Ampliación de Programación

Práctica 4

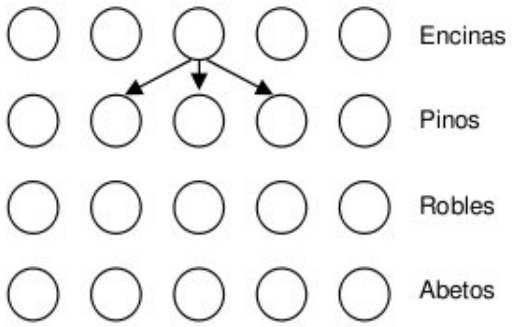
Programación Dinámica

Sergio García Mondaray

Escuela Superior de Informática de Ciudad Real
Universidad de Castilla-La Mancha

1 Enunciado

Un leñador trabaja en una plantación en forma de cuadrícula de MxN árboles. Cada fila del bosque está formada por M árboles del mismo tipo, y el bosque tiene N filas de árboles de diferente tipo. Cada árbol tiene una anchura propia A que determina el tiempo que tarda en ser talado. Un leñador tiene que talar N árboles todos de tipos diferentes. Para ello, el leñador va talando el bosque de fila en fila, de forma ordenada. El leñador sólo puede talar los árboles de la fila siguiente que están o bien en la vertical o en la diagonal con respecto al árbol que acaba de talar. Véase la figura adjunta. El problema consiste en encontrar una solución que minimice el tiempo que tarda el leñador en talar N árboles, cada uno de diferente tipo.



Diseñe algoritmos que resuelvan el problema basándose en: a) Estrategia Voraz, b) Programación Dinámica. Calcule la complejidad de los algoritmos desarrollados y haga una reflexión sobre la optimalidad de la solución según el método empleado.

2 Planteamiento y resolución del problema

2.1 Representación abstracta

2.1.1 El bosque

Nuestra forma de representar el conjunto de árboles será una matriz bidimensional, donde las filas representarán los tipos de árboles. En cada posición de la matriz se almacenará el tiempo necesario para talar el árbol que ocupa esa posición.

	0	1	2	3	4
0	51	12	58	55	22
1	23	49	16	58	51
2	14	12	42	25	25
3	37	11	41	56	58
4	55	36	41	24	18

Por ejemplo, la matriz adjunta representaría un bosque con 5 tipos diferentes de árboles (N), y 5 árboles de cada tipo (M). El árbol que ocupa la posición (2,3) tardaría 25 uds. de tiempo en talarse; el que ocupa la posición (4,1) tardaría 36 uds; el de la posición (0,2) tardaría 58 uds, etc.

2.1.2 La solución

Nuestra solución será almacenada en un vector de tamaño N (número de tipos de árboles). Cada posición representará una fila, y el valor almacenado en ella la columna correspondiente al árbol a talar en dicha fila.

2.2 Estrategia Voraz

2.2.1 Heurística

Nuestra manera de *devorar* el problema estará basada en la heurística voraz definida en los siguientes términos:

1. **Lista de candidatos:** En cada estado, los candidatos serán los árboles accesibles desde el punto en el que nos encontramos.
2. **Conjunto de decisiones ya tomadas:** El conjunto de árboles ya talados.
3. **Función selección:** Seleccionaremos siempre el árbol cuya anchura (o tiempo en talar) sea menor, de entre todos los que podemos talar desde la posición donde estamos.
4. **Función completable:** No se añadirán al conjunto solución los árboles más caros de entre los accesibles.
5. **Función solución:** Habremos finalizado cuando tengamos talados un árbol de cada tipo.
6. **Función objetivo:** Lo que caracterizará lo buena o mala que es nuestra solución es el tiempo total empleado en talar todos los árboles.

En resumen, nuestro algoritmo recorerá las filas de arriba a abajo. En la primera fila talará el árbol que menos tiempo necesite; en este punto los candidatos serían los 3 árboles accesibles de la fila inmediatamente inferior (2 en el caso de las columnas extremas). Talará el mejor candidato (el árbol más rapido en talar) y seleccionara como candidatos los nuevos árboles accesibles, de la fila inferior, desde el árbol recién talado. Y así sucesivamente, hasta que talemos un árbol de la última fila (pues ya tendremos un árbol de cada tipo).

2.2.2 Algoritmo voraz

Según la heurística definida en el apartado anterior, nuestro algoritmo voraz quedaría de la siguiente manera:

```
INICIO
    aux <- menorEnFila(0) //Devuelve la posicion j del menor
    solucion[0] = aux
    PARA(i = 1; i < numeroDeFilas; i++)
        aux <- menorDeAccesibles(i-1,aux) //Devuelve la posicion j (columna)
        solucion[i] = aux;
    FIN PARA
FIN
```

2.2.3 Complejidad voraz

El número de operaciones básicas que realiza el algoritmo anterior, por partes, es:

- M, por recorrer la primera fila del bosque en busca del árbol de menor tiempo.
- $3 \cdot (N-1)$, porque para el elegido de cada fila (menos de la primera) recorremos los 3 elementos accesibles (en el peor caso, puesto que para las columnas extremas sólo hay 2 accesibles), en busca de la mejor opción.

Con todo esto:

$$T_n = M + 3 \cdot (N-1) = M + 3 \cdot N - 3 \in \boxed{\text{O(N+M)}}$$

2.3 Programación Dinámica

Hemos resuelto el problema mediante una estrategia *forward*, por tanto, empleando memoria. La manera de proceder ha sido rellenar una tabla (de las mismas dimensiones que el bosque generado) donde las filas representan los tipos de árboles y cada columna un árbol. En la posición correspondiente a cada árbol se han almacenado dos valores: el tiempo total empleado en talar dicho árbol mas el tiempo mínimo empleado en llegar hasta él, y un valor que representa la columna donde se encuentra el árbol de la fila anterior del camino más corto que nos ha podido llevar hasta el árbol.

De esta manera, al llegar a la última fila tendremos tendremos los tiempos acumulados por los caminos más rápidos que nos llevan hasta cada árbol de ese último tipo. Bastará, por tanto, seleccionar el tiempo menor y reconstruir el camino (con el valor de la columna donde se encuentra el árbol de la fila anterior). Todo esto se verá más claro con el siguiente ejemplo, donde se muestra la tabla forward para el bosque de ejemplo del apartado 2.1.1:

```
   ||  0   |  1   |  2   |  3   |  4   |
-----------------------------------------
 0 || 51(0)| 12(0)| 58(0)| 55(0)| 22(0)|
 1 || 35(1)| 61(1)| 28(1)| 80(4)| 73(4)|
 2 || 49(0)| 40(2)| 70(2)| 53(2)| 98(4)|
 3 || 77(1)| 51(1)| 81(1)|109(3)|111(3)|
 4 ||106(1)| 87(1)| 92(1)|105(2)|127(3)|
```

Para el árbol de la posición (2,1), el camino hasta él será de 28 uds. de tiempo, sumándole el tiempo en talarlo (12), el tiempo queda en 40. El paréntesis adjunto indica que de la fila anterior (la 1) se elige el árbol de la columna 2 (puesto que conduce a una mejor solucion que cualquier otro).

Se puede apreciar que la primera fila siempre será igual en esta tabla que en el bosque, puesto que es el principio de los caminos, y el tiempo comienza siendo el de cada árbol de los primeros.

Para extraer el camino de menor duración en talar, buscamos el menor tiempo de la última fila (87). Ahora, sabemos que de la fila 4 se tala el árbol 1. De la fila anterior se taló el árbol de la columna 1, y fijándonos en ese, observamos que el anterior fue el de la columna 1 también. De éste último pasamos a la columna 2, y de ese a la columna 1.

2.3.1 Algoritmo

```
INICIO

    //Rellenamos la primera fila igual
    PARA (j = 0; j < m; j++)
        sol[0][j].valor = bosque[0][j]
    FIN PARA

    //Para el resto de filas:
    PARA (i = 1; i < n; i++)
        PARA (j = 0; j < m; j++)
            soldin.sol[i][j].yanterior = menorDeAccesibles();
            soldin.sol[i][j].valor = bosque[i][j] +
                   soldin.sol[i - 1][soldin.sol[i][j].yanterior].valor;
```

```
        FIN PARA
    FIN PARA

    // Ahora sacamos el camino
    camino[n - 1] = menorDeUltimaFila();
    y = soldin.sol[n - 1][menorDeUltimaFila()].yanterior;
    PARA (i = n - 2; i >= 0; i--)
        camino[i] = y;
        y = soldin.sol[i][y].yanterior;
    FIN PARA

FIN
```

2.4 Complejidad

El número de operaciones básicas que realiza el algoritmo anterior, por partes, es:

- M, por copiar la primera fila del bosque en la matriz forward.
- $M \cdot (N-1) \cdot 3$, porque para cada celda de la tabla (excepto para la primera fila) calculamos el menor de los 3 árboles accesibles (peor caso).
- N, por construir la solución cogiendo un elemento de cada fila.

Con todo esto:

$$T_n = M + M \cdot (N-1) \cdot 3 + N = M + 3MN - 3M + N = M(3N-3) + N \in \boxed{\text{O(MN)}}$$

3 Comparación de soluciones

El método voraz ofrece una buena solución de manera rápida (orden de complejidad O(N+M)), sin embargo normalmente no será la solución óptima, ya que en algunas ocasiones pasar por un árbol de mayor tiempo puede venirnos bien a la larga, cosa que el algoritmo voraz no contempla. Por el contrario, el método de programación dinámica encontrará siempre la mejor solución, puesto que comprueba todas las posibles, pero a costa de ser más lento que el voraz (orden de complejidad O(NM)).

Ejecutando el programa adjunto se puede observar cómo prácticamente nunca coinciden las soluciones voraz y dinámica, ofreciendo esta última siempre el resultado óptimo.

4 Tiempo de trabajo y valoración de dificultad

Aproximadamente, he invertido unas 8 horas resolviendo esta práctica y unas 2 horas redactando la memoria. La valoración de dificultad total que le asigno a esta tarea es de 6 sobre 10.