Faculdade de Engenharia da Universidade do Porto

BIKE SHARING – SISTEMA DE PARTILHA DE BICICLETAS

Conceção e Análise de Algoritmos – T2G05

Elementos do Grupo:

André Filipe Pinto Esteves up201606673@fe.up.pt

Francisco Manuel Canelas Filipe up201604601@fe.up.pt

Pedro Miguel Sousa Fernandes up201603846@fe.up.pt

11 de abril de 2018

Índice

1. Tema do trabalho

No âmbito da unidade curricular de Conceção e Análise de Algoritmos foi-nos proposto como tema de trabalho: Bike Sharing – Sistema de Partilha de Bicicletas. Este sistema consiste em pontos espalhados na cidade, onde uns números de bicicletas estão disponíveis para aluguer. Após indicação de um número de cartão de crédito, ou outra forma de pagamento eletrónico, o utente pode retirar uma bicicleta, utilizá-la, e devolvê-la em qualquer outro ponto de partilha disponível, onde há vaga disponível.

Este trabalho centrou-se principalmente na implementação de um sistema de gestão que auxilia o utente a identificar o ponto de partilha mais próximo de onde se encontra, com lugar disponível para a devolução da bicicleta. A escolha do ponto de retorno deverá também ter em consideração, para além da distância, a topografia (elevação) das ruas. Para evitar que lugares mais elevados, ou mais afastados na cidade fiquem sem bicicletas, elaborámos também um sistema de incentivos oferece descontos aos utentes que optarem por realizar a devolução em tais sítios.

Avaliámos ainda a conectividade do grafo, a fim de evitar que locais de pontos de partilha se encontrassem em zonas inacessíveis, considerando que as bicicletas circulam nas ruas e também o efeito da circulação ter de obedecer o sentido das vias, e as situações onde as bicicletas podem circular em qualquer sentido.

2. Identificação e Formalização do Problema

De modo a simplificar o problema inicial, optámos por uma abordagem de subdivisão de problemas. Desta forma fomos dividindo um problema complexo em vários mais simples de uma forma sucessiva até estes se tornarem elementares.

2.1. Problemas Encontrados

- Leitura dos dados provenientes do Open Street Maps.
- Gerar a elevação de acordo com as coordenadas fornecidas.
- Peso das arestas do grafo.
- Reconhecer que este problema deriva diretamente de um problema já conhecido e deduzir a melhor solução.
- Implementação de um algoritmo de forma a balancear o grafo (evitar que lugares mais elevados, ou mais afastados na cidade fiquem sem bicicletas).
- Conversão dos dados de modo a possibilitar a utilização do GraphViewer.
- Cruzamentos de ruas com o mesmo nome.
- Acentuação nos nomes das ruas.

Formalizamos agora o problema, de acordo com aquela que achamos ser a melhor forma para resolver aquilo a que nos propusemos.

2.2. Formalização do Problema

Input

G<V, E>,
V: Pontos das ruas.
E: ligações entre pontos (distância entre estes);
P0: ponto inicial
Pi: I=1...n
Pf: ponto final

Output

Caminho= {Vi}, I= 1 … n
Valor

Objetivo

Min (valor);
valor = f(x)= Σ(Eij) ni=1
ij ∈ Caminho

3. Solução Implementada

3.1. Leitura dos Dados

3.1.1. Preparação dos ficheiros de entrada

A criação de um grafo é uma tarefa que requer vários passos: primeiro, é necessário extrair um mapa do *openstreetmaps* e avaliar os dados usando a ferramenta OSM2TXT, fornecida no moodle, que retorna 3 ficheiros: vértices, arestas e nomes.

No entanto, rapidamente se concluiu que tal não é suficiente. Como a altitude toma um papel fundamental neste trabalho, é preciso criar uma forma de a obter, visto que é impossível fazê-lo a partir do *openstreetmaps*. Além disso, deve ser possível ter um ficheiro com informação sobre os locais de partilha. Para tal, foi desenvolvido um programa ***prepFiles*** que recebe um ficheiro de vértices, e retorna dois novos ficheiros: um ficheiro de vértices editado, já com informação sobre a altitude, e uma string que informa se cada linha de texto representa uma localização normal, ou de partilha; e também um ficheiro com informação sobre os locais de partilha, nomeadamente a lotação máxima e o número de vagas disponíveis. Este programa está dividido em dois módulos, expostos de seguida.

3.1.2. Elevação

Esta era uma parte muito importante do nosso trabalho uma vez que a elevação era um dos fatores que contribuia para o peso de cada aresta. Assim, achámos por bem que seria mais interessante a utilização de valores reais de elevação.

Através de uma pesquisa na internet é possível encontrar algumas APIs, que permitem especificar um conjunto de localizações, seja através do nome ou de coordenadas geográficas, e retornar a altitude, em metros. Neste trabalho é utilizada a open-elevation, uma alternativa grátis ao *google maps api* e também de código livre. Tirando partido do comando *cURL* e da API do UNIX, este módulo retira as coordenadas do ficheiro de vértices, e constrói um comando cURL, a ser usado através da chamada system(). Depois, o resultado é analisado e colocado no novo ficheiro.

3.1.3. CreateSharing

Este módulo tem a função de criar as localizações de partilha. Para tal, fá-lo aleatoriamente, criando um número de localizações especificado pelo utilizador, e em cada uma delas, coloca um número aleatório para a lotação máxima e para as vagas disponíveis, isto dentro de um intervalo aceitável.

3.1.4. Utilização

Compilação:

```
g++ -o [nome] prepFiles.cpp createSharing.cpp elevation.cpp -std=c++11
```

Os avisos devem ser ignorados, não têm qualquer influência no resultado.

Para correr, devem ser especificados 4 argumentos:

- nome do ficheiro dos vértices
- nome do ficheiro onde os vértices, já alterados, devem ser guardados
- nome do ficheiro onde as localizações de partilha devem ser guardadas
- número de localizações de partilha a serem criadas

Destes ficheiros, apenas o primeiro deve obrigatoriamente existir.

Eis um exemplo:

```
./prep nodes.txt nodesPorto.txt sharingPorto.txt 50
```

Em caso de esquecimento, também é possível ver as instruções de uso ao correr o programa sem argumentos.

3.2. Peso das arestas do Grafo

Ao contrário da maioria dos outros temas, no nosso tema, para além de termos de considerar a distância como fator relevante para o peso de cada aresta, é ainda necessário ter em conta a elevação de cada local. Desta forma decidimos considerar o tempo como fator que influencia o peso das arestas do grafo. No cálculo deste peso tentámos ao máximo ser coerentes possíveis de modo a que os nossos pesos se aproximassem da realidade. No entanto, devido à imprecisão dos resultados provenientes da API utilizada, os pesos das arestas não se mostraram tão realistas quanto o esperado. Para combater este desequilíbrio nos resultados decidimos diminuir a velocidade média a que um ciclista circula.

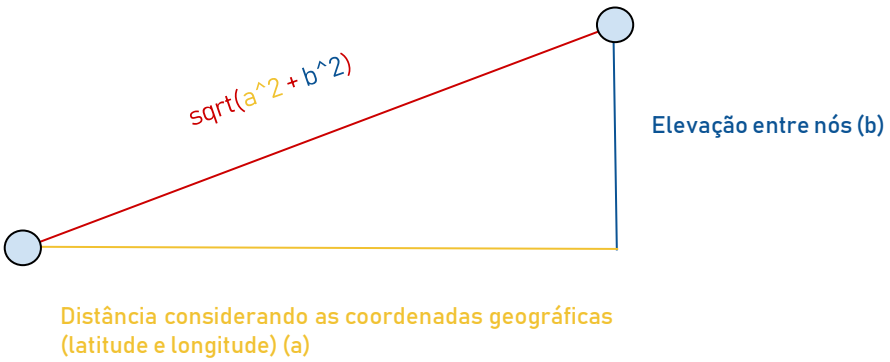
Através da fórmula que nos dá a velocidade média é possível obter o tempo médio que um utente deverá demorar entre cada aresta.

$$v = \frac{\Delta d}{\Delta t} \quad \Delta t = \frac{\Delta d}{v}$$

Assim, tendo conhecimento da distância e da velocidade média conseguimos obter o tempo médio de travessia entre dois nós consecutivos.

3.2.1. Distância entre dois Nós

Para o cálculo deste parâmetro tivemos de ter em conta a diferença de elevação entre os dois nós assim como a distância entre estes excluindo a elevação. O seguinte desenho tenta ilustrar o nosso raciocínio:



Desta forma, começamos primeiro por calcular a distância geográfica[1] e depois, através do teorema de Pitágoras é possível calcular a distância entre dois nós consecutivos tendo em conta a elevação. Obtendo a distância, falta agora saber a velocidade para poder calcular o tempo médio entre nós consecutivos.

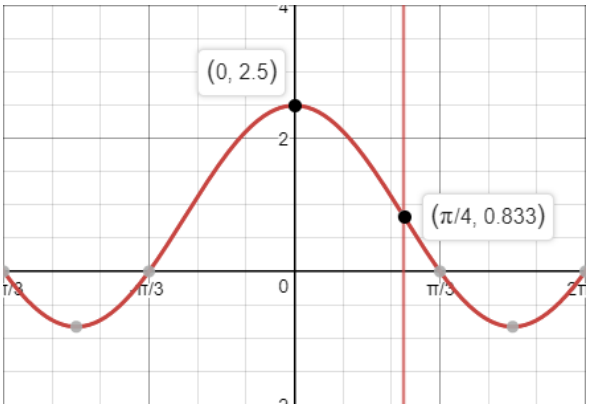
[1] Fórmula para cálculo de distância geográfica: http://www.movable-type.co.uk/scripts/latlong.html

3.2.2. Velocidade média entre dois Nós

A velocidade, tal como a distância, depende também da elevação, mais concretamente a inclinação (declive). Nos cenários em que o nó de origem se situa num local menos elevado que o nó de destino, ou seja, o declive é positivo, a velocidade aumenta com a diminuição deste declive. Nos casos em que o nó de origem se situa num local mais elevado que o nó de destino, onde o declive é negativo, a velocidade aumenta com o aumento deste declive.

Depois de realizada esta análise tentámos averiguar quais os valores mais coerentes de velocidades que uma pessoa consegue atingir numa bicicleta tendo em conta a elevação. Com isto em mente construímos duas funções de velocidade em função da inclinação (uma para o caso do nó de origem ser menos elevado que o nó de destino e outra para o caso inverso). Estas foram as funções a que chegámos que, na nossa opinião, são uma boa aproximação à realidade.

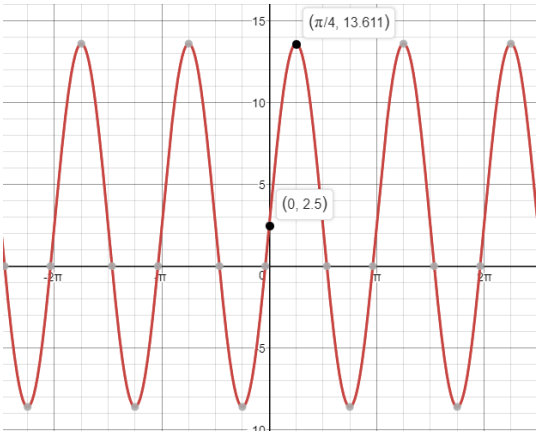
Situações de subida:



$$\frac{5}{3} * cos(2 * x) + \frac{5}{6}$$

Nesta situação apenas interessa ter uma boa aproximação no intervalo de 0 a 45°($\frac{\pi}{4}$) de inclinação. A partir daí torna-se pouco viável a locomoção usando a bicicleta como meio de transporte.

Situações de descida:



$$\frac{100}{9} * sin(2 * x) + \frac{5}{2}$$

Tal como na situação anterior apenas o intervalo de 0 a 45° ($\frac{\pi}{4}$). A partir deste ângulo consideramos que é atingida a velocidade máxima que um ciclista consegue atingir. Este ângulo na realidade aumenta no sentido negativo.

Estas duas funções retornam valores em m/s recebendo a inclinação em radianos.

3.3. Encontrar a melhor Solução

Depois de os dados serem lidos e carregados corretamente para o programa corretamente foi necessário pensar na parte principal do projeto e de como a implementar.

Num primeiro instante procurámos reconhecer que tipo de problema era este. Muito facilmente chegámos a um consenso e percebemos que este problema era um típico problema do caminho mais curto. Desde logo decidimos que o algoritmo mais indicado seria o algoritmo de Dijkstra.

No algoritmo seguinte sejam **G** o grafo inicial, **vi** o vértice de origem, p(u,v) o peso de uma aresta, d{u} a distância até ao vértice u, Q uma fila de prioridade e **V** o conjunto de vértices.

```
Dijkstra (G, vi)

for all u ∈ V \ vi, d{u} = ∞ and path{u} = null
d{s} = 0
let Q = ∅ // priority - queue
insert vi in Q
u = null
while (Q ≠ ∅and (u == null or !u.isAvailable( ) ) )
min = extractMin(Q)
for all vertices v adjacent to u
        if d{v} > d{u} + p(u,v)
                d{v} = d{u} + p(u,v)
                path{v} = u
```

Para os casos em que o destino é conhecido a condição de terminação do algoritmo passa a ser:

```
while (Q ≠ ∅and (u == null or u != vf ) )
```

Em que **vf** é o vértice de destino.

3.3.1. Análise de Correção da Solução

O algoritmo utilizado neste projeto, tal como já foi referido antes, foi o algoritmo de Dijkstra. A correção deste algoritmo já está provada tal como foi visto nas aulas teóricas desta Unidade Curricular (prova de correção do algoritmo presente nos slides sobre Grafos).

Em relação à análise da estratégia implementada caso o utilizador opte pelo incentivo, a sua correção é intuitiva. A nossa implementação baseia-se em reduzir o intervalo entre os números mínimo e máximo de bicicletas (delta) em cada ponto de partilha, de modo a que os valores destes se aproximem de um valor médio. Considerando cada um dos nossos casos de implementação:

- Rent a bike: Quando o utilizador tenciona alugar uma bicicleta, ao sugerirmos a utilização de uma das bicicletas do ponto de partilha que mais bicicletas possui, estamos a reduzir o número de bicicletas neste ponto,diminuindo assim o delta e equilibrando melhor o número de bicicletas nos pontos de partilha.

- Deliver a bike: Quando o utilizador tenciona alugar uma bicicleta, ao sugerirmos a entrega da sua bicicleta no ponto de partilha que menos bicicletas possui, estamos a aumentar o número de bicicletas neste ponto, diminuindo assim o delta e equilibrando melhor o número de bicicletas nos pontos de partilha.

3.3.2. Análise da Complexidade da Solução

3.3.2.1. Análise Teórica

Tendo em conta que a solução encontrada se reduz, na prática, à utilização de um algoritmo já conhecido (Algoritmo do caminho mais curto de Dijkstra), a sua complexidade temporal é também já conhecida, sendo ela de:

$$O((|V| + |E|) * \log |V|)$$

Seleção das "menores" localizações de partilha

```
vector<Vertex *> Graph::discountLocations(bool rent, const int numberOfLocations)
{
    vector<Vertex *> sharingLocations;

    copy_if(vertexSet.begin(), vertexSet.end(), back_inserter(sharingLocations), [&rent](Vertex *vertex) { return string(typeid(*vertex-
>getInfo()).name()) == "class SharingLocation" && vertex->getInfo()->isAvailable(rent); });

    sort(sharingLocations.begin(), sharingLocations.end(), [rent](const Vertex *lhs, const Vertex *rhs) {
        return rent ? ((SharingLocation *)lhs->getInfo())->getSlots() > ((SharingLocation *)lhs->getInfo())->getSlots() : ((SharingLocation
*)lhs->getInfo())->getSlots() < ((SharingLocation *)lhs->getInfo())->getSlots();
    });

    if ((int)sharingLocations.size() > numberOfLocations)
        sharingLocations.resize(numberOfLocations);

    return sharingLocations;
}
```

Esta função percorre o vetor de vértices uma vez, e aplica o método std::sort (quicksort) a um vetor de vértices cujo conteúdo é uma *SharingLocation*. Assim, sendo N o número total de vértices e S o número de *SharingLocations* (sendo que S é uma parte de N) então a complexidade média temporal deste algoritmo é:

$$O(N + S*\log(S))$$

3.3.2.2. Análise Experimental

O programa para cada caso de pesquisa apresenta o tempo decorrido no algoritmo.

De seguida apresentam-se alguns testes efetuados em três ficheiros com tamanho diferente do grafo: FEUP (~1000 nós), Trindade (~4000 nós), Baixa Porto(~800 nós).

- Caminho mais curto entre localização do utilizador e local de partilha mais próximo

Ficheiro	Nó de partida	Nó encontrado	Tempo procura (s)	Tempo de viagem (min)
FEUP	263	857	0.001	03:12
Trindade	3554	1405	0.002	02:07
Baixa Porto	134	508	0.001	04:12

- Caminho mais curto entre localização do utilizador e local de partilha definido pelo mesmo

A não esquecer que o utilizador não pode escolher um ponto qualquer, mas sim um de um conjunto de locais apresentado, e que tem caminho possível.

Ficheiro	Nó de partida	Nó escolhido	Tempo procura (s)	Tempo de viagem (min)
FEUP	546	254	0.010	13:14
Trindade	1548	44	0.049	23:12
Baixa Porto	152	221	0.009	12:37

3.4. Balancear os Pontos de Recolha

De maneira a evitar que Pontos de Recolha fiquem lotados ou vazios, e de modo a equilibrar a distribuição das bicicletas por todos estes pontos, foi-nos também pedido que arranjássemos uma estratégia de incentivo que levasse o utilizador a depositar a sua bicicleta nestes locais. Desta forma tivemos de arranjar um algoritmo que prevenisse este desfasamento. O algoritmo que desenvolvemos consiste no seguinte:

- Percorrer o grafo e encontrar todos os Pontos de Recolha colocando-os num vetor.
- Ordenar o vetor de acordo com o critério desejado:
    1. Rent a bike: Ordenar o vetor segundo o maior número de vagas.
    2. Deliver a bike: Ordenar o vetor segundo o menor número de vagas.
- Tendo o vetor ordenado retornar os 5 primeiros Pontos de Recolha deste.
- Calcular o incentivo de cada um destes pontos de acordo com o peso total do caminho encontrado (aumento do peso ⇒ aumento do incentivo).
- No final o utilizador poderá escolher um destes 5 Pontos de Recolha caso opte por um caminho que não o mais curto.

Desta forma garantimos que, quando o utilizador optar por usufruir do incentivo sugerido, este incentivo será no sentido de balancear os Pontos de Recolha. Note-se ainda que este balanceamento só acontece quando o utilizador opta pelo incentivo. Isto significa que, caso nenhum utilizador siga esta recomendação, existe a possibilidade de o grafo não ficar minimamente bem balanceado.

3.5. Preparar os dados para o GraphViewer

De modo a que o grafo gerado pelo nosso programa tome uma forma aproximada à forma real deste, foi necessário converter as coordenadas geográficas fornecidas de modo a satisfazer esta caraterística. Para isso e após alguma pesquisa encontrámos a nossa resposta em:

https://github.com/pedro-c/CAL-FEUP-EasyPilot/blob/master/src/EasyPilot.cpp

Neste ficheiro estão duas funções que convertem cada uma das coordenadas para coordenadas cartesianas (x e y). Este código não foi desenvolvido por nós e como tal todo o crédito vai para o seu autor e dono do repositório Pedro Costa.

3.6. Influência do sentido das ruas

Após vários testes do programa, consegue-se verificar que o facto de os ciclistas terem que obedecer o sentido das ruas influencia bastante o encontro de uma solução para o seu pedido (caminho mais próximo ou não). Desta forma, ao visualizar o *GraphViewer*, o utilizador pode ficar surpreso pelo facto de ver uma *SharingLocation* relativamente perto da sua localização, mas o programa não o direciona para lá. Tal se justifica pelo facto de parte das ruas do grafo serem unidirecionais, logo não há forma de encontrar caminho, sem violar as regras de trânsito.

3.7. Conectividade do grafo

Uma das funções do programa, é testar a conectividade do grafo, aplicando o método DFS (Depth-First-Search). Nos grafos usados no projeto, não se deu o caso de, com os testes efetuados, terem encontrado localizações de partilha completamente inatingíveis, apesar de estas serem geradas de forma aleatória. No entanto, tal como explicado acima, determinados nós podem ser inalcançáveis se a pesquisa começar num determinado local que apenas parte para sentidos contrários.

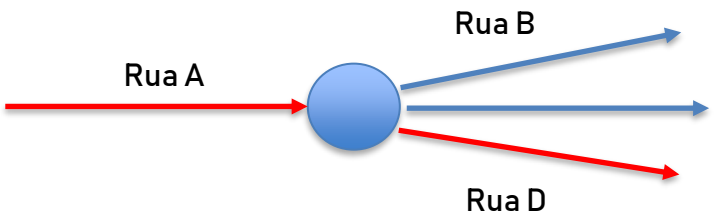
3.8. Strings

3.8.1. Acentuação

Um dos problemas que surgiu no desenvolvimento deste trabalho foi a existência de strings com carateres não ASCII que o nosso programa não conseguia interpretar de forma correta. Devido ao elevado número de strings deste tipo, este problema teve obrigatoriamente de ser corrigido. No entanto a sua correção revelou-se ser relativamente simples. Apenas foi necessário abrir, com um editor de texto,os ficheiros que possuíssem estas strings e substituir estes caracteres por caracteres ASCII.

3.8.2. Interseção das Ruas

Para verificar a interseção entre duas ruas é necessário que detenham um vértice em comum. Ou seja, ao encontrar uma rua apenas é necessário percorrer as ruas do vértice associado verificando se a outra rua está presente nesse conjunto.



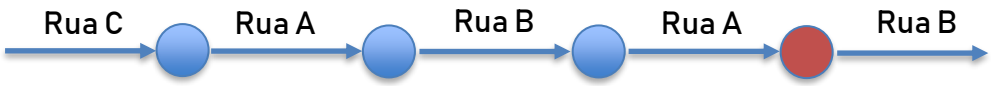
Ex: Interseção entre as ruas A e D

3.8.3. Pesquisa de Strings no Grafo

Numa primeira abordagem, para implementar a interseção entre as ruas dadas como input ao nosso programa, o algoritmo baseava-se no seguinte:

- Percorrer as arestas do grafo procurando as ruas fornecidas pelo utilizador.
- Caso sejam encontradas as ruas, verificar se estas possuem algum vértice em comum.
- Se tal se verificar, verificar se o ponto de interseção é uma Sharing Location.
- Retorna true se encontrou a sharingLocation senão retorna false.

No entanto, deparámo-nos com um problema em que várias arestas possuíam o mesmo nome.



Ex: Pesquisa de uma sharingLocation entre as ruas A e B.

Desta forma o algoritmo anteriormente enunciado não funcionava. pois da primeira vez que encontrasse uma das ruas, apenas era verificado se se intersetava com a outra rua, deixando de pesquisar possívies sharingLocations no resto do grafo(exemplo acima apresentado) .

Como tal, foi necessário alterar o algoritmo de forma a resolver este problema, resultando no seguinte:

- Percorrer as arestas do grafo procurando as ruas fornecidas pelo utilizador.
- Ao encontrar uma rua verificar se esta se cruza com a outra,.
- Caso isto aconteça, verificar se o ponto de interseção é uma Sharing Location.
- Caso o seja, retornar true, caso contrário continuar a procurar até encontrar ou chegar ao fim do vetor.

3.8.4. Algoritmos Utilizados

Foram utilizados como algoritmos de pesquisa de strings os algoritmos KMPMatcher e editDistance. Ambos foram baseados no pseudo-código do slides das aulas teóricas da Unidade Curricular.

3.8.4.1. Análise Teórica

O algoritmo KMPMatcher:

```
vector<int> ComputePrefixFunction(string pattern)
{
    int m = pattern.size();
    vector<int> pi(m);
    int k = 0;

    for (int q = 1; q < m; q++)
    {
        while (k > 0 && pattern.at(k) != pattern.at(q))
            k = pi.at(k - 1);

        if (pattern.at(k) == pattern.at(q))
            k++;

        pi.at(q) = k;
    }

    return pi;
}
```

```
bool KMPMatcher(string text, string pattern)
{
    int m = pattern.size();
    int n = text.size();
    vector<int> pi = ComputePrefixFunction(pattern);
    int q = 0;

    for (int i = 0; i < n; i++)
    {
        while (q > 0 && pattern.at(q) != text.at(i))
            q = pi.at(q - 1);

        if (pattern.at(q) == text.at(i))
            q++;

        if (q == m)
            return true;
    }

    return false;
}
```

Complexidade Temporal: O(|T| + |P|)

Complexidade Espacial: O(|P|)

O algoritmo EditDistance:

```
int editDistance(string pattern, string text)
{
    int pSize = pattern.size();
    int tSize = text.size();
    int oldValue, newValue;
    vector<int> D(tSize + 1);

    for (int j = 0; j <= tSize; j++)
        D.at(j) = j;

    for (int i = 1; i <= pSize; i++)
    {
        oldValue = D.at(0);
        D.at(0) = i;

        for (int j = 1; j <= tSize; j++)
        {
            if (pattern.at(i - 1) == text.at(j - 1))
                newValue = oldValue;
            else
                newValue = 1 + min({oldValue, D.at(j), D.at(j - 1)});

            oldValue = D.at(j);
            D.at(j) = newValue;
        }
    }

    return D.at(tSize);
}
```

Complexidade Temporal: O(|P| · |T|)

Complexidade Espacial: O(|T|)

3.8.4.2. Análise Experimental

Tal como foi feito anteriormente, foram efetuados alguns testes em três ficheiros com tamanho diferente do grafo para avaliar, de forma experimental, os algoritmos implementados.

- Pesquisa exata de uma interseção entre duas ruas.

Ficheiro	Primeira Rua	Segunda Rua	Tempo procura (s)	Resultado
FEUP	Autoestrada de Entre-Douro-e-Minho	Autoestrada de Entre-Douro-e-Minho	0.003	Encontra
Trindade	Rua de Dom Manuel II	Rua de Dom Manuel II	0.004	Encontra
Baixa Porto	Rua do Moreira	Rua da Alegria	0.051	Não encontra

- Pesquisa aproximada de uma interseção entre ruas.

De lembrar que neste caso, as ruas não têm necessariamente de se intersetar: o programa procura ruas 'similares' às dadas, e apresenta aquelas que contêm *SharingLocations*

Ficheiro	Primeira Rua	Segunda Rua	Tempo procura (s)	Resultado
FEUP	Autoestrada de Entre-Douro-e-Minho	Autoestrada de Entre-Douro-e-Minho	1.892	Encontra
Trindade	Rua de Dom Manuel II	Rua de Dom Manuel II	6.434	Encontra
Baixa Porto	Rua do Moreira	Rua da Alegria	0.543	Encontra

3.8.4.3.Complexidade em relação aos dados de entrada

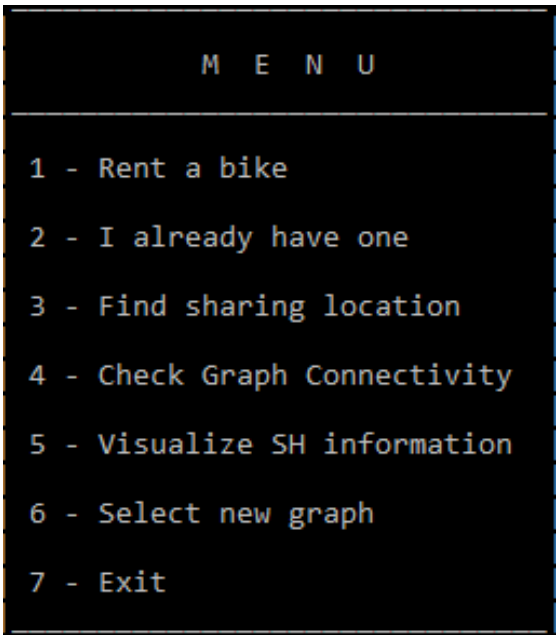
Ambos os tipos de pesquisa percorrem todas as arestas do grafo (enquanto não se estabelecerem as condições de saída) e aplicam sobre os seus nomes os respetivos algoritmos. Portanto, pode-se afirmar que a complexidade temporal é O(|V|.|E|), onde V representa o número de vértices no grafo, e E o número médio de arestas adjacentes por vértice. Quanto à complexidade espacial, na pesquisa exata não é utilizada qualquer memória adicional, portanto é O(1). Na pesquisa aproximada, a memória auxiliar (e retornada pela função) é dependente do número de ruas encontradas.

4. Utilização do Programa e suas funcionalidades

Ao iniciar o programa, uma linha de comandos é executada dando ao utilizador a possibilidade de escolher um dos diferentes mapas disponíveis. Depois de selecionado um mapa, a informação proveniente dos ficheiros de texto correspondentes é carregada para o programa e todas as estruturas de dados são inicializadas com esta.

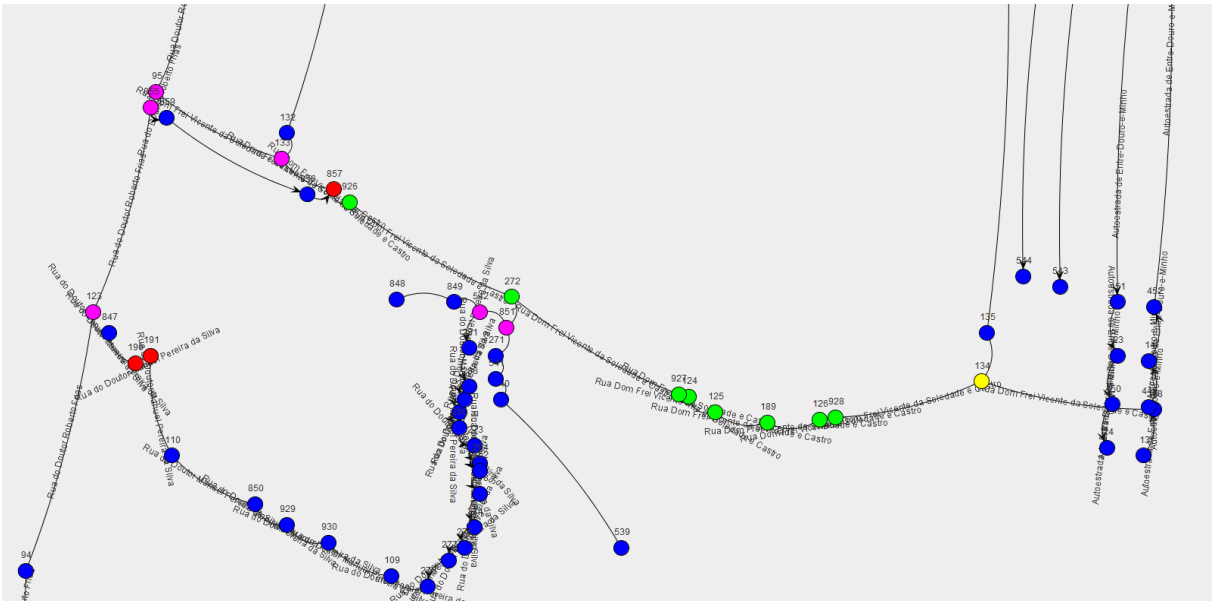
Depois deste passo é mostrado ao utilizador o menu principal com as diferentes opções. É também lançada uma janela gráfica que tem presente toda a informação presente no programa de uma forma mais prática.

O menu apresenta as seguintes opções:



Rent a bike: Se o utilizador selecionar esta opção ser-lhe-á pedido que introduza o local onde se encontra atualmente. Depois disto será apresentado ao utilizador o caminho a tomar até ao local mais próximo onde pode levantar uma bicicleta.

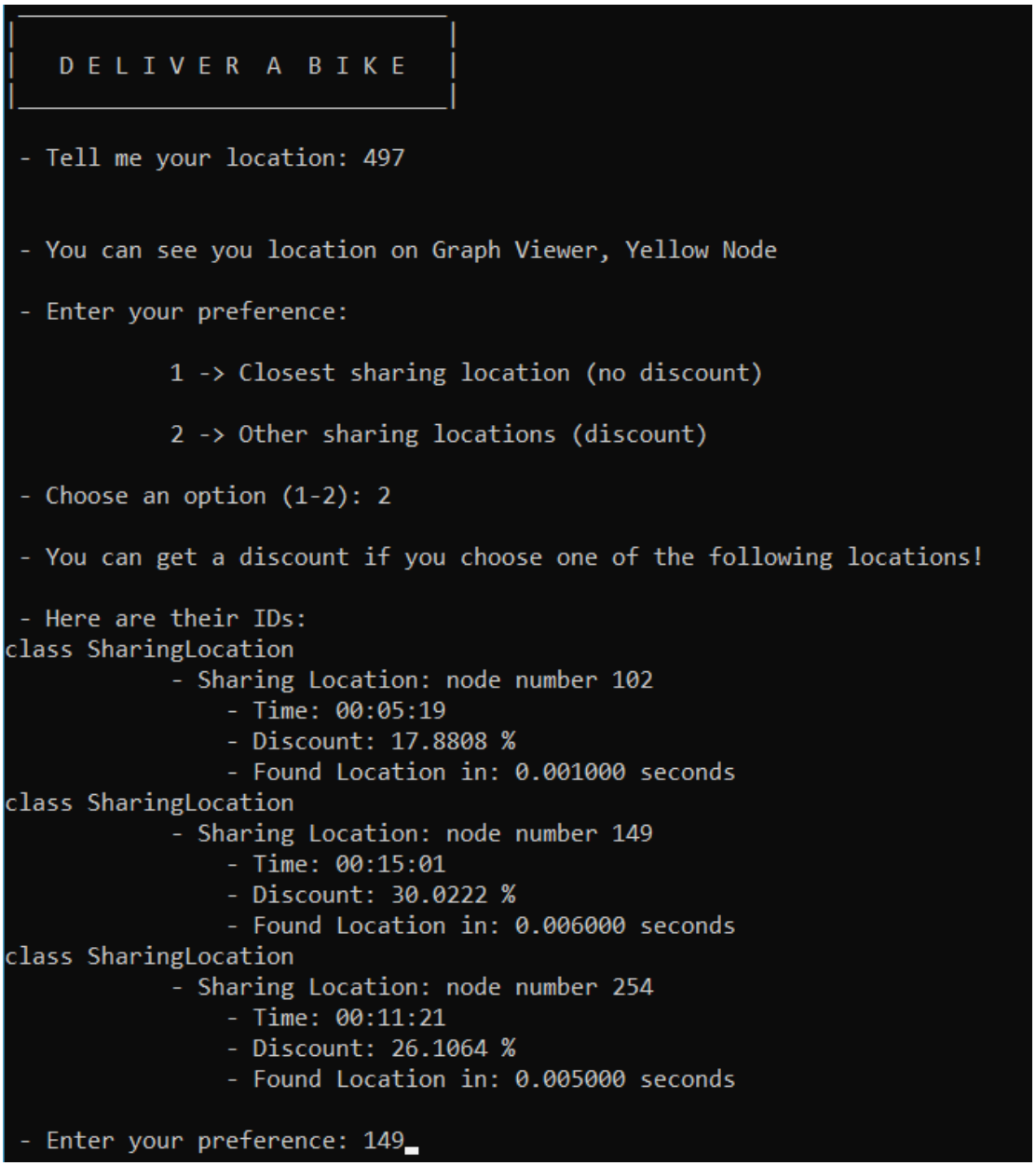
```
 _____________________________
|                             |
|    R E N T    A    B I K E  |
|_____________________________|

- Tell me your location: 134

- You can see you location on Graph Viewer (Red if Sharing Location, Yellow otherwise)

- Enter your preference:

          1 -> Closest sharing location (no discount)

          2 -> Other sharing locations (discount)

- Choose an option (1-2): 1

- Found closest location in: 0.001000 seconds!

- Travel will take you approximately 00:01:57

- Check the map to see the generated path!

       - Yellow Node is the given position

       - Green Nodes are the path to destiny

       - Red Node right next to Green Node is your destiny

       - You have lifted a bike!
```

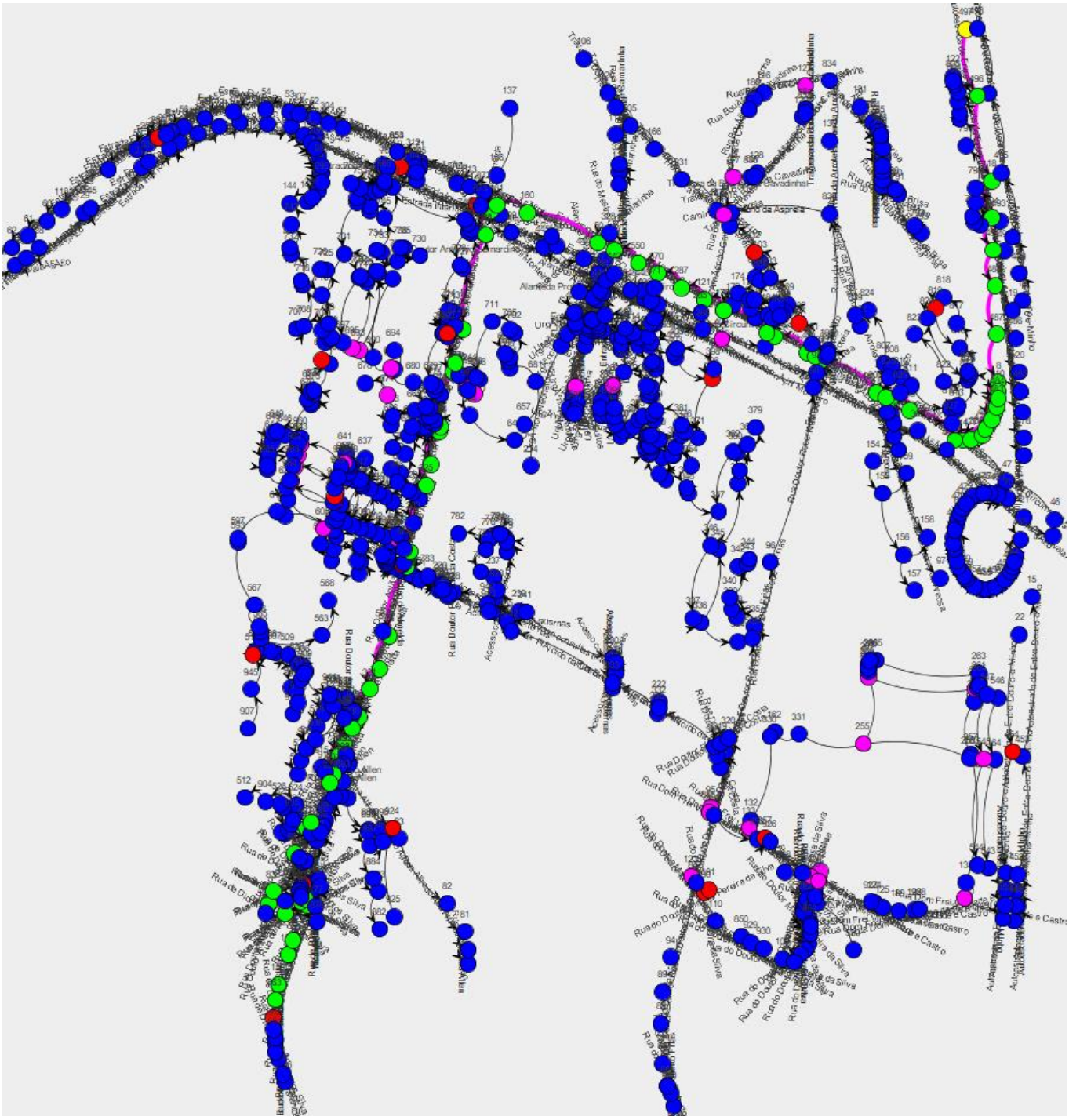


O caminho é representado pela origem (nó amarelo), os nós intermédios (nós verdes) e o destino (nó vermelho).

Deliver a bike: Ao selecionar esta opção, tal como na anterior, o utilizador terá de introduzir o local onde se encontra, sendo de seguida apresentado no ecrã:

- O caminho mais curto a tomar pelo utilizador até ao ponto de recolha mais próximo.
- Uma sugestão que leva o utilizador a um local mais remoto podendo usufruir de um desconto no pagamento do serviço de bicicletas. Esta sugestão vai prevenir que este tipo de locais fique sem bicicletas por serem mais isolados, mantendo assim o equilíbrio no grafo.



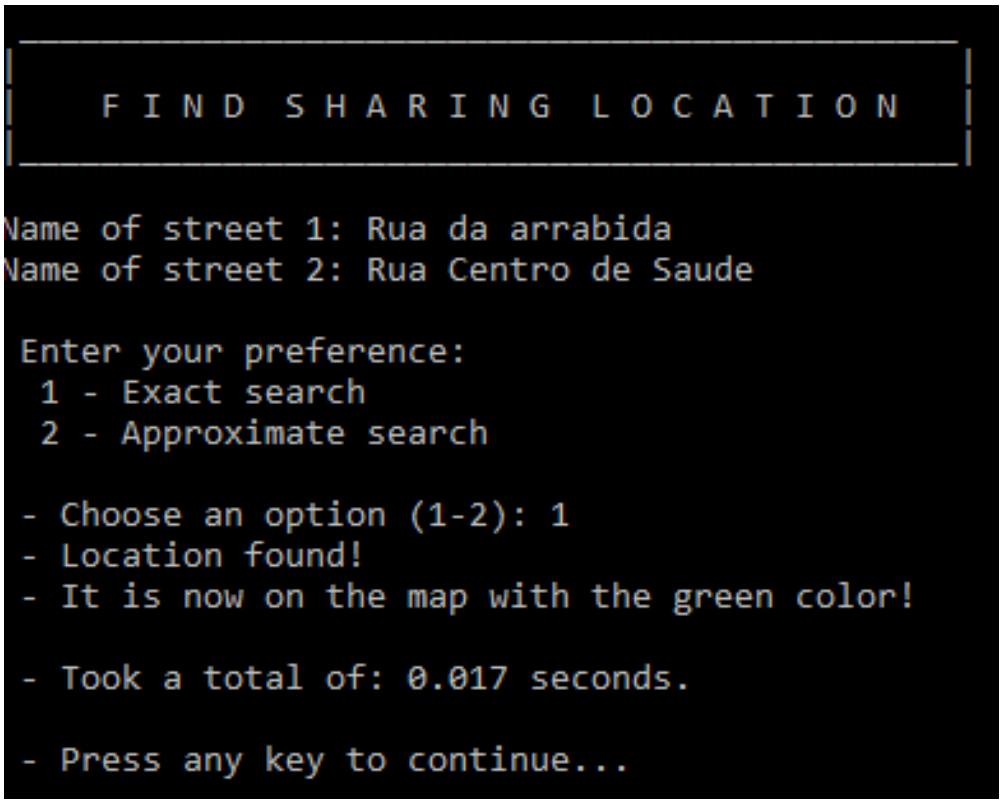


Este grafo representa o caminho desde o nó 497 (nó amarelo) ao nó 149 (nó vermelho final). O caminho está representado a rosa. Tal como foi possível observar no primeiro grafo, as arestas bidirecionais não estão coloridas, pois a nossa implementação não o permite. De modo a tornar o caminho mais percetível decidimos colorir os nós intermédios (nós verdes).

Find Sharing Location: Uma vez selecionada esta opção, vai ser pedido ao utilizador para introduzir o nome de duas ruas. De seguida, o utilizador pode escolher entre dois tipos de pesquisa:

- Exata: Caso o utilizador selecione esta opção o programa irá procurar por uma Sharing Location que se encontre num cruzamento entre as ruas inseridas. Caso não exista uma ser-lhe-á mostrado no ecrã uma mensagem de lugar desconhecido.

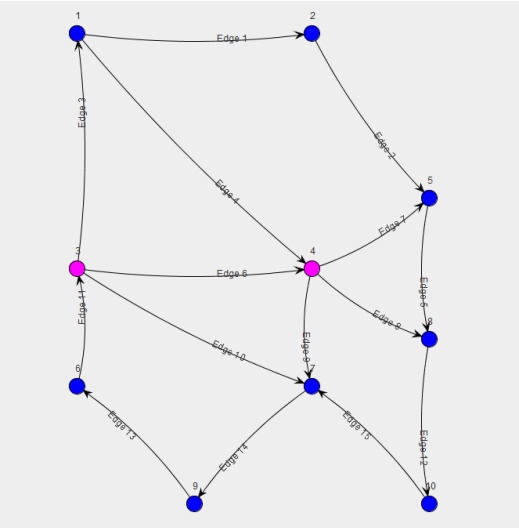
- Aproximada: Caso o utilizador selecione esta opção o programa irá retornar os nomes de ruas mais próximos, ordenados por similaridade, onde poderá haver uma SharingLocation.



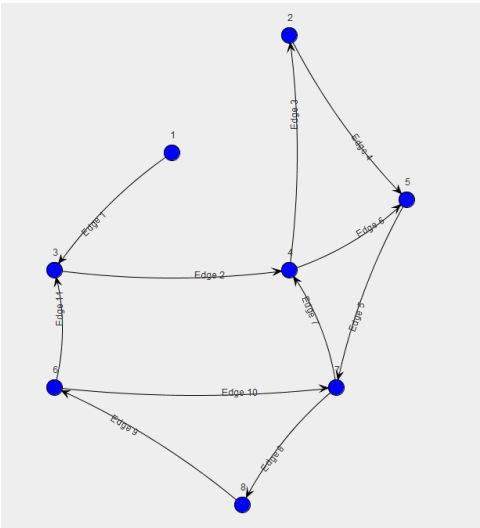
Select a new graph: Esta opção vai redirecionar o utilizador para o ecrã inicial de seleção de um mapa.

```
                                W E L C O M E !

  Choose the Map you want :

      1 - Baixa Porto           -  763 Nodes,  799 Edges, 20 Sharing Locations

      2 - Hospital S.Joao/FEUP  -  969 Nodes, 1051 Edges, 20 Sharing Locations

      3 - Porto Paranhos        - 8670 Nodes, 9278 Edges, 50 Sharing Locations

      4 - Canidelo Gaia         - 1428 Nodes, 1513 Edges, 30 Sharing Locations

      5 - Foz Douro Porto       - 2264 Nodes, 2482 Edges, 30 Sharing Locations

      6 - Trindade, S.Bento     - 4235 Nodes, 4577 Edges, 30 Sharing Locations

  Test files:

      7 - Non Connective Graph -    8 Nodes,   11 Edges   0 Sharing Locations

      8 - Connective Graph     -   10 Nodes,   16 Edges,  0 Sharing Locations

- Choose 0 to Quit.

- Choose an option (0-8): 2
```

Check Graph Connectivity: Verifica se o grafo é conexo ou não. Caso não seja assinala na janela gráfica o nó em que o teste falhou.



Grafo Conexo



Grafo Não Conexo

Exit: Encerra o programa.

5. Principais dificuldades

No desenvolvimento deste projeto foram surgindo algumas dificuldades. A primeira surgiu logo no início com a leitura dos ficheiros. Uma vez que começámos a desenvolver o projeto relativamente cedo, o parser que nos foi fornecido ainda não estava disponível. Desta forma passámos algum tempo a pesquisar formas de conseguir converter a informação presente nos ficheiros do Open Street Maps para ficheiros de texto. Após alguma pesquisa conseguimos obter um parser fornecido em anos anteriores. No entanto como o parser acabou por ser fornecido mais tarde o nosso esforço apenas se provou como desperdício de tempo o que colocou alguma pressão sobre o resto do projeto.

Uma outra dificuldade foi a gestão de tempo. Na nossa opinião o tempo estipulado pelos docentes da Unidade Curricular para o desenvolvimento deste projeto foi relativamente curto. Dado que este não era o único projeto que estava a ser desenvolvido no âmbito do curso, usufruir do tempo disponível para o desenvolvimento, com sucesso, de todos os projetos, tornou-se muito difícil de alcançar.

As últimas dificuldades encontradas estão mais relacionadas com a nossa tentativa de conceção de novas estratégias pedidas pelo tema do projeto. Mais especificamente estratégias como atribuição dos descontos, implementação do incentivo e conciliar a elevação como peso de uma aresta.

Na segunda parte do trabalho, as dificuldades basearam-se na existência de várias ruas com o mesmo nome, que forçou a alteração do algoritmo utilizado para a pesquisa de strings no grafo.

Em suma, apesar do aparecimento destas dificuldades, em geral elas foram ultrapassadas com sucesso e empenho por parte de todos os elementos o grupo.

6. Distribuição do Trabalho

Este projeto foi maioritariamente desenvolvido em conjunto, tendo sido realizadas reuniões sempre que possível. Nestas, cada elemento do grupo reportava aos restantes o seu progresso e eram também discutidas diferentes estratégias de maneira a atingir os objetivos do trabalho a que nos propusemos.

Todos os elementos do grupo contribuíram e empenharam-se de igual forma para o bom desenvolvimento e funcionamento deste projeto.

7. Conclusão

A realização deste trabalho permitiu nos obter uma melhor compreensão da matéria em questão, particularmente do modo de funcionamento de algoritmos de pesquisa em grafos, nomeadamente o Algoritmo de Dijkstra.

A proposta de trabalho continha um intuito educativo, sendo requerido da nossa parte que compreendêssemos a usássemos não só novas estruturas como grafos, mas também algoritmos de pesquisa nos mesmos.

Concluímos, portanto, que os objetivos pretendidos com este projeto de grupo foram atingidos, quer a nível individual quer a nível coletivo, uma vez que cada elemento domina agora os temas lecionados na unidade curricular e é capaz de os aplicar numa componente prática.