Algoritmos e Estruturas de Dados II - SCE-183

Grafos: Principais Representações

Gustavo Batista

Representação

- Existem diversas representações que podem ser utilizadas.
- Importante considerar os algoritmos em grafos como tipos abstratos de dados (TAD).
- É de fundamental importância a independência de implementação para as operações. Dessa forma, pode-se alterar a implementação do TAD sem ter que alterar a implementação do programa que utiliza o TAD.

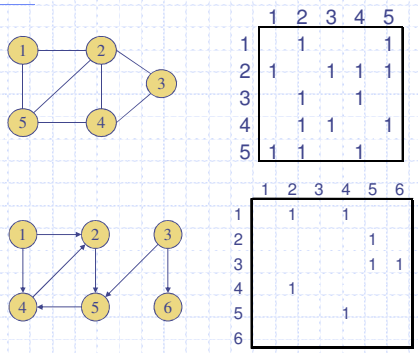
Representação

- Duas representações entre as mais utilizadas são:
  - Matrizes de adjacências, e;
  - Listas de adjacências.
- A escolha de uma representação em particular, depende das operações que serão realizadas no grafo.

Matriz de Adjacências

- Seja $G = (V, A)$ um grafo com n *vértices,* isto é, $n = |V|$, e $n \geq 1$.
- A matriz de adjacências de G é uma matriz M de $n \times n$, tal que $M[v,u] = 1$ se e somente se a existe a aresta *do vértice* v para o vértice u.
- Para grafos ponderados $M[v,u]$ contém o rótulo ou peso associado com a aresta.

Matriz de Adjacências



Matriz de Adjacências - Implementação

- Implementação das operações do TAD utilizando uma matriz de adjacências:
  - As operações sobre arestas serão implementadas de forma que $(u,v) \neq (v,u)$;
  - Entretanto, pode-se modificar a implementação se os grafos a serem utilizados sejam não orientados.

Operações do TAD Grafo

- InicializaGrafo(Grafo, NV): Cria um grafo vazio com NV vértices.
- InsereAresta(v, u, Peso, Grafo): Insere a aresta *(v,u)* no grafo com peso.
- ExisteAresta(v, u, Grafo): Verifica se existe a aresta *(v,u)*.
- RetiraAresta(v, u, Grafo): Retira a aresta *(v,u)* do grafo.
- LiberaGrafo(Grafo): Liberar o espaço ocupado por um grafo.

Operações do TAD Grafo

- ExisteAdj(v, Grafo): retorna **verdade** se existe algum vértice adjacente à *v*.
- PrimeiroAdj(v, Grafo): retorna o endereço do primeiro vértice adjacente à *v*.
- ProxAdj(v, p, Grafo): retorna o endereço do próximo vértice adjacente à *v*, iniciando a busca a partir de *p*.
- RecuperaAdj(v, p, u, Peso, Grafo): retorna o vértice *u* e o peso *Peso* associados à aresta apontada por *p* do vértice *v*.

Matriz de Adjacências - Implementação

```c
#define MAXNUMVERTICES 100

typedef int tpeso;
typedef int tvertice;
typedef int tapontador;

typedef struct {
    tpeso mat[MAXNUMVERTICES][MAXNUMVERTICES];
    int num_vertices;
} tgrafo;
```

Matriz de Adjacências - Implementação

```c
void inicializa_grafo(tgrafo *grafo, int num_vertices) {
    int i, j;

    grafo->num_vertices = num_vertices;
    for (i = 0; i < grafo->num_vertices; i++)
        for (j = 0; j < grafo->num_vertices; j++)
            grafo->mat[i][j] = 0;
}
```

Matriz de Adjacências - Implementação

```c
void insere_aresta(tvertice v, tvertice u,
                   tpeso peso, tgrafo *grafo) {
    grafo->mat[v][u] = peso;
}

int existe_aresta(tvertice v, tvertice u,
                  tgrafo *grafo) {
    return grafo->mat[v][u] != 0;
}
```

Matriz de Adjacências - Implementação

```c
void retira_aresta(tvertice v, tvertice u,
                   tgrafo *grafo) {
    if (grafo->mat[v][u] == 0)
        printf("Erro: Aresta inexistente");
    else
        grafo->mat[v][u] = 0;
}
```

Matriz de Adjacências - Implementação

```
int existe_adj(tvertice v, tgrafo *grafo) {
    tvertice aux;

    for (aux = 0; aux < grafo->num_vertices; aux++) {
        if (grafo->mat[v][aux] != 0)
            return 1;
    }
    return 0;
}
```

Matriz de Adjacências - Implementação

```
tapontador primeiro_adj(tvertice v, tgrafo *grafo) {
    tapontador aux;

    for (aux = 0; aux < grafo->num_vertices; aux++)
        if (grafo->mat[v][aux] != 0)
            return aux;
    return NULO;
}
```

Matriz de Adjacências - Implementação

```
tapontador proximo_adj(tvertice v, tapontador aux,
                       tgrafo *grafo) {

    for (aux++; aux < grafo->num_vertices; aux++)
        if (grafo->mat[v][aux] != 0)
            return aux;
    return NULO;
}

void recupera_adj(tvertice v, tapontador p, tvertice *u,
                  tpeso *peso, tgrafo *grafo) {
    *u = p;
    *peso = grafo->mat[v][p];
}
```

Matriz de Adjacências – Características

- Deve ser utilizada para grafos densos, onde $|A|$ é próximo de $|V|^2$.
- O tempo necessário para acessar um elemento é independente de $|V|$ ou $|A|$.
- É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices.

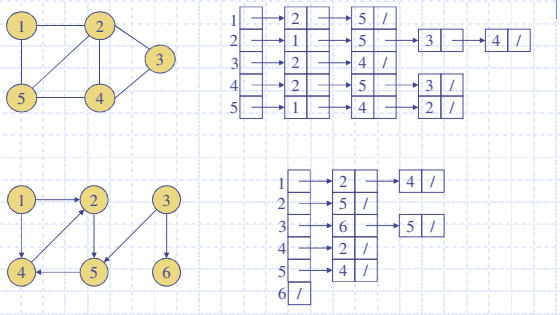
Matriz de Adjacências – Características

- A maior desvantagem é que a matriz necessita $O(|V|^2)$ de espaço. Ler ou examinar a matriz tem complexidade de tempo $O(|V|^2)$.
- Como a matriz de adjacências para um grafo não orientado é simétrica, para grafos não orientados aproximadamente metade do espaço pode ser economizado representando a matriz triangular superior ou inferior.

Lista de Adjacências

- Nesta representação, cada linha da matriz de adjacências é representada por uma lista ligada.
- Os nós na lista u representam os vértices que são adjacentes ao vértice u.
- Cada lista ligada possui um nó cabeça. Os nós cabeça são organizados sequencialmente, fornecendo acesso rápido a qualquer lista ligada.

Lista de Adjacências



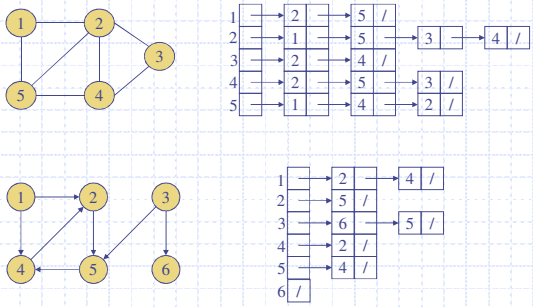
Lista de Adjacências - Características

- Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária.
- Possui uma complexidade de espaço $O(|V|+|A|)$.
- Indicada para grafos esparsos, nos quais $|A|$ é muito menor do que $|V|^2$.

Lista de Adjacências - Características

- Uma desvantagem é que requer $O(|d_v|)$ para determinar se existe uma aresta entre o vértice v e o vértice u, sendo d_v o grau do vértice v (não orientados) ou grau de saída do vértice v (orientados). $d_v \approx |V|$ para vértices que se conectam com muitos vértices.
- Para grafos orientados, determinar o grau de entrada de um vértice requer percorrer todo o grafo.

Lista de Adjacências



Lista de Adjacências

- Existem diversas possibilidades para se implementar as listas de adjacências:
  - Vetor com listas ligada;
  - Vector com listas ligadas;
  - Vector de vectors.
- Vamos analisar a declaração de cada um dessas estruturas.

Lista de Adjacências: LL

```
#define MAXNUMVERTICES 100
#define NULO NULL

typedef int tpeso;
typedef int tvertice;

typedef struct {
    tvertice vertice;
    tpeso peso;
    struct taresta *prox;
} taresta;

typedef taresta* tapontador;

typedef struct {
    tapontador vet[MAXNUMVERTICES];
    int num_vertices;
} tgrafo;
```

Lista de Adjacências: LL e Vectors

```
#include<vector>
#define NULO NULL

typedef int tpeso;
typedef int tvertice;

typedef struct taresta{
    tvertice vertice;
    tpeso peso;
    taresta *prox;
};

typedef taresta* tapontador;

typedef struct {
    std::vector<tapontador> vet;
    int num_vertices;  // Não necessario
} tgrafo;
```

Lista de Adjacências: Vectors de Vectors

```
#include<vector>
#define NULO NULL

typedef int tpeso;
typedef int tvertice;

typedef struct taresta{
    tvertice vertice;
    tpeso peso;
};

typedef taresta* tapontador;

typedef struct {
    std::vector<std::vector<taresta> > vet;
    int num_vertices;     // Não necessario
} tgrafo;
```

Lista de Adjacências – Implementação com Listas Ligadas

- Uma forma bastante comum de implementar a representação de lista de adjacências é utilizando um vetor de ponteiros com listas ligadas simples.
- InicializaGrafo(Grafo, NV):
  - Percorrer o vetor e atribuir NIL a cada posição.
- InsereAresta(v,u,Peso,Grafo):
  - Inserir um nó referente ao vértice *u* na lista ligada do vértice *v* (a inserção pode ser na cabeça da lista).

Lista de Adjacências – Implementação com Listas Ligadas

- ExisteAresta(v,u,Grafo):
  - Percorrer a lista ligada do vértice *v* a procura da aresta *(v,u)*.
- RetiraAresta(v,u,Peso, Grafo):
  - Percorrer a lista ligada do vértice *v* e caso encontrar a aresta *(v,u)* remover o nó.
- LiberaGrafo(Grafo):
  - Percorrer todas as listas ligadas liberando o espaço utilizado por todos os nós.

Lista de Adjacências – Implementação com Listas Ligadas

- ExisteAdj(v, Grafo):
  - Verificar se a posição referente ao vértice *v* do vetor possui valor NIL.
- PrimeiroAdj(v, Grafo):
  - Retorna um ponteiro para o primeiro nó da lista de adjacência de *v*
- ProxAdj(v, p, Grafo):
  - Avança uma posição na lista de adjacência de *v* a partir do ponteiro *p*.

Comparativo de Tempo de Complexidade

Operação	Matriz	Listas
Inicializa	$O(\|V\|^2)$	***O(|V|)***
InsereAresta	$O(1)$	$O(1)$
ExisteAresta	***O(1)***	$O(d_v)$
RetiraAresta	***O(1)***	$O(d_v)$
LiberaGrafo	***O(1)***	$O(\|V\|+\|A\|)$
ExisteAdj	$O(\|V\|)$	***O(1)***
PrimeiroAdj	$O(\|V\|)$	***O(1)***
ProxAdj	$O(\|V\|)$	***O(1)***

Comparativo de Tempo de Complexidade – Outras Operações

Operação	Matriz	Listas
Percorrer um grafo	$O(\|V\|^2)$	***O(|V|+|A|)***
Determinar o grau de um vértice em um grafo não orientado	$O(\|V\|)$	***$O(d_v)$***
Determinar o grau de um vértice em um grafo orientado	$O(\|V\|)$	$O(d_v)$ *(out-degree)* $O(\|V\|+\|A\|)$ *(in-degree)*

Comparativo de Tempo de Complexidade

- Apesar de cada representação ser mais eficiente em diferentes operações do TAD, ***a representação de listas de adjacências é a mais utilizada.***
- Isso ocorre, pois muitos algoritmos clássicos sobre grafos requerem percorrer todos os vértices.
- Por fim, as listas ligadas permitem representar naturalmente multi-grafos.

Exercício

- Implementar na linguagem de programação de sua preferência as operações do TAD Grafo com a representação de listas de adjacências.

Exercício

- O grafo transposto de um grafo direcionado $G = (V,A)$ é definido como $G^T = (V, A^T)$, em que $A^T = \{(u,v) : (v,u) \in A\}$. Ou seja A^T possui as arestas de G com as direções invertidas. Implemente um procedimento que calcule G^T e avalie o desempenho nas duas representações.