MEMORIA PRÁCTICA 5: PROGRAMACIÓN DINÁMICA

David Bacas Posadas
Julián Carrión Tovar
Manuel Marín Rodríguez
Alicia Ruiz Gómez
Raúl Martínez Bustos

UNIVERSIDAD DE GRANADA

Índice

Objetivos

El objetivo de la práctica consiste en que el alumno sea capaz de analizar un problema y resolverlo
mediante la técnica de Programación Dinámica, siendo capaz de justificar su eficacia en términos de
optimalidad. Se expondrá un problema que será resuelto en clase por el profesor. Posteriormente, se
expone un conjunto de problemas que deberán ser resueltos por el estudiante

Río y aldeas(P.1)

A lo largo de un río hay n aldeas. En cada aldea, se puede alquilar una canoa para viajar a otras aldeas que estén a favor de la corriente (resulta casi imposible remar a contra corriente). Para todo posible punto de partida i y para todo posible punto de llegada j más abajo en el río ($i < j$), se conoce el coste de alquilar una canoa para ir desde i hasta j, $c(i, j)$ (si ese trayecto concreto no existe, entonces $c(i, j) = \infty$. Sin embargo, puede ocurrir que el coste del alquiler desde i hasta j sea mayor que el coste total de una serie de alquileres más breves. En tal caso, se puede devolver la primera canoa en alguna aldea k situada entre i y j y seguir el camino alquilando en k una nueva canoa (no hay costes adicionales por cambiar de canoa de esta manera).

Diseñe un algoritmo basado en programación dinámica para determinar el coste mínimo del viaje en canoa desde todos los puntos posibles de partida i a todos los posibles puntos de llegada j ($i < j$). Aplique dicho algoritmo en la resolución del caso cuya matriz de costos es la siguiente:

	1	2	3	4	5
1	0	3	3	∞	∞
2	-	0	4	7	∞
3	-	-	0	2	3
4	-	-	-	0	2
5	-	-	-	-	0

Diseño de resolución por etapas y ecuación recurrente:

Diseño de resolución por etapas:

Tenemos un río y n aldeas alineadas a lo largo del río. Queremos viajar de una aldea a otra en canoa, pero queremos hacerlo de manera que minimicemos el coste total del alquiler de estas mismas . Este coste de alquiler de una canoa va a venir dado por la matriz de costes $C[i][j]$, donde i y j son índices que identifican las aldeas a lo largo del río. Aquí, i y j cumplen con la condición $i<j$ ya que solo se puede viajar corriente abajo.

Dividimos el problema en etapas, donde cada etapa considera la minimización del coste para cada subsegmento del río.

Cada etapa evalúa la posibilidad de devolver la canoa en una aldea intermedia k y alquilar una nueva para continuar el viaje.

Ecuación recurrente:

Sea $dp[i][j]$ el coste mínimo de viajar desde la aldea en la posición i a la j (inclusive). La ecuación recurrente se define como:

$$dp[i][j] = min_{i<k<j}(dp[i][k] + dp[k][j])$$

con la condición inicial:

$$dp[i][i] = 0$$

y:

$$dp[i][j] = C[i][j] \ si \ \ j = i + 1$$

Esta ecuación considera que para encontrar el coste mínimo de i a j, debemos considerar las posibles aldeas intermedias k donde $i < k < j$.

Valor objetivo:

Entendemos por valor objetivo el valor final que buscamos minimizar, en este ejemplo nos referimos al coste del alquiler de las canoas.

Es fácil ver que, en este caso, el valor objetivo se obtiene calculando el mínimo coste entre cada par de aldeas i,j . Por lo que de entre todos los subproblemas posibles habremos buscado la solución óptima gracias a combinarlos. Este valor lo vamos a almacenar en la matriz dp (dynamic programming) , donde dp[i][j] representa el valor objetivo del subproblema correspondiente al coste mínimo de viajar de la aldea i a la j.

Diseño de la memoria:

En este problema, el diseño de la memoria es primordial para almacenar los resultados intermedios y permitir una solución eficiente. Procedemos a explicar cómo se diseña y utiliza la memoria para resolver el problema de las canoas:

Estructura de la memoria

Se utiliza la matriz bidimensional dp de dimensiones nxn para almacenar los valores objetivos obtenidos anteriormente de viajar entre las dos aldeas.

Hemos elegido hacerlo con una matriz bidimensional ya que nos va a permitir ilustrar perfectamente este problema, ya que como es natural, cada celda dp[i][j],

contiene el coste mínimo del alquiler de canoas entre estas localizaciones, puesto que las filas (i) representan la aldea de partida y las columnas (j) la aldea de llegada.

Nosotros hemos creído conveniente implementar una función que inicialice la matriz dp, de manera que dp[i][i] se inicializa a 0 , ya que si estamos en esa aldea, la distancia de viajar a ella misma es 0.

dp[i][j] se inicializa con el coste directo C[i][j], si este viaje directo es posible, en caso de que el camino sea irrealizable, se inicializa con el valor infinito (∞).

Para realizar la transición entre etapas, para cada par de aldeas (i,j) en el cual i < j, calculamos el coste mínimo teniendo en cuenta las aldeas intermedias k (donde i < k <j).

Al completar la etapa para la última aldea, ya habremos determinado el coste mínimo para viajar entre todas las aldeas i y j.

Verificación del P.O.B.

Para verificar el Principio de Optimalidad de Bellman (P.O.B), debemos demostrar que el coste mínimo para viajar en canoa entre las aldeas i y j se puede descomponer en los costes mínimos de viajes entre aldeas intermedias.
Podemos asegurar que el P.O.B se verifica ya que:

- Definimos en subproblemas:
    - Consideramos pares de aldeas i y j en los que se cumple i< j .
    - Queremos encontrar el coste mínimo para viajar entre ambas.
- Descomposición:
    - Descomponemos el problema en subproblemas tomando en cuenta las aldeas intermedias a las cuales denominamos con la letra k , las cuales se explican como: i < k <j.
    - La solución del problema original es la suma de los costes de los subproblemas, tal como explica la ecuación de recurrencia

$$dp[i][j] = min_{i<k<j}(dp[i][k] + dp[k][j])$$

- Certificación de la optimalidad:

  Nos aseguramos de que la optimalidad se cumple ya que por definición de programación dinámica dp[i][k], es el coste mínimo de viajar de i a k y dp[k][j], es el coste mínimo de viajar de k a j, por lo que dp[i][k] + dp[k][j] debe de ser el coste mínimo de viajar i a j pasando por k, siempre y cuando el viaje directo no abarate el coste.

Diseño del algoritmo de cálculo de coste óptimo:

```
1  Función inicializarDP(dp, C, n){
2      for i desde 0 hasta n - 1{
3          for j desde i + 1 hasta n - 1
4              dp[i][j] = C[i][j]
5      }
6  }
7
8  Función calcularCosteMinimo(dp, n){
9      for longitud desde 2 hasta n{
10         for i desde 0 hasta n - longitud{
11             j = i + longitud - 1
12             for k desde i + 1 hasta j - 1{
13                 dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j])
14             }
15         }
16     }
17 }
```

En primer lugar tenemos la ya mencionada función para inicializar la matriz, en la cuál se crea una matriz *dp* de tamaño *nxn* donde se almacenan los costes mínimos del viaje entre aldeas. Esta función llena la matriz con los costes directos que ya se le han proporcionado en la matriz *C*. Para cada par de aldeas *(i,j)*, se inicializa con el valor *C[i][j]*.

Ahora pasamos a la función calcularCosteMinimo, la cual utiliza programación dinámica para calcular el coste mínimo de viaje entre todas las aldeas.Para cada longitud posible de subproblemas (desde 2 hasta n), se iteran los posibles pares de aldeas (i, j).Para cada par (i, j), se consideran todas las posibles aldeas intermedias k (donde i < k < j).

La entrada dp[i][j] se actualiza con el mínimo entre el valor actual dp[i][j] y la suma de los costes mínimos de dp[i][k] y dp[k][j].

Diseño del algoritmo de recuperación de la solución

```
Función recuperarSolucion(dp, i, j){
    if i == j{
        return lista vacía
    }

    for k desde i + 1 hasta j - 1{
        if dp[i][j] == dp[i][k] + dp[k][j]{
            izquierda = recuperarSolucion(dp, i, k)
            derecha = recuperarSolucion(dp, k, j)
            izquierda.agregar(k)
            izquierda.agregar(derecha)
            return izquierda
        }
    }

    return lista vacía
}
```

La función recuperarSolucion se utiliza para reconstruir el camino óptimo a partir de la matriz dp que contiene los costes mínimos.
Si i es igual a j, se retorna una lista vacía porque no hay viaje entre la misma aldea.
Para cada par (i, j), se buscan todas las posibles aldeas intermedias k que dividen el viaje de i a j en dos subproblemas.
Si la suma de dp[i][k] y dp[k][j] es igual a dp[i][j], esto indica que k es una parte del camino óptimo.
Se llama recursivamente a recuperarSolucion para los subproblemas (i, k) y (k, j), y se combinan los resultados para formar el camino completo.
Se agregan las aldeas intermedias a la lista del camino y se retorna la lista combinada.

Implementación(algoritmos cálculo coste óptimo y recuperación solución):

```cpp
#include <iostream>
#include <vector>
#include <climits>
#include <algorithm>

using namespace std;

const int INF = INT_MAX;

// Función auxiliar para inicializar la matriz dp
void inicializarDP(vector<vector<int>>& dp, const vector<vector<int>>& C, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            dp[i][j] = C[i][j];
        }
    }
}

// Calculamos el coste mínimo utilizando programación dinámica
void calcularCosteMinimo(vector<vector<int>>& dp, int n) {
    for (int longitud = 2; longitud <= n; ++longitud) {
        for (int i = 0; i <= n - longitud; ++i) {
            int j = i + longitud - 1;
            for (int k = i + 1; k < j; ++k) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
            }
        }
    }
}
```

```cpp
// Recupera el camino óptimo utilizando la matriz dp
vector<int> recuperarSolucion(const vector<vector<int>>& dp, int i, int j) {
    if (i == j) return {};
    for (int k = i + 1; k < j; ++k) {
        if (dp[i][j] == dp[i][k] + dp[k][j]) {
            vector<int> izquierda = recuperarSolucion(dp, i, k);
            vector<int> derecha = recuperarSolucion(dp, k, j);
            izquierda.push_back(k);
            izquierda.insert(izquierda.end(), derecha.begin(), derecha.end());
            vector<int> izquierda = recuperarSolucion(dp, i, k)
        }
    }
    return {};
}

int main() {
    int n = 5;
    vector<vector<int>> C = {
            {0, 3, 3, INF, INF},
            {INF, 0, 4, 7, INF},
            {INF, INF, 0, 2, 3},
            {INF, INF, INF, 0, 2},
            {INF, INF, INF, INF, 0}
    };

    vector<vector<int>> dp(n, vector<int>(n, INF));
    inicializarDP(dp, C, n);
    calcularCosteMinimo(dp, n);

    int inicio = 0, fin = 4;
    cout << "Coste mínimo de " << inicio + 1 << " a " << fin + 1 << ": " << dp[inicio][fin] << endl;

    vector<int> camino = recuperarSolucion(dp, inicio, fin);
    cout << "Camino óptimo: " << inicio + 1 << " ";
    for (int i = 0; i < camino.size(); ++i) {
        cout << camino[i] + 1 << " ";
    }

    cout << fin + 1 << endl;

    return 0;
}
```

Esta es nuestra implementación de las tres funciones anteriores en c++. Para probar su correcto funcionamiento, hemos implementado un ejemplo concreto en el que le pasamos una matriz C con los siguientes costes:

{0, 3, 3, INF, INF},
{INF, 0, 4, 7, INF},
{INF, INF, 0, 2, 3},
{INF, INF, INF, 0, 2},
{INF, INF, INF, INF, 0}

Podemos ver que nuestro algoritmo de programación dinámica funciona puesto que al ejecutarlo, obtenemos los siguientes resultados:

```
/home/ali/CLionProjects/rios_aldeas/cmake-build-debug/rios_aldeas
Coste mínimo de 1 a 5: 6
Camino óptimo: 1 3 5

Process finished with exit code 0
```

Por lo que el coste mínimo de alquiler será 6 y para llegar de la aldea 1 a la 5 debemos parar en la aldea intermedia número 3.

Viajes aéreos (P.2)

Una empresa realiza planificaciones de viajes aéreos entre n ciudades. Para ir de una ciudad i a una ciudad j puede ser necesario coger varios vuelos distintos. El tiempo de un vuelo directo de i a j será (si existe) $T(i,j)$ (que puede ser distinto de $T(j,i)$). Hay que tener en cuenta que si cogemos un vuelo de i a k y después otro de k a j, será necesario esperar un tiempo de *escala* adicional $E(k)$ en el aeropuerto de k, con lo que el tiempo de ese viaje de i a j será de $T(i,k)+T(k,j)+E(k)$.

Se desea conocer la forma de volar desde cualquier ciudad i hasta cualquier otra j en el menor tiempo posible. Diseñad e implementad un algoritmo de Programación dinámica que resuelva este problema. Aplicadlo para resolver el problema, para $n=4$, cuya matriz de tiempos es la siguiente, suponiendo que $E(k)=1\ \forall k$.

$T[i,j]$	1	2	3	4
1	0	2	1	3
2	7	0	9	2
3	2	2	0	1
4	3	4	8	0

Tras haber leído el enunciado del ejercicio propuesto, podemos destacar un par de apuntes importantes para entender nuestro problema:

- Cuando la empresa planea ir de la ciudad i a la j, esta puede parar entre n ciudades. Nuestro objetivo será ver cuál de entre todos los viajes posibles (con o sin escala) conseguimos minimizar lo máximo posible nuestro tiempo de llegada de i hasta j, y esa será la solución óptima a nuestro problema.

- Como se ha mencionado anteriormente, no será lo mismo el tiempo de viaje $T(i,j)$ que el tiempo de viaje $T(j,i)$, por lo tanto la matriz no será simétrica como en otros problemas que ya habíamos planteado en la asignatura. La matriz presentada para nuestro problema la llamaremos matriz de viajes. En ella notaremos como $T(i,j)$ el tiempo tardado en ir desde una ciudad i hasta otra ciudad j, y como i o j las n ciudades que tendremos disponibles como opción para viajar a ellas. Vease la tabla:

$T(i,j)$	j_0	j_1	…	j_{n-2}	j_{n-1}
i_0	$T(i_0,j_0)$	$T(i_0,j_1)$	…	$T(i_0,j_{n-2})$	$T(i_0,j_{n-1})$
i_1	$T(i_1,j_0)$	$T(i_1,j_1)$	…	$T(i_1,j_{n-2})$	$T(i_1,j_{n-1})$
…	…	…	…	…	…
i_{n-2}	$T(i_{n-2},j_0)$	$T(i_{n-2},j_1)$	…	$T(i_{n-2},j_{n-2})$	$T(i_{n-2},j_{n-1})$
i_{n-1}	$T(i_{n-1},j_0)$	$T(i_{n-1},j_1)$	…	$T(i_{n-1},j_{n-2})$	$T(i_{n-1},j_{n-1})$

Como podemos comprobar, hemos empezado el orden desde 0 hasta n-1, en vez desde 1 hasta n, para darle una mayor coherencia y similitud con la versión que implementaremos y diseñaremos después, ya que utilizaremos diferentes vectores y matrices (los cuales van desde un rango 0 a n-1).

- Por último, cabe destacar que la diagonal de la matriz está llena de ceros, pues 0 es el tiempo que tardamos de viajar de una ciudad i a si misma

$$T(i,i)\,,\ T(j,j) = 0$$

Esto es así ya que si estamos en una ciudad i, no podemos viajar a esta porque ya nos encontramos ahí.

Aclarado esto, estudiemos la naturaleza de nuestro algoritmo de memoria dinámica y la posible solución a nuestro problema.

IMPORTANTE: si miramos las transparencias de teoría, ya se puede ver de por sí que este problema se encamina bastante a lo que podemos ver en las diapositivas 28 y 29, pues parece estar relacionado con el algoritmo de Floyd

---

Diseño de resolución por etapas y ecuación recurrente

Diseño de resolución por etapas

El problema planteado se puede (recomendable para nuestro enfoque) resolver por etapas. Primero, tendríamos que ver el tiempo que se tarda de viaje de las ciudades planteadas i e j ($T(i,j)$). Una vez calculado, vamos probando con cada una de las ciudades que hay disponibles en nuestra matriz para ver si hay alguna de estas que minimice todo lo posible el tiempo del viaje. De esta manera, conseguimos guardar la combinación con escala que menor tiempo de viaje nos de. Finalmente, vemos si las combinación con escala mínima es menor o mayor que el viaje directo, y nos quedamos con ese tiempo mínimo posible.

Por lo tanto, en cada etapa (tras haber calculado tiempo de viaje directo $T(i,j)$, nos encargamos de ir probando una a una todas las combinaciones con escala hasta encontrar el mínimo posible, y guardamos este.

Ecuación recurrente

Una vez especificado nuestro diseño de resolución por etapas, estudiamos qué método sería el más adecuado para poder resolver este problema.
En este caso, la idea principal que se nos viene a la cabeza sería el algoritmo de Floyd (visto en teoría) que suele servir para encontrar el camino más corto entre una pareja de nodos dentro de un grafo. Cogiendo esta idea, implementaremos un

diseño en el que los tiempos de escala adicionales que haya en cada "nodo intermedio" se tengan en cuenta para la posible solución de tiempo final.

Planteado ya el uso de este algoritmo, procedemos entonces a plantear la ecuación recurrente que usamos en este algoritmo. Sabiendo que MV[ ] será la fórmula para obtener nuestro mínimo viaje:

$$MV[i,j] \;=\; min\{MV[i,j]\,,\; MV[i,k] \;+\; MV[k,j] \;+\; E(k)\}$$

Como podemos observar, nuestra fórmula de mínimo tiempo de viaje dependerá fundamentalmente de una de estas dos posibilidades:

- El mínimo tiempo posible se realiza con el **viaje directo** de i a j ($T(i,j)$. (Descartamos escalas)
- El mínimo tiempo posible se realice con una de las múltiples **escalas** posibles propuestas por las n ciudades, donde el mínimo entre i e j será $MV[i,k] \;+\; MV[k,j] \;+\; E(k)$.

Tratando esta fórmula aplicada al algoritmo, vemos que los posibles casos base son los dos que hemos comentado en la fórmula del mínimo y el de las dos mismas ciudades:

1. **Tiempo de viaje de una ciudad a sí misma (ya mencionado):**

   El tiempo de viaje de una ciudad a sí misma es 0, ya que no podemos viajar de una ciudad a si misma.

   $$(c_1 = \; c_2 = i\,) \Rightarrow MV[c_1, c_2] = 0$$

   El tiempo mínimo de viaje entre 2 mismas ciudades, es 0.

2. **Tiempo de viaje directo:**

   Si existe un vuelo directo de la ciudad i a la ciudad j, el tiempo mínimo de viaje inicial es el tiempo del vuelo directo $i\,T(i,j)$.

   Para cualquier par de ciudades i y j existirá un vuelo directo:

   $$MV[i,j] \;=\; T(i,j)$$

3.Tiempo de viaje con escalas:

En caso de no existir vuelo directo, se podrá obtener un vuelo con escalas entre un par de ciudades i y j. Para cualquier par de ciudades i e j que no exista un vuelo directo tenemos que el tiempo de vuelo será,teniendo en cuenta la pertinente escala:

$$T(i,j) = T(i,k) + T(k,j) + E(k)$$

Por lo tanto, nuestra fórmula del mínimo tiempo de viaje se ve fundamentalmente afectada por el tiempo de viaje directo, y por cada uno de los tiempos de viajes con las posibles escalas, y no tendremos mucho caso base donde se viaje de una ciudad i a si misma (i).

Valor objetivo

El valor objetivo en este problema es el tiempo mínimo de viaje entre todas las posibles parejas de ciudades i y j. Este valor se calcula mediante la técnica de programación dinámica, específicamente utilizando una versión modificada del algoritmo de Floyd que incluye el tiempo de escala(k) en las ciudades intermedias propuestas en nuestra matriz de ejemplo.

Diseño de la memoria

Resuelto el anterior punto, pasamos a realizar el diseño de la memoria que va a permitirnos explicar cómo rellenaremos nuestra memoria con los diferentes datos adquiridos a lo largo del problema. El esquema será:

- Para poder resolver este problema, nosotros vamos a hacer uso de dos matrices diferentes:

    1.**MV**[ ]: (matriz guarda tiempos mínimos viajes)Esta matriz tendrá n filas y m columnas. En esta, las filas serán la ciudad i y la columnas la ciudad j, siendo así MV[i,j] el tiempo mínimo de viaje que hay entre las ciudades i y j. Por lo tanto, en esta matriz guardaremos los valores mínimos de viaje entre los pares de ciudades (tengan escala o sean directos).

2.**prox_nodo**[ ]**:** (matriz reconstrucción soluciones, la usaremos finalmente para reconstruir el camino más corto o solución óptima global) Así mismo, esta matriz también tendrá n filas y m columnas. En esta, lo que se hará es guardar el siguiente nodo en el camino más corto para el viaje entre las ciudades i y j que se le introduzca (**prox_nodo**[i, j]) , diferente de la anterior que lo que guardaba eran resultados de mínimos tiempos de viaje en memoria.

Por lo tanto, cada celda de la matriz **MV**[ ] contendrá el mínimo tiempo posible de viaje entre las ciudades i (filas) y j (columnas). En cambio, **prox_nodo**[ ] nos servirá como apoyo para ver cual es el siguiente nodo a escoger en el camino más corto de nuestro viaje de i a j, intentando así minimizar el tiempo de viaje posible.

En conclusión, **prox_nodo**[ ] será una matriz auxiliar que nos servirá de ayuda para recorrer nuestro grado en el problema, mientras que **MV**[ ] será donde guardaremos las soluciones de viaje entre las ciudades.

Introducidas ambas, ahora explicaremos como se iría llenando cada una:

· En el caso de MV[ ] (**memorización**):

- Iríamos usando nuestra fórmula del mínimo basado en el algoritmo de Floyd, e iríamos probando con el viaje directo (si existe) en las ciudades i y j, y después con todas las posibles escalas, y veríamos de las escalas el menor valor de estas. Finalmente, si el menor tiempo es el de viaje directo dejaríamos este tal cual en la matriz MV[ ], y en caso contrario si encontramos el mínimo global en un viaje con escala, actualizaremos el valor de esa celda en MV[ ] con el mínimo valor de tiempo que hayamos obtenido con esa escala .
- De esta manera, decidimos cual sería el mínimo valor posible de todos para rellenar esa celda de la fila i y j de nuestra matriz MV[ ].
- De esta forma, iríamos repitiendo este proceso una y otra vez para cada todos los casos de viajes entre ciudades de la matriz, hasta completar todas las celdas de esta.

·En el caso de prox_nodo[ ] (**reconstrucción solución**):

Una vez hayamos obtenido el tiempo de vuelo directo $T(i, j)$, procedemos a iterar sobre cada una de las posibles escalas k, para ver si alguna de estas puede minimizar el valor aún más:

- Iteramos sobre cada posible ciudad intermedia k, y actualizamos las matrices DP[ ] (si se alcanza nuevo valor mínimo actualizamos) y prox_nodo[ ].
- Si el tiempo de viaje que obtenemos a través de k es menor que el tiempo actual que tenemos en **MV**[i, j], actualizamos el valor de dicha posición con este nuevo mínimo, y se pone en prox_nodo[i, j] el siguiente nodo correspondiente a visitar.

Verificación del P.O.B.

Una vez introducido el diseño de memoria, procedemos a hablar de la verificación del P.O.B. Como bien sabemos, la verificación del Principio de Optimalidad de Bellman verifica si toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas. Por lo tanto, cualquier solución óptima, forzosamente; debe estar formada por subsoluciones óptimas.

Aplicado al contexto de este problema, se establece que cualquier camino óptimo entre dos ciudades i y j debe consistir en caminos óptimos entre ciudades intermedias. El principio de optimalidad se verifica este caso pues, como podemos ver, si el camino óptimo pasa por k, entonces la suma de su tiempo óptimo será $T(i, k) + T(k, j) + E(k)$.

Al ver esa fórmula comprobamos que nuestra solución óptima realmente pasa por una ciudad intermedia k, y que esta está compuesta por dos soluciones óptimas de subproblemas, que serán por lo tanto el viaje de i a k, y de k a j.

Nuestro principio de optimalidad se cumple debido a que nuestra ecuación recurrente basada en el algoritmo de Floyd-Warshall modificado se basa en este principio, permitiendo que el tiempo de viaje mínimo y los caminos se actualicen

de manera óptima a medida que se consideran todas las posibles ciudades intermedias (k) .

Para demostrar lo comentado aquí arriba, vamos a servirnos de un ejemplo aislado (con datos diferentes a nuestra matriz) que demuestre lo anteriormente expuesto:

Imaginemos que queremos viajar desde una ciudad 1 a una ciudad 4 utilizando la ecuación de recurrencia y el algoritmo de Floyd que hemos adaptado para este problema, teniendo en cuenta también los tiempos de escala. Si 1 y 4 (i y j) son las que vamos a usar para el viaje, 2 y 3 serán entonces las ciudades intermedias (k).

-$T(1, 4)$: La ruta directa tiene un tiempo de viaje de 6 horas.

-Ruta pasando por la ciudad **2**:

- Ruta de 1 a 2: 3 horas.
- Escala en 2: 1 hora.
- Ruta de 2 a 4: 2 horas.

En total, esto nos da un tiempo de viaje de 6 horas.

-Ruta pasando por la ciudad **3**:

- Ruta de 1 a 3: 5 horas.
- Escala en 3: 1 hora.
- Ruta de 3 a 4: 1 hora.

En total, esto nos da un tiempo de viaje de 7 horas.

Cuando el algoritmo evalúa las rutas desde 1 hasta 4, considera todas las ciudades intermedias posibles k. En este caso, 2 y 3. El algoritmo calculará el tiempo mínimo para viajar de 1 a 4 pasando por 2 y pasando por 3, y luego seleccionará la ruta más corta.

Lo realmente importante aquí es que cuando se calcula el tiempo mínimo de viaje de 1 a 4 pasando por una ciudad intermedia k, el algoritmo ya ha calculado las rutas más cortas desde 1 hasta k y desde k hasta 4 para todas las ciudades k, lo que garantiza que estamos seleccionando las rutas más cortas en cada etapa del proceso.

Por lo tanto, el POB se cumple en este algoritmo, ya que la solución global óptima (la ruta más corta de 1 a 4) se construye a partir de soluciones óptimas a subproblemas locales (las rutas más cortas de 1 a k y de k a 4, las cuales hemos calculado para cada una de las k antes de hacer la solución final, garantizando así que cogemos estas y nuestra solución óptima está compuesta a partir de soluciones de subproblemas de este). En este caso del ejemplo, nuestra solución óptima final ha sido 6, y como hemos demostrado 6 era una de las soluciones a los subproblemas que teníamos, así que finalmente hemos demostrado que se cumple el Principio de Optimalidad de Bellman (POB).

Diseño del algoritmo de cálculo de coste óptimo:

Para realizar el diseño del algoritmo de cálculo de coste óptimo en este caso, como ya se ha venido diciendo en los apartados anteriores, nos basamos para resolver el problema en el diseño del algoritmo de Floyd, el cuál viene en las transparencias y adjuntamos aquí mismo:

Algoritmo de Floyd (1962): $\Theta(V^3)$

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        D[i][j] = coste(i,j);

for (k=0; k<n; k++)
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (D[i][k] + D[k][j] < D[i][j] )
               D[i][j] = D[i][k] + D[k][j];
```

$$D_k(i,j) = \min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\}$$

Como podemos apreciar, en este se usa una ecuación de recurrencia similar o casi igual a la que nosotros hemos adaptado a nuestro problema. De hecho, esta plantilla sería casi la que buscamos, solo que nos falta tener en cuenta dentro de los bucles que a parte de la distancia entre *i* y *j,* y viceversa, también habría que sumar la escala $E(k)$. A parte de ese detalle, también tendríamos que tener en cuenta la actualización de los nodos de nuestra matriz *prox_nodo[ ]* para poder recuperar posteriormente la solución.

Viendo esta plantilla del diseño, si la adaptamos a nuestro problema, el diseño de nuestro algoritmo de cálculo del coste óptimo sería el siguiente:

```
//T(i,j) el tiempo de viaje entre dos ciudad i y j
//MV[ ] matriz donde realizaremos memorización, y guardaremos tiempos de viaje mínimos

alg_cost_opt(MV [ ][ ], k [ ], prox_nodo[ ][ ]) {
    para cada ciudad i              //Recorremos todas ciudades i (fil)
        para cada ciudad j          //Recorremos todas ciudades j (col)
            if ∃ viaje directo entre i y j  //Comprobamos si existe viaje directo entre ambos
                MV[i,j] = T(i,j)        //A cada celda ij, le asignamos el valor del viaje directo entre esas dos ciudades
                prox_nodo[i,j] = j      //Actualizamos prox_nodo

    para cada escala k              //Recorremos todas ciudades intermedias o escalas
        para cada ciudad i          //Recorremos todas ciudades i (fil)
            para cada ciudad j      //Recorremos todas ciudades j (col)
                si ∃ escala k para i, ∃ misma escala k para j   //
                    min = MV[i,k] + MV[k,j] + E(k)          //Guardamos el min actual
                    si min < MV[i,j]                        //si el min_actual es menor que el min guardado en MV[ ], actualizamos
                        MV[i,j] = min                       // celda con min y actualizamos prox_nodo
                        prox_nodo[i,j] = prox_nodo[i,k]
}
```

En este realizamos lo que hemos explicado anteriormente en el diseño de la memoria. Primero recorremos nuestra matriz de ciudades i y j (filas,columnas) guardando(si existe) el tiempo de viaje directo entre las respectivas ciudades *i,* y *j,* ($T(i, j)$), cada uno en su respectiva celda MV[*i,j,*].

Una vez guardados todo estos valores, recorremos todas las posibles escalas o ciudades intermedias (k), para todos los viajes entre *i,* y *j,* y vamos iterando por lo tanto para cada k posible por todas los viajes de la matriz. En cada una de estas iteraciones calculamos la fórmula del tiempo de viaje con escala, y si esta es menor que el tiempo de viaje directo (o que un viaje con escala que hubiese puesto antes), la sustituimos en la celda MV[*i,j,*] correspondiente en la matriz.

De esta manera conseguimos el diseño del algoritmo de coste óptimo, basándonos en el de Floyd, pero adaptándolo ligeramente para este problema.

Diseño del algoritmo de recuperación de la solución

Para este otro algoritmo, el cuál es crucial para poder realizar la recuperación de la solución con éxito, también vamos a basarnos en el algoritmo de recuperación de solución de Floyd expuesto en las transparencias, con tal de adaptarlo para poder resolver el problema planteado en este caso (que tiene algunos detalles más a tener en cuenta). Aquí lo adjuntamos:

Algoritmo de Floyd (1962):

Reconstrucción de los caminos más cortos

```
path(P,i,j)     // Devuelve el camino más corto de i a j
{
  if (P[i][j]==null) {      // Conexión directa de i a j
     return { (i,j) };
  } else {                  // Camino pasando por k
     k = P[i][j];
     return path(P,i,k) U path(P,k,j);
  }
}
```

Reconstrucción recursiva de los caminos más cortos pasando por un nodo intermedio k entre i y j (i→k→j).

Como podemos ver, este trata de reconstruir la solución del camino más corto mediante una opción recursiva, aunque en nuestro caso con la matriz prox_nodo[ ] se podría implementar iterativo o recursivo. El enfoque que tiene es realmente acertado para nuestro problema, pues va a devolver como camino la unión entre el viaje i y k, y el viaje k y j ($T(i,k) \cup T(k,j)$), que es la reconstrucción del camino solución que precisamente estábamos buscando.

En nuestro caso, nos resulta más lógico e intuitivo plantear un diseño que utilice recursividad. Sin embargo, el anterior diseño expuesto necesita tener en cuenta detalles del problema, como por ejemplo la posibilidad de no existencia de un viaje directo entre *i* y *j*. Veamos el diseño que hemos decidido plantear nosotros:

```
//prox_nodo[ ][ ] aquí será una matriz con los proximo nodo a usar en camino para i e j
// i,j → ciudades i e j, que serán origen y destino del viaje
//camino_sol[ ] vector que vamos modificando mediante recursividad para poder reconstruir solución óptima global (camino más corto)

recup_solucion( prox_nodo[ ][ ], i, j, camino_sol[ ]){

    si ( i es igual a j)               //Si i es igual a j, ya solamente añadimos i y hacemos return, solución reconstruida
        añadimos i a camino_sol
        return;

    si (prox_nodo[i,j] ≠ )      //  si (prox_nodo == -1) entonces hacemos return, ya que no hay próximo nodo
        return;

    añadimos i a camino_sol     //  si no es ultimo nodo, y existe proximo nodo, entonces añadimos este nodo a el camino_sol
    recup_solucion(prox_nodo[ ][ ], prox_nodo[i,j], j, camino_sol[ ])   // y llamamos a la función recursiva para que añada nodos avanzando
}                                                                       // en el camino hasta terminarlo y devolver el camino_sol
```

Este diseño nos permite la recuperación de la solución que buscábamos. En nuestro pseudocódigo hemos explicado paso a paso cómo funciona, pero básicamente lo que hace es; en caso de existir próximo nodo, y que i sea diferente de j, se añade i al camino solución y se vuelve a llamar con recursividad a la función para las posiciones prox_nodo[i,j] y j (siendo estas k y j). De esta manera conseguimos llenar nuestro camino_sol[ ] y reconstruir la solución a nuestro problema. En caso de no existir un siguiente nodo se hará un return, y si ya solo queda meter en el camino_sol[ ] el último nodo, se meterá y se hará return indicando que ya se tiene el camino solución. A la hora de implementarlo podemos usar una función envolvente donde declaremos ese vector camino_sol[ ] si nuestra función recursiva es un void.

Finalmente, nuestro vector camino_sol[ ] habrá quedado modificado conteniendo la solución final u mínimo global del problema que buscamos.

Implementación(algoritmos cálculo coste óptimo y recuperación solución)

Para finalizar, vamos a exponer la implementación de los algoritmos de los que hemos tenido que plantear el diseño anteriormente. Pese a que no expongamos todo el código (el cuál se encuentra en el zip adjuntado con la práctica), ejecutaremos el main con un ejemplo también para ver que ambos algoritmos y el código funcionan correctamente. Antes de seguir vamos a exponer que hemos señalizado los casos de que no haya un viaje directo enre i y j con la variable MAX de la biblioteca“limits”de c++, dando a entender que si no existe un viaje directo,

no sabríamos el número de escalas que deberíamos hacer para poder llegar de i a j. Dada esta aclaración para comprender más fácilmente algunos detalles del código, exponemos entonces el código de ambos algoritmos:

·Algoritmo cálculo coste óptimo:

```
// Función para inicializar las matricesMV y prox_nodo con los viajes directos primero.
void viajes_directos(int n, const vector<vector<int>>& T, vector<vector<int>>&MV, vector<vector<int>>& prox_nodo) {
    for (int i = 0; i < n; ++i) { //Aquí se exponen los casos base que hemos hablado en la parte de la ecuación de recurrencia
        for (int j = 0; j < n; ++j) {
            if (i == j) {MV[i][i] = 0;  // Caso base: tiempo de viaje de una ciudad a sí misma es 0
            } else if (T[i][j] != INF) {MV[i][j] = T[i][j];  // Caso base: tiempo de viaje directo
                prox_nodo[i][j] = j;  // Inicializar prox_nodo para el camino directo
            } else {MV[i][j] = INF;
                prox_nodo[i][j] = -1;  // Indica que no hay camino directo
            }
        }
    }
}

// Algoritmo de Floyd modificado
void coste_optimo(int n, vector<vector<int>>& MV, vector<vector<int>>& prox_nodo, const vector<int>& E) {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (MV[i][k] != INF &&MV[k][j] != INF &&MV[i][k] +MV[k][j] + E[k] <MV[i][j]) {
                    MV[i][j] =MV[i][k] +MV[k][j] + E[k];
                    prox_nodo[i][j] = prox_nodo[i][k];
                }
            }
        }
    }
}
```

La implementación de este algoritmo es casi igual a la que ya habíamos propuesto en el diseño. El diseño tenía una primera parte de bucles anidados que inicializan MV[ ] con los valore de los viajes directos, y una segunda parte que iba probando con todas las escalas en los viajes de i y j, y guardaba el valor del tiempo en caso de que este fuese aún menor que el que antes hubiese en MV[i,j], guardando así en casa caso el mínimo global.

A la hora de programarlo, nosotros hemos decidido dividir estas dos partes en dos funciones diferentes como se puede observar. Para la inicialización con los valores de viaje directo tenemos aquí ***viajes_directos()*:**

```cpp
// Función para inicializar las matricesMV y prox_nodo con los viajes directos primero.
void viajes_directos(int n, const vector<vector<int>>& T, vector<vector<int>>&MV, vector<vector<int>>& prox_nodo) {
    for (int i = 0; i < n; ++i) { //Aquí se exponen los casos base que hemos hablado en la parte de la ecuación de recurrencia
        for (int j = 0; j < n; ++j) {
            if (i == j) {MV[i][i] = 0;  // Caso base: tiempo de viaje de una ciudad a sí misma es 0
            } else if (T[i][j] != INF) {MV[i][j] = T[i][j];  // Caso base: tiempo de viaje directo
                prox_nodo[i][j] = j;  // Inicializar prox_nodo para el camino directo
            } else {MV[i][j] = INF;
                prox_nodo[i][j] = -1;  // Indica que no hay camino directo
            }
        }
    }
}
```

Este tiene la misma explicación que la que se ha dado para el diseño del algoritmo. Por otro lado, tenemos la segunda parte para actualizar MV[ ] en caso de encontrar mínimos globales aún menores que los viajes directos, y actualizando si se encuentran cada vez valores menores en viajes con escala para las celdas de MV[i,j]. A esta función la hemos llamado ***coste_optimo( )***, pretendiendo hacer referencia a este apartado:

```cpp
// Algoritmo de Floyd modificado
void coste_optimo(int n, vector<vector<int>>& MV, vector<vector<int>>& prox_nodo, const vector<int>& E) {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (MV[i][k] != INF &&MV[k][j] != INF &&MV[i][k] +MV[k][j] + E[k] <MV[i][j]) {
                    MV[i][j] =MV[i][k] +MV[k][j] + E[k];
                    prox_nodo[i][j] = prox_nodo[i][k];
                }
            }
        }
    }
}
```

De nuevo, este también funciona como anteriormente hemos explicado en el diseño.

·Algoritmo de recuperación de la solución:

Este otro algoritmo crucial para nuestro problema de memoria dinámica, lo hemos implementado también siendo muy fieles al diseño que habíamos propuesto. Hemos usado una función envolvente para utilizar la recursiva que de tipo void. Aquí las exponemos:

-***camino_sol_rec( )***:

```
// Función recursiva para construir el camino óptimo de i a j
void camino_sol_rec(const vector<vector<int>>& prox_nodo, int i, int j, vector<int>& camino) {
    if (i == j) {
        camino.push_back(i);
        return;
    }
    if (prox_nodo[i][j] == -1) {
        return;  // No hay camino de i a j
    }
    camino.push_back(i);
    camino_sol_rec(prox_nodo, i: prox_nodo[i][j], j, &: camino);
}
```

Esta función se encarga de seguir fielmente el diseño y los pasos de este. Su desarrollo ya ha sido en la parte mencionada, no vamos a redundar en esto.

-**camino_sol**( ):

```
// Función envolvente para la construcción del camino con función camino_sol_rec() de arriba
vector<int> camino_sol(const vector<vector<int>>& prox_nodo, int i, int j) {
    vector<int> camino;
    if (prox_nodo[i][j] == -1) {
        return camino;  // No hay camino de i a j
    }
    camino_sol_rec(prox_nodo, i, j, &: camino);
    return camino;
}
```

Como nuestra función recursiva es de tipo void y no va a poder devolver el vector solución en sí, nos hemos ayudado de esta función auxiliar que nos sirve para poder declarar el vector camino[ ], para llamar entonces aquí dentro a la función recursiva, y una vez se haya llenado el vector camino con la solución, hacemos return de este en esta función.

Una vez expuestos ambos algoritmos, vamos a realizar una ejecución del programa para un caso concreto de la tabla que se nos ha dado en el enunciado del guión de prácticas. Para un n=4, queremos saber el mínimo tiempo de viaje posible de la ciudad 0 a la 3, veamos la tabla que se nos ha dado en el enunciado: (la tabla viene ordenada del 1 al 4 en filas y columnas, pero para hacer más fácil su comprensión con el código nombraremos las columnas desde 0 a n-1 (0 a 3).

$T(i,j)$	**0**	**1**	**2**	**3**
0	0	2	1	3
1	7	0	9	2
2	2	2	0	1
3	3	4	8	0

Viendo esta tabla, vamos a ver el tiempo mínimo de viaje entre las ciudades 1 y 2. El tiempo de viaje directo $T(i,j)$ entre 1 y 2 es 9. Sin embargo, nuestro algoritmo dice que otro camino da menos tiempo aún, el camino con escalas: 1 3 0 2.
Comprobémoslo:

$$V(1,3,0,2) = T(1,3) + T(3,0) + E(k) + T(0,2) =$$
$$= 2 + 3 + 1 + 1 = 8$$

Efectivamente, el camino con escalas 1 3 0 2, nos da un tiempo menor que el viaje directo entre las ciudades 1 y 2, y encima este es el mínimo global de tiempo de viaje que existe en este caso entre estas dos ciudades, con un valor de 8.

Veamos que da entonces la ejecución de nuestra implementación del código:

```
/home/manumarin/Escritorio/pract_alg5/cmake-build-debug/pract_alg5
El tiempo óptimo de 1 a 2 es: 8
El camino óptimo de 1 a 2 es: 1 3 0 2

Process finished with exit code 0
```

Por lo tanto, podemos ver que nuestro código funciona correctamente, y que nuestros algoritmos han sido correctamente implementados. Así mismo, concluimos definitivamente con este problema.

Videojuego (P.3)

El problema es el siguiente:

Un videojuego se juega por turnos y se representa en un mapa cuadriculado bidimensional de f filas y c columnas. El jugador siempre entra al mapa por la esquina superior derecha, y sale por la esquina inferior izquierda. En cada turno, los posibles movimientos del jugador son: ir 1 casilla a la izquierda, ir 1 casilla abajo, o moverse 1 posición a la casilla inferior izquierda. Cada casilla del mapa puede estar vacía, contener un muro, o contener una bolsa de oro. Todas las casillas son transitables salvo las que tienen muros. El objetivo consiste en llegar a la salida pudiendo recoger tanto oro como sea posible (pasar por tantas casillas que contengan una bolsa como se pueda).

Diseño de resolución por etapas y ecuación recurrente:

·Diseño de resolución por etapas:

El problema se puede resolver por etapas. En cada etapa se seleccionará por qué camino seguir, para ello se puntuará cada una de las opciones con un valor, una especie de heurística que nos indica el número de monedas que podríamos recoger si fuéramos por ese camino.

·Ecuación recurrente:

La solución depende de cada etapa, siendo en cada una de estas un nodo origen distinto se calculan los tres caminos posibles, los intermedios para llegar al destino y se selecciona el que más monedas nos pueda dar, así las situaciones posibles son:

- Si estamos en el nodo final, el problema termina y devolvemos que hemos encontrado una solución devolviendo un 0.
- En otro caso calculamos los posibles caminos, seleccionamos entre los candidatos el que nos ofrezca más monedas y comprobamos si se puede llegar al nodo final desde ese camino, si ese camino nos devuelve un 0 entonces es válido y hemos encontrado una solución, construyendo la solución en este caso. En caso de que ningún camino sea válido devolvemos “1” indicando que ese camino no sirve.

La recurrencia quedaría:

$$Mk(i, j) = Max\ \{Mk+1(i, j-1),\ Mk+1(i+1, j-1),\ Mk+1(i+1, j)\}$$

Donde M(i,j), sería el número de monedas que obtendremos desde la casilla fila i columna j.

Ahora vamos a desglosar la ecuación recurrente paso por paso:

Las variables y la notación:

- $Mk(i, j)$.- Representa la máxima cantidad de monedas (oro) que se puede recoger desde la casilla (i,j) en la etapa k.
- $Mk+1(i, j-1)$.- La máxima cantidad de monedas que se puede recoger en la etapa k+1 si el jugador se mueve a la izquierda desde la casilla (i,j).
- $Mk+1(i+1, j-1)$.- La máxima cantidad de monedas que se puede recoger en la etapa k+1 si el jugador se mueve en diagonal hacia abajo-izquierda desde la casilla (i,j).
- $Mk+1(i+1, j)$.- La máxima cantidad de monedas que se puede recoger en la etapa k+1 si el jugador se mueve hacia abajo desde la casilla (i,j).

La explicación:

- **Movimiento a la Izquierda:** $Mk+1(i, j-1)$
  Considera la cantidad de monedas recogidas si el jugador se mueve una casilla a la izquierda desde (i,j) hacia (i,j−1).
  Esto es relevante si la casilla (i,j−1) es transitable y no contiene un muro.
- **Movimiento en Diagonal Inferior Izquierda:** $Mk+1(i+1, j-1)$
  Considera la cantidad de monedas recogidas si el jugador se mueve en diagonal hacia abajo-izquierda desde (i,j) hacia (i+1,j−1).
  Esto es relevante si la casilla (i+1,j−1) es transitable y no contiene un muro.
- **Movimiento hacia Abajo:** $Mk+1(i+1, j)$
  Considera la cantidad de monedas recogidas si el jugador se mueve una casilla hacia abajo desde (i,j) hacia (i+1,j).
  Esto es relevante si la casilla (i+1,j) es transitable y no contiene un muro.

La ecuación recurrente esencialmente busca el valor máximo entre estas tres opciones, lo que representa elegir el camino que maximiza la cantidad de oro recogido desde la casilla (i,j).

Casos base:

- **Casilla de Salida** (**Destino Final**): (**f−1,0**)
  Cuando el jugador alcanza la casilla (f−1,0), el recorrido termina.
  La cantidad de monedas recogidas desde esta casilla es 0, ya que no hay más movimientos posibles.

$M(f-1,0) = 0$

- **Casillas con Muros:**
  Si una casilla contiene un muro (grid[i][j]=−1), no es transitable.
  La cantidad de monedas recogidas desde una casilla con un muro es irrelevante, ya que no se puede pasar por ella.

$M(i,j) = -\infty \quad \text{si grid}[i][j] = -1$

- **Casillas Iniciales:**
  La casilla inicial es la esquina superior derecha (0,c−1).
  Si la casilla inicial es transitable (grid[0][c−1]≠−1), se comienza desde ahí con el valor de esa casilla.

$M(0,c-1) = \text{grid}[0][c-1] \quad \text{si grid}[0][c-1] \neq -1$

Valor objetivo:

El valor objetivo en este problema es la cantidad máxima de oro que se puede recoger al moverse desde la celda inicial (esquina superior derecha del mapa) hasta la celda final (esquina inferior izquierda del mapa). Este valor se obtiene después de explorar todas las rutas posibles permitidas por las reglas del juego (movimientos hacia la izquierda, hacia abajo, y hacia la esquina inferior izquierda) y sumar el oro encontrado en cada celda visitada, evitando los muros.

Diseño de la memoria:

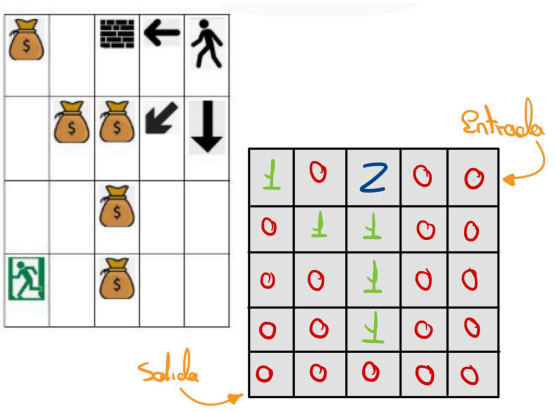
Para desarrollar la solución del problema de maximización de oro recogido en un mapa vamos a utilizar una matriz n×n con programación dinámica, vamos a describir en detalle los pasos necesarios para el diseño de la memoria:

Representación del Mapa

El mapa se representa como una matriz n×n en la que cada casilla puede tener uno de los siguientes valores:

- **0.**- Casilla vacía.
- **1.**- Casilla con una moneda de oro.
- **2.**- Casilla con un muro (no transitable).

Aquí una imagen para verlo visualmente en una matriz cuadrada:



Inicialización de la Matriz de Memoria

Vamos a utilizar una matriz de tamaño n×n para almacenar la máxima cantidad de oro que se puede recoger al llegar a cada casilla (i,j). Inicializamos esta matriz con valores negativos (por ejemplo, −∞, -100, -10, deben ser medio altos) para indicar que inicialmente las casillas no son alcanzables. La casilla de entrada, ubicada en (0,n−1), se inicializa con su propio valor si es transitable (es decir, si no es un muro).

Verificación del P.O.B.:

El Principio de Optimalidad de Bellman se cumple en nuestro problema porque la función que vamos a implementar siempre elige la opción que maximiza la cantidad de monedas recogidas desde cada casilla (**calcula_max**(heur_i, heur_d, heur_a, mov).

Al evaluar todas las rutas posibles desde una casilla, este algoritmo selecciona la mejor en términos de monedas recogidas, asegurando decisiones óptimas en cada subproblema. Dado que cada sub decisión es óptima, la combinación de estas decisiones garantiza que la solución final también sea óptima, maximizando así el total de monedas recogidas al llegar a la salida del mapa.

Diseño del algoritmo de cálculo de coste óptimo:

Para el cálculo del coste óptimo tenemos el siguiente diseño:

```
FUNCIÓN camino_PD (m, size, mon, ruta, fil, col)
    Si estoy en la casilla final (fil == tam-1 y col == 0):
        Devolver 0
    posI = (fil, col - 1)
    Si esa columna < 0 ó la casilla es muro, heur_i = -10
    posD = (fil + 1, col - 1)
    Si la fila >= size, la columna < 0 ó la casilla es muro, heur_d = -10
    posA = (fil + 1, col)
    Si la fila >= size ó la casilla es muro, heur_a = -10

   heur_i = calcularHeurística(posI, m, size)
   heur_d = calcularHeurística(posD, m, size)
    heur_a = calcularHeurística(posA, m, size)

    Declarar mov
    Declarar valor = -1

    Mientras valor != 0:
        Si se ha llegado a la meta (valor == 1):
            Si mov == "izquierda":
            heur_i = -10
            Sino si mov == "diagonal":
            heur_d = -10
            Sino si mov == "abajo":
            heur_a = -10

        Si no hay ningún camino válido
            Devolver 1

        calcula_max(heur_i, heur_d, heur_a, mov)

        Si mov == "izquierda":
            valor = camino_PD(m, size, mon, ruta, posI[0], posI[1])
```

```
        Sino si mov == "diagonal":
            valor = camino_PD(m, size, mon, ruta, posD[0], posD[1])
        Sino si mov == "abajo":
            valor = camino_PD(m, size, mon, ruta, posA[0], posA[1])

    Devolver valor
```

Este pseudocódigo representa un algoritmo recursivo para encontrar el camino óptimo para recolectar monedas en un mapa bidimensional. Aquí está la explicación:

- **Verificación de la casilla final:**
  Se verifica si la posición actual coincide con la casilla final del mapa (size-1, 0). Si es así, se devuelve 0, indicando que se ha llegado a la salida.
- **Cálculo de las posibles rutas:**
  Se calculan las posiciones de las tres posibles direcciones hacia las que se puede mover el jugador: izquierda, diagonal inferior izquierda y abajo.
- **Cálculo de la heurística para cada posible ruta:**
  Para cada posible dirección, se calcula la heurística utilizando la función calcularHeuristica, que determina la cantidad de monedas que se pueden recoger siguiendo ese camino.
- **Exploración Recursiva de las posibles rutas:**
  Se inicia un bucle mientras el valor de retorno (valor) no sea 0, lo que indica que se ha llegado a la salida.
  Si el valor de retorno es 1, significa que no hay ningún camino válido desde la posición actual. En este caso, se marca el camino como inválido y se vuelve a calcular la heurística para cada dirección.
  Si hay al menos un camino válido, se selecciona el que tenga la mayor heurística utilizando la función calcula_max.
  Se realiza una llamada recursiva a la función camino_PD con la nueva posición correspondiente al camino seleccionado.
  Este proceso continúa hasta que se encuentra un camino que lleva a la salidadel mapa.
- **Devolver el valor final:**

Una vez que se alcanza la salida del mapa, se devuelve el valor obtenido en la llamada recursiva, que indica si se ha llegado a la salida (0) o si no hay ningún camino válido disponible (1).

En resumen, este algoritmo explora recursivamente todas las posibles rutas desde la posición actual hasta la salida del mapa, seleccionando en cada paso el camino que maximiza la cantidad de monedas recogidas.

Para el cálculo de la heurística nos hemos ayudado de dos funciones:

FUNCIÓN CALCULARHEURISTICA(pos, matriz, tamaño)

```
// Obtengo las coordenadas de fila y columna de la posición
posf = pos[0]
posc = pos[1]
// Inicializo la heurística
heuristica = 0
// Recorro el área explorada por el camino hasta la posición actual (posf, posc)
Para cada fila desde 0 hasta posf:
    Para cada columna desde 0 hasta posc:
        // Si la casilla contiene una moneda, incremento la heurística
        Si m[fila][columna] == 1:
        heuristica++
// Si la posición actual contiene una moneda, la contabilizo
Si m[posf][posc] == 1:
    heuristica++
// Si la posición actual es un muro o está fuera del mapa, asigno una penalización
Si m[posf][posc] == 2 o posf >= tamaño o posc < 0:
    heuristica = -10

Devolver heuristica
```

FUNCIÓN calcula_max(heur_i, heur_d, heur_a, mov)

Esta función se encarga de devolver en “mov” el máximo valor de “heur_i”, “heur_d” y “heur_a”.

Diseño del algoritmo de recuperación de la solución:

Una vez encontramos un camino que llega al destino, actualizamos el número de monedas que hemos recogido y los pasos totales que ha tenido que ejecutar para llegar al destino.

Aquí el diseño del algoritmo de recuperación:

```
// Actualización del número de monedas recogidas
Si mov == "izquierda":
   Si m [ posI[0] ] [ posI[1] ]== 1:
      Incrementar mon

Sino Si mov == "diagonal":
   Si m [ posD[0] ] [ posD[1] ] == 1:
      Incrementar mon

Sino Si mov == "abajo":
   Si m [ posA[0] ] [ posA[1] ] == 1:
      Incrementar mon

// Actualizamos el camino
camino.agregar(mov)

Devolver valor
```

Aquí la explicación de qué hace el fragmento anterior:

- **Actualización del número de monedas recogidas**
  Se verifica la dirección en la que se expandió la búsqueda (mov).
  Si el movimiento fue hacia la izquierda, se verifica si la casilla a la que se movió el jugador contiene una moneda (m [posI[0]] [posI[1]] == 1). En caso afirmativo, se incrementa el contador de monedas recogidas (mon).
  Si el movimiento fue en diagonal, se realiza la misma verificación en la casilla correspondiente a esa dirección (m [posD[0]] [posD[1]] == 1).
  Si el movimiento fue hacia abajo, se realiza la misma verificación en la casilla correspondiente (m [posA[0]] [posA[1]] == 1).
- **Actualización del camino**

Se agrega la dirección en la que se movió la búsqueda al registro del camino (camino.agregar(mov)).

- **Devolver el valor**

  Se devuelve el valor obtenido en la función (valor). Este valor indica si se ha llegado a la salida del mapa (0) o si no hay ningún camino válido disponible (1).

En resumen, este fragmento de código se encarga de actualizar el contador de monedas recogidas y registrar la dirección del movimiento en el camino, como parte del proceso de recuperación de la solución del algoritmo principal.

Implementación(algoritmos cálculo coste óptimo y recuperación solución):

Para la implementación de nuestro algoritmo hemos realizado el siguiente código:

```
8  void max( int hi, int hd, int ha, string & mov){
9          if (hi >= hd) {
10                 if (hi >= ha) {
11                         mov = "izquierda";
12                 }
13                 else mov = "abajo";
14         }
15         else if (hd >= ha) {
16                 mov = "diagonal";
17         }
18         else mov = "abajo";
19 }
```

Esta es una función auxiliar que hemos utilizado para calcular qué movimiento debemos hacer, si ir para la izquierda, para la diagonal o para abajo.

Ahora la función principal:

```
int camino_PD( int **m, int size, int &mon, vector<string> &ruta, int fil, int col){
        int hi = 0, hd = 0, ha = 0;
        if (fil == size-1 && col == 0) return 0;

        //Todos los caminos
        //izq
        int fili = fil;
        int coli = col - 1;
        if (coli < 0) hi = -10;
        if (m[fili][coli] == 2) hi = -10;

        //diagonal
        int fild = fil + 1;
        int cold = col - 1;
        if (fild >= size) hd = -10;
        if (cold < 0) hd = -10;
        if (m[fild][cold] == 2) hd = -10;

        //abajo
        int fila = fil + 1;
        int cola = col;
        if (fila >= size) ha = -10;
        if (m[fila][cola] == 2) ha = -10;

        //Heurísticas
        //izq
        for (int i = fili; i < size; i++) {
                for (int j = 0; j <= coli; j++) if (m[i][j] == 1) hi++;
        }

        //diagonal
        for (int i = fild; i < size; i++) {
                for (int j = 0; j <= cold; j++) if (m[i][j] == 1) hd++;
        }

        //abajo
        for (int i = fila; i < size; i++) {
                for (int j = 0; j <= cola; j++) if (m[i][j] == 1) ha++;
        }

        string mov;
        int aux = -1;
```

```
        while (aux != 0) {
                if (aux == 1) {
                        if (mov == "izquierda") hi = -10;
                        if (mov == "diagonal") hd = -10;
                        if (mov == "abajo") ha = -10;
                }

                //Sino hay camino válido
                if (hi < 0 && hd < 0 && ha < 0) return 1;

                max(hi, hd, ha, mov);

                if (mov == "izquierda") aux = camino_PD(m, size, mon, ruta, fili, coli);
                else if (mov == "diagonal") aux = camino_PD(m, size, mon, ruta, fild, cold);
                else if (mov == "abajo") aux = camino_PD(m, size, mon, ruta, fila, cola);
        }

        //Cambiamos las monedas
        if (mov == "izquierda") {
                if (m[fili][coli] == 1) mon++;
        }
        else if (mov == "diagonal") {
                if (m[fild][cold] == 1) mon++;
        }
        else if (mov == "abajo") {
                if (m[fila][cola] == 1) mon++;
        }

        ruta.push_back(mov);
        return aux;
}
```

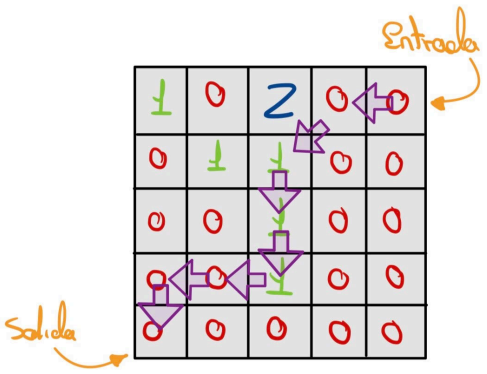
Y por último el main que cómo se puede observar, hemos calculado un ejemplo cómo el que venia en el problema pero para una matriz cuadrada 5x5:

```cpp
int main() {
    // Definir la matriz
    int size = 5;
    int **m = new int*[size];
    for (int i = 0; i < size; i++) {
        m[i] = new int[size];
    }
    m[0][0] = 1; m[0][1] = 0; m[0][2] = 2; m[0][3] = 0; m[0][4] = 0;
    m[1][0] = 0; m[1][1] = 1; m[1][2] = 1; m[1][3] = 0; m[1][4] = 0;
    m[2][0] = 0; m[2][1] = 0; m[2][2] = 1; m[2][3] = 0; m[2][4] = 0;
    m[3][0] = 0; m[3][1] = 0; m[3][2] = 1; m[3][3] = 0; m[3][4] = 0;
    m[4][0] = 0; m[4][1] = 0; m[4][2] = 0; m[4][3] = 0; m[4][4] = 0;

    //Imprimir la matriz
    cout << "Matriz de la que partimos: " << endl;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) cout << m[i][j] << " ";
        cout << endl;
    }

    // Inicializar otras variables
    int mon = 0;
    vector<string> ruta;

    int posf = 0;
   int  posc = size - 1;
    // Llamar a la función para resolver el problema
    int resultado = camino_PD(m, size, mon, ruta, posf, posc);

    // Imprimir resultados
    if (resultado == 0) {
        cout << "Se encontró una solución." << endl;
        cout << "Número de monedas recogidas: " << mon << endl;
        cout << "Ruta óptima: " << endl;
        for (int i = ruta.size() - 1; i >= 0; i--) cout << ruta.at(i) << "    ";
        cout << endl;
    } else {
        cout << "No se encontró una solución." << endl;
    }

    // Liberar memoria
    for (int i = 0; i < size; i++) delete[] m[i];
    delete[] m;

    return 0;
}
```

Aquí podemos ver la ejecución del código anterior:

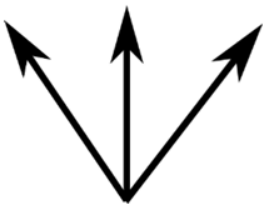
```
usuario@raulmtz:~/Escritorio/ALG/practicas/p5_PD$ gedit videojuego.cpp
usuario@raulmtz:~/Escritorio/ALG/practicas/p5_PD$ g++ videojuego.cpp -o videojuego
usuario@raulmtz:~/Escritorio/ALG/practicas/p5_PD$ ./videojuego
Matriz de la que partimos:
1 0 2 0 0
0 1 1 0 0
0 0 1 0 0
0 0 1 0 0
0 0 0 0 0
Se encontró una solución.
Número de monedas recogidas: 3
Ruta óptima:
izquierda    diagonal    abajo    abajo    izquierda    izquierda    abajo
```

Cómo vemos, para este caso llega desde la entrada a la salida recogiendo 3 monedas y sigue dicha ruta, mostraré el ejemplo gráfico para ver cómo lo ha hecho:



Pixel Mountain (P. 4)

Nuestro ejercicio consiste en:
Tenemos que ponernos en forma, y hemos decidido empezar subiendo la “Pixel Mountain”. Para ello, tenemos que ascender desde una posición baja hasta la cumbre. Evidentemente, una parte muy importante de toda ascensión es decidir por dónde deberíamos subir. En nuestra ascensión siempre subimos (sin desfallecer) pero podemos movernos directamente hacia arriba, o desplazarnos a la izquierda o derecha en diagonal:



En cada posible posición de la montaña tenemos un coste asociado de la dificultad que tiene llegar a esa posición. El objetivo es calcular el recorrido con menor dificultad:

Un ejemplo:

2	8	9	**5**	8
4	4	6	**2**	3
5	7	5	6	**1**
3	2	5	**4**	8

De esta montaña 5x4 se puede observar que el mejor camino (el más fácil) es el marcado en negrita, es decir, el que implica la dificultad de 4+1+2+5 = 12. El plan (secuencia de pasos) serían las posiciones 3-4-3-3 (si numeramos desde 0).

Diseño de resolución por etapas y ecuación recurrente:

·Diseño de resolución por etapas:

El problema se puede resolver por etapas. En cada etapa, se selecciona la celda de la matriz a la que se va a mover. Supondremos que se pueden realizar movimientos hacia arriba, arriba-izquierda o arriba-derecha para asegurar una solución óptima factible desde las etapas más tempranas.

·Ecuación recurrente:

Para resolver el problema 4 “Pixel Mountain” minimizando el coste que supone moverse por la matriz, hemos decidido realizar la siguiente ecuación recurrente dado los siguientes pasos:

- Definimos C(i,j) como el coste mínimo para llegar a la posición (i,j) desde nuestra base de la montaña.

Tenemos en cuenta los movimientos permitidos que son los siguientes:

- Desde la celda actual hacia arriba C(i+1, j)
- Desde la celda actual hacia arriba a la izquierda C(i+1, j-1)
- Desde la celda actual hacia arriba a la derecha C(i+1, j+1)

De esta manera podríamos permitir la siguiente ecuación de recurrencia:

$$T(i,j) = \text{cost}(i,j) + \min \begin{cases} T(i+1,j) \\ T(i+1,j-1) \\ T(i+1,j+1) \end{cases}$$

Dónde cost (i,j) representa el coste inicial de esa misma celda.

Y tenemos que tener en cuenta las siguientes condiciones:

Desde la celda actual hacia arriba a la izquierda C(i+1, j-1), "j" debe ser > 0 por que si no podríamos salirnos de la matriz.

Análogamente, desde la celda actual hacia arriba a la derecha C(i+1, j+1), "j" debe ser < columnas, por que si no, podríamos salirnos de la matriz.

Por último, también debemos de tener en cuenta los **casos base.**

En este caso, el **caso base** será:

Para la fila más baja (la primera verdaderamente):

$$T(0,j) = \text{cost}\ (0,j)$$

Es decir, su valor es el de la celda misma. Análogamente, si esa fila fuera la fila más alta (imaginando que la montaña de pixeles es 1x1):

$$T(\text{nº fila más alta -1 , j}) = \text{cost}\ (\text{nº fila más alta -1 , j})$$

Por lo que finalmente deducimos que en cada posición (i,j), el coste mínimo para llegar, se calcula sumando el coste actual de esa posición, al mínimo de los costes de las posibles posiciones a las que se puede llegar (arriba, arriba-izquierda, arriba-derecha).

Valor objetivo:

Se desea determinar el valor $T(0,j)$, que representa el coste mínimo para ascender hasta la cima de la "Pixel Mountain" partiendo desde la base hasta la última fila, considerando todos los posibles caminos ascendentes (diferentes j) y los costos asociados a cada posición desde la base de la montaña.

Verificación del P.O.B.:

Como ya debemos de saber, Bellman nos dice que su principio de optimalidad se verifica si toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas, pero no nos dice que si tenemos dichas soluciones, podamos combinarlas para obtener la solución óptima del problema original. (Presentación Pg.11, Teoría T6)

Por lo tanto, al aplicar el algoritmo de Bellman a nuestro problema, podemos deducir lo siguiente:
Para cualquier solución T(i,j), debe de haber 2 movimientos que deberemos de considerar para obtener las soluciones óptimas a los subproblemas:

- Movimiento directo T(i,j)
- Movimiento diagonal T(i, j (+ ó -) 1)

Pero ya sabemos que esos valores también serán óptimos ya que la ecuación recurrente siempre va a seleccionar el que nos de la solución mínima entre ellos por lo que no puede existir un valor menor que el mínimo entre estas dos opciones, garantizando que la solución óptima, estará compuesta por el conjunto de sub soluciones óptimas que minimicen finalmente nuestra solución global, cumpliendo el principio de Bellman.
Diseño de la memoria:
Para resolver el problema, T(i,j) se representará como una matriz.
Esta matriz tendrá filas, donde cada fila i estará asociada al nivel de la montaña correspondiente al tipo de movimiento permitido (columnas/j).

La matriz tendrá N+1 columnas, donde cada columna estará asociada a cada posible posición horizontal j de la montaña, desde la columna 0 hasta la columna N.
Cada celda de la matriz T(i,j) contendrá el mínimo coste para llegar a la posición (i,j) desde la base de la montaña, considerando todos los posibles movimientos permitidos desde los niveles anteriores de la montaña.

La memoria se rellenará siguiendo estos pasos:

En primer lugar, se rellenan las celdas correspondientes a los casos base. Es decir, la primera fila T(0,j) se inicializará con valores de la primera fila de la montaña.

En segundo lugar, se rellenan las filas {1,2,...,n} en orden secuencial.
Cada fila se rellenará en orden creciente de columnas {0,1,2,...,N}, calculando el coste mínimo para llegar a cada posición (i,j) utilizando la ecuación recurrente mencionada anteriormente.

Diseño del algoritmo de cálculo de coste óptimo:

Con este diseño construimos el algoritmo de Programación Dinámica como sigue:

```
matriz EscalaPixelMountain(montaña){
    matriz Costes = montaña

    Base Costes = Base montaña

    desde i=1 hasta tamaño de montaña{
        desde j=0 hasta tamaño de montaña{
            minimo=montaña[i-1][j]
            if (j>0){
                minimo= min{minimo,montaña[i-1][j-1]
            }
            if (j<tamaño de montaña){
                minimo= min{minimo, montaña[i-1][j+1]
            }
            Costes[i][j]=minimo + Costes[i][j]
        }
    }
}
```

Nuestro algoritmo se basa en a través de un duplicado de nuestra matriz correspondiente a la montaña, recorrerla fila por fila analizando los posibles movimientos que podemos realizar mientras la escalamos, de forma que en este

duplicado de la matriz al que llamaremos Costes se almacenará el resultado del movimiento que requiera un menor esfuerzo por parte del escalador.

Diseño del algoritmo de recuperación de la solución:

Finalmente, una vez hemos obtenido nuestra matriz de costes, únicamente deberemos recuperar el camino a partir del cuál hemos conseguido llegar a la cima.

Para ello podemos utilizar el siguiente algoritmo:

```
vector RecuperarSolucion(Costes, montaña){
    Cima= ultima fila de Costes
    PosMinimoCoste= min(Costes[Cima])

    vector Camino
    Camino + PosMinimoCoste
    Para cada fila i de costes empezando desde el final{
        if (PosMinimoCoste>0 &&
            Costes posicion izquierda anterior==Costes posicion actual - montaña posicion actual){

            PosMinimoCoste= izquierda
        }
        if (PosMinimoCoste<tamaño de montaña &&
            Costes posicion derecha anterior==Costes posicion actual - montaña posicion actual){

            PosMinimoCoste= derecha
        }
        vector + PosMinimoCoste
    }
    return Camino
}
```

Implementación(algoritmos cálculo coste óptimo y recuperación solución):

Finalmente, para demostrar que nuestro algoritmo funciona, vamos a implementarlo en c++:

Primero implementaremos el código de nuestro algoritmo encargado de encontrar la solución:

```
matriz EscalaPixelMountain(const matriz & mountain) {

    matriz Costes=mountain;

    for (int i = 1; i < Costes.size(); ++i) {
        for (int j = 0; j < Costes[0].size(); ++j) {
            int min_cost = Costes[i-1][j];  // Coste desde arriba
            if (j > 0) {
                min_cost = min(min_cost, Costes[i-1][j-1]);  // Coste desde arriba-izquierda
            }
            if (j < Costes[0].size() - 1) {
                min_cost = min(min_cost, Costes[i-1][j+1]);  // Coste desde arriba-derecha
            }
            Costes[i][j] = mountain[i][j] + min_cost;
        }
    }

    return Costes;
}
```

Y a continuación implementaremos nuestro algoritmo encargado de recuperar el camino hasta la cima:

```
pair<int,vector<int>> RecuperarCamino(matriz Costes, matriz mountain){

    vector<int> Camino;
    int Cima= Costes.size()-1;
    int index=0;
    int minimo=Costes[Cima][index];

    for (int i=1;i<mountain[0].size();i++){
        if (minimo>Costes[Cima][i]){
            minimo=Costes[Cima][i];
            index=i;
        }
    }

    int principio=index;
    Camino.push_back(index);

    for (int i=Cima; i>0;i--){
        if(index>0 && Costes[i-1][index-1]==Costes[i][index]-mountain[i][index]){
            index-=1;
        }
        if(index<Costes[0].size() && Costes[i-1][index+1]==Costes[i][index]-mountain[i][index]){
            index+=1;
        }
        Camino.push_back(index);
    }
    return pair<int,vector<int>>(Costes[Costes.size()-1][principio],Camino);
}
```

De estar forma, al introducir la montaña utilizada como ejemplo anteriormente:

2	8	9	**5**	8
4	4	6	**2**	3
5	7	5	6	**1**
3	2	5	**4**	8

Nos quedaría la siguiente ejecución:

```
leonin04@SurfaceDavid:~/ALG/Practica5$ ./pixel_mountain
La tabla nos queda de la siguiente forma:
2 8 9 5 8
6 6 11 7 8
11 13 11 13 8
14 13 16 12 16

El minimo esfuerzo necesario para escalar la mountain es de: 12
El camino es : 3 4 3 3
```

Ejecución que coincide con la solución vista anteriormente, y por tanto nuestro algoritmo es funcional.