CS3P01 Parallel and Distributed Computing

Final Project

Diego Canez
Maria Lovaton

Friday 30th July, 2021

Contents

1 Introduction

The Travelling Salesman Problem is a well-known NP-hard combinatorial optimization problem with many applications in the real world. In its standard form, the task is the following: Given a source node, find a circuit that starts at the source node, traverses all nodes once and has minimum weight. For implementation purposes, we've chosen to generalize this problem by considering a source and target vertices[1].

Our library consists on a sequential and a parallel implementation of the TSP, using OpenMP. It is accompanied by a humble suite of tests and benchmarks using the Catch2 Library, and an interactive visualization program that allows the user to perform a TSP on a chosen subset of the following districts of Lima: Lima Centro, Lince, Miraflores, Barranco, Rimac, Los Olivos, La Molina, La Victoria, Magdalena, San Borja.

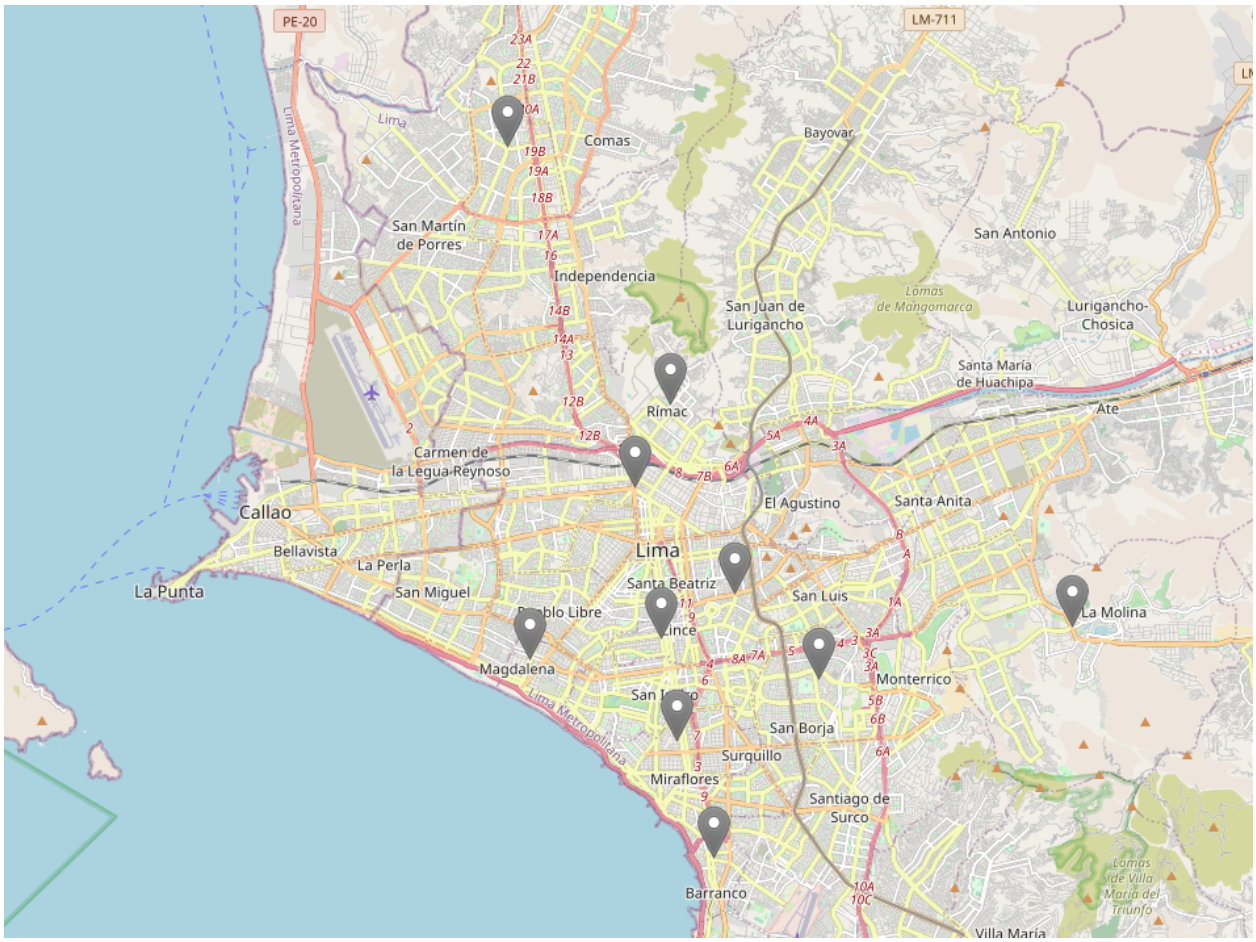


Figure 1: Districts of Lima

[1]To use our algorithm as a subprocedure for solving the classic TSP, one would only need to add a dummy node with the same connections as the source node and fix it as the target node.

2 Methods

Both implementations of the TSP are based on a simple recursive branch and bound algorithm. For example, Figure 2 shows the application of the algorithm and the resulting state-space tree.

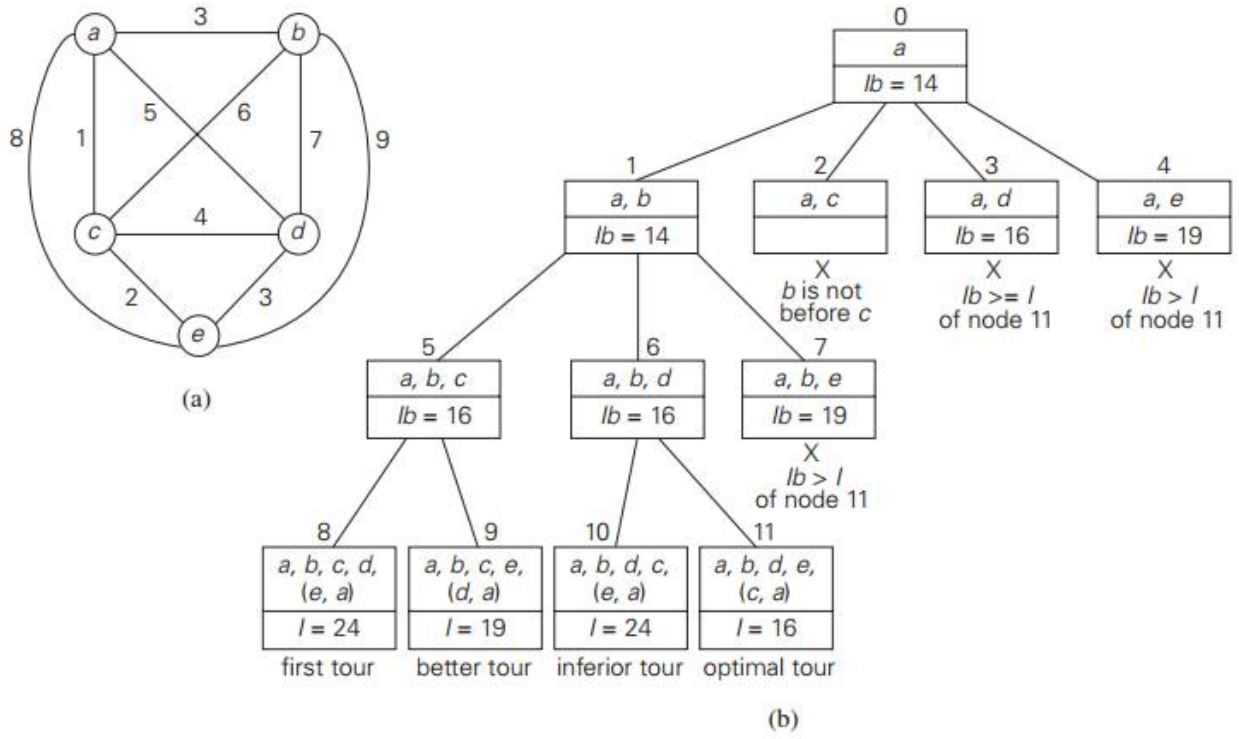


Figure 2: (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find the shortest Hamiltonian circuit in this graph.

Some details of our implementation are the following:

- The recursion tree is stored explicitly to avoid keeping the current order for each recursion call. For the sequential case, at most $O(n)$ nodes are active at any time. This change is specially useful for a seamless parallelization. For the parallel case, at most $O(n \times p)$ nodes are active at any time.

- Since n must be small for algorithm to be tractable, bitmasks are used to store important $O(n)$ data in $O(1)$ memory.

- The template class `DenseGraph` provides a simple interface for any type of edge weight, including `std::tuples` and other comparable containers. A specific application for this could be a multi-parameter TSP, for example `(distance, time, fuel wasted)`.

- A task creation threshold is set based on the number of threads available and the current depth in the recursion tree.

Algorithm 1 Sequential TSP Algorithm

```
1: function SolveTsp(srcNode, dstNode)
2:     bestRoute ← {}
3:     bestCost ← inf
4:     function Explore(node, route)
5:         for every nonvisited neighbour of node do
6:             newRoute ← route + { neighbour }
7:             if newRoute.cost < bestCost then
8:                 Explore(neighbour, newRoute)
9:             end if
10:        end for
11:        if node == dstNode then
12:            bestRoute ← route
13:        end if
14:    end function
15:    Explore(srcNode, { srcNode })
16:    return bestRoute
17: end function
```

Algorithm 2 Parallel TSP Algorithm

```
1: function SolveTsp(srcNode, dstNode)
2:     bestRoute ← {}
3:     bestCost ← inf
4:     function Explore(node, route)
5:         for every nonvisited neighbour of node do
6:             newRoute ← route + { neighbour }
7:             if newRoute.cost < bestCost then
8:                 #pragma omp task
9:                 Explore(neighbour, newRoute)
10:            end if
11:        end for
12:        #pragma omp taskwait
13:        if node == dstNode then
14:            #pragma omp critical
15:            if route.cost < bestCost then
16:                bestRoute ← route
17:            end if
18:        end if
19:    end function
20:    #pragma omp parallel
21:    #pragma omp single nowait
22:    Explore(srcNode, { srcNode })
23:    return bestRoute
24: end function
```

Algorithm	Time Complexity	Memory Complexity
Sequential	$O(n!)$	$O(n)$
Parallel	$O(\frac{n!}{p})$	$O(n \times p)$

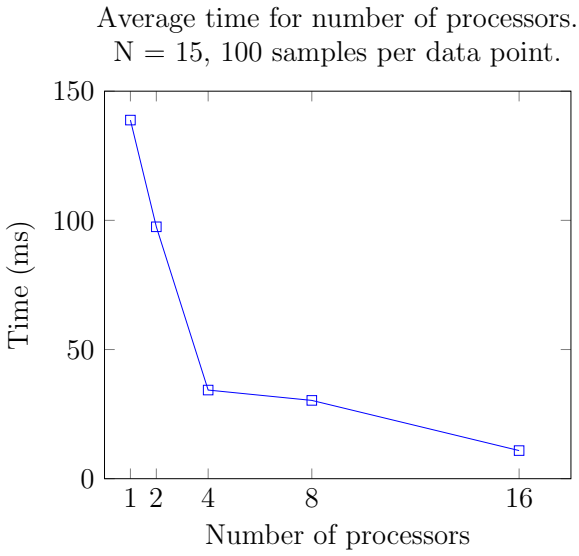
Table 1: Time and memory complexity

Regarding the complexity analysis of the algorithm, we claim *at least* the bounds in Table 1.

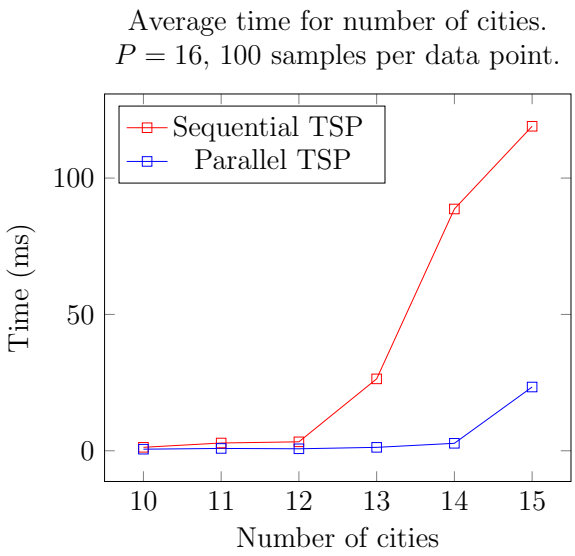
However, this should be a pessimistic bound, since the branch and bound algorithm favors tremendously from faster state search. A way to think about this is the following: The faster the algorithm finds new optimal paths, the more aggressively it can prune non-optimal ones. Therefore, we hypothesize the speedup should be greater than linear. Of course, the efficiency is constant so this algorithm is highly scalable.

3 Results

(a) Figure 3a



(b) Figure 3b



As one can see in figure 3b, the speedup is $O(p)$ (in this case $p = 16$). The decrease in time per number of processors shown in Figure 3a also seems to be linear on the number of processors.

4 Conclusions

In conclusion, we found that the parallel branch and bound TSP favours greatly from an increased number of threads, but we couldn't find strong proof for our hypothesized super-

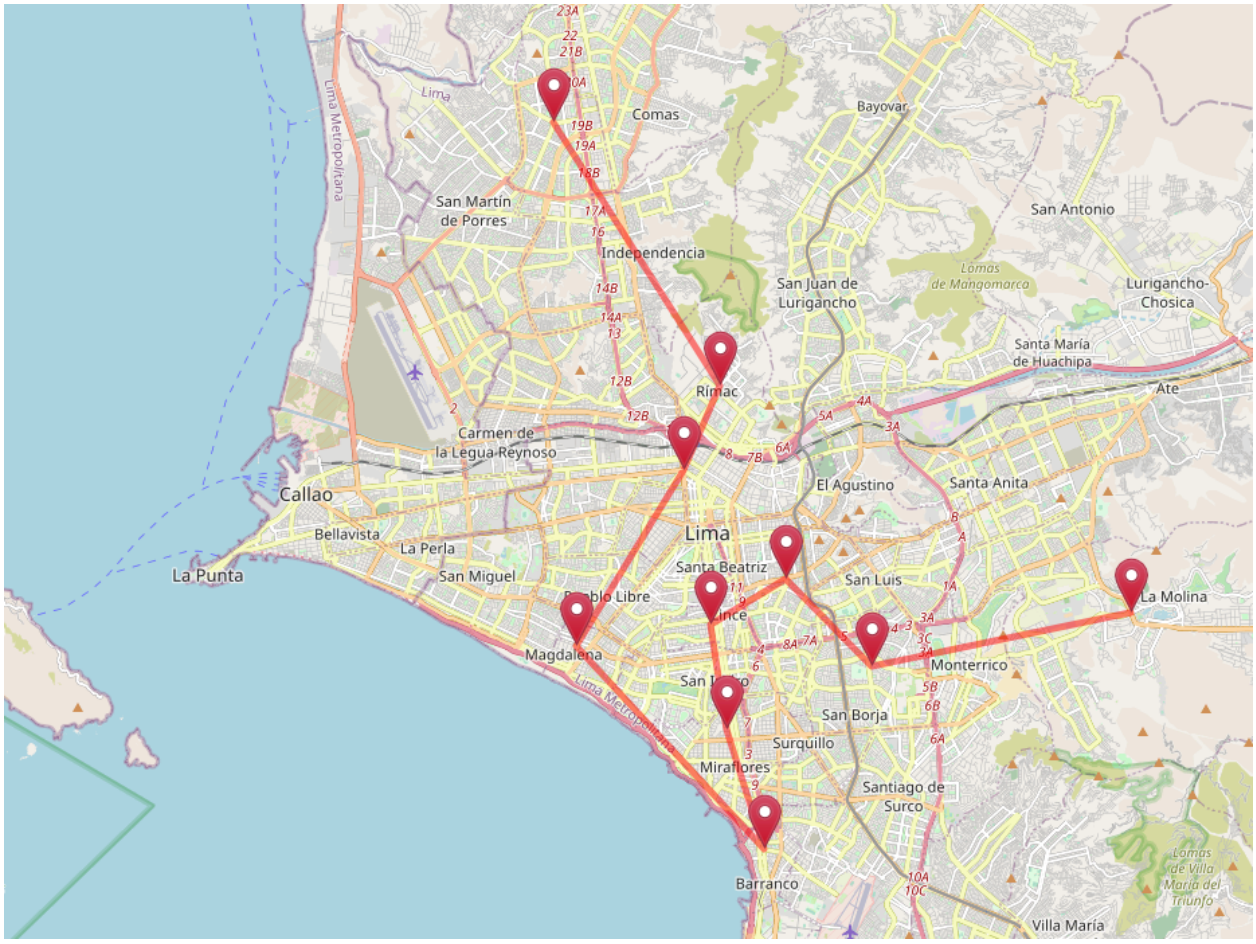


Figure 4: TSP answer from Los Olivos to La Molina

linear speedup. It is left to see if stronger heuristics and bounds could make this superlinear trend more noticeable. The correctness of the algorithm can be shown in Figure 4.

The github repository can be found at **dgcnz/cs3p01-proyecto**.