

MAC 5711 - Análise de Algoritmos

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

23 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 π para cada vértice.

Resposta: Inicializamos os atributos *color*, d e π de cada vértice $v \in V$ do grafo com WHITE, ∞ 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**).

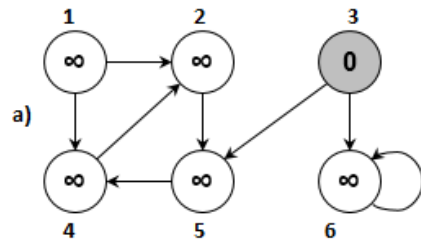
Colocamos s na fila e, então, visitamos a lista de adjacências de cada vértice a partir de s , sendo que u é o vértice retirado da pilha - aquele cuja cor é BLACK ao final do *for* da linha 12.

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 π o seu antecessor - 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 π 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.

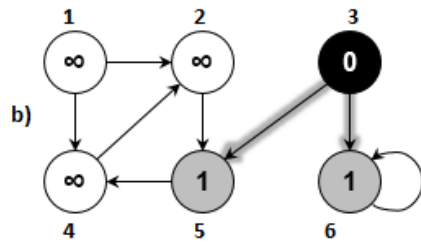
As arestas que estão destacadas formam a *breadth-first tree* e o antecessor de cada nó da árvore é dado pelo atributo π de cada vértice.



Q

3

V	1	2	3	4	5	6
π	nil	nil	nil	nil	nil	nil

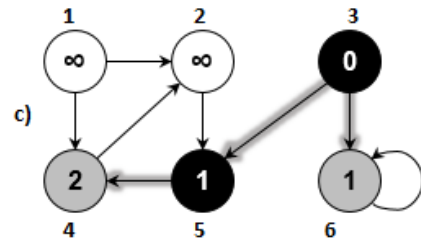


Q

5	6
---	---

V	1	2	3	4	5	6
π	nil	nil	nil	nil	3	3

$u = 3$



Q

6	4
---	---

V	1	2	3	4	5	6
π	nil	nil	nil	5	3	3

$u = 5$

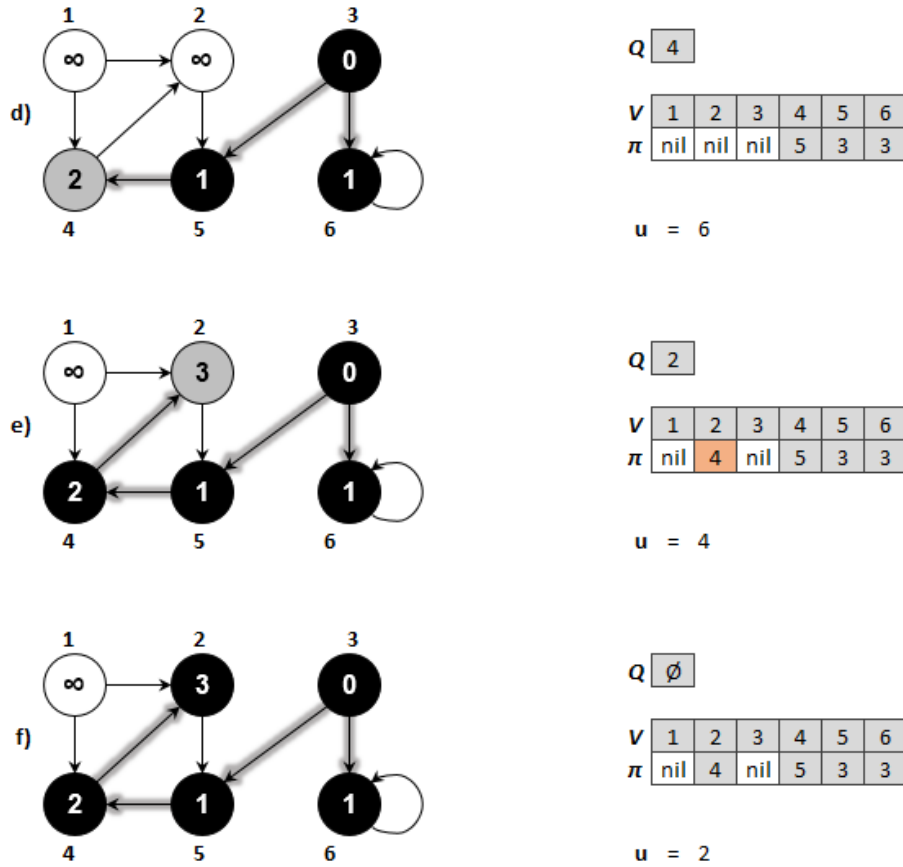
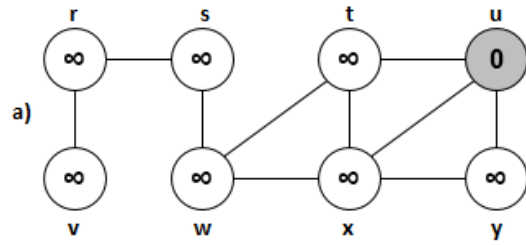


Figura 1: Sequência das operações do algoritmo BFS, sendo $s = 3$.

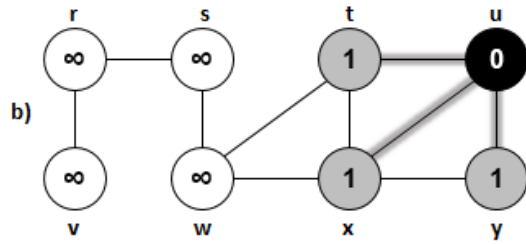
2. (CRLS Ex. 22.2-2) Simule o funcionamento da BFS no grafo da Figura 22.3 do CLRS (segunda edição) a partir do vértice u , determinando os valores de d e π para cada vértice.

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.



Q [u]

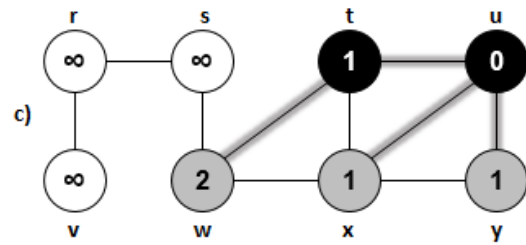
V	r	s	t	u	v	w	x	y
π	nil	nil	nil	nil	nil	nil	nil	nil



Q [t x y]

V	r	s	t	u	v	w	x	y
π	nil	nil	u	nil	nil	nil	u	u

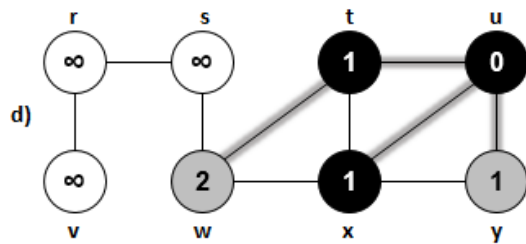
u = u



Q [x y w]

V	r	s	t	u	v	w	x	y
π	nil	nil	u	nil	nil	t	u	u

u = t



Q [y w]

V	r	s	t	u	v	w	x	y
π	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 π do vértice w também muda, já que x , agora, passa a ser o seu antecessor.

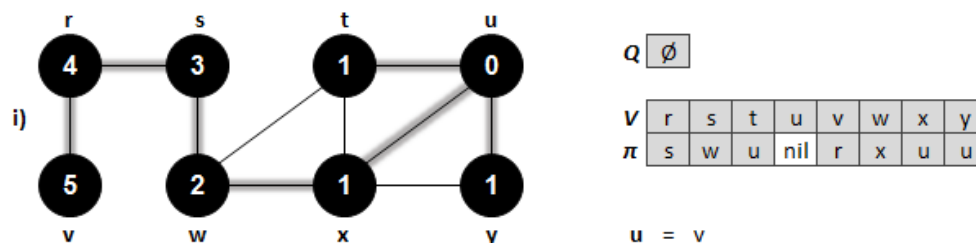


Figura 3: Árvore resultante do algoritmo BFS, sendo $s = u$ e a lista de adjacências na ordem $\{x, y, t\}$.

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_π é um caminho mínimo em D entre u e v , porém, A_π 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_π destacados no grafo da figura 4.

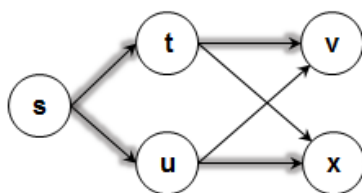


Figura 4: Grafo orientado $D = (N, A)$.

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.

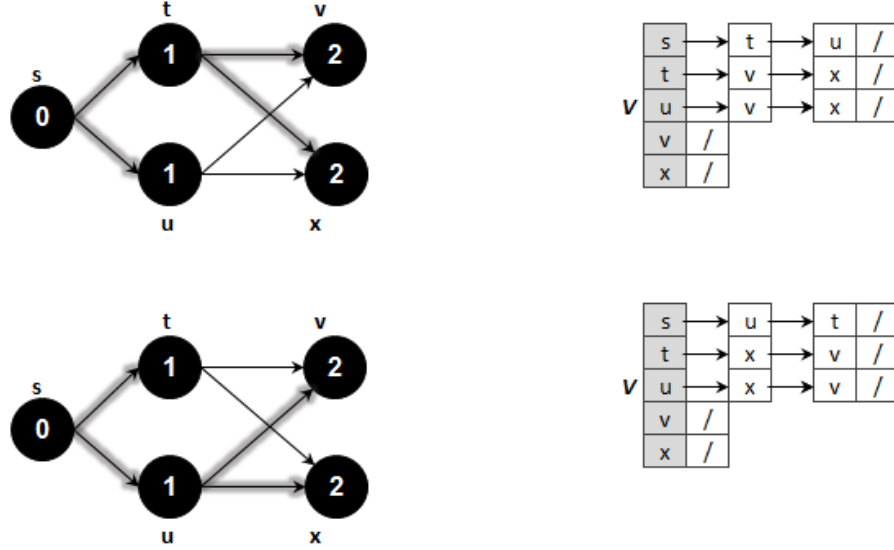


Figura 5: Execução da BFS no grafo orientado $D = (N, A)$ com a lista de adjacências em duas ordens distintas.

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 tomam $\Theta(1)$. 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)

```
1  // let  $S$  be an empty stack
2  for each vertex  $u \in V[G]$ 
3       $color[u] = \text{WHITE}$ 
4       $\pi[u] = \text{NIL}$ 
5       $n = Adj[u].length$ 
6      for  $i = n$  to 1
7           $v = Adj[u][i]$ 
8          ENQUEUE( $Qadj[u], v$ )
9   $time = 0$ 
10 for each vertex  $u \in V[G]$ 
11     if  $color[u] == \text{WHITE}$ 
12         GRAYEN( $u, \text{NIL}$ )
13         DFS-VISIT( $u$ )
```

DFS-VISIT(s)

```
1  PUSH( $S, s$ )
2  while  $S \neq \emptyset$ 
3       $u = \text{TOP}(S)$ 
4      if  $Qadj[u] \neq \emptyset$ 
5           $v = \text{DEQUEUE}(Qadj[u])$ 
6          if  $color[v] == \text{WHITE}$ 
7              GRAYEN( $v, u$ )
8              PUSH( $S, v$ )
9      else
10         BLACKEN( $u$ )
11         POP( $S$ )
```

GRAYEN(v, u)

```
1   $color[v] = \text{GRAY}$ 
2   $time = time + 1$ 
3   $d[v] = time$ 
4   $\pi[v] = u$ 
```

BLACKEN(u)

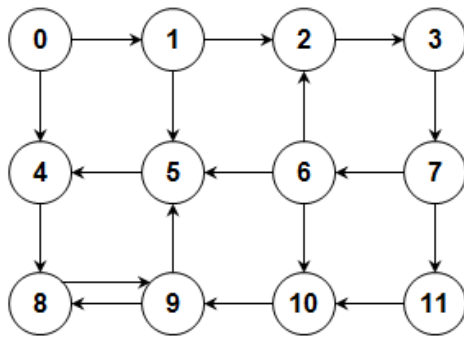
```
1   $color[u] = \text{BLACK}$ 
2   $time = time + 1$ 
3   $f[u] = time$ 
```

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.

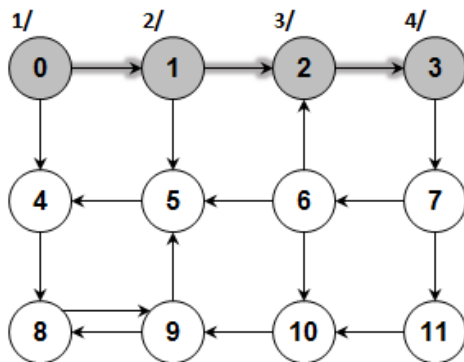
```
0: 1 4
1: 2 5
2: 3
```


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	11
d	1	2	3	4								
f												

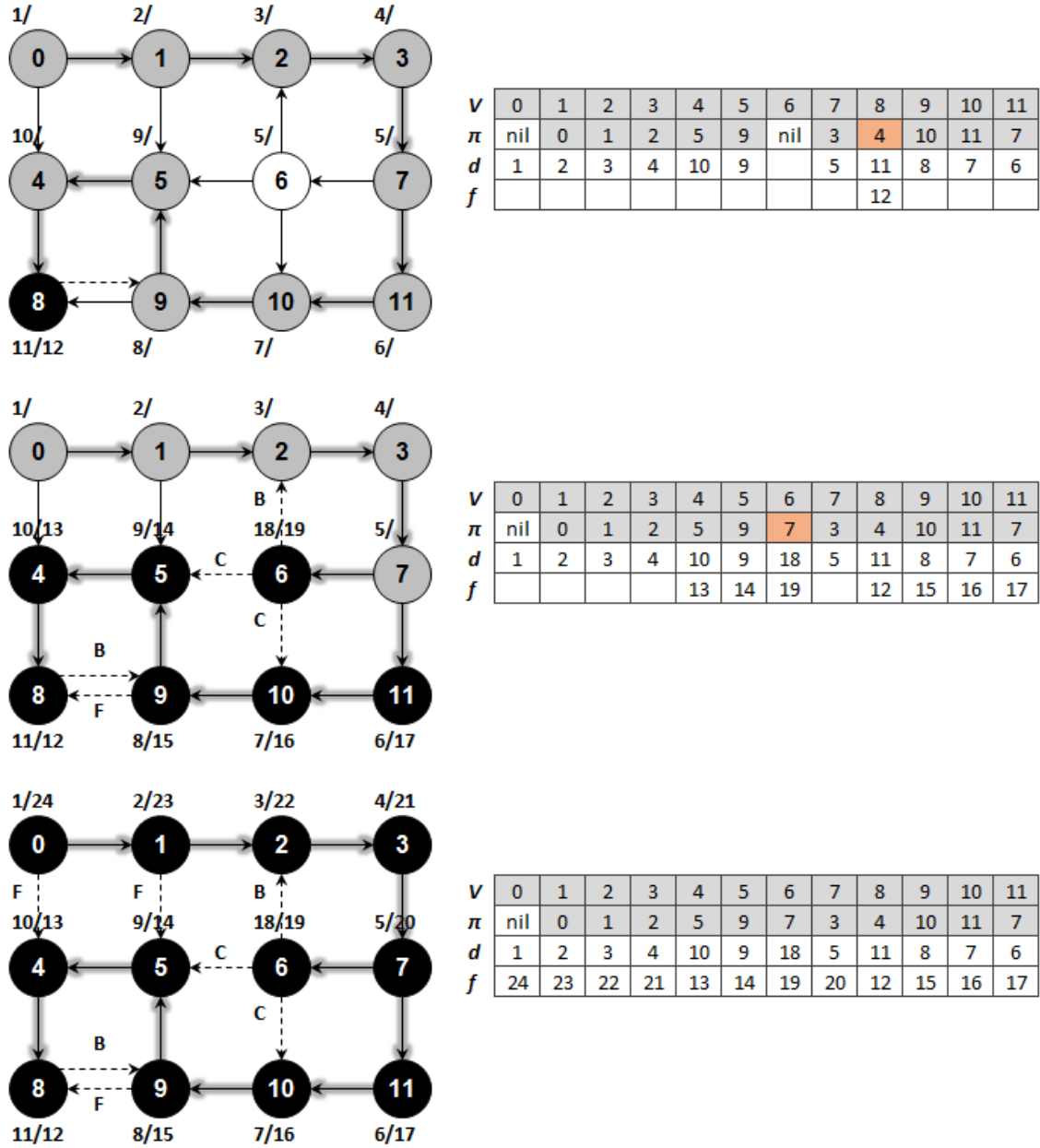


Figura 6: Rastreamento da DFS na lista de adjacências dada.

7. (CRLS 22.3-1) Desenhe uma tabela 3×3 , 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 *Tree*, *Back*, *Cross* e *Forward 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.

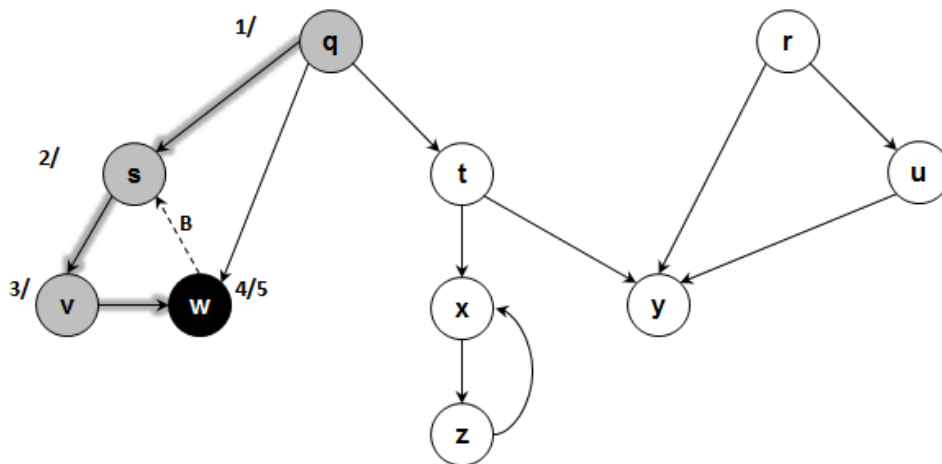
	White	Gray	Black
White	x	x	x
Gray	T	B	x
Black	x	B	B

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 π 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			

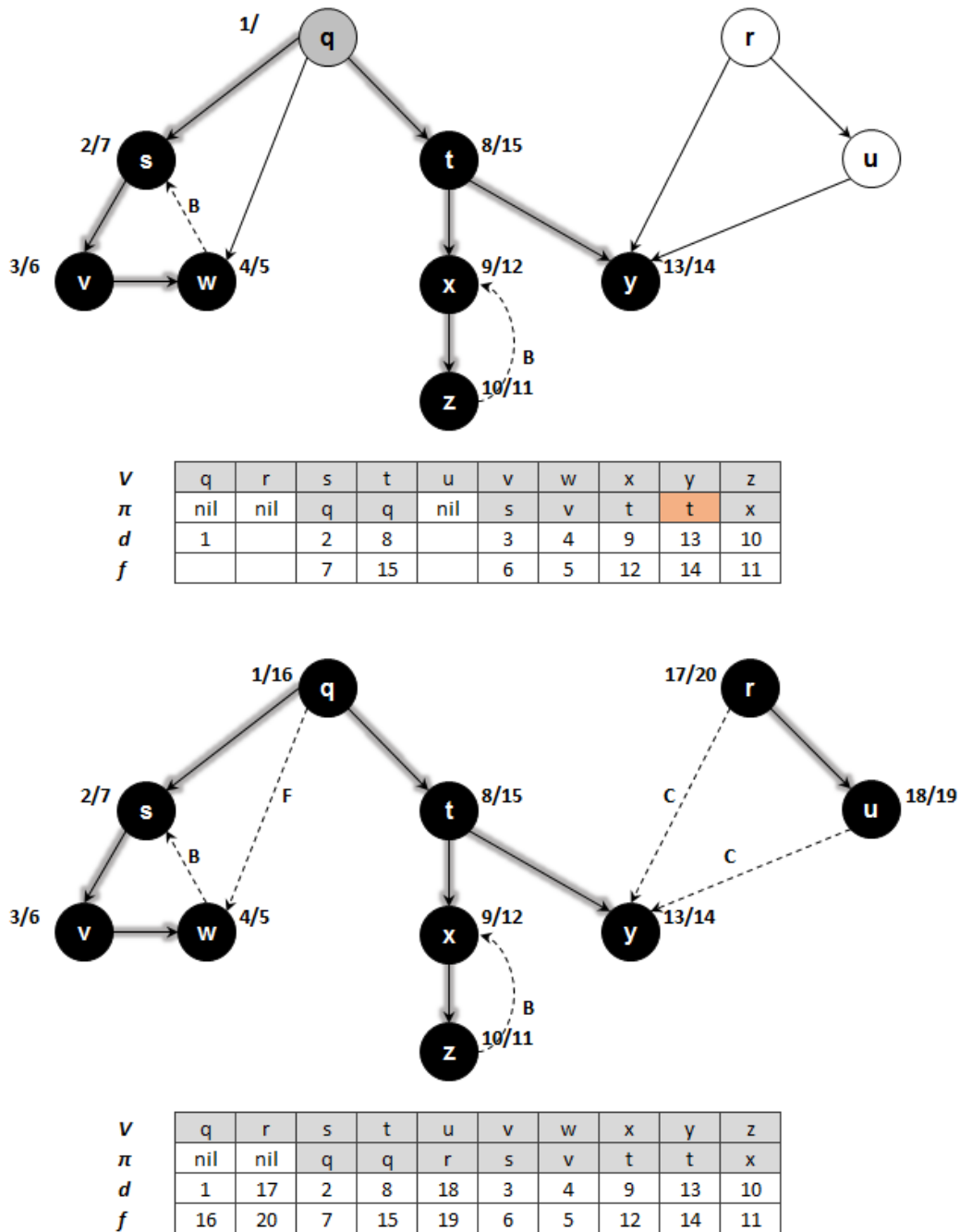


Figura 7: Rastreamento da DFS na figura 22.6 do CLRS.

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 .

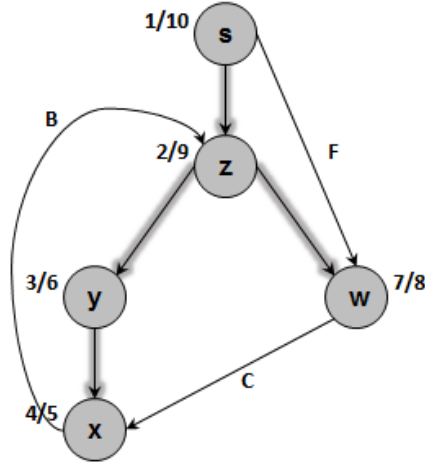


Figura 8: Contraexemplo utilizando parte do grafo da figura 22.5 (c) do CLRS.

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 $d[v] > f[u]$.

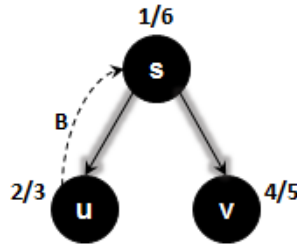


Figura 9: Contraexemplo da conjectura dada.

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 .

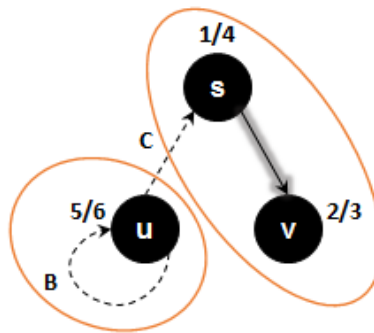


Figura 10: Exemplo onde o vértice u pode ficar isolado.

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 .

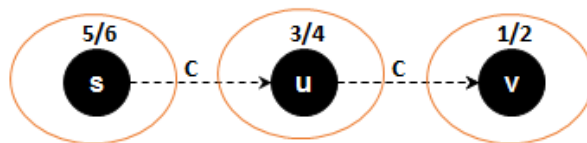


Figura 11: Segundo exemplo onde o vértice u pode ficar isolado.

13. Escreva uma generalização comum das buscas em largura e em profundidade. Sua função deve usar uma estrutura de dados auxiliar que pode operar como fila ou como pilha. Se a estrutura operar como fila, a função executa busca em largura, e se operar como pilha, a função executa busca em profundidade.

Resposta: Basta efetuarmos algumas alterações na versão iterativa da ITERATIVE-DFS para que a estrutura de dados opere de forma tal que ambas as buscas são atendidas. Basicamente, a ordem em que um vértice descoberto é incluído na pilha: se for busca em profundidade, incluímos no topo da pilha, se for em largura, incluímos na base (PUSH-LEFT).

No caso de busca em largura, a sub-rotina GRAYEN atualiza $d[v] = d[u] + 1$, considerando o caso onde o vértice é a origem da busca ($d[v] = 0$). Além disso, BLACKEN só atualiza $f[u]$ se for uma busca em profundidade.

Consumo de tempo: As alterações não afetam o consumo de tempo obtido na ITERATIVE-DFS, permanecendo $O(V + E)$, o que mantém o comportamento original tanto da DFS, quanto da BFS.

GFS($G, type$)

```

1  // let  $S$  be an empty stack
2  for each vertex  $u \in V[G]$ 
3       $color[u] = \text{WHITE}$ 
4       $\pi[u] = \text{NIL}$ 
5       $n = Adj[u].length$ 
6      for  $i = n$  to 1
7           $v = Adj[u][i]$ 
8          ENQUEUE( $Q_{adj}[u], v$ )
9   $time = 0$ 
10 for each vertex  $u \in V[G]$ 
11     if  $color[u] == \text{WHITE}$ 
12         GRAYEN( $u, \text{NIL}$ )
13         GFS-VISIT( $u$ )

```

GFS-VISIT(s)

```

1  PUSH( $S, s$ )
2  while  $S \neq \emptyset$ 
3       $u = \text{TOP}(S)$ 
4      if  $Qadj[u] \neq \emptyset$ 
5           $v = \text{DEQUEUE}(Qadj[u])$ 
6          if  $color[v] == \text{WHITE}$ 
7              GRAYEN( $v, u$ )
8              if  $type == \text{BFS}$ 
9                  PUSH-LEFT( $S, v$ )
10             else
11                 PUSH( $S, v$ )
12         else
13             BLACKEN( $u$ )
14             POP( $S$ )

```

GRAYEN(v, u)

```

1   $color[v] = \text{GRAY}$ 
2   $\pi[v] = u$ 
3  if  $type == \text{BFS}$ 
4      if  $u == \text{NIL}$  // it is a source vertex of a component
5           $d[v] = 0$ 
6      else
7           $d[v] = d[u] + 1$ 
8  else
9       $time = time + 1$ 
10      $d[v] = time$ 

```

BLACKEN(u)

```

1   $color[u] = \text{BLACK}$ 
2  if  $type == \text{DFS}$ 
3       $time = time + 1$ 
4       $f[u] = time$ 

```

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 pilha 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       $count = \text{CONNECTED-DFS}(G)$ 
5      if  $count > 1$ 
6          return FALSE
7       $u = \text{POP}(S)$ 
8       $\text{PUSH-LEFT}(S, u)$ 
9       $n = n - 1$ 
10 return TRUE

```

CONNECTED-DFS(G)

```

1  for each vertex  $u \in V[G]$ 
2       $color[u] = \text{WHITE}$ 
3       $\pi[u] = \text{NIL}$ 
4   $time = 0$ 
5   $count = 0$ 
6  for each vertex  $u \in V[G]$ 
7      if  $color[u] == \text{WHITE}$ 
8           $count = count + 1$ 
9           $\text{DFS-VISIT}(u)$ 
10 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, $\Theta(V + E)$.

16. Um grafo $G = (V, E)$ é bipartido se seu conjunto de vértices V pode ser bipartido em dois conjuntos disjuntos de vértices A e B e toda aresta de E tem uma ponta em A e outra em B . Escreva um algoritmo que, dado um grafo, determine se o grafo é ou não bipartido. Analise o seu consumo de tempo.

Resposta: Podemos efetuar uma alteração na BFS, de modo que quando um vértice u encontra um vértice v que já foi visitado, ou seja, já está GRAY, verificamos se a profundidade tanto de u quanto de v é par, ou se ambas são ímpar.

Isso implica que $d[u]$ e $d[v]$ têm a mesma paridade e, portanto, o grafo **não é bipartido**.

Consumo de Tempo: A modificação não altera o comportamento assintótico original da BFS que permanece $O(V + E)$.

BFS-PARTITE(G, s)

```

1  for each vertex  $u \in V[G] - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.partition = 0$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.partition = 1$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $u.partition == v.partition$ 
14             return FALSE
15         else
16             if  $v.color == \text{WHITE}$ 
17                  $v.color = \text{GRAY}$ 
18                  $v.d = u.d + 1$ 
19                  $v.partition = 3 - u.partition$ 
20                 ENQUEUE( $Q, v$ )
21      $u.color = \text{BLACK}$ 
22 return TRUE

```

17. (CLRS 22.2-8) Dada uma árvore $T = (V, E)$, o diâmetro de T é o número $\max\{d(u, v) : u, v \in V\}$, onde $d(u, v)$ é a distância entre u e v em T . Escreva um algoritmo que, dado T , determine o diâmetro de T . Analise o seu consumo de tempo.

Resposta: Conforme enunciado, o diâmetro de uma árvore T é a distância mais longa entre quaisquer dois nós u e v da árvore.

Para calcular o diâmetro podemos executar a BFS usando cada vértice $v \in V$ como fonte da busca, retornando o maior valor de d para cada busca. A maior distância será, então, o diâmetro de T .

A linha 19 deve retornar o número de vértices que tem o caminho mais longo de u a v . Repare que somamos 1 devido à inicialização de $s.d = 0$.

Consumo de Tempo: A inclusão da linha 18 não muda o consumo de tempo da BFS, que continua $O(V + E)$. Como executamos a BFS uma vez para cada vértice no *loop* da linha 3 da rotina TREE-DIAMETER, temos que o tempo total será $O(V(V + E))$.

TREE-DIAMETER(T)

```
1  diameter = 0
2  for each  $u \in T$ 
3       $max\_d = \text{BFS}(T, u)$ 
4      if  $max\_d > diameter$ 
5           $diameter = max\_d$ 
6  return  $diameter$ 
```

BFS(T, s)

```
1  for each vertex  $u \in V[T] - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in T.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
19 return  $u.d + 1$ 
```

Lista 8

1. (CRLS 23.1-1) Seja e uma aresta de custo mínimo em um grafo G com custos nas arestas. É verdade que e pertence a alguma MST de G ? É verdade que e pertence a toda MST de G ?

Resposta: Seja A um subconjunto de arestas de alguma MST T , tal que $(u, v) \notin A$. Para escolher uma aresta para ser adicionada em A , todas as arestas que atravessam um corte $(S, V - S)$ são consideradas. Como o corte deve respeitar A e a aresta (u, v) tem peso mínimo, ela será escolhida para ser incluída em A , o que dá origem a uma nova MST T' , onde $A \cup \{(u, v)\} \subseteq T'$.

É verdade que e pertence a toda MST de G ? Não é verdade. Se tomarmos uma aresta $(u, v) \in E$ de uma MST T que tenha custo mínimo k e substituirmos por outra (u, w) que não está em T de custo $k = l$, teremos uma nova MST T' , ou seja, é o caso onde há empate na escolha da aresta segura para ser incluída na árvore. Portanto, e não pertence a toda MST de G .

Na figura 12, basta substituirmos a aresta (b, c) pela (b, d) e teremos uma nova MST de mesmo custo mínimo.

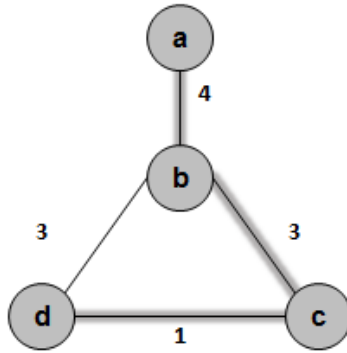


Figura 12: MST com mais de uma aresta de mesmo custo.

2. Suponha que os custos das arestas de um grafo conexo são distintos dois a dois (ou seja, não há duas arestas com o mesmo custo). Mostre que o grafo tem uma única MST.

Resposta: Seja m a quantidade de arestas em um grafo ponderado $G(V, E, w)$. Como não há arestas com pesos iguais, temos que os pesos são estritamente crescentes, ou seja, $w_1 < w_2 < w_3 < \dots < w_m$.

Prova por contradição:

Vamos assumir que existem duas MST's T e T' no grafo. Seja e_1 uma aresta de menor custo que aparece em uma dessas MST's. Sem perda de generalidade, digamos que $e_1 \in T$. Como T' é uma MST, $T' \cup \{e_1\}$ contém um ciclo C e, portanto, uma das arestas deste ciclo, digamos e_2 , não está em T .

Note que $w(e_1) < w(e_2)$ e $T'' = T' \cup \{e_1\} \setminus \{e_2\}$ é uma árvore geradora. O peso total de T'' é menor que o peso total de T' , mas isso é uma contradição, já que nós assumimos que T' é uma MST. Em outras palavras, para que T e T' fossem MST's, deveríamos ter $w(e_1) = w(e_2)$, mas isso é impossível, já que os pesos são diferentes.

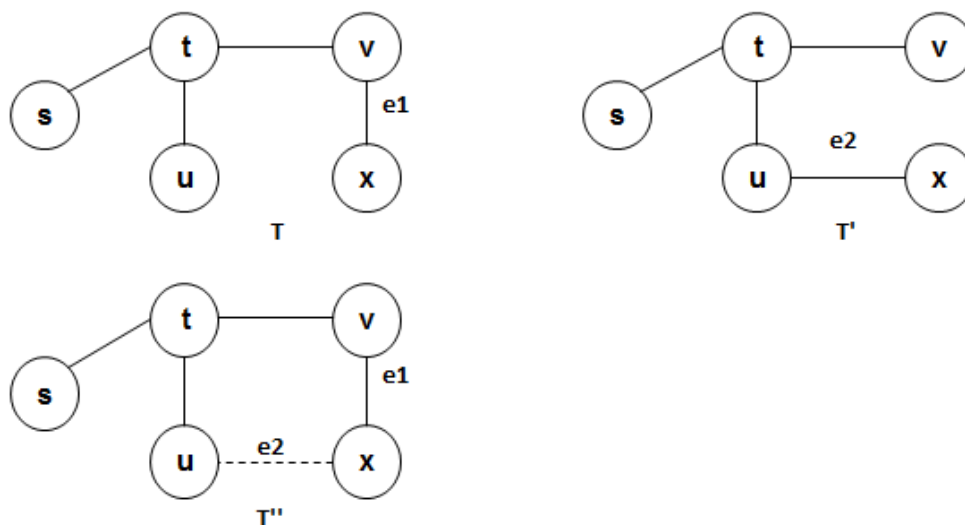


Figura 13: Exemplo de grafo com uma única MST com pesos nas arestas diferentes.

3. Suponha que os custos das arestas de um grafo conexo são distintos dois a dois. Seja C um ciclo não trivial. É verdade que a aresta de custo mínimo em C pertence à (única) MST do grafo?

Resposta: Sim, é verdade. Seja e a aresta de custo mínimo em C . Se e não estivesse na única MST do grafo, ao removermos qualquer outra aresta do ciclo C de custo maior para incluir e , teríamos uma MST de custo menor.

4. (CRLS 23.1-2) Prove ou desprove a seguinte afirmação: Dado um grafo G com pesos nas arestas, um conjunto de arestas A de G , e um corte que respeita A , toda aresta que cruza o corte e que é segura para A tem peso mínimo dentre todas as arestas desse corte.

Resposta: Seja A formado pelas arestas $\{(t, s), (t, u)\}$, como pode ser visto na figura 14.

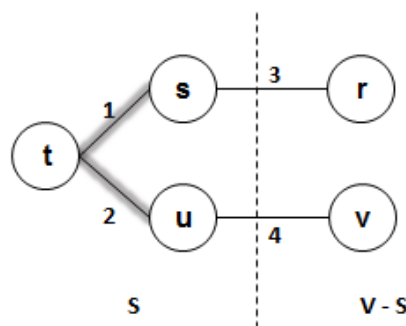


Figura 14: Exemplo de grafo com uma única MST contendo aresta de não peso mínimo no corte.

O corte $(S, V - S)$ respeita A . (u, v) é uma aresta que atravessa o corte $(S, V - S)$ e é segura para A , ou seja, $A \cup \{(u, v)\} \subseteq T$, porém, não é de custo mínimo (*light edge*).

5. (CRLS 23.1-3) Prove ou desprove a seguinte afirmação: Se uma aresta está contida em alguma MST, então tem peso mínimo dentre todas as arestas de algum corte no grafo.

Resposta: Seja T uma MST que contém a aresta (u, v) . Se tirarmos (u, v) da árvore, teremos um corte $(S, V - S)$ que particiona os vértices em dois conjuntos disjuntos S e $V - S$. A aresta (u, v) atravessa esse corte e ela é uma aresta de custo mínimo pois, do contrário, outra aresta de custo menor poderia ser adicionada em T substituindo (u, v) , o que produziria uma outra MST T' , onde $w(T') < w(T)$, contradizendo o fato de que T é uma MST.

Portanto, (u, v) é uma aresta de custo mínimo para o corte $(S, V - S)$.

5.1. (CLRS 23.1-4) Dê um exemplo simples de um grafo tal que o conjunto de arestas $\{(u, v) : \text{existe um corte } (S, V - S) \text{ tal que } (u, v) \text{ é uma aresta de custo mínimo atravessando o corte}\}$ não forma uma MST.

Resposta: Basta pensarmos em um triângulo com 3 vértices conectados por arestas de mesmo peso $w(e_1) = w(e_2) = w(e_3)$. Independente do corte, nós sempre teremos uma aresta de custo mínimo que não estará na MST pois, do contrário, teríamos um ciclo.

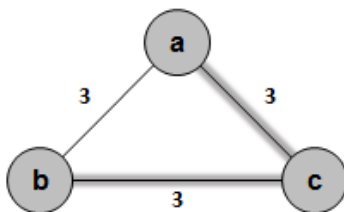


Figura 15: Exemplo de grafo onde uma aresta de custo mínimo não pertence à MST.

6. (CRLS 23.1-7) Prove que se todos os pesos nas arestas são positivos, então qualquer subconjunto de arestas que conectam todos os vértices e tem peso total mínimo forma uma árvore. A propriedade vale se alguns pesos são negativos?

Resposta:

7. Seja T uma MST de um grafo com pesos positivos e distintos nas arestas. Suponha que substituamos cada peso pelo seu quadrado. Verdadeiro ou falso: T ainda é uma MST para o novo grafo.

Resposta: Verdadeiro. Substituir os pesos das arestas em um grafo ponderado G pelo quadrado dos mesmos não muda a MST T do grafo.

Prova por contradição:

Vamos assumir que a alteração dos pesos w muda a MST T de G para uma outra T' em G' , já que pelo menos uma aresta, digamos e , de T deve ser substituída por uma outra e' na nova árvore T' de G' com w^2 . Isso implica que $w(e) < w(e')$ e $w^2(e) \geq w^2(e')$, o que é uma contradição.

8. Dado um grafo conexo G , dizemos que duas MSTs T e T' são vizinhas se T contém exatamente uma aresta que não está em T' , e T' contém exatamente uma aresta que não

está em T . Vamos construir um novo grafo (muito grande) \mathcal{H} como segue. Os vértices de \mathcal{H} são as $MSTs$ de G , e existe uma aresta entre dois vértices em \mathcal{H} se os correspondentes $MSTs$ são vizinhas. É verdade que \mathcal{H} sempre conexo? Prove ou dê um contra-exemplo

Resposta: Seja G um grafo e \mathcal{H} o grafo que tem como vértices todas $MSTs$ de G e dois vértices de \mathcal{H} são adjacentes se e somente se as $MSTs$ que eles representam são vizinhas. Queremos provar que \mathcal{H} é conexo.

Se para qualquer par de $MSTs$ T e T' de G , as árvores são vizinhas então não temos nada a provar, \mathcal{H} é conexo.

Se existem T e T' não vizinhos então achamos um caminho de T a T' em \mathcal{H} da seguinte forma.

Sabemos que uma árvore tem $n - 1$ vértices para $n = |V(G)|$ e se inserimos uma aresta a mais nessa árvore geraremos um ciclo, então podemos montar um caminho de T a T' em \mathcal{H} , inserindo uma aresta e' em T tal que $e' \in E(T')$, $e' \notin E(T)$ e $e' \in E(G)$ formando assim um ciclo em T então removemos uma aresta e desse ciclo em T , tal que $e \in E(T)$, $e \notin E(T')$ e $e \in E(G)$, gerando assim uma MST T'' mais parecida com T' , note que isso é possível por que tanto e como e' pertencem a uma MST e portanto tem pesos mínimos no ciclo formado. Repetindo esse processo chegaremos a uma MST T^* que possui apenas uma aresta diferente de T' , então concluímos que T' e T^* são vizinhas. Portanto podemos concluir que existe um caminho entre T e T' em \mathcal{H} e como isso é verdade para todos pares de $MSTs$ T e T' de \mathcal{H} , logo concluímos que \mathcal{H} é conexo.

9. Seja G um grafo conexo com custos nas arestas. Uma aresta e de G é crítica se o aumento do custo de e faz com que o custo de uma MST de G também aumente. Escreva uma função que determine todas as arestas críticas de G em tempo $O(m \log n)$

Resposta: Seja D_A a floresta representada pela estrutura de conjuntos disjuntos do algoritmo de Kruskal, e $T, T' \in D_A$ árvores mantida nessa floresta. Uma aresta (u, v) tal que $u \in V(T)$ e $v \in V(T')$ é crítica se e somente se (u, v) é uma *light edge* que respeita A e não exista outra *light edge* que junte T e T' em D_A .

Prova:

→ se (u, v) é crítica então (u, v) é uma *light edge* e não existe nenhuma outra *light edge* que junte T e T' em D_A .

Verdade por que caso existisse outra *light edge* (u', v') então poderíamos utiliza-la mantendo o mesmo peso da MST independente do peso de (u, v) , ou seja, poderíamos aumentar o peso de (u, v) sem aumentar o peso de uma MST de G .

← se (u, v) é a única *light edge* que junta as árvores T e T' então (u, v) é uma aresta crítica.

Trivialmente verdade, já que para montar uma MST precisamos utilizar (u, v) então se aumentarmos o peso de (u, v) ou escolhermos outra aresta (u', v') aumentaremos o peso de qualquer MST de G , já que (u', v') tem peso maior que (u, v) já que por hipótese (u, v) é a única *light edge* que junta T e T' .

Portanto ao juntar dois componentes no algoritmo de Kruskal podemos ver se a aresta é crítica ou não conforme algoritmo CRITICAL-KRUSKAL.

CRITICAL-KRUSKAL(G)

```

1   $l = NIL, A = \emptyset, C = \emptyset$ 
2  ordene as arestas de  $E(G)$  em ordem crescente de peso  $w_e$ 
3  for each  $v \in V(G)$ 
4      MAKE-SET( $v$ )
5  for each  $uv \in E(G)$  in the sorted edges
6      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           $A = A \cup \{uv\}$ 
8          UNION( $uv$ )
9          if  $l \neq NIL$ 
10              $C = C \cup \{l\}$ 
11              $l = uv$ 
12      else
13          if  $l \neq NIL$  and  $w(l) = w(uv)$ 
14              $l = NIL$ 
15  return  $C$ 

```

O algoritmo mantém a mesma estrutura de dados do que o Kruskal a linha 6, acha se a aresta sendo analisada estão na mesma árvore se sim, junta e guarda a aresta anterior é *light edge* que junta uv (já que as arestas estão ordenadas), caso contrário se a aresta não junta dois componentes e tem o mesmo peso então a aresta l não é crítica e recebe NIL , para não ser inclusa em C .

10. Mostre que depois de cada execução da linha 12 do algoritmo de Prim tem-se $key[u] < \infty$.

Resposta: Note que antes do *while* da linha 6, todo vértice $u \in Q$ e $key[u] = \infty$, exceto o vértice r pelo qual iniciamos a busca, onde $key[r] = key[u] = 0$. Trivialmente, $key[r] = key[u] < \infty$. Portanto, $r \in V - Q$ e isso nos dá um corte no grafo.

O *loop* das linhas 8-11 atualiza $key[v]$ para todo $v \in Adj[u]$ que ainda está em Q . Como inicializamos key com infinito, ele será atualizado pelo menos uma vez. Logo, a fila de prioridade será reorganizada e o vértice cujo key tiver o menor custo $w(u, v)$ será o novo u na próxima iteração. Isso faz com que os vértices em Q recebendo arestas do corte oriundas do conjunto $V - Q$ sempre tenham $key < \infty$. Como o *loop* é feito até que $Q = \emptyset$, ao final teremos $key[u] < \infty$ para todo $u \in V[G]$.

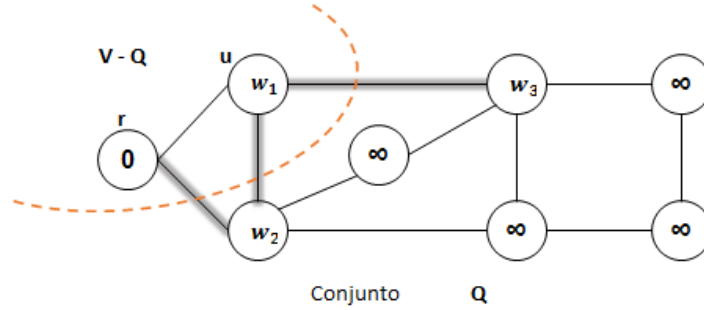


Figura 16: Exemplo de corte no grafo mostrando que o atributo $key[v] < \infty$ para os vértices adjacentes a u . As arestas sombreadas são aquelas atravessando o corte.

11. Suponha que temos um grafo G com pesos nas arestas. Verdadeiro ou falso: Para qualquer MST T de G , existe uma execução válida do algoritmo de Kruskal que produz T como saída? Dê uma prova ou um contra-exemplo.

Resposta: Verdadeiro. Vamos assumir por contradição que existe uma MST T' que nunca será gerada por uma execução do algoritmo de Kruskal. Então existe uma aresta $(u, v) \in E(T')$ que liga dois conjuntos da estrutura UNIONFIND D_A mantida pelo algoritmo de Kruskal. Então devemos analisar 2 casos.

1. Se (u, v) é uma aresta de peso mínimo que liga o conjunto que contém u com o conjunto que contém v então existe uma ordenação em que o algoritmo de Kruskal gera uma MST que a contém
2. Se (u, v) não possui peso mínimo entre todas as arestas que ligam o conjunto que contém u com o conjunto que contém v , então podemos diminuir o peso de T' escolhendo uma aresta de menor peso. Que é uma contradição já que T' não é uma MST .

Note que isso é verdade para qualquer aresta (u, v) , portanto T' não existe. Que é uma contradição. Logo podemos concluir que o algoritmo de Kruskal pode gerar qualquer árvore geradora mínima de um grafo.

13 (CLRS 23.2-4) Suponha que todos os pesos num grafo com n vértices são inteiros no intervalo de 1 até n . Descreva como otimizar os algoritmos de Kruskal e Prim nesta situação. O que acontece se os pesos são intervalo de 1 até W ?

Resposta: Para o algoritmo de Kruskal utilizando a estrutura de conjuntos disjuntos utilizando UNION-BY-RANK e compressão de caminhos o tempo gasto assintoticamente é definido pela ordenação de arestas, no começo do algoritmo, se sabemos que o peso máximo das arestas é n , então podemos usar o COUNTING-SORT e diminuir o tempo de $O(m \lg n)$ para $O(m + n)$, já que o laço abaixo da ordenação gasta tempo linear, se o peso máximo é W então utilizando o COUNTING-SORT o algoritmo de Kruskal gastará tempo $O(m + W)$. Como o algoritmo de Prim utiliza uma fila de prioridades que normalmente é implementada como uma min-heap não é possível melhorar o tempo assintótico do algoritmo.

14. Dado um grafo com n vértices, pesos distintos nas arestas, e no máximo $n+8$ arestas, dê um algoritmo com complexidade $O(n)$ para achar uma MST.

Resposta: Neste caso, a propriedade do ciclo será aplicada por 9 vezes. Nós podemos executar a BFS até encontrar um ciclo no grafo G e, então, nós deletamos a aresta de maior custo neste ciclo. Isso faz com que o número de arestas em G seja reduzido em um, ao mesmo tempo em que G continua conectado, sem alterar a identidade da MST. Se fizermos isso por 9 vezes, teremos um grafo conectado G' com $n-1$ arestas e com a mesma MST de G . Porém, G' é uma árvore e, portanto, ele deve ser uma MST de G .

O tempo de execução de cada iteração toma $O(E+V)$ para a BFS e subsequente verificação do ciclo para encontrar a aresta de maior custo. Como $E \leq n+8$ e há um total de 9 iterações, o tempo de execução total é $O(n)$.

15. O diâmetro de um grafo é o máximo das distâncias entre dois vértices. Escreva código que usa o algoritmo de Dijkstra para calcular o diâmetro de um grafo.

Resposta: Para calcular o diâmetro de um grafo G podemos aplicar o DIJKSTRA usando cada vértice $v \in V[G]$ como fonte da busca, retornando o maior valor de d para cada busca. A maior distância será, então, o diâmetro de G .

GRAPH-DIAMETER(G, w)

```

1  diameter = 0
2  for each  $u \in V[G]$ 
3       $max\_d = \text{DIJKSTRA-DIAMETER}(G, w, u)$ 
4      if  $max\_d > diameter$ 
5           $diameter = max\_d$ 
6  return diameter
```

DIJKSTRA-DIAMETER(G, w, s)

```

1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = V[G]$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each  $v \in \text{Adj}[u]$ 
8          RELAX( $u, v, w$ )
9  return  $u.d$  // the last  $u$  removed from  $Q$ 
```

Consumo de Tempo: A inclusão da linha 9 no DIJKSTRA-DIAMETER não muda o consumo de tempo do DIJKSTRA, que continua $O(V \lg E)$. Como executamos uma vez para cada vértice no *loop* da linha 2 da rotina GRAPH-DIAMETER, temos que o tempo total será $O(V(V \lg E))$.

16. Considere um dígrafo (grafo orientado) com custos positivos associados aos vértices. O custo de um caminho num tal dígrafo é a soma dos custos dos vértices do caminho. Queremos encontrar um caminho de custo mínimo dentre os que começam num vértice s e terminam num vértice t . Adapte o algoritmo de Dijkstra para resolver esse problema.

Resposta: Basta pararmos o algoritmo de Dijkstra quando o vértice t for encontrado. Como a estratégia do algoritmo é gulosa, ao retirarmos o vértice t da fila de prioridade, já teremos o caminho mínimo de s a t .

DIJKSTRA-SINGLE-PAIR(G, w, s, t)

```

1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = V[G]$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      if  $u == t$ 
8          return
9      for each  $v \in \text{Adj}[u]$ 
10         RELAX( $u, v, w$ )

```

Consumo de Tempo: A inclusão das linhas 7-8 não muda o comportamento assintótico original do DIJKSTRA, que continua que continua $O(VlgE)$.

17. Sejam s e t dois vértices de um dígrafo com custos positivos nos arcos. Para cada vértice v do dígrafo, seja $x[v]$ o custo de algum caminho de s a v . Escreva um algoritmo eficiente que verifique se $x[t]$ é a distância de s a t em G . Explique porque seu algoritmo está correto.

Resposta: Assumindo que x é uma função que pressupõe a distância de s a t , basta utilizarmos o algoritmo DIJKSTRA-SINGLE-PAIR do exercício 16 como sub-rotina. Descobrimos a distância de s a t e, logo depois, verificamos se a distância dada por $x[t]$ corresponde ao menor caminho encontrado que fica no atributo d de cada vértice do grafo dado por G .

CHECK-DISTANCE(G, w, x, s, t)

```

1  DIJKSTRA-SINGLE-PAIR( $G, w, s, t$ )
2  if  $x[t] == d[t]$ 
3      return TRUE
4  else
5      return FALSE

```

Consumo de tempo: Como vimos, o algoritmo DIJKSTRA-SINGLE-PAIR toma tempo $O(VlgE)$ e nós o chamamos apenas uma vez na linha 1. As demais linhas consomem tempo constante $\Theta(1)$. Logo, o consumo de tempo total é $O(VlgE)$.

18. (CLRS 24.3-2) Mostre que o algoritmo de Dijkstra pode produzir resultados errados se o dígrafo tiver arcos de custo estritamente negativo.

Resposta: Um exemplo para dígrafos pode ser visto na figura 17.

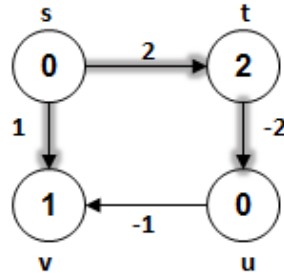


Figura 17: Dígrafo com arcos estritamente negativos.

A execução de todas as iterações do DIJKSTRA produz o caminho mínimo destacado pelas arestas sombreadas. O valor de $d[v]$ está errado, pois existe um caminho mínimo de $s \rightsquigarrow v$ de custo menor que 1, por exemplo, o caminho $p' = s \rightsquigarrow t \rightsquigarrow u \rightsquigarrow v$, onde $w(p') = -1 < 1$.

21. (CLRS 24.3-3) Suponha que trocamos a linha 4 do algoritmo do Dijkstra como segue

4. *while* $|Q| > 1$

Isso faz com que a execução do laço execute $|V| - 1$ vezes no lugar de $|V|$ vezes. Será que o algoritmo continua correto?

Resposta: Sim, o algoritmo continua funcionando. Seja u o vértice restante que não é extraído da fila de prioridades Q . Se u não é alcançável de s , então $d[u] = \delta(s, u) = \infty$. Se u é alcançável de s , existe um caminho mínimo $p = s \rightsquigarrow x \rightarrow u$.

Quando o vértice x foi extraído de Q , $d[x] = \delta(s, x)$ e, então, a aresta (x, u) já foi "relaxada" e, portanto, $d[u] = \delta(s, u)$.

23. Seja $G = (V, E)$ um dígrafo com pesos $w : E \rightarrow \{0, 1, \dots, W\}$ para algum W . Modifique o algoritmo de Dijkstra para que compute os caminhos mínimos a partir de um vértice s em tempo $O(W|V| + |E|)$.

Resposta: O tempo de execução do algoritmo de Dijkstra depende da implementação da *min-priority queue*. No algoritmo de Dijkstra, processamos os vértices mais próximos do vértice s origem da busca primeiro. Como cada aresta tem no máximo peso W , sabemos que o custo máximo possível do caminho mais longo no grafo é $(V - 1)W$. Nós podemos priorizar os vértices com base nos seus atributos $d[]$, lembrando que $d[v]$ é o caminho mais curto entre o vértice s e v .

A fila de prioridades é composta por $(V - 1)W$ *buckets*. O vértice v pode ser encontrado no *bucket* $d[v]$. Uma vez que todos os vértices, exceto s , têm valor $1 < d[v] < (V - 1)W$, eles podem ser encontrados nos *buckets* $1, \dots, (V - 1)W$. Se s é o vértice origem, $d[s] = 0$ e, portanto, s pode ser encontrado no *bucket* 0.

A sub-rotina INITIALIZE-SINGLE-SOURCE garante que para todos os vértices v diferentes da raiz, $d[v] = \infty$. O último *bucket* contém todos os vértices y cujo $d[y] = \infty$, ou seja, aqueles que ainda não foram descobertos.

Após inicializar todos os vértices, nós rastreamos os *buckets* de 0 a $(V-1)W$. Quando um *bucket* não vazio é encontrado, o primeiro vértice é removido e todos os vértices adjacentes a ele são "relaxados". Este passo é repetido até que chegamos ao final da fila, em tempo $O(WV)$. Como nós "relaxamos" um total de E arestas, o tempo total de execução para este algoritmo é $O(VW + E)$.

Lista 9

1. Defina *algoritmo eficiente*. Defina *problema de decisão*. Defina *verificador polinomial* para SIM. Defina *verificador polinomial* para NÃO. Defina as classes P, NP e coNP. Dê um exemplo de um problema em cada uma dessas classes, justificando a sua pertinência à classe.

Resposta:

Algoritmo eficiente: um algoritmo é eficiente se ele resolve um dado problema em tempo polinomial, ou seja, se o seu consumo de tempo no pior caso é limitado por um polinômio no tamanho das instâncias do problema. Em outras palavras, o algoritmo é polinomial se ele resolve o problema em tempo $O(n^k)$ para alguma constante k .

Problema de decisão: aquele cuja solução é uma resposta do tipo SIM/NÃO.

Verificador polinomial para SIM a um problema π é um algoritmo polinomial A tal que

1. para qualquer instância X de π com resposta SIM, existe um Y em Σ^* , tal que $A(X, Y)$ devolve SIM
2. para qualquer instância X de π com resposta NÃO, existe um Y em Σ^* , tal que $A(X, Y)$ devolve NÃO
3. A consome tempo polinomial em $|X|$

Verificador polinomial para NÃO a um problema π é um algoritmo polinomial A tal que

1. para qualquer instância X de π com resposta **NÃO**, existe um Y em Σ^* , tal que $A(X, Y)$ devolve SIM
2. para qualquer instância X de π com resposta **SIM**, existe um Y em Σ^* , tal que $A(X, Y)$ devolve NÃO
3. A consome tempo polinomial em $|X|$

Onde Y é um certificado para SIM/NÃO da instância X de π .

Classe P: conjunto dos problemas de decisão tratáveis, isto é, que são solúveis em tempo polinomial. **Exemplo**: problema da mochila fracionária está em P , já que pode ser resolvido em tempo $O(n \lg n)$.

Classe NP: conjunto dos problemas de decisão que possuem um verificador polinomial para a resposta SIM, ou seja, se existe um algoritmo que, ao receber uma instância X de π e uma suposta solução S de X , responde SIM ou NÃO conforme S seja ou não solução de X , e consome tempo limitado por um polinômio no tamanho de X para responder SIM. **Exemplo**: O problema do ciclo hamiltoniano está em NP, pois é possível verificar em tempo polinomial se uma dada permutação dos vértices é um ciclo do grafo.

Classe coNP: consiste nos problemas de decisão que são complementos de problemas de decisão em NP, ou seja, problemas para os quais existe um certificado (curto) para a resposta NÃO. **Exemplo**: O problema do quadrado perfeito está em coNP, pois é possível verificar em tempo polinomial se um certo número natural n **não** é um quadrado perfeito. Em outras palavras, basta exibir um número natural k tal que $k^2 < n < (k + 1)^2$. Um tal k é um certificado de que n não é um quadrado perfeito.

2. Mostre que SAT está em NP. (Essa é a parte fácil do teorema de Cook.)

Resposta: Para mostrar que o SAT está em NP, temos que mostrar que existe um algoritmo $A(C, X, Y)$ que verifica o certificado Y que devolve SIM se Y pertence ao conjunto de soluções para a instância C, X .

Seja C um conjunto de cláusulas sobre um conjunto de variáveis X , então podemos considerar C e X nossa instância do problema SAT. Seja Y uma atribuição das variáveis de X , podemos considerar Y como nosso certificado, já que se as cláusulas C forem satisfazíveis por Y então a fórmula é satisfazível. É fácil notar que podemos escrever um algoritmo que leia X, C e Y e imprime SIM se e somente se Y satisfaz C , além disso esse algoritmo pode ser implementado de forma a consumir tempo polinomial em $|X|$ e $|C|$, (note que $|Y| = |X|$). Portanto SAT pode ser verificado e certificado em tempo polinomial, logo SAT está em NP.

3. Uma coleção C de cláusulas sobre um conjunto X de variáveis booleanas é uma tautologia se toda atribuição a X satisfaz C . O problema TAUTOLOGIA consiste em, dado X e C , decidir se C é ou não uma tautologia. O problema TAUTOLOGIA está em NP? Está em coNP? Justifique suas respostas.

Resposta:

4. O problema 2-SAT consiste na restrição de SAT a instâncias X e C em que toda cláusula de C tem exatamente dois literais. Mostre que o 2-SAT está em P, ou seja, descreva um algoritmo polinomial que resolva o 2-SAT.

Resposta:

5. Mostre que 2-COLORAÇÃO está em P.

Resposta:

6. Seja $G = (V, E)$ um grafo. Um conjunto $S \subseteq V$ é independente se quaisquer dois vértices de S não são adjacentes. Ou seja, não há nenhuma aresta do grafo com as duas pontas em S . O problema IS consiste no seguinte: dado um grafo G e um inteiro $k \geq 0$, existe um conjunto independente em G com k vértices? Mostre que IS é NP-completo.

Resposta:

7. Seja $G = (V, E)$ um grafo. Uma 3-COLORAÇÃO de G é uma função $c : V \rightarrow 1, 2, 3$ tal que $c(u) \neq c(v)$, para toda aresta $uv \in E$. Considere o

Problema 3-COLORAÇÃO: Dado um grafo, determinar se ele tem ou não uma 3-COLORAÇÃO.

Mostre que o 3-COLORAÇÃO está em NP.

Resposta:

8. Mostre que o problema abaixo é NP-completo.

Problema PARTIÇÃO: Dada uma coleção S de números, decidir se existe uma sub-coleção S' de S cuja soma é igual a soma dos números em S menos S' , ou seja,

$$\sum_{x \in S} x = \sum_{x \notin S} x$$

Resposta:

9. Mostre que o problema abaixo é NP-completo.

Problema MOCHILA: Dado um número W , um número V , um número inteiro positivo n , uma coleção de números w_1, \dots, w_n , e uma coleção de números v_1, \dots, v_n , decidir se existe um subconjunto S de $\{1, \dots, n\}$ tal que

$$\sum_{i \in S} w_i \leq W \text{ e } \sum_{i \in S} v_i \geq V$$

Resposta:

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:

$$\prod_{k=1}^3 2^{2k+1} = 2^{2+1} \times 2^{4+1} \times 2^{6+1}$$

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

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

Portanto:

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