Programação 2 _ T09

Grafos

Rui Camacho
(slides por Luís Teixeira)
MIEEC 2020/2021

INTRODUÇÃO

Ligação entre pares de elementos ocorre em muitas aplicações

Relações associadas às ligações levantam várias questões:

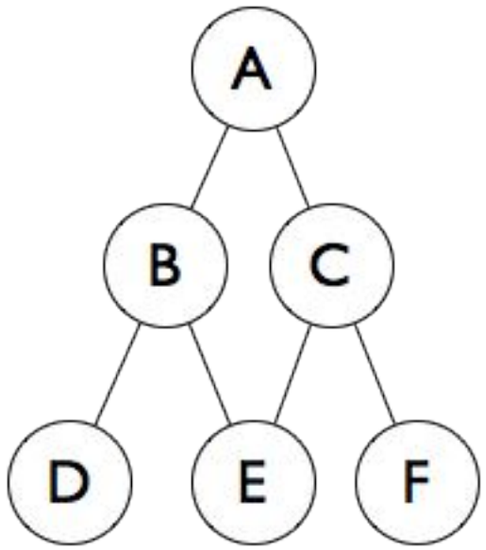
- Existe uma ligação (indireta) entre dois elementos?
- Qual o número de ligações de um elemento?
- Qual é o "caminho" mais curto entre dois elementos (se existir)?

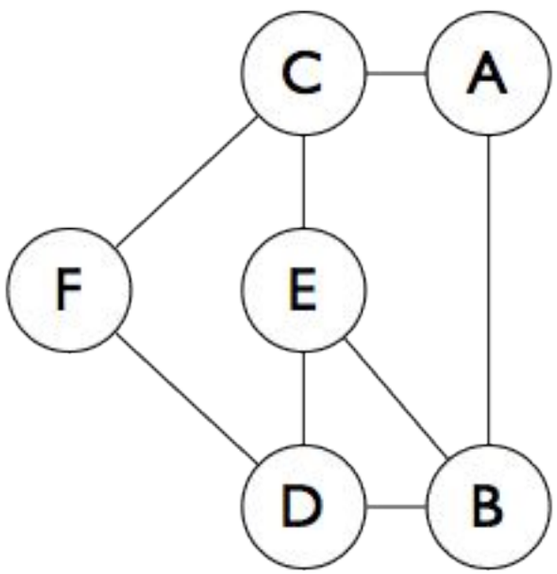
aplicação	elemento	ligação
mapa	povoação	estrada
web	página	link
circuito	componente	fio elétrico
rede social	pessoa	"amizade"

INTRODUÇÃO

Grafos são uma generalização de Árvores:

- ... existem Nós (ou Vértices),
- ... existem Arestas (ou Arcos),
- ... mas um nó pode ter mais que um "pai".

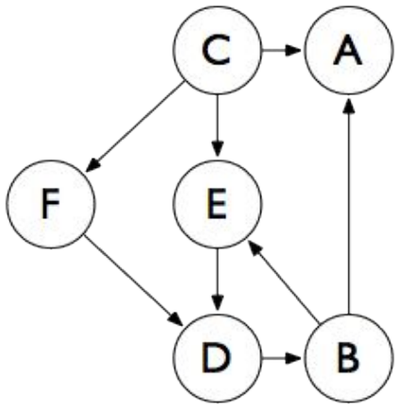


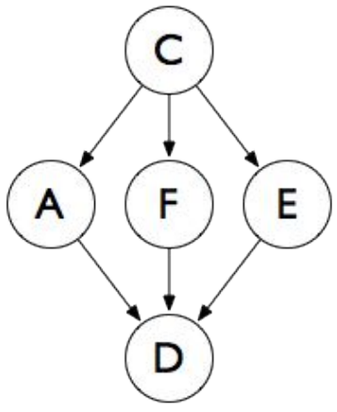


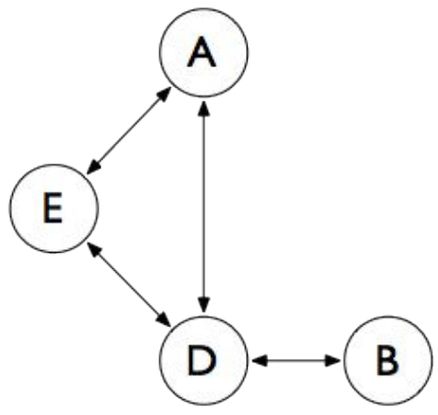
- Não há a noção de "raiz de um grafo" ...
  - ... nem de folha.

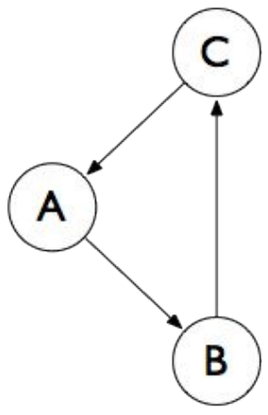
INTRODUÇÃO

- Os grafos podem ser:
  - ... dirigidos (direccionado, orientado, digrafo)

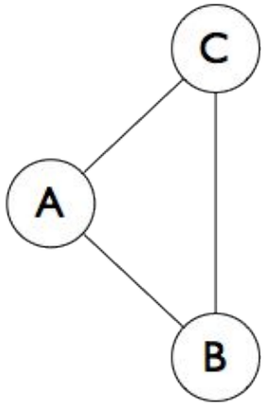


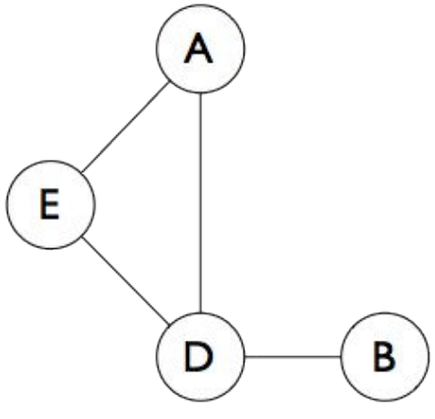


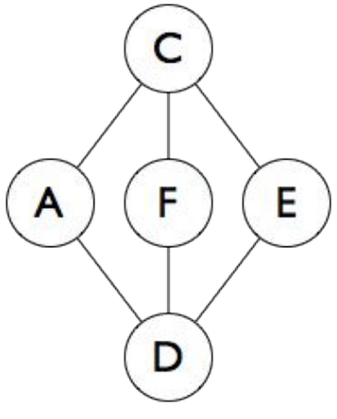


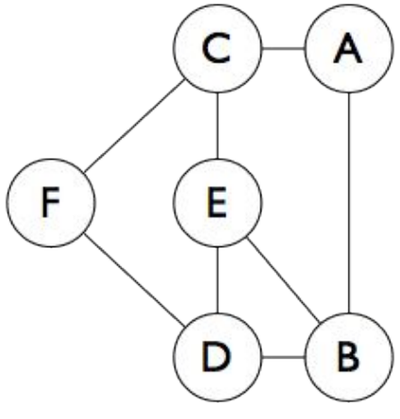


  - ... não dirigidos (não direcionado)



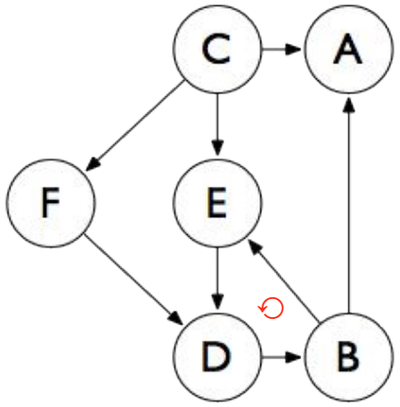


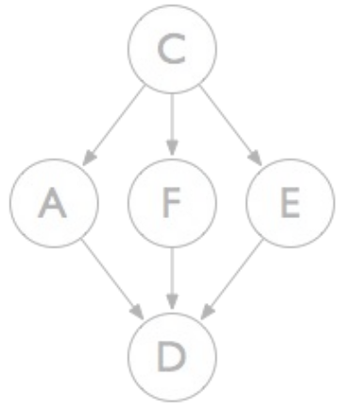


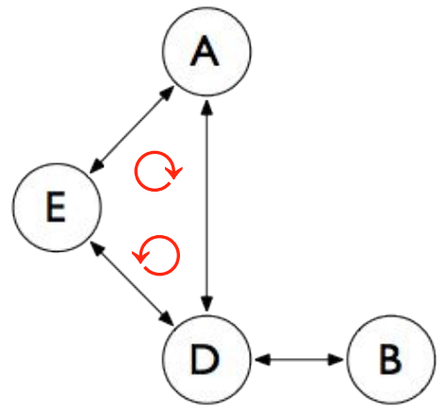


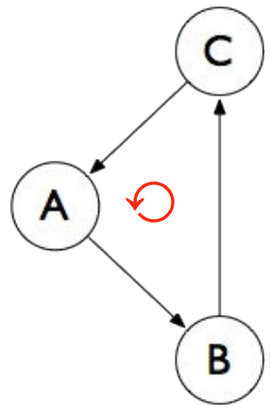
INTRODUÇÃO

- Os grafos dirigidos podem ser...
  - ... cíclicos (permitem ciclos):

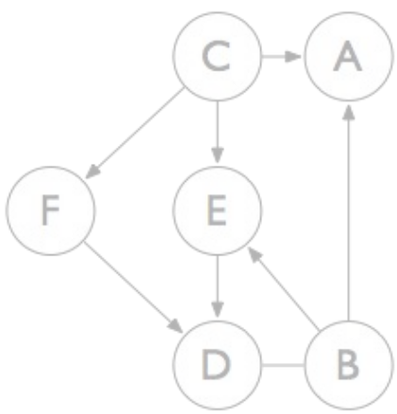


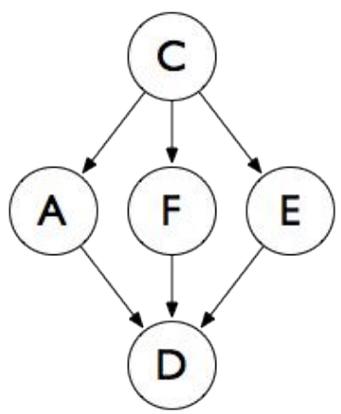


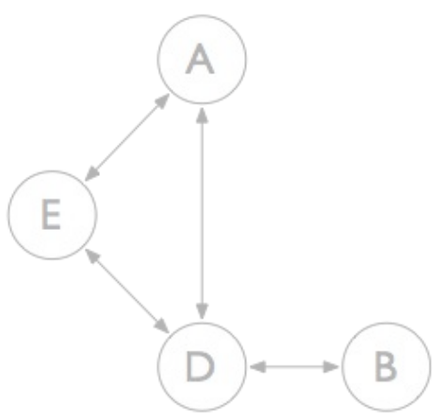


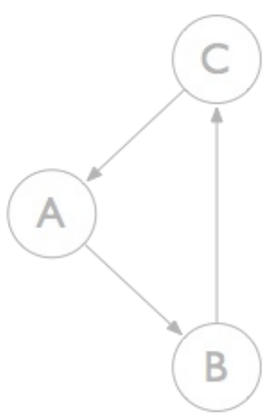


  - ... acíclicos (não permitem ciclos):



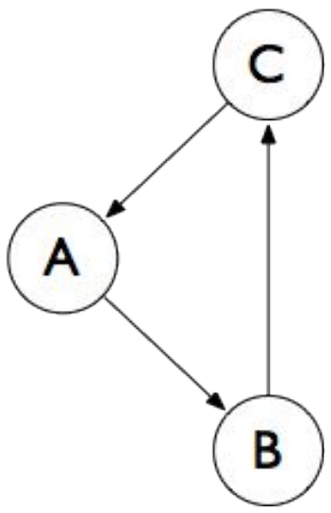






DEFINIÇÃO FORMAL

- Um grafo G =⟨V, E⟩consiste:
  - ... num conjunto finito de vértices V (*vertex, vertices*)
  - ... num conjunto finito de arestas E (*edge, edges*)
  - ... cada aresta é um par (a, b) tal que a, b ∈ E

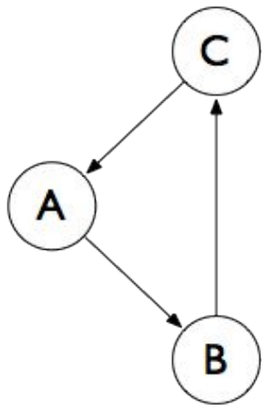


V = {A, B, C}

E = {(A, B), (B, C), (C, A)}

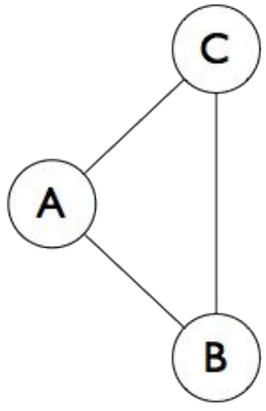
DEFINIÇÃO FORMAL

- Num digrafo, as arestas são pares ordenados:



$(A, B) \neq (B, A)$ $(B, C) \neq (C, B)$

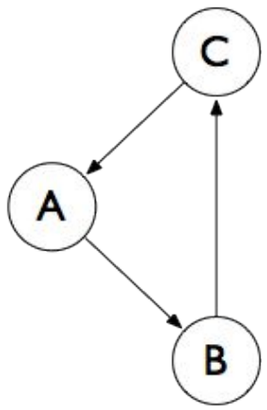
- Num grafo não-dirigido, a ordem é irrelevante:



$(A, B) = (B, A)$ $(B, C) = (C, B)$

TERMINOLOGIA

- Considere o seguinte digrafo:



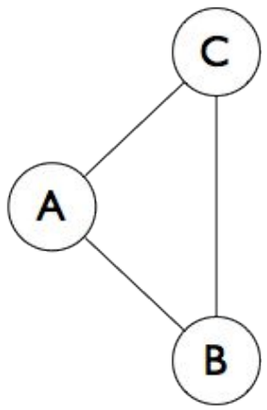
$V = \{A, B, C\}$

$E = \{(A, B), (B, C), (C, A)\}$

- C é adjacente a B, porque $(B, C) \in E$
- A não é adjacente a B já que $(B, A) \notin E$
- A é antecessor de B , B é sucessor de A
- A origem da aresta (C, A) é C, e o destino é A

TERMINOLOGIA

- Considere o seguinte grafo não dirigido:



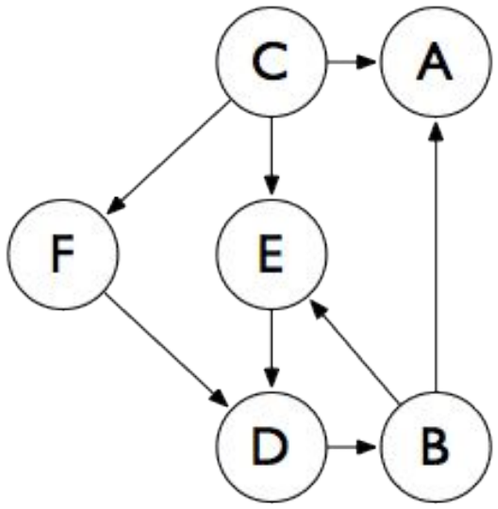
$V = \{A, B, C\}$

$E = \{(A, B), (B, C)\}$

- B é adjacente a A, porque $(A, B) \in E$
- A também é adjacente a B já que $(B, A) = (A, B) \in E$
- Diz-se que A e B são vizinhos.

TERMINOLOGIA

- Um caminho C é definido por uma sequência de vértices $\{w_1, w_2, ...\}$ onde $\forall_{1 \leq i \leq n} : (w_i, w_{i+1}) \in E$.
- ... por outras palavras, para cada dois elementos consecutivos em C (*e.g.* A e B), a aresta (A, B) tem que estar definida.
- O tamanho de um caminho é dado pelo número de arestas.



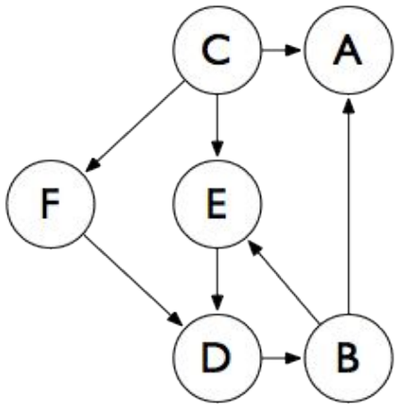
$C_1 = \{F, D, B, A\}$ Tamanho 3

$C_2 = \{C, A\}$ Tamanho 1

$C_3 = \{B, A, C\}$?

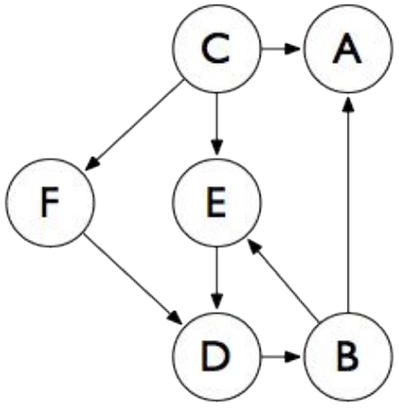
TERMINOLOGIA

- Num caminho acíclico, cada vértice do caminho é único:



$C_1 = \{C, F, D, B, E\}$

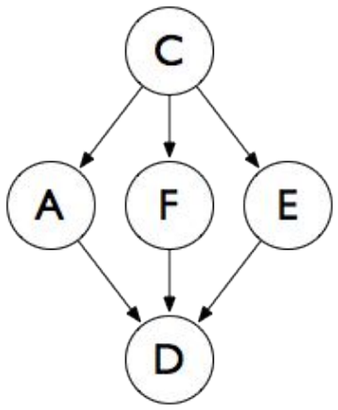
- Num caminho cíclico (size > 2), os vértices inicial e final são iguais:



$C_2 = \{E, D, B, E\}$

TERMINOLOGIA

- Um digrafo sem caminhos cíclicos, é um DAG (*directed acyclic graph*, ou grafo dirigido acíclico).

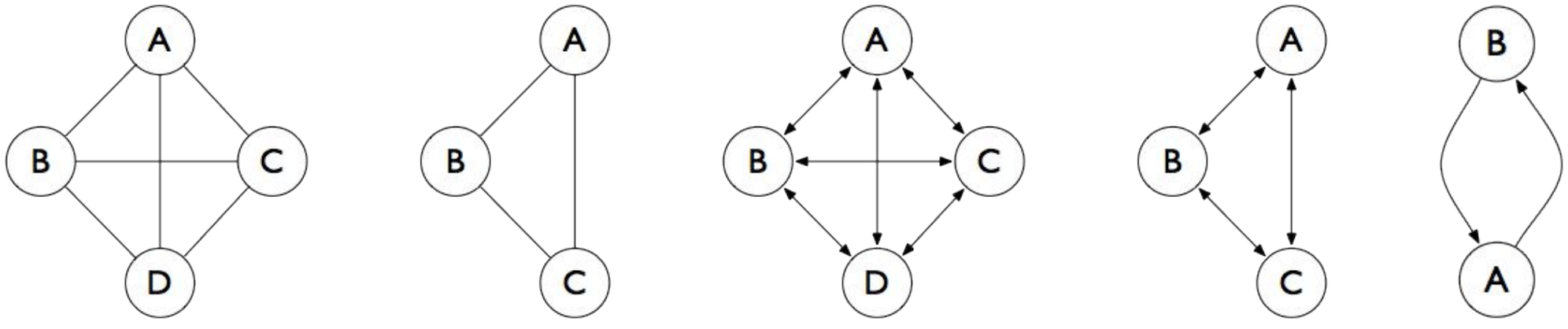


$$Cs = V \cup \{C, A\}, \{C, F\}, \{C, E\} \cup$$

$$\{C, A, D\}, \{C, F, D\}, \{C, E, D\} \cup$$

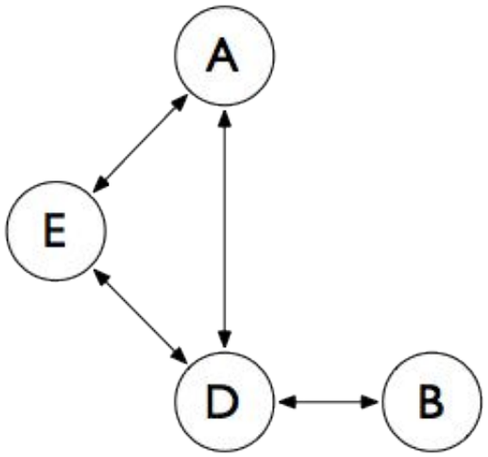
$$\{A, D\}, \{F, D\}, \{E, D\}$$

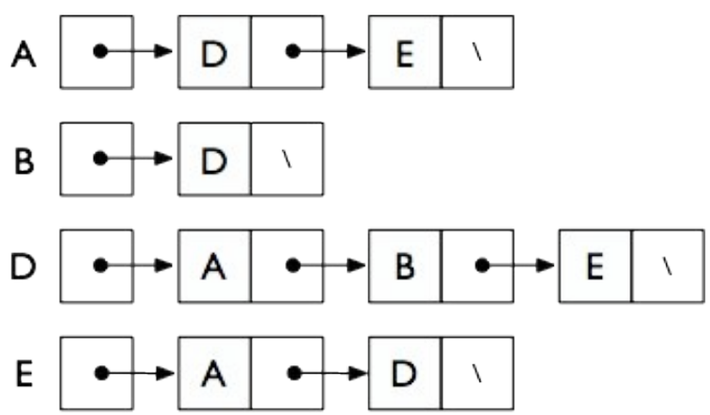
- Um grafo que tem uma aresta entre todos os pares de vertíces é completo:



REPRESENTAÇÃO

- Um grafo pode ser representado por:





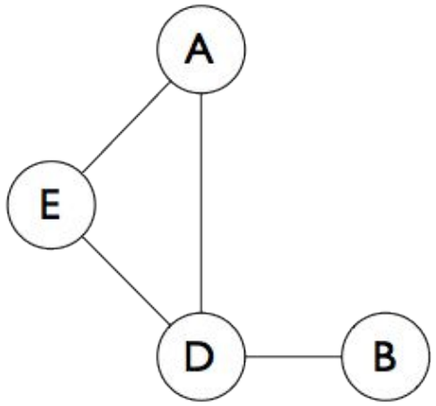
(1) Lista de Adjacências

	A	B	D	E
A	0	0	1	1
B	0	0	1	0
D	1	1	0	1
E	1	0	1	0

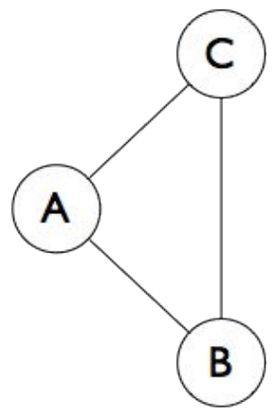
(2) Matriz de Adjacências

REPRESENTAÇÃO

- Num grafo não dirigido, a matriz de adjacências é simétrica:

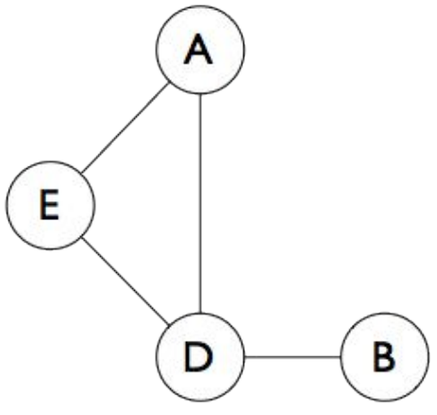


	A	B	D	E
A	0	0	1	1
B	0	0	1	0
D	1	1	0	1
E	1	0	1	0

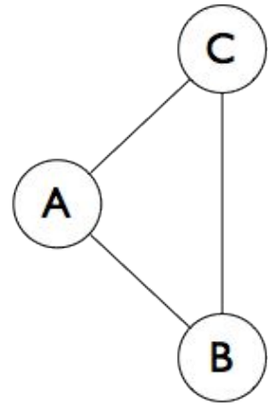


	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

- ... pelo que se pode usar uma matriz triangular:



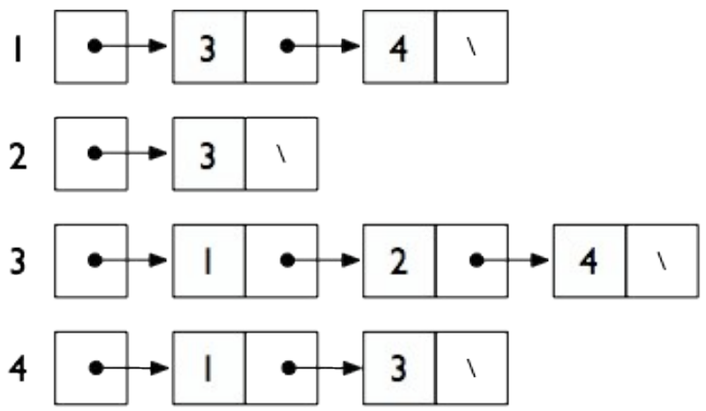
	A	B	D	E
A	0			
B	0	0		
D	1	1	0	
E	1	0	1	0



	A	B	C
A	0		
B	1	0	
C	1	1	0

IMPLEMENTAÇÃO

- Através de uma lista de adjacências:



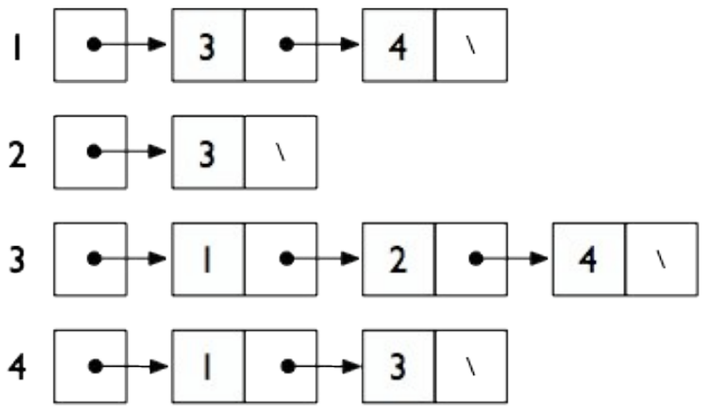

```
typedef struct graphnode {
    int vertex;
    struct graphnode *next;
} Node;

Node** initial(int vertices) {
  return (Node **) malloc(vertices * sizeof(Node *));
}
```

```
void createAdjList(Node** graph, int vertices) {
    int v1, v2, nAdjs, j;
    Node *ptr, *new;

    (...)
}

void printAdjList(Node** graph, int vertices) {
    int i;
    Node *ptr;

    (...)
}
```

```
for (v1 = 0; v1 < vertices; v1++) {
    printf("Quantas adjs para o vértice %d: ", v1+1);
    scanf("%d", &nAdjs);
    ptr = NULL;
    for (j = 0; j < nAdjs; j++) {
        printf("Adjacência %d: ", j+1);
        scanf("%d", &v2);
        new = (Node *) malloc(sizeof(Node));
        if (ptr == NULL) ptr = graph[v1] = new;
        else { ptr->next = new; ptr = ptr->next; }
        ptr->vertex = v2-1;
        ptr->next   = NULL;
    }
}
```

IMPLEMENTAÇÃO

- Através de uma lista de adjacências:



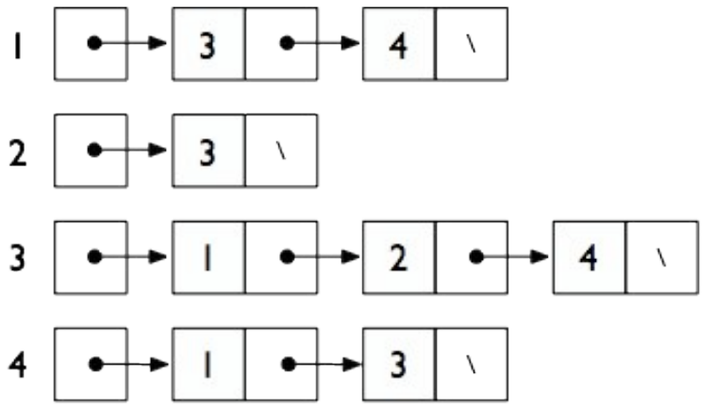

```
typedef struct graphnode {
    int vertex;
    struct graphnode *next;
} Node;

Node** initial(int vertices) {
  return (Node **) malloc(vertices * sizeof(Node *));
}
```

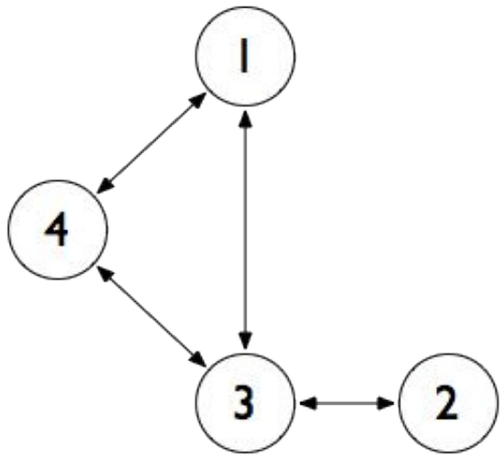
```
void createAdjList(Node** graph, int vertices) {
    int v1, v2, nAdjs, j;
    Node *ptr, *new;

    (...)
}

void printAdjList(Node** graph, int vertices) {
    int i;
    Node *ptr;

    (...)
}
```

```
for (i = 0; i < vertices; i++)
{
    printf(”Arestas %2d", i+1);
    ptr = graph[i];
    while(ptr != NULL){
        printf("  --> %2d", (ptr->vertex+1));
        ptr=ptr->next;
    }
    printf("\n");
}
printf("\n");
```

IMPLEMENTAÇÃO

- Através de uma lista de adjacências:






```
Quantos vértices? 4
Quantas adjacências para o vértice 1: 2
Adjacência 1: 3
Adjacência 2: 4
Quantas adjacências para o vértice 2: 1
Adjacência 1: 3
Quantas adjacências para o vértice 3: 3
Adjacência 1: 1
Adjacência 2: 2
Adjacência 3: 4
Quantas adjacências para o vértice 4: 2
Adjacência 1: 1
Adjacência 2: 3
Arestas 1  --> 3  --> 4
Arestas 2  --> 3
Arestas 3  --> 1  --> 2  --> 4
Arestas 4  --> 1  --> 3
```

IMPLEMENTAÇÃO

- Através de uma matriz:

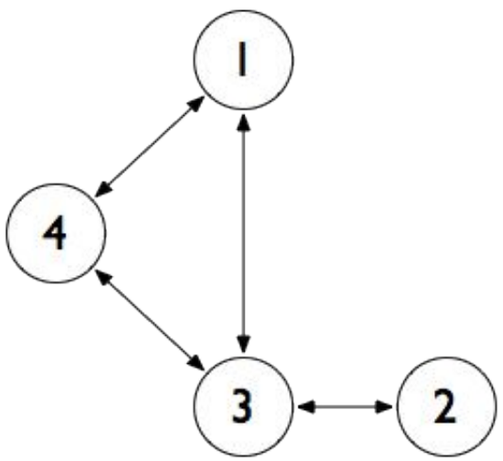
	1	2	3	4
1	0	0	1	1
2	0	0	1	0
3	1	1	0	1
4	1	0	1	0

```
void createFromAdjList(int* graph, int vertices) {
    int v1, v2, nAdjs, j;

    for (v1 = 0; v1 < vertices; v1++) {
        printf("Quantas adjacências para o vértice %d: ", v1+1);
        scanf("%d", &nAdjs);
        for (j = 0; j < nAdjs; j++) {
            printf("Adjacência %d: ", j+1);
            scanf("%d", &v2);
            graph[v1 * vertices + (v2-1)] = 1;
        }
    }
}
void printAdjList(int* graph, int vertices) {
    int i, j;
    int edge;

    for(i = 0; i < vertices; i++) {
        printf("Arestas %2d", i+1);
        for (j = 0; j < vertices; j++) {
            edge = graph[i * vertices + j];
            if (edge == 1) printf("  --> %2d", (j+1));
        }
        printf("\n");
     }
     printf("\n");
}
```

IMPLEMENTAÇÃO

- Através de uma matriz:

	1	2	3	4
1	0	0	1	1
2	0	0	1	0
3	1	1	0	1
4	1	0	1	0




```
Quantos vértices? 4
Quantas adjacências para o vértice 1: 2
Adjacência 1: 3
Adjacência 2: 4
Quantas adjacências para o vértice 2: 1
Adjacência 1: 3
Quantas adjacências para o vértice 3: 3
Adjacência 1: 1
Adjacência 2: 2
Adjacência 3: 4
Quantas adjacências para o vértice 4: 2
Adjacência 1: 1
Adjacência 2: 3
Arestas 1  --> 3  --> 4
Arestas 2  --> 3
Arestas 3  --> 1  --> 2  --> 4
Arestas 4  --> 1  --> 3
```

EXERCÍCIOS

1. Implemente as funções antecessor e sucessor.

   /* retorna 1 se o vertice1 é antecessor do vertice2 no grafo, retorna 0 caso não seja */
   int predecessor(int *graph, int vertices, int vertice1, int vertice2)

   /* retorna 1 se o vertice1 é sucessor do vertice2 no grafo, retorna 0 caso não seja */
   int sucessor(int *graph, int vertices, int vertice1, int vertice2)

2. Implemente a função isValidPath, a qual verifica se um dado caminho é válido para um dado grafo.

   /* retorna 1 se o caminho indicado por path e length é um caminho válido para o grafo, retorna 0 caso não seja */
   int isValidPath(int *graph, int vertices, int *path, int length)