# Parallel Genetic Algorithm Solution to the Traveling Salesman Problem

The provided code for implementing genetic algorithms is completely reusable and was first implemented using the included Word-Game example as a proof of concept. The infrastructure allows the implementation of arbitrary genetic algorithms, not just the ones provided. Additional features include: Parallelization, Stabilization, Reusability and Extensibility.

## Problem Statement

We are given a n by n matrix, containing the cost to travel from city i to city j .  
It is given that for any two cities, there exists a path between them.  
We need to find the cheapest path containing all cities without repetitions.

## Build and Run

1. Download .NET 6 SDK from <https://dotnet.microsoft.com/en-us/download/dotnet/6.0>
2. Install the SDK
3. Extract source-code.zip
4. In a command prompt, navigate to the extracted folder
5. run ‘dotnet test’

## Tests and Results

* The file TravelingSalesmanTestResults.txt contains the result from running all implemented integration tests. The same output can be generated using the steps listed in the previous paragraph.
* The first of the tests is the one from the Homework problem statement, the rest were found online.
* Optimal path lengths have been verified

## Solution

### Population

We are going to represent a possible solution to the Traveling Salesman Problem as a list of numbers.  
Any possible solution is going to contain a sequence of the indices of all cities in any order without repetitions. This is true, because based on the problem statement, a path exists between any two cities.  
This means that each possible solution is also a permutation of the numbers from 0 to n-1 (n being the number of cities).

Our Chromosomes’ DNA will consist of numbers representing indices of cities.

### Initialization

We will initialize a population of predetermined size, by generating random permutations of the numbers [0, n). Each permutation will represent a chromosome of our initial population – an array of numbers.

### Fitness Function

The cost of a route is going to be the sum of the weights of each edge that it contains.  
For the fitness function, we need a number that is inversely proportional to the cost, so that the cheaper Mutation

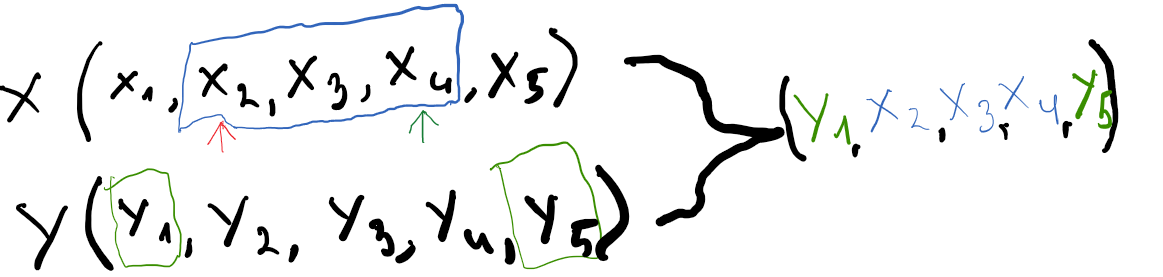
1. Generate a random number x. If the number is less than the configured mutation rate, then break.
2. Generate two more random numbers a,b in the range [0, n-1]
3. Swap the gene at index a with the gene at index b.

### Parent Selection

1. Given a population of size M, generate a random number a in the range [0, M-1]
2. M[a] -> current
3. Take the fitness of current(M[a])-> maxFitness
4. Generate random number b in the range [0, M-1]
5. M[b] -> current
6. Generate random number c in the range (0,1)
7. If (maxFitness\*c < current.fitness) then return current
8. Else, go to 3.

### Crossover

1. Select two random numbers a, b in the range [0, n-1]
2. Take the elements in the range [a,b] from the first parent chromosome and write them to the child chromosome array at the same positions.
3. Sequentially fill the gaps with the first unused element from the second parent



The process ensures we never have repetitions in the resulting array by using a hash set and skipping the elements from the second parent that are already contained in the child. This way we guarantee (albeit with a little extra memory) that the child chromosome is going to be a valid permutation, and hence solution to the TSP.

### Generate Next Population

1. **Evaluate**Calculate the fitness values for each of the chromosomes in our population.

If the fitness of our best chromosome exceeds our TargetFitness (if applicable), then return – we found our solution.

1. **Crossover**  
   Take M/2 pairs of parents and run Crossover to generate a population with M parents.
2. **Mutate**Execute mutate for each chromosome by passing in a mutationRate (chance).

### Parallelization

The implemented solution supports parallelization for an arbitrary number of cores.

1. **Divide**  
   Given a population size of M, and a thread count of T, for each thread a separate population of size M is created and evaluated.
2. **Conquer**GenerateNextPopulation is run in parallel T times using a single population as its source.
3. **Merge**The resulting populations for all threads are merged and sorted by their fitness.  
   The best M chromosomes of the created populations are selected for our new population.

### Stabilization

A stabilization feature has been implemented that exits the algorithm, whenever a StabilizationThreshold number of populations had the same max fitness value.

This allows the Travelling Salesman Problem to complete.