Parte de Algoritmos
de la asignatura de Programación
Master de Bioinformática

Programación dinámica

Web asignatura: http://dis.um.es/~domingo/algbio.html

E-mail profesor: domingo@um.es

Transparencias preparadas a partir de las del curso de Algoritmos y Estructuras de Datos II, del Grado de Ingeniería Informática y An Introduction to Bioinformatics Algorithms

Método general

- La **programación dinámica** se suele utilizar en problemas de **optimización**, donde una solución está formada por una serie de decisiones.
- Resuelve el problema original combinando las soluciones para subproblemas más pequeños.
- Se almacenan los resultados de los subproblemas en una **tabla**, calculando primero las soluciones para los problemas pequeños, y llegando hasta el tamaño deseado con un proceso iterativo.
- Con esto se pretende evitar la repetición de cálculos para problemas más pequeños.

Método general

- **Ejemplo. Cálculo de los números de Fibonacci.**
- **Con método recursivo**

```
Fibonacci (n: integer)
    Si n<2 Devolver 1
    Sino Devolver Fibonacci (n-1) + Fibonacci (n-2)
```

  - **Problema: Muchos cálculos están repetidos, tiempo de ejec. exponencial.**
  - **Solución: Calcular los valores de menor a mayor empezando por 0, e ir guardando los resultados en una tabla.**
- **Con programación dinámica.**

```
Fibonacci (n: integer)
    T[0] = 0; T[1] = 1
    para i = 2,3, ...,n
        T[i] = T[i-1] + T[i-2]
    devolver T[n]
```

  - Se utiliza la misma fórmula que en la versión anterior, pero de forma más inteligente. El tiempo de ejecución es $\Theta(n)$.

Método general

- Aspectos a definir en un algoritmo de programación dinámica:
  - **Ecuación recurrente**, para calcular la solución de los problemas grandes en función de los problemas más pequeños.
  - Determinar los **casos base**.
  - **Definir las tablas** utilizadas por el algoritmo, y cómo se rellenan.
  - Cómo se **recompone la solución** global a partir de los valores de las tablas.

Análisis de tiempos de ejecución

- El tiempo de ejecución depende de las características concretas del problema a resolver.

- En general, será de la forma:

  Tamaño de la tabla*Tiempo de rellenar cada elemento de la tabla.

- Un aspecto **importante** de los algoritmos de programación dinámica es que necesitan una tabla para almacenar los resultados parciales, que puede ocupar mucha memoria.

- Además, algunos de estos cálculos pueden ser innecesarios.

Problema del cambio de monedas

Problema: Dado un conjunto de **n** tipos de monedas, cada una con valor v_i, y con una cantidad de monedas de ese tipo c_i, y dada una cantidad **P**, encontrar el número mínimo de monedas que tenemos que usar para obtener esa cantidad.

- El algoritmo voraz es muy eficiente, pero sólo funciona en un número limitado de casos.

- **Utilizando programación dinámica:**
  - Definimos el problema en función de problemas más pequeños.
  - Determinar los valores de los casos base.
  - Definimos las tablas necesarias para almacenar los resultados de los subproblemas.
  - Establecemos una forma de rellenar las tablas y de obtener el resultado.

Problema del cambio de monedas

Definición de la ecuación recurrente:

- **Cambio (i, Q)**, el problema de calcular el número mínimo de monedas necesario para devolver una cantidad **Q**, usando los **i** primeros tipos de monedas (es decir los tipos 1...i).
- La solución de **Cambio(i, Q)** puede que utilice k monedas de tipo **i** o puede que no utilice ninguna.
  - Si no usa ninguna moneda de ese tipo:

  Cambio(i, Q) = Cambio(i - 1, Q)

  - Si usa k monedas de tipo i:

  Cambio(i, Q) = Cambio(i, Q – k*v_i) + k

- En cualquier caso, el valor será el mínimo:

  Cambio(i, Q) = $\min_{k=0,1,\ldots,\min\{Q/vi,ci\}}$ {Cambio(i-1, Q-k* v_i)+k}

Casos base: Cambio(i, Q)

- Si (i≤0) o (Q<0) entonces no existe ninguna solución al problema, y Cambio(i, Q) = +∞.
- En otro caso para cualquier **i**>0, Cambio(i, 0) = 0.

Problema del cambio de monedas

Definición de las tablas utilizadas:

- Necesitamos almacenar los resultados de todos los subproblemas.
- El problema a resolver será: Cambio (n, P).
- Por lo tanto, necesitamos una tabla de nxP, de enteros, que llamaremos **D**, siendo **D[i, j]= Cambio(i, j)**.
- **Ejemplo.** n= 3, P= 8, v= (1, 4, 6), no límite en c

D	**Cantidad a devolver**								
Monedas	0	1	2	3	4	5	6	7	8
C_1= 1									
C_2 = 4									
C_3 = 6									

Forma de rellenar las tablas:

- De arriba hacia abajo y de izquierda a derecha, aplicar la ecuación de recurrencia:

$$D[i, j] = \min_{k=0,1,\ldots,\min\{Q/vi,ci\}} \{D(i-1, Q-k^* v_i)+k\}$$

Problema del cambio de monedas

- **Algoritmo.**

Devolver-cambio (P: int; V: array [1..n] of int; C: array [1..n] of int; var D: array [1..n, 0..P] of int);

para i = 1,2,...,n
  D[i, 0] = 0
para i = 1,2,...,n
  para j = 1,2,...,P *{Tener en cuenta si el valor }*
    $D[i, j] = \min_{k=0,1,\ldots,\min\{Q/vi,ci\}} \{D(i-1, Q-k^* v_i)+k\}$ *{ cae fuera de la tabla}*

- **Ejemplo.** n= 3, P= 8, v= (1, 4, 6), no límite en c

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1= 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	1	2	3	1	2	3	4	2
C_3 = 6	0	1	2	3	1	2	1	2	2

Problema del cambio de monedas

- ¿Cómo calcular cuántas monedas de cada tipo deben usarse, es decir la solución (x_1, x_2, ..., x_n)?
- Se usa otra tabla de decisiones tomadas:

Aux	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1 = 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	0	0	0	1	1	1	1	2
C_3 = 6	0	0	0	0	0	0	1	1	0

- Algoritmo para obtener una solución:

  para i=n,n-1,...,1

  x_i=Aux[i,P]

  P=P-x_i*v_i

  → Hacer un programa Perl para este problema.

Programación dinámica en Bioinformática

- Interesa conocer la similaridad entre genes, o de varios genes con determinadas cadenas.
- La Programación Dinámica se usa para estudiar similaridad entre genes.
- Veremos el algoritmo de Mayor Subcadena Común (Longest Common Subsequence, LCS)

⇒ trabajo individual (6 puntos en la calificación final): a partir de la explicación y el programa del LCS:

seleccionar otro de los algoritmos de programación dinámica del libro,

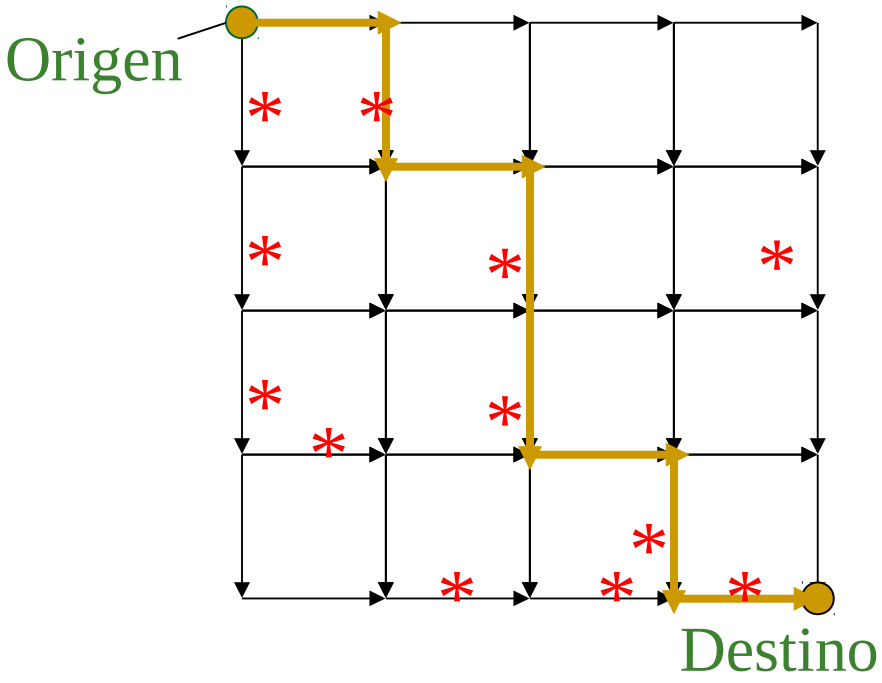
explicar su uso en bioinformática,

explicar el funcionamiento del algoritmo,

programarlo y evaluar el funcionamiento

Problema de la Distancia de Manhattan

Buscamos un camino del Origen al Destino con el que podamos visitar la mayo cantidad de atracciones (*). Solo se puede andar a la derecha y abajo.



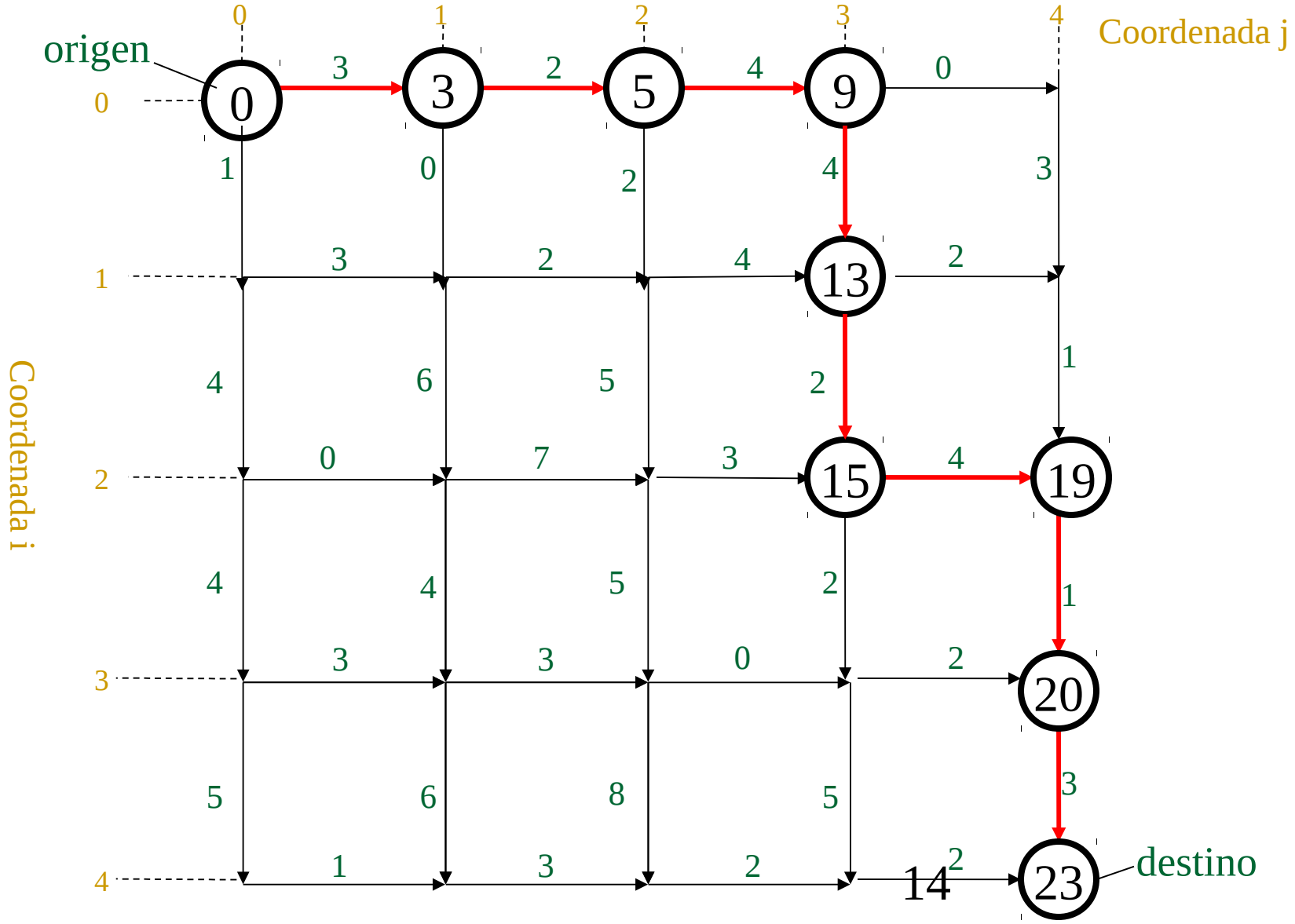
Problema de la Distancia de Manhattan

Se quiere encontrar el camino de longitud mayor en una malla con pesos

<u>Entrada</u>: Una malla con pesos ***G*** con dos vértices distinguidos, Origen y Destino

<u>Salida</u>: Un camino en ***G*** de longitud máxima para ir del Origen al Destino

Problema de la Distancia de Manhattan



Problema de la Distancia de Manhattan

→ ¿Cómo podría ser un algoritmo de avance rápido para este problema?

¿Se obtendría la solución óptima?

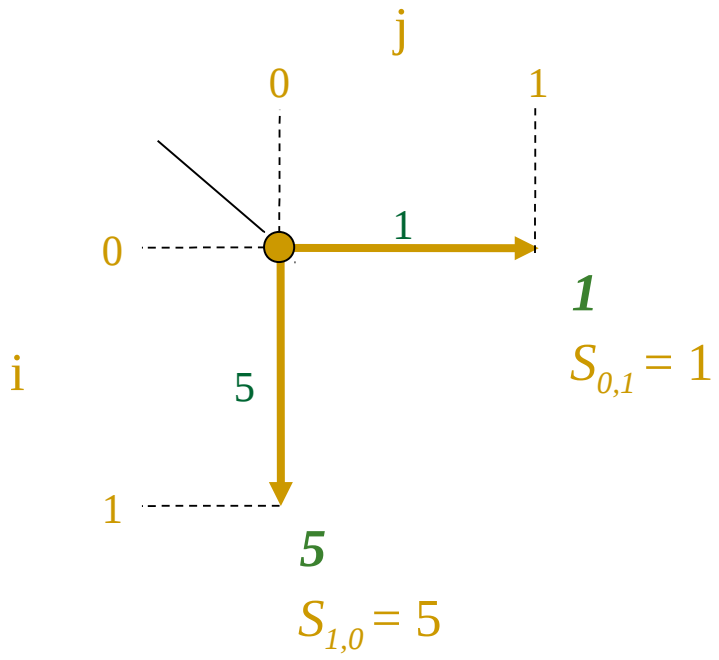
¿Qué tiempo de ejecución tendría?

→ ¿Cómo podría ser un algoritmo por backtracking para este problema?

¿Se obtendría la solución óptima?

¿Qué tiempo de ejecución tendría?

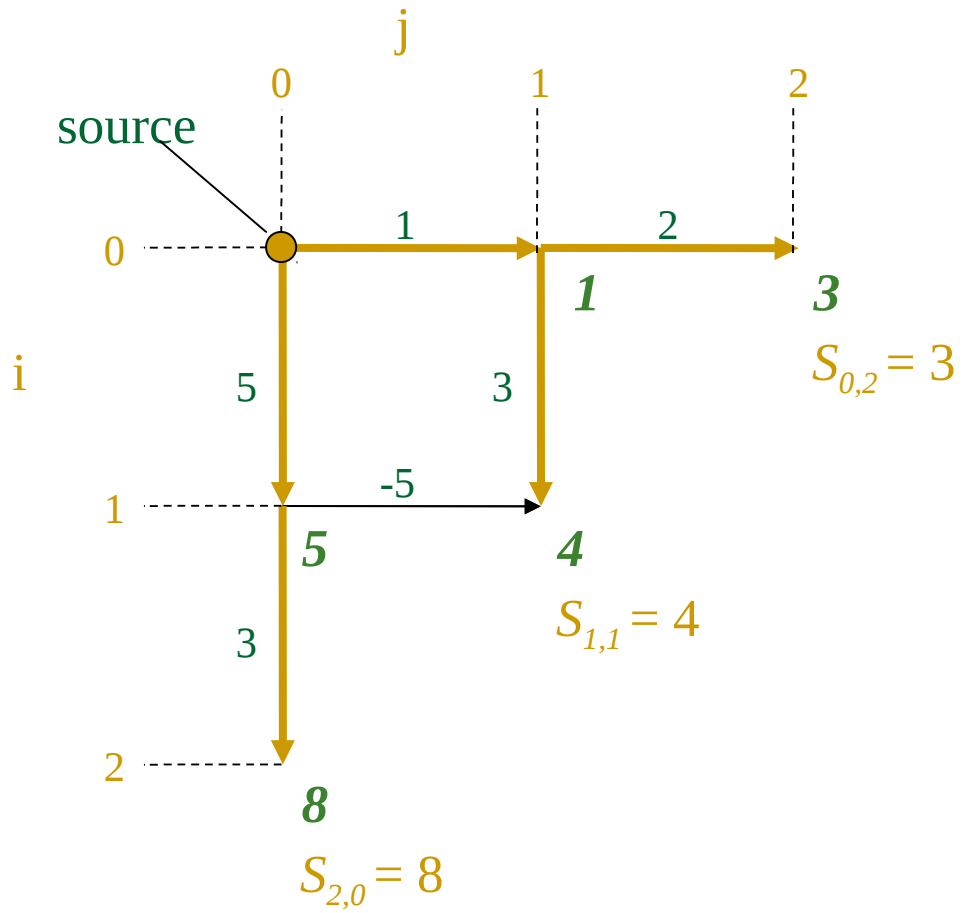
Problema de la Distancia de Manhattan



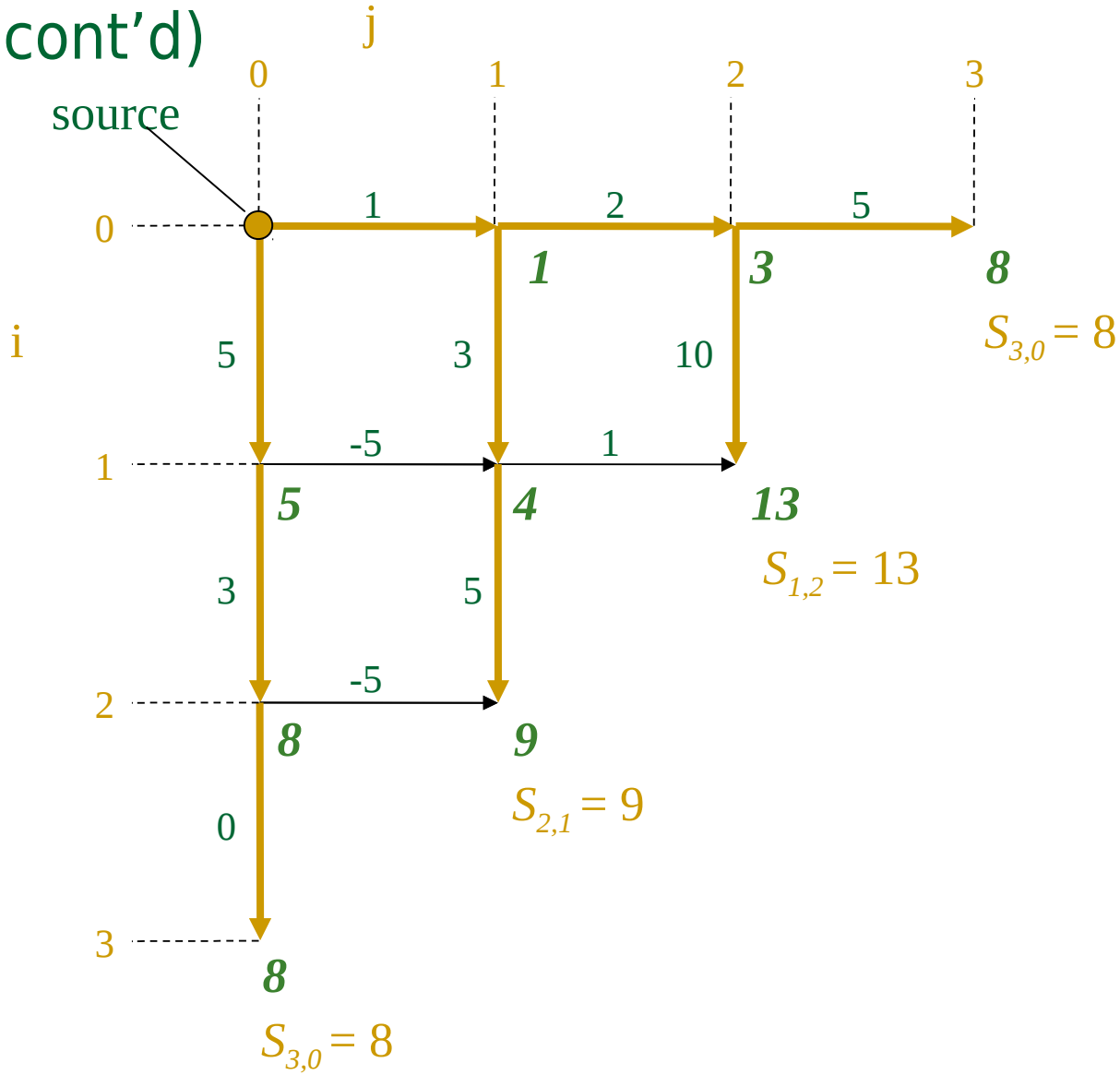
Para hacerlo por Programación Dinámica:

- Calcular el peso del camino óptimo para cada vértice de la malla
- En cada vércice el peso es el máximo del de los vértices anteriores sumado con el peso de la arista que los une

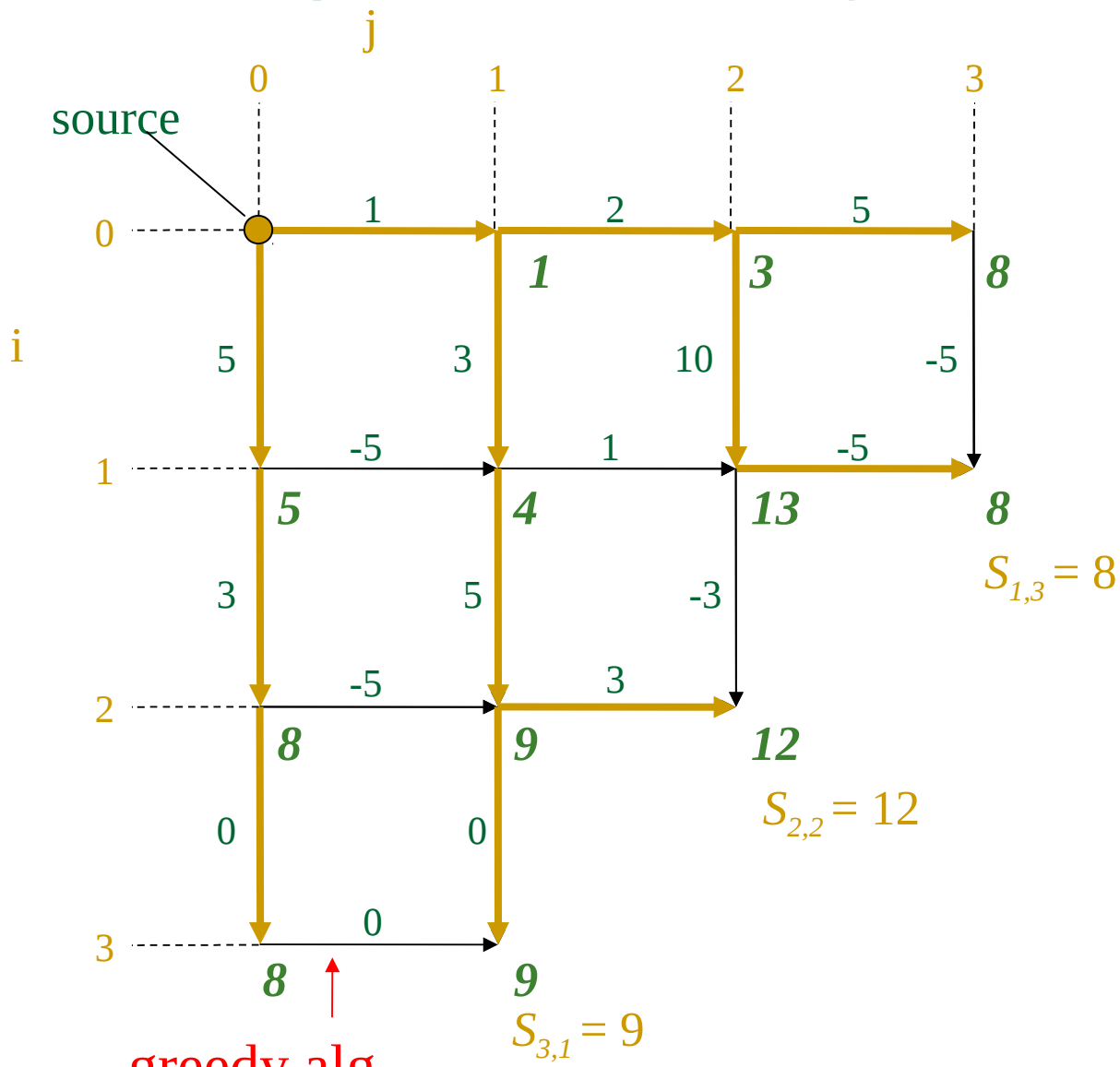
MTP: Dynamic Programming (cont'd)



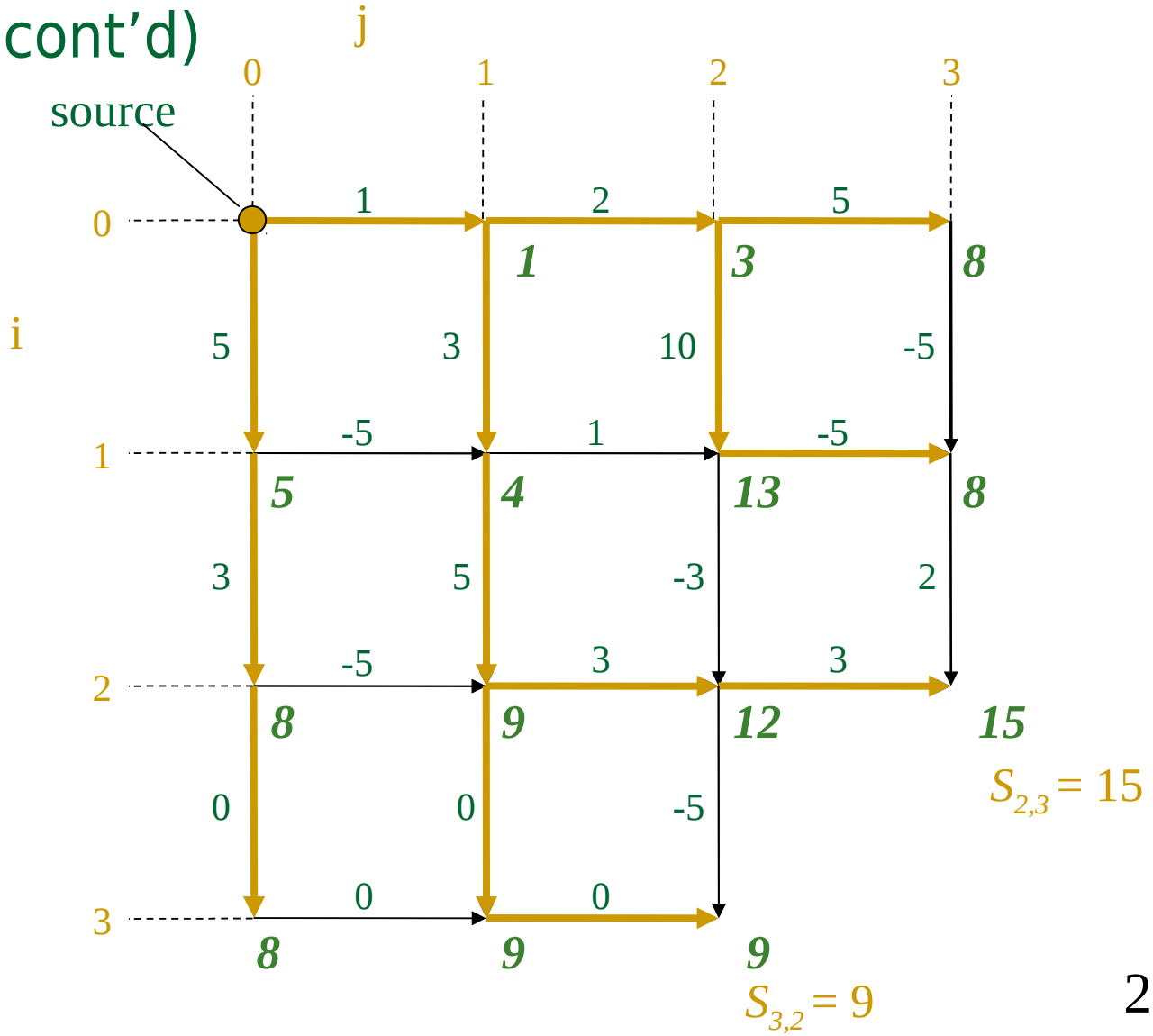
MTP: Dynamic Programming (cont'd)



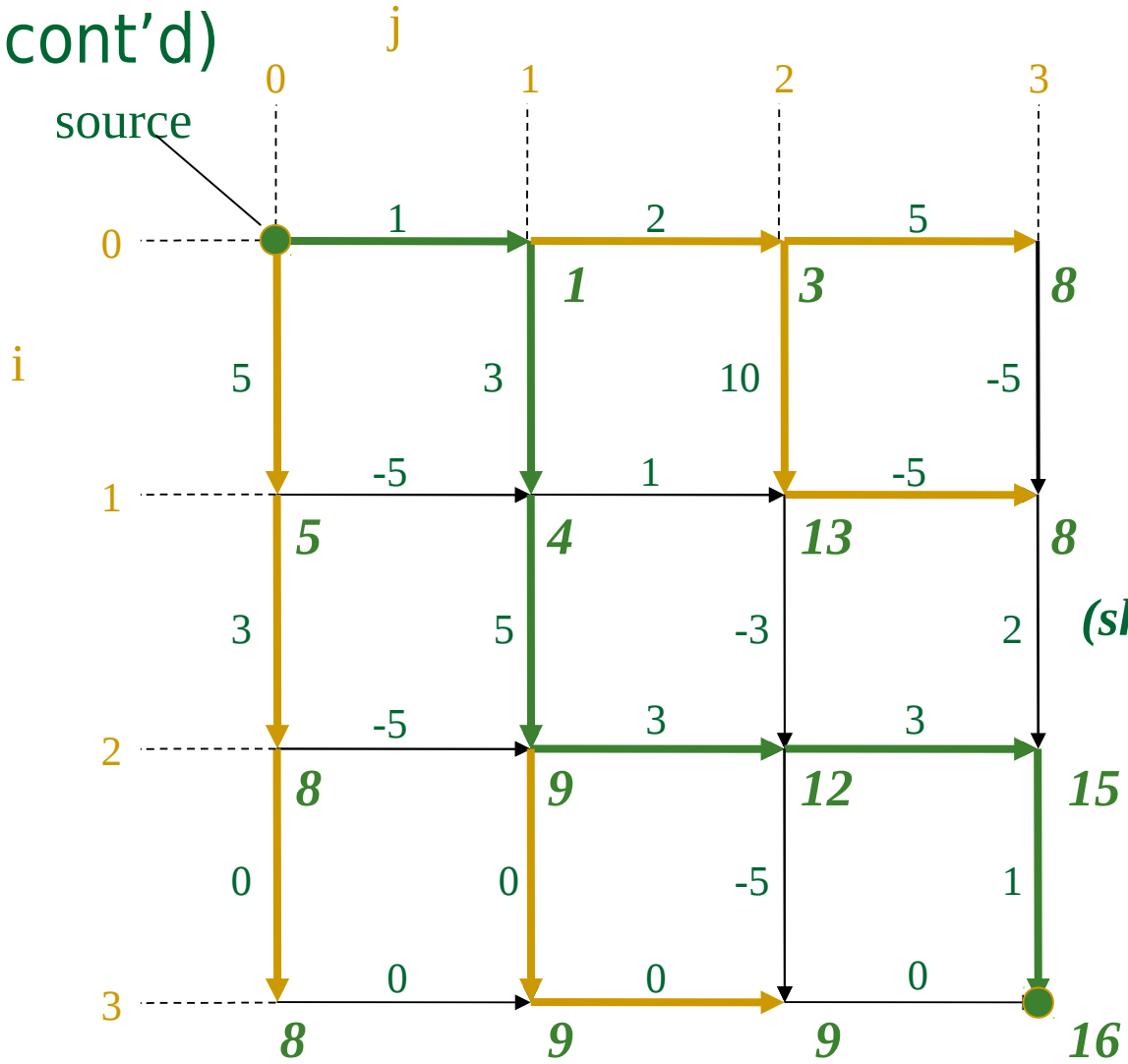
MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



Done!

(showing all back-traces)

$S_{3,3} = 16$

Problema de la Distancia de Manhattan

Se utiliza una ecuación de recurrencia para calcular el valor en cada punto:

$$s_{i,j} = \mathbf{max} \begin{cases} s_{i-1,j} + \text{peso de la arista entre } (i-1, j) \text{ y } (i, j) \\ s_{i,j-1} + \text{peso de la arista entre } (i, j-1) \text{ y } (i, j) \end{cases}$$

El tiempo de ejecución es ***n x m***

Y la solución que se obtiene es óptima

Problema de la Secuencia Común más Larga (LCS)

- Dadas dos secuencias

$$\boldsymbol{v} = v_1\, v_2 \ldots v_m \text{ y } \boldsymbol{w} = w_1\, w_2 \ldots w_n$$

- La LCS de ***v*** *y* ***w*** es una secuencia de posiciones en

$$\boldsymbol{v}\text{: } 1 \leq i_1 < i_2 < \ldots < i_t \leq m$$

y otra secuencia de posiciones en

$$\boldsymbol{w}\text{: } 1 \leq j_1 < j_2 < \ldots < j_t \leq n$$

tal que la i_t-sima letr*a d*e ***v*** es igual a la j_t-*sima* letra de ***w***
y t es máximo

Problema de la Secuencia Común más Larga (LCS)

Coord. i:	0	1	2	2	3	3	4	5	6	7	8
elements of *v*		**A**	**T**	**--**	**C**	**--**	**T**	**G**	**A**	**T**	**C**
elements of *w*		**--**	**T**	**G**	**C**	**A**	**T**	**--**	**A**	**--**	**C**
Coord. j:	0	0	1	2	3	4	5	5	6	6	7

(0,0)→(1,0)→(2,1)→(2,2)→(3,3)→(3,4)→(4,5)→(5,5)→(6,6)→(7,6)→(8,7)

Las coincidencias en rojo

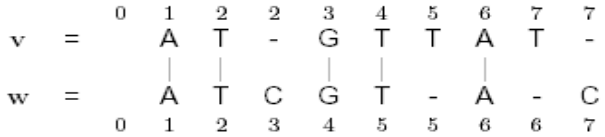
Posiciones de *v*: 2 < 3 < 4 < 6 < 8

Posiciones de *w*: 1 < 3 < 5 < 6 < 7

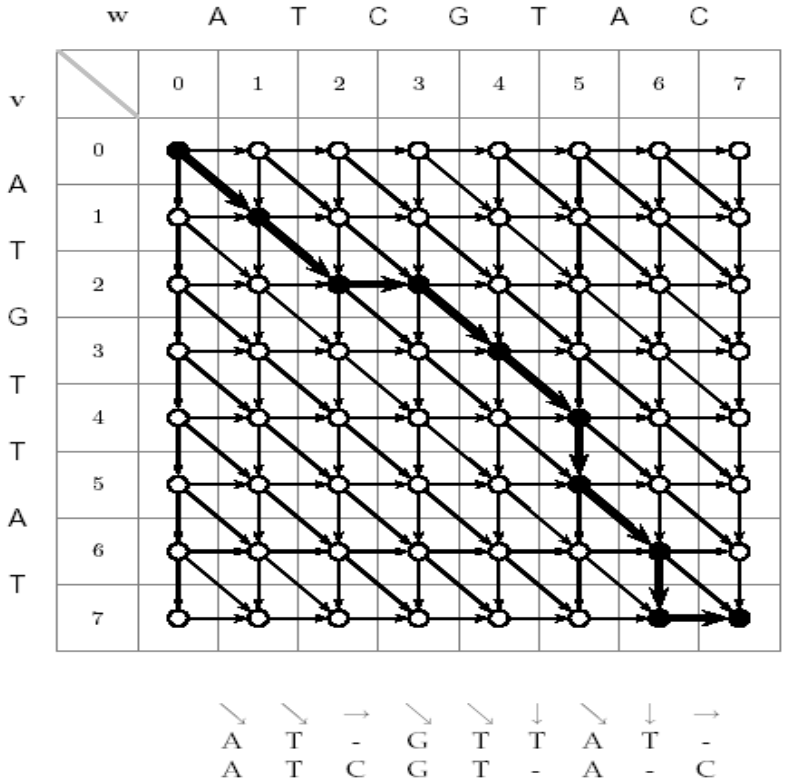
Cada subsecuencia común es un camino en la malla

Problema de la Secuencia Común más Larga (LCS)

Queremos encontrar la LCS de dos cadenas

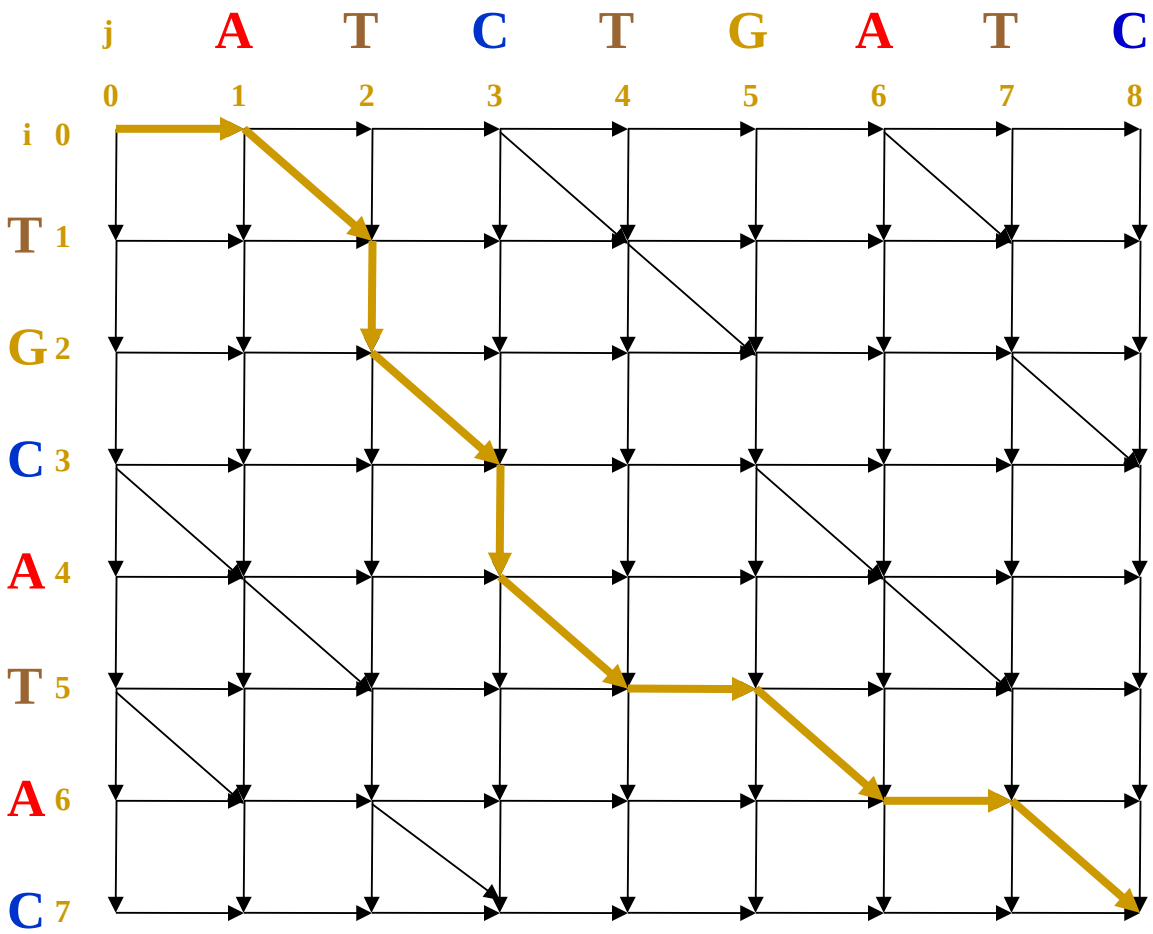


<u>Entrada</u>: Un grafo con pesos *G* con dos vértices distinguidos, Origen y Destino. Hay aristas en horizontal, vertical y diagonal, con peso 0 las verticales y horizontales y 1 las diagonales si coincide el carácter



<u>Salida</u>: El camino más largo en la malla para ir del Origen al Destino

Problema de la Secuencia Común más Larga (LCS)



Cada camino es una subsecuencia común.

Cada elemento añade un carácter a la subsecuencia común.

Problema LCS: *Encontrar un camino con el máximo número de diagonales.*

Problema de la Secuencia Común más Larga (LCS)

La ecuación de recurrencia es:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 \text{ si } v_i = w_j \end{cases}$$

→ Analizar el programa LCS.pl y explicar cómo se ha implementado el algoritmo, cómo se obtiene la subsecuencia y cuál es el coste del tiempo de ejecución.