MO417 – Complexidade de Algoritmos

Nathana Facion - RA 191079 - Lista 8

24/05/2017

IMPORTANTE: Professor, coloquei alguns teoremas e lemas que vou fazer uso na lista para facilitar a correção. Acredito que se eu colocasse apenas os números haveria necessidade de ficar olhando slides ou livro para ver o qual estou falando.

Questão 1.

Algoritmo

Algorithm 1 Detecta zeros e ums.

```
function BFS-ALTERADA(G, s)
    for v ∈ V[G] do
        dist[v] = ∞
        π[v] = NIL
    dist[s] = 0
    Cria fila Q
    Q.enfileiraFrente(s)
    while  Q! = ∅ do
        u = Q.topo()
        for e ∈ Adj[u] do
            Relax(u, e, ω)
    return π, d

    function Relax(u, v, ω)
        if  d[v] > d[u] + ω(u, v) then
            d[v] = d[u] + ω(u, v)
            π[v] = u
            if v[weight] == 1 then
                Q.enfileiraAtras(u)
            else
                Q.enfileiraFrente(u)
```

Complexidade

A inicialização consome tempo $O(V)$. No while percorremos todos vértices e arestas de cada vértice, então temos O(V+E).

Corretude

(Esse lema foi visto no capítulo passado)

Lema 1. *No algoritmo BFS considere que os vértices vão de v_1 até v_r, então os vértices são inseridos na fila em ordem crescente e há no máximo dois valores d[v] para vértices na fila.*

Prova: por indução no número de operações ENQUEUE e DEQUEUE.
Base: Q= s. Trivial.
Passo da indução:
Caso 1: v_1 é removido de Q. Agora v_2 é o primeiro vértice de Q. Então:

$$d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$$

. As outras desigualdade são mantidas.
Caso 2: v = v_{r+1} é inserido em Q. Suponha que a busca é feita em u neste momento. Logo $d[v_1] \geq d[u]$. Então:

$$d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$$

Pela Hipótese de indução segue que $d[v_r] \leq d[u] + 1$. Logo

$$d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$$

As outras desigualdade são mantidas. □

Lema 2. *Se $d[v] < \infty$ então v pertence a árvore T induzida por $\pi[]$ e o caminho de s a v em T tem comprimento d[v].*

Prova: Indução no número de operações Enqueue.
Base: quando s é inserido na fila temos d[s] = 0 e s é a raiz da árvore T. Passo de indução: v é descoberto enquanto a busca é feita em u (percorrendo Adj[u]). Então por HI existe um caminho de s a u em T com comprimento d[u].

Como tomamos d[v] = d[u] + 1 e $\pi[v]$ = u, o resultado segue.

Prova algoritmo:

Devido a similaridade com BFS + Dijkstra, alguns detalhes não serão explicados, pois já foram provados em sala de aula. Considere um ponto na execução da BFS quando você está em um vértice arbitrário u que tem arestas de pesos 0 e 1. Similar ao Dijkstra, só colocamos um vértice na fila se ele já foi relaxado por um vértice anterior (distância é reduzida ao andar nesse vértice) e também mantemos a fila Q.

Sabemos que ao andar por uma aresta (u,v) garantimos que v está, ou no mesmo nível, ou num nível seguinte a u. Isso é verdade pois as arestas tem peso 0 e 1. Uma aresta de peso 0 significa que elas estão no mesmo nível, enquanto que uma aresta de peso 1 significa que ele está um nível abaixo. Essa é a mesma idéia usada pelo Lema 1 acima.

Também sabemos que durante a BFS nossa fila guarda vértices de dois níveis seguidos, no máximo. Então, quando estamos no vértice u, nossa fila contém os elementos do nível N[u] e N[u]+1. E também sabemos que para uma aresta (u,v), v está ou no mesmo nível de u ou no nível seguinte. Portanto, se o vértice v é relaxado e está no mesmo nível, podemos inserí-lo na frente em Q, e se ele está no nível seguinte, podemos inserí-lo atrás em Q. Como nosso algoritmo de BFS sempre desinfileira um vértice de Q sempre que este não está vazio, podemos garantir que ele pegará um vértice de menor distância para a fonte e o insere em π.

O lemma 2 é válido pois iremos enfileirar em Q todos os vértices do grafo G. Ao enfileirar, definimos a distância do vértice v para a fonte. Então, garantimos que todo vértice em G terá uma distância, que está será menor do que ∞ e que este estará na árvore

Questão 2.

Algoritmo

Adicionaremos um vértice s, que terá uma aresta direcionada apontando para cada vértice de G. Esse vértice s, está sendo adicionado para garantir que ciclos contidos em componentes desconexos possam ser detectadas por meio de bellman-ford. Além desse caso desconexo, se o vértice origem não alcança o ciclo do mesmo componente, (por exemplo um vértice sorvedouro ser vértice de inicio), também há necessidade da criação de s, para encontrarmos o ciclo negativo.

Sem esse vértice s, não haveria garantia de encontrar o ciclo negativo, pois dependeria do vértice de inicio alcançar os vértices do grafo que contem o ciclo negativo.

Teorema 3. *Teorema 24.9 (CLRS) Seja $Ax \leq b$ um sistema de restrições de diferença e seja $G = (V, E)$ o grafo de restrições associado. Se G não contém ciclos negativos, então $x = (dist(v_0, v_1), dist(v_0, v_2), \ldots, dist(v_0, v_n))$ e uma solução viável do sistema. Se G contém ciclos negativos, então o sistema não possui solução viável.*

Em nossa solução o v_0 do teorema é o vértice s criado, que tem ligação com todos os outros vértices. Como todo vértice da solução é alcançavel a partir de s, então se existir um ciclo negativo, este será detectado pelo algoritmo.

Se não existir ciclo negativo, então as distâncias computadas pelo algoritmo formam uma solução viável.

Algorithm 2 Encontra ciclo negativo, se houver.

1: **function** BELLMANFORDALTERADO(G, ω, s)
2: $\quad C = \emptyset$
3: $\quad Initialize - single - source(G, s)$
4: $\quad$ **for** $i = 1$ até $|V[G]| - 1$ **do**
5: $\quad\quad$ **for** cada aresta(u,v) $\in$ E[G] **do**
6: $\quad\quad\quad Relax(u, v, \omega)$
7: $\quad$ **for** cada aresta(u,v) $\in$ E[G] **do**
8: $\quad\quad$ **if** $d[v] > d[u] + \omega(u, v)$ **then**
9: $\quad\quad\quad v.visitado = TRUE$
10: $\quad\quad\quad atual = v$
11: $\quad\quad\quad C = \{v\}$
12: $\quad\quad\quad$ **while** $\pi[atual].visitado == FALSE$ **do**
13: $\quad\quad\quad\quad atual = \pi[atual]$
14: $\quad\quad\quad\quad atual.visitado = TRUE$
15: $\quad\quad\quad\quad C = C \cup \{atual\}$
16: $\quad\quad\quad return\ FALSE$,d, C
17: $\quad return\ TRUE$, d, π
18:
19: **function** $Initialize - single - source(G, s)$
20: $\quad$ **for** cada vértice v $\in$ V[G] **do**
21: $\quad\quad d[v] \leftarrow \infty$
22: $\quad\quad \pi[v] \leftarrow NIL$
23: $\quad d[s] \leftarrow 0$
24:
25: **function** $Relax(u, v, \omega)$
26: $\quad$ **if** $d[v] > d[u] + \omega(u, v)$ **then**
27: $\quad\quad d[v] \leftarrow d[u] + \omega(u, v)$
28: $\quad\quad \pi[v] \leftarrow u$
29:

=0

Complexidade

Sem levar em consideração o vértice adicionado:
Temos $O(V)$ para percorrer todos os vértices na linha 4. Dentro deste for, na linha 4, existe outro for, na linha 5, que verifica todas as arestas assim, $O(E)$. Então a complexidade é $O(VE)$. Na linha 7 temos $O(E)$ operações, pois percorreremos todas as arestas, como no máximo o ciclo negativo conterá todos os vértices então teremos $O(VE)$.
Levando em consideração o vértice adicionado: Como V e E foram alterados, então temos na verdade V_1 e E_1 assim a complexidade pode ser analisada de outra forma. Seja A uma matriz de entrada que é V x E. Então o grafo de restrições G possui V_1 = V + 1 vértices e E_1 = E + V arestas. Assim, usando o

algoritmo de Bellman-Ford-Alterado podemos encontrar uma solução em tempo $O(V_1E_1)$ ou $O((V+1)(V+E)) = O(V^2+VE)$

Corretude

Lema 4. *Seja G um grafo orientado ponderado. Seja V um conjunto de vértices e seja E seu conjunto de arestas. Seja ω o peso de cada aresta e a função $\omega : E \longrightarrow \mathbb{R}$. Considere a origem de G em s. Após $|V|$ - 1 iterações do loop for das linhas 3 a 5 de BELLMAN-FORD, temos d[v] $= \delta(s,v)$.*

Corolário 1. *Seja G um grafo orientado ponderado. Seja V um conjunto de vértices e seja E seu conjunto de arestas. Seja ω o peso de cada aresta e a função $\omega : E \longrightarrow \mathbb{R}$. Considere a origem de G em s. Então, para cada vértice v $\in$ V, existe um caminho de s para v se e somente se BELLMAN-FORD termina com d[v] $< \infty$ quando é executado sobre G.*

Teorema 5. *Seja G um grafo orientado ponderado. Seja V um conjunto de vértices e seja E seu conjunto de arestas. Seja ω o peso de cada aresta e a função $\omega : E \longrightarrow \mathbb{R}$. Considere a origem de G em s. Seja o algoritmo BELLMAN-FORD, executado sobre esse grafo G. Se G não contém nenhum ciclo de peso negativo que seja acessível a partir de s, então a resposta retornada será TRUE, então d[v] $= \delta(s,v)$ para todos os vértices $v \in E$, e o subgrafo predecessor G_π é uma árvore de caminhos mais curtos com raiz em s. Caso G contenha um ciclo de peso negativo acessível a partir de s, então o retorno é FALSE.*

Diferenças de algoritmos:

O Algoritmo de BELLMAN-FORD já foi provado em sala de aula,com isso o Lema, Corolário e Teorema acima são válidos. Então apenas provaremos o que é diferente entre este algoritmo alterado, ou seja, o retorno do ciclo negativo.

Se não conseguirmos relaxar nenhuma aresta na linha 8, então não existirão ciclos negativos em G. Se conseguirmos diminuir d[v] usando a aresta (u,v), então sabemos que existe um ciclo negativo e que o vértice u, ou o ancestral de u, fazem parte deste ciclo negativo. Portanto, se seguirmos os ancestrais de u em π, teremos um caminho $P = (v_0, v_1, v_2, ..., v_k = u)$ de tamanho k, e marcamos como visitado cada vértice em P.

Seguindo do resultado de Bellman-Ford de que, se a condição na linha 8 é verdadeira teremos um ciclo negativo, teremos então que um vértice v_x de P terá como ancestral um vértice v_y de P que já foi visitado, e portanto, fechando um ciclo. Caso P não contenha um ciclo negativo de G, então teríamos que o teorema 5 de Bellman-Ford seria falso.

Questão 3.

Algoritmo

Algorithm 3 Identifica ferrovias que podem ser removidas

```
function DIJKSTRA-ALTERADA(G, s)
    G é um grafo com todas as cidades, ferrovias e estradas.
    s é um vértice inserido no grafo que se liga a todas as capitais.
    f contem conjunto de ferrovias
    arestaIncidente vetor de arestas incidentes
    arestaIncidente inicializa como nil para todo vértice
    contFerrovia conta ferrovias usadas, valor inicial zero.
    ehCapital é inicializada como True quando existe capital naquele vértice, senão False.
    for v ∈ V[G] do
        d[v] = ∞
        π[v] = NIL

    S = ∅
    Q = V[G]
    while Q! = ∅ do
        u = EXTRACT − MIN(Q)
        S = S ∪ {u}
        for v ∈ Adj[u] do
            Relax(u, v, ω)
    for v ∈ V[G] do
        if temferrovia[v] and not ehCapital[v] then
            contFerrovia = contFerrovia + 1    ▷ Conta as ferrovias usadas
    //Quantidade de ferrovias para remover
    quantidadeRemover = |f| − contFerrovia
    // Quais ferrovias devem ser removidas
    quaisRemover = f − arestaIncidente
    return π, d, quantidadeRemover, quaisRemover

    function Relax(u, v, ω)
        Empate = (d[v] == d[u] + ω(u, v)) AND (u, v).type == RODOVIA
        IF d[v] > d[u] + ω(u, v) OR Empate THEN
            d[v] = d[u] + ω(u, v)
            π[v] = u
            IF (u, v).type == FERROVIA THEN
                temferrovia[v] = TRUE
                arestaIncidente[v] = (u, v)
            ELSE
                temferrovia[v] = FALSE
                arestaIncidente[v] = NULL
```

Complexidade

A inicialização da linha 9 a 11 tem complexidade O(V). Pois é percorrido todos vértices.

Na inserção em Q, são feitas V chamadas.

O laço da linha 15 vai executar O(V) vezes a chamada EXTRACT-MIN. Na linha 18-19 é executado O(E) vezes a chamada DECREASE-KEY implicitamente. O laço da linha 20, percorre todos os vértices, portanto O(V).

Como estamos fazendo uso do heap então o insert, extract-min e decrease-key gastam tempo O(lgV), resultado em complexidade O((V+E)lgV).

Corretude

Criamos um vértice s, com pesos zeros nas arestas que deve ser ligado as capitais do grafo G. Ele está sendo inserido para facilitar o uso do Dijkstra, já que desta forma fica mais simples de ver que o algoritmo funciona, do que adicionando várias fontes em cada capital.

Adicionamos a variável referente as arestaIncidente, pois a mesma é necessária para sabermos as arestas que foram usadas. A variável contFerrovia tem a finalidade de contar as ferrovias usadas.

Primeiro, no for da linha 9 inicializamos as distâncias de todas as cidades com infinito e colocamos π como NIL.

Depois no while, o EXTRACT-MIN retorna o elemento com a menor distância no grafo, que retornará todas as capitais antes das outras cidades. Então, checamos as adjacências do vértice retornado e atualizamos as distâncias.

Nesse algoritmo, cada aresta tem um tipo, RODOVIA ou FERROVIA. Nessa implementação do RELAX, também temos uma checagem para o caso de a distância nova for igual à anterior. Nesse caso, damos preferência ao caminho que usa RODOVIA. Dentro do IF, atualizamos as distâncias e o estado da variável temFerrovia, para refletir se o caminho que é o melhor para aquele vértice usa uma ferrovia ou não. Se temos ferrovia, adicionamos a aresta em arestaIncidente.

Depois do while, iremos contar o número de cidades cujo menor caminho contém uma ferrovia. Esse será o número de ferrovias que estão sendo usadas. Portanto, o número total de ferrovias no grafo, menos as ferrovias sendo usadas, é o número de ferrovias que podem ser desativadas. E o conjunto de ferroviais, f, menos as arestas adjacentes que são ferrovias informam quais arestas podem ser desativada.

Podemos garantir que o algoritmo funciona pois, como usamos uma variante do Dijkstra, este retorna uma árvore de caminhos mínimos, ou seja, retorna o menor caminho de uma cidade até sua capital mais próxima e usando o menor número possível de ferrovias. Quando chegamos na checagem da linha 34, temos que o menor caminho da cidade v até a capital mais próxima, começa a partir de uma aresta que pode ou não ser uma Ferrovia, marcada pelo vetor $temferrovia$. Como realizamos essa verificação para todas as cidades, e o algoritmo só alcança essa verificação quando encontra um caminho menor, ou que use uma rodovia em lugar de uma ferrovia, basta olharmos a primeira aresta do caminho mínimo de cada cidade, que cobriremos todas as ferrovias que estão sendo usadas.

Ao final, só precisamos contar quantas cidades em que $temferrovia$ é $TRUE$ e subtrair do total de ferrovias, assim teremos quantas ferrovias podem ser retiradas. Como a variável $temferrovia$ foi atualizada no mesmo local que a $arestaIncidente$, então também temos todas as arestas usadas, e as ferrovias totais menos as arestas incidentes retornam o quanto podem ser removidas.

Pelas invariantes de Dikstra abaixo que são aplicadas ao nosso algoritmo, pois é uma versão alterada do original, podemos afirmar que o caminho mínimo para todos os vértices será encontrado.

Invariantes de Dikstra:

- Se $d[x] < \infty$ então $\pi[x] \in$ S.
- O conjunto (x, $\pi[x]$) : x $\in$ S - {s} induz uma árvore com conjunto de vértices S.
- Para cada vértice x em S, o caminho de s a x na árvore tem peso $d[x]$.
- dist(s, x) $\leq d[x]$ para cada vértice x em S.
- e (x, y) $\in E[G]$, x $\in$ S e y $\in V[G]$ - S então d[y] $\leq d[x] + \omega$(x, y).
- Isto vale pois quando x foi inserido em S, foi executado Relax(x, y, ω) e depois disso, $d[x]$ nunca muda e $d[y]$ nunca aumenta.