Grafos - Algoritmos Baseados em Percurso

Tópicos Especiais em Algoritmos - Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília, Campus Taguatinga

Sumário

Sumário

Introdução

- Vimos anteriormente as duas estratégias básicas para realizar o percurso em um grafo.
- Estas estratégias podem ser aumentadas com algumas informações para resolução de diversos problemas.
- Veremos algora uma série de problemas que podem ser resolvidos através de pequenas modificações nas estratégias de percurso em grafos.

Sumário

Menor Distância

SSSP

- Dado um grafo **sem peso**, determine a distância de um vértice para todos os outros vértices.
- Neste caso, a distância de u e v, denotada por $d(u, v)$ é dada pela quantidade de arestas do menor caminho estes dois vértices.
- Extremamente aplicável ao problema de roteamento! Queremos minimizar o número de saltos.

Menor Distância

SSSP

- Entrada: um grafo G sem peso nas arestas e um vértice fonte s.
- Saída: a distância $d(s, v)$ para cada $v \in V$.

Menor Distância

- Para computar a menor distância, podemos recorrer à uma simples busca em largura.
- A cada nível examinado pela busca em largura, estamos descobrindo nós com uma distância uma unidade maior em relação aos nós examinados do nível anterior.

Busca em Largura

Algorithm 2: $\mathrm{BFS}(G, s)$

Input: G, s

Output: $v.d,\ \forall v \in V$

```
for all( u ∈ V )
    u.color ← white
    u.d ← ∞
Q ← ∅ // Fila
s.d ← 0
Q.PUSH(s)
s.color ← grey
while ¬Q.EMPTY() do
    u ← Q.POP()
    for all( (u,v) ∈ E )
        if( v.color = white )
            v.d = u.d + 1
            Q.PUSH(v)
            v.color ← grey
    u.color ← black
```

Busca em Largura

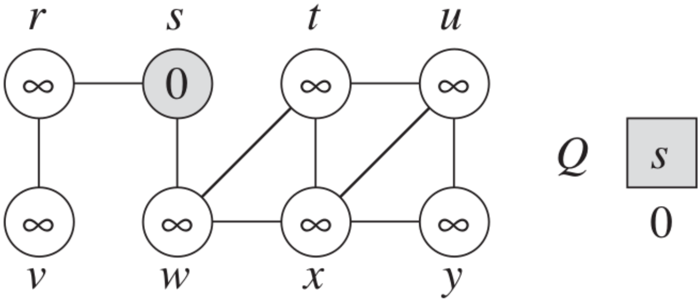


Figura: Busca em largura partindo do nó s.

Busca em Largura

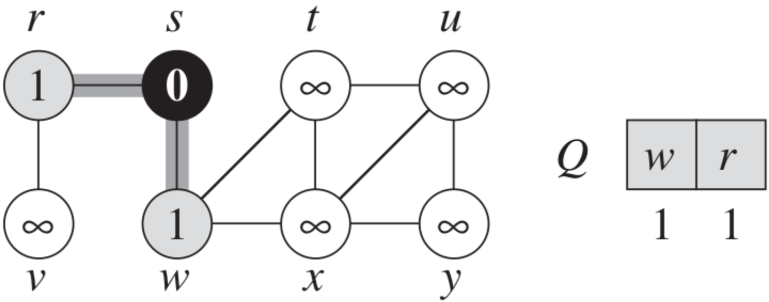


Figura: Busca em largura partindo do nó s.

Busca em Largura

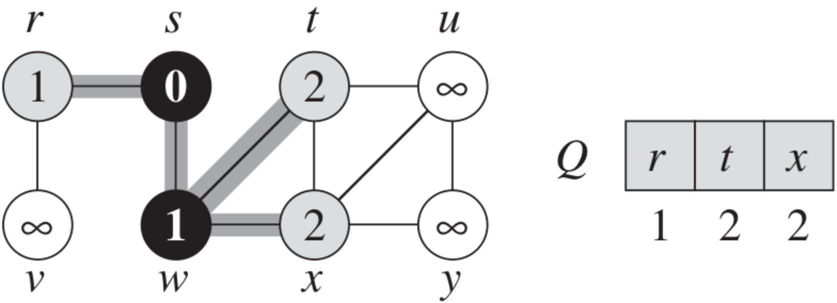


Figura: Busca em largura partindo do nó s.

Busca em Largura

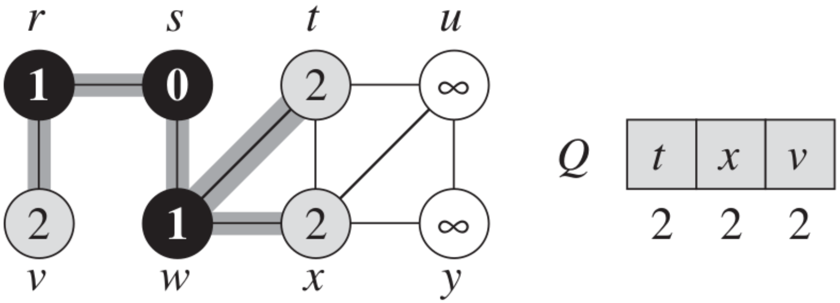


Figura: Busca em largura partindo do nó s.

Busca em Largura

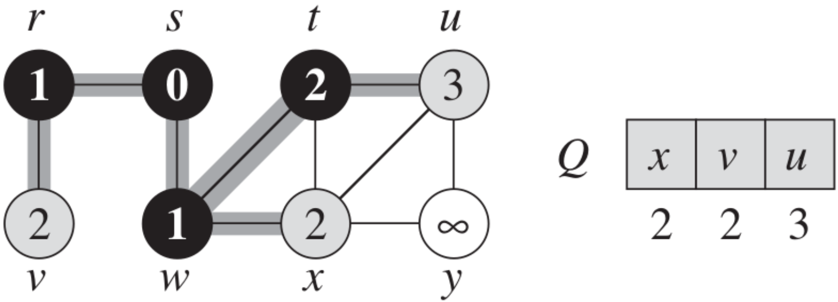


Figura: Busca em largura partindo do nó s.

Busca em Largura

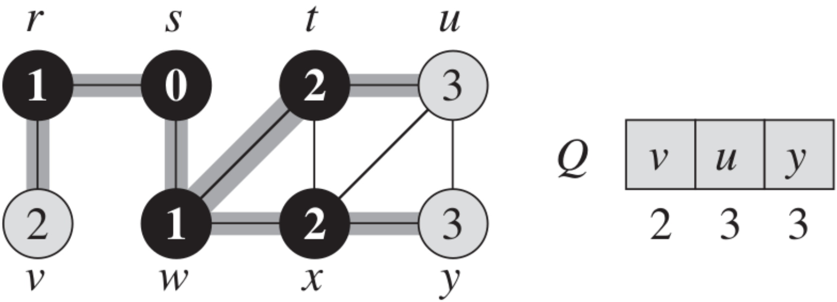


Figura: Busca em largura partindo do nó s.

Busca em Largura

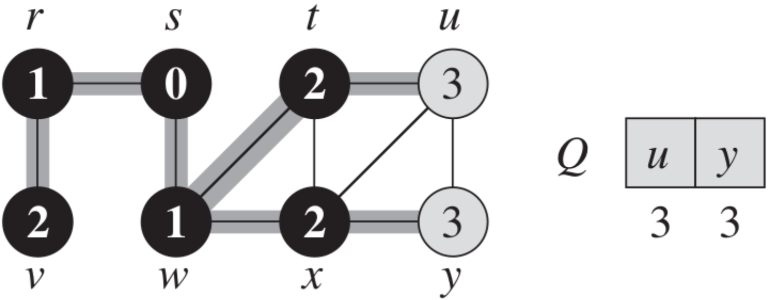


Figura: Busca em largura partindo do nó s.

Busca em Largura

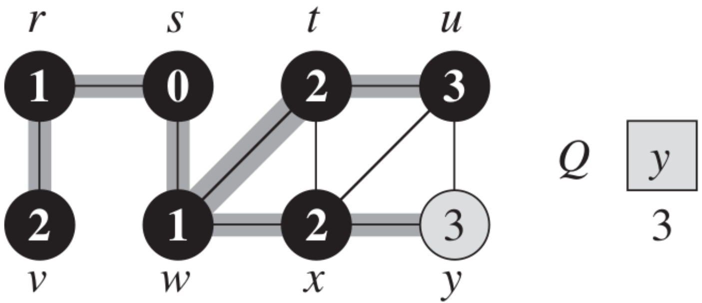


Figura: Busca em largura partindo do nó s.

Busca em Largura

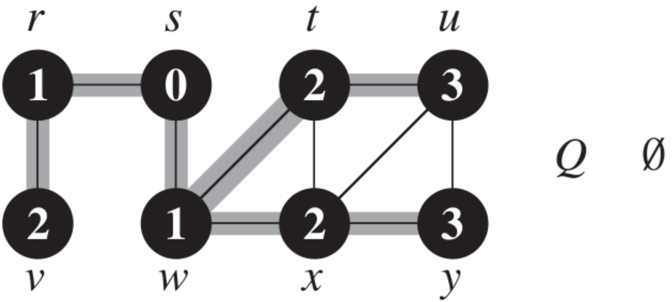


Figura: Busca em largura partindo do nó s.

Sumário

Problemas

Ordenação Topológica

- Dado um grafo **acíclico** dirigido (DAG), produzir uma ordenação topológica é interessante em algumas aplicações.
- A ordenação topológica produz uma ordenação dos vértices tal que, se existe uma aresta (u, v), então u deve estar antes de v no resultado da ordenação.
- Na prática, podemos usar DAGs para indicar precedência de eventos.
- Exemplo: verificar quais disciplinas são pré-requisito de outras.
- Exemplo: representar dependências entre pacotes de software.

Ordenação Topológica

TOPSORT

- Entrada: Um DAG G.
- Saída: uma ordenação topológica de G.

Ordenação Topológica

- Para produzir a ordenação topológica, podemos usar a busca em profundidade para marcar os tempos em que um nó é visitado pela primeira e segunda vez (tempo de início e tempo de término).
- Conforme a busca, adicionamos o nó com tempo de término mais recente no início de uma lista.
- Isso quer dizer que o nó com tempo mais recente obrigatoriamente vem antes do nó com tempo menos recente na ordenação.

Ordenação Topológica

Algorithm 3: $\textsc{topological-sort}(G)$

Input: G

Output: Lista L contendo a ordenação topológica de G.

```
time ← 0
L ← ∅
for all( v ∈ V )
    if( v.color = white )
        time ← DFS(G, v, time, L)
return L
```

Ordenação Topológica

Algorithm 4: $\mathrm{DFS}(G, v, time, L)$ que computa os tempos de início e fim de visitação a partir do nó v.

Input: $G, v, time, \&L$

Output: $v.d$ e $v.f$ para cada $v \in V$

$v.color \leftarrow$ **black**

$time \leftarrow time + 1$

$v.d \leftarrow time$

for all($(v, w) \in E$)

  if($w.color =$ **white**)

    $time \leftarrow \mathrm{DFS}(G, w, time, list)$

$time \leftarrow time + 1$

$v.f \leftarrow time$

$L.\textsc{prepend}(v)$

return $time$

Ordenação Topológica

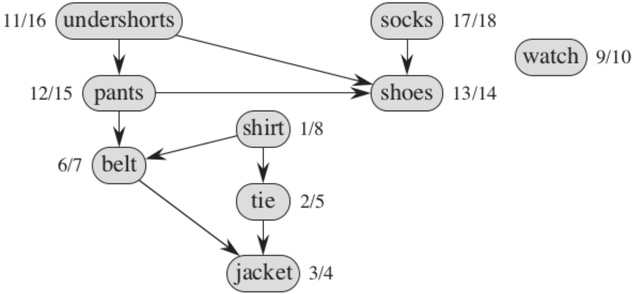


Figura: Ordenação topológica da sequência de vestimento.

Ordenação Topológica

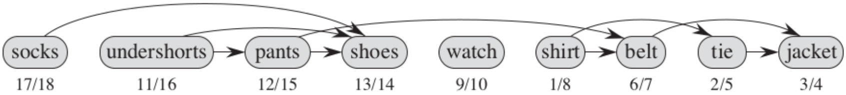


Figura: Ordenação topológica da sequência de vestimento.

Ordenação Topológica

- Na prática não é necessária a informação do tempo.
- Colocamos cada nó da lista em ordem de término de processamento (LIFO).

Ordenação Topológica

Algorithm 5: $\mathrm{DFS}(G, v, L)$.

Input: $G, v, \&L$

Output: A lista L atualizada a partir da componente fortemente conexa de v

$v.color \leftarrow$ **black**

for all($(v, w) \in E$)

  if($w.color =$ **white**)

    $\mathrm{DFS}(G, w, list)$

$L.\textsc{prepend}(v)$

Ordenação Topológica

Complexidade

- Precisamos apenas fazer uma busca em profundidade modificada.
- $\Theta(|V| + |E|)$

Sumário

4 CYCLE

Detecção de Ciclos

Detecção de Ciclos

- Detectar ciclos em grafos dirigidos é muito útil em algumas aplicações.
- Exemplo: detecção de deadlock pelo S.O.
- Exemplo: detecção de incompatibilidade de dependências.

Detecção de Ciclos

CYCLE

- Entrada: Um grafo dirigido.
- Saída: Sim se o grafo possui ciclos, não, caso contrário.

Detecção de Ciclos

- Estamos procurando por uma **back edge**, isto é, uma aresta que volta para um nó que ainda está sendo processado na busca em profundidade.
- Se no grafo existe uma **back edge**, então temos um ciclo.
- Basta adaptar a busca em profundidade.

Detecção de Ciclos

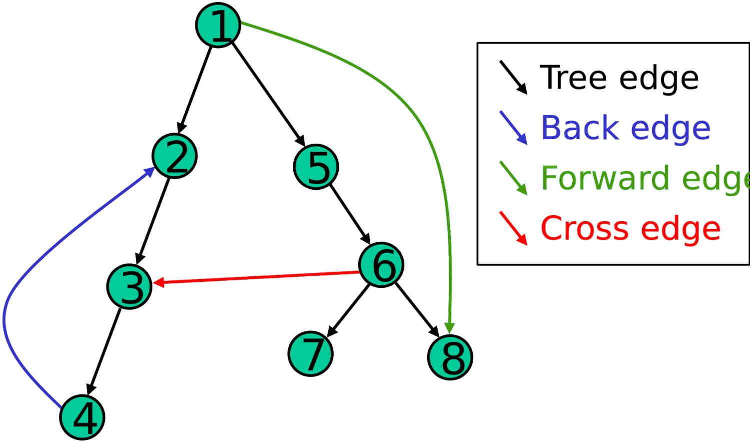


Figura: Tipos de aresta.

Detecção de Ciclos

Algorithm 6: CYCLE-DETECTION(G)

Input: G

Output: **true** se e somente se existe ciclo em G

```
for all( v ∈ V )
    v.color ← white
for all( v ∈ V )
    if( v.color = white )
        if( CYCLE-SEARCH(G, v) )
            return true
return false
```

Detecção de Ciclos

Algorithm 7: CYCLE-SEARCH(G, v)

Input: G, v

Output: Sim se e somente se a componente conexa de v possui ciclos

```
v.color ← grey
ans ← false
for all( (v, w) ∈ E )
    if( w.color = grey )                         // back edge
        ans ← true
    else if( w.color = white )
        if( CYCLE-SEARCH(G, w) )
            ans ← true
v.color = black
return ans
```

Detecção de Ciclos

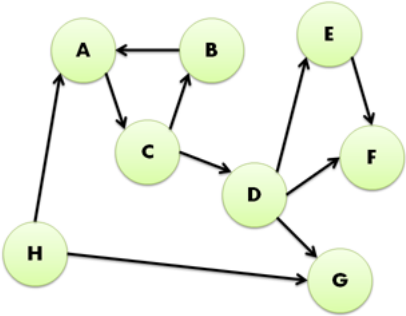


Figura: Detecção de Ciclos

Detecção de Ciclos

Complexidade

- $\Theta(|V| + |E|)$.

Sumário

Componentes Fortemente Conexas

- Todo grafo dirigido pode ser decomposto em várias componentes fortemente conexas.
- Um grafo é dito fortemente conexo se possui apenas 1 componente fortemente conexa.
- Como determinar as componentes fortemente conexas de um grafo?

Componentes Fortemente Conexas

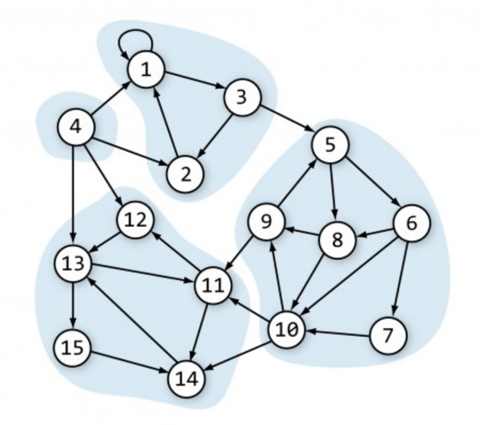


Figura: Componentes fortemente conexas.

Componentes Fortemente Conexas

SCC

- **Entrada:** um grafo dirigido G.
- **Saída:** suas componentes fortemente conexas.

Componentes Fortemente Conexas

- Uma componente é dita fortemente conexa se existe um caminho entre quaisquer dois pares de vértice nesta componente.
- Se a partir de um vértice v chegamos em u, temos que certificar que é possível chegar em v à partir de u.
- Podemos usar o conceito de **back edge**!

Componentes Fortemente Conexas

- Primeiramente, numeramos cada vértice com com a busca em profundidade de acordo com sua ordem de visitação ($v.index$).
- Se durante a busca em profundidade um nó u possui um caminho para um nó v que tem um número menor que u, então sabemos que também existe um caminho de u para v, portanto, eles estão na mesma componente conexa.
- Marcaremos cada nó com um número $v.low$, indicando o vértice com menor número de visitação que é alcançável por v.
- Todos os nós que possuem $v.low \leq v.index$ estão na mesma componente conexa de $v.low$.

Componentes Fortemente Conexas

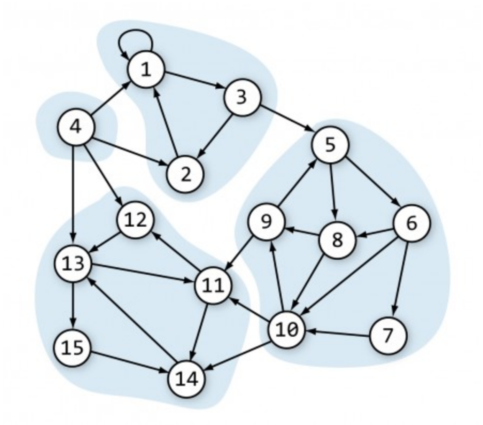


Figura: Componentes fortemente conexas.

Componentes Fortemente Conexas

Algorithm 8: FIND-STRONG-COMPONENTS(G)

Input: G

Output: Lista L de componentes fortemente conexas de G

```
L ← ∅ // Lista de SCCs
S ← ∅ // Pilha de nós em processamento
n ← 0 // índice de visitação
for all( v ∈ V )
    if( v.color = white )
        STRONG-COMPONENTS(G, v, L, S, n)
return L
```

Componentes Fortemente Conexas

Algorithm 9: STRONG-COMPONENTS(G, v, L, S, n)

Input: $G, v, \&L, \&S, \&n$

Output: Componentes Fortemente Conexas alcançáveis a partir de v

$v.index = n$
$v.low = n$
$n \leftarrow n + 1$
$v.color =$ **grey**
S.PUSH(v)
for all($(v, w) \in E$)
    if($w.color =$ **grey**)
        $v.low \leftarrow \min(v.low, w.index)$
    if($w.color =$ **white**)
        STRONG-COMPONENTS(G, w, L, S, n)
        $v.low \leftarrow \min(v.low, w.low)$
if($v.index = v.low$)
    L.APPEND(CREATE-NEW-COMPONENT(S, v))

Componentes Fortemente Conexas

Algorithm 10: $\textsc{create-new-component}(S, v)$

Input: $\&S, v$

Output: Componente Fortemente Conexa que inclui o nó v

```
L' ← ∅
repeat
    w ← S.POP()
    L'.APPEND(w)
    w.color ← black
until w ≠ v
return L'
```

Componentes Fortemente Conexas

Complexidade

- O custo é de uma DFS pelo grafo.
- $\Theta(|V| + |E|)$.

Sumário

Detecção de Pontos de Articulação

Definição (Ponto de Articulação)

- Tome um grafo não-direcionado G.
- Um ponto de articulação é um vértice cuja retirada desconecta G.
- Um grafo que não possui pontos de articulação é 2-conexo.

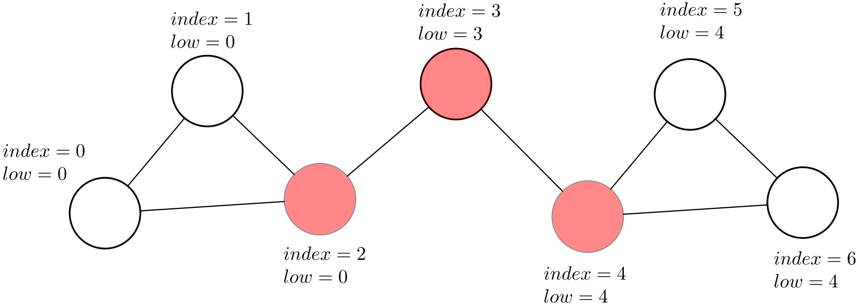
Detecção de Pontos de Articulação

- Em muitas aplicações, pontos de articulação são críticos e precisam ser detectados.
- Usando uma busca em profundidade adaptada é possível dizer se um grafo tem ou não tem pontos de articulação.
- Utilizando o mesmo conceito de $index$ e low do problema anterior é possível detectar pontos de articulação.

Detecção de Pontos de Articulação

- Suponha um nó u e um vizinho v de u. Caso $v.low \geq u.index$ significa que v não consegue alcançar nenhum nó com $index$ menor que o de u, isto é, os ancestrais de u considerando a DFS.
- Isto implica que u é um ponto de articulação!
- Exceção: nó raiz da DFS. Ele é uma articulação caso tenha 2 ou mais vizinhos considerando a **árvore de DFS**.

Pontos de Articulação



Pontos de Articulação

Algorithm 11: ARTICULATION-POINT(G, u, n)

Input: $G, u, \&n$

Output: Marcação de pontos de articulação partindo do nó u

```
u.index ← n
u.low ← n
n ← n + 1
u.color ← black
children ← 0
for all( (u, v) ∈ E )
    if( v.color = white ) // v não visitado
        v.π ← u
        if( u.π = NULL )
            children ← children + 1
        ARTICULATION-POINT(G, v, n)
        u.low ← min(u.low, v.low)
        if( v.low ≥ u.index )
            u.ap ← true
    else if( v ≠ u.π )
        /* Detecção de back-edge que não forme um ciclo
           imediato.                                    */
        u.low ← min(u.low, v.index)
if( u.π = NULL ) // Caso especial: raiz da DFS
    u.ap ← (children > 1)
```

Sumário

Detecção de Pontes

Definição (Ponte)

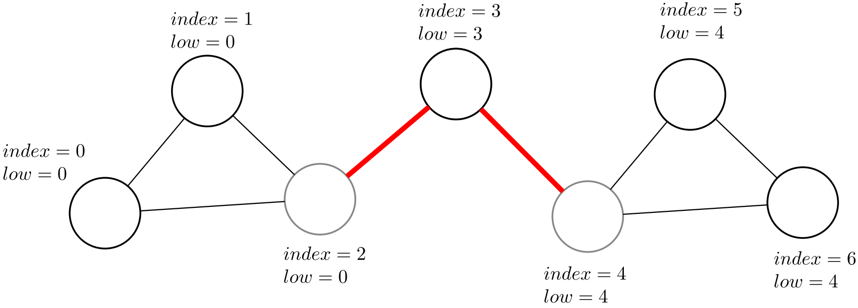
- Tome um grafo não-direcionado.
- O conceito de ponte é o análogo do conceito de ponto de articulação para arestas.
- Uma **ponte** é qualquer aresta cuja retirada desconecta o grafo.

Detecção de Pontes

`BRIDGE`

- Entrada: G.
- Saída: todas as arestas que formam uma ponte em G.

Detecção de Pontes



Detecção de Pontes

- Para detectar pontes podemos usar o algoritmo anterior de detecção de pontos de articulação com uma pequena modificação.
- A partir do momento que temos um nó u e um vizinho v de u e $v.low > u.index$, isso significa que v não consegue alcançar o nó u através de outro caminho, logo, a aresta (u, v), se retirada, aumenta o número de componentes conexas do grafo.

Detecção de Pontes

Algorithm 12: BRIDGE-DETECTION(G, u, n, B)

Input: $G, u, \&n, \&B$

Output: Lista de pontes B de G

$u.index \leftarrow n$
$u.low \leftarrow n$
$n \leftarrow n + 1$
$u.color \leftarrow$ **black**
for all($(u, v) \in E$)
    if($v.color =$ **white**) `// v não visitado`
        $v.\pi \leftarrow u$
        BRIDGE-DETECTION(G, v, n)
        $u.low \leftarrow \min(u.low, v.low)$
        if($v.low > u.index$)
            $B.$APPEND$((u, v))$
    else if($v \neq u.\pi$)
        `/* Detecção de back-edge que não forme um ciclo imediato. */`
        $u.low \leftarrow min(u.low, v.index)$

Sumário

8 Considerações

Considerações

- A partir de simples modificações nas estratégias de percurso, conseguimos resolver diferentes problemas de maneira eficiente.
- Quando trata-se de grafos, muitas das vezes só precisamos adaptar um algoritmo que já existe, não é necessário criar um do zero.