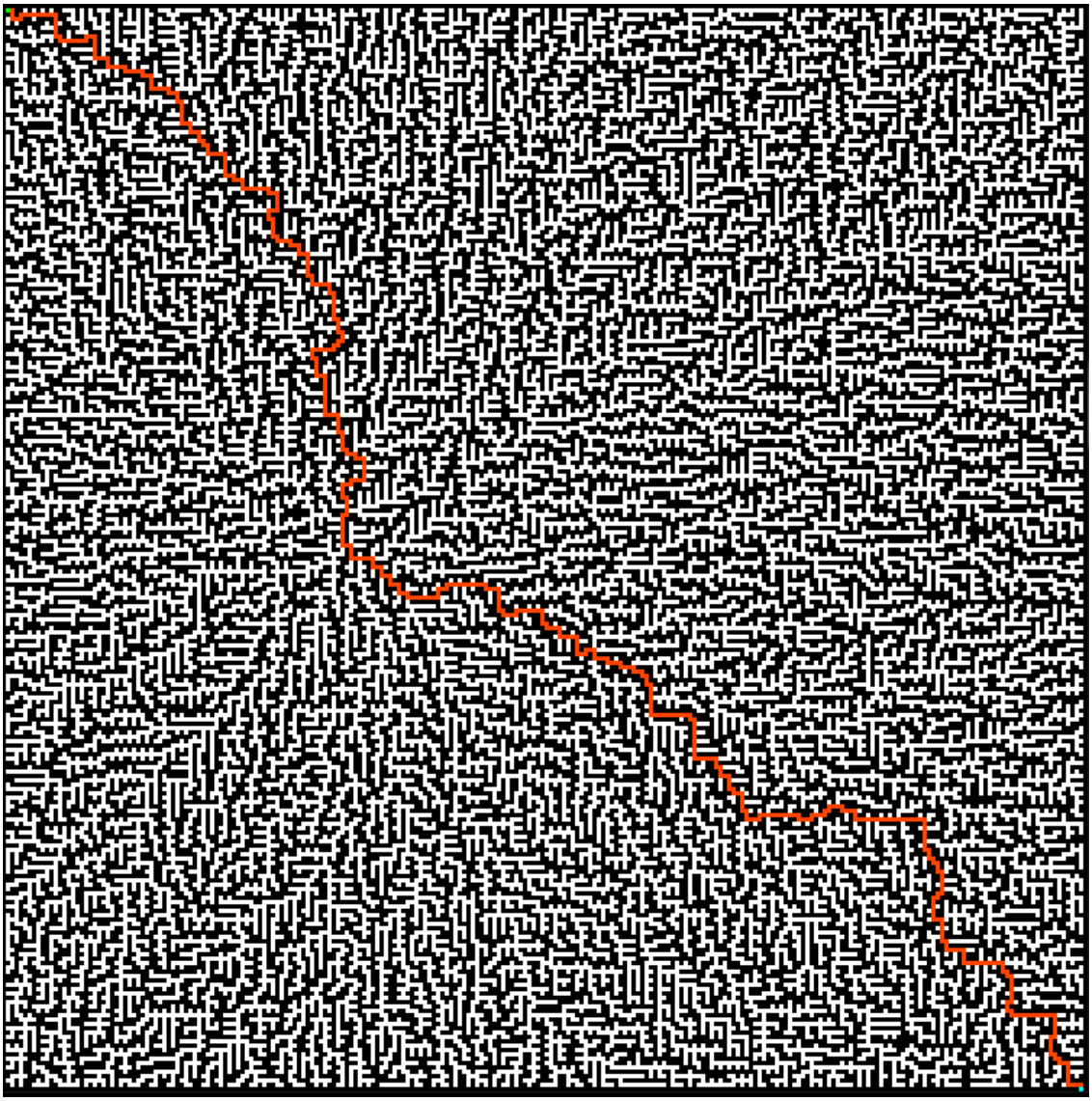
Report for Seminar Assignment 1

Luka Uranič, Nejc Uršič

19.11.2022



1 Introduction

In the first seminar assignment, out goal was to use genetic algorithm to find a path out of a maze, represented as a vector of strings, where # character represents a wall, . represents empty space, and S and E represent the starting and ending points. You can move through the maze in four directions, left, right, up, and down. Out task was to create a function that will be able to find path as short as possible out of any maze represented in such a way.

2 Task 1- Representation

Maze is represented as a vector/array of Strings and then gets transformed into one dimensional integer array that represents the maze matrix. Where # is stored as -1, . as 0, S as 1, E as 2 and T as 3. Start, end and positions of tressures are stored as Vec2 object which is a vector with two int values (maze coordinates).

```
height = mazeString.length;
width = mazeString[0].length();
int[] maze = new int[width * height];
Vec2 start = null, end = null;
Vec2[] tressures = null;
int tressuresCount = 0;
int maxChromosomeLength = width * height - 1; // moves is one less than visited points
for(int y = 0; y < height; y++) {
    String mazeRow = mazeString[y];
    for(int x = 0; x < width; x++) {
        if(mazeRow.charAt(x) == '#') {
            maze[x + y * width] = -1;
            maxChromosomeLength--;
        }else if(mazeRow.charAt(x) == '.') {
            maze[x + y * width] = 0;
        }else if(mazeRow.charAt(x) == 'S') {
            maze[x + y * width] = 1;
            start = new Vec2(x, y);
        }else if(mazeRow.charAt(x) == 'E') {
            maze[x + y * width] = 2;
            end = new Vec2(x, y);
        }else if(mazeRow.charAt(x) == 'T') {
            maze[x + y * width] = 3;
            tressuresCount++;
        }
    }
}
if(tressuresCount > 0) {
    tressures = new Vec2[tressuresCount];
    int i = 0;
    for(int y = 0; y < height; y++) {
        for(int x = 0; x < width; x++) {
            if(maze[x + y * width] == 3) {
                tressures[i] = new Vec2(x, y);
                i++;
            }
        }
    }
    maxChromosomeLength = width * height - 1;
}
```

Solution is a sequence of moves to get from start position to end position (example: LUDU-ULDR). The solution representation is array of Integers where 0 means left, 1 means right, 2 means up and 3 means down.

Fitness function calculates the fitness of a Chromosome. Chromosomes with a better fitness are more likely to survive and reproduce. Fitness function maximum is when distance to end position is 0, never goes through walls, the path is shortest possible and all thresures are on the path.

```java
public void calculateFitness() {
    getPath();
    if(path == null) {
        System.out.println("Path is null");
        fitness = Integer.MIN_VALUE;
    }
    int lastX = path[path.length-1].x;
    int lastY = path[path.length-1].y;
    int distanceToEnd = Math.abs(lastX - end.x) + Math.abs(lastY - end.y);

    int numberOfTimesThroughWall = 0;
    int numberOfThressures = 0;
    List<Vec2> tressuresOnPath = new ArrayList<>();

    for(int i = 0; i < path.length; i++) {
        if(path[i].x < 0 || path[i].y < 0 || path[i].x >= width || path[i].y >= height
                || maze[path[i].x + path[i].y * width] == -1) { // wall or out of maze
            numberOfTimesThroughWall++;
            continue;
        }
        if(maze[path[i].x + path[i].y * width] == 3) { // tressure
            boolean already = false;
            for(int j = 0; j < tressuresOnPath.size(); j++) {
                if(path[i].x == tressuresOnPath.get(j).x && path[i].y == tressuresOnPath.get(j).y) {
                    already = true;
                }
            }
            if(!already) {
                numberOfThressures++;
                tressuresOnPath.add(new Vec2(path[i].x, path[i].y));
            }
        }
    }


    int wallPenalty = maxChromosomeLength;
    int distanceToEndPenalty = maxChromosomeLength/4;
    int tressureReward = maxChromosomeLength * 2;
    fitness =
            -distanceToEnd * distanceToEndPenalty
            -numberOfTimesThroughWall * wallPenalty
            -path.length
            +numberOfThressures * tressureReward;
}
```

3 Task 2,3 - Selection, crossover, mutations, initial population

Our implementation of genetic algorithm (for solving maze problem):

3.1 Selection

3.1.1 Random selection

Randomly selects two parents (fitness doesn't matter):

```
public void selectionRandomly() {
    selectionIndexes = new Vec2[population.size()];

    int parent1 = 0, parent2 = 0;
    for(int i = 0; i < selectionIndexes.length; i++) {
        parent1 = parent2;
        while(parent1 == parent2) {
            parent1 = rand.nextInt(population.size());
            parent2 = rand.nextInt(population.size());
        }
        selectionIndexes[i] = new Vec2(parent1, parent2);
    }
}
```

3.1.2 Linearly biased selection

Randomly selects two parents, but parents with greater fitness have a bigger probabiliy to get selected (linearly biased distrobution):

```
public void selectionLinearlyBiased() {
    if(population.size() < 3) {
        selectionRandomly();
        return;
    }
    int startIndex = population.size() * elitism / 100;
    selectionIndexes = new Vec2[population.size()-startIndex];
    int parent1 = 0, parent2 = 0;
    int sumFrom0ToN = (population.size()) * (population.size()-1) / 2 - 1;
    for(int i = 0; i < selectionIndexes.length; i++) {
        parent1 = parent2;
        while(parent1 == parent2) {
            parent1 = population.size()-1-(int)(1/2+Math.sqrt(1+8*rand.nextInt(sumFrom0ToN))/2);
            parent2 = population.size()-1-(int)(1/2+Math.sqrt(1+8*rand.nextInt(sumFrom0ToN))/2);
        }
        selectionIndexes[i] = new Vec2(parent1, parent2);
    }
}
```

3.1.3 Higher order biased selection

Randomly selects two parents, but parents with greater fitness have a bigger probabiliy to get selected (higer order biased distrobution). We didn't implement it, but different probability distributions might improve the algorithm.

3.2 Crossover

3.2.1 Pick a random parent

Picks a random parent to become a new child:

```
public void simpleRandomCrossover() {
    int startIndex = population.size() * elitism / 100;
    List<Chromosome> children = new ArrayList<>();
    for(int i = 0; i < startIndex; i++) {
        children.add(population.get(i));
    }
    for(int i = 0; i < selectionIndexes.length; i++) {
        Chromosome parent1 = population.get(selectionIndexes[i].x);
        Chromosome parent2 = population.get(selectionIndexes[i].y);
        int randParent = rand.nextInt(2);
        if(randParent == 0) {
            children.add(new Chromosome(parent1.moves, maxChromosomeLength, maze, start, end,
                    tressures, width, height));
        }else {
            children.add(new Chromosome(parent2.moves, maxChromosomeLength, maze, start, end,
                    tressures, width, height));
        }

    }
    population = children;
}
```

3.2.2 Intersections crossover

Go through all intersections between two parents paths than select a random intersection and build a new path like this: randomly select a path to the intersection from one of the parents, randomly select the path after intersection from one of the parents. With this method you have to be careful, because not all the lengths of path are the same so you can easily get unexpected results. We implemented this method, but was computationaly to slow and didn't work better than the randomly pick a parent so we decided to use that one insted.

3.3 Mutations

3.3.1 Random mutation

Randomly mutate a single move along the path.

```
// mutates random index of a move to a new random move
public void mutateRandomly() {
    int index = rand.nextInt(moves.length);
    moves[index] = rand.nextInt(4);

    calculateFitness();
}
```

3.3.2 Random mutation with path correction

Randomly mutate a single move along the path and then corrects all the moves afterwards so that they don't go into walls of the maze. For better algorithm performance you should also make back moves less likely, so that a bigger part of a maze is searched.

```
// mutates random index of a move to a new random move and corrects path afterwards
public void mutateAvoidWalls() {
    if(moves.length == 0) return;
    int index = rand.nextInt(moves.length);
    moves[index] = rand.nextInt(4);
    Vec2 curr = new Vec2(path[index].x, path[index].y);
    boolean moved = false;
    boolean skipBackMove = false;
    int backMoveSkipFirstTimeProb = 90;
    int backMoveSkipNotFirstTimeProb = 70;
    if(tressures == null || tressures.length == 0) {
        backMoveSkipFirstTimeProb = 99; // to avoid while true (probably not a great solution)
        backMoveSkipNotFirstTimeProb = 99;
    }
    for(int i = index; i < moves.length; i++) {
        moved = false;
        skipBackMove = false;
        int dir = moves[i];
        if(dir == 0 && maze[curr.x-1 + curr.y * width] != -1) {
            if(i > 0 && moves[i-1] == 1) {
                if(rand.nextInt(100) < backMoveSkipFirstTimeProb) skipBackMove = true;
            }
            if(!skipBackMove) {
                moves[i] = dir;
                curr.x--;
                moved = true;
            }
        }else if(dir == 1 && maze[curr.x+1 + curr.y * width] != -1) {
            if(i > 0 && moves[i-1] == 0) {
                if(rand.nextInt(100) < backMoveSkipFirstTimeProb) skipBackMove = true;
            }
            if(!skipBackMove) {
                moves[i] = dir;
                curr.x++;
                moved = true;
            }
        }else if(dir == 2 && maze[curr.x + (curr.y-1) * width] != -1) {
            if(i > 0 && moves[i-1] == 3) {
                if(rand.nextInt(100) < backMoveSkipFirstTimeProb) skipBackMove = true;
            }
            if(!skipBackMove) {
                moves[i] = dir;
                curr.y--;
                moved = true;
            }
        }else if(dir == 3 && maze[curr.x + (curr.y+1) * width] != -1) {
            if(i > 0 && moves[i-1] == 2) {
                if(rand.nextInt(100) < backMoveSkipFirstTimeProb) skipBackMove = true;
            }
```

```
}else if(dir == 3 && maze[curr.x + (curr.y+1) * width] != -1) {
    if(i > 0 && moves[i-1] == 2) {
        if(rand.nextInt(100) < backMoveSkipFirstTimeProb) skipBackMove = true;
    }
    if(!skipBackMove) {
        moves[i] = dir;
        curr.y++;
        moved = true;
    }
}

while(!moved) {
    skipBackMove = false;
    dir = rand.nextInt(4);
    if(dir == 0 && maze[curr.x-1 + curr.y * width] != -1) {
        if(i > 0 && moves[i-1] == 1) {
            if(rand.nextInt(100) < backMoveSkipNotFirstTimeProb) skipBackMove = true;
        }
        if(!skipBackMove) {
            moves[i] = dir;
            curr.x--;
            break;
        }
    }else if(dir == 1 && maze[curr.x+1 + curr.y * width] != -1) {
        if(i > 0 && moves[i-1] == 0) {
            if(rand.nextInt(100) < backMoveSkipNotFirstTimeProb) skipBackMove = true;
        }
        if(!skipBackMove) {
            moves[i] = dir;
            curr.x++;
            break;
        }
    }else if(dir == 2 && maze[curr.x + (curr.y-1) * width] != -1) {
        if(i > 0 && moves[i-1] == 3) {
            if(rand.nextInt(100) < backMoveSkipNotFirstTimeProb) skipBackMove = true;
        }
        if(!skipBackMove) {
            moves[i] = dir;
            curr.y--;
            break;
        }
    }else if(dir == 3 && maze[curr.x + (curr.y+1) * width] != -1) {
        if(i > 0 && moves[i-1] == 2) {
            if(rand.nextInt(100) < backMoveSkipNotFirstTimeProb) skipBackMove = true;
        }
        if(!skipBackMove) {
            moves[i] = dir;
            curr.y++;
            break;
```

```
            }else if(dir == 2 && maze[curr.x + (curr.y-1) * width] != -1) {
                if(i > 0 && moves[i-1] == 3) {
                    if(rand.nextInt(100) < backMoveSkipNotFirstTimeProb) skipBackMove = true;
                }
                if(!skipBackMove) {
                    moves[i] = dir;
                    curr.y--;
                    break;
                }
            }else if(dir == 3 && maze[curr.x + (curr.y+1) * width] != -1) {
                if(i > 0 && moves[i-1] == 2) {
                    if(rand.nextInt(100) < backMoveSkipNotFirstTimeProb) skipBackMove = true;
                }
                if(!skipBackMove) {
                    moves[i] = dir;
                    curr.y++;
                    break;
                }
            }
        }
        if(curr.x == end.x && curr.y == end.y) {
            int[] temp = moves;
            moves = new int[i+1];
            for(int j = 0; j <= i; j++) {
                moves[j] = temp[j];
            }
            break;
        }
    }
    calculateFitness();
}
```

3.3.3 Remove unneccesary moves

Go through the path and check if the coordinate was already visited and no thresures were picked between visits. If that happens this part of the path is useless so it should be removed:

```
public void mutateUnNecessaryMoves() {
    boolean somethingUseful = false;
    Vec2 skip = null;
    for(int i = 0; i < path.length-1; i++) {
        somethingUseful = false;
        for(int j = i+1; j < path.length; j++) {
            if(path[j].equals(end)) somethingUseful = true;
            if(tressures != null) {
                for(int t = 0; t < tressures.length; t++) {
                    if(path[j].equals(tressures[t])) somethingUseful = true;
                }
            }
            if(path[i].equals(path[j]) && !somethingUseful) {
                skip = new Vec2(i, j);
                break;
            }
        }
        if(skip != null) break;
    }
    if(skip != null) {
        int[] temp = moves;
        moves = new int[moves.length - (skip.y-skip.x)];
        for(int i = 0; i < moves.length; i++) {
            if(i < skip.x) {
                moves[i] = temp[i];
            }else {
                moves[i] = temp[i+skip.y-skip.x];
            }
        }
    }
}
```

3.4 Generating initial population

3.4.1 Initial population of random paths

Generates a random sequence of moves.

```
public void generateRandomPath() {
    for(int i = 0; i < moves.length; i++) {
        moves[i] = rand.nextInt(4);
    }
}
```

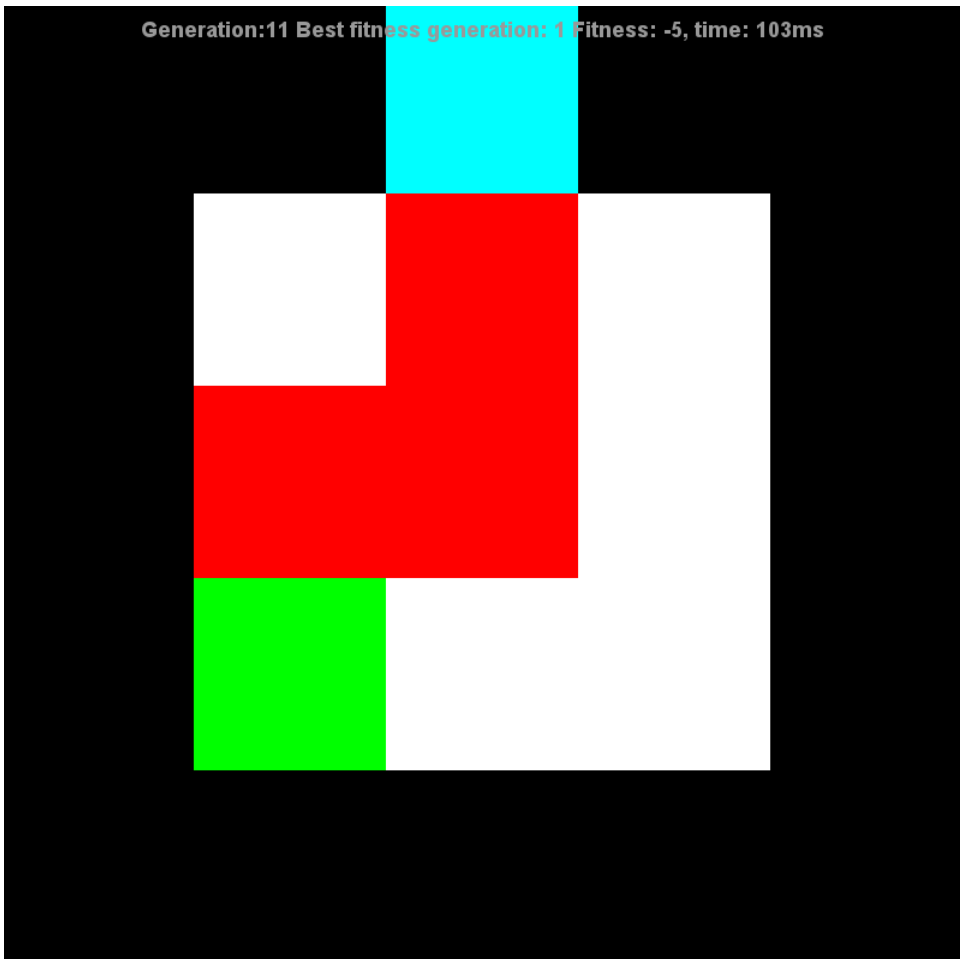
3.4.2 Initial population of random paths without going into walls

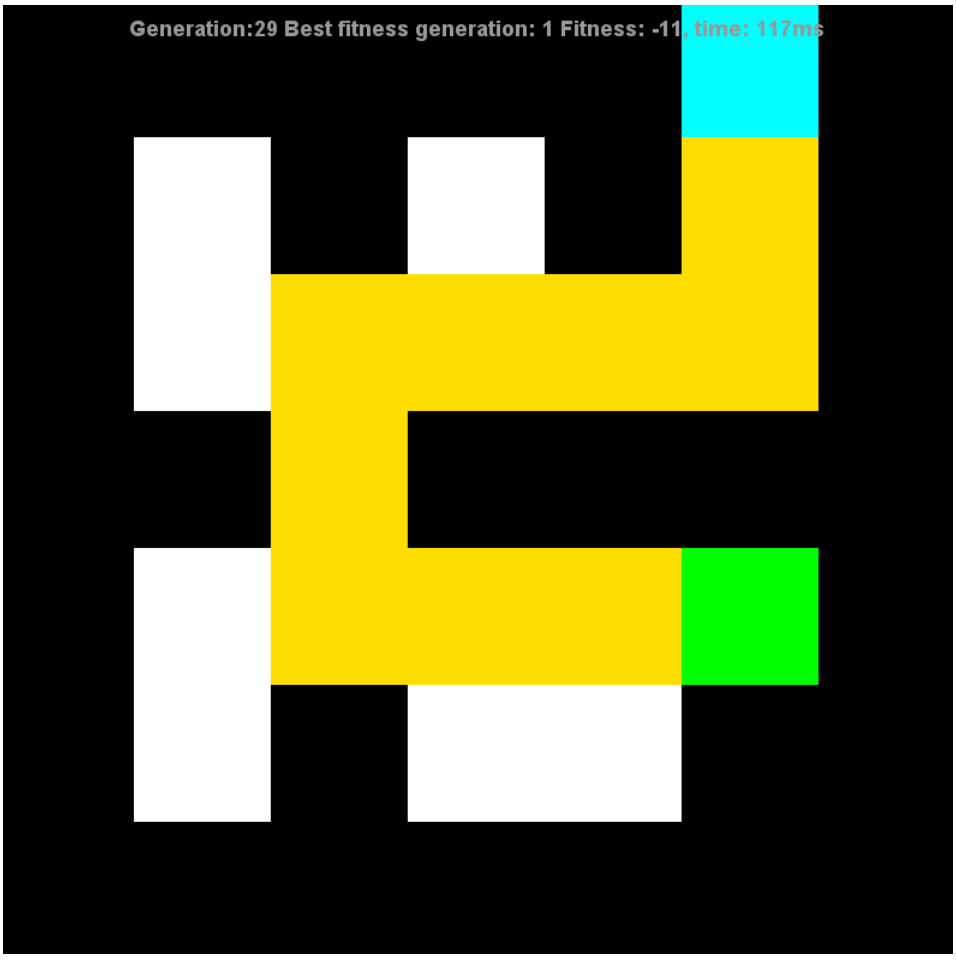
Generates a random sequence of moves that doesn't go through walls.

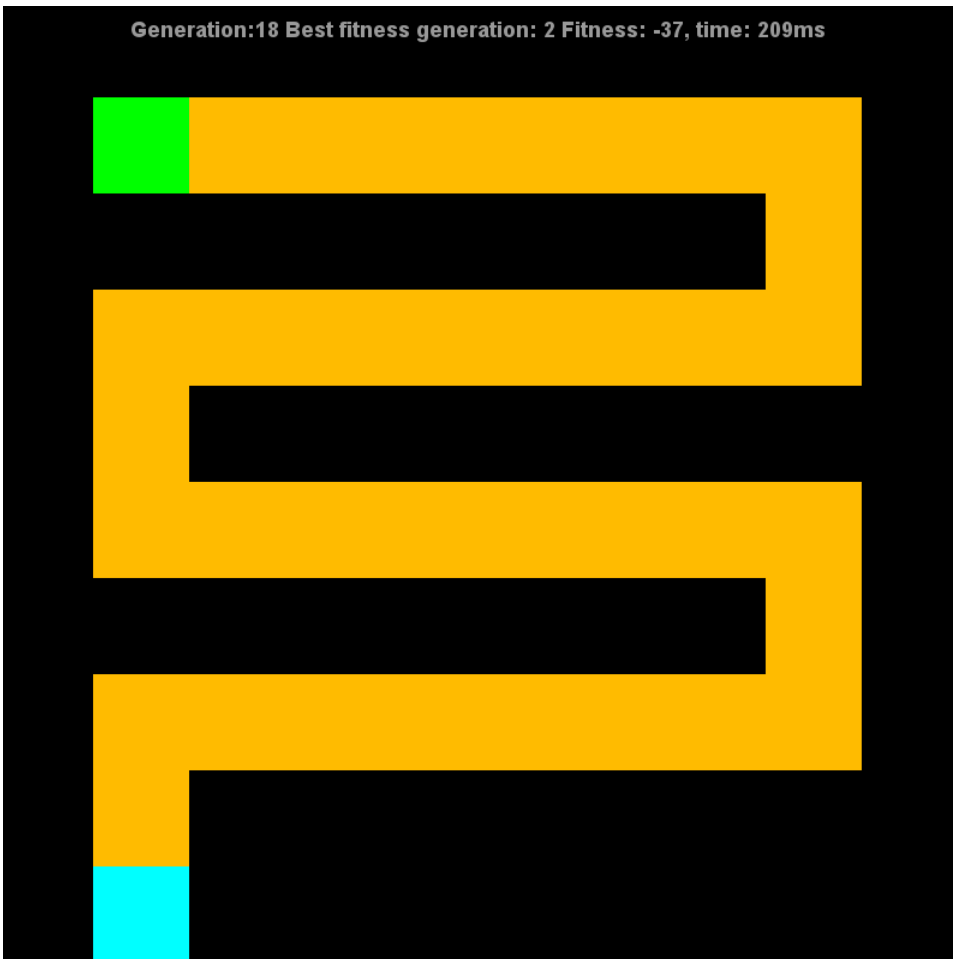
```
public void generateRandomPathAvoidWalls() {
    Vec2 curr = new Vec2(start.x, start.y);
    for(int i = 0; i < moves.length; i++) {
        while(true) {
            int dir = rand.nextInt(4);
            if(dir == 0 && maze[curr.x-1 + curr.y * width] != -1) {
                moves[i] = dir;
                curr.x--;
                break;
            }else if(dir == 1 && maze[curr.x+1 + curr.y * width] != -1) {
                moves[i] = dir;
                curr.x++;
                break;
            }else if(dir == 2 && maze[curr.x + (curr.y-1) * width] != -1) {
                moves[i] = dir;
                curr.y--;
                break;
            }else if(dir == 3 && maze[curr.x + (curr.y+1) * width] != -1) {
                moves[i] = dir;
                curr.y++;
                break;
            }
        }
        if(curr.x == end.x && curr.y == end.y) {
            int[] temp = moves;
            moves = new int[i+1];
            for(int j = 0; j <= i; j++) {
                moves[j] = temp[j];
            }
            break;
        }
    }
}
```

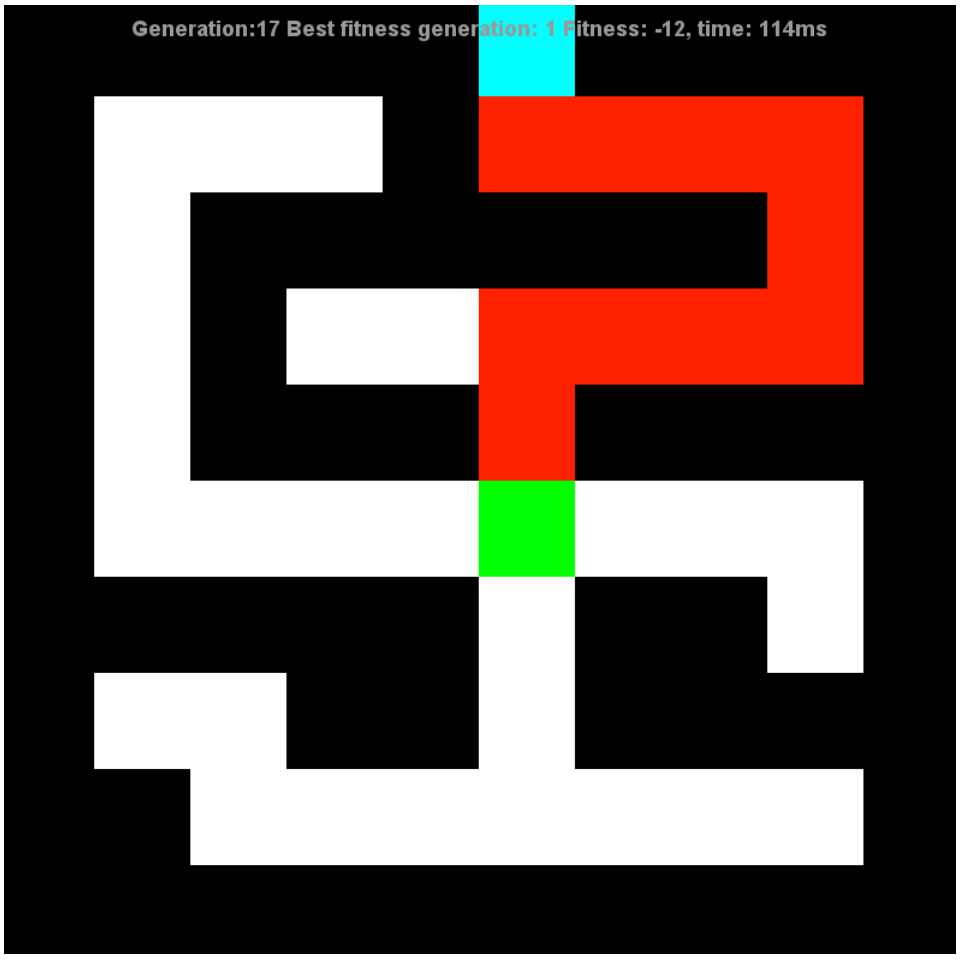
4 Task 4 - Results

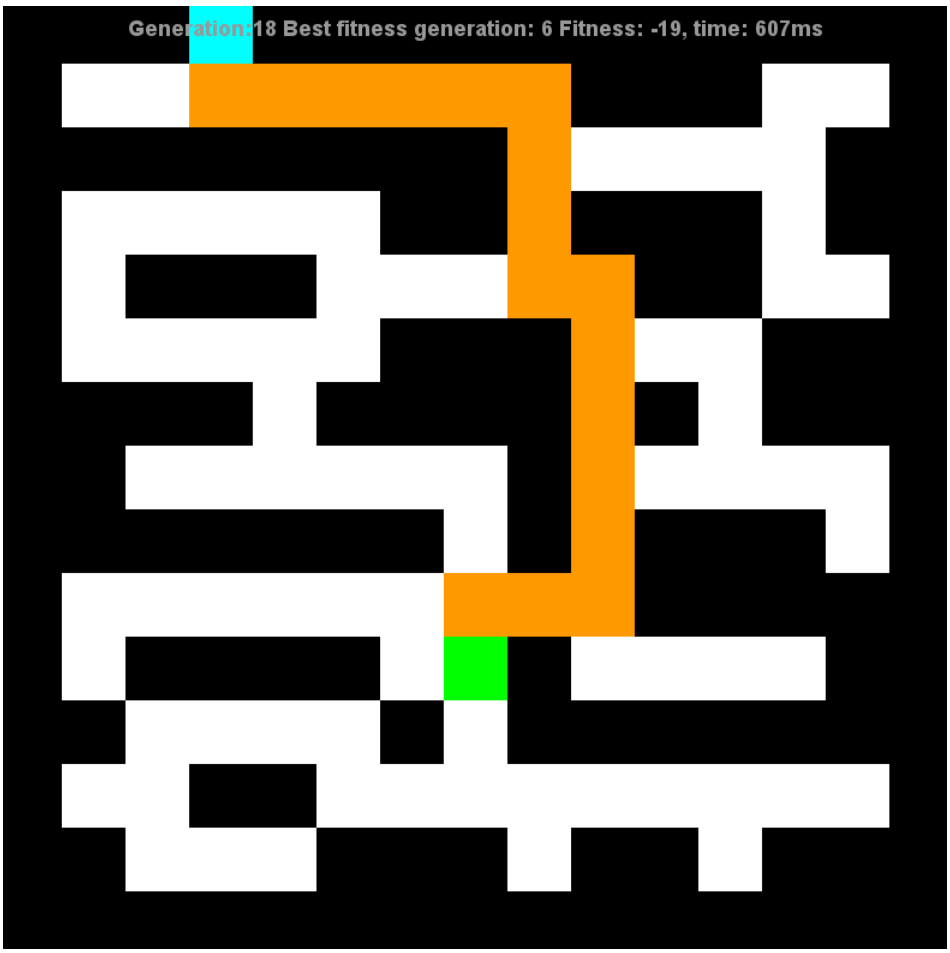
Solutions of mazes:

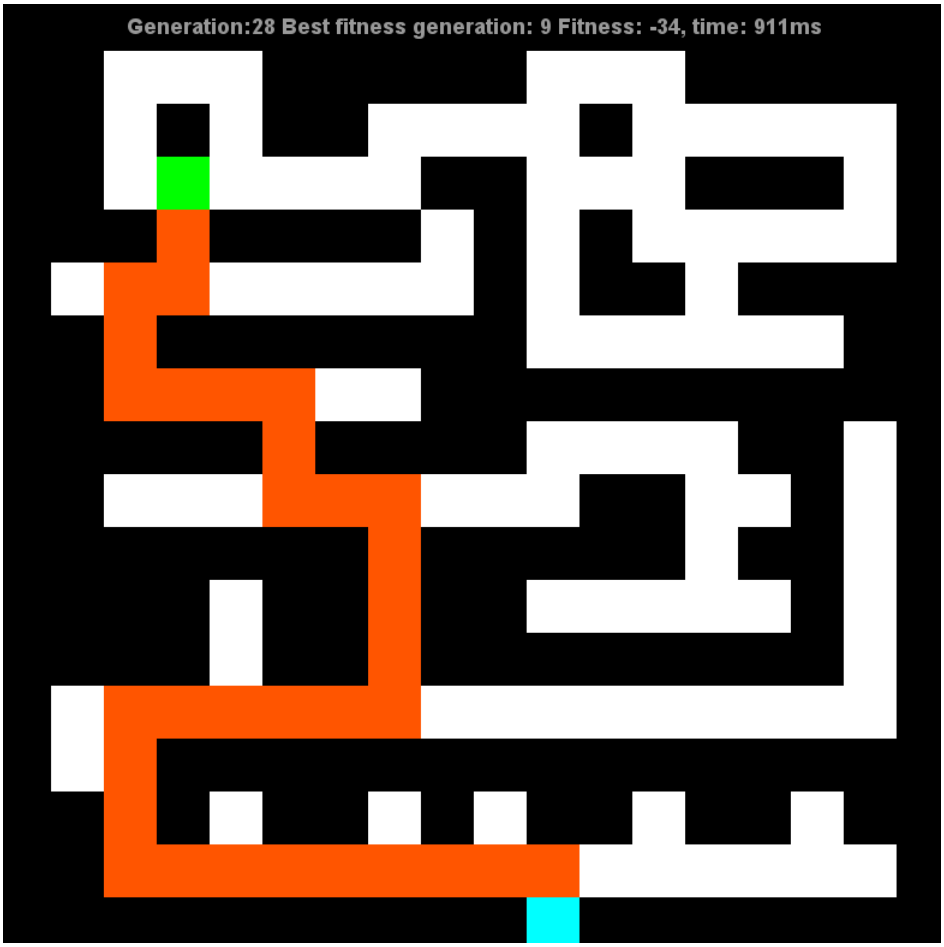

Generation:11 Best fitness generation: 1 Fitness: -5, time: 103ms

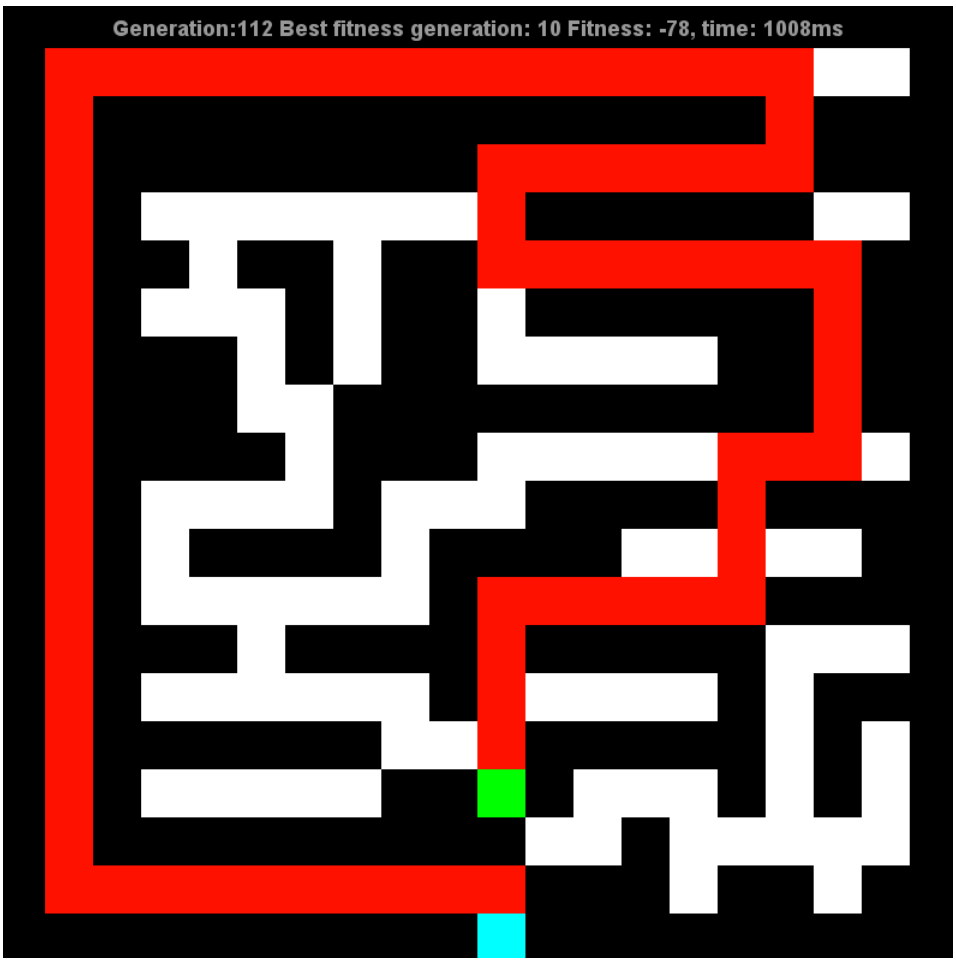

Generation:29 Best fitness generation: 1 Fitness: -11, time: 117ms

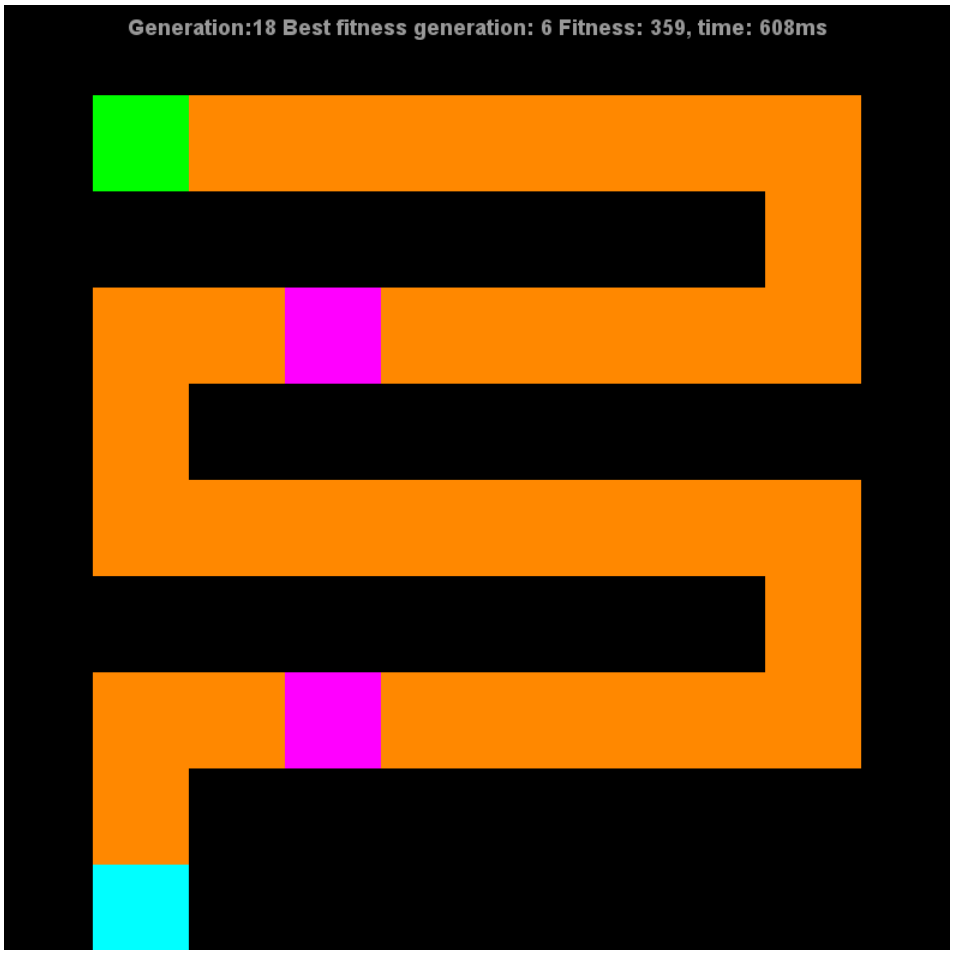

Generation:18 Best fitness generation: 2 Fitness: -37, time: 209ms

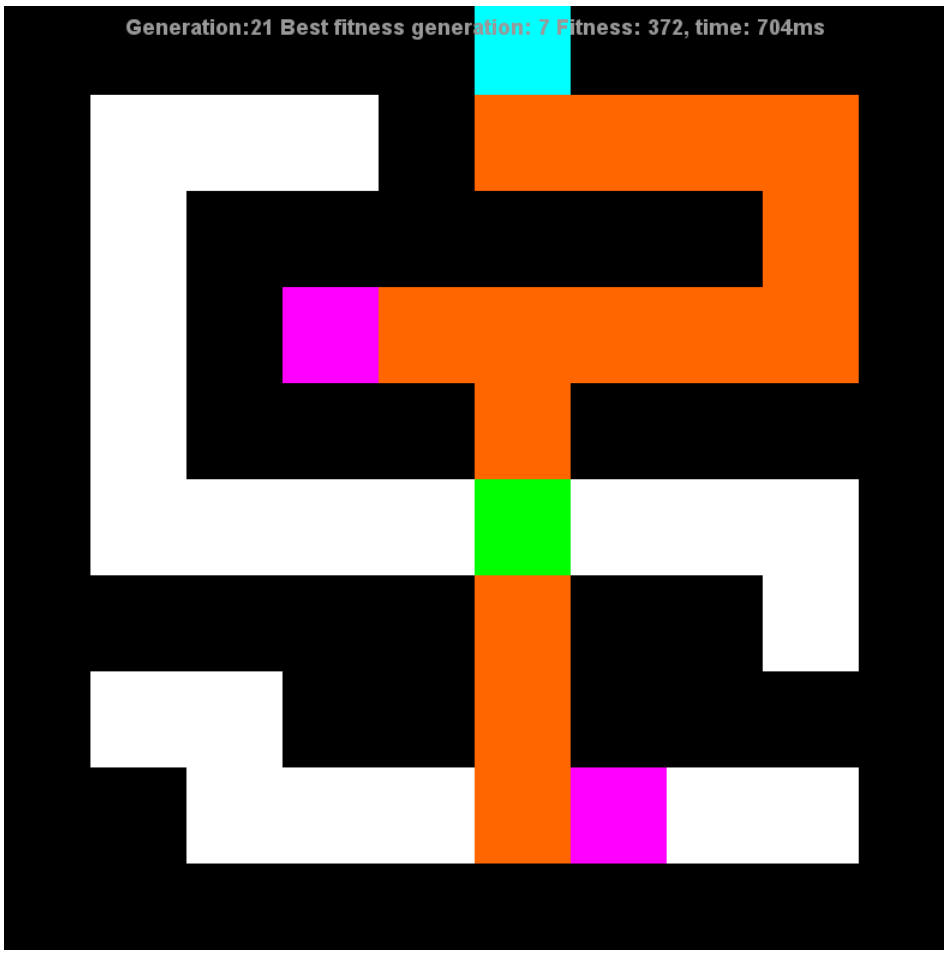

Generation:17 Best fitness generation: 1 Fitness: -12, time: 114ms

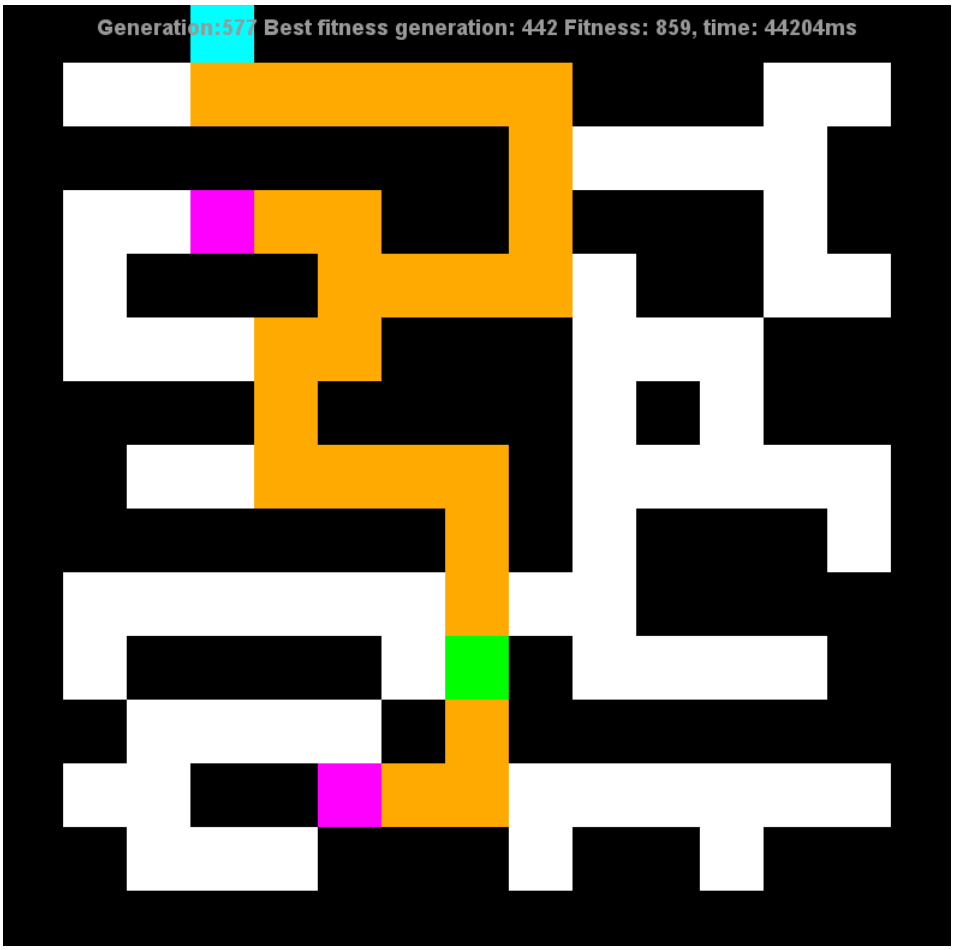

Generation:18 Best fitness generation: 6 Fitness: -19, time: 607ms

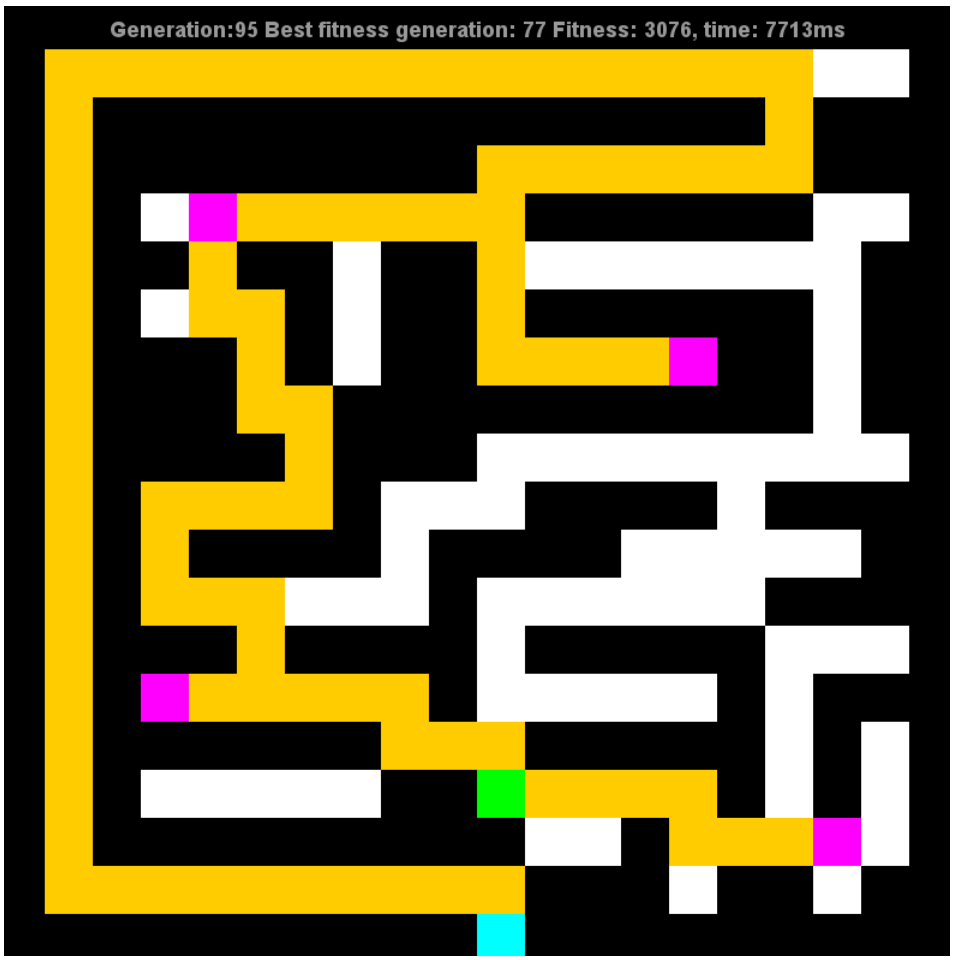

Generation:28 Best fitness generation: 9 Fitness: -34, time: 911ms

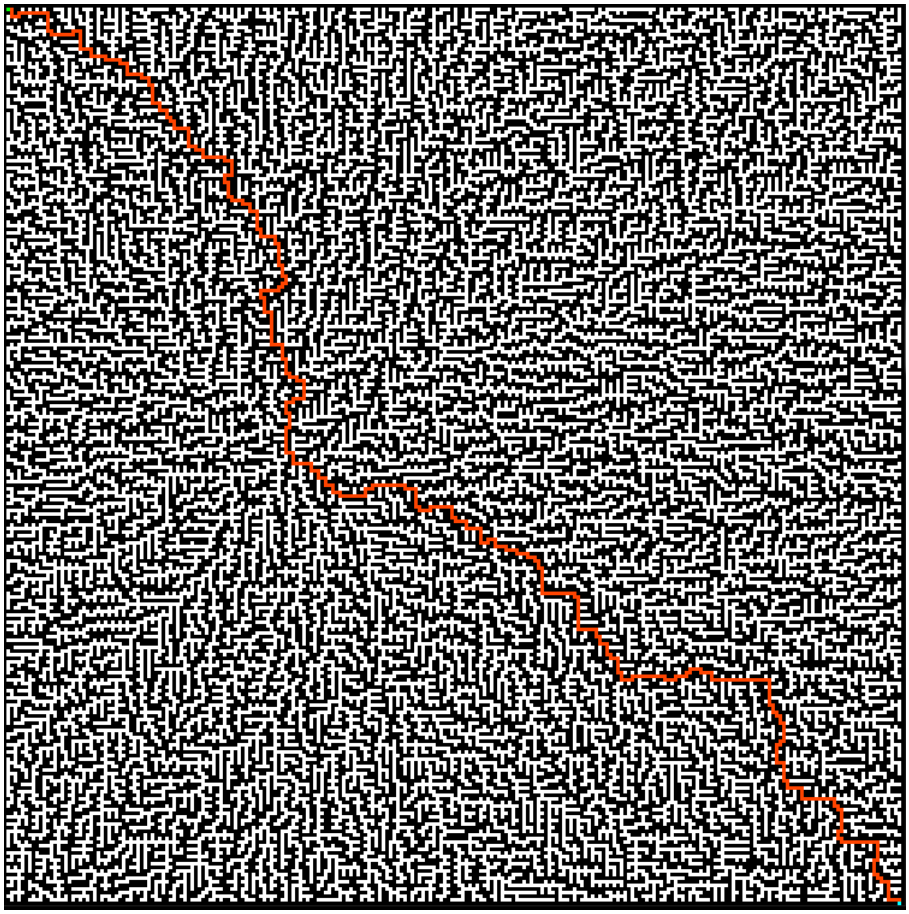

Generation:112 Best fitness generation: 10 Fitness: -78, time: 1008ms


Generation:18 Best fitness generation: 6 Fitness: 359, time: 608ms


Generation:21 Best fitness generation: 7 Fitness: 372, time: 704ms


Generation:577 Best fitness generation: 442 Fitness: 859, time: 44204ms


Generation:95 Best fitness generation: 77 Fitness: 3076, time: 7713ms



4.1 Performance comparison

Selections: linearly biased selection works a lot better than a random selection.

Crossover: choosing a random parent works better than computing the intersections, because it is not computationaly expensive and mutations can make this behaviour very well and fast.

Mutations: Combination of changing a random move along the path and then correcting next moves so that they don't go through walls woked best for us with a mutation that removes unneccesary parts of the path (this mutation is only possible on chromosomes that make it to the end).

Initial population generation: Initial population generation should be generated randomly and should take walls into account, because in that way a big part of a maze is searched in every generation (if mutations and crossovers have the same behaviour). Even better performance would be if back moves are less likely.

Population size: We got the best results with initial population of size 1000. The number depends on the maze but bigger number means more of a maze will be searched in first generation and thus the algorithm is more likely to converge faster. But too big number means every generation will be computed slower so the "good" mutations will take more time to appear.

Elitism: 10 percent seems to be a good number. To small percentage might mean a very good candidate will get mutated and lost. Too big percentage will make algorithm slower and is not neccessary so it means we are wasting memory.

Mutation probability: 30 percent gave us a good results. We don't want to mutate with too big percentage because than all the data gained from genrations will be lost. But too small of a percentage will make local maximums harder to avoid.

Back move probability: If a maze has no tresasures back moves should be avoided so the probability of back moves should be 0 (in our implementation of the algorithm it is 1 percent so that we don't have to change the chromosome length when a dead end is found). If a maze contains tressures, we got very good results with 10 percent probability to allow back move on a first try and 30 percent on every subsequent try.

Note: The results will vary and are hard to evaluate based on one try, because algorithm uses randomness.