Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructuras de Datos y Algoritmos

Tarea 3

2do semestre 2018

Entrega código: 2 de noviembre
Entrega informe: 3 de noviembre

Objetivos

1. Implementar un algoritmo de búsqueda en el espacio de estados.
2. Crear una implementación propia de una tabla de hash Ad hoc al problema.
3. Analizar las colisiones y distribución dentro de la tabla de Hash durante la ejecución.

Introducción

El puzle de 15 y su versión más pequeña (puzle de 8) son problemas clásicos donde el objetivo es llegar desde un tablero desordenado a un tablero ordenado (figura 1).

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

6	2	3	1
5	9		7
13	4	11	12
8	14	15	10

Figura 1: izquierda tablero ordenado, derecha tablero desordenado

Las acciones válidas sobre el tablero tradicional son movimientos de las piezas numeradas hacia la pieza vacía. En esta versión del problema se aceptará también que las piezas se puedan mover hacia la pared y aparezcan en la pared contraria (figura 2).

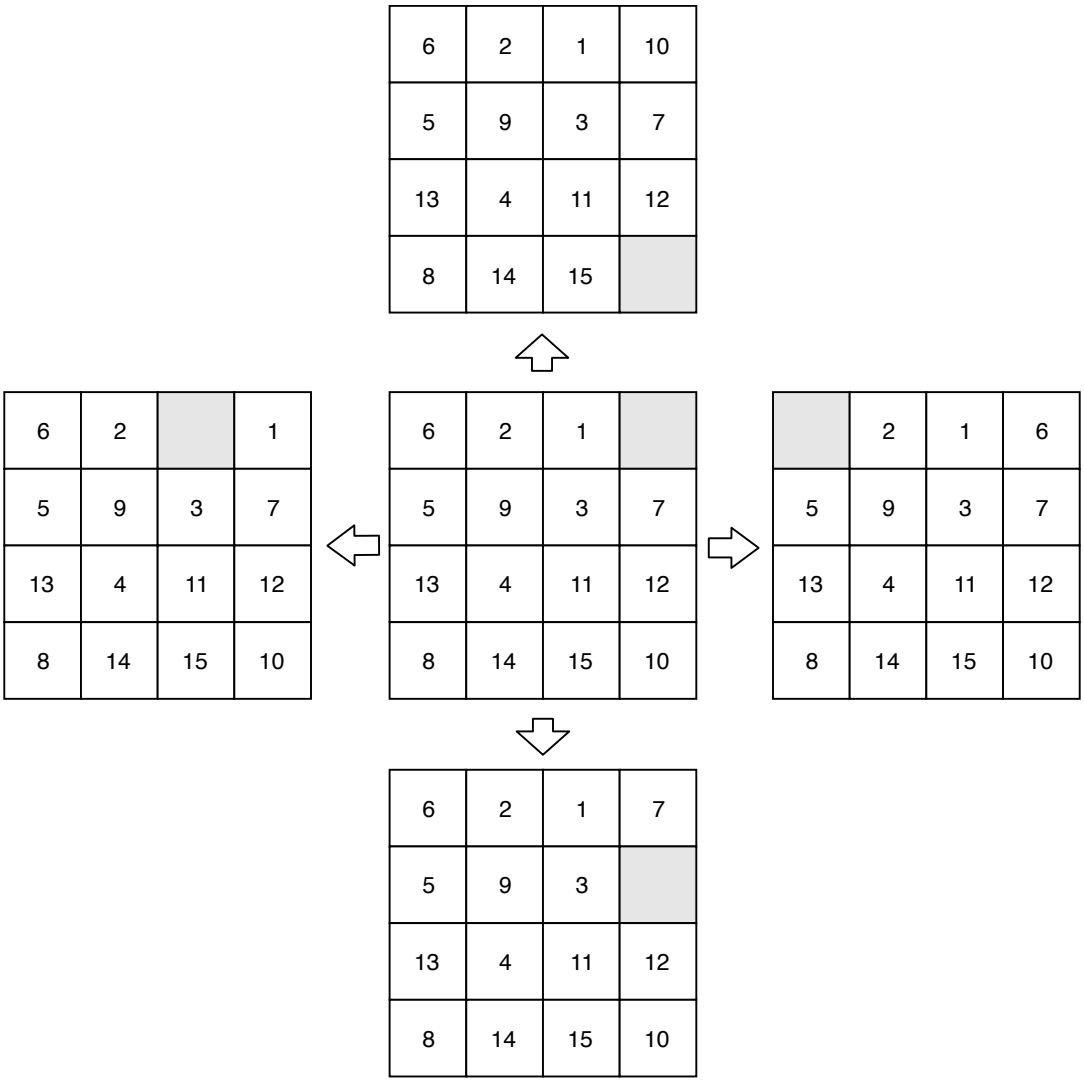

Figura 2: movimientos válidos

Problema

Se les pide entonces implementar una solución para el juego de 15 y su versión más pequeña, el puzle de 8 en el mínimo número de pasos.

Estados

Como podrán notar, a partir de una configuración inicial del tablero tenemos siempre 4 movimientos posibles, y cada uno de estos nos lleva a una configuración distinta a la anterior. Cada configuración (o estado) distinta del tablero se puede representar como un nodo en un

grafo de estados donde las aristas que los unen son los movimientos de las piezas. Este grafo tiene un tamaño muy grande como para ser almacenado durante la ejecución (el puzle de 8 tiene 362880 estados y el de 15 tiene 20922789888000 estados!), es por esto que se van a almacenar solamente los estados necesarios para la búsqueda.

BFS

En particular el algoritmo BFS permite explorar el grafo de estados desde el estado inicial expandiendo los siguientes estados según su lejanía al estado inicial. Esto hace que el algoritmo siempre encuentre los estados en el mínimo de pasos desde el estado inicial (el camino óptimo). Evidentemente BFS no es el único algoritmo que sirve para esta tarea aunque es suficiente para resolver los test de la tarea.

Tabla de Hash y función de Hash

Durante la exploración del grafo se van a encontrar muchos estados repetidos, por lo que es importante poder revisar si un estado ya fue explorado anteriormente para así no revisarlo de nuevo. Para esto es necesario implementar una tabla de hash, la cual almacena los estados que ya fueron explorados por el algoritmo.

Para esto entonces tenemos los dos componentes principales para hashing: función de Hash y tabla de Hash. Esto nos permitirá a partir de un estado del tablero, generar un Hash del tablero para guardar en nuestra tabla, para luego en un tiempo acotado identificar si un estado es o no nuevo, disminuyendo la cantidad de estados a revisar durante la ejecución de nuestro algoritmo BFS, haciendo más eficiente encontrar la solución a nuestra tarea.

1. **Función de Hash**
   La función de Hash obtiene como resultado un valor único por estado. Como vieron en clases, existen muchas maneras de implementar estas funciones, por lo que queda a su criterio cuál elegir. Lo importante que deben lograr siempre es que 2 estados iguales deben dar resultados iguales (previo a hacer **mod n**). Una buena función de Hash distribuye bien dentro de la tabla (usando completamente la tabla). Durante la ayudantía del viernes 19 se hablará sobre distintas funciones que podrían servir.
2. **Tabla de Hash**
   Lo más importante que deben considerar al momento de crear su tabla de Hash es el manejo de colisiones. Si dos estados diferentes terminan en la misma posición de la tabla, deben ser capaces de insertar el nuevo valor en otra posición (o como decidan manejar las colisiones). El tamaño de sus tablas y que tan llenas están puede afectar también el número de colisiones. Por lo que se recomienda que tengan un factor de llenado determinado para que pasado cierto nivel, se pueda aumentar el tamaño de la

tabla, disminuyendo las colisiones. Se pide que para su tarea creen y manejen un **tabla de hash dinámica**. Esto significa que pasada una cierta cantidad de inserciones a la tabla, se aumente el tamaño de la tabla y se vuelvan a insertar todos los elementos a la nueva tabla.

Input

Tu programa debe ejecutarse de la siguiente forma:

```
./solver [test.txt] [output.txt]
```

Donde `[test.txt]` corresponde al archivo con el tablero inicial a resolver. La primera línea del archivo contendrá solo un caracter **n**, y este corresponderá a las dimensiones de la matriz de **n x n**. En las siguientes n líneas se entregarán los valores iniciales del tablero. Estos estarán separados por **comas**. Los números del 1 al $n \cdot n - 1$ corresponden a las piezas numeradas, mientras que el número 0 indica la casilla vacía. Puede asumir que n siempre es 3 o 4.

```
3
1,2,3
7,5,4
0,8,6
```

Para la lectura del input, se recomienda que usen la librería **string.h**.

Output

El archivo `[output.txt]` deberá tener un número que indica el número de pasos ejecutados y luego las posiciones x, y de las piezas movidas en cada paso. Ejemplo:

```
3
0,1
2,1
2,2
```

En este ejemplo para que el output sea correcto debe cumplirse que los pasos ejecutados lleven al estado final desde el estado inicial y el número de pasos debe ser el mínimo.
El estado final debe tener las casilla numeradas ordenadas desde el 1 al $n \cdot n - 1$ y la casilla final debe estar vacía.

Informe y Análisis

Conteste **brevemente** las siguientes preguntas:

1. Presente a partir de gráficos el número de colisiones dentro de cada posición de su tabla de Hash. Ya que la tabla debe ser dinámica, muestre este gráfico solo para la tabla final de su programa. Se recomienda para esta parte mantener un contador en la tabla para llevar el número de veces que se intentó insertar en cada slot. Explique, con respecto a su función de Hash implementada, el motivo por el que se produce esta distribución. [2 pts.]
2. A partir de sus resultados obtenidos en la parte 1, justifique si cree que escogió una buena función de Hash, dadas las características del problema planteado. [2 pts.]
3. Muestre el número de estados creados para cada test y el número de veces que tuvo que agrandar su tabla de Hash. Según esto diga cuánto tiempo se gasta en hacer rehashing de la tabla de Hash. [2 pts.]
4. **BONUS:** En su código, implemente al menos 2 métodos de manejos de colisiones. Presente en un gráfico las diferencias en el número de colisiones dentro de su tabla de Hash respecto a los distintos métodos. [2 pts.]

Evaluación

La nota de tu tarea se dividirá en dos partes:

- 60 % corresponde a la nota de tu código, el cual deberá pasar cada test en menos de 10 segundos.
- 40 % corresponde a la nota del informe.

Tu programa será probado con diferentes tests de dificultad creciente. Para cada uno de ellos, tu programa deberá ser capaz de entregar el output correcto dentro de un tiempo acorde a la complejidad esperada. Pasado ese tiempo el programa será terminado y se te asignará 0 puntos en ese test.

Entrega

- Deberás entregar tu tarea en el repositorio que se te será asignado, asegúrate de seguir la estructura correcta de este.
- Se espera que tu código compile con `make` dentro de la carpeta **Programa** y genere un ejecutable de nombre `solver` en esa misma carpeta.

- Se espera que dentro de la carpeta **Informe** entregues tu informe en formato PDF, con el nombre `Informe.pdf`.

Estas reglas ayudan a recolectar y corregir las tareas de forma automática, por lo que su incumplimiento implicará un descuento en tu nota.
Se recogerá el estado de la rama `master` en tu repositorio 1 minuto pasadas las 23:59 horas del día de la entrega (tanto para el código como para el informe). Recuerda dejar ahí la versión final de tu tarea. No se permitirá la entrega de tareas atrasadas.

Bonus: Manejo de memoria perfecto (+5 % a la nota del Código)

La nota de tu código aumentará en un 5 % si `valgrind` reporta 0 leaks y 0 errores de memoria en tu código. Todo esto considerando que tu programa haga lo que **tiene** que hacer. El bonus solo aplicará en caso que tu nota de código sea sobre 4.

Bonus: Tests Lunatic (2 puntos de la nota de código)

Puedes tener puntaje extra en tu nota de código si logras resolver los tests más difíciles de este problema de manera óptima. Para esto se recomienda investigar sobre IDDFS, A* o IDA*.