380CT_TSP_Bogdan_Stanciu_7269491

April 5, 2019

1 380CT - Coursework

Group members:

- Bogdan A. Stanciu [Greedy Search algorithm] - SID: 7269491
- Vlad Andreescu [Genetic algorithm]

GitHub Link: https://github.coventry.ac.uk/380CT-1819JANMAY/380CT_Coursework_VA_BS
Within this assignment, an *NP-hard* problem is proposed to be observed and analysed, commonly known as "Travelling salesman problem". The problem is as follows:

> *Given a set of cities and the distances between each pair of cities, what is the shortest possible tour that visits each city exactly once, and returns to the starting city?*

2 Introduction

2.1 Background section

2.1.1 Euclidean - Travelling salesman problem (TSP)

Notation Let's consider a completed weighted graph $G = (V, E, W)$ consisting of a set of V of n-vertices(cities), a set of edges $E = \{(i, j) \mid ....\}$ connecting nodes and a set of **non-negative** weights W={w(x,y)} representing the euclidean distances of edge (**x,y**). The graph is *undirected*, meaning that the graph can be traversed in both directions from i to j and also from j to i with respect towards the graph **cycle** definition.

- **Complete**: the graph is undirected, has no self-loops, and each node is connected to all the other vertices.
- **Weighted**: the edges have a weight (a positive integer).
- **Cycle**: a path that visits every vertex once, and goes back to the start point.
- **Total cost of the cycle**: sum of the edge weights of the cycle.
- **representation of a point** - xOy plane: E.g.: $A = (x_1, y_1), B = (x_2, y_2)$
- **Eucledian distance formula** between two points: $d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- **Segment** : two connected nodes (AB)

Problem definition Given G as above, the versions of the TSP are defined as follows:

- **Decisional TSP (D-TSP)**: > Given n cities and a tour from length k. The traveller starts at an arbitrary city, visits every other city just once and returns to the starting point. Does a tour from length k exist?

  NP-complete, because D-TSP $\in$ NP and D-TSP $\in$ NP-hard.

  - D-TSP $\in$ NP: once a cycle is given (a certificate) we can quickly evaluate the its cost in $O(n)$ time to verify it is equal to k.
  - D-TSP $\in$ NP-hard: A similar problem to TSP is the Hamiltonian Circuit problem which was proved by Richard. M Karp that NP-complete is achieved. Therefore, TSP is a NP-complete problem because NP and NP-hard are validated (Kingsford, p.36).

- **Optimization TSP**: > Given G, find a cycle of minimal total cost.

  NP-Hard, because the optimization version of (decision) NP-complete problems are automatically NP-Hard.

2.2 Exhaustive search - brief recap:

As an introduction into the TSP problem, an exhaustive search algorithm has been approached. The method of this algorithm is titled as follows:

> Exhaustive-Search: Generate all possible tours of the cities, and choose the shortest tour (the one with minimum tour length)

Considering the graph G with a number of n-verticies and with a starting vertex 0 as the start and end points. The algorithm presents the following dificulties: - iterate ovell all permutations of the verticies $\{1, 2, \ldots, n-1\}$. - Calculate cost of each permutation and keep track of minimum cost permutation. - Return the cycle with minimum cost.

The number of permutations is equal with $(1)!$ and each permutation has a complexity cost of $O(n)$. Therefore, the time complexity of the `exhaustive-serach algorithm` is $O((n-1!) \times n)$ which is equal to $O(n!)$.

Conclusion: This algorithmic approach will guarantee to solve the problem, but it is inefficient for large sets of cities. The performance of the exhaustive-search algorithm is severely affected by the number of cities, especially when n exceeds a two-digit value ($n > 10$).

2.3 Methodology

2.3.1 Random instances sampling strategy

Let's start with initialising the python libraries.

```
In [3]: import matplotlib.pyplot as plt
        import random
        import time
        import operator
        import numpy as np
        import pandas as pd
```

As mentioned above, we will approach an Euclidean - TSP. This means that the plan is represented on the xOy plane.

From a computational perspective, we can represent cities as points. A class point that hold two-dimensional values (x,y); which can be lately accessed through the class: $p.x$ respective, $p.y$;

The class point inhabits the two-dimensional (*real* x *imaginary*) plane of complex numbers. X - a real number, Y - an imaginary number

Let's consider two points: A(x_1, y_1), B(x_2,y_2); Mathematical expression for distance: d(A,B)=$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

```
In [2]: Point = complex
        City  = Point

        def X(point):
            return point.real

        def Y(point):
            return point.imag

        def distance(A, B):
            return abs(A - B)
```

```
In [3]: A = City(2, 0)
        B = City(0, 7)
        distance(A, B)
```

```
Out[3]: 7.280109889280518
```

The function tour_length represents the sum of distances between each node. In other words, the distance of each tour.

```
In [4]: def tour_length(tour):
            return sum(distance(tour[i], tour[i-1])
                       for i in range(len(tour)))
```

We instantiate the TSP algorithm with a random set of n-cities. Each city subsequently is initialised with a random set of coordinates (i,j);

```
In [5]: def Cities(n, width=900, height=600, seed=42):
            random.seed(seed * n)
            return frozenset(City(random.randrange(width), random.randrange(height))
                             for c in range(n))
```

```
In [6]: Cities(5)
```

```
Out[6]: frozenset({(122+324j), (151+11j), (261+23j), (346+408j), (654+243j)})
```

Note: in Python a fronzenset is **immutable**. Also, it is to be mentioned that, there is a high chance to get cities generated one over another.

2.3.2 Greedy Search

Based on the exhaustive search algorithm, a new algorithmic approach has been proposed to be analysed. A greedy algorithm is usually refered as an intuitive, simple algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem (Brilliant.org, n.d.).

> *Greedy Algorithm: Let's consider an empty set; within this set, we consider that each node is initially represented by its own segment. The algorithm is looking to identify the shortest possible edge that connects two nodes. The distance between these two nodes is labelled as a segment which is later added into the empty set. The process repeats itself until the tour of segments is complete.*

At each step of the algorithm, the shortest possible edge between two *endpoints* is chosen. From a computation perspective, it is required to represent the data as follows:

E.g.: Let's consider the graph *G* with a number of cities equal to 7; each city is labeled A through G:

1. Compute a list of edges sorted by the shortest edge. It must be noted, that each list contain one edge(referred as an **endpoint**) for every distinct pair of cities and if the segment already contains [e.g.: `(A,B)`], the reverse representation cannot exist `(B,A)` and also the list cannot contain `(A,A)`.
2. To represent the full tour of segments, a dictionary that maps the **endpoints** to **segments** is required. > e.g.: `{A: [A,B,C,D,E,F,G]..., G:[A,B,C,D,E,F,G]}` - as it was mentioned, each city at one time can be refered as an endingpoint, but as we join segments togheter, some cities are no longer endpoints and are removed from the dictionary.
3. As we go through the list, in a shortest-first order, we determine that the shortest path from node `A` to node `G` is through node `B`. At the time, `A`, `B` were considered endpoints of different segments, but as they join together into a segment `AB`, the segment itself will represent the endpoint into the dictionary.

Following the example above, a table was created to support the level of understanding:

Shortest Edge	Usage of edge	Resulting Segments
—	—	A; B; C; D; E; F; G;
AB	Join A to B	**AB**; C; D; E; F; G;
EF	Join E to F	AB; C; D; **EF**; G;
BC	Join C to AB	**ABC**; D; EF; G;
CE	Join ABC to EF	**ABCEF**; D; G;
AF	**Discard**	ABCEF; D; G;
CF	**Discard**;	ABCEF; D; G
DG	Join D to G	ABCEF; **DG**;
BF	**Discard**	**ABCEFGD**;
DA	**Complete Cycle**	**ABCEFGD**

2.3.3 Meta-heuristic - Genetic algorithm

As a second approach, we were required to investigate and to implement one meta-heuristic algorithm. As a solution to this problem, our choice was the genetic algorithm. Here is the sketch of the algorithm:

> Genetic Algorithm: For this problem, a genetic algorithm can be used to generate useful solutions to optimization and search problems. The concept of genetic algorithms was inspired by Charles Darwin's theory of natural evolution and represents a selection of the 'fittest' entities for reproduction (Mallawaarachchi, 2017).

At each step, the algorithm selects individuals (nodes) at random from the random sample of cities to be parents and uses them to produce the children for the next generation. Over multiple generations, the population start to evolve towards an optimal solution (Mathworks, 2019).

Primarily rules of the genetic algorithm:

1. The selection process is based on the number of individuals, referred to as parents which contribute to the next generation of population
2. The crossover function - the combination of two individuals, in order to create the next generation(new children)
3. The mutation function - which applies random changes to individuals in order to create new children.

List of variables used in pseudo-code -terminology:

- Gene - genes/traits represented as follows- e.g.: City(x,y)
- Individual - the distance between two nodes | d(A,B), A,B City(x,y)
- Population - a list of all possible routes(individuals)
- Parents - best individuals from the previous generations; eligible to pass the gene to a new generation(child)
- Mating pool - a compounded list of parents chosen for reproduction (using elitism)
- Breed - combination of two routes that servers to create a new route
- Mutation - a fuction that swaps the routes of two cities $\implies$ variation
- Elitism - a function that chooses only the best parents for breeding

3 Testing - conditions for Greedy Search & Genetic Algorithm:

3.0.1 Experimental measurements: time, accuracy and efficiency

3.0.2 Conditions of fair test:

1. For each average case - estimation time of execution is returned, and with each experiment, the samples are randomly generated
2. In order to ensure that the test's results are not altered, we run the test individually.
3. In order to test the algorithms in a proper manner, we ensured that no parameters were changed during the tests(unless is required).

4 Theory

4.1 Greedy search algorithm:

The idea is to:

1. Select a starting node(candidate) from the list of cities
2. Determine the shortest path to the next candidate with respects towards the cycle definition
3. Select the shortest path and join the points together into a "segment" and remove the unnecessary endpoints.
4. Repeat the process until the function indicates that the full tour of segments (starts and ends with the same node).

More formally, the pseudo-code is as follows:
Input: *cities* ← random set of *n* cities
Output: the full *segment* tour (full tour)

```
1.  Function greedy_search(Cities)
2.      edges ← shortest_edge_first(*Cities)
3.      endpoints ← dictionary{endpoint: segment}
4.      for A, B[] in edges do
5.          if A in endpoints and B in *endpoints and endpoints[A] not = endpoints[B]
6.              segment ← function join_endpoints(endPoints, A, B)
7.              if length of segment = length of Cities then
8.                  return segment
9.              end if
10.         end if
11.     end for
```

Input: *cities* ← the random set of cities(already generated)
Output: return the (*edge*, list[distance of edge])

```
1.  Function shortest_edge_first(Cities)
2.      edges ← new list
3.      for A in Cities and B in Cities do
4.          if id(A) < id(B) then
5.              return( edges, distance_list[ ] )
6.          end if
7.      end for _____
```

Input: *endpoints*, point-*A*, point-*B*
Output: returns the *Asegment*

```
1.  Function join_endpoints(endPoints, A, B)
2.      Asegment ← endPoints[A]
3.      Bsegment ← endPoints[B]
4.      if Asegment[-1] not = A then
5.          Reverse Asegment
```

6. **end if**
7. **if** *Bsegment[0]* **not** = B **then**
8. **Reverse** *Bsegment*
9. **end if**
10. **Append** *Bsegment* in *Asegment*
11. **delete** *endPoints[A]
12. **delete** *endPoints[B]
13. endPoints[Asegment[0]] ← endPoints[Asegment[-1]] ← Asegment
14. **return** *Asegment*

The time complexity of the `greedy search algorithm` is determined by the following functions:

1. function `greedy_search` has a time complexity of $O(n)$.
2. function `shortest_edge_first` has a time complexity of $O(n)$.
3. function `join_endpoints` bas a time complexity of $O(1)$.

Therefore, the time complexity of the algorithm is equal to $O(n \times n)$, mathematically wrote as $O(n^2)$.

4.1.1 Meta-heuristic -- Genetic algorithm

The genetic algorithm pattern looks as follow:

1. Random sample of routes, which later become the population
2. Run the fitness function - for each individual and select the parents for the next generation
3. Run the matingPool function - a list which places the 'fittest' individual and the next generations together
4. Breed parents - it takes two parents who will create a new child
5. Run the Mutation function - creates variations
6. Create next Generation
7. Repeat

Pseudo-code:

1. *ListOfCities* random set of n-cities

Input: ListOfCities
Output: A random generated route

1. **Function createRandomRoute**(ListOfCities)
2. *route* new Random Route()
3. **return** route

Input: population
Output: Fitness of every route in population
Notes: For each population member, calculate the fitness and return a sorted list with the best routes

1. **Function rankIndividuals**(population)

```
    fitnessResults  new list
    for i  0,...,length of population do
        fitenssResults[i]  fitness number for population[i]
    end for
    return sorted list of Fitness numbers
```

Input: population, selectionResults
Output: *selectionResults*
Notes: Loops through the population list and based on the *selectionResults* list appends individuals to the matingPool function

```
Function matingPool(population, selectionResults)
    matingPool  new list
    for i  0,...,length of selectionResults do
        index  selectionResults[i]
        Append population[index] in matingPool
    end for
    return matingPool
```

Input: matingPool, eliteSize
Output: *children* individuals resulted from breeding a **Mating Pool**
Notes: breeds the entire population with the most elite individuals, in order to create *children* for the next generation

```
Function breedPop(matingPool, eliteSize)
    children  new list
    length  length of matingPool - eliteSize
    pool  Generate a random matingPool to breen non-elites
    for i  0,...,eliteSize do
        Append matingPool[i] in children (insert all elites in children)
    end for
    for i  0,...,length do
        child  Function breed(pool[i], pool[length of matingPool] - i -1)
        Append child in children (insert breeded remaining individuals into children list)
    end for
    return children
```

Input: population, mutateRate
Output: *mutatedPop* a mutated population
Notes: Create varition in the population

```
Function mutatePoP(population, mutateRate)
    mutatedPop  new list
    for i in range( length of population ) do
        toMutate = Function mutate(population[i], mutateRate)
        Append toMutate in mutatedPop
    end for
    return mutatedPoP
```

Input: population, popSize, eliteSize, mutateRate, generations
Output: *bestRoute* will be calculated in *n*-generation
Notes: The function calculates the best route for a list of n-cities.

1. **Function geneticAlgorithm**(population, popSize, eliteSize, mutateRate, generations)
2. *pop* **Function initialPopulation**(*popSize*, *population*)
3. *initialDistance* 1 / **Function rankIndividuals**(*pop*)[0][1]
4. **for** i 0,...., *generations* **do**
5. *pop* **Function nextGeneration**(*pop*, *eliteSize*, *mutateRate*)
6. **end for**
7. *finalDistance* 1 / **Function rankIndividuals**(*pop*)[0][1]
8. *bestRouteIndex* **Function rankIndividuals**(*pop*)[0][0]
9. *bestRoute* *pop*[*bestRouteIndex*]
10. **return** *bestRoute*

The Big-O notation for the genetic algorithm:

O(population sizeŒmaximum generation ŒO(fitness)Œ(crossover probability ŒO(crossover))+(mutation probability ŒO(mutation)))

Based on loops, the big-O notation is equal to $O(n^{12})$.

5 Practice section

5.1 Generating plots:

```
In [1]: def plot_tour(tour):
            plot_lines(list(tour) + [tour[0]])

        def plot_lines(points, style='bo-'):
            plt.plot(map(I, points), map(J, points), style)
            plt.axis('scaled');
            plt.axis('off')

        def plot_tsp(algorithm, cities):
            t0 = time.clock()
            tour = algorithm(cities)
            t1 = time.clock()
            assert valid_tour(tour, cities)
            plot_tour(tour); plt.show()
            print("{} city tour with length {:.1f} in {:.3f} secs for {}"
                  .format(len(tour), tour_length(tour), t1 - t0, algorithm.__name__))

        def valid_tour(tour, cities):
            return set(tour) == set(cities) and len(tour) == len(cities)
```

In order to visualise the graph's data, different `plot` functions were created. The objective of `plot_tsp` function, is to show the *number of cities generated*, *the tour length*, *the execution time* and also the *algorithm used*.

5.2 Greedy Search - Code and Testing:

5.2.1 Code:

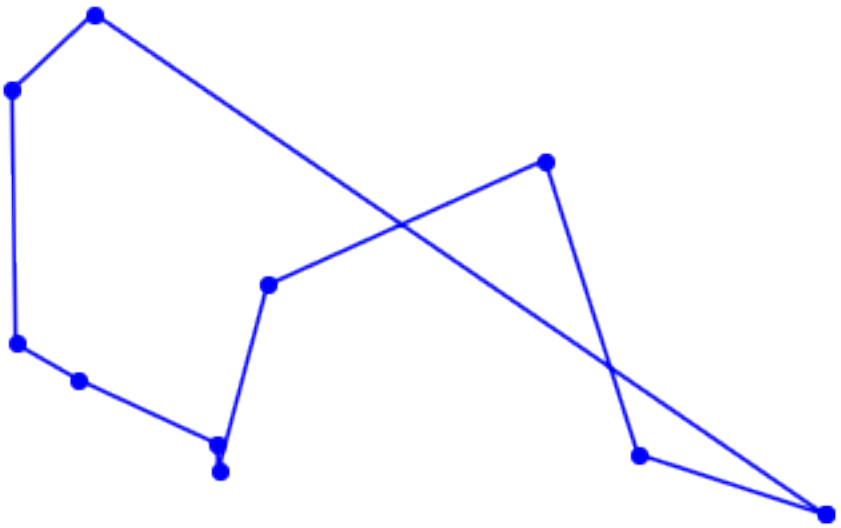
```
In [8]: def greedy_search(cities):
            edges = shortest_edges_first(cities) # A list of (A, B) pairs
            endpoints = {c: [c] for c in cities} # A dict of {endpoint: segment}
            for (A, B) in edges:
                if A in endpoints and B in endpoints and endpoints[A] != endpoints[B]:
                    new_segment = join_endpoints(endpoints, A, B)
                    if len(new_segment) == len(cities):
                        return new_segment

In [9]: def shortest_edges_first(cities):
            edges = [(A, B) for A in cities for B in cities
                            if id(A) < id(B)]
            return sorted(edges, key=lambda edge: distance(*edge))

In [10]: def join_endpoints(endpoints, A, B):
             Asegment, Bsegment = endpoints[A], endpoints[B]
             if Asegment[-1] is not A: Asegment.reverse()
             if Bsegment[0] is not B: Bsegment.reverse()
             Asegment.extend(Bsegment)
             del endpoints[A], endpoints[B]
             endpoints[Asegment[0]] = endpoints[Asegment[-1]] = Asegment
             return Asegment
```

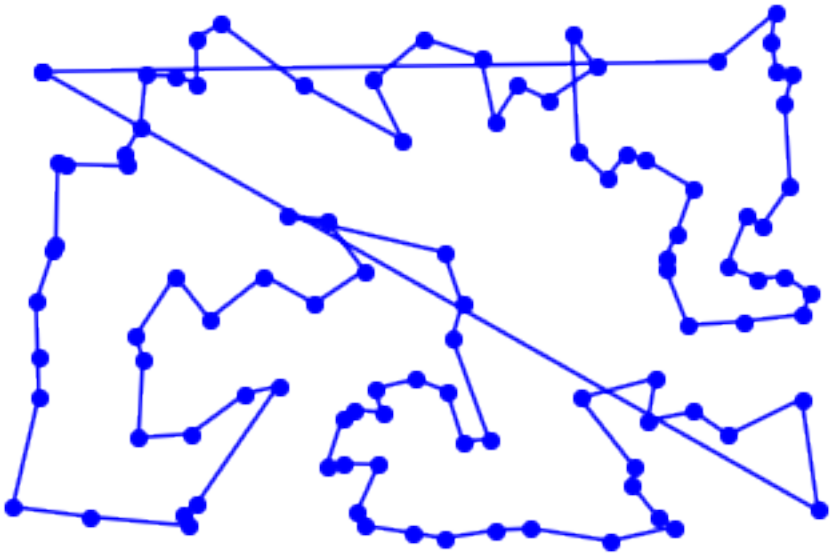
Let us plot this data to see it visually and let's consider a number of cities equal to 10.

```
In [11]: plot_tsp(greedy_search, Cities(10))
```



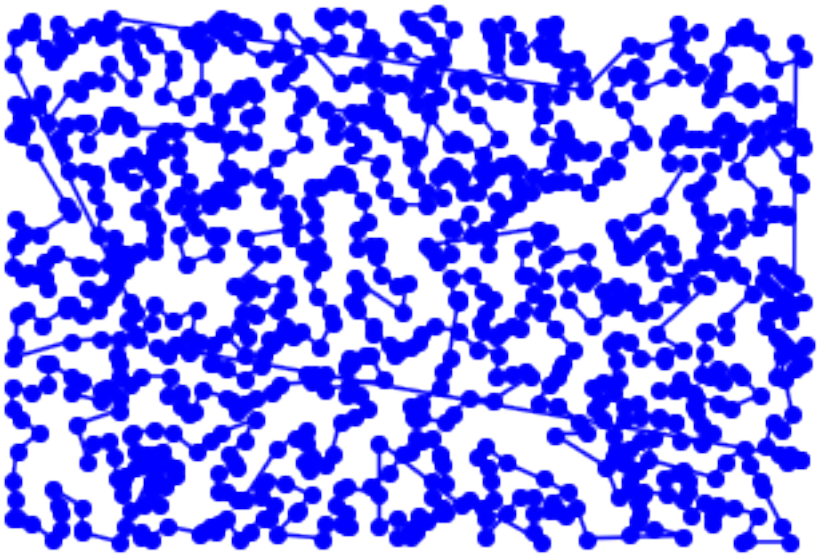
```
10 city tour with length 2700.6 in 0.000 secs for greedy_search
```

```
In [12]: plot_tsp(greedy_search, Cities(100))
```



```
100 city tour with length 7277.9 in 0.013 secs for greedy_search
```

```
In [13]: plot_tsp(greedy_search, Cities(1000))
```



```
1000 city tour with length 20405.2 in 1.432 secs for greedy_search
```

5.2.2 Testing:

```
In [50]: from time      import time
         pnts_n = []
         pnts_t = []
         def time_Greedy():
             print("Cycle - Greedy search" )
             number_cycles = 10
             n = 1000
             cycle = 1
             t0 = t1 = 0
             sum_pnts = 0
             while cycle <= number_cycles:
                 t0 = time()
                 greedy_search(Cities(n))
                 t1 = time()
                 # record time
                 print( str(cycle)+"\t"+str(round(t1-t0,2) ))
                 pnts_n.append( cycle )
                 pnts_t.append( round(t1-t0,2) )
                 cycle += 1
             i = 0
             for i in pnts_t:
```

```
            sum_pnts = (sum_pnts + i)
        average_time = sum_pnts / number_cycles
        print("Average time: " + str(average_time) + " seconds")
```

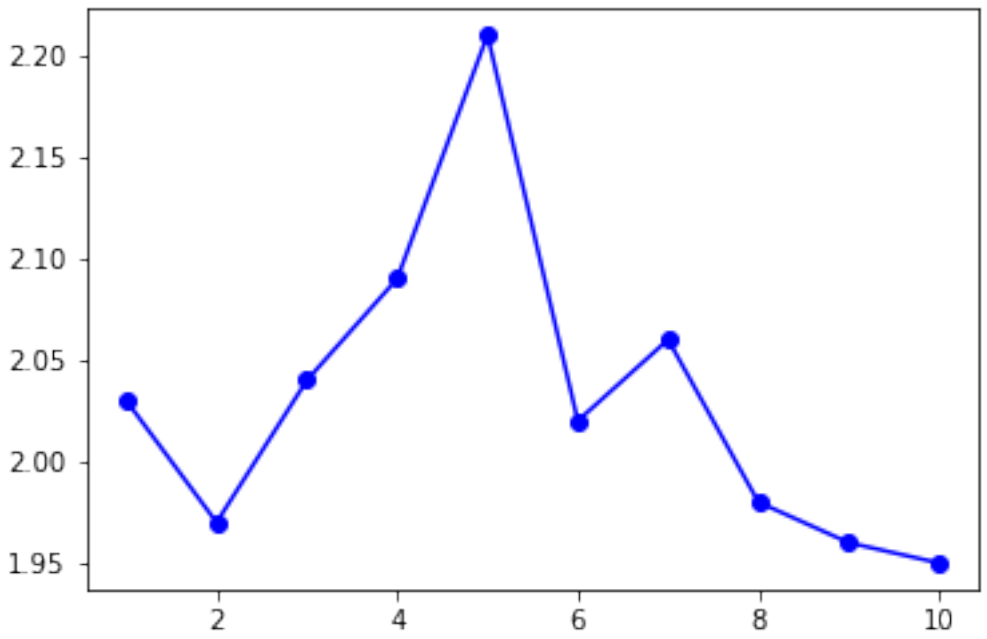
```
In [51]: time_Greedy()

Cycle - Greedy search
1         2.14
2         2.05
3         2.0
4         2.04
5         2.03
6         2.02
7         1.98
8         2.08
9         1.96
10         1.96
Average time: 2.026 seconds
```

From the function above, we can estimate the average time of 10 cycles, which is roughly equal to **2.02 seconds**; also for this test, the input number of cities is 1000. ($n = 1000$)

To enhance the visual representation, let us plot the data:

```
In [47]: plt.plot(pnts_n, pnts_t, 'bo-')
         plt.show()
```



As we can see from the graph, within each cycle the greedy algorithm is not entirely uniform; the reason of this is because we call each time a new sample of random cities. Furthermore, the algorithm's accuracy can be determined from the average time taken to execute 10 cycles.

The accuracy of each cycle can be represented as such:

$$Accuracy = (cycle\ time \times 100) \div Average\_Time\_Number\_Cycles$$

```
In [52]: average_time = 2.02
         accuracy_list = []
         def accuracy():
             for i in pnts_t:
                 accuracy = (i *100)/average_time
                 accuracy_list.append(accuracy)
             max_accuracy = max(accuracy, *accuracy_list)
             min_accuracy = min(accuracy, *accuracy_list)
             overall_accuracy = max_accuracy - min_accuracy
             print("Overall accuracy: " + str(overall_accuracy))
         accuracy()
```

```
Overall accuracy: 8.91089108911
```

As we can see from cycle to cycle the accuracy tends to be +/- **9%**

6 Genetic algorithm - Code and Testing:

6.0.1 Code

In the methodology, we defined the random sampling model, as cities with ($x = real$ x $y = imaginary$) coordinates. But for this case, we will be using float numbers instead of complex numbers. The logic's sample remains the same, but with some slight changes.

The class **City** deals with all the related methods. Each node(city) has a pair of coorinates (i, j).

```
In [25]: class City:
             def __init__(self, x,y):
                 self.x  = x
                 self.y  = y

             #create a function that calculates the distance of a city
             def distance(self, city):
                 xDistance = abs(self.x - city.x)
                 yDistance = abs(self.y - city.y)
                 distance = np.sqrt((xDistance ** 2) + (yDistance ** 2 ) )
                 return distance

             #create a function that display the city
             def __repr__(self):
                 return "(" + str(self.x) + "," + str(self.y) + ")"
```

The following function `Fitness` calculates the fitness of each route.

```
In [26]: class Fitness:
             def __init__(self, route):
                 self.route = route
                 self.distance = 0
                 self.fitness = 0.0

             def routeDistance(self):
                 if self.distance  == 0:
                     path = 0
                     for i in range(0, len(self.route)):
                         currentCity = self.route[i]
                         followCity = None
                         if i + 1 < len(self.route):
                             followCity = self.route[i+1]
                         else:
                             followCity = self.route[0]
                         path = path + currentCity.distance(followCity)
                     self.distance = path
                 return self.distance

             def routeFitness(self):
                 if self.fitness == 0:
                     self.fitness = 1/float(self.routeDistance())
                 return self.fitness
```

By following the Genetic Algorithm's Pseudo-code, we instantiate a random route sample.

```
In [27]: #random route sample
         def createRandomRoute(ListOfCities):
             route = random.sample(ListOfCities, len(ListOfCities))
             return route
```

Create the initial population:

```
In [28]: def initialPopulation(popSize, ListOfCities):
             population = []
             for i in range(0, popSize):
                 #in order to have a population i need to create a function that creates a rou
                 population.append(createRandomRoute(ListOfCities))
             return population
```

Rank individuals by fitness:

```
In [29]: def rankIndividuals(population):
             fitnessResults = {}
             for i in range(0, len(population)):
                 fitnessResults[i] = Fitness(population[i]).routeFitness()
             return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = Tru
```

Based on the fittest selection we determine the elites and the individuals for the *matingPool* function.

```
In [30]: def selection(popRanked, eliteSize):
             selectionResults = []
             df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])
             df['cum_sum'] = df.Fitness.cumsum()
             df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

             for i in range(0, eliteSize):
                 selectionResults.append(popRanked[i][0])

             for i in range(0, len(popRanked)-eliteSize):
                 rng = 100*random.random()
                 for i in range(0, len(popRanked)):
                     if rng <= df.iat[i,3]:
                         selectionResults.append(popRanked[i][0])
                         break
             return selectionResults
```

`matingPool` based on the selection results:

```
In [31]: def matingPool(population, selectionResults):
             matingPool = []
             for i in range(0, len(selectionResults)):
                 index = selectionResults[i]
                 matingPool.append(population[index])
             return matingPool
```

The genetic algorithm requires a helper function that helps generations to breed; the function takes two parents parameters as its input(parent 1/parent 2), and the gene selection for children is totally random.

```
In [32]: def breed(parent1, parent2):
             child = []
             childP1 = []
             childP2 = []

             geneA = int(random.random() * len(parent1))
             geneB = int(random.random() * len(parent1))

             startGene = min(geneA, geneB)
             endGene = max(geneA, geneB)

             for i in range(startGene, endGene):
                 childP1.append(parent1[i])

             childP2 = [item for item in parent2 if item not in childP1]
             child = childP1 + childP2
             return child
```

The following function breeds the entire population:

```
In [33]: def breedPop(matingPool, eliteSize):
             children = []
             length = len(matingPool) - eliteSize
             pool = random.sample(matingPool, len(matingPool))

             for i in range(0, eliteSize):
                 children.append(matingPool[i])

             for i in range(0, length):
                 child = breed(pool[i], pool[len(matingPool)-i-1])
                 children.append(child)
             return children
```

Final step: mutation. The purpose of this function is to create variation in the population, also considered as a helper function. It mutates one individual at a time;

```
In [34]: def mutate(individual, mutateRate):
             for swapped in range(len(individual)):
                 if(random.random() < mutateRate):
                     swapWith = int(random.random() * len(individual))

                     city1 = individual[swapped]
                     city2 = individual[swapWith]

                     individual[swapped] = city2
                     individual[swapWith] = city1

             return individual
```

Mutate each individual(the mutation rate - 0.01); after that mutate the entire population.

```
In [35]: def mutatePop(population, mutateRate):
             mutatedPop = []

             for i in range(0, len(population)):
                 toMutate =  mutate(population[i], mutateRate)
                 mutatedPop.append(toMutate)
             return mutatedPop
```

Instantiate the next generation:

```
In [36]: def nextGeneration(currentGen, eliteSize, mutateRate):
             popRanked = rankIndividuals(currentGen)
             selectionResults = selection(popRanked, eliteSize)
             matingpool = matingPool(currentGen, selectionResults)
             children = breedPop(matingpool, eliteSize)
             nextGeneration = mutatePop(children, mutateRate)
             return nextGeneration
```

All the components are implemented, so the last aspect to do is to run the Genetic ALgorithm and to calculate the best route based on the number of generations:

```
In [37]: def geneticAlgorithm(population, popSize, eliteSize, mutateRate, generations):

             start = time.time() # calculate the start time of execution
             pop = initialPopulation(popSize, population)

             initialDistance = 1 / rankIndividuals(pop)[0][1]

             print("Initial distance: " + str(initialDistance))

             for i in range(0, generations):
                 pop = nextGeneration(pop, eliteSize, mutateRate)

             finalDistance = 1 / rankIndividuals(pop)[0][1]

             print("Final distance: " + str(finalDistance) + " after " + str(generations) +" g

             bestRouteIndex = rankIndividuals(pop)[0][0]
             bestRoute = pop[bestRouteIndex]
             end = time.time()
             print("For " + str(len(population)) + " cities randomly generated it takes: " + s
             improvement =  initialDistance - finalDistance
             percentImprovement = improvement / initialDistance * 100
             print("After " + str(generations) + " generations, the algorithm has a " + str(pe
             return bestRoute
```

Le'ts run the algorithm with a number of 20 cities:

```
In [39]: cityList = []
         for i in range(0, 20):
             cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))

         geneticAlgorithm(population = cityList, popSize = 100, eliteSize = 20, mutateRate = 0

Initial distance: 1666.56691659
Final distance: 744.715604139 after 500 generations

For 20 cities randomly generated it takes: 26.0669999123 seconds using Genetic Algorithm

After 500 generations, the algorithm has a 55.3143893157%  improve rate


Out[39]: [(3,26),
          (50,13),
          (71,16),
          (124,27),
          (198,111),
```

```
          (199,121),
          (162,125),
          (129,144),
          (110,133),
          (103,197),
          (72,138),
          (66,127),
          (29,160),
          (21,156),
          (34,139),
          (30,113),
          (1,85),
          (1,61),
          (29,70),
          (27,40)]
```

6.0.2 Testing:

Let's determine the time taken and the improving rate, when we run a number of 15 cities (GA):

```
In [43]: cityList = []
         for i in range(0, 15):
             cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))

         geneticAlgorithm(population = cityList, popSize = 100, eliteSize = 20, mutateRate = 0

Initial distance: 1242.08104195
Final distance: 767.836763672 after 500 generations

For 15 cities randomly generated it takes: 23.1099998951 seconds using Genetic Algorithm

After 500 generations, the algorithm has a 38.1814279633%  improve rate


Out[43]: [(199,99),
          (159,49),
          (112,1),
          (103,15),
          (5,25),
          (18,151),
          (20,187),
          (38,172),
          (80,198),
          (76,168),
          (62,158),
          (55,116),
          (120,95),
          (138,77),
          (187,128)]
```

Based on the previous test, we can observe that the `improve rate` on the GA algorithm is directly proportional to the number of cities. In other words, while n is increasing, the improved rate will increase too.

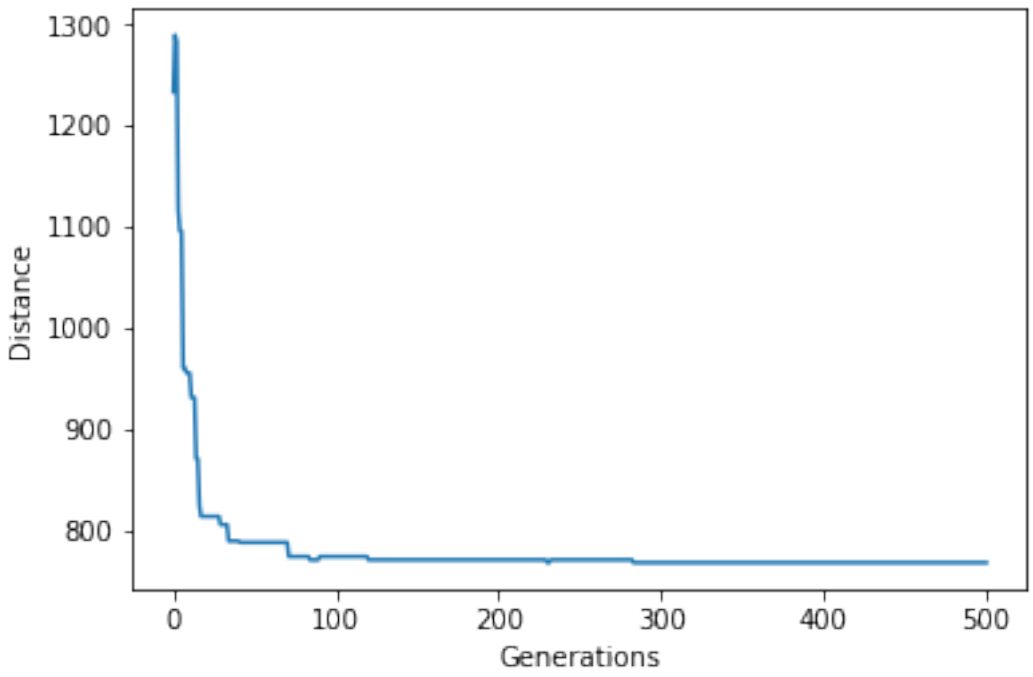
The next plot will display how the distance improved over generations:

```
In [44]: def execute_accuracy_plot(population, popSize, eliteSize, mutationRate, generations):
             pop = initialPopulation(popSize, population)
             progress = []
             progress.append(1 / rankIndividuals(pop)[0][1])

             for i in range(0, generations):
                 pop = nextGeneration(pop, eliteSize, mutationRate)
                 progress.append(1 / rankIndividuals(pop)[0][1])

             plt.plot(progress)
             plt.xlabel('Generations')
             plt.ylabel("Distance")
             plt.show()

         execute_accuracy_plot(population = cityList, popSize = 100, eliteSize=20, mutationRat
```



As we can observe, the GA algorithm reaches the minimum distance before 70^{th} generation. When n = 15, GA is already over-performing. Let's try one more test, for n = 25;

```
In [45]: cityList = []
         for i in range(0, 25):
```

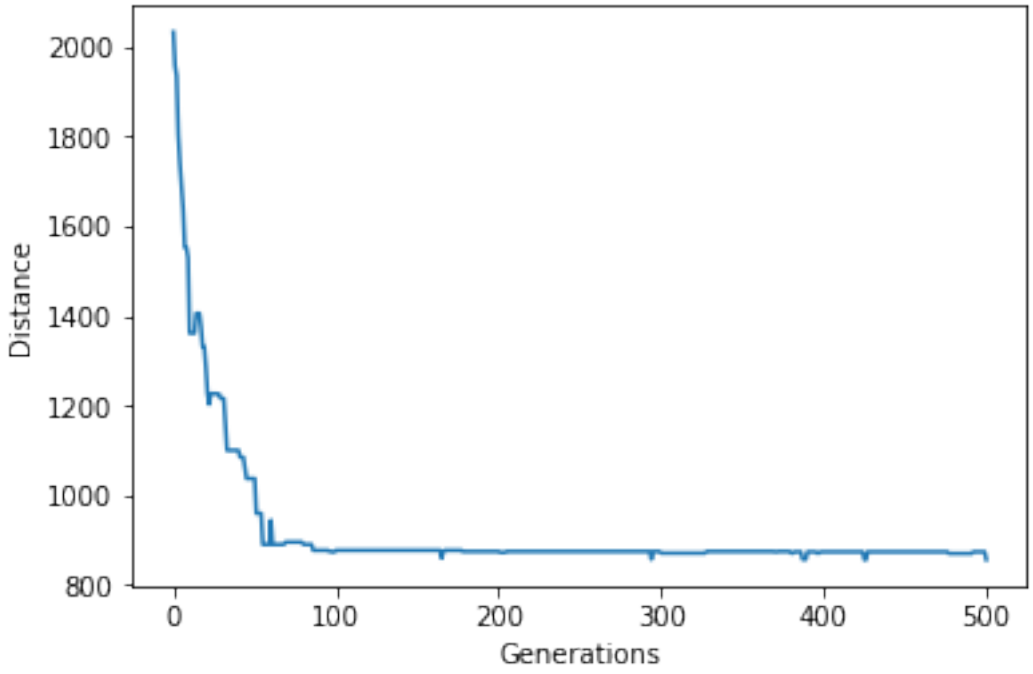
```
        cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))

    geneticAlgorithm(population = cityList, popSize = 100, eliteSize = 20, mutateRate = 0
    execute_accuracy_plot(population = cityList, popSize = 100, eliteSize=20, mutationRat
```

```
Initial distance: 2082.37425595
Final distance: 914.674011451 after 500 generations

For 25 cities randomly generated it takes: 33.6849999428 seconds using Genetic Algorithm

After 500 generations, the algorithm has a 56.0754264591%  improve rate
```



Based on the previous test, we can observe that when n = 25; it does not reach the minimum path until up close to 420^{th} generation.

As a conclusion, we can predict that for n>25 cities, 500 generations it won't be enough to satisfy the problem; and more generations will be required.

GA - Efficiency test: The followin function `improvement_test_GeneticAlgorithm` was created to determine the percentage value of improvement:

```
In [47]: def improvement_test_GeneticAlgorithm(population, popSize, eliteSize, mutateRate, gene
             start = time.time() # calculate the start time of execution
             pop = initialPopulation(popSize, population)
             initialDistance = 1 / rankIndividuals(pop)[0][1]
             for i in range(0, generations):
```

```
            pop = nextGeneration(pop, eliteSize, mutateRate)
        finalDistance = 1 / rankIndividuals(pop)[0][1]
        bestRouteIndex = rankIndividuals(pop)[0][0]
        bestRoute = pop[bestRouteIndex]
        end = time.time()
        improvement =  initialDistance - finalDistance
        percentImprovement = improvement / initialDistance * 100
        return percentImprovement
```

The function `time_test_GeneticAlgorithm` calculates the time taken:

```
In [55]: pnts_cities = []
         pnts_time = []
         pnts_cycles = []
         def time_test_GeneticAlgorithm():
             #test Genetic Algorithm with different number of cities and calculate the time
             print('GENETIC ALGORITHM - TIME/IMPROVEMENT EFFICIENCY TEST\n')
             print("Cities\tGeneticAlgorithm\tImprovement")
             n = 10
             t0 = t1 = 0
             while t1-t0 < 30:
                 t0 = time.time()
                 cityList = []
                 for i in range(0, n):
                     cityList.append(City(x = int(random.random() * 200), y = int(random.rando

                 percentImprovement = improvement_test_GeneticAlgorithm(cityList, popSize = 10
                 t1 = time.time()
                 print( str(n)+"\t"+str(t1-t0)+"\t        "+str(percentImprovement)+"%")
                 pnts_cities.append(n)
                 pnts_time.append(t1-t0)
                 n+=1
```

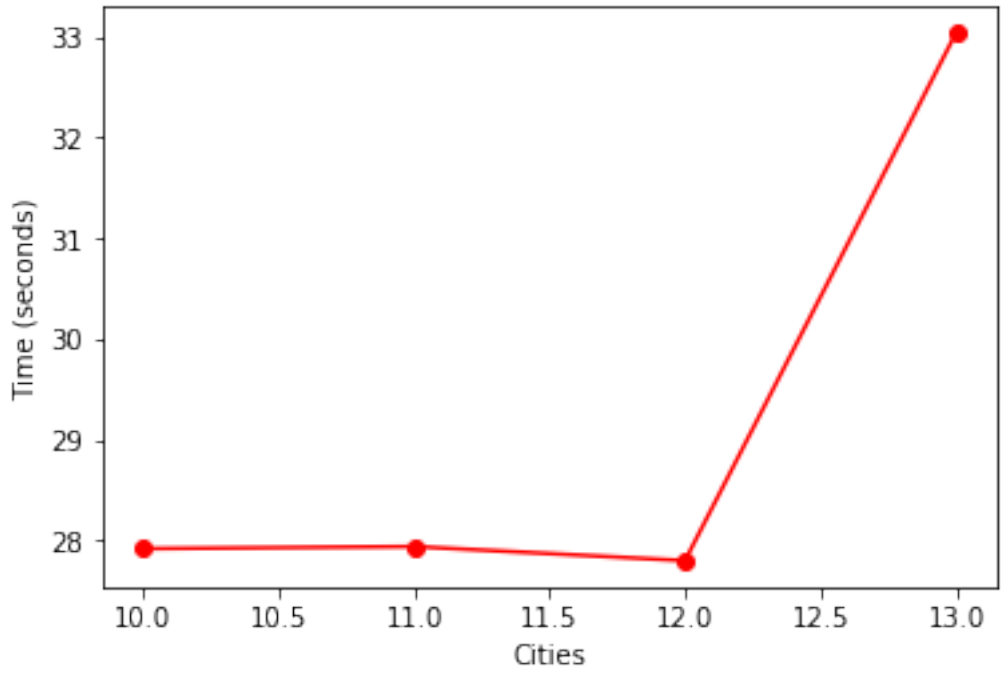
Let us visualise the plot's data:

```
In [56]: def execute_timetaken_test():
             time_test_GeneticAlgorithm()
             plt.plot(pnts_cities, pnts_time, 'ro-')
             plt.xlabel('Cities')
             plt.ylabel('Time (seconds)')
             plt.show()
         execute_timetaken_test()
```

```
GENETIC ALGORITHM  TIME/IMPROVEMENT EFFICIENCY TEST

Cities        GeneticAlgorithm        Improvement
10        27.9219999313               16.0866971582%
11        27.9430000782               30.4749514716%
12        27.8029999733               24.7343655502%
```

```
13          33.0379998684                   41.9278045398%
```



From this efficiency test, we can assume that the improvement rate is also increasinig with the number of cities.

My machine exceeds the processing time (program set to stop if the processing time is exceeding 30 seconds) which made the test to stop; but it can be seen that between n = 11 and n = 12 there is a slight drop; such a dropping rate may be interpreted as a machine overload(It can manifest when the machine is loading multiple files). On the other hand, when n is between 12-13 it can be also seen how the improving rate is increasing.

GA - Accuracy test: The same principle(to the greedy algorithm) is applied here; in order to determine the accuracy test. In addition, the function will return the average time taken over 10 cycles and also the average rate of imrovement.

```
In [59]: def average_timeImprovement_test_GeneticAlgorithm():
             #Test Genetic Algorithm with the same number of cities
             #This function test the average of time and improvement of Genetic Algorithm
             #10 cycles will be run on 10 cities
             print('GENETIC ALGORITHM AVERAGE TIME/IMPROVEMENT TEST\n')
             print("Cycle No.\tTime\tImprovement")
             cityList = []
             averageTimeArray = []
             averageImpArry = []
```

```
            n = 15
            for i in range(0, n):
                cityList.append(City(x = int(random.random() * 200), y = int(random.random() *

            cycles = initialCycles = 10
            t0 = t1 = 0
            sumTime = sumImp = 0
            while cycles >= 0:
                t0 = time.time()
                percentImprovement = improvement_test_GeneticAlgorithm(cityList, popSize = 100
                t1 = time.time()
                print( str(cycles) + "\t        " + str(round(t1-t0, 2)) + "\t " + str(round(p
                pnts_cycles.append(cycles)
                pnts_time.append(t1-t0)
                averageTimeArray.append(round(t1-t0, 2))
                averageImpArry.append(round(percentImprovement, 2))
                cycles -=1

            for i in averageTimeArray:
                sumTime = sumTime + i

            for j in averageImpArry:
                sumImp = sumImp + j

            averageTime  = sumTime / len(averageTimeArray)
            averageImprovement = sumImp / len(averageImpArry)
            print("\n")
            print("After " + str(initialCycles) +" cycles" + ",\n the algorithm which had an i
            if averageImprovement == 0:
                print("\n The number of cities is too low so there is no room for improvement"
```

Let's run the code; 10 cycles - number of cities equal to 15;

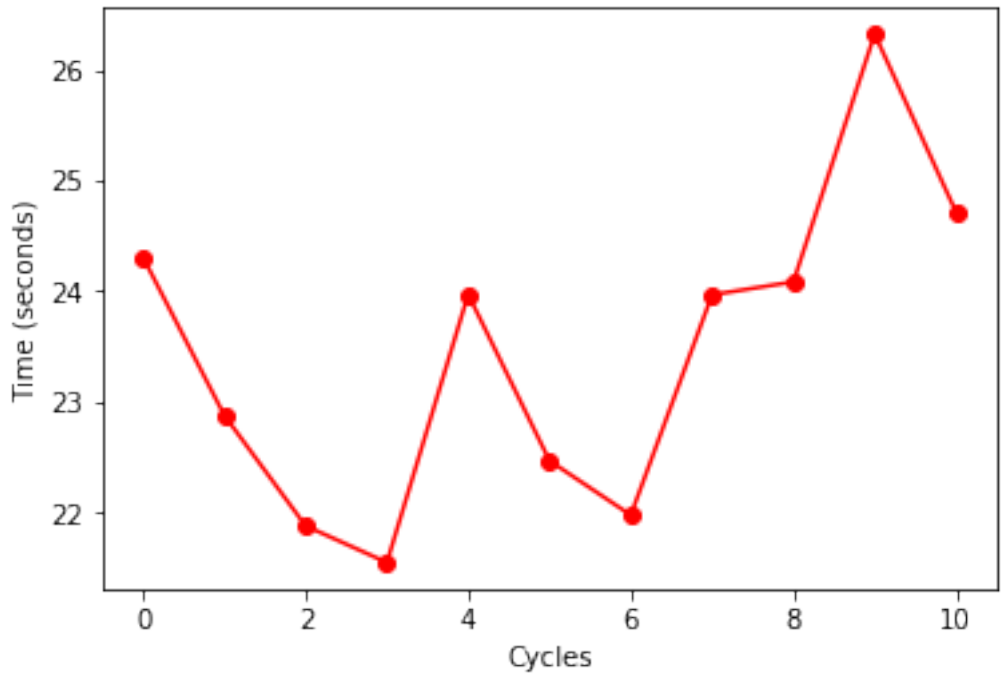
```
In [60]: pnts_cities = []
         pnts_time = []
         pnts_cycles = []
         def execute_average_timeImprovement_test():
             average_timeImprovement_test_GeneticAlgorithm()
             plt.plot(pnts_cycles, pnts_time, 'ro-')
             plt.xlabel("Cycles")
             plt.ylabel("Time (seconds)")
             plt.show()
         execute_average_timeImprovement_test()

GENETIC ALGORITHM AVERAGE TIME/IMPROVEMENT TEST

Cycle No.        Time        Improvement
10                24.72         45.55 %
```

```
9                26.32          38.76 %
8                24.08          44.68 %
7                23.97          43.59 %
6                21.97          40.49 %
5                22.47          27.83 %
4                23.97          45.74 %
3                21.55          36.88 %
2                21.88          44.91 %
1                22.88          34.73 %
0                24.31          39.73 %
```

```
After 10 cycles,
 the algorithm which had an input population of 15 cities,
 has an average time of: 23.4654545455
 And an averange Improvement rate of: 40.2627272727 %
```



As the last experiment, it can be mentioned that even though we ran the same number of cities over a 10 cycle loop; the experimental results concluded that: > the improvement rate is not 'uniform'(constant). Therefore, it can be said that the `Genetic Algorithm` is not entirely the favourable algorithm to determine the best solution(route).

7 Conclusion:

Based on the experimental results we can state the effectiveness of each algorithm: > **Exhaustive-search**: Despite the fact that execution time is not favorable, we can say with ease that it is one of the only algorithms that find the best route. The negative side of this algorithm is reflected when the number of cities exceeds n> 10.

> **Greedy Search**: Of all the algorithms studied, we can conclude that the greedy search algorithm is the best in terms of execution. But, we can accurately say that this algorithm does not analyze the entire set of data. Instead, this algorithm checks the distance between two points and unites them. If you look at the issue in terms of the entire set of data, it can be said that it does not fully meet the problem's requirement.
>
> **Genetic Algorithm**: Just like the greedy search algorithm has its negative parts. During the tests, it is noted that if the number of cities increases, it is recommended to increase the number of generations as well; in order to achieve a better percentile of improvement.

8 Reflection Section

Although we have formed a relatively small team of two members, we can say that we have been able to achieve most of the evaluation points.

As a team, we agreed that each member should choose an algorithm at his choice and to implement it. Although I have the necessary knowledge and experience to create a genetic algorithm, it has been decided that I will do the greedy search algorithm.

From this assessment, I can easily say that I have assimilated knowledge about how to use notations and how to plot in python. From my point of view, this course was exciting because we applied both mathematical and computer skills to a real problem. In addition, this project was written in Jupyter notebook, which allowed me to discover its advantages. Jupyter Notebook is an efficient tool which facilitates rapid-prototyping development and moreover, facilitates the easy collaborations between members.

Difficulties - as a team:

As a team, we encountered some difficulties; from a code-structural point of view, we would have liked to apply the graph's plot functions from the same function, without having to repeat the process, but unfortunately, the time was not on our favour and we did not manage to solve that issue. The second difficulty - is addressing mostly to me; at the beginning of the project I was not really familiar with the Jupyter's notebook syntax, but over time it actually becomes enjoyable.

In the near future, we would love to reproduce the "test section", in the form of a `benchmark` function - that integrates all the test conditions (efficiency, accuracy and execution time) in one place without having to scroll up and down to search for functions.

9 References

Brilliant.org, n.d. Greedy Algorithms. [Online] Available at: https://brilliant.org/wiki/greedy-algorithm/ [Accessed 4 April 2019].

Kingsford, n.d. CMSC 451: SAT, Coloring, HamiltonianCycle, TSP. [Online] Available at: https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/sat.pdf [Accessed 3 April 2019].

Mallawaarachchi, 2017. Introduction to Genetic Algorithms. [Online] Available at: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 [Accessed 3 April 2019].

Mathworks.com, n.d. What Is the Genetic Algorithm?. [Online] Available at: https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html [Accessed 3 April 2019].