Grafos II

Joaquim Madeira

26/05/2020

Ficheiro ZIP

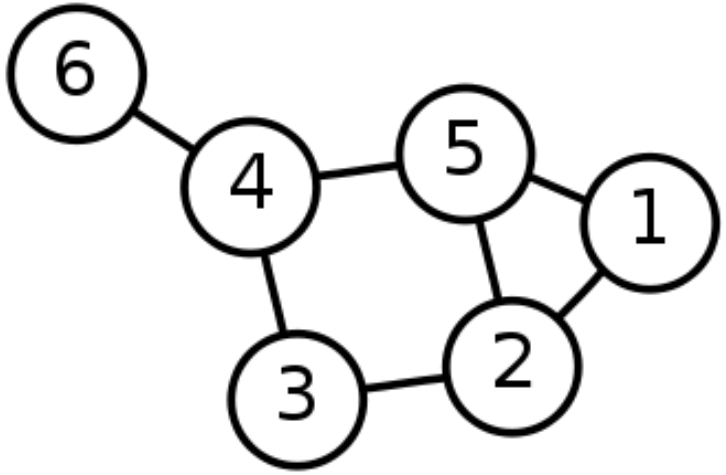
- Está disponível no Moodle um ficheiro ZIP de suporte aos tópicos de hoje
- O tipo abstrato Grafo usando o TAD SortedList
- Versão "simples", que permite trabalho autónomo de desenvolvimento e teste
- Um módulo implementando a Travessia em Profundidade

Sumário

- Recap
- O TAD Grafo
- Travessia em Profundidade ("Depth-First")
- Travessia por Níveis ("Breadth-First")
- Ordenação Topológica
- Sugestão de leitura

Recapitulação

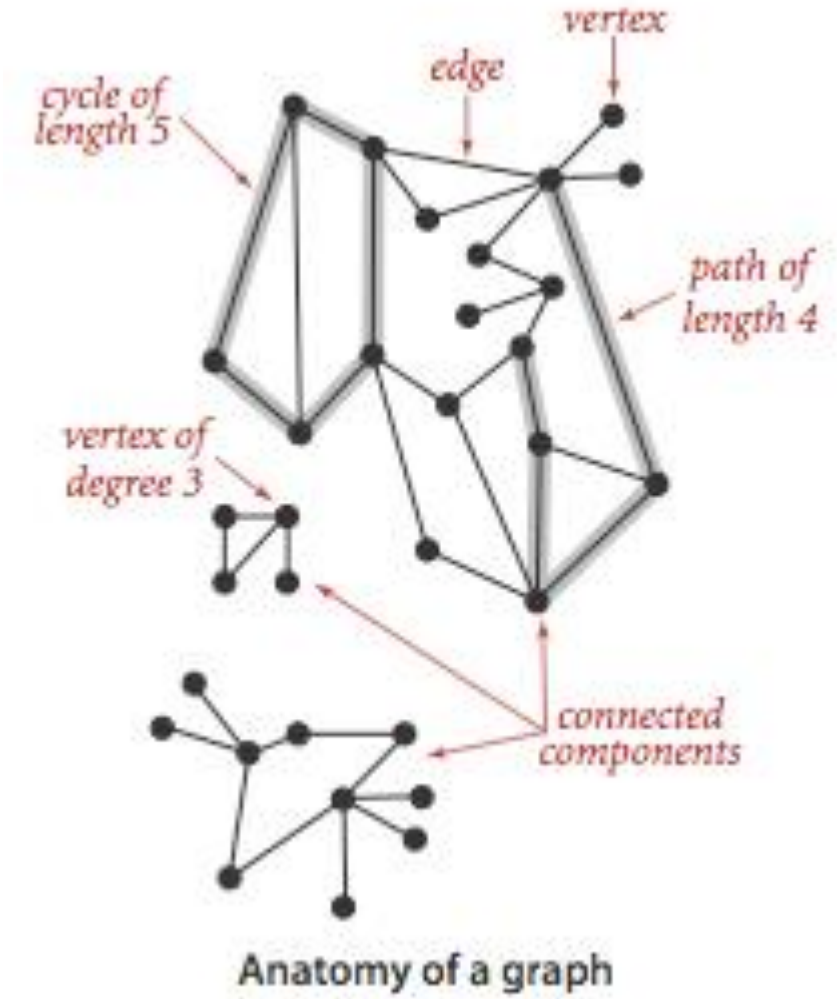
Grafo



[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

Grafo



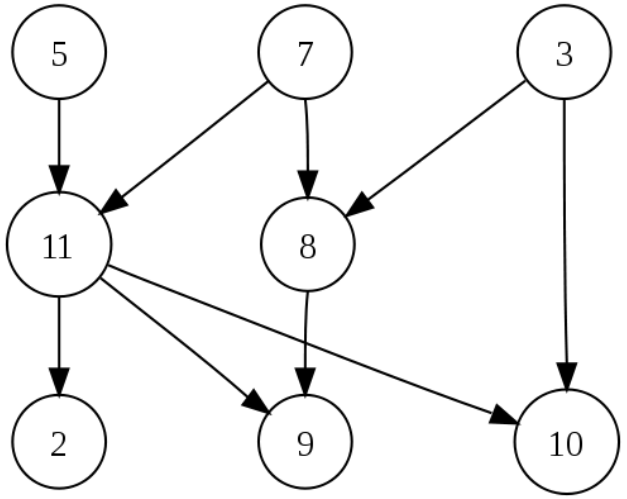
[Sedgewick & Wayne]

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]

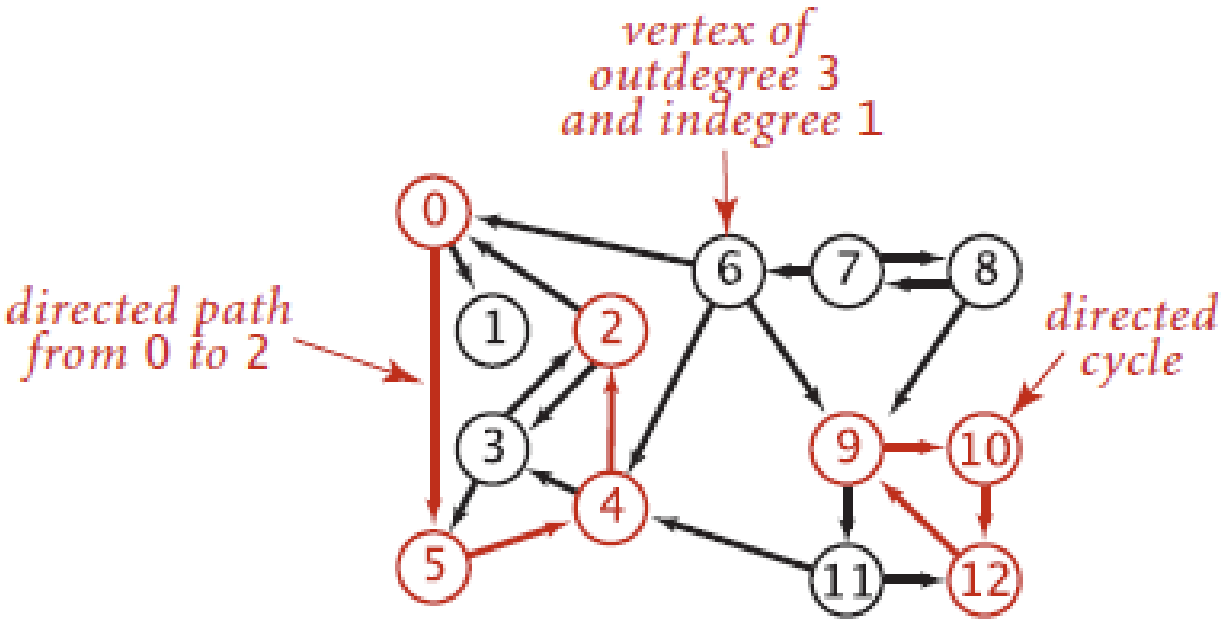
Grafo orientado



[Wikipedia]

- 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

Grafo orientado



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]

Ordenação Topológica

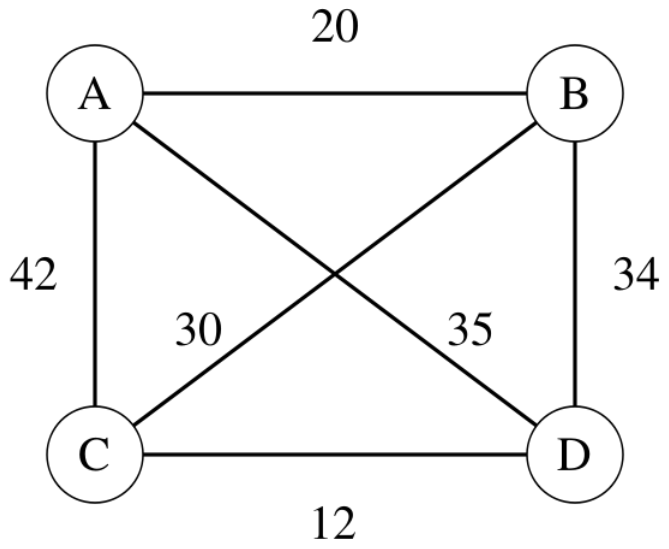
- Podemos desenhar um dado grafo orientado de maneira a que todas as arestas apontem para o mesmo lado ?

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

- Usar para verificar se um grafo orientado é acíclico ou não

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, …

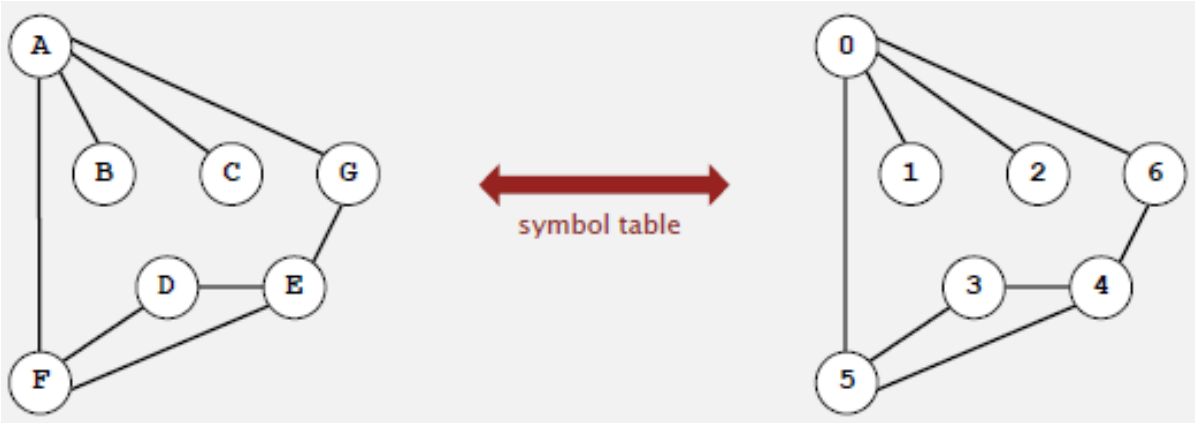


[Wikipedia]

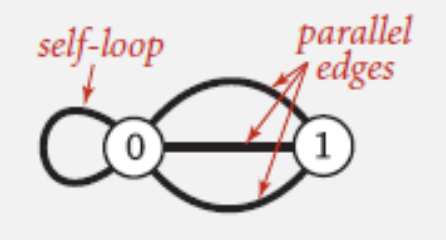
O TAD Grafo

Vértices

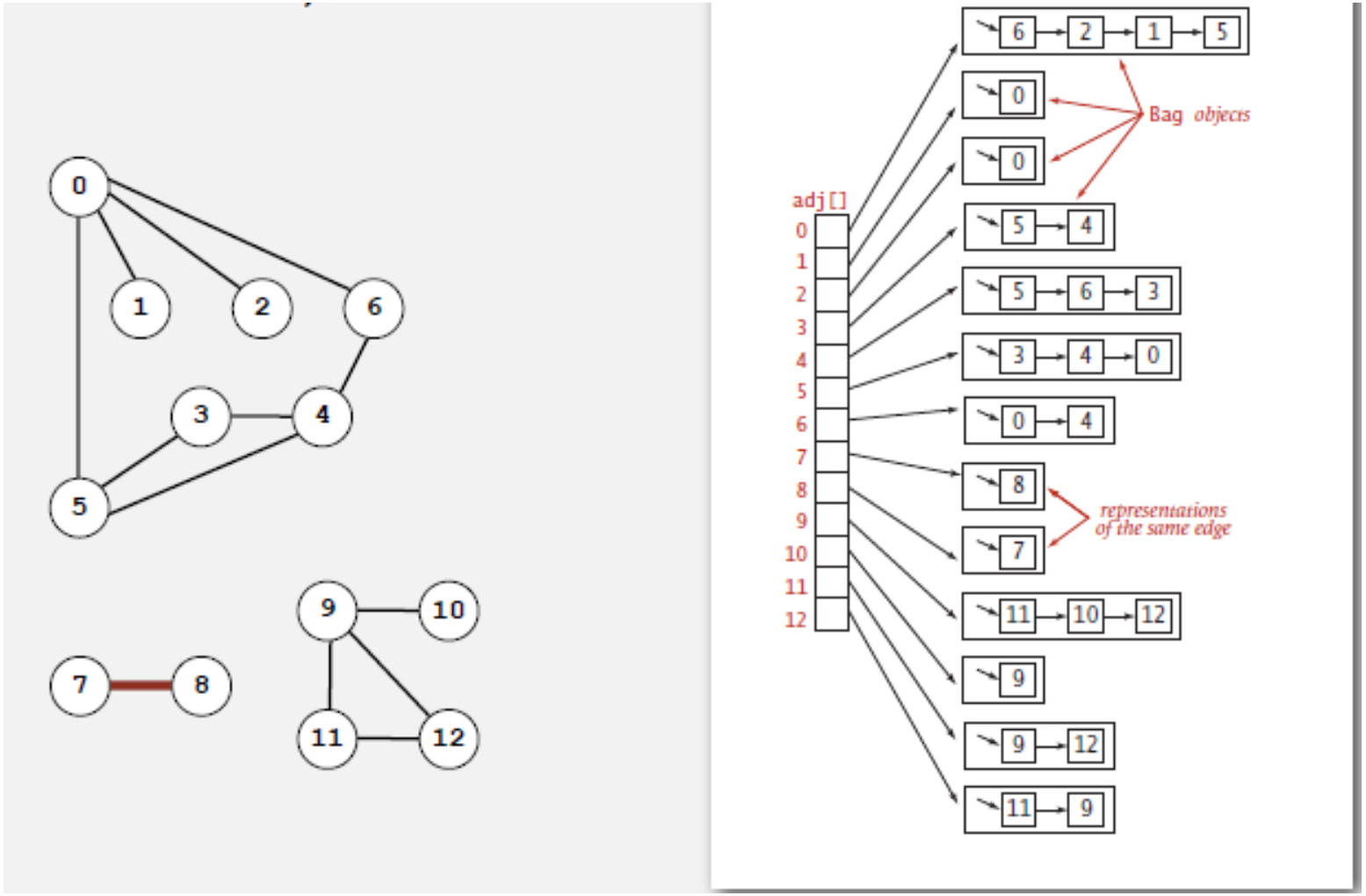
- Identificados por um valor inteiro de 0 a V – 1
- Usar dicionários para mapear esses IDs noutros identificadores



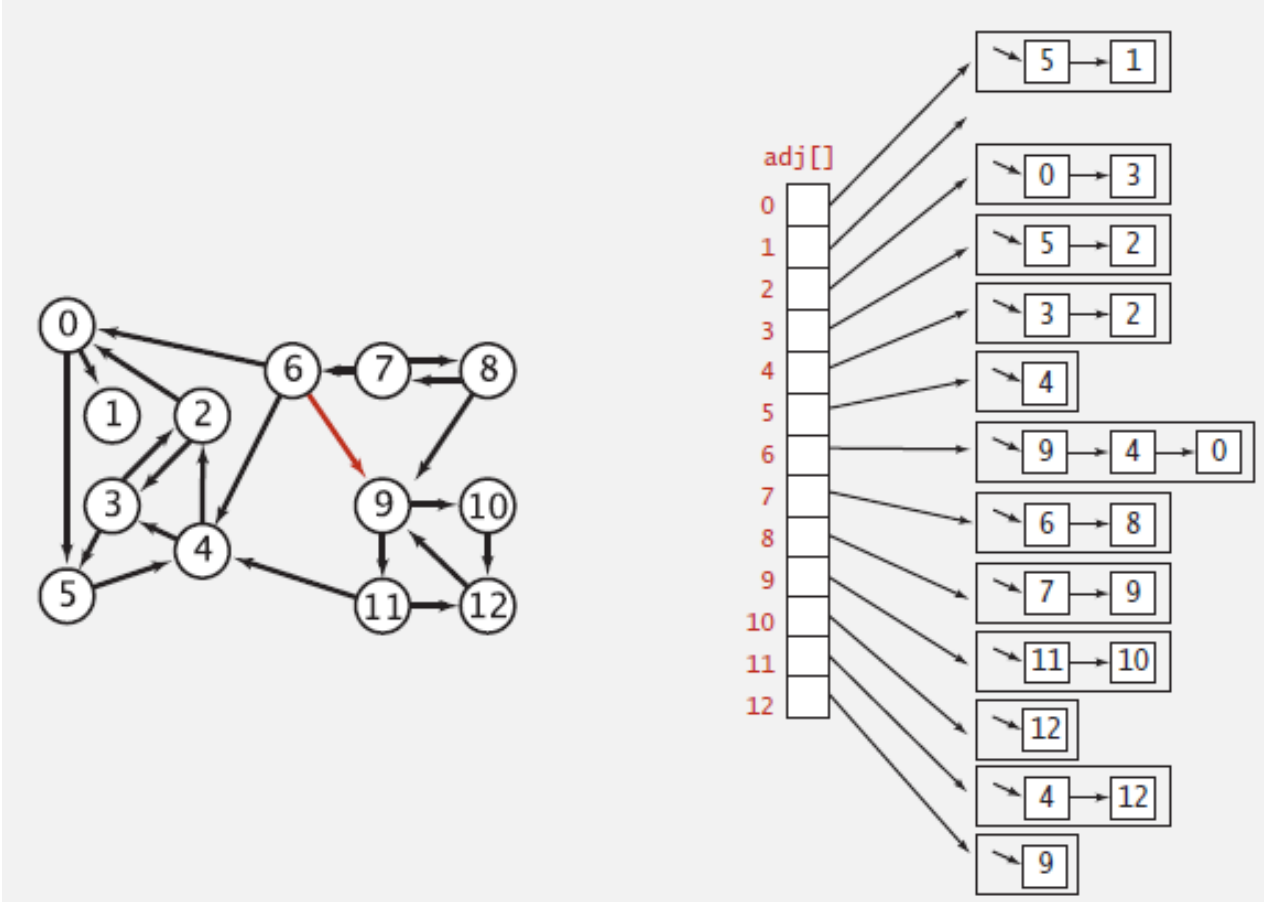
- Não são permitidos lacetes nem arestas paralelas



Representação – Listas de adjacências



Representação – Listas de adjacências



Decisões

- 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

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

```
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);
```

Graph.h

```
unsigned int* GraphGetAdjacentsTo(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);

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

Graph.h

```
// Edges

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

// CHECKING

unsigned short GraphCheckInvariants(const Graph* g);

// DISPLAYING on the console

void GraphDisplay(const Graph* g);

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

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.c – Questões de implementação

- Como atravessar a lista de vértices ?
- Como atravessar uma lista de adjacências ?
- Como adicionar uma aresta ?
- Usar o iterador do TAD Sorted List !!

- Como devolver os índices dos vértices adjacentes ?

- …

Graph.c

- Vamos analisar algumas das funções desenvolvidas

- TAREFA : completar o que falta !!

Travessia em Profundidade

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

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
- Encontrar um caminho entre o vértice inicial e cada um dos outros vértices alcançáveis
- …

Algoritmo recursivo

Travessia em Profundidade (vértice v)

Marcar v como visitado

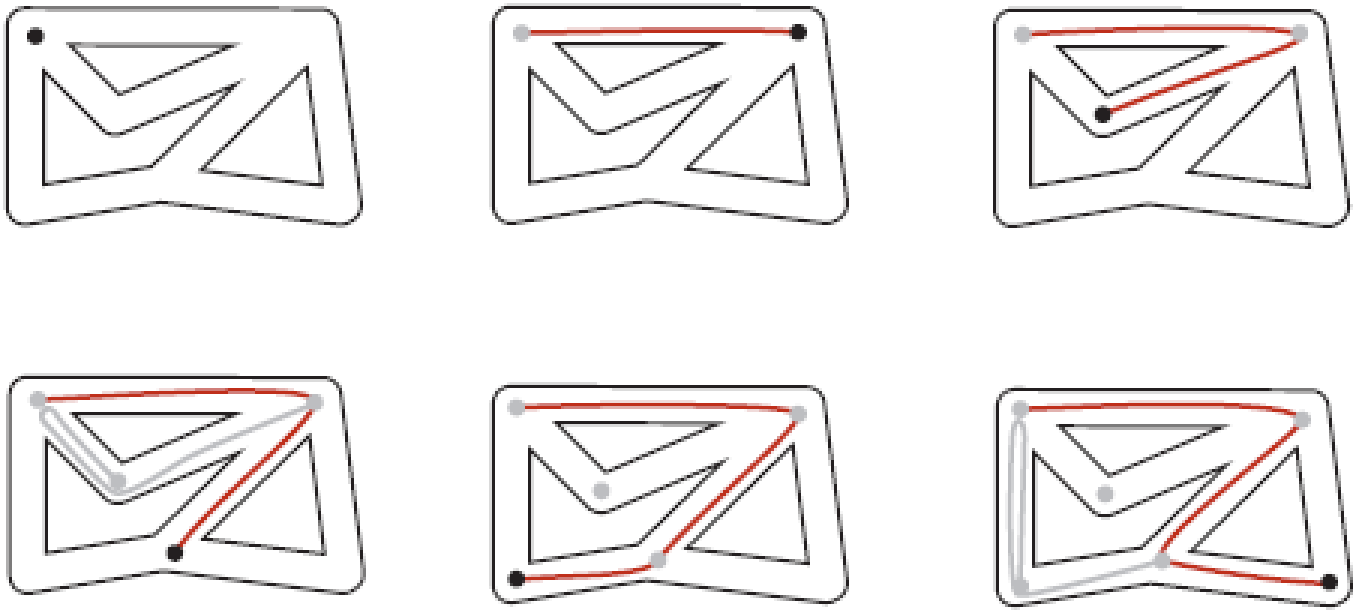
Para cada vértice w adjacente a v

Se w não está marcado como visitado

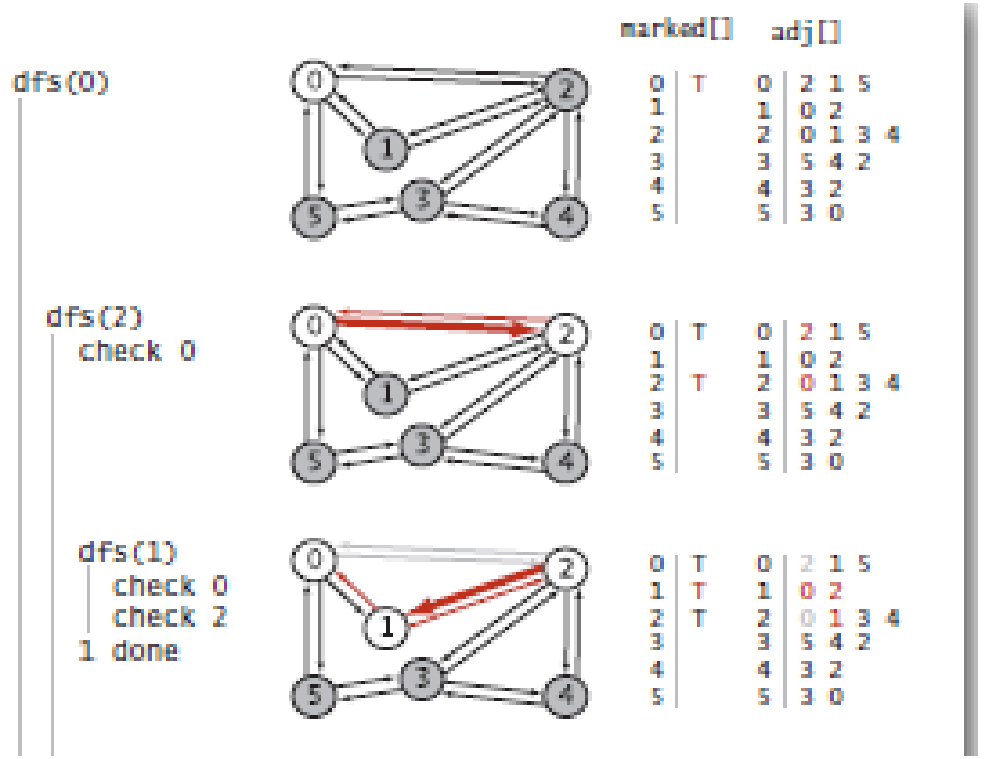
Então efetuar a Travesssia em Profundidade (w)

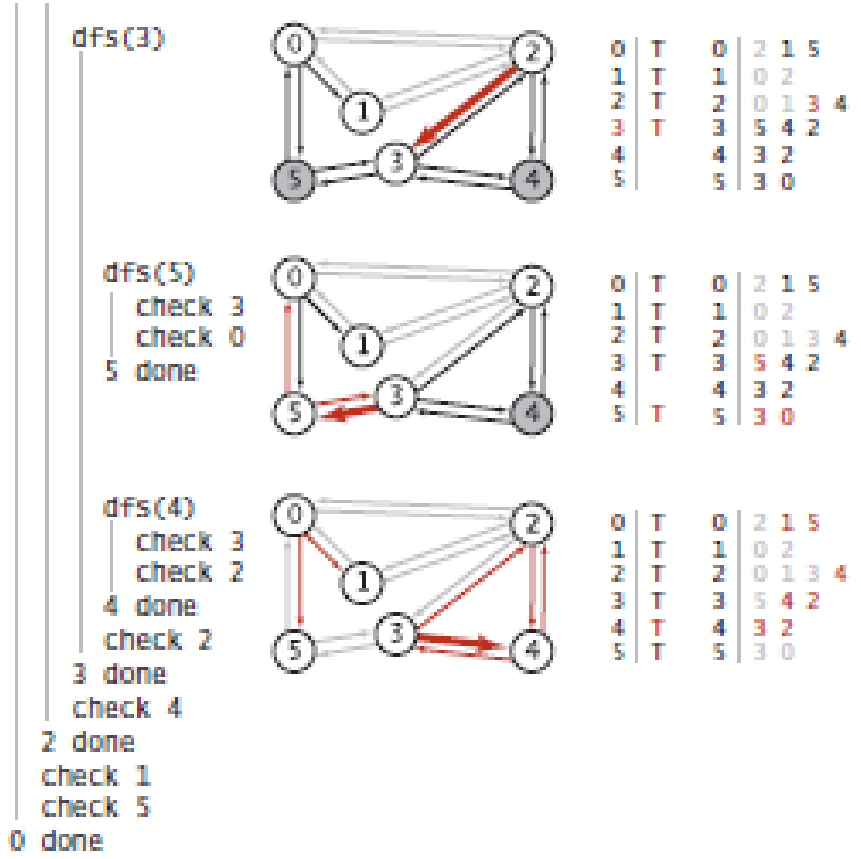
- Resultado ?
- Ficam marcados todos os vértices alcançáveis

Exploração de um labirinto



Exemplo


marked[]
adj[]
dfs(0)
dfs(2)
check 0
dfs(1)
check 0
check 2
1 done


dfs(3)
dfs(5)
check 3
check 0
5 done
dfs(4)
check 3
check 2
4 done
check 2
3 done
check 4
2 done
check 1
check 5
0 done

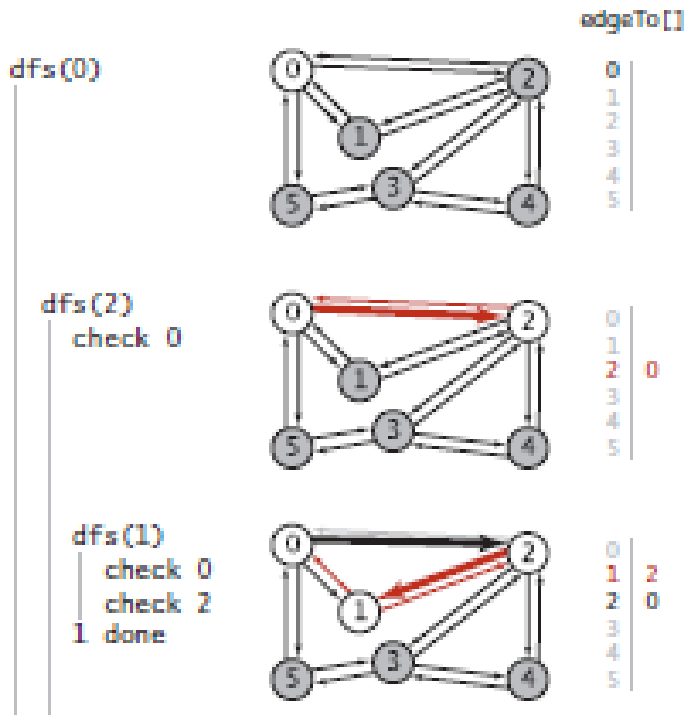
Algoritmo iterativo – A mesma ordem ?

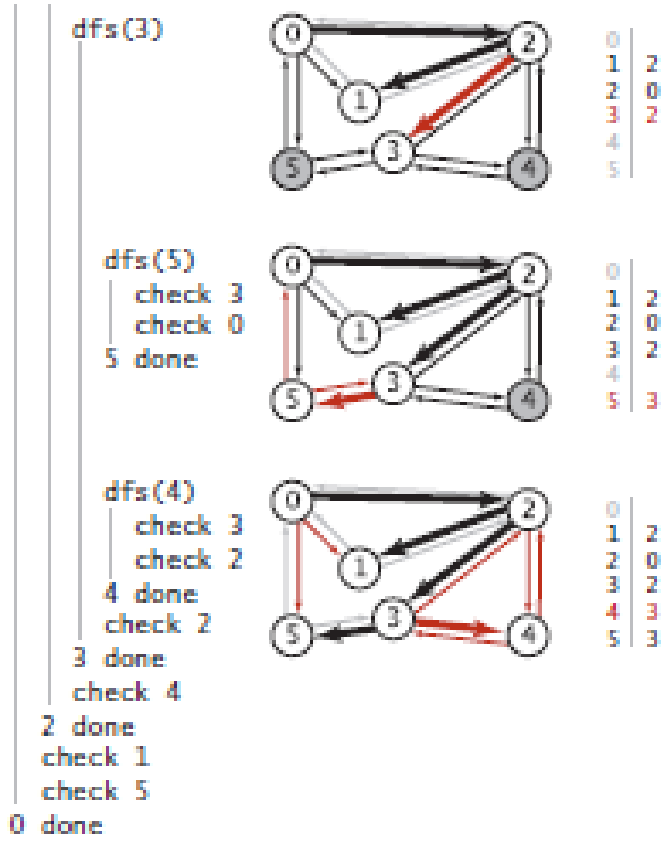
```
Travessia em Profundidade (vértice v)
        Criar um STACK vazio
        Push(stack, v)
        Marcar v como visitado
        Enquanto NãoVazio(stack) fazer
                v = Pop(stack)
                Para cada vértice w adjacente a v
                        Se w não está marcado como visitado
                        Então Push(stack, w)
                                Marcar w como visitado
```

Vértices alcançáveis

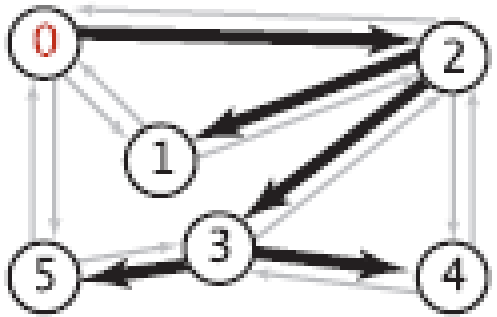
- Determinar o conjunto dos vértices alcançáveis significa encontrar um caminho entre o vértice inicial e cada um dos vértices alcançados
  - Pode não ser o caminho mais curto !!
  - Porquê ?
- Árvore de caminhos com raiz no vértice inicial
- Como registar a árvore ?
- Fácil : registar o predecessor 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



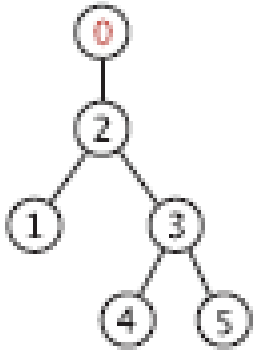


Árvore dos caminhos com origem em 0



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

GraphDFSRec.h

```
GraphDFS* GraphDFSExecute(Graph* g, unsigned int startVertex);

void GraphDFSDestroy(GraphDFS** p);

// Getting the result

unsigned int GraphDFSHasPathTo(const GraphDFS* p, unsigned int v);

Stack* GraphDFSPathTo(const GraphDFS* p, unsigned int v);

// DISPLAYING on the console

void GraphDFSShowPath(const GraphDFS* p, unsigned int v);

void GraphDFSDisplay(const GraphDFS* p);
```

GraphDFSRec.c

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

```c
static void _dfs(GraphDFS* 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);
    }
  }
}
```

GraphDSFRec.c

```c
Stack* GraphDFSPathTo(const GraphDFS* 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;
}
```

Análise + Tarefa

- Analisar o ficheiro GraphDFSRec.c

- TAREFA :
- 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 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 !!

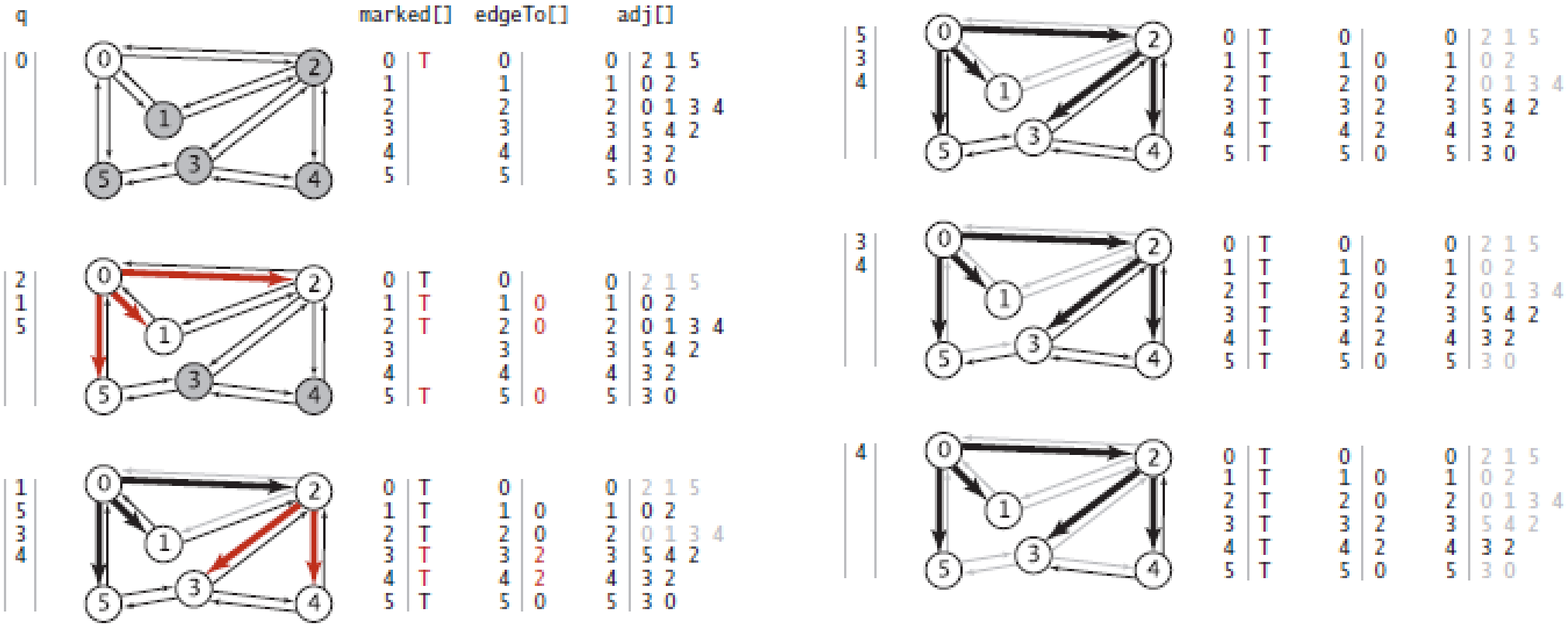
Algoritmo iterativo

```
Travessia por Níveis(vértice v)
        Criar FILA vazia
        Enqueue(queue, v)
        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 mais curto 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 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

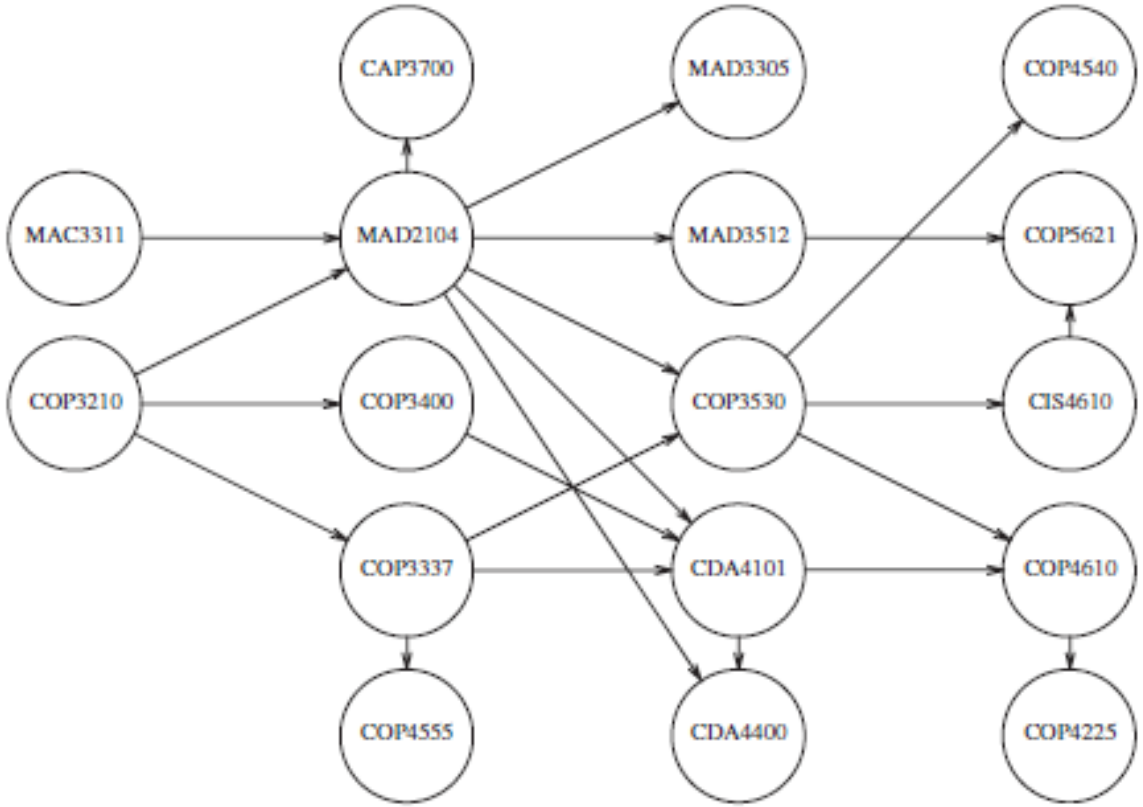


Tarefa

- TAREFA :
- Implementar e testar a travessia por níveis usando uma FILA/QUEUE

Ordenação Topológica

Grafo das precedências das UCs de um curso

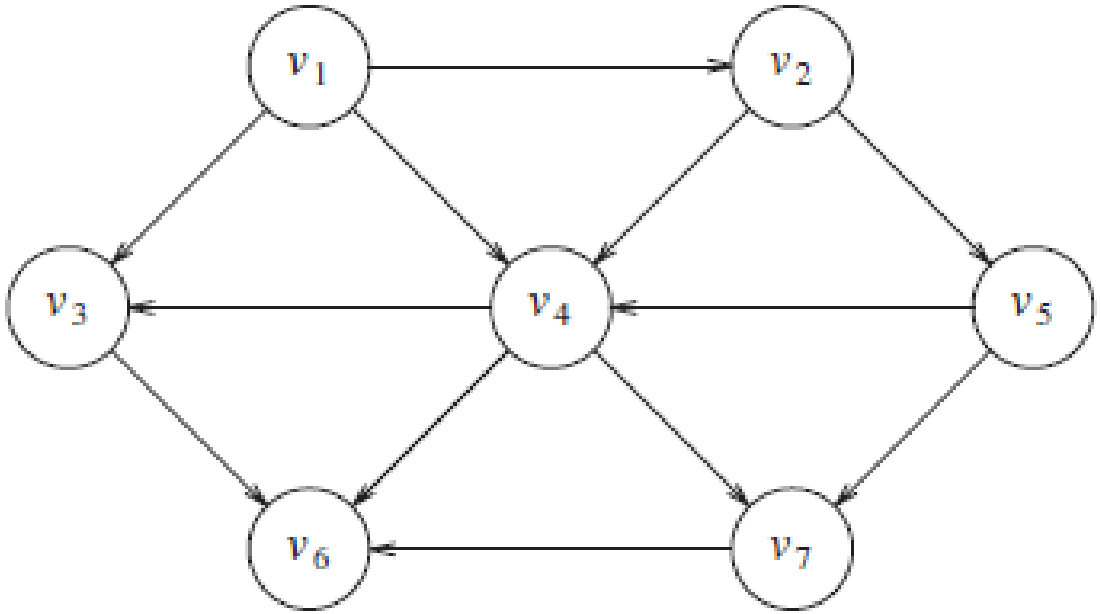


Ordenação Topológica

- 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 ?
- v1, v2, v5, v4, v3, v7, v6 OU v1, v2, v5, v4, v7, v3, v6
- Como proceder ?

1º algoritmo

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

Registar num array auxiliar numEdges o InDegree de cada vértice

Enquanto for possível

    Selecionar um vértice v com numEdges[v] == 0 E não marcado

    Imprimir o seu ID

    Marcá-lo como pertencendo à ordenação

    Para cada vértice w adjacente a v

        numEdges[w]--

    Apagar esse vértice de G´ e as arestas que dele emergem

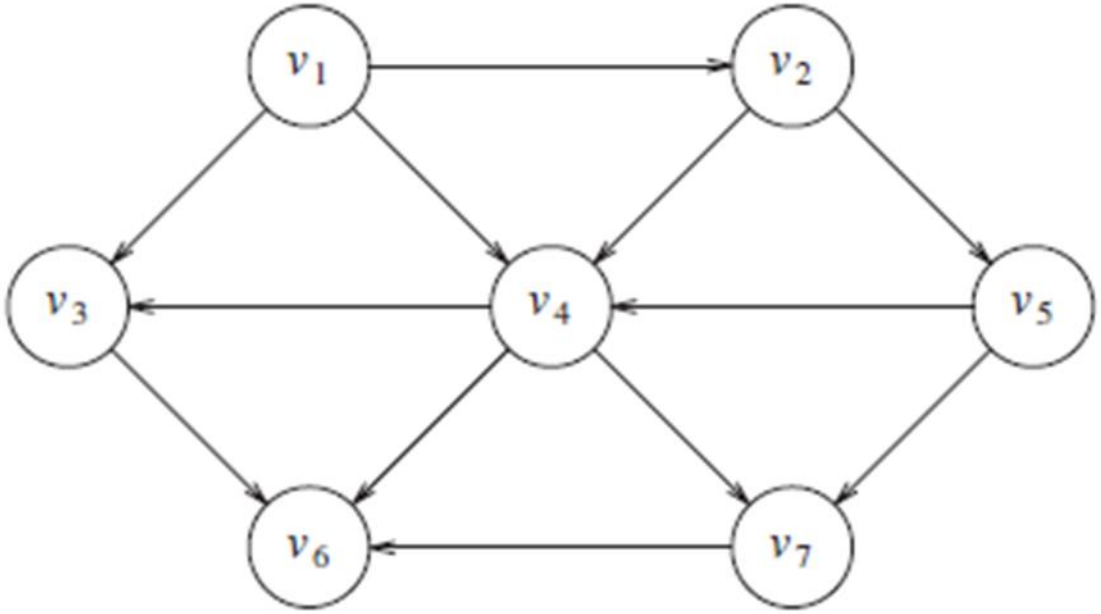
- Ineficiência : sucessivas procuras através do conjunto de vértices

3º alg. – Manter o conjunto de candidatos

Registar num array auxiliar numEdges o InDegree de cada vértice
Criar uma FILA vazia e inserir na FILA todos os vértices v com numEdges[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
        numEdges[w] --
        Se numEdges[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

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