Departamento de Ciência de Computadores FCUP

Desenho e Análise de Algoritmos (CC211) 2012/13

2º Teste Escrito (12.12.2012) *duração: 3h*

Cotação: (2, 1.5+1+0.5, 1.5+1.5+2, 2, 2, 2, 0.5+2+1+0.5, 1.5+0.5)

UMA RESOLUÇÃO DETALHADA
(terá de ser adaptada para a outra versão do enunciado)

1. Utilizando os símbolos $\subseteq$ ou $\subset$, relacione as classes $\Theta(n^2 \log n)$, $O(n^4)$, $\Omega(n^2)$ e $O(1)$, se comparáveis. Justifique formalmente a resposta, recorrendo às definições das notações O, Θ e Ω.

Resposta:

As ordens de grandeza O, Θ e Ω são assim definidas:

$$\begin{aligned} O(g(n)) &= \{f(n) \mid \text{existem } c > 0 \text{ e } n_0 > 0 \text{ tais que } f(n) \leq cg(n) \text{ para todo } n \geq n_0\} \\ \Omega(g(n)) &= \{f(n) \mid \text{existem } c > 0 \text{ e } n_0 > 0 \text{ tais que } f(n) \geq cg(n) \text{ para todo } n \geq n_0\} \\ \Theta(g(n)) &= \{f(n) \mid \text{existem } c_1 > 0,\ c_2 > 0 \text{ e } n_0 > 0 \text{ tais que } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\} \end{aligned}$$

- $\Theta(n^2 \log n) \subset \Omega(n^2)$ porque se $f(n) \in \Theta(n^2 \log n)$ então existem constantes $c_1 > 0$ e $n_0 > 0$ tais que $f(n) \geq c_1 n^2 \log n$ para todo $n \geq n_0$. Como $n^2 \log n \geq n^2$ para $n \geq 10$, concluímos que, se $n_0' = \max(n_0, 10)$ então $f(n) \geq c_1 n^2 \log n \geq c_1 n^2$ para $n > n_0'$. Logo, se $f(n) \in \Theta(n^2 \log n)$ então $f(n) \in \Omega(n^2)$, qualquer que seja a função $f(n)$. Portanto, $\Theta(n^2 \log n) \subseteq \Omega(n^2)$, mas a inclusão é estrita porque, por exemplo, a função $h(n) = n^2$ pertence a $\Omega(n^2)$ mas $h(n) \notin \Theta(n^2 \log n)$. Sabemos que, qualquer que seja $c > 0$, se $n > 10^{1/c}$ então $n^2(1 - c \log n) < 0$ (ou seja, $n^2 < cn^2 \log n$). Assim, $h(n) \notin \Omega(n^2 \log n)$ e, consequentemente, $h(n) \notin \Theta(n^2 \log n)$.
- $\Theta(n^2 \log n) \subset O(n^4)$ porque se $f(n) \in \Theta(n^2 \log n)$ então existem $c_2 > 0$ e $n_0 > 0$ tais que $f(n) \leq c_2 n^2 \log n$ para todo $n \geq n_0$. Como $c_2 n^2 \log n \leq c_2 n^4$ para todo $n \geq 1$, concluimos que se $f(n) \in \Theta(n^2 \log n)$ então $f(n) \in O(n^4)$. A inclusão é estrita porque, por exemplo, a função $h(n) = n^4 \in O(n^4)$ mas $h(n) \notin \Theta(n^2 \log n)$, dado que $n^4 > cn^2 \log n$ para $n > \max(c, 10)$, qualquer que seja $c > 0$ (se aqui, $\log n$ for $\log_{10}(n)$).
- $O(1) \subset O(n^4)$ porque se $f(n) \in O(1)$ então existem constantes $c > 0$ e $n_0 > 0$ tais que $f(n) \leq c$. Logo, $f(n) \leq cn^4$ para todo $n \geq n_0$, e portanto $f(n) \in O(n^4)$. A inclusão é estrita porque, por exemplo, a função $h(n) = n^4$ pertence a $O(n^4)$ mas não a $O(1)$.

Nos restantes casos, as classes não são comparáveis. Por exemplo, $\Theta(n^2 \log n) \not\subseteq O(1)$ pois $h(n) = n^2 \log n \notin O(1)$. Também $O(1) \not\subseteq \Theta(n^2 \log n)$ porque se $f(n) \in O(1)$ então $f(n) \notin \Omega(n^2 \log n)$ e portanto $f(n) \notin \Theta(n^2 \log n)$...

Habitualmente, em Matemática, $f(n)$ designa a imagem de n pela função f, mas, nesta definição, $f(n)$ designa também a função $f : \mathbb{N} \to \mathbb{R}$ que a cada n associa $f(n)$. Do mesmo modo, $g(n)$ designa a função $g : \mathbb{N} \to \mathbb{R}$ que a cada n associa $g(n)$. As notações $f(n)$ e $g(n)$ estão a ser usadas com as duas interpretações. Por exemplo, em $f(n) \in O(g(n))$ referimos as funções f e g e em $f(n) \leq cg(n)$ estamos a referir as imagens de n por f e por g.

$f(n) \in O(g(n))$ sse $f(n)$ é majorada por $cg(n)$ para alguma constante $c > 0$ a partir de uma certa ordem.

$f(n) \in \Omega(g(n))$ sse $f(n)$ é minorada por $cg(n)$ para alguma constante $c > 0$ a partir de uma certa ordem.

$f(n) \in \Theta(g(n))$ sse $f(n)$ é majorada por $c_2 g(n)$ e minorada por $c_1 g(n)$ para algum $c_1 > 0$ e algum $c_2 > 0$ a partir de uma certa ordem.

2. Considere uma estrutura de dados semelhante à dada nas aulas para implementação de uma heap de mínimo (com informação adicional). A estrutura tem três campos x, y, e n, sendo x e y *arrays* e n um inteiro que indicará o número de elementos que estão na heap. Cada elemento de x tem um par de valores $p : id$ em que p define a prioridade desse elemento e id um identificador que lhe está associado. O elemento que está na posição v de y tem o índice da posição que v ocupa na heap. Se $y[v] = 0$ então v não está na heap. Nem $y[0]$ nem $x[0]$ serão usadas.

a) Apresente em pseudocódigo um algoritmo para alterar a prioridade de um elemento v para *novap*, sabendo que *novap* é menor do que o seu valor atual. Deve ter complexidade temporal $O(\log_2 n)$ e

preservar as propriedades de uma heap de mínimo e da estrutura definida. Use, por exemplo, $x[k].p$ e $x[k].id$, $y[v]$ e n para referir variáveis, omitindo a identificação da estrutura correspondente.

Resposta:

O algoritmo é semelhante ao algoritmo da função `decreaseKey` dada nas aulas.

```
i ← y[v];
x[i].p ← novap;
Enquanto (i > 1 ∧ x[i].p < x[i/2].p) fazer
    // trocar o elemento i com o seu pai
    pai ← i/2;
    y[x[pai].id] ← i;
    y[v] ← pai;
    aux ← x[i];
    x[i] ← x[pai];
    x[pai] ← aux;
    i ← pai;
```

b) Suponha que tem doze elementos numerados de 1 a 12, e que o estado da estrutura é o seguinte com $n = 11$.

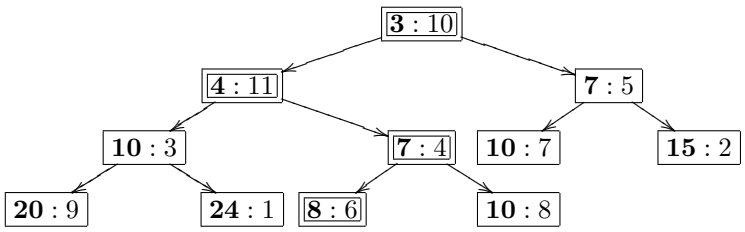
x :	–	3 : 10	4 : 11	7 : 5	10 : 3	7 : 4	10 : 7	15 : 2	20 : 9	24 : 1	8 : 6	10 : 8	

y :	–	9	7	4	5	3	10	6	11	8	1	2	0

Desenhe a árvore correspondente e justifique que se trata de uma heap de mínimo. Desenhe também a árvore que resulta da aplicação do algoritmo que apresentou na alínea anterior se $v = 6$ e $novap = 3$, e assinale na árvore o(s) elemento(s) testado(s) ou alterado(s) na transformação.

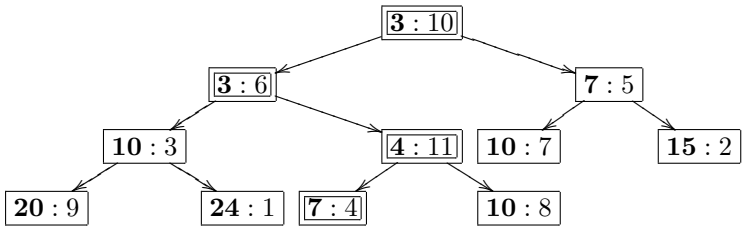
Resposta:

Para desenhar a árvore, vamos usar a propriedade de que o pai do nó i está na posição $i/2$ da heap e a raíz na posição 1 (o que corresponde a tomar os elementos de x de 1 a n e preencher todos os níveis da árvore da esquerda para a direita).



Trata-se de uma heap de mínimo porque, podemos ver que, para qualquer sub-árvore t, a chave que define a prioridade da raíz de t é menor ou igual que as dos restantes elementos de t. Nos nós, as prioridades p, ou seja, as chaves, estão à esquerda (como no enunciado, seguiu-se o formato $p : id$ nas etiquetas).

Os nós assinalados por caixa dupla são os que vão ser testados ou alterados. A árvore final encontra-se a seguir.



c) Use propriedades das heaps para mostrar que a complexidade do algoritmo é $O(\log_2 n)$.

Resposta:

O bloco de instruções do ciclo "enquanto" tem complexidade $\Theta(1)$. Logo, no pior caso, $T(n) = \beta + \alpha h_n$, para constantes e $\alpha > 0$ e $\beta > 0$, sendo h_n a altura da árvore binária e, consequentemente, $T(n) \in O(h_n)$. Como se sabe que a árvore é perfeitamente equilibrada, $h_n = \log_2 n$. Logo, $T(n) \in O(\log_2 n) = O(\log n)$.

3. Considere uma variante do problema "Caixotes de morangos", com m caixas para distribuir e n lojas, conhecendo o lucro do envio de c caixas para a loja i, para $1 \leq i \leq n$ e $1 \leq c \leq m$ (com lucros inteiros positivos). Há que distribuir os caixotes pelas lojas de forma a maximizar o lucro.

```
1  | t[0] ← 0;
2  | ler_vec(y, m);      // ler y[1]...y[m]
3  | Para cada k ← 1 até m fazer
4  |     t[k] ← y[k];
5  | Para cada i ← 2 até n fazer
6  |    ler_vec(y,m);    // loja i
7  |    j ← m;
8  |    Enquanto j ≥ 1 fazer
9  |       Para cada k ← 0 até j − 1 fazer
10 |          Se t[j] < t[k] + y[j − k] então
11 |             t[j] ← t[k] + y[j − k];
12 |       j ← j − 1;
```

a) Apresente a recorrência $t(j, i)$ que define o lucro máximo que se pode obter quando se distribui j caixas pelas primeiras i lojas, com para $1 \leq i \leq n$ e $0 \leq j \leq m$. Explique.

b) O algoritmo dado à esquerda calcula o valor de $t(m, n)$, quando os lucros são dados por lojas. Justifique a correção do algoritmo, começando por definir com rigor o estado das variáveis t e y no final de cada iteração do ciclo "Enquanto" e do segundo ciclo "Para". **NB: Não é necessário uma prova por indução.**

c) (*perg. alternativa a 8.)) Caracterize a complexidade do algoritmo em função de m e n. Explique os detalhes.

Resposta:

(a) Sendo $y_{i,j}$ o lucro do envio de j caixas para a loja i, tem-se:

$$\begin{array}{lcl} t(0,i) & = & 0, \text{ para } 1 \leq i \leq n \\ t(j,1) & = & y_{1,j}, \text{ para } 1 \leq j \leq m \\ t(j,i) & = & \max\{t(k,i-1) + y_{i,j-k} \mid 0 \leq k \leq j\}, \text{ para } 2 \leq i \leq n,\ 1 \leq j \leq m. \end{array}$$

A primeira relação diz que o lucro é 0 se não distribuir qualquer caixote. A segunda, diz que se se considerar só a loja 1, o lucro máximo que se obtem em cada caso é aquele que a loja 1 der. A terceira diz que, se $i \geq 2$ então $t(j, i)$ é o máximo dos lucros máximos das alternativas possíveis, as quais são determinadas pelo envio de $j - k$ caixotes à loja i, sendo os restantes k caixotes distribuídos pelas lojas $1, \ldots, i-1$ de forma óptima, com $k \leq j$.

(b) Este algoritmo é idêntico ao implementado nas aulas para este problema. Denotemos por `t` e `y` as variáveis usadas no algoritmo (em vez de t e y). À entrada do **segundo** ciclo "Para", `t` contém $t(c, 1)$, para $0 \leq c \leq m$. Em cada iteração desse ciclo, a loja i está fixa, e no final da iteração i, os vetores `t` e `y` contêm $t(c, i)$ e $y_{i,c}$ na posição c, com $0 \leq c \leq m$ (a justificação está abaixo). À saída do ciclo, `t` contém $t(c, n)$, para $0 \leq c \leq m$, e, assim, `t[`m`]` contém $t(m, n)$. O estado de `y` é o mesmo à entrada e saída do ciclo "Enquanto", e contém os lucros $y_{i,c}$, para $0 \leq c \leq m$. Em cada iteração do ciclo "Enquanto", i e j estão fixos, e é calculado um novo valor $t(i, j)$ que fica guardado em `t[`j`]`.

Para i e j fixos, o **terceiro** ciclo "Para", altera o valor de `t[`j`]` substituindo $t(j, i-1)$ por $t(j, i)$. Nesse ciclo k varia de 0 a $j-1$, pois se k fosse j, teriamos $t(k, i-1)+y_{i,j-k} = t(j, i-1)+y_{i,0} = t(j, i-1)+0 = t(j, i-1)$. Mas, sendo `t[`$j$`]` igual a $t(j, i-1)$ à entrada do ciclo, esse valor é também tomado em conta na determinação do máximo (na primeira comparação). No final do **terceiro** ciclo "Para", `t[`j`],t[`$j+1$`], ...,t[`m`]` guardam $t(j, i), t(j+1, i), \ldots, t(m, i)$ e `t[0],t[1],t[2], ...,t[`$j-1$`]` guardam $t(0, i-1), t(1, i-1), t(2, i-1), \ldots, t(j-1, i-1)$. A posição `t[0]` não é alterada ao longo do algoritmo depois de ter sido inicializada com zero. Logo, `t[0]` $= 0 = t(0, i-1) = t(0, i)$.

No **segundo** ciclo "Para", para i fixo, a actualização dos valores de `t[`j`],t[`$j+1$`], ...,t[`m`]` foi efetuada antes do cálculo dos valores $t(1, i), t(2, i), \ldots, t(j-1, i)$. Isso é correto porque para todo j' tal que $j' < j$, o valor de $t(j, i)$ só depende dos valores de $t(j', i-1)$ e de $y_{i,j-j'}$, para $0 \leq j' \leq j$, e esses valores estão disponíveis ainda.

(c) A complexidade do algoritmo é $\Theta(nm^2)$ para $n \geq 2$ e é $\Theta(m)$ se $n = 1$.

A complexidade da instrução ler_vec(y, m) é $\Theta(m)$ e é idêntica à do primeiro ciclo "Para". Logo, a complexidade do bloco 1–4 é $\Theta(m)$. Se $n = 1$, então $\Theta(m)$ é a complexidade final pois a inicialização de i e a verificação da condição de paragem do ciclo "Para" (linha 5) é $\Theta(1)$.

O bloco de instruções 7–12 tem complexidade $\Theta(m^2)$ porque o terceiro ciclo "Para" (linhas 9-11) tem complexidade $\Theta(j)$, dado que, no melhor e pior caso, o bloco que se executaria em cada iteração desse ciclo "Para" tem complexidade constante.

O bloco 6–12 é executado $(n-1)$ vezes, e em cada iteração a complexidade é $\Theta(1) + \Theta(m) + \Theta(m^2) = \Theta(m^2)$. Aqui, $\Theta(1)$ corresponde às operações de actualização do valor de i e teste da condição de fim de ciclo. Assim, sendo $n \geq 2$, a complexidade do algoritmo é $\Theta(nm^2)$, porque $\Theta((n-1)m^2) = \Theta(nm^2)$ assintoticamente.

4. Que problema resolve o algoritmo de Prim em grafos não dirigidos com pesos nos ramos? O que é um grafo conexo? De que modo o algoritmo de Prim pode ser usado para verificar se o grafo dado é conexo? Que vantagem ou desvantagem tem face ao algoritmo de pesquisa em largura? E, ao algoritmo de pesquisa em profundidade? E, ao algoritmo de Kosajaru (se se começar por transformar o grafo num grafo dirigido simétrico)?

Resposta:

O algoritmo de Prim determina uma árvore de cobertura com peso total mínimo para um grafo conexo (pode ser adaptado para determinar uma árvore de cobertura com peso máximo).

Um grafo conexo é um grafo em que qualquer vértice é acessível de qualquer outro vértice. A versão do algoritmo de Prim dada nas aulas pode ser usada para verificar se um grafo é conexo. O grafo não é conexo se a chave de algum vértice retirado da fila de prioridade for ∞ (caso contrário, é conexo).

O algoritmo de Prim tem uma complexidade maior do que o algoritmo de pesquisa em largura pois as operações de remoção de um elemento da fila e de redução da chave de um elemento não são realizadas em tempo $O(1)$. Os algoritmos de pesquisa em largura e em profundidade também permitem determinar se o grafo é ou não conexo. Ambos resolvem o problema em tempo $O(m+n)$, sendo $m = |E|$ e $n = |V|$ o número de arestas e de vértices. Basta considerar um vértice s qualquer, realizar uma visita a partir de s (o que permite descobrir quais os vértices acessíveis de s), e verificar se ficou algum vértice por visitar, isto é, com cor "branca". Em qualquer um dos casos, estamos a supor que o grafo é representado por listas de adjacências e como um grafo dirigido (simétrico). Embora o algoritmo de Kosaraju (para determinação das componentes fortemente conexas) tenha a mesma complexidade assintótica, não seria o mais adequado pois iria realizar duas visitas quando bastaria uma.

5. Justifique que qualquer caminho mínimo de x para y num grafo com distâncias positivas é ou um arco (x, y) ou é constituído por caminhos mínimos. Explique de que modo esta propriedade é explorada na determinação dos caminhos mínimos nos algoritmos de Dijkstra (de um nó para todos), Floyd-Warshall (de todos para todos) e Bellman-Ford (de todos para todos).

Resposta:

Assumimos que x e y são distintos, pois a distância de x a x foi definida como 0 e corresponde a não sair de x. Sendo as distâncias positivas, nenhum percurso de x para y que contenha um ciclo é mínimo (se não, se retirassemos esse ciclo, obteriamos um percurso menor, o que é absurdo). Assim, os percursos mínimos não passam duas vezes por nenhum vértice, pelo que são *caminhos* (i.e., caminhos simples) e têm *um ou mais arcos*.

Seja $\gamma^\star_{xy}$ o caminho mínimo de x para y. Se $\gamma^\star_{xy}$ tiver dois ou mais arcos, então terá um vértice intermédio z (que não é x nem y) e $\gamma^\star_{xy}$ é da forma $\gamma_{xz}\gamma_{zy}$, em que γ_{xz} é um caminho de x para z e γ_{zy} um caminho de z para y. Se γ_{xz} não fosse mínimo, existiria um caminho γ'_{xz} mais curto. Então, $\gamma'_{xz}\gamma_{zy}$ seria um percurso de x para y mais curto do que $\gamma^\star_{xy}$, o que é absurdo. Portanto, γ_{xz} tem de ser um caminho mínimo de x para z. Do mesmo modo se conclui que γ_{zy} tem de ser um caminho mínimo de z para y.

Esta propriedade está subjacente à aplicação de programação dinâmica nos algoritmos referidos e é usada assim:

- Algoritmo de Dijkstra: para cada nó v, mantém uma estimativa $d[v]$ da distância mínima $\delta(s, v)$ da origem s ao nó v. Esta estimativa seria igual à distância mínima de s a v se os caminhos só pudessem ter como vértices intermédios os que já saíram da fila. O critério "greedy" de saída da fila, conjuntamente com a atualização das estimativas, garante que $d[v] = \delta(s, v)$ quando v é retirado da fila. A atualização das estimativas $d[w]$ para $w \in Adjs[v]$ baseia-se na propriedade enunciada (nomeadamente, na relação $\delta(s, w) \leq \delta(s, v) + c(v, w)$, sendo $c(v, w)$ o valor no arco (v, w)).
- Algoritmo de Floyd-Warshall: considerando $V = \{v_1, \ldots, v_n\}$ ordenado, mantém uma estimativa $D^k(u, v)$ da distância mínima de u até v que seria igual à distância mínima se os caminhos só pudessem ter como vértices intermédios $v_1, \ldots, v_k$ (ou nenhum). Explora a propriedade quando usa relação $\delta(u, v) \leq D^{k+1}(u, v) = \min(D^k(u, v), D^k(u, v_{k+1}) + D^k(v_{k+1}, v)) \leq D^k(u, v)$ para melhorar as estimativas, com k a tomar valores de 1 a n. Quando k for n, todo o vértice pode ser vértice intermédio e, assim, $\delta(u, v) = D^n(u, v)$.
- Algoritmo de Bellman-Ford (todos para todos): mantém uma estimativa $\tilde{D}^r(u, v)$ da distância mínima que seria igual à distância mínima se restringissemos a r o número máximo de ramos que o caminho podia ter. Explora a propriedade quando usa relação $\delta(u, v) \leq \tilde{D}^{r+1}(u, v) = \min_{1 \leq k \leq n}(\tilde{D}^r(u, v_k) + \tilde{D}^1(v_k, v)) \leq \tilde{D}^r(u, v)$ para melhorar as estimativas, com r a tomar valores de 1 a $n-1$. Um caminho mínimo não pode ter mais do que $n-1$ ramos (se não, conteria ciclos) e, assim, $\delta(u, v) = \tilde{D}^{n-1}(u, v)$.

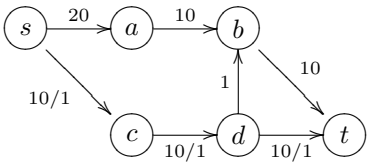
  O algoritmo de Bellman-Ford (um para todos) que calcula $\delta(s, v)$ para todo v, e que pode ser aplicado quando existem pesos negativos, segue uma ideia semelhante, realizando uma última iteração no fim para verificar se o grafo tem ciclos com peso total negativo. Mas, não era o pedido.

6. Dê um exemplo de uma rede de fluxo (com seis nós) para um problema de fluxo máximo (de um nó s para t). Represente um fluxo f na rede que não seja óptimo e tal que o algoritmo de Edmonds-Karp obteria o fluxo óptimo em **duas** iterações a partir de f. Explique (e ilustre).

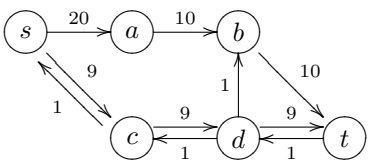
Resposta:

Pelo teorema de Ford-Fulkerson, o fluxo pode aumentar se existir algum caminho de s para t na rede residual correspondente. Caso contrário, é máximo. O algoritmo de Edmonds-Karp constitui uma variante do método de Ford-Fulkerson em que o caminho para aumento é determinado por pesquisa em largura (BFS).

(1) fluxo de 1 unidade

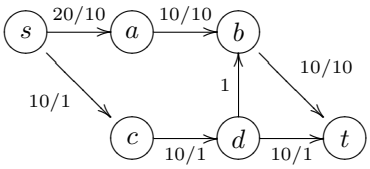


(2) rede residual associada

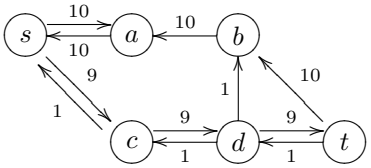


Caminho para aumento em BFS: s, a, b, t com capacidade 10.

(3) fluxo de 1+10=11 unidades

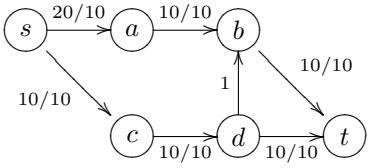


(4) rede residual associada

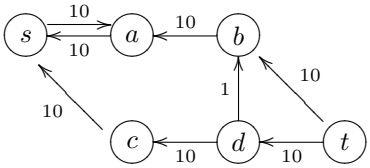


Caminho para aumento em BFS: s, c, d, t com capacidade 9.

(5) fluxo de 11+9=20 unidades

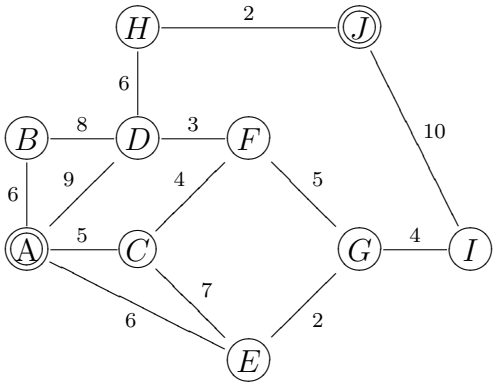


(6) rede residual associada



Não existe caminho de s para t. O fluxo é máximo.

7. Considere uma rede representada por um grafo não dirigido $G = (V, E, p, \{s, t\})$ com valores associados aos ramos, sendo $p(e)$ um inteiro positivo, qualquer que seja $e \in E$, por exemplo, como na figura, e s (origem) e t (destino) dois vértices especiais.



Suponha que G é de facto uma rede de transportes e $p(e)$ o valor da temperatura mínima (esperada) no troço e. Pretende-se um algoritmo para encontrar um caminho óptimo de s para t, sendo *óptimo* qualquer caminho tal que a temperatura mínima registada ao longo do caminho é máxima quando considerados todos os caminhos alternativos de s para t (na figura $s = A$ e $t = J$).

a) Que relação tem este problema com os abordados nas aulas?

Resposta:

É semelhante ao problema "Encomenda", devendo ser encontrado um caminho de capacidade máxima. Aqui, a capacidade do caminho é definida como o mínimo das temperaturas ao longo do caminho.

b) Apresente um algoritmo $O((n+m)\log_2 n)$ para resolver o problema sendo $m = |E|$ e $n = |V|$.

Resposta:

```
DIJKSTRA_ADAPTADO(s, G)      // G = (V, E, p, {s, t})
    Para cada v ∈ V fazer
        cap[v] ← 0;
        pai[v] ← NIL;
    cap[s] ← ∞;
    Q ← MK_PQ_HEAPMAX(cap, V);
    Enquanto(PQ_NOT_EMPTY(Q) ) fazer
        v ← EXTRACTMAX(Q);
        Se (v = t ou cap[v] = 0) então sai do ciclo;
        Para cada w ∈ Adjs[v] fazer
            Se cap[w] < min(cap[v], p(v, w)) então
                cap[w] ← min(cap[v], p(v, w));
                INCREASEKEY(Q, w, cap[w]);
                pai[w] ← v;
    Se cap[t] > 0 então ESCREVECAMINHO(t, pai);
    senão escrever("Não existe caminho");
```

```
ESCREVECAMINHO(v, pai)
    Se pai[v] ≠ NIL então
        ESCREVECAMINHO(pai[v], pai);
    escrever(v);
```

Em alternativa, pode-se construir uma árvore de cobertura com peso máximo e encontrar o caminho de s para t nesse subgrafo não dirigido (ou aplicar o algoritmo de Prim a partir de s e criar a estrutura *pai* correspondente). É conhecido que o caminho de u para v numa árvore de cobertura de peso máximo de G é sempre um caminho de capacidade máxima de u para v em G, qualquer que seja (u, v).

c) Que estruturas de dados utilizou para garantir essa complexidade? Que propriedades explora para garantir a correção do algoritmo?

Resposta:

As estruturas de dados utilizadas são: um grafo representado por listas de adjacências (e dirigido); uma fila de prioridades suportada por uma *heap* de máximo com operações MK_PQ_HEAPMAX e PQ_NOT_EMPTY realizadas em tempo $\Theta(n)$ e $O(1)$, e EXTRACTMAX e INCREASEKEY em tempo $O(\log_2 n)$ (com uma estrutura semelhante à descrita na questão 2); um vetor *pai* em que $pai[v]$ contém o identificador do nó que precede v no caminho encontrado; o vetor *cap* em que $cap[v]$ guarda a capacidade do melhor caminho já encontrado de s até v.

Embora não seja verdade que todos os caminhos de capacidade máxima de s para t sejam constituídos por sub-caminhos de capacidade máxima, é verdade que algum desses caminhos o é. Esta propriedade é usada no algoritmo (na definição e atualização de $cap[v]$). Pode-se provar que, o critério "greedy" de escolha do nó a sair da fila, conjuntamente com a atualização das capacidades dos nós adjacentes, garante que, quando um nó v é extraído da fila, $cap[v]$ é a capacidade máxima de qualquer caminho de s a v (ou seja, $cap[v]$ não poderia ser aumentada por um dos nós que ainda ficam na fila).

d) Determine a solução para o exemplo usando o algoritmo.

Resposta:

	A	B	C	D	E	F	G	H	I	J
	$\infty, -$	$0, -$	$0, -$	$0, -$	$0, -$	$0, -$	$0, -$	$0, -$	$0, -$	$0, -$
sai A	∞,-	$6, A$	$5, A$	$9, A$	$6, A$					
sai D		$8, D$		9,A		$3, D$		$6, D$		
sai B		8,D								
sai H								6,D		$2, H$
sai E			$6, E$		6,A		$2, E$			
sai C			6,E			$4, C$				
sai F						4,C	$4, F$			
sai G							4,F		$4, G$	
sai I									4,G	$4, I$
sai J										4,I

O caminho é: $(J, I, G, F, C, E, A)^{-1} = (A, E, C, F, G, I, J)$

(As alterações e a indicação do caminho poderiam ter sido assinaladas diretamente no grafo).

8. (*perg. alternativa a 3.c)) Suponha que tem um grafo dirigido acíclico $G = (V, E, p)$ com pesos e em que os identificadores dos nós já definem uma ordem topológica. Suponha que um grupo de pessoas vai ser transportado de s (origem) para t (destino) e que será dividido por vários veículos. Cada veículo assegura uma *ligação* (arco da rede) e apenas uma. Todos os veículos são usados apenas uma vez, e os veículos que saem de um nó sairão exatamente ao mesmo tempo (se necessário, aguardam a chegada dos que demoram mais tempo a chegar). Admitindo que $p(u, v)$ designa o tempo que demorará a efectuar o troço (u, v) e que o transbordo de pessoas nos nós é muito rápido (podendo ser negligenciado) pretende-se determinar ao fim de quantas horas o grupo estará novamente reunido em t.

a) Apresente um algoritmo que resolva o problema em $\Theta(m + n)$, sendo $n = |V|$ e $m = |E|$.

Resposta:

```
REENCONTRO(G, s, t)
 1  Se s > t então retornar −1;      // nenhum elemento chegará ao destino
 2  Para i ← s + 1 até t fazer
 3      d[i] ← 0;
 4  d[s] ← 1;      // 1 em vez de 0 para na linha 6. se saber se alguém chegou a um nó i
 5  Para i ← s até t − 1 fazer
 6      Se d[i] ≠ 0 então      // chegaram elementos a i
 7          Se Adjs[i] = ∅ então retornar −1;      // os elementos que chegaram a i não podem avançar
 8          Para todo j ∈ Adjs[i] fazer
 9              Se j > t então retornar −1;      // alguns elementos vão perder-se
10              Se d[i] + p(i, j) > d[j] então
11                  d[j] ← d[i] + p(i, j);
12  retornar d[t] − 1;
```

Assumiu-se que todas as ligações que saem de um nó seriam usadas pelo grupo. No algoritmo apresentado, se $d[i] > 0$ então existe um caminho de s para i e, consequentemente, há pessoas que chegam a i. Sendo G um DAG, se $d[i] = 0$ quando chegar a vez de i ser tratado (na ordem topológica), então pode-se concluir que não existe caminho de s para i em G, e, portanto, ninguém que tenha saído de s chegou a i. Assim, nesse caso o nó i não poderá atualizar os tempos dos seus adjacentes.
Para instâncias como a correspondente ao problema 8b), em que $s = 1$, $t = n$, qualquer nó é acessível de s em G, e t é acessível de qualquer nó em G, o algoritmo tem complexidade $\Theta(m+n)$. No entanto, em geral, para um DAG qualquer, só podemos afirmar que é $O(m+n)$. Seria fácil acrescentar código desnecessário para o transformar num algoritmo $\Theta(m + n)$, por exemplo, para começar por realizar uma visita em largura ao grafo, mas não nos parece interessante (ou razoável).

b) Altere o grafo que está no exemplo dado no problema anterior, substituindo cada ramo por um arco que deverá ser orientado de u para v se a letra que identifica u preceder a que identifica v no alfabeto. Aplique o algoritmo a esse exemplo para $s = A$ e $t = J$.

Resposta:

	A	B	C	D	E	F	G	H	I	J
	1	0	0	0	0	0	0	0	0	0
A	[1]	7	6	10	7					
B		[7]		15						
C			[6]		13	10				
D				[15]		18		21		
E					[13]		15			
F						[18]	23			
G							[23]		27	
H								[21]		23
I									[27]	37

A tabela apresenta a alteração dos valores de $d[\cdot]$ à medida que o algoritmo vai progredindo. Em cada linha encontram-se os valores que são atualizados pelo nó v considerado e o valor de $d[v]$. Para ilustrar a aplicação da função, poderia ter sido usado diretamente o grafo, e anotados os valores nos vértices (como nas aulas).

A função retorna: 37-1 = 36.

(FIM)