Algoritmos em Grafos

Wladimir Araújo Tavares [1]

[1]Universidade Federal do Ceará - Campus de Quixadá

26 de abril de 2017

Grafos

- Uma estrutura matemática formada por um conjunto de n vértices e um conjunto de m arestas.
- As arestas conectam pares de vértices. As arestas podem ser direcionadas ou não.
- Muitos problemas podem ser formulados e resolvidos em termos de grafos:
  - Caminho mínimo
  - Fluxos em redes
  - Emparelhamento
  - 2-SAT
  - Coloração de Grafos

Matriz de Adjacência

- Checar se dois vértices estão conectados: O(1).
- Listar todos os vizinhos de um vértice v: O(n)
- Complexidade de memória: $O(n^2)$
- Mais adequado para grafos densos, ou seja, $E \approx \frac{V(V-1)}{2}$

Lista de Adjacência

- Checar se dois vértices estão diretamente conectado: O(d(v))
- Lista a vizinhança de um vértice: O(d(v))
- Complexidade de memória O(n+m)
- Mais adequado para grafos pouco densos.

Implementando Lista de Adjacência

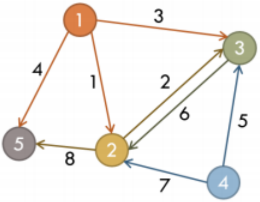
- Solução 1: Usar Listas encadeadas
  - Muito esforço de memória e tempo.
  - Usar ponteiros é ruim
- Solução 2: Usar um vetor de vector
  - Fácil de usar e sem problemas de alocação de memória
  - Acesso mais lento
- Solução 3: Usando vetores
  - Assumindo que o número total de arestas é conhecido
  - Rápido e eficiente em memória.

Usando vetores

- Um vetor de arestas E de tamanho m e um vetor LE de tamanho n.
  - E[k].to indica a extremidade final da aresta k.
  - E[k].nextID indica a posição da próxima aresta com extremidade inicial igual da aresta k
- Inicialize LE[i] = -1 para todo i.
- Inserir uma nova aresta u para v com ID k
  - E[k].to = v
  - E[k].nextID = LE[u]
  - LE[u] = k
- Iterar sobre todas as arestas começada por u:

```
for(ID = LE[u]; ID != -1; ID = E[ID].nextID)
```

Lista de Adjacência usando vetores



ID	To	Next Edge ID
1	2	-
2	3	-
3	3	1
4	5	3
5	3	-
6	2	-
7	2	5
8	5	2

From	1	2	3	4	5
Last Edge ID	4	8	6	7	-

Busca em largura

Algorithm 1 Algoritmo BFS

```
function BFS(G, s)
    for cada v ∈ V(G) do
        cor[v] ← BRANCO
        π[v] ← NULL
        d[v] ← ∞
    cor[s] ← CINZA
    d[s] ← 0
    Q ← ∅
    ENQUEUE(s)
    while Q ≠ ∅ do
        u ← DEQUEUE(Q)
        for cada v ∈ Adj[u] do
            if cor[v] = BRANCO then
                cor[v] ← CINZA
                d[v] ← d[u] + 1
                π[v] ← u
                ENQUEUE(Q, v)
        cor[u] ← PRETO
```

Bons e maus sujeitos

1 Existem dois tipos de lutadores profissionais: "bons sujeitos"e "maus sujeitos". Entre qualquer par de lutadores profissionais pode ou não haver rivalidade. Suponha que temos n lutadores e uma lista de m pares de lutadores para os quais existem rivalidades. Forneça um algoritmo de tempo $O(n+m)$ que determina se é possível designar alguns dos lutadores como bons sujeitos e os restantes como maus sujeitos, de tal forma que a rivalidade ocorra em cada caso entre um bom sujeito e um mau sujeito.

Bons e maus sujeitos

Algorithm 2 Algoritmo BOM_E_MAL

1: **function** BOM_E_MAL(G)
2: Execute a quantidade de BFS necessárias até que todos os vértices sejam visitados.
3: Atribua a todos os lutadores cuja distância é par para ser "bom sujeito"e os lutadores cuja distância é ímpar para ser "mal sujeito".
4: Verifique se a rivalidade em cada aresta acontece entre um "bom sujeito"e um "mal sujeito."

Diâmetro de uma árvore

2 O diâmetro de uma árvore $T = (V, E)$ é dado por $max_{u,v \in V} d(u, v)$, onde $d(u, v)$ é a distância mínima entre os vértices u e v, ou seja, o diâmetro é a maior das menores distâncias na árvore. Forneça um algoritmo eficiente para calcular o diâmetro de uma árvore e analise o tempo de execução de seu algoritmo.

Diâmetro de uma árvore

Algorithm 3 Algoritmo diâmetro de uma árvore

```
1: function DIAMETER(T)
2:     Execute uma BFS a partir de um vértice qualquer. Seja u o vértice mais distante encontrado pela BFS.
3:     Execute uma BFS a partir de u.
4:     A maior distância encontrada pela segunda BFS é o diâmetro da árvore T.
```

Centro de uma árvore

3 A excentricidade de um vértice v, denotado por $E(v)$, é a distância mínima de v até o vértice mais longe de v. O centro de um grafo são os vértices que tem o menor valor de excentricidade. Encontre o centro de uma árvore T.

Diâmetro de um grafo

2 O diâmetro de um grafo $G = (V, E)$ é dado por $max_{v \in V} E(v)$, onde $E(v)$ é a distância mínima entre o vértice v até o vértice mais longe de v, ou seja, o diâmetro é a maior das menores distâncias em um grafo. Forneça um algoritmo eficiente para calcular o diâmetro de um grafo e analise o tempo de execução de seu algoritmo.

Centro de um grafo

3 A excentricidade de um vértice v, denotado por $E(v)$, é a distância mínima de v até o vértice mais longe de v. O centro de um grafo são os vértices que tem o menor valor de excentricidade. Encontre o centro de um grafo G.

Busca em Profundidade

Algorithm 4 Algoritmo DFS

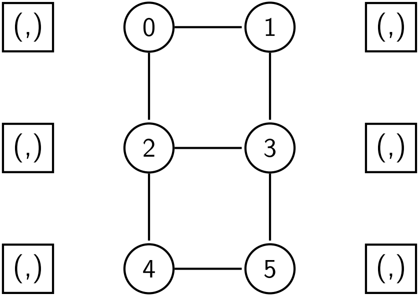
function DFS(G)
    for cada $v \in V(G)$ **do**
        $cor[v] \leftarrow BRANCO$
        $\pi[v] \leftarrow NULL$
    $tempo \leftarrow 0$
    for cada $v \in V(G)$ **do**
        if cor[v] = BRANCO **then**
            $DFS\_VISIT(v)$

Busca em Profundidade

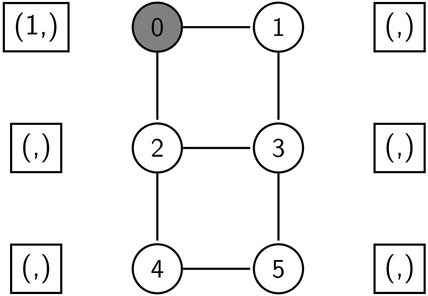
Algorithm 5 Algoritmo DFS_VIST

function DFS_VISIT(v)
    $cor[v] \leftarrow CINZA$
    $tempo \leftarrow tempo + 1$
    $chegada[v] \leftarrow tempo$
    for cada $u \in Adj[v]$ **do**
        if cor[u] = BRANCO **then**
            $\pi[u] \leftarrow v$
            $DFS\_VISIT(u)$
    $cor[u] \leftarrow PRETO$
    $tempo \leftarrow tempo + 1$
    $partida[u] \leftarrow tempo$

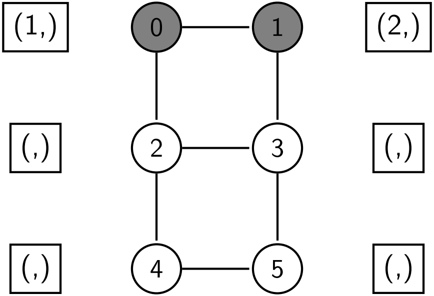
Busca em profundidade



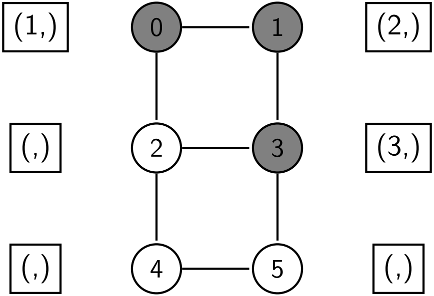
Busca em profundidade



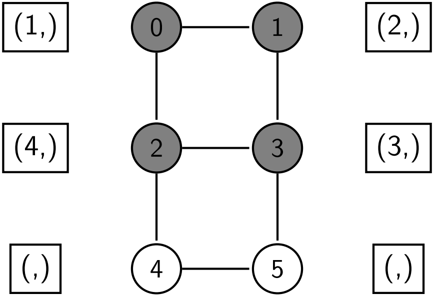
Busca em profundidade



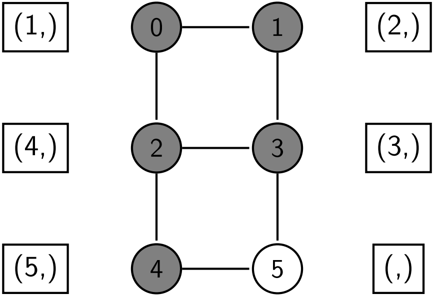
Busca em profundidade



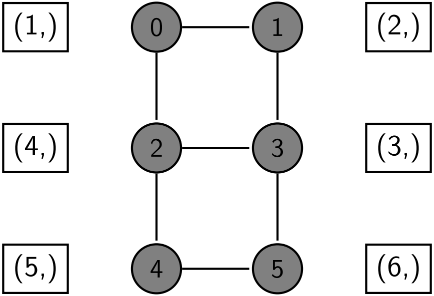
Busca em profundidade



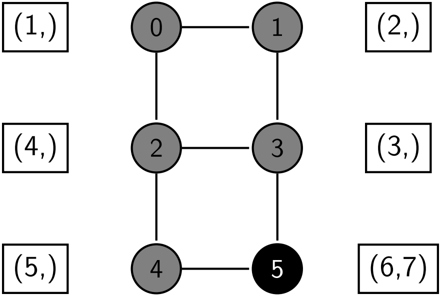
Busca em profundidade



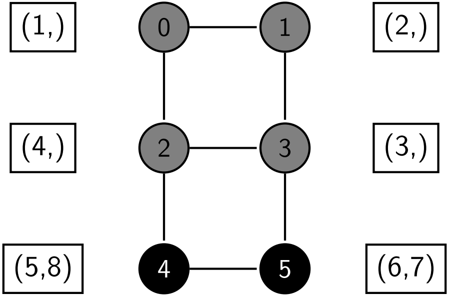
Busca em profundidade



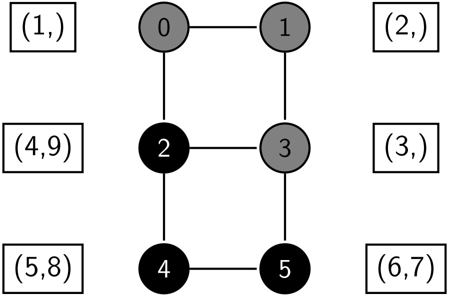
Busca em profundidade



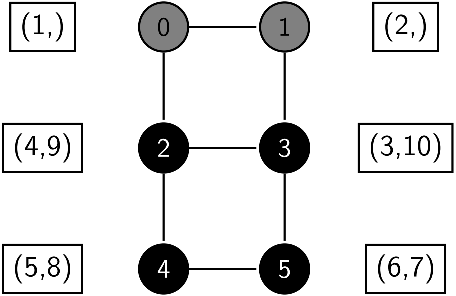
Busca em profundidade



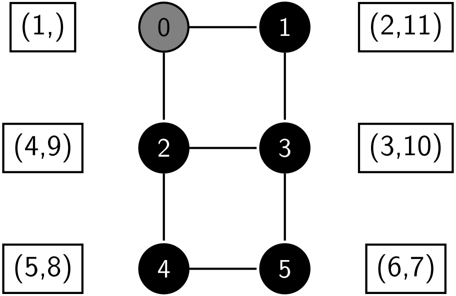
Busca em profundidade



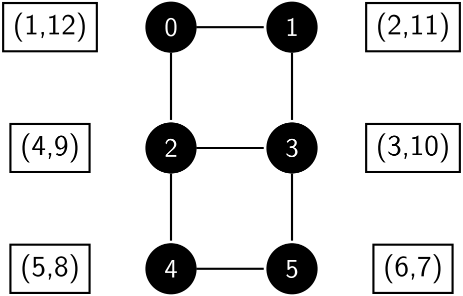
Busca em profundidade



Busca em profundidade



Busca em profundidade



Classificação de arestas em grafos não-direcionado

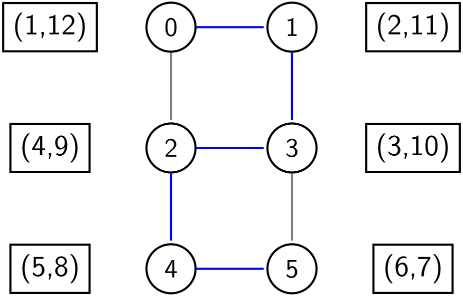
- A aresta (u, v) é uma **aresta da árvore** se v foi descoberto primeiro pela exploração da aresta (u, v).

$$chegada[u] < chegada[v] < partida[v] < partida[u] \tag{1}$$

- A aresta (u, v) é uma **aresta de retorno** se ela conecta um vértice u a um ancestral v em uma árvore da busca em profundidade.. Então,

$$chegada[v] < chegada[u] < partida[u] < partida[v] \tag{2}$$

Classificação de arestas em grafos não-direcionado



Classificação de arestas em grafos não-direcionado

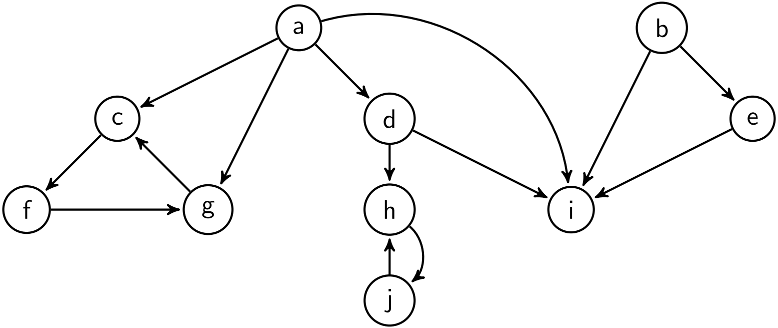
- A aresta (u, v) é uma **aresta da árvore** se v foi descoberto primeiro pela exploração da aresta (u, v).
- A aresta (u, v) é uma **aresta de retorno** se ela conecta um vértice u a um ancestral v em uma árvore da busca em profundidade.
- A aresta (u, v) é uma **aresta direta** se ela conecta um vértice u a um descendente v em uma árvore de busca em profundidade.
- A aresta (u, v) é uma **aresta cruzada** se ela conecta um vértice que não seja ancestral do outro.

Classificação de arestas em grafos direcionado

A classificação pode ser definida quando a aresta (u, v) é explorada primeiro, se a cor de v for

- BRANCO indica uma aresta de árvore.
- CINZA indica uma aresta de retorno.
- PRETO e $chegada[u] < chegada[v]$ indica uma aresta direta.
- PRETO e $chegada[u] > chegada[v]$ indica uma aresta cruzada.

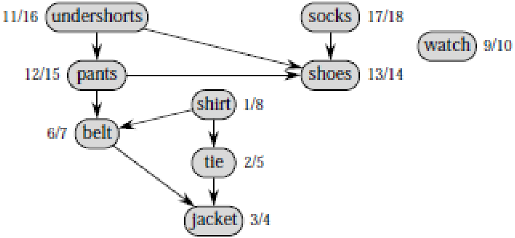
Classificação de arestas em grafos direcionado



Aplicações da DFS com tempo de chegada e partida:

- Ordenação topológica em um grafo direcionado acíclico.
- Detecção de componentes conexas em grafos não-direcionados.
- Determina se um grafo direcionado é fortemente conexo
- Detecção de componentes fortemente conexa em grafos direcionados.
- Detecção de ciclos em grafos direcionados e não-direcionados.
- Encontrar pontes e articulações.

Ordenação Topológica



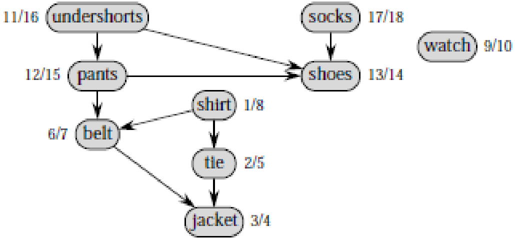
FiguraO professor Bumstead ordena topologicamente sua roupa ao se vestir. Cada aresta orientada (u, v) representa que a peça u deve ser vestida antes da peça v

Ordenação Topológica

Algorithm 6 Algoritmo Ordenação Topológica(G)

```
function TOPOLOGICAL_SORT(G)
    CHAMAR DFS(S) para calcular partida[v] para cada vértice v
    à medida que cada vértice é terminado, inserir o vértice à frente de uma lista ligada
    return lista ligada de vértices
```

Ordenação Topológica



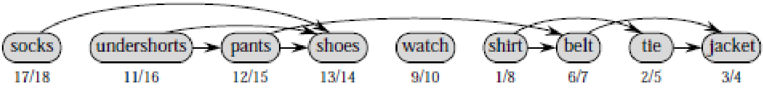


Figura O professor Bumstead ordena topologicamente sua roupa ao se vestir. Cada aresta orientada (u, v) representa que a peça u deve ser vestida antes da peça v

Componentes Conexas em grafos não direcionados

Algorithm 7 Algoritmo DFS_COMPONENTE

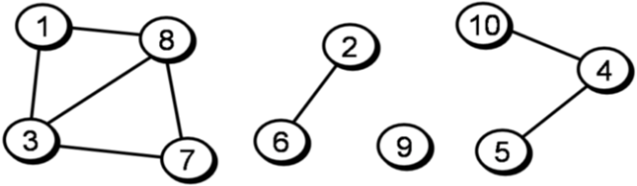
```
function DFS_COMPONENTE(G)
    for cada v ∈ V(G) do
        cor[v] ← BRANCO
        π[v] ← NULL
    tempo ← 0
    cont ← 0
    for cada v ∈ V(G) do
        if cor[v] = BRANCO then
            cont ← cont + 1
            DFS_VISIT(v, cont)
```

Componentes Conexas em grafos não direcionados

Algorithm 8 Algoritmo DFS_VIST

```
function DFS_VISIT(v, cont)
    cor[v] ← CINZA
    cc[v] ← cont
    tempo ← tempo + 1
    chegada[v] ← tempo
    for cada u ∈ Adj[v] do
        if cor[u] = BRANCO then
            π[u] ← v
            DFS_VISIT(u, cont)
    cor[u] ← PRETO
    tempo ← tempo + 1
    partida[u] ← tempo
```

Componentes Conexas em grafos não direcionados



FiguraO algoritmo encontra 4 componentes conexas: {1,3,7,8}

,{2,6},{9} e {4,5,10}

Grafo direcionado fortemente conexo

Um grafo direcionado é fortemente conexo se:

- $DFS\_VISIT(G, v)$ visita todos os vértices de G, então existe um caminho de v para todos outros vértices de G.
- $DFS\_VISIT(G^T, v)$ visita todos os vértices de G, então existe um caminho de todos os outros vértices de G para v.

Grafo direcionado fortemente conexo

Algorithm 9 STRONGLY_CONNECTED(G)

```
function STRONGLY_CONNECTED(G)
    for cada v ∈ V(G) do
        cor[v] ← BRANCO
        π[v] ← NULL
    CHAMAR DFS_VISIT(G, v)
    if ∃v cor[v] = BRANCO then
        return false
    Calcule G^T
    for cada v ∈ V(G) do
        cor[v] ← BRANCO
        π[v] ← NULL
    CHAMAR DFS_VISIT(G^T, v)
    if ∃v cor[v] = BRANCO then
        return false
    else
        return true
```

Componentes Fortemente Conexas em grafos direcionados

- Um componente fortemente conexa em um grafo direcionado em $G = (V, E)$ é um conjunto maximal de vértice $C \subseteq V$ tal que para todo par de vértice u e v em C, temos ao mesmo tempo $u \rightarrow v$ e $v \rightarrow u$.
- Se u e v são acessíveis um a partir do outro em G se somente se eles são acessíveis um a partir do outro em G^T.

Componentes Fortemente Conexas em grafos direcionados

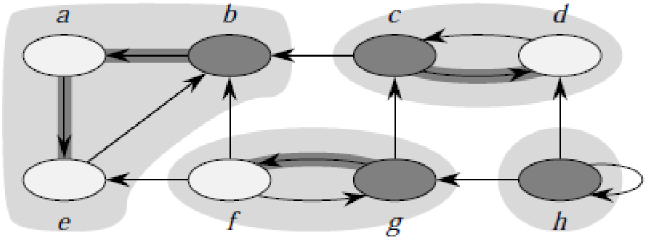


Figura As componentes fortemente conexas de G são mostradas como regiões sombreadas.

Componentes Fortemente Conexas em grafos direcionados

Algorithm 10 STRONGLY_CONNECTED_COMPONENTS(G)

1: **function** STRONGLY_CONNECTED_COMPONENTS(G)
2: CHAMAR DFS(G) para calcular *partida*[v] para cada vértice v em $O(V+E)$
3: Calcular G^T em $O(V+E)$
4: CHAMAR DFS(G^T) mas , no loop principal do DFS, considerar os vértices em ordem decrescente de *partida*[v]
5: Dar sáida aos vértices de cada árvore da floresta da busca primeiro na profundidade como uma componente fortemente conexa separadamente.

Componentes Fortemente Conexas em grafos direcionados

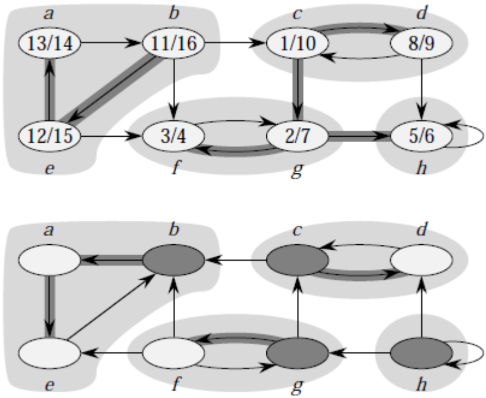


Figura As componentes fortemente conexas de G são mostradas como regiões sombreadas.

Algoritmo de Tarjan - SCC

Algorithm 11 Algoritmo TARJAN

```
function TARJAN(G)
    index ← 0
    for cada v ∈ V do
        v.index ← NULL
        v.onStack ← false
    for cada v ∈ V(G) do
        if v.index = NULL then
            strongconnect(v)
```

Algoritmo de Tarjan - SCC

Algorithm 12 Algoritmo strongconnect(v)

```
function strongconnect(v)
    v.index ← index
    v.lowlink ← index
    index ← index + 1
    push(S, v)
    v.onStack ← true
    for cada u ∈ Adj[v] do
        if u.index = NULL then
            strongconnect(u)
            v.lowlink ← min(v.lowlink, u.lowlink)
        else if u.onStack then
            v.lowlink ← min(v.lowlink, u.index)
    if v.lowlink = v.index then
        inicialize uma componente fortemente conexa
        repeat
            w ← pop(S)
            w.onStack ← false
            Adiciona w a componente fortemente conexa atual
        until w = v
        Imprima a componente fortemente conexa
```

Ponto de articulação

- Um vértice v em um grafo conexo não orientado $G = (V, E)$ é uma **articulação** se sua remoção torna o grafo desconexo.
- Uma aresta e em um grafo conexo não orientado $G = (V, E)$ é uma **ponte** se sua remoção torna o grafo desconexo

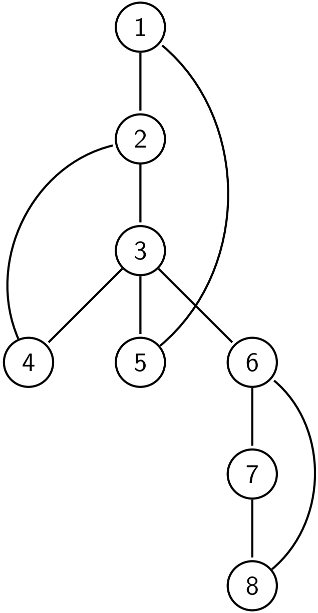


Figura Os vértices 3 e 6 são pontos de articulação e aresta $\{3,6\}$ é uma ponte.

Ponto de articulação

Lemma

Seja v a raiz de G_π, então v é ponto de articulação se somente se ele tem pelo menos dois filhos.

Demonstração.

Se v tem somente um filho, isso significa que qualquer vértice pode ser alcançado a partir desse filho. Logo, tirando v de G não tornaria desconexo. Se v tem mais de um filho, isso significa que não podemos alcançar todos os vértices a partir de qualquer um dos filhos. Nesse caso, devemos passar por v para ir para vértices em subárvores diferentes. Logo, retirar v torna o G desconexo. $\square$

Ponto de articulação

Lemma

Se v é um ponto de articulação se somente se v tem um filho s tal que não existe uma aresta de retorno de s ou de qualquer descendente de s para um ancestral próprio de v.

Demonstração.

se v tem um filho s tal que não existe uma aresta de retorno de s ou de qualquer descendente de s para um ancestral próprio de v. Então a retirada de v torna o grafo desconexo, os vértices da sub-árvore de s não alcançam os vértices ancestrais próprio de v. Caso contrário, se retiramos v, um caminho de s ou qualquer descendente de s para um ancestral próprio de v existe usando a aresta de retorno. □

Ponto de articulação

Algorithm 13 Algoritmo DFS_ARTICULACAO

function DFS_ARTICULACAO(G)
    for cada $v \in V(G)$ **do**
        $cor[v] \leftarrow BRANCO$
        $\pi[v] \leftarrow NULL$
        $articulacao[v] \leftarrow false$
    $altura \leftarrow 0$
    for cada $v \in V(G)$ **do**
        if cor[v] = BRANCO **then**
            $DFS\_VISIT(v, altura)$

Ponto de articulação

Algorithm 14 Algoritmo DFS_VISIT(v, altura)

```
function DFS_VISIT(v, altura)
    cor[v] ← CINZA
    altura ← altura + 1
    nivel[v] ← menor[v] ← altura
    filhos ← 0
    for cada u ∈ Adj[v] do
        if cor[u] = BRANCO then
            filhos ← filhos + 1
            π[u] ← v
            DFS_VISIT(u, altura)
            menor[v] ← min(menor[v], menor[u])
            if nivel[v] ≠ 1 ∧ menor[u] ≥ nivel[v] then
                articulacao[v] ← true
        if cor[u] = CINZA ∧ π[v] ≠ u then
            menor[v] ← min(menor[v], nivel[u])
    if nivel[v] = 1 ∧ filhos > 1 then
        articulacao[v] ← true
    cor[u] ← PRETO
```

Ponte

- Uma aresta de G é uma ponte se somente se ela não reside em qualquer ciclo simples de G.
- Uma aresta $e = \{u, v\}$ é uma ponte em G_Π se somente se $menor[v] = nivel[v]$.

Ponte

Algorithm 15 Algoritmo DFS_VISIT(v, altura)

```
function DFS_VISIT(v, altura)
    cor[v] ← CINZA
    altura ← altura + 1
    nivel[v] ← menor[v] ← altura
    filhos ← 0
    for cada u ∈ Adj[v] do
        if cor[u] = BRANCO then
            filhos ← filhos + 1
            π[u] ← v
            DFS_VISIT(u, altura)
            menor[v] ← min(menor[v], menor[u])
        if cor[u] = CINZA ∧ π[v] ≠ u then
            menor[v] ← min(menor[v], nivel[u])
    if menor[v] == nivel[v] ∧ π[v] ≠ NULL then
        (π[v], v) é uma ponte
    cor[u] ← PRETO
```

Coloração Gulosa

- Uma coloração de um grafo $G = (V, E)$ é um função $cor : V \rightarrow C$, onde C é um conjunto de cores, tal que para cada aresta $\{u, v\} \in E$, tem-se $cor(u) \neq cor(v)$

Coloração Gulosa

- Uma coloração de um grafo $G = (V, E)$ é um função $cor : V \to C$, onde C é um conjunto de cores, tal que para cada aresta $\{u, v\} \in E$, tem-se $cor(u) \neq cor(v)$
- Uma k-coloração de G é uma coloração que utiliza k cores.

Coloração Gulosa

- Uma coloração de um grafo $G = (V, E)$ é um função $cor : V \rightarrow C$, onde C é um conjunto de cores, tal que para cada aresta $\{u, v\} \in E$, tem-se $cor(u) \neq cor(v)$
- Uma k-coloração de G é uma coloração que utiliza k cores.
- O número cromático de um grafo G, denotado por $\chi(G)$, é o menor número de cores k tal que existe uma k-coloração de G.

Coloração Gulosa

- Uma coloração de um grafo $G = (V, E)$ é um função $cor : V \rightarrow C$, onde C é um conjunto de cores, tal que para cada aresta $\{u, v\} \in E$, tem-se $cor(u) \neq cor(v)$
- Uma k-coloração de G é uma coloração que utiliza k cores.
- O número cromático de um grafo G, denotado por $\chi(G)$, é o menor número de cores k tal que existe uma k-coloração de G.
- Uma coloração gulosa considera os vértices de um grafo em uma ordem específica $v_1, v_2, \dots, v_n$ e atribui a cada vértice a menor cor disponível que não está sendo usada por nenhum vizinho de v_i.

Coloração Gulosa

Algorithm 16 Algoritmo GREEDY_COLORING(G, order)

```
function GREEDY_COLORING(G, order)
    for v ∈ V do
        cor[v] ← 0
    max_color ← 0
    for i ← 1 até |V| do
        v ← order[i]
        for j ← 1 até max_color do
            disponivel[j] ← true
        for u ∈ Adj[v] do
            if cor[u] ≠ 0 then
                disponivel[cor[u]] ← false
        k ← 0
        for j ← 1 até max_color do
            if disponivel[j] then
                k ← j
        if k = 0 then
            max_color ← max_color + 1
            cor[v] ← max_color
        else
            cor[v] ← k
```

Exercícios

1 Forneça um algoritmo de tempo linear que tome como entrada um grafo acíclico orientado $G = (V, E)$ e dois vértice s e t, e devolva o número de caminhos de s para t em G. Por exemplo, existem exatamente quatro caminhos do vértice p para o vértice v: *pov*, *poryu*, *posryv* e *psryu*. (Seu algoritmo só precisa contar os caminhos, não listá-los.)

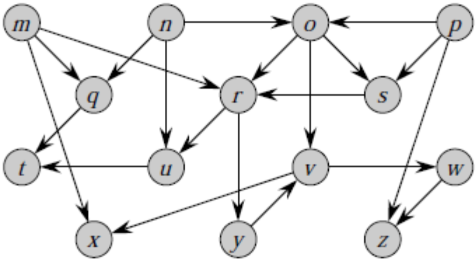


Figura Grafo Direcionado Acíclico

Exercícios

2 Um percurso de Euler de um grafo não direcionado conectado de Euler $G = (V, E)$ é um ciclo que percorre cada aresta de G exatamente uma vez, embora possa alcançar um vértice mais de uma vez.

1. Mostre que G tem um percurso de Euler se e somente se o grau(v) é par, para cada vértice $v \in V(G)$.
2. Descreva um algoritmo de tempo $O(E)$ para encontrar um percurso de Euler de G se existir um.

Exercícios

3 Um percurso de Euler de um grafo direcionado conectado de Euler $G = (V, E)$ é um ciclo que percorre cada aresta de G exatamente uma vez, embora possa alcançar um vértice mais de uma vez.

1. Mostre que G tem um percurso de Euler se e somente se o grau_entrada(v) for igual ao grau_saída[v], para cada vértice $v \in V(G)$.
2. Descreva um algoritmo de tempo $O(E)$ para encontrar um percurso de Euler de G se existir um.