Informe Práctica 3:
Búsqueda con retroceso

Jorge Soria Romeo
872016

Alberto Francisco Solaz García
873373

Asignatura: Algoritmia Básica
Grupo: Jb

Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Marzo 2025

1. Introducción

En la práctica 3 diseñamos e implementamos un algoritmo de backtracking para realizar una búsqueda de caminos sin repetir casillas, con movimientos básicos (arriba, abajo, izquierda y derecha) y restricciones adicionales.

2. Algoritmo

Comenzamos implementando una estructura de datos personalizada, análoga al `std::bitset` de la C++ Standard Library, pero con tamaño definido en tiempo de ejecución (no es `constexpr`). Sirve para almacenar eficientemente los caminos explorados por el algoritmo, minimizando los accesos a memoria.

Nuestro algoritmo optimiza el proceso de poda mediante 2 objetivos:

- Maximizar el número de caminos podados
- Minimizar el cómputo por ejecución recursiva

Para lograrlo, ordenamos los predicados de poda en orden decreciente según su probabilidad de cumplimiento:

Predicado 6: Verificación de casillas no aisladas. Experimentalmente, en nuestros casos de prueba, este predicado tiene mayor probabilidad de activarse que el Predicado 5. Su implementación es costosa computacionalmente (es la única poda cuyo coste no es de $\mathcal{O}(1)$) pero lo compensa al ser un método de poda muy agresivo.

Predicado 5: Validación de alcanzabilidad del siguiente registro con los pasos restantes. Cuanto mayor sea el tablero (menor sea el número de puntos de registros en relación a él) mejores resultados dará permutar el Predicado 5 con el Predicado 6.

Predicado 7: Comprobar que no se haya pasado de pasos para el siguiente registro Junto con el Predicado 6, son la primera línea de comprobación porque son aquellos que se pueden cumplir desde cualquier casilla del tablero.

Predicado 3: Prevención de llegada prematura a checkpoints o meta. Se verifica después de los predicados 5, 6 y 7 por aplicarse en una cantidad de casillas despreciable en relación al total.

Predicado 4: Verificación de llegada exacta al checkpoint. Posicionado después del Predicado 5 porque es más probable llegar temprano que exactamente en el paso necesario.

Objetivo: Comprobación de si la casilla actual es la casilla final. Este predicado tiene prioridad sobre los 1 y 2 porque toda casilla que cumple los predicados anteriores debe verificar si es la final (condición universal), pero la casilla final puede saltarse estos predicados.

Predicados 1 y 2: Validación de movimientos. Ambas comprueban la validez de los movimientos y por las operaciones booleanas cortocircuitadas de C++ es más eficiente comprobarlas juntas.

3. Complejidad temporal

Una búsqueda por fuerza bruta tendría un coste asintótico temporal de $\mathcal{O}(3^{m \times n})$ para un espacio de dimensiones $m \cdot n$.

- Factor de ramificación 3 porque en cada movimiento YuMi podría desplazarse a 3 celdas (4 movimientos, el inverso del que acaba de hacer no es válido)
- Profundidad de exploración $m \cdot n$ porque tiene que cruzar todas las celdas del espacio

Mediante backtracking no podemos reducir el coste asintótico porque el espacio de búsqueda es el mismo. Pese a ello, el coste medio se reduce en función de la calidad de las podas.

4. Complejidad espacial

En el peor caso (existencia de al menos un camino válido), la profundidad de la pila de llamadas recursivas alcanza $\mathcal{O}(m \cdot n)$, ya que el algoritmo debe explorar todas las $m \cdot n$ celdas del tablero. El tamaño de la estructura de datos auxiliar (`BitSet`) también es teóricamente $\mathcal{O}(m \cdot n)$, pero gracias a su diseño optimizado (1 bit por celda frente a los 8 bits de un `bool`), su tamaño real es despreciable frente al coste dominante de la pila de recursión.

5. Pruebas

Se nos ha proporcionado un fichero de entrada con un conjunto de 10 pruebas, cada una de las cuales indica el número de filas, el número de columnas y las posiciones de los registros.

Se ha evaluado el número de recorridos completos que puede realizar YuMi (pasar por todos los checkpoints en el paso preciso y finalizar en la posición de meta) usando una estrategia de ***backtracking* directa** y el método ***meet-in-the-middle***.

El número de recorridos hallados para cada estrategia es el mismo. Sin embargo, la búsqueda de backtracking directa ofrece menor tiempo de ejecución que la estrategia meet-in-the-middle.

Para tableros pequeños meet-in-the-middle no proporciona una mejora significativa con respecto al backtracking directo, posiblemente por la necesidad de combinar los recorridos encontrados.

Además hemos creado un fichero extra con las siguientes pruebas:

1. Caso de prueba con ningún camino válido
   a) Ambos algoritmos van muy deprisa, demostrando que la poda es eficiente y la elección de predicados adecuada.
2. Caso de prueba cuya complejidad temporal es elevada
   a) La complejidad temporal es de aproximadamente $O(3^{m \cdot n})$. Ninguno de los dos algoritmos termina.

Resultados de `pruebas.txt` ejecutadas en `lab000`:

Cuadro 1: Comparación de métodos: Directo vs Meet-in-the-middle

	Tiempo (ms)		
Caminos encontrados	Directo	Meet-in-the-middle	Ratio (MITM/Directo)
1	0.01515	2.44310	161.26
2	0.01663	1.68887	101.53
0	0.00550	1.52498	277.27
2	0.02896	1.94782	67.26
24	0.49020	8.99290	18.35
6	0.89638	15.28946	17.06
67	15.84947	1486.05635	93.76
626	86.57894	29309.56276	338.53
1334	681.37272	29293.58425	42.99
914	328.10656	22502.83502	68.58

Cuadro 2: Nodos generados en búsqueda directa

Nodos generados									
9	83	20	116	2,061	3,672	46,634	306,486	2,614,995	1,146,796

6. Bola extra: Meet-in-the-middle

Para implementar búsqueda *meet-in-the-middle* hemos dividido el espacio de búsqueda en 2 partes de misma complejidad:

- La 1ª mitad busca desde el inicio (0, 0), hasta el 2º registro
- La 2ª mitad busca desde el el 2º registro hasta el final (0, 1)

En lugar de devolver directamente el número de caminos, cada subproceso genera un vector con todos los caminos encontrados en su respectivo subespacio. Al terminar, realizamos una búsqueda de combinaciones válidas entre los resultados de ambas mitades mediante el siguiente proceso de hasheo:

1. Almacenamos los caminos del segundo subespacio en un diccionario indexado por su representación única (codificada como cadena de texto)
2. Para cada camino del primer subespacio, verificamos si existe en el diccionario un camino complementario cuyo índice coincida con la codificación requerida para completar la solución

Paralelizamos tanto la búsqueda de caminos como la búsqueda de permutaciones válidas mediante hilos.

7. Conclusiones

Como principal conclusión destacamos la relevancia de la correcta elección y ordenación de predicados para la poda en ambos algoritmos.

La elección de buenos predicados y su ordenación nos ha permitido recortar significativamente el espacio de búsqueda y así recorrer menos caminos.

Se ha explorado la idea de dividir el problema en subproblemas y luego combinar los resultados (meet-in-the-middle). Puede ser útil para tableros de gran tamaño, aunque para los casos de prueba dados no ofrece una ventaja significativa.

Además, se ha comprobado la importancia de la elección de una buena estructura de datos (bitset) para el problema y la paralelización para mejorar en términos de eficiencia temporal.

A. Anexo: Diagrama de flujo meet-in-the-middle

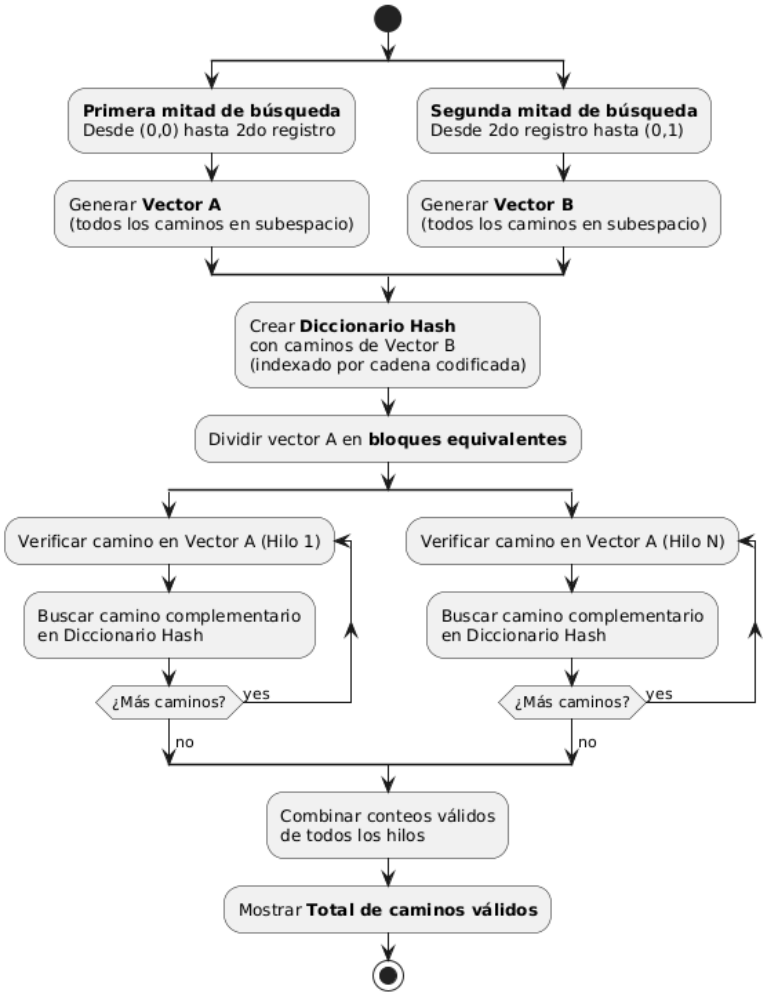

Figura 1: En este diagrama se representa el flujo de ejecución del algoritmo de backtracking meet-in-the-middle y su paralelización