Solving the Traveling Salesman Problem: a multithreaded approach with evolutionary genetic algorithms

Ibsan Acis Castillo Vitar

School of Sciences and Engineering, Tecnológico de Monterrey, Puebla Campus
A01014779@itesm.mx

Luis Fernando Dávalos Domínguez

School of Sciences and Engineering, Tecnológico de Monterrey, Puebla Campus
A01128697@itesm.mx

Salvador Orozco Villalever

School of Sciences and Engineering, Tecnológico de Monterrey, Puebla Campus
A07104218@itesm.mx

November 13th 2017

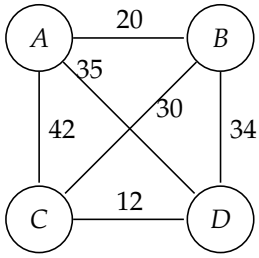
Abstract

The Traveling Salesman Problem is a one of the most common NP-Complete problems. As one of the most complicated problems in the history of Computer Science, there currently are no algorithms that can solve it in polynomial time. The problem's description is simple: given N cities that a salesman has to visit, find the order in which he must visit them to minimize the total distance traveled if he has to visit all cities once and come back to the first city after visiting the last one. For each number of cities N, the number of permutations of cities is N!, which makes the problem's size grow very fast. We approach the problem via genetic algorithms.

I. Introduction

The goal of this project is to obtain the optimal solution for the Traveling Salesman Problem, a problem that consists of a salesman and a set of cities which have to be visited. The salesman has to visit each city once and return to the same city after the last one has been visited.

Consider the following image. The problem lies in finding the shortest path visiting all vertices once. For example the path 1:$A, C; D, B, A$ and 2:A, B, C, D, A visit all the vertices once, but path 1 has a length of 108 while path 2 has a length of 97.



The number of cities in this problem is 4, and there are $4! = 24$ possible routes, which makes the problem very complex as the number of cities grow.

We approach this problem with evolutionary genetic algorithms. We receive the number of cities and their geographic coordinates. We calculate the distance between cities using the Haversine distance formula.

II. Mathematical Formulation

For a given $n \times n$ distance matrix $C = (c_{ij})$, find a cyclic permutation π of the set $\{1, 2, ..., n\}$ that minimizes the function

$$c(\pi) = \sum_{i=1}^{n} C_{i\pi(i)} \tag{1}$$

where $c(\pi)$ is the length of the permutation π, computed through a distance metric. In our case, the metric is the Haversine formula to calculate the distance between two fixed points on Earth given the latitude and longitude of each point.

III. Related Work

The Traveling Salesman Problem was first considered mathematically in the 1930s by Merrill Floyd. In the 1950s and 1960s, the problem became increasingly popular in scientific circles in Europe and the USA. Notable contributions were made by George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson; they expressed the problem as an integer linear programming problem and developed the cutting plane method.

In the following decades, the problem was studied by many researchers from mathematics, computer science, chemistry, physics and other sciences.

A chemist, V. Černý, created a thermodynamical approach to the Traveling Salesman Problem. He created a Monte Carlo algorithm to find approximate solutions of the TSP. The algorithm generates random permutations, with probability depending on the length of the corresponding route. Reasoning by analogy with statistical thermodynamics, it uses the probability given by the Boltzmann-Gibbs distribution. Using that method they could get very close to the optimal solution.

IV. Methods

i. Input

The input to the program is a text file containing a line with an integer N, the number of cities the traveling salesman has to visit. The next N lines contain each two floating-point numbers corresponding to the latitude and longitude of the i-th city for $i = 1, 2, ..., N$.

ii. Output

The output of the program is the best generated chromosome in terms of the distance required to visit all cities and come back to the first one.

iii. Algorithm

The algorithm used to solve this problem is as follows:

iii.1 Genetic Algorithm with Ordered Crossover

Algorithm 1 Genetic Algorithm with ordered crossover

procedure Genetic Algorithm with ordered crossover
  Set C, the total amount of chromosomes per generation
  Create C chromosomes each with a random permutation
  Set G, the total amount of chromosome generations
  for $i \leftarrow 0, G-1$ **do**
    Sort the array with the C chromosomes at generation i
    Let the first $\frac{C}{4}$ *elite* chromosomes live to the next generation
    Create $\frac{3C}{4}$ child chromosomes from the $\frac{C}{4}$ *elite* chromosomes
  Sort the array of chromosomes one last time
  return the first chromosome in the array

iii.2 Pairing Schema

Use the following pairing schema to generate child chromosomes:

1. The i-th chromosome with the $(i+1)$-th chromosome $\left(1 \leq i \leq \frac{C}{4}\right)$

2. The i-th chromosome with the $(i+2)$-th chromosome $\left(1 \leq i \leq \frac{C}{4}\right)$
3. The i-th chromosome with the $(i+3)$-th chromosome $\left(1 \leq i \leq \frac{C}{4}\right)$

Note: all indices are taken modulo $\frac{C}{4}$.

iii.3 Ordered Crossover

Perform the ordered crossover in the following fashion:

Algorithm 2 Ordered crossover

```
procedure ORDERED CROSSOVER(permutation1, permutation2)
    Select a random range [i, j] of the first parent's permutation of cities
    Place those (j − i + 1) cities in the same range of the child permutation
    Get the remaining cities from the other parent's permutation in the same order in which they appear
    return the new permutation
```

iii.4 Multithreading

Algorithm 3 Multithreading

```
procedure MULTITHREADING
    Set P, the number of processors to be used.
    Create an array of P threads
    for i ← 0, P − 1 do
        Start the i-th thread
        Let the i-th thread execute threadSolution()
        Let the i-th thread call solve()
        Let the i-th thread print its winning chromosome
```

iv. Time Complexity Analysis

ID	Operation	Complexity
1	Creating C random chromosomes	$\mathcal{O}(C^2N)$
2	Sorting the array of C chromosomes	$\mathcal{O}(C \log C)$
3	Creating $\frac{3C}{4}$ child chromosomes G times	$\mathcal{O}(CNG)$
4	Multithreading	$\mathcal{O}(P)$

Table 1: *Complexity Analysis of the Algorithm*

As we set the amount of chromosome generations G as 10000 and the amount of chromosomes per generation C as 100 in all cases, we can see that operation 3 has the highest time complexity. Thus, $\mathcal{O}(CNG)$ is the complexity of our algorithm, where C, N and G are the amount of chromosomes per generation, the amount of cities in the input and the amount of generations, respectively.

V. Results

i. Solutions found

The following table shows a comparison of the solutions found across four sets of input ranging from 10 to 25 cities each.

Input size (cities)	Min. Distance (km)	
	1 core	4 cores
10	46657.151	46657.151
15	48277.834	48277.834
20	56045.310	56012.943
25	81808.710	70513.449

Table 2: *Comparison of solutions found across all four sets of input*

The following table shows a comparison of the CPU time across four sets of input ranging from 10 to 25 cities each.

Input size (cities)	CPU Time (s)	
	1 core	4 cores
10	17	8
15	19	10
20	22	14
25	25	13

Table 3: *Performance comparison across four sets of input*

ii. Screenshots of the program execution

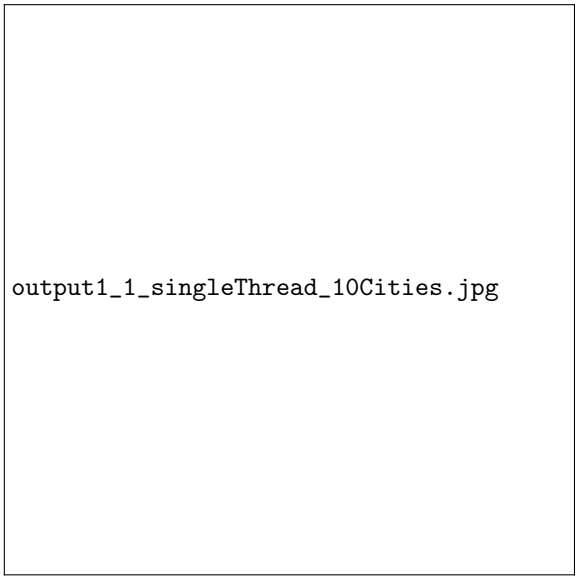


Figure 1: *Output of the single-threaded program with* 10 *cities as input*

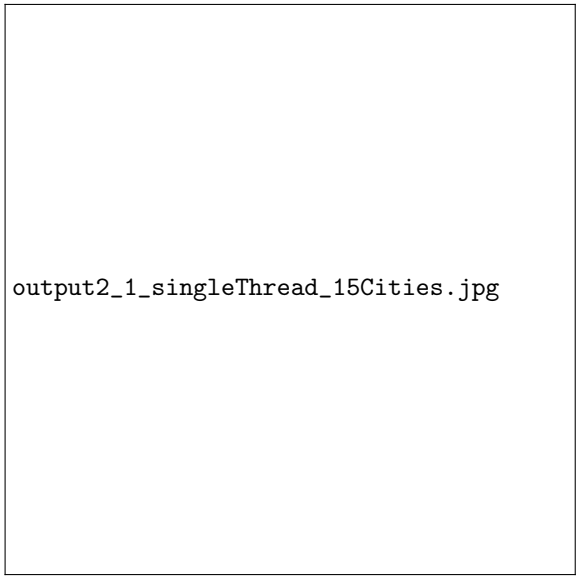


Figure 3: *Output of the single-threaded program with* 15 *cities as input*

Figure 2: *Output of the multi-threaded program with* 4 *cores and* 10 *cities as input*

Figure 4: *Output of the multi-threaded program with* 4 *cores and* 15 *cities as input*

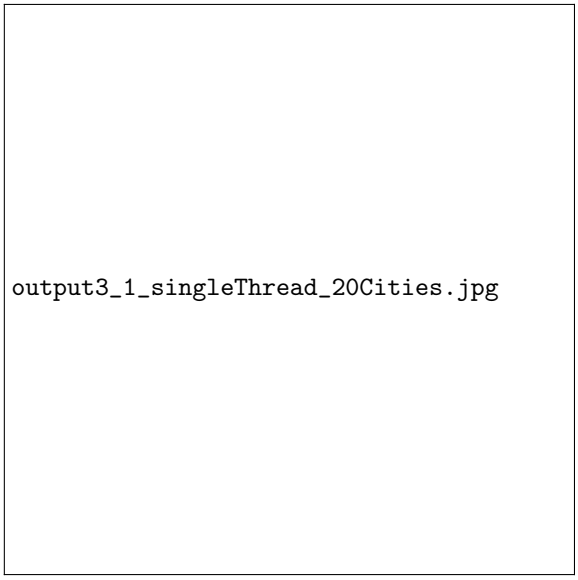


Figure 5: *Output of the single-threaded program with* 20 *cities as input*

Figure 6: *Output of the multi-threaded program with* 4 *cores and* 20 *cities as input*

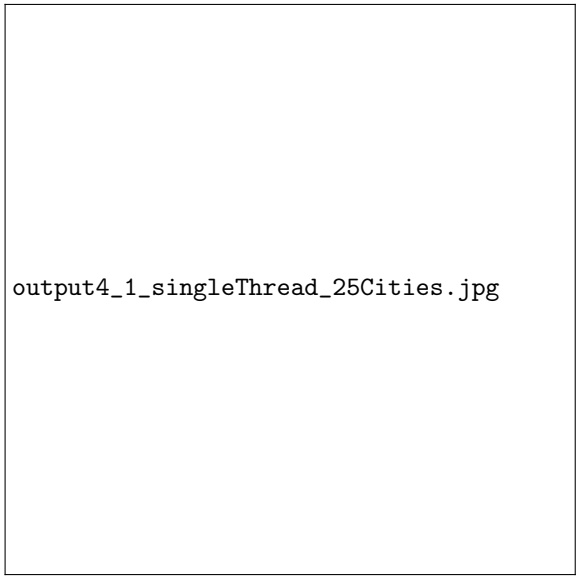


Figure 7: *Output of the single-threaded program with* 25 *cities as input*

Figure 8: *Output of the multi-threaded program with* 4 *cores and* 25 *cities as input*

VI. Discussion

We can see that in all four cases, the multi-threaded approach is around twice as fast as the single-threaded one, taking into account that the former version of the program finds four solutions separately, while the latter only finds one. It is also important to notice that both versions of

the program get the same answers for the first two input cases, whereas the multi-threaded approach clearly beats the single-threaded one for the last two cases.

At first, we hadn't considered rotations of the same cycle as possibilities to reach the same output. However, after a few tests we discovered this, although we also discovered cases in which a two different permutations (none of which is a cycle of the other) reached the same total distance, thus being both valid solutions.

VII. Conclusion

The Traveling Salesman Problem is a very interesting problem not only because it concerns daily tasks in the lives of lots of people, but also because of its difficult nature, yet simple desctription. Although we were a bit afraid to tackle such a problem, we think that, in the end, we developed a solution which could be good enough for general purposes. Nevertheless, if we were to use the program's results for scientific or more serious purposes, we would have to work on the logic of the child generation and the ordered crossover, while also thinking deeper in the amount of generations and the amount of chromosomes per generation.

References

[1] V. Černý. (1985). Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm. JOURNAL OF OPTIMIZATION THEORY AND APPLICATION, 45, 1.

[2] University of Waterloo. (2007). History of the TSP. University of Waterloo www.math.uwaterloo.ca/tsp/index.html

[3] Willi-Hans Steeb. (2005). Genetic Algorithms. The Nonlinear workbook(350.409). London: World Scientific Publishing.