Algoritmos e Estruturas de Dados - IF672

Prof. Dr. Fernando M. de Paula Neto
fernando@cin.ufpe.br
cin.ufpe.br/~fernando

Conteúdo modificado dos slides cedidos pelos professores
Hansenclever Bassani e Renato Vimieiro

cin.ufpe.br

Caminhamento em grafos

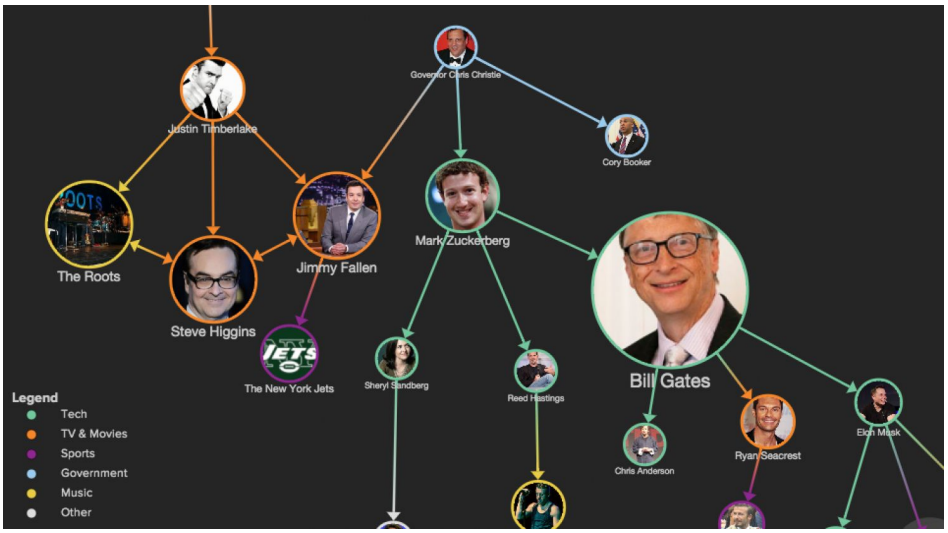
cin.ufpe.br

Objetivo

- Apresentar dois tipos de encaminhamento em grafos:
  - largura
  - profundidade

Introdução

- Os algoritmos para **caminhamento em grafos** a serem estudados hoje têm aplicações em várias situações
- A busca em largura e em profundidade são amplamente usadas em Inteligência Artificial, Otimização Combinatorial, na verificação de propriedades de grafos ...
- Eles são usados não só para o caminhamento explícito em grafos, mas também na construção de algoritmos que executam buscas em espaços de soluções:
  - Exemplos em Mineração de Dados: Apriori (busca em largura); FPGrowth (busca em profundidade)

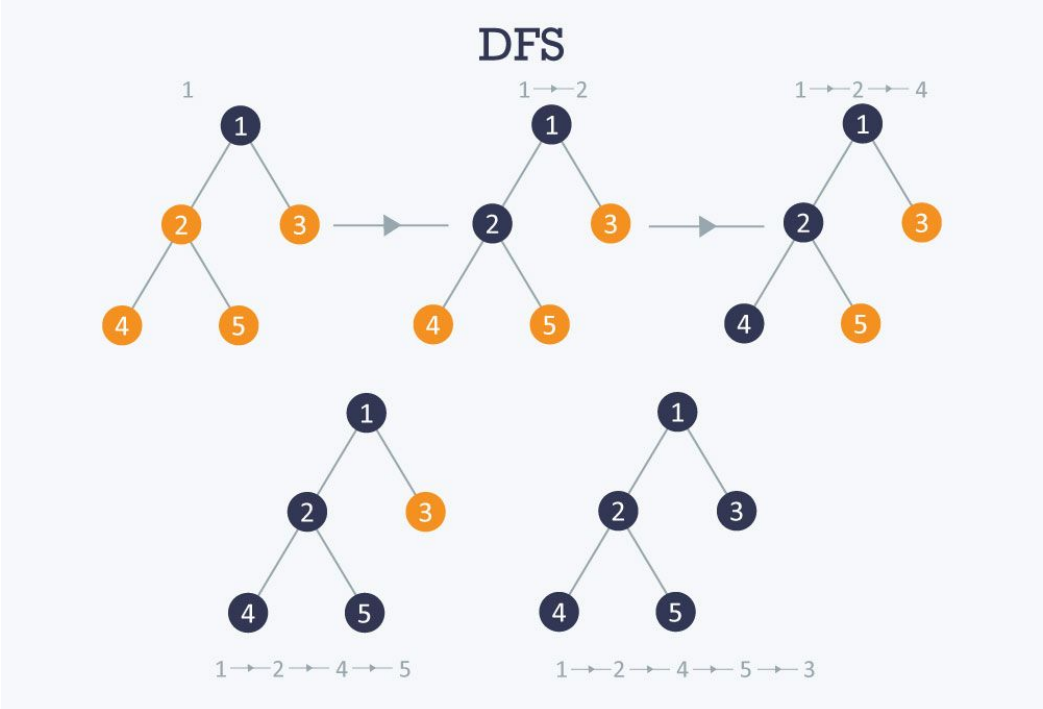


Introdução

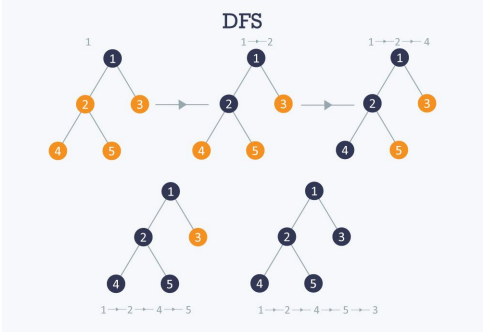
- Em grafos, a busca em largura e em profundidade podem ser usadas para:
  - Determinar se há, e qual é, o caminho mais curto entre dois vértices **u** e **v**
  - Determinar se há ciclos no grafo
  - Identificar os componentes conexos do grafo
  - Determinar a ordem de execução de tarefas onde existe interdependência

Busca em profundidade (Depth-first search)

- Livitin diz que a **busca em profundidade** é o **caminhamento dos bravos**
  - Os caminhos que os afastam cada vez mais de casa são sempre os escolhidos
- A ideia é: ao se deparar com um caminho ainda não percorrido, segui-lo
- No contexto de grafos:
  - Inicia-se a busca em um vértice origem, marcando-o como visitado
  - Visitar vértice adjacente ainda não visitado
  - Se não houver mais vértices adjacentes, retornar



Numa busca, para sabermos o caminho a percorrer até um determinado vértice do grafo, precisamos armazenar o elemento antecessor.



Antecessores:
de 2: 1
de 4: 2
de 5: 2
de 3: 1

vetor de antecessores

-1	1	1	2	2
1	2	3	4	5

Busca em profundidade (Depth-First Search – DFS)

```
def buscaProfundidade(g):
    marcado = g.V * [False]
    antecessor = g.V * [-1]
    for v in range(0,g.V):
        if !marcado[v]: dfs(g,v,antecessor,marcado)
    for i in range(0,g.V): print(antecessor[i])
    del marcado
    del antecessor
```

antecessor[i] é uma posição do vetor que armazena por qual vértice exatamente o vértice **i** foi alcançado

marcado[i] é uma posição do vetor que armazena se o vértice **i** já foi visitado na busca.

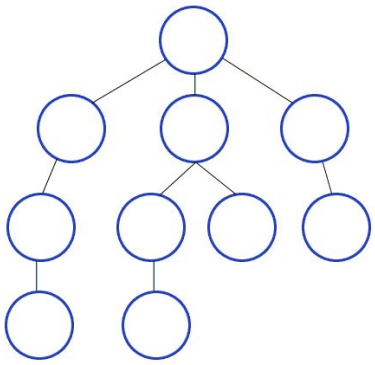
g.V retorna a quantidade de vértices do grafo **g**.

Busca em profundidade (Depth-First Search – DFS)

```
def dfs(g, v, antecessor, marcado):
    marcado[v] = True
    for u in g.adj(v):
        if !marcado[u]:
            antecessor[u] = v
            dfs(g,u,antecessor,marcado)
```

g.adj(**v**) retorna os vértices adjacentes ao vértice **v**.

Busca profundidade: exemplo

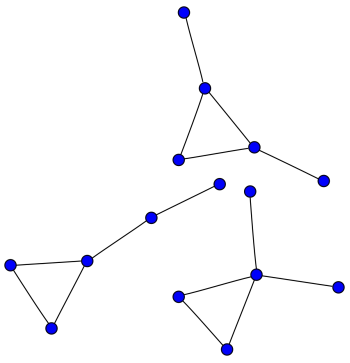


Verificar se grafo é acíclico

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico
- Toda aresta conectando um vértice marcado a um não marcado é chamada de **aresta de árvore**
- Toda aresta conectando um vértice marcado a outro marcado (um antecessor na busca) é chamada de **aresta de retorno**
- ***Arestas de retorno sempre formam ciclos no grafo***

Determinar componentes conexos

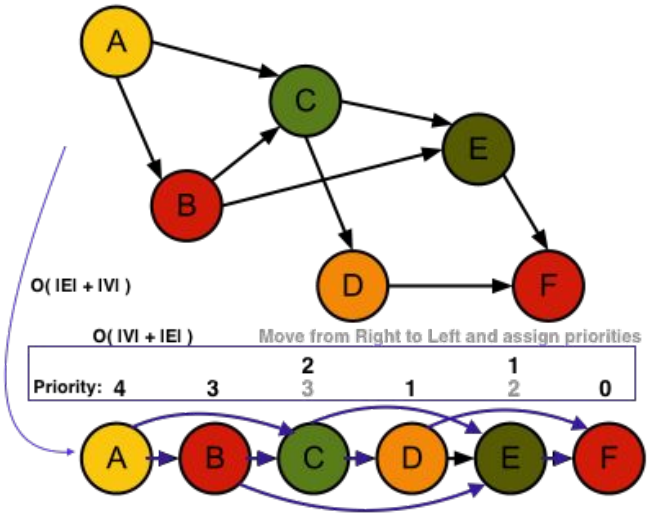
- A busca em profundidade pode ser usada para determinar a **quantidade** e **formação** dos **componentes conexos** do grafo
- **Todos os vértices num mesmo componente conexo são visitados durante uma chamada recursiva da função *dfs* a partir de um vértice de origem**
- Usar um vetor adicional para armazenar o identificador do componente conexo
- Incrementar o contador de componentes a cada iteração do loop na função buscaProfundidade



Ordenação topológica

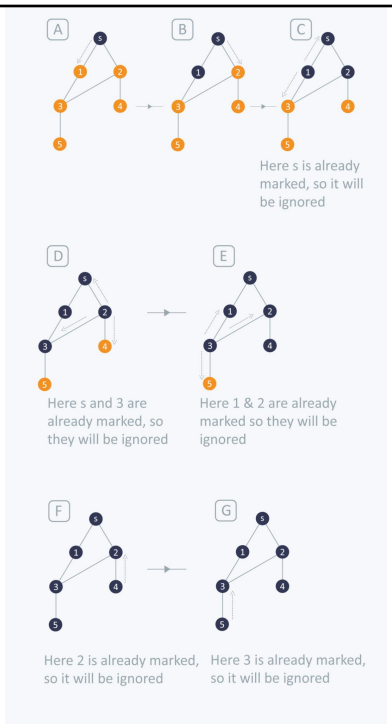
- A **ordenação topológica** de um grafo direcionado acíclico é uma ordenação linear de seus vértices tal que, se existe uma aresta (u,v) no grafo, u aparece antes de v.
- Ordenação topológica é útil na organização de tarefas onde existe interdependência
- A busca em profundidade pode ser usada para encontrar a ordenação topológica de um grafo
- Solução:
  - inserir um vértice na frente de uma lista encadeada sempre que terminar de processá-lo.
  - ao final, a lista contém os vértices na ordem topológica

Ordenação topológica: exemplo



Busca em largura (Breadth-first Search - BFS)

- Livitin chamou essa busca de o caminhamento dos cautelosos
  - Os caminhos mais próximos são visitados antes de se afastar
- A ideia é avaliar todos os vizinhos imediatos para depois explorar os vizinhos dos vizinhos
- Em grafos significa:
  - Iniciar busca na origem
  - Adicionar vértices adjacentes ainda não marcados à lista de vértices a explorar
  - Retirar vértice da lista e marcá-lo
  - Repetir enquanto a lista não estiver vazia



cin.ufpe.br

Busca em largura

```
def bfs(g):
    marcado = g.V*[False];
    antecessor =g.V*[-1];
    vertices = list()  #fila
    for i in range(0,g.V):
        if !marcado[i]:
            vertices.append(i); marcado[i] = True
            while len(vertices) >0:  #visita dos elementos que estao na fila
                v = vertices.pop(0)
                for u in g.adj(v):
                    if !marcado[u]:
                        marcado[u] = True
                        antecessor[u] = v
                        vertices.append(u)
    for i in range(0,g.V): print(antecessor[i])
    del marcado;  del antecessor
```

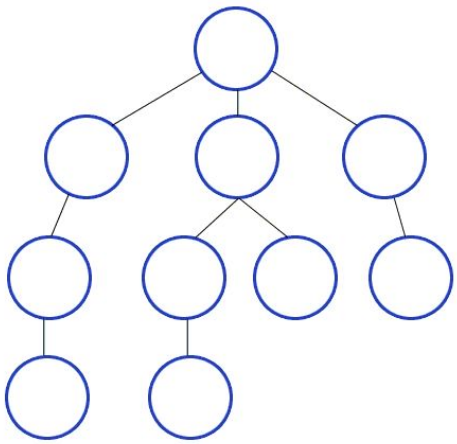
antecessor[i] é uma posição do vetor que armazena por qual vértice exatamente o vértice **i** foi alcançado

marcado[i] é uma posição do vetor que armazena se o vértice **i** já foi visitado na busca.

g.V retorna a quantidade de vértices do grafo **g**.

vertices.pop retorna o primeiro elemento da lista (fila) **vertices**.

Busca em largura: exemplo

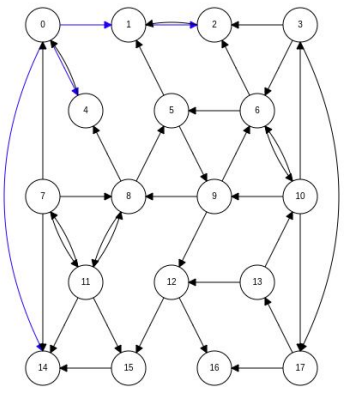


Caminho mais curto a partir de uma origem

- Um efeito colateral da busca em largura é que ela sempre retorna o **caminho mais curto** desde a **origem** a **qualquer outro vértice** no **mesmo componente conexo**
- O caminho mais curto entre a origem e um vértice qualquer pode ser obtido através do **vetor de antecessores**
- Iniciando de **v**, percorre-se seus antecessores até a origem
- Se em algum ponto um antecessor diferente da origem não tiver antecessor, não existe caminho da origem ao vértice

Busca em largura foi configurada para iniciar a partir do vértice 0

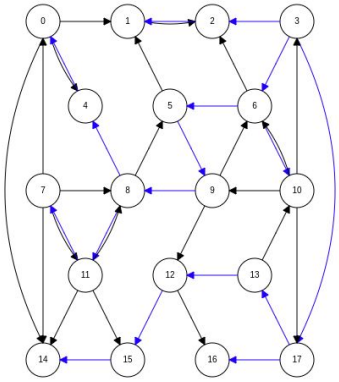
	Parent	Visited
0		f
1	0	T
2	1	T
3		f
4	0	T
5		f
6		f
7		f
8		f
9		f
10		f
11		f
12		f
13		f
14	0	T
15		f
16		f
17		f



Busca em largura foi configurada para iniciar a partir do vértice 3

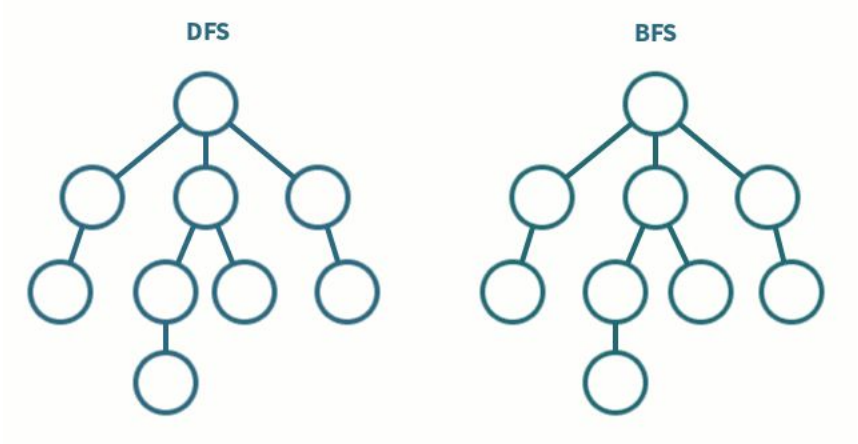
	Parent
0	4
1	2
2	3
3	
4	8
5	6
6	3
7	11
8	9
9	5
10	6
11	8
12	13
13	17
14	15
15	12
16	17
17	3

	Visited
0	T
1	T
2	T
3	f
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T
13	T
14	T
15	T
16	T
17	T



Caminho mais curto a partir de uma origem

```
def caminho(origem, v, antecessor):
    if origem == v: print(v)
    elif antecessor[v] == -1:
      print("nao ha caminho entre origem e destino")
    else:
       caminho(origem, antecessor[v], antecessor)
       print(v)
```



Exercícios

1) Questione ao CHAT-GPT uma aplicação real da busca em profundidade.
2) Questione sobre uma possível implementação que resolva este problema que faça uso de uma linguagem de programação que você tenha domínio.
3) Verifique:
   a) O custo do codigo: qual a complexidade? Poder-se-ia ser escrito de uma forma melhor?
   b) A corretude do código: o código considera todas as situações do problema? Funciona correto para todas as entradas?
4) Repita as questões acima considerando agora a busca em largura.

E se você pedir por mais exemplos de aplicações reais ao CHAT GPT?

Leitura

- Seções 7.3 a 7.6 (Ziviani)
- Seções 3.5 e 4.2 (Livitin)
- Seções 4.1 e 4.2 (SW)

Algoritmos e Estruturas de Dados - IF672

Prof. Dr. Fernando M. de Paula Neto
fernando@cin.ufpe.br
cin.ufpe.br/~fernando

Conteúdo modificado dos slides cedidos pelos professores
Hansenclever Bassani e Renato Vimieiro