Solving a Travelling Salesman Problem using Simulated Annealing and Genetic Algorithm

Evolutionary Computation

By Joshua Foulkes

Table of Contents

Task

The task is to implement Simulated Annealing (SA) and Genetic Algorithm (GA) algorithms to solve the travelling salesman problem.

Once these algorithms have been implemented, they need to have there hyperparameters tuned to give the best results.

There have been limitations added to the task. These limitations are a max number of 10000 fitness evaluations for each algorithm. This limitation has been implemented as the goal of these algorithms is to give a solution that isn't the best possible but to give a "good enough" solution in a reduced amount of time. Another limitation is a maximum of 30 trial runs to tune the parameters.

The results of both algorithms need to be compared statistically using a Wilcoxon signed-rank test.

MATLAB was used to implement both algorithms.

Intro

The travelling salesman problem (TSP) is a combinatorial optimisation problem that involves a given set number of points/coordinates and distances between each point and to find the shortest route that goes through every point only once and returns to the original location.

One method to solve this is the brute-force approach of trying every possible combination of routes and using the shortest, this method will find the shortest route, but can take incredibly larges amounts of time, this is due to the number of possibilities being equal to the number of cities factorial. This method can therefore not be used for large maps, or problem with many possible solutions.

An improved method is the use of algorithms such as genetic algorithms or simulated annealing which use different methods to compute the best path in a set amount of time.

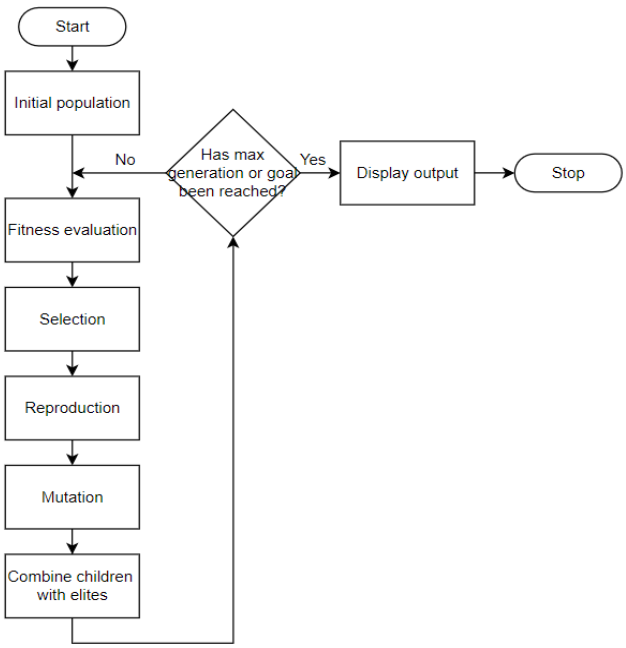
This specific TSP problem involves the 48 capitals of the states of mainland USA; therefore it is more commonly referred to as tsp48. The data such as the locations and distances between each capital has been provided.

Genetic algorithm

A genetic algorithm (GA) is an algorithm that is inspired by the process of natural selection. This algorithm involves the creation of hypothesis of different solutions, evaluating each solution giving them each a score/fitness depending on how well each solution reaches a specified goal. The best solutions are then selected to "reproduce"/be combined to create new solutions, these are then mutated and become part of the new population. The cycle is then repeated until the goal is reached or after a maximum number of generations.

A GA can therefore be split into different sections, which are:

- Representation
- Initial population
- Fitness evaluation
- Selection
- Reproduction
- Mutation



Additional steps can be added to aim for improved results, such as adding elites to a population.

➢ Method

The first step is representation. The representation used a is simple vector notation of each of the indexes of the cities, for this TSP problem there are 48 cities. Ex: [1,2,3,4…24…46,47,48]. This method was used as it is simple, easy to understand and simple to plot on a graph. This method has some issues when it comes to reproduction, but this will be discussed later.

The next step is the creation of the initial population. This was achieved by generating an array of every integer from 1 to 48 (number of cities) and then shuffling them into a random order. This process needs to be repeated for the initial population size. This done by concatenating the arrays together.

The population size is a hyperparameter set at the start. This parameter controls the number of max generations the algorithm can run. Due to max number of fitness calculations being 10000 and each hypothesis requiring 1 fitness calculation per generation the max number of generations can be calculated using the population size.

$$maxg = \frac{10000}{pop}$$

Where maxg is the max number of generations and pop is the desired population size.

The fitness function tells the algorithm how viable a hypothesis is. The fitness function used calculates the length of the path in the order of the points provided. This means the lower the value returned for a path the better, as this means the path is quicker.

The goal of the algorithm is shown below. Where x is the vector of points and f(x) is the fitness function.

$$x^* = argmin[f(x)]$$

A fitness is assigned to each hypothesis.

The next part involves selecting the parents with the best hypothesises. Instead of selecting the top parents for reproduction, roulette wheel resampling will be used, this means there is an element of randomness, meaning there are elements of exploration and exploitation. Roulette wheel resampling was chosen due to its simplicity and having previous experience with the technic.

An issue with this method is that it requires that the fitness of the elements need to be "bigger are better", meaning a larger fitness number means a better fitness. To solve this each fitness was subtracted from the highest fitness of the population, then 100 was added to each, this is to still give the largest value a chance of being selected.

$$fitness(i) = \max(fitarray) + 100 - fitarray(i)$$

Then cumulative sum was done to the array of finesses meaning the fitness at point i is equal to the fitness position i plus fitness at position i-1.

$$fitness(i) = fitness(i) + fitness(i-1)$$

This done in MATLAB using the lines below:

```
fitness_mat_s = max(fitness_mat)+100-fitness_mat;  %reverse fitness
for i = 1:pop_size % Begin: Compute Cumulative Distribution
    if i ==1 %.
        FCD(i) = fitness_mat_s(i); %...
    else %.....
        FCD(i) = FCD(i-1) + fitness_mat_s(i);%...
    end %.
end
```

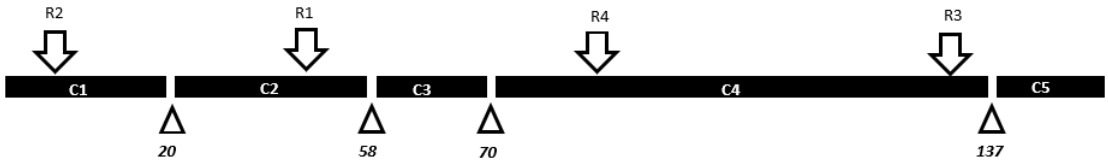
An array of uniformly distributed random numbers was generated of size equal to the number of desired children with the max value being the total of all the finesses or the last value on the cumulative sum array.

The position at where these numbers lie is calculated and then the section where the random number lies becomes a parent.

The diagram below shows roulette wheel resampling.

Previous population	*Fitness value*	*Cumulative sum*
C1	20	20
C2	38	58
C3	12	70
C4	67	137
C5	13	150

	R1	*R2*	*R3*	*R4*
Random numbers up to total fitness (150)	45	6	121	87



Selected Parents
C2
C1
C4
C4

Next comes reproduction. This step involves using the previously selected parents and combining them to create children which are similar.

Firstly, the parents are shuffled randomly. Then the algorithm breeds each parent with the parent next in the array so parent 1 reproduces with parent 2, parent 3 with parent 4, etc. The type of reproduction used is crossover meaning an index in the parents' array is selected randomly. To create the first child the section of parent 1 before the index is concatenated to the section after the point of parent 2, for child 2 the parents are switched (Ucoluk, 2011) [1].

Unfortunately, this method does not work with the vector representation used. To solve this a different representation is used which allows for easy crossover. The different method is shown in (Ucoluk, 2011) [1]. Using this method, the parents are translated into the new representation, the crossover is applied and then the children are translated back to the original representation.

The next stage is mutation. The previous steps have been about exploitation whilst this step is focusing on exploration. This step involves taking the children generated in the previous step and

changing their genome slightly. The algorithm achieves this going through each gene in the chromosome and giving it a probability to mutate. The probability being 1 over the number of genes in this case 1/48. If the probability check is false, the gene is missed and the check proceeds to the next gene. If the probability check is true that gene is selected and a second point is selected at random with a normal distribution around the selected gene. The reason why the randomness is modelled over a normal distribution is to incentivise small changes rather than large ones. After both points have been selected, they switch places and the path between them is flipped to keep the overall path valid, this is shown in the diagram in the method section of simulated annealing This process is repeated for every child. This creates a population of children.

With this current population there is high chance that the algorithm will not converge. To eliminate that possibility "elites" are added. These are the hypothesises with the best fitness'. They are simply selected by ordering the hypothesises by fitness and selecting the top percent. They are then concatenate onto the previously created children. The percentage of elites is a hyperparameter to be tuned.

Finally, after every generation has been run, the best route of the final generation is printed, including its fitness and a visual representation.

- ## Optimizing hyperparameters

This section covers the optimisation of hyperparameters. The hyperparameters to be tuned are the elite's percentage, the population size, and the crossover probability.

The max of 30 trial runs means that the algorithm can be run for 30 runs, therefore the runs are going to be split as follows, each hyperparameter will be run 3 times and the result averaged.

		Elites Percentage					
		0.1		0.3		0.6	
Population Size	5	137653	127742	38082	38960	40451	43180
		122998	129460	37533	38192	40673	41435
	10	48382	45884	37822	41674	42978	43855
		41128	45131	36333	38610	46107	44313
	50	61480	71844	61127	62055	75831	73962
		61877	65067	57896	60359	77282	75692

Key	
fitness 1	fitness 2
fitness 3	avg. fitness

First starting with the population size, it can be observed that on average as the population increases the average fitness decreases, this is due the population being inversely proportional to the max number of generations, meaning that as the population is increased the number of generations decreases not giving enough time for the population to "evolve" causing the fitness to stay high.

The percentage of elites controls the percentage of the population that are elites. This means the ratio of exploration to exploitation of the algorithm can be controlled directly, where a low number of elites favours exploration while a high percentage of elites favours exploitation. The data shows that a good balance of about 30% is a good middle ground. The inferior fitness values when the population is et to 5 and the elites is set to 10% is due the algorithm having to round down the number of elites, causing there to be no elites, therefor the algorithm fails to converge as stated previously what would happen. Similar results can be observed in (Tuning a Traveling Salesman, 2022) [2], these results also reinforce the other hyperparameters chosen.

Crossover Probability	Fitness
0.9	38509
0.7	41513
0.5	38833

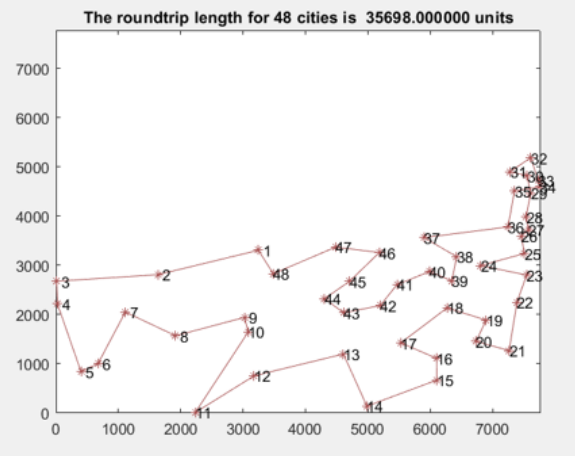
The crossover probability seems to have a very small effect on the results.

The hyperparameters that give the best results are elites percentage of 0.3, a crossover probability of 0.9 and a population size of 5 meaning the algorithm runs for 2000 generations.

- Results

The algorithm was run for 30 iterations using the hyperparameters obtained above, all the runs were averaged out.

The resulting average fitness and best route are shown below:

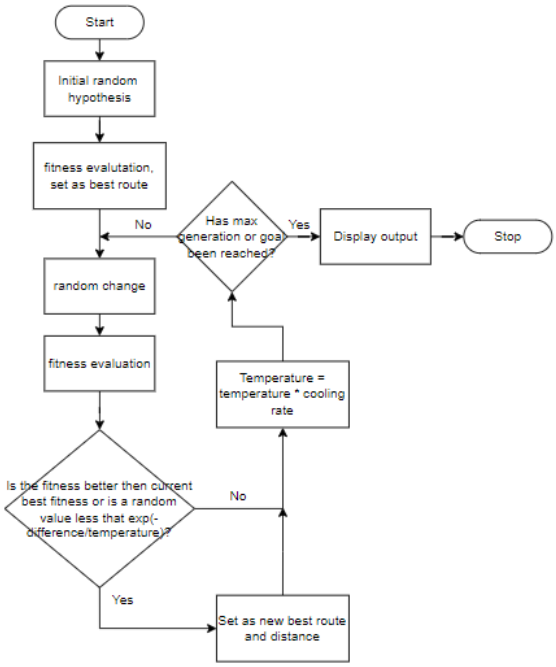


Average Fitness/Distance
39230

Simulated annealing

Simulated annealing is a combinatorial optimisation algorithm based upon the thermodynamics idea of heating and then gradually cooling metal so that the particles will attain the minimum energy state (Hamdar, 2022) [3], the advantage of this method is that it can avoid being trapped in a local minima. This is due to its probability function allowing it to select hypothesis with a worse fitness.

The algorithm follows the flow chart below:



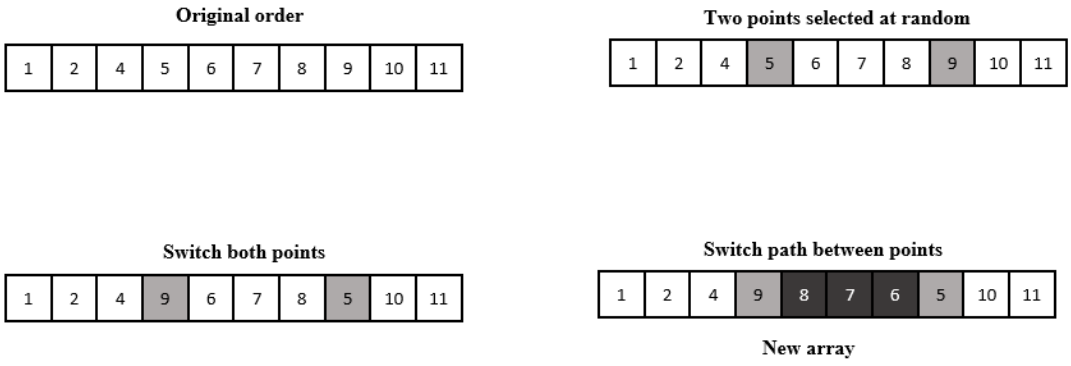
➢ Method

The representation and fitness function used by this Simulated Annealing (SA) algorithm is the same as the ones used by the previous Genetic Algorithm (GA).

Some hyperparameters are initially set, these are the temperature, cooling rate and absoluteTemp.

The first section is to create an initial random hypothesis by shuffling the path into a random order. the fitness of this initial random solution is then evaluated, and this path is set as the best path.

An element of randomness is introduced to the solution by randomly selecting two points in the vector array and switching them and reversing the route between these two points. As shown in the diagram below:



This process is similar to the mutation used in the genetic algorithm.

The fitness of this new array is then evaluated. And the difference between the fitness of this hypothesis and the fitness of best hypothesis is calculated.

Then there is a condition statement. If the difference between the finesses is less than 0, meaning the new hypothesis has a better fitness than the best hypothesis, then this new hypothesis is set as the new best.

An “or” condition is also added which states if the best distance is greater than zero, meaning the new hypothesis has a worse fitness than the best hypothesis a random number is generated between 0 and 1 with a uniform distribution, if this random number is less than exp(-difference/temperature) then this new hypothesis is set as the new best.

If neither condition has been met, then the best distance and fitness is not changed.

The final step says that the temperature is multiplied by the cooling rate and set as the new temperature.

This process is repeated until the max number of generations has been reached, 10000 or the temperature is equal to the absoluteTemp.

Finally, after every iteration has been run, the best route of the final generation is printed, including its fitness and a visual representation.

- Optimizing hyperparameters

This section will cover the optimisation of the hyperparameters to improve results. A limitation has been put into place which is the maximum number of trial runs, being 30.

The hyperparameters to be tuned are the temperature, the cooling rate, and the absolute temp.

The 30 trial runs means that the algorithm can be run for 30 runs, therefore the runs are going to be split as follows, each hyperparameter will be run 3 times and the result averaged.

The outcomes of changing the cooling rate and temperature are shown below, with the absolute temperature set to 0.001:

Temperature	Cooling Rate: 0.99		0.999		0.9999	
10 000	162514	146606	34270	34792	125959	128496
	171137	160090	34417	34493	131586	128680
1000	155337	132115	35048	34767	47318	47357
	171942	143130	34666	34827	47902	47560
100	158979	145347	35929	35829	35036	34959
	170753	158360	35050	35603	34831	34942

Key	
fitness 1	fitness 2
fitness 3	avg. fitness

Starting with the cooling rate it can be observed that a lower cooling rate leads to bad results. This is believed to be due to the temperature decreasing too quickly meaning it reaches the absolute temperature too quickly stopping the algorithm too early. But this can have some issues if the cooling rate is too high, for example when the cooling rate is 0.9999 and the temperature is 10000, this could be due to algorithm excepting bad results to frequently due to the exp(-difference/temperature) is too high causing the random number to be more than likely lower than the threshold making the algorithm except worse results.

Another test was done where the cooling rate was set to 0.999 and the temperature to 10000, due to these hyperparameters giving the best results and then the absolute temp was changed. No result averaging was used due to the 30-test limit.

Absolute Temperature	Fitness
0.01	34429
0.001	35548
0.0001	35901

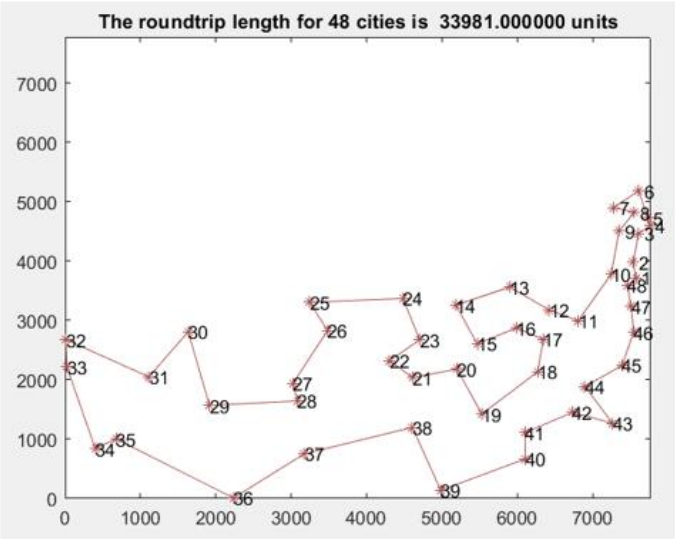
The absolute temperature has very little effect on the results. This might be due to the temperature never reaching the absolute temp before the number of iterations runs out. The absolute temp might have an effect if set higher which will reduce the total number of iterations as the algorithm will terminate when the temperature reaches the absolute temperature. The value of the absolute temperature may have an effect if the algorithm is allowed to run for more iterations, giving time for the temperature to reach the absolute temperature or if the cooling rate is decreased meaning the temperature will drop quicker.

The hyperparameters which give the best results seem to be when the cooling rate is 0.999, the temperature is 10000 and the absolute is 0.01.

- Results

The algorithm was run for 30 iterations using the hyperparameters obtained above, all the runs were averaged out.

The resulting average fitness and best route are shown below:



Average Fitness/Distance
35169

Comparison

The simulated annealing algorithm seems to give better results compared to the genetic algorithm, this may be due to the genetic algorithm not being fully optimised and therefore could be improved to give similar results to the simulated annealing algorithm.

The results of both algorithms when run for 30 iterations were put through a Wilcoxon signed rank test using a MATLAB function. The results of this test are shown below:

p-value	1.7333e-06
z-value	4.7823
signed rank	465

These results indicate that “the test rejects the null hypothesis that there is no difference between the grade medians at the 5% significance level” (signrank, 2022) [4].

References

[1] Ucoluk, G., 2011. *Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation.* [ebook] Ankara, Turkey: Department of Computer Engineering Middle East Technical University. Available at: <https://user.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf> [Accessed 2 March 2022].

[2] Medium. 2022. *Tuning a Traveling Salesman*. [online] Available at: <https://towardsdatascience.com/tuning-a-traveling-salesman-cadfd7d22e1c> [Accessed 6 March 2022].

[3] Hamdar, A., 2022. *Simulated Annealing - Solving the Travelling Salesman Problem (TSP)*. [online] Codeproject.com. Available at: <https://www.codeproject.com/articles/26758/simulated-annealing-solving-the-travelling-salesma> [Accessed 6 March 2022].

[4] Mathworks. 2022. signrank. [online] Available at: <https://uk.mathworks.com/help/stats/signrank.html> [Accessed 7 March 2022].