U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

TourMateApp: rotas turísticas urbanas adaptáveis (tema 3)

Conceção e Análise de Algoritmos
2ºAno MIEIC

24 de abril de 2020

Inês Silva, up201806385@fe.up.pt
Mariana Truta, up201806543@fe.up.pt
Rita Peixoto, up201806257@fe.up.pt
Grupo 8, Turma 3

Índice

Introdução

Descrição do tema

Neste projeto, a intenção é implementar o aplicativo **TourMateApp**, que permite a construção de itinerários turísticos adaptáveis às preferências e disponibilidade do usuário.

A aplicação mantém uma lista de pontos turísticos de interesses *(POI)* e sugere itinerários que incluem as atrações mais adequadas ao perfil do usuário, num caminho que possa ser realizado no tempo indicado, de uma origem a um destino final, indicado pelo mesmo.

As recomendações maximizam o número de atrações turísticas de acordo com o perfil e as preferências do utilizador, que também pode selecionar circuitos a pé, de carro ou de transporte público.

Para além disso, é necessário ter em atenção se as áreas pelas quais os caminhos passam se encontram, no presente momento, inacessíveis.

Esta aplicação irá utilizar mapas reais extraídos do *OpenStreetMaps* (www.openstreetmap.org) e coordenadas geográficas de alguns pontos de interesse turístico.

Formalização do problema

A rede viária pode ser representada por um **grafo dirigido** em que os **vértices** representam interseções, as **arestas** representam vias e os **pesos** representam distâncias, tempos.

Dados de entrada

G (N, E) - grafo dirigido pesado, representando o mapa da cidade em questão. Constituído por:

N - Conjunto de vértices que representam os pontos da cidade. Cada ponto é caracterizado por:

id – identificador único do vértice

type - NULL se não for um ponto de interesse

adj ⊆ E – conjunto de arestas que partem deste ponto

E - Conjunto de arestas que ligam os vértices entre si, que representam os caminhos entre pontos. Cada aresta é caracterizada por:

id - identificador único da aresta

weight – tempo entre vértices

dest ∈ N - vértice destino da aresta

POIs ∈ N - conjunto de todos os pontos de interesse

Preferências do utilizador - tempo máximo para realizar o caminho, meio de locomoção selecionado, pontos turísticos de interesse, cidade em que se encontra, ponto inicial do percurso (ponto onde o usuário se encontra, N_i), ponto final do percurso (ponto de destino do usuário, N_f), entre outros.

Dados de saída

WeightF - peso total de todas as arestas percorridas no percurso (tempo de duração);

P - sequência ordenada dos vértices que representam o melhor caminho entre N_i e N_f, passando pelo maior número de pontos de interesse possíveis de visitar que vão de encontro às preferências selecionadas, sem ordem especifica.

Restrições

- **Para todos os vértices:**
  - type(N[i]) = “information” v “hotel” v “attraction” v “viewpoint” v “guest_house” v “picnic_site” v “artwork” v “camp_site” v “museum” v “*” v “pt_stop” v NULL;
  - type(Ni) = “information” v “hotel” v “attraction” v “viewpoint” v “guest_house” v “picnic_site” v “artwork” v “camp_site” v “museum” v “*” v “pt_stop” v NULL;
  - type(Nf) = “information” v “hotel” v “attraction” v “viewpoint” v “guest_house” v “picnic_site” v “artwork” v “camp_site” v “museum” v “*” v “pt_stop”v NULL.

Nota: O facto de o *type* do vértice ser NULL significa que é um vértice genérico e, portanto, não representa um ponto de interesse.

- **Para todas as arestas:**
  - weight(E[i]) ≥ 0 e WeightF ≥ 0, uma vez que representam distâncias/tempos.

- **Ni ∈ N ∧ Ni = P_0**, isto é, o ponto inicial tem de ser o primeiro vértice na sequência ordenada de vértices que representa o trajeto ótimo;
- **Nf ∈ N ∧ Ni = P_f**, isto é, o ponto final tem de ser o último vértice na sequência ordenada de vértices que representa o trajeto ótimo;
- O **meio de locomoção** escolhido pelo usuário pode ser “a pé” v “carro” v “transportes públicos”.

Função objetivo

A **função objetivo** deve retornar um itinerário ótimo que inclua as atrações mais adequadas ao perfil do usuário e que possa ser realizado no tempo indicado, de uma origem a um destino final, indicado pelo utilizador.

A solução ótima maximiza o número de atrações que o usuário consegue visitar dentro das suas restrições de tempo e preferências, encontrando o caminho mais curto entre os pontos turísticos o que consequentemente permite obter o trajeto mais rápido.

Sendo assim, a **função objetivo** deve minimizar a função P:

$$P = \sum w(e), \qquad e \in E$$

Perspetiva de solução

Técnicas de implementação

A resolução deste problema passa por um conjunto de fases sucintamente descritas de seguida.

1. Pré-processamento do grafo

Nesta fase irá ser removida informação desnecessária e redundante.

Tendo em atenção o modo de locomoção do utilizador, serão tomadas diferentes medidas:

- Se o utilizador tiver selecionado transportes públicos, atendendo ao exemplo disponibilizado (figura 1) e sabendo o ID do nó mais próximo das paragens disponíveis, no grafo principal, o atributo *type* deste nó tomará o valor "pt_stop".
- Se o modo de locomoção escolhido pelo utilizador for carro ou a pé, nada será feito nesta fase.

```
STCP:
(ID nó mais proximo, Código Paragem, Código Linha, Código Zona, Concelho, Freguesia, Morada, Tipo Paragem)
(128560439, 'ADA3', '603', 'C1', 'PORTO', 'PARANHOS', 'RUA VALE FORMOSO', 'PARAGEM SEM ABRIGO')

METRO:
(ID nó mais próximo, Nome Paragem, Lista de Linhas)
(698819576, 'Estádio do Dragão', ['A', 'B', 'E', 'F'])
```

Figura 1 Exemplo da informação relativa a transportes públicos

2. Verificar se é viável a deslocação entre dois pontos

Inicialmente, após serem fornecidos os pontos de partida e de chegada, é necessário verificar se existe pelo menos um caminho possível entre esses pontos, não sendo considerados os pontos de interesse nem o tempo máximo de deslocação.

Nesta fase, não se pretende guardar o caminho encontrado nem o otimizar, mas sim garantir que não estão a ser consideradas zonas inacessíveis e que por isso é possível chegar ao destino. Caso não seja, é necessário informar o utilizador.

3. Encontrar o caminho mais curto entre dois pontos num grafo não dirigido

Na generalidade dos problemas de trajetos, interessa não só encontrar um trajeto que vá de encontro aos requisitos do utilizador, mas também o melhor percurso possível, minimizando a distância percorrida e a duração.

Nesta iteração, ainda não vão ser tidos em conta os pontos de interesse do usuário, focando apenas em otimizar o percurso possível entre dois pontos, isto é, chegar do ponto inicial ao destino dentro dos limites de tempo desejado.

No caso de não haver nenhum caminho possível de se realizar dentro do tempo fornecido, será exposto o melhor caminho conseguinte.

4. Encontrar o máximo de caminhos possíveis cuja soma de tempo não exceda o especificado pelo utilizador

Tendo a certeza da existência de um caminho possível dentro do tempo restringido (ou não, resolvido como esclarecido no ponto anterior), interessa agora encontrar um caminho que maximize o número de pontos de interesse que conferem as preferências do utilizador.

Serão analisados os caminhos possíveis que obedeçam ao modo de locomoção indicado.

5. Maximizar o número de pontos de interesse visitados

Por fim, dentro dos obtidos, é necessário escolher qual o caminho que abrange o maior número de pontos de interesse do usuário.

É importante referir que, ao longo destas fases, é avaliada a conectividade do grafo, uma vez que certas áreas podem encontrar-se inacessíveis.

Algoritmos

A análise da possibilidade de chegar da origem ao destino passa por uma pesquisa no grafo a partir do vértice de origem, a qual pode ser realizada utilizando o algoritmo de pesquisa em profundidade ou de pesquisa em largura.

1. Algoritmo de Pesquisa em Largura

A **pesquisa em largura** (figura 2) é um dos métodos mais simples para a exploração de um grafo e é a base para muitos outros algoritmos, como o de Dijkstra.

Dado um vértice origem, explora-se sistematicamente as arestas do grafo, descobrindo todos os vértices alcançáveis a partir de s (vértices adjacentes), passando posteriormente à exploração do vértice seguinte. Isto é, encontra-se todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância k + 1. Para tal, é utilizada uma fila em que se processa o vértice da frente da mesma e à qual são adicionados no fim os vértices descobertos através desse processamento.

```
BFS(G, s):
1.      for each v ∈ V do discovered(v) ← false
2.      Q ← ∅

3.      ENQUEUE(Q, s)
4.      discovered(s) ← true

5.      while Q ≠ ∅ do
6.          v ← DEQUEUE(Q)
7.          pre-process(v)
8.          for each w ∈ Adj(v) do
9.              if not discovered(w) then
10.                 ENQUEUE(Q, w)
11.                 discovered(w) ← true
12.         post-process(v)
```

Figura 2 Pseudocódigo do algoritmo de pesquisa em largura

2. Algoritmo de Pesquisa em Profundidade

O **algoritmo de pesquisa em profundidade** (figura 3) consiste em explorar todas as arestas a partir do último vértice encontrado, sendo implementado de forma recursiva e utilizando o método de *backtracking*. Quando todas as arestas de um vértice forem exploradas, retorna e explora as restantes arestas do vértice que o antecedia. O algoritmo aprofunda a pesquisa até que encontre o vértice pretendido.

```
G = (V, E)
Adj(v) = {w | (v, w) ∈ E} (∀ v ∈ V)

DFS(G):
1.  for each v ∈ V
2.      visited(v) ← false
3.  for each v ∈ V
4.      if not visited(v)
5.          DFS-VISIT(G, v)

DFS-VISIT(G, v):
1.  visited(v) ← true
2.  pre-process(v)
3.  for each w ∈ Adj(v)
4.      if not visited(w)
5.          DFS-VISIT(G, w)
6.  post-process(v)
```

Figura 3 Pseudocódigo do algoritmo de pesquisa em profundidade

3. Algoritmo de Dijkstra

O **algoritmo de Dijkstra** é um algoritmo ganancioso que tem como objetivo calcular o caminho mais curto entre dois vértices de um grafo dirigido pesado, sem arestas de peso negativo.

Este algoritmo (figura 4) é semelhante ao de pesquisa em largura, diferenciando-se no facto de utilizar uma fila de prioridade (alterável) como estrutura de dados auxiliar para guardar a ordem dos próximos vértices a pesquisar, na qual se prioriza os vértices cuja soma do peso das arestas do caminho origina uma distância mínima (em vez de serem processados pela ordem em que foram descobertos, numa fila simples). É um algoritmo ganancioso pois em cada passo procura maximizar o ganho imediato, ou seja, minimizar a distância entre a origem e o destino.

```
DIJKSTRA(G, s):  // G=(V,E), s ∈ V
1.   for each v ∈ V do
2.      dist(v) ← ∞
3.      path(v) ← nil
4.   dist(s) ← 0
5.   Q ← ∅   // min-priority queue
6.   INSERT(Q, (s, 0)) // inserts s with key 0
7.   while Q ≠ ∅ do
8.      v ← EXTRACT-MIN(Q) // greedy
9.      for each w ∈ Adj(v) do
10.        if dist(w) > dist(v) + weight(v,w) then
11.           dist(w) ← dist(v)+ weight(v,w)
12.           path(w) ← v
13.           if w ∉ Q then  // old dist(w) was ∞
14.              INSERT(Q, (w, dist(w)))
15.           else
16.              DECREASE-KEY(Q, (w, dist(w)))
```

Figura 4 Pseudocódigo do algoritmo de Dijkstra

Em cada vértice processado é guardada a informação relativa ao vértice que o antecede no caminho e, após ser encontrado o vértice final na fila de prioridade, percorre-se o caminho encontrado no sentido contrário, guardando-o para o retornar, até se regressar ao vértice inicial.

Para analisar a eficiência deste algoritmo, considere-se *V* o conjunto dos vértices do grafo e *E* o conjunto das arestas. Sendo que o número de extrações e inserções na fila de prioridades é |*V*| e cada uma destas operações pode ser realizada em tempo logarítmico no tamanho da fila, que é no máximo |*V*|, o tempo de execução das extrações e inserções na fila de prioridades é de $O(|V| * log |V|)$.

A operação de reordenação dos vértices na fila de prioridade é feita, no pior caso, uma vez por cada aresta, ou seja, |*E*| vezes, e pode ser realizada em tempo logarítmico no tamanho da fila, que no máximo é |*V*|, resultando numa complexidade $O(|E| * log |V|)$.

Assim, pode-se afirmar que o tempo de execução do **algoritmo de Dijkstra** é $O((|V|+|E|) * log |V|)$.

4. Algoritmo A*

O **algoritmo A*** constitui uma otimização do algoritmo de Dijkstra que permite um melhoramento (*speedup*) moderado ao utilizar uma função heurística para orientar a busca do vértice de destino, não garantindo, no entanto, que o caminho encontrado seja o ótimo.

O **algoritmo A*** apenas difere no de Dijkstra na ordenação dos vértices na fila de prioridade, dando prioridade aos vértices que se encontram mais próximos do destino ao somar à distância mínima conhecida entre o vértice atual e a origem, o valor da estimativa por baixo da distância mínima do próprio vértice ao vértice de chegada.

Pode-se garantir que a solução encontrada será a ótima se, por exemplo, os pesos das arestas forem distâncias em km e a estimativa da distância do vértice atual ao destino for calculada usando a distância Euclidiana (em linha reta) entre os vértices.

Casos de utilização e funcionalidades

Nesta aplicação, pretende-se utilizar uma interface intuitiva com diversos menus onde o utilizador poderá escolher as opções que mais se adequam aos seus interesses.

O objetivo é permitir a **visualização de mapas** e a **navegação entre locais**. No primeiro caso, serão pedidas algumas **restrições espaciais** para mostrar o mapa com a informação pretendida. No segundo caso, o utilizador terá de preencher um formulário onde irá especificar as suas **preferências** tais como tempo disponível, meio de transporte a utilizar, pontos de interesse, pontos de partida e de destino.

Na navegação entre locais, após se obter a informação necessária, será indicado se existe ou não um caminho possível entre o ponto inicial e o ponto final e, caso exista, apresenta-se o **melhor caminho**, ou seja, o caminho com o maior número de pontos de interesse que possam ser visitados no tempo fornecido. Realça-se que, se não existir uma possibilidade dentro do tempo máximo escolhido, será apresentada uma mensagem de erro bem como a melhor alternativa possível.

Conclusão

Neste relatório, foi discutida uma solução possível para o problema proposto, analisando pormenorizadamente alguns algoritmos, apresentados nas aulas, que possam vir a ser utilizados como uma estratégia para o desenvolvimento do projeto.

Tendo o hábito de realizar relatórios durante ou após a implementação do código, a maior dificuldade encontrada foi conseguir organizar o processo de desenvolvimento sem de facto o implementar.

Toda a pesquisa necessária e a elaboração do relatório foram igualmente divididas pelos três membros do grupo, estando em constante comunicação e discussão. Desta forma, o esforço dedicado por cada elemento é de ⅓.

Bibliografia

- Slides das aulas teóricas
- Relatórios fornecidos pelo docente
- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- https://pt.wikipedia.org/wiki/Busca_em_profundidade
- https://pt.wikipedia.org/wiki/Busca_em_largura
- https://pt.wikipedia.org/wiki/Algoritmo_A*