# Report

December 2, 2019

## 1 Related work

The first time that Branch and Bound was used for the traveling salesman problem was in the work of Little et al[??]. The lower bound was calculated by reducing the matrix of distances ( which means substracting the smallest element of a row for every element of that row and then doing the same for the columns ). The amount of reduction constitutes the lower bound and each time that we choose a new edge, we update the matrix and we reduce again. They found that the computation time grows exponentially.

## 2 Branch And Bound

*Overview* :
We want to find a sequence of cities with a an order that minimize the length of the path.
The solution is then in the form of a list where each adjacent element are connected and where the last element of the list is connected with the first element. The final list has to be of size N ( the number of points in the plane ). We could have tested all the permutations possible but it would have taken too long time ( N! permutations ). That is why we are using Branch And Bound that is aimed to explore all possible permutations that are relevant.
We begin with a root node ( a list with only one point ), and then we expand the tree by adding a new point to the list. We calculate the lower bound of this new list and if the lower bound is superior to the current solution, we prune the subtree by removing the last point added. If the lower bound is inferior to the current solution, we recursively expand the problem. If we find a new solution better than the current solution, then this solution becomes the current solution.
We use the Minimum Spanning Tree ( calculated with the Kruskal's algorithm ) to find the lower bound of the partial solution.

*Strengths and Weaknesses* :

Branch and Bound ensures an exact solution if we wait long enough because we try every relevant potential solution. However, if this approach takes a lot of time and has a bad time complexity.

*Data structure used* :
- an input array Nx2 that represents the list of cities and their coordinates.
- a NxN matrix that represents the distances between each pair of points.
- a NxN matrix where each row represents the list of points sorted according to their distances to a specific point ( each row represents one point ).
- a dynamic list that represents the solution. Each adjacent element are connected and the last element of the list is connected with the first element.
- a list that contains every edges so that we can sort them for the Kruskal's algorithm. There is also a dictionary so that we can find an edge thanks to its distance.
- a dictionary that stores the past Minimum Spanning Trees already calculated so that we don't calculate them a second time.

*Time and space complexities* :
In order to get the time complexity we have to take into account the worst case. And the worst case is a case here we explore every possible solution. Each solution is a permutation of the N points. Then there are N! permutations possible. And then the time complexity is factorial $O(n!)$ .
For the space, we need to store the current and the partial solution which has a complexity $O(n)$ but we also need to store all the distances between every pair of points. As a result, the space complexity of the algorithm is $O(n^2)$. However, if we take into account our strategy to store the results of minimum spanning trees, the space complexity becomes exponential.

*Problem expansion for Branch and Bound* :
The first question is : How do we expand a subproblem ? If we are currently have a partial solution $[P_1, ..., P_i]$ ( of size inferior to N ), we need to add a new city to the list.
We select firstly the cities that are closest to the last point ( $P_i$ here ) and which are not yet in the list. That is why we calculated a matrix that represents for each point its closest neighbors.
In fact, this is a greedy choice that is aimed to save us computation time as we explore firstly the more relevant potential solutions.

*Tactics to reduce the computational time* :
We used the Minimum Spanning Tree because it seemed to be best option to have the highest lower bound ( we use the Kruskal's algorithm ). Also, without any strategies, we can compute the same Minimum Spanning Tree multiple times. For instance, for both the partial solution [1,2,3,4] and [2,4,3,1], we are going to compute the same Minimum Spaninng Tree with the other points. That's why we store the results for each combination of points so that we can save computation time.

Pseudocode :

**Algorithm 1** Branch and Bound

```
1: function RESEARCH(PartialSolution)
2:     if length(PartialSolution)=N then
3:         Sum=CalculateSum(PartialSolution)
4:         if Sum < Minimum then
5:             Minimum=Sum
6:         end if
7:     end if
8:     for Point ∈ RankedNeighbors(PartialSolution[−1]) do
9:         if Point ∉ PartialSolution then
10:            PartialSolution.add(Point)
11:            LowerBound=MinimumSpanningTree(PartialSolution)+CalculateSumPartial(PartialSolution)
12:            if LowerBound < Minimum then
13:                RESEARCH(PartialSolution)
14:            end if
15:            PartialSolution.pop()
16:        end if
17:    end for
18: end function
```

# 3 Empirical evaluation

Before using the Minimum Spanning Tree, we tried to calculate the lower bound with the matrix reduction method. However, this method takes much more time. For instance, with this method we had to wait 10 minutes to have the exact solution for Atlanta.tsp whereas we only need to wait 21 seconds with the Minimum Spanning Tree. Also we can note that we can find the actual results for small datasets ( Atlanta, Cincinnati, UKansasState ) but it fails for bigger datasets. However, we always have a relative error inferior to 20%. Also, the fact that we store in memory the minimum spanning trees already calculated gives a great boost on our computation time. For instance, for Atlanta, I get the optimal solution in 750 seconds without storing the MSTs whereas I can get the solution in 22 seconds with memory. In the same way, for Cincinnati we need to wait 0.27 second without memory and 0.03 with memory.