Gabarito da lista de exercícios 04 - Conceitos e algoritmos em grafos

Este gabarito só considera as questões que envolviam demonstrações. Lembrar que as soluções aqui descritas não são únicas e os exercícios podiam ter sido resolvidos por diferentes vias.

1. Escolha um dos seguintes itens para entregar:

(a) Se um grafo possui um passeio fechado que passa exatamente única vez por cada aresta, dizemos que o grafo é **euleriano**. Suponha que em um grafo conexo G todo vértice tem grau par. Mostre que G é euleriano.

Resposta. Seja G um grafo conexo com $n \geq 3$ vértices onde todo vértice tem grau par. O caso $n = 1$ é trivial e o caso $n = 2$ não pode ser euleriano pois em um grafo conexo com dois vértices, cada vértice tem grau 1, pois só possui uma aresta (não consideramos multigrafos neste exercício).

Realizaremos a prova por indução matemática no número de arestas m.

Como o grafo é conexo e o grau de qualquer vértice é par, cada vértice deve ter pelo menos grau 2.

A soma dos graus dos vértices é duas vezes o número de arestas (Teorema do aperto de mãos), portanto G deve ter pelo menos n arestas ($\sum_{v \in V} d(v) \geq \sum_{v \in V} 2 = 2n$).

- *Caso base.* $m = n$. Nesse caso, todo vértice tem grau 2, pois, caso contrário, pelo Teorema do Aperto de Mãos, haveriam vértices com grau 1 (ímpar) ou 0 (G seria desconexo).

  Como um grafo acíclico maximal tem $n - 1$ arestas, G deve possuir pelo menos um ciclo. Seja C um ciclo de G.

  Se houver um vértice u fora de C, como G é conexo, deve existir um caminho de u até algum vértice de C. Denotemos por v_u o vértice de C mais próximo de u. Então, v_u teria grau pelo menos 3 (as duas arestas de C que incidem nele mais uma aresta do caminho até u), o que contradiz a hipótese de que todos os vértices têm grau 2.

  Logo, não podem existir vértices fora de C. Como em C todos os vértices têm exatamente duas arestas incidentes, concluímos que todas as arestas de G estão em C, e, portanto, G é um grafo euleriano.

- *Passo.* $m > n$. Como um grafo acíclico maximal tem $n - 1$ arestas, G deve possuir pelo menos um ciclo. Seja C um ciclo de G e denote por G_C o grafo obtido ao remover de G as arestas pertencentes a C.

  Observe que, em G_C, o grau dos vértices de C é o grau que tinham em G, menos 2. Portanto, os vértices de C têm grau par em G_C. Para os vértices fora de C, o grau em G_C é o mesmo que em G, que também é par.

  Dessa forma, todos os vértices em G_C têm grau par, e cada componente de G_C é um grafo conexo no qual todos os vértices têm grau par.

  Aplicando a hipótese de indução em cada componente X de G_C, concluímos que existe um passeio fechado P_X que percorre todas as arestas da componente exatamente uma vez.

  Como G é conexo, cada componente X de G_C compartilha ao menos um vértice com o ciclo C. Denote por u_X um vértice comum entre X e C. Como P_X é um passeio fechado, ele pode ser escrito de forma a começar e terminar em u_X.

Assim, podemos substituir o vértice u_X em C pelo passeio P_X, obtendo um novo passeio fechado que percorre exatamente uma vez tanto as arestas de C quanto as de P_X.

Repetindo essa substituição para cada componente de G_C, obtemos um passeio fechado que percorre exatamente uma vez cada aresta de G. Concluindo que G é euleriano.

(b) Se um grafo possui um ciclo gerador, dizemos que é **hamiltoniano**. Suponha que em um grafo $G = (V, E)$, com $|V| \geq 3$, todo par de vértices não adjacentes u, v satisfaz que $d(u)+d(v) \geq |V|$. Mostre que G é hamiltoniano.

Resposta. Para provar este resultado, mostraremos que qualquer grafo não hamiltoniano não satisfaz a condição.

Seja G um grafo não hamiltoniano com $n \geq 3$ vértices, ou seja, um grafo que não possui um ciclo que passe exatamente uma vez por cada um de seus vértices.

A partir de G, construiremos um grafo maximal não hamiltoniano H. Esse grafo H é obtido adicionando arestas a G desde que elas não formem um ciclo hamiltoniano, até que não seja mais possível adicionar arestas sem criar tal ciclo.

Agora, sejam x e y dois vértices não adjacentes de H.

Note que adicionar a aresta (x, y) em H criaria um ciclo hamiltoniano, pela própria construção de H. Logo, já existe em H um caminho de x até y que passa por todos os vértices exatamente uma vez. Seja $P = (v_1, v_2, \ldots, v_n)$ tal caminho de $x = v_1$ até $y = v_n$ em H.

Para cada $1 \leq i \leq n-1$, pelo menos uma das arestas (v_1, v_{i+1}) ou (v_i, v_n) não pode estar presente em H. Caso contrário, o passeio fechado $(v_1, v_2, \ldots, v_i, v_n, v_{n-1}, \ldots, v_{i+1}, v_1)$ seria um ciclo hamiltoniano, o que contradiz a construção de H como um grafo não hamiltoniano.

Portanto, para cada um dos $n-1$ valores de i, pode haver no máximo uma aresta incidindo em v_1 ou v_n. Assim, temos que $d(v_1) + d(v_n) \leq n-1 < n$, o que mostra que a condição não é satisfeita por H.

Como G é um subgrafo gerador de H, a condição também não é válida para G.

Portanto, qualquer grafo não hamiltoniano não satisfaz a condição.

Consequentemente, se um grafo satisfaz a condição, ele deve ser hamiltoniano.

(c) Considere um grafo G. Um **emparelhamento** de G é um conjunto $M \subseteq E$ de arestas, tal que cada vértice em V incide no máximo em uma aresta de M. Uma **cobertura** de G é um conjunto $S \subseteq V$ de vértices tal que $G - S$ não possui arestas. Suponha que M^* é um emparelhamento máximo de G e S^* uma cobertura mínima de G. Mostre que se G for bipartido $|M^*| = |S^*|$.

Resposta. Seja $G = (A \cup B, E)$ um grafo bipartido com partições A e B, e seja M^* um emparelhamento máximo.

Em qualquer cobertura de G, um mesmo vértice não pode cobrir duas arestas diferentes de M^*, já que essas arestas não compartilham vértices. Portanto, uma cobertura mínima deve ter pelo menos $|M^*|$ vértices. Nosso objetivo é provar que a igualdade de fato ocorre.

Denotemos por X o subconjunto de vértices de A que não incidem em nenhuma aresta de M^* (note que X pode ser vazio).

Agora, denotemos por Y o conjunto de vértices que contém todos os vértices de X e aqueles que são alcançáveis a partir de algum vértice de X por um caminho alternante, isto é, um caminho que alterna entre arestas fora de M^* e arestas dentro de M^*.

Como G é bipartido, toda aresta tem uma ponta em A e outra em B.

Note que, para cada aresta de um caminho alternante, a ponta em B também está em Y. Logo, cada aresta em um caminho alternante tem exatamente uma ponta em $B \cap Y$.

Por outro lado, as arestas de G que não fazem parte de um caminho alternante têm uma ponta em $A \setminus Y$, pois as pontas em A dessas arestas não podem estar em Y.

Portanto, o conjunto $S^* = (A \setminus Y) \cup (B \cap Y)$ é uma cobertura de G.

Como $X \subseteq Y$, cada vértice em $A \setminus X$ incide em uma aresta de M^*. Além disso, como os vértices em $B \cap Y$ pertencem a caminhos alternantes que começam em algum vértice de A, cada vértice de $B \cap Y$ também incide em uma aresta de M^*. Assim, concluímos que cada vértice de S^* incide em uma aresta de M^* e $|S^*| = |M^*|$.

(d) Dado um grafo $G = (V, E)$, um emparelhamento $M \subseteq E$ **satura** um conjunto $S \subseteq V$, se cada vértice de S incide em uma aresta de M. M é um **emparelhamento perfeito** de G se satura V. Mostre que se G é bipartido e todo vértice tem o mesmo grau $k \geq 1$, então G possui um emparelhamento perfeito.

Resposta. Seja $G = (A \cup B, E)$ um grafo bipartido com partições A e B, onde todo vértice tem grau $k \geq 1$, e seja M^* um emparelhamento máximo.

Suponha que M^* não é perfeito, então, sem perda de generalidade, existem vértices em A que não possuem arestas incidentes em M^*. Seja u um desses vértices e considere todos os caminhos alternantes (ver item anterior) que começam em u.

Denote por X o conjunto de vértices nesses caminhos que estão em A (incluindo o próprio u) e por Y o conjunto de vértices desses caminhos que estão em B.

Temos que todo vértice em Y incide em uma aresta de M^*, caso contrário, teríamos um caminho alternante que começa e termina em vértices não saturados pelo emparelhamento, negando que este seja máximo. Note que, se existir um tal caminho, trocando as arestas estão em M^* pelas que não estão em M^*, obteríamos um novo emparelhamento que satura todos os vértices do caminho e tem uma aresta a mais.

Como cada vértice de Y incide em uma aresta de M^*, o número de vértices em X deve ser pelo menos o número de vértices em Y (a outra ponta das arestas de M^*), mais 1 (o vértice u, que não está emparelhado). Ou seja, $|X| \geq |Y| + 1$.

Por outro lado, cada vértice de X tem todos os seus vizinhos em Y e, como todo vértice tem grau k, o número de vértices em Y deve ser pelo menos $|X|$ (note que são k arestas saindo de cada vértice de X, isto é, $k|X|$ arestas que devem chegar em Y, onde nenhum vértice tem grau maior que k). Portanto, $|Y| \geq |X|$, o que contradiz que $|X| \geq |Y| + 1$.

Assim, concluímos que a nossa suposição de que M^* não seja perfeito é falsa.

2. Escolha dois dos seguintes itens para entregar:

(a) Analise o tempo e mostre a correção do seguinte algoritmo para encontrar componentes fortemente conexas. O algoritmo 1 é a função principal e o 2 é o chamado recursivo para encontrar as componentes:

Resposta. Começaremos analisando a complexidade:

- A inicialização dos vértices como brancos nas linhas 2-4 do **Main** requer $O(V)$ operações.
- Observe que o *Find_ CC* é chamado apenas se o vértice for branco (linhas 9 e 10 de cada função). Após a chamada do *Find_ CC*, o vértice se torna cinza (linha 2 do *Find_ CC*) e nunca volta à cor branca. Desta forma, durante a execução do algoritmo, temos $O(V)$ chamadas ao *Find_ CC*.
- No *Find_ CC*, o **for** das linhas 7-16 percorre os adjacentes do vértice. Assim, durante todo o algoritmo, isso resulta em $O(E)$ execuções.

```
    input  : Grafo G
 1  begin
 2      for u ∈ V do
 3          cor[u] ← branco
 4      end
 5      tempo ← 0
 6      l ← 0
 7      S ← ∅
 8      for u ∈ V do
 9          if cor[u] = branco then
10              Find_CC(G, u)
11          end
12      end
13  end
```

Algorithm 1: *Main*

```
    input  : Grafo G, vértice u
 1  begin
 2      cor[u] ← cinza
 3      tempo ← tempo + 1
 4      d[u] ← tempo
 5      menor[u] ← tempo
 6      S.push(u)
 7      for v ∈ Adj[u] do
 8          if cor[v] = branco then
 9              Find_CC(G, v)
10              menor[u] ← min(menor[u], menor[v])
11          else
12              if cor[v] = cinza then
13                  menor[u] ← min(menor[u], menor[v])
14              end
15          end
16      end
17      cor[u] ← preto
18      if menor[u] = d[u] then
19          l ← l + 1
20          v ← S.pop()
21          while u ≠ v do
22              comp[v] ← l
23              v ← S.pop()
24          end
25          comp[v] ← l
26      end
27  end
```

Algorithm 2: *Find_CC*

- O **while** das linhas 21-23 do *Find_ CC* é executado removendo vértices da pilha S. Note que cada vértice é inserido apenas uma vez na pilha (linha 6 do *Find_ CC*). Portanto, em todo o algoritmo, esse **while** é executado $O(V)$ vezes.
- O restante das operações em ambas as funções requer tempo $O(1)$.

A análise agregada anterior nos permite concluir que a complexidade do algoritmo é $O(V+E)$.

Para provar que o algoritmo é correto, notemos primeiro o seguinte fato:

"Na linha 18 de Find_ CC, $menor[u] = d[u]$ se, e somente se, u for o primeiro vértice descoberto da sua componente fortemente conexa."

Prova.

$\Rightarrow$ Consideremos o momento em que x é descoberto e suponha que $r \neq x$ foi o primeiro vértice descoberto da componente fortemente conexa C que contém x.

Como x e r estão na mesma componente, existe um caminho formado por vértices de C de x até r. Seja z o primeiro vértice desse caminho descoberto antes de x e y o vértice que o precede.

Temos que z não pode ser branco, pois foi descoberto antes de x. Também não pode ser preto, já que, por estarem na mesma componente, existe um caminho de z até x que passa apenas por vértices de C. Assim, a única opção é z ser cinza, com $d[z] < d[x]$.

Como y não foi descoberto ainda, temos que y é branco no momento da descoberta de x, de modo que y é descendente de x (ou o próprio x). No momento em que y é descoberto, z ainda é cinza. Assim, ao analisar a aresta (y, z), temos que: $menor[y] \leq menor[z] \leq d[z] < d[x]$.

A linha 10 de *Find_ CC* garante que $menor[u] \leq menor[v]$ para qualquer descendente v de u. Portanto, na linha 18, temos que $menor[x] \leq menor[y] < d[x]$.

Assim, qualquer vértice x que não for o primeiro descoberto de C terá $menor[x] < d[x]$. Concluímos que, se $d[u] = menor[u]$, então u deve ser o primeiro vértice descoberto da sua componente fortemente conexa.

$\Leftarrow$ Seja u o primeiro vértice descoberto da sua componente. Em todo momento, para qualquer vértice v, as linhas 5, 4, 10 e 13 de *Find_ CC* garantem que $menor[v] \leq d[v]$. Logo, $menor[u] \leq d[u]$.

Suponha que a desigualdade seja estrita, isto é, $menor[u] < d[u]$. Como no início da descoberta de u, $menor[u] = d[u]$ (linhas 4 e 5), temos duas opções para que essa atualização para $menor[u] < d[u]$ tenha ocorrido nas linhas 10 ou 13 de *Find_ CC*:

- Se foi na linha 10, então existe um vértice v, descendente de u, tal que $menor[v] < d[u]$, o que implica que esse vértice tem uma aresta para um vértice z cinza, descoberto no momento $menor[v]$. Como z era cinza na análise de v, z é antecessor de v e também de u, já que foi descoberto antes de u ($d[z] = menor[v] < d[u]$). Desta forma, existe um caminho de z até u e de u até z.
- Se foi na linha 13, então existe um vértice z, adjacente de u, que estava cinza no momento em que u foi descoberto. Portanto, z é um antecessor de u, e existe um caminho de z até u e outro de u até z (pela aresta (u, z)).

Seja qual for o caso, existe um antecessor de u na mesma componente fortemente conexa, o que contradiz a suposição de que u fosse o primeiro descoberto.

Concluímos, portanto, que a nossa suposição é falsa, e $menor[u] = d[u]$.

Para finalizar a prova, usaremos indução em l, para demonstrar que as componentes fortemente conexas construídas estão corretas.

- *Caso base.* Para $l = 0$, é trivialmente verdade, pois o conjunto de componentes construídas é vazio.

- *Passo.* Para $l > 0$, seja C a componente construída no momento l. Note que essa componente é definida pelas linhas 18-26 do *Find_ CC*.

  A componente começa a ser construída pelo vértice u, tal que $menor[u] = d[u]$, e, pelo fato anterior, sabemos que u é o primeiro vértice da componente descoberto pelo algoritmo. Portanto, todos os outros vértices da componente são descendentes de u e foram empilhados em S acima dele.

  Pela hipótese de indução, nenhum dos vértices de C foi incluído nas componentes construídas anteriormente. Assim, todos os vértices de C estão na pilha nesse momento e serão adicionados à componente.

  Resta apenas verificar que não são adicionados vértices que não pertencem a C. Suponha, para fins de contradição, que um vértice v, que não pertence a C, será adicionado à componente. Isso significaria que v é descendente de u. No entanto, a componente de v não pode ter como primeiro vértice um antecessor x de u, pois, nesse caso, haveria um caminho de x até u e de u até x (passando por v), o que implicaria que tanto x quanto u deveriam estar na mesma componente, contradizendo o fato de que u foi o primeiro descoberto.

  Logo, v teria que estar em uma componente cujo primeiro vértice descoberto foi também um descendente de u, mas tal vértice já teria sido desempilhado anteriormente e, pela hipótese de indução, sua componente já estaria construída corretamente.

  Concluímos, então, que tal vértice v não existe e que em C somente são adicionados os vértices que pertencem à componente.

(b) Projete um algoritmo eficiente que determine se um grafo é euleriano ou não (ver definição na questão anterior). Em caso de ser euleriano, imprima um passeio fechado que passe exatamente uma vez por cada aresta do grafo. Analise a complexidade do seu algoritmo e prove a correção.

Resposta. Um grafo é euleriano se, e somente se, for conexo e todo vértice tiver grau par (um dos sentidos desta afirmação está provado na questão anterior, enquanto o outro foi discutido em sala de aula).

Assim, para saber se o grafo é euleriano, basta executar um *DFS* que conte as componentes conexas (como visto em aula), com complexidade $O(V + E)$. Se o grafo estiver representado por uma lista de adjacências, e cada lista tiver associado um tamanho, é possível verificar em $O(V)$ se todos os vértices têm grau par.

Se o grafo não for euleriano, não há mais o que fazer. No caso de ser euleriano, a prova da questão **1-a** sugere um algoritmo para encontrar o passeio fechado que passa exatamente uma vez por cada aresta:

```
   input  : Grafo euleriano G
1  begin
2  |  P ← ∅ G possui três ou mais vértices Encontrar um ciclo C
3  |  G' ← G − E[C]
4  |  P ← C
5  |  for cada componente H de G' do
6  |  |  P ← P ∪ Passeio_Euleriano(H)
7  |  end
8  |  return P
9  end
```

Algorithm 3: *Passeio_Euleriano*

A união do passeio sendo construído P com o retorno de $Passeio\_Euleriano(H)$ da componente deve ser feita através de um vértice em comum entre eles (como descrito no passo da prova em **1-a**). A correção do algoritmo segue também da prova dada em **1-a**.

Encontrar um ciclo tem complexidade $O(V + E)$ na quantidade de vértices da componente, dado que este é detectado na primeira aresta de retorno de um *DFS*. Obter G' pode ser feito em tempo $O(V + E)$, e, para cada componente H, a união com P pode ser realizada em

$O(E[H])$ (para isso, basta representar os passeios como listas ligadas e garantir que, ao obter as componentes, elas referenciem um vértice em comum com C, para isso as componentes podem ser obtidas iniciando os *DFSs* por vértices de C).

Se $T(n, m)$ descreve o tempo do algoritmo em termos do número de vértices (n) e arestas (m) de G, e G' possui k componentes, cada uma com n_i vértices e m_i arestas ($1 \leq i \leq k$), então:

$$T(n,m) = \begin{cases} O(1), \text{ se } n < 3 \\ \sum_{i=1}^{k} (T(n_i, m_i) + O(m_i)) + O(m+n), \text{ se } n \geq 3 \end{cases}$$

Como $m > \sum_{i=1}^{k} m_i$ e, pelo fato de o grafo e suas componentes serem conexos, $m \geq n$ e $m_i \geq n_i$ para todo i, temos:

$$T(n,m) \leq T(m) = \begin{cases} O(1), \text{ se } m < 3 \\ \sum_{i=1}^{k} T(m_i) + O(m), \text{ se } m \geq 3 \end{cases}$$

Essa recorrência satisfaz $T(m) = O(m^2)$. Portanto, o custo do algoritmo proposto para encontrar um passeio euleriano é $O(E^2)$.

Juntando com o algoritmo para detectar se o grafo é euleriano, temos uma complexidade total de $O(V + E^2)$.

(c) O diâmetro de um grafo é a maior distância entre qualquer par de vértices ($max_{u,v \in V} dist(u, v)$). Proponha um algoritmo eficiente para encontrar o diâmetro em de um grafo geral e outro para o caso do grafo ser uma árvore. Analise a complexidade e a correção dos seus algoritmos.

Resposta. Para calcular o diâmetro em grafos gerais, basta encontrar, para cada vértice, o vértice mais distante e selecionar o par com maior distância.

Dado um vértice u, podemos encontrar o vértice mais distante executando uma busca em largura (*BFS*), que tem complexidade $O(V + E)$. Logo, fazer essa operação para cada vértice nos dá um algoritmo com complexidade $O(V^2 + VE)$.

Se o grafo for uma árvore, é possível projetar uma solução mais eficiente. O primeiro passo é provar que, em uma árvore T, dado um vértice u qualquer, se um vértice v é o vértice mais distante de u, então v é um extremo de um caminho de comprimento máximo de T.

Iremos fazer a prova por contradição. Suponha que um caminho de comprimento máximo em T tem extremos x e y, e que v não é extremo de nenhum caminho de comprimento máximo. Temos duas possibilidades:

- Os caminhos de x até y e de u até v se intersectam. Neste caso, seja z o vértice mais perto de u na interseção dos caminhos. Sem perda de generalidade, considere que z não está mais longe de x que de y:
  - Se $d(z, v) \geq d(z, y)$, temos que, se z está no caminho entre x e v: $d(x, v) = d(x, z) + d(z, y) \geq d(x, y)$; senão, z está no caminho entre y e v: $d(y, v) = d(y, z) + d(z, v) \geq 2d(y, z) \geq d(y, v)$. Em qualquer dos casos, v seria um extremo de um caminho máximo.
  - Caso contrário, temos que $d(u, v) = d(u, z) + d(z, v) < d(u, z) + d(z, y) = d(u, y)$, o que implicaria que y está mais distante de u que v.
- Os caminhos de x até y e de u até v não se intersectam. Como T é conexo, deve haver um caminho conectando u e x. Seja z o último vértice do caminho entre x e y que também está no caminho entre u e x. Seja w o primeiro vértice do caminho entre z e u que está no caminho entre u e v. Sem perda de generalidade, suponha que $d(x, z) \leq d(y, z)$.
  Como $d(v, y) = d(v, w) + d(w, z) + d(z, y)$, temos que $d(v, w) + d(w, z) < d(x, z)$; caso contrário, $d(v, y) \geq d(x, y)$, e v seria extremo de um caminho máximo. Isso implica que:

  $$d(u, v) = d(u, w) + d(w, v) < d(u, w) + d(z, x) = d(u, x).$$

  Portanto, x seria um vértice mais distante de u que v.

Seja qual for o caso, chegamos a uma contradição, o que prova que a suposição inicial é falsa. Logo, o vértice v mais distante de um vértice qualquer u é um extremo de um caminho mais longo em T.

Dessa forma, basta executar um *BFS* a partir de um vértice u qualquer para encontrar o extremo de um caminho mais distante, v. A partir desse vértice, executamos um segundo *BFS*, pois o outro extremo de um caminho mais longo será um vértice mais distante de v. Assim, conseguimos encontrar o diâmetro de uma árvore com apenas dois *BFSs*, resultando em uma complexidade de $O(V + E) = O(V)$, já que em árvores $|E| = |V| - 1$.

(d) Considere o seguinte problema: é dada uma árvore T com n vértices e são feitas m consultas para saber a distância entre pares de vértices. Proponha um pré-processamento em $O(n)$, que permita calcular a distância entre cada par de vértices u, v em $O(dist(u, v))$. Justifique a complexidade e mostre a correção da sua solução.

Resposta. Suponha que enraizamos a árvore em um vértice arbitrário r. Então, dados dois vértices quaisquer u e v, o caminho entre eles em T pode ser obtido juntando o caminho de u até r com o caminho de r até v, eliminando todos os vértices repetidos. Note que os vértices repetidos nesses caminhos são todos os que são ancestrais comuns de u e v na árvore enraizada em r. Se z for o ancestral comum mais longe da raiz, temos que:

$$d(u, v) = d(u, r) + d(r, v) - 2 \cdot d(z, r).$$

Note que, eventualmente z pode ser o próprio u ou v.

Se executarmos um *BFS* a partir de r, o vetor d conterá as distâncias até a raiz e o vetor π os pais de cada vértice. Desta forma, dados dois vértices u e v, só precisamos encontrar o ancestral comum z mais distante da raiz, pois:

$$d(u, v) = d[u] + d[v] - 2 \cdot d[z].$$

Como z é ancestral de u e de v, poderíamos percorrer simultaneamente os pais de u e v, iterando pelos pais dos pais, até chegarmos ao primeiro vértice em comum. Contudo, esse processo só funciona se $d[u] = d[v]$; caso contrário, um deles chegaria em z antes que o outro e o ancestral comum não seria detectado.

Para evitar essa situação, basta começar iterando pelo vértice mais distante da raiz até que ambos tenham a mesma distância à raiz. A partir desse ponto, podemos iterar pelos ancestrais dos dois vértices juntos até encontrarmos o primeiro vértice comum:

```
   input    : Par de vértices de interesse u, v
 1 begin
 2   if d[u] > d[v] then
 3     Antecssor_Comum(v, u)
 4   else
 5     p_u ← u
 6     p_v ← v
 7     while d[p_v] > d[p_u] do
 8       p_v ← π[p_v]
 9     end
10     while p_v ≠ p_u do
11       p_u ← π[p_u]
12       p_v ← π[p_v]
13     end
14     return p_u
15   end
16 end
```

Algorithm 4: *Antecssor_Comum.*

Como o antecessor comum pertence ao caminho entre u e v, a complexidade desse algoritmo é $O(dist(u, v))$. Já o preprocessamento só requer executar um *BFS*, que tem complexidade $O(V + E) = O(V) = O(n)$, pois, em uma árvore, $|E| = |V| - 1$.