Solución Segundo Parcial de Programación 3 (1/12/2009)

Instituto de Computación, Facultad de Ingeniería

Ejercicio 1 (20 puntos)

1) **(10 puntos)**
Forma de la solución: tupla de largo k variable que representa un tour *T* de la forma $T =< t_0,...,t_{k-1} >$, donde $0 \le i < k \quad \forall t_i$, con *t* iteramos el tour *T*. Cada componente t_i representa una ruta de *T*.

Cada elemento t_i es una tupla de largo $m_i + 1$ variable de la forma $t_i =< r_{i0},...,r_{im_i} >$, donde $0 \le j \le m_i \quad \forall r_{ij}$, con r_{ij} iteramos t_i, la ruta t_i del tour *T*.

Restricciones explicitas:

$r_{ij} \in 0,\ldots,cantidadSecciones-1 \quad \forall\, 0 \le j \le m_i, \forall t_i$.

$r_{i0} = r_{im_i} = 0 \quad \forall t_i$.

Restricciones implícitas:

Restricción de capacidad:

$m_i \le C + 2 \quad \forall t_i$

Restricción todos los DVDs del buzón:

$\sum_{i=0}^{k-1} \sum_{j=1}^{m} s_{ij}^h = b_h$, donde

b_h es la cantidad de DVDs en el buzón de la sección *h*.

$s_{ij}^h = \{1 \; si \; r_{ij} = h; \quad 0 \; si \; no\}$

Función objetivo:

$\min \sum_{i=0}^{k-1} \sum_{j=1}^{m} c_{r_{ij-1} r_{ij}}$, donde c_{ab} es el costo de ir de la sección *a* a la *b*.

2) **(10 puntos)**

```
bool bestTour(int cantSecciones, int** costoEntreSecciones,
        int* cantDVDBuzonSeccion, int cantDVDSoportado, Tour* &tour){
  Tour* actual = crearTour();
  tour = crearTour();
  for(int i=0; i<cantSecciones; i++)restan+=cantDVDBuzonSeccion[i];
  if(cantDVDSoportado == 0 && restan > 0) return false;
  backtracking_bestTour(cantSecciones, costoEntreSecciones,
       cantDVDBuzonSeccion, cantDVDSoportado, tour, COSTO_MAX, actual, 0,
       cantDvDActual, restan);
  destruirTour(actual);
  return true;
}

void backtracking_bestTour(int cantSecciones, int** costoEntreSecciones,
       int* cantDVDBuzonSeccion, int cantDVDSoportado, Tour* &tour, int
       costoTour, Tour* &actual, int costoActual,  int cantDVDActual, int
       restan){
  if(restan == 0){
    actual->tour[actual->cantRutas - 1]->ruta[actual->tour[actual->cantRutas –
       1]->largoRuta - 1] = 0;
    actual->tour[actual->cantRutas - 1]->largoRuta ++;
    tour = copiarTour(actual);
    actual->tour[actual->cantRutas - 1]->largoRuta --;
    return;
  }
  for(int i = 0; i < cantSecciones; i++){
    int costo_i = costoEntreSecciones[actual->tour[actual->cantRutas - 1]->
       ruta[actual->tour[actual->cantRutas - 1]->largoRuta - 1][i];
    if(restan == 1) costo_i += costoEntreSecciones[i][0];
    if(tour->costo > costo_i + actual->costo){
      if(cantDVDBuzonSeccion[i] > 0 && cantDVDActual < cantDVDSoportado){
        cantDVDBuzonSeccion[i]--;
        actual->tour[actual->cantRutas - 1]->costo += costo_i;
        actual->tour[actual->cantRutas - 1]->ruta[actual->tour
                            [actual->cantRutas - 1]->largoRuta - 1] = i;
        actual->tour[actual->cantRutas - 1]->largoRuta++;
        backtracking_bestTour(cantSecciones, costoEntreSecciones,
           cantDVDBuzonSeccion, cantDVDSoportado, tour, costoTour, actual,
           actual->costo + costo_i , cantDvDActual + 1, restan - 1);
        cantDVDBuzonSeccion[i]++;
        actual->tour[actual->cantRutas - 1]->costo -= costo_i;
        actual->tour[actual->cantRutas - 1]->largoRuta--;
     }//if
     else if(cantDVDActual > 0 && i == 0){
        actual->tour[actual->cantRutas] = crearRuta();
        actual->cantRutas++;
        actual->tour[actual->cantRutas - 1]->ruta[actual->tour
                    [actual->cantRutas - 1]->largoRuta - 1] = i;
        actual->tour[actual->cantRutas - 1]->largoRuta++;
        actual->tour[actual->cantRutas - 1]->costo = 0;
        backtracking_bestTour(cantSecciones, costoEntreSecciones,
             cantDVDBuzonSeccion,cantDVDSoportado, tour, costoTour, actual,
             actual->costo + costo_i , 0, restan);
        destruirRuta(actual->tour[actual->cantRutas - 1]);
        actual->cantRutas--;
     }//else
   }//if
 }//for
}//backtracking_bestTour
```

Ejercicio 2 (20 puntos)

1) **(14 puntos)**

Este problema se puede modelar considerando las secuencias de traducciones como caminos de un idioma a otro, donde los idiomas representan los nodos de un grafo, y los diccionarios sus aristas. Existe una arista entre dos nodos del grafo si existe un diccionario que traduce los idiomas que representan dichos nodos. Dado que los diccionarios permiten traducciones bidireccionales, por cada diccionario hay dos aristas, cada una en un sentido. El costo de cada arista es 1 ya que se quiere saber la secuencia de traducciones de longitud mínima.

Este problema se resuelve aplicando el algoritmo de Floyd visto en el teórico que permite calcular los caminos mínimos entre cualquier par de nodos de un grafo.

Para poder aplicar el algoritmo de Floyd el grafo no puede contener ciclos de costo negativo. Dado que todas las aristas tienen costo 1, el grafo no contendrá ciclos de costo negativo.

Por lo tanto, $f^k(i,j)$ representa la cantidad de traducciones necesarias (mínimas) para traducir un texto desde el idioma i al idioma j usando únicamente a lo sumo k traducciones intermedias. En caso de no existir solución, se devuelve $+\infty$

Pasos base:

- $f^0(i, j) = 0$ si $i = j$. Este caso representa que se quiere traducir un texto entre el mismo idioma, por lo tanto, se necesitan 0 traducciones. *Nota: al momento de la corrección se consideraron como válidas las soluciones que incluían o no éste caso.*

- $f^1(i,j) = 1$ *si ExisteDiccionario(i, j),* $k = 1$ *y* $1 \leq i, j \leq I$. Esto implica que existe un diccionario que traduce los idiomas i y j.

- $f^1(i,j) = +\infty$ *si no ExisteDiccionario(i, j),* $k = 1$ *y* $1 \leq i, j \leq I$. Esto implica que no existe un diccionario que traduce los idiomas i y j.

Pasos recursivos

Se tienen las siguientes situaciones:

- el camino de menor costo entre los vértices i y j usando los vértices $\{1..k\}$ como vértices intermedios y no pasando por el vértice k, en este caso: $f^k(i,j) = f^{k-1}(i,j)$

- el camino de menor costo entre los vértices i y j usando los vértices $\{1..k\}$ como vértices intermedios y pasando por el vértice k, en este caso:
  $f^k(i,j) = f^{k-1}(i,k) + f^{k-1}(k,j)$

Por lo tanto,

$$f\ k(i,j) = \begin{cases} f^{\ 0}(i,j) = 0 \ \text{si}\ i = j \\ f^{\ 1}(i,j) = 1 \ \text{si ExisteDiccionario}(i,j) \ \text{con}\ 0 \le i,j \le I \\ f^{\ 1}(i,j) = +\infty \ \text{si no ExisteDiccionario}(i,j) \ \text{con}\ 0 \le i,j \le I \\ \min\left\{ f^{\ k-1}(i,j),\ f^{\ k-1}(i,k) + f^{\ k-1}(k,j) \ \text{con}\ 0 \le i,j \le I \ \text{y}\ 1 < k \right\} \end{cases}$$

El problema se resuelve con la invocación $f^{I}(a, b)$ siendo *a* y *b* los idiomas de los textos a traducir.

2) **(6 puntos)**

1. Dado que se tiene que indicar la secuencia de traducciones por las que se tiene que pasar, se necesita una matriz de dimensión *IxI* para almacenar los nodos intermedios por los que pasan los caminos mínimos.

Ejercicio 3 (20 puntos)

1) **(6 puntos)**

a) El problema se puede resolver como la concatenación de los caminos de mínimo costo entre cada par de vértices consecutivos de la secuencia. Se puede utilizar para encontrar cada camino mínimo:

   a. Un algoritmo ávido como lo es *Dijkstra*. Eventualmente puede modificarse este algoritmo para que se detenga una vez alcanzado el vértice final del par.

      Para que la solución propuesta funcione no debe haber aristas con costo negativo, dado que es una precondición para que funcione *Dijkstra.*

   b. El algoritmo de *Floyd*, en este caso la precondición es más laxa ya que se precisa que el grafo no tenga de ciclos de costo negativo. En este caso alcanza con ejecutarlo una sola vez.

b)

Por absurdo supongamos que existe un camino C' de menor costo que el camino C hallado por la solución propuesta. Como C' es solución, pasa por los elementos de la secuencia, en el mismo orden que C. Además, como C' es de menor costo que C, alguno de los tramos entre los elementos de la secuencia tiene que ser de menor costo para C' que para C. Esto es absurdo porque C elige el camino mínimo entre cada elemento de la secuencia, dado que no hay aristas de costo negativo.

2.

a) **(6 puntos)**

Obs. 1) Las instancias de D_n son infinitas, pero se puede observar que dado cualquier algoritmo y su AD asociado, todas las instancias que mantengan el mismo orden relativo entre sus elementos terminarán en el mismo nodo externo.

Obs. 2) En particular D_n contiene al conjunto D'_n formado sólo por las secuencias sin elementos repetidos. Agrupando en este caso aquellas instancias que tienen el mismo orden relativo se tienen n! posibles grupos de secuencias. Cada una de estas agrupaciones terminará en el mismo nodo externo y se tendrá, para D'_n , que m debe ser al menos n!.

Obs. 3) Al considerar D_n en lugar de D'_n lo que puede suceder es que se agreguen hojas al AD y tener un valor de m mayor (nunca puede disminuir m agrandando el conjunto de entradas). Como lo que se busca es el menor valor de m posible se tomará n! como dicho valor.

Es decir: $n! \leq m$

Y por lo tanto: $n! \leq m \leq 2k$

Como se trata de hallar una cota inferior para el peor caso, se trata de encontrar una cota inferior para k y entonces se debe hallar $Fw(n) \leq k$ para cualquier AD.

Entonces: $n! \leq 2k$

Tomando logaritmos en base 2:

- $k \geq \log (n!)$
- $\log(n!) = \sum_{i=1}^{n} \log i$

Como $\sum_{i=1}^{n} \log i \geq n \log n - 1.5n$ entonces

$k \geq n \log n - 1.5n \,.\, F_w(n) \in (n \log n)$

Concluimos que $F_w(n) \geq n \log n - 1.5n$.

De lo anterior se concluye que $F_w(n) \in \Omega(n \log n)$.

El merge sort y heap sort tienen costo $\Theta(n \log n)$ peor caso concluyéndose que la complejidad del problema de sorting basada en comparaciones binarias es de $\Theta(n \log n)$.

a) **(8 puntos)**

Las secuencias de D'$_n$ también son infinitas, por lo que se deberá obtener un punto de vista útil para trabajar con las probabilidades. Efectivamente existen infinitas secuencias, sin embargo es posible agruparlas en grupos equivalentes desde el punto de vista de la ejecución del algoritmo como en el caso anterior (peor caso).
Como todas las secuencias son equiprobables los grupos de secuencias también son equiprobables y considerando cualquier árbol de decisión, todos los nodos externos resultan equiprobables (como fin de ejecución).

Teniendo en cuenta la observación 3 vista para el peor caso, se tiene que la cantidad de nodos externos es exactamente n!.

Por definición se tiene para todo algoritmo de la clase C:

$$T_A(n) = \sum_{S \in D_n'} p(S) * T(S)$$

Donde

p(S) es la probabilidad que se de la secuencia S$\in$ D'n y
T(S) es la cantidad de comparaciones que realiza el algoritmo para S

Planteamos la sumatoria en términos de nodos externos del árbol de decisión:

$$T_A(n) = \sum_{iexterno} p_i k_i$$

Donde:

pi es la probabilidad de alcanzar el nodo externo i o sea pi = 1/n!
ki es el largo del camino de la raíz al nodo i o sea cantidad de comparaciones en ese caso.

Se tiene entonces:

$$T_A(n) = \sum_{iexterno} p_i k_i = \frac{1}{n!} \sum k_i$$

Por el lema 2 se sabe que la sumatoria de profundidades de los nodos externos del AD es mayor o igual a m log m , siendo m la cantidad de nodos externos.

La función FA buscada cumplirá: $F_A(n) \leq T_A(n) = \frac{1}{n!} \sum_{iexterno} k_i$

Como m = n!, y por Lema 2 se tiene: $F_A(n) \leq \frac{1}{n!} n! \log(n!)$

Se concluye:

$$F_A(n) \geq n \log n - 1.5n$$

$$F_A(n) \in \Omega(n \log n)$$

El merge sort, heap sort y quick sort tienen costo $\Theta(n \log n)$ caso promedio concluyéndose que la complejidad del problema de sorting basada en comparaciones binarias es de $\Theta(n \log n)$.