Slovenská Technická Univerzita

Fakulta informatiky a informačných technológii

Zens Garden

Zadanie 2.

Umelá Inteligencia

Martin Čajka

AIS ID: 116158

Školský rok 2022/2023

Martin Čajka
AIS ID: 116158

Description of the assignment:

The task of the assignment is to implement a genetic algorithm able to solve the problem known as Zens Garden. The problem consists of an x*y board, immovable stones at certain positions and a monk, whose goal is to completely rake the garden.

The monk's movements have certain restrictions – his movement starts from the edge of the garden, and he can only move horizontally or vertically, depending on the edge of the garden he started from. If a stone or an already raked field appears in his path, he has to change directions and continue moving in that direction until he reaches another stone, raked field, or returns to the edge of the garden. The monk can move freely on the edges, for example he can start on the left edge of the garden, get to the right edge while raking the garden and then move around the edge to the bottom and start raking from there.

Figure 1 - an (unfinished) example garden *source: http://www2.fiit.stuba.sk/~kapustik/zen.html*

Genetic Algorithm description

A genetic algorithm is a way to solve problems by simulating natural selection. More fit individuals pass their genes or their copies into the next generation, providing a better starting point for the next generation. Genes are passed by creating an offspring from parents and genes are able to mutate, which provides the ability to search in a wider scope.

Genes:

In the context of the given problem, genes are represented by the sequence of starting edge positions and the horizontal and vertical turns the monk takes when his direction is blocked by a stone or a raked field.

The sequence of starting edge positions is a list consisting of edge positions, which have the form of a pair of coordinates - (x, y). The number of selected edge cases is calculated with the following formula:

numOfRows+numOfColumns+numOfStones-2.

This formula has been selected as the maximum limit of genes has to be less than the half of the perimeter plus the number of stone.

The -2 appears in the formula to leave space for the other two genes – the horizontal turn and the vertical turn. The horizontal turn takes values 'u' and 'd' (as in **u**p and **d**own), while the vertical turn takes values 'l' and 'r' (as in **l**eft and **r**ight).

For simplicity, the genes are listed as certain attributes of the class GARDENER. As we can see, their values are initialized randomly. This is the case for the initial population and individuals that do not come from a selection or crossover method.

```python
class GARDENER:
    def __init__(self, rows, columns, i) -> None:
        self.identification = i

        self.starting_position_x = 0  # these positions showcase the current position of the gardener throughout the mov
        self.starting_position_y = 0

        self.turn_vertically = random.choice(['r', 'l'])
        self.turn_horizontally = random.choice(['u', 'd'])
        self.garden = init_garden(rows, columns)
        self.position_path = getRandomPath(rows, columns)
        self.fitness = 0
        self.target_fitness = rows * columns - len(STONE_LOCATIONS)
        self.finished_gardening = False
        self.solved = False
```

Figure 2 - attributes self.position_path, self.turn_vertically, self.turn_horizontally represent the genes.

Selection Methods:

In this solution, two selection methods were implemented – Elitism and Roulette selection. The user is able to select the method of selection via input – 1 for Elitism and 2 for Roulette's.

```
1.  Elitism Selection
2.  Roulette Selection
Enter the number corresponding to the desired selection:
```

Elitism selection: This selection method is quite simplistic. The 15% of individuals with the highest fitness are selected and copied into the next generation.

```
def elitismSelection(sortedArray):
    myList = []
    for i in range(int(NUM_OF_INDIVIDUALS / 100 * 15)):
        myList.append(sortedArray[i])

    return myList
```

Roulette selection: For this selection method all individuals of the generation have a chance to be moved into the next generation, but individuals with higher fitness have a higher probability of being chosen. Since the variability has been too high, the given program only takes 50% of the population with the highest fitness and does a roulette selection from them. With the whole population, generations have been often declining due to the reason of weaker individuals carrying their genes over, but sometimes the increased variability produced strong individuals as modified genes of the weaker individuals proved to be efficient.

This selection has been implemented in the following way. First, a list is created to sort to only contain the 50% of population with higher fitness. Afterwards, a sum of the fitness of the population is calculated, which serves as the upper limit for choosing a random number in the interval from 0 to the sum. The randomly selected number serves as a point for choosing an individual. This number is iteratively decreased by subtracting individuals' fitness, starting with the individual with the lowest fitness. When the number is subtracted below zero, the currently iterated individual is chosen for reproduction and its copy is passed into the next generation. Similarly, as in elitism selection, we select 15% of the population to be passed into the next generation and to reproduce.

```
def rouletteSelection(sortedArray):
    myList = []
    sum = 0

    betterHalf = []

    for i in range(int(NUM_OF_INDIVIDUALS/2)):
        betterHalf.append(sortedArray[i])

    for i in range(len(betterHalf)):
        sum += sortedArray[i].fitness

    for i in range(int(NUM_OF_INDIVIDUALS / 100 * 15)):

        num = random.randint(0, sum)

        for gardener in reversed(sortedArray):

            num -= gardener.fitness
            if num < 0:
                myList.append(gardener)
                break
```

Crossover (Reproduction):

The crossover method in the given program accounts for the remaining 85% of the next generation since the remaining individuals are copies of the resulting 15% of chosen selection. For each crossover, two random parents are selected from the returned individuals from the selection method. First, a random index is selected from 0 to the length of the parents' path. This index serves as the crossover point, where starting positions before the index are of the first parent, and the positions after the index are copied from the second parent. Similarly, the preferred horizontal and vertical turn is randomly selected from the parents horizontal and vertical turn gene.

```
def crossover(selected_individuals, temp):
    parent1 = random.choice(selected_individuals)
    parent2 = random.choice(selected_individuals)

    point_of_crossover = random.randint(0, len(parent1.position_path))

    newPath = []

    for i in range(len(parent1.position_path)):
        if i > point_of_crossover:
            newPath.append(parent2.position_path[i])
        else:
            newPath.append(parent1.position_path[i])

    temp.position_path = newPath
    temp.turn_vertically = random.choice([parent1.turn_vertically, parent2.turn_vertically])
    temp.turn_horizontally = random.choice([parent1.turn_horizontally, parent2.turn_horizontally])

    return temp

```

Mutation:

Genes have a chance to mutate, meaning genes have the ability to change to increase the scope of evolution. The probability of mutation can be chosen by the user. The mutation for the genes varies. The gene of the individuals starting position path mutates by randomly choosing two indexes of the path and switching them. Horizontal and vertical turn gene mutates by changing the direction of the turn, if the vertical turn was left ('l'), the mutation causes the turn to switch to right ('r').

```python
def mutation(gardener):
    pathNum = random.randint(1, 10)
    horiNum = random.randint(1, 10)
    vertNum = random.randint(1, 10)
    p = MUTATION_PROBABILITY * 10

    if p >= pathNum:
        num1 = random.randint(0, len(gardener.position_path) - 1)
        num2 = random.randint(0, len(gardener.position_path) - 1)
        gardener.position_path[num1], gardener.position_path[num2] = gardener.position_path[num2], gardener.position_path[num1]

    if p >= vertNum:
        if gardener.turn_vertically == 'r':
            gardener.turn_vertically = 'l'
        elif gardener.turn_vertically == 'l':
            gardener.turn_vertically = 'r'

    if p >= horiNum:
        if gardener.turn_horizontally == 'u':
            gardener.turn_horizontally = 'd'
        elif gardener.turn_horizontally == 'd':
            gardener.turn_horizontally = 'u'
```

Parameters settings:

Parameters of the genetic algorithm are modifiable, as they are saved in global variables. The number of individuals, the number of generations, mutation probability and selection method percentage are the main attributes of the genetic algorithm, and the pre-set values have been chosen as 100 for the population as well as the number of generations and the probability of mutation is at .1 or 10%. The chosen selection method percentage is 15.

```python
NUM_OF_INDIVIDUALS = 100  # number of individuals in a generation
NUM_OF_GENERATIONS = 100  # number of generations
MUTATION_PROBABILITY = .1  # sets the probability of mutation
SELECTION_METHOD_PERCENTAGE = 15  # sets the percentage of population copied to the next generation and do the crossove
```

Other parameters that are selectable is the size of the board, number of stones and the selection method, all of which are initialized by the user at the start of the program.

```python
def main():
    rows = int(input("Enter number of rows: "))
    columns = int(input("Enter number of columns: "))

    add_stones(rows, columns)

    print("\n1.\tElitism Selection")
    print("2.\tRoulette Selection")

    chooseSelectionMethod = int(input("Enter the number corresponding to the desired selection: "))  # user chooses selection m
```

The add_stones function adds stones to the garden indefinitely, until the user break the loop.

Monk movements:

The monk's movement is defined mostly by his genes. The monk continuously starts raking from positions that are in his starting position gene. If his movement is blocked by a stone, depending on if he was moving horizontally or vertically, his turn is decided by the given gene. If he is not able to turn in that direction, the opposite direction is tried and if he is blocked in that direction as well, the monk is stuck and his movement ends. The movement can also end if the monk has run out of starting positions/ has completed his path, or he was able to cover the whole garden.

```python
def solve(gardener, columns, rows):  # solves the board for the given gardener depending on his genes (path of start:

    counter = 1
    for position in gardener.position_path:

        gardener.starting_position_x = position[0]
        gardener.starting_position_y = position[1]

        if gardener.starting_position_x == 0:
            if gardener.garden[position[0] + 1][position[1]] != '0':
                continue
            moveDown(gardener, counter)
        elif gardener.starting_position_x == rows + 1:
            if gardener.garden[position[0] - 1][position[1]] != '0':
                continue
            moveUp(gardener, counter)
        elif gardener.starting_position_y == 0:
            if gardener.garden[position[0]][position[1] + 1] != '0':
                continue
            moveRight(gardener, counter)
        elif gardener.starting_position_y == columns + 1:
            if gardener.garden[position[0]][position[1] - 1] != '0':
                continue
            moveLeft(gardener, counter)

        if gardener.finished_gardening:
            return
```

The solve function iterates through the monk's path. Depending on which axis (side of garden) he is on, a movement function is called in that direction if the next field was not yet raked or holds a stone.

```python
def moveLeft(gardener, operatorNum):
    new_position_x = gardener.starting_position_x
    new_position_y = gardener.starting_position_y - 1

    if gardener.garden[new_position_x][new_position_y] == 'x':
        return

    if gardener.garden[new_position_x][new_position_y] != '0':
        turnUpDown(gardener, operatorNum)
        return

    if gardener.garden[new_position_x][new_position_y] == '0':
        gardener.garden[new_position_x][new_position_y] = str(operatorNum)
        gardener.starting_position_x = new_position_x
        gardener.starting_position_y = new_position_y
        gardener.fitness += 1
        if gardener.fitness == gardener.target_fitness:
            gardener.finished_gardening = True
            gardener.solved = True
        moveLeft(gardener, operatorNum)
```

Figure 3 - Left direction movement function.

The movement in the given direction continues until the monk reaches the edge of the garden or his direction movement is blocked. In the latter case, the turn function is called. If the monk is able to continue to the next field, the field is rewritten with the number of the move and the fitness attribute of the monk is incremented. The function afterwards calls itself recursively.

```python
def turnRightLeft(gardener, operatorNum):
    if gardener.turn_vertically == 'r':
        if gardener.garden[gardener.starting_position_x][gardener.starting_position_y + 1] == 'x' or \
                gardener.garden[gardener.starting_position_x][gardener.starting_position_y + 1] == '0':
            moveRight(gardener, operatorNum)
        else:
            if gardener.garden[gardener.starting_position_x][gardener.starting_position_y - 1] == 'x' or \
                    gardener.garden[gardener.starting_position_x][gardener.starting_position_y - 1] == '0':
                moveLeft(gardener, operatorNum)
            else:
                gardener.finished_gardening = True
                return

    elif gardener.turn_vertically == 'l':
        if gardener.garden[gardener.starting_position_x][gardener.starting_position_y - 1] == 'x' or \
                gardener.garden[gardener.starting_position_x][gardener.starting_position_y - 1] == '0':
            moveLeft(gardener, operatorNum)
        else:
            if gardener.garden[gardener.starting_position_x][gardener.starting_position_y + 1] == 'x' or \
                    gardener.garden[gardener.starting_position_x][gardener.starting_position_y + 1] == '0':
                moveRight(gardener, operatorNum)
            else:
                gardener.finished_gardening = True
                return
```

Figure 4 - Right-Left turn function

The turn function checks the gene of the monk, and the monk starts moving in the given direction if he is able to. Otherwise, he switches directions and tries to continue in that direction. If he is blocked, the function returns.

Implementation Environment:

As for finding and implementing a solution for this problem, the chosen programming language was Python version 3.10, and the selected IDE was PyCharm 2022.2.2. The choice was made due to an easy access and large variety of libraries. Libraries used were as follows:

random – Used to randomize initial population values.

time – Used to measure time complexity.

pandas – Used to display the garden in the console

matplotlib.pyplot – Used to graph generations line chart.

Testing and Summary:

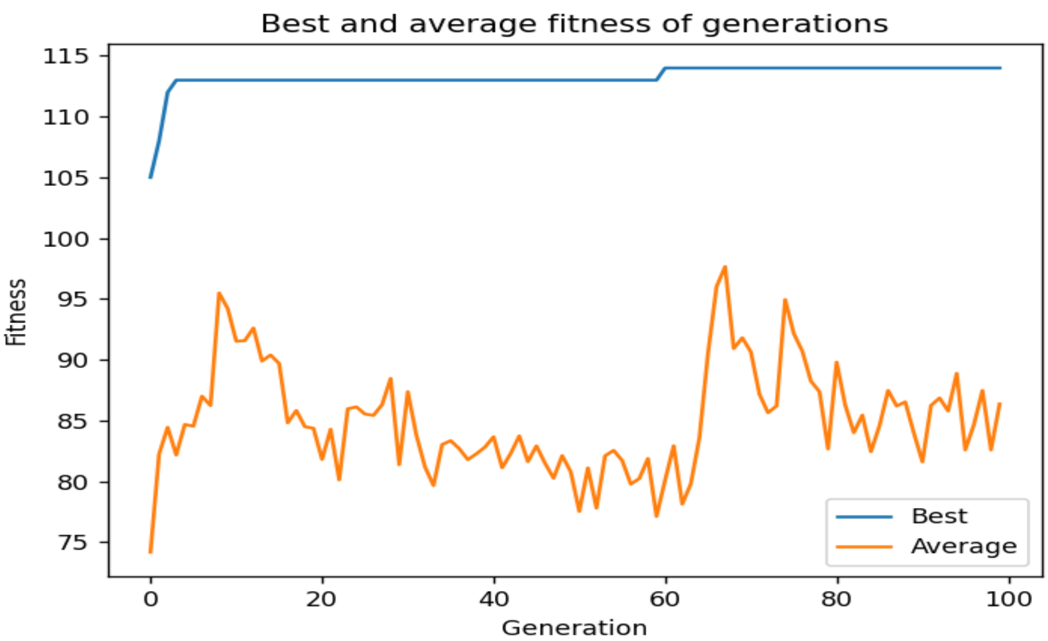
The first testing environment for the board was the example board given in in assignment. All the tests include 100 generations and 100 individuals.

Board size: 10*12.

Stone locations: [(3,2),(5,3),(4,5),(2,6),(7,9),(7,10)]

ELITISM

Test scenario parameters: Mutation 10%, Elitism 15%, Crossover from elitism 85%



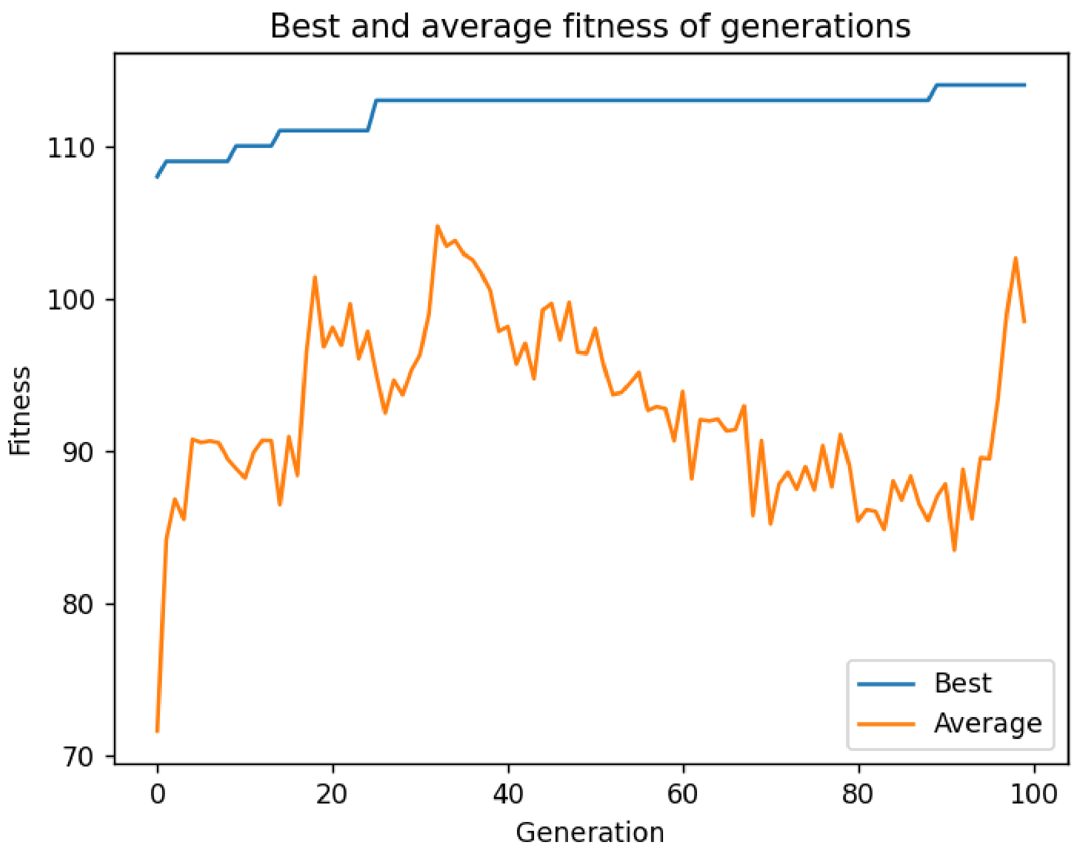

```
Best individual garden:
   0  1  2  3  4  5  6  7  8  9  10 11 12 13
0   x  x  x  x  x  x  x  x  x  x  x  x  x  x
1   x  8  8  5  6  6  5  5  4  7  7  2  9  x
2   x  8  8  5  6  6  s  5  4  7  7  2  9  x
3   x  4  s  5  5  5  5  5  4  7  7  2  9  x
4   x  4  5  5  5  s  5  5  4  7  7  2  9  x
5   x  4  5  s  5  5  5  5  4  7  7  2  9  x
6   x  4  5  5  5  5  5  5  4  7  7  2  9  x
7   x  4  4  4  4  4  4  4  4  s  s  2  9  x
8   x  3  3  3  3  3  3  3  3  3  3  2  9  x
9   x  3  3  3  3  3  3  3  3  3  3  2  2  x
10  x  1  1  1  1  1  1  1  1  1  1  1  1  x
11  x  x  x  x  x  x  x  x  x  x  x  x  x  x
Individuals' starting position path
Best individual generation: 60
Best individual fitness: 114
Vertical turn preference: l
Horizontal turn preference: d
Time to complete algorithm: 1.86 seconds.
```

Figure 5 - Individuals starting path was not printed due to being too long

Martin Čajka
AIS ID: 116158

Cvičenie: Streda 15:00

Test scenario parameters: Mutation 10%, Elitism 20%, Crossover from elitism 80%

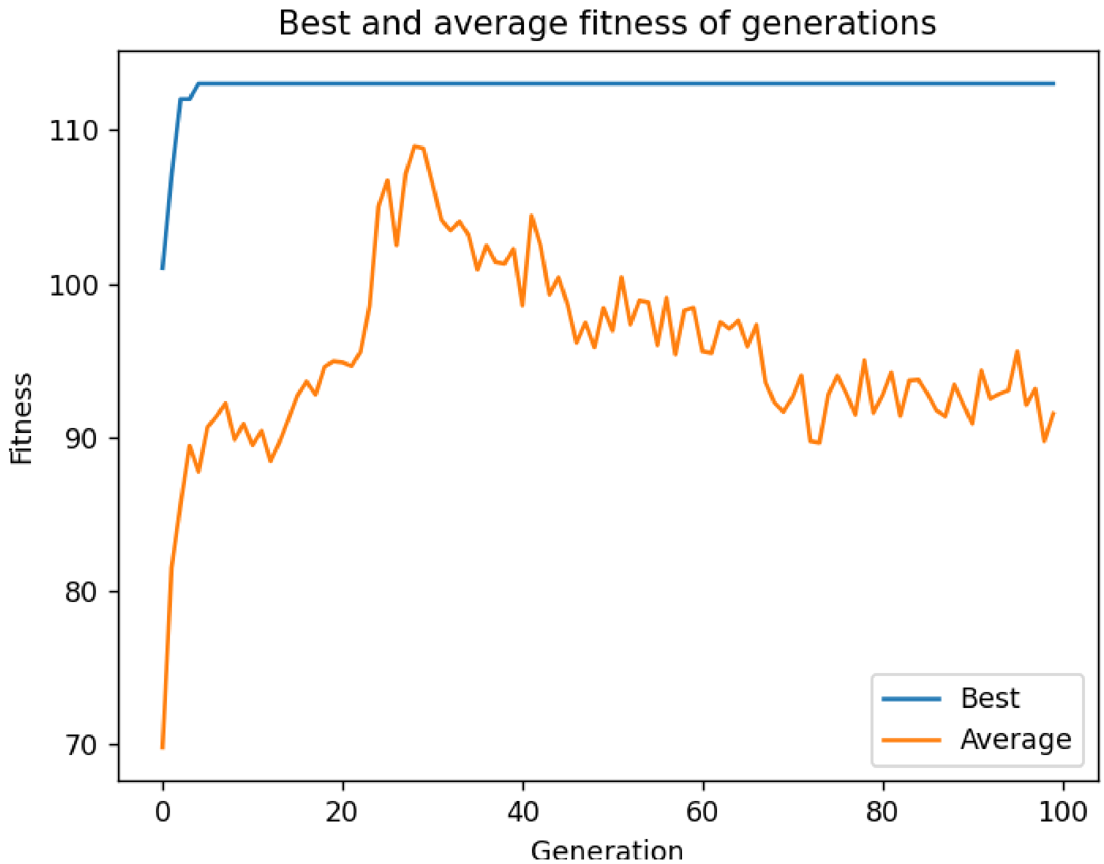



```
Best individual garden:
   0   1   2   3   4   5  6  7  8   9   10 11 12 13
0   x   x   x   x   x   x  x  x  x   x   x  x  x  x
1   x   9   9   9   9   9  9  1  3  11  11  4  5  x
2   x   2   2   2   2   2  s  1  3  11  11  4  5  x
3   x  12   s  12  12   2  2  1  3  11  11  4  5  x
4   x  12  12  12  12   s  2  1  3  11  11  4  5  x
5   x  12  12   s  12  12  2  1  3  11  11  4  5  x
6   x  12  12  12  12  12  2  1  3  11  11  4  5  x
7   x  12  12  12  12  12  2  1  3   s   s  4  5  x
8   x   7   7   7   7   7  2  1  3   6   6  4  5  x
9   x   8   8   8   8   7  2  1  3   6   6  4  5  x
10  x  10  10  10   8   7  2  1  3   6   6  4  5  x
11  x   x   x   x   x   x  x  x  x   x   x  x  x  x
Individuals' starting position path:
[(0, 7), (2, 0), (11, 8), (0, 11), (11, 12), (11, 10), (8, 0), (2, 13), (9, 0), (0, 1),
Best individual generation: 89
Best individual fitness: 114
Vertical turn preference: r
Horizontal turn preference: d
Time to complete algorithm: 1.75 seconds.
```

Martin Čajka
AIS ID: 116158

Cvičenie: Streda 15:00

Test scenario parameters: Mutation 10%, Elitism 40%, Crossover from elitism 60%

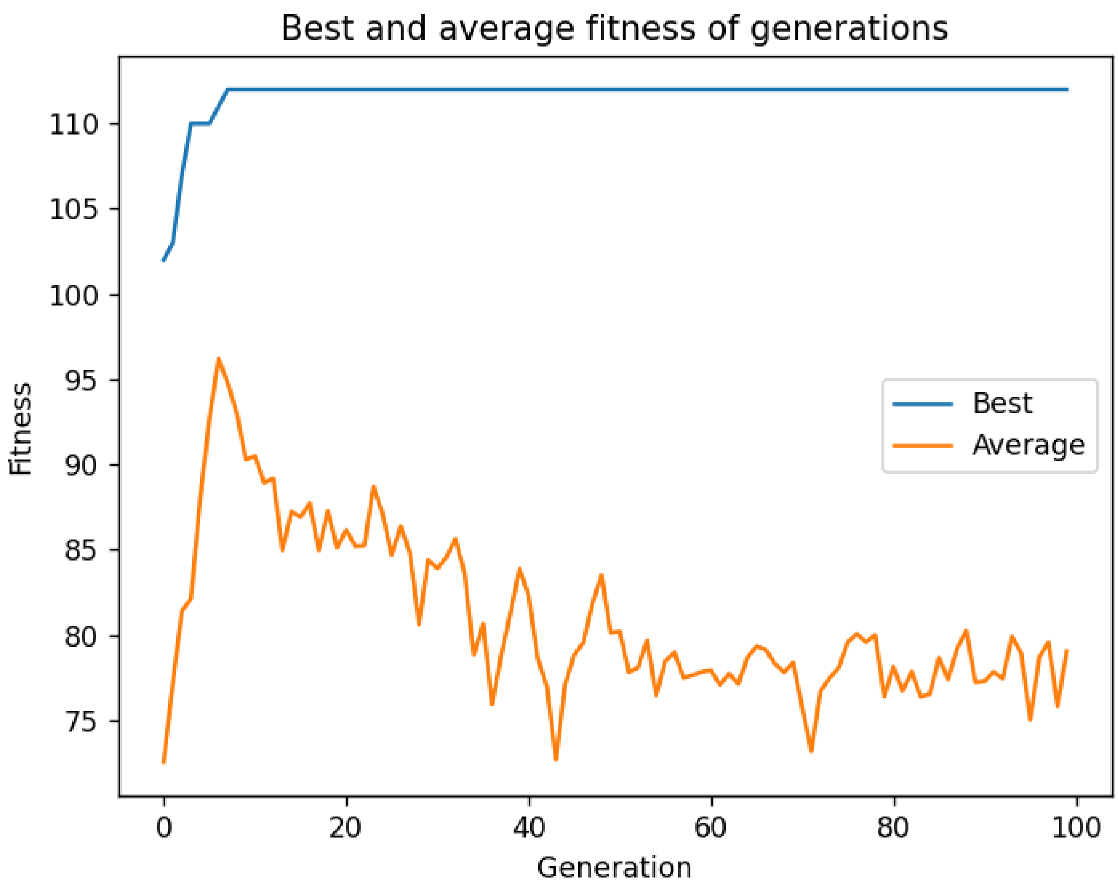



```
Best individual garden:
    0  1  2  3   4   5   6  7  8   9   10 11 12 13
0   x  x  x  x   x   x   x  x  x   x   x  x  x  x
1   x  2  7  0  10  10   8  8  8   8   6  6  6  x
2   x  2  7  7   7   7   s  5  5   5   5  5  5  x
3   x  2  s  7   7   7   7  5  1   1   1  1  1  x
4   x  2  7  7   7   s   7  5  1   1   1  1  1  x
5   x  2  7  s   7   7   7  5  5   5   5  5  5  x
6   x  2  7  1   1   1   1  1  1   1   1  1  1  x
7   x  2  7  1  12  12  12  4  4   s   s  3  3  x
8   x  2  7  1  12  13  12  4  4  11  11  3  9  x
9   x  2  7  1  12  13  12  4  4  11  11  3  9  x
10  x  2  7  1  12  13  12  4  4  11  11  3  9  x
11  x  x  x  x   x   x   x  x  x   x   x  x  x  x
Individuals' starting position path:
[(3, 13), (0, 1), (2, 13), (11, 7), (0, 7), (0, 10), (11, 5), (2, 0), (8, 13), (11, 11),
Best individual generation: 4
Best individual fitness: 113
Vertical turn preference: r
Horizontal turn preference: d
Time to complete algorithm: 1.13 seconds.
```

Martin Čajka
AIS ID: 116158

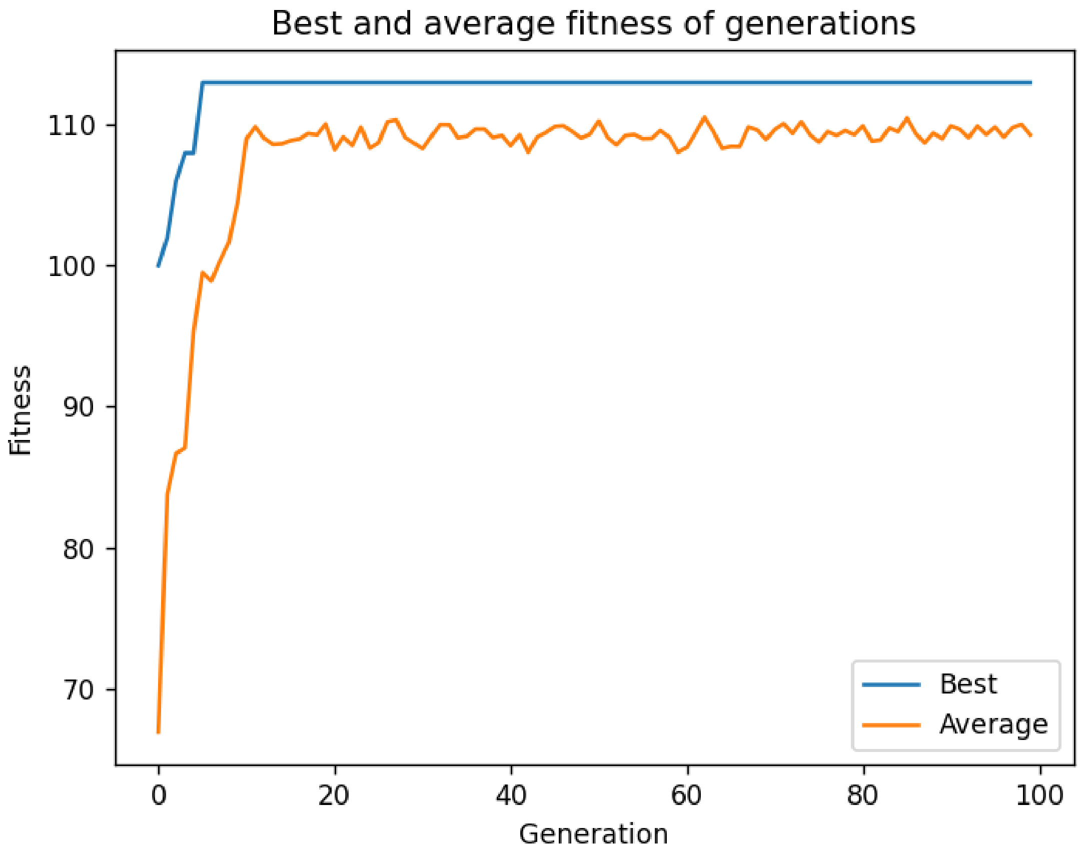
Cvičenie: Streda 15:00

Test scenario parameters: Mutation 40%, Elitism 15%, Crossover from elitism 85%




```
Best individual garden:
   0   1   2   3  4  5   6   7   8   9   10  11  12 13
0   x   x   x   x  x  x   x   x   x   x   x   x   x  x
1   x   1  14  17  4  4   0   3   1   7   9   9  15  x
2   x  14  14  17  4  4   s   3   1   7   9   9  15  x
3   x   0   s  17  4  4   4   3   1   7   9   9  15  x
4   x  12  12  17  4  s   4   3   1   7   7   7   7  x
5   x  12  12   s  4  4   4   3   1  13  13  13  13  x
6   x   3   3   3  3  3   3   3   1  13  13  13  13  x
7   x   1   1   1  1  1   1   1   1   s   s   1   1  x
8   x   2   2   2  2  2   2   2   2   2   2   2   2  x
9   x   5   5   5  5  8   8   8   8   8  10  10  10  x
10  x   6   6   6  5  8  16  16  16   8  10  11  11  x
11  x   x   x   x  x  x   x   x   x   x   x   x   x  x
Individuals' starting position path:
[(0, 11), (0, 3), (1, 0), (11, 10), (7, 13), (11, 4), (11, 12), (10, 0), (0, 4), (11, 9),
Best individual generation: 7
Best individual fitness: 112
Vertical turn preference: l
Horizontal turn preference: u
Time to complete algorithm: 1.29 seconds.
```

Test scenario parameters: Mutation 5%, Elitism 15%, Crossover from elitism 85%

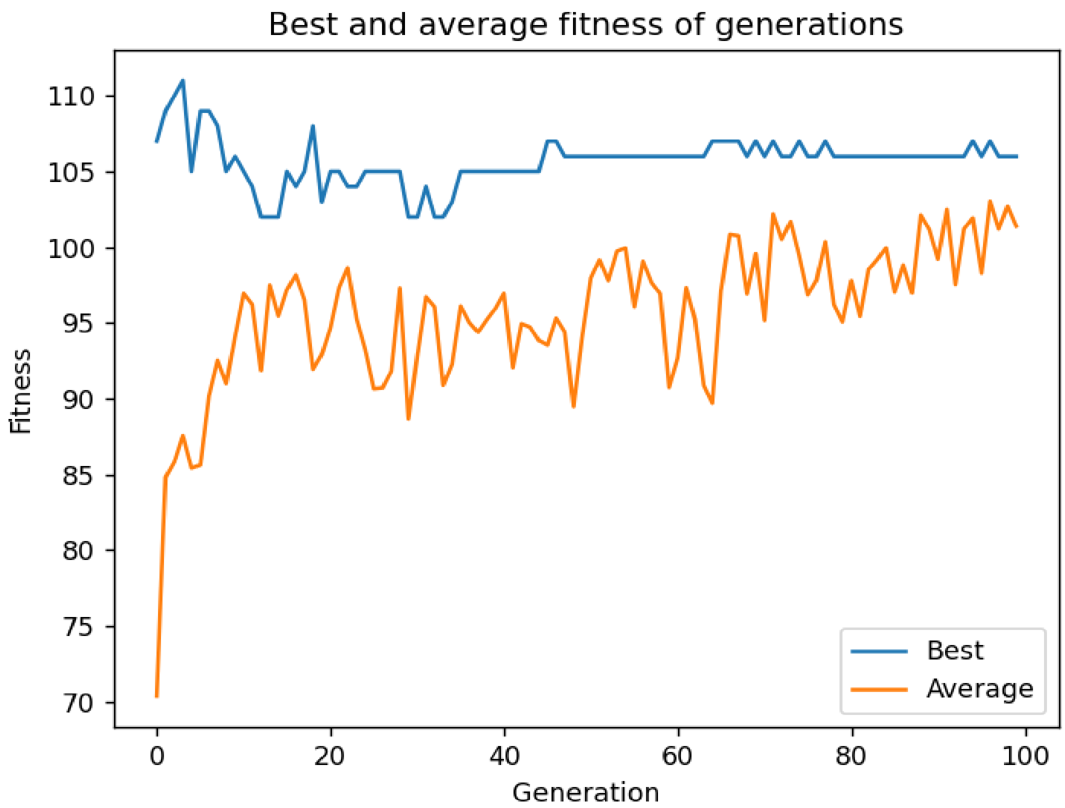



```
Best individual garden:
   0  1  2  3  4   5   6  7  8   9   10 11 12 13
0   x  x  x  x  x   x   x  x  x   x   x  x  x  x
1   x  5  1  9  7  11   0  1  4   6   6  2  3  x
2   x  5  1  9  7  11   s  1  4   6   6  2  3  x
3   x  5  s  9  7  11  11  1  4   6   6  2  3  x
4   x  5  9  9  7   s  11  1  4   6   6  2  3  x
5   x  5  9  s  7   8   8  1  4   6   6  2  3  x
6   x  5  9  1  7   8   8  1  4   6   6  2  3  x
7   x  5  9  1  7   8   8  1  4   s   s  2  3  x
8   x  5  9  1  7   8   8  1  4  10  10  2  3  x
9   x  5  9  1  7   8   8  1  4  10  10  2  3  x
10  x  5  9  1  7   8   8  1  4  10  10  2  3  x
11  x  x  x  x  x   x   x  x  x   x   x  x  x  x
Individuals' starting position path:
[(11, 7), (0, 11), (11, 12), (11, 8), (11, 1), (2, 0), (5, 13), (0, 10), (11, 11), (11, 4), (11, 7),
Best individual generation: 5
Best individual fitness: 113
Vertical turn preference: l
Horizontal turn preference: d
Time to complete algorithm: 1.42 seconds.
```

Martin Čajka
AIS ID: 116158

ROULETTE

Test scenario parameters: Mutation 10%, Roulette 15%, Crossover from Roulette Selection 85%

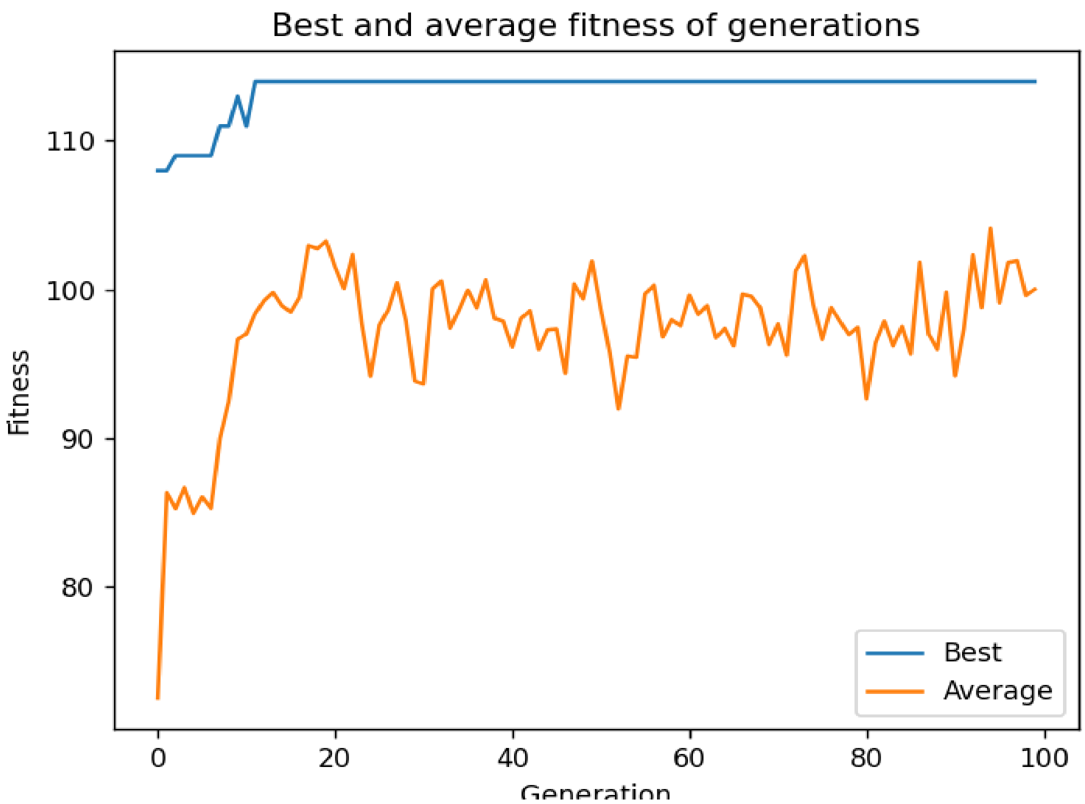



```
Enter the number corresponding to the desired selection: 2
Best individual garden:
   0   1   2   3   4   5   6   7   8  9  10 11 12 13
0   x   x   x   x   x   x   x   x   x  x  x  x  x  x
1   x   5  12  12  12  12  12  10  10  5  4  1  3  x
2   x   5  12  12  12  12   s  10  10  5  4  1  3  x
3   x   5   s  12  12  12  10  10  10  5  4  1  2  x
4   x   5   5   5   5   s  10  10  10  5  4  1  2  x
5   x   5   5   s   5   5   5   5   5  5  4  1  2  x
6   x   4   4   4   4   4   4   4   4  4  4  1  2  x
7   x   6   6   6   6   6   6   6   6  s  s  1  2  x
8   x   7   7   7   7   7   7   7   6  8  8  1  2  x
9   x  11  11  11  11   9   9   7   6  8  8  1  2  x
10  x   0   0   0  11   9   9   7   6  8  8  1  2  x
11  x   x   x   x   x   x   x   x   x  x  x  x  x  x
Individuals' starting position path:
[(0, 11), (3, 13), (1, 13), (0, 10), (5, 13), (11, 0), (0, 1), (10, 13), (5, 0), (5, 0), (7, 0), (11, 7), (0, 12),
Best individual generation: 3
Best individual fitness: 111
Vertical turn preference: r
Horizontal turn preference: d
Time to complete algorithm: 1.85 seconds.
```

Martin Čajka
AIS ID: 116158

Cvičenie: Streda 15:00

Test scenario parameters: Mutation 10%, Roulette 40%, Crossover from Roulette Selection 60%

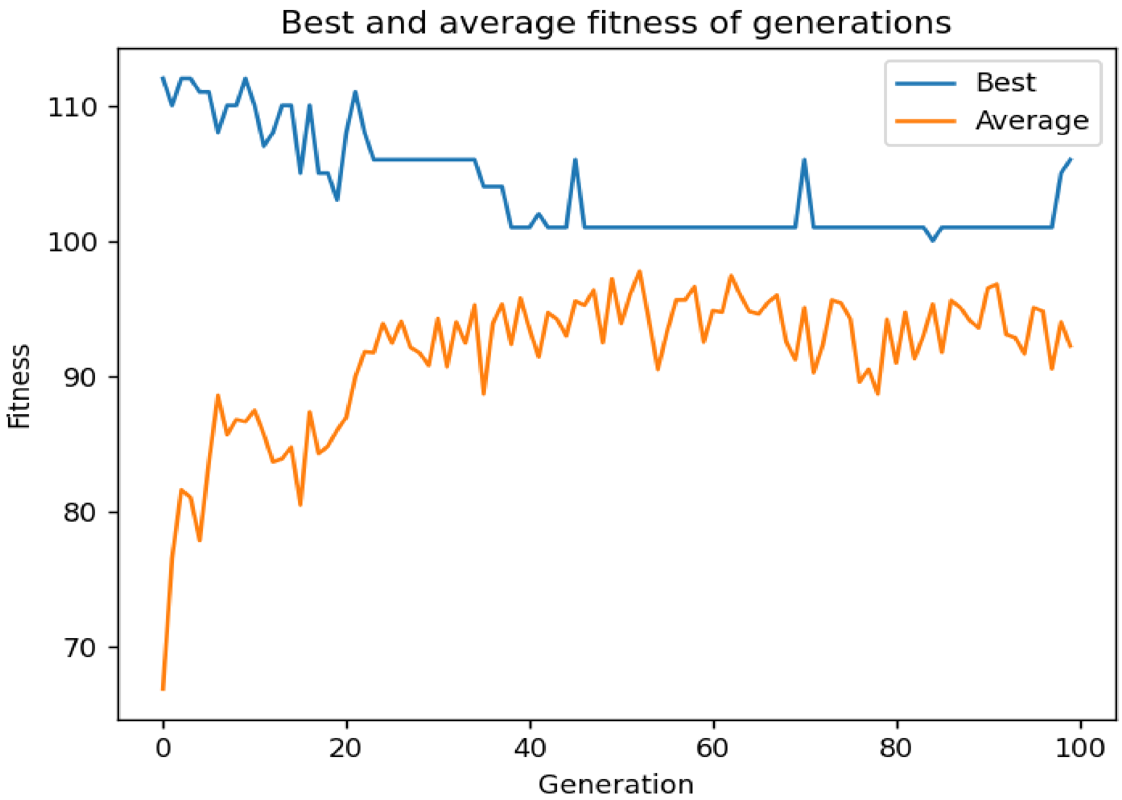



```
Best individual garden:
    0   1  2  3   4   5  6  7  8  9  10 11 12 13
0    x   x  x  x   x   x  x  x  x  x  x  x  x  x
1    x  10  9  8  11  11  8  8  3  6  6  4  5  x
2    x   9  9  8  11  11  s  8  3  6  6  4  5  x
3    x   3  s  8   8   8  8  8  3  6  6  4  5  x
4    x   3  8  8   8   s  8  8  3  6  6  4  5  x
5    x   3  8  s   8   8  8  8  3  6  6  4  5  x
6    x   3  8  8   8   8  8  8  3  6  6  4  5  x
7    x   3  3  3   3   3  3  3  3  s  s  4  4  x
8    x   2  2  2   2   2  2  2  2  2  2  2  2  x
9    x   7  7  7   7   7  7  7  7  7  7  7  7  x
10   x   1  1  1   1   1  1  1  1  1  1  1  1  x
11   x   x  x  x   x   x  x  x  x  x  x  x  x  x
Individuals' starting position path:
[(10, 13), (8, 0), (3, 0), (11, 8), (0, 8), (0, 11), (11, 6), (4, 0),
Best individual generation: 11
Best individual fitness: 114
Vertical turn preference: r
Horizontal turn preference: u
Time to complete algorithm: 1.15 seconds.
```

Martin Čajka
AIS ID: 116158

Cvičenie: Streda 15:00

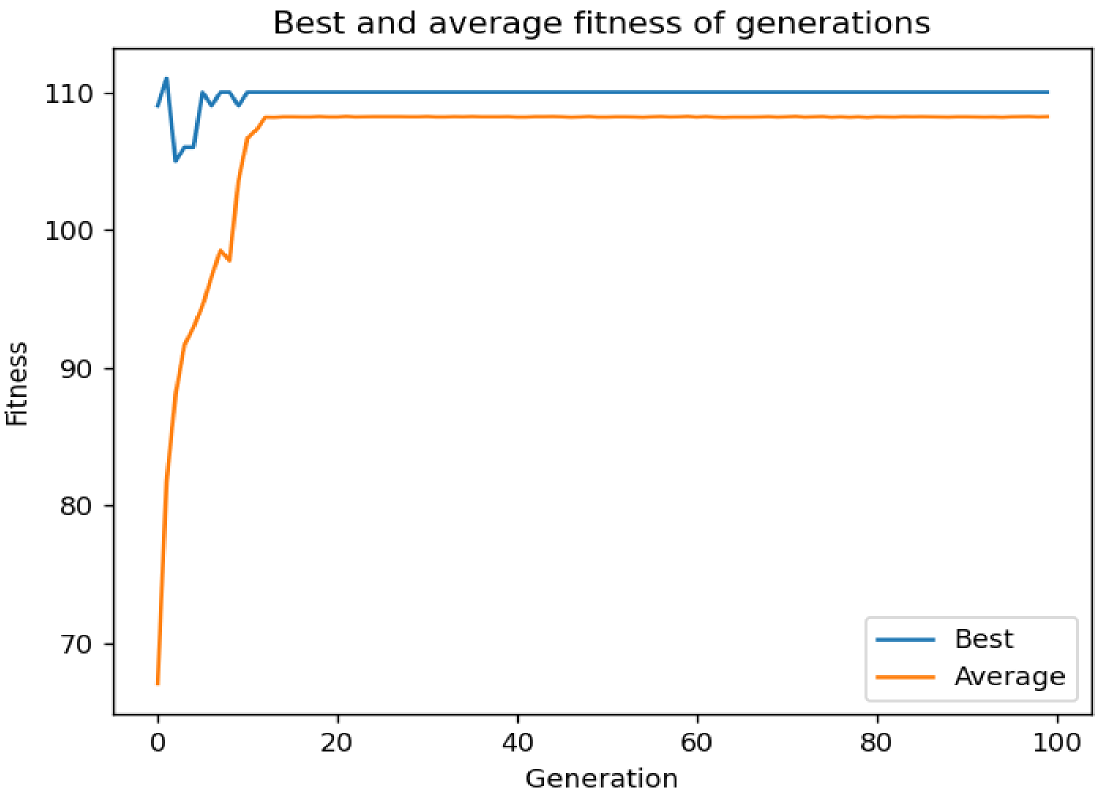
Test scenario parameters: Mutation 40%, Roulette 15%, Crossover from Roulette Selection 85%




```
Enter the number corresponding to the desired selection: 2
Best individual garden:
   0   1   2   3   4   5   6   7  8  9  10 11 12 13
0   x   x   x   x   x   x   x   x  x  x  x  x  x  x
1   x   4   4   4   4   4   4  10  6  8  9  9  1  x
2   x  10  10  10   0   0   s  10  6  8  9  9  1  x
3   x   7   s  10  10  10  10  10  6  8  9  9  1  x
4   x   7  10  10  10   s  10  10  6  8  9  9  1  x
5   x   7  10   s  10  10  10  10  6  8  9  9  1  x
6   x   7  10  10  10  10  10  10  6  8  8  8  1  x
7   x   6   6   6   6   6   6   6  6  s  s  8  1  x
8   x   2   2   2   2   2   2   2  2  2  2  8  1  x
9   x   3   3   3   3   3   3   3  3  3  2  8  1  x
10  x   5   5   5   5   5   5   5  5  3  2  8  1  x
11  x   x   x   x   x   x   x   x  x  x  x  x  x  x
Individuals' starting position path:
[(0, 12), (11, 10), (11, 9), (0, 6), (11, 8), (5, 13), (10, 13), (0, 1), (11, 4),
Best individual generation: 0
Best individual fitness: 112
Vertical turn preference: l
Horizontal turn preference: d
Time to complete algorithm: 1.29 seconds.
```

Martin Čajka Cvičenie: Streda 15:00
AIS ID: 116158

Test scenario parameters: Mutation 5%, Roulette 40%, Crossover from Roulette Selection 60%



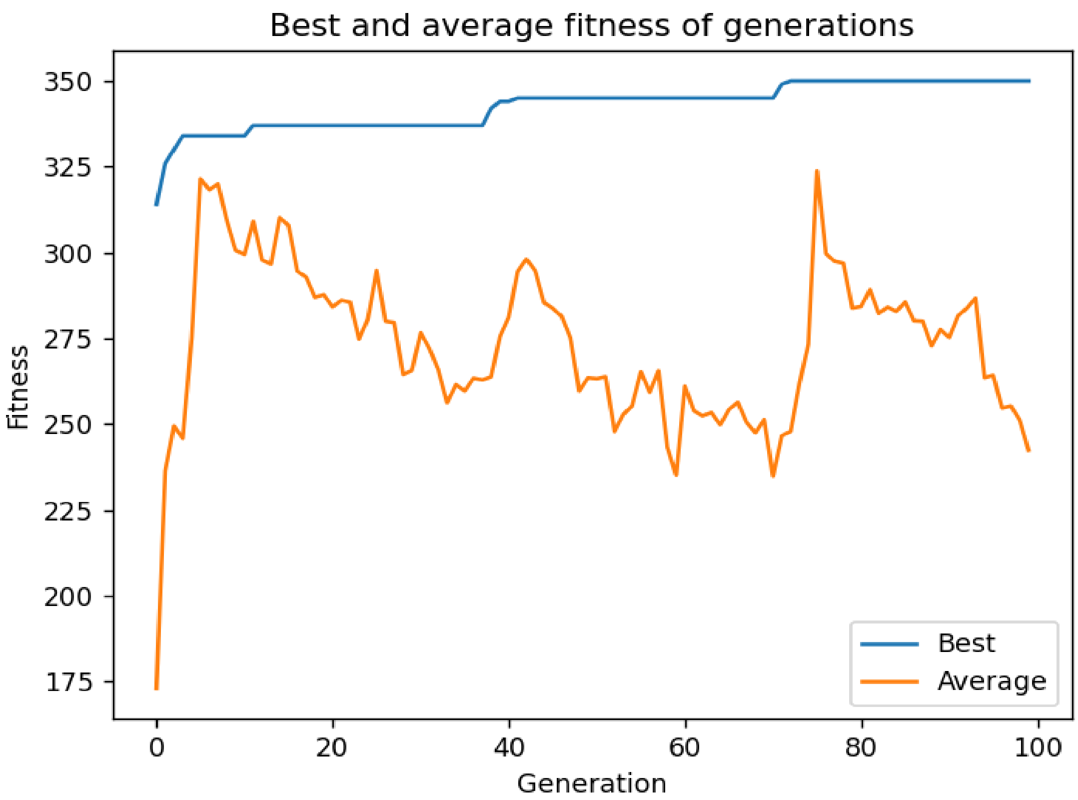

```
Best individual garden:
    0   1   2   3   4   5   6   7   8   9  10 11 12 13
0   x   x   x   x   x   x   x   x   x   x  x  x  x  x
1   x   2   2   2   2   2   2   2   2   2  2  1  7  x
2   x  10  10  10  10  10   s  10  10  10  2  1  7  x
3   x   0   s  10  10  10  10  10   0  10  2  1  7  x
4   x   5   5  10  10   s  10  10   0  10  2  1  3  x
5   x   5   5   s  10  10  10  10  10  10  2  1  3  x
6   x   2   2   2   2   2   2   2   2   2  2  1  3  x
7   x   4   4   4   4   4   4   4   4   s  s  1  3  x
8   x   8   8   8   8   8   8   8   4   9  9  1  3  x
9   x   8   8   8   8   8   8   8   4   9  9  1  3  x
10  x   6   6   6   6   6   6   6   4   9  9  1  3  x
11  x   x   x   x   x   x   x   x   x   x  x  x  x  x
Individuals' starting position path:
[(0, 11), (1, 0), (4, 13), (7, 0), (10, 13), (0, 3), (5, 0), (10, 0), (0, 12),
Best individual generation: 1
Best individual fitness: 111
Vertical turn preference: l
Horizontal turn preference: d
Time to complete algorithm: 1.44 seconds.
```

Martin Čajka Cvičenie: Streda 15:00
AIS ID: 116158

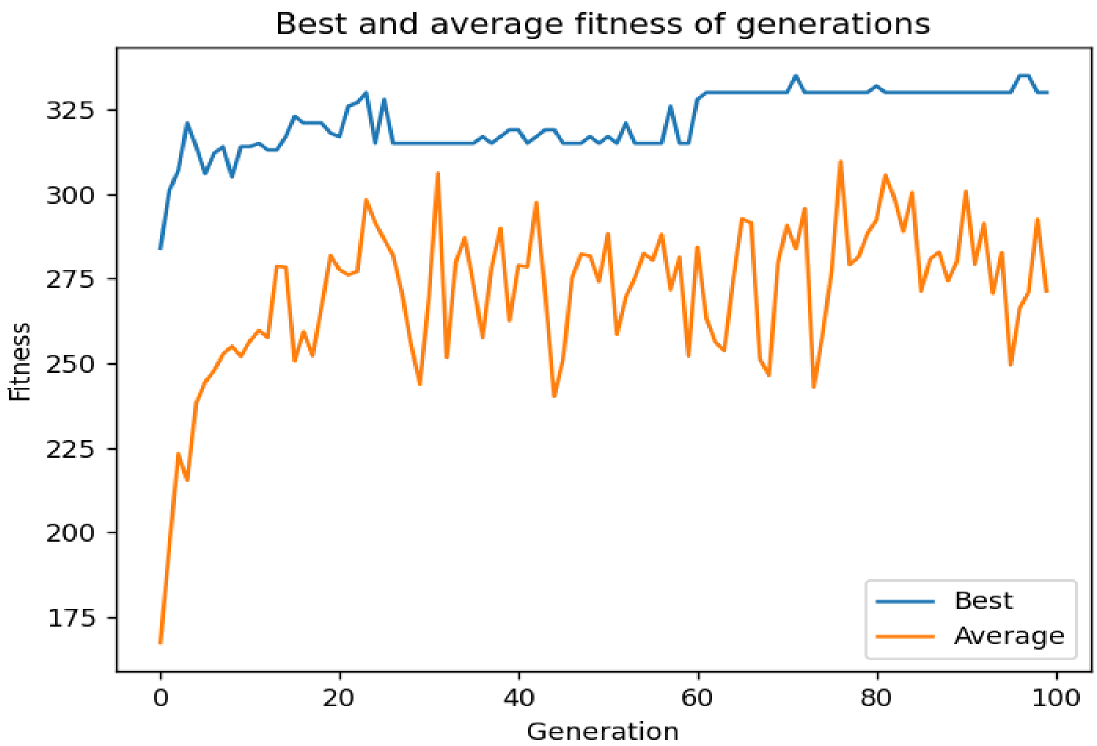
Board size: 20*20 Mutation 10%, Elitism/ Roulette 15%, Crossover 85%

Stone locations: [(2,2),(3,8),(4,5),(4,6),(7,15),(8,10),(11,12),(11,1),(12,10),(14,8)]

Elitism:



Roulette



Summary:

The previous graphs portray the difference between selections and various parameter settings. Overall, elitism has the tendency to find the local maximum in a couple generations and devolve from there. As the percentage of elitism tends to rise, variation goes down and fitness stays closer to the local maximum after teaching that point. Higher levels of mutation on the other hand create a lot of variability and elitism devolves generatively by quite some. If we decrease mutation, elitism tends to get better results with each generation.

In roulette selection under default settings generations progressively evolve. By increasing the percentage of selected individuals, the average fitness does not change much, but the fittest individuals are not lost as they are more often chosen. By decreasing mutation, variability is once again being lost and generations average rises quickly and does not change much. Increased mutation causes the fittest individuals to be lost, but the average does not change much, as mutations finds other fit individuals.

There is definitely room for improvement in my solution, which could be found by experimenting with the parameters of the algorithm. Other selection methods could be implemented, the structure of creating new generations could be done more complex to ensure more evolution, such as setting new rules for mutation, set different percentages for passing whole individuals to next generation and change the way the crossover method works. All of the changes would be quite exciting to see.