SCC0216 - Modelagem Computacional em Grafos

# Travessias

## 2014

Profa. Cristina (cristina@icmc.usp.br )
Profa. Rosane (rminghim@icmc.usp.br)
PAE: Bilzã (bmarques@icmc.usp.br) / Rafael (rmartins@icmc.usp.br) / Jorge Henrique (jorgehpo@gmail.com)

Baseado no material de aula original: Profª. Josiane M. Bueno e de outros docentes e assistentes do ICMC.

## Percorrendo um grafo

Percorrendo um Grafo

- Percorrer um grafo é um problema fundamental
- Deve-se ter uma forma sistemática de visitar as arestas e os vértices
- O algoritmo deve ser suficientemente flexível para adequar-se à diversidade de grafos



## Eficiência

Percorrendo um Grafo

- Eficiência
  - Não deve haver repetições (desnecessárias) de visitas a um vértice e/ou aresta (apenas duas visitas a cada aresta)



## Correção

Percorrendo um Grafo

- Correção
  - Todas os vértices e/ou arestas devem ser visitados



## Solução

Percorrendo um Grafo

- Solução
  - Marcar os vértices com...
    - não visitados
    - visitados
    - processados



## Solução

Percorrendo um Grafo

- Solução
  - Manter uma lista de vértices no estado '*visitados*'
  - Há duas possibilidades de implementação:
    - Fila
    - Pilha

# Travessia em Largura – BFS

Percorrendo um Grafo

- *BFS – Breadth-First Search*
  - Em grafos não-dirigidos cada aresta é visitada duas vezes
  - Em grafos dirigidos cada aresta é visitada uma única vez



# BFS

```
{ Percorre um grafo G a partir de um vértice
  inicial s informado. Pode realizar processa-
  mento à medida que visita vértices e arestas }

"Descobre" todos os vértices alcançáveis a partir
de s;
Calcula a distância de s a cada vértice alcançável;
Gera uma árvore em largura com raiz em s com todos
os vértices alcancáveis v, tal que o caminho na
árvore corresponde ao menor caminho entre s e v.
```

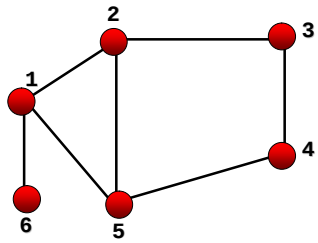


```
BFS (G,s)
 for each vertex u ∈ V[G] – {s} do
     color[u] = "WHITE"
          d[u] = INF
     p[u] = NIL
 end-for
 color[s]= GRAY, d[s] = 0, p[s] = NIL
 initialize(Q)
 enqueue(Q,s)
 while (not empty(Q)) do
   u = dequeue[Q]
   processe o vértice u conforme desejado
   for each v ∈ Adj[u] do
      processe a aresta (u,v) conforme desejado
      if color[v] = "WHITE" then
         color[v] = "GRAY"
         d[v] = d[u] + 1
         p[v] = u
         enqueue(Q,v)
      end-if
   end-for
   color[u] = "BLACK"
 end-while
```



# BFS – exemplo

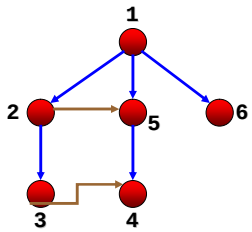
Percorrendo um Grafo: BFS






# BFS Tree

Percorrendo um Grafo: BFS Tree






# Complexidade do BFS

$O(|V| + |E|)$, ou seja, linear em relação ao tamanho da representação de $G$ por lista de adjacências:

- Todos os vértices são empilhados/desempilhados no máximo uma vez. O custo de cada uma dessas operações é $O(1)$, e elas são executadas $O(|V|)$ vezes.
- A lista de adjacências de cada vértice é percorrida no máximo uma vez (quando o vértice é desempilhado). O tempo total é $O(|E|)$ (soma dos comprimentos de todas as listas, igual ao número de arestas)
- Inicialização é $O(|V|)$

# Travessia em Profundidade – DFS

Percorrendo um Grafo

- *DFS – Depth First Search*
  - Recursivo, eliminando assim a necessidade de uma estrutura de lista (fila ou pilha)



# DFS

```
{ Percorre um grafo G. Pode realizar processa-
  mento à medida que visita vértices e arestas }

DFS-graph (G)
   for each vertex u ∈ V[G] do
       color[u] = "WHITE"
            p[u] = NIL
   end-for
   time = 0
   for each vertex u ∈ V[G] do
      if color[u] = "WHITE" then
         inicialize um novo componente
               DFS-visit(u)
      end-if
   end-for
```



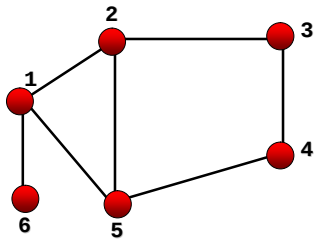
# DFS

```
DFS-visit(u)
   color[u] = "GRAY"
   time = time + 1
   d[u] = time
   processe o vértice u conforme desejado
   for each v ∈ Adj[u] do
        processe a aresta (u,v) conforme desejado
        if color[v] = "WHITE" then
           p[v] = u
           DFS-visit(v)
        end-if
   end-for
   color[u] = "BLACK"
   f[u] = time = time + 1
```



# DFS

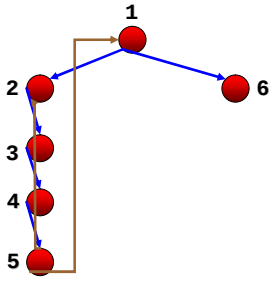
Percorrendo um Grafo: DFS






# DFS Tree

Percorrendo um Grafo: DFS Tree






# Complexidade do DFS

$O(|V| + |E|)$

- No algoritmo principal, cada for é $O(|V|)$. O *DFS-visit* é chamado exatamente uma vez para cada vértice de $|V|$ (na pior das hipóteses)
- No *DFS-visit*, o laço é executado | *adj*[*v*] | vezes, i.e., $O(|E|)$ no total

## DFS

- Uma aplicação clássica do DFS consiste em decompor um grafo direcionado (dígrafo) em componentes fortemente conexos.
- Um grafo direcionado é fortemente conexo se quaisquer dois vértices são mutuamente alcançáveis entre si.
- Um componente fortemente conexo de um grafo é um sub-conjunto maximal *C* de vértices de *V* tal que qualquer par de vértices de *C* é mutuamente alcançável.
- Algoritmo no Cormen, p. 554; ver também no Ziviani (slides de aula 2014 - 05)



## Tarefas

1. Escrever uma versão não-recursiva do DFS
   - Dica: todo algoritmo recursivo usa uma pilha implicitamente
2. Escreva um algoritmo que verifique se um dado grafo G(V,E) é acíclico.
   - Dica: a solução é uma aplicação do algoritmo DFS. Se na busca em profundidade é encontrada uma aresta $(u,v) \in E$ conectando um vértice *u* com um seu antecessor *v* na árvore de busca em profundidade, então o grafo tem ciclo. Igualmente, se *G* tem ciclo uma aresta desse tipo será encontrada em qualquer busca em profundidade em *G*.



## Caminhos mais curtos

- Em grafos não-orientados e não-valorados o algoritmo BFS(***u***) produz uma *árvore* de caminhos mais curtos entre ***u*** (origem) e todos os demais vértices do grafo alcançáveis a partir dele.
- Assim, o vetor antecessor [ ] é capaz de fornecer o caminho mais curto (menor número de arestas) entre *u* e *v*, para qualquer *v* em *V*, se ele existir.



## Caminhos mais curtos (algoritmo)

- Dado o vetor antecessor após BFS(*v*):

```
Imprimir_caminho_mais_curto(origem, v:tipoVértice)
   Se origem = v escreve (origem)
     senão
        Imprimir_caminho_mais_curto(origem, antecessor(v))
        escreve(v)
      fim se
Fim Imprimir_caminho_mais_curto
```

Obs: 'escreve' pode ser qualquer procedimento de armazenamento ou impressão do caminho.