STU Bratislava, Fakulta informatiky a informačných technológií

Umelá inteligencia – Strojové učenie

Zadanie č. 3a – zenová záhrada

Dokumentácia

Meno: Andrii Rybak

AIS ID: 105840

Cvičiaci: Ing. Ivan Kapustík

Akademicky rok: 2021/2022

Andrii Rybak
ID: 105840

Content

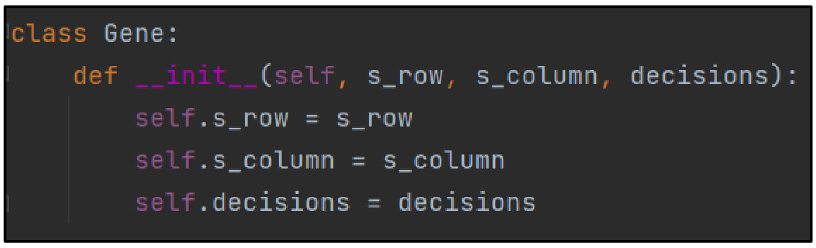
Andrii Rybak
ID: 105840

Algorithm and data representation

The solution is developed in Python 3.9 in PyCharm.

To solve the problem the classic genetic algorithm was used. First of all, the garden is created and is represented as list of lists. Unvisited cells are represented as **0** and stones are -1. Then the function `find_solution` is called. In this function the first generation is randomly generated.

Each individual consists of one chromosome, which is a list of `(columns + rows + stones)` number of genes. Gene is represented as class Gene and contains starting row, starting column and randomly generated list of decisions that specify the direction where the monk has to turn in case of obstacle. Such decision is made only when there are two possible variants of how to turn.



```
class Gene:
    def __init__(self, s_row, s_column, decisions):
        self.s_row = s_row
        self.s_column = s_column
        self.decisions = decisions
```

When the first generation was successfully generated, the fitness value of each individual is calculated and is placed in the list on corresponding index. Then the following loop is executed:

1. End if the best fitness value is the number of cells except stones.
2. Move the best individuals to new generation.
3. Until the new generation is not formed:
    a. chose 2 parents using roulette and reproduce 2 child using crossover method
    b. mutate one gene in each individual (the chance of mutation is predefined)
4. Calculate fitness value for new generation and go to step 1.

Crossover

The one-point crossover was used. The point is chosen randomly in such way, that new child cannot be exact copy of one of the parents. If some of child's gene is duplicated, it is replaced by the parent's unique one.

Elitism

Specified number of the best individuals will be moved to the new generation without changes.

Roulette

To choose parents was implemented classic roulette's logic. The chance to be chosen as parent is (individual's fitness value) / (sum of all individual values in the generation). The same individual can be chosen as parent more than once.

Fitness function

The monk starts from one of the sides of the garden and moves in direction of opposite side. When the obstacle occurs on the way, the monk turns left or right if it is the only possibility. In case when the monk can turn both directions, the decision is made according to the list “decisions” in current gene. The list contains characters ‘r’ and ‘l’, which mean right and left. When the decision was made, the next one will be made according to the next character in list “decisions”. If there are needed more decisions than the gene contains, the pointer on the next decision is set on the very beginning of the list.

If the monk starts from the corner cell, the direction of movement is determined by the first character in decisions list of the gene. If the character is ‘r’, the monk will move horizontally, otherwise if it is ‘l’ – vertically.

If the monk cannot start from the position specified in gene, it (gene) will be simply skipped.

Mutation

After child is created using crossover, there is specified chance that it will mutate. The mutation is provided by changing one random gene of the child to newly generated one. If the newly generated gene is already present in the individual, it (gene) will be generated again until it is unique.

Each individual does not have any duplicated genes, it is provided during the whole program run.

Parameter setting options

There are several parameters, that can be changed:

- **`DECISIONS_N`** – number of randomly generated decisions in every gene
- **`GENERATION_SIZE`** – number of individuals in every generation
- **`ELITISM_N`** – number of the best individuals that will be moved to the new generation without roulette
- **`GENERATIONS_LIMIT`** – maximum number of created generations
- **`NEW_GENE_CHANCE`** – chance that one gene in chromosome will be replaced by randomly generated one (range 0-1)

Andrii Rybak
ID: 105840

Tests

Number of tests ↓	GENERATIONS_LIMIT	DECISIONS_N	GENERATION_SIZE	ELITISM_N	NEW_GENE_CHANCE	Found in generation (average):
10	600	5	50	5	0,1	112
10	600	5	50	5	0,2	36
10	600	5	50	5	0,5	32
10	600	5	50	5	0,8	24
10	600	5	50	5	1	19

As can be seen from the table above, it is good when one random gene in every child is replaced with newly generated one. It significantly speeds up the solution search. Any mutation perfectly dilutes the genetic pool, what helps to get out from local maxima.

There was also tried a mutation with chance to replace 0 and more genes (even more than 1), but after trying all possible settings, the results were not as good as replacing maximum 1 gene.

Let's now play with ELITISM_N setting

Number of tests ↓	GENERATIONS_LIMIT	DECISIONS_N	GENERATION_SIZE	ELITISM_N	NEW_GENE_CHANCE	Found in generation (average):
10	600	5	50	1	1	57
10	600	5	50	2	1	48
10	600	5	50	3	1	41
10	600	5	50	4	1	22
10	600	5	50	5	1	20
10	600	5	50	6	1	22
10	600	5	50	7	1	23
10	600	5	50	8	1	32
10	600	5	50	10	1	70

Results from the table above show that moving 5 the best individuals to the new generation is the best option. So, in this implementation I recommend setting the variable ELITISM_N to 10% from the size of the population.

The results also indicate that moving more than 10% individuals from the population leads to genetic domination, what means that the algorithm gets stuck in some local maxima and cannot get out from it.

Moving less than 10% individuals increases the chance that the best genes will be simply lost, so that the algorithm requires more time to find the solution.

Let's now experiment with generation size:

Number of tests ↓	GENERATIONS_LIMIT	DECISIONS_N	GENERATION_SIZE	ELITISM_N	NEW_GENE_CHANCE	Found in generation (average):
10	600	5	25	3	1	59
10	600	5	50	5	1	20
10	600	5	75	8	1	23
10	600	5	100	10	1	22
10	600	5	125	13	1	19
10	600	5	150	15	1	17
10	600	5	200	20	1	15

The table above shows that the results are getting better by increasing the generation size. This can be explained in such way that with bigger generation there is a bigger chance that during reproduction there will be created a child that is a solution, because more combinations are made. However, it takes much more time to generate populations of such size, so it is recommended to use around 50 individuals for each generation.

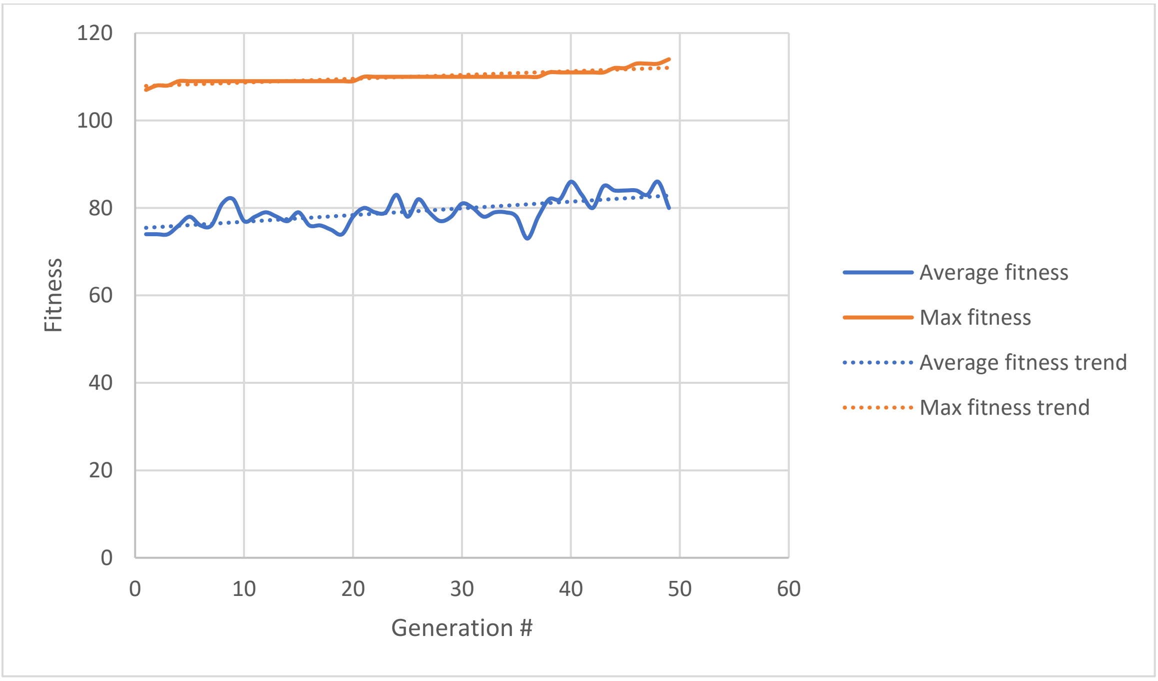
Experiments with decision number showed that the best results can be got by setting the parameter `DECISIONS_N` at least to 3. More decisions do not have any influence on algorithm productivity. Less numbers significantly slow down the program.

Taking into account all tests above, there can be made a conclusion that the best parameter settings are:

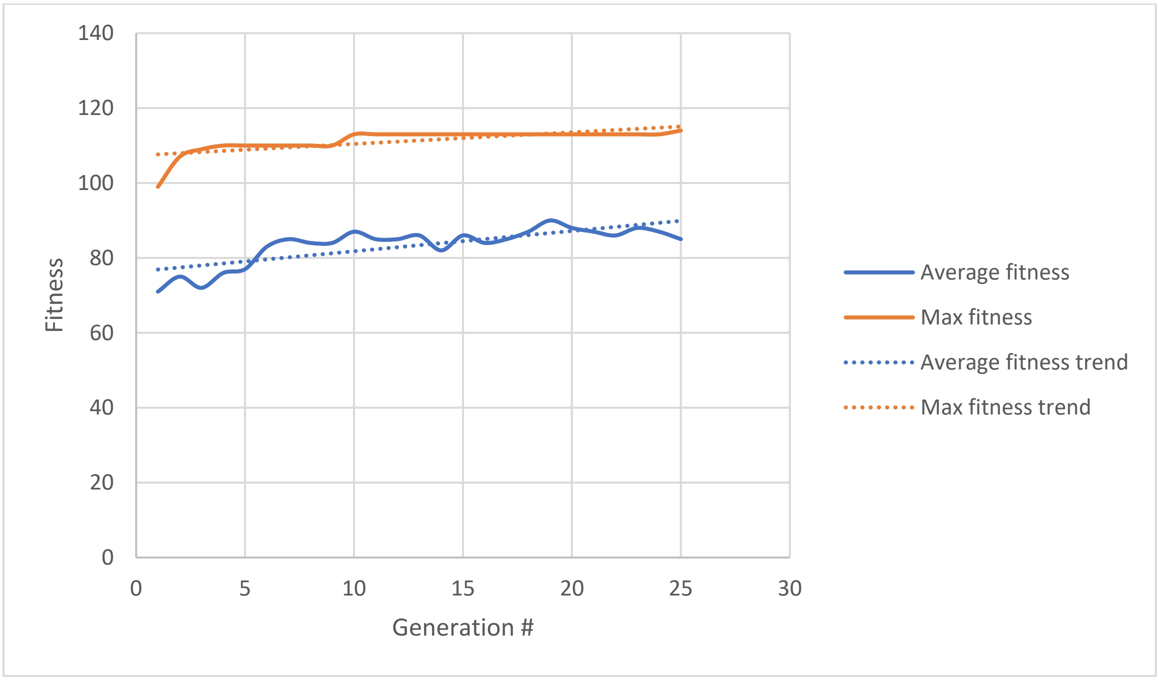
GENERATIONS_LIMIT	DECISIONS_N	GENERATION_SIZE	ELITISM_N	NEW_GENE_CHANCE	Found in generation (average):
600	5	50	5	1	19

The following settings provide the fastest search of the solution. In this case, time to create and get fitness of one population is about **0.0345** seconds.

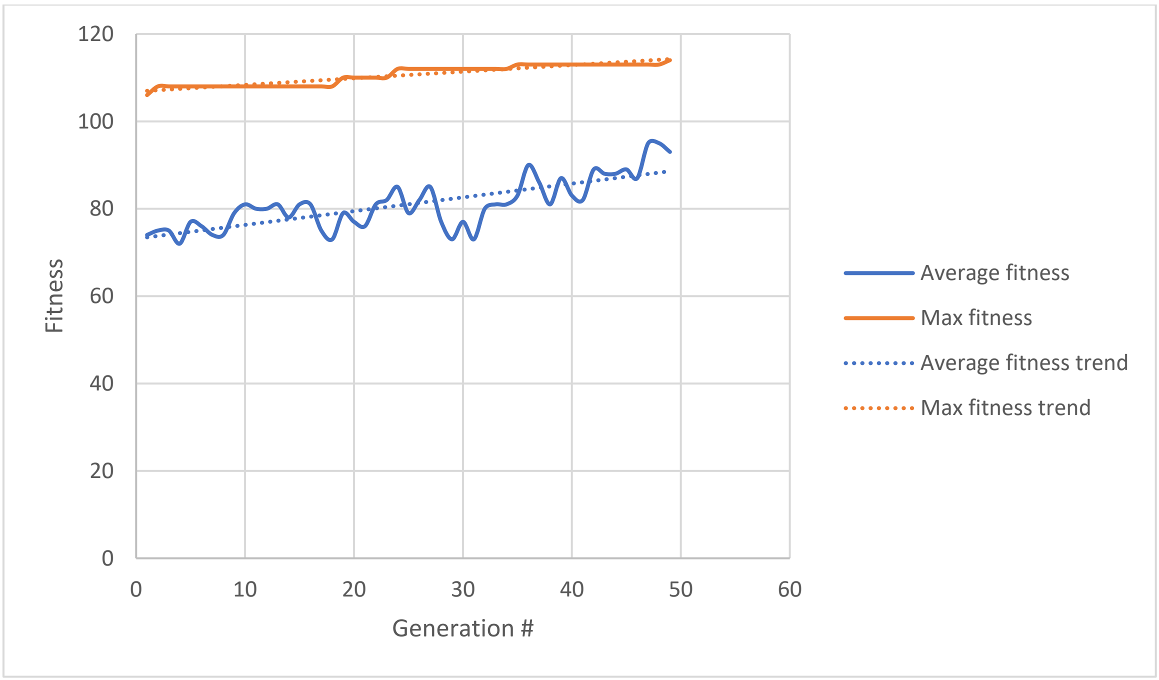
Graphs for the best settings mentioned above:



In this case the solution was found in 48-th population.



The solution was found in 24-th generation.



The solution was found in 48-th generation

As can be seen from the graphs above, the maximum fitness gets better with every new generation. The average fitness varies, but the general trend rises. This indicates that the evolutionary algorithm works, and the further generations are becoming better.

Possible optimizations

The solution can be optimized by using Rank Selection in the end of the run when the individuals in the population have very close fitness values, because on this point each individual have an almost equal share of the pie. In such situation it becomes harder for roulette to choose the best individuals to be parents.

Time and space complexity

Time and space complexity cannot be precisely calculated for evolutionary algorithm but is considered to be linear in both cases. The program stores only current population and needs linear amount of operations to create a new one.

Conclusion

In this implementation were used two types of parent selection: roulette and elitism. Roulette provides relatively high genetic variety. And the elitism significantly speeds up the solution search by moving the best individuals to the new population, saving the best genes from being lost.

Mutations also have significant influence on algorithm productivity. Without random mutations, there is a high chance that the search will get stuck in some local maxima and will not be able to get out from it – the population will simply consist of copies of one individual and the solution will be never found.

The results showed that the best settings for this implementation are:

GENERATIONS_LIMIT	DECISIONS_N	GENERATION_SIZE	ELITISM_N	NEW_GENE_CHANCE
600	5	50	5	1

Such options provide finding the solution in 18-25th generation (in average) for the best execution time. Time to create and get fitness of one population with 50 individuals is about **0.0345 seconds** in average.