Practice 5

Guillermo Girón García

December 29, 2019

Abstract

Attempt to suboptimally solve the Travelling Salesman Problem through the use of the Ant Colony Optimisation algorithm, and some of its extensions: Elitist Ant System and MAX-MIN, stablishing and testing its parameters.

1 Introduction

Through the course of this paper we going to study the resolution of the Traveling Salesman Problem (TSP to shorten) using an Ant Colony Optimisation (ACO from now on) approach.

The reason behind choosing ACO is that approach appears to find good results faster than other algorithms, because the ants are fittingly searching an optimal path with stochastic yet visibly intelligent behaviour, giving more importance to an edge that has a better heuristic and pheromone value, whereas a genetic algorithm approach would be much more randomised since it first buidls random solutions and then chooses and randomly modifies the best ones. Besides, a genetic algorithm is generally more time consuming, both in its bigger computation time an the more parameters that need to be adjusted.

2 Methodology

All the code can be checked in the antColony.cpp file, which takes care of reading the .tsp and .opt.tour files needed concurrently in different threads and finding a result for it.

First of all, we need a structure to store the information about the cities. In this case, a vector of <city> types, where a city simply contains a cartesian pair of float coordinates x and y. Through the readfile function, a file given in the parameter string is read and all of its cities stored in a vector of that caliber.

Next, we build a square matrix which contains the distnaces between every pair of cities, via the getDistances function, which receives the vector of cities and returns the matrix of distances, which we will call 'map' from now on. This matrix is symetrical, and its main diagonal is never accessed.

Afterwards, we hace to set all the parameters for the ACO algorithm:

- Randomseed: A seed to set randomisation.
- Max_it: The amount of iterations the algorithm will do before stopping.
- Num_ants: The amount of ants the colony will simulate.
- Decay_factor: The factor to determine the speed of the pheromones evaporation.
- Alpha: Factor to determine the importance of the heuristic value of the paths for an ant to choose.
- Beta: Factor to determine the importance of the pheromone value of the paths for an ant to choose.

- C_phero: Amount of pheromones that each ant will leave in its path.
- C_elite: Amount of extra pheromones that elite ants will leave in its paths, in the elitist expansion.
- doExpansion: Logical flag to determine whether or not the algorithm must use the elitist expansion.

With all previous steps done, it is time to start with the main algorithm: the ACO function, receiving all the parameters above plus the vector of cities and the map.

The ACO implementation follows the standard algorithm: find paths, put pheromones, evaporate pheromones, repeat; plus the elitist expansion of extra pheromones for the current best path. The heuristic factor is determined as follows: MAX_DIST - current_dist, where MAX_DIST is the maximum distance of the entire map, and current_dist the distance of the given edge.

However, there is one important factor that needs to be mentioned. When an ant is forming a path, it tries to find the smallest circuit from the first city to the last one, but it doesn't take into account the returning trip required to close the cycle. This is because each ant takes its current city and selects the next one based on a roulette selection with the heuristic and pheromone values of all possible edges from said current city. In order to make up for this loss of information, every ant is assigned to one specific starting city, so that every city has an equal amount of ants staring from it (or at least as similar as possible if the divission isn't exact) and the colony as a whole can determine a closed cycle.

Lastly, after the algorithm has finished, the best ant of the colony is returned. Its path cost is then processed and showed, along with the theoretical optimal of this instance.

3 Results

In this ocassion we are going to use three different implementations that are:

1. berlin52.
2. kroA100.
3. eil51.
4. pr2392.

We set most of the parameters to one except decay that is set to 0.6 and we will start for the first one of the files.

Under these circumstances, the behaviour of the algorithm is certainly intreresting.

The optimal solution of the berlin sample is about 7.000, and the algorithm only finds solutions of about 21.000 with the "default" values. Changing the amount of iterations doesn't have an impact, as the solution barely varies from 50 to 50.000 iterations except for the computation time, that can become extremely long.

Also, there is a very small difference between using or not the elitist expansion, as well as varying the pheromone coefficients, even with drastic amounts like tens of thousans. Assigning more ants to each city doesn't seem to factor in the results either.

However, the values of alpha and beta seems to be a key factor in the performance. A high beta value of about ten, along with various ants per city, a very high pheromone coefficient (at least at the same scale as a path cost, 10.000) and a high elite coefficient as well, the algorithm is capable of finding better solutions (~12.000) in around 1.000 iterations.

On the other hand, leaving all the previous values at default, and instead increasing alpha up to 30, returns better results much quicker (~9.000 in under 50 iterations).

Lastly, combining both parametrizations, the algorithm is able to find slightly better solutions (8500-8700), again ir around 1.000 iterations.

For kroA100 using the same parameters we obtain a very good optimization of the solution in only 100 iterations. Despite the low iteration number, it still takes a lot of time to be computed in comparison with the others that are computed almost instantly.

In eil5, again experimenting with the same parameters, we obtain good results aswell, with an optimization of around a 15%.

Lastly, at pr2392 sample, it becomes impossible to compute a solution with this algorithm due every time we try to execute it, it throws a memory problem, forcing me to through it out of the main program to avoid execution errors.

4 Conclussions

Reached this point we can say there is an interesting difference in the two ways that we can adjust the parameters of the algorithm. On one hand, we can give a lot of emphasis to the concept of the ACO approach, giving high values to all the parameters regarding pheromone spreads, number of ants and pheromone importance, and eventually it is able to improve the solutions.

On the other hand, we can treat the algorithm like a series of individual local searches, giving most of the importance to the heuristic values, resulting in good solutions with little iterations, that do not improve overtime.

Finding a good balance between both approaches is able to yield better solutions than either of them alone, but of course with larger computation time than a local search, and the solutions found might be even better than the ones shown here if given enough number of iterations, but taking in acount that time will grows much faster than optimization, and that almost the same patter and values, works well for the three analyzed samples.

References

[1] Ant Colony Optimization for hackers, by Lee Jacobson.

[2] Solving the Travelling Salesman Problem using the Ant Colony Optimization, by Ivan Brezina Jr. and Zuzana Cickova.

[3] Ant Colony Optimization - Optimal number of ants, by Christoffer Lundell Johansson and Lars Pettersson.