Análise e Síntese de Algoritmos

2º Projeto

Tomás Cunha, nº 81201, Grupo 15

1. Introdução

Este projeto tem como objetivo encontrar um ponto de encontro entre várias filiais de forma a minimizar o custo total das rotas, se existir. O problema pode ser reduzido a encontrar a menor soma dos custos dos caminhos mais curtos de todas as filiais para cada localidade, representando os caminhos como arestas de um grafo e os vértices como as localidades. Na resolução do problema utilizei a descrição do algoritmo de Johnson e da estutura de dados Min-Heap disponíveis no livro *Introduction to Algorithms*[1].

2. Descrição da solução

A solução encontrada consiste em realizar uma variação do algoritmo Johnson tomando como vértices de fonte todas as filiais. Em vez de guardar todos os caminhos mais curtos numa matriz, é apenas guardada a soma dos caminhos até cada localidade num vetor, reduzindo o espaço ocupado. No final, este vetor é percorrido para encontrar a soma mínima. Após encontrar o ponto de encontro correto, é calculado o grafo transposto do original e realiza-se o algoritmo Dijkstra a partir do ponto de encontro, de forma a obter os custos individuais dos caminhos de cada filial até ao ponto de encontro.

O algoritmo pode ser representado em pseudocódigo da seguinte forma (os algoritmos de Dijkstra e Bellman-Ford são omitidos uma vez que não foram alterados em relação aos originais):

Algorithm 1: Função principal

```
Let G ← Grafo formado a partir do input;
Let F ← Vértices de G correspondentes às filiais;
Let w(u,v) ← Função de pesos que devolve a perda entre u e v;
meeting-place, total-loss, d ← get-result(G, F, w);
if meeting-place = ∅ then
    print("N");
else
    print(meeting-place, total-loss);
    foreach v ∈ F do
        print(d[v]);
```

Algorithm 2: Descobrir o ponto de encontro, se existir, e as distâncias das filiais a este

```
function get-result(G, F, w)
    foreach f ∈ F do
        reachable[f] ← true;
        sum[v] ← 0;
    G' ← G ∪ s;
    d[v] ← 0 ∀ v ∈ V[G];
    bellman-ford(G', s, w);
    h(v) = δ(s, v) calculado pelo bellman-ford ∀ v ∈ V[G];
    w'(u, v) = w(u, v) + h(u) − h(v) ∀ (u,v) ∈ E[G];
    foreach u in F do
        dijkstra(G, u, w');
        foreach v ∈ δ'(u, v) calculados por dijkstra do
            if δ'(u, v) = ∞ then
                reachable[v] ← false;
            sum[v] ← sum[v] + δ'(u,v) + h(v) − h(u);
    meeting-place ← min({s: reachable[s] ∀ s ∈ sum});
    total-loss ← sum[meeting-place];
    if meeting-place ≠ ∅ then
        d ← dijkstra(G^T, meeting-place, w');
        foreach v ∈ F do
            d[v] ← d[v] + h(meeting-place) − h(v);
    return meeting-place, total-loss, path;
```

3. Análise Teórica

Os dados do enunciado garantem que o número de filiais é sempre muito menor do que o número de localidades. Seja F o número de filiais, V o número de vértices e E o número de arestas. Analisemos o pseudocódigo da função **get-result**, correspondente ao *Algorithm 2* da secção anterior para obter a sua complexidade:

As linhas 5, 8–9, 17, 18 e 22 são executadas em $\Theta(1)$. As linhas 2–4 têm complexidade $\Theta(F)$. A linha 6 é executada em $\Theta(V)$. A linha 7 executa o algoritmo de *Bellman-Ford*, que tem complexidade $\mathcal{O}(VE)$. Há um ciclo nas linhas 10–15 que é executado F vezes. Na linha 11, o algoritmo de Dijkstra é executado, logo a sua complexidade é $\mathcal{O}((V+E)\log V)$. Nas linhas 12–15, há um ciclo que percorre os caminhos mais curtos para todos os vértices do grafo, tendo complexidade $\Theta(V)$. Assim, as linhas 10–15 têm complexidade $\mathcal{O}(\mathrm{F}((\mathrm{V}+\mathrm{E})\log V+\mathrm{V}))$, mas uma vez que $\mathcal{O}(V)$ é majorado por $\mathcal{O}((V+E)\log V)$, é simplesmente $\mathcal{O}(F(\mathrm{V}+\mathrm{E})\log V)$. A linha 16 tem complexidade $\Theta(V)$. A linha 19 tem complexidade $\mathcal{O}((\mathrm{V}+\mathrm{E})\log V)$, na execução do algoritmo de Dijkstra, e $\mathcal{O}(V+E)$ na construção do grafo transposto. Uma vez que $\mathcal{O}(V+E)$ é majorado por $\mathcal{O}((\mathrm{V}+\mathrm{E})\log V)$, tem complexidade $\mathcal{O}((\mathrm{V}+\mathrm{E})\log V)$. As linhas 20–21 têm complexidade $\Theta(F)$.

É fácil observar que a complexidade das linhas 10–15 majora a complexidade de todas as restantes linhas, portanto o algoritmo tem complexidade $\mathcal{O}(\mathrm{F}(\mathrm{V}+\mathrm{E})\log V)$o

Analisando o pseudocódigo da função principal, observa-se que as linhas 1–3 são realizadas em $\mathcal{O}(V+E+F)$, a linha 4 é realizada em $\mathcal{O}(\mathrm{F}(\mathrm{V}+\mathrm{E})\log V)$ (como estudado no parágrafo anterior), as linhas 9–10 são executadas em $\Theta(F)$ e as restantes linhas são executadas em $\Theta(1)$.

Conclui-se então que a complexidade temporal do programa é majorada pela função **get-result**, logo a complexidade total é $\mathcal{O}(\mathrm{F}(\mathrm{V}+\mathrm{E})\log V)$.

Analisando a complexidade espacial, nota-se que é necessário $\mathcal{O}(V+E+F)$, uma vez que para além do grafo (e do seu transposto calculado na função **get-result**) apenas são necessários vectores com V ou F elementos (mais concretamente, o vetor de filiais inicializado na linha 2 da função principal, e os vetores das linhas 3e 4 da função **get-result**).

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 3rd Edition, September 2009