Programación Dinámica y Greedy

Algoritmos y Estructuras de Datos Avanzadas

Informe Teórico

Nielsen Maximiliano - Uliassi Manuel

Docente: Juan Manuel Rabasedas

6to Informática

Instituto Politecnico Superior Gral. San Martin

Informe Teórico

Objetivo

El objetivo de este Informe Teorico es explicar y complementar con teoria la resolución de los siguientes problemas de Programación Dinámica y Algoritmos Greedy asignados.

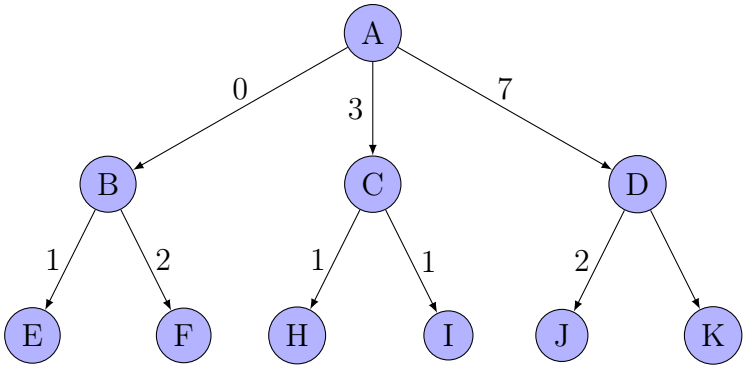
Problema de Programación Dinámica

Dado un arreglo $A[a_1, ..., a_n]$ de enteros no negativos, encontrar una subsecuencia $A[a_1, ..., a_j]$ de tamaño máximo tal que: i = j ó i = j + 1 ó para todo $r \geq 0$, valen las desigualdades $a_{i+r} \geq \sum_{k=i+r+1}^{j-r-1} a_k$ y $a_{j-r} \geq \sum_{k=i+r+1}^{j-r-1} a_k$. Por ejemplo, en el arreglo [1, 8, 2, 1, 3, 9, 10] la subsecuencia [8, 2, 1, 3, 9] cumple la propiedad, ya que $8 \geq 2+1+3$ y $9 \geq 2+1+3(r=0)$ y además $2 \geq 1$ y $3 \geq 1$ (r=1). Escriba un algoritmo de programación dinámica bottom-up para resolver el problema. La salida del algoritmo son dos índices $i \leq j$ tales que $A[a_i, ..., a_j]$ es una subsecuencia de tamaño máximo.

Problema de Algoritmos Greedy

Dado un árbol con pesos no negativos en los lados, se quiere resolver el problema de encontrar el mínimo camino desde la raíz hasta una hoja. Considere el algoritmo greedy que, partiendo desde la raíz como nodo actual, elige siempre el lado l con menor peso. El nuevo nodo actual es el apuntado por l. El algoritmo termina cuando el nodo actual es una hoja.

- Encuentre un ejemplo en que el camino encontrado por el algoritmo no es óptimo.
- Encuentre una condición C sobre árboles de manera tal que, si un árbol cumple con C, entonces el camino encontrado por el algoritmo en ese árbol es optimo. El árbol que se muestra en la figura debe cumplir la condición. Sugerencia: piense en árboles con pesos muy grandes cerca de la raíz y pesos muy chico cerca de las hojas.



- Pruebe que la condición encontrada en el punto anterior asegura que el algoritmo encuentra el óptimo.

Problema Programación Dinámica

Conceptos Básicos

Para resolver este problema utilizamos un algoritmo Bottom-Up, lo que hacemos es primero resolver los problemas más chicos y combinar estas resoluciones para poder resolver los problemas más grandes. Primero vamos a dejar en claro algunas definiciones que vamos a usar en la aplicación del algoritmo: La secuencia principal, aquella que se ingresa, la vamos a llamar **S**, la cantidad de elementos de **S** se va a llamar **N**. El enunciado hace referencia a tres condiciones posibles para que la subsecuencia de **S** cumpla con la propiedad correspondiente, a estas condiciones les asignamos un nombre a cada una. Por un lado tenemos la **C1**, es cuando dada la subsecuencia A[ai,...,aj] j es igual a i, por otro lado tenemos la **C2**, es cuando dada la subsecuencia A[ai,...,aj] i es igual a j+1, y por último tenemos la **C3**, se da cuando tenemos una subsecuencia y para todo r â‰¥ 0, valen las desigualdades $a_{i+r} \geq \sum_{k=i+r+1}^{j-r-1} a_k$ y $a_{j-r} \geq \sum_{k=i+r+1}^{j-r-1} a_k$. Cuando una subsecuencia de **S** cumple con **C1**, con **C2** o con **C3** esta va a pasar a llamarse **SCC**. La **SCC** de mayor longitud va a llamarse **SCCO** y va a ser la subsecuencia que el algoritmo devuelve como resultado.

Explicación del Algoritmo

Para el inicio del algoritmo vamos a determinar dos propiedades, la primera es que si una subsecuencia de **S** es de longitud uno entonces es una **SCC** gracias a que cumple con la **C1** y la segunda es que si una subsecuencia de **S** es de longitud dos entonces es una **SCC** gracias a que cumple con la **C2**. Una vez determinado esto vamos a empezar con el algoritmo.
El primer paso va a ser seleccionar todas las subsecuencias de **S** de longitud tres y comprueba si los elementos de los extremos son mayores o iguales al elemento del medio y si se cumple esta condición entonces se especifica en una matriz cual cumple con esta condición y cuál no.
El siguiente paso consiste en comprobar cuales subsecuencias de **S** de longitud cuatro cumplen la propiedad en que los extremos son mayores o iguales a la sumatoria de los elementos que no son de los extremos, la **C3**,y distinguir en la matriz cuales cumplen con esto y cuales no.
Cuando llegamos al tercer paso, el paso para analizar las subsecuencias de longitud 5, vamos a comprobar para todas las subsecuencias de longitud 5 si los elementos de los extremos son mayores o iguales a la sumatoria de los elementos que no están en los extremos y comprobar que suceda lo mismo en la subsecuencia conformada por los elementos que no se encuentran en los extremos, la **C3**, pero como anteriormente ya calculamos aquellas subsecuencias que pueden ser conformadas por estos elementos que no se encuentran en los extremos entonces vamos a tener que buscar la respuesta a este cálculo en la matriz en donde se guardaron los datos anteriormente.
Por ejemplo: En el caso de la Secuencia [1,8,2,1,3,9,10] en el tercer paso analizando la subsecuencia [8,2,1,3,9] cuando se tenga que comprobar si la subsecuencia [2,1,3] cumple con la **C3** este cálculo ya fue realizado y almacenado en la matriz mencionada anteriormente, entonces solo hay que buscar este resultado y no calcularlo.

Aunque parezca ser un algoritmo de fuerza bruta, este aprovecha los cálculos que se llevan a cabo durante el proceso utilizándolos más adelante. Esta es la diferencia principal entre resolver este problema con fuerza bruta y resolverlo con Bottom-Up.

Utilización de matriz

Utilizaremos la lista [1,8,2,1,3,9,10] de ejemplo como ***S***. Para comenzar con la inicialización crearemos una matriz identidad de 7 por 7 (N*N). Las casillas de valor uno que se encuentran a lo largo de la diagonal de la matriz representan que para las subsecuencias definidas entre las posiciones i y j de S (siendo i y j las coordenadas de la casilla) la ***SCC*** contenida en dicha subsecuencia es de longitud uno.

	1	8	2	1	3	9	10
i/j	1	2	3	4	5	6	7
1	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0
4	0	0	0	1	0	0	0
5	0	0	0	0	1	0	0
6	0	0	0	0	0	1	0
7	0	0	0	0	0	0	1

Una vez calculada la diagonal comenzará el algoritmo, siguiendo el siguiente pseudocódigo:

condicion i j S matriz:
S[i] >= sum S[i+1 ... j-1] and S[j] >= sum S[i+1 ... j-1] and matriz[i+1][j-1] > max(matriz[i+1][j-2],matriz[i+2][j -1],matriz[i+2][j-2])
if condicion i j S matriz:
matriz[i][j] = matriz[i+1][j-1] + 2
else:
matriz[i][j] = max(matriz[i][j-1],matriz[i+1][j],matriz[i+1][j-1])

Primero tomara una casilla y sus coordenadas (i,j). Comprobará si el elemento de $\boldsymbol{S}$ en i y el elemento de $\boldsymbol{S}$ en j son mayores o iguales a la sumatoria de los elementos de $\boldsymbol{S}$ comprendidos entre i y j, también comprobaremos si la subsecuencia sin los elementos i y j de $\boldsymbol{S}$ también es una $\boldsymbol{SCC}$. Si esto se cumple implica que la subsecuencia de $\boldsymbol{S}$ delimitada por i y j es una $\boldsymbol{SCC}$, por ende esta se pudo extender una vez más que la anterior encontrada. Aquí es donde podemos observar la ventaja de nuestro algoritmo DP frente a la fuerza bruta. No hace falta calcular si la condición C3 se cumple hacia dentro recursivamente, dado que ya lo tenemos calculado en las casillas aledañas. únicamente tenemos que tomar la subsecuencia sin los elementos extremos y añadirle 2, la longitud de esta se encuentra en la casilla (i+1,j-1).
Si la comprobación resultara falsa el algoritmo buscará cual es el mayor valor entre las casillas a su izquierda, debajo y debajo a su izquierda de esta. Almacena este valor en vez de la nueva longitud de la $\boldsymbol{SCC}$, ya que este valor será el valor de la $\boldsymbol{SCC}$ contenida en esta subsecuencia delimitada por i y j.

	1	8	2	1	3	9	10
i/j	1	2	3	4	5	6	7
1	1	2	2	2	4	5	5
2	0	1	2	2	4	5	5
3	0	0	1	2	3	3	3
4	0	0	0	1	2	2	2
5	0	0	0	0	1	2	2
6	0	0	0	0	0	1	2
7	0	0	0	0	0	0	1

Una vez calculada la matriz encontraremos la $\boldsymbol{SCCO}$ en base a que la longitud de esta se encuentra almacena en la esquina superior derecha de la matriz, llamaremos a este valor E. Por lo tanto buscaremos en que casilla la $\boldsymbol{SCCO}$ realiza su última extensión.
Para eso nos fijamos cual es el mayor valor entre la casilla izquierda, la de debajo y la de debajo a la izquierda. Si el mayor valor coincide con E nos moveremos a esa casilla y volveremos a mirar las 3 casillas aledañas. Si el valor es menor sabremos que en esa casilla la $\boldsymbol{SCCO}$ realizo su ultima extension. Siguiendo esta recursión de comprobar si el mayor valor aledaño es menor a E podremos encontrar facilmente que subsecuencia de $\boldsymbol{S}$ es la $\boldsymbol{SCCO}$, ya que en las coordenadas i y j de la casilla en donde se realiza la última extensión representan la posición de inicio y la final de la subsecuencia de $\boldsymbol{S}$ respectivamente.

Prueba de optimalidad

Debemos probar que nuestra excursión nos lleva a una solución óptima

DP[i,j] nos devuelve el largo de la SCC en la subsecuencia S[i..j]
SCCO (S[i..j]) nos devuelve la SCCO de la subsecuencia S

Si S[i] >= Sum S[(i+1)...(j-1)] y S[j] >= Sum S[(i+1)...(j-1)] con i = 1
Sea r[1..k] = SCCO (S[i..j]) con DP[i,j] = k
r[k] = S[j] y r[1] = S[i]

Si r[1..k] no incluye a S[i] y S[j] como su primer y Ítimo elemento podríamos extender a r ya que S[i] >= Sum S[(i+1)...(j-1)] y S[j] >= Sum S[(i+1)..(j-1)]
Luego r[2..k-1] es una SCC de S[i+1...j-1]

Que me gustaría que pase?

Me gustaría que:

r[2..k-1] sea SCCO de S[i+1...j-1]
Supongamos que $\boldsymbol{w}$ es una SCC de más larga de S[i+1...j-1]

Es decir que |w| > k - 2 *(k - (1 + 1) dado que es i + 1 y j - 1)*

Luego concatenando w con r[k]

[w[i]] ++ r[k] ++ w[j]

obtenemos una SCC de S[i..j] con | [w[i]] ++ r[k] ++ w[j] | > k

Contradicción!

Entonces, DP[i+1,j-1] = k - 2 lo que implica que DP[i,j] = DP[i+1,j-1] + 2 en el caso S[1] >= Sum S[2..(j-1)] y S[j] >= Sum S[2..(j-1)]

Los otros casos son similares

Implementación

Para la implementación utilizamos el módulo de haskell Data.Matrix que implementa funciones útiles para trabajar con matrices.

La función identity n crea una matriz de identidad de n por n.

La función getElem j i mx obtiene el elemento (i,j) de la matriz mx.

La función setElem a j i mx escribe el elemento (i,j) de la matriz mx con el valor a.

```
import Data.Matrix

--Calcula el maximo entre 3 elementos
max3 :: Ord a => a -> a -> a -> a
max3 a b c = if a >= b then
    (if a >= c then a else
    (if b >= c then b else c)) else
```

```
    (if b >= c then b else c)

--Calcula el maximo entre 3 tuplas de 3 elementos y devuelve una tupla de 2 elementos
max3ij :: Ord a1 => (a1, a2, b) -> (a1, a2, b) -> (a1, a2, b) -> (a2, b)
max3ij (av, ai, aj) (bv , bi, bj) (cv, ci, cj) = if av >= bv then
    (if av >= cv then (ai,aj) else
    (if bv >= cv then (bi, bj) else (ci,cj))) else
    (if bv >= cv then (bi,bj) else (ci,cj))

-- Chekea que la subsecuencia entre i y j sea una SCC en base a si fue extendida en la casilla (i,j)
cond2 :: Int -> Int -> Matrix Int -> Bool
cond2 i j mx |j <= 0    = True
             |otherwise = getElem i j mx > (max3
                (getElem i (j-1) mx)
                (getElem (i+1) j mx)
                (getElem (i+1) (j-1) mx))

--Checkea que los elementos en las posiciones i y j sean maryores a la suma
--de los elementos entre ellas y que la subsecuencia sin los elementos i y j es una SCC
cond :: (Ord a, Num a) => Int -> Int -> [a] -> Matrix Int -> Bool
cond i j orgn mx |j-i == 1 = True
                 |otherwise= let
                        f = [ x | (x,a) <- zip orgn [1..], a > i && a < j ]
                    in
                        (orgn!!(i-1) >= sum f) && (orgn!!(j-1) >= sum f) && cond2 (i+1) (j-1) mx


--Corta una lista apartir de la posicion a hasta b
trim :: (Num a1, Enum a1, Ord a1) => a1 -> a1 -> [a2] -> [a2]
trim a b orgn = [ x | (x,i) <- zip orgn [1..], i >= a && i <= b ]

-- Calcula la matriz que permite hacer el calculo del algoritmo
-- bdp :: Int          Coordenada I
-- -> Int              Coordenada J
-- -> Int              Inicio, generalmente es 2
-- -> Int              Fin, Uno mas que el largo de la lista
-- -> [Int]            Lista de input
-- -> Matrix Int       Matriz en la q va la respuesta
bdp :: (Ord a, Num a) => Int -> Int -> Int -> Int -> [a] -> Matrix Int -> Matrix Int
bdp  i j b e orgn mx | (i == 1 && j == e)    = mx
bdp  i j b e orgn mx | i == e || j == e      = bdp 1 b (b+1) e orgn mx
bdp  i j b e orgn mx | otherwise             =
        if cond i j orgn mx
                then bdp (i+1) (j+1) b e orgn (setElem ((getElem (i+1) (j-1) mx)+2) (i,j) mx)
                else bdp (i+1) (j+1) b e orgn (setElem (max3
                    (getElem i (j-1) mx)
                    (getElem (i+1) j mx)
                    (getElem (i+1) (j-1) mx))
                    (i,j) mx)


-- Recorre la matriz generada por bdp para obtener la subsequencia
-- adp :: Int       Coordenada I, tendria que ser 1
-- -> Int           Coordenada J, tendria que ser el largo de la lista
-- -> [Int]         Lista del input
```

```
-- -> Matrix Int    Matriz precalculada
-- -> Int           Longitud de la subsequencia,
--                  elemento de la matriz en la posicion I = 1 y J = Largo de la lista
adp :: Ord t => Int -> Int -> [a] -> Matrix t -> t -> [a]
adp i j orgn mx last | last > (max3
                                (getElem i (j-1) mx)
                                (getElem (i+1) j mx)
                                (getElem (i+1) (j-1) mx))
                               = trim i j orgn
adp i j orgn mx last | otherwise = adp (fst maximum) (snd maximum) orgn mx last
                        where         maximum = (max3ij
                                              ((getElem i (j-1) mx),i,j-1)
                                              ((getElem (i+1) j mx),i+1,j)
                                              ((getElem (i+1) (j-1) mx), i+1, j-1))

-- Coordina bdp y adp
dp :: (Ord a, Num a) => [a] -> [a]
dp orgn =   let
    l = length orgn
    mx = bdp 1 2 2 (l+1) orgn (identity l)
    ms = getElem 1 l mx
    in
        adp 1 l orgn mx ms
```

Análisis del costo de DP frente a Fuerza Bruta

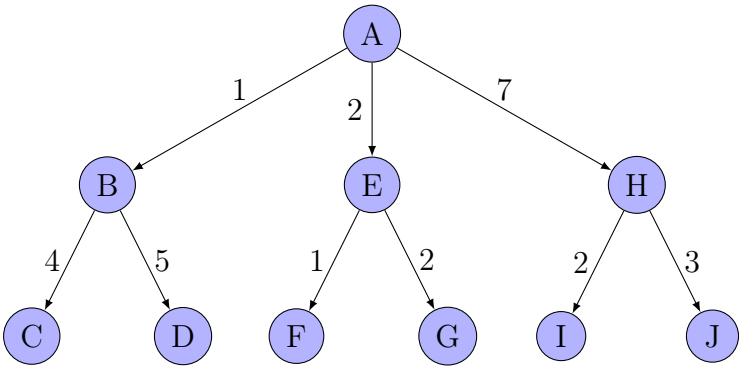
Fuerza Bruta:
calcular si una subsecuencia cumple la condicion: $O(n^2)$
calcular todas las subsecuencias: $O(n^2)$
Costo total: $O(n^4)$
Programación Dinámica:
calcular si una subsecuencia cumple la condición: O(n)
calcular las subsecuencias: $O(n^2)$
Costo total: $O(n^3)$

Como podemos observar nuestro costo al implementar DP disminuye en un factor de N para el largo de la secuencia de entrada.

Problema Greedy

No óptimo

El algoritmo planteado no resulta óptimo para cualquier tipo de árbol. Esto se puede ver en el siguiente ejemplo:



El algoritmo para este árbol devuelve el camino *{A,B,C}* que tiene un peso de 5 mientras que la resolución del problema da como resultado el camino *{A,E,F}*, que tiene peso de 3.
El problema más frecuente de este algoritmo es que al no tener en cuenta el árbol en su totalidad opta por una arista liviana con hijas pesadas cuando la opción más óptima es optar por una arista pesada con hijas livianas.

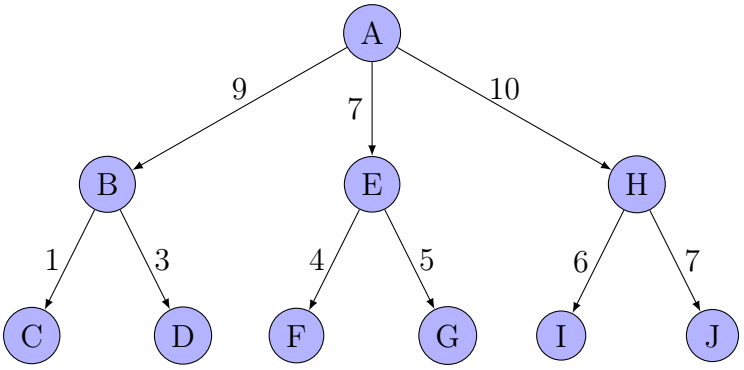
Condiciones específicas

El algoritmo no funciona para cualquier tipo de árbol. Establecimos dos condiciones específicas que debe cumplir un árbol para que el algoritmo de como resultado el mínimo camino de este.
Una rama de un árbol es el camino desde la raíz del mismo hacia cualquiera de sus hojas, el peso de una rama se calcula sumando el valor de cada arista por la que pasa la rama para poder llegar a la hoja correspondiente.

Condición 1 (C1)
La primera condición que debe complir el arbol es que las aristas hermanas deben estar ordenadas de forma creciente.

Condición 2 (C2)
La segunda condición que debe cumplir el árbol es que las ramas de este tienen que estar ordenadas de manera creciente según su peso.

La condición ***C2*** implica que el camino de la raíz a una hoja más liviano siempre será el que se encuentre más a la izquierda. Sin embargo esta condición por si sola no garantiza que tomando siempre la arista de menor peso encontraremos la rama de menor peso.
Veamos el siguiente ejemplo:



Si bien el árbol se encuentra ordenado por ramas, tomar las aristas de menor peso no nos lleva a la rama de menor peso. Tomar las aristas de menor peso resultaría en el camino ***{A,E,F}*** con un costo igual a 11. Mientras que el camino más liviano de la raíz a una hoja se encuentra a la izquierda, ***{A,B,C}*** con un costo de 10.
Esto muestra que además se debe cumplir la condición ***C1***. Esta condición establece que las aristas hermanas deben estar ordenadas por peso de forma creciente. Esto implica que la arista más a la izquierda siempre será la de menor peso. A su vez, gracias a la condición ***C2***, la rama más izquierda será la de menor peso.
De esta forma se prueba que siempre la rama más izquierda contendrá las aristas de menor peso con respecto a sus hermanas y además llevará al camino más liviano hacia una hoja.
Probando así que la elección de la arista más liviana siempre llevará a la rama de menor peso, es decir al camino de menor peso de la raíz a una hoja, siempre que el árbol cumpla con las condiciones ***C1*** y ***C2***.

Implementación del algoritmo Greedy

```haskell
-- El Int es para la arista que da origen al nodo, en el caso de la raiz es 0
data Tree a = N Int a [Tree a] deriving (Show)

-- Funcion que toma una listas de arboles y devuelve el arbol cuya
--arista que le da origen es la de menor peso
tmin :: [Tree a] -> Tree a
tmin [x] = x
tmin (x:xs) = let
minim (x@(N xe xa xs)) (y@(N ye ya ys)) | xe > ye  = y
| xe <= ye = x
in
minim x (tmin xs)

-- Funcion que ejecuta el algoritmo Greedy
greedy :: Tree a -> [a]
greedy (N e a []) = [a]
greedy (N e a xs) = a : greedy (tmin xs)
```