Universiteit Leiden
The Netherlands

Opleiding Informatica

Representing the state space of the Domino cube in graph using BDDs

Fien van Wetten

Supervisors:
dr. H.J. Hoogeboom & dr. W.A. Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

04/08/2022

Abstract

This thesis is about representing the state space of the Domino cube in a graph. For every depth of the graph we use a BDD to represent the states. The ordering of the variables int the BDD have affect on the size of the BDD. Encoding the Domino cube differently changes the ordering of the variables in the BDD. The goal is to find out whether different encodings of the Domino cube have effect on the size of its BDDs. We found that the ordering of the pieces matters for the size of the BDDs.

Contents

1 Introduction

The Rubik's cube is a well-known puzzle of which there are many variants. Some of these are more familiar than others. Of the regular cubes, dimensions can range from the pocket cube ($2 \times 2 \times 2$) to the largest mass-produced cube at $21 \times 21 \times 21$. Other puzzles of the same type which are not cube-shaped also exist. An example of a non cube-shaped puzzle is the Domino cube: this puzzle has two 3×3 sides and four 2×3 sides for total dimensions of $2 \times 3 \times 3$.

We attempted to compactly represent the state space of the Domino cube, containing all the unique possible states of the Domino cube. If we can represent the state space than we can find the shortest path to the solved state for any scramble of the Domino cube and find God's number. God's number is the graph diameter which is the minimum number of turns required to solve the cube from an arbitrary starting position. We have tried to generate this graph consisting of the states of the cube and the moves between them. Every state is a certain number of moves away from the initial solved state, which we call its depth. Every "depth of the graph" contains its own states, which are represented in a Binary Decision Diagram.

A Binary Decision Diagram or BDD represents Boolean functions, so we encoded the states of the Domino cube as a binary string representing an input of a Boolean function. In this way a BDD could be made for every depth on the graph. The BDD has the property that the ordering of the variables of the Boolean function matters for the size of the BDD. So if we have the following Boolean function $(x1 \wedge x2) \vee x3$ then the size of the BDD with ordering $x1 < x2 < x3$ could be smaller than when the ordering is $x2 < x1 < x3$.

The goal of this research is to find out whether different encodings of the Domino cube have effect on the size of its BDDs, of which there is one for each depth. Ideally we want the BDDs to be as small as possible. That the BDDs are small is important due to the large number of states of the cube. If the BDDs are too big, then the graph might not fit in the memory of the PC. With all complete BDDs, we can find the shortest path to the solved state, also called the solution, for any scramble of the cube and determine God's number.

1.1 Thesis overview

The thesis is organised as follows, Section 2 contains the background information. Section 3 describes how the state space of the Domino cube is made and represented. Section 4 describes the experiments and their outcome. Section 5 concludes the thesis.

This bachelor thesis was written as a final project for the computer science bachelor at the Leiden Institute of Advanced Computer Science (LIACS), and was supervised by dr. H.J. Hoogeboom and dr. W.A. Kosters.

2 Background

2.1 Rubik's cube and graphs

The $3 \times 3 \times 3$Rubik's cube is a puzzle game that was invented in 1974 by Hungarian professor Ernő Rubik. The puzzle can be scrambled and solved by turning the faces of the cube. Every face of the cube can be turned 90°, on every turn the cube comes in a different state. Rubik was also the first person who solved the puzzle. It took him over a month to find a solution [McG16]. Through the years, scientists have made many calculations about this puzzle. Only in 2010 it was proven that every cube state can be solved in 20 face turns or less. This is called God's number [vG10]. But this was a process that took over 30 years.

Finding God's number was approached by finding a lower bound and an upper bound for it. The goal is to make the space between the bounds smaller until they eventually meet. The following mentioned lower and upper bounds apply to the half turn metric meaning that the 180° turn is seen as one turn.

In 1980, David Singmaster found a lower bound of 18 moves. This was done by looking at the number of distinct move sequences of 17 or fewer moves, which is less than the number of possible cube states. In 1995, the super flip position was found by Micheal Reid. This is a position of the cube that requires at least 20 moves to solve, so the lower bound became 20 moves. In 1980, an upper bound of 52 moves was found by Morwen Thislethwaite using a complex algorithm. The upper bound was lowered through the years by using more efficient methods. In 2010 a group of scientists (Tomas Rokicki, Herbert Kociemba and John Dethridge) finally proved that the upper bound was also 20 moves. To calculate the proof, 35 CPU years were needed. But with help of Google servers that they could use to run the calculation, they were able to complete the proof in just a few weeks [Rok14].The problem was hard to solve due to the large number of possible states, but before explaining how many states a Rubik's cube has, it is useful to have an understanding of how a Rubik's cube is built.

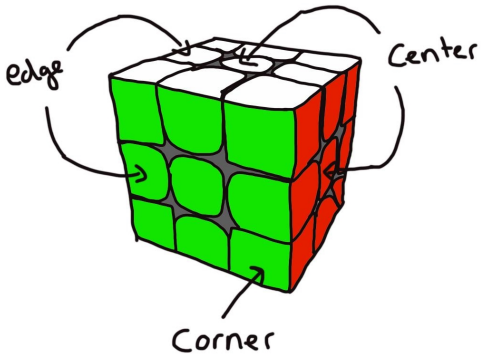

Figure 1: Rubik's cube pieces.

Figure 1 shows that a $3\times3\times3$ Rubik's cube consists of three kinds of smaller cubes, also called pieces. The first type of piece is the center piece. This piece has one visible side, so one colour, and is fixed, so the position of the center pieces is not affected by a face turn. In total, the cube has six center pieces in six different colours. The second type of piece is the edge piece: this piece has two visible sides, so two different colours, and is not fixed, so the pieces move from their position when a face turn is made. The cube has in total twelve different edge pieces. The last type of piece is the corner piece. This piece has three visible sides, so three different colours, and is also not fixed. The cube has in total eight different corner pieces. In total, a $3\times3\times3$ Rubik's cube consist of twenty-six visible pieces.

With this knowledge it is possible to calculate the number of possible positions of the cube [Ban82]. For the corner pieces there are 8! or $40,320$ permutations and every corner piece can be oriented in 3 different ways whereas the orientation of 7 corner pieces can be randomly chosen, which determines the orientation of the 8th corner piece. This makes for 3^7 combinations. This makes the total number of permutations for the corner pieces $8! \times 3^7$.
For the edge pieces there are 12! permutations and every edge piece can be oriented in 2 different ways whereas the orientations of 11 edge pieces can be randomly chosen and decide the orientation of 12th edge piece. This makes 2^{11} combinations. So the total number of permutations for the edge pieces becomes $12! \times 2^{11}$.

Then there is one last thing to consider, which is that it is impossible to swap two corner pieces or two edge piece in isolation without affecting the adjacent pieces. It is not possible to solve the cube when two of the edge pieces or corner pieces are swapped [Ban82]. It turns out that exactly half of the above counted permutations are possible. This makes the following formula:
$8! \times 3^7 \times \frac{12!}{2} \times 2^{11} = 43{,}252{,}003{,}274{,}489{,}856{,}000.$

In perspective "if we put $43.525 * 10^{19}$ cubes of 5.6 cm width, each in an other position, side by side then they bridge a distance of $2.422 * 10^{15}$ km that is about 256 light-years. Or tightly packed the cubes would covering the whole surface of the earth (land and water, 51.10^{17}cm^2) to a height of 15 meters." [Ban82].

We describe a notation for the possible turns of the cube. A Rubik's cube has six faces: front (F), upper (U), right (R), left (L), back (B) and down (D). Usually the white face is seen as the upper face of the cube and the green face is seen as the front face of the cube, see Figure 1. Each of these faces can be turned in three different ways: a 90° turn clockwise (F), a 90° turn counter clockwise (F') and a 180° turn (F2). This makes it so that the cube can be turned in 18 different ways, see Figure 2.

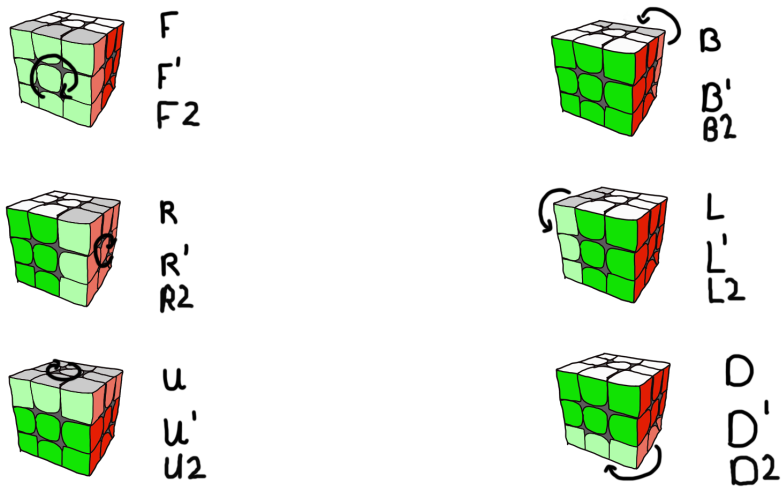

Figure 2: The 18 different Rubik's cube rotations.

Definition 1. *A graph is a pair G that consist of two sets:*

(i) A nonempty finite set V = V(G), whose elements are called vertices or nodes of G.

(ii) A possible empty subset E of V = E(G), of unordered pairs of distinct vertices called edges of G.

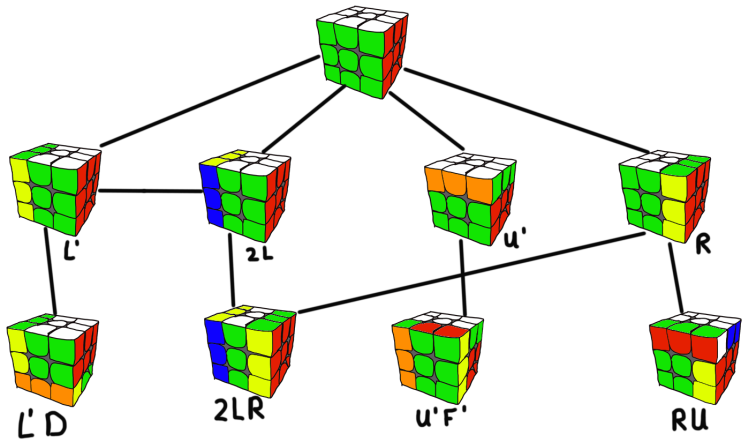

Figure 3: Small section of the state space graph.

The state space of a Rubik's cube, or all 43,252,003,274,489,856,000 states of the Rubik's cube can be modelled as graph (see Definition 1). In this case set V, the nodes, are the states of the cube and subset E, the edges, are the possible face turns of the cube to or from that state. Figure 3

shows a small section of what the graph looks like when modelling the state space of the Rubik's cube as graph.

After some experimenting with the Rubik's cube we decided to look, due to the enormous number of states of the original Rubik's cube, at the Domino cube which has a smaller state space.

2.2 Domino cube

The Domino cube is in essence a Rubik's cube with a layer less, so has dimension $2 \times 3 \times 3$. This cube was invented in 1983, also by Ernő Rubik. The first design of the cube was one White 3×3 layer with numbered dots 1 to 9 on the corner, edge and centre piece. The second layer was Black with the same numbered dots on the pieces [Hod86]. Hence the name Domino cube, see Figure 4. Nowadays the cube is also made with six different colours on each side, where the White and Yellow face are 3×3 and the Red, Blue, Green and Orange face are 2×3.

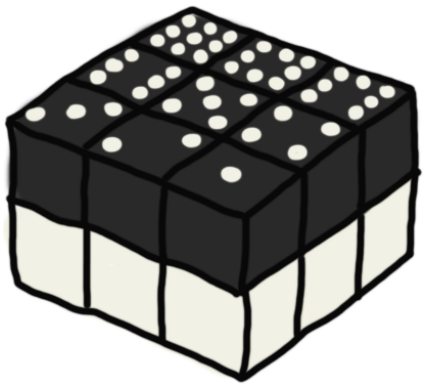

Figure 4: Domino cube.

The number of states of the Domino cube is less than for the original Rubik's cube. This is of course partly because of the missing layer, but also due to the fact that the corner pieces cannot rotate in three different ways and the edge pieces cannot rotate in two different ways. This is because F, R, L and B can only make 180° face turns. This makes it impossible to rotate corner and edge pieces, so the numbered side of the pieces is always on top [Hod86]. There are 8 different corner pieces and 8 different edge pieces. This would make for $8! \times 8!$ permutations, but this is too many. The 2×3 sides do not have center pieces. This makes that the cube can rotate in space by turning the 3×3 sides. If the White face is up, the cube can be oriented in four different ways. So this makes the total number of permutations $\frac{8! \times 8!}{4} = 406,425,600$. This is much smaller than for the original Rubik's cube.

The number of possible unique face turns is also less than with the original cube. Due to the missing layer the R turn, L turn, F turn and B turn can only be 180°, so there is only one possible turn for these faces. As mentioned before when we rotate the up layer clockwise, we will end up in the same state as when we rotate the down face counter-clockwise. The only difference is that the rotation of the whole cube is different, making it seem like it is not the same state. However, the pieces are still in the same place relative to each other, so it is the same state. This is True for the U' turn

and D turns, U and D' turns and for U2 and D2. In total, there are only 7 unique face turns while the original Rubik's cube has 18 unique face turns.

2.3 Binary decision diagrams

To represent a depth in the graph of the state space of the Domino cube, we use Binary Decision Diagrams or BDDs. A BDD can be seen as an acyclic graph, and is mostly used to represent Boolean functions. The BDD consists of multiple decision nodes, a True leaf and a False leaf. Each decision node contains a variable (var) and has a low child (ℓo) and a high child (hi). The edge from a low child represent a zero, which means False, and the edge of a high child represent a one, which means True [CG21]. A BDD works as follows: we follow the path in the BDD according to the evaluation of the variables until we reach a leaf node. If the leaf node is True it means that the path, the input of the Boolean function, is True so the Boolean function will evaluate to True. If the leaf node is False it means that the Boolean function will evaluate to False.

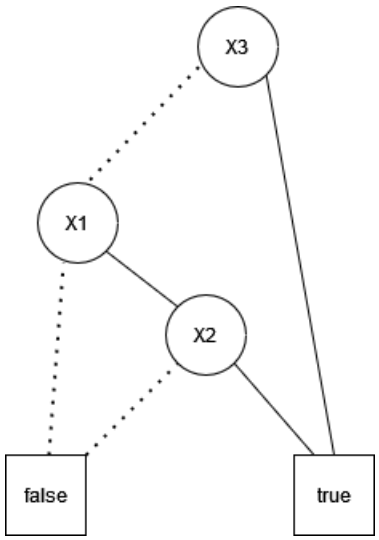

Figure 5: The BDD of $(x1 \wedge x2) \vee x3$.

Figure 5 shows a BDD example for the Boolean function $(x1 \wedge x2) \vee x3$. The dashed lines mean zero and the solid lines mean one. If the dashed line is taken it means that the variable is False and if the solid line is taken it means that the variable is True. The leaves of the BDD show where the function will evaluate to. So if $x3$ is True then the whole function, no matter what the values of $x1$ and $x2$ are, evaluates to True. But when $x3$ is False, then the values of $x1$ and $x2$ matter as to how the function will evaluate.

2.3.1 Ordering and reducing

A BDD can be constructed from a binary decision tree. Bryant [Bry92] describes that there are three reducing rules that should be performed over the decision tree of a Boolean function to obtain the BDD:

"Remove Duplicate Terminals.

Eliminate all but one terminal vertex with a given label and redirect all arcs into the eliminated vertices to the remaining one.

Remove Duplicate Nonterminals.

If nonterminal vertices u and v have $var(u) = var(v)$, $\ell o(u) = \ell o(v)$, and $hi(u) = hi(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.

Remove Redundant Tests.

If nonterminal vertex v has $\ell o(v) = hi(v)$, then eliminate v and redirect all incoming arcs to $\ell o(v)$." [Bry92].

Figure 6 shows how the BDD is constructed for the Boolean function $(x1 \wedge x2) \vee x3$ where the ordering of the variables in the tree is $x3 < x1 < x2$. The blue marked nodes are nonterminal nodes u and v where $var(u) = var(v)$, $\ell o(u) = \ell o(v)$ and $hi(u) = hi(v)$. The red marked nodes are nonterminal nodes v where $\ell o(v) = hi(v)$.

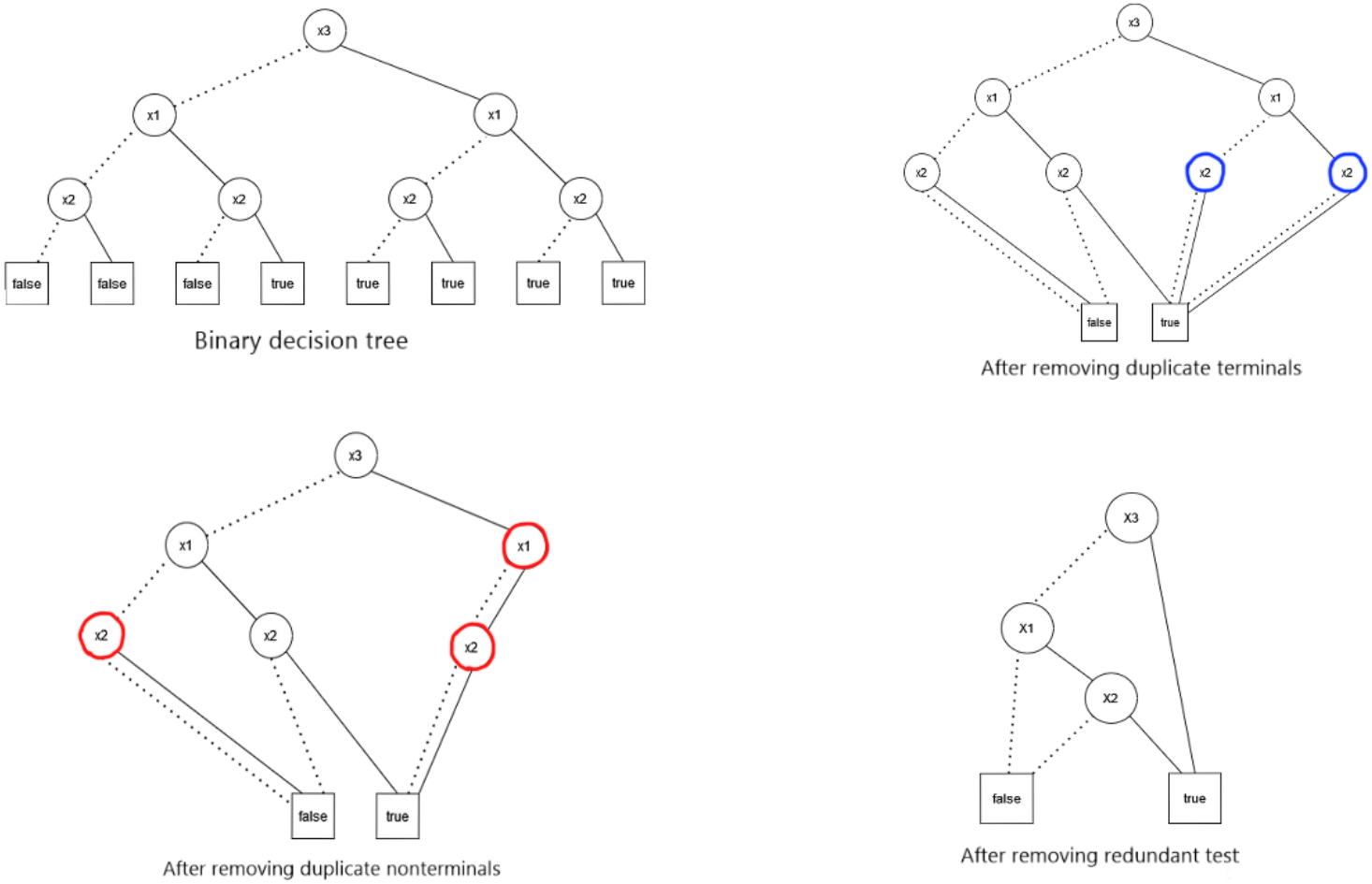

Figure 6: The construction of the BDD of $(x1 \wedge x2) \vee x3$.

3 Representing the state space of the Domino cube

In this section we discuss the method that is used for this research. For this research a program [vW22] was written in C++, that executes what is described below.

Due to the number of states of a Domino cube, significantly less than the original Rubik's cube but nonetheless a large number, the challenge was to store the states in memory as compact as possible.

3.1 Representing a Domino cube

The first challenge was to represent the Domino cube as compact as possible, to use as little memory as possible per state. As said before the center piece is fixed, so in essence it is unnecessary to keep track of the center piece because it is always in the same position. Because of this, we only have to keep track of where the edge and corner pieces are. One way to do this is to store the colours for each face: this means that we have to save eight colours for the up and down faces and six colours for the other faces of the cube. The Domino cube has six colours, so we need three bits to represent these colours:

- White equals 000
- Green equals 001
- Red equals 010
- Orange equals 011
- Blue equals 100
- Yellow equals 101

To represent an up or down face of the cube we get a bit string of $8 \times 3 = 24$ bits. For example:

$$000\ 101\ 000\ 101\ 000\ 000\ 101\ 101$$

which represents:

$$\begin{array}{ccc} W & Y & W \\ Y & & Y \\ Y & W & W \end{array}$$

To represent a left, right, front or back face, a bit string of $6 \times 3 = 18$ bits is needed. In total, we have to store $4 \times 18 + 2 \times 24 = 120$ bits for one domino cube state. But this is not the most compact way to store a Domino cube. Because the edge pieces and corner pieces cannot rotate, the same colour is always on top. This means that when we number the pieces, we do not have to keep track in which way the pieces are rotated but only in which place they are. The Domino cube has 8 corner pieces and also 8 edge pieces (we can still ignore the center pieces) which means that there are 16 different pieces in total [Hod86]. So we need at most 4 bits for every piece. So to store one Domino state we only need a bit string of $16 \times 4 = 64$ bits. So the initial top face would be:

$$0000\ 0001\ 0010\ 0100\ 0111\ 0110\ 0101\ 0011$$

which represents:

```
0 1 2
3   4
5 6 7
```

3.1.1 Simulating rotations

The next part of representing the cube is to simulate the face turns. When explaining the face turns in binary notation it is useful to see the Domino cube as a folded-out 2D map in Figure 7. The top of each edge and corner piece is numbered, left-to-right, top-to-bottom. The sides of the edge and corner pieces are indicated with the colour Black or White.

```
          0  1  2
          3  W  4
          5  6  7
W W W W W W W W W W W W
B B B B B B B B B B B B
         13 14 15
         11 B  12
          8  9  10
```

upper face : 0000 0001 0010 0100 0111 0110 0101 0011
down face : 1101 1110 1111 1100 1010 1001 1000 1011

Figure 7: Solved Domino cube state.

Figure 8 is an unfolded map of the solved Domino cube where the right face is marked in blue. This face will turn 180 degrees. Also the neighbouring sides (marked red) will move with this face turn rotation. The result of the R face turn is seen in the right unfolded map of the cube in Figure 8.

```
          0  1  2
          3  W  4
          5  6  7
W W W W W W W W W W W W
B B B B B B B B B B B B
         13 14 15
         11 B  12
          8  9  10
```

Solved state.

```
          0  1  15
          3  W  12
          5  6  10
W W W W W B  B  B  B  B  W W
B B B B B W  W  W  W  W  B B
         13 14 2
         11 B  4
          8  9  7
```

State after R turn.

Figure 8: R face turn.

In Figure 9 the R face turn is represented in binary, where the first two bit strings represent the solved state of the Domino cube and the last two bit strings represent the state of the Domino cube after the R turn. We only keep track of the up side of every corner and edge piece so only the up face and down face are represented in binary. With clever use of bit masks and bit shifting we can replace the red marked bits in the right place and thus simulate a R turn in the binary string.

upper face : 0000 0001 0010 0100 0111 0110 0101 0011
down face : 1101 1110 1111 1100 1010 1001 1000 1011
upper face : 0000 0001 1111 1100 1010 0110 0101 0011
down face : 1101 1110 0010 0100 0111 1001 1000 1011

Figure 9: R face turn in binary.

The left, back and front turns are done in the same way as the right turn where the right turn is mirrored from the left turn and the front turn is mirrored from the back turn. The upper turns are done in a slightly different way. These turns are done with bit rotation. For example, bit shifting to the left would cause the bits on the left end to fall off, but with bit rotation these bits would be put back at the right end. Figure 10 shows a 90 degrees clockwise U turn.

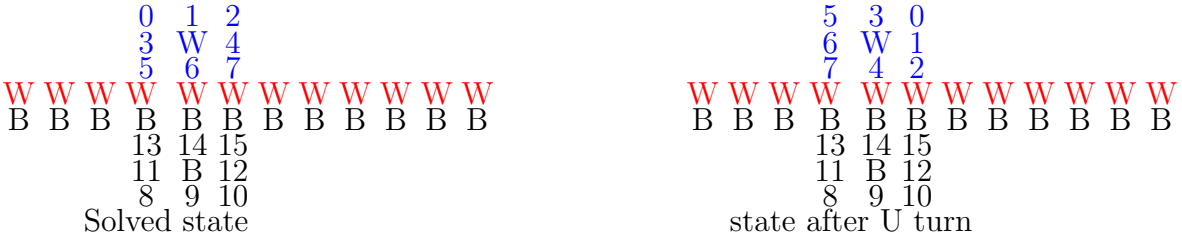

Figure 10: U face turn

upper face : 0000 0001 0010 0100 0111 0110 0101 0011
down face : 1101 1110 1111 1100 1010 1001 1000 1011
upper face : 0101 0011 0000 0001 0010 0100 0111 0110
down face : 1101 1110 1111 1100 1010 1001 1000 1011

Figure 11: U face turn in binary

Figure 11 shows the U turn in binary where first two strings are the solved state of the Domino cube and the last two strings are the state of the Domino cube after the U turn. To perform U' and U2 we simply repeat the U turn three or two times. The D turn is not implemented because we do not need the turn to generate all states.

3.2 Generating states

The next part is about how to efficiently generate states of the cube. Because of the large number of states, it is necessary to make the search of whether the state has already been generated as efficient as possible.

When we model the Domino cube states as a graph we do a breadth-first search, and we start from the solved position. This is called depth zero. From the solved position it is possible to generate depth one. Depth one contains all possible states that can be generated from the solved state with the 7 different face turns. From depth one it is possible to generate depth two and so on until a depth is found where no new states can be generated. This means that all states have been mapped out. Starting at depth two it is important to check the above two depths to check whether the reached state has already been generated. It is unnecessary to check more than two depths above the current depth because a 180 degree turn can also be reached by two times the same 90 degree turn. Above three depths it is not possible to find the same state because we cannot create these states with only one turn.

3.2.1 Saving depths of the graph in BDDs

The main advantage of using a binary decision diagram to store the states at a given depth is the search time for checking if a state has already been generated. In comparison, when using an array to store the states, the program has to go through each element of the array to check if the state already exists. This will take linear time. With big numbers that can take much time. In this case, the numbers will grow with each depth. This will slow the program down when reaching bigger numbers.

When using a BDD to store states at a depth it is pretty fast, because when giving a state to the BDD it will return True when it is already in the BDD at the depth and False if it is not. It does not have to go through every state at that depth to check whether the state is there or not. This will take a constant time, namely the length of the bit string. The second advantage is that it is more space efficient in comparison to a binary decision tree, since fever nodes are needed for a BDD.

3.3 Implementation of the BDD

In Section 3.1, the binary representation of the cube is explained (see Figure 7). So if we combine the two face bit strings, one longer bit string remains. For the solved state it would be:

0000 0001 0010 0100 0111 0110 0101 0011 1101 1110 1111 1100 1010 1001 1000 1011

For depth zero there is only one state, the solved state. The BDD at this depth looks like a line where only one path results in the True leaf.

3.3.1 Adding states to the BDD

One problem with generating the BDD per depth is that we do not know in advance what the full Boolean function at the depth will be. One thing that can be done is to store the states at the depth in an array, construct the Boolean function from all states and then make a BDD. This is not very space efficient because we will obtain very large arrays. So ideally we want to construct the BDD simultaneously with the generation of the states. Knuth has a solution for this in his book *The Art of Computer Programming* [Knu09]. We have to see the state that we want to add as a second BDD, and then we can combine those two BDDs into one.

According to Knuth we can construct a new BDD for any binary logical operation on the two sets represented by the two BDDs. In our case the logical operation is always $\vee$. One BDD is representing a single state, with one single path that evaluates to True and and the other is the existing BDD containing the states that have already been found. Figure 12 shows a simplified example. The first picture shows the two BDDs and the second is where the two BDDs are combined. The idea is that we start in the first node of the single state BDD and then we have to connect the side branches that would evaluate to False in the single state BDD to the corresponding node in the already existing BDD. Finding the corresponding node is relatively simple: from the original BDD, we look at where that side (which in the single state BDD would go to False) will go to and connect the side from the single state BDD to that node. The new beginning of the BDD is the start of the single state BDD. Every node that is not reachable from the newly constructed BDD will be removed. In Figure 12 these nodes are dotted. After that the BDD is not yet reduced [Knu09]. So the reduction rules mentioned earlier are applied to the BDD. Figure 13 shows

the result after the reduction rules. With this method it is possible to reduce the nodes in between adding states.

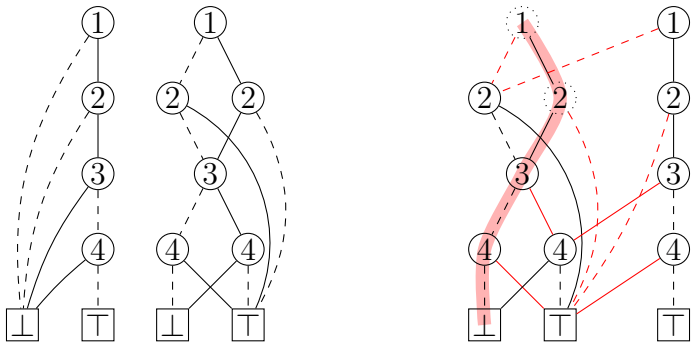

Figure 12: Adding state to existing BDD. Two BDD, one representing a single state, with path π. The copy of π can be found in the other BDD, and is marked red in the construction.

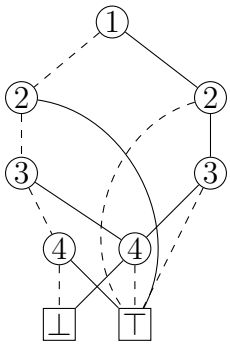

Figure 13: Result after adding the state to BDD.

3.3.2 Naively adding states to the BDD

We cannot add states to a reduced BDD as we would do to a tree. The reason for this can be explained using a simple example. If we have a BDD of the states 01101 and 11001 and we would reduce the nodes in between adding states, the BDD would look like Figure 14. If we then want to add the state 01111, then 11011 becomes True as well. See Figure 15. But this is not a state that we want to result into True. The method described in Section 3.3.1 prevents this problem.

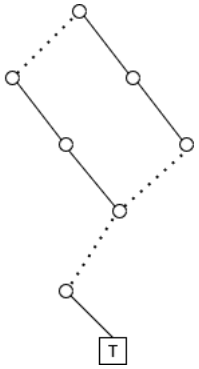

Figure 14: BDD of state 01101 and 11001.

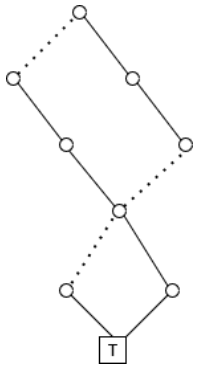

Figure 15: BDD after adding state 01111.

3.3.3 Reducing nodes

For removing duplicate nonterminal nodes in between adding states, we make use of a table. This is a two-dimensional array, that uses the index of the node to keep track of which nodes are in a tree with the same index. The index of a node represents in which depth of the BDD the node is. The root node of the BDD has index zero. In this table we work bottom up, we start from the highest index and we hash the nodes with the value of their children. If a node has the same children, they get the same key and therefore are put in the same bucket.

We simply have to keep track of every key made for each index-level. Then for every key we get all nodes with the same key (these are all nodes that have the same high and low child so only one node has to remain) and redirect all parents of these nodes to the first node in the list. After that we delete the rest, and go one index-level down. We do this until we end up in an index-level where every key has exactly one node. This means that every node is unique and none of the nodes can be reduced. It is important to update the table which means that the node removed from the BDD must also be removed from the table.

To remove redundant tests, nodes where the low and high child are the same, we also use the table. But these nodes are not removed until all states have been added to the BDD. Again we work bottom up starting from the highest index-level. We check for every node if they have the same children. If this is the case, we redirect the parent to the child and remove the node, and than go one index-level down. This is done until the root index-level is reached.

3.3.4 Different encodings of the Domino cube

The way in which the variables are ordered has an effect on how many nodes there are in a BDD, so for the experiment we are going to test different encodings of the Domino cube. The standard method is shown on the left side of Figure 16, where every piece of the cube has its own number. The second encoding is to give the edge and corner pieces the same number, because an edge piece cannot be in the position of a corner piece place or vice versa. We can therefore give an edge piece and a corner piece the same number. This encoding can be seen on the right side of Figure 16. In this encoding one fewer bit is needed for each number to represent the Domino cube, so the total binary string length becomes 48 bits. This 48-bit encoding will be called *encoding two*, whereas the previously mentioned 64-bit encoding is called *encoding one*.

The order of the edge and corner pieces are in the binary string can also be changed. The standard is to alternated the corner pieces with the edge pieces, but we can also do all corner pieces first and then all the

edge pieces, or edge pieces alternated with corner pieces, or first all edge pieces and than all corner pieces. These all make for different encodings of the cube and may result in smaller BDDs, as we shall see.

0	1	2
3		4
5	6	7

0	0	1
1		2
2	3	3

13	14	15
11		12
8	9	10

6	7	7
5		6
4	4	5

Figure 16: The left side shows encoding one and the right side shows encoding two.

4 Experiments

In this section we will show the results of the experiments and discuss the results. For the results below, we used a slightly different method than the one described in the previous section. The difference is in adding the states: the algorithm Knuth describes, reducing the BDD after addition of each new state, became very slow when the BDD became bigger. So for adding states we just build the normal decision tree and then reduce the nodes. By building the decision tree in the normal way we cannot reduce the nodes in between the states, because this would affect the outcome of the BDD. This is explained in Section 3.3.2. Unfortunately this means that the whole tree must be built before it can be reduced it is not really space efficient. Because of this, at one point the tree did not fit in memory and the program [vW22] stopped.

For all the experiments we used the data science lab of LIACS where the program was run on mithril, which is a computer with 1 TB of RAM and 64 cores / 128 threads.

4.1 Encoding one

The result of the first encoding from Figure 16 is shown in Table 1. For each depth, Table 1 shows how many nodes are in the BDD before and after it is reduced. In the encoding the corner pieces are alternated with edge pieces. This is the standard ordering of the pieces.

		Not reduced	Reduced
depth	# states	# nodes in BDD	# nodes in BDD
0	1	64	64
1	7	427	317
2	34	1,872	1,319
3	162	8,295	5,101
4	745	35,852	19,702
5	3,442	154,733	73,252
6	15,639	658,342	261,619
7	70,438	2,760,825	877,081
8	313,668	11,331,487	2,676,075
9	1,381,005	45,436,444	7,475,144
10	5,998,451	176,920,885	20,257,821
11	25,349,313	656,403,373	55,415,450

Table 1: Number of states and nodes with encoding one.

Figure 17 represents the data from Table 1 in a graph, where on the horizontal-axis the number of states is shown and on the vertical-axis the number of nodes in the BDD. It is clearly visible that reducing nodes makes a tremendous difference towards the size of the BDD at each depth when alternating the pieces. Also, when using encoding one, the program will stop at depth 12.

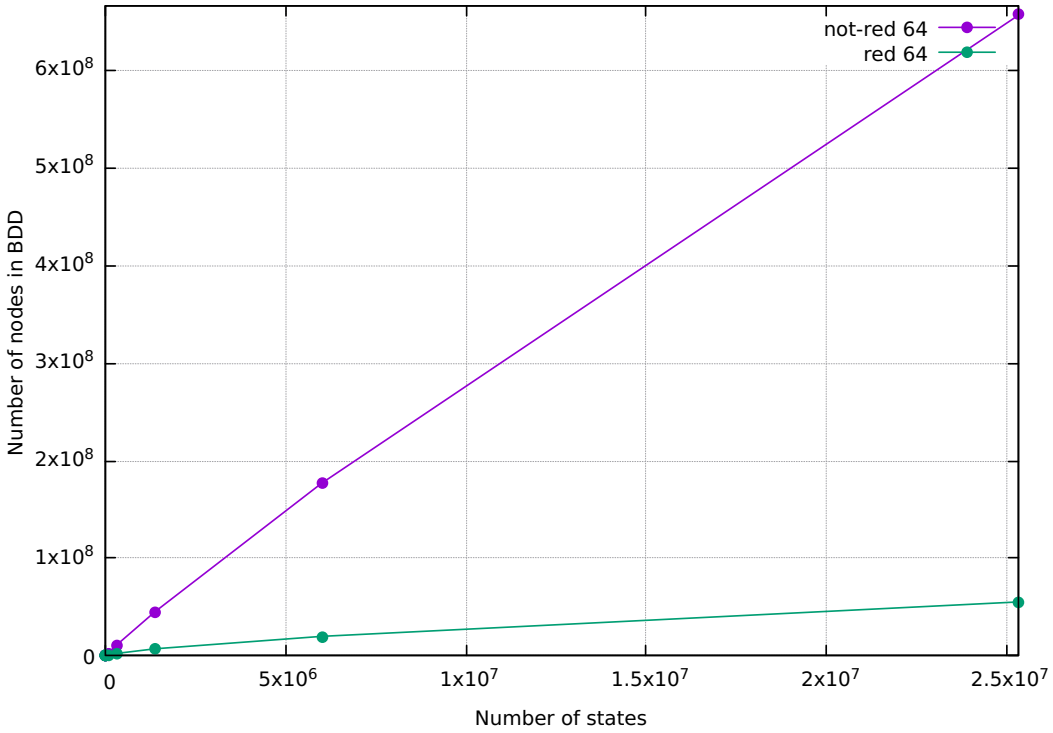

Figure 17: Number of nodes with encoding one.

4.1.1 Switching corners and edges for encoding one

The ordering of the variables in the BDD matters for the final size of the BDD. Some variable orderings in the BDD result in smaller BDD sizes than others. A different order of the corner and edge pieces in the binary string changes the ordering of the variables in the BDD. As mentioned in Section 3.3.4, the order of corner and edge pieces can be changed. The following orderings of the edge and corner pieces have been tried with encoding one:

- $C_1E_1C_2E_2C_3E_3C_4E_4C_5E_5C_6E_6C_7E_7C_8E_8$ is alternating corners with edges.
- $E_1C_1E_2C_2E_3C_3E_4C_4E_5C_5E_6C_6E_7C_7E_8C_8$ is alternating edges with corners.
- $C_1C_2C_3C_4C_5C_6C_7C_8E_1E_2E_3E_4E_5E_6E_7E_8$ is all corners first and than all edges.
- $E_1E_2E_3E_4E_5E_6E_7E_8C_1C_2C_3C_4C_5C_6C_7C_8$ is all edges first and than all corners.

Table 2 shows the results for the different orders of encoding one up to depth nine. Figure 2 shows the results in a graph. From this graph we can see that when the corners are first or the edges are first, this results in smaller BDDs at each depth. Also, the difference between the two alternating orderings is not very large. The same goes for the two non-alternating orderings.

		not reduced	c alternated e	e alternated c	corners first	edges first
depth	# states	# nodes	# nodes	# nodes	# nodes	# nodes
0	1	64	64	64	64	64
1	7	427	317	329	384	405
2	34	1,872	1,319	1,402	1,554	1,604
3	162	8,295	5,101	3,987	5,970	6,176
4	745	35,852	19,702	20,877	21,466	22,835
5	3.442	154.733	73.252	77.230	69.875	73.347
6	15,639	658,342	261,619	272,967	200,481	203,157
7	70,438	2,760,825	877,081	903,808	507,923	505,068
8	313,668	11,331,487	2,676,075	2,731,945	1,224,926	1,241,997
9	1,381,005	45,436,444	7,475,144	7,577,010	3,379,719	3,435,015

Table 2: Switching edge and corner pieces encoding one.

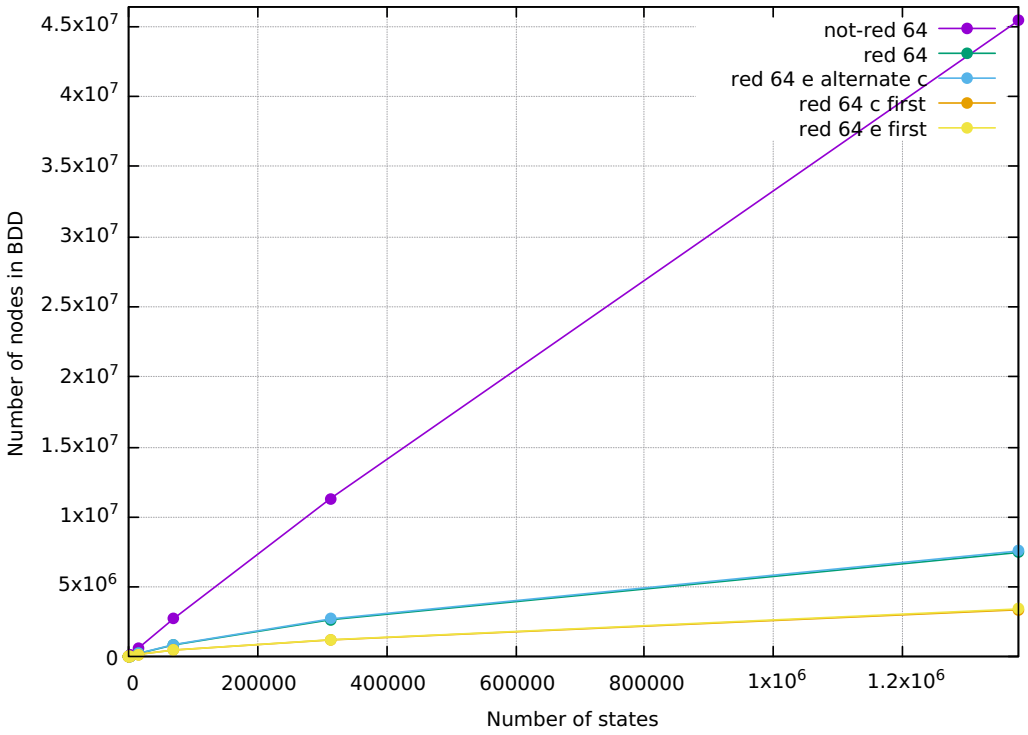

Figure 18: Number of nodes with encoding one.

4.2 Encoding two

For this experiment we used the second encoding system as shown in Figure 16, where the edge pieces and corner pieces both use the same numbers. The bit string representing the states in this encoding is only 48 bits long. The ordering of the corner pieces and edge pieces is the same as when using encoding one where

corner pieces alternate with edge pieces. The results are shown in Table 3 where again the number of nodes in the unreduced and reduced BDDs are shown. Figure 19 shows the same results in a graph. With this encoding, the program stops at depth 13. These results again show that reducing the BDD makes a huge difference as the number of states increases.

		Not reduced	Reduced
depth	# states	# nodes in BDD	# nodes in BDD
0	1	48	48
1	7	317	234
2	34	1,386	969
3	162	6,142	3,744
4	745	26,530	14,412
5	3,442	114,417	53,312
6	15,639	486,400	189,261
7	70,438	2,038,125	629,853
8	313,668	8,355,295	1,907,815
9	1,381,005	33,451,593	5,318,425
10	5,998,451	129,994,942	14,563,952
11	25,349,313	481,058,723	40,398,277
12	100,909,475	1,635,086,535	101,942,026

Table 3: Number of states and nodes with encoding two.

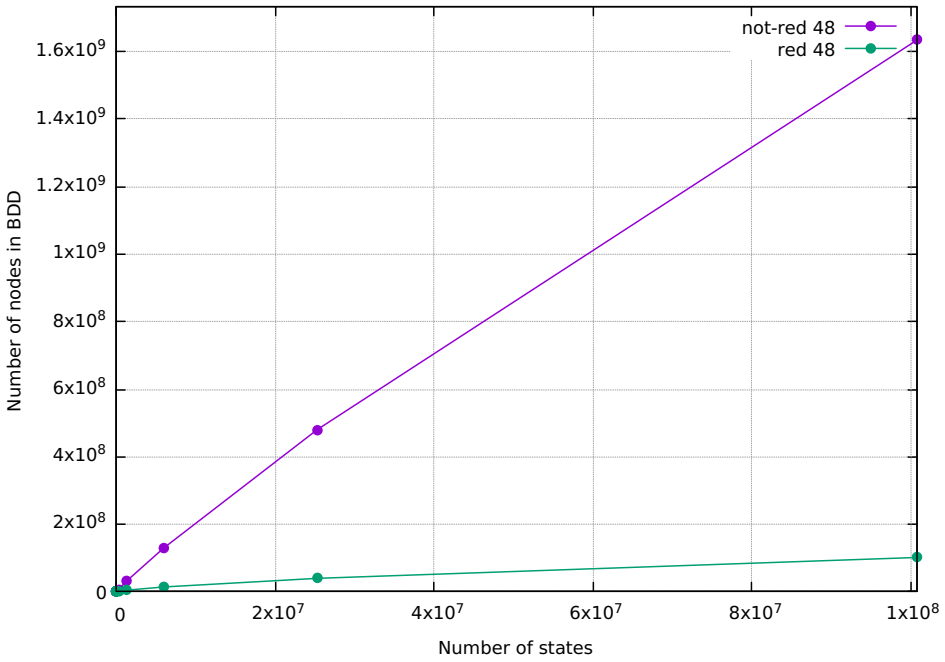

Figure 19: Number of nodes with encoding two.

4.2.1 Switching corners and edges for encoding two

For this experiment, we used the same orderings of edge and corner pieces as mentioned in Section 4.1.1, but using encoding two. Table 4 and Figure 20 show the results. Just like when using encoding one, when the pieces are not alternating this will reduce the number of nodes more than when alternating the corner and edge pieces. Again, the difference between the two alternating orderings is quite small, as is the difference between the two non-alternating orderings (and almost invisible in the graph).

		not reduced	c alternated e	e alternated c	corners first	edges first
depth	# states	#nodes	#nodes	#nodes	#nodes	#nodes
0	1	48	48	48	48	48
1	7	317	234	242	285	301
2	34	1,386	969	1,034	1,149	1,187
3	162	6,142	3,744	3,987	4,407	4,558
4	745	26,530	14,412	15,310	15,773	16,818
5	3,442	114,417	53,312	56,330	51,172	53,878
6	15,639	486,400	189,261	197,863	146,842	148,604
7	70,438	2,038,125	629,853	650,412	373,884	369,109
8	313,668	8,355,295	1,907,815	1,950,542	907,284	913,460
9	1,381,005	33,451,593	5,318,425	5,399,850	2,496,305	2,535,171
10	5,998,451	129,994,942	14,563,952	14,823,452	7,598,450	7,558,549

Table 4: Switching edge and corner pieces encoding two.

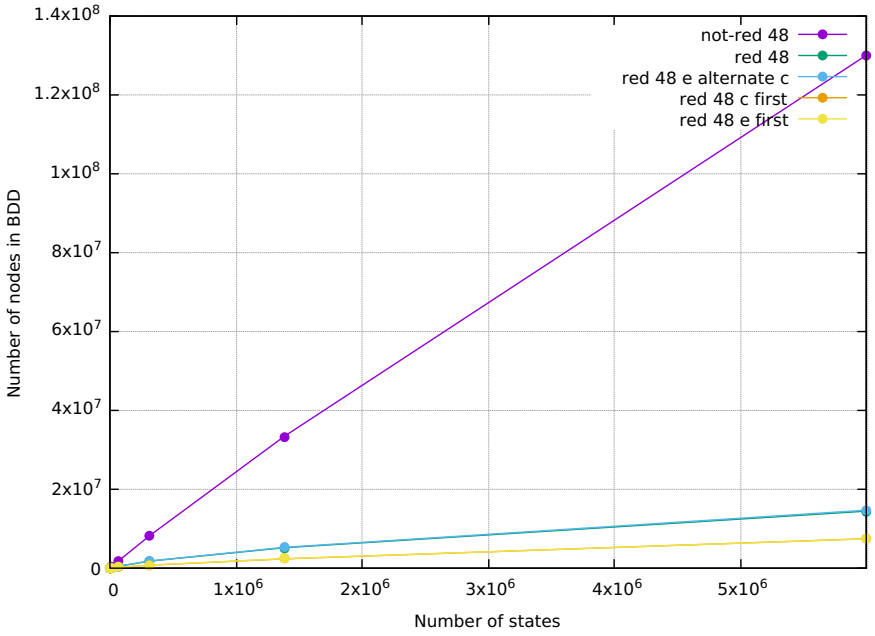

Figure 20: Number of nodes with encoding two.

4.3 Comparing encoding one and two

Figure 21 shows the difference in size of the BDDs before reducing the nodes between encoding one and two. Figure 22 shows the difference in size of the BDDs after reducing the nodes with all different orderings of the edge and corner pieces between encoding one and two. From this we can see that encoding two ensures smaller BDDs.

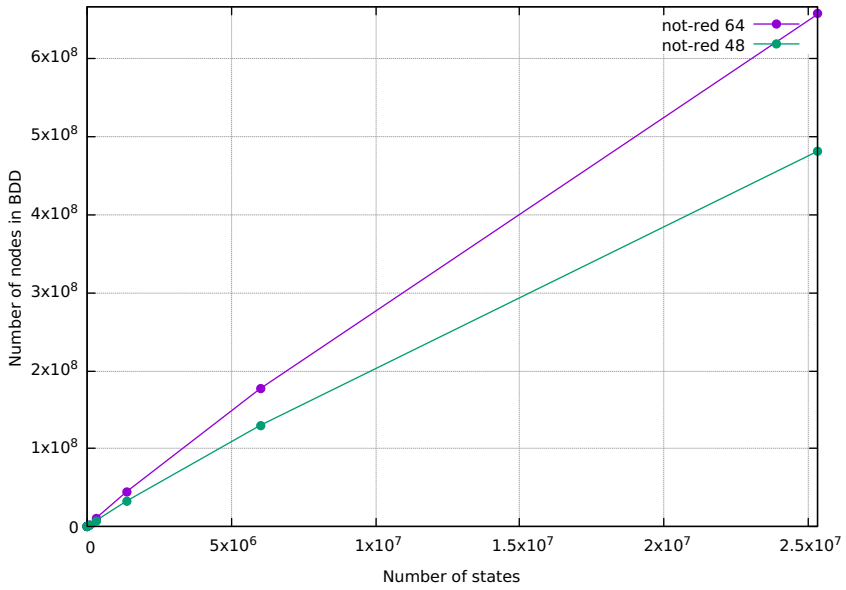

Figure 21: Number of nodes when not reducing.

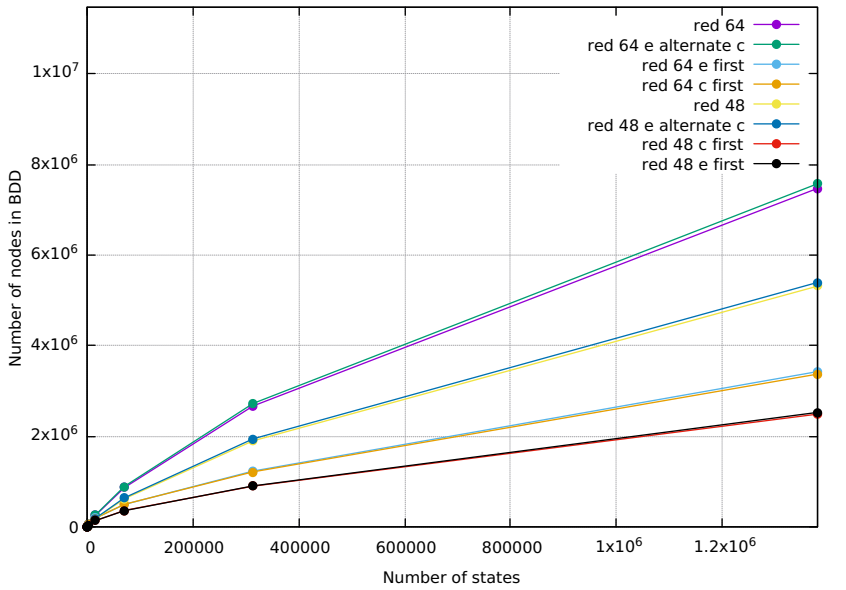

Figure 22: Number of nodes when reducing.

5 Conclusions and Further Research

In this section we will conclude the thesis and discuss the Further Research for this project.

5.1 Conclusions

The goal of this research was to find out if using different encodings of the Domino cube has effect on the size of the BDDs. Encoding two, the encoding where edge and corner pieces have the same numbers, allows the program to go one depth further than when using encoding one. But this is mainly because the tree stays smaller with encoding two than with encoding one, see Figure 21. This has a simple explanation: because encoding two uses fewer bits per state, the tree is always smaller and it takes longer for memory to fill up. If we would look exclusively at the reduced BDDs with the two encodings and different orderings of the edge and corner pieces, we could conclude that encoding the Domino cube differently does affect the size of the BDDs significantly. See Figure 22. Of course, encoding two is more memory-efficient than encoding one simply because we use fewer bits to represent the same data. If we compare the alternated and separated orderings in encoding two, the latter results in even smaller BDDs. So we can say that the ordering of the pieces also affects the number of nodes in the final BDDs, and that in this case encoding two combined with the separated ordering results in the smallest BDDs.

5.2 Further Research

Unfortunately the program that we used was not able to reduce the nodes while adding the states. The current implementation becomes too slow at large numbers of states. So for further research the method described in Section 3.3.1 could be optimized in such a way that adding the states becomes faster. If this is done, this means that it would be possible to reduce in between adding states and then it would not be necessary to build the full decision tree first. Presumably, this means that the state space of the Domino cube would fit in memory and then we could find the shortest path to the solved state for any scramble of the Domino cube.

References

[Ban82] Christoph Bandelow. *Inside Rubik's Cube and Beyond*, chapter The Mathematical Model, pages 26–55. Birkhäuser Boston, Boston, MA, 1982.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[CG21] Julien Clément and Antoine Genitrini. Binary decision diagrams: From tree compaction to sampling. In *Latin American Symposium on Theoretical Informatics*, Lecuture Notes in Computer Science, pages 571–583. Springer, 2021.

[Hod86] J.P.E Hodgson. Interactive problem solving. *ACM SIGART Bulletin*, (98):22–24, 1986.

[Knu09] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 2009. Fascicle 4.1.

[McG16] Ryan T. McGregor. *Finding God's Number for the Rubik's Cube Using Computational Analysis*. PhD thesis, Hofstra University, 2016.

[Rok14] Tomas Rokicki. *Thirty years of computer cubing: The search for God's number*, volume 100. Spectrum, 2014.

[vG10] Rik van Grol. The Quest for God's Number. *Math Horizons*, 18(2):10–13, 2010.

[vW22] F.T. van Wetten. 2x3x3_solver. https://git.liacs.nl/s2269651/2x3x3_solver, 2022.