COMPUTATIONAL INTELLIGENCE IN VIDEOGAMES

AND VIRTUAL REALITY

MODULE 1 – PRACTICAL PART 1: Our first Evolutionary Algorithm

In this practice the students will implement their first Evolutionary Algorithm in Java. They will first revise an example code of a very basic Genetic Algorithm and later they will implement their own.

1. EXAMPLE OF BASIC GA

This Genetic Algorithm has been implemented for solving the OneMax problem (*maximize the number of '1' in a vector of Boolean/Binary values*).

It has the following features:

- Selection: *4-Tournament*
- Crosssover: *Uniform (2 parents => 2 childs)*
- Mutation: *Bitflip*
- Replacement: *Steady State (Elitism)*

There are two classes:

- `Individual.java` → this class defines an individual of the population. It has a *genotype* which is the vector of binary values, and implements the random *initialization* of the vector, and the *fitness function* which evaluates it.

- `GA.java` → this class implements the GA loop and all the operators. It has all the configuration parameters of the algorithm. The *population* is a list of `Individual`.
  - The *selection mechanism* is a 4-Tournament: 4 parents are selected randomly and they compete 2 by 2. The 2 winners will reproduce.
  - The *crossover operator* is a random/uniform interchange of bits between parents. There is also implemented a 1-point crossover.
  - The *mutation operator* flips a random bit.
  - The replacement is done for a part of the population, the rest (the best) remain.

➢ Run the GA:
- Eclipse or Netbeans:
  - Add the `.java` files to a project (as source files).
  - Configure the parameters of the GA.
  - Build the project (compile and generate the `.class` files).
  - Run the algorithm
- Terminal:
  - Go to folder where the `.java` files are located.
  - Configure the parameters of the GA.
  - Compile them: `javac GA.java Individual.java`
  - Run the algorithm: `java GA`

➢ Check different configurations for the algorithm and analyse the results:
- Set a higher difficulty for the problem (SIZE_INDIVIDUAL=500, 1000, 5000, 10000)
- What does it happen when the number of individuals in the population grows?
- Are more generations needed?
- How does the crossover probability affect the results?
- And the mutation probability?

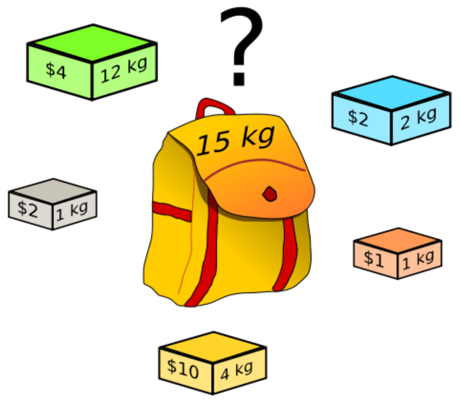
- Change the model to a *Generational* one (the whole population must be substituted) + *Elitism of 1* (the best must remain).

- Change the code in order to transform the algorithm for solving the OneMin problem (*minimum number of '1'*).

- Add a *Local Search* method:
  - Repeat a few times (10):
    - Select an individual (could be a random one or a generated child (after mutation).
    - Randomly change one gene of the individual (flipping it).
    - If the new individual is better than the original it will substitute it in the population.

2. ADAPT THE GENETIC ALGORITHM

Now the student must re-implement the algorithm for solving the ***Knapsack problem***:



Wikipedia:

It is a problem of combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem features are:

MAX 10 ITEMS EQUAL in the Knapsack
MAX_WEIGHT=4200g
1 - weight=150 value=20
2 - weight=325 value=40
3 - weight=600 value=50
4 - weight=805 value=36
5 - weight=430 value=25
6 - weight=1200 value=64
7 - weight=770 value=54
8 - weight=60 value=18
9 - weight=930 value=46
10 - weight=353 value=28

TIPS: The student must think about (and change):

- The representation: binary or integer array?
- The fitness implementation.
- The MAX_ITEMS_EQUAL and MAX_WEIGHT constraints when new individuals are created (initialization, crossover, mutation).
- Or just punish them in their evaluation (assign a very bad fitness).