Water Supply Management Project 1 DA 23/24

•••

G18_3:
David Ranito
Pedro João
Tiago Torres

Classes Diagram

Management -> Management.cpp, Management.h

Graph -> Graph.cpp, Graph.h

Menu -> Menu.cpp, Menu.h

Auxiliar -> Auxiliar.cpp, Auxiliar.h, Folders -> Project1LargeDataSet, Project1DataSetSmall

ServicePoint -> ServicePoint.cpp, ServicePoint.h

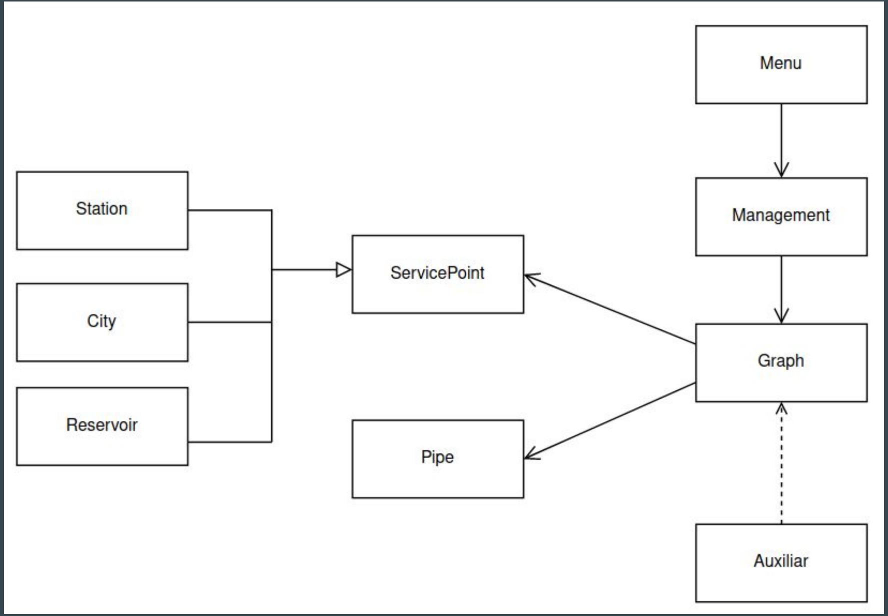
Pipe -> Pipe.cpp, Pipe.h

Reservoir -> Reservoir.cpp, Reservoir.h

Station -> Station.cpp, Station.h

City -> City.cpp, City.h

Reservoir -> Reservoir.cpp, Reservoir.h

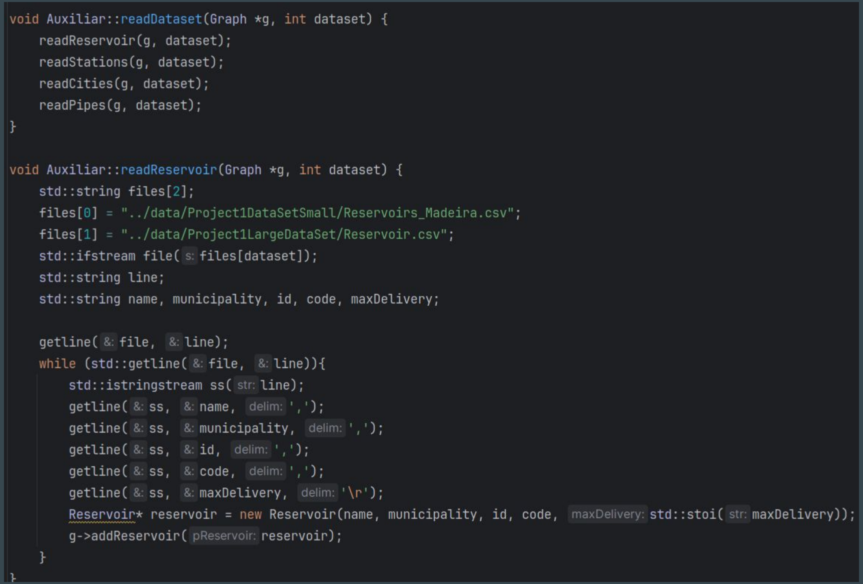


Reading the Dataset

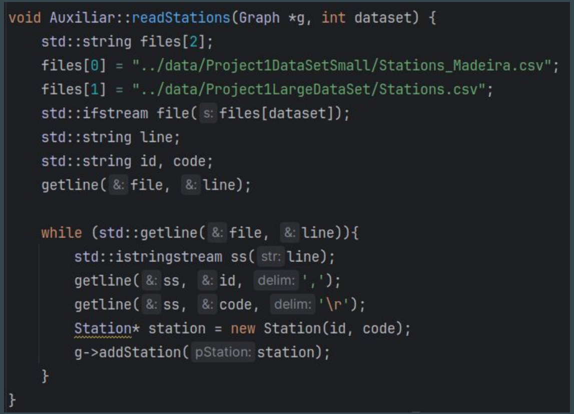
To read the given data, 5 function were added in the “Auxiliar” class:

- readDataset
- readReservoir
- readStations
- readCities
- readPipes

These functions read the respective file line by line, create the appropriate object and adds them to the graph.



```
void Auxiliar::readDataset(Graph *g, int dataset) {
    readReservoir(g, dataset);
    readStations(g, dataset);
    readCities(g, dataset);
    readPipes(g, dataset);
}

void Auxiliar::readReservoir(Graph *g, int dataset) {
    std::string files[2];
    files[0] = "../data/Project1DataSetSmall/Reservoirs_Madeira.csv";
    files[1] = "../data/Project1LargeDataSet/Reservoir.csv";
    std::ifstream file( s: files[dataset]);
    std::string line;
    std::string name, municipality, id, code, maxDelivery;

    getline( &: file,  &: line);
    while (std::getline( &: file,  &: line)){
        std::istringstream ss( str: line);
        getline( &: ss,  &: name,  delim: ',');
        getline( &: ss,  &: municipality,  delim: ',');
        getline( &: ss,  &: id,  delim: ',');
        getline( &: ss,  &: code,  delim: ',');
        getline( &: ss,  &: maxDelivery,  delim: '\r');
        Reservoir* reservoir = new Reservoir(name, municipality, id, code,  maxDelivery: std::stoi( str: maxDelivery));
        g->addReservoir( pReservoir: reservoir);
    }
}
```

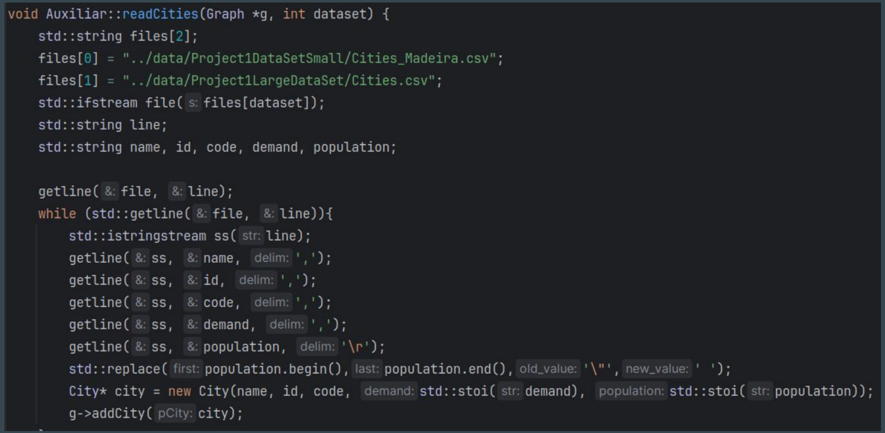


```
void Auxiliar::readStations(Graph *g, int dataset) {
    std::string files[2];
    files[0] = "../data/Project1DataSetSmall/Stations_Madeira.csv";
    files[1] = "../data/Project1LargeDataSet/Stations.csv";
    std::ifstream file( s: files[dataset]);
    std::string line;
    std::string id, code;
    getline( &: file,  &: line);

    while (std::getline( &: file,  &: line)){
        std::istringstream ss( str: line);
        getline( &: ss,  &: id,  delim: ',');
        getline( &: ss,  &: code,  delim: '\r');
        Station* station = new Station(id, code);
        g->addStation( pStation: station);
    }
}
```

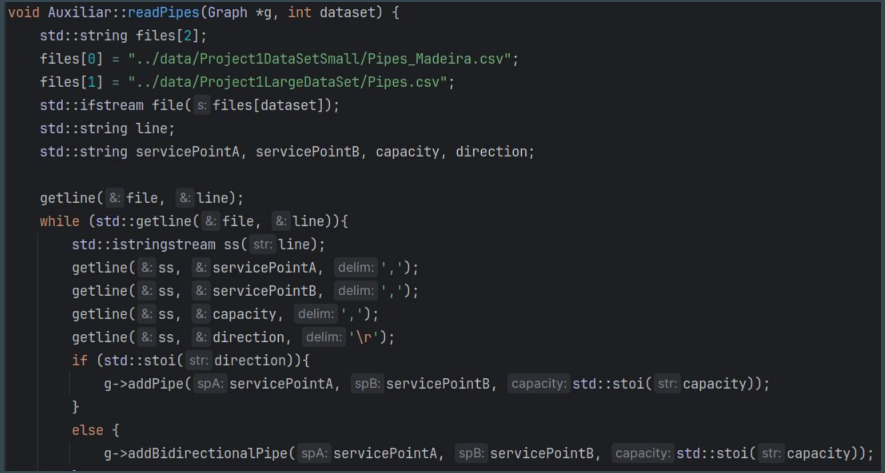
Reading the Dataset

By calling the functions addCity or addPipe the new objects are then added to the appropriate data structure and to the graph.

As an example, in the case of Pipe, it is added to the graph’s pipeSet, pipeByEnds (unordered_map to facilitate pipe search) and to the incoming and adjacent vectors of the respective servicePoints.



```
void Auxiliar::readCities(Graph *g, int dataset) {
    std::string files[2];
    files[0] = "../data/Project1DataSetSmall/Cities_Madeira.csv";
    files[1] = "../data/Project1LargeDataSet/Cities.csv";
    std::ifstream file( s: files[dataset]);
    std::string line;
    std::string name, id, code, demand, population;

    getline( &: file,  &: line);
    while (std::getline( &: file,  &: line)){
        std::istringstream ss( str: line);
        getline( &: ss,  &: name,  delim: ',');
        getline( &: ss,  &: id,  delim: ',');
        getline( &: ss,  &: code,  delim: ',');
        getline( &: ss,  &: demand,  delim: ',');
        getline( &: ss,  &: population,  delim: '\r');
        std::replace( first: population.begin(), last: population.end(), old_value: '\"', new_value: ' ');
        City* city = new City(name, id, code,  demand: std::stoi( str: demand),  population: std::stoi( str: population));
        g->addCity( pCity: city);
```

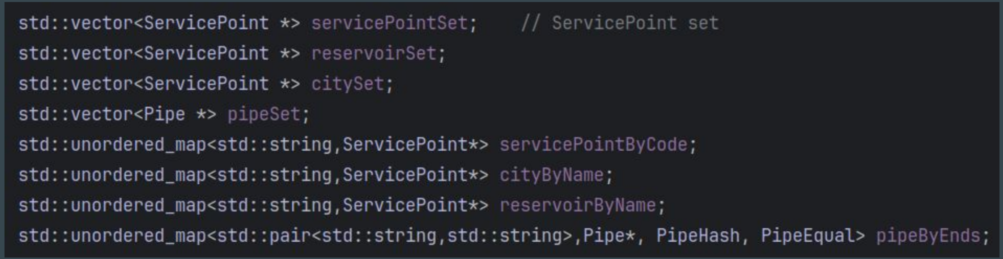


```
void Auxiliar::readPipes(Graph *g, int dataset) {
    std::string files[2];
    files[0] = "../data/Project1DataSetSmall/Pipes_Madeira.csv";
    files[1] = "../data/Project1LargeDataSet/Pipes.csv";
    std::ifstream file( s: files[dataset]);
    std::string line;
    std::string servicePointA, servicePointB, capacity, direction;

    getline( &: file,  &: line);
    while (std::getline( &: file,  &: line)){
        std::istringstream ss( str: line);
        getline( &: ss,  &: servicePointA,  delim: ',');
        getline( &: ss,  &: servicePointB,  delim: ',');
        getline( &: ss,  &: capacity,  delim: ',');
        getline( &: ss,  &: direction,  delim: '\r');
        if (std::stoi( str: direction)){
            g->addPipe( spA: servicePointA,  spB: servicePointB,  capacity: std::stoi( str: capacity));
        }
        else {
            g->addBidirectionalPipe( spA: servicePointA,  spB: servicePointB,  capacity: std::stoi( str: capacity));
```

Graph Description

The graph is built with ServicePoints as vertices and Pipes as edges.

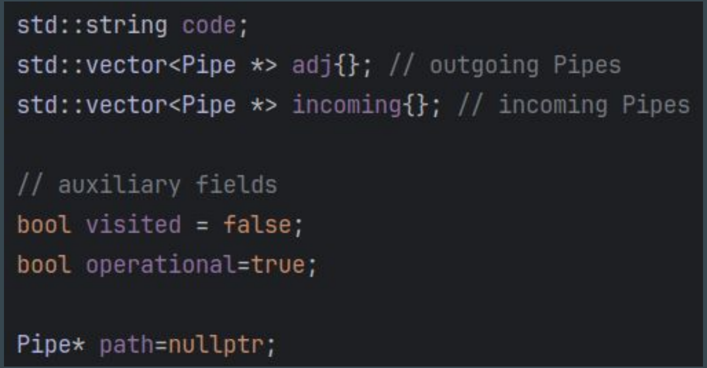
It has vectors to store them and hashmaps to allow constant-time lookup for a specific ServicePoint/Pipe.



```
std::vector<ServicePoint *> servicePointSet;    // ServicePoint set
std::vector<ServicePoint *> reservoirSet;
std::vector<ServicePoint *> citySet;
std::vector<Pipe *> pipeSet;
std::unordered_map<std::string,ServicePoint*> servicePointByCode;
std::unordered_map<std::string,ServicePoint*> cityByName;
std::unordered_map<std::string,ServicePoint*> reservoirByName;
std::unordered_map<std::pair<std::string,std::string>,Pipe*, PipeHash, PipeEqual> pipeByEnds;
```

Each servicePoint has a group of adjacent and incoming pipes.

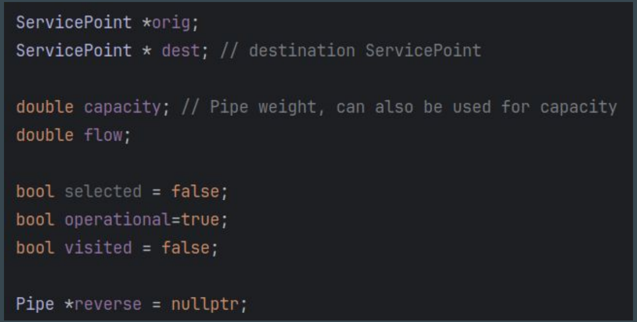
Also has some auxiliary atributtes to help compute the algorithms.



```
std::string code;
std::vector<Pipe *> adj{}; // outgoing Pipes
std::vector<Pipe *> incoming{}; // incoming Pipes

// auxiliary fields
bool visited = false;
bool operational=true;

Pipe* path=nullptr;
```

Graph Description - continuation

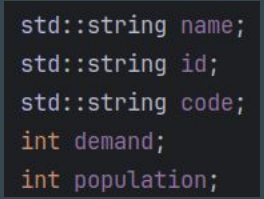
Each pipe has a origin and a destination servicePoint, it also contains capacity, flow and auxiliary fields.



```
ServicePoint *orig;
ServicePoint * dest; // destination ServicePoint

double capacity; // Pipe weight, can also be used for capacity
double flow;

bool selected = false;
bool operational=true;
bool visited = false;

Pipe *reverse = nullptr;
```

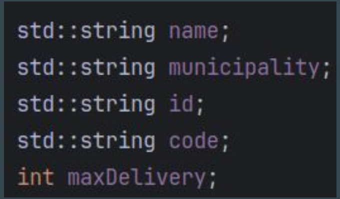
Each servicePoint type has attributes that characterizes them.

City:



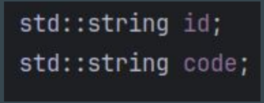
```
std::string name;
std::string id;
std::string code;
int demand;
int population;
```

Reservoir:



```
std::string name;
std::string municipality;
std::string id;
std::string code;
int maxDelivery;
```

Station:



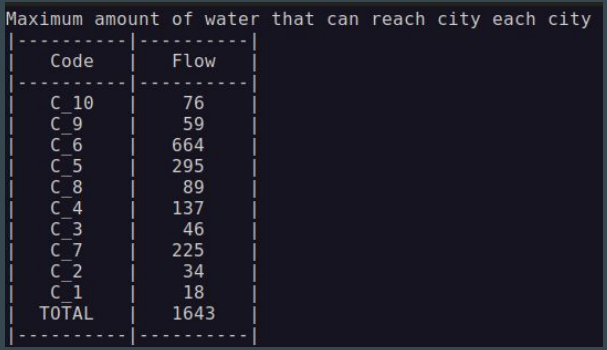
```
std::string id;
std::string code;
```

Max Flow

- The default algorithm used was Edmond's Karp.
- A supersink and a supersource were used to get the maximum flow to each city.
- The flow leaving a reservoir is restricted by its max delivery, by setting the capacity of the pipes that leave the supersource to the max delivery
- The maximum flow reaching a city is always restricted by its demand, by setting the capacity of the pipes that go to the supersink to the city's demand.
- To get the max flow of a city only a supersource was used.

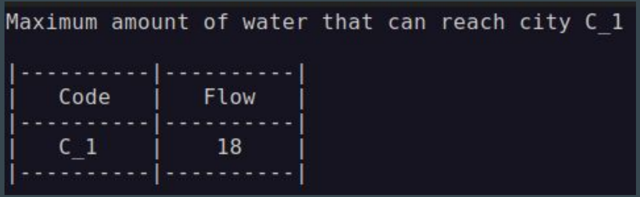
Time Complexity: $O(S*P^2)$

each city:



```
Maximum amount of water that can reach city each city
|----------|----------|
|   Code   |   Flow   |
|----------|----------|
|   C_10   |    76    |
|   C_9    |    59    |
|   C_6    |   664    |
|   C_5    |   295    |
|   C_8    |    89    |
|   C_4    |   137    |
|   C_3    |    46    |
|   C_7    |   225    |
|   C_2    |    34    |
|   C_1    |    18    |
|  TOTAL   |   1643   |
|----------|----------|
```

a city:

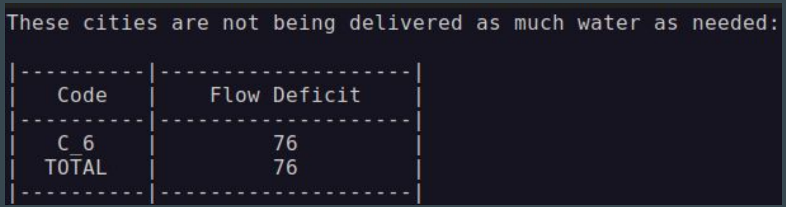


```
Maximum amount of water that can reach city C_1

|----------|----------|
|   Code   |   Flow   |
|----------|----------|
|   C_1    |    18    |
|----------|----------|
```

Network configuration meets the water needs of all customers?

- A city has a flow deficit if its demand is not corresponded, this means that the flow coming to the city needs to be equal to the demand.

Time Complexity: $O(S*P^2)$



```
These cities are not being delivered as much water as needed:

|----------|--------------------|
|   Code   |    Flow Deficit    |
|----------|--------------------|
|   C_6    |         76         |
|  TOTAL   |         76         |
|----------|--------------------|
```

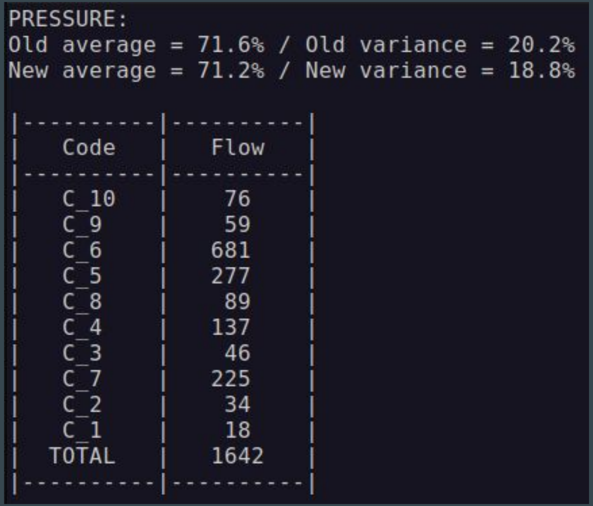
Balance the load across the network

After the computing the max flow, the pressure of the pipes (flow/capacity) is capped to the average pressure of the network and the flow is reset to 0.

The max flow algorithm runs again (this time with the capacities capped).

In the end the max flow algorithm runs again, without resetting the flow and with the original capacities.

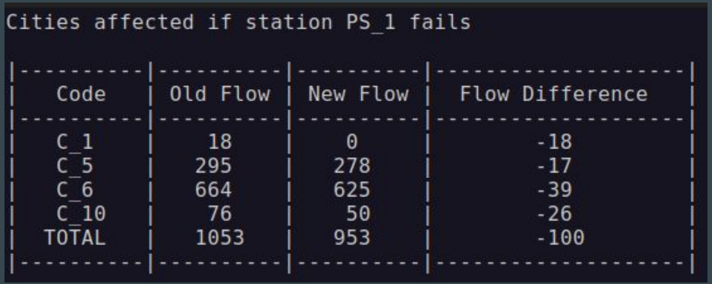
Time Complexity: O(S*P²)



```
PRESSURE:
Old average = 71.6% / Old variance = 20.2%
New average = 71.2% / New variance = 18.8%

|----------|----------|
|   Code   |   Flow   |
|----------|----------|
|   C_10   |    76    |
|   C_9    |    59    |
|   C_6    |   681    |
|   C_5    |   277    |
|   C_8    |    89    |
|   C_4    |   137    |
|   C_3    |    46    |
|   C_7    |   225    |
|   C_2    |    34    |
|   C_1    |    18    |
|  TOTAL   |   1642   |
|----------|----------|
```

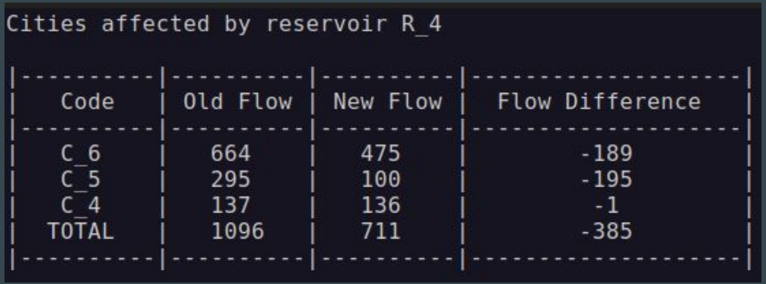
Reliability and Sensitivity to Failures

PS failure



```
Cities affected if station PS_1 fails

|----------|----------|----------|--------------------|
|   Code   | Old Flow | New Flow |  Flow Difference   |
|----------|----------|----------|--------------------|
|   C_1    |    18    |    0     |        -18         |
|   C_5    |   295    |   278    |        -17         |
|   C_6    |   664    |   625    |        -39         |
|   C_10   |    76    |    50    |        -26         |
|  TOTAL   |   1053   |   953    |        -100        |
|----------|----------|----------|--------------------|
```

Reservoir unavailable



```
Cities affected by reservoir R_4

|----------|----------|----------|--------------------|
|   Code   | Old Flow | New Flow |  Flow Difference   |
|----------|----------|----------|--------------------|
|   C_6    |   664    |   475    |        -189        |
|   C_5    |   295    |   100    |        -195        |
|   C_4    |   137    |   136    |         -1         |
|  TOTAL   |   1096   |   711    |        -385        |
|----------|----------|----------|--------------------|
```

- ServicePoints have “operational” boolean attribute.
- To simulate a failure, it is turned to “false”.
- Run the Max Flow algorithm, ignoring non-operational pipes.
- It compares the new flow in each city with the default flow, and if it decreased, it means it was affected by the rupture.

Time Complexity: $O(S*P^2)$

Reliability and Sensitivity to Failures

Reservoir unavailable

Alternatives to running the entire max flow algorithm every time:

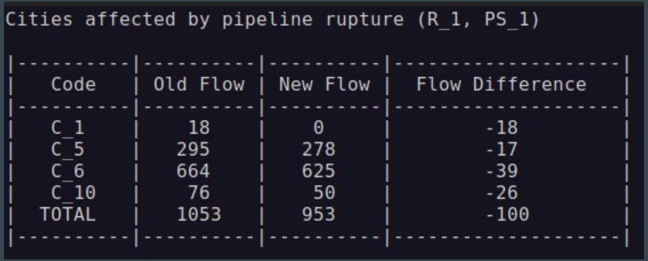
1. Running the max flow algorithm only in the connected component containing the reservoir.

   To obtain the connected components would only take O(S+P) and if the reservoir happens to be in an area isolated from others the max flow algorithm is computed with a smaller graph.

2. Saving the results of a previously computed max flow algorithm without the reservoir.

   By saving the results, we only run the max flow the first time the reservoir fails and store it for the next time. A vector containing a pair <city affected, flow difference> per reservoir would do the trick.

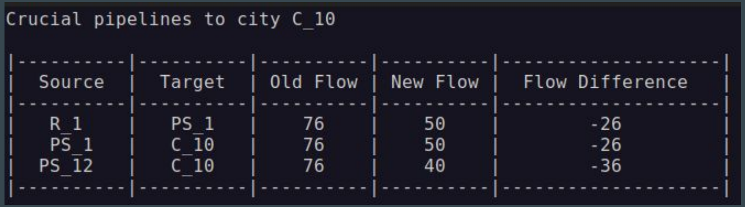
Pipeline Rupture



```
Cities affected by pipeline rupture (R_1, PS_1)

|---------|---------|---------|-------------------|
|   Code  | Old Flow | New Flow |  Flow Difference  |
|---------|---------|---------|-------------------|
|   C_1   |   18    |    0    |        -18        |
|   C_5   |   295   |   278   |        -17        |
|   C_6   |   664   |   625   |        -39        |
|   C_10  |   76    |   50    |        -26        |
|  TOTAL  |  1053   |   953   |       -100        |
|---------|---------|---------|-------------------|
```

- Pipes have “operational” boolean attribute.
- To simulate a rupture, it is turned to “false”.
- Run the Max Flow algorithm, ignoring non-operational pipes.
- It compares the new flow in each city with the default flow, and if it decreased, it means it was affected by the rupture.

Crucial pipelines to a city



```
Crucial pipelines to city C_10

|---------|---------|---------|---------|------------------|
|  Source |  Target | Old Flow | New Flow |  Flow Difference |
|---------|---------|---------|---------|------------------|
|   R_1   |   PS_1  |   76    |   50    |       -26        |
|   PS_1  |   C_10  |   76    |   50    |       -26        |
|  PS_12  |   C_10  |   76    |   40    |       -36        |
|---------|---------|---------|---------|------------------|
```

- For each Pipe stimulate a rupture.
- Run the Max Flow algorithm, ignoring non-operational pipes.
- If the pipe rupture affected the city flow, then the pipe is crucial.

Time Complexity: $O(S*P^2)$

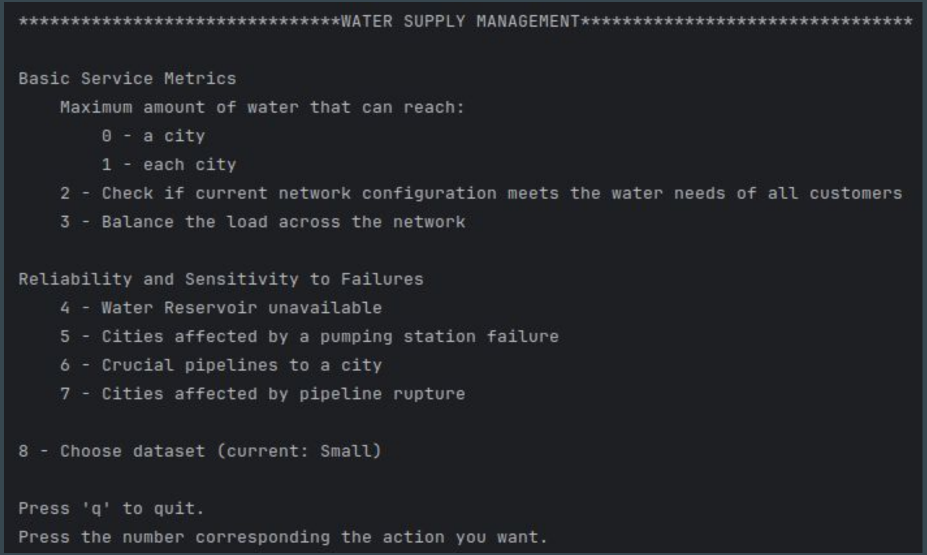
User Interface

The interface is implemented with the Menu class, that is responsible for managing the execution of the program as wished by the user.

All functionalities are displayed in the console by menus.

For each menu the user can choose one option with the corresponding string and then follow the steps that are shown.

The output of the program is saved in a file.



```
*******************************WATER SUPPLY MANAGEMENT********************************

Basic Service Metrics
    Maximum amount of water that can reach:
        0 - a city
        1 - each city
    2 - Check if current network configuration meets the water needs of all customers
    3 - Balance the load across the network

Reliability and Sensitivity to Failures
    4 - Water Reservoir unavailable
    5 - Cities affected by a pumping station failure
    6 - Crucial pipelines to a city
    7 - Cities affected by pipeline rupture

8 - Choose dataset (current: Small)

Press 'q' to quit.
Press the number corresponding the action you want.
```

Highlights

- ❏ A good planning and a good graph structure contributed for the success of this project.
- ❏ Although it was difficult to come up with a balancing algorithm, we implemented one that successfully lowers the average and the variance of the pipe's pressures.
- ❏ For the complexity involved, the speed and responsiveness of the program is very good.
- ❏ The interface has a good look, is intuitive and flexible for user input. The dataset can be chosen and easily changed.

Conclusion

During the project the biggest struggle was to find an algorithm to balance the graph flow.

In general, we accomplished everything that was asked and also added our personal touch.

All elements equally contributed to the project.