

Capítulo II

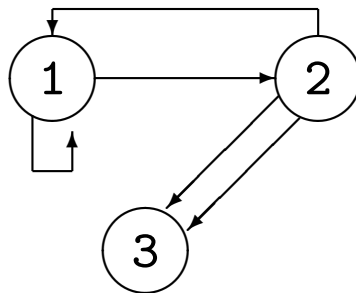
Noções Básicas de Grafos

Grafo $G = (V, A)$

V conjunto de **vértices** ou **nós**

A coleção de **arcos** ou **arestas**

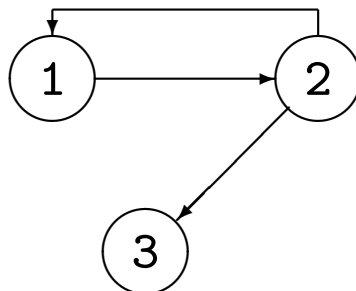
Grafo Genérico — Com arcos paralelos ou com lacetes.



$$V = \{1, 2, 3\}$$

$$A = \langle (1, 1), (1, 2), (2, 1), (2, 3), (2, 3) \rangle$$

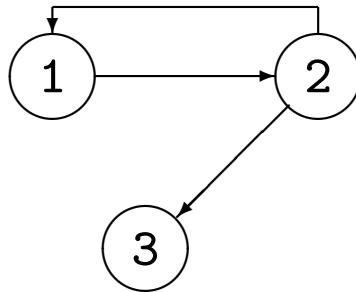
Grafo Simples — $A \subseteq V \times V$ e sem lacetes.



$$V = \{1, 2, 3\}$$

$$A = \{(1, 2), (2, 1), (2, 3)\}$$

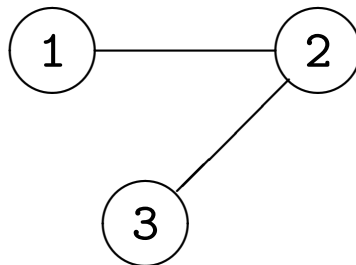
Grafo Orientado — Os arcos têm sentido.



$$V = \{1, 2, 3\}$$

$$A = \{(1, 2), (2, 1), (2, 3)\}$$

Grafo Não Orientado — Os arcos não têm sentido único.



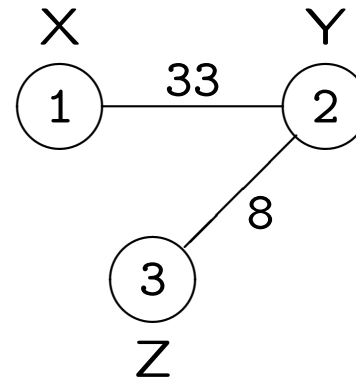
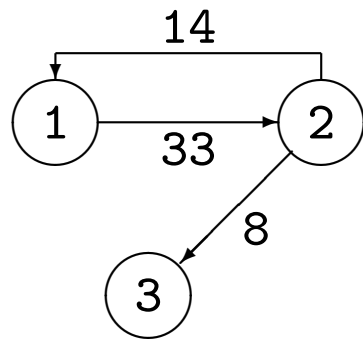
$$V = \{1, 2, 3\}$$

$$A = \{(1, 2), (2, 3)\}$$

Considere-se que $(\forall v, w \in V) (v, w) = (w, v)$.

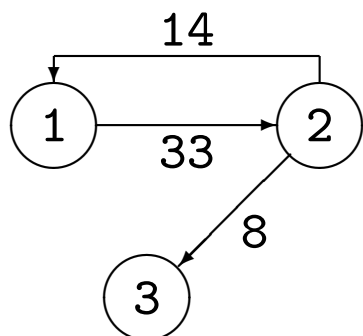
Grafo Etiquetado (ou Pesado)

Os vértices, os arcos ou ambos têm uma **etiqueta**, um **peso** ou um **custo**.



Caminho

É uma sequência não vazia de vértices $v_1, v_2, \dots, v_n$ (com $n \geq 1$), tal que, para qualquer $i = 1, 2, \dots, n - 1$: $(v_i, v_{i+1}) \in A$.



Caminho: 2, 1, 2, 3

Comprimento: 3

Comprimento Pesado: 55

Comprimento do Caminho: o número de arcos do caminho ($n - 1$).

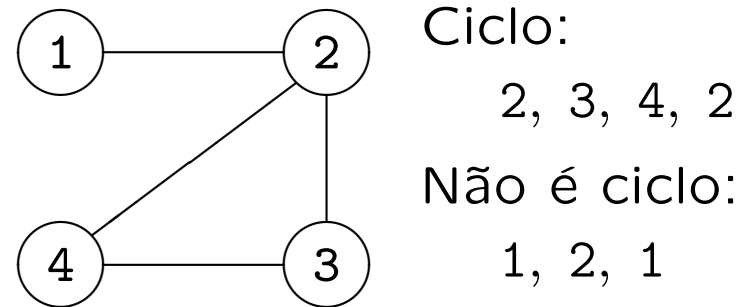
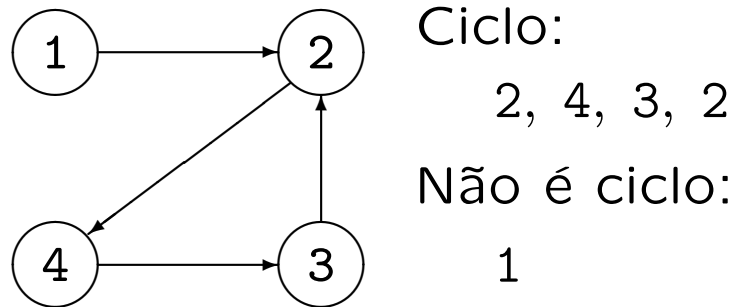
Comprimento Pesado ou Custo do Caminho: a soma dos pesos (numéricos) dos arcos do caminho (num grafo pesado).

Caminho Simples: um caminho cujos vértices são todos diferentes, exceto, possivelmente, o primeiro e o último.

Ciclo ou Circuito

Num Grafo Orientado: um caminho de comprimento positivo onde o primeiro e o último vértices são iguais.

Num Grafo Não Orientado: um caminho de comprimento positivo, onde o primeiro e o último vértices são iguais, que não passa 2 vezes pelo mesmo arco.



Grafo Cíclico / Acíclico: um grafo com / sem ciclos.

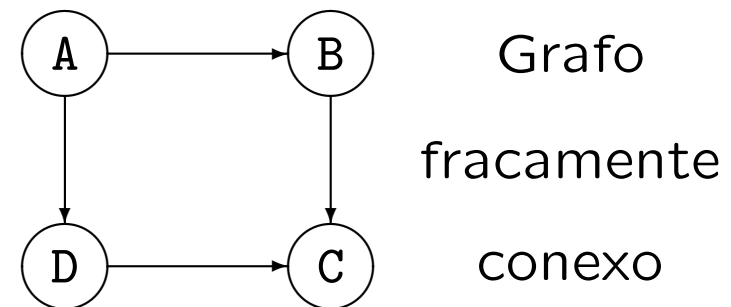
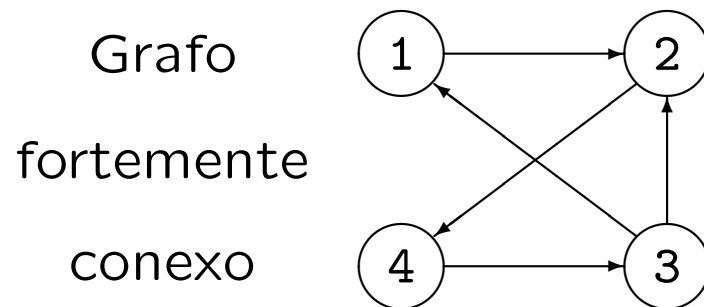
Conectividade

Grafo Fortemente Conexo: um **grafo orientado** tal que:

$(\forall v, w \in V)$ existe um caminho de v para w .

Grafo Fracamente Conexo: um **grafo orientado** tal que, ignorando o sentido dos arcos:

$(\forall v, w \in V)$ existe um caminho de v para w .



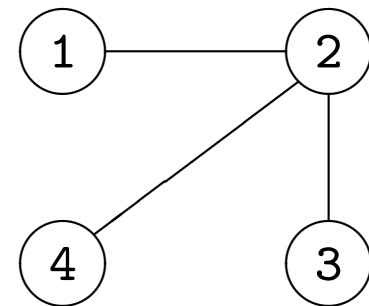
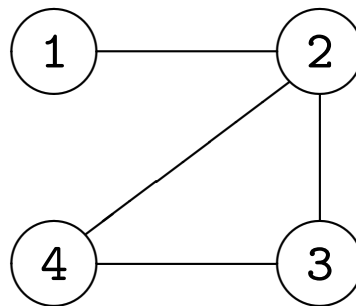
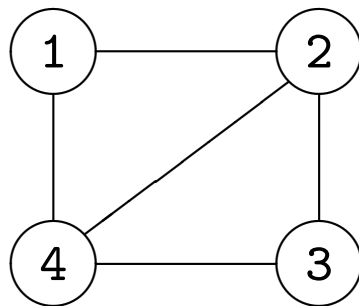
Grafo Conexo: um **grafo não orientado** tal que:

$(\forall v, w \in V)$ existe um caminho de v para w .

Sub-grafos e Árvores

Sub-grafo de (V, A) : um grafo (V', A') tal que $V' \subseteq V$ e $A' \subseteq A$.

Sub-grafo de Cobertura de (V, A) : um sub-grafo (V', A') de (V, A) , com $V' = V$.



Árvore (livre): um **grafo não orientado**, conexo e acíclico.

Árvore de Cobertura de (V, A) : Um sub-grafo de cobertura de (V, A) que é árvore.

Tipos Abstratos de Dados

Vértice, Arco, Grafo Não Orientado e Grafo Orientado

- Os cinco slides que se seguem introduzem os métodos usados nos slides das aulas teóricas.
- Por questões de eficiência, **não implementem** as interfaces correspondentes.
- Por exemplo, em geral, um vértice é um número inteiro (entre zero e número-total-de-vértices – 1), não havendo interface nem classe para os vértices.

ADT Node

// In practice, a node is an integer.

ADT Edge<L> (whose label is of type L)

// Returns the edge label.

L **label**();

// Returns the first endpoint of the edge,

// which is its origin if the edge is directed.

Node **firstNode**();

// Returns the second endpoint of the edge,

// which is its destination if the edge is directed.

Node **secondNode**();

// Returns the edge endpoint that is distinct from the specified node.

Node **oppositeNode**(Node node);

ADT AnyGraph<L>

// Returns the number of nodes.

int numNodes();

// Returns the nodes.

Iterable<Node> **nodes**();

// Returns the number of edges.

int numEdges();

// Returns the edges.

Iterable<Edge<L>> **edges**();

// Returns an arbitrary node.

Node **aNode**();

// Inserts edge (node1, node2) whose label is the specified label.

void addEdge(Node node1, Node node2, L label);

// Returns true iff there is an edge of the form (node1, node2).

boolean edgeExists(Node node1, Node node2);

ADT UndiGraph<L> (extends AnyGraph<L>)

// Returns the degree of the specified node.

int **degree**(Node node);

// Returns the nodes adjacent to the specified node.

Iterable<Node> **adjacentNodes**(Node node);

// Returns the edges incident on the specified node.

Iterable<Edge<L>> **incidentEdges**(Node node);

ADT Digraph<L> (extends AnyGraph<L>)

// Returns the in-degree of the specified node.

int **inDegree**(Node node);

// Returns the out-degree of the specified node.

int **outDegree**(Node node);

// Returns the nodes adjacent to the specified node along incoming
// edges to it.

Iterable<Node> **inAdjacentNodes**(Node node);

// Returns the nodes adjacent to the specified node along outgoing
// edges from it.

Iterable<Node> **outAdjacentNodes**(Node node);

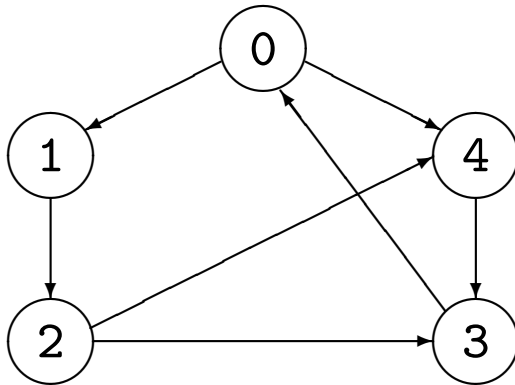
// Returns the incoming edges to the specified node.

Iterable<Edge<L>> **inIncidentEdges**(Node node);

// Returns the outgoing edges from the specified node.

Iterable<Edge<L>> **outIncidentEdges**(Node node);

Matriz de Adjacências



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

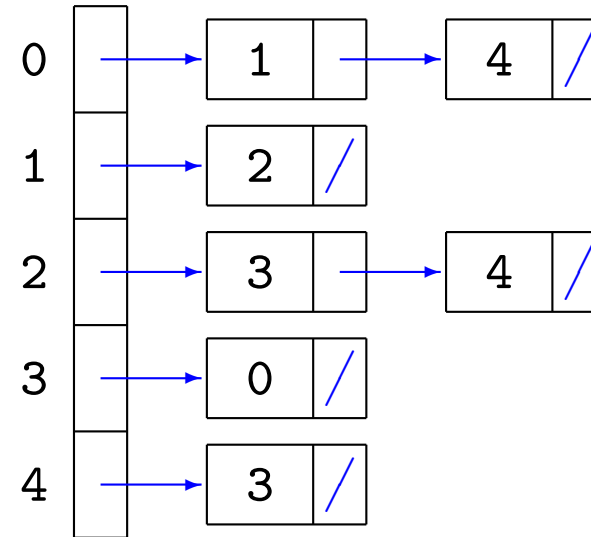
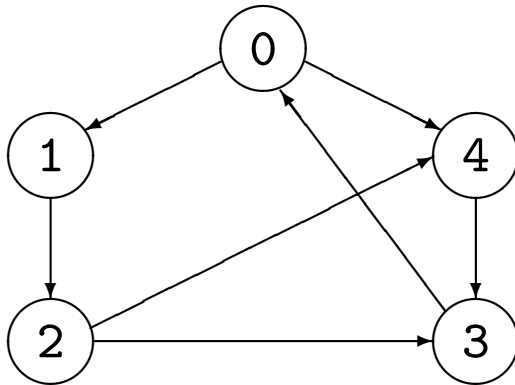
Pesquisar Arco (v_1, v_2) $\Theta(1)$

Obter Sucessores (diretos) v $\Theta(|V|)$

Antecessores (diretos) v $\Theta(|V|)$

Memória Requerida $\Theta(|V|^2)$

Listas Ligadas de Adjacências de Sucessores (diretos)



Pesquisar Arco (v_1, v_2) $O(|\text{Suc}(v_1)|)$

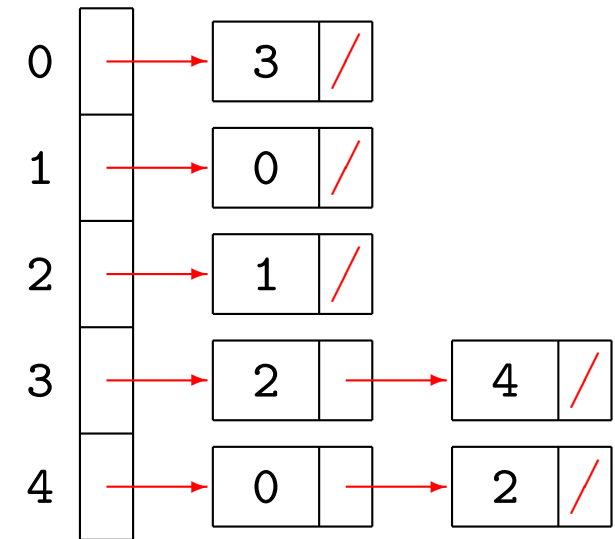
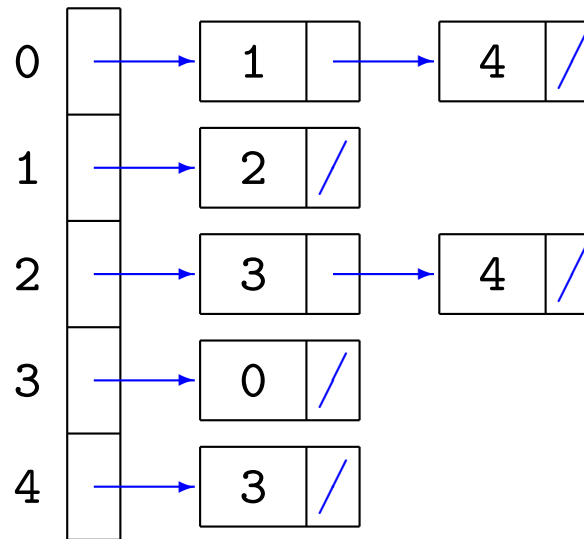
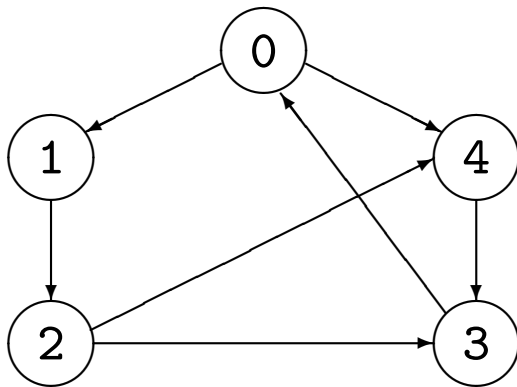
Obter Sucessores (diretos) v $\Theta(|\text{Suc}(v)|)$

Antecessores (diretos) v $O(|V| + |A|)$

Memória Requerida $\Theta(|V| + |A|)$

Listas Ligadas de Adjacências

de Sucessores e de Antecessores



Pesquisar Arco (v_1, v_2) $O(\min(|\text{Suc}(v_1)|, |\text{Ant}(v_2)|))$

Obter Sucessores v $\Theta(|\text{Suc}(v)|)$

Antecessores v $\Theta(|\text{Ant}(v)|)$

Memória Requerida $\Theta(|V| + |A|)$

Exemplo de Tradução do Pseudo-código (1)

```
int algorithm( Digraph graph, Node source ) {  
    for every Node v in graph.nodes()  
        ( ... )  
    for every Node v in graph.outAdjacentNodes(source)  
        ( ... )  
    return ...  
}
```

- **Quais são as operações sobre o grafo?** Percorrem-se os nós e percorrem-se os sucessores (diretos) de um nó arbitrário.
- **Quantos nós e quantos arcos pode ter o grafo?** Entre 2 e 100 000 nós; entre 1 e 500 000 arcos.
- **Como se obtém a informação sobre os nós e os arcos do grafo?** Inicialmente, sabe-se o número de nós; depois, conhecem-se os arcos por uma ordem qualquer.
- **DECIDIR como guardar o grafo e TRADUZIR o pseudo-código de acordo com essa implementação.**

Exemplo de Tradução do Pseudo-código (2)

```
int algorithm( Digraph graph, Node source ) {  
    for every Node v in graph.nodes()  
        ( ... )  
    for every Node v in graph.outAdjacentNodes(source)  
        ( ... )  
    return ...  
}
```

```
class Problem {  
    private int numNodes;  
    private List<Integer>[] edges;
```

**Grafo
implementado
em vetor
de listas ligadas
de adjacências
de sucessores**

```
public int algorithm( int source ) {  
    for ( int v = 0; v < numNodes; v++ )  
        ( ... )  
    for ( int v : edges[source] )  
        ( ... )  
    return ...  
}  
}
```