Travelling Salesman (Longest and Shortest)

Name: Huy Ha

UNI: hqh2101

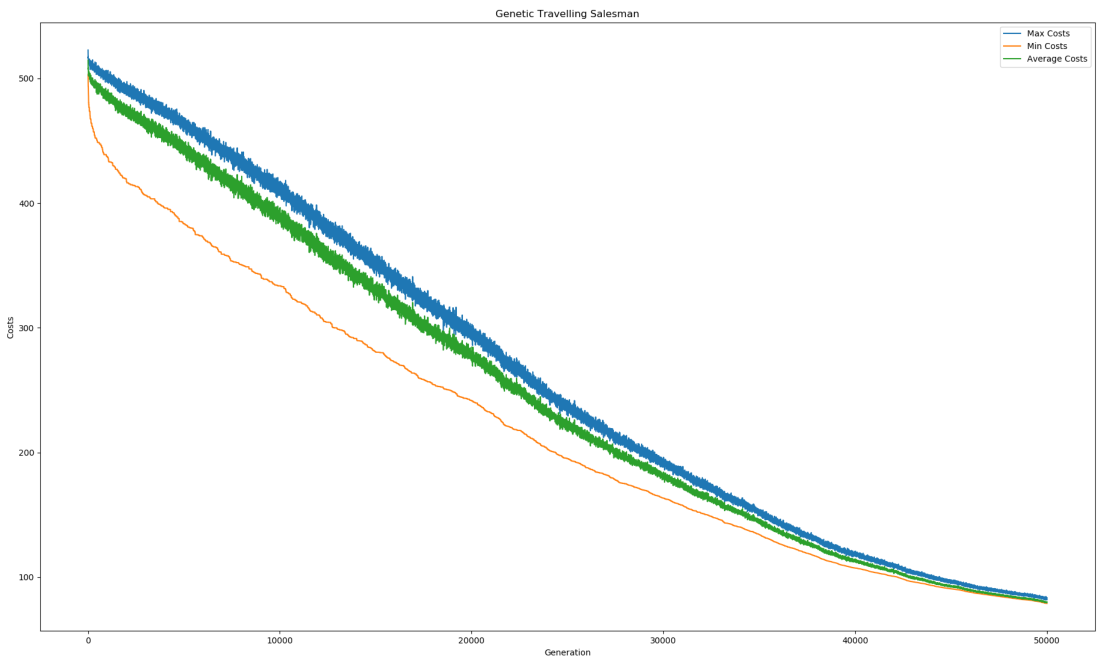
Configuration of runs

The configuration file (list of hyperparameters) I used for these runs are included below. In the graphs below, the blue line represents the population's maximum cost, the green line represents the population's average cost, and the orange line represents the population's minimum cost.

```
Cities File: tsp.txt
FindShortestPath: False
Generation Count: 50000

Population Count: 100
Elitist Percentage: 0.02

Initial Mutation Factor: 0.5
Mutation Factor Decay: 1

Init T: 1
T Decay: 0.9999
```

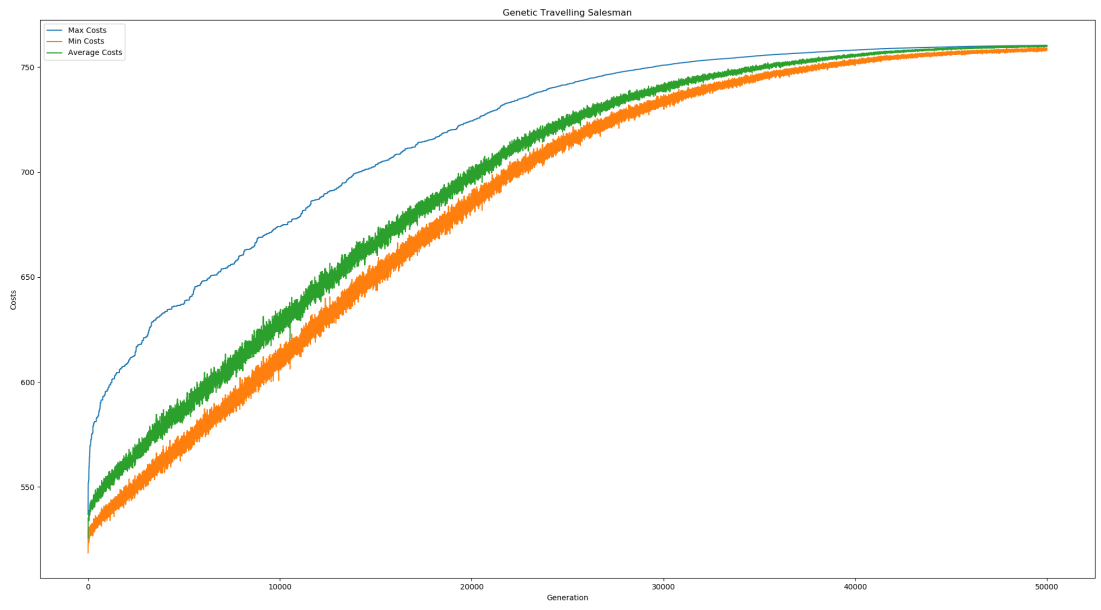
Shortest path

The shortest path I got was `78.90446`



Longest path

The longest path I got was `760.3628`



Code Snippets

I've pasted my code so far for this homework below. My main program is in `C#`, and I use `python` to plot the graphs because I'm already used to using `matplotlib.py`.

Program.cs

```
using System;
using System.IO;
using System.Linq;
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading.Tasks;
namespace EvolutionaryTravellingSalesman
{
    class Program
    {
        static async Task Main(string[] args)
        {
            var filePath = "tsp.txt";
            var cities = File.ReadAllLines(filePath)
                .Select(line =>
                {
                    var coors = line.Split("\t", 2);
                    var x = float.Parse(coors[0]);
                    var y = float.Parse(coors[1]);
                    return new TravellingSalesman.City(x, y);
                });

            // Initialize population
            int populationCount = 100;
            var population = new LinkedList<TravellingSalesman>();
            for (int i = 0; i < populationCount; i++)
            {
                population.AddLast(new TravellingSalesman(cities));
            }
            // Start evolution
            int generationCount = 10000;
            bool findShortestPath = false;
            float elitistPercentage = 0.02f;
            var selector = new SimulatedAnnealingSelected();
            float initMutationFactor = 0.5f;
            float mutationFactor = initMutationFactor;
            float mutationFactorDecay = 1f;

            // Simulated Annealing
            float init_T = 1f;
            float T = init_T; //Probability of choosing a worse gene anyways
            float T_decay = 0.9999f;

            int logCount = 50;
            int logFrequency = generationCount / logCount;

            // Timer for optimization purposes
            selector.Reset();
            Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();

            // Output variables
            string bestSalesmanOutput = "";
            string worstSalesmanOutput = "";
            List<float> minCosts = new List<float>(generationCount);
            List<float> maxCosts = new List<float>(generationCount);
            List<float> avgCosts = new List<float>(generationCount);
            for (int generation = 0; generation < generationCount; generation++)
            {
#if DEBUG
                Console.WriteLine("\nGeneration " + generation);
#endif
                //select
                var parents = selector.Select(population, elitistPercentage,
findShortestPath);
#if DEBUG
                float parentsAvgDist = parents.Average(salesman => salesman.Cost);
                float parentsMaxDist = parents.Max(salesman => salesman.Cost);
                float parentsMinDist = parents.Min(salesman => salesman.Cost);
                Console.WriteLine("Average Parent: " + parentsAvgDist + ", Max: "
+ parentsMaxDist + ", Min: " + parentsMinDist);
#endif
                //mutate
                IEnumerable<TravellingSalesman> offspring = await Task.WhenAll(
                    parents.Select(parent => Task.Run(() =>
                     new TravellingSalesman(parent, T, findShortestPath))));
                while (offspring.Count() < populationCount)
                {
                    offspring = offspring.Concat(                             //
4. Concatenate newly created children with previous children
                        await Task.WhenAll(                                   //
3. Wait until all constructions of children is finished
                            parents.Select(
                                parent => Task.Run(() =>                      //
2. Run the constructor asynchronously
                                new TravellingSalesman(parent, T,
findShortestPath))).ToArray())); // 1. From each parent, create a new
TravellingSalesman
                }
                if (findShortestPath)
                {
                    //elites
                    offspring = population.OrderBy(salesman =>
salesman.Cost).Take((int)(elitistPercentage * populationCount)).Concat(offspring);
                    population = new LinkedList<TravellingSalesman>
(offspring.OrderBy(salesman => salesman.Cost).Take(populationCount));
                }
                else
                {
                    offspring = population.OrderByDescending(salesman =>
salesman.Cost).Take((int)(elitistPercentage * populationCount)).Concat(offspring);
                    population = new LinkedList<TravellingSalesman>
(offspring.OrderByDescending(salesman => salesman.Cost).Take(populationCount));
                }
                T *= T_decay;
                mutationFactor *= mutationFactorDecay;
                // Collect Stats
                float avgDist = population.Average(salesman => salesman.Cost);
                var maxSalesman = population.Aggregate((salesman1, salesman2) =>
salesman1.Cost > salesman2.Cost ? salesman1 : salesman2);
                var minSalesman = population.Aggregate((salesman1, salesman2) =>
salesman1.Cost < salesman2.Cost ? salesman1 : salesman2);
                float maxDist = maxSalesman.Cost;
                float minDist = minSalesman.Cost;

                minCosts.Add(minDist);
                maxCosts.Add(maxDist);
                avgCosts.Add(avgDist);
#if DEBUG
                Console.WriteLine("Average: " + avgDist + ", Max: " + maxDist + ",
Min: " + minDist);
                Console.WriteLine("Population:" + population.Count + ",
MutationFactor:" + mutationFactor);
#endif
                if (mutationFactor == 0) break;
                if (generation % logFrequency == 0 || generation ==
generationCount - 1)
                {
                    Console.WriteLine("Saving Generation " + generation);
                    bestSalesmanOutput += "Generation " + generation + "\n";
                    bestSalesmanOutput += minSalesman.PrintPath() + "\n";
                    worstSalesmanOutput += "Generation " + generation + "\n";
                    worstSalesmanOutput += minSalesman.PrintPath() + "\n";
                }
            }
            stopWatch.Stop();
            TimeSpan ts = stopWatch.Elapsed;
            Console.WriteLine(ts.TotalSeconds);
            // Save Output
            string config = "";
            config += "Cities File: " + filePath + "\n";
            config += "FindShortestPath: " + findShortestPath + "\n";
            config += "Generation Count: " + generationCount + "\n";
            config += "\n";
            config += "Population Count: " + populationCount + "\n";
            config += "Elitist Percentage: " + elitistPercentage + "\n";
            config += "\n";
            config += "Initial Mutation Factor: " + initMutationFactor + "\n";
            config += "Mutation Factor Decay: " + mutationFactorDecay + "\n";
            config += "\n";
            config += "Init T: " + init_T + "\n";
            config += "T Decay: " + T_decay + "\n";
            System.IO.File.WriteAllText("output/Config.txt", config);
            System.IO.File.WriteAllText("output/BestSalesMan.txt",
bestSalesmanOutput);
            System.IO.File.WriteAllText("output/WorstSalesMan.txt",
bestSalesmanOutput);
            System.IO.File.WriteAllText("output/MaxCosts.txt", string.Join("\n",
maxCosts));
            System.IO.File.WriteAllText("output/MinCosts.txt", string.Join("\n",
minCosts));
            System.IO.File.WriteAllText("output/AvgCosts.txt", string.Join("\n",
avgCosts));
        }
    }
}
```

TravellingSalesman.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

public class TravellingSalesman
{
    #region Variables
    private Random m_rand;
    public float Cost
    {
        get;
        protected set;
    }

    public float Fitness
    {
        get
        {
            return 1000 / Cost;
        }
    }

    private List<City> m_path;
    #endregion

    public TravellingSalesman(IEnumerable<City> cities)
    {
        m_path = FindRandomPath(cities);
        CalculateCost();
        m_rand = new Random();
    }

    public TravellingSalesman(TravellingSalesman parent, float T, bool
findShortestPath, float maxMutationFactor = 0.3f)
    {
        m_path = new List<City>(parent.m_path);
        m_rand = new Random();
        Cost = parent.Cost;
        int count = m_path.Count();
        int maxMutations = Math.Max((int)(count * maxMutationFactor), 1);
        int mutations = m_rand.Next() % maxMutations;
        for (int i = 0; i < mutations; i++)
        {
            Mutate(T, count, findShortestPath);
        }
        CalculateCost();
    }

    private void Mutate(float T, int count, bool findShortestPath)
    {
        var testPath = new List<City>(m_path);
        Swap(testPath, m_rand.Next() % count, m_rand.Next() % count);
        float testPathCost = CalculateCost(testPath);
        if ((findShortestPath ? testPathCost < Cost : testPathCost > Cost) ||
m_rand.NextDouble() < T)
        {
            m_path = testPath;
            Cost = testPathCost;
        }
    }

    private void Swap(List<City> path, int i1, int i2)
    {
        var tmp = path[i1];
        path[i1] = path[i2];
        path[i2] = tmp;
    }

    private List<City> FindRandomPath(IEnumerable<City> cities)
    {
        var rand = new Random();
        return new List<City>(cities
            .Zip(
            cities.Select(city => rand.NextDouble()), //create random order
            (city, order) => new { City = city, Order = order }) //create object
with order and city
            .OrderBy(obj => obj.Order) //sort by order
            .Select(obj => obj.City)); //get just the city
    }

    private float CalculateCost(List<City> path)
    {
        float cost = 0;
        for (int i = 0; i < path.Count - 1; i++)
        {
            cost += City.Distance(path[i], path[i + 1]);
        }
        return cost;
    }

    private float CalculateCost()
    {
        Cost = CalculateCost(m_path);
        return Cost;
    }
    public override string ToString()
    {
        return "" + Fitness;
    }

    public string PrintPath()
    {
        return string.Join("|", m_path);
    }

    public class City
    {
        private float m_x = 0;
        private float m_y = 0;

        public City(float x, float y)
        {
            m_x = x;
            m_y = y;
        }

        public static float Distance(City city1, City city2)
        {
            float dx = city1.m_x - city2.m_x;
            float dy = city1.m_y - city2.m_y;
            return MathF.Sqrt(dx * dx + dy * dy);
        }

        public override string ToString()
        {
            return m_x + " " + m_y;
        }
    }
}
```

SimulatedAnnealingSelector.cs

```
using System.Collections.Generic;
using System.Linq;
using System;

public class SimulatedAnnealingSelected
{
    private float m_beta;
    private float m_initT;
    private float m_T;
    private float m_reproductionPercentage;
    public SimulatedAnnealingSelected(float beta = 0.001f, float initT = 100,
float reproductionPercentage = 0.4f)
    {
        m_beta = beta;
        m_initT = initT;
        m_T = m_initT;
        m_reproductionPercentage = reproductionPercentage;
    }

    public void Reset()
    {
        m_T = m_initT;
    }

    //Stochastic Universal Sampling based on pseudocode from
https://en.wikipedia.org/wiki/Stochastic_universal_sampling
    public
    // (IEnumerable<TravellingSalesman>,IEnumerable<TravellingSalesman>)
    IEnumerable<TravellingSalesman>
     Select(IEnumerable<TravellingSalesman> salesmen, float elitistPercentage =
0.1f, bool findShortestPath = true)
    {
        if (findShortestPath)
        {
            return salesmen.OrderBy(salesman => salesman.Cost).Take((int)
(salesmen.Count() * m_reproductionPercentage));
        }
        else
        {
            return salesmen.OrderByDescending(salesman =>
salesman.Cost).Take((int)(salesmen.Count() * m_reproductionPercentage));
        }
    }
}
```

plotter.py

```
import matplotlib.pyplot as plt
import re


def read_file(filename):
    return [float(x) for x in open(filename, "r").readlines()]


def read_path(filename, generation):
    file = open(filename, "r")
    while True:
        curr = int(re.findall('\d+', file.readline())[0])
        if curr >= generation:
            break
    cities = [city.split(' ') for city in file.readline().split('|')]
    x = []
    y = []
    for city in cities:
        x.append(float(city[0]))
        y.append(float(city[1]))
    return x, y


def plot_costs():
    maxCosts = read_file("output/MaxCosts.txt")
    minCosts = read_file("output/MinCosts.txt")
    avgCosts = read_file("output/AvgCosts.txt")

    plt.plot(maxCosts, label="Max Costs")
    plt.plot(minCosts, label="Min Costs")
    plt.plot(avgCosts, label="Average Costs")
    plt.legend()
    plt.title("Genetic Travelling Salesman")
    plt.ylabel('Costs')
    plt.xlabel('Generation')
    plt.show()


def plot_path(generation=0):
    x, y = read_path("output/BestSalesMan.txt", generation)
    plt.plot(x, y)
    plt.title("Genetic Travelling Salesman Path, generation
{}".format(generation))
    plt.show()


if __name__ == "__main__":
    plot_costs()
    # plot_path(generation=98000)
```