Exploring a zero cost solution for Travelling Salesman Problem (TSP)

Carlos Sá - A59905
carlos.sa01@gmail.com

Bruno Barbosa - A67646
a67646@alunos.uminho.pt

March 19, 2016

Abstract

*This report is a result of a study about Monte Carlo algorithm applied to Travelling Salesman Problem (TSP) exploring the Simulated Annealing (SA) meta-heuristic. We've a discrete space of cities and the algorithm finds the shortest route that starts at one of the towns, goes once through everyone of the others and returns to the first one. The main goal is explore the possibility of having a zero cost solution with **n** cities and **p** processors running in parallel. To perform this analysis we'll gonna use a TSP algorithm with MATLAB.*

I. Introduction

The travelling salesman problem (TSP) is a NP-Hard in combinatorial optimization and an important case of study in computer science. In this problem, a discrete set of cities and the distances between each pair of them are given. The main goal is to find the shortest possible route that visits each city once and returns to the original one.

In addiction to the set of cities it's necessary to know the distances between them. In computer science this problem if often seen as a graph problem using a symmetric adjacency matrix where each element represents the cost/distance between each pair of two cities.

In this particular case of study, the algorithm uses Simulated Annealing (SA) to solve the Travelling Salesman problem. SA is a probabilistic technique for approximation the global optimum of a given function. This method explores the **Temperature** concept and accepts worse solutions as it explores the solution space. These exploration of worst solutions is a fundamental property of this meta-heuristic. Sometimes its made an analogy with a mountain: simulated annealing accepts a higher distances (worse solutions) climbing the mountain of solutions if after overcome the moutain peak, we achieve a better (and lower) distances. In this study we are going to explore SA meta-heuristic with a MATLAB code provided and explore the possibility of introducing a zero cost solution for the TSP.

II. MATLAB Implementation of TSP

The first step of this work was explore and analyse the MATLAB's implementation solver for TSP. On this exploration process we noticed that one of given files has an implementation of the *Simulated Annealing* which is a probabilistic optimization technique used on large discrete search spaces like the ones we gonna analyse on this work assignment. It works by leading the system to states of lower energy. Iteratively, the system will achieve a state where it is impossible to get a lower energy state wich cause stop the search. This meta-heuristics uses the concept of Temperature wich will be related with new solutions.

The image below show us the relation between the temperature and the new solutions used in simulated annealing meta-heuristic used in our MATLAB code.

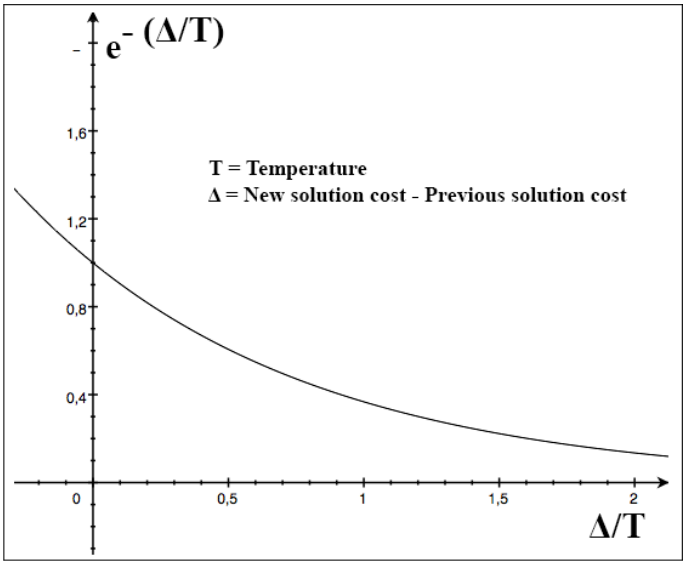

Figure 1: *Relation between new solutions with SA and Temperature*

We have the MATLAB implementation divided into three main matlab files:

- **travelling2.m**: This function uses **rand** MATLAB function to generate pseudo-random values using normal distribution between 0 and 1. The initial route is and new a new route is accepted if its length is less than the previous. This process is continuously applied over a distancies symmetric matrix wich is also known as *adjacency matrix* in computer science area.
- **traveling.m**: This code is based on the previous, but where *Simulated Annealing* is applied to solve the problem.

- **PARtraveling.m**: This function uses *traveling* and *traveling2* functions described before and run the solver with a "parallel" execution of **P** processes and **n** cities.

Note that *"parallel execution"* doesn't mean we made multiple searches for getting one global solution in just one execution. If we run *PARtraveling(ncities,nprocs)* the same simulation for *ncities* is performed *nprocs* times, but each one runs sequentially. Each process executes the same code but with different random numbers and getting different costs. So after calling PARtraveling the parallel solution is the answer we'll get with a smaller cost for all processes.

In the next section III we'll see the main changes we've made in the code for trying to get a zero cost solution. The necessary changes was performed in this **PARtravelling** function.

III. Zero Cost Route

There could be many ways to introduce a route without a cost. Basically we want to allow the algorithm to find a path through all cities where there isn't any cost associated with it. In our particular case we setup a zero cost route by the simplest way which is according to the order of the cities.

For example, consider we have four cities {A,B,C,D}. Then the distance among cities A and B will be zero, as well as the distance among cities B and C, C and D and at last, among D and A. We must remember that beyond this solution it still remains $(4! - 1)$ routes where the total distance is different from zero. The reason of the actual choice was to understand how flexible is the algorithm's capability to find the shortest path through all cities.

The major changes made to the original source code boil down to the insertion of the zero cost route just like we mentioned before.

```
for i=1:n
    j = i+1;
    if(j == n+1)
        j=1;
    end
    D(i,j) = 0;
    D(j,i) = 0;
end
```

This modification was added to the *PARtravelling.m* file just after the loop that fills the distance matrix. As shown on the previous code example, the matrix is represented by the variable D. From what we can see, this procedure simply crosses the distance matrix through the main diagonal by setting the cost between the actual city and the next one to zero, in both directions. A exceptional situation occurs when it reaches the last city and it must joins to the first one. That's why the *if* statement is there for.

Once we used a bidirectional graph representation to store the distance between all cities, we easily realize that the original matrix only has the main diagonal with zero values, which match with the distance between a city and itself. See the matrix presented underneath.

$$D = \begin{bmatrix} 0 & 2.0968 & 1.7297 & 4.5634 & 6.3701 & 3.1868 \\ 2.0968 & 0 & 0.3682 & 2.9365 & 7.6584 & 5.2488 \\ 1.7297 & 0.3682 & 0 & 3.1640 & 7.4248 & 4.8816 \\ 4.5634 & 2.9365 & 3.1640 & 0 & 10.5866 & 7.2211 \\ 6.3701 & 7.6584 & 7.4248 & 10.5866 & 0 & 6.2161 \\ 3.1868 & 5.2488 & 4.8816 & 7.2211 & 6.2161 & 0 \end{bmatrix}$$

After setting up the zero cost route, the distance matrix will now look like the one we present down below.

$$D = \begin{bmatrix} 0 & 0 & 1.7297 & 4.5634 & 6.3701 & 0 \\ 0 & 0 & 0 & 2.9365 & 7.6584 & 5.2488 \\ 1.7297 & 0 & 0 & 0 & 7.4248 & 4.8816 \\ 4.5634 & 2.9365 & 0 & 0 & 0 & 7.2211 \\ 6.3701 & 7.6584 & 7.4248 & 0 & 0 & 0 \\ 0 & 5.2488 & 4.8816 & 7.2211 & 0 & 0 \end{bmatrix}$$

It were also added some lines of code so we could measure the time spent on each situation and make a deeper analysis.

IV. Exploring results

After finished our MATLAB implementation, we performed a set of tests in order to understand the behaviour of the metaheuristic and draw some conclusions about it.

As we said before, we only generate a random x and y once. We'll use the same x and y and run the code for 6, 6^2 and 6^3 cities with 4, 8 and 16 procs. We adopt this strategy so we can verify if there is relation between the results running the code for the different number of cities and different number of processes. In this work we're going analyse **execution time**, and **total distance** for comparing the two solutions achieved: with Simulated Annealing and the solution without Simulated Annealing. This time measurement was made with *tic* and *toc* MATLAB functions.

V. Performance analisys

	Execution Time (sec) **(with SA)**		
NºCities	NP = 4	NP = 8	NP = 16
6	1,697881	2,659874	6,084847
36	1,131469	2,161275	4,396453
216	1,199692	2,248346	3,045308

Table 1: *Execution times with SA for different number of processes*

	Execution time (sec) **(without SA)**		
N°Cities	NP = 4	NP = 8	NP = 16
6	2,59231	2,683813	6,038509
36	1,185353	2,685516	4,6031
216	1,273476	2,240703	3,367285

Table 2: *Execution times without SA for different number of processes*

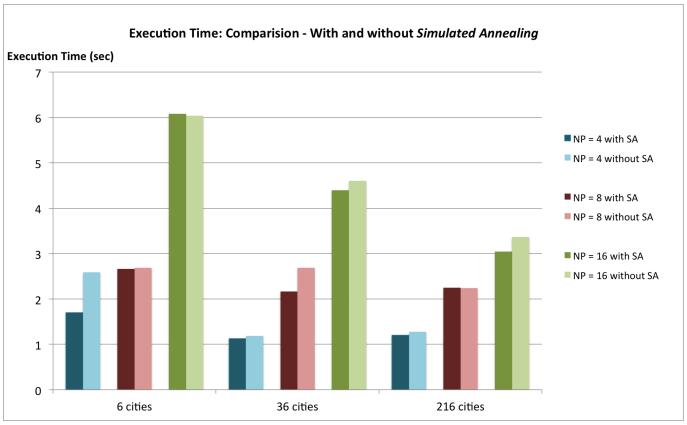

Figure 2: *Solution with SA VS without SA for different number of processes*

As we can see by figure 2, the execution time using SA tends to be smaller. However by all experimentation we made, we conclude that this does not always happen. So we can't get a global conclusion if we can get a better performance using SA.

All results was obtain in MATLAB R2015b using Intel Pentium T4300 @ 2.10Ghz with 4GB RAM DDR2 and Windows 10 Pro 32-bit.

VI. Total distance Analysis

As we saw, the performance analysis we made before doesn't allow us getting a global conclusion. However we could draw some conclusions about total distances comparing the results with SA and without SA with the changes we made in the code, injecting a zero cost path between each pair of cities. The following images show the solution we get with MATLAB:

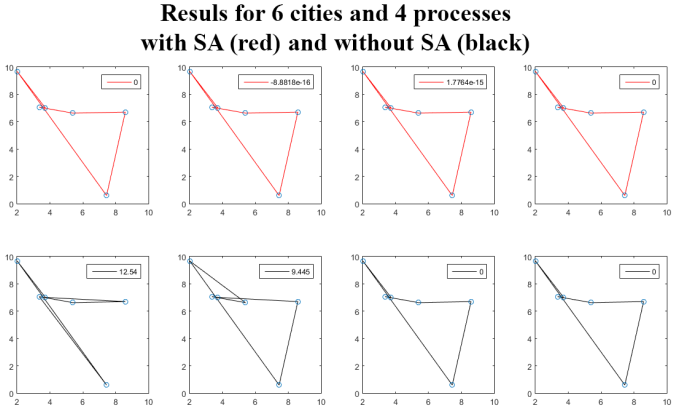

Figure 3: *Solution with SA VS without SA for 4 processes*

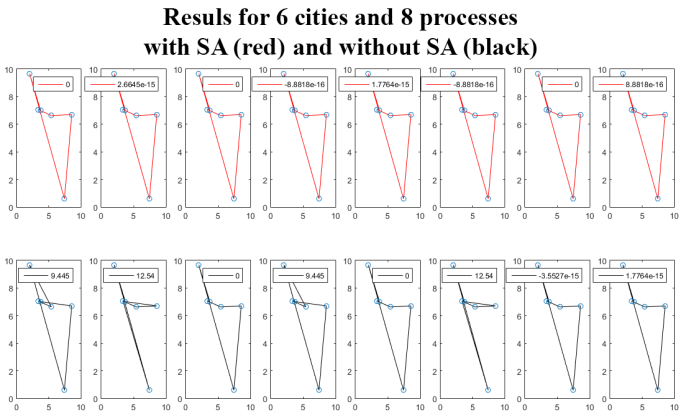

Figure 4: *Solution with SA VS without SA for 8 processes*

As we can see by figure 3 and 4 we could conclude in general that when we run the solver with *Simulated Annealing* algorithm we can get minor distances for different number of processes. Although we can see some zero cost solutions without *Simulated Annealing* too, they are not so obvious as with it. Another thing we can take from this experimentation is that as long we increase the number of processes the algorithm seems to get better results related with the shortest path. We could prove what we learn about simulated annealing. Accepting worse solutions (like the analogy of climbing a mountain) can be advantageous if after overcome the mountain peak, we achieve a lower solution - in this algorithm, a lower total distance.

VII. Conclusions

In this work we made an analysis of TSP algorithm with Simmulated Annealing meta-heuristic. We inject a zero cost/distance path between each pair of cities and made a study about total distances achieved and a performance analysis. This analisis was made for different number of cities and different number of processes. We conclude that Simulated Annealing tends to get a better solution with a lower execution time but it doesn't not always happen. It depends on the number of cities and the number of processes chosen. This fact occurs even when we explore the solutions with different number os processes using the same matrix.

However we realise that climbing to costly solutions is better if after overcome the "moutain peak" we could get a lower global cost - in this case, lower distance to perform the circuit between all cities. The zero cost path we inject in the code also made some changes on the matrix as we saw on section III. After we added the zero cost route, the matrix has passed to have a thicker main diagonal with zero values because its adjacent diagonals became zero too. Furthermore, the first and last elements of the secondary diagonal also turned up to zero since they represent the distance of returning to the initial city.

As soon as we prepared the distances matrix we were ready to start the experimentation. According to the test we made we saw that, in general, the time spent with *Simulated Annealing* was slightly lower than without it. In terms of the shortest path,

we concluded that both algorithms could reach the zero cost route though it seems to be more evident when using *Simulated Annealing* again. We cannot guarantee that the results we get will be equal on other similar tests.

[1] [3] [2]

References

[1] Simulated Annealing - Wikipedia. https://pt.wikipedia.org/wiki/Simulated_annealing.

[2] Simulated Annealing For Beginners. http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6.

[3] David Bookstaber. *Simulated Annealing for Traveling Salesman Problem*. Spring, 1997.