KAKURO

DESCRIPCIÓN DE ALGORITMOS Y ESTRUCTURAS DE DATOS

Judith Almoño Gómez
Álvaro Armada Ruíz
Pau Cuesta Arcos
Pol Vallespí Soro

Estructura usada para representar el tablero:
Para representar el tablero, hemos usado una matriz de Cell(), que es la clase que hemos usado para representar una celda.

Jugar:

Para jugar hacemos un bucle que a cada iteración comprueba si hemos finalizado el kakuro, comprobando los valores que el usuario ha introducido con los valores correctos, previamente calculados por la función resolver. Si decide introducir un número se comprueba que sea un número entre 1 y 9 incluidos y que sea una casilla blanca, en caso contrario se informa al usuario del error. Si todo es correcto, se comprueba si este valor es válido o inválido (no correcto, comprueba por ejemplo que no esté repetido en la fila). De cualquier manera lo introducimos en el kakuro. Esto es debido a que cuando implementemos interfaz gráfica esos valores los marcaremos en rojo, pero será posible escribirlos.
Para el tema de las ayudas, comprobar si un valor es correcto o obtener el valor correcto. Se hace de manera sencilla ya que tenemos los valores correctos calculados al comenzar con la función de resolver.

Resolver:

Estructuras de datos usadas:
Un vector de enteros, vec, de 9 posiciones inicializado todo a cero, que nos permitirá saber en todo momento que valores están disponibles según el estado de la run horizontal en la que nos encontramos.

Descripción del algoritmo:
Para hacer la función de resolver, hemos hecho un algoritmo basado en backtrack, pero evitando valores que no son posibles debido al estado del tablero en ese momento. El algoritmo funciona de la siguiente forma. Recorremos el tablero de izquierda a derecha y de arriba abajo, empezando por la casilla ubicada en la fila 0 y columna 0. Cada vez que llegamos a una celda negra, comprobamos si el parámetro sum es cero, si no lo es, quiere decir que esa solución es imposible y hacemos backtrack. En caso de que sea cero, quiere decir que esa solución parcial es probable. También volvemos a poner el vector vec a zero, ya que todos los valores vuelven a ser posibles, y actualizamos el valor sum con la suma de la siguiente run horizontal.
Si estamos en una celda blanca, hacemos un for de i =1 hasta i = 9, en el que miraremos qué valores están disponibles tanto en la fila, con vec[i] = 0, como en la columna, usando la función checkColumn de la clase kakuro, y en caso que sea posible colocar el valor i en la celda en cuestión, fijamos el valor, marcaremos vec[i]= 1, y miramos la celda del lado derecho.

En caso de que la celda blanca sea la última de la fila, comprobaremos que sum sea cero, si lo es cambiaremos de fila, si no lo es haremos backtrack.
Si conseguimos llegar a la última celda y sum es igual al valor que deberíamos sumar, quiere decir que hemos encontrado una solución.
Para comprobar que no hay valores repetidos por columna y que la suma de las columnas es correcta, usamos una función llamada checkColumn. Esta función mira, cada vez que queremos poner un valor, miramos que ese valor no esté usado ya y que la suma pueda ser posible. Para saber si estamos en la última celda de una run vertical, lo indicamos con un boolean f, y en caso que f sea cierta, quiere decir que nos encontramos en este caso y por lo tanto, la suma de los valores de la columna tiene que ser igual a la suma de la columna. Si no se da esto, quiere decir que ese valor no es posible.

Validar:

Utilizamos el mismo algoritmo que para resolver, pero no para hasta encontrar dos soluciones. Si cuando ha visitado todos los posibles valores solamente ha encontrado una solución sabemos que el kakuro es válido, si no ha encontrado ninguna o bien ha encontrado más de una (2 en este caso, ya que si encuentra 2 paramos de buscar) será no válido.

Dado que no podemos retornar verdadero al encontrar una solución y tenemos que guardar el número de soluciones, se almacene este en un parámetro de la función, concretamente en un array de una posición llamado res.

Generar:

Estructuras de datos usadas:

Una matriz de 2 dimensiones de Cell(), para representar el tablero.
Una matriz de 3 dimensiones de enteros, tempBoard, del tamaño del tablero pero con un vector de nueve posiciones en cada "celda", que nos permite almacenar información adicional que necesitaremos para poder garantizar que el kakuro generado es único.

Descripción del algoritmo:

Para generar el tablero, primero ponemos todas las celdas de la primera fila y columna como celdas negras. Luego generamos la segunda fila y la última de la siguiente forma
c[i][j] = c[size-i][size-j], de esta forma conseguimos un tablero simétrico respecto a la diagonal. El siguiente paso consiste en rellenar las demás celdas del tablero, mirando que se cumplan algunas reglas, comprobadas según su prioridad. Primero de todo, comprobamos que no hay runs ni verticales ni horizontales de más de 9 celdas blancas, si se da el caso que hay una run de 9, la siguiente celda será negra. Luego evitamos que haya celdas blancas solas, es decir, runs de 1 sola celda blanca, pero esto no siempre será posible. Si se da el caso que no pasa nada de lo descrito anteriormente, pondremos una celda blanca o negra aleatoriamente según el entero de dificultad escogido por el usuario.
Para este paso, nos ayudamos de las funciones countWhiteCellsV y countWhiteCellsH, que simplemente cuentan las celdas blancas de las runs a las que pertenecería una celda en concreto si fuese blanca. Para hacer esto, se van llamando recursivamente a si mismas retrocediendo una celda en su respectiva dirección hasta que encuentran una celda negra.
La celda negra devolverá 0 y por cada blanca 1 más el número que nos ha devuelto la llamada recursiva.
Una vez tenemos el tablero rellenado, tenemos que comprobar que no hay celdas blancas solas, en runs de 1. Para esto, usamos un boolean canvi y while, que iterará mientras canvi sea true. Cada vez que encontramos una celda blanca que pertenece a una run de 1, ponemos esa celda como negra y ponemos canvi a true, de esta manera iteramos hasta que en una comprobación entera del tablero no hagamos hecho ningún cambio.
Para saber si una celda blanca pertenece a una run de 1, simplemente miramos en las 4 direcciones (arriba, derecha, abajo y izquierda) de cada casilla negra, si su celda adyacente es blanca y la siguiente es negra. Si pasa eso, hemos encontrado una run de 1.
Una vez hecho esto, solo queda comprobar que el tablero es conexo. Para hacer esto, tenemos que mirar que, desde cualquier celda blanca se puede llegar a todas las celdas blancas del tablero. Utilizamos dos funciones howManyWhites, que cuenta el número de celdas blancas del tablero y DFS. La función DFS, se ayuda de

una matriz del mismo tamaño que el tablero, que usamos para indicar si una celda se ha visitado, es decir, cuando una celda se visita, marcamos la correspondiente celda de la matriz auxiliar con un 1. El DFS mirará todas las celdas adyacentes a una celda dada, y por cada blanca, irá aumentando la suma. Cuando ya no pueda avanzar más, habremos acabado.
En el caso, que la función howManyWhites y DFS retornen valores diferentes, tendremos que hacer un tablero nuevo, ya que el tablero no es válido. Si devuelven el mismo valor, tendremos un tablero válido y podremos proceder a comprobar si puede tener solución única.
Antes de rellenar el tablero con números, creamos un tablero auxiliar de la misma medida que el board del kakuro, pero en cada posición tiene 9 enteros. En caso de que sea una casilla negra en ellos guardaremos información como el número de casillas blancas hacia abajo y hacia la derecha y la suma asignada a esa casilla. En caso de que sea blanca, el número en la posición i indica si el número i+1 es un valor posible para esa casilla. Veámos un ejemplo,
1 0 0 0 1 0 1 0 1, indica que los valores posibles para esa casilla son 1, 5, 7 y 9.
Después de crear este tablero, recorremos filas y columnas para asignar a las casillas negras el número de casillas blancas que tienen hacia abajo y derecha.
Para rellenar el tablero de números buscamos sumas cuyas combinaciones compartan un solo número, como puede ser el caso de 7 en tres casillas (suma 1+2+4) y 13 en dos casillas (sumas 4+9, 5+8, 6+7). Para encontrar estos números, disponemos de un array de arrays, en el que la posición i del primer array contiene las posibles sumas que se pueden conseguir con i casillas. De esta manera, tenemos una suma para el número de blancas en la fila y otra para el número de blancas en la columna. Con estas sumas llamamos a una función que nos dice que valores estan presentes en las posibles combinaciones de números que suman el valor suma y después comprobamos si solamente tienen un valor en común. De esta manera, comenzamos por la primera casilla y vamos iterando por la misma fila, hasta llegar al final, entonces cambiamos de fila.
Una vez hemos encontrado una casilla para la que poder fijar un valor, en el tablero auxiliar fijamos ese valor para la casilla y para el board del kakuro, y calculamos los valores posibles para las casillas de la fila y columna correspondientes. Desde este punto llamamos a una función que intentará fijar más valores a las casillas que ya tengan valores posibles pero que aún le hayamos asignado un valor definitivo en el board de kakuro (en la primera iteración de está función solo serán aquellas casillas de la misma fila o columna que la fijada anteriormente).
Para saber que valor fijar aplicamos lo mismo que antes, buscamos números que sus combinaciones tengan un valor único, pero está vez, único entre los posibles valores que tenia asignada esa casilla. En caso de que encontremos ese valor, se lo asignamos en el tablero auxiliar y en el board del kakuro y propagamos las restricciones para fila y columna. Hacemos una llamada recursiva a la función desde la posición 0,0, para que intente tratar la primera casilla con valores posibles que encuentre.

Es importante remarcar que aquí diferenciamos entre 3 posibles casos, que la casilla haya sido tratado por fila, columna o ambas.
En que caso de que haya sido tratada por ambas solamente fijamos valor si tiene un valor posible en el tablero auxiliar.
Para el caso de fila o columna son parecidos. Básicamente hacemos la distinción ya que, cuando propagamos restricciones por fila o columna, puede ser el caso de que sea la primera vez que visitamos esas celdas. Y dado que detectamos errores cuando los posibles valores de una casilla son todo 0, tenemos que diferenciar claramente cuando son 0 porque el valor asignado no es correcto o porque aun no las hemos tratado.
En caso de que para alguna casilla no pueda encontrar ningún valor, retornará falso, volviendo así a la llamada anterior. En esta llamada se probaran otras combinaciones de números para intentar fijar un valor. En caso de que se rellenen todas las casillas blancas, retornará cierto. En caso de que no pueda rellenar ese tablero, retornará falso y se creará un tablero nuevo.