Ficha 4

Algoritmos sobre Grafos

Algoritmos e Complexidade
LEI / LCC / LEF

1 Representações

Considere os seguintes tipos para representar grafos.

```
#define NV ...

typedef struct aresta {
int dest; int custo;
struct aresta *prox;
} *LAdj, *GrafoL [NV];

typedef int GrafoM [NV][NV];
```

Para cada uma das funções descritas abaixo, analize a sua complexidade no pior caso.

1. Defina a função `void fromMat (GrafoM in, GrafoL out)` que constrói o grafo `out` a partir do grafo `in`. Considere que `in[i][j] == 0` sse **não existe** a aresta $i \longrightarrow j$.

2. Defina a função `void inverte (GrafoL in, GrafoL out)` que constrói o grafo `out` como o inverso do grafo `in`.

3. O grau de entrada (saída) de um grafo define-se como o número máximo de arestas que têm como destino (origem) um qualquer vértice.

   Defina a função `int inDegree (GrafoL g)` que calcula o grau de entrada do grafo.

4. Uma coloração de um grafo é uma função (normalmente representada como um array de inteiros) que atribui a cada vértice do grafo a sua *côr*, de tal forma que, vértices adjacentes (i.e., que estão ligados por uma aresta) têm cores diferentes.

   Defina uma função `int colorOK (GrafoL g, int cor[])` que verifica se o array `cor` corresponde a uma coloração válida do grafo.

5. Um homomorfismo de um grafo `g` para um grafo `h` é uma função f (representada como um array de inteiros) que converte os vértices de `g` nos vértices de `h` tal que, para cada aresta $a \longrightarrow b$ de `g` existe uma aresta $f(a) \longrightarrow f(b)$ em `h`.

   Defina uma função `int homomorfOK (GrafoL g, GrafoL h, int f[])` que verifica se a função `f` é um homomorfismo de `g` para `h`.

2 Travessias

Considere as seguintes definições de funções que fazem travessias de grafos.

```
int DF (GrafoL g, int or,
        int v[],
        int p[],
        int l[]){
    int i;
    for (i=0; i<NV; i++) {
        v[i]=0;
        p[i] = -1;
        l[i] = -1;
    }
    p[or] = -1; l[or] = 0;
    return DFRec (g,or,v,p,l);
}
int DFRec (GrafoL g, int or,
        int v[],
        int p[],
        int l[]){
    int i; LAdj a;
    i=1;
    v[or]=-1;
    for (a=g[or];
         a!=NULL;
         a=a->prox)
      if (!v[a->dest]){
         p[a->dest] = or;
         l[a->dest] = 1+l[or];
         i+=DFRec(g,a->dest,v,p,l);
      }
    v[or]=1;
    return i;
}
```

```
int BF (GrafoL g, int or,
        int v[],
        int p[],
        int l[]){
   int i, x; LAdj a;
   int q[NV], front, end;
   for (i=0; i<NV; i++) {
      v[i]=0;
      p[i] = -1;
      l[i] = -1;
   }
   front = end = 0;
   q[end++] = or; //enqueue
   v[or] = 1; p[or]=-1;l[or]=0;
   i=1;
   while (front != end){
      x = q[front++]; //dequeue
      for (a=g[x]; a!=NULL; a=a->prox)
         if (!v[a->dest]){
            i++;
            v[a->dest]=1;
            p[a->dest]=x;
            l[a->dest]=1+l[x];
            q[end++]=a->dest; //enqueue
         }
   }
    return i;
}
```

Usando estas funções ou adaptações destas funções, defina as seguintes.

1. A função `int maisLonga (GrafoL g, int or, int p[])` que calcula a distância (número de arestas) que separa o vértice `v` do que lhe está mais distante. A função deverá preencher o array `p` com os vértices correpondentes a esse caminho.

2. A função `int componentes (GrafoL g, int c[])` recebe como argumento um grafo não orientado `g` e calcula as componentes ligadas de `g`, i.e., preenche o array `c` de tal forma que, para quaisquer par de vértices `x` e `y`, `c[x] == c[y]` sse existe um caminho a ligar `x` a `y`.

   A função deve retornar o número de componentes do grafo.

3. Num grafo orientado e acíclico, uma ordenação topológica dos seus vértices é uma sequência dos vértices do grafo em que, se existe uma aresta $a \longrightarrow b$ então o vértice a aparece **antes de** b na sequência. Consequentemente, qualquer vértice aparece na sequência depois de todos os seus *alcançáveis*.

   A função `int ordTop (GrafoL g, int ord[])` preenche o array `ord` com uma ordenação topológica do grafo.

4. Considere o problema de guiar um robot através de um mapa com obstáculos.

   O mapa é guardado numa matriz de caracteres em que o caracter `'#'` representa um obstáculo. A posição `(0,0)` corresponde ao canto superior esquerdo do mapa e a posição `(L,C)` corresponde ao canto inferior esquerdo.

   O robot pode-se deslocar na vertical (Norte/Sul): passando da posição `(a,b)` para a posição `(a-1,b)`/`(a+1,b)`; ou na horizontal (Este/Oeste): passando da posição `(a,b)` para a posição `(a,b+1)`/`(a,b-1)`.

   Defina a função `int caminho (int L, int C, char mapa[L][C], int ls, int cs, int lf, int cf)` que determina o número mínimo de movimentos para chegar do ponto `(ls,cs)` ao ponto `(lf,cf)`.

   Sugestão: Em alguns casos as representações habituais de grafos introduzem um grand *overhead* no processo. Neste caso em particular, a informação sobre os adjacentes a um vértice (ponto do mapa) pode ser facilmente obtida por inspecção da matriz que representa o mapa.