Laboratory 1

Introduction to artificial intelligence
Task 1: Genetic Algorithm

Jan Kocoń

Goal

In the first part of the task Genetic Algorithm, the aim of the laboratory is to learn about the optimization problem and the method called genetic algorithm. After getting acquainted with the description of the problem and the description of the operation of the genetic algorithm, the following elements should be implemented:

- task generator for the knapsack problem
- method of loading a task from a file
- a method for creating a random population
- the fitness function of an individual
- tournament selection method
- crossover operator
- mutation operator
- genetic algorithm

Then the following evaluation tasks should be done:

- Analysis of the impact of the crossover probability on the results of the algorithm (min. 3 different probability values)
- Analysis of the influence of the mutation probability on the results of the algorithm (min. 3 different probability values)
- Analysis of the impact of tournament size on the results of the algorithm (min. 3 different tournament size values)
- Analysis of the impact of population size on algorithm performance (min. 3 different population size values)

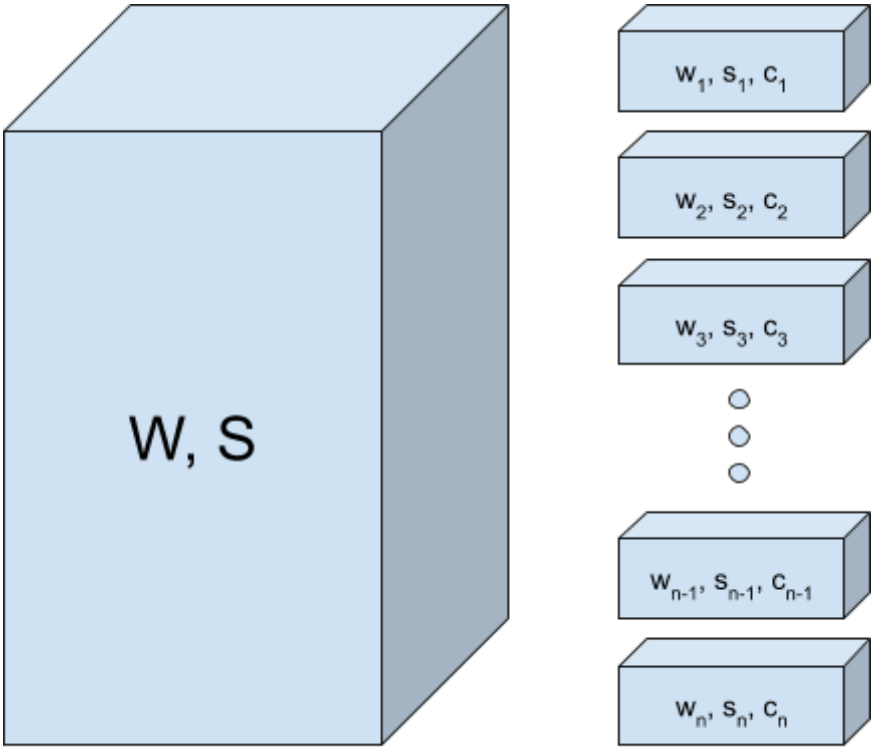
Task description

The knapsack problem (rucksack problem) is a known optimization problem. The classic version of this problem assumes a set of objects, each of which has a certain weight and price. In addition, the knapsack is defined by a parameter that determines the maximum load capacity. The task is to pack selected items into the knapsack in such a way that the total value of the items in the knapsack is as high as possible and at the

same time the total weight of the items does not exceed the maximum permitted carrying capacity. This problem is often presented with the example of a thief who intends to rob a shop. The thief wants the value of the stolen items to be as high as possible, but is not able to take everything as he has a knapsack with a limited carrying capacity. Similar problems are often considered in resource allocation tasks, cryptography, computational complexity theory, etc. There are also many variants of this problem. In the classic variant, the item can be taken away or not. In the unbounded knapsack problem we assume that there are infinitely many copies of a given object, while in the bounded knapsack problem the number of copies of a given object is finite and defined for each type of object. The given variants are discrete variants of the knapsack problem. There is also a continuous/fractional knapsack problem, in which it is possible to take fractions of objects.

Multidimensional knapsack problem

An extended version of the knapsack problem is proposed for implementation in the laboratory. There are n items, each one is a single copy. Apart from the weight, the item is also characterized by its size. Additionally, the knapsack is defined by two parameters: maximum load capacity and size. For simplicity, we do not assume a specific shape of the items. The task is extended by an additional criterion that the sum of the objects' sizes cannot exceed the size of the backpack.



A drawing showing an overview knapsack characterized by parameters W, S and n of objects, each characterized by parameters w, s, c.

Definition

Let it be given n items, each *i*-th item is described by its weight w_i, value c_i and size s_i. The knapsack has a given maximum load capacity W and size S. The solution to the task is a vector x of length n. Formal notation:

- Given:
  w, c, s, W, S
- We maximise the value:
  $\sum_{i=1}^{n} c_i x_i, \quad x_i \in \{0, 1\}$
- Weight limit:
  $\sum_{i=1}^{n} w_i x_i \leq W$
- Size limitation:
  $\sum_{i=1}^{n} s_i x_i \leq S$

Genetic algorithm

A genetic algorithm is a metaheuristics inspired by the theory of evolution and being one of many methods included in so-called evolutionary algorithms. This method reflects a natural selection process in which the strongest individuals are selected for reproduction to produce children in the next generation.

In a genetic algorithm, the population of solutions to the problem (also called individuals) in a given optimisation problem evolves towards better solutions. Each individual has a set of properties (defined by the genotype, i.e., a set of chromosomes) that can mutate and be transmitted in fragments in a crossover process that mimics reproduction in nature. Traditionally, solutions are represented in the binary system (sequences of zeros and ones), but other encodings are also possible.

Evolution usually starts with a population of randomly generated individuals and is an iterative process. The population in the i-th iteration is also referred to as the i-th generation. In each generation the fitness of each individual in the population is assessed; fitness is usually the value of the target function in the optimisation problem being solved. The most fitted individuals (i.e., those that perform better in the environment than others) are selected from the current population, and the genome of each individual is modified (chromosome fragments are exchanged between individuals and possibly randomly mutated) to create a new generation. The new generation of solutions (new population) is then used in the next algorithm iteration (new generation). Usually the algorithm ends when the number of iterations reaches a predefined limit, or there is no statistically significant quality improvement for the best individual in subsequent populations.

A typical genetic algorithm requires the following elements to be defined:

- the genetic representation of the individual in the domain of acceptable solutions
- the fitness function to assess the quality of the individual

The standard representation of an individual in a population is an array with values of 0 or 1. Other types and structures of data are of course possible, but in the classic approach (not always applicable to the optimisation problem) it is very easy to define basic operations such as crossover, due to the fixed size of each individual. Variable length representations can also be used, but in this case the implementation of crossover is more complex.

After defining the individual representation and the fitness function, the next step in the genetic algorithm is to initialise the initial population of solutions. This population is then iteratively improved through mutation, crossover and selection operators.

Initialisation

The size of the population depends on the nature of the problem, but usually contains hundreds or thousands of possible solutions. Often the initial population is generated randomly, allowing you to explore the search space in many directions, without defining a particular direction at the very beginning. Sometimes, however, it is possible to create individuals in the initial population in such a way that they represent a solution that is in the area where optimal solutions are likely to be found.

Fitness function and selection

In each subsequent generation, a part of the existing population is selected for reproduction to form a new generation. Individual solutions are selected in a process that uses the fitness function, in which solutions with a higher value are usually selected more frequently. Some selection methods assess the usefulness of each solution and choose the best of them. Other methods assess only a random sample of the population, as in the previous approach the process can be very time consuming.

The fitness function is defined by the genetic representation of the individual and measures the quality of the solution represented. The fitness function is always dependent on the problem. For example, in a knapsack problem the aim is to maximise the total value of items that can be placed in a knapsack of a certain fixed size and carrying capacity. The solution can be represented by an array of n bits, where each ith bit refers to the ith object, and the value of the bit (0 or 1) represents whether the object is in the knapsack or not. Not every solution is good, because it is possible that the total size of the items may exceed the capacity of the backpack (and/or their total weight may exceed its carrying capacity). The value of the fitness function for a given solution is the sum of the values of all the items in the knapsack if the restrictions are met, or 0 otherwise.

Crossover and mutation

The next step is to create the next generation, i.e. to create a new population of solutions from among those chosen in the selection process, by combining genetic operators: crossover and mutation.

A pair of parents is selected for reproduction from a pre-selected pool. The individual-child is created by using crossover and mutation methods. A new solution is created, which usually has many characteristics inherited from its parents. The process of selecting the parents for reproduction lasts until a new population of individuals of appropriate size is generated. Although reproduction methods that are based on the use of two parents are more nature-inspired, some studies suggest that more than two parents create better solutions.

Successively repeated crossover and mutation for a selected group of parents leads to a completely new population of individuals. The effectiveness of this process is measured using a fitness function that can measure both the average fitness of the entire population and, for example, the fitness of the best individual in the population. In general, the mean fitness increases in subsequent populations because only the best individuals of the previous generation are selected for reproduction, together with a small proportion of the worse individuals. These inferior solutions ensure genetic diversity within the parents' genetic pool and thus ensure genetic diversity for the next generation. It is worth tuning parameters such as the probability of mutation, the probability of crossover and population size to find reasonable settings for the specific problem for which the experiment is being designed.

Termination condition

The process of creating new generations is repeated until the termination condition is reached. Examples of termination conditions are:

- A solution has been found that meets the minimum criteria
- Fixed number of generations generated
- The time limit for looking for better solutions has been exceeded
- Subsequent iterations no longer yield significantly better results
- Combinations of the above criteria

Implementation (10 points)

Please consider the description of each functionality below as an outline of the final solution. Modifications in the form of renaming, parameter passing and their names and types returned by the functions are acceptable. It is crucial to maintain the correct functionality of individual modules.

1. Implementation of the task generator for the knapsack problem (0.5 points)

The task generator should be implemented as a function:

```
generate(n, w, s, output_file)
```

This function takes the following parameters:

- n - number of objects to choose (int)
- w - maximum carrying capacity of the knapsack (int)
- s - maximum knapsack size (int)
- output_file - name of the file into which the task is to be saved

The generator for a given number of items is to randomly generate weights and sizes of items. The weight w_i, size s_i and price c_i of the i-th item must meet the following criteria:

- 1 < w_i < 10*w/n
- 1 < s_i < 10*s/n
- 1 < c_i < n

In addition, a set of items must meet the following criteria:

$$\sum_{i=1}^{n} w_i > 2w, \quad \sum_{i=1}^{n} s_i > 2s$$

The generator thus prepared should generate a solution for a random n, w, s from the following ranges:

- 1000 < n < 2000
- 10000 < w < 20000
- 10000 < s < 20000

The file with the solution should be saved in CSV format. The first line of the file contains numbers n, w, s (separated by a comma). The next lines represent objects. Line i contains the following numbers (separated by a comma): w_i, s_i, c_i. The generated file should be attached to the solution package and sent to the e-portal.

2. Implementation of task loading (0.5 points)

The method of loading a task as a function should be implemented:

```
read(input_file)
```

This function takes a file name in CSV format on the input. The first line of the file contains numbers n, w, s (separated by a comma). The next lines represent objects. Line i contains the following numbers (separated by a comma): w_i, s_i, c_i.

Suggestion: At the output the method can return a Task class object, in which a structure will be defined to store the elements of the knapsack problem.

3. Implementation of the creation of a random initial population (0.5 points)

A method should be implemented to create a random initial population for the knapsack problem as a function:

```
init_population(n_items, size)
```

This function takes the following parameters:

- n_items - number of items from which the subset to be packed into the knapsack is selected
- size - population size

Suggestion: The function can return a Population class object as an output. This class may contain a data structure for storing Individual class elements.

4. Implementation of the fitness function (0.5 points)

A method of assessing the fitness of an individual as a function should be implemented:

```
evaluate(item, task)
```

Suggestion: The function takes the individual as defined in the first part of the task (e.g. `Task` object). This function can also be defined as an `Individual` class function.

The output of the function is the fitness value for the individual. The value of the matching function for a given solution is the sum of the values of all objects in the knapsack if the constraints are met, or 0 otherwise.

5. Implementation of the tournament selection method (0.5 points)

The tournament selection method should be implemented as a function:

```
tournament (population, tournament_size)
```

The function takes the following parameters:

- population - a Population class object
- tournament_size - a size of the tournament

A simple tournament selection can be made as follows:

- `Select k (tournament_size) of individuals at random from the population`
- `Return the best individual in the tournament`

The best individual is the individual with the highest value of the `evaluate` function.

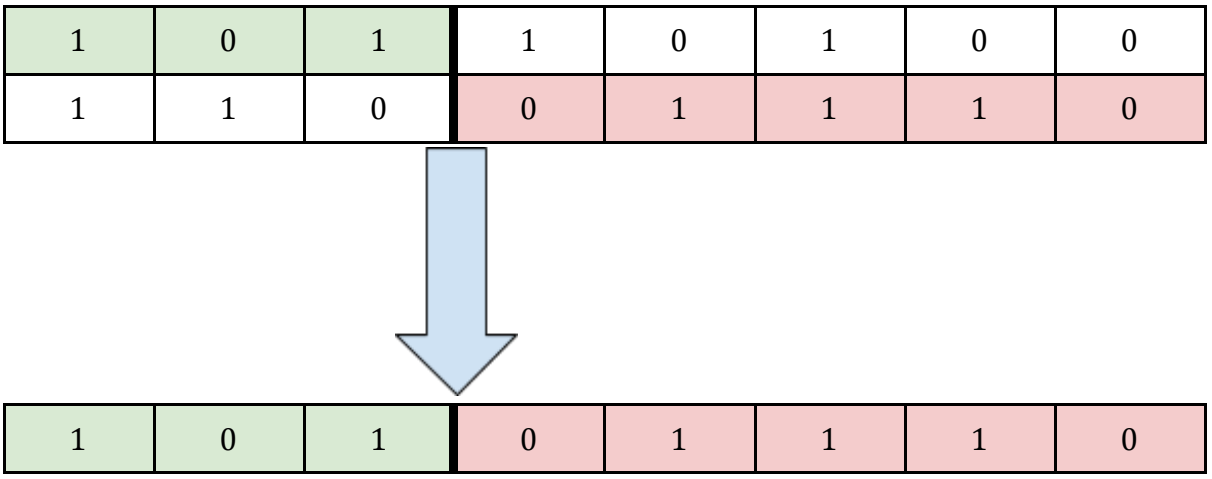
6. Implementation of the crossover operator (0.5 points)

The crossover method should be implemented as a function:

```
crossover(parent1, parent2, crossover_rate)
```

A simple crossover method involves selecting a cutting point for the parents' chromosomes. The initial fragments of the chromosomes are then swapped in places. This produces a child of a pair of parent1 and parent2.

Crossover illustration:



The `crossover_rate` parameter is the probability of a crossover occurring. Before the crossover itself, it is necessary to simulate whether the crossover is to take place. For this purpose, the real number should be drawn from the range [0, 1]. If `rand < crossover_rate`, then crossover occurs and the child is returned. Otherwise, `parent1` is returned.

7. Implementation of the mutation operator (0.5 points)

The mutation method should be implemented as a function:

```
mutate(individual, mutation_rate)
```

A simple mutation method computes how many genes will be mutated on the basis of the size of an individual n (number of genes - in the case of the problem the number of all available items). In case of n=1000 and mutation_rate=0.01 the number of genes that are subject to mutation is: 1000*0.01=10. Therefore 10 positions in the individual chromosome should be randomly selected and for each position the gene value should be changed to 0 if the current value is 1, or 1 if the current value is 0.

8. Implementation of the genetic algorithm (1.5 p.)

The genetic algorithm should be implemented using previously created elements. The pseudocode to the whole looks like this:

```
task = read(input_file)
pop = init_population(task.n_items, POP_SIZE)
i = 0
while i < ITERATIONS:
     j = 0
     new_pop = Population()
```

```
        while j < POP_SIZE:
            parent1 = tournament(pop)
            parent2 = tournament(pop)
            child = crossover(parent1, parent2, CROSSOVER_RATE)
            mutate(child, MUTATION_RATE)
            new_pop.add(child)
            j += 1
        pop = new_pop
        i += 1
    return pop.best()
```

The algorithm returns the best individual in the last population as a result.

9. Analysis of the impact of the crossover probability (1 p.)

The impact of the crossover probability on the results should be investigated. For this purpose, a minimum of 3 different crossover probability values must be selected and the test for each value must be carried out at least 5 times. The averaged values of the best individuals in subsequent generations should be shown on a chart. The conclusions of the study should be described in the report.

10. Analysis of the impact of the mutation probability (1 p.)

The impact of the mutation probability on the results should be investigated. For this purpose, a minimum of 3 different mutation probability values must be selected and the test for each value must be carried out at least 5 times. The averaged values of the best individuals in subsequent generations should be shown on a chart. The conclusions of the study should be described in the report.

11. Analysis of the impact of tournament size (1 p.)

The impact of the tournament size on the results should be investigated. For this purpose, a minimum of 3 different tournament size values must be selected and the test for each value must be carried out at least 5 times. The averaged values of the best individuals in subsequent generations should be shown on a chart. The conclusions of the study should be described in the report.

12. Analysis of the impact of population size (1 p.)

The impact of the population size on the results should be investigated. For this purpose, a minimum of 3 different population size values must be selected and the test for each value must be carried out at least 5 times. The averaged values of the best individuals in subsequent generations should be shown on a chart. The conclusions of the study should be described in the report.

13. Comparison of the best solution with any of the non-evolutionary methods (1 p.)

Compare the genetic algorithm (quality of the output, execution time) using the best set of parameters obtained from the previous evaluations with any of the non-evolutionary methods (but not a random method).

Bibliography

- Koza, J. R., & Koza, J. R. (1992). Genetic programming: on the programming of computers by means of natural selection (Vol. 1). MIT press.
- Davis, L. (1991). Handbook of genetic algorithms.
- Goldberg, D. E. (2006). Genetic algorithms. Pearson Education India.
- Gen, M., & Lin, L. (2007). Genetic algorithms. Wiley Encyclopedia of Computer Science and Engineering, 1-15.
- Mitchell, M. (1998). An introduction to genetic algorithms. MIT press.
- Goldberg, D. E., & Holland, J. H. (1988). Genetic algorithms and machine learning.