Rubary: An Evolutionary Algorithm for Rubik's Cubes

Jason Mahr
jasonmahr@college.harvard.edu

Abstract

Rubary, an amalgam of Rubik's and evolutionary, is an evolutionary algorithm that solves Rubik's Cubes. It was inspired by and iterates on an evolutionary Thistlethwaite Algorithm published by El-Sourani et. al. in 2010 [1]. Key improvements to their algorithm included shared move histories among cubes, improved fitness functions, an expanded final phase, and an improved selection method. The final algorithm solves in an average of 47 seconds with an average solution length of 46 moves, compared to 340 seconds with an average solution length of 50 moves. The 47 second average is for a Python 2.7 implementation and includes time from communicating with the provided GUI. The code for this project is available at github.com/jasonmahr/rubary.

1. Introduction

As discussed in lecture, evolutionary and genetic algorithms are widely used, but still poorly understood. In particular, good representations are critical yet difficult to characterize. In recent years, there has therefore been an increased interest in applying these algorithms to new problems. These efforts have led to a better understanding of their strengths and weaknesses. Particularly interesting areas of research include applications in medicine and engineering [2]. This report introduces an application of the evolutionary algorithm to solving Rubik's Cubes, along with several optimizations that were made to both data structures and the algorithm itself.

The report begins with a description of previous work, including an introduction to the Thistlethwaite Algorithm in Section 2 and to the evolutionary algorithm in Section 3. Section 4 describes improvements to El-Sourani et. al.'s work as well as the theory and intuition behind the innovations. Experimental results are discussed in Section 5. Finally, Section 6 mentions additional features, including a GUI and a cube validity checker, and Sections 7 and 8 summarize the work.

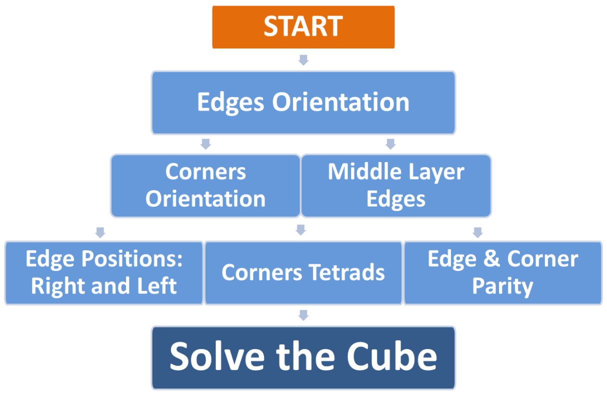


Figure 1. Thistlethwaite Algorithm Flow

2. Thistlethwaite Algorithm

2.1. Background

Mathematician Morwen B. Thistlethwaite developed the Thistlethwaite Algorithm in 1981 with the goal of solving the Rubik's Cube in the fewest number of moves possible. The algorithm can solve in a maximum of 52 moves, which at the time was the method with the fewest number of moves. The algorithm differs from easier-to-memorize layer and corners-first algorithms in that it does not aim to place pieces in their correct positions one by one. Instead, the algorithm works on all pieces simultaneously, restricting them to increasingly fewer possibilities until there is only one possible position for each piece, which represents a solved cube. The algorithm achieves this by progressively limiting the types of moves needed to solve the Rubik's Cube. Mathematically, this represents a sequence of nested groups, each of which satisfies properties of closure when only applying allowed moves for said group. The cube starts at Group 0 (G0), which contains all possible cubes, and ends at G4, which only contains the solved cube and no allowed moves. The steps, which are detailed next, are summarized in Figure 1.

Before discussing the steps, some clarifications are in order. First, L, R, F, B, U, and D represent the left, right, front, back, up (top), and down (bottom) faces, respectively, of a cube. Additionally, each face is associated with the color of its center tile, since these tiles do not move with any face turn. Until the third step, going from G2 to G3, it helps to think of opposite colors as the same color. For this reason, multiple sides are concatenated, e.g. UD refers to U or D. For groups, a single symbol means quarter turns are allowed, and a symbol with a 2 means only half turns (two quarter turns) are allowed. For example, the allowed moves for G0 are <L, R, F, B, U, D>, while the allowed moves for G1 are <L2, R2, F, B, U, D>. Finally, in regards to move notation, L denotes a clockwise turn of the face L, $L2$ denotes a half turn, and L' denotes a counterclockwise turn; this applies to all six faces.

2.2. G0

The cube starts in Group 0, where the algorithm aims to orient edges correctly. Once they are oriented, as mentioned, the cube can be solved without using quarter turns of the L and R faces. It follows that an edge is oriented correctly if and only if, if moved to its home position only moves from G1, the edge will be flipped the correct way. Of course, the algorithm does not need to move the edge there in order to figure this out. Instead, the rule is twofold. First, there must be no F or B colors facing U or D, nor on the edges of L and R that border the F and B sides. Intuitively, edges where one side faces F and B clearly have to have the F or B color on the F or B side (hence not on UDLR edges bordering FB), else at the home position the edge would be wrongly flipped. Additionally, the four U and D edges not bordering F and B also cannot have F or B, since U is allowed in G1, and U could result in those moving to their home position. Similarly, the other twelve edge colors must not be U or D. These are all FB edge pieces and the LR edge pieces bordering UD. If these rules are satisfied, it will seem like nothing has been accomplished, but in fact quarter turns of L and R are no longer required, and the algorithm can move to G1.

2.3. G1

The second step is to put the cube into G2, whose allowed moves are <L2, R2, F2, B2, U, D>. As shown in the figure, there are two goals in this step. The first is that corners have to be oriented correctly. Since this implementation has arbitrarily chosen U, D to be reserved in G2, this means that all corners' UD color has to face the top or bottom. Intuitively, if there is a corner where the UD color is not on the top or the bottom, moves of U or D will not fix this, and neither will any half turns, since this will just make the top corner color that is not UD go to the bottom side, which means the cube would never be solved. The second goal is that middle layer edges must be in the middle layer. This also makes sense: U and D will not move any middle edges out of the middle layer, and neither will any half turns. Thus, the four middle edges must be in the middle layer. If they are, it follows that the eight edges with U or D are bordering the UD layers, and from Step 1 it follows that these UD colors would face the up or down faces. Thus, the rules of G1 can be simply stated as the U and D faces must have all U or D colors. After the third step, quarter turns of F and B are also no longer required. Despite the seemingly simple requirement, this is the hardest step and has the largest factor, as will be discussed.

Table 1. Move Sets, Size, and Factors of Thistlethwaite Groups

	Move Set	Size	Factor
G0	<L,R,F,B,U,D>	4.33e19	2048
G1	<L2,R2,F,B,U,D>	2.11e16	1,082,565
G2	<L2,R2,F2,B2,U,D>	1.95e10	29,400
G3	<L2,R2,F2,B2,U2,D2>	6.63e5	663,552
G4	<>	1	1

2.4. G2

The third step achieves a state where only half turns are needed to solve the cube, after which the algorithm moves to G3 (<L2, R2, F2, B2, U2, D2>). The first requirement is that corners must be in their circuits. Intuitively, in G3 each corner can only move between one of 4 positions, which can be called a circuit. There are only two circuits and each corner belongs to one of those circuits. For example, the corner at the top-left corner of F cannot move to the top-right position of F through only half turns, so these are in different circuits. All half turns swap two corners in one circuit and the two corners in the other circuit that overlap, limiting the possible permutations so that just achieving opposite colors on all sides is not enough to move to G3. To see if corners are in their circuits, bring all U or all D corners to U. Pairs of corners around the top and bottom rows of LRFB will either all match or all mismatch. They need to either all match or all mismatch, and also all be of their own face's color or the opposite color. There is also a separate requirement for edges, of which there are 12 in 3 circuits. The middle circuit edges were already done. For the remaining, the FB circuit edges (the ones at FB and UD) must have FB facing FB, and the LR circuit edges (the ones at LR and UD) must have LR facing LR. This satisfies the edge positions requirement, and either requirement being satisfied will already have satisfied edge and corner parity being even, since in all valid cubes, either both are odd or both even.

2.5. G3

The fourth step moves from G3 to G4, the group which allows no moves and contains only the solved cube. Only half turns can be used for this step. This approach overall is unique in that it harnesses group theory to progressively limit the possible moves, which is well-suited to helping an evolutionary algorithm converge. As mentioned above, each step has a different difficulty level. From the analysis provided by El-Sourani et. al., the outermost G0 has 4.33e19 possible cubes (all possible cubes), G1 has 2.11e16 possible cubes (this many have well-oriented edges), G2 has 1.95e10, G3 has 6.63e5, and of course G4 has 1. This information, which is summarized in Table 1, means G0 has to find 1 of 2048 cubes, which is quite easy. G1, on the other hand, has to find 1 of 1,082,565 cubes, which makes it by far the hardest step. As will be discussed, improvements were made in this algorithm to help with this step as compared to El-Sourani et. al.'s algorithm, but it is still the bottleneck. G2 has to find 1 of 29,400 cubes, and G3 has to find 1 of 663,552, which was a significant bottleneck in El-Sourani's et. al. algorithm. This paper discusses an expanded final phase that makes this step significantly easier by introducing G3B, with its own unique requirements in order to move to the move set <F2, B2, U2, D2>, and G3C, again with its own unique requirements for moving to the move set <U2, D2>. The end result is an algorithm with only one slight bottleneck instead of two major bottlenecks, an improvement reflected in the dramatic 700+% improvement between the two algorithms.

3. Evolutionary Algorithm

This project uses an evolutionary algorithm to solve Rubik's Cubes. As discussed in lecture, these are a variant of stochastic beam search that use biologically inspired mechanisms such as selection, reproduction, and mutation to converge on a solution to a given problem. At each step, or generation, candidate solutions are evaluated according to a fitness function and selected with preference for better-performing candidates. The schematic of a typical evolutionary algorithm is provided in Figure 2. Genetic algorithms, a subclass of evolutionary algorithms, use selection, crossover, and mutation to produce the next generation. The algorithm used in this paper is similar except that mutations occur with a high rate and attach to the end of a move sequence, and crossover is not used. A more traditional genetic algorithm was experimented with, but did not perform well. This is discussed further in the Results section. Overall, the main algorithm functions as seven separate evolutionary algorithms that move from one phase to the next. These phases and their fitness functions are examined in more detail in the next section.

4. Approach

4.1. Overview

This section discusses El-Sourani et. al.'s implementation and then compares it to this paper's best algorithm. The overall approach in both algorithms is very similar. First, a scrambled cube is duplicated λ times to form the first population. El-Sourani et. al. use $\lambda = 50,000$; this paper's constants were optimized to different values, and this

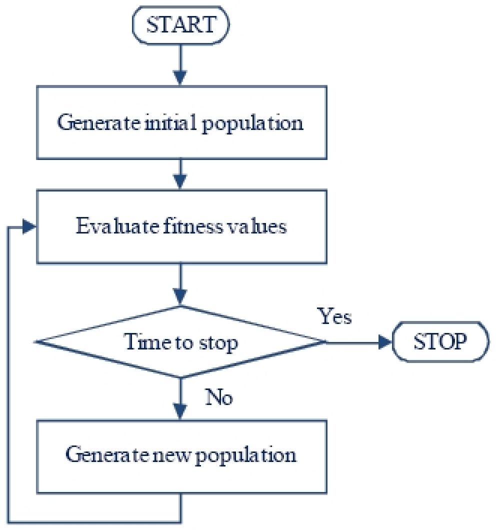


Figure 2. Evolutionary Algorithm Flow

is discussed in the Results section. Each cube is then mutated, which involves applying a few random moves from the move set corresponding to the current phase. The algorithm then selects the next generation of cubes from among the μ best cubes, evaluated according to the fitness function corresponding to the current phase. Size is weighed as part of the fitness functions in order to slant toward shorter solution lengths. El-Sourani et. al. use $\mu = 1000$. This is repeated until μ cubes all solve the current phase, at which point the algorithm progresses to the next phase. This ensures the algorithm does not immediately proceed when it finds a single qualifying cube, which could have a very long move sequence. Instead, it continues to search until there are μ solutions to help achieve shorter move lengths. The algorithm halts when it reaches a phase past the last phase, and it resets when it reaches an upper bound on the number of generations, in order to escape local optima. Whereas El-Sourani et. al. use an upper bound of 150, the algorithm in this paper almost never resets, due to several improvements, but it uses a G1 upper bound of 30 to be safe. The upper bound is only activated if the algorithm is still stuck in G1 when it reaches 30 generations.

4.2. Rubik's Cube

El-Sourani et. al. implemented individual cubes as 6 2D matrices for the six sides coupled with a list of the moves it has undergone, in order to later inform the user of the found solution. This is where this paper's algorithm begins to have noticeable improvements over El-Sourani et. al.'s algorithm.

First, and most importantly, this paper uses shared move histories, as opposed to El-Sourani et. al.'s implementation, where each individual has its own list of moves that have

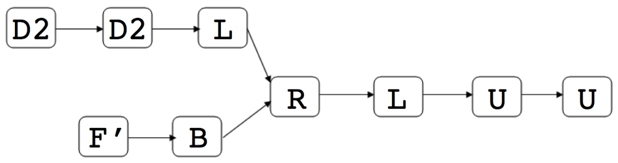


Figure 3. Shared Move Histories

been applied. This is very inefficient in terms of both computation time and memory, since with each generation λ new lists have to be created. Furthermore, by the time the algorithm reaches the final phase, it is usually the case that nearly cubes share the same moves from G0 and G1. For these moves to have to be copied λ times each generation is incredibly wasteful. The alternative, shared move histories, represents move histories as reverse linked lists. When cube b copies cube a, it just redirects its history pointer to cube a's linked list. Then, when b makes a new move, it prepends that move to the linked list and updates its own pointer, so that a's pointer is not affected. Another cube, c, can also make a copy of a's pointer, and attach different moves. This is illustrated in Figure 3, where b starts at d2, a starts at r, and c starts at f'. Reversing the linked list is necessary, since if b were to append its move, it would also affect a's history, since in a linked list each node only points to one other node. With this structure, copying move histories becomes trivial. Given an average solution length of 50, this represents an upper bound of saving 50 times the work per cube, since one pointer is copied as opposed to 50 moves. This bears out in testing, where this insight alone sped up copy time by over 20 times and overall computation time by over 4 times. These speed improvements are as compared to individual move history lists in Python 2.7.

Second, redundant move removal was improved as well. El-Sourani et. al.'s implementation merely addresses back-to-back moves of the same face. Their example is that $F2BB'R2R2F$ reduces to F' by first removing BB', then $R2R2$, then transforming $F2F$ into F'. While this is helpful, their implementation fails to address sandwiches: sequences such as LRL, which also occur very often and can be simplified to $L2R$ or $RL2$. This paper implements both checks, and for the most part they are straightforward even with shared move histories. Consider as an example the pair UU in Figure 3 above. Continuing the previous example, suppose d points to the left U in Figure 3, and e points to the right U. Then when removing redundancies for d, we would edit the node of the first U to hold $U2$ and point to what the second U points to. Immediately, a, b, and c have now all been edited as well, and e is unaffected. Removing each redundancy only once is another benefit of using shared move history.

In addition to so-called non-disappearing pairs, there are three other types of redundancies that are slightly more complex. The first is a pair that disappears. Consider the two $D2$s. b points to the left $D2$, and imagine d now points to the right $D2$ and e now points to a move to the left of the $D2$s. If removing redundancies is called on e, the algorithm will redirect the pointer of that move to the left of the $D2$s to L. Then if removing redundancies is called on b, it will redirect b's history pointer directly to L. This work could not have been done when optimizing e's history since e does not have access to b's history pointer. When both e and b have been updated, d is unaffected. The second additional category is a non-disappearing sandwich. Consider the $LRLU$, which for b reduces to $L2RU$. The algorithm will change the left L to $L2$, but it cannot redirect R's pointer to U, since then a's sequence would have changed from RLU to just RU. Thus, since RLU must be preserved for a, b creates a new node with the value R pointed to by the updated $L2$ and pointing to U. The last additional category is disappearing sandwiches, which is similar to the previous case. If the Ls were instead $L2$s, the rightmost $D2$ would be updated to point to a new node with the value R and which points to U. This preserves necessary moves for all possible cubes sharing this history, while also saving as much work for later cubes as possible. These savings also greatly lowered computation time. In the final version of the algorithm, nested lists were used instead of linked lists, according to the same concept. Compared to linked lists, nested lists were even faster in Python.

In addition to shared move histories and more robust and efficient redundancy removal, a third improvement to Rubik's Cubes is cube verification, which is discussed in detail in the Additional Features Section. This feature was a necessity for any user-facing application, since mistakes in inputting cube colors can result in an impossible search task if the inputted cube is not first verified. Their specific dismissal of the use of verifying cubes implies they did not create a graphical interface for users along with their algorithm. There is a GUI that accompanies this paper's algorithm; it is also discussed in Additional Features. Also discussed later is an experiment involving using integers for the sides instead of lists. For each side there would be $6^8 = 1,679,616$ possibilities. Since it was slightly slower, the idea is discussed in the Other Attempts section.

4.3. Fitness Functions

Each phase of the evolutionary algorithm has its own fitness function, since the phases are evaluating different criteria. Apart from better optimized weights for fitness versus size in the fitness score, the logic behind the fitness function for G0 (Phase 0) was not altered from El-Sourani et. al.'s implementation. The function simply counts the number of wrongly flipped edges, and the progression from G0 to G1 is very easy for the evolutionary algorithm, with only a factor of 2048.

The progression from G1 to G2 is much more difficult, with a factor of 1,082,565, and it is the main bottleneck the evolutionary Thistlethwaite algorithm. El-Sourani et. al.'s idea of separating the step into two functions was maintained, with the added idea of separating the step into two phases. The first phase would solve for middle edges only, and then the second for both middle edges and corner orientation. This proved to help slightly in helping move the algorithm to a more promising of the search space after G1, as it resulted in an average of five fewer generations needed. Additionally, the fitness functions themselves were improved. For the fitness function evaluating middle edges, credit was given for the case where there are exactly two wrong middle edges, since it is easier to fix this than one wrong middle edge: align the two wrong middle edges on the FU and FD borders, or on the BU and BD borders; put the two other middle edges safe on the other side through <L2, R2>; and then a single turn of F or B respectively solves the middle edges. Thus even more credit is granted if the correct middle edges are both on the F or both on the B sides, and the wrong middle edges are both on the other such side. For wrongly oriented corners, credit was given for pairs of wrong corners being at positions in the same corner circuit, as it is easier for corners in the same circuit to rotate and then swap positions. These two changes help the algorithm get through G1 easier, but it is still the hardest step. The current state is an algorithm that sometimes can take up to 15-20 generations on G1, but at least rarely needs to restart.

The progression from G2 to G3 is not as difficult, but it can still be helped significantly. This paper's approach for this step departs significantly from El-Sourani et. al.'s algorithm. It consists of three fitness functions whose weighting is such that the first must be solved for the second to matter, and the second must be solved for the third to matter. In other words, the weight of a single mistake in the second is higher than the max of the third, and likewise for the weight of the first. The first fitness function is to bring all U or all D corners to U. The second fitness function checks pairs of corners around the top and bottom rows of LRFB, which should either all match or all mismatch, and also all be of their own face's color or the opposite color. The information from the second fitness function is meaningless if the first is not satisfied, hence the very disparate weights. The third fitness function requires that LR and FB edges in the top and bottom rows must be LR or FB, respectively. This requirement is easier than the previous two, and the disparate weights ensure that any mistake in the first or second fitness function is addressed immediately. Additionally, as with G1, each fitness function has optimizations. The first outputs the number of turns until U's corners are all U or all D, instead of simply the number of wrong corners. The second returns the difference from all matched or all mismatched, whichever is closer, with credit for both pairs on a side being mismatched or matched, whichever is farther, and only comments on opposite colors if the harder matching requirement is already satisfied. Similar to the first fitness function, the third returns the number of turns until all edges are satisfactory, not just the number of edges that are incorrectly positioned. Together, these work a lot better in driving convergence of G2 than El-Sourani et. al.'s approach, saving about 15 generations on average.

Finally, for the progression from G3 to G4, El-Sourani et. al. simply count wrong colored tiles. However, the factor for this step is way too large at 663,552 for such a simple approach, and indeed this fitness function got stuck in local optima often. Instead, this paper proposes two additional groups to Thistlethwaite's original 5, G3B and G3C, with move sets of <F2, B2, U2, D2> and <U2, D2>, respectively. There are two conditions to get from G3 to G3B. First, the eight edge tiles on FBUD bordering LR must be correct. This is because if the LF and LB edge pieces were swapped, this could not be resolved without L2 or R2. U2 and D2 would not touch these edges, and F2 and B2 would keep them on the wrong FB side. Second, the other tiles on the FBUD faces must be uniform in their rows. Intuitively, this is because there is no way to break apart these rows without turning the L or R sides. Once these two criteria, which are much easier than solving the entire cube at once, are met, the algorithm can enter G3B. At G3B there are three conditions to enter G3C. First, U and D must be fully solved, since U2 and D2 do not change the tiles on U and D. Second, the middle edges must solved, since UD turns do not affect these. Third, the other tiles on L and R must form uniform rows, since UD turns cannot break these rows up. Once these three criteria are satisfied, the cube can be solved with only U2 and D2. Just as solving Thistlethwaite's groups are much easier than solving an entire cube at once, this three-part task is much easier than solving G3 all at once. G3B and G3C could therefore represent a worthwhile addition to Thistlethwaite's original 1981 algorithm. The resulting algorithm has never been stuck in the final phase, and in fact solves all of G3 in an average of only 8 generations versus dozens before.

4.4. Mutation

Mutation is essentially done the same way as in El-Sourani et. al.'s implementation, with a random number of random moves from the current move set being applied to each cube in the population. The random number of moves used by El-Sourani et. al. is between 0 and, depending on the group, 7, 13, 15, or 17, inclusive. These numbers were provided by Thistlethwaite as the maximum sequence lengths needed to transform the cube from one group to another, and they add up to the 52 moves mentioned earlier. The one difference is that, as this paper uses several enhanced phases, extensive testing has determined that for G0, 7 is optimal; for the two steps of G1, 5 and 13; for G2, 15; and for G3A, G3B, and G3C, 10, 8, and 2, respectively.

4.5. Selection

The last aspect of the evolutionary algorithm, the selection mechanism, was slightly optimized as well. El-Sourani et. al. use truncation selection, which they justify with the observation that mutations and phase transitions can abruptly increase or decrease the fitness scores of cubes. Trials conducted using this paper's algorithm corroborate their claim, with rank-based selection mechanisms consistently outperforming fitness-based selection mechanisms in general. They choose randomly among the μ best individuals for the next population. Extensive experimentation showed that it was better to slightly prefer better cubes among the μ best individuals. The best performing selection algorithm was a modified restricted elitist rank selection based on geometric series, which guarantees that all μ individuals will survive at least once, and then assigns probability such that the probability of one individual being chosen is $\frac{\mu-1}{\mu}$ times the probability of the next individual being chosen. This outperformed other ideas as well, including assigning proportional probability of $\mu - rank$ to each individual, where $rank$ is zero-indexed, an approach that favored the top cubes too much.

5. Experimental Results

5.1. Software and Hardware

The evolutionary algorithm was implemented in Python 2.7 and ran on a 2013 baseline Macbook Air. Additionally, the TkInter library was used to create the GUI, and bash scripts generating CSV files were used to automate testing.

5.2. Results

Figure 4 shows average solution length vs. average time over 200 trials for various settings of λ and μ. The shortest time not over 50 moves was 47 seconds, which corresponded to $\lambda = 11700$ and $\mu = 390$. This represents an over 7 times increase over El-Sourani et. al.'s average time of 340 seconds. As can be seen from the graph, faster solutions tend to have more moves. This corresponds to the fact that larger population sizes have more reach through the search space, so they can find more optimal solutions, but they also take longer to complete each generation. Other constants, including weights for each fitness function, were also meticulously optimized.

5.3. Other Attempts

As mentioned previously, a more traditional genetic algorithm was attempted for this problem. It used the same fitness functions as the evolutionary algorithm, but instead of mutating by applying moves to cubes (mutating by

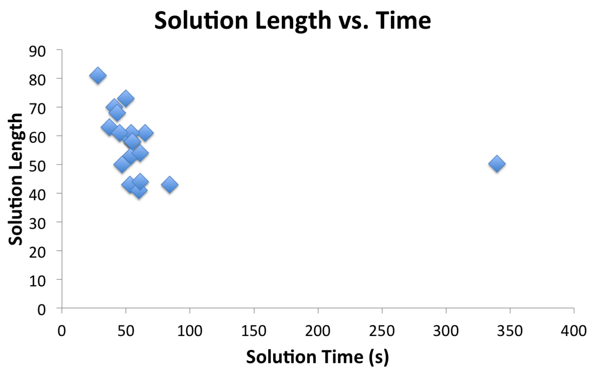


Figure 4. Data for Various Constants

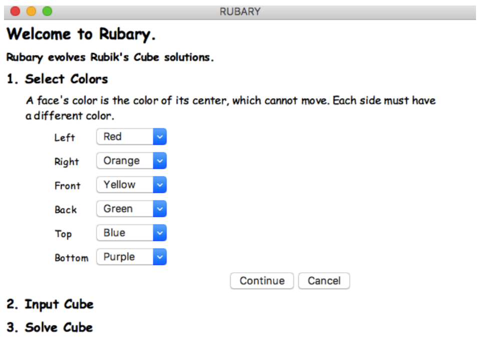
adding moves to the end of the sequence), mutating random moves and random crossover were tried together and separately. The resulting algorithm reached G1 in an average of 10 seconds, which is twice as long as the evolutionary algorithm, and remained stuck in G2 94 times out of 100 trials. Intuitively this makes sense: move sequences for Rubik's Cubes do not work well with crossover or random-point mutation, since changing the first move results in an entirely different cube, whereas changing the last move results in only a slightly different cube. Thus, unlike problems well suited for genetic algorithms, such as 3-SAT, each chromosome of a move sequence is not equal in influence.

Another experiment involved encoding cube sides using integers instead of lists. With lists, each side of the cube is encoded as 8 integers representing tiles, each with 6 possible colors. The result is $6^8 = 1,679,616$ possibilities. The idea was to encode each side as an integer, and then operations would involve dividing by powers of 6 according to the index and then modding by 6. This implementation was promising at first, as it drastically decreased the time the algorithm spent on copying cubes. However, when there were other improvements with lists, they became much faster than using integers. It may be the excessive number of calculations involved with this approach that ultimately rendered it impractical. Code for both experiments are included with the project code.

6. Additional Features

Two major additional features that are provided with the algorithm are a graphical user interface (GUI) and a cube validator. The GUI was built using the TkInter library, and it features a full-color cube input canvas that also has scrambling capability. It updates live on every generation so the user knows what phase the algorithm is currently working on as well as the fitness scores of the lowest rank member of μ. Screenshots are provided in Figures 5, 6, 7 and 8.

Another feature is a cube validation function, which greatly improves the usability of the GUI, since a single


RUBARY

Welcome to Rubary.

Rubary evolves Rubik's Cube solutions.

1. Select Colors

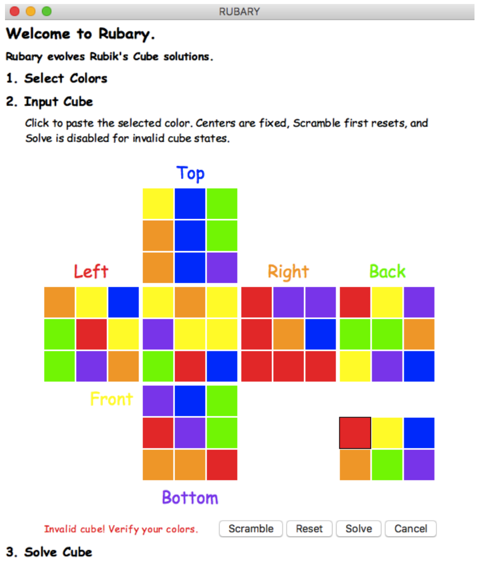
A face's color is the color of its center, which cannot move. Each side must have a different color.

Left: Red
Right: Orange
Front: Yellow
Back: Green
Top: Blue
Bottom: Purple

2. Input Cube

3. Solve Cube

Figure 5. GUI Step 1


RUBARY

Welcome to Rubary.

Rubary evolves Rubik's Cube solutions.

1. Select Colors

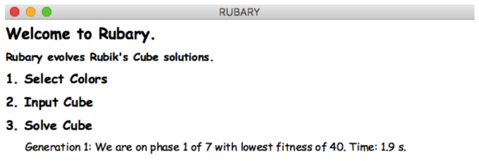
2. Input Cube

Click to paste the selected color. Centers are fixed, Scramble first resets, and Solve is disabled for invalid cube states.

Invalid cube! Verify your colors.

3. Solve Cube

Figure 6. GUI Step 2


RUBARY

Welcome to Rubary.

Rubary evolves Rubik's Cube solutions.

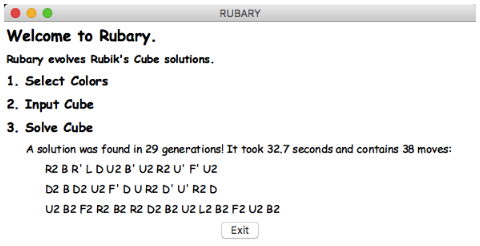
1. Select Colors

2. Input Cube

3. Solve Cube

Generation 1: We are on phase 1 of 7 with lowest fitness of 40. Time: 1.9 s.

Figure 7. GUI Step 3


RUBARY

Welcome to Rubary.

Rubary evolves Rubik's Cube solutions.

1. Select Colors

2. Input Cube

3. Solve Cube

A solution was found in 29 generations! It took 32.7 seconds and contains 38 moves:

```
R2 B R' L D U2 B' U2 R2 U' F' U2
D2 B D2 U2 F' D U R2 D' U' R2 D
U2 B2 F2 R2 B2 R2 D2 B2 U2 L2 B2 F2 U2 B2
```

Figure 8. GUI Result

misclick will not result in an impossible search problem and the user ultimately having to reenter the entire cube. A cube is valid if there are edge and corner parities of 0, as well as an overall even permutation of cube pieces.

7. Conclusion

Through several innovations, Rubary was able to achieve an over 7 times increase in solution time over a published evolutionary Rubik's Cube algorithm despite being built in Python. With regards to data structures, improvements included implementing shared move histories, a more robust approach to consolidating redundant moves, and cube verification. In terms of the theory behind the algorithm itself, Rubary uses guided fitness functions and an expanded final phase in order to avoid bottlenecks at G1 and G3. The concept of G3B and G3C not only worked very well, but also fit nicely in with Thistlethwaite's other ideas, with each step removing two moves like the previous steps until no moves are left. The final results suggest that thinking carefully about a problem's search space can lead to novel insights and understanding that can in turn dramatically improve evolutionary algorithms.

8. Further Work

The several constant variables of the algorithm, including λ, μ, and the fitness function weights, could be further optimized. Due to limited time and computational resources, the focus of this paper was on the constants that seemed to be slowing down the algorithm. Another short-term objective would be to analyze the cube states to see if any configurations in particular are more likely to lead to local optima. For example perhaps there were optimizations in G0 that would facilitate G1 progress. This was thought about during the project, but not much insight was gleaned.

A long-term objective would be to visualize the algorithm's progress. Currently, the GUI provides a message indicating the number of generations, phase, fitness, and time of the algorithm. It would be more intriguing to have a graph that could have the cubes along the x axis, their fitness along the y axis, and optionally the length of their move sequence on the z axis. This is more engaging, and it would allow the user to better understand how the evolutionary algorithm is working under the hood. Another long-term objective would be to apply these concepts to solving higher dimension cubes, for instance a 4×4 cube. And of course, the code could be rewritten in a faster language, which would allow innovations such as shared move histories have the greatest possible impact.

References

[1] N. El-Sourani, S. Hauke, and M. Borschbach. An evolutionary approach for solving the rubiks cube incorporating exact methods. In *European Conference on the Applications of Evolutionary Computation*, pages 80–89. Springer, 2010.

[2] A. Ghaheri, S. Shoar, M. Naderan, and S. S. Hoseini. The applications of genetic algorithms in medicine. *Oman medical journal*, 30(6):406, 2015.