Final project - Travelling Salesman Problem

Fanoarii Boyer - Jérémy Beaugeard - Arthur Guerineau - Adrien Le Saux

CIR3 - Graph Theory - Leandro Montero

30 April 2020

Contents

Chapter 1

Introduction

In this final graph theory project, we try to solve the Travelling Salesman Problem (TSP) using different algorithms and heuristics. Before we get to the code, it is important to consider how we are going to model the graphs.

In the TSP, we work with undirected complete graphs, which means they are very dense. Adjacency lists are useful for sparse graphs, but in our case it will be more appropriate, in terms of complexity, to use adjacency matrix.

To implement the adjacency matrix we will use the C++ Boost library. In this project we seek to compare the performance between different solutions to the same problem. As C++ is a low-level programming language it is very fast and will be suitable for our use. Moreover, the boost library is very well known and widely documented.

Chapter 2

Real-life situations

Chapter 3

Exact algorithm

3.1 Pseudo-code

```
Input: $G$ is an undirected complete weighted graph with $n$ vertices
Output: The optimal path found and the corresponding distance
$finalPath \leftarrow empty$
$finalDistance \leftarrow \infty$
$path \leftarrow$ each vertex number from 1 to $n$
$distance \leftarrow 0$
foreach permutation $P$ of path do
    $distance \leftarrow$ total distance of the path
    if $distance < finalDistance$ then
        $finalDistance \leftarrow distance$
        $finalPath \leftarrow P$
    end
end
return $finalPath$ and $finalDistance$
```

3.2 Time complexity

There are $n!$ permutations for a path of size n. Therefore we are going to execute the *ForEach* loop $n!$ times. Moreover, we need to add the weight between each vertex of the path to find the total distance, so we need to do n. As this calculation is inside the *ForEach* loop we obtain an overall complexity of:

$$O(n * n!)$$

This is a brute-force algorithm, which tries out the exhaustive list of possible solutions. It is the most intuitive way to find an exact solution. However using the Held–Karp algorithm or a branch-and-cut algorithm could greatly improve the complexity. Those algorithms are much more advanced and use knowledge linked to dynamic and linear programming. There were therefore too complex to implement in this project.

3.3 Optimal Solution

3.4 Execution time and performance

Chapter 4

Constructive heuristic

4.1 Pseudo-code

```
Input: $G$ is an undirected complete weighted graph with $n$ vertices
Output: The optimal path found and the corresponding distance
$path \leftarrow empty$
$distance \leftarrow 0$
$firstVertex \leftarrow$ choose randomly a vertex in the graph
$currentVertex \leftarrow firstVertex$
Mark $currentVertex$ as discovered
while there are still undiscovered vertices do
    $minimumWeigth \leftarrow \infty$
    foreach adjacent vertex $V$ not discovered do
        $w \leftarrow$ weight between $currentVertex$ and $V$
        if $w < minimumWeight$ then
            $minimumWeight \leftarrow w$
            $nextVertex \leftarrow V$
        end
    end
    $path \leftarrow path + nextVertex$
    $currentVertex \leftarrow nextVertex$
    $distance \leftarrow distance + minimumWeight$
end
$lastWeight \leftarrow$ weight from $currentVertex$ to $firstVertex$
$distance \leftarrow distance + lastWeight$
return path and distance
```

4.2 Time complexity

The first loop makes sure that each vertex of the graph has been discovered, it is executed n times. The second loop tests each undiscovered adjacent vertex, thus there are never more than $n-1$ iterations of this loop and its number decreases as the algorithm reaches the end. Since the three loops are nested and each of them is n iterations or less, we have an overall complexity of:

$$\mathrm{O}(n^2)$$

We could improve the precision of the solution by running the algorithm n times and starting at each vertex of the graph. By doing this we could keep only the best result but the complexity would become $\mathrm{O}(n^3)$. It slightly improve the precision of the solution but greatly decreases the algorithm performance.

4.3 Optimal Solution

This algorithm is an implementation of the Nearest neighbour algorithm;

4.4 Execution time and performance

Chapter 5

Local search heuristic

5.1 Pseudo-code

5.2 Time complexity

5.3 Optimal Solution

5.4 Execution time and performance

Chapter 6

GRASP meta-heuristic

6.1 Pseudo-code

6.2 Time complexity

6.3 Optimal Solution

6.4 Execution time and performance

Chapter 7

Conclusion