Introduction to Artificial Intelligence

Bachelor's in informatics Engineer and Informatics Engineer – European course

2º Year – 1º semester

Practical classes

Assignment 8: Evolutionary Algorithm for the Knapsack Problem

1. The Problem: Knapsack Problem

The knapsack problem is a problem in combinatorial optimization. Consider a set of objects, each with a weight and a value, and a knapsack with a maximum weight capacity. The goal is to choose a set of objects to include in the knapsack so that the total weight is less than or equal to its capacity and the total value is as large as possible. More formally:

- There is a set I with N objects, each one characterized with a weight W(i) and a value V(i), i=1, ..., N.
- There is a knapsack with a maximum weight capacity **C**.

The optimization goal is to find a set S $\subseteq$ I that respects the following restrictions:

1. The maximum capacity is not exceeded:

$$\sum_{obj_i \in S} W(obj_i) \leq C \tag{1}$$

2. The sum of the values of the chosen objects must be maximized:

$$\text{Maximize} \sum_{obj_i \in S} V(obj_i) \tag{2}$$

2. Evolutionary Algorithm

The goal of this assignment is to explore, understand and test an evolutionary algorithm to solve this problem.

2.1 Representation

To represent an instance of N objects will be used a binary array with size **N**. A position **k** in the array with a "1" indicates that the k^{th} object was chosen to include in the knapsack. If the position **k** has a "0" value, that object was not chosen.

Example: N = 6 objects. This solution – 101100 – indicates that the objects 1, 3 e 4 are in the knapsack

This representation allows the presence of **invalid solutions**, once the limited capacity of the bag can be exceeded. The evaluation function will deal with this situation, penalizing the invalid solutions.

2.2 Initial population and stop criterion

The initial population is created at random. The evolutionary algorithm stops when a fixed number of generations is attained.

2.3 Evaluation and optimization

The quality of a solution S is related to the total value of the objects chosen. Moreover, the evaluation function must distinguish in some way the invalid solutions, penalizing these solutions.

In the code given in Moodle, the evaluation function gives a quality equal to zero to all the invalid solutions. The valid ones are evaluated summing the value of the objects included in the bag. So, the quality of a solution S is given by the expression (see **eval_individual** function, at **function.c** file):

$$Fitness(S) = \begin{cases} \sum_{i=1}^{N} S[i] \times V[Obj_i] & \text{if } \sum_{i=1}^{N} S[i] \times W[Obj_i] \leq C \\ 0 & \text{otherwise (invalid solution)} \end{cases} \quad (3)$$

S[i] represents the binary value that is at the i^{th} position. The goal is to find a solution that maximizes the profit (maximization problem). **V[Obj$_i$]** is the value of each object, **W[Obj$_i$]** is the weight of the object. **C** is the maximum capacity of the knapsack.

2.4 Selection

In the code given in Moodle, the Evolutionary Algorithm uses a tournament selection method with size two (binary tournament). Each parent is selected using the following steps:

- Randomly select tow individuals of the population
- Compare their qualities
- The individual with higher fitness is selected to be a parent.

These 3 steps are repeated a number of times equal to the population size (see **tournament** function, at **algorithm.c** file).

2.5 Genetic operators

In the code given in Moodle, the Evolutionary Algorithm uses two well known genetic operators, suitable for binary representations:

- One-point crossover
- Binary mutation

Each operator is applied with a certain probability. See **genetic_operators** function, at **algorithm.c** file.

3. Implementation Details

Instance files (text files given in Moodle):

Each instance file contains:

- The values to use in the EA's parameters;
- Information about the instance to optimize.

The values in the instance file are in the following order:

- Population size
- Mutation rate
- Crossover rate

- Tournament size
- Number of generations
- Number of objects of the instance
- Maximum capacity of the knapsack
- Information about the *Weight* and the *Profit* of each object:
    - Title Line "*Weight* *Profit*"
    - N Lines (N = number of objects): Each line has two values, the **weight** and the **value** of the i^{th} object.

An example of an instance file with 10 objects is:

```
pop: 20
pm: 0.01
pr: 0.7
tsize: 2
max_gen: 100
obj: 10
cap: 25

Weight      Profit
2           6
5           1
10          10
9           14
3           5
6           6
8           9
...
```

At the beginning of the execution, the program reads an instance file and stores in the memory the parameters and the information of the problem (see function **init_data** in **utils.c**). In Moodle there are 4 instance files with different dimensions: knap_100.txt, knap_200.txt, knap_500.txt, knap_1000.txt.

EA's parameters

The EA's parameters are stored in the variable `EA_param` that is a `struct info`. The fields of this struct are:

- `popsize`: Population size
- `pr`: recombination rate to apply to each pair of parents
- `pm`: mutation rate to apply to each gene
- `tsize`: tournament size
- `ro`: constant to use in penalization
- `numGenes`: number of objects (size of the binary solution)
- `capacity`: capacity of the knapsack
- `numGenerations`: number of generations

The variable `EA_param` in declared in function `main()`. The struct is initialized using the function `init_data()`.

The constant MAX_OBJ (in `algoritmo.h`) specifies the maximum number of objects that can exists in the instance files.

Individuals
Each individual of the population is stored in a struct variable of `struct individual` (or `chrom`, with `pchrom` a pointer to a struct of this kind). This struct has three fields:

- `p`: binary vector representing the solution
- `fitness`: fitness of the solution.
- `valid`: variable that indicates if the solution is valid (1) or invalid (0).

Code:
The project given n Moodle is programmed in C and contains an evolutionary algorithm fully functional to solve the knapsack problem as described before. The project is divided in 4 modules:

- *main.c*: contains the function that controls the global functioning of the program. The *main* function executes calls to other functions:
  - Preparing the setup of the algorithm:
    - Reads the file and initialize the variables
    - Creates and evaluates the initial population;
  - Main cycle (based on the number of generations) of the Evolutionary Algorithm;
  - Writes the obtained results in the screen.
- *function.c*: contains the functions that evaluate the solutions and assess if the solutions are valid or not;
- *algorithm.c*: contains the functions used in the evolutionary algorithm: selection and genetic operators;
- *utils.c*: contains some auxiliary functions, such as random generation of number, creation of the initial population and the reading ot the instance files.

Experiments
An Evolutionary Algorithm is a probabilistic method, so to evaluate the quality of the results, it is necessary to repeat the execution several times and analyze the mean values obtained. At least 10 repetitions must be performed (this value is used in the given code), but 30 repetitions are the best option.
You need to register two important measures used to evaluate and compare the differences in the algorithm:

- Best solution found in all the repetitions
- *Mean best fitness*: average of all the results obtained in the repetitions
- Number of invalid individuals present in the population in the final population.

4. Tasks to perform

4.1 Understand the code
Carefully analyze the given code and understand the goal of each function.

4.2 Perform experiments and analyze the obtained results
Run the algorithm given in Moodle and register the obtained values in the four instance files. The goal is to understand the influence of the parameters of the EA.
Use the Excel file **Results.xls** which has 4 tabs, one for each file.

Fill the columns C and D with the obtained values using the code given in Moodle. At this point you only need to change the parameter values in the instance files, run the code and register the values. After that analyze the influence of the parameters in the obtained results. The conclusions must be detailed in the written report.

4.3 Change the evaluation function (add a penalty)

Implement a new function like **eval_individual**. This new function – **eval_individual_penalty** – must differentiate the illegal solutions using the extra weight that is in the bag.

a) The penalty associated with invalid solutions should vary according to the violation of the restrictions. Implement the following proposal:

$$Fitness(S) = \sum_{i=1}^{N} S[i] \times V[Obj_i] - Pen(S)$$

The penalty is calculated as follows (linear penalty):

i) If S is a legal solution: $Pen(S) = 0$

ii) If S is an illegal solution: $Pen(S) = \rho \times \left(\left(\sum_{i=1}^{N} S[i] \times W[i] \right) - C \right)$

The ρ parameter is equal to: $\rho = \max_{i=1,\ldots,N} \frac{V[i]}{W[i]}$.

This parameter (ro) is available in the structure that contains the algorithm.

After completing the implementation, call this new function in the **evaluate** function.

Run the experiments and fill the columns E and F with the obtained values using this penalty strategy. Repeat the experiments changing the parameters as you made before, run the code and register the values. After that analyze the results and draw some conclusions.

Analyze the variation in the percentage of illegal individuals by changing the evaluation function.

4.4 Repairing methods

Implement a repair method. The repair algorithm must receive an infeasible solution and must repair it: while the solution exceeds the capacity of the bag, remove an item. Try two approaches:

- Remove a random object until the capacity is ok
- Remove the objects with low value until the capacity is ok

The repair algorithm must be called in the **evaluate** function, before the ***eval_individual*** function (function.c).

Run the experiments and fill the columns G to J with the obtained values using these two repairing methods. Repeat the experiments changing the parameters as you made before, run the code and register the values. After that analyze the results and draw some conclusions.

4.5 Implement a new mutation operator

Implement the swap mutation operator. This operator randomly chooses 2 objects (one in the backpack and other outside the bag) and saps them. Run the experiments and fill the columns K and L with the obtained values using this new operator. Repeat the experiments changing the parameters as you made before, run the code and register the values. After that analyze the results and draw some conclusions.

4.6 Implement a new crossover operator

Implement the two-points crossover operator. Run the experiments and fill the columns M and N with the obtained values using this new operator. Repeat the experiments changing the

parameters as you made before, run the code and register the values. After that analyze the results and draw some conclusions.

4.7 Change the selection method

Tournament of variable size. Adapt the tournament selection method in order to make the size of the tournament variable. Run the experiments with tournaments of size 4 and 6 and fill the columns O to R with the obtained values. Repeat the experiments changing the parameters as you made before, run the code and register the values. After that analyze the results and draw some conclusions.

4.8 Hybrid approaches

Implement the hill climb algorithm for this problem

Explore hybrid approaches combining the local search with the EA:

1. Use the hill climbing to improve the initial population;
2. Use the hill climbing to improve the final population;
3. Use the hill climbing to improve some of the solutions of population at each iteration of the algorithm.

Run the experiments with each one of the hybrid approaches and fill the columns S to X with the obtained values. Repeat the experiments changing the parameters as you made before, run the code and register the values. After that analyze the results and draw some conclusions.