Algorithmic Approaches to Solving a Rubik's Cube

Robert Espinoza

October 4, 2020

Abstract

The Rubik's Cube is a puzzle that has fascinated many for decades. There is a rich foundation of mathematics and computer science surrounding a Rubik's Cube. This paper seeks to explore the various facets of these topics. Basic cube notations, search algorithms, and mathematics are set up to ultimately explore the performances of the different algorithms developed by Herbert Kociemba and Morwen Thistlethwaite.

Foreword

It is important to note that these topics, and all of its efforts included, are a complete passion project, and written to document the things I've learned and methods I've used to implement these programs. There has been no institutional guidance, and certainly no academic or professional guidance. Thus, there must be a forewarning that I make no claims to be an expert on any of these subject matters. I have taken great care to attempt to make this paper accessible to any reader, and an appreciable effort has been made to not appear pompous to the average reader, nor arrogant and misguided to any subject matter expert. I ask for grace in any facet of my efforts, from any reader whom I'm ever so grateful for giving their time of day to read any of my writing. I make no claims for any of this subject matter to be innovative or revolutionary, and in fact credit much of the work I've done to be directly inspired, and greatly aided by, the work of many before. I will attempt to give credit where credit is due, but for any piece I do not properly attribute, I apologize beforehand and emphasize that any of this is the product of countless hours of research and debugging, for which would not be made possible without the work of previous subject matter experts.

Thank you to Le Thanh Hoang for necessary guidance in writing these papers and understanding a lot of the theory, Morwen Thistlethwaite and Herbert Kociemba for their algorithms and detailed explanations and implementations, and Jaap Scherphuis and Stefan Pochmann for their invaluable information documenting and implementing all things Rubik's Cubes.

Contents

Chapter 1

Introduction

This following paper seeks to establish the notations and different techniques of transforming one cube state into another desired cube state. It will explore the different algorithms developed to do these things, as well as walk through a few sample implementations.

First, mathematical definitions are established. Explanations of the notations and definitions of the Rubik's Cube are explored as well. Then, different algorithms previously developed by mathematicians are defined with sample implementations. Lastly, the performances of these algorithms are compared.

The reader should be able to use this paper as a supplementary guide to the true source materials should they attempt to embark on the journey of developing a cube solver program from scratch.

Chapter 2

Background

2.1 Rubik's Cube Notation

Understanding of the simple terminology and notation of a Rubik's Cube helps the reader visualize the topics better.

2.1.1 Faces

There are 6 faces on any 3x3 Rubik's Cube, or any cube shaped object for that matter. On a standard-colored solved cube, these faces correspond to the white, green, blue, red, yellow, and orange colors. The faces can alternatively be referred to as the right face(R), left face(L), front face(F), back face(B), up face(U), and down face(D). This naming scheme allows an individual to abstract a cube to locations of the center colors, which can never be moved by any face rotation moves.

2.1.2 Facelets

There are 54 facelets on a standard 3x3 Rubik's Cube. These facelets correspond to what one would call stickers; there are 6 facelets for every face's color.

2.1.3 Cubies

On a standard 3x3 Rubik's Cube, there are 26 cubies. A cubie refers to the smaller cubes which make up the whole cube. A cubie can either be an edge cubie, or a corner cubie. A corner cubie is any cubie that is located on a corner, and an edge cubie is everything else. An entire cube is made of 12 edge cubies, 8 corner cubies,

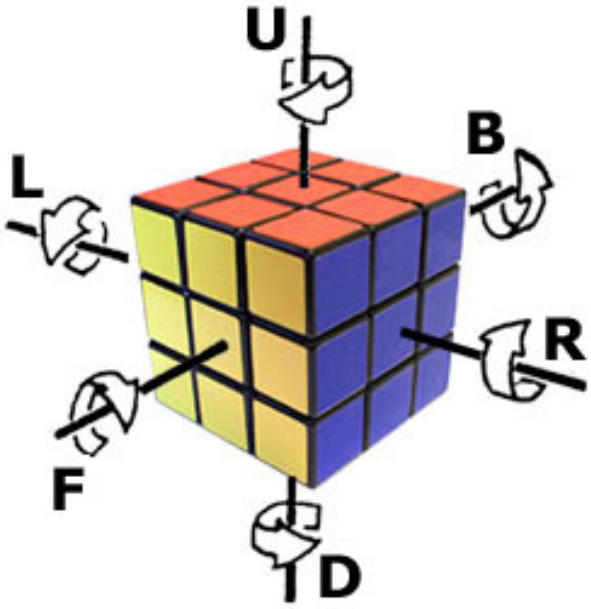

Figure 2.1: Rubik's Cube move faces. [1]

and 6 corner cubies. The center cubies cannot be moved; every edge and corner cubie that is located on the face being rotated by a move swivels around the center cubie.

2.1.4 Naming Conventions

Figure 2.2 details the naming conventions when referring to specific facelets or cubies. The center facelets represent the face's name. A specific corner cubie can be referred to as the UFR corner. These naming conventions can also refer to abstract locations of the cube; we can say the the DBR corner currently occupies the UFR corner location, or that the UFR corner "is replaced by" the DBR corner.

2.1.5 Move Operations

Move operations are at the central core of a Rubik's Cube. Any possible cube state can be reachable by some combination of the 6 move operations. These move operations correspond to the rotation of faces. Before concrete definitions can be understood, one must understand the notation of faces and face rotations. For the purposes of understanding the terminology for the rest of this paper, or any Rubik's Cube related writing, it is imperative that one stays consistent with their Rubik's

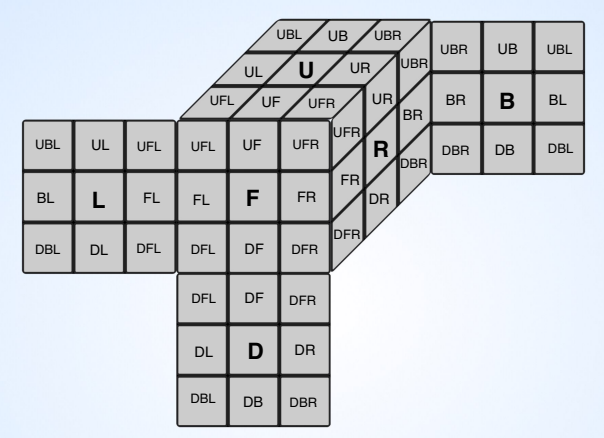

Figure 2.2: Rubik's Cube Naming Conventions

Cube face labeling, and how they label faces with relation to the center cubies. One can define their representation of a Rubik's Cube with the front face's center cubie being the blue center cubie, and the upper face's center cubie being the yellow cubie. From there on, any references to faces must stay consistent, even if we perform a rotation of the cube only for inspection. Regardless of color, the F face always stays the F face, the U face always stays the U face, etc.

With the concrete definitions of a cube's faces in place, we can now better understand move operations. The 6 move operations of a cube corresponds to the rotations of the 6 faces, which we call quarter turns. A counterclockwise turn of the face corresponds to the inverse of a move. As well, half turns are made up of 2 quarter turns. An illustration of these moves makes it easier to understand.

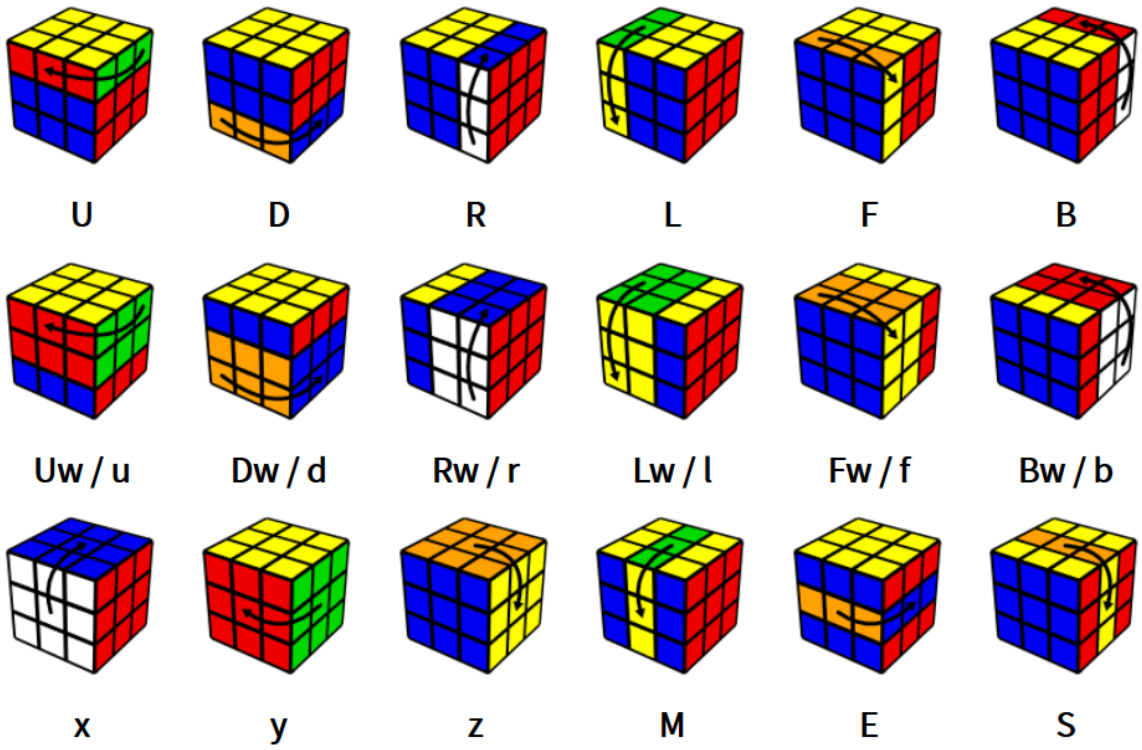

Figure 2.3: Rubik's Cube move operations [2]

U', or U3, corresponds to the move of U in the opposite direction. This applies to the 5 other moves.

2.2 Mathematics

2.2.1 Group Theory

The Rubik's Cube is deeply entrenched in a branch of mathematics called group theory. A group can represent the structure of a Rubik's Cube. In order to define the defintion of a group, it is necessary to define a few terms beforehand.

A ***set*** is a well-defined collection of ***distinct*** objects [wikipedia sets]. A set can be made up of the distinct objects 1,2,3, which, when together, form the set {1,2,3}. This set is equivalent to the sets {2,1,3},{3,2,1},{3,1,2} [3].

A **binary operation** is a calculation that combines **two** elements to produce another element [4].

The Definition of a Group [5, 6]

A group is a set, G, together with an operation, *, that combines any two elements a and b, to form another element, denoted $a*b$, or ab. To qualify as a group, the set and operation $(G,*)$, must satisfy the following four group axioms:
Closure - For all a,b in G, $a*b$ is also in G.
Associativity - For all a,b and c in G, $(a*b)*c = a*(b*c)$.
Identity Element - There exists an element e in G such that, for every element a in G, $a*e = e*a = a$. This element is also unique.
Inverse Element - For each element a in G, there exists an element b in G, such that $a*b=b*a=e$, where e is the identity element.

The set of integers with the binary operation addition together form a group. The group axioms are satisfied as such:
Closure - Any integer a,b summed together, $a+b$, will always yield an integer, thus this group is closed under addition.
Associativity - Any integers a,b,c will yield the same sum grouped in any way; $(a+b)+c = a+(b+c)$ always gives the same result.
Identity Element - For any integer a, $0+a=a+0=a$. We denote 0 as e, the identity element, which is unique.
Inverse Element - Every integer a has an inverse element b such that $a+b=e$, where e is zero, and b is denoted as $-a$.

The Rubik's Cube Group

The notion of a group can be extended to a Rubik's Cube. This group can be referred to as $(G,*)$. This group, $(G,*)$, is the set of all cube moves, defined as any rotation of the cube's faces, with the group operation *, which is defined as the composition of cube moves. Any element in the set G is the effect of any sequence of rotations of cube faces.

This means that any permutation of the cube can be defined as a sequence of cube moves. The available cube moves that generate all possible valid Rubik's Cube states are the moves R,L,F,B,U,D. Any cube move can be represented by the sequence of moves, and any cube position can be represented by the sequence of moves applied in order from the solved state. Starting from the solved state, there is a one-to-one correspondence between any valid Rubik's Cube state, and the elements of G, a sequence of move operations. It is important to reiterate that the elements of the group G are move sequences. As well, move sequences that have the same result are

considered the same element. The binary operation, * , is the concatenation of any two move sequences that preserve the ordering of moves.

The four group axioms can be satisfied as follows:

Closure - For all *a,b* in *G*, *a*b* is also in *G*.

Any sequence of valid, legal moves *a*, followed by any sequence of valid, legal moves *b*, will always generate an element *a*b* that is in *G*. Any move in *G* applied to any state in *G* will yield a valid, legal state that is in *G* as well. Thus, the Rubik's Cube group is closed under the binary operation ***. For example, if *a* was the move sequence *R*F*U*, *b* the move sequence *B*D*R*, *a*b* = *R*F*U*B*D*R* is a valid move sequence, and the state resulting from applying this move sequence to a solved state is always a valid state in *G*.

Associativity - For all *a,b* and *c* in *G*, *(a*b)*c = a*(b*c)*.

Any sequence of moves concatenated on to one another always yields the same sum regardless of grouping, so long as the order of the sequence of moves is preserved. If *a=R*F,b=F*U,c=D*U*, *(R*F*F*U)*(D*U) = R*F*F*U*D*U = (R*F)*(F*U*D*U)* all yield the same move sequences, and the same state when applied to a solved cube.

Identity Element - There exists an element *e* in *G* such that, for every element *a* in *G*, *a*e = e*a = a*. This element is also unique.

For the Rubik's Cube group, this identity element is defined as performing no cube move, or the empty cube move. No set of moves can allow an element in *G* to be itself. Though one may argue that the move sequence *F*F*F*F*, or any quarter move operation performed 4 times in a row yields the same element, any move operations that yield the same result are considered the same element. Thus, *F*F*F*F*, along with similar move sequences, are identical to the empty move, and unique. The identity can also be considered the solved cube state, since no moves are applied to it.

Inverse Element - For each element *a* in *G*, there exists an element *b* in *G*, such that *a*b=b*a=e*, where *e* is the identity element.

It is clear to see that any element in *G* has an inverse element, that simply corresponds to its move sequence's inverses in reverse. A single move, *F*, has an inverse defined as performing *F*F*F*, more succinctly wrote as *F3*, or *F'*. Another move sequence, *R*F*B*D*U* has an inverse of *U3*D3*B3*F3*R3*, which is equal to the identity element, or performing no move at all. The inverse of any element in *G*, a move sequence, is the same move sequence's individual inverse moves applied in reverse.

The Rubik's Cube group is also ***non-abelian***. In other words, the composition of cube moves is not commutative. Applying move sequences out of order does not

yield the same result. The cube state after applying $R*F$ to a solved cube is not the same cube state we get when applying $F*R$ to a solved cube.

2.2.2 Problem Space

The problem space of a Rubik's Cube is mindboggingly large. We can calculate the problem space of a Rubik's Cube quite simply with a few observations.

There are $8! = 40,320$ different ways that we can permute a cube's corner locations.

There are $12! = 479,001,600$ different ways that we can permute a cube's edge locations.

There are $3^7 = 2187$ different orientations that a cube's corners can have.

There are $2^{11} = 2048$ different orientations that a cube's edges can have.

Only half of these states are reachable, since all valid permutations must be even, thus the number of states a cube can have is
$8! * 12! * 3^7 * 2^{11}/2 = 43,252,003,274,489,856,000$
Clearly, one needs to be methodical when solving a Rubik's Cube.

2.2.3 Unique Numbering for Permutations and Combinations

In mathematics, a permutation of a set is an arrangement of its members into a sequence or linear order, or if the set is already ordered, a rearrangement of its elements [7]. In regards to the Rubik's Cube, any state can be represented by a permutation; tracking the edges, corners, and their orientations with regards to their locations. A combination is a selection of items from a collection, such that, unlike permutations, the order of selection does not matter [8]. Representing a combination with regards to a Rubik's Cube, however, is a bit more difficult, and we will delve in deeper later in this paper(see section on 'Representing a Rubik's Cube'). For now, it is important to know that any permutation or combination can be represented by a unique integer. All permutations for a set of integers [0,1,2,3,4,5,6,7] take up a space of $8! = 40320$; i.e., there are 40,320 different permutations for this set. By applying a few tricks in reorganizing the permutation and using a factorial numbering

scheme, we can uniquely map each possible permutation to one of these integers (to be exemplified later). The concept is the same for combinations, again to be exemplified later. The main takeaway is that we can uniquely assign each permutation and combination a value, which saves on memory and indexing costs.

2.2.4 Theorems for the Rubik's Cube

Without going too much into proofs, we now describe some theorems for the Rubik's Cube that will make implementations easier. First, we state that the permutations of the corner cubes and edge cubes are either both even or both odd [9]. We can exploit this when determining overall parity, in that we only need to calculate either corner permutation parity or edge permutation parity, but not necessarily both.

As well, all edge flips and corner flips are even [6]. This is made evident by observing any valid move operation. Any one of these face turns will swap four corners and four edges. Any combination of move operations will perform a multiple of four corner swaps and four edge swaps.

Additionally, the sum of all corner orientations must be divisible by 3, and the sum of all edge orientations must be divisible by 2 [10]. These orientation sum properties allow us to be only concerned with 7 of the 8 corner orientations at once, and 11 of the 12 edge orientations at once. We can deduce the final cubie orientation from the previous ones.

Though not important for implementation's sake, it has been proven that the maximum number of moves to solve any cube state is 20 moves [11]. This gives a lower bound for any algorithm's performance.

2.3 Representing a Rubik's Cube

There are many ways to represent a Rubik's Cube. 54 different items representing each facelet, or a 6x9 matrix representing the 9 stickers on each of the 6 faces are among the different ways to represent a Rubik's Cube. For the remainder of this paper, we will represent the Rubik's Cube using a combination of the permutations and orientations of the edges and corners. In representing the cube this way, our attention shifts focus on only the 12 edges and 8 corners that make up the movable parts of the cube.

2.3.1 Orientations

An edge cubie or a corner cubie can have multiple orientations. An edge that is in its home position, but flipped, has an orientation different than if it were in its home position and solved. A corner cubie can be one of three different orientations. The way that each of these orientations is defined is left to the user, as different orientation definitions suit different needs in some cases.

For the most part, an edge can have two orientations: good or bad, represented by 0 or 1. One may define an edge orientation as good if it can be brought to its original position using an even amount of U or D turns, and bad otherwise.

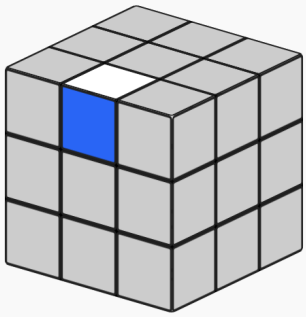

Figure 2.4: Edge(a)

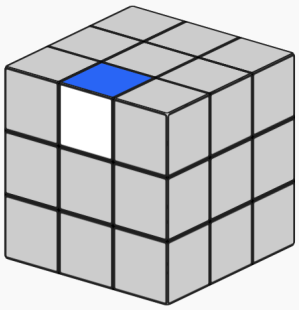

Figure 2.5: Edge(b)

A corner cubie can have three orientations, as it can be rotated in place in three different ways. One may define a corner cubie as 0 if the U or D sticker is facing a U or D face, 1 if this U or D sticker is turned clockwise from the U or D face,

and 2 if this U or D sticker is turned counterclockwise from the U or D face. Again, this definition can change depending on implementation; what is important is that definitions stay consistent within implementations.

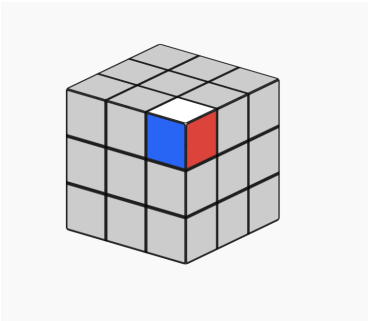

Figure 2.6: Corner(a)

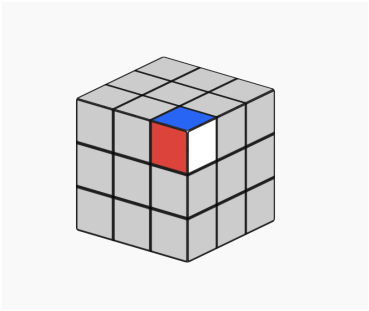

Figure 2.7: Corner(b)

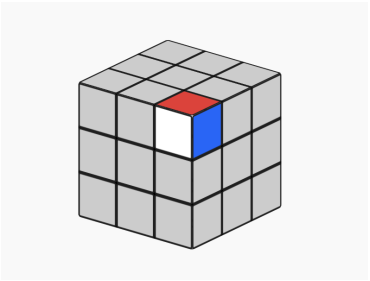

Figure 2.8: Corner(c)

2.3.2 Permutations

When representing a Rubik's Cube with permutations, it is necessary to understand the abstraction that takes place. In essence, the non-physical indices in permutations represent the locations of a Rubik's Cube. In our implementations, permutations that represent a Rubik's Cube are strictly written in the sense of "is replaced by" for each location of a cubie. For example, take a representation of the location of the corners:

$$[\ 0-1-2-3-4-5-6-7\ ]$$

These numbers represent the locations of the corner cubies in their fixed abstract positions. If we define the UBR corner cubie to have index 0, we must be consistent in our naming scheme. One potential way we can define this abstraction is as such:

$$[\ UBR-UFR-UFL-UBL-DBR-DFR-DFL-DBL\ ]$$

Now, with our abstract corner cube locations defined, we can begin representing corner cubie states. The scheme we follow is, again, the "is replaced by" scheme. The value in any index is defined as: "The cubie in this physical location index is replaced by this cubie". An example is as follows:

$$[\ UBR-UFR-UFL-UBL-DBR-DFR-DFL-DBL\ ]$$

$$[\ UFR-UBR-DBL-UBL-DBR-DFL-DFR-UFL\ ]$$

The first array of corners represent the indices for which the cubies are home, and the second array represents the current cubies occupying that space. The location at corner UBR is replaced by corner cubie UFR, the location at corner UFR is replaced by corner cubie UBR, etc. The enumerated indices can be defined as follows:

$$[\ 0-1-2-3-4-5-6-7\ ]$$

$$[\ 1-0-7-3-4-6-5-2\ ]$$

Again, this is merely an example. The definitions are completely arbitrary and can be defined as needs see fit. However, it is absolutely necessary to remain consistent. In representing the entire cube, one can represent it with a 20 element array as such:

$$[\ 0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19\ ]$$

Where the first 12 represent the edges, and the last 8 represent the corners. As well, to be more intuitive, one can assign edges to be a number 0-7 instead of 12-19. If one were to include orientations as their own entity, an extra 20 elements can represent them. Thus, a possible cube representation can be a 40 element array, where the first 12 are edge locations, the next 8 are corner locations, the next 12 are edge orientations, and the last 8 are corner orientations.

One representation of a solved cube state can be as follows:

$$[\ 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,$$

$$0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\ ]$$

Every cubie is in their respective homes, and all orientations are zero.

2.3.3 Move Operations

With the permutations and orientations defined, we can now explore different ways move operations can be defined. In our implementations, move operations are defined by the permutation indices they swap. Irrespective of the current cubie locations at any location of the cube, the move operations swap the cubies occupying the spaces in the same way each operation.

For example, if we define the corner cubies as such:

$$[UBR-UFR-UFL-UBL-DBR-DFR-DFL-DBL]$$

Then the R operation will always swap the indices 0,1,4,5 according to the new space it will occupy. In this case, index 0 is replaced by the cubie located at index 1, index

4 is replaced by the cubie located at index 0, index 5 is replaced by the cubie located at index 4, and index 1 is replaced by the cubie located at index 5. One must also handle the orientation changes, but the rest of the indices are unaffected by this move. With orientations, the new orientation must match the new orientation of the cubie now occupying whatever space it was moved to. Every move does not every every orientation, so one must be careful.

This concept can be applied to any move on the Rubik's Cube. The move operations should ultimately define the new states of a cube after applying the move.

2.3.4 Unique Numbering

Since there are a finite amount of permutations and orientations, we can uniquely assign an integer to each permutation representing the orientations, permutations, or combinations of states. This is especially helpful when implementing minimal perfect hashes for our later implementations. We implement these unique numbering as Herbert Kociemba defines in his algorithm's implementation. This process of numbering permutations can be referred to as Lehmer's Code.

Orientations

All permutations of the orientations of a cube's state can be represented using a ternary or binary number system. Examples for corner and edge orientation permutation numbering can be as follows; the o:2 labeling refers to the orientation of that cubie with respect to the location on the cube:

$$[\ UFR\ UFL\ UBL\ UBR\ DFR\ DFL\ DBL\ DBR\ ]$$

$$[\ DFR; o:2\ UFL; o:0\ UBL; o:0\ UFR; o:1\ DBR; o:1\ DFL; o:0\ DBL; o:2\ UBR; o:2\ ]$$

$$2 * 3^6 + 0 * 3^5 + 0 * 3^4 + 1 * 3^3 + 1 * 3^2 + 0 * 3^1 + 0 * 3^0 = 1494$$

For our corner orientations, the number of any state will be from 0-2186. $3^7 = 2187$ total states are mapped sequentially. Below is a C implementation of uniquely numbering the orientations of a cube's current state. The process for edge orientation numbering is done in a similar way, however, with a binary number system. A C implementation is as follows:

Permutations

In order to sequentially order permutations, we must define a natural ordering on the index locations. This can be done in the same way we define the abstract cube. For brevity, attached is Kociemba's definition of such procedure. In this documentation, Kociemba defines his ordering of corners. As well, his implementation uses the "is replaced by" notation for permutations. This concept is easily extended to edge permutations. All 12 permutations can be sequentially numbered from 0 to 479001600 (12!-1).

The definition of the corner permutation coordinate

The corner permutation coordinate is given by a number from 0 to 40319 (8! - 1).

In this example, we use the permutation of the R-move again, but we ignore the orientations now.

URF	UFL	ULB	UBR	DFR	DLF	DBL	DRB
c:DFR	c:UFL	c:ULB	c:URF	c:DRB	c:DLF	c:DBL	c:UBR
	1	1	3	0	1	1	4

We define a natural order on the corners by URF<UFL<ULB<UBR<DFR<DLF<DBL<DRB.

The number in the third row - below a corner XXX in the second row - gives the number of all corners left of XXX, whose orders are higher than the order of XXX.

Above the entry 4 we have for example the corner UBR.
From the 7 corners left of UBR, 4 corners have a higher order - DFR, DLF, DBL, DRB.

Above the entry 1 we have for example the corner DLF.
From the 5 corners left of DLF, only 1 corner has a higher order - DRB.

We build the permutation coordinate with the numbers of the third row.

1*1! + 1*2! + 3*3! + 0*4! + 1*5! + 1*6! + 4*7! = 21021

Figure 2.9: Kociemba Permutation Numbering

Combinations

Assigning unique numbers to combinations proves to be a bit more difficult, but is doable with some tricky manuevering of indices according to their locations for calculations. For succinctness and correctness, again attached is Kociemba's explanation of such a combination numbering scheme, as well as a sample C++ implementation.

Calculating ES-Slice combination coordinate [10]

```
/* Calculates coordinate of ES slices */
/* size 8c4 (70) */
int ES_coord(vector<char> cube)
    int occupied2[12] = {0};
    // modify here for corner permutation locations
    for(int i = 0; i < 12; i++)
        if( cube[i] == 8 || cube[i] == 9 || cube[i] == 10 || cube[i]
            == 11 )
            occupied2[i] = 1;
    int occupied[8] = {0};
    // 1,3,5,7 edges are already fixed
    // we seek to fix 8,9,10,11
    // 0,2,4,6 get fixed as a result
    occupied[0] = occupied2[0];
    occupied[1] = occupied2[2];
    occupied[2] = occupied2[4];
    occupied[3] = occupied2[6];
    occupied[4] = occupied2[8];
    occupied[5] = occupied2[9];
    occupied[6] = occupied2[10];
    occupied[7] = occupied2[11];
    // calculate combination number
    int k = 3, result=0, n=7;
    while( k >= 0)
        if( occupied[n] )
            k--;
        else
            result += nCr(n, k);
        n--;
    return result;
```

The problem reduces to the following: There are 12 positions, 4 of them are occupied. Assign a unique number c from 0...494 to each possible configuration.

In Cube Explorer we have the following assignment between the position numbers and the edges:

0	1	2	3	4	5	6	7	8	9	10	11
UR	UF	UL	UB	DR	DF	DL	DB	FR	FL	BL	BR

If the UD-Slice edges are in their home positions, the UDSlice coordinate is 0.

Here are a few examples:

0	1	2	3	4	5	6	7	8	9	10	11	
								x	x	x	x	c = 0

0	1	2	3	4	5	6	7	8	9	10	11	
							x		x	x	x	c=1
								C(8,0)				

0	1	2	3	4	5	6	7	8	9	10	11	
			x			x			x		x	c = 62
				C(4,0)	C(5,0)		C(7,1)	C(8,1)		C(10,2)		

0	1	2	3	4	5	6	7	8	9	10	11	
	x			x				x	x			c = 305
		C(2,0)	C(3,0)		C(5,1)	C(6,1)	C(7,1)			C(10,3)	C(11,3)	

0	1	2	3	4	5	6	7	8	9	10	11	
x	x	x	x									c = 494
				C(4,3)	C(5,3)	C(6,3)	C(7,3)	C(8,3)	C(9,3)	C(10,3)	C(11,3)	

The binomial coefficients C(n,k) written under the free places add up to the coordinate. n is the position, k depends on how much occupied places are to the right of the position. The maximal value for k is 3, each occupied place to the right reduces k by one. The minimal value for k is 0, so the free places at the leftmost positions do not count.

C(2,0) + C(3,0) + C(5,1) + C(6,1) + C(7,1) + C(10,3) + C(11,3) =

1 + 1 + 5 + 6 + 7 + 120 + 165 = 305

Figure 2.10: Kociemba Combination Numbering

If we are careful, we can generalize this approach for any combination of cubies. If our goal is to find the number of ways we can choose 3 specific indices from 8 possibilities (8 choose 3), we can manipulate the diagram below and apply the same algorithm to obtain a unique sequential numbering of every combination. The generalized method is as follows: From the current cube state, extract the indices of the relevant cubies we wish to choose from. Create a new array, such that our selection possibilities are the last indices, and any other index is arbitrarily placed before. We can then calculate the combination as Kociemba does to obtain a sequential unique numbering of every combination. Kociemba's definitions and the C++ implementation is as follows:

```
/* Calculates coordinate of ES slices */
/* size 8c4 (70) */
int ES_coord(vector<char> cube)
    int occupied2[12] = {0};
    // modify here for corner permutation locations
    for(int i = 0; i < 12; i++)
        if( cube[i] == 8 || cube[i] == 9 || cube[i] == 10 || cube[i]
            == 11 )
            occupied2[i] = 1;
    int occupied[8] = {0};
    // 1,3,5,7 edges are already fixed
    // we seek to fix 8,9,10,11
    // 0,2,4,6 get fixed as a result
    occupied[0] = occupied2[0];
    occupied[1] = occupied2[2];
    occupied[2] = occupied2[4];
    occupied[3] = occupied2[6];
    occupied[4] = occupied2[8];
    occupied[5] = occupied2[9];
    occupied[6] = occupied2[10];
    occupied[7] = occupied2[11];
    // calculate combination number
    int k = 3, result=0, n=7;
    while( k >= 0)
        if( occupied[n] )
            k--;
        else
            result += nCr(n, k);
        n--;
    return result;
```

Cartesian Product Numbering Scheme [6]

At this point, we have developed a number of ways of representing a cube state and sequentially assigning integers to the different permutations of these states. We can represent a cube's locations and orientations, and calculate the numbers associated with any characteristic we want to extract (orientations, combinations of different selections, etc). If we wish to combine these coordinate integers, we can convert these several coordinates into a single integer using a cartesian product numbering scheme. For example, if we wish to define a coordinate that encapsulates the edge orientations and the ways we can choose 4 specific cubies from 12 selections, we can combine them into a single coordinate. The first coordinate must be a number from 0 to 2047 ($2^{11} = 2048$). The second coordinate must be a number from 0 to 494 ($12 choose 4 = 495$). We can combine them into a single number as follows:

$$NewCoordinate = (coordinate_1) * sizeofCoordinate2 + coordinate_2$$

This concept can be extended to any amount of coordinate combinations, as long as the total space of coordinates is known.

2.3.5 Sample Implementation

What follows is one potential way of implementing a working Rubik's Cube computer program. There are a number of implementing a working representation of a cube; the most important thing is to be able to get the state of any cube, and to be able to modify the state of any cube for any move combination. Tracking cubie permutations and orientations resulting from applying any combination of the 6 possible move operations is the goal.

One possible implementation is to define a cube such that an edge has orientation 0 if in bringing it back to its original place, an even amount of U or D turns is required. Otherwise, it has an orientation of 1. The corner orientations can be defined as having an orientation of 0 if the L or R sticker is on an L or R face, 1 if the L or R sticker is twisted clockwise from the closest L or R face, and 2 if twisted clockwise.

```
char UR = 0;
char UF = 1;
char UL = 2;
char UB = 3;
char DR = 4;
char DF = 5;
char DL = 6;
char DB = 7;
char FR = 8;
char FL = 9;
char BL = 10;
char BR = 11;
char UBR = 0;
char UFR = 1;
char UFL = 2;
char UBL = 3;
char DBR = 4;
char DFR = 5;
char DFL = 6;
char DBL = 7;

// Defining locations of solved cubies
vector<char> cube = {UR,UF,UL,UB,DR,DF,DL,DB,FR,FL,BL,BR,
                     UBR,UFR,UFL,UBL,DBR,DFR,DFL,DBL,
                     0,0,0,0,0,0,0,0,0,0,0,0,
                     0,0,0,0,0,0,0,0};

// Example move operation, for move operation U
if( move == 4) // U move
{
    vector<char> result(cube);
    // +1/+2 to relevant corner orientations
    // affected corners: 0,1,2,3
    // +1 to edge orientations
    // affected edges: 0,1,2,3
    // edge permutations
    result[0] = cube[3];
    result[1] = cube[0];
    result[2] = cube[1];
    result[3] = cube[2];
    // edge orientations
    result[0+20] = (cube[3+20]+1)%2;
    result[1+20] = (cube[0+20]+1)%2;
    result[2+20] = (cube[1+20]+1)%2;
    result[3+20] = (cube[2+20]+1)%2;
    //corner permutations
```

```
    result[0+12] = cube[3+12];
    result[1+12] = cube[0+12];
    result[2+12] = cube[1+12];
    result[3+12] = cube[2+12];
    //corner orientations
    result[0+32] = (cube[3+32]+2)%3;
    result[1+32] = (cube[0+32]+1)%3;
    result[2+32] = (cube[1+32]+2)%3;
    result[3+32] = (cube[2+32]+1)%3;
}
// Performing U move operation on solved cube
vector<char> new_cube = modify_cube(cube,4);
/*
12 edges:              3 0 1 2 4 5 6 7 8 9 10 11
8 corners:             3 0 1 2 4 5 6 7
12 edge orientations:  1 1 1 1 0 0 0 0 0 0 0 0
8 corner orientations: 2 1 2 1 0 0 0 0
--->
new_cube = {3,0,1,2,4,5,6,7,8,9,10,11,
            3,0,1,2,4,5,6,7,
            1,1,1,1,0,0,0,0,0,0,0,0,
            2,1,2,1,0,0,0,0};
*/
```

Again, this is just an example, and cube definitions fit the problem they are trying to solve. From here, we can easily calculate relevant coordinates, since we know the permutations of the locations of edges and corners, as well as the orientations of these permutations.

2.4 Graph Search Algorithms

The search algorithms in this section are presented with the goal of searching for cube states in mind. With a working cube representation, move operations, and coordinate calculations from the preceding chapter, we have the tools to be able to search for cube states from any starting cube state.

We can frame the problem of searching for Rubik's Cube solutions as a graph problem. The starting node is whatever state we start with, and the goal node is whatever goal we're trying to reach, in most cases the solved cube. The branches of any node are the available moves, connecting the node to the state resulting from the move of any branch.

It is easy to see that the branching factor of this graph problem becomes large extremely quickly; a branching factor of 18 (all available moves) has a size of $18^6 = 34,012,224$ at depth 6. Methodical approaches are required to have the solution found quickly. With this problem statement in mind, we can now explore the various ways of finding paths from the starting state to the goal state.

2.4.1 Breadth First Search

The first realistic algorithm we explore is a breadth first search algorithm. This algorithm starts at the source cube, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level [12]. Breadth first search is very useful in proving whether a solution is optimal or not; it will always find the shortest solution. However, with the branching factor of the Rubik's Cube between states, the memory consumption of this problem is a major problem. Below is a C++ pseudolike implementation of this algorithm. It seeks to reduce redundancy by tracking visited nodes.

```
int bfs( source, target )
    q.push( source )
    visited[ source ] = true
    while( q.empty() )
        old_cube = q.front()
        q.pop()
        for( move = 0; move < moveset.size(); move++ )
            new_cube = modify_cube( old_cube, move )
            if new_cube == target
                SOLUTION FOUND, REBUILD PATH ACCORDINGLY
                return true
            if unvisited[ new_cube ]
                q.push( new_cube )
```

```
                visited[ new_cube ] = true
    return false
```

2.4.2 Bidirectional Breadth First Search

The well defined start and end states of the Rubik's Cube problem allows for a bidirectional search to be valid. This bidirectional search can cut the branching factor considerably, since we explore lower depths at a slower rate than a standard breadth first search. The pseudolike C++ implementation included below accomplishes this by tracking the states seen coming from any direction. If we encounter a state that has been identified as having come from the opposite direction, the solution has been found at that intersection. Rebuilding the path accordingly corresponds to rebuilding the path from the start to intersection, and from the intersection to the goal, accounting for adding the inverse of a move to the path.

```
int bdbfs( source, target )
    q.push( source )
    q.push( target )
    visited[ source ] = forward
    visited[ target ] = backward
    unvisited[ source ] = false
    unvisited[ target ] = false
    while( q.empty() )
        current_direction = visited[ q.front() ]
        q.pop()
        for( move = 0; move < moveset.size(); move++ )
            newstate = modify_cube( q.front(), move )
            if(visited[ newstate ] == opposite of current_direction)
                SOLUTION FOUND, REBUILD PATH ACCORDINGLY
                return true
            if unvisited[ newstate ]
                q.push( newstate )
                unvisited[ newstate ] = false
                visited[ newstate ] = current_direction
    return false
```

2.4.3 Iterative Deepening Depth First Search

Iterative deepening depth first search (IDDFS) combines the optimality of breadth first search, with the memory saving features of depth first search. It performs a depth limited search for each depth up to a specified maximum depth. Though this

algorithm goes through the top level of nodes many times, the majority of the nodes in this graph are at the bottom, so not much performance is lost.

```
int iddfs(source,target,max_depth)
    for(int limit = 0; limit < max_depth; limit++)
        // Depth Limited Search
        if( DLS(source,target,limit) )
            return true;
    return false;
int DLS(source,target,limit)
    if( limit == 0 )
        if( source == target )
            return true;
    if( limit > 0 )
        for( int move = 0; move < moveset.size(); move++)
            new_cube = modify_cube(source,move);
            if( DLS( new_cube, target, limit-1 ) )
                return true;
    return false;
```

Chapter 3

Implementation

3.1 Thistlethwaite's Algorithm

3.1.1 Theory

Morwen Thistlethwaite's algorithm uses group theory to transform a cube's state to be elements of groups with successively smaller search spaces. It's easy to see that the total number of reachable states using only half moves is less than the total number of states using quarter, half, and three-quarter moves. Between each phase, moves are further restricted. This restriction helps preserve the characteristics of previous phases, while allowing for reduced search spaces from the fewer allowed amount of moves. The groups are as follows:

$$G0 = \{\ \ R,\ \ L,\ \ F,\ \ B,\ \ U,\ \ D\ \ \}, size = 2,048$$

$$G1 = \{\ \ R,\ \ L,\ \ F,\ \ B,\ \ U2,\ \ D2\ \ \}, size = 1,082,565$$

$$G2 = \{\ \ R,\ \ L,\ \ F2,\ \ B2,\ \ U2,\ \ D2\ \ \}, size = 29,400$$

$$G3 = \{\ \ R2,\ \ L2,\ \ F2,\ \ B2,\ \ U2,\ \ D2\ \ \}, size = 663,552$$

$$G4 = \{I\}$$

[13, 14] As stated in chapter 1, the elements of a group are every state reachable by applying any combination of moves from that group to a solved cube. From one group to another, we are simply searching for any one of these elements.

It is important to note that when trying to move from one phase to another, we are only allowed to use the moves of the preceding group. In getting the cube into group 1, we are only allowed group 0 moves, etc.

3.1.2 Cube Definitions

After chapter 2, we have all the requisite tools to be able to implement this algorithm. First, we must define notation for the algorithm to work.

Corner Orientations

In Thistlethwaite's algorithm, a corner can have 3 orientations: 0,1,2. In this algorithm, a corner is defined as having orientation 0 if the L/R sticker on the cubie is facing the respective L/R face. If the L/R sticker is twisted clockwise with respect to the L/R face, the orientation is defined to be 1. If the L/R sticker is twisted counter-clockwise with respect to the L/R face, the orientation is defined to be 2. Pay close attention to the fact that the corner with orientation 0 need not have the

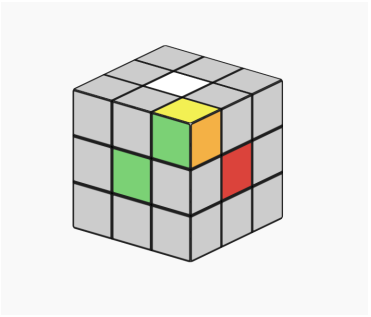

Figure 3.1: Orientation 0

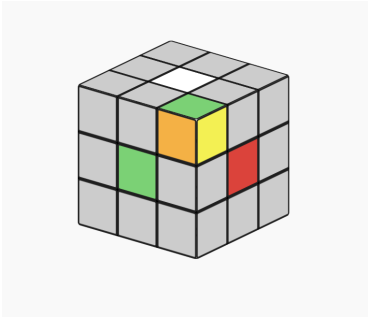

Figure 3.2: Orientation 1

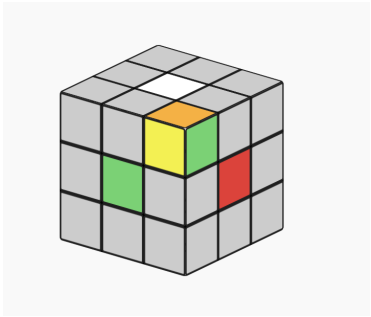

Figure 3.3: Orientation 2

same L/R sticker for the same L/R face.

Edge Orientations

Edge orientations in Thistlethwaite's algorithm are a bit harder to visualize, but they are defined as follows: Any edge that can be brought home with an even number of U or D turns is defined to be a good edge, and a bad edge otherwise. This corresponds to the fact that any U or D move flips every edge on that face. As a result, only U or D moves can affect an edge's orientation (by flipping it).

3.1.3 Coordinate Labeling

The most important function for the coordinate system is to allow for relevant characteristics to be encapsulated in a unique number such that all cubes sharing the

same characteristics map to the same number. For example, if we care only about the edge orientations of a cube, we need not worry about corner permutations or edge permutations when comparing a cube's edge orientations to another. A solved cube that has the move U2 applied to it in phase 1 will have the same coordinate as a solved cube that has the move D2 applied to it in phase 1, because only edge orientations are taken into account. The edge orientations for both of these states are zero, and thus the coordinate is zero. As well, some coordinates do not exist in other coordinate spaces; i.e., a coordinate that only contains edge characteristics carries no information about the corner characteristics. Therefore, when appropriate, the relevant moves must be applied to the entire cube to preserve the other information. This following section focuses on the efforts of extracting a cube's state information in any state. For phases three and four, the theoretical search space is much larger than the actual reachable states.

Phase One

Phase one simply corresponds to fixing the edge orientations. The total number of states corresponding to this particular step is $2^{11} = 2048$. In every subsequent phase, only half turns of U and D are allowed for the U/D faces. Since half turns of U/D moves do not affect edge orientations, and neither do R/L/F/B moves, the edge orientations will be preserved in every subsequent phase. The same concept applies in the following phases, as well.

$$G1 : theoretical \quad size = 2,048; \quad actual \quad size = 2048$$

Phase Two

Phase two corresponds to fixing the corner orientations, and placing the correct edges into the M-Slice. The M-Slice contains all edges between the L and R face. Once we exit phase two and go into phase three, only quarter turns of the L and R faces are allowed. Since L or R moves do not affect corner orientations or any edge in the M-Slice, all edges and corner orientations are preserved. The total number of states corresponding to this particular step is $3^7 * 12choose4 = 1,082,565$.

$$G2 : theoretical \quad size = 1,082,565; \quad actual \quad size = 1,082,565$$

Phase Three

The coordinates in this phase deviate from the pure theory of Thistlethwaite's Algorithm. Phase three corresponds to placing the edges in their correct slices, the

corners in their proper tetrads, fixing the parity of the entire cube, and placing the tetrads in a permutation such that the total twist of each tetrad is fixed. This phase is the trickiest to understand and implement.

The first part corresponds to placing the edges in their correct slices. The correct slices for this phase refers to the M-Slice(slice between L and R face), the S-Slice(slice between F and B face), and the E-Slice(slice between U and D face). From the previous phase, edges in the M-Slice will stay in the M-Slice for any move in the allowed moveset. If we are only allowed quarter turns of L and R, and half turns for every other move, it is clear that any edge in the M-Slice can never leave the M-Slice. For the cube to be solvable by phase four, the edges belonging to the E-Slice must be placed in the E-Slice. The remaining S-Slice edges will naturally fall into place thereafter. The total number of states corresponding to this particular step is $8choose4 = 70$.

The second part of this coordinate corresponds to placing the corners into their proper tetrads. An illustration is provided below.

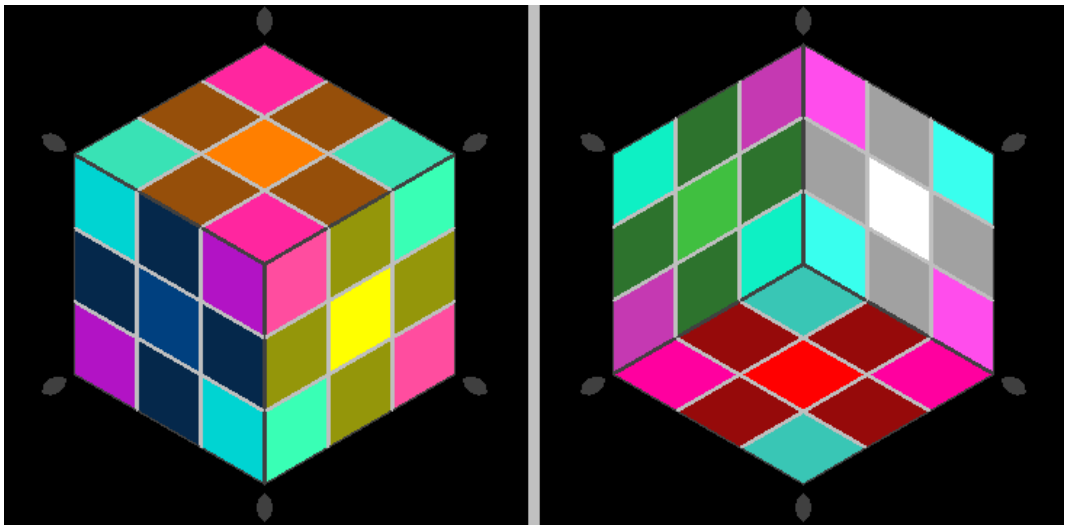

Figure 3.4: Tetrads [15]

As a reminder, the goal of phase 3 is to get the cube into a solvable state by phase 4. Phase 4 only allows half turns. With this moveset, each corner is a member of one of two tetrads. These corners, with half moves, only orbit between the 4 available corner moves. There are 2 total tetrads. It is easier to envision the tetrads by starting with the solved cube. and applying only half turns. The corners UBR, UFL, DFR, and DBL share a tetrad, and UFR, UBL, DBR, and DFL share the other

tetrad.

However, placing the corners into their respective tetrads is not enough for the cube to be solvable by phase four. After separating the corners into their respective tetrads, there are $4!^2 = 576$ possible 2-tetrad permutations. There are 6 different cosets that each permutation belongs to, and only 1 coset is a permutation which leaves the cube solvable by phase 4. Thistlthewaite, in his implementation, has a preprocessing technique, followed by a large lookup table. This is still difficult to understand, and offers us little for calculating the coordinate in a single pass when encountered.

A possible way to check if the tetrad permutation is one of 96 solvable permutations, is to attempt to solve the corners by applying a small set of moves iteratively, and solving corners one by one. When five of the eight corners have been solved, the remaining three corners are one of $3! = 6$ permutations. If the corners have been solved, then the original tetrad permutation was one of the solvable permutations, and the other 5 permutations give information on the parity of corners and respective cosets they belong to. This method of determining the 'tetrad twist', however, is extremely difficult to implement in practice. One needs some standardized way of applying the moves to solve the first 5 corners, and standardizing such a process may not even give valid results.

To mitigate against the difficulty of fixing the tetrad twist, we group the corners in each tetrad with their pair; i.e., we pair UFR with UBL, and DBR with DFL for the first tetrad, and we pair UBR with UFL, and DFR with DBL for the second tetrad [16]. In doing so, getting the corners into their tetrads and fixing the tetrad twist is accomplished in a single step. The total number of states corresponding to this particular step is as such:
$8choose2$ for pairing UFR with UBL
$8choose2$ for pairing UBR with UFL
$8choose2$ for pairing DBR with DFL
$8choose2^3 = 21952$.

The final part of this coordinate is fixing the overall parity of the cube. Referring to the theorems section in 2.2.4, edge and corner parities must coincide with each other. Therefore, we need only to check for total edge parity. Parity refers to the number of cubies that are unsolved. For the cube to be solvable by phase 4, the overall parity must be made even, since half turns always performs an even amount of swaps. The total number of states corresponding to this particular step is 2.

Treating the phase three coordinate as such is not exactly efficient or theoretically perfect, but it is functional. It allows us to uniquely identify any state while also making sure to allow for identical states(according to the phase) to map to the same

coordinate. The total size for this phase three coordinate is:
$8choose2^3 * 8choose4 * 2 = 3,073,280$.

$$G3 : theoretical \quad size = 3,073,280; \quad actual \quad size = 352,800$$

Phase Four

In standardizing cube states such that states with identical relevant characeristics, we need only to worry about uniquely labeling any possible state. To accomplish this, the coordinate in phase four is split up into 5 different parts. The first and second coordinates calculate each of the two tetrad permutations. The third coordinate calculates the M-Slice permutation (edges between the L and R face). The fourth coordinate calculates the E-Slice coordinate(edges between the U and D face). Lastly, the fifth coordinate calculates the S-Slice coordinate(edges between the F and B face). The total number of unique permutations, using a cartesian coordinate numbering scheme, is $24^5 = 7,962,624$. This is a large number, but acceptable since we store the coordinates in a map and need not allocate array space for unseen states.

$$G4 : theoretical \quad size = 663,552; \quad actual \quad size = 663,552$$

3.1.4 Transitioning Between Phases

As a reminder, the goal of this algorithm is to solve the Rubik's Cube in a reasonable amount of moves. We can use our coordinate calculations to determine if a particular state is in the appropriate group. In order to transition to the next group, we are restricted to the group moves of the phase before.

Direct Indexing

Because the elements of the groups generated by G1,G2,G3 and G4 are not too large, with a max size of 1,082,565, we can store the solutions to each group's coordinate. That is, we generate all coordinates and solutions of states reachable in every group, and directly access the solutions as we come across new state coordinates between phases. This approach is taken, as opposed to dynamically searching between states without initializing any sort of database, to highlight the power of Thistlethwaite's algorithm in speed when these databases are generated. As well, the work queue approach in generating these databases serves as a prelude for Kociemba's algorithm implementation. There is a tradeoff, however. The loading of these solution databases, once created, takes more time than dynamically searching between states.

However, once these databases are created and loaded, we can use them for any cube state solution thereafter.

Solution Database Generation

The sizes of databases for our coordinate implementations are as follows, for each group:

$$G0 = 2,048$$

$$G1 = 1,082,565$$

$$G2 = 352,800$$

$$G3 = 663,552$$

Below is a C++ implementation of the work queue approach taken to generate all states reachable at the specified phase.

```
map<int,vi> generate_solutions(int phase)
{
    vc goal_cube = initialize_cube();
    int goal_coordinate = get_coordinates(goal_cube,phase);

    map<int,vi> solutions;
    vi path;
    solutions[goal_coordinate] = path;
    queue<vc> q;
    q.push(goal_cube);

    vi moveset = applicable_moves[phase];
    while(!q.empty())
    {
        vc old_state = q.front();
        q.pop();
        int old_coordinate = get_coordinates(old_state,phase);
        for(int i = 0; i < moveset.size(); i++)
        {
            int move = moveset[i];
            vc new_state = modify_cube(old_state,move);
            int new_coordinate = get_coordinates(new_state,phase);
            if(solutions[new_coordinate].size() == 0 &
               new_coordinate != 0)
            {
                vi new_path(solutions[old_coordinate]);
                new_path.insert(new_path.begin(), inverse(move));
                solutions[new_coordinate] = new_path;
                q.push(new_state);
            }
        }
    }
    return solutions;
}
```

There are opportunities for optimization in the solution generation, such as disallowing previous moves to be performed to prevent redundancy.

3.1.5 Example Solve

We now walk through a standard cycle of randomly scrambling a cube, and then solving it, to illustrate what a typical solved phase looks like.

Scramble

30 random moves are applied to a solved cube.
Scramble Path: D' L2 D' D2 U D D' R' D2 F R F B' L F' B R' L2 F U' R R2 B' R2 B2 F' R2 L R' L

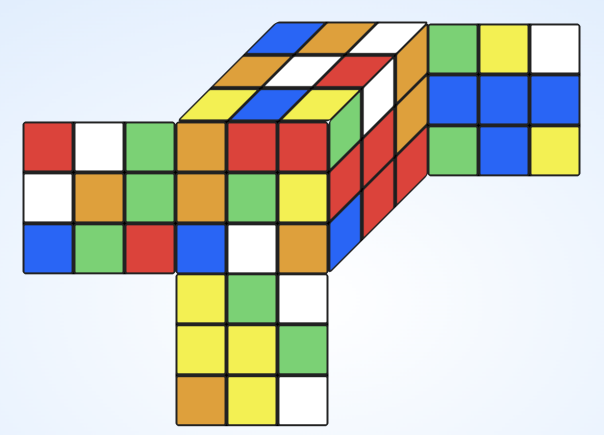

Figure 3.5: Scrambled Cube

Phase One

Phase 1 Solution: U F L2 B' U'

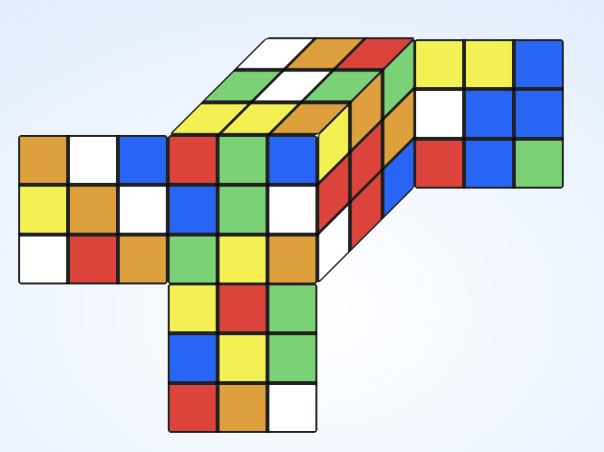

Figure 3.6: Cube After Entering Phase 1

Phase Two

Phase 2 Solution: B' L' R' D2 B' F' F' L' U2 B'

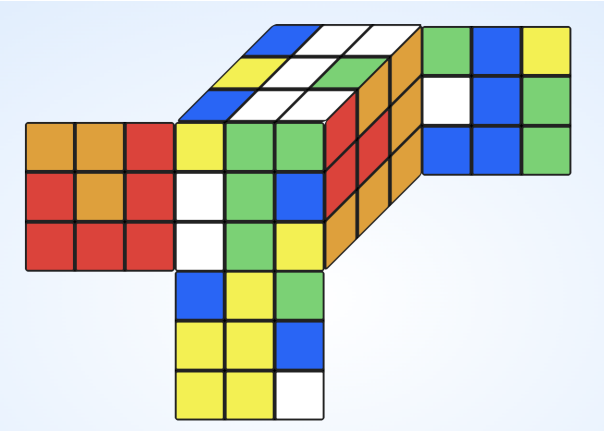

Figure 3.7: Cube After Entering Phase 2

Phase Three

Phase 3 Solution: B2 L' R' R' U2 F2 U2 B2 R'

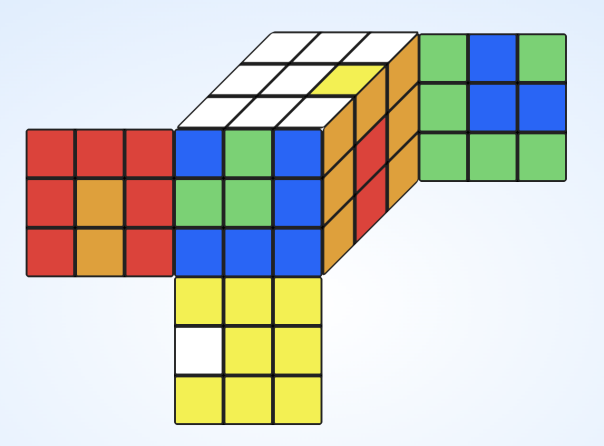

Figure 3.8: Cube After Entering Phase 3

Phase Four

Phase 4 Solution: U2 R2 D2 R2 F2 L2 F2 U2 F2 R2 F2 R2

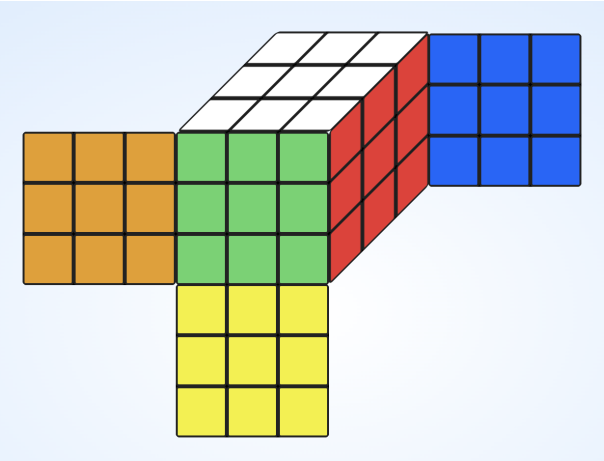

Figure 3.9: Cube After Entering Phase 4

3.2 Kociemba's Algorithm

3.2.1 Theory

Kociemba's Algorithm was a major breakthrough in shortening the solution paths of Rubik's Cube solutions. It brought the maximum solution path from 45 moves down to 30 moves, and is the standard for most cube solving programs. It offers both speed and near optimality solution lengths [10].

Deviations from Thistlethwaite's Algorithm

Kociemba's algorithm works by condensing the subgroups of Thistlethwaite's algorithm into a single group.

$$G0 = \{\ \ R,\ \ L,\ \ F,\ \ B,\ \ U,\ \ D\ \ \}$$

$$G1 = \{\ \ R2,\ \ L2,\ \ F2,\ \ B2,\ \ U,\ \ D\ \ \}$$

$$G2 = \{I\}$$

Getting from G0 into G1 corresponds to fixing the edge and corner orientations, as well as placing the relevant edges into their UD-Slice, or E-slice(the group of edges between the U and D faces).

Getting from G1 into G2 corresponds to fixing the edge permutations of the U and D faces, the corner permutations, and UD-slice permutations.

In addition, there are also minor variances in the cube notations regarding orientations. This is to be explained later.

Search Space

Phase 1 Phase 1 corresponds to reorienting the edge and corner cubies, as well as placing the 4 UD-Slice edges into the UD-Slice. We need only to worry about 7 of the 8 corner orientations, since the total corner orientations sum must be divisible by 3. The same concept applies to the edge orientations.

$$G1 = 3^7 + 2^{11} + 12choose4 = 2187 * 2048 * 495 = 2,217,093,120$$

Phase 2 Phase 2 corresponds to fixing the edge permutations outside of the UD-Slice, fixing the corner permutations, and fixing the UD-Slice edge permutations. Only half of these states are reachable in phase 2, since the parity must be made even.

$$G2 = (8! * 8! * 4!)/2 = 40320 * 40320 * 24/2 = 19,508,428,800$$

These numbers mean that in phase 1, there are 2,217,093,120 different states possible when trying to get into Group 1. In phase 2, there are 19,508,428,800 different states possible when trying to get into Group 2 from Group 1. Group 2 is the solved state.

Optimizations of Thistlethwaite's Algorithm

Kociemba himself states:
"After a long struggle I finally found the ingredients which made the maneuver search work:
- Mapping permutations and orientations to natural numbers and implementing moves as table-lookups for these numbers.
- Computing from these numbers some indices for tables which hold information about the distance to the goal state." [10]

Kociemba's algorithm is made possible by these optimizations. The mapping of permutations and orientations to sequential numbers that allow direct indexing to new states, as opposed to dynamically calculating moves, allows for quick searching between states. As well, these mappings allow us to calculate heuristics for any state we may encounter. These heuristics tell us nothing about how to solve the cube at any given moment, rather they tell us how close or how far away we are from the solution. This allows for efficient pruning of branches that are guaranteed to not have solutions that terminate within a reasonable amount of time. This, coupled with an efficient search algorithm of IDA*, make for powerful tools that lead us to speedy and near optimal results.

3.2.2 Cube Definitions

The cube definitions and notations of Kociemba's algorithm deviate slightly from Thistlethwaite's definitions. These definitions are essential in making the algorithm work, most notably his definitions of orientations. His detailed implementation, along with explanations behind the theory, can be found on his website, http://kociemba.org/cube.htm.

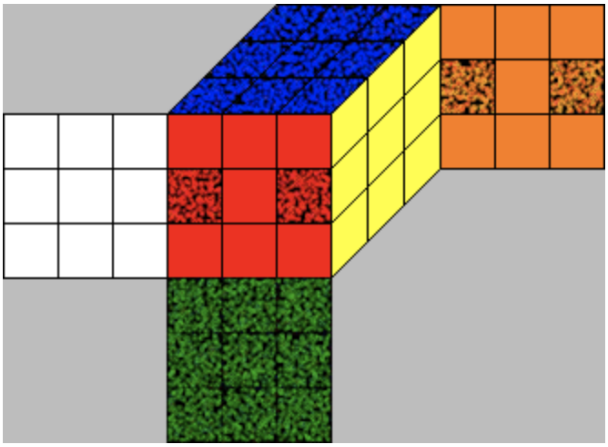

Figure 3.10: Orientation Definitions, Kociemba

Orientations

As per Kociemba, the facelets on the clean cube are the reference for the orientation. If a cubie is moved to any position on the cube, and its reference facelet matches with any reference facelet on the cube, the orientation is defined to be 0.

For an edge, if the reference facelet is not on any reference facelet of the clean cube, then the edge cubie is defined to have orientation 1.

For a corner, if the reference facelet is not on any reference facelet of the clean cube, and the facelet of the corner cubie is twisted clockwise with respect to the nearest reference facelet of the clean cube, it is defined to have orientation 1. If it is twisted counter-clockwise, it is defined to have orientation 2.

What follows is an example of applying an F move to a clean cube. From Figure 3.11, the cubie located at the UFR position has orientation 1. The cubie located at the DFR position has orientation 2, since its reference facelet is twisted counter-clockwise from the reference facelet of the solved cube. The cubie located at the DFL position has orientation 1, since its reference facelet is twisted clockwise from

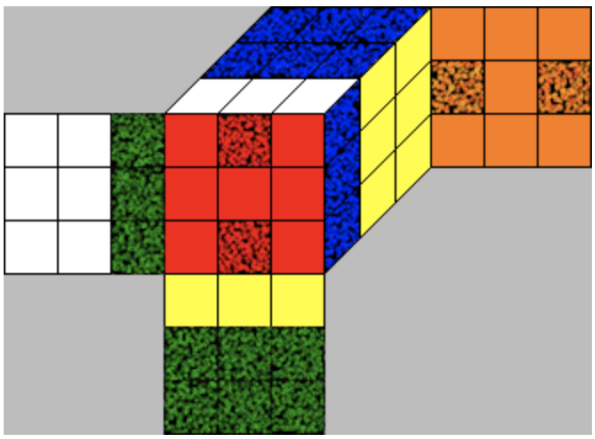

Figure 3.11: Reference cube after F move

the reference facelet of the solved cube, and the cubie located at the UFL position has orientation 2. The edges in the UF, FR, DF, and FL position have been flipped.

These definitions have the property that 10 of the 18 possible moves do not affect orientations. The moves that do not affect any orientations are:

R2, L2, F2, B2, U, U2, U', D, D2, D'

The moves that do affect orientations are:

R, R', L, L', F, F', B, B'

R, R', L, and L' only affect corner orientations. F,F',B, and B' affect both corner and edge orientations. Implementation is up to the programmer's decision, but the end product should be that the end product reflects the new permutations and orientations resulting from any given combination of moves.

Permutations

Permutation labeling is, again, left to the programmer's implementation decisions. In order to remain consistent with Herbert Kociemba's implementation, we define the reference cubie locations as such:

A solved cube permutation has edge cubies in the following order:

UR, UF, UL, UB, DR, DF, DL, DB, FR, FL, BL, BR

A solved cube permutation has corner cubies in the following order:

UFR, UFL, UBL, UBR, DFR, DFL, DBL, DBR

That is, the edge cubie in the edge index position 0 is currently in the UR position. Alternatively, the cubie in the UR position "is replaced by" whatever cubie currently occupies that space. If the the cubie occupying the space matches the reference cubie space defined above, it is home.

3.2.3 Coordinate Labeling

One of the key factors of getting Kociemba's algorithm to work is being able to map any permutation and orientation to a unique number that allows us to quickly move between states. We can do this by implementing a minimal perfect hash function. As detailed in section 2.3.4, we can sequentially map all permutations and orientations and combinations. The relevant coordinate calculations we are interested in will be grouped into phase 1 and phase 2.

Phase 1 Coordinates

There are three coordinates we calculate at any given cube state for phase 1. These coordinates are described as follows:

Edge Orientation Coordinate The first coordinate we calculate is the edge orientation coordinate. The size of this coordinate is $2^{11} = 2048$. In calculating this coordinate, we need only to worry about 11 out of the 12 corners, since the 12th edge orientation can be deduced from the previous 11 edge orientations. The procedure is described in section 2.3.4, and can be used to assign any edge orientation coordinate a value from 0-2047. The solved coordinate is 0.

Corner Orientation Coordinate This coordinate encapsulates the state of the corner orientations. The size of this coordinate is $3^7 = 2187$. We, again, only need to worry about 7/8 corner orientations for reasons similar to the ones described above.

UD-Slice Combination Coordinate This coordinate encapsulates the state of the positions of the edges belonging to the UD-Slice (also known as E-slice). The size of this coordinate is $12choose4 = 495$, since we are mapping every way we can choose 4 edges from the set of 12 edges. Any permutation of UD-edges in the UD-Slice evaluates to a coordinate value of 0. Again, the process of calculating this coordinate is detailed in section 2.3.4.

Phase 2 Coordinates

There are three coordinates we calculate at any given cube state for phase 2, as well. The coordinates in phase 1 do not hold any information that the coordinates in phase 2 can derive information from. Because of this, it is necessary to be able to get information about the starting cube's new state after exiting phase 1. This can be done by applying the move path of phase 1 to the cube. Care must be taken, however, the first successful exit from phase 1 does not necessarily correlate with the quickest or shortest overall solution, so this phase 1 path application should not be absolute. Alternatively, helper coordinates can be calculated to be dragged along the phase 1 path to help with information extraction later.

Edge Permutation Coordinate The first coordinate of phase 2 calculates the permutations of edge cubies. This coordinate is only concerned with the edges located in the M-Slice and S-Slice, or the edge cubies not located in the UD-Slice. The total size of this coordinate is $8! = 40320$.

Corner Permutation Coordinate The next coordinate of phase 2 to be described calculates the permutations of the corners. The total size of this coordinate is $8! = 40320$.

UD-Slice Permutation Coordinate The third coordinate of phase 2 seeks to fix the permutations of the edges in the UD-Slice. The size of this coordinate is $4! = 24$.

3.2.4 Search Algorithm

We now explore a search algorithm used to help navigate through phases to the solved state.

Move Tables

An important part of Kociemba's algorithm is the utilization of move tables mapped to coordinates. This means that before a search has begun, we calculate all possible coordinates and the corresponding coordinates that they map to from possible move operations. We can store this database in a 2-dimensional array, where each integer coordinate maps to another integer coordinate, indexed by the move connecting it. In doing so, no calculations need to be done during the search to a coordinate to get its resulting coordinate from applying a move operation. This speeds up the searching drastically. We need only to calculate the resulting positions from the 6 basic moves; any half or three-quarter turns can be obtained by following the appropriate chain of mapped integers corresponding to the move index. This, again, is made possible from the sequential ordering of permutations, orientations, and

combinations described earlier, as we can now directly index through an array as opposed to needing the overhead that a hash map or other data structure brings.

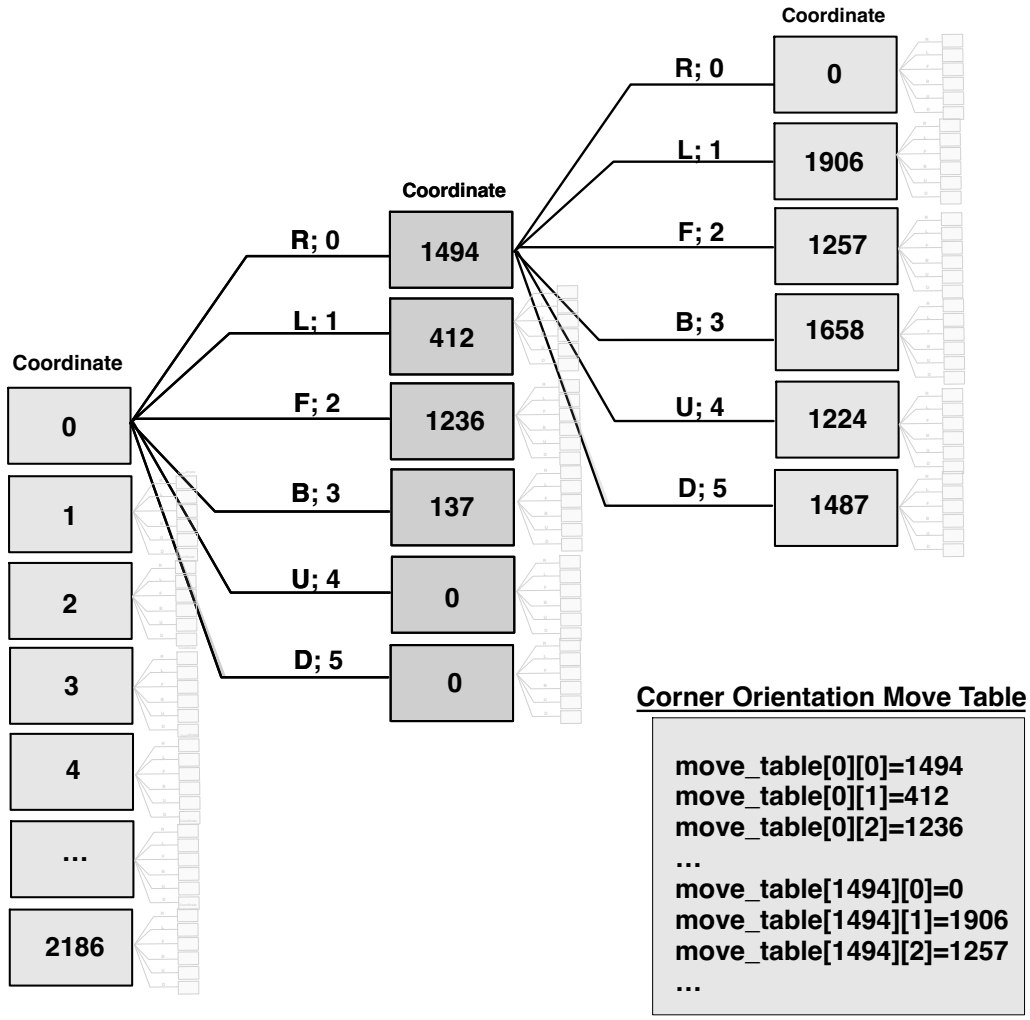

Figure 3.12: Corner Orientation Coordinate Move Tables

```
typedef vector<int> vi;
/* mt_function_ptr is a function pointer to the coordinate functions
     we are currently trying to calculate */
/* moveset is the relevant moveset for the current phase's move
   tables */
/* db is a 2d array initialized with -1 */
void generate_move_tables(vector<vi> & db, mt_function_ptr ptr, vi &
     moveset)
{
    vc cube = initialize_cube();
    queue<vc> q;
    q.push(cube);
    while(!q.empty())
    {
        vc front = q.front();
        q.pop();
        // if we have not seen this state yet, go through all of its
             moves
        if(db[ptr(front)][0] != -1)
        {
            continue;
        }
        int front_coord = ptr(front);
        int iter_coord = 0;
        for(int i = 0; i < 6; i++)
        {
            vc iter = front;
            modify_cube(iter,moveset[i]);
            iter_coord = ptr(iter);
            db[front_coord][i] = iter_coord;

            if(db[iter_coord][0] == -1)
            {
                q.push(iter);
            }
        }
    }
}
```

Heuristic Database

So far we have the move tables which allow for quick modifications between cube states. Another tool used in the search to the solved state are heuristic functions. From Wikipedia, "a heuristic (from Greek "I find, discover") is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut." The heuristic functions used in Kociemba's algorithm very effectively prune branches from the search which we know will not lead to a solution anytime soon. This process is used to help determine when a bounded search branch can no longer lead to a solution within the bounded distance. In this problem task, the heuristic function calculates the shortest distance a coordinate is away from being solved. For example, if we bound the search depth at depth 20, and at depth 15 we calculate every coordinate's heuristics and find at least one heuristic evaluating to more than 5, then there is no possible way that this current path can be solved by depth 20. We then terminate the search during that branch and start a new search from a different branch before we waste any more time searching from that branch. This process is also referred to as pruning.

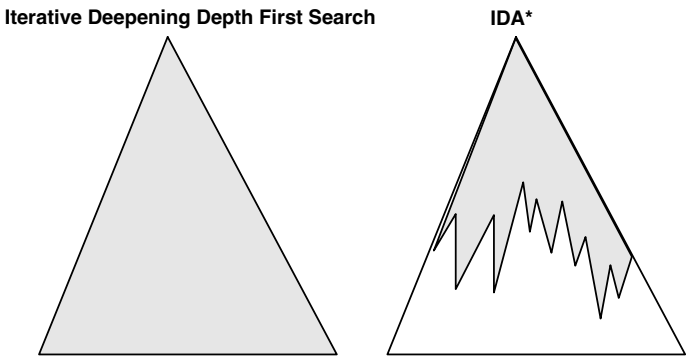

Figure 3.13: Iterative Deepening Depth First Search vs IDA* [6]

```
typedef vector<int> vi;
/* mt_function_ptr is a function pointer to the coordinate functions
     we are currently trying to calculate */
/* moveset is the relevant moveset for the current phase's move
   tables */
/* db is a 2d array initialized with -1 */
void generate_pruning_tables(vi & db, db_function_ptr ptr, vi &
   moveset)
{
    vc cube = initialize_cube();
    queue<vc> q;
    q.push(cube);
    int move_cost = 0;
    while(!q.empty())
    {
        vc front = q.front();
        q.pop();
        for(int i = 0; i < moveset.size(); i++)
        {
            vc iter(front);
            modify_cube(iter,moveset[i]);
            int iter_coord = ptr(iter);
            int front_coord = ptr(front);
            if( ((db[iter_coord]) == 0 ||
                (db[iter_coord] > db[front_coord]+1)) &&
                (iter_coord != 0))
            {
                db[iter_coord] = db[front_coord]+1;
                q.push(iter);
            }
        }
    }
}
```

The heuristic functions tell us nothing about how to solve the cube at that given state. Rather, they only tell us about how close or how far we are from the solution. We use this fact to intelligently embark on paths that are promising, throwing away the paths that quickly lead to no solution.

Optimizing Searches

A smaller optimization we can make on our searches is to prevent redundant move operations. If our previous move was an R operation, then performing another R operation is redundant. A single R2 operation will get us there quicker. As well, performing an R move after an L move that previously had an R move operation performed, could have been achieved quicker with a single R2 move again. Therefore, we do not perform moves on operations on different faces in more than one direction; i.e., we can only perform L after an R, but we can not perorm an R after an L. Similar properties hold for other faces, such as U/D moves, and F/B moves.

IDA*

The search algorithm of choice is IDA*. It is an iterative deepening depth first search (see section 2.4.3 for details) that uses the heuristic functions described above to prune branches deemed undesirable at the current depth.

```
int ida(source,target,max_depth)
    for(int limit = 0; limit < max_depth; limit++)
        // Depth Limited Search
        if( DLS(source,target,limit) )
            return true;
    return false;
int DLS(source,target,limit)
    if( limit == 0 )
        if( source == target )
            return true;
    if( limit > 0 )
        if( heuristic[source] <= limit )
            for( int move = 0; move < moveset.size(); move++)
                new_cube = modify_cube(source,move);
                if( DLS( new_cube, target, limit-1 ) )
                    return true;
    return false;
```

The following pseudocode is an implementation of Kociemba's algorithm. It must be noted that the calculations of heuristics, move tables, and move optimizations are left out for conciseness.

Kociemba's Algorithm:

```
int ida( source, target, max_depth )
    for( int limit = 0 limit < max_depth; limit++ )
        if( phase_one( source, target, limit ) )
            return true
    return false

int phase_one( source, target, limit )
    // Last move cannot occur in phase 2
    if( limit == 0 && ( last_move != move in phase 2 ) )
        if( source == target )
            phase_two_start( source, target, current_depth )
    if( limit > 0 )
        if( heuristic1[source] <= limit )
            for( int move = 0; move < moveset1.size(); move++ )
                new_cube = modify_cube( source, move )
                if( phase_one( new_cube, target, limit-1 ) )
                    return ture
    return false

int phase_two_start( source, target, current_depth )
    /* Get new state of cube if no helper coordinates were used
     * to keep track of corner and edge permutations */
    phase_two_starting_cube = modify_cube( source,
       phase_one_solution)
    source = coordinates( phase_two_starting_cube )
    /* current_depth refers to the depth that phase 1 was solved */
    /* initiate IDA* for the remainder of depth */
    for( int limit = 0; limit < max_depth - current_depth; limit++ )
        if( phase_two( source, target, limit ) )
            return true

int phase_two( source, target, limit )
    if( limit == 0 )
        if( source == target )
            SOLUTION FOUND
    if( limit > 0 )
        if( heuristic2[ source ] <= limit )
            for( int move = 0; move < moveset2.size(); move++ )
                new_cube = modify_cube( source, move )
                if( phase_two( new_cube, target, limit-1 ) )
                    return true
    return false
```

3.2.5 Putting It All Together

We now have all of the tools required to implement a model of Kociemba's algorithm. In implementing this algorithm, one can generate the databases beforehand and load it in. Though, with a C implementation, generating all of the move tables and heuristic databases takes about 0.5 seconds. Hueristic databse implementations were left out, but for every coordinate, there will be a heuristic, or pruning, database. As well, for every coordinate, there will be a cooresponding move table. In the IDA* function, there will be millions of calls to these functions, so efficiency is of utmost importance. A cube can be checked for completion by evaluating the heuristics for each coordinate in the specific phase; a sum of 0 for each heuristic indicates a solution for that phase.

In Kociemba's implementation, when a solution is found, the search continues until all paths for that solution's length has been reached. This is important to do when trying to find the optimal solution. For example, if a solution of length 20 is found, with a phase 1 depth of 13, the program continues looking for solutions at phase 13 and above, at a max depth of 20. If not, then a solution with phase 1 length of 15 and phase 2 length of 2, for a total length of 17, would be missed. If done this way, any solution found is guaranteed to be optimal (for Kociemba's algorithm). However, if speed is all that matters, the search can terminate when a solution is first found.

Example Solve

Scramble
30 random moves are applied to a solved cube.
Scramble path: B R' L R' F' F F2 L' L2 U' F' B2 L' F2 F' R' U L' F' R2 D B' D B' B' L D2 U U2 U
Edge orientation coordinate: 754
Corner orientation coordinate: 1054
UD-Slice combination coordinate: 135

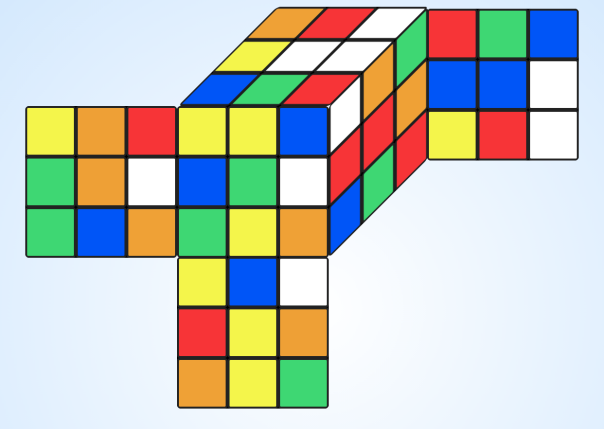

Figure 3.14: Kociemba Scrambled Cube

Phase One

Phase 1 Solution: R' B' D2 B' D' R F' U L
Edge orientation coordinate: 0
Corner orientation coordinate: 0
UD-Slice combination coordinate: 0
Edge permutation coordinate: 37183
Corner permutation coordinate: 30446
UD-Slice permutation coordinate: 9

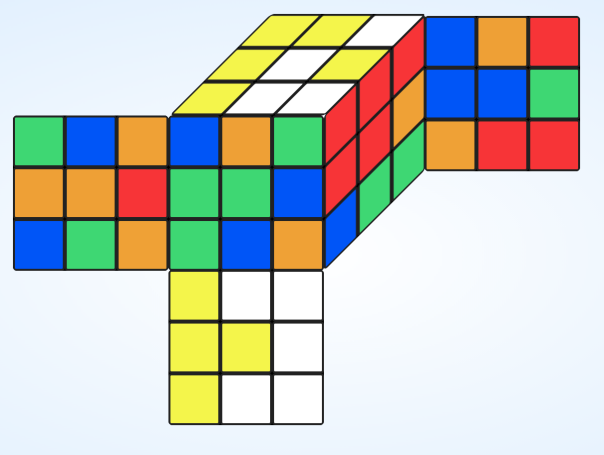

Figure 3.15: Kociemba Cube After Entering Phase 1

Phase Two

Phase 2 Solution: B2 L2 U R2 D R2 F2 U' B2 R2 L2 U B2 D
Edge orientation coordinate: 0
Corner orientation coordinate: 0
UD-Slice combination coordinate: 0
Edge permutation coordinate: 0
Corner permutation coordinate: 0
UD-Slice permutation coordinate: 0

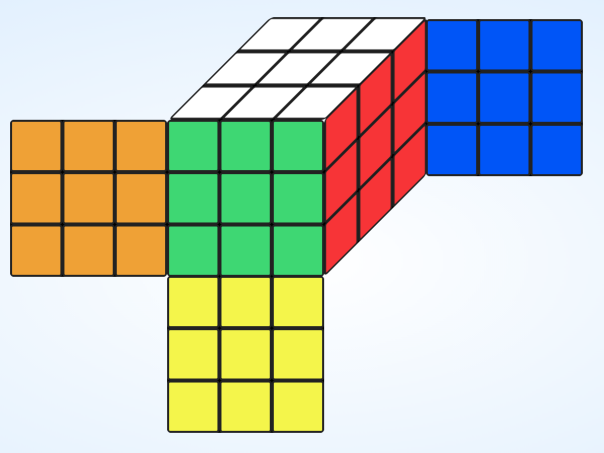

Figure 3.16: Kociemba Cube After Entering Phase 2

Chapter 4

Performance Analysis of Algorithms

After explaining the theory and implementation of Thistlethwaite's and Kociemba's algorithms, as well as attempting to implement an algorithm of our own, it is necessary to evaluate the performance.

4.1 Thistlethwaite's Algorithm Results

We compare the two different implementations of Thistlethwaite's algorithm. These two implementations include one with a bidirectional breadth first search where no solution tables are created and paths are found dynamically, and one where the solutions are loaded in from precalculated solution tables.

We will see that the method with direct indexing from a solution table is significantly faster than any other algorithm implementation described; speed is measured in microseconds as oppposed to milliseconds, because we directly access solutions once they are loaded in. Displayed above is the bidirectional breadth first search implementation of Thistlethwaite's algorithm. It is slower than directly accessing solutions with preloaded tables, but no time has to be spent loading the data in or generating the solutions.

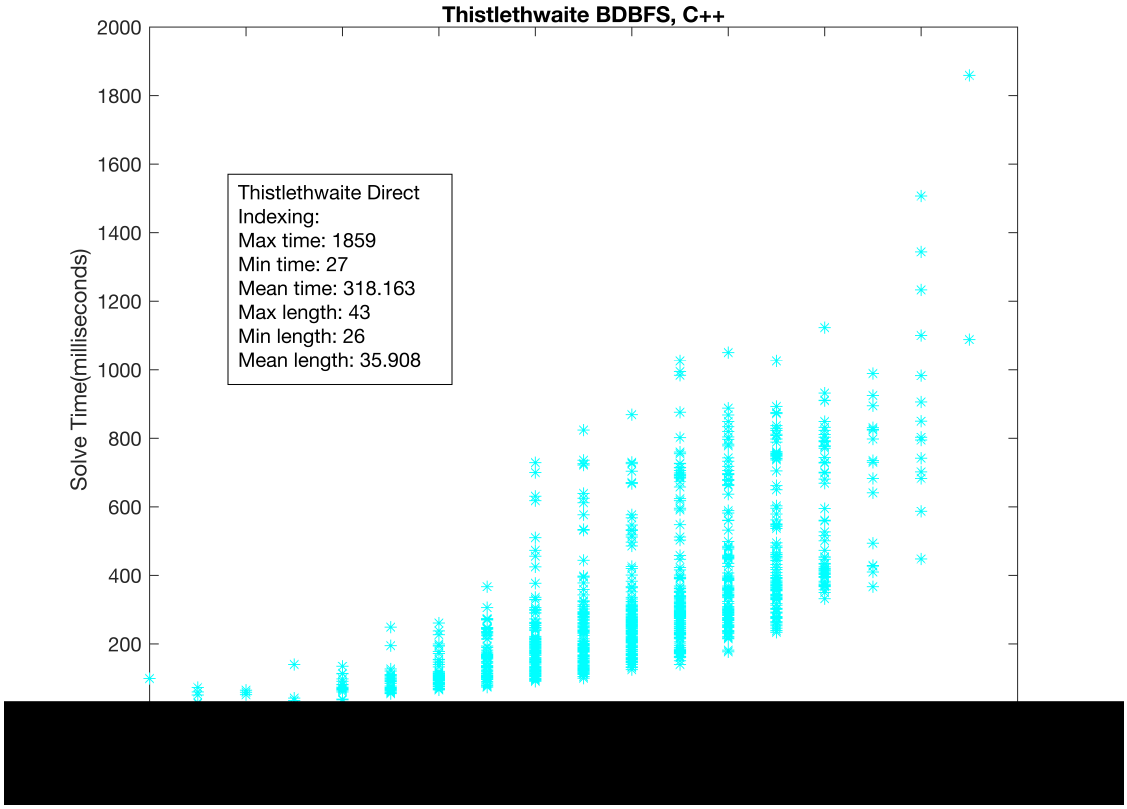

Figure 4.1: Thistlethwaite Bidirectional Breadth First Search

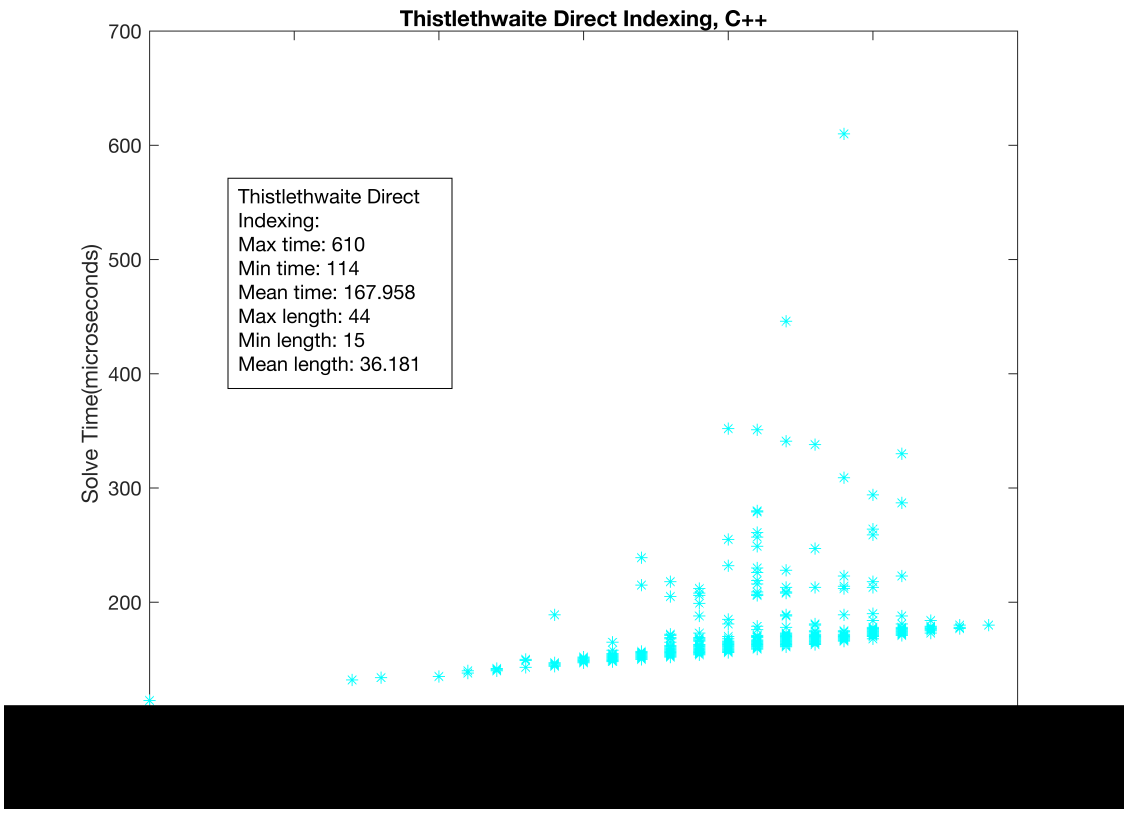

Figure 4.2: Thistlethwaite Direct Indexing

For both of these implementations, the average solution length is about 36 moves. It is far from optimal, but very short when comparing the search space of an entire Rubik's Cube.

4.2 Kociemba's Algorithm Results

In analyzing Kociemba's algorithm, the results are far more appealing. An average solution length of around 23 moves is achieved, which is not far from God's number of 20 [11]. When analyzing pure program speed, which varies between machines and implementations, we can see a difference between the C++ and C implementations of Kociemba's algorithm. Solution lengths are similar, but execution speed nearly doubles from the C to C++ version, with the C implementation being quicker.

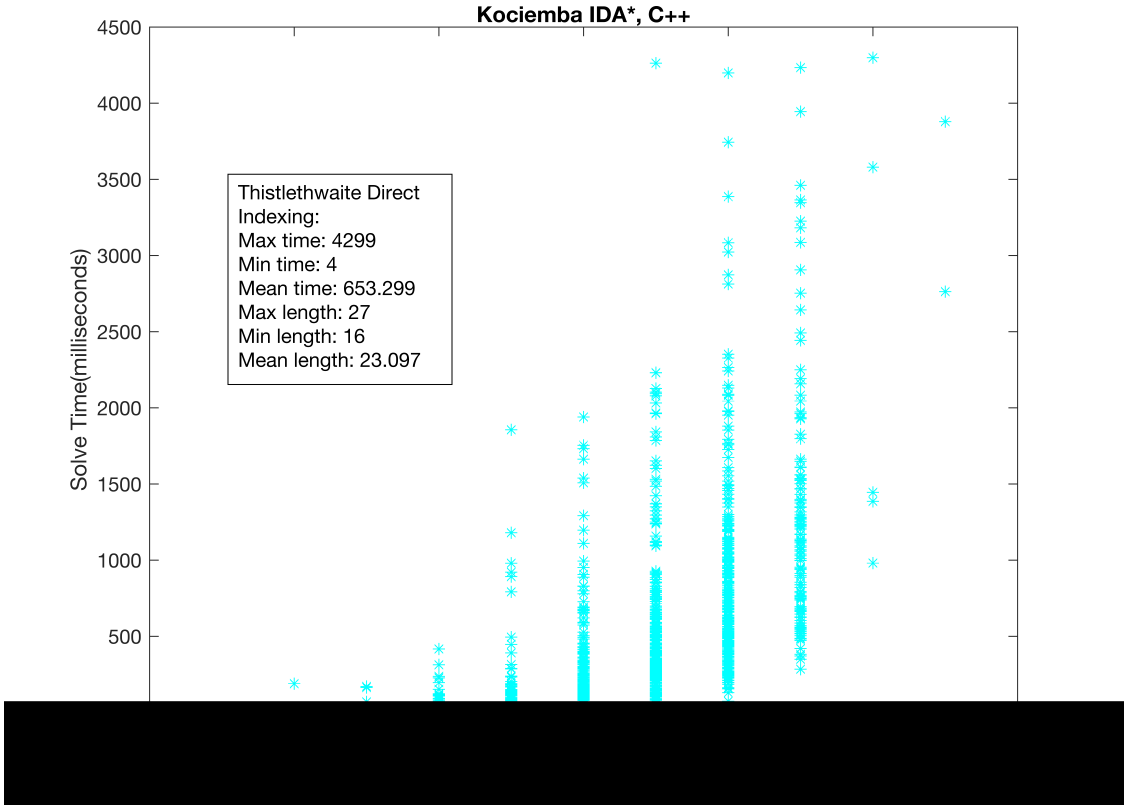

Figure 4.3: Kociemba C++ Results

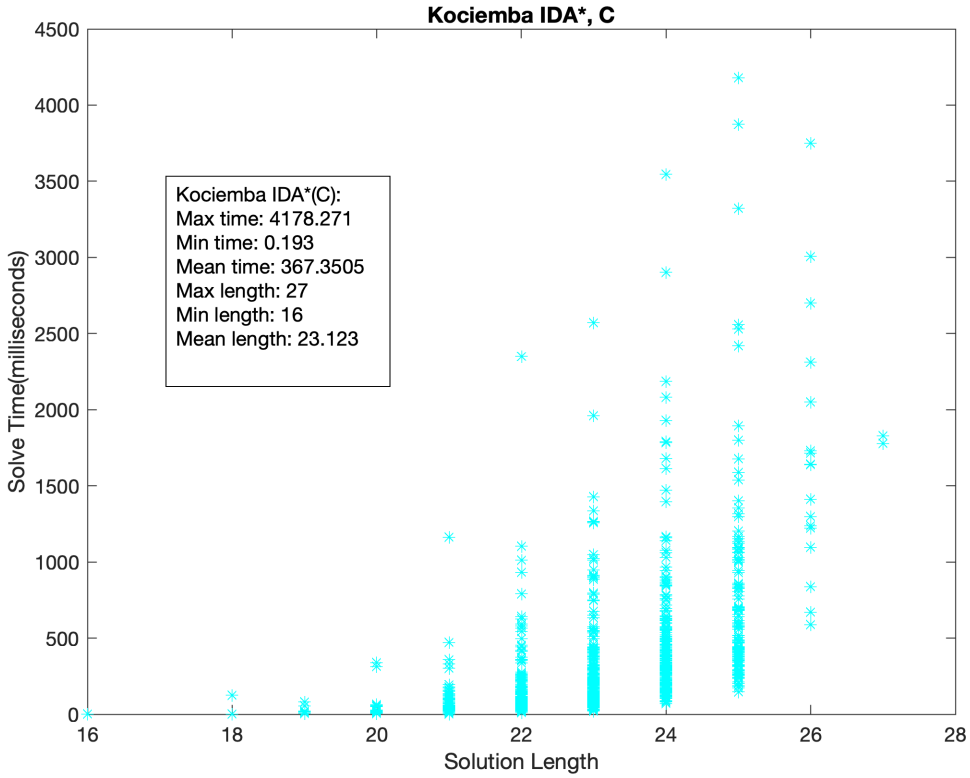

Figure 4.4: Kociemba C Results

4.3 Comparisons

We now display some performance differences between Kociemba's algorithm and Thistlethwaite's algorithm. Most importantly, we compare the performances of the dynamic searches of solutions, and not the direct indexing of solutions.

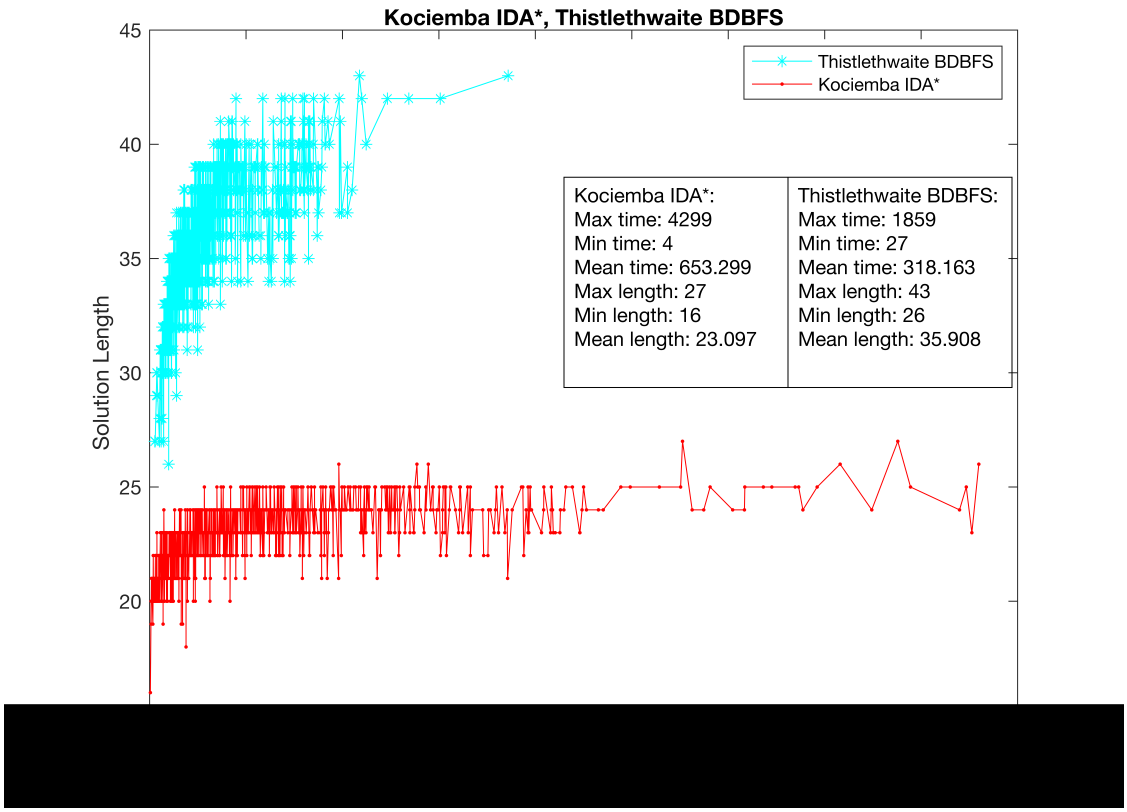

Figure 4.5: Kociemba C Results

The comparisons of Kociemba's algorithm with a C implementation and Thistlethwaite's algorithm with a C++ implementation is as follows:

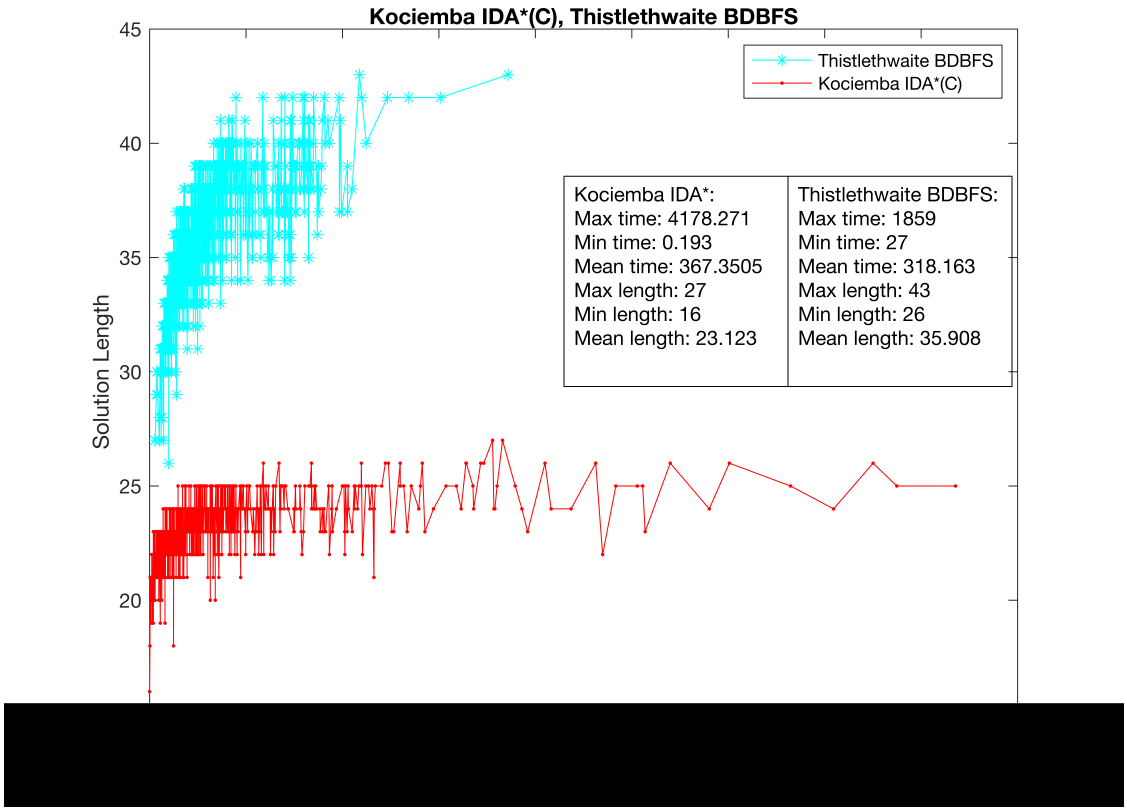

Figure 4.6: Kociemba C Results

Chapter 5

Conclusion

5.1 Thoughts

Though I recognize that this paper offers very little in innovation and clarity, it was a very fun learning experience for me. I conquered my fear of depth first search algorithms, and learned many memory and speed optimization techniques.

5.2 Improvements

Given the time, I would love to improve on some of the techniques described previously. I would like to explore a bidirectional A* search, as well as utilizing some more memory in the pruning tables and move tables to make for quicker IDA* searches in Kociemba's algorithm.

5.3 The Future

Moving forward, there are many things to expand upon with regards to Rubik's Cube solving algorithms. The tools to find solutions to other subproblems are readily available, such as knowing the cube representations and the different search algorithms one can use to get there. Improving current OLL/PLL algorithms, taking into account move execution speeds for a human solver, or applying techniques of deep learning are all topics I would love to get around to exploring some day. As well, I would love to implement Richard Korf's cube solving algorithm with attempts at optimizing the search spaces.

In any case, the Rubik's Cube is not a completely solved problem yet. God's number has been found, but finding the technique of reliably and quickly getting there is still a problem to be solved!

Bibliography

[1] Available at https://softcover.s3.amazonaws.com/2181/rubik_book/images/cube_notation.jpg.

[2] Rubik's cube move notation. Available at https://jperm.net/3x3/moves.

[3] Set (mathematics). Available at https://en.wikipedia.org/wiki/Set_(mathematics), journal=Wikipedia, publisher=Wikimedia Foundation, Sep 2020.

[4] Binary operation. Available at https://en.wikipedia.org/wiki/Binary_operation, Sep 2020.

[5] Group(mathematics). Available at https://en.wikipedia.org/wiki/Group_(mathematics), Sep 2020.

[6] Le Thanh Hoang. Optimally solving a rubik's cube using vision and robotics. Available at https://www.doc.ic.ac.uk/teaching/distinguished-projects/2015/l.hoang.pdf, Jun 2015.

[7] Permutation. Available at https://en.wikipedia.org/wiki/Permutation, Sep 2020.

[8] Combination. Available at https://en.wikipedia.org/wiki/Combination, Sep 2020.

[9] Lindsey Daniels. Group theory and the rubik's cube. Available at https://www.lakeheadu.ca/sites/default/files/uploads/77/docs/Daniels_Project.pdf, 2014.

[10] Herbert Kociemba. Solve rubik's cube with cube explorer. Available at http://kociemba.org/cube.htm.

[11] God's number is 20. Available at https://cube20.org/#:~:text=New%20results%3A%20God's%20Number%20is,requires%20more%20than%20twenty%20moves.

[12] Breadth-first search. Available at https://en.wikipedia.org/wiki/Breadth-first_search, Sep 2020.

[13] Jaap Scherphuis. Jaap's puzzle page. Available at https://www.jaapsch.net/puzzles/compcube.htm.

[14] Jaap Scherphuis. Jaap's puzzle page. Available at https://www.jaapsch.net/puzzles/thistle.htm.

[15] What is the meaning of a "tetrad twist" in thistlethwaite's algorithm? Available at https://puzzling.stackexchange.com/questions/5402/what-is-the-meaning-of-a-tetrad-twist-in-thistlethwaites-algorithm, 2015.

[16] Thistlethwaite 3x3 solver (written in c++). Available at https://www.stefan-pochmann.info/spocc/other_stuff/tools/solver_thistlethwaite/solver_thistlethwaite_cpp.txt.