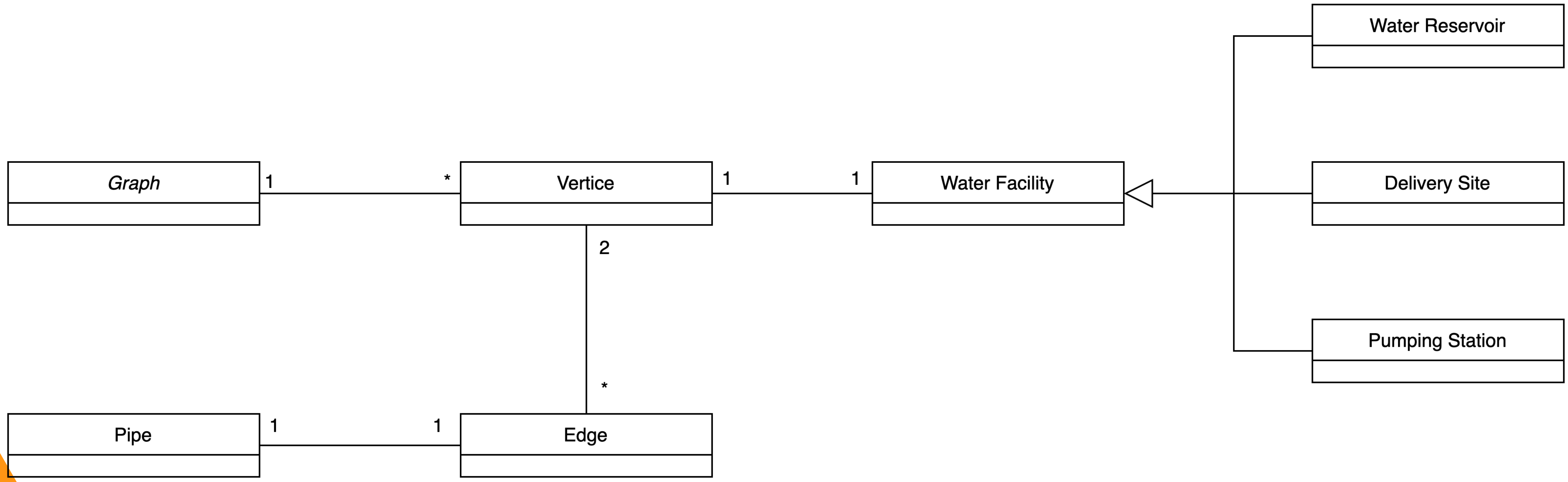
Water Management System

Desenho de Algoritmos

Gonçalo Marques up202206205
Miguel Duarte up202206102
Tómas Teixeira up202208041

Index

Class Diagram



Reading the Dataset

Reservoirs

Reservoir	Municipality	Id	Code	Maximum Delivery (m3/sec)

Delivery Site

City	Id	Code	Demand	Population

Edges

Service_Point_A	Service_Point_B	Capacity	Direction

Pumping Stations

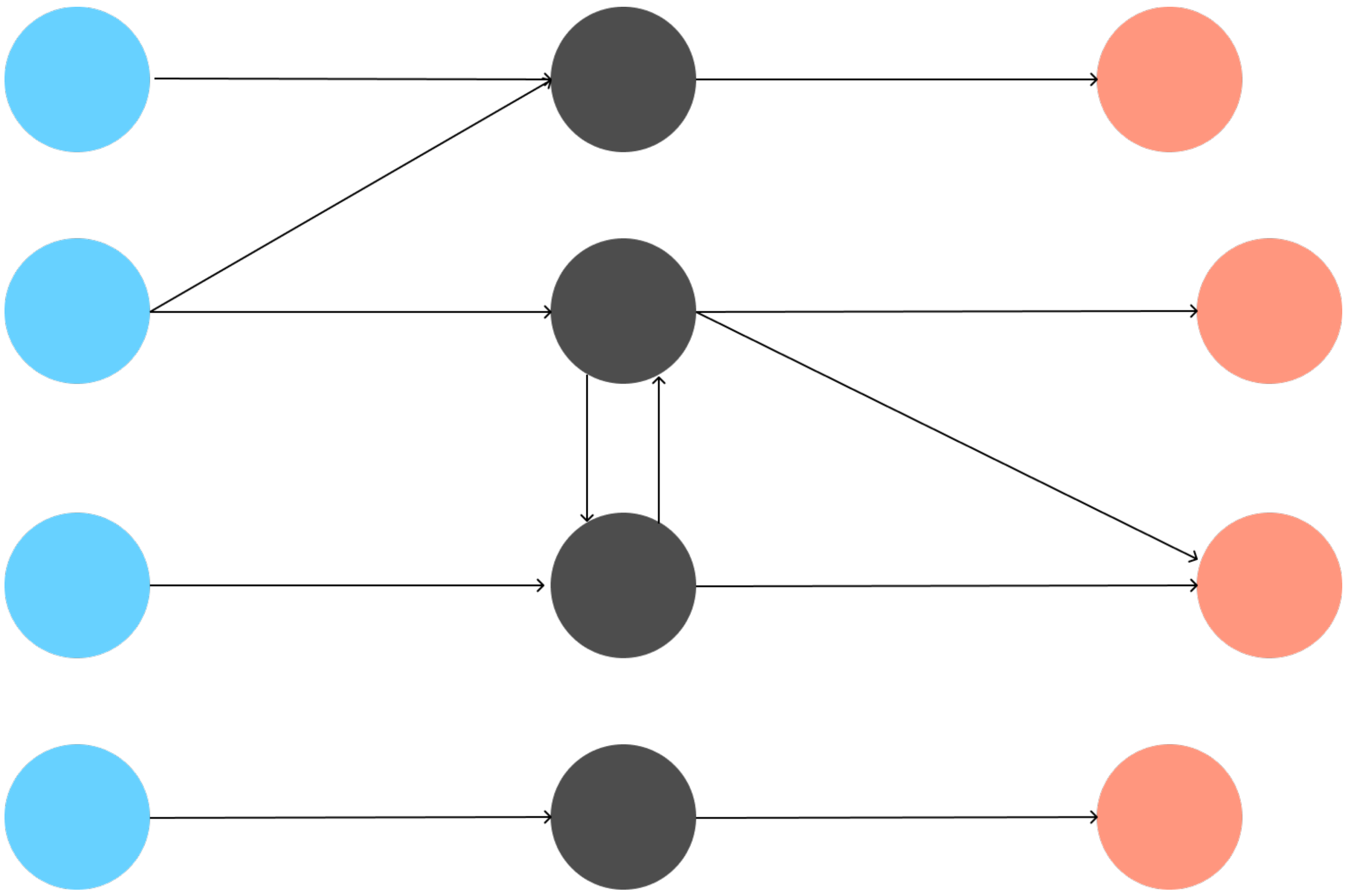
Id	Code

The graph

Water Reservoir

Pumping Station

Delivery Site



Max Flow

```cpp
void Graph::getMaxFlow() {
    Graph* aux = getMaxFlowAuxGraph();
    auto* src : Vertex *  = aux->getVertex( code: "source");
    auto* sink : Vertex *  = aux->getVertex( code: "sink");
    while (bfsEdmondsKarp( g: aux,  source: src, sink)) {
        int path_flow = INT_MAX;

        // Find minimum residual capacity along the path
        for (Vertex* v = sink; v != src; v = v->getPath()->getSource()) {
            Edge* e = v->getPath();
            Pipe* pipe = e->getPipe();
            path_flow = std::min(path_flow, pipe->getCapacity() - pipe->getFlow());
        }

        // Update flows along the path and reverse flows
        for (Vertex* v = sink; v != src; v = v->getPath()->getSource()) {
            Edge* edge = v->getPath();
            int flow = edge->getPipe()->getFlow();
            edge->getPipe()->setFlow(flow + path_flow);

            Edge* reverseEdge = edge->getReverse();
            if (reverseEdge != nullptr) {
                reverseEdge->getPipe()->setFlow(reverseEdge->getPipe()->getFlow() - path_flow);
            }
            else {
                Edge* e = new Edge( src: v,  dest: edge->getSource(),  info: new Pipe( capacity: 0));
                e->setReverse(edge);
                edge->setReverse( edge: e);
                aux->addEdge( src: v,  dest: edge->getSource(),  edge: e);
                e->getPipe()->setFlow(-path_flow);
            }
        }
    }
}
```

Calculate maximum amount of water that can reach each or a specific city, i.e., the max flow.

- Edmonds-Karp algorithm

Deficit calculation

```cpp
std::vector<std::pair<DeliverySite, int>> RequestManager::sufficientDelivery() {
    container.getGraph()->getMaxFlow();
    int difference;
    std::vector<std::pair<DeliverySite, int>> differences;
    for(Vertex* v: container.getGraph()->getCities()) {
        DeliverySite info = *dynamic_cast<DeliverySite*>(v->getInfo());
        int flow = 0;
        for (Edge* e: v->getIncomingEdges()) {
            flow += e->getPipe()->getFlow();
        }
        difference = flow - info.getDemand();
        if (difference < 0) {
            differences.emplace_back( t1: info,  t2: difference);
        }
    }
    return differences;
}
```

Check if the demand of each city is being satisfied.

- After running the max-flow algorithm, check if the flow from the incoming edges of each city is enough to satisfy its demand.

Balance flow

```cpp
Graph* Graph::balanceFlow() {
    Graph* aux = getCopy();
    aux->getMaxFlow();
    std::vector<std::unordered_set<Vertex*>> sccs = aux->getSCC();
    double average = getFlowToCapacityRatioAverage();
    for(auto scc : sccs) {
        std::vector<Edge*> sccEdges;


        for(const Vertex* v : scc) {
            for(Edge* e : v->getAdj()) {
                if(scc.find(e->getDest()) != scc.end()) {
                    sccEdges.emplace_back(e);
                }
            }
        }
        int start = 0;
        int end = (int)sccEdges.size() - 1;
        while (start <= end) {
            bool updateLeft = false;
            bool updateRight = false;
            std::sort(sccEdges.begin(), sccEdges.end(), [](Edge* a, Edge* b) { //sort in flow a
                return a->getPipe()->getFlow() < b->getPipe()->getFlow();
            });
            Edge* fromStart = sccEdges[start];
            Edge* fromEnd = sccEdges[end];
            while(true) {

                if(abs(fromEnd->getPipe()->getRatio() - fromStart->getPipe()->getRatio()) < 0.02) {
                    updateLeft = true;
                    updateRight = true;
                    break;
                }

                if(fromEnd->getPipe()->getRatio() <= average) {
                    updateRight = true;
                    break;
                }
                if(fromStart->getPipe()->getRatio() >= average) {
                    updateLeft = true;
                    break;
                }
```

Balance the differences of flow to capacity across the network.

- The first issue here is to make sure we have a connection between two edges to balance them.
- To tackle this, we calculated the strongly connected components of the graph.
- Each pipe has a connection to other pipe if they belong to the same scc.
- By using a greedy approach in each scc we were able to minimize the differences of the flow to capacity ratio.

Water Reservoir removal

Check if a reservoir is removed, if any delivery site is affected.

- We can simply follow the flow leaving the desired delivery site and remove it from the upcoming edges.
- However, there is an issue, as we don't know where a specific flow comes from.
- In a real-life scenario, we probably would know it, making this idea doable and avoiding running the max-flow algorithm again.

Pumping Station removal

```
std::unordered_map<DeliverySite*, int> RequestManager::getAffectedCities(const std::string& code) {
    std::unordered_map<DeliverySite*, int> res;
    Graph* aux = container.getGraph()->removeVertex(code);
    if(aux == nullptr) return res;
    for(Vertex* originalVertex : container.getGraph()->getCities()) {
        Vertex* vertex = aux->getVertex( code: originalVertex->getInfo()->getCode());
        int flow = 0;
        int originalFlow = 0;
        for(Edge* e : originalVertex->getIncomingEdges()) {
            originalFlow += e->getPipe()->getFlow();
        }
        for(Edge* e : vertex->getIncomingEdges()) {
            flow += e->getPipe()->getFlow();
        }
        auto* info = dynamic_cast<DeliverySite*>(originalVertex->getInfo());
        if(flow < originalFlow) res.insert( x: std::make_pair( &: info,  t2: info->getDemand() - flow));
    }
    return res;
}
```

Check if a pumping station is removed, if any delivery site is affected.

- We removed the pumping station from an auxiliary graph and then we run the max-flow algorithm.
- We then compared each vertex incoming flow with his original incoming flow.

Pipe Removal

```
std::unordered_map<DeliverySite*, int> RequestManager::getAffectedCities(const std::string& source, const std::string& dest) {
    std::unordered_map<DeliverySite*, int> res;
    Graph* aux = container.getGraph()->removePipe(source, dest);
    if(aux == nullptr) return res;
    for(Vertex* originalVertex : container.getGraph()->getCities()) {
        Vertex* vertex = aux->getVertex( code: originalVertex->getInfo()->getCode());
        int flow = 0;
        int originalFlow = 0;
        for(Edge* e : originalVertex->getIncomingEdges()) {
            originalFlow += e->getPipe()->getFlow();
        }
        for(Edge* e : vertex->getIncomingEdges()) {
            flow += e->getPipe()->getFlow();
        }
        auto* info = dynamic_cast<DeliverySite*>(originalVertex->getInfo());
        if(flow < originalFlow) res.insert( x: std::make_pair( &: info,  t2: info->getDemand() - flow));
    }
    return res;
}
```

Check if a pipe is removed, if any delivery site is affected.

- We removed the edge from an auxiliary graph and then we run the max-flow algorithm.
- We then compared each vertex incoming flow with his original incoming flow.

```
Water-Supply-Management-System
===============[ MENU ]===============
1. Max amount of water
2. Can all the water reservoirs supply enough water to all its delivery sites?
3. Balance water distribution
4. Structures out of service

Choose an option:
```

User interface

Highlights

- Our main highlight is the idea behind the balance function, i.e., by finding scc's we were able to use a greedy approach to reduce variance.

Individual contribution

- Everyone worked equally and in many different tasks. We didn't assigned tasks to anyone, so everyone did a little bit on every question.