

# **TourMateApp**

rotas turísticas urbanas adaptáveis

Turma 5 Grupo 4

Jéssica Nascimento up201806723@fe.up.pt

Rafael Cristino up201806680@fe.up.pt

Xavier Pisco up201806134@fe.up.pt

# Índice

Siglas e Acrónimos	2
Descrição do Problema	3
Decomposição do problema	4
Identificação e formalização do problema	6
Dados de entrada	6
Dados de saída	6
Restrições	7
Função objetivo	7
Solução	9
Casos de utilização e funcionalidades implementadas	10
Estruturas de Dados Utilizadas	11
Algoritmos efetivamente implementados	12
Algoritmos lecionados	12
Algoritmos concebidos	12
Algoritmo para a geração de rota entre a origem e o destino	12
Algoritmo para a geração de rota com fim na origem	15
Algoritmo para a geração de rota sem destino	16
Análise de Complexidade dos Algoritmos Implementados	18
Algoritmo de geração de rota	18
Análise temporal	18
Algoritmo para a geração de rota entre a origem e o destino	18
Algoritmo para a geração de rota com fim na origem	19
Algoritmo para a geração de rota sem destino	19
Análise espacial	19
Análise empírica	20
Conectividade dos Grafos utilizados	22
Capturas de execução	23
Conclusão	25
Esforço de cada elemento	26

# Siglas e Acrónimos

- OSM: OpenStreetMap ([openstreetmap.org](https://openstreetmap.org))
- POI: Ponto de Interesse (*Point of Interest*)
- SCC: *Strongly Connected Component(s)*

# Descrição do Problema

A TourMateApp pretende **gerar itinerários eficientes, construídos consoante os interesses e a disponibilidade temporal de cada utilizador, bem como o ponto de partida e o ponto de chegada, baseados num mapa fornecido.**

Estes itinerários são definidos por um ponto de partida e um ponto de chegada, e tanto um como o outro são especificados pelo utilizador:

- **Caso o utilizador especifique ponto de partida** (provavelmente a sua localização atual) **mas não especifique ponto de chegada**, o itinerário será caracterizado como sendo um itinerário primariamente turístico, o que significa que **a aplicação procurará gerar um itinerário que passe pelo número máximo de pontos turísticos, dentro do tempo especificado e dando prioridade aos interesses do utilizador.** Se não for possível visitar nenhum ponto turístico dentro da janela de tempo fornecida, a aplicação dará ao utilizador a opção de alterar o tempo disponível ou cancelar a operação.
- **Caso o utilizador especifique ponto de chegada e ponto de partida**, e tenha ativado a opção de preencher o tempo restante disponível (calculado pela subtração do tempo do percurso ao tempo disponível) por visitas a atrações turísticas, **a aplicação procurará preencher esse tempo disponível com visitas a pontos turísticos, mediante os interesses do utilizador e sem exceder o tempo disponível especificado.** Senão, gera apenas o caminho mais curto desde a origem ao destino.

# Decomposição do problema

Vertentes em que se divide o problema, numa perspectiva computacional.

## 1 - Obtenção do grafo a partir do mapa

Várias alternativas:

- mapas provenientes do OSM (XML) sem processamento prévio
  - ◆ Utilização de um XML parser e utilização do resultado para criar os vértices e arestas, que são posteriormente adicionados ao grafo.
- mapas fornecidos pelos docentes da unidade curricular, processados
  - ◆ GridGraphs
  - ◆ PortugalMaps
  - ◆ TagExamples (PortugalMaps com tags provenientes do OSM)

## 2 - Cálculo e atribuição das distâncias às arestas do grafo

Cálculo com base na diferença entre as coordenadas dos vértices que são conectados pela aresta.

## 3 - Adição da relação dos POIs com os vértices do grafo

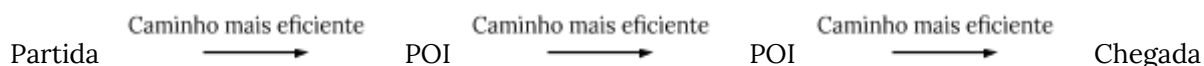
- Determinação do vértice que tem as coordenadas que mais se aproximam às coordenadas do ponto de interesse.
- Atribuição do objeto 'POI' ao vértice escolhido, e do vértice escolhido ao objeto 'POI'.

## 4 - Cálculo do itinerário mais eficiente desde a localização de partida à localização de chegada.

Ainda não tendo em conta as opções do utilizador, e assumindo a receção de ambas as localizações.

## 5 - Adição da capacidade de visitar POIs

Envolve o cálculo do caminho mais eficiente da localização de partida à localização do POI, e do ponto de interesse à localização de chegada.



## 6 - Adição da capacidade de visitar o número máximo de POIs, seguindo as preferências do utilizador (os seus interesses e tempo disponível)

Envolve a comparação dos caminhos resultantes de acrescentar variados POIs, sem ultrapassar o tempo disponível.

**7 - Adição da capacidade de gerar um itinerário sem ser definido um local de chegada (puramente turístico)**

Envolve o ponto anterior, mas sem a restrição do ponto de chegada.

# Identificação e formalização do problema

## Dados de entrada

→  $G(V, E)$  - Grafo do mapa da cidade em que o utilizador se encontra

◆  $V$  - Vértices correspondentes às interseções entre as ruas (arestas) que incluem:

- **Localização** - Coordenadas da sua localização
- **Adj** - Arestas adjacentes
- **Categoria** - categoria do POI (caso o vértice seja POI)
- **Outros dados** - Fornecidos no mapa (tags, no caso de mapas provenientes do OSM)

◆  $E$  - Arestas correspondentes às ruas

- **Destino** - Vértice correspondente ao final da aresta
- **Distância** (weight) - Distância coberta pela aresta (do vértice inicial ao final)
- **Outros dados** - Fornecidos no mapa (tags, no caso de mapas provenientes do OSM, por exemplo nome da rua)

→ **Localização/Origem**  $\in V$  - Vértice correspondente ao início do itinerário

→ **Destino**  $\in V$  - Vértice correspondente ao final do itinerário (opcional, como descrito na descrição do problema)

→ **Tempo livre** - Tempo máximo da duração do itinerário

→  $P_m \subset V$  - Lista dos pontos de interesse que presentes no mapa

→ **Preferências**  $\subset \text{categories}(V)$  - Lista dos tipos de preferências, contida no conjunto de todas as categorias possíveis dos vértices

## Dados de saída

- $M(V, E)$  - Grafo do mapa da cidade em que o utilizador se encontra
- $V_f \subset V$  - Lista dos vértices a percorrer
- $E_f \subset E$  - Lista das arestas a percorrer
- $P_f \in V$  - Lista dos POIs a percorrer
- $\text{dist}(V_f, E_f)$  - Distância estimada para o itinerário no total (incluindo o tempo médio de estadia nos pontos turísticos escolhidos)

## Restrições

- $\forall \text{ origem} \in V$  e  $\forall \text{ destino} \in [V, \text{null}]$  - A origem e o destino têm de existir (o destino pode ser calculado pelo programa caso não seja fornecido) e pertencer ao mapa utilizado
- $\forall e \in E, e.\text{distância} > 0$  - A distância entre dois vértices tem de ser sempre maior do que 0
- $\forall \text{ tempo livre} \in ]0, +\infty)$  - O tempo livre tem de ser maior que 0 e tem de ser maior ou igual que o tempo do menor caminho possível entre a origem e o destino
- $\forall \text{ POI} \in V$  - Os pontos de interesse de um percurso têm de existir e pertencer ao mapa utilizado
- $\forall \text{ Preferências} \subset \text{categories}(V)$  - As preferências do utilizador têm de fazer parte da lista de categorias dos vértices do grafo
- Seja OI e DI a origem e o destino iniciais e OF e DF a origem e os destinos finais,  $\forall OF = OI$  e, se  $DI \neq \text{null}$  então  $\forall DF = DI$  - Nos dados de saída, a origem e o destino têm de ser os mesmos dos dados de entrada (exceto quando não é especificado um destino)
- $\forall v \in V$  - Todos os vértices descritos no percurso resultante têm de pertencer ao mapa utilizado



## Função objetivo

- O itinerário ter o maior número possível de pontos de interesse dos dados de entrada, respeitando as preferências do utilizador quando possível
- O tempo estimado para a conclusão do itinerário não ultrapassar o tempo disponível fornecido nos dados de entrada
- O ponto inicial e o ponto de chegada corresponderem aos fornecidos nos dados de entrada

A função objetivo passa por:

Maximizar  $|Pf|$

Minimizar  $\sum_{e \in Ef} weight(e)$

A nossa solução dá prioridade à segunda equação, minimizar a distância percorrida.

# Solução

A solução que desenvolvemos para este problema assenta na utilização do algoritmo de Dijkstra e de princípios gananciosos para encontrar o caminho mais curto de um vértice para todos os outros.

Começando na origem (vértice atual), o algoritmo procura um POI (candidate) tal que  $category(POI) \in \text{preferências do utilizador}$  e cuja  $dist(POI)$  (distância do vértice atual ao candidato) seja  $\min(dist(POIs))$  (a distância mínima de todos os POIs).

Para além dessa seleção de um candidato a partir da distância ao vértice atual, dependendo dos casos, faz-se uma seleção a partir do ângulo entre os vetores

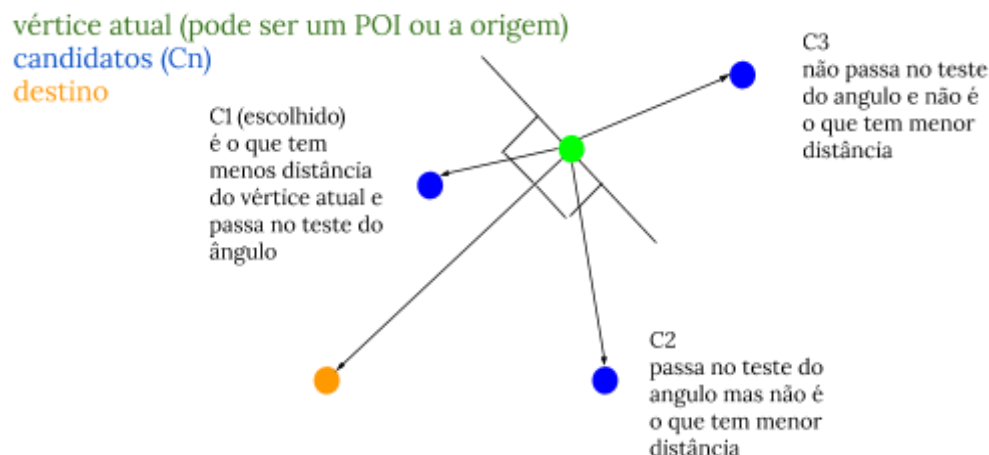
vetor v1, desde o vértice atual ao vértice candidato  
vetor v2, desde o vértice atual ao destino

procurando obter um ângulo menor que  $\Pi/2$  e maior que  $-\Pi/2$ , ou seja, verificar que o vértice candidato se localiza em direção ao destino.

Se após a seleção do candidato, se executar dijkstra com origem no candidato e a distância do candidato ao destino fizer com que o tempo previsto seja maior que o tempo disponível, então descarta-se o candidato e termina-se a rota (candidato = destino). Se for escolhido um candidato que não o destino, então esta busca pelo próximo candidato recomeça outra vez, mas com vértice atual = candidato obtido.

Em maior detalhe, encontra-se a solução na secção de algoritmos implementados.

Exemplo:



# Casos de utilização e funcionalidades implementadas

Estas funcionalidades aplicam-se aos mapas pré-processados e GridGraphs

**1. O utilizador tem um intervalo de x tempo e pretende aproveitá-lo visitando alguns POIs na área, tendo de regressar ao local original.**

Ponto de partida = ponto de chegada

**2. O utilizador está a fazer escala entre dois voos e pretende conhecer a cidade em que se encontra, sem perder o voo.**

Semelhante ao caso anterior

**3. O utilizador tem um local onde pretende estar, mas tem mais tempo do que o necessário para lá chegar.**

Ponto de partida = localização atual

Ponto de chegada = localização final

O utilizador visitará atrações turísticas a caminho do seu destino, sem ultrapassar o tempo disponível.

**4. O utilizador está numa situação puramente turística e não pretende ter uma específica localização final.**

Ponto de partida = localização atual

Ponto de chegada = null

O utilizador visitará o maior número de atrações turísticas sem ultrapassar o tempo disponível.

**5. O utilizador pretende utilizar a aplicação simplesmente como um GPS.**

A aplicação gerará o itinerário mais eficiente entre o ponto de partida e de chegada fornecidos.

Todos os pontos anteriores que envolvem pontos de interesse têm em conta as definições escolhidas pelo utilizador, nomeadamente os seus interesses, que definem o tipo de atrações que ele se sente compelido a visitar.

## **Outras funcionalidades implementadas**

Menu com testes de conectividade aos grafos, por exemplo encontrar pontos de articulação ou devolver as componentes fortemente conexas.

Menu com testes de performance aos algoritmos implementados.

Leitura de mapas XML do OSM, com capacidade de geração de rota (sem POIs). Os vértices podem ser obtidos a partir do seu nome, utilizando o algoritmo de string matching *edit distance*.

A rota pode ser realizada a pé ou de carro.

# Estruturas de Dados Utilizadas

Para a implementação deste trabalho, utilizamos:

- **Grafos:** para converter os mapas em vértices e arestas de modo a que fosse possível escolher os vários pontos necessários para o itinerário. Em todo o trabalho é usada esta estrutura para qualquer funcionalidade.
- **Mutable Priority Queue:** para auxílio na elaboração do algoritmo de Dijkstra.
- **Stack:** para a elaboração do algoritmo Strongly Connected Components que faz a conexão dos grafos.
- **Vector:** esta estrutura é muito usada ao longo do trabalho como auxiliar e como estrutura para guardar dados, como no caso da lista de preferências do utilizador.

# Algoritmos efetivamente implementados

## Algoritmos lecionados

Dijkstra (outros algoritmos de obtenção de caminhos no grafo foram implementados no contexto das aulas práticas, mas não na realização do projeto)

Strongly Connected Components

Find Articulation Points

String Edit Distance

## Algoritmos concebidos

Estes algoritmos foram desenvolvidos e testados nos mapas em TXT, strong e full. Funcionam tanto nos com forte conectividade como nos desconexos. Com os mapas diretamente extraídos do OSM apenas implementámos Dijkstra simples.

## Algoritmo para a geração de rota entre a origem e o destino

Passando por POIs, sem ultrapassar o tempo disponível

### Pseudocódigo

```
// G = (V, E)
// origin ∈ V, destination ∈ V
// poiVertexes ⊂ V
// userPreferences ⊂ categories(V)
// userTime - available time
```

**makeRoute(G, origin, destination, poiVertexes, userPreferences, userTime):**

```
    poiVertexesPreference ← getPOIsWithPreferences(G, userPreferences,
    destination)// vector
```

```
    route ← ∅ // vector
    currVx ← origin
    routeDist ← 0
    estimatedTime ← 0
    dijkstra(G, currVx)
```

```
    if time(destination) == DBL_MAX then
        ERROR
```

```
    if time(destination) > userTime then
        return pathTo(destination)
```

```

else
    while currVx != destination do
        currPath ← getNextPathPart(G, currVx, destination,
            estimatedTime, userTime, poiVertexesPreference, false)

        INSERT(route, currPath)

        routeDist += dist(currPath)
        estimatedTime += time(currPath)

        empty(route) ? currVx = destination : currVx =
            last(route)

    return (route, routeDist)

// G = (V, E)
// userPreferences ⊂ categories(V)
// destination ∈ V

getPOIsWithPreferences(G, userPreferences, destination)
    poiVertexesPreference ← ∅ // vector
    notConnected(G) = StronglyConnectedComponents(G).size() > 1 // SCC O(V + E)
    for each vx ∈ V:
        if category(vx) ⊂ userPreferences then
            INSERT(poiVertexesPreference, vx)
    if notConnected(G) && destination != NULL then
        for each vx ∈ poiVertexesPreference do
            if destination not in bfs(G, vx) then // BFS O(V+E)
                REMOVE(poiVertexesPreference, vx)
    return poiVertexesPreference

// G = (V, E)
// currVx ∈ V, destination ∈ V
// poiVertexesPreference ⊂ V
// categories(poiVertexesPreference) ⊂ categories(V)
// estimatedTime - estimated used time
// userTime - available time
// returnToOrigin - boolean whether it's a return to origin path

getNextPathPart(G, currVx, destination, estimatedTime, userTime,
    poiVertexesPreference, returnToOrigin):
    if dist(currVx) != 0 then //asserts dijkstra was executed on currVx
        dijkstra(G, currVx)

    // greedy
    candidate ← getCandidate(G, currVx,
        destination, poiVertexesPreference, currTime, userTime, returnToOrigin)

```

```

candidateTime ← estimatedTime + time(candidate)
candidatePath ← pathTo(candidate)

if candidate == destination
    then return candidatePath // == destinationPath

destinationPath ← pathTo(destination)

dijkstra(G, candidate)

// backtracking
if candidateTime + time(destination) < userTime then
    REMOVE(poiVertexesPreference, candidate)
    return candidatePath

return destinationPath

// G = (V, E)
// currVx ∈ V, destination ∈ V
// poiVertexesPreference ⊂ V
// categories(poiVertexesPreference) ⊂ categories(V)
// returnToOrigin - boolean whether it's a return to origin path
// currTime - estimated used time
// userTime - available time

getCandidate(G, currVx, destination, poiVertexesPreference, currTime,
userTime, returnToOrigin):
    for each vx ∈ poiVertexesPreference:
        if !returnToOrigin or currTime > userTime/2 then
            lessPreferable(vx) ← false
            if
                currVxToDestinationVector ← getVector(currVx,
destination)
                currVxToVxVector ← getVector(currVx, vx)

                if angle(currVxToDestinationVector, currVxToVxVector) >
PI/2 or angle(currVxToDestinationVector,
currVxToVxVector) < -PI/2 then
                    lessPreferable(vx) ← true

    sort(poiVertexesPreference, [(vx1 ∈ V, vx2 ∈ V) {
        if lessPreferable(vx1) == !lessPreferable(vx2) then
            return !lessPreferable(vx1)
        else
            return dist(vx1) < dist(vx2)
    })

```



```

    return poiVertexesPreference.empty() ? destination :
first(poiVertexesPreference)

```

### Algoritmo para a geração de rota com fim na origem

Passando por POIs, sem ultrapassar o tempo disponível. Usa as mesmas funções auxiliares do algoritmo acima.

```

// G = (V, E)
// origin ∈ V
// poiVertexes ⊂ V
// userPreferences ⊂ categories(V)
// userTime - available time

returnToOrigin(G, origin, poiVertexes, userPreferences, userTime):
    poiVertexesPreference ← getPOIsWithPreferences(G, userPreferences,
origin)// vector

    route ← ∅ // vector
    currVx ← origin
    routeDist ← 0
    estimatedTime ← 0
    starting ← true
    dijkstra(G, currVx)

    while starting || currVx != origin do
        starting ← false
        currPath ← getNextPathPart(G, currVx, destination,
estimatedTime, userTime, poiVertexesPreference, true)

        INSERT(route, currPath)

        routeDist += dist(currPath)
        estimatedTime += time(currPath)

        empty(route) ? currVx = origin : currVx = last(route)

    return (route, routeDist)

```

## Algoritmo para a geração de rota sem destino

Passando por POIs, sem ultrapassar o tempo disponível. Termina no último POI.

```
// G = (V, E)
// origin  $\in V$ 
// poiVertexes  $\subset V$ 
// userPreferences  $\subset \text{categories}(V)$ 
// userTime - available time

noDestination(G, origin, poiVertexes, userPreferences, userTime):
    poiVertexesPreference  $\leftarrow$  getPOIsWithPreferences(G, userPreferences,
    NULL)// vector

    route  $\leftarrow \emptyset$  // vector
    currVx  $\leftarrow$  origin
    routeDist  $\leftarrow 0$ 
    estimatedTime  $\leftarrow 0$ 

    while true do
        currPath  $\leftarrow$  getNextPathPartTouristic(G, currVx, destination,
        estimatedTime, userTime, poiVertexesPreference, true)
        if empty(currPath) then
            break

        INSERT(route, currPath)

        routeDist += dist(currPath)
        estimatedTime += time(currPath)

        currVx = last(route)

    return (route, routeDist)

// G = (V, E)
// currVx  $\in V$ , destination  $\in V$ 
// poiVertexesPreference  $\subset V$ 
// categories(poiVertexesPreference)  $\subset \text{categories}(V)$ 
// estimatedTime - estimated used time
// userTime - available time

getNextPathPartTouristic(G, currVx, estimatedTime, userTime,
poiVertexesPreference):
    dijkstra(G, currVx)

    // greedy
    candidate  $\leftarrow$  getCandidate(G, currVx, currVx, poiVertexesPreference,
-1, userTime, true) // same function as previous algorithms
```

```

if candidate == currVx then
    return ∅

candidateTime ← estimatedTime + time(candidate)
candidatePath ← pathTo(candidate)

if candidateTime < userTime then
    REMOVE(poiVertexesPreference, candidate)
    return candidatePath

return ∅

```

# Análise de Complexidade dos Algoritmos Implementados

## Algoritmo de geração de rota Análise temporal

### Algoritmo para a geração de rota entre a origem e o destino

Começando pela função `getCandidate()`, esta utiliza um `sort` da `stl`, cuja complexidade temporal é  $O(N \log(N))$  e tem um ciclo que percorre todos os vértices do grafo,  $O(N)$ . Os elementos aqui percorridos são apenas uma pequena parte dos vértices (são os pontos de interesse que têm como categoria o que o utilizador escolheu e que não se desvie muito do vetor que liga o vértice anterior ao destino), pelo que na maioria dos casos, a complexidade temporal desta função não será significativa, comparativamente ao resto das funções utilizadas, mas no pior caso, será  $O(V \log(V))$ .

A função `getNextPathPart()` utiliza apenas um algoritmo de dijkstra, cuja complexidade temporal é  $O((V + E) \log V)$  e chama, uma vez, a função `getCandidate`, pelo que será  $O(V \log V) + O((V + E) \log V)$  que pode ser simplificada para  $O((V + E) \log V)$ .

Quanto há `getPoisWithPreference()` começa com o algoritmo `strong connected components`, que tem  $O(V + E)$ , depois corre todos os vértices do grafo,  $O(V)$ , e, por fim, ao correr os vértices cujo POI está nas preferências do utilizador faz uma `breath-first-search`, ou seja, no pior caso, vai ter uma complexidade temporal  $O(V (V + E))$ .

Por fim a função `makeRoute()` começa com uma chamada há função `getPoisWithPreference()`,  $O(V (V + E))$ , tem um ciclo para todos os vértices,  $O(V)$ , chama uma vez a função do algoritmo dijkstra  $O((V + E) \log V)$  e, dentro de um ciclo que percorre alguns dos vértices chama a função `getNextPathPart()`,  $O((V + E)V \log V)$ . Neste último ciclo o número de vértices percorridos é tanto maior quanto maior for o percurso, pelo que, no melhor caso, a parte mais dispendiosa de tempo será a inicial, e, por isso, o algoritmo será  $O(V (V + E))$ , mas no pior caso e no caso médio será o último ciclo da função  $O((V + E) V \log V)$ .

### **Algoritmo para a geração de rota com fim na origem**

Neste algoritmo, a função principal chama a função `getPOIsWithPreference`,  $O(V(V + E))$ , corre uma vez todos os vértices,  $O(V)$ , executa uma vez o algoritmo dijkstra  $O((V + E) \log V)$ , e, por fim, executa um ciclo pelos vértices que chama a função `getNextPathPart()`, cuja complexidade temporal foi analisada anteriormente e é  $O((V + E) \log V)$ .

Tal como no algoritmo anterior, a função `getNextPathPart()` é executada tantas vezes como o número de vértices que são percorridos no percurso, fazendo com que, no melhor caso esta função tenha uma complexidade temporal  $O(V(V + E))$ , mas no pior caso,  $O((V + E)V \log V)$ .

### **Algoritmo para a geração de rota sem destino**

Na função `noDestination()`, tal como nas outras, começa por ser chamada a função `getPoisWithPreference()`, depois existe um ciclo para percorrer todos os vértices do grafo  $O(V)$ , e um outro ciclo `while` que chama a função `getNextPathPartTouristic()`.

Esta função já referida anteriormente tem uma complexidade temporal  $O((V + E) \log V)$ , e é percorrida tantas vezes quantos os vértices pertencentes ao itinerário, logo, no caso médio vai ser percorrida  $V$  vezes e, assim, este algoritmo vai ter uma complexidade temporal  $O((V + E)V \log V)$ .

## **Análise espacial**

Todos estes três algoritmos utilizam a mesma memória do computador, são necessários um Grafo, cujo tamanho será  $V$  vértices e  $E$  arestas. Para além deste grafo, é utilizado um vetor com os vértices que são pontos de interesse relacionados com o utilizador, sendo que esse vetor guarda apontadores para os vértices, pelo que o seu tamanho será o número de vértices que guardar,  $N \in [0, V]$ .

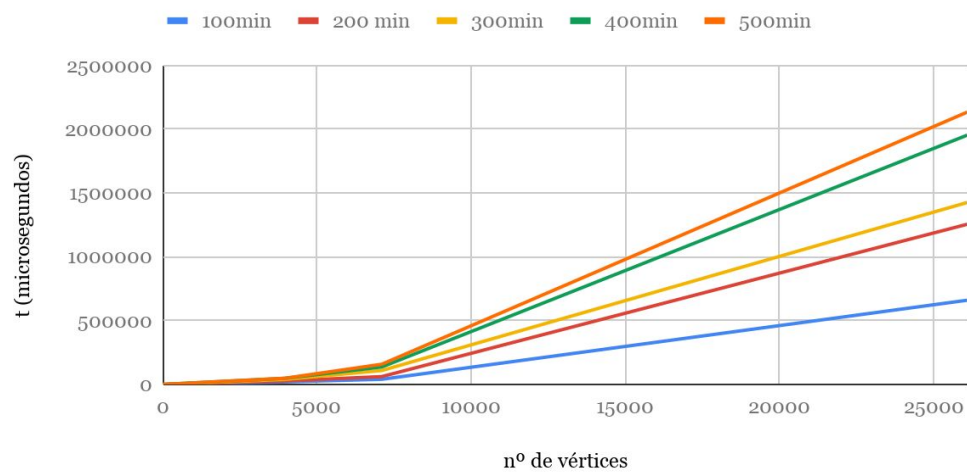
Para além destas duas estruturas também existe um vetor com as preferências do utilizador, que terá um tamanho entre 0 e 12, apontadores para os vértices de origem e destino, 2 inteiros relacionados com o tempo disponível e o tempo do percurso, e um booleano.

Assim, estas últimas variáveis são desprezáveis quando comparadas com o grafo e com o vetor dos pontos de interesse, e, por isso, a complexidade espacial será  $O(V + E) + O(V)$ , que pode ser simplificada para,  $O(V + E)$ .

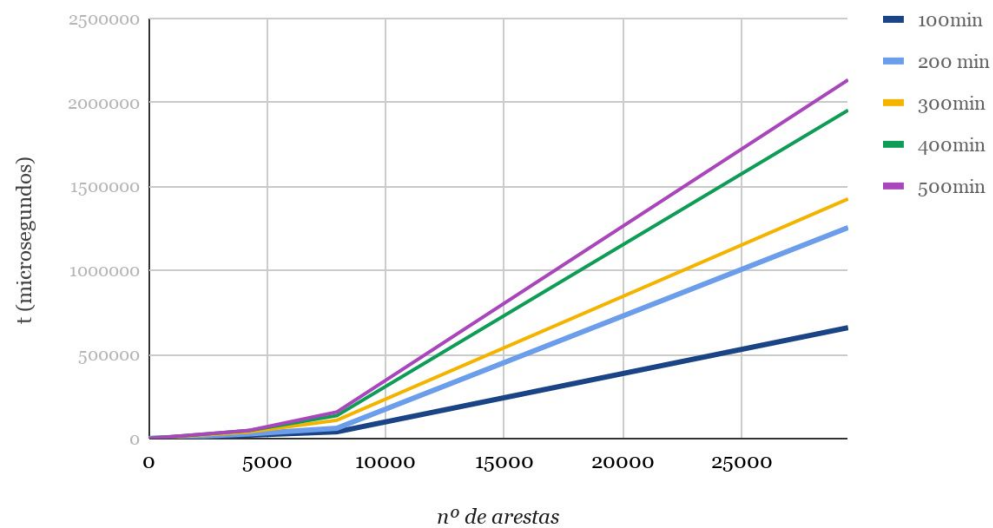
## Análise empírica

Os valores apresentados foram obtidos a partir das versões fortemente conexas dos mapas do Porto, de Espinho e de Penafiel. Os valores são a média dos valores obtidos em 3 execuções consecutivas com os mesmos parâmetros, para cada mapa e para cada set de parâmetros único.

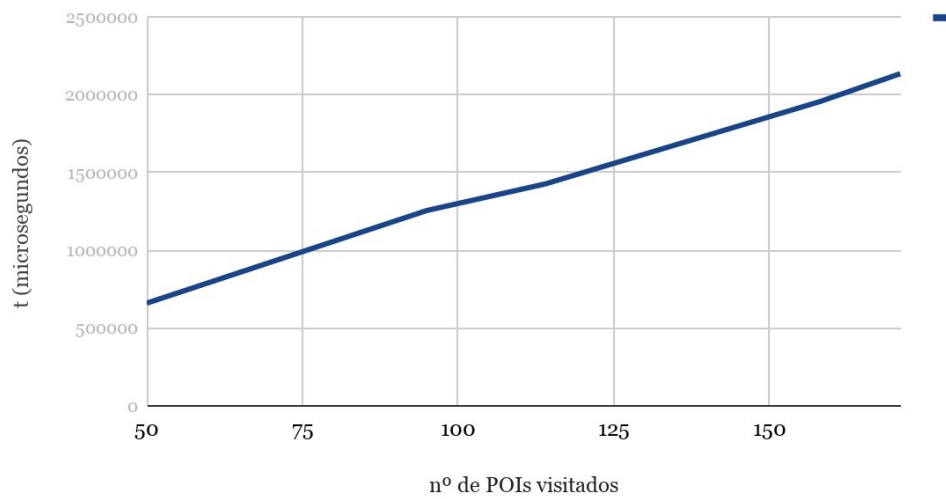
### Evolução do tempo de execução com a quantidade de vértices no grafo



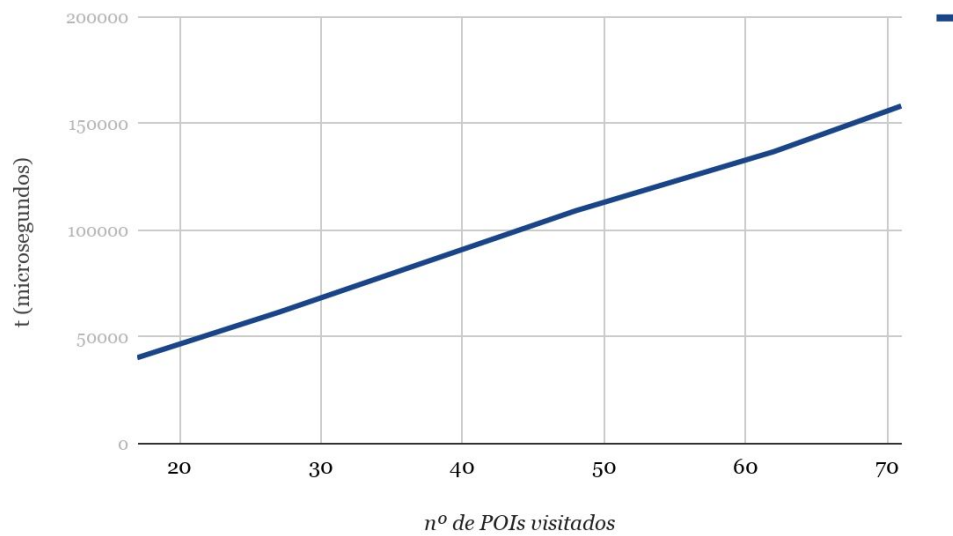
### Variação do tempo de execução com a quantidade de arestas no grafo



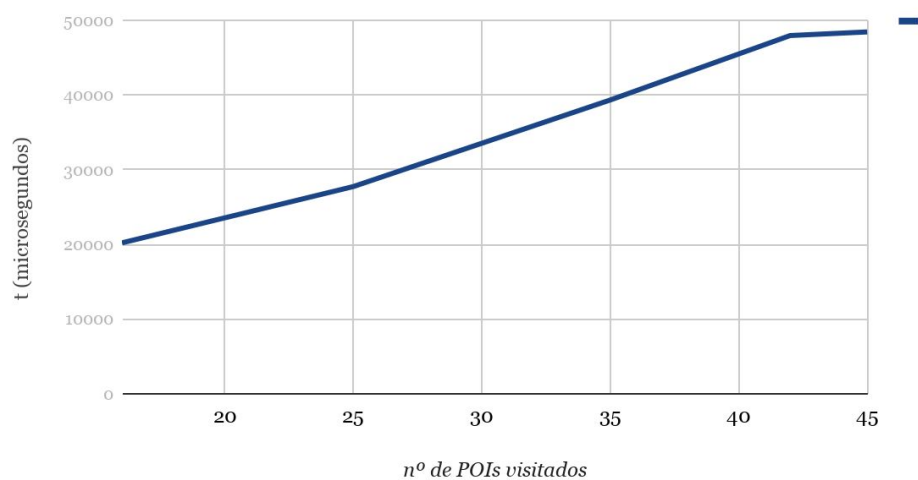
PORTO: Variação do tempo de execução com o número de POIs visitados



ESPINHO: Variação do tempo de execução com o nº de POIs visitados



PENAFIEL: Variação do tempo de execução com o nº de POIs visitados



# Conectividade dos Grafos utilizados

Para testar a conectividade dos grafos utilizados, implementámos algoritmos para encontrar os *Strongly Connected Components* do grafo e também os *Articulation Points*.

Se apenas houver um SCC, então o grafo é **fortemente conexo**, ou seja, existe pelo menos um caminho possível entre qualquer par de vértices. Tal é o caso dos mapas categorizados como *Strong*.

Exemplo obtido ao executar o algoritmo no mapa *strong* do Porto:

```
----- Strongly Connected Components -----
The strongly connected components are represented by the id's of their vertexes
Each line is a strongly connected component

25202; 32743; 38019; 31177; 15244; 14601; 47538; 17376; 13879; 15853; 34572; 30953; 14738; 38327; 9930; 39757; 21882; 22276; 26

Above, each line is a strongly connected component.

Only 1 strongly connected component, which means that the graph is strongly connected!
```

Caso contrário o grafo **não é conexo**, e pode ter vértices inalcançáveis a partir doutros. Tal é o caso dos mapas categorizados como *Full* e dos mapas provenientes do OSM.

Exemplo obtido ao executar o algoritmo no mapa *full* de Espinho:

```
1334;
8285;
13727;
4472;
102;
10754;

Above, each line is a strongly connected component.

The graph is not strongly connected
```

Se não houver *Articulation Points*, então o grafo é **biconexo**. Tal é o caso dos *GridGraphs*, utilizados para testar casos mais simples.

Exemplo obtido ao executar o algoritmo num *GridGraph*:

```
----- Articulation Points -----
The articulation points are represented by their id

No articulation points were found! the graph is biconnected
```



# Capturas de execução

## Resultado da execução num GridGraph

```
*****
----- The Route -----
*****

origin
0 ->
# POI: 1: theme_park
-> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 ->
# POI: 25: theme_park
-> 42 -> 59 -> 76 -> 93 -> 110 -> 127 -> 128 -> 145 -> 146 -> 163 -> 164 -> 165 ->
# POI: 182: aquarium
->
# POI: 199: artwork
-> 216 -> 233 -> 234 ->
# POI: 235: aquarium
-> 218 -> 201 ->
# POI: 184: artwork
-> 201 -> 218 -> 217 -> 234 -> 251 -> 250 -> destination
```

Distance = 7.2km

It will take approximately 13.3333 mins

If you see any POIs you didn't mean to visit, it is because they were in the path anyway

Do you want to see the route in the graph viewer?

If you say yes, and you've closed the graph viewer window, the program will freeze. (y/n)

y

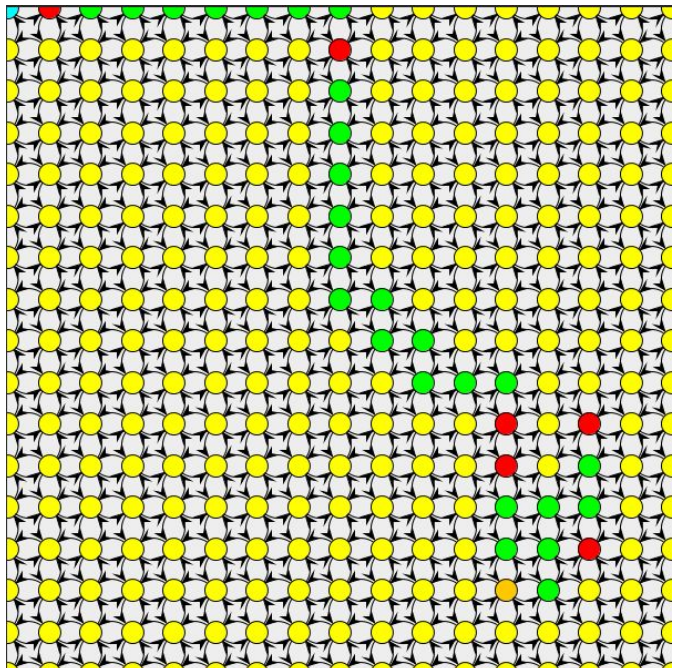
Showing the resulting route in the graph viewer.

In RED, you can see the interest points

In BLUE, you can see the origin point

In ORANGE, you can see your destination

In GREEN you can see the path



## Resultado da execução no mapa de Espinho (strong)

Distance = 12.3985km

It will take approximately 22.9602 mins

If you see any POIs you didn't mean to visit, it is because they were in the path anyway

Do you want to see the route in the graph viewer?

If you say yes, and you've closed the graph viewer window, the program will freeze. (y/n)

y

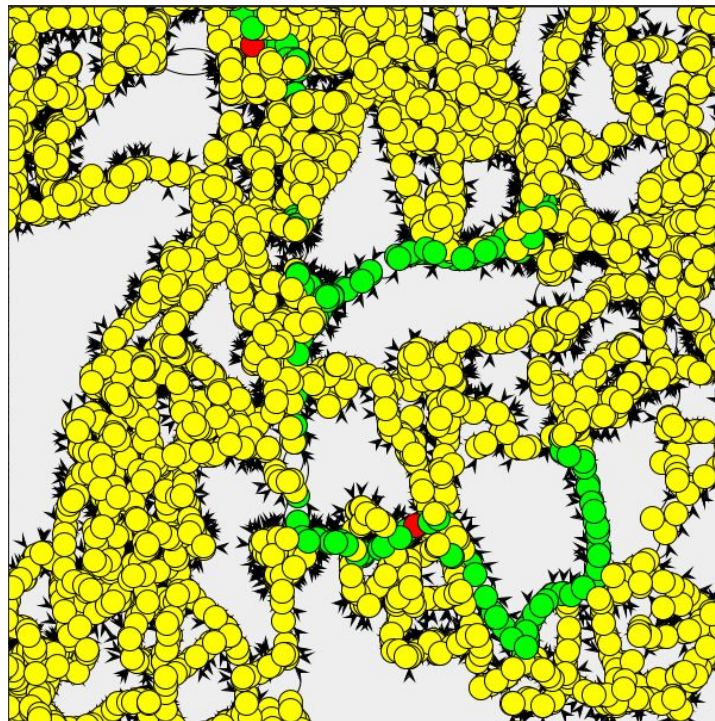
Showing the resulting route in the graph viewer.

In RED, you can see the interest points

In BLUE, you can see the origin point

In ORANGE, you can see your destination

In GREEN you can see the path



# Conclusão

Após uma análise do problema, conseguimos decompô-lo em problemas mais pequenos, como os 3 casos de rotas, e assim aplicar melhor a nossa solução. Esta solução desenvolvida é uma aproximação à função objetivo, mas não é a solução óptima.

Mesmo assim tivemos vários problemas, o que nos fez alterar o nosso rumo de vez em quando, por exemplo, as nossas possíveis soluções apresentadas na primeira entrega acabaram por ser descartadas em função de uma melhor solução nesta segunda entrega.

Tentámos utilizar os mapas extraídos diretamente do OSM, mas mostrou-se ser demasiado complexo para o tempo que tínhamos e decidimos aplicar os algoritmos aos mapas pré processados. No entanto, tempos a funcionalidade de encontrar o caminho mais curto da origem ao destino nos mapas do OSM.

Durante este trabalho existiram várias ideias que foram discutidas entre o grupo, tentando sempre melhorar o resultado final. Para além disso, com estas ideias conseguimos perceber melhor como implementar algoritmos em grafos e quais as vantagens de utilizar alguns algoritmos em detrimento de outros, tanto no tempo de execução como nos gastos de recursos.

# **Esforço de cada elemento**

Este trabalho foi desenvolvido sempre em conjunto. Ao longo de todo o processo, cada elemento foi partilhando as suas ideias e juntos fomos discutindo as melhores estratégias para a implementação do código. Na realização deste cada um teve sempre em conta o que foi definido em grupo.

Cada elemento do grupo contribuiu igualmente para o desenvolvimento de todo o projeto.