# 

# 

# Estructuras de datos y Algoritmos I

## Índice

[Estructuras de datos y Algoritmos](#_ifoqoodq2s2g) 1

[Índice](#_95gsor67gq05) 2

[Algoritmos de Kakuros investigados](#_fy391r4osgcj) 3

[Algoritmos de resolución de Kakuros:](#_25nqlaymyfl2) 3

[Algoritmos de backtracking [1]:](#_f1biarpiifsl) 3

[Dividir y vencer [2] [3]](#_tub0ypst8tkh) 5

[Algoritmo de validación/resolución de kakuros](#_y1tnpj329gmj) 6

[Términos y conceptos](#_9u5jfd2gv9js) 6

[Estructuras recursivas vs iterativas](#_hbiia3g7jp7x) 7

[Deducción iterativa](#_34a5zkmm4w0x) 8

[Correcteza del algoritmo](#_czidony1tf66) 9

[Historial de debugging y mejoras](#_wn51kohy5yyd) 9

[Resumen función a función de la implementación](#_1f446okijlfe) 10

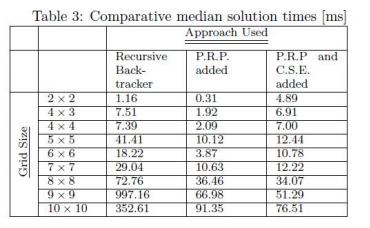
[Bibliografía](#_kg1i21rq60wz) 13

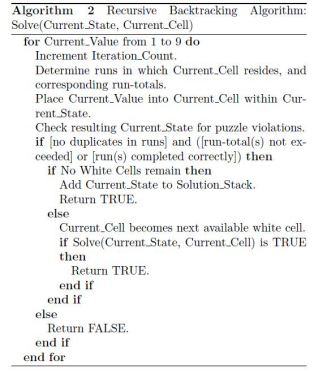
## Algoritmos de Kakuros investigados

Resumiendo las investigaciones ya presentadas, se han utilizado diversos recursos, algunos de ellos citados en la bibliografía para estudiar los posibles algoritmos en los que se puede basar y sobre la que se puede construir nuestra implementación, además de considerar las distintas eficiencias de estos.

## Algoritmos de resolución de Kakuros:

### Algoritmos de backtracking [1]:



**PRP:**

Las verificaciones de validez, denominadas Poda de ejecución proyectada (Projected Run Pruning en inglés), se agregan al algoritmo de backtracking recursivo. Al asignar un valor a una celda en una ejecución que aún posee celdas no asignadas, se realiza un cálculo de la suma de los valores más grandes posibles que todavía pueden ser legítimamente agregado a las celdas restantes de esa ejecución. Si esta suma produce un total de ejecución que al menos coincide con el total de ejecución especificado para esa celda, el backtracker continúa; de lo contrario, esta rama no útil del espacio de búsqueda se poda y se produce el backtracking.

**CSE:**

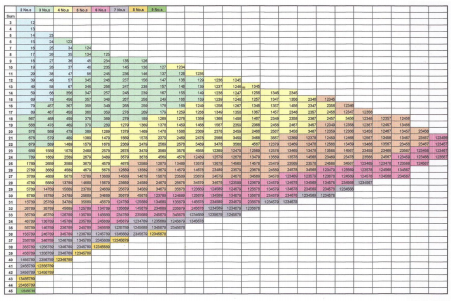
En un enfoque de ordenación de celdas de eliminación de conjuntos candidatos(Candidate Set Elimination), las comprobaciones para la poda de ejecuciones proyectadas se extienden para que el "Valor actual" no se asigne a la "Celda actual" a menos que sea un miembro de la intersección de los conjuntos candidatos de las dos ejecuciones en qué "celda actual" reside. El "Valor actual" aumenta hasta que se alcanza un valor válido o el "Valor actual" se convierte en 9. Además, si una celda tiene solo un valor en la intersección de sus conjuntos candidatos, el "Valor actual" ahora "salta" automáticamente al valor requerido. Por lo tanto, muchas ramas no útiles del espacio de búsqueda se podan debido a la ausencia de ciertos valores en esa intersección. Se espera que la eliminación de conjuntos de candidatos, en combinación con la poda de ejecución proyectada, disminuya aún más el número de iteraciones necesarias para encontrar una solución y, por tanto, el tiempo total de solución necesario. Sin embargo, los gastos generales de procesamiento de la eliminación de Candidate Set no deben tener un efecto perjudicial en la velocidad general del algoritmo; tales gastos generales pueden anular cualquier efecto beneficioso de las reducciones del recuento de iteraciones.

### Dividir y vencer [2] [3]

Este enfoque consiste en dividir el puzzle en trozos más pequeños, si es posible, encontrando “casillas blancas de enlace”, cuyos valores se pueden hallar fácilmente. Estas casillas blancas de enlace son casillas que conectan un trozo del puzzle con el resto del puzzle y, al eliminarla, causaría que el kakuro se dividiera en dos partes. Para rellenar las casillas blancas de enlace que estén en la columna debemos sumar las sumas totales de las casillas que están en la zona vertical y restar las sumas totales de las horizontales, es el valor que tiene esa casilla en valor absoluto.

## Algoritmo de validación/resolución de kakuros

Para realizar la resolución de kakuros, se ha desarrollado un algoritmo basado en lo que llamamos secuencias, que consideran las posibles combinaciones de números del 1 al 9 sin repeticiones para alcanzar una suma dada empleando n (casillas). En la siguiente figura se pueden ver las distintas secuencias posibles según la suma total y el número de casillas (color de las celdas):



### Términos y conceptos

Los conceptos que hemos usado para nuestra solución son que cada casilla negra tiene una “tira” de casillas blancas como “hijas”, y esas casillas blancas de la “tira” tienen a la casilla negra como padre. Entonces cada casilla blanca decimos que tiene un padre vertical y horizontal. Cada casilla negra puede tener hasta 2 tiras de casillas blancas, una vertical y una horizontal.

Cada “tira” tiene ciertas secuencias que son válidas porque una permutación o varias de esa secuencia “encajan” en la tira.

### 

### Estructuras recursivas vs iterativas

Comparativamente, si el algoritmo recursivo CSE recorre un árbol de posibles respuestas que vienen restringidas por los candidatos, nosotros decidimos encarar el problema de manera determinística/iterativamente. Crear un estado del tablero (kakuro) que representará vagamente uno de los nodos del árbol.

Si éramos capaces de tener suficiente información sobre qué posibilidades tenemos para cada casilla y para cada secuencia, y luego establecemos algún tipo de deducción que permita asegurar que siempre “avanzas” a un estado más cercano a la solución, podemos iterar estas “deducciones” hasta llegar a encontrar la solución o encontrar que no podíamos hacer más deducciones. En el segundo caso, posteriormente tendríamos que demostrar que significaba que el kakuro propuesto o no tenía solución o tenía múltiples.

Más concretamente, decidimos que nuestro algoritmo no tenía que tomar decisiones. Analizando iterativamente una serie de deducciones (obviamente no polinomiales, pero quizás eficientes) iríamos poco a poco deduciendo la solución. Un motivo para elegir esta aproximación fue fruto de resolver muchos kakuros y ver que estábamos (como humanos) usando una serie de “teoremas” determinísticos para deducir partes de la solución.

Algunas de las dificultades que acompañaron a la implementación de este algoritmo fueron las modificaciones que recibieron otras clases (y sus drivers como consecuencia) para acomodar y facilitar la implementación de este, además de la creación de clases nuevas necesarias para realizarlo. Durante todo el desarrollo se dio una evolución continua de las clases implementadas.

Las clases que más se vieron afectadas fueron Casilla y las clases que heredan de ella (Blanca y Negra). Esta clase pasó de ser una clase sencilla con unas coordenadas i,j, y un contenido basado en si esta era blanca, negra, o vacía; a incorporar todo un conjunto de información como las secuencias posibles en cada tira y los candidatos en cada casilla para poder emplearlo en el algoritmo. Otro ejemplo de modificaciones de las clases existentes fue que en la clase Kakuro se añadió una lista con sus casillas negras utilizadas para determinar las secuencias en cada tramo, y permitir calcular dinámicamente los candidatos de las casillas blancas.

### Deducción iterativa

Una vez tuvimos el scheduler el siguiente paso era la “deducción”.

El filtrado de casillas blancas fue simple. Definimos una serie de funciones que calculaban la intersección de candidatos de los padres, y restaban a este conjunto la unión de los elementos ya “convergidos” (elementos ya colocados). De esta manera, “tus” candidatos son (intersección de) los candidatos de tus padres excluyendo los números que ya han sido colocados. Luego si tu set de candidatos cambia (como casilla en alguna deducción), tienes que empilar tus padres para que se actualicen en algún momento.

Luego pasas a mirar los casos base de candidatos casilla. Si tienes solo un candidato te asignas ese candidato como numero solución, y actualizas el EnumSet de convergidos de tus padres añadiendo te. Si no tienes ningún candidato, debes notificar que no es posible resolver el kakuro, ya que no hay número que puedas poner en esa casilla.

Cabe resaltar que tras empilar cualquier cosa, la ejecución de la deducción no se interrumpe, simplemente al terminar la ejecución de la deducción, se desempilará el primer elemento, y se ejecutará una deducción sobre él. Nada de deducciones en paralelo o recursivas, todo iterativo, una por una, una tras de otra. Finalmente, tras iterar por todas las casillas blancas de la tira, llegamos al corazón del algoritmo, que es la comprobación (por recursividad) de las secuencias posibles para esa tira. Aquí es donde se oculta el coste NP del algoritmo, y es el último paso de cualquier deducción.

Filtrado de las secuencias candidatas

Como primera aproximación decidimos hacer un filtro que comprobase recursivamente si existe una permutación de la secuencia que estamos analizando que fuera compatible con los candidatos de las casillas. Es decir, dados los candidatos de cada casilla blanca de la tira, si existe una permutación de la secuencia que “cumple con los candidatos”. De no ser así, significaba que tal cadena no era posible, y debemos quitarla de cadenas candidatas para esa tira. Al final de comprobar todas las secuencias, con las que siguen siendo posibles, hacer la unión de todos sus números, y usar ese conjunto como candidatos de esa casilla negra. Como es un poco abstracto, pondré un ejemplo rápido.

Una tira de 2 casillas con suma 5. Inicialmente sus cadenas candidatas son {{1,4},{2,3}}, y sus candidatos iniciales eran entonces el conjunto unión {1,2,3,4}. Pero al analizar las permutaciones de la secuencia {1,4}, nos damos cuenta que no hay ninguna casilla con el 1 por candidato, así que esa cadena ya no es posible y la eliminamos. Ahora las cadenas candidatas son {{2,3}} (porque imaginemos que sí que existe una permutación compatible), y su conjunto candidato ahora es {2,3}. Creo que intuitivamente se entiende pero adjuntamos una documentación completa de la versión entregada del algoritmo.

### Correcteza del algoritmo

Dado que ahora ya estamos en el final del proceso de creación de este algoritmo, podemos hacer un poco de retrospectiva sobre por qué funciona.

La clave esta en la relación reciproca entre los padres Negros y los hijos Blancos de una tira. Cada deducción cuenta con dos pasos, como se ha explicado, en uno, la información de los padres actualiza la de los hijos, restringiendo y acercándonos a la solución. En el segundo paso, lo inverso ocurre, y la información de los hijos es la que se utiliza para comprobar y actualizar información sobre los padres. Esta relación recíproca es el corazón del algoritmo, que se encarga de verificar que efectivamente llegamos a un estado final en que ya no se hacen más deducciones que generen nueva información (cosa a la que el algoritmo presta mucha atención). Una vez las deducciones no pueden generar mas información, toca simplemente comprobar en que estado esta la solucion para determinar si el kakuro tenia solucion, si tenia multiple solucion (no ha colapsado), no tenia solucion (casilla blanca sin candidatos), etc…

### Historial de debugging y mejoras

De las mejoras que hemos ido arrastrando durante el proyecto, una muy notable ha sido el filtro polinomial para las secuencias.

Si bien hemos intentado implementar, sin éxito esta mejora, quedará para la posteridad su versión como código comentado en algún rincón de el .java del algoritmo de creación.

La idea principal no es difícil de explicar. Antes de hacer el backtracking de las secuencias de una casilla negra, para encontrar cuales no son posibles en esa tira, es posible comprobar una serie de condiciones necesarias pero no suficientes. Me explico con un ejemplo.

Pongamos el caso de la tira de dos casillas blancas de suma 5. Una de las secuencias que han sobrevivido hasta esta deducción pongamos que es la {1,4}.

El primer filtro es intuitivo, “cada casilla blanca de la tira tiene que contener como mínimo 1 elemento de la secuencia”, si no, existe una casilla que no podrá contener ningún número que satisface la tira, haciendo la tira invalida. La implementación era una sencilla interacción entre el conjunto de números de la secuencia y los candidatos de cada casilla blanca. Si una de estas intersecciones era vacía, debemos calificar esa secuencia como imposible en esa tira y simplemente eliminarla.

Si la casilla primera de la tira tenía por candidatos {2,3,7}, da igual el resto de la tira, podemos directamente ahora determinar que la tira no puede tener la secuencia {1,4}.

El segundo filtro es algo más difícil porque es el inverso del primero. “Todos los números de la secuencia deben aparecer como mínimo una vez como candidatos de alguna casilla blanca”. Saltaré directamente al ejemplo.

Si las casillas blancas c1 y c2 tienen por candidatos {1,5} {1,2,3,5}, la secuencia {1,4} es imposible porque no aparece nunca el 4 como candidato de ninguna casilla.

La implementación de este filtro se intentó con la resta de los números de la secuencia menos (en operación de conjuntos) la unión de todos los candidatos de todas las casillas. Entonces si el resultado de esta operación no daba el conjunto vacío, era porque un elemento no “había sido restado”, indicando que no aparecía en ningún conjunto de candidatos de ninguna casilla blanca.

Por desgracia, y por falta de tiempo, este código tenía errores o no se incorporó bien al resto del algoritmo y resultaba en veredictos incorrectos en que kakuros que tenían solución resultaban en el veredicto “no tiene solución”. Por motivo de mala implementación o algún efecto lateral la condición era demasiado restrictiva. Por suerte estuvimos a tiempo de quitarla para que el algoritmo de resolución continúe siendo correcto como en las entregas anteriores.

### Resumen función a función de la implementación

En orden de implementación:

-**void** algortimoValidacionInicializacion(kakuro): Se encarga de sobrescribir ciertos parámetros de las casillas para que el kakuro esté listo para el algoritmo.

Asigna a las casillas negras las secuencias candidatas dados sus longitudes de casillas blancas (longitud de las tiras que encabezan), y el número que deben sumar esas tiras.

Si tienen 0 casillas blancas o suman 0 en alguna tira no tienen secuencias candidatas para esa tira. Como segundo paso exploramos las Casillas blancas explorando todo el kakuro (podríamos intentar ir a través de los padres, y de hecho es una de las raras veces en que no lo hacemos). Si la casilla blanca no es modificable significa que venía con un número asignado, que el algoritmo confía que es correcto y añade a la lista de números ya convergidos de sus padres. (los números convergidos se excluirán de los candidatos de cualquier casilla blanca hermana). Si la casilla blanca no es modificable entonces sobreescribimos NumeroSolucion a -1, un efecto lateral que tendrá que tener en cuenta el algoritmo de Creación.

Finalmente, como ahora tenemos información sobre las casillas colapsadas, hacemos un filtrado inicial de las tiras de todas las casillas negras, y como está comentado en el código, esto se debería quitar si nunca se va a usar o casinunca se usan numeros ya fijados en el kakuro. Empila todas las casillas negras par que queden por hacer.

-**int** algoritmoValidacion(kakuro): Esta es la función principal, y lo primero que hace es llamar a la inicializadora. Declara finished qué es el int que devolverá con el veredicto (en el código hay comentarios explicativos que complementan esta guía).

Luego entra en el while principal en que está a la espera de que el veredicto se concrete (deje de ser 0, el valor default) o se quede sin trabajo que hacer (las deducciones no generan nueva información).

Si sale de este bucle principal, mira si ha salido de él por algún motivo que no era no tener solución (tenia solucion, tener más de una solución) en ese caso, llama a la función comprobarSolucion, que en tiempo polinomial verifica que los veredictos de verdad han sido bien encontrados, y que la solución propuesta es correcta. Sirve para asegurar la correcteza del algoritmo y discernir entre distintos veredictos.

El bucle principal consiste en ejercer una deducción sobre una casilla negra que estaba empilada. La deducción se ejecutará simétricamente para su tira vertical como su tira horizontal, independientemente de si ha sido empilada para que se revisara su tira vertical o horizontal. Esto es fruto de la manera en que hemos elegido integrar las tiras en las casillas negras y por cómo empilamos (queríamos no empilar más información, pero se podría haber hecho).

Por la tira vertical y horizontal iteramos sobre sus casillas blancas, y si son modificables, ejecutamos la comprobación de Casilla Blanca sobre ellas, que se encarga de hacer la deducción de casillas blancas.

Una vez terminada todas las deducciones de casillas blancas, llamamos al filtro de secuencias para deducir sobre la casilla negra. Si este proceso devuelve que se ha conseguido descartar secuencias, empilamos otra vez este padre para que en un futuro se vuelva a ejecutar las deducciones sobre las blancas, completando el ciclo.

Una vez esto, se termina la deducción y se desempila la siguiente negra que esté pendiente de hacer. Si al desempilar una casilla negra esta ya estaba marcada como hecha, quiere decir que no se han producido cambios en esta casilla desde la última vez que la hemos desempilado y podemos descartar esta deducción como redundante.

Cabe destacar que lo que retorna la función es un veredicto, el resultado de la solución está almacenado lateralmente en los campos numeroSolucion de las casillas blancas del kakuro.

-**int** comprobarCasillablanca(CasillaBlanca, Kakuro): Esta función se encarga de ejecutar una deducción sobre una casilla blanca a partir de la información de sus padres. Así que lo primero que hace es buscar a los padres en el kakuro mediante la información que tiene guardada (este paso podría saltarse si la función principal le enviase directamente los padres como parámetro, pero asi es mas modular pero no tan eficiente). Una vez tiene la información de quienes son sus padres, les pide que candidatos tienen como posibles para ella, y ejecuta la función calcularCandidatos dados estos dos conjuntos. El resultado de esta llamada se inspecciona para saber si ha actualizado la casilla, en cuyo caso los padres deberían ser empilados. Esto es crucial porque se empilan los 2 padres, no solo el que ha llamado para hacer la deducción de esta casilla blanca. Una vez hecho esto, se comprueba si la casilla ha colapsado, es decir si este conjunto nuevo de candidatos tiene tamaño 1, o si tiene tamaño 0 y la casilla no puede resolverse, que hace retornar un 3 al bucle principal en la variable finished, que como discutido lleva a el veredicto “el kakuro no tiene solución”.

-calcularCandidatos(Casilla, Casilla, int, int)

Dado los dos padres y la posición relativa respectivamente en que se encuentra la casilla blanca que ha llamado a esta función, se retorna la intersección de los candidatos de cada padre para esa casilla.

-Backtracking esta sección del código son la función de inmersión y la recursiva que se encargan de dada una casilla negra filtrar las secuencias que son válidas comprobando recursivamente que por cada secuencia existe una permutación válida que se puede encajar con los candidatos de las casillas blancas de la tira. aquí es donde está el código comentado del filtro polinomial que no llegó a entrar en la versión funcional del algoritmo.

-comprobarSolucion(Kakuro,Pair): Se trata de una implementación fácil en que se comprueba que todas las tiras suman lo que deben sumar, no hay casillas blancas con números no válidos en su numeroSolucion, y no hay repetidos en las tiras. Se devuelven veredictos a través del int que se devuelve. El Pair por parámetro se usó en algún momento del desarrollo para devolver la posición de la casilla que tenía multiple solucion. Esto último pertenece al desarrollo de algoritmoCreacion que se explicó en la segunda entrega.

# Bibliografía

[1] Ryan P. Davies, Paul A. Roach, “The Use of Problem Domain Information in the Automated Solution of Kakuro Puzzles”, 2010.

[2] Tips and tricks: easy ways to solve Kakuro. Nov. 18, 2020. [Online] Disponible en:

https://theory.tifr.res.in/~sgupta/kakuro/simple.html#divide

https://theory.tifr.res.in/~sgupta/kakuro/simple.html#eliminate

https://theory.tifr.res.in/~sgupta/kakuro/simple.html#non unique

https://theory.tifr.res.in/~sgupta/kakuro/simple.html#partitions

[3] Oliver Ruepp, Markus Holzer, “The Computational Complexity of the Kakuro Puzzle, Revisited”, 2010.

[4] Helmut Simonis, “Kakuro as a Constraint Problem”, 2008.

[5] Ryan P. Davies, Paul Alun Roach, Stephanie Perkins, “Properties of, and solutions to, Kakuro and related puzzles”, 2008.

[6] “La tabla del Kakuro”. Feb 6, 2007. [Online] Disponible en:

http://wvw.nacion.com/viva/kakuro/tablakakuro.html

[7] “Creating a Kakuro puzzle with unique solution”, usuario paramesis en stackexchange [Online], 2017. Disponible en:

https://puzzling.stackexchange.com/questions/49927/creating-a-kakuro-puzzle-with unique-solution