Coursework 3 - The Travelling Salesman Problem Report

by Alvaro Martin – s2023748

Algorithm: Genetic TSP

Algorithm description

Having implemented a heuristic algorithm and a greedy algorithm for this problem, I wanted to investigate other ways of solving this problem. The genetic algorithm caught my eye, as it somewhat simulated evolution in the context of solving problems, which I thought was unique and interesting. The main contribution for the ideas for implementation and understanding of the genetic algorithm is taken from Eric Stoltz (2016).

Before implementing the algorithm in this context, I had to research the concept of genetic algorithms, to understand how an implementation would make sense in this context. I decided to follow some of the ideas from Ahmed Gad's (2018) article on Genetic algorithm implementation to structure my algorithm.

Essentially, these algorithms start with a population, assign a fitness level to the members of the population, breed the population and generation after generation "improve" the genes to create a better population. This evolution happens by preserving a portion of the best members (the ones with the best fitness scores) from generation to generation and applying some mutations to prevent the population from becoming equal members.

In the context of this problem, a population of random routes is created at the start, and a fitness level to each route is calculated. A mating pool is created with some randomness, but always having some of the members with the best fitness. For breeding, the members with the best fitness from the previous generation get passed on to the next generation straight away. The other members of the population are created based on a breeding function that switches part of the route from randomly chosen parents of the population. The new population is then mutated by switching some of the nodes within the members themselves, to avoid repetitions within the population.

The structure is as follows:

Population -> Fitness calculation -> Creation of mating pool -> Breed population -> Mutate Population -> Repeat for all generations

Population:

Given a population size n, the algorithm creates a random population of n routes

Fitness calculation:

For the n routes, the fitness calculation is done by doing 1/(route distance). This is so the members with the smallest distance have the best fitness. The population is stored in an array of tuples, each tuple including a list of the route, the route distance, and the fitness of the route.

Creation of mating pool:

A percentage k is specified for the percent of the best members that will get passed on to the next generation straight away. Those members enter the mating pool straight away, and the rest of the members are chosen randomly from the population.

Breed population:

The k percentage from the population is passed on as children for the next generation, and the rest of the parents are randomly chosen between the population. The breed function swaps a section of the route of one parent (of random length, the gene) with the other parent, to create the child.

Mutate population:

Given a probability m, each of the members of the new population is mutated with that percentage. If the mutation happens, two of the nodes of the member are swapped.

Repeat for all generations:

Given a number of generations g, repeat this process g times and store the best global route to return after program ends.

Proof of polynomial time:

```
def Genetic(self, numOfGenerations, initialPopSize, bestFromPopPercent, mutationRate):
```

To prove for polynomial time, I will define the following variables:

n: Number of nodes in a route

g: Number of generations

p: Number of routes in a population

The main loop of the program runs g times, giving a runtime of O(g). To calculate the fitness of all the routes, the program loops through all the population p, and to work out the fitness of each member, the program must loop through the n nodes to find the distance. This means that fitness is worked out in O(np) time, but this is done g times, giving O(g(np)). Creating mating pools is done by looping through the population, so O(p). Overall, O(g(np + p)). Breed mating pool loops through the population p, but the function breed loops through the nodes of the members to create the child. This gives runtime of O(np), which implies overall O(g(2np + p) = O(g(np+p)). Mutate population also loops through the population in time p, so the final runtime would be O(g(np+2p)) = O(g(np+p)).

Experiments

Number of nodes vs Algorithm performance

Implemented a method to generate files within a range of number of nodes and return a list of the filenames. I then created graph objects for each of the files and performed all the implementations of TSP algorithms to see the final tour value for each algorithm, and how this varied depending on the number of nodes of the graph.

After many runs of the test with different parameters for number of nodes, and running all the algorithms with different parameters, several conclusions can be drawn from this:

Greedy solution is overall always good but has a problem because it must return to the first node and does not consider the distance from the last node to the first node in the implementation. It therefore is limited and may never get to an optimal solution for a given graph.

Swap heuristic gets stuck when it finds a decent solution, as it performs certain swaps that do not lead to a better solution. This means that despite giving it a large parameter for k, it will reach a point where the algorithm will not improve the tour value.

A similar issue arises with Two-opt heuristics, but the testing suggests that the algorithm performs better than both greedy and swap heuristic for most routes. It however has the issue so getting stuck after a certain point, and despite increasing the k parameter, the algorithm does not improve.

These three algorithms in general are good for graphs with a small number of nodes, but do not perform as good when a large number of nodes is given. The genetic algorithm, however, has more potential when a substantial number of nodes are provided. Since the algorithm includes randomness in the mutation part, the algorithm will most likely find ways to improve tours at certain points. It was interesting to investigate this algorithm for a large number of nodes, because passing many generations as a parameter gave decent results for tour values, leading to even better values than the other three if a very large number of generations was specified.

Furthermore, tests on the parameters of the genetic algorithm were also performed, as I thought this was an interesting part of the algorithm. The main and clear one was how the number of generations affected the tour value and having a larger number of generations led to a smaller tour value in general. This proved that the algorithm was valid, but also showed that the random element of the algorithm meant that increasing the number of generations did not always lead to a better tour value. The parameter for the best percentage from the population I found was in general good to keep at around 20%, as less meant it was mostly random how the population evolved and having more led to the algorithm getting stuck and evolving at a slow rate. For the parameter of mutation rate percent (the one that I found the most interesting), I found that depending on the size of population, having a higher mutation rate was better. For small populations, high mutation rates helped get closer to a good solution, as swapping two random nodes has a big impact on the tour value. For large populations, smaller mutation rates were better, as it was not so necessary to swap nodes to have variation in the population. Overall, a mutation rate of around

0.1 was the best in most cases. Lastly, increasing population size also led to better performance in general of the algorithm.

Conclusion

The genetic algorithm was a fun and different approach to the TSP, and proved that in some cases, you need randomness to continue to improve in solving the algorithm. Nevertheless, its slow runtime and needing many generations to provide a satisfactory solution shows that the other algorithms are better suited to solve this problem in most cases

References:

https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35

https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6

https://www.hindawi.com/journals/cin/2017/7430125/