Algoritmos e Estruturas de Dados II

Grafos IV:

Busca em Profundidade

Ricardo J. G. B. Campello

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

Organização

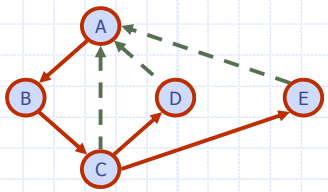
- Definição e Motivação
- Algoritmo DFS
  - Pseudo-código
  - Análise de Complexidade
  - Implementação simples em C
- Exemplo de Execução DFS
- Propriedades do Percurso DFS
- Busca por Ciclos via DFS

Busca em Profundidade

- Busca em Profundidade (DFS) é uma estratégia geral de caminhamento em grafos
- DFS 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, DFS deve ser executada múltiplas vezes, sempre a partir de um vértice não visitado nas anteriores
- DFS pode ser estendida para resolver outros problemas em grafos:
  - Encontrar um caminho entre um dado par de vértices, caso exista
  - Encontrar um ciclo simples, caso exista
  - Encontrar uma floresta geradora de G
    - árvore geradora para G conexo
  - Encontrar uma ordenação topológica em um digrafo acíclico, caso exista

Busca em Profundidade

- Um algoritmo do tipo DFS 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 **retorno**
  - Arestas de descoberta levam a vértices não descobertos
    - são também chamadas arestas de árvore (*tree edges*)
  - As demais são de retorno:
    - levam a vértices antecessores no parentesco de descobrimento
- Alternativamente, pode-se marcar apenas os vértices, como **não descobertos**, **descobertos** e **explorados**
  - Adota-se aqui esta abordagem



Algoritmo DFS

Algoritmo *DFS(G, v)*
- *v.label* ← *DESCOBERTO*
- *process_vertex*(*v*)
- **para todo** *e* ∈ *incidentEdges*(*G*, *v*)
  - *y* ← *opposite*(*G*, *v*, *e*)
  - **se** *y.label* = *NÃO-DESCOBERTO*
    - *y.parent* ← *v*
    - *DFS*(*G*, *y*)
  - **senão**
    - **se** ¬ (*y.label* = *EXPLORADO*)
      - *process_edge*(*e*)
- *v.label* ← *EXPLORADO*

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

Garante que cada aresta não será processada duas vezes.

Em particular, cada aresta (*v*,*y*) só será processada se *y* tiver sido **descoberto**, mas ainda **não explorado**, ou seja:

- Na 2ª avaliação desta aresta

ou

- Quando ciclo é encontrado

- Tempo $O(n+m)$: lista de adjacências ou estrutura alternativa

Implementação C (Skiena & Revilla, 2003)

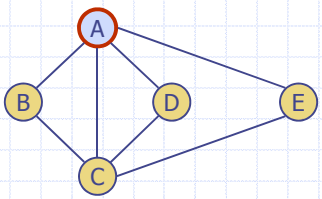
```
bool processed[MAXV];
bool discovered[MAXV];
int parent[MAXV];
```

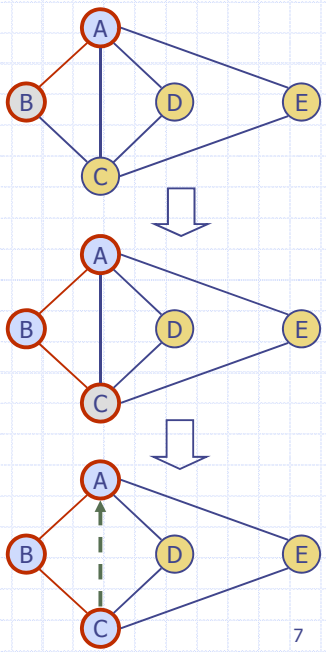
```
dfs(graph *g, int v) {

    int i;                  /* counter */
    int y;                  /* successor vert

    discovered[v] = TRUE;
    process_vertex(v);

    for (i=0; i<g->degree[v]; i++) {
        y = g->edges[v][i];
        if (discovered[y] == FALSE) {
            parent[y] = v;
            dfs(g,y);
        } else if (processed[y] == FALSE) process_edge(v,y);
        if (finished) return;  /* allow for search termination */
    }
    processed[v] = TRUE;
}
```

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

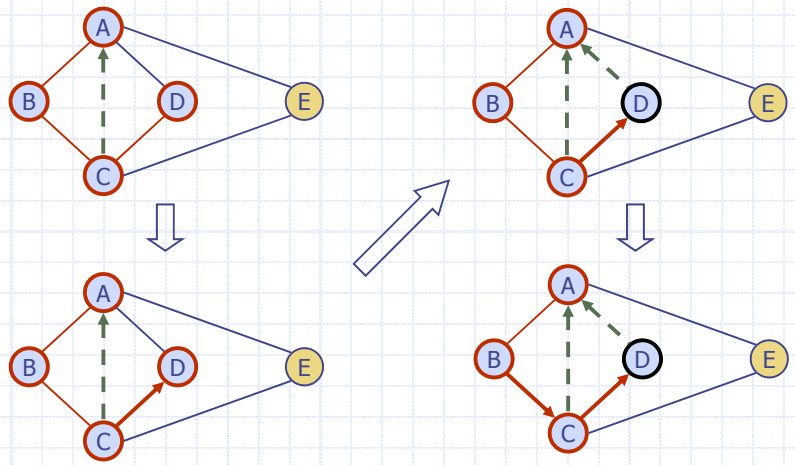

Exemplo

- vértice não descoberto
- vértice descoberto
- vértice explorado
- arestas não processadas
- aresta de descoberta
- aresta de retorno

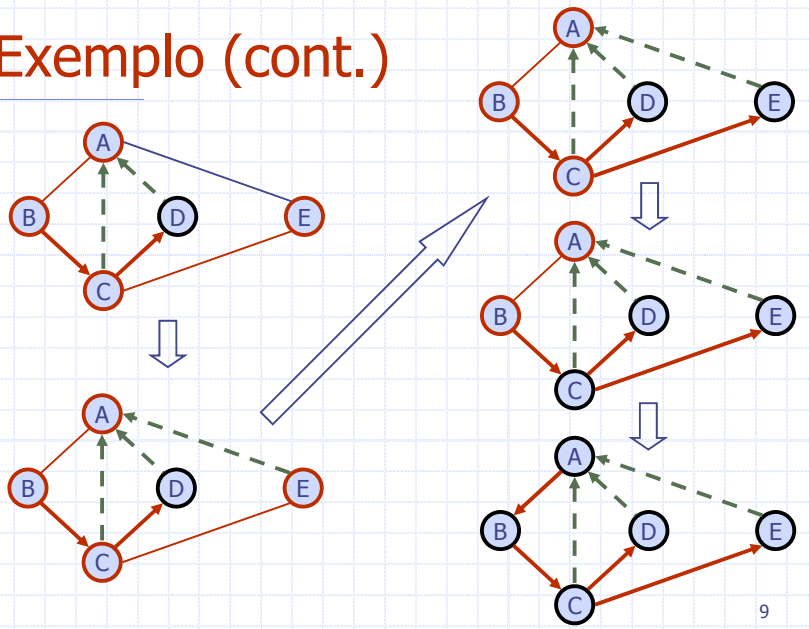




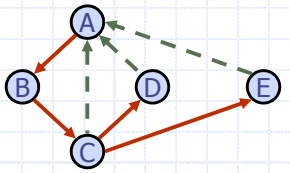
Exemplo (cont.)



Exemplo (cont.)



Propriedades



Propriedade 1:

DFS(G,v) explora todos os vértices e arestas de um componente conexo de ***v***:

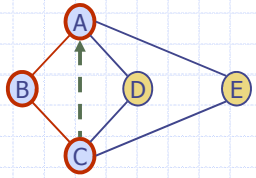
- Por construção, todos os vértices adjacentes a um vértice explorado e as arestas correspondentes serão necessariamente explorados. Se o vértice (aresta) está na parte conexa do grafo, então é necessariamente adjacente (incidente) a um vértice explorado ou a um vértice adjacente a um vértice explorado, ou assim por diante, e portanto será necessariamente explorado.

Propriedade 2:

As arestas de descoberta de ***DFS(G,v)*** formam uma árvore geradora do componente conexo de ***v***:

- Uma aresta nunca é considerada como de descoberta se conecta dois vértices já descobertos, portanto um ciclo nunca é formado com esse tipo de aresta.

Busca por Ciclos

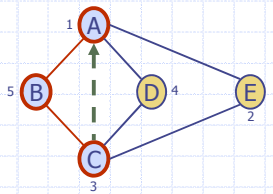


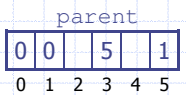
- Vimos que cada aresta (v,y) só é processada:
  - Na 2ª avaliação desta aresta, **ou**
  - Quando ciclo é encontrado (aresta de retorno)
- Isso sugere as seguintes especializações de DFS para buscar ciclos:

```
process_vertex(int v) {
}

process_edge(int x, int y) {
    if (parent[x] != y) {    /* found back edge! */
        printf("Cycle from %d to %d:",y,x);
        find_path(y,x,parent);
        printf("\n\n");
        finished = TRUE;
    }
}
```

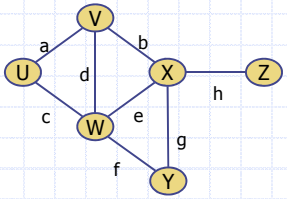

Busca por Ciclos (cont.)

```
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);
    }
}
```





Exercícios



1. Ilustre graficamente, conforme o exemplo dado em aula, as execuções passo a passo do algoritmo DFS com início em cada um dos vértices do grafo acima. Nota: Destaque, em cada execução, quais as arestas de cruzamento, as arestas de retorno, a árvore geradora resultante e a relação (vetor) de parentesco entre os vértices:
2. Elabore outros grafos e repita o Exercício 1 para exercitar a busca DFS.
3. Explique porque a complexidade $O(n+m)$ do tempo de execução de DFS não é válida para grafos implementados com estrutura de dados matriz de adjacências. Analise a complexidade nesse caso.
4. Reescreva o pseudo-código da busca DFS de forma não recursiva. Utilize uma pilha para eliminar a recursividade.

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.