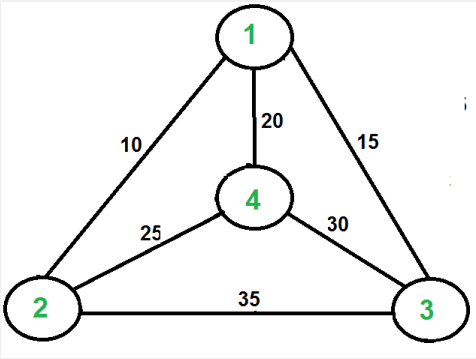
Travelling Salesman Problem

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.

Following are different solutions for the traveling salesman problem.

Naive Solution:
1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.

Ref: https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

Genetic Algorithms: The Travelling Salesman Problem

This week we were challenged to solve The Travelling Salesman Problem using a genetic algorithm. The exact application involved finding the shortest distance to fly between eight cities without visiting a city more than once. The table below shows the distances between each city in kilometres.

Ref: https://medium.com/@becmjo/genetic-algorithms-and-the-travelling-salesman-problem-d10d1daf96a1

	Brighton	Bristol	Cambridge	Glasgow	Liverpool	London	Manchester	Oxford
Brighton	0	172	145	607	329	72	312	120
Bristol	172	0	192	494	209	158	216	92
Cambridge	145	192	0	490	237	75	205	100
Glasgow	607	494	490	0	286	545	296	489
Liverpool	329	209	237	286	0	421	49	208
London	72	158	75	545	421	0	249	75
Manchester	312	216	205	296	49	249	9	194
Oxford	120	92	100	489	208	75	194	0

The search space for this problem involves every possible permutation of routes that visit each city once. As there are eight cities to be visited, and because once a city has been visited that city is not eligible to be travelled to again, the size of the search space stands as 8! (40320) possible permutations.

Initialisation

Upon initialisation, each individual creates a permutation featuring an integer representation of a route between the eight cities with no repetition featured. A corresponding array with the string equivalent of these indexes is created to output when a solution is found.

```
[0, 4, 1, 2, 6, 5, 7, 3] = ["Brighton", "Liverpool",
"Bristol", "Cambridge", "Manchester", "London", "Oxford",
"Glasgow"]
```

A fitness function calculates the total distance between each city in the chromosome's permutation. For example, in the ordering above, the distance between the cities represented by '0' and '4' is added to an overall sum, then the distance between the cities represented by '4' and '1' is added, and so on. The chromosome's fitness is set to the overall sum of all distances within the permutation. The smaller the overall distance, the higher the fitness of the individual.

Selection

There are two popular methods to apply when performing selection in genetic algorithms, roulette wheel selection and tournament selection. Both use probability to create bias in choosing fitter chromosomes to serve as the parents.

In my first implementation, I used the roulette wheel method to select two parents to produce two offspring within each generation. Each chromosome calculates a selection probability with a higher fitness correlating to a higher probability. An overall sum is created of the fitnesses of each chromosome within the population. Each chromosome's probability is then calculated by dividing it's individual fitness by the total probability sum which results in a float value in the range of 0 to 1. An array is created to store the individual probabilities and act as the roulette wheel in which to determine the parents.

To select the parents, a random number is generated between 0 and 1. The number is then tested against the array values to test which two indexes it lies between and therefore which chromosome to select as a parent. This process is iterated twice to select two parents and a check is in place to guarantee that the two parents are different. Below is a representation of the probabilities generated by 5 different chromosomes based upon their individual fitness. On the right is the representation of the array in which to test the randomly generated number against. The first value always serves as 0, the next as 0+chromosome1.probability, the next as chromosome1.probability+chromosome2.probability, and so on for all entries.

```
[0.3, 0.25, 0.2, 0.15, 0.1] --> [0, 0.3, 0.55, 0.75, 0.9, 1]
```

Tournament selection also involves this process of calculating probability bias however features a stronger stochastic element in choosing parents. The tournament consists of considering only a select few chromosomes as parents rather than the whole population. The size of the tournament is assigned at the beginning of the program, and the chromosomes are chosen from a random starting point to create a tournament of the set size. The same process as the roulette wheel is used to determine the two parents for the crossover phase.

Crossover

As the solution requires that no city to be visited more than once, using a classical crossover operator may often lead to generating weaker offspring. A conventional crossover operator may select one half to copy from the first section and the remaining half to copy from the second. This does not prevent copying duplicate cities into the offspring chromosome and will result in a penalty for visiting the same city more than once. However, using the order 1 approach helps to preserve the non-repeating feature of the parent chromosomes. To create the first child, copy part of the first parent's chromosome to the child's chromosome. Then choose valid non-repeating numbers from the second parent in the order that they appear to the empty values in the child's

chromosome. To create the chromosome for the second child, repeat this process inversely by copying part of the second parent's chromosome and using valid values from the first parent for the remaining values.

```
//Classical Crossover Operator
[0, 6, 2, 7, | 1, 4, 3, 5] //Parent 1
[2, 1, 7, 0, | 6, 5, 4, 3] //Parent 2
[0, 6, 2, 7, | 6, 5, 4, 3] //Offspring 1
[2, 1, 7, 0, | 1, 4, 3, 5] //Offspring 2
//Order 1 Crossover Operator
[0, 6, 2, 7, 1, 4, 3, 5] //Parent 1
[2, 1, 7, 0, 6, 5, 4, 3] //Parent 2
[0, 6, 2, 7, 1, 4, 3, 5] //Offspring 1
[2, 4, 7, 0, 6, 5, 1, 3] //Offspring 2
```

Mutation

For this specific problem, the standard mutation action is modified to avoid creating repetition of cities visited. A conventional mutation method of replacing a value with a random city would duplicate cities and not fulfil the problem's criteria. Instead, two random indexes are selected in the array holding the city indexes and these two values are swapped to maintain .

```
[0, 6, 2, 7, 1, 4, 3, 5] --> [0, 4, 2, 7, 1, 6, 3, 5]
```

Modalities

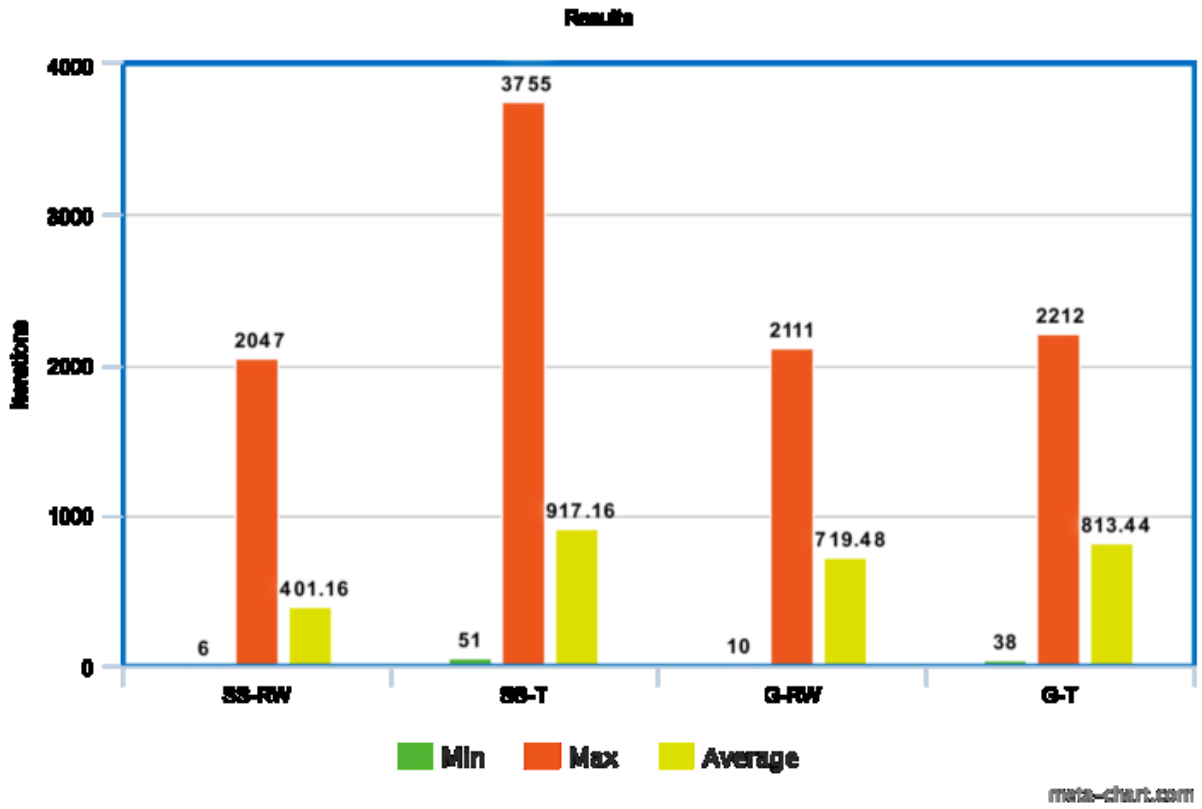
Genetic algorithms have two modalities, steady-state and generational. Steady-state utilises an elitist selection process in which the best n chromosomes are of the population are carried over to the next generation. By keeping the best-ranked chromosome, this implementation does not risk losing it's best solution so far .Generational does not utilise this approach and instead only carries over any offspring produced in the crossover phase.

Comparison

The four variations are noted in the results as:

- Steady-State Approach with Roulette Wheel Selection (SS-RW)
- Steady-State Approach with Tournament Selection (SS-T)
- Generational Approach with Roulette Wheel Selection (G-RW)
- Generational Approach with Tournament Selection (G-T)

Each variation was tested 50 times and the minimum, maximum, and average number of generations needed to reach the solution was recorded. The same mutation rate of 0.15 and population size of 25 was used for each implementation.



As shown, there is a notably wide variation in results between the tournament and roulette wheel selection in the steady-state modality. By using tournament selection, there is a chance that the fittest chromosome will not be considered for crossover and so will not pass it's candidate solution on to successive offspring. This same logic applies to the difference between the roulette wheel and tournament selection in the generational modality although the difference in average is not as significant.

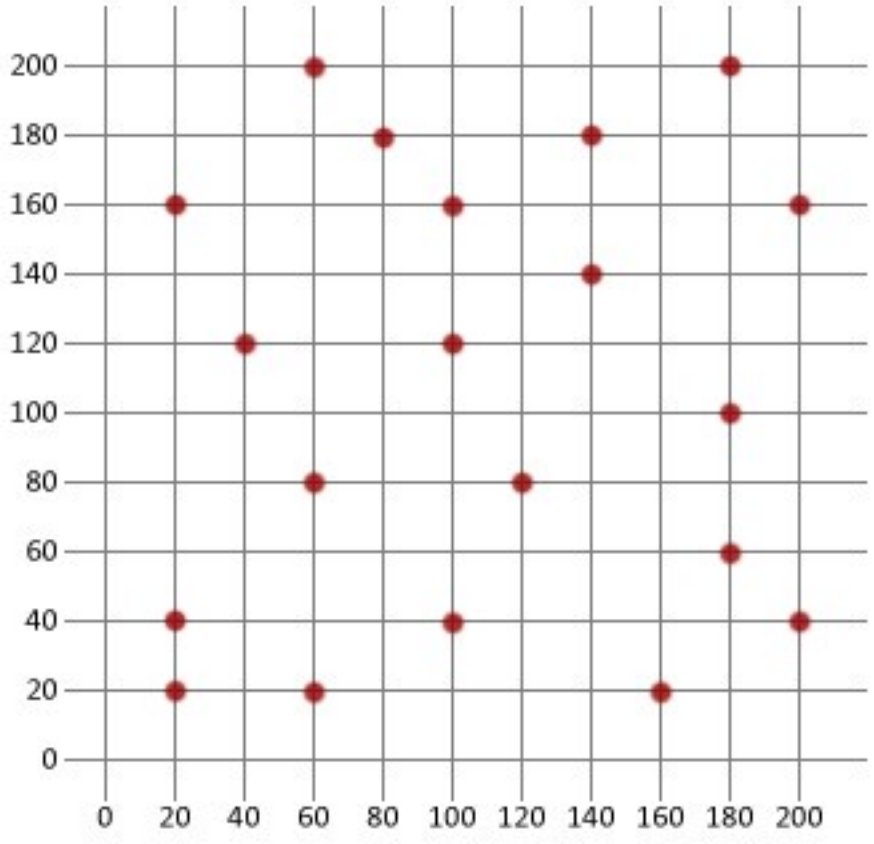
It is surprising that the steady-state tournament selection was outperformed by both generational variations. The steady-state holds onto the best chromosome found, and so there is no danger of losing the best solution of the population. If this best chromosome is in fact a local solution, any chromosome of the new generation that surpass its fitness will replace it as the new best and so avoids pitfalls of local optima. Using the generational approach implies that finding the global solution relies entirely upon the selection, crossover, and mutation process of the chromosomes in order to converge to the target solution vector. This suggests a much slower convergence, as to find the global solution requires a steady and gradual progression from initialisation candidate solutions to the global optima.

However, the stochastic aspect of the mutation threshold and selection process removes consistency from the performance of GA. Another fifty tests may yield entirely different results for each variation that may be more expected or entirely contradict what would be logically assumed. However, if these results do accurately describe the behaviour of the

variations, the steady-state approach and tournament selection method may benefit in more creative applications, where exploration and a slow convergence may demonstrate an auditory or artistic process. Overall, considering the total size of the search space mentioned in the introduction, the genetic algorithm serves well in finding a solution in a relatively small number of generations.

Applying a genetic algorithm to the traveling salesman problem

To understand what the traveling salesman problem (TSP) is, and why it's so problematic, let's briefly go over a classic example of the problem.



Imagine you're a salesman and you've been given a map like the one opposite. On it you see that the map contains a total of 20 locations and you're told it's your job to visit each of these locations to make a sell.

Before you set off on your journey you'll probably want to plan a route so you can minimize your travel time. Fortunately, humans are pretty good at this, we can easily work out a reasonably good route without needing to do much more than glance at the map. However, when we've found a route that we believe is optimal, how can we test if it's really the optimal route? Well, in short, we can't - at least not practically.

To understand why it's so difficult to prove the optimal route let's consider a similar map with just 3 locations instead of the original 20. To find a single route, we first have to choose a starting location from the three possible locations on the map. Next, we'd have a choice of 2 cities for the second location, then finally there is just 1 city left to pick to complete our route. This would mean there are 3 x 2 x 1 different routes to pick in total.

That means, for this example, there are only 6 different routes to pick from. So for this

case of just 3 locations it's reasonably trivial to calculate each of those 6 routes and find the shortest. If you're good at maths you may have already realized what the problem is here. The number of possible routes is a factorial of the number of locations to visit, and trouble with factorials is that they grow in size remarkably quick!

For example, the factorial of 10 is 3628800, but the factorial of 20 is a gigantic, 2432902008176640000.

So going back to our original problem, if we want to find the shortest route for our map of 20 locations we would have to evaluate 2432902008176640000 different routes! Even with modern computing power this is terribly impractical, and for even bigger problems, it's close to impossible.

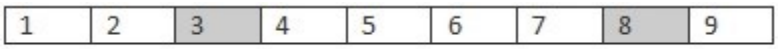
Finding a solution

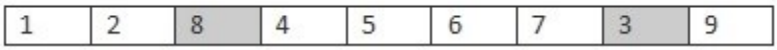
Although it may not be practical to find the best solution for a problem like ours, we do have algorithms that let us discover close to optimum solutions such as the nearest neighbor algorithm and swarm optimization. These algorithms are capable of finding a 'good-enough' solution to the travelling salesman problem surprisingly quickly. In this tutorial however, we will be using genetic algorithms as our optimization technique.

Finding a solution to the travelling salesman problem requires we set up a genetic algorithm in a specialized way. For instance, a valid solution would need to represent a route where every location is included at least once and only once. If a route contain a single location more than once, or missed a location out completely it wouldn't be valid and we would be valuable computation time calculating it's distance. To ensure the genetic algorithm does indeed meet this requirement special types of mutation and crossover methods are needed.

Firstly, the mutation method should only be capable of shuffling the route, it shouldn't ever add or remove a location from the route, otherwise it would risk creating an invalid solution. One type of mutation method we could use is swap mutation. With swap mutation two location in the route are selected at random then their positions are simply swapped. For example, if we apply swap mutation to the following list, [1,2,3,4,5] we might end up with, [1,2,5,4,3]. Here, positions 3 and 5 were switched creating a new list with

exactly the same values, just a different order. Because swap mutation is only swapping pre-existing values, it will never create a list which has missing or duplicate values when compared to the original, and that's exactly what we want for the traveling salesman problem.



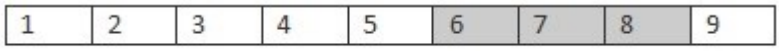


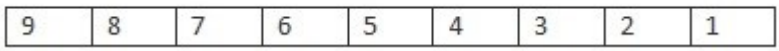
Now we've dealt with the mutation method we need to pick a crossover method which can enforce the same constraint.

One crossover method that's able to produce a valid route is ordered crossover. In this crossover method we select a subset from the first parent, and then add that subset to the offspring. Any missing values are then adding to the offspring from the second parent in order that they are found.

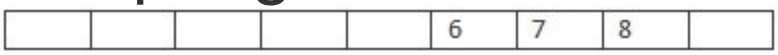
To make this explanation a little clearer consider the following example:

Parents





Offspring



Here a subset of the route is taken from the first parent (6,7,8) and added to the offspring's route. Next, the missing route locations are adding in order from the second parent. The first location in the second parent's route is 9 which isn't in the offspring's route so it's added in the first available position. The next position in the parents route is 8 which is in the offspring's route so it's skipped. This process continues until the offspring has no remaining empty values. If implemented correctly the end result should be a route which contains all of the positions it's parents did with no positions missing or duplicated.