

# Grafos II

29/11/2023

# Ficheiro ZIP

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- O tipo abstrato **Grafo** usando o **TAD SortedList**
- Um módulo implementando a **Travessia em Profundidade**

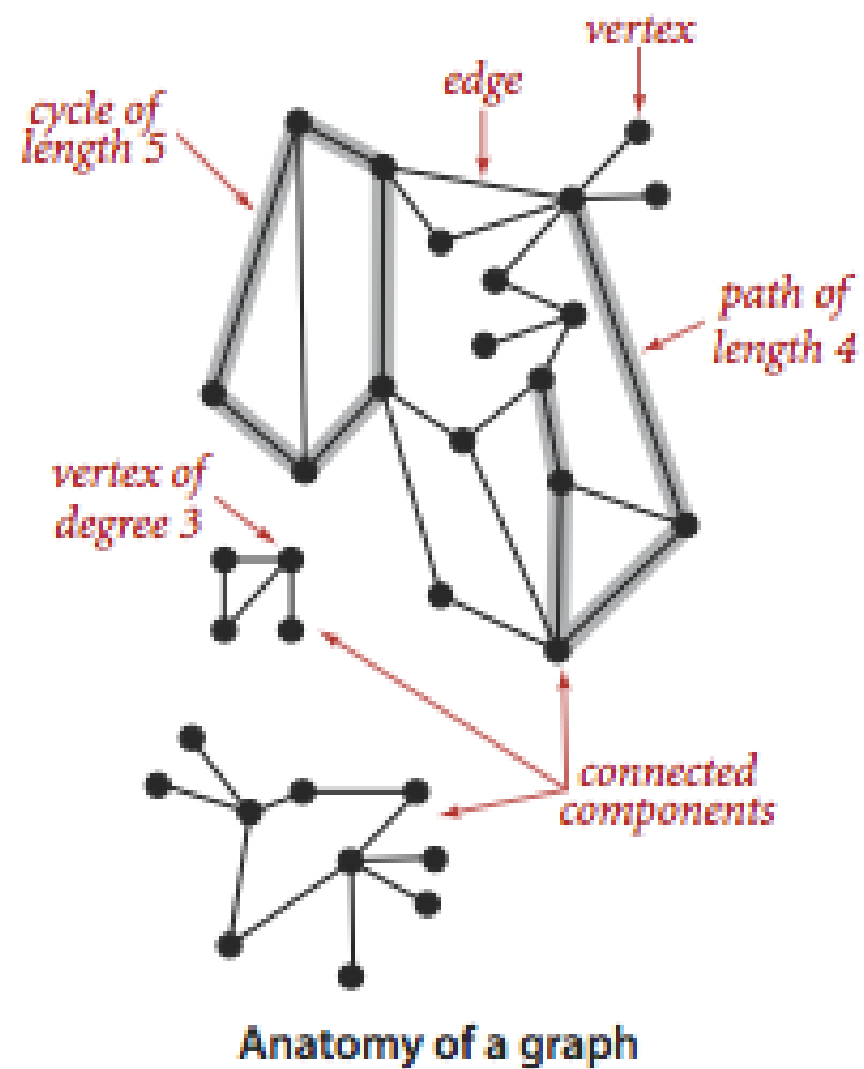
# Sumário

- Recap
- Travessia em Profundidade (“Depth-First”)
- Travessia por Níveis (“Breadth-First”)
- Ordenação Topológica
- Sugestões de leitura

Let's  
RECAP

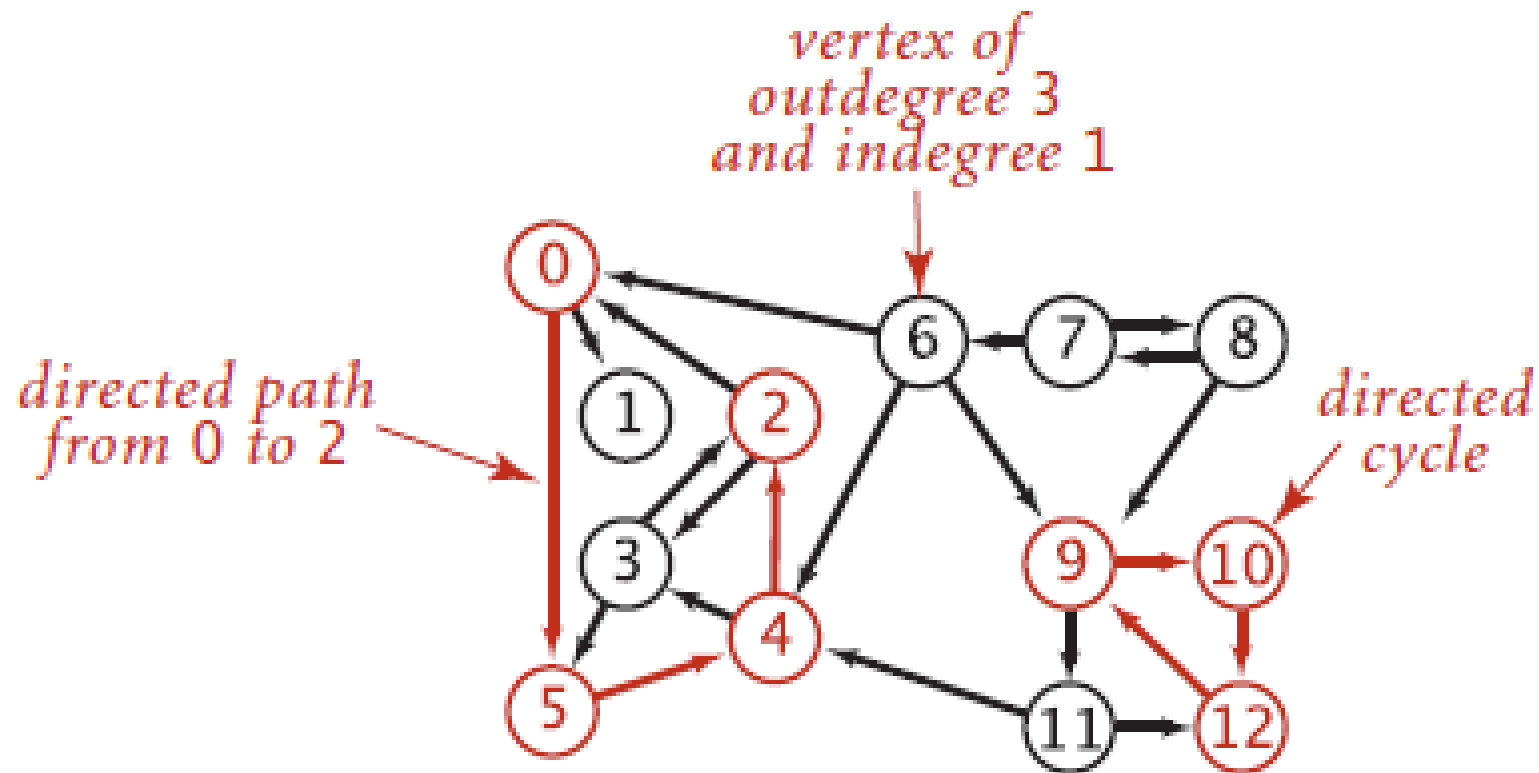
# Recapitulação

# Grafo



[Sedgewick & Wayne]

# Grafo orientado



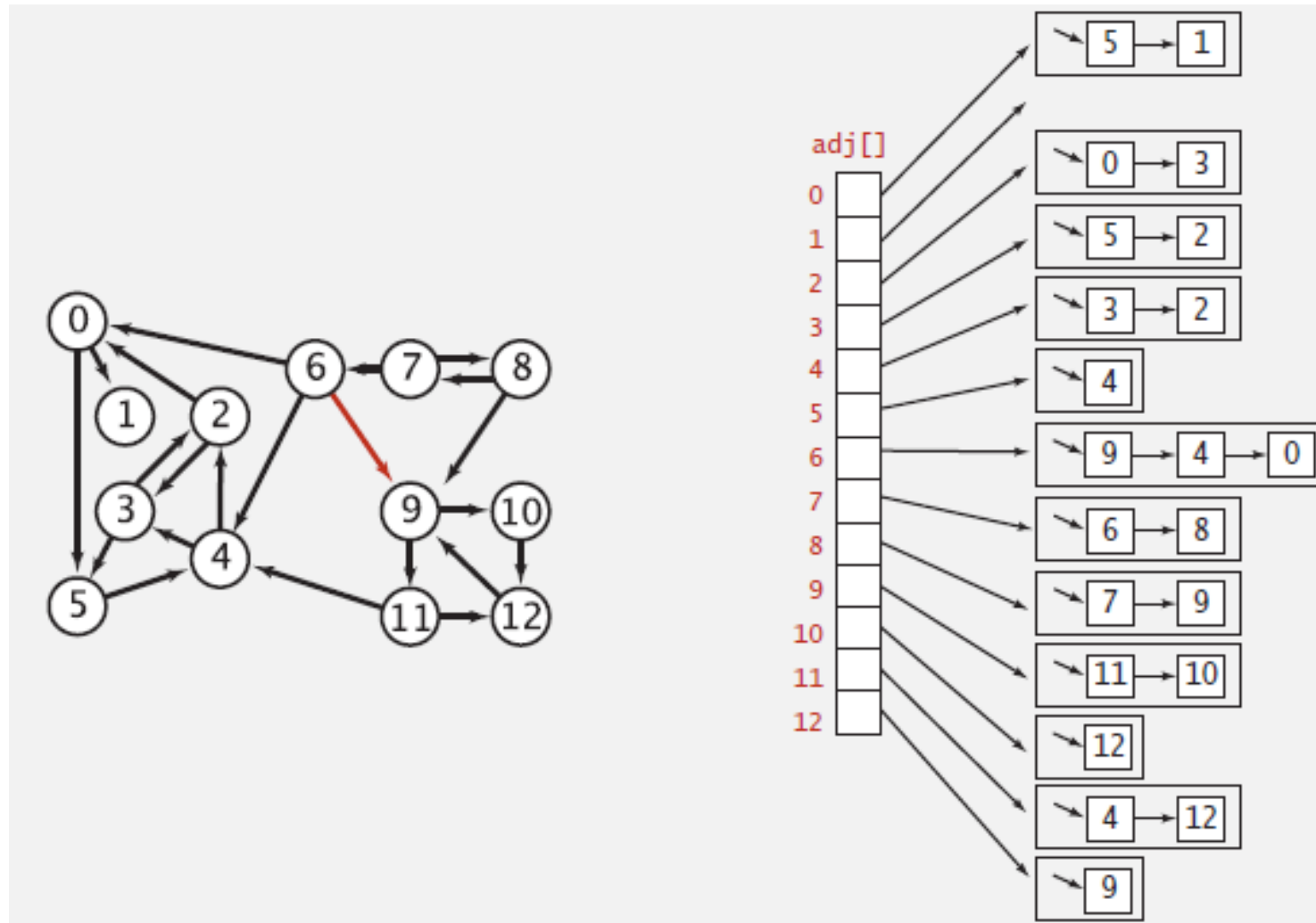
[Sedgewick/Wayne]

# 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** !!




# Representação – Listas de adjacências



[Sedgewick/Wayne]



# 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** ?
- ...

# 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;  
};
```

# 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);
```



# Graph.h

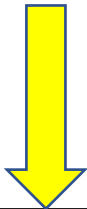
```
// Vertices
unsigned int* GraphGetAdjacentsTo(const Graph* g, unsigned int v);

// *** 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);

// DISPLAYING on the console

void GraphDisplay(const Graph* g);

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



# Graph.c

```
// The comparator for the VERTICES LIST

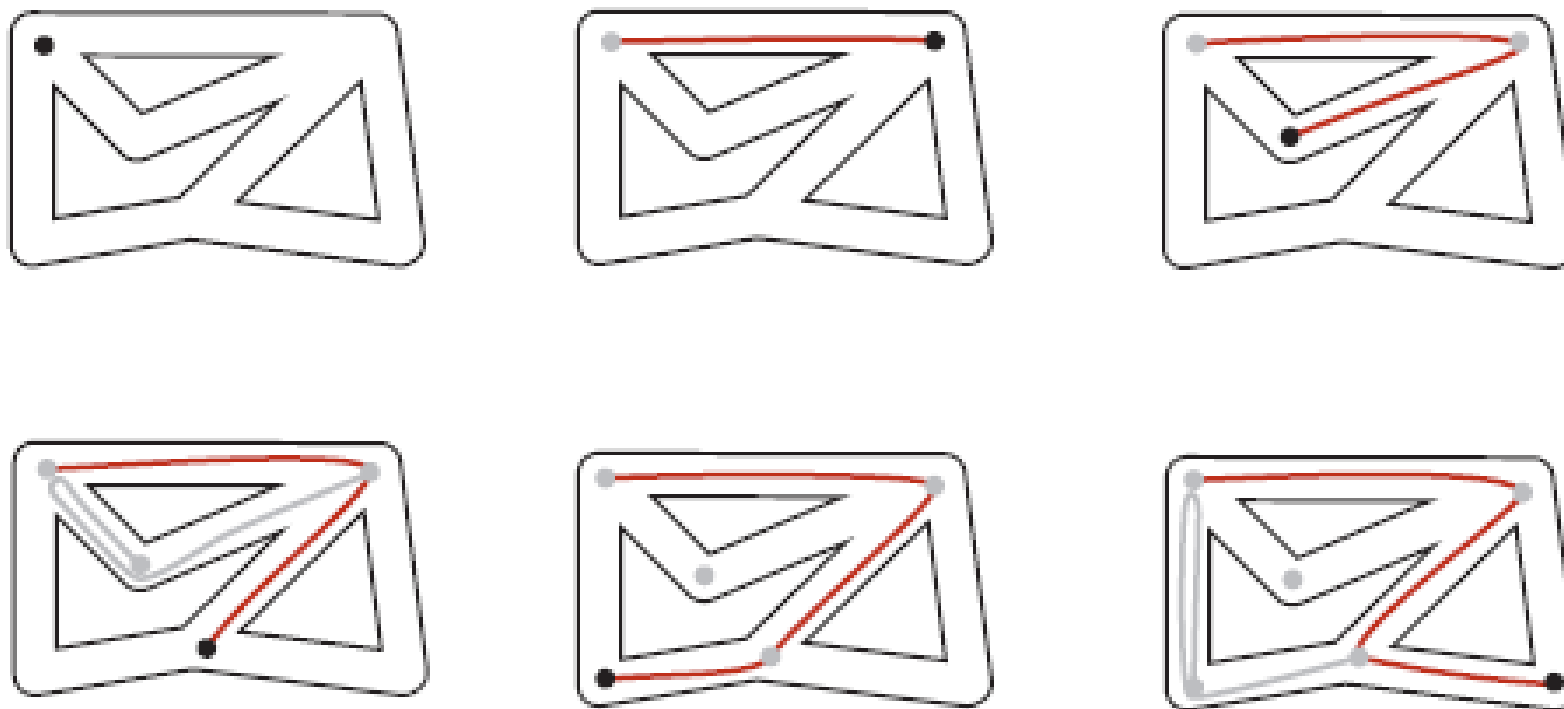
int graphVerticesComparator(const void* p1, const void* p2) {
    unsigned int v1 = ((struct _Vertex*)p1)->id;
    unsigned int v2 = ((struct _Vertex*)p2)->id;
    int d = v1 - v2;
    return (d > 0) - (d < 0);
}

// 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);
}
```

# Travessia em Profundidade

# Exploração de um labirinto



[Sedgewick/Wayne]



# Travessia em profundidade – Depth-First

- Exploração / travessia sistemática de (todo) um grafo ou grafo orientado
- **Aplicações :**
  - Identificar os **vértices alcançáveis** a partir de um vértice inicial
  - Encontrar um **caminho entre dois vértices**, caso exista
  - Encontrar um **caminho** entre o **vértice inicial** e **cada um dos outros vértices alcançáveis**, caso exista
  - ...

# Travessia em profundidade – Depth-First

- Algoritmo **idêntico** ao da travessia em profundidade de uma árvore binária
- Versão **recursiva** / Versão **iterativa** com **PILHA/STACK**
- **DIFERENÇAS :**
  - Há um vértice inicial – **start vertex** – s
  - O número de vértices adjacentes é **variável**
  - Podem haver **ciclos** e/ou **mais do que um caminho** para cada vértice
  - Para não entrar em ciclo, **marcar os vértices visitados !!**

# Algoritmo recursivo

Travessia em Profundidade (vértice  $v$ )

Marcar  $v$  como visitado

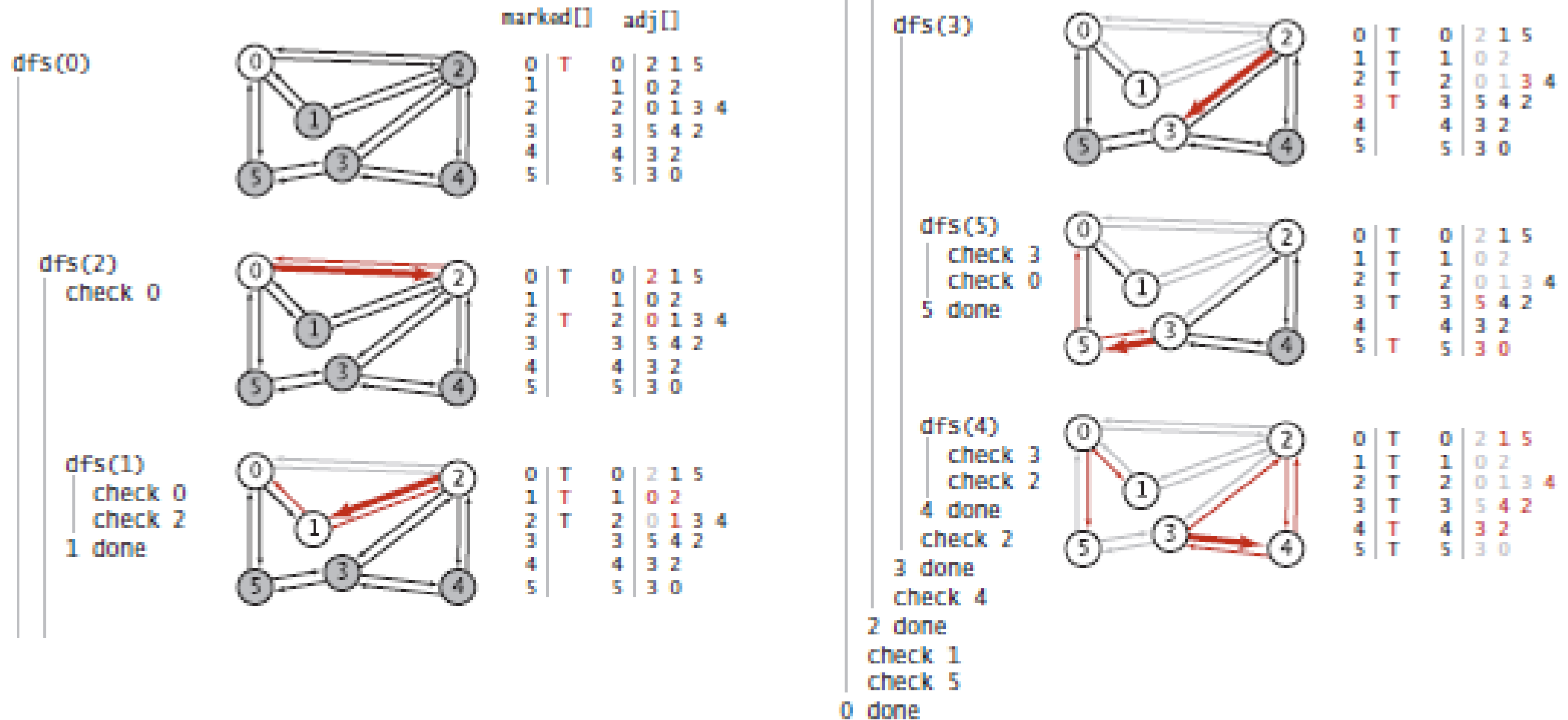
Para cada vértice  $w$  adjacente a  $v$

Se  $w$  não está marcado como visitado

Então efetuar a Travessia em Profundidade ( $w$ )

- Resultado ?
- Ficam **marcados** todos os **vértices alcançados**

# Exemplo



[Sedgewick/Wayne]

# Algoritmo iterativo – A mesma ordem ?

Travessia em Profundidade (vértice  $v$ )

Criar um STACK vazio

Push(stack,  $v$ )

Marcar  $v$  como visitado

Enquanto NãoVazio(stack) fazer

$v = \text{Pop}(\text{stack})$

Para cada vértice  $w$  adjacente a  $v$

Se  $w$  não está marcado como visitado

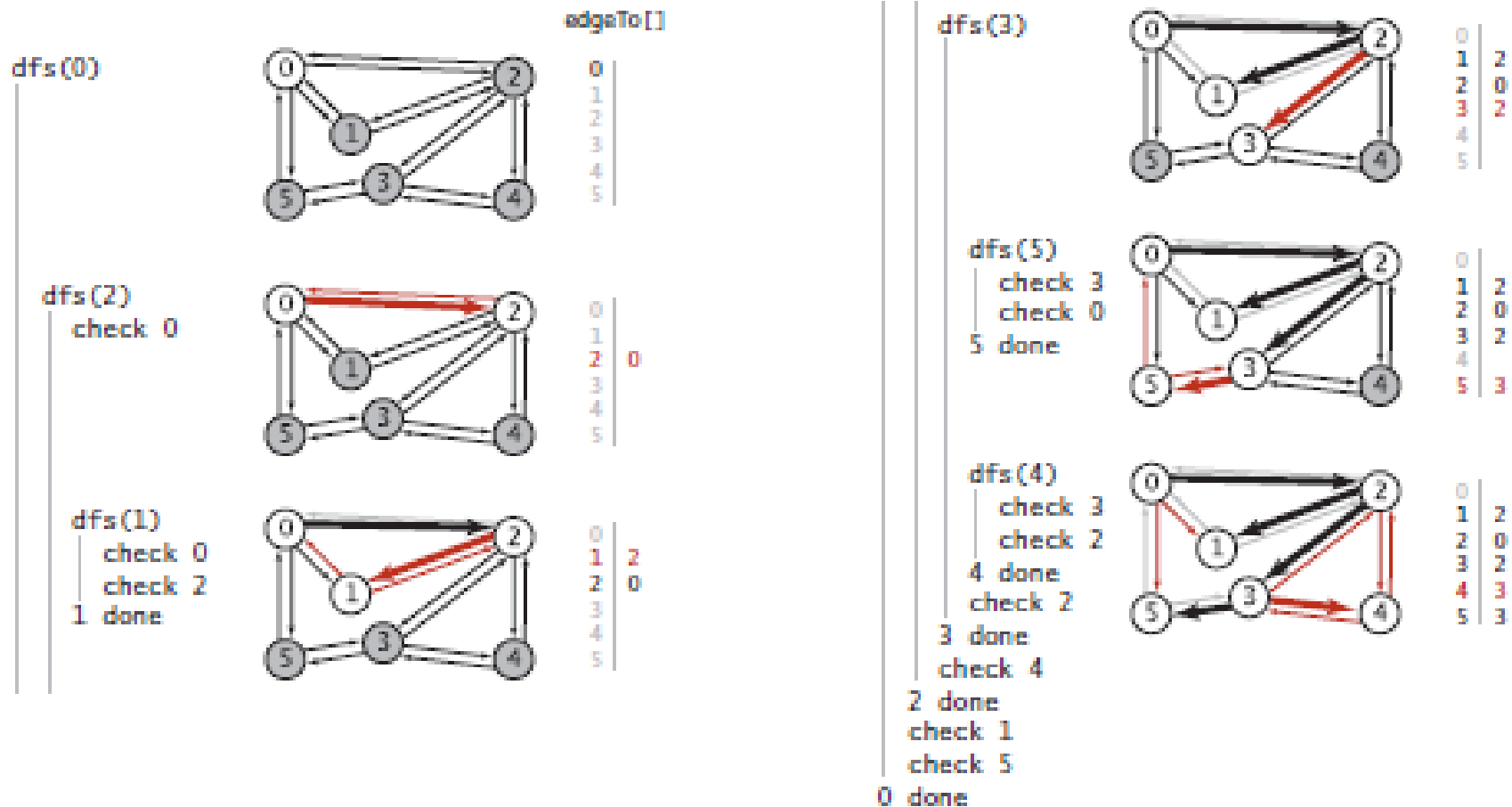
Então Push(stack,  $w$ )

Marcar  $w$  como visitado

# Vértices alcançáveis

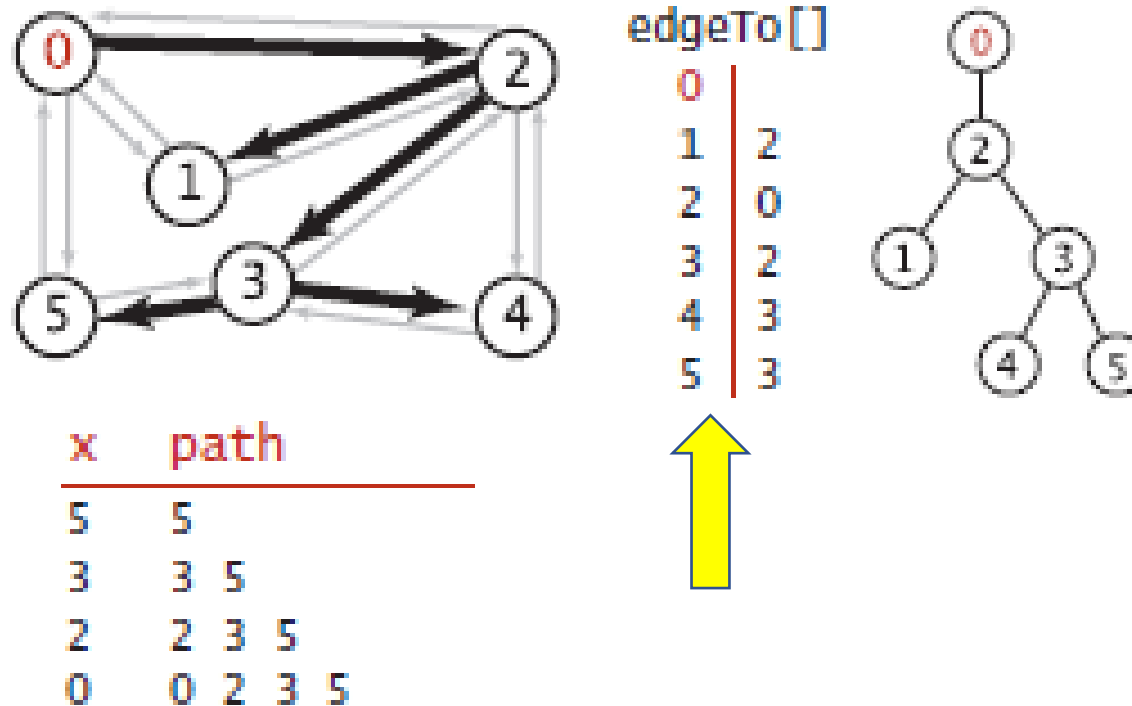
- Determinar o conjunto dos **vértices alcançáveis** significa encontrar um **caminho** entre o **vértice inicial** e cada um dos **vértices alcançados**
  - Pode não ser o caminho mais curto !!
  - **Porquê ?**
- **Árvore de caminhos** com raiz no vértice inicial
- Como **registar** a árvore ?
- **Fácil** : registar o **predecessor** de cada vértice no caminho a partir do vértice inicial
- Fazer o **“traceback”** para obter a sequência de vértices definindo o caminho

# Árvore dos caminhos com origem em 0



[Sedgewick/Wayne]




# Árvore dos caminhos com origem em 0






[Sedgewick/Wayne]








# GraphDFSRec.h



```
typedef struct _GraphDFSRec GraphDFSRec;  
  
GraphDFSRec* GraphDFSRecExecute(Graph* g, unsigned int startVertex);  
  
void GraphDFSRecDestroy(GraphDFSRec** p);  
  
// Getting the result  
  
 unsigned int GraphDFSRecHasPathTo(const GraphDFSRec* p, unsigned int v);  
  
 Stack* GraphDFSRecPathTo(const GraphDFSRec* p, unsigned int v);  
  
// DISPLAYING on the console  
  
 void GraphDFSRecShowPath(const GraphDFSRec* p, unsigned int v);
```

# GraphDFSRec.c

```
struct _GraphDFSRec {  
    unsigned int* marked;  
    int* predecessor;  
    Graph* graph;  
    unsigned int startVertex;  
};
```

```
static void _dfs(GraphDFSRec* traversal, unsigned int vertex) {  
    traversal->marked[vertex] = 1;   
  
    unsigned int* neighbors = GraphGetAdjacentsTo(traversal->graph, vertex);  
  
    for (int i = 1; i <= neighbors[0]; i++) {   
        unsigned int w = neighbors[i];  
        if (traversal->marked[w] == 0) {   
            traversal->predecessor[w] = vertex;  
            _dfs(traversal, w);   
        }  
    }  
  
    free(neighbors);   
}
```

# GraphDSFRec.c

```
Stack* GraphDFSRecPathTo(const GraphDFSRec* p, unsigned int v) {  
    assert(0 <= v && v < GraphGetNumVertices(p->graph));  
  
    Stack* s = StackCreate(GraphGetNumVertices(p->graph));  
  
    if (p->marked[v] == 0) {  
        return s;  
    }  
  
    // Store the path  
    for (unsigned int current = v; current != p->startVertex;  
         current = p->predecessor[current]) {  
        StackPush(s, current);  
    }  
  
    StackPush(s, p->startVertex);  
  
    return s;  
}
```

# Tarefas

- **Analisar** o ficheiro GraphDFSRec.c
- **NOVO MÓDULO :**
- **Implementar** e testar a **versão iterativa** usando uma **PILHA/STACK**
- **Questão :**
- Os vértices de um grafo são atravessados na mesma ordem que na versão recursiva ?

# Travessia por Níveis

# Travessia por níveis – Breadth-First

- Algoritmo **idêntico** ao da travessia por níveis de uma árvore binária
- **Versão iterativa** com **FILA/QUEUE**
- **Idêntico** à travessia em profundidade iterativa de um grafo
- **MAS**, usando uma estrutura de dados auxiliar distinta
- A **ordem** pela qual os **vértices** são **visitados** é **diferente** !!
- Progressão em **círculos concêntricos** a partir do vértice inicial
- **APLICAÇÃO** : determinar **caminhos mais curtos** !!

# Algoritmo iterativo

Travessia por Níveis(vértice  $v$ )

Criar FILA vazia

Enqueue(queue,  $v$ )

Marcar  $v$  como visitado

Enquanto NãoVazia(queue) fazer

$v = \text{Dequeue}(\text{queue})$

    Para cada vértice  $w$  adjacente a  $v$

        Se  $w$  não está marcado como visitado

        Então Enqueue(queue,  $w$ )

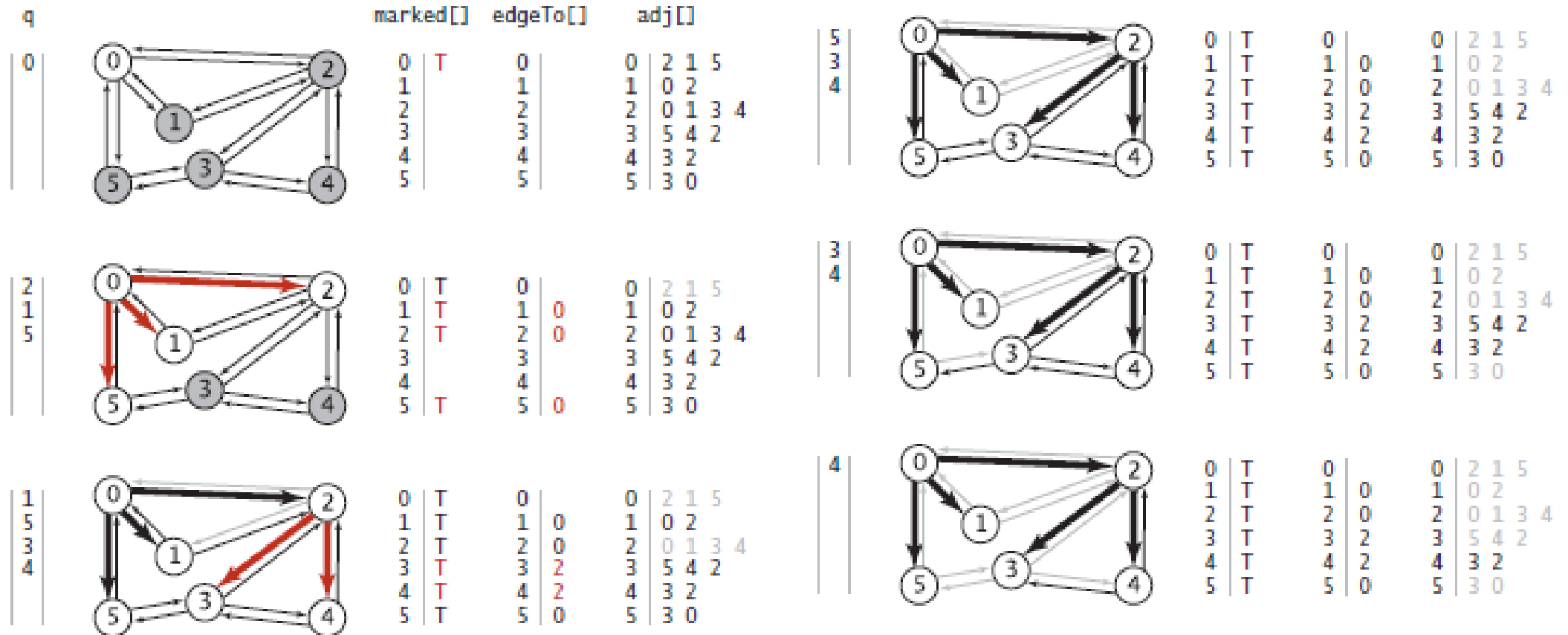
        Marcar  $w$  como visitado

# Caminhos mais curtos

- É encontrado o **caminho com menor número de arestas** entre o **vértice inicial** e cada um dos **vértices alcançados**
  - Porquê ?
- **Árvore de caminhos mais curtos** com raiz no vértice inicial
- Registrar o **predecessor** de cada vértice no caminho a partir do vértice inicial
- E a **distância** (i.e., **nº de arestas**) para o vértice inicial
- Fazer o **“traceback”** para obter a sequência de vértices definindo o caminho



# Árvore dos caminhos mais curtos



[Sedgewick/Wayne]

# Tarefa

- **NOVO MÓDULO :**
- Implementar e testar a travessia por níveis usando uma **FILA/QUEUE**

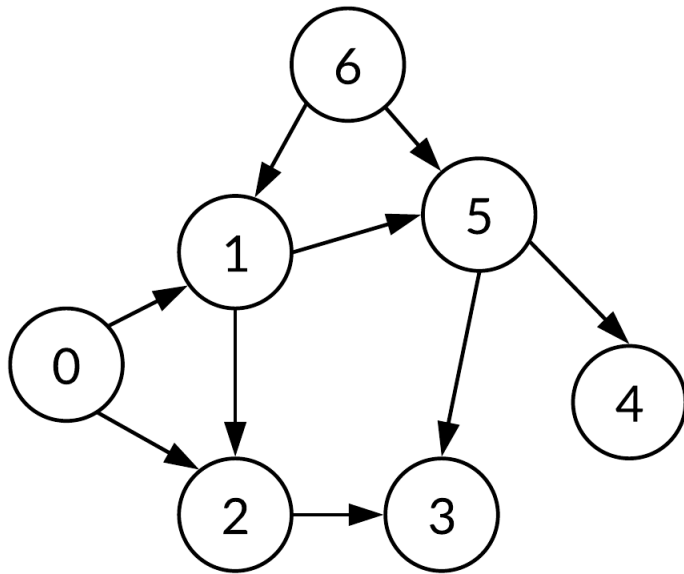
# Ordenação Topológica

# Ordenação Topológica

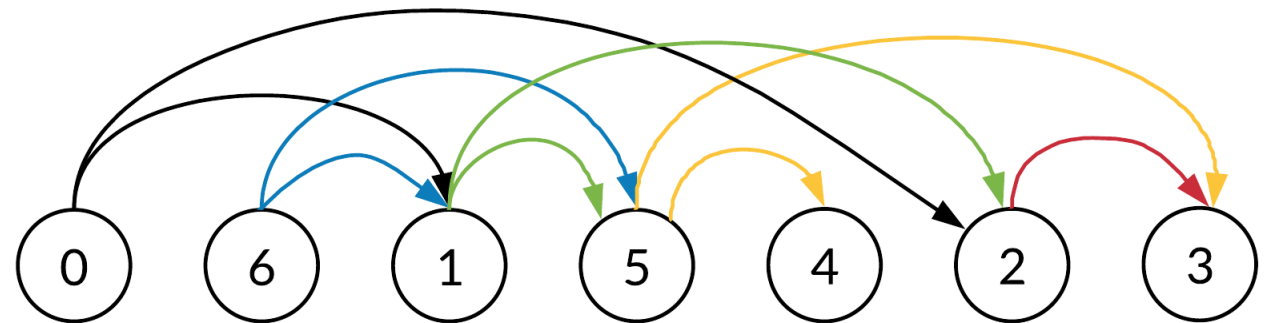
- Podemos **desenhar** um dado **grafo orientado** de maneira a que **todas as arestas apontem para o mesmo lado** ?
- Dado um conjunto de **tarefas** a realizar, e as respectivas **precedências**, qual a **ordem** pela qual devem ser **escaloadas** ?
  - Usar **BFS** ou **DFS** !
  - Representar a solução com um **grafo orientado acíclico** !
- Aplicação : **verificar** se um **grafo orientado** é **acíclico** ou não

# Exemplo – Ordenação dos vértices

Unsorted graph

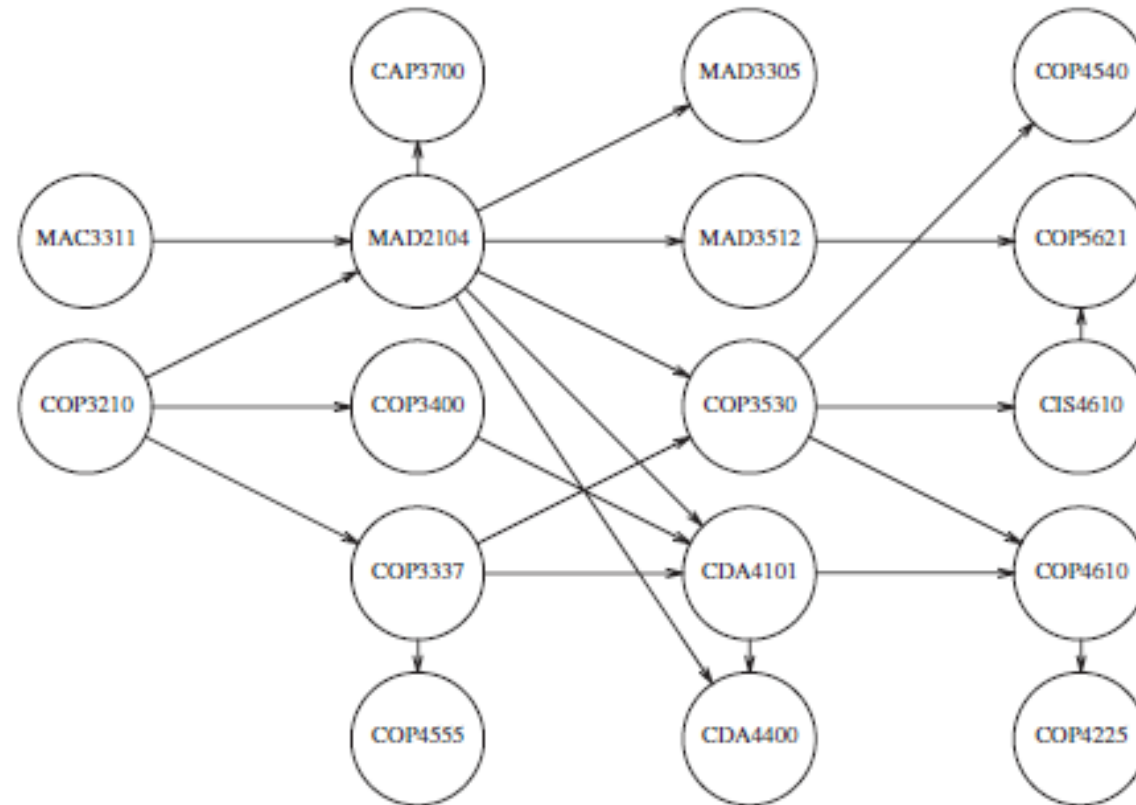


Topologically sorted graph



[guides.codepath.com]

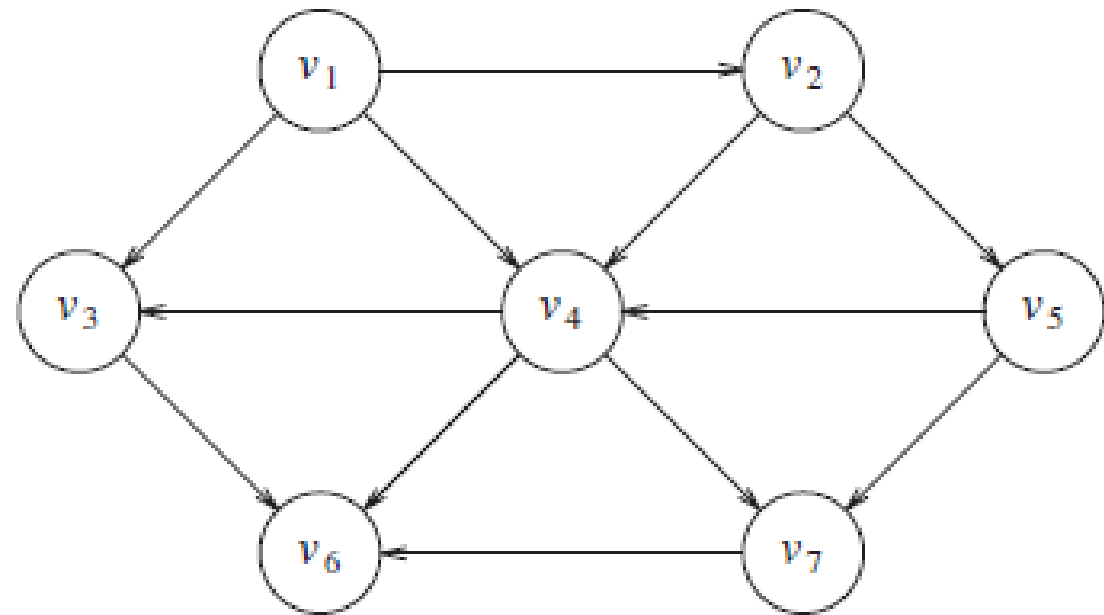
# Grafo das precedências das UCs de um curso



# Ordenação Topológica

- Como **ordenar** as UCs de acordo com as **precedências** definidas ?
- Grafo **orientado** e **acíclico** !!
- Ordem ?
- Se existe um **caminho de v para w**, então **w aparece após v** na sequência de vértices ordenados
- **Não** podem existir **ciclos** !!
- Pode haver **mais do que uma ordenação** válida !!

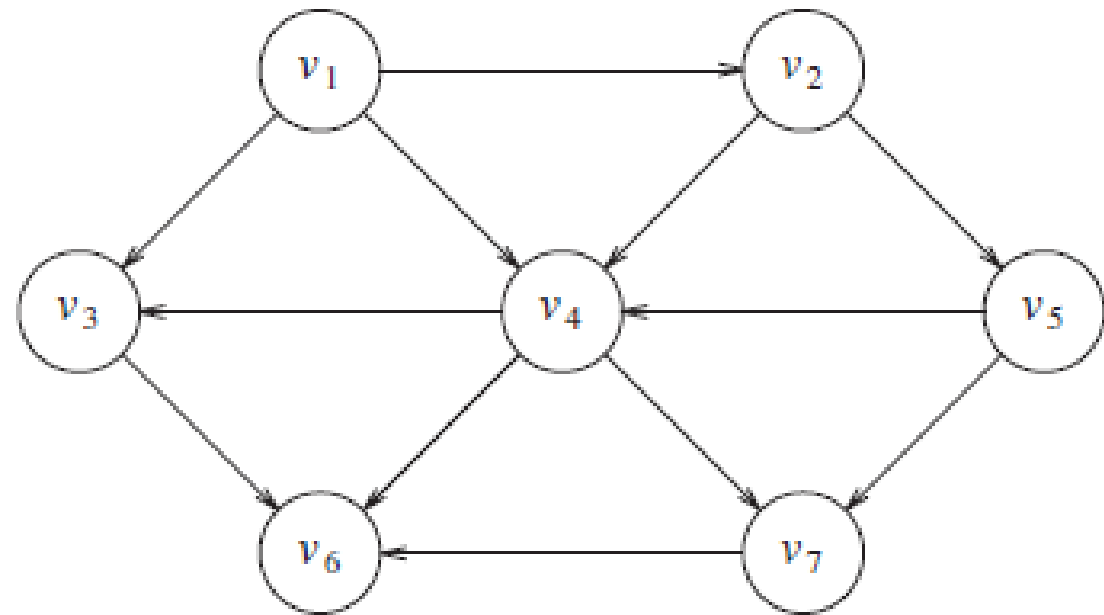
# Exemplo



- Possíveis sequências de vértices ?



# Exemplo



- Possíveis sequências de vértices ?
- v1, v2, v5, v4, v3, v7, v6 **OU** v1, v2, v5, v4, v7, v3, v6
- Como determinar ?

# 1º algoritmo – Cópia do grafo $G$

Criar  $G'$ , uma cópia do grafo  $G$

Enquanto for possível

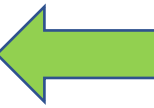
    Selecionar um vértice sem arestas incidentes

    Imprimir o seu ID

    Apagar esse vértice de  $G'$  e as arestas que dele emergem

- Usar o InDegree de cada vértice
- Ineficiência : cópia + sucessivas procuras através do conjunto de vértices

## 2º algoritmo – Array auxiliar



Registrar num array auxiliar **numEdgesPerVertex** o InDegree de cada vértice  
Enquanto for possível

Selecionar **vértice v** com **numEdgesPerVertex[v] == 0** **E** não marcado

**Imprimir** o seu ID

Marcá-lo como pertencendo à ordenação

Para cada vértice **w** adjacente a **v**

**numEdgesPerVertex[w]--**

- **Ineficiência** : sucessivas procuras através do conjunto de vértices

### 3º alg. – Manter o conjunto de candidatos

Registrar num array auxiliar `numEdgesPerVertex` o InDegree de cada vértice

Criar **FILA vazia** e **inserir** na FILA os **vértices  $v$**  com `numEdgesPerVertex[v] == 0`

Enquanto a FILA não for vazia

**$v$**  = retirar **próximo vértice** da FILA

**Imprimir** o seu ID

Para cada vértice  **$w$**  adjacente a  **$v$**

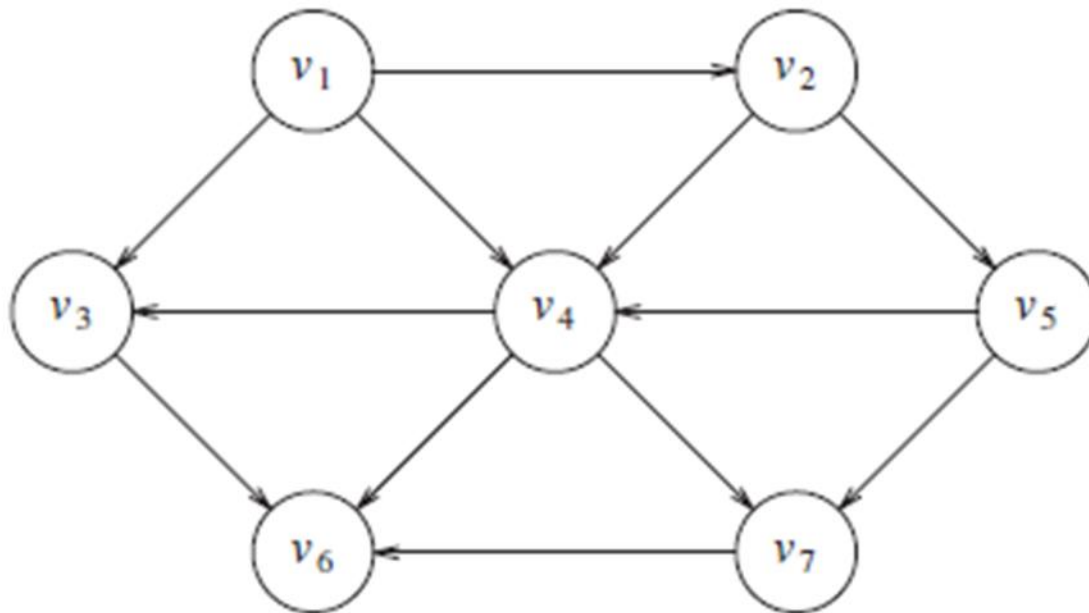
`numEdgesPerVertex[w] --`

Se `numEdgesPerVertex[w] == 0` Então **Inserir  $w$**  na **FILA**



- **PROBLEMA** : o que acontece se existir um ciclo ??

# Exemplo



Vertex	Indegree Before Dequeue #						
	1	2	3	4	5	6	7
$v_1$	0	0	0	0	0	0	0
$v_2$	1	0	0	0	0	0	0
$v_3$	2	1	1	1	0	0	0
$v_4$	3	2	1	0	0	0	0
$v_5$	1	1	0	0	0	0	0
$v_6$	3	3	3	3	2	1	0
$v_7$	2	2	2	1	0	0	0
Enqueue	$v_1$	$v_2$	$v_5$	$v_4$	$v_3, v_7$		$v_6$
Dequeue	$v_1$	$v_2$	$v_5$	$v_4$	$v_3$	$v_7$	$v_6$

# Sugestões de Leitura

# Sugestões de leitura

- M. A. Weiss, *“Data Structures and Algorithm Analysis in C++”*, 4th. Ed., Pearson, 2014
  - Chapter 9
- R. Sedgewick and K. Wayne, *“Algorithms”*, 4th. Ed., Addison-Wesley, 2011
  - Chapter 4