

Relatório Conceção e Análise de Algoritmos

Francisco Veiga, 201201604@fe.up.pt

João Cabral, up201304395@fe.up.pt

João Mota, 201303462@fe.up.pt

Faculdade de Engenharia da Universidade do Porto

março 2015

Conteúdo

1	Introdução	2
2	Problemas a abordar	2
3	Formalização do problema	3
3.1	1º problema/fase	3
3.2	2º problema/fase	4
4	Soluções implementadas	5
4.1	1º Problema/Fase	5
4.1.1	Algoritmo de Dijkstra	5
4.1.2	Algoritmo de Prim	6
4.2	2º Problema/Fase	6
5	Análise de complexidade	7
5.1	Grafo	7
5.1.1	Inserção de Vértices	7
5.1.2	Inserção de Arestas	7
5.1.3	Remoção de Vértices	7
5.2	Algoritmo de Dijkstra	7
5.3	Algoritmo de Prim	8

1 Introdução

No contexto da unidade curricular de Conceção e Análise de Algoritmos foi solicitada a resolução de um problema relacionado com a distribuição de uma rede de fibra ótica pela rede habitacional de um determinado agregado populacional.

2 Problemas a abordar

Numa primeira fase é solicitada uma aplicação que receba como dados de entrada um mapa do referido agregado populacional e produza como saída uma representação gráfica, sob a forma de um gráfico, de uma distribuição ideal da rede de fibra ótica, minimizando o comprimento das ligações utilizadas. Estabelece-se ainda como restrição adicional que cada uma das casas cobertas não pode situar-se fora de uma determinada área a ser definida por uma distância máxima à central de onde parte a rede de fibra ótica.

Numa segunda fase é solicitado que a aplicação alargue o raio de ação de cobertura da rede de fibra ótica, procurando abranger uma maior área, sendo contudo necessário que a aplicação seja capaz de detetar áreas onde a cobertura providenciada por uma única central se revele insuficiente e seja capaz de indicar a necessidade de existirem novas centrais de distribuição da rede.

3 Formalização do problema

3.1 1º problema/fase

Inputs

- Um grafo $G = (V, E)$ conexo, onde V é o conjunto das casas e E o conjunto das suas ligações e para cada aresta $(u, v) \in E$ temos $w(u, v)$ representando a distância entre as arestas u e v e $d(p)$ como sendo a distância de um caminho $p = \langle v_0, v_1, \dots, v_k \rangle$ [1, p. 624];
- Uma distancia $l : l > 0$;
- Um vértice $s : s \in V$.

Outputs

- Um grafo $G_T = (V_T, E_T)$ tal que se verifique a condição [1, p. 643]:

$$\forall v \in V (\delta(s, v) \leq l \leftrightarrow v \in V_T)$$

$$\delta(s, v) = \begin{cases} \min\{d(p) : s \rightsquigarrow^p v\} & \text{se existe caminho entre s e v} \\ \infty & \text{caso contrário} \end{cases}$$

Função Objetivo

Seja

$$x(u, v) = \begin{cases} 1 & \text{se a aresta}(u, v) \in E_T \\ 0 & \text{caso contrário} \end{cases}$$

A função objetivo é [2]

$$\min \sum_{(u,v) \in E} w(u, v) x(u, v)$$

com as restrições

$$\begin{aligned} \sum_{(u,v) \in E} x(u, v) &= |V_T| - 1 \\ \sum_{(u,v) \in (S,S)} x(u, v) &\leq |S| - 1 \quad \forall S \subseteq V_T \end{aligned}$$

3.2 2º problema/fase

Inputs

- Um grafo $G = (V, E)$, desconexo ou não, onde V é o conjunto das casas e E o conjunto das suas ligações e para cada aresta $(u, v) \in E$ temos $w(u, v)$ representando a distância entre as arestas u e v e $d(p)$ como sendo a distância de um caminho $p = \langle v_0, v_1, \dots, v_k \rangle$ [1, p. 624];

Outputs

- Um grafo $G_T = (V_T, E_T)$;
- Um conjunto de grafos $D \subseteq G_T = \{d(V_D, E_D) : \forall d, e \in D (\nexists v \in V_T (v \in d \wedge v \in e)) \wedge \nexists d, e \in D (\exists (u, v) \in E_T (u \in d \wedge v \in e))\}$
- Um conjunto $S = \{s : \exists! d \in D (s \in d)\}$.

Função Objetivo

Seja

$$x(u, v) = \begin{cases} 1 & \text{se a aresta } (u, v) \in E_T \\ 0 & \text{caso contrário} \end{cases}$$

A função objetivo é [2]

$$\min \sum_{(u,v) \in E} w(u, v) x(u, v)$$

com as restrições

$$\begin{aligned} \sum_{(u,v) \in E} x(u, v) &= |V_T| - 1 \\ \sum_{(u,v) \in (S,S)} x(u, v) &\leq |S| - 1 \quad \forall S \subseteq V_T \end{aligned}$$

4 Soluções implementadas

4.1 1º Problema/Fase

Para a resolução deste problema recorreu-se a dois algoritmos conhecidos que operam sobre grafos: o algoritmo de *Dijkstra* para cálculo das distâncias de todos os nós a uma determinada fonte e o algoritmo de *Prim* para cálculo da árvore de expansão mínima do grafo. Para a implementação da fila de prioridades recorreu-se ao *Heap de Fibonacci* da biblioteca *Boost*.

4.1.1 Algoritmo de Dijkstra

```
1: procedure DIJKSTRA(fonte, limite, grafo)
2:    $vertices \leftarrow grafo.vertices$ 
3:   for  $Q \in \text{vértices}$  do
4:      $Q.dist \leftarrow \infty$ 
5:      $Q.path \leftarrow NULL$ 
6:   end for
7:    $fonte \leftarrow 0$ 
8:    $FH.push(fonte)$ 
9:   while  $FH \neq \emptyset$  do
10:     $v \leftarrow FH.pop()$ 
11:    for  $a \in v.adjacencias$  do
12:      if  $v.dist + w < a.dist$  then
13:         $a.dist \leftarrow v.dist + w$ 
14:         $a.path \leftarrow v$ 
15:        if  $a \notin FH$  then
16:           $FH.push(a)$ 
17:        else
18:           $FH.decreaseKey()$ 
19:        end if
20:      end if
21:    end for
22:  end while
23: end procedure
24: procedure PODAR_GRAFO(grafo, limite)
25:   for  $v \in grafo.vertices$  do
26:     if  $v.dist > limite$  then
27:        $grafo.removeVertice(v)$ 
28:     end if
29:   end for
```

30: **end procedure**

O algoritmo de Dijkstra foi artilhado com a possibilidade de remover os nós cuja distância à fonte exceda um determinado limite.

4.1.2 Algoritmo de Prim

```
1: procedure PRIM(grafo)
2:   novoGrafo  $\leftarrow \emptyset$ 
3:   vertices  $\leftarrow$  grafo.vertices
4:   for Q  $\in$  vértices do
5:     Q.key  $\leftarrow \infty$ 
6:     Q.path  $\leftarrow NULL$ 
7:     Q.visited  $\leftarrow false$ 
8:   end for
9:   FH.enqueue(vertices)
10:  while FH  $\neq \emptyset$  do
11:    v  $\leftarrow$  FH.pop()
12:    novoGrafo  $\leftarrow$  path(v, v.path)
13:    v.visited  $\leftarrow true$ 
14:    for a  $\in$  v.adj do
15:      if a.key  $>$  weight(v, a) AND d.visited = false then
16:        a.key  $\leftarrow$  weight(v, a);
17:        a.path  $\leftarrow$  v
18:        a.decreaseKey()
19:      end if
20:    end for
21:  end while
22: end procedure
```

4.2 2º Problema/Fase

Para resolver este problema torna-se necessário identificar conjuntos de sub grafos desconexos. O algoritmo de Prim já identifica a árvore de expansão mínima de cada sub grafo desconexo, mas não efetua a identificação dos sub grafos desconexos em si mesmos. Tornou-se portanto necessário adaptar levemente o dito algoritmo. Em concreto, de cada vez que um vértice é removido da fila e não possui um antecessor o grafo construído até agora é guardado numa fila e começa a ser construído um novo grafo. Fica garantido o facto de não serem abandonados vértices do grafo anterior porque os vértices do próximo grafo são inicializados com uma chave infinita, só sendo retirados da fila quando todo o restante grafo já foi percorrido.

5 Análise de complexidade

5.1 Grafo¹

5.1.1 Inserção de Vértices

No contexto da nossa aplicação optamos por utilizar uma implementação *naïve* que se limita a acrescentar o vértice a um vetor de vértices, sem verificar repetição de elementos. Tem uma complexidade $O(1)$.

5.1.2 Inserção de Arestas

É necessário percorrer o conjunto de vértices para verificar a existência do destino e da fonte da aresta. Tem portanto uma complexidade $O(V)$

5.1.3 Remoção de Vértices

É necessário percorrer o conjunto de vértices para encontrar o vértice pretendido, e depois novamente para encontrar e apagar todas as arestas que apontam para este vértice. Tem por isso uma complexidade $O(V^2 + E)$.

5.2 Algoritmo de Dijkstra

Sendo V o número de vértices a analisar e E o número de arestas, o algoritmo percorre primeiramente toda a lista de vértices para inicializar valores. Esta operação tem complexidade V . Acrescenta-se, com complexidade $O(1)$, a fonte à fila de prioridade. De seguida percorre-se a fila de prioridade, recorrendo a função de extração do menor elemento da fila. Num heap de fibonacci esta operação corre em tempo $\log(V)$. Obtém-se assim para este ciclo uma complexidade $V\log(V)$. Dentro deste ciclo todas as arestas do grafo acabam por ser analisadas, sendo, quando encontrada uma aresta que constitua um caminho mais favorável, chamada a função `DecreaseKey` do Fibonacci heap, que corre em tempo constante. Obtém-se assim uma complexidade linear E .

Por último, e como característica particular do problema em causa torna-se necessário percorrer novamente a lista de vértices no sentido de remover todos aqueles cuja distância à fonte seja excessiva. Dentro deste ciclo pode ou não ser chamada a função `removeVertex`, que na nossa implementação do grafo tem complexidade $V^2 + E$, ficando este ciclo com complexidade $V^3 + VE$. Tem-se assim uma complexidade total $O(V + E + V\log(V) + V(V^2 + E)) = O(E + V(V\log V + V^2 + E))$.

¹Analisam-se exclusivamente as operações do grafo utilizadas no programa desenvolvido.

5.3 Algoritmo de Prim

Tem complexidade semelhante ao algoritmo de Dijkstra no ciclo principal. Contudo, de cada vez que um vértice é retirado da fila de prioridade o mesmo tem que ser adicionado ao grafo resultante (ou um novo grafo ser gerado, no caso de se tratar do primeiro elemento de um novo sub grafo), e uma aresta tem que ser adicionada de e para o seu predecessor. Como é possível ter referências diretas para a nova aresta e o seu predecessor esta operação pode ser feita em tempo constante $O(1)$. Ficamos assim com uma complexidade $O(E + V \log V)$.

Referências

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.
- [2] Dorit S. Hochbaum. Network flows and graphs - lecture 25. Disponível em <http://www.ieor.berkeley.edu/~ieor266/Lecture25.pdf> (2015/03/26).