Practica programación II

Diseño de algoritmos

15/04/2016

Gorka Bravo
Facultad de Informática
Universidad del País Vasco, UPV/EHU

Indice

1. Problema transportar miles de litros de agua:

Tenemos un deposito con L miles de litros de agua, que tenemos que trasvasar completamente. Para el trasvase tenemos que utilizar un tren que transporta otros n depositos con capacidad suficiente, pero que están defectuosos y sabemos que pierden parte del agua que intentemos transportar en ellos. La pérdida estimada de cada deposito depende de la cantidad de litros que se pongan en el. Disponemos de una tabla P(1 ... n; 1 ... L) en la que P(i; k) informa de la cantidad estimada de litros que pueden perderse transportando en el deposito número i la cantidad de miles de litros k.

2. Presentación del código:

Aquí podemos ver un ejemplo de programación dinámica para resolver este problema. El algoritmo se divide en dos partes:

2.1

Primero tenemos la parte donde se rellenarían la matrices de resultados. Tendríamos dos matrices, matriz M y matriz result, en la primera estarían los resultados de perdidas mínimas, las columnas serían los miles de litros y en las filas los depositos de 1 hasta dicha fila. En la segunda matriz las columnas son idénticas y a pesar de que la cantidad de filas cambia el significado es diferente. Las filas son el deposito ese en concreto y una posición de la matriz result significa, cuantos litros tenemos que meter en ese deposito.

```
public void Est(int[][] M, int[][] P, int N, int L) {

        for(int i= 0; i <= N; i++) {
            M[i][0] = 0;
        }
        for(int k = 1; k <= L; k++) {
            M[1][k] = P[1][k];
            result[1][k] = k;
        }

        for(int i=2; i <= N; i++)
             for(int k=1; k <= L; k++) {
                    M[i][k] = M[i-1][k];
                    int count = 0;
                    for(int x=1; x <= k; x++) {
                    if(M[i][k] >
```

```
                        (M[i-1][k-x] + P[i][x])) {
                        M[i][k] = M[i-1][k-x] + P[i][x];
                        count = x;
                    }
                }
                    if(count != 0)
                        result[i][k] = count;
            }
    }
```

(Imagen 2.1)

2.1.1 Variables de entrada:

→ int[][] M: Matriz vacía donde se guardaran los resultados de perdidas mínimas para cada deposito y litros.

→int[][] P: Matriz llena donde están los valores de entrada leídos desde el archivo en la cual indica las perdidas de meter x litros en el deposito y (P[x][y]).

→int N: Numero de depositos, numero de que hay en la matriz.

→ int L: Numero de litros a transportar.

(La matriz result es una variable global int[][], del mismo tamaño que M y se guardan los datos explicados anteriormente en el punto 2.1.)

2.1.2Explicación del código:

Este código se crea a partir de la recursion:

Estima(i; k) = mínimo{ Estima(i - 1; k); P(i;X) + Estima(i - 1; k – X) | 1<= X<= k }

si i > 1 y k > 0

Estima(i; 0) = 0 para todo i >= 1

Estima(1; k) = P(1; k) para todo k >= 1

Estima sería equivalente a la matriz M donde los datos quedan guardados. Primero el algoritmo comenzaría inicializando la matriz para la primera columna a 0 (para cada deposito el mínimo de perdida en 0 litros), el siguiente paso será inicializar la segunda fila, donde los valores del deposito 1 serán los mismos que los de la primera fila de la matriz P (con un solo tipo de deposito las perdidas mínimas serán las perdidas que tenga ese deposito).

Después de inicializar la matriz, pasamos a la recursion general. Recorremos la matriz

de resultados por filas, comenzando por la tercera fila. Para cada litro de la fila se calcula el mínimo entre la fila anterior en la misma columna (depositos anteriores mismos litros) o mínimo de todos los litros anteriores de la fila superior, sumándole para cada litro la perdida de introducir los litros restantes en el deposito en el que nos encontramos. Así sucesivamente para cada fila y columna a partir de los casos básicos que ya estarían rellenas las filas y columnas.

En el código podemos ver que la matriz resultado se va actualizando en los casos que se introduzcan litros en un deposito. Solo cuando se introducen litros en un deposito se le meten los litros introducidos en la matriz resultados para dicho deposito. Esta matriz también se inicializa con los casos base, donde la primera fila será idéntica a los valores de la primera fila de M.

2.2

En la matriz result quedaran guardados los valores de cuantos litros se añaden a que deposito, pero para todos los litros de manera que nosotros necesitamos conseguir los datos de la matriz result para los depositos con L litros. Para ello se ha implementado otra función donde se rescatan esos datos y se guardan en un array de enteros.

```
public int[] getResultado(int N, int L) {
      if(N == 0) {
            return solucion;
            } else if (result[N][L] == 0) {
                  solucion[N] = 0;
                  return getResultado(N - 1, L);
            } else {
                  int aux = result[N][L];
                  solucion[N] = aux;
                  return getResultado(N - 1, L - aux);
            }
   }
```

(Imagen 2.2)

2.2.1 Variables de entrada:

→int N: Numero de depositos, numero de que hay en la matriz.

→ int L: Numero de litros a transportar.

(El array solución es una variable global int[] donde se guardaran para cada deposito como indice del array los litros a introducir en el mismo).

2.2.2 Explicación:

Este es un algoritmo recursivo que recorre la matriz result y guarda los litros de cada deposito en un array. Primero comienza por los litros que se le asignan al ultimo deposito del total, si no le se asignara ninguno pasaría al deposito anterior. En caso de asignarle litros pasaríamos a los litros asignados al deposito anterior, pero ya no del total de los litros sino de los litros restantes.
Cuando se hayan recorrido todos los depositos (se llegue al primer deposito) se devolverá el array donde se almacenan los datos para cada deposito.

3 Estudio analítico.

En este apartado nos centraremos en el estudio de la función de coste del algoritmo. Para ello nos dividiremos en 3 partes; análisis de la inicializacion de los valores en la matriz, algoritmo principal donde se crean las matrices de resultados y por ultimo la función de adquisición de los datos a partir de las matrices con los resultados.

3.1 Inicializacion de las matrices:

```
public EstimacionLitros(int l, int n) {
	result = new int[n+1][l+1];
	solucion = new int[n+1];

	for(int h=0; h <= n; h++) {
		solucion[h] = 0;
		for(int j=0; j <=l; j++)
			result[h][j] = 0;
	}
    }
```

(Imagen 3.1)

Código de inicializacion donde a partir de los datos de entrada del tamaño de litros a transportar y la cantidad de depositos, se crean las matrices de las soluciones inicianizandolas a 0 en todos los valores.

3.1.1 Coste función:

$$f(n)=\sum_{i=0}^{n}\sum_{j=0}^{l}1\,;=n*l\,;=O(n*l)\,;$$

Esta es una función sencilla donde se recorren dos bucles anidados con una asignación.

3.2 Creación matrices solución:

Volviendo a la (Imagen 2.1), donde tenemos el código principal del algoritmo, el cual nos rellena las matrices con los resultados.

3.2.1 Coste función:

$f(n)=\sum_{i=2}^{N}(\sum_{k=1}^{L}(\sum_{x=1}^{k}3)+2);$ (Contando que para el caso ultimo "k" sería igual a L el sumatorio se nos iguala) → $f(n)=\sum_{i=2}^{N}\sum_{k=1}^{L}\sum_{x=1}^{L}3;=\sum_{i=2}^{N}\sum_{k=1}^{L}3\mathrm{L};=N*3\mathrm{L}^2=O(N*L^2)$

El orden de la función se quedaría función de los valores de N y L dependiendo de si N es mayor que el cuadrado de L.

3.2 Obtener soluciones de la matriz "result":

La función de coste de este algoritmo (Imagen 2.2) la calcularemos de forma recursiva debido a que se trata de un algoritmo recursivo.

3.21 Coste función:

El coste de este algoritmo es muy sencillo de calcular simplemente recorre desde el ultimo N hasta el primero, es decir, un coste lineal en N.

$f(n)=f(n-1);$ → esto seguirá hasta n = 0; $f(n)=\sum_{i=0}^{N}1;=N=O(N)$

3.3 Suma de los costes de cada función

Ahora sumaremos las 3 funciones para obtener el coste completo del algoritmo.

$f(n)=N+N*L+N*L^2;$ → De estos costes se escogerá el máximo y será el coste final del algoritmo. → $f(n)=O(N*L^2)$ → Coste del algoritmo para el problema de programación dinámica.

3.4 Coste en memoria del problema

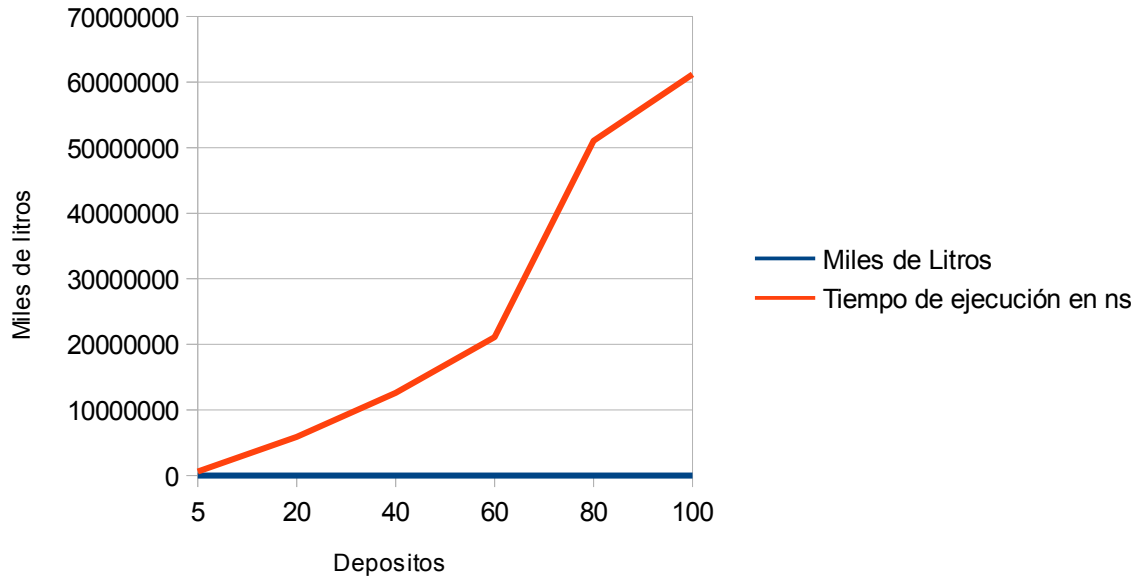
El coste del problema en memoria es sencillo de calcular. Simplemente se tendrán en cuenta el espacio que ocuparán las dos matrices "result" y "M" mas un simple array de enteros que se utiliza para mostrar los resultados. El coste en memoria vendría siendo de 2(N*L) → Por las dos matrices más el coste de un array de tamaño N.

4. Estudio experimental

Para la realización del estudio experimental se han realizado diferentes pruebas con diferentes cantidades de litros y de depositos, se comienza con pruebas pequeñas de 20 depositos y 100 litros y se termina con pruebas de 100 depositos y 500 miles de litros, ademas he decidido añadir una prueba de pocos datos menos de 10 miles de litros y depositos para mostrar en el gráfico la clara diferencia entre la prueba más pequeña y la prueba final de 100 depositos para 500 miles de litros.

	Depositos	Miles de Litros	Tiempo de ejecución en ns
	5	10	625736,75
	20	100	5921692,5
	40	200	12616895,75
	60	300	21123526
	80	400	51059844,75
	100	500	61182737

(Tabla 4.1)



(Imagen 4.2)

4.1 Comentario datos

En esta tabla tenemos los datos para cada prueba con los datos de cada deposito para cada miles de litros y la cantidad de nanosegundos que tarda el algoritmo en calculamos los litros que debemos depositar en cada deposito para conseguir las perdidas mínimas.

En el gráfico se puede apreciar la linea azul para depositos con miles de litros y para cada NxL (depositos x miles de litros) el tiempo que tarda en nanosegundos. Esta mostrado en nanosegundos, ya que, el tiempo en segundos no mostraría un gráfico adecuado en el cual se puedan diferenciar el tiempo de tardanza entre las diferentes cantidades de depositos y miles de litros.

5. Fuentes de información.

Aunque en este proyecto no ha habido muchas fuentes de información externas y el trabajo ha sido elaborado por mis ideas, se podría decir que para llegar a diseñar este algoritmo se ha realizado un proceso de aprendizaje utilizando diferentes fuentes.

(1) Algorithms: S. Dasgupta,C. H. Papadimitriou, and U. V. Vazirani.
(2) https://en.wikipedia.org/wiki/Dynamic_programming
(3) https://www.youtube.com/watch?v=OQ5jsbhAv_M&list=PLfMspJ0TLR5HRFu2kLh3U4mvStMO8QURm
→ Este ultimo enlacé contiene una serie de vídeos con problemas de programación dinámica solucionados y explicados.

6. Conclusiones.

Como conclusiones apelaría a subrayar la importancia de los casos de uso y su importancia. En mi caso particular puede que haya tenido algún retraso por no escoger casos de uso adecuados. En este problema en concreto tienen mucha importancia por que a pesar de que el algoritmo funcione con los casos de uso básicos, los cuales tu puedes resolver, tienes que darle un voto de confianza al algoritmo para casos de uso grandes que no son factibles de ser resueltos por ti mismo. Los casos de uso tiene su importancia en todo proyecto, pero en mi caso han sido esenciales a la hora de localizar problemas concretos.