Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 – Estructuras de datos y algoritmos

Informe 3

Estudiante: Felipe Baitelman

1. Algoritmo de búsqueda

a) Elección y aplicación

La primera disyuntiva al resolver el problema fue decidir si utilizar DFS (Depth-first search) o BFS (Breadth-first search). Dado que las soluciones al problema podían estar en profundidades considerables y que el árbol de estados podía ser muy ancho para ciertos problemas, se decidió utilizar DFS. Esto además podía ser una ventaja en el uso de memoria, ya que BFS podía implicar un alto costo de esta en problemas con grillas grandes.

Considerando lo anterior, el algoritmo de búsqueda que se utiliza finalmente es IDDFS (Iterative deepening depth-first search). Se utiliza este algoritmo ya que es una adaptación de DFS (Depth-first search) que permite realizar una búsqueda en profundidad en la que la profundidad máxima esta limitada. Debido a que el problema exige encontrar una solución con una cantidad de pasos limitadas, este algoritmo permite modelar el problema de forma correcta (existe una cantidad límite de pasos para encontrar la solución de la grilla). De esta forma, la idea contempla iterar sobre estados de las grillas, y que la profundidad se traduce a la cantidad de desplazamientos que se utilizan para iterar sobre la grilla hasta encontrar la solución.

a) Heurísticas

IDDFS, por construcción, mejora dos heurísticas conocidas. Por lo tanto, es posible afirmar que existen extensiones de dichas heurísticas en el algoritmo implementado. Estas son:

Alpha-Beta Pruning: Algoritmo de búsqueda comúnmente utilizado en el juego de la máquina en los juegos con dos jugadores. Básicamente, detiene la búsqueda de un estado siguiente cuando cuando comprueba que cualquier movimiento va significar estar peor que en el estado anterior.

Killer Heursitic: Extensión de Alpha-Beta Pruning.

2. Tabla de Hash

b) Contexto

Para encontrar las instrucciones que solucionan cada problema es necesario iterar sobre la grilla de colores que representa el problema realizando desplazamientos de las columnas y o filas de la grilla. Si se considera la grilla inicial como el estado inicial, luego, en cada desplazamiento se encuentra un nuevo estado del problema. De esta forma, al buscar la solución (estado final) es necesario recorrer una serie de estados intermedios.

De esta forma, se decide implementar la Tabla de Hash para guardar los estados que ya fueron revisados y la profundidad (número de desplazamientos) en los que se encontró dicho estado. Por la construcción del problema, esto nos permite identificar soluciones factibles revisando la tabla. Si en una determinada iteración se llega a un estado que ya fue revisado anteriormente, es necesario comparar las profundidades. Si la profundidad actual es mayor o igual que la de la última vez que se reviso dicho estado, es posible inferir que que no es necesario seguir iterando sobre la grilla ya que, si la última vez no se encontró una solución, esta vez tampoco se podrá (quedan menos pasos para resolver el problema). Por otro lado, si nunca si había llegado a ese estado anteriormente o si ya se había llegado pero la profundidad actual es menor que la de la revisión pasada, entonces es factible seguir iterando sobre ese estado por que aún se puede encontrar una solución al problema.

c) Tipo de tabla y propiedades

Por todo lo mencionado anteriormente, la tabla que se implementa es direccionamiento cerrado, en el que las colisiones se resuelven utilizando encadenamiento. La tabla misma es un arreglo de *chars*, y el encadenamiento se consigue utilizando listas de nodos. Cada nodo en las listas de la tabla tendrá un identificador de una grilla, la profundidad a la que se encontró dicho estado y una referencia al siguiente nodo. Se utiliza direccionamiento cerrado ya que resultó ser más favorable en la practica, ya que existía una cantidad de colisiones no menor, implicaba tiempos de búsqueda e inserción menores.

3. Función de Hash

a) Razonamiento de la función

La Función de Hash que se utiliza recibe un *Key* (identificador único de cada grilla) y retorna un *Value* (posición de la Tabla de Hash donde debe guardarse el estado actual). *Key* es de tipo *mpz_t* (librería de números grandes) y *Value* es de tipo *int*.

b) Recorrido de la función

Key, como se menciona anteriormente, es un identificador único de cada grilla. Básicamente, corresponde a la concatenación de todos los colores de cada espacio de la grilla. Al pasar por la Función de Hash, se aplica la función módulo para obtener un entero (*Value*) que este dentro de los límites del tamaño de la Tabla de Hash. El Tamaño de la tabla es de 15.485.863, que es simplemente un número primo bastante

grande que muestra buenos resultados en la práctica. Por lo tanto, el recorrido de la Función de Hash es de enteros entre 0 y 15.485.862.

c) Complejidad de la función

Inserción: Dado que al agregar un nuevo nodo en la tabla este se fija en el primer lugar de la lista en la posición *Value* de la tabla, la inserción será $\mathcal{O}(1)$ (solo se cambia un puntero).

Búsqueda: La complejidad de búsqueda dependerá del factor de carga, λ, de la tabla, done $\lambda = n/m$, donde n será la cantidad de estados totales incluidos en la tabla y m el número de casillas, que como se menciona antes, corresponde a 130589. Una búsqueda exitosa tardará en promedio , $1 + \lambda$, y una no exitosa por otro lado $2 + \lambda/2 - \lambda/2n$. Ya que, según las pruebas realizadas λ es extremadamente cercano a uno en todos los casos, la búsqueda estará muy cerca de $\mathcal{O}(1)$.

4. Análisis de desempeño

a) Comparación de tiempos entre una lista[1] y la Tabla de Hash

Escena	Tiempos de solución (s)	
	Lista	Tabla de Hash
Hanabi (lunatic)	Sin resultado	0.434
Stars (lunatic)	0.630	0.650
Candy (lunatic)	0.000	0.000

b) Comparación de tiempos con y sin utilización de estados guardados

Escena	Tiempos de solución (s)	
	Estados guardados	Estados no guardados
Hanabi (lunatic)	Sin resultado	0.434
Stars (lunatic)	1.786	0.650
Candy (lunatic)	0.000	0.000

c) Tasa de colisiones con variación del tamaño de la Tabla de Hash

Análisis de colisiones tablero Hanabi (lunatic)			
Tamaño de la Tabla de Hash	Colisiones	Inserciones	Proporción
141.650.939	1	30719	0,00323%
15.485.863	26	30719	0,08464%
103.393	3989	30719	12,98544%

[1] Si bien no se implementa una lista propiamente tal, es cosa de generar una Tabla de Hash con sólo una casilla.

5. Posibles mejoras

- Función de Hash ad-hoc: Es posible definir una Función de Hash que se adapte a cada problema que se intenta de resolver. Específicamente, se puede utilizar un método para determinar la cantidad de colores distintos presente en determinada grilla, y en base a la cantidad, el identificador o *key* de cada grilla se puede trabajar como un número más pequeño. Actualmente se concatenan todos los colores de la grilla, multiplicando cada color en cada espacio de la grilla por nueve elevado al espacio en la grilla. No obstante, si se supiera, por ejemplo, que existen solo cuatro colores se puede elevar cuatro al espacio en la grilla, achichando considerablemente el tamaño del identificador.

- Utilización de A*: Algoritmo muy utilizado para encontrar caminos y recorrer grafos. Siendo una extensión del Algoritmo de Dijkstra (que en este caso, dado que los costos son uniformes, es lo mismo que BFS), hubiese sido otro método de resolución del problema, o un eventual complemento a IDDFS.

- Utilización de IDA* (Iterative Deepening A*): Variante de IDDFS que utiliza que adapta A* para utilizarlo como heurística. Dado que que trabaja en el sentido *depth-first*, hubiese involucrado un menor costo de memoria.

- Utilización de Fringe Search: Algoritmo más o menos nuevo, que se posiciona entre A* e IDA*. Hubiese sido otra forma de enfrentar el problema y buscar una solución.