TRAVELING SALESMAN PROBLEM

Possible solutions

INTRODUCTION

In this project, we attempted to find the most efficient route for a salesman to visit a set of cities, exploring many strategies which offer different solutions with varying degrees of effectiveness.

CLASSES

```
class Script{
private:
  Graph<int> * shipGraph =
new Graph<int>();
Graph<int> * stGraph = new
Graph<int>();
Graph<int> * tmGraph = new
Graph<int>();

Graph<int> * rwg = new
Graph<int>();
Graph<int> * efcg = new
Graph<int>();

}
```

```
class Menu {
private:
   Graph<int> *g;
   Script script;
   TSPSolver tspSolver;
(...)
}
```

```
struct Cluster {
   double centerX;
   double centerY;
   std::vector<int> cities;
};

class TSPSolver{
private:
   Script script;
(...)
}
```

INFORMATION PARSING

show public methods of Script.h

INFORMATION PARSING

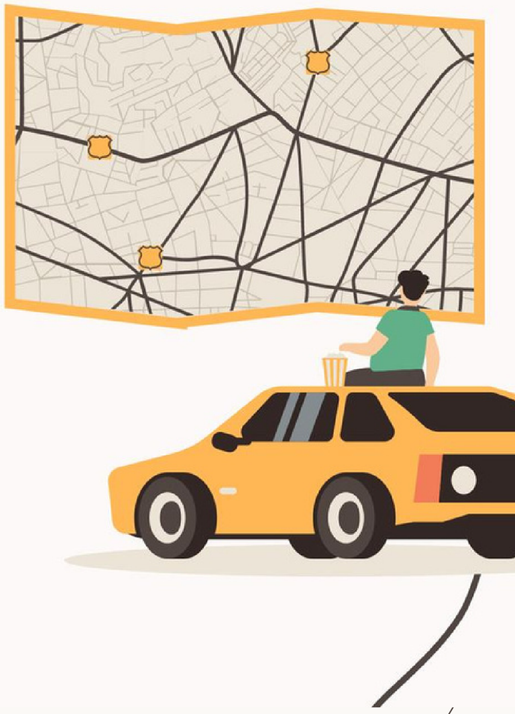
Toy Graphs:

- Each toy graph is parsed by 3 diferent functions.

Real World Graphs and Extra Fully Connected Graphs:

- Both graphs are parsed in one function that receives 2 strings (one with the nodes path and other with the edges path)

```
void Script::read_stadiums() {
    ifstream File1("../cmake-build-debug/datasets/toyGraphs/stadiums.csv");
    if (File1.is_open()){
        string line;
        double distance;
        int origin, destination;
        getline(File1, line);

        while(getline(File1, line)){
            istringstream iss(line);
            string orig, dest, dist;

            getline(iss, orig, ',');
            getline(iss, dest, ',');
            getline(iss, dist, ',');

            origin = stoi(orig);
            destination = stoi(dest);
            distance = stod(dist);

            stGraph->addVertex(origin, 0.0, 0.0);
            stGraph->addVertex(destination, 0.0, 0.0);
            stGraph->addEdge(origin, destination, distance);
            stGraph->addEdge(destination, origin, distance);

        }
    }
    File1.close();
}
```



USER INTERFACE

Firstly, the user can choose in which graph a certain algorithm is applied - **chooseGraph( )**.

Then, the algorithm the user wishes to see executed can also be chosen - **mainMenu( )**.

Used algorithms

01

BACKTRACKING

Tries all possible tours and find best one

02

TRIANGULAR APPROXIMATION

Approximates the solution within 1.5 times the optimal solution

03

DIVIDE IN CLUSTERS AND CONQUER

Solves the TSP for each cluster and combines all tours

04

LOWER BOUND TSP

Always visits the nearest unvisited city

05

NEAREST NEIGHBOR

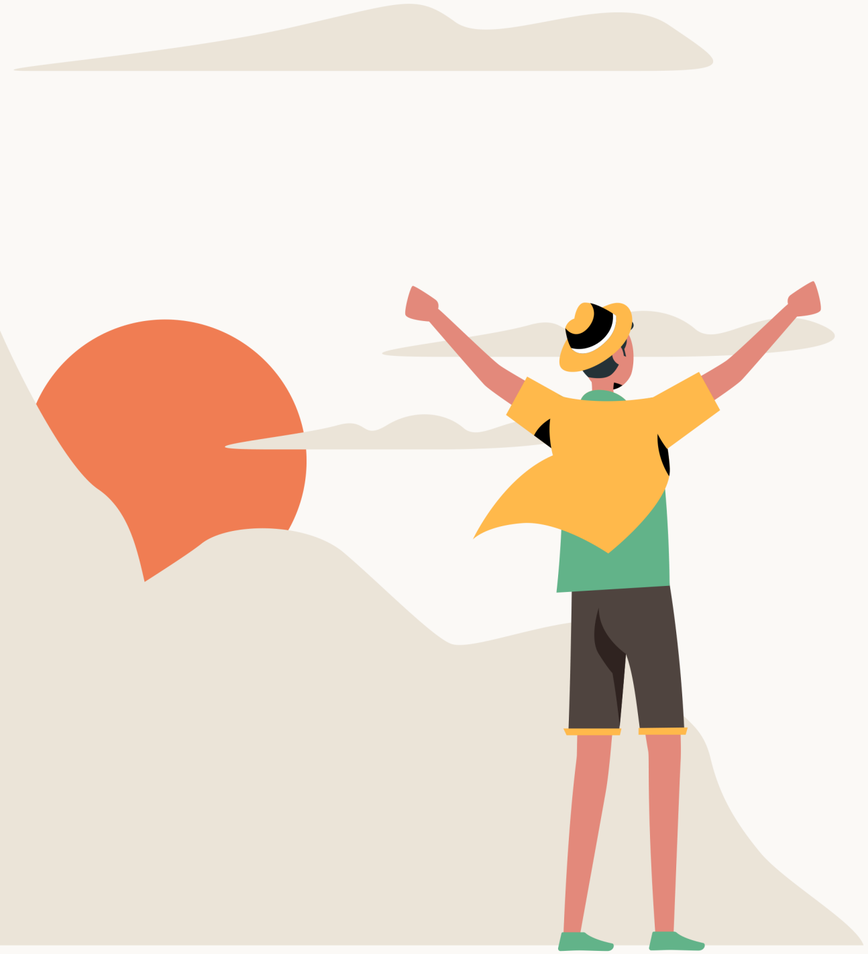
Always visits the nearest unvisited city

BACKTRACKING

Best tour out of all of them

- Exhaustively searches all possible tours to find the optimal one.
- Starts from an initial city and recursively visits all that have not been visited yet.
- With the total distance for each tour, it tracks the minimum distance found and the corresponding tour.

TRIANGULAR APPROXIMATION



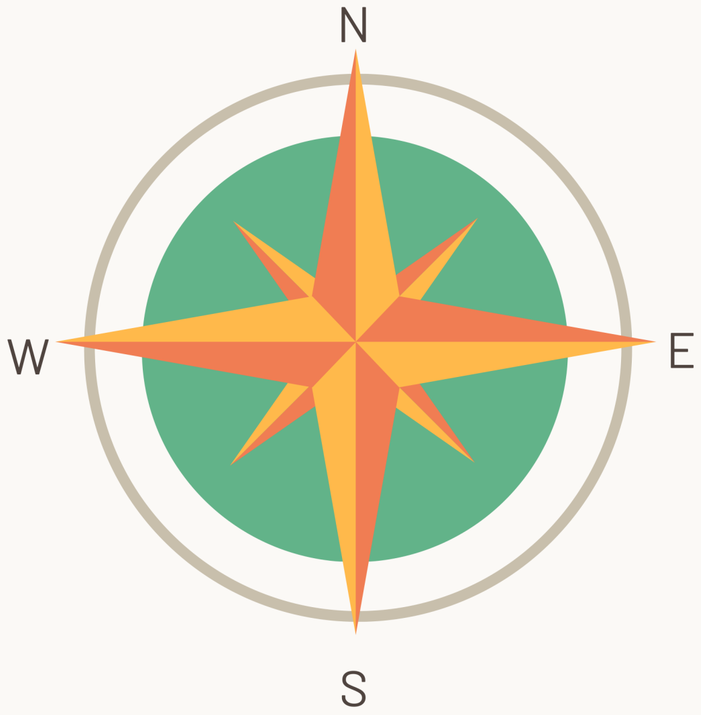
Provides performance guarantee

- Finds the Minimum Spanning Tree .
- Traverses the tree using Preorder Walk.
- Adds Back to the Starting Vertex.

DIVIDE AND CONQUER

Solves TSP for each cluster

- Divides the cities into clusters and uses triangular approximation solution for each.
- Combine the tours of each cluster to form a single tour.



LOWER BOUND TSP

Solves TSP for both dense and sparse graphs

1st Step: findArticulationPoints:

- This function finds articulation points in the given graph using depth-first search (DFS).
- It initializes num and low arrays with -1 values.
- It iterates over each vertex of the graph and calls dfsArticulationPoints if the vertex hasn't been visited yet.
- After DFS traversal, it checks if the root vertex (index 0) is an articulation point by counting its children.
- Finally, it resets the parent pointers of all vertices and returns the set of articulation points found.

Solves TSP for both dense and sparse graphs

2nd Step: mstAndPrim:

- This function should compute the minimum spanning tree (MST) of the given graph using Prim's algorithm.
- It initializes keys for each vertex with infinity except for the source vertex.
- It uses a priority queue to select the next vertex to expand based on the minimum key value.
- It updates keys and parent pointers while traversing the graph.
- It calculates the total weight of the MST and populates the mST vector with the edges of the MST.
- Finally, it returns the total weight of the MST.

Solves TSP for both dense and sparse graphs

3rd Step: tSPTreeLowerBound:

- This function calculates the lower bound for the traveling salesman problem (TSP) using the minimum spanning tree (MST) approach.
- It finds articulation points and skips them during traversal.
- For each non-articulation point vertex, it calculates the MST and selects two shortest edges not adjacent to the vertex.
- It computes the lower bound by adding the weights of the two shortest edges and updates the best lower bound if necessary.
- It returns the best lower bound found.

NEAREST NEIGHBOR

Always visits the nearest unvisited city

- At each visited city, moves to the nearest unvisited city.
- Repeats until all cities are visited.
- Returns to the starting city to complete the tour.

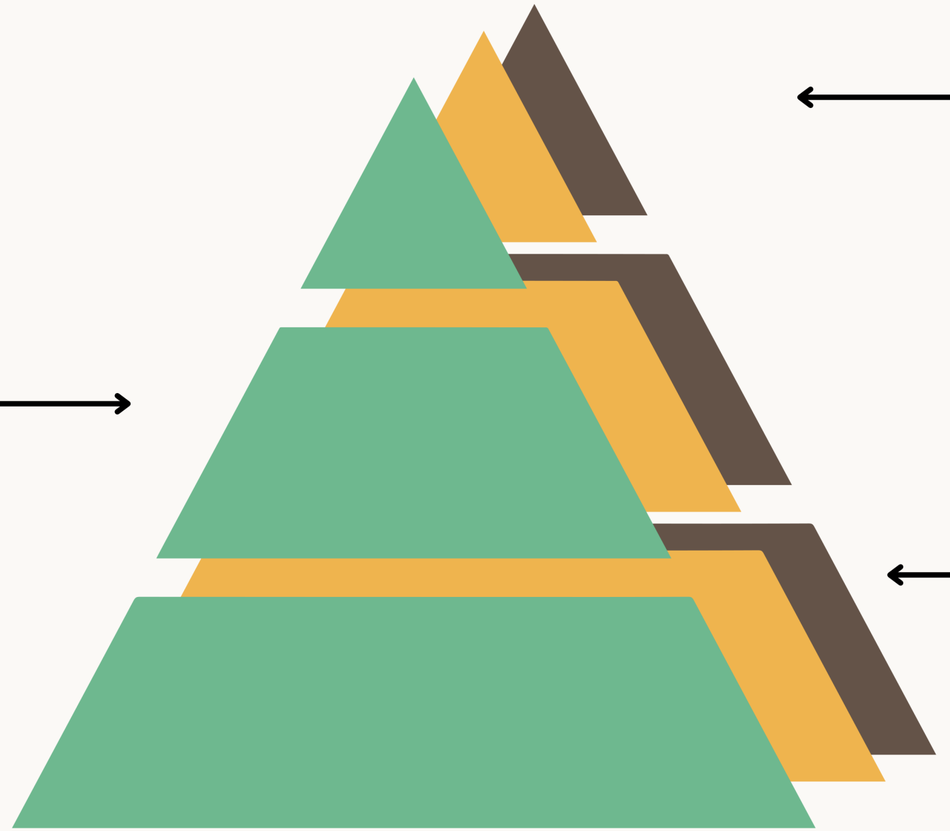
```
class Menu {
private:
    Graph<int> *g;
    Script script;
    TSPSolver tspSolver;
public:
    Menu(TSPSolver tspSolver, Script script);
    void mainMenu();
    void chooseGraph();
    void printToyGraph();
    void printRealWorldGraph();
    void printExtraFullyConnected();
};
```

SOME RESULTS -- TOY GRAPHS

ALGORITHM	Shipping	Stadiums	Tourism
Bactracking	86.7	341	2600
Triangular	------	398.1	2600

PROBLEMS

1- Interpretation of the statement and difficulty in finding viable solutions to the problem presented



2- Another difficulty experienced was the fact that we had to make the program in the most efficient way conceivable (made possible by our data structures).

3- Time Management

OUR TEAM

Bernardo Costa

up202207579

Diana Nunes

up202208247

Teresa Mascarenhas

up202206828