An Analysis Tool for Water Supply Management

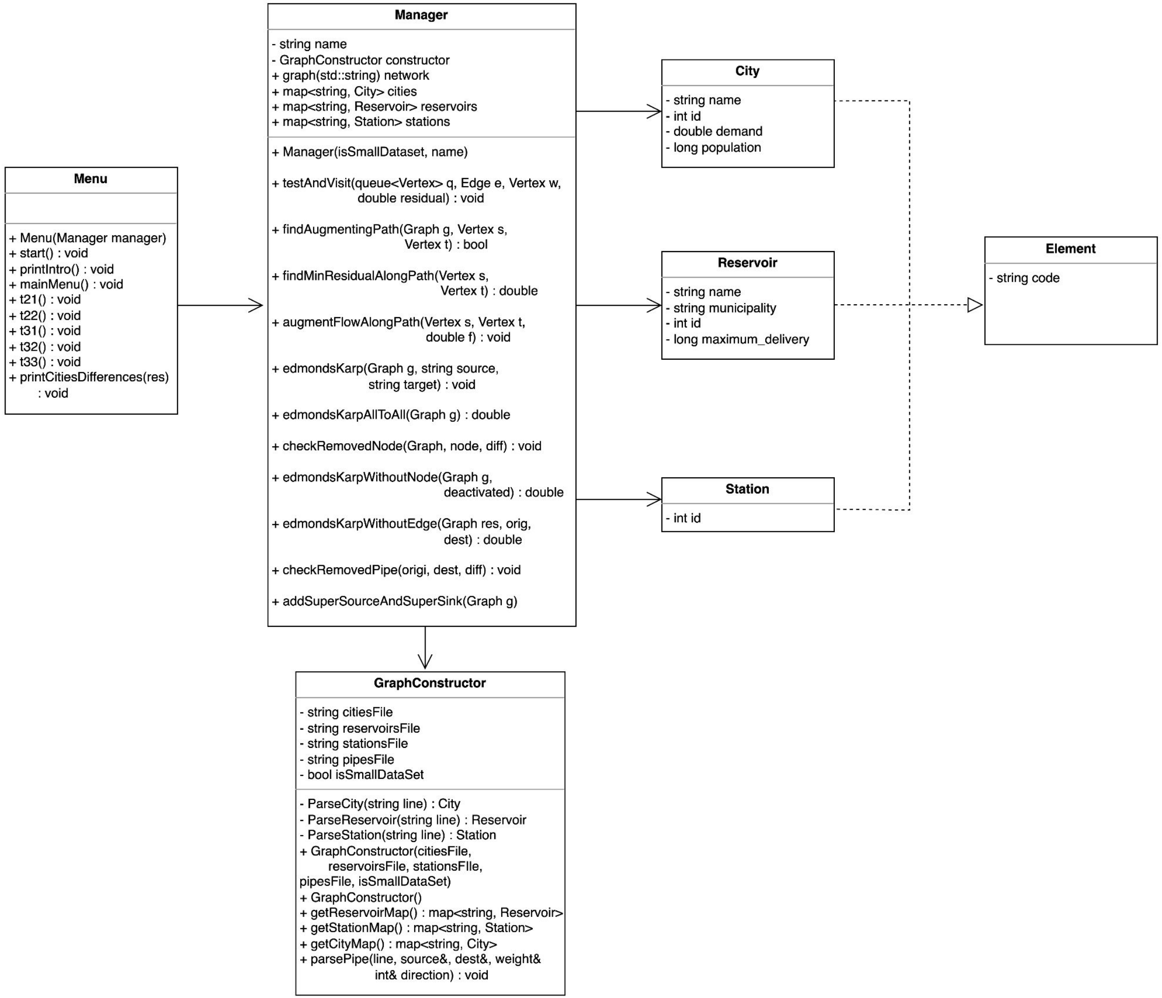
Design of Algorithms Programming Project 1

FEUP AquaFlow

Class: 2LEIC05 Group: 2

Ana Ramos (up201904969) Davide Teixeira (up202109860)

Class Diagram



Datasets Reading and Parsing

- **Utilize C++ streams:** Use input file streams (*ifstream*) to read data from files.
- **Line-by-line parsing:** Read each line from the file and parse it individually.
- **Tokenization:** Split each line into tokens (fields) using *std::istringstream* and *getline()* function.
- **Data conversion:** Convert tokens to appropriate data types (e.g., strings to integers or floats) for creating objects.
- **Conditional handling:** Implement conditional checks for special cases or variations in data formats.

```
std::ifstream inCities(citiesFile);
getline(inCities, line); // for jumping header
while(getline(inCities, line)){
    City city = parseCity(line);
    res.addVertex(city.getCode());
}
```

Datasets Reading and Parsing

- **Extract city data** from CSV lines and create *City*, *Reservoir* and *Station* objects.
- **Extract pipe data** from CSV lines and extract *source*, *destination*, *weight*, and *direction*.
- **Map creation:** Utilize <u>unordered maps</u> to store parsed objects (e.g., cities, reservoirs, stations) for easy access.

```cpp
City GraphConstructor::parseCity(std::string line) {
    std::vector<std::string> lineParsed;
    std::string word;
    std::istringstream iss(line);
    while(getline(iss, word, ',')){
        lineParsed.push_back(word);
    }

    std::string cityName = lineParsed[0];
    int cityId = std::stoi(lineParsed[1]);
    std::string cityCode = lineParsed[2];
    float cityDemand = std::stof(lineParsed[3]);
    long cityPopulation;
    if (isSmallDataSet) {
        std::string cityPopulationComplete = lineParsed[4] + lineParsed[5];
        cityPopulationComplete = cityPopulationComplete.substr(1, cityPopulationComplete.size()-2);
        if (cityPopulationComplete[cityPopulationComplete.size()-1] == '\"')
            cityPopulationComplete = cityPopulationComplete.substr(0, cityPopulationComplete.size()-1);
        cityPopulation = std::stol(cityPopulationComplete);
    }
    else{
        cityPopulation = std::stol(lineParsed[4]);
    }

    return City(cityName, cityId, cityCode, cityDemand, cityPopulation);
}
```

Graphs

- It was used a single graph to represent the whole system. Each node of the graph has a string representing the code of the city, reservoir or pumping station.
- To get the data from each node, 3 *unordered_maps* are used to get the *City*, *Reservoir* and *Pumping Station* objects.
- The graph is then saved on the *Manager* class.

```
class Manager {
private:
    std::string name; /**< Name of the manager. */
    GraphConstructor constructor; /**< Graph constructor instance. Responsible for constructing the graph */
public:
    Graph<std::string> network; /**< Network graph. */
    std::unordered_map<std::string, City> cities; /**< Map of cities. Maps the code to the city */
    std::unordered_map<std::string, Reservoir> reservoirs; /**< Map of reservoirs. Maps the code to the reservoir*/
    std::unordered_map<std::string, Station> stations; /**< Map of stations. Maps the code to the station*/
```

Implemented Functionalities

- **Generic Edmonds-Karp:** O(V * E^2)
  - Algorithm that implements the Edmonds-Karp algorithm from an arbitrary source to an arbitrary sink.

```cpp
void Manager::edmondsKarp(Graph<std::string> *g, std::string source, std::string target) {
    // Find source and target vertices in the graph
    Vertex<std::string>* s = g->findVertex(source); Vertex<std::string>* t = g->findVertex(target);
    // Validate source and target vertices
    if (s == nullptr || t == nullptr || s == t)
        throw std::logic_error("Invalid source and/or target vertex");
    // Initialize flow on all edges to 0
    for (auto v : g->getVertexSet()) {
        for (auto e : v->getAdj()) {
            if (!e->isActivated()){
                std::cout << "Edge not activated" << std::endl;
            }
            e->setFlow(0);
        }
    }
    // While there is an augmenting path, augment the flow along the path
    while( findAugmentingPath(g, s, t) ) {
        double f = findMinResidualAlongPath(s, t);
        augmentFlowAlongPath(s, t, f);
    }
}
```

Implemented Functionalities

- **Breadth First Search:** O(E+V)
  - As it is needed for the Edmonds-Karp Algorithm, a general BFS was implemented.
  - This algorithm is adapted to operate on active vertices and edges.

```
bool Manager::findAugmentingPath(Graph<std::string> *g, Vertex<std::string> *s, Vertex<std::string> *t) {
    // Mark all vertices as not visited
    for(auto v : Vertex<string> * : g->getVertexSet()) { v->setVisited(false);
    }
    // Mark the source vertex as visited and enqueue it
    s->setVisited(true); std::queue<Vertex<std::string> *> q; q.push( v: s);
    // BFS to find an augmenting path
    while( ! q.empty() && ! t->isVisited()) {
        auto v : Vertex<string> * = q.front();
        q.pop();
        if (!v->isActiveted()) continue;
        // Process outgoing edges
        for(auto e : Edge<string> * : v->getAdj()) {
            if (!e->isActivated()) {
                std::cout << "Deactivated edge, ignore!" << std::endl;
                continue;
            }

            testAndVisit( &: q, e,  w: e->getDest(),  residual: e->getWeight() - e->getFlow());
        }
        // Process incoming edges
        for(auto e : Edge<string> * : v->getIncoming()) {
            if (!e->isActivated()){
                continue;
            }
            testAndVisit( &: q, e,  w: e->getOrig(),  residual: e->getFlow());
        } }
    // Return true if a path to the target is found, false otherwise
    return t->isVisited();
}
```

Implemented Functionalities

- **Check if the system can supply all cities:** O(V*E^2)
    - It was used an algorithm that creates a supersource that connects to all reservoirs, and a supersink in which all cities connect to.
    - Then, it uses the Edmonds-Karp between the supersource and supersink
    - Lists the cities' deficit
    - Saves everything to a file

```cpp
double Manager::edmondsKarpAllToAll(Graph<std::string>* res){

    Graph<std::string> graph = constructor.createGraph();

    addSupersourceAndSuperSink( &: graph);

    edmondsKarp( g: &graph,  source: "supersource",  target: "supersink");

    *res = graph;

    int maxflow = 0;
    for (int i = 0; i < graph.getNumVertex(); i++){
        if (graph.getVertexSet()[i]->getInfo()[0] == 'C') {
            for (int j = 0; j < graph.getVertexSet()[i]->getIncoming().size(); j++){
                maxflow += graph.getVertexSet()[i]->getIncoming()[j]->getFlow();
            }
        }
    }

    return maxflow;
}
```

Implemented Functionalities

- **Out of Commission Testing:** O(V*E^2)
  - Tests what happens if a *Reservoir*, *Pumping Station* or *pipe* are out of commission.
  - Saves all data into a file, with the before and after the removal.

```
double Manager::edmondsKarpWithoutNode(Graph<std::string>* res, std::string deactivated){

    Graph<std::string> graph = constructor.createGraph();

    addSupersourceAndSuperSink(graph);

    graph.findVertex(deactivated)->disable();

    edmondsKarp(&graph, "supersource", "supersink");

    *res = graph;

    int maxflow = 0;
    for (int i = 0; i < graph.getNumVertex(); i++){
        if (graph.getVertexSet()[i]->getInfo()[0] == 'C') {
            for (int j = 0; j < graph.getVertexSet()[i]->getIncoming().size(); j++){
                maxflow += graph.getVertexSet()[i]->getIncoming()[j]->getFlow();
            }
        }
    }

    return maxflow;
}
```

Implemented Functionalities

- **Small Algorithm to balance the flow across the network:** O(E + V)
    - Redistributes excess capacity found in the pipes

```
void Manager::balanceFlow(Graph<std::string>* graph) {
    // Identify nodes with unmet demand
    std::vector<Vertex<std::string>*> nodesWithUnmetDemand;
    for (Vertex<std::string>* vertex : graph->getVertexSet()) {
        if (vertex->getInfo()[0] == 'C') {
            double demand = this->cities.at( k: vertex->getInfo()).getDemand();
            double totalInflow = 0;
            for (Edge<std::string>* edge : vertex->getIncoming()) {
                totalInflow += edge->getFlow();
            }
            if (totalInflow < demand) {
                nodesWithUnmetDemand.push_back(vertex);
            }
        }
    }

    // Calculate excess capacity for each pipe
    std::unordered_map<Edge<std::string>*, double> excessCapacity;
    double totalExcessCapacity = 0;
    for (Vertex<std::string>* vertex : graph->getVertexSet()) {
        for (Edge<std::string>* edge : vertex->getAdj()) {
            double capacity = edge->getWeight();
            double flow = edge->getFlow();
            double remainingCapacity = capacity - flow;
            if (remainingCapacity > 0) {
                excessCapacity[edge] = remainingCapacity;
                totalExcessCapacity += remainingCapacity;
            }

        }
    }
```

```
    // Redistribute flow
    for (Vertex<std::string>* vertex : nodesWithUnmetDemand) {
        double demand = this->cities.at( k: vertex->getInfo()).getDemand();
        double totalInflow = 0;
        for (Edge<std::string>* edge : vertex->getIncoming()) {
            totalInflow += edge->getFlow();
        }
        for (Edge<std::string>* edge : vertex->getIncoming()) {
            if (excessCapacity.find( k: edge) != excessCapacity.end()) {
                double redistributionFactor = excessCapacity[edge] / totalExcessCapacity;
                double additionalFlow = redistributionFactor * (demand - totalInflow);
                double currentFlow = edge->getFlow();
                edge->setFlow(currentFlow + additionalFlow);
            }
        }
    }
}
```

User Interface (Menu)

- A text-based menu that allows the user to navigate between the option by inputting a number option.
- This is all done by the *Menu* class.
- All of the user's input is validated.

```cpp
void Menu::printIntro() {
        std::cout << "+-----------------------------------+\n";
        std::cout << "|                                   |\n";
        std::cout << "|        DA PROJECT 1 - G05_2       |\n";
        std::cout << "|                                   |\n";
        std::cout << "+-----------------------------------+\n\n";
}
```

User Interface (Menu)

```
1 - Large DataSet (Continent)
2 - Small DataSet (Madeira)
Select DataSet: 2
```

```
+--------------------------------+
|                                |
|       DA PROJECT 1 - G05_2     |
|                                |
+--------------------------------+

   :::: MENU ::::
0 - Exit
1 - T2.1 - Maximum amount of water that can reach each or a specific city
2 - T2.2 - Water Demand vs. Actual Flow
3 - T3.1 - Water Reservoir Out of Comission Test
4 - T3.2 - Pumping Station out of Comission Test
5 - T3.3 - Pipe out of Comission Test

Choose an option:1
```

```
1 - Each city
2 - A specific city

Choose an option:1
```

```
Each city
Porto Moniz | C_1 -> 18
São Vicente | C_2 -> 34
Santana | C_3 -> 46
Machico | C_4 -> 137
Santa Cruz | C_5 -> 295
Funchal | C_6 -> 664
Câmara de Lobos | C_7 -> 225
Ribeira Brava | C_8 -> 89
Ponta do Sol | C_9 -> 59
Calheta | C_10 -> 76
The total Max Flow is: 1643
```

Highlighted Functionalities

- Constructor Pattern with *GraphConstructor* class that increased modularity of the code.

```
class GraphConstructor {
private:
    std::string citiesFile; /**< File containing city data. */
    std::string reservoirsFile; /**< File containing reservoir data. */
    std::string stationsFile; /**< File containing station data. */
    std::string pipesFile; /**< File containing pipe data. */
    bool isSmallDataSet; /**< Flag indicating whether the dataset is the small one. */

    /**
     * @brief Parses a line of input to create a city object.
     * @param line Line containing city data.
     * @return City object parsed from the line.
     * @par Complexity
       *    - Time: O(1)
       *    - Space: O(1)
     */
    City parseCity(std::string line);
```

Main Difficulties and Participation

- 2.3 point - The creation of an heuristic to balance some metrics on the graph. The implemented algorithm only displays a small improvement.

- **Participation**
  - Davide Teixeira: 60%
  - Ana Ramos: 40%