Solución Examen de Programación 3 y III (14/12/2009)

Instituto de Computación, Facultad de Ingeniería

Parte Obligatoria

<u>Solución Ejercicio 1 (10 puntos)</u>

Forma de la solución:
La tupla representa el camino desde el vértice **u** al vértice **v**, como no se conoce la cantidad de vértices de dicho camino, se usará una tupla de largo variable.

Tupla de largo variable $<x_1, x_2,..., x_k>$, $k \leq |V|$, $\forall$ i con $1\leq i \leq k$ x_i representa un vértice del grafo.

Restricciones explícitas:
Cada componente de la tupla representa un vértice del camino

1. $x_1 = \mathbf{u}$ — La primer componente de la tupla es el vértice inicial del camino (**u**)
2. $x_k = \mathbf{v}$ — La última componente de la tupla es el vértice final del camino es (**v**)
3. $x_i \in V \ \forall$ i con $2 \leq i \leq k-1$ — Las restantes componentes representan vértices del grafo

Restricciones implícitas:
La tupla debe representar un camino, entonces los vértices consecutivos de la misma son adyacentes entre sí
1. $(x_{i-1}, x_i) \in E \ \forall$ i con $2 \leq i \leq k$

Los vértices del camino deben ser distintos. Al no haber costos negativos no se obtiene un camino de costo mínimo al pasar más de una vez por el mismo vértice.
2. $x_i \neq x_j$ si $i \neq j \ \forall$ i con $1 \leq i \leq k$ y $\forall$ j con $1 \leq j \leq k$

Función objetivo:
Se debe hallar un camino de costo mínimo. El costo del camino es la suma de los costos de las aristas definidas por los pares de vértices consecutivos.

$f = \min(Costo(t))$ donde 'Costo' es el costo del camino que representa la tupla solución $t=<x_1, x_2,..., x_k>$:

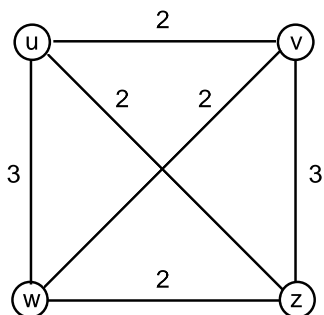
$Costo(t) = \sum_{i=2}^{i=k} c(\langle x_{i-1}, x_i \rangle)$ siendo $c(\langle x_{i-1}, x_i \rangle)$ el costo de la arista $\langle x_{i-1}, x_i \rangle \in E$

$f = \min_{t\in T}(Costo(t))$, donde $T = \{ t = <x_1, x_2,..., x_k>/ \text{ t es solución } \}$

Predicado de poda:
Se poda una rama si la tupla en construcción $<x_1, x_2,\ldots,x_h, 0 ,\ldots,0>$ tiene un costo mayor que la mejor solución hasta el momento:
$Costo(<x_1, x_2,\ldots, x_h, 0, \ldots, 0>) > Costo(<y_1, y_2, \ldots , y_k>)$, $\forall$ h con $1 \leq h \leq k$, siendo $<y_1, y_2,\ldots, y_k>$ una tupla solución y la mejor hasta el momento.

Nota: Según los apuntes publicados no debería ir en este ítem una expresión derivada de la función objetivo. Sin embargo, como hasta el momento se ha visto de ambas formas, se incluye este predicado. De todas maneras, no se controla para la corrección del examen.

Solución Ejercicio 2 (10 puntos)



El algoritmo de Dijkstra para cualquier vértice devuelve un árbol de cubrimiento de costo 7, que esta formado por las 3 aristas incidentes en cada vértice. Sin embargo el árbol de cubrimiento de costo mínimo es 6 (formado por las aristas <u, v>, <w, z> y una de las siguientes: <u, z> ó <w, v>)

Solución Ejercicio 3 (10 puntos)

Considerando como operación básica la *asignación* a un elemento de ***vector***.

1. ¿De qué depende el costo (cantidad de asignaciones) del algoritmo ***anf***?

El costo depende exclusivamente del tamaño de la entrada, o sea, del n de la invocación `anf(vector, n, 0).`

2. Calcule el costo en los casos mejor, peor y promedio.

A raíz de la parte anterior se deduce que $T_B(n) = T_W(n) = T_A(n) = T(n)$.

- Se hacen dos invocaciones de *anf* con vectores de largo $n/2$.
- Se hacen $n/2$ inicializaciones en el arreglo vector.

$T(n) = 2T(n/2) + n/2$
$T(1) = 0$

Siendo $n = 2^k$.

$T(2^k) = 2T(2^{k-1}) + 2^{k-1}$
$T(2^{k-1}) = 2T(2^{k-2}) + 2^{k-2}$
$T(2^{k-2}) = 2T(2^{k-3}) + 2^{k-3}$
……….
$T(2^2) = 2T(2^1) + 2^1$
$T(2^1) = 2T(2^0) + 2^0$

Operando:

$T(2^k) = 2T(2^{k-1}) + 2^{k-1}$
$2T(2^{k-1}) = 2^2T(2^{k-2}) + 2^{k-1}$
$2^2T(2^{k-2}) = 2^3T(2^{k-3}) + 2^{k-1}$
……….
$2^{k-2}T(2^2) = 2T(2^1) + 2^{k-1}$
$2^{k-1}T(2^1) = 2^k T(2^0) + 2^{k-1}$

Sumando todas las ecuaciones resulta $T(2^k) = k\ 2^{k-1}$

Como $n = 2^k$ entonces se tiene que $T(n) = (log\ n\ )2^{log\ n\ -1}$

Simplificando: $T(n) = ½\ n\ log\ n$

3. Calcule los órdenes exactos de los tres casos justificando su razonamiento. (**Esta parte se corrige solamente si calculó los costos satisfactoriamente).**

$$O(f) = \{g : N \rightarrow R^* / \exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \leq c \cdot f(n)\}$$

$$\Omega(f) = \{g : N \rightarrow R^* / \exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \geq c \cdot f(n)\}$$

Definición: Orden exacto.

$$\Theta(f) = O(f) \cap \Omega(f)$$

Es claro que $T(n)$ pertenece a $O(n\ log\ n)$ y $\Omega(n\ log\ n)$ con $k=1/2$ y $n_0= 0$.

Por lo que:

$T_B(n) = T_W(n) = T_A(n) = T(n) \in \Theta(n\ log\ n)$

4. ¿Si además de considerar la asignación también se consideran las lecturas del arreglo vector cambiaria alguno de los ordenes? **(Esta parte se corrige solamente si hizo satisfactoriamente la parte anterior).**

Solución 1: se observa que las operaciones básicas se siguen ejecutando en la misma línea de código que en el caso anterior y lo único que cambia que en vez de contar 1 se cuenta 3, por lo que se el nuevo costo resulta ser tres veces más que el costo anterior ya que las veces que se ejecuta esa línea depende exclusivamente del *n* como se vio en la parte 1).

Por lo que $T(n) = 3/2\ n\ log\ n$.

Solución 2: en el caso que se consideren también las lecturas del arreglo vector resulta que:

- Se hacen dos invocaciones de *anf* con vectores de largo $n/2$.
- Se hacen $n/2$ inicializaciones en el arreglo vector.
- Se hacen n lecturas del arreglo vector.

$T(n) = 2T(n/2) + 3/2\ n$

$T(1) = 0$

Siendo $n = 2^k$.

$T(2^k) = 2T(2^{k-1}) +3\ 2^{k-1}$

$T(2^{k-1}) = 2T(2^{k-2}) +3\ 2^{k-2}$

$T(2^{k-2}) = 2T(2^{k-3}) +3\ 2^{k-3}$

……….

$T(2^2) = 2T(2^1) +3\ 2^1$

$T(2^1) = 2T(2^0) +3\ 2^0$

Operando:

$T(2^k) = 2T(2^{k-1}) + 3\ 2^{k-1}$
$2T(2^{k-1}) = 2^2T(2^{k-2}) + 3\ 2^{k-1}$
$2^2T(2^{k-2}) = 2^3T(2^{k-3}) + 3\ 2^{k-1}$
……….
$2^{k-2}T(2^2) = 2T(2^1) + 3\ 2^{k-1}$
$2^{k-1}T(2^1) = 2^k\ T(2^0) + 3\ 2^{k-1}$

Sumando todas las ecuaciones resulta $T(2^k) = k3\ 2^{k-1}$

Como $n = 2^k$ entonces se tiene que $T(n) = 3\ (log\ n\ )2^{log\ n\ -1}$

Simplificando: $T(n) = 3/2\ n\ log\ n.$

Para ambas soluciones:

Es claro que $T(n)$ pertenece a $O(n\ log\ n)$ y $\Omega(n\ log\ n)$ con $k=3/2$ y $n_0 = 0$.

Por lo que:
$T_B(n) = T_W(n) = T_A(n) = T(n) \in \Theta(n\ log\ n)$

Solución Ejercicio 4 (10 puntos)

a. Describa la forma general de la técnica **Greedy** en pseudocódigo. Explique cada paso.

Los algoritmos voraces son aquellos los cuales para resolver un problema se basan en construir una solución utilizando decisiones que son óptimos locales logrando un optimo global (en caso de ser correcto el algoritmo).

```
Greedy ( C )
{
    S := Φ
    while ( !esVacio(C) and  !esSolucion(S)  )
    {
           x :=  Select(C)
           C = C - {x}

           if ( esFactible (S U {x} )
           {
                  S := S U {x}
           }
    }

    if ( esSolucion(S))
           return S
    else
           return Φ
}
```

C es el conjunto de candidatos.
S es la solución.
La función *select* se encarga de la selección del mejor de los candidatos basándose en una decisión local.

b. Explique qué resuelve el algoritmo de *Prim* e indique las correspondencias con la forma general de la técnica que utiliza.

El algoritmo de *Prim* es un algoritmo Greedy que calcula un árbol de cubrimiento de costo mínimo de un grafo (ACCM).

Dado un grafo G = (V,E) se empieza a construir el ACCM desde un vértice del grafo arbitrario. Durante las distintas iteraciones se mantiene un árbol que va creciendo en cada una de ellas. Esto significa que si en una iteración cualquiera se tiene un árbol parcial T formado por un conjunto de vértices U, se agrega en la siguiente iteración la arista que cumple lo siguiente: es la arista de mínimo costo de todas las aristas existentes en el grafo entre un vértice de U y un vértice de V-U (esto asegura no formar ciclos). El algoritmo finaliza cuando el árbol T contiene todos los vértices del grafo.

```
Prim ( G )
{
	if ( V = Φ)
		return Φ

	U := { any( V ) }
	T := Φ

	while (  U != V )
	{
		(u, v) := choose( U , G )

		if ((u, v) = null )
			return Φ

		U := U ∪ {v}
		T := T ∪ { (u,v) }
	}
	return T
}
```

- La función `choose` busca la arista que tiene menor costo tal que uno de sus vértices pertenece al conjunto U y el otro a V-U

La correspondencia con la forma general de la técnica Greedy es la siguiente:

- El conjunto de candidatos son las aristas del grafo, la iteración se realiza sobre los vértices porque el ACCM debe contener a todos los vértices del grafo.
- La función `choose` la que se encarga de elegir uno de los candidatos tomando una decisión localmente óptima y además por la forma que lo elige asegura la factibilidad (no formar ciclos), por lo tanto se corresponde con las funciones Select y esFactible de la forma general.
- El árbol T es la solución generada por el algoritmo
- La función esSolucion no es necesaria en este caso, dado que para obtener el ACCM se deben considerar todos los vértices del grafo.

Nota: para la parte b no es necesario incluir la implementación del algoritmo de Prim.

Problemas

Solución Problema 1 (30 puntos)

a) (20 puntos)

Sea *f(i, j)* la cantidad mínima de golosinas que se pueden comprar con hasta *j* pesos utilizando las golosinas entre *0* e *i* del vector *v*, teniendo en cuenta la cantidad de golosinas que se tiene para cada tipo de golosina de *0* a *i* en el vector *c*.

Pasos bases:

$f(i,0)=0 \text{ con } 0 \leq i < g$

$$f(0,j)=\begin{cases} k & \text{si } j=k.v[0] \text{ y } k \leq c[0] \quad con \quad 0 \leq j \leq monto \\ k & si \quad j > c[0].v[0] \\ +\infty & en \quad caso \quad contario \end{cases}$$

El tercer paso base considera que el niño compra todas las golosinas del almacén y además le sobra dinero.

Pasos recursivos:

$$f(i,j)=\begin{cases} \min\{f(i-1, j-v[i].k)+k\} & con \quad 1 \leq i < g, \quad 0 \leq j \leq monto, \quad 0 \leq k \leq c[i] \quad y \quad v[i].k \leq j \\ f(i-1,j) \quad en \quad caso \quad contrario \end{cases}$$

donde *k* indica la cantidad de golosinas del tipo *i* que se utilizan.

La solución se obtiene con $f(g-1, monto)$.

b) (10 puntos)
b.1

$T(n) = T(\lfloor (n/2 \rfloor) + T(\lceil n/2 \rceil) + (n-1)$

$T(1)=0$

Considerando n potencia de 2, $\mathbf{n=2^p}$ se asegura que cada vez que se divida por 2 la división será exacta. Entonces la recurrencia queda:

$T(n) = 2T(n/2)+n-1 \qquad T(1)=0$

Expandiendo la recurrencia y multiplicando sucesivamente por 2:

$T(n) = 2T(n/2)+n-1$

$2^1 T(n/2) = 2^2 T(n/2^2)+n-2^1$

$2^2 T(n/2^2) = 2^3 T(n/2^3)+n-2^2$

$\vdots$

$2^{p-1} T(n/2^{p-1}) = 2^p T(n/2^p)+n-2^{p-1}$

$2^p T(n/2^p) = 2^p T(1) = 0$

Sumando y eliminando términos queda:

$$T(n)=\sum_{i=0}^{p-1}(n-2^i) \qquad T(n)=np-\sum_{i=0}^{p-1}2^i \qquad T(n)=np-2^p+1$$

como habíamos tomado n=2^p , p = log n

$$T(n)=n\log n-n+1$$

b.2

1) $T_W(n)$ = T(n) = nlogn -n + 1 por parte (b.1). Por lo tanto, T(n) ∈ O(nlogn) ⇒ $T_W(n)$ ∈ O(nlogn)

2) El caso medio debe tener un costo menor, a lo sumo igual que el peor caso por definición de éstos, entonces: $T_A(n) \leq T_W(n) \Rightarrow T_A(n) \in O(n\log n)$

3) Por letra: $T_A(n) \in \Omega(n \log n)$

4) Por (2) y (3) y por definición de Θ ⇒ $T_A(n) \in \Theta(n\log n)$

Solución Problema 2 (30 puntos)

a) (12 puntos)

Recordando que la *distancia* entre 2 vértices *v* y *w* se define como el largo del camino más corto entre ambos vértices, el requerimiento puede resolverse tomando como base una recorrida *BFS* del grafo comenzando por el vértice *w*, dado que, por las características de esta recorrida, el camino desde *w* a cualquier otro vértice *z* (al cual se puede llegar desde *w*) formado por las aristas usadas en la misma es un camino más corto entre *w* y *z*.

En una recorrida *BFS* se utiliza una *Cola* como estructura auxiliar. Para resolver el problema se debe guardar, además del vértice, la distancia de éste a *w* o un indicador booleano para imprimir o no el vértice.

Se puede implementar entonces con una *Cola* de pares (vértice, distancia) o (vértice, booleano), o en su defecto, con dos *Colas*.

NOTAR que no sirve una recorrida en profundidad (*DFS*) porque ésta recorre cualquier camino entre 2 vértices y no necesariamente el más corto, que es el usado para obtener la *distancia*.

```
void ImprimeImparesBFS(Grafo *g, int w) {
     ListaEnteros* adyacentes;
     int cantNodos = GrafoCantNodos(g);
     int *marcados = new int[cantNodos];
     for (int i = 0; i < cantNodos; i++) {
         marcados[i] = 0;
     }

     ColaEnteros* vertices = ColaCrearVacia();
     ColaEnteros* distancias = ColaCrearVacia();
     marcados[w] = 1;
     int verticeInicial = w;
     int distanciaInicial = 0;
     ColaEncolar(vertices, verticeInicial);
     ColaEncolar(distancias, distanciaInicial);

     while (!ColaEsVacia(vertices)) {
           int verticeActual = ColaPrimero(vertices);
           ColaDesencolar(vertices);
           int distanciaActual = ColaPrimero(distancias);
           ColaDesencolar(distancias);

           if ((distanciaActual % 2) == 1) {
              printf ("%i ", verticeActual);
           }

           adyacentes = GrafoAdyacentes(g, verticeActual);

           while (!ListaEsVacia(adyacentes)) {
                 int v = ListaPrimero(adyacentes);
                 adyacentes = ListaResto(adyacentes);

                 if (marcados[v] == 0) {
                    marcados[v] = 1;
                    ColaEncolar(vertices, v);
                    ColaEncolar(distancias, distanciaActual + 1);
                 }
           }
     }

     ColaDestruir(vertices);
     ColaDestruir(distancias);
     delete [] marcados;
}
```

b) (18 puntos)

La idea es realizar una recorrida *DFS* a partir de cada vértice, contando en dicha recorrida la cantidad de veces que este vértice es alcanzado. En la recorrida *DFS* cada vértice será marcado al aplicar *DFS* desde él y desmarcado al terminar a los efectos de que todos los ciclos sean considerados.

```
void ContarDFS(Grafo *g, int origen, int v, int &cantCiclos, int *marcados) {
	marcados[v] = 1;
	ListaEnteros* adyacentes = GrafoAdyacentes(g, v);

	while (! ListaEsVacia(adyacentes)) {
		int w = ListaPrimero(adyacentes);
		adyacentes = ListaResto(adyacentes);

		if (w == origen) {
		   cantCiclos++;
		}
		else {
		     if (!marcados[w]) {
		        ContarDFS(g, origen, w, cantCiclos, marcados);
		     }
		}
	}
	marcados[v] = 0;
}

int CuentaCiclos(Grafo *g) {
	int cantNodos = GrafoCantNodos(g);
	int *marcados = new int[cantNodos];
	int v, cantCiclos = 0;

	for (v = 0; v < cantNodos; v++)
		marcados[v] = 0;

	for (v = 0 ; v < cantNodos; v++)
		ContarDFS(g, v, v, cantCiclos, marcados);

	delete [] marcados;
	return cantCiclos;
}
```