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 y guardados en la carpeta correspondiente. 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. En caso de que el valor sea incorrecto, en la interfaz gráfica el número será de color rojo[1]. Si fuese posible, se marcaría de color negro[1]. Finalmente también existe la opción de lápiz, que es una manera de anotar los valores posibles para que sea más fácil visualizarlo.
En cuanto a las ayudas que se le ofrece al usuario, diferenciamos entre dos, corregir un valor o obtener el valor correcto. Se hace de manera sencilla ya que guardamos la solución para todos los kakuros. Por lo tanto, para la primera opción solo hemos de comprobar si el valor escrito en la casilla es el mismo que en el kakuro solucionado, y para la segunda, escribimos directamente en la casilla el valor que tenga el kakuro solucionado.

[1] Los colores mencionados son los colores por defecto, se pueden cambiar en el botón de opciones.

Resolver:

Estructuras de datos usadas:

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

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 saber, para las casillas blancas que valores son posibles introducir en cada momento del proceso.

Una clase de Pair, en la que el primer elemento corresponde con la posición vertical de la casilla en el kakuro, y el segundo con la horizontal.

Un set de Pairs, uniques, en el que se guardaran las coordenadas de casillas blancas con valor único, como explicaré en la descripción del algoritmo.

Descripción del algoritmo:

Para comenzar con la resolución de un kakuro, haremos un recorrido por todo el todo el tablero desde la casilla de arriba a la izquierda hasta abajo a la derecha, parandonos en las casillas negras. Por cada casilla negra que encontremos, miraremos de cuantas casillas es la run y que número ha de sumar esta. Una vez hecho esto calcularemos los diferentes valores que aparecen en las combinaciones que suman el numero indicado con el numero de casillas blancas disponibles. Para aquellas casillas negras que tengan una run vertical asociada a ellas, introduciremos en tempBoard los valores posibles que tienen todas las casillas blancas de esta run. Al tratar el tablero desde la primera fila hasta la última y de izquierda a derecha, nos aseguramos que para cualquier casilla blanca, siempre hayamos mirado su casilla negra vertical antes que la horizontal. Esto es importante, ya que lo mismo que hacemos para las runs verticales, lo haremos para las horizontales. Pero con una pequeña modificación, dado que sabemos que las casillas ya tendrán valores posibles (porque habremos llegado anteriormente a su casilla negra vertical), en vez de fijar directamente los valores que aparecen en las diferentes combinaciones para sumar el numero de la casilla negra con tantas casillas como tenga la run, haremos una intersección con los valores posibles que tenia la casilla y los nuevos. Véamos un ejemplo, supongamos una casilla blanca que tiene como casilla negra vertical un 7 con una run de 3 casillas blancas, y como horizontal un 13 con una run de 2 casillas blancas. Las diferentes combinaciones para formar 13 son: 4+9, 5+8 y 6+7. Para 7 solamente existe una, 1 + 2 + 4. Como podemos ver, antes de llegar a la casilla negra con valor 13, la casilla blanca correspondiente tendría como valores posibles 1, 2 y 4. Al hacer la intersección con las combinaciones de 13, nos quedaría un único valor posible. Cada vez que una casilla blanca se queda con un único valor posible, creamos un Pair que guarde sus coordenadas, y lo introducimos en el set uniques.

Una vez todo el tablero recorrido, cada casilla tiene guardados en tempBoard sus valores posibles. Ahora es el momento de propagar restricciones, es decir, si sabemos que en una casilla hay un dos fijado, podemos posiblemente eliminar valores de su fila y columna. Por lo tanto, vamos sacando elementos del set uniques. Con cada elemento buscamos su casilla negra asociada tanto vertical como horizontalmente, a la vez que miramos el tamaño de la run. Una vez sabemos el tamaño de la run y el valor de la casilla negra, podemos volver a calcular las posibles combinaciones para sumar el valor de la casilla negra, pero esta vez sabiendo que tenemos un cierto valor fijo. Con estos valores posibles, realizaremos la misma intersección que he explicado anteriormente, y en caso de que alguna casilla tenga valor único la añadiremos al set uniques, para poder tratarlo a continuación.

Una vez el set se quede sin elemento, si hemos rellenado todas las casillas blancas, habremos resuelto el kakuro. Por el contrario, si no hemos conseguido rellenar todas, llamaremos a una función recursiva que resolverá el kakuro mediante backtrack.

La función 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. Además, ahora que hemos reducido considerablemente los valores posibles de cada casilla, solamente probaremos con los valores que aparezcan en tempBoard para esa casilla. 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, con alguna pequeña modificación. En el momento de fijar valores a casillas, si en tempBoard alguna casilla se queda sin valores posibles, sabremos que el kakuro no es válido.

Si ha conseguido fijar todos los valores simplemente en el proceso de propagación de restricciones, el kakuro será válido.

Para la función recursiva, ahora no para al encontrar una solución, sigue hasta encontrar dos. Cuando ha probado todos los posibles valores para cada casilla (solamente los valores que aparezcan en tempBoard) 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.

Dificultad:

Estructuras de datos usadas:

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 saber, para las casillas blancas que valores son posibles.

Descripción del algoritmo:

Para decidir la dificultad de un kakuro miramos diferentes características de este. Para empezar, contamos el número de casillas blancas y casillas negras. Con esto sabremos de manera fácil el tamaño del kakuro y también el porcentaje de blancas respecto del total. Consideramos que como mas grande un kakuro, más difícil. Aunque para un ordenador puede no haber diferencia, para una persona es mucho más complicado completar un 4x4 que un 100x100, aunque las características sean parecidas. También, mientras más grande sea el porcentaje de blancas, más será la dificultad, ya que habrá más casillas a rellenar. También miramos como de largas son las runs de media, y nos apuntamos el valor de la run más larga.

Luego hacemos algo parecido a lo que se hace en resolver para encontrar casillas con valor único. Recorremos todo el tablero, y por cada casilla negra fijamos los valores de su run correspondiente, calculando las combinaciones de números que suman el valor de la casilla negra con tantos números como larga sea la run. Al igual que en resolver, en caso que la casilla haya sido tratada, hacemos una intersección con los nuevos valores y con los que ya tenia fijados. En este algoritmo, en vez de guardar las casillas únicas, simplemente sumamos uno a un contador por cada casilla única. Una vez recorrido el tablero, fijamos valores en aquellas casillas cuyo valor sea trivial, es decir, que todas las otras casillas de sun run tengan valores fijados. Esto también lo tenemos en cuenta para la dificultad, mientras más casillas con valor único tengamos al final de este proceso, más facil el kakuro. Después, mientras modifiquemos algún elemento, por cada casilla única propagamos restricciones igual que hacemos en resolver, es decir, si sabemos que una casilla tiene un 2, no puede haber un dos ni en la fila ni en la columna, y las casillas de fila y columna solo pueden tener numero que aparezcan en combinaciones de número en las que aparezca el dos.

Una vez hecho esto, contamos los valores posibles de cada casilla, y hacemos una proporción entre el número de casillas blancas y los posibles números. Como más números haya, más difícil será el kakuro. También se tiene en cuenta la media de largada de las runs y como de larga es la run más larga.

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.
Una matriz de 2 dimensiones posComb que guarda los valores posibles de las sumas de x casillas, siendo x un valor entre 0 (no hay posibles valores) y 9. Con esta matriz podemos saber los posibles valores de una run de x casillas.
Una matriz de 3 dimensiones mat, de tamaño 45x9x9, donde la casilla mat[x][y][z] nos indica si hay una combinación que sume x con y celdas y utilice el valor z+1. Esta matriz se inicializa antes de generar el tablero, y la usamos para evitar tener llamar a la función computePosSums para el mismo valor muchas veces.

Descripción del algoritmo:

Para generar, dividimos el algoritmo en dos partes: generar un tablero con algunas restricciones, y dar valores a las runs para que el kakuro sea único.

Como ya hemos mencionado, lo primero que se hace cuando se llama a generar es inicializar la matriz mat y asignarle los valores. Para esto, recorremos la matriz y calculamos computePosSums por cada posible suma y por cada numero de celdas, y guardamos el valor en la correspondiente "celda" de mat.
Por ejemplo, si queremos calcular los posibles valores que suman 45 con 9 celdas, llamaremos a computePosSums(45,9,0), que nos devolverá un vector de 9 posiciones con todos los valores a 1, y lo guardaremos en mat[45-1][9-1].
Una vez hecho esto, podemos empezar a generar el tablero.

Para generar un tablero, lo primero que hacemos es crear una matriz de Cell del tamaño indicado por el usuario, y inicializamos la primera fila y columna con celdas negras, puesto que obligatoriamente tienen que serlo. Ahora, generamos la segunda fila con la función firstColRow.

Esta función, recorre la primera fila y inicializa las celdas en blanca o negra, según los siguientes criterios: (1) si hay una celda blanca solitaria justo delante, esta celda obligatoriamente será blanca (cont = 1), (2) si hay 9 celdas blancas seguidas antes que una celda, esta celda obligatoriamente será negra, (3) si no se da ninguna de las condiciones anteriores, la celda será blanca o negra aleatoriamente, teniendo en cuenta el valor dificultad indicado por el usuario y evitando generar runs mayores que 3.

Como nosotros hemos decidido que nuestro tablero sea simétrico respecto a la diagonal, cuando inicializamos la primera fila, tambien estamos inicializando la última, por lo tanto, la casilla [1][1] será igual a la casilla [tamaño-1][tamaño-1].

Una vez hecho esto, inicializamos las demás celdas con la función randomCells. Para inicializar una celda como blanca o negra, seguiremos criterios parecidos a los anteriores, pero mirando que se cumplan tanto horizontal como verticalmente.
Ahora ya tenemos todo el tablero inicializado y nos queda comprobar que su estado actual es correcto. Lo primero que hacemos es comprobar que no hay runs de 9 ni horizontal ni verticalmente y, si se da el caso que hay alguna run de 9, repetimos el tablero de 0.

Una vez hecho esto, tenemos que mirar que no hay celdas blancas solitarias, es decir, runs de 1. Usamos un bucle while y un booleano canvi que nos indica cuando tenemos que parar. Cada vez que encontramos una celda blanca solitaria, la cambiamos por una negra y ponemos el booleano a true, por lo tanto el bucle while iterará siempre que en la iteración anterior se haya hecho alguna modificación.
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.
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, comprobamos que el número de celdas blancas del tablero no es inferior a 0.4*(tamaño del tablero), ya que de esta forma evitamos tener tableros con muy pocas celdas blancas.
Una vez hecho esto, si no tenemos que empezar de nuevo, quiere decir que el tablero actual podría ser válido, ahora solo nos queda comprobar si es posible tener un kakuro con solución única con este tablero.

Para generar una solución única, necesitamos la matriz tempBoard indicada anteriormente, en la cual, en caso de que sea una casilla negra 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, con las funciones nineCellsRow y nineCellsCol.

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, accedemos a la posición correspondiente de mat (matriz indicada anteriormente) y obtenemos los valores que estan presentes en las posibles combinaciones de números que suman el valor indicado y después comprobamos si solamente tienen un valor en común. De esta forma, miramos si para la primera casilla hay un valor posible, si no se da el caso, empezamos otro tablero, ya que si para ninguna combinación es posible encontrar un valor único, no es necesario seguir trabajando con ese tablero.
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 una cierta posición x e y, que dependerá del tipo de casilla que hemos resuelto.
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, y en caso de fijar un valor, la llamada recursiva indicará la posición 1,1, ya que queremos volver a mirar el tablero desde el principio.
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 estos dos casos, si fijamos un valor, la llamada recursiva se

hará con la posición i,j+1, ya que buscaremos una celda que este después de la que 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.