Seeking the Diameter of Rubik's Cube Using Symbolic Methods

Benjamin Smith

March 17, 2010

Abstract

The Rubik's Cube is a puzzle with a very large state space. Using explicit state-space generation techniques, a problem this size is far beyond the capacity of any existing computer. This project completed the generation of the Rubik's Cube state space in a short amount of time using inexpensive computer hardware and symbolic state space generation and storage techniques, which are vastly more efficient than using explicit breadth-first search. The state space generation used no simplifying concessions such as ignoring the orientation of center squares. The symbolic techniques used for this project can also generate and store the distance to the goal state for every reachable state. The ultimate aim of this project is to determine the diameter of Rubik's Cube. If the diameter is greater than the previously best known lower-bound, this project will also discover new permutations of the cube that require more moves to solve.

1 Introduction

The Rubik's Cube is claimed by its manufacturer to be both the "World's #1 Puzzle," and the "Ultimate Brain Teaser" [11]. Generating the state space of this puzzle requires using symbolic methods since the state space is so large. The method used in this project was to design a Petri net model of the cube, and use the SMART2 software package to generate the reachable state space. The SMART and SMART2 software packages use multi-way decision diagrams, or MDDs, to store the state space [4]. The saturation algorithm is used to efficiently generate the state space using MDDs both as storage for states and to store the next-state function.

The Rubik's Cube is an excellent problem for pushing the boundaries of the SMART software package. Its size and complexity reveal limitations that remain invisible in simpler problems. The many subproblems available with the Rubik's Cube allow for learning what works on a small scale so that large-scale computations can be done more efficiently. It is likely that finding a feasible model would have been impossible if solvable subproblems didn't exist.

The remainder of this paper is organized as follows: Section 2 contains the problem description discussing the specifics and scale of the Rubik's Cube problem. Section 3 contains the results of related work, and the current status of finding the Rubik's Cube diameter.

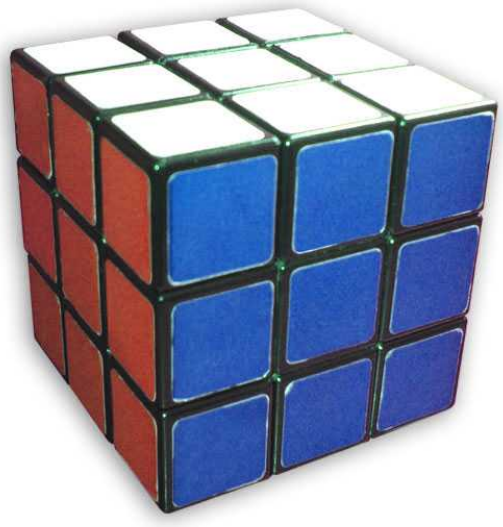
Section 4 contains a description of the implementation and design decisions for the Rubik's Cube model. Section 5 has the results of relevant experiments. Section 6 describes directions for further study, and Section 7 contains conclusions.

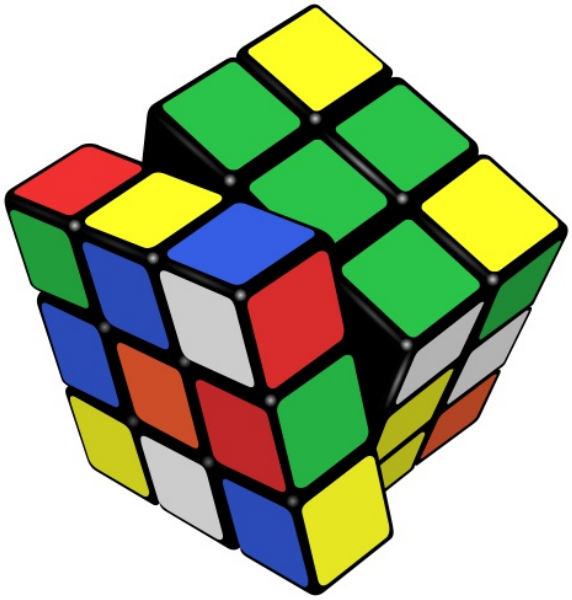
2 Problem Description

The problem of determining how many twists are required in the worst case to solve the Rubik's Cube puzzle is unsolved. This number is known as its diameter. The solution to this problem has failed to be found by all attempts using explicit searching. The state space of the puzzle is too large to allow any existing computer to store or generate each state individually. The problem of generating the state space of the Rubik's Cube is interesting, and generating such a large state space demonstrates the power of using decision diagrams and the saturation algorithm. The difference between the capabilities of symbolic methods versus explicit methods is extraordinary, and hopefully success in this project will bring these techniques some well-deserved attention. This project harnesses the popularity of the Rubik's Cube puzzle, especially among computer scientists and mathematicians, to generate awareness of symbolic methods.

2.1 Description of the Cube

The Rubik's Cube puzzle is a 3-dimensional cube with each side a different color. As seen in the diagram below, the cube is divided into 3x3x3 subcubes which are referred to as cubelets.





A side consists of nine cubelets, and can be rotated in 90 degree increments. As seen above, the center cubelet on a side rotates, but its position relative to the other center cubelets never changes. In fact, if you dismantle the puzzle completely you find that the six center squares are all part of the same piece as seen below.

Corner cubelets have three different colored squares, and can be oriented three different ways. Edge cubelets have two different colored squares, and can be oriented in two ways. An edge cubelet can never be twisted to become a corner cubelet, and vice versa. This means that if either the edges or corners are ignored, the remaining cubelets can be viewed as their own sub-puzzle that is easier to solve. Since most available Rubik's Cubes have solid colors for the squares, rotating the center square does not have a visual impact on the puzzle itself. The puzzle is considered solved, even though the orientation of the center squares may be different than the original state. There do exist Rubik's Cubes with pictures on each side instead of single-colored squares. Solving these puzzles requires orienting the center square

correctly, giving a considerably larger state space. The cube problem that includes center-square orientation has generally been ignored, probably due to the great difference in the state space size when viewed through an explicit mindset.

Each twist or turn of a side is considered a move for counting. The same state space is generated if 180 degree half-turns are counted as a single move, instead of counting them as two quarter-turns. If half-turns are counted as a single move, this is known as the face-turn metric because any orientation of a cube face can be reached in one move. If half-turns are disallowed and counted as two quarter turns, this is known as the quarter-turn metric.

2.2 Known Sizes of Reachable State Spaces

The number of arrangements of the pieces of the Rubik's Cube can be calculated easily. It is also quite simple to determine which of the arrangements are reachable through legal sequences of moves. Some arrangements can only be achieved through dismantling the cube and reassembling it manually. Parity means that not all states are reachable through legal moves.

The number of possible states the puzzle can reach is very large, but the number is easy to calculate. The eight corner cubelets can each be oriented three different ways. Each of the corners can likewise be moved to any of the eight corners of the cube. This leads to $8! * 3^8 =$ 264,539,520 possible states for the corners, however the corners can only reach a third of these states through twisting from the solved state. So, the total number of combinations reachable without taking apart the puzzle as shown above is $8! * 3^7 =$ 88,179,840. The 12 edge cubelets can each be oriented two ways. Each of the edges can be moved to any of the twelve edge positions of the cube. This leads to $12! * 2^{12} =$ 1,961,990,553,600 possible states for the edges. Like the corners, some of the states cannot be reached through twists from the solved state. Only half of these states are reachable, so $12! * 2^{11} =$ 980,995,276,800 states for the edges are reachable. In total for the entire cube, there are $8! * 3^7 * 12! * 2^{10} = 4.3 * 10^{19}$ states [13][7].

Puzzle	Number of States
Corners Only	88,179,840
Edges Only	980,995,276,800
Corners and Edges	43,252,003,274,489,856,000
Cube with Center Orientation	88,580,102,706,155,225,088,000

3 Related Work

3.1 Iterative Deepening A* Search to Prove Lower Bounds

If a scrambled position of the Rubik's cube is thought to require a large number of moves to solve, proving this fact produces a lower bound on the diameter of the cube. Proving that a very scrambled position requires a large number of moves is a difficult problem on its own. The A* search algorithm has the property that a good heuristic can allow this proof to be done with less work [6]. The search is guided by a count of how many moves have been made, combined with a heuristic estimating how many moves remain to reach the goal.

This combination allows the algorithm to always follow the most promising path first. If the heuristic is admissible, A* search not only guarantees that it finds the shortest path, it also finds it visiting fewer nodes than any other algorithm using that heuristic. Being an admissible heuristic for the A* search requires that the estimate of how many moves remain before reaching the goal state is never over-estimated. A better heuristic produces an estimate that is as close to the true value as possible, and can be calculated quickly. For the Rubik's Cube problem, many good and admissible heuristics are available. Using the number of moves required to solve the corner-only subproblem is one good choice. The edges-only subproblem is a little too large to be used as there are nearly a trillion states. Using this heuristic would likely be more difficult than the search itself. So, in the past, the corners heuristic has been combined with a heuristic for solving half the edges [6]. These heuristics are admissible because it is at least as difficult to solve the entire cube as it is to solve the sub-problems. They therefore can't overestimate the distance to the goal state.

The basic A* search algorithm can require an exponential amount of memory, so it needs to be adjusted to be useful for large search problems like this one. For this reason, iterative deepening A* or IDA* search can be used [6]. This algorithm uses a depth-first strategy instead of breadth-first with the same A* heuristic method. The search is first done at a depth of 1, then 2, and so on until a solution is found. The run time of IDA* is only slightly longer, but the memory consumed is negligible.

3.2 "Superflip" Twenty Face-Moves Lower-Bound Result

One position is known to require twenty face-turns to solve, under the model where center square orientations are ignored. This position is known as "superflip", and is one of a handful of positions that are at the maximum known distance from the solved state [8]. Superflip is the solved state, with each of the edge orientations reversed. Since the known lower bound for the Rubik's Cube is 20 when center square orientations are ignored, this also represents a lower bound when center squares are considered. There are 2048 orientations of the center squares reachable for every Cube permutation, it is a safe prediction that the diameter of the cube is larger if center square orientations are part of the problem.

3.3 Reid 26 Quarter-Turns Lower-Bound Result

For any challenging position, the optimal sequence of moves required in the face-turn metric is rarely also optimal in the quarter-turn metric. A position called "superflip composed with four spot" has been found to require 26 quarter turns to solve [9]. This position was also proven to be locally maximal, meaning that any quarter-turn from this position goes toward the solved state [9].

3.4 Rokicki Twenty-Two Face-moves Suffice Upper-Bound Result

Upper bounds are proven by showing that all possible positions can be solved in at most a certain number of moves. This means that using a limited number of moves, every position can be transformed into one of a set of positions with known upper-bounds. If more positions are able to be proven solvable in a smaller number of moves, the upper bound can be

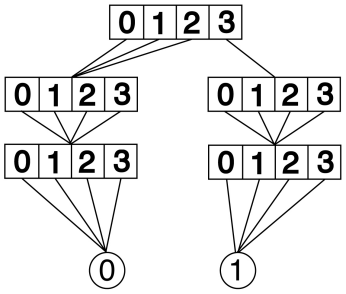


Figure 1: A 3-level MDD encoding a set where the first inputs are 3. Since the number of paths in the structure can be exponential with respect to the number of levels, very large sets can be represented with few nodes.

improved. Rockiki used about 50 CPU-years to eventually prove that all Rubik's Cube positions can be solved in at most 22 moves [10]. This leaves only three possible answers for the Rubik's Cube diameter in the face-turn metric: 20, 21, or 22.

3.5 Symbolic Methods Background

3.5.1 MDD

A multi-way decision diagram or MDD is a abstract structure that can be used to store sets [3]. MDDs differ from binary decision diagrams, BDDs, in that any number of outgoing edges are allowed instead of only two. The structure consists of nodes, and edges pointing to other nodes as seen in figure 1. Nodes are organized by levels counting down from the top to level zero at the bottom. Every level corresponds with a decision about which path to take, given by the input. A path through the structure can represent an input to a function, and the destination of the path determines the output. In the most basic definition of an MDD, the output is either a 1 representing membership in the set, or a 0 otherwise. For this project, the decisions at each level represent parts of the Rubik's Cube. A path through the diagram represents a description of the cube in a given arrangement. If the path goes to the 1 node at level zero, the path describes a cube state reachable through legal twists.

MDDs can be used to store and generate reachable state spaces very efficiently. The efficiency of algorithms used to add new reachable states to an MDD is related to the number of edges, not the number of states. A next-state function can be created using a MDD with twice as many levels as the MDD storing the states. The next-state MDD represents, in alternating levels, stored states and reachable states. For example, a path through the next-state MDD represents both a starting state and next state. If the next state is reachable from the starting state, the path will lead to the 1 node at level zero. Otherwise, the next state is not reachable from the given state. The relational product algorithm used to update the state space MDD is more efficient when there are fewer edges in both MDDs.

3.5.2 MDD Saturation

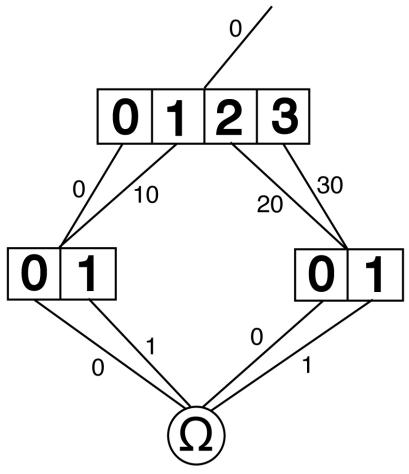
Repeatedly applying a global next-state function to an MDD can produce a very large MDD, even if the entire reachable state space can eventually be stored in relatively few nodes. This is because the number of nodes in the MDD can grow and shrink as new states are added. The memory requirements for generating a reachable state space using an MDD are therefore concerned with the peak size of the MDD.

The peak size of the MDD during state space generation can be improved through a number of techniques. Chaining achieves this by firing the same transitions over and over instead of firing all transitions at once [3]. Doing this means that the states are no longer found in breadth-first order. This also means that the diameter can no longer be determined by counting how many applications of the next-state function have been performed.

The best known way to lower the peak size of the MDD is to use saturation [3]. A saturated node has reached a fixed state meaning that firing a transition will no longer change it or its descendants. All nodes in a final MDD are saturated. The saturation algorithm makes use of the observation that saturated nodes no longer need to be changed, and only visits them once. The algorithm is called on the root node of the MDD, and recursively on all the children. The saturation thus starts at the bottom levels of the MDD, and fires local transitions repeatedly until no more changes occur. A node is then considered saturated, and the same process is repeated at the next level up in the MDD. A node can only appear in the final MDD if it is saturated, so the saturation algorithm works to lower the number of nodes in the MDD during generation.

3.5.3 EV+MDDs, Storing the Distance Function

An EV+MDD is an MDD with positive integer values assigned to each edge in the diagram [5]. A path through the entire structure will reach a node called omega in level 0 if the input is a member of the set. The sum of the edge values along the path can represent the output of a function. For the purposes of this project the value of the function represents the minimum number of moves required to reach the state. The figure below shows an EV+MDD where the function stored on the edge values is the input number in decimal. So, the sum of the edge values for the input 3, 1 would follow the path with an edge valued 30, then 1 for a total of 31.



The distance function can be computed using the saturation algorithm with a few modifications. When a transition is fired, new states are added and encoded so that their edge-value path is one greater than the previous state. Since saturation does not generate states in breadth-first order, when existing states are regenerated the distance must be stored as the lower of either the previous or the new value. When this process is complete the EV+MDD represents the reachable state space, and the sum of the edge values along a state's path equals the minimum number of moves required to reach that state. This means that, when completed, the EV+MDD can be used to quickly determine the diameter of the reachable state-space, and just as easily produce the paths necessary to reach the furthest state.

3.6 Ordering Heuristics

Every level of the MDDs used in this project corresponds with a part of the Rubik's Cube. Arranging the parts in different orders has dramatic effects on the time and memory required. Finding an optimal ordering is an NP-Complete problem [1]. Several heuristics were used to determine better orderings that could require less time and memory to compute. Whenever possible, state-space generation computations should be done with a good variable ordering, but to guarantee a variable ordering is good, you need to wait for state-space generation to be complete. At that point you probably have all the information you need. So, we must find good variable orderings before starting the computations. Some heuristics exist that give a value to an ordering that correlates with how well saturation will work [12]. This saves time by allowing good orderings to be found without doing the full state-space generation.

3.6.1 Level Orderings

Level ordering heuristics start by looking at the levels that are affected by each transition. As an example, the transition corresponding to turning the right side of the cube would affect the levels of the MDD containing Petri net places for the corners and edges on the right side of the cube. These affected levels are found for all the transitions in the Petri net model.

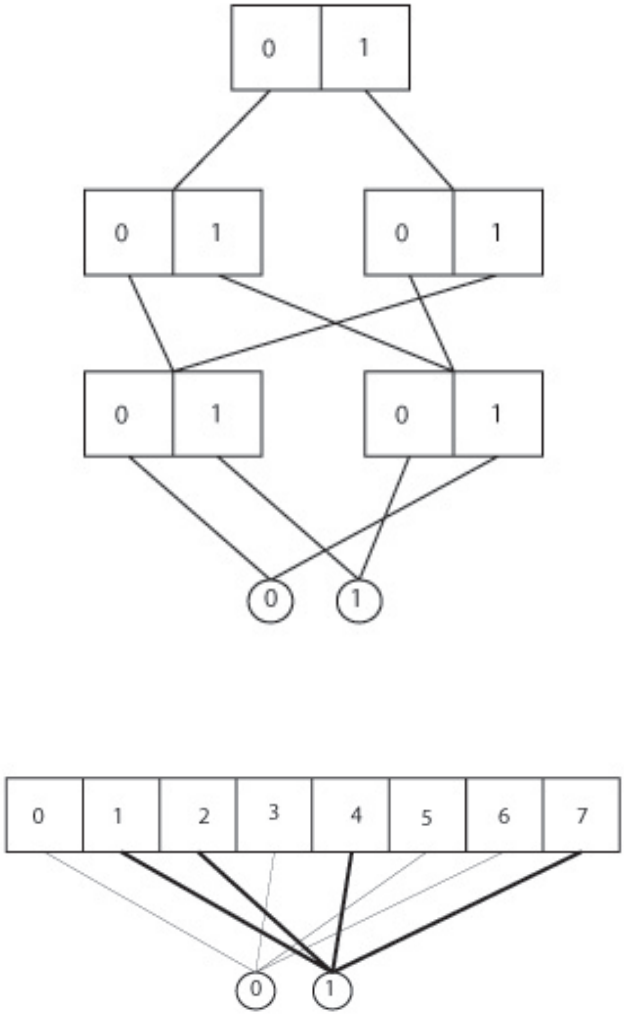
The *tops* heuristic seeks to take advantage of the fact that transitions affecting levels closer to the bottom of the MDD require less computation. The sum of tops adds up the total value of the topmost levels affected by each transition. Lower values for this number are considered better.

The *spans* heuristic tries to take maximum advantage of the local updates performed by the saturation algorithm. If the levels of the MDD affected by a transition are closer together, less computation should be required. The sum of spans adds up the difference between the top and bottom levels affected by each transition. Lower values are considered better.

3.6.2 Level Mergings

Merging two or more levels together can have dramatic effects on the run-time and memory required, both good and bad. If a level in an MDD is uniquely determined by the values of the levels above it, it might be a good idea to merge it [2]. In the Rubik's Cube problem, this

means that the value of the lowest level corner in the MDD can be determined by looking at the other corner values above it in the MDD. Merging levels too much leads to a reduction of the usefulness of the MDD to represent many states since the nodes have more outbound edges.



The effect of merging can be seen in the diagrams above which both encode the same set. The first diagram is a BDD, since there are only two decisions available and two outgoing edges for each node. The second diagram is a one-node MDD encoding the same set. A balance should be found that merges levels only as long as it is beneficial to do so.

4 Implementation and Design

4.1 Computing Hardware Available

A thorough accounting of the hardware used for this project allows for comparing results with other work, and a better understanding of the time and memory requirements for solving

similar problems. The greatest concern with hardware has been overall storage capacity. Execution speed is of little importance if the computer can't store the MDDs at their peak size. The algorithms used do not make use of more than one CPU core, so single core machines are adequate. The largest memory required so far has been around 25 gigabytes, which is necessary to store the MDD at its peak during the state space generation for the whole-cube model without center orientations. Two of the computers used for this project have enough memory for this computation.

Computer	Memory	CPU
Widowmaker	32 Gigabytes	8 cores, 2.3GHz Opteron 2376
Walnut	37 Gigabytes	4 cores, 2.5GHz Intel Nehalem

4.2 Implementation Details

4.2.1 Petri Nets and SMART

A Petri net combines a set of places for holding tokens, and transitions that give and take them. An arc from a place to a transition removes tokens from a place, and an arc from a transition to a place adds tokens to the place. The cardinality of an arc specifies how many tokens are changed. If a Petri net place doesn't have as many tokens as the cardinality of the arc, the transition will not fire. In a typical Petri net, when an arc from a transition to a place fires it will add the number of tokens to the place specified by the arc cardinality. For the Rubik's Cube models this limitation can make the model very complicated. It is much simpler for a place in the Petri net to correspond with a physical piece of the cube. When a twist of the cube is made, the new number of tokens in a place depends on another place in the model. For this reason, the models used for this project had to use marking-dependent arc cardinalities. This means that the number of tokens removed from or added to a place are defined according to how many tokens are present in other places. This means that firing a single transition can make all of the changes to the model without adding new places. Each transition firing can correspond with a twist of the cube.

The SMART and SMART2 software packages were used to compute the results in this project [4]. The original SMART package requires that when using marking-dependent arc cardinalities, the places whose Petri net markings depend on each other must be in the same partition. This means that the number of tokens in dependent places must be combined into one number, and stored in one level of an MDD. This is very inefficient for this problem, and loses most of the benefits of using decision diagrams for storage. This requirement meant only the smallest subproblem would complete using SMART, the corners-only with one corner held in place. SMART2 allows any partitioning of the Petri net model to be used. Without this feature a considerably more complex model of the cube would have been necessary if more efficient MDDs were desired.

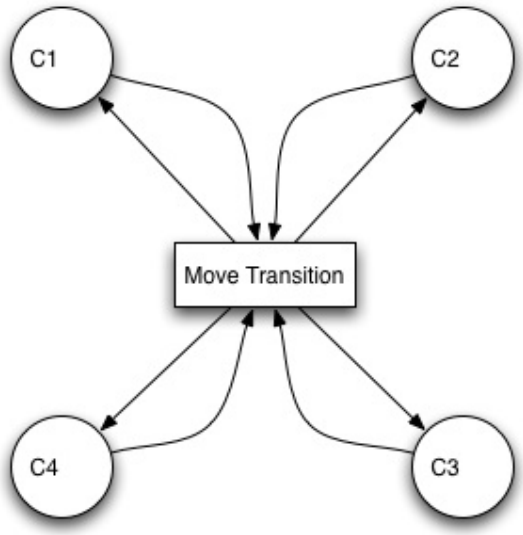
4.3 Rubik Models as Petri Nets

In an attempt to make the generation of the large state-space of the Rubik's Cube more manageable, several Petri net models were used. Testing different model ideas has been the only consistent way to figure out what works. At times it seemed like the success or failure of

an idea was completely counterintuitive. Persistence paid off, and eventually a usable model was found.

4.3.1 Single-Step vs. Multiple-Step Moves

A single-step model refers to the fact that every twist of the Rubik's Cube is accomplished by one transition in the Petri net. Every corner, edge, and orientation changed by a single move occurs simultaneously in one step. A multi-step model decomposes the effect of a single twist into multiple Petri net transitions. The first transition associated with a twist might move the corner tokens and store a code to remember the move. The second transition could read the stored code, and move the appropriate edges. This multi-step approach could be used to separate every change into its own transition if practical, but this has not been the case. There has been some benefit to using the multi-step approach, but it also always adds complexity to the models. The added complexity means that the MDD must also include the stored codes to remember the moves. This made finding good MDD orderings more challenging. In every case where an improvement in speed was found using a multi-step model, applying what was learned to a single-step model worked even better. The diagram below depicts how a move transition works to remove and replace tokens to many places at once when fired.



4.3.2 Square-Color-Only Model

The square-color model is the simplest model to understand. There are eight moving squares on each side of the cube, leading to a total of 48 places required in the Petri net model. Each place in the model holds a number of tokens associated with the color of the square on the cube. The transitions simply remove the tokens from each place that moved, and put them back in the appropriate place for the cube. The simplicity of this model unfortunately never led to good results. In all tested orderings, the square-color-only model performed worse using saturation than with explicit methods.

4.3.3 Cubelet Position and Orientation Model

Using separate Petri net places to represent the cubelets and their orientations has been the most effective model used. In this model, every Petri net place either represents what cubelet is in the corresponding location on the cube, or how it is oriented there. There are 8 places representing the corners, and 12 representing the edges. There are 8 places representing the corner orientations, and 12 representing edge orientations. In models that include center orientations, there are six places, one for each center square. This gives a total of 40 or 46 Petri net places in this type of model.

Because this model is split into cubelet positions and orientations, an added dimension of meaningful subproblems is available.

4.3.4 Submodels

Generating the state space of the cube is easier and faster when some factors can be ignored. Just like when a human tries to solve a Rubik's Cube, it is easier to start by trying to get one side of the cube correct while ignoring the others. Much can be learned about what orderings work better by working on smaller parts of the problem.

The simplest division of the cube is working on the corners or the edges in isolation. Working on testing models for the corners alone led to run times less than a second. Working on the edges alone was not as productive, since the best orderings found through trial and error took almost two days to complete. Improved edge orderings eventually had to be found through other means.

A second simplification of the problem comes from only allowing quarter-moves, or even only quarter-moves in a single direction. Including half-turns almost universally has detrimental effects on both running time and peak storage requirements. The same state-space is created whether or not half-moves are used, since a half-move is equivalent to two quarter-turns. For the same reason, allowing quarter-turns in only one direction also generates the same state space. A counter-clockwise quarter-turn is equivalent to three clockwise quarter-turns. Limiting the type of moves in this way generates the state space faster, but is not sufficient when distance-to-goal calculations are desired.

A third simplification of the problem is to only allow moving a few sides of the cube. Moving only one side of the cube should always lead to a total of four states, unless the one move allowed is a half-turn. This simple observation made it easy to test the model for errors. No matter what side is being turned, or which direction, it should always lead to four states. Likewise, any two opposite sides should lead to 16 states. Moving two adjacent sides is significantly more complicated, but should still lead to the same number of states for any two adjacent sides chosen.

When limiting the edge-only submodel to allow only two adjacent sides, it was observed that an edge always had the same orientation when in the same position. This idea could be extended to include four adjacent sides if the two sides left out were opposite each other. Even more profitable is the observation that with the four move limit, the initial states of the edge orientations can be set so that they never change at all. This meant that the edge orientations could be left out of the model completely with no effect on the outcome. With this new weapon the model could include all the Petri net places that would be in the whole

cube, but execute much faster since the edge orientations would never change.

The four-side subproblem was the most important discovery, because it actually finished running. Other attempts at whole-cube models failed because the models all ran for an indefinite amount of time. Occasionally they were left running for weeks and months without apparent progress. Little knowledge was gained from those experiments because progress was difficult to measure. There was no way to tell if the memory usage increasing slowly was due to slow execution of a bad variable ordering, or whether completion was seconds away.

With a sub-model found that actually finished computing, knowledge could be obtained. Since the four-side subproblem originally took nearly a day to execute, this was an upper bound on the length of time a new ordering needed to go. If a change to the ordering ran longer, it was halted and something new was tried. With four processing cores on the Walnut computer, and eight cores on Widowmaker, 12 different orderings could be tried in parallel. The first one to complete would be the new time to beat. Eventually, the running times became so short that the missing edge orientations could be added piece by piece.

The addition of twisting a fifth side to the model took longer as expected, but it still completed. It was almost anti-climactic when the addition of the remaining sixth side was included, as it completed more quickly with six than with five. Generating the complete state space of the Rubik's Cube only ended up taking about an hour. Adding half-turns to the model increased run time to around 8 hours.

The last addition to the model was the center orientations. This state space is 2048 times larger than the traditional Rubik's Cube problem, but unexpectedly finishes faster. The center orientations are added to the top of the MDD. This means that every twist of the cube affects levels corresponding to the center squares at the top, and corners at the bottom of the MDD. According to both sum-of-tops and sum-of-spans heuristics, this seems like a bad idea. Using the new heuristic for ordering designed for this project, the anomaly could be explained. Corner levels depend only on other corner levels, and center orientations only depend on themselves. So, within each large span of levels affected by each transition the levels are tightly grouped into levels depending only on each other. It is also possible that the center square orientations organize the lower level nodes more efficiently.

4.4 Ordering Heuristic System

This project needed to deal with the problem that computing a state space can require an unknown amount of time. For this reason it helped to have some static ordering techniques that could be applied before starting computation. A good ordering can be dramatically more efficient to compute than a bad one, so it makes good sense to use every bit of knowledge available.

We created a Java program to aid in determining good variable orderings. The first step was to implement the sum-of-tops and sum-of-spans calculations as described above. The basic approach to finding better orderings was to first make a random swap of levels in the MDD. Second, the sum-of-tops and sum-of-spans were calculated on the new ordering. If the new heuristic values were better than the best found so far, the new ordering is stored. Otherwise, the the process repeats and makes a new swap. After a number of swaps without improvement, the existing ordering is changed back to the best known order. Generally only a few seconds are required before the ordering finds a good local minimum. Globally optimal

orderings are not known, since even the corners-only model has 16! permutations which is far too many to test.

It was observed during experimentation that having levels representing cubelet positions close to levels representing their orientations did not perform well. This disagrees with the sum-of-tops and sum-of-spans heuristics since both positions and orientations are affected by the same transitions. This could be caused by a number of factors, the first of which is because the orientation of a cubelet is completely independent of its position. A cubelet in a given position can be oriented any possible way. The second reason could be that keeping positions and their orientations close together pushes further apart the positions and orientations that are dependent on each other.

A new heuristic was developed that uses the knowledge that marking-dependent cardinalities introduce dependence within a transition. Rather than just counting which levels are affected by a transition, this heuristic finds the sum of spans between levels with marking dependent cardinalities. As an example, if levels A, B, C, and D are affected by a transition they should be close to each other in the MDD in the old sum of spans heuristic. The new heuristic can use the extra information available if A depends on C, and D depends on B. Whereas the old heuristic sees no difference between any ordering of A, B, C, and D, the new heuristic will favor an ordering like ACDB. Further work would be required to find out how to weight this new heuristic in combination with sum of spans and sum of tops, but the results seem to more closely match the orderings that experimentation found to be superior. It is difficult to optimize orderings using this new heuristic for both quarter and half moves. This could explain the difference in running times when half moves are used.

5 Results

5.1 Time and Peak Storage for all Relevant Computations

The following chart contains run time data for relevant computations of the Rubik's Cube models in SMART2 on the Walnut Computer.

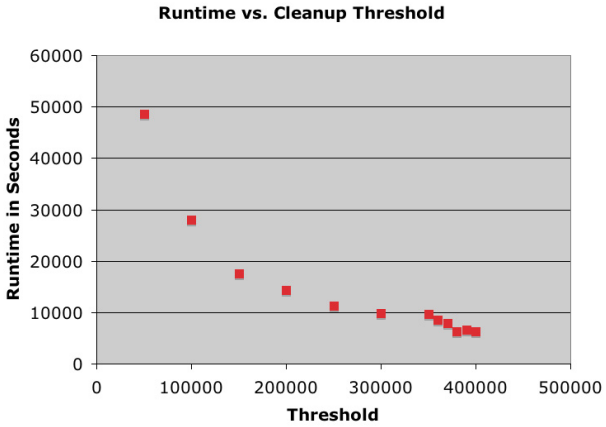
Model	Time in Seconds
Whole cube, quarter and half-moves	28140
Whole cube quarter-moves only	3900
Whole cube clockwise quarter-moves only	2100
Edges alone, quarter-moves only	108
Corners alone, quarter-moves only	0.5

Model	Memory (MB)	Seconds	Cleanups
Whole cube with center, half and quarter	5426	7799	82
Whole cube with center, quarter only	2982	3203	52
Whole cube with center, clockwise quarter only	3002	2746	50
Whole cube, quarter only	3556	8074	286
Whole cube, clockwise quarter only	2117	2745	37
Edges alone, quarter only	650	111	2
Edges alone, clockwise quarter only	652	55	1
Corners alone, quarter and half	39	2.48	1
Corners alone, quarter only	29	1.14	1
Corners alone, clockwise quarter only	13	.73	1

The above chart contains peak memory usage, run times, and the number of cleanups for computations done on the Widowmaker computer. Since both Widowmaker and Walnut are 64-bit computers running Linux, the peak memory and number of cleanups for each model are the same. Cleanups contribute significantly to the runtimes on larger models.

5.2 SMART2 Tool Observations

Saturation is a very efficient algorithm, but specific implementations of the data structures call for different strategies for choosing parameters. One example is the cleanup threshold which determines when unused memory will be cleaned up. To improve performance on the Rubik's Cube models using SMART, it has been beneficial to use the maximum cleanup threshold available so that unnecessary nodes are cleaned up as infrequently as possible. This operation is expensive when MDDs are at their peak size, since every node in the MDD must be visited.



Using a cleanup threshold as large as possible is recommended, and significant improvements in run times are achieved. The chart above shows how doubling the cleanup threshold approximately cuts the run time in half. There are diminishing returns since there are only a finite number of cleanups performed. With a high enough threshold, only one cleanup will be performed, and there will be no more benefits. However, since it is often more likely that peak memory requirements will be a limiting factor, a smaller threshold will clean up unused nodes more often, and use less memory.

5.3 Model Observations

Corners, edges, and their orientations should be in separate MDD levels generally, but merging levels led to some performance improvements.

Corners, edges, and their orientations should be clustered together in adjacent levels. Corners should not be interleaved with their orientations, for example.

Merging of levels can be very helpful, or very harmful, and deserves further consideration. Merging levels has led to some performance improvement, but no observable patterns exist that should be used as a rule of thumb as to when merging should be done. Trying as many merges as possible is recommended.

Single-step moves are generally faster than multi-step moves, and the results are easier to interpret. For the Rubik's Cube problem, any good ordering for multi-step models has led to a faster and less complicated single-step ordering.

State space generation is faster and uses less memory without half-turns. There doesn't seem to be a good explanation for this phenomenon available without doing more testing.

State space generation is faster with quarter-turns in only one direction. This could be due to tested orderings fitting one-direction moves better. There might very well be orderings that are faster for quarter-turns in both directions.

Orderings in Petri nets with marking-dependent cardinalities should use some heuristic that will minimize the distance between the places on which the cardinalities depend.

5.4 Achievements

This project generated the state-space of the Rubik's Cube, with and without including the center-square orientation. The knowledge gained in this project is likely sufficient to allow the use of EV+MDDs to compute the shortest path to solve any state of the Rubik's Cube in the near future.

6 Future Work

Having generated the entire state space of the Rubik's Cube puzzle, what remains is to use EV+MDDs to compute the distance function for the cube and other problems. The distance function computation will take at least as much memory and computation as generating the state space. Since the state space generation requires nearly all the memory on the available computing hardware, it may take additional resources to accomplish this goal.

The orderings used to generate the state spaces are good enough to allow computation to only take a few hours. This does not mean that there aren't orderings that are vastly superior that haven't yet been found. Improved orderings may be the only chance to compute the distance function without access to more memory. Thankfully, this project has found a large group of subproblems that might be solved even if the complete Rubik's Cube distance function is not easily computable.

The techniques used in this project allow other puzzles to be solved in addition to the Rubik's Cube. The 15-puzzle with a 4x4 grid of sliding tiles was also used in this project, and its state space can also be generated quickly. This puzzle's maximum distance is known, but EV+MDDs could be used to confirm that the distance is 80.

There is no reason that the techniques used in this project couldn't be applied to larger and more important problems. There has been a number of mathematicians working on the Rubik's Cube problem for years, and the great leaps made using saturation are begging to be applied in new areas. It is conceivable that saturation could in the very near future be used to solve problems such as whether or not the white pieces can force a win in chess. Explicit searches recently proved that checkers was a draw after years of computation, but symbolic methods may make solving this type of problem commonplace.

7 Conclusions

The generation of the state space for the Rubik's Cube problem has completed. The computation can be done on relatively inexpensive computing hardware, and still finish in less than a day. The entire reachable state space can be stored in an MDD containing around 5000 MDD nodes, although up to 25 GB can be required during generation. This demonstrates the superiority of the symbolic approach for state space generation using MDDs and the saturation algorithm. What remains to be seen is what results will be produced when EV+MDDs are used to compute the distance function.

The Rubik's Cube was a good choice for testing the boundaries of knowledge about what actually works well with the saturation algorithm. Level ordering heuristics work well on small problems, but the Rubik's Cube model has shown exceptions to some of the rules-of-thumb that appear when marking-dependent cardinalities are used. These exceptions led to a better general-purpose ordering heuristic that considers the dependence of one MDD level on another. The computations performed for this project have met and exceeded the capabilities of the SMART software package default settings. This leads to the conclusion that low cleanup thresholds can severely impact runtimes.

References

[1] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comp.*, 45(9):993–1002, Sept. 1996.

[2] G. Ciardo, G. Lüttgen, and A. J. Yu. Improving static variable orders via invariants. In J. Kleijn and A. Yakovlev, editors, *Proc. 28th International Conference on Application and Theory of Petri nets and Other Models of Concurrency (ICATPN)*, LNCS 4546, pages 83–103, Siedlce, Poland, June 2007. Springer-Verlag.

[3] G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state-space exploration. In *STTT International Journal on Software Tools for Technology Transfer Vol 8, Issue 1*, pages 4–25, 2006.

[4] G. Ciardo and A. S. Miner. SMART: Stochastic model checking analyzer for reliability and timing. Website, Present. `http://www.cs.ucr.edu/~ciardo/SMART/index.html`.

[5] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 256–273. Springer-Verlag, Nov. 2002.

[6] N/A. A* Search Algorithm. Website, Present. `http://en.wikipedia.org/wiki/A*`.

[7] N/A. Rubik's Cube. Website, Present. `http://en.wikipedia.org/wiki/Rubik%27s_Cube`.

[8] M. Reid. Cube Lovers: superflip requires 20 face turns. Website, Present. `http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__superflip_requires_20_face_turns.html`.

[9] M. Reid. M-symmetric positions. Website, Present. `http://www.math.ucf.edu/~reid/Rubik/m_symmetric.html`.

[10] T. Rokicki. Twenty-Two Moves Suffice. Website, Present. `http://cubezzz.homelinux.org/drupal/?qode/view/121`.

[11] Rubik. Rubik's Official Online Site. Website, Present. `http://www.rubiks.com/`.

[12] R. Siminiceanu and G. Ciardo. New metrics for static variable ordering in decision diagrams. In H. Hermanns and J. Palsberg, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 3920, pages 90–104, Vienna, Austria, Mar. 2006. Springer-Verlag.

[13] The GAP Group. Analyzing Rubik's Cube with GAP. Website, Present. `http://www.gap-system.org/Doc/Examples/rubik.html`.