Escritura del problema del camino más largo de vecinos en una matriz cuadrada

David Enrique Palacios García[1] Karen Sofia Coral Godoy[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{david_palacios, corallg_ksofia}@javeriana.edu.co

29 de septiembre de 2022

Resumen

En este documento se presenta la formalización del problema de encontrar el camino más largo de vecinos en una matriz cuadrada, junto con la descripción de un algoritmo de programación dinámica que lo soluciona. Además, se presenta un análisis experimental de este algoritmo. **Palabras clave:** matriz, algoritmo, formalización, experimentación, programación dinámica.

Índice

1. Introducción

Encontrar los vecinos de cada posición de una matriz suena muy trivial, pues son aquellas casillas adyacentes a esta. Esto nos permite encontrar caminos entre las casillas, pues cada una de estas tiene a su vez más vecinos. No obstante, para este taller, únicamente nos interesan las casillas que contienen un valor numérico una unidad mayor que la casilla actual, lo que podría generar caminos basados en estos valores. Es así, como se plantea el objetivo de este taller: encontrar el camino más largo de vecinos en una matriz cuadrada, donde cada valor de la casilla es único y está determinado en el rango $[1, n]$, siendo n el tamaño de la matriz cuadrada. Aquí, se busca solucionar el problema utilizando una estrategia de programación dinámica, con el objetivo de presentar: la formalización del problema (sección 2), la escritura formal del algoritmo (sección 3) y un análisis experimental del mismo (sección 4).

2. Formalización del problema

Cuando se piensa en *hallar los vecinos secuenciales de una casilla* la solución inmediata puede ser muy simplista: inocentemente, se piensa en tomar cada casilla y evaluar cuál de sus vecinos es mayor por una unidad. Sin embargo, con un poco más de reflexión, hay tres preguntas que pueden surgir:

1. ¿Cuántos vecinos tiene cada casilla?
2. ¿Existen vecinos diagonales?
3. ¿Una unidad se refiere únicamente a números naturales?

Para iniciar, recordemos que una matriz cuadrada A cuya cardinalidad es mayor a 3, cuenta con casillas de "centro", esto es, que no son adyacentes al borde de la matriz. Si imaginamos un momento una casilla de este tipo, entenderemos que tiene 4 vecinos adyacentes. Aquellas que son esquinas, cuentan con 2 vecinos, y los que están al borde de la matriz, tienen con 3. Esto es importante porque hay que verificar qué tipo de casilla se está evaluando, para saber si es posible evaluar vecinos encima, debajo, a su izquierda o derecha. Por lo que venimos hablando, vecinos diagonales no se tendrán en cuenta, únicamente verticales y horizontales. Para este problema, un vecino válido es aquel que tiene una unidad más que la casilla actual, por lo que únicamente se tendrán en cuenta números naturales entre $[1, n]$, siendo n una de las dimensiones de la matriz cuadrada.

2.1. Definición del problema de la "multiplicación matricial"

Así, el problema de encontrar la secuencia más larga de vecinos en una matriz cuadrada se define como

1. Una matriz cuadrada $A = \langle a_{1,1}, a_{1,2}, ..., a_{n,n} \rangle \in \mathbf{N^2}$ y existe una relación de igualdad y diferencia
2. Entonces, el camino más largo de vecinos adyancentes de la casilla $A_{i,j}$ es:

$$S_{i,j} = \begin{cases} 0 & ; \quad i < 1 \vee j < 1 \vee i > |A| \vee j > |A| \\ max \begin{Bmatrix} 1 + S_{i-1,j} & ; \quad A_{i,j} + 1 = A_{i-1,j} \wedge i > 1 \\ 1 + S_{i+1,j} & ; \quad A_{i,j} + 1 = A_{i+1,j} \wedge i < n \\ 1 + S_{i,j-1} & ; \quad A_{i,j} + 1 = A_{i,j-1} \wedge j > 1 \\ 1 + S_{i,j+1} & ; \quad A_{i,j} + 1 = A_{i,j+1} \wedge j < n \end{Bmatrix} & ; \quad otherwise \end{cases} \tag{1}$$

Producir una secuencia de números enteros W que indique el camino más largo de vecinos adyacentes encontrado en una matriz cuadrada.

- Entradas:
  - $A = \langle a_{1,1}, a_{1,2}, ..., a_{n,n} \rangle \in N^2 \mid a_{i,j} = v \in [1, n]$ y es único en A
- Salidas:
  - $W = \langle v \in A \rangle \mid W$ es una secuencia de números enteros que cumple la ecuación (1).

3. Algoritmo de solución

3.1. Programación dinámica- Backtracking

La idea de este algoritmo es: encontrar la longitud del camino más largo de vecinos de cada una de las casillas de la matriz, para así determinar quienes hacen parte del mismo. Para evitar cálculos repetidos, se utiliza la matriz M, que guarda la mayor longitud calculada desde esa casilla. En cuestión del *Backtracking*, se utiliza una matriz de duplas, donde se almacena la ubicación de la siguiente casilla del camino.

A pesar de que se sigue utilizando la recursión para hallar cada camino posible de cada casilla, la pila de ejecución no se llena porque únicamente se elige el vecino que cumple la restricción de ser una unidad mayor que la casilla actual. Al haber máximo un único valor que cumpla este requisito, no se evalúan todos los caminos sino el único que realmente es útil para resolver el ejercicio.

El resultado final quedará guardado en la casilla donde se tenga el mayor camino posible, creando a partir de allí el backtracking para encontrar la secuencia final.

Algoritmo 1 Calcular el camino más largo de la casilla i,j

Require: $A = \langle a_{1,1}, a_{1,2}, ..., a_{n,n} \rangle \in N^2 \mid a_{i,j} = v \in [1, n]$ y es único en A

```
1: procedure SecuenciaCrecienteAux(A, i, j, M, B)
2:     if i < 1 ∨ j < 1 ∨ i > |A| ∨ j > |A| then
3:         return 0
4:     end if
5:     if M[i][j] ≠ 0 then
6:         return M[i][j]
7:     end if
8:     q ← 1
9:     sup, inf, izq, der ← 0
10:    if i > 1 then
11:        if A[i][j] + 1 = A[i − 1][j] then
12:            sup ← 1 + SecuenciaCrecienteAux(A, i − 1, j, M, B)
13:            B[i][j] ← (i − 1, j)
14:        end if
15:    end if
16:    if i < |A| then
17:        if A[i][j] + 1 = A[i + 1][j] then
18:            inf ← 1 + SecuenciaCrecienteAux(A, i + 1, j, M, B)
19:            B[i][j] ← (i + 1, j)
20:        end if
21:    end if
22:    if j > 1 then
23:        if A[i][j] + 1 = A[i][j − 1] then
24:            izq ← 1 + SecuenciaCrecienteAux(A, i, j − 1, M, B)
25:            B[i][j] ← (i, j − 1)
26:        end if
27:    end if
28:    if j < |A| then
29:        if A[i][j] + 1 = A[i][j + 1] then
30:            der ← 1 + SecuenciaCrecienteAux(A, i, j + 1, M, B)
31:            B[i][j] ← (i, j + 1)
32:        end if
33:    end if
34:    q ← max(q, sup, inf, izq, der)
35:    M[i][j] ← q
36:    return M[i][j]
37: end procedure
```

Algoritmo 2 Obtener secuencia creciente de vecinos en matriz cuadrada

Require: $A = \langle a_{1,1}, a_{1,2}, ..., a_{n,n} \rangle \in N^2 \mid a_{i,j} = v \in [1, n]$ y es único en A

```
1: procedure SecuenciaCrecienteEnMatriz(A)
2:     M ← Matrix(A.row, A.row, unsigned int) = 0
3:     B ← Matrix(A.row, A.row, Pair(A.row, (1, 1))
4:     q ← 1
5:     for i ← 1 to |A| step 1 do
6:         for j ← 1 to |A| step 1 do
7:             B[i][j] ← (i, j)
8:         end for
9:     end for
10:    posActual ← (1, 1)
11:    for i ← 1 to |A| step 1 do
12:        for j ← 0 to |A| step 1 do
13:            k ← SecuenciaCrecienteAux(A, i, j, M, B)
14:            q ← max(q, k)
15:            if q = M[i][j] then
16:                posActual ← (i, j)
17:            end if
18:        end for
19:    end for
20:    P ← List(unsigned int)
21:    while posActual ≠ B[posActual.first][posActual.second] do
22:        append(P, A[posActual.first][posActual.second])
23:        posActual ← B[posActual.first][posActual.second]
24:    end while
25:    append(P, A[posActual.first][posActual.second])
26:    return P
27: end procedure
```

3.1.1. Análisis de complejidad

Por inspección de código: Se puede observar que, el algoritmo más complejo es $SecuenciaCrecienteEnMatriz$, con 2 ciclos anidados. Lo cierto es que por la naturaleza de la programación dinámica, se ahorran cálculos repetidos, por lo que se acelera el proceso y se busca evitar llenar la pila de ejecución del procesador. Esto resulta en una complejidad de $O(n^2)$, siendo esta una expresión cuadrada. A pesar de que la solución es un poco demorada, se debe a la complejidad del problema en sí, puesto que se necesitan calcular todos los posibles caminos de cada uno de las casillas de la matriz.

3.1.2. Invariante

La invariante al igual que la de todos los algortimos de programación dinámica es que se asegura que la tabla en cada iteración se esta completando correctamente.

4. Análisis experimental

En esta sección se presentarán los resultados del experimento para confirmar el orden de complejidad de los algoritmos presentados en la sección 3 y la precisión de la solución.

4.1. 10 tamaños de matrices

Acá se presentan los experimentos cuando el algoritmo se ejecuta tomando como párametro de entrada, números de matrices diferentes para cada iteración.

4.1.1. Protocolo

1. Definir una muestra de 10 tamaños de matrices
2. El algoritmo se ejecutará 10 veces con un parámetro de entrada diferente entre 2 y 100, que representa el tamaño de la matriz de cada iteración.
3. Se genera el gráfico necesario para realizar un análisis.

4.1.2. Experimentación

1. Se utilizó el programa *run_random_experiment* al cual le llega como parámetro un n de la muestra de 10 números que corresponden al tamaño de la matriz que se contempla en cada iteración del experimento, estos son:

   - `2,12,22,32,42,52,62,72,82,92`

   Adicionalmente, el programa teniendo en cuenta el tamaño de la matriz, completa internamente el contenido de esta de forma aleatoria cumpliendo con las restricciones definidas en secciones anteriores.

2. El programa logró finalizar sin complicaciones con todos los tamaños de matrices propuestos, el tiempo empleado en cada iteración se registra en la siguiente tabla: (ver Tabla 1).

N	tiempo de ejecución (µs)
2	10,1
12	485,6
22	2615,9
32	7433,5
42	17280,5
52	34115,1
62	54284,5
72	99640,2
82	132224
92	187268

Figura 1: DataExperiment

3. Se obtuvo las 10 secuencias crecientes de cada matriz cuadrada planteada con el tiempo de ejecución que se empleo para llegar a dicha solución, logrando así crear la siguiente gráfica: (ver Figura 1)

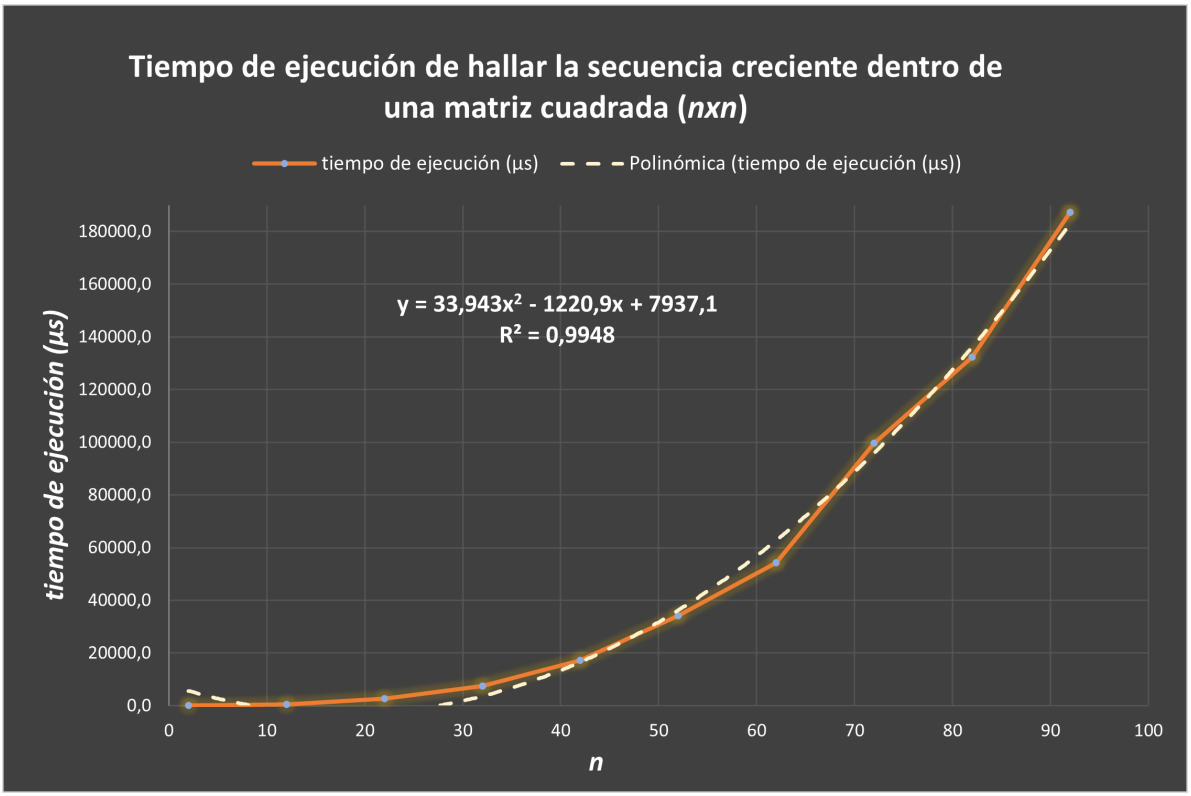

Figura 2: Tiempo de ejecución del algoritmo respecto a las dimensiones de la matriz

4.1.3. Análisis del experimento

Como se expresó previamente, el algoritmo expuesto tiene una complejidad de $O(n^2)$, sin embargo, en términos de tiempo de ejecución, es un proceso que llega a tomar mucho más tiempo cuando las dimensiones de la matriz es mayor. Esto pudo confirmarse, utilizando herramientas de Microsoft Excel, en donde se aplicó una regresión polinomial, hallando la línea de tendencia del algoritmo, su ecuación y valor R^2 que estaba muy cercano a 1, lo cual indica que se empleó el modelo más preciso, y por ende se reafirma la complejidad algorítmica de la solución presentada.

Por otro lado, para este experimento, se realizó la una prueba de escritorio de la iteración con n más pequeño para poder tener una comparación o evaluación del algoritmo. Los resultados se expresan a continuación:

- `Iteración: 1`
- `Tamaño de la matriz: 2x2`
- `A:`

4	3
1	2

- `Solución- : : [1, 2, 3, 4]`

Con esta iteración se puede validar que el algoritmo funciona, ya que se hallo la secuencia creciente de mayor longitud teniendo en cuenta las restricciones del diseño del algoritmo.

5. Conclusiones

El experimento fue útil para concluir:

1. Al aumentar las dimensiones de la matriz y por ende la cantidad de elementos de esta, los caminos que debe evaluar y el tiempo de ejecución también aumentan.

2. Para plantear una solución de programación dinámica es importante considerar los parámetros de entrada, como en el caso de hallar la secuencia creciente más larga dentro de una matriz cuadrada en donde, posiblemente usar un algoritmo inocente llenaria la pila de ejecución al experimentar con entradas de valores grandes.

3. La programación dinámica se puede aplicar en los problemas de optimización ya que se basa en la idea de recursividad. En esta, se resuelven problemas descomponiéndolos en subproblemas similares más pequeños y usando dichas soluciones parciales de los subproblemas para llegar a una respuesta final. Cada subproblema se resuelve sólo una vez y su resultado es almacenado, evitando el trabajo de recalcular cada vez que el subproblema es nuevamente encontrado.

4. En general, el tiempo de un algoritmo de programación dinámica depende del número de subproblemas y del número de opciones a revisar para cada subproblema.