

Concepção e Análise de Algoritmos

Security Van

MIEIC – Turma 5 – Grupo y

(26 de abril de 2019)

Miguel Silveira Rosa **up201706956@fe.up.pt**

Paulo Roberto Dias Mourato **up201705616@fe.up.pt**

Fellipe Ranheri de Souza **up201700127@fe.up.pt**

Índice

Descrição	3
Formalização do problema	4
Perspetiva de resolução	6
Casos de utilização	8
Conclusão	9
Bibliografia	10

Descrição

Uma empresa dedicada ao transporte de grandes quantias de dinheiro, utiliza um veículo de transportes de valores (VETV). Esta empresa planeia implementar um sistema de transporte inteligente, que descobre as rotas mais eficientes entre ponto inicial e o seu destino, podendo realizar diversas entregas durante o percurso.

A empresa possui diversos clientes (bancos, museus, correio urgente e juntas), todos espalhados pela cidade.

O novo sistema de transporte inteligente, esta decomposto em dois modos de operação.

Sendo que o VETV é um transporte de valores, o fator mais importante no sistema a implementar é diminuir o tempo entre transporte, sendo que a carga do veículo não é um fator de risco. O sistema tem assim o caminho não necessariamente mais curto em termos de distância, mas o caminho que levará menos tempo a ser percorrido.

1) Um único VETV, com carga ilimitada;

O objetivo trata-se de determinar a melhor rota desde a recolha de valores até a sua entrega, sendo a rota mais curta e desviando das obras públicas. Como temos apenas um único VETV qualquer ponto de interesse que esteja bloqueado pelas obras torna-se inacessível pelo que é um cuidado a ter quando contruindo o grafo. Caso esse ponto não esteja acessível, esse ponto será excluído da solução.

2) Diversos VETV especializados;

Nesta fase, em vez de haver apenas um VETV existem diversos VETV, cada um especializado no transporte de uma espécie de carga. Nesta fase cada um dos VETV tem um caminho distinto sendo têm pontos de interesse diferentes. Os VETV podem ter pontos de inicio diferentes. Caso haja dois VETV do mesmo tipo o VETV que demorar menos tempo (tanto o tempo a chegar ao ponto como o tempo do caminho anterior). Esta funcionalidade tem como objetivo minimizar o tempo total das entregas, fazendo com que os bens fossem entregues no mínimo de tempo possível.

Formalização do problema

Formalizando as ideias mencionadas no capítulo anterior, podemos decompor a formalização em 3 etapas.

1)Dados de entrada

Lnames – Lista de nomes de estabelecimentos a entregar valores

Ci - sequência de VETV da central, sendo $Ci(i)$ o seu i -ésimo elemento. Cada um é caracterizado por:

- Type - Tipo de VETV (na primeira iteração o tipo é All)

Gi = (Vi, Ei) - grafo dirigido pesado, composto por:

- **V** - vértices (que representam pontos da cidade) com:
 - Type - Tipo de estabelecimento ((Bancos, Museus, Finanças, Camara Municipais, Correios de risco, Tribunais)
 - Name- Nome do estabelecimento
 - Coordinates- Coordenadas do vértice (x,y)
 - Adj $\subseteq E$ - conjunto de arestas que partem do vértice
- **E** - arestas (que representam vias de comunicação) com:
 - Weight - Peso da aresta (em termos de tempo médio que demora a chegar ao destino)
 - Dest - Destino da aresta

S $\in Vi$ - vértice inicial (empresa)

2)Dados de saída

Vf - sequência ordenada de vértices que constituem o caminho mais eficiente

W - Peso total do percurso, representa o tempo que demora a percorrer o conjunto de vértices

3) Restrições

Os dados declarados anteriormente, contem as seguintes restrições:

3.1) Restrições nos dados de entrada

Para todas as arestas (**E**), o seu peso (Weight) tem de ser superior a zero, pois o peso das arestas representa o tempo que leva a percorrer até alcançar o próximo vértice. Pesos negativos significariam que o tempo voltou a traz o que não faz sentido no problema em questão.

Todos os vértices (**V**) tem de ter um tipo não nulo, ou seja, todos os vértices são pontos de potencial interesse.

3.2) Restrições nos dados de saída

O Peso final do percurso (**W**) tem de ser superior a zero devido a restrição feita ao peso das arestas nos dados de entrada.

O vértice inicial (**S**) tem de ser o primeiro membro da sequência ordenada de vértices (**Vf**)

A sequencia ordenada de vetrices (**Vf**) deve conter todos os estabelecimentos cujos nomes estão inseridos na lista de estabelecimentos (**Lnames**)

4) Funções objetivo

Como já foi descrito durante a introdução do problema, o principal objetivo da rota a escolher e minimizar o tempo percorrido (que esta descrito pelo peso final do percurso). Sendo assim a função é a função que minimiza o peso total do percurso final, como a função f:

$$f = \sum \text{Weight}(e), e \in Vf$$

Perspetiva de solução

Para a resolução deste problema necessitaremos de diversas classes e algoritmos.

Em termos de classes necessitaremos de classes a representar os vértices, as arestas, o grafo e o VETV.

A classe vértice terá o tipo e nome do estabelecimento, bem como as arestas adjacentes.

A classe aresta terá o peso, e o vertível destino.

A classe grafo terá um conjunto de vértices e arestas.

A classe VETV terá o tipo de VETV (na primeira parte será ALL).

Quanto aos algoritmos teremos de usar algoritmos que funcionem em grafos pesados, pois as nossas arestas têm um peso associado (a representar o tempo necessário para as percorrer). Como tal utilizaremos o algoritmo Dijkstra, um algoritmo ganancioso, que descobre o caminho mais curto de um vértice para os outros. Este algoritmo será especialmente útil para decidir qual o caminho mais curto entre os pontos de interesse. O tempo de execução do Dijkstra é:


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

Exemplo do pseudo código do algoritmo Dijkstra, retirado dos slides das aulas teóricas de concepção e análise de algoritmos:

Algoritmo de Bellman-Ford

```
BELLMAN-FORD (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.  for i = 1 to |V|-1 do
6.    for each (v, w) ∈ E do
7.      if dist(w) > dist(v) + weight(v,w) then
8.        dist(w) ← dist(v) + weight(v,w)
9.        path(w) ← v
10. for each (v, w) ∈ E do
11.   if dist(v) + weight(v,w) < dist(w) then
12.     fail("there are cycles of negative weight")
```

Tempo de execução:
 $O(|E| |V|)$

 **FEUP** Universidade do Porto
Faculdade de Engenharia

CAL, Algoritmos em Grafos: Caminho mais curto

Outro algoritmo que pode ser utilizado para encontrar o ponto de interesse mais próximo é o *nearest neighbor*. Este algoritmo encontra numa lista o ponto mais próximo, utilizando um algoritmo de busca ganancioso. Este algoritmo apesar de mais complexo, é um algoritmo relativamente fácil de implementar, sendo que pode utilizar o *Dijkstra* como algoritmo de busca. O tempo de execução do *nearest neighbor*, utilizando o *Dijkstra* é o seguinte:

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

O tempo de execução do algoritmo, foi adicionado ao programa e nos teste realizados, era executado em pouco tempo.

Para além das classes apresentadas anteriormente, para o funcionamento do programa serão necessárias a class Map e a class Menu.

Seguidamente encontrasse uma descrição detalhada das classes que são utilizadas no projeto (as classes fornecidas pelos professores não estarão aqui descritas).

- *Class Estrada*

Esta classe é a ligação entre dois nodes (descrição dos nodes mais a frente no relatório), e é constituída por um id (*int*), um apontador para um node destino e inicio (*Node**) e o peso entre os dois nodes (*double*).

- *Class Node*

Esta classe é um ponto no mapa, e é constituído por um id (*int*), coordenadas x e y (*float*), um *vector* que tem as estradas que saem deste ponto (*vector<Estrada* >*), um tipo que representa o que é que este ponto representa para a solução do problema (se representa um baco, museu, e, ou se é um ponto intermedio) e um indicador se este ponto já foi visitado (*bool*) e um apontador para um node (*Node**) ambos necessários na utilização do Dijkstra.

- *Class VETV*

Esta *class* é um veículo de transporte de valores e é constituído por um ponto de inicio (*Node**), um tipo a indicar que valores a transportar e um caminho solução a percorrer (*vector<Node*>*)(antes de percorrer o algoritmo este *vector* esta vazio).

- *Class Map*

Esta *class* é a abstração de um mapa e é constituído por um *graphviewer* (*Graphviewer*)(necessário para mostrar o mapa de uma forma compreensível), um *vector*

aglomerando todos os pontos, um *vector* com todos os pontos de interesse para a solução, um *vector* com todos os pontos de recolha, um *vector* com todos os pontos de entrega e um *vector* solução para cada uma das soluções, com um carro ou para diversos carros (*vector<Node*>*).

- *Class Menu*

Esta *class* contem todas as interações com o utilizador (a descrição destas interações estará presente no capítulo *Casos de utilização*) e tem como membro dado apenas um mapa (*Map*).

A conectividade do grafo não é testada, pois para a segunda solução pode existir carros em pontos que estão separadas de do conjunto principal de pontos e por isso conseguem aceder a pontos posteriormente inacessíveis. Caso um ponto esteja fora do alcance, este será ignorado e a solução irá prosseguir.

Casos de utilização

A aplicação terá uma interação com o utilizador simples, de modo a facilitar a manipulação e compreensão da mesma.

A interface permitirá o utilizador carregar um mapa sobre a forma de um grafo, que estará guardado na memória do programa. Dentro de um mapa o utilizador poderá colocar, retirar pontos de interesse existentes para o percurso selecionado, ver o percurso mais rápido para o percurso selecionado, ver todos os pontos de interesse (ou um tipo de pontos de interesse de um tipo específico) e colocar pontos de recolha/entrega. Para além disso o utilizador será capaz de colocar carros fortes no grafo e ver o caminho mais eficiente para o conjunto.

Estas interações têm como objetivo permitir ao utilizador interagir e obter informações sobre os pontos de interesse e permite ao utilizador descobrir a rota mais rápida para os pontos escolhidos.

Conclusão

Com a resolução deste projeto ganhámos uma melhor noção dos algoritmos implementados bem como das suas limitações. Para além disto, ganhámos um melhor conhecimento sobre aplicações e programas que encontram o menor caminho.

Apesar disso, durante a implementação deste projeto deparámo-nos com diversos problemas, como, por exemplo, no *nearest neighbor*, o algoritmo apesar de escolher sempre o ponto mais próximo, não tem noção do caminho em geral, podendo de vez em quando escolher um caminho com um peso total maior. Para a resolução deste problema era necessário um pré-processamento, no entanto devido a restrições de tempo isso não foi possível.

Durante a realização do trabalho, todos os membros contribuíram igualmente para o resultado final.

Bibliografia

- Slides e apresentações das aulas teóricas de concepção e análise de algoritmos
- Andrews, L. (November 2001). "*A template for the nearest neighbor problem*"
<http://www.drdobbs.com/cpp/a-template-for-the-nearest-neighbor-prob/184401449>