{pre}_o(v)$. The operator o can be applied in s to obtain a new state $s' = o(s)$, with $s'(v) = \mathrm{eff}_o(v)$ for every variable in eff_{o*} and $s'(v) = s(v)$ else. For a given planning problem, a plan consists of a sequence of operators $p = (o_1, o_2, \ldots, o_r)$, such that o_i is applicable in s_i, with $s_i = o(s_{i-1})$, $s_0 = I$ and $s_n \in G$. This thesis only deals with unis cost problems, where every operator has a cost of one. Therefore, the length r of the plan is also called the cost of the plan. If p is a plan such that no other plan with a lower cost exists, then p is called optimal. In this thesis we are particularly interested in finding the set of all optimal plans.

The planning problem is known to be PSPACE-hard [Byl94]. Optimal plans can have an exponential size in the number of state variables. However, in practice this is rarely the case.

3 Related Work

Two areas of automatic planning are closely related to the approach of using DDs to represent sets of plans. In this chapter we explain how these fields relate to our approach and what important differences are.

3.1 Symbolic Planning

The general idea of symbolic planning is to represent sets of states implicitly through BDDs. This was first proposed by [EH01]. Operators are described by a transition relation which is used to transform one set of states into another. Applying the transition relation to a BDD is essentially the same as applying an operator on a state, except that the transition relation can apply multiple operators on multiple states at once.

States are defined by their characteristic function $s\colon X \longrightarrow \{\textsc{True}, \textsc{False}\}$ which maps a set of variables X to True or False values. A state can be interpreted as a BDD by building a conjunction out of the characteristic function, further a set of states S can be constructed by disjoining multiple conjunctions. A transition relation T represents all possible operators of the planning problem. It is defined over two sets of variables X, X', where the variables in X represent state variables from the previous states and X' represents state variables after the application of the operators from T. Basically, it defines the set of state tuples

$$T = \{(s_1, s_2) \mid s_1, s_2 \text{ are plannig states and } \exists o : o(s_1) = s_2\}$$

where s_2 can be reached by applying an operator o on s_1. By conjoining T with S a BDD over the variables $X \cup X'$ is obtained. Trough existential quantification a BDD S' over X' is constructed. It represents all states that can be reached by applying an operator from T on a state in S. By repeatedly applying T, a sequence of state sets $S_1, S_2, \ldots, S_q$ can be obtained, they correspond to layers of a breadth first search. The S_i are conjoined with a BDD representing the Goal of the planning problem. If one of these conjunctions is satisfiable the planning problem has a valid plan and a solution can be obtained by backtracking through the S_i Layers.

During the International Planning Competition 2014 symbolic planning outperformed many other competitors on the optimal planning track [EKT15]. One of the best competitors was SymBA* [Tor+14], which is a symbolic bidirectional A* planner.

Because symbolic planning and our approach both used DDs to represent structures of the planning problems, we want to highlight the key differences between the usage of DDs. The DDs of symbolic planning only represent sets of states and not sets of plans. They only contain variables for planning states but not for planning operators. This makes the DDs of symbolic planning much smaller and easier to compute but they lack the ability to perform many queries that are easy on DDs representing plans. Informally speaking, the operator variables in our approach glue together the different layers of symbolic planning together. Therefore, information about how to reach states in different layers is stored. In symbolic planning, this knowledge is lost, because it is not stored in a DD representing only a set of states. This makes our DDs more complex but also able to answer more sophisticated queries, like counting the number of states.

3.2 Top-k Planning

Interest in the complete solution space of planning problems was first shown by [RSU14]. They introduced the definition of the Top-k planning problem, which is solved by a set of k optimal plans.

Problem 3.1 (Top-k planning problem):
Given a planning Problem Π and a parameter k, find a set P of k plans such that for all plans $p \in P$ and all other plans $p' \notin P$ the cost of p is smaller or equal than the cost of p', i.e. $|p| \leq |p'|$.

Three Algorithms were proposed by [RSU14] to solve Top-k planning:

- *Iterative Replanning*: a regular planner is used to find a single plan to the planning problem. Afterwards, new problems are created that are similar to the original problem but forbid previously found plans. This way, the best optimal planners can be used to solve the Top-k problem, but the main disadvantage is, that the optimal planner has to be invoked a large number of times, which results in a long runtime.
- *Branch and Bound*: The state search space is explored by the branch and bound framework, where the cost of the kth most expansive plan can be used as an upper bound for the search. Once an additional plan has been found, the search for plans does not have to start from the beginning.
- *K^* Search*: This is an adaptation of the K-shortest path algorithm by [AL11]. It is similar to A* in the sense that is can use a heuristic to guide the search. As soon as the algorithm has created a large enough portion of the state space graph, it can find many plans in a short time.

From these three algorithms, the K* algorithm performs best.

A modification of the iterative replanning approach called Forbid-K was introduced by [KSUW18]. Multiple plans are derived from a single plan, by reordering operators and using structural symmetries [Shl+15]. They also propose a new encoding that allows multiple plans to be forbidden at once. These modification are a significant improvement over the previous iterative replanning procedure because the optimal planner has to be invoked a much smaller amount of times. It was shown to solve more problems than K^* for values of $k \leq 10000$, while K* outperforms replanning for larger k.

The latest addition to Top-k planning is the symbolic Top-k planner Sym-K [SMN20b]. It works very similar to a usual optimal symbolic planner. The main difference it that the Top-k version does not stop once a single plan is found. Plans are reconstructed by performing a greedy backward search on the obtained state layers $S_0, S_1, S_2 \ldots S_q$. Sym-K starts by picking a goal state s from S_q and iterating over all possible operators. The procedure searches for a state $s' \in S_{q-1}$, which is reached by un-applying an operator on s. By recursively continuing from s' until a state in S_0 is found, Sym-K can enumerate all possible plans. Sym-K was shown to solve more Top-k planning problems than all other planners for $k \leq 10000$

Top-q planning and diverse planning Other closely related research areas are Top-q planning and diverse planning. In contrast to Top-k planning, Top-q bounds the set of plans not by the amount but by solution quality.

Problem 3.2 (Top-q planning problem):
Given a planning Problem Π and a parameter q, find all valid plans p for Π with cost smaller or equal to q, $|p| \leq q$.

The work of [KSU20] also introduces a variation to this problem definition that does not require an explicit listing of all valid plans, but a set of representatives of equivalence classes. They define the equivalence class of a plan as all valid reorderings of the operators in the plan. Not all reorderings solve the planning problem, these are not considered to be in the equivalence class. This allows the planner to represent a much larger set of solutions with less space and is similar to our symbolic representation of plans with DDs. Nevertheless, working with these equivalence classes is much harder than working with DDs. For example, it is not always possible to determine the actual size of an equivalence class or retrieve other valid plans from it.

Diverse Planning [KS20] is focused on finding multiple sufficiently different plans, and not multiple optimal plans. The use cases overlap with the ones for Top-k or Top-Q planning. Multiple implementations use different metrics for the diversity of plans and there is no clear consensus on which metric is the best. Because the goal of diverse planning is too different from our aims, we do not compare to diverse planners in this thesis.

4 Algorithms

The process of constructing a DD from a planning problem can be divided into three major steps. An overview of these steps can be seen in Algorithm 1 and is explained in the following sections.

First, the planning problem has to be encoded into a formula of propositional logic. Different encodings are discussed in Section 4.1. In the next step, variables and clauses of the encoding have to be ordered according to a heuristic. Since the construction of a DD can be very sensitive to the used ordering, this step is crucial. A key contribution of this thesis is to determine how information about the planning problem can be used to derive a good ordering for the encoding. This step is described in Section 4.2. In the last step the DD is built according to the ordering calculated in the previous step. This is done by combining smaller DDs until they represent the whole encoding. Section 4.3 describes different possibilities in more detail.

In Section 4.4 we present possible queries on DDs for planning problems and show how Top-k and Top-q planning can be solved with our approach.

Algorithm 1: Build a DD from a planning problem

```
Data: $P$, planning problem
Result: $r$, root node of a DD
1 $V, C \longleftarrow$ Encode($P$)        // encode the planning problem, Section 4.1
2 $V \longleftarrow$ OrderVariables(V)       // determine variable order, Section 4.2
3 $C \longleftarrow$ OrderClauses(C)           // determine clause order, Section 4.2
4 $r \longleftarrow$ NewDD()
5 $r \longleftarrow$ SetVariableOrder($r, V$)
6 $r \longleftarrow$ BuildDDFromClauses(r, C)              // Build the DD, Section 4.3
7 return $r$
```

4.1 Encoding

The specific encoding that is used for the DD construction is of great importance. In this section we define and explain different encodings that translate a planning problem into a formula of propositional logic. We start with the most naive possibility and introduce improvements that build upon each other. Since not only the encoding but also the order of its clauses and variables is important for building a DD, we categorize the different parts of the encoding so that they can be ordered in the second step.

Since SAT is NP-complete and classical planning is PSPACE-complete, it is unlikely that an encoding of polynomial size exists. Therefore, planning problems are encoded incrementally. This means, that given a planning problem Π and a parameter $q \in \mathbb{N}$, a CNF instance $\texttt{Enc}(\Pi, q)$ is created, that is solvable exactly when a plan of length q exists for Π. We call q the amount of steps that a SAT instance represents.

Most of this thesis will focus on the case where q is known when creating the encoding. This can be achieved by letting a conventional optimal planner determine the length of an optimal plan and setting q relative to this optimal length. If q is not known, it is also possible to extend the formula during the construction of the DD. But this has several disadvantages: Most importantly, it is harder to order the clauses and variables if the total number of them is not know beforehand. We go into more detail on this topic in Section 4.4.

The first SAT encoding of planning problems was proposed by [KMS96]. We made small modification to their encoding, so that it works with the SAS+ formalism, which allows for variables with multiple values.

Definition 4.1 (Naive encoding)**:**
For a planning Problem $\Pi = (V, O, I, G)$ and a parameter q, the number of steps, the instance $\mathtt{Enc}(\Pi, q)$ consists of the following logic variables:

- *Planning variables:* For every step $t = 0, 1, \ldots, q$ and every planning variable $v \in V$, d_v new logic variables are introduced. These are denoted as $x_{\text{var},t,v,val}$ $(val \in \{1, 2, \ldots d_v\})$. The interpretation is that $x_{\text{var},t,v,val}$ is TRUE iff the value of the variable v in the state at step t is val.
- *Planning operators:* For every step $t = 1, \ldots, q$ and every planning operator $o \in O$ one new logic variable is introduced. They are denoted as $x_{\text{op},t,o}$ and are interpreted as TRUE iff the plan uses operator o to reach the state in step t.

The following clauses are included in the encoding:

- *Initial state:* To ensure that the initial state holds in step 0, the following unit clauses are added to the encoding. $\forall v \in V$:
$$x_{\text{var},0,v,I(v)}$$
- *Goal:* Similar unit clauses are added to ensure that the goal holds at the last step. $\forall v \in G_*$:
$$x_{\text{var},q,v,G(v)}$$
- *Exact one value:* At every step a variable should have exactly one value. This is achieved by adding clauses that ensure at least one and at most one value is set for a variable. $\forall t = 1, 2, \ldots, q, \forall v \in V$:
$$\bigvee_{d \in D_v} x_{\text{var},t,v,d} \text{ and } \forall d_1, d_2 \in D_v, d_1 \neq d_2 : \neg x_{\text{var},t,v,d_1} \vee \neg x_{\text{var},t,v,d_2}$$
- *Exact one operator:* In the same way as above, in every step exactly one operator should be applied. $\forall t = 1, 2, \ldots, q$:
$$\bigvee_{o \in O} x_{\text{op},t,o} \text{ and } \forall o_1, o_2 \in O, o_1 \neq o_2 : \neg x_{\text{op},t,o_1} \vee \neg x_{\text{op},t,o_2}$$
- *Precondition:* If an operator is applied in step t, all preconditions have to be met in the state at step $t - 1$. $\forall t = 1, 2, \ldots, q, \forall o \in O$:
$$x_{\text{op},t,o} \Rightarrow \bigwedge_{v \in \text{pre}_{o*}} x_{\text{var},t-1,v,\text{pre}_o(v)}$$

- *Effect:* If an operator is applied in step t, all its effects are visible in the resulting state at step t. $\forall t = 1, 2, \dots, q, \forall o \in O :$

$$x_{\text{op},t,o} \implies \bigwedge_{v \in \text{eff}_{o*}} x_{\text{var},t-1,v,\text{eff}_o(v)}$$

- *Frame:* No variable gets assigned a new value without an operator supporting the change. The set of operators that support the change of variable v to value d is defined as $\text{support}(v \leftarrow d) := \{o \in O \mid v \in \text{eff}_{o*} \wedge \text{eff}_o[v] = d\}$. $\forall t = 1, 2, \dots, q, \forall v \in V, \forall d \in D_v :$

$$(x_{\text{var},t,v,d} \wedge \neg x_{\text{var},t+1,v,d}) \implies \bigvee_{o \in \text{support}(v \leftarrow d)} x_{\text{op},t,o}$$

It is to note that the variables for the planning variables span $q+1$ steps, including the zeroth step, while the variables for operators only span q steps. The clauses for the *initial state* or the *goal* only effect the first or last step and all other clauses effect multiple steps. A simple visualization of the variables and clauses is given by Figure 4.1.

Not all of the logic expressions above are clauses but they can be transformed into pure disjunctions using the following identity:

$$\bigwedge_{x \in X} x \implies \bigvee_{y \in Y} y \cong \bigvee_{x \in X} \neg x \vee \bigvee_{y \in Y} y$$

A smaller number of variables and clauses is beneficial for the creation of DDs, but the naive encoding needs a lot of variables and clauses. Let $D := \max\{d_v \mid v \in V\}$ be the size of the biggest domain and $E := \max\{|\text{pre}_{o*}| \mid o \in O\} \cup \{|\text{eff}_{o*}| \mid o \in O\}$ be the size of the biggest precondition or effect. For the planning variables at most $(q+1) \cdot D \cdot |V|$ new variables are created and for the operators $q \cdot |O|$ new variables are created. For the *initial state* and *goal* clauses at most $|V|$ unit clauses are added. The clauses that guarantee *exact one value* or *operator* to be true need $q \cdot D^2 \cdot |V|$ and $q \cdot |O|^2$ new clauses. *Preconditions* and *effects* both need at most $q \cdot E \cdot |O|$ clauses. There are $q \cdot D \cdot |V|$ *frame* clauses.

In total there are $\mathcal{O}(q(D \cdot |V| + |O|))$ variables and $\mathcal{O}(q \cdot (D^2 \cdot |V| + |O|^2 + E \cdot |O|))$ clauses. For most planning problems D and E will be relatively low. The biggest influence on the encoding size comes from the quadratic growth of the operators. This poses a problem for bigger problems with more operators.

Ladder encoding In the naive encoding, constraints of the form *at most one variable/operator* is true introduce a quadratic blowup in the number of clauses. This can be circumvented by using a more advanced encoding. A common way to encode at most one constraints is the *ladder encoding*. We follow the description of the ladder encoding from [HN13]

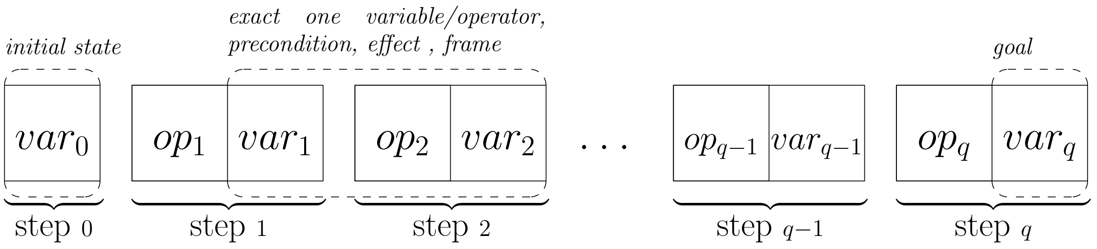

Figure 4.1: Visualization of variables and clauses of the naive encoding. Variables are represented by a box, each step contains planning variables and planning operators. Clauses and the variables they contain are represented by the dashed regions.

Definition 4.2 (Ladder Encoding):
Given the variables $x_1, x_2, \ldots, x_n$, the ladder encoding consists of $n-1$ new ladder variables $y_1, y_2, \ldots, y_{n-1}$ and two sets of clauses are added:

- $\forall i = 1, 2, \ldots n-2 : y_i \vee \neg y_{i+1}$,
- $\forall i = 1, 2, \ldots n : (y_i \wedge \neg y_{i-1}) \Leftrightarrow x_i$.

The idea is that the first set of clauses forbids two consecutive ladder variables to be FALSE and then TRUE. Because of this, the sequence of ladder variables will be TRUE up to an index k and then stay FALSE. The second set of clauses ensures that only the variable x_k will be set to true, where the switch from TRUE to FALSE of the ladder variables occurs. This encoding reduces the number of needed clauses from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ but also introduces $\mathcal{O}(n)$ new variables.

Direct exact one encoding Another possibility to realize *at most one* constraints is by constructing the corresponding BDD directly. One way to achieve such a construction is explained by [FBAH20]. A BDD on the variables $x_1, x_2, \ldots x_n$ representing an exact one constraint will always look the same under a given variable order. Figure 4.2 gives an example of such a BDD, it uses $2n+1$ nodes (including the 0 terminal node). This BDD can be constructed directly by Algorithm 2 without the detour of constructing any clauses. We only implemented this procedure for BDDs and not SDDs.

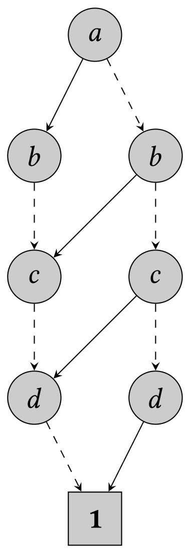

Figure 4.2: Example of a BDD that ensures that exactly one variable in $\{a, b, c, d\}$ is TRUE. Edges to the 0 terminal node are omitted for clarity.

The algorithm constructs the BDD bottom-up, starting with the variables closest to the terminal nodes. To prevent reordering, the variables have to be handled in the order given by the BDD. Two sets of internal nodes are created, corresponding to the left and right column in Figure 4.2. Internal nodes are created by a call to `BDDIfThenElse`(a, r_1, r_2) which returns a node for variable a whose 1 edge points to the node r_1 and the 0 edge points to r_2. The first set of nodes represent BDDs where one variable has already been set to TRUE and the second set represents BDDs where no variable has been set to TRUE.

The direct construction is an improvement over the naive and the ladder encoding. It does not introduce a quadratic blowup and needs no additional variables.

Algorithm 2: Construct exact one constraint directly as BDD

```
Data: $x_1, x_2, \dots x_n$, variables in a BDD
Result: $r$, root node of a BDD
1 sort variables $x_1, x_2, \dots x_n$ according to the BDD variable order;
2 $one \leftarrow$ BDDTrue();                                // constant true node
3 $zero \leftarrow$ BDDFalse();                              // constant false node
4 for $i \leftarrow n$ to 2 do
5     $zero \leftarrow$ BDDIfThenElse($x_i$, $one$, $zero$);          // zero variables true
6     $one \leftarrow$ BDDIfThenElse($x_i$, BDDFalse(), $one$);        // one variable true
7 end
8 return BDDIfThenElse($x_1$, $one$, $zero$);
```

Binary Encoding In the previous paragraphs we proposed methods that reduce the number of clauses. The binary encoding reduces the number of variables, while the complexity of the clauses increases. Instead of encoding planning variables and operators unary, they can be represented with a logarithmic amount of bits. Only one operator variable can be TRUE in a single step therefore it is unnecessary to use a variable for every operator. Instead, $\log(|O|)$ variables can be used, to represent the bits of a binary number, selecting exactly one operator. The same is true for the different values of the planning variables.

For n values, $g(n) := \lceil \log_2(n) \rceil$ is the minimum number of necessary bits to represent every value. Given a set W of $|W| = n$ values and a step t, let $X_{W,t}$ be a set of $g(|W|)$ logic variables that represent the values in W at step t. W can either be the set of operators O or variable values D_v for some $v \in V$. The variable sets $X_{W,t}$ are pairwise disjunct.

A conjunction using every variable in such a set $X_{W,t}$ can be constructed by either adding the variable itself or its negation to the conjunction. Every conjunction will contain exactly $g(n)$ variables. For $w \in W$, we call $X_{W,t}[w]$ the conjunction which is true iff the value w is selected from W in step t. This mapping from value to conjunction can be arbitrary as long as it is injective. In general, it will not be bijective, since $2^{g(n)} > n$ if n is no power of two. A simple mapping is obtained by interpreting a value w as an integer and the variables in $X_{W,t}[w]$ as the bits of w in base of two. Let $X^*_{W,t}$ be set of conjunctions that represent no value.

Definition 4.3 (Binary Encoding):
For a planning Problem $\Pi = (V, O, I, G)$ and parameter q, the logic variables of $\mathtt{Enc}(\Pi, q)$ consists of the set

$$\bigcup_{v \in V, t=0,1,\dots,q} X_{D_v,t} \cup \bigcup_{t=1,2,\dots,q} X_{O,t}$$

The following clauses are included in the encoding:

- *Initial state* and *Goal:* To ensure that the initial state and goal holds, unit clauses are added, similar to the naive encoding.

$$\forall v \in V : X_{D_v,0}[I(v)] \text{ and } \forall v \in G_* : X_{D_v,q}[G(v)]$$

- *Forbid impossible values:* Some conjunctions represent no values. These are prevented by the following clauses.

$$\forall t = 1, 2, \dots, q, \forall v \in V, \forall Y \in X^*_{D_v,t} : \neg Y \text{ and } \forall t = 1, 2, \dots, q, \forall Y \in X^*_{O,t} : \neg Y$$

- *Precondition:* If an operator is applied in step t, all preconditions have to be met in the state at step $t-1$. $\forall t = 1, 2, \dots q, \forall o \in O$:
$$X_{O,t}[o] \Rightarrow \bigwedge_{v \in \mathrm{pre}_{o*}} X_{D_v,t-1}[\mathrm{pre}_o(v)]$$
- *Effect:* If an operator is applied in step t, all its effects are visible in the resulting state at step t. $\forall t = 1, 2, \dots q, \forall o \in O$:
$$X_{O,t}[o] \Rightarrow \bigwedge_{v \in \mathrm{eff}_{o*}} X_{D_v,t}[\mathrm{eff}_o(v)]$$
- *Frame:* No variable gets assigned a new value without an operator supporting the change. $\forall t = 1, 2, \dots q, \forall v \in V, \forall d \in D_v$:
$$(X_{D_v,t}[d] \wedge \neg X_{D_v,t+1}[d]) \Rightarrow \bigvee_{o \in \mathrm{support}(v \leftarrow d)} X_{O,t}[o]$$

The initial state and goal clauses are similar to the naive encoding, although multiple unit clauses are added for every value instead of just one. No *exact one is true* clauses are needed, since this is implicitly guaranteed by the binary encoding. Instead clauses that *forbid impossible values* are added. These are necessary because the clauses in $X^*_{W,t}$ represent no existing value. For correctness, it is mandatory to add these clauses for the operator values. Otherwise a solver could pick impossible operators which have no restrictions through precondition and effect clauses. The *forbid* clauses for the variable values are not mandatory, since the *preconditions* and *effects* guarantee that no impossible value is ever reached. Because the right side of the implication in the *preconditions*, *effects* and *frame* clauses contains a conjunction, they can not be translated into clauses directly. Instead, they formulate a *DNF* but this poses no problem for the construction of a DD.

In total, the number of variables is in $\mathcal{O}(q \cdot (\log(D) \cdot |V| + \log(|O|)))$. The total number of clauses + DNFs is $\mathcal{O}(q \cdot (E \cdot |O|) + D|V|)$. This number is less representative, since the logic primitives are more complex, compared to the naive encoding. Additionally, multiple unit clauses could be interpreted as a single DNF, which further reduces the number of logic primitives. The important thing is that the number of operator variables and state variables has been reduced to a logarithmic factor. Furthermore, the quadratic increase in *exact one* clauses is eliminated.

Parallel Encoding The prior encodings only allow a single operator to be selected per step. If this restriction is dropped it is possible to construct a DD with less than q steps, that still represents all plans of length q. Allowing multiple operators to be taken in a single step is known as a *Parallel Plan.* In each step of a parallel plan, not a single but multiple operators can be selected. It is important that every parallel plan corresponds to at least one actual plan. Therefore, the operators in a step of a parallel plan must not conflict with each other. Multiple notions of conflicting operators exist and they are more or less restrictive. For this thesis, we choose a more restrictive but simpler definition which was proposed by [KMS96].

Definition 4.4 (Parallel Plan):
Two partial assignments of state variables s_1, s_2 are *consistent* if there is no variable $v \in s_{1*} \cap s_{2*}$, such that $s_1(v) \neq s_2(v)$. Two Operators o_1, o_2 are *non-conflicting* if all pairs of partial assignments in $\{\mathrm{pre}_{o_1}, \mathrm{eff}_{o_1}\} \times \{\mathrm{pre}_{o_2}, \mathrm{eff}_{o_2}\}$ are consistent.

A *parallel plan* is a sequence of sets of operators $[O_1, O_2, \dots, O_r]$, such that in every set O_i no two operators are conflicting. The sets O_i are called *parallel steps.*

If one specific ordering of operators from O_i transforms the state s_i into s_{i+1} by applying the operators in order, then all possible orderings will result in the same transformation from s_i to s_{i+1}. Therefore, if $[O_1, O_2, \dots O_r]$ results in an actual plan for a planning problem for some ordering in every parallel step, then every other ordering will also result in valid plan. This way, at most $\Pi_{i=1}^{r} |O_i|$ plans can be constructed from a single parallel plan.

The concept of parallel plans can be easily used with the naive encoding. Instead of the prior *exact one operator* clauses that disallow all pairs of operators, only pairs of conflicting operators are disallowed

$$\forall t = 1, 2, \dots, r, \forall o_1, o_2 \in O, \text{ such that } o_1 \text{ and } o_2 \text{ are conflicting} : \neg x_{\text{op},t,o_1} \vee \neg x_{\text{op},t,o_2}$$

The size of the parallel encoding and the naive encoding it the same for a single step but the parallel encoding will use less steps in most cases. If a DD is constructed from this encoding, it will not represent the set of all possible plans but the set of all possible parallel plans. In most cases this is not a big problem because plans can be reconstructed from parallel plans, but certain operations on the DD (e.g. counting the number of plans) become more difficult.

It is to note that parallel plans cannot be used with the binary encoding. This is because the binary encoding implicitly only allows a single operator per step but parallel plans need multiple operators per step. The binary encoding reduces the number of variables and the parallel encoding reduces the amount of steps. While it seems difficult to use the advantages of both encodings at the same time, a compromise can be found. We call this compromise colored encoding.

Colored Encoding This encoding aims to interpolate between the benefits of the binary and the parallel encoding. The key idea is to partition the set of operators into blocks $A_1, A_2 \dots, A_s$. Per step at most one operator can be selected from each block. Because of this, a binary encoding can be used inside each A_i. The maximum size of a parallel step is bounded by s, the number of blocks in the partition. Multiple criteria can be optimized to obtain a small encoding. The most important factors are:

- The number of blocks should be kept small. Fewer blocks allow for a more efficient binary encoding and results in less variables.
- A small amount of operators from different blocks should conflict. Every conflicting pair of operators from different blocks has to be prohibited by an additional clause in the encoding. Having few conflicts between blocks results in a smaller and simpler encoding.
- Most operators inside a block should conflict with each other. Each non-conflicting pair of operators inside a block restricts the degree of parallelism of the parallel plan. In the optimal case, all operators inside a block conflict with each other. This way, no parallel plans are prohibited.

It is helpful to represent the operators as a graph in order to think about the conflicts. Each operator is represented by a node and edges are drawn between non-conflicting operators. Partitioning the graph into a small amount of blocks with few edges inside each block and many edges between blocks will result in a good encoding. An example of such a graph is given in Figure 4.3.

Optimizing different criteria will result in different encodings. In most cases it is not possible to optimize all aspects at the same time. For example, the binary encoding only uses one block but the degree of parallelism is restricted by having a lot of non-conflicting operators

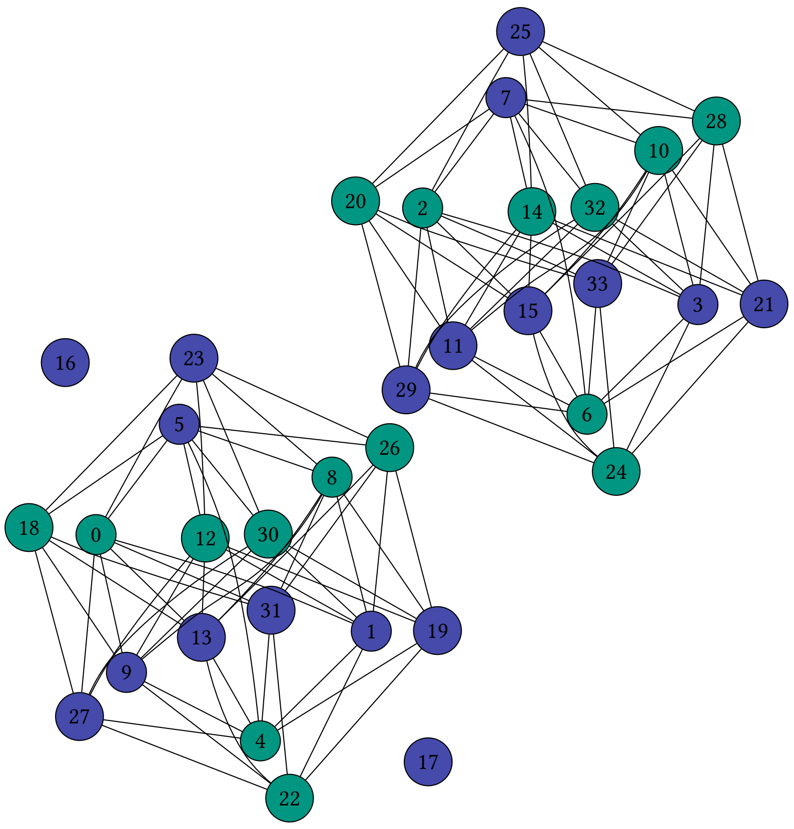

Figure 4.3: Conflict graph of the first problem from the GRIPPER Domain. The problem contains 34 operators, each represented by the nodes. Non-conflicting operators are connected with an edge. The graph can be colored with two color classes, green and blue. The two large connected components correspond to gripper operations in two different rooms. Operations from different rooms always conflict with each other. The two isolated nodes correspond to switching rooms.

in the block. The opposite is true for the parallel encoding: Each operator is assigned to its own block, so this allows the highest degree of parallelism but $|O|$ variables are needed for this. It is also possible to minimize the number of edges between blocks plus the number of non-edges inside each block. This problem is known as *correlation clustering* [BBC04].

In the approach we call *colored encoding*, we disallow all edges inside the blocks and then try to minimize the number of blocks. This is a strict improvement over the parallel encoding because in both cases the edges inside the blocks are minimized but the colored encoding will have less blocks in general. They can be determined by finding a coloring of the graph of conflicting operators. Each color class becomes one block. Because no two nodes of the same color share an edge in a proper coloring, no block will contain conflicting operators. And if the number of colors is minimal, then the number of blocks is also minimal. Finding a minimal coloring for a graph is a well known NP-hard problem [Kar72], therefore, we used a greedy approximation to determine a small coloring.

A known greedy approach for graph coloring is the DSATURE Algorithm [Bré79]. An outline of the algorithm is given in Algorithm 3. The key concept of this algorithm is the saturation degree. For a not necessarily complete coloring, the saturation degree `saturation`(v) of node v is the smallest color number not occurring in the neighborhood of v. At every step the node with the highest saturation degree is chosen and gets colored with its saturation degree. If multiple nodes have the same saturation degree, the degree of these nodes in G is used as a tiebreak.

For an efficient runtime, a priority queue is used to store saturation degrees. Additionally, each node stores a sorted list of the colors in its neighborhood. Let n be the number of nodes and m be the number of edges in the graph. In total $\mathcal{O}(n+m)$ `insert` and `update` operations are executed on the priority queue. The saturation degree of a node has to be reevaluated $\mathcal{O}(m)$ times. Using the sorted list, the new saturation can be calculated in $\mathcal{O}(\log(n))$ time. Putting everything together, a runtime of $\mathcal{O}(\log(n)(n+m))$ can be achieved.

Algorithm 3: DSature

```
Data: $G = (V, E)$, graph
Result: $C$, array of colors
1  $Q \leftarrow$ PriorityQueue()  // orders nodes by saturation degree (decreasing) and
     then degree in $G$ (decreasing)
2  for each $v \in V$ do
3      $Q$.insert(v)             // priority queue contains all uncolored nodes
4  while $Q$ is not empty do
5      $v \leftarrow Q$.pop()          // node with most colors in its neighborhood
6      $c \leftarrow$ saturation($v$)                        // saturation degree
7      $C[v] \leftarrow c$
8      update neighbors of $v$ in $Q$
9  end
10 return $C$
```

4.2 Clause and Variable Ordering

The ordering of clauses and variables is of great importance to the construction of a DD. Ordering heuristics for DDs can be divided into static and dynamic approaches. Static ordering analyses the encoding and derives an ordering before the construction of the DD starts. Dynamic ordering also takes into account the current state of the DD during construction and can change multiple times during this phase. A comprehensive overview of existing static variable ordering techniques for BDDs is given in [RK08].

Dynamic strategies are more powerful because they have more information to operate on. It is possible that an ordering is optimal for the final DD but suboptimal for intermediate products. While a static strategy would not be able to find an ordering that is good for all intermediate DDs, a dynamic approach can achieve this.

Despite all this, we decided to focus on static ordering heuristics for this Thesis. Since dynamic heuristics are executed multiple times during the construction of a DD, they have to be fast. The most prominent heuristics achieve this by having a very local view on the ordering. Static ordering is only executed once and can take longer, which allows for a more global view on the problem. Therefore, a static approach is more suited to incorporate knowledge of the planning problem into the ordering. Also, it is straight forward to apply static ordering for SDDs, because a linear order can be converted to a v-Tree. Dynamic reordering algorithms cannot be applied on a variable order and a v-tree and have to be tailored to BDDs or SDDs.

Custom Ordering The logic formulas that are created in Section 4.1 are no ordinary logic formulas. They represent a planning problem an therefore contain a lot of structure. The *custom ordering* aims to use knowledge about the underlying planning problem to come up with a better ordering than a heuristic for general logic formulas. It provides a flexible framework that defines multiple orderings on clauses. The encoding of a planning problem consists of different types of clauses reappearing in every step. We use the following abbreviations, when working with these types: *Initial State* = **i**, *Goal* = **g**, *Exact one variable* = **v**, *Exact one operator* = **o**, *Precondition* = **p**, *Effect* = **e**, *Frame* = **f**. The parts of a planning encoding can be categorized by two different characteristics:

- The planning step $t \in \{0, 1, 2, \dots, q\} = Q$, described by the clause,
- The type of constraint $u \in \{\mathbf{i}, \mathbf{g}, \mathbf{v}, \mathbf{o}, \mathbf{p}, \mathbf{e}, \mathbf{f}\} = U$, represented by this clause.

All clauses except for the **i** and **g** type clauses appear in every planning step. The **i** clauses only effect the 0th step and the **g** clauses only affect the qth step.

An order on the tuples $Q \times U$ implicitly defines an ordering on all clauses. Instead of defining an order on all the clauses, the custom order only sorts these tuples. Multiple clauses can correspond to a single tuple (t, u), in this cases the ordering between those clauses is arbitrary. The number of possible orderings on $Q \times U$ is still too large and some orderings seem unintuitive. Sorting the tuples either by the step, the type or both seems to encapsulate the structure of the planning problem more than an arbitrary ordering. Therefore, we decided to restrict the possible orderings by two main constraints:

- The steps of tuples for the same type have to be increasing. If $t_1 < t_2$, then $(t_1, u) < (t_2, u), \forall u \in U$
- The types U are partitioned into sets U_1, U_2. Tuples with a type from U_1 are ordered lexicographic first by type and then by step, tuples with a type in U_2 are ordered lexicographic first by step and then by type

Using these two restrictions it is sufficient to first define an order on U and then partitioning it into U_1, U_2, to get a total order on all tuples. An example for this would be to choose the ordering $U = [\mathbf{o}, \mathbf{v}, \mathbf{i}, \mathbf{g}, \mathbf{p}, \mathbf{e}, \mathbf{f}]$ and the partition $U_1 = \{\mathbf{o}, \mathbf{v}, \mathbf{i}, \mathbf{g}, \mathbf{f}\}$, $U_2 = \{\mathbf{p}, \mathbf{e}\}$, which results in the order

$$(\mathbf{o}, 0), \dots, (\mathbf{o}, q), (\mathbf{v}, 0), \dots, (\mathbf{v}, q), (\mathbf{i}, 0), (\mathbf{g}, q), (\mathbf{p}, 0), (\mathbf{e}, 0), \dots, (\mathbf{p}, q), (\mathbf{e}, q)(\mathbf{f}, 0), \dots, (\mathbf{f}, q).$$

The amount of possible orderings is still large but manageable. A comprehensive overview on which ordering produces the best results is given in Section 5.2.

This framework can also be used to order the variables of an encoding, since they can also be assigned a step and a type. For this thesis we only differentiated between two types of variables: variables that describe the value of a planning variables, and variables that describe operators. Because variables only have two different types, the total number of orderings on variables is much less. In the case of the ladder encoding the third type of ladder variables is added.

Force Ordering In addition to the custom heuristic which is specifically designed for planning problems, we also implemented different generic heuristics. The Force algorithm [AMS03] is a general ordering heuristic that does not rely on the logic formula to be a planning problem. It is described as a fast, simple and competitive in the survey [RK08]. We decided to use the Force heuristic, because it is easy to implement and does not depend on other external software.

The main usage of Force is to order variables, but it can also be used to order clauses. Clauses can be ordered by reversing the roles of variables and clauses. Force tries to move variables close together that occur in common clauses. This is supposed to keep the influence of variables local in the DD Graph and reduce the total number of edges. Variables are ordered by a one dimensional simulation that applies repulsing and attracting forces on the variables. Variables with common clauses are attracted to each other, while variables with no common clauses are repelled. Doing so, Force tries to minimize the average span of the clauses.

Definition 4.5 (Span):
Given a set of clauses C, a variable order O_{var} and clause order O_{cls}. Let C_v be the subset of C that contains all clauses that contain v. The span S of a clause c under O_{var} is the difference of positions in the ordering between the first and last variable of the c. The span S of a variable v under O_{cls} is the difference of positions in the ordering between the first and last clause in C_v:

$$S(c) := \max\{O_{var}(|v|) \mid v \in c\} - \min\{O_{var}(|v|) \mid v \in c\}$$
$$S(v) := \max\{O_{cls}(c) \mid c \in C_v\} - \min\{O_{cls}(c) \mid c \in C_v\}$$

Pseudocode of Force is given by Algorithm 4; it assigns each variable in V a position $O(v)$, based on clauses in C. It starts by assigning each variable a random integer position by choosing a random permutation over $|V|$. This random assignment is improved during multiple iterations. In each iteration the position of a clause c is calculated by averaging the position of all variables in the clause

$$P(c) := \left(\sum_{v \in C} O(|v|)\right) / |C|.$$

No distinction is made between positive or negative variables inside the clauses. The attracting forces between variables are simulated by determining the average position of all clauses that contain variable v.

$$O(v) := \left(\sum_{c \in C_v} P(c)\right) / |C_v|.$$

After the previous step, the coordinates in O are not necessarily integers. To obtain new integer coordinates, the variables are sorted by their coordinates, the new integer coordinate is determined by the position in the sorted order. If multiple variables have the same coordinate during sorting, they are shuffled randomly.

The initial ordering of variables can have a great influence on the quality of the resulting order. Instead of choosing a random initial permutation it is also possible to provide Force with the custom ordering of the previous step. In this case, Force would try to improve on the custom ordering by further reducing the average span of all clauses.

Bottom up The *bottom up* heuristic is another generic ordering algorithm that can only be used for clauses. It was proposed by [AMS04] in combination with the Mince variable order. The Mince order is similar to Force and both produce comparable results, but Mince uses more complex hypergraph partitioning and is slower.

The bottom up algorithm depends on the current variable ordering. It selects clauses that only affect variables in lower levels of the ordering O first. More formally, the position of a clause c is defined by $P(c) := \min\{O(|v|) \mid v \in c\}$ and the clauses are sorted by decreasing positions. The DD is built from the lowest levels to the highest, with the idea that adding clauses this way, has a more local influence on the DD and its ordering.

Algorithm 4: Force

```
Data: $V$, set of variables, $C$, set of clauses over $V$
Result: $O$, ordering of the clauses
$O \leftarrow$ RandomPermutation()                      // random or custom order
$P \leftarrow$ EmptyArray()                    // holds average position of clauses
while $i < maxIterations$ do
    for each $c \in C$ do
        $P[c] \leftarrow$ AveragePositionOfVariables($c$)
    for each $v \in V$ do
        $O[v] \leftarrow$ AveragePositionOfClauses($v$)
    $O \leftarrow$ UniqueIntegerCoordinates($O$)          // make coordinates integer
end
return $O$
```

Variable Grouping A common feature of BDD libraries is the grouping of variables. Grouping variables forces the dynamic reordering procedure to keep them together in the order. It is important, that variables are grouped in such a way as they appear in a good variable order. This can speed up the dynamic reordering, because the dynamic reordering procedure has to make less decisions. For SDDs the v-trees have a more complicated structure and we did not explore the possibilities of restricting it. We propose three different methods to group variables for BDDs:

- *Operators*: Grouping all logic for the operators of a step together,
- *Variables*: Grouping the logic variables for all planning variables of a step together,
- *Values*: Grouping the logic variables for all values of a single planning variable in a single step together.

4.3 DD Construction

The third step in building a DD from a planning problem is the actual construction. Starting from DDs that represent single variables, smaller DDs are combined into bigger ones by a sequence on `Conjoin` or `Disjoin` operations until a DD is reached that represents the whole planning problem. This section will present different strategies to group combine operations. We also investigate, how the structure of the planning problem encoding can be used for a more efficient construction.

The algorithms in this section expect an ordered CNF in most cases. Some of the encodings in Section 4.1 do not only produce clauses but also DNFs and other logic primitives, resulting in a logic formula that consist of a conjunction of different logic primitives. By minor modifications, the algorithms can be adapted to handle logic formulas that are not pure CNFs. For the sake of simplicity we keep calling these logic formulas CNFs.

Clause by Clause The most simple way to construct a DD from a set of clauses, is to build a DD for every clause and conjoin these DDs together one by one. This is also known as bottom-up construction. The procedure is outlined in Algorithm 5. It uses the `Disjoin` and `Conjoin` routines of the DD. An important characteristic of this algorithm is that the DDs which represent the clauses are relatively small, compared to the main DD with which they get conjoined. The conjoin operation usually conjoins a small DD s with a large DD r. The clauses are processed in the order obtained from Section 4.2.

Algorithm 5: Build a DD clause by clause

```
Data: $C$, ordered set of clauses
Result: $r$, root node of a DD
$r \leftarrow$ DDTrue()                 // represents whole planning problem at the end
for each $c \in C$ do                                 // according to clause order
    $s \leftarrow$ DDFalse()                          // represents a single clause
    for each $v \in c$ do
        if $v > 0$ then                               // if the variable is positive
            $s \leftarrow$ DDDisjoin(s, DDVar(v))        // disjunction with variable
        else
            $s \leftarrow$ DDDisjoin(s, DDNot(DDVar(v)))   // disjunction with negation
        end
    $r \leftarrow$ DDConjoin(r, s)
return $r$
```

Step by Step The encoding of a planning problem shows a lot of structure. Most importantly, all clauses except for the *initial state* and *goal* clauses repeat every step. The other clauses are structurally identical but use different variables. This repeating structure is used by the step by step construction, sketched in Algorithm 6. At the start, a DD is build from the subset of C which only contains *initial state* and *goal* clauses. After that, a DD for every step in the encoding is constructed by the `DDForStep` procedure. There are two possibilities to create the DD $r_{step,t}$ for a single step:

- *Clause by Clause*: The respective clauses for step t are collected and conjoined in the same way as the clause by clause construction.
- *Copy and Rename*: Since each step has structurally identical clauses, the DDs will also be structurally identical if they use the same variable order. The DD for step t can be constructed by using the DD for step $t-1$, copying it and then renaming the variables.

If $r_{step,t}$ is constructed with the clause by clause option, the step by step construction is similar to the clause by clause construction. The only difference is that the step by step construction conjoins bigger DDs with the main DD. The copy and rename approach is less similar. Although renaming the variables seems simple in theory, it can be a rather expansive procedure. If the the variable order of the renamed variables is the same as the ordering of the original variables, renaming the variables is as simple as copying the DD graph and remapping all the variables. But if the order of the variables do not match, the structure of the graph changes and the operation becomes more complex. The variable order of different steps can change, due to dynamic reordering. Consecutive steps in the encoding share variables, because of the *precondition, effect* and *frame* clauses, see Figure 4.1. For example in the naive Encoding from Definition 4.1 the variable $x_{\text{var},t-1,v,val}$ is used in the *effect* clauses for step $t-1$ and the *precondition* clauses for step t. If the order of some variables in step $t-1$ is changed, it is likely that the order will no longer match with step t. This can result in a more expansive copy and rename operation.

Algorithm 6: Build a DD step by step

```
   Data: $C$, ordered set of clauses
   Result: $r_{ini}$, root node of a DD
 1 $C_{ini} \leftarrow$ initial state and goal clauses from $C$      // possibly more clauses
 2 $r_{ini} \leftarrow$ DDFromClauses($C_{ini}$)
 3 for each $t \in \{1, 2, \ldots, q\}$ do                            // for each step
 4     $r_{step,t} \leftarrow$ DDForStep($t$)                         // clause by clause or copy and rename
 5     $r_{ini} \leftarrow$ DDConjoin($r_{ini}, r_{step,t}$)
 6 return $r_{ini}$
```

The step by step construction is less flexible regarding the clause order. Yet, the framework of the custom order can still be applied. Types in U_1 are conjoined together with r_{ini} and types in U_2 are conjoined in their respective order during the construction of $r_{step,t}$.

Bidirectional A technique from symbolic planning [SMN20a] inspired this bidirectional approach. Instead of just starting from step 0 and incrementing it, the construction is also started from step q. This can be done by using two DDs and adding increasing steps to the first DD and decreasing steps to the second DD.

The motivation is that a DD becomes exponentially more complex with the number of steps that are added to it. By having two DDs representing $q/2$ steps, rather than one DD representing q steps the total size could be significantly reduced. In addition, this approach can make more use of the restricting *goal* unit clauses. Adding unit clauses to a DD makes its size smaller in most cases. But an unidirectional approach can only use the restricting nature of *goal* unit clauses when the last step is added.

An outline of this construction is given in Algorithm 7. Two separate root nodes are constructed, one for the forward construction and one for the backward construction. In the end, both root nodes are conjoined. It is possible to use the same root node for both directions. In this case, there is more interference between both construction directions but the last `DDConjoin` can be saved. This can be significant because it conjoins two large DDs with each other, which can be costly.

Algorithm 7: Build DD Bidirectional

```
    Data: $C$, ordered set of clauses
    Result: $r_{ini,0}$, root node of a DD
 1  $C_{ini} \leftarrow$ initial state and goal clauses from $C$
 2  $r_{ini,0} \leftarrow$ DDFromClauses($C_{ini}$)
 3  $r_{ini,q} \leftarrow$ DDFromClauses($C_{ini}$)                  // possibly same as $r_{ini,0}$
 4  for each $t \in \{1, 2, \ldots, \lfloor q/2 \rfloor\}$ do          // forward steps
 5      $r_{step,t} \leftarrow$ DDForStep($t$)                        // clause by clause or copy and rename
 6      $r_{ini,0} \leftarrow$ DDConjoin($r_{ini,0}, r_{step,t}$)
 7  for each $t \in \{q, q-1, \ldots, \lfloor q/2 \rfloor + 1\}$ do    // backward steps
 8      $r_{step,t} \leftarrow$ DDForStep($t$)                        // clause by clause or copy and rename
 9      $r_{ini,q} \leftarrow$ DDConjoin($r_{ini,q}, r_{step,t}$)
10  $r_{ini,0} \leftarrow$ DDConjoin($r_{ini,0}, r_{ini,q}$)          // can be saved if $r_{ini,0} = r_{ini,q}$
11  return $r_{ini,0}$
```

Exponential The *exponential* construction tries to make more use of the *copy and rename* option. Instead of building DDs that represent a single step of the planning problem, it creates DDs that span multiple steps. In this way a sequence of DDs is constructed that represent an exponentially growing amount of. Algorithm 8 outlines this construction. The DD with root node $r_{step,2^k}$ will represent the steps $1, 2, \dots, 2^k$. This DD is copied and the variables are renamed to represent the steps $2^k + 1, 2^k + 2, \dots, 2^{k+1}$. Conjoining both DDs results in a DD that represents twice as much steps. In total $\lfloor \log_2(q) \rfloor$ DDs are. To construct a DD that captures the whole planning problem, the smallest set of DDs is selected whose total sum of represented steps does not exceed q. By renaming these DDs accordingly and conjoining them, the final DD is obtained.

Algorithm 8: Build DD Exponential

```
   Data: $C$, ordered set of clauses
   Result: $r_{ini}$, root node of a DD
1  $C_{ini} \leftarrow$ initial state and goal clauses from $C$
2  $r_{ini} \leftarrow$ DDFromClauses($C_{ini}$)   // represents whole planning problem at the end
3  $r_{step,2^0} \leftarrow$ DDForStep(1)                  // represents the clauses of step 1
4  $k \leftarrow 1$
5  while $2^k \leq q$ do                                // build DDs spanning multiple steps
6      $temp \leftarrow$ RenameVariables($r_{step,2^k}, 2^k$)
7      $r_{step,2^{k+1}} \leftarrow$ DDConjoin($r_{step,2^k}, temp$)                // twice as big
8      $k \leftarrow k + 1$
9  end
10 $size \leftarrow 0$    // counts number of steps that are currently represented by $r_{ini}$
11 while $size < q$ do                                      // conjoin DDs together
12     if $size + 2^k < q$ then                                  // If DD still fits
13         $r_{step,2^k} \leftarrow$ RenameVariables($r_{step,2^k}, size$)
14         $r_{ini} \leftarrow$ DDConjoin($r_{ini}, r_{step,2^k}$)
15         $size \leftarrow size + 2^k$
16     end
17     $k \leftarrow k - 1$
18 end
19 return $r_{ini}$
```

Under the assumption that the copy and rename operation is comparatively cheap, a lot of work can be saved since this algorithm needs less `DDConjoin` operations than in the other construction algorithms. A disadvantage of this approach is that the `DDConjoin` operations have to conjoin relatively big DDs multiple times.

4.4 Queries on DDs

Most Queries on DDs can be performed in polynomial time. Querying a DD often only takes a fraction of the time that is necessary to create it. This section proposes possible queries on DDs and their implication for the underlying planning problem.

- *Plan Counting and Plan Enumeration*: Except for the parallel encodings, there is a one to one correspondence between plans of the planning problem and solutions to the logic formula. Therefore, counting the number of models for a given DD also counts the number of possible plans for a planning problem. Enumerating plans is as easy as enumerating solutions of the DD.
- *Selecting Plans Uniformly at Random*: Instead of enumerating all plans, it is also possible to select a plan uniformly at random. This is a powerful operation, since the number of plans can grow exponentially and selecting plans at random can give an approximation of the solution space, without the need to construct all plans.
- *Cost Optimal Plans*: It is possible to find an optimal solution for a DD, given a cost function. The function determines the cost of setting a variable to True or False. These costs are assigned to the edges in a DD graph. An optimal solution corresponds to a shortest path in the graph. Although only unit-cost problems have been considered in this thesis, this makes it possible to find optimal solutions to planning problems with operator costs. Even state variables can be assigned a cost value, allowing it to find plans that avoid or prefer certain states.
- *Restricting the Solution Space*: The solution space of a planning problem can be reduced by adding clauses to the encoding. If we conjoin the DD with a clause that is True iff a specific operator was chosen, the number of represented plans shrinks. This allows us to count the number of plans that uses this operator. The most or least common operator can be determined this way.

Our planner can be used to solve the Top-k and Top-q planning problem. Both problems are closely related to plan enumeration. In this case, our planner will keep constructing a DD for an encoding that represents more and more steps, until enough plans are found. The problem is, that the total size of the encoding is not known until enough plans are found. We propose two configurations for this problem. Both methods use the length of an optimal solution from Fast Downward q' to obtain a lower bound on the number of steps.

- *Restart*: This method restarts the planner with an increased number of steps until enough solutions are found. The advantage of this approach is that the planner always knows the size of the encoding and can make the best use out of the ordering and grouping heuristics.
- *Incremental*: The incremental approach constructs the DD for q' steps and counts the number of solutions. If not enough solutions are found, it removes the goal clauses from the DD and extends the encoding by one step. This way, the planner does less redundant work, since it does not create the DD from scratch every time the number of steps is increased. A disadvantage of this approach is that adding and removing the goal clauses can disturb the variable order.

The performance of Top-k an Top-q planning, selecting several uniformly distributed plans and finding the most frequent operator by which the goal is reached is evaluated at the end of Section 5.2.

5 Evaluation

In this chapter, we evaluate our contributions from Chapter 4. We explain our implementation and test setup in Section 5.1. Afterwards, the different configurations of our algorithms are evaluated in Section 5.2.

5.1 Implementation and Experimental Setup

All algorithms are implemented in C++ and are available as the planner PLANDD[1] on GitHub. In order to translate planning problems from the `.pddl` into SAS+ format, the translation part of the planning framework Fast Downward [2] is used. Fast Downward is also used to retrieve the length of optimal plans, using A* search and the `lmcut` heuristic. BDDs are constructed through the well known CUDD[3] library and SDDs are constructed by the SDD package[4].

Experiments were performed on two different test sets. The first set is larger and was used in most cases, while the second set was used when testing a particularly large number of configurations.

- The first set contains every optimal strips unit-cost problem from the Fast Downward benchmark collection[5]. This set contains 1190 problems from 37 different domains. A timeout of 300 sec was used on problems from this test set.
- The second set is a subset of the first and contains easy problems. From the first set 200 problems have been sampled without replacement with a probability proportional to $1/t$, where t is the time it took Fast downward to solve the problem. For the easy test set, a timeout of 30 sec was used.

We used two different machines for our experiments. The first, containing two Intel Xeon E5-2683 processors with two times 16 cores, 2.1 GHz and 512 GiB of RAM and the second with an AMD EPYC Rome 7702P processor, 64 cores, 2.0 GHz and 1024 GB of RAM. Experiments that we compare against each other were executed on the same machine. As many tests as cores were run concurrently, with four cores reserved for the operating system and Runwatch[6]. Runwatch schedules the problems and enforces memory and time constraints.

[1]https://github.com/Vraier/planDD
[2]https://www.fast-downward.org/
[3]https://github.com/ivmai/cudd
[4]http://reasoning.cs.ucla.edu/sdd/
[5]https://github.com/aibasel/downward-benchmarks
[6]https://github.com/domschrei/runwatch

5.2 Experimental Results

In this section we discuss the experimental evaluation of our algorithms. We start with the parameters that have the most significant impact on the performance of our planner. After determining the best configuration of our planner we make a comparison with the standard DD compilers. At the end, a comparison is made against state of the art Top-K and Top-Q planners.

Naive BDD and SDD implementation This approach is intended to provide a baseline for further comparison with more advanced techniques. It uses a standard CNF to DD compiler, to construct the DD for the planning problem. Only the SDD package comes with a default compiler, so we implemented our own BDD compiler. Our BDD compiler uses Force to compute the variable order and the bottom up heuristic to compute the clause order. For dynamic reordering we use the shifting procedure [Rud93] which comes with the CUDD library. The default SDD compiler does not perform static variable ordering but also uses the bottom up heuristic for clause ordering. A procedure called local v-tree search [CD13] is used for dynamic reordering of SDDs. Both compilers use the clause by clause construction of Algorithm 5. Fast Downward is used to obtain the length of an optimal plan q. A CNF, representing q steps, is calculated with the naive encoding.

We tested both compilers on the large test set with a timeout of 300 seconds. Out of the 1190 problems, Fast Downward only solved 538 within the timeout. This is a practical upper bound for our planner, since it requires the solution of Fast Downward to determine the size of the encoding. Of the 538 problems solved by Fast Downward, the encoder creates a valid CNF for 423 problems. Both compilers solve a similar number of problems, with the BDD compiler constructing 66 BDDs and the SDD compiler completing 71 SDDs before the timeout. This can be seen in the left plot of Figure 5.1. The y-axis shows how many DDs were constructed within the given time limit on the x-axis.

The right plot of Figure 5.1 compares the performance of the two compilers on every individual test instance. One point is drawn for every test instance, where the x coordinate is the time taken by the BDD compiler and the y coordinate is the time taken by the SDD compiler to finish building the DD. If one of the compilers timed out, the point is drawn on the vertical or horizontal gray line. We can see that the SDD compiler is able to solve some problems significantly faster the the BDD compiler and vice versa. More specifically, the BDD compiler seems to perform better on the movie domain and the SDD compiler performs better on the psrsmall, gripper and visitall domains. A more detailed analysis of the performance of our planner per domain is given at the end in Table 5.5 and Table 5.6. In total, the DD for 76 problems can be created by either using the BDD or SDD compiler.

The divergence in performance suggests that both DD approaches can complement each other. However, it should be noted that the significance of these results is limited because both compilers react very sensitively to the variable and clause order. Changing the way the CNF is constructed will have a significant impact on both compilers. To decouple the encoding of the planning problem from the DD construction, we will determine good variable and clause orders in the next paragraph.

Variable and clause orders The clause and variable order has the most significant influence on how efficiently a DD can be constructed. In Section 4.2 we proposed using knowledge on the planning problem to order the encoding. The number of possible orders of clauses is too large and we have limited ourselves to 540 different orders from the custom ordering.

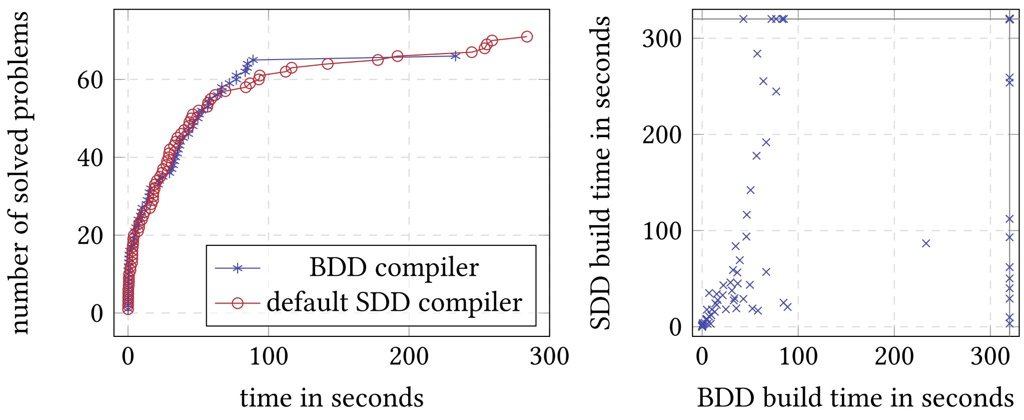

Figure 5.1: Comparing the performance of the BDD and SDD compiler.

These were tested on the smaller test set with a timeout of 30 seconds and dynamic variable reordering. We also tested four different variable orders on the large test set with a timeout of 300 seconds. Both configurations use the clause by clause construction for BDDs with the naive encoding.

The results for the clause orders can be seen in Figure 5.2. The worst orders solve only 7 problems and the best orders solve up to 70 problems, which is 4 problems more than the default SDD compiler with a ten times smaller timeout. One of the best orders sorts the types of clauses by $U = [\mathbf{i}, \mathbf{g}, \mathbf{v}, \mathbf{o}, \mathbf{p}, \mathbf{e}, \mathbf{f}]$ with the partition $U_1 = \{\mathbf{i}, \mathbf{g}\}, U_2 = \{\mathbf{v}, \mathbf{o}, \mathbf{p}, \mathbf{e}, \mathbf{f}\}$.

A gap can be seen in the histogram in Figure 5.2 between configurations that solve more than 50 problems and configurations that solve less than 36. Two properties appear to significantly reduce the performance of an order:

- sorting clauses of type **v** and **o** after clauses of type **p**, **e** and **f**,
- choosing $\mathbf{p}, \mathbf{e}, \mathbf{f} \in U_1$ and not sorting them by step but by type.

Some general insights about clause ordering for planning problems can be obtained from these results. Clauses of type **i**, **g**, **v**, **o** should be added as early as possible. They restrict the solution space the most and are likely to reduce the size of the BDD. All clauses except for the **i** and **g** type clauses should be sorted by the step and then by type. This way clauses affecting the same variables are closer together in the order, reducing their average span. This is in consistent with the rules of thumb of other ordering heuristics such as Force and Mince.

We also evaluated four different static variable orders:

- *Order A*: Sort all variables for values by step, then sort all operator variables by step,
- *Order B*: Sort all operator variables by step, then all variables for values by step,
- *Order C*: Sort everything by step and put value in front of operator variables,
- *Order D*: Sort everything by step and put operator in front of value variables.

Each order can be combined with the dynamic reordering techniques from the BDD and SDD libraries. The results of these experiments are listed in Table 5.1. It lists the number of solved problems and the average peak size of the BDD DAG (counted in number of internal nodes). For dynamic reordering, the average age of time spent on reordering is measured.

Order C and B perform comparatively well, with C performing the best and solving 99 problems, which is slightly more than the dynamic reordering approaches. Order A and B perform worse, solving only 46 and 49 problems without dynamic reordering. There is a large gap between the number of solved problems for A and B with and without dynamic

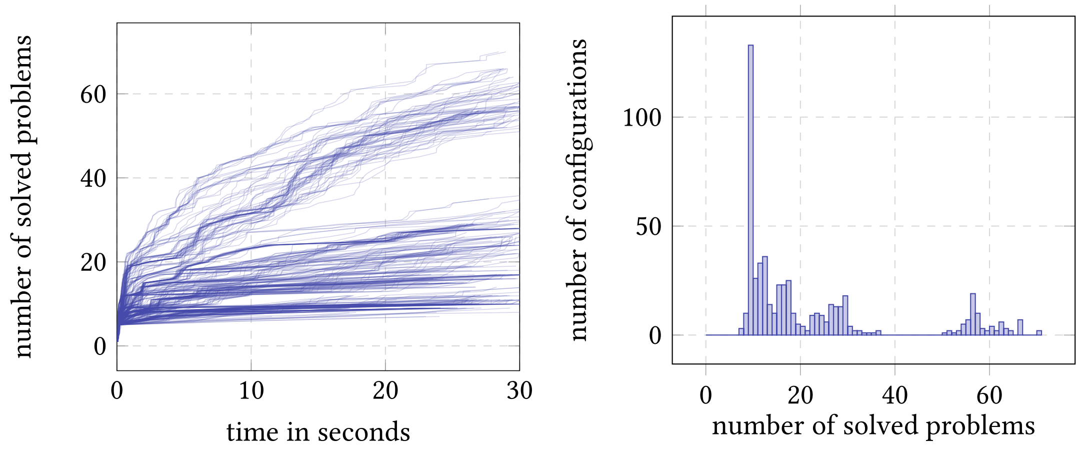

Figure 5.2: Performance of 540 different clause orders from Section 4.2 on the easy test set of 200 test cases. On the left side, one line is drawn for every clause order. On the right, a histogram showing how many configurations solved exactly x number of problems.

		order A	order B	order C	order D
no reordering	solved problems	46	49	**99**	88
	average peak DAG size	627712	638436	104304	137764
dynamic reordering	solved problems	97	97	94	95
	average % of time reordering	70.3	72.2	**67.8**	69.4
	average peak DAG size	55818	54445	**52043**	53392

Table 5.1: Results for four different static variable orderings, with and without dynamic reordering for Section 4.2.

reordering. This can be explained by the fact that A and B are *bad* orders and the dynamic reordering corrects the errors of the static order. Overall, the dynamic reordering approaches have smaller maximum DAGs, which is an indicator of a good variable order. The fact that order C performs better without reordering can be explained by the fact that reordering takes a lot of time. Up to 72% of the total construction time for the BDD is consumed by dynamic reordering in some cases. The number of solved problems for order C and the dynamic reordering approaches seem to converge, indicating that it is difficult to improve order C.

Overall, the results suggest that ordering variables by their step is better than ordering them by their type. This makes intuitive sense since variables of the same step appear in common clauses. If two variables are more than one step apart, they do not appear in common clauses. This is consistent with the rule of thumb of general static variable orders like Force which tries to reduce the average variable span.

Construction Algorithms In Section 4.3 we proposed different methods to build a DD from a set of clauses. They differ in the way they structure the conjoin operations and whether they copy the DD for a step or create it clause by clause. We tested each construction method using the best variable and clause order obtained from the previous paragraph and dynamic reordering on the large test set. The naive encoding was used to construct the CNF formula.

Figure 5.3 shows a selection of the best configurations from these experiments. No construction method was able to outperform the clause by clause construction from Algorithm 5, which solved 94 problems. The step by step construction of Algorithm 6 solved the same amount of problems when no *copy and rename* was used. This is not surprisings, since the construction only differs slightly when the step DDs are constructed clause by clause. If the DD for the next step was created by copying the previous step, the step by step construction solves only 91 problems. This can be explained by the costly copy and rename operation. On average 41% of the construction time is spent on copying and renaming the step DD and 32% of the time is spent conjoining the step DDs with the main DD.

The results for the bidirectional construction are similar. The bidirectional construction solves 90 problems with the best configuration. When the step DDs are constructed with copying, it solves only 80 problems and if both directions use the same DD root node, it solves 75 problems.

The exponential construction performs the worst. It solves 68 problems, which is the lowest of all construction configurations. The first phase, constructing the DDs that span multiple steps, was completed for 74 problems. The poor performance can be explained by the fact that this construction method has to copy and conjoin the biggest DDs, with both operations being costly. The exponential construction also has the largest average peak DAG size of 181182, despite dynamic reordering. This seems to indicate that the complicated construction of DDs spanning multiple steps disturbs up the variable order.

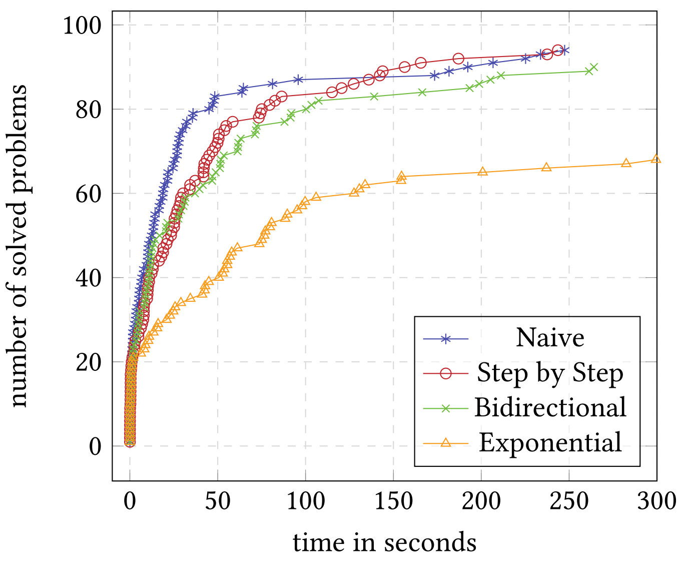

Figure 5.3: Performance of different construction algorithms from Section 4.3.

All in all, the clause by clause construction performs the best. It is the simplest construction method. The other construction procedures seem to conjoin DDs that are too large or add too much complexity and disturb the variable order.

Encoding The encoding of the planning problem has a great influence on the time it takes to construct the DD. In Section 4.1 we proposed different encodings, that aim to reduce the size either by reducing the number of clauses, variables or steps. We evaluated these encodings with the best clause and variable order together with the clause by clause construction on the large test set.

Using the binary encoding from Definition 4.3 instead of the naive unary encoding has the greatest benefit on the performance of our planner. It is possible to encode either the planning variables, operators or both binary. Since the addition of clauses that prohibit impossible variable values is optional, we tested configurations with and without these extra clauses. Figure 5.4 shows a selection of these configurations. If only the variable or the operators are encoded binary, the planner solves 108 or 127 problems respectively. If both are encoded binary, a total of 152 problems are solved, and if impossible variables are also forbidden, the planner constructs the DD for 167 problems. The main reason for these improvements is the reduced number of variables. Although the clauses for the binary encoding are more complicated, a reduction in variables leads to a smaller graph size and faster DD operations. This reduction in encoding size can be seen in Table 5.2. The average encoding has 45 times less clauses and 8 times less variables. The encoder is also able to construct the CNF for more problems than the naive encoding. It constructs the CNF for 538 problems, which is 33 problems less than the practical maximum of 538 problems solved by Fast Downward.

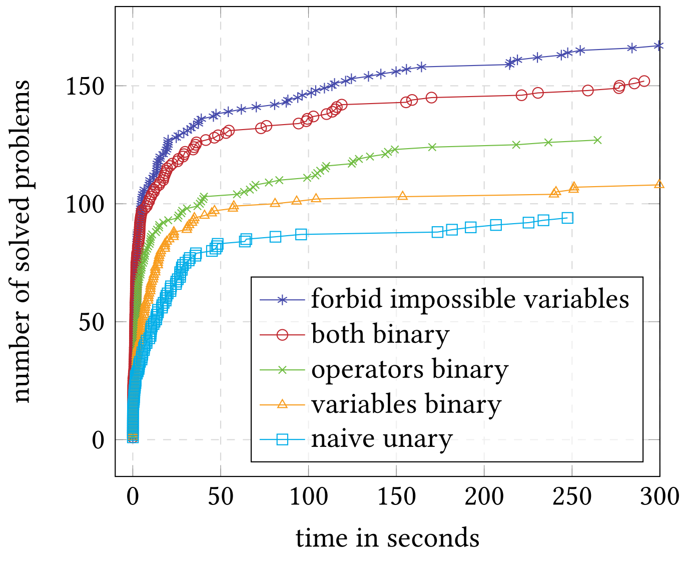

Figure 5.4: Comparing performance of different binary encodings.

Encoding exact one constraints with the ladder encoding of Definition 4.2 or directly as a BDD by Algorithm 2 reduces the size of the encoding but does not manage to significantly outperform the naive encoding. As expected, the direct encoding has a similar amount of clauses as the binary encodings but as many variables as the unary encoding, see Table 5.2. It solves slightly more problems than the unary encoding which can be explained by the fact that it needs less time to construct the CNF. The direct encoding manages to create the CNF for the most problems out of all encodings. The ladder encoding has twice as many clauses and variables as the direct encoding which is the most variables out of all the encodings. It solves 44 problems, which is the lowest amount for all encodings. The poor performance is due to the fact that all the added ladder variables have to be integrated into the variable order. We tried different ways to order them but none of them performed well. With more effort the variable order of the ladder encoding could be improved but it is unlikely to perform better than the direct way to construct exact one constraints.

configuration	solved problems	#constructed CNFs	average #clauses	average #variables
naive unary	94	427	5976958	13604
ladder encoding	44	**517**	335355	27049
direct exact one	100	**517**	155976	13604
variables binary	108	426	5759469	11647
operators binary	127	505	146997	4083
both binary	152	505	**117645**	**1745**
forbid impossible variables	**167**	505	130804	**1745**

Table 5.2: Number of solved problems and size of the encoding for different encoding configurations. The table contains three unary encodings and four binary encodings

We also proposed two encodings that try to reduce the number of steps, the parallel encoding and the colored encoding. In the previous experiments we used Fast Downward to compute the length of an optimal plan q but the length of an optimal parallel plan q' is usually less than q. Therefore, we used our own planner to determine q'. This comes with two disadvantages: First, the planner executes more conjoin operations, because it has to check at each step whether the goal is achieved, and second, it is harder to determine a good ordering of clauses and variables if the full encoding is not known at the beginning. The parallel encoding manages to solve 106 problems and the colored encoding solved 131 problems, see the left plot of Figure 5.5.

In most cases, the number of color classes found by Algorithm 3 is relatively small. On average 11.8 color classes are needed, with the median number of color classes being 4 and the most needed classes being 171. This can be seen in the right histogram in Figure 5.5. A smaller number of color classes results in an encoding with fewer variables, this supports the better performance of the colored encoding compared to the parallel one. We believe that using a SAT-based planner to determine q' can further improve the performance of both parallel encodings.

Across all encodings, reducing the number of variables has the greatest impact on the amount of solved problems. The number of clauses and their complexity are of secondary importance.

Generic ordering algorithms Multiple static ordering heuristics already exist for BDDs. In this paragraph we want to evaluate how well they work for encodings of planning problems. All experiments use the clause by clause construction with the best binary encoding and dynamic reordering.

A key concept of the Force algorithm is the span of a clause or a variable. The attracting forces of the heuristic aim to minimize this metric. We evaluated four different configurations of the ordering procedure. Force starts with a random permutation to improve the ordering, we also propose to use our own custom order for this initial permutation. We distinguish between applying Force on the variable and the clause order. Table 5.3 shows the results of the experiments together with the median clause and variable span. Force managed to improve the clause/variable span of the custom order when they were used together. However, if Force starts from a random permutation, the average span is larger than just using the custom ordering alone. Interestingly, Force with an initial random permutation and not the custom order performs worse for clause ordering but better for the variable order, although

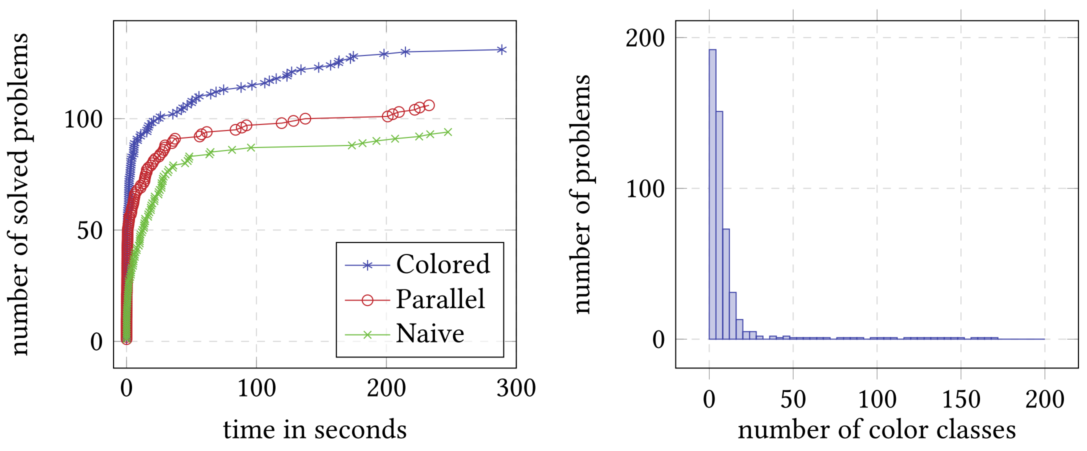

Figure 5.5: Comparing performance of different parallel encodings (left). Number of color classes for different problems (right).

the average span increases in both cases. The good performance in the case of the variable order, could be explained by the dynamic reordering, which fixes suboptimal initial orders. This seems to indicate that the clause/variable span is not sufficient to determine a good ordering. The best Force configuration solves 170 problems, which is slightly more than the custom ordering. Overall, Force is a competitive alternative for ordering the variables of planning problems. The clause ordering of Force can benefit greatly from using the custom order during initialization.

configuration	solved problems	median clause span	median variable span
custom ordering	167	2556	46
Force random variable ordering	**170**	2556	94
Force random clause ordering	84	3977	46
Force custom variable ordering	**170**	2556	**24**
Force custom clause ordering	168	**1455**	46

Table 5.3: Number of solved problems for the custom ordering and four different Force configurations. The table also shows the median of the average clause/variable span of the orderings.

Figure 5.6 visualizes two Force variable orders on the first Gripper problem with four steps. The first order uses the custom order as a starting point for Force, the second one uses a random permutation. The variables are drawn as dots on a horizontal line with x-coordinates according to the ordering. For each clause, a dot is drawn above the horizontal line with the x-coordinate being the average of its variables and y-coordinate proportional to its span. Each clause is connected to its variables by a line. In the first image, the four steps of the planning encoding are easily visible but in the second image this structure is broken up. The figure also shows the greater average span of clauses when a random initial permutation is used.

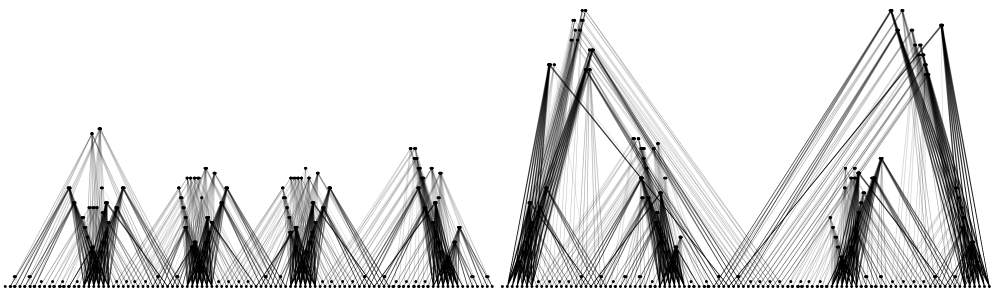

Figure 5.6: Visualization of two different variable orders on a planning problem encoding with four steps. On the left side, Force with custom order. On the right side, Force with an initial random permutation.

Using the bottom up heuristic to order the clauses of the encoding is not feasible at all. In our experiments, the planner was only able to solve 101 problems, using the bottom up order. Similar to the Force order with a random permutation, it is not able to capture the structure of the planning problem.

Overall, Force is a competitive alternative for ordering the variables of planning problems. The clause order of Force can benefit greatly from using the custom order during initialization. No variable order solves significantly more problems than our custom approach. And no other clause ordering heuristic is able to produce a good order without our custom order.

Variable Grouping We experimented with all suggested possibilities to group variables. The SDD package does not support a similar feature and thus only our BDD planner can make use of variable grouping. CUDD uses the shift algorithm to reorder the groups and also uses it within each group. The clause by clause construction, binary encoding and custom order is used in these experiments, the results can be seen in Table 5.4.

configuration	solved problems
no grouping	167
operator grouping	168
variable grouping	137
value grouping	**173**

Table 5.4: Number of solved problems for different variable groupings.

Grouping the logic variables for the values of a planning variables together has the best effect on the number of solved problems. It uses the smallest variable groups and solves 173 problems, which is the most of all previous configurations. Grouping all logic variables for the planning variables of a step together has a negative impact on the performance. This configuration solves only 137 problems. The results provide insights into good variable orderings of planning problems. Grouping all logic variables for planning variables together is worse than grouping the logic variables for the planning values and grouping operators has no significant effect. This seems to indicate that in a good ordering the operator variables form a consecutive block. However, the logic variables for individual planning variables have to be split up and either be placed before or after the operator block.

Improvements over naive implementation We started this section with an evaluation of the standard compilers for the construction of DDs and went on to present several improvements. Using the binary encoding and the custom order to sort the variables and clauses offers the biggest improvements on the number of solved problems. If Force is used to order the variables and some logic variables are grouped, the performance can be increased slightly more. Figure 5.7 compares our best BDD and SDD configuration against the naive implementation. The BDD configuration solves 173 which is 107 problems more than the naive implementation. On problems solved by both planners, the best BDD configuration achieves an a speedup of 43.6. The best SDD configuration solves 158 problems, which is 87 problems more than the default SDD compiler. Our SDD configuration has an average speedup of 76.7.

This is a significant improvement over the standard compilers. In particular, the improvements for clause orderings show that we can successfully exploit the structure of planning problems to improve the performance of DD generation.

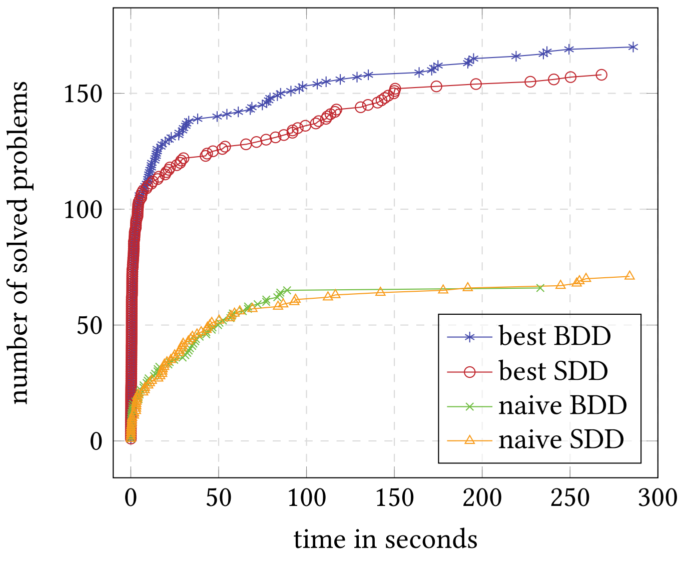

Figure 5.7: Performance of the best BDD and SDD configurations against the standard compilers.

We can construct a theoretical portfolio solver that always selects the best configuration for a given planning problem. In this scenario all our proposed configurations can solve a total of 176 different planning problems from the large test set, which is 32.7% of the problems solved by Fast Downward.

Top-k and Top-q comparison As mentioned in Section 3.2, two areas of automated planning are closely related to the types of queries our planner can answer: Top-k and Top-q planning. Since no existing planner can answer exactly the same queries as our approach, we compare planDD with planners from these two areas.

We selected the Sym-K [SMN20a] and the K^* planner [KSUW18] as Top-k competitors. Sym-K is the best existing Top-k planner and K^* performs well for large values of k. We tested them against our planner planDD on the large test set. Both competitors write every plan they find to disk, for large values of k this takes up a significant amount of search time. Since our planner represents the solution space as a DD, it does not need to write every plan to disk. To make the comparison as fair as possible, we have modified the source code of both competitors, so that they skip writing the plan to disk. They still have to enumerate each plan in memory but this can save a large amount of time. We tested the competitors against two different configurations of planDD, the restarting and the incremental configuration from Section 4.4.

A common metric for evaluating Top-k planners is the k-coverage. For a given timeout and k it counts for how many problems the planner was able to find at least k plans. Each planner was given a timeout of 300 seconds and was tasked to find $k = 10^9$ plans. The k-coverage of the two competitors and two of our configurations can be seen in Figure 5.8. The restarting configuration scales better to larger ks than the incremental one, with the likely reason being, that the restarting approach can use better variable orders than the incremental configuration.

We can also see that the existing planners significantly outperform our approach for small values of k. When the planner is tasked to find a single plan, our configuration finds a plan for 170 problems, K^* finds a plan for 329 problems and Sym-K solves 445 problems. For $k = 2 \cdot 10^4$ plans, K^* overtakes Sym-K and finds enough plans for 325 problems. Sym-K outperforms our approach up to a value of $k = 5 \cdot 10^5$ and K^* outperforms planDD up to $k = 10^7$. An evaluation per domain is given in Table 5.5, 5 domains were omitted because no planner solved a single problem in this domain. Our planner is outperformed on almost all domains for $k \leq 10^6$. However, no competitor is able to find solution sets containing more than $5 \cdot 10^7$ plans.

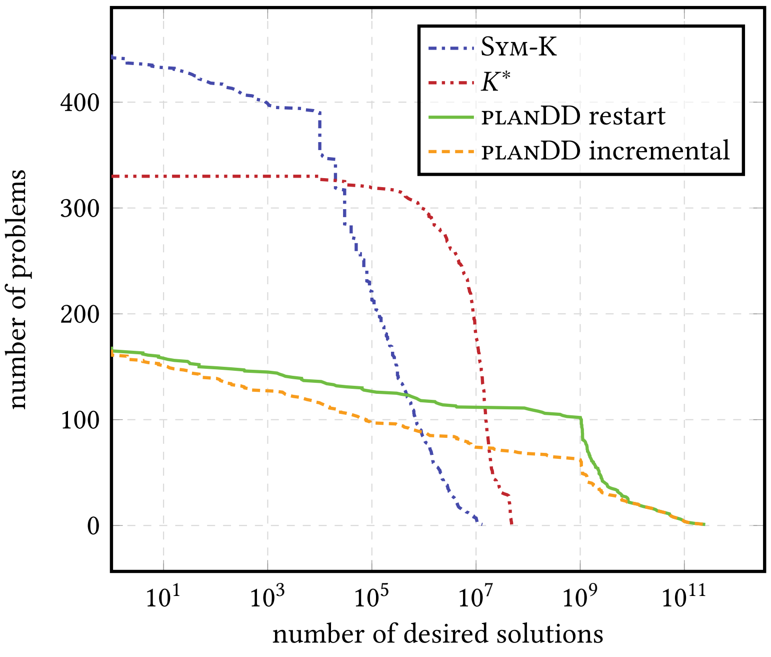

Figure 5.8: K-Coverage for different Top-K planner. A timeout of 300 seconds was used together with $k = 10^9$.

We also wanted to compare the performance of our solver on the Top-q planning problem. Unfortunately, we were unable to run the competitor Forbid-Q [KSU20]. We had problems with Forbid-Q exhausting the `/tmp` directory and crashing as a result. Therefore, we only performed the experiments for our planner and will use the results from the paper [KSU20] (Table 1, column tq) to compare the two planners. The test set-up is slightly different in the paper: It also tested Forbid-Q on non unit-cost problems and used a larger timeout of 3000 seconds. This planner solves the unordered version of the Top-q planning problem problem, were only a single plan is needed to represent all reorders. This is similar to our symbolic representation of solution sets. We tested our planner on four different relative quality bounds $q \in \{1.1, 1.2, 1.3, 1.4\}$ with the restarting configuration, the results can be seen in Table 5.6.

A precise comparison between both planners is not possible due to of the different test setups. Nevertheless, a significant difference in the performance on some domains can be identified. In fact, the domains where the planners perform well are somewhat complementary. Forbid-Q solves significantly more problems of the airport, logistics00, mprime and mystery domain, while planDD solves more problems on miconic, movie, openstacks and psr-small. A large proportion of the problems solved by our planner come from relatively easy domains with a large solution space, like the movie domain. These have a large number of solutions that use different operators, which can not be represented by the reorders of Forbid-Q. We would like to note that the output of planDD is more powerful than that of Forbid-Q. A DD representing all plans can be used more easily to perform queries than the equivalence classes of Forbid-Q. The results suggest that planDD is a competitive alternative for Top-q planning on some domains.

Depending on the length, storing an average plan explicitly needs about 1KB of memory. To evaluate the needed size to represent the solution sets with DDs, we let planDD output the final DD for every problem as a `.dot` file in the graph description language[7]. For the parameter $q = 1.1$ the average size of these files is 409KB, for $q = 1.4$ the average size is 1.6MB. For solution sets containing millions of plans this is a significant improvement.

Other queries on DDs Our DDs can answer more queries than just solving Top-k and Top-q planning. In Section 4.4 we proposed different queries on DDs that have useful applications for the planning problem. As a proof of concept, we have implemented two of these queries. The first one, randomly selects up to 1000 uniformly distributed plans. The second one, finds the most common last operators in all plans, by conjoining the DD with unit clauses representing an operator and counting the number of remaining solutions.

Both implementations construct a DD representing all optimal plans to perform the queries on. These DDs represent solution sets of a size between 1 and 10^{11}. We kept the time limit of 300 seconds to construct the DD and perform the query. Out of the 170 problems where our planner finished the construction of the DD, it was able to sample the random plans for 166 problems. On average it took 3.3 seconds to sample the set. Our planner was able to determine the most common operator for all 170 problems and it took 0.07 seconds on average. This shows that most queries can be performed in a short amount of time once the DD is constructed.

[7]https://graphviz.org/doc/info/lang.html

Planner	Sym-K			K^*			planDD		
Domain \ k	10^3	10^6	10^9	10^3	10^6	10^9	10^3	10^6	10^9
airport	**14**	**9**	0	12	0	0	1	0	0
blocks	**18**	**18**	0	**18**	2	0	6	5	**3**
depot	**4**	**3**	0	**4**	0	0	1	1	**1**
driverlog	6	**6**	0	**11**	0	0	1	1	0
freecell	**14**	**14**	0	8	0	0	0	0	0
grid	**1**	0	0	0	0	0	0	0	0
gripper	7	**7**	0	**20**	1	0	2	1	**1**
hiking14	8	**8**	0	**13**	1	0	2	1	**1**
logistics00	10	**10**	0	**16**	2	0	2	1	0
logistics98	2	**2**	0	**3**	0	0	0	0	0
miconic	45	**40**	0	**81**	15	0	24	20	**20**
movie	**30**	**30**	0	**30**	**30**	0	**30**	**30**	**30**
mprime	7	**2**	0	**8**	0	0	1	0	0
mystery	**17**	**14**	0	**17**	0	0	4	0	0
nomystery11	**8**	**8**	0	**8**	0	0	2	0	0
openstacks	**7**	**6**	0	**7**	0	0	5	5	**5**
pathways	4	**4**	0	**5**	0	0	1	1	0
pipesworld-nt	**13**	**13**	0	8	0	0	2	0	0
pipesworld-t	**7**	**7**	0	6	0	0	2	1	0
psr-small	47	**44**	0	**49**	15	0	36	29	**24**
rovers	4	**4**	0	**14**	3	0	4	3	**2**
satellite	4	**4**	0	**7**	1	0	2	2	**1**
snake18	**1**	0	0	0	0	0	0	0	0
storage	**14**	**13**	0	13	2	0	7	5	**4**
termes18	6	**6**	0	**7**	0	0	0	0	0
tidybot11	**10**	**6**	0	3	0	0	1	0	0
tidybot14	**2**	0	0	0	0	0	0	0	0
tpp	5	**5**	0	**8**	4	0	4	4	**3**
trucks	4	**4**	0	**5**	0	0	0	0	0
visitall11	9	**9**	0	**11**	5	0	5	5	**5**
visitall14	3	**3**	0	**6**	0	0	0	0	0
zenotravel	7	**7**	0	**8**	0	0	2	2	**2**
$\Sigma(1190)$	**406**	81	0	338	**336**	0	147	117	**102**

Table 5.5: K-coverage for different planners and values of k. Best performing planner for a given domain and k is highlighted. Domains where no planner solved a problem are omitted.

Planner	planDD			
Domain \ q	1.1	1.2	1.3	1.4
AIRPORT	6	6	6	6
BLOCKS	8	7	6	6
DEPOT	1	1	1	1
DRIVERLOG	1	1	1	1
GRIPPER	2	1	1	1
HIKING	1	1	1	1
LOGISTICS00	2	1	1	1
MICONIC	20	20	20	17
MOVIE	30	30	30	30
MPRIME	1	1	1	1
MYSTERY	9	8	6	6
NOMYSTERY	2	2	2	2
OPENSTACKS	4	0	0	0
ORGANIC-SYNTHESIS	7	7	7	7
PATHWAYS	1	1	1	1
PIPESWORLD-NOTANK	1	1	1	1
PIPESWORLD-TANK	2	2	2	2
PSR-SMALL	31	29	28	27
ROVERS	4	3	3	3
SATELLITE	2	2	2	2
STORAGE	7	7	6	5
TIDYBOT	1	1	1	1
TPP	4	4	4	4
VISITALL	5	5	5	5
ZENOTRAVEL	4	2	2	2
$\Sigma(1190)$	156	143	138	133

Table 5.6: Evaluating the performance of planDD for Top-q planning. Domains where no configuration solved a problem are omitted.

6 Conclusion

In this thesis we have presented new ways of representing sets of plans as Decision Diagrams. This compact representation of the solution set opens up many possibilities for queries. We have implemented our planner planDD to investigate different strategies for generating DDs. By improving the encoding and its ordering, we were able to achieve significant improvements over general DD compilers. We were able to successfully use knowledge of the planning problem to find a good ordering of its encoding. This shows that the structure of a planning problem can be used to speed up the construction of DDs. We also investigated the effects of general heuristics on DD construction. Existing generic variable orders have a slight advantage over our own order, but no clause ordering compares to our ordering tailored to the planning problem.

A significant contribution is made by planDD for solving simple problems with large solution spaces. This can be seen in the Top-q planning problem. In some domains planDD performs particularly well, managing to generate solution sets for problems that the existing planners cannot solve.

Nevertheless, the construction of DDs for planning problems remains a challenging problem. In particular, the large number of variables in many encodings of larger planning problems makes the generation of DDs challenging. Existing Top-k approaches can solve much harder problems while allowing for a similar set of queries. This makes it difficult to use planDD as a competitive planner for such problems. Our planner dominates only for simpler problems with very large solution spaces of sizes larger than 10^6 plans.

We believe that with further work our approach can be made competitive for at least moderately large solution spaces. In particular, the use of parallel plans and a better ordering of the encodings leave room for improvement.

6.1 Future Work

Our planner already contributes to Top-q planning for some domains, but does not scale well on harder problems. Even our best configurations have difficulties solving problems of practical size. Our main goal is to make planDD competitive enough to solve problems of moderate size and to make practical use of the wide range of queries it provides. In this section we would like to present possibilities for future improvements and applications of our planner.

Further improve the clause and variable order Our custom order is already producing good orders but the results of the Force and variable grouping experiments show that there is still room for improvement. The use of causal graphs, which show dependencies between planning variables, could help to order logical variables even better. We have also not yet tried to apply techniques of symbolic planning to our problem. Symbolic planning does not need to order operators, but the variable ordering can be used for our approach. It may also be helpful to use encoding techniques of symbolic planning for the state variables of our approach.

Improving the parallel encoding Our coloued encoding has proven to be a significant improvement over the naive encoding, but it solves fewer problems than the binary encoding. A major reason for this is that the length of an optimal parallel plan is not known in advance. This problem could be easily solved by using a SAT-based planner in the preprocessing phase to find out the length. It also seems fruitful to use a stronger version of parallel plans. This can further reduce the number of necessary steps and variables. With enough effort, it may be possible for the parallel encoding to outperform the binary encoding.

Getting more use out of the features of SDDs In some of our approaches, we have not fully exploited the possibilities of SDDs and have focused more on BDDs. This is partly because the BDD library offers a more sophisticated interface for DD manipulation than the SDD package. Nevertheless, we have left out some opportunities to adapt the structure of v-trees to planning problems. Manipulating the v-tree allows for more freedom when trying to model the structure of a planning problem. It may also allow us to simulate the grouping of variables for SDDs. If we can make better use of the strengths of SDDs, they could be more efficient than our BDD configuration.

Solving non unit-cost problems Throughout the work, we have limited ourselves to unit-cost problems. One way of solving non unit-cost problems with DDs has already been presented in Section 4.4. We have not yet implemented this feature, but using a cost function for the edges of the DD opens up new possibilities for solving non unit-cost problems. There are some considerations to be made on how to determine the size of the DD, since it no longer depends on the length of a plan but on its cost. Implementing this idea would allow us to identify non unit-cost domains where our planner works particularly well.

Using DDs as a heuristic for planning In section Section 1.1, we suggested that our approach could help to develop heuristics for automated planning. We would build on the idea of planning pattern databases. Reducing a planning problem to a simpler one and solving it completely fits well into the scheme of our approach. However, it remains to be seen whether PLANDD is able to represent problems hard enough to make meaningful decisions for heuristics.

Bibliography

[AL11] Husain Aljazzar and Stefan Leue. “K*: A heuristic search algorithm for finding the k shortest paths”. In: *Artificial Intelligence* Volume 175.18 (2011), pp. 2129–2154.

[AMS03] Fadi A Aloul, Igor L Markov, and Karem A Sakallah. “FORCE: a fast and easy-to-implement variable-ordering heuristic”. In: *Proceedings of the 13th ACM Great Lakes symposium on VLSI.* 2003, pp. 116–119.

[AMS04] Fadi A Aloul, Igor L Markov, and Karem A Sakallah. “MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation.” In: *J. Univers. Comput. Sci.* Volume 10.12 (2004), pp. 1562–1596.

[BBC04] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. “Correlation clustering”. In: *Machine learning* Volume 56.1 (2004), pp. 89–113.

[BN95] Christer Bäckström and Bernhard Nebel. “Complexity results for SAS+ planning”. In: *Computational Intelligence* Volume 11.4 (1995), pp. 625–655.

[Bov16] Simone Bova. “SDDs are exponentially more succinct than OBDDs”. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 30. 1. 2016.

[Bré79] Daniel Brélaz. “New methods to color the vertices of a graph”. In: *Communications of the ACM* Volume 22.4 (1979), pp. 251–256.

[Bry86] Randal E Bryant. “Graph-based algorithms for boolean function manipulation”. In: *Computers, IEEE Transactions on* Volume 100.8 (1986), pp. 677–691.

[BW96] Beate Bollig and Ingo Wegener. “Improving the variable ordering of OBDDs is NP-complete”. In: *IEEE Transactions on computers* Volume 45.9 (1996), pp. 993–1002.

[Byl94] Tom Bylander. “The computational complexity of propositional STRIPS planning”. In: *Artificial Intelligence* Volume 69.1-2 (1994), pp. 165–204.

[CD13] Arthur Choi and Adnan Darwiche. “Dynamic minimization of sentential decision diagrams”. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 27. 1. 2013, pp. 187–194.

[Coo71] Stephen A Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing.* 1971, pp. 151–158.

[Dar11] Adnan Darwiche. “SDD: A new canonical representation of propositional knowledge bases”. In: *Twenty-Second International Joint Conference on Artificial Intelligence.* 2011.

[Ede14] Stefan Edelkamp. “Planning with pattern databases”. In: *Sixth European Conference on Planning.* 2014.

[EH01] Stefan Edelkamp and Malte Helmert. “MIPS: The model-checking integrated planning system”. In: *AI magazine* Volume 22.3 (2001), pp. 67–67.

[EKT15] Stefan Edelkamp, Peter Kissmann, and Alvaro Torralba. “BDDs strike back (in AI planning)”. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 29. 1. 2015.

[Epp98] David Eppstein. “Finding the k shortest paths”. In: *SIAM Journal on computing* Volume 28.2 (1998), pp. 652–673.

[FBAH20] David Fernández-Amorós, Sergio Bra, Ernesto Aranda-Escolástico, and Ruben Heradio. “Using extended logical primitives for efficient BDD building”. In: *Mathematics* Volume 8.8 (2020), p. 1253.

[HN13] Steffen Hölldobler and Van-Hau Nguyen. “An efficient encoding of the at-most-one constraint”. In: *Technical Report 1304. Technische Universitäat Dresden* (2013).

[Kar72] Richard M Karp. “Reducibility among combinatorial problems”. In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20--22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department.* Springer. 1972, pp. 85–103.

[KMS96] Henry Kautz, David McAllester, and Bart Selman. “Encoding plans in propositional logic”. In: *KR* Volume 96 (1996), pp. 374–384.

[KS20] Michael Katz and Shirin Sohrabi. “Reshaping diverse planning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 34. 06. 2020, pp. 9892–9899.

[KSU20] Michael Katz, Shirin Sohrabi, and Octavian Udrea. “Top-quality planning: Finding practically useful sets of best plans”. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 34. 06. 2020, pp. 9900–9907.

[KSUW18] Michael Katz, Shirin Sohrabi, Octavian Udrea, and Dominik Winterer. “A novel iterative approach to top-k planning”. In: *Twenty-Eighth International Conference on Automated Planning and Scheduling.* 2018.

[RK08] Michael Rice and Sanjay Kulhari. “A survey of static variable ordering heuristics for efficient BDD/MDD construction”. In: *University of California, Tech. Rep* (2008), p. 130.

[RSU14] Anton Riabov, Shirin Sohrabi, and Octavian Udrea. “New algorithms for the top-k planning problem”. In: *Proceedings of the scheduling and planning applications workshop (spark) at the 24th international conference on automated planning and scheduling (icaps).* 2014, pp. 10–16.

[Rud93] Richard Rudell. “Dynamic variable ordering for ordered binary decision diagrams”. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD).* IEEE. 1993, pp. 42–47.

[Shl+15] Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. “Heuristics and symmetries in classical planning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 29. 1. 2015.

[SMN20a] David Speck, Robert Mattmüller, and Bernhard Nebel. “Symbolic Top-k Planning”. In: *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020).* Edited by Vincent Conitzer and Fei Sha. AAAI Press, 2020, pp. 9967–9974.

[SMN20b] David Speck, Robert Mattmüller, and Bernhard Nebel. “Symbolic top-k planning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 06. 2020, pp. 9967–9974.

[Tor+14] Alvaro Torralba, Vidal Alcázar, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp. “SymBA*: A symbolic bidirectional A* planner”. In: *International Planning Competition*. 2014, pp. 105–108.

[TTTX18] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. “Action schema networks: Generalised policies with deep learning”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.