EC Assignment 1 – TSP

Harry Stevenson - 2341315

Simulated Annealing

Design Decisions

Simulated annealing is a stochastic search algorithm that aims to find an optimal solution. It does this by implementing a random element as to not get stuck in a local optima. To implement the Simulated Annealing algorithm for the TSP problem, I need to define an objective function to calculate the quality of a given solution, and a neighbour function to define how to move from one solution to a solution with a slight difference.

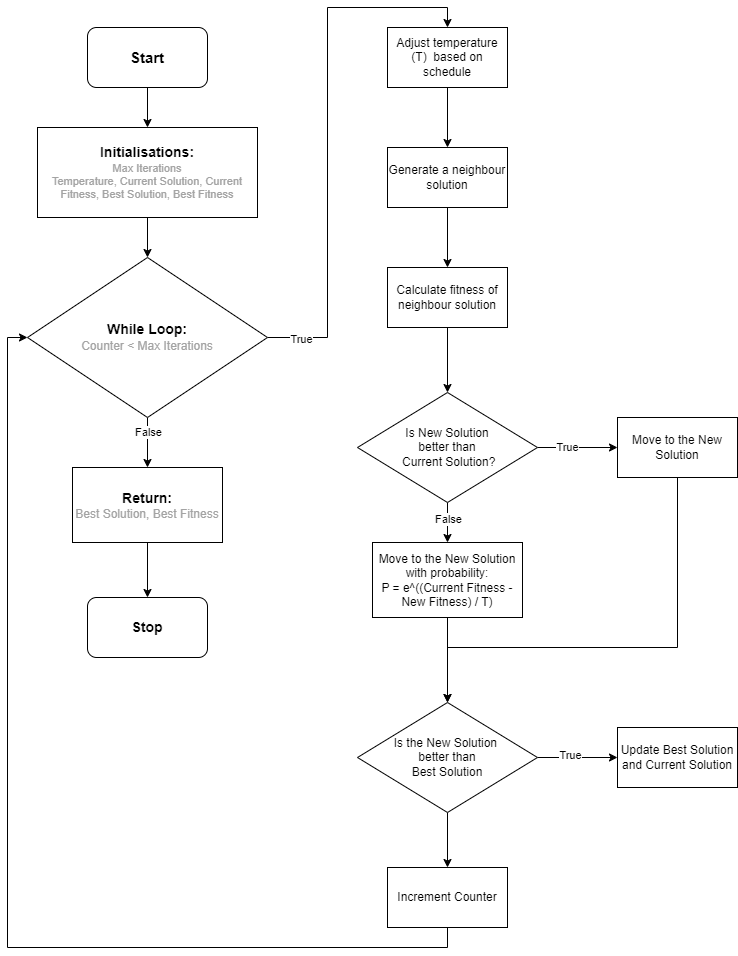
One key step that I implemented in both approaches is ordering the solution as part of the initialisations. This led to more of a focus on exploitation than exploration as the solutions were generated closer to that local optima. I chose to do this as the results it produced were significantly better than starting from an unordered solution, approximately halving the total distance.

For my neighbour solution I decided to swap the locations of 2 cities within the solution. As the quality of a TSP solution depends on the order of the elements within it, this effectively gives a new solution that doesn’t differ greatly from the original, while giving chance for a large change in the fitness.

The objective function of TSP is given clearly in the definition: to minimise the total distance travelled. Therefore, my implementation calculates the sum of the Euclidian distance from one city to the next in the solution.­­­­

To allow Simulated Annealing to converge to an optima after exploring the search space, we use a temperature value which determines the probability of moving to the neighbour solution. ­ To control the temperature, I use initial temperature and cooling rate parameters. The cooling rate determines is multiplied by the temperature at each step, determining how quickly the temperature decreases, meaning it is more likely to move to a good solution as the algorithm runs.

Flowchart



Genetic Algorithm

Design Decisions

In order to create an effective genetic algorithm, a suitable encoding scheme must first be chosen, then selection, variation, fitness and reproduction methods need to be designed with the problem and encoding scheme in mind.

To encode the TSP problem, I decided to maintain the scheme found in the data file. With each city being represented by a tuple of size 2 and type int, each solution is a list of these tuples. I also chose to maintain this throughout the Variation step. My main reason behind this is because the solution only allows coordinates defined, so randomly flipping bits and mapping these to solutions will have the same effect as randomly swapping cities, as I have implemented. This is very convenient as it allows me to do comparisons and calculations on the solutions very easily. To encode a population, I simply have a list of solutions that I can iterate through. I start the algorithm by generating a population of neighbour solutions.

The fitness of each solution is calculated in the same way as in Simulated Annealing, allowing an insightful comparison of the two approaches. One subtle difference between the two objective functions is that the GA objective function contains a penalty function component. This gives a penalty for every duplicate city, multiplied by a parameter ‘penalty weight’. By doing this I discourage the selection of the constraint-violating solutions and increase the likelihood that they are replaced during reproduction. I chose to punish for duplicate cities as it handles the constraint that every city must appear once and only once in the solution.

For the selection step, I decided to use tournament selection. This selects a given number solutions at random and returns the one with the greater fitness. This is repeated until we have a specified number of parents. This type of selection takes 2 parameters so it will be interesting to see the effect of these on the results of the algorithm.

For the variation step I used both mutation and crossover. For the mutation step I used the neighbour function from Simulated Annealing (swapping 2 random cities in the solution). This happens with a probability of 1/population size, meaning approximately 1 parent will undergo mutation in each iteration. For the crossover step, I decided to implement ordered crossover. This crossover operator is extremely useful in this case as it preserves the ordering of the two parents within the offspring. This approach does however violate constraints as the same city could appear in different locations in both parents, meaning both could be copied to the offspring.

For reproduction, I decided to go with generational reproduction, using all of the new offspring to replace the worse-performing solutions in the population. I chose this as a middle ground between Steady-state and elitism, allowing for a good number of new offspring, while conserving a relatively stable population.

Flowchart

A diagram of a diagram

Description automatically generated

Parameter Tuning

**SCRIPTS ARE READY FOR TUNING**

Just define ranges!

ga after 1000 runs

A screen shot of numbers

Description automatically generated

fitness of ordered cities

59026.124

* Parameters
  + How they affected performance
* List average results and Standard deviations from 30 runs
* How I compare SA and GA statistically