Optimally Solving the Rubik's Cube with Pattern Databases

Adam S. Hayse
CS 462 - Artificial Intelligence
Central Connecticut State University
New Britain, CT 06050
ahayse501@gmail.com

Abstract

This paper discusses an undergraduate student's implementation of a heuristic search algorithm along with pattern databases as discussed in Richard E. Korf's 1997 paper titled "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases." The code can be found at https://github.com/AdamHayse/optimal-solve-rubikscube. The implementation generates a pattern database concerned with subsets of pieces in the Rubik's cube; specifically, a subset concerned with only the corners pieces and subsets concerned with varying numbers of edge pieces were generated and examined. Pattern database generation was done via a breadth-first search and without the need for a queue. A highly efficient encode and decode function dramatically reduced the time and space requirements during the breadth-first search, which allowed for generating very large pattern databases. Additionally, the iterative deepening A* (IDA*) search algorithm was used to generate scrambles whose solutions were the same number of turns. Thousands of scrambles of this nature were created and tested using pattern databases of varying sizes to show how the increase in pattern database size increases the efficiency of the heuristic search algorithm.

Introduction

A Rubik's cube is a 3-dimensional combination puzzle. It can be visualized as a cube composed of 27 sub-cubes, making its measurements 3×3×3 units. In its solved state, the exterior of each face of the cube has its own distinct color compared to its other faces. The puzzle can be scrambled by performing 90, 180, or 270 degree turns on any of the 6 faces. The goal of the puzzle is to restore the cube to the state where each face of the cube has the same uniform color by permuting the sub-cubes with legal turns of the cube.

Solving the cube is considered very difficult since there are $(12! \times 2^{12}) \times (8! \times 3^{8}) / 12 = 43{,}252{,}003{,}274{,}489{,}856{,}000$ valid combinations of the sub-cubes that can be solved. Coming up with an effective heuristic is also difficult because solving the puzzle is somewhat counter-intuitive; if you try to solve the cube one face at a time, then you need to essentially mess up the cube to get closer to the solution.

Turn Metric

Before optimal solutions to the cube can be found, what is considered a legal move must be defined. This is called the turn metric. There are many different turn metrics that can be used to define a legal turn. Three of the most common are the quarter turn metric, the half turn metric, and the slice turn metric. The quarter turn metric considers any 90 degree turn on one of the six faces a legal move. This also means that a 180 degree turn on any given face would count as two moves. The half turn metric is the same as the quarter turn metric, but it considers 180 degree turns to be one move. This means that 3 moves can be performed per face, totaling up to 18 possible moves from a given state of the cube. The slice turn metric is the same as the half turn metric, but it allows turns through the middle of the cube. For example, performing a slice turn on the cube yields the same result as if two separate

turns were performed on two opposing faces of the cube.

A large amount of testing on the Rubik's cube has been done using the half turn metric. In 2010, it was exhaustively proven that the maximum number of moves needed to solve any solvable combination of the Rubik's cube is 20 using the half turn metric. Since this metric had the most information available, I used the half turn metric in my IDA* search algorithm.

For the purposes of this project, the Rubik's cube can be thought of as fixed on a 3-dimensional Cartesian plane where each axis goes through the center of each sub-cube at the center of each face. This means that these center pieces can only rotate about these axes. Since these center pieces cannot move to other faces of the cube and their orientation does not matter, they are not considered by the functions that describe the possible moves. They can be thought of as indicative for which colors must end up on a given side for the puzzle to be considered solved.

Notation

I use the standard notation accepted by people within the Rubik's cube community to describe the moves that can be performed. One of six letters is used to describe a face, followed by a symbol that describes one of the three moves to perform. These symbols and operators are as follows:

Symbols	**Operators**
U – Upper face	(none) – 90° clockwise
F – Front face	2 - 180°
L – Left face	' - 90° counter-clockwise
B – Back face	
R – Right face	
D – Down face	

For further clarification, here is an example:

R U R' U2 means turn the right face clockwise 90°, then turn the upper face clockwise 90°, then turn the right face counter-clockwise 90°, then turn the upper face 180° (turn direction doesn't matter).

Heuristics

There are a couple of heuristics that can be considered with the Rubik's cube. The Rubik's cube can be thought of as a 3-demensional version of the sliding puzzle. A 3-dimensional version of the Manhattan distance can be applied to this puzzle. I do not consider this heuristic in this project. This admissible heuristic has an expected value of 3 for the corner pieces and 5.5 for the edge pieces [Korf, 1997]. In 2010, it was shown that the average number of moves needed to solve any state of the cube is about 17.702396 [Rokicki, Kociemba, Davidson, and Dethridge, 2010]. Korf estimated that the typical 18 move solution would take over 250 years on a Sun Ultra-Sparc Model 1 computer. Using a processor that is 20 times faster would still make this heuristic insufficient for solving a random scramble optimally in a reasonable amount of time.

The second heuristic is called a pattern database. Most of the time, a heuristic value is the result of algorithmic computation (like is the case with the Manhattan distance). However, there is no reason why a heuristic can't be obtained from a lookup table. This is what the pattern database does. It breaks up the cube into subsets of pieces and then finds the required number of moves to solve any combination of these subsets. For example, with 8 corner pieces and 3 orientations per corner piece, there are $8! \times 3^8 = 264{,}539{,}520$ different ways that these pieces can be combined. However, the state space of the Rubik's cube is a disconnected graph consisting of 12 subgraphs of equal size. It turns out that the orientation of the 8^{th} corner piece is determined by the orientations of the other 7 corner pieces. Therefore, this value can be divide by 3 to get a total of 88,179,840 solvable combinations of only the corner pieces. This is a large difference compared to the 43 quintillion different combinations of the entire cube. This property makes pattern databases of subsets of the cube much more manageable.

For the pattern database as a heuristic to work, there needs to be multiple pattern databases. The union of the subsets of pieces that each pattern database describes must cover all the movable pieces of the cube. At every node examined during the heuristic search, every

pattern database is consulted and the largest heuristic is used. Using many small pattern databases (subsets of 3 or 4 pieces) is memory efficient but has an average heuristic that is similar to the 3-dimensional Manhattan distance. Reducing the number of pattern databases increases their sizes exponentially. For example, reducing to one pattern database essentially means having a perfect heuristic for finding solutions, but there would be 43 quintillion entries in the lookup table.

Generating the Pattern Databases

For this project, I used three different sets of pattern databases in my search algorithm. All these sets do not have a heuristic value within them that exceeds 15. Therefore, every value within all these databases can be stored in 4 bits each, reducing the total number of bytes needed by a factor of 2.

Set 1 – 82.65 MiB:
Set of all corners
2 sets of 6 edges

Set 2 – 529.31 MiB:
Set of all corners
2 sets of 7 edges

Set 3 – 4.80 GiB:
Set of all corners
2 sets of 8 edges

The 7 total pattern databases used are generated using a breadth-first search from the solved state of the cube. Each increase in depth is an additional move performed on a given state of the cube. The BFS algorithm finds shortest path solutions in the case where every action cost is the same. Therefore, if a state has already been seen, then it does not need to be expanded in the search. By finding the shortest path from the solved state to any possible combination of the subset of pieces, the shortest path from any possible combination of the subset of pieces to the solved state is also found.

I originally used a queue to execute the BFS. This meant that a significant amount of memory had to be reserved to hold the combinations in queue to be expanded. I was uncertain of how much memory was needed since there is no easy way to find the maximum number of combinations in queue at once, so I allocated enough memory to hold all combinations being considered. This brute force strategy would not work later on for the larger databases.

When a combination was popped off the queue to be expanded, all 18 possible moves were performed on this combination. The resulting 18 states were input into an encode function, which calculated offsets from the database pointer for where the current depth of the BFS was to be stored for future reference. If the offset in the database contained a value of 0xF, then this combination had not been seen before and it should be added to the queue. If the offset in the database contained a value other than 0xF, then this combination had been seen before and it should be ignored. The allocated memory for the database is initialized to all 1's at the start since no states have been seen yet, and the search continues until there is no longer a state that hasn't been seen.

Encode Function

I originally used an encode function that I derived from the lexicographic permute algorithm. I calculated the permutation offset iteratively by factorial expansion and multiplying by the relative position of a piece among the remaining pieces being looked at. Once I found this permutation offset, I scaled it by the number of ways that the set of pieces could be oriented. Then, I finally added an orientation offset to the accumulated offset to find where a combination was to be stored in memory. Since I store two values per byte, I divide this offset by 2 to find the location in memory as well as find the remainder to determine which side of the byte that it should be stored in. This encode function had $O(n^2)$ efficiency where n is the number of cubes in the subset.

I continued to look for improvements to this encode function because it was the basic operation of the pattern database generation, and it would also be the basic operation of the IDA* search algorithm. I found a linearithmic function that involved using a binary tree. However, the overhead cost of this function made it considerably worse than the $O(n^2)$ encode

function I started with initially. I put too much value in the worst-case efficiency. When dealing with very small values of n, the worst-case time analysis is not too valuable.

However, my mischaracterization of the value of the asymptotic growth rate of these algorithms led me to find an O(n) algorithm that was faster than my original algorithm. It made clever use of the shift operation of the CPU. This encode function only works for sets containing 16 or fewer elements. I also found a corresponding O(n) decode function from the same source [chqrlie, 2016].

Decode Function

The decode function allowed me to make a very favorable time-space tradeoff in the BFS algorithm that generated the pattern databases. It eliminated the need for a queue, which saved about 16 or 24 times the amount of memory used depending on whether it was a corner or edge database being generated. So, the addition of the decode function dramatically reduced the space needed at the cost of a little more time to obtain a combination. However, the new BFS strategy also allowed for easy implementation of a divide and conquer strategy with multithreading.

This new BFS strategy works without a queue by looking at previously entered values in the database. Every time the depth of the search is incremented, the database of heuristic values is scanned from first entry to last entry. Whenever a value equal to the previous depth is seen, this location in memory is decoded into the combination that corresponds to that memory location, moves are performed on this combination, and then the resulting combinations are encoded and depth is written to their corresponding locations if the state has not been seen yet. It's easy to see how a divide and conquer strategy can be implemented with this technique. However, there is a potential race condition since two values are stored per byte. This can be easily fixed by making the process of writing to the database atomic.

Search Algorithm

There are two heuristic search algorithms considered by Korf. The first is A*. A* finds optimal solutions within a graph if the heuristic used is admissible and consistent, which is the case with how the pattern databases are used. According to Korf, the search tree has an effective branching factor of about 13.34847. This means that the space and time efficiency is about $O(13.34847^d)$ where d is the depth of the solution. Exponential time efficiency can be managed, but the space efficiency is simply too poor to find optimal solutions to most scrambles.

The second algorithm is called iterative deepening A*. This algorithm is an iterative deepening depth-first search with a depth limit that is imposed by a heuristic. This algorithm has the same worst-case efficiency as A*, but it has a slightly worse average case efficiency due to the overhead cost of having to restart the search multiple times. However, this overhead cost becomes less of an issue in search problems where the effective branching factor is large. In this case, the additional cost is about 8%.

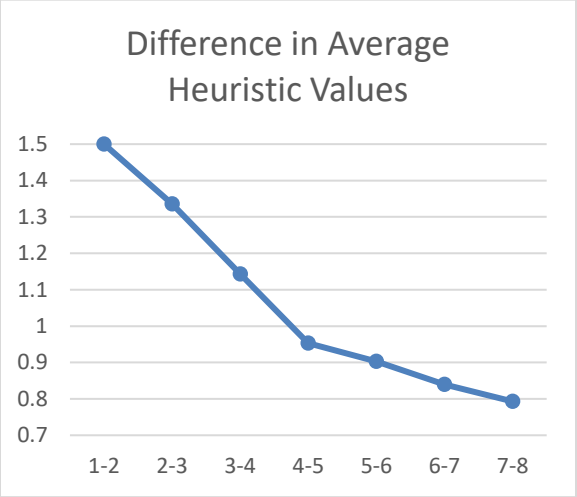
IDA* is the clear winner due to its space efficiency of only O(bd). It only needs to store a number of nodes proportional to the effective branching factor times the depth of the solution. In addition to making the search possible for any solvable combination of the Rubik's cube, the low memory cost allows the use of large pattern databases to give a better heuristic.

Average Heuristic of Pattern Databases

In this project, I generated a corners database, two 6 edge databases, two 7 edge databases, and two 8 edge databases for use in my search algorithm. The average heuristic for the corners database is 8.764. I also generated 1, 2, 3, 4, and 5 edge databases to examine their average heuristic. The following table shows the results.

Edges	Size (bytes)	Average Heuristic
1	12	1.833
2	264	3.333
3	5,280	4.669
4	95,040	5.812
5	1,520,640	6.765
6	21,288,960	7.668
7	255,467,520	8.508
8	2,554,675,200	9.301

Adding an additional edge to the database generation increases the database size exponentially. It also appears to increase the average heuristic linearly. However, there is a strange phenomenon when examining the increases in the average heuristic as more edge pieces are considered. For edge databases 1, 2, 3, 4, and 5, the rate of change of the difference in average heuristic is very similar. However, for edge databases 5, 6, 7, and 8, the rate of change of the difference in average heuristic takes a sharp turn. I'm not sure what the reason for this is, but it seems like an interesting result that is worth mentioning. The following chart demonstrates this phenomenon.



Experimental Results

Using the IDA* search algorithm and the generated pattern databases, I was able to find optimal solutions to random scrambles of the Rubik's cube. I modified this search algorithm to generate random scrambles that had the same number of moves as their optimal solutions. For scrambles with a small number of turns (11 or 12), the optimal solution was often the same as the scramble but done backwards. As the scramble length increased, the number of possible ways to solve a given state also increased. I found some 17 and 18 move scrambles that had solutions of the same length, but the solution took a completely different path.

After I generated these scrambles, I fed them into the IDA* search algorithm and found the number of explored nodes using pattern databases of different sizes. The following table shows the average number of nodes explored with different sized edge databases. The leftmost column shows scramble length and quantity.

Moves	6 edges	7 edges	8 edges
11 (1000)	5.45E+04	1.63E+04	5.80E+03
12 (1000)	5.20E+05	1.43E+05	4.33E+04
13 (1000)	5.33E+06	1.45E+06	3.83E+05
14 (1000)	6.86E+07	1.90E+07	4.98E+06
15 (400)	9.16E+08	2.53E+08	6.33E+07
16 (100)	-	1.76E+09	7.98E+08
17 (9)	-	-	2.11E+09
18 (1)	-	-	2.57E+08

My strategy for finding scrambles of a given length is not reasonable for finding scrambles whose solutions are of length 19 or 20. I found the scrambles I used by generating a random scramble of a given length, finding the solution, and keeping the solution if it was the same length as the scramble. The exhaustive search of the cube in 2010 showed that 3.43% of combinations required 19 moves to solve, and that 1.12E-09% of combinations have 20 move optimal solutions. The probability of finding a 20-move scramble with a 20-move solution is close to 1 in 100 billion.

The following table shows the percent increase in efficiency of the IDA* search algorithm as the pattern databases grow larger. It can be derived from the previous table.

Moves	6 to 7	7 to 8
11	235%	181%
12	265%	230%
13	269%	278%
14	262%	281%
15	263%	299%

The variance in number of nodes explored is very large. For example, when using the 8-edge database to test the 1000 11-move scrambles, the smallest number of nodes explored for a scramble was 73 while the largest number was 358,281. It was also occasionally the case that solution with the least or most number of explored nodes was not the same when tested with the different sized pattern databases.

Conclusions

In this project, I was able to generate pattern databases via a breadth-first search. The encode and decode function allowed the BFS to operate without the need for a queue. Multithreading was also implemented to speed up pattern database generation. The generated pattern databases were successfully used in the IDA* search algorithm to find solutions to scrambles of the Rubik's cube.

There are a few ideas that I did not implement in code. One idea would be to modify the code to work with other combination puzzles. However, I did not implement this idea because I was skeptical that it would work. The Rubik's cube appears to be special in that each turn performed moves 8 of the 20 movable pieces. The fact that 40% of the pieces move in any given turn means that the depth of the solution is within a reasonable depth range. This search method will not work effectively for puzzles where the percentage of total blocks moved in a turn is low.

Another idea I had is to use the slice turn metric. This would not be very difficult. It would simply require 9 additional turn functions. This would increase the branching factor of the search, but it would also reduce the depth of the search.

My final idea was to employ an explanation-based learning strategy by adding common sets of turns as legal moves. This would also increase the branching factor, but the solution would be at the same depth. It would be interesting to see how this affects the number of nodes explored.

I think that this project is a good exercise of the ideas taught near the start of the semester in CS 462. I would highly recommend this project to future students that will take this class. There is clearly room to do more with this project, so I would be interested in seeing someone build off what I have done so far.

Acknowledgements

I want to thank Robert Clausecker from Zuse Institute Berlin for answering my questions about pattern database generation and for general advice in C programming over the course of 2 months.

References

[Korf, 1997]
Richard E. Korf, *Finding Optimal Solutions to Rubik's Cube Using Pattern Databases*, AAAI/IAAI, p. 700-705, 1997.

[chqrlie, 2016]
Username chqrlie, answer to question *How can I effectively encode/decode a compressed position description?*, Stack Overflow, https://stackoverflow.com/a/39627199/417501, September 2016.

[Clausecker, 2017]
Robert Clausecker, *Notes on the Construction of Pattern Databases*, Bachelor thesis, Zuse Institute Berlin, Berlin, Germany, 2017.

[Rokicki, Kociemba, Davidson, and Dethridge, 2010]
T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, *God's Number is 20*, http://www.cube20.org, 2010.