Camino Mínimo en una Matriz: Comparación de Paradigmas Algorítmicos

Curso: Análisis y Diseño de Algoritmos

Universidad: EAFIT

Estudiante: Samuel Moncada Mejía

Cédula: 1022003803

Fecha: noviembre de 2025

Tabla de Contenidos

---

1. Introducción

El problema del camino de costo mínimo en una matriz pertenece al campo de la optimización combinatoria y tiene aplicaciones relevantes en áreas como la robótica, los videojuegos, la logística y las redes de comunicación. El objetivo principal consiste en comprender cuándo y por qué ciertos enfoques algorítmicos resultan más apropiados que otros, considerando aspectos como la complejidad temporal, el uso de memoria y las características específicas del problema.

---

2. Descripción del Problema

2.1 Definición Formal

Dada una matriz "A" de dimensiones n x n, donde cada elemento A[i][j] representa un costo no negativo, se debe encontrar el camino de costo mínimo desde la celda inicial (0,0) hasta la celda final (n−1, n−1). Solo se permiten movimientos hacia abajo o hacia la derecha, y el costo de un camino es la suma de los valores de todas las celdas visitadas. El objetivo es hallar la ruta cuyo costo total sea el menor posible.

Este problema presenta varias propiedades fundamentales. En primer lugar, posee subestructura óptima, ya que el camino óptimo hacia una celda (i, j) incluye los caminos óptimos hacia (i−1, j) y (i, j−1). En segundo lugar, existen subproblemas superpuestos, debido a que distintos caminos comparten celdas intermedias. Además, el espacio de búsqueda crece combinatoriamente, con un total de C(2n-2, n-1) posibles rutas.

---

3. Análisis Teórico

3.1 Paradigmas Algorítmicos

Backtracking

Técnica de búsqueda exhaustiva que explora sistemáticamente el espacio de soluciones mediante recursión, descartando ramas que no pueden conducir a soluciones óptimas

3.1.2 Algoritmos Greedy

Estrategia que toma decisiones localmente óptimas en cada paso, esperando llegar a un óptimo global. Funciona cuando el problema cumple la propiedad de elección greedy y subestructura óptima

3.1.3Programación Dinámica

Técnica que resuelve problemas dividiéndolos en subproblemas superpuestos, almacenando sus soluciones para evitar recalcularlas. Requiere subestructura óptima y subproblemas superpuestos

3.2 Algoritmo de Dijkstra

Algoritmo greedy para caminos mínimos en grafos con pesos no negativos. Mantiene un conjunto de nodos visitados y expande siempre el nodo con menor distancia acumulada

Condiciones:

- Pesos no negativos
- Grafo dirigido o no dirigido
- Garantiza solución óptima

4. Paradigmas Implementados

4.1 Backtracking con Poda (DFS)

Se explora recursivamente todos los caminos posibles desde (0,0) hasta (n-1, n-1), podando ramas cuyo costo parcial ya supera el mejor costo encontrado.

```
BACKTRACKING(matriz A, dimensión n):
    mejor_costo ← +∞
    DFS(i, j, costo_acumulado):
        // Poda por bound
        SI costo_acumulado ≥ mejor_costo:
            RETORNAR
        // Caso base: llegamos al destino
        SI (i = n-1) Y (j = n-1):
```

```
        mejor_costo ← min(mejor_costo, costo_acumulado + A[i][j])
        RETORNAR
    // Moverse hacia abajo
    SI i+1 < n:
        DFS(i+1, j, costo_acumulado + A[i][j])
    // Moverse hacia la derecha
    SI j+1 < n:
        DFS(i, j+1, costo_acumulado + A[i][j])

DFS(0, 0, 0)
RETORNAR mejor_costo
```

4.2 Greedy: Algoritmo de Dijkstra

Utiliza una cola de prioridad para expandir siempre la celda con menor costo acumulado desde el origen, así garantiza que cuando se alcanza el destino, se halle el camino óptimo.

```
DIJKSTRA(matriz A, dimensión n):
    dist[i][j] ← +∞ para todo i,j
    dist[0][0] ← A[0][0]

    cola_prioridad Q ← {(A[0][0], (0,0))}
    MIENTRAS Q no esté vacía:
        (d, (i,j)) ← extraer_minimo(Q)
        // Ya procesamos este nodo con mejor distancia
        SI d ≠ dist[i][j]:
            CONTINUAR
        // Llegamos al destino
```

```
        SI (i = n-1) Y (j = n-1):
            RETORNAR d
        // Explorar vecinos (abajo y derecha)
        PARA cada vecino (ni, nj) en {(i+1,j), (i,j+1)}:
            SI (ni,nj) está dentro de límites:
                nueva_dist ← d + A[ni][nj]

                SI nueva_dist < dist[ni][nj]:
                    dist[ni][nj] ← nueva_dist
                    insertar(Q, (nueva_dist, (ni,nj)))

    RETORNAR dist[n-1][n-1]
```

4.3 Programación Dinámica (Bottom-Up)

Se construye una tabla dp[i][j] donde cada entrada almacena el costo mínimo para llegar desde (0,0) hasta (i,j). Llena la tabla iterativamente usando la recurrencia.

```
PD_BOTTOM_UP(matriz A, dimensión n):
    dp[i][j] ← +∞ para todo i,j
    dp[0][0] ← A[0][0]

    PARA i ← 0 hasta n-1:
        PARA j ← 0 hasta n-1:
            SI (i,j) ≠ (0,0):
                mejor ← +∞
                // Venir desde arriba
                SI i > 0:
```

```
            mejor ← min(mejor, dp[i-1][j])
        // Venir desde la izquierda
        SI j > 0:
            mejor ← min(mejor, dp[i][j-1])
        dp[i][j] ← A[i][j] + mejor

RETORNAR dp[n-1][n-1]
```

5. Análisis de Complejidad

5.1 Análisis Temporal

5.1.1 Backtracking

Con Poda:

Mejor caso: $O(n^2)$ si la poda es muy efectiva

Caso promedio: depende de la distribución de los costos

Peor caso: O(2^(2n)) cuando no hay poda efectiva

Recurrencia:

T(i,j) = T(i+1,j) + T(i,j+1) + O(1)

T(n-1,n-1) = O(1)

Esta recurrencia tiene solución exponencial O(2^n)

5.1.2 Dijkstra

Nodos en el grafo: n^2

Aristas por nodo: 2 (abajo, derecha) en promedio

Aristas totales: $2n^2$

Complejidad total:

$T(n) = O(V \log V + E \log V)$

$= O(n^2 \log n^2 + 2n^2 \log n^2)$

$= O(n^2 \log n)$

5.1.3 Programación Dinámica

Complejidad:

$T(n) = \Sigma(i=0 \text{ a } n-1) \text{ y } \Sigma(j=0 \text{ a } n-1)\ O(1)$

$= n \times n \times O(1)$

$= O(n^2)$

Recurrencia:

$T(i, j) = T(i-1, j) + T(i, j-1) + O(1)$

5.2 Análisis Espacial

Paradigma	Espacio	Componentes
BackTracking	O(n)	Pila de recursión
Dijkstra	$O(n^2)$	Matriz dist[n][n] + Cola de prioridad
Programación Dinámica	$O(n^2)$	Tabla dp[n][n]

Conclusión espacial:

Backtracking: Más eficiente en memoria, pero a costa de tiempo exponencial

Dijkstra: Requiere una estructura adicional (heap)

Programación Dinámica: Puede optimizarse a espacio lineal sin perder eficiencia temporal

5.3 Tabla Comparativa de Complejidades

Algoritmo	Tiempo (Peor Caso)	Tiempo (Mejor Caso)	Espacio	Garantiza Óptimo
BackTracking	$O(2^{2n})$	$O(n^2)$	O(n)	Si
Dijkstra	$O(n^2 \log n)$	$O(n^2 \log n)$	$O(n^2)$	Si
Programación Dinámica	$O(n^2)$	$O(n^2)$	$O(n^2)$ o O(n)	Si

---

6. Metodología Experimental

6.1 Casos de Prueba

Se diseñaron 3 casos de prueba con diferentes características:

1. Caso 0 (n=3): Matriz pequeña básica
2. Caso 1 (n=4): Matriz con trampa (costo alto en centro)
3. Caso 2 (n=6): Matriz con camino claro (poda efectiva)

---

7. Resultados Experimentales

7.1 Tabla de Resultados Completa

Caso	N	Algoritmo	Costo	Tiempo (ms)
0	3	BackTracking Dijsktra PD	7 7 7	0 0 0
1	4	BackTracking Dijsktra PD	10 10 10	0 0 0

2	6	BackTracking Dijsktra PD	11 11 11	0 0 0

Observaciones:

Todos los algoritmos encuentran el mismo costo óptimo

Los tiempos en milisegundos son tan pequeños que aparecen en 0

8. Conclusiones

8.1 Conclusiones Específicas por Paradigma

Backtracking

Ventajas: Garantiza solución óptima, bajos tiempos y es simple de implementar

Desventajas: El tiempo puede aumentar exponencialmente

Dijkstra (Greedy)

Ventajas: Garantiza óptimo con pesos no negativos, es versátil para grafos generales y es eficiente

Desventajas: Se puede generar un Overhead en la cola de prioridad

---

Programación Dinámica

Ventajas: Máxima eficiencia temporal $O(n^2)$, es un código simple que aprovecha la estructura del problema

Desventajas: Requiere memoria $O(n^2)$ aunque es optimizable a $O(n)$, solo aplicable a problemas con recurrencia clara

---

8.2 Conclusiones Generales

Podemos concluir que no existe un paradigma superior, ya que la elección del enfoque depende directamente del tamaño de la instancia, la estructura del problema, las restricciones de tiempo y memoria, y los requisitos de implementación. La teoría de complejidad se ve confirmada en la práctica: el algoritmo de Backtracking evidencia un crecimiento exponencial, mientras que Dijkstra y la Programación Dinámica mantienen un comportamiento. Entre estos dos últimos, la Programación Dinámica resulta más eficiente para este problema específico, debido a su menor tiempo de ejecución y simplicidad en la implementación.

9. Referencias Bibliográficas

Bellman, R. (1957). *Programación dinámica*. Princeton University Press.

GeeksforGeeks. (2024). *Camino más corto en una cuadrícula usando el algoritmo de Dijkstra*

Brassard, G. y Bratley, P. (1996). *Fundamentos de algoritmos*. Prentice Hall.

10. Uso de IA

Para este trabajo utilice la inteligencia artificial con fines de compresión de conceptos y teoría acerca de mi problema, además también la utilice para realizar partes complicadas del código, los pseudocódigos y también para el análisis de complejidades para que me proporcionara como debía ser las soluciones.