Algoritmos e Complexidade

Algoritmos sobre Grafos

José Bernardo Barros
Departamento de Informática
Universidade do Minho

Conteúdo

1 Introdução

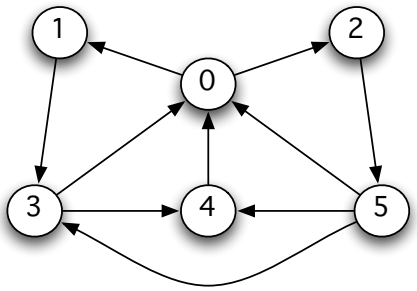
Grafos são estruturas matemáticas usadas nas ciências de computação (e noutras) para modelar relações entre elementos de um determinado conjunto.
Um grafo $G = (V, E)$ é composto por duas partes.

- Um conjunto V finito de **vértices** (ou nodos) que corresponde aos elementos que queremos relacionar.
- Um conjunto $E \subseteq V^2$ de `arestas` (ou arcos) que traduzem o relacionamento entre os vários elementos. Cada uma destas arestas relaciona dois vértices (chamados as extremidades da aresta).

Quando a ordem das extremidades das arestas não é relevante dizemos que se trata de um **grafo não orientado** (e corresponde a uma relação simétrica); por outro lado, se tal não acontecer o grafo diz-se **orientado** e as extremidades de uma aresta chamam-se a **origem** e o **destino** dessa aresta.
Nalguns casos é ainda usado, para cada aresta, um atributo adicional – o seu **peso** ou custo. Nesse caso dizemos que se trata de um **grafo pesado**.
Alguns exemplos de aplicação de grafos em informática são:

- sistemas de navegação (mapas)
- redes de distribuição
- hipertexto
- redes sociais

Na representação gráfica de grafos é costume usar círculos para representar os vértices e linhas (ou setas) para representar as arestas.
À direita apresenta-se um grafo orientado (e não pesado) cujos vértices são o conjunto $\{0, 1, 2, 3, 4, 5\}$. Este grafo tem 10 arestas que relacionam alguns dos vértices do grafo.



1.1 Implementações

Na implementação de grafos é costume separar-se a representação do conjunto dos vértices e do conjunto das arestas.
A representação do conjunto de arestas deve permitir, de uma forma eficiente:

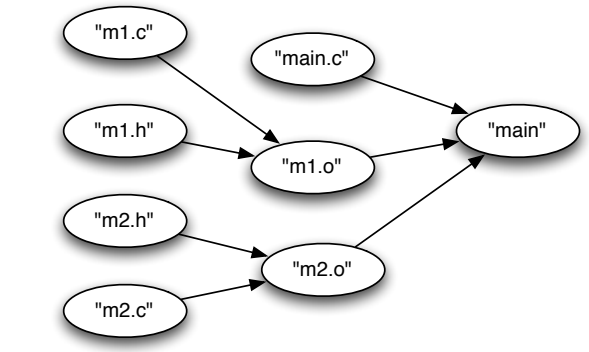
- determinar se uma aresta existe (e no caso dos grafos pesados, qual o seu peso)
- percorrer o conjunto dos vértices adjacentes a um dado vértice (i.e., percorrer as arestas com uma dada origem)

Como veremos mais adiante, há alguns algoritmos que precisam de percorrer de uma forma eficiente a lista dos antecessores de um dado vértice.
A forma mais eficiente de aceder à informação associada a um item é usando um array. Mas os arrays, na generalidade das linguagens de programação, usam como índices números inteiros (não negativos). Por essa razão é costume guardar a informação das arestas de um grafo como se os vértices desse grafo fossem índices de um array.
Na representação do conjunto de vértices, e se de facto esses vértices não forem os índices referidos atrás, deve ser armazenada a correspondência entre esses e os índices que lhe estão associados na representação das arestas.

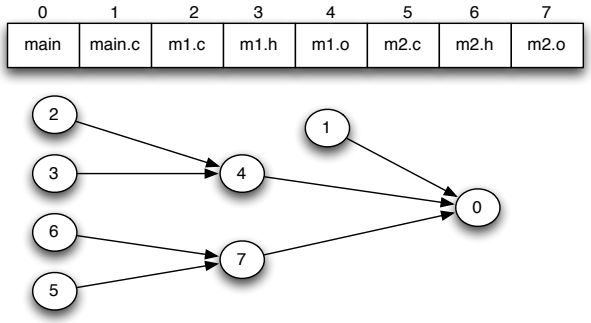
Exemplo 1 Suponhamos que queremos usar um grafo para representar a relação de dependências de ficheiros presente na seguinte *makefile*:

```
main: main.c m1.o m2.o
   gcc -o main main.c m1.o m2.o
m1.o: m1.h m1.c
   gcc -c m1.c
m2.o: m2.h m2.c
   gcc -c m2.c
```

O conjunto dos vértices de tal grafo é V = { `"main"`, `"main.c"`, `"m1.o"`, `"m2.o"`, `"m1.h"`, `"m1.c"`, `"m2.h"`, `"m2.c"`}
As dependências entre os vários ficheiros são dadas pelo grafo ao lado.



Uma forma alternativa de representar os dados deste problema passa então por estabelecer uma relação entre os vértices do grafo e um conjunto de índices (por exemplo usando um array em que cada posição contém o nome de um ficheiro) e um grafo cujos nodos são os índices desse array.



Tal como evidenciado por este exemplo, o problema da representação das arestas de um grafo com N vértices pode ser resolvido assumindo que os vértices são os números $\{0 \ldots N-1\}$. As representações que se apresentam a seguir assumem por isso esse conjunto fixo de vértices.

1.1.1 Matrizes de Adjacência

A forma mais imediata de representar as arestas de um grafo consiste em usar uma matriz em que na posição (i, j) se encontra a informação sobre a aresta com origem i e destino j.

Num grafo não pesado precisamos apenas de saber se essa aresta existe ou não. Trata-se por isso de uma matriz de booleanos.

Num grafo pesado precisamos de saber qual o peso da aresta, caso ela exista. Na maioria dos casos que vamos analizar os custos das arestas nunca são nulos, pelo que podemos usar 0 para assinalar que uma dada aresta não existe. por exemplo

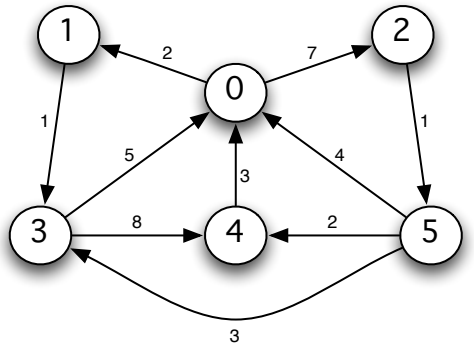
Esta representação é conhecida como **matrizes de adjacência**.
Quando o grafo é não orientado, a aresta com origem i e destino j é indistinguível da aresta com origem j e destino i. Por isso, se usarmos esta representação teremos uma matriz simétrica.

Uma possível definição de tipos para esta representação será:

```
#define NV ...  // numero de vertices
#define NE 0    // peso da aresta que nao existe

typedef int GraphMat [NV][NV];
```

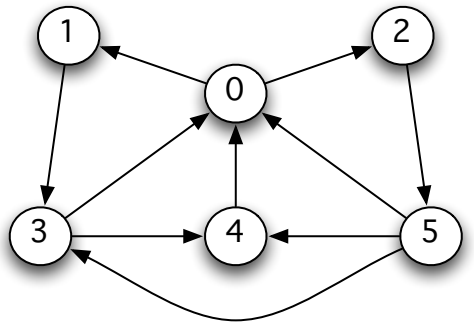
Exemplo 2 O grafo apresentado à direita poderia ser implementado em matrizes com as definições que se apresentam à esquerda.

```
#define NV 6

GraphMat g1
  = {{NE,  2,  7, NE, NE, NE},
     {NE, NE, NE,  1, NE, NE},
     {NE, NE, NE, NE, NE,  1},
     { 5, NE, NE, NE,  8, NE},
     { 3, NE, NE, NE, NE, NE},
     { 4, NE, NE,  3,  2, NE},
    };
```



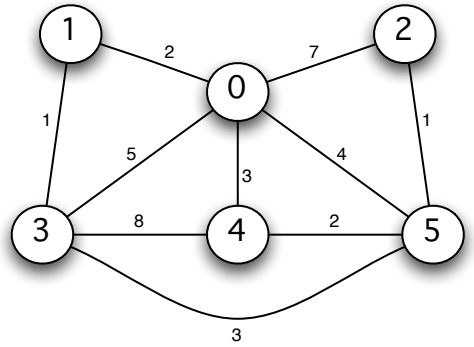
Exemplo 3 O grafo não pesado da página 2 (e reproduzido abaixo) tem a seguinte representação em matrizes (seguindo a tradição do C, usamos `0` para representar `Falso`).

```
GraphMat g2
  = {{ 0,  1,  1,  0,  0,  0},
     { 0,  0,  0,  1,  0,  0},
     { 0,  0,  0,  0,  0,  1},
     { 1,  0,  0,  0,  1,  0},
     { 1,  0,  0,  0,  0,  0},
     { 1,  0,  0,  1,  1,  0},
    };
```



Exemplo 4 O grafo não orientado apresentado à direita poderia ser implmentado em matrizes com as definições que se apresentam à esquerda.

```
GraphMat g3
  = {{NE,  2,  7,  5,  3,  4},
     { 2, NE, NE,  1, NE, NE},
     { 7, NE, NE, NE, NE,  1},
     { 5,  1, NE, NE,  8,  3},
     { 3, NE, NE,  8, NE,  2},
     { 4, NE,  1,  3,  2, NE},
    };
```



O custo (em termos de memória) da representação em matriz de um grafo com V vértices e E arestas é proporcional a V^2, e por isso independente do número de arestas do grafo.

Na maioria das aplicações de grafos o número de arestas é muito inferior ao limite superior. Isto acontece porque a maioria dos vértices está relacionada (i.e., existe uma aresta) com uma percentagem muito pequena dos vértices.

- No caso de um mapa, os vértices correpondem aos cruzamentos e as arestas às estradas.
- No caso das redes sociais, cada vértice corresponde a um utilizador e as arestas a relações (directas) entre eles.

Nesses casos torna-se pouco ajuízado (e muitas vezes proibitivo) usar como representação do grafo uma matriz – a maioria dos elementos dessa matriz é nula. Por isso é costume usar técnicas conhecidas de armazenamento de matrizes esparsas.

1.1.2 Listas de Adjacência

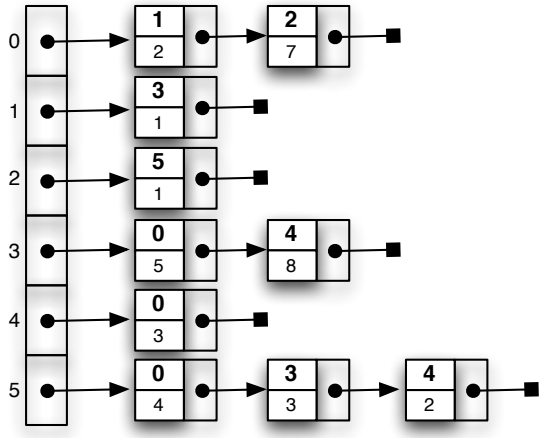
Uma forma de armazenar uma matriz esparsa passa por armazenar apenas os elementos não nulos, agrupados por linhas (por exemplo numa lista ligada). Esta representação é normalmente referida por **listas de adjacência**.
Uma possível definição de tipos para esta representação será:

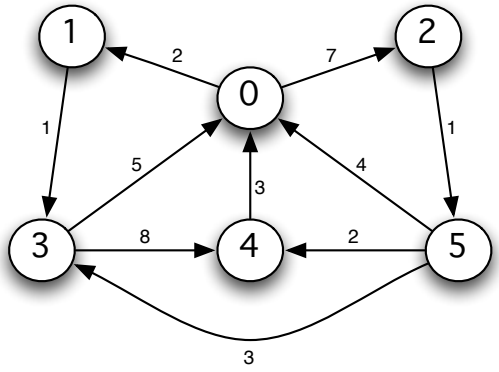
```
#define NV ... // numero de vertices

typedef struct edge {
  int dest; // destino da aresta
  int cost; // peso da aresta
  struct edge *next;
} *EList;
typedef EList Graph [NV];
```

Para cada linha da matriz, i.e., para cada vértice do grafo, armazenamos apenas os elementos não nulos da matriz (i.e., as arestas que têm esse vértice como origem), indicando a coluna (i.e., o destino da aresta) e o seu valor (i.e., o peso da aresta).
No caso dos grafos não pesados, a definição poderá ser simplificada, não incluindo o campo `cost` em cada aresta.

Mais uma vez, no caso dos grafos não orientados, é costume armazenar duas arestas (direccionadas) por cada aresta do grafo.

Exemplo 5 O grafo não orientado do exemplo 4 (apresentado à direita) poderia ser implementado em listas de adjacência como se mostra abaixo.





Vejamos então como construír um grafo em listas de adjacência a partir de uma matriz de adjacências.
Para isso devemos percorrer toda a matriz e, para cada aresta encontrada, acrescentá-la à lista correspondente. Uma forma de garantir que, usando inserção à cabeça nas listas, estas ficam ordenadas por ordem crescente do destino (o que é pouco relevante na generalidade dos casos), podemos percorrer cada linha por ordem inversa.

```
void matTolist (GraphMat go, Graph gd){
  int i, j;
  EList tmp;
  for (i=0;i<NV;i++){
    // preencher a linha i
    gd[i]=NULL;
    for (j=NV-1;j>=0;j--)
      if (go[i][j] != NE) {
        // acrescentar aresta i,j
        tmp = malloc (sizeof(struct edge));
        tmp->dest = j;
        tmp->cost = go[i][j];
        tmp->next = gd[i];
        gd[i] = tmp;
      }
  }
}
```

Exercício 1 Defina a função `void listTomat (Graph go, GraphMat gd)` de conversão de listas de adjacência para matrizes de adjacência (inversa da anterior).

De seguida, e para exemplificar como um grafo representado em listas de adjaçência pode ser consultado, vamos apresentar a definição de uma função que conta o número de arestas de um grafo nesta representação.

```
int nArestas (Graph g){
  int r = 0; int i;
  EList it; // para percorrer as listas

  for (i=0; i<NV; i++)
    // percorrer os adjacentes a i
    for (it = g[i]; it!=NULL; it=it->next)
      // existe aresta de i para it->dest
      r++;

  return r;
}
```

A análise da complexidade desta função deve ser feita em função do tamanho da representação do grafo. É costume desdobrar este tamanho em dois itens: V – o número de vértices do grafo e E – o número de arestas do grafo.
O ciclo mais exterior faz V iterações. As várias instâncias do ciclo interior não têm um número fixo de iterações: cada instância corresponde às arestas que saem de um dado vértice. No entanto, o somatório do número de iterações dessas instâncias corresponde ao número de arestas do grafo.
Desta forma podemos concluir que

$$T_{\text{nArestas}}(V, E) = \Theta(V + E)$$

Note-se que esta função, para a representação de grafos em matrizes de adjacência, tem uma complexidade $\Theta(V^2)$ uma vez que todos os elementos da matriz terão de ser consultados.

1.1.3 Vector de Adjacência

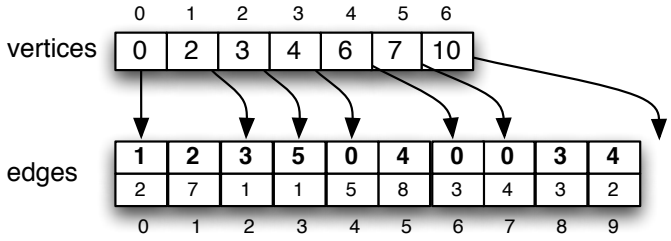
Como vimos acima, a grande motivação para usar listas de adjacência em vez de matrizes de adjacência prende-se com a memória necessária.
Uma outra representação, menos comum, e que resulta da *compactação* da representação anterior consiste em guardar todas as arestas do grafo num único vector, mantendo contíguas as arestas que têm a origem em comum.
Desta forma, e para ter acesso aos vértices adjacentes a um dado vértice precisamos apenas de guardar os índices onde se encontram o primeiro e o último.
De facto precisamos apenas de guardar o índice onde se encontra o primeiro sucessor: o índice onde se encontra o último pode ser obtido a partir do índice onde começam os sucessores do vértice seguinte.
Uma possível definição de tipos para esta representação será:

```
#define NV   ... // numero de vertices
#define NEd  ... // numero de arestas
```

```
typedef struct edgeV {
  int dest; // destino da aresta
  int cost; // peso da aresta
} Edge;
typedef struct {
  int vertices [NV+1];
  EdgeV edges [NEd];
  } GraphVect;
```

A definição do primeiro array com mais uma componente, como veremos adiante, permite uma maior simplicidade nas definições das várias funções de acesso aos vértices adjacentes a um vértice.

Exemplo 6 O grafo não orientado do exemplo 5 poderia ser implementado em vectores de adjacência da seguinte forma.



Vamos terminar esta secção apresentando funções de conversão entre estas duas representações.

```
void listToVect (Graph go, GraphVect *gd){
  int i, k;
  EList it;

  for (i=k=0;i<NV;i++) {
    gd->vertices[i] = k;
    for (it=go[i]; it!=NULL; it = it->next){
      gd->edges[k].dest = it->dest;
      gd->edges[k].cost = it-> cost;
      k++;
    }
  }
  gd->vertices[i] = k;
}

void vectToList (GraphVect *go, Graph gd){
  int i, k;
  EList tmp;

  for (i=0; i<NV; i++){
```

```
    gd [i] = NULL;
    for (k=go->vertices[i+1]-1;k>=go->vertices[i];k--){
      tmp = malloc (sizeof(struct edge));
      tmp->dest = go->edges[k].dest;
      tmp->cost = go->edges[k].cost;
      tmp->next = gd[i];
      gd[i]=tmp;
    }
  }
}
```

2 Consultas

De forma a comparar as formas de aceder à informação de um grafo nas várias implementações vistas vamos apresentar algumas funções de consulta dessa informação.

2.1 Peso de uma aresta

Pretende-se, dado um grafo e dois vértices, determinar, caso exista, o peso da aresta que os liga. A função deverá retornar verdadeiro ou falso e, em caso afirmativo, colocar num dado endereço o peso da aresta.
No caso das matrizes de adjacência esta função executará em tempo constante pois temos acesso directo à correspondente posição da matriz.

```
int edgeWMat (GraphMat g, int o, int d, int *w){
  *w = g[o][d];
  return (*w != NE);
}
```

No caso das listas de adjacência, a aresta deve ser procurada na lista dos vértices adjacentes ao primeiro.

```
int edgeW (Graph g, int o, int d, int *w){
  EList it;
  int found=0;
  for (it=g[o]; (it != NULL) && !found; it=it->next)
      found = (it->dest == d);
  if (found) *w = it->cost;
  return found;
}
```

No pior caso esta função percorre todos os adjacentes da lista (que podem ser no máximo V) e por isso o seu tempo de execução é $\mathcal{O}(V)$. No melhor caso o vértice `o` não tem adjacentes e a função executa em tempo constante.
No caso dos vectores de adjacência a única diferença reside na forma como os vértices adjacentes são percorridos. A complexidade da função é por isso igual a esta última.

```
int edgeWVect (GraphVect *g, int o, int d, int *w){
  int k;
  int found=0;
  for (k=g->vertices[o]; (k<g->vertices[o+1]) && !found; k++)
      found = (g->edges[k].dest == d);
  if (found) *w = g->edges[k].cost;
  return found;
}
```

2.2 Grau de entrada e saída

Dado um grafo e um vértice desse grafo, o grau de entrada desse vértice define-se como o número de arestas que têm esse vértice como destino. Da mesma forma, o grau de saída de um vértice corresponde ao número de arestas que têm esse vértice como origem.
No caso da implementação em matrizes de adjacência, estes dois problemas têm uma resuloção idêntica uma vez que o acesso aos antecessores de um vértice se faz de forma análoga ao acesso aos sucessores.

```
int indegreeMat (GraphMat g, int v){
  int i, r;
  for (i=r=0; i<NV; i++)
    if (g[i][v] != NE) r++;
  return r;
}
```

```
int outdegreeMat (GraphMat g, int v){
  int i, r;
  for (i=r=0; i<NV; i++)
    if (g[v][i] != NE) r++;
  return r;
}
```

Ambas as funções percorrem uma linha ou coluna da matriz que representa o grafo e têm por isso uma complexidade linear no número de vértices do grafo ($\Theta(V)$).

Nas outras representações apresentadas, o acesso aos sucessores de um vértice é substancialmente mais eficiente do que o acesso aos seus antecessores. Dessa foram o cálculo do grau de entrada será bastante menos eficiente.
Vejamos em primeiro lugar as definições para grafos implementados em listas de adjacência.
O cálculo do grau de saída resume-se ao cálculo do comprimento de uma lista ligada.

```
int outdegree (Graph g, int v){
  int r=0; EList it;

  for (it=g[v]; it!=NULL; it=it->next)
    r++;
  return r;
```

```
}
```

Uma vez que as listas de adjaçência têm no máximo V elementos, a complexidade desta função é limitada superiormente por esse valor – $\mathcal{O}(V)$ – o que normalmente é um comportamento melhor do que o apresentado para a definição baseada em matrizes.

No caso do cálculo do grau de entrada nesta representação, e como não temos as arestas com um destino comum organizadas de qualquer forma, teremos que percorrer todas as arestas e por isso o função terá complexidade linear no tamanho do grafo ($\Theta(V + E)$).

```
int indegree (Graph g, int v){
  int r, i; EList it;

  for (i=r=0; i<NV; i++)
    for (it=g[i]; it!=NULL; it=it->next)
      if (it->dest == v) r++;
  return r;
}
```

No caso dos vectores de adjacência o cálculo do grau de saída é particularmente eficiente.

```
int outdegreeVect (GraphVect *g, int v){
  return (g->vertices[v+1] - g->vertices[v]);
}
```

No caso do cálculo do grau de entrada a complexidade é muito próxima do que se apresentou para listas de adjacência.

```
int indegreeVect (GraphVect *g, int v){
  int r, i; EList it;

  for (i=r=0; i<NE; i++)
    if (g->edges[i].dest == v) r++;
  return r;
}
```

Estes exemplos de consulta da informação de um grafo põe em evidência que a principal diferença na implementação dos vários algoritmos reside na forma como os vértices adjacente a um dado vértice são percorridos. Daí que nos problemas que vamos apresentar apenas nos foquemos na represenatação mais comum, i.e., usando listas de adjaçência.

Exercício 2 Num grafo orientado e pesado, a capacidade de um vértice v define-se como a soma dos pesos das arestas que têm v como destino (entradas em v) subtraída da soma dos pesos das arestas que têm v como origem (saídas de v).

1. Defina uma função `int capacidade (Graph g, int v)` que calcula a capacidade de um dado vértice.

2. Defina uma função `int maxCap (Graph g)` que, dado um grafo determina o vértice com capacidade máxima. Garanta que a função executa em tempo linear do tamanho do grafo (e por isso que não se baseia na chamada repetida da função da alínea anterior).

Exercício 3 Dado um grafo $g = (V, E)$, um subconjunto $C \subseteq V$ diz-se uma **cobertura** de g se e só se para todas as arestas $(o, d) \in E$ se tem que $(o \in C) \vee (d \in C)$, i.e., o conjunto c contem pelo menos uma das extremidades de todas as arestas do grafo.
Considere que se representam sub-conjuntos de vértices usando um array de *flags*. Defina uma função `int vCoverage (Graph g, int c[])` que, dado um grafo `g` e um conjunto `c`, testa se esse conjunto é um cobertura do grafo.

Exercício 4 Um problema muito conhecido sobre grafos consiste em determinar se é possível atribuír cores aos vértices do grafo de tal forma que dois vértices ligados por uma aresta tenham cores diferentes.
Defina uma função `int colorOK (Graph g, int cor[])` que, dado um grafo `g` e uma coloração `c` (`c[x]` representa a cor do vértice `x`), testa se essa coloração é válida, i.e., se para cada aresta (o, d) do grafo, `c[o]` e `c[d]` são diferentes.

Exercício 5 Dado um grafo orientado e acíclico $g = (V, E)$, uma sequência $s = \langle v_1, v_2 \dots v_n$ dos vértices do grafo diz-se uma **ordenação topológica** de g se e só se, para cada aresta (o, d) o vértice o aparece na sequência **antes** do vértice d.
Defina uma função `int testTop (Graph g, int v[])` que testa se uma dada sequência de vértices `v` é uma ordenação topológica de `g`.
Assuma que o array `v` contém todos os (`NV`) vértices do grafo sem repetições. Garanta que a sua solução executa em tempo linear no tamanho do grafo ($\mathcal{O}(V + E)$).

3 Procura e Travessias

Dado um grafo $G = (V, E)$, um **caminho** em G é uma sequência de vértices

$$\langle v_1, v_2, \dots, v_n \rangle$$

em que existe uma aresta entre cada par de vértices consecutivos, i.e., $(v_i, v_{i+1}) \in E$. Os vértices v_0 e v_n chamam-se a **origem** e o **destino** do caminho.
Uma classe importante de problemas sobre grafos consiste em determinar se dados dois vértices o e d existe um caminho com origem o e destino d. Nesse caso dizemos que o vértice d é **alcançável** a partir de o
Este problema pode ser visto como um caso particular de determinar os vértices que são alcançáveis a partir de um dado vértice. Este problema é normalmente conhecido como uma **travessia** do grafo (a partir de um destino).

3.1 Depth first → Travessia em profundidade

A definição de caminho entre dois vértices dada acima tem uma formulação recursiva que facilmente se traduz num processo de determinação de caminhos:

Uma sequência de vértices $\langle v_1, v_2, \ldots, v_n \rangle$ é um caminho no grafo $g = (V, E)$ sse

1. $n = 1$, i.e., é uma sequência singular e por isso trata-se de um caminho vazio (sem arestas);
2. $(v_1, v_2) \in E$ e além disso $\langle v_2, \ldots, v_n \rangle$ é um caminho no grafo g (de v_2 a v_n).

Vejamos então como esta definição recursiva pode ser usada para determinar se há um caminho no grafo g entre dois vértices específicos.

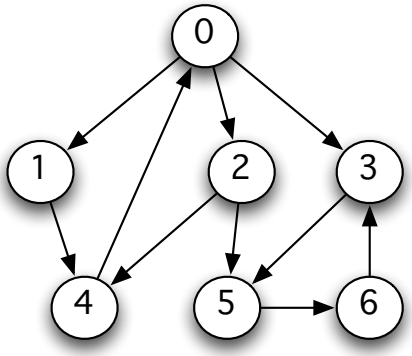
```
int procura (Graph g, int o, int d){
  int found = 0;
  EList it;
  if (o==d) found = 1;
  else
    for (it = g[o]; it != NULL && !found; it = it->next)
      found = procura (g,it->dest,d);
  return found;
}
```

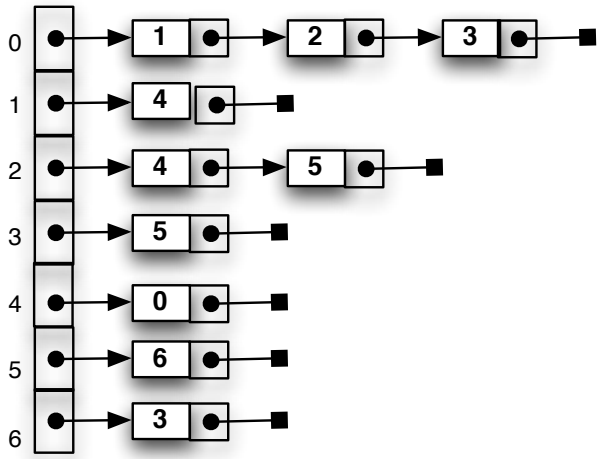
Esta definição tem no entanto um problema: num grafo cíclico é possível que esta função não termine. Isto porque não temos nenhuma forma de determinar se um determinado vértice já foi incluído no caminho, i.e., se o caminho que estamos a construír não se trata de um caminho cíclico.
Uma forma de contornar este problema passa por adicionar aos argumentos da função a informação sobre os vértices que já foram incluídos no caminho e evitar dessa forma a procura usando caminhos cíclicos.
Vamos então usar um vector adicional, passado como argumento, que marca todos os vértices já incluídos na procura. Esse array terá de ser inicializado (uma única vez) e por isso devemos definir a função em dois passos: um (recursivo) que faz a procura ao longo do grafo e um outro, anterior que começa por inicializar o referido array.

```
int procura (Graph g, int o, int d){
  int visitados [NV]; int i;
  for (i=0;i<NV;i++)
    visitados[i] = 0;
  return (procuraRec (g,o,d,visitados));
}

int procuraRec (Graph g, int o, int d, int v[]){
  int found = 0;
  EList it;
  v[o] = 1;
  if (o==d) found = 1;
  else
    for (it = g[o]; it != NULL && !found; it = it->next)
      if (! v[it->dest])
```

```
        found = procuraRec (g,it->dest,d,v);
    return found;
}
```

Exemplo 7 Para melhor entender o funcionamento desta função, vamos simular o seu comportamento para um caso em concreto: determinar se o vértice 3 é alcançável a partir do vértice 0 no grafo que se apresenta abaixo (apresenta-se ainda a representação do grafo em listas de adjacência).





A invocação da função `procura` começa por inicializar o array `visitados` com

`visitados` =

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

De seguida invoca a função `procuraRec` com `o=0`, `d=3` e `v=visitados`. Vejamos agora a sequência de invocações desta função e o valor dos argumentos.

`v` =

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

, `o=0`, `d=3`

os sucessores de 0 são 1, 2 e 3

- (para `it->dest=1`)

  `v` =

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

  , `o=1`, `d=3`

  o único sucessor de 1 é 4

  - (para `it->dest=4`)

    `v` =

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

    , `o=4`, `d=3`

    o único sucessor de 4 é 1 e `v[1]!=0`

- (para `it->dest=2`)

  `v` =

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

  , `o=2`, `d=3`

  os sucessores de 2 são 4 e 5. Mas `v[4]!=0`

  - (para `it->dest=5`)

    `v` =

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

    , `o=5`, `d=3`

    o único sucessor de 5 é 6

* (para `it->dest=6`)

  v = | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
      | 1 | 1 | 1 | 0 | 1 | 1 | 0 | , o=6, d=3

  o único sucessor de 6 é 3

  · (para `it->dest=3`)

    v = | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
        | 1 | 1 | 1 | 0 | 1 | 1 | 1 | , o=3, d=3

    atingimos o caso de base, i.e., o ramo não recursivo da função. Vai ser retornado o valor 1.

O array v tem como valor final | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | , que mostra que tivemos que percorrer todos os vértices para concluir que 3 é alcançável a partir de 0.

Este exemplo põe em evidência uma particularidade da passagem de arrays como argumentos em C: as alterações feitas a um array passado como argumento são efectivas mesmo após o fim da evocação.
Daí que no final o array `visitados` tenha a informação sobre quais os vértices que foram visitados.
No exemplo em análise foram percorridos todos os vértices do grafo. Mas tal não era de todo obrigatório (basta ver o que aconteceria se a função fosse invocada com `0=0` e `d=1`).
No entanto o exemplo mostra que é relativamente fácil adaptar a definição acima de forma a visitar todos os vértices alcançáveis a partir de um dado vértice.
Vejamos então como fazer tal modificação, definindo uma função que, dado um grafo e um nodo desse grafo, determina quantos vértices são alcançáveis a partir desse vértice. Este problema é, tal como referido acima, conhecido como uma travessia do grafo a partir de um vértice. A estratégia que vamos usar (idêntica à da função de procura anterior) é conhecida como **travessia depth-first** (da mesma forma que a função de procura apresentada é conhecida como **procura depth-first**).

```
int travessiaDF (Graph g, int o){
  int visitados [NV]; int i;
  for (i=0;i<NV;i++)
    visitados[i] = 0;
  return (DFRec (g,o,visitados));
}

int DFRec (Graph g, int o, int v[]){
  int count = 1;
  EList it;
  v[o] = 1;
  for (it = g[o]; it != NULL; it = it->next)
      if (! v[it->dest])
        count += DFRec (g,it->dest,v);
  return count;
}
```

A principal diferença entre as funções de procura e de travessia apresentadas consiste no teste de paragem: enquanto que na procura, a função termina mal encontre o destino pretendido, na travessia são esgotados todos os destinos possíveis e, por isso são visitados todos os nodos alcançáveis a partir do vértice inicial.

Exemplo 8 Um grafo não orientado diz-se **ligado** sse existe pelo menos caminho entre qualquer par de vértices.
Uma **componente ligada** de um grafo não orientado G não é mais do que um sub-grafo ligado de G.
A função `DFRec` acima pode ser usada para determinar o tamanho (número de vértices) da maior componente ligada de um grafo não orientado.

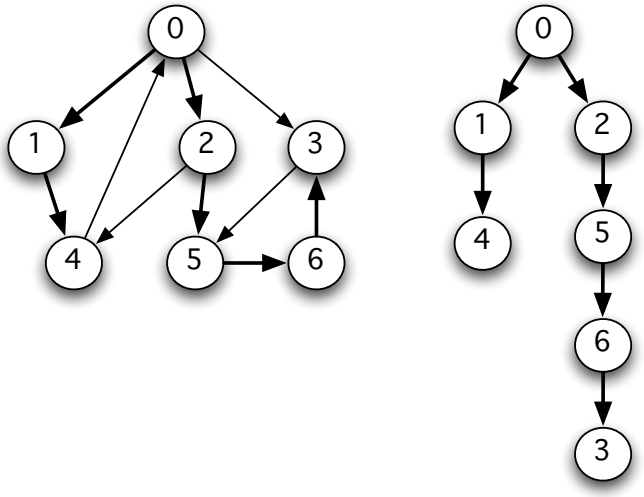
```
int maiorCL (Graph g) {
  int visitados [NV]; int i;
  int max=0, c;
  for (i=0;i<NV;i++)
    visitados[i] = 0;

  for (i=0; i<NV; i++)
    if (visitados [i] == 0){
      c = DFRec (g,i,visitados);
      if (c>max) max=c;
    }

  return (max);
}
```

Mais uma vez estamos a tirar partido do mecanismo de passagem de arrays como argumentos do C, em que as alterações feitas a tais arrays persistem após a invocação da função.

Exercício 6 Um grafo não orientado diz-se **bi-partido** sse é possível particionar o conjunto de vértices V em dois conjuntos disjuntos V_0 e V_1 tais que todas as arestas do grafo têm uma das extremidades num dos conjuntos e a outra no outro.
Defina uma função `int biPartite (Graph g)` que testa se um grafo não orientado é bi-partido.

A função de travessia apresentada dá como resultado o número de vértices alcançáveis. Há ainda outro resultado que é fácil retornar: o conjunto dos vértices visitados. Essa informação está presente no array (local) `visitados` após a invocação de `DFRec`: os vértices alcançáveis são exactamente aqueles para os quais o array `visitados` tem o valor `1`.
Para que essa informação seja visível do exterior será necessário que o dito array deixe de ser local e passe a ser passado como argumento à função de travessia.
Uma informação adicional que pode ser obtida com uma pequena alteração da função de travessia diz respeito às arestas usadas nessa travessia. A cada vértice visitado está associado o caminho que foi usado para lá chegar a partir da origem. Todos esses caminhos, com uma origem em comum formam uma árvore.

Exemplo 9 Se no grafo do exemplo anterior fizermos uma travessia depth-first a partir do vértice 0, as arestas usadas são as que se mostram a *bold*. Do lado direito podemos ver essas arestas como uma árvore cuja raíz é o vértice 0.



Uma forma usual de representar estas árvores baseia-se numa propriedade simples das árvores: cada nodo da árvore (com excepção para a raíz) tem **exactamente um** antecessor. Dessa forma, a informação pode ser representada com um array em que na posição i se encontra o antecessor do vértice i.

Exemplo 9 (continuação) A árvore apresentada no exemplo anterior pode ser representada pelo seguinte array de antecessores:

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

O valor -1 como antecessor do vértice 0 assinala que a raíz da árvore é o vértice 0.

A função seguinte de travessia depth-first incorpora a construção desta árvore na definição apresentada atrás.

```
int travessiaDF2 (Graph g, int o, int p[]){
  int visitados [NV]; int i;
  for (i=0;i<NV;i++){
    visitados[i] = 0;
    p[i] = -2;
  }
  p[o] = -1;
  return (DFRec2 (g,o,visitados,p));
}

int DFRec2 (Graph g, int o, int v[], int p[]){
  int count = 1;
  EList it;
  v[o] = 1;
  for (it = g[o]; it != NULL; it = it->next)
      if (! v[it->dest]) {
```

```
        p[it->dest] = o;
        count += DFRec2 (g,it->dest,v,p);
      }
  return count;
}
```

O uso do array visitados não é estritamente necessário. Pode-se verificar que `v[i] == 0` sse `p[i] == -2`. Desta forma a definição acima pode ser reescrita.

```
int travessiaDF2 (Graph g, int o, int p[]){
  int i;
  for (i=0;i<NV;i++)
    p[i] = -2;
  p[o] = -1;
  return (DFRec2 (g,o,p));
}

int DFRec2 (Graph g, int o, int p[]){
  int count = 1;
  EList it;
  for (it = g[o]; it != NULL; it = it->next)
      if (p[it->dest] == -2) {
        p[it->dest] = o;
        count += DFRec2 (g,it->dest,p);
      }
  return count;
}
```

3.2 Breadth First -> Travessia em largura

A estratégia de travessia apresentada acima usa a recursividade como mecanismo de navegação sobre o grafo. Uma outra forma de fazer uma travessia passa pela manutenção explícita da informação sobre quais os vértices que devem ainda ser visitados.
Uma forma de armazenar tal informação é usando uma fila (queue) onde os vários vértices candidatos a serem visitados vão sendo adicionados. O processo termina quando já não houver vértices nessa fila.
Esta estratégia é conhecida como **breadth-first**.

```
int travessiaBF (Graph g, int o){
   int visitados [NV];
   Queue q;
   int count=0;
   EList it;

   q = emptyQueue();
   enqueue (q,o);
   visitados[o] = 1;
```

```
    while (! empty (q)) {
     o = dequeue (q);
     count++;
     for (it=g[o]; it != NULL; it=it->next)
       if (! visitados[it->dest])
         enqueue (q,it->dest)
    }
    return count;
}
```

A queue usada nesta função tem algumas características que nos permitem uma implementação muito simples: (1) nunca tem mais do que `NV` elementos e (2) nunca um vértice pode entrar na queue mais do que uma vez.
Assim sendo podemos implementar esta queue com um array de (`NV`) inteiros (vértices) juntamente com dois índices `inicio` e `fim` que marcam as extremidades da queue.

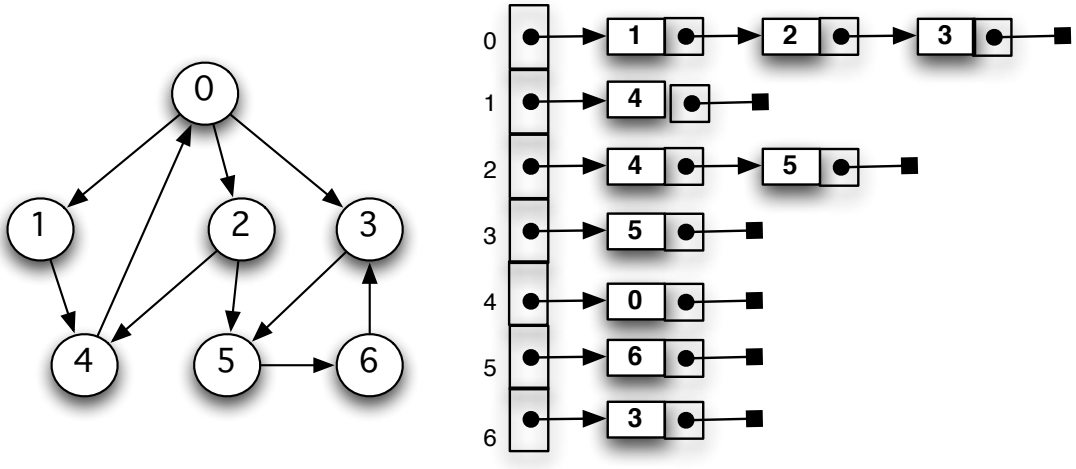
```
int travessiaBF (Graph g, int o){
    int visitados [NV];
    int q[NV]; int inicio, fim; //Queue q;
    int count=0;
    EList it;

    inicio = fim = 0;              //q = emptyQueue();
    q[fim++] = o;                  //enqueue (q,o);
    visitados[o] = 1;
    while (inicio < fim) {         //! empty (q)
     o = q[inicio++];              //o = dequeue (q);
     count++;
     for (it=g[o]; it != NULL; it=it->next)
       if (! visitados[it->dest])
         q[fim++] = it->dest;      //enqueue (q,it->dest)
    }
    return count;
}
```

Tal como fizemos com a travessia em profundidade, podemos construir a árvore associada à travessia usando um array de antecessores que é passado como argumento (e que servirá ainda para desempenhar o papel do array `visitados`).

```
int travessiaBFTree (Graph g, int o, int ant[]){
    int q[NV]; int inicio, fim, i;
    int count=0;
    EList it;

    for(i=0; i<NV; i++)
      ant[i] = -2;
    inicio = fim = 0;
    q[fim++] = o;
```

```
    ant[o] = -1;
    while (inicio < fim) {
     o = q[inicio++];
     count++;
     for (it=g[o]; it != NULL; it=it->next)
       if (ant[it->dest] == -2){
         ant[it->dest] = o;
         q[fim++] = it->dest;
       }
    }
    return count;
}
```

Exemplo 10 Vamos exemplificar o funcionamento da função de travessia em largura usando para isso o grafo apresentado no exemplo 7 a partir do vértice `0`. Para isso vamos apresentar os estados do array `ant` e da queue `q` para cada iteração do ciclo `while`.



1. inicio = 0; fim = 1; count = 0;

   q = | 0 | ? | ? | ? | ? | ? | ? | ; ant = | -1 | -2 | -2 | -2 | -2 | -2 | -2 |

   O vértice que será removido da queue será `0 = q[inicio]`. Serão acrescentados à queue os seus sucessores não visitados (1, 2 e 3).

2. inicio = 1; fim = 4; count = 1;

   q = | 0 | **1** | **2** | **3** | ? | ? | ? | ; ant = | -1 | 0 | 0 | 0 | -2 | -2 | -2 |

   O vértice que será removido da queue será `1 = q[inicio]`. Será acrescentado à queue o seu sucessor não visitado (4).

3. inicio = 2; fim = 5; count = 2;

   q = | 0 | 1 | **2** | **3** | **4** | ? | ? | ; ant = | -1 | 0 | 0 | 0 | 1 | -2 | -2 |

   O vértice que será removido da queue será `2 = q[inicio]`. Será acrescentado à queue o seu sucessor não visitado (5).

4. inicio = 3; fim = 6; count = 3;

	0	1	2	3	4	5	6
q =	0	1	2	**3**	**4**	**5**	?

; ant =

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

O vértice que será removido da queue será 3 = `q[inicio]`. Não será acrescentado à queue nenhum vértice (o sucessor de 3 é apenas 5 que já foi acrescentado à queue.

5. inicio = 4; fim = 6; count = 4;

	0	1	2	3	4	5	6
q =	0	1	2	3	**4**	**5**	?

; ant =

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

O vértice que será removido da queue será 4 = `q[inicio]`. Não será acrescentado à queue nenhum vértice.

6. inicio = 5; fim = 6; count = 5;

	0	1	2	3	4	5	6
q =	0	1	2	3	4	**5**	?

; ant =

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

O vértice que será removido da queue será 5 = `q[inicio]`. Será acrescentado à queue o seu sucessor não visitado (6).

7. inicio = 6; fim = 7; count = 6;

	0	1	2	3	4	5	6
q =	0	1	2	3	4	5	**6**

; ant =

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

O vértice que será removido da queue será 6 = `q[inicio]`. Não será acrescentado à queue nenhum vértice.

8. inicio = 7; fim = 7; count = 7;

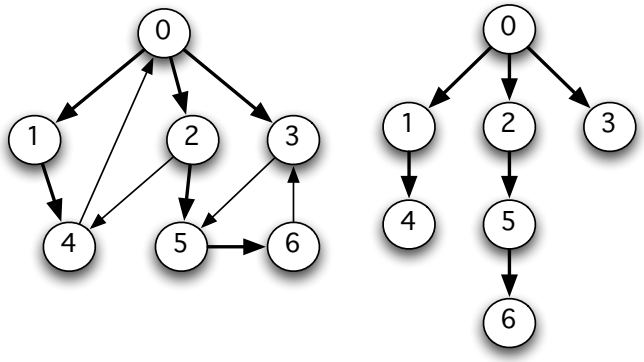
	0	1	2	3	4	5	6
q =	0	1	2	3	4	5	6

; ant =

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

A queue está vazia e por isso o ciclo termina.

As arestas usadas são as que a seguir se apresentam a *bold*. Do lado direito podemos ver essas arestas como uma árvore cuja raíz é o vértice 0 e que se encontra armazenada no array `ant`.



Esta estratégia de percorrer todos os vértices alcançáveis a partir de um dado vértice pode ser generalizada permitindo resolver outros problemas de grafos.
Assim, o processo resume-se a manter o conjunto de vértices do grafo particionado em três conjuntos disjuntos:

- visitados V_0
- em visita V_1

- não visitados V_2

Com a seguinte propriedade (invariante):

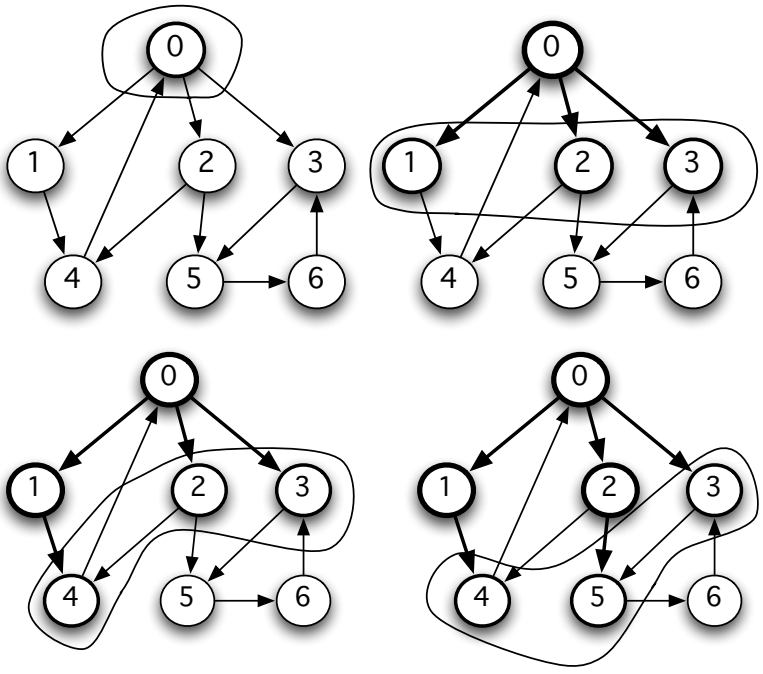
> os vértices em V_1 são todos os vértices que não pertencem a V_0 mas que têm uma aresta que os liga a um elemento de V_0.

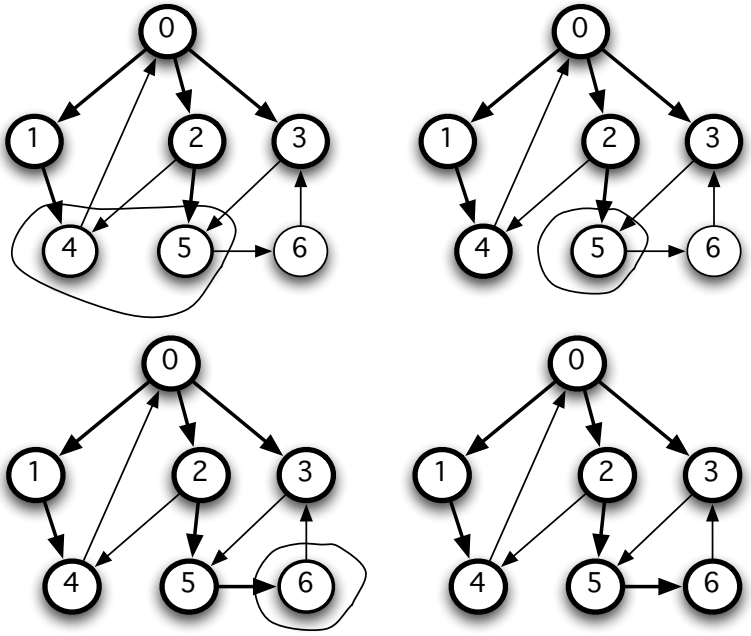
O conjunto V_1 é muitas vezes referido como a **orla** (ou **fronteira**).
Os algoritmos baseados neste conceito de orla evoluem então da seguinte forma:

> `enquanto o conjunto` V_1 `não for vazio`
>
> - `escolher um vértice` v `de` V_1
> - `adicionar` v `a` V_0 `removendo-o de` V_1
> - `para todos os vértices` x `adjacentes a` v `que estejam em` V_2
>   - `adicionar` x `a` V_1 `removendo-o de` V_2

No caso da travessia breadth-first o conjunto V_1 (orla) é representado pela queue `q` e o conjunto V_2 corresponde aos vértices `i` para os quais `ant[i] == -2`.

Exemplo 10 (continuação) Abaixo apresenta-se a evolução do algoritmo de travessia breadth-first em termos do estado da orla.





A estratégia abstrata apresentada garante-nos algumas propriedades importantes:

- **Terminação** Uma vez que o conjunto de vértices é finito e cada iteração do ciclo faz com que o número de vertices visitados (V_0) aumente sempre (cada iteração acrescenta a V_0 um vértice que lá não estava).
- **Exaustivo**, i.e., todos os vértices alcançáveis são visitados.

No caso da travessia breadth-first a orla (V_1) é implementada como uma queue: a operação de escolha de um vértice da orla é implementada com uma operação de dequeue. Isto significa que os vértices são retirados por ordem *cronológica* (*first-in-first-out*). Este facto garante duas outras propriedades adicionais.

- Os vértices são visitados por ordem crescente da sua distância (número de arestas do caminho) à origem.
- Os caminhos encontrados (do vértice inicial a todos os outros vértices alcançáveis) são os mais curtos (em termos do número de vértices do caminho).

Exercício 7 Defina uma função `int adist (Graph g, int o, int k)` que calcula quantos vértices do grafo não pesado `g` estão à distância `k` do vértice `o`.

Exercício 8 Considere que se usa um grafo pesado e orientado para representar uma rede de distribuição de água. Os vértices correspondem às bifurcação enquanto que os pesos das arestas representam a secção do tubo que liga as bifurcações.
Defina uma função `int haLigacao (Graph g, int o, int d, int sec)` que determina se há uma ligação do vértice `o` até `d` usando apenas tubos com secção superior a `sec`.

Exercício 9 Considere um grafo com 101 vértices (0 a 100) onde existe uma aresta do vértice `a` para o vértice `b` sse `b` é um múltiplo de `a` (i.e., `a` é um divisor de `b`). Por exemplo, os sucessores do vértice `15` são `0`, `15`, `30`, `45`, `60`, `75` e `90`. Apresente a árvore produzida pelas travessias breadth-first e depth-first quando invocadas neste grafo a partir do vértice `10`.

Exercício 10 Defina uma função `int caminhoValido (Graph g, int o, char p[])` que, dado um grafo pesado `g` cujos pesos são caracteres, um vértice `o` e uma string `p`, determina se existe no grafo `g` um caminho a partir do vértice `o` cujos pesos correspondam à string `p`.

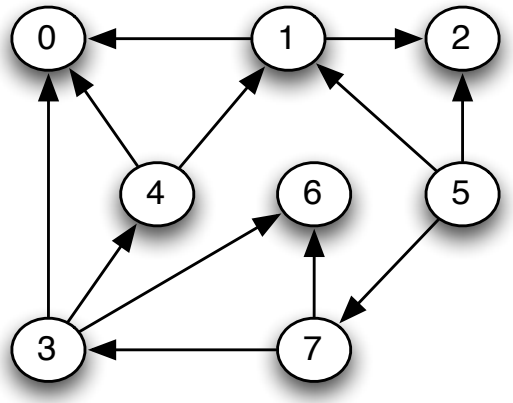
3.3 Ordenação topológica

Dado um grafo orientado e acíclico $g = (V, E)$, uma sequência $s = \langle v_1, v_2 \dots v_n$ dos vértices do grafo diz-se uma **ordenação topológica** de g se e só se, para cada aresta (o, d) o vértice o aparece na sequência **antes** do vértice d.
O cálculo de uma ordenação topológica pode ser feito baseado numa das travessias apresentadas atrás.

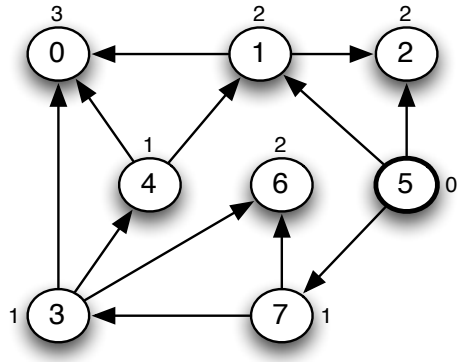
3.3.1 Algoritmo de Kahn

A estratégia usada por este algoritmo baseia-se em construir a sequência usando em cada momento a informação sobre quantos antecessores de cada vértice é que ainda não apareceram na dita sequência.
O primeiro passo do algoritmo inicializa essa informação, contando, para cada vértice, o número de antessores (*indegree*).
Este passo permite-nos determinar quais o(s) vértices que podem iniciar a ordenação topológica – aqueles que não têm antecessores.
O algoritmo prossegue escolhendo um vértice que possa aparecer na sequência e marcando todos os seus sucessores como tendo menos um antecessor por visitar.
Para optimizar o processo de escolha de um vértice que possa aparecer na sequência, os vértices candidatos são armazenados, tipicamente numa queue.

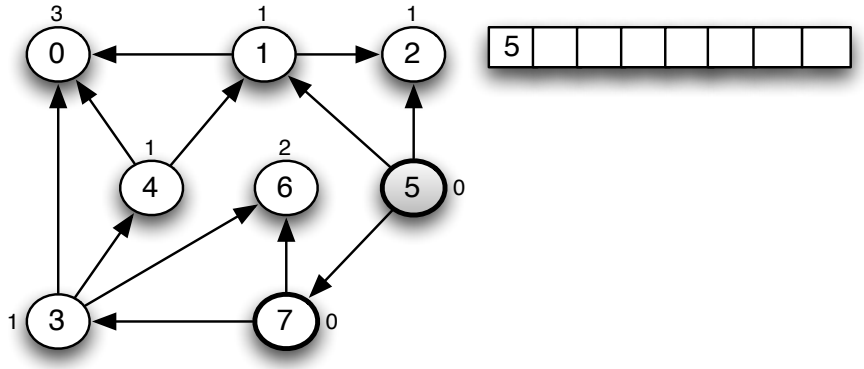
Exemplo 11 Considere-se o seguinte grafo orientado



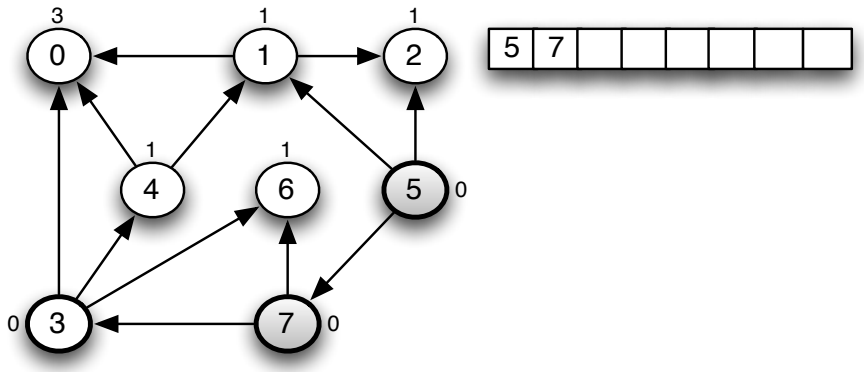
0. A inicialização calcula o grau de entrada de cada vértice. Daqui resulta que foi encontrado um vértice sem antecessores.



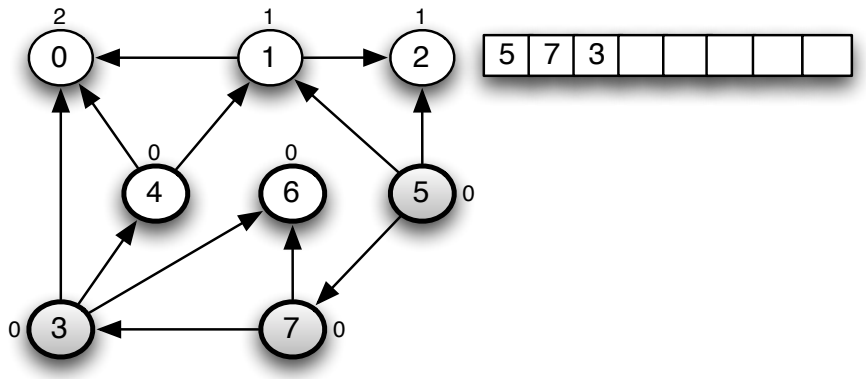
1. A inclusão do vértice 5 vai decrementar o número de antecessores não visitados de 1, 2 e 7. Este último passa a ter 0 como número de antecessores a visitar e por isso será o próximo a ser incuído na ordenação.



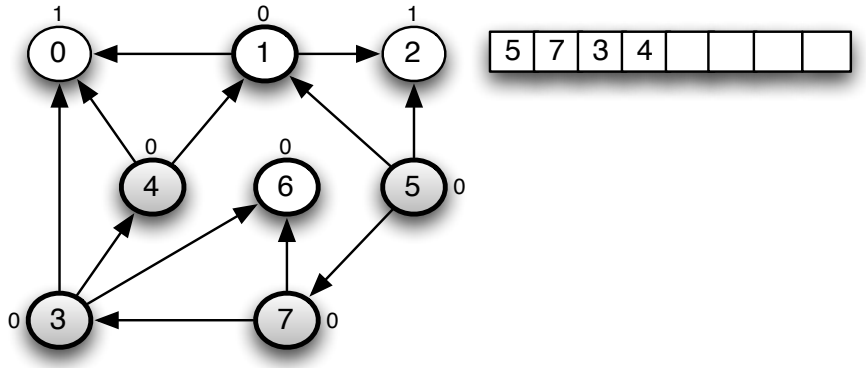
2. A inclusão do vértice 7 vai decrementar o número de antecessores não visitados de 6 e 3. Este último passa a ter 0 como número de antecessores a visitar e por isso será o próximo a ser incuído na ordenação.



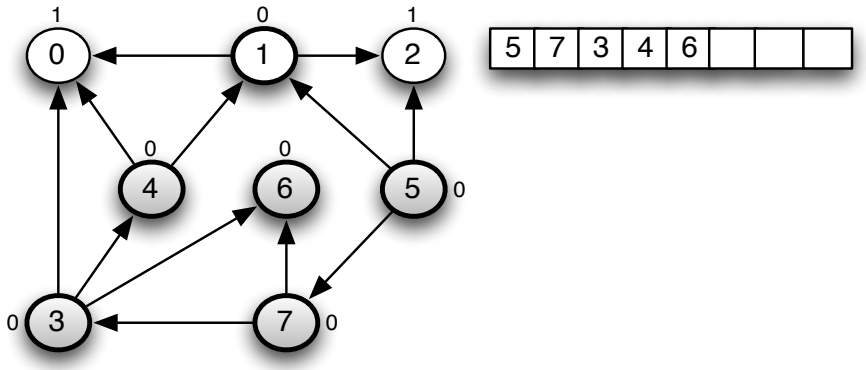
3. A inclusão do vértice 3 vai decrementar o número de antecessores não visitados de 0, 4 e 6. Estes dois últimos passam a ter 0 como número de antecessores a visitar e por isso podem ser incuídos na ordenação.



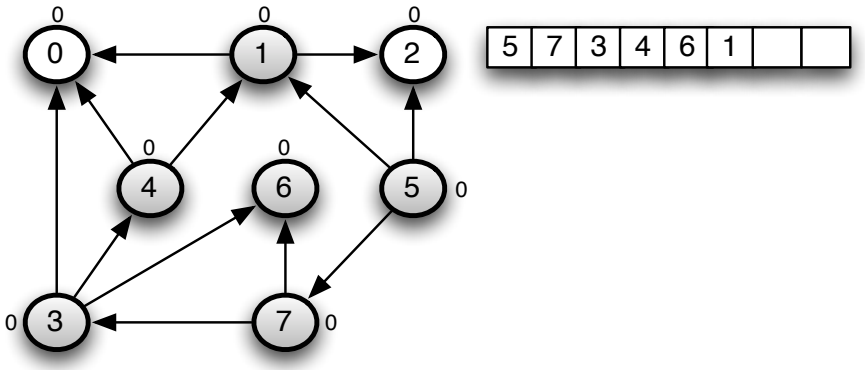
4. A inclusão do vértice 4 vai decrementar o número de antecessores não visitados de 0 e 1. Este último passa a ter 0 como número de antecessores a visitar e por isso pode ser incuído na ordenação.



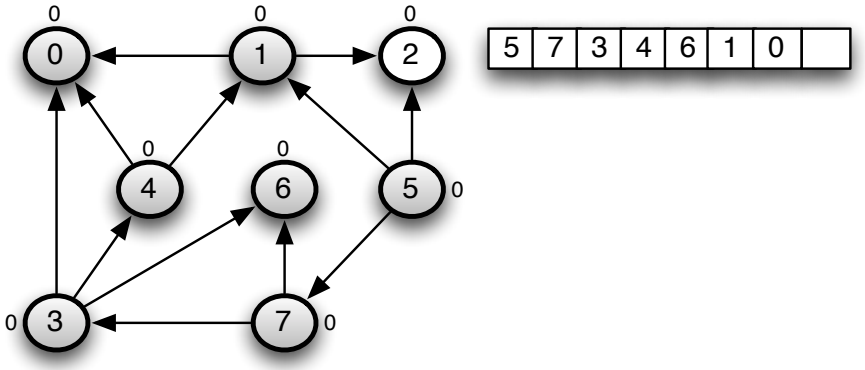
5. A inclusão do vértice 6 não altera o número de antecessores não visitados de qualquer vértice.



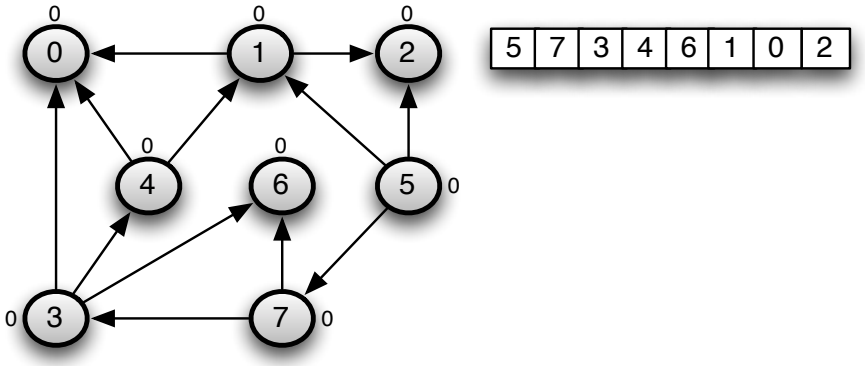
6. A inclusão do vértice 1 vai decrementar o número de antecessores não visitados de 0 e 2, que passam a ter 0 como número de antecessores a visitar.



7. A inclusão do vértice 0 não altera o número de antecessores não visitados de qualquer vértice.



8. A inclusão do vértice 2 não altera o número de antecessores não visitados de qualquer vértice.



9. O algoritmo termina pois já não há vértices disponíveis para acrescentar à sequência.

Vejamos então a codificação das várias partes deste algoritmo.

- O passo inicial corresponde a calcularmos o grau de entrada de cada vértice.

```
for (v=0; v<NV; v++) nant[v]=0;
for (v=0; v<NV; v++)
    for (it=g[v]; it!=NULL; it=it->next)
      nant[it->dest]++;
```

- depois de calculado o grau de entrada, podemos inicializar a queue com todos os vértices sem antecessores.

```
inicio=fim=0; // queue vazia
for(v=0; v<NV; v++)
  if (nant[v]==0)
    q[fim++]=v; // enqueue (v)
```

- o algoritmo prossegue retirando por cada passo um elemento da queue e actualizando os numeros de antecessores de cada vértice afectado.

```
i=0;
while (inicio<fim) { // queue nao vazia
  v = q[inicio++]; //dequeue
  seq[i++] = v;
  for (it=g[v]; it!=NULL; it=it->next){
      nant[it->dest]--;
      if (nant[it->dest] == 0)
          q[fim++] = it->dest; // enqueue
  }
```

Uma optimização que é costume fazer baseia-se na observação de que nesta última fase as variáveis `inicio` e `i` têm o mesmo valor, pelo o array `seq` pode ser usado para guardar a queue.

```
int khanTS (Graph g, int seq[]){
  EList it;
  int v;
  int inicio, fim; // queue
  int nant[NV]; // numero de antecessores
  // calculo do grau de entrada
  for (v=0; v<NV; v++) nant[v]=0;
  for (v=0; v<NV; v++)
     for (it=g[v]; it!=NULL; it=it->next)
       nant[it->dest]++;
  // inicializacao da queue
  inicio=fim=0; // queue vazia
  for(v=0; v<NV; v++)
    if (nant[v]==0)
      seq[fim++]=v; // enqueue (v)
  // construcao da ordenacao
  while (inicio<fim) { // queue nao vazia
    v = seq[inicio++]; //dequeue
    for (it=g[v]; it!=NULL; it=it->next){
       nant[it->dest]--;
       if (nant[it->dest] == 0)
          seq[fim++] = it->dest; // enqueue
```

```
    }
  }
  return inicio;
}
```

Esta função retorna o valor da variàvel `inicio` que, como já dissemos, corresponde ao número de elementos colocados na sequência. Em caso de sucesso, i.e., se fôr possível obter uma ordenação topológica do grafo este valor será igual ao número de vértices do grafo.
Mas nem sempre é possível obter tal ordenação. E isso acontece exactamente quando o grafo tiver pelo menos um ciclo. Nesse caso, os vértices que compõe os ciclos e todos os seus alcançáveis não aparecerão na sequência (o número de antecessores nunca chega a ser 0).
Esta observação é no entanto útil para outro propósito: uma forma de determinar se um grafo é ou não cíclico consiste em calcular o tamanho da sequência retornada por esta função e determinar se esse número correponde ao número de vértices do grafo.

3.3.2 Algoritmo de Tarjan

O algoritmo apresentado atrás tem algumas semelhanças com uma travessia breadth-first: é controlado por uma queue com os vértices candidatos a serem incluídos na sequência.
Uma forma alternativa de fazer uma ordenação topológica é baseada na travessia depth-first. Há no entanto alguns problemas que têm que ser tratados para que tal estratégia possa produzir o efeito desejado.
Em primeiro lugar deve ser possível determinar se a ordenação é possível, i.e., se o grafo tem ou não ciclos. Em segundo lugar devemos ter em atenção que à partida não sabemos se um dado vértice pode ou não aparecer no início da sequência.
O segundo problema é relativamente fácil de resolver se começarmos por calcular a sequência por ordem inversa: a inclusão de um vértice em tal sequência (invertida) far-se-á **depois** de incluídos todos os vértices alcançáveis a partir dele. Além disso, uma travessia a partir de um único vértice pode não ser suficiente para percorrer todo o grafo, pelo que devem ser iniciadas travessias suficientes para garantir que todo o grafo é visitado.
A forma de resolver o primeiro problema consiste em incluir no processo a informação sobre os vértices que estão a ser visitados.
O algoritmo de travessia depth-first apresentado atrás usa um array `visitado` em que cada posição determina se já foi ou não iniciada uma travessia a partir desse vértice. A primeira acção que a função faz é colocar `1` (`true`) na posição correspondente ao vértice a ser visitado.
Em vez disso vamos usar um array `estado` em que cada posição `i` pode estar um de três valores possíveis:

- `0` se **nunca foi iniciada** uma travessia a partir do vértice `i`. Dizemos que o vértice `i` está **não visitado**.
- `1` se **já foi iniciada** uma travessia a partir do vértice `i` mas **ainda não terminou**. Dizemos que o vértice `i` está **em visita**.

- `2` se **já terminou** uma travessia iniciada no vértice `i`. Dizemos que o vértice `i` está **visitado**.

Esta alteração pode ser feita facilmente: a actualização do estado de cada vértice deve ser feita (1) no início de cada travessia (a primeira instrução) o vértice deve ser marcado como **em visita** e (2) no final (última instrução) o vértice deve ser marcado como **visitado** e adicionado ao final da sequência.
Por uma questão visual é ainda costume caracterizar estes estados dos vértices usando o seguinte *código de cores*:

- `Branco` para os vértices não visitados,
- `Preto` para os vértices visitados,
- `Cinzento` para os vértices em visita.

Antes de prosseguirmos na codificação desta solução convém atentar numa propriedade (invariante) importante sobre esta transformação:

> Em cada estado do processo existe um caminho que liga todos os vértices marcados como **em visita**.

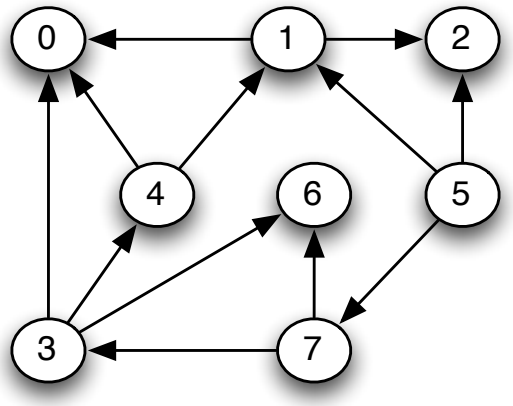
Esta observação é crucial para a deterinação da existência de ciclos (e consequente insucesso da ordenação topológica): se nos depararmos com um sucessor do vértice em análise para um outro cujo estado seja **em visita** foi detectado um ciclo.

```
int tarjanTS (Graph g, int seq[]){
  int color[NV];
  int v,j,t,r;
  for (v=0;v<NV;v++)
      color[v] = 0; // WHITE
  r=1; t=0;
  for (v=0; v<NV; v++)
    if (color[v] == 0)
      t+=dfirstTopSort (g,v,color,&r,seq+t);
  //inverter sequencia
  for (v=0, j=t; v<j; v++,j--)
     swap (seq,v,j);
  return r;
}
```

O primeiro passo desta função consiste em inicializar o estado de cada vértice (array `colors`) como não visitado (`0`).
De seguida vai-se invocando a função de travessia (`dfirstTopSort`) até que todos os vértices passem a visitados. A função `dfirstTopSort` retorna o número de vértices que foram colocados em `seq`. Daí que, de uma invocação para a outra tenha que lhe ser passado como argumento uma secção do array diferente (os primeiros `t` elementos já estão preenchidos).
Finalmente a função `tarjanTS` inverte a ordem da sequência produzida.
O valor de retorno da função (valor da variável `r`) é inicializado a `1` e eventualmente mofificado pela função `dfirstTopSort` caso seja detectado um ciclo.

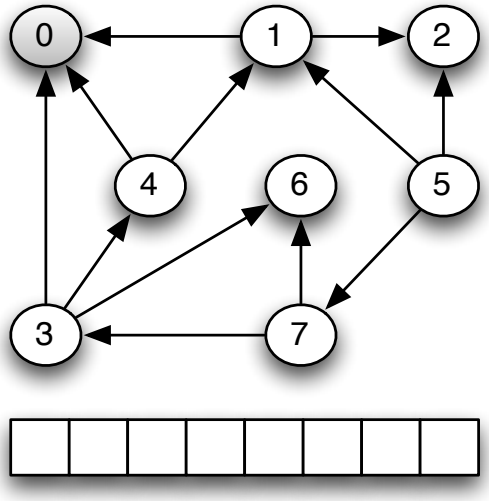
```
int dfirstTopSort (Graph g, int o, int color[],
                   int *success, int seq[]) {
    int t;
    EList it;
    color[o]=1; // GREY
    t=0;
    for (it=g[o]; it!=NULL; it=it->next)
        if (color[it->dest] == 0) // WHITE
            t+=dfirstTopSort(g,it->dest,color,success,seq+t);
        else if (color[it->dest]==1) //GREY
            *success=0;
    seq[t++]=o;
    color[0]=2; //BLACK
    return t;
}
```

Exemplo 12 Vejamos então a evolução deste algoritmo para o grafo usado no exemplo anterior.

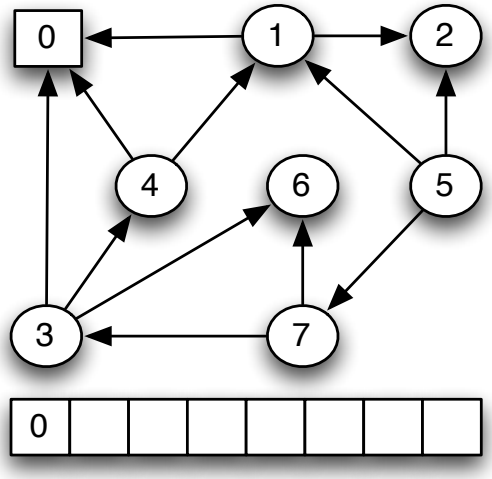


Vamos analisar os estados dos vários vértices bem como do array seq. Para facilitar a leitura vamos representar o estado dos vértices usando um círculo sombreado para representar os vértices em visita e um quadrado para representar os vértices já visitados. Os vértices não visitados serão representados por um círculo não sombreado.

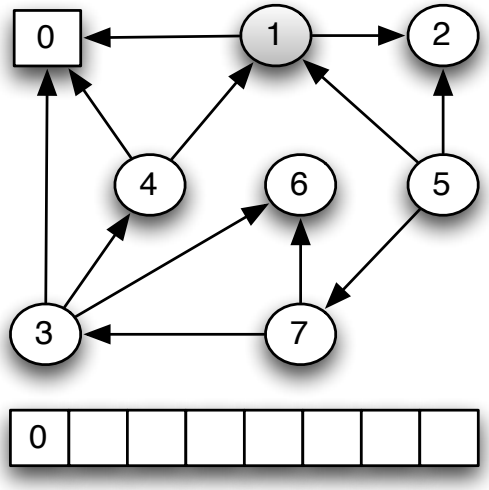
1: Inicia-se uma travessia a partir do vértice 0. Este é marcado como em visita.



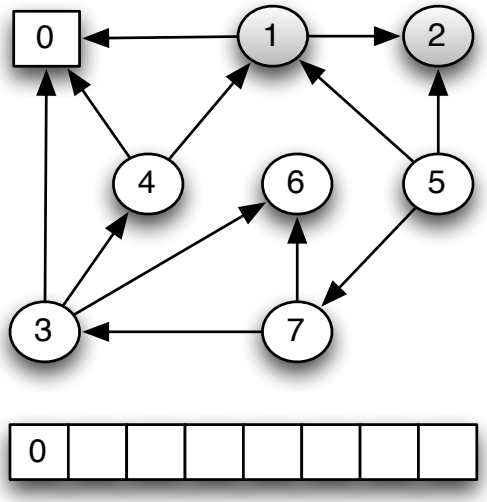
2: Como o vértice 0 não tem sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



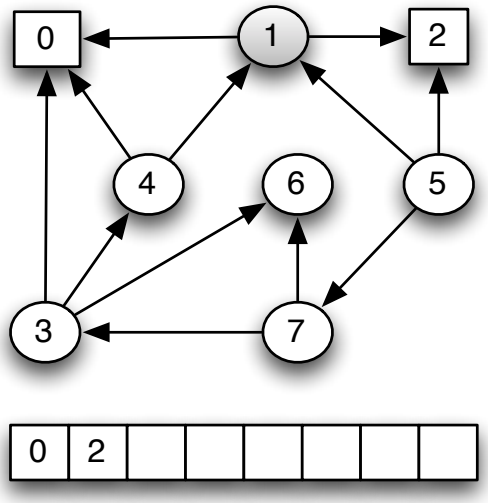
3: Inicia-se uma travessia a partir do vértice 1. Este é marcado como em visita.



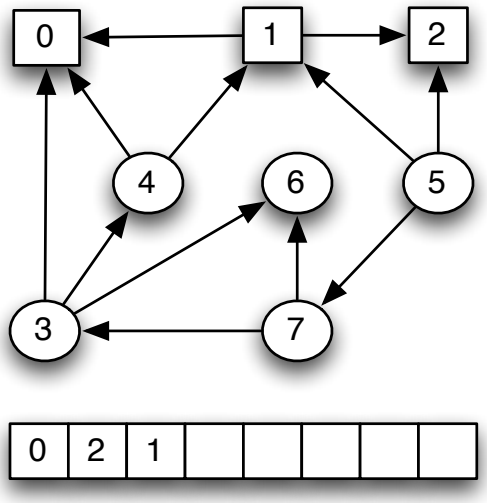
4: Inicia-se uma travessia a partir do vértice 2. Este é marcado como em visita.



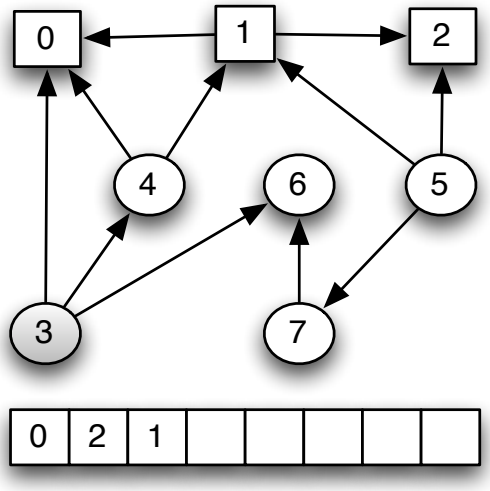
5: Como o vértice 2 não tem sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



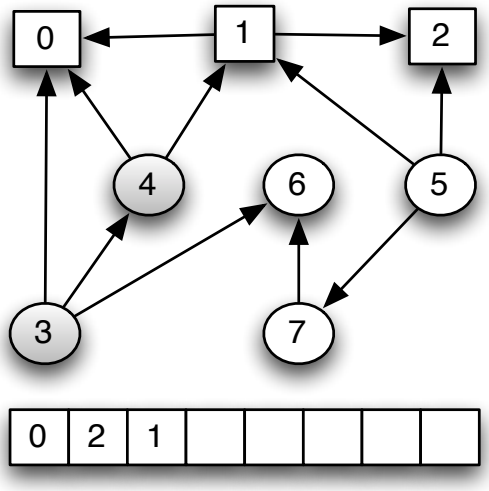
6: Como o vértice 1 não tem mais sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



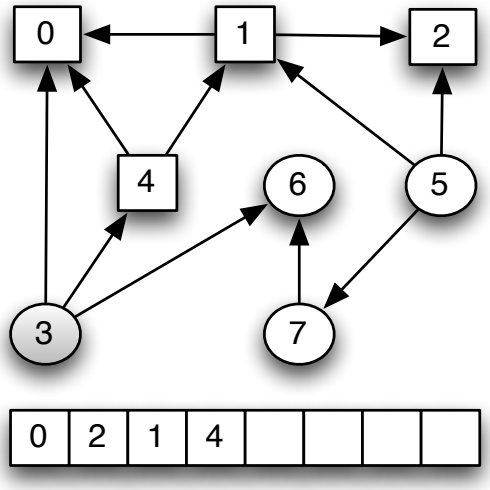
7: Inicia-se uma travessia a partir do vértice 3. Este é marcado como em visita.



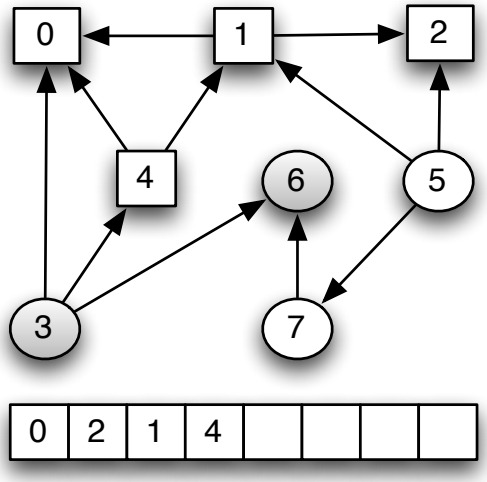
8: Inicia-se uma travessia a partir do vértice 4. Este é marcado como em visita.



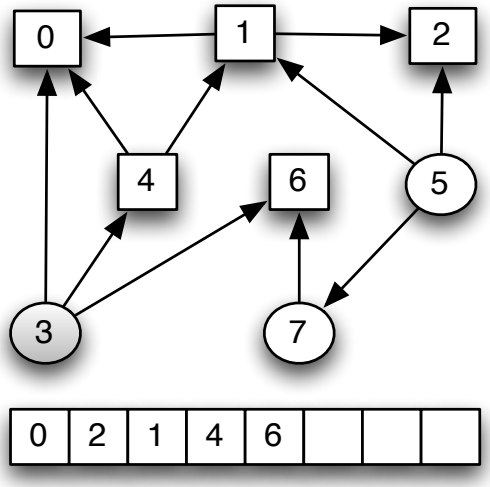
9: Como o vértice 4 não tem sucessores não visitados termina a visita: o vértice é marcado como visitado e adicionado à sequência.



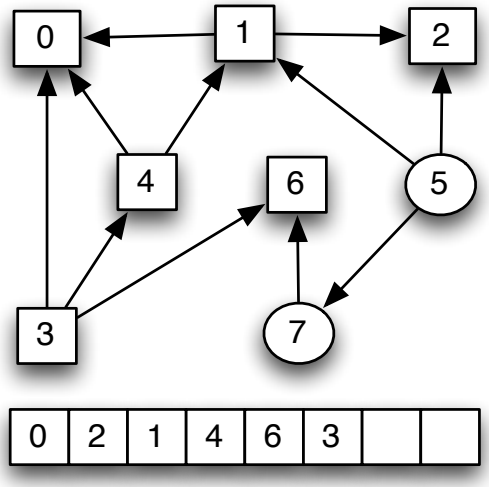
10: Inicia-se uma travessia a partir do vértice 6. Este é marcado como em visita.



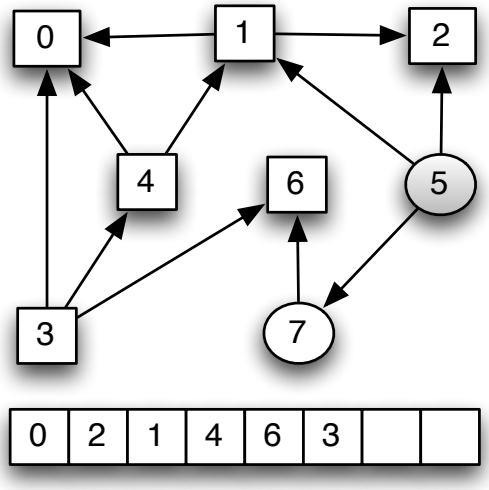
11: Como o vértice 6 não tem sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



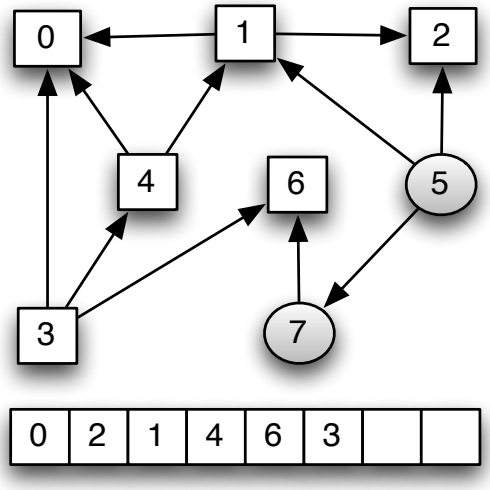
12: Como o vértice 3 não tem mais sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



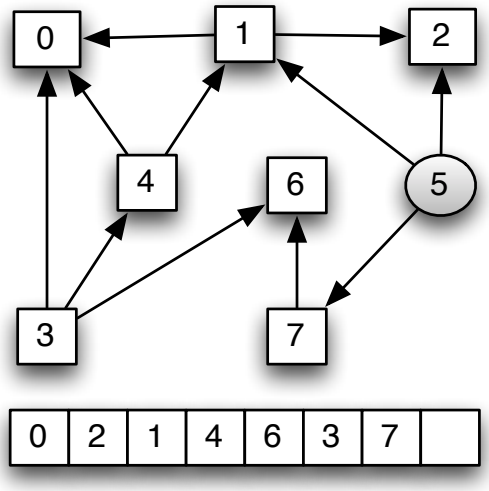
13: Inicia-se uma travessia a partir do vértice 5. Este é marcado como em visita.



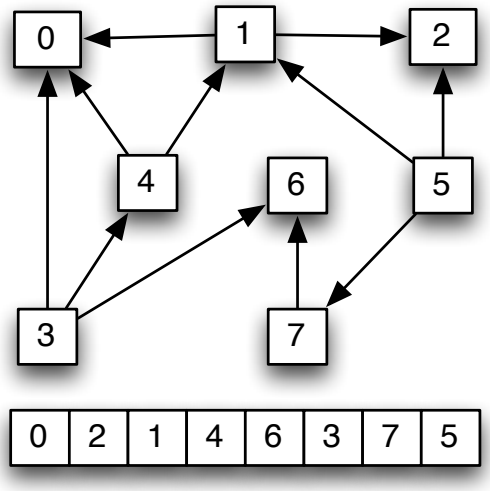
14: Inicia-se uma travessia a partir do vértice 7. Este é marcado como em visita.



15: Como o vértice 7 não tem sucessores não visitados termina a visita: o vértice é marcado como visitado e adicionado à sequência.

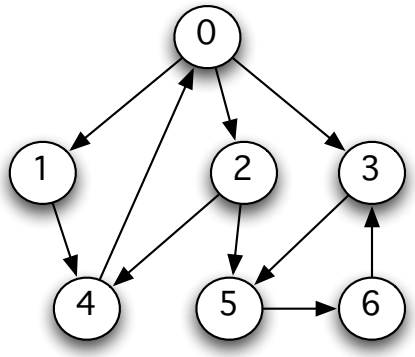


16: Como o vértice 5 não tem sucessores não visitados termina a visita: o vértice é marcado como visitado e adicionado à sequência.



Uma diferença funcional importante entre este algoritmo e o algoritmo de Kahn é que neste casoé produzida uma sequência com todos os vértices do grafo, mesmo quando o grafo é cíclico. A informação sobre a validade de tal sequência como uma ordenação topológica pode ser obtida a partir do valor (lógico) que a função retorna.

Exercício 11 Considere o grafo g ao lado tal como apresentado no exemplo 10.
Diga qual o conteúdo do array `seq` após a invocação de `tarjanTS (g,seq)`



3.4 Componentes fortemente ligadas

Um grafo orientado $G = (V, E)$ diz-se fortemente ligado sse para qualquer par de vértices $(a, b) \in V^2$ existe um caminho em G de a para b.
Uma componente fortemente ligada de um grafo é um subgrafo fortemente ligado.
O problema que vamos resolver nesta secção é a determinação das componentes fortemente ligadas de um grafo orientado.
O algoritmo que vamos estudar, da autoria de Tarjan, é uma adaptação da travessia depth-first apresentada atrás e baseia-se na seguinte propriedade:

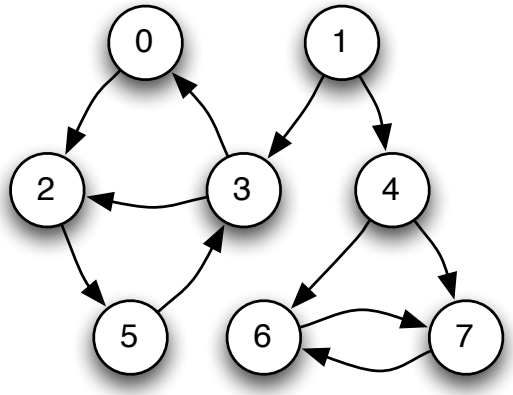
> Dois vértices u e v estão na mesma componente ligada se e só se $adj(u) = adj(v)$.

O cálculo de uma ordenação topológica a partir de uma travessia depth-first foi possível graças a aumentarmos a informação disponível sobre cada vértice ao longo do processo. No algoritmo original apenas precisámos de saber se um dado vértice estava ou não visitado. No cálculo da ordenação topológica passámos a incluir a informação sobre o início e o fim da visita de cada vértice.
A primeira ideia chave para conseguir identificar as componentes fortemente ligadas de um grafo consiste em atribuír a cada nodo um número sequencial que marca o início da visita de cada nodo. Além disso, calcula-se para cada vértice, o menor dos números de entrada de todos os seus alcançáveis. Uma vez que todos os vértices de uma componente fortemente ligada têm o mesmo conjunto de vértices alcançáveis, este valor mínimo vai ser comum.
A segunda ideia chave deste algoritmo será usada para determinar de uma forma eficiente os vértices que têm essa componente em comum.
A primeira versão que vamos apresentar apenas explora a primeira ideia e por isso calcula apenas quantas componentes fortemente ligadas existem num dado grafo.
Tal como acontecia com a travessia depth-first, vai ter que existir um primeiro passo (não recursivo) de inicialização das várias variáveis de estado: `color` (estado de cada vértice) e `tStamp` (número de ordem do início das travessias).
Além disso, e de forma a percorrer todos os vértices do grafo, esta função deverá fazer tantas invocações à função recoursiva quantas sejam necessárias.

```
int tarjanSCC (Graph g){
```

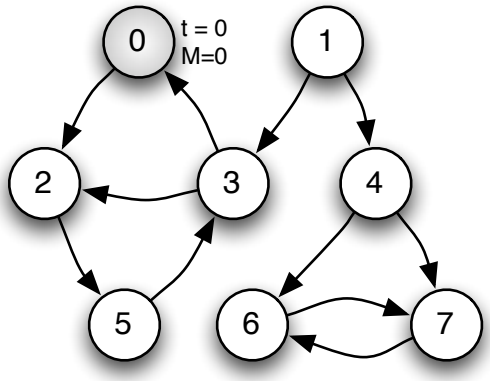
```
  int color[NV], tIn[NV], mtIn[NV];
  int i; int tStamp = 0;
  int r=0;

  for (i=0; i<NV; i++)
      color [i] = 0; // WHITE

  for (i=0;i<NV;i++)
      if (color[i]==0) // WHITE
          r+=sccRec (g, &tStamp, i, color, tIn, mtIn);
  return r;
}

int sccRec (Graph g, int *tStamp, int v,
            int color[], int t[], int M[]){
    EList it;
    int r=0;
    color[v]= 1; //GREY
    M[v]= t[v]= *tStamp++;
    for (it=g[v]; it!=NULL; it=it->next)
        if (color[it->dest] != 2) { // not BLACK
            if (color[it->dest == 0]) // WHITE
                r+=sccRec (g,tStamp,it->dest,color,t,M);
            if (M[it->dest] < M[v])
               M[v]=M[it->dest];
        }
    if (M[v]==t[v]) r++;
    color[v]=2;//BLACK
    return r;
}
```

Exemplo 13 Vejamos então a evolução deste algoritmo para o seguinte grafo.

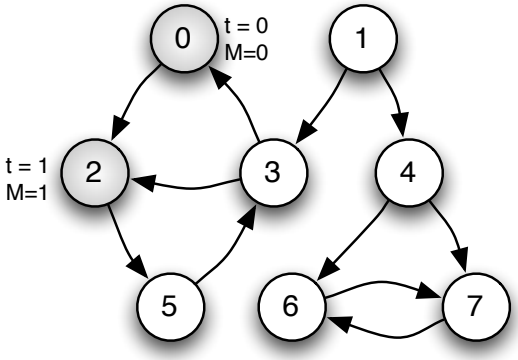


Vamos analisar os estados dos vários vértices. Vamos representar o estado dos vértices usando um círculo sombreado para representar os vértices em visita e um quadrado para representar os vértices já visitados. Os vértices não visitados serão representados por um círculo não sombreado.

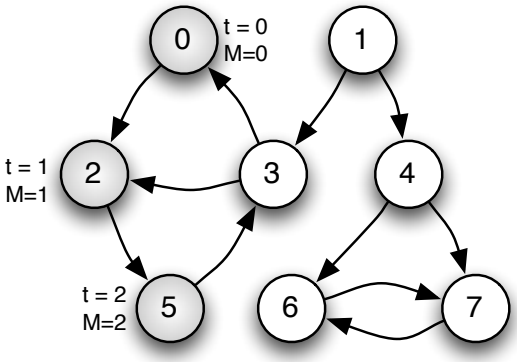
1: Inicia-se uma travessia a partir do vértice 0 no instante 0. Este é marcado como em visita.



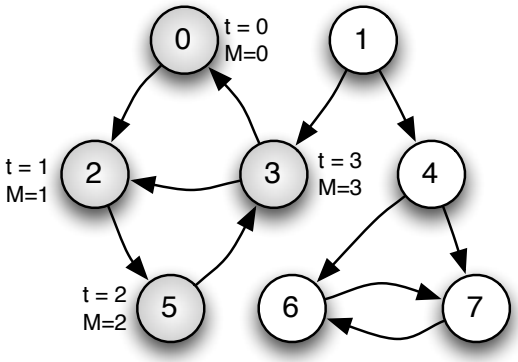
2: Inicia-se uma travessia a partir do vértice 2 no instante 1. Este é marcado como em visita.



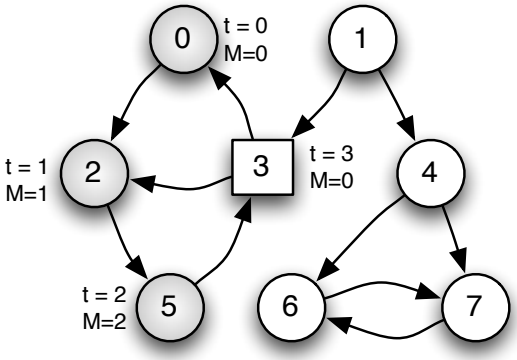
3: Inicia-se uma travessia a partir do vértice 5 no instante 2. Este é marcado como em visita.



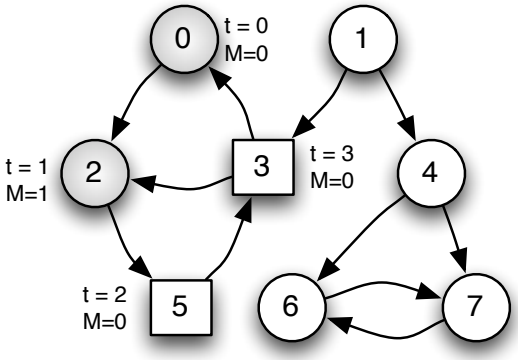
4: Inicia-se uma travessia a partir do vértice 3 no instante 3. Este é marcado como em visita.



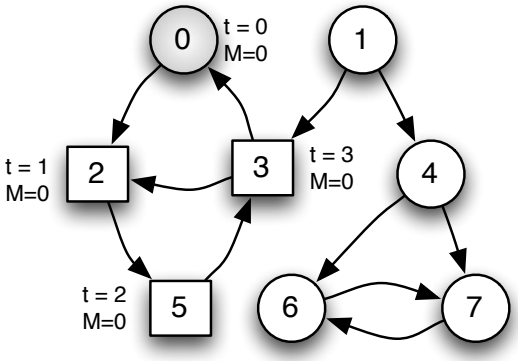
5: Como o vértice 3 não tem mais sucessores não visitados não se iniciam quaisquer travessias. O valor de `M[3]` é actualizado para 0 (`M[0]`).



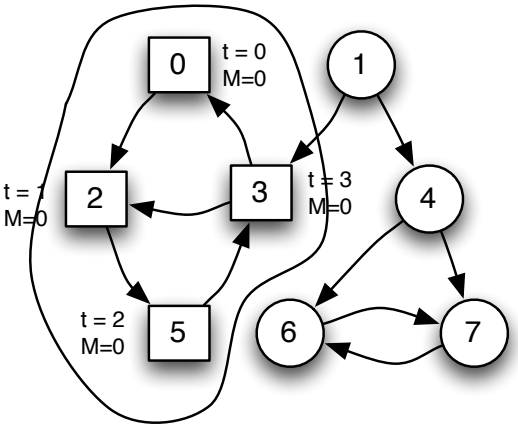
6: Como o vértice 5 não tem mais sucessores não visitados não se iniciam outras travessias. O valor de `M[5]` é actualizado para 0 (`M[3]`).



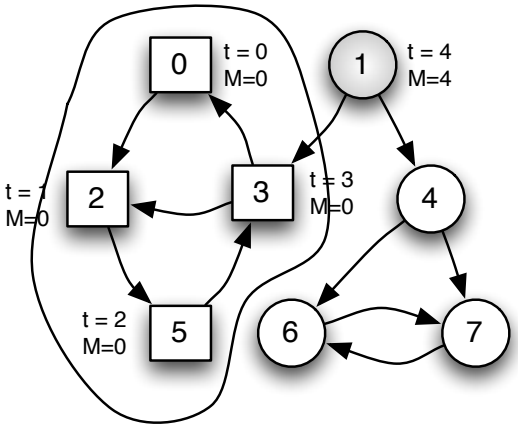
7: Como o vértice 2 não tem mais sucessores não visitados não se iniciam outras travessias. O valor de `M[2]` é actualizado para 0 (`M[5]`).



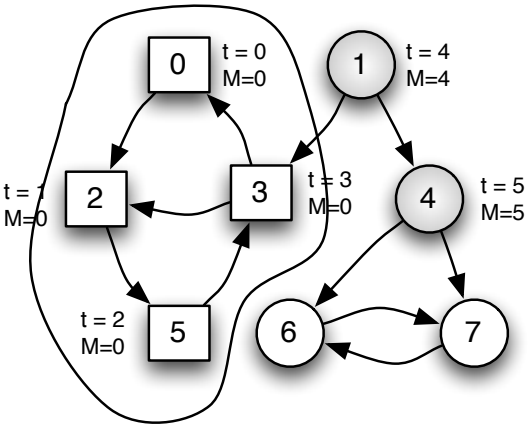
8: Como o vértice 0 não tem mais sucessores não visitados não se iniciam outras travessias. O valor de `M[0]` é actualizado para 0 (`M[3]`). Como `M[0]==t[0]` foi descoberta uma componente fortemente ligada.



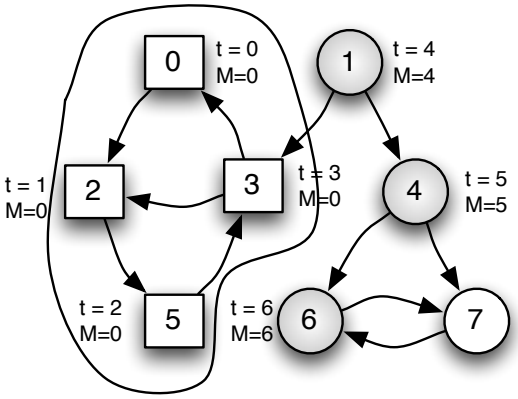
9: Inicia-se uma travessia a partir do vértice 1 no instante 4. Este é marcado como em visita.



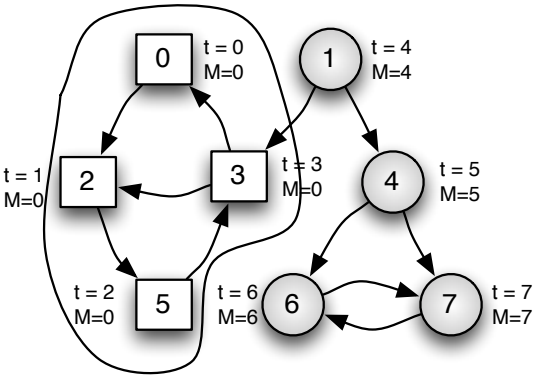
10: Inicia-se uma travessia a partir do vértice 4 no instante 5. Este é marcado como em visita.



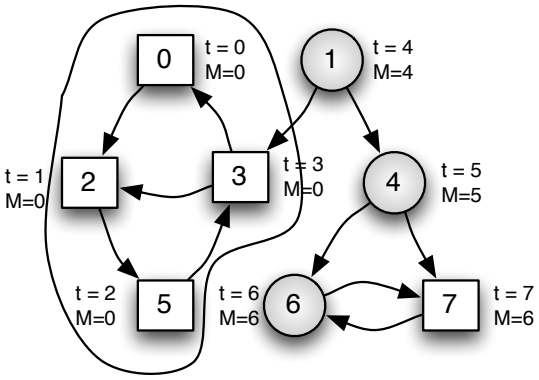
11: Inicia-se uma travessia a partir do vértice 6 no instante 6. Este é marcado como em visita.



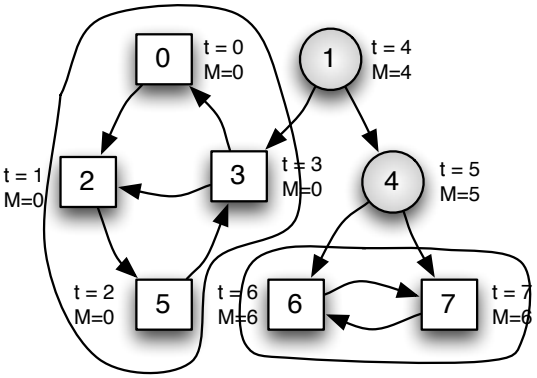
12: Inicia-se uma travessia a partir do vértice 7 no instante 7. Este é marcado como em visita.



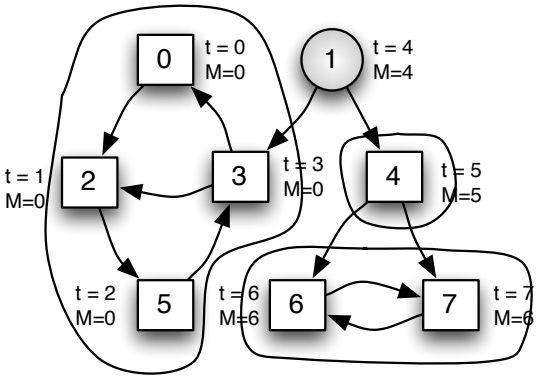
13: Como o vértice 7 não tem mais sucessores não visitados não se iniciam outras travessias. O valor de `M[7]` é actualizado para 6 (`M[6]`).



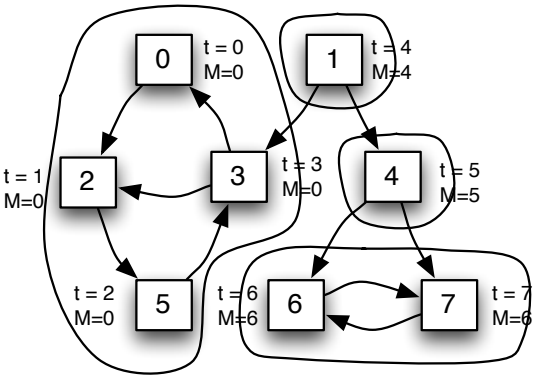
14: Como o vértice 6 não tem mais sucessores não visitados não se iniciam outras travessias. Como `M[6]==t[6]` foi descoberta uma componente fortemente ligada.



15: Como o vértice 4 não tem mais sucessores não visitados não se iniciam outras travessias. Como `M[4]==t[4]` foi descoberta uma componente fortemente ligada.



16: Como o vértice 1 não tem mais sucessores não visitados não se iniciam outras travessias. Como `M[1]==t[1]` foi descoberta uma componente fortemente ligada.



A função apresentada apenas calcula o número de componentes fortemente ligadas de um grafo. O exemplo acima ilustra como no final da função há informação suficiente para determinar para cada par de vértices se eles se encontram ou não na mesma componente ligada. De facto dois vértices `u` e `v` estarão na mesma componente ligada sse `M[u]==M[v]`. A segunda ideia chave do algoritmo de Tarjan resolve o problema de determinar quais os vértices que pertencem a uma mesma componente com muito pouco esforço adicional. Para isso usa-se uma stack onde os vértices vão sendo adicionados à medida que as travessias se iniciam. Mal uma componente fortemente ligada é descoberta, os vértices que a compõe são os que se encontram no topo da dita stack, limitados pelo vértice onde a componente foi descoberta.

Exercício 12 Considere que no cálculo das componentes fortemente ligadas de um grafo se pretende que se pretende obter como resultado, para além do número de componentes existentes, os vértices que compõem essas componentes. Essa informação será retornada em dois arrays (passados como argumento):

- `seq` onde os vértices das várias componentes são listados
- `comp` onde se indica a posição no array `seq` onde termina cada componente.

Relativamente ao exemplo analisado, a função deverá retornar 4 e os arrays deveriam ser preenchidos com:

- `seq` =

0	1	2	3	4	5	6	7
3	5	2	0	7	6	4	1

- `comp` =

0	1	2	3
4	6	7	8

3.5 Fecho transitivo

Dado um grafo não pesado $G = (V, E)$ o seu **fecho transitivo** $G^+ = (V, E^+)$ em que existe uma aresta de o para d em E^+ sse existe em G um caminho com origem o e destino d.
O grafo G^+ é muitas vezes referido como o grafo dos caminhos ou da conectividade.
O cálculo do fecho transitivo de um grafo pode ser feito por repetidas travessias, uma a partir de cada vértice) do grafo original.
A definição seguinte usa a função `DFRec` apresentada atrás e calcula o resultado sob a forma de uma matriz de adjacência.

```
void tclosure (Graph g, GraphMat gp){
    int v,i;
    for (v=0;v<NV;v++){
        // inicialização da linha v
        for (i=0; i<NV; i++)
            gp[v][i] = 0;
        DFRec (g,v,gp[v]);
    }
}
```

Uma vez que a função DFRec é linear no tamanho do grafo, i.e., $T_{\texttt{DFRec}}(V, E) = \Theta(V+E)$ e esta definição implica a invocação dessa função tantas vezes quantos os vértices do grafo, o esforço computacional envolvido é dado por

$$T_{\texttt{tclosure}}(V, E) = \Theta(V * (V + E))$$

No caso de grafos pouco densos, i.e., em que $E = \Theta(V)$, esta definição é bastante eficiente ($\Theta(V^2)$). Mas para o caso de grafos densos, i.e., em que $E = \Theta(V^2)$ a complexidade desta função passa a $\Theta(V^3)$.

Exercício 13 Defina a função `void tclosure (Graph g, Praph GP)` como uma adaptação da função `tclosure` apresentada acima de forma a calcular o resultado sob a forma de listas de adjacência.

3.5.1 Algoritmo de Warshall

O algoritmo de Warshall para o cálculo do grafo da conectividade de um grafo não orientado é um exemplo clássico de programação dinâmica e baseia-se em iterar sobre todos os vértices do grafo:

$I = \{\}$
$G^+ = \cdots$
`while` $(I \neq V)$
    `let` $x \in V - I$
    $G^+ = \cdots$
    $I := I \cup \{x\}$

Preservando o seguinte invariante:

> Em cada iteração do ciclo o grafo G^+ contem a informação sobre os caminhos em G que usam **como nodos intermédios** os vértices pertencentes a I.

Este invariante, aliado à condição do ciclo garante que no final o grafo $G+$ contem a informação pretendida. De facto quando o ciclo termina, i.e., $(I = V)$ o invariante coincide com a definição de G^+.
As partes por completar nesta definição devem garantir o resto da correcção do ciclo, i.e.,

1. que o invariante é válido antes da primeira iteração e
2. que a sua validade é preservada de uma iteração para a seguinte.

De forma a garantirmos que o invariante começa por ser verdadeiro temos que iniciar G^+ com um grafo que tem uma aresta de u para v sse existe um caminho em G **sem vértices intermédios** (I é vazio inicialmente). Ora isto quer dizer que existirá uma aresta em G^+ sse existir uma aresta em G.
Quanto ao segundo ponto, devemos garantir que para o vértice x escolhido incluímos todos os caminhos em que esse vértice surge como vértice intermédio. Para tal basta acrescentar uma aresta (u, v) sempre que já tenha sido descoberto um caminho de u para x e outra de x para v.

```
void warshall (Graph g, GraphMat gp){
    EList it;
    int u, v, x;
    // inicialização de gp
    for (u=0; u<NV; u++) {
        for (v=0; v<NV; v++)
            gp[u][v] = NE;
        for (it=g[u]; it!=NULL; it=it->next)
            gp[v][it->dest] = 1;
    }
    //adição de arestas
    for (x=0; x<NV; x++)
```

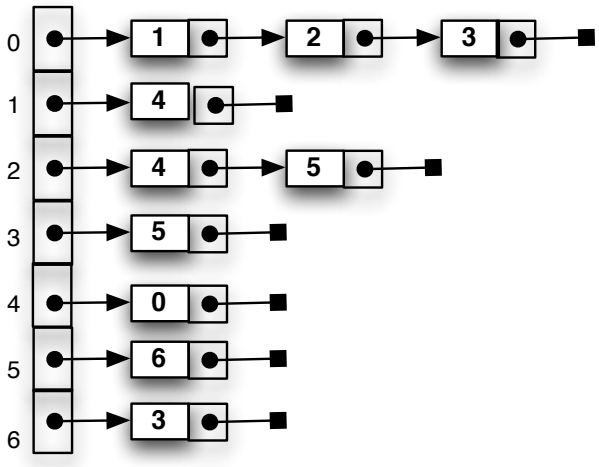
```
        for (u=0; u<NV; u++)
            for (v=0; v<NV; v++)
                gp[u][v] = gp[u][v] ||
                           (gp[u][x] && gp[x][v]);
}
```

O ciclo de adição de arestas pode ser optimizado, evitando algumas iterações do ciclo mais interior.

```
    for (x=0; x<NV; x++)
        for (u=0; u<NV; u++)
            if (gp[u][x])
            for (v=0; v<NV; v++)
                if (gp[x][v])
                    gp[u][v] = 1;
}
```

Os três ciclos (aninhados) no final da função fazem com que a sua complexidade seja $\mathcal{O}(V^3)$, comparável à alternativa anterior para grafos densos. No caso de grafos com (relativamente) poucas arestas, a primeira alternativa tem um comportamento melhor.

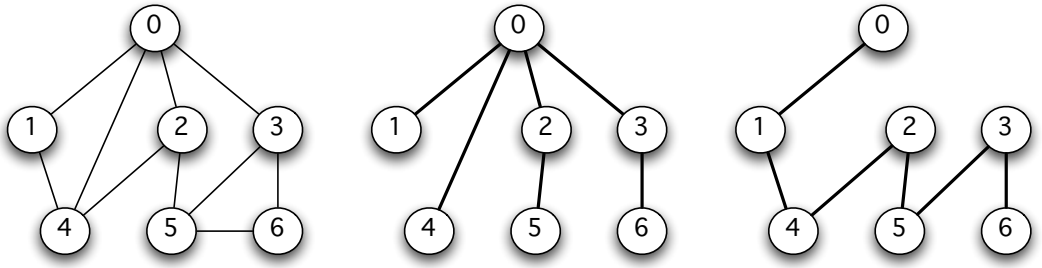
Exercício 14 Apresente a matriz resultante de executar o algoritmo de Warshall quando o grafo de entrada é o apresentado no exemplo 10.



4 Árvore geradora de custo mínimo

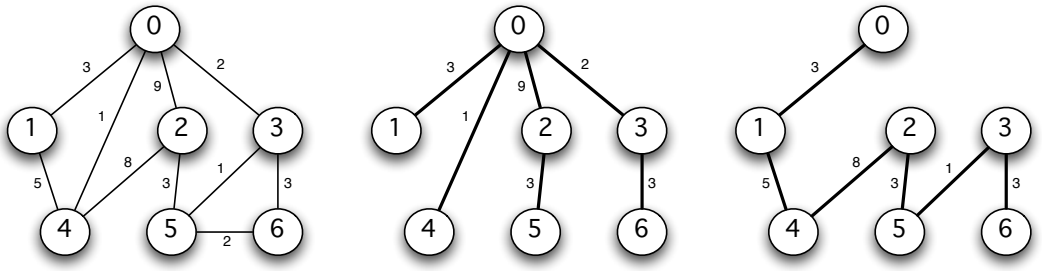
Dado um grafo não orientado e ligado, uma **árvore geradora** do grafo é um sub-grafo ligado e acíclico do grafo.
De uma forma geral, dado um grafo ligado existem várias árvores geradoras.

Exemplo 14 Uma travessia de um grafo é uma forma de determinar uma árvore geradora desse grafo. As árvores apresentadas abaixo correspondem às árvores geradoras resultantes de se fazer uma travessia breadth-first e depth-first a partir do vértice 0 no grafo que se apresenta mais à esquerda.



Um problema com muitas aplicações prácticas é o de, num grafo não orientado, ligado e pesado determinar uma árvore geradora que tenha custo mínimo, i.e., que minimize a soma dos pesos das arestas envovidas.

Exemplo 14 (continuação) Se os pesos das arestas do grafo anterior forem como se apresenta abaixo, o custo das árvores geradoras resultantes das travessias breadth-first e depth-first a partir do vértice 0 serão respectivamente 25 e 22. Nenhuma delas é de custo mínimo (12).



4.1 Algoritmo de Prim

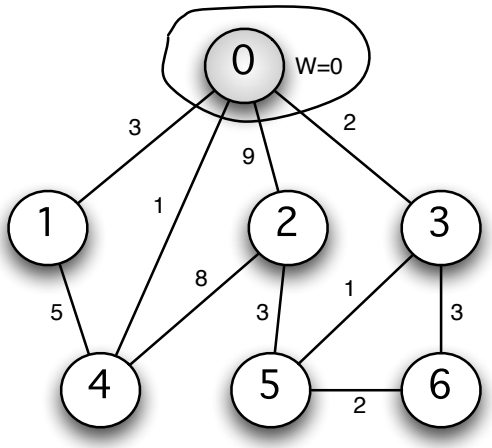
O algoritmo de Prim para o cálculo de uma árvore geradora de custo mínimo tem algumas semelhanças com a travessia breadth-first apresentada atrás.
Também aqui vamos particionar o conjunto dos vértices do grafo em três conjuntos (identificados por uma côr):

- **Visitados** (`BLACK`)
- **Não visitados** (`WHITE`)
- **Orla** (`GREY`)

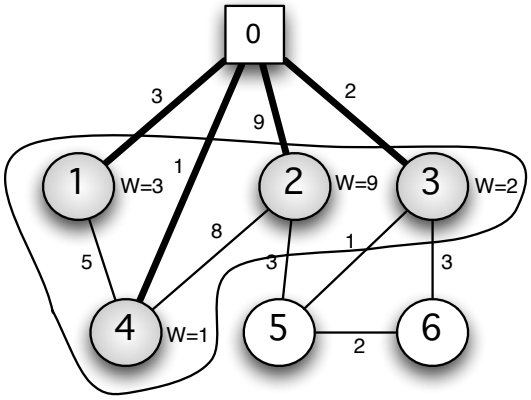
Vamos manter como invariante que se um vértice está na orla então existe uma aresta que o liga a um vértice visitado.
Tal como na travessia breadth-first, o algoritmo evolui retirando um elemento da orla para ser incluído no conjunto dos vértices visitados, mantendo válidas as propriedades da orla.
A grande diferença face à travessia é o critério com que os vértices são escolhidos para saírem da orla. Na travessia a orla era implementada como uma queue e por isso o vértice a ser retirado seria aquele que há mais tempo estava na orla.
No caso do algoritmo de Prim vamos associar a cada vértice que esteja na orla o menor peso entre as arestas que o ligam à parte visitada do grafo. O vértice escolhido será então aquele que tiver o menor destes pesos.

Exemplo 15 Para exemplificar a evolução deste algoritmo vamos usar o grafo do exemplo anterior com a orla inicializada com o vértice 0. Neste caso o peso associado ao vértice 0 será 0 pois corresponde ao custo de adicionar esse vértice ao resultado.

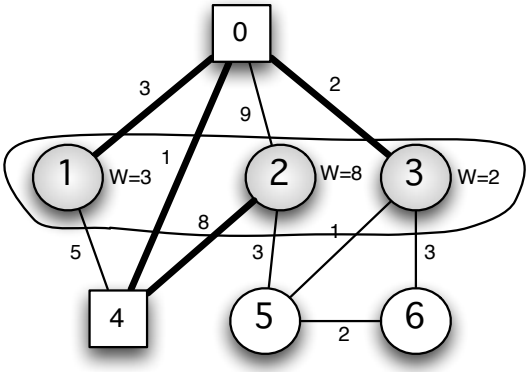
1: A orla é inicializada com um vértice qualquer (neste caso o 0). O peso associado a escolher este vértice é 0. O vértice escolhido é por isso o vértice 0.



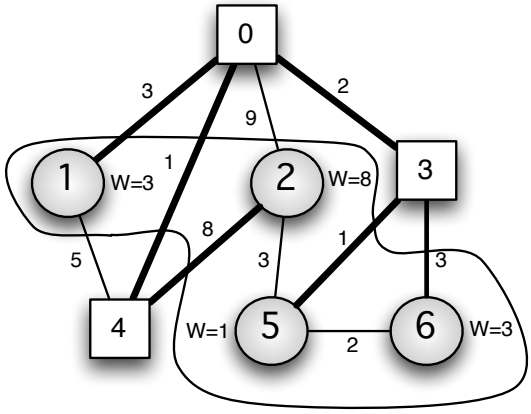
2: São acrescentados à orla os sucessores de 0. Como todos estavam não visitados, o peso que lhes está associado é o peso da aresta que os liga a 0. Destes o que tem o menor peso é o vértice 4 que será o escolhido para saír da orla.



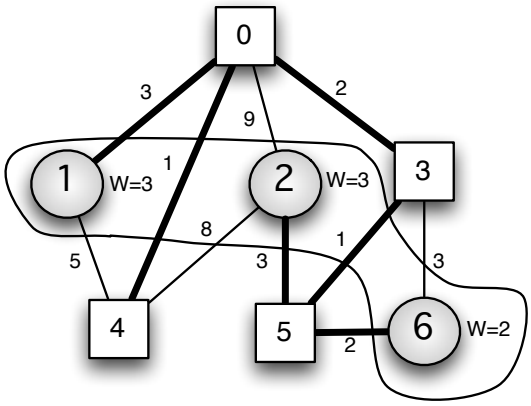
3: Os vértices adjacentes a 4 ou já estão marcados como visitados (1) ou já estão na orla. No caso do vértice 2, o peso da aresta que o liga a 4 é inferior ao peso desse vértice, pelo que esse peso é actualizado. O vértice da orla com o menor peso é o vértice 3.



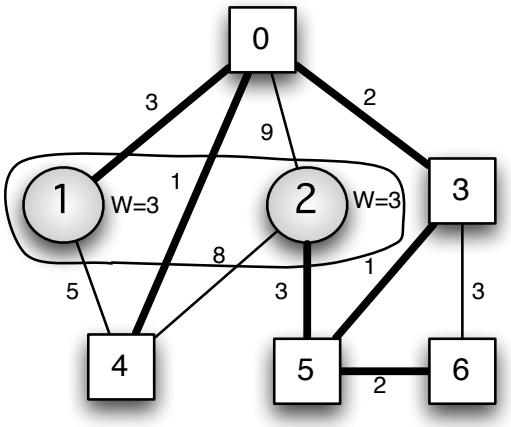
4: Os vértices adjacentes a 3 não visitados (5 e 6) são acrescentados à orla com pesos iguais ao ceso da aresta que os liga a texttt3. O vértice da orla com menor peso é o vértice 5.



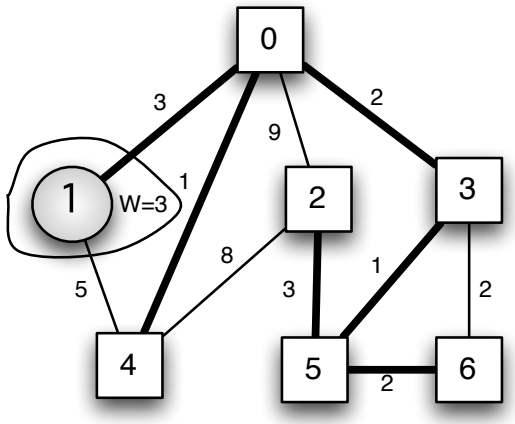
5: O peso das arestas que ligam o vértice 5 a 2 e a 6 é menor do que os pesos desses vértices, pelo que estes são actualizados. O vértice da orla com menor peso é o vértice 6.



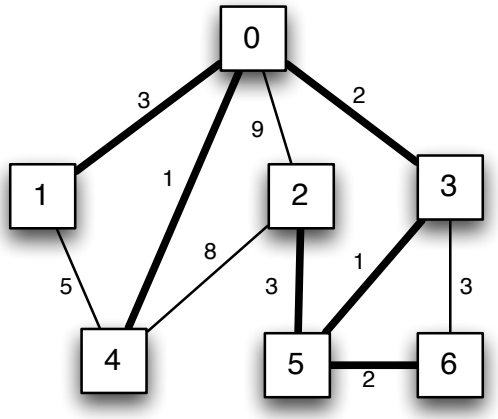
6: O vértice 6 não tem qualquer adjacente não visitado. Ambos os vértices da orla têm peso 3, pelo que pode ser escolhido qualquer um para saír da orla.



7: O vértice 2 não tem qualquer adjacente não visitado.



8: O vértice 1 não tem qualquer adjacente não visitado. O processo termina pois a orla está vazia.



A codificação deste algoritmo vai depender da forma como os nodos da orla (*fringe*) são guardados. Vamos por isso assumir a existência de um tipo de dados `Fringe` para representar a orla, bem como algumas funções de manipulação dessa estrutura de dados. Mais à frente veremos algumas alternativas para a implementação desse tipo.
As funções que iremos precisar são

- inicialização da orla (`initFringe`)
- adição de um novo vértice à orla (`addEdgeFringe`)
- modificação do peso associado a um vértice que pertence à orla (`updateFringe`)
- remoção do vértice de menor peso da orla (`getEdge`)

Os tipos destas funções são os seguintes.

```
typedef struct fringe *Fringe;

int initFringe (Fringe f);
int addEdgeFringe (Fringe f, int v, int cost[]);
int updateFringe (Fringe f, int v, int cost[]);
int getEdge (Fringe f, int cost[]);
```

Para além do uso de um array para marcar o estado de cada vértice (`color` como acontecia na travessia breadth-first, a função vai usar a tal estrutura para guardar a orla.

A fase de inicialização deve colocar todos os vértices como não visitados (`WHITE`) e inicializar a orla com um vértice.

```
for (v=0;v<NV;color[v++] = 0)
    ;
initFringe(f); fringesize=0;
v=0;
color[v]=1; cost[v]=0;
addEdgeFringe(f,v,cost); fringesize++;
```

De seguida, e tal como acontecia na travessia breadth-first, o processo baseia-se em escolher um vértice da orla para ser visitado, mantendo válidas as propriedades (invariante) enunciadas.

```
while (fringesize>0) {
    v = getEdge (f,cost); fringesize--;
    color[v]=2;//BLACK
    ...
}
```

Para cada vértice que é retirado da orla devemos analisar todas as arestas que o têm como origem de forma a preservar as propriedades enunciadas. Dependendo do estado (côr) do destino da aresta devemos fazer uma das seguintes acções:

- Se o vértice ainda não tiver sido analizado (`WHITE`) então deve ser acrescentado à orla.
- Se esse vértice já estiver na orla (`GREY`) e o peso desta aresta fôr menor do que o custo associado ao vértice, devemos actualizar esse custo bem como sinalizar qual é a aresta que o liga aos vértices visitados.
- Os vértices já visitados (BLACK) não são processados.

```
int PrimMST (Graph g, int cost[], int ant[]){
    int res=0; int v;
    EList it;
    int color[NV];
    struct fringe ff, *f;
    int fringesize;
    f = &ff;
```

```
    for (v=0;v<NV;color[v++] = 0)
        ;
    initFringe(f); fringesize=0;
    v=0;
    color[v]=1; cost[v]=0;
    addEdgeFringe(f,v,cost); fringesize++;
    while (fringesize>0) {
        v = getEdge (f,cost); fringesize--;
        res+=cost[v];
        color[v]=2;//BLACK
        for (it=g[v]; it; it=it->next)
            if ((color[it->dest] == 0) || // WHITE
                (color[it->dest] == 1  && // GREY
                 cost[it->dest] > it->cost)) {
                ant[it->dest]=v;
                cost[it->dest]=it->cost;
                if (color[it->dest] == 0) {
                   addEdgeFringe(f,it->dest,cost);
                   fringesize++;
                } else updateFringe(f,it->dest,cost);
            }
    }
    return res;
}
```

Antes de apresentarmos alternativas para a implementação da orla vamos fazer uma análise informal da complexidade deste algoritmo.

- A parte de inicialização inclui a inicialização do array `color` (V) e a inicialização da orla (`initFringe`).
- A função `getEdge` é executada uma vez por cada iteração do ciclo `while`. Este é executado V vezes (assumindo que o grafo é ligado, todos os vértices são incluidos na árvore).
- O ciclo `for` é usado para percorrer todas as arestas do grafo e por isso o número total de execuções é E: destas, V correspondem a chamadas à função `addEdgeFringe` e as restantes correspondem a chamadas à função `updateFringe`.

Se usarmos as letras i, g, a e u para representar o custo das funções `initFringe`, `getEdge`, `addEdgeFringe` e `updateFringe`, respectivamente, a complexidade desta função é:

$$T_{\texttt{PrimMST}}(V,E) = \mathcal{O}(V + i + V * g + V * a + (E - V) * u)$$

Uma alternativa para armazenar a orla consiste em usar um vector sem qualquer ordem. A inserção far-se-á no final do array e a escolha do vértice de menor peso terá de percorrer todos os elementos da orla.

```
struct fringe {
   int size;
   int edges[NV];
};
```

A operação de inicialização da orla é feita em tempo constante.

```
int initFringe (Fringe f){
   f->size=0;
   return 0;
}
```

O mesmo acontecendo com a operação de adição de um novo vértice à orla.

```
int addEdgeFringe (Fringe f, int v, int cost[]) {
  f->edges[f->size++]=v;
  return 0;
}
```

A operação de actualização do peso de um vértice da orla não precisa de fazer nenhuma acção uma vez que não existe qualquer ordem na forma como os vértices estão armazenados.

```
int updateFringe (Fringe f, int v, int cost[]){
  return 0;
}
```

A única operação que não é feita em tempo constante é a operação de selecção (e remoção) do vértice da orla com menor custo.

```
int getEdge (Fringe f, int cost[]){
  int i, minind, r;

  minind=0;
  for (i=1; i<f->size; i++)
    if (cost[f->edges[i]] < cost[f->edges[minind]])
       minind=i;
  r=f->edges[minind];
  f->size--;
  f->edges[minind]=f->edges[f->size];
  return r;
}
```

Esta função executa em tempo linear ao tamanho da orla. E esse valor está limitado pelo número de vértices do grafo.

Com esta implementação da orla, as complexidades referidas são por isso $i = 1$, $g = V$, $a = 1$ e $u = 1$ e a complexidade da função de cálculo da árvore geradora vem então.

$$
\begin{aligned}
T_{\texttt{PrimMST}}(V,E) &= \mathcal{O}(V + i + V * g + V * a + (E - V) * u) \\
&= \mathcal{O}(V + 1 + V * V + V * 1 + (E - V) * 1) \\
&= \mathcal{O}(V + 1 + V^2 + V + E - V) \\
&= \mathcal{O}(V + V^2 + E) \\
&= \mathcal{O}(V^2)
\end{aligned}
$$

Uma outra alternativa seria manter os vértices da orla ordenados (por ordem decrescente do seu peso).
A operação de inicialização da orla não sofre qualquer alteração.
A operação de adição de um novo vértice terá de o fazer ordenadamente. Tem por isso complexidade linear no tamanho da orla.

```
int addEdgeFringe (Fringe f, int v, int cost[]) {
  int i;
  for (i=f->size++; i>0 && cost[f->edges[i]] < cost[v]; i--)
    f->edges[i]=f->edges[i-1];
  f->edges[i]=v;
  return 0;
}
```

A operação de actualização do peso de um vértice da orla pode ter que alterar a posição desse vértice na orla, *puxando-o* para a frente uma vez que só é feita a actualização se o custo fôr menor). A sua complexidade é por isso linear no tamanho da orla (no pior caso).

```
int updateFringe (Fringe f, int v, int cost[]){
  int i;
  for (i=0;f->edges[i]!=v; i++)
      ;
  for (;i<f->size && cost[f->edges[i]]<cost[f->edges[i+1]];i++)
     swap (f->edges, i, i+1);
  return 0;

}
```

Esta operação poderia ser tornada mais eficiente se guardássemos para cada vértice a posição onde ele se encontra na orla. Isso evitaria o primeiro ciclo que procura um dado vértice na orla. No entanto essa informação deverá ser mantida actualizada, implicando um maior esforço tanto no segundo ciclo como na operação de inserção de um novo elemento: alguns elementos da orla vão passar a estar em posições diferentes.
A operação de selecção (e remoção) do vértice da orla com menor custo é feita em tempo constante (trata-se do último vértice da orla).

```
int getEdge (Fringe f, int cost[]){
  f->size--;
  return (f->edges[f->size]);
}
```

Com esta implementação da orla, as complexidades referidas são por isso $i = 1$, $g = 1$, $a = V$ e $u = V$ e a complexidade da função de cálculo da árvore geradora vem então.

$$\begin{aligned} T_{\texttt{PrimMST}}(V, E) &= \mathcal{O}(V + i + V * g + V * a + (E - V) * u) \\ &= \mathcal{O}(V + 1 + V * 1 + V * V + (E - V) * V) \\ &= \mathcal{O}(V + V + V^2 + E * V - V^2) \\ &= \mathcal{O}(V + E * V) \end{aligned}$$

Comparando as complexidades obtidas, e uma vez que o número de arestas de um grafo varia entre V (grafo esparso) e V^2 (grafo denso) temos que

- para grafos densos ($E = \Theta(V^2)$) a primeira solução é melhor
- para grafos esparsos ($E = \Theta(V)$) as soluções são equivalentes.

Uma terceira alternativa para armazenamento da orla consiste em organizar o array numa heap, ordenada por ordem crescente dos custos. A remoção, inserção e actualização da orla fazem-se todas em tempo logorítmico no tamanho da orla.
Nesse caso as complexidades referidas são $i = 1$, $g = \log(V)$, $a = \log(V)$ e $u = \log(V)$ e a complexidade da função de cálculo da árvore geradora vem então.

$$\begin{aligned} T_{\texttt{PrimMST}}(V, E) &= \mathcal{O}(V + i + V * g + V * a + (E - V) * u) \\ &= \mathcal{O}(V + 1 + V * \log(V) + V * \log(V) + (E - V) * \log(V)) \\ &= \mathcal{O}((V + E) * \log(V)) \end{aligned}$$

Esta solução é mais eficiente do que as duas anteriores. No entanto a solução mais eficiente é um melhoramento desta em que se usam várias heaps, ligadas entre si e que permite melhorar o tempo de actualização do custo de um vértice para constante (mais precisamente, permite obter uma definição que tem custo amortizado constante) sem comprometer a eficiência da operação de remoção do menor elemento (que permanece logarítmica).
Com essa implementação, conhecida como *Fibonacci heap*, a complexidade da função `PrimMST` vem

$$\begin{aligned} T_{\texttt{PrimMST}}(V, E) &= \mathcal{O}(V + i + V * g + V * a + (E - V) * u) \\ &= \mathcal{O}(V + 1 + V * \log(V) + V * \log(V) + (E - V) * 1) \\ &= \mathcal{O}(V * \log(V) + E) \end{aligned}$$

Exercício 15 De forma a que a implementação usando heaps tenha uma complexidade logarítmica é necessário evitar procurar um vértice na heap. Para isso podemos usar um array `index` que indica, para cada vértice, em que posição da heap ele se encontra.

```
struct fringe {
    int size;
    int index[NV];
    int edges[NV];
};
```

Apresente definições das operações sobre a orla referidas (`initFringe`, `getEdge`, `addEdgeFringe` e `updateFringe`) para esta implementação.

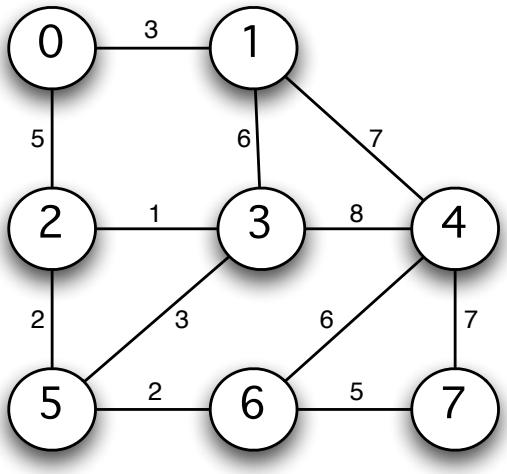
4.2 Algoritmo de Kruskal

O algoritmo de Kruskall para o cálculo de uma árvore geradora de custo mínimo baseia-se na seguinte propriedade.

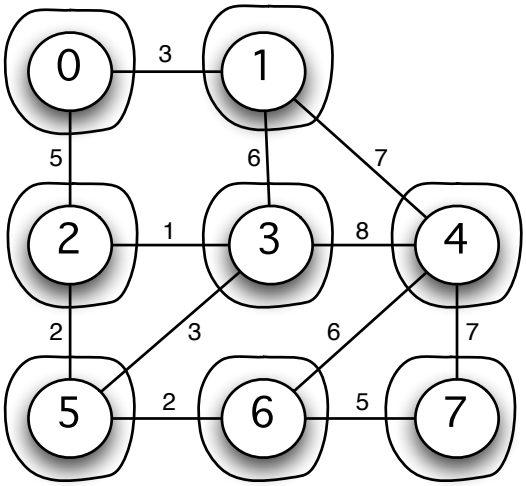
> Uma árvore geradora de um grafo (ligado e não orientado) com V vértices é um sub-grafo não orientado acíclico com $V - 1$ arestas.

Para obter uma árvore geradora de custo mínimo o algoritmo selecciona as $V - 1$ arestas de menor peso que não formam qualquer ciclo.
De forma a determinar se a adição de uma determinada aresta forma ou não um ciclo com as arestas já adicionadas, vamos agrupar os vértices em conjuntos de tal forma que dois vértices estão no mesmo conjunto sse já foram adicionadas arestas que os ligam. Desta forma evitaremos adicionar arestas que ligam vértices já ligados (provocado um ciclo).

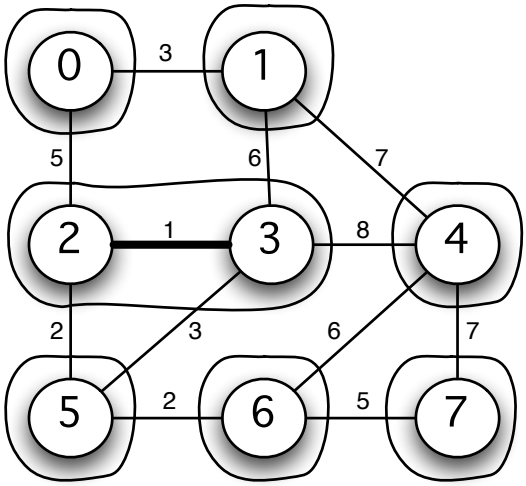
Exemplo 16 Vejamos o comportamento deste algoritmo para o seguinte grafo:



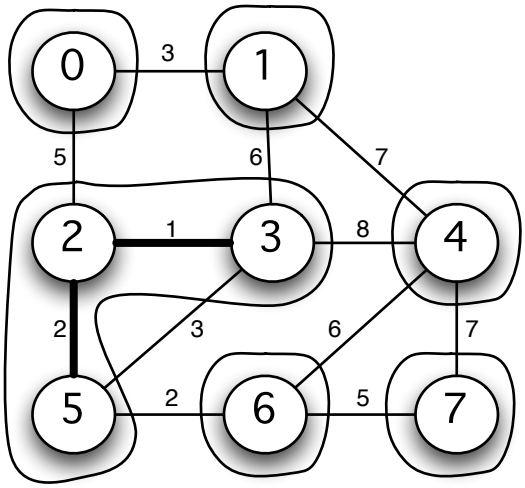
1: Inicialmente todos os vértices se encontram num conjunto diferente.



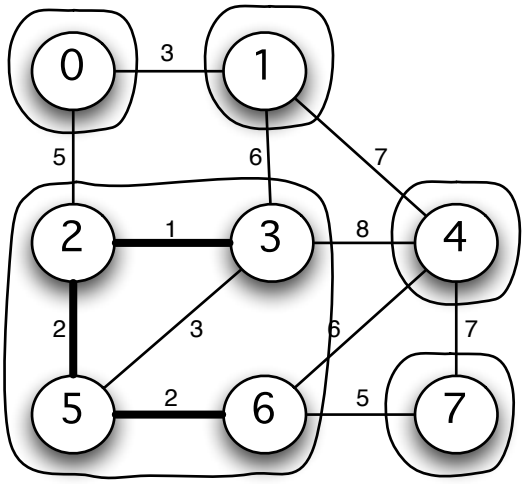
2: A aresta de menor peso (que liga 2 a 3) não gera nenhum ciclo e por isso é adicionada. Os vértices 2 e 3 passam a estar dentro do mesmo conjunto.



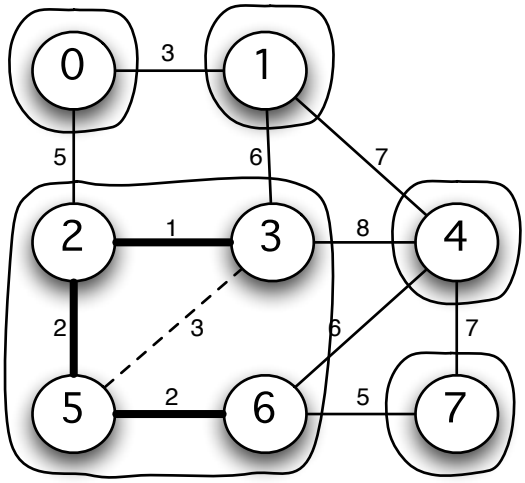
3: A aresta de menor peso (que liga 2 a 5) não gera nenhum ciclo e por isso é adicionada. Os conjuntos que contêm 2 e 5 são reunidos num só.



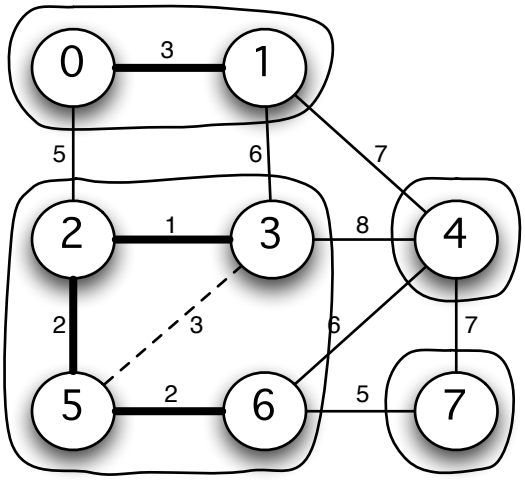
4: A aresta de menor peso (que liga 5 a 6) não gera nenhum ciclo e por isso é adicionada. Os conjuntos que contêm 5 e 6 são reunidos num só.



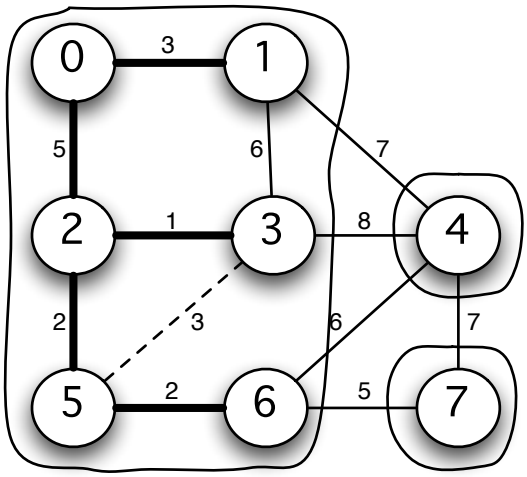
5: A aresta de menor peso (que liga 5 a 3) gera um ciclo (5 e 3 estão no mesmo conjunto) e por isso é descartada.



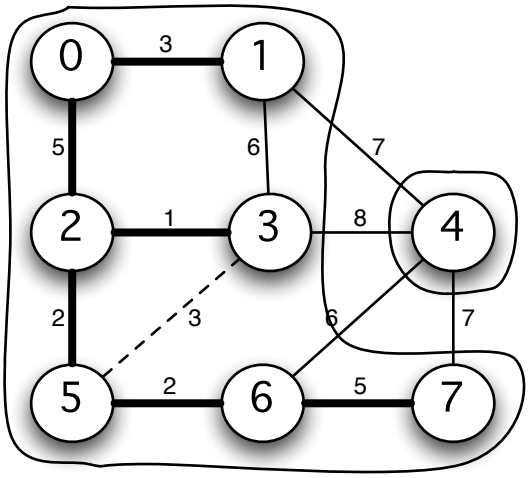
6: A aresta de menor peso (que liga 0 a 1) não gera nenhum ciclo e por isso é adicionada. Os vértices 0 e 1 passam a estar dentro do mesmo conjunto.



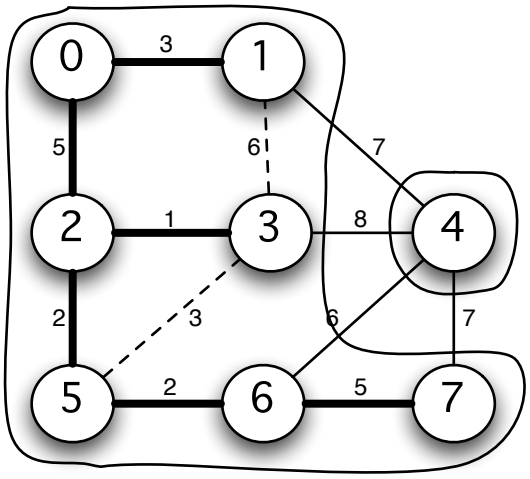
7: A aresta de menor peso (que liga 0 a 2) não gera nenhum ciclo e por isso é adicionada. Os conjuntos que contêm 0 e 2 são reunidos num só.



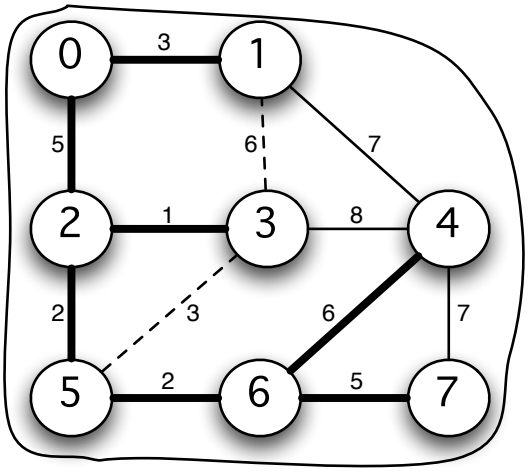
8: A aresta de menor peso (que liga 6 a 7) não gera nenhum ciclo e por isso é adicionada. Os conjuntos que contêm 6 e 7 são reunidos num só.



9: A aresta de menor peso (que liga 1 a 3) gera um ciclo e por isso é descartada.



10: A aresta de menor peso (que liga 6 a 4) não gera nenhum ciclo e por isso é adicionada. Os conjuntos que contêm 6 e 4 são reunidos num só.



Na implementação deste algoritmo temos que dar particular atenção a dois aspectos essenciais.

- como escolher as arestas por ordem crescente do peso
- como determinar se a adição de uma nova aresta gera um ciclo com as arestas já existentes

Para resolver o primeiro destes a função deverá começar por percorrer todo o grafo, armazenando numa estrutura intermédia a informação de todas as arestas de forma a ser fácil determinar qual a próxima aresta a ser incluída.
Isto pode ser feito usando um array (ordenado pelo peso) das arestas (ver exercício 16) ou usando uma min-heap também ordenada pelo peso (ver exercício 17).
Qualquer uma destas alternativas pode ser feita em tempo $\mathcal{O}(E\log(E))$. Uma vez que $E = \mathcal{O}(V^2)$, temos que $\log(E) = \mathcal{O}(\log(V))$ e por isso esta fase pode ser feita em tempo $\mathcal{O}(E\log(V))$.

Exercício 16 Considere o seguinte tipo para representar arestas de um grafo.

```
typedef struct sedge {
    int o, d;
    int cost;
} Edge;
```

Defina uma função `void sortedEdges (Graph g, Edge e[], int N)` que, dado um grafo com `N` arestas, preenche o array e com a informação das `N` arestas do grafo ordenadas por ordem crescente do peso.

Garanta que a função definida executa em tempo $\mathcal{O}(N\log(N))$.

Exercício 17 Considere o seguinte tipo para representar arestas de um grafo.

```
typedef struct sedge {
    int o, d;
    int cost;
} Edge;
```

Defina uma função `void minHEdges (Graph g, Edge e[], int N)` que, dado um grafo com `N` arestas, preenche o array e com a informação das `N` arestas do grafo organizado como uma *min-heap* (ordenada por peso).
Garanta que a função definida executa em tempo $\mathcal{O}(N\log(N))$.

4.2.1 Find & Union

Vamos agora apresentar algumas soluções para resolver a segunda questão identificada no algoritmo de Kruskal, mas que tem aplicações na resolução doutros problemas.
Aquilo que pretendemos é gerir uma partição do conjunto dos vértices de um grafo, isto é, organizar os vértices de um grafo em sub-conjuntos disjuntos de forma a ser possível determinar de forma eficiente se dois vértices pertencem ou não ao mesmo sub-conjunto.
As operações fundamentais para este tipo abstracto de dados são:

- **find** – que determina o identificador do sub-conjunto a que um vértice pertence.
- **union** – que, dados os identificadores de dois sub-conjuntos faz a união dos correspondentes sub-conjuntos num só.

Por essa razão é costume chamar a este tipo de dados **find and union**.

A forma mais imediata de implementar este tipo abstracto de dados é usar um array, cujos índices são os vértices, onde, para cada vértice, se guarda o identificador do conjunto a que esse vértice pertence.

Exemplo 17 Para um conjunto de 7 vértices (`NV==7`) o array c com o seguinte valor

	0	1	2	3	4	5	6
c =	3	2	2	3	4	5	3

indica que os vértices 0, 3 e 6 estão num mesmo sub-conjunto (cujo identificador é 3), assim como os vértices 1 e 2 também pertencem ao mesmo sub-conjunto (cujo identificador é 2). Os restantes vértices estão em sub-conjuntos singulares. Representa por isso a partição

$$\{\{0,3,6\},\{1,2\},\{4\},\{5\}\}$$

A operação de inicialização, que coloca cada vértice num sub-conjunto singular tem a seguinte definição.

```
void initFAU (int c[]){
  int i;
  for (i=0;i<NV;i++)
    c[i]=i;
}
```

A operação de cálculo do identificador associado a um vértice consiste apenas na consulta do array de identificadores, e tem por isso uma complexidade constante ($T_{\texttt{findFAU}}(N) = \Theta(1)$).

```
int findFAU (int c[], int v){
  return (c[v]);
}
```

A operação de união de dois subconjuntos vai precisar de percorrer todo o array para alterar ocorrências de um dos identificadores para o outro e tem por isso uma complexidade linear ($T_{\texttt{unionFAU}}(N) = \Theta(N)$).

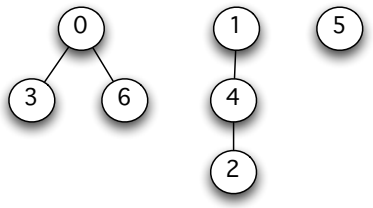
```
int unionFAU (int c[], int id1, int id2){
  int i;
  for (i=0;i<NV;i++)
    if (c[i] == id2) c[i]=id1;
  return id1;
}
```

Uma solução alternativa consiste em usar um array para implementar uma floresta de vértices (tal como fizemos nas funções de travessia). Dois vértices estão no mesmo subconjuto sse pertencerem à mesma árvore. O identificador de cada sub-conjunto será então o vértice raíz da árvore a que esse vértice pertencer.
Como o que guardamos em cada posição do array é o antecessor de cada vértice, um vértice é uma raíz se na correpondente posição estiver armazenado um valor negativo (por exemplo `-1`).

Exemplo 18 Para um conjunto de 7 vértices (`NV==7`) o array c com o seguinte valor

	0	1	2	3	4	5	6
c =	-1	-1	4	0	1	-1	0

representa a floresta.



Representa por isso a partição

$$\{\{0,3,6\},\{1,2,4\},\{5\}\}$$

Vejamos então as definições das funções de manipulação da estrutura ndesta definição alternativa.
A operação de inicialização, que coloca cada vértice num sub-conjunto singular tem que produzir uma floresta com uma árvore por vértice.

```
void initFAU (int c[]){
  int i;
  for (i=0;i<NV;i++)
    c[i]=-1;
}
```

A operação de cálculo do identificador associado a um vértice tem de determinar qual o vértice raíz da árvore a que o vértice argumento pertence. Para isso deverá *subir* pela árvore até atingir a raíz.

```
int findFAU (int c[], int v){
  while (c[v] >= 0)
    v = c[v];
  return v;
}
```

O número de iterações do ciclo depende da altura a que o vértice em questão esteja na respectiva árvore. No pior caso a árvore encontra-se totalmente desbalanceada, todos os vértices estão nessa árvore e o vértice em questão é a folha dessa árvore. Neste caso extremo o ciclo faz V iterações, e por isso ($T_{\texttt{findFAU}}(N) = \Theta(N)$).
A operação de união de dois subconjuntos pode ser feita em tempo constante ($T_{\texttt{unionFAU}}(N) = \Theta(1)$). Basta fazer com que uma das árvores passe a ser sub-árvore da outra.

```
int unionFAU (int c[], int id1, int id2){
  c[id2]=id1;
  return id1;
}
```

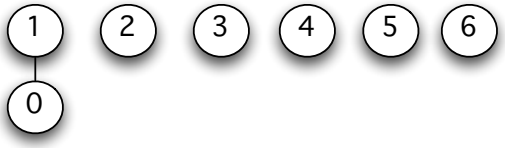
Exemplo 19 Vejamos o que acontece quando, partindo do estado inicial (tal como produzido pela função `initFAU`), num universo com 7 vértices (de 0 a 6), se processam as arestas `(1,0)`, `(0,3)`, `(6,5)`, `(1,6)`, `(1,4)` e `(1,2)` por essa ordem.
Note-se que o processamento de uma aresta `(a,b)` corresponde a: (1) determinar os identificadores correspondentes a `a` e `b` e (2) juntar os conjuntos correspondentes.

1: O estado inicial corresponde a ter uma árvore por cada vértice.

	0	1	2	3	4	5	6
c =	-1	-1	-1	-1	-1	-1	-1

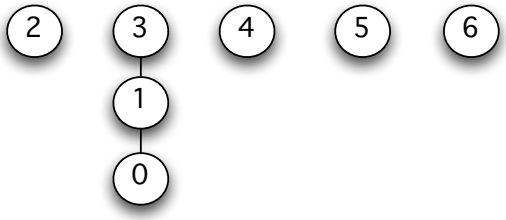
2: O processamento da aresta `(1,0)` junta a árvore que contém 1 à que contém 0.

	0	1	2	3	4	5	6
c =	1	-1	-1	-1	-1	-1	-1



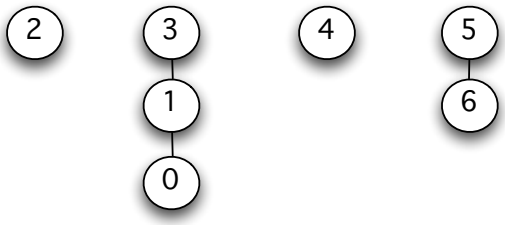
3: O processamento da aresta `(0,3)` junta a árvore que contém 3 à que contém 0 (cuja raíz é 1).

	0	1	2	3	4	5	6
c =	1	3	-1	-1	-1	-1	-1



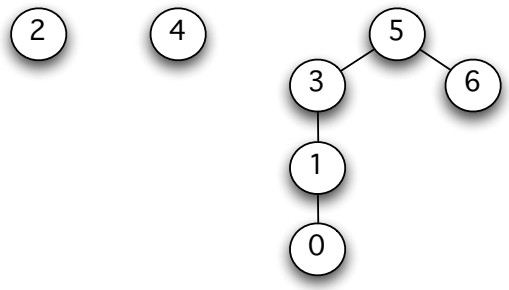
4: O processamento da aresta `(6,5)` junta a árvore que contém 6 à que contém 5.

	0	1	2	3	4	5	6
c =	1	3	-1	-1	-1	-1	5



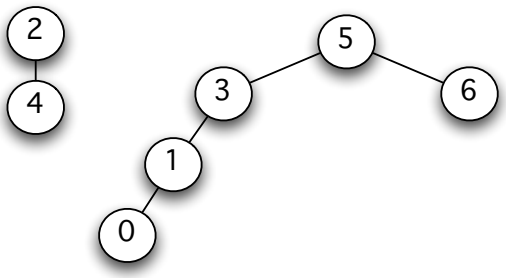
5: O processamento da aresta `(1,6)` junta a árvore que contém 1 (cuja raíz é 3) à que contém 6 (cuja raíz é 5).

	0	1	2	3	4	5	6
c =	1	3	-1	5	-1	-1	5



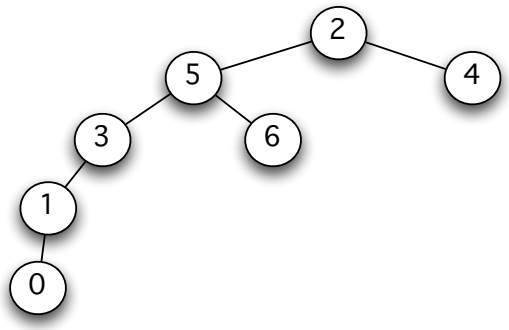
6: O processamento da aresta `(2,4)` junta a árvore que contém 2 à que contém 4.

	0	1	2	3	4	5	6
c =	1	3	-1	5	2	-1	5



7: O processamento da aresta `(1,2)` junta a árvore que contém 1 (cuja raíz é 5) à que contém 2.

	0	1	2	3	4	5	6
c =	1	3	-1	5	2	2	5



Este exemplo evidencia como as árvores produzidas podem ter apenas um elemento por nível, piorando a performance da função `find`.

Exercício 18 Tal como no exemplo anterior, considere que se parte do estado inicial (produzido pela função `initFAU`), num universo com 7 vértices (de 0 a 6). Determine uma sequência de arestas cujo processamento produz uma árvore completamente desbalanceada (com peso 7).

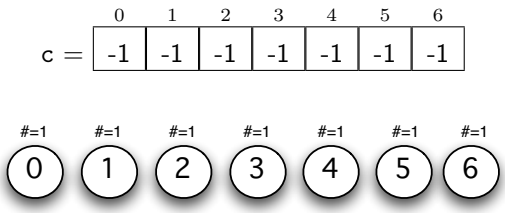
A análise deste exemplo dá ainda algumas pistas em como evitar que tais configurações sejam atingidas.
Uma forma de atingir tal objectivo consiste em modificar a função de junção de duas árvores de forma a, sempre que possível não aumentar a altura da maior das árvores.

Para isso devemos ter informação, para cada árvore da sua altura. Ou, alternativamente, do número de elementos dessa árvore.
Como no array que representa a floresta apenas temos que marcar a raíz das árvores com um número negativo, podemos para cada árvore com `N` elementos guardar `-N`.
A nova definição da função de união de duas árvores será então.

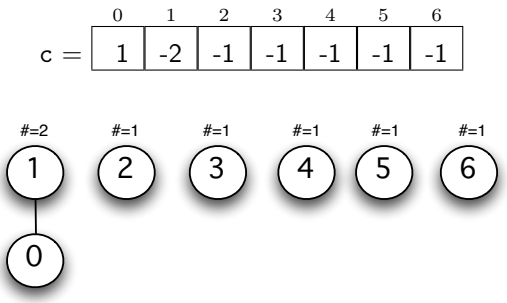
```
int unionFAU (int c[], int id1, int id2){
  int p, f;
  if (c[id1] < c[id2]) {
    // id1 tem mais elementos
    p=id1; f=id2;
  } else {
    // id2 tem mais elementos
    p=id2; f=id1;
  }
    c[p] += c[f];
    c[f] = p;
    return p;
}
```

Exemplo 19 (continuação) Vejamos então o que acontece se repetirmos o processamento da mesma sequência de arestas com esta nova definição da função `findFAU`.

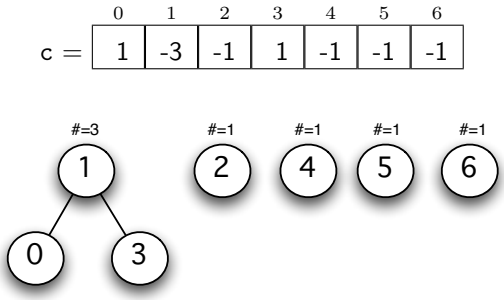
1: O estado inicial corresponde a ter uma árvore por cada vértice. Todos os conjuntos têm 1 elemento.



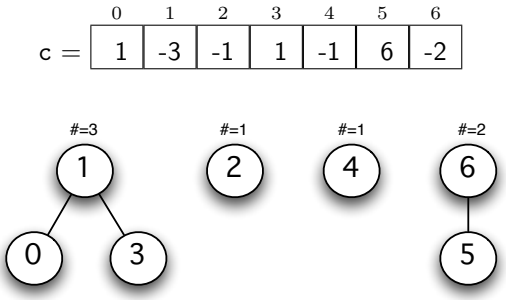
2: O processamento da aresta `(1,0)` junta a árvore que contém `1` à que contém `0`. A raíz resultante (`1`) passa a ter 2 elementos.



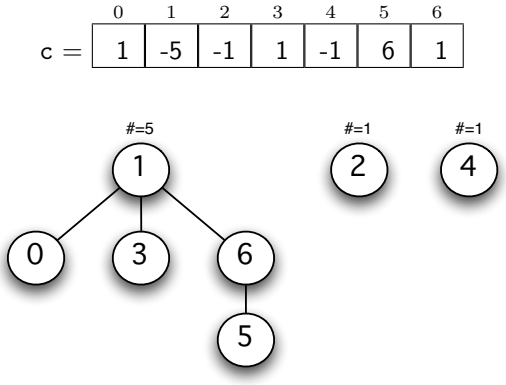
3: O processamento da aresta `(0,3)` junta a árvore que contém `0` à que contém `3`. A raíz resultante (`1`) passa a ter 3 elementos.



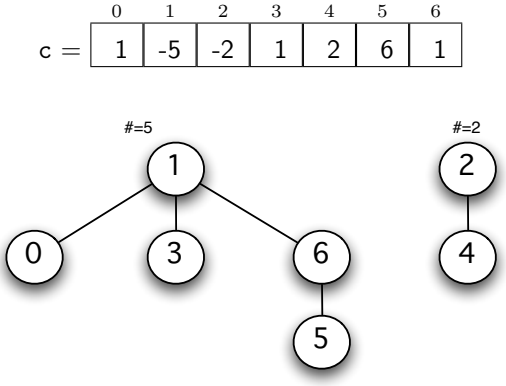
4: O processamento da aresta `(6,5)` junta a árvore que contém `6` à que contém `5`. A raíz resultante (`6`) passa a ter 2 elementos.



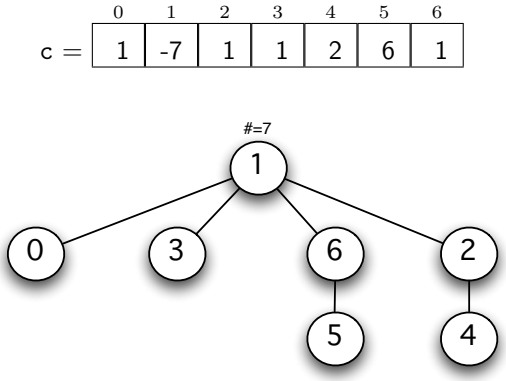
5: O processamento da aresta (1,6) junta a árvore que contém 1 à que contém 6. A raíz resultante (1) passa a ter 5 elementos.



6: O processamento da aresta (2,4) junta a árvore que contém 2 à que contém 4. A raíz resultante (2) passa a ter 2 elementos.



7: O processamento da aresta (1,2) junta a árvore que contém 1 à que contém 2. A raíz resultante (1) passa a ter 7 elementos.



Exercício 19 Determine o estado final de processsar a sequência de arestas calculada no exercício 18 com esta nova definição da função `unionFAU`.

O exemplo apresentado mostra que esta definição da função de união mantém as árvores com uma altura inferior à da definição anterior.
Não é difícil mostrar que a altura destas árvores (e consequentemente a complexidade da função `findFAU`) está limitada pelo logarítmo do número de elementos, i.e., que para qualquer árvore a da floresta se h_a e n_a forem respectivamente a altura e o número de elementos da árvore, verifica-se que

$$h_a \leq 1 + \log_2(n_a) \quad \text{ou, equivalentemente,} \quad n_a \geq 2^{h_a - 1}$$

Suponhamos que pretendemos juntar duas árvores a e b cujo número de elementos é n_a e n_b e cujas alturas são h_a e h_b respectivamente. Sem perda de generalidade seja $n_a \geq n_b$.

A operação de união das duas árvores vai construir uma nova árvore c (com $n_c = n_a + n_b$ elementos) que resulta de acrescentar a árvore b como sub-árvore de a.

Vamos assumir que as árvores a e b satisfazem a dita propriedade, i.e.,

$$h_a \leq 1 + \log_2(n_a) \quad \text{e} \quad h_b \leq 1 + \log_2(n_b)$$

O que é equivalente a

$$n_a \geq 2^{h_a - 1} \quad \text{e} \quad n_b \geq 2^{h_b - 1}$$

e vamos tentar mostrar que a nova árvore c também satisfaz essa propriedade.

Como $n_a \geq n_b$ sabemos que essa relação também se verifica quanto às alturas das árvores, i.e., $h_a \geq h_b$.

Temos dois casos:

Caso em que $h_a > h_b$

Neste caso a altura da árvore c, h_c, é a mesma da da árvore a e então

$$h_c = h_a \leq 1 + \log_2(n_a) \leq 1 + \log_2(n_a + n_b) = 1 + \log_2(n_c)$$

pelo que $h_c \leq 1 + \log_2(n_c)$

Caso em que $h_a = h_b$

Neste caso a altura da árvore c, $h_c = 1 + h_a$, ou seja, $h_a = h_c - 1$. Por hipótese, $n_b \geq 2^{h_b - 1} = 2^{h_a - 1}$. Por isso,

$$n_c = n_a + n_b \geq 2^{h_a - 1} + 2^{h_b - 1} = 2^{h_a - 1} + 2^{h_a - 1} = 2 * 2^{h_a - 1} = 2^{h_a} = 2^{h_c - 1}$$

pelo que $n_c \geq 2^{h_c - 1}$

Exercício 20 Uma estratégia alternativa consiste em guardar para cada uma das árvores a sua altura em vez do número de elementos da árvore.

1. Redefina a função `unionFAU` segundo essa estratégia.
2. Mostre que a função que esta estratégia também assegura que a altura de cada uma das árvores da floresta está limitada pelo logarítmo do número de elementos.

Uma outra optimização que se pode fazer consiste em usar as invocações da função `findFAU` para ir diminuindo a altura das árvores.
A definição da função `findFAU` apresentada acima percorre um troço da árvore, desde o vértice em questão até ao vértice que se encontra na raíz. Com uma definição recursiva da função podemos aproveitar esse processo para colocar todos os vértices desse troço como descendentes directos da raíz.

```
int findFAU (int c[], int v){
    int r;
    if (c[v] < 0) r = v;
```

```
    else {
     r = find (c, c[v]);
     c[v] = r;
    }
    return r;
}
```

Exercício 21 Repita a simulação feita no exemplo 19 assumindo esta nova definição da função `findFAU`.

5 Caminhos mais curtos

Um problema comum em grafos pesados (em que a cada aresta está associado uma distância) consiste em determinar, para um dado par de vértices qual o caminho mais curto (menor distância) que liga esses vértices.
Um problema relacionado, e normalmente com o mesmo custo consiste em determinar, para um dado vértice, os caminhos mais curtos entre esse vértice e todos os seus alcançáveis.
Um terceiro problema consiste em determinar os caminhos mais curtos entre cada par de vértices.
Este terceiro problema pode ser solucionado por sucessivas invocações de uma função que resolva o segundo. Há no entanto formas alternativas e por vezes mais eficientes de o fazer.

5.1 Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford para o cálculo do caminho mais curto entre um vértice v e todos os seus alcançáveis é um exemplo de programação dinâmica.
Baseia-se no facto de que se o caminho

$$\langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$$

é o caminho mais curto que liga v_0 a v_n então

$$\langle v_0, v_1, \ldots, v_{n-1} \rangle$$

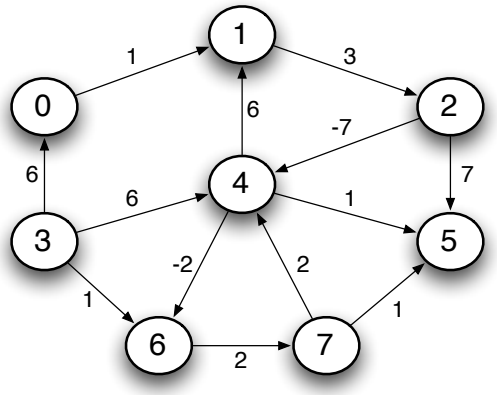
é o caminho mais curto que liga v_0 a v_{n-1}.
Para além disso, e a menos que existam ciclos com custo negativo o caminho mais curto entre dois vértices nunca tem mais do que `NV-1` arestas.
Se existirem ciclos com peso negativo, o caminho mais curto entre dois vértices pode ser um caminho infinito, pelo que o resultado deste algoritmo pode não ser significativo.
O algoritmo itera sobre o número de arestas que compõe os caminhos mais curtos ($i \in [0 \ldots N]$), mantendo actualizado um array w de forma a preservar a seguinte propriedade (invariante):

> $w[v] = x$ sse o caminho mais curto que liga o vértice inicial a v usando no máximo i arestas tem peso x.

Para manter válida esta propriedade, cada iteração deve percorrer todas as arestas e testar se elas podem ser usadas para terminar um caminho.
Para uma aresta (u, v) com peso w isto vai acontecer quando já tiver sido descoberto um caminho até u com peso w_u e além disso $w_u + w$ fôr menor do que o peso do caminho associado a v.

```
void bellmanfordSP (Graph g, int o, int w[], int ant[]){
  int i; int newcost;
  EList it;

  for (i=0; i<NV; i++){
      w[i]   = NE;
      ant[i] = -1;
  }
  w[o] = 0;
  for (i=0; i<NV-1; i++)
     for (v=0; v<NV; v++)
        if (w[v] != NE)
           for (it=g[v]; it; it=it->next) {
              newcost = w[v] + it->cost;
              if (w[it->dest]==NE ||
                   newcost < w[it->dest])){
                 w[it->dest] = newcost;
                 ant[it->dest] = v;
              }
           }
}
```

Exemplo 20 Vejamos o comportamento deste algoritmo quando executado sobre o grafo abaixo a partir do vértice 3.



0. Os arrays w e ant são inicializados com os seguintes valores.

	0	1	2	3	4	5	6	7
w =	NE	NE	NE	0	NE	NE	NE	NE

	0	1	2	3	4	5	6	7
ant =	-1	-1	-1	-1	-1	-1	-1	-1

1. As únicas arestas consideradas são as que têm origem no vértice 3: $(3, 0)$ com custo 6, $(3, 4)$ com custo 6 e $(3, 6)$ com custo 1. Isto vai alterar os arrays nas posições 0, 4 e 6.

	0	1	2	3	4	5	6	7
w =	6	NE	NE	0	6	NE	1	NE

	0	1	2	3	4	5	6	7
ant =	3	-1	-1	-1	3	-1	3	-1

2. As arestas consideradas são as que têm origem nos vértices 0 ($(0, 1)$), 3 ($(1, 2)$), 4 ($(4, 1)$ e $(4, 5)$) e 6 ($(6, 7)$). Isto vai alterar os arrays nas posições 1, 2, 5 e 7.

	0	1	2	3	4	5	6	7
w =	6	7	10	0	6	7	1	3

	0	1	2	3	4	5	6	7
ant =	3	0	1	-1	3	4	3	6

3. Todas as arestas são consideradas uma vez que já foi encontrado pelo menos um caminho para todos os vértices.

	0	1	2	3	4	5	6	7
w =	6	7	10	0	3	4	1	3

	0	1	2	3	4	5	6	7
ant =	3	0	1	-1	2	4	3	6

4. Mais uma vez todas as arestas são consideradas.

	0	1	2	3	4	5	6	7
w =	6	7	10	0	3	4	1	3

	0	1	2	3	4	5	6	7
ant =	3	0	1	-1	2	4	3	6

   Apesar de todas as arestas terem sido consideradas não houve qualquer alteração nos arrays `w` e `ant`. Pelo que será este o resultado final

Este exemplo evidencia uma optimização que é costume ser feita no algoritmo de Bellman-Ford. Se uma iteração do ciclo mais exterior não produz quaisquer alterações, a função pode terminar.
Em termos de complexidade, podemos ver que para um grafo com V vértices o ciclo mais exterior faz no máximo $V - 1$ iterações. Cada iteração consiste em percorrer todo o grafo. Pelo que, a complexidade desta função é

$$T_{\texttt{bellmanfordSP}}(V, E) = \mathcal{O}((V - 1)(V + E)) = \mathcal{O}(V^2 + V * E)$$

Exercício 22 Tal como foi referido, o resultado deste algoritmo só é significativo se não houver no grafo nenhum ciclo com custo negativo.
Nos casos em que existe um ciclo com peso negativo a inclusão desse ciclo num caminho diminui o custo do caminho. Dessa forma não podemos garantir que o caminho mais curto tem no máximo `NV-1` arestas.
Altere a definição acima de forma a que a função retorne um código de sucesso (`0` em caso de não haver ciclos com custo negativo).

5.2 Algoritmo de Dijkstra

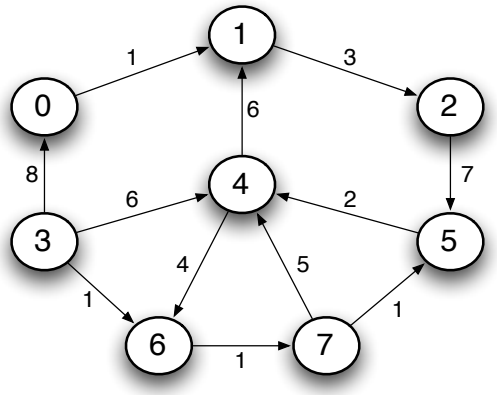
O algoritmo de Dijkstra para cálculo do caminho mais curto entre dois vértices é uma variante do algoritmo de Prim apresentado atrás para cálculo de uma árvore geradora de custo mínimo.
Trata-se de um algoritmo *greedy* que só funciona para grafos com pesos não negativos. Uma consequência de não existirem pesos negativos é que ao adicionarmos novas arestas a um caminho o seu custo total nunca diminui.
Tal como no algoritmo de Prim, o algoritmo de Dijkstra particiona o conjunto dos vértices em três zonas:

- Os vértices já processados (`BLACK`) e para os quais já se conhece o peso do caminho mais curto que os une ao vértice de partida.
- Os vértices ainda não analizados (`WHITE`)
- A orla (`GREY`), composta pelos vértices não processados que têm pelo menos uma aresta a ligá-los aos já processados.

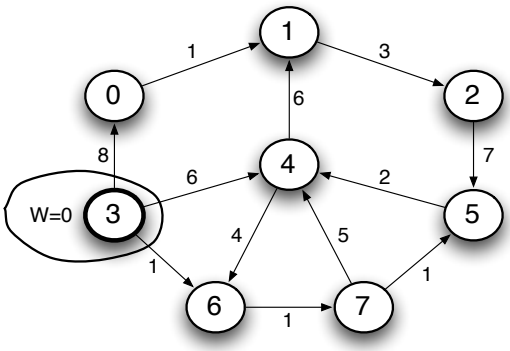
Para cada elemento da orla vamos guardar o custo do melhor caminho conhecido até à altura. Para calcular esse custo, e como cada elemento da orla está ligado a pelo menos um dos vértices já processados, teremos que ter o mínimo entre a soma das referidas arestas e dos custos das origens dessas arestas.
Tal como acontecia no algoritmo de Prim, o vértice da orla que será seleccionado para ser processado será aquele que tem o menor custo.
O algoritmo termina quando a orla não tiver quaisquer vértices (e nesse caso não há caminho para o destino pretendido) ou quando o vértice destino fôr seleccionado para saír da orla.

Uma variante deste problema consiste em determinar o caminho mais curto desde um vértice até todos os alcançáveis. Neste caso o algoritmo termina apenas quando se esgotam os elementos da orla.
Antes de apresentarmos o código deste algoritmo vejamos a sua evolução num pequeno exemplo.

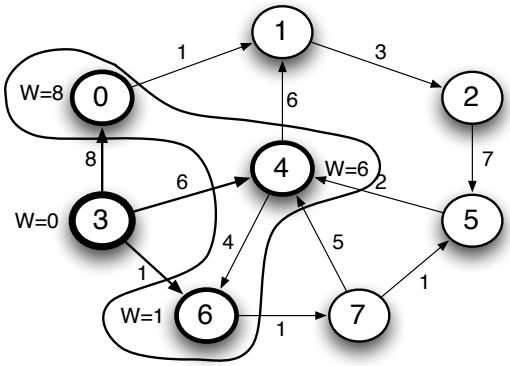
Exemplo 21
Vejamos então a evolução do algoritmo de Dijkstra quando invocado sobre o grafo apresentado abaixo a partir do vértice 3, i.e., queremos determinar os pesos dos caminhos mais curtos com origem no vértice 3.



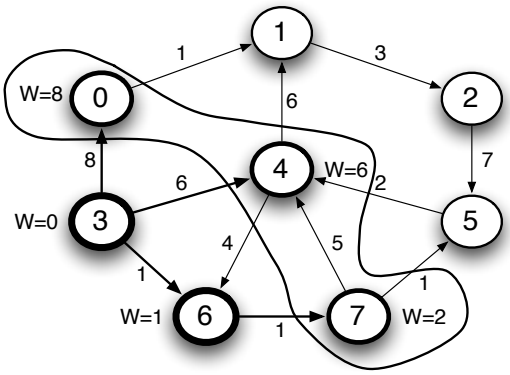
1: A orla é inicializada com o vértice origem. O seu peso é 0 pois é o peso do caminho conhecido. O único vértice da orla é o que será escolhido para ser processado.



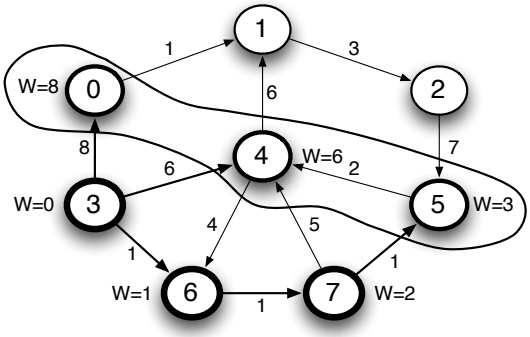
2: Os vértices adjacentes a 3 são adicinados à orla. O peso de cada um obtem-se somando ao peso de 3 (0) ao peso das arestas que os ligam a 3. O próximo vértice a saír da orla será o vértice 6.



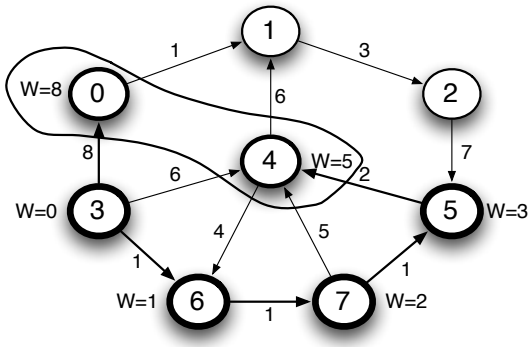
3: O vértice 7 é o único adjacente a 6 eé adicionado à orla. O seu peso calcula-se como o peso de 6 mais o da aresta que liga 6 a 7. O próximo vértice a saír da orla será o vértice 7.



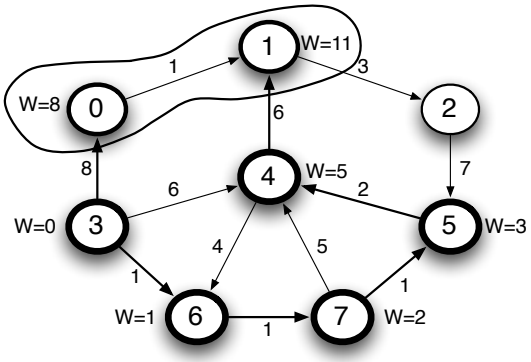
4: Dos vértices adjacentes a 7, o vértice 5 ainda não pertence à orla e por isso é adicionado com peso 3; o vértice 4 já pertence à orla mas o seu peso tem que ser actualizado uma vez que o caminho por que inclui o vértice 7 é de menor custo. O próximo vértice a saír da orla será o vértice 5.



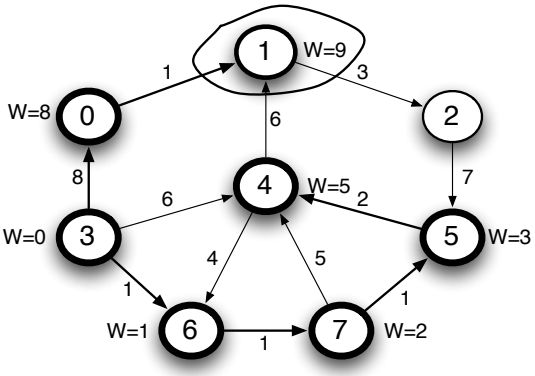
5: O único adjacente ao vértice 5 é o vértice 4 que já está na orla. No entanto o seu peso terá de ser actualizado. O próximo vértice a saír da orla será o vértice 4.



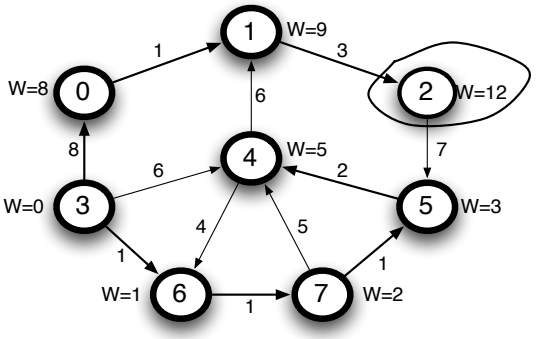
6: O único adjacente ao vértice 4 é o vértice 4 que é acrescentado à orla. O próximo vértice a saír da orla será o vértice 0.



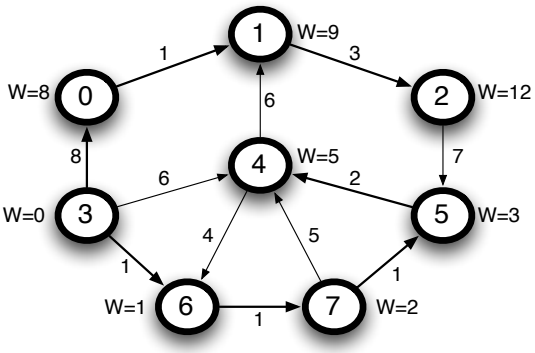
7: O único adjacente ao vértice 0 é o vértice 4 que já está na orla. No entanto o seu peso terá de ser actualizado. O próximo vértice a saír da orla será o único vértice da orla.



8: O único adjacente ao vértice 1 é o vértice 2 que é acrescentado à orla. O próximo vértice a saír da orla será o único vértice da orla.



9: O vértice 2 não tem adjacentes por processar. A orla fica vazia pelo que o algoritmo termina.



A grande diferença que existe entre este algoritmo e o de Prim é a forma como os pesos dos vértices da orla são calculados. Enquanto que no algoritmo de Prim este peso era o

peso de uma aresta, no algoritmo de Dijkstra o peso é o custo de um caminho desde a origem em causa.

```
int dijkstraSP (Graph g, int v, int cost[], int ant[]){
    int res=0;
    int newcost;
    EList it;
    int color[NV];
    struct fringe ff, *f;
    int fringesize;
    f = &ff;
    for (v=0;v<NV;color[v++] = 0)
        ;
    initFringe(f); fringesize=0;
    color[v]=1; cost[v]=0;
    addEdgeFringe(f,v,cost); fringesize++;
    while (fringesize>0) {
        v = getEdge (f,cost); fringesize--;
        res++;
        color[v]=2;//BLACK
        for (it=g[v]; it; it=it->next){
            newcost = cost[v] + it->cost;
            if ((color[it->dest] == 0)          // WHITE
                || ((color[it->dest] == 1) && // GREY
                      cost[it->dest] > newcost)) {
                ant[it->dest]=v;
                cost[it->dest]=newcost;
                if (color[it->dest] == 0) {  // WHITE
                   addEdgeFringe(f,it->dest,cost);
                   fringesize++;
                } else updateFringe(f,it->dest,cost);
            }
          }
    }
    return res;
}
```

Exemplo 21 (continuação) Vejamos então a evolução dos arrays color, cost e ant no exemplo apresentado.

1.

	0	1	2	3	4	5	6	7
color =	0	0	0	1	0	0	0	0
cost =	?	?	?	0	?	?	?	?
ant =	-1	-1	-1	-1	-1	-1	-1	-1

2.

	0	1	2	3	4	5	6	7
color =	1	0	0	2	1	0	1	0
cost =	8	?	?	0	6	?	1	?
ant =	3	-1	-1	-1	3	-1	3	-1

3.

	0	1	2	3	4	5	6	7
color =	1	0	0	2	1	0	2	1
cost =	8	?	?	0	6	?	1	2
ant =	3	-1	-1	-1	3	-1	3	6

4.

	0	1	2	3	4	5	6	7
color =	1	0	0	2	1	1	2	2
cost =	8	?	?	0	6	3	1	2
ant =	3	-1	-1	-1	3	7	3	6

5.

	0	1	2	3	4	5	6	7
color =	1	0	0	2	1	2	2	2
cost =	8	?	?	0	5	3	1	2
ant =	3	-1	-1	-1	5	7	3	6

6.

	0	1	2	3	4	5	6	7
color =	1	1	0	2	2	2	2	2
cost =	8	11	?	0	5	3	1	2
ant =	3	4	-1	-1	5	7	3	6

7.

	0	1	2	3	4	5	6	7
color =	2	1	0	2	2	2	2	2
cost =	8	9	?	0	5	3	1	2
ant =	3	0	-1	-1	5	7	3	6

8.

	0	1	2	3	4	5	6	7
color =	2	2	1	2	2	2	2	2
cost =	8	9	13	0	5	3	1	2
ant =	3	0	1	-1	5	7	3	6

9.

	0	1	2	3	4	5	6	7
color =	2	2	2	2	2	2	2	2
cost =	8	9	13	0	5	3	1	2
ant =	3	0	1	-1	5	7	3	6

As implementações que referimos para a orla na apresentação do algoritmo de Prim são as mesmas que se usam para este algoritmo.
Também em termos de complexidade os dois algoritmos são similares. Mesmo com a pior das implementações da orla (array ordenado) este algoritmo tem um comportamento no pior caso melhor do que o de Bellman-Ford. A principal limitação deste algoritmo, tal como referimos no início é não poder ser aplicado a grafos com pesos negativos.

Exercício 23 Apresente a evolução do algoritmo de Dijkstra quando aplicado sobre um grafo com pesos negativos. Use como exemplo o grafo do exemplo 20. Note que o resultado obtido é diferente (e errado) do do obtido nessa altura.

5.3 Algoritmo de Floyd-Warshal

O problema de determinar o caminho mais curto entre todos os pares de vértices pode ser resolvido por sucessivas invocações do problema abordado atrás. Cada uma dessas invocações determina os caminhos mais curtos com origem num dos vértices do grafo.

- Se o grafo não tiver custos negativos pode-se usar o algoritmo de Dijkstra, que na sua versão mais eficiente tem uma complexidade no pior caso de $\mathcal{O}(V*\log(V)+E)$. O custo desta alternativa será então de

$$\mathcal{O}(V*(V*\log(V)+E)) = \mathcal{O}(V^2*\log(V)+V*E)$$

- No caso de existirem arestas com pesos negativos pode-se usar o algoritmo de Bellman-Ford cuja complexidade no pior caso é $\mathcal{O}(V^2+V*E)$. O custo desta alternativa será então de

$$\mathcal{O}(V*(V^2+V*E)) = \mathcal{O}(V^3+V^2*E)$$

Uma terceira alternativa consiste em usar uma variante do algoritmo de Warshal apresentado atrás (pag. 42) agora para grafos pesados.
O princípio deste algoritmo é similar ao apresentado na altura. Também aqui se trata de um exemplo de programação dinâmica em que ao longo das várias iterações se preserva a validade da seguinte propriedade (invariante):

> Em cada iteração do ciclo estão calculados os caminhos mais curtos em G que usam **como nodos intermédios** apenas os vértices já processados.

O processamento de um vértice x consiste em analisar todos os caminhos em que x é um vértice intermédio.

Se existirem (já calculados) um caminho de u até x com peso p_u e um caminho de x até v com peso p_v, existe um caminho de u até v (passando por x) com peso $p_u + p_v$ que deve ser registado se ainda não tiver sido descoberto nenhum caminho de u até v ou se este custo fôr menor.

Tal como acontecia com o algoritmo de Warshall, o grafo dos caminhos deverá ser inicializado com o grafo original (caminhos sem vértices intermédios).

```
void floydwarshall (Graph g, GraphMat gp){
    EList it; int cost;
    int u, v, x;
    // inicialização de gp
    for (u=0; u<NV; u++) {
        for (v=0; v<NV; v++)
            gp[u][v] = NE;
        for (it=g[u]; it!=NULL; it=it->next)
            gp[v][it->dest] = it->cost;
    }
    //adição de arestas
    for (x=0; x<NV; x++)
        for (u=0; u<NV; u++)
            for (v=0; v<NV; v++)
                if (gp[u][x] != NE && gp[x][v] != NE){
                  cost = gp[u][x] + gp[x][v];
                  if (gp[u][v] == NE || gp[u][v] > cost)
                     gp[u][v] = cost;
                }
}
```

O ciclo de adição de arestas pode ser optimizado, evitando algumas iterações do ciclo mais interior.

```
    for (x=0; x<NV; x++)
        for (u=0; u<NV; u++)
            if (gp[u][x]!=NE)
            for (v=0; v<NV; v++)
                if (gp[x][v]!=NE){
                  cost = gp[u][x] + gp[x][v];
                  if (gp[u][v] == NE || gp[u][v] > cost)
                     gp[u][v] = cost;
                }
```

A função apresentada acima apenas calcula os custos dos caminhos mais curtos entre cada par de vértices. Um cálculo extra que pode ser incorporado sem aumentar a complexidade desta função diz respeito aos caminhos (sequência de vértices). Para isso usa-se uma matriz em que na posição (u, v) se guarda um de dois valores possíveis:

- Uma marca (tipicamente `-1` ou qualquer outro número negativo) que significa que o caminho mais curto entre u e v é composto por uma única aresta (não tem qualquer vértice intermédio)

- Um número x significando que o caminho mais curto entre u e v é a concatenação do caminho mais curto de entre u e x e entre x e v.

A definição abaixo resulta de acrescentarmos este cálculo na definição anterior.

```
void floydwarshall (Graph g, GraphMat gp, int path[NV][NV]){
    EList it; int cost;
    int u, v, x;
    // inicialização de gp e path
    for (u=0; u<NV; u++) {
        for (v=0; v<NV; v++)
            gp[u][v] = NE;
        for (it=g[u]; it!=NULL; it=it->next){
            gp[v][it->dest] = it->cost;
            path[v] = -1;
    }
    //adição de arestas
    for (x=0; x<NV; x++)
        for (u=0; u<NV; u++)
            for (v=0; v<NV; v++)
                if (gp[u][x] != NE && gp[x][v] != NE){
                  cost = gp[u][x] + gp[x][v];
                  if (gp[u][v] == NE || gp[u][v] > cost){
                     gp[u][v] = cost;
                     path[u][v]=x;
                }
}
```

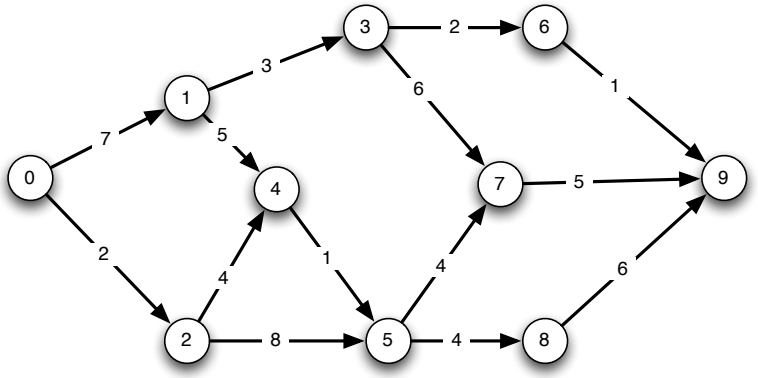
A complexidade desta função é fixada pelos três ciclos `for` e é por isso

$$T_{\text{floydwarshall}}(V, E) = \mathcal{O}(V^3)$$

Comparando com as outras alternativas para resolver este problema podemos constatar que este algoritmo é melhor do que a repetida chamada ao algoritmo de Bellman-Ford. Para grafos densos é ainda uma melhor opção à repetida invocação do algoritmo de Dijkstra. Para grafos esparsos esta solução é mais demorada do que o uso do algoritmo de Dijkstra. Não tem no entanto a limitação de só poder ser usada para grafos sem pesos negativos.

6 Caminho mais longo

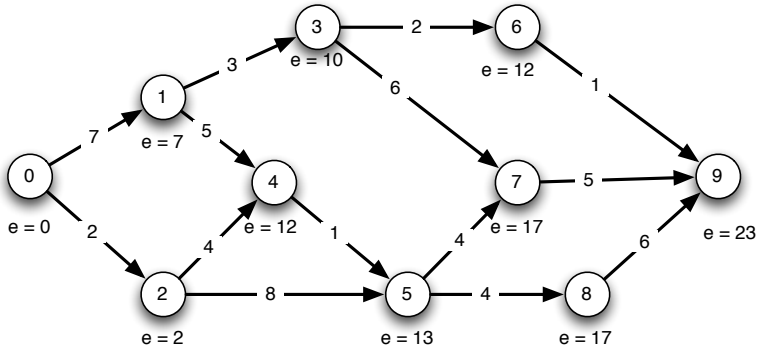
A determinação do caminho não cíclico mais longo de um grafo é, em geral um problema difícil (*NP-hard*).
No entanto tem uma solução linear se o grafo fôr orientado e acíclico (*Directed Acyclic Graph*).
Vejamos um exemplo. Seja G o seguinte grafo



A restrição de o grafo ser acíclico permite-nos obter uma ordenação topológica dos vértices do grafo.
No grafo acima, uma possível ordenação topológica dos vértices é:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

Esta ordenação pode ser usada para *escalonar* a ordem pela qual vamos analisar o grafo. Assim, e seguindo esta sequência, vamos calcular o caminho mais longo que contem apenas os vértices já visitados.
Para isso, e sempre que visitamos um nodo v, testamos se, para cada um dos seus sucessores x (e que, como o grafo é acíclico, ainda não foram visitados) o caminho correspondente ao maior caminho até v seguido de x é um caminho maior do que o descoberto até à altura.
Precisamos então de guardar, para cada nodo x o custo do maior caminho (até à altura) terminado em x; vamos denotar esse valor por e_x.
No final deste processo os custos calculados para cada vértice são:



Assumindo a existência de uma função que calcula uma ordenação topológica de um grafo, esta função pode ser definida como:

```
int longestPath (Graph g){
  int OT[N], e[N], i, r;
  EList it;
  ordTop (g,OT);
```

```
  for (i=0;i<N;e[i++]=0);
  for (i=0;i<N;i++){
    v = OT[i];
    for (it=g[v]; it!=NULL; it=it->next)
      if (e[v]+it->cost > e[it->dest])
        e[it->dest]=e[v]+it->cost;
  }
  r=0;
  for (i=0;i<N;i++)
    if (e[i] > r) r = e[i];
  return r;
}
```

Exercício 24 A função `longestPath` calcula o custo do caminho mais longo num grafo (orientado e acíclico). Adapte-a de forma a calcular qual é o caminho (sequência de vértices) mais longo.
Sugestão: Use um vector de antecessores para poder reconstruír o caminho.

Exercício 25 A função `longestPath` calcula, para cada vértice o custo do caminho mais longo de um dos vértices iniciais (sem antecessores) a esse vértice.
Adapte esta função para, nas mesmas condições (i.e., num grafo orientado e acíclico) determinar o peso do caminho mais curto entre dois vértices em tempo linear.

A Exercícios de exames

1. *(Exame de recurso, 2004/2005)*
   Pretende-se representar a informação referente a uma rede de metropolitano de uma cidade. As estações serão identificadas por um nome e as várias linhas associadas a cores. Note que duas estações podem ser ligadas por mais do que uma linha, o que revela que a estrutura a considerar deverá ser um *multigrafo não orientado*.

   (a) Defina as estruturas de dados adequadas para armazenar a informação referente ao multigrafo descrito. Considere para o efeito que as cores são o vermelho; amarelo; azul e verde.

   (b) Defina funções simples de manipulação das estruturas definidas, como sejam:
   - Inicialização da estrutura;
   - Adicionar uma estação;
   - Adicionar uma ligação entre duas estações por uma dada linha;

   (c) Defina uma função que determine quais as linhas que passam por uma dada estação.

   (d) Defina uma função que, dadas duas estações e uma cor, verifique se existe ligação entre as estações pela linha associada à cor dada.

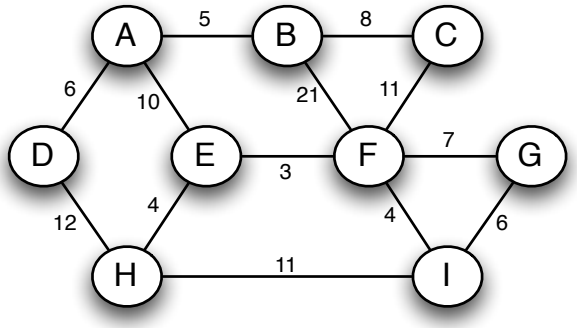
2. *(Teste, 2005/2006)*
   Uma forma de determinar se um grafo orientado é cíclico é tentar calcular uma ordenação topológica desse grafo. Se tal for possível o grafo é acíclico.

   (a) Escreva uma definição em C de uma função (representado em listas de adjacência) que testa se um grafo é ou não cíclico.

   (b) Descreva o tempo de execução da função apresentada em função do número de vértices e arestas do grafo.

   (c) Descreva ainda o tempo de execução desse mesmo algoritmo assumindo que o grafo estava representado em matrizes de adjacência.

3. *(Teste, 2005/2006)*
   Considere o seguinte grafo (ligado, pesado e não orientado).



(a) Mostre qual o comportamento do algoritmo de Dijkstra de cálculo do caminho mais curto de um vértice a todos os outros quando aplicado a este grafo a partir do vértice A. Nomeadamente, diga em cada iteração, qual o valor da fronteira e da árvore geradora construída até à altura.

(b) Mostre, usando por exemplo a árvore obtida na alínea anterior, que a árvore geradora construída pelo algoritmo de Dijkstra não é necessariamente uma árvore geradora de custo mínimo.

(c) Relembre que um dos resultados do algoritmo de Dijkstra é a árvore geradora dos caminhos mais curtos, representada por um vector de *ascendentes*. Defina uma função que, dado um destes vectores de ascendentes e um vértice imprima, caso exista, o caminho até esse vértice.

4. *(Exame de recurso, 2005/2006)*
Dado um grafo orientado, o algoritmo de Tarjan calcula as componentes fortemente ligadas do grafo em tempo linear no número de arestas e de vértices ($\Theta(E+V)$). A versão apresentada constrói essa informação num vector com uma componente por vértice cujo valor é o número de ordem da componente a que pertence esse vértice.

   (a) Defina uma função `void graphComp (Graph g, graph r)` que, dado um grafo orientado `g` , calcula um novo grafo `r` , com um vértice por cada componente ligada do grafo original e com uma aresta a ligar dois vértices sempre que exista um caminho no grafo original dos vértices de uma das componentes para a outra.

   (b) Descreva o tempo de execução desse algoritmo em função do número de arestas e vértices do grafo original.

5. *(Exame de recurso, 2005/2006)*
Considere o problema de determinar se um dado grafo orientado é ou não simétrico.

   (a) Defina em C uma função para resolver este problema. Apresente duas soluções, uma para cada uma das representações estudadas (matrizes e listas de adjacência).

   (b) Para cada uma das funções apresentadas atrás, descreva o tempo de execução $T(N,E)$ e a memória extra usada $M(N,E)$ em função do número de vértices (N) e de arestas (E).

6. *(Exame de recurso, 2005/2006)*
Dado um grafo não orientado, não pesado e ligado, o **diâmetro do grafo** define-se como o mais longo dos caminhos mais curtos do grafo (relembre que o comprimento de um caminho é o número de arcos desse caminho).

   (a) Usando os algoritmos estudados, apresente uma algoritmo para calcular o diâmetro de um grafo.

   (b) Baseado ainda nas análises de complexidade das várias componentes usadas na solução apresentada na alínea anterior, caracterize a complexidade dessa solução.

7. *(Época especial, 2005/2006)*
   Sejam $G_1 = \langle V_1, E_1 \rangle$ e $G_2 = \langle V_2, E_2 \rangle$ grafos. Uma função $f : V_1 \to V_2$ é um *homomorfismo de grafos* quando preserva adjacências (i.e. para quaisquer $x, y \in V_1$, temos que $(x, y) \in E_1 \Rightarrow (f(x), f(y)) \in E_2$). Se f possuir uma inversa que seja um homorfismo de grafos, então dizemos que f é um *isomorfismo de grafos*.

   Defina uma função polinomial que, dados dois grafos e um homomorfismo de grafos, determine se esse homomorfismo é um isomorfismo de grafos.

8. *(Época especial, 2005/2006)*
   O diâmetro de um grafo define-se como a maior distância entre dois quaisquer vértices. Por sua vez, a distância entre dois vértices define-se como o número mínimo de arestas dos caminhos entre esses vértices.

   (a) Apresente um algoritmo para resolver este problema.
   (b) Assumindo que cada vértice está ligado a no máximo outros 3, qual a complexidade máxima do algoritmo que apresentou na alínea anterior (em função do número de vértices do grafo).

9. *(Época especial, 2005/2006)*
   Um grafo G não pesado diz-se transitivo sse $G \circ G \subseteq G$.

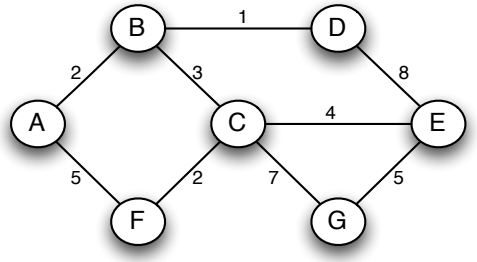
   (a) Defina uma função de composição de grafos.
   (b) Use (ou adapte) a função referida na alínea anterior para definir uma função que testa se um dado grafo é transitivo.

10. *(Teste, 2006/2007)*
    Assuma que a função `void dijkstraSP (Graph g, int v, int pai[], int dist[])` faz o cálculo dos caminhos mais curtos com origem num dado vértice, segundo o algoritmo de Dijkstra, num grafo pesado.

    Note que, dados o grafo `g` e o vértice de origem `v`, esta função guarda a árvore dos caminhos mais curtos em `pai`, e as respectivas distâncias em `dist`.

    Usando a função `dijkstraSP`, defina a função `int camMaisCurto (Graph g, int v, int d)` que dado o grafo `g`, o vértice origem `v` e o vértice destino `d`, imprime no `stdout` a sequência de vértices do caminho mais curto (da origem para o destino), um vértice por linha. Esta função devolve 1 se não houver caminho, e 0 se existir.

11. *(Exame de recurso, 2006/2007)*
    Considere o seguinte grafo:



Relembre o algoritmo de Prim para o cálculo da *Árvore Geradora de Custo Mínimo*. Ilustre as diversas etapas de execução deste algoritmo, sobre o grafo da figura. Identifique os conjuntos dos nós da árvore, dos nós da orla e dos arcos candidatos, ao longo da execução do algoritmo. Considere que o nó A é o primeiro nó a ser incluído na árvore.

12. *(Exame de recurso, 2006/2007)*
Relembre o algoritmo de Dijkstra para cálculo do caminho mais curto de um dado vértice para todos os outros, num grafo pesado e orientado (assuma que os pesos são números inteiros positivos). Assuma que a função `dijkstraSP`, definida como

```
dijkstraSP (Graph g, int v, int st[], int pesos[]);
```

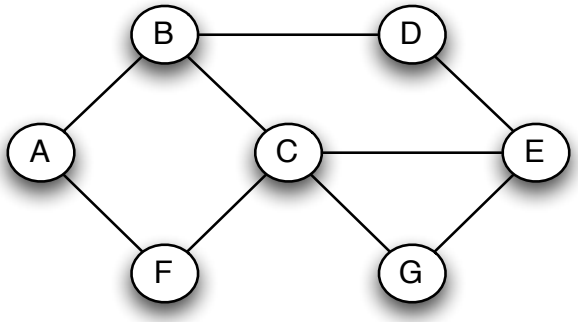
devolve nos vectores `st` e `pesos` a árvore construída pelo algoritmo e o peso do caminho mais curto para cada um dos vértices. Assuma ainda que o número de vértices é dado pela constante `NV`.

   (a) Considere que após se invocar a função `dijkstraSP`, se pretende determinar qual a aresta de maior peso usada. Escreva uma função que determine qual o peso dessa aresta.

   (b) Usando a função `dijkstraSP` escreva uma função que preencha uma matriz quadrada (`NV`×`NV`) com os pesos dos caminhos mais curtos entre todos os vértices (assuma que um peso de -1 corresponde a não existir caminho). Por outras palavras, defina a função de cálculo do fecho transitivo de um grafo pesado usando a função de caminho mais curto.

   (c) Assumindo que a função `dijkstraSP` executa em tempo $\Theta(\texttt{NV}\log(\texttt{NV}))$, determine um limite superior para o tempo de execução da função referida na alínea anterior.

13. *(Época especial, 2006/2007)*
Considere o seguinte grafo não pesado:



Desenhe uma árvore de antecessores passível de ser produzida pelo algoritmo de travessia em largura (breadth-first). Indique também a ordem porque os vértices são alcançados. Considere que o vértice E é utilizado como ponto de partida da travessia.

14. *(Época especial, 2006/2007)*
    Assuma que tem definida uma função `void travDF (Graph g, int pai [NV])` que implementa o algoritmo da questão anterior (onde `NV` é o número de vértices do grafo).

    O array preenchido por esta função pode ser usado para determinar o comprimento do caminho mais curto (i.e., com menos arestas) entre dois vértices do grafo. Defina a função `int distancia (Graph g, int origem, int destino)` que calcula esse comprimento, devolvendo `-1` caso não exista caminho.

15. *(Época especial, 2006/2007)*
    Suponha definidos o tipo `VERTICE` e uma função `Vert2Int` **injectiva** de conversão deste tipo de dados num número inteiro (sendo injectiva, quando aplicada a vértices diferentes dá resultados diferentes).

    Pretende-se armazenar um grafo pesado com `N` destes vértices usando tabelas de hash (open addressing) para armazenar os vértices e listas de adjacência para representar as arestas.

    (a) Defina os tipos de dados necessários, bem como a função de *hash* que, dado um vértice retorna o índice que lhe corresponde, usando a função `Vert2Int` como auxiliar. Note que esta função pode dar resultado fora da gama de definição da tabela. Note ainda que como não temos disponível qualquer função de igualdade entre vértices, esse teste tem de ser feito também usando a função `Vert2Int`.

    (b) Apresente definições de funções que permitam:
        i. Calcular o peso de uma aresta (se existir).
        ii. Calcular o número de antecessores de um vértice.
        iii. Calcular o número de vértices que não são acessíveis de um dado vértice.

16. *(Exame de recurso, 2006/2007)*
    Defina a função `int minPeso (Graph g)` que devolve o peso do arco de menor peso de um grafo.

17. *(Teste, 2007/2008)*
    Defina a função `int haCaminho(Graph g, int o, int d)` que determina se há caminho entre dois vértices.

18. *(Teste, 2007/2008)*
    Defina uma função (o mais eficiente possível) que receba a árvore resultado de uma travessia de um grafo (vector dos pais) e faça uma travessia pre-order dessa árvore (imprimindo os índices correspondente). Considere que o grafo tem `N` vértices e que a raíz da árvore tem pai `-1`.

19. *(Exame de recurso, 2007/2008)*
    Num grafo não orientado, um *click* é um subconjunto de vértices em que quaisquer dois elementos estão ligados por uma aresta.

    Defina uma função que dado um grafo e um conjunto de vértices determina se esse conjunto é um *click*.

20. (*Exame de recurso 2008/2009*)
Defina uma função `int maisLongo(Graph g, int o)` que, dado um grafo **orientado e acíclico** (representado em matriz de adjacências) e um vértice, calcula o comprimento do caminho mais longo que tem como origem esse vértice.

21. *(Exame de recurso, 2008/2009)*
Dados dois grafos R e S com o mesmo conjunto de vértices V, a composição dos grafos R e S, denota-se por $R \circ S$ e tem uma aresta de a para b sse existe um vértice c tal que a aresta $a \to c$ existe em S e a aresta $c \to b$ existe em R. Defina a função `void compoe (GraphMat r, GraphMat s, GraphMat ros)` que coloca no terceiro argumento o resultado da composição dos 2 primeiros grafos. Assuma que os grafos estão representados em matrizes de adjacência.

22. *(Exame de recurso, 2009/2010)*
Defina uma função `int valid_path(Graph g, int path[], int n)` que determina se o *array* `path` de dimensão `n` contém ou não uma sequência de nós correspondente a um caminho existente no grafo. Analise o tempo de execução da função que definiu.

23. *(Teste, 2010/2011)*
Defina uma função que calcula o inverso de um grafo (um grafo que tem uma aresta `i->j` se e só se existe uma aresta `j->i` no grafo original).

24. *(Teste, 2010/2011)*
Faça a análise da complexidade da função que apresentou na alínea anterior e justifique porque não é possível obter uma definição com uma complexidade assimptótica inferior a$\Theta(V + E)$ (em que V e E são respectivamente o número de vértices e de arestas do grafo).

25. *(Exame de recurso, 2010/2011)*
Considere a seguinte representação de um grafo por listas de adjacência:

```
A -> B, F
B -> A, C, G
C -> B, G, D
D -> C, E
E -> D, G, F
F -> A, E, G
G -> B, C, E, F
```

(a) Classifique o grafo de acordo com os seguintes critérios: orientado/não-orientado, pesado/não pesado, cíclico/acíclico, ligado/não ligado. Justifique todas as suas respostas.

(b) Qual o algoritmo (nome) que utilizaria para encontrar o caminho mais curto entre o nó A e o nó D? Apresente uma execução desse algoritmo para o grafo indicado.

26. *(Exame de recurso, 2010/2011)*
Por *capacidade total* de um vértice entende-se a diferença entre a capacidade de entrada (soma dos pesos de todos os arcos que se dirigem para o vértice) e a capacidade de saída (soma de todos os pesos de arcos que partem do vértice).

Defina uma função que calcula a capacidade total de todos os vértices do grafo. Certifique-se que a solução apresentada é eficiente, e refira qual a respectiva complexidade.

27. *(Exame de recurso, 2010/2011)*
Relembre o algoritmo de Dijkstra para calcular o caminho mais curto entre um vértice e todos os que dele são alcançáveis. Suponha que esse algoritmo está implementado na função `int dijkstraAll (Graph g, int o, int pais [], int pesos [])` que devolve ainda o número de vértices alcançáveis a partir do vértice dado.

   (a) Defina uma função `int maisLonge (Graph g, int v)` que calcula dentro dos vértices alcançáveis de `v`, aquele a que está a uma maior distância (considerando a distância entre dois vértices como o peso do caminho mais curto que os une).

   (b) Assumindo que a função `dijkstraAll` tem uma complexidade de $\mathcal{O}(V^2)$, qual a complexidade assimptótica da função que apresentou na alínea anterior? Justifique.

28. *(Época especial, 2010/2011)*
Em grafos não pesados a composição de grafos (com um mesmo conjunto de vértices) define-se como: existe uma aresta `i->j` em `apos(g,f)` sse para algum vértice `k` existem as arestas `i->k` em `f` e `k->j` em `g`.

   (a) Defina uma função que implemente a composição de grafos não pesados quando representados em matrizes de adjacência de inteiros.

   (b) Uma possível generalização deste conceito para grafos pesados consiste em dizer que o peso da aresta `i->j` em `apos(g,f)` é o mínimo da soma das arestas `i->k` em `f` e `k->j` em `g` para todos os vértices `k`.

   Defina uma função `void apos (GraphMat g, GraphMat f, GraphMat res)` que implemente a composição de grafos pesados quando representados em matrizes de adjacência de inteiros (considere que um peso negativo corresponde à ausência de aresta.

29. *(Época especial, 2010/2011)*

   (a) Defina uma função que calcule o número de antecessores de um dado vértice.

   (b) Defina ainda uma função que determina qual o vértice do grafo que tem mais antecessores. Note que esta função deve executar em $\mathcal{O}(V + E)$ em que V e E são respectivamente o número de vértices e arestas do grafo. Apresente uma análise do tempo de execução para justificar a sua resposta.

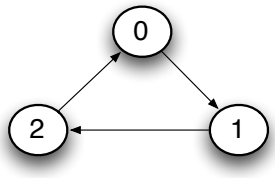
30. *(Teste, 2011/2012)*
Defina uma função `int colorOK (Graph g, int color[])` que, dado um grafo **não orientado** `g` e um vector de inteiros `cor` verifica se essa coloração é válida. Diz-se que uma coloração é válida sse vértices adjacentes tenham cores diferentes.

31. *(Exame de recurso, 2011/2012)*

(a) Defina uma função `int alcancaveis (Graph g, int o, int ant[])` que faz uma travessia do grafo `g` a partir de um vértice `o` e preenchendo um vector de antecessores para todos os vértices visitados.

(b) Defina uma função `EList caminho (Graph g, int o, int d)` que calcula um caminho no grafo `g` com origem em `o` e destino `d`. A função deve retornar uma lista ligada com os destinos das arestas do caminho (NULL caso tal caminho não exista).

32. *(Exame de recurso, 2011/2012)*
Dados dois grafos não pesados $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, uma função $f : V_1 \to V_2$ diz-se um homomorfismo de grafos sse para cada aresta $a \longrightarrow b$ do grafo G_1 existe uma aresta $f(a) \longrightarrow f(b)$ no grafo G_2. Por exemplo, a função `{0 -> 1, 1 -> 2, 2-> 0}` (que pode ser representada num array com `{1,2,0}`) é um homomorfismo entre o grafo apresentado abaixo e ele próprio (de facto é um isomorfismo, uma vez que é uma função bijectiva).



Defina uma função `int homoGraph (Graph g1, Graph g2, int f[])` que testa se a função representada no vector `f` é um homomorfismo entre os grafos `g1` e `g2` (assuma que o conjunto dos vértices é o mesmo). Apresente uma análise da complexidade assimptótica dessa função.

33. *(Época especial, 2011/2012)*
Defina uma função `int naoalcancavel (Graph g, int o)` que calcula, caso exista, um vértice que **não seja** alcançável a partir de `o`. A função deverá retornar `-1` caso tal vértice não exista. (Sugestão: use uma travessia do grafo).

34. *(Teste, 2012/2013)*
Usando uma travessia, defina uma função `int succN (Graph g, int v, int N)` que, dado um grafo `g`, um vértice `v` e um inteiro `N`, determina quantos vértices em `g` estão a uma distância de `v` menor ou igual a `N`, i.e., para os quais existe um caminho com `N` ou menos arestas.

35. *(Teste, 2012/2013)*
Em alguns algoritmos sobre grafos (Prim, Dijkstra) uma optimização possível consiste em implementar a orla com uma fila de prioridades. Uma forma de se implementar essa estrututa consiste em organizar os vértices numa *min-heap* ordenada pelo custo de cada vértice. Contudo, como o custo de cada vértice pode mudar, precisamos ainda de ter acesso à posição de cada vértice na heap, e implementar operações de actualização do custo de um vértice, com a consequente mudança da posição desse vértice na *heap*.

Considere então as seguintes definições para implementar uma fila de prioridades dos vértices de um grafo:

```
int custo[MaxV];    // custo de cada vértice
int posicao[MaxV];  // posição na heap
int heap[MaxV];
int heapsize;       // número de elementos na heap;
```

Por exemplo, se a orla for composta pelos vértices 1, 2, 4, 7 e 11, com custos (respectivamente) 6, 3, 5, 2 e 4, uma possível configuração destas variáveis seria:

	0	1	2	3	4	5	6	7	8	9	10	11	...
`custo` =		6	3		5			2				4	
`posicao` =		4	2		3			0				1	
`heap` =	7	11	2	4	1								

`heapsize` = 4

Defina a função `void updateCusto (int v, int c)` que altera o custo do vértice `v` para um custo menor `c`.

36. *(Exame de recurso, 2012/2013)*
Considere que se usa um grafo pesado e não orientado para representar uma rede de distribuição de água. Os vértices correspondem a bifurcações enquanto que os pesos das arestas correspondem à secção do tubo.

Defina uma função `int ligacao (Graph g, int v1, int v2, int seccao)` que, dado um grafo e dois vértices, determina se existe uma ligação entre esses dois vértices envolvendo apenas tubos com uma secção superior a um dado valor.

37. *(Época especial, 2012/2013)*
Considere um grafo com `101` vértices, numerados de 0 até 100, e onde existe uma aresta de `x` para `y` se e só se `x` é um divisor de `y`. Assuma que os sucessores de cada nodo se encontram armazenados **por ordem crescente**. Por exemplo, os sucessores do vértice 9 serão

`0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99`

por esta ordem.

Indique quais, e por que ordem, os vértices visitados ao efectuar uma travessia em profundidade a partir do vértice `10`.

38. *(Teste, 2013/2014)*
Diz-se que um grafo está *bem colorido* se nenhum arco liga dois vértices com a mesma cor, i.e., todos os pares de vértices adjacentes são pintados com cores diferentes.

Escreva uma função `int checkcoloring(Graph g, Colors c)` tão eficiente quanto possível, que verifica se um grafo satisfaz esta definição, devolvendo um valor booleano. Efectue depois a análise do seu tempo de execução no melhor e no pior caso, dizendo quando ocorre cada caso.

39. *(Teste, 2013/2014)*
Considere que se usa um grafo não orientado em que cada vértice corresponde a um país e as arestas correspondem a fronteiras terrestres: existe uma aresta de a para b (e logo de b para a) sse existe uma fronteira terrestre entre os países a e b.

Defina uma função `int Maior_cont (Graph)` que calcula o número de países do maior continente. Considere que um continente é um conjunto de países que têm fronteira terrestre e que o tamanho de um continente é o número de países desse continente.

40. *(Exame de recurso, 2014/2015)*
Defina em C uma a função `int pesoC (Graph g, int V[], int k)` que calcula o custo do caminho com `k` vértices (armazenados no vector `V`) no grafo `g`. Assuma que os vértices do vector correspondem realmente a um caminho, i.e., que existe uma aresta entre cada par de valores consecutivos do array.

41. *(Exame de recurso, 2014/2015)*
Relembre o algoritmo de Dijkstra para o cálculo de caminhos de menor peso em grafos pesado.

Apresente a evolução desse algoritmo quando é invocado, a partir do vértice 2, sobre o grafo com 5 vértices representado na matriz ao lado. O elemento da linha x coluna y representa o peso da aresta com origem x e destino y. Um peso `0` marca a inexistência de aresta. Na sua resposta deve apresentar os vários estados da orla bem como dos vectores de antecessores e pesos.

	0	**1**	**2**	**3**	**4**
0	0	0	1	2	0
1	7	0	0	0	2
2	9	1	0	0	0
3	1	1	5	0	0
4	3	0	0	1	0

42. *(Teste, 2015/2016)*
Um grafo $G = (V, E)$ diz-se **bi-partido** sse é possível determinar um sub-conjunto V_1 tal que para cada aresta $(a, b) \in E$, **exactamente uma** das extremidades (a ou b) pertence a V_1.

   (a) Defina uma função `int bipartition (Graph g, int V1[])` que recebe como argumentos um grafo e um array de booleanos (que representa um conjunto de vértices: um vértice `x` pertence ao conjunto sse `V1[x]==1`) e que testa se para cada aresta do grafo exactamente uma das suas extremidades pertence ao conjunto `V1`. A função deverá retornar `0` se existir uma aresta para a qual tal não se verificar.

   (b) Defina uma função `int bipartido (Graph g, int V1[])` que testa se um grafo é bipartido. A função deverá retornar 1 se tal acontecer e 0 no outro

caso. Em caso afirmativo, a função deverá ainda preencher o array `V1` com 1 ou 0 de forma a que o conjunto representado por esse array demonstre que o grafo é bi-partido (`V1[i]=1` significará que `i` pertence ao conjunto representado por `V1`).

Sugestão: Faça uma ou mais travessias de forma a visitar a totalidade dos vértices. Para cada vértice visitado teste se é possível inclui-lo no conjunto `V1`.

43. *(Teste, 2015/2016)*
Considere que ao executar o algoritmo de Floyd-Warshall (*all-pairs shortest paths*) sobre um grafo G com 5 vértices, as matrizes de pesos (p) e caminhos (c) produzidas são as que se apresentam ao lado.
Responda, justifiicando, às seguintes questões sobre o grafo G.

$c =$

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

$p =$

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

   (a) Qual o caminho mais curto (i.e., a sequência de vértices) que liga o vértice 0 ao vértice 3?

   (b) Qual o caminho mais curto (i.e., a sequência de vértices) entre os vértices mais distantes entre si?

   (c) Será o grafo cíclico?

44. *(Exame de recurso, 2015/2016)*
Considere que existe disponível uma função `int dijkstraSP (Graph g, int o, int pais[] int pesos[])` que calcula o caminho mais curto do vértice `o` para todos os vértices do grafo `g`. Essa função preenche os dois vectores: `pais` com a árvore dos caminhos (antecessores)e `pesos` com as respectivas distâncias. Considere que se `peso[x] == -1` após a invocação da função então o vértice `x` não é alcançável a partir de `o`.

   (a) Usando esta função, defina a função `int maisArestas (Graph g, int o)` que calcula o número de arestas do caminho mais curto que liga o vértice `o` ao que lhe é mais distante.

   (b) Admitindo que a função `dijkstraSP` executa em tempo $\Theta(V^2 + V.E)$ qual a complexidade assimptótica da função que definiu? Justifique.

45. *(Exame de recurso, 2015/2016)*
O algoritmo de travessia *depth-first* pode ser usado para determinar a existência de ciclos se o controlo dos vértices visitados for feito colorindo os vértices com 3 valores (VISITADO, NÃO VISITADO e EM VISITA). Sempre que um vértice adjacente ao vértice a visitar se encontra no estado EM VISITA, foi detectado um ciclo. Os vértices desse ciclo podem ser determinados através do array de antecessores.

Apresente uma definição de uma função `int ciclo (Graph g, int c[])` que determina se um grafo tem ciclos. Em caso afirmativo a função deverá retornar `1` e preencher o vector `c` com os vértices que compõe o ciclo encontrado.

46. *(Época especial, 2015/2016)*

    (a) Defina a função `int vizinhanca (Graph g, int v, int d)` que calcula o número de vértices do grafo que estão a uma distância de `v` (soma dos pesos do caminho) inferior a `d`.

    (b) Que alterações faria à solução apresentada se se tratasse de um grafo não pesado?

47. *(Exame de recurso, 2016/2017)*
    Defina uma função `maxcomp` que calcula o número de vértices da *maior* componente ligada de um grafo não orientado.

48. *(Exame de recurso, 2016/2017)*
    Suponha que um dado grafo `g` tem 15 elementos (`#define NV 15`) e que se executa o seguinte extracto de código:

```
int i; int pesos [NV], int pais [NV];
for (i=0;i<NV;pais [i++] = -2);

dijkstraSP (g,3,pais,pesos);
```

    Depois disso os arrays `pais` e `pesos` têm o seguinte conteúdo:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`pais` =	5	3	3	-1	-2	2	14	6	-2	1	1	-2	-2	-2	3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`pesos` =	20	4	3	0	2	9	8	13	20	5	8	13	5	42	2

    Indique, justificando, quais das seguintes afirmações são verdadeiras e quais são falsas. Considere que a distância entre dois vértices é o peso (soma dos pesos das arestas) do caminho mais curto que liga o primeiro ao segundo. *Sugestão: comece por desenhar a árvore calculada pela função* ***`dijkstraSP`***.

    (a) os vértices `0` e `8` estão à mesma distância do vértice `3`.

    (b) o vértice mais distante do vértice `3` é o vértice `0`.

    (c) a distância do vértice `3` a `1` é igual à distância do vértice `1` ao vértice `10`.

    (d) existe no grafo uma aresta que liga o vértice `9` ao vértice `5` com peso `3`.

49. *(Época especial, 2016/2017)*

Considere a definição usual de grafos usando listas de adjacência.

Defina uma função `int subMST (Grafo g, int o, int lim, int m[])` que, dado um grafo `g`, um vértice `o` e um custo `lim` calcula um subgrafo ligado e acíclico de `g` que contém o vértice `o` a soma dos custos das arestas é menor ou igual a `lim` e tem o maior número possível de vértices.

```
#define NV ...
typedef struct aresta {
   int destino, peso;
   struct aresta *prox;
} *Grafo[NV];
```

Sugestão: adapte o algoritmo de Prim de cálculo duma árvore geradora de custo mínimo.