ACH2024

Aula 8 – Grafos: Busca em Profundidade (parte 2) e Busca em Largura

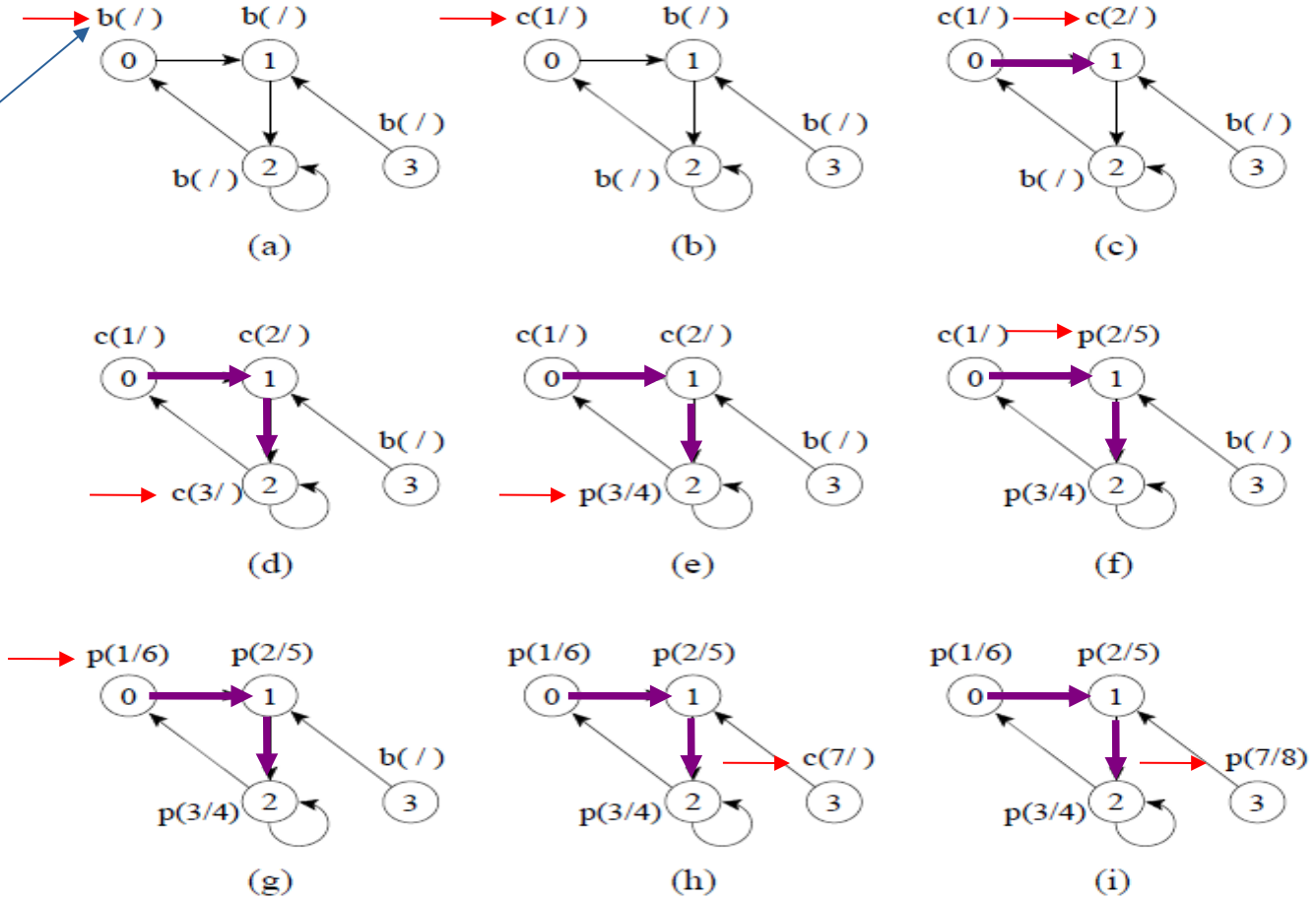
Profa. Ariane Machado Lima

Aula anterior

- Busca em Largura

Busca em Profundidade: Exemplo

Cada vértice tem:
cor(d[v],t[v]) e
antecessor (→)



Busca em Profundidade: Implementação

```
buscaProfundidade(grafo){
      Aloca vetores cor, tdesc, tterm, antecessor com tamanho grafo->nrVertices
       tempo ← 0;
       Para cada vertice v
              cor[v] ← branco;  tdesc[v] = tterm[v] = 0;  antecessor[v] ← -1;
       Para cada vertice v
              Se cor[v] = branco visitaBP(v, grafo, &tempo, cor, tdesc, tterm, antecessor);
}


visitaBP(v, grafo, tempo, cor, tdesc, tterm, antecessor){
       cor[v] ← cinza; tdesc[v] ← ++(*tempo);
       Para cada vertice u da lista de adjacência de v
              Se u é branco
                     antecessor[u] ← v;
                     visitaBP(u, grafo, &tempo, cor, tdesc, tterm, antecessor);
       tterm[v] ← ++(*tempo);
       cor[v] ← preto;
}
```

Função recursiva

O(?)

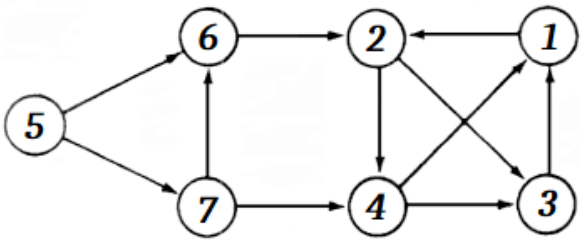
Busca em Profundidade: Implementação

```
buscaProfundidade(grafo){
      Aloca vetores cor, tdesc, tterm, antecessor com tamanho grafo->nrVertices
       tempo ← 0;
       Para cada vertice v
              cor[v] ← branco;  tdesc[v] = tterm[v] = 0;  antecessor[v] ← -1;
       Para cada vertice v
              Se cor[v] = branco visitaBP(v, grafo, &tempo, cor, tdesc, tterm, antecessor);
}


visitaBP(v, grafo, tempo, cor, tdesc, tterm, antecessor){
       cor[v] ← cinza; tdesc[v] ← ++(*tempo);
       Para cada vertice u da lista de adjacência de v
              Se u é branco
                     antecessor[u] ← v;
                     visitaBP(u, grafo, &tempo, cor, tdesc, tterm, antecessor);
       tterm[v] ← ++(*tempo);
       cor[v] ← preto;
}
```

Função recursiva

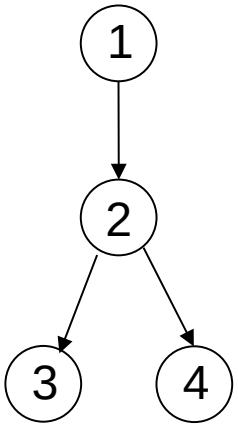
O(V + A)

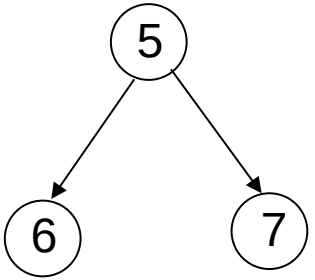
Qual a ordem em que os nós são descobertos durante a busca em profundidade?



Subgrafos de G que são árvores que descrevem a ordem em que os nós foram descobertos (tornados cinza)

Ordem de descoberta





Árvores de busca em profundidade do grafo

Investigações nessas árvores nos dão várias informações…
Para isso vamos ver alguns conceitos

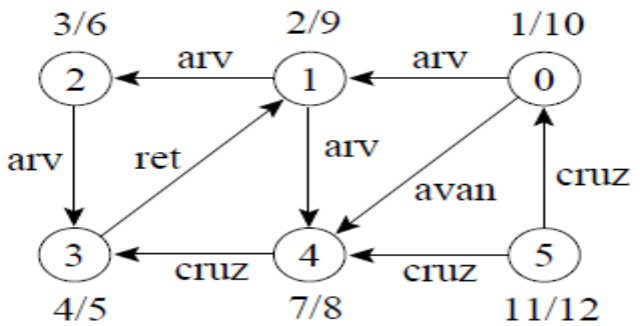
Classificação de Arestas

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).
3. **Arestas de avanço**: não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade.
4. **Arestas de cruzamento**: podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

Neste caso (vértices da mesma árvore de busca), cruza ramos desta árvore, já que conecta um vértice a um outro que **não é seu antecessor nem descendente**

Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta (u,v) pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de árvore.
  - Cinza indica uma aresta de retorno.
  - Preto indica uma aresta de avanço quando u é descoberto antes de v ou uma aresta de cruzamento caso contrário.



(u,v) é
avanço se tdesc[u] < tdesc[v]
cruzamento caso contrário

Algumas aplicações de busca em profundidade

Aplicações de busca em profundidade

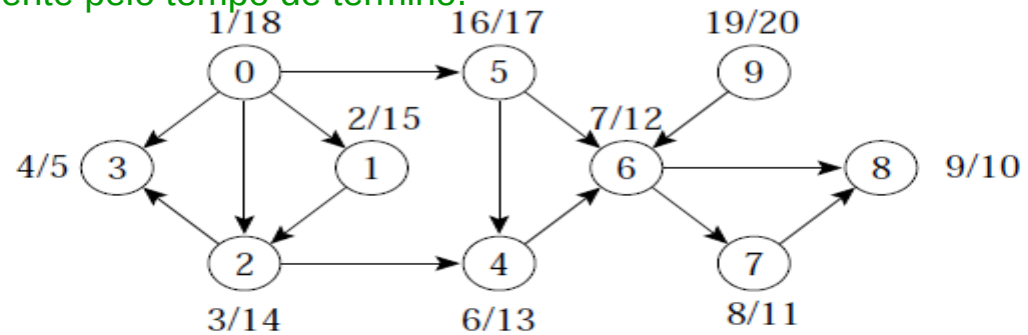
- Identificação se um grafo (direcionado ou não direcionado) é acíclico ou não
- Ordenação topológica (grafo direcionado)
- Identificação de se existe um caminho entre dois nós u e v em um grafo (direcionado ou não)
- Identificar os componentes conectados (em um grafo não direcionado) ou fraca/fortemente conectados (em um grafo direcionado)

Ordenação Topológica

Como poderíamos obter essa informação dos vértices “sem adjacentes” (ou adjacentes já processados), a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!

Como? Insere o vértice no início da lista quando ele se tornar preto… Desta forma, os vértices ficaram ordenados decrescentemente pelo tempo de término.



Aula de hoje

Aplicações de busca em profundidade

- Identificação se um grafo (direcionado ou não direcionado) é acíclico ou não
- Ordenação topológica (grafo direcionado)
- Identificação de se existe um caminho entre dois nós u e v em um grafo (direcionado ou não)
- Identificar os componentes conectados (em um grafo não direcionado) ou fraca/fortemente conectados (em um grafo direcionado)

Busca em Largura

EP 1 disponível

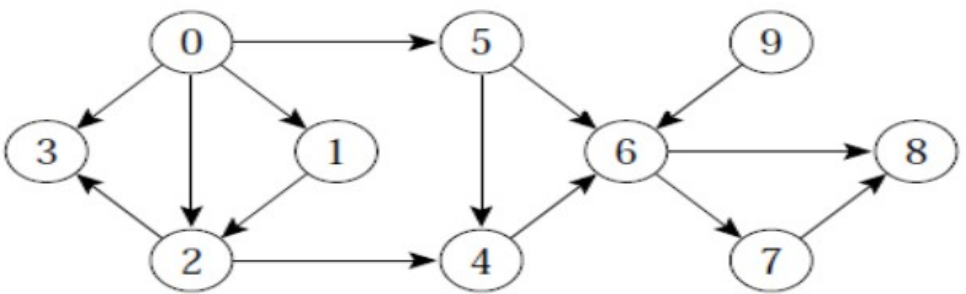
Exploraremos os conceitos de:

- Grafos não direcionados
- Serão exigidas as DUAS implementações (por matriz e lista de adjacência)
- Makefile
- Manipulação de grafos PRECISA usar a interface → pode-se incluir outras funções, ex:
  - obtemVerticeDestino

Aplicações de busca em profundidade

- Identificação se um grafo (direcionado ou não direcionado) é acíclico ou não
- Ordenação topológica (grafo direcionado)
- Identificação de se existe um caminho entre dois nós u e v em um grafo (direcionado ou não)
- Identificar os componentes conectados (em um grafo não direcionado) ou fraca/fortemente conectados (em um grafo direcionado)

Encontrar **um** caminho entre dois vértices u e v

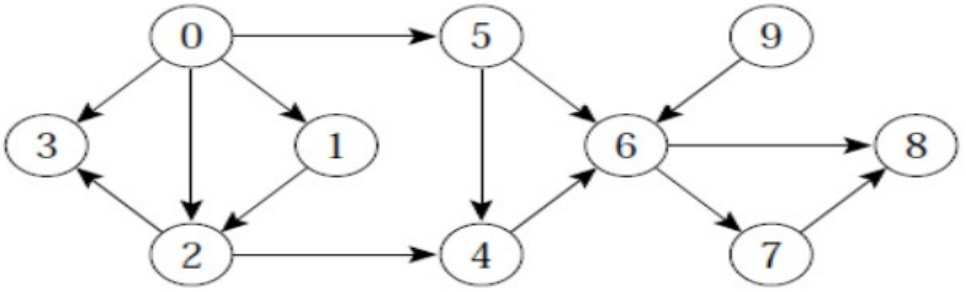


Encontrar **um** caminho entre dois vértices u e v

Busca em profundidade pode ser utilizado para encontrar (se existir) UM caminho entre dois vértices u e v

Partindo de u, o vértice v deve ser visitado (ou seja, v devendo pertencer à árvore de busca em profundidade com raiz em u)

Esse caminho é o mais curto?

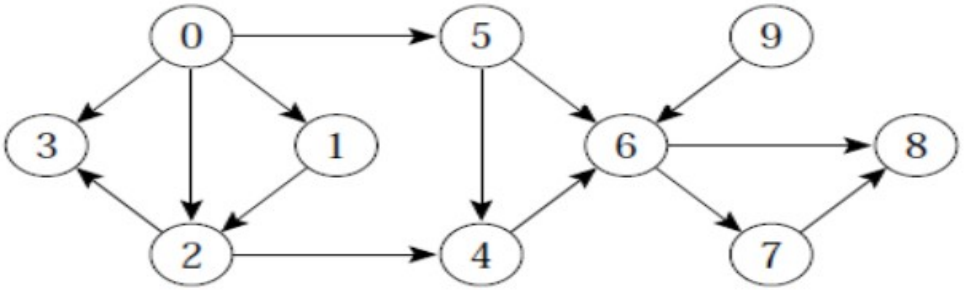


Encontrar **um** caminho entre dois vértices u e v

Busca em profundidade pode ser utilizado para encontrar (se existir) UM caminho entre dois vértices u e v

Partindo de u, o vértice v deve ser visitado (ou seja, v devendo pertencer à árvore de busca em profundidade com raiz em u)

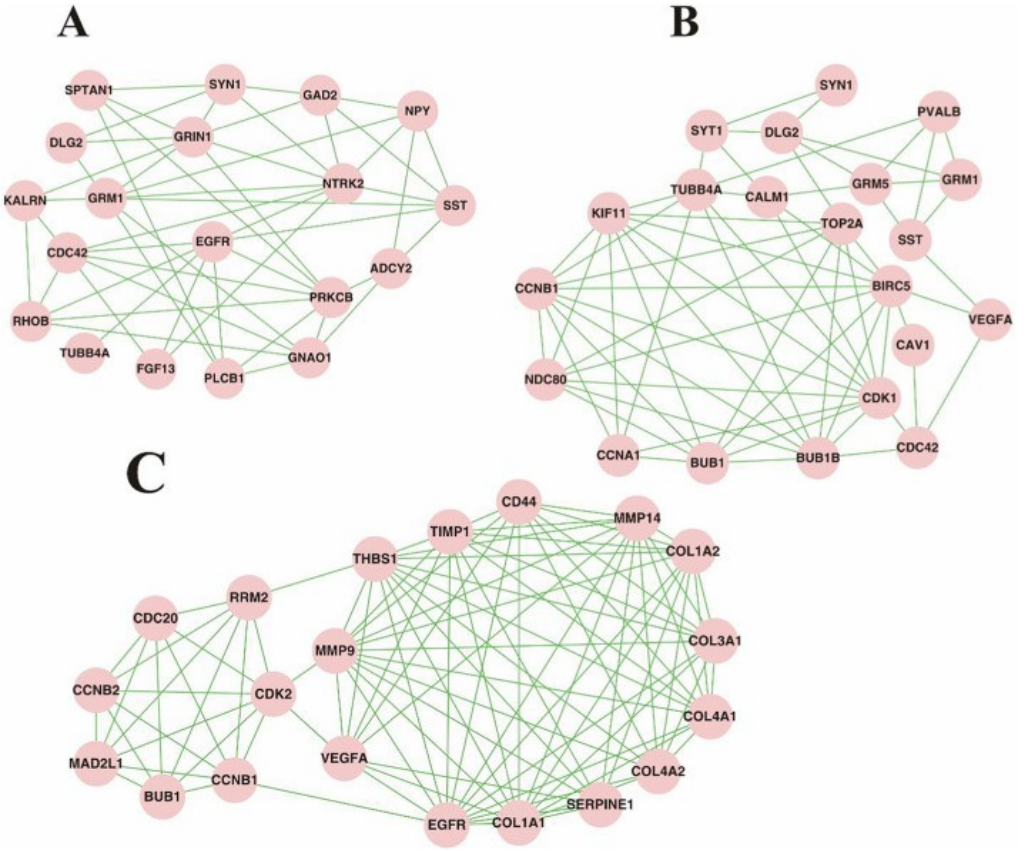
Esse caminho é o mais curto? Não necessariamente...



Componentes conexos

Ex:

- redes sociais (bolhas)
- redes de interação proteína-proteína

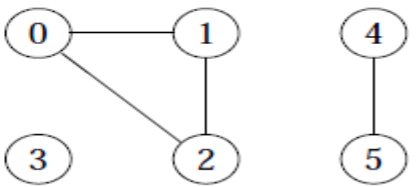


Grafo conectado (ou conexo)

- Um grafo não direcionado é conectado se cada par de vértices está conectado por um caminho.

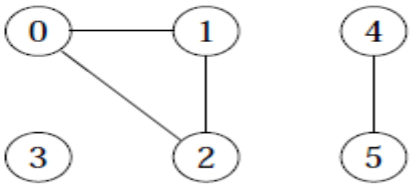
Ex: esse grafo é conectado?

Não!



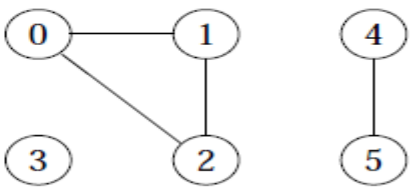
Componentes conectados (ou componentes conexos)

Ex.: Os componentes são: ?



Componentes conectados (ou componentes conexos)

Ex.: Os componentes são: $\{0, 1, 2\}$, $\{4, 5\}$ e $\{3\}$.

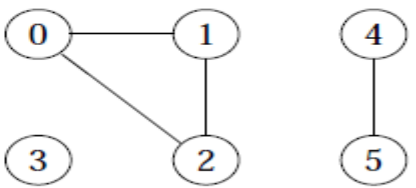


Poderia ser {0}, {1, 2}, {4, 5} e {3} ?

Componentes conectados (ou componentes conexos)

Um subgrafo conexo H de um grafo G é **maximal** se H não é subgrafo próprio de algum subgrafo conexo de G. Um **componente** (ou **componente conexo**) de um grafo G é qualquer subgrafo conexo maximal de G. É claro que cada vértice de um grafo pertence a um e um só componente. É claro também que um grafo é conexo se e somente se tem um único componente.

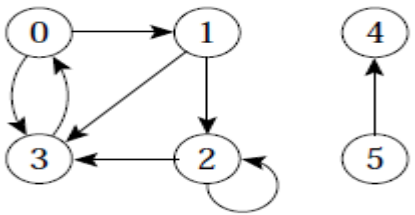
Ex.: Os componentes são: $\{0, 1, 2\}$, $\{4, 5\}$ e $\{3\}$.



Poderia ser {0}, {1, 2}, {4, 5} e {3} ? NÃO

Componentes Fortemente Conectados

- Um grafo direcionado $G = (V, A)$ é **fortemente conectado** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os **componentes fortemente conectados** de um grafo direcionado G são os subgrafos fortemente conexos maximais de G.
- Um **grafo direcionado fortemente conectado** tem apenas um componente fortemente conectado.

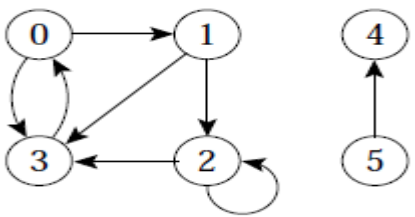


Quais são os componentes fortemente conectados deste grafo?

?

Componentes Fortemente Conectados

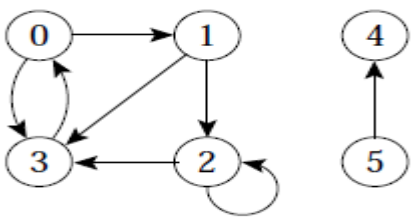
- Um grafo direcionado $G = (V, A)$ é **fortemente conectado** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os **componentes fortemente conectados** de um grafo direcionado G são os subgrafos fortemente conexos maximais de G.
- Um **grafo direcionado fortemente conectado** tem apenas um componente fortemente conectado.



Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.

Componentes Fortemente Conectados

- Um grafo direcionado $G = (V, A)$ é **fortemente conectado** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os **componentes fortemente conectados** de um grafo direcionado G são os subgrafos fortemente conexos maximais de G.
- Um **grafo direcionado fortemente conectado** tem apenas um componente fortemente conectado.



Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.

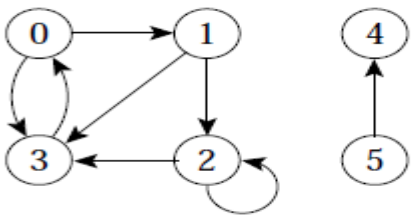
Existiria grafo direcionado conectado porém não fortemente?

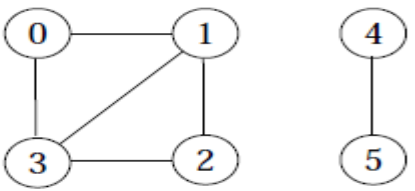
Grafos fracamente conexos

Um grafo direcionado é fracamente conexo se sua versão não direcionada for conexa

Versão Não Direcionada de um Grafo Direcionado

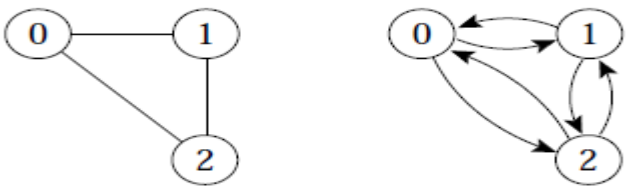
- A versão não direcionada de um grafo direcionado $G = (V, A)$ é um grafo não direcionado $G' = (V', A')$ onde $(u, v) \in A'$ se e somente se $u \neq v$ e $(u, v) \in A$.
- A versão não direcionada contém as arestas de G sem a direção e sem os *self-loops*.





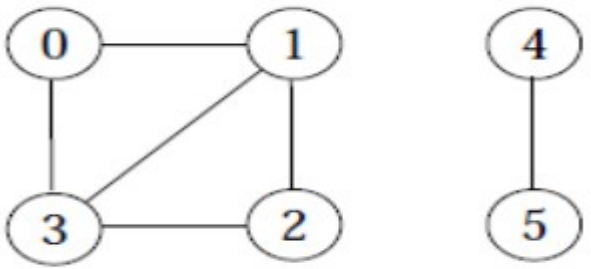
Versão Direcionada de um Grafo Não Direcionado

- A versão direcionada de um grafo não direcionado $G = (V, A)$ é um grafo direcionado $G' = (V', A')$ onde $(u, v) \in A'$ se e somente se $(u, v) \in A$.
- Cada aresta não direcionada (u, v) em G é substituída por duas arestas direcionadas (u, v) e (v, u)



Componentes conexos de um grafo não direcionado

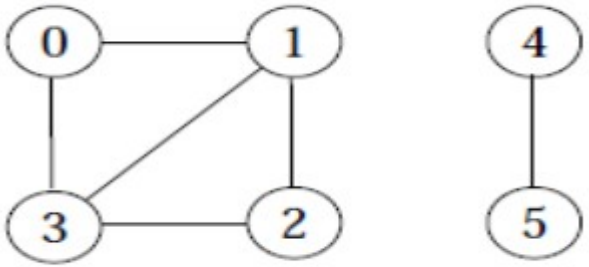
Como identificar?



Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

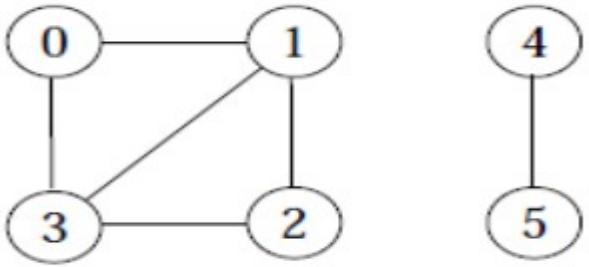
- cada vértice está contido em exatamente um componente conexo
- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v
- Então...



Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo
- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v
- Então...cada árvore de busca em profundidade corresponde a um componente conexo

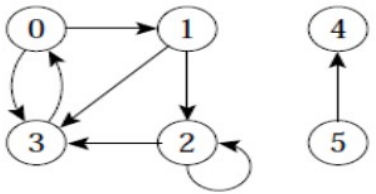


Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo
- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v
- Então...cada árvore de busca em profundidade corresponde a um componente conexo

Isso também vale para grafos direcionados e componentes fortemente conectados?



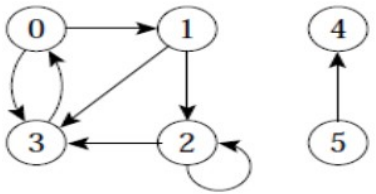
Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo
- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v
- Então...cada árvore de busca em profundidade corresponde a um componente conexo

Isso também vale para grafos direcionados e componentes fortemente conectados? SIM!



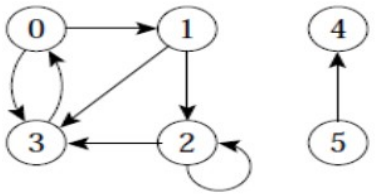
Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo
- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v
- Então...cada árvore de busca em profundidade corresponde a um componente conexo

Isso também vale para grafos direcionados e componentes fortemente conectados?



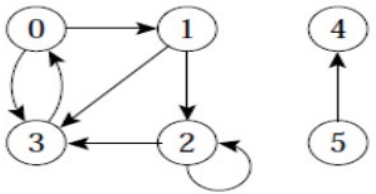
Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo
- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v
- Então...cada árvore de busca em profundidade corresponde a um componente conexo

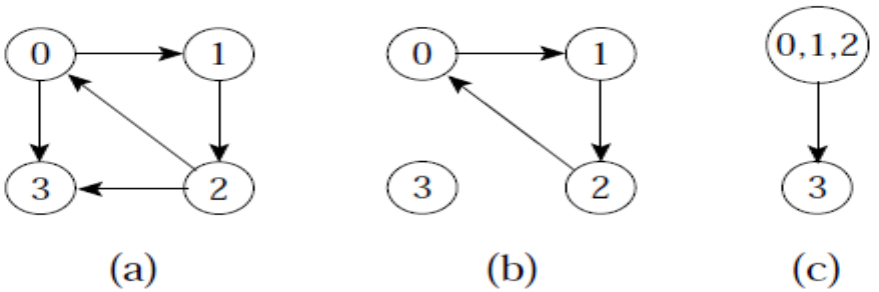
Isso também vale para grafos direcionados e componentes fortemente conectados? NÃO!



Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.

Componentes Fortemente Conectados

- Um componente fortemente conectado de $G = (V, A)$ é um conjunto maximal de vértices $C \subseteq V$ tal que para todo par de vértices u e v em C, u e v são mutuamente alcançáveis
- Podemos particionar V em conjuntos V_i, $1 \leq i \leq r$, tal que vértices u e v são equivalentes se e somente se existe um caminho de u a v e um caminho de v a u.

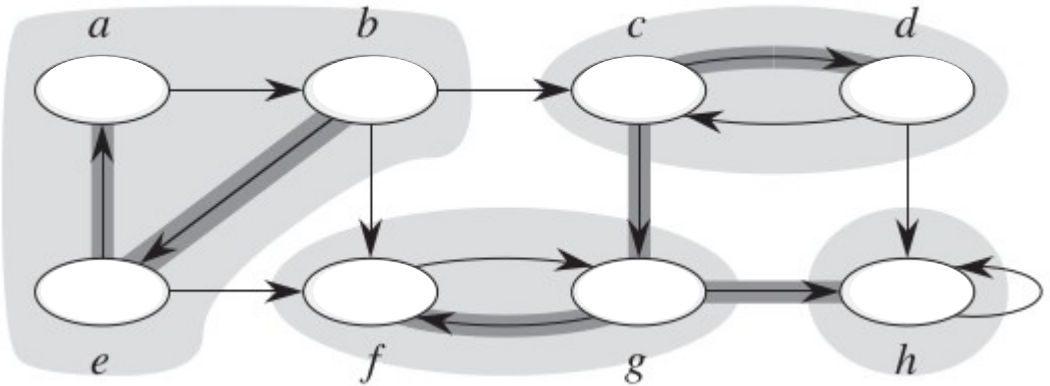


Grafo

Componentes fortemente conexos

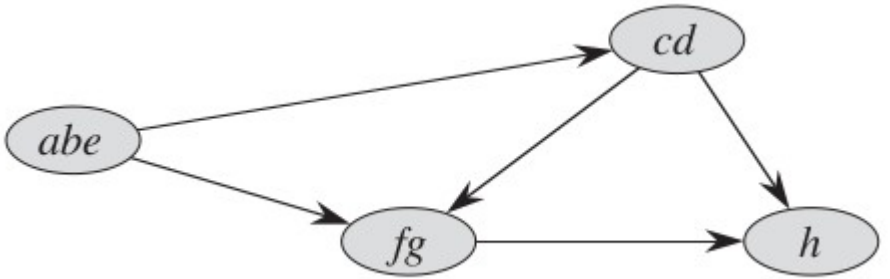
Grafo simplificado (sempre ACÍCLICO!!! - DAG!)

Outro exemplo:



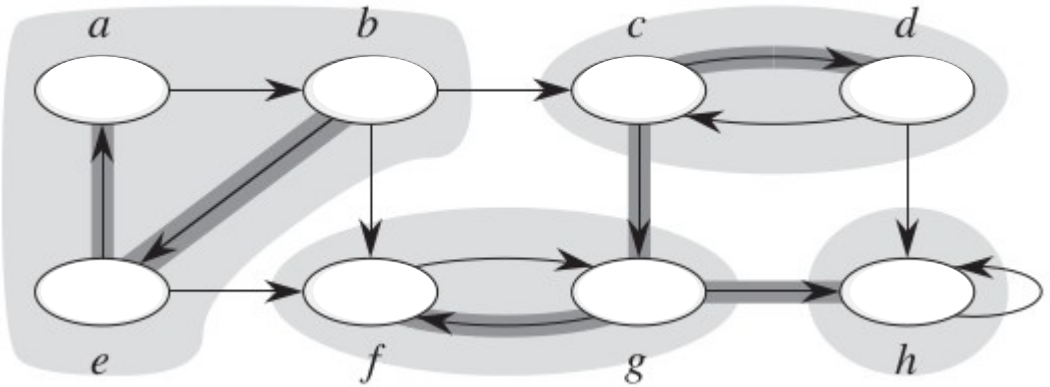
Grafo

(componentes fortemente conexos nas áreas cinzas; ignore as arestas grossas por enquanto)



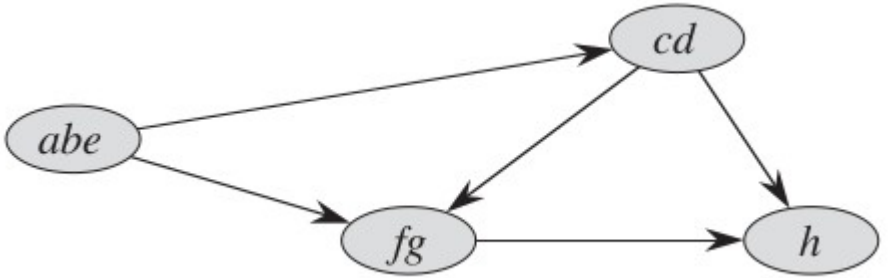
Grafo simplificado
(sempre ACÍCLICO!!! - DAG!)
Por quê?

Outro exemplo:



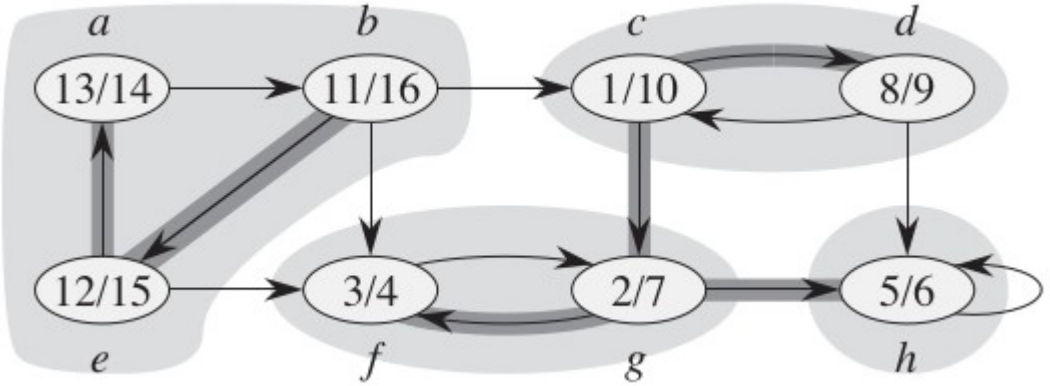
Grafo

(componentes fortemente conexos nas áreas cinzas; ignore as arestas grossas por enquanto)



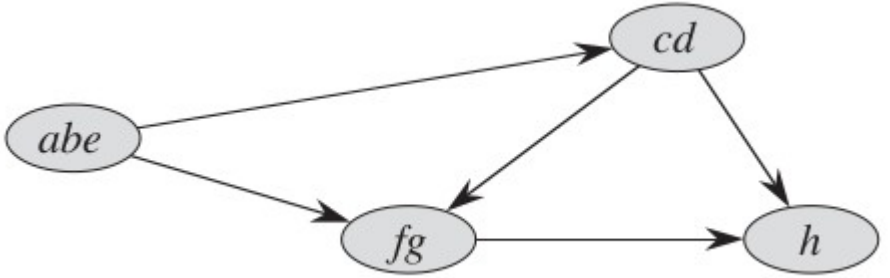
Grafo simplificado
(sempre ACÍCLICO!!! - DAG!)
Por que se eu tivesse um ciclo, todos os vértices deste ciclo pertenceriam ao mesmo componente conexo!

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

(componentes fortemente conexos nas áreas cinzas)

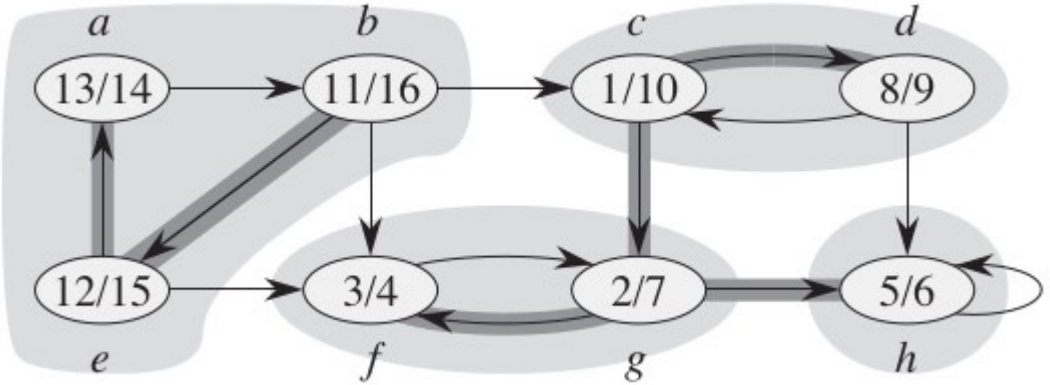


Grafo simplificado

(sempre ACÍCLICO!!! - DAG!)

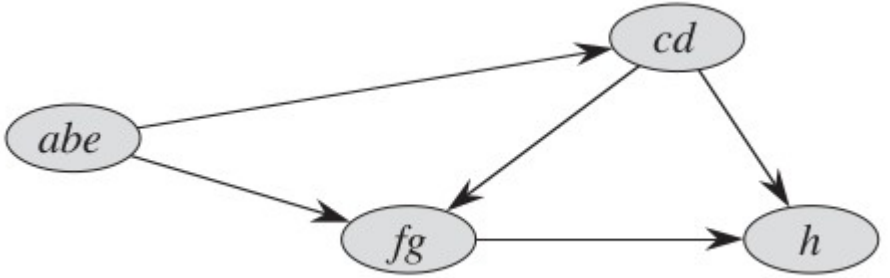
O que acontece se eu fizer uma busca em profundidade no grafo **transposto**, começando pelo vértice b (com maior tempo de término?)

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

(componentes fortemente conexos nas áreas cinzas)

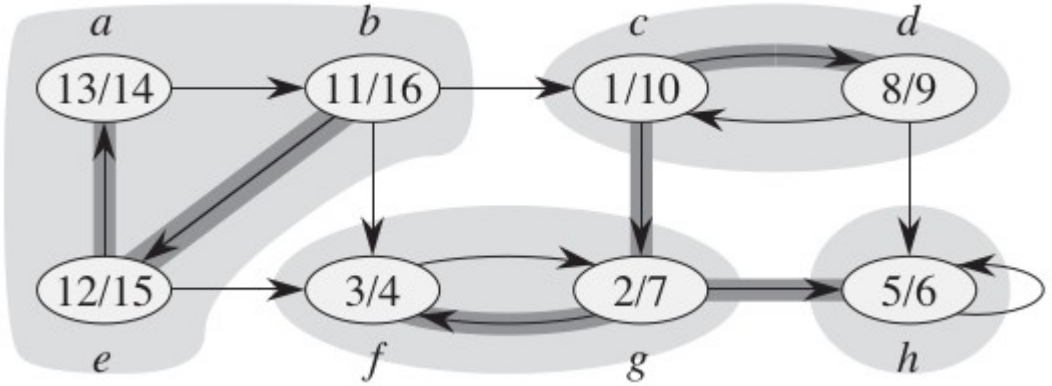


Grafo simplificado

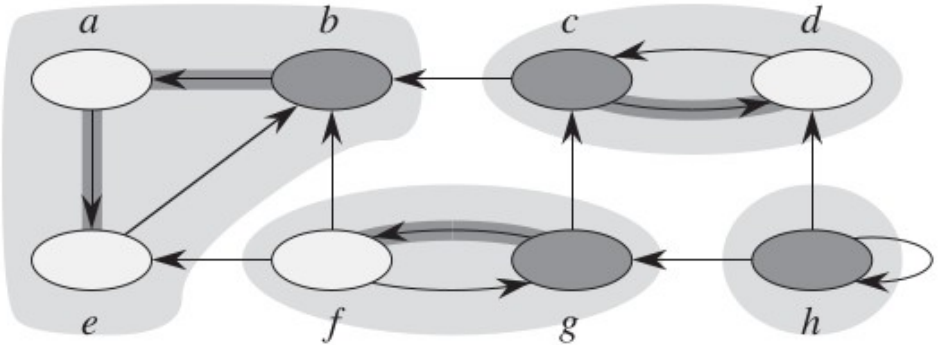
(sempre ACÍCLICO!!! - DAG!)

O que acontece se eu fizer uma busca em profundidade no grafo **transposto**, começando pelo vértice b (com maior tempo de término?) Continuarei achando todos os vértices do componente fortemente conectado de b, mas como o grafo está transposto não consigo acessar os demais componentes...

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

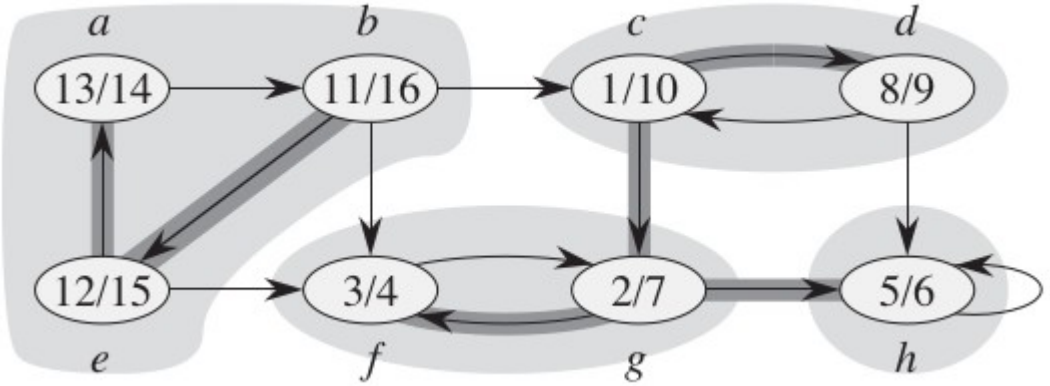


Grafo transposto

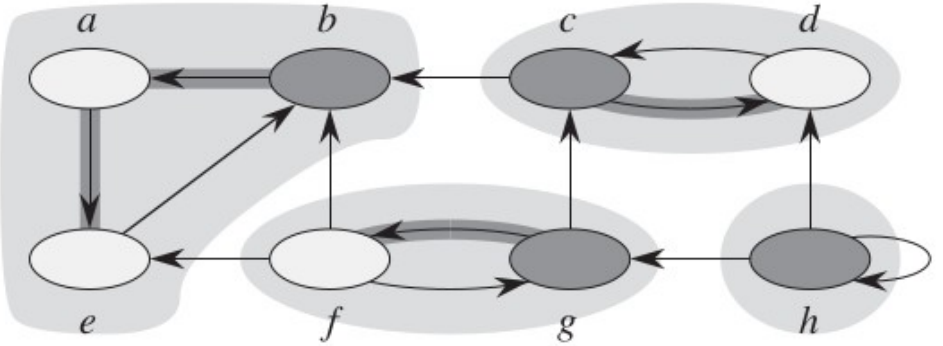
(componentes fortemente conexos nas áreas cinzas)

O que acontece se eu fizer uma busca em profundidade no grafo **transposto**, começando pelo vértice b (com maior tempo de término?) Continuarei achando todos os vértices do componente fortemente conectado de b, mas como o grafo está transposto não consigo acessar os demais componentes… Da mesma forma, se retomar a busca em profundidade a partir do vértice c, irei...

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

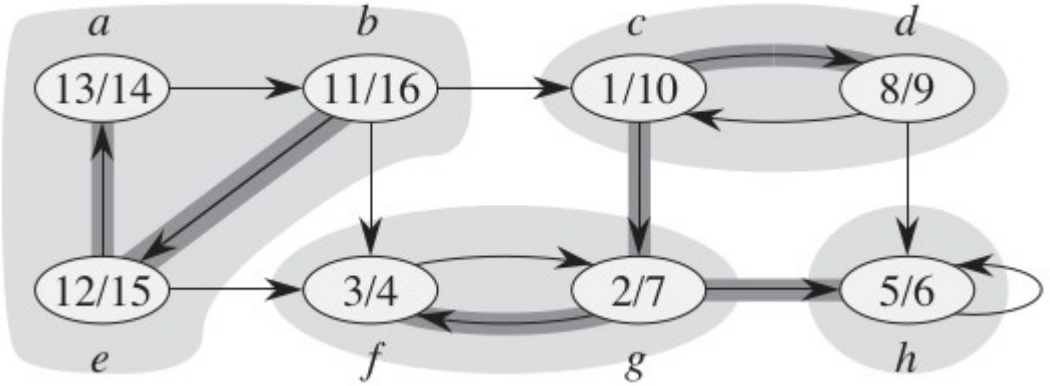


Grafo transposto

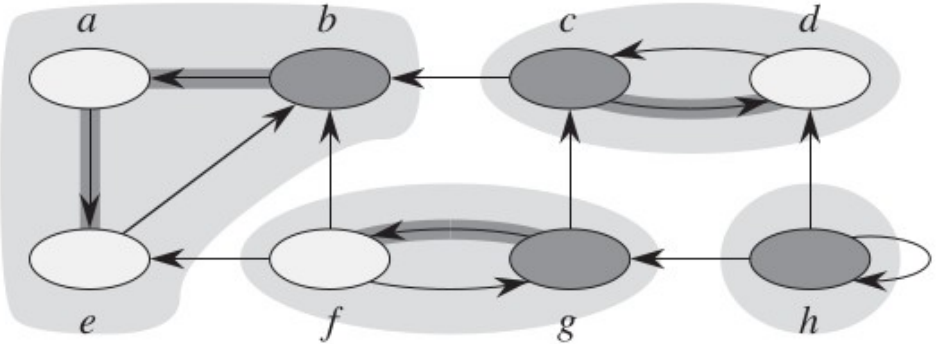
(componentes fortemente conexos nas áreas cinzas)

O que acontece se eu fizer uma busca em profundidade no grafo **transposto**, começando pelo vértice b (com maior tempo de término?) Continuarei achando todos os vértices do componente fortemente conectado de b, mas como o grafo está transposto não consigo acessar os demais componentes… Da mesma forma, se retomar a busca em profundidade a partir do vértice c, irei achar todos os vértices do componente de c, mas não dos demais. O mesmo retomando do g, e depois do h.

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

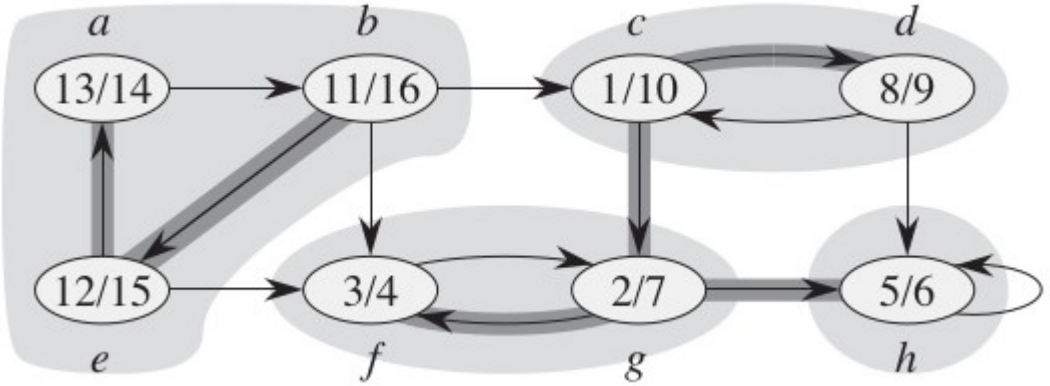


Grafo transposto

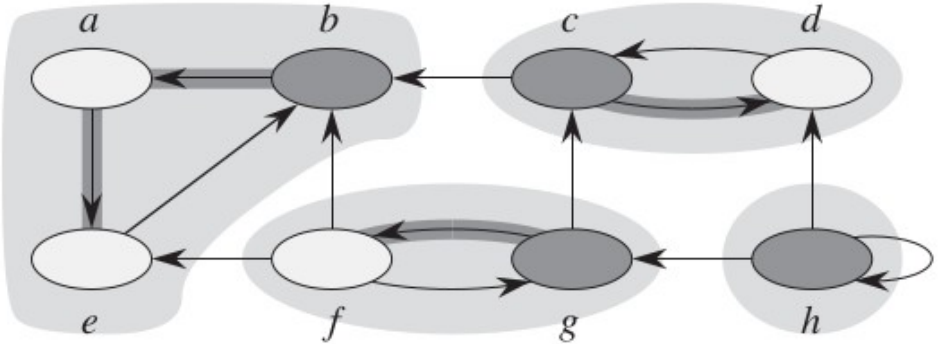
(componentes fortemente conexos nas áreas cinzas)

Ou seja, cada árvore em profundidade da segunda busca é um componente fortemente conectado!!!

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

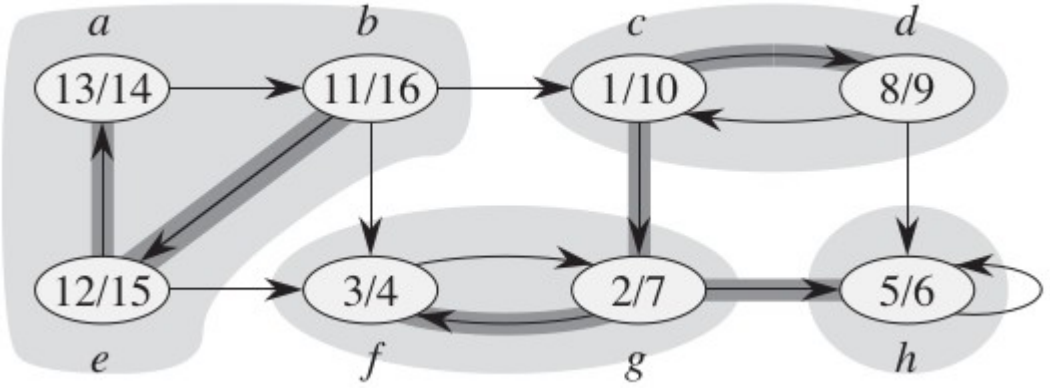


Grafo transposto

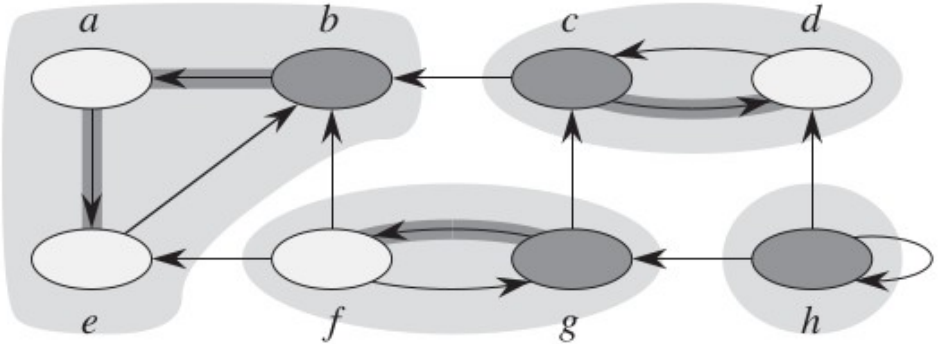
(componentes fortemente conexos nas áreas cinzas)

Por que essa ordem de vértices (decrescente pelo tempo de término) para iniciar a busca em profundidade no grafo transposto foi importante para dar certo?

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

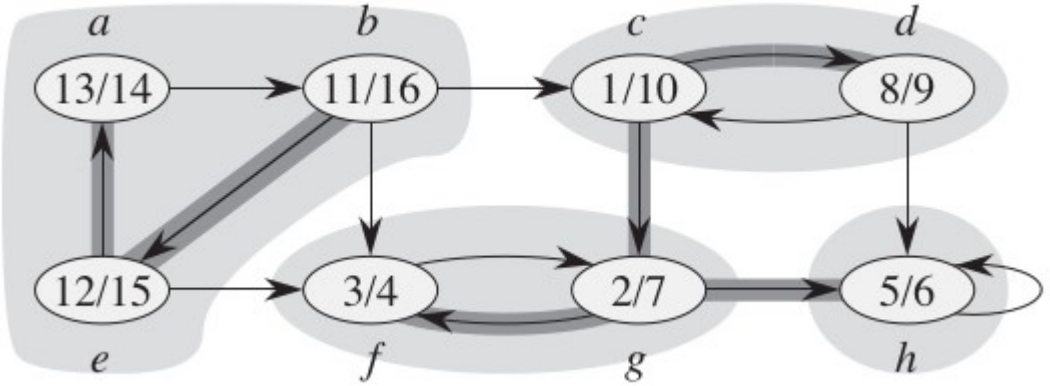


Grafo transposto

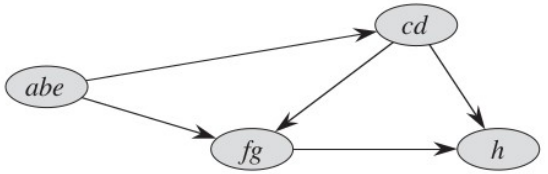
(componentes fortemente conexos nas áreas cinzas)

Por que essa ordem de vértices (decrescente pelo tempo de término) para iniciar a busca em profundidade no grafo transposto foi importante para dar certo? Se eu tivesse começado pelo f (por ex), acessaria vértices de componentes diferentes…

Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo

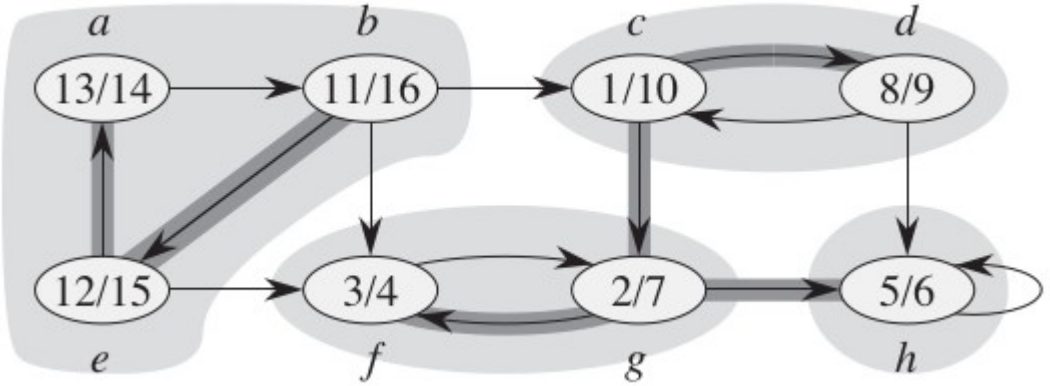


Grafo simplificado

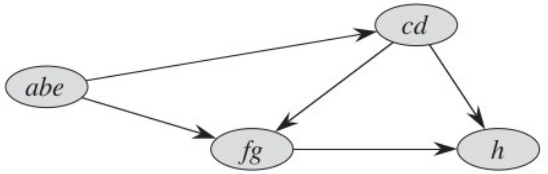
(componentes fortemente conexos nas áreas cinzas)

Por que essa ordem de vértices (decrescente pelo tempo de término) para iniciar a busca em profundidade no grafo transposto foi importante para dar certo? Se eu tivesse começado pelo f (por ex), acessaria vértices de componentes diferentes… Começar pelo vértice de maior tempo de término corresponde a uma ordenação topológica do grafo simplificado (abe)→ (c,d)→ (f,g) → (h)

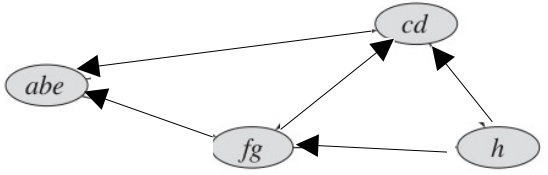
Agora considere a seguinte busca em profundidade no grafo original (arestas grossas são as arestas de árvore):



Grafo



Grafo simplificado



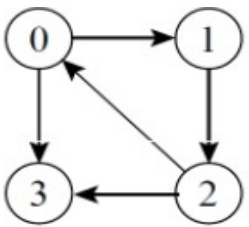
Grafo simplificado transposto

(componentes fortemente conexos nas áreas cinzas)

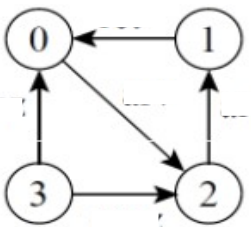
Por que essa ordem de vértices (decrescente pelo tempo de término) para iniciar a busca em profundidade no grafo transposto foi importante para dar certo? Se eu tivesse começado pelo f (por ex), acessaria vértices de componentes diferentes… Começar pelo vértice de maior tempo de término corresponde a uma ordenação topológica do grafo simplificado (abe)→ (c,d)→ (f,g) → (h), mas como a segunda busca em profundidade é realizada no grafo transposto, cada árvore de busca em profund. só alcança vértices do mesmo componente!

Componentes Fortemente Conectados: Algoritmo

- Usa o **transposto** de G, definido $G^T = (V, A^T)$, onde $A^T = \{(u, v) : (v, u) \in A\}$, isto é, A^T consiste das arestas de G com suas direções invertidas.
- G e G^T possuem os mesmos componentes fortemente conectados, isto é, u e v são mutuamente alcançáveis a partir de cada um em G se e somente se u e v são mutuamente alcançáveis a partir de cada um em G^T.



(a) Grafo original



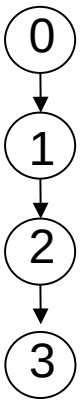
(b) Grafo transposto

Componentes Fortemente Conectados: Algoritmo

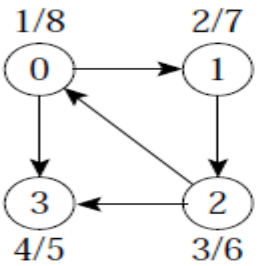
1. Chama *BuscaEmProfundidade*(G) para obter os tempos de término $t[u]$ para cada vértice u.
2. Obtem G^T.
3. Chama *BuscaEmProfundidade*(G^T), realizando a busca a partir do vértice de maior $t[u]$ obtido na linha 1. Inicie uma nova busca em profundidade a partir do vértice de maior $t[u]$ dentre os vértices restantes se houver.
4. Retorne os vértices de cada árvore da floresta obtida como um componente fortemente conectado separado.

Componentes Fortemente Conectados: Exemplo

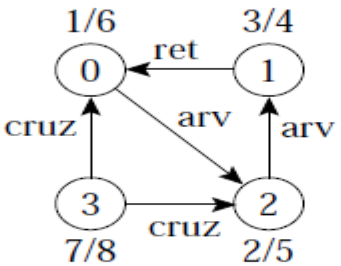
- A parte (b) apresenta o resultado da busca em profundidade sobre o grafo transposto obtido, mostrando os tempos de término e a classificação das arestas.
- A busca em profundidade em G^T resulta na floresta de árvores mostrada na parte (c).



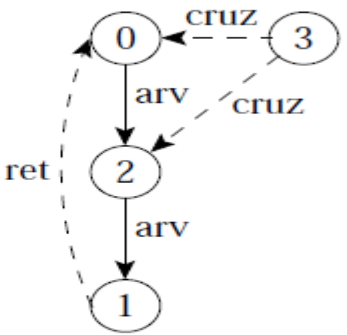
Árvore de busca em G



(a) G



(b) G^T



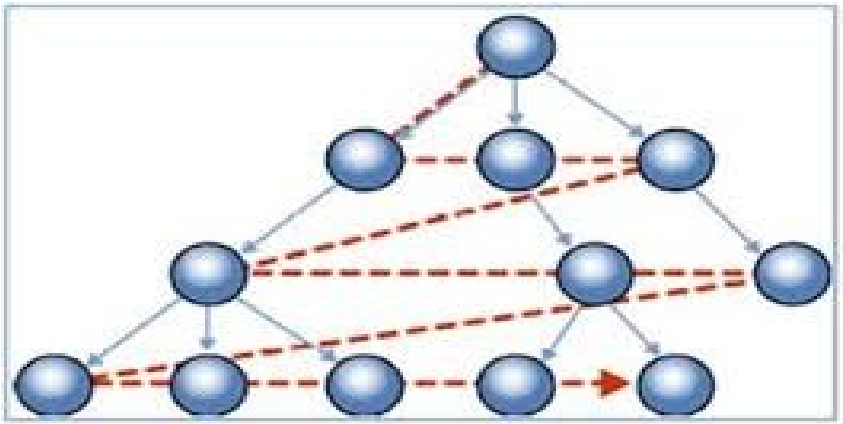
(c) Árvores de busca de G^T

Componentes Fortemente Conectados: Análise

- Utiliza o algoritmo para busca em profundidade duas vezes, uma em G e outra em G^T.
- Logo, a complexidade total é $O(|V| + |A|)$.

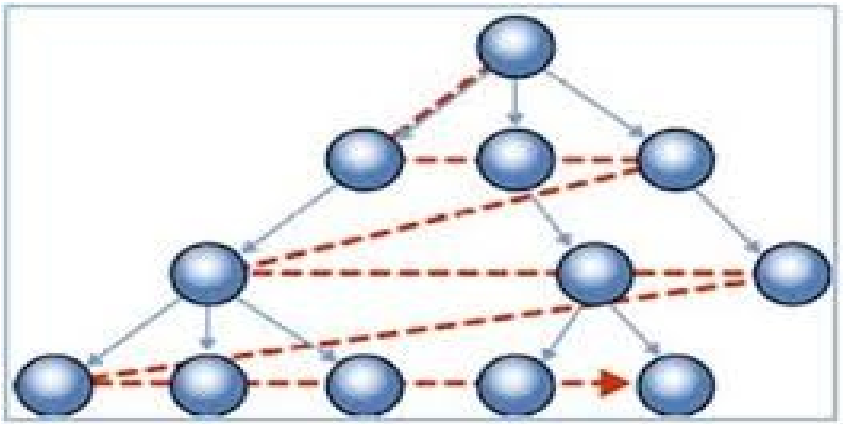
Considerando que para calcular G^T também gastamos O(V + A)

Busca em Largura



Busca em Largura

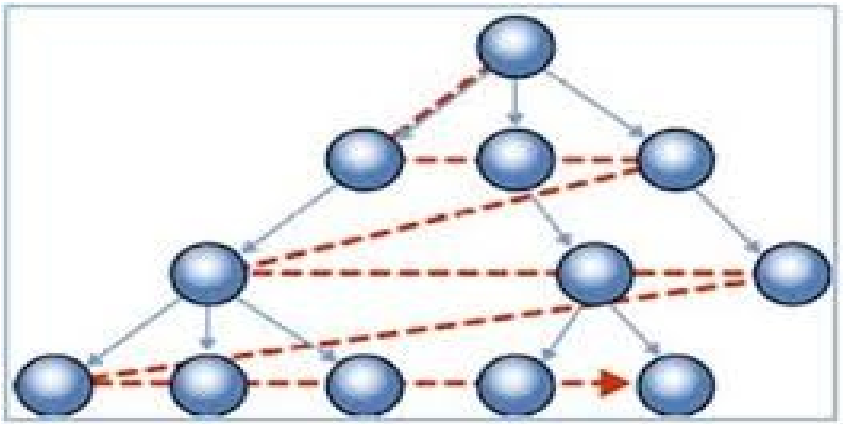
- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.



Que estrutura de dados de suporte precisamos para gerenciar a ordem de vértices sendo processados?

Busca em Largura

- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.



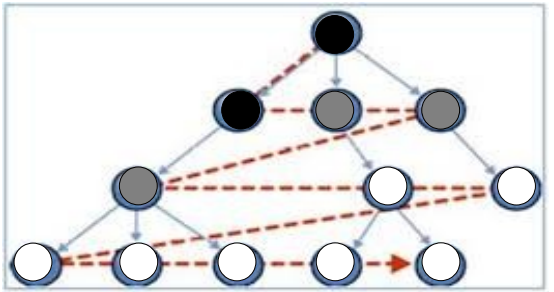
Que estrutura de dados de suporte precisamos para gerenciar a ordem de vértices sendo processados?

FILA!

Busca em Largura

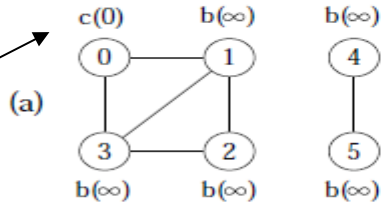
- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.

Busca em Largura



- Cada vértice é colorido de branco, cinza ou preto
- Todos os vértices são inicializados branco (que significa que ainda não foi descoberto)
- Quando um vértice é descoberto pela primeira vez torna-se cinza (isto é, vai para a fila de processamento – terei que olhar seus adjacentes mais tarde)
- Quando termino de processar todos os adjacentes de um vértice (cinza), ele torna-se preto
- Vértices cinza e preto, portanto, já foram descobertos, mas a distinção de cor garante que a busca ocorra em largura
- Durante a busca, se (u,v) é uma aresta e u é preto, então v tem que ser cinza ou preto
- Vértices cinza podem ter alguns vértices adjacentes brancos, e eles representam a fronteira entre vértices descobertos e não descobertos.

Busca em Largura: Exemplo



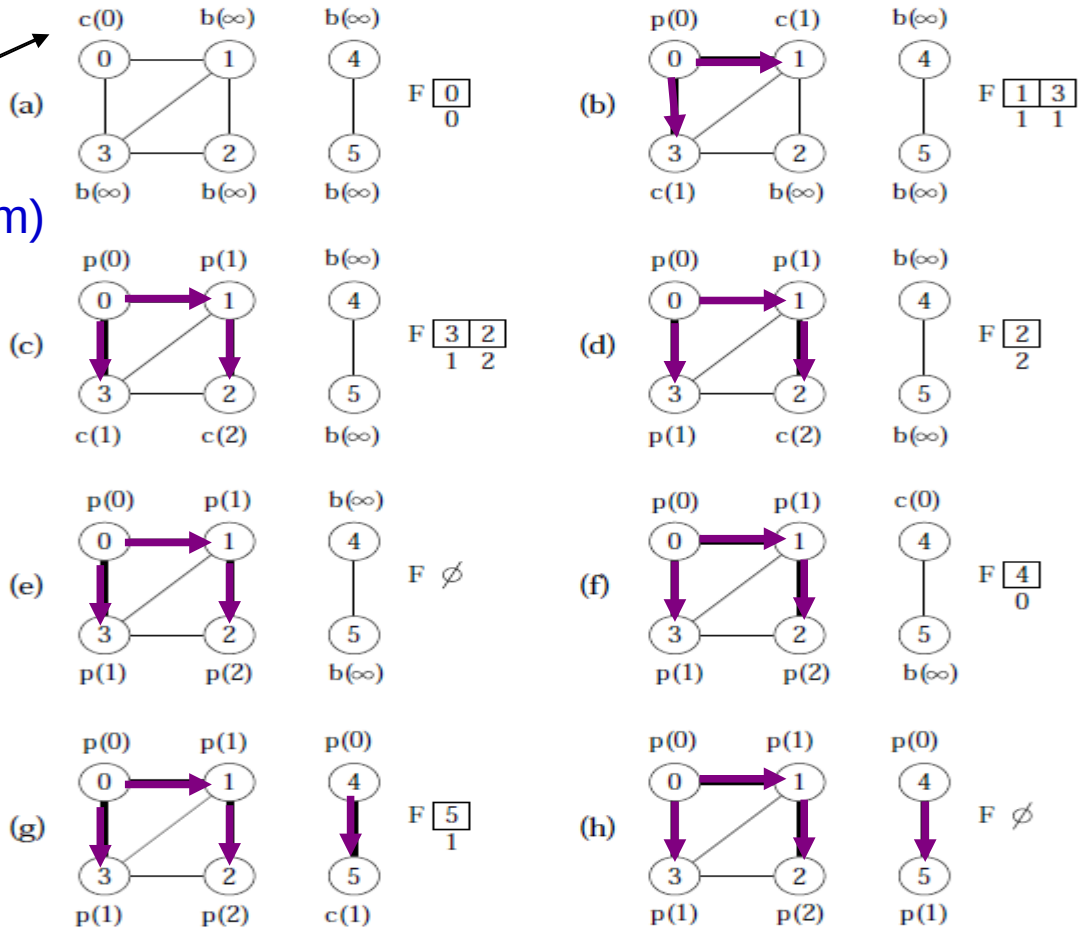
F 0
0

Fila:
Número do vértice (cinza)
distância desse vértice à “origem”

Cada vértice tem:
cor(distância da “origem”)
e antecessor (→)

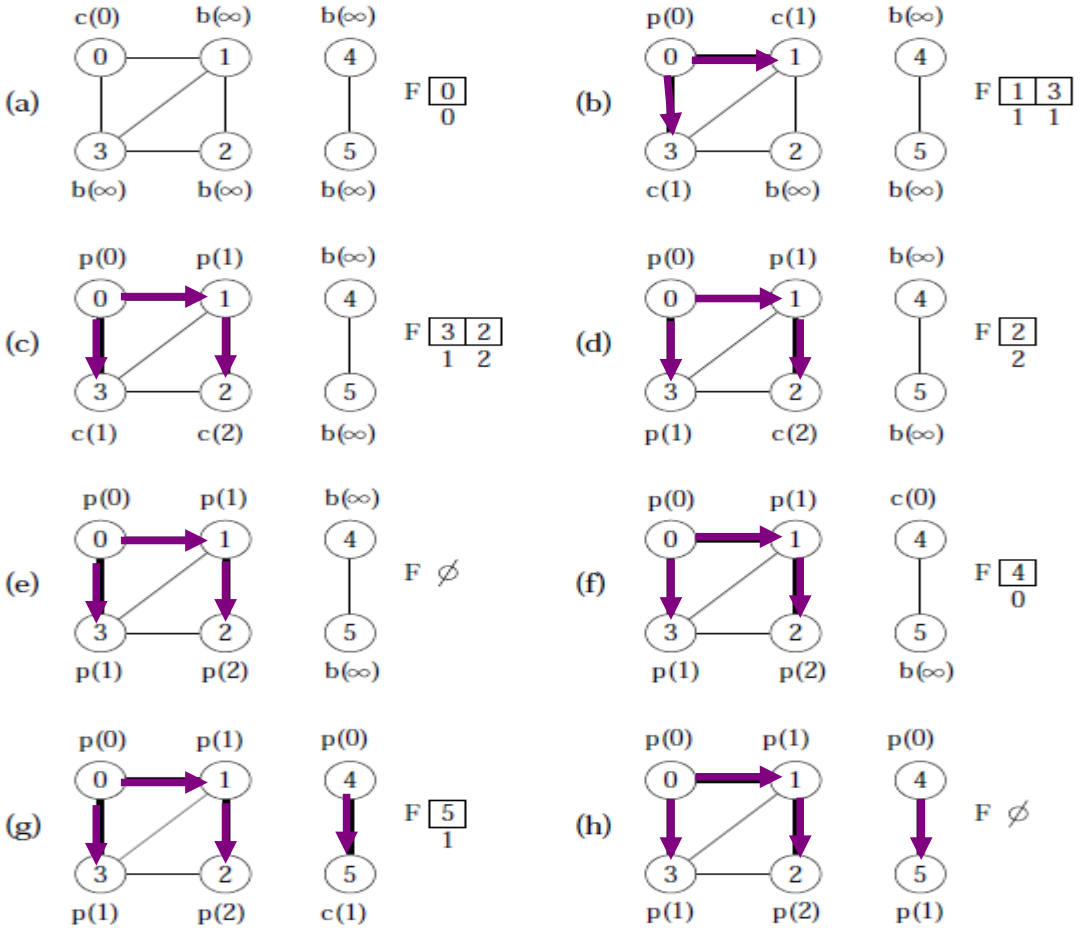
Busca em Largura: Exemplo

Cada vértice tem:
cor(distância da origem)



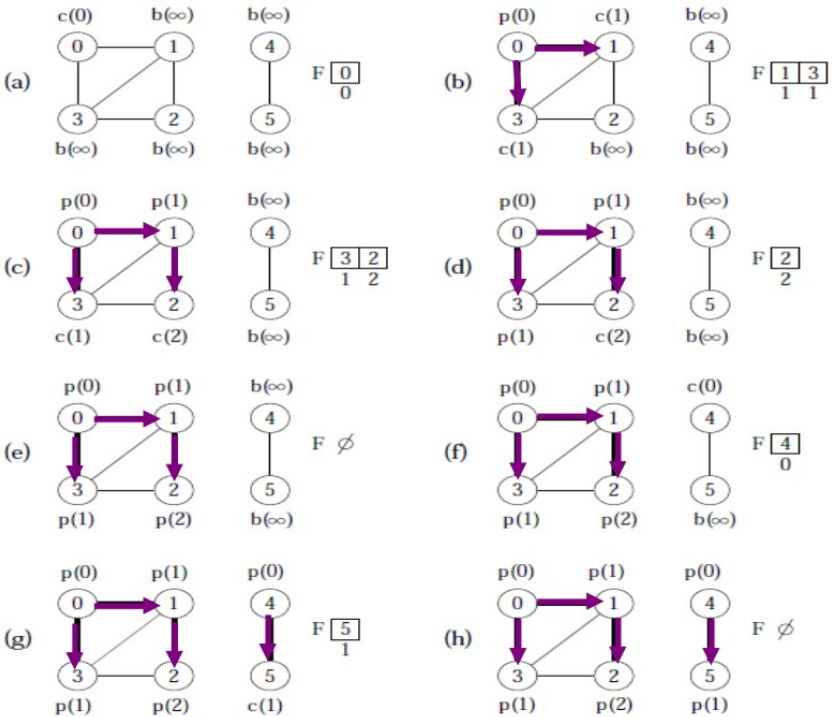
Busca em Largura: Exemplo

Cada vértice tem:



Implementação

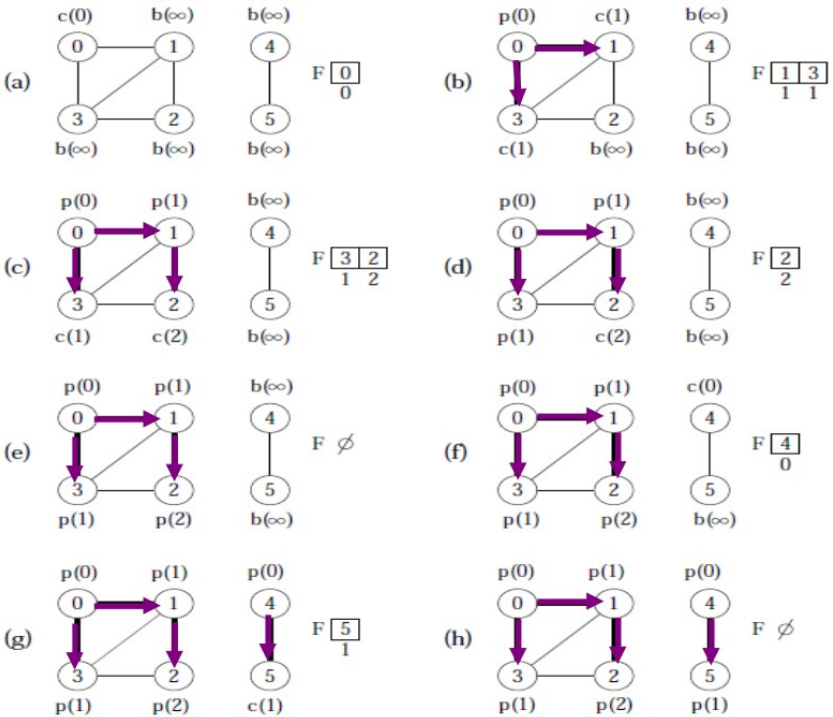
```
buscaEmLargura(grafo){

    Inicializações e chamadas à visitaLargura

}

visitaLargura(s, grafo, cor, antecessor, distancia){
```



Implementação

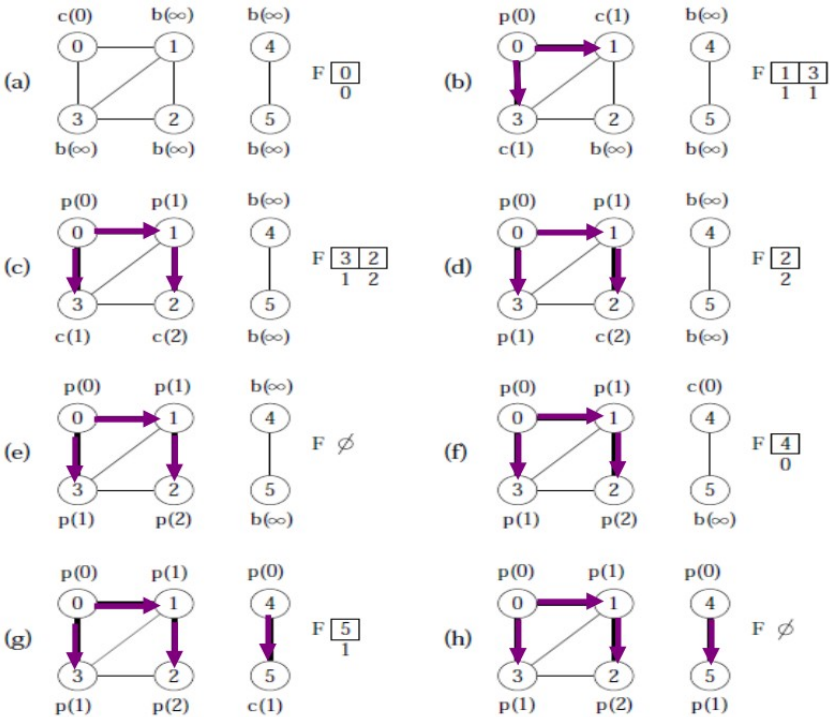
```
buscaEmLargura(grafo){
      Aloca vetores cor, antecessor, distancia com tamanho grafo->nrVertices
       Para cada vertice v
              cor[v] ← branco;  antecessor[v] ← -1; distancia[v] ← ∞;
       Para cada vertice v
              se cor[v] = branco
                visitaLargura(v, grafo, cor, antecessor, distancia);
}

visitaLargura(s, grafo, cor, antecessor, distancia){
```

Como usaremos uma fila, essa função NÃO será recursiva



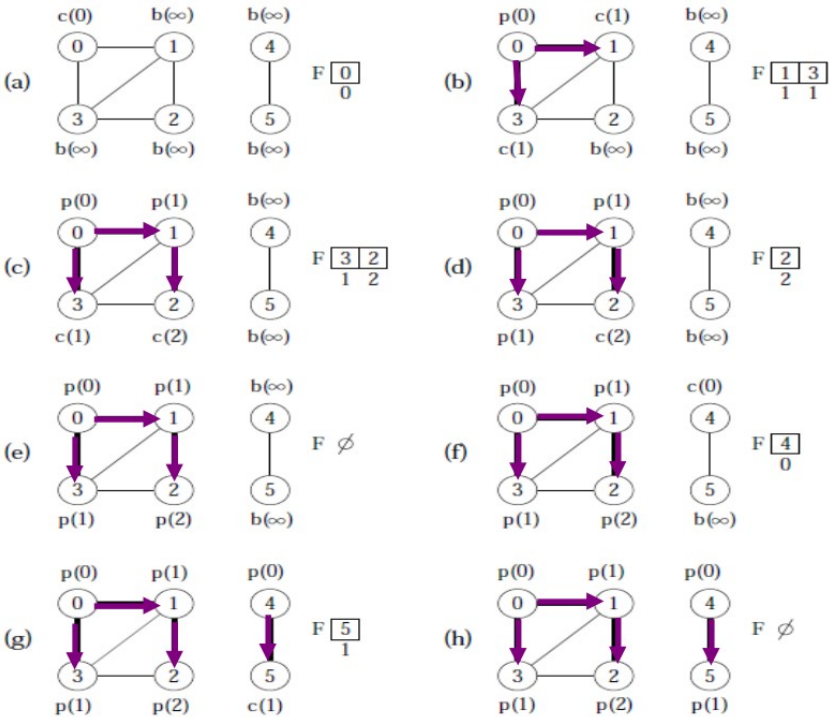
Implementação

```
buscaEmLargura(grafo){
      Aloca vetores cor, antecessor, distancia com tamanho grafo->nrVertices
         Para cada vertice v
                  cor[v] ← branco;  antecessor[v] ← -1; distancia[v] ← ∞;
         Para cada vertice v
                  se cor[v] = branco
                  visitaLargura(v, grafo, cor, antecessor, distancia);
}

visitaLargura(s, grafo, cor, antecessor, distancia){
      cor[s] ← cinza;
      distancia[s] ← 0;
      F  ← Ø;
      insereFila(F, s);
      enquanto F ≠ Ø
            w  ←  removeFila(F)
            para cada vertice u da lista de adjacência de w
                  se cor[u] = branco
                        cor[u] ← cinza;
                        antecessor[u] ← w;
                        distancia[u] ← distancia[w] + 1;
                        insereFila(F, u);
            cor[w] ← preto;
}
```



Implementação

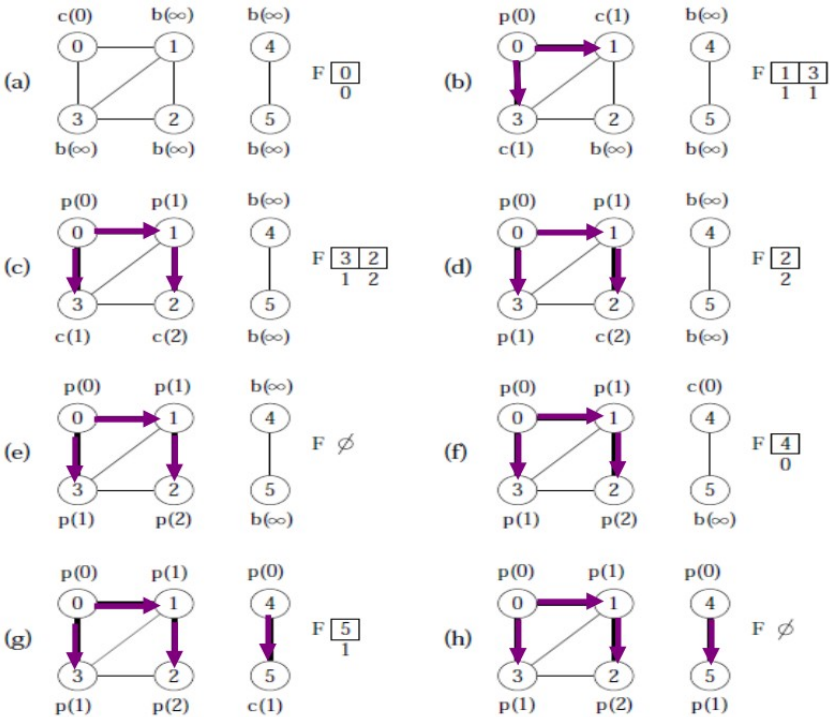
```
buscaEmLargura(grafo){
      Aloca vetores cor, antecessor, distancia com tamanho grafo->nrVertices
         Para cada vertice v
                  cor[v] ← branco;  antecessor[v] ← -1; distancia[v] ← ∞;
         Para cada vertice v
                  se cor[v] = branco
                  visitaLargura(v, grafo, cor, antecessor, distancia);
}

visitaLargura(s, grafo, cor, antecessor, distancia){
      cor[s] ← cinza;
      distancia[s] ← 0;
      F  ← Ø;
      insereFila(F, s);
      enquanto F ≠ Ø
            w  ←  removeFila(F)
            para cada vertice u da lista de adjacência de w
                  se cor[u] = branco
                        cor[u] ← cinza;
                        antecessor[u] ← w;
                        distancia[u] ← distancia[w] + 1;
                        insereFila(F, u);
            cor[w] ← preto;
}
```



Complexidade: ?

Implementação

```
buscaEmLargura(grafo){
    Aloca vetores cor, antecessor, distancia com tamanho grafo->nrVertices
      Para cada vertice v
            cor[v] ← branco;  antecessor[v] ← -1; distancia[v] ← ∞;
      Para cada vertice v
            se cor[v] = branco
           visitaLargura(v, grafo, cor, antecessor, distancia);
}

visitaLargura(s, grafo, cor, antecessor, distancia){
    cor[s] ← cinza;
    distancia[s] ← 0;
    F  ← Ø;
    insereFila(F, s);
    enquanto F ≠ Ø
        w  ←  removeFila(F)
        para cada vertice u da lista de adjacência de w
            se cor[u] = branco
                cor[u] ← cinza;
                antecessor[u] ← w;
                distancia[u] ← distancia[w] + 1;
                insereFila(F, u);
        cor[w] ← preto;
}
```



Complexidade: O(V+A)

Implementação

```
buscaEmLargura(grafo){
      Aloca vetores cor, antecessor, distancia com tamanho grafo->nrVertices
         Para cada vertice v
                  cor[v] ← branco;  antecessor[v] ← -1; distancia[v] ← ∞;
         Para cada vertice v
                  se cor[v] = branco
                 visitaLargura(v, grafo, cor, antecessor, distancia);
}

visitaLargura(s, grafo, cor, antecessor, distancia){
      cor[s] ← cinza;
      distancia[s] ← 0;
      F  ← Ø;
      insereFila(F, s);
      enquanto F ≠ Ø
            w  ←  removeFila(F)
            para cada vertice u da lista de adjacência de w
                  se cor[u] = branco
                        cor[u] ← cinza;
                        antecessor[u] ← w;
                        distancia[u] ← distancia[w] + 1;
                        insereFila(F, u);
            cor[w] ← preto;
}
```

"VisitaBfs" no livro do Ziviani (slides seguintes)

Complexidade: O(V+A)

Busca em Largura: Análise

- O custo de inicialização do primeiro anel em *BuscaEmLargura* é $O(|V|)$ cada um.
- O custo do segundo anel é também $O(|V|)$.
- *VisitaBfs*: enfileirar e desenfileirar têm custo $O(1)$, logo, o custo total com a fila é $O(|V|)$.
- Cada lista de adjacentes é percorrida no máximo uma vez, quando o vértice é desenfileirado.
- Desde que a soma de todas as listas de adjacentes é $O(|A|)$, o tempo total gasto com as listas de adjacentes é $O(|A|)$.
- Complexidade total: é $O(|V| + |A|)$.

Aplicações

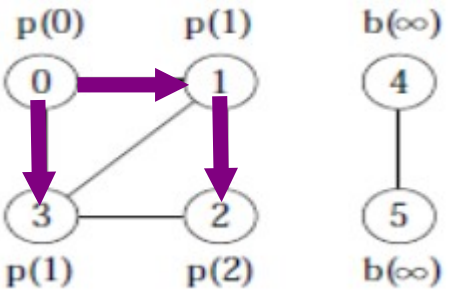
- Para encontrar os vértices vizinhos dentro de um certo raio (por exemplo, em sistemas de computação móvel, navegação GPS, etc)
- Pessoas a uma certa distância em uma rede social
- Coleta de lixo em memória (melhor localidade de referência do que se usar busca em profundidade)
- Caminhos mais curtos (NÃO é o mesmo que caminho mínimo)

Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Como uso a busca em largura?

Ex: caminho de 0 a 2



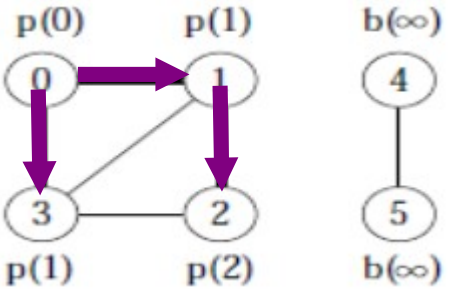
Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Como uso a busca em largura?

Chamo o visitaLargura em u (origem) até encontrar v

Ex: caminho de 0 a 2



Caminhos Mais Curtos

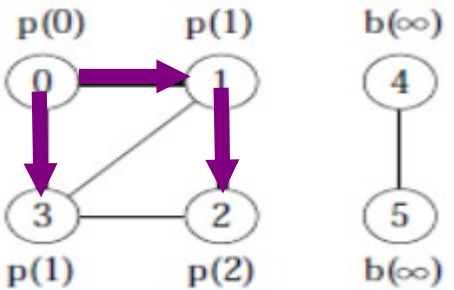
- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Imprimindo o caminho: (u tendo sido a origem da **visitaLargura**)

```
if (d[v] == ∞)
        printf ("Nao existe caminho de %d ate %d" , u, v) ;
else imprimeCaminho(u, v, antecessor);


void imprimeCaminho(int u, int v, int antecessor[])
{
```

DICA: usando recursão!

Ex: caminho de 0 a 2



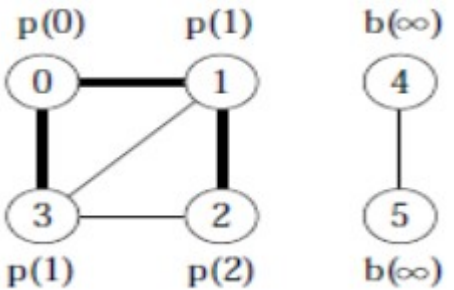
Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Imprimindo o caminho: (u tendo sido a origem da **visitaLargura**)

```
if (d[v] == ∞)
        printf ("Nao existe caminho de %d ate %d" , u, v) ;
else imprimeCaminho(u, v, antecessor);

void imprimeCaminho(int u, int v, int antecessor[])
{
        if (u == v) { printf ( "%d " , u ); return; }
        else {
                imprimeCaminho(u, antecessor[v], antecessor);
                printf ( "%d " , v);
        }
}
```

Ex: caminho de 0 a 2



Exercício de programação

Implementem Busca em Largura **com nossa interface**

Referências

Livro do Ziviani: cap 7 (seções 7.3 a 7.7)

Livro do Cormen: seção 22.2