## 

## 

# Estructuras de datos y Algoritmos II

## Índice

[**Índice**](#_pqjog0h1v4ab) **2**

[**Algoritmo de generación de kakuros**](#_a2en5zg6mlk2) **3**

[Construcción de tablero](#_n2yvr26267gd) 3

[Rellenar tablero](#_lnciwq8rj33) 5

[Notas sobre la recursividad usada](#_2muwppb5o55e) 5

## 

## Algoritmo de generación de kakuros

Para la generación hemos dividido el proceso en dos grandes partes. La primera es la construcción del tablero, donde elegimos que casillas serán blancas y cuales negras. En la segunda parte se rellena el tablero y para cada casilla negra que deba tener suma total le ofrecemos una.

Debido a la gran dificultad del problema y el océano de casuísticas respecto a la detección de si un par de casillas contiguas convergerá al mismo número provocando un backtracking, se ha decido usar un algoritmo que sea más laxo en el control de la formación que el de resolución.

El resultado ha sido un algoritmo de backtracking para la resolución del tablero que tiene coste exponencial, pero como ya mencionado en las anteriores entregas, esto es fruto de los muy pocos recursos académicos que hemos encontrado, lo que también queda reflejado en las pocas oportunidades que hemos encontrado de mejorar sustancialmente el algoritmoCreacionSimple, que tiene ese nombre puramente como herencia de la segunda entrega, porque no ha sido tan simple como nos hubiera gustado.

### 

### Construcción de tablero

Esta parte del algoritmo consiste en definir dónde se van a localizar las casillas blancas y negras en el tablero. Respecto a la primera entrega aquí es donde se han producido los cambios importantes, ya que hemos añadido el patrón aleatorio que es el usado por el usuario, el resto son demostraciones de patrones simples que se pueden introducir como demostraciones estéticas. El caso del patrón diagonal, que es completamente funcional, es un claro ejemplo de que ciertos patrones constantes pueden ser interesantes.

La sofisticación de patronAleatorio está en el concepto de plantilla. A nivel teórico, sabemos que mientras se cumplan ciertas condiciones, podemos unir patrones de casillas blancas, y el resultado seguirá siendo un kakuro válido. Estos patrones básicos serían las plantillas. Dadas las normas que decidimos que debían cumplir los kakuros que aceptamos (no pueden tener casillas blancas que pertenecen a una tira de longitud menor a 2), la plantilla más pequeña es el cuadrado 2x2 de blancas. La unión de estos cuadrados, mientras no crea ninguna tira de longitud de más de 9, mantiene la correcteza del tablero. A pesar de que no hemos estudiado si ciertos patrones de tablero no tienen solución única porque todas sus soluciones son múltiples (demasiados grados de libertad).

Lo que nos gustaría que se atreviera es que teníamos el objetivo de demostrar que se pueden utilizar plantillas más complejas, y construir una serie de códigos análogos a los que se usan para el caso específico de la plantilla 2x2. Esto permitiría la creación de kakuros muy estilizados y con gran nivel de detalle a cambio de programar las plantillas, que no es tan difícil y es relativamente escalable.

### Rellenar tablero

Una vez dispuestas las casillas en su sitio, debemos poner a todas las casillas blancas un valor de solución que es ficticio. Apartir de este, encontramos unos valores de suma de tira para los padres negros. Y con estos valores de suma en los padres negros llamamos al algoritmo de resolucion para comprobar si tiene solucion unica, que correspondera en el caso de que exista con la solucion que se ha generado previamente.

#### Notas sobre la recursividad usada

Para asegurar resultados aleatorios (que sea “muy difícil” generar 2 kakuros iguales), el backtracking tiene que usar algún tipo de aleatoriedad. En nuestro caso, el árbol que construyen las recursividades representa por cada nodo una casilla blanca, y cada hijo es una posible asignación de valor a esa casilla blanca. El primer elemento de randomización es desordenar el orden en que visitamos los hijos, así en vez de empezar con un kakuro lleno de 1’s e ir probando, en cada nivel el orden en que se visitan las posibilidades es aleatorio. Similarmente una manera de hacer vagamente mejor el algoritmo, es que los “niveles” o en qué profundidad asignamos valores a una casilla, también sea aleatorio (evitar bias por posición).

Así pues al terminar de probar todas las combinaciones de una casilla, su padre pasará a la siguiente llamada recursiva. Pero como padre e hijo son relaciones aleatorias, los efectos que tiene llamar recursivamente con el siguiente valor del padre, alteran por completo el kakuro entero. Si la recursividad tuviera relación con las coordenadas, los cambios serían mucho más locales, lo que sería problemático en el caso medio. Es preferible alterar partes aleatorias del kakuro que enfocarse en una zona local, porque tienes más posibilidades de por suerte resolver el conflicto que impide tener solución única. Si cambias localmente, quizás tienes mala suerte y las casillas problemáticas están en las primeras capas de la recursión, obligándome a explorar muchas posibilidades. En resumen, eliminamos bias negativo que puede aportar la ordenación de la recursividad en función de las posiciones de las casillas blancas en el tablero.

Si ninguna de las soluciones ficticias retorna que tiene solución única, esa casilla itera recursivamente por todos los valores de esa tira de números candidatos, descartando los que no han funcionado. Aquí es donde entran en juego 2 conceptos. El primero es la poda, si el número que se ha asignado esa casilla ya lo tiene asignado alguna casilla hermana (que pertenece a la misma tira, vertical o horizontal) esa rama se poda y no se recorren todas las posibilidades que nacen de esa decisión. El segundo concepto es el de poda ciega o “loteria” (“como de afortunado te sientes”). Al crear el vector de números candidatos, podemos crearlo con todos los números {1,2,3,4,5,6,7,8,9}, o podemos hacerlo con un subconjunto aleatorio de estos. El scroller de generación de kakuros ajusta este parámetro, en que el usuario puede decidir que solo se probaran 4, o 7, etc, números por cada casilla blanca. Esto degrada el algoritmo a uno no determinista, pero ofrece importantes ventajas de velocidad, porque está decreciendo la base de la exponente (el árbol pasa de tener 9 ramas por tronco a un número menor).

## Resumen función a función de algoritmoCreacionSimple

En orden de implementación:

-crearKakuro(): esta función se encarga de, dados los parámetros, intentar crear inicialmente el patrón que se ha pedido (si los valores de altura y anchura son correctos). Si al llamar al backtracking este retorna que no se puede resolver ninguna combinación de números sobre las casillas blancas que genera unas sumas (un kakuro) con solución única, el algoritmo salta a un comportamiento predeterminado que selecciona el patrón aleatorio e intenta un número limitado de veces seguir intentando generar un patrón e intentar rellenarlo (controlado por loop\_cap, que pretende ser una abreviatura de loop count cap). Si tras esto sigue sin poder encontrar un kakuro como encargado, se rinde y devuelve un kakuro vacío.

Otras variables que merecen una mención es la q, que es la dificultad de la lotería, o la cantidad de números que tendrá cada casilla en su árbol de recursión (9 son todos, convirtiendo el algoritmo en determinístico). Test es un bool que se usa para comprobar si se ha encontrado una solución y progress y r son variables que durante el proceso de debugado se usaron para medir cuán rápido progresaba el algoritmo.

-patronBlancoNegro(): selecciona el patrón encomendado y aleatoriamente mezcla las casillas blancas para que no haya preferencia en la construcción del árbol recursivo mas adelante.

-patronCuadrado/Escalera(): se trata de patrones estéticos heredados de la segunda entrega, como muestra de que se pueden pedir patrones predefinidos al detalle. Nada notable que comentar debido a su poco uso en la aplicación (uso nulo).

-patronAleatorio(): Se encarga de usar la plantilla del cuadrado 2x2 para generar patrones aleatorios. Representaremos la presencia de casillas blancas o negras en un boolean map, true para blanca, false para negra. Y como la primera fila y columna siempre son casillas negras, las omitimos para la declaración de la variable Map. Este concepto de mapa lo aplicaremos también para marcar la presencia o ausencia de la plantilla. En este caso el mapa de la plantilla es proporcionalmente menor a Map debido a su tamaño. Si se usaran plantillas más grandes (que podrían ser estéticamente más refinadas, e incluso generadas a partir de plantillas más pequeñas, generar una biblioteca de plantillas etc…) entonces se podría crear un mapa para cada plantilla usada. Y por cada plantilla usada es necesario una función que pase de mapa de plantilla a Map (pharsePlantilla2x2).

-generate2x2Map(); esta función genera un mapa válido de la plantilla con la limitación de nunca sobrepasar el número de casillas blancas solicitado (upperbound). El proceso es el de mediante una función que es proporcional al tamaño del kakuro solicitado generar (con algo de aleatoriedad para hacerlo más divertido) generamos un número de veces que intentaremos añadir la plantilla al Map. Aquí podremos reprogramar la función para crear kakuros más o menos esparsos, hemos decidido hacerlo poco esparso para que la condición de casillas blancas máximas se comporte como una buena aproximación, si es posible, de la cantidad de casillas blancas que el kakuro generado tendrá. Para verificar que nunca se pasa, mira por cada intento de añadir una iteración de la plantilla, cuantas casillas blancas está añadiendo, y eso sumado al número de casillas ya presentes en el kakuro le permiten verificar la condición. Pero eso no es todo, por cada plantilla añadida tenemos que revisar que no hemos creado ninguna tira de más de 9 blancas seguidas. Para eso llamamos chackMap estratégicamente en el menor número de casillas que hemos añadido al insertar la iteración de la plantilla, en nuestro caso, cualquier par de casillas esquina opuestas son suficientes para comprobar que ninguna casilla añadida pertenece, verticalmente o horizontalmente a una tira de más de 9 casillas blancas. Finalmente usamos variables locales para restaurar rápidamente el valor anterior de las casillas modificadas si la inserción falla.

-pharsePlantilla2x2(): una función simple que transforma de Map de plantilla a Map que se usa para generar el kakuro. (esta función sería específica para cada plantilla)

-checkMap(): dado un mapa y un par de coordenadas, comprueba (asumiendo que las coordenadas son de una casilla blanca) que no se trata de una tira vertical o horizontal de más de 9 blancas. Esto se calcula mirando en las cuatro direcciones (por encima, por bajo, a la derecha, izquierda) de las coordenadas si hay tiras de más de 9, o si la suma de direcciones opuestas es más de 9 (sin incluir la casilla de la coordenada x y , menor que 8). Para simplificar el while se externaliza a la función inRange() la tarea de mantener las coordenadas que se iteran dentro de rango. (esta función es general para todas las plantillas posibles)

-inRange(): devuelven si unas coordenadas están en rango de un Map. (esta función es de uso general/reutilizable)

-pharseBinMap(): esta función se encarga de transformar un Map (ya de kakuro, no de plantilla) en el tablero, como hacían los patrones Escalera, pero esta vez a partir de el Map, y no ninguna regla. (esta función es muy versátil y la clave de que se puedan hacer todas las mejoras creativas a cómo se generan los maps, ya que luego solo hay que llamar a esta función que terminara el trabajo)

-printMap(): función usada durante el desarrollo, obsoleta o no usada al final.

-inicializarKakuro(): esta función tiene la tarea fundamental de recoger el tablero creado por los patrones y entrelazar las tiras, actualizando la información de las casillas negras para que correspondan con sus tiras, y actualizando las casillas blancas para que sepan quienes son sus padres.

-backtracking(): función recursiva que constituye el árbol de combinaciones que se intentarán rellenar en las casillas blancas para encontrar (posiblemente) una que tenga solución única. El caso base es si ya se han creado y asignado números en el vector solución igual al número de casillas blancas, entonces podemos hacer una asignación uno a uno. Una vez asignados estos valores, comprobamos si tiene números repetidos (código que se ha heredado de cuando no exista la poda de números repetidos). Si no tiene números repetidos, comprobamos la condición más fuerte de si tiene solución única llamando a algoritmo resolución. En el caso de que el veredicto no sea que tiene solución única, se restaura la solución que se estaba probando. Esto es heredado del debug en que mirábamos como cambiaba las distintas maneras en que se alteraba el kakuro en esta recursividad.

El caso recursivo es que queden nodos del árbol por crear (que no estemos en la profundidad de recursividad igual al número de casillas blancas). En este caso creamos una permutación de 1,2,3,4,5,6,7,8,9 aleatoria. Podamos ciegamente como solicitado por el usuario. Aquí está heredado del proceso de testeo intentar calcular las sumas dinámicamente para no tener que calcularlas en el caso base antes de llamar al algoritmo de resolución. La llamada recursiva se encuentra en un while en que iteramos por todos los valores de la permutación que no hemos quitado. Pero antes de llamarla miramos si hay que podar la llamada recursiva.

Antes del return final se imprimía progreso si estábamos en la profundidad de recursión deseada para imprimir información (la variable r).

-calcularSumas(Kakuro): Esta función calcula las sumas (una vez los números ficticios han sido asignados a los campos numeroSolucion de las casillas blancas). En el caso de que encuentre un numero repetido retorna falso. Si no encuentra ningun problema (todos los valores son nominales y no hay repetidos) devuelve true y las casillas negras del kakuro tienen la suma calculada.

-podarSiRepetidos(): Esta función revista la tira vertical de una casilla negra y la tira horizontal de otra casilla negra, efectivamente revisando si una casilla blanca con esos dos padres pertenece a una tira con números repetidos. En ese caso retorna cierto indicando que se debe podar esa rama. En caso de que tenga casillas sin numeroSolucion asignado, las ignora.