Simone Quadrelli

Genetic algorithm Monte Carlo

Academic year 2019 - 2020

Contents

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 if for any possible set of cities in input. The appromximately optimal solution was computed by the simulated annealing, while the possible cycles are produced by a genetic algorithm.

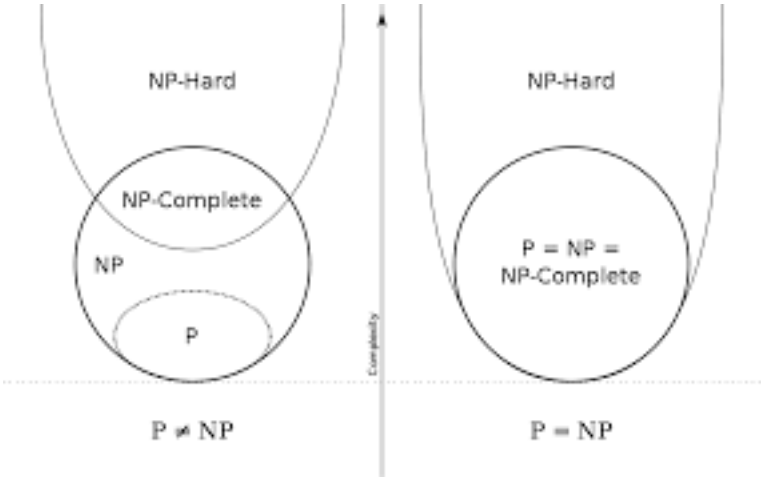
1 Introduction

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 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 probelm the weights correspond to the euclidean between each couple of cities. THe mailtonian cycle can be defined as a sequence $(n_0, .., n_{k-1})$ of length k such that $n_i \neq n_j \ \forall i, j \in \{1, ..., |N|\}$ and such that $(i, j) \in E \ \forall i, j \in \{1, ..., |N|\}$ 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 that 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 expnential time complexity and would not be solved in finite time for large n. More techically, 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 noone was able to prove that such algorithms does not exists. Bearing in mind the complexity problem to face, the most reasonable solution is to solve the problem approximately. The technique I 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 mimic 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 pproximates 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 heuristic and that can be used to reduce the search space of an algorithm, the genetic algorithm in our case.
SA very usefull since it does not require any prior information, indeed it just supposes that the

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



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

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

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_3, y_2)

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

```
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

4 Genetic Algotirthm

Genetic algorithms are a widespread class of algotirhms that mimic nature. Indeed there are three fundamental operations they perform, each operation is inspired by the theory of evolution by

Darwin. Among a specie evolution selects the fittest individuals and therefore the algorithm will chose the fittest (i.e shortest) hamiltonian cycles. 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. As random mutations can occur in nature but they are not so likely to accur, the algorithms swaps two nodes (cities) with a fixed and low probability. My own implementation of the algorithms performs the selection operation as follows: where **population** is an array of 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
```

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

5 dataset

6 results

```
1: function CROSSBREEDING(fittest_subpopulation)
2:     for i in 1, ..., n_paths do
3:         choose parent1 uniformely among the fittest_subpopulation
4:         Choose parent2 uniformely 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 ← parent1
26:            child2 ← parent1
27:        end if
28:    end for
29:    new_population is a vector of the children
30: end function
```

```
1: function MUTATE(new_population, p)
2:     test ← 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 ← extract n_parents in order from the sorted population
10:    return fittest_subpopulation
11: end function
```

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