MCTA027-17 - Teoria dos Grafos

Representando um grafo

Maycon Sambinelli
m.sambinelli@ufabc.edu.br

2020.QS

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Outline

Introdução

Introdução

- Vamos assumir que os rótulos de um grafo com n vértices são os inteiros no intervalo $[0, n-1]$.

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

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

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 $|E(G)|$.

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.

Introdução

UFABC

- 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

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:

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

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.

Grafo Denso e Esparso

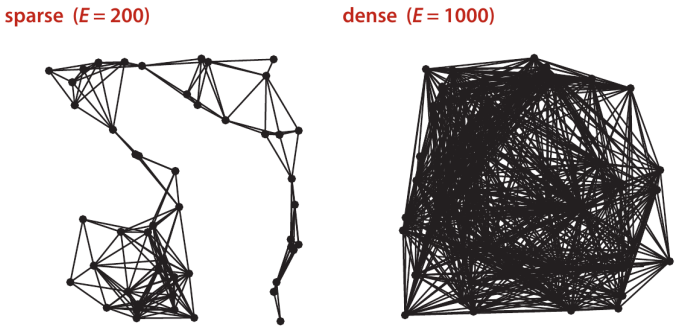
- Um grafo G é *denso* se $E = \Theta(|V^2|)$.

Grafo Denso e Esparso

- Um grafo G é *denso* se $E = \Theta(|V^2|)$.
- Um grafo G é *esparso* se $E = O(V)$.

Grafo Denso e Esparso

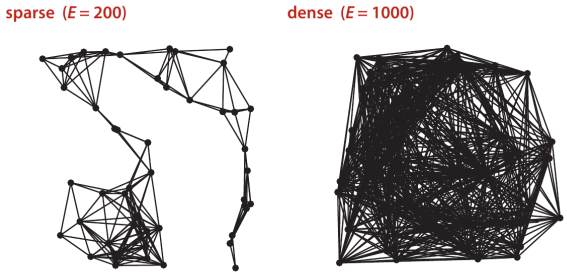
- Um grafo G é *denso* se $E = \Theta(|V^2|)$.
- Um grafo G é *esparso* se $E = O(V)$.



Two graphs ($V = 50$)

Densidade

- A *densidade* de um grafo simples G é $\frac{|E(G)|}{\binom{|V(G)|}{2}}$



Two graphs ($V = 50$)

Densidade: ≈ 0.16 (Esquerda) e ≈ 0.81 (Direita)

Grafo Denso e Esparso: Algoritmos

UFABC

Saber se um grafo é denso ou esparso é um fator chave na hora de selecionar um algoritmo

Exemplo

	Esparso	Denso
V	50,000	50,000
E	4,000,000	980,000,000
Densidade	0,32%	78,4%
Algoritmo V^2	2.500.000.000	2.500.000.000
Algoritmo $E \lg E$	87.726.274	29.270.842.378

- No caso esparso, Algoritmo $E \lg E$ é 28 vezes mais rápido que Algoritmo V^2
- No caso denso, Algoritmo V^2 é 11 vezes mais rápido que o Algoritmo $E \lg E$

Tipo Abstrato de Dados (TAD): Grafo i

UFABC

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

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

Edge edge(int, int);

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 G, Edge* edges);

int graph_has_edge(Graph, Edge);
```

Tipo Abstrato de Dados (TAD): Grafo ii

UFABC

```
23  void graph_edges(Graph, Edge*);
24
25  int graph_degree(Graph, int);
26  int graph_neighbors(Graph, int u, int*);
27
28  Graph graph_copy(Graph);
29
30  void graph_print(Graph);
31  void graph_print_edges(Graph);
32
33  Graph graph_squared(Graph);
34  Graph graph_GNP(int V, double p);
35  #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 tamanho do grafo, respectivamente

Tipo Abstrato de Dados (TAD): Grafo iii

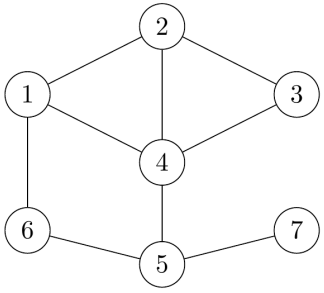
UFABC

- Precisamos lidar com arestas paralelas e laços

Matriz de Adjacências

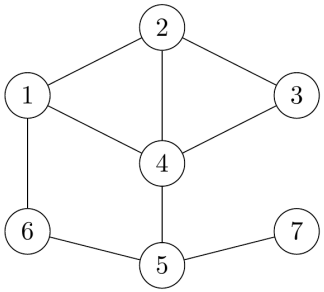
Matriz de Adjacências

A *matriz de adjacências* de um grafo 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.



	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

Matriz de Adjacências (ii)



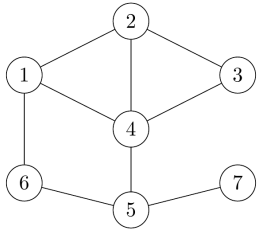
	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

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



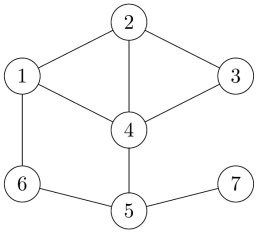
	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

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

Código: Inserção e Remoção de Arestas

```c
void graph_insert_edge(Graph G, Edge 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 graph_remove_edge(Graph G, Edge 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: $O(1)$



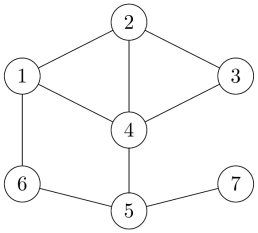
	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

Código: Teste de Pertinência de uma Aresta

UFABC

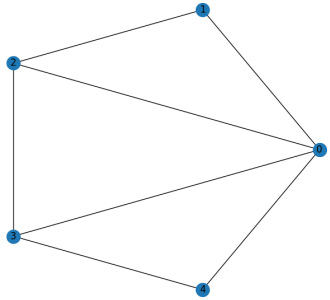
```c
int graph_has_edge(Graph G, Edge e) {
    return G->adj[e.u][e.v];
}
```

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

Código: Impressão de um Grafo



Exemplo

```
5 7
0 1
0 2
0 3
0 4
1 2
2 3
3 4
```

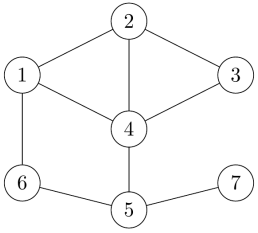
Exemplo

```
V: 5, E: 7
0: 1, 2, 3, 4,
1: 0, 2,
2: 0, 1, 3,
3: 0, 2, 4,
4: 0, 3,
```

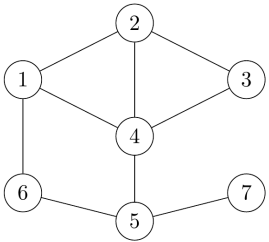
Código: Impressão de um Grafo

```
1  void graph_print(Graph G) {
2
3      printf("V: %d, E: %d\n", G->V, G->E);
4      for (int u = 0; u < G->V; u++) {
5          printf("%2d: ", u);
6          for (int v = 0; v < G->V; v++)
7              if (G->adj[u][v])
8                  printf("%d, ", v);
9          printf("\n");
10     }
11 }
```

Impressão: (V^2)



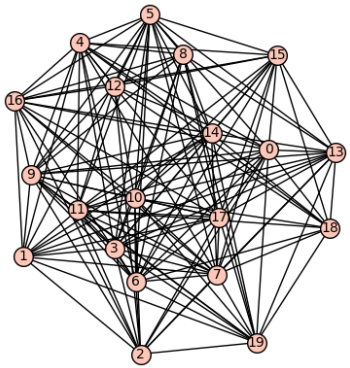
	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0



	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

- Armazenar apenas a diagonal superior
- Usar um vetor de bits
  - Graph6

- Formato *Graph6* (https://rdrr.io/rforge/rgraph6/man/graph6.html)




```
S^v\\~{]~zn~zjN|]J~k}^z}n~|[U^ffzK
```

- $V = 20$ e $E = 146$

Matriz de Adjacências para grafos densos

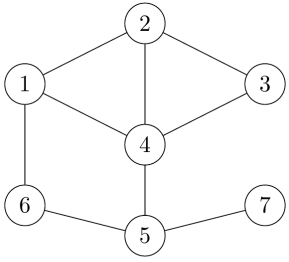
UFABC

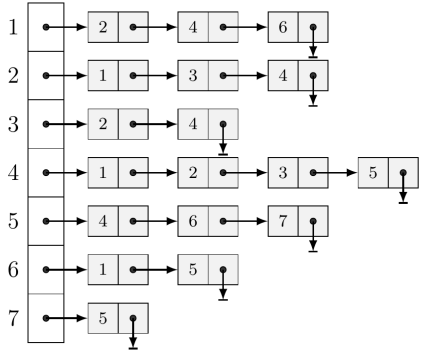
A matriz de adjacências é uma representação que não é adequada pra grandes grafos esparsos:

- A matriz requere V^2 bits e V^2 passos para inicialização.
- Grafo denso
  - tem um número de arestas proporcional a V^2
  - Poucas entradas são deixadas em branco
  - O custo V^2 de inicialização é compensado pela leitura das $\approx V^2$ arestas.

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.

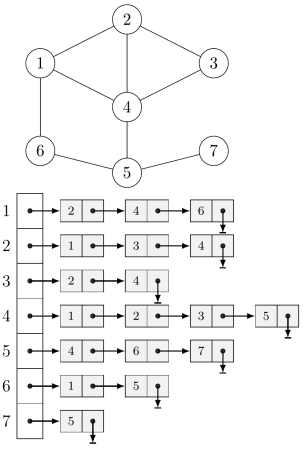




O espaço de armazenamento necessário por essa representação é $\Theta(V + E)$.

Código

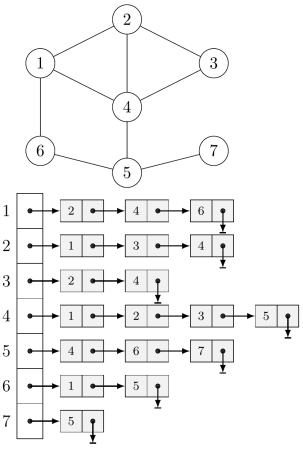
```
1   typedef struct node *link;
2
3   struct node {
4       int w;
5       link next;
6   };
7
8   struct graph {
9       int V;
10      int E;
11      link *adj;
12  };
```



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  }
```

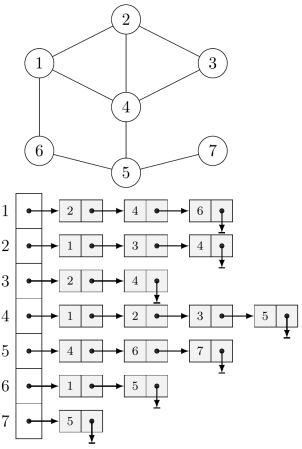
Inicialização: $O(V)$



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  }
```

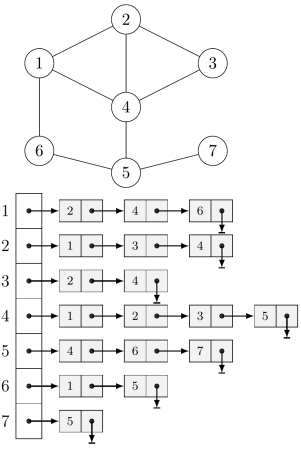
Teste de pertinência: $O(d(u))$



Código: Inserção de Aresta

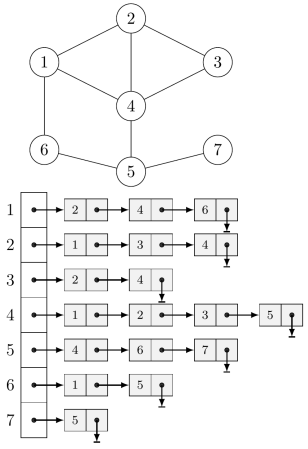
```
1  link list_insert(link head, int w) {
2      link p = Malloc(sizeof(*p));
3      p->w = w;
4      p->next = head;
5      return p;
6  }
7
8  void graph_insert_edge(Graph G, Edge e) {
9      G->adj[e.u] = list_insert(G->adj[e.u], e.v);
10     G->adj[e.v] = list_insert(G->adj[e.v], e.u);
11     G->E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

Sumário: Matriz vs Lista de Adjacências

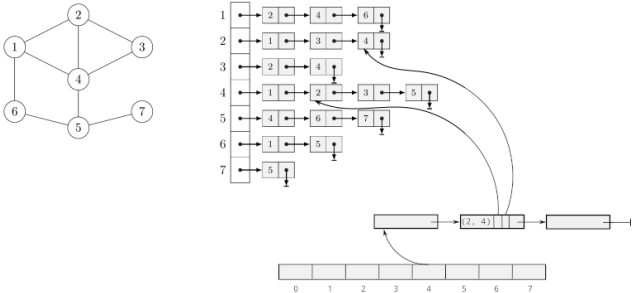
UFABC

	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	$O(V)$	$O(E)$

UFABC

- Em vários casos, nós podemos modificar a representação para fazer uma operação simples mais eficiente
- Precisamos tomar cuidado pra não tornar outras operações mais custosas

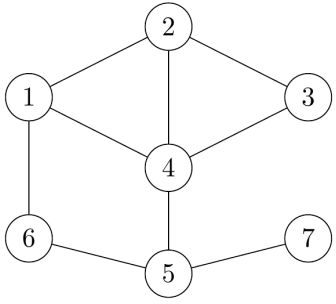
Otimizações: pertinência e remoção de aresta em $O(1)$



- Usando uma tabela hash:
  1. podemos testar a pertinência de uma aresta em $O(1)$, na média (tempo amortizado)
  2. junto com uma lista duplamente encadeada, podemos remover uma aresta remover uma aresta em tempo $O(1)$, na média (tempo amortizado)
- Para grandes grafos estáticos, podemos usar vetores, ao invés de listas encadeadas, para armazenar os vizinhos de um vértice.

Comentários

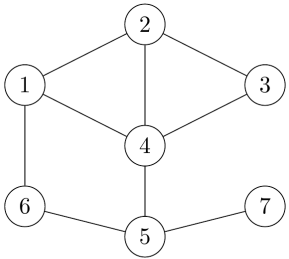
Remoção de vértices: Matriz de Adjacências

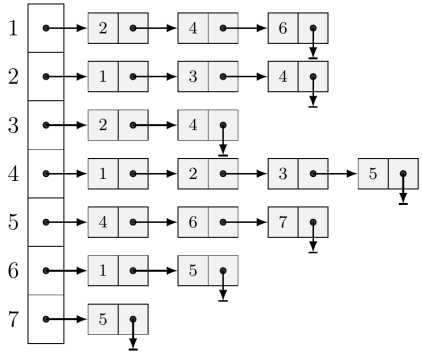


	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

- Remover uma linha e uma coluna
  - Praticamente o mesmo custo que refazer o grafo

Remoção de vértices: Lista de Adjacências

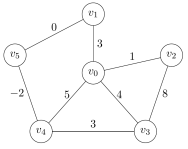




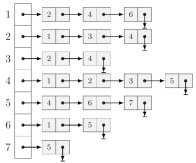
- Remover uma lista e uma entrada do vetor

UFABC

Um *grafo ponderado* é um grafo cujos vértices e/ou arestas possuem pesos associados.



	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



- Matriz de Adjacências: preenchemos a matriz com pesos ao invés de valores booleanos (usamos um peso inválido para representar a ausência de uma aresta)
- Lista de Adjacências: inserimos o peso nos nós das listas

Adicionando Propriedades aos Vértices e/ou Arestas

UFABC

- Podemos precisar associar dados aos vértices ou arestas (ex. pesos).
- Podemos fazer isso:
  - Estendendo as definições
  - Usando um array externo

Adicionando Propriedades (Estendendo as definições)

UFABC

```c
// graph_matrix.c
struct data {
    int status;
    int weight;
    int color;
    char[100] label;
};

struct graph {
    int V;
    int E;
    int *degree;
    struct data **adj;
};
```

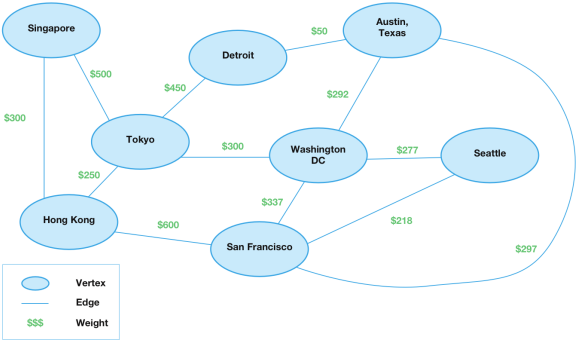
```c
// graph_adjacency_list.c
struct node {
    int w;
    int weight;
    int color;
    char[100] label;
    struct node *next;
};

struct graph {
    int V;
    int E;
    int *degree;
    struct node *adj;
};
```

Adicionando Propriedades (Array externo)

UFABC

```
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 };
13
14 // ... {
15
16     G = graph(n);
17     int **weight = squared_int_matrix(graph_order(G), 5.2);
18     int *degree = Malloc(graph_order(G) * sizeof(*degree));
19     int *visited = Malloc(graph_order(G) * sizeof(*visited));
```

Grafos cujos rótulos dos vértices não são inteiros




```
SymbolTable st = ST();
ST_insert(st, "Singapore");

graph_insert_edge(G, edge(ST_get("Seattle"), ST_get("San Franscisco")));
```

Grafos cujos rótulos não são inteiros

UFABC

Um programa cliente pode associar os rótulos de um grafo com n vértices a inteiros entre 0 e $n-1$ de maneiras distintas.

- Isso leva a construção de matrizes e listas de adjacências distintas

Grafos cujos rótulos não são inteiros

UFABC

Um programa cliente pode associar os rótulos de um grafo com n vértices a inteiros entre 0 e $n-1$ de maneiras distintas.

- Isso leva a construção de matrizes e listas de adjacências distintas
  - Não podemos reconhecer isso fácil: problema de isomorfismo

Grafos cujos rótulos não são inteiros

UFABC

Um programa cliente pode associar os rótulos de um grafo com n vértices a inteiros entre 0 e $n-1$ de maneiras distintas.

- Isso leva a construção de matrizes e listas de adjacências distintas
  - Não podemos reconhecer isso fácil: problema de isomorfismo
    - No caso de listas de adjacências, mesmo com um mapeamento fixo!

Grafos cujos rótulos não são inteiros

UFABC

Um programa cliente pode associar os rótulos de um grafo com n vértices a inteiros entre 0 e $n-1$ de maneiras distintas.

- Isso leva a construção de matrizes e listas de adjacências distintas
  - Não podemos reconhecer isso fácil: problema de isomorfismo
    - No caso de listas de adjacências, mesmo com um mapeamento fixo!
- A ordem em que os vértices aparecem na vizinhança de um vértice ditará o funcionamento do algoritmo

Grafos cujos rótulos não são inteiros

UFABC

Um programa cliente pode associar os rótulos de um grafo com n vértices a inteiros entre 0 e $n-1$ de maneiras distintas.

- Isso leva a construção de matrizes e listas de adjacências distintas
  - Não podemos reconhecer isso fácil: problema de isomorfismo
    - No caso de listas de adjacências, mesmo com um mapeamento fixo!
- A ordem em que os vértices aparecem na vizinhança de um vértice ditará o funcionamento do algoritmo
  - Execuções distintas

Grafos cujos rótulos não são inteiros

UFABC

Um programa cliente pode associar os rótulos de um grafo com n vértices a inteiros entre 0 e $n-1$ de maneiras distintas.

- Isso leva a construção de matrizes e listas de adjacências distintas
  - Não podemos reconhecer isso fácil: problema de isomorfismo
    - No caso de listas de adjacências, mesmo com um mapeamento fixo!
- A ordem em que os vértices aparecem na vizinhança de um vértice ditará o funcionamento do algoritmo
  - Execuções distintas
  - O algoritmo deve funcionar independentemente da ordem do mapeamento

Grafos cujos rótulos não são inteiros

UFABC

Um programa cliente pode associar os rótulos de um grafo com n vértices a inteiros entre 0 e $n-1$ de maneiras distintas.

- Isso leva a construção de matrizes e listas de adjacências distintas
  - Não podemos reconhecer isso fácil: problema de isomorfismo
    - No caso de listas de adjacências, mesmo com um mapeamento fixo!
- A ordem em que os vértices aparecem na vizinhança de um vértice ditará o funcionamento do algoritmo
  - Execuções distintas
  - O algoritmo deve funcionar independentemente da ordem do mapeamento
  - Problemas que admitem múltiplas soluções: respostas distintas dependendo da ordem do mapeamento