Apartado A:

Para resolver este apartado, lo principal es plantear el problema del cuadrado mágico mediante una nueva clase “ProblemaCuadradoMagico”. Esta clase consta de dos subclases, la primera ,“EstadoCuadradoMagico”, permite inicializar los diferentes estados del problema, está formado por un constructor, dos getters para poder leer sus valores, un método toString y los métodos equals y hashcode que permiten diferenciar dos estados diferentes. La segunda subclase “AcciónCuadrado” implementa todos las funciones necesarias para saber si una acción es aplicable y el resultado de aplicar la misma a un estado concreto. Esta es la que más dificultades causó, el principal problema que encontramos fue como evitar que se modificasen las casillas con un valor definido en el estado inicial, para solucionarlo creamos un nuevo parámetro que se iguala a la matriz inicial, el cual nos permite saber cuáles son las casillas modificables. Otra de las dificultades fue detectar por que el nuevo estado sobreescribia el estado inicial, esto se debía a que Java trabaja con punteros, por lo que al copiar la matriz directamente también copiaba su dirección en memoria, por lo que modificaba el estado inicial, para solucionarlo, modificamos la forma de copiar la matriz del estado, duplicandola casilla por casilla.

Además también cuenta con un método que inicializa todas las acciones que se pueden dar ,”ProblemaCuadradoMagico”, y otro para comprobar si el estado es o no meta, este último simplemente suma las filas ,columnas y diagonales, devuelve falso si alguna de esas sumas no coincide con el valor que buscamos (N\*((N^2)+1)/2) y verdadero en si todas las

sumas coinciden.

Las otras dos clases son las que se encargan de buscar la solución usando la búsqueda en profundidad, “EstrategiaBusquedaProfundidad”, y anchura, “EstrategiaBusquedaAnchura”.

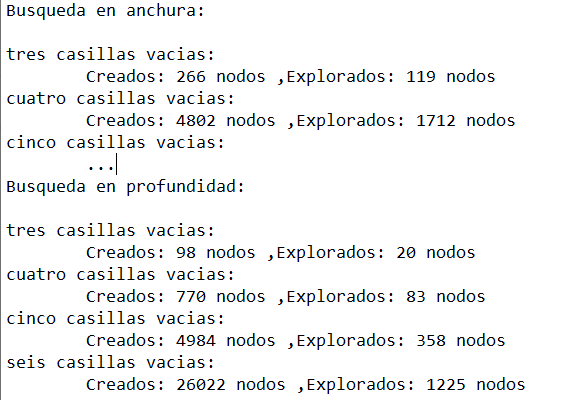
Para implementar estas estrategias usamos la misma estructura que en la búsqueda en grafo, pero cambiando la forma en la que se usa la frontera, en el caso de la búsqueda en profundidad usamos una cola LIFO y en el caso de la búsqueda en anchura usamos una cola FIFO. Estas dos clases cuentan con una función Soluciona que comprueba si el estado actual es meta, y una función Sucesores que expande la frontera.

Mientras las implementamos, descubrimos un fallo en el método equals de la clase “EstadoCuadradoMagico”, el cual no comparaba bien las matrices de los dos estados, mostrando siempre que el nuevo estado nunca se encontraba en la frontera ni en la lista de explorados.

A mayores de esto tenemos también una clase Main que nos sirve para comprobar que todo funciona correctamente.

Tras varias pruebas con diferentes estados iniciales, se demuestra que la búsqueda en profundidad es más rápida que la búsqueda en anchura.

–Prueba con el ejemplo dado en el enunciado–



–Prueba con el ejemplo dado en el enunciado–

Esto se debe a que, en el caso de la búsqueda en anchura, si la meta se encuentra en una altura H del grafo, se tienen que generar todos los nodos de esa altura previamente, almacenando todos los caminos posibles . En el caso de la búsqueda en profundidad, solo se almacena uno de esos caminos a la vez. Aún así , ninguna de estas dos soluciones es óptima , ya que ninguna asegura que el nodo que se encuentra es el más próximo al estado inicial.

Apartado B:

Para este apartado, el primer paso es escoger una heurística que sea tanto admisible como consistente. La función heurística escogida consiste en comprobar qué porcentaje de todas las combinaciones que tiene el estado (suma de las filas, columnas y diagonales) son correctas. La heurística es admisible ya que en ningún momento el coste va a superar el coste real de alcanzar la meta (el valor va a ser siempre menor que uno excepto en el estado meta), y consistente ya que una vez aplicada una acción, el coste de llegar a la meta va a ser igual o mayor que en el estado anterior, por lo que el camino desde el nodo padre a la meta va a ser menor que el coste de llegar al nodo hijo y, acto seguido, a la meta.

Después de esto creamos la clase “EstrategiaBusquedaInformada”, la cual corresponde a la estrategia de búsqueda A\*. Para implementarla aprovechamos la estrategia de busqueda por grafo, y añadimos una nueva función “ordenarFrontera” que ordena la frontera en función del coste de los nodos.

Apartado C:

Para este apartado se han implementado comprobaciones mayores al comprobar si una acción es aplicable o no. En este caso se comprueba si el valor que se va a añadir no cumple los requisitos de suma de su columna o fila, o diagonal en caso de que forme parte de ella.

A mayores de los procedimientos implementados, se podrían crear threads que comprueben cada uno una acción de la frontera aplicable y sus sucesores, además se podría utilizar un array con los números que quedan por utilizar y comprobar en una fila o columna casi completa si su valor hace que la suma de la misma sea correcta, para añadirlo así directamente y eliminarlo de la lista.

Es posible que se pueda optimizar mucho mas adaptando la estrategia de búsqueda al problema del cuadrado mágico.