IS553 Programación IV – Proyecto.

8 Puzzle

Escriba un programa para resolver el problema del 8-puzzle (y su generalización natural) usando el algoritmo de búsqueda A*.

El Problema. El problema del 8-puzzle es un rompecabezas popularizado por Sam Loyd en la década de 1870. Se juega en una cuadrícula de 3-por-3 con bloques marcados del 1 al 8 y uno en blanco. Su objetivo es reorganizar los bloques para que estén en orden. Está permitido deslizar los bloques horizontal o verticalmente dentro del cuadro en blanco. A continuación se muestra una secuencia de movimientos legales desde un tablero inicial (izquierda) a la posición objetivo (derecha).

```
   1 3       1   3       1 2 3       1 2 3       1 2 3
 4 2 5  =>  4 2 5  =>  4   5  =>  4 5    =>  4 5 6
 7 8 6      7 8 6      7 8 6       7 8 6       7 8
inicial                                         meta
```

Búsqueda mejor el primero. A continuación se describe una solución algorítmica al problema que muestra una metodología de inteligencia artificial conocida como Algoritmo de búsqueda A*. Se define un estado del juego como: la posición del tablero, el número de movimientos realizados para alcanzar la posición del tablero y los estados anteriores. Primero, se inserta el estado inicial (el tablero inicial, 0 movimientos y, un estado anterior nulo) dentro de una cola de prioridad. Luego, se borra de la cola de prioridad el estado con la prioridad mínima y se inserta, dentro de la cola de prioridad, todos los estados vecinos (aquellos que pueden ser alcanzados en un movimiento). Repita este procedimiento hasta que el estado sacado sea el estado objetivo. El éxito de este enfoque depende de la elección de la función de prioridad para un estado. Se consideran dos funciones de prioridad:

- ***Función prioridad de Hamming***. El número de bloques en posición incorrecta, más el número de movimientos realizados para alcanzar el estado. Intuitivamente, estados con un número menor de bloques en posición incorrecta están cerca del estado objetivo y se prefiere estados que hayan sido alcanzados usando un pequeño número de movimientos.
- ***Función prioridad Manhattan***. La suma de las distancias (suma de la distancia vertical y horizontal) de los bloques a su posición objetivo, más el número de movimientos realizados hasta ahora para alcanzar el estado.

Por ejemplo, las prioridades Hamming y Manhattan del estado inicial a continuación son 5 y 10, respectivamente.

```
 8 1 3      1 2 3      1 2 3 4 5 6 7 8     1 2 3 4 5 6 7 8
 4   2      4 5 6      ---------------     ---------------
 7 6 5      7 8        1 1 0 0 1 1 0 1     1 2 0 0 2 2 0 3
inicial      meta      Hamming = 5 + 0    Manhattan = 10 + 0
```

Observación clave: para resolver el puzzle desde un estado dado en la cola de prioridad, el número total de movimientos que debemos hacer (incluidos los que ya se han hecho) es al menos su prioridad, usando la función Hamming o Manhattan. (Para la prioridad Hamming, esto es cierto, ya que cada boque fuera de lugar debe moverse al menos una vez para llegar a su posición objetivo. Para la prioridad Manhattan, esto es cierto, ya que cada bloque debe mover su distancia Manhattan a su posición objetivo. Observe que no se cuenta el bloque en blanco cuando se calcula las prioridades Hamming o Manhattan.)

Consecuentemente, tan pronto como se saca un estado, no sólo se ha descubierto una secuencia de movimientos desde el tablero inicial al tablero asociado con el estado, sino que se tiene el menor número de movimientos. (Desafío para aquellos con inclinación matemática: probar este hecho).

Una optimización importante. Después de implementar la búsqueda mejor el primero, se dará cuenta de una característica molesta: estados correspondiendo a la misma posición del tablero están en la cola de prioridad muchas veces. Para evitar exploraciones innecesarias de estados no útiles, cuando considere los vecinos de un estado, no introduzca el vecino si su posición de tablero es la misma que la del estado anterior.

```
 8 1 3      8 1 3      8 1 3
 4   2      4 2        4 2
 7 6 5      7 6 5      7 6 5
anterior    estado     evitar
```

La tarea. Escriba un programa ***Solver.java*** que lea el tablero inicial desde ***stdin*** e imprima a ***stdout*** una secuencia de posiciones de tablero que resuelva el puzzle en el menor número de movimientos. También imprima el número total de movimientos y el número total de estados encolados.

La entrada consistirá de un tablero de dimensión N seguido por la configuración inicial N-por-N del tablero. El formato de entrada usa 0 para representar el bloque en blanco. Como un ejemplo,

```
% more puzzle04.txt
3
0 1 3
4 2 5
7 8 6

% java Solver < puzzle04.txt
  1 3
4 2 5
7 8 6

1   3
4 2 5
7 8 6
```

```
1 2 3
4   5
7 8 6

1 2 3
4 5
7 8 6

1 2 3
4 5 6
7 8

Número de estados encolados = 10
Número de movimientos = 4
```

Tenga en cuenta que su programa deberá trabajar para tableros arbitarios de N-por-N (para todo N mayor que 1), aunque puede ser demasiado lento resolver algunos de ellos en una cantidad de tiempo razonable.

Detectando puzzles sin solución. No todas las posiciones de tableros iniciales pueden alcanzar el estado objetivo. Modifique su programa para detectar y reportar tales situaciones.

```
% more imposible-puzzle3x3.txt
3
1 2 3
4 5 6
8 7 0

% java Solver < imposible-puzzle3x3.txt
Sin solución posible
```

Sugerencia: usar el hecho de que las posiciones del tablero están divididas dentro de dos clases de equivalencia respecto a su alcanzabilidad: (i) aquellas que llevan a la posición objetivo y (ii) aquellas que llevan a la posición objetivo si se modifica el tablero inicial mediante el intercambio de cualquier par de bloques adyacentes (no-banco). Hay dos formas de aplicar esta sugerencia:

- Ejecutar el algoritmo A* simultáneamente sobre dos instancias de puzzle - una con el tablero inicial y la otra con el tablero inicial modificado mediante el intercambio de un par de bloques adyacentes (no-blanco). Exactamente uno de los dos llevará a la posición objetivo.
- Derive una fórmula matemática que le diga cuando un tablero tiene o no solución.

Resultados. Organice su programa en un número apropiado de tipos de datos. Como mínimo, se está obligado a implementar la siguiente API (Board y Solver). Sin embargo, se es libre de añadir métodos o tipos de datos (tales como State).