Solving Travelling Salesman Problems using Genetic Algorithms

SEM6120 - Assignment 2

Alexander D Brown (adb9)

Contents

1 Introduction

1.1 Genetic Algorithms

Genetic algorithms are a biologically-inspired approach to heuristic search that mimic natural selection. Unlike many other evolutionary strategies and evolutionary programming, they are not designed to solve a specific problem, but are designed to solve the problem of optimisation which is made difficult by substantial complexity and uncertainty[1].

1.2 Travelling Salesman Problem

The Travelling Salesman Problem is a well-known NP-hard optimisation problem which asks the question: *"Given a number of cities and the distances between these cities, find the shortest touch which visits each city exactly once and returns to the first city."*

Assuming each city is connected to every other city, this problem reaches a complexity of $O(n!)$ and is very resource intensive to brute force a problem with any decent number of nodes quickly become too larger problem to solve within a reasonable amount of time.

There are many heuristic algorithms which have been applied successfully to the Travelling Salesman Problem, including both Evolutionary and Genetic Algorithms.

Though the nature of Genetic Algorithms are suited to the optimisation of a Travelling Salesman Problem, normal methods of crossover and mutation cannot, generally, be applied directly to the problem. The representation of chromosomes has to be ordered (i.e. each city must appear once and only once) and additional methods of crossover and mutation have had to be designed for these ordered chromosomes.

2 Design

2.1 System Design

2.1.1 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 turple 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.

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

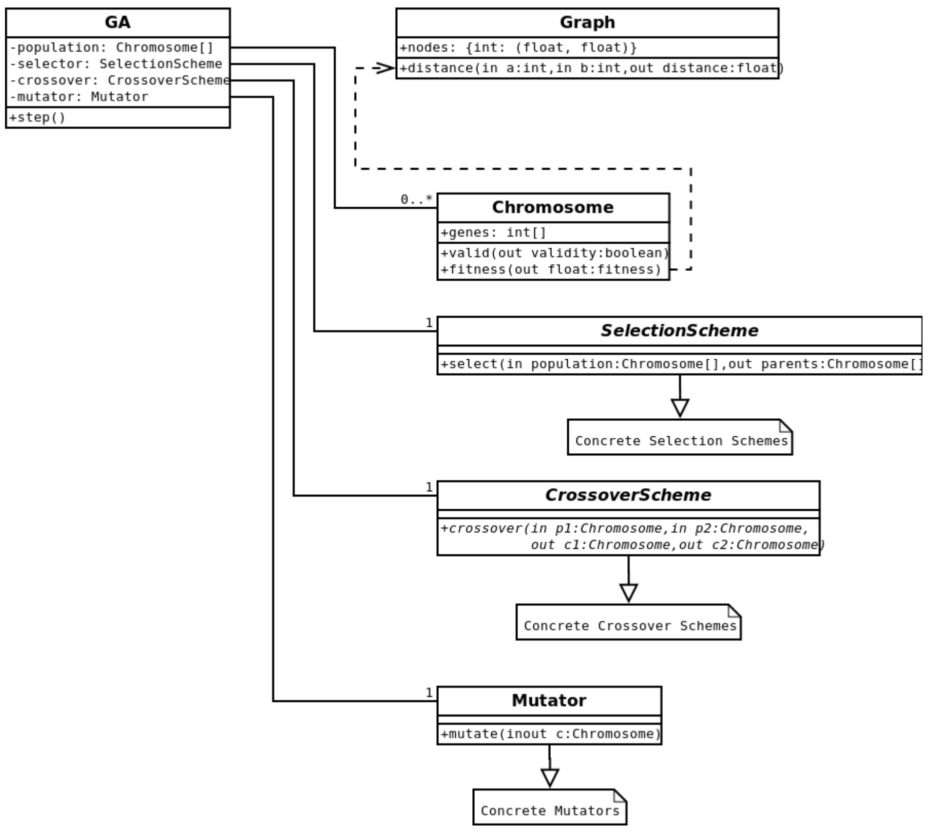
GA
-population: Chromosome[]
-selector: SelectionScheme
-crossover: CrossoverScheme
-mutator: Mutator
+step()
Graph
+nodes: {int: (float, float)}
+distance(in a:int,in b:int,out distance:float)
0..*
Chromosome
+genes: int[]
+valid(out validity:boolean)
+fitness(out float:fitness)
1
SelectionScheme
+select(in population:Chromosome[],out parents:Chromosome[])
Concrete Selection Schemes
1
CrossoverScheme
+crossover(in p1:Chromosome,in p2:Chromosome, out c1:Chromosome,out c2:Chromosome)
Concrete Crossover Schemes
1
Mutator
+mutate(inout c:Chromosome)
Concrete Mutators

Figure 1: UML Class Diagram for the Genetic Algorithm

a `do_crossover` method unless the scheme required a different behaviour of child generation.

2.1.2 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:

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

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

```python
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

2.2 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 Pythons 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 sustinctness and readability

3 Results

3.1 Best and Mean Population Fitness against Number of Generations

3.2 Best and Mean Population Fitness against Runtime

3.3 Convergence

4 Conclusions

References

[1] John H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press, April 1992.