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

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 $p = \langle v_0, v_1, \ldots, v_k \rangle$[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]:

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

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

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

$$\sum_{(u,v) \in E} x(u, v) = |V_T| - 1$$

$$\sum_{(u,v) \in (S,S)} x(u, v) \leq |S| - 1 \quad \forall S \subseteq V_T$$

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 $p = \langle v_0, v_1, \ldots, v_k \rangle$[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

$$\sum_{(u,v) \in E} x(u,v) = |V_T| - 1$$

$$\sum_{(u,v) \in (S,S)} x(u,v) \leq |S| - 1 \quad \forall S \subseteq V_T$$

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

```
procedure DIJKSTRA(fonte, limite, grafo)
    vertices ← grafo.vertices
    for Q ∈ vértices do
        Q.dist ← ∞
        Q.path ← NULL
    end for
    fonte ← 0
    FH.push(fonte)
    while FH ≠ ∅ do
        v ← FH.pop()
        for a ∈ v.adjacencias do
            if v.dist + w < a.dist then
                a.dist ← v.dist + w
                a.path ← v
                if a ∉ FH then
                    FH.push(a)
                else
                    FH.decreaseKey()
                end if
            end if
        end for
    end while
end procedure
procedure PODAR GRAFO(grafo, limite)
    for v ∈ grafo.vertices do
        if v.dist > limite then
            grafo.removeVertice(v)
        end if
    end for
```

```
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

```
procedure Prim(grafo)
    novoGrafo ← ∅
    vrtices ← grafo.vrtices
    for Q ∈ vértices do
        Q.key ← ∞
        Q.path ← NULL
        Q.visited ← false
    end for
    FH.enqueue(vrtices)
    while FH ≠ ∅ do
        v ← FH.pop()
        novoGrafo ← path(v, v.path)
        v.visited ← true
        for a ∈ v.adj do
            if a.key > weight(v, a) AND d.visited = false then
                a.key ← weight(v, a);
                a.path ← v
                a.decreaseKey()
            end if
        end for
    end while
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 construido 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 Grafo[1]

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 $Vlog(V)$. 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 $O(V + E + Vlog(V) + V(V^2 + E)) = O(E + V(VlogV + V^2 + E)$.

[1] Analisam-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. CEsta operação pode ser feita em tempo constante $O(V)$. Ficamos assim com uma complexidade $O(E + V(V + logV))$.

6 Análise Empírica

Para geração dos grafos a utilizar na análise empírica da performance dos algoritmos foi utilizado o modelo de *Erdős–Rényi*. Este modelo considera o grafo aleatório $G(n, p)$ como sendo um grafo de n nós em que cada par de nós único tem uma probabilidade p de ser ligado por uma aresta, também ela de peso aleatório. Os algoritmos até agora descritos foram corridos sobre grafos gerados segundo este modelo. Num primeiro momento o número de nós foi fixado em 200 e fez-se variar a conectividade dos grafos. Posteriormente fixou-se a probabilidade de cada aresta e fez-se variar o número de arestas. Para que que a aleatoriedade associada ao modelo usado tivesse impacto mínimo na análise efetuou-se cada teste 50 vezes, calculando depois a média dos tempos necessários. Os gráficos abaixo apresentam um sumário desses resultados.

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).