UNIVERSIDAD DE GRANADA

FUNDAMENTOS DE INGENIERÍA DEL SOFTWARE: PRÁCTICA 1

DOBLE GRADO EN INGENIERÍA INFORMÁTICA + ADE

LISTA ESTRUCTURADA DE REQUISITOS

Autores

ISMAEL SALLAMI MORENO

JULIÁN CARRIÓN TOVAR

JESÚS RODRÍGUEZ GONZÁLEZ

ALICIA RUIZ GÓMEZ

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

E.T.S. DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, a 16 de septiembre de 2025

ÍNDICE GENERAL

1

PROBLEMA 1

1.1 DESCRIPCIÓN DEL PROBLEMA

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 sea mayor que el coste total de una serie de alquileres más breves. En tal caso, se puede devolver la primera 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

Figura 1: matriz de costes

1.2 SOLUCIÓN: DISEÑO DE COMPONENTES Y DEL ALGORITMO

1.2.1 *Resolución del problema por etapas*

En cada etapa se seleccionaría ir o no a una aldea próxima. Supondremos que las aldeas están ordenadas de menor a mayor coste para asegurar una solución óptima factible desde las etapas más tempranas.

1.2.2 *Ecuación recurrente*

Nuestra ecuación de recurrencia consta de el calculo del mínimo coste de entre ir directamente de la aldea i hasta la aldea j o tomar otras canoas en otras aldeas anteriores. Siendo Coste(i,j) el coste directo de ir de i a j en canoa y MIN(i,k)+MIN(k,j) siendo el calculo de otro camino para ir desde la aldea i hasta la aldea j alquilando otra canoa en algún pueblo anterior. A esta ecuación de recurrencia se le añaden las restricciones de que no podemos cambiar de canoas en aldeas mas arriba del río debido a la corriente. Esto lo reflejamos con k>i y k<j. k siendo una aldea intermedia de nuestro recorrido.

$$MIN(i,j) = min \begin{cases} Coste(i,j) \\ MIN(i,k) + MIN(k,j) \text{si } i < k < j \end{cases}$$

1.2.3 *Valor objetivo*

Se desea conocer el valor OPT(i,j), el mínimo coste para realizar un viaje desde la primera aldea hasta la aldea a la que se desea llegar.

1.2.4 *Verificación del cumplimiento del P.O.B*

Para demostrar que se aplica el principio de optimalidad de Bellman tenemos que demostrar que la solución óptima se consigue a partir de las soluciones óptimas los subproblemas que componen a nuestro problema. En este caso estos subproblemas son los costes mínimos de viajar desde una aldea (i) hasta otra aldea (j) para diferentes valores de i. En este caso se cumple debido a que nuestra ecuación de recurrencia siempre va escogiendo el mínimo coste no es posible que exista una forma alternativa para ir desde la aldea i hasta la aldea j con un menor coste.

1.2.5 *Diseño de la memoria*

Para el diseño de la memoria, decidimos usar una matriz en la que almacenar los costes mínimos ya calculados para ir de una aldea *i* a otra *j*. Para ello, usamos una matriz en la que se irán almacenando los valores de costes calculados. Cuando esta matriz este completa (solo la mitad superior, debido a la naturaleza del problema) se nos quedara la solución en la primera fila en forma de costes. Cuando esto ocurra pasaremos estos costes a nuestro camino ya con el numero correspondiente a cada aldea por las que tendremos que pasar en forma de secuencia.

1.2.6 *Diseño del algoritmo de cálculo de coste óptimo*

```
calcviajeopt( matriz rio, matriz opt, salida, destino){
    para i=destino-1 hasta salida{
        para j=destino hasta i {
            opt[i][j]=rio[i][j] + min(fila j de opt)
        }
    }
    Devolver min(opt[salida])
}
```

Para la resolución de este problema, hemos simplificado la idea de programación dinámica agregando las restricciones que nos imponía el enunciado. De esta forma, y como se puede observar en el pseudocódigo anterior, únicamente rellenaremos la mitad de la tabla(debido a la imposibilidad de acceder a una canoa que hayamos dejado atrás). Así, rellenaremos cada casilla de nuestra tabla con la distancia desde la aldea de la que procedamos hasta la que estemos comprobando en el momento, sumada a la mínima distancia posible desde esta última aldea a nuestro destino final. Una vez la tabla se encuentre rellena, nuestra solución final corresponderá al mínimo almacenado en la fila correspondiente al punto de salida de nuestra tabla.

1.2.7 *Diseño del algoritmo de recuperación de la solución*

```
camino recuperarsol(matriz rio, matriz opt, salida, destino){
    camino = vacio

    minimo=min(opt[salida])
    camino += salida
```

```
        camino += minimo

        mientras minimo sea distinto de destino{
            minimo=min(opt[minimo])
            camino + =  minimo
        }
        Devolver camino
    }
```

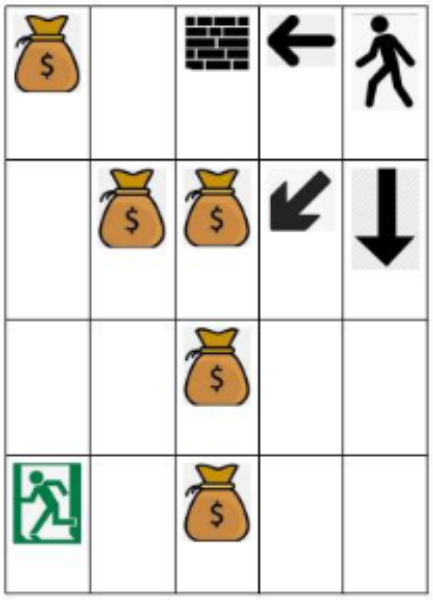
Conociendo la tabla rellenada tras la ejecución del algoritmo anterior, buscaremos conocer en que aldeas hemos parado para obtener el camino a recorrer. Por ello, buscaremos el mínimo de nuestra fila de salida, añadiendo a nuestro camino tanto nuestra aldea de partida como la aldea en la que realizaremos la primera parada(correspondiente al mínimo). A continuación realizaremos esto sucesivamente hasta llegar al destino, tomando como partida la aldea en la que hayamos realizado la parada y añadiendo a la solución únicamente las siguientes en las que vayamos a parar.

2

PROBLEMA 3

2.1 DESCRIPCIÓN DEL PROBLEMA

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). En el ejemplo siguiente, el jugador puede conseguir un máximo de 3 bolsas de oro con los movimientos permitidos.



2.2 SOLUCIÓN: DISEÑO DE COMPONENTES Y DEL ALGORITMO

2.2.1 *Resolución del problema por etapas*

Este problema puede resolverse mediante etapas. En cada etapa podemos barajar que posibilidad nos sale más rentable acorde a nuestros objetivos. Como nuestros objetivos son maximizar la cantidad de oro que vamos recogiendo en función de las casillas por las que pasamos, debemos de escoger aquellas posiciones, que en cómputo total, al salir del mapa del juego consigamos la máxima cantidad de bolsas de oro. Cabe destacar que para las explicaciones que vamos a realizar a continuación vamos a pensar que la representación del mapa va a corresponder con una matriz.

2.2.2 *Ecuación recurrente*

La solución depende de las etapas (a que posición debo de mover el personaje del videojuego para obtener la mayor cantidad de oro). Vamos a denotar como $T(i, j)$ el máximo número de bolsas de oro que se puede conseguir al llegar a la casilla (i, j). Como sabemos i y j corresponde con las posiciones dentro de nuestro mapa del videojuego.

Debemos de considerar el tipo de casilla, ya que puede ser casilla con muro, sin muro, y dentro de sin muro que haya oro o no. Lo que haremos será denotar a las casillas con un valor negativo muy grande, es decir, casillas = $-\infty$. Para inicializar toda la matriz, y ya mediante el algoritmo de programación dinámica vamos a ir resolviendo el problema y devolver una matriz final T, la cual explicaremos a continuación. Por lo que podemos afirmar que si la casilla *(i,j)* tiene un muro $T(i, j) = -\infty$. Esto lo ampliamos a que valdrá $T(i, j) = -\infty$ cuando, en general, la casilla sea no transitable. También el valor puede llegar a ser no accesible debido a la propagación de la innaccibilidad. Al calcular los valores de *T(i)(j)*, si todos los caminos posibles hacia una celda provienen de celdas inaccesibles (es decir, que contienen *INT_MIN*), entonces esa celda también será inaccesible. De esta manera, la inaccessibilidad se propaga correctamente a través de la matriz. Esta representación ayuda a asegurar que las celdas inalcanzables no se utilicen en cálculos posteriores.

En el caso de que no sea un muro, podemos establecer el valor de $T(i, j)$ en función de que si hay oro o no en la casilla. Así que podemos expresar $T(i, j)$ cuando no hay un muro de la siguiente manera:

$$T(i,j) = max\{T(i-1,j-1), T(i-1,j), T(i,j-1) + Oro(i,j)\}$$

Cabe destacar que va a variar en función de la posición en la que nos encontremos, ya que en el caso de que *(i,j)* sea *(o,o)* no se pueden acceder a diversas posiciones. La función *Oro(i,j)* devuelve *1* en el caso de que si haya oro en la casilla *(i,j)* y *o* en caso contrario.

En base a este planteamiento, podemos establecer que nuestra ecuación de recurrencia sería:

$$T(i,j) \begin{cases} T(i,j) = -\infty \text{ si hay un muro en (i,j) o es una casilla no transitable} \\ T(i,j) = max\{T(i-1,j-1), T(i-1,j), T(i,j-1) + Oro(i,j)\} \text{ si no hay muro en (i,j)} \end{cases}$$

Los casos base para esta ecuación serían:

- Si nos encontramos en la casilla inicial estamos en $T(0,0) = Oro(0,0)$. El valor depende si hay oro o no en la casilla inicial.
- Si nos encontramos una casilla se encuentra fuera del mapa o que posee un muro, se considera una casilla no transitable.

2.2.3 *Valor objetivo*

Se desea conocer $T(i,j)$, el valor máximo de bolsas de oro que se pueden conseguir transitando desde la casilla *(o,o)* hasta la casilla (n-1,m-1) teniendo la posibilidad de moverse a las casillas *(i-1,j-1), (i-1,j) y (i,j-1)*, podemos acceder a esas posiciones siempre y cuando $i \in 0..n-1$ y $j \in 0..m-1$

2.2.4 *Verificación del cumplimiento del P.O.B*

En este caso siempre estamos calculando la solución óptima ya que la optimalidad de la casilla de *(i,j)* se ha calculado en base a otros casos que a su vez también son óptimos, por lo que podemos concluir que también es óptimo. Básicamente, Cualquier solución óptima debe estar formada por subsoluciones óptimas. En este caso se cumple, dado que *T(i,j)*, que asumimos óptimo, podrá tener valores *T(i-1,j-1), T(i-1,j) o T(i,j-1)*. Estos valores también son óptimos, dado que la ecuación recurrente siempre

va escogiendo el máximo entre los valores mencionados anteriormente. No es posible que exista un valor mayor que el máximo entre los valores para *T(i,j).*

2.2.5 *Diseño de la memoria*

- Para resolver el problema, usaremos una matriz, por lo que, *T (i,j)* se representará como una matriz.
- Esta matriz tendrá n filas y m columnas, que estarán asociadas a el lugar correspondiente dentro del mapa. Únicamente, podrán optar por valores entre 0...n-1.
- Cada celda de la matriz *T(i,j)* contendrá el máximo valor de bolsas de oro que se pueden obtener hasta llegar a la casilla *(i,j)* del mapa teniendo en cuenta los movimientos disponibles que se exponen en el problema.
- La manera en la que se va a ir rellenando la memoria (matriz) corresponde con ir asignando el correspondiente valor *T(i,j)* a cada casilla de la matriz.

2.2.6 *Diseño del algoritmo de cálculo de coste óptimo*

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

```
ALGORITMO T, V = Max_oro(mapa, n, m)
T <- matriz de n filas y m columnas indexadas {0...n-1} y {0...m-1}

Para cada posicion de la matriz (i, j) con \( i,j \in {0..n-1} \) hacer:
    T(i, j) = 0

T(0, 0) = Oro(0, 0)  # Inicializamos la primera casilla

Para cada fila i en {0...n-1}, hacer:
    Para cada columna j en {0...m-1}, hacer:
        Si mapa[i][j] es muro, entonces:
            T(i, j) = -infintio o INT_MIN
        Sino:
            oro = Oro(i, j)
            Si i > 0 y j > 0, entonces:
                T(i, j) = max(T(i, j), T(i-1, j-1) + oro)
            Si i > 0, entonces:
                T(i, j) = max(T(i, j), T(i-1, j) + oro)
            Si j > 0, entonces:
                T(i, j) = max(T(i, j), T(i, j-1) + oro)
```

```
21
22      V <- T(n-1, m-1) //valor maximo de bolsas de oro
23      Devolver T, V (siendo V el valor maximo de bolsas de oro)
24
25  FUNCION Oro(i, j)
26      Si mapa[i][j] == 'O', entonces:
27          devolver 1
28      Sino:
29          devolver 0
```

Vamos a explicar en detalle el algoritmo del cálculo del coste óptimo. En este creamos T que es la matriz, la cual definimos en base a un número de filas y de columnas determinadas. Posteriormente, inicializamos todas las posiciones de la matriz en 0, es decir, inicializamos la matriz. Inicializamos la casilla inicial. Luego recorremos la matriz por filas y columnas y si vemos que en la posición (i,j) de la matriz hay un muro, le asignamos el valor de $-\infty$, si no hay muro entonces vemos si podemos comparar con casillas anteriores y con ayuda de la ecuación de recurrencia, calculamos el valor de bolsas de oro para esa casilla, teniendo en cuenta las casillas anteriores. Finalmente almacenamos el valor de bolsas máximas de oro que hay en la última casilla de la matriz, es decir, en la casilla *(n-1,m-1)* y T que es la matriz resultante.

2.2.7 *Diseño del algoritmo de recuperación de la solución*

```
1       ALGORITMO ruta = Recuperar_solucion(T, mapa, n, m)
2       ruta <- lista vacia
3       i <- n-1
4       j <- m-1
5
6       Mientras i > 0 o j > 0 hacer:
7           anadir (i, j) a ruta
8           Si i > 0 y j > 0 y T(i, j) == T(i-1, j-1) + Oro(i, j), entonces:
9               i <- i-1
10              j <- j-1
11          Sino si i > 0 y T(i, j) == T(i-1, j) + Oro(i, j), entonces:
12              i <- i-1
13          Sino si j > 0 y T(i, j) == T(i, j-1) + Oro(i, j), entonces:
14              j <- j-1
15
16      anadir (0, 0) a ruta
17      invertir ruta  # La ruta se construyo desde el final al inicio
18
```

```
    Devolver ruta

FUNCION Oro(i, j)
    Si mapa[i][j] = 'O', entonces:
        devolver 1
    Sino:
        devolver 0
```

Vamos a explicar el diseño del algoritmo de recuperación de la solución. En este caso creamos una lista vacía donde vamos a almacenar la ruta del algoritmo, y vamos a comenzar por la última posición de la matriz. De manera que mientras haya posiciones por las que volver a la posición inicial vamos a ir comparando si la cantidad de oro es la misma, en el caso de que si quiere decir que esa posición de la matriz es una de las posiciones por la cual podemos pasar desde el inicio hasta el final para obtener la máxima cantidad de bolsas de oro posible. Añadimos esa posición a nuestra lista y finalmente añadimos el inicio para tener la lista completa. Invertimos la ruta para tenerla en el orden correcto y la devolvemos. La notación *'O' y 'M'*, hace referencia a si hay oro o si hay un muro, respectivamente.

La función Oro, únicamente comprueba si la posición (i,j) de la matriz esta etiquetada con *'O'* (hay oro), si lo esta devuelve uno como respuesta de que hay una bolsa de oro, y en caso contrario 0.