Contents

Tom Wilkinson

OCR NEA

Rubik’s Cube Solver

[Contents 1](#_Toc98970931)

[Analysis 4](#_Toc98970932)

[The Problem 4](#_Toc98970933)

[Computational Methods 4](#_Toc98970934)

[User Requirements / Stakeholders 5](#_Toc98970935)

[Similar Systems 10](#_Toc98970936)

[DeepCubeA 10](#_Toc98970937)

[Korf’s PDB Based Solver 14](#_Toc98970938)

[Online Virtual Rubik’s Cube 15](#_Toc98970939)

[Essential Features 17](#_Toc98970940)

[Limitations 18](#_Toc98970941)

[Solution Requirements 18](#_Toc98970942)

[Success Criteria 18](#_Toc98970943)

[Design 20](#_Toc98970944)

[Top Down Design 20](#_Toc98970945)

[GUI 21](#_Toc98970946)

[Human Algorithm Solver 22](#_Toc98970947)

[Cube Data Structure 22](#_Toc98970948)

[Solving the Cube Object 23](#_Toc98970949)

[PDB Solver 34](#_Toc98970950)

[Pattern Databases 34](#_Toc98970951)

[Hash Function 35](#_Toc98970952)

[Tree Traversal 36](#_Toc98970953)

[Validation 37](#_Toc98970954)

[Test Data 37](#_Toc98970955)

[White Box Testing 37](#_Toc98970956)

[Black Box Testing 39](#_Toc98970957)

[Development 41](#_Toc98970958)

[Human Solver 41](#_Toc98970959)

[Tests 58](#_Toc98970960)

[GUI prototype 1 67](#_Toc98970961)

[Testing 73](#_Toc98970962)

[Result 76](#_Toc98970963)

[PDB Solver 77](#_Toc98970964)

[Data Structures and hashing 77](#_Toc98970965)

[Pattern Database Generation 85](#_Toc98970966)

[Search Function 89](#_Toc98970967)

[Evaluation 91](#_Toc98970968)

[White Box Testing 91](#_Toc98970969)

[Black Box Testing 91](#_Toc98970970)

[Success Criteria 91](#_Toc98970971)

[Limitations/Maintenance and Future Development 92](#_Toc98970972)

[References 93](#_Toc98970973)

[Final Code 94](#_Toc98970974)

[Directory Structure 94](#_Toc98970975)

[Tests.py 96](#_Toc98970976)

[Cube.py 97](#_Toc98970977)

[Solver.py 98](#_Toc98970978)

[Common.py 99](#_Toc98970979)

[Hasher.py 100](#_Toc98970980)

[Ida.py 101](#_Toc98970981)

[generatePDBs.py 102](#_Toc98970982)

# Analysis

## The Problem

The Rubik’s Cube is the world’s top selling puzzle game, having 350 million cubes sold as of 2009. Solving the cube requires the application of a number of algorithms, ranging from less than a dozen unique sequences to almost a hundred for more advanced methods.

When people first pick up a Rubik’s Cube their first instinct is usually to start turning the faces of the cube randomly, hoping that it will eventually solve itself. Unfortunately, this method would almost certainly never work, as there are over 43 quintillion different permutations of the puzzle, so to solve it this way would take longer than the predicted lifetime of the universe. The next thing people think to try is solving a single face, which they often succeed in doing but neglect to consider that the edges of the face must be correct too in order to solve the 4 adjacent faces. This is usually when they give up and never touch a Rubik’s Cube ever again.

The ‘correct’ way to solve the cube (At least for beginners) is to execute a series of algorithms intended to solve the cube one layer at a time. The most basic implementation of this is known as The Beginner Method, and involves around a dozen algorithms, making it easy to learn but not particularly fast or efficient. The most common advanced method is called CFOP and consists of close to a hundred individual sequences that need to be memorized, however this method can result in solve times under 10 seconds, making it popular amongst those who solve the Cube competitively.

I intend to create an application that will display a simulated Rubik’s Cube, allow the user to manipulate the cube and have the capability to automatically solve the cube using several different types of AI. The variety of methods allows the program to be useful to a wider range of possible users as some will want to see the ideal move from a given state and some will want to see the implementation of a simpler method

## Computational Methods

The use of a computational approach is ideal for this problem as the process of solving a Rubik’s Cube mandates the use of several computational methods; to attempt it without pattern recognition, abstraction and algorithms would be ineffective (Doing 1 random move a second would take about 10^9 years) so my computational approach would allow for greatly decreased solve times vs those of a regular person, and some of the faster implementations will be able to compete with or even beat the top speedcubers.

My project will be suitable for decomposition, abstraction, divide and conquer and pattern recognition. Decomposition can be used as the program can be broken down into sections, the GUI, the solver and so on. These can be further decomposed into smaller parts, i.e. the solver can be broken down into the data structure for the cube and each step in solving the scrambled cube can be thought of as its own section. From here I can use divide and conquer to tackle each section one at a time. Moreover, the use of decomposition means I can test and get feedback on each part of the program before I incorporate it into the main program. This makes debugging significantly easier as I can test out each section of code independently and identify where issues occur without having to search through my entire program.

Additionally, abstraction can be used when representing the Cube as the internal mechanism offers needless complexity and has no effect on the user experience so does not need to be included. Abstraction can be applied to the graphics, as I can use a simplified version of them for testing the code, allowing me to see what is happening without having to settle on a final design, before moving to a more complex version for the final product.

Furthermore, I will be using pattern recognition in the code, as an integral part of solving the Cube is knowing when to apply certain algorithms and when to apply others, which can only be deduced by looking at the current state of the cube and matching it against known patterns, something which computers are increasingly good at.

Finally, I will be using algorithms throughout the code, for turning the faces of the cube both in the data structure and the graphics, and for the sequences of moves that must be executed in order to complete each step in solving the cube. This allows me to use pre-existing methods of solving the cube, rather than trying to compose my own methodology which would not be feasible.

## User Requirements / Stakeholders

There are several types of people who could use the program, some of which I have mentioned previously. The first is those who are learning how to solve the cube, who would benefit from the visual representation of the cube, as well as its ability to solve the cube using a method they are likely familiar with. Admittedly, beginners are less likely to use the program than those who are more experienced, however by offering functionality to them they will be familiar with the software when they get to a stage when it is more useful, increasing the chance they use my program later.

The program would be most useful to those who are familiar with solving the cube, or are looking for a better method, because getting better at solving the cube is quite an arduous process, and they would gain from the built-in solving methods, more so than those who are new to the cube or who are well versed in solving. They would be likely to use the virtual cube to test new methods, which saves having to find a physical cube which could be inconvenient.

As I mentioned previously, not everyone will find the software useful for them, however by giving it as much functionality as possible and providing a good user experience it should be appealing to both those who will be able to benefit from it as well as those who perhaps will not.

To determine the needs of my users, I made a survey on google forms and distributed it to my end users.

Chart, bubble chart

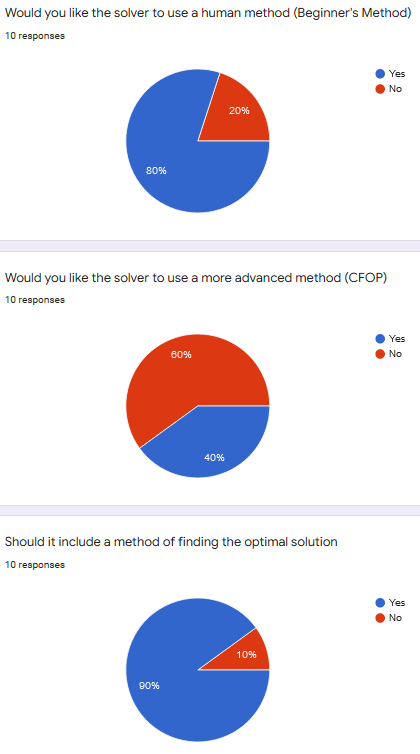
Description automatically generated

First I gather some demographics on who is taking the survey. Unsurprisingly, everyone who completed the survey has used a Rubik’s Cube before. This ensures everyone’s opinion is useful as those who haven’t used one will likely not understand the purpose of the program.

Chart, pie chart

Description automatically generated

Only half of them can successfully solve the cube, meaning there is a useful split of people who are experienced with the cube and those who aren’t.



To avoid including features that aren’t of interest to the users, I asked a series of questions about which solvers they would like to have. Most wanted to see the beginner method and a form of optimal solver. There was less interest in the more advanced CFOP method.

Chart, bar chart

Description automatically generated

Aesthetics are important, as they’re the first impression a user gets of the program. In this question I asked how detailed they would like the graphics, and the Cube in particular, to be. Most responded in the middle, or towards the less detailed side, which tells me that they are important, but not vital to a user’s enjoyment of the program. From here I know to try and maximise the visuals without taking too much time away from the main logic.

Graphical user interface, chart, application

Description automatically generated

Next I asked how the users would like to control the cube. Manipulation of the cube is the main way in which the user interfaces with the program, so it is important that the users like the implementation. Users responded positively to all the options presented, especially to buttons, so I will try to include all of them if possible.

Chart, pie chart

Description automatically generated

I also asked if they would like to see tutorials featured in the program, to help learn how to solve a Rubik’s Cube. The majority of users responded negatively to this idea, so I will not be including them.

For those using the software, the requirements are clear: it should be easy to use, visually appealing, be able to use solving algorithms to solve a scrambled cube quickly and efficiently. To achieve this the program will need a responsive GUI that is able to render the 3D model efficiently on a range of hardware.

In addition to the user’s requirements, I require the program to be functional on both Windows and Linux operating systems as I use both systems when writing the code and iterative testing is much more useful than final testing, so compatibility with both systems is essential.

## Similar Systems

### **DeepCubeA**

DeepCubeA is an AI developed by the University of California, which combines deep learning, classical reinforcement and weighted A\* searching to solve combinatorial puzzles such as the Rubik’s Cube and 15 puzzle (Agostinelli, et al., 2019). It can solve the Rubik’s Cube from any given solvable test case, finding the ideal solution in 60.3% of cases. I chose to investigate DeepCubeA because it is highly effective and open source which allows me to consider their approach when I design my program, and the accompanying journal article allows me to see *why* they chose to approach the problem in the manner they did.

The AI uses approximate value iteration to train a deep neural network (DNN) to approximate a function that outputs the cost to reach the goal (the cost-to-go function). It trains the DNN by starting in the goal (solved) state and randomly moving in reverse. After it has been trained, the cost-to-go function is used as a heuristic in a weighted A\* search.

Value iteration is a dynamic programming algorithm which iteratively improves a cost-to-go function *J*. Traditionally the functionis represented by a lookup table where *J*(s) is stored for all possible states s, however this does not work for the Rubik’s Cube as there are simply too many possible states. Therefore, DeepCubeA uses approximate value iteration, where *J* is represented by a parameterized function expressed via a DNN. The DNN is trained to reduce the error between *J*(s) and the updated cost-to-go estimation *J’*(s)

Where is the state obtained by performing action a in state s, and is the cost of moving from state s to s’, which is always 1 for a Rubik’s Cube. The program loops through each state s to update until , meaning it has found the most accurate function it is capable of.

The resulting algorithm is called ‘Deep Approximate Value Iteration’ (DAVI). The above equation is a one-step lookahead, meaning it only looks one move into the future unlike multi step lookaheads such as depth-N-search or MCTS which look at the subsequent possible moves too. In the article they note that those methods resulted in ‘at best, similar performance’ to the one step lookahead.

To ensure the model they created was valid for all the possible states, each test state was created by randomly scrambling the goal state times where is uniformly distributed between 1 and .

DAVI

**INPUT:**

B= Batch size  
 K= Max number of scrambles  
 M= Number of training iterations  
 C= How often to check if   
 = Error threshold

**OUTPUT:**

= Trained parameters

*initialize\_parameters*()

*e*

for m=1 to M:

X = *get\_scrambles\_states*(B, K)  
 for x in X:

Y =   
 , loss = train(, X, y)  
 if M MOD C = 0 and loss <   
 *e*

Return

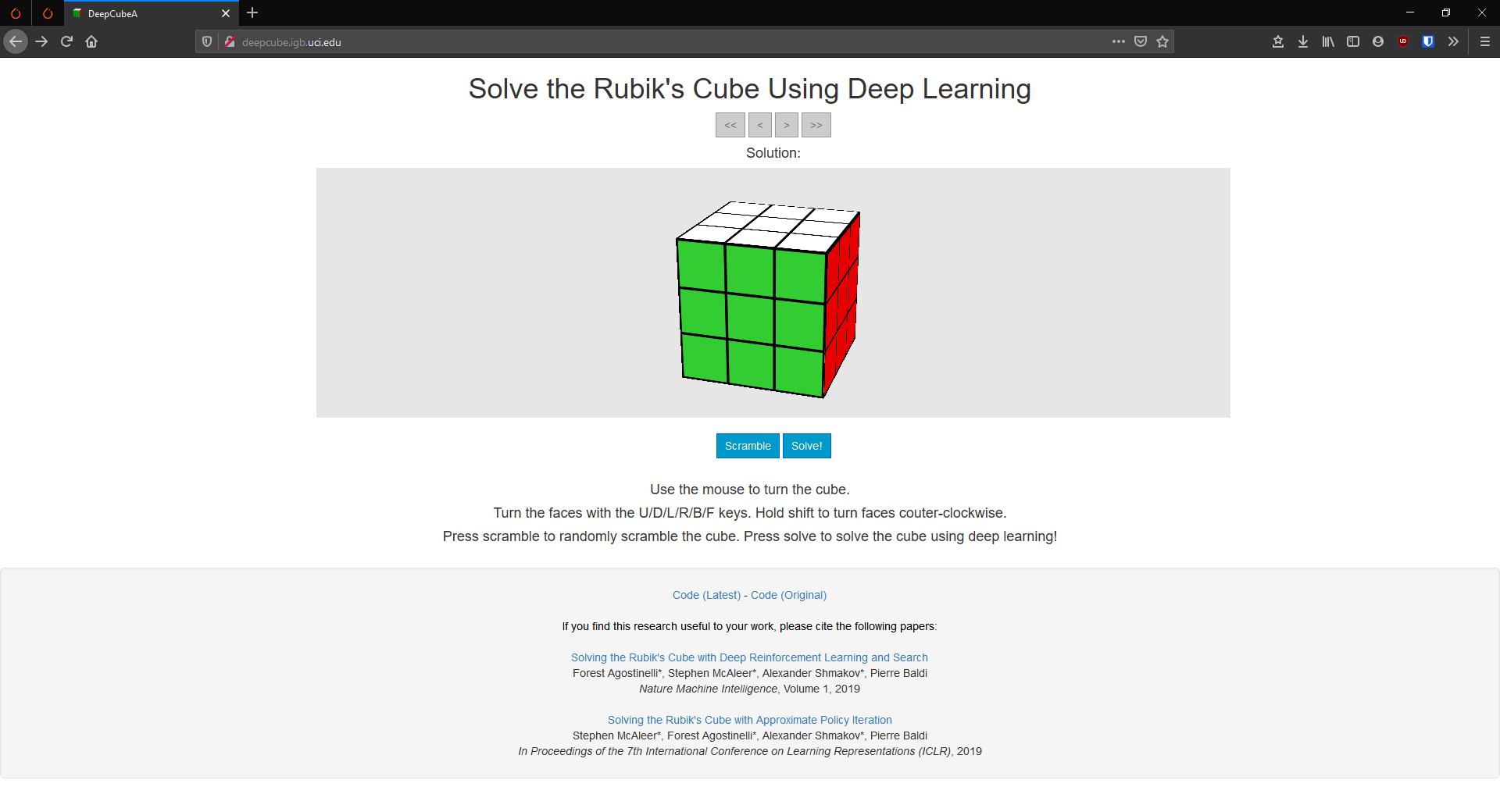
To train the DNN two sets of parameters are used, is the parameters being trained and *e* is the parameters used to obtain an improved estimation of the cost-to-go. The target for the DNN’s training is to reduce the mean squared error between its estimation of the cost to go and the estimation given by the equation on the previous page. Every C iterations the algorithm checks if the error has fallen below the set threshold, if so *e* is set to . The developers of DeepCubeA tried to update *e* every iteration but that caused performance saturation and instability, whereas by modifying *e* only when the error drops below the threshold it led to increased stability

After a cost-to-go function was determined, they used it as a heuristic in a Batch Weighted A\* search (BWAS). BWAS is a best-first search algorithm which iteratively expands the node with the lowest cost until the node associated with the goal state is selected. To determine the cost of each node they use the function where is a weighting factor between 1 and 0, g(x) is the distance between the starting state and x, and h(x) is the heuristic function used to determine how close each state is to being solved.

The A\* search creates 2 sets, OPEN and CLOSED. OPEN is where it iteratively removes the lowest cost node, which is then added to the set CLOSED, and any of its children that are not in CLOSED are added to OPEN. The search starts with just the starting node in OPEN and finishes when the goal node is removed from OPEN. DeepCubeA uses a variant of A\* called weighted A\* which uses a weighting factor, which trades potentially longer solutions for potentially less memory usage.

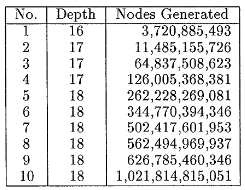
The most time-consuming aspect of their solution is computing the heuristic, but they mitigate that by expanding multiple nodes in parallel using several GPUs. Unfortunately, I will not be able to implement this in my program due to hardware limitations, however it may be possible to implement something similar using threading.

In summary, DeepCubeA uses Deep Approximate Value Iteration to train a Deep Neural Network which in turn is implemented as the heuristic in a Batch Weighted A\* Search to find the shortest path between the starting state and the solved state. By doing this it achieves 100% solve rate with over 60% being the shortest possible solution.

So far as I can tell, DeepCubeA does not natively include a GUI, however they have an example of it running on a webserver which includes a rather simplistic GUI with minimal animation or detail. I would like my software to have significantly more detailed graphics than the website’s, and I plan to have it be an all-in-one application rather than relying on an external website 

I intend to use a similar approach to DeepCubeA, using a DNN and A\* search, however due to time and hardware constraints I will develop a simpler system, that will still use DAVI but generate a less precise heuristic as I do not have a powerful enough computer to train a model to the same standard within a reasonable timeframe. I will also use a lower batch size to reduce hardware requirements and hopefully increase the speed of the program.

### **Korf’s PDB Based Solver**

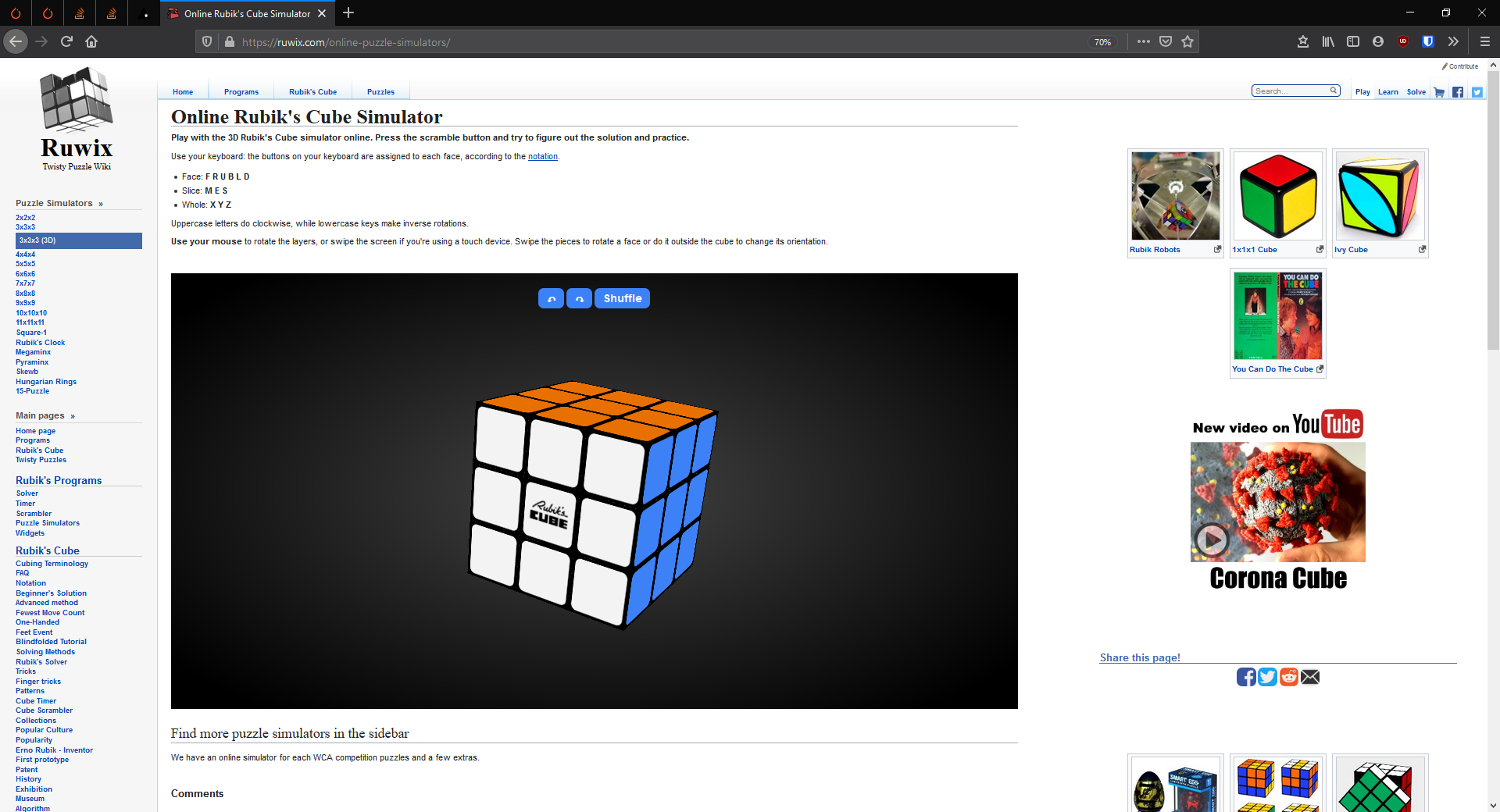
In 1997 Richard E. Korf published *Finding Optimal* *Solutions to Rubik’s Cube Using Pattern Databases* (Korf, 1997)in which he proposed using ‘Pattern Databases’ to act as the heuristic of an A\* search. A pattern database, or PDB, is a large memory-based lookup table which stores the exact number of moves required to solve various subgoals of the problem. In this case, Korf used 3 PDBs, one for the 8 corner pieces of the cube, one for 6 of the 12 edge pieces and the final for the other 6 edge pieces. The use of 3 separate databases is required because the total number of different states that can be reached from a given state is 4.3e19 which is too large to deal with, as the PDB would be exabytes or larger. On the other hand, the 3 separate PDBs use approximately 82MB. The database for the corners uses about 42MB of disk space and the two for the edges occupy 20MB each (More recent implementations will have 7 edge pieces in each of the PDBs requiring 500MB rather than 40. Using 8 edges would require around 4PB). It took Korf about an hour to generate the tables in 1997, so it is safe to assume that time will be significantly shorter now.

Korf’s workstation could generate and evaluate ~700,000 nodes per second, meaning it took several hours to compute solutions of depth 16, days for depth 17, and depth 18 “should take less than 4 weeks”. Using 7 edge PDBs will reduce this somewhat, and faster computers will further reduce the time required, but for high depth solutions it will still take a considerable amount of time.

To find the solution to the cube, Korf used an Iteratively Deepening A\* search (IDA\*), which is a depth-first search and looks for solutions of increasing length in a series of iterations, using a lower bound heuristic to prune branches if their length exceeds the current iteration bound

There are a few advantages to using PDBs, they are much simpler to implement than a DNN, and if implemented properly they can find the optimal solution 100% of the time, versus DeepCubeA’s 60%. These advantages come at a cost, as PDB solvers are much, much slower even on modern hardware, and the disk space required for the databases is larger than the space required for an AI implementation.

### **Online Virtual Rubik’s Cube**

The website ruwiks.com features a range of virtual Rubik’s Cubes for people to use. Unlike to previous two programs, this site does not contain a built-in solver, it is just a GUI that allows the user to manipulate the cube.

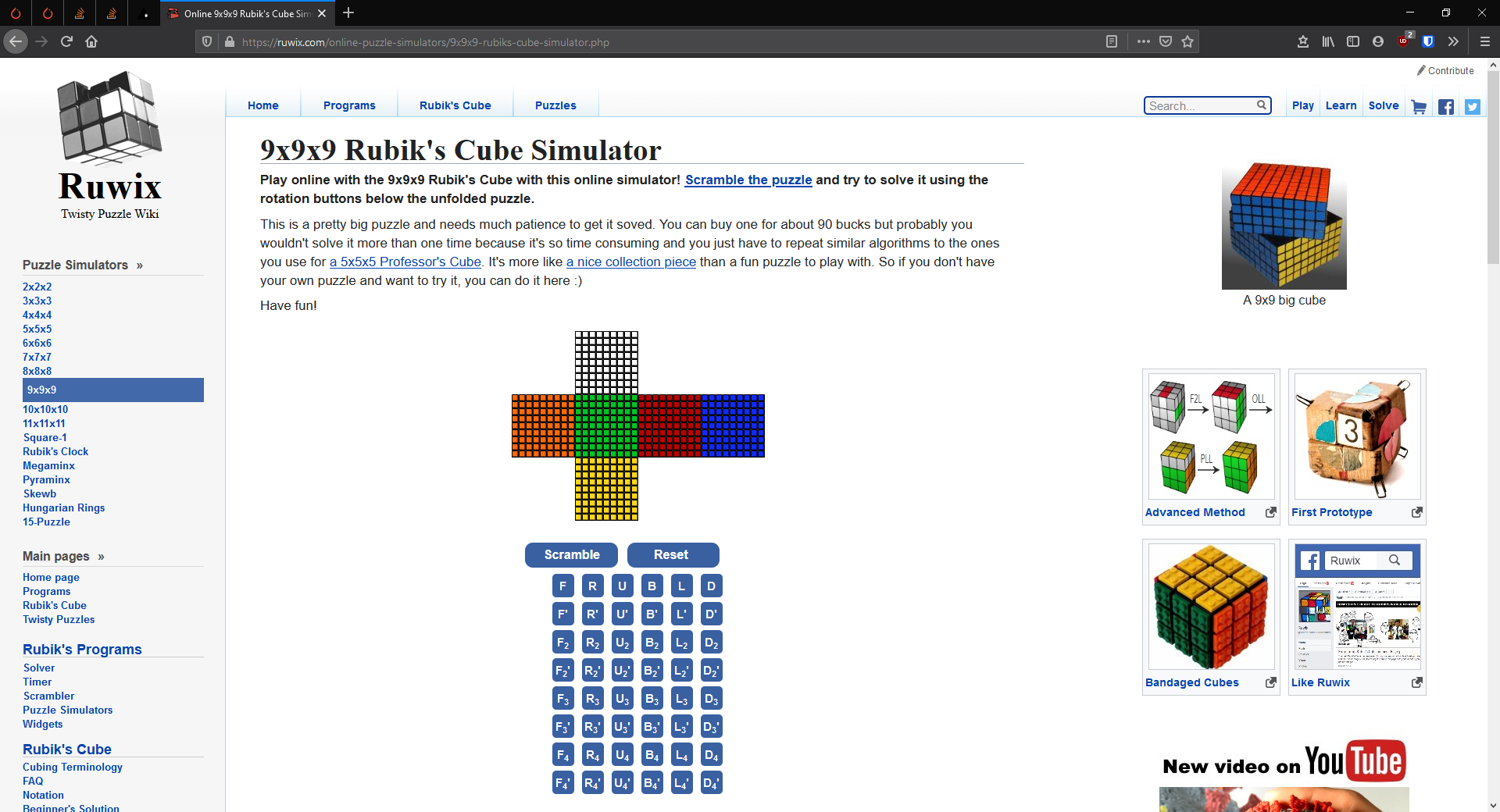
Adverts for other products related to them

Logo

Sidebar with other pages on the site

3D cube in the centre of the page

Manipulate the cube by click dragging the model

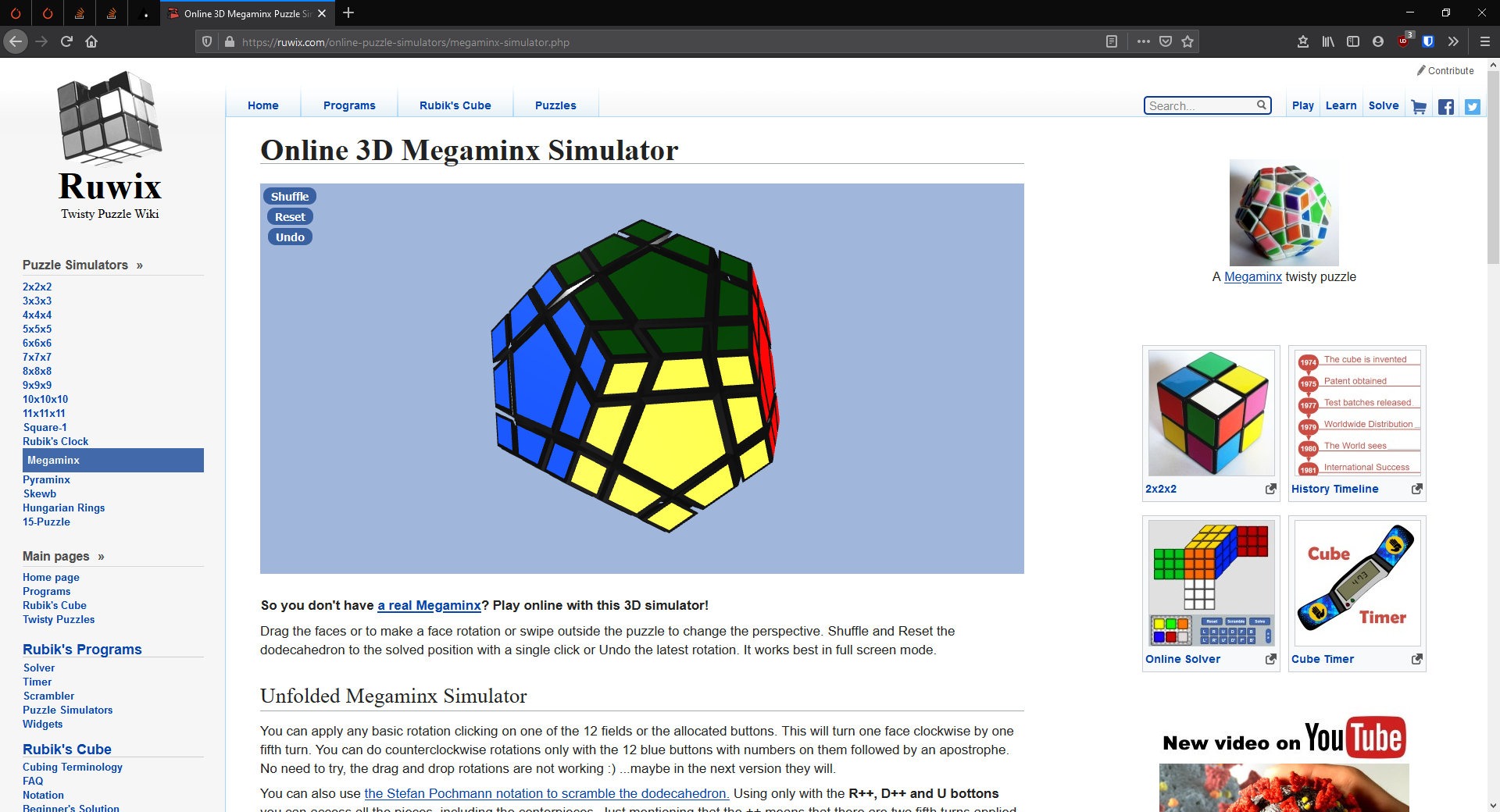


Description of the puzzle

Different way to represent the cube

Manipulate the cube with buttons

Picture of the cube in real life



Other shapes of cube also available

Ruwiks.com offers some good inspiration from which I can construct the GUI for my software, I like the 3D model of the cube and the subtle details that make it look more realistic than the DeepCubeA GUI, and I also like how it offers multiple ways to manipulate the cube. On the other hand, I do not plan to include sizes of cube other than 3x3x3 because I will not be including solvers for it except possibly the 2x2x2 as the PDB for the 3x3x3 corners would probably work for it. Additionally, I will not be including advertisements in the program because I do not intend to commercialise it, and the logo will be excluded because brand recognition is not required.

## Essential Features

From the investigation into similar systems I can see the features that are common between Rubik’s cube solvers and what I should include in mine, which I have summarised below.

A User Interface

It is important that my software be easy to use for those unfamiliar with it, and the easiest way to achieve this is via a GUI, rather than using CLI like DeepCubeA. By using a GUI, it makes the program more accessible to those who are not familiar with the more advanced side of computers, as most people have likely never had cause to open the terminal, let alone be familiar with how to use it whereas anyone who has used windows will be accustomed to programs using a graphical interface.

Manipulating the Cube

The user will be able to turn the faces of the cube using either their keyboard, click dragging the model of the cube or by using buttons displayed on screen, which will have an option to hide them if the user prefers not to have them using up space on the screen. The variety of options is essential because it allows the program to cater to more users, as each method has pros and cons depending on the user’s needs.

Automatic Solver

I intend to include multiple different methods of automatically solving the cube, as it will demonstrate to the user the different types of automation, and again allow the program to serve more people by providing methods with different pros and cons. The inclusion of a solver is essential as it offers the program more depth and functionality and makes the software more interesting.

Options Menu

The program needs to have an options menu, where the user can change the appearance of the program by showing/hiding the movement buttons, change the level of detail on the model depending on their hardware constraints, and switch the notation being used.

Custom Scrambles

The user should be able to load a scramble onto the system without having to move to it, this can be achieved by allowing the user to click on individual cublets to change them or similar. This allows the user to use the software to solve a real cube by inputting it into the system and generating a solve, because it is equally difficult to move the solved cube into a specific scrambles state as it is to move the scrambled state into the solved state, so if they cannot solve the cube themselves they will not be able to recreate it in the software using the legal moves either.

## Limitations

The biggest limitation of my proposed solution is the time required to solve the Cube with AI. Creating the PDBs takes a long time and a lot of computations, and the A\* search is also slow to complete. It is much quicker to use human algorithms, but this leads to solution lengths that are 5 or 6x larger than the AI produces. These limitations can be addressed to some extent, as the PDBs only need to be generated once, and by including more edges in each it reduces the time required for the A\* to complete, although this is only possible up to 7 edges as after that the disk space requirement would be measured in petabytes. There are other ways to speed up the PDB solver by applying group theory, but that is too complicated to be applied to the software, as to apply it enough to bring the solve times down far enough would increase the size of the PDBs to around 200GB (Agostinelli, et al., 2019). The human algorithm can be optimised by using advanced methods like CFOP rather than the methods used by beginners, although this only reduces the move count by a small amount compared to its overall length, whilst increasing the length and complexity of the code considerably.

A further limitation of the solution will be the size of the software, as between the PDBs and the code for the GUI, the program is likely to take up well over a gigabyte of space. This could cause a problem for users who do not have a lot of space left in their file system as there is no good way to reduce the storage requirement without severely impacting performance. It could be possible to generate the databases on the client side to save them having to download them, however for users on all but the slowest connections this is probably not worth the hassle. The solvers also require a large amount of memory whilst they are running, whilst most computers should not struggle too much, it could be a bottleneck and cause issues for those on less powerful systems. If the solvers begin to use virtual memory this could cause significant slowdowns as the algorithms need to read potentially billions of nodes, so any time loss will quickly add up.

## Solution Requirements

I will be using Python 3.8 to write most of the code, which has effectively no minimum system requirements, so long as it is being run on a platform that has an existing interpreter which all modern architectures do. The libraries I use will have individual system requirements which are hard to determine ahead of time, but rendering the GUI will require OpenGL support, which is present on all modern GPUs.

To run the solvers will require a substantial amount of memory, a minimum 4GB of ram should be sufficient but 8GB+ is recommended. Whilst producing the PDBs doesn’t require a powerful CPU, it will speed up the process to have one as it allows more nodes to be checked each second so less overall time required.

## Success Criteria

1. When run the product should open a GUI.  
   The GUI is essential as it will be the user’s gateway to the rest of the program. Without it the program would become almost impossible to use as the output of the solvers and the data structures used are hard to visualise.

This can be tested by running the main python script and waiting to see if the GUI opens successfully.

1. The GUI should display a 3D model of a 3x3x3 Rubik’s Cube.   
   The visual representation is the main way for the user to see what the computer is doing, so it needs to be properly displayed otherwise the user will feel disconnected from the program, and will struggle to interact with the software, as it will be impossible to set the cube’s state for the solvers, rendering the rest of the program essentially useless.  
     
   The model should be an accurate representation of a Rubik’s Cube, with the colours in the correct places and enough detail to look vaguely realistic. It should be rendered in a way that provides good performance on a range of hardware. Testing this is achievable by loading the GUI and observing the model, checking that it is properly rendered and responsive.
2. The user should be able to manipulate the faces of the cube using a variety of methods.  
   This means that users will be able to enter information into the program in a variety of ways, each of which has benefits and drawbacks, so rather than limiting the users to one method, it will allow them to use each style for what it is best for, and not have to use it when it isn’t the best.  
   The different methods should be click dragging the faces of the cube, which provides a close approximation to a real cube; Buttons located towards the side of the screen which allow easy entering of move sequences, but can sometimes be intrusive; and by entering the colour of each individual face one at a time, which is best for recreating scrambles from real life cubes.  
   This can be tested by trying to use each method and checking that all 12 possible moves can be achieved with each method, except the third method which should be tested by checking it is possible to select all 6 colours, and that the resulting cube is passed into the rest of the program correctly.
3. The software needs to include an options menu which allows the user to affect various aspects of the system.  
   This is required as certain features such as the movement buttons are only required some of the time, but some users may find them intrusive. It should also allow the user to alter certain aspects of the model to allow it to work better on slower hardware.   
     
   This can be tested by opening the options menu and ensuring that all the settings presented there affect the program in the intended manner
4. The program needs to include multiple methods of automatically solving the cube.  
   These methods will be a pattern database heuristic A\* search and a human algorithm (beginner method). The inclusion of multiple methods is done so that there is at least one method available on any hardware spec, and to showcase the pros and cons of each type of AI in a way that makes them useful for the user.  
   The AI will be tested by running a set of scrambled states of varying depth through the solvers and ensuring that they all give valid solutions.

# Design

## Top Down Design

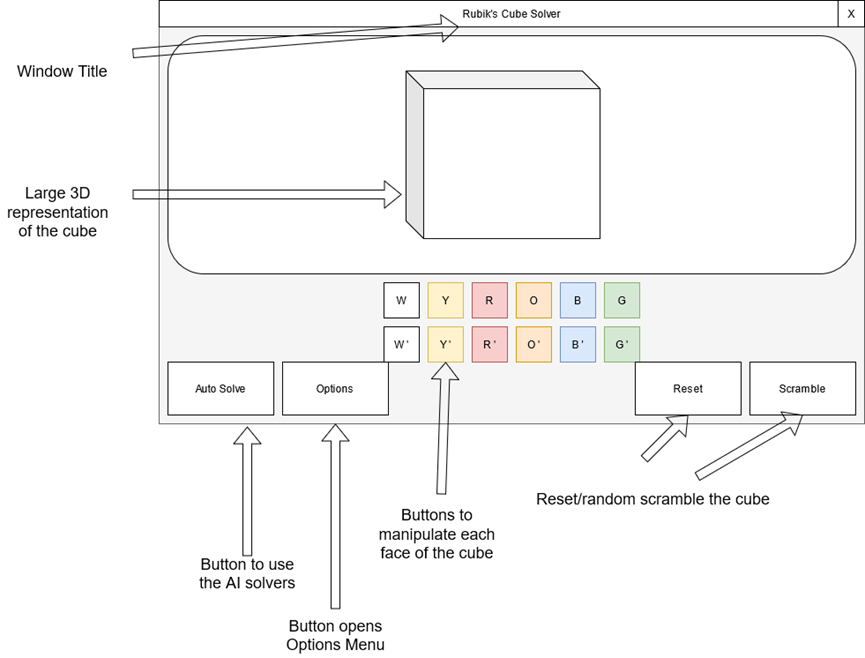
Diagram

Description automatically generated

The above diagram is an abstracted, decomposed representation of the program which breaks the software into its fundamental components.  
The use of a top-down design allows me to visualise the program both as a whole and as a collection of more manageable subsections, which I can develop and test independently before connecting them together when they are complete.

The program will be composed of 3 main sections, each of which will be independent of the others. The two solvers I have chosen to implement don’t need to interact with each other, so they can be produced and implemented modularly. The GUI will serve to combine the rest of the program. It will mostly be able to function independently and will be how the user interacts with the Rubik’s Cube.

## GUI



Visualization of how the main page will be split up.

All the program’s functionality can be quickly accessed from this screen, including the AIs and all types of cube manipulation. This makes the GUI both easier to navigate and easier to construct, as less unique pages exist.

The Auto Solve button will open a drop-down menu allowing the user to choose which method to apply to the current cube state. Having a single button open a menu is preferable to several individual buttons as it simplifies the UI and saves space.

The options menu will allow the user to change parts of the program, such as the method of controlling the cube.

The reset and scramble buttons both affect the cube state, either by moving it back to the solve state or executing a random series of moves on it.

There will be 12 colour coded buttons in the centre of the GUI which each correlate to a face on the cube and a direction. These buttons will be used as one of the methods for turning faces of the cube, so they need to be central and obvious to users, hence they are colour coded to make it more easily apparent which face they turn.

As well as the buttons, the window will feature a large space containing a 3D model of the Rubik’s Cube which will display the current state as well as being animated when a move is performed.

## Human Algorithm Solver

The human algorithm will use the Beginner’s Method to solve the cube. For this solver I will use object-oriented programming.

### Cube Data Structure

Text

Description automatically generated The cube will be represented using a class, with separate 2D array attributes for each side (multiple 2D arrays vs one 3D array will make the code more readable). Each of these arrays will be populated with characters representing the colour of the sticker in that section.  
The class also needs a way to store the previous moves executed on the cube, so that after the solver has completed the solution can be returned.

+Move\_history: char array

The class will also have procedures to turn a face of the cube.



Above is pseudocode for the rotation of the sides. (Double Click to expand)  
When rotating each side, there are 2 stages. First the face itself is turned, then the adjacent edges must be rotated around. Rotating the array itself can be done efficiently in python, either within the standard lib or using something such as numpy.array.rot90. The adjacent sides can be rotated using iteration to reduce repeated code.

### Solving the Cube Object

Diagram

Description automatically generated

The above diagram identifies the steps involved in the method I will use. Each box represents a step in the process that needs to be fully functional before the subsequent steps can be implemented. The diagram is heavily abstracted and does not show the implementation of each stage, as there is too much to communicate on a single diagram, so I will further decompose each stage below.

#### Create Daisy

During the first stage, the program needs to find each white edge, and move them onto the yellow face. The order they appear on the face is irrelevant. To do this, it needs to know which yellow edges already have white faces in them and be able to find where the white stickers are.

To achieve this, I will have a function which inspects every edge sticker on the cube to see if it is white, and then return a list of where the white stickers are. Pseudocode for the function is on the following page.

By giving the function the ability to ignore certain sides of the cube, it can be used to prevent the program trying to move sides that are already on the yellow face.



Also, there needs to be a data structure containing which yellow edges already have a white sticker in them. This could be implemented as an array, but I believe a dictionary would allow for better readability. The find function can then be run and the result iterated over to set the initial state of the yellow sides dictionary.

From here the program needs to find a white edge not already located on the yellow face, rotate the yellow face so that there is a free spot in the correct place, and then execute the correct algorithm to move it from where it is to the space on yellow. If the white edge is on the white face, it is simple to rotate yellow until the edge is above a free space, and then rotate the side twice to move it down. If it is on another side, then there is a different algorithm for each of the 4 possible locations on the side however it is reasonably trivial to determine which to use by inspecting the location of the face within the array.

This will be implemented using a while loop to perform the find -> orient -> move cycle until each yellow edge contains a white sticker.

#### White Cross

The next stage is quite simple, moving the white edges from the yellow face to the correct spot on the white face. Each edge cubelet has 2 stickers on it, in this case there will be a white sticker and either a red, orange, blue, or green one. For the cube to solve correctly, the white-red edge must be placed on the border of the white and red edges, the white-blue on the border of the white and blue edges and so on.

To implement this stage, I will again use conditional iteration. For each of the four sides red, orange, blue, green, I will check if the cubelet stored in that edge of yellow is the correct one, by confirming it still has a white sticker, and that the other sticker is the same as the colour of that side. If it is the correct one then the side can be rotated twice to move it into the correct location. The condition will be all four white edges containing white stickers. Pseudocode below.



In the pseudocode 3 dictionaries are used, the first stores the functions used to turn each side, the second and third store the stickers on the cubelets in the yellow edges. During each pass of the loop the latter 2 are updated as the yellow face has been rotated.

#### White Corners

The next stage in solving the cube is to position the white corners correctly, which completes the white face, and the first layer of the adjacent sides. The way I will do this is like the white cross by moving all the corners down onto the yellow face and then back up into the correct positions.

First the code needs to find where the white corners are. I will have a function that checks every corner to see if the cubelet contains a white sticker.

The next stage is to move any corners that are already on the white face down to the yellow face. This may occasionally cause a corner that is already solved to be moved, but it would vastly complicate the solver to prevent that, and as this solver isn’t going for the ideal solution it isn’t necessary.

The finder function returns an array of numbers representing the locations of the corners, which can be iterated over to see if there are any on the white face. If there are, then they need to be moved down to the yellow face. The algorithm to move them is simple, place it in the upper right of the cube and R’D’R will swap it with the corner below it. Pseudocode for the section is below. Double click to expand.



After this stage, all the white corners will be on the yellow face.

Next, the yellow face needs to be rotated until one of the corners is below the correct spot on white. Once there is a corner ready to move, it needs to have a series of moves executed to move it up. The series of moves being executed depends which side of the cubelet has the white sticker on. The pseudocode for this process is below. Double click to expand.



After this has finished, the first layer of the cube is completely solved.

#### Middle Layer

The next stage of the process is to solve the middle layer of the cube, i.e. the 4 edges that don’t have white or yellow stickers on them.

The process to achieve this is to rotate the yellow face until one of the edges is below the correct center. This makes a T shape of solid colour on the side. Then one of 2 algorithms is applied to move it into place depending on which side it belongs. (Double Click to expand the pseudocode below)



By rotating the yellow face and applying those 2 algorithms, it allows for completion of the second layer of the cube.

#### Yellow Cross

The next step in solving the cube is to create a cross on the yellow face. This is much simpler than the white cross because there are only 4 possibilities for the state of the cube.

In the best case, the yellow cross is already present,and this stage can be skipped.

The second possibility is that there is a line of 3 yellows going through the center of the face. In this case to resolve the stage the cube needs to be oriented with the line going left to right and the sequence *FRUR'U'F'*

The third possibility is an L shape through the center of the face, in which case the yellow edges need to be oriented on the far and left sides and the sequence *FURU'R'F'* Executed.

The final possibility is that there are no parts of the cross completed, simply a yellow dot on the face. In this case the line algorithm needs to be executed on one face, followed by the L algorithm on the opposite face.



#### Yellow Edges

Next, the white edges need to be oriented so that the second stickers are on the correct faces.

To do this, the yellow face is rotated until 2 of the stickers are on the correct sides. They will either be in an L shape or a line through the center.

If they are in an L shape, the correct stickers need to be on the back and the left sides, then *RUUR'U'RU'R'U'* is executed. Otherwise, if there is a line through the center then the cube must be positioned so the line goes from top to bottom and the previous algorithm is executed without the final *U’*, then the cube is rotated clockwise and the sequence is executed again to solve the section.

#### Yellow Corners

The penultimate step is to move the yellow corners to the correct location. The key algorithm being used at this stage is *L'URU'LUR'U',* which cycles 3 of the edges around.

The first step is to find a corner that is in the correct place. This can be done by constructing sets containing the 3 stickers on the cublets and comparing them to a premade set of the correct stickers. Using sets is helpful here as the order of elements within them doesn’t matter, and the cubelet could be in any rotation.  
If no corners are correct initially, then the sequence can be executed on any corner to produce one. If there is a corner correct then it needs to be positioned in the close right and the sequence executed until the others are correct too.

#### Yellow Rotation

The last step is to orient the yellow corners correctly. The algorithm used to manipulate this stage is *RUUR'U'RU'R'L'UULUL'UL* Which turns the two yellow corners on the right of the cube inwards.

There are 4 patterns which can appear at this stage. If 1 corner is correctly rotated then performing the above algorithm on any side will advance it into a more solvable state. If 2 diagonally opposite edges are correct, then running the sequence on one of the sides with the yellow sticker on will make it easy to solve.

After those have been dealt with, there are 2 possible states, either both stickers are on the same side, and look like headlights, or they are on opposite sides, and bear a resemblance to Perry the Platypus’s eyes.  
The pseudocode to deal with these states is below, where move(side) executes the previously stated algorithm. (Double Click to expand)



The logic to solve all the possible states is described in the following pseudocode.

After this stage has executed, the cube will be fully solved.

## PDB Solver

Diagram

Description automatically generated

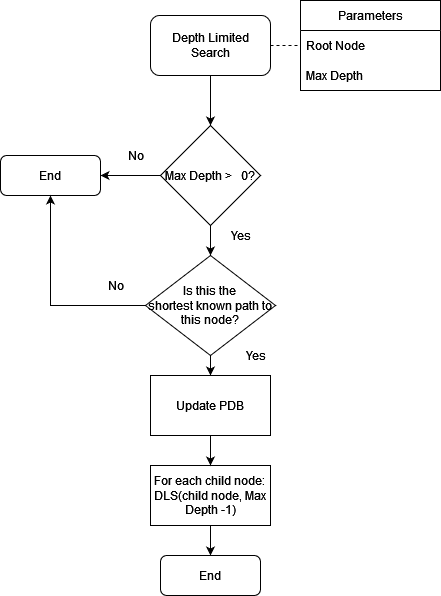
Above is a decomposition of the Pattern Database based solver. It consists of 3 databases which each store the distance to being solved of a subset of the cube, a method to represent the possible states of the cube as a graph and a way to traverse the graph.

Table

Description automatically generatedWhen the solver is given a cube state to be solved, it first needs to be turned into the correct data structure, which in this case will be 2 2D arrays, one for the corners and one for the edges. Each member of the array will have a number to identify which cubelet it is and a value to determine which way it is facing. This structure is used as it makes it easier to hash the cube state. It will also need a stack to store the move history so it can be returned after the solver has finished.

### Pattern Databases

The databases need to be populated before they can be used, and this will be achieved by creating a tree of all possible cube states with the solved state as the root node and each child node being the result of a single move executed on that state. Then an Iteratively Deepening Depth First Search (IDDFS or IDS) is used to traverse the graph and record the minimum depth required to reach each node.



Graphical user interface

Description automatically generated with low confidence

### Hash Function

To ensure maximum efficiency, the pattern databases need to be implemented as hash tables to make lookups faster, and to create a hash table requires using a hash function.

Because the hash tables store so many values, using an imperfect hash would cause a lot of space to be wasted and any collisions would slow down the program so a perfect hash function needs to be created.

#### Corners

The corners are stored as a 2D array of length 8, and each subarray is length 2 storing which cubelet it is and the orientation. Each corner can be facing one of 3 directions, so can be represented as the values 0 to 2. The cubelets can be enumerated 0 to 7.

The first step is to consider the permutation of the cubelets. The first corner could hold any of the 8 cubelets, then the second could hold any of the 7 remaining, then the third has 6 options and so on. The value of the final corner is determined by the first 7, so can be ignored. These properties can be utilised by subdividing the table into 8 sections and placing it into one determined by the first corner, then further subdividing that into 7 sections, which are determined by the second corner and so on. However, for this to work, it is important to consider which cubelets have already appeared, as if a cubelet appears in a previous corner, then it cannot appear again so all the cubelets after it should have their value reduced by one. Implemented in pseudocode it looks like so:



The second stage of the hash is to consider the orientation of the corners. This can be done easily by reading the corner orientations as a 7-digit ternary number and adding it to the value produced in the previous step. Again, the value of the last corner is determined by the first 7, so it doesn’t need to be considered.

#### Edges

The process for hashing the edges is almost identical to the corners, except that there are 12 edges in total, however only 7 hashed at a time. This means that the final value of the permutation matters and cannot be discarded. Furthermore, the edge cubelets can only be oriented in 2 directions, not 3, so they need to be interpreted as a binary number rather than ternary.

### Tree Traversal

The IDDFS visits each node in the same order that a breadth first search would but has rather than spacial complexity where is the depth of the graph and is the width of the widest layer. This makes it significantly more memory efficient as it only needs to go to a maximum depth of 11 moves, but the 11th layer of the graph would have 3,266,193,870,720 nodes (Korf, 1997) which is too many to store in memory. The increase memory performance does come at a minor time cost, as IDDFS must examine nodes higher in the tree several times, but due to the branching factor being high this proves to be trivial.

Next an Iterative Deepening A\* search is used to find the ideal path to the goal state. The pattern databases are used as the heuristic, where the heuristic value is the max of the values from each pattern database for the node’s state.

IDA\* is a form of iterative deepening search, where it performs a depth first search until it reaches a node where g()+h() is greater than the threshold. If all paths terminate before the goal node is found, then the threshold is increased to the smallest value that allows more nodes to be visited. Initially the threshold is h(root). When it finds the goal it returns the path it took to get there, which is guaranteed to be the shortest path.

## Validation

Due to the nature of the Rubik’s Cube, most of the combinations of stickers are not possible to solve. In each solver described above they need to be given a solvable state to have any hope of finding a solution, and if they are given something unsolvable, they may get caught in an infinite loop or return a solution that does not work. To avoid this, it is important to check the state is solvable before passing it to them. This will be achieved using a slightly modified version of the human solver as it executes quickly and will produce predictable results.

As there are no other opportunities for the user to input data (other than button presses) there is no other areas where input validation is required.

## Test Data

### White Box Testing

White box testing is where the tests are performed by someone with knowledge of the code behind the program and the ability to make changes accordingly.

The following is a test table used to ensure each section of the code works as expected.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | Expected  outcome | Justification | Actual Outcome | Changes |
| Unsolvable cube state to the solvers | Catches the error and displays a message to the user | Passing an invalid state to the solvers would break them |  |  |
| Solvable cube state to the solvers | Solves the cube and displays the solution to the user | If the solution isn’t correctly solved and displayed for the user there would be no reason to use the software |  |  |
| Cube state into hashing algorithm  (PDB Solver) | Unique hash value given to each corners state, in range 0 to 88,179,840 | To ensure maximum memory efficiency there needs to be a perfect hashing algorithm, meaning no collisions and no unused slots in the hash table |  |  |
| Cube state into hashing algorithm  (PDB Solver) | Unique hash value given to each first 7 edges permutation in range 0 to 510,935,040 | To ensure maximum memory efficiency there needs to be a perfect hashing algorithm, meaning no collisions and no unused slots in the hash table |  |  |
| Cube state into hashing algorithm  (PDB Solver) | Unique hash value given to each last 7 edges permutation in range 0 to 510,935,040 | To ensure maximum memory efficiency there needs to be a perfect hashing algorithm, meaning no collisions and no unused slots in the hash table |  |  |
| Use hashed state as index for PDB | Return the correct distance from solved for that permutation | If the wrong value is returned it will result in an invalid heuristic |  |  |
| Solvable cube state passed from main program to PDB solver | Cube state transformed into correct structure | The state passed from the main process will not be formatted correctly for the solver so it needs to be turned into the correct structure before the rest of the function can run |  |  |
| Solvable cube state passed from main program to PDB solver | Ideal solution is found and returned to the main program | Solution needs to be returned to the main program so that it can be displayed in the GUI to the user.  PDB solver should find the ideal solution to a given state. |  |  |
| Solvable cube state passed to Human solver | Solution found and returned to main program | Same as above, however human algorithm is not expected to find ideal solution. |  |  |
| GUI displays the cube correctly | 3x3x3 Rubik’s Cube with correct colours and size | The model of the Cube should be reasonably similar to the real thing. |  |  |
| Turning a side of the cube displays on the GUI correctly | The correct 9 cubelets rotate 90 degrees along a circular path about the center of the middle cubelet. | If the wrong cubelets rotate, it will disassociate with the underlying data structure, causing the wrong cubelets to move on every subsequent rotation. |  |  |
| Calling the solvers from the GUI works correctly | When the button is pressed, the current cube state is passed to the solver and evaluated, then the solution is executed on the cube | The GUI is how the user interacts with the program, so if the solvers can’t be accessed from it, the user won’t be able to run them. |  |  |

### Black Box Testing

Black box testing is performed by someone who does not have knowledge of the code

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input | Expected Outcome | Justification | Actual Outcome | Changes | Success Criteria |
| Load the program | Program opens to show main GUI window | Ensures the program loads as intended |  |  | 1 |
| Load the program | Main GUI should feature a 3D model of the Rubik’s Cube in its solved state | Ensures the model is working |  |  | 2 |
| Use movement buttons | The correct face should turn in the correct direction for every button | Ensures the model updates correctly for every input |  |  | 3 |
| Click and drag a face of the cube model | The face should turn in the direction it is being dragged | Ensures that method of affecting the cube is working |  |  | 3 |
| Click a sticker on the cube | The sticker being clicked should cycle colour each time it is clicked | Ensures that method of affecting the cube is working |  |  | 3 |
| Click the autosolve button | Should open a menu allowing the user to select a solver | Ensures users have access to the solvers |  |  | 5 |
| Select the human algorithm from the autosolve menu when the cube is in a solvable state | The human solver should be used to find a solution to the current cube state | Ensures the human solver is being called correctly and functions properly |  |  | 5 |
| Select the human solver from the autosolve menu when the cube is in an unsolvable state | The GUI should show a popup message saying that the cube isn’t currently in a solvable state | Ensures that unsolvable states are being caught |  |  | 5 |
| Select the PDB algorithm from the autosolve menu when the cube is in a solvable state | The PDB solver should be used to find a solution to the current cube state | Ensures the PDB solver is being called correctly and functions properly |  |  | 5 |
| Select the PDB solver from the autosolve menu when the cube is in an unsolvable state | The GUI should show a popup message saying that the cube isn’t currently in a solvable state | Ensures that unsolvable states are being caught |  |  | 5 |

# Development

## Human Solver

Calendar

Description automatically generated*Cube.py*

This is the definition of the cube object being used by the human solver, this image shows the attribute definitions. Each side is represented as a 2D array of characters representing the sticker on that face. Whilst it would be more memory efficient to use integers rather than characters to represent each sticker, this would make the code significantly harder to read and debug as it would be harder to visualise.

A picture containing table

Description automatically generated

A picture containing table

Description automatically generated

All the elif statements are the same indentation but the screenshots don’t line up

This is the method for rotating a side. For any given side to be rotated, first the side itself must rotate 90 degrees, then all the stickers around the edge must move round.

Text

Description automatically generated with medium confidence

A slight abstraction of cube.turn\_side() which iterates over a string to execute a series of moves

Text

Description automatically generated

Method to check if the cube is fully solved

Text, letter

Description automatically generated

Execute a random series of moves on the cube to scramble it. 80 moves is more than enough to ensure it is.

Text, letter

Description automatically generated

A method only used for debugging, this outputs all the sides in a human-readable format

A picture containing logo

Description automatically generated

A (not very pythonic) method for resetting the move history.

Text

Description automatically generated

Unit tests at the bottom of the program help to ensure it is operating correctly.

*Solver.py*

Graphical user interface, text

Description automatically generated

A picture containing text

Description automatically generatedFirst the class is imported. Solve() acts as a wrapper for the different stages of solving the cube, which ensures they are carried out in the correct order.

The first step is to create the daisy.

The preceding underscore declares the subroutine as private, and prevents them from being imported by  
from cube import \*  
as there are very limited circumstances where these procedures would need to be used independently.

The function find() is used to locate the white stickers on the cube.

Text

Description automatically generated

This section of the program maintains a dictionary of which yellow face edges have white stickers in them, and a method to turn the yellow face whilst updating the dictionary

Text

Description automatically generated

First check which parts (if any) are already solved.

Table

Description automatically generated with medium confidence

Move any white edges that are on the white face down to an empty spot on yellow.

Whilst it would be slightly more efficient to leave them there, it would also introduce a lot of complexity by having to get them in the right order and correct places.

A picture containing text

Description automatically generated

A picture containing text

Description automatically generated

Then move any other white edges from the side faces down to the yellow face. The algorithm to be executed on the cube varies by where on the face the edge is, and each algorithm puts the edge in a different spot so it is important to make sure the cube is set up correctly before the algorithms are executed.

After this has completed, there will be a daisy on the bottom meaning

cube.yellow == [[x, w, x], [w, y, w], [x, w, x]] where x can be any colour.

Text

Description automatically generated

The next top-level function moves the white edges up to the white face. It does this by checking each edge of the yellow face to see if it lines up with the white and moving if it does. Then it turns the yellow face to make new pairings. Originally the break condition was at the top, however this led to an extra yellow turn being performed, so I moved it to an if statement within the loop.

Text

Description automatically generated

The next top-level function moves the white corners to the correct place on the white face, without moving the white edges out of place. The first step in doing that is finding where they currently are, so the 8 corners on the cube are enumerated and if the corner has a white sticker, then the id of the corner is stored.

Text

Description automatically generated with medium confidence

Again, by moving all the white corners to the bottom before moving them back up greatly simplifies the program, and even if they were moved in situ there would only be a slight reduction in the number of moves performed.

Text

Description automatically generated

Subroutines for the next stage of execution need to be defined. Move\_up() is used to move a corner corner from yellow to white, by applying the correct moves to ensure the white sticker faces upwards.

Stage\_solved() checks if all the corners have been moved to the correct places by looking at the non-white stickers present

A picture containing graphical user interface

Description automatically generated

Move the corners to the correct places using the subroutines defined previously.

Now the white face will be completely solved, with the edges and corners in the correct places.

Text, letter

Description automatically generated

On the second row, the edge pieces will need to be put onto the yellow face, and then moved to the correct 2nd layer edge.

Text

Description automatically generated

The first step is to flip any of the edges that are already in the correct place but are the wrong way around.

Graphical user interface

Description automatically generated with medium confidence

If two of the edges are in each other’s spots, it is simple to fix by moving one of them out, and then moving it into its own spot, which will move the second edge to the top so it can be put into where the first edge was.

The code looks for edges which are ready to be moved down and then puts them into the suitable location and rotates the cube to make new pairings. If there are no edges left to move down, but the stage isn’t solved, then it looks for edges in the wrong places.

Now the second layer has also been solved, so it is just yellow left.

Text

Description automatically generated

The first step is to create a cross on the yellow face. After the second layer has been solved one of four things will happen: either there will be a dot on the yellow face, an L shape, a straight line through the center, or the cross will already exist.

Text

Description automatically generated

If the cross is already there, then nothing needs to be done at this stage.

If there is a dot then the orientation of the cube is irrelevant so a fixed series of moves can be executed.

Otherwise it is necessary to determine the orientation of the line/L shape before moves can be executed on it.

Text

Description automatically generated

At this stage, the yellow cross exists, but the edge pieces aren’t in the right order, so to fix this first the state must be analysed to see if it can be solved, or already is. Ideally there would be two edges in an L shape with the r/b/o/g stickers correct, although sometimes they are directly opposite each other. The yellow face is turned until one of these happens. If the edges are all correct anyway then the code moves to the next step.

Text

Description automatically generated

When 2 of the edges are correct, this section identifies which are correct and executes the correct moves to position the other 2 edges correctly.

Now only the yellow corners are left to solve.

Text

Description automatically generated

The first stage of solving the corners is to get them in the correct places, without caring about the orientation of each cubelet. The algorithm identified in move\_corners() rotates 3 of the corners clockwise, so the first step is to find a corner already in the correct place which is done by creating a set of the stickers on the cubelet and comparing it to the solved cube. By using a set rather than a touple or list it prevents the order of the stickers from being considered. Once a correct cubelet is found, another is checked to see if the stage is already solved. If it isn’t then the corners are moved until it is done.

If no corners are correct initially then a move is performed which should fix it.

Text

Description automatically generated

The final stage is to rotate the yellow corners and solve the cube. A number of patterns can occur at this stage, but all can be solved using the algorithm in move(), which rotates 2 adjacent corners. Headlights() and perry() are solutions to the most common patterns, and the two other possibilities require a slightly more complicated solution:

This code finds how many of the corners are correct. If 2 are correct it must be a bowtie because the other 2 possibilities would have been solved by this point, so the correct moves are executed to solve that, if there is only 1 solved then a side is turned to move the edges and hopefully make some new options

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

At the end of the code is a section that tests the functionality and finds the average solve time

### Tests

Text

Description automatically generated with medium confidence

After writing each step, I created a unit test to ensure it works correctly. The above screenshot shows the tests for creating the RubiksCube object, ensuring all the attributes are correct. Below that is the tests for turning the white face and ensuring it updates the other sides correctly. I wrote a test for each side in both directions.

Text

Description automatically generated

I also wrote tests for the other methods, to ensure everything is working correctly before testing the solver section.

Text, letter

Description automatically generated

Next, I wrote tests for each stage of the solver, starting with the daisy. This test checks that the yellow face edges contain white stickers.

Text, letter

Description automatically generatedTesting the white cross checks that all the white edge stickers are on the white face, and then checking that they are all in the correct position relative to the sides.

Text, letter

Description automatically generated

Testing the white corners involves checking that the white face is fully solved, as well as the top layer of each side face.

Text, letter

Description automatically generated

Checking the second layer has completed successfully requires confirming that the middle layer of each side face is correct.

Text, letter

Description automatically generated

By checking the yellow edges are all correct, it verifies the yellow cross stage.

Text

Description automatically generated

Testing the edges are in the correct order is a similar process to the white cross.

Text

Description automatically generated

Because the yellow corners can be in any orientation, sets are used to compare the contents of each cubelet to the solved state as the order isn’t preserved by them.

Text

Description automatically generated

To test the final stage, all the steps are performed and then the entire cube is checked to make sure it is solved.

Text, letter

Description automatically generated

The solve() function is also tested by essentially the same process, however the additional step of checking it returns the move list is also performed.

Graphical user interface, text, email

Description automatically generated

The advantage of using the unittest library, rather than just assert statements, is that it offers a much more verbose output if there are any errors, which helps to diagnose where the errors are. It also provides a nice, simple CLI to view the result of the tests.

In a table form, the unit tests are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TEST** | **EXPECTED OUTPUT** | **JUSTIFICATION** | **ACTUAL OUTCOME** | **CHANGES** |
| RubiksCube object initialises correctly | 6 2D arrays, one for each face, containing 9 chars relating to the colour of the face. Empty list called move\_list | This ensures that the object has all the necessary attributes for the program to function | As expected | N/A |
| Turning the white face clockwise works correctly | Cube.white rotates 90 degrees clockwise, .red, .blue, .orange, .green all update correctly to reflect the change | If the faces turn incorrectly it will introduce hard to diagnose logic errors later in the program. | As expected | N/A |
| Turning the white face anticlockwise works correctly | Cube.white rotates 90 degrees anticlockwise, .red, .blue, .orange, .green all update correctly to reflect the change | “ | As expected | N/A |
| Turning the yellow face clockwise works correctly | Cube.yellow rotates 90 degrees clockwise, .red, .blue, .orange, .green all update correctly to reflect the change | “ | As expected | N/A |
| Turning the yellow face anticlockwise works correctly | Cube.yellow rotates 90 degrees anticlockwise, .red, .blue, .orange, .green all update correctly to reflect the change | “ | As expected | N/A |
| Turning the red face clockwise works correctly | Cube.red rotates 90 degrees clockwise, .yellow, .blue, .white, .green all update correctly to reflect the change | “ | As expected | N/A |
| Turning the red face anticlockwise works correctly | Cube.red rotates 90 degrees clockwise, .yellow, .blue, .white, .green all update correctly to reflect the change | “ | As expected | N/A |
| Turning the orange face clockwise works correctly | Cube.orange rotates 90 degrees clockwise, .yellow, .blue, .white, .green all update correctly to reflect the change | “ | As expected | N/A |
| Turning the orange face anticlockwise works correctly | Cube.orange rotates 90 degrees anticlockwise, .yellow, .blue, .white, .green all update correctly to reflect the change | “ | As expected | N/A |
| Turning the blue face clockwise works correctly | Cube.blue rotates 90 degrees clockwise, .yellow, .red, .white, .orange all update correctly to reflect the change | “ | As expected | N/A |
| Turning the blue face anticlockwise works correctly | Cube.blue rotates 90 degrees anticlockwise, .yellow, .red, .white, .orange all update correctly to reflect the change | “ | As expected | N/A |
| Turning the green face clockwise works correctly | Cube.green rotates 90 degrees clockwise, .yellow, .red, .white, .orange all update correctly to reflect the change | “ | As expected | N/A |
| Turning the green face anticlockwise works correctly | Cube.green rotates 90 degrees anticlockwise, .yellow, .red, .white, .orange all update correctly to reflect the change | “ | As expected | N/A |
| Parse method functions correctly | When given a series of moves to execute, they are all executed in the correct order | The algorithms being executed on the cube must be performed in the correct order, else they will leave the cube in an unpredictable state, requiring the solver to go back to step 1 | The moves are executed in the correct order, and move\_list is update d correctly to reflect them | N/A |
| RubiksCube.is\_solved correctly identifies when the cube is solved | Returns True if the cube is in the solved state, else returns False | This method is used in the final stage of the solver to break from a loop, so if it does not function then the program will get stuck in an infinite loop or break too early and fail to solve the cube | Correct value is returned for both solved and unsolved states | N/A |
| Scramble method successfully scrambles the cube | After the method has been called, a long series of random moves are performed to move the cube away from the current state | Whilst not essential to the program, being able to scramble the cube allows for effective testing of solver methods | After the method is called, the cube is no longer in a solved state. | Reduce the number of moves performed during the process as 80 to 120 is unnecessarily high |
| Solver daisy step functions correctly | .yellow[0][1], .yellow[1][0], .yellow[1][2], .yellow[2][1] all contain ‘w’ | Each step in the solver requires the previous step to execute correctly, as there is no checking present between stages so if a stage fails the next one almost certainly will too | The .yellow array contains the correct values | Consider combining this step with the next by constructing the white cross immediately, which will reduce overall move count |
| Solver white cross step functions correctly | There is a white cross on the white face, and the other sticker on those sides lines up with the colour on those faces | “ | The cross is present and the edges are correct | See above |
| Solver white corners step functions correctly | The white face, as well as the top row of each adjacent face is the correct colours | “ | The required values are correct | N/A |
| Solver second layer step functions correctly | The middle layer of each face (except yellow) is correct | “ | As expected | N/A |
| Solver yellow cross step functions correctly | There is a yellow cross present on the yellow face | “ | As expected | N/A |
| Solver solve cross step functions correctly | The edges of the yellow cross are in the correct places | “ | As expected | N/A |
| Solver yellow corners step functions correctly | The yellow corner cubies are in the correct places but not necessarily the correct orientations | “ | As expected | N/A |
| Solver yellow rotations step functions correctly | The cube should be fully solved after this stage | If the final stage does not result in the cube being fully solved then the entire program is useless | As expected | N/A |
| Solver function operates correctly | The scrambled cube should become fully solved and the function should return the list of moves used to solve it | It is important that the list of moves used is returned so that they can be displayed to the user | As expected | Implement a check at the start to prevent trying to solve already solved cubes. |

Acting on feedback from the testing results in the following changes:

Text, letter

Description automatically generated

The number of moves being performed in the scramble method is halved.

Text

Description automatically generated

The solve method now checks if the cube is solved before it starts.

Unfortunately, it was not possible to combine the daisy and white cross steps due to time constraints.

## GUI prototype 1

My first GUI prototype was constructed using the vpython library. This library is quick to write but has numerous limitations that will be addressed as they appear in this section.

Text

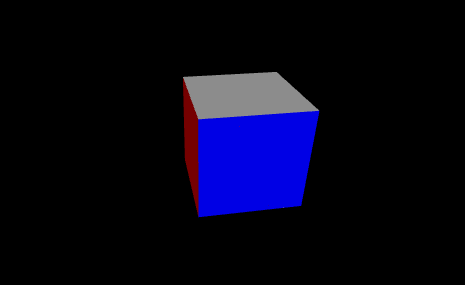
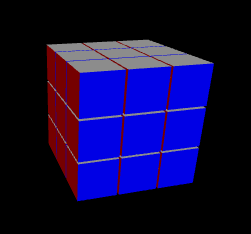
Description automatically generated

At the top of the file is the imports, followed by a simple generator used to create a 2D range, rather than using nested for loops which would prevent break statements working properly.

Text

Description automatically generated

Next is the definition of the model itself. The model is made of 27 cubelets, each of which has 6 faces of different colours so when arranged in a grid they form the faces of the Rubiks Cube. To make a multicoloured cube in this library, it must be constructed of 6 pyramids facing inwards, so only the base of each pyramid can be seen. Each cubelet measures 1x1x1 units, with a 0.05 gap between each cubelet.



Model with gaps between cubelets

Model without gap between cubelets

Graphical user interface, text

Description automatically generated

Next, I defined a subclass used to store which cubelets are on which side. This is achieved by making 6 objects based on this class, one for each face of the cube. When a face of the cube is rotated the object representing that side make a list of the cubelets currently on that side, rotates them 90 degrees and then puts them back into the array of cubelets.

I found that using sub-objects was the best way to achieve this due to the difficulty in rotating the side of a 3D array.

A picture containing text

Description automatically generated

The rotate\_side method is what the side object uses to manipulate the cubelet array.

Text

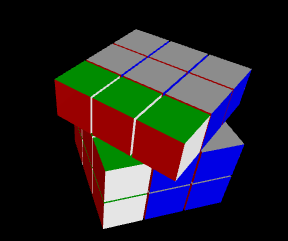
Description automatically generated with medium confidence

Creating the 3D array of the cubelets automatically causes them to appear in the GUI, which is opened via the system default browser (in my case Firefox). LOOP\_FRAMES denotes how many parts the 90 degree turn should be split into. I found that 5 was the ideal number as anything much higher causes the animation to be too slow, and anything lower is visibly choppy. Below that is the initialisation of the side subobjects.

Text

Description automatically generated with low confidence

Each face has a method associated with it to turn it 90 degrees. The turn is split into 5 equal turns, which are performed by moving each individual cubelet via a for loop. Vpy methods are used to rotate the cubelets about an axis which runs normal to the central piece. Vpy.sleep is used rather than time.sleep because it allows for interaction with the rest of the gui such as the buttons and also rotating the cube whilst the move is being executed. The final step of turning the side is to update the cubelet array so that the correct cubelets are rotated on the next move.

A picture containing text

Description automatically generated

Rotating the cube after correctly updating cubelet array

Rotating the cube after incorrectly updating the cubelet array

Text, letter

Description automatically generated

The methods to turn the other sides are largely similar, however they rotate around different axes, and the red, yellow, and blue sides rotate -90 degrees on a clockwise turn rather than 90 due to having to go the opposite direction.

Text

Description automatically generated

Again, I have abstracted the movement via a parser, which take a string representing a series of moves and executes them in sequence.

Text, letter

Description automatically generated

Also implemented is a quick version, which temporarily sets the delay between frames to 0, allowing the moves to execute much quicker. This is intended for when the cube is being shuffled or when a solver is being executed to reduce the amount of time the user spends waiting.

Graphical user interface, text, application, table

Description automatically generated

I created a separate class to implement the buttons and menus. Upon initialisation, it creates the model, which loads the GUI. By putting this before the button creation it causes the model to be fully generated before the buttons appear thereby preventing the user turning a side before all the cubelets are present. Also in this section is the initialisation of data objects for the human and PDB solver. Because the objects have a small memory footprint there is no disadvantage compared to converting to them during the solver call.

When a button is pressed, it passes a pointer to the button into the redir function. The function then passes the button.st attribute to the parser.

Text

Description automatically generated

I implemented the scramble method using a similar approach to the cube class used in the human solver, and with the reduced move count identified in the testing phase of that implementation.

Graphical user interface, text, application

Description automatically generated

Also present is the solvers button, implemented via a vpy.menu object. When an option is selected from the dropdown it calls the respective algorithm which returns a string of moves. The string is then passed to the execute method.

Text

Description automatically generated

Initially I had trouble implementing a reset, as vpython does not allow for the deletion of objects. Instead the function turns each cubelet invisible and creates a new cube on top of the old one. The solver structures are simply reset to their initial states. This would cause memory issues if

Text

Description automatically generated

To ensure all the data structures are representing the same cube, a method is implemented to combine the parse functions of each of them, thereby preventing a move occurring on only one structure.

Text, letter

Description automatically generated

Also, there is a similar method which implements the model.quickparse() method to perform the moves much quicker. This is intended for use when scrambling, and is also helpful for debugging.

### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TEST** | **EXPECTED OUTPUT** | **JUSTIFICATION** | **ACTUAL OUTCOME** | **CHANGES** |
| 2D range generator functions correctly | (0, 0), (0, 1), (0, 2) etc | Range\_2d is used throughout the code. If it is incorrect then the cube movement will not work correctly. | As expected | N/A |
| Cubelets have the correct colours and shape | Cube shape, size 1x1x1, each side has the correct colour. | Cubelets make up the cube. If they are incorrect then the cube will look wrong. | As expected | N/A |
| Cubelet array is correct shape and contains the cubelet objects | 3D array storing 27 cubelet objects | The Rubik’s cube is hollow, and made of 26 cubelets, however it is simpler to model it as solid by using 27 cubelets | As expected | N/A |
| Side objects refer to the correct cubelets, and update them correctly | The white side contains the cubelets on the white side, etc. | If the sides update incorrectly, the wrong cubelets will be moved. | White, green, and red sides update correctly, yellow, orange and blue turn the wrong way | Turn the affected sides the opposite direction |
| Turn\_side methods operate correctly | When turn\_white() is called, the white face turns clockwise. | Being able to turn the sides is an integral part of the GUI. | Sometimes the wrong cubelets are moved due to the error documented above. Additionally the movement is very slow. | Fix the sides error. Reduce the number of frames, which will allow the program to execute faster. |
| Parse and quickparse methods work correctly | When given a string, the methods will execute the moves described in them. Quickparse will set the delay to 0 before executing them and put it back to the correct value afterwards. | The methods are used for the UI buttons so if they fail the UI will be unresponsive. If quickparse does not reset the values then the parse function will not work correctly | As expected | N/A |
| The UI movement buttons work correctly. | Each button will turn the correct side in the correct direction. | These buttons are the main way for the user to interact with the cube. | As expected, other than the errors with the movement functions themselves | Fix the movement functions |
| The cube models initiate correctly | The models are initiated and stored as attributes of the UI object | The models are required for the solvers to function | As expected | N/A |
| The Solvers button functions correctly | When pressed, it should open a dropdown to allow the user to pick the solving method they would like to use | The button is required to give the users access to the solvers. | As expected | N/A |
| The scramble function works correctly | When pressed, a 40-60 length series of moves should be executed on the model, and the cube models | Scrambling the cube using the movement buttons would take a while, so this hastens the process | As expected | N/A |
| Execute and quickexecute methods work correctly | All 3 cube models should be updated simultaneously, and the quickexecute method should correctly implement quickparse() | Keeping the 3 objects consistent is essential to the solvers updating the gui correctly | As expected | N/A |

Text

Description automatically generatedText

Description automatically generatedA picture containing text

Description automatically generated

To fix the error of certain sides turning the wrong way, I added a flag to the side class, which inverts the direction of rotation for the affected sides. This in turn fixed the problems being encountered by the gui buttons

Cubelet array incorrectly updated

Graphical user interface, application

Description automatically generated

As can be seen in this screenshot, originally turning a single side would take almost 16 seconds. The ideal time per turn is about 0.1 seconds.

A picture containing text

Description automatically generated

The long move time was cause by me being a little overambitious with the number of frames being used. To reduce the time to a reasonable amount I greatly reduced the amount of sections the move was broken into. Because reducing the number of frames increases the speed, it reduces the effect of having the moves being more granular. Eventually I settled on having 5 frames rather than the initial 1000.

### Result



When the code is run the above page is opened in the system default browser. The cube can be rotated by right clicking and dragging the view window and zoomed using the middle mouse button. The buttons are not in the exact layout as in the design. The Options button was also omitted as all the selections I intended to implement there weren’t feasible in this prototype.

## PDB Solver

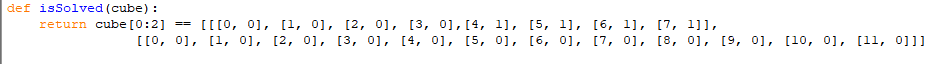
### Data Structures and hashing

#### Data structures and miscellaneous functions

Graphical user interface, text, application

Description automatically generated

First, I define the data structure used in this solver. The array is composed of 2 subarrays, the first represents the corners of the cube and the second represents the edge pieces. The cube is split into the two arrays to allow for quicker hashing of the cube state. The subarrays are 2D, with each element being a list of 2 Integers representing the cubelet in that spot and the orientation of it. The final part of the array is an empty string which is used to contain the move history.



A simple method to check if the cube array is in a sorted state by comparing the corners and edges to the solved state.

A picture containing text

Description automatically generated

I abstracted the process of rotating a side by implementing a parse procedure. Unlike in the Human Solver, here the procedures for each side are nested within parse() rather than being separate. This is because in the early versions the human solver would use the turn\_side methods directly however this caused a lot of repeated code and made it very hard to read so the parse function was added to wrap them, whereas here I used parse from the start.

The res subprocedure is used to rotate the edge pieces. It takes the 4 edge pieces in the order they appear on the face, and the direction to turn them, then moves them to reflect the move that is being performed.

Graphical user interface, text, application, email

Description automatically generated

The procedure to turn each side looks like the one above. First the edges on the face (read clockwise) and the direction are passed to the res procedure. Turning the corners is a similar method, however the corner’s direction must be changed if it isn’t facing the side being rotated. This is dealt with in the if statement. Whilst it would be possible to place the if-else statement into another procedure and pass it the value maps as parameters this would significantly reduce the readability of the code.

The procedure is essentially the same for the white, yellow, blue, and green sides. Only the list being passed to res and the numbers in the if-else statement change

A picture containing text

Description automatically generated

It is only necessary to change the direction of the edge pieces in the red and orange procedures which is done using the above code. Again, the values in the list are different for the orange side to reflect the different edges being moved, however the logic is identical.

Text, letter

Description automatically generated

After the procedures for each side is a for loop which iterates over each character in the given string and executes the respective move.

#### Hashing the cube state

To create a perfect hash of the cube, my initial plan was to calculate Lehmer codes, however I had difficulty understanding the process and as such couldn’t create any meaningful program using that methodology.

Instead, I split the database into 8 sections as there are 8 possible corners in the first section, then split each of those into 7 sections and so on. To decide which of the 8 sections a state belongs in, the first corner is inspected and compared to how many should have been before it if the cube was solved.

When implemented as code, it looks like this: Text

Description automatically generated

The first step is to consider the corner orientations. A given cubelet will only be able to face 3 sides, (i.e. if it is in a location that allows it to face red, then it wouldn’t be possible to face orange from that location as it is on the opposite side of the cube) so the value of the direction it faces can be interpreted as a base 3 number.

By constructing a string from the direction of the cubelets and using python’s int function to read it as base 3 this creates a value between 0 and 218610 which is set as the initial value of index.   
Then the permutation is considered using the method described above.

Hashing the edges uses a similar process: Text

Description automatically generated

The key difference being that hashing the edges only considers 7 of the 12 edge cubelets, so the end parameter tells the function which to inspect.

Text

Description automatically generated

The 3 hashes are called and returned as a tuple.

#### Unit Tests

Text, letter

Description automatically generated

First, I import the relevant libraries and define a function to return a copy of the cube array. In the regular program it isn’t necessary to prevent programs editing CUBE\_INIT, as they only ever have one copy of it, however here each test needs a fresh copy.

The first test checks that common,isSolved() correctly identifies a solved cube.

Text

Description automatically generated with medium confidence

Next, I check the parse function, with tests defined for each face and each direction. Above are the tests for white clockwise and white counterclockwise, and the tests for the others are functionally identical.

Scatter chart

Description automatically generated with low confidence

To test the hashing algorithms, I construct a generator which yields every possible combination of edges/corners and feeds it to the hash function, which checks if the hash has occurred before, fails the test if it has or records it in the set if it hasn’t. This test is a bit slow but is effective at checking the hash has no collisions.

A picture containing text

Description automatically generated

The test for the edge hashing operates very similarly, constructing all possible combinations of 7 sides and checking for collisions. I only check the hash function for end=False because the logic is the same either way.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TEST** | **EXPECTED OUTPUT** | **JUSTIFICATION** | **ACTUAL OUTCOME** | **CHANGES** |
| common.isSolved | True if the cube is solved, else False | This function is used as the end goal in the IDA\* so needs to return the correct value | Correct | N/A |
| Common.parse White cw | Cube array modified to reflect move | If the cube turns incorrectly, the generated solution will be inaccurate | Correct | N/A |
| Common.parse White ccw | As above | As above | Correct | N/A |
| Common.parse Yellow cw | As above | As above | Correct | N/A |
| Common.parse Yellow ccw | As above | As above | Correct | N/A |
| Common.parse Red cw | As above | As above | Correct | N/A |
| Common.parse Red ccw | As above | As above | Correct | N/A |
| Common.parse Orange cw | As above | As above | Correct | N/A |
| Common.parse Orange ccw | As above | As above | Correct | N/A |
| Common.parse Blue cw | As above | As above | Correct | N/A |
| Common.parse Blue ccw | As above | As above | Correct | N/A |
| Common.parse Green cw | As above | As above | Correct | N/A |
| Common.parse Green ccw | As above | As above | Correct | N/A |
| Hashing corners | Each combination of corners results in a unique hash value | If the hash algorithm has collisions, it reduces the access speed which would have a significant impact on the speed of the solver | No collisions, also all hash values are consecutive so there are no gaps in the table | N/A |
| Hashing Edges | As above | As above | As above | N/A |

#### Multiprocessing

Text

Description automatically generatedIn an attempt to increase the speed of the hash I used python’s multiprocessing library to implement parallel processing, however this had the opposite effect due to overhead and increased hash time by at least 100x so I reverted to running them sequentially.

### Pattern Database Generation

#### Early Version

My first attempt at the PDB generation created each database one at a time

##### Corners

Text

Description automatically generated

Imports, creating some globals and a method to turn the cube array into a more useful format

Text

Description automatically generated

A function which given a cube state (which contains the move history), determines if the state has been reached previously in less moves. If this is the shortest path, it updates the array to reflect that and tells the DLS to continue exploring the branch, but if it isn’t the best path it tells the DLS to stop. To allow the subsequent iterations of the DLS to work, the function must return true when the path is equal to the best.

There is also a print statement called every 10000 values to allow monitoring of the program.

Text

Description automatically generated

An Iterative Deepening Depth First Traversal implementation, which repeatedly calls a Depth Limited Traversal with an increasing limit. To prevent evaluating unnecessary states, the previous move is recorded to prevent turning the same side more than once, as this would be guaranteed to not be the shortest path.



The IDDFS is called with a slightly modified version of the data structure that only contains the corners, depth and previous move.

##### Edges

Generating the corner PDB with the previous program took a considerable amount of time, so with the edges I tried to implement multiprocessing to allow multiple branches to be explored simultaneously. Unfortunately, I wasn’t able to get the code to function so I went back to using a single process.

Text

Description automatically generated

Like the corners, I determine if the branch should continue being explored or not. To prevent multiple processes trying to access the array at the same time it was necessary to lock the array whilst it’s being accessed and modified.

Text

Description automatically generated

The tree traversal is implemented in the exact same way as the corners PDB.

Graphical user interface, text, application, email

Description automatically generated

Rather than call the IDDFS on the solved cube, the program executes a move on it and then spawns a process to explore that branch. This should result in 18 sub processes exploring the graph simultaneously, however despite lots of work I couldn’t get it to function correctly.

Chart, scatter chart

Description automatically generated

Instead, I reverted to not using multiprocessing.

#### Final Version

Text, letter

Description automatically generated

For the final version, I combined the 3 traversals into 1. It starts with the relevant imports (common and hasher are the modules described in [Data Structures and Hashing](#_Data_Structures_and)). Numpy arrays are created to serve as the pattern databases and prepopulated with a value higher than will be encountered by the traversal to ensure the first visit to a state updates the array.

Text

Description automatically generated

Again, a function is used to determine whether a branch should be explored. The cube state is hashed and compared to the values in the pattern database. If it is shorter than the previous best route then the database is updated, else if it is equal to the best route then True is returned, else if it is worse than ideal False is returned. checked\_states tracks the number of times the arrays have been updated, which provides a way to observe the execution of the code.

Text

Description automatically generated

The traversal implementation remains largely unchanged, with the main change being the removal of previous move checking as it had minimal impact on the execution speed.

### Search Function

To find the ideal solution to the cube, and Iterative Deepening A\* search is applied using the PDBs as the heuristic.

Text, letter

Description automatically generated

First the module attempts to load the Pattern Databases, but if they aren’t present then it prompts them to be generated.

Text

Description automatically generated with low confidence

The heuristic used by the search is the maximum of the PDBs, which is found and returned in this function

Text

Description automatically generated

A generator is defined which takes a cube state as input and iterates over each possible state that can be reached by executing a single move on it.

Text

Description automatically generated

Implementation of the IDA\* search.

Text

Description automatically generated with medium confidence

Performing the search on a known, simple scramble offers an easy way to perform iterative testing during development.

#### Testing

Text

Description automatically generated

The search is tested by executing some moves on the cube and confirming the solution is the same length or shorter than the scramble.

# Evaluation

For the final evaluation I will be collating the tests performed on each section and using them as well as black box/user testing to determine how successful the solution is.

## White Box Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | Expected  outcome | Justification | Actual Outcome | Changes |
| Unsolvable cube state to the solvers | Catches the error and displays a message to the user | Passing an invalid state to the solvers would break them | There is no mechanism to produce an unsolvable state, so this test is no longer applicable |  |
| Solvable cube state to the solvers | Solves the cube and displays the solution to the user | If the solution isn’t correctly solved and displayed for the user there would be no reason to use the software | The Human Solver can find a solution. The PDB can find one if it is less than 10 moves from being solved | Fully generate the pattern databases |
| Cube state into hashing algorithm  (PDB Solver) | Unique hash value given to each corners state, in range 0 to 88,179,840 | To ensure maximum memory efficiency there needs to be a perfect hashing algorithm, meaning no collisions and no unused slots in the hash table | Each possible corners state generates a unique value. The hash is quite slow. | Rewrite the hash function to be quicker |
| Cube state into hashing algorithm  (PDB Solver) | Unique hash value given to each first 7 edges permutation in range 0 to 510,935,040 | To ensure maximum memory efficiency there needs to be a perfect hashing algorithm, meaning no collisions and no unused slots in the hash table | Each possible edges state generates a unique value. The hash is quite slow. | Rewrite the hash function to be quicker |
| Cube state into hashing algorithm  (PDB Solver) | Unique hash value given to each last 7 edges permutation in range 0 to 510,935,040 | To ensure maximum memory efficiency there needs to be a perfect hashing algorithm, meaning no collisions and no unused slots in the hash table | Each possible edges state generates a unique value. The hash is quite slow. | Rewrite the hash function to be quicker |
| Use hashed state as index for PDB | Return the correct distance from solved for that permutation | If the wrong value is returned it will result in an invalid heuristic | The PDBs are an admissible heuristic unless the state is further than 10 moves from solved, in which case they aren’t generated. | Fully generate the PDBs |
| Solvable cube state passed from main program to PDB solver | Cube state transformed into correct structure | The state passed from the main process will not be formatted correctly for the solver so it needs to be turned into the correct structure before the rest of the function can run | I chose to have all the cube representations present at once, to avoid having to convert between them. |  |
| Solvable cube state passed from main program to PDB solver | Ideal solution is found and returned to the main program | Solution needs to be returned to the main program so that it can be displayed in the GUI to the user.  PDB solver should find the ideal solution to a given state. | The shortest path is found if the state is within 10 moves of being solved | Fully generate the PDBs |
| Solvable cube state passed to Human solver | Solution found and returned to main program | Same as above, however human algorithm is not expected to find ideal solution. | A non-optimal solution is found and returned |  |
| GUI displays the cube correctly | 3x3x3 Rubik’s Cube with correct colours and size | The model of the Cube should be reasonably similar to the real thing. | The cube displays correctly. When the program loads it generates 1 cubelet at a time |  |
| Turning a side of the cube displays on the GUI correctly | The correct 9 cubelets rotate 90 degrees along a circular path about the center of the middle cubelet. | If the wrong cubelets rotate, it will disassociate with the underlying data structure, causing the wrong cubelets to move on every subsequent rotation. | The cube rotates correctly. All 3 parallel data structures update accordingly |  |
| Calling the solvers from the GUI works correctly | When the button is pressed, the current cube state is passed to the solver and evaluated, then the solution is executed on the cube | The GUI is how the user interacts with the program, so if the solvers can’t be accessed from it, the user won’t be able to run them. | The solvers are called when the user presses them. If the PDB solver cannot work, it does not inform the user, instead nothing happens. | Allow the PDB solver to feedback to the user if no solution can be generated. |

## Comparison to Success Criteria

|  |  |  |
| --- | --- | --- |
| Criteria | Has it been met | Why/Why not |
| 1 | Yes | The GUI works |
| 2 | Yes | The cube generates correctly, although it is of a lower quality than I would’ve liked. |
| 3 | Partially | The only method possible to implement was buttons, although the downsides of them was mitigated by having them below the viewing window. To implement the others would require using a library other than vpython |
| 4 | No | Everything that I intended to place in the options menu was cut either due to time or library constraints, so the menu proved unnecessary |
| 5 | Yes | The solvers work. Due to the hashing algorithm being slow it takes a long time to generate the PDBs, so I have only tested that to depth 10. |

## Usability Testing

To test the usability, I gave the following questionnaire to some of my stakeholders to gain feedback on the usability of my program.

1. How did you find loading the program?
2. What do you think of the GUI? Any features you particularly like or dislike?
3. How did you find moving the cube?
4. How easy was it to utilise the solvers?
5. What do you think of the miscellaneous features (reset/scramble)

|  |  |  |  |
| --- | --- | --- | --- |
| **Question Number** | **Barney** | **Murray** | **Will** |
| 1 | I found the program easy to boot up, having been given some instruction | The code was simple to run and loaded quickly | If I wasn’t familiar with python this wouldn’t be as intuitive. |
| Discussion | All three users found the program easy to load, however it was highlighted that for users unfamiliar with python and .py files, it may not be apparent how to run the software | | |
| 2 | I like the GUI, however the buttons are a bit small. | I like how the cube loads one bit at a time. The GUI is very simplistic but that isn’t necessarily a bad thing | I don’t like how the GUI doesn’t scale with the window, but it can be done manually so it isn’t too big of an issue |
| Discussion | Users noted that the buttons are too small and the disconnection between the browser size and the model size. Unfortunately, both are limits of the vpython library so cannot be fixed. | | |
| 3 | Moving the cube was very easy, although slightly annoying having to press the buttons so many times | I would’ve preferred to be able to move the cube with the mouse, rather than just pressing buttons. | Not being able to directly input cube states makes the program much less useful for me. |
| Discussion | There were mixed feelings about the movement, one user was happy with it however the other 2 would’ve liked to see the other methods implemented as well | | |
| 4 | The solvers were easy to use and worked quickly. Somewhat annoying that [the PDB solver] only works some of the time | I struggled to get the PDB solver to work, but the human one worked fine | I liked how the solvers were implemented, and it’s interesting to watch them work. |
| Discussion | I was expecting some complaints about the inconsistency of the PDB solver, and comments other than those were positive. I would like to generate more of the PDBs, but this would require rewriting the hash to be considerably faster. | | |
| 5 | They were simple and worked well. | I like how the scramble looks, the cube looks like it’s being randomized rather than just turned | I like how the cube loads back after it’s reset, it looks very fancy. |
| Discussion | Positive feedback regarding the reset and scramble functions. The way the cube reloads after a reset is not intentional, simply vpython is slow enough that the time between cubes loading is noticeable. | | |

I also performed the following tests

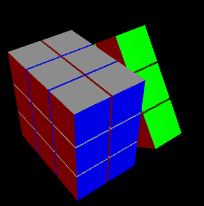
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input | Expected Outcome | Justification | Actual Outcome | Changes | Success Criteria |
| Load the program | Program opens to show main GUI window | Ensures the program loads as intended | Works as Expected |  | 1 |
| Load the program | Main GUI should feature a 3D model of the Rubik’s Cube in its solved state | Ensures the model is working | Works as Expected |  | 2 |
| Use movement buttons | The correct face should turn in the correct direction for every button | Ensures the model updates correctly for every input | Works as Expected |  | 3 |
| Click and drag a face of the cube model | The face should turn in the direction it is being dragged | Ensures that method of affecting the cube is working | Not implemented | For future development | 3 |
| Click a sticker on the cube | The sticker being clicked should cycle colour each time it is clicked | Ensures that method of affecting the cube is working | Not implemented | For future development | 3 |
| Click the autosolve button | Should open a menu allowing the user to select a solver | Ensures users have access to the solvers | Works as Expected. |  | 5 |
| Select the human algorithm from the autosolve menu when the cube is in a solvable state | The human solver should be used to find a solution to the current cube state | Ensures the human solver is being called correctly and functions properly | Works as Expected |  | 5 |
| Select the human solver from the autosolve menu when the cube is in an unsolvable state | The GUI should show a popup message saying that the cube isn’t currently in a solvable state | Ensures that unsolvable states are being caught | Cube cannot be placed into an unsolvable state, so this wasn’t implemented |  | 5 |
| Select the PDB algorithm from the autosolve menu when the cube is in a solvable state | The PDB solver should be used to find a solution to the current cube state | Ensures the PDB solver is being called correctly and functions properly | Works as Expected |  | 5 |
| Select the PDB solver from the autosolve menu when the cube is in an unsolvable state | The GUI should show a popup message saying that the cube isn’t currently in a solvable state | Ensures that unsolvable states are being caught | Cube cannot be placed into an unsolvable state, so this wasn’t implemented |  | 5 |
|  |  |  |  |  |  |

A screenshot of a computer

Description automatically generated with medium confidence

When run, the program opens in the browser as expected.

The Cube is clearly visible at the top of the window, and the buttons are loaded at the bottom.



When the movement buttons are pressed, the relevant move is executed on the cube.

A picture containing company name

Description automatically generated

When clicked, the solvers menu opens with the two solvers listed as options.



When the solvers are selected in the menu the program uses them to solve the cube.

## Limitations/Maintenance and Future Development

Using the Vpython library to construct the GUI is convenient, however it is very limited in functionality, for example the reset button has to hide the previous cube rather than being able to delete it, so after a few resets the program gets slow. I was also not able to implement the alternate ways of setting the cube. In future I would like to replace it with a different method, ideally one which does not require the web browser and exists in its own window. This would hopefully also allow me to closer align the design of the GUI with my original intentions and implement the alternate methods of manipulating the cube.

To implement the other methods of moving the cube may be possible in Vpython, as it is possible to find what the user clicks on in the view window, however it would be very complicated to determine which part of the cube is there, and then update that part of the cube as vpython doesn’t allow easy modification of objects that have already been created. It would be similarly complex to implement click-drag, so I would prefer to move to another graphics library that offers more functionality.

Furthermore, the hashing algorithm used is quite inefficient, which has prevented me generating the entire pattern databases, so currently they are only accurate to depth 8, meaning if more than 9 moves are executed on the cube, it is very unlikely the PDB solver could generate the correct solution. I would like to implement the hash in a faster, compiled language such as C or C++ to allow it to run faster.

The program should be quite maintainable, the only concern is Vpython as it is the only obscure library used, however it is based on WebGL which is well maintained so it should retain its functionality. Other than that, the program only relies on numpy and the python standard library, neither of which will become unsupported in the foreseeable future.

# References

Agostinelli, F. M. S. S. A. e. a., 2019. Solving the Rubik’s cube with deep reinforcement learning and search. *Nat Mach Intell,* Volume 1, pp. 356-363.

Anon., 2018. *SolveTheCube.* [Online]   
Available at: https://solvethecube.com/  
[Accessed 24 April 2021].

Duin, R. P. W. & Pekalska, E., 2007. *The Science of Pattern Recognition. Achievements and Perspectives.* [Online]   
Available at: http://rduin.nl/papers/cschallanges\_07\_prscience.pdf  
[Accessed 4 2 2021].

Korf, R. E., 1997. Finding Optimal Solutions To Rubik's Cube. *aaai ?,* pp. 700-705.

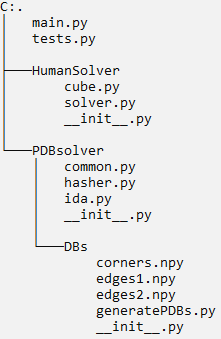
Wikipedia, n.d. *Iterative Deepening A\*.* [Online]   
Available at: https://en.wikipedia.org/wiki/Iterative\_deepening\_A\*  
[Accessed 4 1 2022].

Wikipedia, n.d. *Iterative Deepening Depth-First Search.* [Online]   
Available at: https://en.wikipedia.org/wiki/Iterative\_deepening\_depth-first\_search  
[Accessed 4 1 2022].

# Final Code

This section contains all my final code, using some fancy word formatting. Double click any section of code to expand it.

### Directory Structure



The tree of the code directory, obtained by using tree /F in command prompt

The three files called \_\_init\_\_.py are blank however they are required for relative imports to work.

The three ending in .npy are the PDBs

Main.py



### Tests.py



### Cube.py



### Solver.py



### Common.py



### Hasher.py



### Ida.py



### generatePDBs.py

