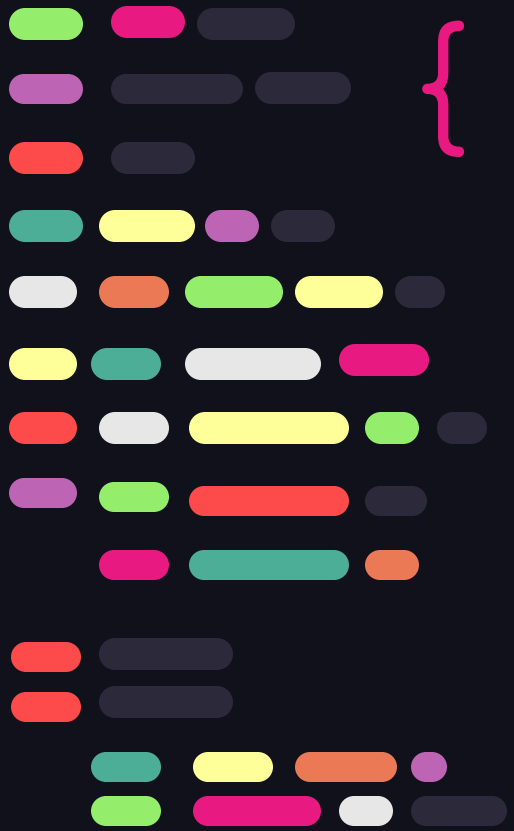
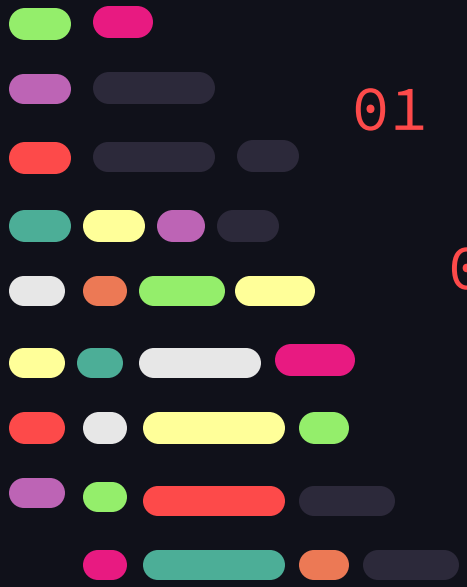


Desenho de Algoritmos

L.EIC Delivery Management

Work by: Joana Pimenta up202206120; Dinis Galvão up202207217 ; Miguel Sousa up202207986
Group G07_6

Topics of the presentation



01 Classes

02 Parsing

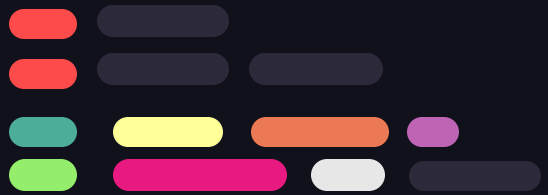
03 Functionalities and Algorithms

04 Interface & Features

01 { ..

Classes

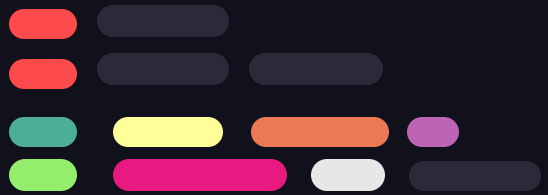
< Brief introduction to the classes of our project >



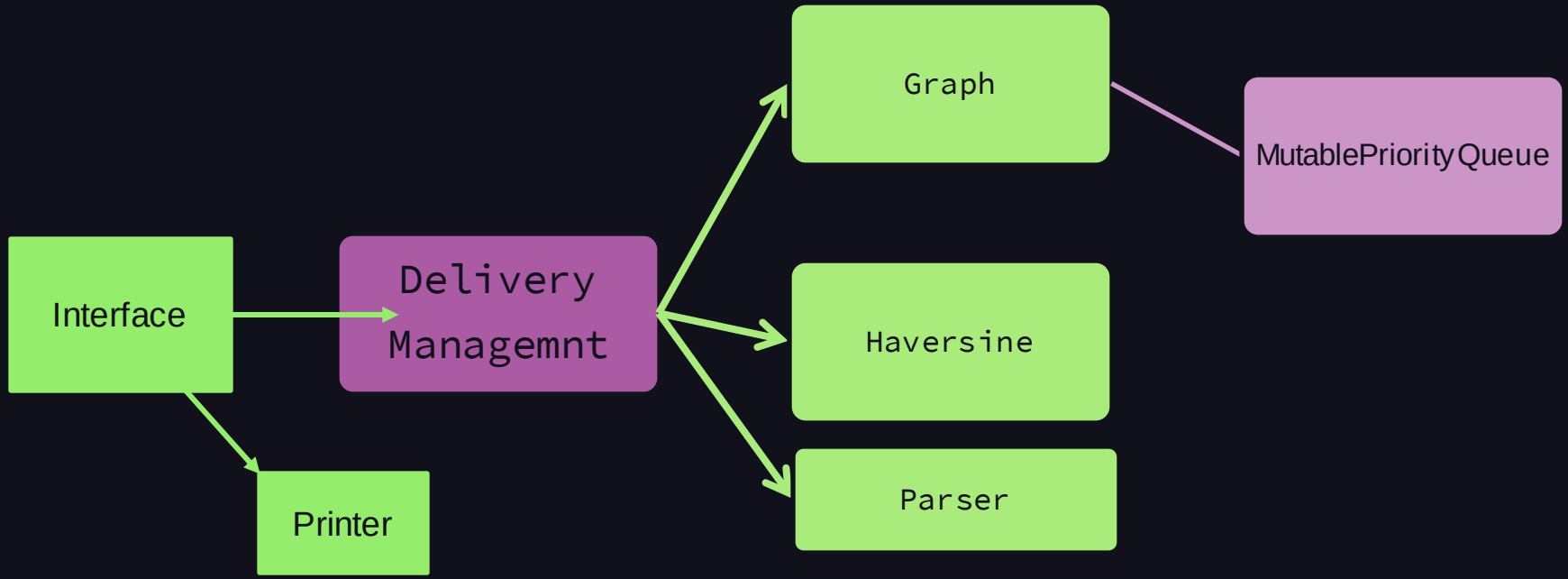
} ..

All our classes

- Parser
- DeliveryManagement
- MutablePriorityQueue
- Graph
- Haversine
- Printer
- Interface



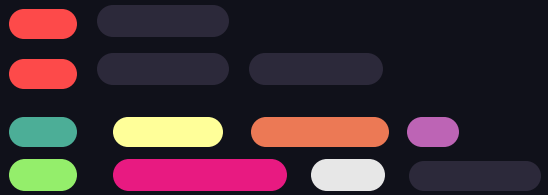
Classes Diagram



02 {..

Dataset reading

< Our Parser >



} ..

```cpp
#include "Parser.h"
Parser::Parser(const std::string &fName) {
    file_.open(fName);
    if (!file_) {
        throw std::runtime_error("Could not open file: " + fName);
    }
}

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(start, end - start);
}

std::vector<std::vector<std::string>> Parser::getData() {
    std::string line;
    while (std::getline(file_, line)) {
        if(std::isalpha(line[0])) {
            std::getline(file_, line);
        }
        std::istringstream iss(line);
        std::string value;
        std::vector<std::string> v;
        while (std::getline(iss, value, ',')) {
            value = trim(value);
            v.push_back(value);


        }

        data_.push_back(v);
    }
    return data_;
}
```

Parser

This class is used to parse the files.

In this class we have four functions:

- Parser
- trim
- getData

Delivery Manager Constructor

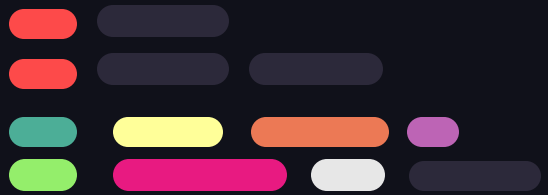
DeliveryManager::DeliveryManager(std::string vertex_file, std::string edge_file)

This constructor dinamicaly parses the wanted csv file to the graph, in order to be used later on the problems.

```
*/
DeliveryManager::DeliveryManager(std::string vertex_file, std::string edge_file):deliveryGraph_(std::make_unique<Graph<int>>())
,vertex_(std::unordered_map<int, Vertex<int>*,vert_struct>())
{
    if(edge_file=="") {
        Parser edges( fName: vertex_file);

        for (std::vector<std::string> line : edges.getData()){
            if(vertex_.find( x: std::stoi( str: line.at( n: 0)))==nullptr){
                vertex_.insert( x: { x: std::stoi( str: line.at( n: 0)), y: new Vertex<int>( in: std::stoi( str: line.at( n: 0)))});
                deliveryGraph_->vertexSet.push_back(vertex_[std::stoi( str: line.at( n: 0))]);

            }
            if(vertex_.find( x: std::stoi( str: line.at( n: 1)))==nullptr){
                vertex_.insert( x: { x: std::stoi( str: line.at( n: 1)), y: new Vertex<int>( in: std::stoi( str: line.at( n: 1)))});
                deliveryGraph_->vertexSet.push_back(vertex_[std::stoi( str: line.at( n: 1))]);

            }
            int orig =std::stoi( str: line.at( n: 0));
            int dest =std::stoi( str: line.at( n: 1));
            double distance= std::stod( str: line.at( n: 2));

            vertex_[orig]->addEdge( d: vertex_[dest], w: distance);
            vertex_[dest]->addEdge( d: vertex_[orig], w: distance);
        }

    }
    else {
        Parser nodes( fName: vertex_file);
        Parser edges( fName: edge_file);


        for (std::vector<std::string> line : nodes.getData()){
            vertex_.insert( x: { x: std::stoi( str: line.at( n: 0)), y: new Vertex<int>( in: std::stoi( str: line.at( n: 0)))});
            deliveryGraph_->vertexSet.push_back(vertex_[std::stoi( str: line.at( n: 0))]);
            vertex_[std::stoi( str: line.at( n: 0))]->setLongitude( l: std::stod( str: line.at( n: 1)));
            vertex_[std::stoi( str: line.at( n: 0))]->setLatitude( l: std::stod( str: line.at( n: 2)));
        }

        for (std::vector<std::string> line : edges.getData()){
            int orig =std::stoi( str: line.at( n: 0));
            int dest =std::stoi( str: line.at( n: 1));

            double distance= std::stod( str: line.at( n: 2));

            vertex_[orig]->addEdge( d: vertex_[dest], w: distance);
            vertex_[dest]->addEdge( d: vertex_[orig], w: distance);
        }
    }
}
```

03 { .. Funtionalities and Algorithms

< functionalities and algoritms used>



} . .

Backtracking

We use to functions two backtrac:

- backtrack_tsp(it's called recursively)
- backtracking(funstion with the final result)

```
void DeliveryManager::backtrack_tsp(std::unique_ptr<Graph<int>>& g,int vis, Vertex<int>* v,double& res,double cost) {
    if(vis==g->getVertexSet().size()) {
        for(Edge<int>* e: v->getAdj()) {
            if(e->getDest()->getInfo()==0) {
                res=std::min(res,e->getWeight()+cost);
            }
        }
        return;

    }
    for(Edge<int>* e: v->getAdj()) {
        Vertex<int>* dest=e->getDest();
        double newcost=cost+e->getWeight();
        if(!dest->isVisited()) {
            if(newcost<res) {
                dest->setVisited(true);
                vis++;
                backtrack_tsp( &: g,vis, v: dest, &: res, cost: newcost);
                dest->setVisited(false);
                vis--;
            }
        }
    }
}
```

```
std::pair<double, double> DeliveryManager::backtracking(std::unique_ptr<Graph<int>>& g) {
    auto start : time_point<...> =std::chrono::high_resolution_clock::now();
    std::ios_base::sync_with_stdio( sync: false);
    double res=INT_MAX;
    for(Vertex<int> * v : g->getVertexSet()) {
        v->setVisited(false);
    }
    g->findVertex( in: 0)->setVisited(true);
    backtrack_tsp( &: g, vis: 1, v: g->findVertex( in: 0), &: res, cost: 0);
    auto end : time_point<...> =std::chrono::high_resolution_clock::now();
    double res_t=std::chrono::duration_cast<std::chrono::nanoseconds>( d: end-start).count();
    res_t*=1e-9;
    return  std::make_pair( &: res, &: res_t);

}
```

Triangular Approximation

```
void DeliveryManager::dfsPrim(std::unique_ptr<Graph<int>>& g,int v,std::vector<int>& res,std::vector<int> prim) {
    if(!g->findVertex( in: v)->isVisited()) {
        g->findVertex( in: v)->setVisited(true);
        bool flag=true;
        res.push_back(v);
        for(int i=0;i<prim.size();i++) {
            if(prim[i]==v && !g->findVertex( in: i)->isVisited()) {
                dfsPrim( &: g, v: i, &: res,prim);
                flag=false;
            }
        }


    }
}
/++
std::vector<int> DeliveryManager::mstPrim(std::unique_ptr<Graph<int>>& g,int start) {
    std::vector<int> pre( n: g->getVertexSet().size(),  value: -1);
    std::vector<double> key( n: g->getVertexSet().size(),  value: INT_MAX);
    std::priority_queue<std::pair<double, Vertex<int>*>, std::vector<std::pair<double, Vertex<int>*>>, std::greater<std::pair<double, Vertex<int>*>>> V;

    for(Vertex<int>* v :g->getVertexSet()) {
        if(v->getInfo()==start) {
            V.emplace( a: start, b: v);
            key[start]=0;
            v->setVisited(true);
        }
        else {
            V.emplace( a: INT_MAX, b: v);
            v->setVisited(false);
        }

    }

    while(!V.empty()) {
        std::pair<double,Vertex<int>*> tp=V.top();
        V.pop();
        Vertex<int>* u=tp.second;
        u->setVisited(true);

        for(Edge<int>* e:u->getAdj()) {
            Vertex<int>* dest=e->getDest();
            double w=e->getWeight();
            if(!dest->isVisited()&& w<key[dest->getInfo()]) {
                pre[dest->getInfo()]=u->getInfo();
                key[dest->getInfo()]=w;
                updateQueue( &: V,w, v: dest);
            }
        }
    }
    return pre;
}
```

```
void DeliveryManager::updateQueue( std::priority_queue<std::pair<double, Vertex<int>*>, std::vector<std::pair<double, Vertex<int>*>>, std::greater<std::pair<double, Vertex<int>*>>>& V,double w,Vertex<int>* v)
    std::vector<std::pair<double,Vertex<int>*>> tmp;
    while(!V.empty()) {
        std::pair<double,Vertex<int>*> tp=V.top();
        if(tp.second==v) {
            tp.first=w;
        }
        V.pop();
        tmp.push_back(tp);
    }


    for(std::pair<double,Vertex<int>*>& t:tmp) {
        V.push( x: t);
    }
}
```

```
std::pair<double,double> DeliveryManager::tsp2Approximation(std::unique_ptr<Graph<int>>& g) {
        double cost=0;
        auto start : time_point<...> =std::chrono::high_resolution_clock::now();
        std::ios_base::sync_with_stdio( sync: false);
        std::vector<int> res;
        std::vector<std::vector<int>> ans;
        std::vector<int> pri =mstPrim( &: g, start: 0);
        for(Vertex<int>* v:g->getVertexSet()) {
            v->setVisited(false);
        }

        for(int i=0;i<pri.size();i++) {
            dfsPrim( &: g, v: i, &: res, prim: pri);
        }
        res.push_back(0);
        int prev=res[0];
        int aft;
        for(int i=1 ;i!=res.size();i++){
            aft=res[i];
            if(findEdge( &: g,  in: prev, dest: aft)!=nullptr) {
                double w=findEdge( &: g, in: prev, dest: aft)->getWeight();
                cost+=w;
                prev=aft;
            }
            else {
                double w=calculate_distance( latitude1: g->findVertex( in: prev)->getLatitude(), longitude1: g->findVertex( in: prev)->ge
                g->addBidirectionalEdge( sourc: prev, dest: aft,w);
                cost+=w;
                prev=aft;
            }
        }
    auto end : time_point<...> =std::chrono::high_resolution_clock::now();
    double res_t=std::chrono::duration_cast<std::chrono::nanoseconds>( d: end-start).count();
    res_t*=1e-9;
    return  std::make_pair( &: cost, &: res_t);


}
```

Other heuristic and real World

```
std::vector<Vertex<int>*> DeliveryManager::findOddDegreeVertices(std::unique_ptr<Graph<int>>& g, const std::vector<int>& mst) {
    std::unordered_map<int, int> degreeCount;
    for(int i = 0; i < mst.size(); ++i) {
        if(mst[i] != -1) {
            degreeCount[i]++;
            degreeCount[mst[i]]++;
        }
    }
    std::vector<Vertex<int>*> oddVertices;
    for(auto& pair : pair<const int, int> &  : degreeCount) {
        if(pair.second % 2 == 1) {
            oddVertices.push_back(g->findVertex( in: pair.first));
        }
    }
    return oddVertices;
}
```

```
std::pair<double,double> DeliveryManager::heuristicTSP(std::unique_ptr<Graph<int>>& g) {
    auto start : time_point<...> =std::chrono::high_resolution_clock::now();
    std::ios_base::sync_with_stdio( sync: false);
    std::vector<int> mstPredecessors = mstPrim( &: g, start: 0);

    std::vector<Edge<int>*> mstEdges;
    for(int i=0;i<mstPredecessors.size();i++) {
        if(mstPredecessors[i]!=-1) {
            for(Edge<int>* e:deliveryGraph_->findVertex( in: mstPredecessors[i])->getAdj()) {
                if(e->getDest()->getInfo()==i) {
                    mstEdges.push_back(e);
                }
            }
        }
    }
    std::vector<Vertex<int>*> oddVertices = findOddDegreeVertices( &: g,  mst: mstPredecessors);
    std::vector<std::pair<Vertex<int>*, Edge<int>*>> mwpmVertices = minimumWeightPerfectMatching( &: oddVertices);
    std::vector<Edge<int>*> mwpmEdges;
    for(auto e : pair<...> :mwpmVertices) {
        mstEdges.push_back(e.second);
    }

    std::vector<Edge<int>*> eulerianEdges = combineMSTandMWPM(mstEdges, mwpmEdges);
    std::vector<int> eulerianCircuit1 = eulerianCircuit( &: g, eulerianEdges, start: 0);
    double cost=calculateTSPPath( eulerianCircuit: eulerianCircuit1,  &: g,eulerianEdges);
    auto end : time_point<...> =std::chrono::high_resolution_clock::now();
    double res_t=std::chrono::duration_cast<std::chrono::nanoseconds>( d: end-start).count();
    res_t*=1e-9;

    return  std::make_pair( &: cost, &: res_t);
}
```

```
td::vector<std::pair<Vertex<int>*, Edge<int>*>> DeliveryManager::minimumWeightPerfectMatching(std::vector<Vertex<int>*>& oddVertices) {
    std::vector<std::pair<Vertex<int>*, Edge<int>*>> matching;
    for(int i = 0; i < oddVertices.size(); ++i) {
        double minDistance = INT_MAX;
        Edge<int>* edge = nullptr;
        Edge<int>* closestEdge = nullptr;

        for(int j = 0; j < oddVertices.size(); ++j) {
            if(i != j) {
                Vertex<int>* orig=oddVertices[i];
                Vertex<int>* dest=oddVertices[j];
                double distance=INT_MAX;
                bool flag=true;
                for(auto e : Edge<int> * :orig->getAdj()) {
                    if(e->getDest()==dest) {
                        distance=e->getWeight();
                        edge=e;
                        flag=false;
                    }

                }
                if(flag) {
                    distance+=calculate_distance( latitude1: deliveryGraph_->findVertex( in: orig->getInfo())->getLatitude(), longitude1: deliveryGrap

                }

                if(distance < minDistance) {
                    minDistance = distance;
                    closestEdge = edge;
                }
            }
        }
        matching.push_back({ &: oddVertices[i],  &: closestEdge});
    }
    return matching;
```

Other heuristic and real World

```cpp
std::vector<Edge<int>*> DeliveryManager::combineMSTandMWPM(const std::vector<Edge<int>*>& mstEdges, const std::vector<Edge<int>*>& mwpmEdges) {
    std::vector<Edge<int>*> eulerianEdges;
    for(auto& edge : Edge<int> *const &  : mstEdges) {
        eulerianEdges.push_back(edge);
    }
    for(auto& edge : Edge<int> *const &  : mwpmEdges) {
        eulerianEdges.push_back(edge);
    }
    return eulerianEdges;
}
```

```cpp
std::pair<double,double> DeliveryManager::heuristicTSP(std::unique_ptr<Graph<int>>& g) {
    auto start : time_point<...> =std::chrono::high_resolution_clock::now();
    std::ios_base::sync_with_stdio( sync: false);
    std::vector<int> mstPredecessors = mstPrim( &: g, start: 0);

    std::vector<Edge<int>*> mstEdges;
    for(int i=0;i<mstPredecessors.size();i++) {
        if(mstPredecessors[i]!=-1) {
            for(Edge<int>* e:deliveryGraph_->findVertex( in: mstPredecessors[i])->getAdj()) {
                if(e->getDest()->getInfo()==i) {
                    mstEdges.push_back(e);
                }
            }
        }
    }
    std::vector<Vertex<int>*> oddVertices = findOddDegreeVertices( &: g,  mst: mstPredecessors);
    std::vector<std::pair<Vertex<int>*, Edge<int>*>> mwpmVertices = minimumWeightPerfectMatching( &: oddVertices);
    std::vector<Edge<int>*> mwpmEdges;
    for(auto e : pair<...> :mwpmVertices) {
        mstEdges.push_back(e.second);
    }

    std::vector<Edge<int>*> eulerianEdges = combineMSTandMWPM(mstEdges, mwpmEdges);
    std::vector<int> eulerianCircuit1 = eulerianCircuit( &: g, eulerianEdges, start: 0);
    double cost=calculateTSPPath( eulerianCircuit: eulerianCircuit1,  &: g,eulerianEdges);
    auto end : time_point<...> =std::chrono::high_resolution_clock::now();
    double res_t=std::chrono::duration_cast<std::chrono::nanoseconds>( d: end-start).count();
    res_t*=1e-9;

    return  std::make_pair( &: cost, &: res_t);
}
```

```cpp
std::vector<int> DeliveryManager::eulerianCircuit(std::unique_ptr<Graph<int>>& g, const std::vector<Edge<int>*>& eulerianEdges,int start) {
    std::unordered_map<int, std::vector<int>> adjList;
    for(auto& edge : Edge<int> *const &  : eulerianEdges) {
        adjList[edge->getOrig()->getInfo()].push_back(edge->getDest()->getInfo());
    }
    std::vector<int> circuit;
    std::stack<int> stack;
    int currentVertex = start;
    stack.push( x: currentVertex);
    while(!stack.empty()) {
        if(adjList[currentVertex].empty()) {
            circuit.push_back(currentVertex);
            currentVertex = stack.top();
            stack.pop();
        } else {
            stack.push( x: currentVertex);
            int nextVertex = adjList[currentVertex].back();
            adjList[currentVertex].pop_back();
            currentVertex = nextVertex;
        }
    }
    return circuit;
}
```

Other heuristic and real World

```
double DeliveryManager::calculateTSPPath(const std::vector<int>& eulerianCircuit, std::unique_ptr<Graph<int>>& g,const std::vector<Edge<int>*>& eulerianEdges) {
    std::vector<bool> visited( n: eulerianCircuit.size(), value: false);
    double cost = 0.0;
    int prev = eulerianCircuit[0];
    visited[prev] = true;
    for(auto e : Edge<int> * :deliveryGraph_->findVertex( in: prev)->getAdj()) {
        if(e->getDest()->getInfo()==0) {
            cost+=e->getWeight();
            break;
        }
    }

    for(size_t i = 1; i < eulerianCircuit.size(); ++i) {
        int current = eulerianCircuit[i];
        if(!visited[current]) {
            visited[current] = true;
            bool flag=true;
            for(auto e : Edge<int> * :eulerianEdges) {
                if(e->getOrig()->getInfo()==prev) {
                    cost+=e->getWeight();
                    flag=false;
                    break;
                }
            }
            if(flag) {
                cost+=calculate_distance( latitude1: deliveryGraph_->findVertex( in: prev)->getLatitude(), longitude1: deliveryGraph_->findVertex( in: prev)->getLongitude(), latitude2: deliveryGraph_->findVertex( in: current)->

            }
            prev = current;
        }
    }
    return cost;
}
```

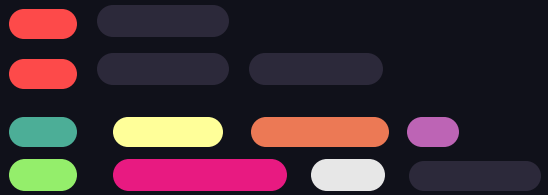
Real World

```
std::pair<double,double> DeliveryManager::realtsp(std::unique_ptr<Graph<int>>& g, int s) {
    auto start : time_point<...> =std::chrono::high_resolution_clock::now();
    std::ios_base::sync_with_stdio( sync: false);
    std::vector<int> mstPredecessors = mstPrim( &: g, start: s);

    std::vector<Edge<int>*> mstEdges;
    for(int i=0;i<mstPredecessors.size();i++) {
        if(mstPredecessors[i]!=-1) {
            for(Edge<int>* e:deliveryGraph_->findVertex( in: mstPredecessors[i])->getAdj()) {
                if(e->getDest()->getInfo()==i) {
                    mstEdges.push_back(e);
                }
            }
        }
    }
    std::vector<Vertex<int>*> oddVertices = findOddDegreeVertices( &: g,  mst: mstPredecessors);
    std::vector<std::pair<Vertex<int>*, Edge<int>*>> mwpmVertices = minimumWeightPerfectMatching( &: oddVertices);
    std::vector<Edge<int>*> mwpmEdges;
    for(auto e : pair<...> :mwpmVertices) {
        mstEdges.push_back(e.second);
    }

    std::vector<Edge<int>*> eulerianEdges = combineMSTandMWPM(mstEdges, mwpmEdges);
    std::vector<int> eulerianCircuit1 = eulerianCircuit( &: g, eulerianEdges, start: s);
    double cost=calculateTSPPath( eulerianCircuit: eulerianCircuit1,  &: g,eulerianEdges);
    auto end : time_point<...> =std::chrono::high_resolution_clock::now();
    double res_t=std::chrono::duration_cast<std::chrono::nanoseconds>( d: end-start).count();
    res_t*=1e-9;

    return  std::make_pair( &: cost, &: res_t);
}
```

03 { . .

Interface & Features

< Our Menu, respective features and utils >



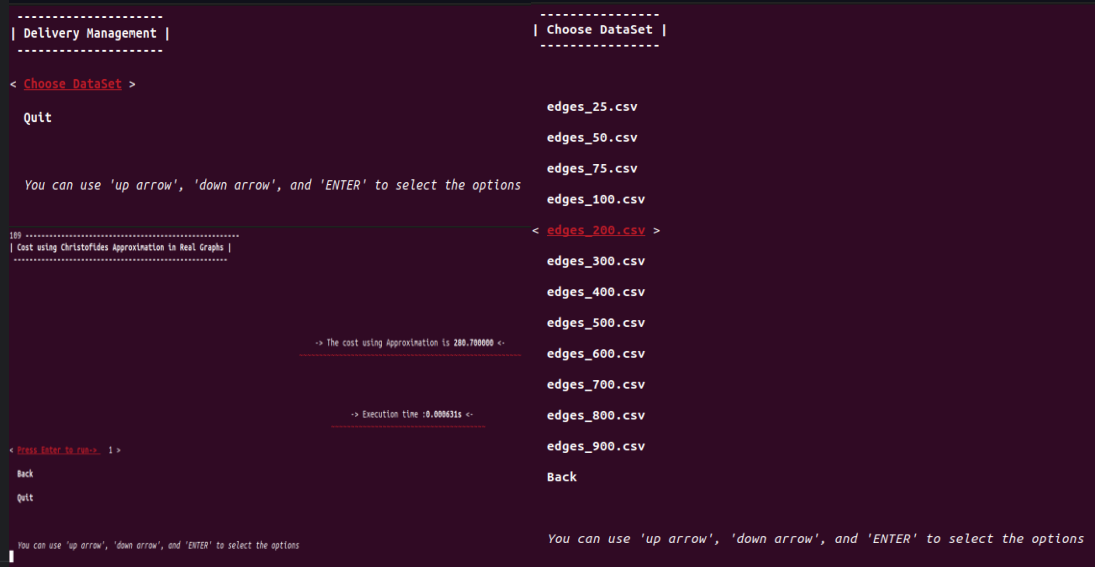
} . .

```
void Interface::run(){
    setlocale( category: LC_CTYPE,  locale: "en_US.UTF-8"); // enconding to UTF-8 for extended characters such as "ç"

    struct termios oldt{}, newt{};
    tcgetattr( fd: STDIN_FILENO,  termios_p: &oldt); // Get the current terminal settings

    newt = oldt; // Copy the current settings to the new settings

    newt.c_lflag &= ~(ICANON | ECHO); // Disable canonical mode (line buffering) and echoing

    tcsetattr( fd: STDIN_FILENO,  optional_actions: TCSANOW,  termios_p: &newt); // Set the new settings


    while(location != -1){

        system( command: "clear");

        switch (location) {
            case 0:
                printBoldTitle( title: deliveryManagement);
                printOptions( options: options[location],selected,  table_mode: false);
                printHelper(helpers,  selections: {0});
                inputer();
                break;
            case 1:
                printDirectory(directory);
                printOptions( options: options[location],selected,  table_mode: false);
                printHelper(helpers,  selections: {0});
                inputer();
                break;
            case 2:
                printDirectory(directory);
                printOptions( options: options[location],selected,  table_mode: false);
                printHelper(helpers,  selections: {0});
                inputer();
                break;

            case 3:
                printDirectory(directory);
```

Interface

```
Interface anInterface;
anIterface.run()
```

This function runs the interface.



```
case 2:
    switch (selected) {
        case 3:
            enterInputHandler( loc: 0,  sel: 0,  back: true,  main_menu: false,  main_menu2: false);
        break;
        default:
            edge_choice=converter.to_bytes( wstr: options[location][selected]);
            edge_file="";
            vertex_file="../dataset/"+data_choice+"/"+edge_choice;
            enterInputHandler( loc: 5,  sel: 0,  back: false,  main_menu: false,  main_menu2: false);
            dev_man=std::make_shared<DeliveryManager>( data_choice: vertex_file, edge_choice: edge_file);
        break;
    }


break;
case 3:
    switch (selected) {
        case 12:
            enterInputHandler( loc: 0,  sel: 0,  back: true,  main_menu: false,  main_menu2: false);
        break;
        default:
            edge_choice=converter.to_bytes( wstr: options[location][selected]);
            vertex_file="../dataset/"+data_choice+"/"+edge_choice;
            edge_file="";
            enterInputHandler( loc: 5,  sel: 0,  back: false,  main_menu: false,  main_menu2: false);
            dev_man=std::make_shared<DeliveryManager>( data_choice: vertex_file, edge_choice: edge_file);
            break;
    }
break;
case 4:
    switch (selected) {
        case 3:
            enterInputHandler( loc: 0,  sel: 0,  back: true,  main_menu: false,  main_menu2: false);
        break;
        default:
            edge_choice=converter.to_bytes( wstr: options[location][selected]);
            edge_file="../dataset/"+data_choice+"/"+edge_choice+"/edges.csv";
            vertex_file="../dataset/"+data_choice+"/"+edge_choice+"/nodes.csv";
            enterInputHandler( loc: 5,  sel: 0,  back: false,  main_menu: false,  main_menu2: false);
```

Choose DataSet Menu

Lets the user choose both the arguments of our constructor

```
case 6:
    printDirectory(directory);
    printMonoInfo( wstr: L"The cost using Backtracking is "+bold +converter.from_bytes( str: std::to_string( val: backtrack_res.first))+end_effect);
    printMonoInfo( wstr: L"Execution time :" + bold +converter.from_bytes( str: std::to_string( val: backtrack_res.second)) + L"s"+ end_effect);
    printOptions( options: options[location],selected, table_mode: false);
    printHelper(helpers, selections: {0});
    inputer();

    break;
case 7:
    printDirectory(directory);
    printMonoInfo( wstr: L"The cost using Approximation is "+bold +converter.from_bytes( str: std::to_string( val: approximation_res.first))+end_effect);
    printMonoInfo( wstr: L"Execution time :" + bold +converter.from_bytes( str: std::to_string( val: approximation_res.second)) + L"s"+ end_effect);
    printOptions( options: options[location],selected, table_mode: false);
    printHelper(helpers, selections: {0});
    inputer();
break;
case 8:
    printDirectory(directory);
    printMonoInfo( wstr: L"The cost using Approximation is "+bold +converter.from_bytes( str: std::to_string( val: christofile_res.first))+end_effect);
    printMonoInfo( wstr: L"Execution time :" + bold +converter.from_bytes( str: std::to_string( val: christofile_res.second)) + L"s"+ end_effect);
    printOptions( options: options[location],selected, table_mode: false);
    printHelper(helpers, selections: {0});
    inputer();
break;
case 9:
    writeOptionDefaulterInput();
    printDirectory(directory);
    if (do_function) {
        real_res=dev_man->realtsp( &: dev_man->getDeliveryGraph(), v: std::stoi( str: converter.to_bytes( wstr: write)));
        printMonoInfo( wstr: L"The cost using Approximation is "+bold +converter.from_bytes( str: std::to_string( val: real_res.first))+end_effect);
        printMonoInfo( wstr: L"Execution time :" + bold +converter.from_bytes( str: std::to_string( val: real_res.second)) + L"s"+ end_effect);
        do_function = false;
    }
    printOptions( options: options[location],selected, table_mode: false);
    printHelper(helpers, selections: {0});
    inputer();
break;


}
```

Results

On every option selected is presented the cost that resulted from that tsp solution and the time it takes for the function to run.

```
 -------------------------
| Toy-Graphs tourism.csv |
 -------------------------

< Change DataSet >
  Cost using Backtracking
  Cost using Trinagular Approximation
  Cost using Christofides Approximation
  Cost using Christofides Approximation in Real Graphs
  Back
  Quit

  You can use 'up arrow', 'down arrow', and 'ENTER' to select the options
```

```
109 ----------------------------------------------------
| Cost using Christofides Approximation in Real Graphs |
 ----------------------------------------------------

                                  -> The cost using Approximation is 3150.000000 <-

                                  -> Execution time :0.000212s <-

< Press Enter to run->   3 >
  Back
  Quit

  You can use 'up arrow', 'down arrow', and 'ENTER' to select the options
```

Participation

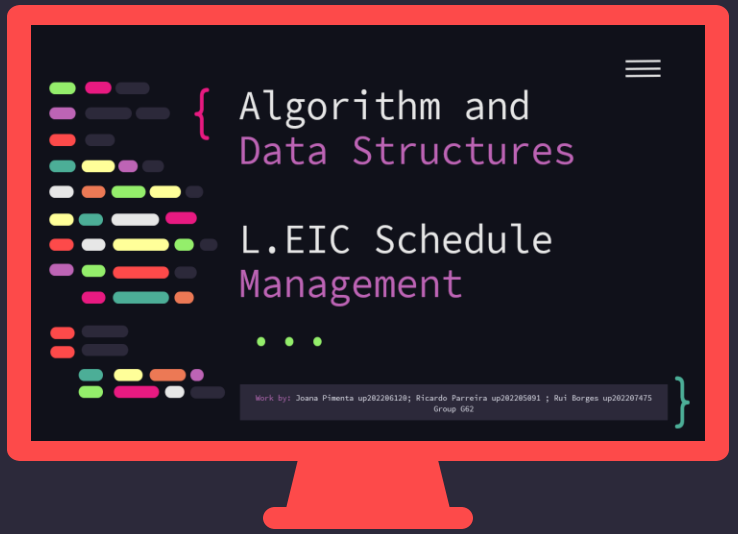
Dinis Galvão 100%

Joana Pimenta 100%

Miguel Sousa 100%

END

THE END


Algorithm and
Data Structures
L.EIC Schedule
Management
Work by: Joana Pimenta up202206120; Ricardo Parreira up202205091 ; Rui Borges up202207475
Group G62