Introduction to Artificial Intelligence **Anna Miklewska**
Task 1: Genetic Algorithm 238461

1. Introduction and goals

The knapsack problem is an optimization problem. Given a set of items, each with weight and value, it aims to find the subset which has the highest total value without exceeding a given maximum total weight. In this case an extended version of the problem, in which each item is additionally characterized by its size and the subset cannot exceed a given maximum total size will be considered. It is a 0-1 knapsack problem, which means each item can be taken only once.

A solution that can be used to solve the knapsack optimization problem is genetic algorithm. It is a metaheuristic, inspired by natural selection, which uses operations such as crossover, mutation and selection to evolve a set of possible solutions and find the optimal one.

The goal of this assignment was to implement and evaluate the implementation of genetic algorithm used for solving a 0-1 knapsack problem. Analyze the impact of factors such as crossover probability, selection and population size on the results and performance. Moreover, the task generator for the knapsack problem and task loading had to be implemented.

Additionally, the result obtained with genetic algorithm using the optimal factors had to be compared with a result obtained by a non-evolutionary method. A dynamic programming solution was used.

2. Setup and implementation

2.1. Task generation and loading

A set of randomly generated individuals is created and saved to a csv file. The file contains additionally parameters for the task: number of items to choose (n), maximum carrying capacity of the knapsack (w), and the maximum knapsack size (s).
Generated items meet the following criteria, given in the assignment.

- For each item the weight w_i, size s_i and value c_i:
  - $1 < w_i < 10 \cdot \frac{w}{n}$
  - $1 < s_i < 10 \cdot \frac{w}{n}$
  - $1 < c_i < 10 \cdot n$
- For sets of items:
  - $\sum_{i=1}^{n} w_i > 2w$
  - $\sum_{i=1}^{n} s_i > 2s$

2.2. Initialization of a population

The population is initialized with individuals created at random. Each individual is represented by an object with a fitness value and a list of 0 and 1 values representing, whether a corresponding item was taken, referred to as a backpack. For individuals in the initial population the probability of taking an item is set at 0.2 to reduce the number of individuals who exceed the maximum size or capacity of their knapsacks.

2.3. Fitness function of an individual

The fitness function evaluates the individual based on the items they it is carrying. The fitness value is equal to total value of items it is carrying, unless maximum size or capacity are exceeded. In such case the fitness value is zero.

2.4. Selection method

The selection method used is tournament. It selects a given number of individuals at random from the population and returns the one with the highest fitness value. This number is referred to as tournament size.

2.5. Elitism

In addition to tournament selection an elitist selection is used. In the process of constructing a new population the fittest individuals of the previous one are used. This way the best solutions are kept.

The percent of the individuals taken from the previous is set as a parameter.

2.6. Crossover operator

The crossover is a way to create new individuals. The offspring is created by combining backpack of two parents. The lists representing backpacks are split at a cutting point chosen at random and concatenated.

Additionally, a crossover rate is set, which is a parameter defining the probability at which the crossover occurs. If it does not, one of the parents are taken instead.

2.7. Mutation operator

Mutation is used to maintain diversity between populations. It is implemented as leaving or taking items. The number of items to be changed depends on the number of items to choose from (size of the backpack) and a mutation rate parameter.

Particular items to be changed are selected at random.

2.8. Genetic algorithm

The genetic algorithm is implemented based on the pseudocode provided in the assignment. The most significant difference is adding elitism.

2.9. Dynamic programming solution

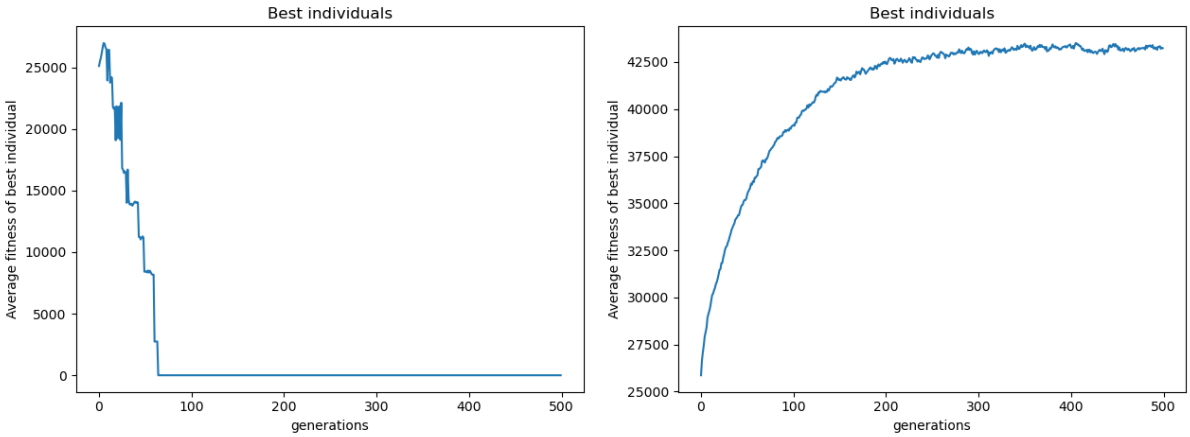
An alternative solution to genetic algorithm was implemented. It divides the problem into subproblems and uses them to find the best solution. In this case the size of items is not used for optimization, only as an additional constraint.

2.10. Default initial setup

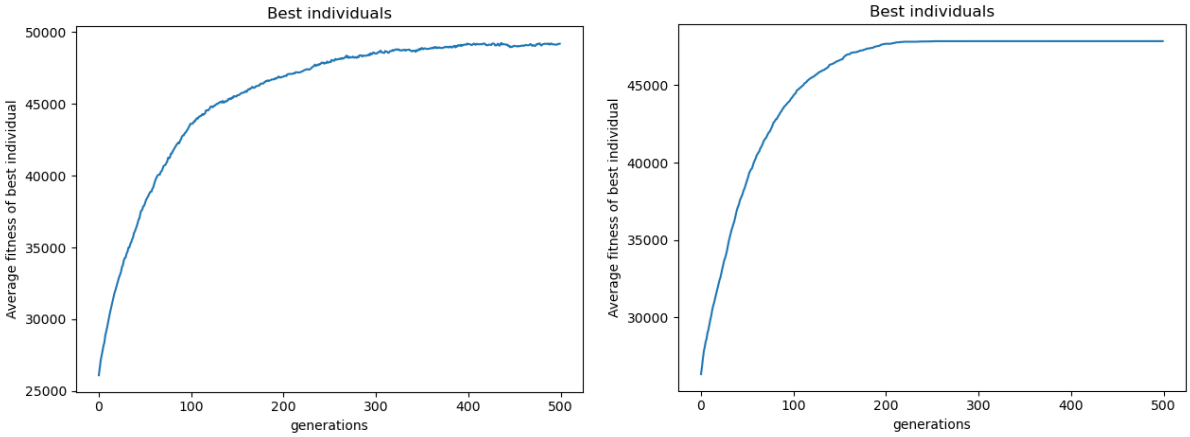
Population size	100
Crossover rate	0.8
Tournament size	0.3
Mutation rate	0.002
Elitism rate	0.2
Generations	500
No. of tests performed	10

Table 1 Default input values

3. Analysis of the impact of the crossover probability



Crossover = 0.2 Crossover = 0.5



Crossover = 0.8 Crossover = 1

Fig. 1 Average values of best individuals per generation for different crossover rates

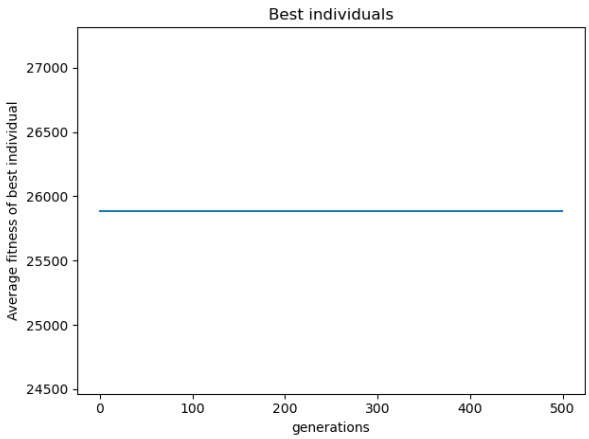
Crossover rate	Avg. time to execute [s]	Avg. best individual's fitness val.
0.2	26.9601	0
0.5	42.9011	43,235
0.8	57.53586	49,197
1	56.86254	47,841

Table 2 Results for different crossover rates

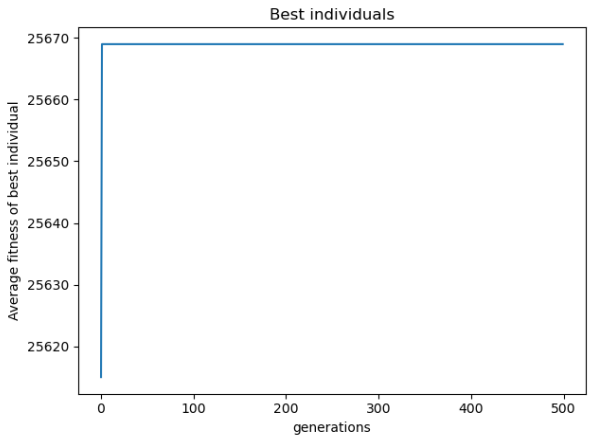
The crossover rate influences the time of execution of the algorithm. Lower crossover rate means that more individuals from previous generations will remain in the new one. For lower values of crossover rate the best value is achieved slightly slower and there are more noticeable differences between average value of the best individual in following generations.

If the crossover rate is too low, in this case 0.2, the algorithm does produce better individuals in each generation.

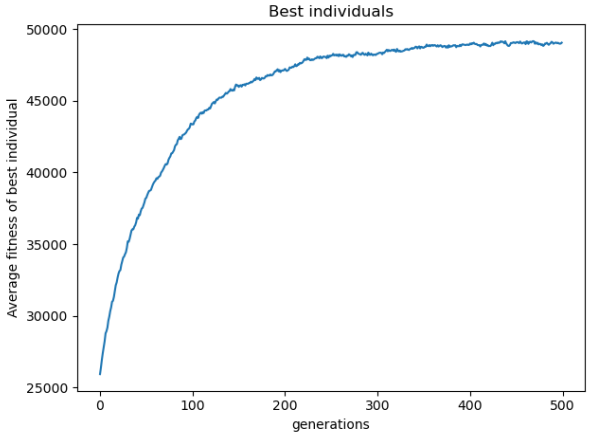
4. Analysis of the impact of the mutation probability



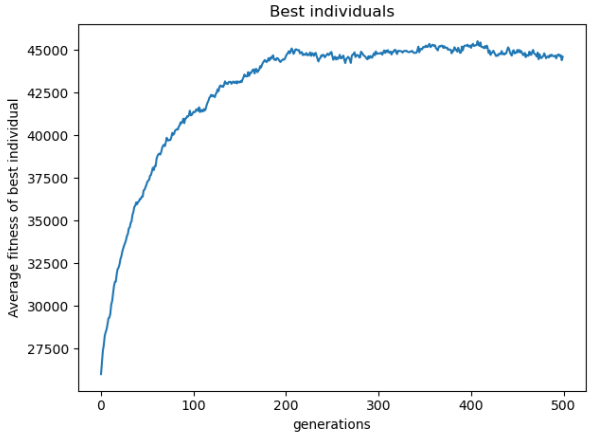
Mutation rate: 0.0001



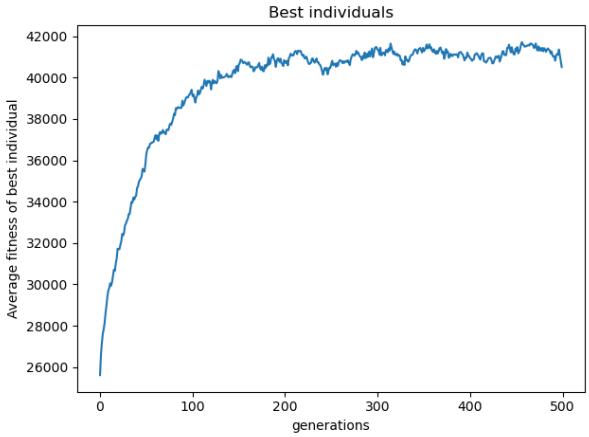
Mutation rate: 0.001



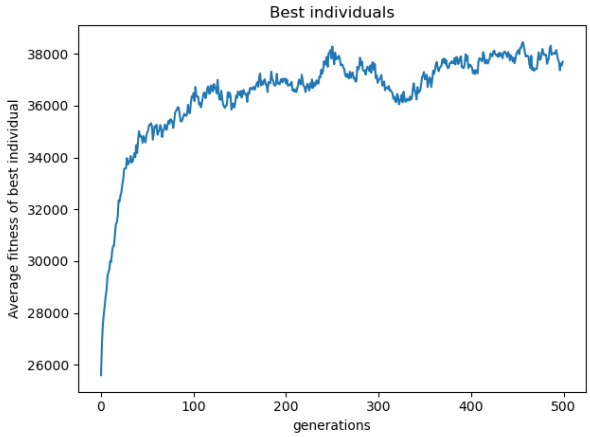
Mutation rate: 0.002



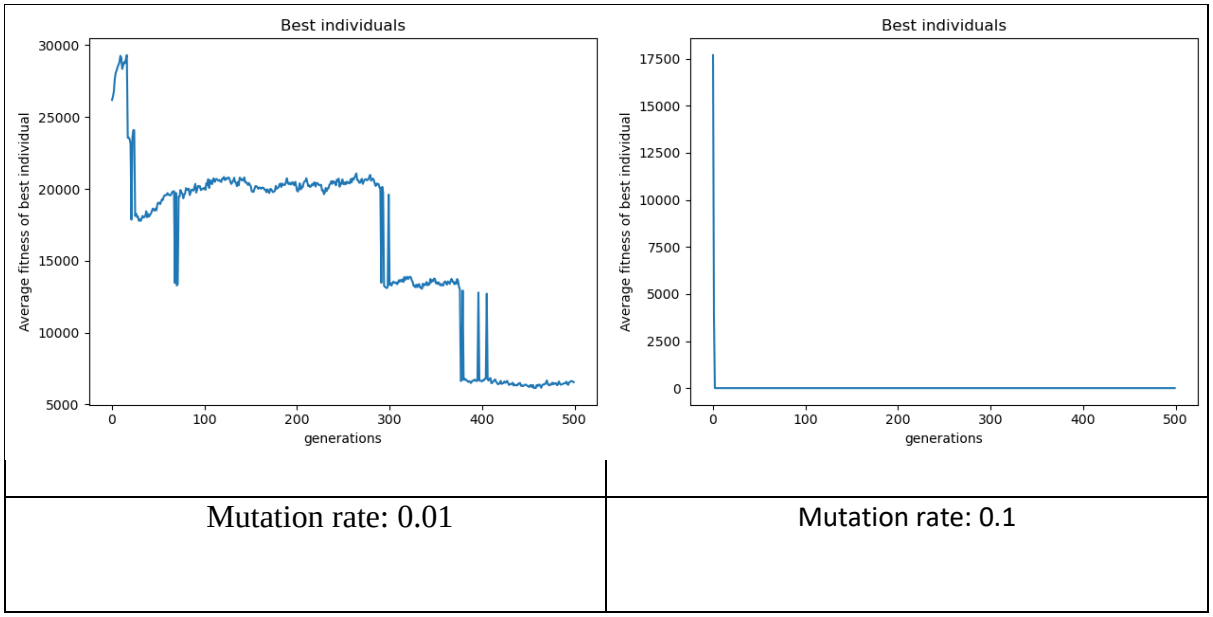
Mutation rate: 0.004



Mutation rate: 0.006



Mutation rate: 0.008



Mutation rate: 0.01	Mutation rate: 0.1

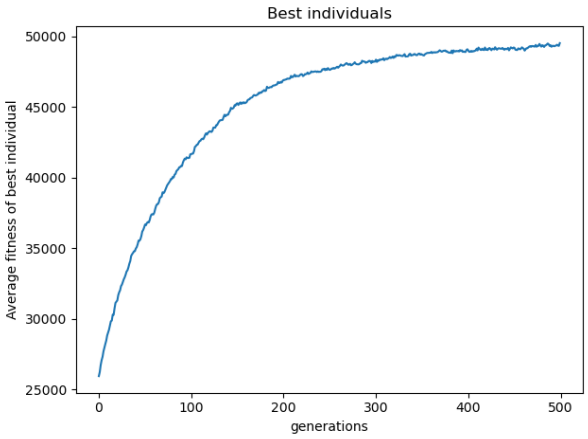
Fig. 2 Average values of best individuals per generation for different mutation rates

Mutation rate	Avg. time to execute [s]	Avg. best individual's fitness val.
0.0001	61.30326	25,889
0.001	59.75354	25,669
0.002	52.1254	49,057
0.004	47.11288	44,603
0.006	47.9494	40,514
0.008	45.883	37,701
0.01	42.71794	6,551
0.1	28.18636	0

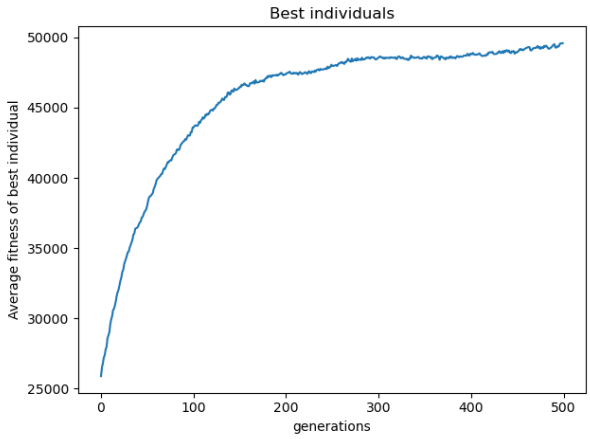
Table 3 Results for different mutation rates

Mutation rate influences the number of individuals added to the new generation that are randomly changed. It influences heavily both the produced solution and the time of execution. The smaller mutation rate, the higher the time of execution. Additionally, if the mutation rate is too low the results will not improve between generations. If the mutation rate is too high, the algorithm produces individuals with lower fitness value in each generation, to the point of producing a generation in which the best individual has a fitness value of zero.

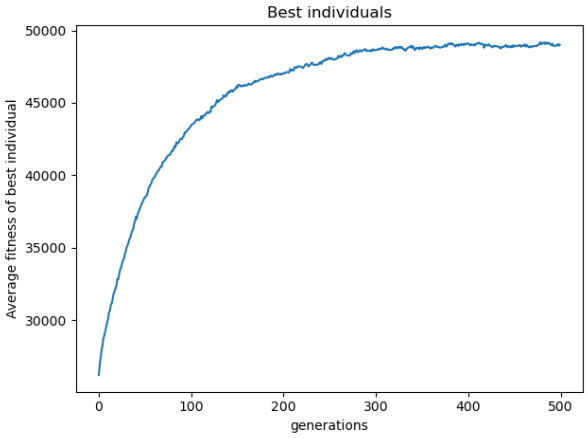
5. Analysis of the impact of tournament size



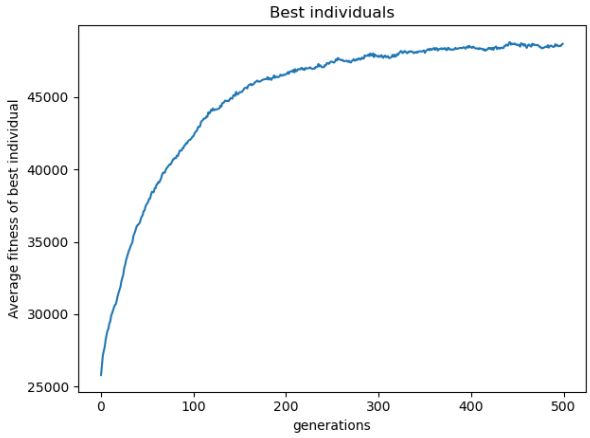
Tournament size: 10/100



Tournament size: 30/100



Tournament size: 50/100



Tournament size: 80/100

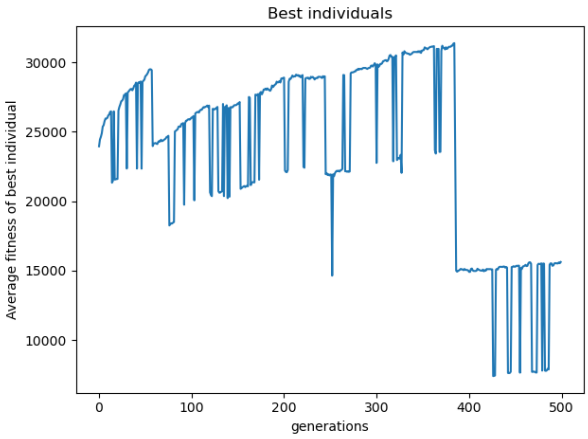
Fig. 3 Average values of best individuals per generation for different tournament sizes

Tournament size	Avg. time to execute [s]	Avg. best individual's fitness val.
10/100	51.93634	49,518
30/100	51.82948	49,596
50/100	52.15194	49,009
80/100	56.03068	48,670

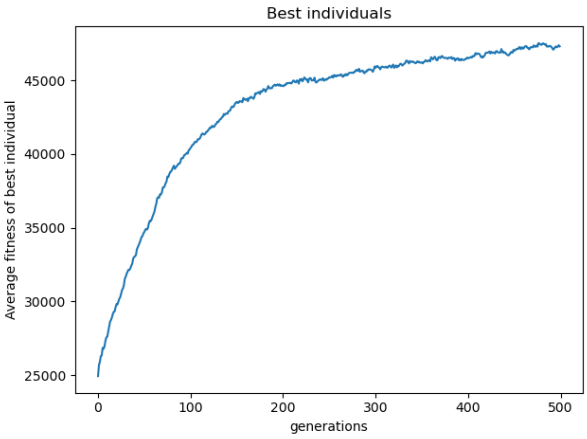
Table 4 Results for different tournament sizes

The tournament size dos not influence results of the algorithm greatly. There is no significant difference between times of execution or average best individual's fitness value for different tournament sizes.

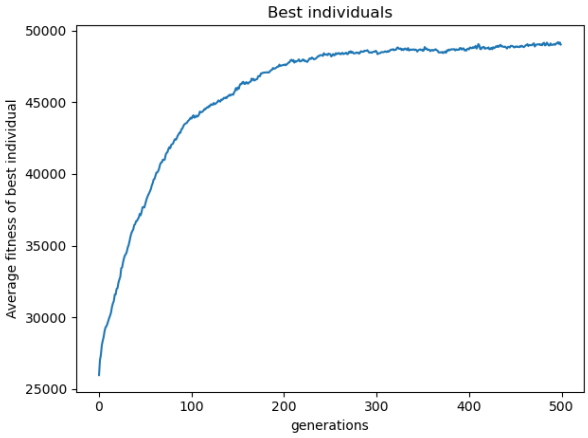
6. Analysis of the impact of population size



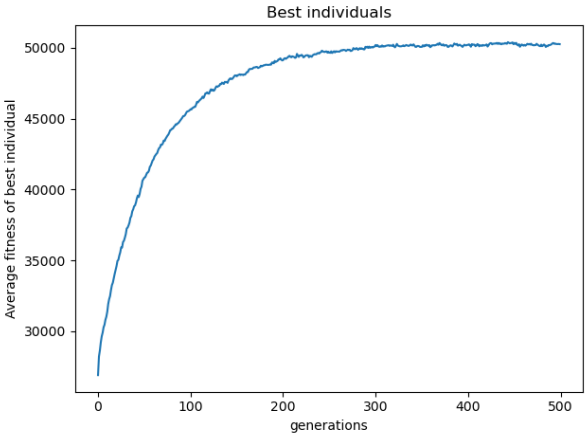
Population size: 10



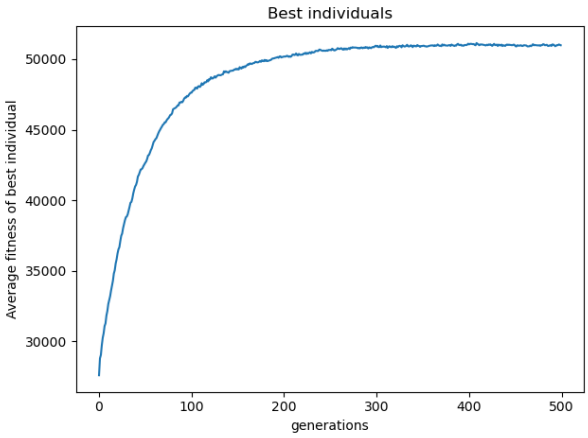
Population size: 50



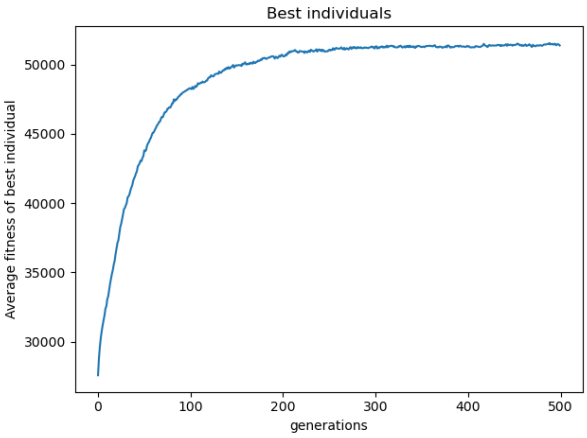
Population size: 100



Population size: 200



Population size: 500



Population size: 1000

Fig. 4 Average values of best individuals per generation for different populations sizes

Population size	Avg. time to execute [s]	Avg. best individual's fitness val.
10	4.42096	15,619
50	26.69108	47,297
100	51.07424	49,017
200	115.64274	50,260
500	290.63994	50,988
1000	909.8153	51,376

Table 5 Results for different population sizes

Both the time of execution and average best individual's fitness values grow with the size of the population until a certain point, in this case until the size is about 200 to 500. Over that there is no significant improvement in the avg. best individual's fitness value, but the time needed for execution keep growing. For that reason, the optimal population size out of those tested is 500

7. Comparison of the best solution with a non-evolutionary method

The non-evolutionary method used obtains the solution by dividing the problem into subproblems and aims to find the best solution. It means, that it is better for small problems. In case of problems with great number of items that can be taken genetic algorithm provides the possibility of finding an approximation of the best solution, which can save time and resources, making the genetic algorithm a better practical solution in such case.

8. Conclusion

Best parameters for the genetic algorithm:

Population size	100
Tournament size	0.3
Mutation rate	0.002
Elitism rate	0.2
Generations	500
No. of tests performed	5

Table 6 Best input values

The results yielded by genetic algorithm are heavily influenced by values of factors used for computation. It is crucial to test which are the best in order to find the best solution. Additionally, it should be decided what is considered a solution, whether the algorithm should stop after a certain number of iterations, or after creating a solution that is just close enough to the desired result.

In this implementation elitism was used as one of selection methods. It leads to finding better solutions quicker but introduces a risk of going to the nearest local maximum. This problem could be partially solved for instance by mutating the fittest individuals at random.