Algorítmica Carlos Quesada Pérez

2º GII Miguel Ángel López Sánchez

Grupo B3 David Serrano Domínguez

**Práctica 4 - Programación Dinámica**

**Ejercicio 1**

**(1.1) Diseño de resolución por etapas y ecuación recurrente.**

Se ha tomado como representación del problema un grafo dirigido, donde cada idioma representa un vértice, y los distintos diccionarios que relacionan tuplas de idiomas son representados como aristas dirigidas de una unidad entre ese par de vértices (idiomas).

Visto de esta forma, el problema se puede resolver por etapas. De hecho, se adecúa perfectamente con el problema expuesto en clase de teoría del camino mínimo. En cada etapa se obtendría la traducción más óptima para poder alcanzar el idioma final con el menor número de traducciones posibles.

Sea D la matriz después de la k-ésima iteración. La ecuación recurrente puede expresarse como:

Dk(i,j) = Min { Dk-1(i,j), Dk-1(i,k) + Dk-1(k,j) }

**(1.2) Diseño de la memoria**

La memoria se ha diseñado como una matriz cuadrada D de N x N idiomas. En ella se almacena, para cada pareja de idiomas, el número mínimo de traducciones necesarias (valor infinito si no es posible) y la secuencia de traducciones. Es decir, la casilla D(i,j) almacenará el número de mínimo de traducciones a realizar partiendo desde i como idioma origen hasta llegar al idioma j destino, así como la secuencia de sucesivas traducciones para lograrlo {i, a, b, …, j}.

Es importante notar que los idiomas van indexados, por lo que la diagonal de la matriz es nula, ya que para traducir del idioma i al idioma i’ (siendo i=i’, el mismo idioma) no es necesario realizar ninguna traducción (por ejemplo: traducir del español al español, ¡no es necesario!).

**(1.3) Verificación del P.O.B.**

Supongamos que la traducción español-francés-inglés-italiano-griego es UNA (puesto que puede haber varias con el mismo número de traducciones, pero nunca menor) forma óptima de realizar la traducción desde español a griego (sólo 4 traducciones). Comenzando en español, en la primera etapa decidimos realizar la traducción a francés. Como resultado, ahora el estado del problema está definido por el vértice francés, y lo que se necesita es encontrar una(s) traducción(es) o camino mínimo/óptimo del francés al griego.

Entonces francés-inglés-italiano-griego debe constituir un camino mínimo entre el francés y el griego, porque si no; si, por ejemplo la secuencia de traducciones francés-alemán-ruso-italiano-griego fuera la más óptima entre el francés y el griego. Entonces español-francés-alemán-ruso-italiano-griego sería un camino del español al griego más corto que el camino inicial español-francés-inglés-italiano-griego (supuesto óptimo) y esto no es así. Como esto es una contradicción, se verifica el P.O.B.

**(1.4) Diseño del algoritmo de cálculo de coste óptimo.**

El algoritmo utilizado para la resolución del problema ha sido el algoritmo de Floyd Wharshall, el cual es el siguiente, donde D, P y L son todas matrices cuadradas de N x N. La matriz almacena la distancia mínima entre cada par de vértices. La matriz P almacena la secuencia de vértices del camino mínimo entre cada par de vértices. Y la matriz L es la matriz original de distancias del problema.

ALGORITMO FLOYD (D[1..n, 1..n], P[1..n,1..n], L[1..n,1..n])

begin

# Inicialización

For i = 1 to n do

For j = 1 to n

D[i,j] = L[i,j]

For i = 1 to n

D[i,i] = 0

# Cálculo de la memoria

For k = 1 to n do

For i = 1 to n do

For j = 1 to n do

If D[i,k] + D[k,j] < D[i,j] Then Begin

D[i,j] = D[i,k] + D[k,j]

P[i,j] = k

End

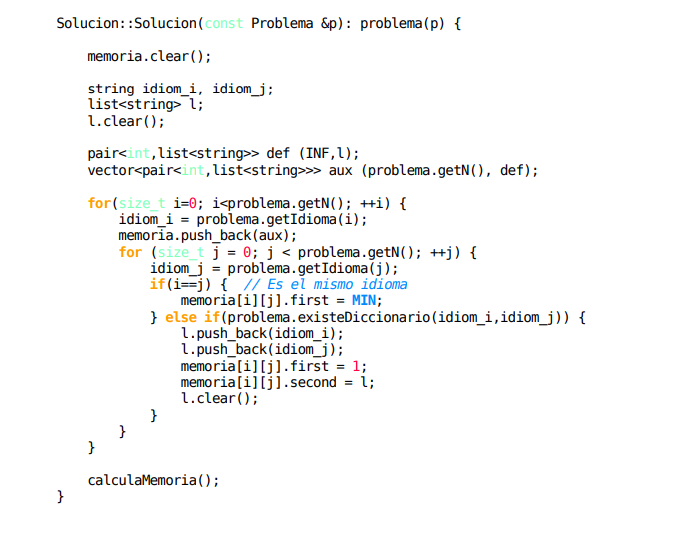
End

**(1.5) Diseño del algoritmo de recuperación de la solución.**

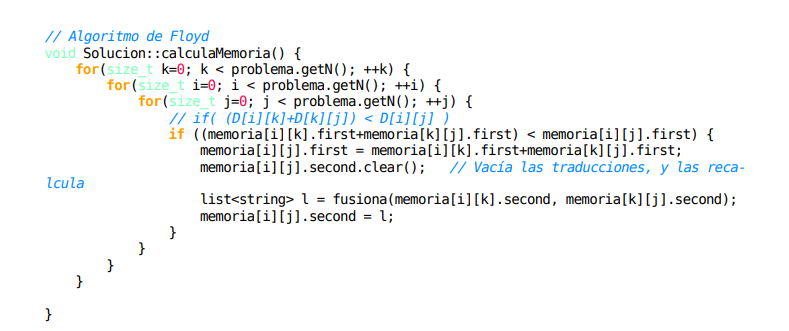
Para recuperar la solución no es necesario ningún algoritmo. La solución para cualquier par de idiomas se puede obtener accediendo a la matriz de memoria con los índices del idioma inicial i, y el idioma final j, es decir, acceder a la casilla D(i,j).

**(1.6) Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución.**

La implementación en código de los algoritmos se encuentra en la implementación de la clase Solucion (en el archivo solucion.cpp del ejercicio 1). En el constructor se encuentra la primera parte del algoritmo, la de inicialización y preparación de la memoria. La macro MIN equivale a 0.

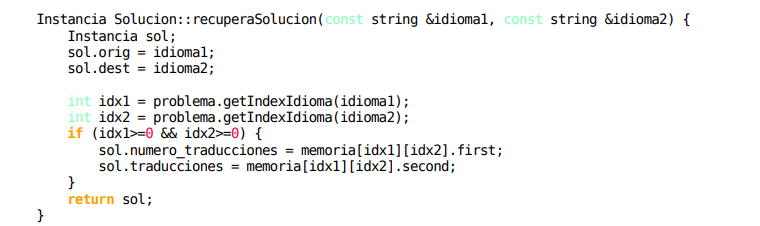


La segunda parte del algoritmo se encuentra en el método calculaMemoria() el cual es llamado también en el constructor de la clase, y cuya implementación es la siguiente. El método fusiona únicamente une dos listas de idiomas (secuencias de traducción).



Para la recuperación de la solución, se ha definido una estructura que guarde los datos específicos de una solución, puesto que el se desea tanto el número de traducciones mínimo cómo la secuencia de traducciones. También guarda el idioma inicial y el final, como información adicional.





**Ejercicio 2**

**(2.1) Diseño de resolución por etapas y ecuación recurrente.**

El problema es de naturaleza n-etápica por lo que se puede resolver por etapas. Como consiste en un tablero por posiciones, si se fija una de las dos posiciones (inicial o final), podemos considerar cada posición del tablero como un subproblema menor de tamaño menor al original. En cada casilla se podrá obtener el máximo número de bolsas cogidas hasta llegar a ese punto.

La ecuación en recurrencias está determinada por el movimiento por el que se pudo llegar a la posición (i,j) considerada. Como los únicos movimientos permitidos son Izquierda, Diagonal Inferior Izquierda y Abajo, las posiciones a considerar son justamente las opuestas (Derecha, Diagonal Superior Derecha y Arriba). También se considera el tipo de casilla de la posición (si es bolsa de oro o pared) mediante la función P.

-1 si es muro

P(i,j) = 0 si es casilla normal

1 si es bolsa de oro

Caso Base: M(1,m) = P(1,m)

Recurrencia:

si P(i,j)<0 entonces M(i,j)=-1

M(i,j) = en otro caso

M(i,j)=P(i,j) + max( M(i-1,j), M(i-1,j+1), M(i,j+1) )

**(2.2) Diseño de la memoria**

La matriz se ha diseñado con el mismo tamaño que el tablero, es decir N x M. Había dos posibilidades de crear o interpretar la memoria, dependiendo del enfoque utilizado para resolver el problema. En nuestro caso, hemos optado por el enfoque adelantado, es decir, partiendo del inicio, llegar hasta el fin. Por tanto, para cada casilla de nuestra memoria M(i,j), se almacena el número máximo de bolsas de oro cogidas desde el origen (esquina superior derecha, hasta la posición (i,j) incluida, con los movimientos permitidos.

**(2.3) Verificación del P.O.B.**

Gracias a la restricción de movimientos, es posible verificar el Principio del Óptimo de Bellman, que enuncia lo siguiente: “Una política óptima sólo puede estar formada por subpolíticas óptimas”. En nuestro problema no nos importa el camino, únicamente nos interesa el número de bolsas de oro cogidas, por lo que para considerar el número de bolsas de una casilla (i,j) sólo tendremos que tener en cuenta las de sus casillas inmediatamente precedentes max {(i-1,j), (i-1,j+1), (i,j+1)}. Todas estas casillas a su vez han considerado lo mismo para su obtención. Esto verifica positivamente el P.O.B.

**(2.4) Diseño del algoritmo de cálculo de coste óptimo.**

El algoritmo de cálculo de coste óptimo para este problema comparte alguna idea con el algoritmo de Dijkstra, pero aprovechando el enfoque tabular de la PD. Se basa fundamentalmente en la ecuación recurrente descrita en el apartado 2.1 de esta práctica.

ALGORITMO (M[1..n,1..m], P[1..n,1..m])

begin

# Inicialización: caso base (posición inicial)

M[1,m] = P[1,m]

# Inicialización: primera fila

For j = m-1 to 1 do

M[1,j] = P[1,j] + M[1,j+1]

# Inicialización: última columna

For i = 2 to n do

M[i,m] = P[i,m] + M[i-1,m]

# Cálculo de la memoria / matriz

For i=2 to n do

For j=m-1 to 1 do

If P[i,j]<0 Then M[i,j] = -1

Else M[i,j] = P[i,j] + max( M[i-1,j], M[i-1,j+1], M[i,j+1] )

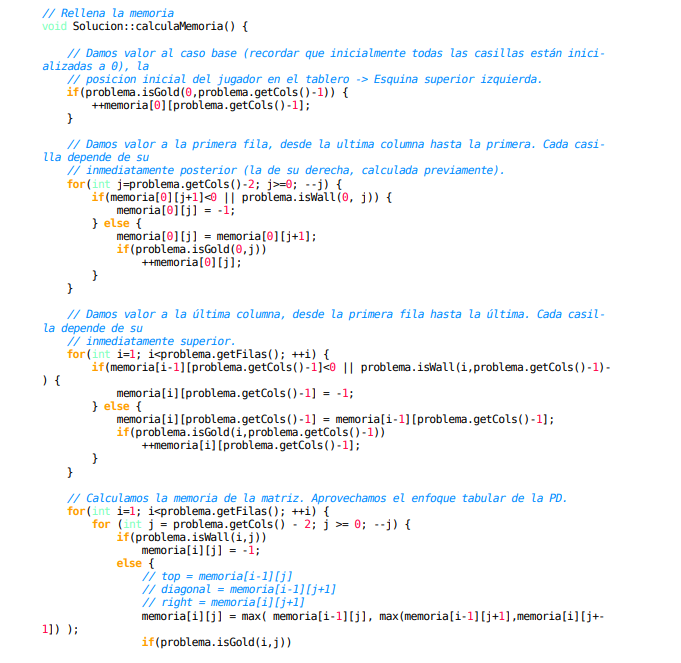
end

**(2.5) Diseño del algoritmo de recuperación de la solución.**

Para recuperar la solución no es necesario ningún algoritmo. Se obtiene accediendo a la posición de la memoria correspondiente con la salida del tablero. Esto es: solución = M[n,1], obteniendo así el máximo número de bolsas de oro obtenidas desde el inicio hasta la solución.

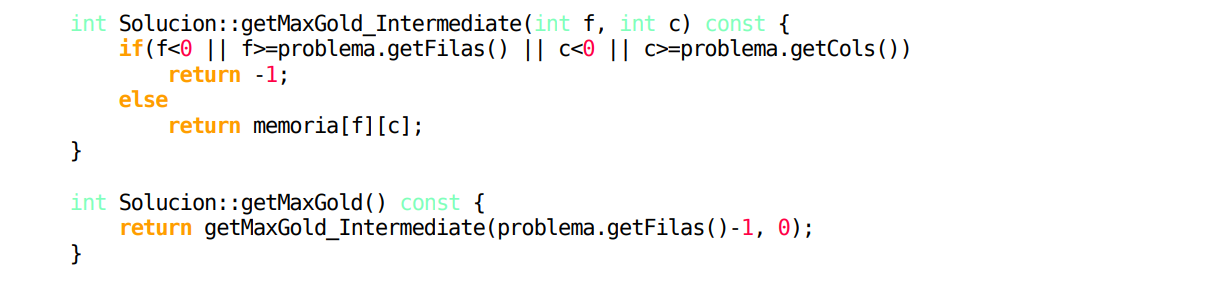
**(2.6) Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución.**

La implementación en código de los algoritmos se encuentra en la implementación de la clase Solucion (en el archivo solucion.cpp del ejercicio 2), principalmente en el método calculaMemoria(), el cual es llamado en el constructor de la clase.





Para la recuperación de la solución, se debe invocar al método de la clase solución getMaxGold().



**Ejercicio 3**

**(3.1) Diseño de resolución por etapas y ecuación recurrente.**

Este ejercicio es similar al anterior, la diferencia es que en este problema queremos hacer uso de la menor batería posible del personaje hasta un objetivo. La ecuación de recurrencias está determinada por el movimiento por el que se pudo llegar a la posición (i,j) considerada, en el cual, cada casilla tiene un valor distinto: Amarilla valen 2 de batería, Rojas valen 0 de batería y Negras valen 1 de batería. Dicha ecuación de recurrencia está definida por tres movimiento que poder realizar que son son Izquierda, Diagonal Inferior Izquierda y Abajo, y todo dependerá del costo de avance a cada casilla.

Consideramos el tipo de casilla de la posición mediante la función B.



0 si es ROJO

B(i,j) = 1 si es NEGRO

2 si es AMARILLO

Caso Base: M(m,1) = B(m,1)

Recurrencia: M(i,j)=B(i,j) + min( M(i,j-1), M(i+1,j-1), M(i+1,j) )

**(3.2) Diseño de la memoria.**

La matriz la hemos diseñado con un tamaño N x M, el cual almacenará en cada casilla el gasto que consumirá de antemano desde el inicio hasta el final. Por tanto por cada casilla M(i,j) se almacenará el valor consumido anteriormente hasta esa casilla más su coste de avance en su casilla, teniendo en cuenta el coste específico de cada tipo de casilla.

**(3.3) Verificación del P.O.B.**

Tras tener una restricción de movimientos(arriba, arriba-der,derecha) podemos afirmar que se cumple el Principio del Óptimo de Bellman. Lo que tenemos como objetivo en nuestro problema es el camino que toma el personaje en cada momento ya que elegirá el más óptimo así que lo único que tendremos que tener en cuenta en cada casilla es el minimo de bateria que se usa al avanzar( min( M(i,j-1), M(i+1,j-1), M(i+1,j) ) por lo que esto verifica el P.O.B

**(3.4) Diseño del algoritmo de cálculo de coste óptimo.**

Nuestro algoritmo se basa en inicializar la matriz primero la primera fila para saber su coste ya que es simplemente empezando desde abajo sumarle el valor de la casilla anterior. Y hacemos lo mismo con la última fila, ya que es donde empezaremos(m,1).

Ahora, calculamos el valor de lo que cuesta cada casilla en bloques de 4 ya que los movimientos son limitados y sabemos lo que vale cada casilla.

Una vez terminado de rellenar la matriz, calculamos el camino menos costoso.

CalcularMemoria (M[0..n-1,0..m-1], B[0..n-1,0..m-1])

begin

// CASO BASE

M[m,1] = P[m,1]

# Inicialización: primera fila

bateriaGastada = 0

For j = m-2 to 0 do

M[j,0] += bateriaGastada

bateriaGastada = M[j,0]

# Inicialización: primera columna

bateriaGastada = 0

For j = 1 to n do

M[filas-1,j] += bateriaGastada;

bateriaGastada = M[filas-1,j]

# Cálculo de la matriz ‘M’

For i=n-2 to 0 do

For j=1 to m do

M[i,j] += min(M[i,j-1],M[i+1,j-1],M[i+1,j])

#Guardamos el coste final que necesitamos para llegar al objetivo

bateriaGastada =memoria[0,M-1]

end

**(3.5) Diseño del algoritmo de recuperación de la solución.**

No hemos usado ningún algoritmo de recuperación de la solución sino que lo hemos guardado en la posición de la matriz donde está el objetivo y lo hemos guardado en una variable bateriaGastada para saber la solución

**(3.6) Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución.**

****

La recuperación de la solución se indica al final del algoritmo.