Navegação nos transportes públicos do Porto

Faculdade de Engenharia da Universidade do Porto

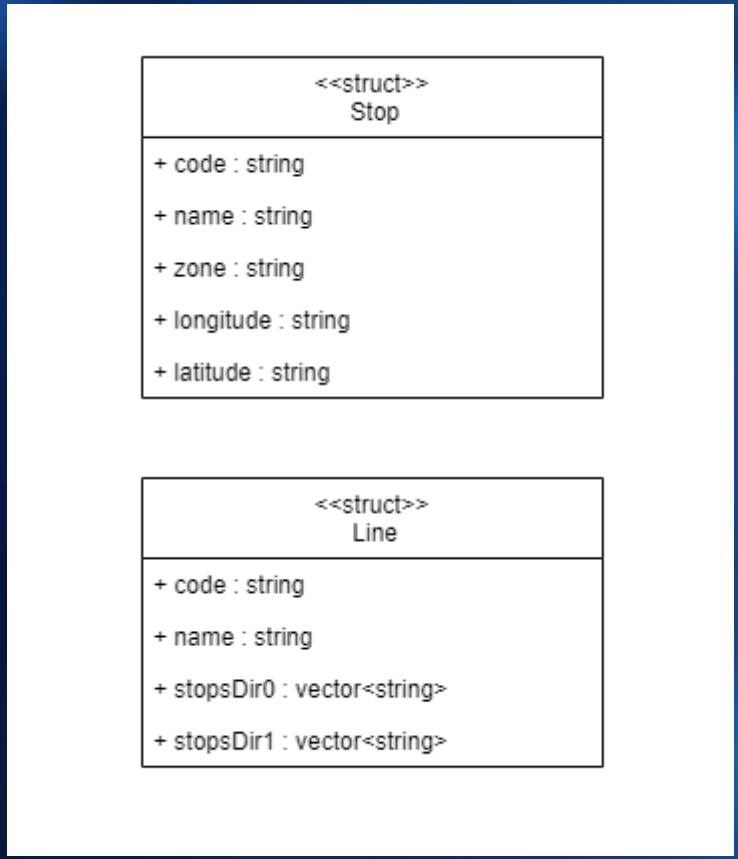
Algoritmos e Estruturas de Dados
2021/22 **L.EIC – 2ºano**

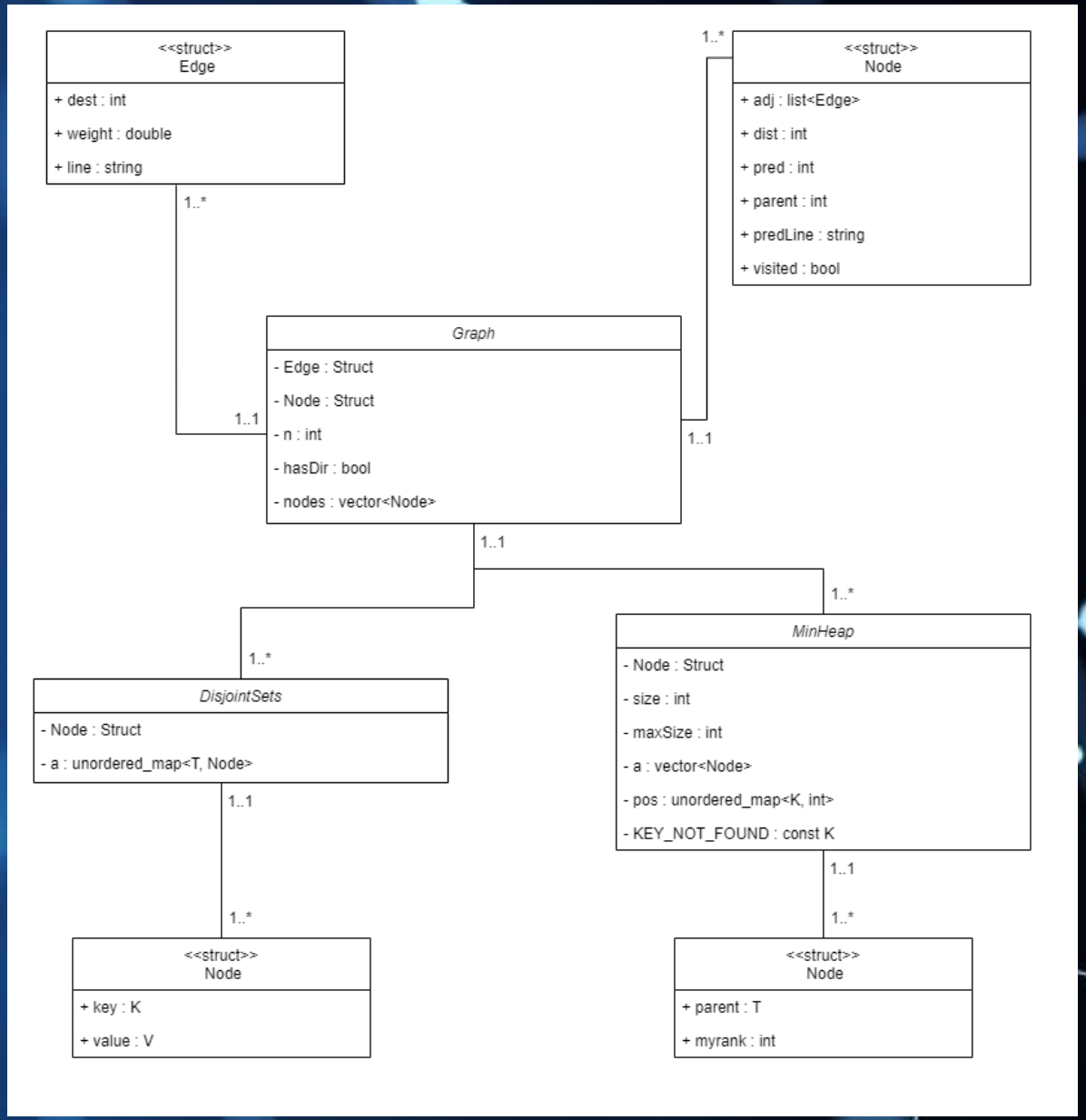
Trabalho prático 2

STCP

Trabalho realizado por: Pedro Macedo (up202007531), Pedro Balazeiro (up202005097) e Rúben Viana (up202005108) - Grupo 11

Diagrama de classes





Descrição da leitura do dataset a partir dos ficheiros dados

Temos três tipos de ficheiros no dataset que são lidos de forma diferente:

- Ficheiro **stops.csv** (2487 paragens)
- Ficheiro **lines.csv** (73 linhas de autocarro)
- Ficheiros **lines_[LINECODE]_[DIR].csv** (LINECODE é um código de linha, DIR é 0 ou 1)

```
struct Stop {        //
    string code;
    string name;
    string zone;
    string latitude;
    string longitude;
};
```

```
struct Line {
    string code;
    string name;
    vector<string> stopsDir0;
    vector<string> stopsDir1;
};
```

```
vector<Line> readLinesFile(const string &filename) {
    vector<Line> allLines;
```

Utilizamos a função readLinesFile que lê os ficheiros lines.csv e retorna a sua informação para um vetor de lines/linhas (através da Struct Line) que guarda o código e o nome de cada linha.

```
vector<Stop> readStopsFile(const string &filename, map<string,int> &stopsIndex) {
    vector<Stop> stops;
```

Utilizamos a função readStopsFile que lê o ficheiro stops.csv e retorna a sua informação para um vetor de stops/paragens (através da Struct Stop) que guarda o código, o nome, a zona, a latitude e a longitude de cada paragem. O map stopsIndex serve para sabermos a posição de cada paragem no vetor de stops.

```
void readLineStops (vector<Line> &allLines) {
```

Utilizamos a função readLineStops que lê os ficheiros lines_[LINECODE]_[DIR].csv e retorna a sua informação para um vetor de lines/linhas (através da Struct Line) que guarda as paragens no vetor stopsDir0 quando o DIR é 0 ou no vetor stopsDir1 quando o DIR é 1 de cada linha.

Descrição do(s) grafos usado(s) para representar o dataset, com imagens (parciais) ilustrativas

```
class Graph {
    struct Edge {
        int dest;   // Destination node
        double weight; // An integer weight
        string line; //
    };

    struct Node {
        list<Edge> adj; // The list of outgoing edges (to adjacent nodes)
        int dist;
        int pred;
        int parent;
        string predLine;
        bool visited;
    };

    int n;              // Graph size (vertices are numbered from 1 to n)
    bool hasDir;        // false: undirect; true: directed
    vector<Node> nodes; // The list of nodes being represented
```

Temos um grafo constituído por duas Structs (Edge e Node), um inteiro n que nos dá o número vértices do grafo, um booleano que nos diz se o grafo é direto ou indireto e um vetor de nodes/nós que representa todos os nós do grafo. A Edge é constituída pelo nó de destino (int dest), pelo peso que representa a distância entre duas paragens (double weight), e uma linha à qual pertence a paragem (string line). O Node é constituído por uma lista de nós adjacentes (list<Edge> adj), por uma distância (int dist), por um predecessor (int pred), por um parente (int parent), por uma linha predecessora (string predLine) e por um booleano que nos diz se foi visitado (bool visited), sendo estes últimos atributos bastantes úteis na utilização de alguns algoritmos.

Descrição do(s) grafos usado(s) para representar o dataset, com imagens (parciais) ilustrativas

```
Graph directedGraph(stopsIndex.size(), dir: true);
Graph undirectedGraph(stopsIndex.size(), dir: false);
```

Temos dois grafos criados, um direcionado e outro que não é direcionado.

```
for (int j = 0; j < allLines.size(); j++){
    for (int i = 0; i < allLines[j].stopsDir0.size() - 1; i++) {
        directedGraph.addEdge(stopsIndex[allLines[j].stopsDir0[i]], dest: stopsIndex[allLines[j].stopsDir0[i + 1]],
                              weight: haversine( lat1: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i]] - 1].latitude),
                                                 lon1: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i]] - 1].longitude),
                                                 lat2: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i + 1]] - 1].latitude),
                                                 lon2: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i + 1]] - 1].longitude)),
                              line: allLines[j].code);
        undirectedGraph.addEdge(stopsIndex[allLines[j].stopsDir0[i]], dest: stopsIndex[allLines[j].stopsDir0[i + 1]],
                                weight: haversine( lat1: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i]] - 1].latitude),
                                                   lon1: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i]] - 1].longitude),
                                                   lat2: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i + 1]] - 1].latitude),
                                                   lon2: stod( _Str: stops[stopsIndex[allLines[j].stopsDir0[i + 1]] - 1].longitude)),
                                line: allLines[j].code);
    }
    if (allLines[j].stopsDir1.size() != 0) {
        for (int i = 0; i < allLines[j].stopsDir1.size() - 1; i++) {
```

Para guardar a informação dos ficheiros no grafo percorremos todas as linhas e consequentemente os vetores de paragens de cada linha em ambas as direções (0 e 1) e com a função addEdge do grafo adicionamos ao vetor de nodes as paragens, o destino delas, e a sua distância com a ajuda do algoritmo de haversine.

Descrição das funcionalidades implementadas e algoritmos (e grafos) associados (indicar complexidades)

Origem/Destino: diretamente uma paragem para outra? de um local/conjunto de paragens para outro local? outros?

A origem e o destino podem ser dados pelo cliente em latitude e longitude ou em paragens no menu principal.

```
"\tMAIN MENU\n\n";
"1 - INPUT : LATITUDE/LONGITUDE\n";
"2 - INPUT : STOP_CODE\n";
```

Conceito de "melhor" caminho: nº de paragens? distância? nº mudanças de linha? nº zonas? outros?

O cliente pode escolher o método de procura entre as opções menor distância ou menor número de paragens no menu de procura. Podem também retornar ao menu principal.

```
cout << "SEARCH METHOD :\n\n";
cout << "1 - Less Distance\n";
cout << "2 - Less Stops\n";
cout << "0 - Return\n";
```

Descrição das funcionalidades implementadas e algoritmos (e grafos) associados (indicar complexidades)

Algoritmo dijkstra :

Algoritmo de complexidade *0 (N * log N)* que dá o caminho mais curto. Neste caso, a função representada ao lado dá-nos o caminho mais curto começando na paragem "s".

```
int Graph::dijkstra_distance(int a, int b) {
    dijkstra(a);
    if (nodes[b].dist == INF) return -1;
    return nodes[b].dist;
}
```

Na função acima é nos dada a distância entre duas paragens utilizando este algoritmo.

```
/**
 * Computes the shortest path starting in node "s" to all other nodes of the graph.
 * Temporal Complexity: O(N * log N)
 * @param s (starting stop)
 */
void Graph::dijkstra(int s) {
    MinHeap<int, int> q(n, notFound: -1);
    for (int v=1; v<=n; v++) {
        nodes[v].dist = INF;
        q.insert(v, value: INF);
        nodes[v].visited = false;
    }
    nodes[s].dist = 0;
    q.decreaseKey(s, value: 0);
    nodes[s].pred = s;
    while (q.getSize()>0) {
        int u = q.removeMin();
        nodes[u].visited = true;
        for (auto e : nodes[u].adj) {
            int v = e.dest;
            int w = e.weight;
            if (!nodes[v].visited && nodes[u].dist + w < nodes[v].dist) {
                nodes[v].dist = nodes[u].dist + w;
                q.decreaseKey(v, nodes[v].dist);
                nodes[v].pred = u;
                nodes[v].predLine = e.line;
```

Descrição das funcionalidades implementadas e algoritmos (e grafos) associados (indicar complexidades)

Algoritmo dijkstra :

```
list<int> Graph::dijkstra_path(int a, int b) {
    list<int> path;
    dijkstra(a);
    if (nodes[b].dist == INF) return path;
    while(a != b) {
        path.push_front(b);
        b = nodes[b].pred;
    }
    path.push_front(a);
    return path;
}
```

Na função ao lado é nos dado todas as paragens pela qual tem se de passar para chegar ao destino pretendido utilizando o algoritmo de dijkstra.

```
list<string> Graph::dijkstra_pathLines(int a, int b) {
    list<string> path;
    dijkstra(a);
    if (nodes[b].dist == INF) return path;
    while(a != b) {
        path.push_front(nodes[b].predLine);
        b = nodes[b].pred;
    }
    path.push_back( x: "---");
    return path;
}
```

Na função ao lado é nos dado todas as linhas pela qual tem se de passar para chegar ao destino pretendido utilizando o algoritmo de dijkstra.

Descrição das funcionalidades implementadas e algoritmos (e grafos) associados (indicar complexidades)

Algoritmo bfs :

Algoritmo de complexidade *0 (n)* que dá o caminho que passa por menos paragens. Neste caso, a função representada ao lado dá-nos o caminho que passa por menos paragens começando na paragem "v".

```
list<int> Graph::bfs_path(int a, int b){ //
    list<int> path;
    bfs(a);
    while(a != b) {
        path.push_front(b);
        b = nodes[b].pred;
    }
    path.push_front(a);
    return path;
}
```

Na função acima é nos dado todas as paragens pela qual tem se de passar para chegar ao destino pretendido utilizando o algoritmo bfs.

```
/**
 * Computes the path that goes through less nodes.
 * Temporal Complexity: O(n)
 */
void Graph::bfs(int v) {
    for (int v=1; v<=n; v++){
        nodes[v].visited = false;
        nodes[v].dist = -1;
    }
    queue<int> q; // queue of unvisited nodes
    q.push(v);
    nodes[v].dist = 0;
    nodes[v].visited = true;
    nodes[v].pred = v;
    while (!q.empty()) { // while there are still unvisited nodes
        int u = q.front(); q.pop();
        for (auto e : nodes[u].adj) {
            int w = e.dest;
            if (!nodes[w].visited) {
                q.push(w);
                nodes[w].visited = true;
                nodes[w].dist = nodes[u].dist + 1;
                nodes[w].pred = u;
                nodes[w].predLine = e.line;
            }
        }
    }
}
```

Descrição das funcionalidades implementadas e algoritmos (e grafos) associados (indicar complexidades)

Algoritmo bfs :

```
/**
 * Computes the shortest path starting in node "s" to all other nodes of the graph.
 * @param a
 * @param b
 * @return Path that goes trough less nodes
 */
list<string> Graph::bfs_pathLines(int a, int b) {
    list<string> path;
    bfs(a);
    if (nodes[b].dist == INF) return path;
    while(a != b) {
        path.push_front(nodes[b].predLine);
        b = nodes[b].pred;
    }
    path.push_back( x: "---");
    return path;
}
```

Na função ao lado é nos dado todas as linhas pela qual tem se de passar para chegar ao destino pretendido utilizando o algoritmo bfs.

Descrição das funcionalidades implementadas e algoritmos (e grafos) associados (indicar complexidades)

Algoritmo Prim :

```
/**
 * Calculates the MST (minimum spanning tree) of the graph.
 * Temporal Complexity: O(N * log N)
 * @param r
 * @return Minimum distance in order to pass through all the nodes
 */
int Graph::prim(int r) {
```

Na função acima é nos dada a MST do nosso grafo. É de notar que este é o único algoritmo em que foi utilizado o grafo não direcionado, nos outros foi sempre utilizado o direcionado. A complexidade deste algoritmo é *O (N * log N)*.

Algoritmo Kruskal :

```
/**
 * Calculates the MST (minimum spanning tree) of the graph.
 * Temporal Complexity: O(N * log N)
 * @return Minimum distance in order to pass through all the nodes
 */
int Graph::kruskal() {
```

Na função acima é nos dada a MST do nosso grafo tal como no algoritmo Prim. A complexidade deste algoritmo é *O (N * log N)*.

Demonstração de todas as paragens e respetivas linhas:

```
void Graph::print () {
    for (int i = 1; i <= n; i++){
        for (auto &d: nodes[i].adj)
            cout << i << " -- " << d.weight << " -> " << d.dest << " : Line " << d.line << endl;
    }
}
```

Descrição do interface com o utilizador (pode incluir exemplo de utilização)

Menu principal :

```
cout << "\tMAIN MENU\n\n";
cout << "1 - INPUT : LATITUDE/LONGITUDE\n";
cout << "2 - INPUT : STOP_CODE\n";
cout << "3 - GLOBAL NETWORK MST\n";
cout << "0 - EXIT\n";
cout << endl << "Option : ";
```

No menu principal é permitido ao utilizador escolher o seu percurso em coordenadas, em paragens e ainda permite-lhe saber a MST global.

Método de procura :

```
cout << "SEARCH METHOD :\n\n";
cout << "1 - Less Distance\n";
cout << "2 - Less Stops\n";
cout << "0 - Return\n";
cout << endl << "Search Method : ";
```

No caso do cliente selecionar o seu percurso, é lhe pedido se prefere ir pelo de menor distância, pelo de menor número de paragens ou se pretende retornar ao menu principal. Se escolher o de menor distância, é lhe dado as paragens e respetivas linhas a percorrer e o mesmo se aplica se escolher o de menor número de paragens.

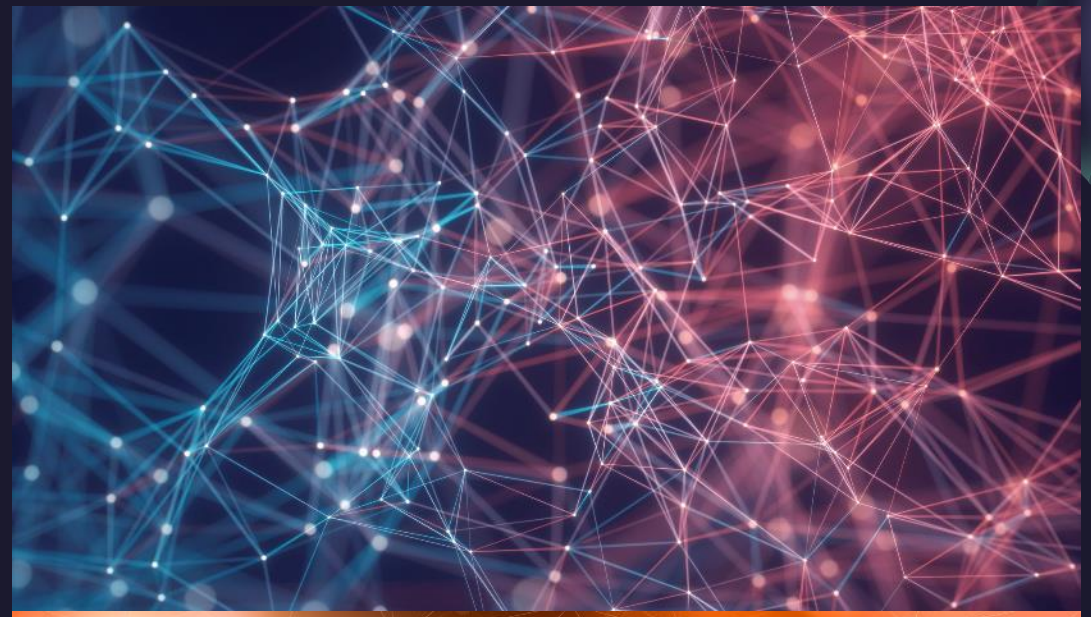
Destaque de funcionalidade (o que mais vos deixou orgulhosos no vosso trabalho?)

Algoritmo dijkstra e funcionalidade menor distância :

A resposta mais provável é a utilização do algoritmo referido para calcular a menor distância e tudo o que é demonstrado com esta funcionalidade utilizando o algoritmo (nos diapositivos anteriores está tudo bem explicito, sendo assim nada a acrescentar aqui).

Principais dificuldades encontradas e esforço de cada elemento do grupo

Não foram encontradas grandes dificuldades além do tempo escasso e a própria agenda deste trabalho prático, que coincidia com muitos outros projetos de outras cadeiras e um aproximar de uma época de exames… Tal não permitiu uma exploração mais ampla e aproveitamento melhor do tema deste projeto como pretendíamos ter feito (por exemplo, explorar mais algoritmos e novas funcionalidades para os clientes da stcp). Relativamente ao trabalho de cada elemento do grupo este foi uniforme.



Obrigado