

Algoritmos e Estruturas de Dados II

Grafos III: Busca em Largura

Ricardo J. G. B. Campello

Parte deste material é baseado em adaptações e extensões de slides disponíveis em <http://ww3.datastructures.net> (Goodrich & Tamassia).

1

Organização

- ◆ Definição e Motivação
- ◆ Algoritmo BFS
 - Pseudo-código
 - Implementação simples em C
- ◆ Exemplo de Execução BFS
- ◆ Propriedades do Percurso BFS
- ◆ Algumas Aplicações de BFS
 - Caminhos com número mínimo de arestas
 - Busca de ciclos
- ◆ Análise de Complexidade do Algoritmo BFS

2

Busca em Largura

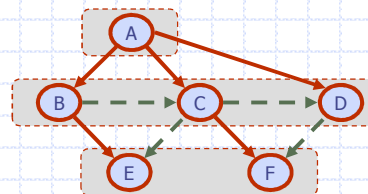
- ◆ Busca em Largura (BFS) é uma estratégia geral de caminhamento em grafos
- ◆ BFS em um grafo G :
 - Visita todos os vértices e arestas de G
 - Descobre os componentes conexos de G
 - ◆ logo, se G é conexo ou não
 - Para percorrer todo um grafo não conexo, BFS deve ser executada múltiplas vezes, sempre a partir de um vértice não visitado nas anteriores
- ◆ BFS pode ser estendida para resolver outros problemas em grafos:
 - Encontrar um caminho entre um dado par de vértices, *com a menor quantidade de arestas*, caso exista
 - Encontrar um ciclo simples, caso exista
 - Encontrar uma floresta geradora de G
 - ◆ árvore geradora p/ G conexo

3

Busca em Largura

- ◆ Um algoritmo do tipo BFS usa marcadores para direcionar o percurso (caminhamento)
- ◆ A forma mais geral é marcar ambos vértices e arestas
- ◆ As arestas são marcadas como de **descoberta** ou **cruzamento**
 - Arestas de descoberta levam a vértices não descobertos
 - ◆ são também chamadas arestas de árvore (*tree edges*)
 - Arestas de cruzamento são as demais arestas

- ◆ Alternativamente, pode-se marcar apenas os vértices, como **não descobertos**, **descobertos** e **explorados**
 - Adota-se aqui esta abordagem



4

Algoritmo BFS

Algoritmo *BFS*(*G*, *v*)

Q ← nova fila vazia

enqueue(*Q*, *v*)

v.label ← *DESCOBERTO*

enquanto ¬ *empty*(*Q*)

v ← *dequeue*(*Q*)

process_vertex(*v*)

v.label ← *EXPLORADO*

para todo *e* ∈ *incidentEdges*(*G*, *v*)

y ← *opposite*(*G*, *v*, *e*)

se *y.label* = *NÃO-DESCOBERTO*

enqueue(*Q*, *y*)

y.label ← *DESCOBERTO*

y.parent ← *v*

se ¬ (*y.label* = *EXPLORADO*)

process_edge(*e*)

Garante que cada aresta será processada apenas uma vez, não duas (uma para cada vértice adjacente).

Assume-se que inicialmente os vértices de *G* são rotulados como “não-descobertos”.

5

Implementação C (Skiena & Revilla, 2003)

```
bfs(graph *g, int v) {
    queue q; /* queue of vertices to visit */
    int y; /* adjacent vertex */
    int i; /* counter */
    init_queue(&q);
    enqueue(&q, v);
    discovered[v] = TRUE;
    while (empty(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex(v);
        processed[v] = TRUE;
        for (i=0; i<g->degree[v]; i++) {
            y = g->edges[v][i];
            if (discovered[y] == FALSE) {
                enqueue(&q, y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            if (processed[y] == FALSE) process_edge(v, y);
        }
    }
    /* Assume-se inicialização tal que parent[i] = 0 e
    processed[i] = discovered[i] = FALSE i=0,..., MAXV */
}
```

```
bool processed[MAXV];
bool discovered[MAXV];
int parent[MAXV];
```

```
/* graph.h */
.....
typedef struct {
    tipo_elem vertices[MAXV+1];
    int edges[MAXV+1][MAXDEGREE];
    int degree[MAXV+1];
    int nvertices;
    int nedges;
} graph;
```

6

Implementação C (Skiena & Revilla, 2003)

```
process_vertex(int v)
{
    printf("processed vertex %d\n", v);
}

process_edge(int x, int y)
{
    printf("processed edge (%d,%d)\n", x, y);
}
```

Realização para
imprimir cada
vértice e aresta uma
única vez (quando
explorado)







Exercício:

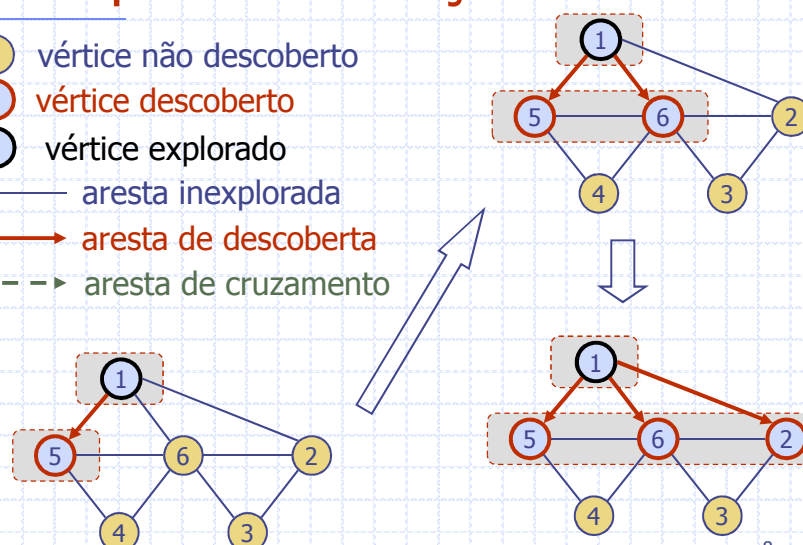
- Modifique as duas rotinas acima para que BFS imprima apenas o conteúdo armazenado nos vértices. Para tanto, assumo que o tipo dos elementos de vertices seja:

```
typedef char tipo_elem[30]; /* string */
```

7

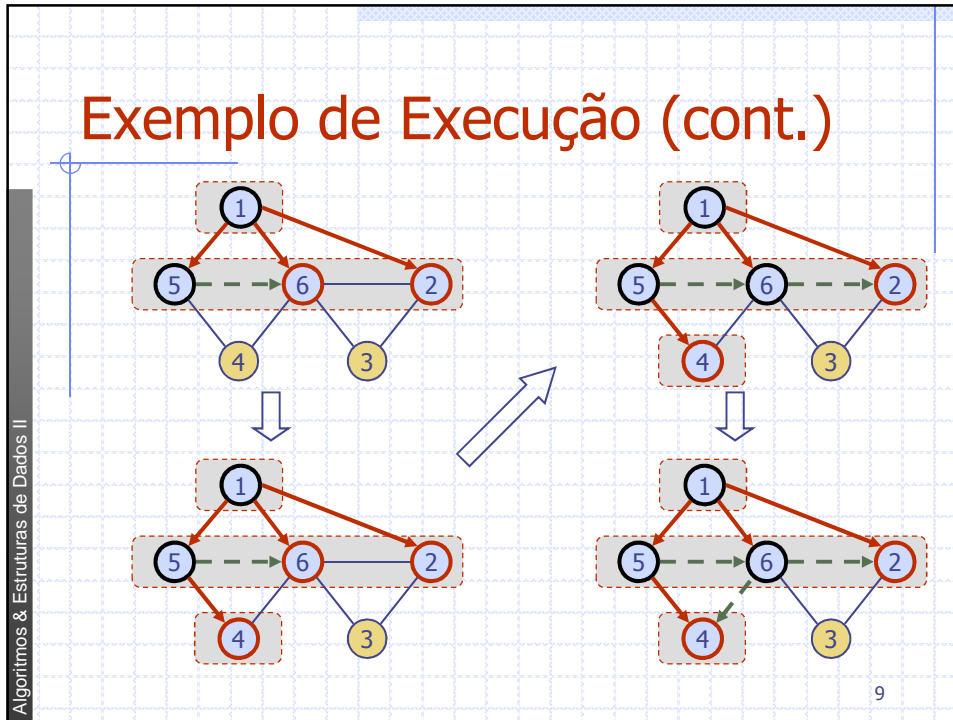
Exemplo de Execução

-  vértice não descoberto
-  vértice descoberto
-  vértice explorado
-  aresta inexplorada
-  aresta de descoberta
-  aresta de cruzamento

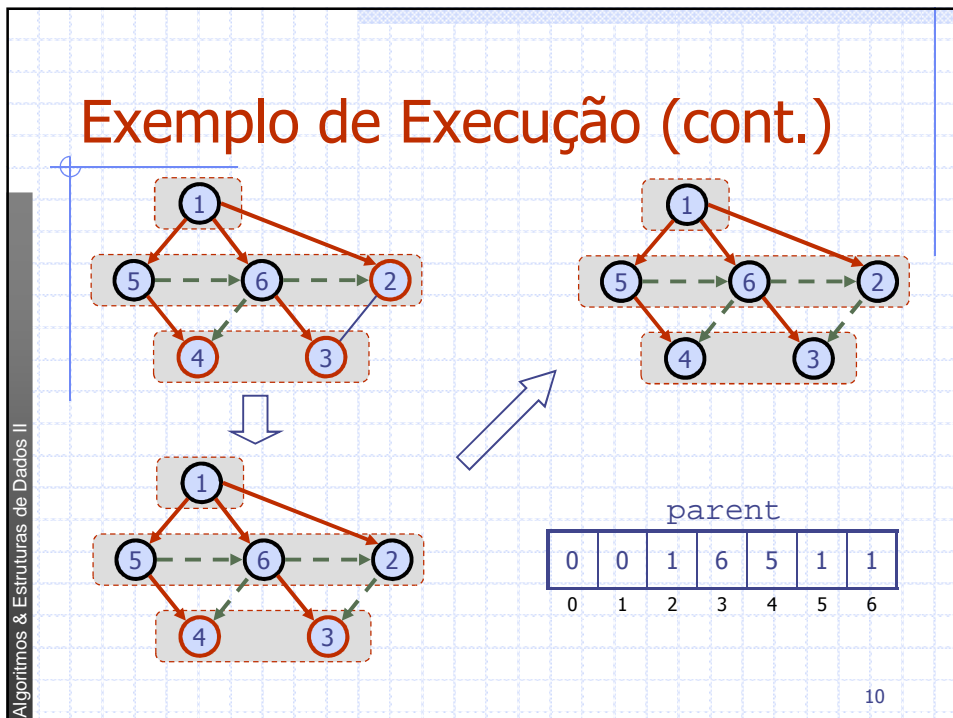


8

Exemplo de Execução (cont.)



Exemplo de Execução (cont.)



Propriedades

Notação:

G_v : componente conexo que contém v .

Propriedade 1:

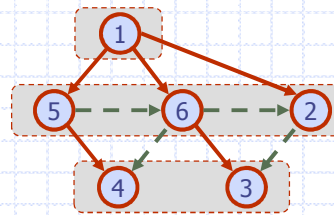
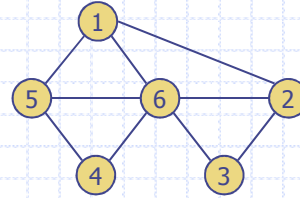
$BFS(G, v)$ explora todos os vértices e arestas de G_v .

Propriedade 2:

As arestas de descoberta formam uma árvore geradora de G_v .

Propriedade 3:

Qualquer vértice $s \neq v$ a um no. mínimo de i arestas de v será descoberto por um vértice a um no. mínimo de $i - 1$ arestas de v .



11

Caminhos Mínimos

- ◆ A propriedade 3 garante que a sucessão de descobertas produz caminhos com no. mínimo de arestas da origem v até $\forall s \neq v$.

- ◆ Nota: Caminhos não são únicos.

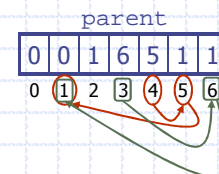
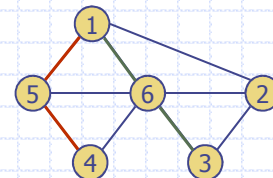
- ◆ Idéia de Algoritmo Recursivo:

- Caminho mínimo entre o vértice v de início da busca e um vértice s qualquer é dado por:

caminho mínimo entre v e o antecessor (pai) de s na sucessão de descobertas

+

caminho entre seu antecessor e s via aresta incidente aos dois



12

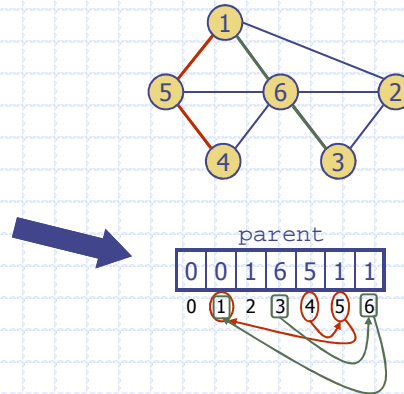
Caminhos Mínimos

- ◆ A propriedade 3 garante que a sucessão de descobertas produz caminhos com no. mínimo de arestas da origem v até $\forall s \neq v$.
- ◆ Nota: Caminhos não são únicos.

Algoritmo Recursivo:

```
find_path(int start, int end, int parent[]) {
    if (start == end)
        printf("\n%d", start);
    else {
        find_path(start, parent[end], parent);
        printf(" %d", end);
    }
}

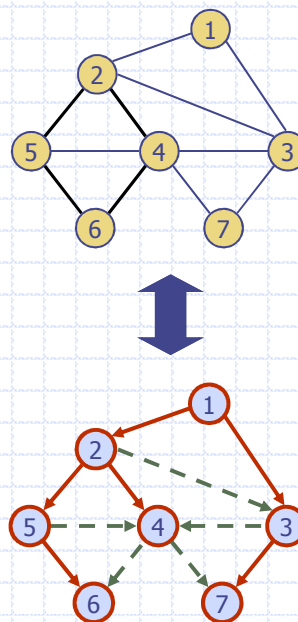
/* start deve ser o vértice inicial de bfs */
```



13

Ciclos

- ◆ Como as arestas de descoberta formam uma árvore geradora, qualquer aresta de cruzamento fecha um ciclo.
- ◆ O ciclo é dado pela aresta de cruzamento e pelos caminhos a partir dos seus vértices finais até um antecessor comum ao longo dos parentescos de descobertas.
- ◆ Exemplos:
 - aresta (4,6):
 - ◆ antecessor comum: vértice 2
 - aresta (7,4):
 - ◆ antecessor comum: vértice 1



14

Análise

- ◆ A rotulação de um vértice leva tempo $O(1)$.
 - Cada um dos n vértices é rotulado três vezes:
 - NÃO DESCOBERTO – DESCOBERTO – EXPLORADO $\Rightarrow O(n)$.
- ◆ Inserir ou remover um vértice da fila leva tempo $O(1)$.
 - Cada vértice é inserido e removido uma vez da fila $Q \Rightarrow O(n)$.
- ◆ Para cada vértice, cada uma de suas arestas incidentes e o respectivo vértice adjacente é verificado:
 - Para um vértice v tem-se que isso leva tempo $O(\deg(v))^*$.
 - Logo, lembrando que $\sum_{v \in G} \deg(v) = 2m$, tem-se $O(m)$ no total.
- ◆ Portanto, assumindo que **process_vertex**(v) e **process_edge**(v) executam em tempo $O(1)$, tem-se que BFS executa em tempo $O(n+m)$.

* PS. Esse tempo é válido para as implementações de grafos em lista de adjacências e estrutura alternativa.

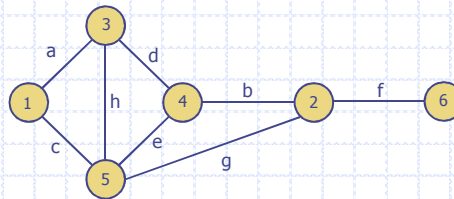
Exercícios

1. Na implementação de grafos em C discutida na aula de EDs para grafos, vimos que as rotinas de inserção e remoção de arestas já atualizam o contador `nedges` de forma apropriada, assim como devem fazer as rotinas de inserção e remoção de vértices com relação ao contador `nvertices`. Suponha, apenas hipoteticamente, que essas rotinas não disponham desse mecanismo de atualização simples e implemente as rotinas `process_vertex` e `process_edge` para que a execução de `bfs` atualize esses contadores.
2. O **teorema das quatro cores** declara que qualquer mapa planar pode ser colorido utilizando apenas quatro cores sem que qualquer região seja colorida com a mesma cor de uma região vizinha. Após permanecer aberto como uma conjectura por mais de 100 anos, esse teorema foi provado em 1976 com a ajuda de um computador. Um problema mais simples que este é determinar se um grafo qualquer que seja não direcionado, simples e conexo pode ser bicolorido, isto é, ter seus vértices pintados com duas cores sem que vértices adjacentes sejam pintados com uma mesma cor. Explique como resolver esse problema com BFS e modifique o pseudo-código BFS para tal.

16

Exercícios

3. Ilustre graficamente, conforme o exemplo dado em aula, as execuções passo a passo do algoritmo BFS com início em cada um dos vértices do grafo abaixo. Nota: Destaque, em cada execução, quais as arestas de cruzamento, as arestas de descoberta, a árvore geradora resultante e a relação (vetor) de parentesco entre os vértices:



4. Elabore outros grafos e repita o Exercício 3 para exercitar a busca BFS.

17

Exercícios

5. Considere que a matriz abaixo represente as conexões aéreas bilaterais existentes (x) ou não (em branco) entre um dado conjunto de 16 cidades. Aplique BFS para descobrir as rotas com o menor número de conexões ligando a cidade 1 às demais cidades e apresente as 15 rotas. Dica: Visualize o grafo como um grid 4 x 4, sendo a cidade 1 na posição (1,1), a cidade 2 na posição (1,2), etc, até a cidade 16 na posição (4,4).

1		x			x	x									
2	x		x			x									
3		x		x			x								
			x				x	x							
	x					x			x						
	x	x			x				x						
			x	x						x	x	x			
...				x							x				
					x	x				x				x	x
							x		x		x				
							x		x					x	x
								x							x
									x						
										x					
											x				
												x			
16													x		

18

Referências

- ◆ M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in C++/Java*, John Wiley & Sons, 2002/2005.
- ◆ N. Ziviani, *Projeto de Algoritmos*, Thomson, 2a. Edição, 2004.
- ◆ T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 2nd Edition, 2001.
- ◆ S. Skiena e M. Revilla, *Programming Challenges: The Programming Contest Training Manual*, Springer-Verlag, 2003.