ANALYSIS AND SYNTHESIS OF ALGORITHMS DESIGN OF ALGORITHMS (DA) PROGRAMMING PROJECT I

An Analysis Tool for Water Supply Management

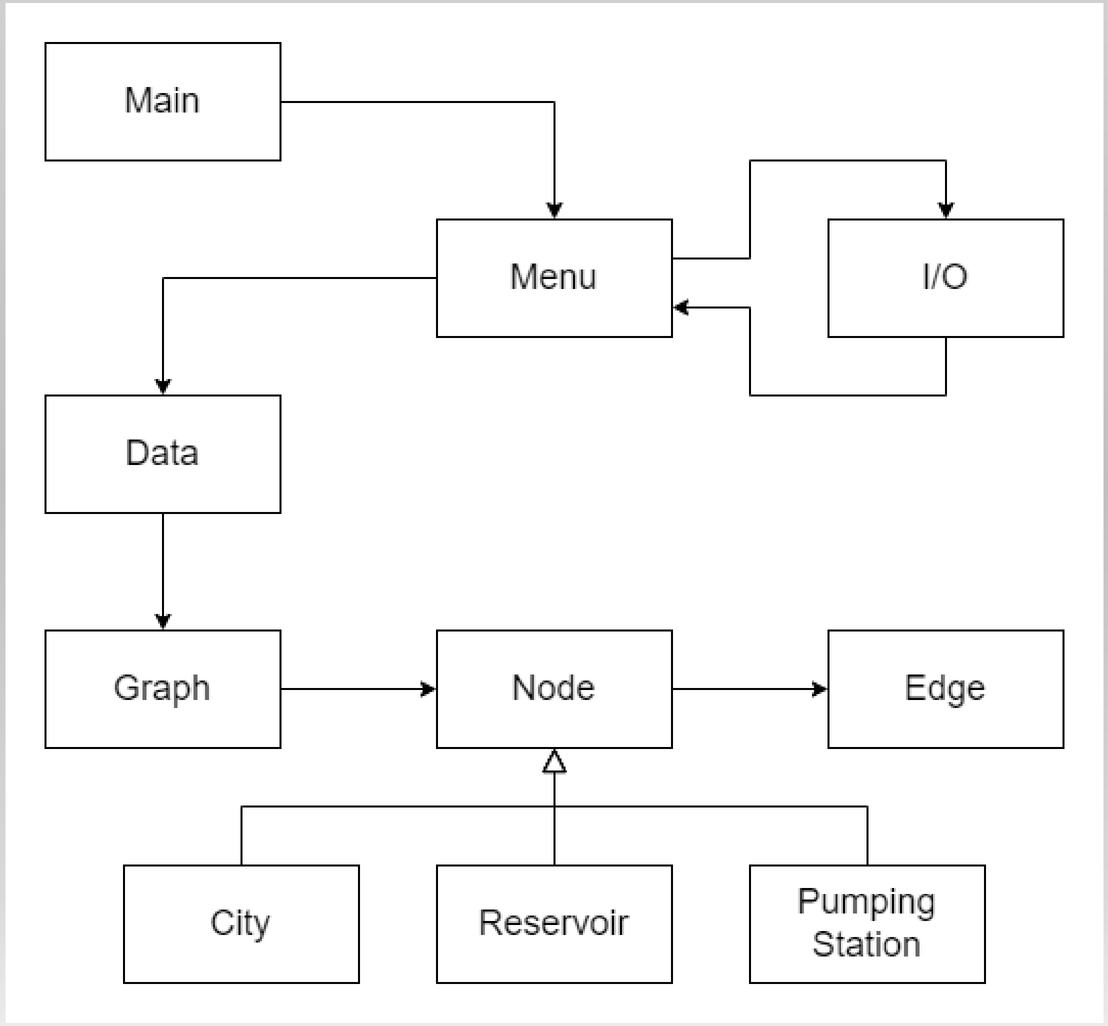
Group Members:

Gabriela Dias Salazar Neto Silva – up202004443

Diogo Alexandre da Silva Santos – up202009291

Mário Costa Rêgo da Silva - up202108841

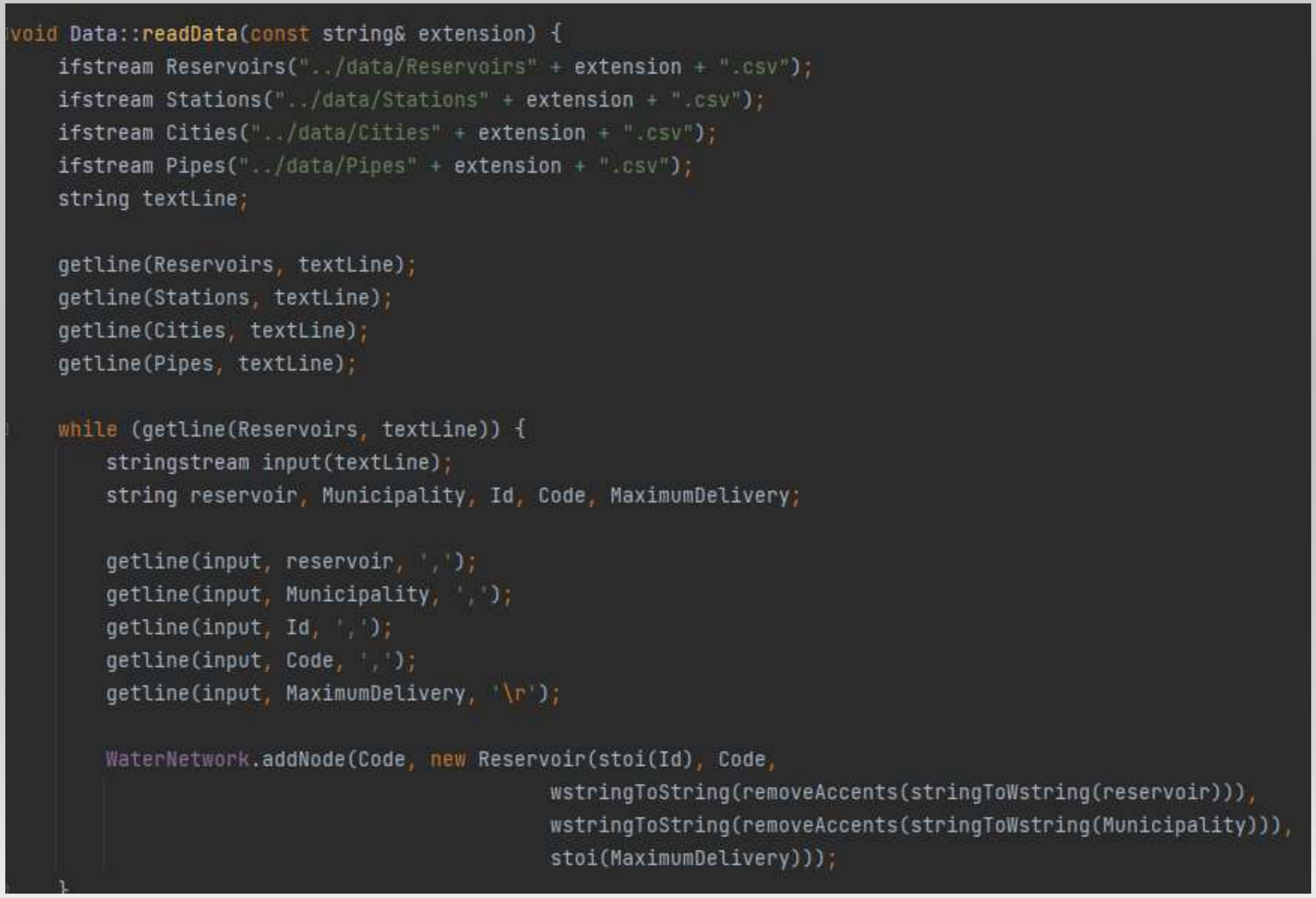
CLASS DIAGRAM



- Menu class manages the input and output of data related to the water supply network.
- Data class interacts with the project’s files and is the bridge between the Menu and the Graph classes
- Graph class represents a directed graph data structure used to model the water supply network.
- Node class serves as a base class for representing nodes in the graph.
- The City class inherits from Node and represents city nodes in the graph.
- The Reservoir class also inherits from Node and represents reservoir nodes in the graph.
- The PumpingStation class, again inheriting from Node, represents pumping station nodes in the graph.
- The Edge class represents the pipelines, designed as the edges of the graph.

DESCRIPTION OF READING THE GIVEN DATASET

- The ‘WaterNetwork’ graph is initialized.
- The Data class reads the provided dataset, parses its contents, and utilizes the Graph class to construct a graph representation reflecting the dataset's information.

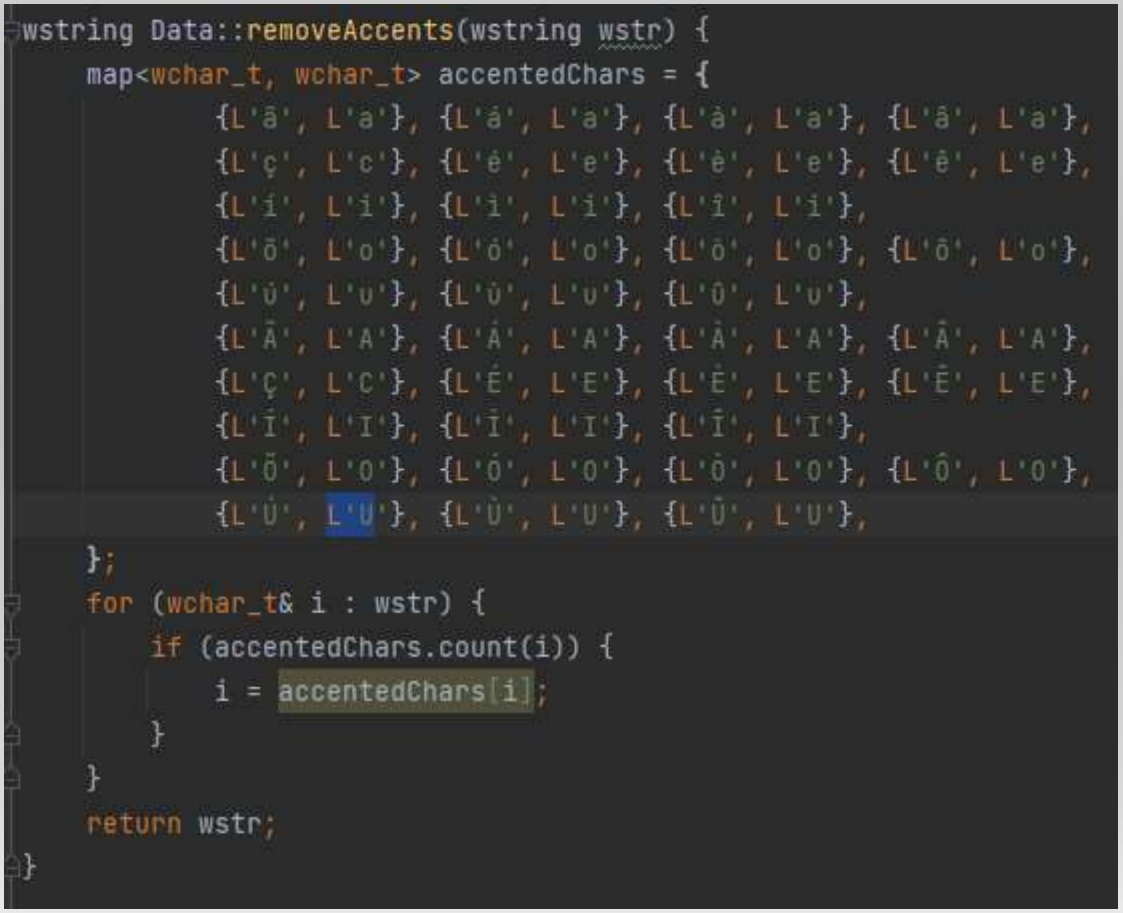


```
void Data::readData(const string& extension) {
    ifstream Reservoirs("../data/Reservoirs" + extension + ".csv");
    ifstream Stations("../data/Stations" + extension + ".csv");
    ifstream Cities("../data/Cities" + extension + ".csv");
    ifstream Pipes("../data/Pipes" + extension + ".csv");
    string textLine;

    getline(Reservoirs, textLine);
    getline(Stations, textLine);
    getline(Cities, textLine);
    getline(Pipes, textLine);

    while (getline(Reservoirs, textLine)) {
        stringstream input(textLine);
        string reservoir, Municipality, Id, Code, MaximumDelivery;

        getline(input, reservoir, ',');
        getline(input, Municipality, ',');
        getline(input, Id, ',');
        getline(input, Code, ',');
        getline(input, MaximumDelivery, '\r');

        WaterNetwork.addNode(Code, new Reservoir(stoi(Id), Code,
                                     wstringToString(removeAccents(stringToWstring(reservoir))),
                                     wstringToString(removeAccents(stringToWstring(Municipality))),
                                     stoi(MaximumDelivery)));
    }
```

Illustrative exemple of the dataset being read and reservoir nodes being created

DESCRIPTION OF READING THE GIVEN DATASET

- String conversion methods handle Unicode characters, while accents are removed from city names using a predefined mapping.
- The MaxFlow method employs the Edmonds-Karp algorithm to determine the maximum flow within the water network. It then generates two output files: "FlowGraph.csv" contains flow details between source and destination nodes, while "MaxFlow.csv" provides information on the maximum flow for each city in the network.



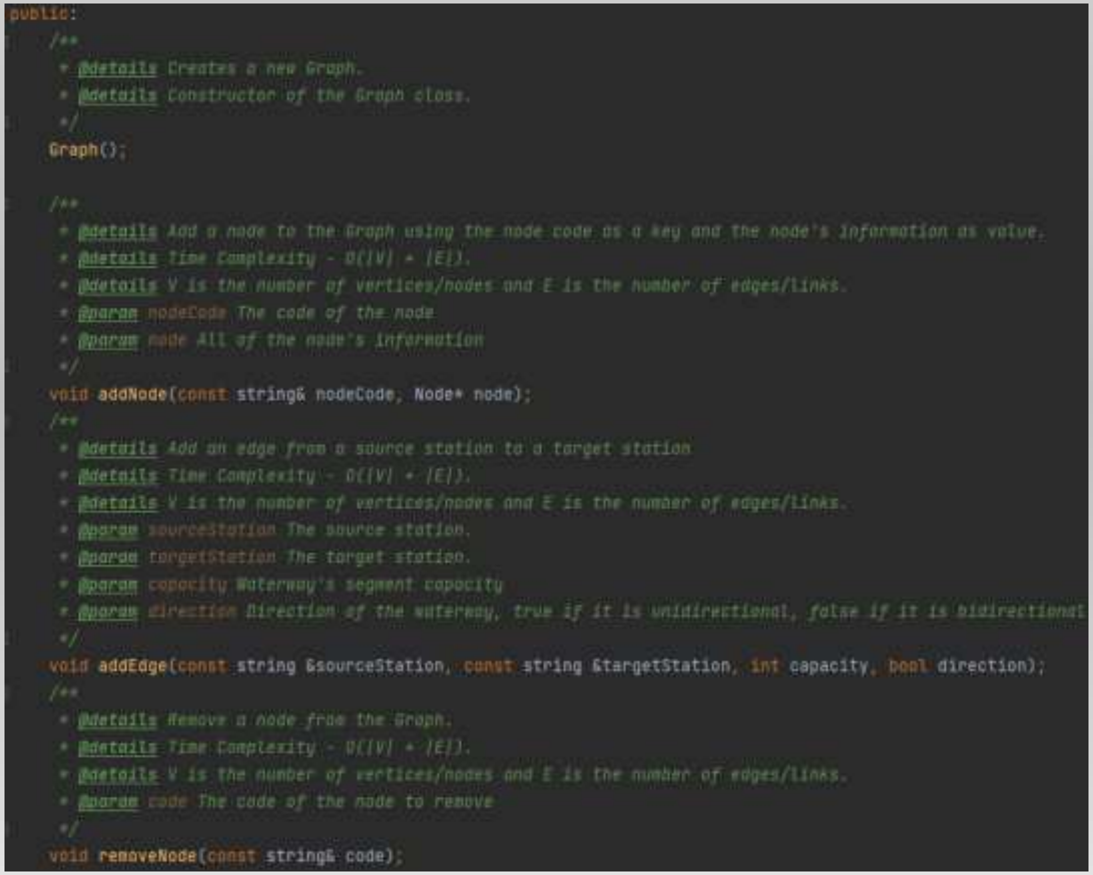
```
wstring Data::removeAccents(wstring wstr) {
    map<wchar_t, wchar_t> accentedChars = {
            {L'ã', L'a'}, {L'á', L'a'}, {L'à', L'a'}, {L'â', L'a'},
            {L'ç', L'c'}, {L'é', L'e'}, {L'è', L'e'}, {L'ê', L'e'},
            {L'í', L'i'}, {L'ì', L'i'}, {L'î', L'i'},
            {L'õ', L'o'}, {L'ó', L'o'}, {L'ò', L'o'}, {L'ô', L'o'},
            {L'ú', L'u'}, {L'ù', L'u'}, {L'û', L'u'},
            {L'Ã', L'A'}, {L'Á', L'A'}, {L'À', L'A'}, {L'Â', L'A'},
            {L'Ç', L'C'}, {L'É', L'E'}, {L'È', L'E'}, {L'Ê', L'E'},
            {L'Í', L'I'}, {L'Ì', L'I'}, {L'Î', L'I'},
            {L'Õ', L'O'}, {L'Ó', L'O'}, {L'Ò', L'O'}, {L'Ô', L'O'},
            {L'Ú', L'U'}, {L'Ù', L'U'}, {L'Û', L'U'},
    };
    for (wchar_t& i : wstr) {
        if (accentedChars.count(i)) {
            i = accentedChars[i];
        }
    }
    return wstr;
}
```

Illustrative exemple of the implementation of the removeAccents method

DESCRIPTION OF THE GRAPH USED TO REPRESENT THE DATASETS

The dataset is represented using a directed graph, where each node represents a component of the water distribution system and each edge represents a connection or pipeline between two components. Here is a brief description of the graph:

- The graph consists of three types of nodes: Reservoirs, Pumping Stations, and Cities, each represented by their respective classes.
- Nodes store essential information such as ID, code, and specific properties like maximum delivery for reservoirs, demand for cities, and other relevant attributes.
- Edges in the graph represent the connections between different nodes, indicating the flow of water from one point to another and contain information such as the source node, destination node, capacity (maximum flow), and current flow.



```
public:
    /**
     * @details Creates a new Graph.
     * @details Constructor of the Graph class.
     */
    Graph();

    /**
     * @details Add a node to the Graph using the node code as a key and the node's information as value.
     * @details Time Complexity - O(|V| + |E|).
     * @details V is the number of vertices/nodes and E is the number of edges/links.
     * @param nodeCode The code of the node
     * @param node All of the node's information
     */
    void addNode(const string& nodeCode, Node* node);
    /**
     * @details Add an edge from a source station to a target station
     * @details Time Complexity - O(|V| + |E|).
     * @details V is the number of vertices/nodes and E is the number of edges/links.
     * @param sourceStation The source station.
     * @param targetStation The target station.
     * @param capacity Waterway's segment capacity
     * @param direction Direction of the waterway, true if it is unidirectional, false if it is bidirectional
     */
    void addEdge(const string &sourceStation, const string &targetStation, int capacity, bool direction);
    /**
     * @details Remove a node from the Graph.
     * @details Time Complexity - O(|V| + |E|).
     * @details V is the number of vertices/nodes and E is the number of edges/links.
     * @param code The code of the node to remove
     */
    void removeNode(const string& code);
```

IMPLEMENTED FUNCTIONALITIES AND ASSOCIATED ALGORITHMS

Display All Nodes:
- Retrieve and display information about all nodes in the water network graph.
- Provides an overview of all reservoirs, pumping stations, and cities present in the network.
- Time Complexity: O(N + M), where N is the total number of nodes in the graph and M is the total number of edges.

Display Specific Node:
- Retrieve and display detailed information about a specific node in the water network graph.
- Users can input the node code or identifier to retrieve specific details.
- Time Complexity: O(N), where N is the total number of nodes in the graph.

Max Flow:
- Calculate and display the maximum flow reaching each or a specific city in the water network.
- Time Complexity: O(N + M), where N is the total number of nodes in the graph and M is the total number of edges. This complexity arises from the MaxFlowMenu function, which involves maximum flow using algorithms such as Edmonds-Karp.

IMPLEMENTED FUNCTIONALITIES AND ASSOCIATED ALGORITHMS

Water Deficit Calculation (WaterDeficit):

- This method calculates the deficit of water supply in cities by comparing the demand of each city with the incoming flow from reservoirs.
- Time Complexity: O(n log n). Iterating through all nodes in the graph: $O(n)$; Calculating the maximum flow and comparing with demand for each city: $O(n\log n)$ due to sorting; Total: $O(n)+O(n\log n)=O(n\log n)$.

Water Redistribution (balanceLoad):

- This method redistributes water flow within the network to address deficits and balance supply and demand. This functionality also includes the comparison between some metrics, before and after the water redistribution.
- Time Complexity: O(n log n); ComputeMetrics: $O(n\log n)$ due to sorting; Calculating total excess capacity and unmet demand: $O(n)$; Redistributing flow based on excess capacity and unmet demand: $O(n)$; Total: $O(n\log n)+O(n)+O(n)=O(n\log n)$.

IMPLEMENTED FUNCTIONALITIES AND ASSOCIATED ALGORITHMS

Reservoir Removal Resilience Check:

- Simulates reservoir removal by unique code.
- Recalculates flow via Edmonds-Karp algorithm.
- Assesses impact on city demands, revealing potential water shortage risks.
- Time Complexity: O(n·m) where *n* is the number of cities in the network and *m* is the average number of edges per city.

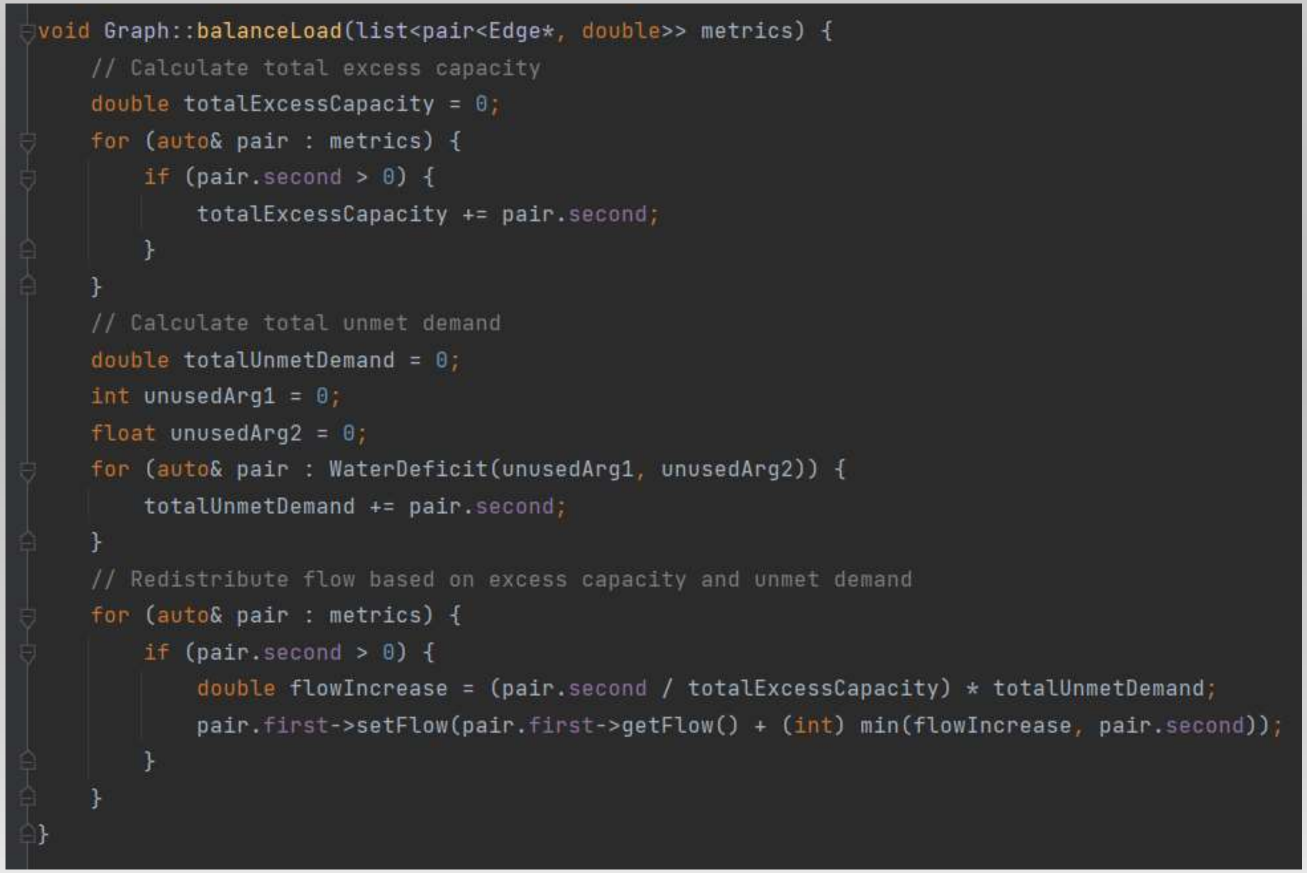
Pumping Station Removal Resilience Check:

- Evaluates impact of removing pumping station by code.
- Temporarily removes station's incoming edges.
- Recalculates flow, ranks cities by deficit, highlighting critical areas.
- Time Complexity: *O*(*n*·*m*); *n* is the number of cities in the network; *m* is the average number of edges per city.

Individual Edge Removal Resilience Check:

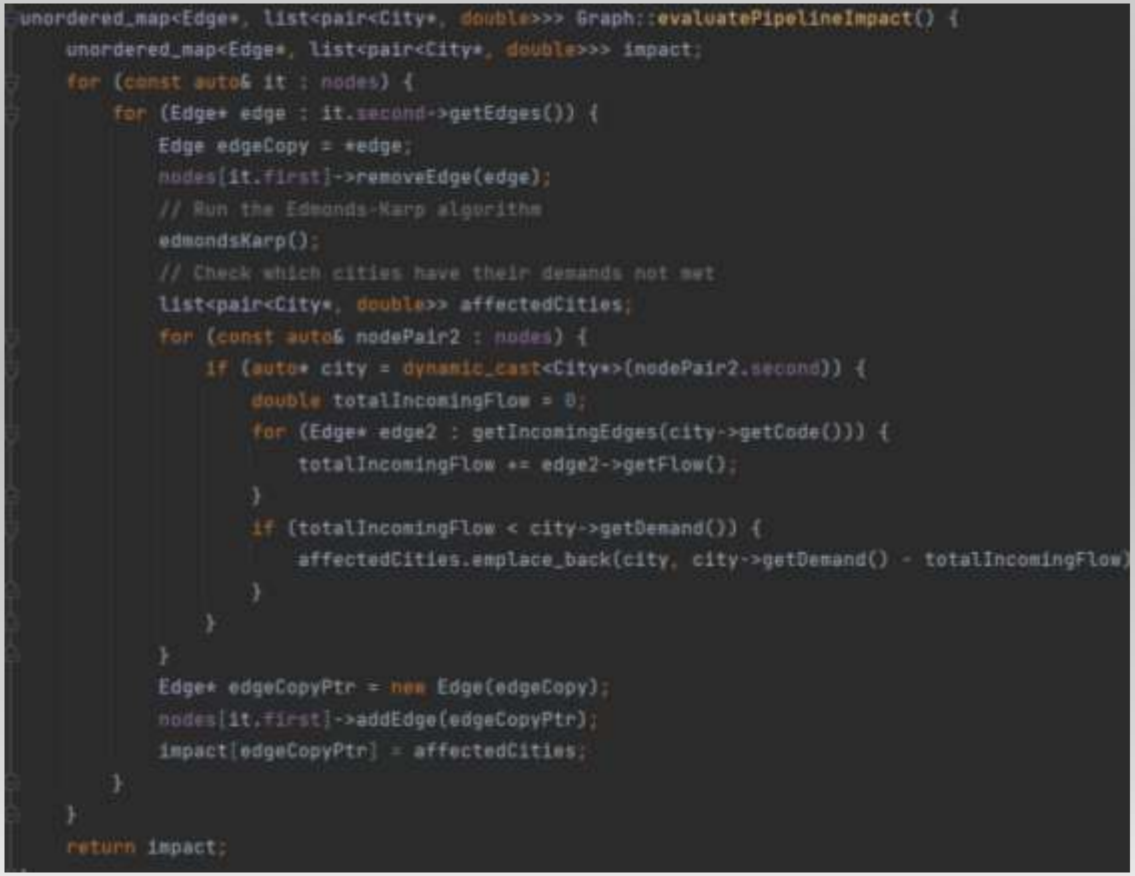
- Systematically removes each edge.
- Recalculates flow after removal.
- Identifies cities with unmet demand, gauging network vulnerability.
- Time Complexity: O(n^2·m); n is the number of cities in the network; m is the average number of edges per city.

IMPLEMENTED FUNCTIONALITIES AND ASSOCIATED ALGORITHMS



```cpp
void Graph::balanceLoad(list<pair<Edge*, double>> metrics) {
    // Calculate total excess capacity
    double totalExcessCapacity = 0;
    for (auto& pair : metrics) {
        if (pair.second > 0) {
            totalExcessCapacity += pair.second;
        }
    }
    // Calculate total unmet demand
    double totalUnmetDemand = 0;
    int unusedArg1 = 0;
    float unusedArg2 = 0;
    for (auto& pair : WaterDeficit(unusedArg1, unusedArg2)) {
        totalUnmetDemand += pair.second;
    }
    // Redistribute flow based on excess capacity and unmet demand
    for (auto& pair : metrics) {
        if (pair.second > 0) {
            double flowIncrease = (pair.second / totalExcessCapacity) * totalUnmetDemand;
            pair.first->setFlow(pair.first->getFlow() + (int) min(flowIncrease, pair.second));
        }
    }
}
```

Illustrative exemple of the implementation of the balanceLoad method



```cpp
unordered_map<Edge*, list<pair<City*, double>>> Graph::evaluatePipelineImpact() {
    unordered_map<Edge*, list<pair<City*, double>>> impact;
    for (const auto& it : nodes) {
        for (Edge* edge : it.second->getEdges()) {
            Edge edgeCopy = *edge;
            nodes[it.first]->removeEdge(edge);
            // Run the Edmonds-Karp algorithm
            edmondsKarp();
            // Check which cities have their demands not met
            list<pair<City*, double>> affectedCities;
            for (const auto& nodePair2 : nodes) {
                if (auto* city = dynamic_cast<City*>(nodePair2.second)) {
                    double totalIncomingFlow = 0;
                    for (Edge* edge2 : getIncomingEdges(city->getCode())) {
                        totalIncomingFlow += edge2->getFlow();
                    }
                    if (totalIncomingFlow < city->getDemand()) {
                        affectedCities.emplace_back(city, city->getDemand() - totalIncomingFlow)
                    }
                }
            }
            Edge* edgeCopyPtr = new Edge(edgeCopy);
            nodes[it.first]->addEdge(edgeCopyPtr);
            impact[edgeCopyPtr] = affectedCities;
        }
    }
    return impact;
```

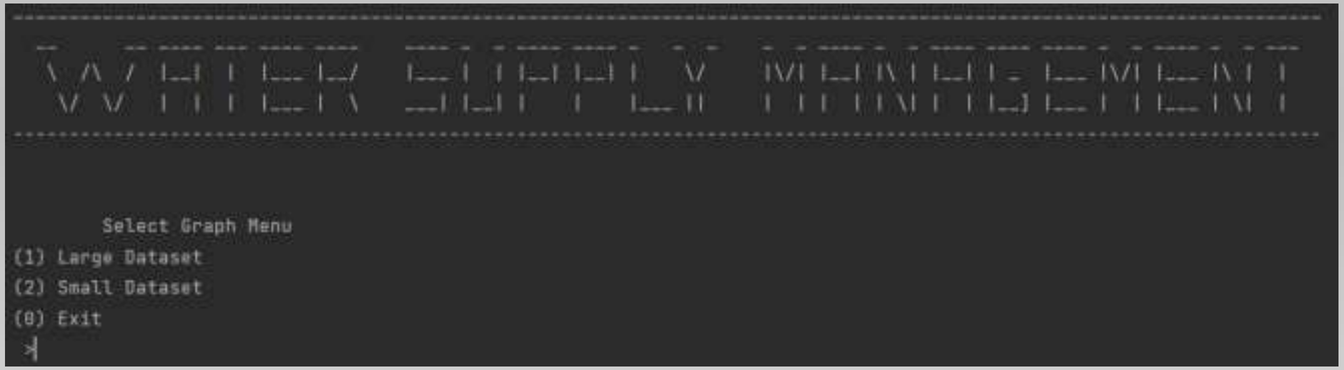
Illustrative exemple of the implementation of the evaluatePipelineImapct method

DESCRIPTION OF THE USER INTERFACE

The user interface includes functions to retrieve user input, ensuring that the input is valid and within the specified options. These functions are ‘getUserInput’ for numeric input and ‘getUserTextInput’ for string input.

Select Graph Menu

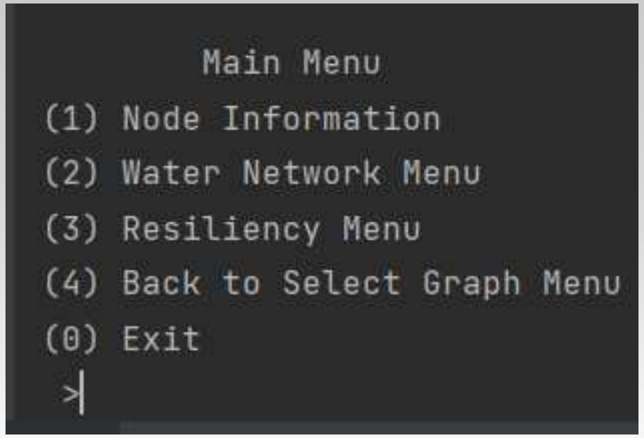
The user is prompted to select between a large dataset and a small dataset.



```
      Select Graph Menu
(1) Large Dataset
(2) Small Dataset
(0) Exit
 >
```

Main Menu

After selecting a dataset, the user is presented with the main menu options.



```
      Main Menu
(1) Node Information
(2) Water Network Menu
(3) Resiliency Menu
(4) Back to Select Graph Menu
(0) Exit
 >
```

FUNCTIONALITIES TO HIGHLIGHT

User Interface Design:

Our code implements a well-structured user interface with clear menus and options for the user to navigate through different functionalities of the application. The use of ASCII art for title display adds a visually appealing touch to the UI.

Resiliency Checks:

The evaluateReservoirImpact, evaluatePumpingStationImpact, and evaluatePipelineImpact functions perform resilience checks on the water network by simulating the removal of reservoirs, pumping stations, and individual edges. These functionalities assess the impact on city demands and identify potential water shortage risks.

MAIN DIFFICULTIES AND PARTICIPATION OF EACH GROUP MEMBER

Maintainability and Extensibility:

As the project grows, maintaining and extending the codebase can become challenging. Ensuring that the code is well-organized, modular, and follows best practices can facilitate future development and maintenance efforts. Additionally, accommodating new features or requirements while maintaining backward compatibility requires careful planning and design.

Participation of each group member

Gabriela Dias Salazar Neto Silva – 40%
Diogo Alexandre da Silva Santos – 60%
Mário Costa Rêgo da Silva – 0%