NOMBRE: Jose Maria de la Torre

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

IIC2133 —Estructuras de Datos y Algoritmos

Informe 3

1 ¿Qué algoritmo de búsqueda usaste y por qué?

1. Para el problema aplique el algoritmo IIDFS (Iterative deepening depth-first search). Escogí este algoritmo ya que es mas eficiente que el DFS ya que utiliza menos memoria y para ciertos casos (en especial para resolver puzles) puede ser más rápido. El IIDFS funciona cambiando la profundidad a revisar en cada iteración, revisando en cada iteración más profanidades (osea más cantidad de movimientos). Esto es mas eficiente que utilizar un DFS ya que, es muy probable que la solución sea en menos cantidad de movimientos que el permitido. Entonces el IIDFS logra encontrar la solución de una manera mas rápida que el simple DFS. El algoritmo funciona haciendo los shift pertinentes y los unshifts en los casos que no encontró la solución. En el caso de que encuetra la solución, la recesión retorna true e imprime la solución. En el caso contrario imprime IMPOSIBRU.
   Ademas del IDDFS, implemente una tabla hash que guarda para cada estado de la matriz la profundidad ya vista. El estado lo guarda recorriendo toda la matriz, creando un string donde cada letra es el numero del color de la columna/fila respectiva. Luego en cada iteración revisa si existe o no el estado actual y si existe se compara la profundidad con el estado que existía previamente. Si la profundidad del estado actual es mayor que la guardada, actualizo la profundidad en la tabla y sigo avanzando normalmente, en caso contrario (profundidad guardada es mayor) simplemente dejo de buscar, ya que en algún momento busque con esa matriz con mayor cantidad de movimientos y no llegue a la solución.
   Las heurística utilizada es "killer heuristic". El Killer Heuristic consiste en reducir la mayor cantidad de movimientos, para ir avanzando hasta llegar al limite. De esta manera el árbol no se recorre hasta el final desde un principio sino va avanzando gradualmente. Esto se ve en el código cuando voy aumentando la profundidad máxima en la búsqueda.
   La complejidad de la búsqueda IIDFS para el peor caso es $O(b^d)$ donde b es la cantidad de movimientos (4 en este caso) y d la profundidad maxima. Para la tabla de hash este análisis se hará más adelante en el informe

2 ¿Qué tipo de tabla de hash usaste y por qué?

1. Use una tabla de hash con direccionamiento cerrado. Este tipo de tabla hash maneja las colisiones con listas. Por ejemplo, cuando se calcula el key, y esta key ya se utilizo para otro dato este objeto se pone al principio de la lista (así se ahorra tiempo y no hay que ir hasta el final de la lista) apuntando hacia el próximo nodo. Elegí este tipo de direccionamiento ya que era mas fácil de implementar y además funcionaba de forma eficiente para resolver todos los problemas entregados.

2. Las propiedades que nos ayudan a este problema es que la inserción es de O(1) ya que toma un tiempo constante ir hasta un arreglo, ver si hay algo y si hay algo simplemente hay que apuntar desde el objeto nuevo al antiguo. Por lo tanto es muy eficiente agregar. Para buscar en la tabla es casi lo mismo, ya que simplemente hay que al haber más de un objeto hay que recorrer esa lista, pero como ocurren muy pocas colisiones esa búsqueda es muy rápida.

3. En la tabla lo que guardo es la profundidad del estado (key). Entonces dado un estado de la matriz, reviso la profundidad y en base al estado que estoy con el que esta guardado. Cuando el que esta guardado es mayor que el que estamos parado en ese momento, eso significa que con mayor cantidad de movimientos disponibles no pude llegar a la solución, entonces simplemente retorno y no sigo buscando. Esto mejora muchísimo el IIDFS ya que se tienen que hacer muchas menos comparaciones.

3 ¿Funcion de Hash?

1. Debido a la naturaleza del problema, no fue necesario buscar una función de hash muy desarrollada. Simplemente utilizando un mpz para representar el estado de la matriz (donde cada dígito del numero es un color), y luego utilizando un mod para el tamaño de la matriz (utilizando siempre un numero primo) obtengo la posición del arreglo. Lo importante es que el largo de la tabla de hash sea un numero primo esto ayuda, ya que si fuera un numero no primo, todos los multiplos de ese numero irian a la misma casilla causando muchas colisiones.

2. El tamaño del recorrido sera el tamaño del arreglo iniciado inicialmente. Este tamaño es un numero primo, que escogí basándome en el rendimiento de mi programa probando varios números primos. Se podía escoger números mas grandes, pero estos ocupan mas memoria por lo tanto no son más eficientes.

3. Para la complejidad tenemos tres operaciones

   - Calcular hash: El calculo del hash es de O(1) ya que simplemente toma el estado de la matriz calculado anteriormente (Que tarda O(columnasactivas*filasactivas) en calcular cada estado) y luego le hace modulo por el tamaño fijo de la tabla del hash.
   - Insertar: Para insertar calcula el hash con la función descrita anterior y luego revisa si existe un elemento en esa posición. Si no existe simplemente la pone, si existe algo cambia el puntero del nodo nuevo al nodo que existía antes, creando así una lista. En ambos casos la inserción es de O(1)
   - Buscar: Para buscar un dato dentro de la tabla hash es básicamente lo mismo que anterior. Con O(1) en el caso de que solo haya un dato en ese espacio del arreglo, sino tardara O(1+λ) donde λ es el factor de carga (m/n donde m es la cantidad de datos en la matriz y n el tamaño de la tabla de hash)

4. Las propiedades que la hacen útil para el programa fue lo descrito anteriormente: La rápida búsqueda, incersion, pocas colisiones etc. Se puede ver el impacto en el análisis hecho a continuación, ya que los tiempos que tarda en encontrar la solución son bastante más cortos.

4 Análisis de desempeño

Comparamos utilizar lista o hash:

Tablero	Tiempo Con Hash (segundos)	Tiempo con Lista (segundos)
Charmander	1.364	No lo resuelve
Hanabi	0.246	6.905
Stars	0.03799	0.0359
Candy	0.001	0.001
Disco	0.0043	27.213

Como podemos observar, es muchísimo más rápido ocupar la tabla de Hash que con la lista. Esto se basa en que el tiempo de busqueda en la lista puede tardar muchísimo, ya que para ciertos puzles existen muchisimos estados. Por lo tanto la busqueda se convierte mucho mas grande. Basicamente es lo mismo a que si hubiera una tabla de hash de espacio 1.

Ahora comparamos guardando estados y no guardando

Tablero	Tiempo Gardando Estado (Segundos)	Tiempo No guardando Estado (Segundos)
Charmander	1.364	5.9
Hanabi	0.246	No lo resuelve
Stars	0.0379	3.32
Candy	0.001	0.001
Disco	0.0043	0.0067

Como podemos observar es muchísimo más rápido guardar los estados, incluso hay puzles (como el hanabi) que ni siquiera se resuelvan al no usar los estados anteriores. Eso si pueden haber casos mucho más simple en los cuales utilizar una tabla de hash puede ser menos eficiente dado que el problema es demasiado fácil pero aun así hay que guardar los estados. Aun así el tiempo es despreciable.

Por ultimo comparamos el tamaño de la tabla de hash

Largo Tabla Hash	Tiempo
101	9.810
277	4.817
15797	1.722
492659	1.435
999983	1.393
1068149	1.647
1299827	1.39

Escogiendo solo números primos, podemos observar que los tiempos varían dependiendo del largo de la tabla de hash. Cuando la tabla es muy chica existen muchas colisiones, entonces el tiempo de búsqueda en la tabla de hash es más grande. Y cuando la tabla es muy grande, tarda más tiempo en inicializarse y además gasta mucha más memoria por lo que es más eficiente encontrar un punto medio (en mi caso fue 999983).

5 Bonus

- Para mejorar nuestro trabajo lo que podría hacer es encontrar una función de hash más eficiente que simplemente utilizar el estado de la matriz. Ya que pueden haber valores del hash table que no se están utilizando. También el tamaño del hashtable se puede hacer dinámico dependiendo de la cantidad de colisiones, haciendo rehashing cuando ya existen demasiadas colisiones. O en ves de hacer direccionamiento cerrado podemos utilizar el método de direccionamiento abierto. Una ultima forma de mejorar para mejorar el trabajo es utilizar otro algoritmo de búsqueda, como por ejemplo utilizar A* en ves de IDDFS. O