

Solving Travelling Salesman Problems using Genetic Algorithms

SEM6120 - Assignment 2

ALEXANDER D BROWN (ADB9)

Contents

1	Design	3
1.1	UML Class Diagram	3
1.2	Representation of Graphs	4
1.3	Use of Functional Programming Paradigms	4

1 Design

1.1 UML Class Diagram

Figure 1 shows the initial UML Class diagram for this project. There are some elements which break from typical object-orientated design, noticeably the representation of nodes in a graph as a map of integers to a tuple of float, these parts are done so to re-use internal data structures of the Python programming language to speed up the implementation of many features.

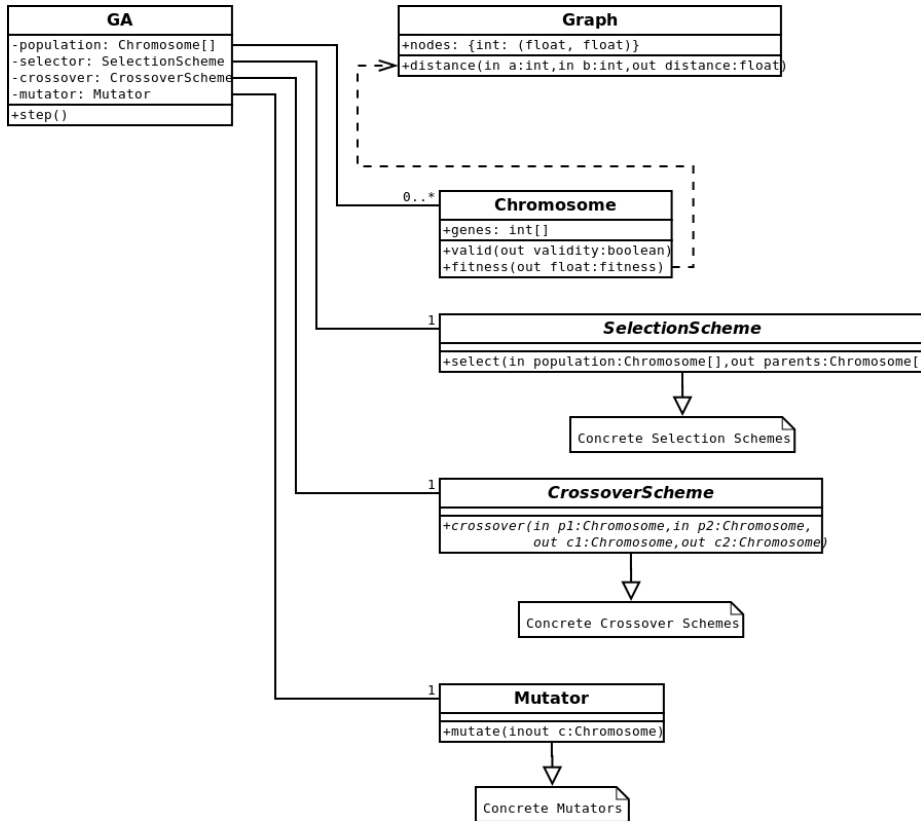


Figure 1: UML Class Diagram for the Genetic Algorithm

This design is such that factories exist to create selection schemes, crossover schemes and mutators based on an input string to facilitate the switching of these elements via command line arguments.

This design was slowly improved through the project; the **CrossoverScheme** class implemented the **crossover** method, which then called a separate method, **do_crossover**, to generate **c1** based on **do_crossover(p1, p2)** and **c2** on **do_crossover(p2, p1)** to make the processing more uniform. Subclasses were still able to override the **crossover** method, but were encouraged to implement a **do_crossover** method unless the scheme required a different behaviour of child generation.

1.2 Representation of Graphs

The actual of representation of a graph is a map of the node identifier to the x and y co-ordinates of that node (as a tuple). This allows easy look up of nodes within the map to get the position of the node. This was chosen because of the representation of the problem in chromosomes - each gene represents a node in the graph.

With this representation, the fitness function would be the distance of the tour represented by these nodes:

$$d_{tour} = \sum_{i=0}^N \begin{cases} d(n_i, n_{i+1}) & \text{if } i + n < N \\ d(n_i, n_0) & \text{else} \end{cases} \quad (1)$$

Programmatically, with the advantage of the functional and in-built elements of Python, this can be simplified to:

```
def d_tour():
    # Move the first element of the array to the end.
    shifted = nodes[1:] + nodes[:1]
    return sum([distance(i, j) for (i, j) in zip(nodes, shifted)])
```

Listing 1: Distance of a tour

1.3 Use of Functional Programming Paradigms

As Python implements several different programming paradigms a lot of problems can be solved with a different approach than other languages can. Both genetic algorithms and the travelling salesman problem lend themselves towards a more functional approach with a lot of list processing. Using Python's list comprehensions and the in-built list functions shortened the amount of code required and makes the code a lot easier to understand for those who are used to this approach.

A very good example of this is the method for evaluating chromosomes in a population, returning a list sorted from the best to the worst:

```
def eval(population):
    return sorted(map(eval_single, population),
                  key = lambda chromosome: chromosome.score)

def eval_single(chromosome):
    chromosome.score = chromosome.fitness()
    return chromosome
```

Listing 2: Using function elements to improve succinctness and readability