7529. Teoría de Algoritmos I
Trabajo Práctico 3

Ferreyra, Oscar, *Padrón Nro. 89563*
fferreyra38@gmail.com

Martin, Débora, *Padrón Nro. 90934*
demartin@fi.uba.ar

Eisner, Ariel, *Padrón Nro. 90697*
aeeisnerg@gmail.com

2do. Cuatrimestre de 2016

Índice

1. Clases de complejidad

1.1. Ciclos con peso negativo en un grafo Hamiltoniano

Para la resolución de este problema se propone la utilización del Algoritmo de Bellman-Ford. El mismo se utiliza para encontrar el camino más corto entre un vértice origen y todos los demás vértices.

El Algoritmo de Bellman-Ford es, en su estructura básica, muy parecido al algoritmo de Dijkstra, pero en vez de seleccionar vorazmente el nodo de peso mínimo aun sin procesar para relajarlo, simplemente relaja todas las aristas, y lo hace $|V| - 1$ veces, siendo $|V|$ el número de vértices en el grafo. Las repeticiones permiten a las distancias mínimas recorrer el árbol, ya que en la ausencia de ciclos negativos, el camino más corto solo visita cada vértice una vez. A diferencia de la solución voraz, la cual depende de la suposición de que los pesos sean positivos, esta solución se aproxima más al caso general.

Tiene una complejidad de tiempo de $O(V * A)$, donde V es el número de vértices y A el número de aristas.

A continuación mostramos un pseudocódigo del mismo.

```
bool BellmanFord( Grafo G, nodo_origen s )
  // inicializamos el grafo. Ponemos distancias a INFINITO menos el nodo origen que tiene distancia 0
  for v ∈ V[G] do
    distancia[v]=INFINITO
    predecesor[v]=NULL
  distancia[s]=0

  // relajamos cada arista del grafo tantas veces como número de nodos -1 haya en el grafo
  for i=1 to |V[G]| - 1 do
    for (u, v) ∈ E[G] do
      if distancia[v] > distancia[u] + peso(u, v) then
        distancia[v] = distancia[u] + peso (u, v)
        predecesor[v] = u

  // comprobamos si hay ciclos negativo
  for (u, v) ∈ E[G] do
    if distancia[v] > distancia[u] + peso(u, v) then
      print ("Hay ciclo negativo")
      return FALSE

  return TRUE
```

1.2. Ciclos de peso nulo en un grafo Hamiltoniano

Este problema es NP-Completo.

Primero, dado un ciclo simple en el grafo G, podemos determinar si la suma de los pesos de sus vértices en tiempo polinomial, luego el problema de encontrar un ciclo de peso 0 es NP.

Luego, reducimos el problema de la suma de subconjuntos a este problema. Este problema es: dado un conjunto de enteros, ¿existe algún subconjunto cuya suma sea exactamente cero? A continuación describimos el algoritmo de resolución de dicho problema:

Consideramos un conjunto de enteros $S = \{a_1, ..., a_n\}$. Construimos un digrafo ponderado G con $2n$ vértices, para el cual cada elemento a_i corresponde con dos vértices v_i y u_i

for each v_i
  agregar una arista (v_i , u_i) con peso a_i y agregar aristas (v_i , u_j) con peso 0, $\forall i, j, i \neq j$.
for each u_i
  agregar aristas (u_i , v_j) con peso 0, $\forall i, j, i \neq j$.
if (encontramos un ciclo de peso 0 en G) **then**
  todos los pesos desde v_j hasta u_j a lo largo del ciclo deben ser cero.
if (obtenemos un subconjunto $S_0 \subseteq S$ cuya sumatoria da 0)
  construimos un ciclo tomando todas las aristas (v_j, u_j) correspondientes al elemento en S_0 y conectamos sus aristas a través de las aristas de peso cero en G, y finalmente obtenemos un ciclo de peso 0.

Por lo tanto, el problema es al menos tan difícil como el de suma de subconjuntos. Dado que éste es un problema NP-Completo, el problema de los ciclos de peso 0 también lo es.

1.3. Tareas con ganancia igual a un k dado

Estamos ante un problema de Programación con deadlines, beneficios y duraciones ($PDBD$). Es un problema NP-Completo. Para probarlo, veremos que el tamaño de la resolución es mayor a otro problema NP-Completo, conocido como suma de subconjuntos (SDS).

El problema SDS tiene como entrada un vector $x = (a_1, a_2, ..., a_m, t)$ donde t y todos los a_i son enteros no negativos en formato binario.

Queremos ver que $SDS \leq_p PDBD$.

Para probar esto, sea x un input del problema de $PDBD, x = ((d_1, g_1, t_1), ..., (d_m, g_m, t_m), t))$, con todos los valores enteros no negativos en formato binario, y sea $f(x) = ((d_1, g_1, t_1), ..., (d_m, g_m, t_m), t, t))$. Supongamos que x es una instancia de SDS, por otra parte podemos dejar que $f(x)$ sea una cadena fuera de $PDBD$.

Es claro que f es computable en tiempo polinomial, y $x \in SDS \Leftrightarrow f(x) \in PDBD$.

1.4. Tareas con ganancia mayor a un k dado

Esta es una versión extendida del problema explayado en el punto anterior. La demostración es muy similar, pero debemos tener en cuenta que ahora podemos revisar una cantidad mayor de valores, es decir, todos los que cumplen la condición de ser mayores al k dado.

2. Algoritmos de aproximación

2.1. El problema del viajante de comercio

Para implementar la solución aproximada a este problema se utilizó el algoritmo de Prim para calcular el árbol de tendido mínimo. La complejidad de dicho algoritmo depende de las operaciones de una cola de prioridad que contiene los vertices, de las iteraciones que se realizan sobre todas las aristas de cada vértice. Se tomará $|V|$ como la cantidad de vértices del grafo y $|A|$ como la cantidad de aristas. Las operaciones más importantes de la cola de prioridad son $O(\log |V|)$ cada una, excepto el contains que es $O(|V|)$. Por otro lado el ciclo principal es $O(|V|^2 * |A|)$ porque recorre para cada vertice, sus aristas, y verifica si su arista adyacente está en la cola de prioridad. El mencionado sería el peor caso, pero a medida que va avanzando la ejecución, la cola se va vaciando y el contains pasa a ser insignificante. Por último, se agregan todos los vertices y las aristas pertenecientes al árbol en un nuevo grafo. Esta operación es $O(|A| \log |A|)$, en el peor caso que sería que todas las aristas del grafo original debieran ser agregadas al nuevo grafo. Pero en la mayoría de los casos será mucho menor y dependerá de la cantidad de aristas del árbol.

Además del algoritmo de Prim, luego se busca el preorder del árbol de tendido mínimo. Este recorre todos los vértices del grafo por lo que es $O(|V|)$.

Los tiempos de ejecución fueron llamativamente inferiores a los obtenidos por programación dinámica, aunque las soluciones obtenidas no resultaron siempre iguales a las anteriores. A continuación se muestra una comparativa entre la solución encontrada, el costo de esa solución y los tiempos de ejecución, para aquellas corridas que pudieron ser completadas. Todas ellas utilizaron el archivo de 48 ciudades dado como ejemplo. Las corridas con programación dinámica que no pudieron completarse fue por causa de falta de memoria. Es de notar que con la aproximación sí fue posible realizar la corrida para grafos con mayor cantidad de vertices.

Cantidad de ciudades	Camino Aproximación	Camino Dinámica
15	[0, 7, 8, 14, 6, 5, 11, 10, 12, 13, 2, 4, 1, 9, 3, 0]	[0, 7, 8, 6, 5, 14, 11, 10, 12, 13, 4, 9, 3, 1, 2, 0]
17	[0, 7, 8, 14, 6, 5, 16, 11, 10, 12, 13, 4, 1, 9, 3, 15, 2, 0]	[0, 15, 2, 1, 3, 9, 4, 13, 12, 10, 11, 14, 16, 5, 6, 8, 7, 0]
19	[0, 7, 8, 14, 11, 10, 12, 13, 4, 1, 9, 3, 17, 6, 5, 18, 16, 15, 2, 0]	[0, 15, 2, 1, 3, 9, 4, 13, 12, 10, 11, 14, 16, 18, 5, 6, 17, 8, 7, 0]
21	[0, 7, 8, 14, 11, 10, 12, 13, 4, 1, 9, 3, 20, 19, 17, 6, 5, 18, 16, 15, 2, 0]	—
23	[0, 7, 8, 21, 2, 22, 10, 11, 14, 17, 6, 5, 18, 16, 19, 13, 4, 1, 9, 3, 12, 20, 15, 0]	—
48	[0, 7, 8, 37, 30, 43, 17, 6, 27, 5, 29, 36, 18, 26, 16, 42, 35, 39, 14, 11, 10, 22, 13, 24, 12, 38, 31, 47, 4, 28, 1, 41, 9, 23, 25, 3, 34, 44, 33, 40, 46, 20, 32, 19, 45, 21, 2, 15, 0]	—

Cuadro 1: Camino en función de la cantidad de ciudades

Cantidad de ciudades	Costo del camino Aproximación	Costo del camino Dinámica
15	24357	20357
17	25167	22459
19	27980	22514
21	28197	—
23	29558	—
48	43980	—

Cuadro 2: Costo del camino en función de la cantidad de ciudades

Cantidad de ciudades	Tiempo de ejecución Aproximación (seg)	Tiempo de ejecución Dinámica (seg)
15	$4{,}86 * 10^{-3}$	16
17	$5{,}74 * 10^{-3}$	299
19	$8{,}82 * 10^{-3}$	7237
21	$6{,}35 * 10^{-3}$	—
23	$6{,}15 * 10^{-3}$	—
48	$13{,}4 * 10^{-3}$	—

Cuadro 3: Tiempo en función de la cantidad de ciudades

Como se puede apreciar, la aproximación no devuelve el mejor camino, ya que el por programación dinamica habíamos encontrado caminos de menor costo. Sin embargo, la diferencia no es tanta si se consideran los tiempos de ejecución de cada método, sumado a la imposibilidad de obtener un resultado con dinámica en los casos con mayor cantidad de ciudades debido a las restricciones de memoria.

2.2. El problema de la mochila

Se implementó el algoritmo propuesto en el enunciado.

Este algoritmo se ejecuta en $O(n^2 * sumaValores)$ siendo n el numero de elementos a introducir en la mochila, y sumaValores la suma de los valores (no pesos) de todos los objetos.

A continuación se muestra una tabla con los tiempos de ejecución del algoritmo con algunos de los archivos de prueba, sin utilizar un factor de normalización. Como vemos, el algoritmo tarda bastante más que la solución anterior, pero calcula el valor óptimo (omitimos mostrarlo).

Archivo de prueba	Tiempo de ejecución aprox (seg)	Tiempo de ejecución original (seg)
knapPI_11_500_1000	0,872808402	0,037109291
knapPI_12_500_1000	0,952657729	0,015955889
knapPI_13_500_1000	0,563738462	0,020642450
knapPI_14_500_1000	0,992060191	0,011673355
knapPI_15_500_1000	0,887167806	0,011940223
knapPI_16_500_1000	1,001233293	0,013271072

Cuadro 4: Tiempo en función de la cantidad de paquetes y el tamaño de la mochila

Ahora mostramos los resultados, tanto en tiempo de ejecución como en valor resultante hallado, aplicando distintos factores de normalización. Dicho factor se aplica de la siguiente manera:

$u_i = \lceil v_i / factorNormalizacion \rceil$

siendo su factor normalización

$factorNormalizacion = (epsilon/n) * max_i v_i$

Cada u_i corresponde a cada uno de los valores a cargar normalizado.

Por lo tanto, el algoritmo utilizado calculando óptimo de la mochila con este factor es del orden $O(n^3 * sumaValores)$, ya que tiene que recorrer todos los valores y aplicarles dicho factor.

A continuación la tabla comparativa de tiempos y valores hallados, respecto del óptimo, para distintos valores de *epsilon*.

epsilon	0.1	0.5	0.9	
Archivo de prueba	t (seg) / val	t (seg) / val	t (seg) / val	Valor óptimo
knapPI_11_500_1000	3,175835430 / 17000	0,735377199 / 3400	0,426913432 / 1904	3468
knapPI_12_500_1000	3,327861014 / 11500	0,850884103 / 2300	0,449287974 / 1288	2231
knapPI_13_500_1000	2,508101185 / 13692	0,477538811 / 2761	0,362973003 / 1555	3198
knapPI_14_500_1000	3,838261431 / 50123	0,797174056 / 10042	0,495847542 / 5592	12746
knapPI_15_500_1000	3,384585335 / 12583	0,692684757 / 2551	0,431822064 / 1425	2508
knapPI_16_500_1000	− − −	0,917216243 / 11425	0,35268085 / 6358	13145

Cuadro 5: Tiempo y valor obtenido en función del epsilon, la cantidad de paquetes y el tamaño de la mochila