Ficha 4

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 contém o peso da aresta)

   (b) Listas de adjacência

2. Defina funções de conversão entre as duas 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. A *capacidade* de um vértice `v` num grafo define-se como a diferença entre a soma dos pesos das arestas que têm `v` como destino e a soma dos pesos das arestas que têm `v` como origem. Defina uma função `int capacidade (Grafo g, int v)` que calcula a capacidade de um vértice num grafo.

   Analise o esforço computacional dessa função.

6. Defina uma função `int maxCap (Grafo g)` que calcula o vértice do grafo com maior capacidade. Certifique-se que a solução apresentada é eficiente, e refira qual a respectiva complexidade.

7. 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 tiverem cores diferentes.

8. Defina em C 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). Apresente definições para as diferentes representações de grafos descritas acima.

2 Travessias

1. 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.

   (a) Uma solução para resolver este problema de uma forma eficiente, devida a Tarjan, baseia-se numa travessia em profundidade, e na observação de que um vértice deve ser colocado na ordenação antes de todos os seus descendentes. Basta então preencher a ordenação topológica por ordem inversa, colocando os vértices no momento em que se termina todas as travessias iniciadas nos seus adjacentes (i.e. no momento em que o vértice passa a "preto").

   (b) Uma outra solução (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.

   (c) Qual o comportamento destes algoritmos no caso de o grafo conter ciclos? Note que a noção de ordenação topológica só faz sentido em grafos acíclicos.

   (d) Analise o comportamento assimptótico no pior caso de ambos os algoritmos.

   (e) Defina, com base em cada um dos algoritmos, uma função que testa se um grafo é acíclico.

2. Usando uma travessia, defina uma função `int succN (Grafo 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.

3. 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 (Grafo 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.

4. 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)

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. 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?

4. 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.

5. Defina uma função `int quantosD (Grafo g, int v, int N)` que, dado um grafo `g`, um vértice `v` e uma distância `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 desde `v` com peso não superior a `N`.

4 Problemas

Nesta secção apresentam-se alguns problemas (típicos de concursos de programação) cuja resolução pode ser feita usando grafos.

1. O objectivo deste problema é determinar o tamanho do maior continente de um planeta. Considera-se que pertencem ao mesmo continente todos os países com ligação entre si por terra. Irá receber uma descrição de um planeta, que consiste numa sequência de linhas, onde cada linha lista uma sequência de países que são vizinhos entre si. O programa deverá devolver o tamanho do maior continente do planeta.

   Exemplo de entrada:

   ```
   portugal espanha
   espanha franca
   franca belgica alemanha
   franca suica italia
   belgica holanda alemanha
   inglaterra irelanda
   suica austria italia
   alemanha polonia
   brasil uruguai paraguai
   brasil peru paraguai
   brasil colombia venezuela
   colombia equador peru
   islandia
   ```

   Saída correspondente:

   ```
   10
   ```

2. O objectivo deste problema é determinar a área interna de uma figura desenhada com caracteres. A entrada consistirá nas coordenadas (horizontal e vertical, medidas a partir do canto superior esquerdo) de um ponto arbitrário dentro de uma figura. O limite da figura estará desenhado usando asteriscos. A área interna consiste no número de caracteres contidos dentro da figura. Assuma que a figura ocupa no máximo 100 por 100 caracteres.

   Exemplo de entrada:

   ```
   3 2
     ***
    *   *
   *     *
    *   *
     ***
   ```

   Note que neste caso o ponto inicial é coincidentemente o ponto central da imagem.

   Saída correspondente:

   ```
   11
   ```

3. Podemos usar um (multi) grafo pesado para representar um mapa de uma cidade: cada nó representa um cruzamento e cada aresta uma rua (etiquetada com o respectivo

comprimento). O objectivo deste problema é listar os cruzamentos de uma cidade por ordem crescente de criticidade: um cruzamento é tão mais crítico quanto o número de ruas que interliga. A entrada consistirá numa sequência (não vazia) de nomes de ruas de uma única palavra, com no máximo 100 minúsculas. Os identificadores dos cruzamentos correspondem a uma letra do alfabeto, e cada rua começa (e acaba) no cruzamento identificado pelo primeiro (e último) caracter do respectivo nome. A saída deverá listar os nomes dos cruzamentos por ordem crescente de criticidade, listando cada linha um cruzamento e o número de ruas que interliga. Apenas deverão ser listados os cruzamentos que interliguem alguma rua, e os cruzamentos com o mesmo nível de criticidade deverão ser listados por ordem alfabética.

Exemplo de entrada:

```
raio
central
liberdade
chaos
saovictor
saovicente
saodomingos
souto
capelistas
anjo
taxa
```

Saída correspondente:

```
t 1
a 2
e 2
l 2
r 2
c 3
o 3
s 6
```

4. Podemos usar um (multi) grafo pesado para representar um mapa de uma cidade: cada nó representa um cruzamento e cada aresta uma rua (etiquetada com o respectivo comprimento). O objectivo deste problema é determinar o tamanho de uma cidade: a distância entre os seus cruzamentos mais afastados. A entrada consistirá numa sequência (não vazia) de nomes de ruas de uma única palavra, com no máximo 100 minúsculas. Os identificadores dos cruzamentos correspondem a uma letra do alfabeto, e cada rua começa (e acaba) no cruzamento identificado pelo primeiro (e último) caracter do respectivo nome. O comprimento de uma rua é o tamanho do seu nome. A saída deverá consistir no tamanho do maior caminho mais curto entre quaisquer dois cruzamentos da cidade.

   Exemplo de entrada:

   ```
   raio
   central
   liberdade
   ```

```
chaos
saovictor
saovicente
saodomingos
souto
capelistas
anjo
taxa
```

Saída correspondente:

```
25
```

Este resultado corresponde ao tamanho do caminho

```
taxa
anjo
souto
chaos
central
```

5. O objectivo deste problema é encontrar o caminho através de um labirinto. Irá receber um inteiro positivo N e um labirinto quadrado de dimensão N cujas paredes estão desenhadas com o caracter `#`. A entrada do labirinto é na posição (0,0) e saída na posição (N-1,N-1). Apenas se poderá movimentar para cima, para a direita, para baixo ou para a esquerda. Deverá imprimir o número mínimo destes movimentos necessários para chegar da entrada à saída. Em todos os labirintos há sempre caminho entre a entrada e saída.

   Exemplo de entrada:

   ```
   10
     ########
   # # #    #
   # # #### #
   # #      #
   # # # ####
   # # #    #
   #   # #  #
   ##### ####
   #        #
   ########
   ```

   Saída correspondente:

   ```
   24
   ```

6. O objectivo deste problema é determinar quantos movimentos são necessários para movimentar um cavalo num tabuleiro de xadrez entre duas posições. Irá começar por receber a dimensão do tabuleiro de xadrez, seguida de uma sequência de pares de coordenadas, dois pares de coordenadas por linha, que identificam a origem e destino pretendido. Para cada par de coordenadas deverá imprimir o número de movimentos que é necessário para atingir o destino a partir da origem.

Exemplo de entrada:

```
8
4 4 6 5
0 0 1 1
0 0 7 7
```

Saída correspondente:

```
1
4
6
```

7. O objectivo deste problema é determinar o preço mais barato para voar entre 2 aeroportos. Irá receber o código de 2 aeroportos entre os quais pretende viajar, seguida de uma sequência de itinerários disponíveis. Cada itinerário consiste nos códigos dos aeroportos e do respectivo preço. Assuma que o preço é o mesmo nos dois sentidos. Deverá imprimir o custo da viagem mais barata entre os 2 aeroportos pretendidos. Em todas os testes essa viagem será sempre possível.

   Exemplo de entrada:

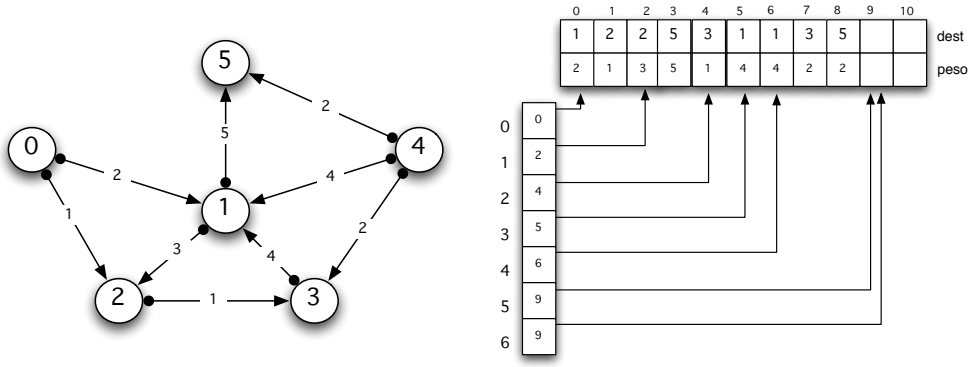
   ```
   FRA NRT
   OPO LIS 100
   OPO FAO 70
   LIS FAO 100
   MAD OPO 200
   LIS LON 300
   FRA OPO 300
   LIS NRT 1200
   LON NRT 800
   LON FRA 200
   LIS FRA 300
   ```

   Saída correspondente:

   ```
   1000
   ```

5 Exercícios Adicionais

1. Na representação de grafos orientados e pesados (peso inteiro positivo) por *Vectores de adjacência*, para cada vértice guarda-se a posição num vector de adjacências onde começam os seus adjacentes, tal como é exemplificado abaixo.



(a) Apresente definições de tipos apropriadas para esta representação.

(b) Defina funções de conversão entre esta representação e as da Secção 1.

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

(d) Analise o esforço computacional das funções apresentadas na alínea anterior.

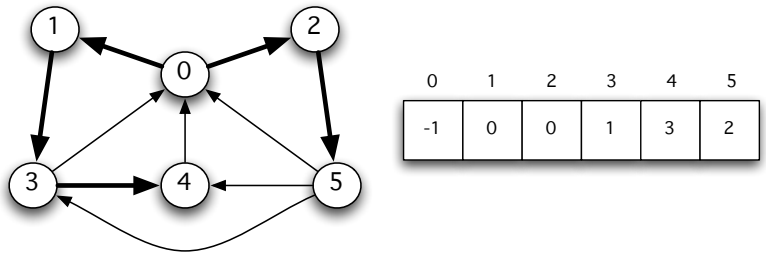
2. Considere a seguinte definição para representar (as arestas de) um grafo.

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

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

Note que 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.



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

(a) 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).

(b) 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. Usando esta função, defina as seguintes:

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.

3. 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.

4. Num grafo não orientado e ligado, a **excentricidade** de um vértice define-se como a maior distância entre esse vértice e qualquer outro vértice (no caso de grafos não pesados, a distância entre dois vértices corresponde ao número de arestas do caminho mais curto entre esses vértices). Usando uma variante do algoritmo de travessia *breadth-first*, defina uma função `int excentricity (Grafo g, int v)` que calcula a excentricidade de um vértice num grafo não pesado.

5. 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.