DA Project 2 - TSP

Class 17, Group 5

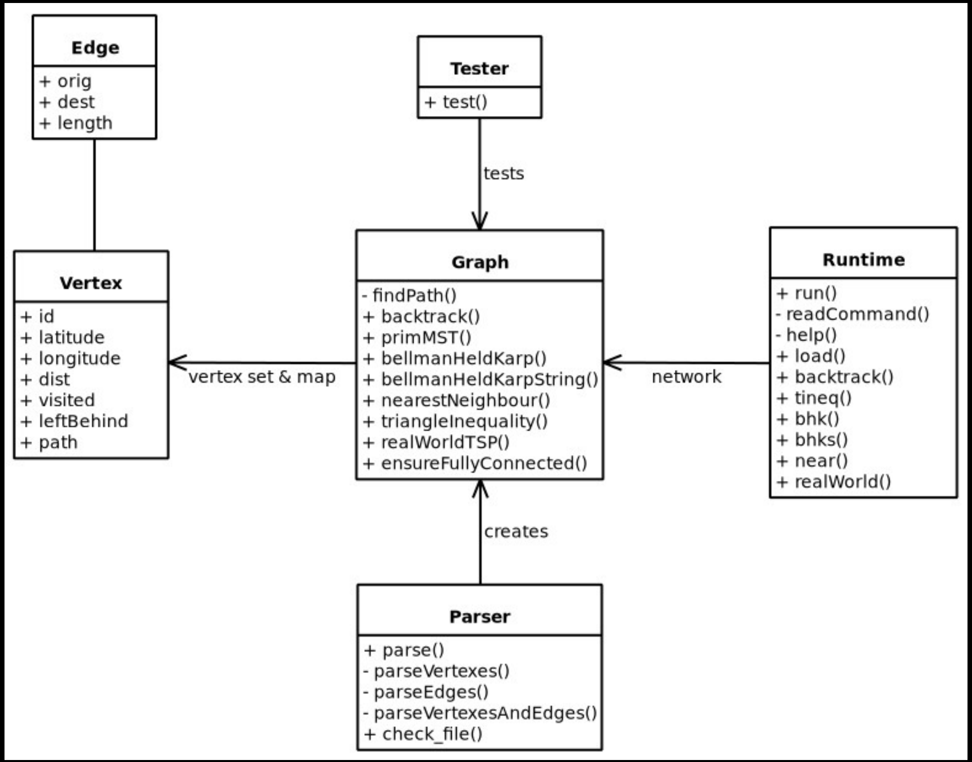
Introduction

- This project consists of a simulation of the famous Travelling Salesperson Problem
- This problem consists of finding the minimal Hamiltonean cycle, i.e., given a set of nodes, go through each node once and only once and then return to the starting one
- In order to solve said problem, we implemented a graph, which will soon be described
- Group members:

Duarte Assunção	up202208319	33.(3)%
Pedro Gorobey	up202210292	33.(3)%
Lucas Bessa	up202208396	33.(3)%

How the Graph Works

- The Graph is a very basic one, consisting of nodes (vertexes) connected by undirected weighted edges
- In the diagram class, each class is in the files with the same name, one ending with .cpp and the other with .hpp



File Structure

- csv/ - CSV files with Graph data
- doc/ - Documentation folder
- lib/ - Header files
- src/ - Source files
- build.ninja - Compilation file
- CMakeLists.txt - Compilation file
- Doxyfile - Documentation file
- LICENSE - The Unlicense
- Makefile - Compilation file
- README.md - Project description

```
.
├── build.ninja
├── CMakeLists.txt
├── csv
│   ├── Extra_Fully_Connected_Graphs
│   │   ├── edges_100.csv
│   │   ├── edges_10.csv
│   │   ├── edges_15.csv
│   │   ├── edges_200.csv
│   │   ├── edges_20.csv
│   │   ├── edges_25.csv
│   │   ├── edges_300.csv
│   │   ├── edges_400.csv
│   │   ├── edges_500.csv
│   │   ├── edges_50.csv
│   │   ├── edges_5.csv
│   │   ├── edges_600.csv
│   │   ├── edges_6.csv
│   │   ├── edges_700.csv
│   │   ├── edges_75.csv
│   │   ├── edges_800.csv
│   │   ├── edges_900.csv
│   │   └── nodes.csv
│   ├── load_commands.txt
│   ├── Real-world Graphs
│   │   ├── graph1
│   │   │   ├── edges.csv
│   │   │   └── nodes.csv
│   │   ├── graph2
│   │   │   ├── edges.csv
│   │   │   └── nodes.csv
│   │   └── graph3
│   │       ├── edges.csv
│   │       └── nodes.csv
│   └── Toy-Graphs
│       ├── shipping.csv
│       ├── stadiums.csv
│       └── tourism.csv
```

```
├── doc
│   ├── Doxyfile
│   ├── PRJ2.odp
│   ├── PRJ2.pdf
│   ├── PRJ2.pptx
│   └── Project2Description.pdf
├── lib
│   ├── Auxil.hpp
│   ├── Edge.hpp
│   ├── GraphException.hpp
│   ├── Graph.hpp
│   ├── Macros.hpp
│   ├── Parser.hpp
│   ├── Runtime.hpp
│   ├── Tests.hpp
│   └── Vertex.hpp
├── LICENSE
├── Makefile
├── README.md
└── src
    ├── Auxil.cpp
    ├── Edge.cpp
    ├── Graph.cpp
    ├── main.cpp
    ├── Makefile
    ├── Parser.cpp
    ├── Runtime.cpp
    ├── Tests.cpp
    └── Vertex.cpp
```

Parsing (1/2)

- This function is called when the user loads the graphs. For more information on this loading, please consult the README.md file.

```cpp
//! Time Complexity: O(t), Space Complexity: O(t), t - total number of lines
void parse(Graph &network, const std::string& edges_csv, const std::string& vertexes_csv) {
    std::ifstream Edge(edges_csv);
    std::ifstream Vertex(vertexes_csv);
    if (check_file(edges_csv, Edge)) {
        if (vertexes_csv.empty()) {
            parseVertexesAndEdges(network, edges_csv);
            std::cout << WHITE << std::setw(25) << "Parse Vertexes and Edges " << std::setw(18) << edges_csv;
            std::cout << GREEN << " OK" << WHITE << '\n';
        } else if (check_file(vertexes_csv, Vertex)) {
            unsigned int index = edges_csv.find_last_of('_');
            unsigned int limit;
            try {
                // try to get number of nodes
                limit = std::stoi(edges_csv.substr(index + 1, edges_csv.length() - 4 - index));
                std::cout << "Found a number after the last '_'! Will only read " << limit << " nodes" << std::endl;
            } catch (std::invalid_argument &e) {
                // if it failed to get number of nodes, just read them all
                limit = UINT_MAX;
            }
            parseVertexes(network, vertexes_csv, limit);
            std::cout << WHITE << std::setw(17) << "Parse Vertexes " << std::setw(45) << vertexes_csv << GREEN << " OK" << '\n';
            std::cout << "Number of nodes: " << network.getVertexSet().size() << std::endl;
            parseEdges(network, edges_csv);
            std::cout << WHITE << std::setw(17) << "Parse Edges " << std::setw(45) << edges_csv;
            std::cout << GREEN << " OK" << WHITE << '\n';

            size_t n_edges = 0;
            for (auto v : network.getVertexSet())
                n_edges += v->getAdj().size();
            std::cout << "Number of edges: " << n_edges << std::endl;
        } else
            std::cout << "The file " << vertexes_csv << " is invalid\n";
    } else
        std::cout << "The file " << edges_csv << " is invalid\n";
}
```

Parsing (2/2)

- This fetches the data from the respective CSV file and stores it in temporary variables used to create the Vertex
- A parsing function might execute solo (if both edges and nodes are derived from the same file) or in conjunction with another one

```cpp
//! Time Complexity: O(V), Space Complexity: O(V)
void parseVertexes(Graph &network, const std::string& file, unsigned int limit) {
    std::ifstream fs(file);
    if (check_file(file, fs)) {
        check_first_line(fs);
        std::string id, latitude, longitude;
        while (getline(fs, id, ',') && limit > 0) {
            getline(fs, latitude, ',');
            getline(fs, longitude);
            network.addVertex(new Vertex(std::stoul(id), std::stod(latitude), std::stod(longitude)));
            limit--;
        }
    }
}

//! Time Complexity: O(E), Space Complexity: O(E)
void parseEdges(Graph &network, const std::string& file) {
    std::ifstream fs(file);
    if (!check_file(file,fs)) return;
    check_first_line(fs);

    std::string src, dest, dist;
    while (getline(fs, src, ',')) {
        getline(fs, dest, ',');
        getline(fs, dist);
        Vertex* a = network.findVertex(std::stoul(src));
        Vertex* b = network.findVertex(std::stoul(dest));
        a->addEdge(b, std::stod(dist));
        b->addEdge(a, std::stod(dist));
    }
}

//! Time Complexity: O(E), Space Complexity: O(E + V)
void parseVertexesAndEdges(Graph &network, const std::string& file) {
    std::ifstream fs(file);
    if (!check_file(file,fs)) return;
    check_first_line(fs);

    std::unordered_map<unsigned long, Vertex*> vertexes;
    std::string src, dest, dist, line;
    while (getline(fs, line)) {
        std::istringstream iss(line);
        getline(fs, src, ',');
        getline(fs, dest, ',');
        getline(fs, dist);
        if (src.empty()) break;
        unsigned long srcID = std::stoul(src), destID = std::stoul(dest);
        if (vertexes.find(srcID) == vertexes.end())
            vertexes.insert(std::make_pair(srcID, new Vertex(srcID, 0, 0)));
        if (vertexes.find(destID) == vertexes.end())
            vertexes.insert(std::make_pair(destID, new Vertex(destID, 0, 0)));
        vertexes.at(srcID)->addEdge(vertexes.at(destID), std::stod(dist));
    }
    for (auto p : vertexes)
        network.addVertex(p.second);
}
```

Vertex

- Serves as a point in the graph
- Used as pointers throughout the project
- Methods consist of getters, setters and addition/removal of Edges
- The latitude and longitude are used to compute the point’s distance to another one if the corresponding edge isn’t given
- Check the Doxygen Documentation for all the attributes and methods

Edge

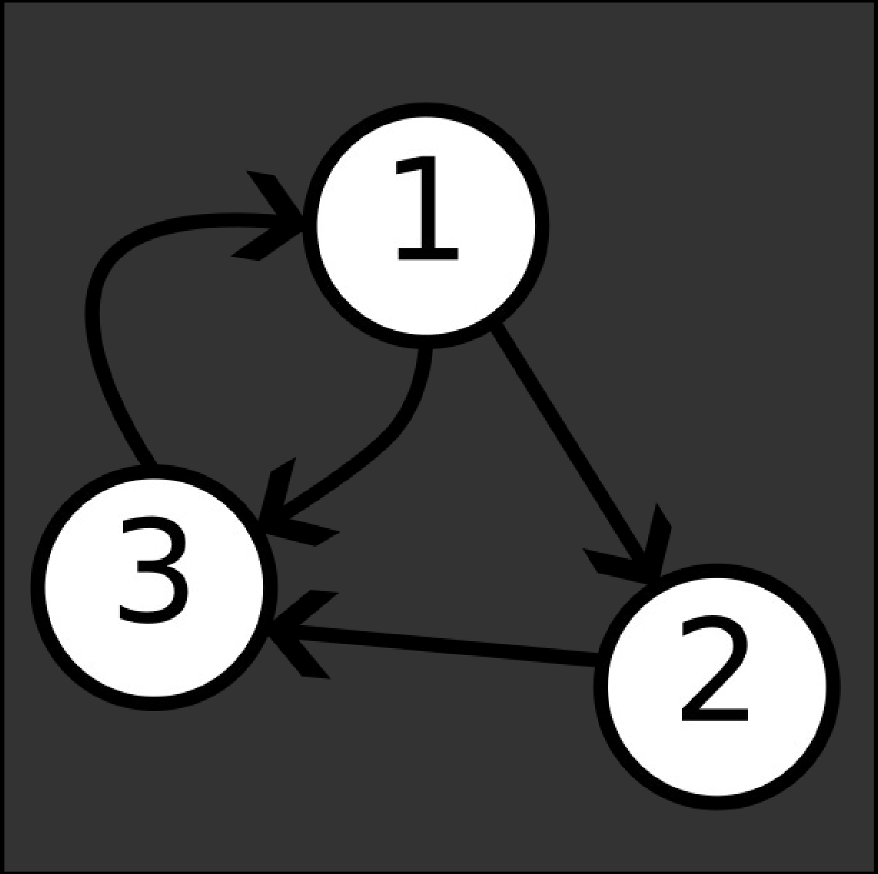
- Serves as roads / connections between two points
- The length is the distance between the two points
- Also used as pointers
- Check the Doxygen Documentation for all the attributes and methods

Graph (1/3)

- Simulated network or map
- It is composed by a vector containing all the vertexes, while edges are represented as adjancency lists in each vertex
- The Graph contains various algorithms to solve the TSP
- The unordered maps are useful to find certain vertexes
- The methods range from getters/setters to max-flow algorithms
- Check the Doxygen Documentation for all the attributes and methods

Graph (2/3)

- Unless otherwise stated, all edges are undirected
- The graph works like this (over-simplified):



Graph (3/3)

- Alongside the vertexSet, the Graph also contains an unordered map of ID to Vertex
  - This helps with vertex and edge searches
- Functions which help certain algorithms but are not intended to execute on their own are private or in the Auxil files instead

Bellman-Held-Karp

- Dynamic Programming
- Currently the most efficient algorithm to solve TSP, with time complexity $O(n^3 2^n)$
- Requires computation of subsets
- As unordered maps don’t support hashes of vectors of custom objects, a simple map was used, which lead to a complexity of $O(n^3 2^n)$

```cpp
double Graph::bellmanHeldKarp(unsigned int source) {
    std::cout << "Bellman Held Karp\n";
    Vertex* src = this->findVertex(source);
    if (!src)
        return -1;
    std::vector<Vertex*> set;
    for (Vertex* vertex : this->vertexSet) {
        if (vertex != src) set.push_back(vertex);
    }

    std::cout << "Computing Subsets\n";
    std::vector<std::vector<Vertex*>> subsets;
    computeSubsets(set, subsets);

    std::map<std::pair<std::vector<Vertex*>, Vertex*>, double> dp;

    for (Vertex* v : set) {
        Edge* e = this->findEdge(src->getId(), v->getId());
        std::vector<Vertex*> temp = {v};
        dp.insert(std::make_pair(std::make_pair(temp, v), e ? e->getLength() : DBL_MAX));
    }

    for (unsigned int s = 2; s < this->vertexSet.size(); s++) {
        std::cout << s << '\n';
        for (std::vector<Vertex*> subset : subsets) {
            if (subset.size() == s) {
                for (Vertex* v : subset) {
                    std::vector<Vertex*> temp;
                    for (Vertex* v2 : subset) {
                        if (v2 != v)
                            temp.push_back(v2);
                    }
                    double min = DBL_MAX;
                    for (Vertex* v3 : subset) {
                        if (v3 != v) {
                            Edge* e = this->findEdge(v3->getId(), v->getId());
                            min = std::min(min, dp.at(std::make_pair(temp, v3)) + (e ? e->getLength() : DBL_MAX));
                        }
                    }
                    dp[std::make_pair(subset, v)] = min;
                }
            }
        }
    }

    double optimal = DBL_MAX;
    for (Vertex* v : set) {
        Edge* e = this->findEdge(v->getId(), source);
        optimal = std::min(optimal, dp.at(std::make_pair(set, v)) + (e ? e->getLength() : DBL_MAX));
    }
    return optimal;
}
```

Bellman-Held-Karp String Variation

- Also Dynamic Programming
- This variation uses less memory and only strings
- However, it is slower due to the various find-erase it executes
- As unordered maps support hashing of strings, the time complexity is slightly smaller: $O(n^2 2^n)$

```cpp
double Graph::bellmanHeldKarp(unsigned int source) {
    std::cout << "Bellman Held Karp\n";
    Vertex* src = this->findVertex(source);
    if (!src)
        return -1;
    std::vector<Vertex*> set;
    for (Vertex* vertex : this->vertexSet) {
        if (vertex != src) set.push_back(vertex);
    }

    std::cout << "Computing Subsets\n";
    std::vector<std::vector<Vertex*>> subsets;
    computeSubsets(set, subsets);

    std::map<std::pair<std::vector<Vertex*>, Vertex*>, double> dp;

    for (Vertex* v : set) {
        Edge* e = this->findEdge(src->getId(), v->getId());
        std::vector<Vertex*> temp = {v};
        dp.insert(std::make_pair(std::make_pair(temp, v), e ? e->getLength() : DBL_MAX));
    }

    for (unsigned int s = 2; s < this->vertexSet.size(); s++) {
        std::cout << s << '\n';
        for (std::vector<Vertex*> subset : subsets) {
            if (subset.size() == s) {
                for (Vertex* v : subset) {
                    std::vector<Vertex*> temp;
                    for (Vertex* v2 : subset) {
                        if (v2 != v)
                            temp.push_back(v2);
                    }
                    double min = DBL_MAX;
                    for (Vertex* v3 : subset) {
                        if (v3 != v) {
                            Edge* e = this->findEdge(v3->getId(), v->getId());
                            min = std::min(min, dp.at(std::make_pair(temp, v3)) + (e ? e->getLength() : DBL_MAX));
                        }
                    }
                    dp[std::make_pair(subset, v)] = min;
                }
            }
        }
    }

    double optimal = DBL_MAX;
    for (Vertex* v : set) {
        Edge* e = this->findEdge(v->getId(), source);
        optimal = std::min(optimal, dp.at(std::make_pair(set, v)) + (e ? e->getLength() : DBL_MAX));
    }
    return optimal;
}
```

Triangle Inequality

- Any side of the triangle is always smaller than the sum of the other two
- This means a directed path between two points , if it exists, is always the smallest one
- This algorithm uses DFS and tree traversal
- Time Complexity: O(V * logV + E)

```cpp
double Graph::triangleInequality(unsigned int source, std::vector<Vertex*>& path) {
    std::cout << "Triangle Inequality\n";
    Vertex* src = this->findVertex(source);
    if (!src)
        return -1;
    primMST(source);
    triangleInequalityDFS(src, path);
    double min = 0;
    for (unsigned int i = 0; i < path.size(); i++) {
        for (Edge* e : path[i]->getAdj()) {
            if (e->getDest() == path[(i+1) % path.size()]) {
                min += e->getLength();
                break;
            }
        }
    }
    path.push_back(src);
    return min;
}
```

Nearest Neighbour

- For each outgoing edge, it selects the smallest one and proceeds forward
- It uses this approach for all vertexes, starting on all vertexes
- It yields a fairly good solution, although there are special cases where it can yield the worst solution
- Complexity: O(V * (V + E))

```cpp
double Graph::nearestNeighbour(unsigned int source, std::vector<Vertex *> &path) {
    std::cout << "Nearest Neighbour\n";

    double min = DBL_MAX, sum = 0;
    unsigned int count = this->vertexSet.size() - 1;

    for (Vertex *v: this->vertexSet) {
        v->setVisited(false);
        v->setPath(nullptr);
    }

    Vertex *src = this->findVertex(0);
    Vertex *current = src;
    path.push_back(src);
    while (count > 0) {
        Edge *shortest = nullptr;
        unsigned int l = UINT_MAX;
        for (Edge *e: current->getAdj()) {
            if (!e->getDest()->isVisited() && e->getLength() < l) {
                l = e->getLength();
                shortest = e;
            }
        }
        if (!shortest) {
            std::cout << "The algorithm cannot complete.\n";
            path.clear();
            return -1;
        }
        current->setVisited(true);
        current = shortest->getDest();
        current->setPath(shortest);
        sum += shortest->getLength();
        path.push_back(current);
        count--;
    }
```

```cpp
    if (count == 0) {
        Edge *end = nullptr;
        for (Edge *e: current->getAdj()) {
            if (e->getDest() == src) {
                end = e;
                break;
            }
        }
        if (!end) {
            std::cout << "The algorithm cannot complete.\n";
            path.clear();
            return -1;
        } else {
            sum += end->getLength();
            if (sum < min) {
                min = sum;
                src->setPath(end);
                path.push_back(src);
            }
        }
    }

    return min;
}
```

Real World Graphs TSP Algorithm

- This is a tweaked version of the 2-approximation algorithm to work on real world graphs
- First, we compute the MST of the graph and, then, we try to compute a tweaked version of the pre-order walk of the, as you can see on the next slide
- This tweak consists of, when we encounter a vertex that cannot go further in the pre-order walk, we leave it behind (not including in the path) in the hope of being able to return there in the future
- Complexity: O(V * logV + E)

```
double Graph::realWorldTSP(unsigned int source, std::vector<Vertex *> &path) {
    std::cout << "Real World Algorithm\n";

    Vertex *src = this->findVertex(source);
    if (!src) return -1;
    primMST(source);
    for (Vertex *v: this->getVertexSet()) {
        v->setVisited(false);
        v->setLeftBehind(false);
    }

    double cost = 0;
    unsigned int leftBehindCounter = 0;
    int ret = findPath(src, path, cost, leftBehindCounter);
    if (ret < 0) return ret;
    return cost;
}
```

Find Path in MST

- This is a recursive function that tries to find which vertex to go next
- When a said vertex A is left behind, and, in the future, we manage to go from a said vertex B to A and back to a child (in the MST representation) of B, we do it
- Complexity: O(V + E)

```cpp
int Graph::findPath(Vertex *v, std::vector<Vertex *> &path, double &cost, unsigned int &leftBehindCounter) {
    path.push_back(v);
    v->setVisited(true);
    if (path.size() == this->getVertexSet().size()) {
        for (Edge *e: v->getAdj()) {
            if (e->getDest() == path[0]) { // If we can go to source, we succeeded!
                path.push_back(e->getDest());
                cost += e->getLength();
                return 0;
            }
        }
        return -1; // If we cannot go to source, we failed!
    }
    if (path.size() + leftBehindCounter == this->getVertexSet().size()) {
        return -1; // In this case, we conclude that there is no path.
    }
    // Try to go to a left behind vertex:
    for (Edge *e1: v->getAdj()) {
        if (leftBehindCounter == 0) break;
        if (e1->getDest()->isLeftBehind()) { // For each left behind adjacent vertex:
            for (Edge *e2: e1->getDest()->getAdj()) {
                if (e2->getDest()->getPath() == nullptr) continue;
                if (e2->getDest()->getPath()->getOrig() == v) {
                    // If the left behind vertex can go back to a vertex that is a
                    // child (in the MST representation) of the vertex v
                    path.push_back(e1->getDest());
                    e1->getDest()->setLeftBehind(false);
                    leftBehindCounter--;
                    cost += e1->getLength() + e2->getLength();

                    int ret = findPath(e2->getDest(), path, cost, leftBehindCounter);
                    if (ret == -2) { // If the vertex was left behind
                        path.pop_back();
                        e1->getDest()->setLeftBehind(true);
                        leftBehindCounter++;
                        cost -= e1->getLength() + e2->getLength();
                        continue;
                    } else {
                        return ret;
                    }
                }
            }
        }
    }
```

```cpp
    // Try to go to the next vertex in the MST:
    for (Edge *e: v->getAdj()) {
        if (e->getDest()->getPath() == nullptr) continue;
        if ((!e->getDest()->isVisited()) && (e->getDest()->getPath()->getOrig() == v)) {
            // For each child (in the MST representation) of v
            cost += e->getLength();
            int ret = findPath(e->getDest(), path, cost, leftBehindCounter);
            if (ret == -2) { // If the vertex was left behind
                cost -= e->getLength();
                continue;
            } else return ret;
        }
    }
    // If it isn't possible to go down the MST, try to go up to find a vertex to go next:
    if (v->getPath() == nullptr) return -1;
    Vertex *intermediate = v->getPath()->getOrig();
    while (true) {
        for (Edge *e1: intermediate->getAdj()) {
            if (e1->getDest()->getPath() == nullptr) continue;
            if ((!e1->getDest()->isVisited()) && (e1->getDest()->getPath()->getOrig() == intermediate)) {
                for (Edge *e2: v->getAdj()) {
                    if (e1->getDest() == e2->getDest()) {
                        cost += e2->getLength();
                        int ret = findPath(e2->getDest(), path, cost, leftBehindCounter);
                        if (ret == -2) { // If the vertex was left behind
                            cost -= e2->getLength();
                            continue;
                        } else return ret;
                    }
                }
            }
        }
        if (intermediate == path[0] || intermediate->getPath() == nullptr)
            break; // If the intermediate is the source, we cannot go up in the MST.
        intermediate = intermediate->getPath()->getOrig();
    }
    v->setLeftBehind(true);
    leftBehindCounter++;
    path.pop_back();
    return -2;
}
```

User Interface (1/2)

- We developed an interface to allow the user to easily interact with the Network.
- A menu with the valid options appears and the user can type either the option number or the option name followed by its arguments, if required.
- First, you need to load the desired graph to the program, using the option 0 (or load) followed by the path to the edges and (optionally) vertexes files.
- You can, then, execute the provided TSP algorithms.

User Interface (2/2)

```
Welcome to our project!
Please, read the README.md file to know how to properly use the program.

List of available algorithms:
 0 - load <edges_csv> [<nodes_csv>] : Loads the graph using the given filenames
 1 - backtrack                      : Solves the TSP using the Backtracking algorithm
 2 - tineq                          : Solves the TSP using the 2-approximation Triangle Inequality algorithm
 3 - near                           : Solves the TSP using the Nearest Neighbour algorithm
 4 - bhk                            : Solves the TSP using the Bellman-Held-Karp algorithm
 5 - bhks                           : Solves the TSP using the Bellman-Held-Karp String Version algorithm (slower but uses less memory)
 6 - rwtsp [<src_id>]               : Solves the TSP using the Real World adaptation of the 2-approximation Triangular Inequality algorithm
 7 - exit | quit                    : Quits the program
Notes:
    - src_id is an optional argument to choose the node to start at
    - You can either type the algorithm name (as shown above) or its correspondent number

> load csv/Real-World/graph3/edges.csv csv/Real-World/graph3/nodes.csv
```

Bibliography

- Slide 3: https://upload.wikimedia.org/wikipedia/commons/1/15/3_node_Directed_graph.png?20230623211051

- Font used (by Nerd Fonts, clicking here will download the font): https://github.com/ryanoasis/nerd-fonts/releases/download/v3.1.1/Hack.zip

- All other images and descriptions were made by us