

SIMONE QUADRELLI



GENETIC ALGORITHM MONTE CARLO

Academic year 2019 - 2020

Contents

1	Introduction	1
2	Simulated annealing	1
3	Metropolis algorithm	2
4	Genetic Algotirthm	3
5	dataset	3
6	results	3

List of Tables

List of Figures

Abstract

The aim of this project is to solve the travelling salesman problem (TSP) which consists in finding the shortest cycle among all the cities. The problem is a *NP-hard* problem and therefore there exist no feasible algorithm to solve it for any possible set of cities in input. The approximately optimal solution was computed by the simulated annealing, while the possible cycles are produced by a genetic algorithm.

1 Introduction

The travelling salesman problem consists in finding the shortest route a travelling salesman has to take to visit all the city he must reach only once and to return to the starting point. The route described is called in graph theory *hamiltonian cycle* (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.

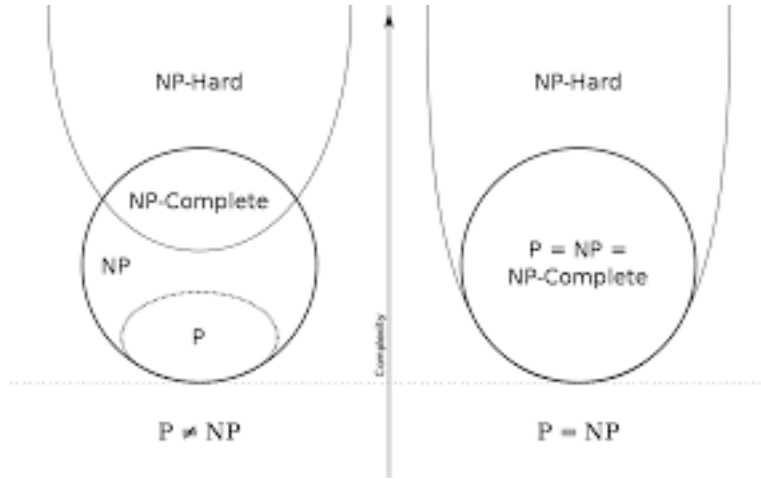
An algorithm that provides a solution to TSP can be exploited in a wide variety of applications: indeed it can be used in logistics to optimize the path of vehicles and therefore to reduce costs. It is possible to model the problem using an undirected, weighted and fully connected graph ¹ $G = (N, E)$, where N is the set of the cities to visit and $E = N \times N$ is the weight of the path. In this particular instance of the problem the weights correspond to the euclidean between each couple of cities. The hamiltonian cycle can be defined as a sequence (n_0, \dots, n_{k-1}) of length k such that $n_i \neq n_j \forall i, j \in \{1, \dots, |N|\}$ and such that $(i, j) \in E \forall i, j \in \{1, \dots, |N|\}$. The most naive solution is to compute all the possible paths but they are factorial in the number of cities n in input. The number of possible cycles is $(n - 1)!$ and grows faster than 2^n (i.e $2^n = o(n!)$). Therefore even if it was possible to find them in constant time the algorithm will have an exponential time complexity and would not be solved in finite time for large n . More technically, TSP is an NP-hard problem. A NP-hard problem is a problem at least as hard as the hardest NP-complete problem. NP-complete problem is a problem 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 no one was able to prove that such algorithms does not exist. 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 fittest hamiltonian cycle is done by the simulated annealing algorithm

2 Simulated annealing

Simulated annealing (SA) is a technique that mimics the behaviour of heated materials in metallurgy that can be slowly cooled down, this process decreases the impurity and increases the size of the crystals. It is a technique that approximates the global optimal solution when the solution is to be searched in an enormous space. SA is a metaheuristic meaning that it is not a problem specific heuristic and that can be used to reduce the search space of an algorithm, the genetic algorithm in our case.

SA is very useful since it does not require any prior information, indeed it just supposes that the

¹For an hamiltonian cycle to exist the graph must be at least connected



states (hamiltonian cycles in our case) are distributed accordingly to the Boltzmann distribution:

$$\exp^{-\beta(cost(newstate)-cost(oldstate))} \quad (1)$$

The temperature T stated before is $T = \frac{1}{\beta}$, therefore we start with low values of betas (high temperature) and as the simulation proceeds we increase the values of β , thus mimicing the decrease in temperature. The only other information needed is the cost function. For this specific application the cost function is the euclidean distance between the cities. The (x, y) are expressed in latitude and longitude (CONTROLLA L'ORDINE). Therefore given the coordinates of two cities (x_1, y_1) and (x_2, y_2)

$$cost((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2)$$

```

1: function SIMULATED ANNEALING
2:   s ← generate starting state
3:   for b in betas do
4:     for n in 1,..., number of simulations do
5:       ns ← sample new states with metropolis algorithm
6:     end for
7:   end for
8:   return ns
9: end function

```

3 Metropolis algorithm

Metropolis algorithm is used to sample values from a probability distribution moving at each step with high probability from a state with lower probability from a state with higher probability

```

1: function METROPOLIS(old_state, new_state)
2:   if cost(new_state) < cost(old_state) then
3:     return new_state
4:   end if
5:   threshold  $\leftarrow$  unif[0,1]
6:    $p \leftarrow \min\{1, \exp^{-\beta(\text{cost}(\text{new\_state}) - \text{cost}(\text{old\_state}))}\}$ 
7:   if p > threshold then
8:     return new_state
9:   else
10:    return old_state
11:   end if
12: end function

```

where new_state and old_state are hamiltonian cycles. Metropolis algorithm converges to the stationary distribution in logarithmic time, having fixed the approximation. Therefore we execute it for a sufficient number of iterations to converge to the stationary distribution. Then we change the value of β in the simulated annealing to proceed toward the optimum.

4 Genetic Algorithm

Genetic algorithms are a widespread class of algorithms that mimic nature. Indeed there are three fundamental operations they perform, each operation is inspired by the theory of evolution by Darwin. Among a species evolution selects the fittest individuals and therefore the algorithm will choose the fittest (i.e. shortest) hamiltonian cycles. The breeding of the fittest individuals produces new individuals that inherit traits from the parents, therefore in the crossbreeding phase we mix some subsection of the fittest hamiltonian cycles. As random mutations can occur in nature but they are not so likely to occur, the algorithm swaps two nodes (cities) with a fixed and low probability. My own implementation of the algorithm performs the selection operation as follows:

```

1: function SELECTION(population, n_paths)
2:    $n\_parents \leftarrow \text{unif}[2, n\_paths/2]$ 
3:   order the population by fitness
4:   fittest_subpopulation  $\leftarrow$  extract  $n\_parents$  in order from the sorted population
5:   return fittest_subpopulation
6: end function

```

where **population** is an array of hamiltonian cycles

```

1: function CROSSBREEDING(fittest_subpopulation)
2:   for i in 1, ..., n_paths do
3:     choose parent1 uniformly among the fittest_subpopulation
4:     Choose parent2 uniformly among the fittest_subpopulation
5:     if parent1 different from parent2 then
6:       choose position1 randomly among unif([0,length of cycles-1)
7:       choose position2 randomly among unif([0,length of cycles-1)
8:       divide parent1 in three sections:
9:       [parent10:position1]
10:      parent1[position1:position2]
11:      parent1[posotion2:]
12:      divide parent2 in three sections:
13:      parent2[0:position1]
14:      parent2[position1:position2]
15:      parent2[posotion2:]
16:      breed child1 justapposing in the order:
17:      parent1[0:position1]
18:      all the cities not in parent1[0:position1] and not in parent2[posotion2:]
19:      parent2[posotion2:]
20:      breed child2 justapposing in the order:
21:      parent2[0:position1]
22:      all the cities not in parent2[0:position1] and not in parent1[posotion2:]
23:      parent1[posotion2:]
24:     else
25:       child1  $\leftarrow$  parent1
26:       child2  $\leftarrow$  parent1
27:     end if
28:   end for
29:   new_population is a vector of the children
30: end function

```

2) Crossbreeding – i obtain new hamiltonian cycles exchanging the gines of the parents 3) Mutation – i exchange genes randomly

```

1: function MUTATE(new_population, p)
2:   test  $\leftarrow$  unif[0,1]
3:   if test < p then
4:     choose position1 randomly among unif([0,length of cycles-1)
5:     choose position2 randomly among unif([0,length of cycles-1)
6:     swap the cities in position1 and position2
7:   end if
8:   order the population by fitness
9:   fittest_subpopulation  $\leftarrow$  extract  $n\_parents$  in order from the sorted population
10:  return fittest_subpopulation
11: end function

```

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

5 dataset

6 results