

Representações computacionais

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.

<http://github.com/malbarbo/na-grafos>

Geralmente medimos o tamanho de um grafo $G = (V, E)$ em termos do número de vértices $|V|$ e do número de arestas $|E|$.

Dentro da notação assintótica, e apenas neste caso, o termo V representará $|V|$, e o termo E , representará $|E|$.

Por exemplo, quando dissermos que o tempo de execução de um algoritmo é $O(VE)$, significa que o tempo de execução é $O(|V| \cdot |E|)$

Um grafo $G = (V, E)$ é

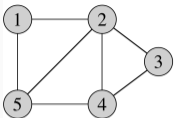
- **Esparso** se $|E|$ é muito menor que $|V|^2$
- **Denso** se $|E|$ está próximo de $|V|^2$

No exemplo do número de Erdős representamos um grafo por uma lista de arestas. Na prática esta representação não é muito utilizada (Por quê?).

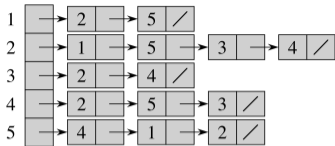
As maneiras mais comuns para representar um grafo $G = (V, E)$ são

- Lista de adjacências
- Matriz de adjacências

Exemplos



(a)

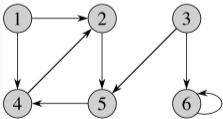


(b)

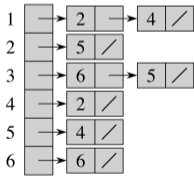
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

22-1



(a)



(b)

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

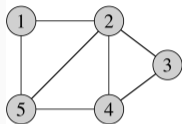
(c)

22-2

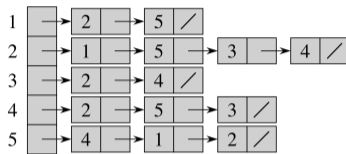
A **representação de lista de adjacências** consiste de um arranjo Adj de $|V|$ listas, uma para cada vértice.

Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém todos os vértices (ou referências para os vértices) v , tal que v é adjacente a u , isto é $(u, v) \in E$.

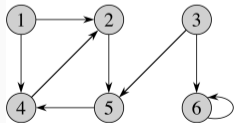
Lista de adjacências



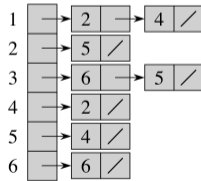
(a)



(b)



(a)



(b)

Qual é a soma dos comprimentos de todas as listas de adjacências?

- Se G é um grafo orientado?
 $|E|$
- Se G é um grafo não orientado?
 $2|E|$

Qual é a quantidade de memória requerida?

$$\Theta(V + E)$$

Vantagens

- Flexível, é possível estender a representação para multigrafos
- A quantidade de memória é assintoticamente ótima (adequada para grafos esparsos)

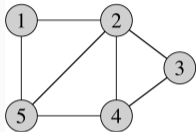
Desvantagem

- Não existe nenhum modo rápido para determinar se uma dada aresta (u, v) está presente no grafo.

Na **representação de matriz de adjacências**, usamos uma matriz $A = (a_{ij})$, de tamanho $|V| \times |V|$, tal que

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{caso contrário} \end{cases}$$

Neste caso, supomos que os vértices são numerados $1, 2, \dots, |V|$



(a)

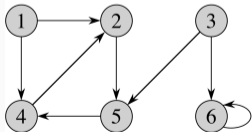
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Qual é quantidade de memória requerida?

$\Theta(v^2)$ (independe de $|E|$).

Adequada para grafos densos.



(a)

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

(c)

Em um grafo não orientado, a matriz é igual a sua transposta, desta forma é possível usar apenas os elementos abaixo (ou acima) da diagonal principal.

Vantagens

- Simplicidade
- Permite consultar se uma aresta faz parte do grafo em tempo constante

Desvantagem

- Uso excessivo da memória para grafos esparsos

Nos pseudo códigos vamos tratar o conjunto de vértices, o conjunto de arestas e a lista de adjacências um atributos do grafo:

- $G.V$ - conjunto de vértices
- $G.E$ - conjunto de arestas
- $G.Adj$ - arranjo com as listas de adjacências

Para fazer uma repetição passado por cada vértice vamos escrever:

```
1 for each vertex  $v \in G.V$   
2     ...
```

Para fazer uma repetição passando por cada vértice adjacente a um vértice u vamos escrever:

```
1 for each vertex  $v \in G.Adj[u]$   
2     ...
```

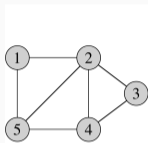
Como podemos implementar a representação de matriz de adjacências?

- Diretamente usando matriz na linguagem de programação
- Criar uma abstração para matriz transposta e economizar memória na representação de grafos não orientados

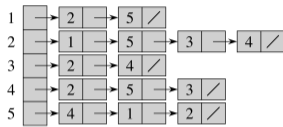
Como podemos implementar a representação de lista de adjacências?

- De muitas formas, depende da linguagem, do algoritmo, etc
- Uma forma simples é representar os vértices com inteiros no intervalo de 0 a $V - 1$ e a lista de adjacências com lista de listas (arranjo de arranjos), sem criar nenhum abstração
- Ou criar uma abstração simples para deixar o código mais legível e evitar que o grafo seja alterado de forma inconsistente
- Ou ainda usar orientação objeto e criar classes para grafo, vértice, aresta, etc.

Implementação simples



(a)



(b)

Como representar esse grafo em Python?

```
g = [  
    [1, 4],  
    [0, 4, 2, 3],  
    [1, 3],  
    [1, 4, 2],  
    [3, 0, 1],  
]
```

Como fazer uma repetição passando por cada vértice?

```
for v in range(len(g)): # v = 0, 1, 2, ..., len(g) - 1  
    ...
```

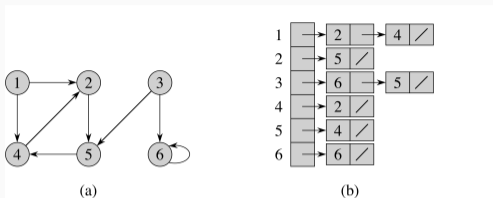
Como fazer uma repetição passando por cada vértice adjacente de um vértice u ?

```
u = 2  
for v in g[u]: # v = 1, 3  
    ...
```

Como calcular o número de arestas?

```
m = sum(len(adjacentes) for adjacentes in g) / 2
```


Implementação simples



Como representar esse grafo em C?

```
// Usamos -1 para denotar o fim de uma lista de adj
// Usamos NULL para denotar o fim da lista de adjs
int* g[7] = {
    (int[]) { 1, 3, -1 },
    (int[]) { 4, -1 },
    (int[]) { 5, 4, -1 },
    (int[]) { 1, -1 },
    (int[]) { 3, -1 },
    (int[]) { 5, -1 },
    NULL,
};
```

Como fazer uma repetição passando por cada vértice?

```
for (int v = 0; g[v] != NULL; v++) {
    // v = 0, 1, 2, ..., n - 1
    ...
}
```

E uma repetição por cada vértice adjacente de um vértice u ?

```
int u = 2;
for (int i = 0; g[u][i] != -1; i++) {
    int v = g[u][i]; // 5, 4
    ...
}
```

ou

```
int u = 2;
for (int* v = g[u]; *v != -1; v++) {
    ... *v // 5, 4
}
```

Como calcular o número de arestas? Fica como exercício.

Muitos algoritmos que operam em grafos precisam manter atributos para vértices e/ou arestas (como o número de Erdős para cada vértice).

Em pseudo código indicamos os atributos da seguinte maneira:

- $v.d$, atributo d do vértice v
- $(u, v).f$, atributo f da aresta (u, v)

Como estes atributos podem ser implementados?

- Depende da linguagem de programação, algoritmo, etc
- Os atributos da arestas podem ser armazenados diretamente na lista ou matriz de adjacência
- Se os vértices são enumerados de $1..|V|$ os atributos podem ser representados em arranjos, tal como $d[1..|V|]$
- Atributos de vértices podem ficar nos registros que representam os vértices
- Atributos de arestas podem ficar nos registros que representam as arestas

Veja na página da disciplina.

22.1-1) Dada uma representação de lista de adjacências de um grafo orientado, qual o tempo necessário para computar o grau de saída de todo o vértice? Qual o tempo necessário para computar os graus de entrada?

Antes de fazer a análise do tempo de execução é necessário escrever o pseudo código do algoritmo.

GRAUS-DE-SAIDA(G)

```
1 for each vertex  $v \in G.V$ 
2    $v.grau-de-saida = 0$ 
3 for each vertex  $u \in G.V$ 
4   for each vertex  $v \in G.Adj[u]$ 
5      $u.grau-de-saida = u.grau-de-saida + 1$ 
```

Análise do tempo de execução

- O laço das linhas 1 a 2 tem tempo de execução $\Theta(V)$
- A cada interação do laço da linha 3, o laço das linha 4 a 5 é executado $|G.Adj[u]|$ vezes, como o laço das linha 4 a 5 é executado uma vez para cada vértice, temos que seu tempo de execução é $\sum_{u \in G.V} |G.Adj[u]| = |E|$. Ou seja, o tempo de execução das linhas 3 a 5 é $\Theta(V + E)$
- Portanto, o tempo de execução do procedimento GRAUS-DE-SAIDA é $\Theta(V + E)$

GRAUS-DE-ENTRADA(G)

1 **for** each vertex $u \in G.V$

2 $u.grau-de-entrada = 0$

3 **for** each vertex $u \in G.V$

4 **for** each vertex $v \in G.Adj[u]$

5 $v.grau-de-entrada = v.grau-de-entrada + 1$

Análise do tempo de execução

- Mesmo do procedimento GRAUS-DE-SAIDA

Thomas H. Cormen et al. Introduction to Algorithms. 3rd edition. Capítulo 22.1.