Solución Examen de Programación 3 y III (07/02/2015)

Instituto de Computación, Facultad de Ingeniería, UdelaR

Ejercicio 1 (10 puntos)

1.Devuelve la cantidad de parejas del arreglo cuya suma es igual al tercer argumento.

2.Observemos que se revisan todas las parejas con índices diferentes, es decir, hay $\binom{n}{2}$ ejecuciones del `if`. Como en cada `if` hay dos accesos al arreglo, concluimos que $T(n)=n\times(n-1)$.

3.

a)**Verdadero**. $f\in O(f)$ siempre.

b)**Falso**. Por órdenes de infinitos, sabemos que $lim\frac{T(n)}{log\,n}=+\infty$. La regla del límite nos permite concluir que $T(n)\notin O(log(n))$.

c)**Verdadero**. Sabemos que $lim\frac{\frac{1}{2}n^2}{T(n)}=\frac{1}{2}\in R^{+}$. La regla del límite nos permite concluir que $T(n)\in O(\frac{1}{2}n^2)$ y $\frac{1}{2}n^2\in O(T(n))$. Además una propiedad del teórico establece que $f\in\Omega(g)\Leftrightarrow g\in\Omega(f)$, y por lo tanto también se cumple que $T(n)\in\Omega(\frac{1}{2}n^2)$. Luego concluimos que $T(n)\in\Theta(\frac{1}{2}n^2)$.

Ejercicio 2 (10 puntos)

Puedo obtener una solución proporcionando dos funciones que crezcan a la misma tasa, pero cuya razón cambie continuamente. Para ello, tomo una función constante y una que oscila continuamente a su alrededor. Concretamente, consideremos las funciones *f* y *g* que cumplen:

$$(\forall n\in N::f(n)=2+cos(n)\ y\ g(n)=1)$$

Mostraremos que $f\in O(g)$, $g\in O(f)$ y $lim_n\frac{f(n)}{g(n)}$ no existe.

$f\in O(g)$

$$\begin{gathered}f\in O(g)\\ \Leftarrow(Def.)\\ (\exists c\in R^{+}:(\exists n_0\in N::(\forall n\in N:n>n_0:f(n)\le c\times g(n))))\\ \Leftarrow(Def.)\\ (\exists c\in R^{+}:(\exists n_0\in N::(\forall n\in N:n>n_0:2+cos(n)\le c\times 1)))\\ \Leftarrow(Tomando\ c=3,n_0=0)\\ (\forall n\in N:n>0:2+cos(n)\le 3\times 1)\\ \Leftarrow\\ (\forall n\in N::cos(n)\le 1)\end{gathered}$$

y esto vale porque la función coseno está acotada superiormente por el 1.

$g\in O(f)$

$$\begin{gathered}g\in O(f)\\ \Leftarrow(Def.)\\ (\exists c\in R^{+}:(\exists n_0\in N::(\forall n\in N:n>n_0:g(n)\le c\times f(n))))\\ \Leftarrow(Def.)\\ (\exists c\in R^{+}:(\exists n_0\in N::(\forall n\in N:n>n_0:1\le c\times(2+cos(n)))))\\ \Leftarrow(Tomando\ c=1,n_0=0)\\ (\forall n\in N:n>0:1\le 2+cos(n))\\ \Leftarrow\\ (\forall n\in N::-1\le cos(n))\end{gathered}$$

y esto último vale porque la función coseno está acotada inferiormente por el -1.

$lim_n \frac{f(n)}{g(n)}$ **no existe**

Observemos que $\frac{f(n)}{g(n)} = 2 + cos(n)$. Sin embargo, $lim_n cos(n)$ no existe, y como el límite de la función constante 2 es simplemente 2, el límite de $2 + cos(n)$ no existe.

Ejercicio 3 (10 puntos)

1.Cada edificio se modela como un vértice del grafo *G*. Cada posible tendido eléctrico entre un edificio y otro se modela como una arista, con peso igual a la distancia entre ellos. La red eléctrica buscada será un árbol de cubrimiento de costo mínimo para *G*.

Se debe utilizar Prim por la restricción de tener que contar un única componente conexa durante el armado de la red (1 tendido a la vez) lo cual es garantizado por el algoritmo de Prim y no Kruskal pese a que ambos obtienen un MST.

Solución con Prim

```
Prim (Grafo G, Grafo &arbol)
{
ListaEdificios U;
red = CrearGrafo();
U = CrearLista();
Agregar(U,1);
mientras (U != V)
     sea <u,v> tendido con distancia[u,v] mínima / u in U y v in V-U
         AgregarTendido(red,<u,v>);
         Agregar(U,v);
fin mientras
}
```

2.La técnica aplicada es *Greedy,* basada en la selección de la mejor arista (costo mínimo) al agregar en cada iteración (óptimo local) del algoritmo para finalmente obtener un MST.

Ejercicio 4 (10 puntos)

```
1.MERGE(S1,S2)=
      Si Vacia(S1)⇒ S2
      Sino
         Si Vacia(S2)⇒ S1
         Sino
            Si Primero(S1)< Primero(S2)
               InsFront(Primero(S1), MERGE(Resto(S1),S2))
            Sino
               InsFront(Primero(S2), MERGE(S1,Resto(S2))


      void MergeSort(arreglo &A, int p, int q){
           int medio;
           if (p<q) { // quedan al menos 2 elementos
                medio = (p+q) / 2;
                MergeSort(A, p, medio);
                MergeSort(A, medio+1, q);
                MERGE(A, p, medio, medio+1, q);
           } //if
      } // fin Mergesort
```

2.Para una entrada de N elemenntos, el Merge sort tiene $T_B(n), T_W(n), T_A(n) \in O(n log n)$.

Problemas

Problema 1 (30 puntos)

1.El conjunto de empleados y la relación de trabajo usual ente ellos se modela como un grafo no dirigido G, sin lazos ni aristas múltiples, en donde cada vértice corresponde a un empleado y cada arista entre dos vértices indica que los empleados correspondientes trabajan juntos usualmente.

La condición pedida implica que el grafo debe ser **bipartito completo**.

2.En una recorrida BFS de un grafo bipartito completo se tienen solamente tres niveles: 0, 1 y 2. La idea de la solución propuesta es hacer una recorrida BFS de G, en la que de acuerdo al nivel del nodo visitado se ejecut an determinadas acciones y controles. Se emplea un vector auxiliar *niveles* para guardar el nivel de los nodos de G en la recorrida BFS.

Se elige como raíz del árbol BFS un vértice arbitrario (por ejemplo el vértice 1), el cual será el único vértice en el nivel 0, que se marca como visitado y se encola. A su vez también en forma arbitraria se le asigna el *grupo1* y se lo inserta en la lista de dicho grupo.

Cuando se desencola un vértice, se discriminan según su nivel las tareas a realizar:

Si es el vértice desencolado es de nivel 0, a cada uno de sus vértice adyacentes se lo marca como visitado, se lo encola, se lo inserta en la lista del *grupo2* y se le asigna el nivel 1. También se calcula el número de vértices en el nivel 1, que es igual a la cantidad de vértices adyacentes al vértice de nivel 0.

Si el vértice desencolado es de nivel 1, se discrimina si es el primero de nivel 1 a procesar o no. Si es el primero, a cada uno de sus adyacentes no visitado se lo marca como visitado, se le asigna el nivel 2 y el *grupo1*, se lo inserta en la lista de dicho grupo y se lo encola. También se incrementa un contador de vértices de nivel 2. Si uno de sus adyacentes ya fue visitado, se comprueba si es de su mismo nivel 1 y de ser así se termina el algoritmo devolviendo FALSE ya que existiría arista entre dos vértices de un mismo nivel. Si el vértice desencolado de nivel 1 no es el primero procesado de este nivel, para que el grafo sea bipartito completo sus vértices adyacentes deben coincidir estrictamente con los vértices adyacentes del primero de nivel 1 procesado, no debiendo tener adyacentes no visitados ni adyacentes de su mismo nivel 1. A su vez se lleva la cuenta de la cantidad de adyacentes de nivel 2 que tiene, que debe coincidir con el número de vértices de nivel 2 identificados al procesar el primer vértice desencolado de nivel 1. Si no se cumple alguna de estas condiciones, el algoritmo termina devolviendo FALSE.

Si el vértice desencolado es de nivel 2, lo que se tiene que comprobar es que no tenga adyacentes de su mismo nivel 2 ni adyacentes no visitados todavía. No es necesario verificar que tenga arista con todos los vértices de nivel 1 pues esto ya se verificó al procesar los vértices de nivel 1.

Finalmente al quedar vacía la cola del BFS tenemos que verificar que el grafo G sean conexo, lo que se hace comparando la suma de vértices en cada uno de los tres niveles del árbol BFS con el número de vértices *n* del grafo *G*.

```
bool armarGrupos(Grafo G, int n, ListaVertices &grupo1, ListaVertices &grupo2)
{
     bool * visitados = new int [n + 1]; // marcas de vertices en la recorrida BFS
     bool * niveles = new int [n + 1];   //niveles de cada vertice en el arbol BFS
     int cantVert1,cantVert2; //cantidad de vertices en los niveles 1 y 2
     bool primero_nivel1;
     ListaVertices adyacentes;
     int i;
     for (i = 1; i <= n; i++)
     visitados [i] = false;
     cantVert1 = 0;
     cantVert2 = 0;
```

```
primero_nivel1 = true;
int raiz;
raiz = 1; /* se elige arbitrariamente el vertice 1 como raiz del árbol BFS */
visitados[raiz] = true;
niveles[raiz] = 0;
AgregarVertice(grupo1,raiz); // se elige el grupo 1 para el vértice raíz
Cola cola = CrearCola();
Encolar (cola, raiz);
int nivel;
while (! EsVaciaCola (cola))
{
     v = Desencolar (cola);
     adyacentes = ListaAdyacentes (G, v);
     nivel = niveles[v]
     switch(nivel)
      {
           case 0:
               while (!EsVaciaLista(adyacentes))
               {
                    w = Primero(adyacentes);
                    adyacentes = Resto(adyacentes);
                    Encolar(cola,w);
                 visitados[w] = true;
                 niveles[w] = 1;
                 AgregarVertice(grupo2,w);
                 cantVert1 = cantVert1+1;
               }
              break;
          case 1:
               if primero_nivel1
               {
                    while (!EsVaciaLista(adyacentes))
                    {
                         w = Primero(adyacentes);
                         adyacentes = Resto(adyacentes);
                         if (!visitados [w])
                         {
                              Encolar(cola,w);
                           visitados[w] = true;
                           niveles[w] = 2;
                           AgregarVertice (grupo1,w);
                               cantVert2 = cantVert2+1;
                         }
```

```
                    else if (niveles[w] == 1)
                        return false;
                }
                primero_nivel1 = false;
            }
            else
            {
                ady_nivel2 = 0;
                while (!EsVaciaLista(adyacentes))
                {
                    w = Primero(adyacentes);
                    adyacentes = Resto(adyacentes);
                    if ((!visitados [w]) || (niveles[w] == 1))
                        return false;
                    else if (niveles{w] ==2)
                        ady_nivel2 = ady_nivel2+1;
                }
                if (ady_nivel2 != cantVert2)
                    return false;
            }
            break;
        case 2:
            while (!EsVaciaLista(adyacentes))
            {
                w = Primero(adyacentes);
                adyacentes = Resto(adyacentes);
                if ((!visitados [w]) || (niveles[w] == 2))
                    return false;
            }
            break;
        }
    }
    return ((1+cantVert1+cantVert2)==n) // control de que el grafo sea conexo
}
```

3.El peor caso del costo de ejecución del algoritmo se da cuando el grafo *G* es bipartito completo, de forma de que no haya una terminación anticipada del mismo. Sean c_1 y c_2 el número de vértices en los grupos 1 y 2 respectivamente, que verifican $c_1+c_2=n$. El número de aristas a del grafo bipartito completo se puede expresar como $a=c_1 c_2=c_1(n-c_1)$, que resulta tener un máximo para , con valor máximo .

4.En el peor caso, el algoritmo visita una vez a cada nodo al desencolarlo y recorre dos veces cada arista del grafo. Todas las operaciones que involucran el procesar una arista son de $O(1)$. El algoritmo resulta ser entonces $O(2a)$ ($2a \geq n$ en un grafo bipartito completo con $n \geq 2$).

➔ Empleando el resultado de la parte c), el algoritmo propuesto resulta $O(n^2)$.

Problema 2 (30 puntos)

Forma de la tupla

Tupla de largo fijo 3 T= <x, y, c>, donde:
•x = <x_1, ..., x_J> es una tupla de largo variable j < 4 y representa los vuelos tomados a la ida en forma ordenada (primero el vuelo x_1, luego el x_2, etc.).
•y = <y_1, ..., y_K> es una tupla de largo variable k < 4 y representa los vuelos tomados a la vuelta en forma ordenada
•c representa el crucero elegido.

Restricciones explícitas

1. $J, K \in \{1,...,3\}$ Se realizan hasta 3 escalas tanto en el vuelo de ida como en el de vuelta.
2. $x_i \in vuelos$, $y_i \in vuelos$. Los vuelos tomados tanto a la ida como a la vuelta están dentro de los V vuelos disponibles.
3. $c \in cruceros$ El crucero elegido es uno de los C disponibles.
4. $x_i.llegada \leq x_i.partida \; \forall i \in \{1,...J-1\}$ y $y_i.llegada \leq y_i.partida \; \forall i \in \{1,...K-1\}$ Las horas de llegada y partida en los puntos de conexión son adecuadas.
5. $x_i.destino = x_{i+1}.origen \; \forall i \in \{1,...J-1\}$, $y_i.destino = y_{i+1}.origen \; \forall i \in \{1,...K-1\}$ las escalas entre vuelos están bien conectadas.
6. $x_1.origen = y_K.destino = pareja.ciudadResidencia$ La pareja parte en el primer vuelo desde su ciudad de residencia y retorna con el último vuelo a dicha ciudad.

Restricciones implícitas

1. $\sum_{i=1}^{J} x_i.precio + \sum_{i=1}^{K} y_i.precio + c.precio \leq P$ El costo de todos los vuelos efectuados más el del crucero no supera el presupuesto.
2. $x_J.llegada \leq c.partida$ y $c.llegada \leq y_1.partida$ La pareja llega a tiempo para la partida del crucero y para tomar el primer vuelo de retorno.
3. $x_J.destino = c.origen$ y $y_1.origen = c.destino$ El último vuelo realizado a la ida lleva a la ciudad de donde parte el crucero, y el primer vuelo de regreso parte de la última ciudad del crucero.

Función objetivo

Se quiere elegir, dentro de las soluciones que maximizan los puntos, las que minimicen la cantidad de vuelos.
La función objetivo es $min_{t \in MaxPtos} cantVuelos(t)$
donde:
$MaxPtos = \{t \in T \mid G(t) = máx_{t' \in T}\{G(t')\}\}$ MaxPtos es el conjunto de las tuplas solución con mayor puntaje
$cantVuelos(\langle\langle x_1,...,x_J\rangle,\langle y_1,...,y_K\rangle,c\rangle) = J + K$ cantVuelos es la suma de la cantidad de vuelos tomados a la ida y a la vuelta
$G(\langle\langle x_1,...,x_J\rangle,\langle y_1,...,y_K\rangle,c\rangle) = \sum_{h=1}^{c.cantLugares} puntosPreferencia[c.lugares[h]]$ la ganancia de una tupla se calcula como la suma de los puntos de preferencia de los lugares recorridos por el crucero
T es el conjunto de tuplas solución.