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.

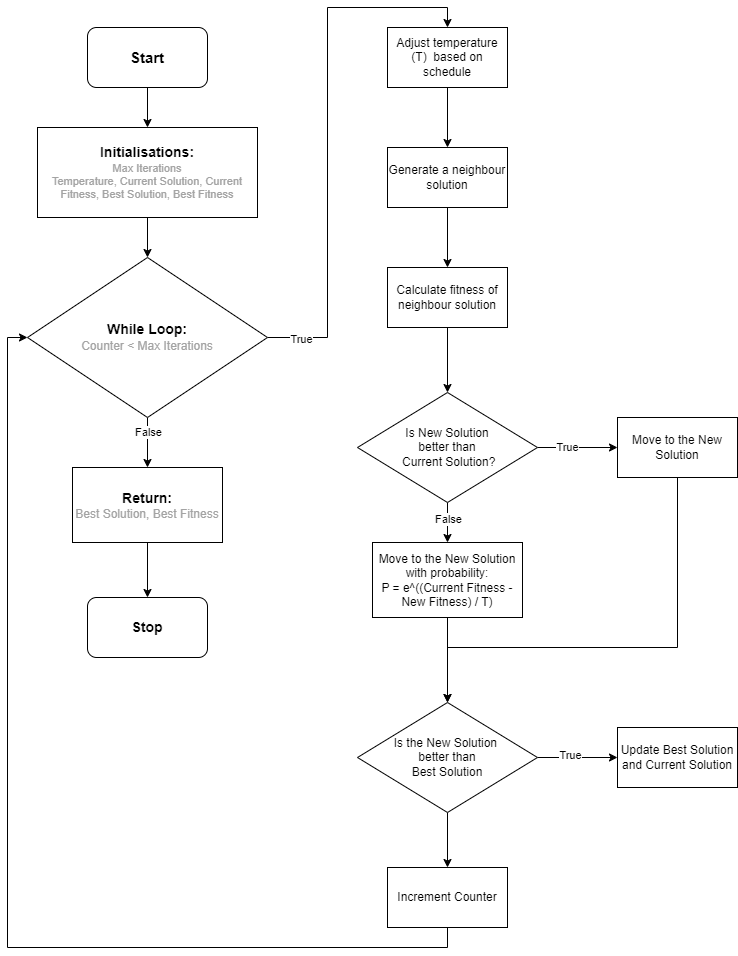
One key step that I implemented in both approaches is ordering the solution as part of the initialisations. This led to more of a focus on exploitation than exploration as the solutions were generated closer to that local optima. I chose to do this as the results it produced were significantly better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

A diagram of a diagram

Description automatically generatedFlowchart

Parameter Tuning

Simulated Annealing

Initial Temperature and Cooling Rate

For the initial temperature, [1] suggests an initial temperature of 1000, again setting a tuning range of +-20%, from the 100 runs an initial temperature of 1131 gave the best solution. For cooling rate study [2] suggests a value between 0.8 and 0.99 for slow learning. For this parameter I decided to maintain this suggested range, and the final best value was 0.95.

The temperature determines the likelihood of accepting a worse solution, and the cooling rate influences how the temperature changes during execution. As I decided to use a geometric scheduling scheme, the cooling rate had a greater effect on the fitness than the initial temperature, as it has a ripple-down effect on all iterations. One observation I made was that runs with an initial temperature in the higher end of the range tended to perform better with a cooling rate in the upper end of its range, and vice versa. This shows there is a relationship between the two that also can be optimised.

Final Parameters (SA):

Initial Temp:

Cooling Rate:

Genetic Algorithm

Population Size and Mutation Rate

Study [3] tests population sizes from 1,000 to 10,000 and concludes that population size is a ‘crucial factor for success in GA applications, I decided to use the lower range of 800 and upper range of 1000 as this would give at least 100 iterations for the 30 independent runs with the tuned parameters. For my implementation of this problem, runs with higher population sizes scored the best, with the top 9 results having a population higher than 9000 and the best performing having a size of 929. The paper also highlights the fact that the mutation rate has a smaller effect as the population size increases. I implemented a constant mutation rate, meaning that on average one solution will mutate each iteration.

Tournament Size and Offspring Size

With binary tournament selection being most common [4] and a size of 6 selected by [5] I decided to use these as my range. This will keep a good balance of minimising the fitness of my offspring and maintaining a stable population. Increasing offspring size means more possible solutions are generated, increasing the area of the search space that is explored, however this also has the effect of increasing the runtime. To balance exploration and runtime I decided to set the range to 5 to 10% of the population size.

Penalty Weight

Because of how I have designed my penalty function, for it to have a meaningful effect on the objective function it needs to be multiplied. Without this it would have essentially no effect on the fitness value as the scale is in the 10,000s and the penalty is in the 10s. Because of this, I set the range to be from 1,000 to 10,000. This proved to be effective as populations with no valid solutions had a penalty weight near to 1000. After running, the best value was 5,539

Final Parameters (GA):

Population Size: **929**

Tournament Size: **3**

Offspring Size: **55**

Penalty Weight: **5539**

* List average results and Standard deviations from 30 runs
* How I compare SA and GA statistically

Reference List:

1. Geng, Xiutang, et al. "Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search." Applied Soft Computing 11.4 (2011): 3680-3689.
   1. <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=80f17858976250b043f0946f927e454a37ef3943>
2. Zhan, Shi-hua, et al. "List-based simulated annealing algorithm for traveling salesman problem." Computational intelligence and neuroscience 2016 (2016).
   1. <https://downloads.hindawi.com/journals/cin/2016/1712630.pdf>
3. Rexhepi, Avni, Adnan Maxhuni, and Agni Dika. "Analysis of the impact of parameters values on the Genetic Algorithm for TSP." International Journal of Computer Science Issues (IJCSI) 10.1 (2013): 158.
   1. <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=76fe335e709579fc5c636a2d4cd99c5627561cd0>
4. Razali, Noraini Mohd, and John Geraghty. "Genetic algorithm performance with different selection strategies in solving TSP." Proceedings of the world congress on engineering. Vol. 2. No. 1. Hong Kong, China: International Association of Engineers, 2011.
   1. <https://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf>
5. Zhong, Jinghui, et al. "Comparison of performance between different selection strategies on simple genetic algorithms." International conference on computational intelligence for modelling, control and automation and international conference on intelligent agents, web technologies and internet commerce (CIMCA-IAWTIC'06). Vol. 2. IEEE, 2005.
   1. <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1631619>