PROGRAMMING PROJECT II

Routing Algorithm for Ocean Shipping and Urban Deliveries

>> DA - June 2023 - Group G09_1

OUR GROUP

Rúben Correia Pereira

up202006195

Tomás Ballester Carvalho Sousa Pereira

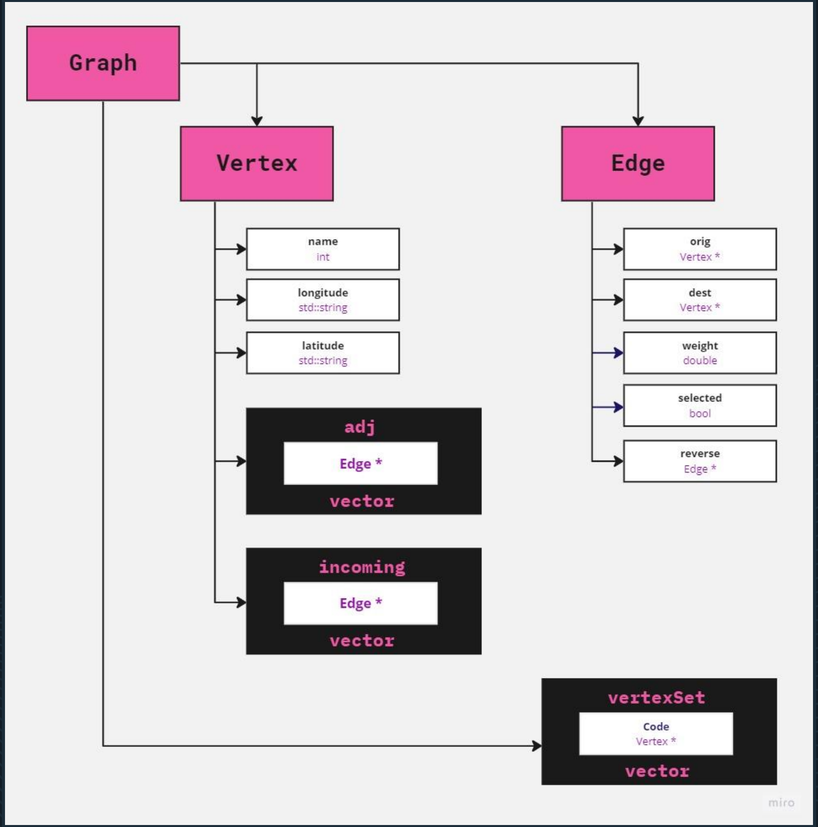
up202108812

Simão Queirós Rodrigues

up202005700

CLASS DIAGRAM

We have 3 classes implemented into our project: Graph, Vertex and Edge.



INITIAL MENU:
Choosing the graph

```
----------------------------------------------------------------
            Please select the desired graph to test:

     [>]     Toy-Graphs
         [1]     shipping
         [2]     stadiums
         [3]     tourism

     [>]     Real-World Graphs
         [4]     graph1
         [5]     graph2
         [6]     graph3

     [7]     Extra-Fully-Connected Graphs

     [>]     Other files...
         [8]     Using 1 file
         [9]     Using 2 files

     [0]     Exit

----------------------------------------------------------------

Enter an option:
```

In the initial menu, you are asked to choose one of the following **default** options of graphs or you can also choose **other** graphs by inputting their paths (if it has **one or two** files).

INITIAL MENU:

Choosing the graph

```
Enter an option: 8

Path of the desired .csv file: ../data/Toy-Graphs/tourism.csv
Parsed!
```

Example: [>] Other Files > [8] Using 1 file

In this case we used the **tourism.csv** file from the **Toy-Graphs** to make sure the program would parse the data if we provided its file path, which worked. Then you could just choose other files if you provide **a valid path**!

```cpp
bool Graph::parser_onefile(const string& file_p){
    fstream file;
    file.open( s: file_p, mode: ios::in);
    if(!file.is_open()){
        return false;
    }
    string ori,dest,distance;
    getline( &: file, &: ori);
    if(count( first: ori.begin(), last: ori.end(), value: ',')==2) {
        while (getline( &: file, &: ori, delim: ',')) {
            getline( &: file, &: dest, delim: ',');
            getline( &: file, &: distance);
            if (findVertex( name: stoi( str: ori)) == nullptr) {addVertex( name: stoi( str: ori));}
            if (findVertex( name: stoi( str: dest)) == nullptr) {addVertex( name: stoi( str: dest));}
            addBidirectionalEdge( sta_A: stoi( str: ori), sta_B: stoi( str: dest), cap: stod( str: distance));
        }
    }
    else if(count( first: ori.begin(), last: ori.end(), value: ',')==4){
        string label_o,label_d;
        while (getline( &: file, &: ori, delim: ',')) {
            getline( &: file, &: dest, delim: ',');
            getline( &: file, &: distance, delim: ',');
            getline( &: file, &: label_o, delim: ',');
            getline( &: file, &: label_d);
            if (findVertex( name: stoi( str: ori)) == nullptr) {addVertex( name: stoi( str: ori));}
            if (findVertex( name: stoi( str: dest)) == nullptr) {addVertex( name: stoi( str: dest));}
            addBidirectionalEdge( sta_A: stoi( str: ori), sta_B: stoi( str: dest), cap: stod( str: distance));
        }
    }
    file.close();
    printf( format: "Parsed!");
    return true;
}
```

Parser function using one file

```cpp
bool Graph::parser_twofiles(const string& edges_p, const string& nodes_p){
    fstream edges_f,nodes_f;
    nodes_f.open( s: nodes_p, mode: ios::in);
    if(!nodes_f.is_open()){
        return false;
    }
    string ori,lon,lat;
    getline( &: nodes_f, &: ori);
    if(count( first: ori.begin(), last: ori.end(), value: ',')==2) {
        while (getline( &: nodes_f, &: ori, delim: ',')) {
            getline( &: nodes_f, &: lon, delim: ',');
            getline( &: nodes_f, &: lat);
            addVertex( name: stoi( str: ori), longitude: stod( str: lon), latitude: stod( str: lat));
        }
        nodes_f.close();
        edges_f.open( s: edges_p, mode: ios::in);
        if(!edges_f.is_open()){
            return false;
        }
        string dest,dist;
        getline( &: edges_f, &: ori);
        while (getline( &: edges_f, &: ori, delim: ',')) {
            getline( &: edges_f, &: dest, delim: ',');
            getline( &: edges_f, &: dist);
            addBidirectionalEdge( sta_A: stoi( str: ori), sta_B: stoi( str: dest), cap: stod( str: dist));
        }
        edges_f.close();
    }
    printf( format: "Parsed!");
    return true;
}
```

Parser function using two files

SECOND MENU:

Select the algorithm

Backtracking

The backtracking algorithm systematically explores possible solutions, making choices and backtracking when paths are not optimal or dead ends are reached. It efficiently prunes the search space, helping find viable solutions for problems with specific conditions or constraints.

Triangular Approximation

The TSP approximation algorithm based on triangular inequality ensures a 2-approximation solution. It utilizes geographic node data, starts and ends the tour on the zero-labeled node, and efficiently selects and backtracks using the triangular inequality. Comparing it with the backtracking algorithm on small graphs provides insights into their performances and solution strategies.

Other Heuristic

For the other heuristics section, we though of implementing the Christofides algorithm, a renowned TSP heuristic, ensures a 1.5-approximation solution. It constructs a minimum spanning tree and finds a minimum-weight perfect matching. By combining these structures, the algorithm produces near-optimal solutions for TSP. Sadly, **we couldn't**.

SECOND MENU:

[Select the algorithm]

BACKTRACKING

```
Enter an option: 2
Parsed!

----------------------------------------------------------------
              Please select the desired algorithm:

      [1]     Backtracking
      [2]     Triangular Approximation
      [3]     Christofides Algorithm

      [0]     Change Graph
----------------------------------------------------------------

Enter an option: 1
Minimum Distance: 341
Execution Time: 4.108 seconds
Press any key to continue . . .
```

Example: [>] Toy-Graphs > [2] stadiums

We used the **stadiums.csv** file from the **Toy-Graphs** to compare the two algorithms we've implemented. This took **4.108 seconds** and gave us **341 meters** as the minimum distance.

```cpp
void Graph::backtrack_tsp(){
    clock_t start = clock();

    double min_cost = std::numeric_limits<double>::max();
    std::vector<int> path = {0}; // Start from vertex 0
    std::vector<bool> visited(getNumVertex(), false);
    visited[0] = true; // Mark the starting vertex as visited

    // Call the recursive function to find the minimum cost Hamiltonian cycle
    backtrack_tsp_rec(path, visited, min_cost, 0.0);

    clock_t end = clock();

    std::cout << "Minimum Distance: " << min_cost << std::endl;
    std::cout << "Execution Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds" << std::endl;

}
void Graph::backtrack_tsp_rec(std::vector<int>& path, std::vector<bool>& visited, double &min_cost, double cost_so_far) {
    if (path.size() == vertexSet.size()) {
        int start_vertex = path.front();
        int last_vertex = path.back();
        for (auto edge : vertexSet[last_vertex]->getAdj()) {
            if (edge->getDest()->getName() == start_vertex) {
                double cycle_cost = cost_so_far + edge->getWeight();
                if (cycle_cost < min_cost) {
                    min_cost = cycle_cost;
                }
                break;
            }
        }
        return;
    }

    int last_vertex = path.back();
    for (auto edge : vertexSet[last_vertex]->getAdj()) {
        if (!visited[edge->getDest()->getName()]) {
            path.push_back(edge->getDest()->getName());
            visited[edge->getDest()->getName()] = true;
            backtrack_tsp_rec(path, visited, min_cost, cost_so_far + edge->getWeight());
            path.pop_back();
            visited[edge->getDest()->getName()] = false;
        }
    }
}
```

Backtracking main function and its recursive function.

SECOND MENU:

[Select the algorithm]

TRIANGULAR APPROXIMATION

```
Enter an option: 2
Parsed!

----------------------------------------------------------------
             Please select the desired algorithm:

     [1]     Backtracking
     [2]     Triangular Approximation
     [3]     Christofides Algorithm

     [0]     Change Graph
----------------------------------------------------------------

Enter an option: 2
Minimum Spanning Tree:
0 - 1
0 - 2
0 - 3
5 - 4
10 - 5
10 - 6
5 - 7
4 - 8
6 - 9
2 - 10
Path: 0 -> 3 -> 2 -> 10 -> 6 -> 9 -> 5 -> 7 -> 4 -> 8 -> 1 -> 0
Optimal cost: 391.4 meters
Execution Time: 0.025 seconds
Press any key to continue . . .
```

```cpp
void Graph::triangularApproximation() {
    clock_t start = clock();

    std::vector<int> parent(vertexSet.size(), -1);
    primMST(parent);


    std::vector<bool> visited(vertexSet.size(), false);
    std::vector<int> path;
    std::stack<int> cityStack;
    dfs(0, parent, visited, cityStack, path);

    std::cout << "Path: ";
    for (int i = 0; i < path.size(); ++i) {
        std::cout << path[i];
        if (i != path.size() - 1)
            std::cout << " -> ";
    }
    std::cout << " -> 0"<< std::endl;

    double total_distance = calculateTotalDistance(path);
    clock_t end = clock();

    std::cout << "Optimal cost: " << total_distance << " meters" << std::endl;
    std::cout << "Execution Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds" << std::endl;
}
```

Example: [>] Toy-Graphs > [2] stadiums

Triangular Approximation main function.

This took **0.025 seconds** and gave us **391.4 meters** as the minimum distance, a lot faster than the last algorithm.

THE END

Thank you!

Presentation Template: SlidesMania