Universidade de Lisboa - Instituto Superior Técnico
Licenciatura em Engenharia Informática e de Computadores
Análise e Síntese de Algoritmos

2º Projeto

Nuno Amaro, 81824 Afonso Tinoco, 81861

Introdução

Com este projeto pretendemos expor um algoritmo eficiente para o problema proposto, explicar a sua implementação e fazer uma análise teórica e experimental da complexidade temporal e espacial deste. Pretendemos ainda comparar diversas implementações do algoritmo utilizando estruturas diferentes como suporte para o algoritmo de Dijkstra (Nomeadamente binary heap e fibonacci heap).

Descrição do problema

O problema descrito pedia para encontrar o local para o encontro de várias cidades que minimiza o custo das viagens desde as várias filiais. Ora, este problema pode ser descrito como um problema num grafo direcionado em que:

- cada localidade é um nó na rede
- cada rota consiste numa aresta cujo peso é o custo da rota
- o grafo não tem ciclos negativos ("não existe um percurso entre localidades que forme um ciclo com valor de perda total negativo")

Assim, o problema é equivalente ao de encontrar num grafo conexo G(V,E) com um conjunto de nós S (com $N = \#V$ e $C = \#E$ e $F = \#S$) o nó $u \in V : \min(\sum_{v \in S} \text{dist}(v, u))$ (e identificar a soma e as distancias desde cada v em S a esse u)

Algoritmo utilizado

Para encontrar a solução do problema foi utilizado o algoritmo de johnson para encontrar para cada $u \in S$ o vetor SSSP(u) e foi-se efetuando a soma de todos os SSSP(u). De seguida percorre-se o vetor das somas linearmente para encontrar o valor mínimo que corresponde ao vértice solução.

Estruturas utilizadas:

- **G**[u][**i**] - O grafo G foi representado como uma lista de adjacências (foi utilizado um std::vector (array dinâmica) em vez de std::list (lista duplamente ligada), pelo facto de a implementação do std::vector ser bastante mais eficiente que a de std::list [1]).
- **dijkstra::Q** - A fila de prioridade utilizada no algoritmo de Dijkstra. No nosso código encontram-se 5 implementações diferentes do algoritmo de Dijkstra, que variam na estrutura utilizada como suporte.

Explicação do algoritmo

O algoritmo de Johnson efetua em primeiro lugar uma repesagem às arestas utilizando o algoritmo de Bellman-Ford de modo a transformar o grafo num grafo com os caminhos minimos equivalentes mas sem arestas negativas. Começa-se por inserir um super nó x ligado a todos os nós do grafo através de arestas de peso 0 a partir do qual efetuamos o algoritmo de Bellman-Ford.
Usando as distâncias calculadas pelo algoritmo de Bellman-Ford, é efetuada a repesagem de todas as arestas. O novo peso de cada aresta será igual à soma do peso original com a distância do nó origem da aresta subtraindo a distância do nó destino da aresta. Após ser feita esta repesagem a todas as arestas do grafo, teremos um grafo com arestas de pesos positivos.
Finalmente, retiramos o nó x e efetuamos o algoritmo de Dijkstra para encontrar os caminhos mais curtos entre cada $s \in S$ e todos os outros vértices.
O nosso algoritmo utiliza um vetor P de tamanho N que contabiliza a soma das distâncias de todas os vértices $s \in S$ a cada vértice $v \in V$. Este valor é inicializado a 0's e é atualizado após cada chamada ao algoritmo de Dijkstra somando-se os valores repesados do vetor retornado pelo algoritmo de Dijkstra a este vetor.
Após ter sido obtido o valor final do vetor P, procura-se pelo vértice r com soma mínima em P. Caso o valor desse vértice seja infinito, não existe solução para o problema e a resposta é "N". Caso este vértice exista, esse vértice é a solução do problema.
Para se encontrar as distâncias das fontes $s \in S$ a r, a solução mais eficiente seria trocar a direção de todos as arestas do grafo e calcular a distância de r a todos os s. No entanto a nossa implementação simplesmente volta a correr o dijkstra todos e guarda o resultado das distâncias a r, uma vez que a complexidade temporal se mantem igual.

Prova de correção do algoritmo

No nosso algoritmo calculamos as distâncias de todos os vértices $u \in S$ a todos os vértices $v \in V$ e testamos todos os v para encontrar o que minimiza a soma das distancias desdes todos os s, pelo que é feita uma pesquisa completa e não existem soluções não consideradas, sendo que a resposta do nosso algoritmo é correta se o algoritmo de johsons for correto para grafos sem ciclos negativos.

O algoritmo de johnsons realiza um função de repesagem que transforma o grafo $G(V, E)$ com pesos negativos num grafo $G'(V, E')$ sem pesos negativos em que os caminhos minimos utilizam os mesmos vértices que no grafo original[4]. Uma vez que G' têm apenas pesos não negativos, o algoritmo de dijkstra encontra a solução ótima sempre [3], sendo possível recuperar a distancia em G de dois vértices $s, t \in V$ através da soma da repesagem de (s, t).

Análise assintótica temporal téorica do algoritmo

O passo de repesagem do Johnson's foi efetuado com o algoritmo Bellman-Ford com a heuristica de paragem. Este algoritmo tem uma complexidade $O(X_{\max}C)$, sendo $X_{\max}$ o comprimento do maior caminho mais curto. Uma vez que no pior caso $X_{\max} = N - 1$, a complexidade de pior caso deste algoritmo é $O(NC)$ [2].

O passo de SSSP a partir das fontes é executado F vezes e utiliza o algoritmo de Dijkstra cuja complexidade temporal é $O(C * f_{\text{decrease key}} + N * f_{\text{extract min}}))$, sendo f as complexidades amortizadas das operações descritas.[3]

Para a implementação com fibonacci heap, temos que as complexidades amortizadas de pior caso são: $f_{\text{decrease key}} = O(1)$ e $f_{\text{extract min}} = O(logN)$. Assim a complexidade de pior caso geral para cada dijkstra será: $O(C + NlogN)$.

Para a implementação com a binary heap, temos que as complexidades de pior caso são: $f_{\text{decrease key}} = O(logN)$ e $f_{\text{extract min}} = O(logN)$ Assim a complexidade de pior caso geral para cada dijkstra será: $O((N + C)logN)$.

Para os outros algoritmos baseados em estruturas de dados lazy. Não é executado o passo decrease key, havendo apenas um insert, sendo que pode ser retirado um mesmo vértice várias vezes da estrutura, mas o seu processamento apenas executado da primeira vez que é executado. Nesse caso a complexidade de pior caso do algoritmo é $O((N + C)logC) = O((N + C)logN)$ (pois $C \leq N^2$).

O passo da soma dos F vetores tem complexidade $O(F * N)$.

No total, o algoritmo tem complexiade: $O(f_{\text{reweight}} + 2 * Ff_{\text{Disjktra}} + f_{\text{vector sum}}) = O(f_{\text{repesagem}} + Ff_{\text{Disjktra}} + f_{\text{vector sum}})$. Para as implementações com binary heap / lazy decrease key, temos que a complexidade de pior caso será: $O(NC + F(N + C)logN + FN) = O(NC + F(N + C)logN)$. Para as implementações com fibonacci heap, temos que a complexidade de pior caso será: $O(NC + F(NlogN + C) + FN) = O(NC + F(C + NlogN))$.

Para grafos densos (C proporcional a N^2), na implementação com fibonacci heap, temos: $O(N * N^2 + FNlogN) \subset O(N^3)$ (Uma vez que F $\leq$ N). Que é a mesma complexidade que teriamos com o algoritmo de Floyd-Warshall, no entanto devido ao overhead

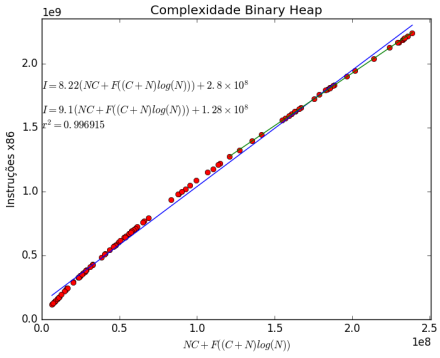
associado às estruras de dados utilizadas, a implementação com Floyd-Warshall será bastante mais rápida para grafos densos (em troca de um aumento na memória utilizada). Como para o problema em análise estamos a analisar grafos genéricos, com F < N, o nosso algoritmos tem uma eficiência bastante superior ao de Floyd-Warshall.

Análise assintótica espacial

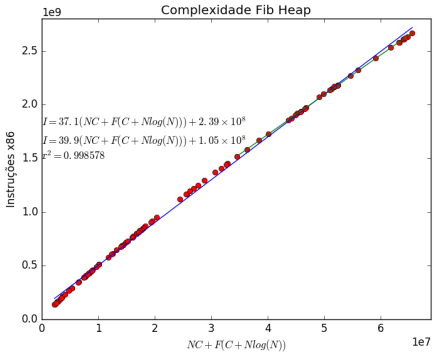
O algoritmo utiliza um número constante de estruturas de suporte que utilizam memória $O(N)$ e um número constante de estruturas de suporte de memória $O(F)$.
O algoritmos não tem passos recursivos.
No dijkstra, nas versões que utilizam o decrease key, são guardadas no máximo N estruturas $O(1)$ na memória. Nas versões lazy do dijkstra são guardadas no máximo C estruturas $O(1)$ na memória. Para representar o grafo, é utilizada uma estrutura com memória $O(N + C)$.
Assim, a complexidade espacial é $O(N + F + C)$ para ambos os casos. Se não for contabilizada a memória utilizada para representar o grafo o algoritmo tem complexidade espacial $O(N + F)$ para as representações versões com o decrease-key, e $O(N + F + C)$ para as versões lazy.

Análise assintótica temporal experimental do algoritmo

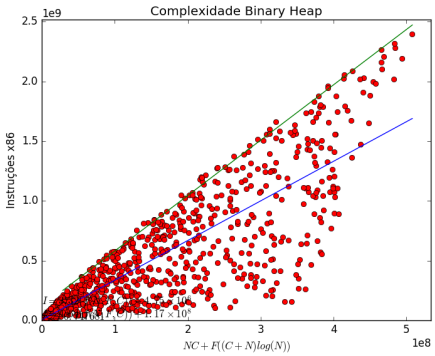
A analise experimental foi efetuada para grafos aleatórios, tendo sido obtidos os resultados das figuras deste relatório. Os grafos (a) e (b) analisam principalmente o passo do dijkstra enquanto os graficos (c) e (d) o passo do Bellman-Ford. Devido às diversas heuristicas utilizadas e ao facto de algumas das estruturas terem facilmente tempo melhor que o de pior caso (nomeadamente as filas de prioridade utilizados), verifica-se uma grande diferença entre a complexidades de pior caso e a de caso médio.



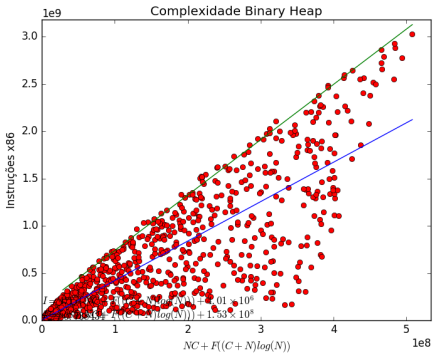
(a) Análise experimental da binary heap com parâmetros $250 \leq N \leq 1000, C = N\sqrt{N}, F = 0.95N$ (100 pontos)



(b) Análise experimental da fibonacci heap com parâmetros $250 \leq N \leq 1000, C = N\sqrt{N}, F = 0.95N$ (100 pontos)



(c) Análise experimental da binary heap com parâmetros $1 \leq N \leq 5000, C < N, F = 0.05N$ (100 pontos)



(d) Análise experimental da fibonacci heap com parâmetros $1 \leq N \leq 5000, C < N, F = 0.05N$ (100 pontos)

Referências

[1] Programming languages - c++. *ISO/IEC 14882:2003*, page 879, 2003.

[2] Charles E.; Rivest Ronald L.; Stein Clifford Cormen, Thomas H.; Leiserson. *The Bellman–Ford algorithm*, chapter 24.1. MIT Press and McGraw–Hill, 2001.

[3] Charles E.; Rivest Ronald L.; Stein Clifford Cormen, Thomas H.; Leiserson. *Dijkstra's algorithm*, chapter 24.3. MIT Press and McGraw–Hill, 2001.

[4] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, January 1977.