University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Solving a Rubik's Cube by Sequentially Restricting the Rotation of Faces

Joseph N. Parker

March 21, 2024

Abstract

With over 450 million sold worldwide, Rubik's Cubes are the most sold puzzle to date. Consisting of 26 small cubes that move freely through rotation of each face, the aim is to align the pieces such that each face of the cube contains only a single colour. This project introduces a method for solving such a puzzle, sequentially restricting which faces a cuber – someone that solves Rubik's Cubes – is allowed to use for the remainder of the solving process. For example, a cuber may be allowed to rotate five of six sides initially, but in stage two of solving, they may only rotate four. Methods of solving exist that utilise subsets of rotations at different stages of the solving process, but these convert between different subgroups, not necessarily decrementing the total number of rotations available.

An implementation of this method is presented for a predefined order of restriction, before a revised and optimised version is introduced that greatly enhances the performance. In addition, a means of visualising the method is implemented, providing explanations of the method, the underlying concept of restrictions, and a 3D-Model of a cube on which the method can be performed.

The visualisation is evaluated through two waves of user-testing, and is found to be fit for purpose. An investigation is made to find an acceptable worst-case number of cubes stored both concurrently and overall, for all possible orders in which rotation can be restricted. Finally, implementations for each possible order of restriction are ranked based on the two worst-case metrics, first by concurrently stored permutations, then the total stored overall.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Joseph Newton Parker Date: 29 November 2023

Contents

1 Introduction

Created by Hungarian inventor Ernő Rubik in 1974, the Rubik's Cube is the world's most popular puzzle [30]. Selling almost half a billion individual cubes to date, not including the hundreds of similar puzzles and imitations that can be found, the Rubik's Cube has solidified its position as a perfect balance between easy to understand and difficult to master. Comprised of a series of smaller cubes with colourful sides, the puzzle is considered solved when each faces is made entirely of one colour. Although this sounds simple, there are no fewer than 43 quintillion ways in which the pieces can be permuted; if we had considered 100 permutations per second, every second since the universe was formed, we might still be looking for a solution [1, 28].

A series of solving methods exist, all aiming to reduce the complexity of the challenge. Typically, solving methods focus on visual pattern matching and 'block-building', where the cube is solved section by section through a series of memorised sequences of rotations. Most early methods were devised to optimise solving by hand, but these do not result in the shortest sequences that solve each permutation. As computer-aided proof became increasingly accepted in mathematical publications, methods infeasible to execute by hand but yielding more efficient solutions were devised. Computer-focused methods tend to focus on solving the puzzle through various inherent properties, rather than through visual patterns. Through their implementation, shorter sequences to solve permutations can be found than is possible manually.

This dissertation introduces a new method of solving, wherein rotation of the faces of the cube are sequentially restricted, one at each stage, such that their use is no longer permitted for the remainder of the solving process. Before a rotation is removed from the available set, the cube is rotated through those still in use until the rotation being removed is no longer required for completion of the solve.

Utilising the benefits that arise from a reduced rate of expansion in a traversal problem, an implementation is presented for this method. A series of optimisations are also introduced, the amalgamation of which resulting in a near-instant execution. Alongside this implementation is a visualisation of the method, presenting a series of buttons through which a user can interact with a 3D-Model of a Rubik's Cube. Both implementation and visualisation are evaluated, first for correctness and fitness for purpose, then for efficiency. While it is a challenge in itself to produce a working implementation, this dissertation aims to highlight the benefits of problem reductions and symmetry breaking to optimise traversal of a search space. This is done through comparison of upper-bounds with regard to the number of permutations that must be stored both concurrently and across an entire execution. These worst-case upper-bounds are found and compared for optimised and unoptimised implementations of the method, ultimately reaching a conclusion as to which order the rotation of faces should be restricted when aiming to minimise the number of permutations that must be processed.

Finally, suggestions for future work are made, proposing means through which the visualisation could be extended to better communicate the concept of restriction of faces, and ways in which the evaluation of the methods itself could be improved.

2 Terminology

This chapter introduces the terminology used throughout the dissertation to refer to each individual cube, its components, the rotations that can be performed on each cube, and the sequences formed by concatenation of various rotations.

2.1 Cube Structure

2.1.1 Components of a Rubik's Cube

A cube made from stacking smaller cubes such that there are three in each row, column, and aisle consists of 27 total pieces. A standard 3×3 Rubik's Cube has one less; a puzzle would be much more difficult if it required the solving of an impossible-to-see central piece.

The pieces are labelled in a straightforward way, as corners, edges, and centres. Each piece consists of several coloured sides, with each corner having three distinct colours, each edge having two, and each centre having one.

Regardless of how a cube is turned, each piece remains its own type. A corner will always be a corner, an edge an edge, and the centre pieces, centres. These are highlighted in Figure 2.1.

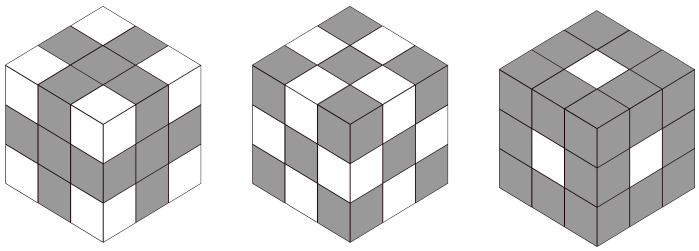


Figure 2.1: *Cubes with corners (left), edges (middle), and centres (right), highlighted in white.*

2.1.2 Referencing Components

The entire cube is referred to as a cube and each smaller cube as a cubie, as is common practice in related literature [3]. Additionally, each individual coloured segment of a cubie is referred to as a tile. We can refer to each face of a cube via an assigned letter. Common practice is to refer to faces in the following way:

- *F* to represent the face on the front of the cube, i.e. the 'Front' face;
- *B* for the 'Back' face;
- *U* for the 'Up' face;
- *D* for the 'Down' face;
- *L* for the 'Left' face;
- *R* for the 'Right' face.

An individual piece can be referenced through the sides to which it resides when a cube is solved. For example, the *UF* edge is the edge present between the *U* and *F* faces when the cube is solved, regardless of where it is in a given permutation.

2.1.3 Representing Permutations

Modelling. To avoid confusion, the same angle of observation is maintained across all diagrams. The angle in question will centre around the *UFR* slot, see Figure 2.2 (right).

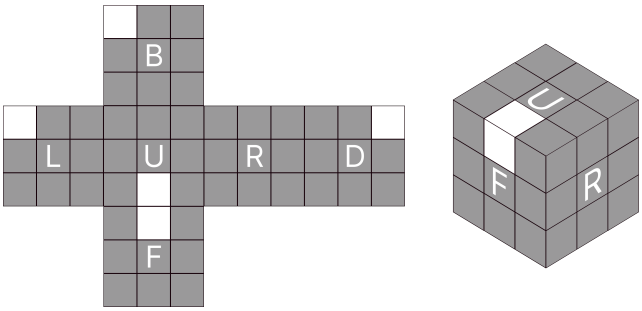


Figure 2.2: *A cube viewed as a net and from the UFR corner, with the UF and DBL slots highlighted.*

While three-dimensional representations provide insight into what a cube looks like, information regarding the reverse sides are lost. To regain this missing information, a second method of representing a cube will be used where this information is desired.

This second representation is a net, see Figure 2.2 (left). This splits individual cubies into different sections, but is required when transposing a three dimensional object to a plane with only two.

Colouring. The standard 'Minus Yellow' colouring will be applied to the cube [8]. For exampl, in Figure 2.3 each each pair of opposing faces can be converted to the other by addition or removal of a yellow tone. To ensure only a single possible colouring, a clockwise transition between yellow, orange, and green is also required.

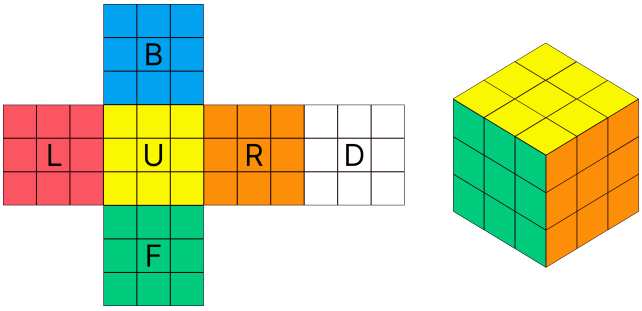


Figure 2.3: *Coloured representation of a solved cube.*

When a face of a cube is rotated, the corners and edges move between the faces, but the centres stay still. From this, it is always possible to tell which face will be a certain colour when a solve is finished by looking at the centre piece. A side with a blue centre will be blue when the cube is solved [22].

2.2 Rotating the Cube

We refer to each rotation of a cube using *Singmaster* notation, which represents the clockwise 1/4 rotation of a face by the letter used to represent it [25]. For example, an R rotation represents rotation of the R face. This result of such a rotation is demonstrated in Figure 2.4. We can also use sequences of rotations to represent the rotating of a cube multiple times; denoted by tuples, sequences are performed by conducting rotations in the order they are written. For example, the sequence (R, U) represents rotation of the R face, then the U face.

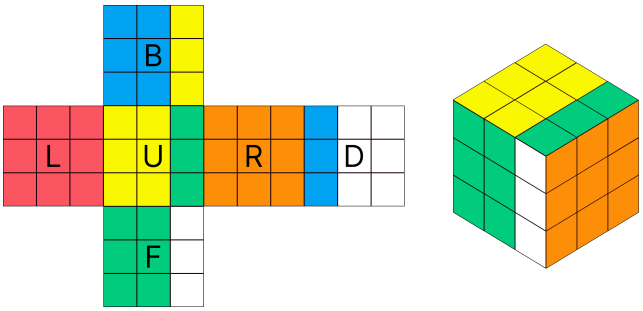


Figure 2.4: *A cube rotated by R.*

While this would be sufficient for representing any sequence that can be applied to the cube, we introduce common notation extensions to represent other magnitudes of rotation directly. A half turn of the R face could be represented by the sequence (R, R), or a three-quarter turn by (R, R, R), however we will instead refer to a half turn of the right face by $R2$, pronounced 'R-Two', and a three quarter turn as R', pronounced 'R-Prime'. R' can alternatively be thought of as an anticlockwise 1/4 turn.

With this additional notation we are able to represent certain sequences in reduced formats. For example, (R, R, R, U, U) can be rewritten as the shorter $(R', U2)$.

Note that we choose to represent a null rotation as N, though this will never appear in a sequence. Any sequence involving a rotation by N can be simplified to one without by removing the character entirely, i.e., $(R', N, U2)$ may be simplified to $(R', U2)$, or the null sequence of (N) to $()$. While this rotation is is equivalent to performing no rotation at all, it is required for mathematical correctness in following Chapters. An example is shown in Figure 2.5.

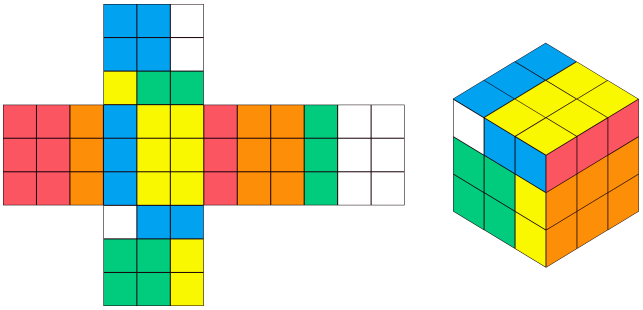


Figure 2.5: *A cube rotated by* $(R', N, U2)$*, simplified to* $(R', U2)$*.*

3 Background

This chapter introduces the mathematics used throughout the dissertation, various relevant properties of a cube, some methods of solving, and bitboards – the key to efficient encoding of a cube. All mathematical concepts are as described in 'A Concise Introduction to Pure Mathematics' by Martin Liebeck [17].

3.1 Relevant Mathematics

3.1.1 Groups

We define a group, G, as a set of objects and an associated binary operator, $*$, such that when the operator is applied to elements of the set, four conditions are satisfied. These conditions are:

- (Closure) $h * g \in G$ for all $g, h \in G$;
- (Associativity) $(f * g) * h = f * (g * h)$ for all $f, g, h \in G$;
- (Identity) there exists $e \in G$ such that $e * g = g * e = g$ for all $g \in G$;
- (Inverse) for any $g \in G$ there exists g^{-1} such that $g * g^{-1} = e$.

A common example of a group is the set of all integers under the binary operator addition. Naturally these are closed as the sum of any two integers will be a third integer. They are associative as integers can be added in any order to the same total. The identity is zero, and the inverse of any element is its negation, i.e., its sign-flipped counterpart.

3.1.2 Subgroups

Given a group G, we define a subgroup of G, $H \subseteq G$, as a subset of the elements of G still satisfying the group axioms, i.e., the four defined conditions above.

An example subgroup of the group of integers under addition is the group of even integers. The identity and inverses are maintained from the original group, and as the addition of any two even integers remains even, i.e., the subgroup is closed. Associativity is a fundamental property of the group structure itself, and any subset inheriting the group operation will preserve this property.

3.1.3 Equivalence Classes

An equivalence relation is a relationship on a set, A, often donated by '$\sim$', under which three fundamental properties hold:

- (Reflexivity) $a \sim a$ for all $a \in A$;
- (Symmetry) if $a \sim b$, then $b \sim a$ for all $a, b \in A$;
- (Transitivity) if $a \sim b$ and $b \sim c$, then $a \sim c$ for all $a, b, c \in A$.

An equivalence class of set A is a subset of A all equivalent under some equivalence relation. As a result of transitivity, A is partitioned into a collection of disjoint sets, such that each element of the original set is in exactly one equivalence class.

Take the set of all integers, $\mathbb{N}$, and the relation where two integers relate if neither or both are zero as an example. This relation is:

- reflexive as two elements relate trivially if they are equal,
- symmetric, as the ordering of elements is inconsequential, and
- transitive, as if $a \sim b$ and $b \sim c$, then either all three are zero or none are.

Since all three properties hold, this is indeed an equivalence relation. The set of integers $\mathbb{N}$ is partitioned under this relation into two distinct classes: all integers except zero, and zero alone.

3.2 Groups of Sequences

This section introduces a means of representing the set of cubes as a group, and details some relevant properties of the resulting group.

3.2.1 Sequences as a Group

We are able to represent the sequences that can be applied to a cube as elements of a group, G, under the operation of concatenation of two sequences [5].

This group is trivially closed, as any physical turn of a cube will result in another possible permutation. It is associative, as we can concatenate sequences in any order of operation; (R, U), (R), and (U) could be concatenated correctly in either order, i.e. via $((R, U) * (R)) * (U)$, or $(R, U) * ((R) * (U))$. Inverting a rotation is as simple as conducting a 1/4 turn a further three times on the same face, or as would be done in practice, turning it back again.

While any example of sequence inversion concatenated to what is being inverted satisfies the identity axiom of a group, we will use the null sequence, $()$, as the identity in its simplest form.

3.2.2 Minimal Generating Set

A *generating set* of a group is a subset of the group elements such that any element of the group may be expressed as a combination under the group operation of finitely many elements of the subset and their inverses. A *minimal generating set* is a generating set of the minimum possible size. If S is a subset of G, then $\langle S \rangle$ is the subgroup generated by the set S. If S is a generating set of G, then $\langle S \rangle = G$.

If a permutation is rotated by the sequence $(R, L, F2, B2, R', L', U, R, L, F2, B2, R', L')$, the outcome is equivalent to a single rotation of the D face. In other words, we are able to simulate a D rotation without its own use, and therefore it is not required for a minimal generating set. As these rotations can be directly mapped to other frames of reference, equivalent sequences exist to simulate rotation of any face indirectly. Hence, no more than five generators are required to generate all possible sequences [13].

At least one of each pair of opposite sides must be present in the generating set, as otherwise edges cannot be 'flipped' in a slot, as shown in Figure 3.1. At least one of each adjacent face must also be present, as otherwise their shared edge cannot be interacted with. A 1/4 rotation can simulate all other magnitudes of rotation of a face. With these requirements, all permutations can be transformed to each other, hence the minimal generating set of sequences to transform a solved permutation to any other contains a 1/4 rotation of exactly five of the six faces [11].

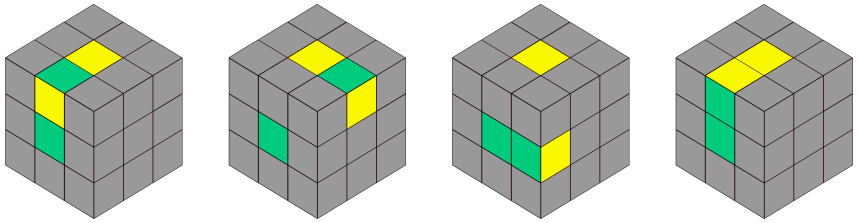


Figure 3.1: *Flipping an edge in place with the sequence* (U', R', F').

3.2.3 Subgroups of Sequences

It is appropriate to refer to a subset of sequences generated by a reduced generating set as a subgroup of the entire group of sequences, G. We define such subgroups with subscript containing the maintained generating elements; the subgroup G_U is the group generated by the set $\{U\}$ under the same concatenation operation.

For ease of representation, we assume the permutation resulting from the null sequence is a solved cube. From a different initial permutation, we would maintain sets and subsets of the same orders, though their relationship may be harder to interpret. We demonstrated the subgroup of G, G_U, in Figure 3.2.

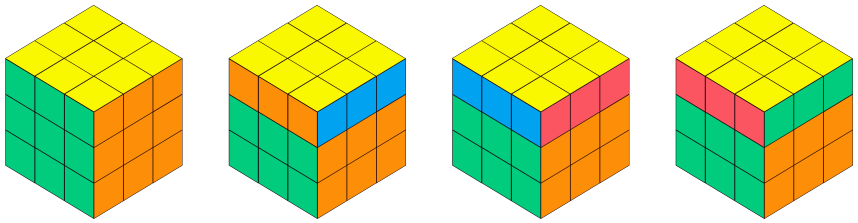


Figure 3.2: *All members of* $G_U = \{U\}$*, a subgroup of* G.

3.3 Compositionally Equivalent Cubes

It can be useful to associate permutations with each other, even if they are distinct. While two unique permutations should not be considered equal, they can be partitioned into equivalences classes under the relation of certain visual or intrinsic properties.

We consider cubes with similar compositional elements to be *compositionally equivalent* under certain defined equivalence relations. For example, we may consider the presence of a specific piece in a specific slot in two different permutations of the cube to indicate equivalence.

Consider the equivalence relation of the *UF* bar, i.e. the permutation and orientation of the pieces in the *UFL*, *UF*, and *UFR* slots. We can partition the set of all possible permutations into equivalence classes of cubes with matching *UF* bars; four members of the equivalence class with a solved *UF* bar are shown in Figure 3.3. For this particular equivalence relation, all classes will have the same cardinality, i.e., the same number of elements.

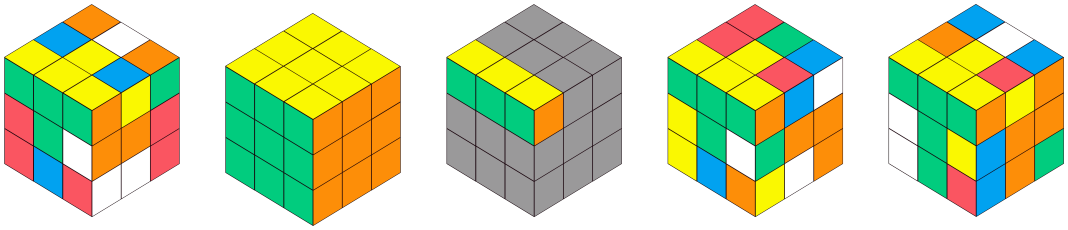


Figure 3.3: *Four cubes equivalent with respect to the UF bar, and their relevant commonality (centre).*

This principle leads to significant computational benefits when transitioning between permutations, explored later during the optimisation of the solving algorithm.

3.4 Other Relevant Cube Properties

We observe that we are able to rotate a corner through its three possible orientations with rotations of only two sides, provided these sides share an edge and that the corner is located on one of the faces. Figure 3.4 demonstrates that the sequence (R, U) can be used to repeatedly rotate the UFR corner one step clockwise. As in Figure 3.1, to correctly orient an edge three sides must be used, and all must be adjacent with each other.

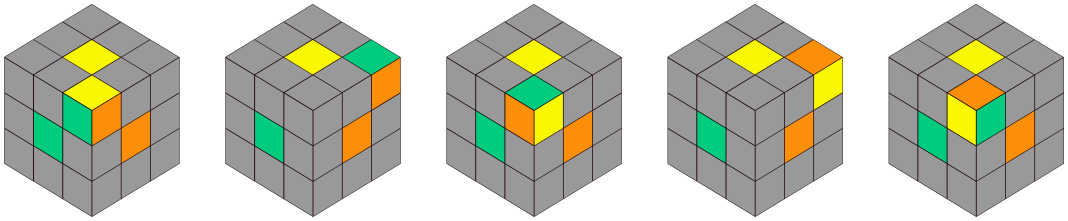


Figure 3.4: *Each orientation of a corner found by the sequence* (R, U, R, U).

The differing number of sides required for orientation of corners and edges comes as a result of their positions on a face. Each pair of adjacent faces shares two corners and only one edge, meaning we are able to rotate a corner in a cycle over two alternating faces, as the corner is always present on both. In the case of an edge, however, only one slot is shared between two adjacent faces. If we rotate a cube by one of these faces, the other can no longer interact with the cubie that was shared, as it is no longer in the mutual slot. To return it to this position, the first rotation must be inverted, and as a result will have had no effect.

We define an edge solvable with rotation of only two of the three pairs of opposing faces as a *good edge*, and one that requires three as a *bad edge* [4]. This terminology is most commonly used when discussing more advanced human solving methods, which will be explored later.

3.5 Solving A Cube

Here we introduce existing methods used to solve a cube. This includes both human-oriented methods as well as existing computational methods.

3.5.1 The Devil's Algorithm

As there are only a finite number of possible permutations, a sequence of rotations that when repeated cycles through each permutation would be sufficient for solving any cube, from any

starting position, given enough time [7]. A sequence of this kind is referred to as a Devil's Algorithm, and the minimum length of such a sequence as the Devil's Number [6].

Unfortunately, cubers hoping for an easy solution will have to wait, as no such sequence has been found so far. The lowest proposed length comes as a product of Hamiltonian Cycles over the permutations of a cube, and is thought to be no fewer than 34,326,986,725,785,601 rotations [2]. At worst, this must be repeated no fewer than 1,260 times to complete a solve.

3.5.2 The Beginner Method

A slightly faster method used when first learning to solve a cube is the beginner method is depicted in Figure 3.5 [26]. It involves visual pattern matching, with six distinct sequences to be memorised. The cuber must solve the edges of the D face, then the D corners, then the rest of the edges not present on the U face. Next, they orient the remaining pieces over two steps, then permute the corners, and finally, permute the remaining edges.

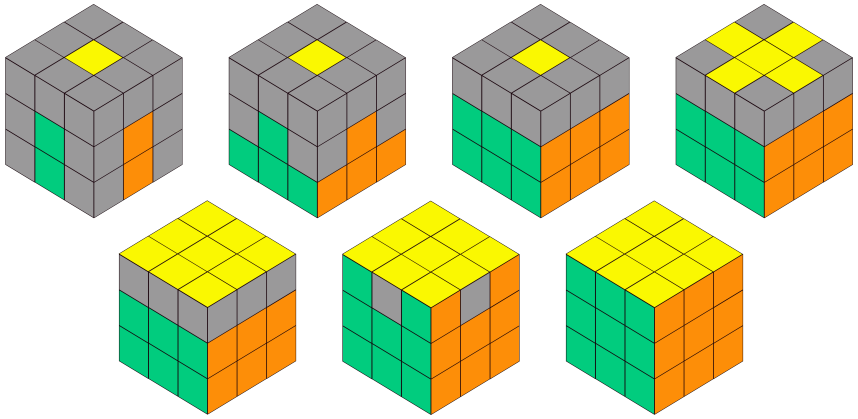


Figure 3.5: *Step-by-step representation of the official Beginner Method.*

3.5.3 The ZZ Method

The ZZ Method, Figure 3.6, makes use of the ergonomic benefits of using a reduced number of sides; it is easier to rotate the L and R faces than the F and B faces [12].

To do this, cubers first orient the edges such that all are *good edges*, then solve the DF and DB slots. After these steps, the first two layers can be solved only with rotations of the U, L, and R faces, equivalent to solving via the subgroup $G_{U,L,R}$. From this point onward, however, other rotations are reintroduced to solve the rest of the cube efficiently.

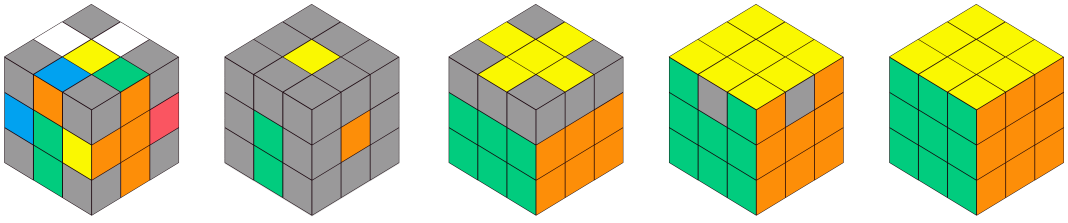


Figure 3.6: *Step-by-step representation of the ZZ-Method.*

The initial steps have the added benefit of guaranteeing correctly oriented U edges, requiring a set of only 28 sequences to permute and orient the corners simultaneously, and a further four to

permute the edges [32, 33]. However, as the recognition of which sequence to use at a given time is often much harder for these new sets of sequences, the method is typically slower in practice.

Some top cubers are able to combine the final two stages into one via the set of 'Zborowski-Bruchem Last Layer' sequences, however, this requires the memorisation and efficient recognition for 492 separate sequences, infeasible for the average solver [34].

3.5.4 Two-Phase Solving

Two Phase Solving is an algorithmic solving method that uses an iterative process to solve optimisation problems over two distinct phases [19]. The first phase attempts to solve a simplified version of the problem, and the second iteratively applies information learned in the previous phase to solve the original. In relation to solving a cube, this may involve solving all corners, all slots on a certain face, or any other sub-problem.

From here, an implemented algorithm will iteratively generate sequences to solve the results of increasingly less optimal phase one solutions, in search of a combined total sequence length less than that of any solution previously found. This process repeats until a phase two algorithm of size zero is found, at which point the phase one sequence is taken as the overall solution.

This method can be infeasible if the problem is initially oversimplified for phase one, but if an appropriate heuristic is used then this is not a problem.

3.5.5 The Kociemba Method

The most widely recognised application of a two-phase solving method is the Kociemba Method [23]. The first phase aims to transition a cube to one that can be solved with only the subset of rotations $\{U, D, F2, B2, L2, R2\}$ [15]. One such cube can be seen as the first stage of Figure 3.7.

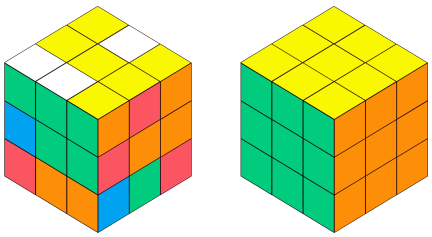


Figure 3.7: *Step-by-step representation of the Kociemba Method.*

As the iterative deepening of a two-phase algorithm can sometimes be extensive, the Kociemba Method is often implemented through the amalgamation of a number of optimisation techniques. These include search restrictions of unlikely paths, parallelisation of branch searches, and use of symmetrical search pruning, as symmetrical states occur more frequently when the above set is chosen as the heuristic for phase one [20]. This increase in occurrence stems from the prohibiting of most 1/4 rotations, as this forces correct orientation of all edges and corners, and the insertion of edges not solved onto the U and D faces into the middle layer.

3.5.6 The Thistlethwaite Method

This method follows a four step process, transitioning the cube between a series of nested subgroups of the group G. It does this by first restricting the U and D faces to 1/2 turns, $U2$ and $D2$ only, then does the same for the F and B faces, then L and R, before solving the rest of the cube with only half turns [29].

It was first implemented as a series of lookup tables for each stage, though these did not encompass all permutations as there are too many to feasibly store and lookup from. As a result, a series of preliminary moves would be made before the lookup table was considered, and as such the optimal sequence was not found between each stage. The algorithm has since been optimised to produce a sequence with fewer rotations in the worst case, and even fewer if slight changes are made to stages three and four, though this still does not guarantee optimal transitioning between stages.

All four stages of the original method are shown in Figure 3.8.

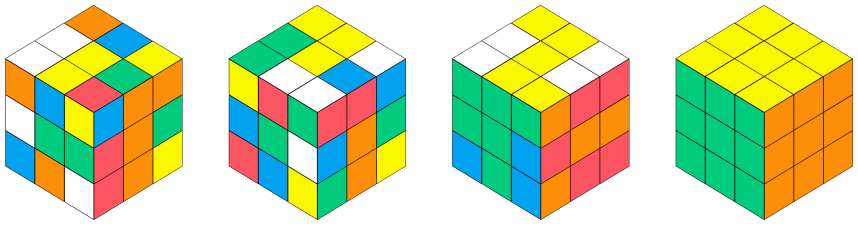


Figure 3.8: *Step-by-step representation of the Thistlethwaite Method.*

The transitions between subgroups are successful in reducing the number of rotations at the disposal of the algorithm, however, while this method does utilise the benefits of solving a cube via a series of nested subgroups, it does not make explicit use of any computational benefits that may arise from a reduced minimal generating set. Even during the final stage, five of the six faces may still be required to complete a solve.

3.6 Bitboards

Most commonly used for the encoding of chessboards, from which they get their name, bitboards are a powerful tool for representation data efficiently [24]. These make use of the storage of integers as binary values, inferring problem-specific information from the bits themselves.

Common practice for the representation of a chessboard is to use sixteen separate 64-bit integers as bitboards, with each bitboard representing the placements of one type of piece, and each bit representing a single square of the board [14]. If a bit of a given bitboard is a '1', then there is a piece of that type present in the associated square. While the efficient use of memory is a benefit of bitboards, the true advantage come through *bitwise operations*.

Checking for an open file, i.e., a column with no pawns, can be done by *masking* the bitboards for the two sets of pawns with the *and* operator, such that bits are set to zero if not in the desired file. The two masked bitboards will then equate to zero if there are no pawns in the file. This avoids the explicit checking of each position, which can greatly improve performance.

While best-known for chess-modelling applications, bitboards can be used for other purposes; the representation of sparse matrices in graphing algorithms, the game of connect-four, and as shown in the following chapters, for the representation of a permutation of a cube [18, 31].

4 The Solving Method

This chapter introduces the solving method itself, along with an example. The proposed method is to solve a cube by sequentially restricting the rotation of each face, at each stage reducing the size of the generating set for sequences that can still be applied until the cube is solved. The method begins with the subgroup of G with the minimal generating set $S_5 = \{(U), (R), (L), (D), (F)\}$, referred to as G_5. Note that $G_5 = G$. From here, the method transitions between nested subgroups in the following order:

- $G_5 = \langle S_5 \rangle$, where $S_5 = \{(U), (R), (L), (D), (F)\}$
- $G_4 = \langle S_4 \rangle$, where $S_4 = \{(U), (R), (L), (D)\}$
- $G_3 = \langle S_3 \rangle$, where $S_3 = \{(U), (R), (L)\}$
- $G_2 = \langle S_2 \rangle$, where $S_2 = \{(U), (R)\}$
- $G_1 = \langle S_1 \rangle$, where $S_1 = \{(U)\}$
- $G_0 = \langle S_0 \rangle$, where $S_0 = \{()\}$

At which point the sequence to generate the current permutation of the cube must be contained within G_0. As this subgroup contains only the identity, i.e., (), the current permutation must be solved by the application of the null sequence. The cube is solved!

4.1 An Example Solve

Starting from a randomly selected initial permutation, Figure 4.1, found by application of the sequence $(L2, F, U', L2, D, U2, B', R2, D', L', U, R2, D2, L, U', D, R', U)$ to a solved cube, we solve the cube through the proposed method.

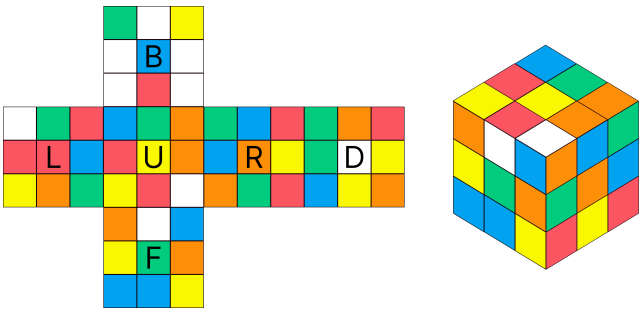


Figure 4.1: *A member of* $G_5 = \langle \{(U), (R), (L), (D), (F)\} \rangle$.

Solving for G_4. The first step is to transform the permutation into one that can be solved without the use of F rotations. This is done by orienting each edge, such that all are *good edges*. The shortest transition from G_5 to G_4 using only rotations in S_5 is $(D2, F', R', U', L', F')$, and the resulting cube is shown in Figure 4.2.

Note that the process for reducing to G_4 is equivalent to the first stage of solving with the ZZ Method, where the edges are oriented for ergonomic solving of the first two layers. In both cases, all edges are oriented before any further solving is done.

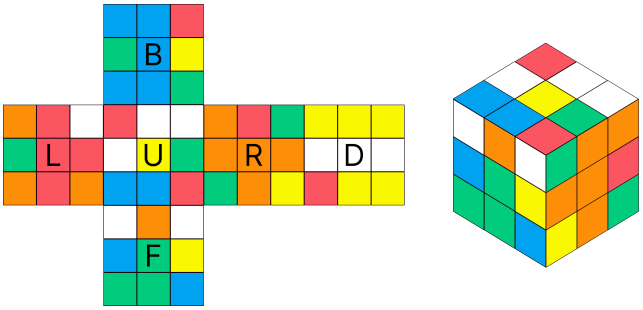


Figure 4.2: *A member of* $G_4 = \langle\{(U), (R), (L), (D)\}\rangle$.

Solving for G_3. To further reduce the order of the minimal generating set required to generate a sequence to solve the current permutation, we must correctly position the edges that cannot be interacted with by any sequence in S_3.

The edges in question are the *DF* and *DB* edges — the same that must be solved in stage two of the ZZ Method. The shortest way to place these in their correct slots is the sequence $(U, R2, D)$.

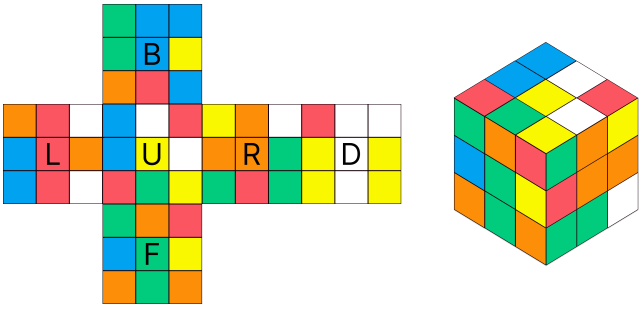


Figure 4.3: *A member of* $G_3 = \langle\{(U), (R), (L)\}\rangle$.

These edges can be seen in the net of Figure 4.3 as a white line across the *D* face, and two adjacent tiles on the *F* and *B* faces.

Solving for G_2. Conversion to a permutation solved by G_2 is the most involved step of the process. Similar to solving for G_3, the process involves building a $1 \times 2 \times 3$ block centred on the bottom left of the cube. This can be seen in Figure 4.4 as a red section on the the *L* face, and an extension to the white section on the *D* face. The pieces inserted in the previous stage to the *DF* and *DB* slots have maintained their position; these cannot be interacted with now the available set of rotations has been reduced to no longer include the *D*, *F*, and *B* faces.

While constructing the left-block, we must ensure the remaining corners are appropriately permuted. It is easiest to check their permutation by inserting the *DFR* and *DBR* edges into their slots without orienting them. If the remaining corners on the *U* face are in the correct cyclic order, i.e. clockwise we have $UFR \rightarrow UFL \rightarrow UBL \rightarrow UBR$, then they are correctly permuted. This could be achieved instead after building the block by application of one of a series of specific sequences in G_3, but this would then increase the total number of rotations unnecessarily.

To transition from G_3 to G_2, we apply the sequence $(L, U, L', U, L, U, R2, U, L', U', L2)$, and reach the permutation as in Figure 4.4. The correctness of the remaining corners can be validated by inserting the DFR and DBR edges with (U', R'), and observing that the corners on the *U* face follow the expected cycle.

Solving for G_1. Subgroup G_1, has four elements, the permutations solved by (U), $(U2)$, (U'), and $()$, as shown in Figure 3.2. Hence, when completing this stage there is a 25% chance to completely solve a permutation. A permutation in G_1 is at most one rotation from solved.

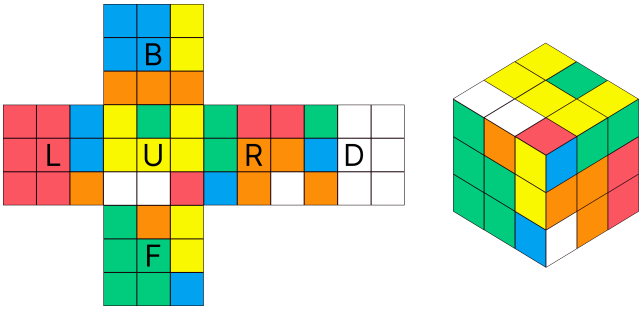


Figure 4.4: *A member of* $G_2 = \langle\{(U), (R)\}\rangle$.

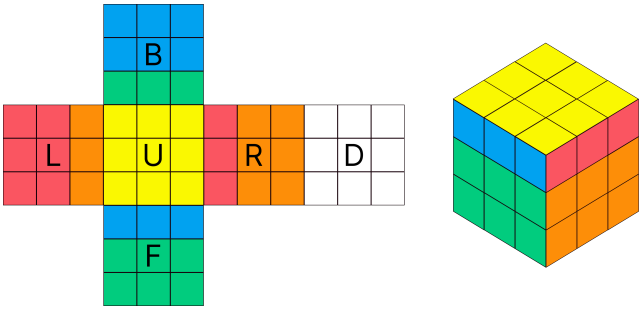


Figure 4.5: *A member of* $G_1 = \langle\{(U)\}\rangle$.

To reach the permutation of Figure 4.5, we apply the optimal sequence $(U', R, U', R', U2, R, U2, R', U', R')$.

Solving for G_0. The final stage requires only one rotation of the U face, and is trivial to find; in our case, the rotation required is $U2$. As in Figure 2.3, the cube is now solved!

A Note On Reduction Order. The order in which the generating set is minimised could be changed; there are six possible faces, therefore 6! or 720 different orders in which rotations could be restricted. The select order is used as it is the closest replication of the common practices of human solving, starting with the D face, and finishing with U. B is removed before F as this makes rotations easier to track visually, and L before R because R rotations are easier for a right-handed solver to perform.

Though not a factor in the initial selection of an order, different orders of restriction produce subgroups of different sizes [16]. For example, $S_{UD} = \{(U), (D)\}$ has $4^2 = 16$ elements, as rotations of opposite faces commute, whereas $S_{UF} = \{(U), (F)\}$ has 73,483,200 elements. As a result, alternative orders may result in varying levels of difficulty when transitioning between permutations. Comparisons between these orders are made in the evaluation, once a means of doing so has been established.

A Note On Subgroup Naming. Out of context, the terminology of numbers in the definition for each paired group and generating set provides little insight into their elements. However, it is more concise to represent each traversed subgroup and set by the number of unrestricted faces; the order of restriction is guaranteed by the method specification, and each set has a unique number of rotations available. The equivalent generating sets are listed below:

- $S_5 = S_{URLDF}$
- $S_4 = S_{URLD}$
- $S_3 = S_{URL}$
- $S_2 = S_{UR}$
- $S_1 = S_U$
- $S_0 = S_N$

5 The Solving Algorithm

This chapter details the implementation of the solving method from Chapter 4; the encoding of each permutation, the storing of cubes, and an explanation of how the search space is traversed to find each optimal sequence. A selection of more impactful optimisations are also explored before a final, improved algorithm is presented.

5.1 Implementing A Cube

This section explains the storage and rotation of each individual cube.

5.1.1 Representing A Permutation

We store the permutation of a cube in a *uint* array of length six, with an index for each face. Each individual face is represented as a bitboard.

The bitboard for each face comprises of 32 individual bits, and as we are able to infer the centre tile from the index of the array, we can assign four bits to each of the remaining eight tiles. Each tile is assigned a value to represent the face to which they reside when the cube is solved, i.e., its colour. The values are assigned in the following way:

- U is represented by `0b1100`, or `0xC`,
- D is represented by `0b1010`, or `0xA`,
- F is represented by `0b1001`, or `0x9`,
- B is represented by `0b0110`, or `0x6`,
- R is represented by `0b0101`, or `0x5`,
- L is represented by `0b0011`, or `0x3`.

The values were chosen in such a way that the '1's of one tile are not encapsulated by the '1's of any other time. For example, `0b1011` could not have been chosen to represent a value as the D, F, and B faces are composed entirely of bits present in this value. This facilitates the use of bitwise comparisons, as there is no overlap in pattern matching for each tile. Additionally, the numerical ordering of the representations permit the utilisation of the greater than or equal to ($\geq$) operator for identifying *good* and *bad* edges through their tiles. We can check if a tile is greater than or equal to `0xA` to identify U and D tiles, and the equivalent for `0x6` for F and B tiles, provided the U and D check was false.

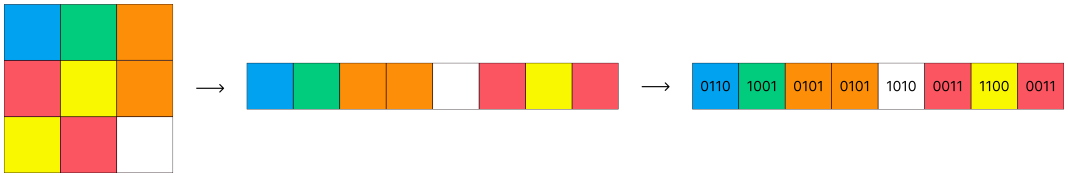


Figure 5.1: *The conversion of a face to the bitboard* `0b01101001010101011010001111000011`.

An example of the conversion from a face to a bitboard can be seen in Figure 5.1. In each index of the bitboard array, i.e., each face, the tiles are stored as a clockwise rotation, starting from the top left corner. The *U* face is viewed with the *UF* edge at the bottom, *D* as though the cube was rotated through the *R* direction twice, and the rest with the edges shared with the *U* face at the top. This decision was made to optimise *U* and *R* rotations, as these are likely to occur most often due to their presence across the most stages of the proposed method as in Section 4. This improves the efficiency of the overall solution, as it reduces the number of bitboards that must be preprocessed before rotation on average.

5.1.2 The Cube Class

The representation of each permutation is packaged into the *Cube* class, along with two additional parameters. The first parameter, *lastMove*, represents the rotation made most recently. The second, *sequence*, represents a sequence relevant to the cube. The parameter *lastMove* is used for search optimisations, and *sequence* in order to preserve which rotations were made to transition from another permutation to the current one. We consider two *Cubes* to be equal if they are of the same permutation, regardless of their other components.

Cubes can be created through a collection of two public constructors and five methods returning a *Cube*, all producing *Cubes* in slightly different ways. The constructors allows for a *Cube* to be created with the solved cube, or matching a bitboard given as a parameter, and the other methods allows for an existing *Cube* to be copied with or without *sequence* and *lastMove*, with *lastMove* only, or as a result of rotation, with or without the *lastMove* and *sequence* resulting from the rotation. While this logic could be simplified it allow for less methods to be implemented, allowing flexibility in the instantiation of a cube reduces the number of operations. As hundreds of thousands of *Cube* objects are created during each solve, the pros of additional methods outweigh the cons.

5.1.3 Rotating A Cube

All methods that perform a rotation do so via the *rotatedBitboards* method, which takes a move as a parameter and returns the bitboards resulting from the operation. Each rotation is represented by an integer, with moves of the same face being equivalent under modulo six. For example, $U = 1$, $U2 = 7$, and $U' = 13$ are equivalent.

To perform the rotation of a face, two steps are taken. The first rotates the bits of the face being turned, and is implemented through the helper method *RotateBitboard()*, which requires both a bitboard and a 'shift' as parameters. To rotate a 1/4 turn, for example a *U* turn, the shift value is eight. This shifts a piece forward two tiles, i.e., corners shift to the next corner, and edges to the next edge. The shift for a 1/2 turn is 16, effectively skipping a slot, and 24 for a 3/4 turn. The method makes use of the bitwise 'shift' operators, << and >>, for left and right shifts, as well as the 'or' operator, |. To rotate a bitboard b of size n forward by s bits, we use the equation:

$$b_{shifted} = (b >> s) \mid (b << (n - s)) \tag{5.1}$$

For the second half of the operation, the tiles associated but not directly on the rotating face must be permuted in a cycle. This is the *band* of a face. To do this, a bitmask is created such that the bits remaining unchanged can be found via the bitwise conjunction operator (&), and the bits rotating towards a face by the bitwise inversion of the mask using the negation operator (¬). This process is completed in one step, with mask m, by the following equation:

$$b_{combined} = (m \ \& \ b_{maintained}) \mid (\neg m \ \& \ b_{new}) \tag{5.2}$$

The masking process is simple for the *U* and *D* faces, masking the the bits remaining unchanged in a bitboard with `0x000FFFFF` and `0xFFFF000F` respectively. This is a result of the ordering

of the tiles on each bitboard; the bands of these faces consist of tiles all equally positioned on their respective bitboard.

For R, the mask `0xFF000FFF` is sufficient for coverage of the U, F, and D faces, however, the tiles maintained on the B face are found by `0x0FFFFF00`. To circumvent the issues that arise from using an invalid mask, the bits on the B face are shifted by 16. This enables the original mask to covers these bits, at which point (Section 5.2) can be used to combine the bitboards. The bitboard produced for the B face must be shifted backwards by 16 (see Section 5.1), conceptually the inverse of the preprocessing. An equivalent process is required for rotation of the L.

For F and B, the bits composing each band are found by four different masks. The mask for U is chosen arbitrarily as the desired format, and the remaining bitboards are shifted appropriately to match.

5.2 Storing Multiple Cubes

This section explores the implementation of the *CubeStore* class, used to store a collection of permutations, and its nested components.

5.2.1 The CubeStore Class

In order to efficiently interact with a significant number of unique cubes in the way described in Section 5.5, the final version, we require a data structure with the following three requirements:

1. the cubes are accessible in a first-in, first-out order (FIFO);
2. there is an efficient way to ascertain if a permutation has been added to the structure; and
3. we are able to retrieve the sequence associated with a stored cube of a specific permutation.

To do this, we implement a class, *CubeStore*. Each *CubeStore* has two instance variables, *queue* and *cubeTree*. *Queue* is a *FIFO* collection of *Cubes*, and handles the first requirement. The *cubeTree* variable is of the type *CubeTree*, a nested class, and is responsible for the two other requirements.

The *CubeStore* class implements methods for adding to the structure and for *FIFO* retrieval of elements, namely *Add* and *Dequeue*. There are two additional methods, *Contains* and *GetSequenceOfEqualCube*, responsible for requirements two and three above. These simply call methods of *cubeTree* with associated names, where the logic is handled.

5.2.2 The CubeTree Class

The *CubeTree* class stores permutations in a tree, with each level representing a single bitboard, i.e., face, of a permutation. Each leaf node contains the sequence associated with the *Cube* that resulted in its addition to the structure. This is done through a mapping of bitboard to node, i.e. from$uint \rightarrow CubeNode$, where *CubeNode* is a class representing each node in the tree. *CubeTree* has three methods, one to add a cube to the tree, one to check if the tree contains a passed cube, and a third to retrieve a sequence. The order in which *CubeTree* levels represent the bitboards is: $U \rightarrow R \rightarrow F \rightarrow B \rightarrow L \rightarrow D$.

This decision was made such that the *Contains* method requires checking, on average, as shallow a depth through the tree as possible. As the permutation is transformed, unchanging patterns are built; permutations solved by G_3 will have white tiles in the 2nd and 6th positions of the D face, and once solved by G_2, the value of five of the eight bitboard positions are known for both the D and L faces. As there are therefore fewer cases for some bitboards than others, the probability that distinct permutations are identical on certain faces is higher than on others. This probability decides their ordering through the tree.

5.2.3 The CubeNode Class

The *CubeNode* class represents each node of the tree. Each has two variables, *children* and *sequences*, with the former storing each branch node in a dictionary from *uint* → *CubeNode*, and the latter being used only for the leaf nodes, storing sequences in a dictionary from a *uint* representing the D face of each permutation to the associated sequence. The decision to use a dictionary to represent each pairing of D face to sequence, rather than each sequence having their own node, was made to limit the number of objects instantiated when adding a cube to the tree. Conceptually, this is still the same depth.

The core logic for the *Contains* and *GetSequence* processes in previous sections are contained in this class. If at any stage through traversal a dictionary does not contain a key representing the next bitboard, then the permutation in question is not present. *GetSequence* follows a similar process, recursively traversing the tree until the leaf node is reached and the sequence returned.

5.3 The Basic Algorithm

The algorithms to transition between each stage of the method are largely consistent. From an initial *Cube*, p, breadth-first search is used to transition between G_n and G_{n-1}, where each new *Cube*, C, is checked to be solvable by the sequences contained in the next subgroup. The traversal is done through an adjacency list related to the current group, A_n, at the index defined by the *lastMove* variable of each *Cube* object. A_n contains only rotations valid in relation to S_n; when unoptimised, each index in A_n is equal to S_n such that any rotation can succeed any other.

Assuming the initial permutation, p, is unsolvable by G_{n-1}, the method is given in Algorithm 1.

```
FIND-SEQUENCE-N-TO-N-1(p)
let CS be an empty CubeStore;
create cube object for initial permutation and add to CS;
while true do
    get next cube C from CS;
    foreach rotation r ∈ A_n do
        let C_r be the result of rotating C by r;
        try add C_r to CS;
        if C_r ∈ CS and is solved by a sequence in G_{n-1} then
            return sequence of C_r;
        end
    end
end
```

Algorithm 1: Transitioning from G_n to G_{n-1}.

The difficulty of implementation comes largely from identifying if a permutation can be solved by an element of each subgroup. As stated in the introduction of the method:

- G_4 sequences can only solve permutations with *good edges*;
- G_3 sequences in addition require permutations to have solved *DF* and *DB* edges;
- G_2 sequences in addition require a left-block and correctly permuted corners; and
- G_1 sequences require all but the cycle of the U face cubies to be correct.

Checking C_r is solved by G_3, G_1, or G_0. It is straightforward to check whether a permutation can be solved by a member of G_3, G_1, or G_0, as a permutation can be directly observed as either having or not having the relevant solved pieces.

Checking C_r is solved by G_4. G_4 can also be found through pattern matching, though instead of determining the exact position of cubies, it ensures each edge is correctly oriented via a hierarchy of tiles and faces. We value edge tiles in an order matching their bitboard representation, *U,D,F,B,R,L*, and refer to the greater-value tile of an edge as the *dominant* tile. An edge is considered a *good edge* in this system if it's highest value tile is on its highest value face. For example, if the *UF* edge were in the *LB* slot, it would be considered a *good edge* if the *U* tile was on the *B* face. This concept is applied to each edge, and if all edges are *good edges* then a permutation is solvable by G_4.

Checking C_r is solved by G_2. The process to check if a cube can be solved by a member of G_2 is slightly more involved, as while the left block can be confirmed in the same explicit fashion as the properties required for G_3, G_1, and G_0, additional checks must be made to verify the permutation of the corners is possible by rotation of a solved cube by only the *U* and *R* faces.

To do this, we find which of eight possible positions the *DFL* cubie is in, and insert it into the *DBL* slot with a specific sequence associated with each position. We do the same for the *DBL* cubie, determining which of the now seven available slots the piece could be in, and inserting the piece into the *DBL* slot. Simultaneously, the *DFL* piece is further rotated into the *DFL* slot, such that both are in their correct positions (though not necessarily oriented correctly).

The remaining unsolved cubies are then used to generate a bitboard representing their order around the *U* face. Each cubie maps to the value of the tile clockwise from the *U* tile. For example, the *UFR* tile maps to the value `0x5`, the value assigned to *R*. This bitboard is then compared to the bitboard `0x30905060`, which represents correctly permuted *U* corners. If a shifted version of the generated bitboard matches the desired one, then its cubies are of an appropriate cycle. Hence C_r can be solved by a sequence in G_2.

5.4 Optimisations

The implementation described in the previous section would yield a sequence that solves any cube eventually. However, a technically correct implementation is of no use if it does not execute in reasonable time. As such, a series of optimisations are introduced.

5.4.1 Three Rotations Per Face

When considering half turns as a single rotation rather than two, the total rotations required to solve a permutation in the worst case decreases from 26 to 20 [21]. Similarly, while the rotations *U*, *R*, *L*, *D*, and *F* are sufficient to solve any cube, introducing half and three-quarter rotations into the adjacency lists reduces the total number of rotations significantly.

If we were to rotate a cube by (F, R) and then attempt to transition from G_5 to G_4, the shortest sequence to do so would be (R, R, R, F). With inverse rotations included, this transition can be done with (R', F), half the rotations. To find a sequence of length four with only 1/4 clockwise turns, we need to consider in the worst case $\sum_{j=0}^{4} 5^j = 781$ rotations. With three rotations per face, the shortest solution is instead only $\sum_{j=0}^{2} 15^j = 241$ permutations.

5.4.2 Adjacency List Restrictions

A second optimisation is the reduction of size of the adjacency lists. Since all three rotations are now available per face, it is no longer required for a rotation to succeed itself as any sequence doing so can be transposed to one with only single turns. As a result, we are able to remove rotations of a face from its own adjacency list.

Beyond this, it is known that some sequences are equal. Trivial examples include sequences of only opposing sides, such as (U, D) and (D, U), as these rotations commute. Since these will result

in identical permutations, there is no need to consider both orderings of opposing faces in the adjacency lists, and so we limit the adjacency lists of D and R such that they can not precede U and L respectively. The effects of these changes to the adjacency list are shown in Figure 5.2, with N representing no previous rotation. Note that only the faces themselves are shown in the adjacency lists, as either all three rotations of a face are in an index, or none are.

Adjacency list A_5	
N	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R \rightarrow F$
U	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R \rightarrow F$
D	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R \rightarrow F$
L	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R \rightarrow F$
R	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R \rightarrow F$
F	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R \rightarrow F$

Optimised Adjacency list A'_5	
N	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R \rightarrow F$
U	$\rightarrow D \rightarrow L \rightarrow R \rightarrow F$
D	$\rightarrow L \rightarrow R \rightarrow F$
L	$\rightarrow U \rightarrow D \rightarrow R \rightarrow F$
R	$\rightarrow U \rightarrow D \rightarrow F$
F	$\rightarrow U \rightarrow D \rightarrow L \rightarrow R$

Figure 5.2: *The adjacency-list A_5, before and after optimisation.*

It is worth mentioning that the algorithm outlined earlier will prevent duplicate cubes from being added to the *CubeStore* with or without this optimisation. However, an attempt to add a *Cube* requires a full traversal of the tree where already present. As this would otherwise occur tens of thousands of times, optimising the adjacency lists guarantees that less traversal is required.

5.4.3 Final Rotation Guarantee

If permutation p is solvable by G_n but not G_{n-1}, a rotation in S_n but not S_{n-1} must be contained in the sequence that solves p. If not, the sequence would be composed of elements in S_{n-1}, and therefore p would also be solved by G_{n-1}. From this, when converting from G_n to G_{n-1}, we must only check if a permutation is solvable by G_{n-1} when rotations from $S_n \setminus S_{n-1}$ are conducted.

As an example, no rotation of U, R, or L will transition permutation p between G_4 and G_3, as all are permitted in both. We must only check $p \rightarrow G_3$ after a rotation of the D face. For $G_5 \rightarrow G_4$, we must only check after rotation by F or F'; rotating by $F2$ is equivalent to no rotation when converting edges from *bad* to *good*. This optimisation saves significant time for $G_n \rightarrow G_{n-1}$, as only a fraction of permutations are checked to be solvable by G_{n-1}, particularly in earlier stages, where many moves are available in each list.

5.5 Bidirectional Search

This section reduces our problem to one that can incorporate bidirectional search.

5.5.1 An Alternative Problem

Consider the alternative problem of finding a sequence between two specific permutations. In this case, we could conduct breadth-first search from both permutations, and find a connecting path when the two searches both find the same permutation. Regarding each permutation as a vertex of a graph, each rotation as an edge, this is equivalent to finding the midpoint of the shortest path connecting the two permutations. To find the same path with breadth-first search, without considering equivalent sequences, with r unique rotations, we must consider in the worst case $\sum_{j=0}^{m} r^j$ permutations before a sequence of length m is found. However, if we instead implement breadth-first search in both directions, we need to consider no more than $2\sum_{j=0}^{\lceil m/2 \rceil} r^j$.

For a sequence of length six and all eighteen possible rotations, this is the difference between $\sum_{j=0}^{6} 18^j = 36,012,943$ and $2 \cdot \left(\sum_{j=0}^{3} 18^j\right) = 12,350$ individual permutations.

5.5.2 Reinterpreting Our Problem

Our problem is one of finding a sequence to transform a permutation such that it can be solved with a smaller set of rotations. Reducing it into one of finding a sequence between two distinct permutations enables the use of bidirectional search, reducing the number of permutations considered.

We write that sequence s is a member of group G_n as $s \in G_n$, and introduce the notation '$\leftarrow$' such that $p \leftarrow G_n$ signifies permutation p can be solved by a member of G_n. Additionally, we represent *compositional equivalence* as '$\equiv$'.

Consider a permutation p, where $p \leftarrow G_n$ but $p \nleftarrow G_{n-1}$, and permutation q', such that for any permutation $q \leftarrow G_{n-1}$ we have $q \equiv q'$. If we can find permutation p', where $p' \equiv p$, and a sequence $s \in G_n$ such that rotating p' by s yields q', then, by the properties of p' and q', rotating p by s will result in a permutation solved by a sequence of G_{n-1}.

5.5.3 Implementing Bidirectional Search

Solving permutation $p \leftarrow G_n$ requires recursively converting to a permutation solved by G_{n-1} through the rotations of S_n, until $p \leftarrow G_0$.. By finding a generic representation of all permutations solved by G_{n-1}, q', and an appropriate p', a sequence to transition $p' \rightarrow q'$ applied to p results in $p \leftarrow G_{n-1}$. This is

The requirements for initial permutation p' and target permutation q' differ between stages, however, the means of initialisation of q' and transposition from $p \rightarrow p'$ are consistent throughout. Each instance of q' can be explicitly defined, and the process for generating p' is generalised as shown in Algorithm 2.

```
GENERATE-P'()
let p′ represent a blank permutation;
foreach face f of p do
    foreach tile t of f do
        if t is represented in q′ then
            add t to p′;
        end
    end
end
return p';
```

Algorithm 2: Pseudocode for generating p', where $p' \leftarrow G_n$.

While all stages can be generalised in this way, each implementation is unique. Specific requirements for each instance of p' and q' follow.

$G_5 \rightarrow G_4$**.** For the first step, we follow the edge tile hierarchy as defined in Section 5.3. As per Algorithm 2, a tile t is required for q' (and therefore p') if it is the *dominant* tile of an edge. As in Figureřef4g-p, rather than directly adding t to the bitboard representation of p', a generic value is used to signify the presence of a *dominant* edge. Only the orientation of each edge is required to determine if a permutation can be solved by a member of G_4.

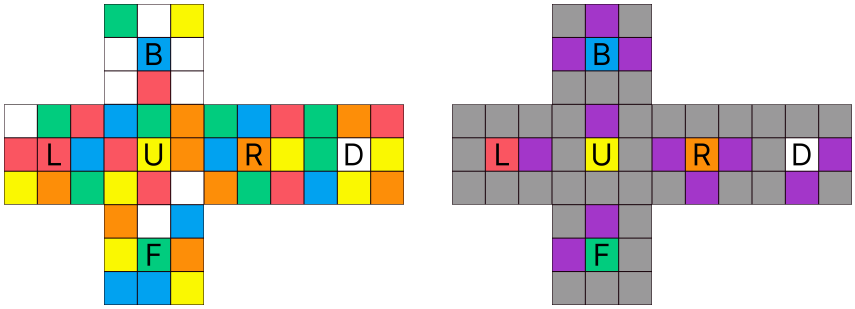


Figure 5.3: *Representations of p and p' as required for $G_5 \rightarrow G_4$.*

As is the case with all versions, the G_4 representation of q' could be found by applying the G_4 implementation of Algorithm 2 to a solved cube. However, the implementation defines q' explicitly; for a given stage q' is always the same, regardless of p or p'. For G_4, we define q' as a *Cube* object with bitboard {`0x0F0F0F0F`, `0x0F0F0F0F`, `0`, `0`, `0x000F000F`, `0x000F000F`}, shown left in Figure 5.4. This bitboard can be thought of as a representation of any cubes with oriented edges; the *dominant* tile of each edge is in the *dominant* position.

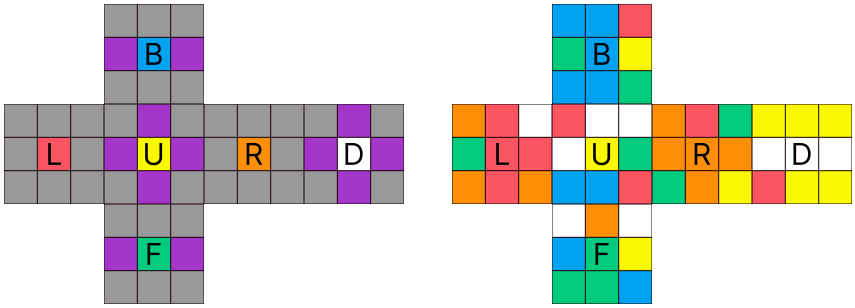


Figure 5.4: *Representations of q' and q as required for $G_5 \rightarrow G_4$.*

The shortest path found via bidirectional search between p', equivalent to the example of 4, and q', is $(D2, F', R', U', L', F')$. Resulting in the permutation of Figure 5.4, this matches the solution found previously by breadth-first search.

$G_4 \rightarrow G_3$**.** For this transition, only the non-D tiles of the DF and DB edges are required for representation of all $q \leftarrow G_3$. The resulting conversion is demonstrated in Figures 5.5 and 5.6,

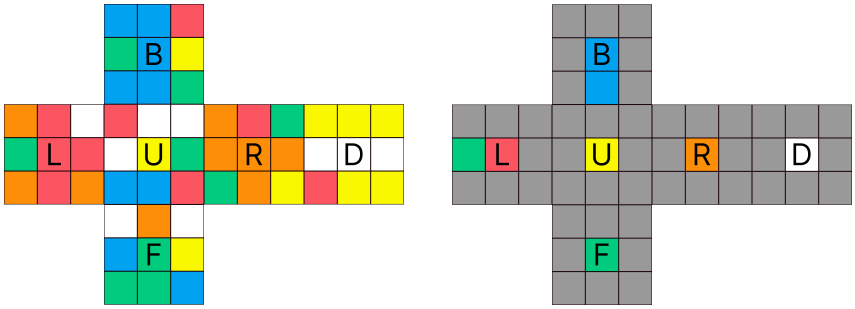


Figure 5.5: *Representations of p and p' as required for $G_4 \rightarrow G_3$.*

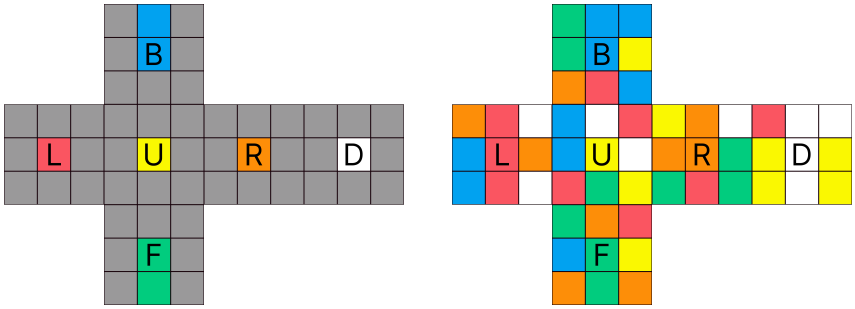


Figure 5.6: *Representations of q' and q as required for $G_4 \rightarrow G_3$.*

$G_3 \rightarrow G_2$. The transition from $G_3 \rightarrow G_2$ is still slightly more involved. By introducing a series of rotations not physically possible to represent final rotations from q', complex permutations of each corner cubie, bidirectional search would yield a solution satisfying the problem. However, this would require the introduction of over hundred of initial rotations from q' for optimal transitioning to G_2, found in Section 7.6.2. Instead, the reduction to p' and q' is only utilised for the building of the left-block, at which point the original method for confirming a permutation can be solved by G_2 is used.

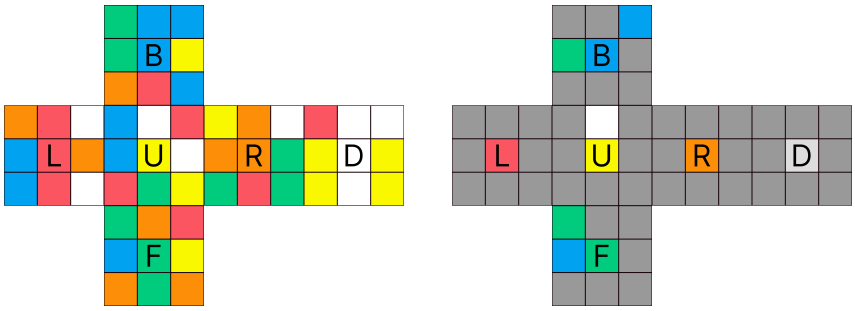


Figure 5.7: *Representations of p and p' as required for $G_3 \rightarrow G_2$.*

For building the left-block, only the positions of the non-L tiles of the L edges and the positions of the B and F tiles of the DF corners are required. Application to the example is shown in Figures 5.7 and 5.8.

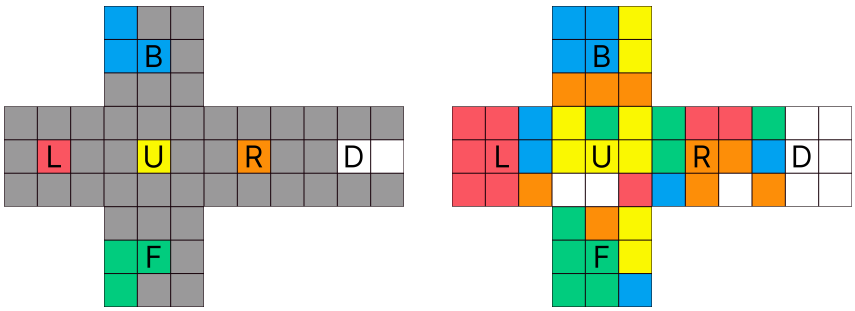


Figure 5.8: *Representations of q' and q as required for $G_3 \rightarrow G_2$.*

$G_2 \rightarrow G_1$ and $G_1 \rightarrow G_0$. The transposition from $p \rightarrow p'$ and each $q \rightarrow q'$ is beneficial in previous stages as it facilitates the use of bidirectional search. This is not the case for the transitions $G_2 \rightarrow G_1$

and $G_1 \rightarrow G_0$. The solved cube is equal to q' for G_0, and as G_1 consists only of permutations a single rotation from being solved, it is more efficient to use the solved cube as q', and use rotations of the U face to check if a permutation is a member of G_1. In the case of G_1, this final rotation of U is removed from the returned sequence.

5.6 The Optimised Algorithm

To utilise the previous optimisations, a revised version of Algorithm 1 is required. First, we must conduct our search in both directions. To do this, we introduce a second *CubeStore*, and modify the body of the while loop such that a step is taken in two directions on each iteration, rather than in one. The conditional statements must also be modified slightly, as there is no longer a requirement to check if a cube can be solved by G_{n-1}; as this property is now guaranteed when matching permutations are found.

We also introduce our improved adjacency lists, denoted A'_n, which contain three separate rotations per face, and stop trivial equivalent sequences from occurring. Finally, we introduce multiple initial target permutations into the implementation, all one rotation by the face being dropped in the transition from $G_n \rightarrow G_{n-1}$ from q'. This handles the final rotation guarantee by explicitly conducting the first tier of the breadth-first search from $q' \rightarrow p$.

A modified version of the solving algorithm is presented in Algorithm 3, with parameters for initial permutation p' and list of target permutations $q'[]$. Note that in the case of $G_3 \rightarrow G_2$, the combined sequences are returned only if the result of applying the combined sequence is explicitly confirmed to be a member of G_2 in the original way. If not, the process continues.

```
OPTIMISED-FIND-SEQUENCE-N-to-N-1(p', q'[])
let CS'_p and CS'_q be empty CubeStores;
create cube object for initial permutation p' and add to CS'_p;
create cube objects for permutations from q'[] and add to CS'_q;
while true do
    get next cube C'_p from CS'_p;
    foreach rotation r ∈ A'_n do
        let C_p'r be the result of rotating C'_p by r;
        try add C_p'r to CS'_p;
        if C_p'r ∈ CS'_p and C_p'r = C_q'm, C_q'm ∈ CS'_q then
            return the combined sequences of C_sr and C_q'm;
        end
    end
    get next cube C'_q from CS'_q;
    foreach rotation r ∈ A'_n do
        let C_q'r be the result of rotating C'_q by r;
        try add C_q'r to CS'_q;
        if C_q'r ∈ CS'_q and C_q'r = C_p'm, C_p'm ∈ CS'_p then
            return the combined sequences of C_q'r and C_p'm;
        end
    end
end
```

Algorithm 3: Optimised transitioning from $G_n \rightarrow G_{n-1}$.

5.7 Other Attempted Methods

As is often the case with larger projects, a number of methods and techniques were implemented but ultimately not contained in the final version. A selection of these are discussed in this section.

5.7.1 Representing A Cube

The implementation of a single cube evolved throughout development. Original versions stored each tile as a two dimensional *Char* array, then as a single *String*, utilising string modifications in a way similar to bitwise operations of the bitboards, except significantly less efficient. Both versions were present in functional implementations of the solving algorithm, and while the single *String* implementation was an improvement over the *Char* array, neither ran in an acceptable amount of time.

5.7.2 Representing Multiple Cubes

The process to store multiple cubes went through five distinct stages throughout development.

Implementations In Java. The initial versions were developed in Java. The first implemented the storage of cubes as an *ArrayList*, with *FIFO* retrieval completed by removing the first item in the list when a new cube was required, and search for a sequence completed through the already implemented *Contains* method. However, as the size of each *ArrayList* grew exponentially in size, the time taken to determine if a permutation was present in the list was insurmountable, as this action is completed in $O(n)$ time complexity.

To avoid this issue, a parallel data structure was introduced in the form of a *HashSet*, such that the existence of a permutation could be determined via a lookup table in constant time. This set would not be modified when cubes were removed from the *ArrayList*, only when they were added. This was functional, and relatively fast, however the maintenance of four separate data structures for cubes — bidirectional search and two structure each way — was overly complex.

As such, the third implementation was a modified *LinkedHashSet*, an existing Java data-structure that stores a set in order of insertion. This structure did not have implemented *FIFO* retrieval, a challenge overcome by the introduction of a class extending *LinkedHashSet* with a method initialising an iterator over the structure and storing the first element, before removing the element from the set. When introduced, this resulted in a near-instant solution.

Implementations In C#. Due to development issues explored in Chapter 6, the algorithm was converted to C#. There is no equivalent in C# to the *LinkedHashSet*, and so this was replaced with encapsulated *Queue* and *Set* objects. This implementation was unsuccessful. On exploration of the documentation, it was found that methods to generate the *hashCode* of an array differ between C# and Java. Java provides the means to lookup an array by its elements an array from its elements, but no such functionality is available in C# — it can be done only in relation to an objects location in memory.

An attempt to circumvented this issue by overriding the *equals* and *hashCode* functions to be explicitly related to the contents of the each bitboard. This implementation was successful almost every time, but fell victim to the pigeonhole principle. A *hashCode* in C# in a 32-bit integer, able to represent 4,294,967,296 unique value, but as we have *significantly* more permutations than this it is a guarantee that multiple permutations will have the same *hashCode*. This entirely corrupts their use.

At this stage of development, the *CubeStore* was designed and implemented. Avoiding the issue by implementing a lookup table per node for subsequent bitboards, this version solved the difficulties in conversion to C#.

5.7.3 Recursive Solving

A separate direction was taken during development in search for a less memory-intensive algorithm. Rather than storing a sequence for each cube, a recursive version of the method to find a path between two permutations was created, such that a list of adjacent permutations would

be produced instead. The recursive method still required storing each permutation to confirm whether or not a matching state had been found, however, it avoided the overhead of storing an associated sequence with each permutation.

Algorithm 4 presents an approach that finds a path of adjacent permutations between permutations C'_p and C'_q.

```
GET-PATH(C'_p, C'_q)
let C_m equal GET-MIDPOINT(C'_p, C'_q);
if C_m = C'_q then
    return a list containing C'_q;
end
return GET-PATH(C'_p, C_m) + GET-PATH(C_m, C'_q);
```

Algorithm 4: Recursive transitioning from $G_n \rightarrow G_{n-1}$.

Algorithm 5 presents an implementation of the algorithm to find the midpoint between two permutations, when provided with permutations C_p and C_q.

```
GET-MIDPOINT(C'_p, C'_q)
let CS'_p and CS'_q be two lists;
add C'_p to CS'_p and C'_q to CS'_q;
while true do
    get next cube C'_p from CS'_p;
    foreach rotation r ∈ A'_n do
        let C_p'r be the result of rotating C'_p by r;
        try add C_p'r to CS'_p;
        if C_p'r ∈ CS'_p and C_p'r = C_q'm, C_q'm ∈ CS'_q then
            return C_q'm;
        end
    end
    get next cube C'_q from CS'_q;
    foreach rotation r ∈ A'_n do
        let C_q'r be the result of rotating C'_q by r;
        try add C_q'r to CS'_q;
        if C_q'r ∈ CS'_q and C_q'r = C_p'm, C_p'm ∈ CS'_p then
            return C_p'm;
        end
    end
end
```

Algorithm 5: Finding the midpoint of two permutations C'_p and C'_q.

As demonstrated, the permutation found in both directions is regarded as the mid-point between the starting permutation and its target. The process repeats recursively, finding new mid-points between the start and old mid-point, and the old mid-point and target. If the start and target are one apart, the mid-point is equal to the target. This equality is the base case, at which point the target is added to the list of adjacent permutations. Note that the starting permutation must also be added to this list.

After all branches return, each pair of adjacent permutations are compared to determine which rotations were conducted, and the concatenation of these rotations is returned as the solution.

This method was only implemented in Java. While the memory requirements were lower, it required more time to execute on average. This is likely due to overhead from the significant number of recursive function calls, and the shifting of control between each accessed function.

6 Visualising The Method

While the solutions produced by the implementation of the method could be manually performed on a cube, this is time consuming, prone to human error, and feels incomplete. To demonstrate the method, we desire a way to visualise a solution. This requires a suitable representation of a permutation, and an appropriate means of interaction.

This Chapter details the development of such a tool, with the intention of introducing the method not just to those with existing knowledge of group theory or those already able to solve a Rubik's Cube, but to a wider audience.

6.1 Planning The Visualisation

This section defines a plan for the visualisation of the method, split between design of the interface, and of the cube itself.

6.1.1 The Cube

The desired representation for the cube is a 3D model.

A net was considered as an appropriate pairing to this, but as it cannot show active rotation in a way that's easy to interpret, it is no benefit to this process. The benefit of using nets throughout the dissertation was the unambiguous representation of the reverse side of the cube, but this issue will instead be avoided in the visualisation by enabling a 3D-Model of a cube to be rotated around its own axes.

The initial perspective will remain from the *UFR* corner, and the colour scheme will be consistent with that used previously; green for F, blue for B, yellow for U, white for D, red for L, and orange for R.

6.1.2 The Interface

In conjunction with the cube, the interface will contain a series of buttons each responsible for one stage of the method. There will be buttons present to generate a new permutation, as well as to repeat the most recent permutation provided one has been generated previously. A text field will be shown also, showing the current set of rotations required to solve the current permutation, i.e. the last button pressed.

When a button is clicked, the associated sequence will appear next to the button itself, such that a user could 'follow along' on their own cube, or simply to see which rotations were made. If the user skips a step of the method, the sequences to solve the previous stages will be conducted and displayed next to their respective buttons as though each were pressed in order.

If no sequence is required for a step, i.e., a permutation is by chance already solved by a further nested subgroup, a relevant message will be displayed to avoid confusion. This will occur most often for the final step of the method, where a quarter of solutions require no rotations. Additional information will be found via a 'help' button and pop-up window, containing both instructions of use and a digestible explanation of the method itself.

6.2 Implementing The Interface

The visualisation was ultimately made in Unity, meeting all informal requirements from Section 6.1 buttons are present for each stage of the solve, the text is displayed in a sensible fashion, and there exists a 3D model of a cube which can be spun by the user and rotated via the buttons. A help guide is also available for explanation of both the interface and the method, all of which laid out appropriately as in Figure 6.1.

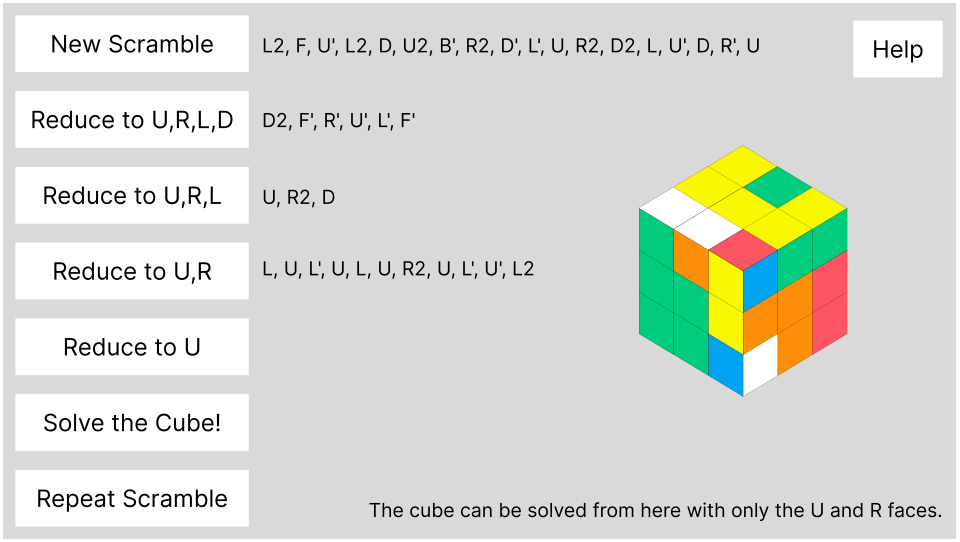


Figure 6.1: *An abstraction of the final interface.*

6.3 Implementing The Cube Object

The visualisation contains an implementation of a cube wholly different to that of the solving algorithm. Implemented in Unity, the 3D model consists of a series of *quads*, Unity's 2D rectangle object.

More precisely, each cubie consists of 6 black squares to form the colourless cube, and up to three coloured *quads* to represent its tiles. These components are grouped into containers named after their piece, i.e., the *quads* for the *UFR* cubie are grouped into the *UFR* container. These 26 *cubies* are then grouped together into a *Cube* object, along with three additional objects. These objects are *Core* and *CoreTarget*, used for the rotation of each side, and *SpinTarget*, used for rotation of the whole cube. Finally, the *Cube* object is placed in a *CubeHolder* object, to isolate any rotations of the cube from the overall scene.

6.4 Rotating The Cube

As Unity projects are often real-time environments, i.e., live, the implementation provides the ability to check conditions on each frame by extension of the *MonoBehaviour* class. Utilising this, we can execute a function each frame a cursor is hovered over the cube.

The state of interaction with the cube is always one of six stages;

1. not being interacted with;
2. being hovered over;

3. actively being clicked;
4. being held and spun;
5. actively being released;
6. 'snapping' to an orientation centered around the nearest corner on release.

The rotation process begins when the cursor is hovered above the *Cube*. While the cursor is hovering, a check is made each frame to determine whether the mouse has been clicked. This two-step process prevents clicks not on the cube from triggering a rotation. If the mouse is clicked the cursor is over the cube, then two variables associated with the cube are modified; the *holding* flag is set to true, and the *initialMousePosition* vector is set to the current position. While the cube is being held, i.e., *holding* is true, the cube continuously rotates towards the cursor. If the mouse is released, then *holding* is set back to false and the cube rotates towards its closest appropriate position.

The *Update* method, called every frame, is shown in Algorithm 6.

```
UPDATE()
if holding then
    if mouse is being released then
        set holding to false;
        determine final orientation
    else
        rotate towards cursor
    end
else
    rotate towards final orientation
end
```

Algorithm 6: Rotation of the Cube Object, called each frame.

Rotation towards the cursor is performed in the direction of the vector found as the difference between the *initialMousePosition*, the position of the cursor when the cube was clicked, and its position in the current frame.

This approach does limit rotation of the cube to two axes, as the cursor can move in only two dimensions, but rotation of the third can be achieved by combination of the existing two. Initial versions tested the use of a button to flip which axes were in use, but this was abandoned- the existing method was sufficient for the purpose.

The final stage of rotation — where the cube 'snaps' to an appropriate orientation — is the most complex. This process is done via the *SpinTarget* and *Cube* objects, by orienting *SpinTarget* to the desired orientation for the cube, and then rotating the *Cube* in the direction of this point. *Spin Target* is oriented within its own method, by rounding each angle of rotation of the cube object across the x, y, and z axes to the nearest 90 degrees.

If the cursor is dragged only a short distance, the closest orientation may be the initial one. In order for a rotation to a new orientation to be made in these instances, as is more natural in practice, the orientation of the *SpinTarget* object must be determined appropriately. This is done with explicit rotation in the direction the cursor, with bias towards rotation around the y axis, i.e., the direction of the U face, regardless of the magnitude of the direction of the vector.

6.5 Rotating The Sides

As each cubie is represented separately,it is not immediately known which cubie are on a face if it is rotated. This is handled by 'sticking' a series of cubies together, and rotating them around their average position.

Taking rotation of the *U* face as an example, we first rotate the *CoreTarget* object to the orientation $(0, 90, 0)$. We ensure the orientation of the *Core* object is $(0, 0, 0)$, and set the parent of each cubie on the *U* face to be the object *Core*. This can now be rotated frame-by-frame towards *CoreTarget*, rotating each cubie around the axis of this central piece rather than its own. Without this attachment to the *Core* object, each cubie would spin in place.

After the rotation has completed, i.e., the core is of the orientation $(0, 90, 0)$, we set the parent of each *Cubie* back to the overall *Cube* object, and reset the rotation of *Core* to $(0, 0, 0)$.

Multiple rotations can be done in sequence by their addition to the *moveQueue* instance variable, present in the class responsible for rotation of each side. This can be done via two class methods, one which adds an individual rotation to the queue, and another appending an entire sequence.

On each frame, the *Update* method of this class checks whether a rotation is currently taking place. If so, then this rotation continues. If not and the queue has elements, then the next rotation begins.

6.6 Interactions With The Solving Algorithm

This section outlines the internal process that occurs when a button is clicked.

6.6.1 The ButtonHandler Class

As intended, the interactions with the algorithm are handled through a series of buttons. Each run various methods of the *ButtonHandler* class, initialised automatically at runtime by *UnityEngine*.

The *ButtonHandler* class has five instance variables. Two are references to other active scripts; the first being the *RotateSide* script, responsible for rotating each side, and the second responsible for the displaying of text, aptly named *TextHandler*. Along with these are the classes responsible for the algorithm execution itself; a *Cube* as implemented in Section 5.1; a *CubeSolver*, which generates each sequence; and a *uint* array, responsible for storing the sequence to generate the most recent initial permutation such that it can be repeated if the user wishes to do so.

Also contained in the class are methods for each button and a helper method to solve the cube. When a permutation is restarted, or a new one generated, the previous state is solved with this.

The method for generating a new initial permutation clears the sequences from the screen via the *Clear* method of *TextHandler*. It then solves the cube as described in Algorithm 3, generates a new sequence, and stores this in the appropriate instance variable. This is then applied to the cube such that it is now a new permutation. This sequence must also be applied to the 3D model of the cube, done so through the *RotateSide* script. A textual representation of the new sequence is displayed.

The method for repeating the previous initial permutation clears the screen and solves the cube in the same way as for a new permutation, but instead of generating a new sequence by which to permute the cube, it retrieves the one currently stored by the class. If none are stored, i.e., the *NewScramble* button has not yet been clicked, the button is unresponsive.

6.6.2 Conducting Each Step Of The Method

A button exists for each stage of the method. As these stages must be completed in order, but can be clicked freely, each method must first ensure all previous stages have been executed. To implement this, each method recursively calls that of the previous stage before executing its own step (except the first stage, which does not call another method). This process is shown in Algorithm 7 for method step n.

```
SOLVE-STEP(n)
if n ≠ 1 then
    call SOLVE-STEP(n − 1);
end
let sequence s solve G_n → G_{n−1};
apply s to the cube;
add s to the 3D Model queue with RotateSide;
display s with TextHandler;
```

Algorithm 7: The method called when button $G_n \rightarrow G_{n-1}$ is pressed.

Note that the sequences are only displayed by the *TextHandler* if there is currently no text present in their respective location. This means that when non-consecutive buttons are clicked all previous text fields are filled, but when subsequent buttons are clicked, if a sequence is already displayed then it is not overwritten.

6.7 The Initial Version

Initially, the project was to be completed in Java. This decision was made due to familiarity with the language and its offerings, however there were significant issues in creating the interface.

While the 3D model of a cube was implemented successfully, after the solving algorithm itself was, there were difficulties with buttons and direct interactions. The technology first selected, Java Processing, provided sufficient ability to interact with the screen via keyboard inputs, but fell short with mouse interactions. The aim of the technology is to utilise visual demonstrations for learning to code, not a project of this kind.

Due to these limitations, the project was converted to Unity, written in C# and designed specifically for real-time 3D modelling. The internal processes for rotation of the cube were near-identical in both implementations, with only the choice of language differentiating the two.

While this change introduced the issues previously discussed when storing multiple cubes, the finished product is notably more polished than it would have been had the original technologies been maintained.

7 Evaluation

This chapter evaluates the effectiveness of the visualisation, performance of the implementation, and the benefits of the various optimisations. An upper-bound is found for the worst-case number of permutations stored at a given time, for all possible orders of reduction of the minimal generating set. The search for an upper-bound was conducted to guarantee that any permutation could be solved in reasonable time and as a means of comparison between potential implementations.

7.1 Visualisation

As part of the development process, the visualisation was tested to ensure the solving method is accurately conveyed to users. Participants were selected randomly from available peers, including students from all five undergraduate levels, and all colleges of the University of Glasgow. This approach was taken in line with the requirements of Chapter 6, where the visualisation should be sufficient in introducing the method to an audience not necessarily experienced in relevant areas.

7.1.1 Initial Testing

The initial testing involved assigning each participant to a group based on experience with group theory and ability to solve a Rubik's Cube; participants were split in an attempt to avoid relevant experience from impacting the results. They were then asked to undertake a series of tasks on the visualisation, encapsulating all of the main functionality.

Participants were observed in conducting these tasks to yield implicit feedback, before being asked to quantitatively evaluate how intuitive they found the interface. Finally, they were asked to complete a series of qualitative questions, across which the participant could suggest any potential changes, and give their best explanation of the method. These questions were contained in the survey to ensure both the users opinion and knowledge were captured- including only quantitative or qualitative feedback would not paint the full picture.

The results of initial testing were negative. While there was only one task uncompleted by all participants, many tasks took significantly longer than expected. Transformation to specific generating sets seemed particularly difficult, seemingly due to the initial information provided to users being inadequate. The quantitative feedback on how intuitive the tasks felt averaged 3.625 out of 5 on a Likert scale, less than desired. In describing the solving method, most participants mentioned group theory, yet only one provided an explanation conveying an acceptable level of understanding.

7.1.2 Secondary Testing

Before further testing was conducted, a series of changes were made to the interface. The buttons for each step were renamed, the action to spin the cube was changed from right-click to left-click, and the 'Help' section was expanded and divided into sections for the interface, method, and relevant mathematics. Importantly, the explanation of the method was significantly more detailed, and significantly more correct.

Testing the updated interface, with new participants, was more successful. The completion rate of the tasks was notably higher, and though a larger percentage of participants had no experience, the group on average demonstrated an improved level of understanding. Omitting a significant outlier, the average of the quantitative feedback increased from 3.625 to 4.444 out of 5.

The written feedback for the updated interface was also more positive; participants seemed to find the interface more intuitive post introduction of clearer explanations of the method. The only issue that persisted was the language used in explaining the underlying mathematics, though this was expected; there is a balance between instructions that cover all relevant information, and instructions too long for a user to maintain interest.

That said, the quality of the explanations of the method given by participants were of a high enough standard to conclude that the visualisation is fit for purpose.

7.2 Testing the Implementation

As part of the development process, a series of unit tests were introduced. While it is infeasible to test all possible cases, each method is tested in line with all potential interactions. This includes all unique rotations of a *Cube*, all methods and cases for addition to and retrieval from the *CubeStore*, as well as tests for each cube-masking and sequence-generation method in the *Solver* class.

Though significant testing was done, and correctness of the implementation is vital, the focus of the evaluation is the effectiveness of the optimisations in improving the efficiency of the implementation in the worst case. As each permutation can lead to a number of others, the total number of permutations stored at a time increases exponentially as the length of a sequence grows. We consider the number of permutations stored concurrently during execution to be our measure of success.

As it is infeasible to exactly calculate the number of states stored in the worst case, as there are simply too many initial permutations to consider, we calculate upper bounds for the basic and optimised versions of the implementation and use these for our comparison.

Calculations are done under the assumption that all rotations of a face are distinct, i.e. *R* and *R2* are both possible rotations. The recursive implementation is not mentioned, as more permutations are guaranteed to be stored than for the optimised implementation in all cases.

7.3 An Initial Upper Bound for Implemented Subgroups

While the number of sequences each generating set can produce is infinite, there are a finite number of permutations solved by each set of sequences. We now find this number for each implemented subgroup, i.e., for G_n where $n \in \{5, 4, 3, 2, 1, 0\}$.

In each case, we calculate the total number of permutations as the product of:

- the number of permutations of edge cubies;
- the number of orientations of edge cubies;
- the number of permutations of corner cubies;
- the number of orientations of corner cubies.

Since, two pieces cannot be interchanged with no other impact to a permutation, this product must be divided by two [27]. This division can be thought of as the mapping from each permutation with a pair of interchanged cubies to the permutation where they are swapped back.

Using this calculation we find:

$$\frac{12! \cdot 2^{11} \cdot 8! \cdot 3^7}{2} = 4.3252\text{e+}19 \text{ permutations in } G_5. \tag{7.1}$$

For G_4, we guarantee the orientation of each edge, and so the number of orientations of edge cubies can be removed from our calculation. Therefore, there are:

$$\frac{12! \cdot 8! \cdot 3^7}{2} = 2.1119\text{e+}16 \text{ permutations in } G_4. \tag{7.2}$$

For G_3, we fix the location of two edges. This reduces the number to:

$$\frac{10! \cdot 8! \cdot 3^7}{2} = 1.5999\text{e+}14 \text{ permutations in } G_3. \tag{7.3}$$

For G_2, we fix the location of an additional edge and two corners. Each permutation with this property contains either correctly permuted corners, two adjacent corners that must be swapped, or two corners diagonally opposite on a face that must be swapped before it is a guarantee that a permutation is solved by G_2. There are $4! = 24$ possible cases after solving the D corners, partitioned into six equivalence classes of four permutations, equivalent where U edges are of the same cycle. Of these six classes, one contains the permutations solved by G_2 — the class with correctly cycled edges. As the pre-processing steps can be reversed without changing this property, we find we have:

$$\frac{7! \cdot 6! \cdot 3^5}{2 \cdot 6} = 73,483,200 \text{ permutations in } G_2. \tag{7.4}$$

Sequences in G_1 can only rotate the U face, and as such we have only four permutations. For G_0 we have a generating set of only the identity sequence, and as this cannot affect a permutation in any way, G_0 solves exactly one permutation.

7.4 Total Permutations For All Subgroups

It was to be expected that decreasing the size of the generating sets would yield a decreasing number of possible permutations, as each generating set is a subset of the previous sets. However, only one possible order of removal has so far been considered. The method removes rotations from generating sets in a pre-defined order, but modifying this order impacts the number of permutations solved by each generated group; while S_{UR} can generate 73,483,200 permutations, the set S_{RL}, of the same order, generates only 16.

Although there are $6! = 720$ distinct reduction orders, there are few compositionally distinct ways to form a set due to the mappings that occur between them. A bijective function exists between generating sets with rotations that can be transposed to each other through observing a permutation from a different frame of reference before applying a sequence. For example, S_U could be transposed to S_R by rotating a permutation through the B before rotation.

The presence of a bijective function between sets is an equivalence relation. Table 7.1 provides an example from every class of generating set formed by the relation induced by such bijections.

ID	Generating Set Description	Example Set	Permutations
A	Any five rotations	S_{URLDF}	4.3252e+19
B	Four rotations, adjacent removed	S_{URLF}	1.8022e+18
C	Four rotations, opposites removed	S_{URLD}	2.1119e+16
D	Three rotations, one per axis	S_{URF}	1.7066e+14
E	Three rotations, only two axes	S_{URL}	1.5999e+14
F	two rotations, adjacent	S_{UR}	73,483,200
G	two rotations, opposite	S_{RL}	16
H	one rotation	S_R	4
I	no rotations	S_0	1

Table 7.1: *A member of each equivalence class of generating sets for sequences, under bijection.*

We can infer from Table 7.1 that if rotations cannot be reintroduced, there are only five ways to traverse through the classes from A to I which are:

1. $A \to B \to D \to F \to H \to I$
2. $A \to B \to E \to F \to H \to I$
3. $A \to B \to E \to G \to H \to I$
4. $A \to C \to E \to G \to H \to I$
5. $A \to C \to E \to F \to H \to I$

where ordering five is the chosen solving method.

Under the greedy heuristic of the minimum number of permutations available per step, ordering four would be chosen in which to remove rotations from the available set. However, this does not consider the benefits that arise from the optimisations of Section 3.

7.5 An Upper Bound for the Basic Implementation

To calculate an upper bound for the unoptimised implementation, for each step we sum the number of permutations reached when the desired sequence is the maximum required length for the attempted transition. If at each step we are able to make one of m different moves, and we make r rotations total, the overall number of permutations checked in finding the sequence in the worst case is given by $\sum_{j=0}^{m} r^j$ different moves.

In practice the values are likely much lower than the above estimation. Sequences occasionally result in the same permutation, and hence 'branches' are pruned as these duplicate permutations are not reintroduced into the queue, however this summation still yields a valid upper bound. Using this worst case bound and Algorithm 1, in Table 7.2 we present the total number of *Cube* objects stored in each stage in the worst case.

Step	r	m	Cubes Checked	Available Permutations	Worst-Case
$G_5 \to G_4$	15	11	9,267,595,563,616	4.3231e+19	9,267,595,563,616
$G_4 \to G_3$	12	4	22,621	2.0959e+16	22,621
$G_3 \to G_2$	9	20	1.3677e+19	1.5999e+14	1.5999e+14 + 1
$G_2 \to G_1$	6	20	4.3873e+15	73,483,196	73,483,196 + 1
$G_1 \to G_0$	3	1	4	4	4

Table 7.2: *An upper bound for the total cubes stored at each stage by the core algorithm.*

Note that the worst case value of m was calculated through brute force for the first two, and last, steps of the method, with all three rotations per face. However, this was not computationally feasible for the other steps. Twenty was selected for m as it is the maximum length required to solve any permutation with all rotations, and though it is not guaranteed to be the longest required for these subgroups, it provides a consistent means of comparison between implementations, and ultimately does not affect the conclusion.

The available permutations are lower than as found in the previous section as these do not take into account the number of permutations also solvable by nested generating sets, which can be removed from each total. For example, permutations solved by both G_5 and G_4 need not be considered for $G_5 \to G_4$, as the implementation returns a sequence if one is found.

We find our worst case from Table 7.2 is the transition from $G_3 \to G_2$, considering up to 1.5998e+14 + 1 permutations — all permutations solved by G_3, and one solved by G_2.

7.6 An Upper Bound for the Optimised Implementation

7.6.1 Bidirectional Search

Bidirectional search permits the division of our sequence into two halves, instead of a single sequence of length m. The final rotation guarantee allows us to convert from $\sum_{j=0}^{m} r^j$ to $r_q \sum_{j=0}^{m-1} r^j$, where r_q is the number of possible final rotations. The adjacency list allows us to reduce the number of possible rotations to at most three less than the number of rotations, r', except for the first rotation from p' which must remain r.

Dividing the sequence into two is more complex than completing an entire step in one direction at a time. Each iteration, only a single permutation from p' and q' are extended, and if $r \neq r_q$, then each 'tier' of the breadth-first search relating to the lesser of the r and r_q will be completed first. The worst-case is therefore calculated as twice the number of permutations added in the direction that completes the tier bringing the total number completed to the value m. Table 7.3 presents the worst-case permutations considered for each step, using the listed values as parameters.

Step	r_p	r	r_q	m	Worst-Case
$G_5 \rightarrow G_4$	15	12	2	11	1,085,811
$G_4 \rightarrow G_3$	12	9	3	4	244
$G_3 \rightarrow G_2$	9	6	3	20	217,678,234
$G_2 \rightarrow G_1$	6	3	3	20	354,292
$G_1 \rightarrow G_0$	3	0	0	1	5

Table 7.3: *An upper bound for the permutations considered by the optimised algorithm per step.*

Note that while the result for $G_1 \rightarrow G_0$ is worse than that of the basic implementation, in practice the worst-case is still four permutations; the implementation does not use this method to complete this step.

For $G_3 \rightarrow G_2$, we also consider a mapping that entirely reduces our problem of $p \rightarrow q, q \in G_2$ to one of $p' \rightarrow q'$. In this context, q' is a representation of all permutations where the left-block is solved, two corners and three edges, and where the permutation —but not orientation— of the remaining six corners are correct. To ensure each valid permutation of corners is the same distance to q', we require 2160 unique final rotations from q'. This is $6! \cdot 3$ — each of the 6! possible permutation of the unsolved corners, rotated further by *L*, *L2*, and *L'*, and requires the traversal over significantly more permutations than required when using bidirectional search for only the left-block-only method.

We hence observe from Table 7.3 that while our worst-case remains the transition from $G_3 \rightarrow G_2$, there is a significant improvement from the upper bound of the basic algorithm.

7.6.2 Symmetrical Reductions

In this section, we explore how *compositional equivalence* enables efficient pruning of branches where equivalent branches have already been explored. This is known as *symmetry breaking*, a powerful tool for the solving of graph problems [10, 9].

For the first step, $G_5 \rightarrow G_4$, the transformation from initial permutation p to p' regards only the orientation of each edge as relevant to the solve. This implies our reduced problem contains only $2^{11} = 2,048$ unique permutations, hence we conclude the maximum number of permutations stored at a given time is $2,048 + 1$. At this point an overlap is guaranteed between the two distinct sets of permutations via the pigeonhole principle.

For $G_4 \rightarrow G_3$, we consider only two edges. Guaranteed by the previous step to be oriented correctly, we have $12 \cdot 11 = 132$ unique permutations, where the two edges can be in any two

slots. Therefore we require the storage of no more than 132 + 1 or 133 permutations in the worst-case.

For $G_3 \rightarrow G_2$, we first consider the complete mapping as in Section 7.6.1. t For this, we have $10 \cdot 9 \cdot 8 = 720$ locations for the edges in the left-block, $8! = 40,320$ locations for each corner, and $3^2 = 9$ different orientations of the *DL* corners. This is $720 \cdot 40,320 \cdot 9 = 261,273,600$ permutations total.

However, if we instead solve only the left-block through this method, we instead have only $8 \cdot 7 = 56$ permutations of corners, or $720 \cdot 56 \cdot 9 = 362,880$ unique permutations total. It is not as simple as considering $362,881$ as our worst-case though, as only a sixth of sequences to transition p to a permutation with a left-block also result in $p \leftarrow G_2$, as found in Section 7.3. Therefore, we requires the checking of five sixths of unique permutations before a correct sequence is guaranteed to be found. Our overall worst-case then becomes $362,880 + 362,880 \cdot \frac{5}{6} + 1 = 665,281$ permutations.

For the final steps of the method, no problem reductions are performed — no benefits of symmetry are utilised.

7.6.3 Comparison Of Implementations And Reduction Orders

We compare the worst-case of each previous section in Table 7.4.

Step	Permutations	Basic	Optimised	Symmetrical	Best
$G_5 \rightarrow G_4$	4.3231e+19	9,267,595,563,616	1,085,811	2049	2049
$G_4 \rightarrow G_3$	2.0959e+16	22,621	244	133	133
$G_3 \rightarrow G_2$	1.5999e+14	1.5999e+14	217,678,234	665,281	665,281
$G_2 \rightarrow G_1$	73,483,196	73,483,197	354,292	73,483,197	354,292
$G_1 \rightarrow G_0$	4	4	4	4	4

Table 7.4: *The overall worst-case number of permutations stored at a single time.*

From Table 7.4 we see that the overall worst-case number of cubes stored is $665,281$, and that in all cases where the problem is reduced to one of $p' \rightarrow q'$, the resulting decrease in possible permutations is the most significant of the optimisations. Compared to the worst case of the basic implementation, we have a percentage decrease of 99.9999996%.

We now compare the worst-cases for the solving method and the four others distinct orders of removal of rotations in the generating sets, introduced in Section 7.4. To do this, we look at each possible reduction of the minimal generating set not used by the implementation, namely $A \rightarrow B$, $B \rightarrow D$, $D \rightarrow F$, $B \rightarrow E$, $E \rightarrow G$, and $G \rightarrow H$.

For $A \rightarrow B$, we require the solving of only a single edge. This edge can be placed in any slot in either rotation, yielding $12 \cdot 2 = 24$ different permutations, where one is the solution. This requires 25 permutations to be stored in the worst case.

For $B \rightarrow D$, we require the solving of two edges and a corner. The corner is placed freely, the edges in all but one slot, and so we yield a total of $(8 \cdot 3^1) \cdot (11 \cdot 10 \cdot 2^2) + 1 = 10,561$ permutations in the worst case.

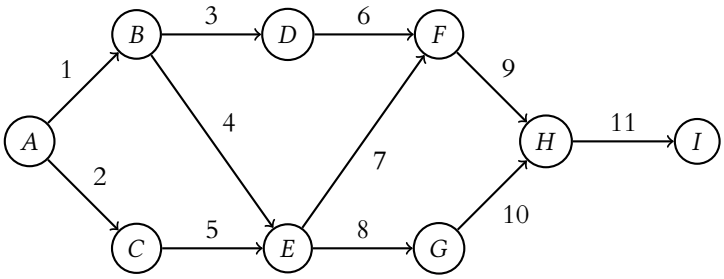
As we have rotation over three axes, and hence no guarantee that edges are oriented edges, $D \rightarrow F$ consists of building a left-block, while permuting corners, while orienting edges. This requires the orientation of all unsolved edges and one corner, and the permutation of two edges and one corner. This is $(9 \cdot 8 \cdot 2^8) \cdot (7! \cdot 3^1) + 1 = 278,691,841$ permutations. Solving without care for the permutation of corners, as implemented for $G_3 \rightarrow G_2$, reduces this to $(9 \cdot 8 \cdot 2^8) + ((7 \cdot 3^1) \cdot \frac{5}{6}) + 1 = 709,633$ permutations. Instead ignoring the orientation of edges

results in $(9 \cdot 8 \cdot 2^2) \cdot (7! \cdot 3^1) = 4,354,560$ permutations and almost all must be stored twice; only 1/64 permutations with a left-block have oriented edges.

For $B \rightarrow E$, we require the orientation of all remaining edges and the solving of one. This is equivalent to $(2^{10}) + 1 = 1,025$ permutations stored in total.

For $E \rightarrow G$, we solve the central band, correctly permuting and orienting all pieces to one of 16 possible configurations. These reductions require all state information, and so no additional benefit arise from symmetry. The optimised algorithm solves $E \rightarrow G$ via the values $r = 9$, $r' = 6$, $r_q = 3$, and $m = 20$ to yield a worst case of 217,678,234, equivalent to the that of $G_3 \rightarrow G_2$. For all other reductions in this section, the equivalent bidirectional calculation yields a greater, and therefore unused, worst-case.

Finally, $G \rightarrow H$ requires no more than three rotations in the worst case, or four cubes total.



Edge	Weight
1	25
2	2,049
3	10,561
4	1,025
5	133
6	709,633
7	665,281
8	217,678,234
9	354,292
10	4
11	4

Figure 7.1: *A directed graph of the potential reduction orders, and their associated weights.*

The directed graph in Figure 7.1 depicts potential orders in which rotations can be removed from the generating sets, through the lettering introduced in Section 7.4. We find the order minimising the worst-case for permutations stored concurrently as the path from $A \rightarrow I$ with the least total weight, done by iteratively removing the edge with the greatest weight and checking if there remains a valid path. Edge 7 is the first which cannot be removed, and hence the worst-case is 665,281 permutations, regardless of reduction order.

Considering the total permutations across an entire execution, we follow the removal process as before, but rather than stopping when the edge of greatest weight cannot be removed, we attempt to remove the edge with the second greatest, then the third, and so on, until no edges can be removed without introducing a cut. At this point, our network will have only five edges- the path with the least weight. This path is $A \rightarrow B \rightarrow E \rightarrow F \rightarrow H \rightarrow I$ and corresponds to ordering two from Section 7.4. It has a combined total of 1,020,627 permutations, as opposed to the implemented order for which the total is 1,021,759- for completeness, ordering one has 1,074,515, ordering two has 217,679,292, and ordering four has 217,680,424. We therefore observe that the implemented order is ranked second amongst the possible reduction orders; in the worst-case, 1,132 more permutations are stored in total, though the same quantity are stored concurrently.

As was the goal of the evaluation, we therefore find that to solve a Rubik's Cube by decreasing the order of the set of rotations required to complete the solve, one face at a time, the most efficient order of removal is ordering two. However, the implemented order is more similar to human-based methods, and easier to comprehend from a single perspective (as results of all steps are visible from observation of any corner to some degree). As such, the implemented order is preferred for explanation of the set reduction concept, regardless of its slight inefficiencies.

8 Conclusion

As explained in Chapter 3, a group, G, can be formed representing the sequences that can be applied to a cube. Utilising the concept of the *minimal generating set* of a group, i.e. a set with the fewest elements such that through a combination of elements and inverses, any element in the group can be constructed, we transition the cube between permutations that can be reached by members of subgroups of G with minimal generating sets iteratively decreasing in size. The group of sequences to generate all permutations has a minimal generating set comprised of quarter rotations of any five of the six faces, and the subgroups of sequences at each stage of the method have a minimal generating set equal to that of the previous stage, but with one less element. Once rotation of the sixth face has been restricted, the only available rotation is the null rotation. As this has no impact on the permutation of a cube, once the minimal generating set contains only this element, the cube must be solved.

Though there are 6! potential orders in which rotations can be restricted to choose from, only the order $B \rightarrow F \rightarrow D \rightarrow L \rightarrow R \rightarrow U$ was implemented. This order was chosen as it is the closest representation of human-based methods, where priority is given to the easiest faces to see and rotate when holding the cube naturally. These are the front and top faces, and the right-side; most cubers prefer rotations that are conducted with their dominant hand.

An initial implementation using breadth first search, although correct, was found to be inefficient. Computational limitations meant the rate of expansion of the search space was quickly unmanageable, and therefore optimisations were required. The optimisations implemented were: an extended set of rotations, increasing the rate of expansion but decreasing the total number of rotations required in the worst-case for a sequence; modification of the adjacency lists through which rotations succeed others, such that simple cases of equivalent sequences are avoided, for example (U, D), (D, U); and a conversion of each stage of the method to a task of finding a sequence between two distinct permutations, permitting the use of bidirectional search. With these techniques, the execution time of the implementation is almost instant.

Alternative approaches for implementation were attempted, such as a recursive version of the proposed solving method, but these performed worse on average during execution. Representing each permutation as a series of bitboards was the fastest implementation, as this allowed bitwise operations to be utilised for rotations of each face. Conversion to C# as part of the development process introduced a series of challenges with the ambiguous representation of each cube through hash-codes. All of these challenges were mitigated through the introduction of a tree-like structure for the storage of cubes. This structure allowed bitboards to be used individually to distinguish between permutations, and for an efficient search and retrieval of permutations and their associated sequences to take place. Also implemented was a visualisation of the method, allowing for interaction with a 3D-Model of a Rubik's Cube through a user-friendly interface. This visualisation was tested in two waves, ultimately concluding that it was fit for purpose; participants conveyed an acceptable level of understanding of the method after only a short amount of use.

After development, an evaluation was conducted and comparisons were made between all major versions of implementation. This evaluation found that while there are 720 orders in which faces can be restricted, see Section 7.4, only five are symmetrically distinct. Regardless of the order

of restriction, only eleven distinct restrictions can occur, see Figure 7.1. Also investigated was the worst-cases for permutations stored concurrently and overall for each of the five orders. The implemented order of restriction stored in the worst-case $665,281$ permutations concurrently, and $1,021,759$ permutations across an entire execution. Compared with the other potential orders of restriction, the implemented order was joint first with regard to minimising the total concurrently stored permutations, and second when minimising the total permutations stored overall. However, as the difference in performance is negligible when compared to the order of restriction requiring the least total permutations, and the proposed method better reflects the block-building of typical human-based methods, it remains preferred for the visualisation.

Future Work. In the future, it is likely that lower upper-bounds for the total permutations stored could be found. These may come as a combination of the evaluation of the search space through symmetry and bidirectional search, as these are disjoint in evaluation but not in practice. Though highlighted in the evaluation, the pruning of branches arising from duplicate permutations is not fully explored. Doing so may yield a lower upper-bound, as the rate of expansion of the search space decreases once sequences become longer – an increased number of duplicate permutations are found from equivalent sequences not avoided through optimisation of the adjacency lists.

The visualisation could be updated to better represent the restriction of faces, by allowing a user to freely choose which face is restricted at each stage. For this, a user may be provided a button for each face of a cube, rather than for each stage of a predefined order of restriction. These could be clicked in any order, allowing the user freedom over which order of restriction is utilised for each solve. Maintaining the button to restart the previous scramble, a user would be able to solve the same permutation via multiple orders of restriction. Instead of implementing unique methods for all possible transitions between subgroups, this functionality could be introduced through eleven distinct methods, each responsible for a single symmetrically distinct reduction. These transitions are the eleven edges of the graph in Figure 7.1, and the subgroups are representations of the vertices A to I in the same figure. After transposing each permutation and generating set to a frame of reference for which methods are implemented to define appropriate starting and target permutations, Algorithm 3 could be used to find a sequence as implemented for the optimised version. The resulting sequence must then be transposed back to the original frame of reference, such that it can applied correctly to the original permutation. Generation of the starting and target permutations can be implemented matching the description for transition along each edge of the graph, as in Section 7.6.3.

The above changes to the visualisation could allow for better comprehension of the underlying concept, as the user is provided much more freedom in exploration of the restriction of rotations. Coupled with an ability to rotate sides directly, a user could see the effects of different restrictions in ways missed by the final product. However, there is potential for confusion, as the lack of a defined order for restriction of faces may hide the underlying concept. It can be useful to visualise restriction by focusing on areas of the cube solved as a byproduct of restriction, and these may be harder to track due to differences in their composition across the potential orders. The interface presented in Figure 6.1 avoids overwhelming the user, as their actions are limited to a clearly defined sequence of operations. Instead of entirely redesigning the interface, allowing the user to toggle between the current implementation and the one described above may be the best solution. This would grant the same level of understanding as the visualisation in its current state does, while also allowing for further exploration of the concept once a user is familiar.

Concluding Remarks. With or without this future work, the dissertation provides an efficient method for solving a Rubik's Cube by restriction of the available rotations. Explanation of an efficient algorithm through which this method is implemented is given, and the associated visualisation is successful in utilising the implementation to communicate the method to a user.

A Appendices

A.1 Instructions to Run the Application

Unzip the downloaded submission. Navigate to the visualisation folder, and run the file *CubeTutorial.exe*. This runs the project.

To view the code for the solver, instead navigate through the directories working-versions\final-version\ Assets\Solver, where the C# files are contained. To run the project in a development environment, install Unity, and open the *final-version* folder. The tests for the implementation can also be run from here.

All previous versions mentioned in the dissertation can be found in working-versions\old-versions.

A.2 Calculations for Evaluation

A.2.1 Naive Worst-Case Total Permutations

This function calculates the worst-case number of permutations found by a naive approach, adding all cubes to CubeStore each time, as in core-permutations.py

```
    m = [0, 1, 20, 20, 4, 11]
r = [0, 3, 6, 9, 12, 15]

for g in range(5, 0, -1):

    permutations = sum(
                        [r[g]   * n
                            for n in range(m[g]+1)]   )

    print("\n", g, "to", (g-1), ": ", permutations)
```

A.2.2 Worst-Case for the Basic Implementation

This function calculates the worst-case number of permutations found by the basic algorithm as in LongestSequence.java. Note that in this implementation of each cube, the *sequence* parameter has been replaced by a single integer representing the number of rotations made to reach the associated permutation, accessed through *Cube.getRotations()*.

```
package main;

import java.util.LinkedList;

public class LongestSequence {

    private int findMaxSequenceLength(int n) {
```

```
    //get new cube
    Cube p = getPn(n);

    //Adjacency List
    LinkedList<Integer> An = getAdjacencyList(n);

    //define a storage for p and add c
    CubeStore cs = new CubeStore();
    cs.add(p);

    System.out.println(An);

    while (true) {
        Cube c = cs.next();

        for (int r: An){
            Cube c_r = c.copy();
            c_r.rotateByMove(r);
            cs.add(c_r);
        }
        if (cs.isEmpty()) return c.getRotations();
    }
}

private Cube getPn(int n) {

    //c -> c'
    return switch (n) {
        case 5 -> new Cube(new int[]{0xF0F0F0F, 0xF0F0F0F, 0, 0, 0xF000F,
        0xF000F});
        case 4 -> new Cube(new int[]{0, 0, 0, 0, 0x900, 0x600});
        case 3 -> new Cube(new int[]{0, 0xA00000AA, 0x33333, 0, 0x99000,
        0x66});
        default -> new Cube(new int[]{0xCCCCCCCC, 0xAAAAAAAA, 0x33333333,
        0x55555555, 0x99999999, 0x66666666});
    };

}

private LinkedList<Integer> getAdjacencyList(int n){

    LinkedList<Integer> An = new LinkedList<>();

    switch(n){
        case 5: //F
            An.add(18);
            An.add(12);
            An.add(6);
        case 4: //D
            An.add(14);
            An.add(8);
```

```
                An.add(2);
            case 3: //L
                An.add(15);
                An.add(9);
                An.add(3);
            case 2: //R
                An.add(16);
                An.add(10);
                An.add(4);
            case 1: //U
                An.add(13);
                An.add(7);
                An.add(1);
        }

        return An;
    }

    public static void main(String[] args) {

        LongestSequence ls = new LongestSequence();
        System.out.println(ls.findMaxSequenceLength(2));

    }
}
```

A.2.3 Worst-Case for the Optimised Implementation

This function calculates the worst-case number of permutations found by bidirectional search, adding all cubes to CubeStore each time, as in total-permutations.py.

```
def calculate_worst_case(r_p, r, r_q, m):

    p_iterations = [1] + [r_p*sum([r**n for n in range(m_p+1)])
    for m_p in range(m)]

    #-r_q as the iterations to form first sequence don't count
    q_iterations = [r_q] + [r_q*sum([r**n for n in range(m_q+1)])-r_q for m_q
    in range(m)]

    m_p, m_q = 0, 0
    pt = True

    while m_p + m_q < m: # -1 as final step of sequence was preprocessed

        if p_iterations[m_p] <= q_iterations[m_q]:
            m_p += 1
            pt = True
        else:
            m_q += 1
            pt = False

    #p', r_q because first sequence step is preprocessed for q in solver
```

```
    implementation (final rotation guarantee)

    # +1 for p', r_q for preprocessing
    worst_case = 2*(p_iterations[m_p] if pt else q_iterations[m_q]) + 1 + r_q


    print(worst_case, m_p, m_q)

print("\nG5 to G4")
calculate_worst_case(15, 12, 2, 11)

print("\nG4 to G3")
calculate_worst_case(12, 9, 3, 4)

print("\nG3 to G2")
calculate_worst_case(9, 6, 3, 20)

print("\nG2 to G1")
calculate_worst_case(6, 3, 3, 20)

print("\nG1 to G0")
calculate_worst_case(0, 1, 3, 1)

print("\nE to G")
calculate_worst_case(9,6,3,20)

print("\nOther G3 to G2")
calculate_worst_case(9,6,2160,20)
```

A.3 Ethics Checklist

The next page contains a signed Ethics Checklist, as required by the University of Glasgow when conducting user testing, as done so in Chapter 7.

School of Computing Science
University of Glasgow

Ethics checklist form for assessed exercises (at all levels)

This form is only applicable for assessed exercises that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, or getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please contact the Chair of the School of Computing Science Ethics Committee (matthew.chalmers@glasgow.ac.uk) for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your assessed work.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.

   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.

   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.

   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.

   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.

12. All the data collected from the participants is stored in an anonymous form.
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Course and Assessment Name **COMPSCI4025P Level 4 Individual Project (Single Hons)**

Student's Name **Joseph Parker**

Student Number **2558907p**

Student's Signature *Joseph Parker*

Date 17/10/2023

A.4 Participant Instructions

The next page contains the participant instructions associated with the User Testing of Chapter 7.

User Testing – Participant Instructions

Joseph Parker - 2558907p

November 17, 2023

1 Overview of Testing

As the main focus of the project is to analyse the mathematics behind the solver, the aim of the user testing is to determine if the visualisation is useful for the understanding of these underlying concepts. The intended understanding from the visualisation is aimed at the cube terminology, rather than any abstracted mathematical features it may have.

You, the participant, will be assigned a group, and asked to complete a series tasks in the order defined in the table. The group you are given will not be linked to your GUID or any other personal identifier, and so your responses and opinions will be entirely anonymous. There are multiple people in each group.

The data recorded will be largely quantitative, though there are qualitative questions also. This will allow the researched to understand both if actions that can be taken on the site are intuitive, as well as if the user is able to understand the concepts beyond just interacting with the correct buttons. The responses will be compiled and analysed by the surveyors without third party influence, and the results will be examined. A conclusion over the success of the change will be drawn from the results.

2 Task Assignment

Participants will be assigned a group, from A-D. These participants will fall under one of four categories, related to their relevant experiences with Rubik's Cubes, and Group Theory.

To ensure participants are selected from all groups, participants will be requested specifically from related groups, such as those who have taken Mathematics Courses at University, and those which are members of the Glasgow University Cubing Society.

This group assignment will be made with relation to the characteristics defined in Table 1.

Relevant Experiences	Cube Experience	No Cube Experience
Group Theory Experience	A	B
No Group Theory Experience	C	D

Table 1: A table depicting the required characteristics for each group assignment

The tasks completion will be moderated to avoid technical issues with the software used, and all participants will complete the task in the same location to avoid the impact of confounding variables where possible. The tasks will be completed in order, with four participants from each section completing the tasks. While it is possible that some tasks may be easier as a result of coming later than others in the process, many tasks can not physically be completed for others. For example, a cube cannot be solved before it is scrambled. For this reason, the task order is predetermined.

3 Task Descriptions

There is no requirement to finish each task. The testing should take no more than twenty minutes. With the purpose of leaving half the time to complete the survey, each tasks is designed to fit comfortably within one minute. If a task is taking significantly longer than this, please move on. The tasks are as follows:

1. View the general information regarding the purpose of the visualiser.
2. Rotate the cube such that you can see the blue, yellow, and red faces, with red on top. After this, rotate it back to a position with yellow on top, and green facing left.
3. Scramble the cube.
4. Reduce the cube to a state that can be found by turning three sides.
5. Restart the previous scramble.
6. Reduce the cube into a position exactly one turn from being solved.
7. Generate a different scramble to the original.
8. Solve the cube.

4 Participant Response

After the tasks have been attempted, please complete and submit the survey in its entirety. This survey is anonymous and will not record your email, though we ask you tell us your group so we can identify any potential bias that could be caused by the relevant past experience you may have. The Survey can be found at the end of this document, along with links to both sites.

5 Provided Documents

This section contains links to the relevant documentation.

- Post-Task Survey

8 Bibliography

[1] C. Animesh. Calculating the number of permutations of the rubik's cube, 2021. URL https://medium.com/@chaitanyaanimesh/calculating-the-number-of-permutations-of-the-rubiks-cube-121066f5f054. Accessed on February 29, 2024.

[2] C. Bruce. A hamiltonian circuit for Rubik's cube, 2012. URL https://bruce.cubing.net/ham333/{R}ubikhamiltonexplanation.html. Accessed on January 24, 2024.

[3] J. Chen. Group theory and the Rubik's cube, 2004. URL https://people.math.harvard.edu/~jjchen/docs/Group%20Theory%20and%20the%20{R}ubik's%20Cube.pdf. Accessed on January 24, 2024.

[4] A. Chuang. Analyzing the Rubik's cube gRoup of vaRious sizes and solution methods, 2021. URL https://math.uchicago.edu/~may/{R}EU2021/{R}EUPapers/Chuang,Alex.pdf. Accessed on January 24, 2024.

[5] C. Cooke. Solving the Rubik's cube using group theory, 2017. URL https://digitalcommons.ric.edu/cgi/viewcontent.cgi?article=1164&context=honors_projects. Accessed on January 24, 2024.

[6] Cubing.co. The devils number - history of the cube mystery, 2022. URL https://cubing.co/blog/devils-algorithm. Accessed on January 24, 2024.

[7] T. Davis. Group theory via Rubik's cube, 2006. URL http://geometer.org/Rubik/group.pdf. Accessed on January 24, 2024.

[8] D. Ferenc. The Rubik's cube color schemes, n.d. URL https://ruwix.com/the-Rubiks-cube/japanese-western-color-schemes/. Accessed on January 24, 2024.

[9] I. P. Gent, K. E. Petrie, and J. Puget. Symmetry in constraint programming, 1 2006. URL https://doi.org/10.1016/s1574-6526(06)80014-3.

[10] M. J. H. Heule. Optimal symmetry breaking for graph problems. *Mathematics in Computer Science*, 13(4):533–548, 5 2019. doi: 10.1007/s11786-019-00397-5. URL https://doi.org/10.1007/s11786-019-00397-5.

[11] D. Holt. minimal generating set of Rubik's cube group, 2016. URL https://math.stackexchange.com/questions/1836898/minimal-generating-set-of-Rubiks-cube-group. Accessed on January 25, 2024.

[12] M. Hordecki. Zz speedcubing system, 2008. URL https://web.archive.org/web/20100503082728/http://www.emsee.110mb.com/Speedcubing/ZZ%20speedcubing%20system.html. Accessed on January 24, 2024.

[13] M. Jakobsen. Minimal move set for Rubik's cube, 2021. URL https://hrjakobsen.medium.com/minimal-move-set-for-the-Rubiks-cube-fbb7855e9983. Accessed on January 25, 2024.

[14] P. Keller. Representation of a chess board with a bitboard, n.d. URL https://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/rep.html. Accessed on February 1, 2024.

[15] Kociemba.org. The two-phase algorithm coordinates, n.d. URL http://kociemba.org/math/twophase.htm. Accessed on January 24, 2024.

[16] J. Li. The mathematics of the Rubik's cube, 2009. URL https://web.mit.edu/sp.268/www/{R}ubik.pdf. Accessed on January 25, 2024.

[17] M. W. Liebeck. *A concise introduction to pure mathematics.* Chapman and Hall/CRC eBooks, 2018.

[18] M. Monge. On perfect hashing of numbers with sparse digit representation via multiplication by a constant. *Discrete Applied Mathematics*, 159(11):1176–1179, 2011. ISSN 0166-218X. doi: https://doi.org/10.1016/j.dam.2011.03.007. URL https://www.sciencedirect.com/science/article/pii/S0166218X11000837.

[19] J. A. Nelder. A simplex method for function minimization. *The Computer Journal*, 7(4): 308–313, 1965.

[20] T. Rokicki. Twenty-five moves suffice for Rubik's cube, 2008. URL http://kociemba.org/math/papers/{R}ubik25.pdf. Accessed on January 24, 2024.

[21] T. Rokicki and M. Davidson. God's number is 26 in the quarter-turn metric, n.d. URL https://www.cube20.org/qtm/. Accessed on January 25, 2024.

[22] Rubiks. You can do the Rubik's cube, 2024. URL https://{R}ubiks.com/en-US/solve-it/. Accessed on January 24, 2024.

[23] Ruwix. Herbert Kociemba's optimal cube solver – cube explorer, n.d. URL https://ruwix.com/the-Rubiks-cube/herbert-kociemba-optimal-cube-solver-cube-explorer/. Accessed on January 24, 2024.

[24] P. S. Segundo, R. Galan, F. Matia, D. Rodriguez Losada, and A. Jimenez. Efficient search using bitboard models. In *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pages 132–138, 2006. doi: 10.1109/ICTAI.2006.53.

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

[26] A. Sood. How to solve a Rubik's cube (beginner's method), 2022. URL https://www.cubelelo.com/blogs/cubing/how-to-solve-a-Rubiks-cube-beginners-method. Accessed on January 24, 2024.

[27] speedsolving.com. General information - permutations, n.d. URL https://www.speedsolving.com/wiki/index.php?title=General_Information#Permutations. Accessed on January 25, 2024.

[28] P. State. The age of the universe | astronomy 801: Planets, stars, galaxies, and the universe, n.d. URL https://www.e-education.psu.edu/astro801/content/l10_p5.html. Accessed on March 17, 2024.

[29] M. B. Thistlethwaite. Thistlethwaite's 52-move algorithm, 1980. URL https://www.jaapsch.net/puzzles/thsistle.htm. Accessed on January 24, 2024.

[30] Time.com. The Rubik's cube: a puzzling success, 2009. URL https://content.time.com/time/subscriber/article/0,33009,1874509,00.html. Accessed on March 17, 2024.

[31] S.-J. Yen, J.-K. Yang, K.-Y. Kao, and T.-N. Yang. Bitboard knowledge base system and elegant search architectures for connect6. *Knowledge-Based Systems*, 34:43–54, 2012. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2012.05.002. URL https://www.sciencedirect.com/science/article/pii/S0950705112001293. A Special Issue on Artificial Intelligence in Computer Games: AICG.

[32] F. Zemdegs. Coll algorithms (corners and orientation of last layer), 2017. URL https://www.cubeskills.com/uploads/pdf/tutorials/coll-algorithms.pdf. Accessed on January 24, 2024.

[33] F. Zemdegs. Pll algorithms (permutation of last layer), 2017. URL https://www.cubeskills.com/uploads/pdf/tutorials/pll-algorithms.pdf. Accessed on January 24, 2024.

[34] G. Zussman. SpeedCubeDB - zbll, n.d. URL https://www.speedcubedb.com/a/ZBLL. Accessed on January 24, 2024.