Estruturas de Dados
2018/2019

Aula 12

- Grafos
- Redes
- Travessias
- Árvore Geradora
- Caminho Mais Curto

Ricardo Santos
rjs@estg.ipp.pt

ESTG POLITÉCNICO DO PORTO

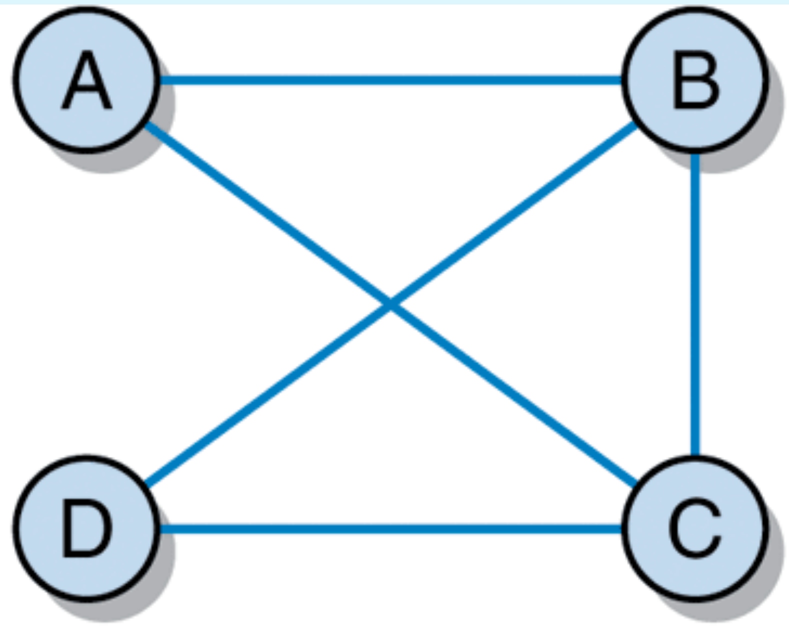
Grafos

- Tal como uma árvore, um grafo é constituído por nós e conexões entre os nós
- Na terminologia de grafos, referimo-nos aos nós como vértices e às ligações como arestas
- Os vértices são normalmente referenciados pelo rótulo (ex. **A**, **B**, **C**, **D**)
- As arestas são referenciadas por uma relação de vértices (por exemplo (**A**, **B**) que representa uma aresta entre **A** e **B**)

Grafos não direccionados

+ Um grafo não direccionado é um grafo onde os pares de vértices que representam as arestas não estão ordenados
+ Ao listar uma aresta como (**A**, **B**) significa que existe uma aresta entre **A** e **B** que pode ser percorrida em qualquer direcção
+ Para um grafo não direccionado, (**A**, **B**) significa exactamente a mesma coisa que (**B**, **A**)

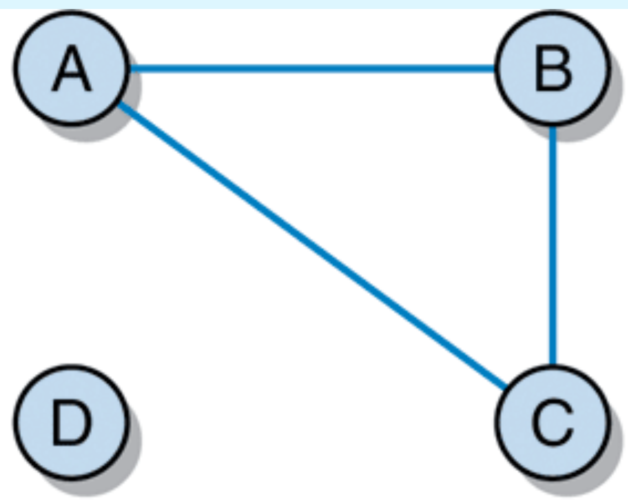
Exemplo de grafo não direccionado



- Dois vértices num grafo são adjacentes se existir uma aresta a ligá-los
- Vértices adjacentes são muitas vezes referidos como vizinhos
- Uma aresta de um grafo que liga um vértice a ele próprio é chamado de laço
- Um grafo não direccionado é considerado completo se tiver o número máximo de arestas a ligar os vértices **(n (n-1) / 2)**

- Um caminho é uma sequência de arestas que liga dois vértices num grafo
  - **A**, **B**, **D** é um caminho de **A** a **D**, no nosso exemplo anterior
- O comprimento de um caminho é o número de arestas no caminho (número de vértices - 1)
- Um grafo não direccionado é considerado conexo se para quaisquer dois vértices do grafo, existir um caminho entre eles
  - O gráfico no exemplo anterior é conexo
  - O gráfico apresentado de seguida não é conexo

Exemplo de grafo não direccionado não conexo



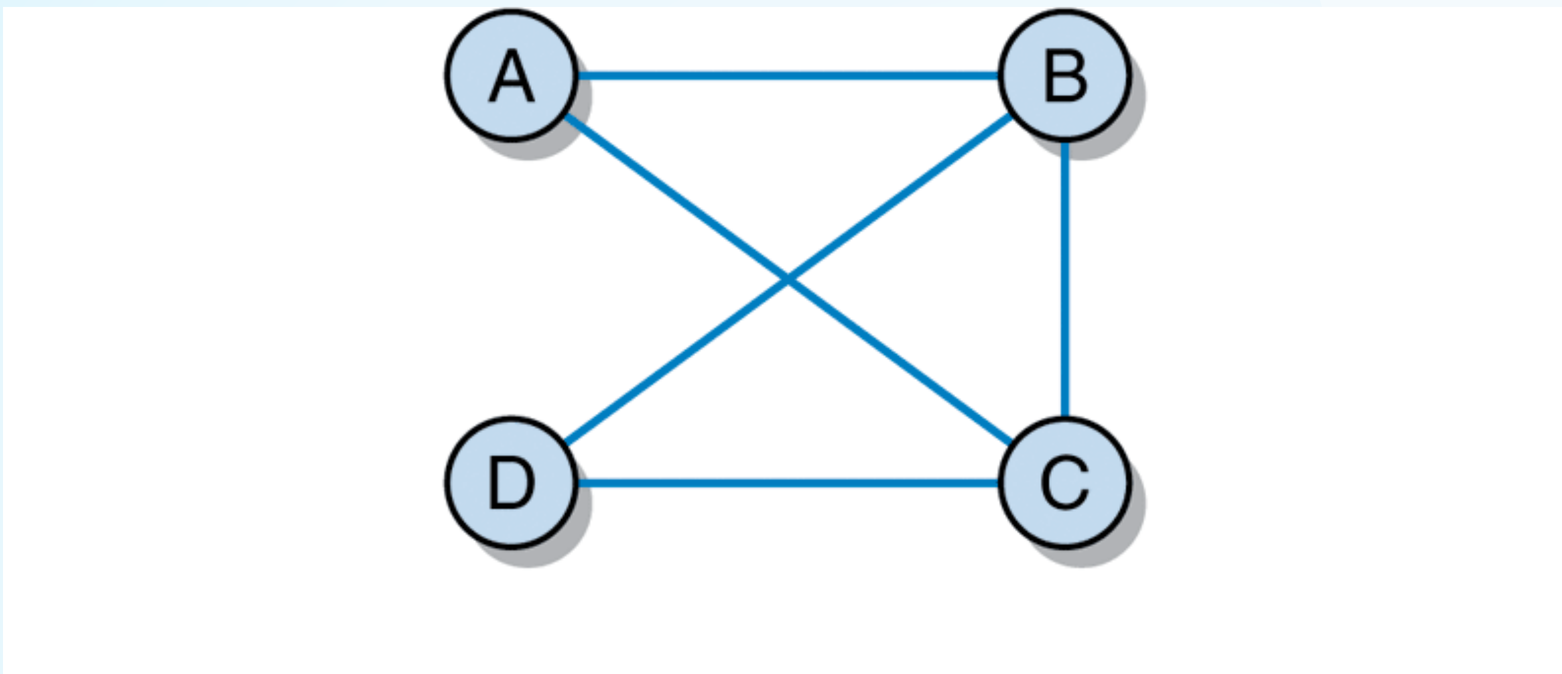
- Um circuito é um caminho em que o primeiro e último vértice são repetidos
- Um circuito simples é um circuito em que todos os vértices aparecem no máximo uma vez, à excepção do primeiro e últimos vértices

- Por exemplo, no slide anterior, **A**, **B**, **C**, **A** é um circuito

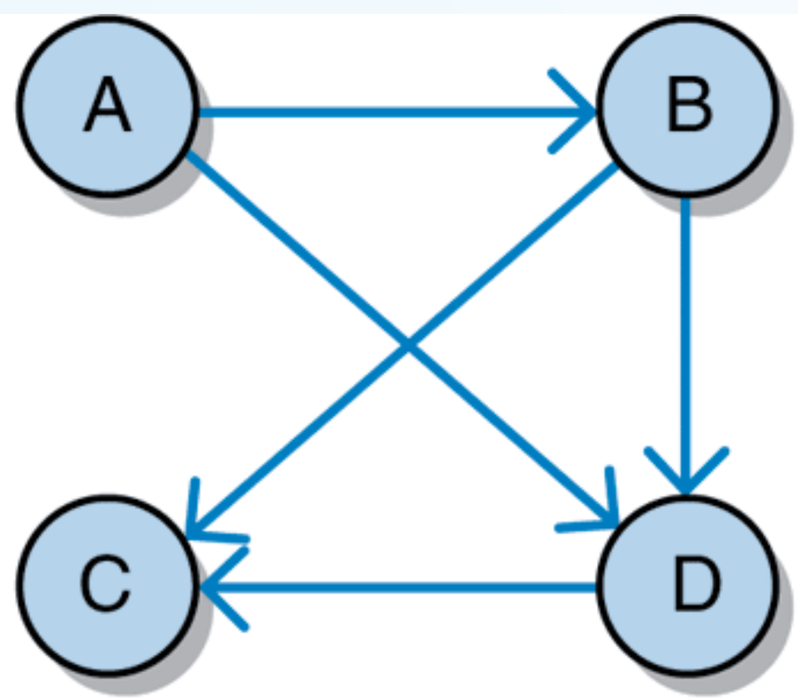
Grafos Direccionados

- Um grafo direccionado ou dirigido ou dígrafo, é um grafo onde as arestas são pares ordenados de vértices
- Isto significa que a aresta (**A**, **B**) e (**B**, **A**) são arestas direccionadas separadas

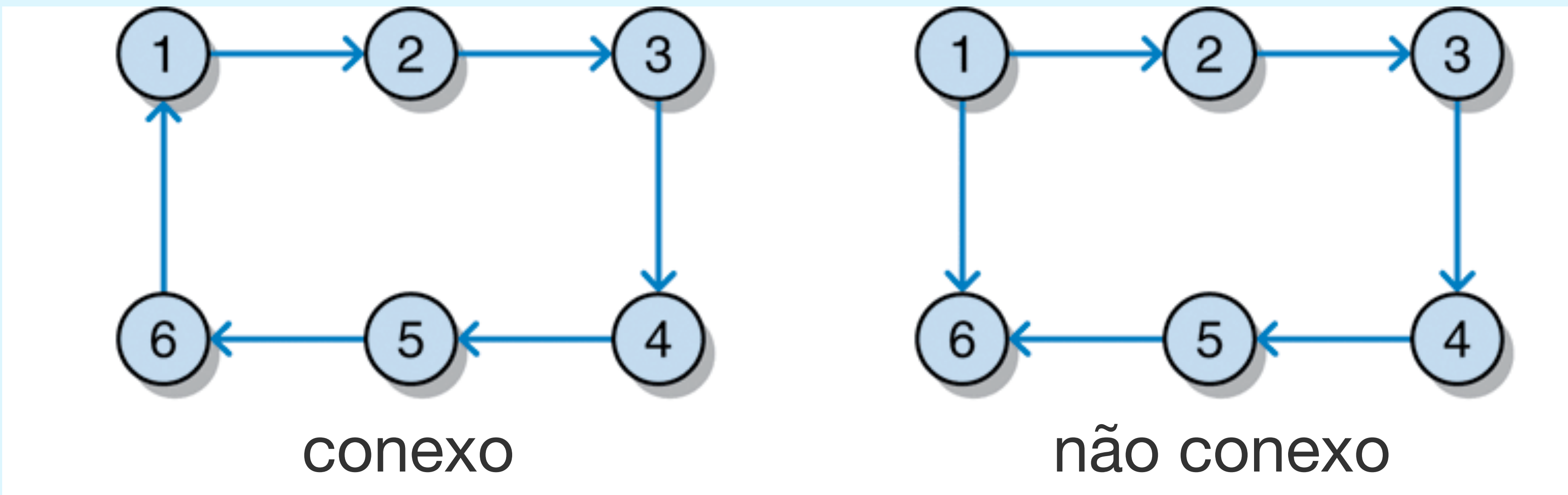
+ **O grafo:**



- É interpretado da seguinte forma:
  - Vértices: **A**, **B**, **C**, **D**
  - Arestas: (**A**, **B**), (**A**, **C**), (**B**, **C**), (**B**, **D**), (**C**, **D**)
- Por vezes podemos ter algo diferente em mente, como o exemplo de grafo direccionado apresentado de seguida



Grafos direccionados conexos e não conexos

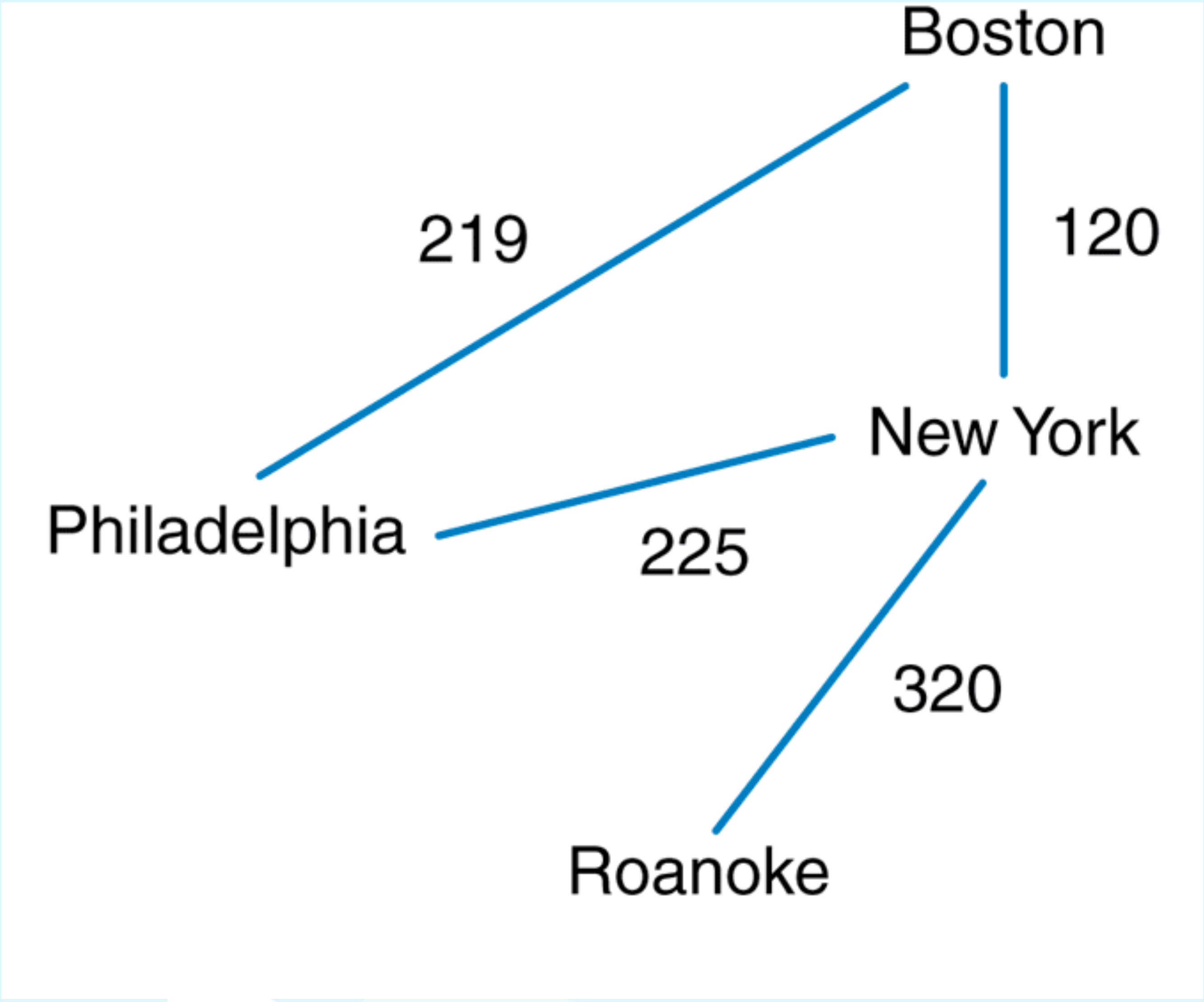


- Se um grafo não tem ciclos, é possível organizar os vértices tal que o vértice **A**, precede o vértice **B** se existe uma aresta de **A** para **B**
- Esta ordem dos vértices é denominada de ordem topológica
- Uma árvore direccionada é um grafo direccionado que tem um elemento designado como a raiz e tem as seguintes propriedades
  - Não existem ligações de outros vértices para a raiz
  - Cada elemento não-raiz tem exactamente uma ligação à raiz
  - Há um caminho da raiz para todos os outros vértices

Redes

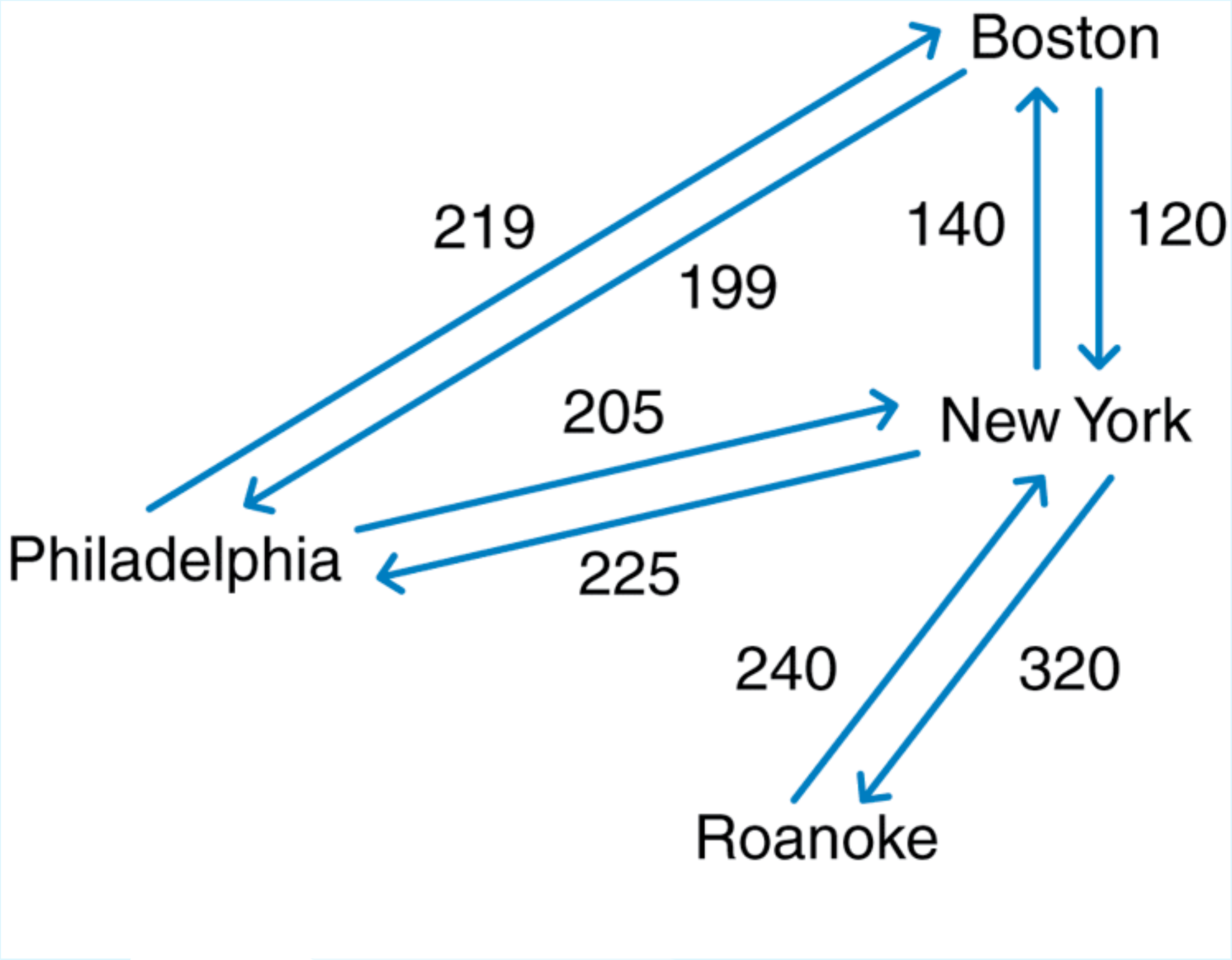
- Uma rede ou um grafo pesado é um grafo com pesos ou custos associados a cada aresta
- Podemos ver na figura seguinte uma rede não direccionada de conexões que exemplifica passagens aéreas entre cidades

Rede ou grafo pesado



Rede direccionada

+ As redes podem ser direccionadas



ESTG POLITÉCNICO DO PORTO

- Para redes, que representamos cada aresta com um tripleto incluindo o vértice de partida, o vértice final, e o peso
- **(Boston, Nova York, 120)**

Algoritmos comuns para grafos

- No caso das árvores, foram definidos quatro tipos de travessias
- Geralmente, as travessias para grafos são divididas em duas categorias:
  - travessia em largura (*breadth-first traversal*)
  - travessia em profundidade (*depth-first traversal*)

- Podemos implementar uma travessia em largura (*breadth-first traversal*) de um grafo semelhante à nossa travessia nível-ordem (*level-order*) de uma árvore
- Para isso:
  - usar uma fila e uma lista não-ordenada
  - usar a fila para gerir a travessia
  - usar a lista não-ordenada para construir o nosso resultado

Implementação das travessias

+ De seguida é apresentado o iterador para a travessia em largura (*breadth-first iterator*)

```
/**
 * Returns an iterator that performs a breadth first search
 * traversal starting at the given index.
 *
 * @param startIndex  the index to begin the search from
 * @return  an iterator that performs a breadth first traversal
 */
public Iterator<T> iteratorBFS(int startIndex) {
   Integer x;
   LinkedQueue<Integer> traversalQueue = new LinkedQueue<Integer>();
   ArrayUnorderedList<T> resultList = new ArrayUnorderedList<T>();

   if (!indexIsValid(startIndex))
      return resultList.iterator();

   boolean[] visited = new boolean[numVertices];
   for (int i = 0; i < numVertices; i++)
      visited[i] = false;

   traversalQueue.enqueue(new Integer(startIndex));
   visited[startIndex] = true;
```

```
while (!traversalQueue.isEmpty())
      {
         x = traversalQueue.dequeue();
         resultList.addToRear(vertices[x.intValue()]);

         /** Find all vertices adjacent to x that have
             not been visited and queue them up */
         for (int i = 0; i < numVertices; i++)
         {
            if (adjMatrix[x.intValue()][i] && !visited[i])
            {
               traversalQueue.enqueue(new Integer(i));
               visited[i] = true;
            }
         }
      }
      return resultList.iterator();
   }
```

- Podemos também implementar uma travessia em profundidade para um grafo semelhante à nossa travessia nível-ordem (*level-order*) de uma árvore substituindo a fila por uma pilha
- Para isso:
  - usar uma pilha e uma lista não-ordenada
  - usar a pilha para gerir a travessia
  - usar a lista não-ordenada para construir o nosso resultado
- De seguida é apresentada o iterador para a travessia em profundidade (*depth-first iterator*)

```java
/**
 * Returns an iterator that performs a depth first search
 * traversal starting at the given index.
 *
 * @param startIndex  the index to begin the search traversal from
 * @return       an iterator that performs a depth first traversal
 */
public Iterator<T> iteratorDFS(int startIndex)
{
   Integer x;
   boolean found;
   LinkedStack<Integer> traversalStack = new LinkedStack<Integer>();
   ArrayUnorderedList<T> resultList = new ArrayUnorderedList<T>();
   boolean[] visited = new boolean[numVertices];

   if (!indexIsValid(startIndex))
      return resultList.iterator();

   for (int i = 0; i < numVertices; i++)
      visited[i] = false;

   traversalStack.push(new Integer(startIndex));
   resultList.addToRear(vertices[startIndex]);
   visited[startIndex] = true;
```

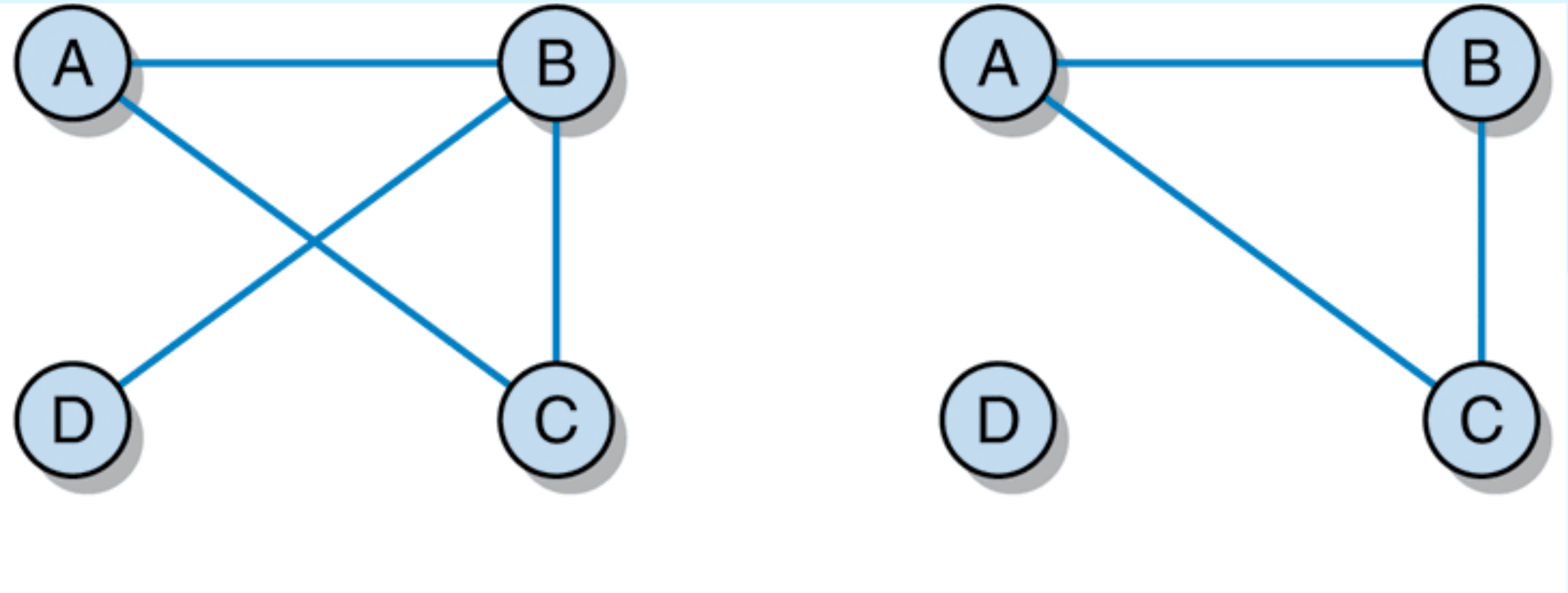
```
while (!traversalStack.isEmpty())
      {
         x = traversalStack.peek();
         found = false;

         /** Find a vertex adjacent to x that has not been visited
             and push it on the stack */
         for (int i = 0; (i < numVertices) && !found; i++)
         {
            if (adjMatrix[x.intValue()][i] && !visited[i])
            {
               traversalStack.push(new Integer(i));
               resultList.addToRear(vertices[i]);
               visited[i] = true;
               found = true;
            }
         }
         if (!found && !traversalStack.isEmpty())
            traversalStack.pop();
      }
      return resultList.iterator();
   }
```

+ Claro que ambos os algoritmos podiam ter sido implementados de forma recursiva...

```
DepthFirstSearch(node x){
   visit(x)
   result-list.addToRear(x)
   for each node y adjacent to x
      if y not visited
         DepthFirstSearch(y)
}
```

- Outro algoritmo comum para os grafos é o teste da conexidade
- O grafo é conexo se e somente se para cada vértice **v** num grafo que contém **n** vértices, o tamanho do resultado de uma travessia em largura a partir de **v** é **n**

Conexidade num grafo não direccionado



Travessia em largura para um grafo conexo não direccionado

Vértice Inicial	Travessia em Largura
A	A, B, C, D
B	B, A, D, C
C	C, B, A, D
D	D, B, A, C

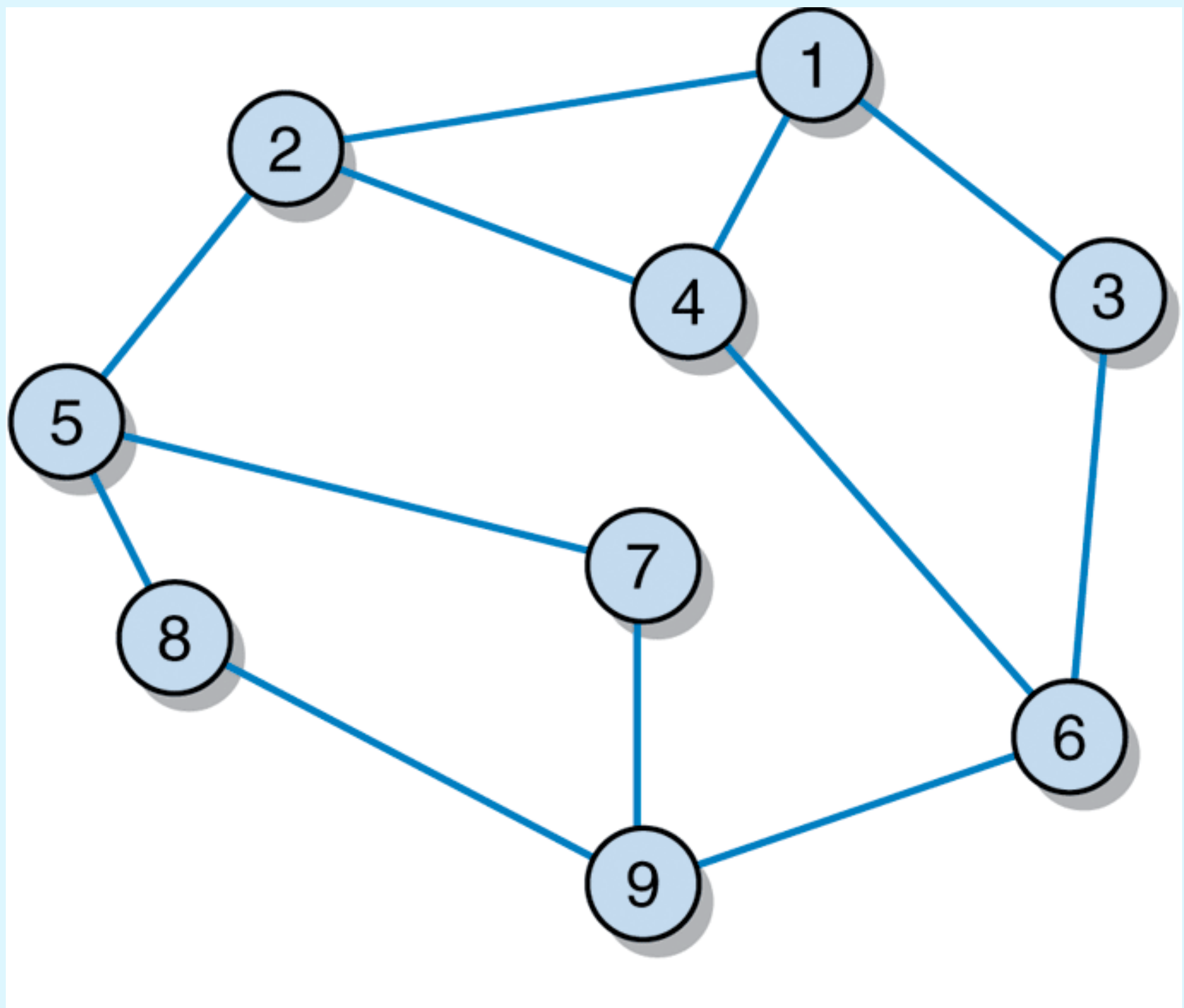
Travessia em largura para um grafo não conexo não direccionado

Vértice Inicial	Travessia em Largura
A	A, B, C
B	B, A, C
C	C, B, A
D	D

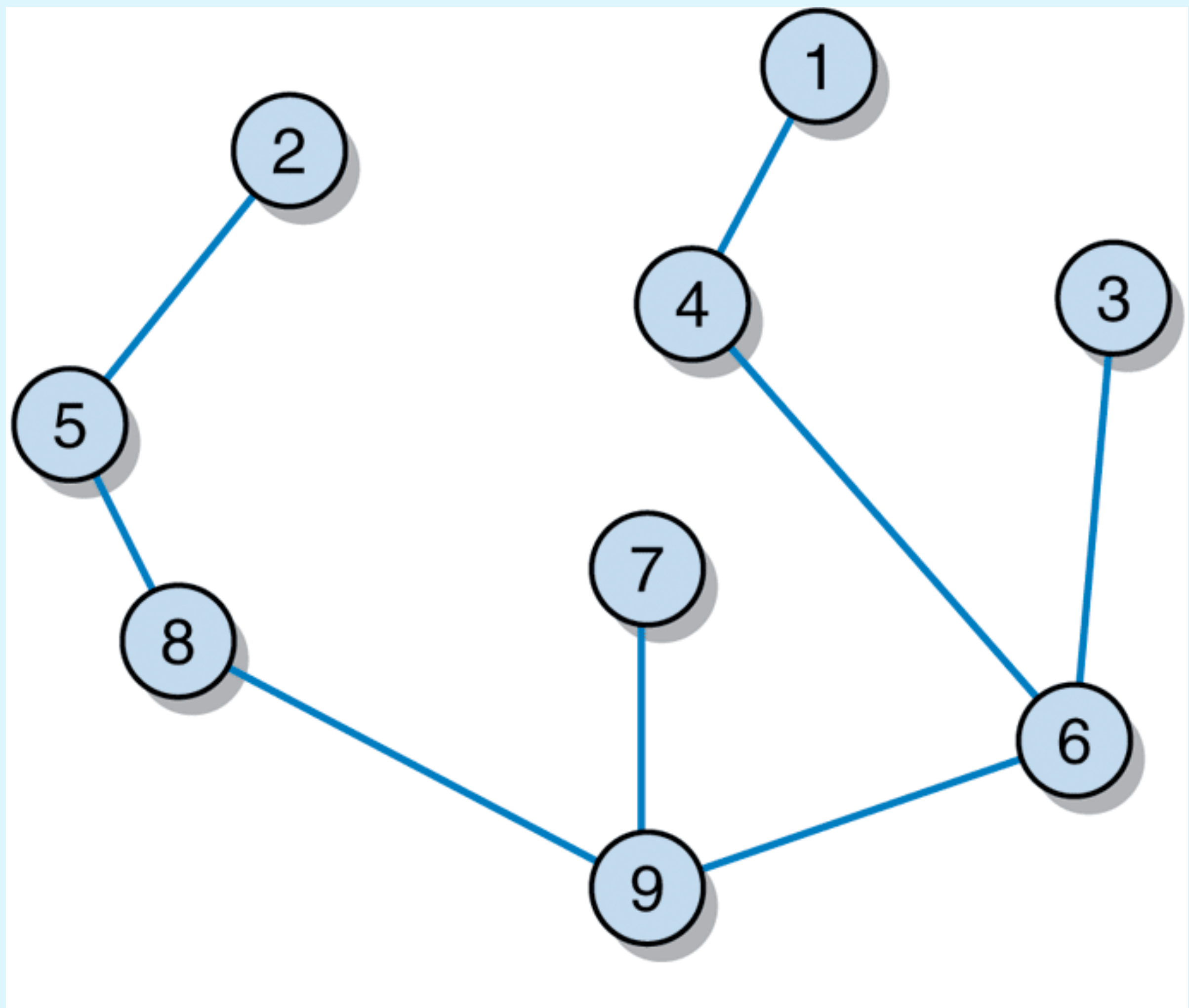
Árvore Geradora (Spanning Tree)

+ Uma árvore geradora é uma árvore que inclui todos os vértices de um grafo
+ O exemplo apresentado de seguida mostra um grafo e, de seguida, uma árvore geradora desse mesmo grafo

Exemplo de grafo



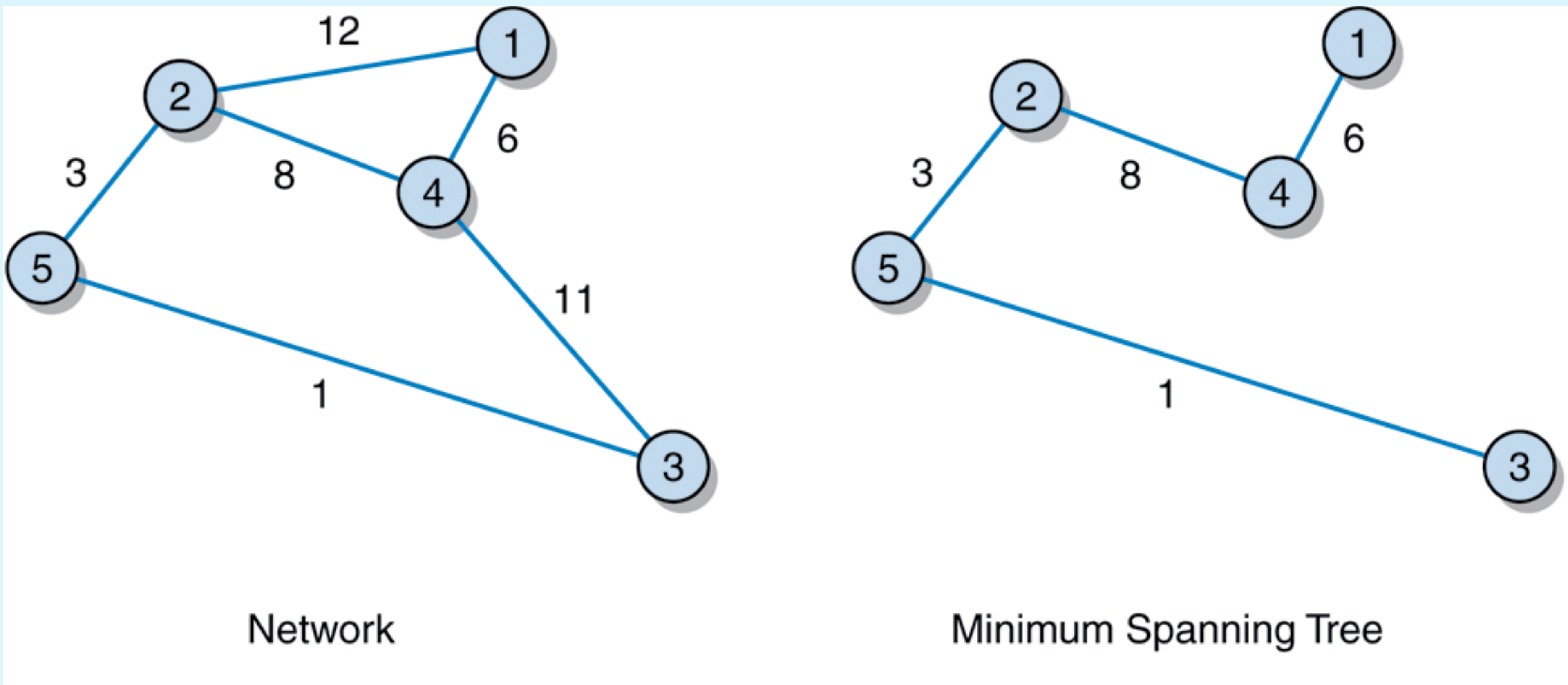
Árvore Geradora



Árvore Geradora de Custo Mínimo (Minimum Spanning Tree)

- Uma árvore geradora de custo mínimo é uma árvore geradora, onde a soma dos pesos das arestas é menor ou igual à soma dos pesos de qualquer outra árvore geradora para o mesmo grafo
- O algoritmo para a criação de uma árvore geradora de custo mínimo faz uso de uma ***minheap*** para ordenar as arestas

Árvore Geradora de Custo Mínimo



```
/**
 * Returns a minimum spanning tree of the network.
 *
 * @return  a minimum spanning tree of the network
 */
public Network mstNetwork()
{
   int x, y;
   int index;
   double weight;
   int[] edge = new int[2];
   Heap<Double> minHeap = new Heap<Double>();
   Network<T> resultGraph = new Network<T>();

   if (isEmpty() || !isConnected())
      return resultGraph;

   resultGraph.adjMatrix = new double[numVertices][numVertices];
   for (int i = 0; i < numVertices; i++)
      for (int j = 0; j < numVertices; j++)
         resultGraph.adjMatrix[i][j] = Double.POSITIVE_INFINITY;
   resultGraph.vertices = (T[])(new Object[numVertices]);
```

```
boolean[] visited = new boolean[numVertices];
for (int i = 0; i < numVertices; i++)
   visited[i] = false;

edge[0] = 0;
resultGraph.vertices[0] = this.vertices[0];
resultGraph.numVertices++;
visited[0] = true;

/** Add all edges, which are adjacent to the starting vertex,
    to the heap */
for (int i = 0; i < numVertices; i++)
      minHeap.addElement(new Double(adjMatrix[0][i]));

while ((resultGraph.size() < this.size()) && !minHeap.isEmpty())
{
   /** Get the edge with the smallest weight that has exactly
       one vertex already in the resultGraph */
   do
   {
      weight = (minHeap.removeMin()).doubleValue();
      edge = getEdgeWithWeightOf(weight, visited);
   } while (!indexIsValid(edge[0]) || !indexIsValid(edge[1]));
```

```
x = edge[0];
y = edge[1];
if (!visited[x])
   index = x;
else
   index = y;

/** Add the new edge and vertex to the resultGraph */
resultGraph.vertices[index] = this.vertices[index];
visited[index] = true;
resultGraph.numVertices++;

resultGraph.adjMatrix[x][y] = this.adjMatrix[x][y];
resultGraph.adjMatrix[y][x] = this.adjMatrix[y][x];
```

```
        /** Add all edges, that are adjacent to the newly added vertex,
            to the heap */
        for (int i = 0; i < numVertices; i++)
        {
           if (!visited[i] && (this.adjMatrix[i][index] <
                                        Double.POSITIVE_INFINITY))
           {
              edge[0] = index;
              edge[1] = I;
              minHeap.addElement(new Double(adjMatrix[index][i]));
           }
        }
    }
    return resultGraph;
}
```

Determinar o Caminho Mais Curto (Shortest Path)

- Existem duas possibilidades para a determinar o caminho mais curto de um grafo
  - Determinar o caminho mais curto em termos de número de arestas
  - Determinar o caminho menos caro numa rede

- A solução para a primeira possibilidade é uma simples variação do nosso algoritmo anterior da travessia em largura
- Temos simplesmente de armazenar dois elementos de informação adicionais para cada vértice
  - O comprimento do percurso desde o ponto de partida até este vértice
  - O vértice que é o antecessor do vértice no caminho
- Por fim modificar o ciclo para terminar quando chegar ao vértice destino

- Para a segunda possibilidade a solução é procurar o caminho mais barato em na rede
- *Dijkstra* desenvolveu um algoritmo para esta alternativa que é muito semelhante ao algoritmo anterior
- Escolhido um vértice como raiz da busca, este algoritmo calcula o custo mínimo deste vértice para todos os demais vértices do grafo
- É bastante simples e com um bom nível de performance
- Não garante, contudo, a exactidão da solução caso haja a presença de adjacências com valores negativos

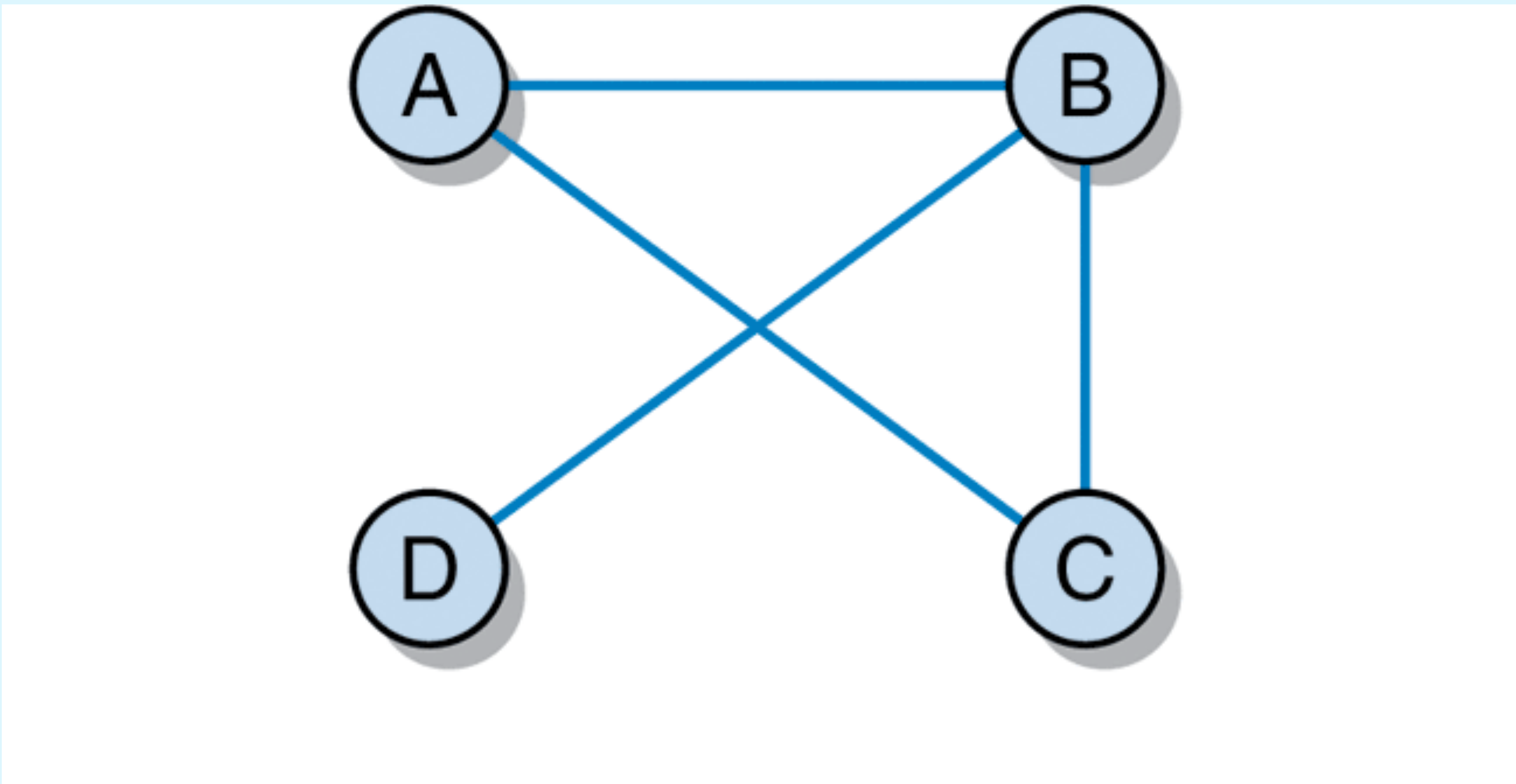
Estratégias para implementar grafos

- Existem duas abordagens principais para a implementação de grafos
  - Lista de adjacências
  - Matriz de adjacências

- A abordagem da lista de adjacências é muito semelhante à implementação de árvores com listas ligadas
- No entanto, em vez de criarmos um nó de grafo com um número fixo de referências (como fizemos com a `BinaryTreeNode`) criamos um nó de grafo que simplesmente mantém uma lista ligada de referências para outros nós
- Esta lista é chamada de lista de adjacências
- Representa um grafo usando **n** listas ligadas onde **n** é o número de vértices

- A segunda estratégia para implementar grafos é com recurso a uma matriz de adjacências
- Uma matriz de adjacências é simplesmente uma matriz bidimensional, onde ambas as dimensões são "indexadas" pelos vértices do grafo
- Cada posição da matriz contém um valor booleano que será *true* se os dois vértices associados estiverem ligados por uma aresta, e *false* caso contrário
- Os *slides* seguintes mostram dois exemplos de matrizes de adjacências, uma para um grafo não direccionado, o outro para um grafo direccionado
- Uma matriz de adjacências de uma rede pode armazenar os pesos em cada célula, em vez de um valor boolean

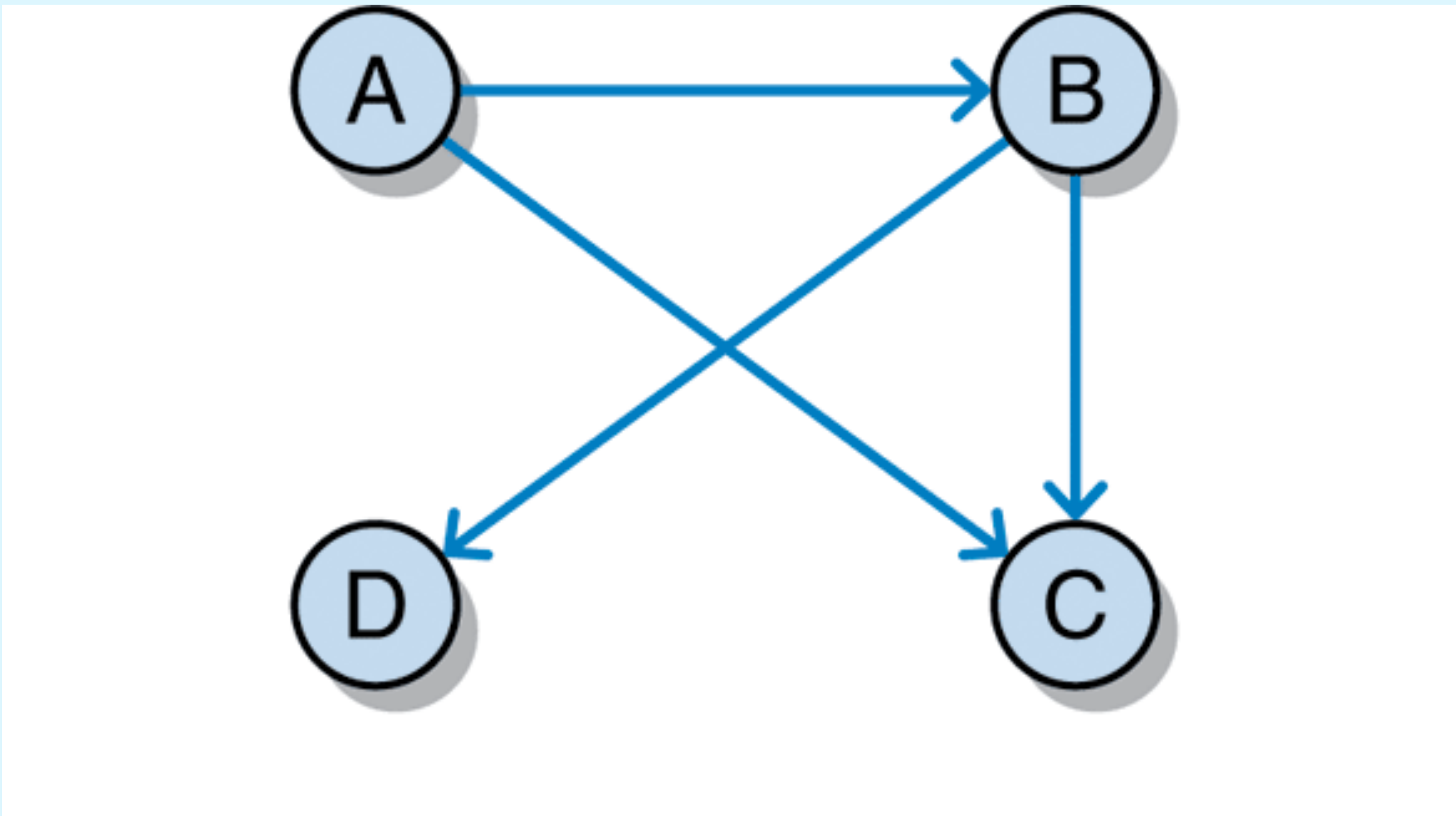
Grafo não direccionado



Matriz de adjacência para um grafo não direccionado

	A	B	C	D
A	F	T	T	F
B	T	F	T	T
C	T	T	F	F
D	F	T	F	F

Grafo direccionado



Matriz de adjacência para um grafo direccionado

	A	B	C	D
A	F	T	T	F
B	F	F	T	T
C	F	F	F	F
D	F	F	F	F

Interface GraphADT


```
/**
 * GraphADT defines the interface to a graph data structure.
 *
 */

public interface GraphADT<T>
{
   /**
    * Adds a vertex to this graph, associating object with vertex.
    *
    * @param vertex  the vertex to be added to this graph
    */
   public void addVertex (T vertex);
```

```java
/**
 * Removes a single vertex with the given value from this graph.
 *
 * @param vertex  the vertex to be removed from this graph
 */
public void removeVertex (T vertex);

/**
 * Inserts an edge between two vertices of this graph.
 *
 * @param vertex1  the first vertex
 * @param vertex2  the second vertex
 */
public void addEdge (T vertex1, T vertex2);

/**
 * Removes an edge between two vertices of this graph.
 *
 * @param vertex1  the first vertex
 * @param vertex2  the second vertex
 */
public void removeEdge (T vertex1, T vertex2);
```

```
/**
 * Returns a breadth first iterator starting with the given vertex.
 *
 * @param startVertex  the starting vertex
 * @return             a breadth first iterator beginning at
 *                     the given vertex
 */
public Iterator iteratorBFS(T startVertex);

/**
 * Returns a depth first iterator starting with the given vertex.
 *
 * @param startVertex  the starting vertex
 * @return             a depth first iterator starting at the
 *                     given vertex
 */
public Iterator iteratorDFS(T startVertex);
```

```
/**
 * Returns an iterator that contains the shortest path between
 * the two vertices.
 *
 * @param startVertex   the starting vertex
 * @param targetVertex  the ending vertex
 * @return              an iterator that contains the shortest
 *                      path between the two vertices
 */
public Iterator iteratorShortestPath(T startVertex, T targetVertex);

/**
 * Returns true if this graph is empty, false otherwise.
 *
 * @return  true if this graph is empty
 */
public boolean isEmpty();

/**
 * Returns true if this graph is connected, false otherwise.
 *
 * @return  true if this graph is connected
 */
public boolean isConnected();
```

```
    /**
     * Returns the number of vertices in this graph.
     *
     * @return  the integer number of vertices in this graph
     */
    public int size();

    /**
     * Returns a string representation of the adjacency matrix.
     *
     * @return  a string representation of the adjacency matrix
     */
    public String toString();
}
```

Interface NetworkADT


```
/**
 * NetworkADT defines the interface to a network.
 *
 */

public interface NetworkADT<T> extends GraphADT<T>
{
   /**
    * Inserts an edge between two vertices of this graph.
    *
    * @param vertex1  the first vertex
    * @param vertex2  the second vertex
    * @param weight   the weight
    */
   public void addEdge (T vertex1, T vertex2, double weight);
```

```
/**
   * Returns the weight of the shortest path in this network.
   *
   * @param vertex1  the first vertex
   * @param vertex2  the second vertex
   * @return         the weight of the shortest path in this network
   */
  public double shortestPathWeight(T vertex1, T vertex2);
}
```

Implementação de um Grafo


```
/**
 * Graph represents an adjacency matrix implementation of a graph.
 *
 */

public class Graph<T> implements GraphADT<T> {
   protected final int DEFAULT_CAPACITY = 10;
   protected int numVertices;   // number of vertices in the graph
   protected boolean[][] adjMatrix;   // adjacency matrix
   protected T[] vertices;   // values of vertices

   /**
    * Creates an empty graph.
    */
   public Graph() {
      numVertices = 0;
      this.adjMatrix = new boolean[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
      this.vertices = (T[])(new Object[DEFAULT_CAPACITY]);
   }
```

```
/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param vertex1  the first vertex
 * @param vertex2  the second vertex
 */
public void addEdge (T vertex1, T vertex2) {
   addEdge (getIndex(vertex1), getIndex(vertex2));
}

/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param index1  the first index
 * @param index2  the second index
 */
public void addEdge (int index1, int index2) {
   if (indexIsValid(index1) && indexIsValid(index2))
   {
      adjMatrix[index1][index2] = true;
      adjMatrix[index2][index1] = true;
   }
}
```

```
/**
 * Adds a vertex to the graph, expanding the capacity of the graph
 * if necessary.  It also associates an object with the vertex.
 *
 * @param vertex  the vertex to add to the graph
 */
public void addVertex (T vertex)
{
   if (numVertices == vertices.length)
      expandCapacity();

   vertices[numVertices] = vertex;
   for (int i = 0; i <= numVertices; i++)
   {
      adjMatrix[numVertices][i] = false;
      adjMatrix[i][numVertices] = false;
   }
   numVertices++;
}
```