Trabalho DA2023_PRJ1_G16_8

Realizado por:

- Tiago Cruz (up202108810)
- João Lourenço (up202108863)
- Tomás Xavier (up202108759)

Files

To facilitate the development of this project, we created 6 different header files, with 4 new classes total:

- Graph.h – In this file, we define the Graph class, which is the building block for every method written.
- NodeEdge.h – In this file, we define both the Node and Edge classes, used to represent the data provided in the dataset.
- parse.h – This is an auxiliary file used to read and parse the data in the csv files.
- calculations.h – This is an auxiliary file used for small yet useful calculations, such as converting degrees to radians, for example.
- print.h – small file with only a function, used to print the calculated path .
- UFDS.h – Represents the Union-Find Disjoint Set, required by the Kruskal's algorithm.

Problem Data and Basic Interface

To read and parse the provided data set files we created 9 methods total in the read.h header file. Because not all the graph provided have the same structuring and format, we needed to create functions more specific regarding the type of graph, hence the large number of functions needed just for parsing.

1. read – Receives a line of text and parses it to correctly divide it and insert the right values into the right fields (both Node and Edge).

2. isAlreadyInEdges – This method iterates over the vector of edges passed as parameter and returns true if it finds the id passed as parameter in the destination of one of the edges.

3. fillInBlanks – Iterates through every node and checks if there is an edge that connect that node with every other node, if not then creates an edge between the nodes and calculates the distance between them with the haversine formula. Used to make a real-world graph fully-connected.

Problem Data and Basic Interface

4. fillInBlanksWithOne – Essentially the same thing as the previous function, except this one is used for the toy graphs; therefore, the distance is not calculated through the haversine formula, it's set to 1.

5. readRealWorldNodes – parses the file line by line and inserts the info read into the graph as a node. Used for the real-world graph files, since their format is different from the other ones.

6. readRealWorldEdges – The same as the previous function, except this one inserts into the graph the info read as an edge. Also used for the real-world graphs.

7. readToyGraph – Also the same as the readRealWorldNodes, except the file read by this function includes both the nodes and edges, therefore it's inserted into the graph both the nodes' information and edges' as well. Used for the toy graph.

Problem Data and Basic Interface

8. readToyGraphWith1Distance – The same as the previous function, except this function calls the fillInBlanksWithOne function, used to make the graph fully-connected, with 1 distance between the edges.

9. readExtraFullyConnectedGraph – Also essentially the same as the previous read functions, except this one is used for the extra fully connected graphs.

Calculations.h

1. convertToRadians – Function that converts the value given in degrees to radians.
2. haversineDistance – Function that utilizes the haversine formula to calculate distance between 2 points using their coordinates.
3. calculateMean – Function that calculates the mean of the vector provided as parameter.
4. calculateStandardDeviation – Function that calculates the standard deviation of the vector provided as parameter.

Print.h

Print.h consists of a small auxiliary function called **printPath** used for printing the calculated path for the TSP. It receives a vector containing the path and its weight and prints it to the terminal in a more user-friendly and easier to understand way.

Backtracking algorithm

For the backtracking algorithm, we created the tspBT function. This function sets every node's visited field as false and returns the result of the tspBTRec function. The tspBTRec solves the tsp recursively by checking if the cost of the path followed in the current iteration is better than the current best possible path. If so, then that path becomes the current new best path, along with its cost. If not, then the function backtracks and tries to find a different path. The function will continue until there are no more possible paths to follow, where it will end and return the best path and its cost.

Bounding techniques

As bounding techniques, we sort the edges by their weight and, on each recursion, we check if node 0 still has edges whose destination node has not yet been visited. Furthermore, before calling a recursion we check if following said path will lead to a worse solution than the one we already have, if so then we don’t follow that path.

Triangular approximation heuristic

The function we used for this segment is called **triangularApproximationHeuristic**, which will firstly create an MST of the graph, using the Kruskal algorithm. Then it will visit the MST in pre-order and if it finds a node which is not a part of the path it adds that node to the path and, because we assume the graph is fully-connected, we add the distance between the last added node and the current node to be added to the path. Finally, because we begin with the node 0, the node 0 will be the last node of the path as well, and we add the distance between the last added node and the node 0. At this point, the path is concluded, being stored by reference in an array. Then the function returns the path's cost.

Other heuristics

We decided, for this exercise, to implement the K-means algorithm, which is a clustering algorithm, along with a divide-and-conquer approach, this is done in the **kMeansDivideAndConquer** function. Meaning, we start by creating √NodeSet.size centroids (if there are 100 nodes, then 10 centroids will be created). Then the clustering algorithm will find which nodes fit better for each centroid, using the coordinates to determine the distance between the nodes and the centroids. The nodes closer to a centroid form a cluster. Then we recalculate the position of the centroid based on the positions of the nodes inside that centroid's cluster. We repeat this process until there is no change to the centroid's position. After this, we call the function recursively for each centroid until the cluster's size is less than or equal to 3, or if the standard deviation is less than or equal to 10% of the mean of the distance of the nodes to that cluster's centroid, which means that the nodes are similarly distanced so that we can use the triangular approximation method to formulate a path between them, utilizing the same function developed for the previous exercise **triangularApproximationHeuristic**.

Other heuristics (cont.)

Finally, with every cluster having its own path, we merge the paths of different clusters by first locating the best nodes to connect them, then we iterate through one of the paths until it finds the node calculated previously where the two paths will be connected. We repeat this process of merging the clusters' paths until the path encapsulates the entirety of the graph's NodeSet. In the end, because we wanted the path to begin and end with the node 0, we took it out of the whole clustering ordeal. Therefore, after having the final path, we call the joinSolvedTSP (the function that merges the clusters' paths) with the final path and a vector with only the node 0. This way, its like merging 2 different paths, except one of them only has one node, the node 0. Then the **kMeansDivideAndConquer** function returns the path's cost, which we print to the terminal, along with the path.

Other heuristics note

Because our kMeans clustering algorithm relies on the coordinates of each node to group them into clusters, the algorithm does not work for the Extra Fully-Connected and toy graphs, since those graphs did not provide the coordinates of each node. Instead, the algorithm was designed to work for the Real-World Graph 1, since this graph is not only fully-connected, but also provides the longitude and latitude of each node.

Backtracking plot execution

Fun fact: Our algorithm would take 54 googol milliseconds to finish (54*10^100) with the edges75 graph (75 nodes and 2775 edges).

Calculation:

Knowing that the toyshipping graph (14 nodes and 34 edges) took 125ms to finish and the time-complexity of the algorithm is O((V-1)! * E), then:

execution_time = 0.125 * ((74!*2775)/(13!*34))

execution_time = 5.42*10^98 seconds

Backtracking vs Triangular Approximation Heuristic

- **toy shipping**

Backtrack: 145ms (Minimum total distance value: 86.7)

Triangular: 1ms (Minimum total distance value: 126.1)

- **toy stadiums**

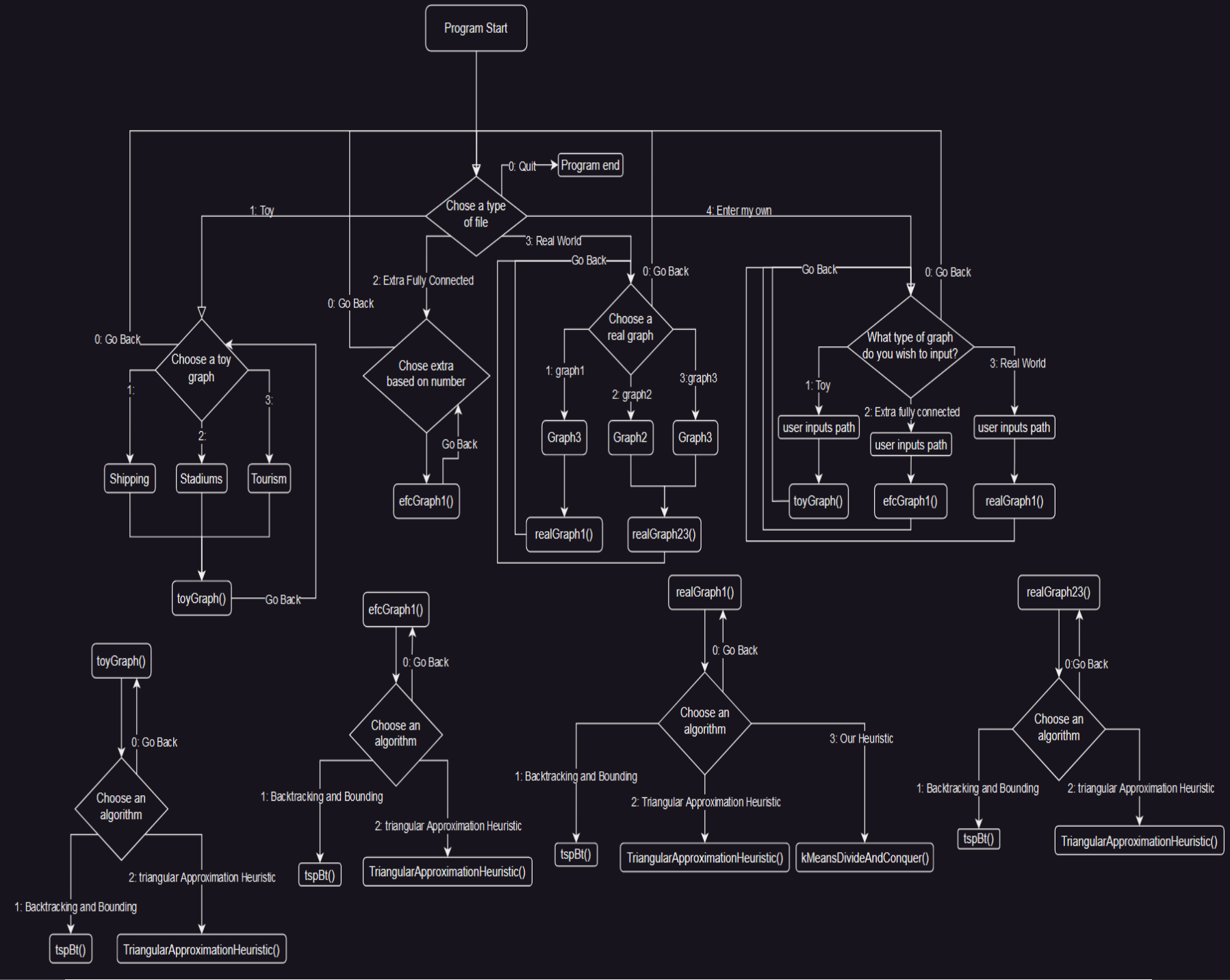
Backtrack: 1346 ms (Minimum total distance value: 341)

Triangular: 0ms (Minimum total distance value: 398.1)

- **Tourism**

Backtrack: 0ms (Minimum total distance value: 2600)

Triangular: 0ms (Minimum total distance value: 2600)



MENU DIAGRAM