Compact Representation of Labeled Trees

Máté Simon

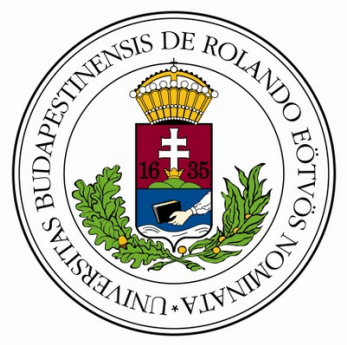
Thesis

Applied Mathematics MSc

Supervisor:

Péter Madarasi

Eötvös Loránd University,
Department of Operations Research



Eötvös Loránd University

Faculty of Science

Budapest, 2023

Acknowledgements

I am deeply grateful to my supervisor, Péter Madarasi, who has been guiding me on my path in the world of operations research since my BSc studies. I would like to express my heartfelt thanks for the tremendous amount of time, consultations, and ideas he dedicated to helping me create my thesis. Furthermore, I would like to express my gratitude to my family for their unconditional support.

Contents

1 Introduction

Mass customization aims to create products tailored to individual requirements while upholding the economic efficiency typically associated with large-scale manufacturing. Mass customization has recently received great attention with the introduction of personalization into large-scale manufacturing. One way to conceptualize mass customization is by considering a foundational product with a fixed set of properties which together define a unique, customized product. Each property is thought of as a set of options, exactly one of which is to be selected for each property. However, the challenge arises from our inability or unwillingness to produce every possible combination. Take car manufacturing as an example. Suppose the retailer sells our dream car in two different type; convertible and cabriolet. The cabriolet one is not sold with air conditioning, while in the convertible one we have the freedom to buy it with air conditioning with is. So, it is easy to imagine that for larger products such as cars or motorcycles, where there are many more sets of properties, there are even more such restrictions to consider. The problem addressed in this thesis revolves around finding the most concise way to store all of the allowed product variants.

The structure of this thesis is as follows. In the rest of this section, we present the motivation of our work and define the central problem that is investigated in this thesis.

Throughout this study, several representations are considered to represent the allowed product variants. Section 2 presents the most natural representation that arises intuitively — which is called the *DAG representation*, where exactly the paths of length n correspond to valid combinations. Then we turn our attention to investigating a way more widespread data structure, the so-called *binary decision diagrams*. We present a variety of decision diagram types, demonstrate how they can be used to represent our problem, and compare them to the DAG representation. After this, the most important issue will be investigated regarding all of the representations: based on what variable (property set) order should we construct these representations? We investigate the complexity of this problem, and present several exact and heuristic algorithms for finding good orderings.

Section 3 investigates fundamentally different representations. While in Section 2, representations exploiting subtree repeats are presented, in Section 3 representations that exploit the internal structure are presented. First, we investigate walk

representations, where exactly the root n-walks — the walks of length n rooted in a given node — correspond to the valid combinations as opposed to the n-paths in the DAG representation. Then we proceed to discuss a compression scheme that uses so-called top trees.

Section 4.2 presents our main contribution, which is a data structure for the compression problem. This section presents an algorithm for swapping two consecutive levels in our data structure, which algorithm is the basis of every reordering algorithm. It also contains a brief presentation of the state-of-the-art package called CUDD used for the manipulation of decision diagrams, and an extensive comparison between our results and the results obtained by CUDD. Lastly, we show how our program could be improved in the future.

1.1 Motivation

This section describes the everyday problem that served as the inspiration for the research in this thesis. Mass customization is the combination of product customization and mass production. Back in the day, when Henry Ford originally started mass producing his renowned Ford Model-T, it was only available in color black [30]. If customers wanted the car in red, then the factory should have constructed another assembly line for those cars, provided that the car makers were able to manufacture cars in that color as well. In contrast, nowadays customization and mass production is ubiquitous. Today, car factories produce a lot of different variations of a base car model, most of the time according to the customer's wishes. Therefore, to meet the customer's expectations, assembly lines should have great flexibility regarding the number of different variations they can produce of a base model. In order to produce a car, one must specify a lot of properties along which these cars can differ and make the assembly line handle these cases. For example, properties along which we can distinguish the cars are:

- size of the engine (how many horsepower the engine should have),
- the size of the wheels,
- whether it is the cabriolet variant or not,
- the color of the car or

- whether it should have air conditioning or not.

In addition, there are restrictions on the possible customizations, for example a cabriolet car with air conditioning in it cannot be produced. We only investigate products that are characterized by a fixed number of properties. If only such products are considered, then choosing one property from each property set characterizes the product. Since we are only interested in these type of products, the possible combinations, for which the corresponding product might be produced, can be stored in so-called variant tables. In a variant table the columns correspond to the properties such as the color, the size of the engine, etc., while the rows correspond to the feasible combinations.

Regarding this real-life problem, many interesting questions arise. For example, how can one store these tables in such a way that the representing data structure has the following properties:

- It stores the feasible combinations that are listed in a given variant table as compactly as possible.
- This data structure can be used to decide whether a given combination is feasible or not, and it supports queries — such as, how many different product can be produced, if we already fixed k properties — quickly.

Now that we have seen the motivational background, a formal problem statement follows.

Let n be the number of properties. Let $A_1, A_2, \ldots, A_n$ be the property sets, i.e., where each set corresponds to a property. Then, we get a *combination* by selecting an element from each set. These combinations can be either valid or not which are determined by some kind of rules. We suppose, that these rules are already given for us, in the form of a table, listing all of the valid combinations. In the example above, it would be determined if it is valid whether or not the combination can be produced as a car. So these valid combinations form a subset of the Cartesian product of the sets $A_1, A_2, \ldots, A_n$. Therefore, our objective is to encode these valid combinations as succinctly as possible.

In order to ease the subsequent references, let us formulate the above defined concepts in the form of a problem:

Problem 1.1. *Let $A_1, A_2, \ldots, A_n$ be sets of few elements. Consider a subset $C \subseteq A_1 \times A_2 \times \ldots \times A_n$. Encode the combinations listed in C in a data structure as efficiently as possible so that the devised data structure supports certain operations efficiently amongst the combinations, such as modifying or searching.*

2 Representations Using Subtree Repeats

This section deals with the most obvious representation that intuitively comes to mind when one tries to represent the above defined problem. This representation is the *DAG representation*, where exactly the n-paths in a directed acyclic graph correspond to the valid combinations. After getting familiar with the concept of the DAG representation, we turn our attention toward investigating a ubiquitous data structure, the so-called binary decision diagrams, which can be used as a representation for our problem as well. We present a variety of decision diagram types, demonstrate how they can be used to describe our problem, and highlight their similarities and differences compared to the DAG representation. After this, the most important problem is investigated regarding these representations: based on what variable (property) order should we construct these representations? As we will see, the size of the representations greatly depends on this variable order, and since our goal is to encode these representations as compactly as possible, finding a good order is essential for us. We investigate the hardness of this problem and present several exact and heuristic algorithms for finding good orderings.

2.1 The DAG Representation

Perhaps the most natural way to represent Problem 1.1 is via a decision tree. We might get a decision tree T that encodes the valid combinations in the following way.

Let C^k be the set of the valid combinations over the property sets $A_1, \ldots, A_k$ (so we forget about the $A_{k+1}, \ldots, A_n$ property sets). We fix in advance which level of our decision tree T corresponds to which property set. For simplicity, suppose that the property set corresponding to level i is A_i. Furthermore, suppose that we have already produced the representing decision tree up to the i^{th} level. Let us denote the vertices of the decision tree on level j with V_j. Then, the following holds: all the vertices in V_i represent sub combinations of C, where exactly the first $(i-1)$

properties have been fixed and every such sub combination is represented; in other words, every vertex corresponds to exactly one element of C^i, and vice versa.

We now construct the representing decision tree up to the $(i+1)^{\text{th}}$ level. Add $|A_i|$ new edges to all of the vertices in V_i. Then iterate through V_i: let $v_j \in V_i$ be the actual node. This node v_j corresponds to a sub combination $c^i_j \in C^i$. Then for every value $a_{il} \in A_i$, check whether $c^i_j + a_{il} \in C^{k+1}$ holds. If not, then delete the edge corresponding to value a_{ij} incident to v_j.

By performing this method n times, starting from the empty combination set C_0, we obtain a tree, which is, of course, a good representation of the valid combinations. Figure 1 shows an example decision tree that represents the variant table shown in Table 1.

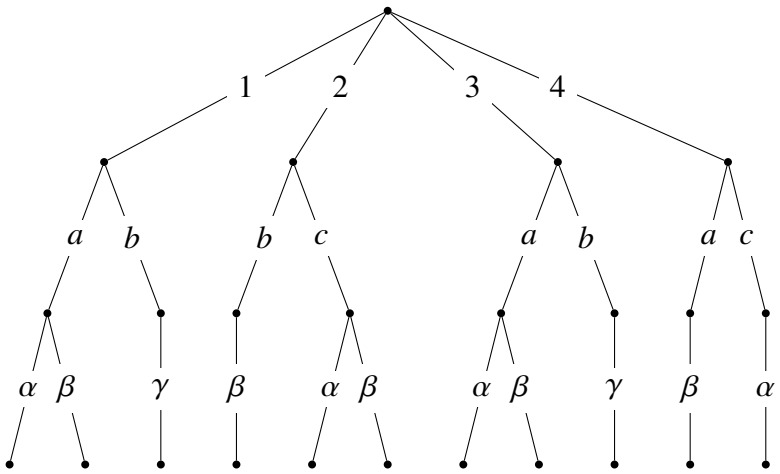


Figure 1: The decision tree representing the variant table shown in Table 1 in the (x_1, x_2, x_3) variable order.

Additionally, it is very important to note that the size of the representing decision tree T heavily depends on the order in which we build it up.

However, recall that our objective is to give a representation as compact as possible. Therefore the following two questions arise:

1. What ordering of the property sets should we choose, since the size depends on it so heavily?

x_1	x_2	x_3
1	a	α
1	a	β
1	b	γ
2	b	β
2	c	α
2	c	β
3	a	α
3	a	β
3	b	γ
4	a	β
4	c	α

Table 1: A variant table, where the first property set $A_1 = \{1, 2, 3, 4\}$, the second property set $A_2 = \{a, b, c\}$ and the third property set $A_3 = \{\alpha, \beta, \gamma\}$.

2. How can one compress such a labeled tree such that the compressed representation is equivalent to the original tree in the sense that it support basic navigational queries, such as returning the parent of a node v, the depth of v, the size of the subtree rooted in v etc?

We deal with the first question later. Regarding the second one, a natural idea is to construct a directed acyclic graph (DAG) from the decision tree by identifying and combining identical subtrees using a hashing algorithm. More precisely, an edge incident to a node $v \in V_1$ pointing to the subtree T', can instead point to any other subtree T' isomorphic to T, thus achieving that every subtree appears only once. This way, it is possible to represent T as a Directed Acyclic Graph (DAG). Over all possible DAGs that can represent T, the smallest one is unique and can be computed in $O(n)$ time [21]. We call this unique DAG the DAG representation for the combination set C for the given order of the property sets. The operations of this hashing algorithm that computes this unique DAG will be demonstrated through an example in Section 4.1 in more detail. In Figure 2 we present what the DAG representation looks like for the variant table shown in Table 1, and variable order (x_1, x_2, x_3). As we can see the size of the decision tree is greatly reduced, by merging the identical subgraphs.

In the algorithm described above, it was predetermined which property set is at each level in advance. In this manner, a so-called static representation was obtained. However, it is a restriction to say that each level should correspond to one and only

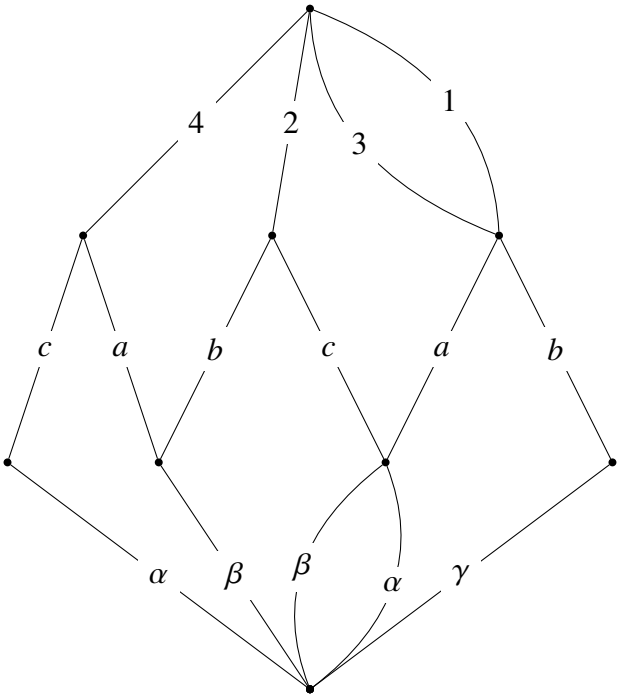


Figure 2: The DAG representation obtained from the decision tree in Figure 1 by merging identical subtrees.

one property set in the decision tree. We could have the freedom to at every node to set any variable corresponding to any property set, without the restriction that at each level every vertex should set the same variable. The only restriction is that every variable should be included exactly once in each root-sink path. The non-static representation is a generalization of the static representation, since in a non-static construction we could get back a static case, simply by choosing from the same set on each level.

2.2 Binary Decision Diagrams (BDDs)

A decision diagram is a graphical data structure that was first used to represent Boolean functions [2, 40] and has found widespread use in formal verification and circuit design [15, 31]. This section presents a variety of decision diagram types, demonstrates how they can be used to describe Problem 1.1, and highlights their similarities and differences compared to the DAG representation.

A binary decision diagram (BDD) is a type of data structure that represents a Boolean function f [39, 58]. A Boolean function can be represented as a rooted, directed

acyclic graph (rooted DAG) that consists of multiple decision nodes and two terminal nodes. The two terminal nodes are denoted by the labels 0 (FALSE) and 1 (TRUE). Each (decision) node u is labeled by a Boolean variable x_i and has two child nodes called low child and high child. The edge from node u to a low (or high) child represents an assignment of the value FALSE (or TRUE, respectively) to variable x_i. A BDD is said to be 'ordered' if different variables appear in the same order on all paths from the root. A BDD is also considered "reduced" if it does not take up unnecessary space. To be more specific, the following prerequisites must be met:

1. There cannot be a node u whose low and high children are the same.

2. There cannot be two isomorphic rooted subtrees in the representing DAG.

From an ordered BDD, we can get a reduced one by eliminating every node where **1.** is met and by merging the isomorphic subgraphs. If we do this until none of **1.** and **2.** holds, we obtain a DAG, which is our BDD representation for f.

In general, in the literature, when BDDs are mentioned, they are almost always referred to as ordered, reduced binary decision diagrams (ROBDD). The benefit of an ROBDD is that it is canonical for a specific function and variable order. Variable ordering in BDDs is just as important as the order of the property sets in the DAG representation, as we have mentioned in Section 2. Furthermore, to back up this claim, an example is shown in Section 2.2.2, and the variable ordering itself is covered in Section 2.3 in more depth. From now on, if we say BDD, we mean ROBDD.

Here as well, the root-terminal paths correspond to variable assignments. A root-(1-terminal) path in the representing DAG corresponds to a (possibly partial) variable assignment for which the represented Boolean function is true. On the other hand, the root-(0-terminal) paths correspond to the false assignments. From a path, one can get the corresponding variable assignment in the following way: When the path descends to a node's low (or high) child, the variable of that node is set to 0 (respectively 1).

Above, the possibly partial variable assignment shall be understood as the value of the other variables that are not included in the paths does not matter since the value of the variables that are already included decides the value for the corresponding Boolean function.

An example For a better understanding of these concepts, let us look at an example for the Boolean function $f = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge x_3)$. The truth table for f is the following:

x_1	x_2	x_3	Output
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 2: Truth table for $f = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge x_3)$.

The decision tree corresponding to f is the following tree:

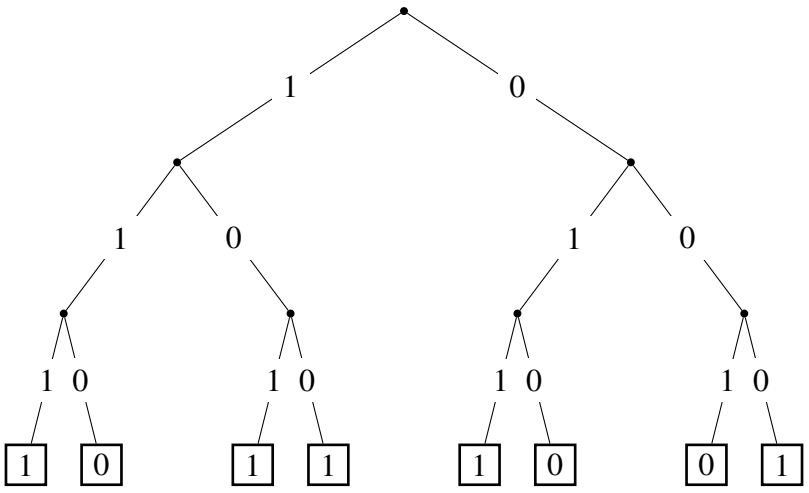


Figure 3: The decision tree representing corresponding to the Boolean function $f = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge x_3)$.

At last, we can get the BDD by executing the elimination and merging rules to this decision tree. The final BDD looks like the one shown in Figure 4 for the variable ordering (x_1, x_2, x_3). If a figure in this study shows a BDD, then we generated that figure using the CUDD C++ package [56], which we describe in more detail in Section 4.2.

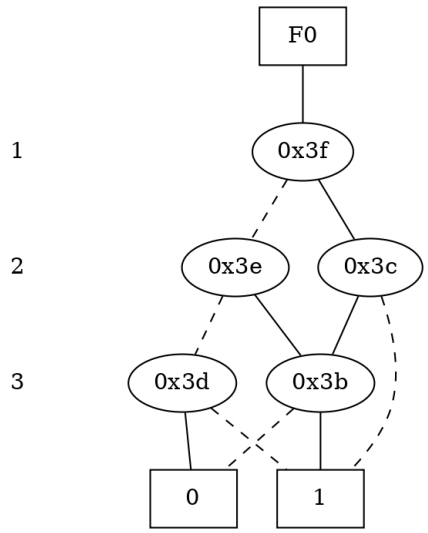


Figure 4: The BDD for the Boolean function f.

Usage of complemented arcs Using complemented edges, one might represent BDDs even more compactly. Every low edge can be complemented or not. If they are not, then they behave exactly as previously described. However, if they are complemented, then it refers to the negation of the Boolean function that corresponds to the node that the edge points to. Let us explain what this means: Suppose at a decision node v, the low edge of v points to the rooted subtree T_1. However, there is another rooted subtree, T_2, which is the complement of T_1; that is, a variable assignment in T_1 yields an output 1 of the represented Boolean function if and only if T_2 yields an output 0. Then we can complement the low edge of v, and instead of pointing to T_1, we point it to T_2. If this was the only occurrence of T_1, then that whole subgraph can be deleted. If we use complemented edges, then the root node of a BDD is not the first decision node, as we have seen in Figure 5, but another newly added so-called reference node. This reference node has outdegree 1, the only incident edge is a low one, and it points to the first decision node.

Figure 5 shows the previous example, but with the usage of complemented edges. The dashed lines correspond to normal low edges, while the dotted lines correspond to the complemented low edges.

Some properties of this complemented edges representation:

1. There is only one leaf.

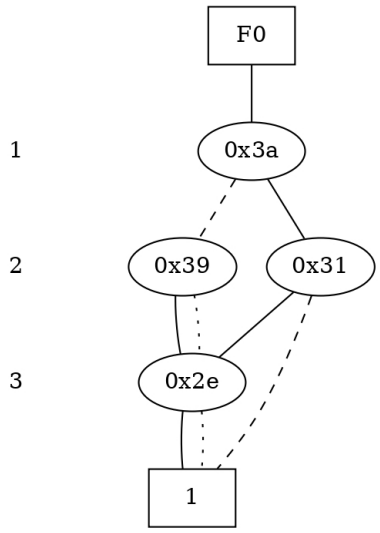


Figure 5: The BDD for $f = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge x_3)$ when using complemented edges. The dashed lines are the normal low edges, while the dotted ones are the complemented low edges.

2. Since we are not complementing high edges, a canonical representation can be obtained in this case as well.

How can we find the value of the Boolean function f in this representation for a given variable assignment? Let us take the path P corresponding to this variable assignment. Then the output — which is 1, since there is only one leaf — has to be negated as many times as there are complemented edges in this path P. So if there are an even number of complemented edges in the path P, then the value of f is 1 for this variable assignment, and if there are an odd number of complemented edges, then the value of f is 0.

Some of the advantages of using complemented edges are:

1. The size of the BDD is reduced.
2. Computing the negation of a BDD takes constant time (all we have to do is complement the edge incident to the reference node).

But there are also drawbacks to using this representation. For example, if we want to visualize our BDD, then visually it is much more difficult to read out feasible solutions looking at the graphical representation using complemented edges. Furthermore, implementing BDD manipulating algorithms becomes slightly more

complicated.

Application to our problem In Section 2.1 we have seen how we can represent Problem 1.1 using DAG representation. Now, we show how the DAG representation can be transformed into a BDD representation. This transformation will be handy especially when, in Section 4.2 we investigate what results our methods can obtain on the DAG representation compared to what state-of-the-art solvers can obtain on the corresponding BDD.

So, let us see how one can construct the BDD from our DAG representation, which represents a given variant table. The BDD is constructed with the help of the gadgets presented in Figure 6.

Figure 6 can be understood as follows: Let v be a node, and $u_1, u_2, \ldots, u_k$ be the children of v. Then the gadget H_v corresponding to v is a path of length k, with an extra edge added to each of the k vertices except the last one.

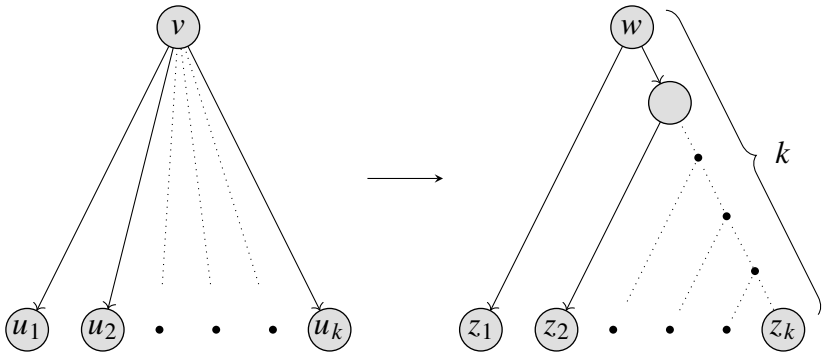


Figure 6: Gadget for the construction of the BDD from the DAG representation.

Now we can construct the BDD with the help of these gadgets. Replace every node v in our DAG representation with the corresponding gadget H_v as if every node z_i in the gadget H_v would be u_i for all $i \in \{1, 2, \ldots, k\}$.

This way we get a binary DAG, but it is not a BDD yet, because the elimination rule does not necessarily hold. After applying the elimination rule as many times as we could, we obtain the desired BDD.

Notice, that this BDD was constructed based on the exact variable order found in the DAG representation as well. In order to construct this BDD in a different variable order other means must be used. But for more details regarding this transformation we refer the reader to Section 4.2.

Furthermore, we would like to note, that usually there is way more node in this BDD representation compared to the DAG representation, since the elimination rule does not decrease the node number by nearly as much as the gadgets increased.

2.2.1 A Brief History

The basic idea behind decision diagrams was introduced by Lee [40]. In his work, he investigated so-called binary-decision programs, which are a type of computer program that represents switching circuits. Shannon demonstrated that switching circuits can be described using Boolean algebra [53]. Lee's goal was to create an alternative representation that is more conducive to the actual computation of switching circuit outputs.

Figure 7, was taken from [10], and it presents a simple switching circuit. In this circuit, the binary variables x, y, z correspond to the switches. The gate x is open if the assigned value of x is 1, otherwise closed, and x' is open. This holds similarly for all the variables. The output of the circuit is 1 for a given variable assignment if there is a continuous path from left to right. For example, $(x, y, z) = (1, 0, 1)$ leads to an output 1, while $(x, y) = (0, 0)$ leads to an output 0, regardless of the value of z.

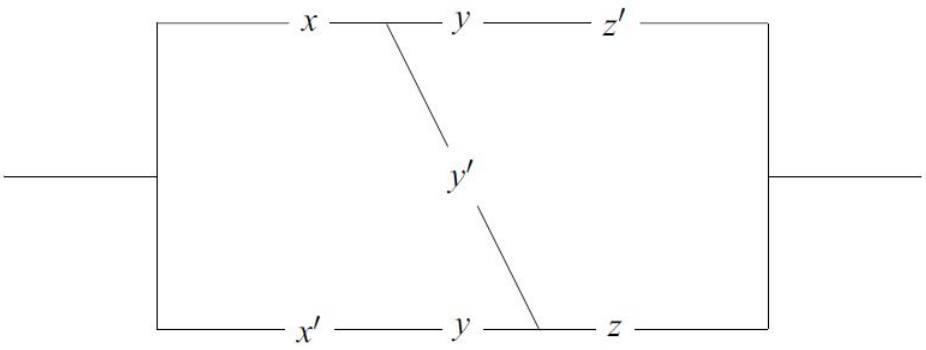


Figure 7: The example switching circuit [10].

In binary-decision programs, there is a single type of command, which Lee calls T. The command looks like the following:

$$T : x; A, B,$$

and it means that if $x = 0$ then go to the instruction at address A, otherwise to

the instruction at address B. For the switching circuit represented in Figure 7, the program looks like the following:

$$T : x; 2, 4$$
$$T : y; \theta, 3$$
$$T : z; \theta, I$$
$$T : y; 3, 5$$
$$T : z; I, \theta$$

where we used Lee's denotations: θ is corresponding to the output zero, and I to the output one. Now if we are thinking in decision diagrams these five instructions correspond conceptually to the nodes of the BDD, the first value is the case when we go to the low child, and the second case is when we go to the high child. However we do not get back the binary decision diagrams exactly, because the nodes corresponding to one variable are not required to be on the same level. Akers is credited with developing both the graphical structure known as a binary decision diagram and the term itself [2]. Akers utilized BDDs for the analysis of specific Boolean functions and as a means of test generation; that is, for finding a set of inputs that can be used to confirm that a given implementation performs correctly.

The advance that led to the widespread application of BDDs was due to Bryant [15]. He said that only that case should be investigated when the order of the variables is fixed. This way, he introduced the concept of ordered binary decisions. And after this, it was natural to introduce the reduced (ROBDD) variant (in line with the definitions in the previous section) to these OBDDs as well. These ideas led to the fundamental result that ROBDDs provide a canonical representation of Boolean functions. That is, for any given variable ordering, every Boolean function has a unique representation as an ROBDD. This enables one to determine whether a logic circuit implements a desired Boolean function, for example, by building a ROBDD for both and determining whether they are the same.

Another benefit of using ROBDDs, i.e., a graphical representation over an abstract program, is that operations on Boolean functions, such as conjunction or disjunction, can be performed directly on the representational diagrams using properly adjusted operations. The time complexity of an operation is obviously bounded by the product

of the sizes of the BDDs. Unfortunately, even when reduced, the size of BDDs for some widely used circuits can be exponentially large. For instance, the ROBDD grows exponentially for a multiplier circuit but linearly for an adder circuit.

Furthermore, the variable ordering can have a significant impact on the reduced BDD's size. Bollig and Wegener showed that it is NP-complete to determine the ordering that produces the minimal BDD [14]. Finding compact BDDs for real-world applications may consequently require the usage of ordering heuristics. We go into more detail about the hardness of the problem and ordering heuristics in Section 2.3.

Bryant's idea made BDDs into the useful tools they are today. Thanks to him, several researchers started investigating these decision diagrams. For various theoretical and practical reasons, several versions of the basic BDD data structure have been proposed. In the latter Sections, 2.2.3 and 2.2.4, we present some of these variants.

Donald Knuth refers to BDDs as "one of the only really fundamental data structures that came out in the last twenty-five years" in his video presentation "Fun With Binary Decision Diagrams (BDDs)" [38], and mentions that Bryant's article from 1986 was once the most-cited publication in computer science.

2.2.2 The Effect of Variable Orderings on the Size of a BDD

For a given Boolean function f, the size of its BDD depends only on the ordering of its variables, since BDDs yield a canonical representation. Furthermore, the size is heavily dependent on it. In this section, an example is presented where, in one variable order, the size of the corresponding BDD is exponentially bigger than in another one.

So, let us take a look at the following example: On $2n$ variables, define the following Boolean function: $f(x_1, x_2, \ldots, x_{2n}) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \ldots \vee (x_{2n-1} \wedge x_{2n})$.

Let π_1 be the variable ordering $(x_1, x_2, \ldots, x_{2n-1}, x_{2n})$, and π_2 be the variable ordering $(x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n})$. In the first case, there is exactly one node at every level in the corresponding BDD since the variables are arranged in a nice sequential way. Figure 8 illustrates the case $n = 4$ for π_1.

However, in the second case, using the variable ordering π_2, the size of the BDD is at least 2^n. This is due to the fact that in order to have at least one clause we can

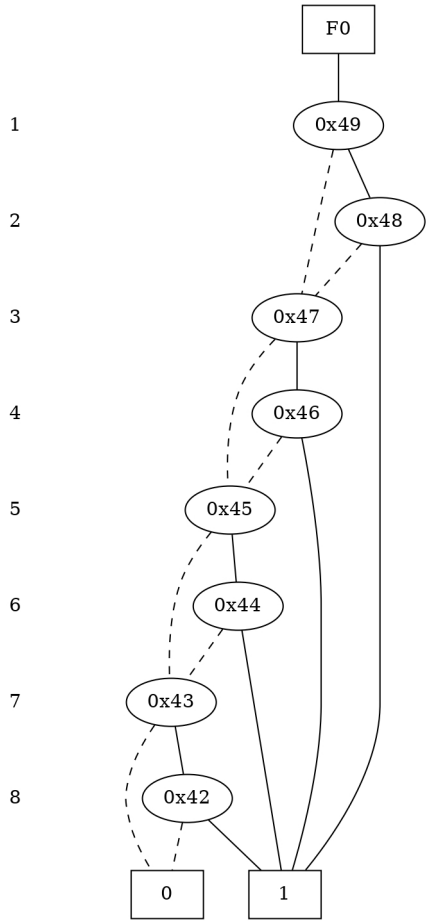


Figure 8: The BDD of f with variable ordering π_1.

evaluate, we have to get down to the $(n+1)^{\text{th}}$ level. Figure 9 illustrates the case $n = 4$ for π_2.

This proves that there are cases when we can obtain an exponentially smaller representation in one variable ordering than another.

This example motivates us to find the best variable ordering for a given BDD. Unfortunately, finding the optimal ordering is NP-hard [28]. Furthermore, for any constant $c > 1$ it is even NP-hard to compute a variable ordering resulting in an BDD with a size that is at most c times larger than an optimal one [54]. However, there exist efficient heuristics to tackle the problem [51]. Section 2.3 discusses the complexity of variable ordering in more depth.

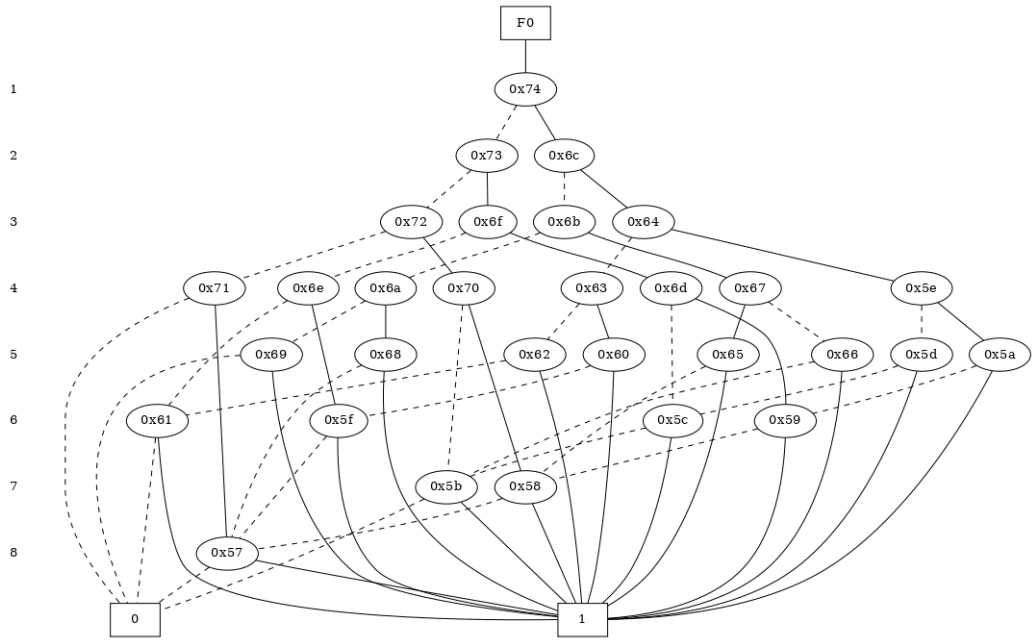


Figure 9: The BDD of f under the variable ordering π_2.

2.2.3 Multi-valued and Algebraic Decision Diagrams (MDDs and ADDs)

A number of variants of BDDs have been introduced in the literature. In the following two sections, we present some of these new variations.

Algebraic decision diagrams (ADDs) Bahar et al. [8] introduced the concept of Algebraic decision diagrams (ADDs). ADDs differ from BDDs in that the terminal nodes of an ADD may take any value from a fixed, finite set S. These ADDs also have the fundamental property of BDDs: they have a canonical representation for a particular ordering of the variables. Nonetheless, it is not clear how edge complementation could be imported to this algebraic case as well.

Bahar et al. [8] showed that matrices can be represented with ADDs and implemented sparse matrix multiplication algorithms and shortest path finding algorithms using this representation. Their devised data structure could not beat sparse matrix data structures in terms of worst-case space complexity. However, according to them "recombination of isomorphic subgraphs may give a considerable practical advantage to ADDs over other data structures", and based on their computations, merging (recombination of isomorphic subgraphs) that is present in every decision diagram indeed gave them an advantage over other data structures usually used when deal-

ing with sparse matrix multiplication and shortest path finding. Furthermore, they started investigating possible applications of ADDs to logic synthesis, verification, and testing of digital circuits and systems.

Multi-valued decision diagrams (MDDs) Now let us take a look at the Multi-valued decision diagrams (MDDs). In MDDs, we replace the binary variables with integer variables. That is, every variable x_i may take values from a predefined set of integers S_i for that x_i. MDDs are usually defined in two separate ways, depending on the actual application. One might define them analogously to BDDs (considering the ordered case): Every level corresponds to one variable. At the corresponding levels, every node has an outdegree of $|S_i|$, and it has two terminal nodes, one corresponding to the TRUE, one to the FALSE value. However, in the other kind of definition only those variable assignments are represented that correspond to the TRUE output. Thus, it does not necessarily hold that every node on the same level has the same outdegree. In this second definition, using MDDs as the representation for Problem 1.1, we obtain a very similar representation to the DAG representation presented in Section 2. These MDDs also have the elimination rule, compared to our DAG representation.

Unfortunately, MDDs did not get nearly as much attention as BDDs, so the main solvers used for manipulating and working with decision diagrams — such as CUDD, which is presented in Section 4.2.3 — have no implementation for MDDs.

Despite this, there has been research using MDDs. For example, Andre et al. proposed a novel approach to solving generic sequencing by using MDDs [20]. Or Andersen et al. [7] investigated the usage of MDDs in constraint programming. Based on experiments, their "MDD-based constraint store can substantially accelerate solution of multiple-alldiff problems". So as we can see, MDDs can also be used in a variety of problems to obtain better results than the existing approaches.

2.2.4 Zero-suppressed Decision Diagrams (ZDDs)

A zero-suppressed decision diagram (ZDD) is a special variation of a BDD that is particularly suitable for solving combinatorial problems.

Both BDDs and ZDDs can be viewed as decision trees that have been simplified using two reduction procedures that ensure the canonicity of the representation [48].

The second reduction rule, the merging of isomorphic subgraphs, applies to both BDDs and ZDDs; however, the first reduction rules are different. In a BDD, a node is eliminated if the children of this node are the same. Compared to this, in a ZDD, a node is eliminated if its high child is the FALSE terminal. Figure 10 shows that this rule is not always beneficial.

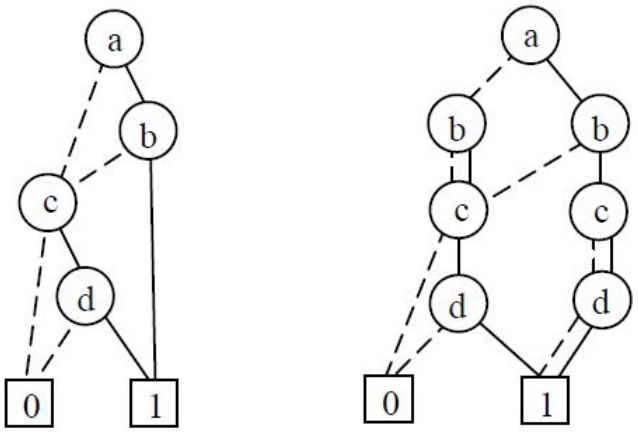


Figure 10: The BDD (left) and ZDD (right) representing the Boolean function $f = (a \wedge b) \vee (c \wedge d)$ [48] in the same variable order.

Then the question arises: why are we investigating ZDDs? ZDDs were introduced by Minato [47], and his variant of the BDDs aims to represent and manipulate sparse sets of bit vectors. The variation in the first rule, thus, aims to improve the efficiency of ZDDs when handling sparse sets.

One might construct decision diagrams from combination sets as well. A combination set is basically a family of subsets (actually, in the variant tables, we store combination sets, for which every combination is of a fixed size k). In order to represent such families, we can put a set in a one-to-one correspondence with its characteristic function, which we can get in the following way. Let us be given a set of subsets F. Then in its characteristic Boolean formula, there are as many variables as there are distinct elements in the subsets. Let the union of the elements of the subsets be S. There is a clause C_i for each subset S_i. Let C_i consist of all variables and negate exactly those variables that do not appear in S_i.

For example, if the combination set $F = \{\{a, b\}, \{a, c\}, \{c\}\}$ is given, then the corresponding Boolean formula is $f = (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge \neg b \wedge c)$. Now if we consider Figure 11, we can see both the ZDD and the BDD representing F, and that in this case the ZDD provides a better representation than the BDD.

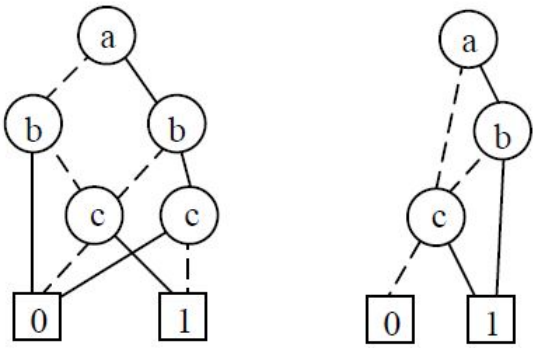


Figure 11: The BDD (left) and ZDD (right) representation of the combination set $F = \{\{a, b\}, \{a, c\}, \{c\}\}$ in the same variable order [48].

Application To see that ZDDs are indeed a useful tool in practice, let us investigate an application from [39]. We can use ZDDs to represent simple paths in an undirected graph. Let us be given the graph shown in Figure 12, which is a so-called 3×3 grid graph, and we would like to list all the paths going from the corner labeled with 1 to the corner labeled with 9.

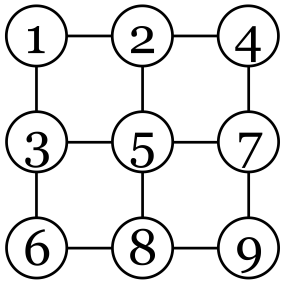


Figure 12: The 3x3 grid graph [39].

Figure 13 shows all the 12 possible paths. But how can we get these paths? A possible approach is to use ZDDs.

These paths can be represented by the ZDD shown in Figure 14 in the following way: A node labeled with ij represents the decision about whether our path includes the edge ij. The low edge means no, while the high edge means yes. So for example,

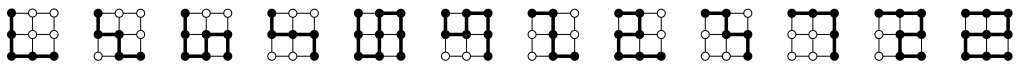


Figure 13: All 12 possible paths going from the top left vertex to the bottom right vertex [39].

traversing this ZDD only using the high edges, starting from the node labeled with 13, we get the following path out of the 12: 1 – 3 – 5 – 2 – 4 – 7 – 9.

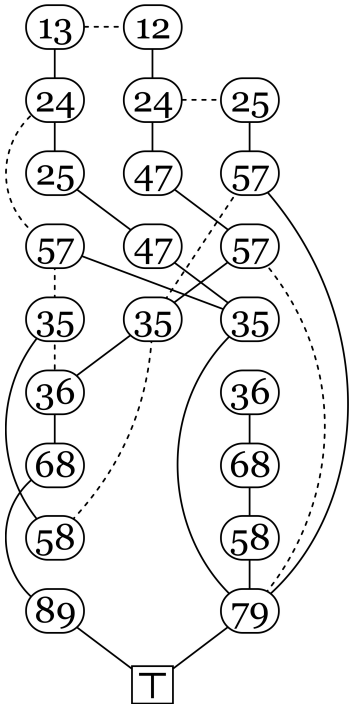


Figure 14: Illustration of the ZDD representing the possible paths in our example [39].

Although the ZDD shown in Figure 14 may not seem that useful, since only 12 paths are needed to be represented here, the advantages of a ZDD representation becomes obvious as the grid gets larger. For example, for an eight by eight grid, the number of simple paths from corner to corner proves to be 789,360,053,252 [39]. In contrast, we can define a ZDD for that problem as well, similar to the one shown in Figure 14, and after constructing and reordering it, it turns out [39], that the ZDD has only 33,580 nodes! So instead of listing all of the 789,360,053,252 paths, we can have a concise representation of these paths in form of a ZDD.

2.3 Variable Ordering

As we have seen in Section 2.2, for a given Boolean function f, the size of its corresponding BDD depends only on the ordering of its variables. Furthermore, as shown using Figures 8 and 9, we have also seen that the size can depend on it greatly, even exponentially. So a natural problem is whether or not we can find an optimal

ordering of the variables in polynomial-time, that is, the ordering for which the corresponding BDD has a minimal size (the MinBDD problem). In Section 2.3.1, we present that it is not possible unless P=NP, amongst other hardness results. Since the problem is hard, one might wonder, what can we say about the approximability of the problem? Section 2.3.1 also provides an insight into this issue.

Though the problem turns out to be NP-hard, we can do better than trying out all possible permutations. Section 2.3.2 presents an exact algorithm that solves the MinBDD problem exactly with an exponential running time — instead of a factorial running time.

Based on the results that are presented in Sections 2.3.1 and 2.3.2, the usage of heuristic algorithms is verified, also from a theoretical viewpoint. Thus, Section 2.3.3presents heuristic algorithms — most of which we ourselves also implemented for our data structure, which is presented in Section 4.2.

2.3.1 Hardness Results

In this section, we investigate the problem of finding the best variable order for a given Boolean function f from a hardness and approximability viewpoint.

NP-completeness First, let us investigate the hardness of the problem.

Instance: A BDD representing the Boolean function f under the variable ordering π_0.

Problem (MinBDD): Determine a variable ordering π, that minimizes the size of the BDD representation of f under the variable ordering π.

We would like to remind the reader that in this work, BDDs mean ordered binary decision diagrams, unless stated otherwise. Let $\mathrm{BDD}(f, \pi)$ be the BDD for the Boolean function f, corresponding to the variable ordering π. The size of a BDD is the number of nodes the BDD has, and it is denoted with $|\mathrm{BDD}(f, \pi)|$.

In order to discuss hardness results, a decision variant of the problem has to be defined. The decision variant of the problem, let us call it $s-\mathrm{BDD}$, is therefore the following:

Input: A Boolean function f and an integer s.

Output: Yes, if there exists a variable ordering π, such that $|BDD(f, \pi)| \leq s$, no otherwise.

Bollig and Wegener [14] proved that the s – BDD problem is NP-complete. It can be decided in polynomial-time whether two given BDDs represent the same Boolean function or not [25], thus the problem is in NP. Now, to prove that the problem is hard, they give a reduction from the *optimal linear arrangement problem*. In the optimal linear arrangement problem, we are given a graph $G = (V, E)$, and a bound b. Let us denote the nodes with the $\{1, 2, \ldots, n\}$ numbers. Furthermore, define a cost c_π for every permutation π on the nodes as follows:

$$c_\pi := \sum_{uv \in E} |\pi(u) - \pi(v)|.$$

Then we want to decide whether there is a permutation π' of the nodes such that the corresponding cost function $c_{\pi'} \leq b$. The output for the given instance is yes if there is such a permutation, and no otherwise.

Approximation The NP-completeness result we have seen so far do not provide any information regarding the degree of difficulty involved in approximating the MinBDD problem. For the MinBDD problem, we say that an algorithm is an approximation algorithm with an approximation ratio r if, for all instances, it gives a variable ordering π, such that $|BDD(f, \pi)| < r|BDD(f, \pi^*)|$, where π^* is the optimal ordering.

Regarding approximation algorithms, Sieling [54] proved a very strong result in a remarkable 30-plus page proof. He proved that, for any $c > 1$ there is no polynomial-time approximation algorithm with an approximation ratio r for the MinBDD problem unless P=NP.

So far, the results presented suggest that we cannot hope to give an approximation algorithm with a constant approximation ratio. However, to the best of our knowledge, there is still an interesting family of open questions in this field. More precisely, approximation algorithms with a non-constant approximation ratio. We think that even constructing a polynomial-time approximation algorithm with a approximation ratio of $n^{1-\epsilon}$, for any ϵ is not possible unless P=NP. For example, if this were true, then there could not be an approximation algorithm with a approximation ratio of $\sqrt{n}$. Similar results are known for the following two problems:

1. For every real number $\epsilon > 0$, there can be no polynomial-time algorithm that approximates the maximum clique to within a factor better than $O(n^{1-\epsilon})$, unless $P = NP$ [60].
2. For every real number $\epsilon > 0$, there can be no polynomial-time algorithm that approximates the chromatic number to within a factor better than $O(n^{1-\epsilon})$, unless $P = NP$ [60].

For example, a possible way to prove this conjecture would be by giving a reduction from one of the two problems we have just mentioned.

2.3.2 Exact Algorithms

To identify the best variable order, the most straightforward method is to try out all possible orders. For n variables, there are $n!$ different orderings. Building up a BDD from the ground up can take exponentially long in the worst case. Thus determining the optimum in this case takes $O(n!2^n)$ time. So the question arises: is there an algorithm that finds the exact optimum in exponential time? The answer is yes, and the algorithm is based on the fact — which is proved later in this section — that if two permutations of the variables share a common suffix, then their corresponding BDD's lower part are identical too (the part that corresponds to the suffix) [57]. In this section, we present an algorithm that solves the MinBDD problem optimally in $O(n3^n)$ time, thus getting rid of the factorial factor. Friedman et al. [26] originally presented an $O(n^2 3^n)$ time algorithm, but by a result of Sieling and Wegener [55], the running time can be reduced to $O(n3^n)$ time.

Let us start by introducing some notations. Let $N = \{1, 2, \ldots, n\}$ be the set of the indices of the variables. If B is a subset of N, then let $\prod(B)$ be the set of the variable orderings in which the last $|B|$ members are exactly the members of B, formally

$$\prod(B) = \{\pi | \pi(n-i) \in B, i = 0, 1, \ldots, |B| - 1\}.$$

For a given index set B we refer to the last $|B|$ variables of $\prod(B)$ as the bottom variables and to the remaining variables as the top variables. Besides, for an order π and variable x_i, let $N_i(f, \pi)$ denote the number of nodes corresponding to the decision variable x_i in the BDD ordered according to π. The subsequent lemma [26] states that when a horizontal cut is made in a BDD representing the function f,

separating the top variables from the bottom variables, the ordering of the top variables has no impact on the part of the BDD below the cut if the order of the bottom variables are fixed.

Lemma 2.1. *Let $B \subseteq N, k = n - |B| + 1$ and x_i be a variable such that $i \in B$. Then there is a constant c such that for each $\pi \in \prod(B)$ satisfying $\pi(k) = i$ we have $N_i(f, \pi) = c$.*

The original proof can be seen in [57], but we give a much simpler proof.

Proof. Let $\pi_1, \pi_2 \in \prod(B)$ be two permutations of the variables. Since they are both in $\prod(B)$, they share a suffix of length $|B|$. If we take the corresponding decision trees and apply the two reduction rules (merging and elimination) bottom up, then since the bottom $|B|$ levels are the same, the rules are applied the same way in both cases. Thus, the lower part — corresponding to the last $|B|$ variables — of the BDDs must be identical. □

Lemma 2.1 is quite useful in a variety of areas related to the investigation of the sizes of the BDDs. Using this lemma, we now present a dynamic programming algorithm that solves the MinBDD problem in $O(n^2 3^n)$ time [26, 57].

The algorithm calculates the best ordering, regarding the bottom $|B|$ variable, for every $B \subseteq \{1, 2, \ldots, n\}$. In a usual dynamic programming fashion, if we know the optimal size of the lower part of our BDD and the corresponding variable order for all of the subsets of size $t - 1$, then we can derive the optimal size of the lower part of the BDD for all subsets of size t in the following way.

Suppose that for all subsets B' of size $t - 1$, we know the optimal size and the corresponding ordering of our BDD's lower part, where the variables of B' are the bottom variables of our BDD, and let us be given a subset B of size t. For every variable x_i whose index i is in B, we would like to determine the optimal size of the bottom part of the BDD, such that x_i is on top of the bottom variables. This number is the number of nodes corresponding to the variable x_i, plus the optimal size of the lower part of the BDD, where the bottom $|B| - 1$ variables are the ones whose indices are in $B \setminus \{i\}$, but luckily we already know the optimal bottom size for all index sets of size $|B| - 1$. According to Lemma 2.1, the number of nodes corresponding to the variable x_i does not depend on the ordering of the other bottom variables, so it is indeed sufficient to save the size of the best order for every $B \setminus \{i\}$,

and the corresponding order, since by knowing them we can simply determine the optimal size for each x_i on top. To determine the number of nodes corresponding to the variable x_i simply construct any BDD where the variables whose indices are in B are the bottom variables and x_i is at the top of the others, and read out the number of nodes corresponding to x_i. After computing the optimal bottom size for each x_i on top, we can obtain the optimal size for B by saving the best one with the corresponding order.

The next step of the algorithm involves gradually raising the value of t. It is known that the size of the lower part of the BDD for $t = 0$ is 1 (the constant node). The algorithm continues upwards until it has determined the overall order that produces the optimal size of the BDD. As we have seen, Lemma 2.1 provides that the number of nodes of the variable x_i does not depend on the ordering of the variables of $B \setminus \{i\}$, but it says nothing about how to compute it. As shown above, one possibility is to compute the BDD from the ground up for any order that is optimal for $B \setminus \{i\}$ and that has x_i on top of the bottom variables. Then we can simply determine the number of nodes corresponding to x_i (which are at level $n - |B| + 1$) by inspecting the BDD. However, building up the BDD every time would be wasteful. Instead, we can store the actual BDD and obtain all the desired orders by permuting the variables in that BDD. Section 4.2.2 presents how the swap of two adjacent levels can be done in our data structure, but it is very easy to modify the procedure for BDDs as well. Since any permutation can be obtained by a series of adjacent swaps, it suffices to show how these swaps are done.

What remains is how we should schedule the solving of the subproblems. The main point we have to notice is that in this algorithm, it always suffices to swap two variables at a time — which can always be achieved by at most n swaps of adjacent levels — since from any subset of size t, we can list all of the other subsets of size t such that every subsequent set differs in exactly one element. We do not get into more details in this work. For more details regarding these swaps, the reader is referred to [26]. The running time described in this form of this algorithm is $O(n^2 3^n)$, however, we would like to mention that using the two-phase bucket sort technique introduced by Sieling and Wegener [55] the running time can be reduced to $O(n3^n)$. To the best of our knowledge, this is the best exact algorithm to this day.

Note that the algorithm presented in this section, can be sped up in practice, sometimes dramatically. For example we might know of some variables that they are

symmetric. In this case, the swap of these variables can be omitted, in order to save running time, since this swap would not change the size of our BDD [34].

2.3.3 Ordering Heuristics

In practice, the nature of the problem usually offers a natural order of the variables [39]. However, there are situations when no obvious order exists, leaving us to only hope we can calculate some better orders with the aid of the computer. Moreover, even if we do know a good approach to start a computation, the ordering of the variables that works best in the beginning may become unacceptable in later phases. So, if we don not insist on a strict a priori order, we could become way more flexible in the handling of our BDDs. Instead, whenever a our current BDD becomes complicated, we might attempt to use ordering heuristics to decrease the size of our BDD.

For instance, we might repeatedly swap x_{j-1} and x_j in the sequence, for$1 \leq j \leq n$, redoing the swap if it increases the size of the BDD but keeping it otherwise [39]. Although simple to adopt, this strategy offers too little improvements in the size of the BDD. Rudell [52] proposed a new, superior reordering method. His strategy, which he calls "sifting," has been very effective and is one of the most widely used dynamic reordering algorithms to this day. The basic idea is to start with one variable of the ordering, say let it be at level i, and try swapping it up or down to all other levels, that is removing x_i from the ordering and then inserting it again at every other possible level, picking the level that minimizes the size of the BDD. The algorithm can be done by doing only swaps between adjacent levels.

Sifting Algorithm This algorithm puts every variable x_i to its optimal position with respect to the current ordering of the variables $\{x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\}$. It works by repeatedly swapping two adjacent levels. First swapping it to the bottom of the BDD, then swapping it the top, keeping the best one. Section 4.2.2 describes how this swap procedure of adjacent levels can be done. Let G be the current BDD during the run, and S be the size of G. Then the pseudocode for the algorithm:

Algorithm 1 Sifting algorithm

1: **for** $i \leftarrow 1$ to n **do**
2: $j \leftarrow$ the index of variable x_i in the current order
3: $s \leftarrow S$ ▷ Initialization
4: $G_0 \leftarrow G$
5: $G_B \leftarrow G_0$
6: **while** $j > 1$ **do**
7: Swap $x_{j-1} \leftrightarrow x_j$, and $j \leftarrow j - 1$ ▷ Upward swaps
8: **if** $S < s$ **then**
9: $s \leftarrow S$ and $G_B \leftarrow G$
10: **end if**
11: **end while**
12: $G \leftarrow G_0$
13: **while** $j < n$ **do**
14: Swap $x_{j+1} \leftrightarrow x_j$, and $j \leftarrow j + 1$. ▷ Downward swaps
15: **if** $S < s$ **then**
16: $s \leftarrow S$ and $G_B \leftarrow G$
17: **end if**
18: **end while**
19: $G \leftarrow G_B$.
20: **end for**

Let us investigate this algorithm in more detail. In the second row, we cannot say $j \leftarrow i$ after the first iteration since it might not be true that variable x_i is at the i^{th} place. So in order to run this algorithm, one must maintain an auxiliary array, which tells us exactly which variable is at which position at any given point.

This algorithm is not the one Rudell originally described. The original algorithm only operated with swaps. We simplified it by storing two additional BDDs, G_B and G_0. Although this version greatly reduces running time, space complexity is slightly increased.

Algorithm 1 can be further improved regarding running time on several fronts without seriously influencing the outcome. For example, if S gets too big, say greater than 1.2, 1.1 or even 1.05 times the size of G_0 at any point in any iteration, we can terminate the corresponding — upwards or downwards — swapping procedure. Additional swaps in the same direction are not expected to reduce our BDD size under these circumstances [39], which we also tested.

The sifting algorithm can be rerun multiple times until there are no changes, which means a local optimum has been found. This local optimum is 2-optimal, swapping

any two variable cannot result in a smaller BDD, by the definition of our algorithm. According to Knuth [39], the additional gain the multiple iterations would provide is usually not worth the extra effort. We will see in Section 4.2, that, in practice, the improvement in the first iteration is way more significant than in the second and onward.

Simulated Annealing Simulated annealing is a probabilistic, heuristic, combinatorial optimization algorithm that can be used to solve a variety of combinatorial problems [45, 59]. Next, we briefly present the basic idea on which this metaheuristic is based and then describe a simulated annealing algorithm for the MinBDD problem.

A simulated annealing algorithm tries to find a solution close to the global optimum for a given function f. Let us suppose that we have a system, where we can move between so-called states of the system by changing it locally. Furthermore, every state can be evaluated by a cost function c, and in this framework we look for the cheapest state. Now that we have defined the framework, the simulated annealing algorithm works in the following way: First of all, we give our system a quite high temperature T, then choose a random state of our system S_0, and evaluate f at this random state. During the algorithm, the temperature T gradually decreases. Second, we choose another random state S_1, evaluate it, and accept this new state with a probability $P_{T,\Delta}$, where this acceptance probability is determined by the temperature of our system and the difference $\Delta := c(S_2) - c(S_1)$ of the cost of the two states S_0 and S_1. Then, we typically repeat this procedure until our system has substantially cooled down. The acceptance probability must fulfill the following aspects. The probability $P_{T,\Delta}$ must be greater than zero, even when Δ is positive. This feature is for the sake of preventing the method from getting stuck at local minima. When the temperature tends to zero, so should $P_{T,\Delta}$, and when the temperature tends to some positive number, then $P_{T,\Delta}$ should too tend to some positive number. Furthermore, as T gets smaller, the system should more likely prioritize "downhill" (cost decreasing) moves and penalize "uphill" moves. The algorithm gives back the greedy algorithm when $T = 0$, which only performs downhill transitions. There are many variants of these kinds of simulated annealing algorithms. For example, in the original definition, $P_{T,\Delta}$ was defined such that if $\Delta < 0$ then $P_{T,\Delta} = 1$. However, this requirement is not essential for the method to work.

Next, we present the pseudocode of a simulated annealing algorithm for the MinBDD problem. Throughout the algorithm, let S denote the current size of our BDD.

Algorithm 2 Simulated annealing for solving MinBDD

```
1:  noChange ← 0
2:  Temp ← k_1
3:  while noChange < k_2 do
4:      for i ← 1 to k_3 do
5:          S' ← S
6:          Swap two randomly selected levels
7:          Δ ← S − S'
8:          if P_{T,Δ} ≤random(0,1) then
9:              Swap back the two levels
10:         end if
11:     end for
12:     Temp ← g(Temp)
13:     if minSize > S then
14:         noChange ← 0
15:     else
16:         noChange ← noChange + 1
17:     end if
18: end while
```

As we can see, there are a lot of parameters in an algorithm like this, and the efficiency of the algorithm greatly depends on whether or not we can find the appropriate parameters. In this case, we have to define the starting temperature (k_1), stopping criteria (k_2), and how long each iteration should take (k_3); moreover, the probability function ($P_{T,\Delta}$) has to be defined in line with the properties described above. Needless to say, one might construct infinitely many such probability functions, but some of them are used more frequently than others. Lastly, the system needs to be cooled down over time, and that is what the function g describes — what the cooling rate should be. Implementing a simulated annealing algorithm involves the fine-tuning of these parameters. Section 4.2 shows, for a concrete, real-life example, how we set the parameters.

We have presented two algorithms that are widely used in practical applications and that we ourselves have implemented for our data structure as well (see Section 4.2). But there are many other variable ordering heuristics that have been investigated, including metaheuristics and more. Metaheuristics like genetic algorithms [1, 41] and particle swarm techniques [49] have been devised, but these are just two examples of the many. Plenty of other heuristic — not necessarily metaheuristics — could

be devised, perhaps more applicable for our compression problem. For example, an idea that could be implemented is the following: Fix a small number k, and find the best order for each consecutive k levels. Note that k shall be pretty small, since in each "window" there are $n2^n$ cases we have to try out. This idea could be refined in many ways, but we do not go into details since we were just trying to show an example.

In this section, we investigated so-called dynamic reordering algorithms, that is, reorderings that manipulate the BDD itself with a series of swaps. But there is another kind of reordering, the so-called static reordering, that attempts to establish the variable ordering a priori, that is, prior to constructing the actual decision diagram. Since static heuristics generate the final variable ordering before the decision diagram is constructed, there is no assurance that the resultant order produces a high-quality BDD. Alternatively, dynamic variable ordering is generally more effective in providing efficient orderings since it allows for the adjustment of the variable order during the actual construction of the decision diagram or after it is built. However, in practice, dynamic reordering is typically much more time-consuming than the more straightforward, static heuristics. There has been several studies regarding static variable ordering as well, for example in [3, 22, 27], and there is a survey about static variable ordering heuristics by Rice et al. [51].

2.4 Scheduling the Swaps

As we have seen in Section 2.3 every dynamic variable ordering algorithm is based on one thing: the swapping of adjacent levels. The first question that arises is how to do these swaps. Section 4.2.2 describes how to do swaps and investigate their implementation. The second question arises after noticing that the algorithms are composed of a sequence of swaps; therefore, there are many ways to schedule the swaps. Jiang et al. [35] investigated these schedules first. In this section, we present heuristics to schedule these swaps and compare their performances to one another.

Inversion between two orders

First, some basic notations. Let π_t be the desired, target order of variables in our BDD, and π_a be the actual order. If the relative order of two variables x_i and x_j is different in π_a and π_t, that is, if either $x_i \prec_{\pi_a} x_j$ and $x_j \prec_{\pi_t} x_i$ or $x_j \prec_{\pi_a} x_i$ and $x_i \prec_{\pi_t} x_j$, then we say that these variables form an inversion. We say that an

inversion is swappable if x_i and x_j are adjacent in our current order. Let $I(\pi_a, \pi_t)$ denote the total number of inversions between π_a and π_t:

$$I(\pi_a, \pi_t) = \sum_{1 \leq i,j \leq n, x_i \leq_{\pi_a} x_j} I_{i,j}(\pi_a, \pi_t),$$

where $I_{i,j}(\pi_a, \pi_t)$ is 1 if x_i and x_j form an inversion, and zero otherwise. In the literature, $I(\pi_a, \pi_t)$ is called the Kendall tau distance [37].

Notice that it is always possible to transform π_a to π_t using exactly $I(\pi_a, \pi_t)$ swaps. Suppose, that we do not use more swaps than that, so our task is to schedule these $I(\pi_a, \pi_t)$ swaps in such a way that it is the fastest or uses the least memory.

Heuristics

We begin by outlining four intuitive scheduling heuristics:

- Bring Up (BU): Choose the swappable inversion with the highest variable in π_t not yet in its final position.
- Sink Down (SD): Choose the swappable inversion with the lowest variable in π_t not yet in its final position.
- Lowest Inversion (LI): Choose the lowest swappable inversion.
- Highest Inversion (HI): Choose the highest swappable inversion.

Notice that the cost of a specific swap, which is proportional to the number of nodes corresponding to the top variable being swapped (see Section 4.2.2), is ignored by all of these heuristics. Thus, the following heuristic is convenient since it always chooses the locally optimal one:

- Lowest Cost (LC): Choose the swappable inversion where the variable on top has the fewest associated nodes.

This heuristic tries to minimize computing time, but another aspect we could try to minimize is total memory usage.

- Lowest Memory (LM): Choose the swappable inversion that results in the smallest BDD next.

To the best of our knowledge, there is no way to inexpensively and accurately predict the size of our BDD after a swap. Therefore, in order to determine which swappable inversion should be the next at a given point, we have to calculate each possible swap. At first glance, this would lead to a tremendously long running time, especially compared to the other heuristics. But notice that after a swap, only those cases have to be recalculated in which the two variables we swapped participate (at most two new ones), since a swap only changes the representation at the corresponding two levels (see Section 4.2.2).

Jiang et al. [35] investigated how these heuristics compare to one another, and now we present their computational results. What is not surprising is that in almost all cases, LM was the slowest (not by that large of a margin), but used the least memory out of the six heuristics. We are not surprised by this because that is the way this heuristic was devised. However, there are two interesting observations.

First, in most of the cases, LC was the second slowest after LM. This is very interesting since the way the heuristic LC was devised is to choose the swap that locally takes the least time. This also shows the fact, that taking locally optimal steps does not guarantee a globally optimal run in the end. In fact, in most of the runs, it was quite far from the globally optimal running time since all of the four basic heuristics resulted in a faster running time. This is how we arrive at the second interesting observation. Which is that in general, the four basic heuristics outperformed both LC and LM time-wise and were closer to LM than to LC memory-wise, though these four heuristics were not really specifically defined for our BDD reordering problem, while LC and LM were.

2.5 C-tuples

As we have seen in Section 1.1 mass customization aims to offer personalized variants of products that share a basic structure but possess distinct properties. This is achieved by assigning a value to specific product features that define their customized characteristics.

One might wonder how big companies handle these product configurations. Let us take, for example, a particular configurator, the SAP variant configurator [13]. Each configurator has its own way of modeling a product, i.e., how to represent the valid combinations of product variants. Product models often include tables as their basic

representation since they are intuitive, easy to implement, and easily understood by everyone [30]. Tables are a common modeling technique, but they do not scale with a lot of options, which limits their usefulness. To overcome this problem, the idea might arise to compress the table itself into a more concise table, such that it retains the functionality of a regular table. One such compression scheme is to compress regular tables into a table of so-called *c-tuples* [36]. A c-tuple is a row in a table whose cells might contain multiple values.

In the concept of SAP variant configurator modeling [13], the term variant table is the same as we defined it, i.e., it is used to refer to a table listing valid (or excluded) combinations of product features. Each column in the variant table represents a different aspect of the product (such as the imprint, size, or color of a T-shirt). Table cells with values like "Color = Red" indicate that the corresponding property in that column has been assigned that value.

Let us see an example of a variant table. In this variant table, Table 3 we encoded possible combinations for a T-shirt.

Imprint	Size	Color
Batman	Small (S)	Black
Batman	Medium (M)	Black
Batman	Large (L)	Black
Star Wars	Medium (M)	Black
Star Wars	Medium (M)	Blue
Star Wars	Medium (M)	Red
Star Wars	Medium (M)	White
Star Wars	Large (L)	Black
Star Wars	Large (L)	Blue
Star Wars	Large (L)	Red
Star Wars	Large (L)	White

Table 3: A variant table, representing possible product variants for a T-shirt.

In Table 3, each row equals a tuple of values, which we refer to as an r-tuple, short for relational tuple. In line with our definitions, an r-tuple is a combination. In contrast to an r-tuple, which can only store a single value for all properties, a c-tuple can store several values from the same column's property in a single table cell. The term c-tuple is short for Cartesian tuple.

Using c-tuples, a variant table may be compressed simply by reorganizing and partitioning it into unconstrained groups of combinations and then replacing each

such set by a c-tuple. Table 4 shows how Table 3 can be rewritten using just two c-tuples.

Imprint	Size	Color
Batman	S,M,L	Black
Star Wars	M,L	Black, Blue, Red, White

Table 4: A possible representation using c-tuples for the variant table shown in Table 3.

But here comes the difficulty: how should we reorganize the combinations to get the most concise representation using c-tuples when using c-tuples for compression? Clearly, the decomposition of a variant table into c-tuples is not unique; different decompositions may have different sizes. Heuristics here as well are key to finding good decompositions [36].

Another way to obtain a c-tuple representation was given by Haag [29]. He was inspired by the idea that even the c-tuple representation could be compressed. In order to compress this representation, he defined a special decision diagram, which he named Variant decomposition diagram (VDD). This type of decision diagram also has the same fundamental properties as other decision diagrams, such as the VDD itself is determined uniquely by specifying the order of the variables corresponding to the products. Furthermore, VDDs entail a decomposition into c-tuples; thus, they are particularly useful in determining a good c-tuple decomposition. By iterating over all possible paths in the VDD, the c-tuple compression stored in the VDD can be extracted. However, this decomposition into a VDD is even more powerful than decomposition into c-tuples. So summed up, Haag, instead of investigating the problem of finding the best c-tuple decomposition directly, defined a special decision diagram, which yields a more concise representation than the c-tuple itself, and investigated the problem of finding the minimal size of these VDDs. VDDs were further improved in [30].

The interesting point of Haag's results is that by starting to investigate an alternative storing method for our product variant storing problem, he ended up defining a new type of decision diagram. This also shows how, in a variety of areas, decision diagrams can be used and why they are so popular.

One might wonder how a representation using c-tuples compares to a DAG representation. First, if we are given a table representation using c-tuples, we can define a

decision tree where the paths correspond to rows of the table, that is to the c-tuples. This shows that the DAG representation is a stronger representation because in the DAG representation we then merge identical subgraphs, so if there are any, we obtain a smaller representation. However, defining c-tuples was not in order to get the most concise representation but to reduce the input's size, and that can definitely be done using c-tuples.

3 Representations Exploiting Internal Structures

In the previous section, we saw that the most natural way to represent a variant table is via a decision tree. Then, in the rest of the section, we investigated compressions of that decision tree that utilized subtree repeats. However, this is not the only way a tree can be compressed.

Previous work on tree compression can be divided into three major categories:

- Subtree repetitions,
- Tree pattern repeats,
- Succinct data structures.

Representations using subtree repetitions were presented in Section 2. This section investigates the two other major categories. First, compression schemes exploiting so-called tree pattern. Second, we briefly present succinct data structures.

3.1 Motivation of Walk Representations

In Section 2, we saw that for a variant table T or combination set C, it is possible to create a decision tree in which there is a one-to-one correspondence between the valid combinations and the root-leaf paths. By merging the identical subtrees, we obtained the so-called DAG representation of C. After having learned about this representation, we asked ourselves what results could be obtained if we made a representation where instead of the n-paths, exactly the root n-walks corresponded to the valid combinations. To the best of our knowledge no representation that is

exactly like this have been investigated in the literature. This section, therefore, aims to motivate further investigation of this "walk" representation.

Notice that the DAG representation itself is also a walk representation since n-paths are root n-walks as well. Regarding this representation, we do not define an algorithm that defines a unique walk representation for the combination set C, like we did with the DAG representation. We will just take graphs that meet the conditions above (like a DAG representation as mentioned above) and show that, if defined correctly, this representation can vastly outperform the DAG representation.

So, let us define the following quantity for a given combination set C, and a representing walk representation W:

$$M(C, W) = \frac{\text{size of the corresponding DAG representation}}{\text{size of the representing walk representation } W}.$$

Since the DAG representation is uniquely defined for C the definition is correct. Regarding this representation, we show that there exists a series of combination sets C_k and representing walk representations W_k such that $M(C_k, W_k)$ is arbitrarily large. More precisely, we give a combination set series C_k whose DAG representation's size is k times the size of W, i.e., if k goes to infinity, then we obtain a combination set series for which M can be arbitrarily large. We prove this with the help of Figure 15.

Consider Figure 15. In Figure a), we illustrate the initial graph D_k that represents our combination set C_k. From now on, we consider this graph class, D_k. The triangles represent copies of a given graph H. Let D_k consist of $1+2+4+\ldots+2^k = 2^{k+1}-1$ copies of H connected as shown in the Figure. Now let us define this graph, H. Let H be any graph that has the following properties:

1. Let it be a rooted graph with only one edge incident to the root.
2. Let it be already compressed with the hashing algorithm mentioned in Section 2.

Now that we have defined H, let us turn our attention towards showing the size of the different representations for the graph class D_k. How does the DAG representation look like? The schematic illustration of the DAG representation of D_k is illustrated in Figure b). Let us denote the representing DAG with D_k^{DAG}.

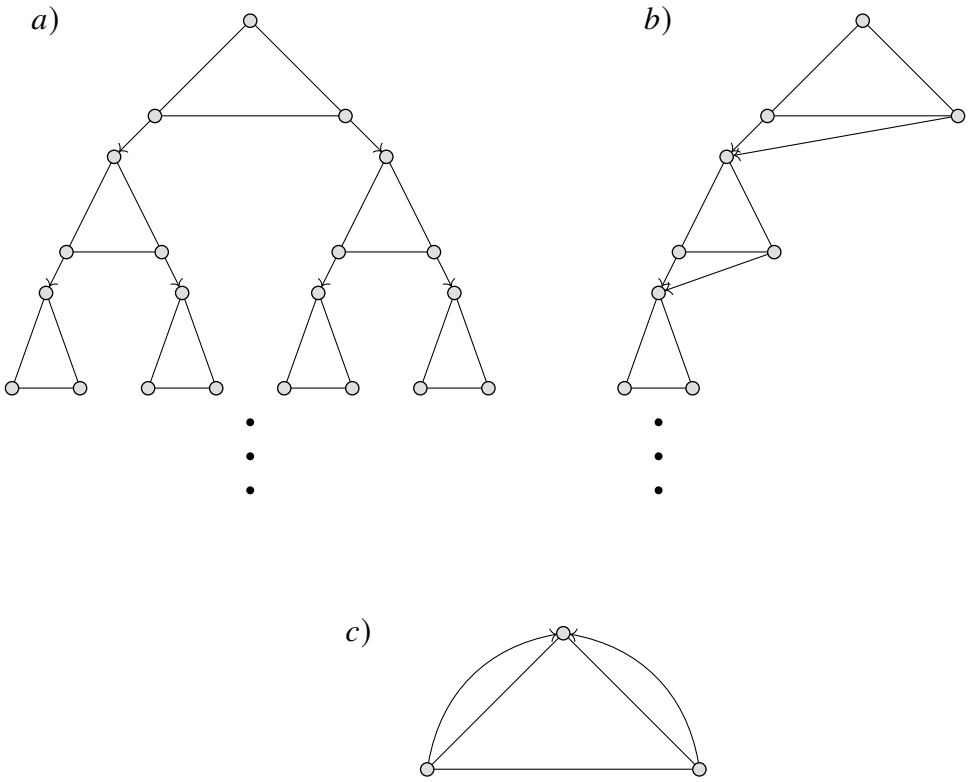


Figure 15: Figure a) shows the schematic representation of the graph D_n; figure b) the schematic representation of the corresponding DAG; and figure c) a corresponding walk representation. Each triangle represents a schematic copy of the graph H.

However, we can define a walk representation W that looks exactly like the graph shown in Figure c) for any k.

Now we can see that $|V(D_k^{DAG})| \sim |V(H)|k$, while $|V(W)| \sim |V(H)|$ for every k. It is easy to see that we can obtain a similar result for the number of edges. As $|V(D_k^{DAG})|$ is linearly proportional to k, while $|V(W)|$ is constant, we obtained a graph class D_k, such that for every C_k, which can be represented with D_k, we can also define a representing walk representation W such that $M(C_k, W)$ is arbitrarily large, thus we have proved our objective.

We began our study of this representation with a literature search. While DAG representation uses subtree repeats, this walk representation can exploit tree patterns as well. Specifically, this walk representation has not yet been addressed in the literature, but articles have been published that reinforced their compression with also exploiting the inner structure of the original graph [11, 12, 32]. They did this using so-called *top trees*. Top trees are rooted, labeled, binary trees whose vertices

represent so-called *clusters*. These clusters correspond to subtrees of the original tree. We discuss this method in more depth in Section 3.2. Having looked at these works, we conclude that the method studied in these articles is similar in principle since both exploit tree patterns, but still, investigation in more detail of the walk representation could provide some interesting results.

As we have seen, the construction only depends on the graph H, therefore one can create infinitely many graphs that have the two properties listed in 3.1, and infinitely many combination sets, which can be represented with D_k, thus W is an appropriate walk representation, there are also infinitely many cases for which $M(C,W)$ is arbitrarily large.

3.2 Compression with Top Trees

Although walk representations have not yet been addressed in the literature to the best of our knowledge, other compression schemes that exploit tree patterns have. One such compression scheme is the work of Bille et al. [12], which they call "tree compression with top trees".

In this section, we would like to briefly present the work of Bille, Gørtza, Landau, and Weimann [12]. In their work, they introduced a new tree compression scheme called top tree compression, where they not only use subtree repeats but also exploit tree pattern repeats of a graph. A tree pattern of T is any connected subgraph of T. a tree pattern repeat is an identical occurrence (both in structure and labels) of a tree pattern of T. Figure 16 illustrates the difference between subtree repeats and tree pattern repeats.

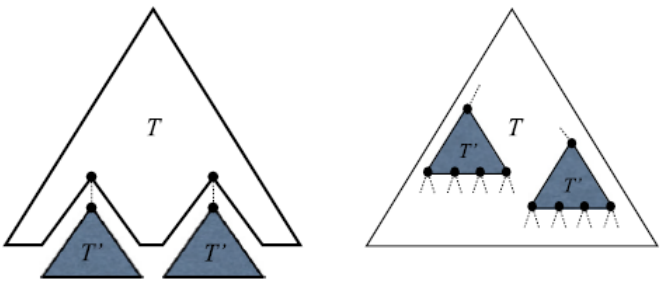


Figure 16: Illustration of the difference between subtree repeats (left) and tree patterns (right) of a tree T [12].

3.2.1 Introduction of Top Trees

Let us start by introducing the basic definitions. Let v be a node in T with children $v_1, v_2, \ldots, v_k$ in left-to-right order. Then let us define $T(v)$, as the rooted subtree rooted in v (a rooted subtree with root v is the subtree that contains v and all of its descendants). Define $F(v)$ as $T(v) - \{v\}$, that is, $F(v)$ is a forest. Let $T(v, v_s, v_r)$ be the tree pattern induced by the nodes of $\{v\} \cup T(v_s) \cup \ldots \cup T(v_r)$, for $1 \leq s \leq r \leq k$, where $v_s, \ldots, v_r$ are the children of v.

We define two kinds of clusters. The first one, the cluster with top boundary node v, is a tree pattern of the form $T(v, v_s, v_r)$, $1 \leq s \leq r \leq k$. The second one, the cluster with top boundary node v and bottom boundary node u, is a tree pattern of the form $T(v, v_s, v_r) - F(u)$, $1 \leq s \leq r \leq k$, where u is a node from $T(v_s) \cup \ldots \cup T(v_r)$. The nodes of a cluster that are not boundary nodes are called internal nodes. Every edge takes the form $(p(v), v)$ for some v, where $p(v)$ is the parent of v. So, for example, every edge $(p(v), v)$ is a cluster with top boundary node $p(v)$ and bottom boundary node v, unless $(p(v), v)$ is a leaf edge, because then this edge is a cluster with only a top boundary node $p(v)$.

Two edge disjoint clusters A and B whose vertices overlap on a single boundary node can be merged if their union $C = A \cup B$ is also a cluster. Clusters can be merged in five different ways, as shown in Figure 17. Notice that the first two types of the merges can only be done if there are no other clusters whose either boundary node is the same as the common boundary node of A and B. Furthermore, the last three types can only be done if at least one of the clusters has no bottom boundary node, since we do not define clusters with two or more boundary nodes.

The main idea of this compression is to create another tree $\mathcal{T}$ from T, called a top tree, such that the tree pattern repeats become subtree repeats in the transformed tree. After this transformation, by applying the classic DAG compression to $\mathcal{T}$ a so-called top DAG, $\mathcal{TD}$, is obtained. This top DAG forms the basis of this compression.

Top Trees A top tree $\mathcal{T}$ of T is a cluster-based hierarchical decomposition of T. It is an ordered, binary, rooted, labeled tree defined as follows:

1. The nodes of $\mathcal{T}$ correspond to clusters of T.
2. The root of $\mathcal{T}$ correspond to the cluster T itself.

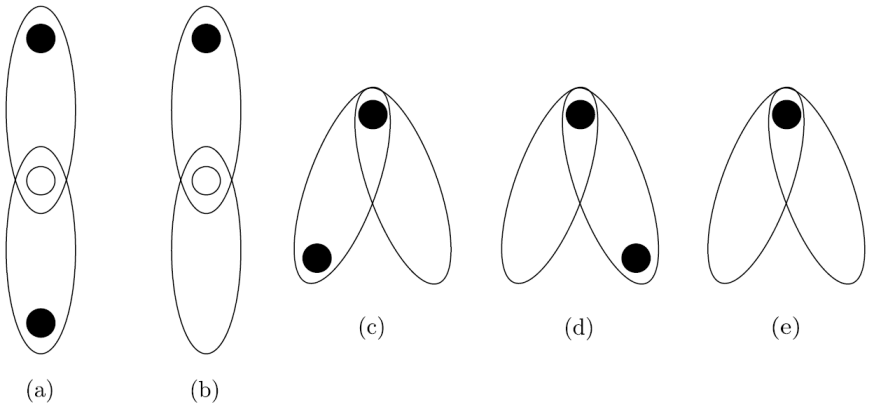


Figure 17: Five possible ways of merging the clusters. The filled nodes are the boundary nodes that remain boundary nodes in the merged cluster, while the empty ones that become internal ones. [12]

3. The leaf nodes of $\mathcal{T}$ exactly correspond to the edges of T. The label of each leaf node is composed of the labels of the ends of the associated edges (u, v) in T, ordered.

4. Each internal node of $\mathcal{T}$ corresponds to the merged cluster of its two children. The label of each internal node is the type of the five merging choices the node represents. The children are ordered so that the left child is the child cluster visited first in a preorder traversal of T.

Top trees were initially introduced by Alstrup et al. [4, 6, 5] for maintaining an uncompressed, unordered, and unlabeled tree under link and cut operations. Bille et al. [12] extended top trees for ordered and labeled trees as well. Their construction is based on the work of Alstrup et al. but has been modified in several ways. Since the objective of this representation is to obtain a superb compression, the construction needs to be carefully organized. Furthermore, some combinations in the merging of clusters have been disallowed in order to remain consistent with the definition of clusters as tree patterns.

3.2.2 Construction of Top Trees

Now, a greedy algorithm is described to construct a top tree $\mathcal{T}$ from the tree T of height $O(\log n)$. The algorithm constructs the top tree in a bottom-up fashion. It starts with the edges of T as the leaf nodes of $\mathcal{T}$. During the algorithm, $\mathcal{T}$ will be a forest, and we maintain an auxiliary rooted ordered tree T', initialized as $T' := T$.

We will be working on this auxiliary tree, T', and based on the operations in this tree, we will be updating $\mathcal{T}$. The edges of T' will correspond to the nodes of $\mathcal{T}$, that is, to the clusters of T. The algorithm consist of $O(\log n)$ iterations, and in each iteration, some edges of T' are merged, which corresponds to the merging of clusters, in our original tree, and thus adding new nodes to $\mathcal{T}$, representing these merges. Furthermore, it is shown that a single iteration shrinks the size of T' by a constant factor, thus achieving the $O(\log n)$ runtime.

So the merging happens amongst the overlapping edges of T' with one of the five options presented in Figure 17. Suppose we want to merge the edge uv with the edge vw. Then there are two cases. **1.** u is the parent of v, and v is the parent of w. **2.** v is the parent of both u and w. In the first case, either type a) or b) of the merges can be applied, depending on whether w is a leaf node or not. The merge contracts the two edges; that is, they are replaced with a single uw edge. In the second case, either type c), d), or e) can be applied, but only if at least one of the clusters has no bottom boundary. This merge replaces the two edges with either the uv or the vw edge.

Let us now present how each iteration is performed.

Step 1 (Horizontal merge): For each vertex v in T' that has at least 2 children, for $i \in \{1, \ldots, \lfloor \frac{k}{2} \rfloor\}$ merge the edges $(v, v_{2i-1}), (v, v_{2i})$ if v_{2i-1} or v_{2i} is a leaf node. Furthermore, if k is odd, v_k is a leaf node, and none of v_{k-2} and v_{k-1} were leaves, then merge the edges $(v, v_{k-1}), (v, v_k)$.

A maximal induced path $v_1, v_2, \ldots, v_p$ in a rooted tree is a maximal path such that v_{i+1} is the parent of v_i for all $i \in \{1, \ldots, p-1\}$, and all of the $v_2, \ldots, v_{p-1}$ vertices have exactly one child.

Step 2 (Vertical merge): For every maximal induced path $v_1, v_2, \ldots, v_p$ in T', if p is even, then merge the following pairs of edges:

$$\{(v_1, v_2), (v_2, v_3)\}, \{(v_3, v_4), (v_4, v_5)\}, \ldots, \{(v_{p-3}, v_{p-2}), (v_{p-2}, v_{p-1})\}.$$

If p is odd, then merge the following pairs of edges:

$$\{(v_1, v_2), (v_2, v_3)\}, \{(v_3, v_4), (v_4, v_5)\}, \ldots, \{(v_{p-4}, v_{p-3}), (v_{p-3}, v_{p-2})\},$$

and if the edge (v_{p-1}, v_p) was not merged in Step 1, then also merge the pair

$\{(v_{p-2}, v_{p-1}), (v_{p-1}, v_p)\}$.

Lemma 3.1. *Each iteration shrinks T' by a factor of $c \geq \frac{8}{7}$.*

Proof. Assume that at the start of an iteration, our tree T' has n' nodes. It is easy to see that any rooted tree on n' nodes has at least $\frac{n'-1}{2}$ nodes that have one or zero children. Consider the edges $(p(v_i), v_i)$ of T', where v_i has less than two children. Let W be the set of these edges. We show that at least half of these edges are merged during an iteration. Since the number of these edges is at least $n/2$, $n/4$ edges are merged, and since merging replaces two edges with one, there are at least $n/8$ times less edges after every iteration. This is proved by assigning each non-merged edge $uv \in W$ to a unique, merged edge $f(uv)$. If we show such an f, then that means that there are at least as many merged edges as there are non-merged edges. And now define f:

Case 1: Suppose that v_i is a leaf and has at least one sibling (i.e. $p(v_i)$ has more than one child). Assume that $(p(v_i), v_i)$ is not merged after the iteration. This can only happen if it has no right sibling, and the previous — left — sibling, w is already merged, that is, $(p(v), u)$ and $(p(v_i), w)$ have been merged, where u is the left sibling of w. They could be merged only if one of them was a leaf. If u is a leaf, then let $f((p(v_i), v_i))$ be $(p(v_i), u)$. Otherwise, if w is a leaf, then let $f((p(v_i), v_i))$ be $(p(v_i), w)$.

Case 2: Suppose that v_i is a leaf with no siblings. Then, the only reason for not merging $p(v_i), v_i$ with $(p(p(v_i)), p(v_i))$ in Step 2 is because $(p(p(v_i)), p(v_i))$ has been merged in Step 1. Therefore, if $(p(v), v)$ is not merged, then let $f((p(v_i), v_i))$ be $(p(p(v_i)), p(v_i))$. Notice that we have not assigned any not-merged edge to $(p(p(v_i)), p(v_i))$ in Case 1, because $p(v_i)$ is not a leaf.

Case 3: Suppose that v_i has exactly one child w and that $(p(v_i), v_i)$ was not merged in Step 1, otherwise we are done. Then the only reason for not merging $(p(v_i), v_i)$ with (v_i, w) in Step 2 is because, based on the length of our path, we already merged (v_i, w) with (w, u), where u is the only child of w. Thus, we can assign $(p(v_i), v_i)$ to (v_i, w). Notice that we have not assigned any not-merged edge to (v_i, w) in Case 1, because w is not a leaf, nor in Case 2, since v_i has only one child, while in the previous case it had to have at least 2 children. □

Due to the fact that each iteration can be calculated in linear time and reduces the

size of T' by a factor greater than one, we obtain the following result:

Corollary Given a tree T, the greedy top tree construction described above creates a top tree of size $O(n)$ and height of $O(\log n)$ in $O(n)$ time.

For the final part of this compression scheme, let us define the top DAG of a tree. The top DAG of T, denoted $\mathcal{TD}$, is the minimal DAG representation of the top tree $\mathcal{T}$. Recall that the minimal DAG representation can be computed in $O(n)$ time 2.1, thus the entire top DAG construction can be done in $O(n)$ time.

3.2.3 Efficiency of Top Tree Compression

In the remainder of this section, we briefly present the results regarding this compression scheme without any calculations or proofs.

First, the compression ratio of the top tree compression is always at least $\log_\sigma^{0.19} n$. More precisely, for a given ordered rooted tree T, whose labels are from the alphabet σ, let the corresponding top DAG be $\mathcal{TD}$. Then $n_{\mathcal{TD}} = O(\frac{n_T}{\log_\sigma^{0.19} n})$ [12]. This result is particularly interesting if we take the information-theoretic lower bound into account, which is $\Omega(\frac{n_T}{\log_\sigma n_T})$. This lower bound is obtained by simply noting that there are $\Omega(\sigma^{n_T})$ string of length n_T over an alphabet of size σ, implying a lower bound of $\Omega(n_T \log \sigma)$ bits or $\Omega(\frac{n_T}{\log_\sigma n_T})$ words. Comparing this to the standard DAG compression, we get a much better result since, in the standard DAG compression, the worst-case bound is $\Theta(n_T)$, given that a single route cannot be compressed by using subtree repeats. Since then, Hübschle-Schneider et al. [32] improved the above-presented tree compression scheme and managed to prove that $n_{\mathcal{TD}} = O(\frac{n_T}{\log_\sigma n} \cdot \log\log_\sigma n_T)$. Furthermore, regarding this latest result, Dudek and Gawrychowski [23] showed that unfortunately no smaller bound is possible for this compression scheme, exploiting a weakness of this scheme. They proved this by constructing a family of trees for which the size of the top DAG is $\Omega(\frac{n}{\log_\Sigma n} \log\log_\Sigma n_T)$, where $\Sigma = \max\{2, |\sigma|\}$.

Second, one might wonder how the top tree compression compares with ordinary DAG compression. Bille et al. [12] managed to prove that their devised top tree compression can compress exponentially better than standard DAG compression and is never worse than the DAG compression by more than a $\log n$ factor, though in the original paper it remained open whether this bound is strict. Later in [11] Bille et al. showed that there exists a family of trees such that the DAG compression is always

smaller by a factor $\Omega(\log n_T)$ than the top DAG. Furthermore, this bound can be achieved even for an alphabet of size 1.

To the best of our knowledge, there has not yet been a comprehensive study comparing computationally the original top tree comparison introduced by Bille et al.[12], the improved version introduced by Hübschle-Schneider et al. [32], and other kinds of methods used for compression in the literature, for example, DAG compression. It might be worthwhile comparing all these different compression schemes.

3.3 Tree Grammars and Succinct Data Structures

In this section, we present two other approaches for tree compression. First tree grammars: These compression schemes exploit tree pattern repetitions, just like top tree compression. Second succinct data structures that are found to be significantly different compression schemes compared to the ones presented in this study so far.

Tree grammars.
Tree grammars can take advantage of tree pattern repeats. Tree grammars, which were examined in [16, 17, 43, 44, 46] generalize grammars from deriving strings to deriving trees. A tree grammar can be exponentially smaller than the minimal DAG in comparison to DAG compression [43]. Unfortunately, it is NP-Hard to compute a minimal tree grammar [18], and all currently used tree grammar-based compression schemes can only support navigational queries in time proportional to the grammar's height, which can be $\Omega(n)$.

Succinct data structures.
A different approach to tree compression is via succinct data structures that compactly encode trees. Jacobson was the first to notice that the simple pointer-based tree representation using $\Theta(n \log n)$ bits is inefficient. Since the number of unlabeled binary trees on n node is $\frac{1}{n+1}\binom{2n}{n}$, the number of bits needed to differentiate these trees is the logarithm of this quantity, which is $2n + o(n)$, so this quantity is an obvious lower bound to the storage complexity of binary trees. Jacobson presented a data structure [33], which used $2n + o(n)$ bits for all trees, not just binaries, and which supports various queries by inspection of $\Theta(\log n)$ bits. This space bound is asymptotically optimal with the information-theoretic lower bound. Munro and Raman [50] demonstrated how to obtain the same bound using only constant time for queries in the RAM model. Such representations are called succinct data structures

and have been generalized to include a richer set of queries such as subtree-size queries [9, 50].

Regarding the labeled case, Ferragina et al. [24] gave a representation using $2n \log |\Sigma| + O(n)$ bits (where the labels are from Σ), that supports basic navigational operations among the immediate neighbors of a node v, such as finding the parent of node v, the i^{th} child of v, or any child of v with a label l.

Ferragina et al. also developed the concept of k^{th} order tree entropy H_k in a restricted model. In this model, used by popular XML compressors [19, 42], the label of a node is a function of the labels of all its ancestors. Ferragina et al. provided a representation that required no more than $nH_k(T) + 2.01n + o(n)$ bits for such a tree T. It should be noted that the aforementioned space constraints do not ensure a compact representation when the input contains numerous tree pattern repeats or subtree repeats, unlike in other representations. In particular, the total space is never $o(n)$ bits.

4 Efficient Implementation of the Ordering Heuristics

So far we considered Problem 1.1 mainly from theoretical perspective. This section presents the approaches from a practical point of view. We implemented a data structure to handle Problem 1.1. This section presents the basic functionalities of our data structure and provide a comprehensive comparison of the results obtained by our data structure versus a state-of-the-art solver.

4.1 Compact Representation of the Solutions of a Sudoku

The first example we investigated — which we later extended, see Section 4.2 — was the following problem: for a given Sudoku board, encode all of its solutions as compactly as possible. At first glance, this might not seem like an attractive problem, since usually for a given Sudoku board there is only one solution. However, if some of the predefined values are deleted from the cells, the number of solutions increases, thus resulting in an interesting problem. We chose this task in the hopes that the results obtained in this problem could be used in the future to compress real-life

variant tables.

In the classic Sudoku problem, our objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid contain all of the digits from 1 to 9. One might define other games by changing the size of the grid. For now, let us consider the classic game. As a first step, we needed a Sudoku solver that could list all possible solutions of a Sudoku table (which could even be empty). This can be solved with a backtracking algorithm. We present this algorithm, because it will be modified later to not only list all of the possible solutions of our given table, but to construct the DAG representation of the solutions straight away. Let us be given the processing order of the cells; T, based on which we fix the values in the cells. At a node v in the backtracking tree at level i, process the cell $T[i]$: for each of the possible values $k \in 1, \ldots, 9$ that causes no collision, test whether the Sudoku can be solved by setting the value of the cell $T[i]$ to k, i.e., add $T[i] = k$ to our table and descend to the next level. If we get down to the last level and there is a value we can write into $T[89]$ (that is, into the last cell to be processed) such that it does not cause any collision, then it means that we obtained a feasible solution for our Sudoku. Save this solution, and the backtracking can return and continue the traversal of the backtracking tree. At a general step, after trying out every value in $T[i]$ at our node v, let the recursion return true to the parent of v, $p(v)$, if the Sudoku was solvable with any of the values, and let it return false if the Sudoku was not solvable with any of the values.

At the end of this backtracking algorithm, we have all of the solutions to the given input Sudoku table listed. But our objective is to store these solutions concisely. Next, we show how the algorithm can be modified not only to enumerate all solutions but to build a representing DAG during the execution, which can be read out at the end of the run.

One way to obtain the DAG representation encoding all of the solutions of a given Sudoku table would be by using the algorithm described in Section 2.1. Create a decision tree from the solutions, then merge the identical subtrees. But this method would require the construction of a decision tree, which we would like to avoid. Now we show how to construct the same DAG representation D during the run without constructing the decision tree.

A dictionary is a data structure that has a set of keys, and each key an associated value. For example hash tables are dictionaries. Let S be a new dictionary whose

keys are sets of tuples and whose values are integers. During the construction of our DAG D, as D grows, so will the number of different rooted subdags too. Each rooted subdag of D receives a label, which tells us what was the first time a subdag like that was encountered. This dictionary S will contain every piece of information required about these rooted subdags. Next, we present the structure of S in order to understand the usage of S during the algorithm more easily.

- Each entry in the dictionary corresponds to a vertex v of the DAG representation.
- The values are the labels of the subdag rooted at v.
- The key corresponding to the entry of v is an ordered tuple of pairs. Each pair (x, y) represents a child c of v, where x is the label of the subdag rooted in c and y is the value of the edge vc.

We now show how to construct the DAG representation during the run of the backtracking algorithm.

Suppose that we are at a node v of the backtracking tree, and the algorithm has just finished traversing the subtrees of the backtracking tree rooted in the children of v, i.e., it has just stepped back from the last child. Then, by induction, we know which subdags are rooted in the children, since below v we have already compressed the subtrees of the children into DAGs. Let the children be $v_1, \ldots, v_k$, the values of the edges, by which we entered the children, be $e_1, \ldots, e_k$ — that is the value we wrote into the corresponding cell — and the label of the subtrees rooted underneath be $l_1, \ldots, l_k$. Let $w := [(l_1, e_1), (l_2, e_2), \ldots, (l_k, e_k)]$, where this list is sorted lexicographically. Then there are two possibilities: Either$w \in S$ or $w \notin S$.

In the first case, read the corresponding value of w from S and return it. In the second case, add a new entry: $(key, value)$ to our dictionary S. Let the key be w, and the corresponding value be the index of the subdag that has been defined most recently plus one (this will be the label of the subdag rooted in v). After expanding the dictionary, return the value.

This backtracking algorithm builds the dictionary S while running, from which the DAG representation can be constructed at the very end. Since parent-child relations are stored in the keys, it is possible to build the DAG in a nice sequential way. However, due to the structure of the dictionary, we do not get the explicit

representation, so in addition, an explicit representation must be built during the algorithm if we want to meet the conditions to have a representation on which various queries can be done quickly. The most important thing to keep in mind when devising this explicit representation was that after the swapping of two adjacent levels, the new representation should be easily calculable since we would like to implement ordering algorithms for our data structure as well, and these algorithms are based on the swaps of adjacent levels. Section 4.2.2 presents how this swap operation can be performed in our data structure.

Once we have the representation and a swapping algorithm, we can focus on figuring out how the variables should be ordered. So to recall, our task is to determine the solutions to a given Sudoku table and represent them in the most compact way possible. In our case, we considered and encoded the solutions to an empty 4×4 Sudoku table. As a first step, we implemented a similar procedure to the sifting algorithm [52], which was discussed in Section 2.3. We then implemented our own heuristic algorithm based on simulated annealing, see Algorithm 2. Simulated annealing algorithms were also investigated in Section 2.3 in more depth. A comparison between these two algorithms can be seen in Figure 18. In this example, we have taken the solutions of an empty Sudoku and examined the sizes of their DAG representations for different variable orderings before and after reordering. The horizontal axis shows the original size of the representation for different initial variable orderings before reordering, while the vertical axis shows the size of the representation after reordering. The blue dots correspond to the simulated annealing algorithm, while the red ones correspond to the sifting-based algorithm.

As we can see, our algorithm based on simulated annealing found a representation of size 501 in all cases — which is supposedly the optimal size in our representation to encode all of the solutions of a 4×4 Sudoku — while the heuristic based on the sifting algorithm found the best representation only in a few instances.

In order to compare our algorithm with the state-of-the-art reordering algorithms in the Python library DD, we had to switch from our MDD-like representation to a BDD-like representation. To do this, we also modeled the problem using logical formulas and then modified the algorithms accordingly. The running results are shown in the tables in Figure 19. Each row corresponds to a concrete run. The first column includes the size of the representation before reordering, the second the DD achieves, while the third our algorithm achieves after reordering.

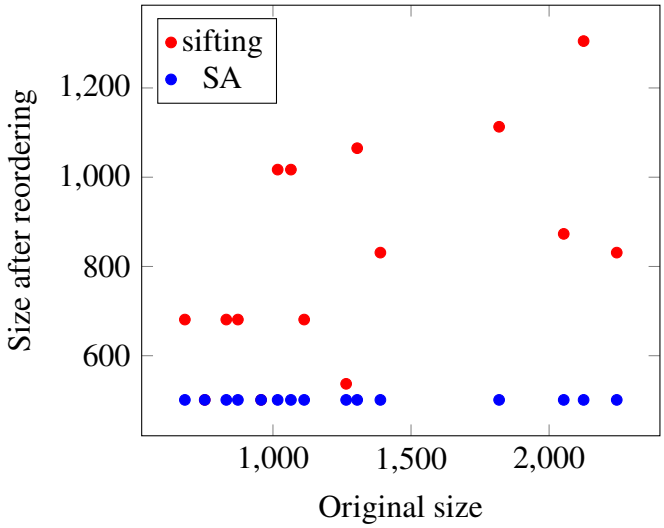


Figure 18: Comparison between the two approaches for the Sudoku problem; sifting and simulated annealing based.

Observe that, for smaller BDDs, there is not much difference between the reorderings, but as we increase the size of the BDD, in general, our simulated annealing heuristic performs better and better compared to the best algorithm implemented in the DD library.

We do not discuss running times regarding this Sudoku problem because, at the time we investigated it, the implementation of our representation still lacked the implementation of the efficient level swapping algorithm described in Section 4.2.2. The fact that we had to repeat the backtracking constructing process for each swap in order to calculate the size of the representation for the new order slowed down the execution of our heuristics remarkably, making it much more time-consuming than the techniques from the DD library.

Although we examined this task in Python, we have implemented the more general representation in C++. In the following sections, we switch to analyzing this more general program.

Original	DD	SA
105	79	81
336	196	198
336	196	219
336	196	198
336	196	186
336	196	225
336	196	192
1,336	843	879
1,336	843	880
1,336	843	876
1,336	843	873
1,336	843	935
1,515	933	889
1,515	936	901
2,496	1,241	1,148

Original	DD	SA
2,496	1,225	1,222
2,496	1,260	1,183
2,496	1,330	1,103
2,496	1,284	1,368
2,496	1,196	1,270
4,340	2,325	2,274
4,340	2,220	2,252
5,991	4,017	4,112
5,991	3,958	2,765
5,991	3,978	3,290
5,991	3,911	4,112
5,991	3,938	3,904
5,991	4,032	3,069
5,991	3,973	4,007
5,991	4,094	2,923

Figure 19: The heuristic based on simulated annealing compared to the reordering algorithm found in the DD Python library.

4.2 Implementation details

In this section, we present our data structure that we devised to handle Problem 1.1.

To begin, let us review the specific real-life problem we aim to address. We are given a variant table that were defined in Section 1.1. A variant table is used to store combinations. A fragment of a concrete real-life variant table we were given to compress is shown in Table 5.

CUCO_OBJE	GEN_S_MODEL	GEN_S_VAR	GEN_S_VERSION
MF7S.145D6	Q7I	6	Q7IST2
MF7S.145DVT	Q7I	1	Q7IST2
MF7S.145D6	Q7J	7	Q7IST2
MF7S.145D6	Q7A	2	Q7IST2

Table 5: A fragment of a real-life variant table.

Our goal is to encode the combinations in this variant table as efficiently as possible so that in the resultant representation certain operations can be performed efficiently, such as searching among or modifying the represented combinations. We implemented a data structure that is based on the DAG representation (see Section 2.1) to tackle this problem. In the next section, we present the construction of our data

structure, then we demonstrate how the swap operation may be carried out in our data structure. After that, we briefly present a state-of-the-art solver, the CUDD package, and then compare our reordering algorithms to the algorithms implemented in the CUDD package. Finally, we outline the plans for the future of this data structure.

4.2.1 Construction

Let us be given a variant table. We construct our representation exactly the way presented in Section 2. First, we construct the decision tree corresponding to the variant table, then we compress this tree into a DAG using the hashing algorithm, i.e., merging identical subtrees, see Section 2.1. In addition, we create and store a dictionary that we described in the backtracking algorithm in Section 4.1.

After obtaining the DAG representation for the given order, we can turn our attention toward the reordering algorithms. Even at this point, we could implement reordering algorithms, reconstructing the DAG representation from scratch each time we were interested in the size of a different variable order, but this approach would be way too wasteful. Instead, by implementing the swap process for our data structure, the running time for the reordering algorithms can be significantly decreased. In the next section, we investigate how one might execute the swap process.

4.2.2 Swap of Adjacent Levels

As we saw in Section 2.3, the foundation of any algorithm for ordering variables is the swap of adjacent levels. This begs the question of how to carry out these exchanges. This section discusses the process of making these exchanges and investigate their implementation.

Suppose we want to swap the first row with the second one; since it is easier to talk about the first and second levels instead of the i^{th} and the $(i+1)^{\text{th}}$. The process is the same for the general case with a few exceptions, which are discussed at the end of this section.

Let us take a concrete example. At the upper part of Figure 20, we can see the two levels before the swap, and at the bottom part, after the swap. A swap only changes the number of nodes on the second level. The nodes of every other levels are kept intact, including the first and third levels as well. However, the edge number might

change between both the first and second and also second and third levels. As we can see in the figure, we have one more node after the swap, so ultimately, this swap increases the total number of nodes in our representation.

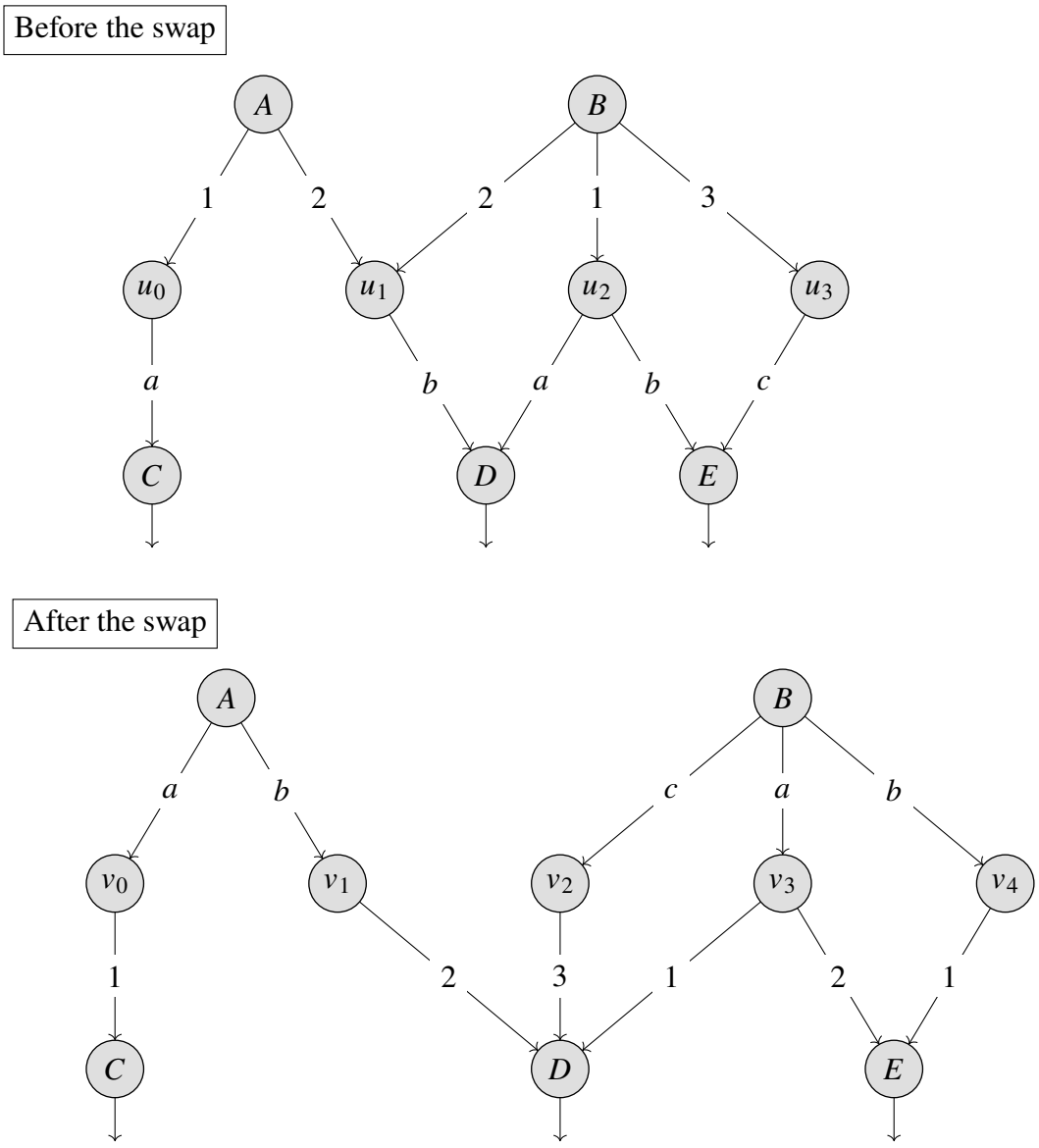


Figure 20: Example for the swap process.

Let us now describe the swap process precisely in our DAG representation. Let the current DAG representation be D. We start the procedure by deleting every entry from our dictionary corresponding to the nodes on the second level. Let Q_0 be the set of vertices at the second level at the beginning, and Q be the set of the new vertices on the second level. At the start, Q is empty. We do the following for each vertex v at the first level: Let P_v be the set of the paths of length two rooted in

v whose second vertex is in Q_0, and let R_v be an auxiliary vertex set — to which vertices are added later. Then iterate through P_v, let (e_1, e_2, w) be a triple, where e_1 is the value of the first edge, e_2 is the value of the second edge, and w is the label of the third node of the current path. Then there are two cases:

1. There is a node $u \in R_v$ such that the value of the edge vu is e_1.
2. There is no such u.

In the first case, add the arc uw with value e_2 to D. In the second case, add the node u to R_v, the arc vu with a value e_1 and the arc uw with a value of e_2 to D. After iterating through P_v, consider R_v. For each node u in R_v check whether there is another node w in Q such that w spans the same rooted subtree as u. If there is, then merge the nodes u and w. Otherwise, delete u from R_v and add it to Q.

After iterating through the nodes on the first level, delete all vertices in Q_0 and all incident arcs. Thus, we obtained the DAG representation where the first and second variables are swapped. In addition, we need to keep the dictionary updated during the run (in practice, that is how we check if there is another identical subtree). We do this by adding a new entry each time a node is placed into Q. Moreover, at the end, all of the dictionary entries have to be updated on the first level accordingly.

In the general case, the swap of the i^{th} and the $(i+1)^{\text{th}}$ levels can be done identically, with the exception that the nodes in the i^{th} level have incoming arcs, and those ones have to be kept intact.

4.2.3 Experimental Results

In this section, we present experimental results using our data structure.

Let us begin by presenting results regarding the sifting algorithm. We implemented Algorithm 1 with a few modifications. As we have mentioned in Section 2.3.3, Algorithm 1 might be rerun until we obtain a 2-optimal solution.

Figure 21 illustrates how successive iterations managed to improve the size of our representation. Each bar shows the percentage of reduction in size of the current BDD compared to its size at the start of the corresponding iteration. As we can see, the first iteration reduced the size of our DAG by over 80% in general, the second by around 10%, and the other iterations only marginally. We iterated Algorithm 1 until

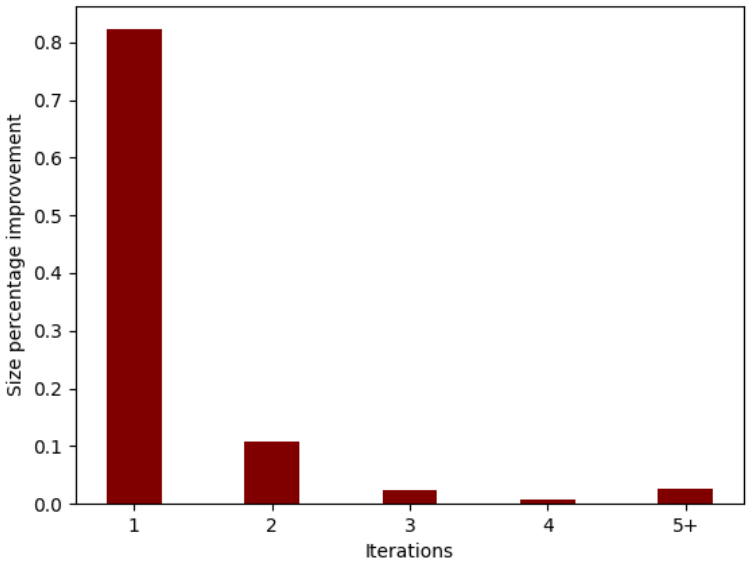


Figure 21: The column denoted by the label 5+ is the sum of all of the gain obtained after the fourth iteration.

a 2-optimal solution was found, so there were runs which took up to 12 iterations, but usually it ended in $6-8$ iterations. Observe that in general, we were about 3% off from the 2-optimal solution after the 4^{th} run. The results are based on multiple variant tables taken from real-life examples.

Figure22 illustrates the running times for each iteration. Here, the last column should be understood as from the fifth iteration onwards (basically from the second), all iterations running times are the same in general. As we can see, the first iteration, where the big improvement is, usually takes around six times longer than any other iteration. According to Knuth [39], additional iterations are usually not worth the extra effort, but based on our measurements, it is usually worth running two iterations since it only increases the running time by 15% and reduces the number of nodes in the representation by almost 10%.

Another interesting aspect is how the size of the representation changes over time during reordering. Figure 23 illustrates this for a concrete real-life variant table and for some random-shuffled initial variable orders. The curves illustrate how this algorithm reduces the size of our representation in the course of its execution.

We also implemented Algorithm 2. As we have mentioned in Section 2.3.3, implementing an algorithm like this involves a lot of fine-tuning of the parameters.

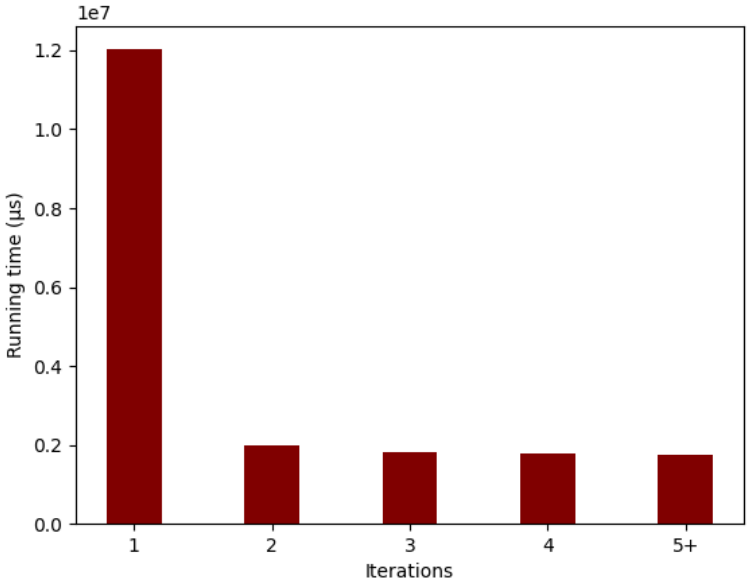


Figure 22: Running time for each iteration in Algorithm 1.

Figure 24 illustrates the same as Figure23, just for this second algorithm. As we can see, both Algorithms 1 and 2 obtain similar results despite their fundamentally different approaches.

We also wanted to compare our reordering algorithms with state-of-the-art reordering methods. We chose the CUDD package for the sake of comparison. CUDD is an abbreviation for Colorado University Decision Diagram and was developed by Fabio Somenzi [56]. As the name suggests, the CUDD package can be used for the manipulation of multiple types of decision diagrams, including BDDs, ADDs and ZDDs. In order to be able to compare our results to the results of CUDD, we had to apply some tricks since CUDD deals with binary decision diagrams.

In Section 2.2, we have already seen how a BDD B representing our DAG representation D can be constructed with the help of the gadget illustrated in Figure 6. However, we cannot apply reorderings to this BDD just yet because we would like to keep the levels corresponding to the same gadgets adjacent in our BDD B during reordering, since at the end, only this way could we obtain a BDD that corresponds to a real order on the original variables — which we need in order to compare the results. Fortunately, there is a feature in CUDD that lets us "group" variables.If we group variables, then CUDD handles the reordering as if the grouped variables would be glued together.

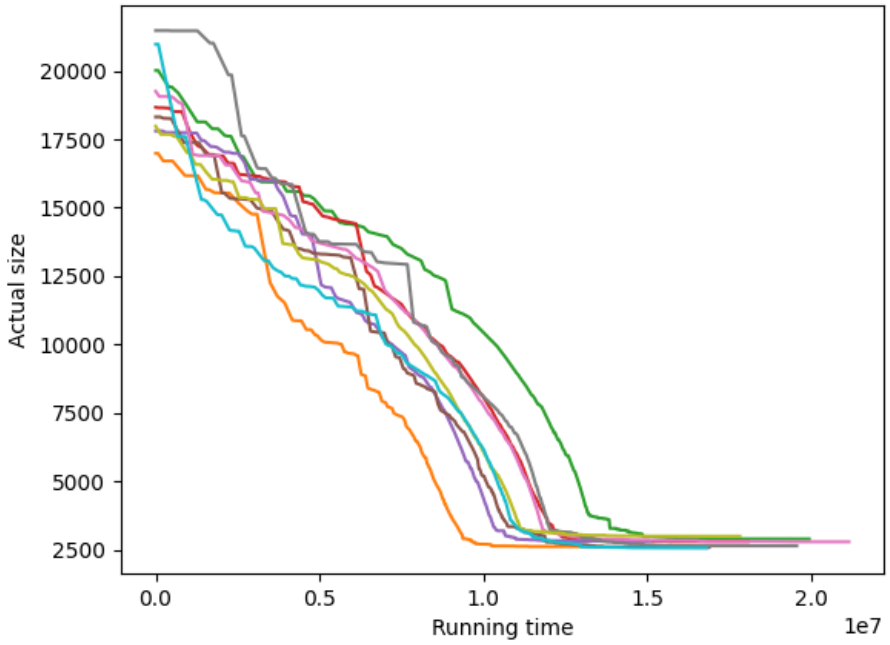


Figure 23: Running time for each iteration in Algorithm 1.

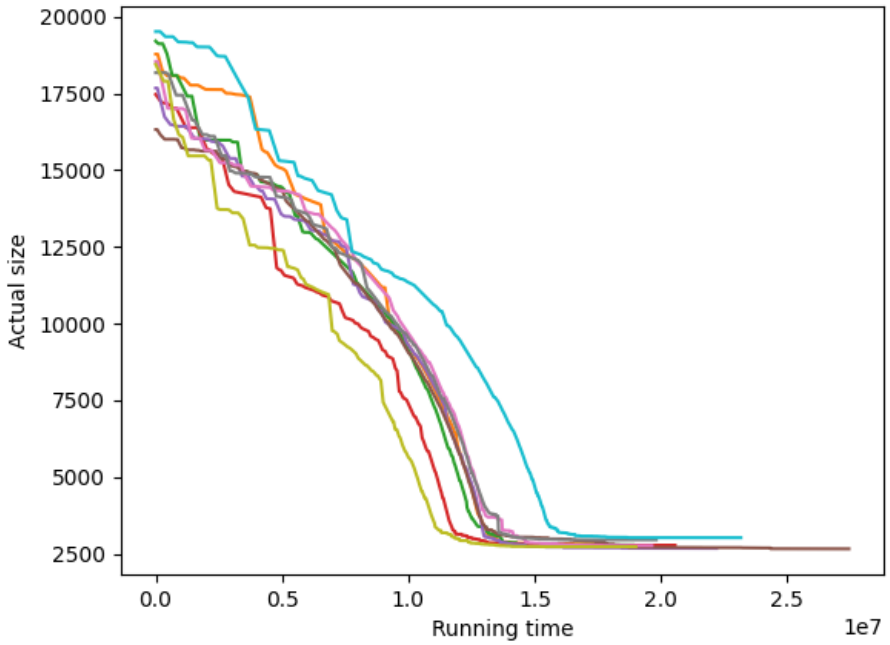


Figure 24: Running time for each iteration in Algorithm 2.

Notice that during the comparisons, we give CUDD a small advantage since, in theory, we could reinforce our DAG representation with a generalized form of the elimination rule, but after testing it in practice, this rule would improve the size of the representation by less than 1%. Since adding this rule would also mean we lose the property that exactly the n-paths correspond to the valid combinations, we chose not to continue with that rule. Tables 6 and 7 present how our algorithms implemented using our data structure compare to the ones in the CUDD package. Table 6 compares the sizes of the representations our Algorithms (A_1, A_2, A_3) obtain to the sizes of the representations the algorithms in the CUDD package ($C_1, \ldots, C_5$) obtain on different inputs. In the tables the row correspond to the sam input, and the column to the specific algorithms. Our algorithm A_1 corresponds to the implementation of Algorithm 1, A_2 to Algorithm 2 while A_3 to a combination of these two algorithms, i.e., after a variant of the A_2 algorithm, we run a variant of the A_1 algorithm to find a 2-optimal solution.

	A_1	A_2	A_3	C_1	C_2	C_3	C_4	C_5
$VT1_1$	444	456	450	418	836	366	514	619
$VT1_2$	448	460	400	411	419	390	440	532
$VT1_3$	451	436	390	391	973	382	385	514
AVG	447.7	450.7	413.3	406.7	742.7	379.3	446.3	555
$VT2_1$	2706	3064	2501	2587	3050	2510	3050	4162
$VT2_2$	2705	2988	2426	2914	3337	2541	3337	3199
$VT2_3$	2878	3339	2490	2786	3366	2414	2689	3340
AVG	2763	3130.3	2472.3	2762.3	3251	2488.3	3025.3	3567
$P1_1$	9252	9307	9197	9258	9272	9224	9207	9185
$P1_2$	9223	9300	9162	9241	9232	9196	9213	9252
AVG	9237.5	9303.5	9179.5	9249.5	9252	9210	9210	9218.5

	A_1	A_2	A_3
$P2_1$	14558	14517	14508
$P2_2$	14516	14510	14496
AVG	14537	14513.5	14502

	C_1	C_2	C_3	C_4	C_5
$P2_1$	14534	14546	14526	14499	14535
$P2_2$	14501	14569	14547	14496	14576
AVG	14517.5	14557.5	14536.5	14497.5	14555.5

Table 6: Comparison of the representation sizes obtained by several algorithms.

The first column contains the names of the input data sets. The subscript means that we are given the same variant table, but we constructed our representation in a different variable order and applied our algorithms to the representation obtained this way. On the one hand, we compared our results based on variant tables taken from real-life applications ($VT1, VT2$). On the other hand, recall the path encoding problem investigated in Section 2.2.4. In that problem, we were given a grid graph, and we wanted to encode all of the paths going from one corner to the opposite corner. Motivated by that problem, we investigated the encoding of monotone paths in a grid graph, i.e., the paths, only including edges that go from left to right, or from top to bottom if we want to encode the paths from the top left corner to the bottom right corner. The problem defined exactly this way is not that interesting since the grid graph itself would be the most compact way one might represent all of the paths using the DAG representation, which solution is found by all of the algorithms. So we investigated the problem where we only wanted to encode a certain subset of the possible monotone paths. The data sets P_1 and P_2 correspond to these cases.

	A_1	A_2	A_3	C_1	C_2	C_3	C_4	C_5
$VT1_1$	1.18	8.43	12.77	0.02	0.01	4.5	2.72	0.02
$VT1_2$	1.13	8.2	10.23	0.03	0.01	3.94	4.19	0.02
$VT1_3$	1.17	12.49	11.38	0.03	0.01	2.7	3.35	0.03
AVG	1.16	9.71	11.46	0.03	0.01	3.71	3.42	0.02
$VT2_1$	24.2	106.5	156.6	0.5	1	74.8	58.3	0.7
$VT2_2$	24.8	145.5	160	0.4	0.9	79.3	66	0.8
$VT2_3$	26.2	140.8	180	0.5	0.8	85.2	54.8	0.3
AVG	25.1	130.9	165.5	0.5	0.9	79.8	0.5	0.4
$P1_1$	35.9	72.8	154.7	10.6	10.4	17.3	16.4	10.2
$P1_2$	36.4	85.6	159.8	9.5	11.3	20.9	17.7	11.3
AVG	36.2	157.3	79.2	10.1	10.9	19.1	17.1	10.8
$P2_1$	58.9	238.9	282.5	9.9	11.6	32	24.6	10.3
$P2_2$	56.9	273.6	457.1	10.9	11.4	35.8	26.2	8.8
AVG	57.9	256.3	369.8	10.4	11.5	33.9	25.4	9.6

Table 7: The running times of the considered algorithms. All data are in seconds.

Based on Tables 6 and 7 algorithms C_3 and C_4 from the CUDD package Let us now examine the results obtained by our data structure. Based on Tables 6 and 7 from the CUDD package algorithms C_3 and C_4 obtain the best results, although they are the slowest. Surprisingly, on different inputs, different algorithms performed better. On the variant tables taken from real-life applications C_3 performed better while on

the path encoding problem C_4 performed better. Now considering our algorithms, A_3 outperformed both A_1 and A_3 in almost all cases, although it achieved the better orderings over a longer period of time. Comparing A_3 to C_3 and C_4, we can see that usually — except in the smallest case — it outperformed both.

Regarding the running times, our future is to beat the baseline C_i algorithms as well; therefore, in the next section, we give potential approaches for speeding up our implementation.

4.3 Future Plans for the Data Structure

As we saw in the previous section, our data structure can handle Problem 1.1 quite well as far as the quality of the output is concerned; nevertheless, this section presents several areas where our data structure could be enhanced.

Algorithms A_1, A_2 and A_3 are usually slower than the ones in the CUDD package. There are several ways we could speed up our reordering algorithms. For example, one way would be to accelerate the swap procedure. The algorithm now implemented an order of magnitude slower than the one presented in Section 4.2.2, so implementing the presented one would likely decrease the running times in practice as well. The algorithms themselves could also be sped up by experimenting with stronger stop conditions than the ones implemented now.

Our data structure currently uses a balanced binary search tree as the dictionary introduced in Section 4.1. A way to enhance our data structure would be to replace this data structure with a hash table. This would probably speed up the search for identical subtrees significantly. Since this search for identical subtrees would be interesting in the construction of our representation, this observation brings us to the most interesting possible extension of our data structure. Our future goal is to extend the DAG representation in such a way that it is capable of generating all the valid combinations based on a set of rules and building up the representation simultaneously.

References

[1] B. Abdalhaq, A. Awad, and A. Hawash. A fast binary decision diagram (BDD)-based reversible logic optimization engine driven by recent meta-heuristic reordering algorithms. *Microelectronics Reliability*, 123:114–168, 2021.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, 27(06):509–516, 1978.

[3] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Mince: A static global variable-ordering heuristic for SAT search and BDD manipulation. *J. Univers. Comput. Sci.*, 10(12):1562–1596, 2004.

[4] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming: 24th International Colloquium, ICALP'97 Bologna, Italy, July 7–11, 1997 Proceedings 24*, pages 270–280, 1997.

[5] S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *Acm Transactions on Algorithms (talg)*, 1(2):243–264, 2005.

[6] S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *Algorithm Theory-SWAT 2000: 7th Scandinavian Workshop on Algorithm Theory Bergen, Norway, July 5–7, 2000 Proceedings 7*, pages 46–56. Springer, 2000.

[7] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 118–132. Springer, 2007.

[8] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 188–191, 1993.

[9] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, 2005.

[10] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[11] P. Bille, F. Fernstrøm, and I. L. Gørtz. Tight bounds for top tree compression. In *String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017, Proceedings 24*, pages 97–102. Springer, 2017.

[12] P. Bille, I. L. Gørtz, G. M. Landau, and O. Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015.

[13] U. Blumöhr, M. Münch, and M. Ukalovic. *Variant configuration with SAP*. Galileo Press, 2012.

[14] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on computers*, 45(9):993–1002, 1996.

[15] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[16] G. Busatto, M. Lohrey, and S. Maneth. Grammar-based tree compression. Technical report, 2004.

[17] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4):456–474, 2008. Selected Papers from the Tenth International Symposium on Database Programming Languages (DBPL 2005).

[18] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[19] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proceedings DCC 2001. Data Compression Conference*, pages 163–172. IEEE, 2001.

[20] A. A. Cire and W.-J. Van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

[21] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[22] R. Drechsler. Evaluation of static variable ordering heuristics for MDD construction [multi-valued decision diagrams]. In *Proceedings 32nd IEEE International Symposium on Multiple-valued Logic*, pages 254–260. IEEE, 2002.

[23] B. Dudek and P. Gawrychowski. Slowing down top trees for better worst-case compression. In *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[24] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, 57(1):1–33, 2009.

[25] S. Fortune, J. Hopcroft, and E. M. Schmidt. The complexity of equivalence and containment for free single variable program schemes. In *Automata, Languages and Programming: Fifth Colloquium, Udine, Italy, July 17–21, 1978 5*, pages 227–240. Springer, 1978.

[26] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 348–356, 1987.

[27] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 38–41. IEEE, 1993.

[28] M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of NP-completeness, 1979.

[29] A. Haag. Managing variants of a personalized product: Practical compression and fast evaluation of variant tables. *Journal of Intelligent Information Systems*, 49:59–86, 2017.

[30] A. Haag and L. Haag. Further empowering variant tables for mass customization. *International Journal of Industrial Engineering and Management*, 10(2):155–170, 2019.

[31] A. J. Hu. *Techniques for efficient formal verification using binary decision diagrams*. Stanford University, 1996.

[32] L. Hübschle-Schneider and R. Raman. Tree compression with top trees revisited. In *International Symposium on Experimental Algorithms*, pages 15–27. Springer, 2015.

[33] G. Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554. IEEE Computer Society, 1989.

[34] S. W. Jeong, T. S. Kim, et al. An efficient method for optimal BDD ordering computation. In *ICVC: International Conference on VLSI and CAD*, volume 3, pages 252–256, 1993.

[35] C. Jiang, J. Babar, G. Ciardo, A. S. Miner, and B. Smith. Variable reordering in binary decision diagrams. In *26th International Workshop on Logic & Synthesis*, 2017.

[36] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 379–393. Springer, 2007.

[37] M. G. Kendall. Rank correlation methods. 1948.

[38] D. Knuth. Fun with binary decision diagrams (BDDs).

[39] D. E. Knuth et al. *The art of computer programming*, volume 3. Addison-Wesley Reading, MA, 1973.

[40] C.-Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

[41] W. Lenders and C. Baier. Genetic algorithms for the variable ordering problem of binary decision diagrams. In *Foundations of Genetic Algorithms: 8th International Workshop, FOGA 2005, Aizu-Wakamatsu City, Japan, January 5-9, 2005, Revised Selected Papers 8*, pages 1–20. Springer, 2005.

[42] H. Liefke and D. Suciu. Xmill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 153–164, 2000.

[43] M. Lohrey and S. Maneth. The complexity of tree automata and xpath on grammar-compressed trees. *Theoretical Computer Science*, 363(2):196–210, 2006. Implementation and Application of Automata.

[44] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using repair. *Information Systems*, 38(8):1150–1167, 2013.

[45] C. Luck. Simulated annealing explained by solving sudoku - artificial intelligence.

[46] S. Maneth and G. Busatto. Tree transducers and tree compressions. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, pages 363–377, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[47] S.-i. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, pages 272–277, 1993.

[48] A. Mishchenko. An introduction to zero-suppressed binary decision diagrams. In *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, volume 8, pages 1–15, 2001.

[49] A. Mitra and S. Chattopadhyay. Variable ordering for shared binary decision diagrams targeting node count and path length optimisation using particle swarm technique. *IET Computers & Digital Techniques*, 6(6):353–361, 2012.

[50] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[51] M. Rice and S. Kulhari. A survey of static variable ordering heuristics for efficient BDD/MDD construction. *University of California, Tech. Rep*, page 130, 2008.

[52] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47. IEEE, 1993.

[53] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 57(12):713–723, 1938.

[54] D. Sieling. The nonapproximability of OBDD minimization. *Information and Computation*, 172(2):103–138, 2002.

[55] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, 48(3):139–144, 1993.

[56] F. Somenzi. CUDD: Cu decision diagram package release 2.3. 0. *University of Colorado at Boulder*, 621, 1998.

[57] F. Somenzi. Binary decision diagrams. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:303–368, 1999.

[58] Wikipedia. Binary decision diagram — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Binary%20decision%20diagram&oldid=1142348989, 2023. [Online; accessed 07-June-2023].

[59] Wikipedia. Simulated annealing — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Simulated%20annealing&oldid=1154654876, 2023. [Online; accessed 06-June-2023].

[60] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 681–690, 2006.

NYILATKOZAT

Név: Simon Máté

ELTE Természettudományi Kar, szak: Alkalmazott matematikus

NEPTUN azonosító: GHLKWP

Diplomamunka címe:
Compact Representation of Labeled Trees

A **diplomamunka** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2023. június 7.

a hallgató aláírása