MCTA027-17 - Teoria dos Grafos

Representando um grafo

Maycon Sambinelli
m.sambinelli@ufabc.edu.br

2023.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Introdução

- Vamos assumir que os rótulos de um grafo com n vértices são os inteiros no intervalo $[0, n-1]$.
- Vamos sempre fazer análise de pior caso
  - Embora ela represente uma estimativa pessimista de desempenho
- Quando analisando o tempo de execução de um algoritmo em um grafo G, vamos escrever V para denotar $|V(G)|$ e E para denotar $|E(G)|$.
- Vamos assumir que $V \leq E$, assim podemos abreviar expressões como $V(V+E)$ para VE.
- Vamos nos preocupar com grafos simples
- Vamos lidar com grafos estáticos:
  1. Construímos o grafo
  2. Processamos ele: calculamos algum valor numérico ou encontramos algum subconjunto de arestas.

Tipo Abstrato de Dados (TAD): Grafo i

UFABC

```c
// graph.h
#ifndef __GRAPH_H_
#define __GRAPH_H_

typedef int Vertex;
typedef struct {Vertex u; Vertex v;} Edge;

Edge edge(Vertex, Vertex);

typedef struct graph* Graph;

Graph graph(int);
void graph_destroy(Graph);

int graph_order(Graph);
int graph_size(Graph);

void graph_insert_edge(Graph, Edge);
void graph_insert_edges(Graph, Edge*);
void graph_remove_edge(Graph, Edge);
void graph_remove_edges(Graph, Edge*);

```

Tipo Abstrato de Dados (TAD): Grafo ii

UFABC

```
23  int graph_has_edge(Graph, Edge);
24  void graph_edges(Graph, Edge*);
25
26  int graph_degree(Graph, Vertex);
27  int graph_neighbors(Graph, Vertex, Vertex*);
28
29  Graph graph_copy(Graph);
30
31  void graph_print(Graph);
32  void graph_print_edges(Graph);
33
34  Graph graph_squared(Graph);
35  Graph graph_GNP(int, double);
36  #endif // __GRAPH_H_
```

- Veremos duas implementações desse TAD: matriz de adjacências e lista de adjacências
- Vamos assumir que cada implementação desse TAD contém os campos V e E, contendo a ordem e o número de arestas do grafo, respectivamente

Tipo Abstrato de Dados (TAD): DiGrafo i

UFABC

```c
// graph.h
#ifndef __DIGRAPH_H_
#define __DIGRAPH_H_

typedef int Vertex;
typedef struct {Vertex u; Vertex v;} Arc;

Arc arc(Vertex, Vertex);

typedef struct digraph* DiGraph;

DiGraph digraph(int);
void digraph_destroy(DiGraph);

int digraph_order(DiGraph);
int digraph_size(DiGraph);

void digraph_insert_arc(DiGraph, Arc);
void digraph_insert_arcs(DiGraph, Arc*);
void digraph_remove_arc(DiGraph, Arc);
void digraph_remove_arcs(DiGraph, Arc*);

```

Tipo Abstrato de Dados (TAD): DiGrafo ii

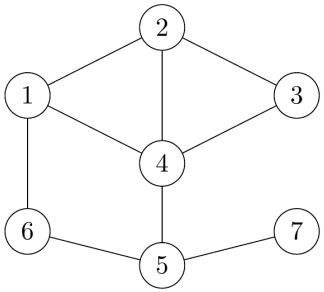
UFABC

```
23  int digraph_has_arc(DiGraph, Arc);
24  void digraph_arc(DiGraph, Arc*);
25
26  int digraph_in_degree(DiGraph, Vertex);
27  int digraph_out_degree(DiGraph, Vertex);
28  int digraph_in_neighbors(DiGraph, Vertex, Vertex*);
29  int digraph_out_neighbors(DiGraph, Vertex, Vertex*);
30
31  DiGraph digraph_copy(DiGraph);
32
33  void digraph_print(DiGraph);
34  void digraph_print_arcs(DiGraph);
35
36  #endif // __DIGRAPH_H_
```

- Veremos duas implementações desse TAD: matriz de adjacências e lista de adjacências
- Vamos assumir que cada implementação desse TAD contém os campos V e E, contendo a ordem e o número de arestas do digrafo, respectivamente

Matriz de Adjacências

Matriz de Adjacências (ii)



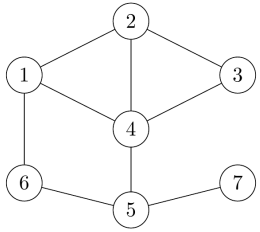
	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

Observações:

- A matriz é simétrica
- O espaço de armazenamento necessário por essa representação é $\Theta(V^2)$.
- Armazena grafo simples
  - O que fazemos se o grafo possui arestas paralelas?

Código

```c
// graph_matrix.c
#include "graph.h"

struct graph {
    int V, E;
    char **adj;
};

Graph graph(int V) {
    Graph G = malloc(sizeof(*G));
    G->V = V;
    G->E = 0;

    G->adj = malloc(V * sizeof(*G->adj));
    for (Vertex u = 0; u < V; u++) {
        G->adj[u] = malloc(V * sizeof(G->adj[u]));
        for (Vertex v = 0; v < V; v++)
            G->adj[u][v] = 0;
    }
    return G;
}
```



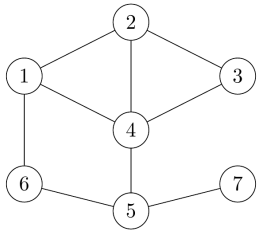
	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

- Espaço: $\Theta(V^2)$
- Inicialização: $\Theta(V^2)$

Código: Inserção e Remoção de Arestas

```
1  void graph_insert_edge(Graph G, Edge e) {
2      if (!G->adj[e.u][e.v])
3          G->E += 1;
4      G->adj[e.u][e.v] = 1;
5      G->adj[e.v][e.u] = 1;
6  }
7
8  void graph_remove_edge(Graph G, Edge e) {
9      if (G->adj[e.u][e.v])
10         G->E -= 1;
11     G->adj[e.u][e.v] = 0;
12     G->adj[e.v][e.u] = 0;
13 }
```

Inserção e remoção de uma aresta: $O(1)$



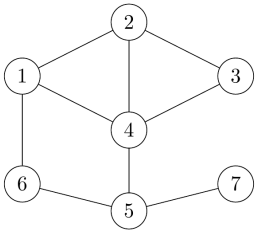
	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

Código: Teste de Pertinência de uma Aresta

UFABC

```c
int graph_has_edge(Graph G, Edge e) {
    return G->adj[e.u][e.v];
}
```

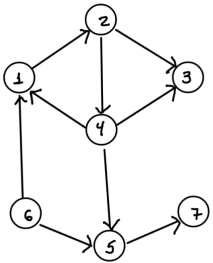
Teste de pertinência de uma aresta: $\Theta(1)$



	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

A *matriz de adjacências* de um digrafo G com n vértices é uma matriz quadrada M de dimensões $n \times n$ tal que $M[u][v] = 1$ se a aresta $uv \in E(G)$ e $M[u][v] = 0$, caso contrário.

- A definição é igual a de grafos, mas vamos interpretar uv como par ordenado



	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	1	0	0	0
3	0	0	0	0	0	0	0
4	1	0	1	0	1	0	0
5	0	0	0	0	0	0	1
6	1	0	0	0	1	0	0
7	0	0	0	0	0	0	0

- Obs note que a matriz deixa de ser simétrica
- Podemos adaptar facilmente os códigos vistos anteriormente para lidarem com digrafos

Códigos para Digrafos

UFABC

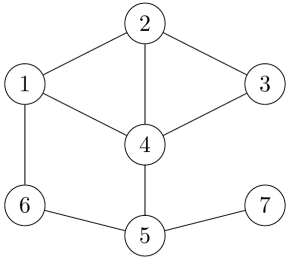
```c
void digraph_insert_arc(DiGraph G, Arc e) {
    if (!G->adj[e.u][e.v])
        G->E += 1;
    G->adj[e.u][e.v] = 1;
    // G->adj[e.v][e.u] = 1;
}

void digraph_remove_arc(DiGraph G, Arc e) {
    if (G->adj[e.u][e.v])
        G->E -= 1;
    G->adj[e.u][e.v] = 0;
    // G->adj[e.v][e.u] = 0;
}
```

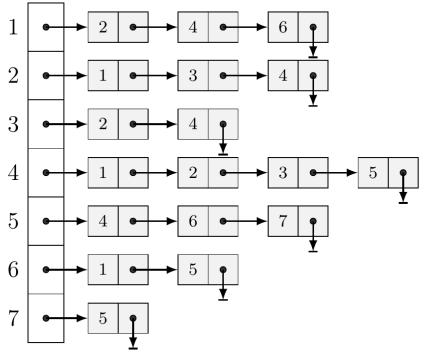
Inserção e remoção de uma aresta: $O(1)$

Lista de adjacências

Lista de Adjacências

A *lista de adjacências* de um grafo G com n vértices é uma coleção de n listas encadeadas, uma para cada vértice. Dado um vértice u do grafo, a lista encadeada associada a u contém todos os vizinhos de u.

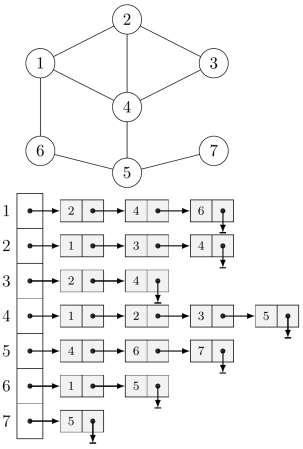




O espaço de armazenamento necessário por essa representação é $\Theta(V + E)$.

Código

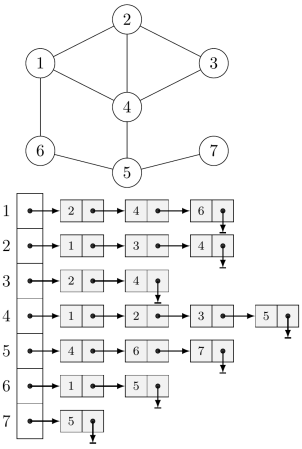
```
1   typedef struct node *link;
2
3   struct node {
4       int w;
5       link next;
6   };
7
8   struct graph {
9       int V;
10      int E;
11      link *adj;
12  };
```



Código: Inicialização

```
1  Graph graph(int V) {
2
3      Graph G = malloc(sizeof(*G));
4
5      G->V = V;
6      G->E = 0;
7      G->adj = malloc(V * sizeof(link));
8
9      for (int u = 0; u < V; u++)
10         G->adj[u] = NULL;
11     return G;
12 }
```

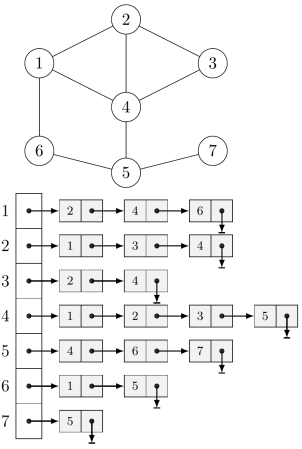
Inicialização: $O(V)$



Código: Teste de Pertinência de Aresta

```
1  int graph_has_edge(Graph G, Edge e) {
2
3      for (link p = G->adj[e.u]; p != NULL; p = p->next)
4          if (p->w == e.v)
5              return 1;
6      return 0;
7  }
```

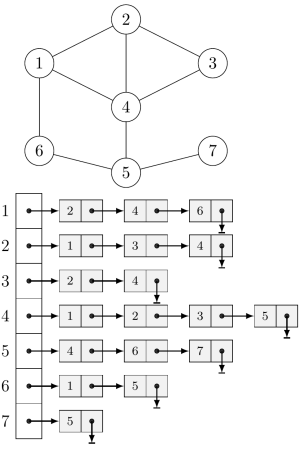
Teste de pertinência: $O(d(u))$



Código: Inserção de Aresta

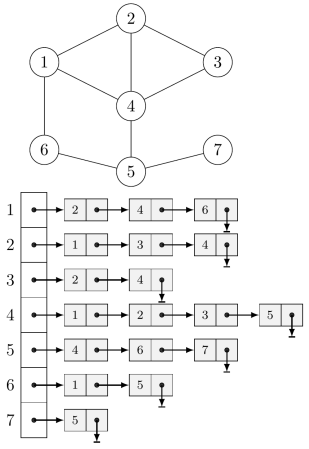
```
1  link list_insert(link head, int w) {
2      link p = malloc(sizeof(*p));
3      p->w = w;
4      p->next = head;
5      return p;
6  }
7
8  void graph_insert_edge(Graph G, Edge e) {
9      G->adj[e.u] = list_insert(G->adj[e.u], e.v);
10     G->adj[e.v] = list_insert(G->adj[e.v], e.u);
11     G->E += 1;
12 }
```

- Inserir $\Theta(1)$
- Arestas paralelas?
  - Essa implementação não evita a inserção de arestas paralelas
  - Verificar a duplicidade de uma aresta requer $O(d(u))$



Código: Remoção de Aresta

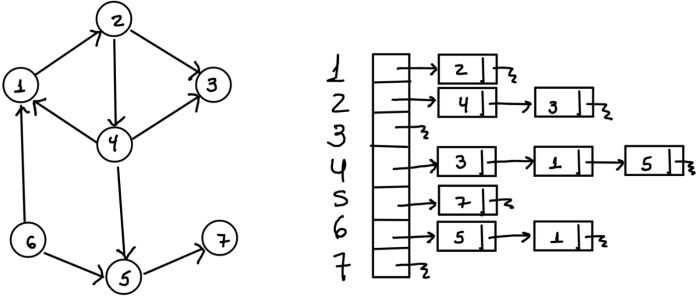
```
1  link list_remove(link head, int w) {
2
3      if (head == NULL) return NULL;
4
5      if (head->w == w) {
6          link p = head->next;
7          free(head);
8          return p;
9      } else {
10         head->next = list_remove(head->next, w);
11         return head;
12     }
13 }
14
15 void graph_remove_edge(Graph G, Edge e) {
16
17     if (!graph_has_edge(G, e))
18         return ;
19
20     G->E -= 1;
21     G->adj[e.u] = list_remove(G->adj[e.u], e.v);
22     G->adj[e.v] = list_remove(G->adj[e.v], e.u);
23 }
```



- Remover uma aresta: $O(d(u))$

Digrafos e Lista de Adjacências

A *lista de adjacências* de um digrafo G com n vértices é uma coleção de n listas encadeadas, uma para cada vértice. Dado um vértice u do grafo, a lista encadeada associada a u contém todos os vértices dominados por u.



- O espaço de armazenamento necessário por essa representação é $\Theta(V + E)$.
- Podemos adaptar facilmente os códigos vistos anteriormente para lidarem com digrafos

Digrafos e Código

UFABC

```
1  void digraph_insert_arc(DiGraph G, Arc e) {
2    G->adj[e.u] = list_insert(G->adj[e.u], e.v);
3    //G->adj[e.v] = list_insert(G->adj[e.v], e.u);
4    G->E += 1;
5  }
6
7  void digraph_remove_arc(DiGraph G, Arc e) {
8    if (!digraph_has_arc(G, e))
9        return ;
10   G->E -= 1;
11   G->adj[e.u] = list_remove(G->adj[e.u], e.v);
12   //G->adj[e.v] = list_remove(G->adj[e.v], e.u);
13 }
```

Sumário: Matriz vs Lista de Adjacências

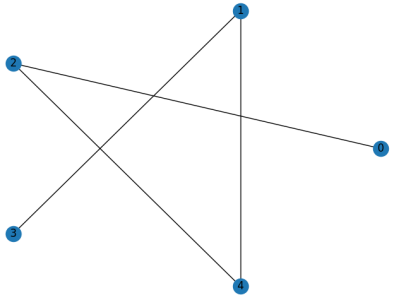
UFABC

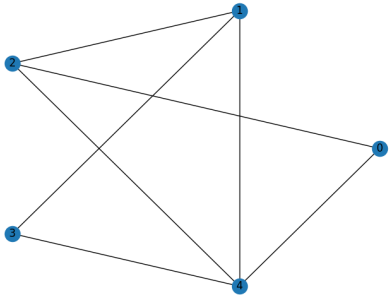
	Matriz	Lista
Espaço	$O(V^2)$	$O(V+E)$
Inicialização	$O(V^2)$	$O(V)$
Verificar Pertinência	$O(1)$	$O(d(u))$
Inserção de Arestas	$O(1)$	$O(1)$
Remoção de Aresta	$O(1)$	$O(d(u))$
Copiar	$O(V^2)$	$O(V+E)$
Destruir	$O(V)$	$O(E)$

Aplicação

UFABC

Dado um grafo G, o grafo G^2 é o grafo tal que $V(G^2) = V(G)$ e os vértices u e v são adjacentes em G^2 se $\text{dist}_G(u, v) \leq 2$.





Programa

```c
#include <stdio.h>
#include "graph.h"
int main(int argc, char const *argv[]) {
    int V, E;
    scanf("%d %d", &V, &E);

    Graph G = graph(V);
    Graph G2 = graph(V);
    for (int i = 0; i < E; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        graph_insert_edge(G, edge(u, v));
        graph_insert_edge(G2, edge(u, v));
    }

    for (int w = 0; w < V; w++)
        for (int u = 0; u < V; u++)
            for (int v = u + 1; v < V; v++)
                if (u != w && v != w
                    && graph_has_edge(G, edge(u, w))
                    && graph_has_edge(G, edge(v, w)))
                    graph_insert_edge(G2, edge(u,v));
    graph_print_edges(G2);

    return 0;
}
```