Grafos II

26/11/2025

Ficheiro ZIP

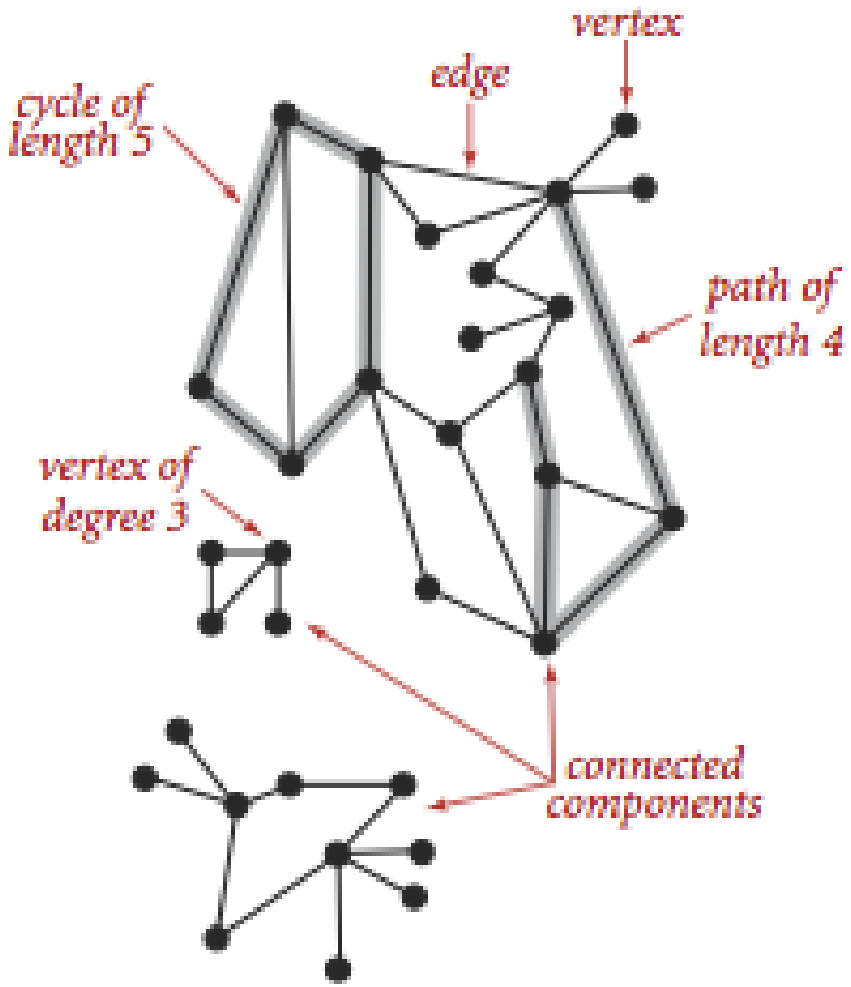
- Está disponível no Moodle um ficheiro ZIP de suporte aos tópicos de hoje
- O tipo abstrato Grafo usando o TAD SortedList
- Um módulo implementando a Travessia em Profundidade

Sumário

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

Recapitulação

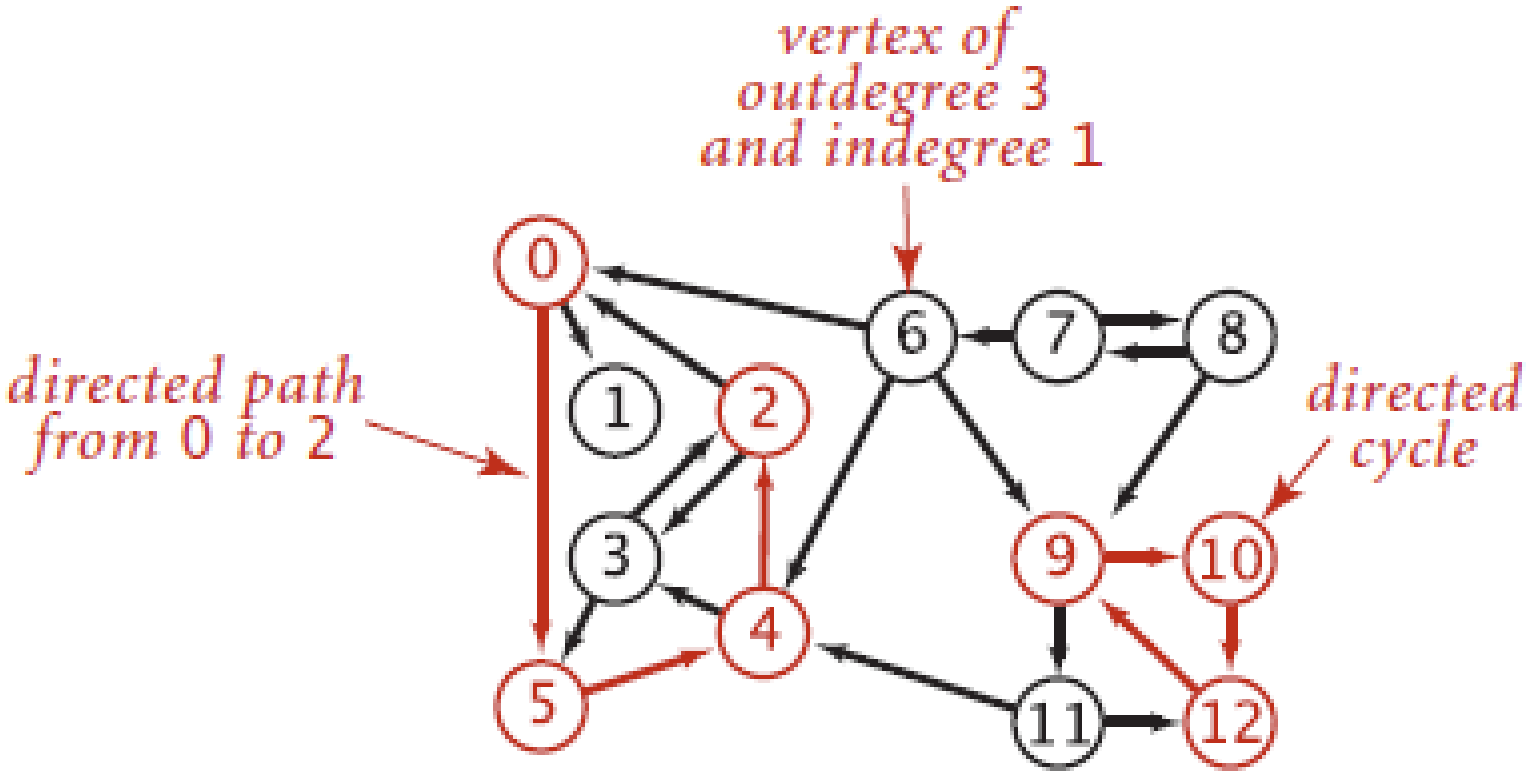
Grafos – Resumo



Anatomy of a graph

[Sedgewick & Wayne]

Grafos Orientados – Resumo



[Sedgewick/Wayne]

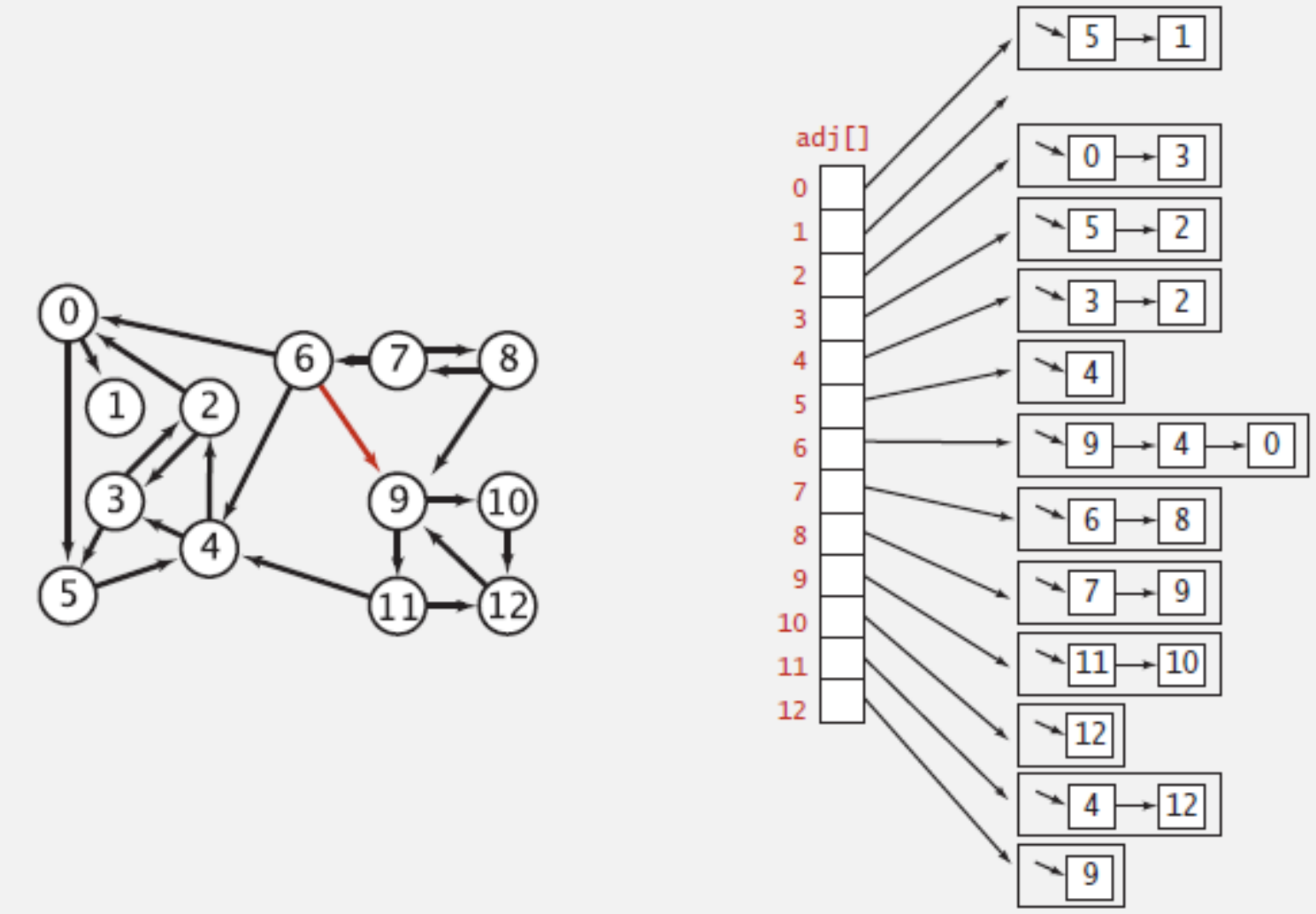
TAD GRAFO – Decisão – Um só TAD !!

- 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]

TAD GRAFO – 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 ?

- …

TAD GRAFO – Cabeçalho / Vértice / Aresta

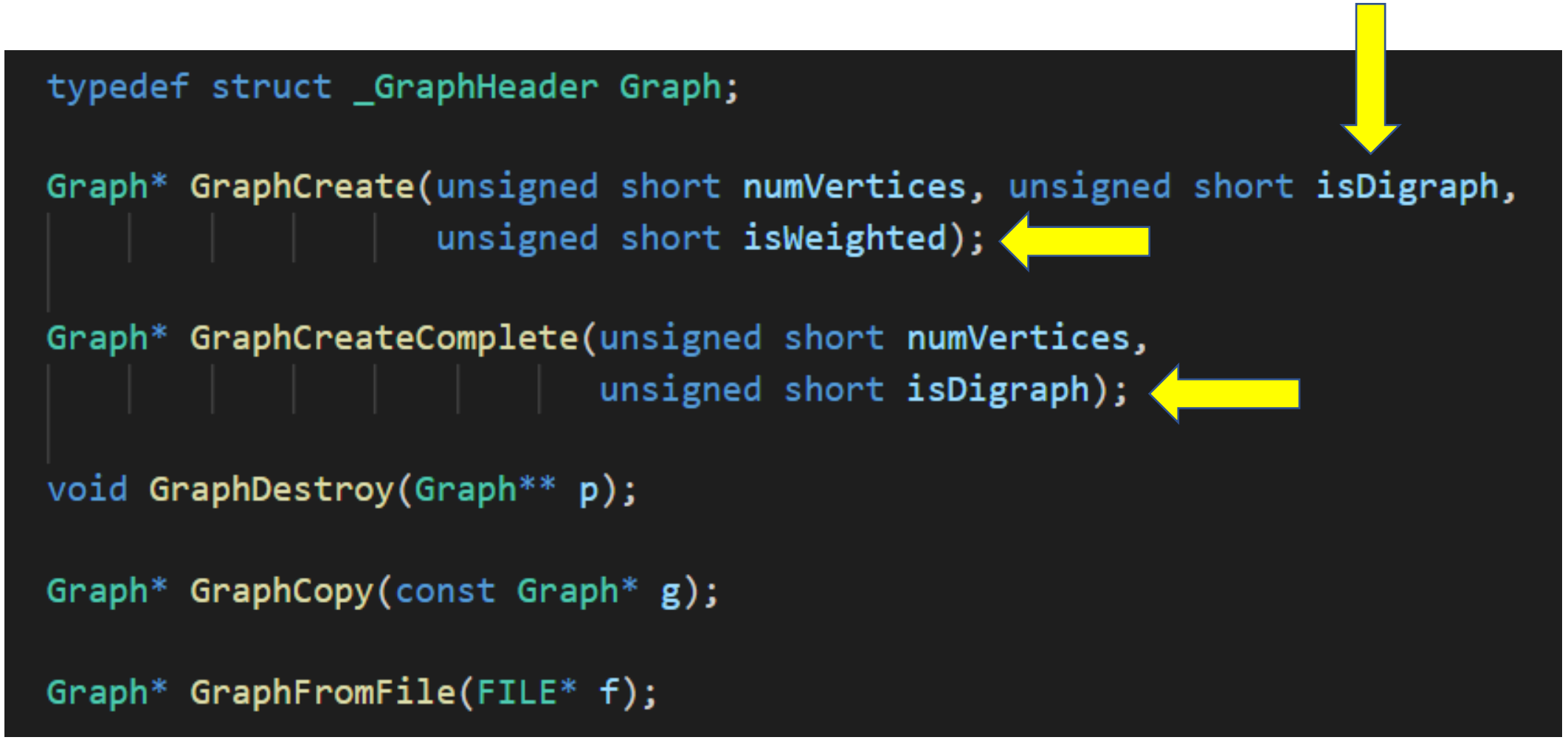
```
struct _GraphHeader {
  unsigned short isDigraph;
  unsigned short isComplete;
  unsigned short isWeighted;
  unsigned int numVertices;
  unsigned int numEdges;
  List* verticesList;
};
```

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

```
struct _Edge {
  unsigned int adjVertex;
  int weight;
};
```

- Os atributos do cabeçalho permitem classificar o grafo
- Se o grafo for não-orientado, é suficiente armazenar o (out)Degree de cada vértice

TAD GRAFO – Criar e destruir grafos



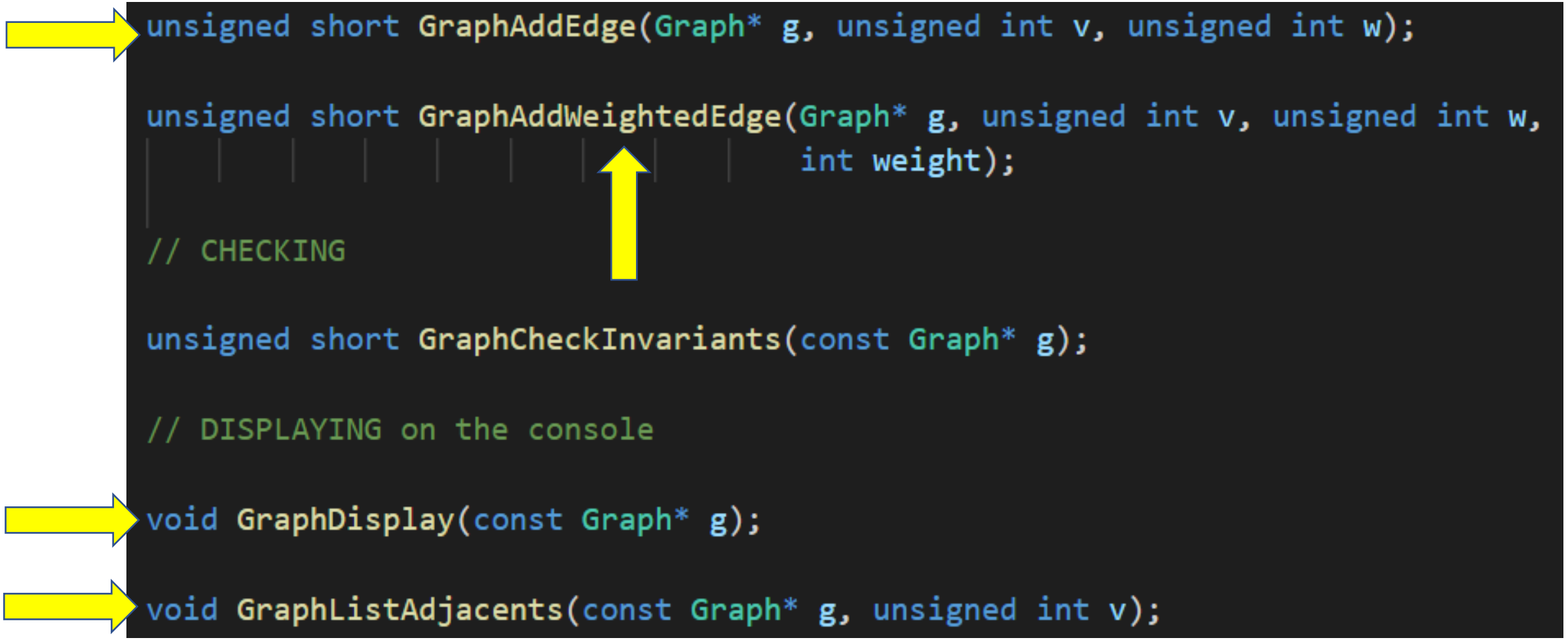
```
typedef struct _GraphHeader Graph;

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);
```

TAD GRAFO – Propriedades de um vértice

- Array com os IDs dos vértices adjacentes
- Array com as distâncias aos vértices adjacentes

```
// 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);
```

TAD GRAFO – Adicionar arestas



```
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);
```

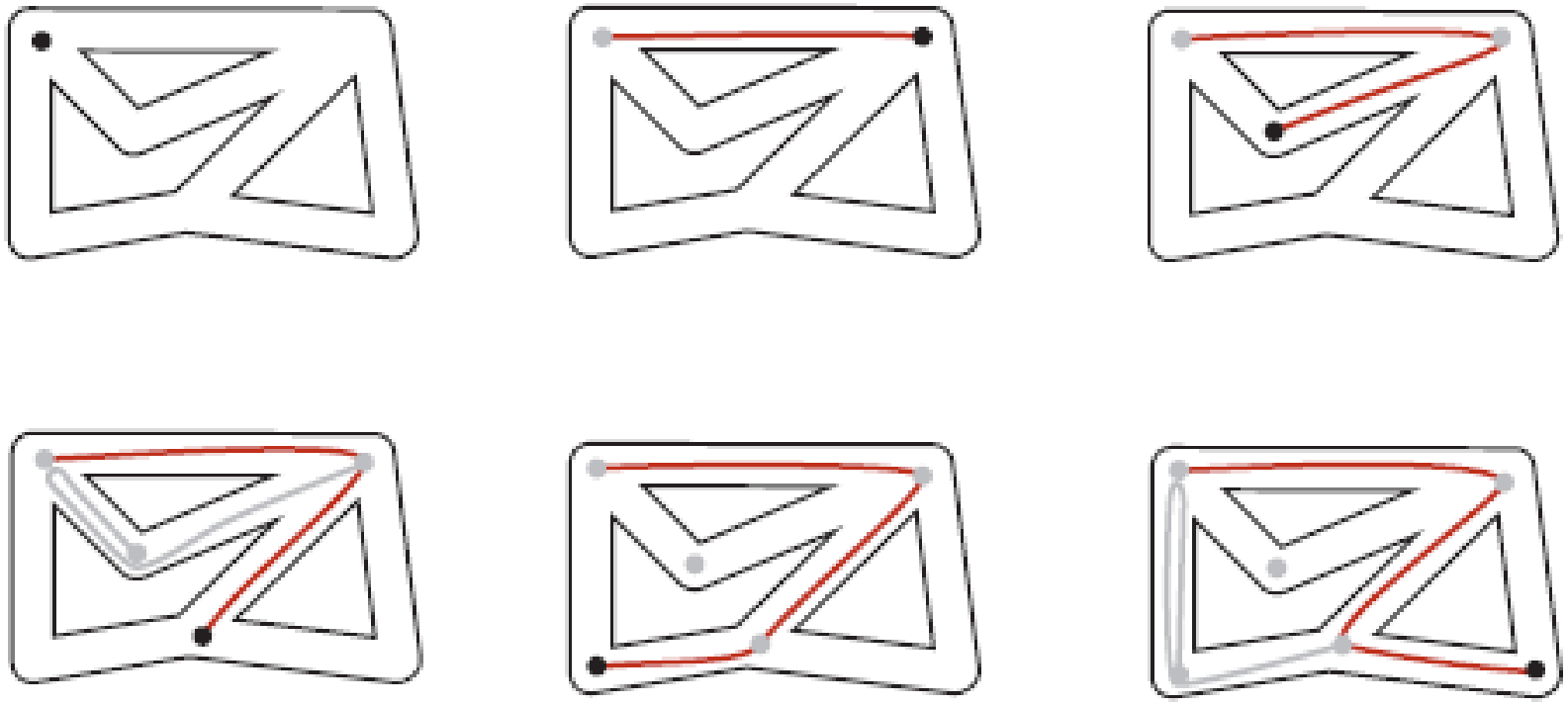
TAD GRAFO

- Funções de comparação para o TAD SORTED LIST
- Para a lista de vértices, comparar vértices usando os seus IDs
- Para a lista de arestas adjacentes, comparar arestas usando o seu vértice final

```
// 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 – Depth-First Traversal

Como explorar um labirinto ?



[Sedgewick/Wayne]

Travessia em Profundidade – Depth-First

- Exploração / travessia sistemática de (todo) um grafo ou grafo orientado

- **Aplicações** :
- Encontrar um caminho entre dois vértices, caso exista
- Identificar os vértices alcançáveis a partir de um vértice inicial
- 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
- Para cada vértice, o número de vértices adjacentes é variável
- Poderá haver ciclos e/ou mais do que um caminho para cada vértice
- Para não entrar em ciclo, **marcar os vértices visitados !!**

Depth-First Traversal – Algoritmo recursivo

Travessia em Profundidade (vértice v)

Marcar v como visitado

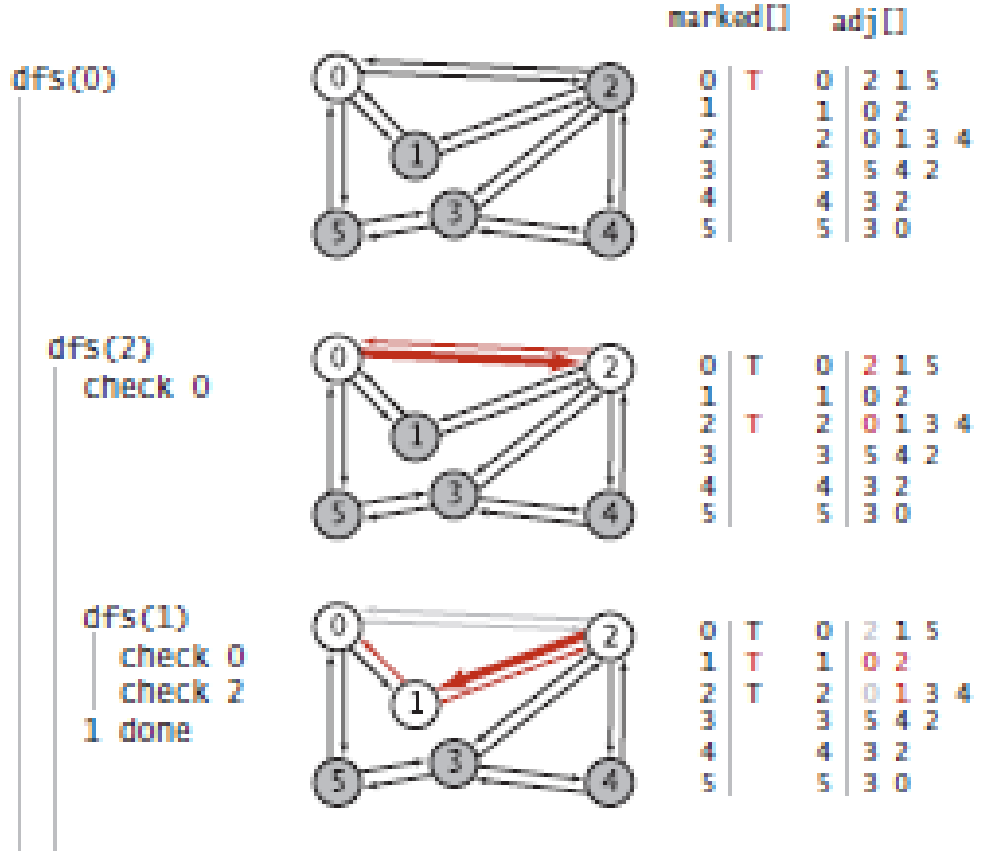
Para cada vértice w adjacente a v

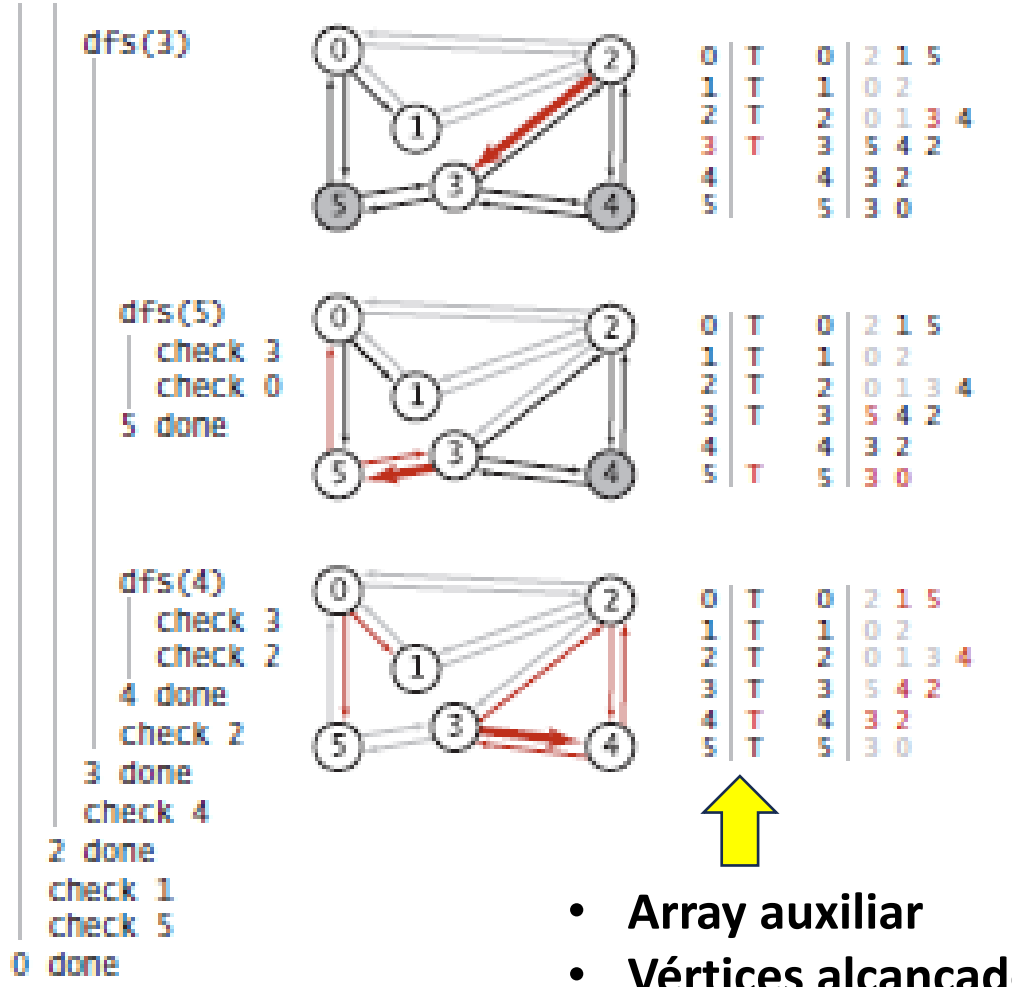
Se w não está marcado como visitado

Então efetuar a Travesssia em Profundidade (w)

- Resultado ?
- Ficam marcados todos os vértices alcançados
  - Array auxiliar

Exemplo – Algoritmo recursivo





- **Array auxiliar**
- **Vértices alcançados**

[Sedgewick/Wayne]

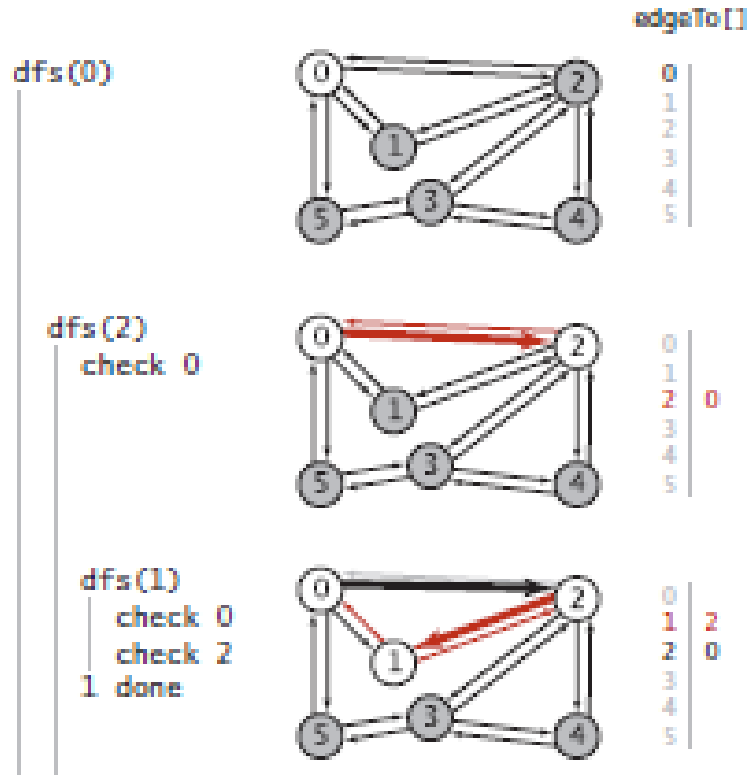
Alg. iterativo – Travessia na mesma ordem ?

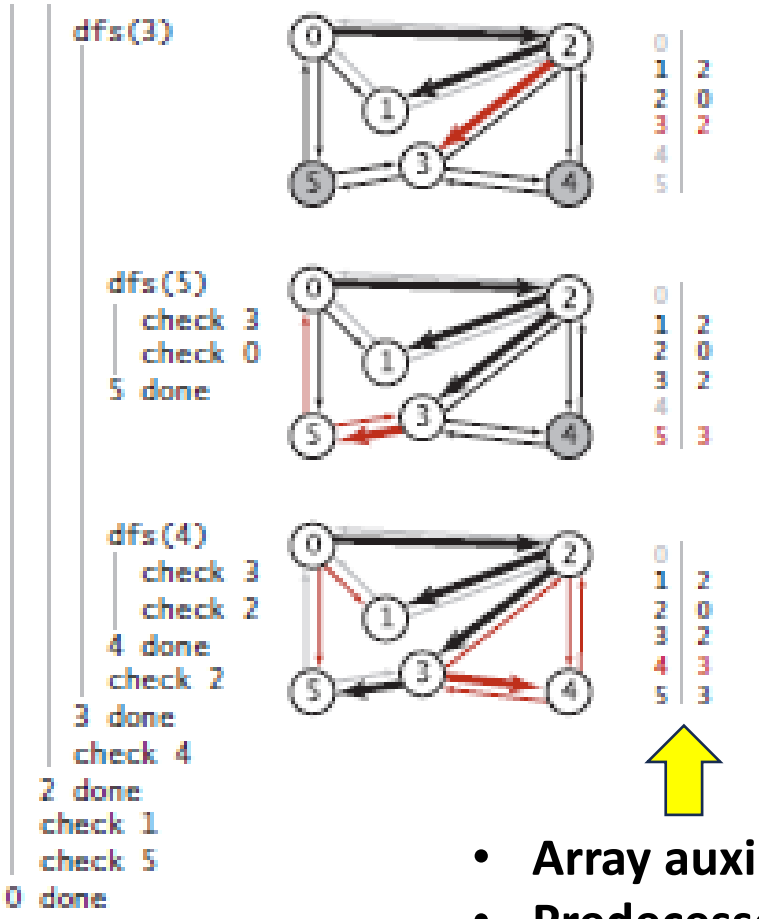
```
Travessia em Profundidade (vértice v)
        Criar um STACK vazio
        Push(stack, v)                          // Vértice inicial
        Marcar v como visitado
        Enquanto NãoVazio(stack) fazer
                v = Pop(stack)
                Para cada vértice w adjacente a v
                        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 ?
- Obtém-se uma árvore de caminhos com raiz no vértice inicial
- Como construir a árvore ?
- Fácil : registar, num array auxiliar, o predecessor de cada vértice no caminho a partir do vértice inicial, caso exista
- Fazer o "traceback" para obter a sequência de vértices definindo o caminho

Árvore dos caminhos com origem no vértice 0

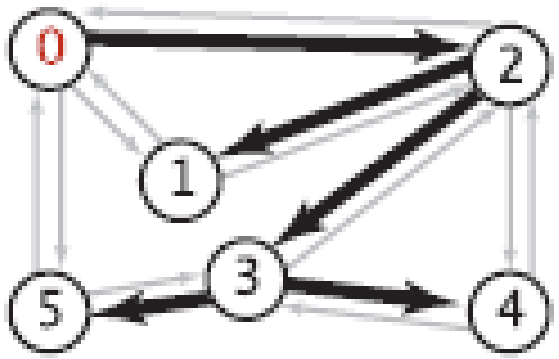




- **Array auxiliar**
- **Predecessores**

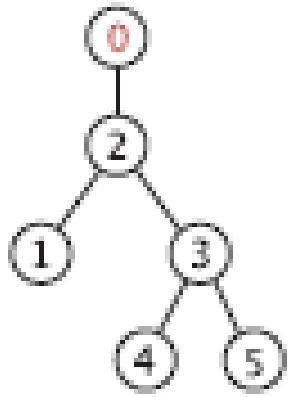
[Sedgewick/Wayne]

Árvore dos caminhos com origem no vértice 0



edgeTo[]

0	
1	2
2	0
3	2
4	3
5	3



x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

[Sedgewick/Wayne]

- Array auxiliar representa a árvore dos caminhos com origem no vértice inicial
- A cada vértice está associado o seu predecessor nesse caminho, caso exista

Travessia Recursiva em Profundidade – O módulo GraphDFSRec

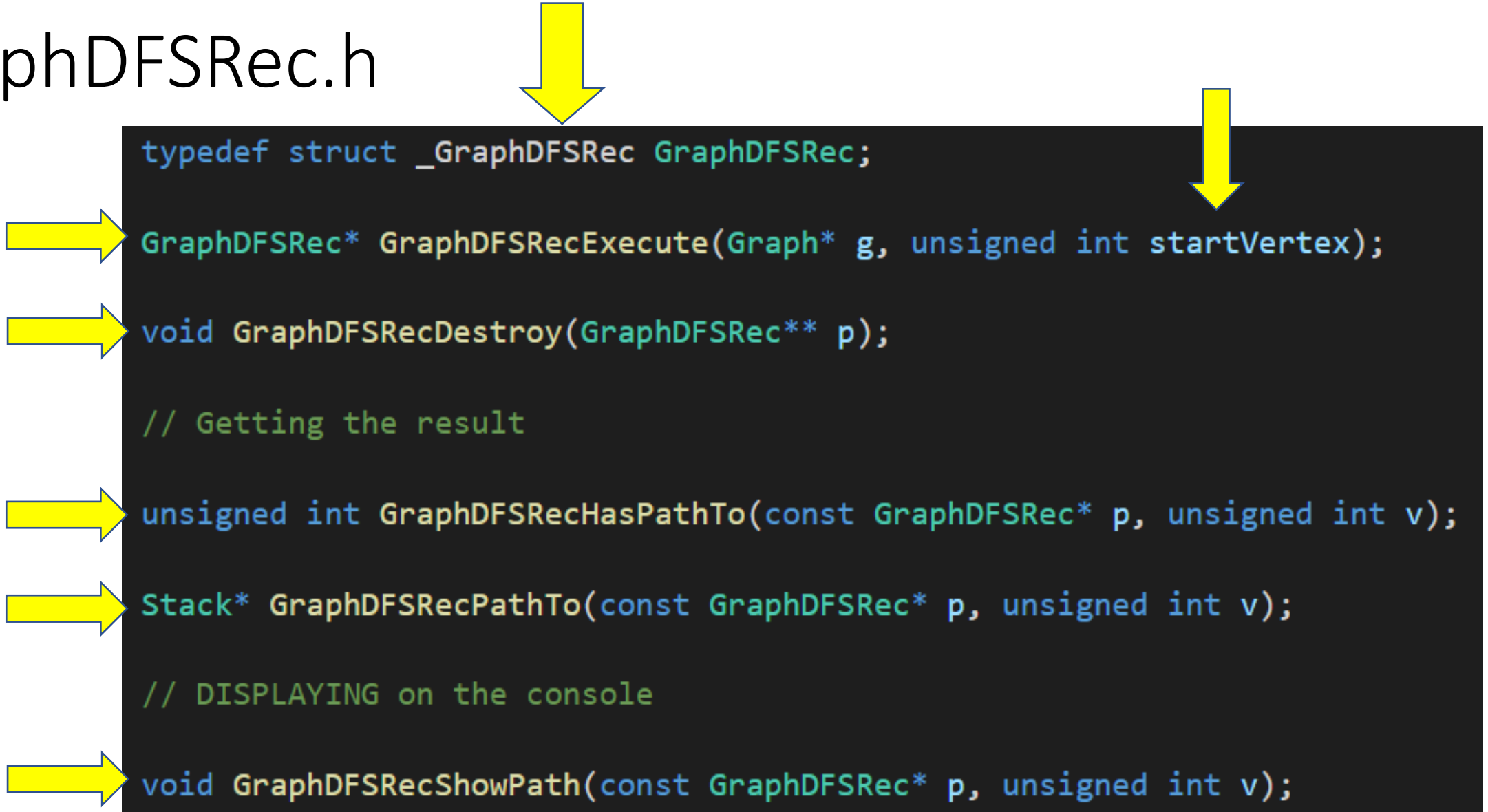
GraphDFSRec – Módulo adicional

- Permite executar a travessia recursiva em profundidade
- A partir de um vértice inicial

- Aloca memória para registar o resultado da travessia
- Array assinalando os vértices visitados, i.e., alcançados
- Array associando a cada vértice o seu predecessor no caminho encontrado, caso exista

- Programador é responsável por libertar a memória alocada

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);
```

Estrutura de dados + Função recursiva

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

```c
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);
}
```

Caminho do vértice inicial até ao vértice v

- Após a travessia do grafo ter sido efetuada
- Não há caminho se o vértice v não tiver sido visitado
- Obter o caminho desde o vértice inicial até ao vértice v, consultando os predecessores
- A sequência de vértices do caminho é devolvida numa pilha/stack

```
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;
}
```

Travessia por Níveis
– Breadth-First Traversal

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 um 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** !!

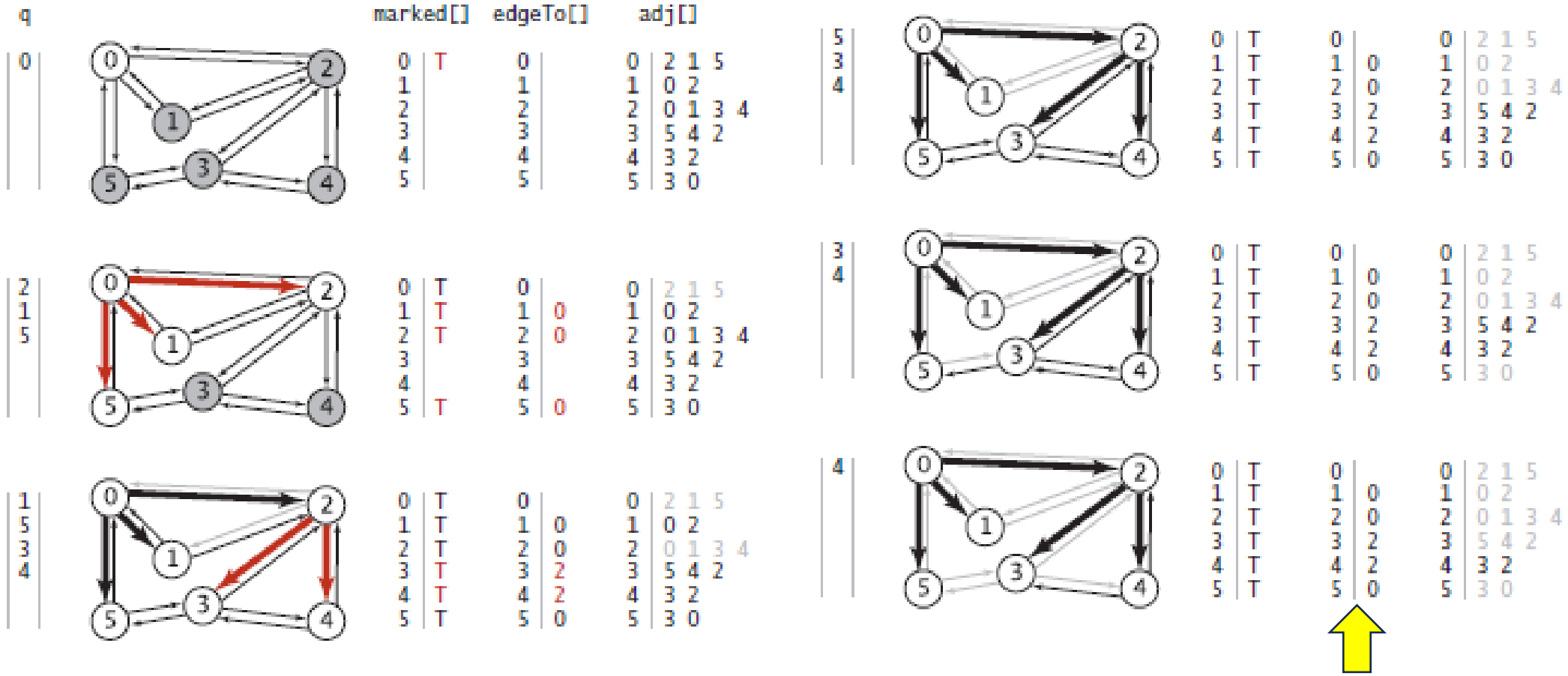
Breadth-First Traversal – Algoritmo iterativo

```
Travessia por Níveis(vértice v)
        Criar FILA vazia
        Enqueue(queue, v)                          // Vértice inicial
        Marcar v como visitado
        Enquanto NãoVazia(queue) fazer
                v = Dequeue(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
- Registar o predecessor de cada vértice no caminho mais curto 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]

- **Array auxiliar**
- **Predecessores**

Ordenação Topológica – Topological Sorting

Ordenação Topológica

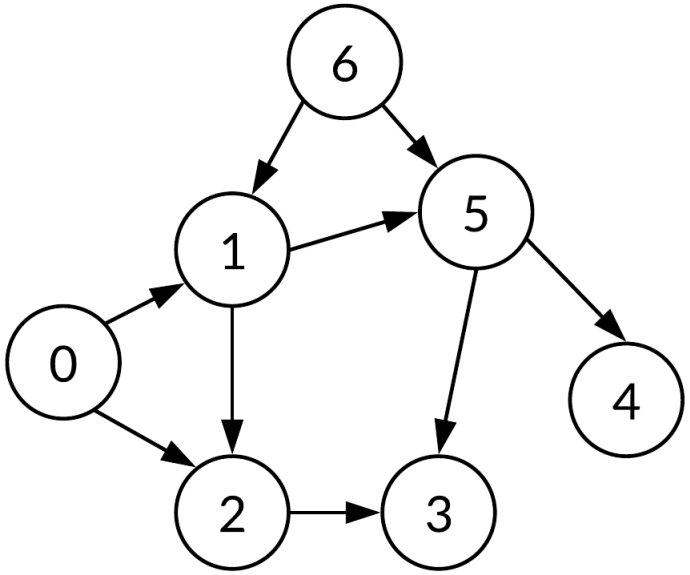
- Podemos desenhar um dado grafo orientado de maneira a que todas as arestas apontem para o mesmo lado ?

- Dado um conjunto de tarefas a realizar, e as respetivas precedências, qual é a ordem pela qual devem ser escalonadas/executadas ?
  - Usar BFS ou DFS !
  - Representar a solução com um grafo orientado acíclico !

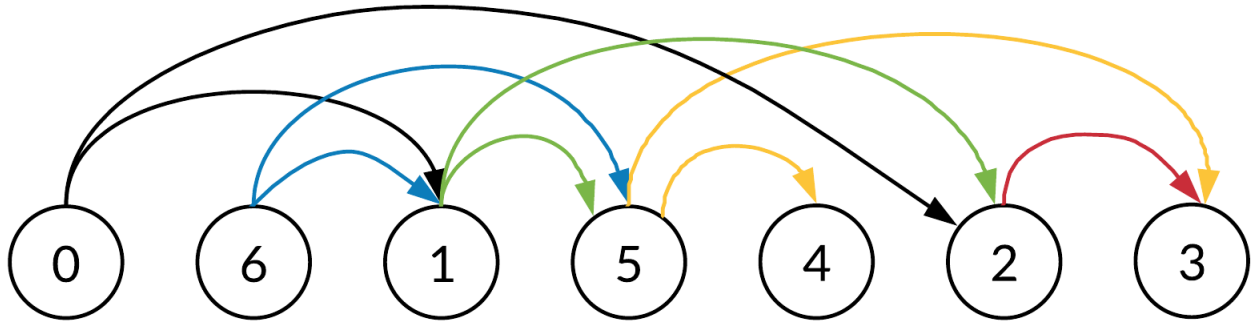
- Aplicação : verificar se um grafo orientado é acíclico ou não

Ordenação Topológica – Precedências

Unsorted graph



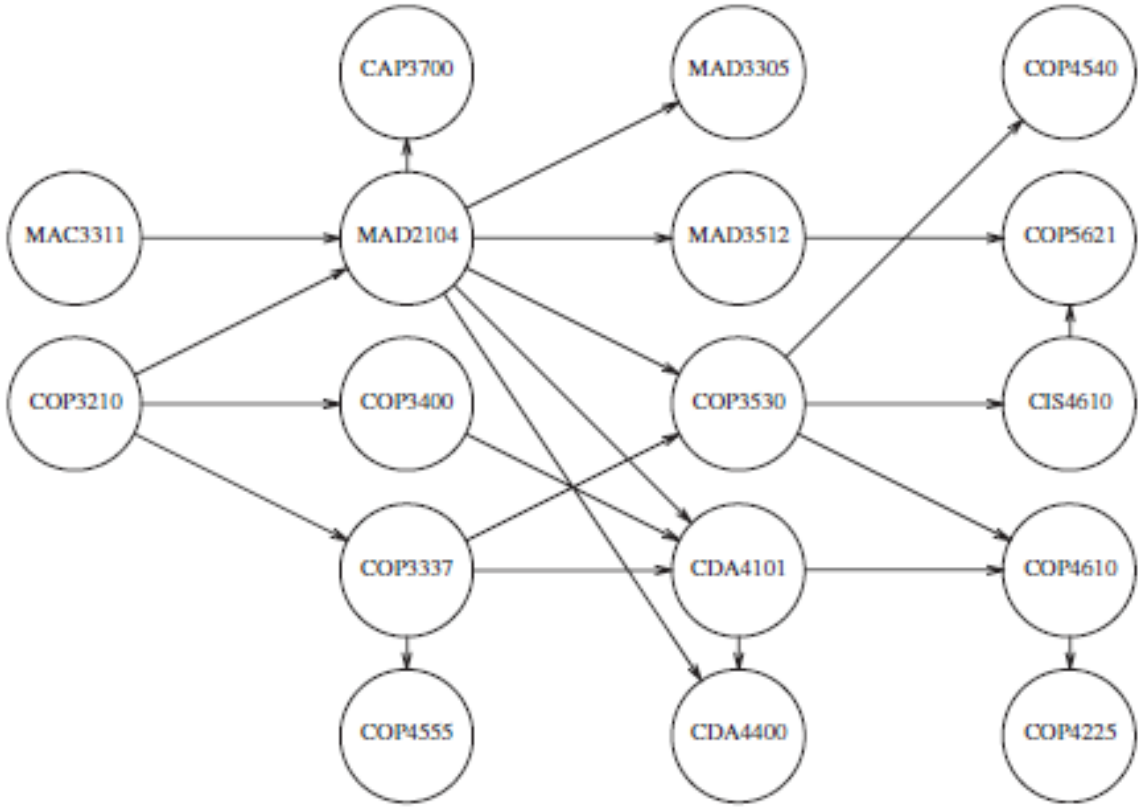
Topologically sorted graph



[guides.codepath.com]

- Há uma só solução ?
- Ou há várias soluções equivalentes ?

Grafo das precedências das UCs de um curso

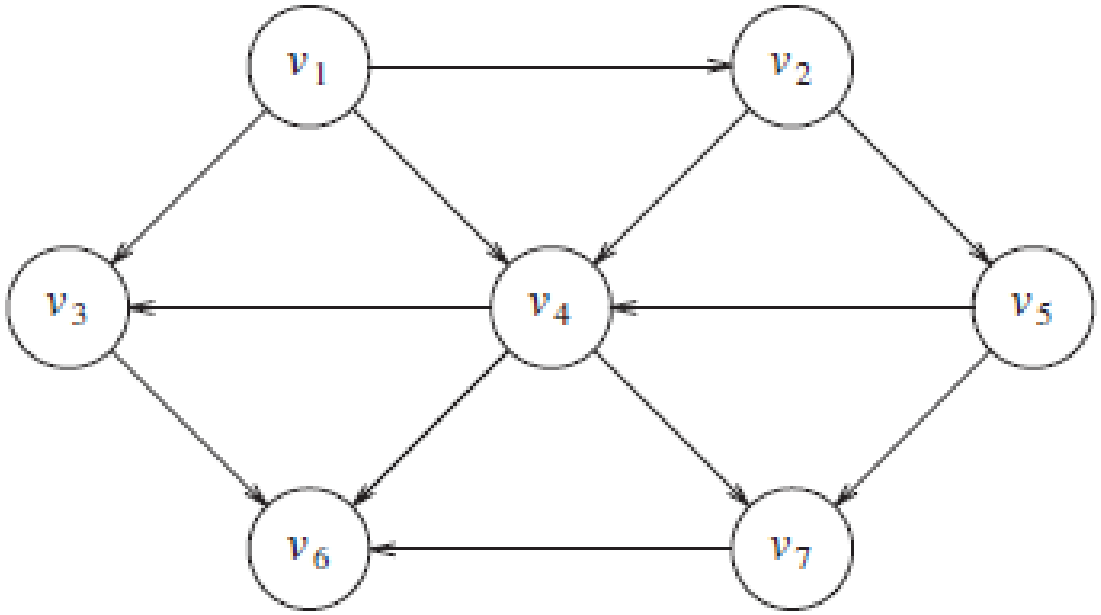


Ordenação Topológica

- Como ordenar as UCs de acordo com as precedências definidas ?
- Grafo orientado e acíclico !!

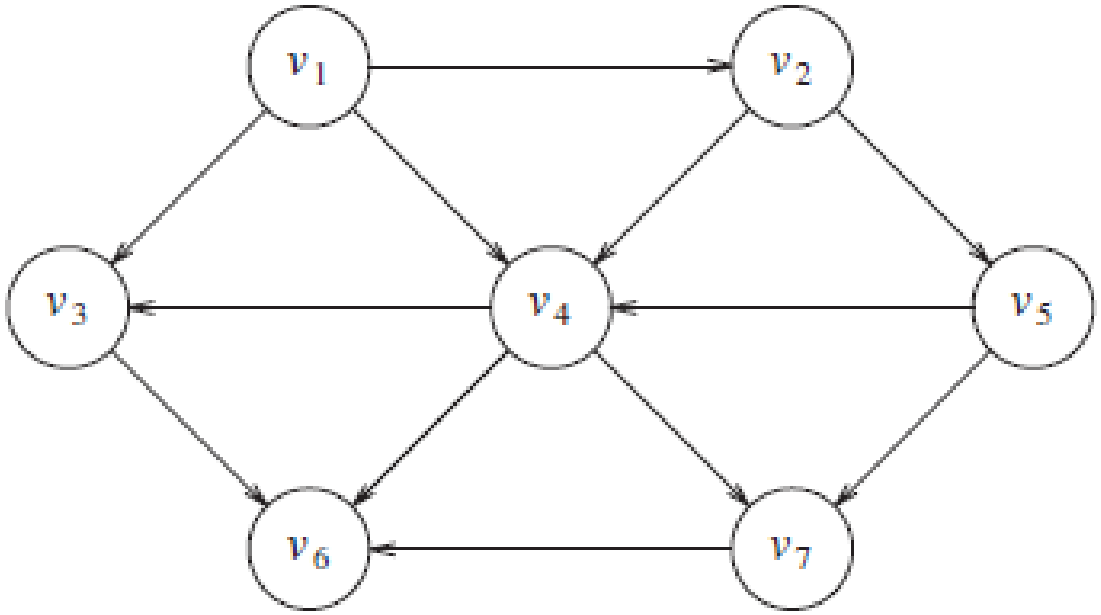
- Ordem ?
- Se existe um caminho de v para w, então w aparece após v na sequência de vértices ordenados
- Não podem existir ciclos !!
- Pode haver mais do que uma ordenação válida !!

Exemplo



- Possíveis sequências ordenadas de vértices ?

Exemplo



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

1º algoritmo – Cópia inicial 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

Registar 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

Registar 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

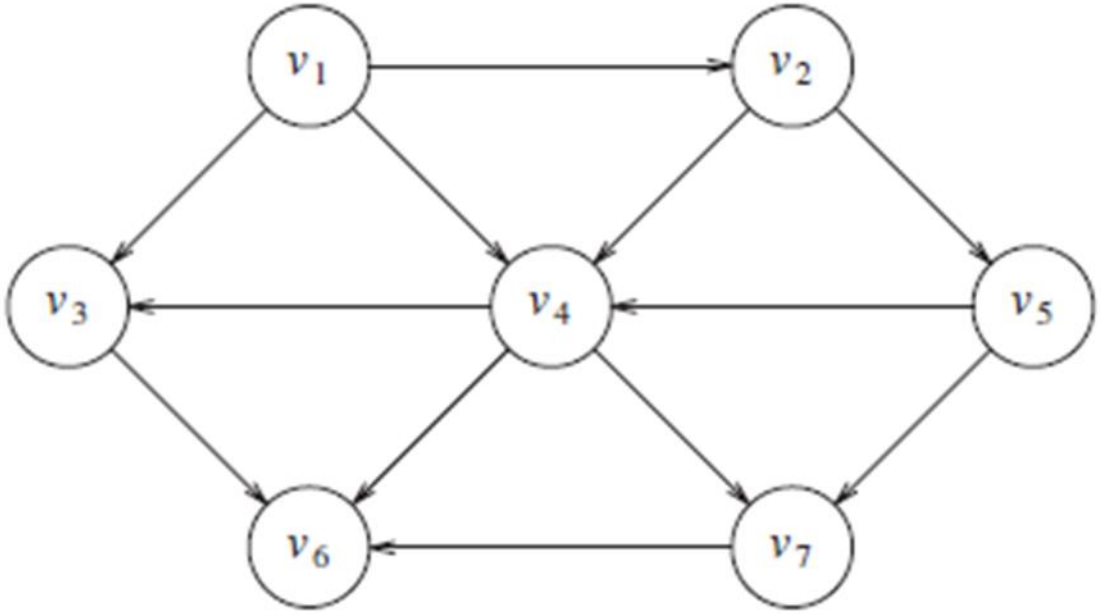
    Para cada vértice w adjacente a v

        numEdgesPerVertex[w] --

        Se numEdgesPerVertex[w] == 0 Então Inserir w na FILA

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

Exemplo



	Indegree Before Dequeue #						
Vertex	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

Exercícios / Tarefas

Tarefa – Travessia Iterativa em Profundidade

- Analisar o ficheiro GraphDFSRec.c

- **DESENVOLVER UM 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 ?

Tarefa – Travessia Iterativa por Níveis

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

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