Capítulo II

Noções Básicas

de

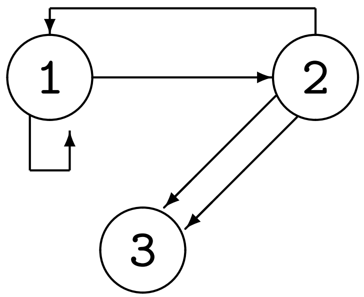
Grafos

Grafo $G = (V, A)$

V conjunto de **vértices** ou **nós**

A coleção de **arcos** ou **arestas**

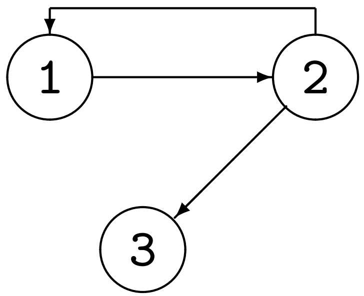
Grafo Genérico — Com arcos paralelos ou com lacetes.



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

$A = <(1,1),\ (1,2),\ (2,1),\ (2,3),\ (2,3)>$

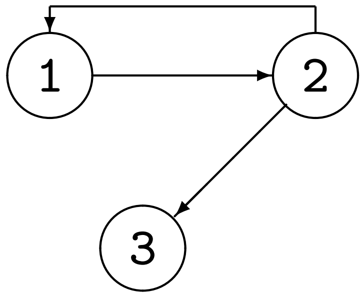
Grafo Simples — $A \subseteq V \times V$ e sem lacetes.



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

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

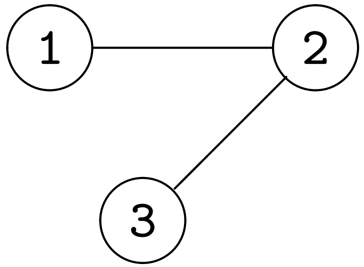
Grafo Orientado — Os arcos têm sentido.



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

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

Grafo Não Orientado — Os arcos não têm sentido único.



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

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

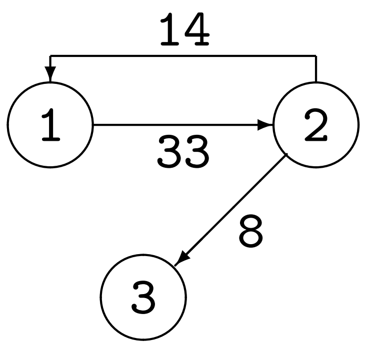
Condidera-se que $(\forall v, w \in V)\ \ (v,w) = (w,v)$.

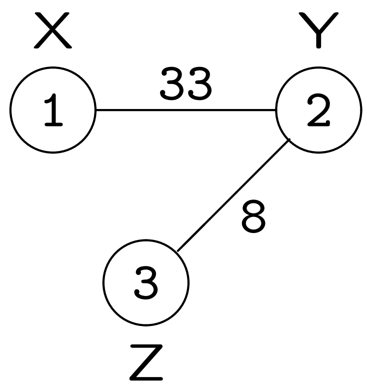
Grafo Etiquetado (ou Pesado)

—

Os vértices, os arcos ou ambos têm

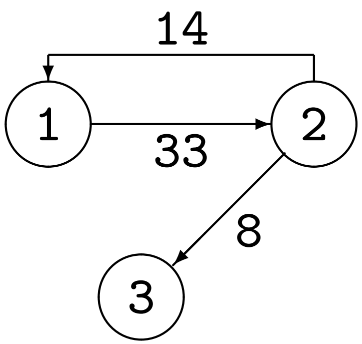
uma **etiqueta**, um **peso** ou um **custo**.





Caminho

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



Caminho: 2, 1, 2, 3

Comprimento: 3

Comprimento Pesado: 55

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

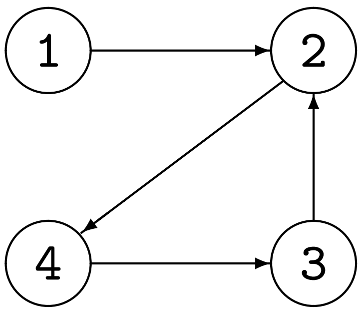
Comprimento Pesado ou Custo do Caminho: a soma dos pesos (numéricos) dos arcos do caminho (num grafo pesado).

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.

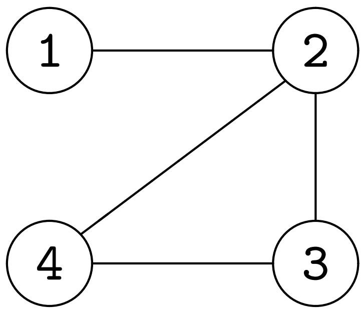


Ciclo:

2, 4, 3, 2

Não é ciclo:

1



Ciclo:

2, 3, 4, 2

Não é ciclo:

1, 2, 1

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

Conectividade

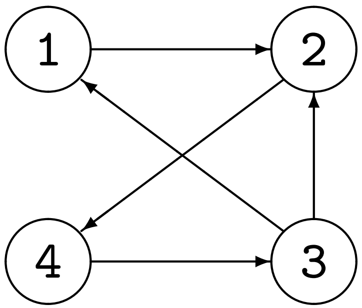
Grafo Fortemente Conexo: um **grafo orientado** tal que:

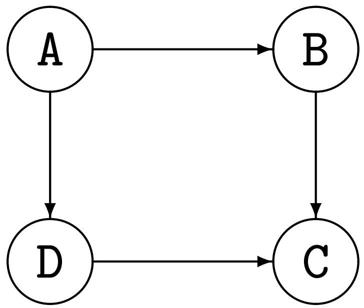
$$(\forall v, w \in V) \text{ existe um caminho de } v \text{ para } w.$$

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

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

Grafo fortemente conexo





Grafo fracamente conexo

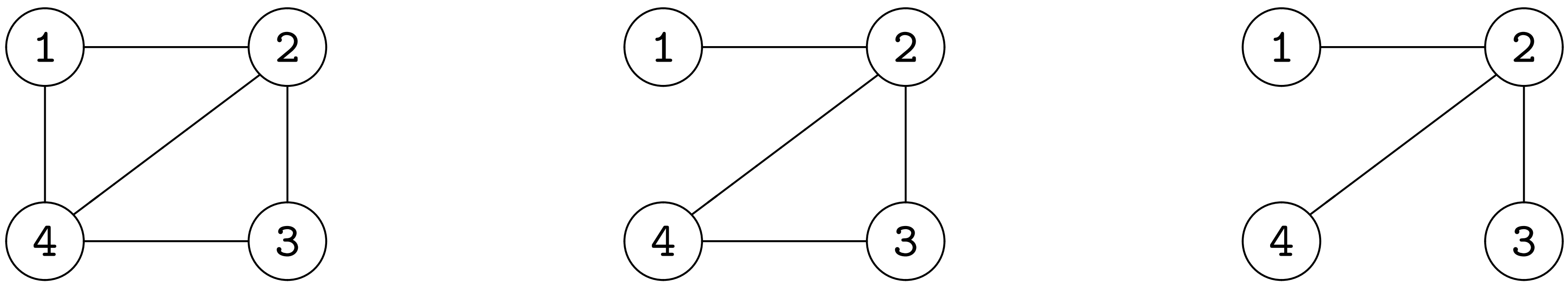
Grafo Conexo: um **grafo não orientado** tal que:

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

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

- As interfaces que se seguem introduzem os métodos usados nos slides das aulas teóricas.

- Por questões de eficiência, **não implementem** estas interfaces.

- 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.

TAD Vértice

```
public interface Node
{
    // In practice, a node is an integer.
}
```

TAD Arco (com etiqueta do tipo L)

```
public interface Edge<L>
{
   // Returns the edge label.
   L label( );

   // Returns an array of length 2 with the edge end-nodes.
   Node[] endNodes( );

   // Returns the edge end-node that is distinct from the specified
   // node.
   Node oppositeNode( Node node );
}
```

TAD Qualquer Grafo (L) (1)

```
public interface AnyGraph<L>
{
   // Returns the number of nodes.
   int numNodes( );

   // Returns the number of edges.
   int numEdges( );

   // Returns the nodes.
   Iterable<Node> nodes( );

   // Returns the edges.
   Iterable<Edge<L>> edges( );
```

TAD Qualquer Grafo (L) (2)

```
// Returns an arbitrary node.
Node aNode( );

// Inserts the edge (node1, node2) and associates it with 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 );

}
```

TAD Grafo Não Orientado (L)

```
public interface 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 upon the specified node.
   Iterable<Edge<L>> incidentEdges( Node node );

}
```

TAD Grafo Orientado (L) (1)

```
public interface 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 );
```

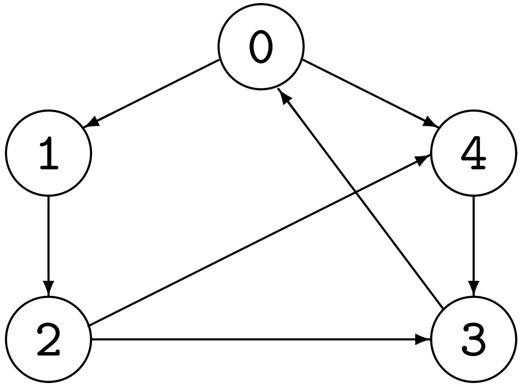
TAD Grafo Orientado (L) (2)

```
    // 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

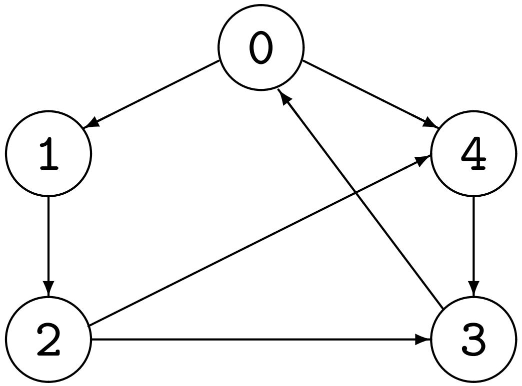


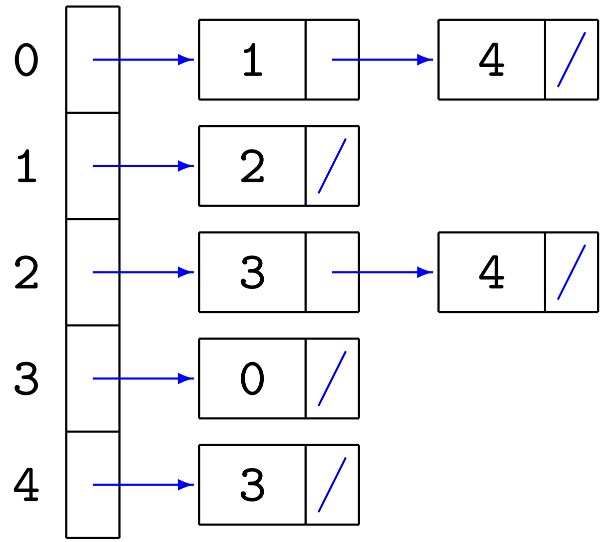
$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 \\ 3 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

Pesquisar	Arco (v_1, v_2)	$\Theta(1)$
Obter	Sucessores v	$\Theta(\lvert V \rvert)$
	Antecessores v	$\Theta(\lvert V \rvert)$
Memória Requerida		$\Theta(\lvert V \rvert^2)$

Listas Ligadas de Adjacências

de Sucessores (diretos)

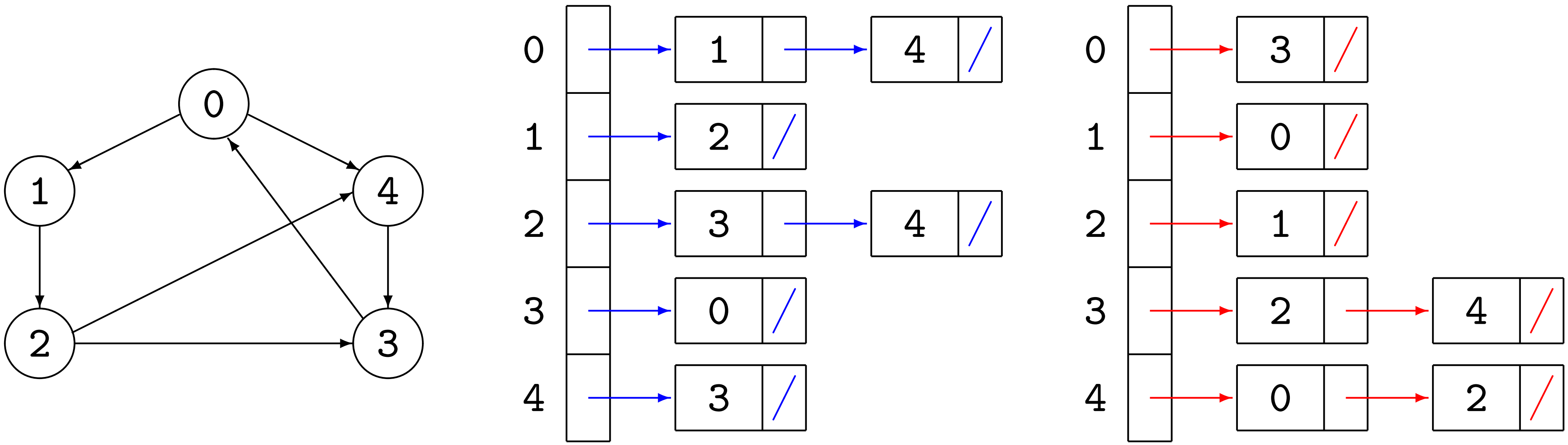




Pesquisar	Arco (v_1, v_2)	$O(\lvert \mathrm{Suc}(v_1) \rvert)$
Obter	Sucessores v	$\Theta(\lvert \mathrm{Suc}(v) \rvert)$
	Antecessores v	$O(\lvert V \rvert + \lvert A \rvert)$
Memória Requerida		$\Theta(\lvert V \rvert + \lvert A \rvert)$

Listas Ligadas de Adjacências

de Sucessores e de Antecessores



Pesquisar	Arco (v_1, v_2)	$O($ min$(\lvert\text{Suc}(v_1)\rvert, \lvert\text{Ant}(v_2)\rvert)$ $)$
Obter	Sucessores v	$\Theta(\lvert\text{Suc}(v)\rvert)$
	Antecessores v	$\Theta(\lvert\text{Ant}(v)\rvert)$
Memória Requerida		$\Theta(\lvert V\rvert + \lvert A\rvert)$

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 ...
}
```

Grafo implementado em vetor de listas ligadas de adjacências de sucessores

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

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