DISCIPLINA: ESTRUTURA DE DADOS II

2018

UEA
UNIVERSIDADE DO ESTADO DO AMAZONAS

Busca em Grafos Implementação

Prof. Luis Cuevas Rodríguez, PhD

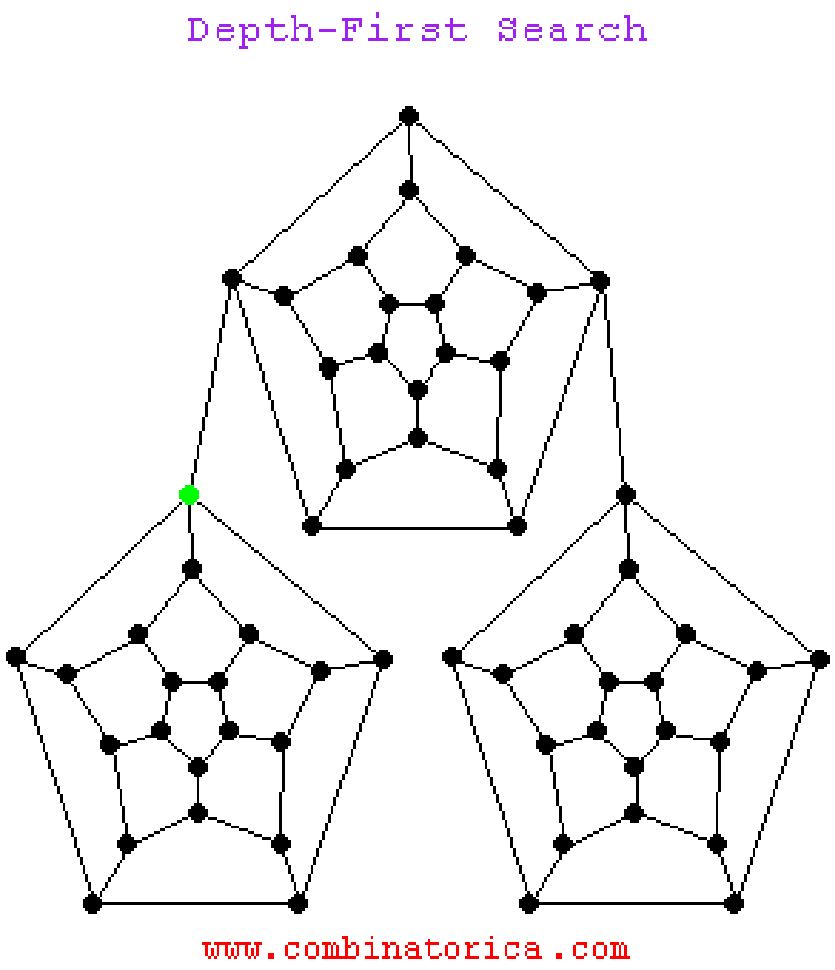
EST
Escola Superior de Tecnologia da UEA

NUCOMP
NÚCLEO DE COMPUTAÇÃO

Conteúdo

- Busca em profundidade
- Busca em largura

Busca em profundidade (DFS)



TDA necessários

- O grafo: lista de adjacência ou matriz de adjacência.
- Uma PILHA: armazenar o caminho atual.
- Um vetor: armazenar os no visitados.

Prof. Luis Cuevas Rodríguez, PhD

UEA
UNIVERSIDADE DO ESTADO DO AMAZONAS

Lista de adjacência - TDA

```
typedef struct adjacencia {
int vertice;
int peso;
struct adjacencia *prox;
}ADJACENCIA;

typedef struct vertice {
ADJACENCIA *cad;
}VERTICE;

typedef struct grafo{
int vertices;
int arestas;
VERTICE *vert;
}GRAFO;
```

```
GRAFO* criaGrafo(int v);
bool criaAresta(GRAFO* gr, int vi, int vf, TIPOPESO p);
void imprime(GRAFO* gr);
```

DFS – Algoritmo Recursivo

método DFS (source):
    marcar a origem como visitado
    para cada vértice v adjacente à origem no gráfico:
        se v não foi visitado:
        marcar v como visitado
        chamar recursivamente DFS (v)

DFS – Algoritmo Recursivo

```
void dfs_visitar(GRAFO* gr, int no, bool v[]){
     v[no]= true;
     cout << "No " << no << " visitado" << endl;
     ADJACENCIA* ad=gr->vert[no].cad;
     while (ad){
          if (v[ad->vertice]==false)
               dfs_visitar(gr, ad->vertice, v);
          ad = ad->prox;
     }
     cout << "Terminou com No " << no << endl;
}
```

Algoritmo - Iterativo

```
método DFS (origem):
    criar uma Pilha S
    adicionar origem à Pilha S
    marcar a origem como visitada
    enquanto S não está vazio:
        remover um elemento v, da pilha S
        para cada vértice w adjacente a v no gráfo:
            se w não foi visitado:
                marcar w como visitado
                insir w na pilha S
```

DFS – Algoritmo Iterativo

- Usar uma Pilha

```
typedef struct aux{
    int vert;
    aux* proximo;
}ELEMENTO;

typedef struct {
    ELEMENTO* topo;
}PILHA;
```

```
void inicializaPila(PILHA* p);
int tamanho (PILHA* p);
bool inserirVert(PILHA* p, int v);
ELEMENTO pop_pilha (PILHA* p);
```

DFS – Algoritmo Iterativo

```
void dfs_vist_iterativo(GRAFO* gr, int no, bool v[]){
    PILHA p;
    int v_atual;
    int abj[gr->vertices];
    inicializaPila(&p);
    inserirVert(&p,no);
    v[no]= true;
    cout << "No " << no << " visitado" << endl;
    while (tamanho(&p)!=0){
            v_atual=pop_pilha(&p).vert;
            ADJACENCIA* ad=gr->vert[v_atual].cad;
            while (ad){
                if (v[ad->vertice]==false){
                    v[ad->vertice]= true;
                    cout << "No " << ad->vertice << " visitado" << endl;
                    inserirVert(&p,ad->vertice);
                }
                ad = ad->prox;
            }
    }
}
```

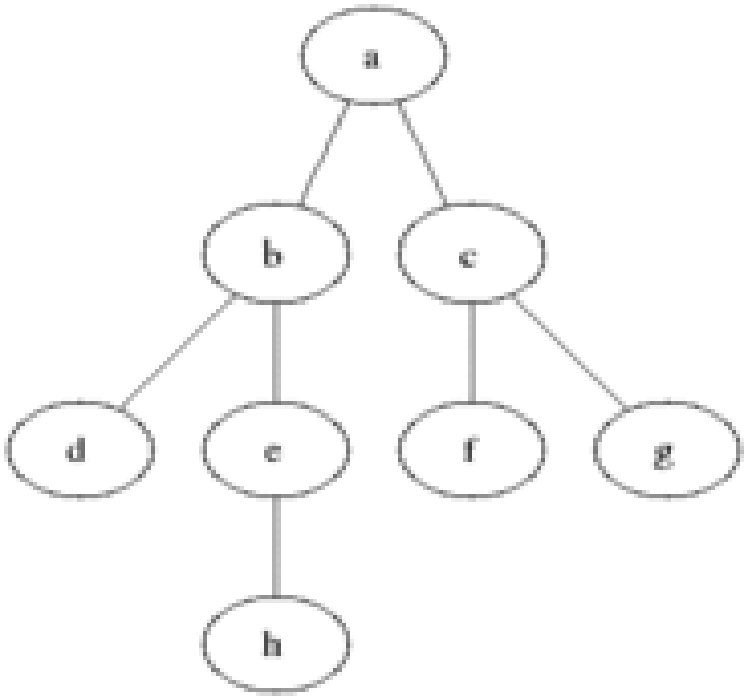
DFS – Exemplo

```
int main()
{
    int tam = 7;
    int arvores =0;
    GRAFO* gr = criaGrafo(tam);
    criaAresta(gr,0,1,20);
    criaAresta(gr,1,2,40);
    criaAresta(gr,2,0,12);
    criaAresta(gr,2,4,40);
    criaAresta(gr,3,1,30);
    criaAresta(gr,4,3,80);
    criaAresta(gr,1,5,80);
    imprime(gr);

    bool visitados[tam];
    for (int i= 0; i<tam; i++)
        visitados[i]=false;

    for (int i = 0; i<tam; i++){
        if (visitados[i]==false){
            //dfs_visitar(gr,i,visitados); //recursivo
            dfs_vist_iterativo(gr,i,visitados); //iterativo
            arvores++;
        }
        i++;
    }

    cout << "Arvores: " << arvores << endl;
    return 0;
}
```

Exercício

- Adaptar os códigos para um grafo não dirigido.
- Adaptar os códigos para representar o grafo usando matriz de adjacência.

Busca em largura (BFS)



TDA necessários

- O grafo: lista de adjacência ou matriz de adjacência.
- Uma FILA: armazenar no descobertos.
- Um vetor: armazenar os no visitados.

BFS- Algoritmo

```
Método BFS (Graph, origem):
    criamos uma Fila Q
    adicionar origem à Fila Q
    marcar a origem como visitada
    enquanto Q não está vazio:
        tirar um elemento da Fila Q chamado v
        para cada vértice w adjacente a v no gráfico:
            se w não foi visitado:
                marcar w como visitado
                inserir w à Fila Q
```

BFS

- Usar uma Fila

```
typedef struct aux{
    int vertice;
    aux* proximo;
}ELEMENTO;


typedef struct {
    ELEMENTO* inicio;
    ELEMENTO* fim;
}FILA;
```

```
void inicializaFila(FILA* f);
int tamanho (FILA* f);
void reinicializar(FILA* f);
ELEMENTO returnVertice (FILA* p);
bool inserirVertice(FILA* f, int vert);
```

Busca em largura (BFS)

```
void bfs_vist(GRAFO* gr, int no, bool v[]){
     FILA f;
     int v_atual;
     inicializaFila(&f);
     inserirVertice(&f,no);
     v[no]= true;
     cout << "No " << no << " visitado" << endl;
     while (tamanho(&f)!=0){
              v_atual=returnVertice(&f).vertice;
              ADJACENCIA* ad=gr->vert[v_atual].cad;
              while (ad){
                   if (v[ad->vertice]==false){
                        v[ad->vertice]= true;
                        cout << "No " << ad->vertice << " visitado" << endl;
                        inserirVertice(&f,ad->vertice);
                        }
                   ad = ad->prox;
              }
     }
}
```

Exemplo

```
int main()
{
    int tam = 7;
    int arvores =0;
    GRAFO* gr = criaGrafo(tam);
    criaAresta(gr,0,1,20);
    criaAresta(gr,1,2,40);
    criaAresta(gr,2,0,12);
    criaAresta(gr,2,4,40);
    criaAresta(gr,3,1,30);
    criaAresta(gr,4,3,80);
    criaAresta(gr,1,5,80);
    imprime(gr);

    bool visitados[tam];
    for (int i= 0; i<tam; i++)
        visitados[i]=false;

    for (int i = 0; i<tam; i++){
        if (visitados[i]==false){
            bfs_vist(gr,i,visitados);
            arvores++;
        }
        i++;
    }

    cout << "Arvores: " << arvores << endl;
    return 0;
}
```

Exercício

- Adaptar os códigos para um grafo não dirigido.
- Adaptar os códigos para representar o grafo usando matriz de adjacência.