Simone Quadrelli

Nature-inspired Monte Carlo algorithm for travelling salesman problem

Academic year 2019 - 2020

Contents

List of Figures

Abstract. The aim of this project is to solve the travelling salesman problem (TSP) which consists in finding the shortest hamiltonian cycle (i.e. cycle that pass through each vertex of a graph just once) among all the vertices of a graph.
The problem is a *NP-hard* problem and therefore there exist no feasible algorithm to solve it exactly for any possible input. Therefore, the project explores the possibility to exploit simulated annealing and a genetic algorithm to compute with adequate approximation the shortest hamiltonian cycle.
The report shows the result of the simulations computed on some cities of northern Italy and compares the exact feasible solutions with the approximated ones. It also provides an analysis of the computational time needed to obtain an exact optimal solution and to obtain an approximately optimal solution.

1 Introduction

The travelling salesman problem (TSP) consists in finding the shortest route a travelling salesman has to take to visit all the city only once and to return to the starting point. The route described is called *hamiltonian cycle* in graph theory (i.e. a cycle that pass through each vertex of a graph just once), indeed the oldest and more technical formulation of the problem was proposed by Hamilton. An algorithm that provides a solution to the travelling salesman problem can be exploited in a wide variety of applications: indeed it can be used in logistics to optimize the path of vehicles and therefore to reduce costs. The solutions that exploit euclidean distances can can easily be applied to schedule drones shipping in cities without too high skyscarpers and buildings. It is possible to model the problem usign an undirected, weighted and fully connected graph [1] $G = (N, E)$, where N is the set of the cities to visit and $E = N \times N$ is the weight of the path. In this particular instance of the problem the weights correspond to the euclidean distance between each couple of cities. Formally, an hamiltonian cycle can be defined as a sequence $(n_0, .., n_{k-1})$ of length k such that $n_i \neq n_j \ \forall i, j \in \{1, ..., |N|\}$.

1.1 Hardness of the travelling salesman problem

Solving the travelling salesman problem pose very tricky computational problems. Indeed, the most intuitive solution to the problem consist in computing all the possible paths and then evaluate all their lengths to find the shortest. But the number of possible cycles is $\frac{(n-1)!}{2}$ and grows faster that 2^n (i.e. $2^n = o(n!)$) and thus the problem cannot be solved in finite time for large n. Morover, even if it was possible to generate each hamiltonian cycle in constant time independently from their length, the algorithm will have an exponential time complexity to find the best result.
More techically, TSP is an *NP-hard* problem, meaning that it is at least as hard as the hardest NP-complete problem. *NP-complete* problems are problems that can be solved in polynomial time by a non-deterministic algorithm. For such a class of problem there exists no polynomial algorithm that can solve the problems and may not exist, still noone was able to prove that such algorithms does not exists.

[1] For an hamiltinian cycle to exist the graph must be at least connected

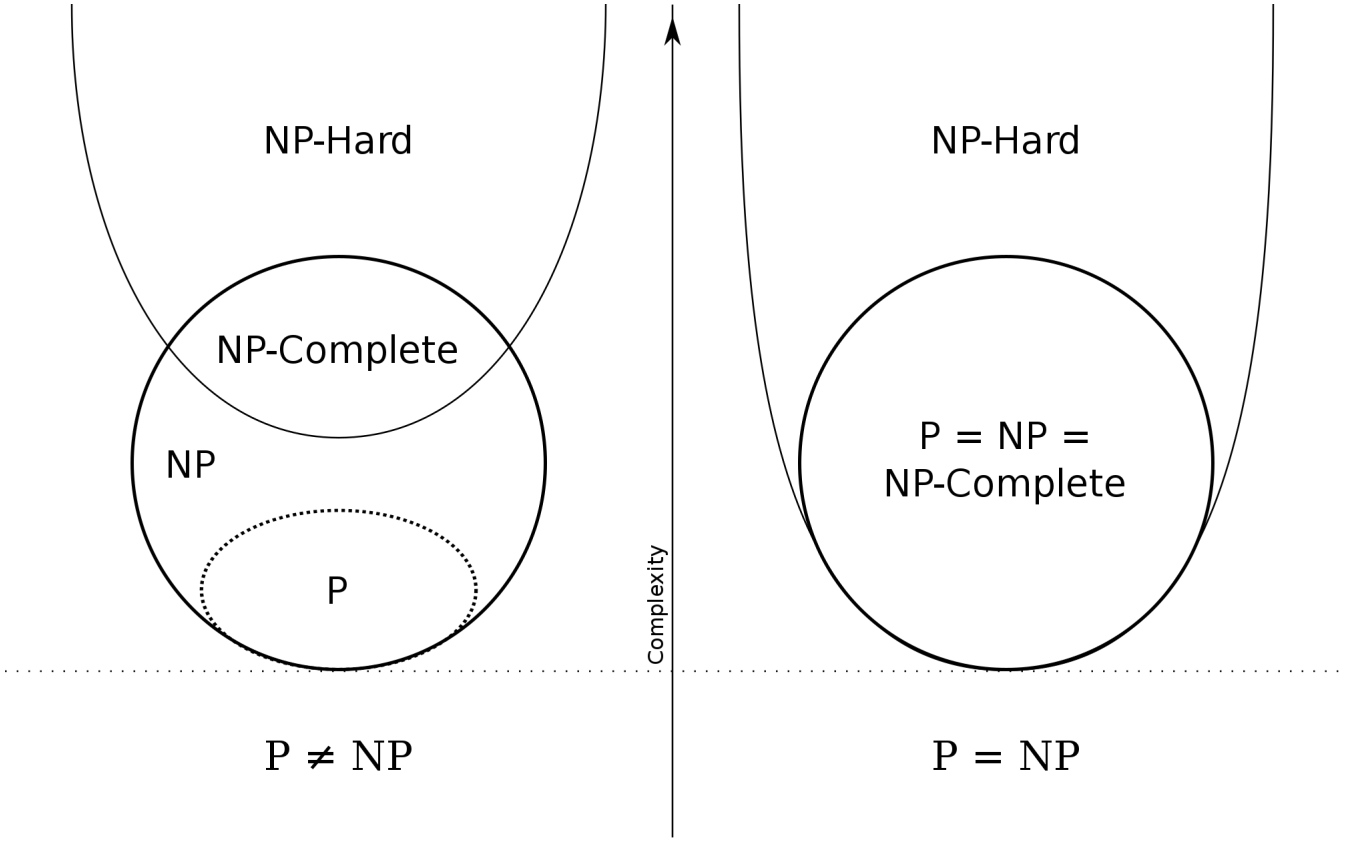

Figure 1: Complexity schema

Bearing in mind the complexity problem to face, the most reasonable solution is to solve the problem approximately. The technique I used to generate the possible hamiltonian cycles is a genetic algorithm, while the choice of the shortest hamiltonian cycle is done by the *simulated annealing* algorithm.

2 Literature review

There are many processes seeking a stable state in nature and they can be seen as natural optimization processes. Many efforts were done to find out global optimization techniques that simulate natural optimization. There can be depicted three main class of optimization algorithms

1. Simulated Annealing
2. Artificial Neural Networks
3. Evolutionary Algorithms

Among the evelutionary algorithms can be found different subsclasses of algorithms such as evolutionary programming (1962), evolution strategies (1965) and genetic algorithms (1975) which

exploit in different ways mimic Darwin's theory. Overall, evolutionary algorithms are stochastic search algorithms simulating the natural evolution processes. [5]
It was discovered that such algorithms are suitable optimization techniques for NP-hard problems. Not surprisingly they are very fit for biological pharmaceutical and chemistry application.
On the other hand, Artificial Neural Networks were exploited only recently since they require an enormous amount of data and a huge computational power. Those algorithms does not provide a strightforward interpretation and therefore may not be suitable in industrial logistic applications. [6]

3 Algorithms and techniques

In this section we provide a brief overview of all the techniques and aglorithms exploited. They are: Simualted Annealing, Metropolis algorithm and Genetic algorithm.

3.1 Simulated annealing

Simulated annealing (SA) is a technique that mimic the behaviour of heated materials slowly cooled down in metallurgy, this process decreases the impurity and increases the size of the crystals meaning that allows materials to reach the state that minimizes the energy. It is a technique to approximate the global optimal solution when the solutions is to be searched in an enormous space. SA is a metaheuristic meaning that it is not a probelm-specific stochastic heuristic. As every metaheuristic algorithm it relies on two principles *intensitification* and *diversification*, indeed, the algorithm intensifies the search near different possible optimal solutions. SA is very useful since it does not require any prior information, other than the cost, indeed it just supposes that states (hamiltonian cycles in our case) are distributed accordingly to the Bolzmann distribution:

$$e^{-\beta(cost(state))} \tag{1}$$

The Boltzmann distribution is suitable to model the TSP problem since it is most unbiased distribution among all the possible distributions. The undiasedness derives from the maximization of the Shannon entropy, given the cost. [METTI UNA CITAZIONE]
The underlying assumption of simulated annealing is that the optimal states are the states with the lowest cost, and they are reached as the temperature decreases. he temperature T is the inverse of β ($T = \frac{1}{\beta}$), therefore we start with low values of β (high temperature) and as the simulation proceeds we increase the values of β, thus mimicking the decrease in temperature. The only other information needed is the cost function. For this specific application the cost function is the sum of euclidean distances between the cities in the cycle. The coordinate of a city (x, y) are expressed in longitude and latitude.
In the following we provide a schematic overview of the simulated annealing implemented.

```
function SIMULATED ANNEALING(betas,number of simulations)
    s ← generate starting state with the genetic algorithm
    for b in betas do
        for n in 1,.., number of simulations do
            ns ← generate a new state with the genetic algorithm starting from s
            ns ← METROPOLIS(s,ns,β)
        end for
    end for
    return ns
end function
```

where **betas** is a set of the values of β.
The code executes all the procedures using a vectorial implementation to speed up the computation.

3.2 Metropolis algorithm

Metropolis is a Markov Chain Monte Carlo (MCMC) algorithm used to generate values X_n from a finite set of values S accordingly to a target probability distribution π.
An homogeneous and finite Markov chain with state space S, transition matrix P and initial distribution μ is a sequence of random variables $\{X_n\}_{n\in N}$ such that

1. $\forall i \in S \, Pr(X_0 = i) = \mu(i)$
2. $\forall n > 0 \; \forall i_n, ..., i_0 \, Pr(X_{n+1} = j | X_n = i_n, ..., X_0 = i_0) = Pr(X_{n+1} = j | X_n = i)$
3. $\forall n > 0 \, Pr(X_{n+1} = j | X_n = i) = p(i, j)$

Metropolis algorithm is designed in such a way that it is very likely to move from a state with lower probability from a state with higher probability.
The following algorithm describes the metropolis algorithm when the target distribution is $\pi(\text{state}) = e^{-\beta \, \text{cost(state)}}$ and S is the set of all possible hamiltonian cycles having fixed their length. In the following we suppose to have generated two hamiltonian cycles (i.e states) old_state, new_state and we want to understand if new_state can come from π. If new_state is likely to be a sample of π we use it as new state otherwise we keep old_state as the current state.

```
function METROPOLIS(old_state, new_state, β)
    if cost(new_state) < cost(new_state) then
        return new_state
    end if
    threshold ← unif[0,1]
    p ← min{1, e^{-β(cost(new_state)-cost(old_state))}}
    if p > threshold then
        return new_state
    else
        return old_state
    end if
end function
```

3.2.1 Properties of metropolis algorithm

We can model the generation of new hamiltonian cycles using a undirected and connected graph $G = (S, E)$ whose set of states S contains all the possible hamiltonian cycles and E contains the transition probability from an hamiltonian cycle to another one. Given the graph, we can define the transition matrix $P = p(i,j)]_{i,j \in S}$ as

$$p(i,j) = \begin{cases} 0 & \text{if } \{i,j\} \notin E \\ \frac{1}{d_i} \min\{1, \frac{\pi(j)}{\pi(i)}\} & \text{if } \{i,j\} \in E \ , \ i \neq j \\ 1 - \frac{1}{d_i} \sum_{l \in adj(i)} \min\{1, \frac{\pi(l)}{\pi(i)}\} & \text{if } i = j \end{cases} \tag{2}$$

Since G is a connected graph, the P is irriducible meaning that $\forall i, j \in S \, p^n(i,j) > 0$, where $p^n(i,j)$ means that exist a path from i to j of length n.
It can also be proved that P is aperiodic [1]. By aperidoicity we mean each node of the graph has a loop. More formally, th graph G is aperiodic if $\forall i \in S$, $d(i) = 1$ where

$$d(i) = mcd\{n \in N | \text{ exist a cycle from i to i of length } n\} \tag{3}$$

Metropolis algorithm defines a probability distributin π that is a reversible distribution, meaning that

$$\pi(i)p(i,j) = \pi(j)p(j,i) \ \ \forall i, j \in S \tag{4}$$

This is a key feature of the algorithm since a reversible distribution π is also a stationary distribution $\pi' P = \pi'$. This condition holds by the following theorem

Theorem 1. *If a probability distribution π is a reversible distribution for a markov chain $\{X_n\}$, then it is also a stationary distribution.*

Proof. $(\pi' P)_j = \sum_{i \in S} \pi(i)p(i,j) = \sum_{i \in S} \pi(j)p(j,i) = \pi(j)$ □

If this distribution was unique than we are certain to correctly generate the data accordingly to the target distribution. The stationary distibution is unique if the transition matrix is primitive as stated by the follwing theorem.

Theorem 2. *Let $\{X_n\}$ be a markov chain with a primitive (i.e. irreducible and aperiodic) transition matrix on a finite set, then there exist only one stationary distribution and it as also the limit distribution* $\lim_{n \to \infty} p^n(i,j) = \pi(j) \, \forall i \in S$

Since P is primitive, it is enough to compute the limit distribution to find the unique stationary distribution.
The stationary distribution can be approximated in a logarithmic number of iterations starting from any distribution. Therefore to sample hamiltonian cycle from the stationary distribution is enough to let the simulation go on for a suitable number of iterations [2], [3].

3.3 Genetic algorithm

Genetic algorithms are a widespread clarclengthss of algorithms that mimic nature. Indeed there are three fundamental operations they perform to generate new states (hamiltonian cycles in our

case), each operation is inspired by the theory of evolution by C. Darwin.
Among a specie, evolution selects the fittest individuals and therefore the algorithm will chose the fittest (i.e shortest) hamiltonian cycles.

```
1: function SELECTION(population,n_paths)
2:     n_parents ← unif[2,n_paths/2]
3:     order the population by fitness
4:     fittest_subpopulation ← extract n_parents in order from the sorted population
5:     return fittest_subpopulation
6: end function
```

The breeding of the fittest individuals produces new individuals that inherits traits from the parents, therefore in the crossbreeding phase we mix some subsection of the fittest hamiltonian cycles.

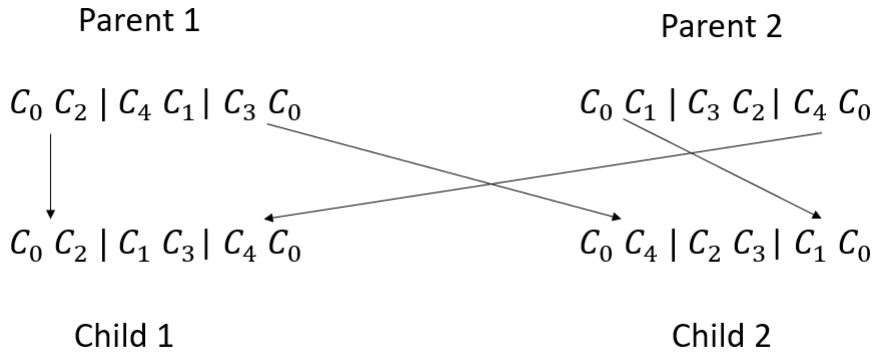

Figure 2: Crossbreeding schema

As random mutations can occur in nature but they are not so likely to occur, the algorithms swaps two cities with a fixed and low probability. The mutation probability should be low since otherwise if can disrupt the optimal subsequence of the cycle inherited.

```
function MUTATE(new_population, p)
    for path in new_population do
        test ← unif[0,1]
        if test < p then
            choose position1 randomly among unif[0,length of cycles-1)
            choose position2 randomly among unif[0,length of cycles-1)
            swap the cities in position1 and position2
        end if
    end for
    return new_population
end function
```

To easily apply all the operations, a function was created.

```
function EVOLVE(population)
    fittest_population ← SELECT(population)
    new_population ← CROSSBREED(fittest_population)
    population ← MUTATE(new_population)
end function
```

There are many possible implementations of the basic operations performed by a genetic algorithm. Therefore I deicde to implement the from scratch. The implementation of those functions is not very interesting and requires many implementative details, therefore we omit it in the present work.

4 Application

This section provides the reader with the necessary information about the choice of the distance function, the dataset and the precise algorithmical and python implementation of the key functions.

4.1 Distance

The first problem to overcome is to choose an appropriate notion of distance. When the construction of routes faces few natural constraints such as in the Po valley streets are built to minimize the distance between the cities they connect. The easiest way to minimize the distance is to use the euclidean distance since small portions of earth can be considered as a plain. Therefore to realistically model the routes of the highways that connect important cities in northern Italy euclidean distance seems to be a suitable measure.
As can be seen from Figure 3, the euclidean distance seems to be indeed a good approximation :

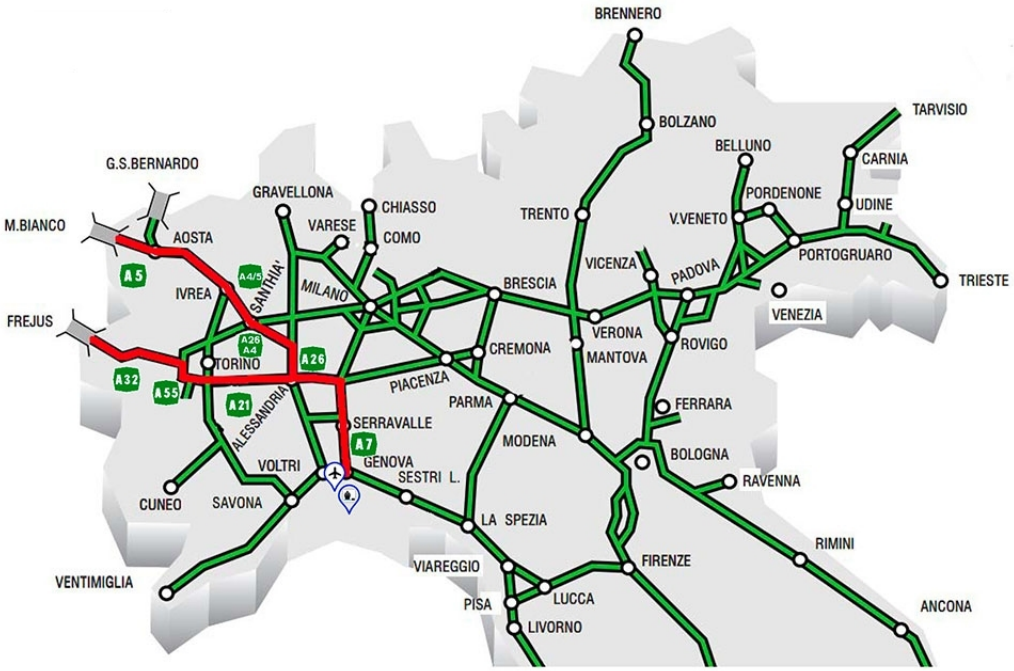

Figure 3: Scheme of the highways by *Autostrade per l'Italia*[8]

On the other hand, euclidean distance seems not to connect too well cities in the other italian regions like Tuscany and Emilia-Romagna since the Apennine Mountains prevent to connect the city on the west coast of central italy to cities on the east cost.

4.2 Dataset

Taking into account the constraint fixed by the choice of the distance the dataset used refers to cities in northern Italy: *Varese, Novara, Torino, Milano, Como, Bergamo, Brescia, Cremona, Lecco, Lodi, Mantova, Monza, Pavia, Sondrio, Verona, Vicenza, Padova, Venezia.*
The dataset consists in the coordinates of the cities above in terms of latitude and longitude. Since latitude and longitude is a global scale the cities are very close one another and therefore the cost of the hamiltonian cycles is very low.
The informations were downloaded using the python library *geopy* as follows:

```python
from geopy.geocoders import Nominatim
import pickle
geolocator = Nominatim(user_agent="python project")
cities = ['Varese', 'Novara', 'Torino', 'Genova','Milano', 'Como', 'Bergamo', 'Brescia', 'Cremona', 'Lecco',
          'Lodi', 'Mantova', 'Monza', 'Pavia', 'Sondrio','Verona', 'Vicenza', 'Padova','Venezia','Firenze',
          'Pisa']
x_coord = []
y_coord = []
for city in cities:
    location = geolocator.geocode(city)
    x_coord.append(location.longitude)
    y_coord.append(location.latitude)
```

Figure 4: Python code to download the coordinates of the cities

4.3 Software used

The objective of this simulation is to show compare the optimal solutions of the TSP computed by the appromated algorithm and the exact solution. However, as explained before, the exact solution of the TSP is not feasible when the number of cities grows too much therefore we compute exact solutions just as long as it is computationally feasible.
The algorithm select a growing set of cities each iteration, executes in parallel some predefined number of simulated annealing and select the best solution among the results of the simulated annealings. Then it computes, if feasible, the exact solution. The values are saved at each iteration and then are plotted to easily show the results.
The code was also instrumented to keep track of the execution time of both algorithms. The algorithm below provides a schematic description of the experiement run.

```
1:  function EXPERIMENT(all_cities)
2:      for i in 1,..., n_cities do
3:          cities ← all_cities[0:i]
4:          for for all threads do
5:              run simluted annealing in parallel
6:          end for
7:          extract the best solution among the solutions computed by the threads
8:          if i ¡ feasibility_threshold then
9:              compute the exact solution
10:         end if
11:         compare the results by plotting them
12:     end for
13: end function
```

The following python codes is the implementation of the algorithm described above. In the simulation run the starting and ending city is *Varese*

```
x_coord_matrix = []
y_coord_matrix = []
n_cities_matrix = []
for i in range(3,len(x_coord)):
    x_coord_matrix.append(x_coord[0:i])
    y_coord_matrix.append(y_coord[0:i])
    n_cities_matrix.append(i)

feasibility_threshold = 9
n_simulations = 3
n_paths = 20
mutation_probability = 0.01
betas = np.arange(1,40,1)
n_iterations = 20
approximate_costs = []
exact_costs = []
execution_times_approximated = []
execution_times_exact = []
for i in range(len(n_cities_matrix)):
    n_cities = n_cities_matrix[i]
    parameters = [n_cities,n_paths,mutation_probability,betas,n_iterations,
                  x_coord_matrix[i], y_coord_matrix[i]]
    best_paths = []
    pool = Pool()
    start_time = time.time()
    for _ in range(n_simulations):
        best_paths.append(pool.apply_async(run_simulation, parameters).get(timeout=20))
    execution_times_approximated.append(time.time() - start_time)
    if i < feasibility_threshold:
        start_time = time.time()
        exact_best_path, exact_cost =  best_exact_path(n_cities, distances)
        execution_times_exact.append(time.time() - start_time)
        exact_costs.append(exact_cost)
    best_path, cost= get_best_path(distances,best_paths,n_simulations,n_cities)
    approximate_costs.append(cost)
```

Figure 5: Python code of the experiment

The most importan function used in the code of the experiment is run_simulation that executes an instance of simulated annealing. The genetic algorithm generates a starting population which is a matrix (number of cycles × length of cycles - 2) then it evolves the whole population. In the end, if a cycle state is likely to come from the target distribution is accepted otherwise the evolution is rejected.

```
function RUN_SIMULATION(betas, n_iterations)
    generate a starting population
    for beta in betas do
        for i in 1,...,n_iterations do
            EVOLVE the population
            obtain the new state by metropolis algorithm
        end for
    end for
    return the shortest cycle
end function
```

The follwing code show the python implementation of the algorithm explained above. It is worth noting that the method acceptance is the implementation of metropolis algorithm.

```python
def run_simulation(n_cities,n_paths,mutation_probability,betas,n_iterations, x_coord, y_coord):
    ga = GeneticAlgorithm(n_cities,n_paths, mutation_probability)
    #set un the coordinates and the distanes between the cities
    ga.initialize_cities(x_coord,y_coord)
    # SIMULATED ANNEALING
    # number of itarations for each level of temperature (1/beta)
    for beta in betas:
        for i in range(n_iterations):
            cost_starting_state = ga.compute_fitness()
            starting_state = ga.population
            ga.evolve()
            cost_new_state = ga.compute_fitness()
            for i in range(ga.n_paths):
                # metropolis
                if not acceptance(cost_starting_state[i], cost_new_state[i], beta):
                    ga.population[i] = starting_state[i]
    #extract the best path
    return ga.best_path()
```

Figure 6: Python code of a simulation

A last remark, to be able to control and modify the code as wished, all the algorithm described were implemented from scratch exploiting just some support libraries as *numpy*.

5 Results

In this section we compare the simulated annealing with the exact algorithm to solve the TSP problem.
As expected the computational time required to compute an exact solutions becomes exponentially high as the length of the cycles increases. However, for small instances of the problem, it is more efficient that the simulated annealing since it perform faster and fewer operations. On the coutrary, the execution time of the simulated annealing is almost independent from the length of the cycles and remains very low. It is worthe notig that the slight variations and the peaks in the plot may be dued to the scheduling of the operating system and the multithreading.

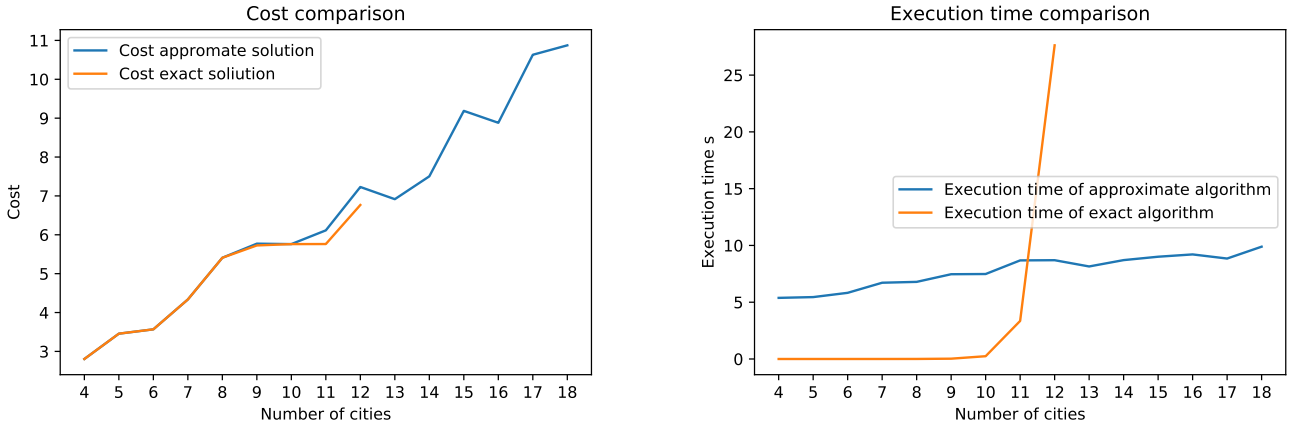

Figure 7: Comparison

As the plot above shows, simulated annealing is capable of finding optimal or slightly sub-optimal solutions. However, it is reasonable to expect that as the length of the cycles increases the optimal solutions of the simulated annealing move further away from the very best optimal solution. Since the algorithm is stochastic, can happen that longer cycles may have smoller cost than that of shorter cycles. For examplem, the cost of the cycle with 13 cities is lower than the cost of the cycle with 12 cities.
In the following we also plotted the shortest cycle computed by the simulated annealing when using all the possible cities. As mentioned before, the starting point is *Varese*.

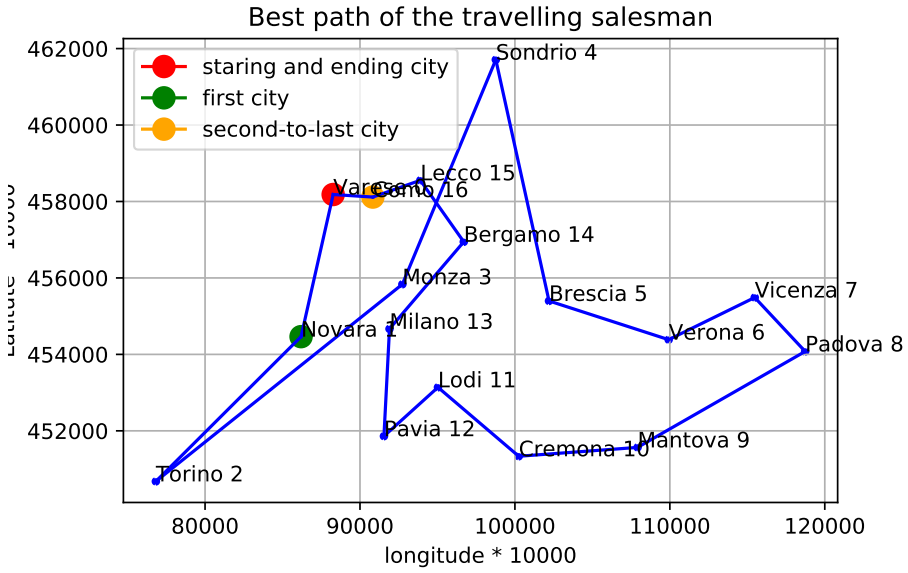

Figure 8: Optimal cycle starting from Varese

6 Conclusion

Overall, the simulated annealing and the genetic algorithm seem to have quite remarkable performance. The cost function is almost always increasing and as far as it is possible to compare the simulated annealing finds very optimal cycle while keeping the execution time very low.
As far as execution time is concerned, the length of the cycles seems to have a very limited influnce on the execution time, indeed, it exhibits a very sublinear behaviour.
As for the optimal cycle computed, for all of the cities the euclidean distance provides a suitable approximation of the routes of the highways present in northern Italy. However, this notion of distance does not take into account the travel of the highways that can significantly change during the day and may not be correlated to significantly with the length of the highways.

References

[1] Oliver Catoni *Simulated annealing algorithms and Markov chains with rare transitions*, Sminaire de probabilits (Strasbourg), tome 33 (1999), p. 69-119.

[2] Marius Iosifescu *Finite markov process and their applications*, John Wiley & Sons, 1980.

[3] M. Mitzenmacher, E. Upfal *Probability and Computing*, Cambridge university Press, 2005.

[4] Heinrich Braun *On solving the travelling salesman problems by genetic algoritms*

[5] P. Larranaga, C.M.H. Kuijpers, R.H. Murga, I. Inza and S. Dizdarevic *Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators*, Artificial Intelligence Review 13: 129170, 1999.

[6] C. Peterson *Parallel Distributed Approaches to Combinatorial Optimization: Benchmark Studies on Traveling Salesman Problem*

[7] https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/P_np_np-complete_np-hard.svg/2000px-P_np_np-complete_np-hard.svg.png

[8] http://www.autostrade.it/it/autostrade-per-genova?nomobile=true