Ficha 5

Algoritmos e Complexidade

Grafos

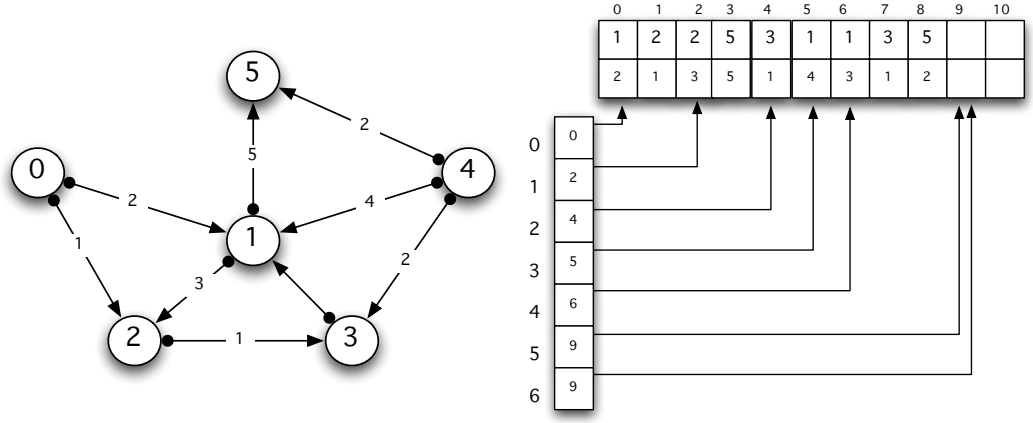
1 Representações

1. Apresente definições de tipos para representar grafos orientados e pesados (peso inteiro positivo) em cada uma das seguintes formas

   (a) Matrizes de adjacência (cada posição da matriz contem o peso da aresta).

   (b) Listas de adjacência.

   (c) Vectores de adjacência (para cada nodo guarda-se a posição num vector de adjacências onde começam os seus adjacentes, tal como é exemplificado abaixo).



2. Defina funções de conversão entre as três representações acima.

3. Para cada uma das representações defina funções de cálculo do grau de entrada e de saída de um vértice.

4. Analise o esforço computacional das funções apresentadas na alínea anterior.

5. 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 em C uma função que calcula a capacidade total de todos os vértices de um grafo representado em listas de adjacência. Certifique-se que a solução apresentada é eficiente, e refira qual a respectiva complexidade.

2 Travessias

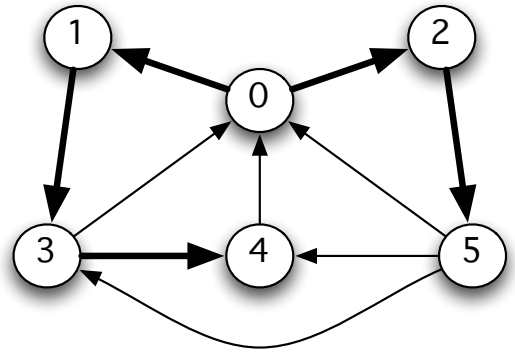
Considere a seguinte definição para representar (as arestas de) um grafo.

```
#define MaxV ...
#define MaxE ...

typedef struct edge {
     int dest;
     int cost;
     struct edge *next;
} Edge, *Graph [MaxV];
```

1. Uma árvore pode ser vista como um caso particular de um grafo: trata-se de um grafo ligado e acíclico. Numa árvore, cada vértice (com excepção da raiz) tem exactamente um antecessor. Daí que seja comum representar uma árvore num vector de antecessores (em que para cada vértice se guarda o índice do seu antecessor; na componente correspondente à raiz, guarda-se -1).

   No grafo que abaixo se apresenta, a árvore a *bold* é representada no vector da direita.



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

   Cada um destes vectores pode ainda ser usado para representar um conjunto (disjunto) de árvores, a que é costume referir-se por *floresta*.

   (a) Defina uma função `int raiz (int f[], int v)` que, dada uma floresta `f` e um vértice `v` determine (o índice d)a raiz da árvore a que o vértice pertence.

   (b) Usando a função anterior, defina as seguintes funções:

      i. `int inTree (int f[], int a, int b)` que testa se dois vértices estão na mesma árvore.

      ii. `void joinTree (int f[], int v1, int v2)` que, dada uma floresta `f` e dois vértices pertencentes a árvores distintas, junta essas árvores ligando a raiz da árvore de `v1` a `v2`.

   (c) Defina uma função `int custo (Graph g, int f[])` que calcula o custo total de uma árvore num grafo (a soma dos custos de todos os arcos). A função deve retornar 0 se algum dos ramos da árvore não for um ramo do grafo. Certifique-se que a função que definiu não tem uma complexidade assimptótica superior a $\mathcal{O}(V + E)$ em que V e E são os números de vértices e arestas do grafo.

(d) Defina uma função `int altura (int f[])` que calcula a altura de uma árvore representada num vector de antecessores. Analise o tempo de execução da função apresentada em função do número de vértices. Identifique o melhor e pior casos.

2. Considere a seguinte definição de uma procura *depth-first* (que testa se um vértice `d` é alcançável a partir de um vértice `o` num grafo `g`).

```
int DF_aux (Graph g, int o, int d, int v[]) {
    int r = 0;
    Edge *aux;
    v[o] = 1;
    if (o==d) r = 1;
    else for (aux=g[o]; (aux && !r); aux = aux->next)
             if (! v [aux->dest]) r = DF_aux (g,aux->dest,d,v);
    return r;
}

int DF_search (Graph g, int o, int d){
    int i, vis [MaxV];
    for (i=0;(i<MaxV); vis[i++] = 0);
    return (DF_aux(g,o,d,vis));
}
```

Apresente uma definição da função `int travessia_DF (Graph g, int o, int f[])` que percorre um grafo a partir de um dado vértice segundo uma estratégia *depth-first*. Esta função deverá retornar o número de vértices alcançáveis a partir do vértice dado. Deverá ainda preencher o vector `f[]` com a árvore correspondente à travessia (i.e., com as arestas usadas).

3. Dado um grafo orientado e acíclico, uma ordenação topológica é uma sequência de vértices `[v0, v1, v2,...]` em que cada vértice só aparece na sequência depois de todos os seus antecessores.

   Uma solução para resolver este problema de uma forma eficiente (Kahn, 1962) consiste em guardar, para cada vértice o número de antecessores que ainda não apareceu na ordenação. Assim, sempre que um vértice é adicionado à ordenação, decrementa-se o contador associado a cada um dos seus sucessores (sempre que esse contador passar a zero, pode ser acrescentado à ordenação).

   Apresente uma implementação deste algoritmo e analise o seu comportamento assimptótico no pior caso.

4. Qual o comportamento do algoritmo anterior para o caso de o grafo não ser acíclico? Use esse facto para definir uma função que testa se um grafo é acíclico.

3 Grafos Pesados

1. Considere a seguinte definição para representar (as arestas de) um grafo, bem como uma implementação do algoritmo de Prim para cálculo de uma árvore geradora de custo mínimo de um grafo não orientado, pesado e ligado.

```
#define MaxV ...
#define WHITE 0
#define GRAY  1
#define BLACK 2

typedef struct edge {
     int dest;
     int cost;
     struct edge *next;
} Edge, *Graph [MaxV];


int primMST (Graph g, int p[], int w[]) {
int i, v, r=0, fsize, col [MaxV];
Fringe f = newFringe (MaxV);

for (i=0;i<MaxV;i++){
   p[i] = -1; col [i] = WHITE;
  }
col [0] = GREY; w [0] = 0;
f = addV (f, 0, 0);
fsize=1;
while (fsize) {
   fsize--;
   f = nextF(f, &v);
   col [f] = BLACK;´
   r += w[v];
   for (x=g[v]; x; x = x->next)
       switch (col [x->dest]) {
       case WHITE: col [x->dest] = GREY;
                   fsize++;
                   f = addV (f, x->dest, x->cost);
                   w[x->dest] = x->cost; p[x->dest] = v;
                   break;
       case GREY : if (w[x->dest] > x->cost) {
                      f = updateV (f, x->dest, x->cost);
                      w[x->dest] = x->cost; p[x->dest] = v;
                   break;
       default   : break;
      }
  }
}
```

A função `primMST` baseia-se numa estrutura auxiliar – `Fringe` – para armazenar a orla. As funções necessárias sobre esta estrutura são `newFringe` (inicialização), `nextF` (remoção de um elemento), `addV` (adição de um elemento) e `updateV` (actualização do custo de um elemento). Considere como alternativas para implementar a orla: (A) Um vector com os pesos de cada vertice da orla, (B) uma lista com os vértices ordenados pelo seu peso, (C) uma *minheap* dos vértices ordenada pelo peso. Para cada uma destas alternativas,

(a) Apresente definições das funções referidas.

(b) Analise a complexidade (pior caso) da função `primMST` resultante.

2. O algoritmo de Dijkstra para cálculo do caminho mais curto entre dois vértices (ou mais genéricamente, para o cálculo dos caminhos mais curtos de um vértice até todos os outros) segue uma estratégia semelhante à do algoritmo de Prim, diferindo apenas no significado dos pesos atribuídos aos vértices da orla.

   Apresente uma implementação deste algoritmo, bem como o invariante que descreve o significado dos pesos dos elementos da orla.

   ```
   int dijkstraSP (Graph g, int o, int p[], int w[])
   ```

3. Um algoritmo alternativo ao de Prim para cálculo de uma árvore geradora de custo mínimo deve-se a Joseph. B. Kruskal (1956) e pode ser descrito como *seleccionar as (n-1) arestas de menor peso do grafo que não formam ciclos*. Para isso começa-se por construir uma floresta com uma árvore (unitária) para cada vértice. De seguida vão-se acrescentando arestas por ordem crescente do seu peso (que unam árvores disjuntas, evitando assim os ciclos) juntando as respectivas árvores numa só.

   Apresente uma implementação deste algoritmo e analise o seu comportamento assimptótico no pior caso. Tenha especial atenção à estratégia usada para a escolha da aresta de menor peso.

4. O fecho transitivo de um grafo $G = (V, E)$ é um grafo com o mesmo conjunto V de vértices e com ma aresta $a \longrightarrow b$ se e só se existe em G um caminho de a até b.

   (a) Uma alternativa para o cálculo do fecho transitivo de um grafo consiste em efectuar uma travessia a partir de todos os vértices do grafo, marcando como adjacentes a um vértice no fecho transitivo todos os vértices alcançáveis no grafo a partir desse vértice.

   Defina uma função em C que calcule o fecho transitivo de um grafo usando esta estratégia.

   Qual a complexidade da função?

   (b) Uma outra estratégia (Warshal, Floyd e Roy, 1962), percorre os vértices V do grafo mantendo o seguinte invariante:

   > O grafo resultado contem uma aresta $a \longrightarrow b$ desde que exista em G um caminho de a até b cujos vértices intermédios sejam vértices já visitados.

   Defina uma função que calcule o fecho transitivo de um grafo, representado numa matriz de adjacências, tal como descrito acima.

   Qual a complexidade dessa função?

   Qual a complexidade dessa função se o grafo estivesse representado em listas de adjacência?

5. A generalização do fecho transitivo para grafos pesados, define-se usando para peso de cada aresta $a \longrightarrow b$ o custo do caminho mais curto entre a e b. Generalize os algoritmos apresentados atrás para grafos pesados.

4 Caminho Crítico

Considere-se que, para a execução de um determinado projecto se identificam:

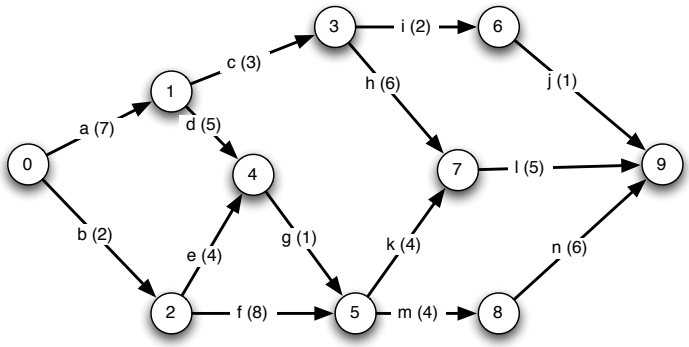
1. a lista das tarefas do projecto,
2. a duração prevista para cada tarefa,
3. as dependências entre as várias tarefas.

Pretende-se determinar qual a duração global do projecto, bem como quais as tarefas críticas, i.e., as tarefas que determinam esta duração global.
Uma forma de resolver este problema consiste em modelá-lo como um grafo (pesado e acíclico) em que:

- as arestas correspondem às tarefas (pesadas pela duração da tarefa),
- os vértices correspondem aos pontos de sincronismo, i.e., servem para explicitar as dependências entre tarefas: se uma tarefa t_1 tem que ser feita antes da tarefa t_2 então a aresta correspondente a t_1 tem como destino o vértice origem da tarefa t_2.

O grafo que se apresenta à direita resulta da informação presente na tabela da esquerda.

Tarefa	**duração**	**depois de**
a	7	
b	2	
c	3	a
d	5	a
e	4	b
f	8	b
g	1	e, d
h	6	c
i	2	c
j	1	i
k	4	f, g
m	4	f, g
n	6	m



A forma de resolver este problema consiste em precorrer os vértices do grafo, duas vezes, segundo uma ordenação topológica.

1. da primeira travessia, calcula-se para cada vértice, o mais cedo que cada uma das tarefas que nele se iniciam pode começar.

   $$e_i = \max_{j \in \text{ant}(i)} (e_j + w(i,j))$$

   Em que $w(i,j)$ corresponde ao peso da aresta $i \longrightarrow j$ e $e_0 = 0$.

2. Na segunda travessia, por ordem inversa à primeira, calcula-se para cada vértice o mais tarde que cada uma das tarefas que aí terminam pode acabar.

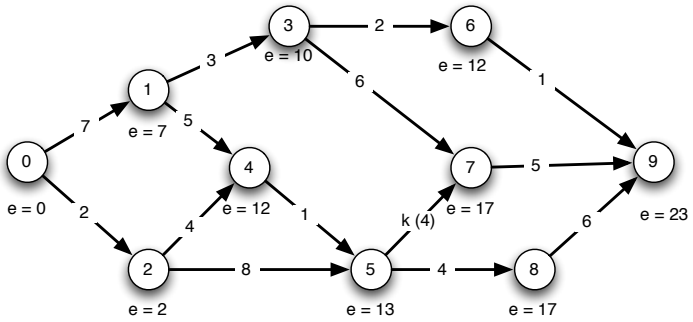
   $$l_i = \min_{j \in \text{suc}(i)} (l_j - w(i,j))$$

   Em que, para o último vértice v_n, $l_n = e_n$

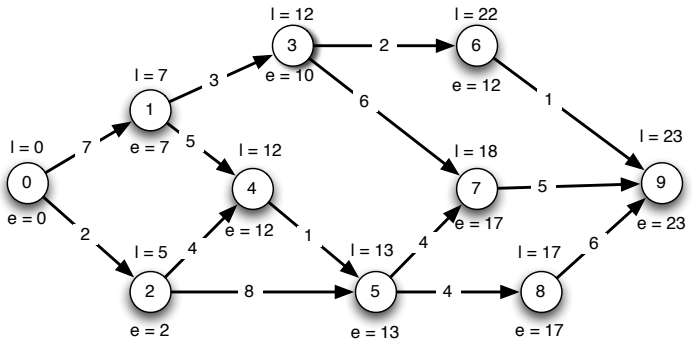
Uma ordenação topológica possível do grafo acima será:

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

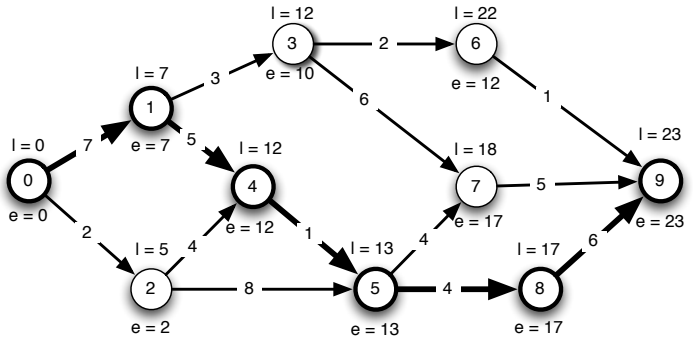
No final da primeira etapa do algoritmo (cálculo dos e_i) teremos



No final da segunda fase (cálculo dos l_i) teremos



Pelo que as tarefas críticas são as que se apresentam realçadas no seguinte grafo.



Correspondendo a um tempo total do projecto (caminho mais longo) de **23**.

Apresente uma definição em C de uma função `int pert (Graph g, int v[])` que calcula o caminho mais longo de um grafo não pesado e acíclico (devolvendo a sequência de vértices no vector `v[]` e o peso do caminho como resultado da função), de acordo com a estratégia apresentada.