

# Grafos I

27/11/2023

# Ficheiro ZIP

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- **Ficheiros de texto** que armazenam diferentes tipos de **grafos**
- O tipo abstrato **Grafo** usando o **TAD SortedList**
- **Versão “simples”**, que permite trabalho autónomo de desenvolvimento e teste

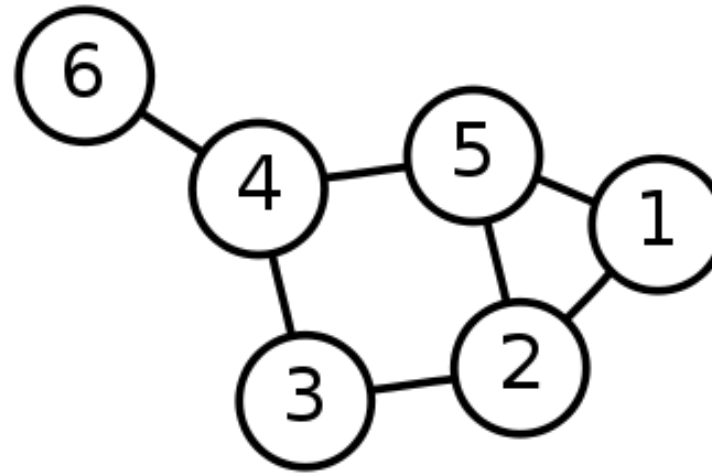
# Sumário

- Grafos: Terminologia; Exemplos de aplicação; Propriedades
- O Tipo de Dado **Grafo**
- Possíveis estruturas de dados
- Operações básicas
- Desempenho computacional
- Sugestão de leitura

# Grafos

## – Motivação + Exemplos

# Grafo



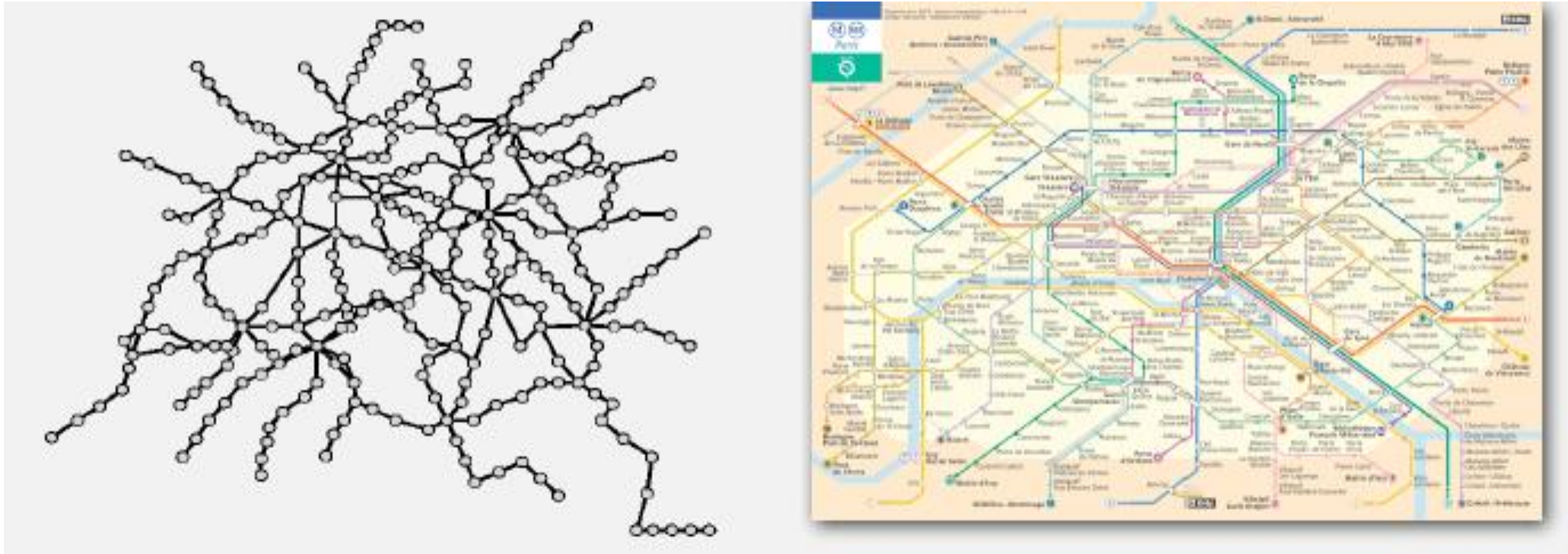
[Wikipedia]

- $G( V, E )$
- Quando muito **uma aresta** ligando qualquer **par de vértices distintos**
- $e_i = ( v_j, v_k )$ 
  - $v_j$  e  $v_k$  são vértices **adjacentes**
  - $e_i$  é **incidente** em  $v_j$  e em  $v_k$

# Porquê estudar ?

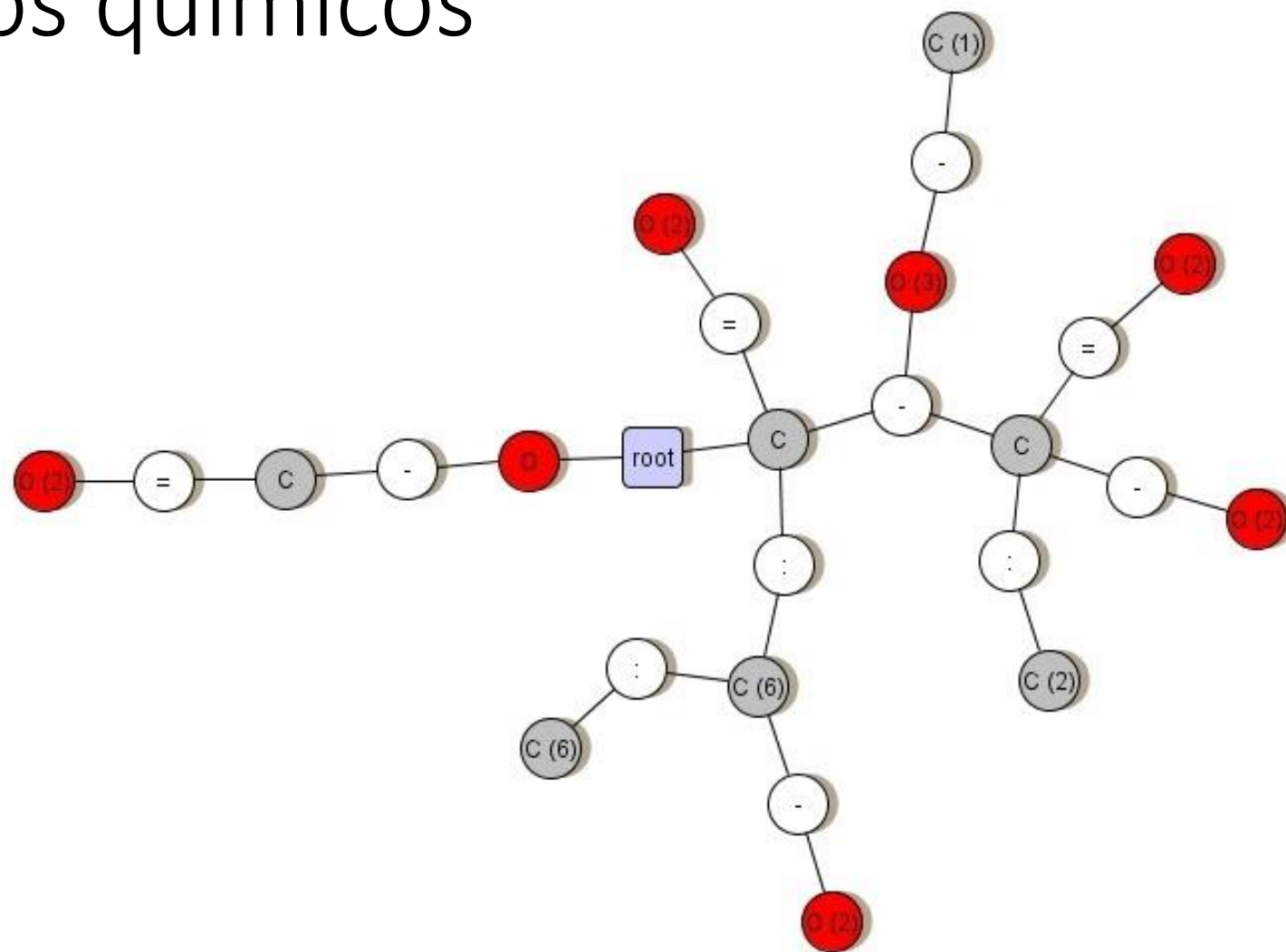
- Abstração útil
- Subárea das Ciências da Computação e da Matemática Discreta
  - Problemas / Algoritmos / Aplicações
- Centenas de algoritmos
- Milhares de aplicações práticas

# Redes de transportes



[Sedgewick & Wayne]

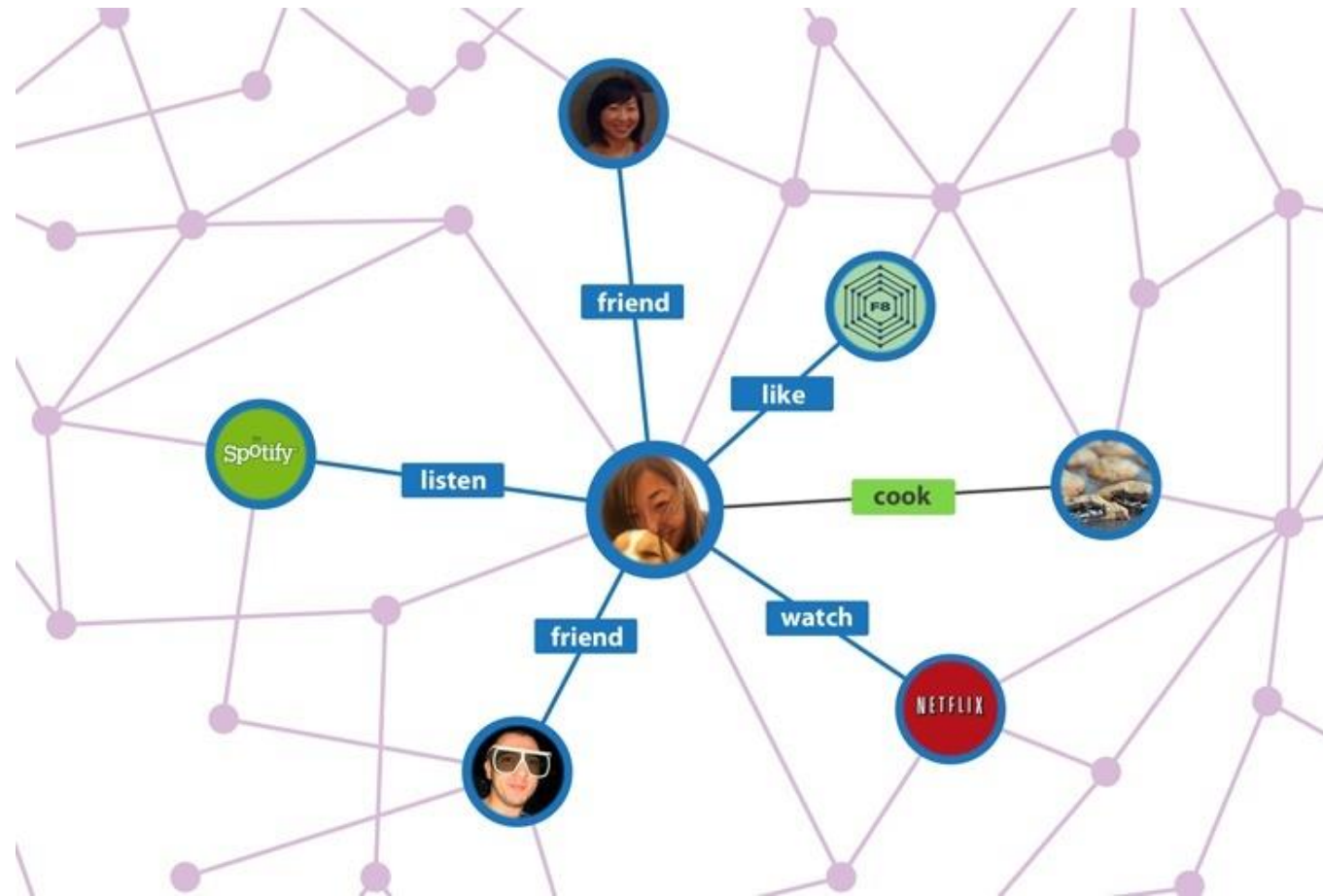
# Modelos químicos



[Quora]



# Redes sociais



[Quora]

# Aplicações

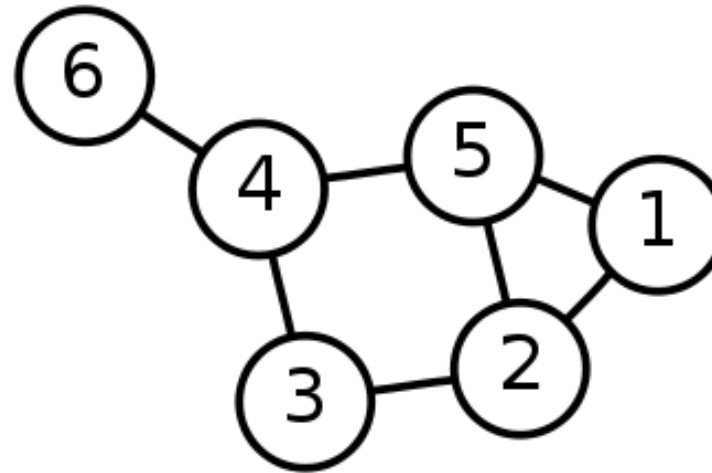
Graph	Vertex	Edge
communication	telephone, computer	cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transaction
transportation	street intersection, airport	highway, airway route
Internet	class C network	connection
game	board position	legal move
relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	bond

[Sedgewick & Wayne]

# Grafos

## – Terminologia + Propriedades

# Grafo



[Wikipedia]

- $G( V, E )$
- Quando muito **uma aresta** ligando qualquer **par de vértices distintos**
- $e_i = ( v_j, v_k )$ 
  - $v_j$  e  $v_k$  são vértices **adjacentes**
  - $e_i$  é **incidente** em  $v_j$  e em  $v_k$

# Grafos

- Número máximo de arestas =  $V \times (V - 1) / 2$ 
  - Grafo completo :  $K_V$
- Grau de um vértice
  - Número de arestas incidentes nesse vértice
  - Grau máximo ?
- Grafo regular
  - Todos os vértices têm o mesmo grau  $k$  : grafo  *$k$ -regular*

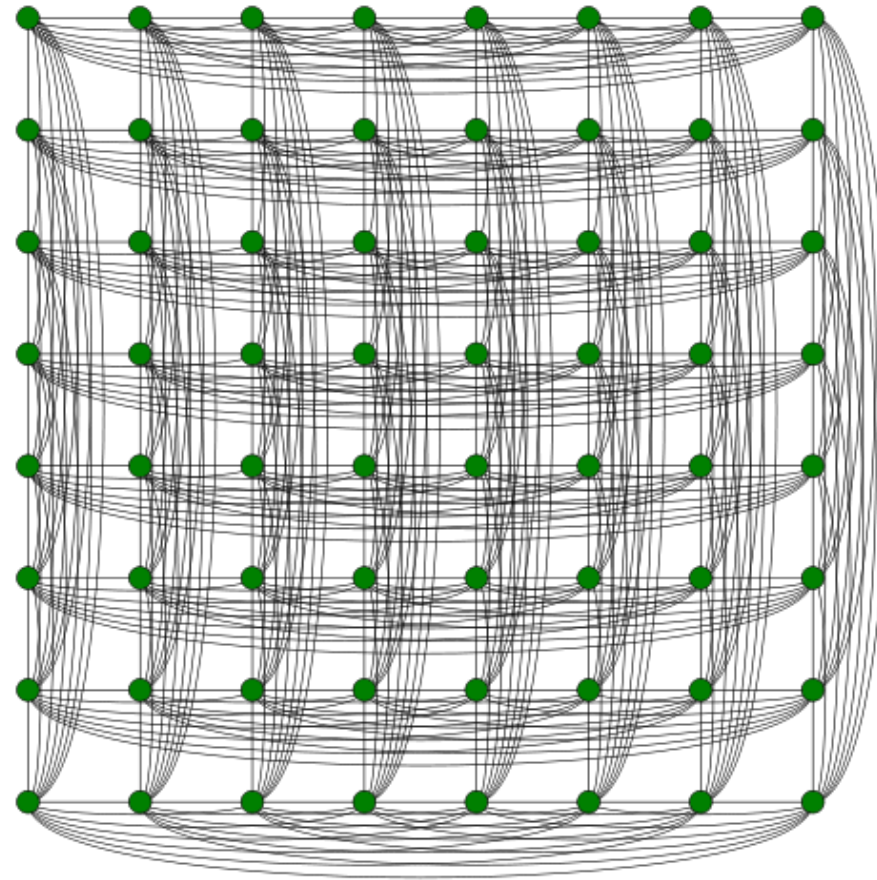
# Grafo dos movimentos da torre no xadrez

64 vértices

448 arestas

14-regular

número cromático 8



[Wikipedia]

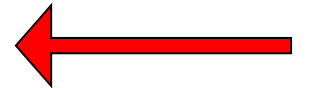
# Grafos

$$\sum grau(v) = 2 \times E$$

- Grau médio – Average vertex degree

$$AVD = (2 \times E) / V$$

- Grafo **denso** :  $E$  em  $\Theta(V^2)$  e  $AVD$  em  $\Theta(V)$ 
  - Grafos densos vs. grafos **esparsos**



- Densidade

$$D = (2 \times E) / (V \times (V - 1))$$

# Densidade – Exemplo

sparse ( $E = 200$ )

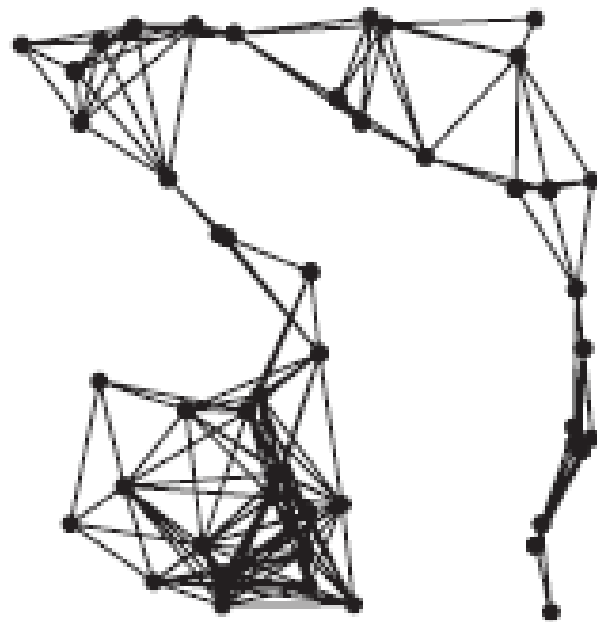


Grafico esparso



dense ( $E = 1000$ )

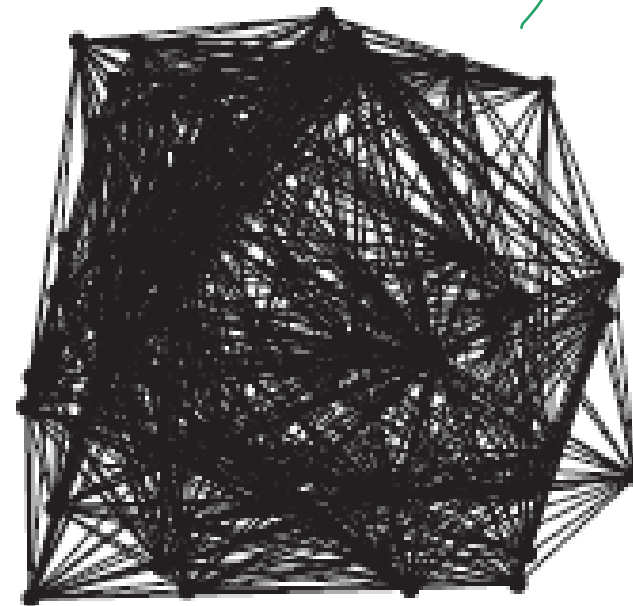


Grafico densidade



Two graphs ( $V = 50$ )

Existem algoritmos para cada um dos tipos

[Sedgewick & Wayne]



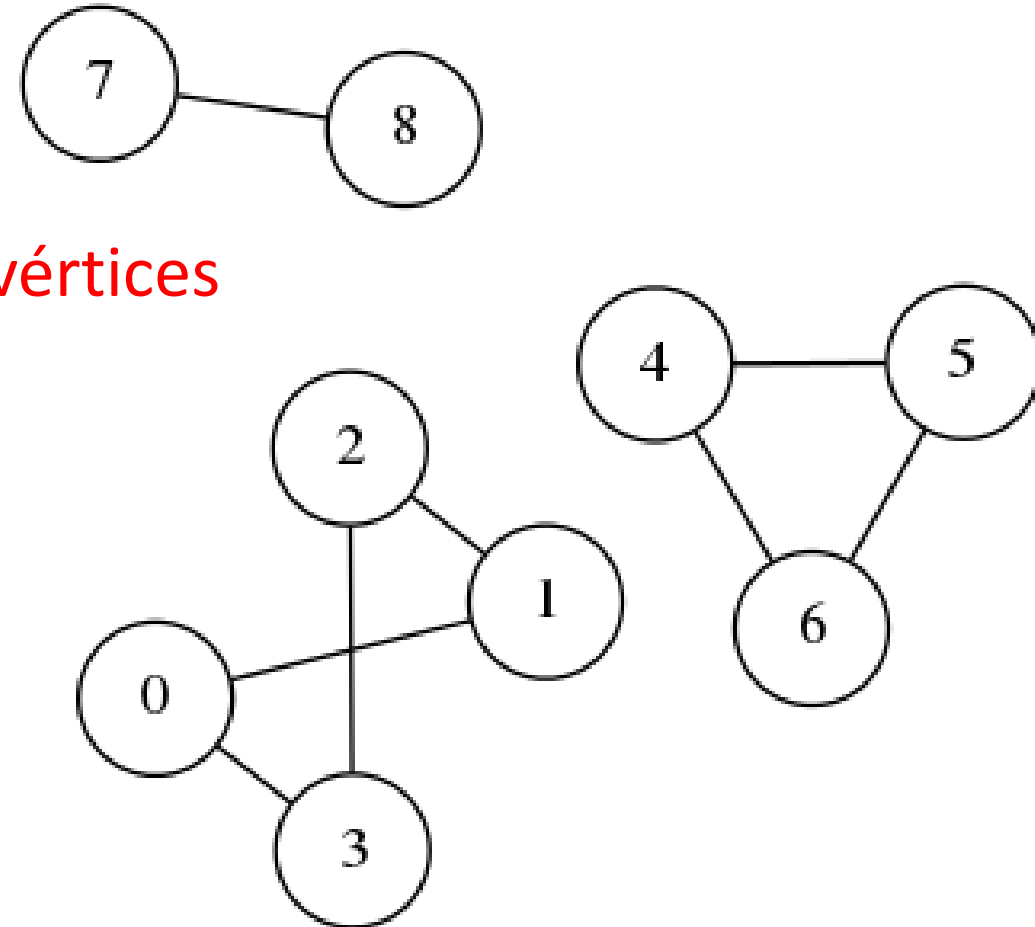
# Grafos

- Um **passeio** é uma qualquer sequência de vértices adjacentes
  - **Comprimento** do passeio: **nº de arestas** que o constituem
- Um **trajeto** é um passeio constituído por **arestas distintas**
  - Um **circuito** é um trajeto de comprimento não nulo, que começa e acaba no mesmo vértice
- Um **caminho** é um passeio constituído por **arestas e vertices distintos**
  - Um **ciclo** é um caminho de comprimento não nulo, que começa e acaba no mesmo vértice

# Grafos

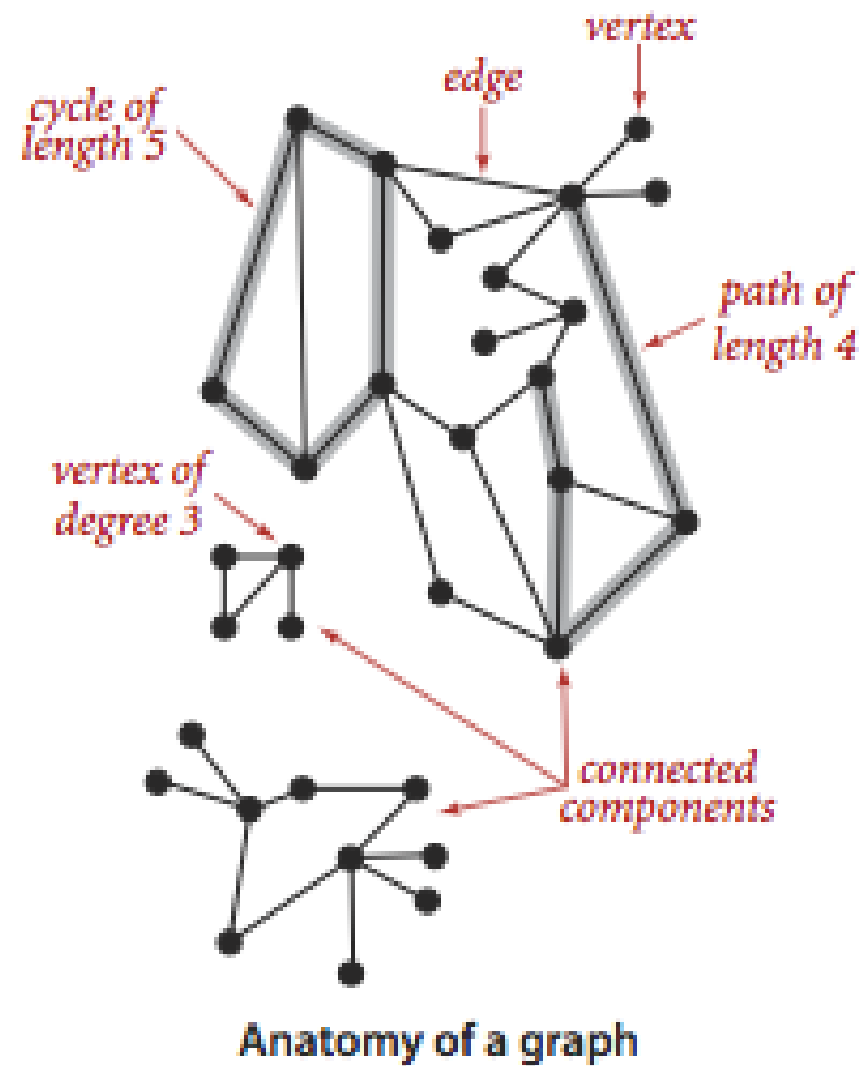
- Grafo **conexo**
  - Existe um **caminho** entre cada **par de vértices**
  - Um único componente conexo

cada 1 é conexo, mas o 3 ligados não são conexos



[martinbroadhurst.com/]

# Grafos

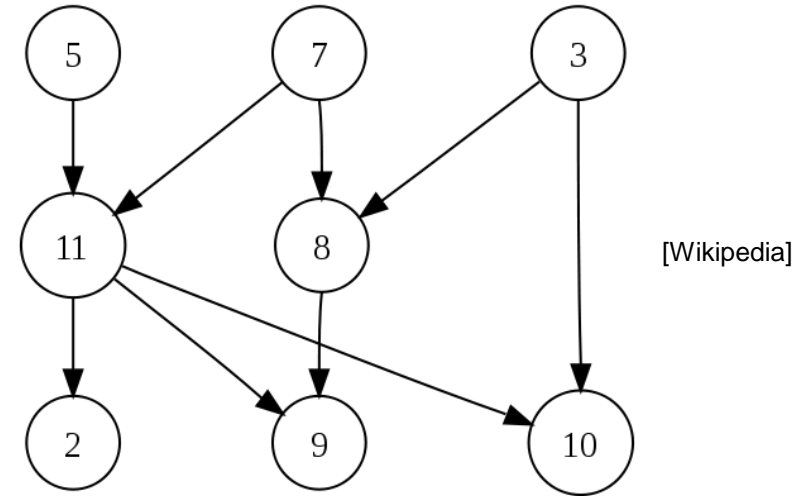


[Sedgewick & Wayne]

# Grafos Orientados

# Grafos orientados

- $G(V,E)$
- Grafo **orientado**
  - As **arestas orientadas** definem uma adjacência unidirecional
- $e_i = (v_j, v_k)$ 
  - $v_j$  é o vértice **origem** e  $v_k$  o vértice **destino**
  - $v_k$  é **adjacente** a  $v_j$
  - $e_i$  é **incidente** em  $v_k$



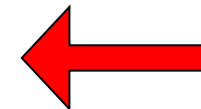
# Aplicações

Digraph	Vertex	Directed Edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object class	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

[Sedgewick/Wayne]

# Grafos orientados

- Nº máximo de arestas =  $V \times (V - 1)$ 
  - Grafo orientado completo
- In-Degree e Out-Degree associado a cada vértice
  - Vértice fonte (“source”) ?
  - Vértice sumidouro (“sink”) ?
- Densidade de um grafo orientado
  - Grafos orientados densos vs. esparsos

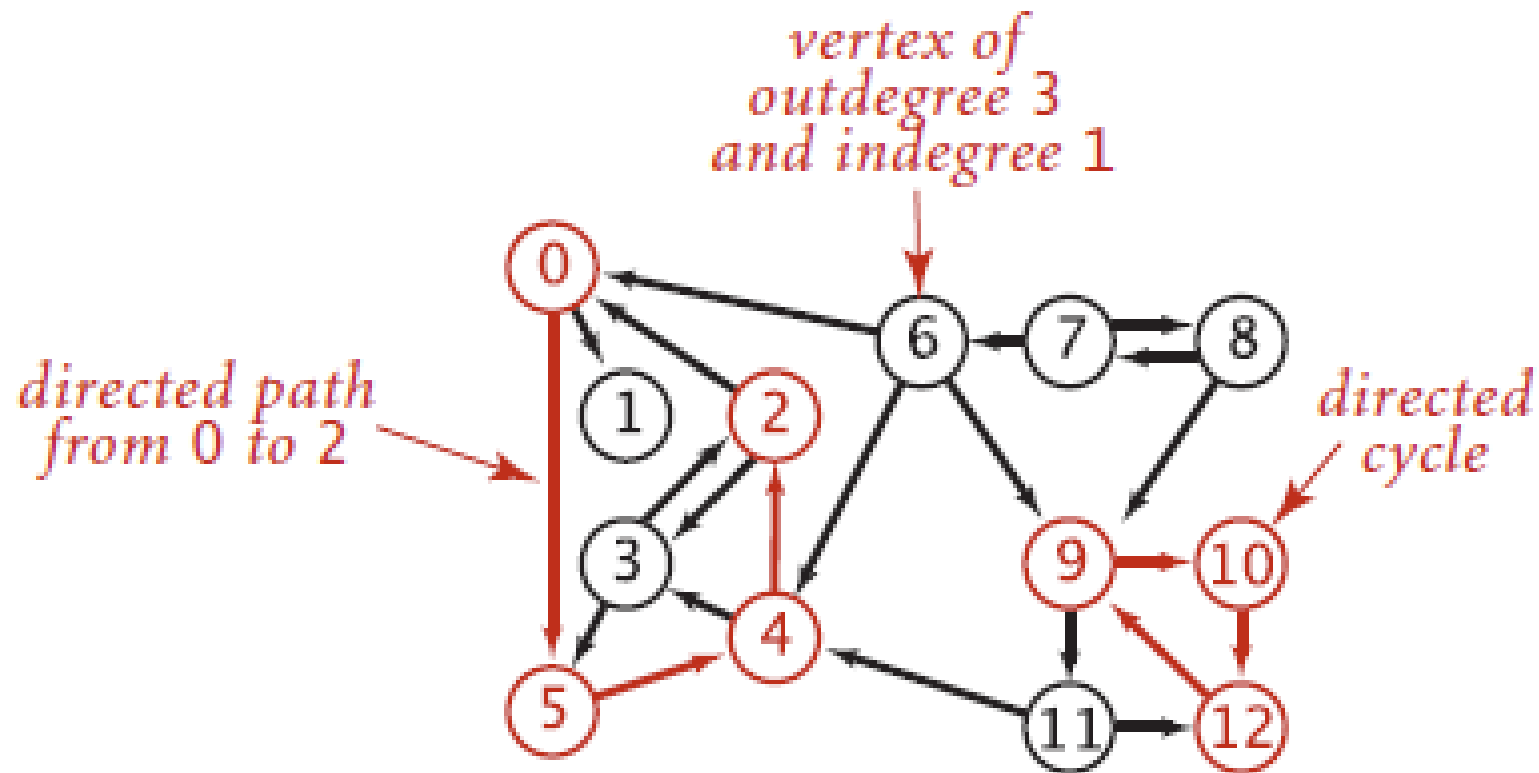


# Grafos orientados

- Um **passeio orientado** é uma sequência de vértices
  - Cada vértice (exceto o primeiro) é **adjacente** ao seu **predecessor**
- **Caminho orientado** : arestas e vértices distintos
- **Ciclo orientado** : caminho orientado com o mesmo vértice inicial e final
- Vértice  $t$  é **alcançável** a partir do vértice  $s$  ?
  - Existe um caminho orientado de  $s$  para  $t$



# Grafo orientado

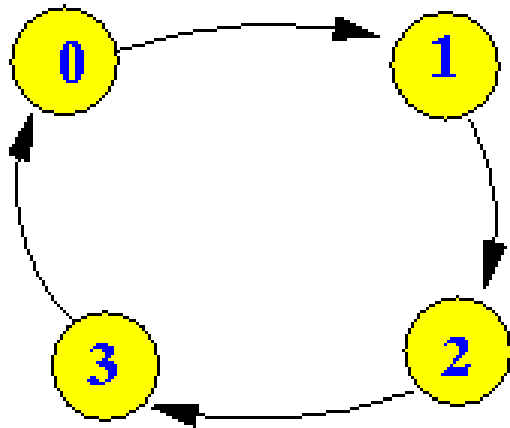


[Sedgewick/Wayne]

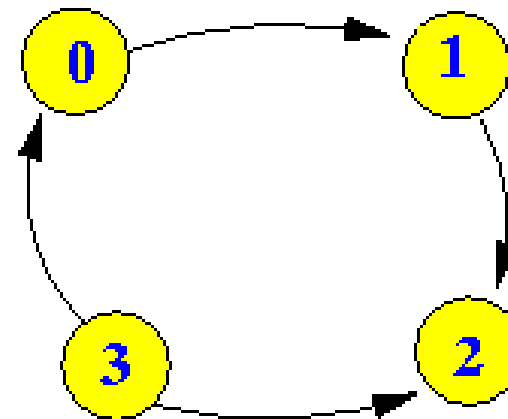
# Grafos orientados

- Grafo orientado fracamente conexo
  - Substituir as arestas orientadas por **arestas não-orientadas**
  - O grafo resultante é **conexo**
- Grafo orientado fortemente conexo O habitual...
  - Existe um caminho entre cada par de vértices
    - Vértices **mutuamente alcançáveis** !!
  - **Um único componente** fortemente conexo

# Exemplo



***Strongly connected***



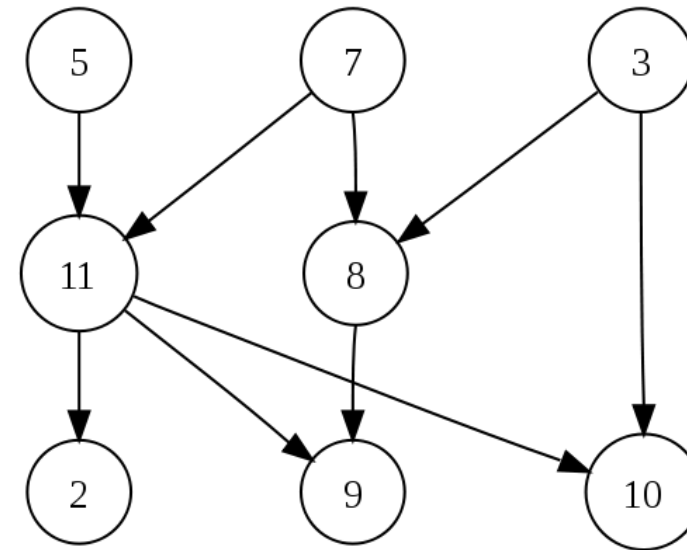
***Not strongly connected***

Mas se tirar a orientação fica conexo

[cs.emory.edu]

# Grafos orientados **acíclicos**

- Directed Acyclic Graphs (**DAGs**)
- Um grafo orientado que **não** contém **qualquer ciclo** !
  - Relações de **precedência**



[Wikipedia]

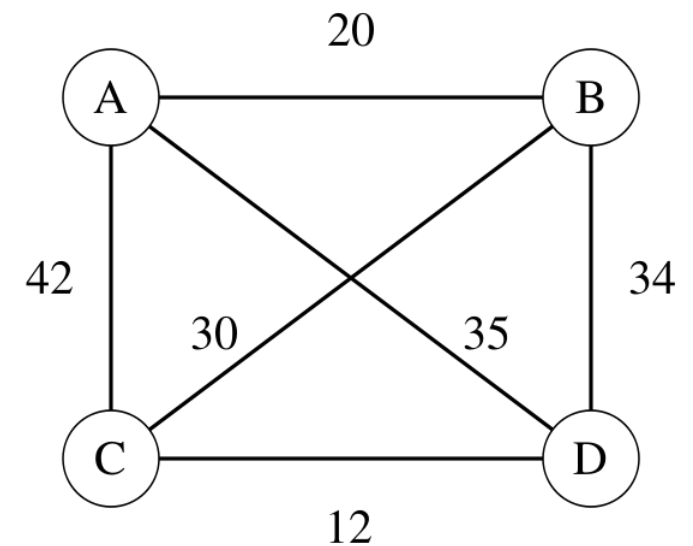
# Redes

Grafo orientado com informação associada as arestas

Associamos custos...

# Rede

- Uma rede é um grafo / grafo orientado com “pesos” associados às suas arestas
  - Weighted graph / digraph
  - Associar um ou mais valores a cada aresta
  - Custo, distância, capacidade, ...

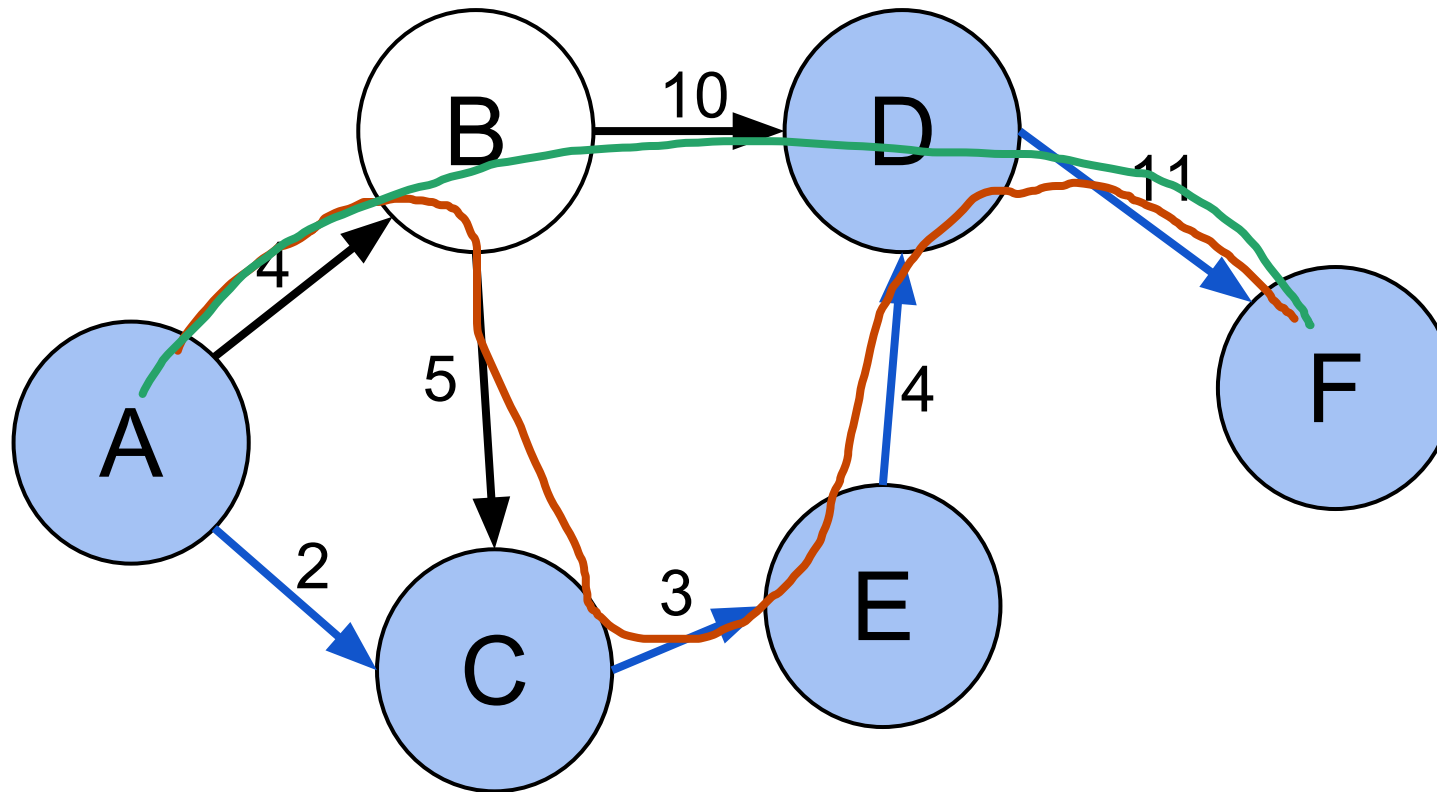


[Wikipedia]

# Caminhos

- Existe um **caminho** entre os vértices **s** e **t** ?
- Qual é o **caminho mais curto** entre **s** e **t** ?
  - **Soma** das **distâncias** associadas a cada **aresta**

# Caminho mais curto entre A e F – Custo ?




[Wikipedia]



Como representar?  
Que funcionalidades?

# TAD Grafo

Pode ser um digrafo... é genérico!



```
Graph* GraphCreate(unsigned int V);           // Apenas com V vértices
GraphDestroy(Graph** g);
Graph* GraphCopy(Graph* g);
Graph* GraphFromFile(FILE* f);
unsigned int GraphGetNumVertices(Graph* g);
unsigned int GraphGetNumEdges(Graph* g);
...
```

# TAD Grafo

...

```
int GraphGetVertexDegree(Graph* g, unsigned int v);
```

```
int GraphGetDegree(Graph* g);
```

```
double GraphGetAverageDegree(Graph* g);
```

...

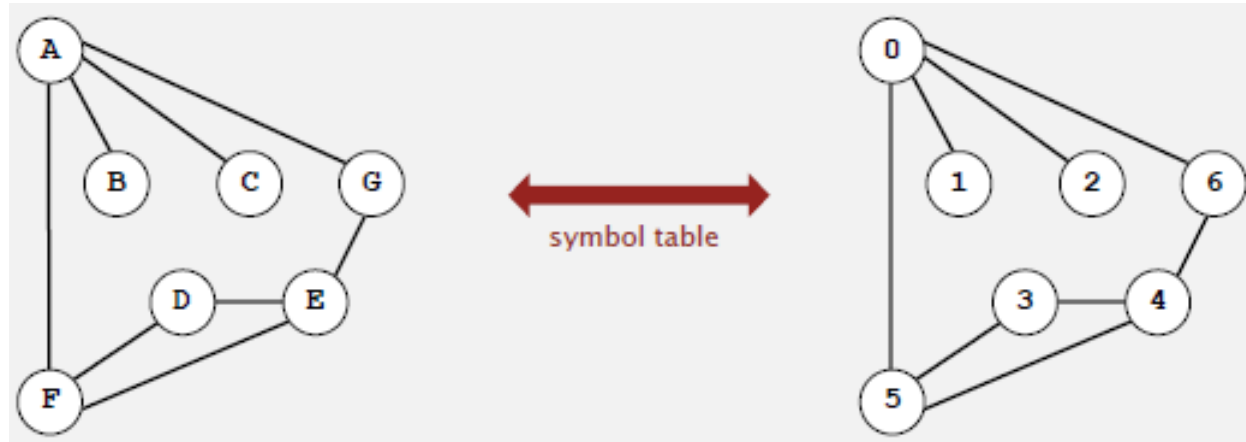
```
int GraphAddEdge(Graph* g, unsigned int v, unsigned int w);
```

```
List* GraphGetAdjacentTo(Graph* g, unsigned int v);
```

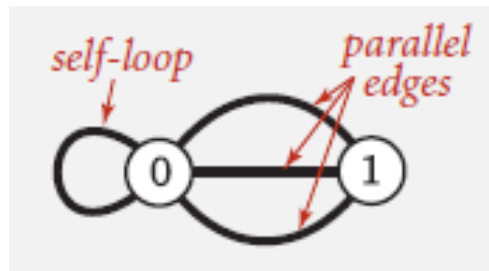
...

# Vértices

- Identificados por um valor inteiro de **0** a  **$V - 1$**
- Usar dicionários para mapear esses IDs noutros identificadores



- **Não** são permitidos **lacetes** nem **arestas paralelas**



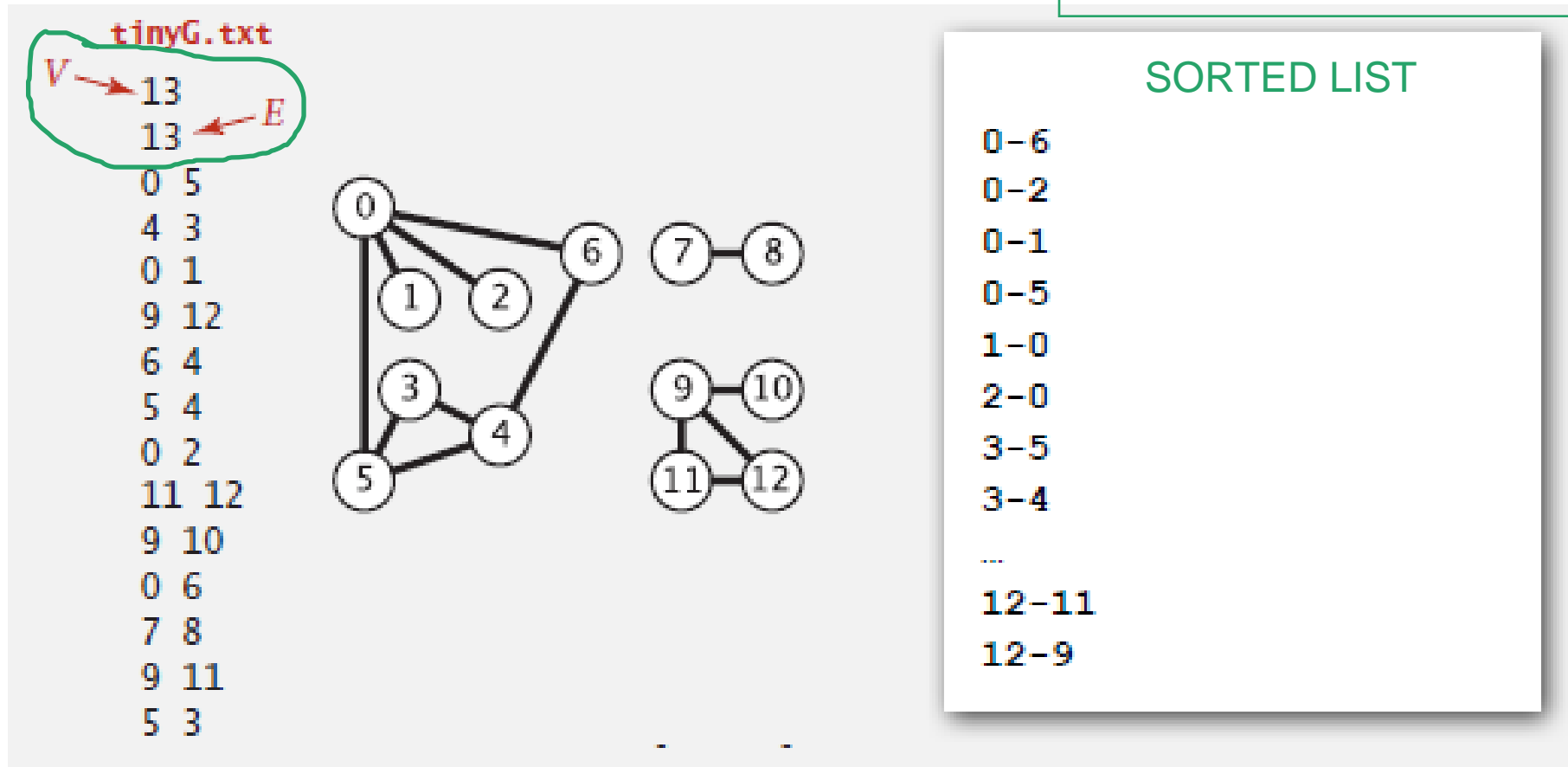
Não existem lacetes aqui!

Não existem arestas paralelas!

[Sedgewick/Wayne]

# Representação em ficheiro – Lista de arestas

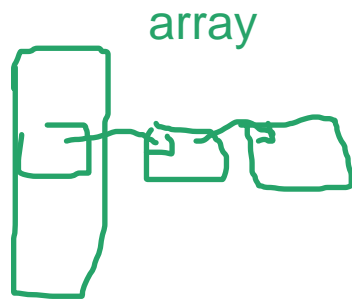
Para remover precisamos de eliminar todas as resp



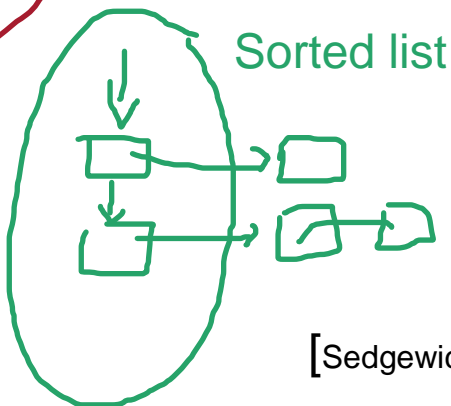
[Sedgewick/Wayne]

# Representação – Lista ordenada de arestas

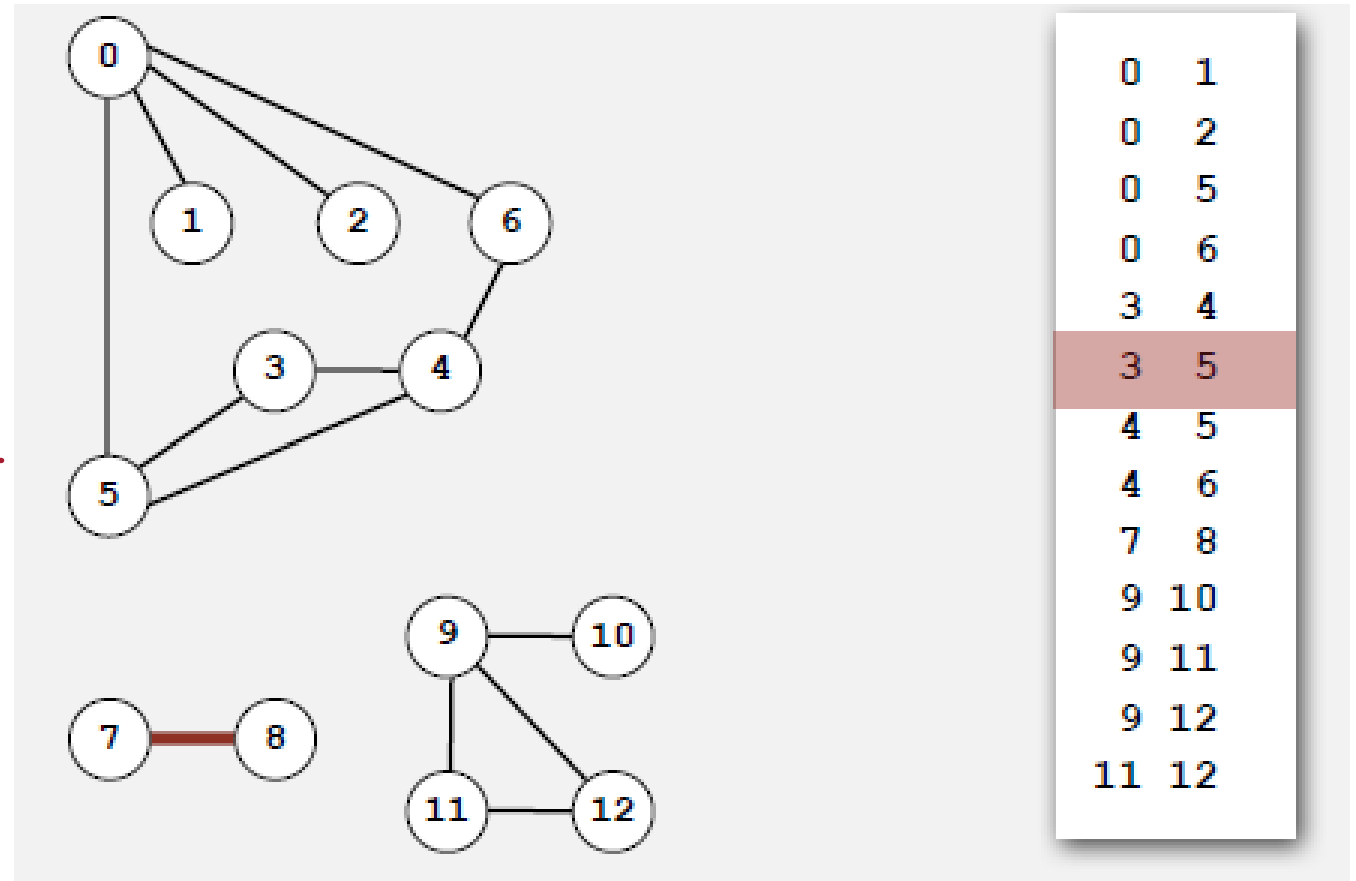
- Lista ligada de arestas



OU!



[Sedgewick/Wayne]

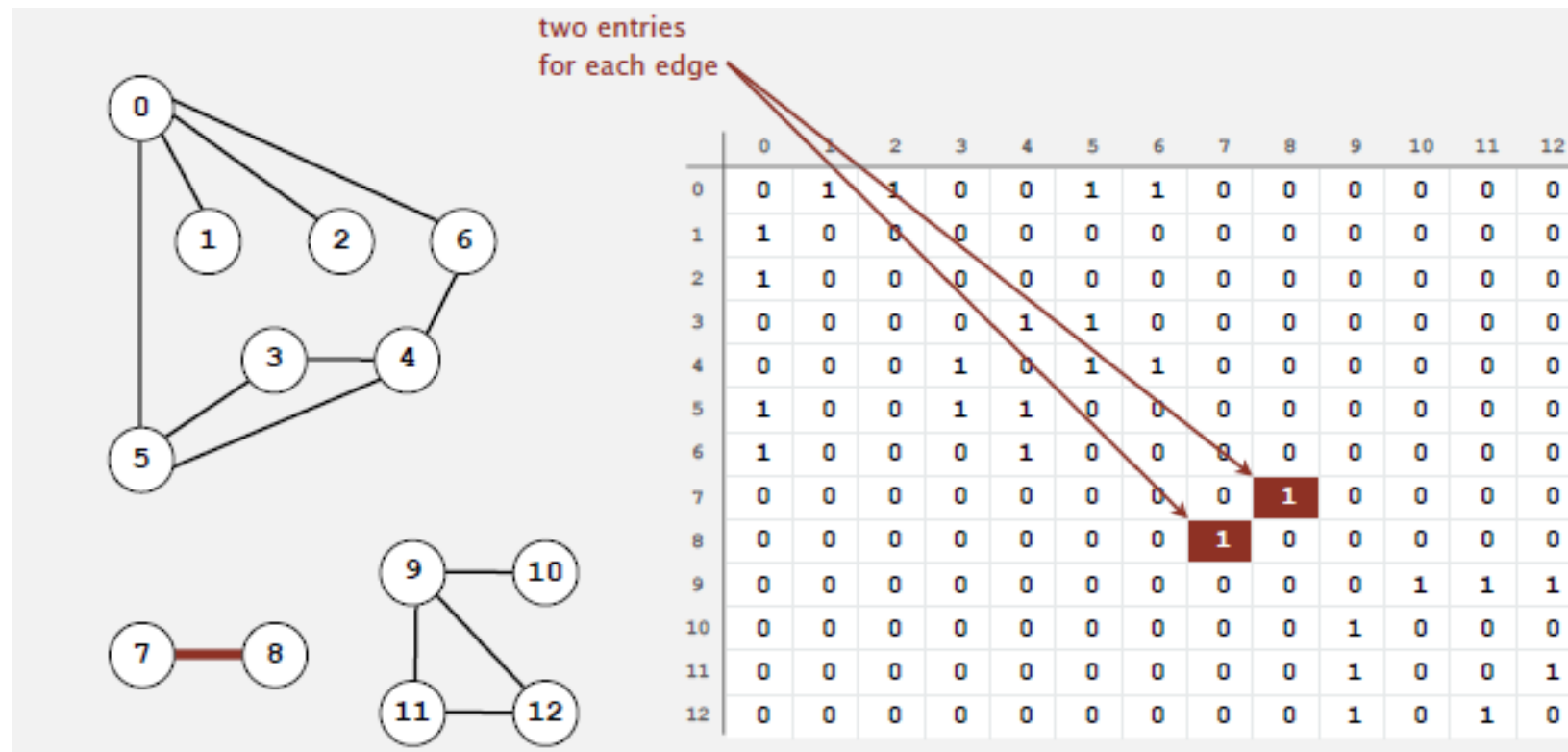


# Representação – Matriz de adjacência

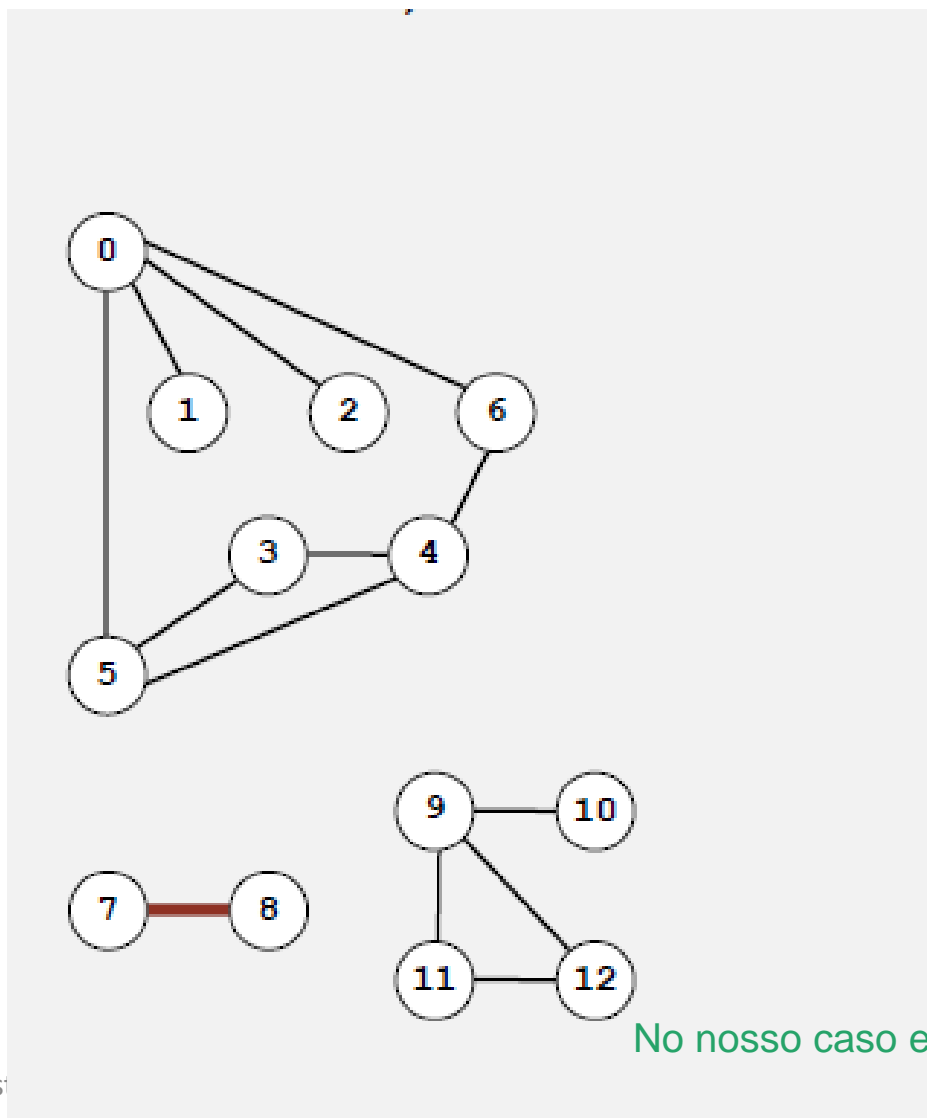
- Array de  $V^2$  booleanos
- Cada aresta é representada duas vezes
  - Porquê ?

Se o grafo for esparsos, uma rede tem muitos 0,  
e se for não orientado é simétrica

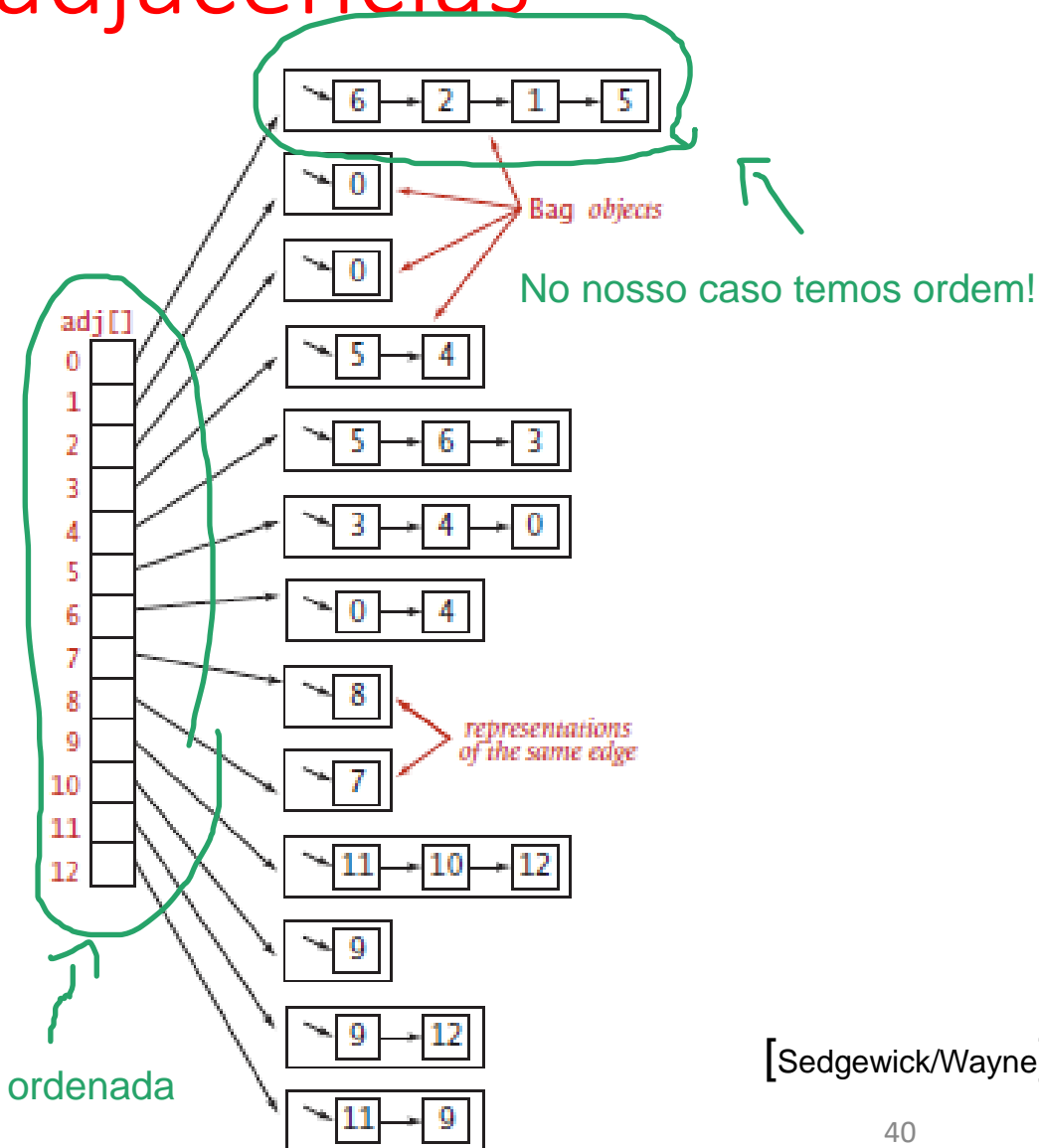
[Sedgewick/Wayne]



# Representação – Listas de adjacências



No nosso caso está ordenada



[Sedgewick/Wayne]

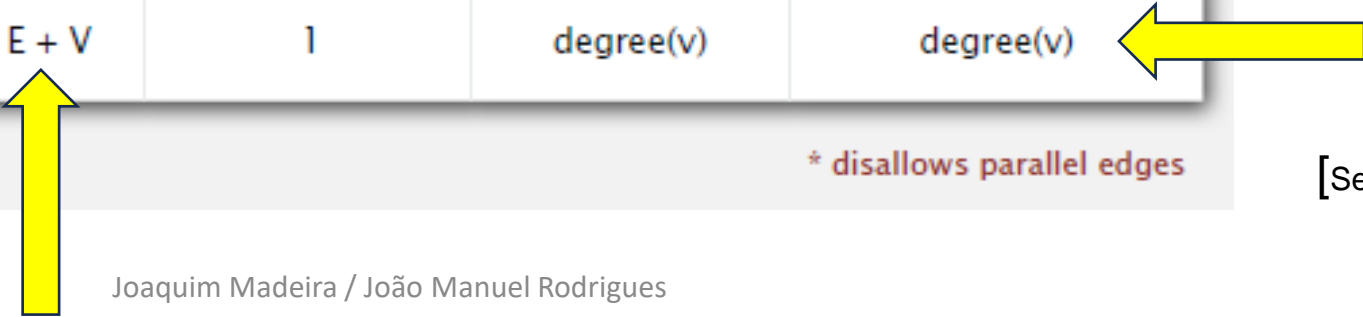


# Desempenho

- Na prática: usar a representação em **listas de adjacências**
- Os **grafos** do mundo real são habitualmente **esparsos** !!
- Algoritmos iteram sobre os vértices adjacentes a um vértice dado

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	$E$	1	$E$	$E$
adjacency matrix	$V^2$	1 *	1	$V$
adjacency lists	$E + V$	1	degree(v)	degree(v)

\* disallows parallel edges



[Sedgewick/Wayne]

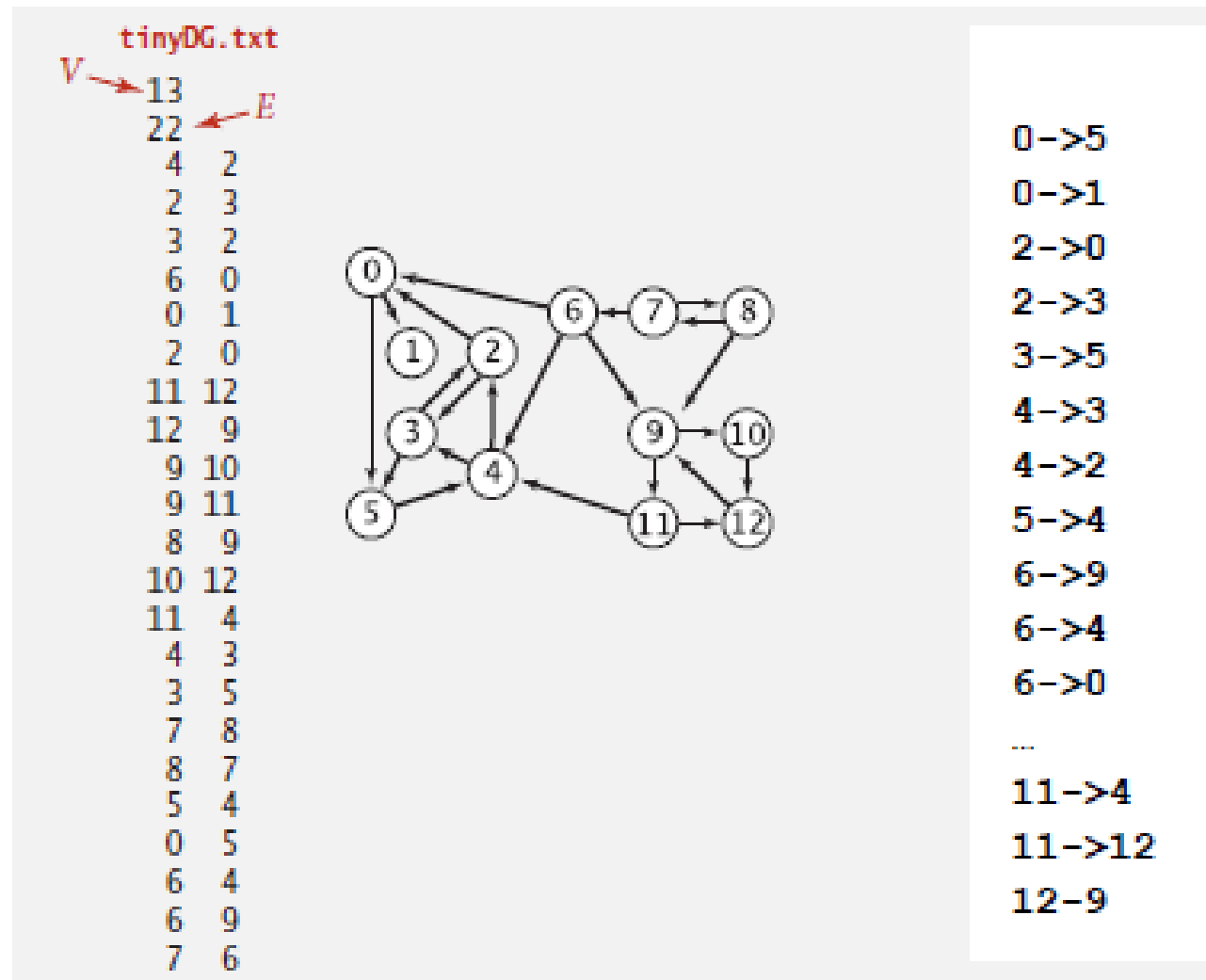
# TAD Grafo Orientado

No nosso caso não fazemos isto!



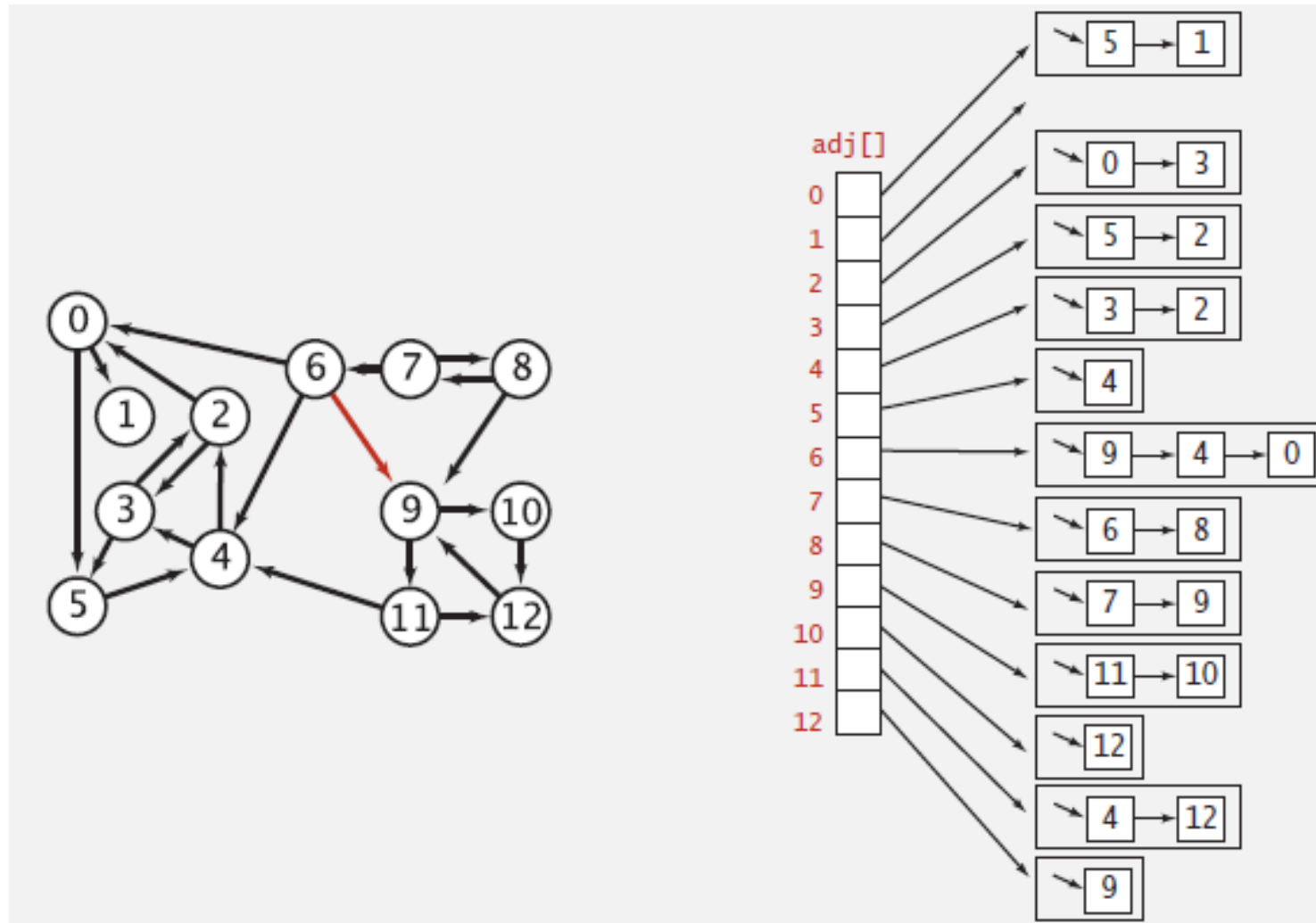
```
Digraph* DigraphCreate(unsigned int V);    // Apenas com V vértices
DigraphDestroy(Digraph** g);
Digraph* DigraphCopy(Digraph* g);
Digraph* DigraphCreateReverse(Digraph* g);
Digraph* DigraphFromFile(FILE* f);
unsigned int DigraphGetNumVertices(Digraph* g);
unsigned int DigraphGetNumEdges(Digraph* g);
...
```

# Representação em ficheiro – Lista de arestas



[Sedgewick/Wayne]

# Representação – Listas de adjacências



[Sedgewick/Wayne]

# Desempenho

- Na prática: usar a representação em **listas de adjacências**
- Os **grafos orientados** do mundo real são habitualmente **esparsos** !!
- Algoritmos iteram sobre os vértices adjacentes a um vértice dado

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices adjacent from v?
list of edges	E	1	E	E
adjacency matrix	$V^2$	1 †	1	V
adjacency list	E + V	1	outdegree(v)	outdegree(v)

† disallows parallel edges

[Sedgewick/Wayne]

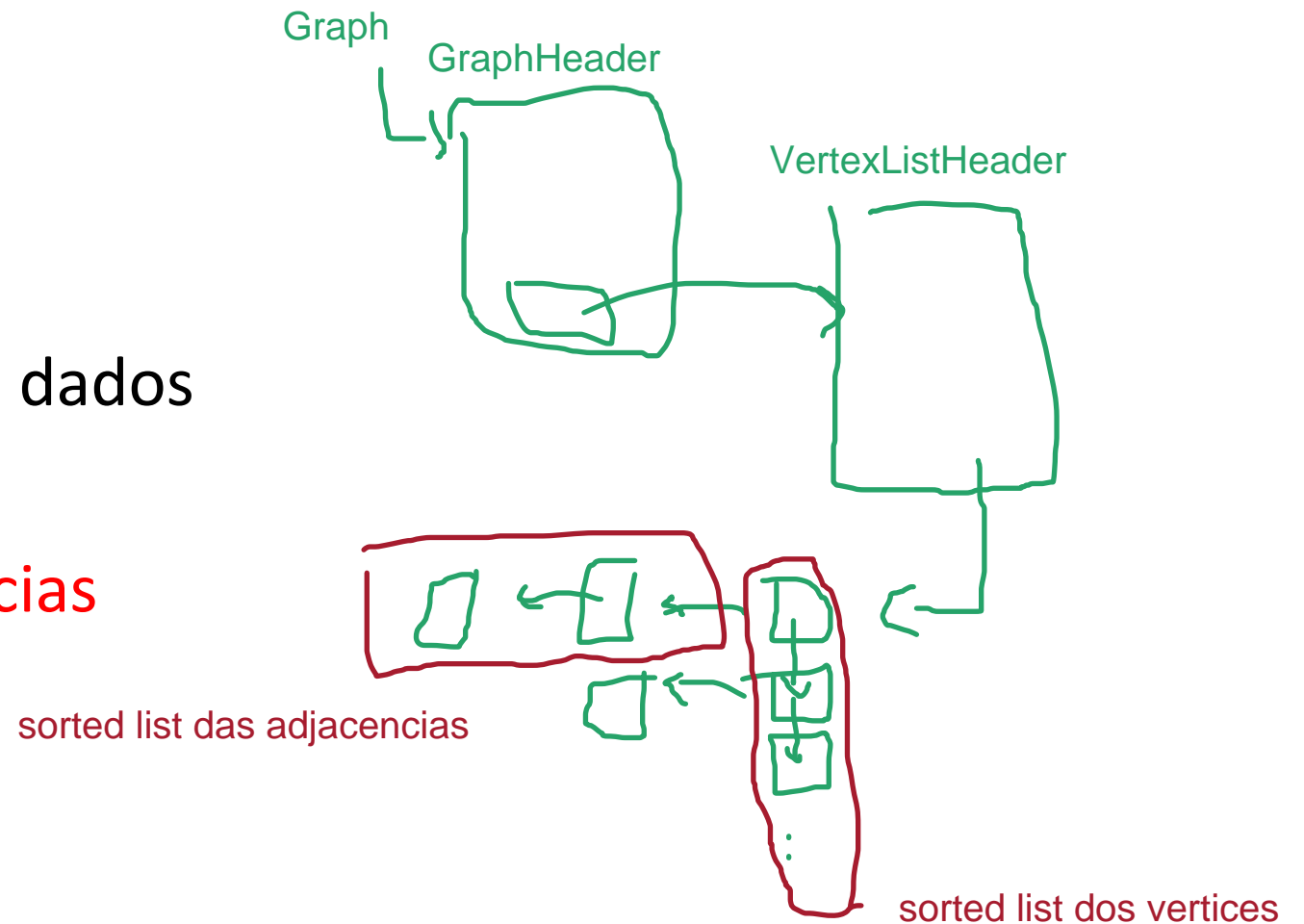
# Decisão – Um só TAD !!

- Representar **Grafos / Grafos Orientados / Redes**
- O que é **comum / diferente** ?
- Operações **básicas**, apenas !!
- **Lista** ligada de **vértices** + **Listas** ligadas de **adjacências**
- Usar o **TAD Sorted List** !!
- **Módulos adicionais** para os vários **algoritmos** !!



# Questões – Como definir ?

- As **operações básicas**
- Operações **auxiliares**
- O **cabeçalho** da estrutura de dados
- Um **nó** da **lista de vértices**
- Um **nó** das **listas de adjacências**



# Estrutura de dados



```
struct _GraphHeader {  
    unsigned short isDigraph;  
    unsigned short isComplete;  
    unsigned short isWeighted;  
    unsigned int numVertices;  
    unsigned int numEdges;  
    List* verticesList;  
};
```



```
struct _Vertex {  
    unsigned int id;  
    unsigned int inDegree;  
    unsigned int outDegree;  
    List* edgesList;  
};
```



```
struct _Edge {  
    unsigned int adjVertex;  
    int weight;  
};  
= 1, se não houver
```



# Graph.h

```
typedef struct _GraphHeader Graph;





Graph* GraphCreate(unsigned short numVertices, unsigned short isDigraph,
                  unsigned short isWeighted);

Graph* GraphCreateComplete(unsigned short numVertices,
                          unsigned short isDigraph);

void GraphDestroy(Graph** p);

Graph* GraphCopy(const Graph* g);

Graph* GraphFromFile(FILE* f);
```



Ler do ficheiro, precisa de ler as primeiras linhas,  
para saber o numVertices,....

# Graph.h


```
unsigned short GraphIsDigraph(const Graph* g);  
unsigned short GraphIsComplete(const Graph* g);  
unsigned short GraphIsWeighted(const Graph* g);  
unsigned int GraphGetNumVertices(const Graph* g);
```

# Graph.h


```
//  
// For a graph ←  
//  
double GraphGetAverageDegree(const Graph* g);  
  
//  
// For a graph ←  
//  
unsigned int GraphGetMaxDegree(const Graph* g);  
  
//  
// For a digraph ←  
//  
unsigned int GraphGetMaxOutDegree(const Graph* g);
```

assert(é digrafo?)

# Graph.h



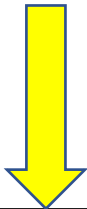
```
// Vertices
unsigned int* GraphGetAdjacentsTo(const Graph* g, unsigned int v);

// *** NEW ***
int* GraphGetDistancesToAdjacents(const Graph* g, unsigned int v);

//
// For a graph
//
unsigned int GraphGetVertexDegree(Graph* g, unsigned int v);

//
// For a digraph
//
unsigned int GraphGetVertexOutDegree(Graph* g, unsigned int v);
```

# Graph.h



```
unsigned short GraphAddEdge(Graph* g, unsigned int v, unsigned int w);

unsigned short GraphAddWeightedEdge(Graph* g, unsigned int v, unsigned int w,
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
int weight);



// CHECKING

unsigned short GraphCheckInvariants(const Graph* g);
                                ex: numero de arestas é a soma dos nós das listas de adjacencia


// DISPLAYING on the console

void GraphDisplay(const Graph* g);

void GraphListAdjacents(const Graph* g, unsigned int v);
```



# Graph.c – Questões de implementação

- Como **atravessar** a lista de vértices ?
- Como **atravessar** uma lista de adjacências ?
- Usar o **iterador** do TAD Sorted List !! 
- Como **comparar vértices** ou **arestas** ?
- Como **adicionar** uma aresta ?
- Como devolver os índices dos **vértices adjacentes** ?
- ...

# Graph.c

```
// The comparator for the VERTICES LIST

int graphVerticesComparator(const void* p1, const void* p2) {
    unsigned int v1 = ((struct _Vertex*)p1)->id;
    unsigned int v2 = ((struct _Vertex*)p2)->id;
    int d = v1 - v2;
    return (d > 0) - (d < 0);
}

// The comparator for the EDGES LISTS

int graphEdgesComparator(const void* p1, const void* p2) {
    unsigned int v1 = ((struct _Edge*)p1)->adjVertex;
    unsigned int v2 = ((struct _Edge*)p2)->adjVertex;
    int d = v1 - v2;
    return (d > 0) - (d < 0);
}
```

# Graph.c

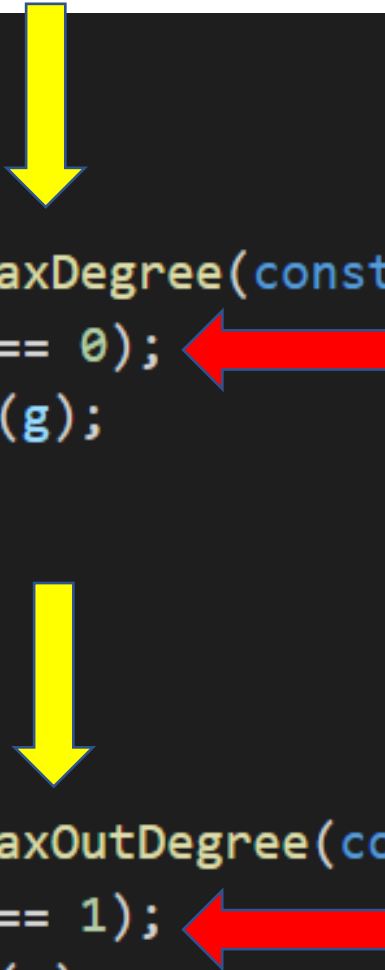
privada

```
static unsigned int _GetMaxDegree(const Graph* g) {  
    List* vertices = g->verticesList;  
    if (ListIsEmpty(vertices)) return 0;  
  
    unsigned int maxDegree = 0;  
    ListMoveToHead(vertices);  
    int i = 0;  
    for (; i < g->numVertices; ListMoveToNext(vertices), i++) {  
        struct _Vertex* v = ListGetCurrentItem(vertices);  
        if (v->outDegree > maxDegree) {  
            maxDegree = v->outDegree;  
        }  
    }  
    return maxDegree;  
}
```



# Graph.c

```
//  
// For a graph  
//  
unsigned int GraphGetMaxDegree(const Graph* g) {  
    assert(g->isDigraph == 0);  
    return _GetMaxDegree(g);  
}  
  
//  
// For a digraph  
//  
unsigned int GraphGetMaxOutDegree(const Graph* g) {  
    assert(g->isDigraph == 1);  
    return _GetMaxDegree(g);  
}
```



# Tarefas

- Analisar as funções desenvolvidas
- Completar o que falta !!
- Melhorar algumas das funções !!

# Sugestão de Leitura

# Sugestão de Leitura

- R. Sedgewick and K. Wayne, “*Algorithms*”, 4th. Ed., Addison-Wesley, 2011
  - Chapter 4