# Disciplina MCTA027-17 - Teoria dos Grafos

## Representação de grafos em computadores

**Profa. Carla Negri Lintzmayer**

carla.negri@ufabc.edu.br

www.professor.ufabc.edu.br/∼carla.negri

Centro de Matemática, Computação e Cognição – Universidade Federal do ABC

# Introdução

# Algumas observações iniciais

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

## Algumas observações iniciais

- Vamos assumir que os rótulos de um grafo de ordem $n$ são os inteiros no intervalo $[0, n-1]$;
- Vamos sempre fazer análise de pior caso;

# Algumas observações iniciais

- Vamos assumir que os rótulos de um grafo de ordem $n$ são os inteiros no intervalo $[0, n-1]$;
- Vamos sempre fazer análise de pior caso;
- Vamos sempre passar a escrever $V$ para $|V(G)|$ e $E$ para $|E(G)|$;

# Algumas observações iniciais

- Vamos assumir que os rótulos de um grafo de ordem $n$ são os inteiros no intervalo $[0, n-1]$;
- Vamos sempre fazer análise de pior caso;
- Vamos sempre passar a escrever $V$ para $|V(G)|$ e $E$ para $|E(G)|$;
- Sempre que $V - 1 \leq E$ (conexo), note que $O(V + E)$ é $O(E)$;

# Algumas observações iniciais

- Vamos assumir que os rótulos de um grafo de ordem $n$ são os inteiros no intervalo $[0, n-1]$;
- Vamos sempre fazer análise de pior caso;
- Vamos sempre passar a escrever $V$ para $|V(G)|$ e $E$ para $|E(G)|$;
- Sempre que $V - 1 \leq E$ (conexo), note que $O(V + E)$ é $O(E)$;
- Vamos lidar com grafos estáticos:

## Algumas observações iniciais

- Vamos assumir que os rótulos de um grafo de ordem $n$ são os inteiros no intervalo $[0, n-1]$;
- Vamos sempre fazer análise de pior caso;
- Vamos sempre passar a escrever $V$ para $|V(G)|$ e $E$ para $|E(G)|$;
- Sempre que $V - 1 \leq E$ (conexo), note que $O(V + E)$ é $O(E)$;
- Vamos lidar com grafos estáticos:
  - Construímos o grafo e não adicionamos/removemos nenhum vértice/aresta.

# Densidade de grafos

Seja $G$ um grafo:

- dizemos que $G$ é *denso* se $E = \Theta(V^2)$ e é *esparso* se $E = O(V)$;

# Densidade de grafos

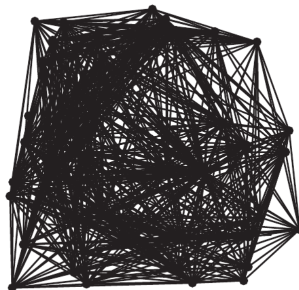
Seja $G$ um grafo:

- dizemos que $G$ é *denso* se $E = \Theta(V^2)$ e é *esparso* se $E = O(V)$;
- a *densidade* de $G$ é $\frac{E}{\binom{V}{2}}$.

# Densidade de grafos

Seja $G$ um grafo:

- dizemos que $G$ é *denso* se $E = \Theta(V^2)$ e é *esparso* se $E = O(V)$;
- a *densidade* de $G$ é $\dfrac{E}{\binom{V}{2}}$.




Two graphs ($V = 50$)

Densidade: $\approx 0.16$ (esquerda) e $\approx 0.81$ (direita)

# Densidade de grafos e implicações

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

Exemplo

| | Esparso | Denso |
|---|---|---|
| $V$ | 50.000 | 50.000 |
| $E$ | 4.000.000 | 980.000.000 |
| $V + E$ | 4.050.000 | 980.050.000 |
| $V^2$ | 2.500.000.000 | 2.500.000.000 |
| Densidade | 32% | 78,4% |
| Algoritmo $V^2$ | 2.500.000.000 | 2.500.000.000 |
| Algoritmo $E \log_2 E$ | 87.726.274 | 29.270.842.378 |

# Densidade de grafos e implicações

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

Exemplo

| | Esparso | Denso |
|---|---|---|
| $V$ | 50.000 | 50.000 |
| $E$ | 4.000.000 | 980.000.000 |
| $V + E$ | 4.050.000 | 980.050.000 |
| $V^2$ | 2.500.000.000 | 2.500.000.000 |
| Densidade | 32% | 78,4% |
| Algoritmo $V^2$ | 2.500.000.000 | 2.500.000.000 |
| Algoritmo $E \log_2 E$ | 87.726.274 | 29.270.842.378 |

- No caso esparso, o algoritmo $E \log_2 E$ é 28 vezes mais rápido que o outro
- No caso denso, o algoritmo $V^2$ é 11 vezes mais rápido que o outro

# Tipo Abstrato de Dados (TAD): Grafo i

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

typedef int Vertex;
typedef struct {Vertex u; Vertex v;} Edge;
typedef struct graph* Graph;

Edge edge(Vertex, Vertex);
Graph graph(int);

void graph_destroy(Graph);
int graph_order(Graph);
int graph_num_edges(Graph);

void graph_insert_edge(Graph, Edge);
void graph_insert_edges(Graph, Edge*, int);
```

# Tipo Abstrato de Dados (TAD): Grafo ii

```
18  void graph_remove_edge(Graph, Edge);
19  void graph_remove_edges(Graph, Edge*, int);
20
21  int graph_has_edge(Graph, Edge);
22  int graph_edges(Graph, Edge*);
23
24  int graph_vertex_degree(Graph, Vertex);
25  int graph_neighbors(Graph, Vertex, Vertex*);
26
27  Graph graph_copy(Graph);
28
29  void graph_print(Graph);
30  void graph_print_edges(Graph);
31
32  Graph graph_squared(Graph);
33  Graph graph_GNP(int, double);
34
35  #endif // __GRAPH_H_
```

# Tipo Abstrato de Dados (TAD): Grafo

- Veremos duas implementações desse TAD: matriz de adjacências e lista de adjacências

# Tipo Abstrato de Dados (TAD): Grafo

- 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

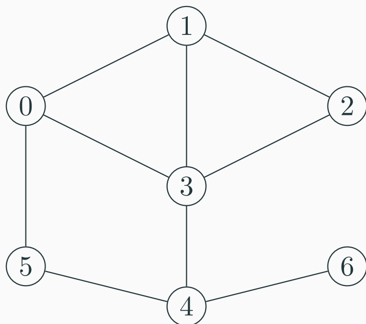
# Tipo Abstrato de Dados (TAD): Grafo

- 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
- Precisamos lidar com arestas paralelas e laços

# Matriz de Adjacências

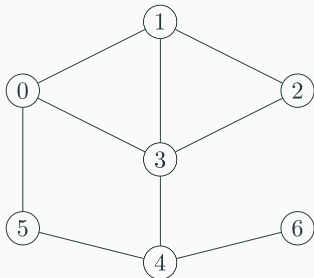
# Matriz de Adjacências

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




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# Matriz de Adjacências

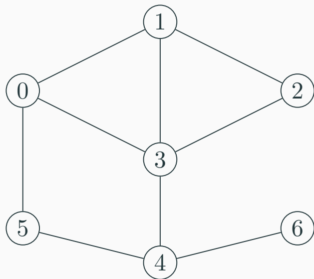



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Observações:

- A matriz é simétrica

# Matriz de Adjacências

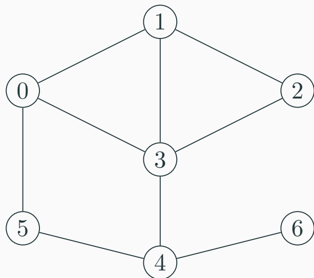



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Observações:

- A matriz é simétrica
- O espaço de armazenamento necessário por essa representação é $\Theta(V^2)$.

# Matriz de Adjacências

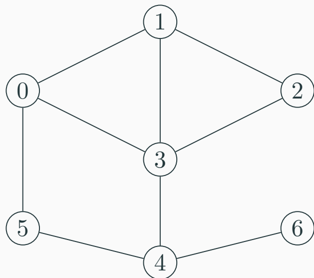



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Observações:

- A matriz é simétrica
- O espaço de armazenamento necessário por essa representação é $\Theta(V^2)$.
- Armazena grafo simples

# Matriz de Adjacências




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Observações:

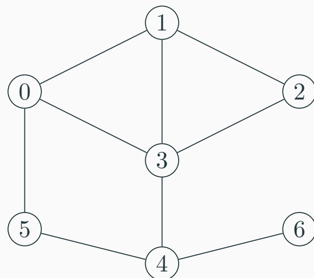
- A matriz é simétrica
- O espaço de armazenamento necessário por essa representação é $\Theta(V^2)$.
- Armazena grafo simples
    - E se o grafo possui arestas paralelas?

# Código

```
1   // graph_matrix.c
2   #include "graph.h"
3
4   struct graph {
5      int V, E;
6      char **adj;
7   };
8
9   Graph graph(int V) {
10     Graph G = malloc(sizeof(*G)); // sizeof(struct graph)
11     G->V = V;
12     G->E = 0;
13
14     G->adj = malloc(V * sizeof(*G->adj)); // sizeof(*char)
15     for (Vertex u = 0; u < V; u++) {
16        G->adj[u] = malloc(V * sizeof(G->adj[u])); // sizeof(char)
17        for (Vertex v = 0; v < V; v++)
18           G->adj[u][v] = 0;
19     }
20     return G;
21  }
```

# Código

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

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

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

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



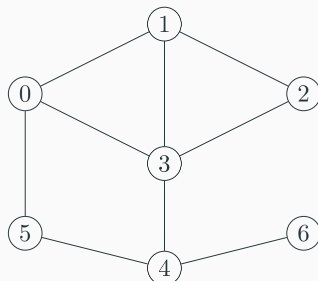

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

- Espaço: $\Theta(V^2)$
- Tempo para 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;
}
```

Tempo para inserção e remoção de uma aresta:

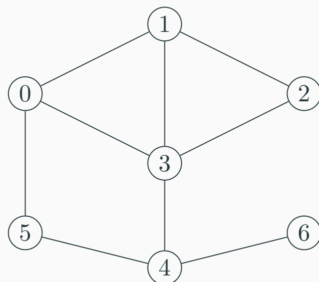



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# 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;
}
```

Tempo para inserção e remoção de uma aresta: $O(1)$

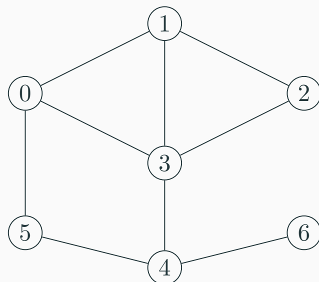



|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# Código: Teste de Pertinência de uma Aresta

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

Tempo para o teste de pertinência de uma aresta:

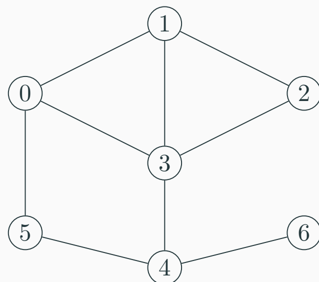



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# Código: Teste de Pertinência de uma Aresta

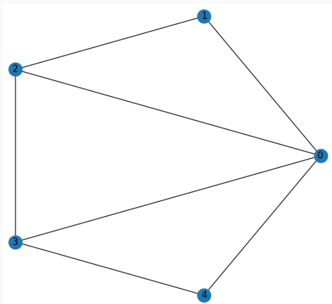
```
1  int graph_has_edge(Graph G, Edge e) {
2      return G->adj[e.u][e.v];
3  }
```

Tempo para o teste de pertinência de uma aresta: $O(1)$




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# Problema: Impressão de um Grafo




**Exemplo**

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

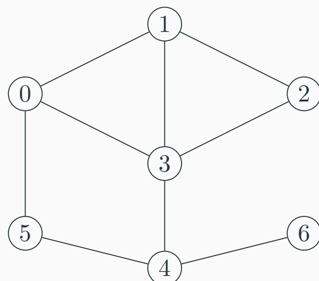
**Exemplo**

```
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

```c
void graph_print(Graph G) {

    printf("V: %d, E: %d\n", G->V, G->E);
    for (Vertex u = 0; u < G->V; u++) {
        printf("%2d: ", u);
        for (Vertex v = 0; v < G->V; v++)
            if (G->adj[u][v])
                printf("%d, ", v);
        printf("\n");
    }
}
```

Tempo para impressão:

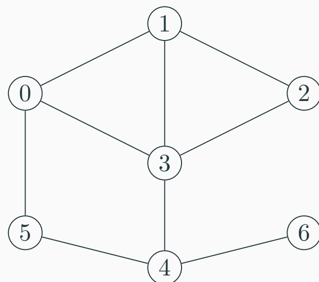



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# Código: Impressão de um Grafo

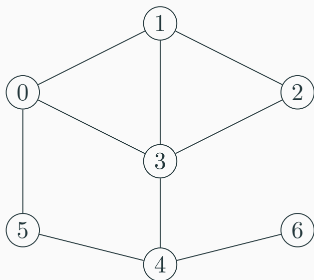
```
void graph_print(Graph G) {

    printf("V: %d, E: %d\n", G->V, G->E);
    for (Vertex u = 0; u < G->V; u++) {
        printf("%2d: ", u);
        for (Vertex v = 0; v < G->V; v++)
            if (G->adj[u][v])
                printf("%d, ", v);
        printf("\n");
    }
}
```

Tempo para impressão: $O(V^2)$




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# Otimizações

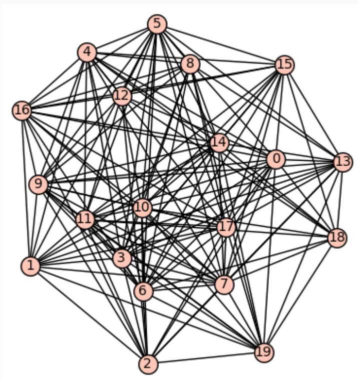



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

- É possível armazenar apenas a diagonal superior
- Usar um vetor de bits
  - Graph6

# 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

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

- A matriz requer $V^2$ de espaço e tempo $V^2$ para inicialização

# Matriz de Adjacências para grafos densos

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

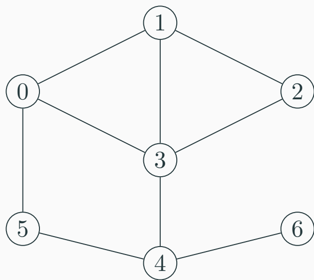
- A matriz requer $V^2$ de espaço e tempo $V^2$ para inicialização
- Grafo denso
  - tem um número de arestas proporcional a $V^2$
  - poucas entradas são deixadas em branco
  - o custo $V^2$ de inicialização é compensado pela leitura das $\approx V^2$ arestas.

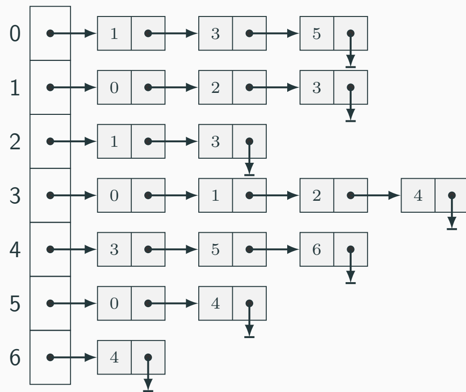
# Lista de adjacências

# Lista de Adjacências

A *lista de adjacências* de um grafo $G$ vértices é uma coleção de $V$ listas encadeadas, uma para cada vértice.

Dado um vértice $u$ do grafo, a lista encadeada associada a $u$ contém todos os vizinhos de $u$.

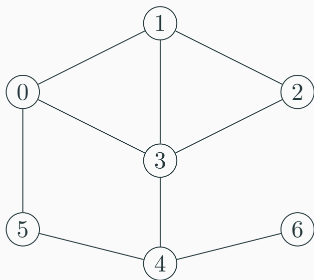


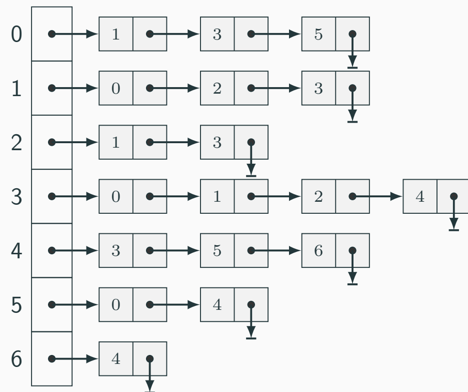



# Lista de Adjacências

A *lista de adjacências* de um grafo $G$ vértices é uma coleção de $V$ listas encadeadas, uma para cada vértice.

Dado um vértice $u$ do grafo, a lista encadeada associada a $u$ contém todos os vizinhos de $u$.

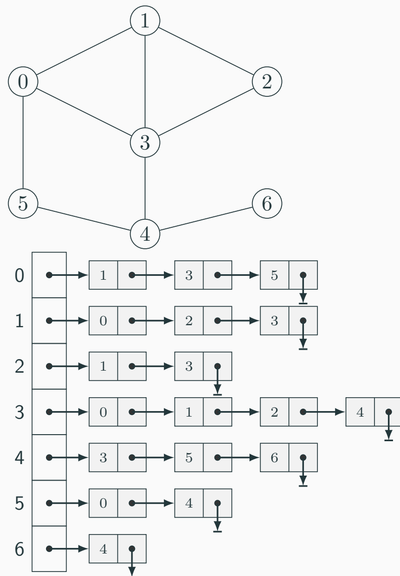



O espaço de armazenamento necessário por essa representação é $\Theta(V + E)$.

# Código

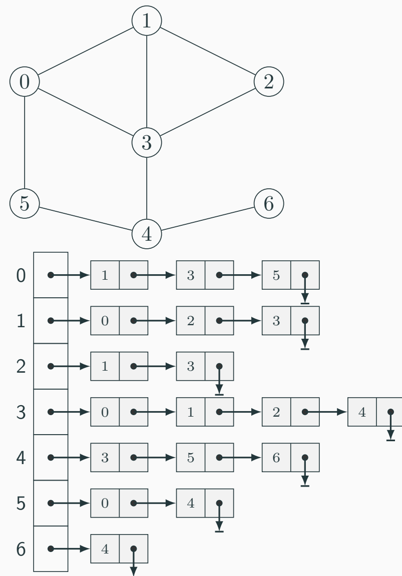
```c
// graph_adjacency_list.c
#include "graph.h"

typedef struct node *link;

struct node {
    Vertex w;
    link next;
};

struct graph {
    int V;
    int E;
    link *adj;
};
```



# 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 (Vertex u = 0; u < V; u++)
10         G->adj[u] = NULL;
11     return G;
12 }
```

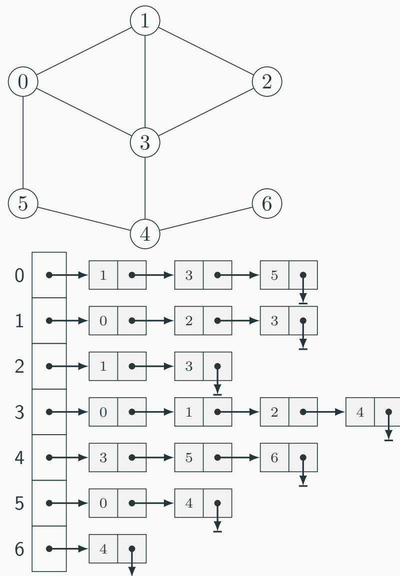
Tempo para inicialização:



# 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 (Vertex u = 0; u < V; u++)
10         G->adj[u] = NULL;
11     return G;
12 }
```

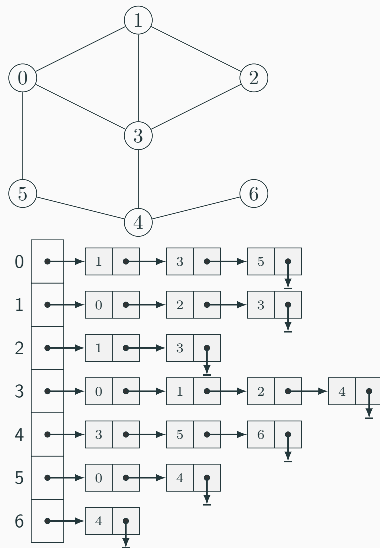
Tempo para inicialização: $O(V)$



# 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  }
```

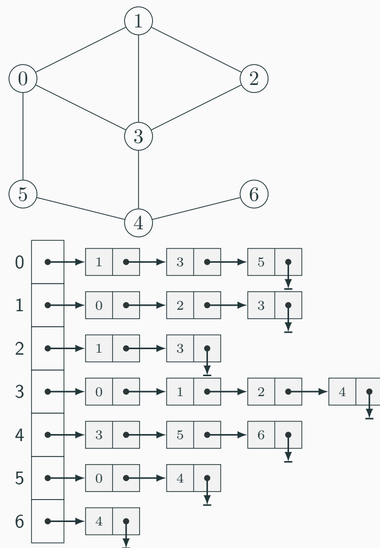
Tempo do teste de pertinência:



# 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  }
```

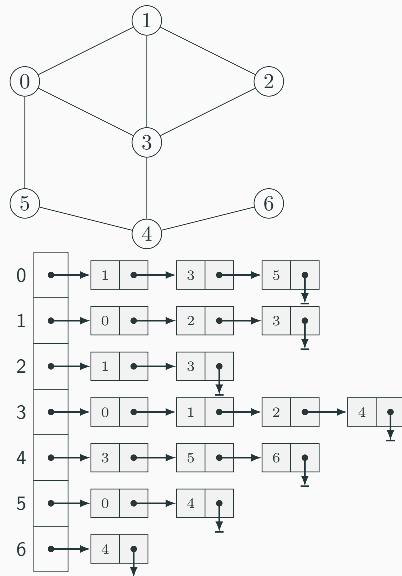
Tempo do teste de pertinência: $O(d(u))$, que é $O(V)$



# Código: Inserção de Aresta

```
link list_insert(link head, int w) {
    link p = malloc(sizeof(*p));
    p->w = w;
    p->next = head;
    return p;
}

void graph_insert_edge(Graph G, Edge e) {
    G->adj[e.u] = list_insert(G->adj[e.u], e.v);
    G->adj[e.v] = list_insert(G->adj[e.v], e.u);
    G->E += 1;
}
```

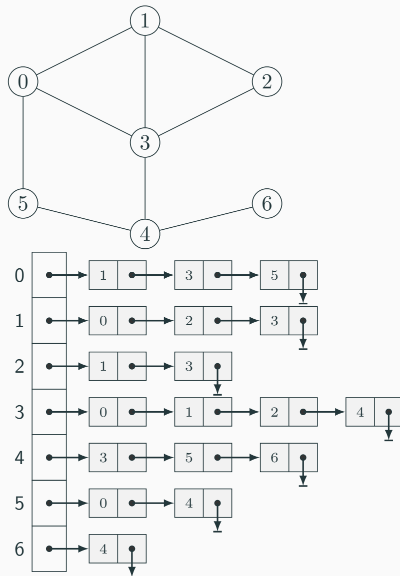
- Tempo para inserir:
- Arestas paralelas?
    -
    -



# 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 }
```

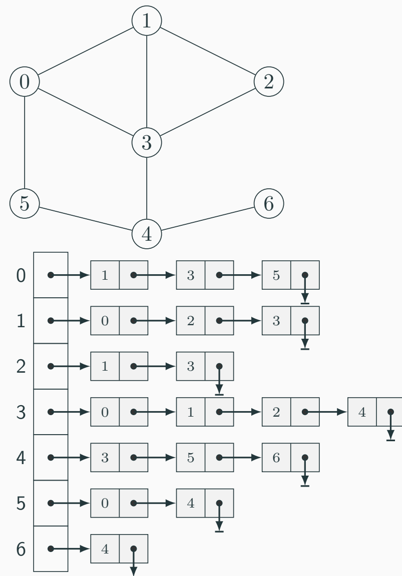
- Tempo para inserir: $\Theta(1)$
- Arestas paralelas?
    - 
    - 



# 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 }
```

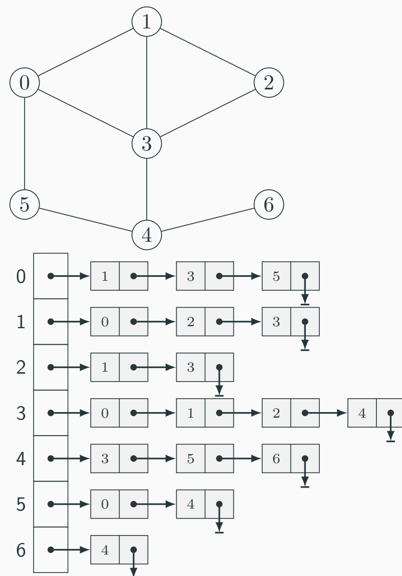
- Tempo para inserir: $\Theta(1)$
- Arestas paralelas?
  - Essa implementação não evita a inserção de arestas paralelas
  -



# Código: Inserção de Aresta

```
link list_insert(link head, int w) {
    link p = malloc(sizeof(*p));
    p->w = w;
    p->next = head;
    return p;
}

void graph_insert_edge(Graph G, Edge e) {
    G->adj[e.u] = list_insert(G->adj[e.u], e.v);
    G->adj[e.v] = list_insert(G->adj[e.v], e.u);
    G->E += 1;
}
```

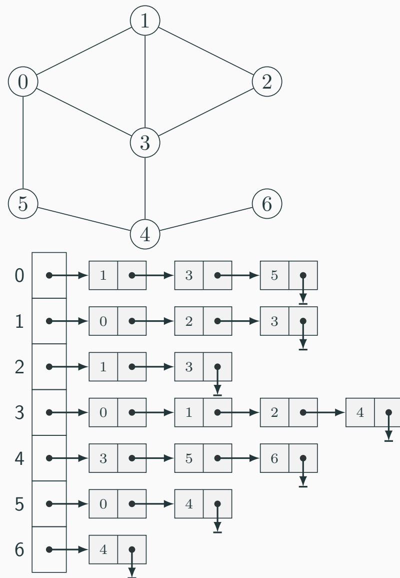
- Tempo para inserir: $\Theta(1)$
- Arestas paralelas?
  - Essa implementação não evita a inserção de arestas paralelas
  - Verificar a duplicidade de uma aresta requer tempo $O(d(u))$, ou $O(V)$



# Código: Remoção de Aresta

```
link list_remove(link head, int w) {
    if (head == NULL) return NULL;

    if (head->w == w) {
        link p = head->next;
        free(head);
        return p;
    } else {
        head->next = list_remove(head->next, w);
        return head;
    }
}

void graph_remove_edge(Graph G, Edge e) {
    if (!graph_has_edge(G, e))
        return;

    G->E -= 1;
    G->adj[e.u] = list_remove(G->adj[e.u], e.v);
    G->adj[e.v] = list_remove(G->adj[e.v], e.u);
}
```

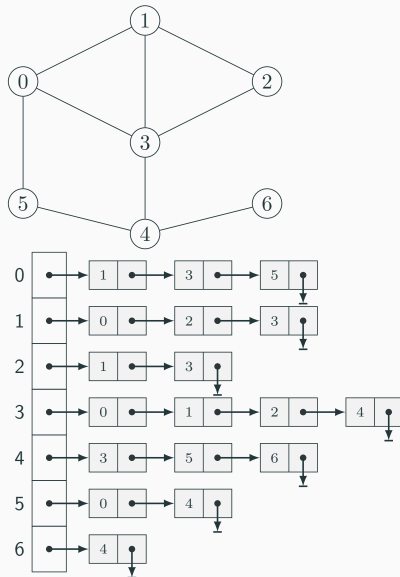
Tempo para remover uma aresta:



# Código: Remoção de Aresta

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

Tempo para remover uma aresta: $O(d(u))$, que é $O(V)$



## Lista de Adjacências para grafos esparsos

A lista de adjacências é uma representação que não é adequada pra grandes grafos densos:

- A lista requer $V + E$ bits e $V$ passos para inicialização.
- Grafo esparsos têm poucas arestas

# Sumário: Matriz vs. Lista de Adjacências

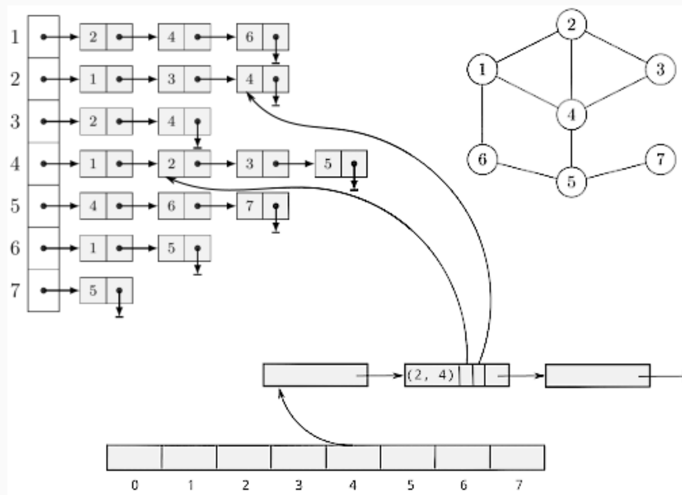
| | Matriz | Lista |
|---|---|---|
| Espaço | $O(V^2)$ | $O(V+E)$ |
| Inicialização | $O(V^2)$ | $O(V)$ |
| Verificar Pertinência | $O(1)$ | $O(d(u)) = O(V)$ |
| Inserção de Arestas | $O(1)$ | $O(1)$ |
| Remoção de Aresta | $O(1)$ | $O(d(u)) = O(V)$ |
| Copiar | $O(V^2)$ | $O(V+E)$ |
| Destruir | $O(V)$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

# Comentários

# Otimizações

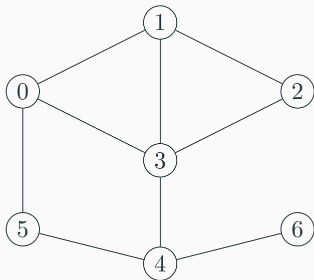
- Em vários casos, podemos modificar a representação para fazer uma operação simples mais eficiente
- Precisamos tomar cuidado pra não tornar outras operações mais custosas

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




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

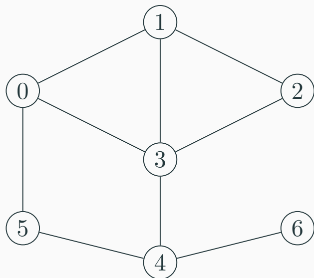
# Remoção de vértices em matriz de adjacências

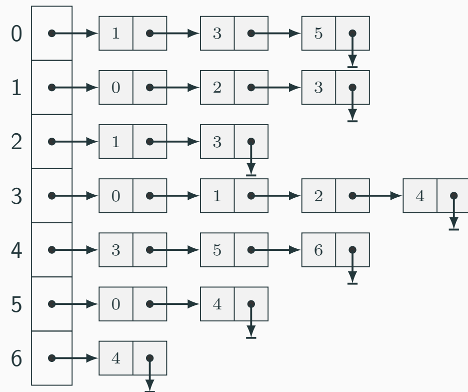



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

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

# Remoção de vértices em lista de adjacências

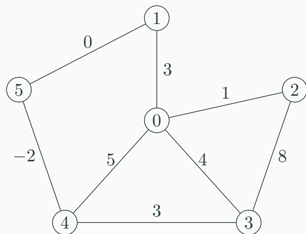


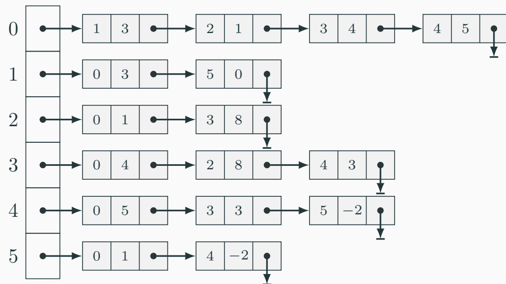

- Remover uma lista, uma entrada do vetor e $d(u)$ nós das outras listas

# Grafos ponderados

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




|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 999 | 3 | 1 | 4 | 5 | 999 |
| 1 | 3 | 999 | 999 | 999 | 999 | 0 |
| 2 | 1 | 999 | 999 | 8 | 999 | 999 |
| 3 | 4 | 999 | 8 | 999 | 3 | 999 |
| 4 | 5 | 999 | 999 | 3 | 999 | -2 |
| 5 | 999 | 0 | 999 | 999 | -2 | 999 |




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

# Adicionando propriedades aos elementos

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

# Adicionando propriedades estendendo as definições

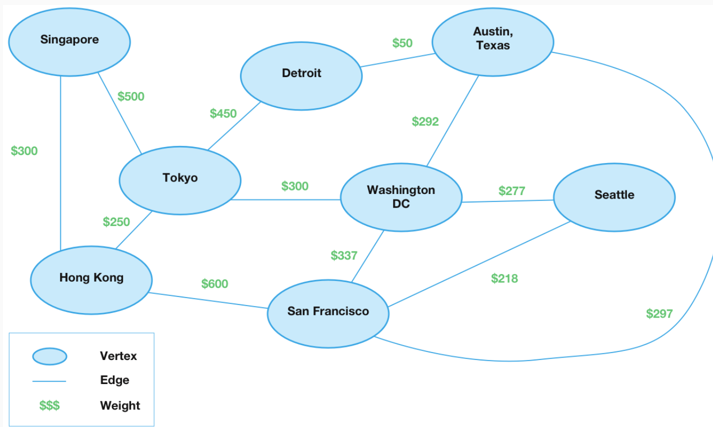
```c
// graph_matrix.c
struct data {
    int status;
    int weight;
    int color;
    char[100] label;
};

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

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

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

# Adicionando propriedades vetor externo

```
typedef struct node *link;

struct node {
    Vertex w;
    link next;
};

struct graph {
    int V;
    int E;
    link *adj;
};

// ... {

    G = graph(n);
    int **weight = squared_int_matrix(graph_order(G), 5.2);
    int *degree = malloc(graph_order(G) * sizeof(*degree));
    int *visited = malloc(graph_order(G) * sizeof(*visited));
```

# Grafos cujos rótulos dos vértices não são inteiros




```
1  SymbolTable st = ST();
2  ST_insert(st, "Singapore");
3
4  graph_insert_edge(G, edge(ST_get("Seattle"), ST_get("San Franscisco")));
```

# Aplicação

# Caminho entre dois vértices

Dados um grafo $G$ e dois vértices $u, v \in V(G)$, determinar se existe $uv$-caminho.

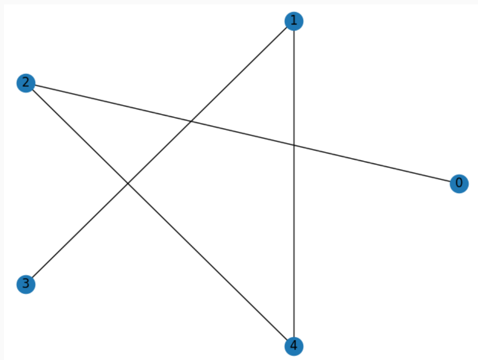
# Caminho entre dois vértices

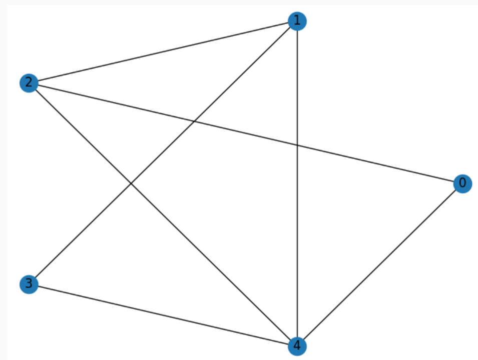
Dados um grafo $G$ e dois vértices $u, v \in V(G)$, determinar se existe $uv$-caminho.

```
1  int graph_path(Graph G, Vertex u, Vertex v) {
2      char* visited = malloc(G->V * sizeof(char));
3      for (Vertex x = 0; x < G->V; x++)
4          visited[x] = 0;
5      return path_rec(G, u, v, visited);
6  }
7
8  int path_rec(Graph G, Vertex u, Vertex v, char* visited) {
9      if (u == v)
10         return 1;
11
12     visited[u] = 1;
13
14     for (Vertex x = 0; x < G->V; x++)
15         if (graph_has_edge(G, edge(u, x)) && !visited[x])
16             if (path_rec(G, x, v, visited))
17                 return 1;
18     return 0;
19 }
```

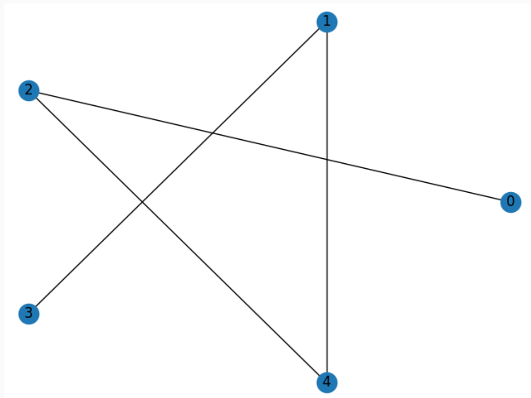
# Quadrado de um grafo

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 $\text{dist}_G(u, v) \leq 2$.



# Entrada




**Entrada**

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

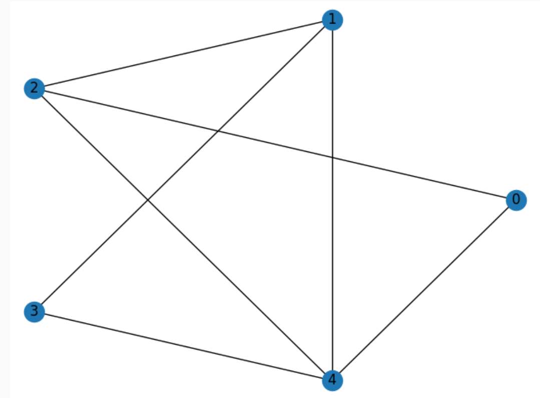
```
./graph-viewer grafo.in --layout circular
```

```
# pip install mathplotlib
# pip install networkx
import matplotlib.pyplot as plt
import networkx as nx
```

# Saída

**Saída**

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



## Programa completo

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

    Graph G2 = graph_squared(G);
    graph_print_edges(G2);

    return 0;
}
```

# Programa para geração do grafo quadrado

```
Graph graph_squared(Graph G) {
   int V = graph_order(G);
   Graph G2 = graph(V);

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

   return G2;
}
```

# Programa para impressão de arestas

```
void graph_print_edges(Graph G) {
    printf("%d %d\n", G->V, G->E);
    for (int u = 0; u < G->V; u++)
        for (int v = u + 1; v < G->V; v++)
            if (G->adj[u][v])
                printf("%d %d\n", u, v);
}
```

# Compilação

```
gcc -o g2 g2.c utils.c graph.c graph_matrix.c
```

```
gcc -o g2 g2.c utils.c graph.c graph_adjacency_list.c
```

```c
// graph.c
Edge edge(int u, int v) {
    Edge e = {u, v};
    return e;
}

void graph_insert_edges(Graph G, Edge* edges, int size) {
    for (int i = 0; i < size; i++)
        graph_insert_edge(G, edges[i]);
}
```

# Execução

```
./g2 < grafo.in
5 7
0 2
0 4
1 2
1 3
1 4
2 4
3 4
```

ou

```
./g2 < grafo.in > grafo.out
./graph-viewer grafo.out --layout circular
```