Evolutionary Optimization

Introduction to the problem

In this document, we will work on finding a solution to the Traveling Salesperson Problem (TSP). This problem involves a person that must find the shortest route between a set of points that must be visited. Each path between two points has a value corresponding to the length of said path.

For exemple, our Salesperson as a map:

```
graph LR;
    A<-- 13 --->B;
    B<-- 30 --->C;
    C<-- 33 --->D;
    D<-- 66 --->A;
    A<-- 70 -->E;
    B<-- 55 -->E;
    C<-- 42-->E;
```

Here, the shortest path has a length of 209 units and one way to do so would be

`C -> E -> B -> A -> C`

The goal of this work is to implement an evolutionary algorithm that will be able to find a solution for any given map (i.e., a graph).

Getting started

The graph

In this experiment, we will represent our graph as a Python class. This class will contain:

- a list of every vertices in the graph
- a list of all edges
- a list of all edges' weight

The class contains some methods to handle the graph:

- get_verticies: to get all vertices of the graph
- get_vertex: get an edge by its value
- get_total_len: get the total length of th graph by summing up all weights
- get_edge_weight: retreive the weight of a given edge
- add_vertex: add a new vertex in the graph
- add_egde: add a new edge in the graph
- get edges: get all edges within the graph
- get neighbours: retreive all neighbours of a given vertex

The algorithm

For an evolutionary algorithm to be evolutionary, it needs 5 parts:

- a population;
- a way to quantify the quality of each individuals (called fitness)
- a way to select individuals
- a reproduction system
- some mutations;

The population

The population is the set of all proposed solutions of an evolutionary algorithm considered in one iteration. These solutions are also called individuals, in biological terms.

Since our goal is to find the shortest path between every point of the graph, we will take as population a set of 5 possible route. We will define a possible route as an arrangment of each and every points of the graph, no matter if a route exists between two given points.
This population might not be the best, but it will do the job at least for the beginning.

Fitness

The fitness function will be the one responsible to know of an individual is good enough to serve as the solution of the problem.

To start slowly, let us say that a individual is good enough if a complete loop visiting each points in order is possible.

Selection

The selection part may be the most important part of the evolution algorithm logic. It is this part that is in charge of picking some individual from the population, and remodeling them to create a brand new population.

For a first version of the algorithm, we will remove the less efficient route, according to the fitness function, and duplicate a random one from the 4 remaining individual. Then we will make them reproduce.

Reproduction system

The reproduction system, or crossover, is the process responsible of creating a new population from the old one.

In our case, the crossover will happen between the 5 selected individuals. Pairs of individuals will be created and fro each pair, a random number will be generated. This random number will be use to cut the path in two parts for each individual of the pair. Then, the last part of each individual will be swapped.

Mutation

The mutation is the last part of the cycle and will be executed on the newly created population. This process' purpose is to modify each individual in order to may be find a better solutions.

Without mutation, the repoduction process will eventually led to a population based on the same individuals.

The mutation in our algorithm will consists of swapping two random points in each individual's proposed solution.

Testing

- Parameters:

For the tests we will take the same graph as in the introduction:

```
graph LR;
    A<-- 13 --->B;
    B<-- 30 --->C;
    C<-- 33 --->D;
    D<-- 66 --->A;
    A<-- 70 -->E;
    B<-- 55 -->E;
    C<-- 42-->E;
```

- Steps:

To start, five individuals are selected to form the very first population

```
[['C', 'D', 'B', 'E', 'A'], ['E', 'D', 'B', 'C', 'A'],
['A', 'D', 'B', 'E', 'C'], ['E', 'B', 'A', 'C', 'D'],
['C', 'E', 'A', 'D', 'B']]
```

Then, the quality of each individuals is computed by the fitness function, which gives:

```
[20.0, 0.0, 20.0, 80.0, 60.0] # Note that each value correspond to the
percentage of fitness.
```

As none of the individuals satisfies completely the required fitness (which is 100.0), the algorithm will have to try new solutions.

After the selection, reproduction and mutation of the population, we are left with a brand new one:

```
[['C', 'D', 'E', 'B', 'C'], ['A', 'E', 'B', 'D', 'A'],
['A', 'D', 'E', 'E', 'B'], ['B', 'C', 'A', 'C', 'D'],
['A', 'D', 'E', 'E', 'B']]
```

which is evaluated by the fitness function as:

```
[20.0, 40.0, 20.0, 80.0, 20.0]
```

The quality of this population does not match our threshold. Therefore, we have to regenerate a new population.

After two new cycle, our algorithm eventually found a solution that match our expectations. Or has it ?

Let's evaluate the proposed solution:

C --> D --> A --> C --> A

If we look at our graph, we can easily make this route without going through any non existing path. But there is one major problem concerning the rules of the TSP: we can not visit a point more than once AND we have to visit every point of the graph. If we take a closer look on the proposed solution, we will rapidly find out that we visit two times points A and C and never points B and E. So the proposed solution is not a solution.

The origin of the problem comes from the fact that when we cross two individuals, the part one take from one to plug into the other may contain points that are already visited in the original part.

To fix this problem, we can either:

- modify the fitness function to check if there are no points that are visited more than once, or
- modify the cross over part to change the duplicates points to the one not visited

Let's implement both separately to see which one is the best.

Algorithm with modified fitness function:

The algorithm do not find any solutions in most of tries. The reason is that we did not fix the problem of 'visiting a point more than once' instead we just said that this kind of individual is not fit enough. So modifying the fitness function is not the solution here

Algorithm with modified crossover function

This modification allows us to get a solution every time that respects the rule of visiting every point of the graph once. In average, this algorithm finds a suitable solution after 3 generations with each generation process taking around 0.05ms. Which is quite efficient.

However, the given solution is not the optimum.

Path length

One caracteristics of the TSP, and the one that make this problem so difficult to solve is that every path between two point has a length. So eventhough we find a route that visits every point in the graph, this may be not the most efficient one, and it might exists an other solution that is faster.

To implement this, we shall change the fitness function to not only check if the edge exists but also compute the total length of the route. But we can not just compute this length and add it to the fitness score, the result would make no sense in the end. Instead, one way to do it would be to take the difference between

the length of the proposed route and the total length of the graph. The total length of the graph shall be the sum of all path length in the graph. Let's implement the new fitness function

Test

After implementing this feature, it seems the algorithm can't find any solutions. There are two explainations. First, since we change the fitness function, it is now much more strict, thus giving lower grades to each individuals. As it is now impossible to reach 100.0 fitness score, as it it impossible to have a route length of 0, we should lower the threshold of acceptance.

Then, the selection and crossover function are quite elitist. Because they keep only the most efficient individuals and duplicate one of them, the population tends to be composed of the same individuals.

Optimization

Let's start by lowering the fitness threshold. First, we shall change the way of computing the fitness of the proposed route length of each individual. Since it is impossible to hit a 0-length route, we will obviously always travel a certain distance. So we should add a 'minimum authorized length'. We can easily say that the shortest route will cover at least 70% of the graph total length.

Already, changing the fitness threshold allows us to get a solution. Moreover, it seems to find the shortest path for the example graph every time. With an average of 5 generation and around 0.07ms per generation, the efficiency of this algorithm is quite acceptable. At least for this graph. What about is an other graph ? Here is a graph with more inside connections, where the shortest path length is 96 units:

```
graph LR;
    A<-- 10 --->B;
    A<-- 15 --->C;
    A<-- 20 --->D;
    A<-- 25 --->E;
    B<-- 12 --->C;
    B<-- 27 --->D;
    B<-- 22 --->E;
    C<-- 17 --->D;
    C<-- 30 --->E;
    D<-- 35 --->E;
```

After some simulations, it seems it always finds a solution in the first generation. It does so because the shortest path length is much smaller than the total length of the graph. So, obviously it would be very likely to find a path short enough to meet our threshold of acceptance. In 100 simultations, the algorithm seems to find the shortest path only 33.0% of the time.

We could be fine with the proposed solution, given the algorithm finds it quite fast. However, the goal of the algorithm is to find the shortest path, not to find a solution among other in the shortest computation time (of course, finding the shortest path in the fastest way would be perfect).

With that being said, we must find a solution to be sure, the found solution is actually the shortest path. We could change the fitness threshold, but it will fix the problem only for this graph. What we should do, is to have a threshold in accordance of the fitness score of the generations. To do so, we can run the simultation

for a few generation, take the maximum fitness score of all of these generation, and set it as the threshold of acceptance. But how many generations have to be generated before being sure the threshold will allow us to find the shortest path 100% of the time ? Well, let's test multiple values, like 5, 10, 50, 100:

Each simultation has been ran 100 times:

```
5 generations:
average total time: 0.72ms
average number of generations: 11
average time per generation: 0.06ms
success rate: 100.0%
shortest path found: 82.0%
average path length: 96.54

10 generations:
average total time: 1.3ms
average number of generations: 22
average time per generation: 0.06ms
success rate: 100.0%
shortest path found: 90.0%
average path length: 96.3

50 generations:
average total time: 3.51ms
average number of generations: 61
average time per generation: 0.06ms
success rate: 100.0%
shortest path found: 100.0%
average path length: 96.3

100 generations:
average total time: 5.97
average number of generations: 112
average time per generation: 0.05ms
success rate: 100.0%
shortest path found: 100.0%
average path length: 96.3
```

Based on the results, we can conclude that 10 generations is the best as it allows to find the shortest path 90% of the time and does not take too much generations to find a solution.

But the efficiency of the algorithm can still be improved. As said before, the selection process is kind of elitist, which can led to a population based on the same individual.

Optimisation

As a reminder, the actual selection process remove the least efficient individual and replace it by duplicating one individual among the remaining in the popultation. This is especially this last step that is the most likely to led to a population of the same individual. To fix this, instead of duplicating an individual, we can create

one from scratch. This will increase the diversity of the population, thus allowing the algorithm to have a wider range of research.

Test

After 100 simulations, we get those results:

```
Average total time: 0.82
Average number of generations: 12
Average time per generation 0.06ms
success rate: 100.0%
Shortest path found: 100.0%
Average pathe found: 96
```

Again, each simulations waited for at least 10 generations before eventually giving a solution.

What can we say about these results ? Well, it shows that the fix we made to the algorithm are really efficient. We decreased the total computation time by 63%, the number of generations is 54% less and now, we find the shortest path 100% of the time.

So what can we say about elitism in evolutionary algorithm ? The tests results proved that strict elitism (all new individuals are from the best of the old population) reduces the efficency of the algorithm, both time and accuracy speaking. Instead a selection with a greater diversity allows to find the best solution faster with less generations, as it increase the area of search for a solution.

Larger graphs

Now that we have a fairly efficent algorithm that always finds the best solution, let's try it with a bunch of differently shaped graphs.

```
graph LR;
  A((A)) <-- 29 --> B((B))
  A <-- 20 --> C((C))
  A <-- 21 --> D((D))
  A <-- 18 --> E((E))
  A <-- 10 --> F((F))
  B <-- 15 --> C
  B <-- 28 --> D
  B <-- 17 --> E
  B <-- 14 --> F
  C <-- 23 --> D
  C <-- 11 --> E
  C <-- 8 --> F
  D <-- 27 --> E
  D <-- 25 --> F
  E <-- 12 --> F
```

This graph has 6 vertices, 15 edges and a shortest path of 95

Results:

```
Average total time: 1.47ms
Average number of generation 16
Average time per generation: 0.09ms
success rate: 100.0%
Shortest path found: 64.0%
Average shortest path: 95.88
```

```
graph TD
  A((A)) <-- 29 --> B((B))
  A <-- 20 --> C((C))
  A <-- 21 --> D((D))
  A <-- 18 --> E((E))
  A <-- 10 --> F((F))
  B <-- 15 --> C
  B <-- 28 --> D
  B <-- 17 --> E
  B <-- 14 --> F
  C <-- 23 --> D
  C <-- 11 --> E
  C <-- 8 --> F
  D <-- 27 --> E
  D <-- 25 --> F
  E <-- 12 --> F
  A <-- 30 --> G((G))
  A <-- 31 --> H((H))
  A <-- 32 --> I((I))
  B <-- 24 --> G
  B <-- 26 --> H
  B <-- 33 --> I
  C <-- 19 --> G
  C <-- 22 --> H
  C <-- 34 --> I
  D <-- 16 --> G
  D <-- 35 --> H
  D <-- 36 --> I
  E <-- 37 --> G
  E <-- 38 --> H
  E <-- 39 --> I
  F <-- 40 --> G
  F <-- 41 --> H
  F <-- 42 --> I
  G <-- 43 --> H
  G <-- 44 --> I
  H <-- 45 --> I
```

This graph contains 9 vertices, 35 edges with a shortest path of 194

Results:

```
Average total time: 35.16ms
Average number of generation 24
Average time per generation: 2.41ms
success rate: 100.0%
Shortest path found: 5.0%
Average path length: 208.94
```

From the lasts two tests, we can conclude that the algorithm is not really efficient on large graphs. The increasing time is quite logic, as the fitness function has to compare an edge in an individual's path to all eges in the graph, which in the worst case gives us a complexity of O(nm), with n the number of vertices in the graph and m the number of edges.

However the algorithm seems to struggle finding the shortest path. Since the more edges and vertices there is in the graph, the more possible paths. So we need to increase the searching range. One way to do this will be to increase the number of individuals per generation. Currently there are 5 individuals per generation. We should first test with 10 individuals per generation. We will use the latest graph used.

Results:

```
Average total time: 46.92ms
Average number of generation 54
Average time per generation: 1.81ms
success rate: 100.0%
Shortest path found: 3.0%
Average path length: 205.87
```

What about with 20 individuals per generations ?

Results:

```
Average total time: 57.44ms
Average number of generation 47
Average time per generation: 2.73ms
success rate: 100.0%
Shortest path found: 1.0% (189)
Average path length: 203.71
```

The improveness is not really significant. Maybe we should try increasing the number of generation needed before choosing a solutions. Let's try with 20 generations minimum and 10 individuals per generations.

Results:

```

```

```
Average total time: 62.18ms
Average number of generation 106
Average time per generation: 1.05ms
success rate: 97.0%
Shortest path found: 1% (190)
Average path length: 199.96
```

It seems increasing the number of generated generations or increasing the number of individuals per generations will not improve the efficiency of the algoritm, at least not enough for it to be worth the time and ressources taken.

But there is still one optimization we can try. Currently, the mutation function swaps a two random vertices in an individual. This help increase a bit the diversity of the population, but we can do much better. Instead of randomly swap verticies, we will swap vertices only if it decreases the length of the paths.

After implementing the new mutation function and testing with 5 generation minimum and 5 individuals per population, we get this results:

```
Average total time: 49.15ms
Average number of generation 6
Average time per generation: 7.82ms
success rate: 99.0%
Shortest path found: 9% (189)
Average path length: 192
```

Just out of curiosity, here are the results for the first weighted graph we used:

```
Average total time: 0.8ms
Average number of generation 6
Average time per generation: 0.12ms
success rate: 99.0%
Shortest path found: 98.0%
Average path lenght: 96.1
```

As we can see, the algorithm is twice as slow than before, but finds the best solution easier. Also when it does not find the best solution, the results are much more grouped towards the best solution than before

Space complexity

Currently, when generating permutations, the algorithm generates them all. Which means it computes and stores n! values (with n the number of vertices in the graph). So, we need to change the way permutations are done. Instead of generating all permutations possible we will generate only the number we want. This will save both running time and memory usage.

Here are the results, with the same parameters as last time:

```
Average total time: 7.43ms
Average number of generation 6
Average time per generation: 1.21ms
success rate: 99.0%
Shortest path found: 8% (189)
Average path length: 195.19
```

We have just divided the computation time by 7, which is a great increase of time efficiency

Even larger graph

Let's test our algorithm to its limits. First, we have a graph with 20 verticies and 190 edges, with the shortest path found being 187.

```
Average total time: 310.06ms
Average number of generation 6
Average time per generation: 51.17ms
success rate: 100.0%
Shortest path found: 1.0% (187)
Average path length: 223.19
```

Here, with a graph that has 50 vertices, 1225 edges and a shortest path of 486.

```
Average total time: 16551.0
Average number of generation 6
Average time per generation: 2758.5ms
success rate: 100.0%
Shortest path found: 1.0% (486)
Average shortest path: 573.35
```

Those tests were executed with at least 5 generations and 10 individuals in each population. Let's now test with 10 generations minimum.

With the graph with 20 vertices, it found a shortest path of 181:

```
Average total time: 621.43ms
Average number of generation 11
Average time per generation: 56.38ms
success rate: 100.0%
Shortest path found: 1.0% (181)
Average path length: 219.47
```

And there are the results for the graph with 50 vertices:

```
Average total time: 32916.96ms
Average number of generation 11
Average time per generation: 2992.45ms
success rate: 100.0%
Shortest path found: 1.0% (487)
Average shortest path: 573.02
```

We can see that the bigger the graph the less affected the efficency by the number of generation minimum. Although, the time taken per generation increased insignificantly, so it would still be worth to wait for more generations before giving a result, as demonstrated further up in this report.

Conlusion

During this experiment, we saw multiple ways of implementing and optimizing an evolutioonary algorithm. We can take two important factors which are elitism and diversity. We saw that two much elitism, that is taking the best individuals and work only with them, might and will lead to a population composed of the same individual, which sorely reduces our chances to find the best solution, at least in a reasonable time. This is correlated with diversity. Diversity helps increase our range of search, which helps finding different solutions and thus the best one. But to much diversity may increase the computational time so much that the improvement are not worth it any more.

So, to have an evolutionary algorithms that keeps some of the best individuals from a quite large range of search to find the best solution in a reasonable time, we must find the perfect balance between elitism and diversity.

Lastly, we demonstrated that the size of graph affect the efficiency of the algorithm. Obviously it would take more time before eventually finding a solution. But most of all, it makes the algorithm struggle more to find the best solutions every time, since the larger the graph is, the more permutation it exists, so the less likely it is to find the best solution.