# INTRODUCTION TO ARTIFICIAL INTELLIGENCE

TASK 1: GENETIC ALGORITHM

Misrraim Suárez Pérez - misrraimsp@gmail.com

# 1. Genetic Algorithm Model

(a) Individual:

- A population is modelled as a fixed size set of Individuals.
- An Individual is modelled as a sequence of Arcs. More precisely, as a sequence of Arc identifiers.
- Each Arc is modelled as a union of 4 fields, namely, node1, node2, weight and a unique identifier.
- An Individual is only permitted to be formed by a maximum number N of Arcs. Such maximum equals the number of Arcs in the ProblemGraph. So that, an Individual presents the form:

$$Individual \equiv \{id_1\, id_2 \cdots id_n\} \qquad 1 \leqslant n \leqslant N$$

Note that no restriction is made about repeated id's. That is, one could have several Arc id's repeated (see for instance Bottom Individual).

(b) Fitness Function:

In this particular problem, the goal is to minimize the weight. It has been designed a fitness function in such a way that minimization of weight implies maximization of fitness score. Also, the fitness function takes into account if a particular individual is or is not a valid solution of the problem.
The idea is to take as a 0-scored individual the worst possible one, called Bottom Individual. This Individual must have the worst possible weight, and so that, the worst possible fitness score, zero. Then, another Individual's fitness is computed as a difference with the worst one. If an Individual is not a valid solution, then the fitness computation introduces a Punish Factor as a value that reduce considerably the fitness.
The following formulas defines the computation:

$$Bottom\,Weight = Number\,of\ ProblemGraph\,Arcs * Worst\ Arc\,Weight$$

$$Individual\,Weight = \sum_{i=1}^{n} ArcWeight_i$$

$$Fitness_{isValidSolution} = Bottom\,Weight - Individual\,Weight$$

$$Fitness_{isNotSolution} = Fitness_{isValidSolution} \div Punish\ Factor$$

As an example, lets measure the fitness of an individual with summation of weights 53:
Let

ProblemGraph has 7 Arcs.
The Arc's id 4 has the worst weight, 26.
Punish Factor equals 10.

Then

Bottom Individual would be {4,4,4,4,4,4,4} and the Bottom Weight would be 7*26 = 182
If Individual to be measure is a valid solution: **Fitness** = 182 – 53 = **129**
If Individual to be measure is NOT a valid solution: **Fitness** = (182 – 53)/10 = **12**

Note that if in the unlikely case that the individual to be measured is the Bottom Individual, then the fitness equals zero.

(c) Crossover:

The population is thought as a set of pairs. Then, with probability *cP*, each pair produces exactly two children by cross combinations of genes (Arc id's).
The crossover operator takes two Individuals as arguments. The first one give his first half genes to the child, and the second one gives his second half genes to the child. So, the combination point is half genes of each parent. Let show an example. Supose the parents:

Parent1: {0,0,**9**,**5**,**6**,**3**,**7**,**8**,**4**,**1**}
Parent2: {0,0,0,0,0,0,0,**2**,**4**,**9**}

Then, the above parents yields, with probability *cP,* the following two children:

child1 = crossover(Parent1, Parent2) = {0,0,0,0,0,**9**,**5**,**6**,**3**,**9**}
child2 = crossover(Parent2, Parent1) = {0,0,0,0,**2**,**4**,**7**,**8**,**4**,**1**}

Finally, to complete the fixed size population, if needed, an immigration phenomenon is simulated by producing the proper number of new random individuals, like in the initialization process.

(d) Mutation:

The mutation operator goes throwout the population, and for each gene in each Individual, with probability *mP*, change that gene for a randomly selected one.
For example, the Individual {0,0,**9,5,6,3,7,8,4,1**} can be mutated in last gene to become the Individual {0,0,**9,5,6,3,7,8,4,9**}

(e) Selection:

Over each generation N Individuals are selected with the widely known **Roulette Wheel** selection method. Note that better Individuals are more likely to be selected, and so that such better Individuals would be repeated in the selected population.

(f) Elitism:

A reduced set of Individuals are able to skip the selection, crossover and mutation operators by passing directly to the next generation. These Individuals are the best fitted, and such a mechanism is called elitism. The number of Individuals that could be inside the elite is tune by the user, and could be zero. Note that in such a case the elitism is switched off.

## 2. Genetic Algorithm Implementation

One of the most interesting issue about the implementation is the way Indivials are created. There are three different methods, codified as constructors of the class Individuals: randomly, by crossover or by mutation. But the point is, regardless the constructor used, in the last part of them there exist a call to the private method *sortFenotype*.

*fenotype =* ***sortFenotype****(preFenotype);*

This method is intended for speeding up the genetic algorithm in the following way. Let say that the StartNode is only inside Arc **3** and the EndNode is only inside Arc **9**. Also let the fenotype of an Individual before calling this method be {**9,5,6,3,7,8**}. So, this is not a valid solution, but maybe

there exist some permutation of these genes in such a way that the resulting fenotype would be a valid solution. This method tries to determine if such a permutation exists, and if so, return the new sorted fenotype. By using this method the range of valid solutions is expanded notably, so the algorithm becomes quite faster.

Elitism is implemented as a sorted array of Individuals. The first element is the most fitted Individual. The class Elite encapsulate all necessary logic for guaranteeing the proper array order as well as the correct operation of insertion, deletion and so on.

It is possible to obtain data about best Individual and population average fitness throwout each generation. This data is properly stored in a file (in the same folder that the input graph file), and can be used to build diagrams that present the performance of the genetic algorithm. As an example, in Figure 1 is shown a diagram built with execution parameters:

Graph = ProblemGraph
StartNode = 13
EndNode = 5
Population Size = 100
EliteSize = 5
Crossover Probability = 85%
Mutation Probability = 2%
PunishFactor = 5

The graphic representation of the ProblemGraph and the solution is made by using the GraphStream Library. As an example, Figure 2 shows the ProblemGraph with the highlighted solution for above execution.

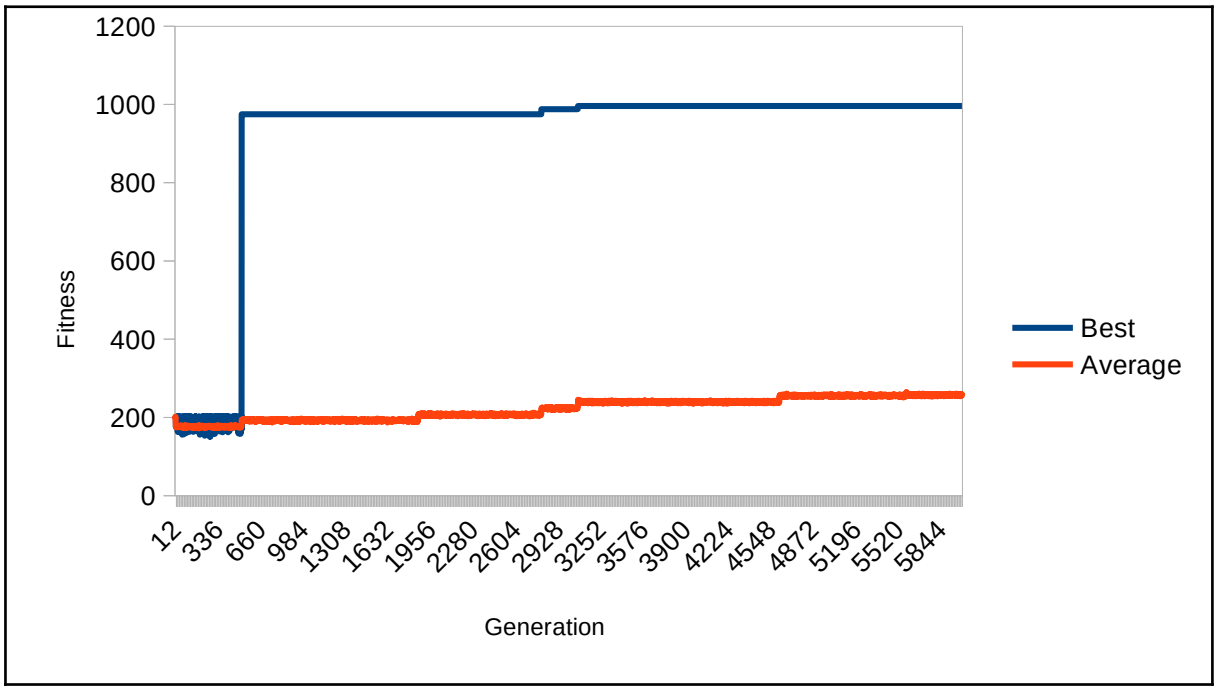


Figure 1

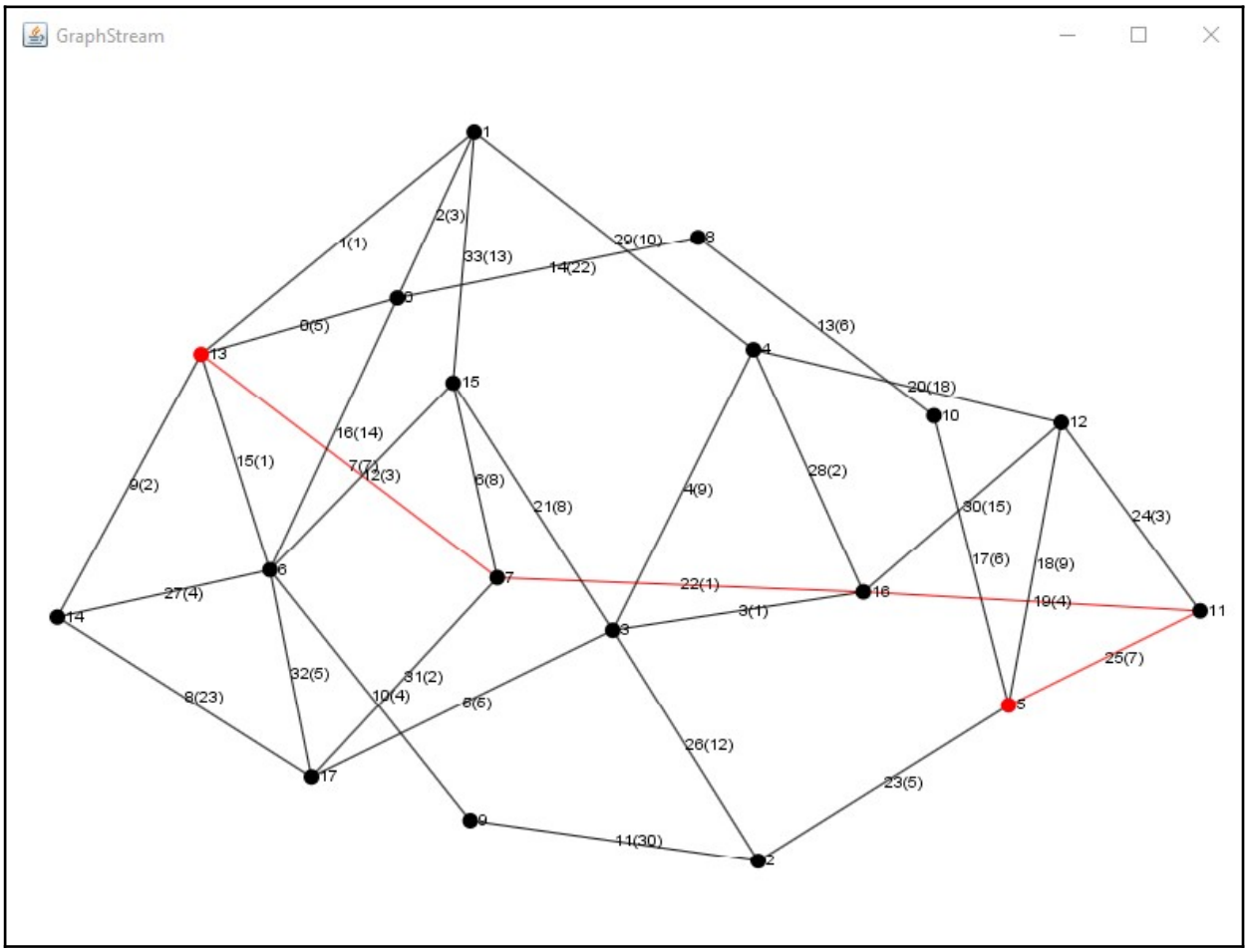


Figure 2

## 3. Parameter Testing

In this section is presented a brief study about the influence of different parameters on the performance of genetic algorithm.

The performance is measured as the number of generations needed until reaching the known best solution. For each particular value of the parameter under test the algorith has been proved automatically 100 times, saving the number of generations in a file. Then, the average number of generation is computed, as well as the average absolute deviation from such average value. The results are displayed in logaritmic diagrams. All test are performed under following fixed parameters:

Graph = ProblemGraph
StartNode = 13
EndNode = 5
TestSize = 100
EliteSize = 2
PunishFactor = 5
IterationLimit = 100000

(a) Crossover Probability

Fixed parameters: [Mutation Probability = 2%, Population Size = 100]

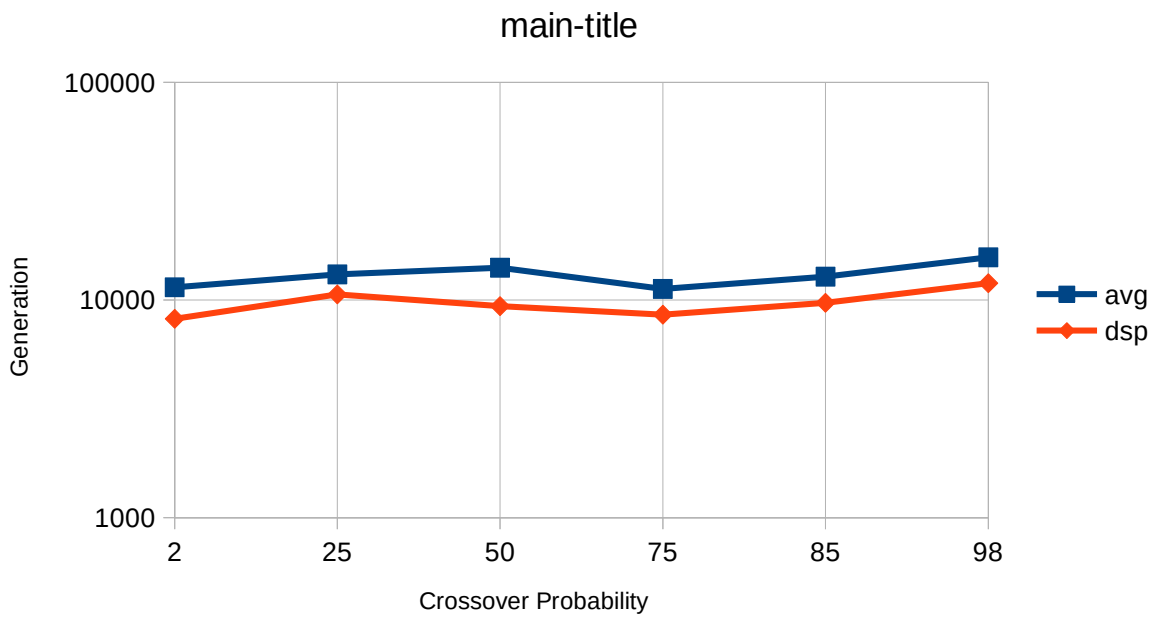


Figure 3

(b) Mutation Probability

Fixed parameters: [Crossover Probability = 2%, Population Size = 100]

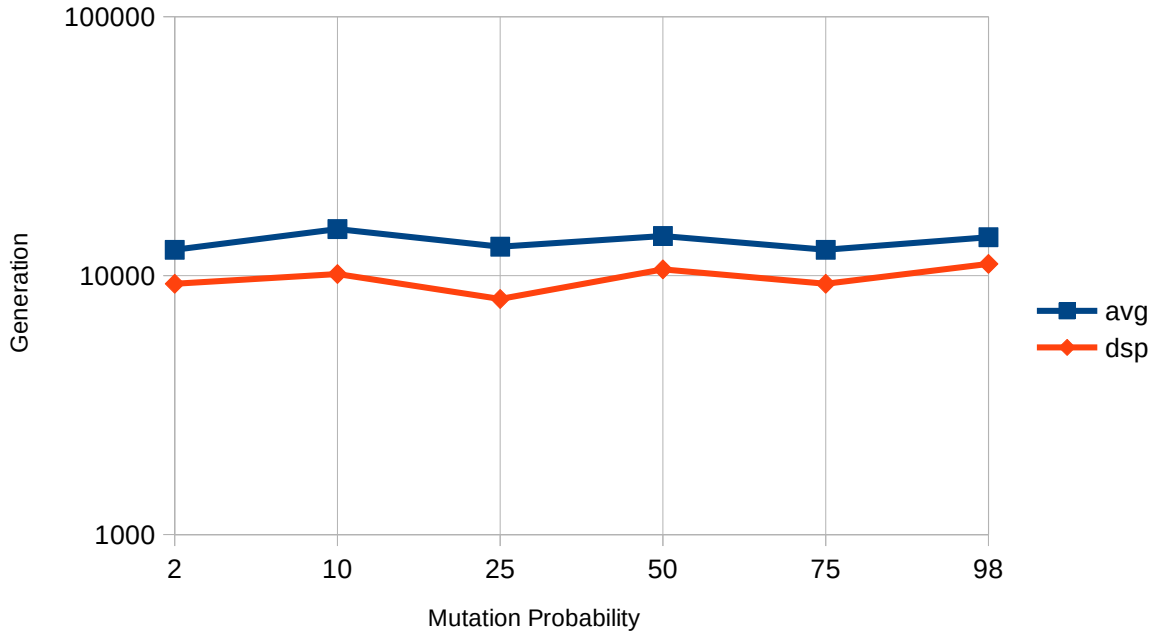


Figure 4

(c) Population Size

Fixed parameters: [Crossover Probability= 85%, Mutation Probability = 2%]

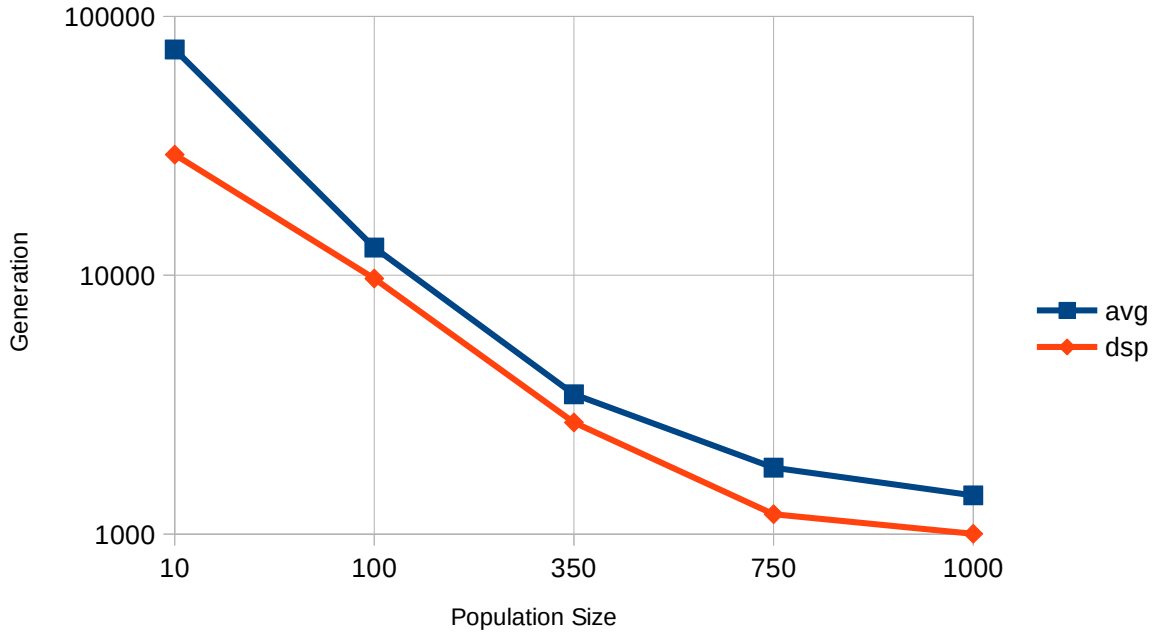


Figure 5

(d) Discussion

Several conclusions can be postulate under above diagrams inspection.

First, the performance of genetic algorithm, in the terms that has been defined, present a strongly random behavior. The average deviation in all cases has the same order that the average value itself.

Secondly, the crossover probability and the mutation probability present no practical influence in the performance of the algorithm developed, Figure 3 and Figure 4. This can be explained in terms of the fitness measure method. Such method works but it is inefficient, because is ‘black or white’. That is, an Individual is a valid solution or not, without ‘grey’ cases, and so that the Punish Factor impact all non valid solutions in the same way. This kind of mechanism leads the algorithm to behave in strongly randomly manner, because for this particular problem is hard to find a valid solution. An alternative to be develop, and without any doubt would make the algorith quite better, is to improve the fitness measure method to take into account the non valid solutions that are ‘good partial solution’ at the same time. Fortunately, the elitism mechanism balance somehow this inefficient behavior.

Thirdly, taking into account the above discussion, it is clear to see that the more Individuals in the population more probability to find the solution. This behavior can be checked in Figure 5. The generations needed to reach the solution decrease with the population size. But, this does not mean that the algorith would be faster, because each generation takes more time due to the large number of Individuals.

## 4. Non evolutionary strategy: Dijkstra algorithm

The Dijkstra algorithm has been proved with the aid of GraphStream Library. The method is such fast that no comparison worths with the genetic algorithm developed. But, for a large problem size (Dijkstra complexity is O(nlogn)), and with another, more efficient and optimized, genetic algorithm, thing could be different.

## 5. References

GraphStream Library: http://graphstream-project.org

Genetics Algoriths. [K.F. Man, K.S. Tang, S. Kwong] Springer