Solving Rubik's Cube with Lego Mindstorms

Will Garside

03-05-2018

I Signed Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.
Name: Will Garside
Signature:
Date:

II Abstract

This should be two or three short paragraphs (100-150 words total), summarising the report. A suggested flow is background, project aims, and achievements to date. It should not simply be a restatement of the original project outline

III Acknowledgements

Thanks to my parents, who raised me since I was a boy. And Rik van Grol, who raised me afterwards.

IV Definitions and Notations

Table IV.1: Terminology used in this document

Abbreviation/Word	**Definition**
[Rubik's] Cube	A standard 3x3x3 Rubik's Cube[1].
Face	One of the size faces on a Cube, Denoted by position or colour. e.g. U-face
Slice	A central layer between two faces. Usually referenced by the faces it spans.
Quarter Turn	A clockwise rotation of a face or slice by 90°
Half Turn	A clockwise rotation of a face or slice by 180°
Quarter Turn Metric	When counting moves, a Quarter Turn is counted as a single move and a half is two moves.
Half Turn Metric[2]	Quarter turns and half turns are both counted as single moves.
Cubie	One of the twenty-six smaller cubes that make up a Cube.
Facelet	One of the fifty-four coloured squares on the faces of the Cube.
Goal State	All the cubies on a given face match the colour of the centre cubie. i.e. a solved Cube.
Position	A Cube's state (mixed or solved).
Moveset	A set of moves, denoted with curly braces. e.g. $\{\mathtt{U\ D'\ L^2}\}$
Move Sequence	A series of moves **performed consecutively**, enclosed by colons . e.g. $\mathtt{:F\ D\ F'\ D^2\ L'\ B'\ U\ L\ D\ R\ U\ L'\ F'\ U\ L\ U^2\ :}$
Solve Sequence	A move sequence which leads to the goal state.
Depth n	A Cube which has been moved n times away from the goal state.

[1] This Dissertation is only dealing with 3x3x3 Rubik's Cubes, and any discussion of alternative dimensions will be explicitly stated.

[2] For this Dissertation the Half Turn Metric is used unless explicitly stated.

Contents

List of Figures

List of Tables

1 Introduction

> Like after a nice walk when you have seen many lovely sights you decide to go home, after a while I decided it was time to go home, let us put the cubes back in order. And it was at that moment that I came face to face with the Big Challenge: What is the way home?
>
> Ernö Rubik [4]

1.1 Background

In 1974, Ernö Rubik was struggling to create a cube with independently moving parts which remain together, regardless of how much they moved. His first attempts made use of elastic, which broke and rendered the cube unusable. Rubik persevered in his attempts to hold the blocks (now called 'cubies') together - eventually concluding that the best way was to have the cubes hold themselves together. He called this design '*The Magic Cube*', and it would go on to be one of the world's best-selling puzzles [5]. It was later re-branded to '*Rubik's Cube*' to overcome an oversight involving patenting and copyrighting the design.

In an unpublished manuscript [4], Rubik described first randomising his new cube,

> *It was wonderful to see how, after only a few turns, the colors became mixed, apparently in random fashion. Like after a nice walk when you have seen many lovely sights you decide to go home, after a while I decided it was time to go home, let us put the cubes back in order. And it was at that moment that I came face to face with the Big Challenge: What is the way home?*

It took Rubik over a month to solve this first cube - he knew intuitively that there must be a method to solving the cube, but lacked the finer methodology [6]. Since Rubik devised the first method, hobbyists and mathematicians alike have been immersed in solving the Cube as quickly and efficiently as possible. Whilst many solutions are markedly successful when it comes to optimisation, others only better them in quirkiness or internet fame [7].

There is one method which, despite having been proved mathematically, is still little more than speculation: God's Algorithm. The theory behind this algorithm states that an omniscient being would always make the most efficient moves and that they would be able to solve a Cube from any given position in a certain number of moves or less. This is referred to as God's Number, and was finally proved to be twenty in 2010 by a group of four researchers [1].

1.2 General Objectives

The primary objective of this project is to successfully implement an algorithm to solve a Cube with a Mindstorms robot. The efficiency of this algorithm is initially non-imperative, but will ideally be improved over the project time-line. The most recent iteration of the Mindstorms line, the Lego EV3 31313, will be used in the construction of the robot. This provides a wireless connectivity via Bluetooth (or WiFi with a USB dongle), three motors, a colour sensor, and a touch sensor amongst other peripherals. A custom operating system can also be installed on a microSD card to allow for greater expandability.

Secondary objectives include implementing other algorithms from various sources, devising and refining my own algorithms, and comparing the performance, efficiency and solve-length of the algorithm. The robot will have to be of sound construction, with little-to-no room for error when manipulating the Cube.

1.3 Limitations and Constraints

The first problem which will be encountered will undoubtedly be during the design and build of the robot: the number of Lego pieces supplied in the EV3 set is quite limited at only six-hundred-and-one pieces - approximately a quarter of which are only for aesthetics. The design will be supplemented by sets sourced externally, thus allowing a robot of sufficient quality to be built.

Despite being of high quality and uniform across all sets, Lego is still fundamentally a toy - which will lead to errors with the precision of the robot [8]. An example of this is the motors: there is a somewhat significant degree of freedom/play in the motors, which means that they can't be relied on to move to accurate positions. A worm gear is the ideal solution to this problem: despite reducing the speed, it provides a considerable increase in motor accuracy.

One of the larger issues that this project will encounter is the run-time of the program to find a solution. Many programs have been known to take upwards of three days to find a solve sequence - this project currently aims to find a solution in approximately fifteen minutes or less (an arbitrary number determined only by the attention span of the author). This will require extensive optimisation of search spaces and group theory through symmetrical elimination and set covering in order to reduce the necessary coverage of the search algorithm.

Finding the optimal solve sequence for any given position has been deliberately omitted from the objectives of this project: there are over forty-three quintillion valid positions of a Cube, each requiring an extensive amount of time to find the optimal solution for. Based on current hardware capabilities and previous studies and research, an estimated timespan for finding all optimal solutions is in the order of millennia[1].

[1]This is an estimation based on data presented at cube20.org which states that finding optimal solutions takes 11,800 times as long as finding sequences or twenty moves or less [1]

2 Literature Review

> Please forgive me, but to give birth to a machine is wonderful progress. It's more convenient and it's quicker, and everything that's quicker means progress. Nature had no notion of the modern rate of work. From a technical point of view, the whole of childhood is quite pointless.
>
> Mr. Fabry, in Karel Čapek's '*Rossum's Universal Robots [9]*'

2.1 Robotics

The term robot was first brought to the collective consciousness of the general public by Karel Čapek in 1921 in his play '*Rossum's Universal Robots*' [9]. It comes from the Czech for 'forced labour' and was coined by Čapek's brother, Josef, for a short story written some years earlier [10]. One of the main discussions in the field of robotics has been about ethics and morality. This discussion was started in Čapek's play, when a visiting scientist tries to destroy the robot-manufacturing business, in the hope of giving the robots a soul and making them happier. This plan goes awry when, as is the norm in fiction surrounding Artificial Intelligence (AI), the robots become sentient and start to overthrow the humans that created them. Twenty-one years later, the novelette '*Runaround*' by Isaac Asimov was featured in the magazine '*Astounding Science-Fiction*' [11]. It included his now-prominent Three Laws of Robotics, which were a turning point in the field of robotics and meta-ethics.

The twentieth century generated many ideas about the future of robotics, such as 'Kitchen units will be devised that will prepare "automeals", heating water and converting it to coffee' alongside grandiose aspirations such as 'An experimental fusion-power plant or two will already exist in 2014' [12]. Whilst some predictions have been fulfilled by the technology of the past two decades, others are still nowhere near completion and could be considered impossible for the next century. This is a clear demonstration that the field of Robotics has made vast improvements, but is still - in some respects - very much in its youth.

In the same way as the field of Robotics, AI has vast potential and we have only touched the tip of the iceberg. The most advanced commercial AI systems are intended to make people's lives easier through organisation and saving precious seconds in performing simple tasks on their smartphones and more recently in their homes. However, Google's Assistant, Apple's Siri, and Amazon's Alexa still fall short when it comes to the intelligence demonstrated in Asimov's book '*I, Robot*'. We are still far from being unable to 'differentiate between a robot and the very best of humans' [13].

2.2 Robotics in Education

There has always been a clear use of 'edutainment' style teaching in one form or another for hundreds of years. One of the earliest known uses of edutainment is in '*Poor Richard's Almanack*', which was written by Benjamin Franklin and published annually between 1732 and 1758. Alongside all the usual information contained in an Almanac, Franklin (under the pseudonym '*Poor Richard*') included maths exercises, puzzles, and aphorisms [14], [15]. Walt Disney was another pioneer of edutainment in the time immediately before, during, and following the World Wars. His first piece of edutainment was a short film, '*Tommy Tucker's Tooth*', which was commissioned by a dental institute in 1922.

In the past few decades, there has been a marked change in the form of edutainment in line with advances in technology. We have progressed from paper-based, so called 'serious games' in the 1970s, to early-era video games such as the infamous Oregon Trail; and then in the past decade electronic and robotic kits have made their way into classrooms.

Robotics has been described as 'the most effective way of motivating and supporting the study of many areas', and has also been shown to aid the development of social and teamwork skills [16]. In the past decade or so, robots and electronic kits have been a common extra-curricular activity and are slowly being used as teaching aids and for practical sessions in actual lessons.

One of the leaders of modern edutainment is the Lego Mindstorms kit. The Lego Mindstorms kit has undergone two major revisions since the launch of the first generation of the Mindstorms kit in 1998 (the RCX): in July 2006, the second-generation NXT was released, updating the sensors and adding Bluetooth amongst other improvements; then in September 2013, Lego released the EV3 Home and Education sets. This release cemented Lego's position at the forefront of edutainment [17].

2.3 Lego Mindstorms

A study at the University of Calabria in 2009 looked at how Lego Mindstorms worked as a learning tool when used in team-building type tasks [18]. Twenty-eight students were divided into six groups, each given a Lego robot and preliminary training which covered the Lego Mindstorms and programming environment. The groups were all given the same task, and their progress studied throughout the sessions. When studying the building and programming as two separate sub-tasks, three main categories of work subdivision were found: each group member had a set role in building, but all shared the programming; both sub-tasks were equally divided amongst members; each member had a set role in either sub-task, and there was a clear leader. This closely follows real-world teamwork mechanics and scenarios, and the use of robots was found to stimulate the student into exploring and sharing critical knowledge within the group. The study's conclusion was that Lego Mindstorms - and inherently other variable morphology robots - stimulates further process analysis, information selection, and to observe and experiment with the consequences of their actions [18].

The Lego Mindstorms' native programming environment is makes use of drag-and-drop code blocks to form programs which can run natively on the main 'brick'. This provides a simplistic user experience and allows novice programmers to create complex programs with relative ease. Drag-and-drop programming can drastically reduce the amount of syntax errors in a program, allowing programmers to better focus on the functionality of their program [19].

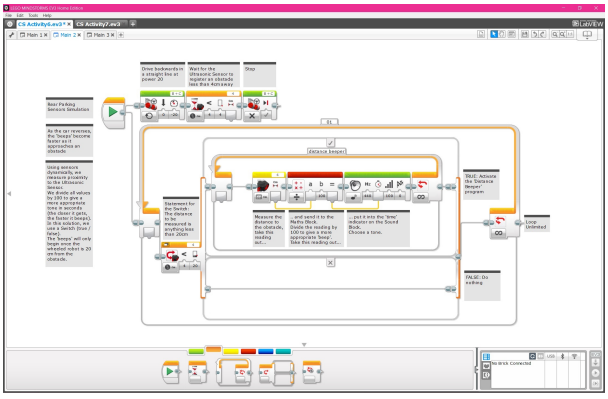


Figure 2.1: A sample program in '*EV3 Programming Software*'

2.4 Solving Puzzles with Algorithms

When it comes to playing games and solving puzzles, Google's '*AlphaGo Zero*' is the clear winner. Whilst Google's first Go-playing program defeated the world champion at Go, it took months of supervised

learning from experts and reinforcement learning from self-play. The new version started *tabula rasa* and was completely self-taught - it had no interaction with any humans, and used no historical data. Through reinforced learning alone by simulating games against itself, it took three days to reach the level of AlphaGo and in forty days had surpassed any Go player's performance - human or artificial [20], [21].

When a computer solves a Cube (or any other similar puzzle), it does so by following set algorithms and rules until the solved state is achieved. Computers cannot use humanlike instincts, so we often try to provide a heuristic to make the task simpler. These heuristics can be pre-computed data tables to reduce the amount of processing at run time, or a reduction in the original data set by eliminating certain elements according to a set of rules.

In his 1997 paper on the use of pattern databases to increase solve efficiency, Richard Korf uses different heuristic functions and characterises how effective they are in reducing the number of moves in a solve sequence. His first method used an Iterative-Deepening A* algorithm, combined with the heuristic of the Manhattan distances from the edge cubies' current position and orientation to that which is desired. This reduced the time required to solve a Cube at depth-14 to three days[1], however when increased to depth-18 the time increased exponentially to two hundred and fifty years. After a re-evaluation, Korf modified the heuristic by pre-computing the Manhattan distance of each cubie from all of its possible positions and orientations and storing the table in memory at run-time. This created a table of nearly ninety-million entries - which would have been bigger but was restricted by the available memory (the table was approximately forty-two megabytes). The newer heuristic reduced the time to search at depth-18 to less than four weeks - a reduction of approximately 99.97%. Korf suggested that the speed of the algorithm used would increase linearly with the amount of memory available, meaning that if it were run today the depth-18 search would take under three hours[2] [22].

Table 2.1: Morwen Thistlethwaite's five groups for his algorithm [3]

Group Denotation	**Mathematical Representation**	**Summary**
G_0	$\{\texttt{L R F B U D}\}$	All valid positions
G_1	$\{\texttt{L R F B U}^2\texttt{D}^2\}$	All positions that can be reached with quarter turns of the `L R F B` faces and half turns of the `U D` faces
G_2	$\{\texttt{L R F}^2\texttt{B}^2\texttt{U}^2\texttt{D}^2\}$	Positions reachable from quarter turns of `L R` faces and half turns of the `F B U D` faces
G_3	$\{\texttt{L}^2\texttt{R}^2\texttt{F}^2\texttt{B}^2\texttt{U}^2\texttt{D}^2\}$	Positions only reachable from half turns of any face
G_4	$\{I\}$	The goal state

There have been many historical landmark algorithms since the inception of Rubik's Cube in 1974. One of the earliest instances was created by Morwen Thistlethwaite circa 1981 and is based on group theory. Thistlethwaite's algorithm splits all possible positions of the cube into five groups of decreasing size as seen in Table 2.1. David Singmaster, Thistlethwaite's colleague at London South Bank University (then called Polytechnic of the South Bank), describes the methodology of the algorithm: 'Once in G_i, one only uses moves in G_i to get into G_{i+1}. The ratio $\frac{|G_i|}{|G_{i+1}|}$ is called the index of G_{i+1} in G_i.' Using this algorithm, Thistlethwaite proved that the maximum number of moves required to solve any valid position is fifty-two [3]. This was the first development of God's Number, so named because it is the number of moves that an omniscient being would use to solve a Cube in the most efficient manner without failure.

[1]The simulation was run on a Sun Ultra-Sparc Model 1 workstation

[2]This is based on the memory capacity of the Model 1 Workstation and a modern computer being 64MB and 16GB respectively.

2.5 God's Number

The lower bound for God's Number was originally recognised to be eighteen moves through analysis of the number of sequences of seventeen moves or fewer, and the discovery that there were more Cube positions than seventeen-moves-or-fewer sequences. In 1995, the lower bound was increased to twenty by Michael Reid upon discovering that the 'Superflip' position requires twenty moves to solve. He included his findings in an email to the Cube-Lovers mailing list, 'superflip is now known to require 20 face turns . . . this is the first improvement to the lower bound... given by a simple counting argument' [23].

The next refinement to the upper bound of God's Number came from Dutch mathematician Hans Kloosterman in December of 1989. Kloosterman's findings were published in a newsletter '*Cubism for Fun*' (CFF) as an article titled '*Rubik's Cube in 44 Moves*'. In this article Kloosterman discusses his existing solution to the Magic Domino, a subset of Rubik's Cube, and how he has adapted it into a more efficient Cube solution with the help of Thistlethwaite's algorithm. Kloosterman reduced the estimation of God's Number by adapting Thistlethwaite's G_3 into a subgroup of G_2 where all of the U-cubies are on the U face and all the D-cubies are similarly on the D face. This reduces the index $\frac{|G_2|}{|G_3|}$ from 15 to 8 (whilst also reducing the $(G_1 : G_2)$ index by 3 and increasing the $(G_3 : G_4)$ index by 1), creating a net decrease of 8 moves when compared to Thistlethwaite's algorithm. '*Rubik's Cube in 44 Moves*' ended with an editor's note stating that not all of the details of the algorithm were certain and more (including a more efficient algorithm) would be revealed in a later issue [24].

Sure enough, a year later Kloosterman wrote an article titled '*Rubik's Cube in 42 Moves*' in the twenty-fifth issue of CFF. This stated that the new algorithm was in fact the final version of Kloosterman's forty-four move algorithm. In the same way as its predecessor, this article is split into four stages to solve a Cube. The first three stages are based on 'solving' a Cube with only certain cubies coloured - the positions of the other remain irrelevant until a later stage. The final stage is solving the Cube in a non-particular manner, and is calculated to take no more than eighteen moves [2].

In the following twenty years, the upper bound was gradually refined to match the lower bound - meaning that God's Number is exactly twenty. The final development came from a group of four Rubik's Cube enthusiasts in July of 2010, and was only achieved through a brute force method: the four-man team first took every possible position of a Cube and reduced it from forty-three quintillion positions down to just over one quintillion - still a vast number, but a reduction by a factor of forty-three - by using rules of symmetry and mirroring. They then wrote a program to find solve sequences of length twenty or less, and ran it on a 'large number of computers at Google' to find all solve sequences in a few weeks. The parallel computation took the equivalent of thirty-five years [1].

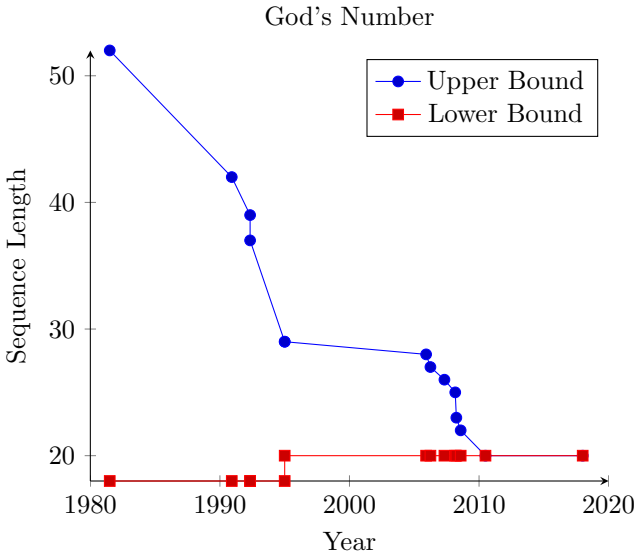


Figure 2.2: The upper bound was decreased to 20 over the course of twenty-nine years and across thirteen separate studies [1]

2.6 Python as a Scientific Scripting Language

Traditionally when performing large calculations and processes, compiled languages (C, C++, Fortran etc.) are the preferred choice over scripting languages such as Ruby, JavaScript, (pure) Python, etcetera. This is because compilation is done before run-time, allowing the program to run unhindered by on-the-fly translation/interpretation [25].

For general purpose applications, scripting languages are preferable because they allow small changes to be made without re-compiling large classes before testing. In the thirteenth issue of '*Scientific Programming*', Cai, Langtangen and Moe [25] explain how MATLAB is often a preferable choice for less intensive scientific computation due to its many features, such as: integrated simulation and visualisation tools; clear syntax; immediate feedback of commands; impressive documentation. They describe the use of MATLAB in computational science as paradoxical, as it is a scripting language which inherently is interpreted at run-time.

One of MATLAB's biggest selling points, the IDE with integrated support for matrices, graphs and other mathematical features, can be replicated in a few Python IDEs or with the addition of Python packages. Python's main core can be expanded massively by using any of the tens of thousands of available packages, which can provide any of MATLAB's advantages alongside extra flexibility and capabilities. One of the most popular packages, '*Numerical Python*' (NumPy), adds support for most scientific calculations. It expands Python's inbuilt data structures to use multi-dimensional homogeneous arrays of any Python data type. NumPy takes Python from a well-structured general-purpose language to a powerful mathematical language which can be applied to many different tasks [25], [26].

As well as the multitude of available packages, Python has another major tool in its arsenal: it can be extended with a compiled language by design. This means that processor-intensive tasks like nested for loops can be migrated to a compiled language such as C++ or Fortran, increasing run speed up to a factor of ten. This extension process is now a trivial matter thanks to automated tools such as '*f2py*', which allows for automated calls to compiled Fortran code [26].

2.7 Conclusion

Lego Mindstorms is a strong candidate for the construction of the robot: its variable morphology allows adaptation to a range of tasks; the uniformity between pieces means that the main EV3 kit can be supplemented with other kits to extend its ability; and previous studies have shown that it is ideal for stimulating analysis and observations. Instead of using a single algorithm to solve the Cube, a range will be implemented in order to find the strongest method for finding a solve sequence. This will allow cross-comparison between, and analysis of, individual algorithms with the intention of finding the algorithm best suited to this project. Tree and group based algorithms alike will be implemented.

MATLAB is incompatible with the Lego EV3, so will not be used for this project. Python's wide compatibility, clear high-level syntax, and extendibility to compiled code for more complex calculations, makes is the ideal language for this project. The main program will be written in Python, with any intensive processing written in C++ to improve the speed of the algorithms.

3 Requirements and Analysis

> I have yet to see any problem, however complicated, which when you looked at in the right way, did not become still more complicated.
>
> Paul Anderson, New Scientist [27]

3.1 Introduction

This chapter covers the specific requirements needed for the project to succeed: the particular tasks to complete the project; the hardware required to build the robot and run the solving program; and the software needed to write and run the program. As well as the requirements, an in-depth analysis of the project and its goals is made, discussing their feasibility, difficulty, and how necessary they are for the success of this project.

3.2 Specific Aims and Objectives

1. Build a robot which can manipulate a Cube accurately

   This is the first of the two primary objectives for this project, and is imperative to the project's success. As such it will be the first objective to be completed. This task, despite being of a medium difficulty, is fairly straightforward by nature: the robot simply needs to manipulate a Cube in any valid move.

2. Implement a system which successfully generates a solve sequence for any given position.

   The second primary objective ensures that any of the forty-three quintillion positions of Rubik's Cube can be solved correctly. The time taken to generate the solve sequence and the actual length of the solve sequence will both be used when measuring an algorithm's success.

3. When the robot is provided with a move sequence...

   (a) Convert any input sequence to be robot-compatible

   All of the pre-existing algorithms return a solution which a human being can follow to solve a real-world Cube. In the context of this project the manipulation is done by the robot, which cannot perform all of the standard moves, so any sequences provided must be translated to one which the robot can follow.

   (b) Correctly follow the sequence to achieve the intended output

   Once a robot-compatible move sequence has been generated, the robot must move the Cube in this exact sequence with no errors or move-failures. If a single move is performed incorrectly then the entire move sequence will produce an incorrect output.

4. Ensure the runtime of the system is an acceptable length

   Previous solve sequence generator algorithms have had a run-time in the order of weeks: the desired run-time of the generation by this project will be in the order of minutes. This will require careful analysis of the trade off between generator efficiency and solve sequence length.

5. Implement a program to use a range of algorithms to generate a solve sequence

   In order to effectively compare the performance of several algorithms, there must be a method of running them under the same conditions and with the same overheads (e.g. transmission time to the robot). For this reason, there will be one main program with an option for the algorithm to use.

6. Compare the performance of different algorithms, especially the difference between human and robot compatible move sequences

   The performance of each algorithm will be tracked and compared fairly to show each one's merits and faults. This will allow the choice of one algorithm to be improved to create the most efficient method - or the potential combination of algorithms to form a stronger one.

3.3 Analysis

3.3.1 Software

PyCharm/CLion

An Integrated Development Environment (IDE) makes development, running programs, and testing and debugging much easier than using a simple text editor and command line. PyCharm and CLion by JetBrains [28] will be used for developing in Python and C++ respectively. PyCharm and CLion provide strong code completion abilities, project-wide refactoring, and software development kit (SDK) modification amongst many other abilities. This will allow more time to be allocated to tasks of higher value rather than struggling with relatively minor issues.

Python 3.6 Runtime Environment

As discussed in the Literature Review, Python is the ideal language for this project. It requires a native runtime environment for programs to be run: this consists of the Python interpreter, libraries, and any packages used in the development process.

MinGW

C++ requires an environment to develop and compile in. MinGW (Minimalist GNU for Windows) is a popular choice when developing C++ applications.

ev3dev

ev3dev [29] is a custom operating system (OS) for the EV3 which contains a low-level framework for the peripherals of an EV3. It allows users to write their own programs in a multitude of languages (including Python) to create complex functions and models. It is installed by flashing the ev3dev OS to a microSD card which is then inserted into the EV3, avoiding affecting the EV3's original firmware.

A Python wrapper is available on GitHub [30] for the ev3dev OS, allowing the creation of Python programs with packaged support for the actuators and sensors of the EV3.

3.3.2 Software Requirements

This table shows the requirements of the software to be used in the creation of this project, and the capabilities of the computer that they will be running on. Any

Table 3.1: Software requirements for the main development computer and its capabilities

Software	**Requirements/Capabilities**			
	Minimum OS	**Processor**	**Memory**	**Connectivity**
PyCharm	Windows 2003 (or newer)		1GB minimum 2GB recommended	
CLion	Windows 7 64-bit (or newer)		2GB minimum	
Lego EV3 Programming Software[31]	Windows 2003 (or newer)	Dual Core 2GHz	2GB	1 USB Port
Python 3.6	Windows Vista (or newer)			
MinGW				
Main Development Computer	*Windows 10 Pro 64-bit*	*Quad Core i5 6500 3.6GHz*	*32GB*	*WiFi, Bluetooth, 12 × USB Port*

3.4 Feasibility Study

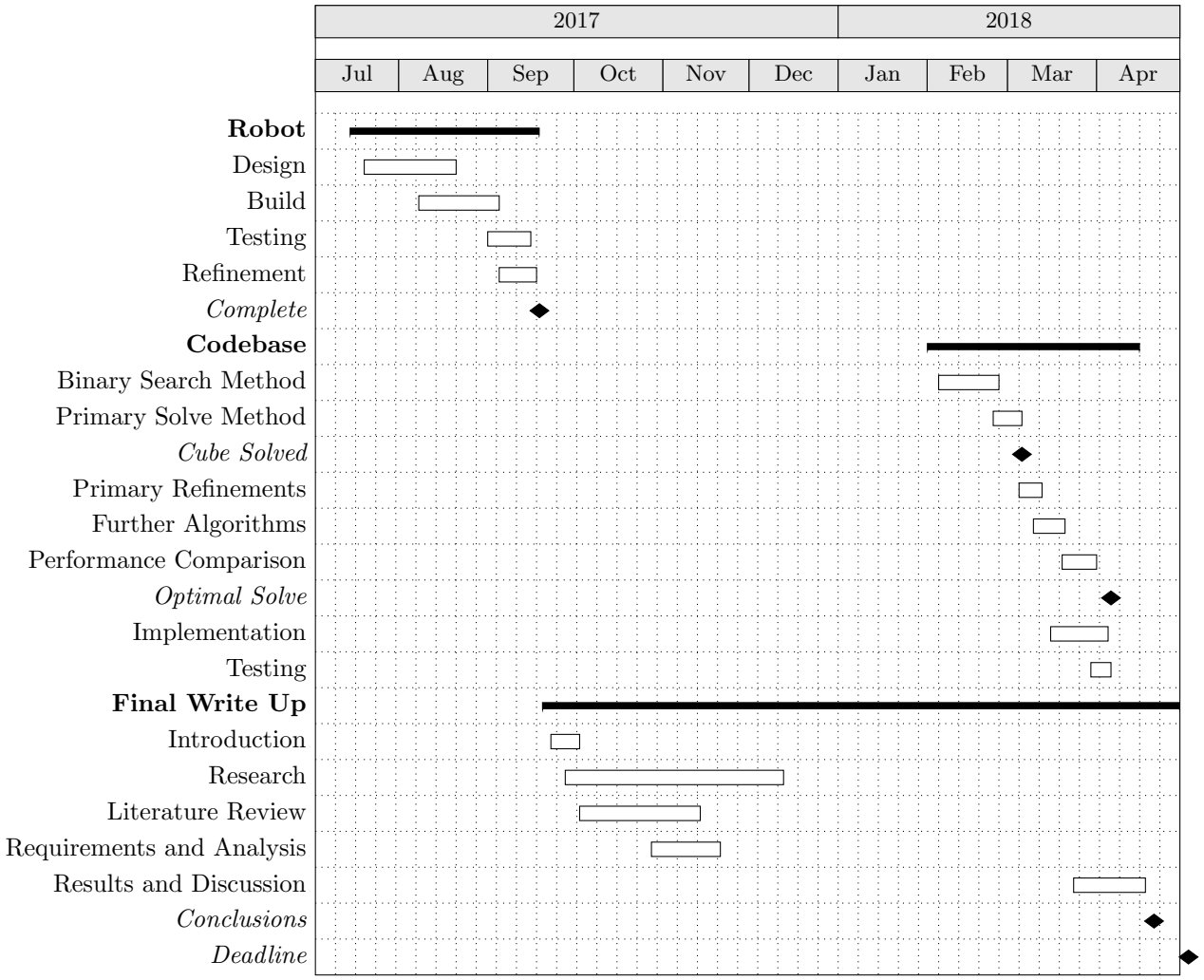
3.4.1 Optimal Solutions

3.4.2 Tree Based Methods

3.4.3 Group Based Methods

3.5 Evaluative Techniques

The progress of this project is estimated to follow the below Gantt Chart. For the duration of its lifespan, there will be a continual measurement of the actual progress against predicted.


2017
2018
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
Robot
Design
Build
Testing
Refinement
Complete
Codebase
Binary Search Method
Primary Solve Method
Cube Solved
Primary Refinements
Further Algorithms
Performance Comparison
Optimal Solve
Implementation
Testing
Final Write Up
Introduction
Research
Literature Review
Requirements and Analysis
Results and Discussion
Conclusions
Deadline

4 Design, Implementation and Testing: The Robot

> I don't spend my time pontificating about high-concept things. I spend my time solving engineering and manufacturing problems.
>
> Elon Musk[32]

4.1 Introduction

The nature of this project lends itself to a spiral design cycle, where both the robot and the solving program can be designed, implemented, tested and then optimised to re-start the cycle. This non-linearity of the *Design* → *Implementation* → *Testing* process means that the structure of this dissertation must follow the progress of the project, rather than having distinct sections for 'Design' and 'Implementation and Testing'. Consequentially the Design, Implementation and Testing component of this document is split into two chapters - 'Robot' and 'Codebase' - each with introductory sections for fundamental components that are invariant through the different iterations, and further sections to discuss each iteration individually.

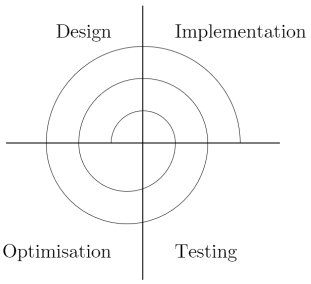


Figure 4.1: The spiral model used for this project

4.2 Robot Fundamentals and Prototyping

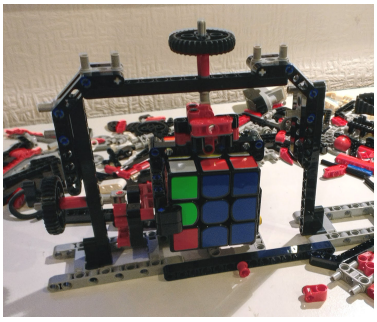
4.2.1 Sensors and Actuators

The Lego EV3 comes with three DC motors (two large (#**45502**) and one medium (#**45503**)), a colour sensor and a touch sensor amongst other peripherals[1]. The quantity of the motors will be the largest restriction in designing the robot: it will only be able to rotate a Cube about two of the three axes; and the colour sensor will only have one degree of freedom so will rely on the rotation of the Cube to scan all of the facelets. The touch sensor can be used to allow limited user input to the robot at runtime, such as an emergency stop if the Cube becomes dislodged, or to confirm that the Cube is in place.

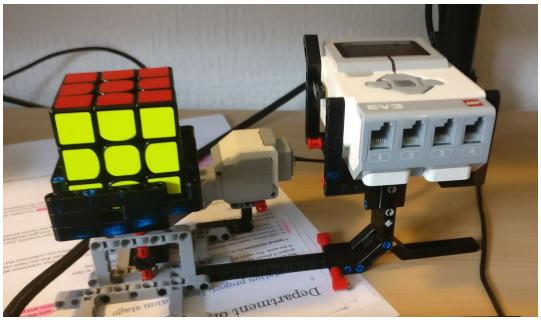
[1] The other peripherals are an infrared sensor and an infrared beacon, and are not used in this project.

4.2.2 Prototyping with Lego

The first step in designing the robot was to prototype some simple designs for holding a Cube and successfully manipulating it. The first concept design was to grip each face of the Cube, and use a large motor to power the 'grippers'. The second large motor could control which gripper was being powered at any time. This would allow a strong manipulation of the Cube and mean no delay between the movement of different faces. Figure 4.2a shows the initial prototype that was built to fit this design. Once it was built, it became apparent that it was actually an impossible design (or at least, impossible with Lego) because if one of the grippers rotated by a quarter turn, it would block two of the other faces from turning. This can been seen below, where the upper gripper is blocking the left-hand one from turning. The solution to this would be to have the grippers retract when they are not in use, however this would be very difficult - if not impossible - to achieve with the available Lego pieces.



(a) The first prototype



(b) The model used to test the motor's torque

Figure 4.2: A cross-section of the cradle and Cube

One of the other problems with the first prototype was that the Cube was unbalanced and would often fall out of the robot. This led to the design to have a 'cradle' for the Cube to sit in. This is a good support mechanism for the Cube, and also handle the rotation about the Y axis. Once the cradle was built, it was a good milestone to check that the large motors had enough torque to rotate the face of a Cube against the friction between faces and slices. The motor was set to rotate indefinitely with the Cube in the cradle (as seen in Figure 4.2b) and the `L`–`R`-slice manually held in place. There was no visible reduction in the rotational velocity of the motor, so this first small test was passed.

Although the cradle was a good solution to the problem outlined above, it also meant that only the `D`-face could be rotated, therefore requiring a mechanism to 'flip' the Cube in place so different faces could sit in the cradle. The most obvious design for this was to apply a force at the upper corner to create a resultant force to pivot about the wall of the cradle. This is shown by the free body diagram in Figure 4.3: the force F will be applied by a mechanism powered by the second large motor; this will be combined with the resistive force R of the cradle wall; the resultant force (shown in red) will be about the pivot point (also in red) to rotate the Cube. Once the Cube is partially out of the cradle, it will slide back into place due to the height difference between the wall and the base of the cradle, and the low coefficient of friction between the plastic of the Cube and the Lego pieces.

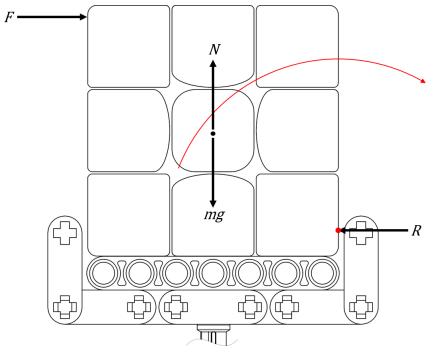


Figure 4.3: A free body diagram showing how the Cube will be rotated

4.2.3 Fundamental Component Definitions

Following the construction and analysis of small Lego prototypes, it became apparent that there would be three fundamental components for the archetypal robot: a cradle would be needed to hold the Cube and rotate about the Y axis; an arm to rotate the Cube about the X axis and hold it in place for performing D-moves; and a moving arm for the colour sensor to scan the Cube. The essential requirements for these components are described below.

Cradle

The Cube will sit in a cradle which rotates about the Y axis to provide the movement needed for the moveset $\{\mathtt{Y}\ \mathtt{Y'}\ \mathtt{Y}^2\ \mathtt{D}\ \mathtt{D'}\ \mathtt{D}^2\}$. This will have an inner dimension of 7 studs × 7 studs (56 mm × 56 mm) to closely match the dimensions of the Valk 3: 55.5 mm × 55.5 mm × 55.5 mm. The 0.5 mm tolerance means that the Cube doesn't have to be absolutely perfectly aligned when it is rotated (i.e. the program won't fail if the cube is a degree out of line). The cradle must rotate as accurately as possible to 90° increments to ensure the success of the X-moves and avoid the Cube falling out of the cradle.

X-Move Arm

To perform an X-move, some kind of arm will need to 'flip' the Cube about its X axis. This should be a quick movement, with high reliability and good structural soundness. The arm must not interfere with the other components of the robot at any point. Furthermore the arm will also need to incorporate functionality to hold the L−R-slice and the U-face in place whilst the cradle rotates to perform a permutation of the D-move.

Colour Sensor

The colour sensor will be attached to the end of an extending arm which can be moved by a rack and worm gear to ensure the sensor can be accurately placed above the correct facelet. The worm gears will have to be driven by the medium sized motor, which produces less torque than the large motors but has a larger maximum rotational velocity - making it ideal for overcoming the inherent speed reduction of the worm gears. The colour sensor will be in a fixed position relative to the arm, and will be centred relative to the Cube.

4.3 Robot Design MkI

When designing the first version of the robot (hereinafter referred to as the **MkI**), there was a clear hierarchy of components and an inherent build order: the X-move arm relied on the cradle to rotate the cube; and the colour sensor relied on both the arm and the cradle to rotate the cube for the scanning process. The design process for the components comprised technical drawings, followed by a real-world build and the concurrent creation of a virtual 3D model using BrickLink's Stud.io software [33]. Rendered images of the virtual model are displayed below alongside the technical drawings to show the finer details of the mechanism and as evidence of the consideration used in the design of the MkI.

4.3.1 Cradle

The cradle is a solid base of 7 studs × 7 studs × 2 studs (length × breadth × height) with a surrounding wall with a height and depth of 1 stud. This ensures that the weight of the cradle is uniformly distributed and the centre of gravity is low. These are important factors in the design of the cradle because a potential design is for it to be mounted on a singular vertical shaft, which is quite flexible under relatively low amounts of strain. The upper layer of the cradle's base has a smooth solid surface - as opposed to the sides of Lego Technic pieces which have lots of holes - to allow the Cube to slide properly when moved

out of the cradle for an X-move; and the lower layer provides an interface to allow mounting on either a vertical shaft or otherwise.

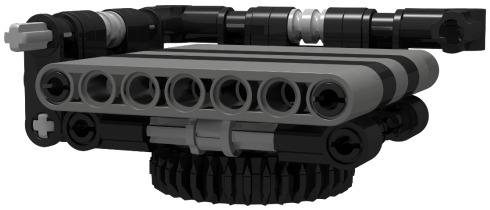
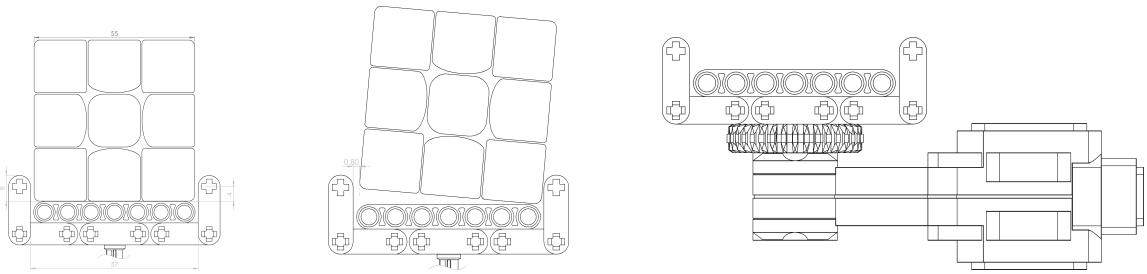


Figure 4.4: A view of the cradle with some parts removed

The wall will have a semi-circular profile to allow the Cube to rotate smoothly out of the base of the cradle as demonstrated in Figures 4.5a and 4.5b. The curved edge will also allow the Cube to slide back into the cradle completely, rather than resting on the upper corner of the edge piece. The method for rotating the cradle is for it to be mounted directly (with no gear train or intermediary mechanism) onto the rotor of a large motor. This gives it stability from the secure mount, high rotational velocity because there is no gearing down, and it will be accurate due to the in-built tachometer which provides 'precise control to within one degree of accuracy' [34]. Figure 4.5c shows the cradle mounted on the rotor via a large gear, which allows a strong interface with the motor and gives the cradle sufficient clearance from the motor to make a full rotation without collision.



(a) Resting position (b) Starting an X-move (c) The cradle is mounted on a motor

Figure 4.5: Technical drawgins showing the the Cube, the cradle, and its mounting

Figure 4.6, below, is a rendered image of the 3D aforementioned model. It shows how the cradle and the motor will be supported by a framework which will link to the rest of the MkI. The main body of the motor is supported both vertically and horizontally to stop it from leaning under its own weight and also to prevent the motor rotating due to the inertia of the Cube.

Figure 4.6: A render showing the original cradle mount design

4.3.2 X-Move Arm

To rotate the Cube about the X axis, a rotating arm will apply the force F shown in 4.3 to cause the Cube to pivot about the opposite cradle wall. The Cube will then slide back into the cradle to complete the rotation. On one end of the arm there will be a wheel with a rubber tire to provide enough friction to tip the Cube over, and the other end will have a set of guards which can hold the top two slices of the Cube in place when performing a D-move or D′-move. The length of the guards will be staggered so that they don't collide with the cradle mid-rotation. Small right-angled pieces have been used to reinforced the corners of the arm, and liftarms will be used to hold the guards together to resist the outwards pushing force exerted on them by the rotating Cube. The design of the arm and its positioning relative to the cradle can be seen in Figure 4.7.

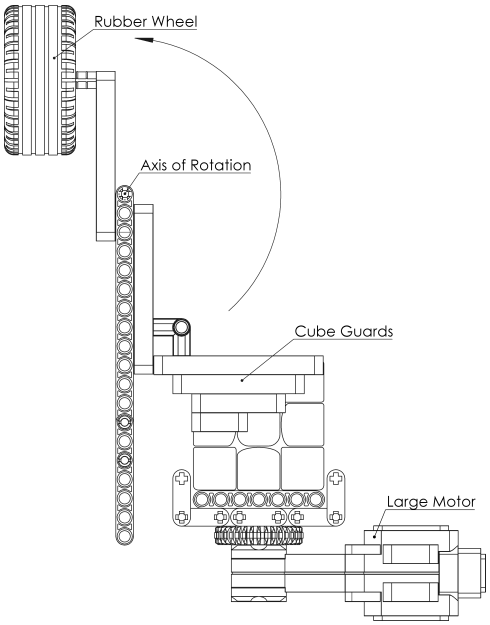


Figure 4.7: The rotating arm used to flip the Cube

Three primary positions of the arm can be seen below in Figure 4.8. Figure 4.8a shows the 'guarded' position of the arm, where the guards are positioned either side of the Cube to hold the L–R-slice and the U-face in place when performing a D-move, D′-move or D^2-move. The two liftarms (#**40490**) that have been attached to the upper part of the guards to hold them together are visible just above the right-angled connectors (#**55615**) used to attach the guards to the arm. The guard assembly is attached to a central axle which is rotated by the large motor visible at the far side of the model. The entire assembly's weight is balanced sufficiently well to ensure that there is a smooth rotation about the axle and no unnecessary strain is placed on the motor.

(a) Guarded position (b) Starting the X-move (c) Unguarded position

Figure 4.8: Three different positions of the X-Move Arm

4.3.3 Colour Sensor

Lego Design

The colour sensor will be powered by the medium motor and a gear train which ends with a worm gear and rack (#**3743**). The rack will be directly connected to a liftarm which has two long parallel shafts (#**3708**) mounted above it, both of which feed through a liftarm attached to the main frame of the MkI. These shafts are the only fixed mounting points for the whole colour sensor assembly - the worm gears and rack provide lateral movement but little to no structural support. The distance between the sensor and the top of the Cube will be approximately 8 mm. From previous tests and forum data, the optimal distance for the colour sensor appears to be between 10 mm and 16 mm [35]. Although the chosen distance is less than optimal, it ensures that the 'beam' from the colour sensor remains inside the facelets. The accuracy of the readings will not be affected.

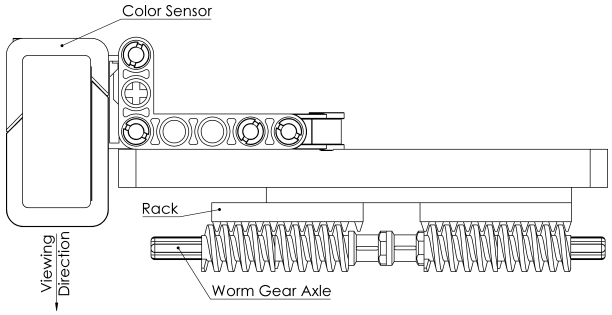


Figure 4.9: The colour sensor mounting arm is driven by a set of worm gears

The medium motor powers a 24-tooth gear (#**60c01**), which then transmits power to an 8-tooth gear (#**3647**), which is on the same axle as the worm gears. The rotational velocity of the worm gear is 63.75 $\mathrm{rad\,s^{-1}}$ (608 RPM), which means that a large 40-tooth gear would have a rotational velocity of 1.59 $\mathrm{rad\,s^{-1}}$ (15.2 RPM). Using this rotational velocity, the surface velocity of the large gear and therefore the lateral velocity of the rack can be calculated to be $\sim$0.0662 $\mathrm{m\,s^{-1}}$. Given that the range of the colour sensor assembly's moveable distance is 80 mm, the time taken for the colour sensor to travel from its neutral position to above the centre facelet is 1.21 s. Whilst this may seem like a relatively long time for the motor to move a light load, the assembly will only have to make the full journey between each face and at the start and end of the scanning process.

(a) A full view of the mechanics

(b) The assembly's mounting structure

Figure 4.10: The colour sensor assembly

Where the motors used for the cradle and the X-move arm can both rotate indefinitely without a mechanical issue (colliding with other parts of the MkI, for instance), the one powering the colour sensor cannot rotate indefinitely due to the limited freedom of the colour sensor assembly. When the assembly reaches the end of its travel, it will be stopped by the liftarm that holds the shafts, causing it to stop the gear train and in turn provide a level of resistance to the motor that could either damage the motor or force the MkI to break apart. For this reason, the 24-tooth gear (#**60c01**) (which is the first in the gear train) is a special type of 'clutch gear', shown in Figure 4.11. This means that it will only transmit

Figure 4.11: The clutch gear used to protect the motor

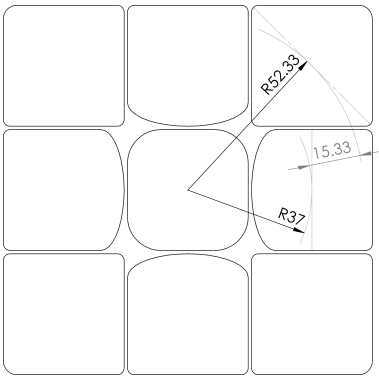


Figure 4.12: A Cube, with arcs drawn at the centre points of the cubies

a maximum torque value of between 2.5 N cm and 5.0 N cm. The theoretical output torque of the gear train is 2.21 N cm, so the clutch gear will function as a regular gear for the duration of colour sensor movement. When the assembly reaches its limit, the resistance to the gear train will cause the clutch gear to slip and the motor can continue turning without damaging itself or the MkI. This all ensures that if an error occurs which causes the motor to remain in a running state, no components will be damaged. Clearly an error of that magnitude should never occur if `try...except` statements are used correctly, but this is a final fail-safe measure.

Figure 4.12 shows that the radial distance between the centre points of the middle and corner cubies is 15.33 mm. The colour sensor will have to travel this distance forty-two times during the scanning process, as well as the distance to the centre six times, making the total theoretical travelling time of the colour sensor approximately 21.26 s.

Algorithm Design

In order to scan a full Cube, the sequence `:X X X Y X X X:` will be followed, and the `U`-face scanned every time a new face is seen, as shown by algorithms 1 and 2:

Algorithm 1 Scanning the Up-Face of a Cube

```
1:  function SCAN_UP_FACE
2:      facelet_list ← [ ]                          ▷ Initialise as empty list
3:      for i ← 0 to 4 do                           ▷ 4 Corner-Edge pairs
4:          MOVE_COLOR_SENSOR(edge_cubie)
5:          facelet_list ← color_value
6:          ROTATE_CRADLE(45)
7:
8:          MOVE_COLOR_SENSOR(corner_cubie)
9:          facelet_list ← color_value
10:         ROTATE_CRADLE(45)
11:     end for
12:     MOVE_COLOR_SENSOR(centre_cubie)
13:     facelet_list ← color_value
14:     return facelet_list
15: end function
```

Algorithm 1 is called six times by the main scanning process, and is used to move the cradle and the colour sensor to the correct position to scan each facelet in turn. In the algorithm, an empty list is declared for the nine facelets to be appended to. `move_color_sensor` takes a single parameter: the stopping point for the motor. The variables `edge_cubie`, `corner_cubie` and `centre_cubie` are all integer constants which are used as these stopping points. The motor is rotated until the tachometer

value is *greater than or* equal to the variable - or less than if it is rotating backwards - so that if it is rotating faster than one 'tacho-unit' at a time, it will not skip the stopping variable and continue indefinitely.

Similarly to `move_color_sensor`, `rotate_cradle` takes a single parameter - but not a stopping point. Rather, it is the value which the current position of the motor is to be incremented by, in degrees. This value will then be multiplied as necessary to account for the gear ratios to create an output rotation matching the provided angle.

Algorithm 2 Scanning an entire Cube

```
1:  function SCAN_CUBE
2:      faces ← [ [ ], [ ], [ ], [ ], [ ], [ ] ]          ▷ Initialise as 6 empty lists
3:      for i ← 0 to LENGTH(faces) do
4:          faces[i] ← SCAN_UP_FACE
5:          if i == 3 then
6:              ROTATE_CRADLE(90)          ▷ Allows L-face and R-face to be scanned after X-move
7:          end if
8:          if i < 5 then
9:              X_MOVE_CUBE                ▷ X-move the Cube to get a new Color on the U-face
10:         end if
11:         if i == 4 then
12:             X_MOVE_CUBE                ▷ Extra X-move to get to D-face
13:         end if
14:     end for
15:     return faces
16: end function
```

4.3.4 Full Model

Figure 4.13: The full robot assembly

Figure 4.13 demonstrates the full assembly of the MkI, including the main EV3 control brick. This model is the final result after any modifications made during the building process. As a result of the in-depth design process and any minor modifications, the MkI was optimised as much as possible with regards to functionality, structural integrity, and efficiency to work under restrictions from piece availability.

The build of the MkI is quite compact, and the design is largely influenced by the pieces available: during the modelling process, small sections of the model were built with actual Lego to check functionality translated from the virtual model to the physical model as intended. Due to the limited quantity of Lego pieces, some less-than-optimal sections arose where two parts had to be connected but the necessary liftarm was not available. This situation is visible behind the cradle, where six small red pieces (#**32062**) are used to connect two right-angled liftarms. Another example of 'design by necessity' is in the colour sensor's gear train: one medium and two small gears were used consecutively instead of two medium, or one large and one small. The functional impact of some of these sections is negligible thanks to workarounds and extra pieces, however the aesthetics have unfortunately suffered minor setbacks in places.

A touch sensor (#**95648**) is mounted on the side of the frame that surrounds the cradle and the X-Move Arm. This adds the ability for user input when the program is running, without needing to access the operating system through the control brick. Simple tasks can be bound to the button at each stage of the program (e.g. pause the scanning process if a move fails or confirm initialisation). The EV3 control brick is mounted on the side of the MkI to allow easy access to the controls which allow navigation of the file system and turning the brick on and off. It is held in place by several long pins (#**32054**) which can be easily retracted to allow quick removal of the brick so the batteries can be changed.

4.3.5 Implementation and Testing

The MkI was built according to the design specifications laid out above, with little to no modification throughout. To thoroughly test the functions of the MkI (and any subsequent designs), a simple algorithm was designed to test and optimise the motorised functions of the MkI.

Algorithm 3 Testing the MkI's Functionality

```
1: function TEST_MK_I
2:     test_pass ← false
3:     repeat
4:         Optimise Cube          ▷ Either modify parameters for motors or do real-world optimisations
5:         SCAN_CUBE
6:         ROBOT_MOVE_SEQUENCE(:Y Y' X Y' D' D² D' X:)
7:         test_pass ← (actual_cube == expected_cube)
8:     until test_pass
9: end function
```

The first part of the test was to run the scanning process from start to end, as described by Algorithms 1 and 2, to test the movement of all three fundamental components. The `repeat...until` loop allows optimisations to be made between each test: for the first few runs, the optimisations were quite simple, such as refining the constants used for setting the stopping points for the components; further optimisations involved making small modifications to the MkI design to reinforce the framework, or changes to the rotational velocity of the motors amongst other changes. The second part of the loop (where the move sequence `:Y Y' X Y' D' D² D' X:` is run) allows the functionality of the MkI during the solving process to be tested. This covers the accuracy of the cradle for `D`-moves and `Y`-moves, the accuracy and the effectiveness of the `X`-move arm's guards, and the success of the `X`-moves. Minor modifications that were easily solved are subsequently ignored for the sake of conciseness and efficiency.

Major Issues

The first major issue with the functionality of the MkI was actually software-based - however it is discussed here because it produced a hardware issue which affected the MkI's design. The details of the code used are discussed in section 5.4.1. In Algorithm 3, the stopping condition for the main loop is a comparison between the virtual Cube and the real-world Cube. This comparison was a manual one, where the position of the Cube was checked against an on-screen printout. The colour sensor has different modes to read colours: `COL_COLOR`, which returns an integer between zero and seven (inc.) to represent a specific colour; `RGB_RAW`, which returns the RGB values read by the sensor; and other modes which read light intensity rather than the colour. For this application, the `COL_COLOR` is the most suitable mode because it will not require the RGB values to be parsed into one of the six colours.

The only problem was, the on-screen printout was always a mix of blue, red and white rather than the expected mix of the usual six colours. The first solution to this problem was to change the magnitude of the space between the colour sensor and the top of the Cube: testing a number of distances between 5 mm and 25 mm produced the same colours, with slight variances in darkness and little else. To properly debug this problem, the mode was set to `RGB_RAW` and the RGB values returned by the colour sensor were compared with values read by the camera on a OnePlus 3[2].

Table 4.1 compares the values read by both sensors, along with the RGB values. The `COL_COLOR` values have been overlaid on the colours returned by the Lego sensor to show how they are interpreted by ev3dev. After many more tests, readings of different surfaces, and hours of research, it became apparent that the issue was that the colour sensor wasn't capable of detecting fluorescent colours. Unfortunately, the Valk 3 only comes in fluorescent colours, a standard (darker) Rubik's branded Cube wouldn't fit in the cradle and has a lower build quality, and other speedcubes weren't a viable option mainly due to their cost.

The solution to this problem was to print readable colours on standard white sticky labels and apply them over the stickers which were being incorrectly read (only the yellow, green and orange sides) on

[2] A simple RGB colour picking app was used [36]

Table 4.1: A comparison of the colour values returned by a OnePlus 3 and the Lego Colour Sensor

Face	OnePlus 3				Lego Colour Sensor			
	Red	Green	Blue	RGB	Red	Green	Blue	RGB
White	229	224	221		79	106	53	White
Yellow	181	195	0		73	97	76	White
Red	223	25	35		52	24	9	Red
Orange	251	127	21		68	93	75	White
Green	34	186	8		25	71	48	Blue
Blue	0	130	255		10	31	31	Blue

the Valk 3. The end result is visible in Figure 4.14, next to a non-stickered Valk 3 (a variation with no black borders, but still the same type of Cube). The orange side has been replaced with brown stickers because the colour sensor reads the five other colours, and then brown - not orange. For the duration of this project however, the colour will be referred to as orange for consistency.

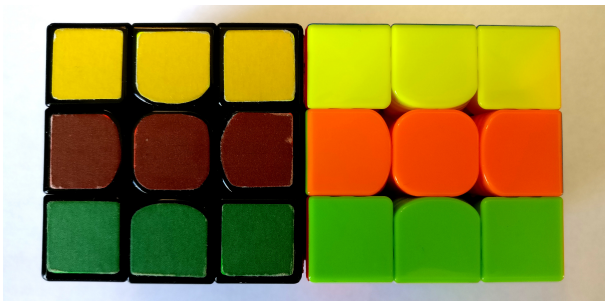


Figure 4.14: Side-by-side comparison of the Valk 3 with and without stickers

Table 4.2 shows that the new stickers greatly improved the accuracy of the colour sensor's readings. Although the colours in the final column of the table are all very dark and almost indistinguishable, the colour sensor apparently has no difficulty in telling the colours apart: approximately 500 facelets were scanned, and none were scanned incorrectly. As such, the sensor's accuracy with the new stickers is sufficient for this project.

Table 4.2: A comparison of the colour values returned before and after the application of coloured stickers

Face	Original Stickers				Modified Stickers			
	Red	Green	Blue	RGB	Red	Green	Blue	RGB
White	79	106	53	White	75	101	51	White
Yellow	73	97	76	White	58	46	8	Yellow
Red	52	24	9	Red	41	18	5	Red
Orange	68	93	75	White	24	14	5	Orange
Green	25	71	48	Blue	11	33	7	Green
Blue	10	31	31	Blue	9	24	17	Blue

The addition of the stickers was a double-edged sword: the coefficient of friction between the Cube and the curved edge of the cradle wall was now too high for the Cube to slide back into the cradle. Rather than spend time redesigning the cradle to be angled to make the Cube slide back in, or changing the arm design to pull the Cube back in, the easiest solution was simply to tilt the entire robot:

Figure 4.15: The MkI had a distinctive tilt to ensure the success of X-moves

The second major issue with the MkI was caused by the newly added tilt and the design of the X-move arm: when the wheel on the end of the arm pushed the Cube to start an X-move, the Cube would often fall out of the cradle instead of pivoting about the opposite wall. There was no obvious solution to this problem, other than making the wheel larger to spread the force across the face of the Cube. However, a larger wheel meant that the arm became unbalanced and the strength of the arm was compromised. This led to the liftarm that held the wheel falling off the MkI mid-process more than once, as shown in Figure 4.16 (three frames extracted from a video of the scanning process).

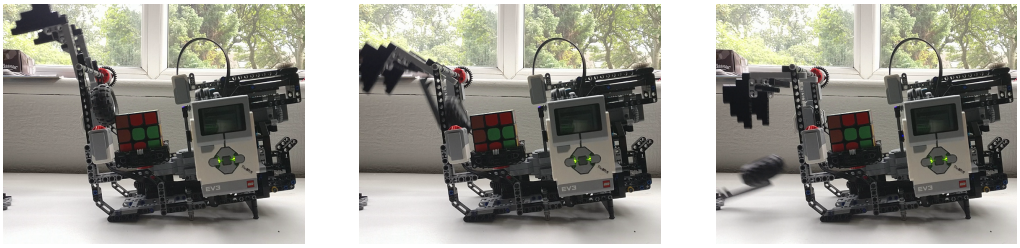


Figure 4.16: The MkI X-move arm broke under greater resistance

The final issue with the MkI, which prompted a complete rebuild, was an inherent issue with the large motor that rotated the cradle. As with all DC motors, there is a slight amount of 'play' in the rotor: it has a few degrees of freedom either side of its current position. This is usually not a problem, but because the cradle requires very high accuracy it can cause a complete failure for any part of the process. The play in the motor reduced the accuracy of the cradle, as it effectively introduced a zero error for each rotation. If the cradle only rotated in one direction then the zero error would only have to be offset at the very start of the program because the cradle would 'sit' at one end of the region of play. However, this would have reduced the size of the available moveset by 50% and thus have had a worst case runtime increase of 100% (Sequentially, $|$`:D' X:`$|$ = 2 would have changed to $|$`:D D D X:`$|$ = 4). A problem of this magnitude required a full analysis and re-design of the robot.

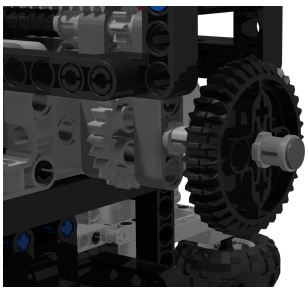
4.4 Robot Design MkII

4.4.1 Improvements on MkI

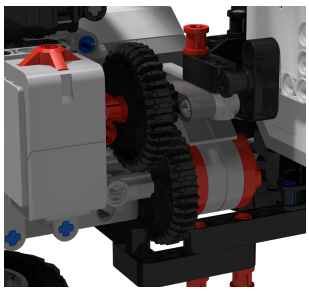
Aside from the few major issues outlined above, the time spent testing and using the MkI led to the discovery of many more smaller issues that would be addressed during the design of the second iteration of the robot's design: the MkII. These minor issues are discussed below, as proof of the careful consideration used and to validate the choices made in designing the MkII.

Motor Initialisation

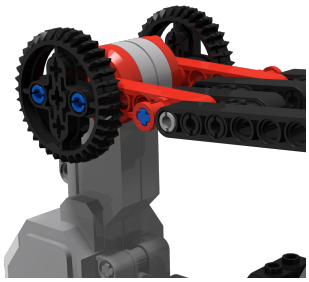
During the testing and general usage of the MkI, there were many occurrences of a fatal runtime error which would leave the motors out of their starting position. The usual cause of this was a `StallError` from one of the motors or the voltage supply from the batteries becoming too low. The incorrect positioning required the motors to be initialised when restarting the program. To assist in this procedure, large black gears were added to the motors which allows them to be returned to their respective initial positions. A method was also added to the program disable the active braking on the motors so they weren't damaged when being rotated.



(a) Colour Sensor initialiser



(b) Cradle initialiser

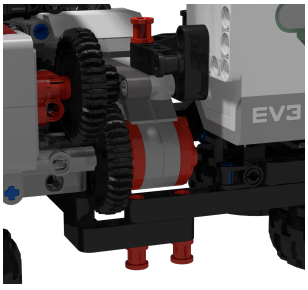


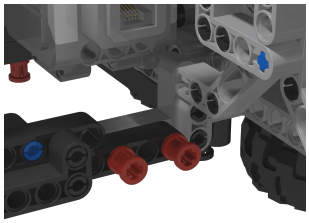
(c) X-move Arm initialiser

Figure 4.17: The initialising gears used for each motor

Fragility and Adaptability

One of the MkI's greatest flaws was the fragility of the frame that held the X-move arm in place. It was also very difficult to make any adjustments to the MkI because of the complexity of the design. With this in mind the MkII was constructed of two main modules, each of which could be structurally tested independently. These modules were then connected with seven red pins (#**32054**) that are easily accessible: if any modifications need to be made to the MkII, the pins can be removed and the modules separated for quick and efficient modification.





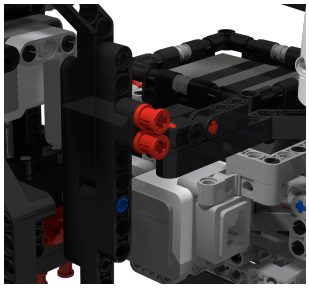
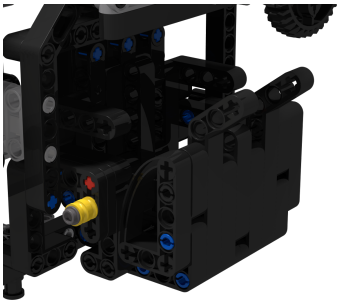


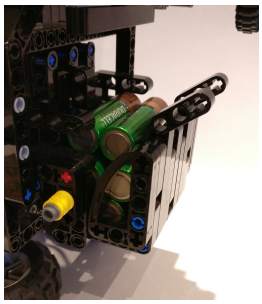
Figure 4.18: The pins used for connecting the modules

Battery Usage

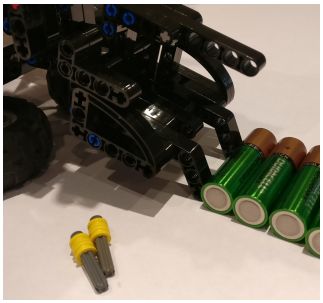
The final of the issues discovered by repeated testing of the MkI was how power-hungry the EV3 control brick really is. After only one to one-and-a-half hours of testing, it would run six AA 2500 mA h batteries completely flat. Lego do sell a rechargeable battery for the EV3, however is is only 2050 mA h by itself and its costs £81.99 [37]. A cheaper (and higher capacity) option was to purchase twelve rechargeable 2500 mA h batteries and a charger for approximately £25. Whilst the EV3 was voraciously consuming the power from six of the batteries, the other six could either be charging or waiting ready. This led to the design and creation of a battery holder attached to the MkII. It holds six AA batteries and has two pins which can be removed to release the batteries, as shown in Figure 4.19c.



(a) The design of the battery holder, rendered



(b) Six AA batteries ready for use

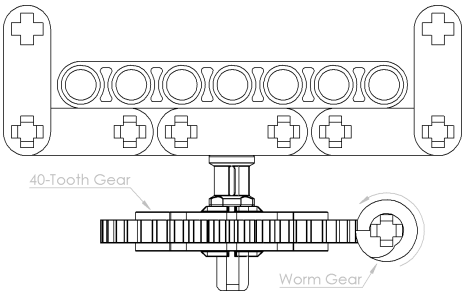


(c) The battery holder is opened by removing the pins

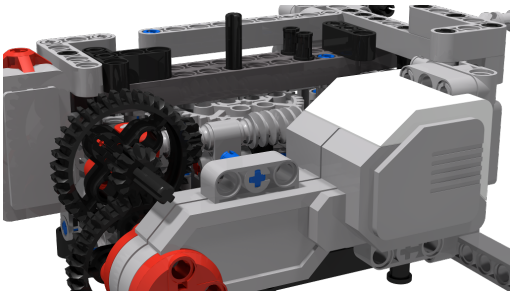
Figure 4.19: The battery holder, as a rendered model and in real life

4.4.2 Cradle

The MkI implementation of the cradle largely failed because of the play in the large motor upon which it was mounted. The accuracy of the motor was reduced and the rotations made unreliable. A worm gear (#**4716**) has solved both of these problems: it has been combined with a large gear (#**3649**) to create a gear ratio of 1:40, increasing the accuracy of the motor by a factor of forty (see Figure 4.20a). This also slowed the maximum rotational velocity of the cradle to 0.209 $\mathrm{rad\,s^{-1}}$ (2 RPM) at 7.5V, so a 90° turn will take 7.5 seconds[3]. A compromise between maximum rotational velocity and accuracy has been found by adding more gears to reduce the ratio. The problem presented by the play in the motor has been successfully minimised due to the unidirectional nature of the gear train.



(a) The rotation of the worm gear drives the cradles rotation



(b) The gear train which provides power to the cradle

Figure 4.20: The battery holder, as a rendered model and in real life

Figure 4.20b shows how the cradle is driven by a large motor. A large 36-tooth gear (#**32498**) is directly attached to the rotor with two small pins to reduce the need for an extended shaft which will flex when placed under strain (i.e. at the start of a rotation). This gear drives a 12-tooth gear (#**32270**) to triple the rotational velocity at that point of the gear train. This connection is immediately repeated and used

[3]This figure is the result of a measurement of the maximum rotational velocity of a large motor, and will fluctuate depending on the voltage supplied by the batteries in the EV3 brick

to create a 90° turn onto the shaft with the worm gear. At this point the speed ratio between the worm gear and the motor is 9:1, which is then decreased forty times by the large 40-tooth gear to create a final ratio of 1:4.444. This means the maximum rotational velocity of the cradle will be 1.843 $\mathrm{rad\,s^{-1}}$ (17.6 RPM) at 7.5V and the original torque is increased from 17.3 N cm to 76.8 N cm, which will allow a face of the Cube to be rotated with ease. The vertical black shaft seen at the top of the image provides a mounting point for the cradle.

4.4.3 X-Move Arm

The largest change between the two design iterations was the complete update of the X-move arm - in functionality, aesthetics, and reliability. Where the MkI X-move arm relied on pushing the Cube, the MkII *pulls* the Cube towards it. This creates a much more controlled movement, as the Cube is bound on all four sides at all times. As shown in Figure 4.21, the Cube cannot fall out of the cradle during the 'pull' part of the rotation due to the small right-angled piece behind the cradle. The pulling motion causes the Cube's rotation, then the Cube is immediately pushed back into the cradle to complete the move.

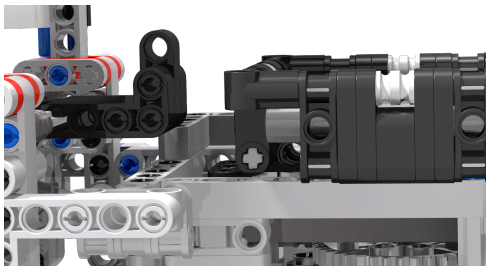


Figure 4.21: A side elevation to show the pieces which block the Cube from falling

The arm is controlled by a single large motor, and is suspended from a tall frame by a liftarm which is free to pivot at both ends. This means that the arm can move *along* the Z-axis as well as about the X-axis - albeit with limited freedom. Figure 4.22 show the incremental steps for an X-move:

a Starting position, the front of the arm is just touching the F-face

b Maximum pull position, the Cube has rotated and is resting on the block piece from Figure 4.21

c The Cube is pushed back into the cradle

d Push is completed

e The arm moves out of the way to end the X-move

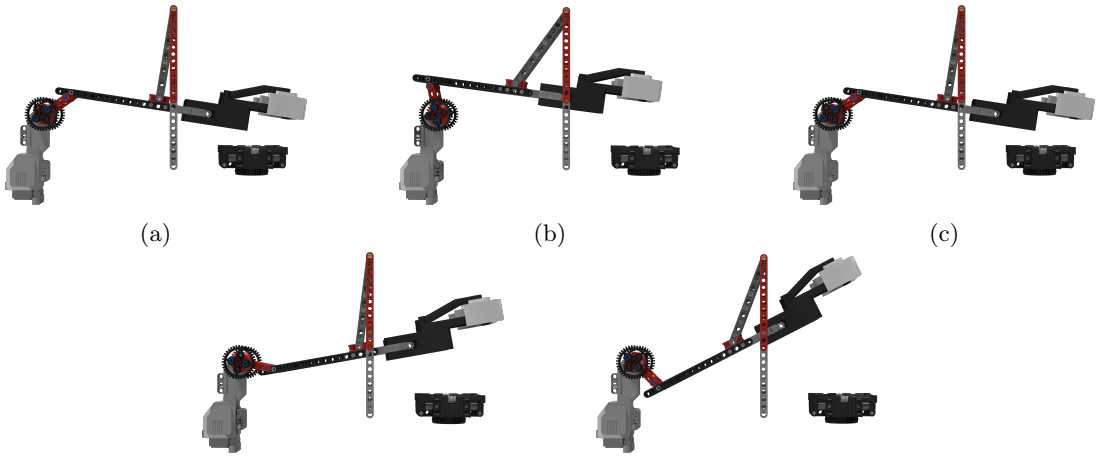


Figure 4.22: The steps of an X-move, in order

4.4.4 Colour Sensor

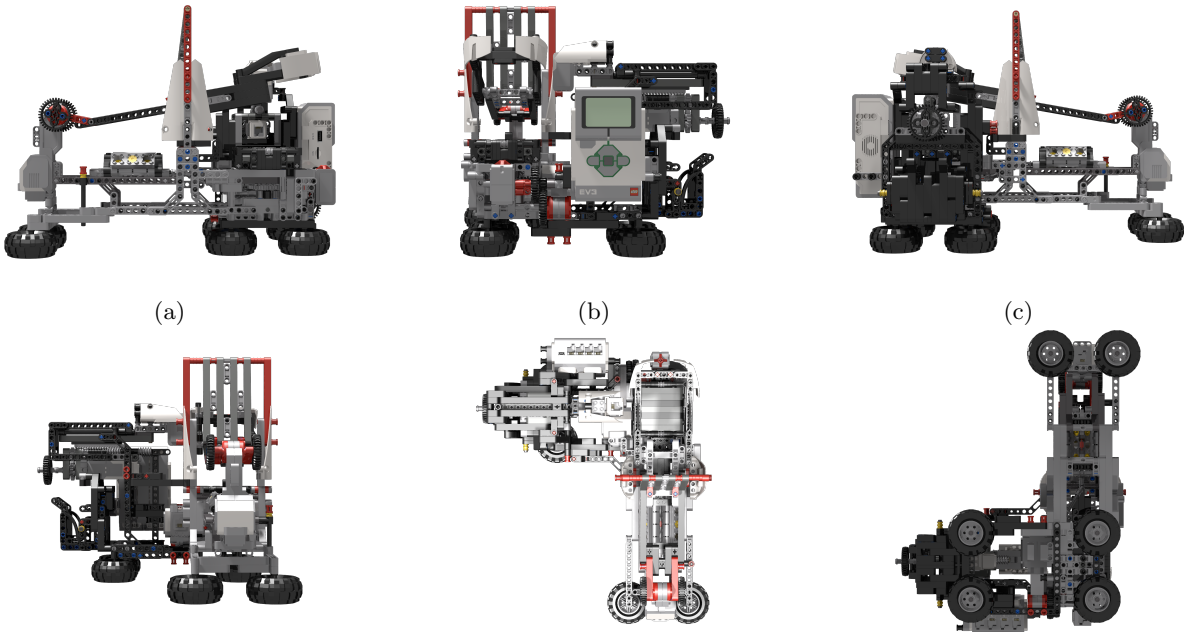
The colour sensor was undoubtedly the most successful component of the MkI, and very little has been changed between design iterations. The clutch gear (#**60c01**) used in preventing any damage to the motor of the MkI framework actually became a hindrance: if the assembly encountered any resistance when moving (e.g. when the rack moves onto the second worm gear), then the clutch would slip and then the whole assembly would be out of line. This rarely caused incorrect scanning results, however it is easier to restrict damage by following good coding practices - which were developed with the use of the MkI - than with the clutch gear. The only other functional change to the colour sensor was the shortening of the axle which it rested on: the worm gear closest to the cradle was superfluous so was removed to avoid over-complicating the mechanism.

4.4.5 Full Model

Figure 4.23: The full MkII robot assembly

Figure 4.23 is the full assembly of the MkII. As with the full render of the MkI, the model has been updated to reflect any modifications made during the building process. The construction of the MkII is of much higher quality, with fewer shortcuts and workarounds used to make up for missing pieces. It is also considerably larger than the MkI due to the increased size of the X-move arm. The extended framework initially caused some flexing across the body of the MkII simply due to it's size and the forces exerted by the movement of the X-move arm. A very simple solution to this was to mount the entire MkII on 'feet' made from wheels on their sides. The flexibility of the tyres allows the feet to act as dampers for the X-move arm, and the high friction between the tyres and the surface they sit on means that the MkII will not move around. The framework which the EV3 control brick is mounted on is actually mostly the same design as the MkI: this was one part there truly were no issues with. Reusing the framework meant less time had to be spent on the more mundane parts of the build, and more time could be spent

getting the core components right. Once again, the touch sensor has been mounted on the front of the robot for easy access at runtime.



(a) (b) (c)

(d) (e) (f)

Figure 4.24: Elevation and plan views of the MkII

4.4.6 Implementation and Testing

The implementation/building process of the MkII was fairly straightforward following the MkI's implementation. The more common errors and weaknesses in the design were easier to spot, and the knowledge gained from the previous design iteration greatly aided in making the MkII an all-round strong design. The use of the spiral model in designing the robot has generated a clear process from the design stage through to the testing stage, producing a robot with very few, if any, hardware-based bugs or issues.

The definitions described in section 4.2.3 have been briefly quantified below for quick referencing:

Cradle

1. The cradle should rotate about the Y axis to perform the moveset {`Y Y' Y2 D D' D2`}
2. The cradle must rotate as accurately as possible to 90° increments
3. The Cube must not fall out of the cradle

X-Move Arm

1. The `X`-move should be a quick movement
2. The `X`-move needs to be highly reliable and structurally sound
3. The arm must no interfere with other components
4. It must hold the `L`–`R`-slice and the `U`-face in place to perform a permutation of the `D`-move

Colour Sensor

1. The sensor must be accurately positioned above the correct facelet

Each of these definitions have been followed in the design of the robot, through thorough consideration of hardware capabilities and leaving plenty of room for improvement with both hardware and software. The MkI did not achieve items Cradle 2 and 3, and X-move Arm 2 so these were the largest changes as the spiral model progressed. The MkII, however, has successfully fulfilled all of the definitions with a very small margin of error. It has therefore been marked as a success, and will not be developed further.

5 Design, Implementation and Testing: The Codebase

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> C.A.R. Hoare[38]

5.1 Introduction

This is the second of the two chapters on the Design, Implementation and Testing of the project. This chapter covers the methods used, code written, and algorithms implemented to solve a Cube with the robot built in Chapter 4. Three main methods were implemented, with varying degrees of success, and will be discussed in their own sections. Prior to the details of the solve methods, the general structure of the software is explained to aid in understanding the methods' advantages and disadvantages.

The general outline of the software is shown in Figure 5.1. Regarding the decision 'Is Cube valid?', the most thorough method of validating a Cube is to check each individual cubie for correctness and validity - for this purpose, this is unnecessary because an invalid Cube is outside the parameters of the project's objectives. The validation used after scanning the Cube is simply to check the right number of colours have been scanned, to double check there were no issues with the colour sensor. If the Cube is invalid, it will only be rescanned once: after the second attempt, an invalid Cube will be rejected so that the program doesn't get caught in an infinite loop.

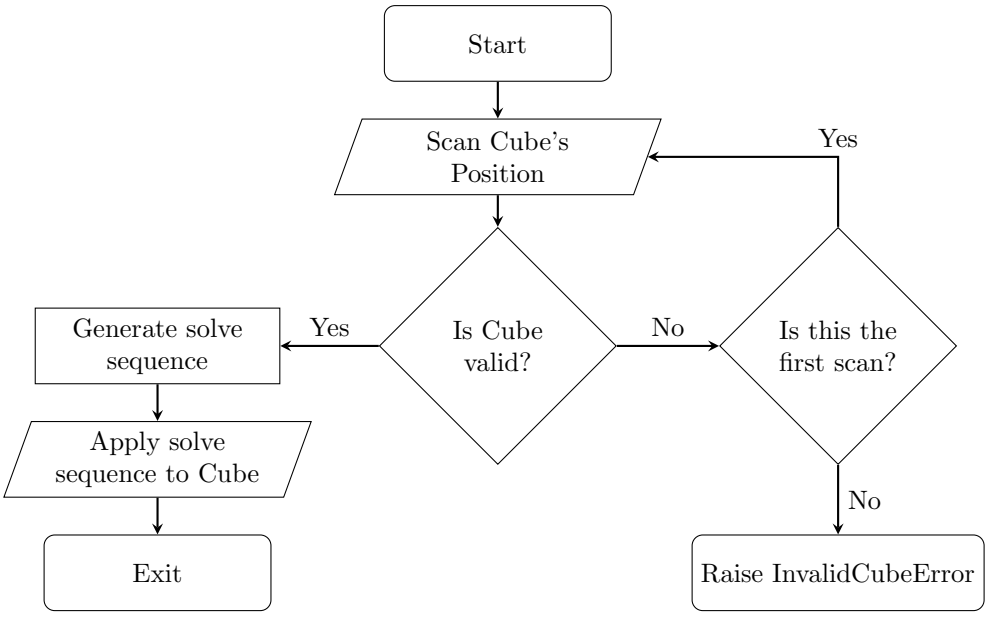


Figure 5.1: Flowchart to show the processing in the software

5.2 Device-Specific Software

The hardware capabilities of the EV3 control brick are perfectly adequate for controlling the EV3's sensors and actuators, performing non-processor intensive tasks, and managing wireless connections. However, it is absolutely inadequate for solving a Cube: one of the computers used by Richard Korf to generate a solution had an (at best) 167 MHz processor, with 64 MB of RAM, and he gave estimates of four weeks to search to depth-18 [22]. The control brick has a 300 MHz processor and 64 MB of RAM - not much better than Korf's computer from 1997. The computer (referred to as the DevPC) that is being used for developing this project has a 3.6 GHz processor and 32 GB of RAM, meaning that the processing will be performed much quicker on it than on the EV3. If both the EV3 and the DevPC are to be used, two programs which connect to each other need to be written.

Python's '*socket*' module is used to connect the two programs: the DevPC creates a socket and binds it to a set port on its local IP address to listen for connections, and the program waits until a device connects; the EV3 creates a socket and attempts to connect to a hard-coded constant IP address. The timeout on the sockets is over two minutes, so the programs do not have to be started simultaneously. The methods used to establish the connection both return a connection object, which can be used to send and receive data at any point in the program. The sockets can be run asynchronously, but there is no advantage to this as there are no processes which can be run concurrently in the software, as shown above.

With the introduction of a second program, the flow chart in Figure 5.1 has been updated to reflect this in Figure 5.2 (overleaf). The flowchart is divided into the processes performed by the EV3 and those performed by the DevPC. The first process on each side is 'Connect to [other device]': to be accurate, these should be followed by a decision to check if the connection has been completed successfully, and if not, the chart should go back to the connection process. This has been omitted to avoid overcomplicating the diagram, and can be assumed to be in place on both sides.

One of the processes added to this flowchart that wasn't mentioned in Figure 5.1 is 'Translate solve sequence'. This is a vital step in solving a Cube with a Lego Robot, and is explained below.

5.3 Move Sequence Translation

The algorithms considered and discussed in the Literature Review (as well as others used in determining God's Number) are designed to work with a standard moveset {L R U D F B} (as well as half turns and counter-clockwise turns). The design of the robot means that it is restricted to the moveset $\{\mathtt{X\ Y\ Y'\ Y^2\ D\ D'\ D^2}\}$ - and will therefore require any incompatible sequences to be translated to the robot's moveset.

When the entire Cube is rotated, the *colour* of the faces will change but the faces' *orientations* will not (analogous to when one turns on the spot: left and right have changed, but east and west have not). This means that the frame of reference for the moves will have to remain invariant throughout a sequence by compensating for any full Cube rotations.

This is best explained with an example: the first step in performing the sequence `:U F:` would be to do perform $\mathtt{:X^2\ D:}$ in order to complete the U-move; but now that the original F-face has moved (due to the $\mathtt{:X^2\ :}$) - it must be located, the Cube must be rotated to get the original F-face into the cradle, and then the sequence can be completed. The simplest method of overcoming this complication is to convert the original move sequence to an equivalent sequence of colour-based moves: `:U F:` $\equiv$ `:W G:`.

Once the original sequence has been converted to a colour-based sequence, a deep copy of the original Cube is made and the coloured sequence is iteratively translated and applied to this Cube. On each element of the sequence, the colour is converted back to a face move (allowing preservation of the correct frame of reference). Once this new face-based move is found, it is converted to a short move sequence that will move the face into the cradle and rotate the face correctly to match the original move. This short sequence comprises moves compatible with the robot.

Each of the moves in the short sequence is applied to the copied Cube, and appended to a list of moves which is returned once the original sequence is processed. The application of the moves to the Cube

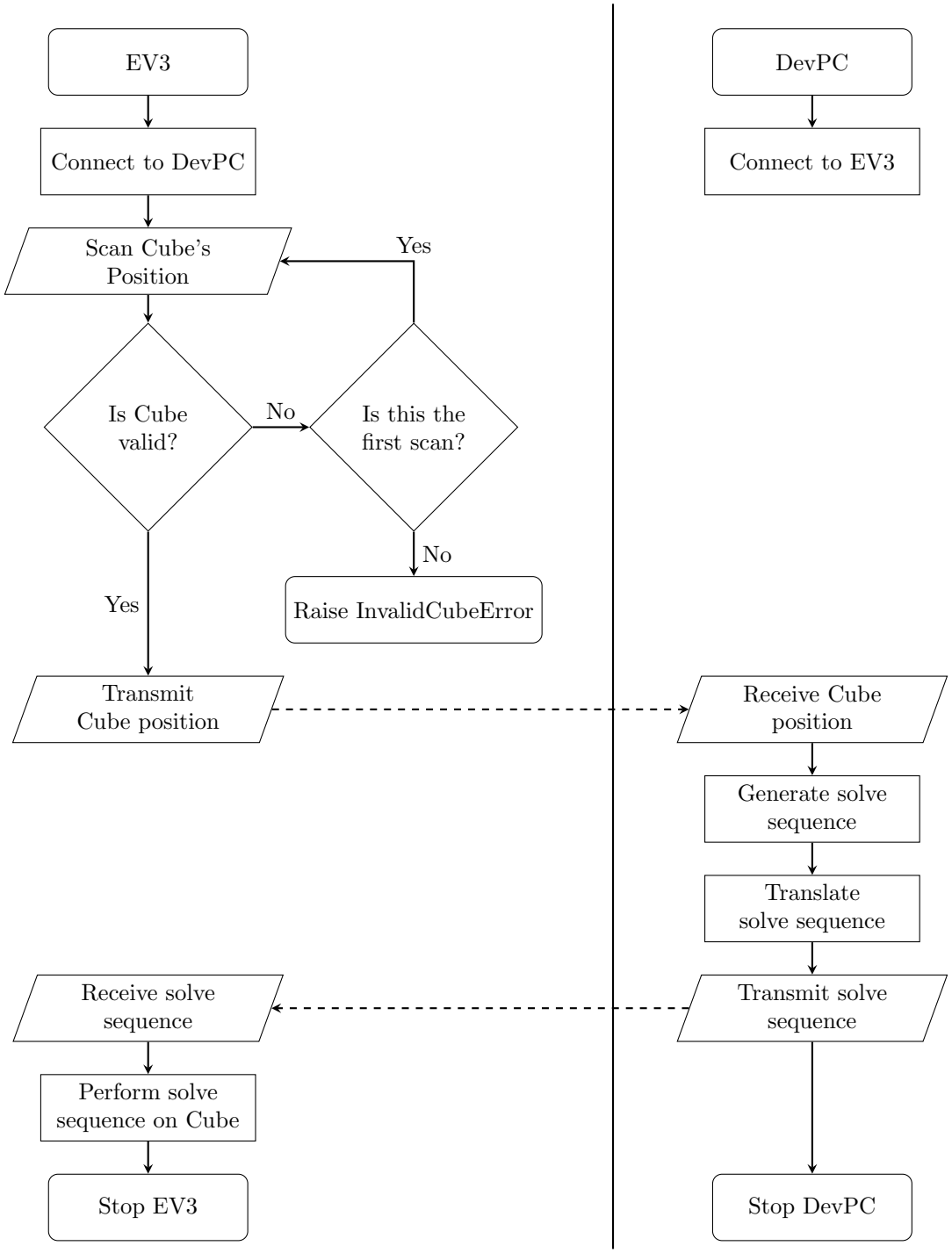


Figure 5.2: The EV3 and DevPC work synchronously but separately

allows following moves to be translated correctly (as the Cube will have changed orientation in real life so needs to in this simulation). Continuing the previous example of `:U F:`, the translation is written

below. Colours are underlined and the output sequence is emboldened for clarity:

:U F: → :W G:
  :W: → :U:
  :U: → :X^2 D: → **:X X D:**
      ⟹ :U F: → **:X X D** G:
  :G: → :B:
  :B: → :X D:
      ⟹ :U F: → **:X X D X D:**

This final move sequence will then be transmitted to the robot and used to control the motors to manipulate the Cube.

5.4 Object Classes

This project has two main object classes, which are created at the start of the program and used throughout. These classes are both discussed, explained and examined below as an introduction to the methods used in solving a Cube.

5.4.1 Robot

The first class is the `Robot`: an object which is used to represent the real world Lego robot, provide a controller for the actuators, and read real world data using the sensors.

The `Robot` class has one class attribute for each peripheral (sensors and actuators) to allow individual control. Upon initialisation of the `Robot`, each peripheral's connection is checked and the motors are then manually initialised. To do this, the braking method for the motors is set to `coast`, which means that all resistance is removed from the rotor and they can be turned to the correct position using the large gears shown in Figure 4.17 on page 24. Once the motors are initialised, the touch sensor can be pressed to continue the program: this sets the braking method of the motors to `hold`, providing an active resistance to the rotors to ensure there is no unwanted rotation.

The `Robot` has a number of methods for controlling the core components: Algorithms 1 and 2 have been implemented to scan the Cube; constant values are used to set the endpoints of an `X`-move performed by the arm; `rotate_cradle` is a method which has an optional parameter (with a default value of 90) by which to rotate the cradle; and there are moves for each method in the set {X X′ X^2 Y Y′ Y^2 D D′ D^2} (The `X′`-move is interpreted as `:X X X:`).

5.4.2 Cube

The `Cube` Class

The second object defined in the software is the `Cube` object. As well as the main `Cube` class, there are four `Enum` classes which are used in defining a `Cube`: `Color`, which has eight members - each representing a colour which the Lego sensor can 'see' (`NONE`, `DARK`, `WHITE`, `ORANGE`, `GREEN`, `BLUE`, `RED` and `YELLOW`); `Move`, an Enum with one member for each possible move; `Face`, which has one member for each Cube face; and `Rotation`, which is a simple Enum with only two members - `CLOCKWISE` and `COUNTER_CLOCKWISE`.

To store the position of a Cube, the three-dimensional Cube is turned into a two-dimensional net, which is then turned into a single string by 'reading' the facelets from top to bottom, left to right as shown in Listing 5.1. The `position` attribute can be accessed by calling `Cube.position`, and a coloured net can be output by printing the `Cube` object. A `Cube` has two optional parameters upon initialisation. The first is the position, as a string. If this is omitted, it defaults to the solved position. The second argument is a boolean to state whether the Cube is temporary. If set to `True`, some less commonly used attributes and methods are ignored to reduce memory usage and processing requirements.

```
>>> from cube_class import Cube
>>> cube = Cube()
>>> print(cube.position)
WWWWWWWWWOOOGGGRRRBBBOOOGGGRRRBBBOOOGGGRRRBBBYYYYYYYYY
>>> print(cube)
      W W W
      W W W
      W W W
O O O G G G R R R B B B
O O O G G G R R R B B B
O O O G G G R R R B B B
      Y Y Y
      Y Y Y
      Y Y Y
```

Listing 5.1: The two different ways of accessing a Cube's position

There are also six class attributes to represent the faces of the Cube. Each of these is a string of six characters, each representing the colour of a facelet when the face is 'read' top to bottom, left to right.

Updating a `Cube`

In order to replicate each of the twenty-seven possible moves, twenty-seven corresponding methods were created to simulate each move. They make use of the `Cube`'s setter methods, which each take a single string as an argument and set the corresponding characters in the `position` attribute. They also use the method `rotate_face` which takes a direction (of type `Rotation`) and a face (of type `Face`) and moves the facelets 90° in the given direction.

To ensure that all the class attributes accurately describe the Cube's position at any given time, methods have been written to update all of the necessary attributes when the object is modified. As well as the setter methods, there are two getter methods. The first takes a keyword argument and returns a `Color` object: the keyword is either `'facelet'` which is an integer facelet reference, or `'face'` to lookup that face's colour. The second getter method takes a `Color` as an argument and returns the face which is that colour.

5.5 Solve Method 1: Tree Solving

During the development of this project, three main methods of solving the Cube were implemented. Each had it's own advantages and disadvantages and are discussed in this section and the following two.

5.5.1 Implementation

The very first step in creating a reliable solve method was to create a reliable object class, and a data structure to store the object in. The `Position` class defines an object with four attributes: the depth of the position; the position's string representation; the move sequence used to get to that position; and the ID of that position. Richard Korf discusses using a search tree and an iterative deepening A* algorithm to solve a Cube in his 1997 paper [22]. Korf also mentions using the Manhattan distance of the cubies as a heuristic to help improve the efficiency of the search algorithm.

It is important to note that a lot of the research into efficient solving algorithms was done between twenty and forty years ago, and is now lost to the abandonment of websites, mailing lists, and newsletters in favour of much simpler websites and blog posts with much less information and community interaction. Some of the data from the 'golden age' of solving Cube's is available as large, disorganised archives - but a lot of it was done before the days of commonplace online interaction, and the amount of data available to assist in the design and implementation of solve methods is far more limited than is desirable if one has no contacts in the world of Rubik's Cube solving. This all means that some of the development

hit a premature dead end and was abandoned in favour of attempting a larger range of methods for cross-comparison.

Having evaluated the available information about the use of search trees and spaces in solving a Cube, it seemed like it could actually be a fairly trivial task: use a moveset to manipulate a Cube until it is solved; a lot of positions can be eliminated through duplication checking and other optimisations; and the code to create a recursive search space will also be relatively simple. This assumption was soon proved very wrong.

A dictionary was used for storing the search space, with the positions at each depth stored as the values and the depths as the keys. Inside a `while not solved` loop, a nested `for` loop iterated through the values in the dictionary at the previous depth, and a second nested `for` loop applied each move in the defined moveset to each position. The only optimisations applied to the generations was the use of a set to hold the previously generated positions for duplication checking. This optimisation reduced the time required to solve a Cube at depth-n by an approximate factor of $0.45n$, and the number of positions by $0.58n$.

In order to observe the progress being made by the tree generator, a graphical user interface (GUI) was created to display the time elapsed, current depth, position count and move sequence, alongside a graphical representation of the Cube. To take full advantage of Python's versatility, the '*multiprocessing*' package was imported to separate the generation and GUI processes on to different cores in the computer's CPU. Multiprocessing was chosen over multithreading due to the complete separation of processes - different threads still operate on the same CPU core so don't take full advantage of a computer's processing power - and inherent increase in processing speed. Although this complete separation meant the processing speed was increased, it presented a challenge in getting the positions from the generator to the GUI across completely distinct memory locations.

A `LifoQueue` was used to bridge the gap between the two processes. It was used as a parameter in the creation of the two processes, providing a pointer back to its memory location whenever necessary. A *LIFO* queue was used rather than a *FIFO* queue because positions were generated faster than the GUI could update, causing a lag in the GUI as it worked through the ever-growing queue and reducing its effectiveness. The disadvantage of the LIFO format was that positions which were skipped by the GUI were left in the queue with no further use. Although this increased total memory usage, the older positions were committed to a pagefile, reducing the impact on memory to an acceptable level.

5.5.2 Testing

The tree solving method is run with a single argument: the position of a Cube, either returned by the robot after scanning a real Cube or entered manually. The first test for the tree solve was the position of a Cube at depth-3 (created by `:L'D'U':`). The solve time for this depth was 0.522 s - but it soon became clear that this was best case for depth-3. The solve time varied greatly depending on the move sequence used to mix the Cube (assuming the invariance of the tree generator's moveset): the closer the moves used were to the start of the moveset, the closer the solve position was to the start of the depth in the tree.

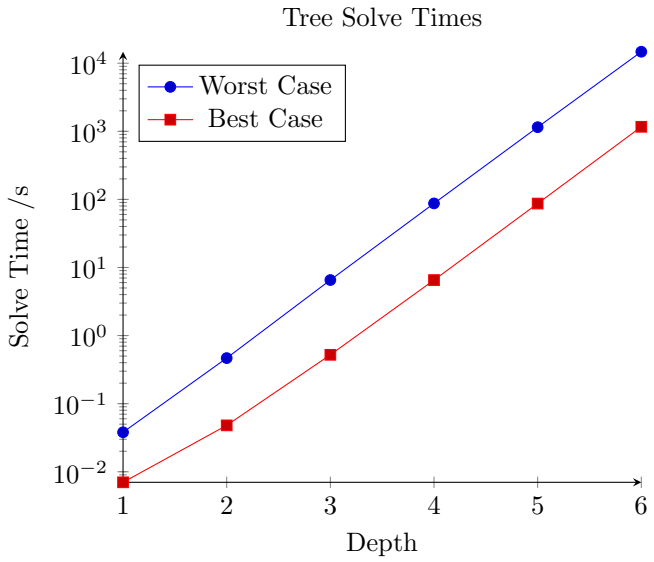


Figure 5.3: The time difference increased exponentially for each depth

Figure 5.3 is a graph of the solve depths against the time taken to solve them, and shows the best case and worst case times for each depth. The data used to create the graph is attached in Appendix A.2, and shows the sequence used to mix the Cube as well as the solve time and the number of positions generated to find the solution. The position numbers for the depth-n worst case and depth-$(n+1)$ are consecutive, which proves that the positions are definitely the best and worst cases for each depth.

The Y axis of the graph uses a logarithmic scale because the time taken (regardless of best or worst case) for each depth increased by an average factor of 13.59. This also meant that the variance in the times increased exponentially, with a recorded maximum of 15053.42 at depth-6. The average growth factor for the solve time times can be used to generate a predicted time for each depth by using the following formula, where t_d is the time to solve depth-d:

$$\text{Best Case:} \quad t_d = 1165.35 \times 13.59^{d-6}$$
$$\text{Worst Case:} \quad t_d = 15053.42 \times 13.59^{d-6}$$

However, the prediction breaks down past depth-17, where the predicted number of unique positions surpasses the maximum number of available positions. The prediction of unique positions per depth follows a constant trajectory, whilst in reality the number at each depth increases steadily then sharply decreases at depth-17. The predicted times and positions are listed in Appendix A.3.

The extensive time required for a solution to be returned meant that the method was completely impractical past depth-5. Further optimisations to the tree generator were either too small to make a sizeable difference, or too complex or poorly documented to be implemented successfully. An example of the latter is the use of 'pruning tables', which is discussed later.

5.6 Method 2: Multiphase Solver

5.6.1 Design and Implementation

Having accepted the implementation of a tree-based solve method to ultimately be non-viable, the next best method was to follow the research performed decades ago when Rubik's Cube was first being efficiently solved by computers. Of the eleven major developments made in finding God's Number [1], the majority rely solely on group theory combined with optimisations. The main form of documentation for these developments was a mailing list that approximately three hundred people subscribed and contributed their work on Rubik's Cube to [39]: the most complete archive of the mailing list was retrieved from a German research university [39], [40] to try and extract the most useful method. After

tedious hours spent poring through the archived emails, Hans Kloosterman's method of solving a Cube in forty-two moves seemed the most viable for this project. Rik van Grol, a Dutch mathematician and Rubik's Cube enthusiast who ran the newsletter '*Cubism for Fun*', was kind enough to provide images of a copy of the issue containing Kloosterman's article [2], which are attached in Appendix B.1. The majority of this method is *inferred* from Kloosterman's article, rather than read as stated facts and rules, and as such may not be true to the original methodology. 'Rubik's Cube in 42 Moves' describes the use of four phases, each with a specific purpose to allow progression into the next phase. The first phase revolves around the concept of 'good' and 'bad' cubies.

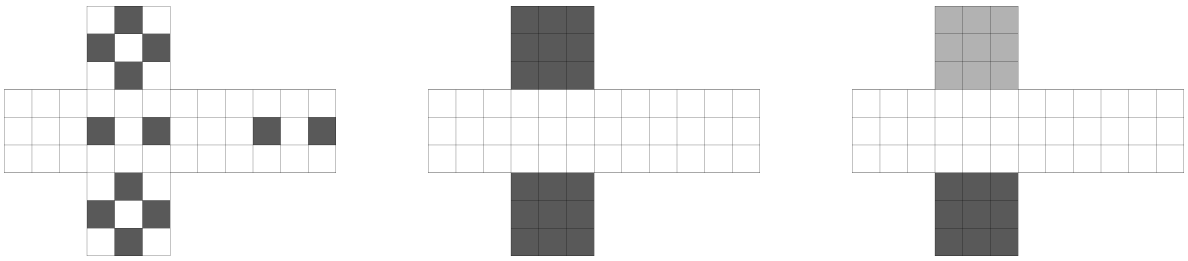
Phase Zero[1]

The concept of goodness is defined by a set of conditions, any of which can be met:

1. A cubie must have a facelet which is on the face which matches its colour
2. A cubie must have a facelet which is on its opposite face
3. A cubie must not have a facelet on the outer edge of either its home or opposite face

The result of these conditions is that if any of the moves from the set {`U U' D D'` } are used in getting the cubie to its correct position and orientation, then the cubie is bad. Kloosterman's first phase makes all of the cubies good, and because *only* the moves {`U U' D D'` } toggle goodness, then the moves used in Phase Zero are {`U D` L^2 R^2 F^2 B^2 }. To quantify the complex concept, an algorithm was written to set each of the fifty-four facelets to a boolean state depending on their home position. To condition the Cube in this manner, the colours were set to either `Color.DARK` (if the facelet needed to be made good) or `Color.NONE` if it was irrelevant to this phase. This monochrome Cube was then used as the target position to generate the first part of the solve sequence.

To create this first part, the code used to generate the tree in section 5.5 was adapted to take a target position and a finite moveset as arguments, then generate a table of all reachable positions and the sequences used to generate them. The monochrome mixed position of the Cube could then be used as the parameter of a lookup query which returns the corresponding sequence. An inversion of the retrieved sequence is then used as the first part of the final solve sequence.



(a) Zero - 'Good' facelets (b) One - `U`-face and `D`-face (c) Two - `U`-face and `D`-face

Figure 5.4: The reduced 'monochrome' Cubes used in Phases Zero - Two

Phase One

The remaining three phases use the same base algorithm to generate the lookup tables and for the search functions. The `U`-face and `D`-face are both coloured `DARK`, and all other faces are coloured `NONE`, as shown in Figure 5.4b. This phase moves all of the `U` and `D` facelets to either the `U`-face or `D`-face (the specific face is immaterial), and correctly orients the corners.

The colour scheme used in Figure 5.4a for Phase Zero shows the locations of the good facelets at the end of Phase Zero/start of Phase One. To avoid corrupting these positions, the moveset {`F F' B B'` } must be omitted from any further sequences. Therefore the moveset for Phase One is {`U D L R` F^2 B^2 }.

[1] The phases are zero-indexed for clarity and ease of use in `for` loops

Phase Two

Phase Two differentiates the U-face and D-face in order to move all of the U-Cubies and D-Cubies into their respective faces. As the cubies are already in either the U-face or D-face already, none of the faces on the side of the Cube should be move by a quaerter turn; ergo, this phase's moveset is {U D L^2 R^2 F^2 B^2 }.

Phase Three

The final phase of Kloosterman's 42-move method is to restore a full-coloured Cube with the same moveset as Phase Three. Kloosterman states that this phase has a maximum depth of eighteen, making it the largest phase by eight search depths. The `monochrome` function for this phase returns the same Cube because no reduction is required.

Optimisations

Kloosterman suggests a few possible improvements to be made, such as avoiding sequences like :L^2 R^2 L^2 : and avoiding mirrored moves. Reflections of sequences are best explained with an example: let sequence A = :U′ R: ; the reflection of A, A' is :U L′ :. Although these sequences are reflections, the results they produce are quite distinct from one another:

```
>>> from cube_class import Cube
>>> cube = Cube()
>>> not_u(cube)
>>> r(cube)
>>> print(cube)
      W W O
      W W G
      W W G
B B B O O Y R R G W R R
O O O G G Y R R G W B B
O O O G G Y R R G W B B
      Y Y B
      Y Y B
      Y Y R
```

Listing 5.2: Performing sequence A

```
>>> from cube_class import Cube
>>> cube = Cube()
>>> u(cube)
>>> not_l(cube)
>>> print(cube)
      R W W
      G W W
      G W W
G O O Y R R B B B O O W
G O O Y G G R R R B B W
G O O Y G G R R R B B W
      B Y Y
      B Y Y
      O Y Y
```

Listing 5.3: Perform A's reflection, A'

6 Results and Discussion

7 Conclusions

Appendices

A Tables

Table A.1: Developments in God's Number

Date	Lower Bound	Upper Bound
July, 1981	18	52
Dec, 1990	18	42
May, 1992	18	39
May, 1992	18	37
Jan, 1995	18	29
Jan, 1995	20	29
Dec, 2005	20	28
Apr, 2006	20	27
May, 2007	20	26
Mar, 2008	20	25
Apr, 2008	20	23
Aug, 2008	20	22
July, 2010	20	20

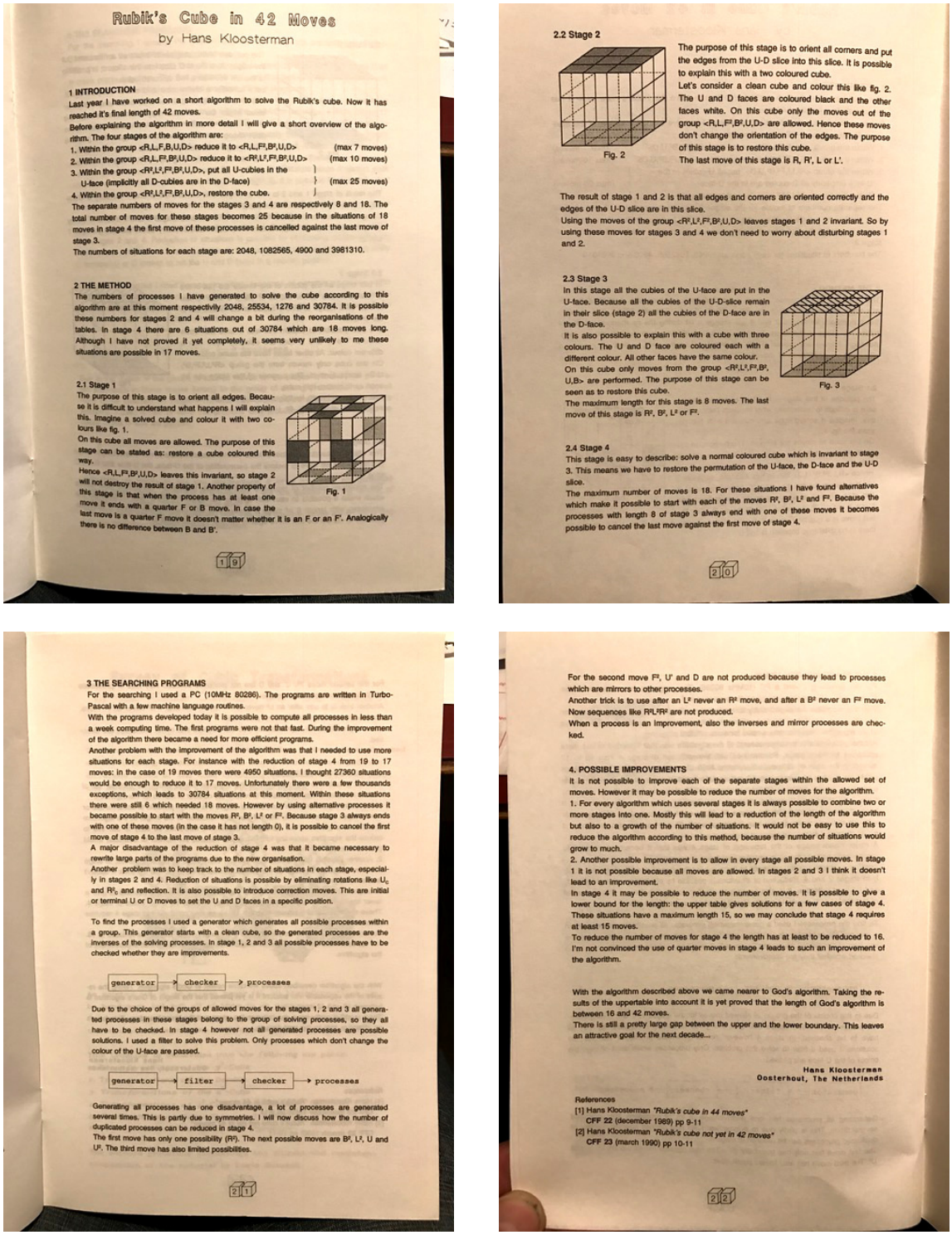
Table A.2: The best and worst case times for solving each depth

Depth	Best Case			Worst Case		
	Mix	Time /s	Pos Num	Mix	Time /s	Pos Num
1	:U':	0.007	0	:B^2:	0.038	17
2	:U'U:	0.048	18	:R^2B^2:	0.467	261
3	:L'D'U':	0.522	262	:$B^2R^2B^2$:	6.545	3501
4	:U'L'D'U':	6.530	3502	:$R^2B^2R^2B^2$:	87.324	46740
5	:D'U'L'D'U':	86.904	46741	:B $R^2B^2R^2B^2$:	1149.732	621648
6	:L'D'U'L'D'U':	1165.350	621649	:R^2B $R^2B^2R^2B^2$:	15053.421	8240086

Table A.3: The predicted times and position counts up to depth-20

Depth	**Best Case**		**Worst Case**	
	Time /s	Pos Num	Time /s	Pos Num
7	1.57×10^4	8.24×10^6	2.03×10^5	1.11×10^8
8	2.13×10^5	1.11×10^8	2.75×10^6	1.50×10^9
9	2.87×10^6	1.50×10^9	3.71×10^7	2.03×10^{10}
10	3.88×10^7	2.03×10^{10}	5.01×10^8	2.74×10^{11}
11	5.23×10^8	2.74×10^{11}	6.76×10^9	3.70×10^{12}
12	7.07×10^9	3.70×10^{12}	9.13×10^{10}	5.00×10^{13}
13	9.55×10^{10}	5.00×10^{13}	1.23×10^{12}	6.75×10^{14}
14	1.29×10^{12}	6.75×10^{14}	1.67×10^{13}	9.12×10^{15}
15	1.74×10^{13}	9.12×10^{15}	2.25×10^{14}	1.23×10^{17}
16	2.35×10^{14}	1.23×10^{17}	3.04×10^{15}	1.66×10^{18}
17	3.18×10^{15}	1.66×10^{18}	4.10×10^{16}	2.25×10^{19}
18	4.29×10^{16}	2.25×10^{19}	5.54×10^{17}	3.03×10^{20}
19	5.79×10^{17}	3.03×10^{20}	7.48×10^{18}	4.09×10^{21}
20	7.82×10^{18}	4.09×10^{21}	1.01×10^{20}	5.53×10^{22}

B Figures



Rubik's Cube in 42 Moves

by Hans Kloosterman

1 INTRODUCTION

Last year I have worked on a short algorithm to solve the Rubik's cube. Now it has reached it's final length of 42 moves.

Before explaining the algorithm in more detail I will give a short overview of the algorithm. The four stages of the algorithm are:

1. Within the group <R,L,F,B,U,D> reduce it to <R,L,F²,B²,U,D> (max 7 moves)
2. Within the group <R,L,F²,B²,U,D> reduce it to <R²,L²,F²,B²,U,D> (max 10 moves)
3. Within the group <R²,L²,F²,B²,U,D>, put all U-cubies in the U-face (implicitly all D-cubies are in the D-face) (max 25 moves)
4. Within the group <R²,L²,F²,B²,U,D>, restore the cube.

The separate numbers of moves for the stages 3 and 4 are respectively 8 and 18. The total number of moves for these stages becomes 25 because in the situations of 18 moves in stage 4 the first move of these processes is cancelled against the last move of stage 3.

The numbers of situations for each stage are: 2048, 1082565, 4900 and 3981310.

2 THE METHOD

The numbers of processes I have generated to solve the cube according to this algorithm are at this moment respectivily 2048, 25534, 1276 and 30784. It is possible these numbers for stages 2 and 4 will change a bit during the reorganisations of the tables. In stage 4 there are 6 situations out of 30784 which are 18 moves long. Although I have not proved it yet completely, it seems very unlikely to me these situations are possible in 17 moves.

2.1 Stage 1

The purpose of this stage is to orient all edges. Because it is difficult to understand what happens I will explain this. Imagine a solved cube and colour it with two colours like fig. 1.

Fig. 1

On this cube all moves are allowed. The purpose of this stage can be stated as: restore a cube coloured this way.

Hence <R,L,F²,B²,U,D> leaves this invariant, so stage 2 will not destroy the result of stage 1. Another property of this stage is that when the process has at least one move it ends with a quarter F or B move. In case the last move is a quarter F move it doesn't matter whether it is an F or an F'. Analogically there is no difference between B and B'.

19

2.2 Stage 2

Fig. 2

The purpose of this stage is to orient all corners and put the edges from the U-D slice into this slice. It is possible to explain this with a two coloured cube.

Let's consider a clean cube and colour this like fig. 2. The U and D faces are coloured black and the other faces white. On this cube only the moves out of the group <R,L,F²,B²,U,D> are allowed. Hence these moves don't change the orientation of the edges. The purpose of this stage is to restore this cube.

The last move of this stage is R, R', L or L'.

The result of stage 1 and 2 is that all edges and corners are oriented correctly and the edges of the U-D slice are in this slice.

Using the moves of the group <R²,L²,F²,B²,U,D> leaves stages 1 and 2 invariant. So by using these moves for stages 3 and 4 we don't need to worry about disturbing stages 1 and 2.

2.3 Stage 3

Fig. 3

In this stage all the cubies of the U-face are put in the U-face. Because all the cubies of the U-D-slice remain in their slice (stage 2) all the cubies of the D-face are in the D-face.

It is also possible to explain this with a cube with three colours. The U and D face are coloured each with a different colour. All other faces have the same colour.

On this cube only moves from the group <R²,L²,F²,B², U,B> are performed. The purpose of this stage can be seen as to restore this cube.

The maximum length for this stage is 8 moves. The last move of this stage is R², B², L² or F².

2.4 Stage 4

This stage is easy to describe: solve a normal coloured cube which is invariant to stage 3. This means we have to restore the permutation of the U-face, the D-face and the U-D slice.

The maximum number of moves is 18. For these situations I have found alternatives which make it possible to start with each of the moves R², B², L² and F². Because the processes with length 8 of stage 3 always end with one of these moves it becomes possible to cancel the last move against the first move of stage 4.

20

3 THE SEARCHING PROGRAMS

For the searching I used a PC (10MHz 80286). The programs are written in Turbo-Pascal with a few machine language routines.

With the programs developed today it is possible to compute all processes in less than a week computing time. The first programs were not that fast. During the improvement of the algorithm there became a need for more efficient programs.

Another problem with the improvement of the algorithm was that I needed to use more situations for each stage. For instance with the reduction of stage 4 from 19 to 17 moves: in the case of 19 moves there were 4950 situations. I thought 27360 situations would be enough to reduce it to 17 moves. Unfortunately there were a few thousands exceptions, which leads to 30784 situations at this moment. Within these situations there were still 6 which needed 18 moves. However by using alternative processes it became possible to start with the moves R², B², L² or F². Because stage 3 always ends with one of these moves (in the case it has not length 0), it is possible to cancel the first move of stage 4 to the last move of stage 3.

A major disadvantage of the reduction of stage 4 was that it became necessary to rewrite large parts of the programs due to the new organisation.

Another problem was to keep track to the number of situations in each stage, especially in stages 2 and 4. Reduction of situations is possible by eliminating rotations like U_2 and R^2_2 and reflection. It is also possible to introduce correction moves. This are initial or terminal U or D moves to set the U and D faces in a specific position.

To find the processes I used a generator which generates all possible processes within a group. This generator starts with a clean cube, so the generated processes are the inverses of the solving processes. In stage 1, 2 and 3 all possible processes have to be checked whether they are improvements.

generator → checker → processes

Due to the choice of the groups of allowed moves for the stages 1, 2 and 3 all generated processes in these stages belong to the group of solving processes, so they all have to be checked. In stage 4 however not all generated processes are possible solutions. I used a filter to solve this problem. Only processes which don't change the colour of the U-face are passed.

generator → filter → checker → processes

Generating all processes has one disadvantage, a lot of processes are generated several times. This is partly due to symmetries. I will now discuss how the number of duplicated processes can be reduced in stage 4.

The first move has only one possibility (R²). The next possible moves are B², L², U and U². The third move has also limited possibilities.

21

For the second move F², U' and D are not produced because they lead to processes which are mirrors to other processes.

Another trick is to use after an L² never an R² move, and after a B² never an F² move. Now sequences like R²L²R² are not produced.

When a process is an improvement, also the inverses and mirror processes are checked.

4. POSSIBLE IMPROVEMENTS

It is not possible to improve each of the separate stages within the allowed set of moves. However it may be possible to reduce the number of moves for the algorithm.

1. For every algorithm which uses several stages it is always possible to combine two or more stages into one. Mostly this will lead to a reduction of the length of the algorithm but also to a growth of the number of situations. It would not be easy to use this to reduce the algorithm according to this method, because the number of situations would grow to much.
2. Another possible improvement is to allow in every stage all possible moves. In stage 1 it is not possible because all moves are allowed. In stages 2 and 3 I think it doesn't lead to an improvement.

In stage 4 it may be possible to reduce the number of moves. It is possible to give a lower bound for the length: the upper table gives solutions for a few cases of stage 4. These situations have a maximum length 15, so we may conclude that stage 4 requires at least 15 moves.

To reduce the number of moves for stage 4 the length has at least to be reduced to 16. I'm not convinced the use of quarter moves in stage 4 leads to such an improvement of the algorithm.

With the algorithm described above we came nearer to God's algorithm. Taking the results of the uppertable into account it is yet proved that the length of God's algorithm is between 16 and 42 moves.

There is still a pretty large gap between the upper and the lower boundary. This leaves an attractive goal for the next decade...

Hans Kloosterman
Oosterhout, The Netherlands

References

[1] Hans Kloosterman *"Rubik's cube in 44 moves"*
CFF 22 (december 1989) pp 9-11
[2] Hans Kloosterman *"Rubik's cube not yet in 42 moves"*
CFF 23 (march 1990) pp 10-11

22

Figure B.1: Hans Kloosterman's article in '*Cubism for Fun*' [2], provided by Rik van Grol

Bibliography

[1] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, "cube20," 2010.

[2] H. Kloosterman, "Rubik's Cube in 42 Moves," *Cubism For Fun*, vol. December, no. 25, pp. 19–22, 1990.

[3] D. Singmaster, *Notes on Rubik's 'Magic Cube'*. Enslow, 1981.

[4] E. Rubik, "The Perplexing Life of Erno Rubik," *Discover Magazine*, p. 81, mar 1986.

[5] O. Waxman, "The 13 Most Influential Toys of All Time," 2014.

[6] Rubik's Cube, "Rubik's UK Website," 2017.

[7] C. Chan, "This Guy Can Solve 3 Different Rubik's Cube While Juggling Them at the Same Time," 2016.

[8] R. T. Cook and S. Bacharach, *Lego and Philosophy: Constructing Reality Brick by Brick*. John Wiley & Sons, 2017.

[9] K. Čapek, *Rossum's Universal Robots*. ebooks@Adelaide, 1921.

[10] Etymonline, "Robot," 2017.

[11] I. Asimov, "Runaround," *Astounding Science-Fiction*, pp. 94–103, 1942.

[12] I. Asimov, "Visit to the World's Fair of 2014," aug 1964.

[13] I. Asimov, *I , Robot*. 1970.

[14] G. Beato, "Turning to Education for Fun," 2015.

[15] B. Franklin, *Poor Richard's Almanack*. 1732.

[16] J. Johnson, "Children, Robotics, and Education," *Artificial Life and Robotics*, vol. 7, no. 1, pp. 16–21, 2003.

[17] K. Becker, "The Edutainment Era - A Look at What Happened and Why."

[18] E. Bilotta, L. Gabriele, R. Servidio, and A. Tavernise, "Edutainment robotics as learning tool," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5940 LNCS, pp. 25–35, 2009.

[19] C. Kelleher, D. Cosgrove, and D. Culyba, "Alice2: programming without syntax errors," *Proceedings of the 15th Annual Symposium on the User Interface Software and Technology*, pp. 3–4, 2002.

[20] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, and L. Sifre, "Article Mastering the game of Go without Human Knowledge," *Nature Publishing Group*, vol. 550, no. 7676, pp. 354–359, 2017.

[21] R. Cellan-Jones, "Google DeepMind: AI becomes more alien," oct 2017.

[22] R. Korf, "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases," *American Association for Artificial Intelligence (AAAI)*, pp. 700—-705, 1997.

[23] M. Reid, "Superflip Requires 20 Face Turns," *Cube Lovers' Mailing List*, 1995.

[24] H. Kloosterman, "Rubik's Cube in 44 Moves," *Cubism For Fun*, vol. December, no. 22, pp. 9–11, 1989.

[25] X. Cai, H. P. Langtangen, and H. Moe, "On the performance of the Python programming language for serial and parallel scientific computations," *Scientific Programming*, vol. 13, no. 1, pp. 31–56, 2005.

[26] T. E. Oliphant, "Python for Scientific Computing," *Marine Chemistry*, pp. 10–20, 2006.

[27] P. Anderson, “unknown,” *New Scientist*, 1969.

[28] JetBrains, “PyCharm: Python IDE for Professional Developers by JetBrains.”

[29] Ev3dev.org, “ev3dev Home.”

[30] Ev3dev and R. Hempel, “ev3dev-lang-python.”

[31] Lego, “EV3 Programming Software,” 2017.

[32] A. Ohnsman, “A new Henry Ford: Elon Musk and his Model S,” jul 2013.

[33] BrickLink, “Stud.io,” 2016.

[34] Lego, “EV3 Large Servo Motor.”

[35] UlfR and Ernest3.14, “How to get good color readings from the color sensor?,” 2015.

[36] Rango Apps, “Camera Color Picker,” 2015.

[37] Lego, “EV3 Rechargeable DC Battery,” 2018.

[38] C. A. R. Hoare, “The Emperor’s Old Clothes,” *Communications of the ACM*, vol. 24, no. 2, 1981.

[39] M. Schoenert, “Cube Lovers: Index by Author,” 1996.

[40] RWTH Aachen University, “Rheinisch-Westfaelische Technische Hochschule.”