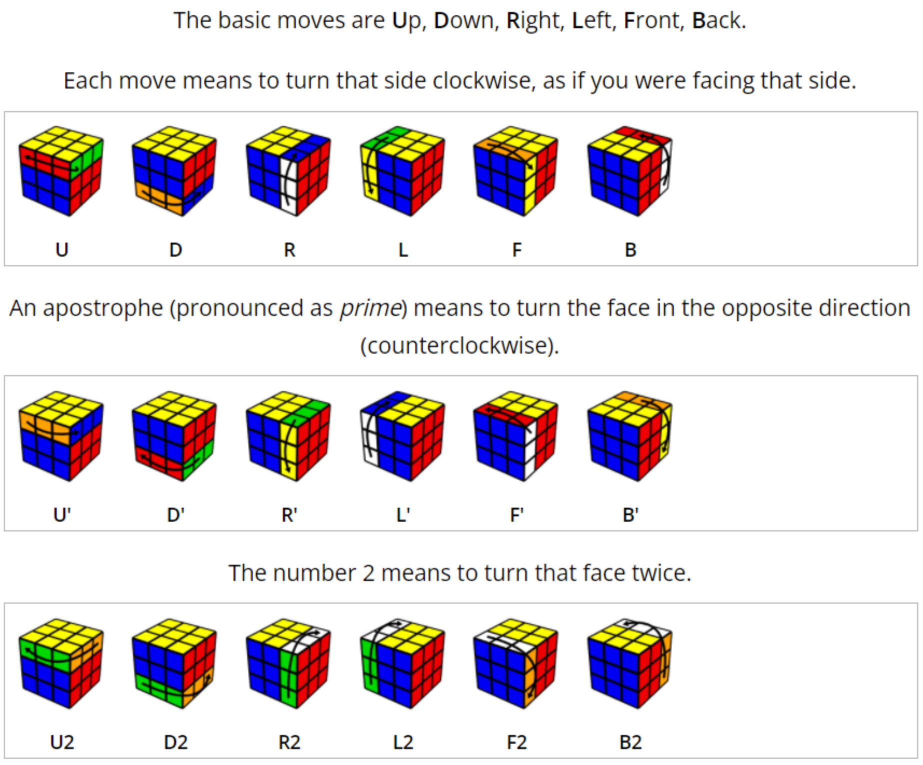
Project 6

Danny Foley
Computer Science and Engineering
Speed School of Engineering
University of Louisville, USA
danny.foley@louisville.edu

Introduction

In Project 6, the problem was to be chosen from a selection of NP-complete problems. For this project the problem of the NxNxN Rubik's Cube was chosen, more specifically the problem is to solve a 3x3x3 Rubik's Cube. There are many differences between the TSP problem and the Rubik's Cube problem. The first being that there are not a defined set of cities or in this case moves, to be chosen. There could, technically, be one hundred moves in the solution to the problem. However, it is known through other research that the minimum number of moves to solve a cube is 20[2]. Therefore we have limited the number of moves to be 20 in our program. When speaking about moves, in this paper, there are 18 separate moves. These moves are U, U', U2, D, D', D2, L, L', L2, R, R', R2, F, F', F2, B, B', B2. The meanings of these turns can be found in Figure 1[1].


The basic moves are **U**p, **D**own, **R**ight, **L**eft, **F**ront, **B**ack.

Each move means to turn that side clockwise, as if you were facing that side.

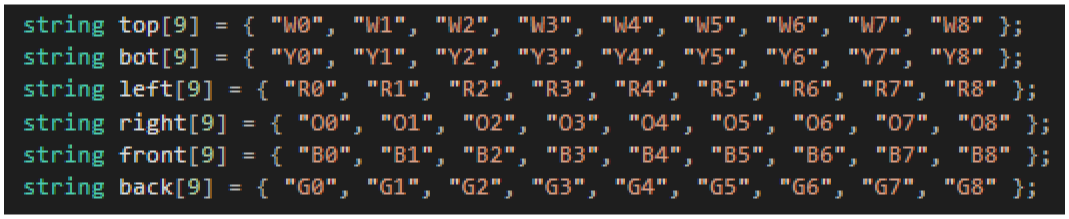
An apostrophe (pronounced as *prime*) means to turn the face in the opposite direction (counterclockwise).

The number 2 means to turn that face twice.

Figure 1: Showing the different moves possible.

Approach

For this assignment a genetic algorithm approach and wisdom of crowd's approach were to be implemented. Given that there is no need to make sure that the solution to the cube hits every move once, like TSP would, we can use a different crossover method and mutation method than from previous projects. Representation of the cube was broken down into 6 arrays of length 9 shown in Figure 2. These arrays represent each face of the cube and can be seen on a net diagram of the cube in Figure 3.



```
string top[9] = { "W0", "W1", "W2", "W3", "W4", "W5", "W6", "W7", "W8" };
string bot[9] = { "Y0", "Y1", "Y2", "Y3", "Y4", "Y5", "Y6", "Y7", "Y8" };
string left[9] = { "R0", "R1", "R2", "R3", "R4", "R5", "R6", "R7", "R8" };
string right[9] = { "O0", "O1", "O2", "O3", "O4", "O5", "O6", "O7", "O8" };
string front[9] = { "B0", "B1", "B2", "B3", "B4", "B5", "B6", "B7", "B8" };
string back[9] = { "G0", "G1", "G2", "G3", "G4", "G5", "G6", "G7", "G8" };
```

Figure 2: Showing the arrays of the cube in its solved state.

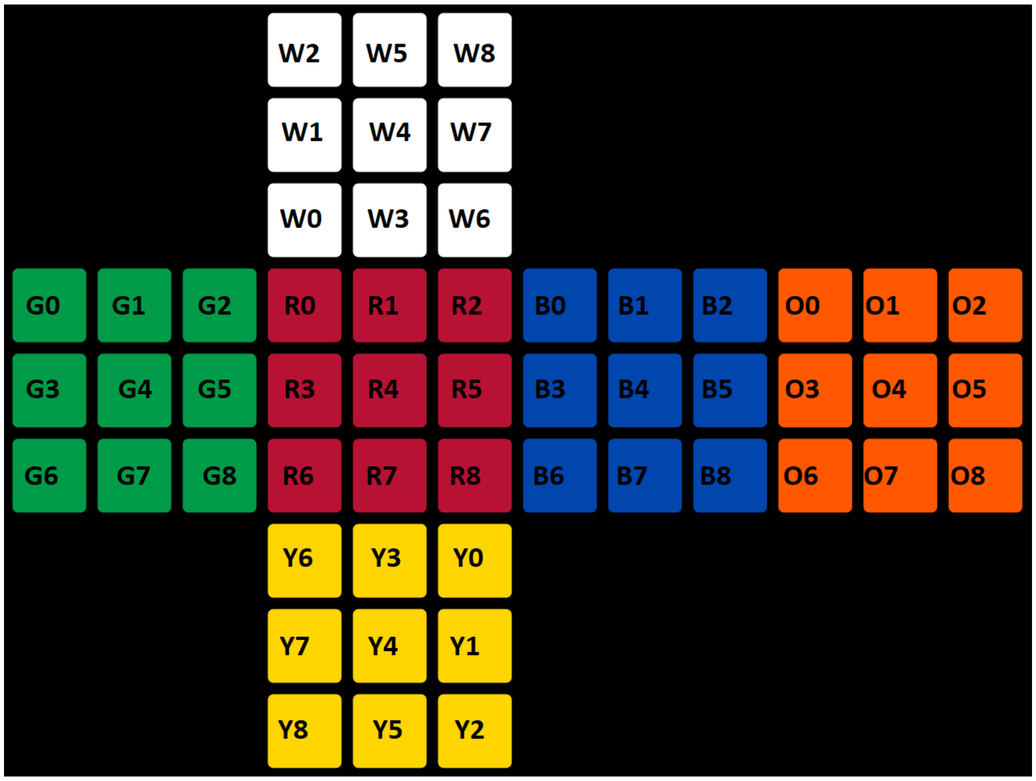

W2 W5 W8
W1 W4 W7
W0 W3 W6
G0 G1 G2 R0 R1 R2 B0 B1 B2 O0 O1 O2
G3 G4 G5 R3 R4 R5 B3 B4 B5 O3 O4 O5
G6 G7 G8 R6 R7 R8 B6 B7 B8 O6 O7 O8
Y6 Y3 Y0
Y7 Y4 Y1
Y8 Y5 Y2

Figure 3: Showing the solved representation of the cube

The scoring for the cube is determined when the set of moves are done to the cube trying to be solved. When a square does not match what it is supposed to in Figure 3, then there will be an additional point added to the score. Therefore, if all the squares are not where they are supposed to be the max score possible would be 48, because the middle of each face cannot move from its position. If all the squares are in the correct position, the score will be zero and the cube will have been solved. If the cube is solved somewhere before the end of the move set, then the cube is solved with a score of zero, even if there are more moves to be continued, once this is done the move set will be cut down to the smaller solution set. The score is also dependent on whether it meets certain guidelines, these guidelines are followed through a filtering process. Once filtered, the score of the cube will equal the previous score found and multiplied by the filtering score. This will help to push all the bad sets of moves down to the bottom of the population when looking at fitness.

Filtering

When the moves are created, there are some aspects that would make certain moves be less useful or completely useless in terms of helping to find the best 20 moves solution. There are certain moves for a face that when completed in order, would be the same as a different singular move. Examples:

- U -> U = U2
- U -> U’ = NO MOVE
- U -> U2 = U’
- U2 -> U2 = NO MOVE

On top of this, when a side is turned, there is always a side that is not changed. For example, when the right side of the cube is turned, the left side stays in the exact same state. This means that groups of moves containing opposing faces could make useless extra moves as well. For example:

- U -> D -> U = U2 -> D or D -> U2
- R2 -> L -> R = R’ -> L or L -> R’

Given this information that means there can never be two moves of the same face next to each other in the move set, also there can only ever be a pair of moves from opposing faces together in a move set like “U->F->B2->U->F” where the pair of F and B2 are surrounded by moves of different sides.

If two moves from the same face are found to be next to each other like “U2->U” then the move set will be given a filtering score of 2.

If there are more than two moves of opposing faces found together like “U2->D->U” then the move set will be given a filtering score of 3.

If nothing is found the filtering score remains as 1.

This filtering score is used to score the move set and also to help with making sure that a mutation does not create a bad move set.

The code for this filtering can be found in Figure 4 in the Figures section.

Crossover Method

The crossover method used is a single point crossover between two pairs of parents. The parents were chosen from the top 300 set of moves. The best parents from each pair are joined together and then the worst parents from each pair are joined together. This will result in the creation of four children. Having the worst parents cross is to help see if there are some better moves that are not being found based off the first part of the parents moves. Code can be found in Figure 5 in the below Figures section.

Mutation Method

The mutation method is a swap mutation, where a single move out of the set of moves is chosen and then the move is replaced with a different move, whether that is a move for the same face or a different face of the cube. Once the move is selected and changed, the filtering algorithm is used to check that the move set does not go against any of the filtering described previously. If the filtering finds nothing wrong, then the mutation is finalized. Code for the mutation method can be found in Figure 6 in the below Figures section.

Genetic Algorithm

The main algorithm was used with an iteration cap of 200, meaning that after 200 generations of no better move set found, the generations would no longer be created. If there is a solution found with a score of zero, the algorithm will also stop (regardless if the filtering score is 1 or not). After creating a random population, of size 1000, the population is ranked and then a crossover and mutation are done on the population. If a better score than a previous population is found or if the score is the same, then the population is added to the resulting set of populations, so that they can be viewed later. The code can be found in Figure 7 of the Figures section.

Wisdom of Crowds Algorithm

The algorithm created uses the results from the genetic algorithm and processes through the top 10 percent of all the resulting populations. The best two moves are found based on the score of the move set. Then the next best pair of moves will be chosen. This will continue until the full 20 moves are created and a resulting set of moves will be returned. The code can be found in Figure 8 of the figures section.

UI

The UI is found within the terminal and shows a simple printout of the array shown in Figure 1. When the genetic algorithm is complete, a printout of the best moves set and its resulting cube array will be shown. Once the wisdom of crowds is complete, a printout of the cube and its move set solution will be shown as well. The resulting output can be found in Figure 9 and 10 of the Figures section.

Results

The results for different two shuffled cubes is shown below.

CUBE SHUFFLE OF 4 MOVES	GA TIME (s)	GENERATIONS	GA SCORE	WOC TIME (ms)	GA + WOC TIME (s)	GA + WOC SCORE
RUN 1	1404	115	0	1034	1405	46
RUN 2	732	61	0	517	732	48
RUN 3	14	1	0	279	14	48

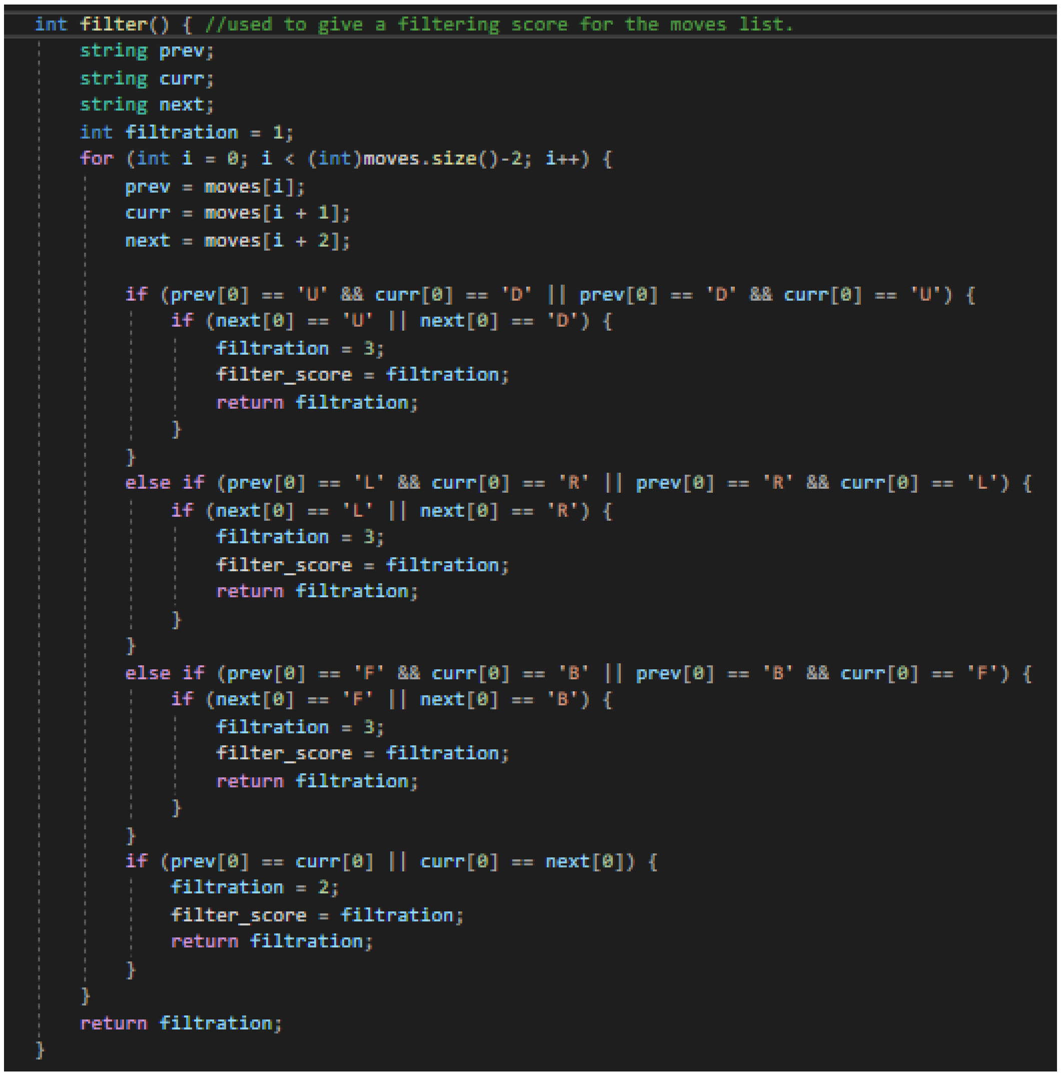
CUBE SHUFFLE OF 30 MOVES	GA TIME (s)	GENERATIONS	GA SCORE	WOC TIME (ms)	GA + WOC TIME (s)	GA + WOC SCORE
RUN 1	2863	200	28	196	2864	196
RUN 2	4329	313	29	804	4330	135
RUN 3	4289	309	29	554	4289	45

Discussion

Based on the results, it can be said that there is no good use of the genetic algorithm and wisdom of crowd's alone when it comes to solving a Rubik's cube. Generally speaking, when compared to the TSP problem there is not as clean of a way to represent a score for a result. When working with a Rubik's cube, there is no real way to see that a certain set of moves is the best other than if the cube gets solved. Even with a single quarter turn of a face on the cube the score for the cube will change from 0 to 20. A single move at the end of the solution is worth more of a score difference than any other move. This kind of complexity and difference makes it much different than a TSP problem. In many instances, a score of zero was not found by the end of the iterations, even with an increase in iterations there was no found solution that made a score of zero when dealing with more than 4 moves. Usually if there was a score of zero found, it would be based solely on luck as the genetic algorithm seems to have a hard time getting anywhere close to a score of zero. With the difficulty of finding a solution with GA, there then becomes a difficulty of finding a solution with WoC, since the crowd is not a good group of subjects to pull from. Compared to the TSP problem there is no fast increase in score, most of the scores are random within a range of 30-40 and then sometimes a score of less than 30 will appear, with no real progress being made. When a cube is scrambled by more than a few moves, it becomes much more difficult to find the final solution as can be seen in the results between a cube shuffled by 4 moves and a cube shuffled by 30 moves. If given an unlimited amount of time, there also tends to be a local minimum, where the best score is the same and never gets better throughout the ending generations.

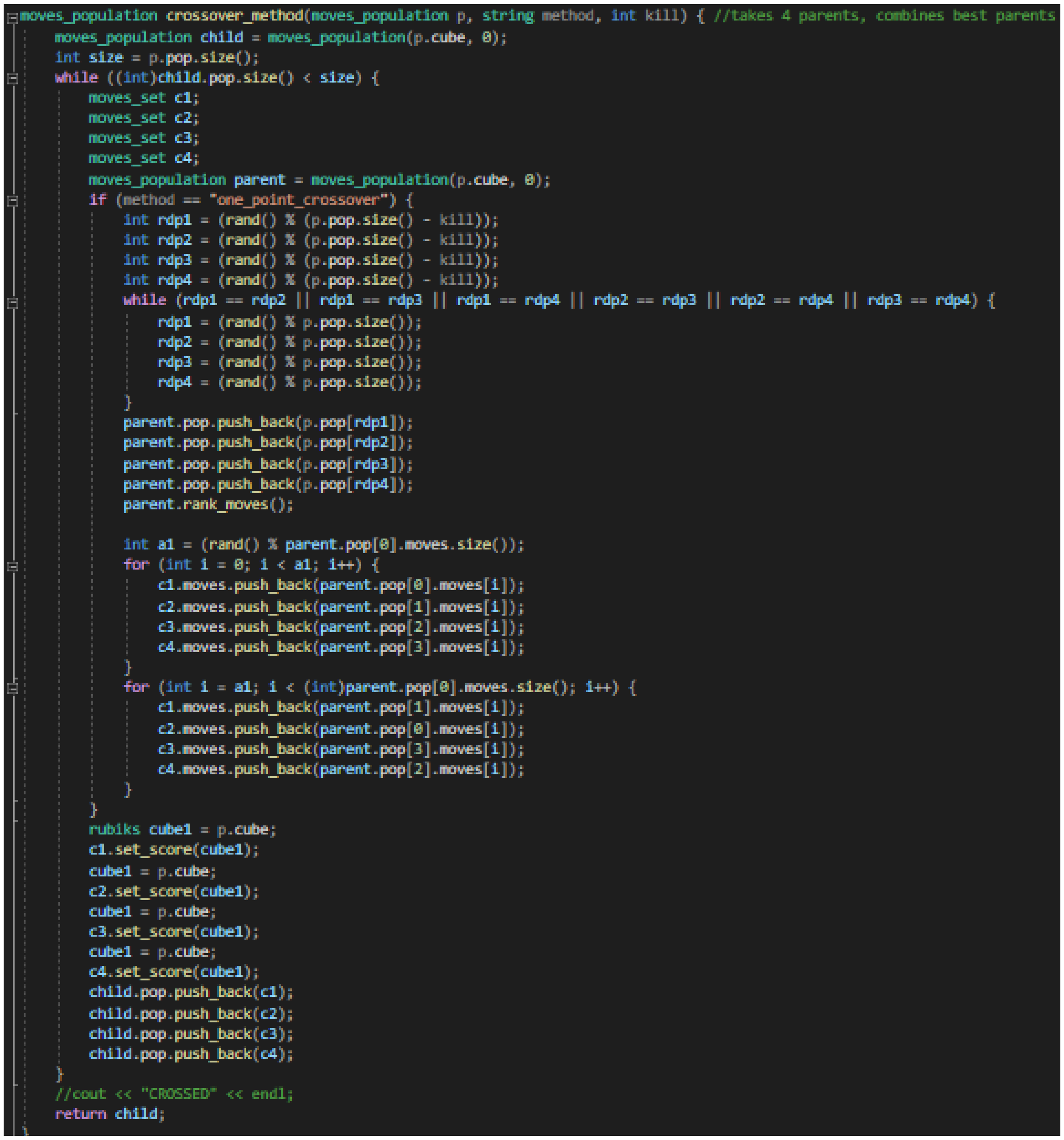
After less than successful approaches were taken, more research into solving a Rubik's Cube with artificial intelligence was required. A research paper[3] was able to solve a Rubik's cube using a genetic algorithm and group theory, where certain groups of moves were taken out similar to that of the filtering done in this project. After having done that research, the program changed into what it is now. However, the results with this program are no where near the desired results, nor are they close to the results of the research paper. Based on the opinions of others in the coding sphere[4], it seems that GA is not a very suitable method by itself, for solving a Rubik's cube.

Figures



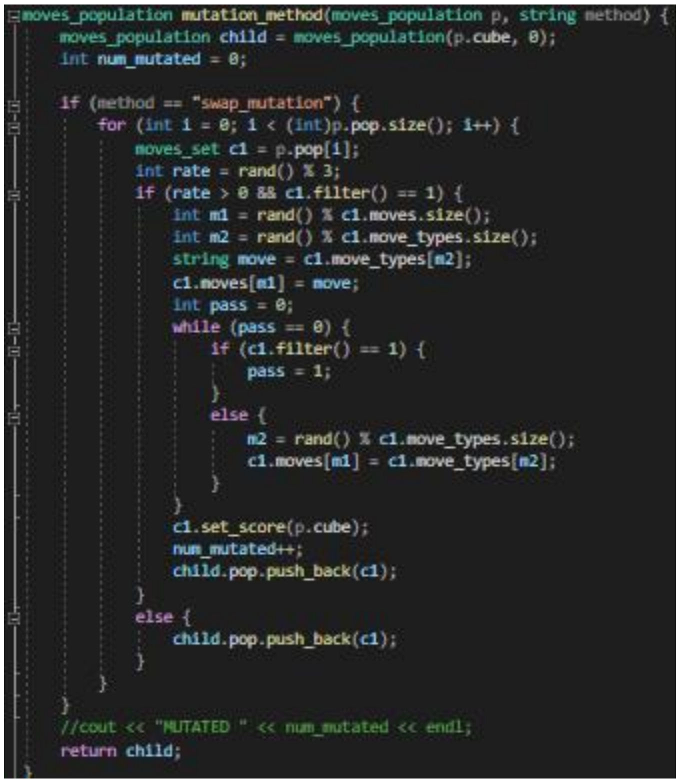
```
int filter() { //used to give a filtering score for the moves list.
    string prev;
    string curr;
    string next;
    int filtration = 1;
    for (int i = 0; i < (int)moves.size()-2; i++) {
        prev = moves[i];
        curr = moves[i + 1];
        next = moves[i + 2];

        if (prev[0] == 'U' && curr[0] == 'D' || prev[0] == 'D' && curr[0] == 'U') {
            if (next[0] == 'U' || next[0] == 'D') {
                filtration = 3;
                filter_score = filtration;
                return filtration;
            }
        }
        else if (prev[0] == 'L' && curr[0] == 'R' || prev[0] == 'R' && curr[0] == 'L') {
            if (next[0] == 'L' || next[0] == 'R') {
                filtration = 3;
                filter_score = filtration;
                return filtration;
            }
        }
        else if (prev[0] == 'F' && curr[0] == 'B' || prev[0] == 'B' && curr[0] == 'F') {
            if (next[0] == 'F' || next[0] == 'B') {
                filtration = 3;
                filter_score = filtration;
                return filtration;
            }
        }
        if (prev[0] == curr[0] || curr[0] == next[0]) {
            filtration = 2;
            filter_score = filtration;
            return filtration;
        }
    }
    return filtration;
}
```

Figure 4: Showing the filtering code



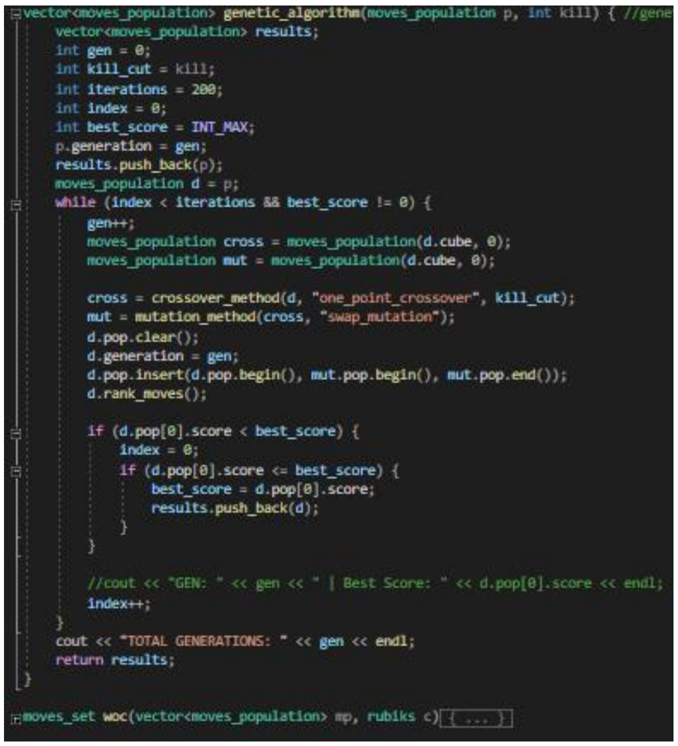
```
moves_population crossover_method(moves_population p, string method, int kill) { //takes 4 parents, combines best parents
    moves_population child = moves_population(p.cube, 0);
    int size = p.pop.size();
    while ((int)child.pop.size() < size) {
        moves_set c1;
        moves_set c2;
        moves_set c3;
        moves_set c4;
        moves_population parent = moves_population(p.cube, 0);
        if (method == "one_point_crossover") {
            int rdp1 = (rand() % (p.pop.size() - kill));
            int rdp2 = (rand() % (p.pop.size() - kill));
            int rdp3 = (rand() % (p.pop.size() - kill));
            int rdp4 = (rand() % (p.pop.size() - kill));
            while (rdp1 == rdp2 || rdp1 == rdp3 || rdp1 == rdp4 || rdp2 == rdp3 || rdp2 == rdp4 || rdp3 == rdp4) {
                rdp1 = (rand() % p.pop.size());
                rdp2 = (rand() % p.pop.size());
                rdp3 = (rand() % p.pop.size());
                rdp4 = (rand() % p.pop.size());
            }
            parent.pop.push_back(p.pop[rdp1]);
            parent.pop.push_back(p.pop[rdp2]);
            parent.pop.push_back(p.pop[rdp3]);
            parent.pop.push_back(p.pop[rdp4]);
            parent.rank_moves();

            int a1 = (rand() % parent.pop[0].moves.size());
            for (int i = 0; i < a1; i++) {
                c1.moves.push_back(parent.pop[0].moves[i]);
                c2.moves.push_back(parent.pop[1].moves[i]);
                c3.moves.push_back(parent.pop[2].moves[i]);
                c4.moves.push_back(parent.pop[3].moves[i]);
            }
            for (int i = a1; i < (int)parent.pop[0].moves.size(); i++) {
                c1.moves.push_back(parent.pop[1].moves[i]);
                c2.moves.push_back(parent.pop[0].moves[i]);
                c3.moves.push_back(parent.pop[3].moves[i]);
                c4.moves.push_back(parent.pop[2].moves[i]);
            }
        }
        rubiks cube1 = p.cube;
        c1.set_score(cube1);
        cube1 = p.cube;
        c2.set_score(cube1);
        cube1 = p.cube;
        c3.set_score(cube1);
        cube1 = p.cube;
        c4.set_score(cube1);
        child.pop.push_back(c1);
        child.pop.push_back(c2);
        child.pop.push_back(c3);
        child.pop.push_back(c4);
    }
    //cout << "CROSSED" << endl;
    return child;
```

Figure 5: Showing the crossover method



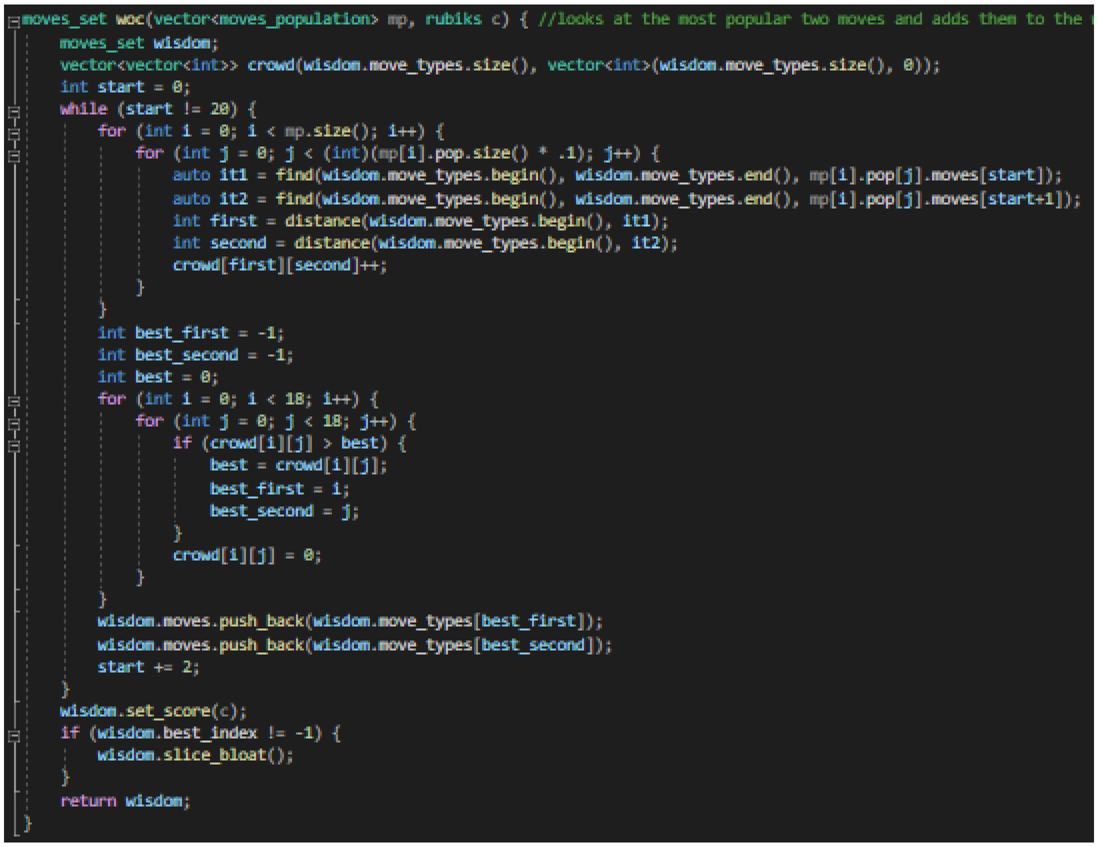
```
moves_population mutation_method(moves_population p, string method) {
    moves_population child = moves_population(p.cube, 0);
    int num_mutated = 0;

    if (method == "swap_mutation") {
        for (int i = 0; i < (int)p.pop.size(); i++) {
            moves_set c1 = p.pop[i];
            int rate = rand() % 3;
            if (rate > 0 && c1.filter() == 1) {
                int m1 = rand() % c1.moves.size();
                int m2 = rand() % c1.move_types.size();
                string move = c1.move_types[m2];
                c1.moves[m1] = move;
                int pass = 0;
                while (pass == 0) {
                    if (c1.filter() == 1) {
                        pass = 1;
                    }
                    else {
                        m2 = rand() % c1.move_types.size();
                        c1.moves[m1] = c1.move_types[m2];
                    }
                }
                c1.set_score(p.cube);
                num_mutated++;
                child.pop.push_back(c1);
            }
            else {
                child.pop.push_back(c1);
            }
        }
    }
    //cout << "MUTATED " << num_mutated << endl;
    return child;
}
```

Figure 6: Showing the mutation method



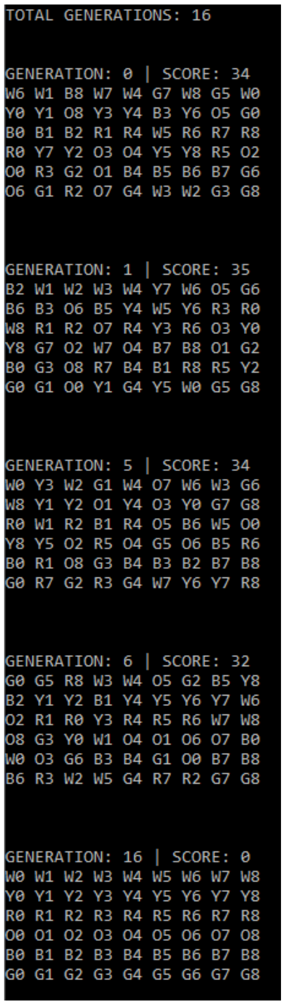
```
vector<moves_population> genetic_algorithm(moves_population p, int kill) { //gene
    vector<moves_population> results;
    int gen = 0;
    int kill_cut = kill;
    int iterations = 200;
    int index = 0;
    int best_score = INT_MAX;
    p.generation = gen;
    results.push_back(p);
    moves_population d = p;
    while (index < iterations && best_score != 0) {
        gen++;
        moves_population cross = moves_population(d.cube, 0);
        moves_population mut = moves_population(d.cube, 0);

        cross = crossover_method(d, "one_point_crossover", kill_cut);
        mut = mutation_method(cross, "swap_mutation");
        d.pop.clear();
        d.generation = gen;
        d.pop.insert(d.pop.begin(), mut.pop.begin(), mut.pop.end());
        d.rank_moves();

        if (d.pop[0].score < best_score) {
            index = 0;
            if (d.pop[0].score <= best_score) {
                best_score = d.pop[0].score;
                results.push_back(d);
            }
        }

        //cout << "GEN: " << gen << " | Best Score: " << d.pop[0].score << endl;
        index++;
    }
    cout << "TOTAL GENERATIONS: " << gen << endl;
    return results;
}

moves_set woc(vector<moves_population> mp, rubiks c){ ... }
```

Figure 7: Showing the genetic algorithm method



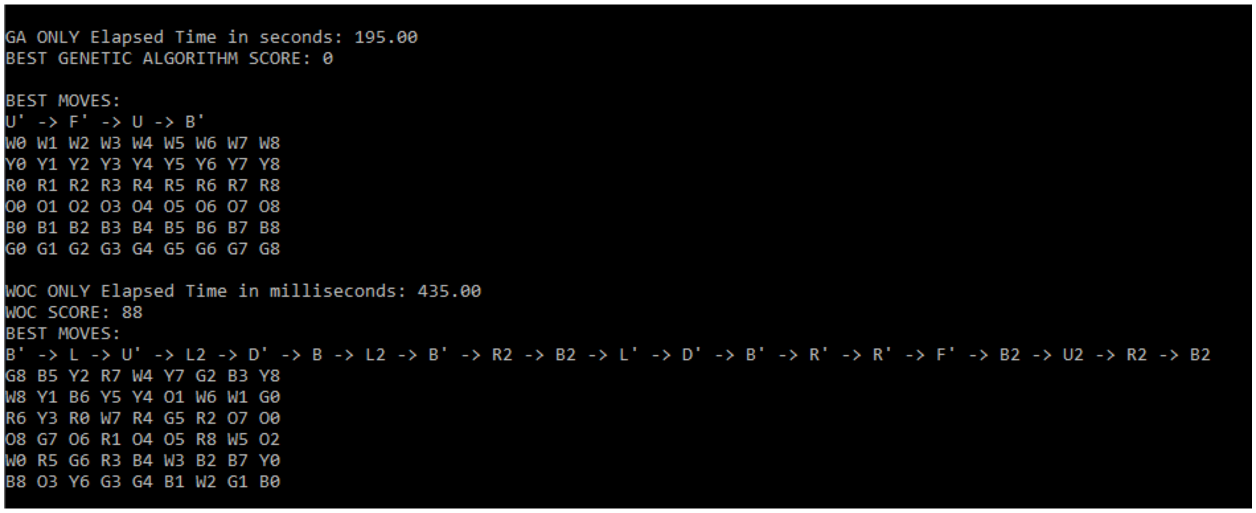
```cpp
moves_set woc(vector<moves_population> mp, rubiks c) { //looks at the most popular two moves and adds them to the
    moves_set wisdom;
    vector<vector<int>> crowd(wisdom.move_types.size(), vector<int>(wisdom.move_types.size(), 0));
    int start = 0;
    while (start != 20) {
        for (int i = 0; i < mp.size(); i++) {
            for (int j = 0; j < (int)(mp[i].pop.size() * .1); j++) {
                auto it1 = find(wisdom.move_types.begin(), wisdom.move_types.end(), mp[i].pop[j].moves[start]);
                auto it2 = find(wisdom.move_types.begin(), wisdom.move_types.end(), mp[i].pop[j].moves[start+1]);
                int first = distance(wisdom.move_types.begin(), it1);
                int second = distance(wisdom.move_types.begin(), it2);
                crowd[first][second]++;
            }
        }
        int best_first = -1;
        int best_second = -1;
        int best = 0;
        for (int i = 0; i < 18; i++) {
            for (int j = 0; j < 18; j++) {
                if (crowd[i][j] > best) {
                    best = crowd[i][j];
                    best_first = i;
                    best_second = j;
                }
                crowd[i][j] = 0;
            }
        }
        wisdom.moves.push_back(wisdom.move_types[best_first]);
        wisdom.moves.push_back(wisdom.move_types[best_second]);
        start += 2;
    }
    wisdom.set_score(c);
    if (wisdom.best_index != -1) {
        wisdom.slice_bloat();
    }
    return wisdom;
}
```

Figure 8: Showing the Wisdom of Crowds method



```
TOTAL GENERATIONS: 16


GENERATION: 0 | SCORE: 34
W6 W1 B8 W7 W4 G7 W8 G5 W0
Y0 Y1 O8 Y3 Y4 B3 Y6 O5 G0
B0 B1 B2 R1 R4 W5 R6 R7 R8
R0 Y7 Y2 O3 O4 Y5 Y8 R5 O2
O0 R3 G2 O1 B4 B5 B6 B7 G6
O6 G1 R2 O7 G4 W3 W2 G3 G8



GENERATION: 1 | SCORE: 35
B2 W1 W2 W3 W4 Y7 W6 O5 G6
B6 B3 O6 B5 Y4 W5 Y6 R3 R0
W8 R1 R2 O7 R4 Y3 R6 O3 Y0
Y8 G7 O2 W7 O4 B7 B8 O1 G2
B0 G3 O8 R7 B4 B1 R8 R5 Y2
G0 G1 O0 Y1 G4 Y5 W0 G5 G8



GENERATION: 5 | SCORE: 34
W0 Y3 W2 G1 W4 O7 W6 W3 G6
W8 Y1 Y2 O1 Y4 O3 Y0 G7 G8
R0 W1 R2 B1 R4 O5 B6 W5 O0
Y8 Y5 O2 R5 O4 G5 O6 B5 R6
B0 R1 O8 G3 B4 B3 B2 B7 B8
G0 R7 G2 R3 G4 W7 Y6 Y7 R8



GENERATION: 6 | SCORE: 32
G0 G5 R8 W3 W4 O5 G2 B5 Y8
B2 Y1 Y2 B1 Y4 Y5 Y6 Y7 W6
O2 R1 R0 Y3 R4 R5 R6 W7 W8
O8 G3 Y0 W1 O4 O1 O6 O7 B0
W0 O3 G6 B3 B4 G1 O0 B7 B8
B6 R3 W2 W5 G4 R7 R2 G7 G8



GENERATION: 16 | SCORE: 0
W0 W1 W2 W3 W4 W5 W6 W7 W8
Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
R0 R1 R2 R3 R4 R5 R6 R7 R8
O0 O1 O2 O3 O4 O5 O6 O7 O8
B0 B1 B2 B3 B4 B5 B6 B7 B8
G0 G1 G2 G3 G4 G5 G6 G7 G8
```

Figure 9: Showing the generational improvements



```
GA ONLY Elapsed Time in seconds: 195.00
BEST GENETIC ALGORITHM SCORE: 0

BEST MOVES:
U' -> F' -> U -> B'
W0 W1 W2 W3 W4 W5 W6 W7 W8
Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
R0 R1 R2 R3 R4 R5 R6 R7 R8
O0 O1 O2 O3 O4 O5 O6 O7 O8
B0 B1 B2 B3 B4 B5 B6 B7 B8
G0 G1 G2 G3 G4 G5 G6 G7 G8

WOC ONLY Elapsed Time in milliseconds: 435.00
WOC SCORE: 88
BEST MOVES:
B' -> L -> U' -> L2 -> D' -> B -> L2 -> B' -> R2 -> B2 -> L' -> D' -> B' -> R' -> R' -> F' -> B2 -> U2 -> R2 -> B2
G8 B5 Y2 R7 W4 Y7 G2 B3 Y8
W8 Y1 B6 Y5 Y4 O1 W6 W1 G0
R6 Y3 R0 W7 R4 G5 R2 O7 O0
O8 G7 O6 R1 O4 O5 R8 W5 O2
W0 R5 G6 R3 B4 W3 B2 B7 Y0
B8 O3 Y6 G3 G4 B1 W2 G1 B0
```

Figure 10: Showing the results of the program.

References

[1] https://jperm.net/3x3/moves

[2] http://www.cube20.org/

[3] https://link.springer.com/article/10.1007/s42979-019-0054-4

[4] https://stackoverflow.com/questions/36068550/rubiks-cube-genetic-algorithm-solver