Simone Quadrelli

Nature-inspired Monte Carlo algorithm for the 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. 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, as mentioned before, in finding the shortest route a travelling salesman has to take to visit all the cities only once and to return to the starting point. That route 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 sir W. R. Hamilton.
An algorithm that provides a solution to the travelling salesman problem can be exploited in a wide variety of applications: for instance, it can be used in logistics to optimize the path of vehicles and consequently to reduce costs. The solutions that exploit euclidean distances can easily be applied to schedule drone shipping in cities without too high skyscrapers and buildings.
Formally, it is possible to model the problem using 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 matrix of weights. In this particular instance of the problem, the weights correspond to the euclidean distance between each couple of cities.
Therefore, the route of the travelling salesman is an hamiltonian cycle defined as a sequence of nodes $(n_0, .., n_{k-1})$ of length k such that $n_i \neq n_j \ \forall i, j \in \{1, ..., |N| + 1\}$ and $n_0 = n_{|N|+1}$.

1.1 Complexity of the travelling salesman problem

Solving the travelling salesman problem pose very tricky computational problems. Indeed, the most intuitive solution to the problem consists in computing all the possible paths and then evaluate all their lengths to find the shortest one. However, the number of possible cycles is $\frac{(n-1)!}{2}$ and grows faster than 2^n (i.e. $2^n = o(n!)$) and thus making it unsolvable in finite time for large n. Moreover, even if it was possible to generate each hamiltonian cycle in constant time, independently from their length, the algorithm will still 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 do not exist any known polynomial algorithm to solve them and may not exist. However, noone was able to prove that deterministic polynomial algorithms cannot be found.

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

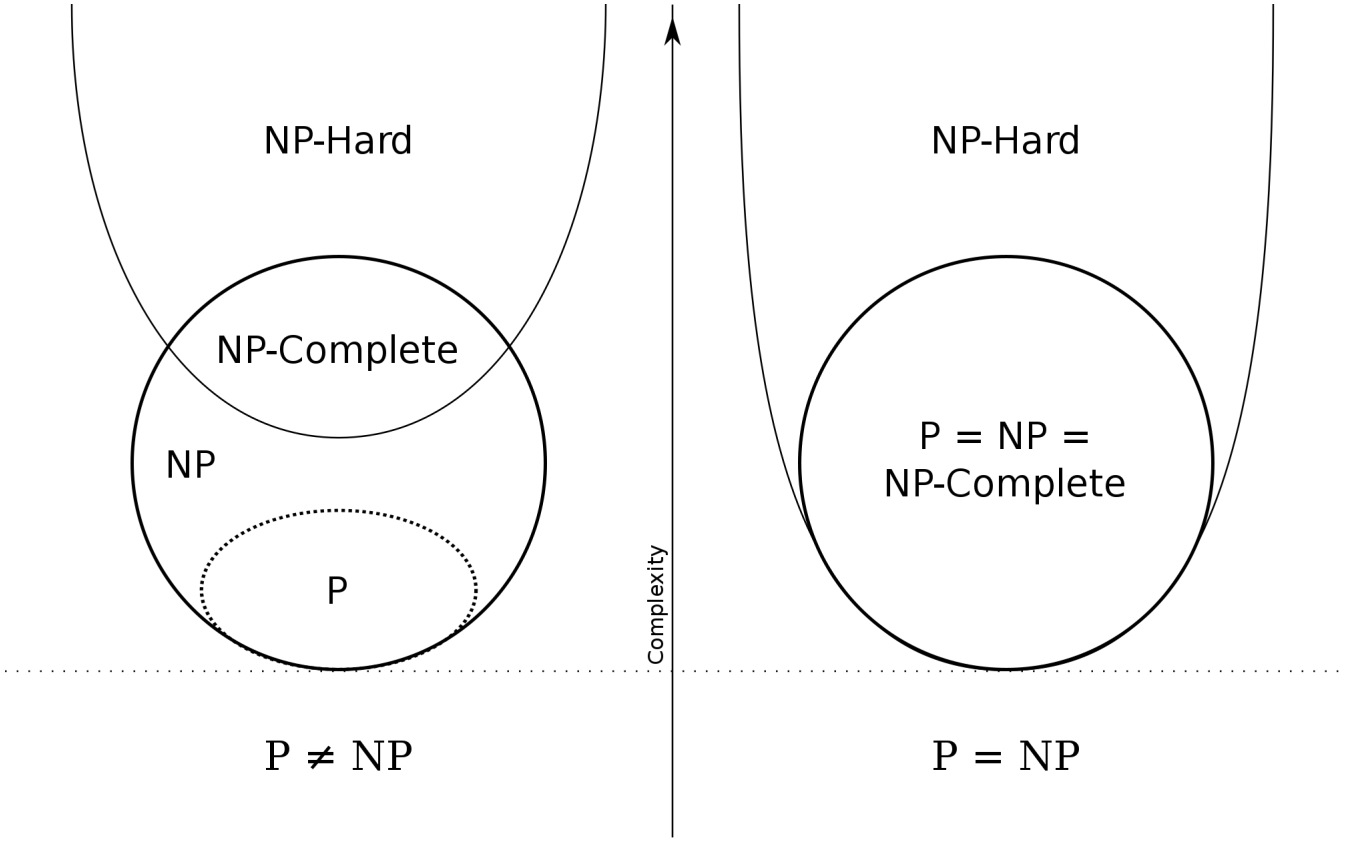

Figure 1: Computatinal complexity schema

Bearing in mind the complexity of the problem to face, the most reasonable approach relies on approximation. 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*.

2 Literature review

Many efforts were done to find out global optimization techniques that simulate optimization in nature. There can be depicted three main classes of algorithms:

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

Among the evelutionary algorithms, there can be found different subsclasses: evolutionary programming (1962), evolution strategies (1965) and genetic algorithms (1975) that mimic in different ways 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 and, not surprisingly, they are very fit for biological, pharmaceutical and chemical applications.
On the other hand, Artificial Neural Networks were exploited only recently since they require an enormous amount of data and a huge computational power. [6]
Moreover, those algorithms do not provide a strightforward interpretation of their parameters and therefore may not be suitable in logistic applications.

3 Algorithms and techniques

This section provides a top-down overview of all the techniques and algorithms exploited. They are: Simualted Annealing, Metropolis algorithm and a genetic algorithm.

3.1 Simulated annealing

Simulated annealing (SA) is a technique that mimics 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 the materials to reach the minimum energy state.
It is a technique to approximate the global optimal solution when it is to be searched in an enormous space. Indeed, SA is a metaheuristic that is a general-purpose 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 versatile since it does not require any prior information other than a cost function to be minimized, indeed it just supposes that states (hamiltonian cycles in this work) 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 models the distribution of states with the smallest possible bias. Indeed, its unbiasedness derives from the maximization of the Shannon's entropy, given the cost.
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.
Temperature T is the inverse of β ($T = \frac{1}{\beta}$), therefore the SA starts with low values of β (high temperature) and, as the simulation proceeds, the values of β increase, thus mimicking the decrease in temperature.
As mentioned before, the only other information needed is the cost function, that, for this specific application, is the sum of euclidean distances between the cities in the cycle, whose coordinates are expressed in longitude and latitude.
In the following algorithm provides a schematic overview of the simulated annealing implemented.

```
1: function SIMULATED ANNEALING(betas,number of simulations)
2:     s ← generate starting state with the genetic algorithm
3:     for b in betas do
4:         for n in 1,.., number of simulations do
5:             ns ← generate a new state with the genetic algorithm starting from s
6:             ns ← METROPOLIS(s,ns,β)
7:         end for
8:     end for
9:     return ns
10: 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

As SIMULATED ANNEALING algorithm above shows, Metropolis algorithm is exploited to sample states from the Boltzmann distribution. Therefore, this section describes the algorithm and analyzes in details its properties.
Metropolis algorithm 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.
The sequence of X_n can be modelled by Markov chains. Indeed, 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. $\forall n > 0 \, \forall i_n, ..., i_0 \, Pr(X_{n+1} = j | X_n = i_n, ..., X_0 = i_0) = Pr(X_{n+1} = j | X_n = i_n)$
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 π is the Boltzmann distribution:

$$\pi(\text{state}) = e^{-\beta \, \text{cost(state)}}$$

and S is the set of all possible hamiltonian cycles with fixed length.
The following disseratation assumes to have generated two hamiltonian cycles (i.e states) old_state, new_state, whose generation is described in section 3.3. Metropolis algorithms works as follows: if new_state is likely to be a sample of π it is used as new state otherwise, with a certain probability, old_state is 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

The sampling of new hamiltonian cycles is modelled by an undirected and connected graph $G = (S, E)$ whose set of states S contains all the possible hamiltonian cycles and E contains the transition probabilities from an hamiltonian cycle to another one. Given the graph G, the transition matrix $P = p(i,j)]_{i,j \in S}$ is defined 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, P is irriducible meaning that $\forall i,j \in S \, p^n(i,j) > 0$, where $p^n(i,j) > 0$ means that exist a path of length n from i to j.
It can also be proved that P is aperiodic [1], that is: each node of the graph has a loop. More formally, the graph G is aperiodic if $\forall i \in S$, $d(i) = 1$ where

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

Metropolis algorithm defines a probability distribution π that is *reversible*:

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

This is a key feature of the algorithm since a reversible distribution π is also a stationary distribution, i.e. $\pi' P = \pi'$. This condition holds due 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. $(\pi' P)_j = \sum_{i \in S} \pi(i)p(i,j) = \sum_{i \in S} \pi(j)p(j,i) = \pi(j)$ □

The last condition needed to assure to correctly generate the data accordingly to the target distribution π is the uniqueness of the distribution. However, stationary distibution is unique if the transition matrix of the markov chain 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.
By the convergence theorem, stationary distributions 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

To explain the functioning of the metropolis algorithm, it was assumed to have two states: new_state and old_state. In the following is explained how to obtain those states by the means of a genetic algorithm.
Genetic algorithms are a widespread class of algorithms that mimic nature. Indeed there are three fundamental operations inspired by the theory of evolution by C. Darwin that they perform to generate new states (hamiltonian cycles).
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 the crossbreeding operation mixes some subsection of the fittest hamiltonian cycles.

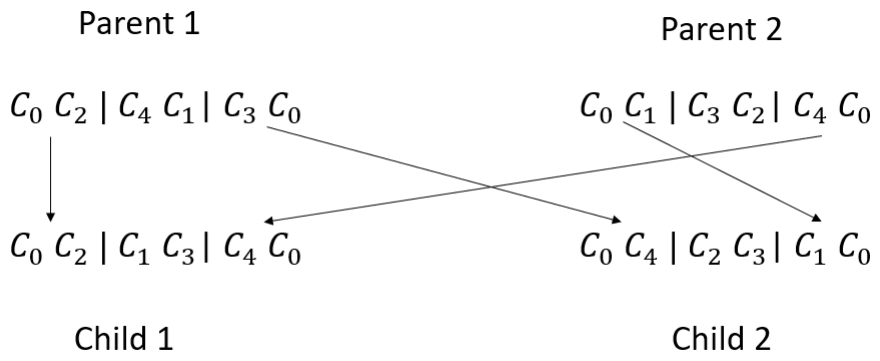

Figure 2: Crossbreeding schema

As random mutations can occur in nature but they are unlikely to, the algorithm swaps two cities with a fixed and low probability. Otherwise mutations will disrupt the optimal inherited subsequences.

```
1: function MUTATE(new_population, p)
2:     for path in new_population do
3:         test ← unif[0,1]
4:         if test < p then
5:             choose position1 randomly among unif[0,length of cycles-1)
6:             choose position2 randomly among unif[0,length of cycles-1)
7:             swap the cities in position1 and position2
8:         end if
9:     end for
10:    return new_population
11: end function
```

To easily apply all the operations, an EVOLVE function was created.

```
1: function EVOLVE(population)
2:     fittest_population ← SELECT(population)
3:     new_population ← CROSSBREED(fittest_population)
4:     population ← MUTATE(new_population)
5: end function
```

There are many possible implementations of the basic operations performed by a genetic algorithm. Therefore I decided to implement them from scratch. However, the implementation of those functions is not very interesting and requires many implementative details, therefore it is omitted 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 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.

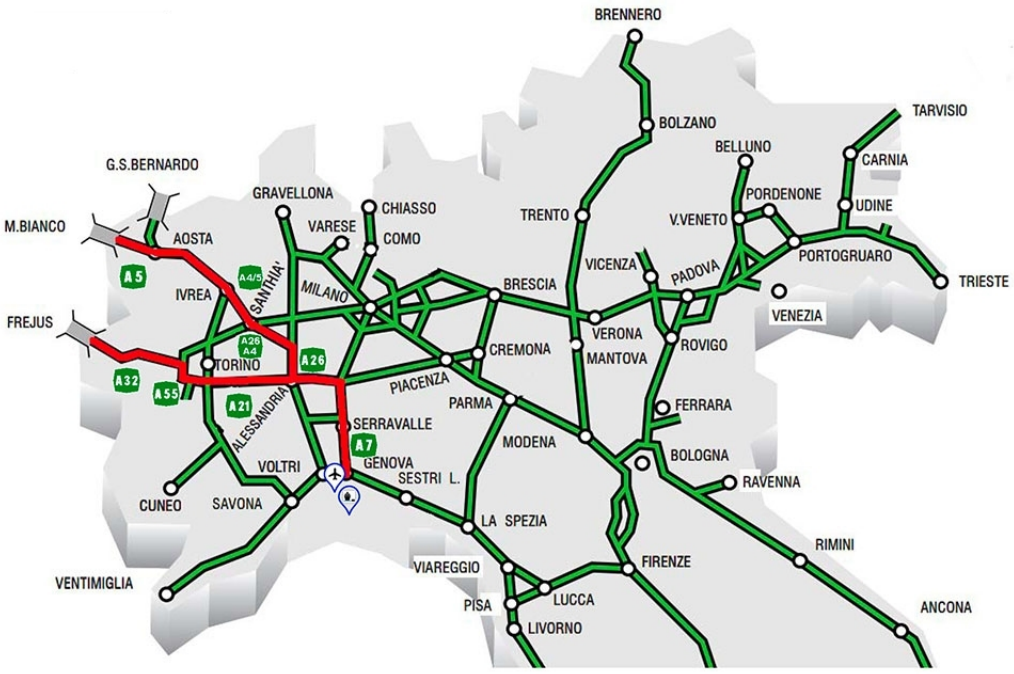

Figure 3: Scheme of the highways by *Autostrade per l'Italia*[8]

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 easily 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 information 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 compare the optimal solutions of the TSP computed by the appromated algorithm and the exact solutions. However, as explained before, the exact solution of the TSP is not feasible when the number of cities grows too much, therefore they are computed 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.

```
function EXPERIMENT(all_cities)
    for i in 1,..., n_cities do
        cities ← all_cities[0:i]
        for for all threads do
            run simluted annealing in parallel
        end for
        extract the best solution among the solutions computed by the threads
        if i < feasibility_threshold then
            compute the exact solution
        end if
        compare the results by plotting them
    end for
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*.

```python
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 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 by Metropolis algorithm.

```
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 shows the python implementation of the algorithm explained above.

```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 algorithms described were implemented from scratch exploiting just some support libraries such as *numpy*.

5 Results

This section compares the simulated annealing with the exact algorithm to solve the TSP problem. As expected the computational time required to compute an exact solution becomes exponentially high as the length of the cycles increases. However, for small instances of the problem, it is more efficient than the simulated annealing since it perform faster and fewer operations. On the contrary, the execution time[2] of the simulated annealing is almost independent from the length of the cycles and remains very low. It is worth noting that the slight variations and the peaks in the execution time plot may be caused by the scheduling of the operating system and the multithreading.

[2]All time measurement rely on a Intel Core i5-7200U dual core processor and 8 GB DDR4 RAM

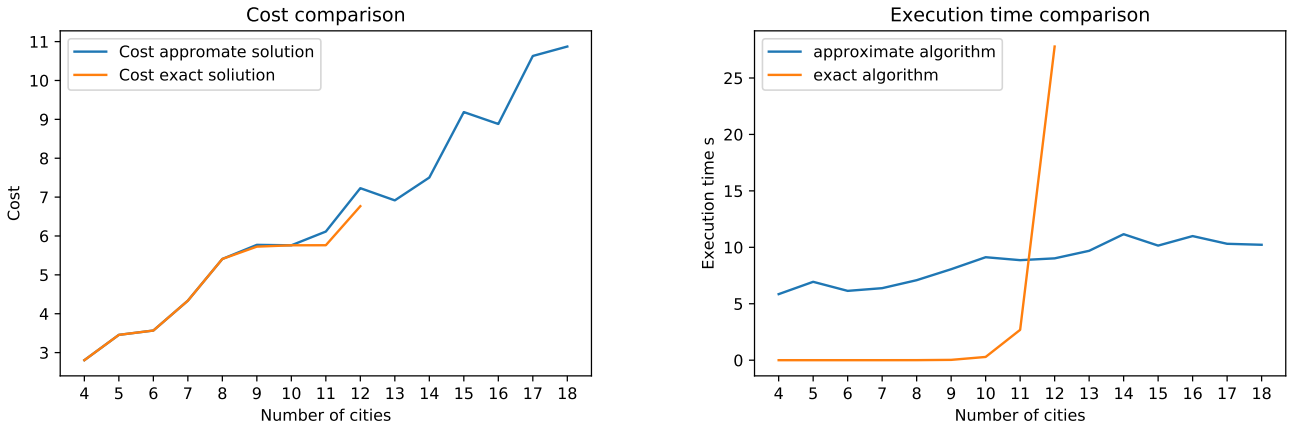

Figure 7: Comparison between algorithms.

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 smaller cost than that of shorter cycles. For instance, the cost of the cycle with 13 cities is lower than the cost of the cycle with 12 cities.

In the following, it is plotted the shortest cycle computed by the simulated annealing when using all the possible cities. As mentioned before, the starting point is *Varese*.

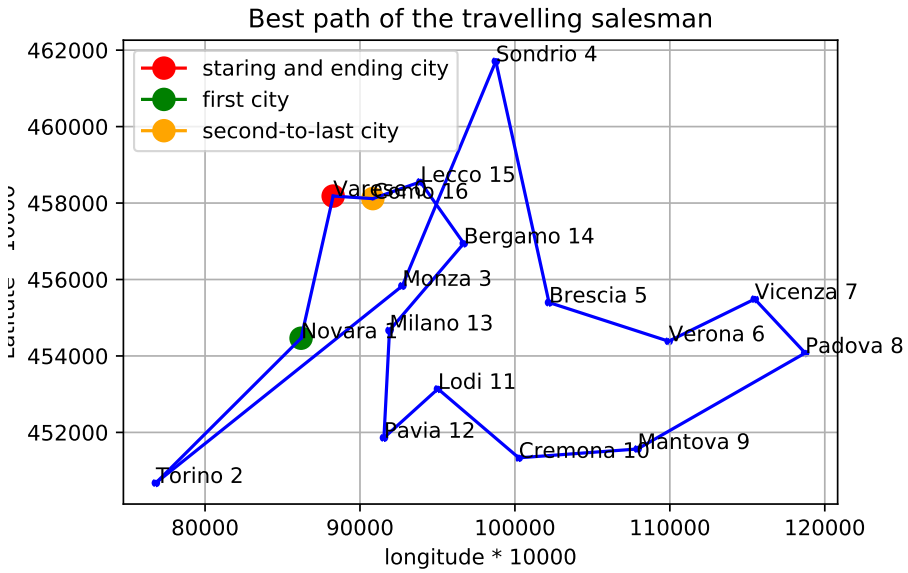

Figure 8: Optimal cycle starting from Varese

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 influence on the execution time, indeed, it exhibits a very flat slope.
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