ALGORITMIA BÁSICA

Práctica 3 - Backtracking

Nicolás de Rivas Morillo (843740)
Cristina Embid Martínez (842102)

Resumen

En esta práctica se ha realizado un algoritmo de *backtracking* para resolver un problema de caminos hamiltonianos. Para ello se han planteado unos predicados de descarte para aumentar la eficiencia del algoritmo de backtracking (Tree pruning). Además, se ha implementado la técnica de *meet-in-the-middle* para mejorar aún más la eficiencia del mismo.

Implementación

La implementación se ha realizado en C++ planteando 3 clases:

1. **Yumi**: Representa el robot que resuelve el problema. Tiene varios atributos de control.
2. **Tablero** (Grafo): Representa un tablero donde Yumi puede moverse. Es una matriz de casillas.
3. **Casilla** (Nodo): Representa una casilla del tablero. Cuenta con 3 naturales los cuales representan número de enlaces de entrada, salida y dobles de esa casilla, además tiene dos booleanos, uno de si se ha visitado o no y otro que se usa para marcar la visita de una búsqueda *DFS*.

Planteamos ahora los predicados para el descarte de soluciones:

1. No salir del tablero
2. No alcanzar un checkpoint demasiado pronto
3. Terminar en el fin
4. No visitar una casilla ya visitada
5. Poder llegar al siguiente checkpoint a tiempo
6. No dejar casillas no recorribles
7. Alcanzar los checkpoints
8. No dejar casillas no alcanzables

Los predicados 1 y 4 vienen dados por condicionales en nuestra llamada recursiva, el predicado 3 viene dado por un condicional en nuestro criterio de conteo de soluciones, los criterios 2, 5 y 7 se pueden comprobar directamente teniendo en cuenta que la distancia mínima entre (a_1, a_2) y (b_1, b_2) es $|a_1 - b_1| + |a_2 - b_2|$.

El predicado 6 lo hemos implementado mediante 3 atributos en la clase Casilla que guardan los enlaces que tienen con el resto de casillas, en cualquier momento, toda casilla no visitada debe poder visitarse, es decir, debe poder entrar y salir. Para ello necesitamos que se cumpla uno de los siguientes casos:

Caso 1	Hay entrada y enlace doble	**Caso 2**	Hay dos enlaces dobles
Caso 3	Hay entrada y salida	**Caso 4**	Hay enlace doble y salida
Caso 5	Yumi está ahí y hay salida	**Caso 6**	Es el fin y tiene entradas
Caso 7	Es el fin y Yumi está en el fin		

Así, si ninguno de esos casos se da en alguna casilla no visitada en algún momento, la posición actual no tiene soluciones. Notemos encima que no es necesario recalcular todas las casillas en cada iteración, únicamente las adyacentes a la última visitada.
El predicado 8 lo hemos implementado mediante una búsqueda *DFS* en el tablero comenzando en el final, si no visitamos todas las casilla que quedan, es decir, $m \cdot n - pasos$, entonces tenemos una desconexión del grafo y por lo tanto no tiene soluciones.

Además hemos aplicado la técnica de *meet-in-the-middle*, para ello hemos creado una clase nueva, la cual lanza dos Yumis, una desde el inicio hasta el checkpoint medio y otra desde el checkpoint medio hasta el final. Para esta segunda Yumi se nos plantean ciertos casos por los que no podemos aplicar el predicado 6 ya que el primer Yumi puede haber pasado por ahí y que la solución sea válida. Además después de calcular todos los caminos hay que mezclarlos, esto tiene un coste elevado ya que debemos recorrer las dos listas de soluciones y comparar las matrices recorriendolas, es decir, 4 fors anidados.
También hemos planteado una optimización para la mezcla de soluciones parciales, no hemos tenido oportunidad de implementarla pero la explicamos a continuación.
Planteamos usar un hashmap para almacenar las soluciones parciales. Usar una matriz de booleanos como clave y un natural como valor, así todas las matrices que sean iguales pero Yumi haya llegado por caminos diferentes estarán bajo la misma llave y como valor asociado tendrán el número de caminos por los que se puede llegar, para buscar bastaría con usar como clave el inverso de la solución parcial que queremos completar. Esto plantea ventajas de búsqueda, de los 4 bucles anidados podríamos reducir a 3 no anidados.

Análisis de las pruebas realizadas

Todas las pruebas han sido ejecutadas en un macbook air 2020 (M1).
Las primeras pruebas realizadas han sido las adjuntadas con el enunciado, hemos podido así verificar los resultados que venían en el propio enunciado, viendo el funcionamiento básico correcto del sistema. Los resultados han sido los siguientes: 1, 2, 0, 2, 24, 6, 67, 626, 1334 y 914.
Además hemos realizado también pruebas generadas por nosotros. (C: checkpoint, F: fin e I: Inicio)

C	C
I-C	F

Prueba 1

	C	
C		C
I	F	

Prueba 2

C	C	
		C
I	F	

Prueba 3

La primera es el tablero más básico que se nos ha ocurrido, con una única solución.

Como segunda prueba planteamos un tablero 3x3, estos nunca tienen solución, independientemente de dónde estén colocados los checkpoints, aún así, están puestos en lugares donde Yumi llega a tiempo.
Como tercera prueba tenemos el mismo tablero que en la anterior pero esta vez con los checkpoints en posiciones a las cuales Yumi no llegaría, este tablero lo descartamos antes siquiera de empezar a solucionar.

C		C		
I	F	C		

Prueba 4

	C			
			C	
			C	
I	F			

Prueba 5

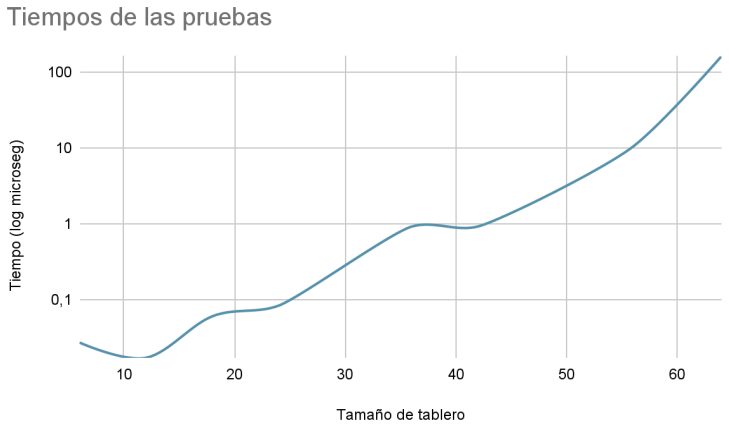
				C	
	C				
				C	
I	F				

Prueba 6

Estas últimas 3 pruebas las hemos calculado a mano y por lo tanto conocemos los resultados.

Análisis de tiempos y eficiencia

Para analizar los tiempos hemos utilizado la librería *ctime*. El tiempo se inicia antes de llamar al constructor e incluye la llamada a la función que resuelve el problema en cuestión. Tras su ejecución se muestra el tiempo como se especifica en el documento de la práctica. Se observa que para problemas pequeños, se obtienen tiempos de ejecución insignificantes (del orden de microsegundos).
A priori a fuerza bruta se trataría de un problema de coste exponencial, a grosso modo (Sin contar bordes) en cada casilla tenemos 4 vecinas, por lo tanto sería $O(4^{m \cdot n})$. Nuestras pruebas tienen los siguientes tiempos:



Así, estimamos también un crecimiento exponencial pero claramente con muchos mejores tiempos absolutos de resolución.