Project 6 (Final) – GA & WOC Rubik's Cube

Stone Barrett
Computer Science Engineering
Speed School of Engineering
University of Louisville, USA
scbarr04@louisville.edu

Charles Weiss
Computer Science Engineering
Speed School of Engineering
University of Louisville, USA
c0weis02@louisville.edu

1. Introduction

In this sixth and final project, we were asked to find classmates to work with as a team in order to choose a problem considered to be NP-complete and adapt the wisdom of artificial crowds algorithm used in the previous project to solve the selected problem. The NP-complete problem our team chose to research and implement the algorithm for is the Rubik's Cube puzzle. The Rubik's Cube puzzle was invented in 1974 by an architecture professor named Ernő Rubik. (Wiki) In essence, the puzzle structure is that of a cube where each face contains a three-by-three grid of squares. Each of these squares is filled with one of six colors: white, blue, red, yellow, green, or orange. The cube can be transformed by rotating any face or slice of the cube from any perspective. The puzzle begins when the colors of each square have been sufficiently scrambled. The action required is to transform the cube through a series of rotations. This transformation is complete when the series of moves leads to the complete goal – every face of the cube only contains one color; all of the squares on a face must look the same. Now that the problem space is explained and the goal state clearly identified, how can a solution be generated using a genetic algorithm with wisdom of artificial crowds implementation?

2. Approach

As stated previously, the algorithm being used for this project is a variation of the genetic algorithm, the wisdom of artificial crowds. Such as when implemented in previous projects, a genetic algorithm is one that simulates natural selection and generational development in order to quickly and efficiently arrive at increasingly better solutions. This is done by using data structures to represent chromosomes, genes, individuals, genetic crossover and mutation, breeding, and populations. In essence, an initial population is generated with (potential) solutions to a problem. These solutions will act as individuals which will have their evolutionary fitness scored and weighed to determine likelihood of who will get to reproduce. When two solutions reproduce, their genes crossover, meaning parts of one parent solution are passed on to a child solution and the rest of the steps are filled in by the other parent solution. This occurs repeatedly until a cutoff criteria is met – sometimes thousands of generations down the line – and the final generation's solutions should be very good if not optimal. In this typical scenario, two parents reproduce sexually to create offspring solutions and the very best individual is taken to represent the problem's solution at the end. For the sake of this project's problem, those aforementioned commonalities may not stand; there will be more on that later.

The difference between a standard genetic algorithm (GA) and one using the wisdom of artificial crowds (WOC) approach is that in WOC, there are multiple individuals considered in the end instead of the singular best. Sections from the whole final generation or even multiple iterations of a GA can be considered as individual "expert opinions". These so-called experts are not influenced by what the other expert solutions are via any methods. Therefore, multiple individual solutions can be compared and contrasted discretely to find common steps and form an aggregate solution. This aggregate solution, as was explored with the Traveling Salesperson Problem (TSP), can often be more optimal than a standard GA result.

The difference between the natures of TSP and Rubik's Cube solutions made this task seem to be more difficult than it turned out to be. With TSP, any order of cities that makes a Hamiltonian Path through the map will result in a solution. It may not be a very good solution, but it is a solution nonetheless. It's a rather analog problem, where the solutions gradually get better as the GA evolves. In comparison, Rubik's Cube could be considered a binary problem, where the solution either returns it back to its starting state or not; it is either solved or it isn't. At first glance, one could wonder just how a GA would be useful on a problem such as the Rubik's

Cube. The fitness function could either be geared toward the lowest number of moves in a string of transformations or just how close the current Cube layout is to the starting position. The starting position can be referred to as the "identity Cube" from here on, similarly to a 3-Dimensional identity matrix in linear algebra. Through series of testing during software development, we decided that in the interest of time, the fitness function and as such the GA's evolutionary goal should align with how close the series of moves is to a solution, not the fewest moves possible. The idea of fewest moves could certainly be explored further, as most competitions involving Rubik's Cube solving determine the winner based on how fast they can solve the puzzle. However, it is difficult to say with certainty that there is a linear relationship between the raw number of moves taken and the time it takes a human hand to perform those moves. This idea has been explored ad nauseum by researchers with much more time and computational power than us. Interestingly, there is proof that every possible position on a Rubik's Cube – all 43,252,003,274,489,856,000 of them – can be solved in 20 moves or less. (Cube20) However, as stated previously, this algorithm will focus on returning a solution, not how many moves are in that solution.

Moving onward, the GA in this project will need a way to generate populations and breed. To begin, the Cube will need to be modeled in some way that the data can be compared easily. This is done using a JavaScript library called Rubiks-Cube, which handles the Cube's positions, rotations, and comparisons. (NPM) The ability to compare easily will be essential when assigning a fitness score to individuals. To create an initial population, the identity Cube will have *n* random moves performed to it *p* times, where *p* is the desired population size and *n* is a predetermined 20. The reason we settled on 20 moves being a sufficient scrambling is that 20 is that standard number of moves used in scrambling Cubes in official competitions. (SpeedSolving) Typically, a software known as *TNoodle* generates scramble sequences for the *WorldCubeAssociation*. (WCA) This initial population will now be iterated through for the individuals to be assigned fitness scores. Fitness scores are given based on how close the current Cube is to the identity Cube. For every square on any face that is different, the fitness score for that individual is incremented. Therefore, the closer to 0 an individual's fitness score is, the closer to a solution that string of moves is. The population is then sorted by fitness scores and the elite size rule is applied. Elitism in a GA is when an individual is so genetically fit based on the fitness function that it is automatically carried over to the next generation. In this case, for a 10 entered for elite size when running the program, the lowest 10% of fitness-scored individuals will be carried over. As for the rest

of the population's breeding, this is where a Rubik's Cube GA truly differentiates from other GAs we've observed in this class. Since breeding two parents and crossing over the genes in a way that one might do when solving TSP would possibly (likely, even) result in a series of impossible moves based on the current orientation, reproduction in this problem is achieved asexually. This means no mating pairs of individuals are formed, only individuals producing offspring alone. Therefore, the crossover function in this case will be represented by appending the genes of an individual to create an offspring. This is done by calling what will represent the mutation function for this project – a method that uses random number generation to select what series of actions will be taken to modify the individual. Depending on the number rolled, the parent may have random permutations of moves added to its move list or it may have its orientation flipped or rotated about a certain axis. It may even have multiple of these things done in different orders. Once these actions are taken and the offspring has been created, it is placed in the subsequent generation for the process to continue. This idea for asexual reproduction and permutation lists used for appending move sets comes from Roberto Vaccari, who beautifully documented their Rubik's Cube GA experimentation in Python. (Vaccari) Vaccari also tipped us off to one of the online Rubik's Cube simulators we used to test our generated solutions and visually represent them. These simulators can be seen in Section 3.2.

With solutions being generated by a GA, now it's time to implement the WOC approach. For this project – just like with GA – trying to insert moves where they weren't already would eventually result in impossible moves being tried. Even a greedy algorithm fix would not be of assistance in beating this solution into proper form. So, despite choices made about the nature of the GA, we decided that the number of moves could be taken into consideration by the WOC. As such, the crowd will be generated by the best opinions from multiple iterations of the GA running at once. At the end, the expert individual that completed solving the Cube first will be the solution chosen. With this, the speed of a veritable solution is now accounted for, too.

Some additional information about this implementation is that after 50 generations without a solution to the Cube, the GA starts from the beginning again. During testing, this was found to prevent the process from getting stuck in move loops and stalemating in evolutionary progress. Now that the stage is set, let us see how this approach performed at solving the Rubik's Cube.

3. Results

3.1 Data

The data used in this experiment is relatively simple. A standard three-by-three Rubik's Cube will be the object being manipulated with moves defined by the JS library mentioned in Section 2. Random rotations and permutations to be selected from during the mutation phase are defined by Vaccari's work as mentioned in Section 2 as well.

The parameters to be entered when calling the program are: the number of random moves to scramble the Cube in the beginning, the number of individuals to be placed in one generation, the number of generations to allow before resetting the process, and the percentage of each generation that will be considered genetic elites.

Figure 3.1

```
./index.js 20 1000 50 30
```

In the next section, we will see figures displaying the results of our program with different datasets, with a focus on the number of generations it takes to generate a solution. The data sets will use the first and third variables in Figure 3.1 as control variables, while we will see how the outcome changes when modifying the population size and genetic elite percentage. To help visualize evolutionary improvement curves, the JS program output is ran through a Python program that uses the *MatPlotLib* library seen in previous projects.

The next section will be broken into two main parts – the GA results and the results when WOC is added. The GA section will show results for four different datasets:

I. Population size = 500, Elite percentage = 10
II. Population size = 500, Elite percentage = 30
III. Population size = 1000, Elite percentage = 10
IV. Population size = 1000, Elite percentage = 30

The WOC section will capitalize on the trends discovered by the GA by taking the attributes from the dataset with the best results and amplifying them to discover the most optimal setup for a run in this implementation.

3.2 Results

Genetic Algorithm

In the following figures, note that "max" refers to the average fitness score of an *initial* generation, while "min" refers the average fitness score of *generation n-1*, where *n* is the *final* generation. This is because *n*'s fitness score is always zero, since every square on the Cube matches every square on the Identity Cube. Furthermore, "gens" refers to the number of generations it took to evolve into a working solution. There are eight runs for each dataset and the average and standard deviation are also given for each set. Let's begin.

I. P = 500, E = 10

Figure 3.2

	Max	Min	Gens
Run 1	31.764	6.468	188
Run 2	32.008	6.966	43
Run 3	28.174	6.978	247
Run 4	30.05	6.636	28
Run 5	26.902	6.726	288
Run 6	30.836	7.078	89
Run 7	30.886	7.352	37
Run 8	32.524	7.386	485
Avg	29.89533	7.026	195.6667
Stdv	1.826781	0.306229	0.306229

Figure 3.3


```
{
  best: {
    perms: "U2 R U2 R' F2 L F' L' U2 L' U2 L F' z' y2 L' U2 L R' F2 R x U B2 D2 R F2 D2 B2 L U x' U' B2 D2 L' F2 D2 B2 R' U' R' U2 R L' B2 L U' B2 D2 L' F2 D2 B2 R' U'
D' R' D R2 U' R B2 L U' L' B2 U R2 F R B L U L' U B' R' F' L' U' L U' z' x2 F' U B U' F U B' U' F R B L U L' U B' R' F' L' U' L U' x' z F' U B U' F U B' U' U2 B U2 B' R
2 F R' F' U2 F' U2 F R' z' D' R' D R2 U' R B2 L U' L' B2 U R2 F' U B U' F U B' U' L U' R U2 L' U R' L U' R U2 L' U R' U z' x2 U2 R U2 R' F2 L F' L' U2 L' U2 L F' D L D'
 L2 U L' B2 R' U R B2 U' L2 D L D' L2 U L' B2 R' U R B2 U' L2 x z' D' R' D R2 U' R B2 L U' L' B2 U R2 x z' F' U B U' F U B' U' y2 R' U L' U2 R U' L R' U L' U2 R U' L U'
 U2 R U2 R' F2 L F' L' U2 L' U2 L F' U2 B U2 B' R2 F R' F' U2 F' U2 F R' L' U2 L R' F2 R R' U2 R L' B2 L L' U2 L R' F2 R z' U2 B U2 B' R2 F R' F' U2 F' U2 F R' y U2 R U
2 R' F2 L F' L' U2 L' U2 L F' z2 U' B2 D2 L' F2 D2 B2 R' U' F' L' B' R' U' R U' B L F R U R' U z' F' L' B' R' U' R U' B L F R U R' U y' F' L' B' R' U' R U' B L F R U R'
 U x2 z2 F R B L U L' U B' R' F' L' U' L U'",
    fitness: 0
  },
  start: "z L B' E2 E' B F F' D' F2 R2 D2 U2 y B' D' R2 E' F' M2",
  gens: 247,
  fitness_avgs: [
    28.174, 27.206,  25.64, 24.046, 22.812,  21.83, 20.756,  19.64,
    18.712, 17.746, 16.932, 16.176, 15.566, 14.902, 14.598,  14.05,
    13.918, 13.572, 13.186, 13.036, 12.834, 12.232, 12.052, 11.582,
    11.224, 10.848,  10.31, 10.016,  9.936,   9.25,  9.252,  9.052,
     8.988,  9.018,  8.902,  8.988,  8.858,  8.906,   8.81,  8.864,
     8.588,  8.674,  8.472,  8.518,   8.37,  8.272,  8.254,  8.126,
     7.888,   7.85, 28.154, 27.058, 25.486, 24.052, 22.832, 21.822,
    20.796, 19.882, 18.858, 18.076,  17.39, 16.634, 16.192, 15.494,
    14.932, 14.412,  13.93, 13.212, 12.766,  12.62, 12.168, 11.904,
    11.784,  11.42, 11.058, 10.506, 10.002,  9.876,  9.784,  9.644,
     9.646,  9.474,  9.266,  9.156,   9.06,  9.122,  9.274,  9.126,
     9.182,   9.28,  9.188,  9.096,  9.294,  9.084,  9.162,  9.128,
      9.11,   9.19,   9.07,  9.046,
    ... 148 more items
  ]
}
```

Figure 3.4

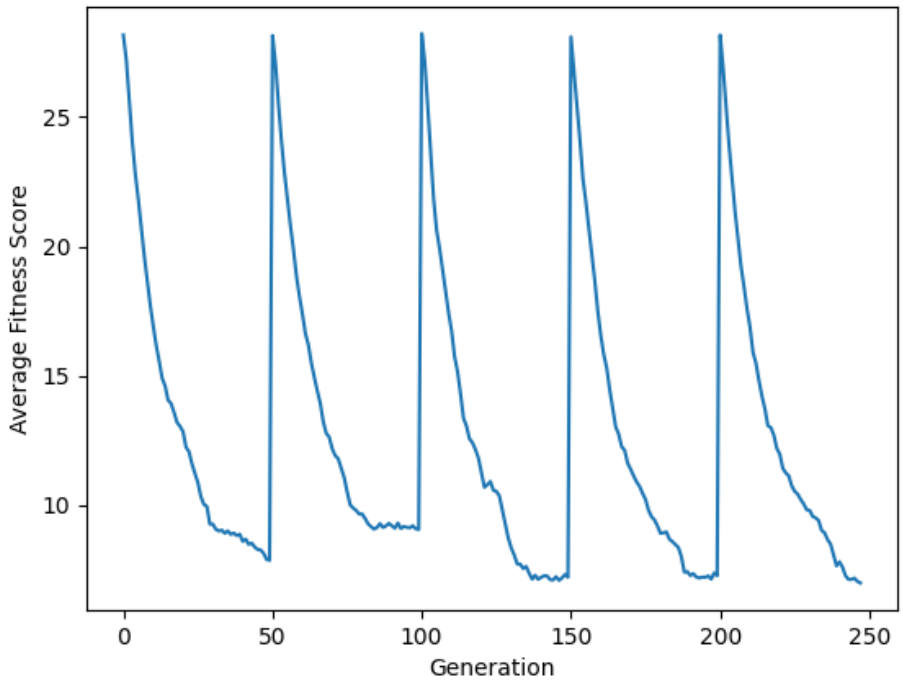


Figure 3.5

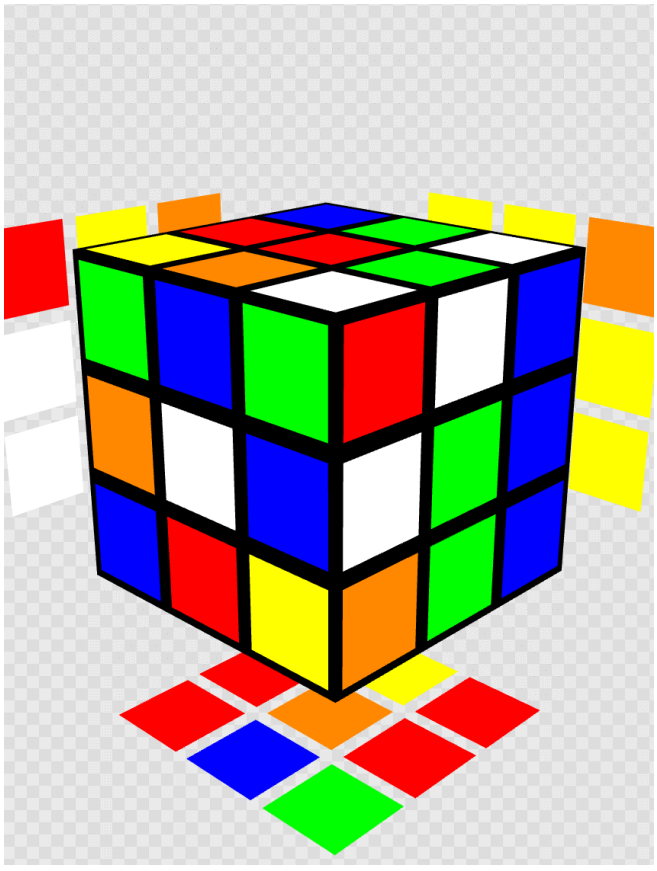


Figure 3.6

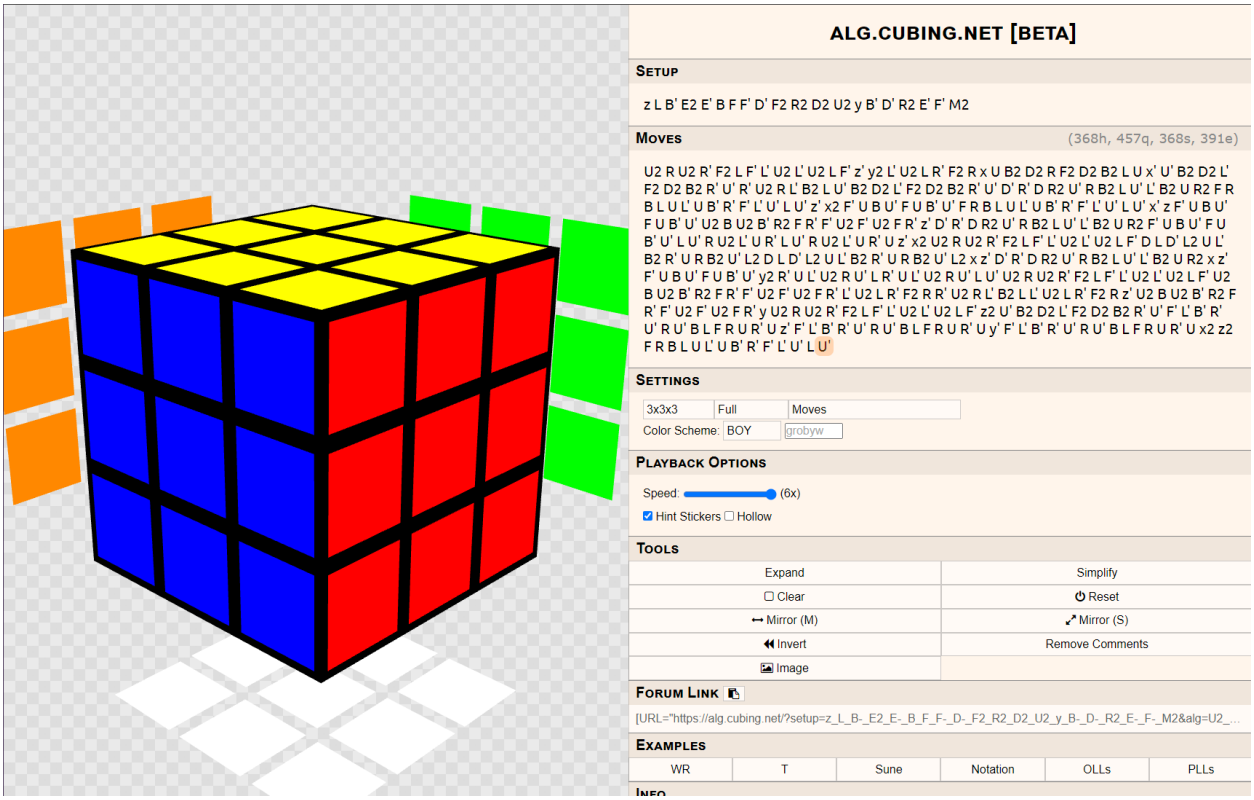


Figure 3.7

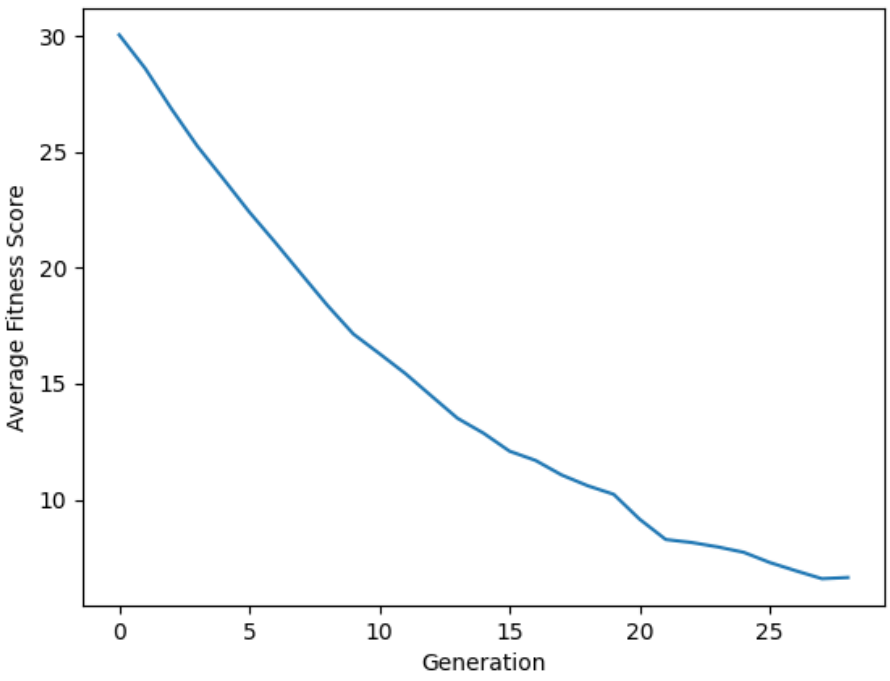
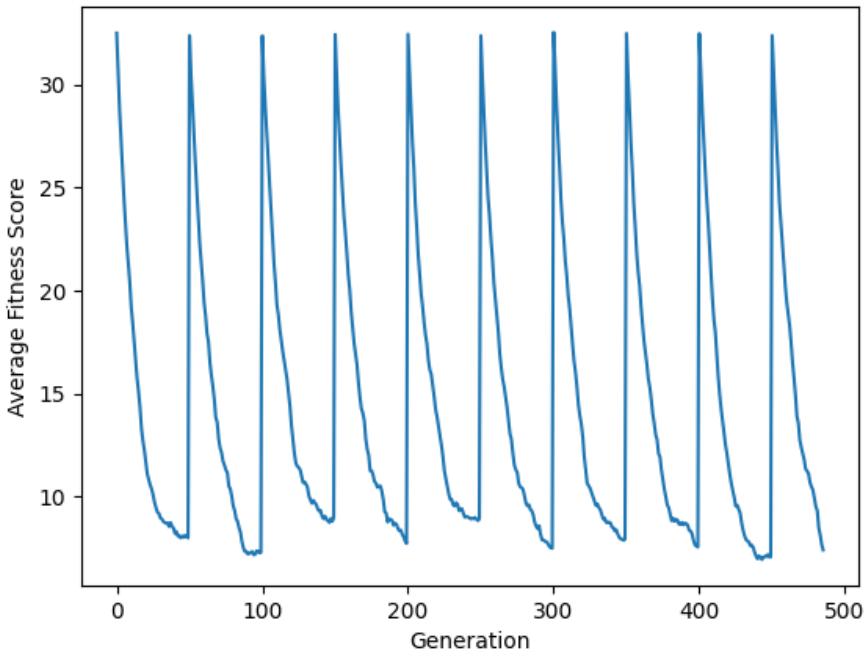


Figure 3.8



Demonstration: https://www.youtube.com/watch?v=g-6R0gPu-To&ab_channel=Ophion

II. P = 500, E = 30

Figure 3.9

	Max	Min	Gens
Run 1	30.914	9.344	481
Run 2	30.306	6.764	187
Run 3	31.744	7.11	131
Run 4	27.854	6.972	178
Run 5	33.254	7.434	27
Run 6	32.786	7.22	27
Run 7	35.736	6.596	89
Run 8	21.2	7.986	219
Avg	30.47425	7.42825	167.375
Stdv	4.115042	0.826733	136.1983

III. P = 1000, E = 10

Figure 3.10

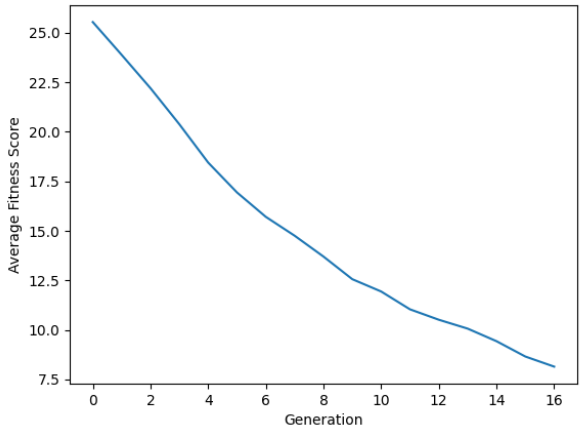
	Max	Min	Gens
Run 1	29.336	8.102	30
Run 2	29.43	8.192	31
Run 3	32.731	7.176	93
Run 4	31.965	7.249	43
Run 5	29.478	7.31	39
Run 6	22.495	7.437	73
Run 7	33.793	6.867	44
Run 8	28.376	7.256	95
Avg	29.7005	7.448625	56
Stdv	3.2594	0.431339	25.20417

IV. P = 1000, E = 30

Figure 3.11

	Max	Min	Gens
Run 1	31.137	7.411	126
Run 2	25.539	8.147	16
Run 3	33.748	7.586	121
Run 4	33.781	7.656	21
Run 5	32.615	7.192	277
Run 6	30.272	6.8	133
Run 7	28.217	6.784	31
Run 8	23.221	7.212	97
Avg	29.81625	7.3485	102.75
Stdv	3.628236	0.426484	80.26635

Figure 3.12



Final Table

Figure 3.13

GENS	e = 10	e = 30
p = 500	min = 28, max = 485, avg = 195.6, stdv = .3	min = 27, max = 481, avg = 167.4, stdv = 136.2
p = 1000	min = 30, max = 95, avg = 56, stdv = 25.2	min = 16, max = 277, avg = 102.8, stdv = 80.3

Genetic Algorithm + Wisdom of Crowds

V. P = 1500, E = 5

Figure 3.14

Figure 3.15

Figure 3.16

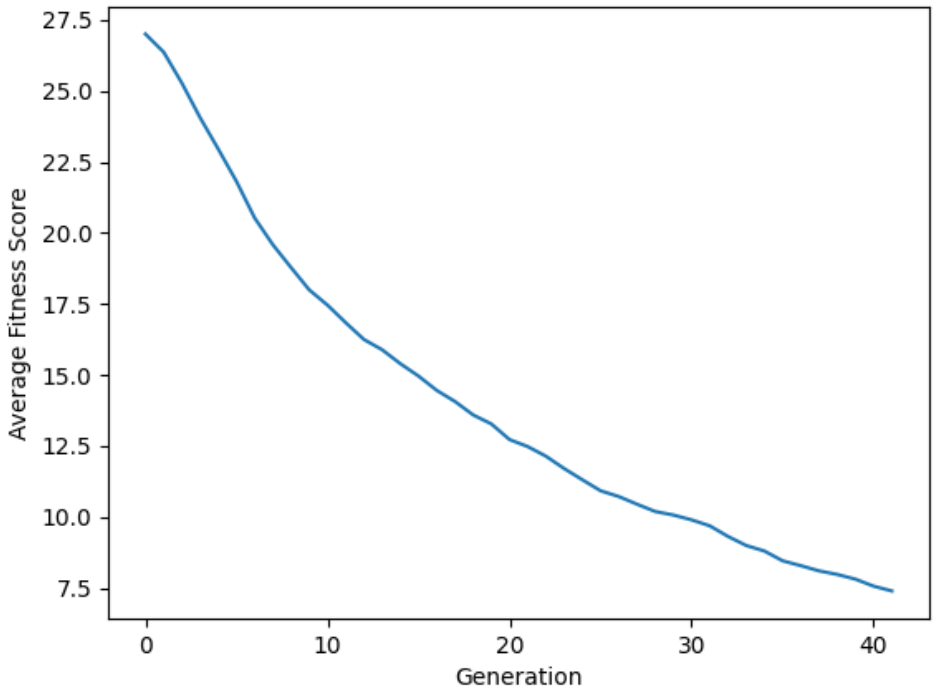


Figure 3.17

	Max	Min	Gens
Run 1	27.002	7.398	41
Run 2	27.036	8.27	43
Run 3	26.9	7.98	38
Avg	26.97933	7.882667	40.66667
Stdv	0.057789	0.362585	2.054805

4. Discussion

For the first dataset, we can see that performance varies wildly. Sometimes a solution is found in a handful of generations, while sometimes it takes upwards of 500. However, there are still solutions found. For this dataset, we also provided figures showing the terminal output of our program, evolution curves of multiple runs, and the solution being verified in the online simulator. Along with this, there is a link to a demonstration video showing the Cube being solved with a solution generated in one of these runs. This is specifically not done for subsequent datasets strictly due to time constraints.

However, there are figures showing the data collected for the following datasets. The second one varies wildly much like the first and the average is still fairly high. In comparison, the third dataset sees the average and standard deviation plummet, where the average generation count before a solution is found being 56. These runs were very quick, not taking nearly the time or processing power as the ones that came before. As for the fourth and final dataset of the GA, it sees an increase back to relatively high values for average and standard deviation. The final data table in Figure 3.13 shows us a serious outlier among these collections – the third dataset. The generational average was half of the second lowest. This tells us that a higher population count and lower elite percentage result in faster solutions. Because of this, we decided to exaggerate those features when running the WOC implementation.

For the WOC runs, we decided that to get a fairer dataset, the scramble moves that are enacted on the identity Cube needed to be predetermined instead of random. This would allow each expert to have an equal chance of reaching a solution fastest. Figure 3.14 shows this predetermined scramble sequence, as well the P and E values. For WOC, we decided to see how efficiently the program could find solutions by exaggerating the success trend found when running GA by itself. As such, the population size was set to 1500 and the elite percentage was set to 5. This does show an improvement even over the best dataset from GA alone. Although the minimum time to find a solution isn’t as low, the average is notable better.

In conclusion, these findings lead us to believe that the higher the population size and the lower the elite percentage (to an extent), the faster our program will find solutions. It also shows, like in the previous project, that using the wisdom of artificial crowds approach results in a more effective problem solver than that of a genetic algorithm by itself.

5. References

Alg.cubing.net. alg.cubing.net. (n.d.). Retrieved November 15, 2021, from https://alg.cubing.net/.

God's number is 20. God's Number is 20. (n.d.). Retrieved November 15, 2021, from https://www.cube20.org/.

Main page. SpeedSolving the Rubik's Cube - Speedsolving.com Wiki. (n.d.). Retrieved November 15, 2021, from https://www.speedsolving.com/wiki/index.php/Main_Page.

Online rubik's cube solver. Rubik's Cube Solver. (n.d.). Retrieved November 15, 2021, from https://rubiks-cube-solver.com/.

Rubiks-cube. npm. (n.d.). Retrieved November 15, 2021, from https://www.npmjs.com/package/rubiks-cube.

Thewca. (n.d.). *Thewca/tnoodle: Development for the official WCA Scramble Server*. GitHub. Retrieved November 15, 2021, from https://github.com/thewca/tnoodle.

Vaccari, R. (2020, July 7). *Solving Rubik's Cube using genetic algorithms*. Roberto Vaccari. Retrieved November 15, 2021, from https://robertovaccari.com/blog/2020_07_07_genetic_rubik/.

Visualization with python¶. Matplotlib. (n.d.). Retrieved October 22, 2021, from https://matplotlib.org/

Welcome to the World Cube Association. World Cube Association. (n.d.). Retrieved November 15, 2021, from https://www.worldcubeassociation.org/.

Wikimedia Foundation. (2021, October 14). *Rubik's Cube*. Wikipedia. Retrieved November 15, 2021, from https://en.wikipedia.org/wiki/Rubik%27s_Cube.