DA Project 1 – Water System

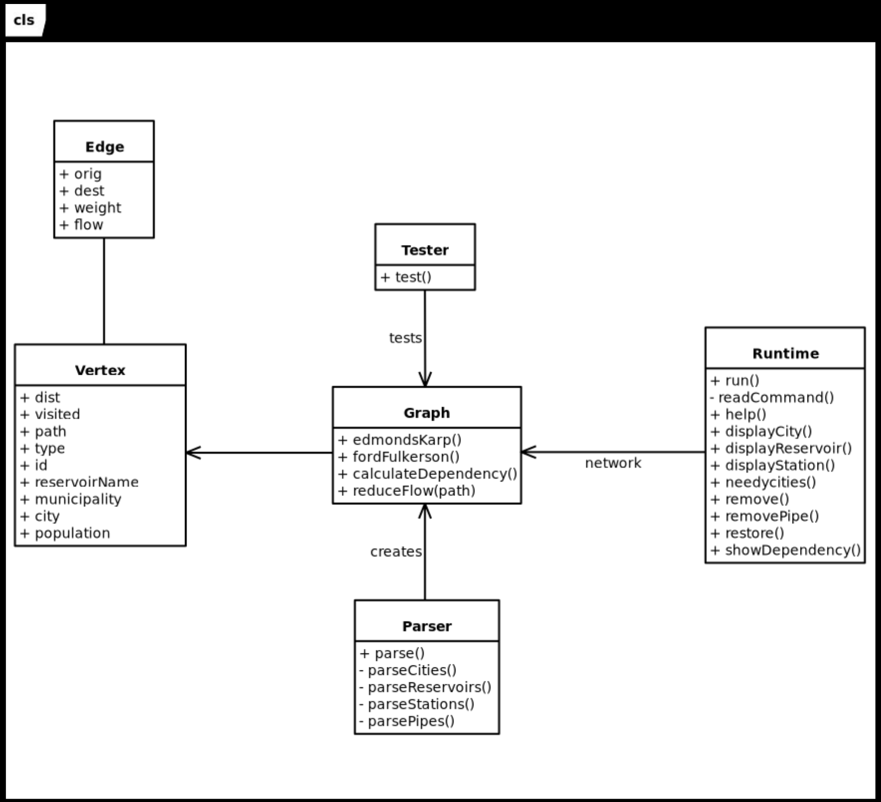
Class 17, Group 5

Introduction

- This project consists of a simplified simulation of a real-life water network
- The water originates from the various reservoirs, travels through pipes and stations until arriving at its final destination, the cities
- In order to simulate said system, 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 directed 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

```
[4.7M]  .
├── [ 553]  build.ninja
├── [1.1K]  CMakeLists.txt
├── [ 11K]  csv
│   ├── [ 698]  Cities.csv
│   ├── [ 338]  Cities_Madeira.csv
│   ├── [3.1K]  Pipes.csv
│   ├── [ 739]  Pipes_Madeira.csv
│   ├── [ 813]  Reservoirs.csv
│   ├── [ 220]  Reservoirs_Madeira.csv
│   ├── [ 804]  Stations.csv
│   └── [ 114]  Stations_Madeira.csv
├── [4.5M]  doc
│   ├── [1.7M]  Description.pdf
│   ├── [2.4M]  LargeDataSetMap.pdf
│   ├── [370K]  Map-Madeira.pdf
│   ├── [ 17K]  PRJ1.odp
│   └── [ 86K]  PRJ1.pptx
├── [124K]  Doxyfile
├── [ 16K]  lib
│   ├── [1.1K]  Edge.hpp
│   ├── [ 499]  GraphException.hpp
│   ├── [3.2K]  Graph.hpp
│   ├── [ 477]  Macros.hpp
│   ├── [ 993]  Parser.hpp
│   ├── [ 244]  Runtime.hpp
│   ├── [1.1K]  Tests.hpp
│   └── [3.9K]  Vertex.hpp
├── [1.2K]  LICENSE
├── [ 164]  Makefile
├── [3.2K]  README.md
└── [ 35K]  src
    ├── [ 782]  Edge.cpp
    ├── [8.1K]  Graph.cpp
    ├── [ 838]  main.cpp
    ├── [1.2K]  Makefile
    ├── [5.1K]  Parser.cpp
    ├── [ 11K]  Tests.cpp
    └── [3.8K]  Vertex.cpp
```

Parsing (1/2)

- This function is called on the main.cpp

```cpp
Graph parse(
        const std::string cities_csv,
        const std::string pipes_csv,
        const std::string reservoirs_csv,
        const std::string stations_csv) {
    Graph network;

    parseCities(network, cities_csv);
    std::cout << WHITE << std::setw(17) << "Parse Cities " << std::setw(18) << cities_csv << GREEN << " OK" << '\n';

    parseReservoirs(network, reservoirs_csv);
    std::cout << WHITE << std::setw(17) << "Parse Reservoirs " << std::setw(18) << reservoirs_csv << GREEN << " OK"
              << '\n';

    parseStations(network, stations_csv);
    std::cout << WHITE << std::setw(17) << "Parse Stations " << std::setw(18) << stations_csv << GREEN << " OK" << '\n';

    parsePipes(network, pipes_csv);
    std::cout << WHITE << std::setw(17) << "Parse Pipes " << std::setw(18) << pipes_csv << GREEN << " OK" << WHITE
              << '\n';

    return network;
}
```

Parsing (2/2)

- This fetches the data from the respective CSV file and stores it in temporary variables used to create the Vertex
- The code is ignored
- The other functions (parsePipes, parseStations and parseReservoirs) are very similar to this one

```cpp
void parseCities(Graph &network, const std::string file) {
    std::fstream fs(file);
    if (check_file(file, fs)) {
        std::string city, id, demand, population;
        getline(fs, city);
        while (getline(fs, city, ',')) {
            getline(fs, id, ',');
            getline(fs, demand, ',');
            getline(fs, demand, ',');
            getline(fs, population);
            unsigned int d = std::stoul(demand);
            Vertex *v = new Vertex(
                    CITY,
                    (unsigned int) (std::stoul(id) * 10 + 1),
                    "",
                    "",
                    city,
                    (unsigned int) std::stoul(population));
            v->addEdge(network.getSink(), d);
            network.addVertex(v);
        }
    }
}
```

Vertex

- Serves as a station, reservoir or city (only one at a time)
- Used as pointers throughout the project
- Methods consist of getters, setters and addition/removal of Edges
- Because we wanted each Vertex to have a unique id, we actually store in the Vertex an altered id depending on the type. The last digit is different: 1 for Cities, 2 for Reservoirs and 3 for Pumping Stations. Wen we want their actual id, we use the method getTypeId( ) that extracts this last digit

```cpp
class Vertex {
  private:
    double dist = 0;
    bool visited = false;
    Edge* path = nullptr;

    std::vector<Edge*> adj;
    std::vector<Edge*> incoming;

    node_type type;
    unsigned int id;
    std::string reservoir = "";
    std::string municipality = "";
    std::string city = "";
    unsigned int population = 0;
```

Edge

- Serves as pipes connecting stations, cities and reservoirs
- The “weight” is the capacity of the pipe
- Also used as pointers

```cpp
class Edge {
  private:
    Vertex* orig = nullptr;
    Vertex* dest = nullptr;
    double weight = 0;
    double flow = 0;

  public:
    Edge(Vertex *orig, Vertex *dest, double w);

    double getFlow() const;
    double getWeight() const;
    Vertex* getOrig() const;
    Vertex* getDest() const;

    void setFlow(double flow);
};
```

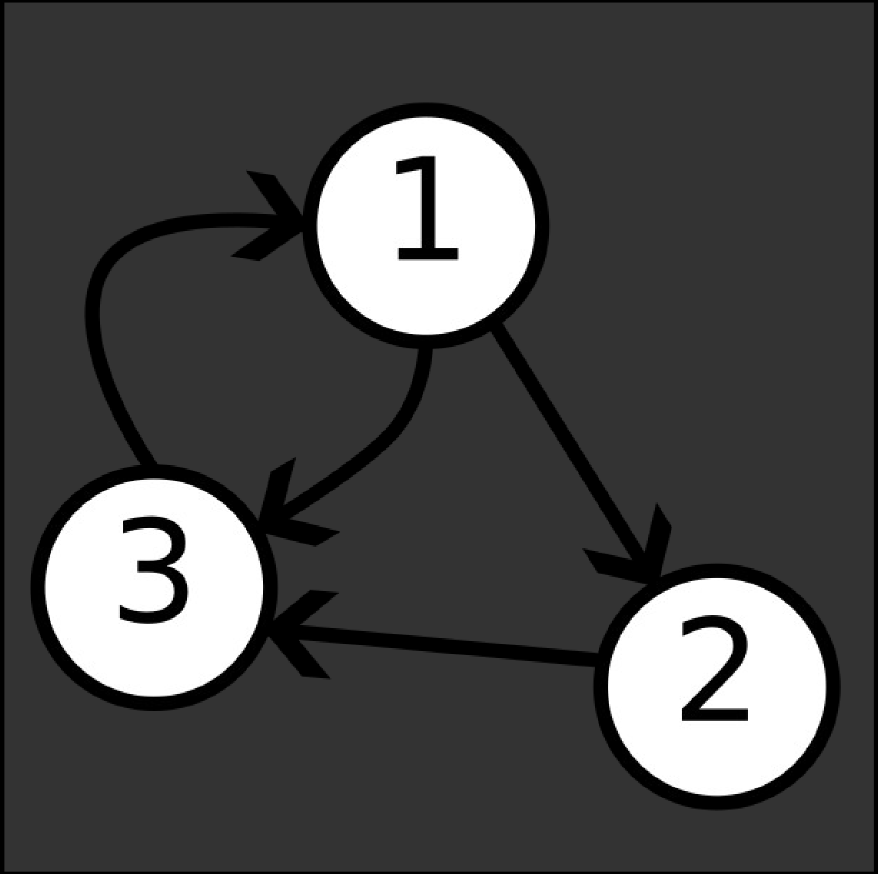
Graph (1/3)

- Simulated water network
- The unordered maps are useful to find certain vertexes
- The methods range from getters/setters to max-flow algorithms

```
class Graph {
  private:
    std::vector<Vertex*> vertexSet;
    Vertex* source = new Vertex(SOURCE, 0, "", "", "", 0);
    Vertex* sink = new Vertex(SINK, 0, "", "", "", 0);
    std::unordered_map<unsigned int, Vertex*> cityVertexes;
    std::unordered_map<unsigned int, Vertex*> reservoirVertexes;
    std::unordered_map<unsigned int, Vertex*> stationVertexes;
```

Graph (2/3)

- There are two special extra vertexes: the source and the sink
- The graph works like this (over-simplified):



Graph (3/3)

- A couple of alterations were made to the graph:
  - The Parser creates the Vertex and addVertex() receives a pointer to the Vertex;
  - findEdge() and removeEdge() now receive pointers to the vertexes instead of their id’s

Tests

- Checks graph integrity and correctness
- Some change the graph, some don't
- Displays useful information about the whole water network

```
//! Checks if the number of edges is correct
void test_edges(Graph& g);

//! Checks if the number of vertexes is correct
void test_vertexes(Graph& g);

//! Checks if all cities are correct
void test_cities(Graph& g);

//! Checks if all reservoirs are correct
void test_reservoirs(Graph& g);

//! Displays information about flow vs. demand
void test_demand(Graph& g);

//! Displays information about flow vs. capacity
void test_capacity(Graph& g);

//! Displays information about flow vs. max delivery
void test_max_delivery(Graph &g);

//! Tests the removal of vertexes and its impact on the graph
void test_remove_vertexes(Graph& g);

//! Tests the removal of edges and its impact on the graph
void test_remove_edges(Graph& g);

//! Test how much the flow changes after a removal
void test_flow_after_remove(Graph g);

//! Calculates some statistics
void test_edge_statistics(Graph& g);

//! Executes all tests
void test_suite(Graph &g);
```

Edmonds-Karp

- Uses BFS to compute paths
- Avoids double search after computing path by storing maximum flow until each Vertex from the source
- Fixes negative flow after completion
- Uses a label and a goto statement to avoid creating a bool to break out of the loop
- Time Complexity: $O(V * E^2)$

```cpp
//! Time Complexity: O(V * E²), Space Complexity: O(1)
void Graph::edmondsKarp() const {
  for (Vertex* v : vertexSet)
    for (Edge* e : v->getAdj())
      e->setFlow(0);

  edmondsKarpLoop: // label avoids creating a bool to break out of the BFS
  for (Vertex* v : this->vertexSet)
    v->setVisited(false);

  std::queue<Vertex*> q;
  q.push(this->source);
  this->source->setDist(INF);
  this->sink->setPath(nullptr);
  while (true) {
    while (!q.empty() && this->sink->getPath() == nullptr) { // BFS
      Vertex *v = q.front();
      q.pop();

      for (Edge* e : v->getAdj()) { // outgoing edges
        Vertex* d = e->getDest();
        if (!d->isVisited() && e->getFlow() < e->getWeight()) {
          d->setPath(e);
          if (d != this->sink) {
            d->setVisited(true);
            q.push(d);
            d->setDist(std::min(v->getDist(), e->getWeight() - e->getFlow()));
          } else {
            Edge* previous = this->sink->getPath();
            double min = std::min(previous->getOrig()->getDist(), previous->getWeight() - previous->getFlow());
            for (Vertex* aux = this->sink; aux != this->source; aux = aux->getPrevious()) {
              Edge* path = aux->getPath();
              path->setFlow(path->getFlow() + (aux == path->getDest() ? min : -min));
            }
            goto edmondsKarpLoop;
          }
        }
      } // outgoing edges

      for (Edge *e : v->getIncoming()) { // incoming edges
        Vertex *d = e->getOrig();
        if (!d->isVisited() && e->getFlow() > 0) {
          d->setVisited(true);
          d->setPath(e);
          q.push(d);
          d->setDist(std::min(v->getDist(), e->getFlow()));
        }
      } // incoming edges
    }
    if (q.empty())
      return;
  }
}
```

Ford–Fulkerson (1/2)

- Uses DFS to compute paths
- Has the same complexity as the Edmonds-Karp algorithm
- Uses an auxiliary recursive function

```cpp
//! Time Complexity: O(V * E²), Space Complexity: O(1)
double Graph::fordFulkerson() {
  for(Vertex *v: this->getVertexSet()) {
    for (Edge *e: v->getAdj())
      e->setFlow(0); // Setting the flow of all the edges as 0.
  }

  double totalFlow = 0;
  bool found = false;
  double flow = 0;
  std::vector<Edge*> toIncreaseFlow;
  std::vector<Edge*> toDecreaseFlow;

  do {
    totalFlow += flow;
    for (Edge *e: toIncreaseFlow) e->setFlow(e->getFlow() + flow); // Update flow (increase)
    for (Edge *e: toDecreaseFlow) e->setFlow(e->getFlow() - flow); // Update flow (decrease)

    toIncreaseFlow = {}; toDecreaseFlow = {}; flow = INF; found = false;  // Resetting values

    for (Vertex *v: this->getVertexSet()) v->setVisited(false); // Set all vertexes as non visited.

    recursiveAugmentingPath(this->source, toIncreaseFlow, toDecreaseFlow, flow, found); // Try to find path
  } while (found);

  return totalFlow;
}
```

Ford-Fulkerson (2/2)

- Auxiliary function for Ford-Fulkerson
- For each Vertex, searchs the outgoing and incoming edges
- The max flow obtained is equal to the Edmond-Karps one
- Time Complexity: $O(V * E^2)$

```cpp
//! Time Complexity: O(E) for one call, O(V * E) for recursion, Space Complexity: O(1)
void Graph::recursiveAugmentingPath(Vertex* v, std::vector<Edge*>& toIncreaseFlow, std::vector<Edge*>& toDecreaseFlow, double& flow, bool& found) {
  v->setVisited(true);
  for (Edge* e: v->getAdj()) {
    Vertex* w = e->getDest();
    if (!w->isVisited() && e->getFlow() < e->getWeight()) {
      if (w == this->sink) { // If the vertex is the target
        found = true;
        flow = e->getWeight() - e->getFlow();
        toIncreaseFlow.push_back(e);
        return;
      }
      recursiveAugmentingPath(w, toIncreaseFlow, toDecreaseFlow, flow, found);
      if (found) { // If the target was found ...
        flow = std::min(flow, e->getWeight() - e->getFlow()); // ... the flow for that path must be updated
        toIncreaseFlow.push_back(e);
        return;
      }
    }
  }

  for (Edge* e: v->getIncoming()) { // The same loop as before, but now, we consider back-edges.
    Vertex* w = e->getOrig();
    if (!w->isVisited() && e->getFlow() > 0) {
      // Assuming that the vertex is not the target.
      recursiveAugmentingPath(w, toIncreaseFlow, toDecreaseFlow, flow, found);
      if (found) { // If the target was found ...
        flow = std::min(flow, e->getFlow()); // ... the flow for that path must be update
        toDecreaseFlow.push_back(e);
        return;
      }
    }
  }
}
```

Reduce Flow

- Reduces the flow from a vertex to another by an amount smaller or equal to the value provided

- Based in Edmonds Karp algorithm, buts decreases the flow instead

- Time Complexity: $O(V * E^2)$ in the worst case

- Although the complexity is the same as in Edmonds Karp algorithm, this one is more efficient, because it doesn't have to go through all the augmenting paths

```cpp
unsigned Graph::reduceFlow(Vertex *src, Vertex *dst, double limit) const {
    unsigned BFSes = 1; // there is always at least one BFS
    reduceFlowLoop:
    while (limit > 0) {
        this->setAllVisitedFalse();
        src->setVisited(true);
        std::queue<Vertex *> q;
        q.push(src);
        while (true) { // BFS
            Vertex *v = q.front();
            q.pop();
            for (Edge *e: v->getAdj()) {
                Vertex *d = e->getDest();
                if (!d->isVisited() && e->getFlow() > 0) {
                    d->setPath(e);
                    if (d != dst) {
                        d->setVisited(true);
                        q.push(d);
                    } else {
                        double min = limit;
                        for (Vertex *aux = dst; aux != src; aux = aux->getPrevious())
                            min = std::min(min, aux->getPath()->getFlow());
                        for (Vertex *aux = dst; aux != src; aux = aux->getPrevious())
                            aux->getPath()->setFlow(aux->getPath()->getFlow() - min);
                        limit -= min;
                        ++BFSes;
                        goto reduceFlowLoop;
                    }
                }
            }
            if (q.empty())
                return BFSes;
        } // BFS
    } // external loop
    return BFSes;
}
```

Removal of Vertex and Removal of Edge

- Before removing a Vertex, the algorithm to reduce the flow (as discussed in the previous slide) is executed from the Super Source to the Vertex to be removed and then again from it to the Super Sink.
- Following the same principle, before removing an Edge, the algorithm to reduce the flow is executed from the Super Source to the origin of the Edge and then again from the destination of the Super Sink
- After removing a Vertex or an Edge, we add them to a vector of removed Vertexes and Edges in the Graph to allow the user to easily restore them

Calculate Dependencies (1/2)

```
void Graph::calculateDependency() {
    for (Vertex *a: this->vertexSet) {
        if (a->getType() == CITY) continue;
        for (Edge *to_remove: a->getAdj()) {
            Graph copy = Graph(*this);
            this->removeEdge(to_remove->getOrig(), to_remove->getDest());
            this->edmondsKarp();
            for (auto c: this->cityVertexes) {
                double difference = copy.getCityVertexes().at(c.second->getTypeId())->getAdj()[0]->getFlow() -
                                    c.second->getAdj()[0]->getFlow();
                if (difference > 0) {
                    c.second->addDependency(to_remove, difference);
                }
            }
            this->restore();
            this->edmondsKarp();
        }
    }
}
```

Calculate Dependencies (2/2)

- This algorithm calculates changes in flow and unfulfilled cities demand after the removal of every actual edge in the graph (one at a time, restoring after calculations)
  - The edges SOURCE - cities and cities - SINK are ignored
- It also marks the edge as essential to the city (in other way, the city dependent on the edge) if the flow through the city decreases after its removal
- Time Complexity: $O(V * E^3)$, Space Complexity: $O(1)$
- This algorithm is executed in the beginning of the program

Flow vs. Capacity

- (extract of output)
- test_capacity()
- Cyan - not used (0 flow)
- Green - 0% < usage ≤ 85%
- Orange - 85% < usage < 100%
- Red - Completely used

```
413 ->  393:   141 / 200
413 ->  403:   116 / 400
423 ->  443:  8250 / 8250
433 ->   61:    80 / 80
433 ->  443:     0 / 200
443 ->   61:    80 / 80
443 ->  433:    80 / 200
443 ->  453:  8090 / 9000
453 ->  161:    26 / 35
453 ->   61:    70 / 80
453 ->  443:     0 / 9000
453 ->  513:  7994 / 9000
463 ->  181:    55 / 55
473 ->  181:    55 / 55
483 ->  181:    50 / 55
```

Flow vs. Demand

- Obtained with test_demand()
- The white numbers are the IDs of the cities
  - When parsing, the actual IDs are multiplied by 10 and added 1
- Some cities had their demand fulfilled, while other didn't

```
test_demand
Water Received vs. Demand
 221:   330 /   397, missing 67
 211:   130 /   161, missing 31
 201:   100 /   168, missing 68
 191:   780 /   780
 181:   200 /   200
 171:  5650 /  6324, missing 674
 161:    96 /    96
 151: 12250 / 12250
 141:   406 /   406
  11:    52 /    52
  21:   515 /   515
  31:   110 /   160, missing 50
  41:  1208 /  1208
  51:   130 /   152, missing 22
  61:   230 /   230
  71:   896 /   896
  81:   100 /   122, missing 22
  91:    53 /    53
 101:   220 /   313, missing 93
 111:   407 /   407
 121:   177 /   177
 131:   123 /   158, missing 35
```

Flow vs. Max Delivery

- Obtained with test_max_delivery()
- Green - 0% <= usage <= 85%
- Orange - 85% <= usage <= 100%
- All of them are used
- Some of them have really low usages (like 182 - 0.8%)

```
test_max_delivery
Flow vs. Max Delivery
  12:  1715 / 2750
  22:  1100 / 2080
  32:    40 / 40
  42:   250 / 250
  52:   665 / 2000
  62:  4558 / 8500
  72:   500 / 500
  82:   100 / 500
  92:   177 / 242
 102:   205 / 280
 112:   100 / 100
 122:    30 / 30
 132:  8562 / 10000
 142:   110 / 110
 152:   288 / 430
 162:  3000 / 3000
 172:  2063 / 3000
 182:    20 / 2500
 192:    40 / 56
 202:    35 / 35
 212:   208 / 208
 222:   210 / 267
 232:   132 / 260
 242:    55 / 64
Total Flow: 24163
```

Statistics

- Obtained with command pipes_statistics
- As the edges/pipes have very different capacities, the ratio was used instead
- At least 25% of them are being fully used

```
> pipes_statistics

Statistics (flow / capacity)
   Number of pipes: 208
              Mean: 0.548448
          Variance: 0.180801
Standard Deviation: 0.425207
            Median: 0.625
                Q1: 0.0841667
                Q3: 1
               AQI: 0.915833
 Maximum Amplitude: 1
       Empty Pipes: 36
        Full Pipes: 63
```

User Interface (1/3)

```
Welcome to our project!
Type help to learn the available commands.
> help

List of available commands:

    exit:              takes no arguments
        Terminates the program.

    help:              takes no arguments
        Prints this list of commands.

    display_city:      takes 1 argument    display_city <city_id | -all>
        Displays the City with that id, or all of them in case the flag -all is used.

    display_reservoir: takes 1 argument    display_reservoir <reservoir_id | -all>
        Displays the Reservoir with that id, or all of them in case the flag -all is used.

    display_station:   takes 1 argument    display_station <station_id | -all>
        Displays the Pumping Station with that id, or all of them in case the flag -all is used.

    needy_cities:      takes no arguments
        Displays the cities whose water demand is not fulfilled by the network.
```

User Interface (2/3)

```
remove:             takes 1 argument    remove <code>
    Removes a City, Reservoir, or a Pumping Station from the network and prints the Cities/Reservoirs affected.
    Do not forget to use the restore command to add the removed location again to the network.

remove_pipe:        takes 2 arguments   remove_pipe <code_origin> <code_destination>
    Removes a Pipe from the network and prints the Cities/Reservoirs affected.
    Do not forget to use the restore command to add the removed location again to the network.

restore:            takes no arguments
    Undoes the previous removal. The network will be as in the beginning.

show_dependency:    takes 1 argument    show_dependency <city_id>
    For a given City, shows which Pipelines, if ruptured, would affect the amount of water reaching the City.

pipes_statistics:   takes no arguments
    Displays statistics about the flow/capacity ratio of the pipes, like their average ratio, amount of empty/full pipes, etc.

>
```

User Interface (3/3)

```
> pipes_statistics

Statistics (flow / capacity)
    Number of pipes: 208
               Mean: 0.548448
           Variance: 0.180801
 Standard Deviation: 0.425207
             Median: 0.625
                 Q1: 0.0841667
                 Q3: 1
                AQI: 0.915833
  Maximum Amplitude: 1
        Empty Pipes: 36
         Full Pipes: 77
```

```
> display_reservoir 1

  1: Ermida
    Max delivery: 2750
    Water supplying: 1385
    Municipality: Aveiro
```

```
> display_city 17

 17: Porto
    Demand: 6324
    Water reaching: 5582
    Population: 948613
```

Problems

1. After Edmonds-Karp, some pipes were left with negative flow and some were overused
2. How to Balance Edges

Solutions (1/2)

1. This was rather quickly fixed - it was due to an external loop and return condition deemed unnecessary, which would not alter the max-flow but would fix negative flows and overusage

Solutions (2/2)

2.

```
Lets say that the ratio of an edge is its flow divided by its capacity.

function balancePipes(G){
    while true{
        minAP = getMinRatioAP(G);
        maxAP = getMaxRatioAP(G);
        if minAP is the same as previous minAP and maxAP is the same as previous maxAP: break;
        F = ((ratio(maxAP) - ratio(minAP)) / 2) * capacity(maxAP);
        transfer flow F from maxAP to minAP as much as possible;
    }
}

/* getMinRatioAP(G) is an algorithm based on DFS to find an AP (Augmenting Path). At each step of the
recursion, it chooses the lowest ratio edge. getMaxRatioAP(G) does the same, but chooses the highest
ratio edge at each step. */
/* ratio(path) is the ratio of the lowest ratio edge in that path. */
/* capacity(path) is the capacity of the lowest capacity edge in that path. */
```

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