Advanced Algorithms

Assignment 2:
Traveling Salesman Problem

May 16, 2022

Budai Matteo	2057217
Burke Jamie	2044062
Tlepin Sanjar	2041606

Contents

1 Introduction

For this assignment, we implemented and analyzed the running times and the errors of three algorithms for the Traveling Salesman Problem. The algorithms implemented are:

1. Nearest Neighbor Algorithm
2. Random Insertion Algorithm
3. 2-approximate Algorithm

2 Nearest Neighbor Algortihm

For the first constructive heuristic algorithm we have chosen Nearest Neighbor. The goal of the algorithm is to start from a single-node path, and then add the neighbor with the minimum weight not in the existing path. The steps are the following:

1. **Initialization:** start from the single-node path 0;
2. **Selection:** let (v0, ..., vk) be the current path. Find the vertex vk+1 not in the path with minimum distance from vk;
3. **Insertion:** insert vk+1 immedaitely after vk;
4. repeat from (2) until all vertices are inserted in the path.

2.1 Functions

The implementation doesn't require particular data structure so we created two functions for the algorithm:

- **NearestNeighbor()**: method that builds the Hamiltonian circuit and returns its weight and the execution time;
- **minDist(p)**: method that finds the neighbor with the minimum weight between the last vertex insert in the path p and its neighbors that are in some indexes of the list g.

2.2 Implementation

The final solution is the following:

- We start the time;
- We execute a deep copy unvisited of the original list v;
- We follow the algorithm and we start from the node 0 *(startingNode)* of the unvisited list;
- We remove the starting node from the unvisited list and we add always the starting node in our path *(is a list)*;
- We initialize our solution *(dist)* to zero and we start our cycle until we have visited all vertices *(len(unvisited)>0))*;
- In the while cycle:
    1. We find the neighbor with the minimum distance with the function minDist:
        - It initalize the distance to infinite, the vertex to none and it take the last node of the path *(p)*;
        - In the cycle it checks all the neighbors of the node and returns the distance and the vertex with the minimum distance that is not just in p.

2. We remove the vertex found from the list unvisited and we add the vertex in the path;
3. We add the distance of the vertex found and we continue with the next iteration until we removed all the vertex from unvisited.

- When we finish the while cycle we add to dist the distance between the last node in the path and the startingNode and to close the Hamiltonian cycle we add the vertex to the path;
- We end the time and we calculate the time cost;
- We return the solution and the time.

3 Random Insertion Algorithm

Idea of the algorithm is to take randomly a node and find the cheapest path going through that random k from pair of nodes in our partial circuit. As it was mentiond erlier, the graph initialization functions create graph as a list of edges and also provide us a list of all nodes. The Implementation of the code follow the based concept:

- **Initialization**: Take first node (we always took first node from the graph and make it as partial circuit;
- **Selection**: Random select node not in partial circuit;
- **Insertion**: Find edge in partial circuit that minimize the triangle inequality: w(i,k)+w(k,j)-w(i,j) and insert k between i and j;
- **Repeat**: step **Selection** and **Insertion** until add all nodes to partial circuit.

3.1 Functions

We implement 3 functions:

- **RandomInsertion()**: Main function that call other in process of counting the triangle inequality and check the distance.
- **buildPath(p,k)**: Function to get the distance required to get to random k and also position after we need to insert.
- **minDist(u)**: Function to get closest node to input one.

3.2 Implementation

The final solution is the following:

- We start the time;
- We execute a deep copy unvisited of the original list v;
- We follow the algorithm and we start from the node 0 *(startingNode)* of the unvisited list;
- We remove the starting node from the unvisited list and we add always the starting node in our path *(is a list)*;
- We found the vertex j that minimize w(0,j) with the function *minDist* and we remove it from the unvisited list and add it to the path;
- We add starting node to the end because it should be a circle list;
- We initialize our solution *(dist)* to the distance found with the function *minDist* counted twice and we start our cycle until we have visited all vertices *(len(unvisited)>0))*;
- In the while cycle:
    1. We take the random node from the unvisited list;

2. We call the function *buildPath*:
    - It initalize the distance to infinite;
    - In the cycle it checks for all elements of our path what is the position where the random node k minimize the total weight *"min(w(i,k)+w(k,j)-w(i,j))"*;
    - It returns the position where we have to insert the node and the distance required to add the random node.
3. We remove the random node from the unvisited list and we add it in the path;
4. We update the distance and we add the random node in the correct position.

- When we finish the while cycle we end the time and we calculate the time cost;
- We return the solution and the time.

Regarding originality, we implemented a method in the minDist function that runs through a slice of the graph rather than through the entire graph, which ultimately makes the function more efficient.

4 2-approximate Algorithm

The 2-approximate algorithm starts by defining a starting node from the list of vertices within a graph, and then a minimum spanning tree is constructed using Prim's algorithm with the starting node as the root. The vertices of the MST are then visited in a preorder walk/depth first search, and added to a list in the order of which they are visited. Finally, the starting node is added at the end of the preorder list in order to complete the Hamiltonian cycle in the MST.

```
1 APPROX_METRIC_TSP (G)
2     V = { v_1, v_2, ..., v_n }
3     r ← v_1
4     T* ← PRIM (G, r)
5     < v_i1, v_i2, ..., v_in >= H' ← PREORDER (T*, r)
6     return < H', v_i1 > = H
```

4.1 Input and Data Structure

The 2-approximate algorithm requires a minimum spanning tree to be constructed using Prim's algorithm, and in order to be efficient, Prim's algorithm requires a heap data structure. We reused Prim's algorithm and the min heap data structure that were implemented in Assignment 1 for this course; however, the input files for Assignment 1 provided a list of connected nodes, whereas the input files for Assignment 2 only provide a list of distinct vertices and their x and y coordinates. Consequently, we made the following modifications to the Graph and Node classes, and added two new functions:

- **Node**: three new instance variables are initialized:
  - xcoord: x-coordinate of vertex
  - ycoord: y-coordinate of vertex
  - weighttype: weight type of the vertex, either Euclidean or Geographic
- **Graph**: the buildGraph function restructures the input file to resemble the input file format of Assignment 1, before calling the createNodes and makeNodes functions. Specifically, the buildGraph function takes each vertex and connects it to all other vertices in the input file.
- **Weight**: new function that takes the weight type, x-coordinate, and y-coordinate for two vertices as input, and returns the weight between the two vertices. For the Euclidean weight type, there is no coordinate conversion and we simply calculate the Euclidean distance rounded to the nearest integer. For the Geographic weight type, the x- and y-coordinates are converted to radians and then the geographic distance is calculated using the instructions in the FAQ site provided by the assignment.
- **Preorder**: new function that takes a MST graph, starting node, and empty set as input. The vertices of the MST are visited in a preorder walk/depth first search and

added to the list in that order. The list is returned once all vertices of the MST have been visited and added to the list.

4.2 Implementation

The solution to the cost of the Hamiltonian cycle in the MST is performed in the following steps:

1. Create the Graph object through the Graph() class and call the buildGraph method
2. Calculate the MST of the graph using Prim's algorithm
3. Call the Preorder function, passing the inputs of the MST and the same starting node used as the root in Prim's algorithm.
4. Append the starting node to the end of the list returned by the Preorder function in order to complete the Hamiltonian cycle in the MST
5. Call the Weight function to calculate the distance between each node in the Hamiltonian cycle, sum all of the edge weights to calculate the total cost of the cycle

5 Results

5.1 Table of results

Instance	Nearest Neighbor			Random Insertion			2-Approximation		
	Solution	Time (s)	Error (%)	Solution	Time (s)	Error (%)	Solution	Time (s)	Error (%)
tsp_dataset/burma14.tsp	4048	0.00005	21.82%	3336	0.00009	0.39%	4003	0.00011	20.46%
tsp_dataset/ulysses16.tsp	9988	0.00006	45.62%	7076	0.00010	3.16%	7788	0.00013	13.54%
tsp_dataset/ulysses22.tsp	10586	0.00012	50.95%	7342	0.00019	4.69%	8308	0.00022	18.47%
tsp_dataset/eil51.tsp	511	0.00044	19.95%	471	0.00093	10.56%	605	0.00076	42.02%
tsp_dataset/berlin52.tsp	8980	0.00045	19.07%	8252	0.00088	9.41%	10402	0.00080	37.92%
tsp_dataset/kroA100.tsp	27807	0.00125	30.66%	23796	0.00261	11.81%	30516	0.00251	43.39%
tsp_dataset/kroD100.tsp	26947	0.00141	26.55%	23543	0.00287	10.56%	28599	0.00257	34.31%
tsp_dataset/ch150.tsp	8191	0.00272	25.47%	7327	0.00609	12.24%	9126	0.00595	39.80%
tsp_dataset/gr202.tsp	49336	0.00613	22.85%	43574	0.00906	8.50%	52615	0.01018	31.01%
tsp_dataset/gr229.tsp	162430	0.01025	20.67%	151426	0.01156	12.50%	179335	0.01320	33.23%
tsp_dataset/pcb442.tsp	61979	0.04577	22.06%	57296	0.05349	12.84%	72853	0.04758	43.47%
tsp_dataset/d493.tsp	41660	0.05979	19.02%	38368	0.06851	9.62%	45595	0.06093	30.26%
tsp_dataset/dsj1000.tsp	24630960	0.15553	32.00%	20921807	0.34281	12.12%	25526005	0.32737	36.80%

Figure 1: Table of results

5.2 Graph of the execution time comparison

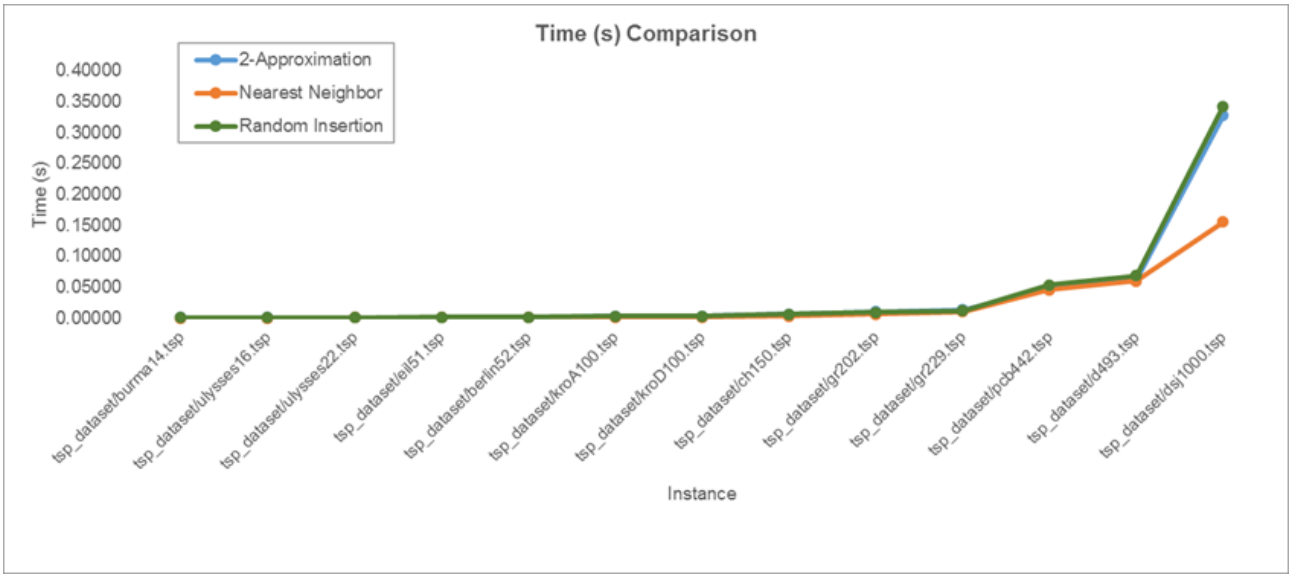

Figure 2: Execution time comparison

5.3 Graph of the error comparison

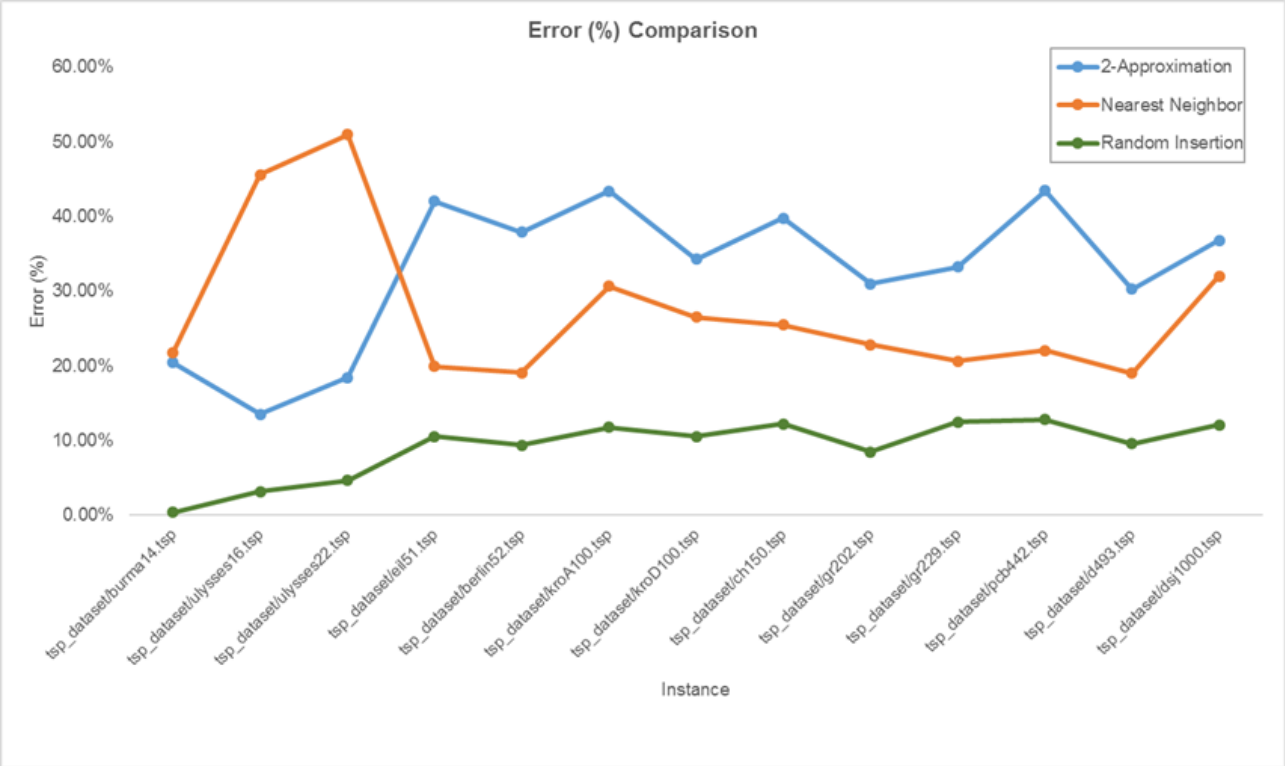

Figure 3: Error comparison

6 Conclusion

From the Error Comparison graph and Results table, we initially observe for the instances less than 50 coordinates the approximation error of nearest neighbor grows rapidly as compared to random insertion and 2-approximate. However, as the number of coordinates grows past 50, nearest neighbor outperforms 2-approximate, and eventually each algorithm has a steady approximation error. 2-approximate hovers around 40%, nearest neighbor around 20%, and random insertion around 10%.

From the Time Comparison graph and Results table, we can observe that for instances less than 493 coordinates, all three algorithms have similar execution times. At 1000 coordinates, the execution times of nearest neighbor and 2 approximate algorithms increase to around 30 seconds, while the random insertion algorithm only increases to 15 seconds.

With respect to the approximation error, the random insertion algorithm overwhelmingly out-performs nearest neighbor and 2-approximate.

Considering both the approximation error and the execution time, we observe that the random insertion algorithm is the most efficient.