# MAC 5711 - Análise de Algoritmos

Rodrigo Augusto Dias Faria
Departamento de Ciência da Computação - IME/USP

10 de novembro de 2015

## Lista 7

**1. (CRLS 22.2-1)** Simule o funcionamento da BFS no grafo da Figura 22.2(a) do CLRS (segunda edição) a partir do vértice 3, determinando os valores de $d$ e $\pi$ para cada vértice.

**Resposta:** Inicializamos os atributos *color*, $d$ e $\pi$ de cada vértice $v \in V$ do grafo com WHITE, $\infty$ e NIL, respectivamente, conforme a primeira iteração do algoritmo BFS, exceto para o nó 3 que é o parâmetro $s$ neste caso, cujo $d = 0$ e sua cor é GRAY, conforme pode ser visto em **a)**.

Colocamos $s$ na fila e, então, visitamos a lista de adjacências de cada vértice a partir de $s$, sendo que $u$ é o vértice retirado da pilha - aquele cuja cor é BLACK ao final do $for$ da linha 12.

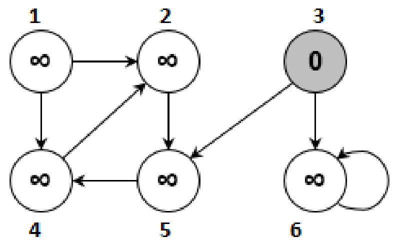
Note que o valor de $d$ está dentro de cada vértice do grafo e, a cada novo vértice que descobrimos a partir de $u$, ou seja, aquele que ainda tem a cor WHITE, marcamos em seu atributo $\pi$ o seu antecessessor - o próprio $u$ - que destacamos em laranja.

Note, também, que há casos em que nenhum novo vértice é descoberto, como em **d)**, por exemplo.

Quando a pilha estiver vazia, concluímos a execução do algoritmo. Note que, neste caso, o vértice 1 não pode ser atingido a partir de 3 e, portanto, seu antecessor $\pi$ fica marcado como NIL. O vértice $s$ também tem seu antecessor NIL, já que é a entrada da busca no grafo e isso ocorre sempre.

As arestas que estão destacadas formam a *breadth-first tree* e o antecessor de cada nó da árvore é dado pelo atributo $\pi$ de cada vértice.

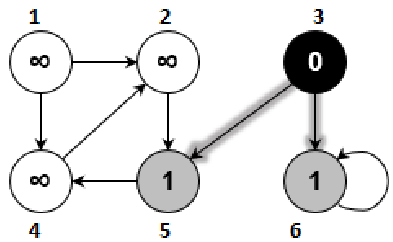
a)




| | |
|---|---|
| **Q** | 3 |

| **V** | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **π** | nil | nil | nil | nil | nil | nil |

b)

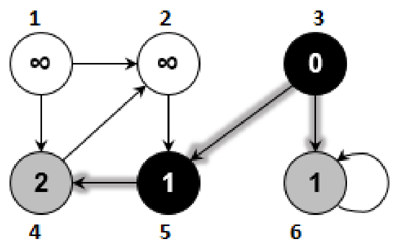



| | | |
|---|---|---|
| **Q** | 5 | 6 |

| **V** | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **π** | nil | nil | nil | nil | 3 | 3 |

u = 3

c)




| | | |
|---|---|---|
| **Q** | 6 | 4 |

| **V** | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **π** | nil | nil | nil | 5 | 3 | 3 |

u = 5

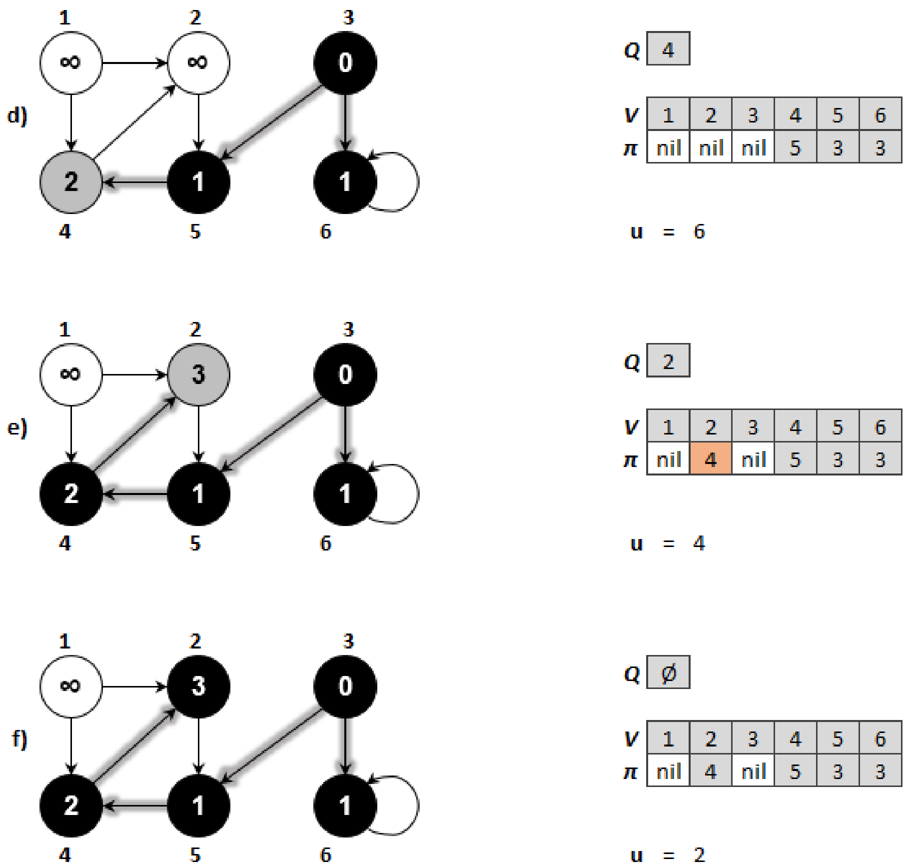
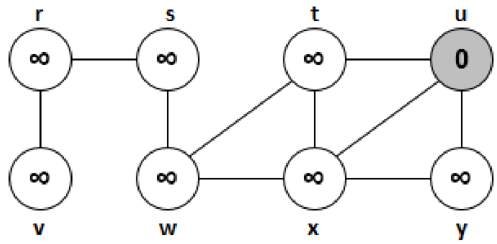



Figura 1: Sequência das operações do algoritmo BFS, sendo $s = 3$.

**2. (CRLS Ex. 22.2-2)** Simule o funcionamento da BFS no grafo da Figura 22.3 do CLRS (segunda edição) a partir do vértice $u$, determinando os valores de $d$ e $\pi$ para cada vértice.

**Resposta:** A mesma analogia aplicada na questão anterior pode ser utilizada aqui, mesmo tratando-se de um grafo não orientado, o algoritmo BFS funciona em ambos os casos, conforme vimos em sala/CLRS.

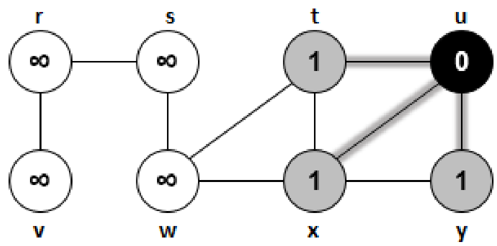
a)




| $Q$ | u |
|---|---|

| $V$ | r | s | t | u | v | w | x | y |
|---|---|---|---|---|---|---|---|---|
| $\pi$ | nil | nil | nil | nil | nil | nil | nil | nil |

b)

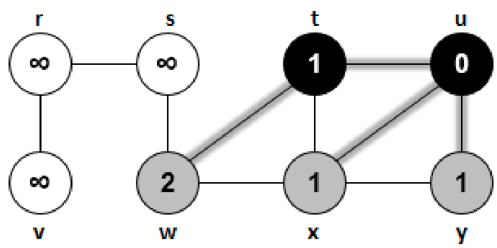



| $Q$ | t | x | y |
|---|---|---|---|

| $V$ | r | s | t | u | v | w | x | y |
|---|---|---|---|---|---|---|---|---|
| $\pi$ | nil | nil | u | nil | nil | nil | u | u |

u = u

c)

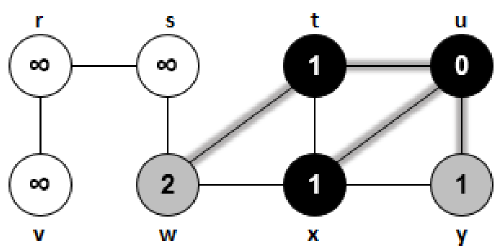



| $Q$ | x | y | w |
|---|---|---|---|

| $V$ | r | s | t | u | v | w | x | y |
|---|---|---|---|---|---|---|---|---|
| $\pi$ | nil | nil | u | nil | nil | t | u | u |

u = t

d)




| $Q$ | y | w |
|---|---|---|

| $V$ | r | s | t | u | v | w | x | y |
|---|---|---|---|---|---|---|---|---|
| $\pi$ | nil | nil | u | nil | nil | t | u | u |

u = x

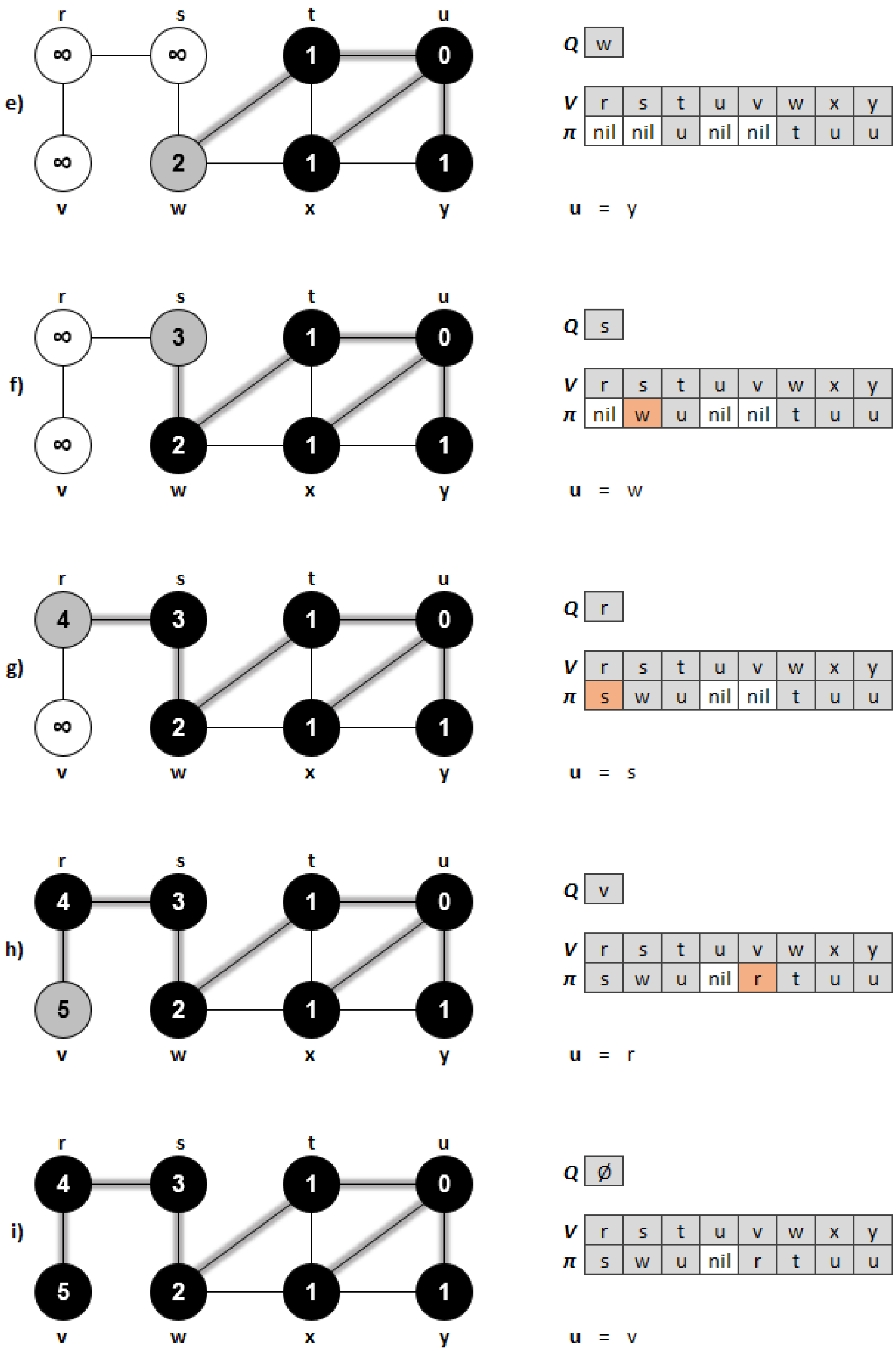



Figura 2: Sequência das operações do algoritmo BFS, sendo $s = u$.

**3. (CRLS 22.2-4)** Argumente que o valor de $d[u]$ atribuído ao vértice $u$ na BFS é independente da ordem em que os vértices das listas de adjacências são dados. Por outro lado, mostre, usando o exemplo da Figura 22.3 do CLRS, que a árvore BFS depende da ordem dos vértices nas listas de adjacências.

**Resposta:** O atributo $d$ de cada vértice $v$ é calculado uma única vez na linha 15, sendo que este valor é incrementado a cada nível da árvore que algoritmo descobre a partir de $s$.

Se tomarmos, por exemplo, o vértice $u$ do exercício anterior, de qualquer forma que organizarmos a sua lista de adjacências ($\{t, x, y\}, \{x, y, t\}, \{y, t, x\}$), o atributo $d$ de cada um destes vértices sempre será 1, o que nos mostra que eles estão em um nível imediatamente abaixo de $u$ na árvore.

Por outro lado, a ordem da lista de adjacências influencia na árvore resultante após a aplicação do algoritmo. Usando o mesmo exemplo que citamos no caso anterior, se tomarmos a lista de adjacências de $u$ na ordem $\{x, y, t\}$, a sub-árvore do segundo nível terá, agora, o vértice $x$ como a raiz, e não mais $t$, como visto no exercício 2. Consequentemente, o atributo $\pi$ do vértice $w$ também muda, já que $x$, agora, passa a ser o seu antecessor.

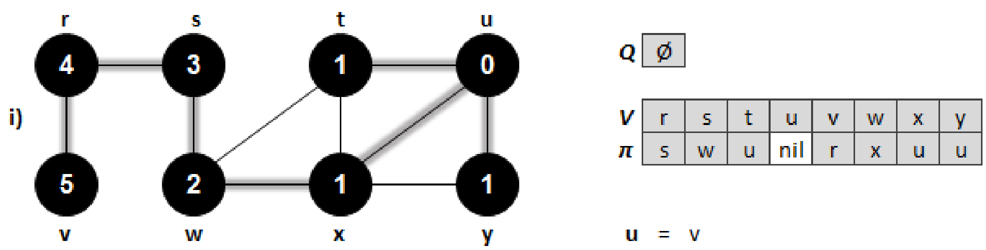



Figura 3: Árvore resultante do algoritmo BFS, sendo $s = u$ e a lista de adjacências na ordem $\{x, y, t\}$.

**4. (CRLS 22.2-5)** Considere um grafo orientado $D = (N, A)$. Dê um exemplo de um conjunto $A_\pi \subseteq A$ de arcos em $D$ que formam uma árvore tal que, entre quaisquer dois nós $u$ e $v$ em $D$, o único caminho entre $u$ e $v$ em $A_\pi$ é um caminho mínimo em $D$ entre $u$ e $v$, porém, $A_\pi$ jamais seria produzida por uma execução da BFS em $D$, independente da ordem dos nós nas listas de adjacências de $D$ e do nó inicial $s$.

**Resposta:** Seja o conjunto de arcos $A_\pi$ destacados no grafo da figura 4.

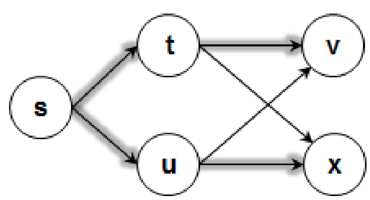



Figura 4: Grafo orientado $D = (N, A)$.

Note que, independente da ordem dos elementos adjacentes a $s$, $t$ e $u$, essa representação jamais poderá ser obtida pela BFS. A ordem das listas de adjacências dos elementos $t$ e $u$ também não influenciam na forma com que a árvore será gerada. A figura 5 mostra o resultado da execução da BFS, bem como a lista de adjacências em ordens diferentes em cada caso.

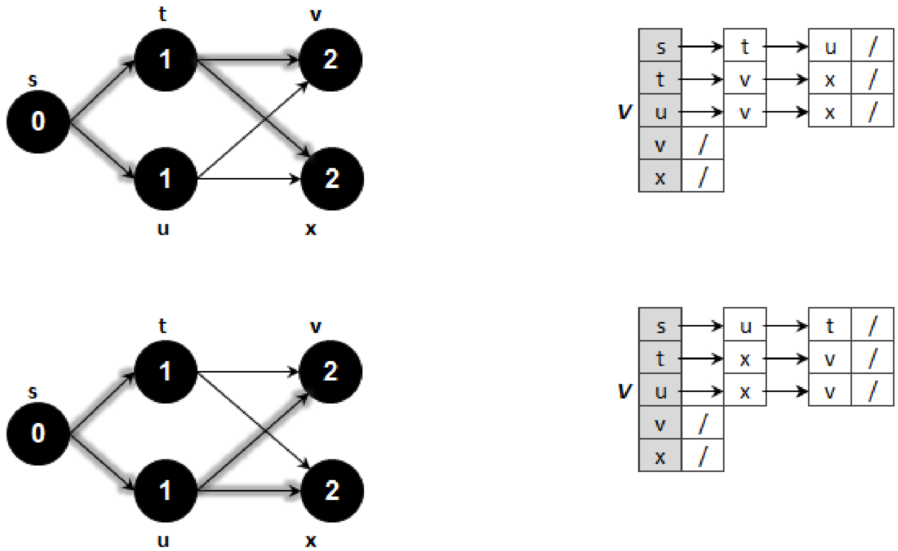



Figura 5: Execução da BFS no grafo orientado $D = (N, A)$ com a lista de adjacências em duas ordens distintas.

**5.** Escreva uma versão não recursiva da busca em profundidade.

**Resposta:** Podemos utilizar uma pilha como apoio para aprofundar na lista de adjacências de cada nó da árvore, substituindo a recursão.

Além disso, nós usamos uma fila FIFO para a lista de adjacências de cada vértice $u$ e, assim que um novo vértice $v$ adjacente à $u$ é encontrado e ele ainda não foi visitado, nós o visitamos e empilhamos $v$ em $S$.

Note que nós somente retiramos $u$ da pilha (linha 11 da DFS-Visit) quando todos os vértices da lista de adjacências dele foram visitados, ou seja, o momento em que devemos marcar $u$ como finalizado (sub-rotina Blacken).

Outro ponto importante é que, como visto na linha 12 da Iterative-DFS, o vértice $s$ que dá origem à busca tem seu ancestral como *nil*, o que garante o funcionamento da mesma forma que a Dfs original. Isso também garante que a busca funciona nos casos em que tivermos mais de uma componente conexa no grafo.

**Consumo de tempo:** O *loop* das linhas 2-8 da Iterative-DFS tomam $\Theta(V)$ para inicializar cada vértice $u \in V[G] + \Theta(E)$ para montar a fila de adjacências de cada vértice $u$. As sub-rotinas Blacken, Grayen, bem como as operações na fila/pilha tomamm $\Theta(1)$. O *loop* das linhas 10-13 consome tempo $\Theta(V)$, ou seja, executa no máximo uma vez para cada vértice $v \in G$, já que DFS-Visit é executado apenas em vértices que ainda não foram descobertos - àqueles que ainda são brancos. Como a fila de adjacências é visitada no máximo uma vez na sub-rotina DFS-Visit e a soma do comprimento da fila de adjacências de cada vértice $v$ é $\Theta(E)$, o tempo gasto na DFS-Visit é $O(E)$.

Portanto, o consumo de tempo total será $O(V + E)$, o que mantém o comportamento assintótico original da DFS.

```
ITERATIVE-DFS(G)
// let S be an empty stack
for each vertex u ∈ V[G]
    color[u] = WHITE
    π[u] = NIL
    n = Adj[u].length
    for i = n to 1
        v = Adj[u][i]
        ENQUEUE(Qadj[u], v)
time = 0
for each vertex u ∈ V[G]
    if color[u] == WHITE
        GRAYEN(u, NIL)
        DFS-VISIT(u)
```

```
DFS-VISIT(s)
PUSH(S, s)
while S ≠ ∅
    u = TOP(S)
    if Qadj[u] ≠ ∅
        v = DEQUEUE(Qadj[u])
        if color[v] == WHITE
            GRAYEN(v, u)
            PUSH(S, v)
    else
        BLACKEN(u)
        POP(S)
```

```
GRAYEN(v, u)
color[v] = GRAY
time = time + 1
d[v] = time
π[v] = u
```

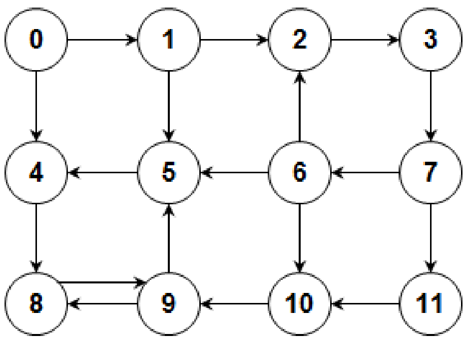
```
BLACKEN(u)
color[u] = BLACK
time = time + 1
f[u] = time
```

**6.** Execute uma busca em profundidade a partir do vértice 0 no grafo orientado dado pelas listas de adjacência a seguir. Exiba o rastreamento da busca.

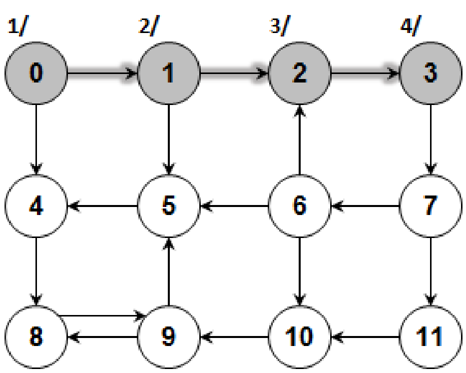
0: 1 4
1: 2 5
2: 3

3: 7
4: 8
5: 4
6: 5 10 2
7: 11 6
8: 9
9: 5 8
10: 9
11: 10

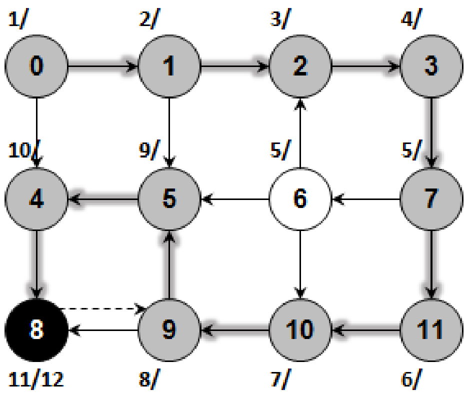
**Resposta:** A figura 6 mostra o grafo formado pela lista de adjacências dada, bem como o rastreamento no momento em que o vértice 3, 6 e 8 são descobertos, e o resultado final da DFS ao final de todas as chamadas recursivas, respectivamente. Os tempos $d$ e $f$ estão no vetor à direita e, também, acima de cada vértice.



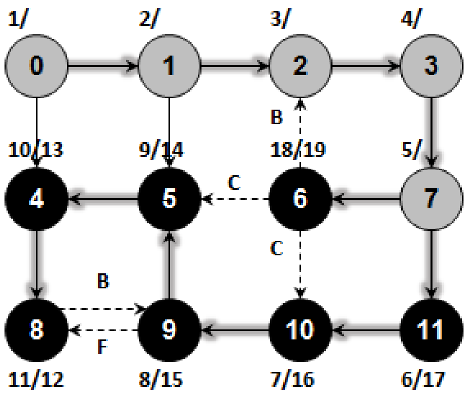
| **V** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **π** | nil | nil | nil | nil | nil | nil | nil | nil | nil | nil | nil | nil |
| **d** | | | | | | | | | | | | |
| **f** | | | | | | | | | | | | |



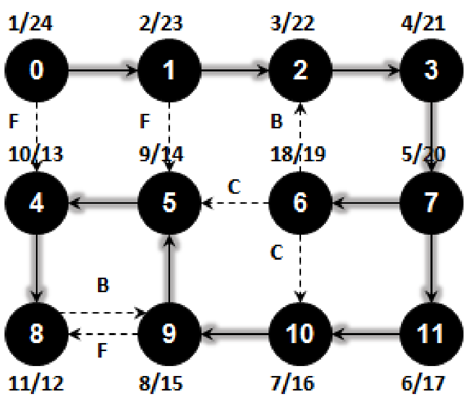

| **V** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **π** | nil | 0 | 1 | 2 | nil | nil | nil | nil | nil | nil | nil | nil |
| **d** | 1 | 2 | 3 | 4 | | | | | | | | |
| **f** | | | | | | | | | | | | |




| V | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| π | nil | 0 | 1 | 2 | 5 | 9 | nil | 3 | 4 | 10 | 11 | 7 |
| d | 1 | 2 | 3 | 4 | 10 | 9 | | 5 | 11 | 8 | 7 | 6 |
| f | | | | | | | | | 12 | | | |




| V | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| π | nil | 0 | 1 | 2 | 5 | 9 | 7 | 3 | 4 | 10 | 11 | 7 |
| d | 1 | 2 | 3 | 4 | 10 | 9 | 18 | 5 | 11 | 8 | 7 | 6 |
| f | | | | | 13 | 14 | 19 | | 12 | 15 | 16 | 17 |




| V | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| π | nil | 0 | 1 | 2 | 5 | 9 | 7 | 3 | 4 | 10 | 11 | 7 |
| d | 1 | 2 | 3 | 4 | 10 | 9 | 18 | 5 | 11 | 8 | 7 | 6 |
| f | 24 | 23 | 22 | 21 | 13 | 14 | 19 | 20 | 12 | 15 | 16 | 17 |

Figura 6: Rastreamento da DFS na lista de adjacências dada.

**7. (CRLS 22.3-1)** Desenhe uma tabela $3x3$, com as linhas e colunas indexadas pelas cores branco, cinza e preto. Em cada entrada $(i, j)$, indique se, em qualquer ponto durante uma DFS de um grafo orientado, pode existir um arco de um nó de cor $i$ para um nó de cor $j$. Para cada arco possível, indique as classificações que ele pode ter (de árvore, de retorno, para frente, cruzado). Faça um segundo quadro considerando um grafo não orientado.

As tabelas 1 e 2 mostram as classificações dos arcos para o grafo orientado e não orientado, respectivamente.

As siglas significam ***T**ree*, ***B**ack*, ***C**ross* e ***F**orward edge*.

| | **White** | **Gray** | **Black** |
|---|---|---|---|
| **White** | x | x | x |
| **Gray** | T | B | F/C |
| **Black** | x | x | B |

Tabela 1: Classificação dos arcos no grafo orientado para a DFS.

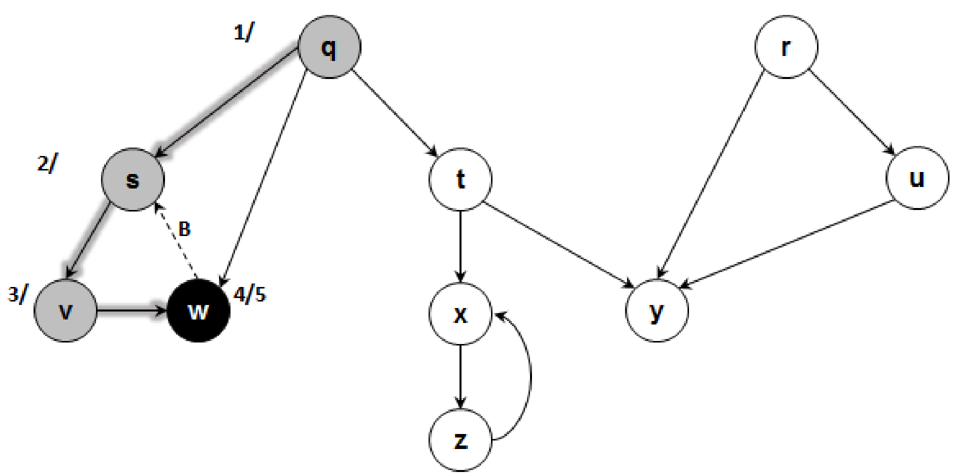
| | **White** | **Gray** | **Black** |
|---|---|---|---|
| **White** | x | x | x |
| **Gray** | T | B | x |
| **Black** | x | B | B |

Tabela 2: Classificação dos arcos no grafo não orientado para a DFS.

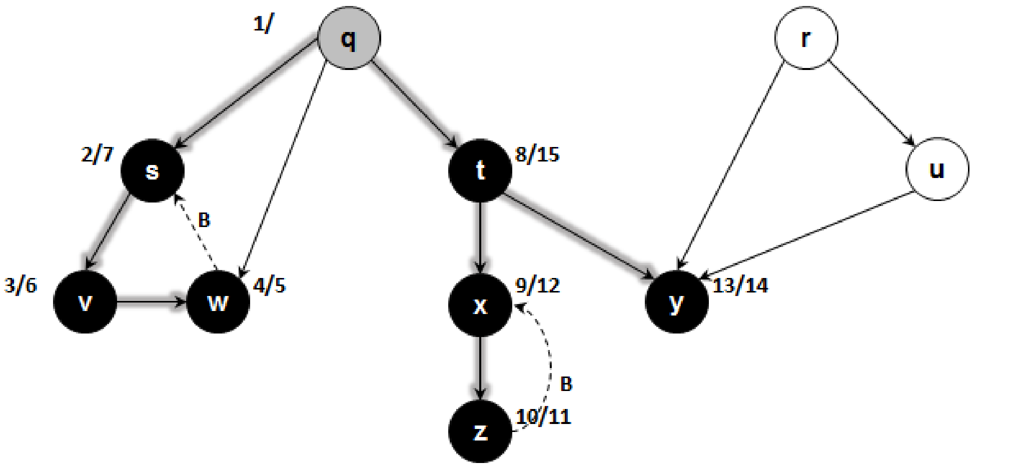
**8. (CRLS 22.3-2)** Mostre como a DFS funciona no grafo da Figura 22.6 do CLRS (segunda edição). Assuma que o laço das linhas 5-7 da DFS visitam os vértices em ordem alfabética, e que os vértices se encontram em ordem alfabética nas listas de adjacências. Mostre os valores de $d$ e $f$ para cada vértice ao final da DFS.

**Resposta:** A figura 7 mostra o rastreamento da DFS em 3 momentos distintos: quando todos os vértices adjacentes à $w$ são visitados, todos os vértices adjacentes à $t$ são visitados e a árvore com todas as chamadas recursivas concluídas, respectivamente.

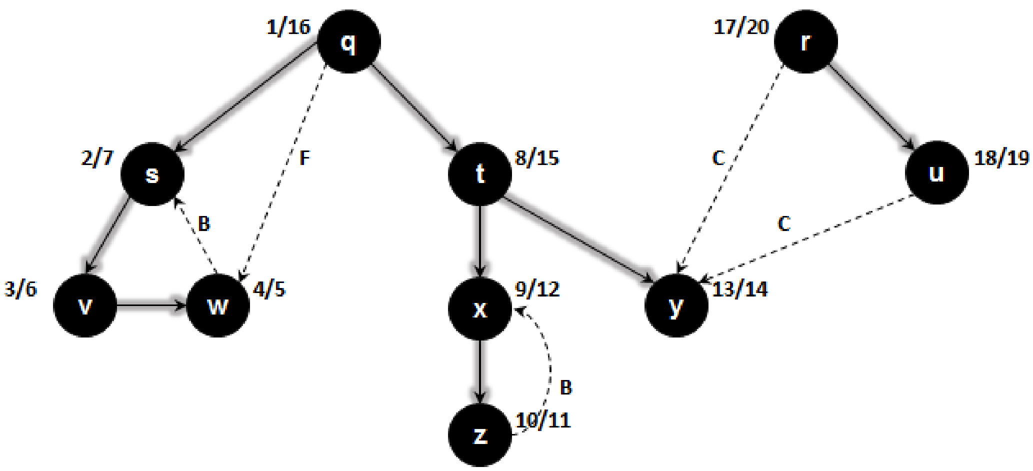
Os valores $d$ e $f$, bem como o antecessor de cada vértice dado por $\pi$ estão na tabela de rastreamento abaixo de cada imagem. Os tipos de arestas também estão devidamente destacados.




| *V* | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|
| *π* | nil | nil | q | nil | nil | s | v | nil | nil | nil |
| *d* | 1 | | 2 | | | 3 | 4 | | | |
| *f* | | | | | | | 5 | | | |




| | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|
| **V** | q | r | s | t | u | v | w | x | y | z |
| **π** | nil | nil | q | q | nil | s | v | t | t | x |
| **d** | 1 | | 2 | 8 | | 3 | 4 | 9 | 13 | 10 |
| **f** | | | 7 | 15 | | 6 | 5 | 12 | 14 | 11 |




| | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|
| **V** | q | r | s | t | u | v | w | x | y | z |
| **π** | nil | nil | q | q | r | s | v | t | t | x |
| **d** | 1 | 17 | 2 | 8 | 18 | 3 | 4 | 9 | 13 | 10 |
| **f** | 16 | 20 | 7 | 15 | 19 | 6 | 5 | 12 | 14 | 11 |

Figura 7: Rastreamento da DFS na figura 22.6 do CLRS.

**9. (CRLS 22.3-7)** Mostre um contraexemplo para a conjectura que se existe um caminho de $u$ a $v$ em um grafo orientado $G$, e se $d[u] < d[v]$ numa DFS de $G$, então $v$ é descendente de $u$ na floresta DFS produzida.

Podemos observar a própria árvore à esquerda da figura 22.5 (c) do CLRS como um contraexemplo. Seja a DFS produzida a partir de $s$, se tomarmos os vértices $x$ e $w$, $d[x] < d[w]$, existe um caminho de $x$ a $w$, mas $w$ não é descendente de $x$.

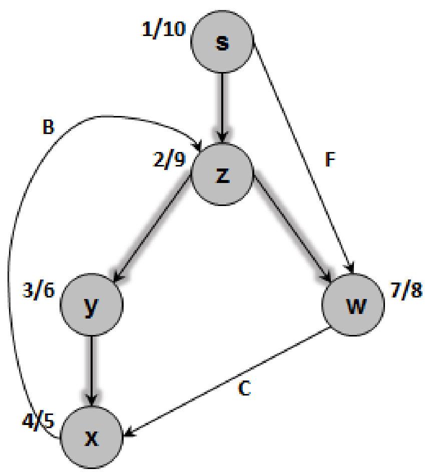



Figura 8: Contraexemplo utilizando parte do grafo da figura 22.5 (c) do CLRS.

**10. (CRLS 22.3-8)** Mostre um contraexemplo para a conjectura que se existe um caminho de $u$ a $v$ em um grafo orientado $G$, então qualquer DFS deve resultar em $d[v] \leq f[u]$.

A figura 9 mostra um contraexemplo da conjectura. Temos um caminho de $u$ a $v$ no grafo, porém, aplicando a DFS a partir de $s$, temos que $d[v] > f[u]$.

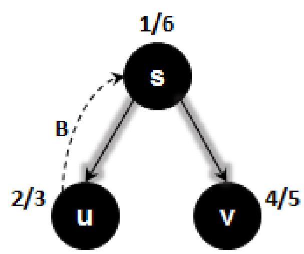



Figura 9: Contraexemplo da conjectura dada.

**11. (CRLS 22.3-10)** Mostre como um vértice $u$ num grafo orientado pode terminar sozinho numa árvore de uma floresta DFS mesmo tendo arcos saindo e entrando dele em $G$.

A figura 10 mostra um exemplo onde o vértice $u$ pode ficar sozinho numa árvore de uma floresta DFS gerada a partir do vértice $s$.

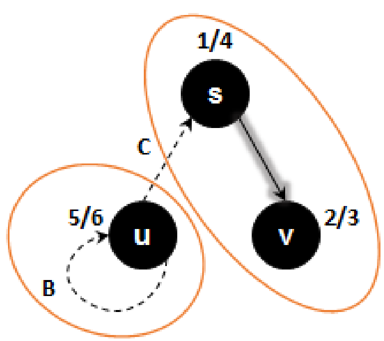



Figura 10: Exemplo onde o vértice $u$ pode ficar isolado.

Um outro exemplo pode ser visto na figura 11, sendo que cada árvore é formada por um único vértice, já que a busca começa em $v$.

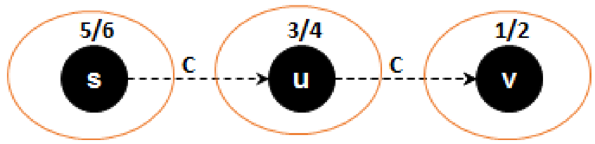



Figura 11: Segundo exemplo onde o vértice $u$ pode ficar isolado.

**14.** Escreva um algoritmo que decida se um grafo é conexo. Analise o seu consumo de tempo.

**Resposta:** Um grafo é conexo se e somente se, para cada par $s$ e $t$ de seus vértices, existe um caminho com origem $s$ e término $t$.

Podemos fazer uma alteração no algoritmo da DFS para contar o número de componentes conexas do grafo e executar a DFS usando cada vértice $u \in V[G]$ como origem da busca. Se houver uma única componente conexa para a busca a cada execução, o grafo é conexo.

Os vértices estão organizados em uma pilha, utilizamos as rotinas Pop e Push-Left, para remover um elemento da última posição da fila e inseri-lo na primeira posição, respectivamente, de modo que todos os vértices sejam origem da busca uma vez.

**Consumo de tempo:** A mudança em Connected-DFS não muda o comportamento assintótico original do algoritmo, que permanece $\Theta(V + E)$.

A criação da pilha na linha 2 na rotina Graph-Connected toma tempo $\Theta(V)$. Já as operações na pilha gastam $\Theta(1)$, bem como as demais instruções. Como nós fazemos a busca uma vez para cada vértice, temos que o consumo de tempo total será $\Theta(V(V + E))$.

Graph-Connected($G$)
1 $n = |V[G]|$
2 $S = V[G]$ // $S$ is a stack with the vertices of $G$
3 **while** $n > 1$
4     *count* = Connected-DFS($G$)
5     **if** *count* $> 1$
6         **return** FALSE
7     $u$ = Pop($S$)
8     Push-Left($S, u$)
9     $n = n - 1$
10 **return** TRUE

```
Connected-DFS(G)
for each vertex u ∈ V[G]
    color[u] = WHITE
    π[u] = NIL
time = 0
count = 0
for each vertex u ∈ V[G]
    if color[u] == WHITE
        count = count + 1
        DFS-Visit(u)
return count
```

**15.** Escreva um algoritmo que determine o número de componentes conexas de um grafo. Analise o seu consumo de tempo.

**Resposta:** A rotina Connected-DFS do exercício 7-14 já retorna a quantidade de componentes conexas de um grafo $G$.

O consumo de tempo é o mesmo da DFS original, ou seja, $\Theta(V + E)$.

## CLRS (Outros)

A.1-7 Avalie o produtório $\prod_{k=1}^{n} 2(4^k)$.

$$\prod_{k=1}^{n} 2(4^k) = \prod_{k=1}^{n} 2({(2^2)}^k) = \prod_{k=1}^{n} 2(2^{2k}) = \prod_{k=1}^{n} 2^{2k+1}$$

Se avaliarmos o produtório para $n = 3$, por exemplo:

$$\prod_{k=1}^{n} 2^{2k+1} = 2^{2+1} \times 2^{4+1} \times 2^{6+1}$$

Percebemos que o expoente de 2 cresce em uma série aritmética:

$$\sum_{k=1}^{n} 2k + 1 = \sum_{k=1}^{n} 2k + \sum_{k=1}^{n} 1 = 2\sum_{k=1}^{n} k + n = 2(\frac{n(n+1)}{2}) + n = n(n+2)$$

Portanto:

$$\prod_{k=1}^{n} 2(4^k) = 2^{n(n+2)}$$