Water Supply Management System

DA Project

Project by: G07_6

up202207217 Dinis Galvão ,
up202206120 Joana Pimenta,
up202207986 Miguel Sousa

CLASS DIAGRAM

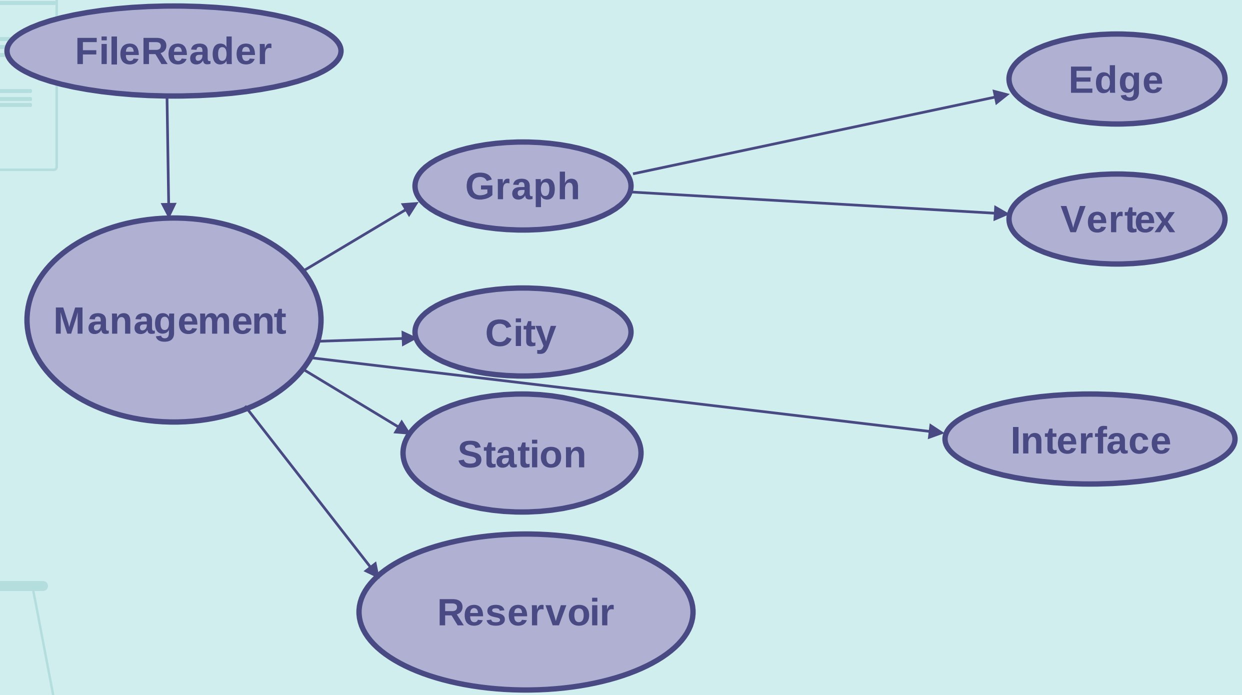


TABLE OF CONTENTS

1

Our File Parser

Management function and FileReader

```cpp
#include "FileReader.h"

std::string trim(const std::string& str) {
    size_t start = 0;
    size_t end = str.length();

    while (start < end && std::isspace(str[start])) {
        start++;
    }

    while (end > start && std::isspace(str[end - 1])) {
        end--;
    }

    return str.substr( pos: start,  n: end - start);
}



FileReader::FileReader(const std::string &fName) {
    file_.open( s: fName);
}

std::vector<std::vector<std::string>> FileReader::getData() {
    std::string line;
    std::getline( &: file_,  &: line);
    while (std::getline( &: file_,  &: line)) {
        std::istringstream iss( str: line);
        std::string value;
        std::vector<std::string> v;
        while (std::getline( &: iss,  &: value,  delim: ',')) {
            value = trim( str: value);
            v.push_back(value);
        }
        data_.push_back(v);
    }
    return data_;
}
```

Our File Parser

FileReader(const std::string &fName)

Class used to read the cvs files.

getData()

This function reads the data from the file and returns it as a vector of vectors of strings.
Each inner vector represents a line from the file, and each string represents a field separated by commas.

Our File Parser

```cpp
Management::Management(int dataSet) : reservoirs_(std::make_unique<std::unordered_map<std::string, Reservoir>>()),
                                      stations_(std::make_unique<std::unordered_map<std::string, Station>>()),
                                      cities_(std::make_unique<std::unordered_map<std::string, City>>()),
                                      waterNetwork_(std::make_unique<Graph<std::string>>())
{

    FileReader reservoirsData("../Data/SmallDataSet/Reservoirs_Madeira.csv");
    FileReader stationsData("../Data/SmallDataSet/Stations_Madeira.csv");
    FileReader citiesData("../Data/SmallDataSet/Cities_Madeira.csv");
    FileReader pipesData("../Data/SmallDataSet/Pipes_Madeira.csv");


    if (dataSet){
        reservoirsData = FileReader("../Data/BigDataSet/Reservoir.csv");
        stationsData = FileReader("../Data/BigDataSet/Stations.csv");
        citiesData = FileReader("../Data/BigDataSet/Cities.csv");
        pipesData = FileReader("../Data/BigDataSet/Pipes.csv");
    }
  for (std::vector<std::string> line : reservoirsData.getData()){
    reservoirs_->insert({ line.at(3), Reservoir(line.at(0), line.at(1), line.at(2), line.at(3), std::stoi(line.at(4)))});
  }
  for (std::vector<std::string> line : stationsData.getData()){
    stations_->insert({ line.at(1), Station(line.at(0), line.at(1))});
  }
  for (std::vector<std::string> line : citiesData.getData()){
    cities_->insert({ line.at(2), City(line.at(0), line.at(1), line.at(2), std::stof(line.at(3)), std::stoi(line.at(4)))});
  }

  for (const auto& reservoir : *reservoirs_){
    waterNetwork_->addVertex(reservoir.first);
  }
  for (const auto& station : *stations_){
    waterNetwork_->addVertex(station.first);
  }
  for (const auto& city : *cities_){
    waterNetwork_->addVertex(city.first);
  }

  for (const std::vector<std::string> &line : pipesData.getData()){
    std::string pointA = line.at(0);
    std::string pointB = line.at(1);
    double capacity = std::stod(line.at(2));
    int direction = std::stoi(line.at(3));

    if (direction){
      waterNetwork_->addEdge(pointA, pointB, capacity);
    }
    else{
      waterNetwork_->addBidirectionalEdge(pointA, pointB, capacity);
    }
  }
}
```

The management function in the Management class initializes its object based on the dataset selected.
This function is responsible for the parsing with the help of the FileReader

2

The Graph Used

Data set representation

The Graph Used

```cpp
    for (const auto& reservoir : *reservoirs_){
        waterNetwork_->addVertex(reservoir.first);
    }
    for (const auto& station : *stations_){
        waterNetwork_->addVertex(station.first);
    }
    for (const auto& city : *cities_){
        waterNetwork_->addVertex(city.first);
    }

    for (const std::vector<std::string> &line : pipesData.getData()){
        std::string pointA = line.at(0);
        std::string pointB = line.at(1);
        double capacity = std::stod(line.at(2));
        int direction = std::stoi(line.at(3));

        if (direction){
            waterNetwork_->addEdge(pointA, pointB, capacity);
        }
        else{
            waterNetwork_->addBidirectionalEdge(pointA, pointB, capacity);
        }
    }
}
```

We organized our data in only one graph by using the cities, reservoirs and pumping stations as vertexes and the pipes as edges.

Previously we organized the information regarding the cities, reservoirs and pumping stations in unorder maps called reservoirs_ , stations_ and cities_

3

Functionalities and Algorithms

Management and Utils

```cpp
double Management::maxFlow(const Graph<std::string>& g, const std::string& code){

    std::vector<std::vector<std::string>> vector_of_paths;

    Graph<std::string> max_flow=g;

    max_flow.addVertex("sink");
    max_flow.addVertex("source");
    Vertex<std::string> *source=max_flow.findVertex("source");
    Vertex<std::string> *sink=max_flow.findVertex("sink");

    for(Vertex<std::string> *v : max_flow.getVertexSet()){
        if(v->getInfo().substr(0, 1)=="R"){
            for(const auto& r :*reservoirs_){
                if(r.first == v->getInfo()){
                    source->addEdge(v, r.second.getMaxDelivery());
                }
            }

        }
        if(v->getInfo().substr(0, 1)=="C"){
            for(const auto& c :*cities_){
                if(c.first == v->getInfo()){
                    v->addEdge(sink, c.second.getDemand());
                }
            }
        }

    }

    for(Vertex<std::string> *v:max_flow.getVertexSet()){
        for(Edge<std::string> *e:v->getAdj()){
            e->setFlow(0);
        }

    }

    while(augmentationPathFinder(&max_flow,source,sink)){
        std::vector<double> val;
        std::vector<std::vector<std::string>> path;
        double mini=INF;

        for(Vertex<std::string> *v=sink; v!=source;){
            Edge<std::string> *e=v->getPath();
            if(v==e->getDest()){
                mini=std::min(mini,e->getWeight() - e->getFlow());
                v=e->getOrig();
            }
            else{
                mini=std::min(mini,e->getFlow());
                v=e->getDest();
            }
        }
        for(Vertex<std::string> *v=sink; v!=source;){
            std::vector<std::string> step;
            Edge<std::string> *e=v->getPath();
            double flow=e->getFlow();
            if(v==e->getDest()){
                e->setFlow(flow+mini);
                v=e->getOrig();
                step.push_back(e->getDest()->getInfo());
                step.push_back(e->getOrig()->getInfo());
            }
            else{
                e->setFlow(flow-mini);
                v=e->getDest();
                step.push_back(e->getOrig()->getInfo());
                step.push_back(e->getDest()->getInfo());
            }

            step.push_back(std::to_string(e->getFlow()));
            path.push_back(step);
        }
        flowPaths_.push_back(path);
    }
```

maxFlow

This function calculates the maximum flow in the given water network graph using the Edmonds-Karp algorithm . It adds a source and a sink vertex to the graph and connects them to the reservoirs and cities, respectively. Then, it iteratively finds augmentation paths from the source to the sink using BFS until no more paths can be found. During each iteration, it updates the flow along the found paths. Finally, it retrieves the maximum flow value from the incoming edges of the target vertex specified by the code.

Complexity O(V * E^2), where V is the number of vertices in the graph and E is the number of edges.

```cpp
    for (auto v : waterNetwork_->getVertexSet()){
        for (auto e : v->getAdj()){
            Vertex<std::string> *dest = e->getDest();
            edgesFlow_.insert({key(v->getInfo(), dest->getInfo()), std::to_string(e->getFlow())});
        }
    }

    double res=0;
    for(Edge<std::string> *e: max_flow.findVertex(code)->getIncoming()){
        res+=e->getFlow();
    }
    return res;
}
```

CheckWaterNeeds

```cpp
std::vector<std::vector<std::string>> Management::checkWaterNeeds() {
    std::vector<std::vector<std::string>> result;
    for (const auto& city : *cities_) {
        float waterNeeded = city.second.getDemand();
        double waterDelivered = 0;

        for (auto e : waterNetwork_->findVertex(city.first)->getIncoming()){
            waterDelivered += std::stod(edgesFlow_[key(e->getOrig()->getInfo(), e->getDest()->getInfo())]);
        }

        std::cout << city.first << " " << waterDelivered << std::endl;

        if (waterDelivered < waterNeeded) {
            std::vector<std::string> v;
            v.emplace_back(city.first);
            v.emplace_back(std::to_string(waterNeeded - waterDelivered));
            v.emplace_back(city.second.getName());
            v.emplace_back(std::to_string(waterNeeded));
            result.emplace_back(v);

        }
    }
    return result;
}
```

This function calculates the water needs for all cities in the water network and computes any deficits. It iterates through each city, retrieves its demand, calculates the total water delivered to the city from incoming edges in the water network graph, and compares it with the demand. If the delivered water is less than the demand, information about the city with the deficit is added to the result vector.

complexity O(N + E), where N is the number of vertices (cities) and E is the number of edges (pipes)

CheckWaterNeeds pump / reservoir / pipe

```cpp
std::unordered_map<std::string, std::string> Management::checkWaterNeedsReservoir(const std::vector<std::wstring>& reservoirs){

    std::unordered_map<std::string, std::string> res;

    Graph<std::string> g = createGraphCopy(*waterNetwork_);

    for (const auto& ws : reservoirs){
        g.removeVertex(converter1.to_bytes(ws));
    }
    maxFlow(g, "sink");
    for (const auto& city : *cities_){
        double val = 0;
        for(Edge<std::string> *e: g.findVertex(city.first)->getIncoming()){
            val += e->getFlow();
        }
        res.insert({city.first, std::to_string(val)});
    }
    return res;
}
std::unordered_map<std::string, std::string> Management::checkWaterNeedsPumps(const std::vector<std::wstring>& pumps){

    std::unordered_map<std::string, std::string> res;

    Graph<std::string> g = createGraphCopy(*waterNetwork_);

    for (const auto& ws : pumps){
        g.removeVertex(converter1.to_bytes(ws));
    }
    maxFlow(g, "sink");
    for (const auto& city : *cities_){
        double val = 0;
        for(Edge<std::string> *e: g.findVertex(city.first)->getIncoming()){
            val += e->getFlow();
        }
        res.insert({city.first, std::to_string(val)});
    }
    return res;
}
std::unordered_map<std::string, std::string> Management::checkWaterNeedsPipes(const std::vector<std::wstring>& pumps){

    std::unordered_map<std::string, std::string> res;

    Graph<std::string> g = createGraphCopy(*waterNetwork_);

    for (const auto& ws : pumps){
        std::string dest = stringDivider(ws, 0, '|');
        std::string orig = stringDivider(ws, 1, '|');
        g.removeEdge(orig, dest);
    }
    maxFlow(g, "sink");
    for (const auto& city : *cities_){
        double val = 0;
        for(Edge<std::string> *e: g.findVertex(city.first)->getIncoming()){
            val += e->getFlow();
        }
        res.insert({city.first, std::to_string(val)});
    }
    return res;
}
```

This functions all work in similar way, this function calculates the water needs for cities after removing specified arguments from the water network. It creates a copy of the water network graph, removes the specified arguments, calculates the maximum flow in the modified graph, and then computes the water needs for each city based on the incoming flow to their corresponding vertices in the graph, without needing further calls to the maxFlow function

complexity O(N + E), where N is the number of vertices (cities) and E is the number of edges (pipes)

Balancing algorithm

1. For each pipe in the network:
   - Calculate the difference between flow and capacity
   - Store the difference for each pipe

2. Compute the average and maximum difference across all pipes:
   - Initialize sum_difference = 0
   - Initialize max_difference = 0
   - For each pipe:
     - Update sum_difference by adding the pipe's difference
     - Update max_difference if the pipe's difference is greater

3. If the total flow exceeds the total capacity:
   - Calculate the excess flow (total flow - total capacity)
   - Initialize remaining_excess_flow = excess flow

   - For each pipe:
     - If the pipe has spare capacity (difference > 0):
       - Calculate the proportion of excess flow to distribute evenly among pipes with spare capacity
       - Increase the flow on the pipe by the calculated proportion of excess flow
       - Update remaining_excess_flow by subtracting the distributed excess flow
       - Break the loop if remaining_excess_flow becomes zero

4. Recalculate the difference between flow and capacity for each pipe
   - Repeat step 1

5. Check if the average and maximum difference have improved:
   - Calculate the new average difference and maximum difference (similar to step 2)
   - Compare the new average and maximum differences with the previous values

6. Repeat steps 3-5 until the metrics stop improving or until a certain number of iterations is reached.

7. Output the final metrics.

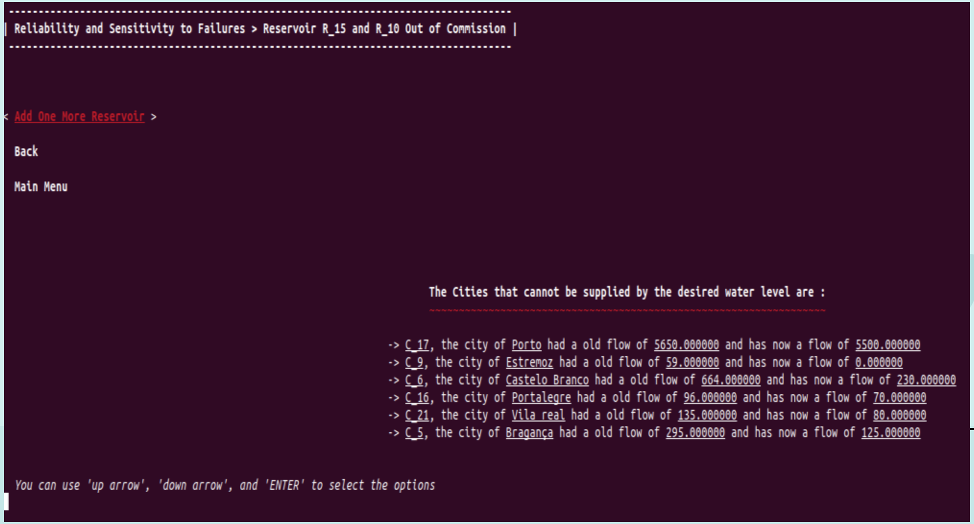
```cpp
void Management::balanceBasicMetrics (const Graph<std::string>& g){
    double sum_diff = 0.0;
    double sum_diff_squared = 0.0;
    double max_diff = 0.0;
    std::string d;
    std::string o;

    for (const auto& pair : pipes_) {
        o = stringDivider(converter.from_bytes(pair.first), 0, '|');
        d = stringDivider(converter.from_bytes(pair.first), 1, '|');
        for (auto e : g.findVertex(o)->getAdj()){
            if (e->getDest()->getInfo() == d){
                double diff = pair.second - e->getFlow();
                sum_diff += diff;
                sum_diff_squared += diff * diff;
                max_diff = std::max(max_diff, std::abs(diff));
            }
        }

    }

    double average_diff = sum_diff / pipes_.size();
    double variance_diff = sum_diff_squared / pipes_.size() - (average_diff * average_diff);

    std::wcout << L"Initial Metrics:" << std::endl;
    std::wcout << L"Average Difference: " << average_diff << std::endl;
    std::wcout << L"Variance of Difference: " << variance_diff << std::endl;
    std::wcout << L"Maximum Difference: " << max_diff << std::endl;
}

Graph<std::string> Management::balance(const Graph<std::string>& g){
    Graph<std::string> a = createGraphCopy(g);
    maxFlow(a, "source");
    //Logica que procura as bifurcaçoes, analiza a que tem mais espaço e redireciona mais agua
    return a;
}
```

4

Interface

Interface

The Interface forbids the print of the user's input, directing it to the basucInputResponse, where it's handled. The two main variables are selected and location, and they cooperate in a bunch of switch cases, bringing the Interface to life.



5

Highlights

Highlights

The quick parsing and data structuring, good solutions for the problems presented to us and a very user friendly and intuitive interface are some of the features that we are most proud of.

Participation

Members	UP	Participation
Dinis Galvão	up202206120	100%
Joana Pimenta	up202207217	100%
Miguel Sousa	up202207986	100%

We had some problems with the cmake between computers and sometimes it requires reloading the cmake.
We also tried to pass the paths in the graph and the flows fof does path to a data structure so we minimise calls of the maxflow function but we couldn't find a solution that worked completely in time.