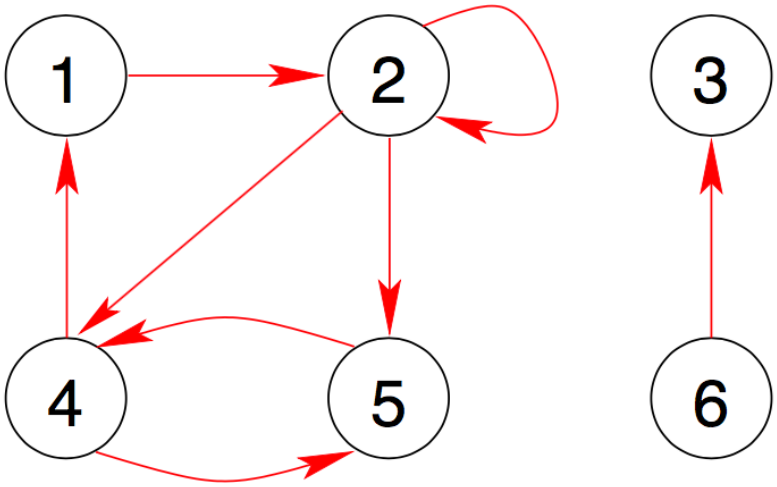
G1. Representação de Grafos em Computador

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:
https://codeboard.io/projects/10725

Grafos orientados

Um *grafo orientado* é um par (V, E) com V um conjunto finito de *vértices* ou *nós* e E uma relação binária sobre V – o conjunto de *arestas* ou *arcos* do grafo.

Exemplo:
$V = \{1, 2, 3, 4, 5, 6\}$
$E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$



Grafo orientado G1

2 é adjacente a 1, mas 1 não é adjacente a 2

- Se $(i, j) \in E$, j diz-se *adjacente* a i
  i e j são respectivamente os vértices de *origem* e *destino* da aresta (i, j)

- O *grau de entrada* de um vértice é o número de arestas com destino nesse vértice
  5 → grau de entrada 2
- O *grau de saída* de um vértice é o número de arestas com origem nesse vértice
  5 → grau de saída 1

- Uma aresta (i, i) designa-se por **anel**

- O número máximo de arestas de um grafo com conjunto de vértices V é V^2
- Um grafo diz-se *esparso* se o número de arestas for muito inferior a V^2, e *denso* em caso contrário

As aplicações dos grafos orientados são inúmeras; além da evidente modelação de redes (por exemplo de estradas), que discutiremos mais à frente, estes grafos têm aplicação por exemplo em *gestão de projectos*, fazendo corresponder os vértices a **tarefas.** As arestas podem exprimir uma relação de precedência: se $(t_1, t_2) \in E$, então a tarefa t_1 deve sempre ser realizada antes da tarefa t_2.

Representação em Computador de Grafos Orientados

Representação por uma Matriz de Adjacências

Assumindo uma **ordenação natural** (enumeração) dos vértices do grafo, a representação mais imediata é por uma matriz de valores Booleanos (ou 0/1) tal que:

- a posição (i, j) contém o valor 1 sse $(i, j) \in E$

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	1	0	1	1	0
3	0	0	0	0	0	0
4	1	0	0	0	1	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0

Em **C** poderemos ter, por exemplo,

```
#define MAX 100
typedef char GraphM[MAX][MAX];
```

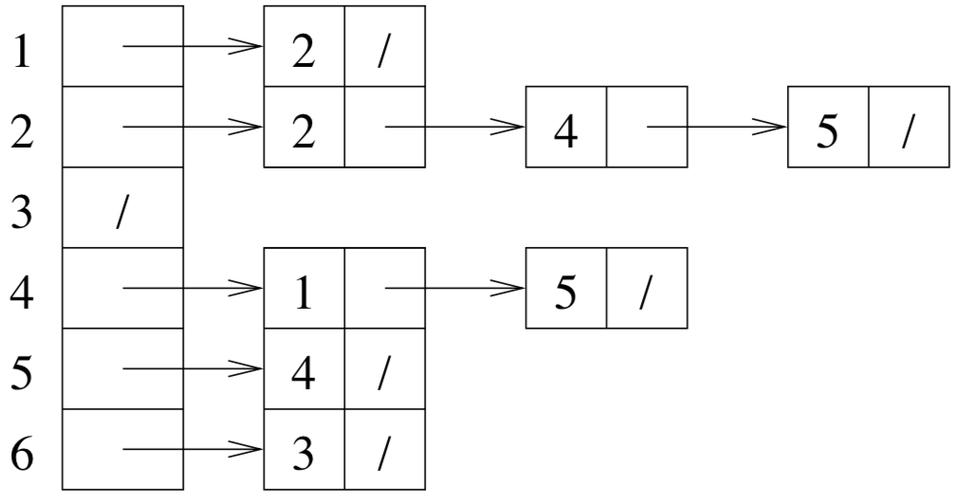
↳ tipo mais simples para representar um booleano

Representação por Listas de Adjacências

Uma representação alternativa consiste em associar a cada vértice do grafo uma lista contendo os vértices que lhe são adjacentes.

Em termos mais concretos, numa linguagem de programação como o C, assumindo uma **ordenação natural** (enumeração) dos vértices do grafo, podemos considerar

- um *array de apontadores*
- cujos índices correspondem aos vértices do grafo, e
- contendo em cada posição o (endereço do) primeiro elemento da lista de adjacência do vértice respectivo



Observe-se que *a soma dos comprimentos das listas ligadas é* $|E|$.

Uma possível definição de tipos em **C** será:

```
#define MAX 100
struct edge {
  int dest;
  struct edge *next;
};
typedef struct edge *GraphL[MAX];
```

Observe-se que o último `typedef` define o tipo `GraphL` como sendo o dos *arrays*, de comprimento `MAX`, de apontadores para estruturas `struct edge`.

Comparação das representações

A escolha de representação poderá passar pela consideração da **densidade** do grafo, e também por uma estimativa do número de **testes de adjacência** que serão efectuados durante a vida desta estrutura, e também da consulta de conjuntos de vértices adjacentes que serão efectuadas sobre o grafo

Com uma matriz de adjacências,

- O espaço de memória necessário para representar um grafo (V, E) é $\Theta(V^2)$, independente do número de arcos, e portanto da densidade do grafo;
- é possível verificar em tempo constante se dois vértices são adjacentes;
- para conhecer os adjacentes a um determinado vértice u, é necessário percorrer *todos* os vértices v do grafo, consultando para cada um a posição (u, v) da matriz.

Por outro lado, com listas de adjacências,

- O espaço de memória necessário para esta representação de um grafo (V, E) é $\Theta(V + E)$, o que a torna uma representação eficiente no caso de grafos esparsos;
- o teste de adjacência de dois vértices obriga à travessia de uma lista ligada, executada no pior caso em tempo $\Theta(V)$;
- a consulta dos adjacentes a um vértice u implica apenas percorrer a lista de adjacências de u, em vez de percorrer todos os vértices. Num grafo pouco denso, o impacto na implementação de um algoritmo que efectue esta operação repetidamente pode ser muito grande. É o caso por exemplo dos algoritmos de travessia de grafos.

Em resumo, se o espaço utilizado não for uma questão crítica, a escolha de representação passará pelo tipo de operação de consulta (adjacência isolada par a par VS. lista de adjacências de um vértices) preponderante nos algoritmos a implementar.

Grafos com Pesos

Em certos contextos é útil associar informação (**pesos**) às arestas de um grafo, em particular numérica. Uma utilização típica de grafos é na modelação de redes de infraestruturas, por exemplo:

- Uma rede de estradas (“mapa”), em que os vértices correspondem a localidades e as arestas a estradas entre elas. Os pesos das arestas neste caso podem corresponder ao comprimento dessas ligações / distância entre duas localidades ligadas por uma estrada directa.

- Uma rede de transporte ou abastecimento de algum recurso, como por exemplo água. Neste caso as ligações corresponderão a canais entre nós da rede, e o peso de uma aresta poderá ser a *capacidade* desse canal.

Representação em Computador de Grafos com Pesos

É imediato adaptar qualquer uma das duas representações acima para grafos com pesos numéricos associados às arestas.

Representação por uma Matriz de Adjacências

Basta armazenar na matriz:

- em vez do valor Booleano 1 para representar a existência de uma aresta, o peso dessa aresta;
- em vez do valor Booleano 0 para representar a inexistência de uma aresta, um valor especial **NE**

A escolha do valor **NE** depende da aplicação em questão. Considerando os dois exemplos anteriores:

- Num grafo que modela uma rede de estradas, a escolha natural para representar a ausência de uma ligação entre duas localizações será **NE** = ∞ (distância infinita)

- Já num grafo que modela uma rede, em que os pesos correspondem a capacidades de canais, a escolha natural para representar a ausência de um canal entre dois nós da rede será **NE** = 0

Em **C** poderemos ter por exemplo a seguinte definição de tipo:

```
#define NE 0
#define MAX 100
```

```
typedef int WEIGHT;
typedef WEIGHT GraphM[MAX][MAX];
```

Representação por Listas de Adjacências
As arestas do grafo são aqui representadas por nós das listas ligadas de adjacências. Sendo assim, basta criar um campo adicional nestas estruturas (nós das listas) para guardar o peso das arestas.

Note-se que, uma vez que nesta representação o teste de existência de uma aresta não é feito pela consulta do valor numérico do peso (como era o caso com uma matriz de adjacências), mas sim pela existência ou não de um nó com um determinado destino na lista ligada, não é agora necessária a utilização de um valor especial **NE** para representar o peso de uma aresta inexistente.

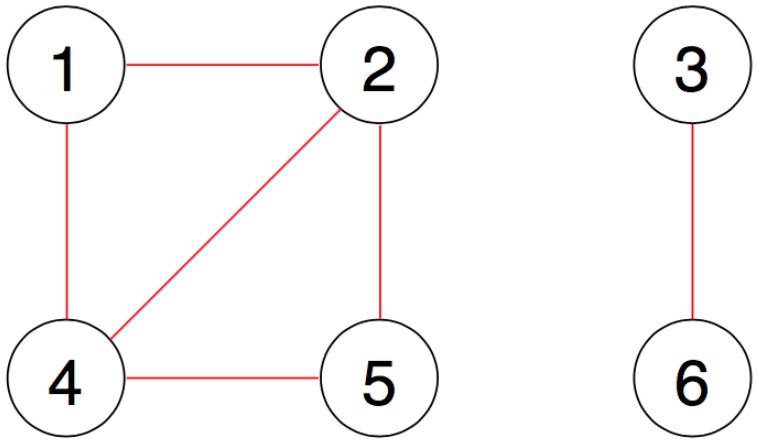
Em **C**:

```
#define NE 0
#define MAX 100
typedef int WEIGHT;
struct edge {
  int dest;
  WEIGHT weight;
  struct edge *next;
};
typedef struct edge *GraphL[MAX];
```

Grafos Não-orientados

Num grafo não-orientado as arestas são conjuntos com dois vértices $\{u, v\} \in E$ em vez de pares ordenados. Por outras palavras, as aresta são bi-direccionais, o que é

adequado, por exemplo, para modelar redes em que todas as ligações entre pares de vértices funcionam nos dois sentidos.



Num grafo não-orientado não se considera habitualmente a existência de anéis (arestas da forma $\{v, v\}$).

Tal como os grafos orientados, estes grafos podem ou não ter pesos associados às arestas.

Representação em Computador de Grafos Não-orientados

A representação típica de um grafo não-orientado passa pela sua conversão para um grafo orientado *simétrico,* em que se $\{u, v\} \in E$ então também $\{v, u\} \in E$.

Note-se que uma tal representação contém *redundância:*

- No caso da representação por uma matriz de adjacências poder-se-á eliminar esta redundância representando de forma eficiente apenas uma matriz triangular.

- No caso da representação por listas de adjacências a eliminação da redundância será quase de certeza uma má ideia. Se se representar a aresta $(u, v) \in E$ apenas por um nó, na lista de adjacência de u ou de v, então, para ter acesso a todos os vértices adjacentes a um qualquer nó não bastará percorrer a sua lista de adjacências; será necessário percorrer *todas* as listas de adjacências do grafo.

EXCERCÍCIOS

No projecto codeboard, o ficheiro `main.c` contém código de teste das funções que deverá definir. Os protótipos das funções encontram-se no ficheiro `conversion.c` [https://codeboard.io/projects/10725]

Considere a representação em C de grafos com pesos, com os tipos de dados fornecidos acima.

1. Defina funções de conversão entre as duas representações (matrizes e listas de adjacências).

2. Para cada uma das representações defina funções de cálculo do **grau de entrada** e de **saída** de um vértice.
   a. Analise o tempo de execução das 4 funções definidas.

3. Assuma agora uma representação por listas de adjacências.
   A *capacidade* de um vértice v num grafo define-se como a diferença entre a soma dos pesos das arestas que têm v como destino e a soma dos pesos das arestas que têm v como origem. Defina uma função que calcula a capacidade de um vértice num grafo.
   a. Analise o esforço computacional desta função.

4. Defina uma função `maxCap` que determina o vértice do grafo com maior capacidade. A função deverá executar em tempo $\Theta(|V| + |E|)$.

EXCERCÍCIOS ADICIONAIS

1. Defina uma função `colorOK (GraphL g, int color[])` que, dado um grafo não orientado `g` e um vector de inteiros `cor` verifica se essa coloração é válida. Diz-se que uma coloração é válida sse vértices adjacentes tiverem cores diferentes.

2. Defina em C uma função que calcula o inverso de um grafo (um grafo que tem uma aresta (u, v) se e só se existe uma aresta (v, u) no grafo original).

```c
int readGraphM_sol(GraphM g) {
    int i, j, n;

    printf("Número de vértices do grafo? (<=%d)\n", MAX);
    scanf("%d", &n);
    if (n > MAX) return 0;

    printf("\nIntroduza matriz de adjacência do grafo:\n");

    for(i=0; i<n; i++)
       for(j=0; j<n; j++)
           scanf("%d", &g[i][j]);

    return n;
}
```

```c
void printGraphM_sol(GraphM g, int n) {
    int i, j;

    for(i=0; i<n; i++) {
        for(j=0; j<n; j++)
            printf("%d ", g[i][j]);
        printf("\n");
    }
}
```

```c
void printGraphL_sol(GraphL g, int n) {
    int i;
    struct edge *p;

    for(i=0; i<n; i++) {
        printf("%d", i);
        for(p=g[i]; p; p=p->next)
            printf(" --> (%d,%d)", p->dest, p->weight);
        printf("\n");
    }
}
```

```c
// assume listas de adjacência ordenadas
// por ordem crescente do vértice destino
int inDegree_sol (GraphL g, int j, int n) {
    int i, k = 0;
    struct edge *p;

    for (i=0; i<n; i++) {
        for (p = g[i]; p && p->dest < j ; p=p->next) ;
        if (p->dest == j) k++;
    }

    return k;
}
```

```c
int outDegree_sol (GraphL g, int i) {
    int n = 0;
    struct edge *p;

    for (p = g[i]; p ; p=p->next) n++;

    return n;
}
```

```c
// assume listas de adjacência ordenadas
// por ordem crescente do vértice destino
void graphLtoM_sol(GraphM gm, GraphL gl, int n) {
    int i, j;
    struct edge *p;

    for (i = 0; i<n; i++)
        for (j = 0, p = gl[i]; j<n; j++) {
            if (p && p->dest == j) {
                gm[i][j] = p->weight;
                p = p -> next;
            }
            else
                gm[i][j] = NE;
        }
}
```

```c
void graphMtoL_sol(GraphL gl, GraphM gm, int n) {
    int i, j;
    struct edge *head, *new;

    for (i = 0; i<n; i++) {
        head = NULL;
        for (j = n-1; j>=0; j--)
            if (gm[i][j] != NE) {
                new = malloc(sizeof(struct edge));
                new -> dest = j;
                new -> weight = gm[i][j];
                new -> next = head;
                head = new;
            }
        gl[i] = head;
    }
}
```

```c
int capacidadeL_sol (GraphL g, int v, int n) {
    int i, k = 0;
    struct edge *p;

    for (i=0; i<n; i++)
        if (i==v)
            for (p = g[i]; p; p = p->next)
                if ( p->dest != i) k -= p->weight;
        else
            for (p = g[i]; p && p->dest <= v ; p = p->next)
                if (p->dest == v) k += p->weight;

    return k;
}
```

```c
int maxCap_sol (GraphL g, int n) {
    int i, k = 0;
    int caps[n];
    struct edge *p;

    for (i=0; i<n; i++)
        caps[i] = 0;

    for (i=0; i<n; i++)
        for (p = g[i]; p; p=p->next) {
            caps[i] -= p-> weight;
            caps[p->dest] += p->weight;
        }

    k = 0;
    for (i=1; i<n; i++)
        if (caps[i] > caps[k]) k = i;

    return k;
}
```