Travelling Salesman Problem

Zihao Hong
Computational Intelligence Group, Departamento de Inteligencia Artificial, Universidad Politécnica de Madrid, Spain

01/07/2020

Abstract

We happened to exist in this world, we happened to acquire knowledge from this world, this world, either perfect or imperfect, there are still immense of things left for us to research and learn. In this paper, we are going to solve the Travelling Salesman Problem (TSP), one of the most studied optimization problem ever designed. The methods we are implementing to resolve are not designed from strict formulas, but from the natural performance of the environment.

1 Introduction

TSP is a NP-Hard optimization problem in which asks the following question:

"Let's say there are N cities **(Nodes)** where all the cities are connected each other **(Edges)** but differs in distance **(Weighted Edges)**, as a travelling salesman, we want to sell our products to all the cities, for reducing the cost, we want to pass through each city only once and the back to our home town **(Hamiltonian path)** and of course, the cost has to be the minimum."

2 Motivation

To resolve the problem, in this paper we do implement two of all the many algorithms that exists at the moment and compare their results.

The inpiration of both algorithms comes from the nature, more specifically from insects: **Ant and Bee**. This makes out a little funny conclusion, why research and think out complex formulas instead of "cheating" from our environment.

3 Data processing

The test data we are extracting are from the following http://www.math.uwaterloo.ca/tsp/data/link (TSP Test Data).

3.1 Input

The input from the test data are the follows:

- **Name**: This name of the file
- **Comment**: Comments for this file
- **Comment**: Comments for this file
- **Type**: Type of the file

- **Dimension**: Number of cities
- **Edge Weight Type**: Method recommended to calculate distances
- **Node Coord Section**: Nodes information, represented as *index, latitude, longitude*
- **EOF**: End of file

3.2 Process

For all the information provided from the file, we only need the nodes information. And the nodes recollected will be structured as the follows:

```
[
    [1, 4], // City 0
    [3, 6], // City 1
    [7, 3], // City 2
    [2, 6], // City 3
    [7, 5]  // ...
]
```

It consists of a tuple of tuples of 2 dimensions where the first value is **latitude** and the second value is **longitude**. And the index is the City i

The graph generated is a complex graph, that means all nodes are connected each other. In order to use the minimum memory, the graph is designed as follows:

```
[
    [],                 // City 0 has no information with the other cities
    [2],                // City 1 has distance 2 with city 0
    [5, 7],             // City 2 has distance 5 with city 0 and distance 7 w
    [5, 8, 3],          // City 3 has distance 5 with city 0 and distance 8 w
    [7, 9, 1, 2],       // ...
]
```

Supposing that the distance between the city A and the city B must be equal to the distance between city B and the city A. Then just need to record one of the edges. Establishing that the node with higher value is the from node.

3.3 Output

The output is an object with two elements, where the first is the total cost of the path, and the second element is a list of tuples with dimension 2, in which inside is also a tuple with dimension 2. The first level tuple indicates the "from" city and the "to" city, while the second level means the "longitude" and "latitude".

```
{
    cost: 20
    path: [         // 2 - 1 - 3 ...
        [
            [7, 3], // City 2
            [3, 6], // City 1
        ],          // Cost 7
        [
            [3, 6], // City 1
            [2, 6], // City 3
```

```
            ],              // Cost 8
            [
                [2, 6],     // City 3
                [1, 4],     // City 0
            ]               // Cost 5
                            // ...
        ]
    }
```

4 Particle Swarm Optimization (PSO)

Inspired from swarm of bees, which looks for polen in locations where the density of flowers are greater.

4.1 Pseudocode

We do identify some parameters which have influence on the computational cost for the later conclusion.

- **X**: Number of iterations
- **N**: Number of cities or nodes
- **B**: Number of population or bees

Here is the high level of the transcription of the implementation of the algorithm.

- Creation of **B** bees.
  - For each bee generated is initialized a solution with **N** nodes.
  - For each bee generated, calculate the solution of **N** nodes.
- For **X** iterations.
  - For each iteration all **B** bees are sorted from minor cost to mayor cost.
  - For each bee **B**.
    - The best personal solution of the bee is switched with the values of the current solution of the bee according to the **N** nodes of the solution.
    - The best global solution of the bee is switched with the values of the current solution of the bee according to the **N** nodes of the solution.
    - For each switch before performed, which could be **2 * N**, is performed in the actual solution of the bee.
- The bees **B** are sorted from minor cost to mayor in order to obtain the best solution.

4.2 Summary of the implementation

The implementation consists of iterations of a process in which involves the bees to find out the best solution, the way of the bee to find out the solution is by comparing its actual solution the its best personal and the best global solution of all bees.

The cost for the solution will be: **((B * 2N) + (X * (B + (B * 4N)) + B)**

Simplified: **X * B * N**

5 Ant Colony Optimization (ACO)

Inspired from ants, which looks for food by sending some "adventure" ants to initialize the path, and leaving more or less pheromones for the other "soldiers" depending on the time it took.

5.1 Pseudocode

We do identify some parameters which have influence on the computational cost for the later conclusion.

- **X**: Number of iterations
- **N**: Number of cities or nodes
- **H**: Number of population or ants

Here is the high level of the transcription of the implementation of the algorithm.

- Initialization of the first path with **N** nodes.
- Establish the initial amount of pheromones to each edge in the complex graph **N * N**.
- Creation of **H** ants.
- For **X** iterations.
  - For each iteration each **H** ant calculates its new path and cost
    - With the same initial node, it goes through all **N** nodes not visited yet
      - Calculate the total probability of the **N** not visited nodes.
      - Choose from **N** nodes not visited the node with the best probability.
  - For each edge of the complex graph **N * N**.
    - Calculate the total pheromone left by ants **H** in this iteration and set to the edges.
- The ants **B** are sorted from minor cost to mayor in order to obtain the best solution.

5.2 Summary of the implementation

The implementation consists of two parts, the first part is the ant to build the path basing on the pheromones distributed, the second part is to distribute the pheromones basing on the ants solutions.

The cost for the solution will be: **((N) + (N * N) + (H) + (X * ((H * N * 2N) + (N * N * H)) + (H))**

Simplified: **X * H * N * N**

6 Conclusions and future research

There is no the best algorithm to solve a problem. For example, depending on how distributed the numbers in an array, an quadratic cost algorithm could be faster than a logarithmic.

By simple comparison of the computational cost between ACO and PSO, its clear that PSO is faster as it only involves 3 level loops, while ACO is 4 level loops. However, by experimenting both algorithms in the program, the variability of the best solution is minor in ACO as computationally it is much slower. So if we do increase the parameters for to be as slower as ACO, it seems that the variability is also less.

Both algorithms have their pros and cons, so it is recommendable to analyse the problem before choosing the algorithm to be implemented.

Something particular I have found during the implementation is that, all these algorithms we are implementing are inspired from nature. But we are implementing them sequentially, so I would like to add another aspect as a proposal, asynchronization.

For example, for ACO, my inspiration for the idea is based on **bash**, we do create an environment of **N** cities or nodes with directories and simbolic links to link each directory. The directories will have the information for the pheromones left by ants, but its calculation is not expected from a formula but for time. While the ant, the could sleep, get lost or find the path by only checking the information in directories. When an ant finishes one round, it will expose its solution to the "queen" and the "queen" stores the best solution.

For problems with greater and greater nodes, edges, etc. Its computational cost is immense. Our goal is not to get the best solution in a period of time, but to get the best solution for the period of time, and the program could keep running in case of a better solution. That is why of adding the aspect of asynchronization. Leaving as to get the solution anytime.

References

[TSP Test Data] Page where test data is extracted: http://www.math.uwaterloo.ca/tsp/data/