Ficha 4

Algoritmos sobre Grafos

Algoritmos e Complexidade
LEI / LCC / LEF

Encontra-se em `https://codeboard.io/projects/301404/summary` um projecto onde poderá experimentar as soluções.

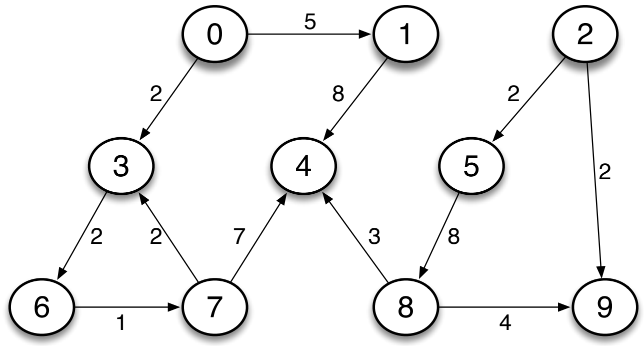
1 Representações

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];
```



Estas definições, bem como do grafo apresentado, estão disponíveis na seguinte página.
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. O grau de entrada do grafo acima é 3 (correspondente ao grau de entrada do vértice 4).

   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; l[or]=0; p[or]=-1; //redundante
   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 distância entre dois vértices define-se como o comprimento do caminho mais curto

entre esses dois vértices. Num grafo não pesado, tal corresponde ao número de arestas mínimo que ligam os dois vértices.

Defina a função `int maisLonga (GrafoL g, int or, int p[])` que calcula a distância (número de arestas) que separa o vértice `or` do que lhe está mais distante. A função deverá preencher o array `p` com os vértices correpondentes a esse caminho. No grafo apresentado acima, a invocação `maisLonga (g, 0, p)` deve dar como resultado 3 (correspondendo, por exemplo, à distância entre 0 e 7).

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 **antes** de todos os seus *alcançáveis.*

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

4. Defina a função `int ciclo (GrafoL g, int c[])` que testa se o grafo tem (pelo menos) um ciclo. Se tal acontecer, a função deve ainda preencher o array fornecido com os vértices do ciclo encontrado.

5. 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-1,C-1)` corresponde ao canto inferior direito.

   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], 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)`.

   Por exemplo, para o mapa à direita, a invocação `caminho (10, 10, mapa, 1,1,1,8)` deverá dar como resultado 31.
   Pode ainda generalizar essa função de forma a imprimir no ecran a sequência de movimentos necessários.
   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.

```
char *mapa [10]
    = {"##########"
      ,"# #   #  #"
      ,"# # # #  #"
      ,"#   # #  #"
      ,"##### #  #"
      ,"#     #  #"
      ,"## ####  #"
      ,"#  #     #"
      ,"#     #  #"
      ,"##########"};
```

3 Algoritmos sobre Grafos Pesados

Considere a definição da função `dijkstraSP` que imlementa o algoritmo de Dijkstra para cálculo dos caminhos mais curtos com origem num dado vértice.

```
int dijkstraSP (GrafoL g, int or, int pais[], int pesos[]) {
    int r, i, v, cor [NV],
        orla[NV], tam;
    LAdj x;
    // inicializacoes
    for (i=0; i<NV; i++){
         pais[i] = -2; cor[i] = 0 ; // nao visitado
    }
    r = 0; orla[0] = or; tam = 1;
    pesos[or] = 0; pais[or] = -1; cor [or] = 1; // na orla
    // ciclo
    while (tam>0) {
       // seleccionar vertice de menor peso
       i = minIndPeso (orla, pesos,tam);
       swap (orla, i, --tam);
       v = orla[tam];
       r++; cor[v] = 2; //visitado
       for (x=g[v]; x!=NULL; x=x->prox){
           if (cor[x->dest] == 0){
              cor[x->dest] = 1; orla[tam++] = x->dest;
              pais[x->dest] = v;
              pesos[x->dest] = pesos[v] + x->custo;
           }
           else if (cor[x->dest] == 1 &&
                     pesos[v] + x->custo < pesos[x->dest]) {
                     pais[x->dest] = v;
                     pesos[x->dest] = pesos[v] + x->custo;
           }
       }
    }
    return r;
}
```

Note que a função identifica que um vértice `i` não foi alcançado colocando em `pais[i]` o valor `-2`.

1. A excentricidade de um vértice define-se como a distância (comprimento do caminho mais curto) que liga esse vértice ao que lhe é mais distante. Defina a função `int excentricidadeV (GrafoL g, int v)`.

2. Considerando que os pesos das arestas correspondem aos kilómetros entre localidades (vértices), modifique a função apresentada de forma a incluir a seguinte restrição: só poderão ser usadas ligações com pesos inferiores a um determinado valor (passado como parâmetro) que corresponde à autonomia do veículo.

3. Adapte a função apresentada acima para calcular uma árvore geradora de custo mínimo segundo o algoritmo de Prim.

4. O diâmetro de um grafo define-se como o máximo das excentricidades dos seus vértices.

   Apresente uma definição da função `int diametro (GrafoL g)` usando uma implementação do algoritmo de Floyd Warshal.

   Como se comparam a complexidade (em termos de tempo de execução e memória usada) da função que definiu e uma outra versão que repetidamente usa a função `dijkstraSP` (assuma que a função `dijkstraSP` apresentada acima tem uma complexidade $\mathcal{O}(V * V + E)$)