Concepção e Análise de Algoritmos (CAL)
Turma 7, Grupo 5
2019/2020

TourMateApp
Rotas Turísticas Urbanas Adaptáveis

João Carlos Machado Rocha Pires (up201806079)
João Oliveira da Rocha (up201708566)
Pedro Alexandre Vieites Mendes (up201704219)

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

24-04-2020

Conteúdo

1. Descrição do Tema

O nosso tema está relacionado com Rotas Turísticas Urbanas Adaptáveis, ou seja, de forma sucinta, a aplicação que criaremos permitirá ao utilizador obter uma sugestão de uma rota turística urbana que se adaptará às limitações e exigências do mesmo.

Seja em passeio ou em trabalho, os turistas terão diferentes interesses e disponibilidades para visitar as atracções turísticas de uma cidade. Em viagens curtas de trabalho, podem ter o interesse de realizar visitas de muito curta duração, tendo a nossa aplicação de identificar circuitos mais curtos que lhes levem aos pontos de interesse mais importantes, segundo as suas preferências. Turistas em viagem de lazer já terão potencialmente flexibilidade de horários e maior disponibilidade para itinerários mais longos que incluirão mais pontos de atracção.

Neste projecto, pretendemos, portanto, implementar a aplicação *TourMateApp*, que permite a construção de itinerários turísticos adaptáveis às preferências e disponibilidades do utilizador.

A aplicação deverá manter uma lista dos pontos de interesse turísticos e ser capaz de sugerir itinerários que incluam as atracções mais adequadas ao perfil de cada utilizador. Esses mesmos itinerários deverão poder ser realizados no tempo indicado desde a origem até ao destino final, sendo que as condicionantes serão indicadas pelo utilizador.

Partindo de um exemplo, suponhamos que um determinado utilizador viaja em trabalho e deseja realizar uma caminhada de 30 minutos, num intervalo de tempo livre, próxima do local onde está a ter reuniões. A aplicação deverá sugeri-lhe um itinerário capaz de ser realizado nesse período de tempo e que inclua o maior número de pontos turísticos localizados nas proximidades do local onde se encontra em trabalho, minimizando assim o tempo de deslocação e aumentando o tempo de visita a essas atracções.

Este foi apenas um de muitos exemplos. Um turista que deseje fazer um circuito a passar pelos principais museus da cidade, numa visita de um dia completo, poderá ser um outro exemplo.

As recomendações deverão maximizar o número visitas a atracções turísticas de acordo com o perfil e preferências do utilizador, o tempo disponível, e o seu meio de deslocamento: automóvel ou caminhando.

2. Identificação e Formalização do Problema

Nas próximas secções, definir-se-ão os dados de entrada e de saída esperados, bem como as restrições a esses mesmos dados e a função objectivo do trabalho prático.

2.1 Dados de entrada

Como dados de entrada para o cálculo da Rota Turística Urbana Adaptável, deverão ser fornecidos os seguintes dados:

- **Identificação do Cliente/Utilizador** - nome do cliente. O único propósito deste dado será fornecer uma solução personalizada. Não tem qualquer impacto no funcionamento da aplicação.
- **Preferências do Cliente/Utilizador** - dados relativos às preferências do utilizador, como são exemplo as áreas de interesse (Estátuas, Museus, Parques, etc.), que permitirão personalizar as rotas sugeridas de forma a torná-las mais pessoais e adaptadas aos gostos do cliente.
- **Informações da viagem que efectua** - dados relativos ao tipo de viagem, tais como a duração (em minutos), a localização inicial e o tempo disponível.
- **Descrição da rede da cidade** - dados relativos ao mapa da cidade em questão, permitindo assim identificar quais as ligações (ruas, praças, avenidas, etc.) que permitem chegar a determinado endereço, bem como saber o tempo estimado que a deslocação comportará.
- **Pontos turísticos** - dados relativos aos pontos turísticos, como a sua localização, tempo médio de visita, área de interesse (relacionado com a área de interesse do utilizador) e classificação / *rating* atribuída pelos utilizadores que a visitam (a classificação será introduzida assim que o ponto de interesse (POI) for registado, usando um número aleatório entre 1 e 5, e assumirá esse valor daí em diante).
- **Transportes** - De entre os transportes disponíveis (deslocação a pé ou de carro), o utilizador deverá indicar qual o seu preferido de modo a aumentar as probabilidades desse transporte ser o responsável pelas deslocações no percurso que lhe for sugerido.

2.2 Dados de saída

Como dados de saída do programa, deverá ser mostrado, recorrendo à API *GraphViewer*, o mapa da cidade do Porto com a informação do percurso. Deverá também ser gerado um *output* que permita ver o percurso a ser percorrido através de coordenadas. Mais informação sobre o tipo de dados de saída na secção 4.

Desta forma, o cálculo do melhor percurso em função das condicionantes mencionadas na secção 2.1 ficará a cargo da função objectivo, descrita na secção 2.4.

Tais dados deverão estar organizados de forma a que a aplicação possa aceder aos mesmos e assim apresentá-los ao utilizador de forma graficamente amigável.

2.3 Restrições

As restrições que o programa deve ter em conta para que a informação apresentada ao utilizador seja a correcta deverão ser as seguintes:

- **Preferências válidas e correspondentes aos tipos de pontos turísticos** - as preferências de cada utilizador deverão coincidir com as áreas de interesse e com os tipos de POI disponíveis na cidade em causa. Isto é, se o utilizador manifestar a sua preferência por museus e/ou monumentos históricos, a aplicação deverá ter em consideração essas mesmas preferências e restringir os POI incluídos no percurso que sugere ao utilizador a estas mesmas condições. Se as preferências do utilizador não puderem ser incluídas no percurso, ou porque simplesmente a cidade não inclui tais tipos de POI ou por outro motivo qualquer, a aplicação deverá sugerir um percurso baseado apenas na deslocação por ruas da cidade em causa.

- **Condições da viagem válidas** - o tipo de viagem deverá ser descrito em minutos. A localização inicial deverá ser obtida perguntando em que coordenadas o utilizador se encontra. Após o *input* das coordenadas, a nossa aplicação tentará encontrar o nó mais próximo. Caso o utilizador insira coordenadas fora do alcance, este será novamente questionado para inserir valores até que correspondam a valores compreendidos entres os limites do mapa.

- **Mapa da cidade actualizado e completo** - o mapa da cidade deverá estar actualizado e completo, a fim de fornecer informação actual e fidedigna. O tempo estimado de deslocamento (peso de cada Aresta) será calculado através da Fórmula de *Haversine*: uma fórmula para calcular distâncias entre dois pares de coordenadas, tendo em conta o raio da Terra. A aplicação também fará uma verificação nas Arestas de um ponto para si mesmo, evitando assim quaisquer erros de processamento redundante.

- **Pontos Turísticos disponíveis para visita** - evitar que sejam escolhidos pontos turísticos incompatíveis com os Tipos de Pontos de Interesse escolhidos pelo utilizador.

2.4 Função objectivo

A função objectivo deste trabalho é maximizar o número de pontos turísticos incluídos na sugestão fornecida ao utilizador, na medida em que esses mesmos POI deverão corresponder, sempre que possível, ao maior número de edifícios cujas características se assemelhem às preferências do utilizador.

3. Perspectiva de Solução

A aplicação deverá apresentar um itinerário em forma de lista de pontos de interesse, sendo o primeiro o ponto de partida. Todos os pontos deverão formar um percurso que inclua o maior número de Pontos de Interesse a visitar no tempo útil escolhido pelo utilizador.

A lista a apresentar dependerá dos parâmetros escolhidos pelo utilizador. Este deverá receber uma sugestão adaptada a critérios como, por exemplo, o tempo disponível para o percurso e a escolha entre visitar apenas museus ou parques.

Para chegar à solução, usaremos um algoritmo de *Prim* para pré-processar o grafo. Ou seja, um algoritmo ganancioso que cria uma árvore de expansão mínima a partir do ponto de partida, garantindo assim que as distâncias entre o ponto inicial e os outros pontos de interesse são mínimas. Usaremos este algoritmo uma vez que podemos adaptá-lo facilmente e limitar a expansão da árvore consoante o tempo definido pelo utilizador, evitando o processamento desnecessário dos restantes pontos do grafo que, por não cumprirem as condições de satisfação, não serão utilizados para a solução, ao contrário do algoritmo de *Kruskal* que apenas liga arestas, independentemente da sua posição. Ponderamos utilizar o algoritmo de *Kruskal* para a criação da árvore de expansão mínima, mas decidimos utilizar o de *Prim* pois este funciona melhor em grafos densos como um grafo de uma cidade:

$$\theta(|\mathrm{E}|\ \log|\mathrm{V}|) \text{ - } \textit{Prim}$$
$$\theta(|\mathrm{E}|\ \log\ |\mathrm{E}|) \text{ - } \textit{Kruskal}$$
(E - número de arestas ; V - número de vértices)

Quando temos uma grafo denso, o número de arestas (E) é maior que o de vértices (V) e, por isso, teremos

$$\mathrm{E} = \mathrm{V}{+}\mathrm{x}$$
$$(|\mathrm{E}|\ \log|\mathrm{V}|) < (|\mathrm{E}|\ \log|\mathrm{V}{+}\mathrm{x}|)\ (\mathrm{x} > 0)$$

Depois de pré-processar o grafo, usaremos um algoritmo de ordenação topológica, $\theta(|\mathrm{V}| + |\mathrm{E}|)$, para encontrar o percurso mais adequado, tendo em conta não só os pesos das arestas, mas também se compensa gastar tempo em certo ponto de interesse ao invés de passar para outro ponto mais distante na esperança de ewncontra um ponto com melhor *rating*. Por

este motivo, talvez seja melhor descartar um POI com um elevado tempo médio de visita e menor classificação (*rating*) e optar por um outro POI próximo com melhor classificação. Para cada vértice, guardaremos a opção de visitar ou não visitar certo POI, com o objectivo de, depois de concluído o algoritmo, comparar qual a melhor opção. Para tal dilema, sempre que o algoritmo encontrar um POI da preferência do utilizador, ele duplicará esse ponto (nó), copiando as arestas, pesos, etc., e tratará esse ponto como um ponto de passagem, somando apenas os pesos das arestas. Ainda assim, tratará o seu clone como um ponto de passagem, em que soma os *ratings* e junta o tempo médio de visita ao peso desse nó.

Para utilizar o algoritmo, precisamos de representar qualquer mapa como um conjunto de vértices e arestas. Cada vértice representa um Ponto de Interesse e cada aresta o caminho entre dois POI's. Os vértices deverão conter informações, tais como um ID, classificação (*rating*), tipo de ponto (museu, monumento, etc.), tempo aconselhado de visita e coordenadas geométricas / xy.

As arestas representam caminhos entre dois pontos de interesse. Deverão, portanto, conter um ID, informação dos dois pontos da ligação e a distância. O peso de cada aresta será calculado usando a Fórmula de **Haversine**: uma fórmula que utiliza as coordenadas dos dois pontos da aresta e o raio da Terra para obter a distância entre eles.

As próximas ilustrações servem para demonstrar o funcionamento dos algoritmos em conjunto. Usaremos um grafo simples em que cada aresta terá o mesmo peso e o tempo disponível fornecido pelo utilizador será de 3 arestas. Isto significa que o utilizador, numa situação limite, apenas poderá deslocar-se até a um máximo de 3 vértices a partir do vértice inicial. Os tempos médios de visita serão ignorados nesta ilustração.

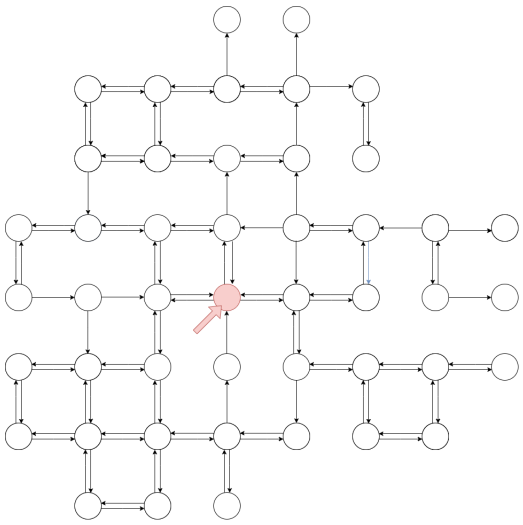

Figura 3.1: Grafo inicial (a vermelho, o ponto de partida).

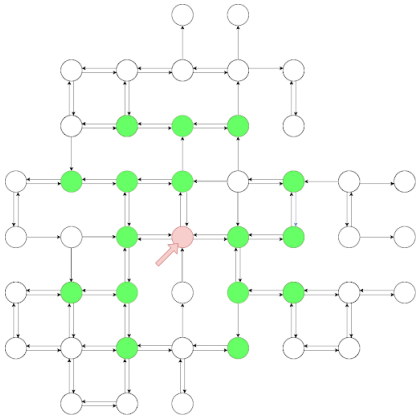

Figura 3.2: Após pré-processamento usando o Algoritmo de *Prim* (a verde, os vértices alcançáveis para o tempo disponível).

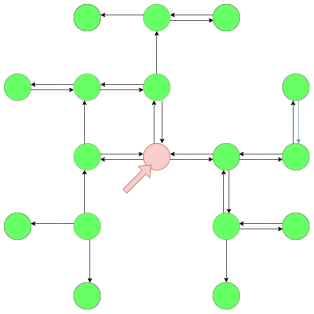

Figura 3.3: O Algoritmo de Ordenação Topológica irá apenas percorrer estes vértices.

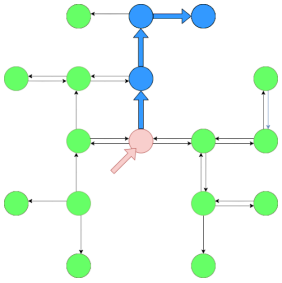

Figura 3.4: Após comparação de todos os caminhos possíveis e escolha da melhor opção.

4. Casos de Utilização Suportados e Respectivas Funcionalidades

O programa começará por pedir ao utilizador para introduzir o seu nome, de forma a poder tratar o mesmo pelo nome.

De seguida, irá iniciar o carregamento dos dados necessários do mapa da cidade do Porto. Este processo inclui o carregamento do conteúdo dos seguintes ficheiros:

- *porto_full_nodes_latlng.txt*
- *porto_full_edges.txt*
- *porto_full_nodes_xy.txt*

Estes ficheiros permitirão "povoar"o grafo utilizado com a informação completa dos *nodes* (em coordenadas XY e Geográficas) e dos *edges*.

De seguida, será apresentado ao utilizador um menu que permitirá ao mesmo escolher os seguintes factores que influenciarão o caminho sugerido:

- Seleccionar a localização inicial
- Definir o tempo disponível
- Escolher o meio de transporte (a pé ou de carro)
- Escolher os POI's favoritos

Assim que estiverem definidos pelo utilizador estes factores, e só nesse caso, é possível calcular assim o caminho que lhe será sugerido.

Após o caminho estar correctamente calculado, será apresentado ao utilizador de forma visual recorrendo à API *GraphViewer*. Esta representação gráfica contempla a seguinte informação:

- Mapa da cidade do Porto com todos os nós e arestas a preto
- Ponto de partida a azul
- Ponto de chegada a verde
- Percurso a vermelho
- POI a visitar a vermelho

Nota: Se a intenção do utilizador for de apenas visitar, por exemplo, POI do tipo "Museus", o utilizador poderá seleccionar apenas essa opção no menu e a aplicação mostrará uma sugestão de um percurso em que o utilizador apenas visitará POI que estejam classificados como "Museus". Pelo percurso, poderão aparecer POI de outros tipos, mas esses estarão representados apenas como nós como qualquer outro e não pertencerão ao itinerário da visita.

A seguinte sequência de imagens permite demonstrar um exemplo de execução.

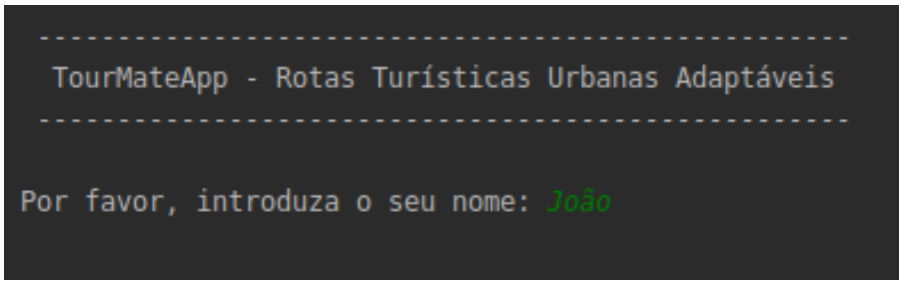

Figura 4.1: Escolha do Nome de Utilizador

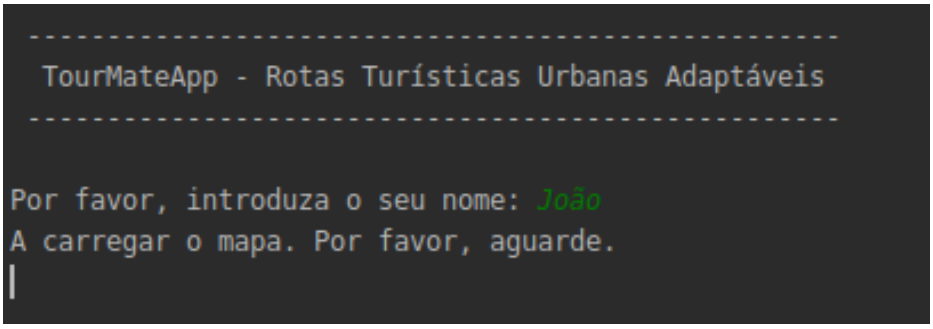

Figura 4.2: Carregamento do Mapa

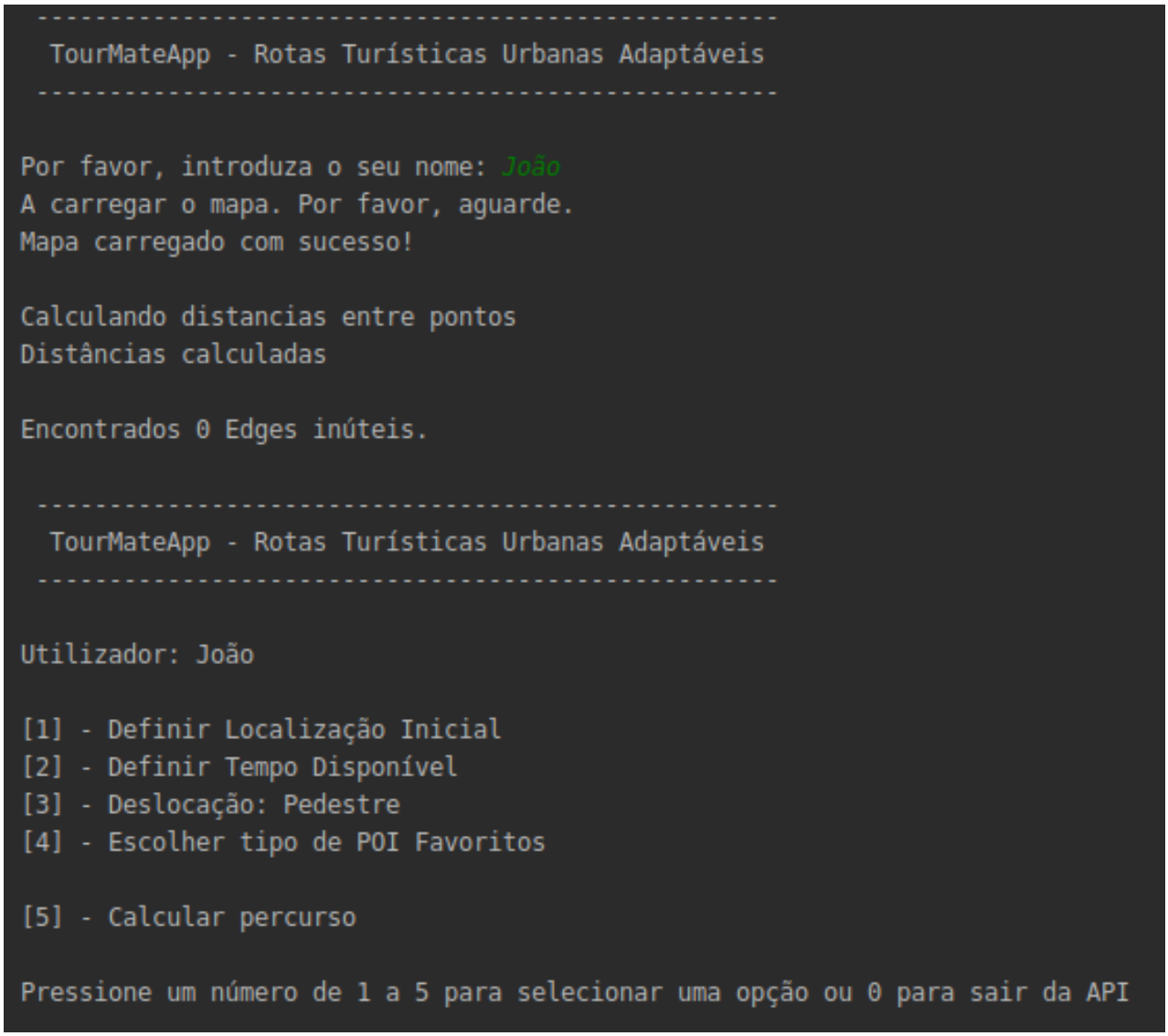

Figura 4.3: Mapa Carregado e Menu Principal

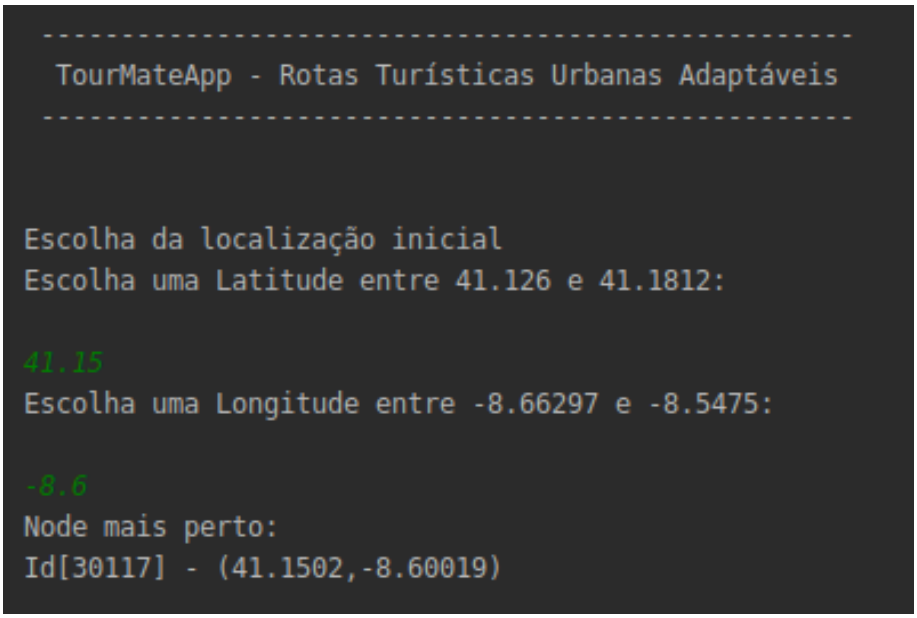

Figura 4.4: Escolha da localização inicial

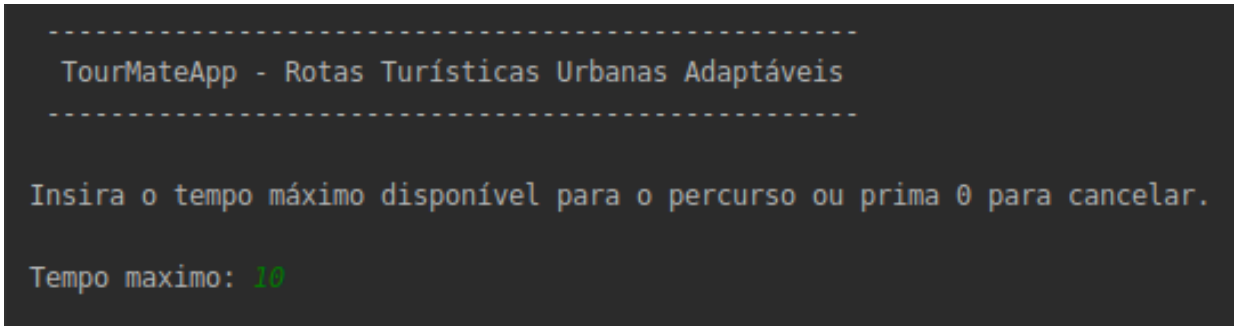

Figura 4.5: Definição do tempo disponível

```
 ---------------------------------------------------
  TourMateApp - Rotas Turísticas Urbanas Adaptáveis
 ---------------------------------------------------

    [1] -  [S] Estatua
    [2] -  [S] Museu
    [3] -  [N] Estadio
    [4] -  [N] Livraria
    [5] -  [S] Palacio
    [6] -  [N] Parque

Escolha os seus POI favoritos.
S - Selecionado ; N - Não Selecionado
Pressione 0 para voltar ao menu principal e 7 para selecionar/desselecionar todos

POI a alterar:
```

Figura 4.6: Escolha dos POI's favoritos

```
 ---------------------------------------------------
  TourMateApp - Rotas Turísticas Urbanas Adaptáveis
 ---------------------------------------------------

Utilizador: João

[1] - Localização Inicial [ Lat: 41.1502; Lon: -8.60019 ]
[2] - Tempo Disponível: [ 10 min ]
[3] - Deslocação: Pedestre
[4] - Alterar tipo de POI Favoritos

[5] - Calcular percurso

Pressione um número de 1 a 5 para selecionar uma opção ou 0 para sair da API
```

Figura 4.7: Menu Principal após *inputs* introduzidos

```
Last Node  | Lat:    41.1518 | Lon:   -8.59788
First Node | Lat:    41.1502 | Lon:   -8.60019

Streets to pass by:

Street | From Lat:    41.1491 | Lon:   -8.59878 to Lat:    41.1491 | Lon:   -8.59878
Street | From Lat:     41.149 | Lon:   -8.59893 to Lat:     41.149 | Lon:   -8.59893
Street | From Lat:     41.149 | Lon:   -8.59871 to Lat:     41.149 | Lon:   -8.59871
Street | From Lat:    41.1514 | Lon:   -8.59753 to Lat:    41.1514 | Lon:   -8.59753
Street | From Lat:    41.1493 | Lon:   -8.59936 to Lat:    41.1493 | Lon:   -8.59936
Street | From Lat:    41.1494 | Lon:   -8.60036 to Lat:    41.1494 | Lon:   -8.60036
Street | From Lat:    41.1494 | Lon:   -8.60032 to Lat:    41.1494 | Lon:   -8.60032
Street | From Lat:    41.1491 | Lon:    -8.5991 to Lat:    41.1491 | Lon:    -8.5991
Street | From Lat:    41.1491 | Lon:   -8.59878 to Lat:    41.1491 | Lon:   -8.59878
Street | From Lat:    41.1502 | Lon:    -8.6003 to Lat:    41.1502 | Lon:    -8.6003
Street | From Lat:      41.15 | Lon:   -8.59827 to Lat:      41.15 | Lon:   -8.59827
Street | From Lat:    41.1517 | Lon:   -8.59758 to Lat:    41.1517 | Lon:   -8.59758
Street | From Lat:    41.1489 | Lon:   -8.59886 to Lat:    41.1489 | Lon:   -8.59886
Street | From Lat:    41.1494 | Lon:    -8.6004 to Lat:    41.1494 | Lon:    -8.6004
Street | From Lat:    41.1491 | Lon:   -8.59877 to Lat:    41.1491 | Lon:   -8.59877
Street | From Lat:    41.1494 | Lon:   -8.59957 to Lat:    41.1494 | Lon:   -8.59957
Street | From Lat:    41.1501 | Lon:   -8.59822 to Lat:    41.1501 | Lon:   -8.59822
Street | From Lat:    41.1494 | Lon:   -8.59976 to Lat:    41.1494 | Lon:   -8.59976
Street | From Lat:    41.1514 | Lon:    -8.5975 to Lat:    41.1514 | Lon:    -8.5975
Street | From Lat:    41.1494 | Lon:   -8.59951 to Lat:    41.1494 | Lon:   -8.59951
Street | From Lat:    41.1517 | Lon:   -8.59754 to Lat:    41.1517 | Lon:   -8.59754
Street | From Lat:    41.1504 | Lon:   -8.59804 to Lat:    41.1504 | Lon:   -8.59804
Street | From Lat:    41.1517 | Lon:   -8.59754 to Lat:    41.1517 | Lon:   -8.59754
Street | From Lat:    41.1517 | Lon:   -8.59756 to Lat:    41.1517 | Lon:   -8.59756
Street | From Lat:    41.1494 | Lon:   -8.60044 to Lat:    41.1494 | Lon:   -8.60044
Street | From Lat:    41.1491 | Lon:   -8.59878 to Lat:    41.1491 | Lon:   -8.59878
Street | From Lat:    41.1494 | Lon:   -8.60042 to Lat:    41.1494 | Lon:   -8.60042
Street | From Lat:    41.1515 | Lon:   -8.59755 to Lat:    41.1515 | Lon:   -8.59755
Street | From Lat:    41.1492 | Lon:   -8.59876 to Lat:    41.1492 | Lon:   -8.59876
Street | From Lat:    41.1516 | Lon:   -8.59754 to Lat:    41.1516 | Lon:   -8.59754
Street | From Lat:    41.1518 | Lon:   -8.59783 to Lat:    41.1518 | Lon:   -8.59783
Street | From Lat:    41.1514 | Lon:   -8.59754 to Lat:    41.1514 | Lon:   -8.59754
Street | From Lat:    41.1494 | Lon:   -8.59967 to Lat:    41.1494 | Lon:   -8.59967
Street | From Lat:    41.1491 | Lon:   -8.59906 to Lat:    41.1491 | Lon:   -8.59906
Street | From Lat:    41.1496 | Lon:   -8.59849 to Lat:    41.1496 | Lon:   -8.59849
Street | From Lat:    41.1502 | Lon:   -8.60023 to Lat:    41.1502 | Lon:   -8.60023
Street | From Lat:    41.1491 | Lon:   -8.59877 to Lat:    41.1491 | Lon:   -8.59877
Street | From Lat:    41.1514 | Lon:   -8.59755 to Lat:    41.1514 | Lon:   -8.59755
Street | From Lat:    41.1518 | Lon:   -8.59768 to Lat:    41.1518 | Lon:   -8.59768
Street | From Lat:    41.1516 | Lon:   -8.59753 to Lat:    41.1516 | Lon:   -8.59753
Street | From Lat:    41.1494 | Lon:   -8.59962 to Lat:    41.1494 | Lon:   -8.59962
Street | From Lat:    41.1505 | Lon:   -8.59799 to Lat:    41.1505 | Lon:   -8.59799
Street | From Lat:    41.1502 | Lon:   -8.60019 to Lat:    41.1502 | Lon:   -8.60019
```

Figura 4.8: *Output* dos Nós inicial e final e das ruas a percorrer

```
POI's to visit:

POI | Lat:    41.1517 | Lon:   -8.59758
POI | Lat:    41.1502 | Lon:   -8.60023
POI | Lat:    41.1494 | Lon:   -8.59951
```

Figura 4.9: *Output* dos POI a visitar

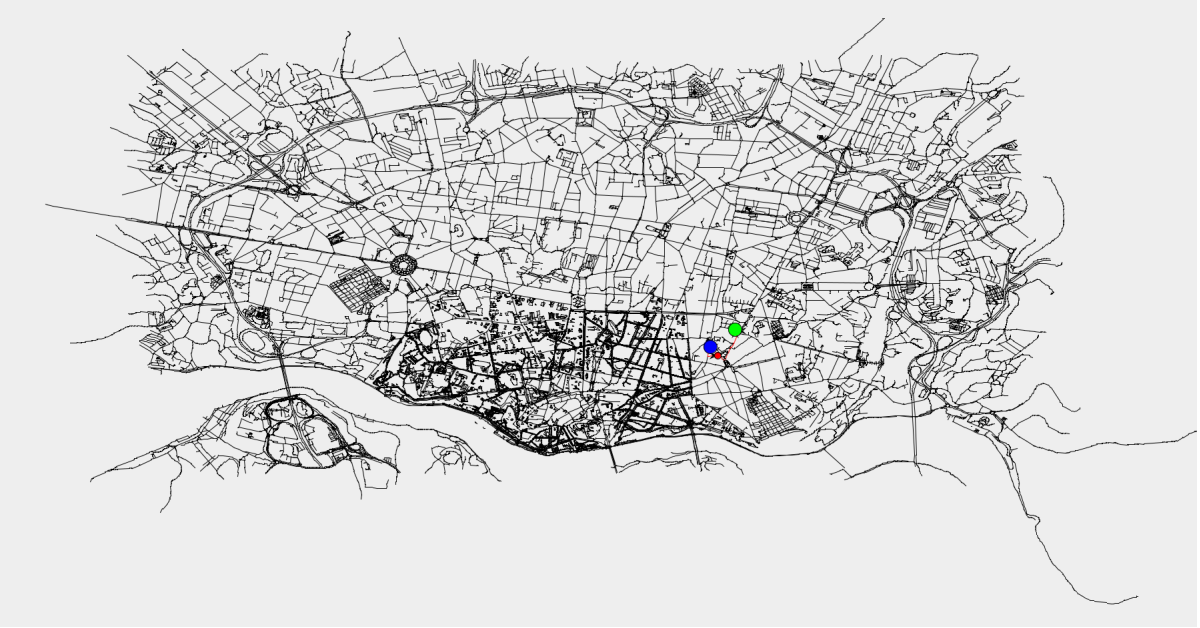

Figura 4.10: Visualização gráfica do percurso

5. Estruturas de dados utilizadas

- ***Menu States***

  Usamos um *enum* para representar todos os tipos de estados que o menu do nosso programa pode ter.

```
    enum MENUSTATES {
        START,
        MAINMENU,
        CHOOSINGLOCATION,
        CHOOSINGTIME,
        CHOOSINGTYPESPOI,
        CALCULATING,
        CALCULATED,
        DONE,
    };

```

- ***POI Types***

  Usamos um *enum* para os tipos de POI que o nosso programa contempla.

```
    enum POITYPE {
        NOT_INTERESTING
        ESTATUA,
        MUSEU,
        ESTADIO,
        LIVRARIA,
        PALACIO,
        PARQUE
    };

```

- ***POI Types - Node version***

  Usamos uma *struct* para poder atribuir a cada nó o tipo de POI que representa, se se aplicar.

```
    struct Tourism {
        int estatua;
        int museu;
```

```
        int estadio;
        int livraria;
        int palacio;
        int parque;
    };

```

- ***Node***

  Para representar os nós no nosso programa, e a fim de incluir toda a informação necessária, definimos a seguinte classe *Node*:

```
class Node {

private:

    int id;                       // Node ID
    double lat, lon;              // Geographical Coordinates
    double x, y;                  // XY Coordinates
    std::vector<Edge *> adj;      // Connected Edges
    struct Tourism tags;          // Enum for POI type
    bool visited;                 // If can be visited or not
    bool mostEfficient = false; // If it's visited or not
    bool lastNode = false;        // End point of track
    double dist = 0;              // Distance to start node
    double pastWeight = 0;
    Node *path = nullptr;         // Next node of path
    int rating;                   // Rating of POI (if applied)
    int visitTime;                // Visit Time of POI (if applied)
    int pastRating = 0;

public:

    int queueIndex = 0;      // Required by MutablePriorityQueue

    Node();
    Node(int id, double lat, double lon, double x, double y);

    bool isLastNode() const;
    void setLastNode(bool lastNode);

    int getId() const ;
    double getLat() const ;
    double getLon() const ;
    double getX() const ;
    double getY() const ;

    double getPastWeight() const;
    void setPastWeight(double pastWeight);

```

```
39      Node *getPath() const;
40      void setPath(Node * path);
41
42      bool isMostEfficient() const;
43      void setMostEfficient(bool mostEfficient);
44
45      void setId(int id);
46      void setCoords(double lan, double lon);
47      void setXY(double x, double y);
48
49      void printInfo();
50
51      void addAdjEdge(Edge * edge);
52      vector<Edge *> getAdjEdge();
53
54      bool operator== (Node other_node);
55
56      void addTag(string type);
57      struct Tourism getNodeTags() const;
58
59      double calcDist(double lat, double lon);
60      void setDist(int dist);
61      double getDist() {return this->dist;}
62
63      void setVisited(bool visited);
64      bool getVisited() {return this->visited;}
65
66      bool operator<(Node & node) const {
67          return this->dist < node.getDist();
68      } // required by MutablePriorityQueue
69
70      void setRating(int rating);
71      int getRating();
72
73      void setVisitTime(int time);
74      int getVisitTime();
75
76  };
77
```

- ***Edge***

  Para representar as arestas no nosso programa, e a fim de incluir toda a informação necessária, definimos a seguinte classe *Edge*:

```
1   class Edge {
2
3   private:
4
5       Node* orig;      // Origin Node of Edge
```

```
    Node* dest;      // Destination Node of Edge
    double weight;  // Weight of Edge
    double flow;
    Edge(Node* o, Node* d, double w, double f);
    Edge() {};

public:

    double getFlow() const;

    double getWeight() const;
    void setWeight(double weight);

    Node* getDest() const;

    Node* getOrig() const;

    friend class Graph;
    friend class Node;
};

```

- ***Graph***

  Para representar o grafo que será objecto de modificações no nosso programa, e a fim de incluir toda a informação necessária, definimos a seguinte classe *Graph*:

```
class Graph {

private:

    std::vector<Node *> nodes;  // Nodes do Grafo
    std::vector<Edge> edges;    // Edges do Grafo

public:

    Graph();

    Node * findNode(double id);
    void printNodeInfo();
    void addNode(int id, double lat, double lon, double x, double y);
    vector<Node *> getNodes();

    Edge findEdge(Node* orig, Node* dest);
    void printEdgeInfo();
    Edge * addEdge(Node* orig, Node* dest);
    vector<Edge>* getEdges();

    void addTag(int id, string type);

```

```
24      Node getInitLoc(double lat, double lon);
25
26      vector<Node *> calculatePrim(Node* start, int timeAv);
27
28      void setNodes(vector<Node *> newNodes);
29
30      void addDuplicateEdges(Node * dest);
31
32      void addVisitedEdges(Node * dest);
33
34      void addLeavingEdges(Node * anterior, Node * novo);
35
36      void userPreferences(struct Tourism userPref);
37  };
38
```

6. Algoritmos Implementados

Como referimos anteriormente, a nossa implementação do Algoritmo de *Prim* terá um critério de paragem que será fornecido pelo utilizador. Como prevemos que não seja necessário, na maioria das vezes, percorrer todo o grafo em buscar da árvore de expansão mínima total, inserimos um critério de paragem chamado "Tempo Disponível", que, se o Algoritmo verificar que a distância de um nó à origem for superior ao tempo disponível, não irá proceder à análise desse nó nem de qualquer nó que proceda esse, "cortando"a extensão da árvore.

A decisão de utilização deste critério deve-se à desnecessidade de obtenção de uma árvore de expansão mínima aplicada a todo o grafo, pois na maioria dos casos de utilização previstos o tempo disponível fornecido pelo utilizador não será suficiente para passar por todos os pontos. Ou seja, como o utilizador não terá tempo para visitar quase todos os pontos de interesse existentes na cidade então não precisaremos de saber a distância mínima de todos os pontos à origem: interessam apenas aqueles a que o utilizador poderá chegar.

6.1 Algoritmo de *Prim*

Para testar a implementação do Algoritmo de *Prim*, aplicar o algoritmo a um ponto e ir variando o seu tempo disponível. Seguem-se os resultados do algoritmo, tendo como ponto inicial a Rotunda da Boavista (coordenadas: 41.158132, -8.629149).

Figura 6.1: Tempo Disponível: 15 minutos

Figura 6.2: Tempo Disponível: 30 minutos

Figura 6.3: Tempo Disponível: 45 minutos

Figura 6.4: Tempo Disponível: 60 minutos

Podemos ver a vermelho as arestas que ligam os pontos alcançáveis pelo Algoritmo de *Prim*. Como seria de esperar, um menor tempo disponível levará a uma passagem por menos pontos do grafo.

Para a nossa aplicação, o Algoritmo de *Prim* permite conhecer o alcance máximo possível de pontos dado um determinado tempo. Mesmo que o utilizador decida não gastar tempo a visitar nenhum Ponto de Interesse, não conseguirá alcançar pontos que não sejam os pontos cobertos pelo algoritmo.

Segue-se o pseudo-código relativo ao Algoritmo de *Prim* implementado:

```
calculatePrim(Node * origem, tempoDisponivel) {

    // Reset auxiliary info
    for each (Node no : nodes){
        no.resetInfo();
        no.DistanciaàOrigem = INF       // INF = -20000
    }

    Node V = origem;                    // começar com a Origem
    V.DistanciaàOrigem = 0;

    MutablePriorityQueue<Node> PriorityQueue;        // iniciar Fila Prioridade
    PriorityQueue.insert(V);                         // vamos começar pela Origem

    while ( !(PriorityQueue.empty() )) {            // enquanto não estiver vazia
        V = PriorityQueue.proximo();
        V.visitado = true;

        for each (Edge Aresta : v->getAdjEdge()) {

            Node* Destino = Aresta->getDestino();

            if (!(Destino.visitado == true)) {      // se nunca tiver sido visitado

                oldDist = Destino.DistanciaàOrigem; // util mais à
                Destino->setPeso( V.peso + edge.peso);             // peso de um nó = peso da origem até ele

                if (V.DistanciaàOrigem >= Destino.DistanciaàOrigem && Destino.peso < tempoDisponivel) {
                    Destino.DistanciaàOrigem += 1;
                    Destino.setPath(v);                             // marcar o vértice de onde veio

                    if (oldDist == INF)                             // se for a primeira vez
                        PriorityQueue.insert(Destino);              // meter Vertice na Queue para processar
                    else
                        PriorityQueue.decreaseKey(Destino);         // Baixar o valor de Destino
                }                                                   // para que seja chamado mais rápidamente
            }}}
}
```

Figura 6.5: Pseudo-código do Algoritmo de *Prim*

6.2 Algoritmo de Pesquisa e Peso

A nossa aplicação não depende só da distância do ponto de origem e do tempo disponível. Teremos também em conta as preferências do utilizador nos tipos de Pontos de Interesse e o seu método de deslocamento.

Para resolver o dilema de visitar ou não um POI e não exceder o tempo disponível indicado pelo utilizador, resolvemos "duplicar"os nós. O algoritmo, ao encontrar um nó que seja Ponto de interesse ($rating > 0$), fará uma cópia exacta desse nó, mantendo Arestas de e para os mesmo nós que o original, mesmo *path*, mesmo *peso*, etc. Usará depois um nó como ponto de

passagem. O peso desse nó será o peso do nó anterior somado com o peso da aresta que o ligou. No entanto, utilizará o nó duplicado como Ponto de Interesse, ou seja, o peso nesse nó será igual ao peso do nó anterior, somado com o da aresta mais o tempo de visita associado a esse ponto de interesse. Esta duplicação só acontecerá caso o tipo de Ponto de Interesse corresponda aos tipos indicados pelo utilizador. Caso essa condição não se verifique, então esse ponto será apenas um ponto de passagem.

Pseudo-código associado:

```
for each (Node NODE : nodes) {
        boolean duplicate = false;

        verificarPreferencias() = NODE.preferencias
        // retira ratings e tempo de visitar caso
        // o Node não seja da preferencia do utilizdor

        if (duplicate) {
            duplicarNode(NODE);
        }
    }

    // Buscar Node com maior distancia
    int ultimaDistancia = 0;
    int rating = 0;
    Node nodeMaisAfastado;
    <Node *> highestPath;

    for each (Node NODE : nodes){
        if( NODE.distanciaOrigem > lastDistancia  ){
            nodeMaisAfastado; = NODE;
            ultimaDistancia = NODE.distanciaOrigem;
        }
    }

    // verificar Ratings até à origem
    Node* lastPath = nodeMaisAfastado;
    while(! lastPath == Origem ){
        highestPath.adicionar(lastPath.path );    // path obtido em Prim
        highestPath = lastPath.path;
    }

    return highestPath;    // retorna o Node com maior Rating
```

Figura 6.6: Pseudo-código do Algoritmo de *Pesquisa e Peso*

7. Análise de complexidade dos algoritmos efectivamente implementados

7.1 Análise teórica (temporal e espacial)

O algoritmo de *Prim*, usando lista de adjacências como nós utilizamos, tem como complexidade temporal

$$O(|E|log|V|),$$

em que E representa o número de Arestas e V o número de Vértices.

O nosso Algoritmo de Pesquisa e Peso é baseado num Algoritmo de Pesquisa em Largura. Como é aplicado a um Árvore de Expansão Mínima, só no pior caso é que percorrerá o mesmo número de Arestas, já que as Arestas mais pesadas para um Nó serão eliminadas no Algoritmo de *Prim*. A sua complexidade será de

$$O(|V + E|).$$

Portanto, a complexidade total dos algoritmos na nossa aplicação será de

$$O(|E|^2log|V| + |E|),$$

caso o tempo disponível fornecido seja suficiente para percorrer todos os nós.

7.2 Análise temporal empírica

De seguida encontra-se um gráfico da evolução do tempo de execução com o parâmetro tempo disponível para a viagem. A unidade de tempo utilizada foi o *clock tick* em milisegundos. Utilizou-se também sempre o mesmo ponto de partida (45.15, -8.6) e os mesmo pontos de interesse para todas as medições.

Começamos as medições no 5 minutos e medimos em intervalos de 5 minutos até aos 70 minutos. Fizemos duas medições para cada intervalo de tempo e utilizamos a média dos dois para o nosso gráfico.

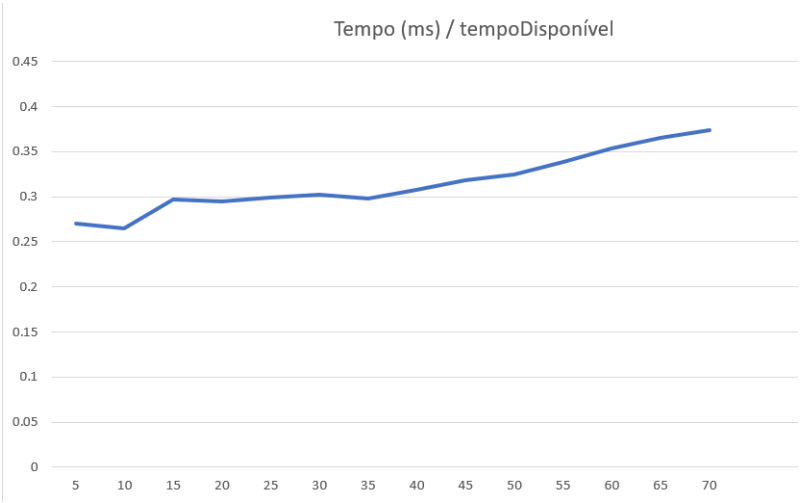

Figura 7.1: Evolução do tempo de Processamento consoante o *input* tempo disponível.

Como se pode verificar, não se observou nenhuma diferença significativa entre a primeira medição (0.270) e a última (0.374). Também executamos duas medições extra para 300 e 6000 minutos e os resultados não se mostraram muito afastados dos resultados utilizados para criação do gráfico (0.389 e 0.414, respectivamente). O objectivo destas medições extra é fornecer tempo necessário ao Algoritmo de *Prim* para alcançar todos os pontos do grafo, tornando assim o nosso Algoritmo de Pesquisa e Peso a "única"diferença.

Considerando que, no nosso mapa, os ficheiros de Arestas e de Nós contêm, aproximadamente, o mesmo número de entradas (aproximadamente 53000 Nós e 59000 Arestas) e que existem conjuntos de Nós e Arestas que podem nem ter conexão ao Nó de origem, podemos justificar o *plateau* como o ponto em que a aplicação percorre todos os pontos alcançáveis e, portanto, fornecer um maior *tempo disponível* não afectará o Algoritmo de *Prim*, mas sim apenas o nosso Algoritmo de Pesquisa e Peso.

De notar que as medições foram realizadas na aplicação, depois de compilada em modo *Release* no *CLion*, e executada num portátil a correr o Sistema Operativo *Windows*.

8. Conectividade dos grafos utilizados

Utilizamos os mapas fornecidos na página do Moodle da unidade curricular. Utilizamos o Mapa do Porto com menos detalhe para ir construindo a aplicação, já que a utilização do mapa mais detalhado resultava numa espera de 3 a 5 minutos só para o carregamento do mapa, algo que não seria eficiente para testes rápidos. No final, fizemos poucas alterações às estruturas de dados e leitura de ficheiros para conseguir ler o mapa mais detalhado.

Observamos que existem cerca de 53000 Nós e 59000 Arestas nos ficheiros do mapa. Olhando para os número podemos afirmar que se trata de um grafo esparso, pois o número de Arestas não é muito superior ao de Nós: apenas mais 11% de Arestas.

Pela visualização do grafo, verificamos que existem várias zonas sem ligação a outras. Temos uma exemplo de um *zoom-in* numa área com zonas não conexas. A vermelho temos as arestas percorridas pelo Algoritmo de *Prim* e a negro as arestas não percorridas. Podemos observar que, se não existir aresta de ligação, as arestas não são percorridas pelo algoritmo.

Figura 8.1: Exemplo de zonas não conexas

O facto do grafo ser não-conexo pode levar a soluções não ideais ou irrealistas. Por exemplo, se o ponto inicial coincidir com um ponto pertencente a uma dessas pequenas zonas isoladas, a solução fornecida não corresponderá à realidade, pois o percurso lido pelos algoritmos estará bastante limitado e nunca alcançará todos os pontos do grafo, ou mesmo pontos adjacentes. Uma solução para este problema seria, efectivamente, utilizar um ficheiro com um maior número de arestas.

9. Conclusão

De seguida, encontram-se as conclusões para a primeira e segunda parte do trabalho prático, que se traduz no presente relatório e código fornecido.

9.1 Análise e principais considerações

Durante a primeira parte do trabalho, dedicamos a nossa atenção para a análise dos algoritmos leccionados nas aulas teóricas a fim de escolher quais os que nos seriam mais úteis.

Tendo procedido a uma análise cuidada e ao levantamento de todos os detalhes que nos seriam indispensáveis para a elaboração da aplicação, fizemos este relatório tendo em vista a concretização do que neste mencionamos já na segunda parte do trabalho, de cariz prático.

Na segunda entrega, após *feedback* do Professor Henrique Lopes Cardoso, alteramos os erros do relatório e procedemos à aplicação do que inicialmente propusemos.

Através da API *GraphViewer* e do nosso conhecimento da cidade do Port,o consideramos que a nossa aplicação devolve uma solução realista, pelo menos quanto ao percurso sugerido dependendo do tempo fornecido. Sobre a exactidão dos tipos de Pontos de Interesse, não podemos garantir uma solução realista pois os *ratings* e categoria de Ponto de Interesse são atribuídos de forma aleatória, já que não encontramos nenhum ficheiro com informação usáveis sobre esse assunto e o esforço a criar uma lista dessas através de coordenadas e ID's seria bastante demoroso e propenso a erros.

9.2 Aspectos a Melhorar

Como referimos anteriormente, os tipos de Ponto de Interesse (p.e. Estádios, Museus ou Livrarias) são atribuídos aleatoriamente aquando da inicialização dos nós a partir do ficheiro respectivo. Para garantir o funcionamento ideal, deveríamos ter criado um ficheiro nosso com *ratings* e tempo médio de visita atribuídos a vários pontos do mapa. Não o fizemos pois não encontramos forma rápida, eficiente e verdadeira de identificar pontos a partir de coordenadas.

Outra funcionalidade que nos propusemos inicialmente a implementar foi a de utilizar as informações de *tags* da Sociedade de Transportes Colectivos do Porto (STCP) para criar

uma hipótese de deslocamento baseada em transportes públicos como Autocarros e/ou Metro. Decidimos descartar essa opção principalmente pela falta de tempo e pela dificuldade acrescida de ter de criar outros algoritmos tendo em conta a distância de Pontos de Interesse aos vários Pontos de paragem de Autocarro. Isto é, teríamos de obter os Pontos de Interesse mais próximos de cada paragem, para que o utilizador pudesse caminhar até eles, e distribuir o tempo disponível entre viagens, deslocação e tempo de visita.

9.3 Esforço dedicado por cada elemento do grupo

Durante a elaboração do relatório, todos os membros do grupo contribuíram activamente para que o mesmo atingisse o seu objectivo, o de estar completo e pronto para servir de base à parte prática que se seguiu.

Todos os capítulos do relatório foram escritos e revistos por todos os membros do grupo.

Relativamente ao código, todos os elementos contribuíram para que o mesmo chegasse ao ponto em que se encontra.