COMPSCI 765 FC
Advanced Artificial Intelligence
2001

Towards Optimal Solutions for the Rubik's Cube Problem

Aaron Cheeseman, Jonathan Teutenberg

'Being able to solve Rubik's cube very fast is a near useless skill, that takes a lot of time to acquire, and does not impress the opposite sex. So if you think you have better things to do, I can only agree. You probably have.' **- Lars Petrus.**

Contents

Contents **2**
Introduction **3**
What is the Rubik's Cube problem? **3**
Aims of the project **3**
Initial expectations 3
Eventual Goals 3
Representing the Cube **4**
Cubies vs. Squares **4**
Cubies 4
Squares 4
Abstract representation **4**
Data structures **5**
States 5
State Sets 6
Representing moves **6**
Figure 3: Possible Moves for a Rubik's Cube 7
Search Technique **8**
Search Method ..... Error! Bookmark not defined.
Heuristics 8
Constraints 9
State Constraints *9*
Move Constraints *9*
Constraints encoding *9*
The Petrus method **10**
Lars Petrus 10
The method 10
Step 1 ..... *10*
Step 2 ..... *10*
Step 3 ..... *11*
Step 4 ..... *11*
Step 5 ..... *11*
Sub-Goals **11**
Initial Sub-Goal Encoding 11
Spatial Reasoning 12
Knowledge *12*
Reasoning *13*
Language *13*
Final Process 14
Results ..... Error! Bookmark not defined.
Bibliography **15**

Introduction

What is the Rubik's Cube problem?

The creation of the Rubik's Cube dates back to 1974 in communist Hungary. The creator was Erno Rubik, a lecturer in the Department of Interior Design at the Academy of Applied Arts and Crafts in Budapest. In 1978 the Cube had swept Hungary, and by 1982, the world. It is estimated that well over 100 million Rubik's Cubes and their imitators have been sold – an unprecedented amount for any toy.

The problem itself, with which you are no doubt familiar, is a cube made up of 26 smaller cubes that are able to rotate through all 3 dimensions. The aim is to rotate these so that each face of the large cube is made up of nine squares of the same colour.

Aims of the project

Initial expectations

To begin with we hoped to implement a system by which a computer could solve a Rubik's Cube from any configuration – a task with which we had no prior experience.. This implementation was to use a combination of constraints satisfaction and traditional planning techniques. Our initial investigation of the problem showed that such a task was completely infeasible. Recent research has uncovered methods of finding optimal solutions [1][5], but none of these can be performed in reasonable time, and they are very inelegant in their approach.

By the end of our project we had, in fact, managed to devise a possible system for a planning algorithm based on spatial relationships within the cube, but this had to be left as future work, and is outlined in that section.

Eventual Goals

Our final set of goals was to find a method for generating near-optimal solutions to the Rubik's Cube in an efficient manner relative to both space and time. We wished to achieve this efficiency through the use of constraints on all features of the search. As you shall see, we have been relatively successful in achieving this goal, and are left with a robust, mature solving algorithm.

Representing the Cube

Cubies vs. Squares

There are a number of ways of interpreting the structure of a Rubik's Cube, and therefore a number of ways of representing it. The two most common views of the Cube are as a set of 9x6 squares, or as 3x3x3 smaller cubes called cubies [1].

Cubies

A representation based on cubies has two parts. The first is a labeling of each mobile cubie, which excludes the center cubie that doesn't rotate, and the six middle cubies that retain their relative positioning regardless of state [1]. The second part is to keep a location and orientation of each cubie, which is what defines a unique state in the search space.

The advantages of such a representation are in the fact that it is the most compressed format, and thus a large number of states can be held with minimal memory cost. The exact number of bits used to represent a single state is 92 bits [2]. The disadvantage of a cubie representation is in the difficulty of comprehension by people. We believe that the time cost in trying to realise a mental model of a cubie-based state outweighs any space savings, which brings us to the squares representation.

Squares

This is the most natural way for a newcomer to visualise a state of a Rubik's Cube. The cube is seen as a collection of six faces, each made up of nine squares that can be coloured with one of six colours. We chose to use this representation, as we are unfamiliar with the Rubik's Cube, and did not have time to get our heads around the more complex models.

Abstract representation

In the Rubik's Cube, each face can be represented by a unique colour. This is the colour of the center square, which never changes in relation to the others. Our colours are simply integer values from 0 to 5, plus the number 7, which is used as a wild card in the representation of goal states.

Locations on the cube are given as the pair (face, square), where the squares are a number from 0 to 8, as shown in figure 2.

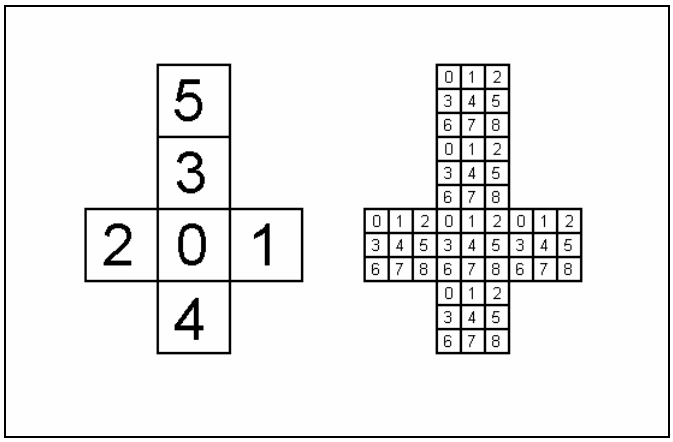


Figure 2: Representation of squares of a Rubik's Cube

Data structures

'We turn the Cube and it twists us.' **– Erno Rubik.**

States

First we shall find the optimal size of a state using our square representation – although we must keep in mind that this is not necessarily the best as far as comparison efficiency. For each face we must hold eight squares of information, each of which can be one of seven values. This requires a total of 126 bits to represent.

Initially we began with a simple state structure as a double array of integers. This proved swift for comparisons, but caused memory overflow at a very low search depth. As Java uses 32 bit integers, each state used 1728 bits, well above the optimal number. We then considered moving to double bit arrays, but there was some concern over the additional information Java stores within an array object, so we took one further step.

The final encoding was as six integers, one for each face. Each square is 3 bits of the integer, using a total of 27 bits (so 5 bits are superfluous), as shown in figure 3. This representation uses only 192 bits, and also has the nice feature that it can be easily manipulated by moves and for comparisons.

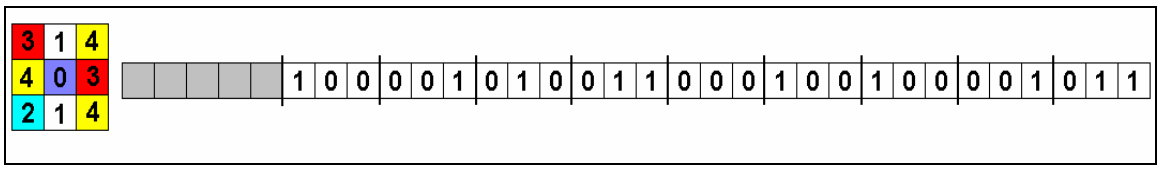


Figure 3: Encoding of a random face

State Sets

After each step towards optimisation of state size was made, the focus of our attention moved from space to speed. As our preliminary searching was done in pure brute-force style, with each new state being compared to all previous ones, the speed of lookup within the state data structure was of vital importance.

Our first foray into finding a suitable data structure was the use of a sorted vector, using a binary search for lookup. States were sorted by the differences in the values for each of their faces - first sorting by the front face, breaking ties with the top face, then with the back face etc.

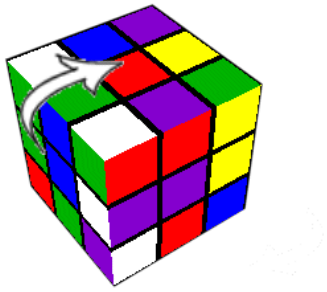
This worked fine for the initial tests, where memory overflow was of greatest concern, but once we had optimised the state representation, problems began to occur. Once the number of states reached the hundreds of thousands, searching the state set became a major task. We moved to a representation based on a hash table of sorted vectors (what we call buckets). We experimented with a variety of keys, until a good solution was found. In the end, each bucket contained an average of around one thousand to five thousand states, which were indexed by the sum of the integer values of their faces giving around 30 million potential entries in the hash table.

Representing moves

There are three basic movements available to someone trying to solve a Rubik's cube. These moves can then be further broken into 3 sub moves for each movement, defined as follows:

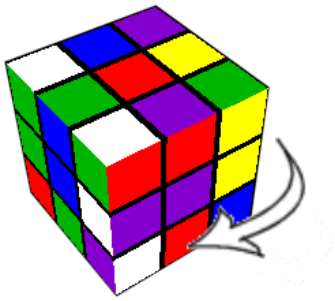
Vertical Movement :

UP Move (90°)
UPDOWN Move (180°)
DOWN Move (270°)



Horizontal Movement :

LEFT Move (90°)
LEFTRIGHT Move (180°)
RIGHT Move (270°)



Clockwise Movement :

CLOCKWISE Move (90°)
ANTICLOCKWISE Move (180°)
COUNTERANTICLOCKWISE (270°)

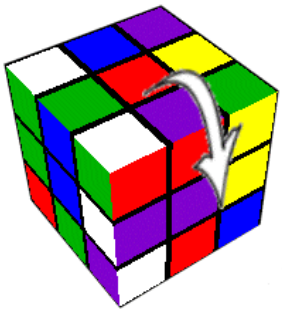


Figure 3: Possible Moves for a Rubik's Cube

Our initial approach was to take all possible moves and use them on each of the three rows, to give us a total of 27 possible moves to be applied to each state. This number of possible moves was larger than desired, so we set out to cut this number down. After a bit of research we discovered that most cubists do not in fact use the middle row as a move, so we substituted two outside moves for a single middle row move. For example UP 1 (an UP on the middle row) became a DOWN 0 + DOWN 2.

This too, however, proved to be problematic, as it extended our search technique to using 2 move groups instead of the single move search. There seemed to be no reason to not leave these move combinations to the next depth of the search, so we decided to remove these moves altogether, cutting our search space down to just 18 moves from each state.

Search Technique

Heuristics

Initially we had hoped to perform a single search, from start to goal, on any configuration of a Rubik's Cube. Because of the enormity of the search space, it is impossible to perform such a search in a complete manner, so some form of heuristic is required. As anyone who has picked up a Rubik's Cube knows, unless you have prior knowledge on solving Rubik's Cubes, there is no way to know how close you are to completing it at any stage.

The most obvious bases for heuristics on a Rubik's Cube are those using the number of cubies or faces in their correct positions. Plots of two such relations are given in figure 4. As you can see, these are of little use in predicting distance to the goal, and as of today there has been only one admissible heuristic created for the Rubik's Cube [4].

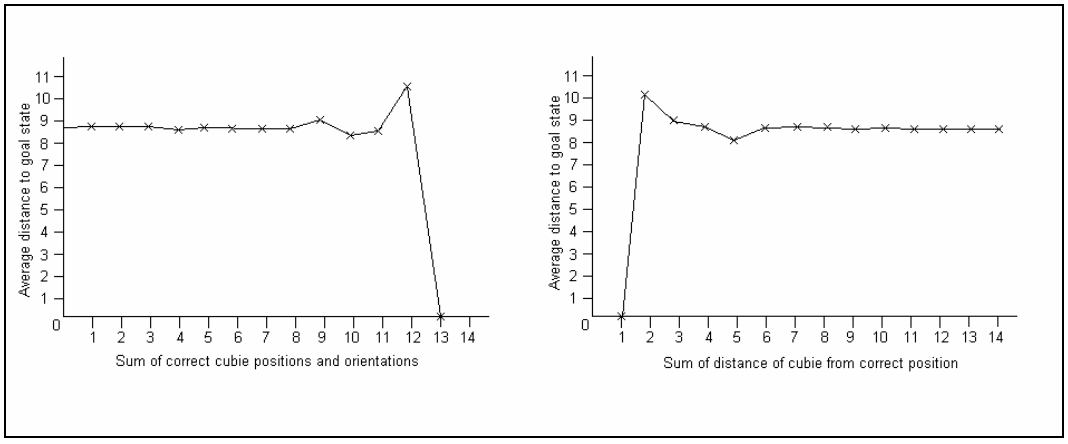


Figure 4: Two impossible heuristics for the Rubik's Cube

The idea behind this admissible heuristic, called the Center-Corner Heuristic, is to compile a hash table that stores the minimum length solution to solve every pattern of corner cubies, another hash table also stores up to six of the edge cubies. Korf [5] estimated that solving a problem of depth 18 it would take over 250 years to solve without the heuristic, as opposed to as little as four days using it (note that this research was conducted in 1997, so times may be different on modern computers).

Sadly, we did not have the luxury of being able to run 4-day tests, so this heuristic too was not a feasible option. Instead we focused entirely on applying various constraints to the search, which are discussed below.

Constraints

Several types of constraints were applied to the problem. Some of these provide speed increases; others also remove portions of the search space.

State Constraints

Every time we applied a move to a state, we come up with a new state. If we have already visited this state during the search then we do not want to continue down this path, as we have already progressed from here. So we needed to store each state as we visited it, to know whether we are looping back on ourselves and hence producing redundant results.

Move Constraints

As we progress through our search space, we found that we had a lot of moves that provided not only a state we had reached previously, but would *always* come up with an existing state. In other words moves such as UP 0 + DOWN 0, which will leave us in exactly the same state that we started with. With each step, if we searched through these states, the space to search is enormous, and provides no gain. So we provided constraints to prune these paths, hence narrowing our search space, so that we would only move in a forward direction taking us into a state that we had not previously visited.

While testing all possible states, and enforcing state constraints, we discovered that moves of depth 2 were all that was needed for us to sufficiently cover the redundant portion of the move space. We determined this by applying moves of increasing depth to the initial state, finding those moves that were either identities or equivalent. Using depth 2 constraints the following results were obtained: At depths of one, two and three no superfluous moves existed, and at a depth of 4 there were only 15. In the end we decided that the slight gains in coverage to be had from using larger depths were not worth the expense that it would incur in the lookup function.

Constraints encoding

The constraints were implemented in a file, in which constraints are listed on each line, in the format U # + D #. We also provided a means to iterate over all possible combinations, so that we wouldn't have to type out every single constraint. This was done using the wildcard *, which corresponds to each of the rows of the cube, i.e. U * + D * ➔ U 0 + D 0, U 2 + D 2.

Because of our discovery that the constraints we were going to remove would only reach a depth of 2, we decided to encode the actual constraint lookup using a hash table. The hash table index was an integer, which represented a unique value for each 2-move combination. This greatly improved search speed over our initial encoding which was simply using an unsorted vector of constraints. If we had wanted to go to greater

depth with our constraints, this method may not have been feasible, as the production of a hashing function would become rather cumbersome.

A full list of the constraints file is given in Appendix I.

The Petrus method

In our approach, we broke the goal into a series of sub goals, these sub goals were obtained from Lars Petrus, who has defined a set of steps for solving the Rubik's cube.

Lars Petrus

Lars Petrus is a *world-class* Rubik's cube player, who has helpfully listed on his very own web site, a step-by-step approach to solving the Rubik's cube. He boasts that his approach is much better than the typical 'Layer by Layer' approach that is used by most people who have not studied the cubes to the same extent.

We have used his steps to provide our solver with 'Sub Goals' that we allow our Rubik's World solver to search for, to give ourselves sub problems, and basically breaks down our search space. So although our approach will not give an optimum solution to solving the Rubik's Cube, we believe that it will give us a good approximation of an optimal solution, which is sufficient for the bounds of our assignment.

The method

Lars breaks the cubes solution down into steps, for which he then describes approaches on how to solve them, plus special 'tricks of the trade' that he has learned through his experience with the cube, and different methods he has learned off others. In our implementation, we tried at first to include these special tricks / moves that he provides for us, but in the end we found that we were much more likely to find a better solution not using these approaches. We instead used a full search from each sub-goal, which is guaranteed to find the shortest path between these goals.

Step 1

The first step is solving a 2x2x2 square, based around a corner.

Step 2

This involves extending this 2x2x2 square into a 2x2x3 oblong.

Step 3

Next we solve the cube for bad edges, Lars defines the bad edges as 'The idea is to build the entire cube now by just turning the 2 free sides. If you try to do that, you'll discover that some [bad] edges are twisted the wrong way'. So this step is to re orient these edges such that they end up in a position that is not *sub-optimal*.

Step 4

This step can be broken down into 2 separate parts, the first part Lars suggests is attempting to solve an 'L' shape.

The second step was solving 'two layers' which involves having our L transformed into a solution of the bottom half of the cube.

Step 5

'Now it's time for the endgame. Here we do not think. We recognise patterns and apply rules' – Lars Petrus.

The final steps that Lars presents are merely patterns upon which different rules could be applied to achieve the end goal (a solved Rubik's Cube).

Sadly we were unable to reach this step with our solver. This is the stage at which it would have most likely generated the greatest performance increase over a normal human being. Where a human would just see a pattern, our solver can work through every solution and get to the next goal, providing the optimal solution for this sub goal.

Sub-Goals

For each of the steps in Lars' solution we created a sub-goal to search to. As with most parts of this project, this process went through several evolutionary stages, which we shall discuss below.

Initial Sub-Goal Encoding

Initially we wanted a simple, comprehensive method of encoding the possible sub-goals to search for. Our first attempt was certainly comprehensive, but not particularly simple. For each step, we provided every possible combination of states that could be a permissible sub-goal. We then performed pattern matching with the previous sub-goal, and were left with a small set of goals to search for.

The encoding included a set of all possible face configurations, and then a series of valid cubes using these configurations. An example of this is given in figure 5.

```
2x2x2
01-2--,23--0-,1-03--,3-2-1-,-0-0-2,-2--20,--11-3,--3-31
7??7??777
???7??777
77??7??7?
777??7??7
2x2x3
6107--,732-6-,45-20-,06-4-2,27--40,5-431-,1-65-3,3-7-51,-016-7,-23-76,-4-024,--5125
7??7??777
???7??777
77??7??7?
777??7??7
7??7??7?7
???7??7?7
???7??777
77??7??7?
'L'
85226-,93436-,a1471-,b5070-,28-044,49-240,4a-402,0b-424,5-8351,3-9155,1-a535,5-b513,6608-
2,6169-3,106a-7,061b-7,273-86,327-96,737-a1,772-b0,-41628,-05639,-2517a,-4307b
7??7??777
???7??777
77??7??7?
777??7??7
7??7??7?7
???7??7?7
???7??777
77??7??7?
7??7??7?7
???7??7?7
????7??7?
????7??7?
```

Figure 5: Initial encoding of the sub-goals

As you can see, the encoding was quite reasonable for the initial sub-goals, but as the search deepens, goals become increasingly more complex. By the time the third sub-goal was reached there were 24 possible goals to be aiming for, but only four of these would be used on any given search. Another disadvantage was in the fact that we were unable to encode the fixing of the edges in this format.

Our solution was to dispose of this method entirely, and introduce some spatial knowledge into the solver, allowing it to determine sub-goals independently.

Spatial Reasoning

Knowledge

There were three separate pieces of knowledge the solver has access to. The first is a list of all eight pairs of squares that form an edge cubie. The second is the set of 3-tuples of the faces that form the eight corner cubies. The final piece of knowledge was the set of what are known as ‘layers’ of the cube. These are the 3 central lines encircling the cube, one for each dimension. These are shown below in figure 6.

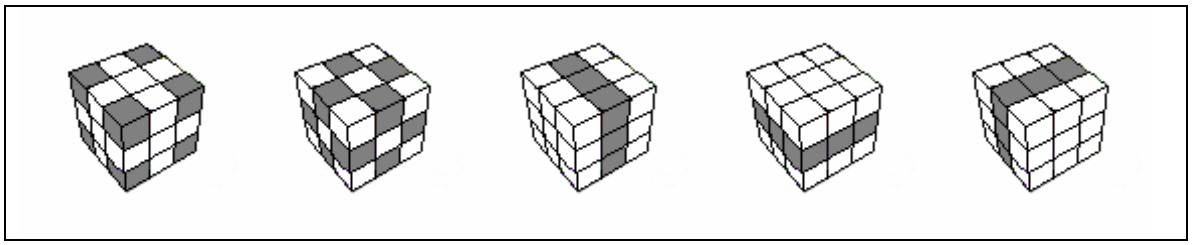


Figure 6: Corners, edges, and the three layers

These three notions are sufficient to completely cover the structure of the Rubik's Cube, so any reasoning methods need use only these facts.

Reasoning

The reasoning components were devised to remove the *major* headache of working out all possible goal permutations, which was required in our initial encoding. The functions we encoded are run on partially filled goal states, and are as follows:

- Determining if two corners are adjacent
- Determining whether a given edge is correctly oriented
- Listing all squares adjacent to a given square
- Listing all squares adjacent to a corner
- Finding all corners adjacent to n filled corners

Further methods can be implemented as sub-goal requirements increase, but for now this is sufficient.

Language

We replaced the contents of the initial sub-goal file with a simple interface to out reasoning methods. For each sub-goal, one line was given to describe how to compile it. The first token defines the requirements of the sub-goal, which also determines how many goals exist at that step. Each successive token defines an action to perform on each of these goals. For example, the sub-goal from 2x2x2 to 2x2x3 is defined as follows:

addAdjacentCorners1, addAdjacentSquares

This says to first take all corners that are adjacent to one existing corner (whatever we achieved in finding the 2x2x2) which will give three possible goals. For each of these we are then told to add all adjacent squares, which fills out the corner to make it a full 2x2x3.

The final encoding is given in figure 6 below.

```
2x2x2
addAdjacentCorners0, addAdjacentSquares
2x2x3
addAdjacentCorners1, addAdjacentSquares
```

```
Fix bad edges
allOtherEdges, correntEdges
'L'
addAdjacentCorners1, addAdjacentSquares
2 layers
addAdjacentCorners2, addAdjacentSquares
```

Figure 6: Final sub-goal encoding

Final Process

Without a heuristic, we are left to do a full search between sub-goals. We used a form of bi-directional breadth-first search. The search process begins with an initial search from the goal state(s) to a set depth. This search was fairly shallow, as goal comparisons are a more computationally difficult process than those between full states, but it still enabled us to remove the last, most expensive, stages.

After each sub-goal was achieved, the final state was used as the start state for the next step. Several goals were produced by the reasoning component,which the backward search expanded to a large number of possible goal states and their associated moves. The primary search from the start continued until it reached a goal. At this stage, the current move was appended to the overall solution, plus the inverse of the move from the sub-goal to that point. And then it begins again.

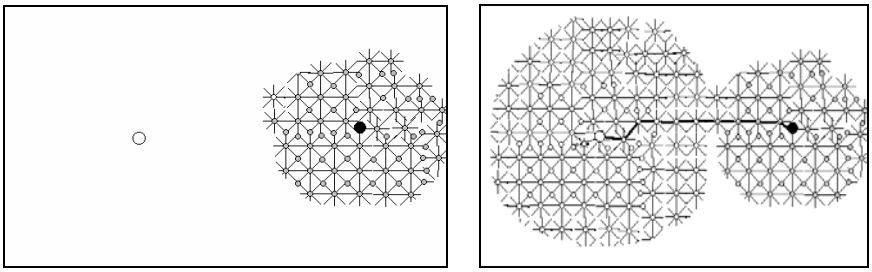


Figure 7: Process of sub-goal search

Overall, this method performs fairly well. The largest problem is the fact that the search is constrained to a human-defined set of steps. This means that while each sub-goal search is optimal, the solution to the whole search cannot be guaranteed to be optimal, in fact it is almost certainly not.

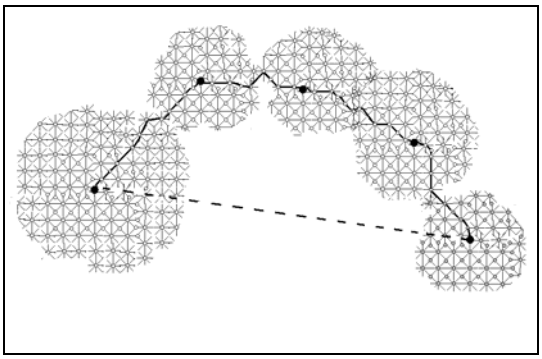


Figure 8: The overall search

Conclusion

Results

The Rubik's Cube is an interesting problem. Although it is not overly difficult to solve [6], as soon as you attempt at a knowledge-free optimal solution, many problems arise [5]. Firstly, we assess the success of the individual sub-goal searches. Here we performed to expectation, producing consistently shorter solutions than Lars Petrus, whose method we based our sub-goals on. On the later steps we were unable to make comparisons, as our search method was unable to consistently find solutions to these sub-goals. A graph of our solver's (squares) and Lars' (circles) average solution lengths on each step is given in figure 9.

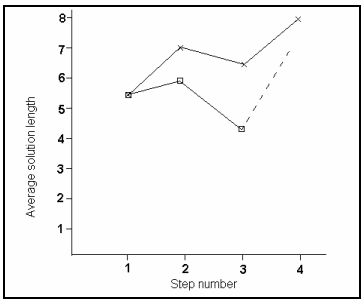


Figure 9: Lars vs. The Machine

Our solver performs as good as or better on any given step, although clearly more work is needed, as it is unable to find solutions of depth greater than around eight moves. The other area that we should consider is speed. Partial solutions of up to about 20 moves were the limit, but these were generated in times of around one minute. While these are not optimal paths, it is certainly an improvement over the four days reported by Korf [5].

Future work

From the outset it was going to be a difficult task to complete a solver in the three weeks provided for this project. Nevertheless, the approach we took was thorough and meticulous. At each step, any number of revisions were made, until we were satisfied that it was the best solution. This has left us with a solid foundation on which to base future work.

The area into which we believe the greatest benefit may be achieved is that of sub-goal heuristics. While it is possible to continue encoding sub-goals until the steps are short enough for the solver, there is little to be gained by doing so. Having analysed the Rubik's Cube and the methods by which people solve them, we believe that it may be possible to create a heuristic, or at least decompose goals, based on similar spatial knowledge and reasoning to that used in our sub-goal generator.

Rather than viewing solutions as the act of putting cubies into their correct positions in the cube, it is possible to view them as putting cubies in correct positions *relative to each other.* This would enable each goal to be broken into a number of sub-goals that could be achieved in parallel, thereby allowing the problem to be solved through classical planning methods. To efficiently program this, it would probably be necessary to move to a cubie-based representation, though this does not pose too great a difficulty. Using this method it may even be possible to move towards planning on the problem as a whole, rather than as separate sub-steps. This is an exciting proposition, as all current methods are much more cumbersome, inefficient and inelegant, and elegance is of utmost importance in all recreational computation.

***'In its arranged state it suggests calm, peace, a sense of order, security . . . in sharp contrast to all that the working object means once it is brought to life, to motion. There is something terrifying in its calm state, like a wild beast at rest, a tiger in repose, its power lurking.'*- Ernos Rubik.**

Bibliography

[2] Bauer, Kevin. Data Structures for the Rubik's Cube. North Carolina State University, 1999

.

[6] Dan Knight's Rubik's Cube
http://benjerry.middlebury.edu/~knights/Cube/CubeInfo1.html#method

[5] Korf, Richard E. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. Fourteenth National Conference on Artificial Intelligence: 700-705.

[1] Miller, David Lee Winston. Solving Rubik's Cube Using Bestfast Algorithm and Profile Tables. State University of New York Institute of Technology at Utica/Rome, 1998.

[3] Petrus, Lars. Solving Rubik's Cube for Speed
http://ng.netgate.net/~mette/lars/cubedude/

[4] Priedities, Armand E. 1993. Machine Discovery of Effective Admissible Heuristics. Machine Learning. Vol. 12, no. 1-3: 117-141.