Estructuras de datos y Algoritmos

Índice

Algoritmos de Kakuros investigados

Antes de la implementación de los algoritmos confeccionados para la resolución y generación de kakuros, 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]:

Table 3: Comparative median solution times [ms]

		Approach Used		
		Recursive Back-tracker	P.R.P. added	P.R.P and C.S.E. added
Grid Size	2×2	1.16	0.31	4.89
	4×3	7.51	1.92	6.91
	4×4	7.39	2.09	7.00
	5×5	41.41	10.12	12.44
	6×6	18.22	3.87	10.78
	7×7	29.04	10.63	12.22
	8×8	72.76	36.46	34.07
	9×9	997.16	66.98	51.29
	10×10	352.61	91.35	76.51

Algorithm 2 Recursive Backtracking Algorithm: Solve(Current_State, Current_Cell)

```
for Current_Value from 1 to 9 do
  Increment Iteration_Count.
  Determine runs in which Current_Cell resides, and
  corresponding run-totals.
  Place Current_Value into Current_Cell within Cur-
  rent_State.
  Check resulting Current_State for puzzle violations.
  if [no duplicates in runs] and ([run-total(s) not ex-
  ceeded] or [run(s) completed correctly]) then
    if No White Cells remain then
      Add Current_State to Solution_Stack.
      Return TRUE.
    else
      Current_Cell becomes next available white cell.
      if Solve(Current_State, Current_Cell) is TRUE
      then
        Return TRUE.
      end if
    end if
  else
    Return FALSE.
  end if
end for
```

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

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.

Algoritmos de generación de Kakuros:

Algunos criterios en cuanto a la generación de kakuros se hallan en algunas de las fuentes empleadas. [3] [6]

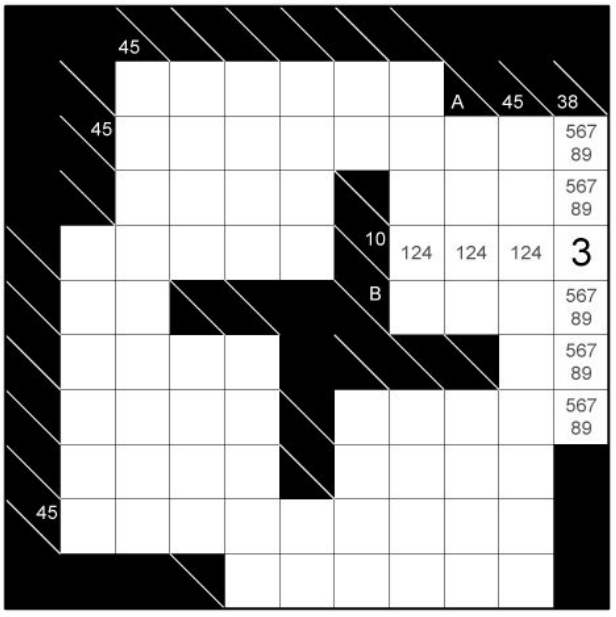
A continuación comentamos algunos de los posibles pasos a seguir para realizar una generación adecuada.

Patrones para creación:
Usar patrones con soluciones únicas para simplificar la creación. Podemos establecer las sumas de algunas casillas negras para que solo nos den una opción posible en su intersección con otras casillas negras (e.g: la suma 4 en 2 casillas y la suma 7 en 3 casillas solo tienen en común el número 1).

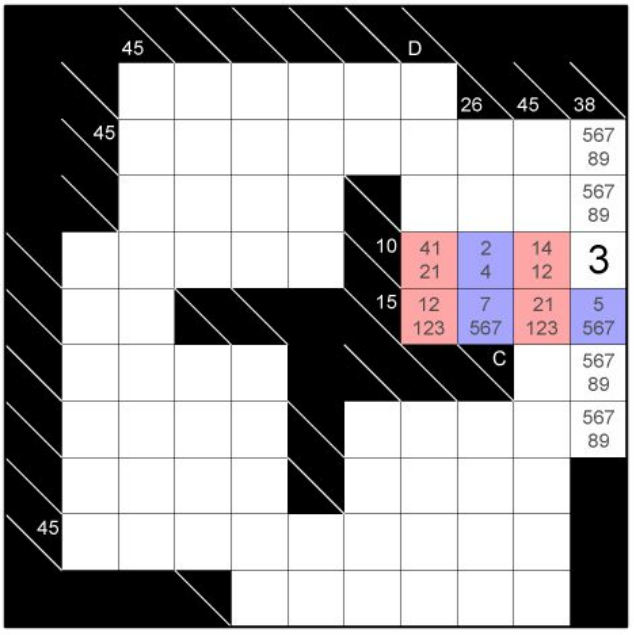
Backtracking:
Se utiliza un algoritmo de elección en el que a cada casilla negra se le da un valor coherente (el valor dado debe ser válido con las otras casillas negras) y cuando no hay más opciones se hace backtrack hasta dar con una solución válida (habrá que comprobar si tiene una única solución).

Dicho esto, la principal fuente empleada para investigar y diseñar el algoritmo de generación de kakuros es una entrada en la página web de stackexchange que detalla paso a paso un posible proceso de creación de uno de estos puzles [7], aunque posteriormente se han ideado e implementado algunos enfoques propios para esta primera entrega tal y como se comenta más adelante.

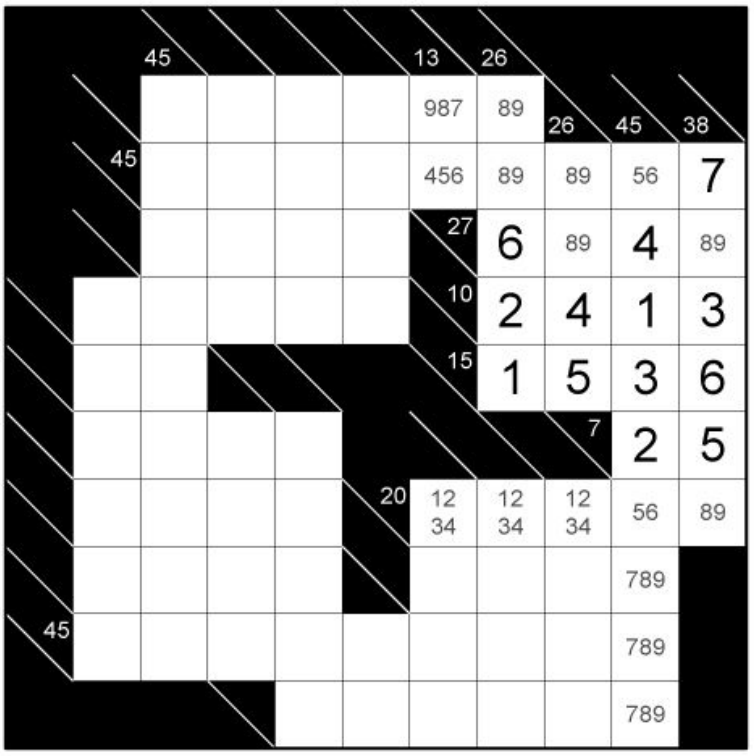
La principal idea de este procedimiento es, partiendo de una casilla en concreto sobre la que se fija un valor, y se elige entre las posibles combinaciones de secuencias con el mismo número de casillas que el tramo que contengan el número elegido, los valores para la suma vertical y horizontal de sus tramos, anotando para el resto de casillas los posibles valores candidatos.



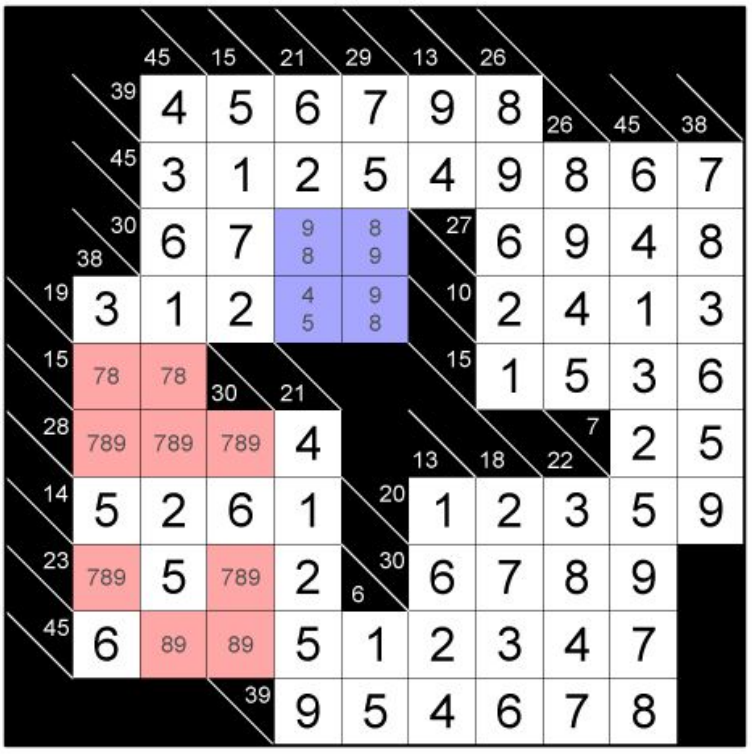
Una vez hecho esto, se estudian los posibles valores de las sumas de las filas y columnas adyacentes considerando los candidatos de las casillas ya marcadas:

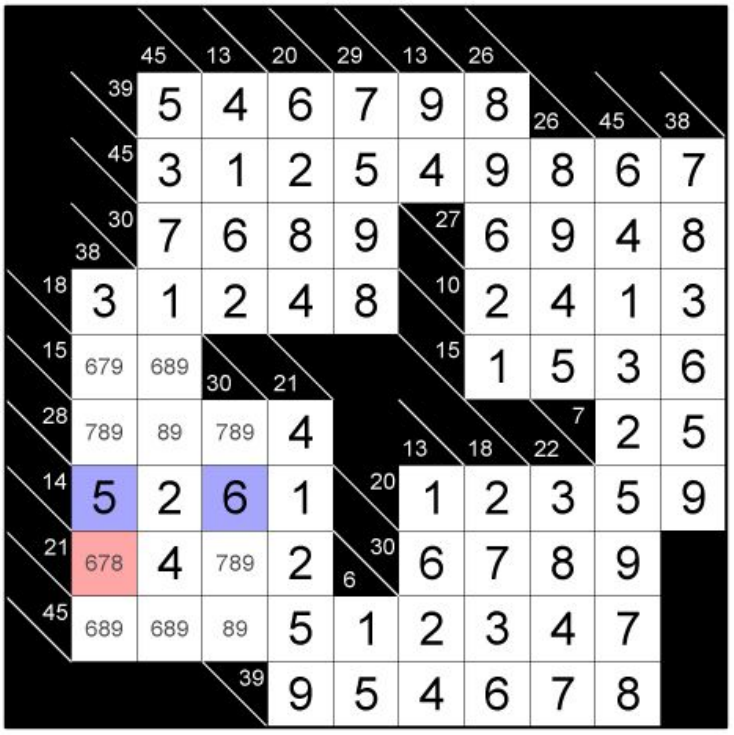


Este procedimiento se realiza hasta completar el kakuro, mientras que, a medida que se va rellenando el kakuro, las restricciones añadidas causan que los valores que puede tomar cada casilla blanca se vean reducidos hasta quedar un solo valor, y entonces consideramos que el valor de esa casilla ha convergido a la solución (si no se da backtracking posterior).

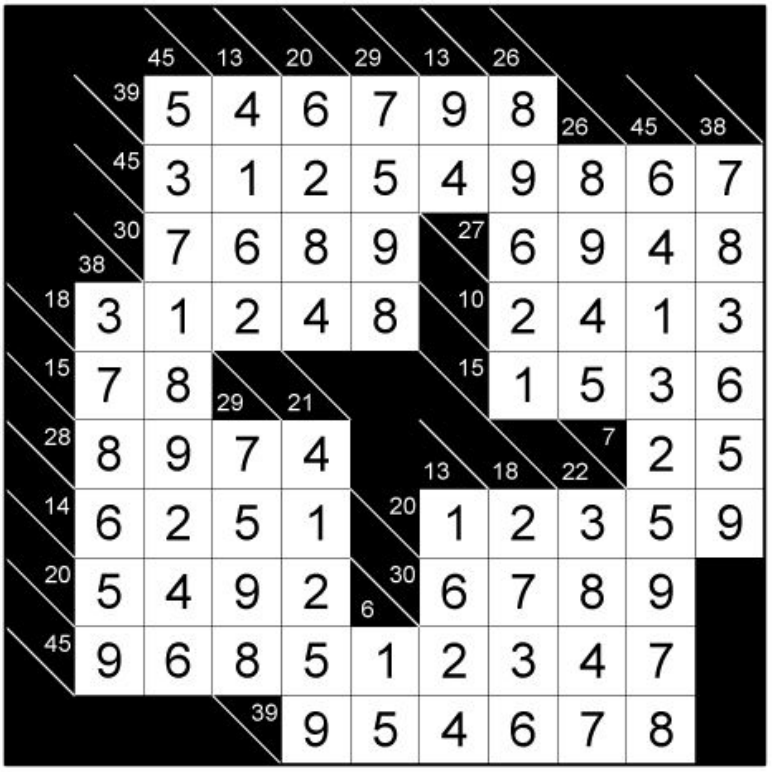


Se continúa con este proceso. Si en algún momento se detectan posibles errores con la unicidad de la solución, se realiza backtracking sobre las decisiones tomadas y se toma una distinta.





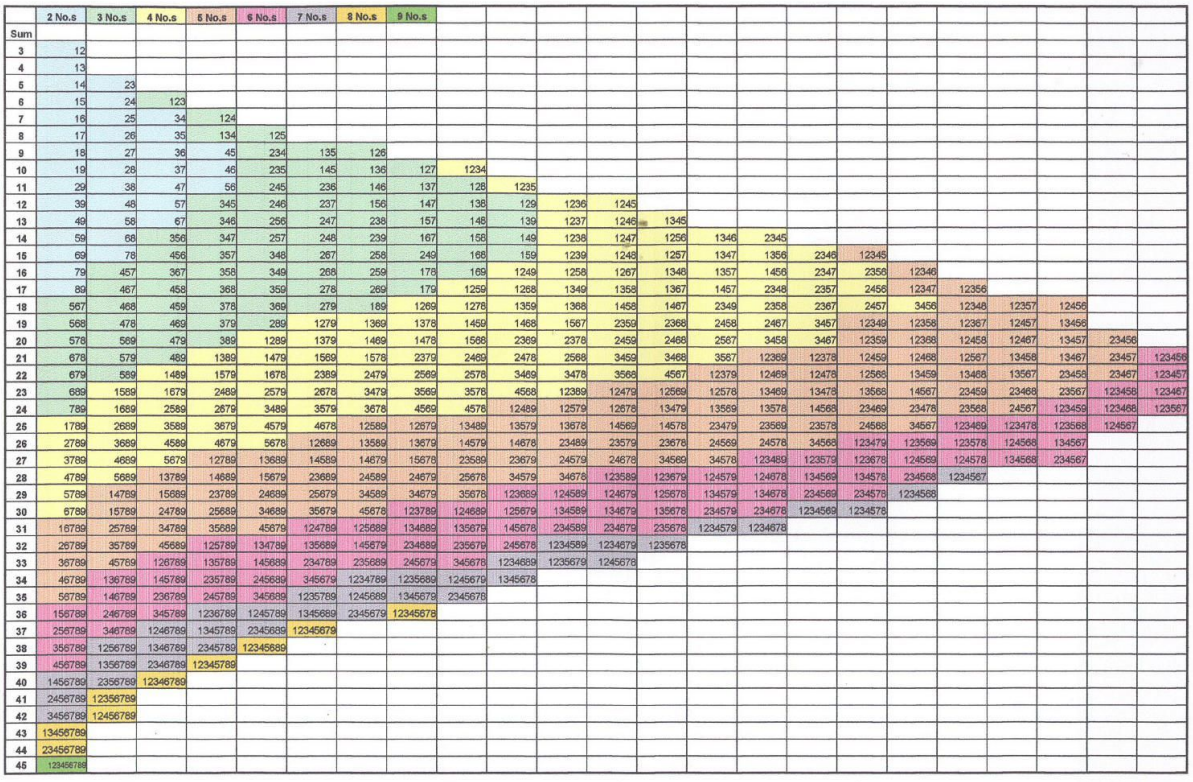
Al final de todo se dispone de un kakuro completado y se puede verificar el resultado obtenido.



Estructuras de datos y algoritmos implementados

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):



Sum	2 No.s	3 No.s	4 No.s	5 No.s	6 No.s	7 No.s	8 No.s	9 No.s															
3	12																						
4	13																						
5	14	23																					
6	15	24	123																				
7	16	25	34	124																			
8	17	26	35	134	125																		
9	18	27	36	45	234	135	126																
10	19	28	37	46	235	145	136	127	1234														
11	29	38	47	56	245	236	146	137	128	1235													
12	39	48	57	345	246	237	156	147	138	129	1236	1245											
13	49	58	67	346	256	247	238	157	148	139	1237	1246	1345										
14	59	68	356	347	257	248	239	167	158	149	1238	1247	1256	1346	2345								
15	69	78	456	357	348	267	258	249	168	159	1239	1248	1257	1347	1356	2346	12345						
16	79	457	367	358	349	268	259	178	169	1249	1258	1267	1348	1357	1456	2347	2356	12346					
17	89	467	458	368	359	278	269	179	1259	1268	1349	1358	1367	1457	2348	2357	2456	12347	12356				
18	567	468	459	378	369	279	189	1269	1278	1359	1368	1458	1467	2349	2358	2367	2457	3456	12348	12357	12456		
19	568	478	469	379	289	1279	1369	1378	1459	1468	1567	2359	2368	2458	2467	3457	12349	12358	12367	12457	13456		
20	578	569	479	389	1289	1379	1469	1478	1568	2369	2378	2459	2468	2567	3458	3467	12359	12368	12458	12467	13457	23456	
21	678	579	489	1389	1479	1569	1578	2379	2469	2478	2568	3459	3468	3567	12369	12378	12459	12468	12567	13458	13467	23457	123456
22	679	589	1489	1579	1678	2389	2479	2569	2578	3469	3478	3568	4567	12379	12469	12478	12568	13459	13468	13567	23458	23467	123457
23	689	1589	1679	2489	2579	2678	3479	3569	3578	4568	12389	12479	12569	12578	13469	13478	13568	14567	23459	23468	23567	123458	123467
24	789	1689	2589	2679	3489	3579	3678	4569	4578	12489	12579	12678	13479	13569	13578	14568	23469	23478	23568	24567	123459	123468	123567
25	1789	2689	3589	3679	4579	4678	12589	12679	13489	13579	13678	14569	14578	23479	23569	23578	24568	34567	123469	123478	123568	124567	
26	2789	3689	4589	4679	5678	12689	13589	13679	14579	14678	23489	23579	23678	24569	24578	34568	123479	123569	123578	124568	134567		
27	3789	4689	5679	12789	13689	14589	14679	15678	23589	23679	24579	24678	34569	34578	123489	123579	123678	124569	124578	134568	234567		
28	4789	5689	13789	14689	15679	23689	24589	24679	25678	34579	34678	123589	123679	124579	124678	134569	134578	234568	1234567				
29	5789	14789	15689	23789	24689	25679	34589	34679	35678	123689	124589	124679	125678	134579	134678	234569	234578	1234568					
30	6789	15789	24789	25689	34689	35679	45678	123789	124689	125679	134589	134679	135678	234579	234678	1234569	1234578						
31	16789	25789	34789	35689	45679	124789	125689	134689	135679	145678	234589	234679	235678	1234579	1234678								
32	26789	35789	45689	125789	134789	135689	145679	234689	235679	245678	1234589	1234679	1235678										
33	36789	45789	126789	135789	145689	234789	235689	245679	345678	1234689	1235679	1245678											
34	46789	136789	145789	235789	245689	345679	1234789	1235689	1245679	1345678													
35	56789	146789	236789	245789	345689	1235789	1245689	1345679	2345678														
36	156789	246789	345789	1236789	1245789	1345689	2345679	12345678															
37	256789	346789	1246789	1345789	2345689	12345679																	
38	356789	1256789	1346789	2345789	12345689																		
39	456789	1356789	2346789	12345789																			
40	1456789	2356789	12346789																				
41	2456789	12356789																					
42	3456789	12456789																					
43	13456789																						
44	23456789																						
45	123456789																						

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.

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.

Desglose de estructuras de datos por clase

Entonces teníamos un primer marco teórico, en el que la primera pregunta fundamental era encontrar la información que define ese “estado de solución”. Esta fase del diseño del algoritmo fue muy importante porque no teníamos claro la implementación del algoritmo, pero elaboramos un pseudocódigo inicial del cual pudimos extraer que tipo de estructuras de datos usar. Este pseudocódigo definió que dividiríamos la información entre distintas clases.

Estructuras de datos del algoritmo de resolución

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 tramo 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.

Definición de las estructuras

Casilla Blanca

Las casillas blancas (a pesar de que no lo sabíamos entonces) terminarían por capturar la idea de “estado de resolución”, ya que tendrían el atributo “posibles”, que representa “en este momento, el algoritmo que candidatos cree que son válidos/posibles para esta casilla blanca”.

Casilla Negra

Las casillas negras son el reverso de las casillas blancas, ya que las casillas negras son las que decidimos que contendrían las secuencias válidas para la “tira” de casillas blancas que cuelgan de la casilla negra en cuestión. Esto también puede interpretarse como el estado de la solución pero es importante ver que se puede separar de las casillas blancas, es decir, que podemos ir haciendo deducciones primero en las casillas negras, luego en las blancas, luego otra vez en las negras etc…

Otra cosa importante de las casillas negras es que tienen también un atributo candidatos, este no es una copia de candidatos de casilla negra. Candidatos en casilla negra nace de una necesidad que identificamos rápidamente y se explicará más adelante. En resumen, candidatos en la casilla negra representa, dada una casilla cualquiera que sea “hija” de esta casilla negra, que candidatos podrías deducir que tiene esa casilla blanca a sabiendas de que el padre tiene estas secuencias válidas. Como es aparente dada esta explicación, podríamos hacer el cálculo de estos candidatos a partir de el atributo secuencias candidatas de casilla negra, que contiene las secuencias que el algoritmo cree que pueden ir en esa tira.

Pero debido a que ya veíamos venir (en el pseudocódigo) que íbamos a consultar mucho este cálculo (una vez por cada casilla blanca de la tira que se estuviera analizando) decidimos tenerlo como un atributo “precalculado” que iríamos actualizando si modificamos el atributo secuencias candidatas.

Secuencia

La clase Secuencia, que engloba todas las posibilidades de secuencias según la suma total de un tramo y su número de casillas, tal y como se puede ver en la imagen anterior. Usa las mismas estructuras de datos que Casilla Negra para guardar estas cadenas, por motivos de compatibilidad/simplicidad.

Estructuras de datos usadas

Una vez “implementamos” una versión de pruebas del algoritmo, formalizando poco a poco más y más, empezamos a ver qué estructuras de datos eran las más adecuadas para los atributos anteriormente explicados, lo cual nos ahorró bastante tiempo, pues habíamos dedicado mucho tiempo a discutir qué usar.

Decidimos que por motivos de lógica e interacción con la estructura de datos, usar alguna de las implementaciones de Set, ya que en el algoritmo abundaban operaciones de “intersección de candidatos” o “unión de elementos” .

Y para las secuencias decidimos usar ArrayList por simplicidad y por las operaciones de push y pop del final de la lista, que nos permiten la construcción por algoritmos recursivos de cadenas, que sabíamos que era una necesidad.

Pero tras mucho discutir e investigar, decidimos usar una implementación especifica de Set llamada EnumSet. Esto vino a raíz de la promesa de eficiencia que tiene esta implementación vs un HashSet. También valoramos que encajaba la idea de Enum con la noción de que los números posibles en muchas de las estructuras de datos eran el set {1,2,3,4,5,6,7,8,9}. Y si esto acarreó mucha documentación sobre cómo funcionaban los Enum, incluso en la versión entregada seguimos valorando positivamente el uso de Enum que tiene de nombre Candidatos.

Como nota final sobre estas estructuras de datos, también decidimos investigar el uso de JumboEnumSet para implementar algún atributo relacionado con secuencias, pero no le encontramos utilidad para nuestro código y simplemente fue investigacion que no tuvo impacto final en el algoritmo.

Sobre el resto de estructuras de datos

Muchas variables resultaron ser combinaciones o anidaciones de las estructuras de datos anteriores, que por uniformidad decidimos usar en (casi) todas partes (como en la clase Secuencia). Una notable excepción en en el algoritmo recursivo *backtrackingFiltroImmersion* en que uno de los parámetros decidimos que fuera un int[] por simplicidad.

Pasando ahora a hablar sobre “los elefantes en la habitación”, el HashSet *pendientes*, y el Stack *prioridad*. Indagaré más en cómo funciona este “scheduler” que guía las deducciones que ejecuta el algoritmo.

Scheduler de las iteraciones

El algoritmo, como he comentado con anterioridad, hace deducciones hasta llegar a la solución. Estas deducciones, de modo simplificado, corresponder a mirar una tira, actualizar las casillas blancas con los candidatos de sus casillas negras padres, y finalmente, se actualiza el padre negro de la tira, revisando qué secuencias siguen siendo posibles, y como consequencia, qué candidatos siguen siendo validos para la tira.

Este proceso se retroalimenta, cualquier cambio de candidatos en una casilla blanca corresponde a tener que revisar las secuencias, y viceversa. Para ordenar estas “deducciones”, cada vez que el algoritmo “gana” informacion (descarta una cadena o reduce el numero de candidatos de una casilla blanca), empila en el stack los padres negros que quedan afectados por este descubrimiento. En el caso de la casilla blanca descartando candidatos, se empila los dos padres. En el caso de la casilla negra descartando secuencias, se empila a sí misma, ya que tiene que volver a comprobar sus casillas blancas.

El algoritmo de la primera entrega usa el HashSet pendientes para saber si un elemento de la pila prioridad ya ha sido revisado o no. En realidad podríamos haber usado un array de booleanos, ya que simplemente estamos mirando si el elemento del stack está o no en el set, pero como es una parte susceptible a cambios hemos decidido no invertir tiempo en mejoras marginales de eficiencia hasta tener claro que este scheduler será el definitivo.

Deduccion 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 añadiéndote. 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.

Errores teóricos, y sus soluciones, en el filtrado de secuencia

A nivel meramente teórico esta implementación inicial del “filtrado de secuencias” es errónea. Pero creemos que era instructivo explicar este proceso porque era bastante costoso encontrar porque era incorrecto, y revela la esencia del algoritmo entregado, y la corrección de este último.

Errores

En el proceso de filtrado miramos las distintas permutaciones intentando encontrar si la cadena es o no posible, pero como en todo en este algoritmo, está conectando los candidatos de casilla blanca con los de casilla negra. El error es “perder información” al hacer la unión de los elementos de la cadena. Al hacer la unión y tratar todas las casilla hijas por igual, suponemos que existe una permutación con ese elemento para cada casilla, dando a cada casilla los mismos candidatos por parte de ese padre, y eso es simplemente falso. Quizás no hay permutación que tenga un elemento concreto de la secuencia en una de las casillas de la tira, pero la unión nos diría que sí que es candidato (no se reducen suficientemente los candidatos).

Este error no era “demasiado grave”, y con esta implementación éramos ya capaces de resolver kakuros fáciles que solo con esta deducción “simple” se podían resolver. Por suerte este primer paso fue de gran ayuda y pudimos utilizar mucha parte del código para la versión definitiva del algoritmo, sobretodo la recursividad para explorar las permutaciones, que había sido bastante costosa de programar.
En resumen, como el algoritmo trabaja con información perfecta (no toma decisiones), no podemos permitirnos el lujo de destruir o simplificar la información que extraemos de ninguno de los procesos, incluyendo el análisis de las permutaciones de cada secuencia.

Soluciones

Una vez solucionamos esto, implementando un ArrayList de sets de candidatos, con un set por cada posición en la tira, el algoritmo es correcto. Entonces el algoritmo definitivo busca todas las permutaciones que son posibles dadas las restricciones de candidatos de la tira. Luego hace la unión del primer elemento de todas las permutaciones, y esos serán los candidatos de la primera casilla. I hacemos lo mismo con cada posición de la tira. De esta manera preservamos información indirecta sobre las posiciones y las permutaciones, tratando cada casilla como “única y con derecho a sus propios candidatos”. Esto es más costoso en promedio que simplemente buscar si existe una o ninguna permutación, pero el coste peor es muy similar o igual. Nuestro coste no es mucho peor, pero ahora finalmente, el algoritmo es correcto.

Historial de debugging

En cuanto a la implementación del algoritmo, nos encontramos con algunas de las dificultades y errores. Algunos de ellos errores menores propios del algoritmo en cuanto a la consideración de tramos horizontales y verticales, o problemas a la hora de determinar si se resolvía o no un kakuro dado (ahora resueltos). Como siempre, este tipo de error y otros de código se resuelven mediante debugging y análisis de código.

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 convergerán 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.

Además, aunque somos conscientes de las bondades del backtraking, hemos querido explorar otros algoritmos mucho más eficientes y rápidos.

Construcción de tablero

Esta parte del algoritmo consiste en definir donde se van a localizar las casillas blancas y negras en el tablero. En este apartado actualmente tenemos una creación simple, ya que siempre marcamos como casilla negras toda la primera primera fila y columna del tablero.
Más adelante, cuando tengamos un algoritmo más sofisticado, expandiremos en las posibilidades de creación de tableros.

Rellenar tablero

Una vez dispuestas las casillas en su sitio, debemos poner suma total vertical y/o horizontal a todas las casillas que deban tenerlo.

Se proporcionará una suma total horizontal a todas las casillas negras que tengan una casilla blanca adyacente a su derecha y se proporcionará una suma total vertical a todas las casillas negras que tengan una casilla blanca adyacente debajo suya.

En un primer enfoque, para que el algoritmo elija qué suma total tendrá una casilla negra se usa el singleton combinación, que dado un número y un número de casillas, obtiene una lista de sumas totales compatibles con ese número. Con este objeto hacemos coincidir las secuencias de las sumas totales de las casillas negras y de este modo, podemos seguir obteniendo sumas totales que hagan que las secuencias de las casillas tengan algún número en común.

Debido a la laxitud del algoritmo, con este método no conseguimos que cada casilla blanca solamente tenga un candidato posible, sino que nos aseguramos que cada casilla blanca al menos tenga una posible. Por eso, una vez indicadas todas las sumas totales, para asegurarnos de que las casillas blancas solamente tienen un candidato posible, enviamos el kakuro al algoritmo de resolución. Este, indicará si el kakuro generado tiene solución única. En el caso de no tener solución única, indicará una de las casillas en que no encajan los posibles candidatos y para cada una de estas, sus posibles candidatos.

Con esta información, los padres de casilla blanca deberán “conversar” y elegir otras sumas totales que tengan sentido con las secuencias posibles. Repetimos la comprobación del kakuro y hacemos los arreglos pertinentes hasta obtener un kakuro con solución única.

Actualmente, este enfoque no ha sido fructífero. Tras debugar el código y estudiar las posibles fuentes de error (conceptuales y de código), hemos optado por ofrecer para esta entrega un algoritmo de creación más sencillo a fin de poder realizar una primera aproximación al algoritmo de generación. Esto será explicado más adelante.

Definición de las estructuras

Además de las estructuras mencionadas anteriormente, para el algoritmo de creación se ha ideado una nueva clase auxiliar para facilitar el proceso de rellenar el tablero.

Combinacion

La clase Combinacion, tal como se ha mencionado anteriormente, contiene las sumas totales que se pueden obtener dado un número de casillas y el valor de una casilla entre el 1 y el 9. Esta clase nos permite reducir rápidamente las secuencias que son compatibles con los números ya fijados en el tablero y asegurarnos que, para una casilla, su tramo horizontal es compatible con su tramo vertical.

Errores con el algoritmo previo

En la parte de recalcular las sumas totales, si cambias sus valores por otros solamente dependiendo de las secuencias, puede hacer que sigas teniendo más de una solución, este proceso puede pasar siempre, haciendo que nunca se termine.

Posiblemente, si en lugar de almacenar las sumas totales, nos quedásemos con las propias secuencias posibles, podríamos llegar a deducir correctamente los valores de las casillas blancas considerando tanto el tramo vertical como el horizontal, y considerar también solventar las sumas incompatibles antes. Actualmente no está implementada la recogida de esos datos y el correcto tratado. Quedará para futuro trabajo.

Algoritmo provisional

Debido a las limitaciones de tiempo, hemos programado un algoritmo provisional que se basa en un backtracking para probar distintas configuraciones dado un kakuro, buscando una solución única.

Tiene una serie de pasos que segmentan la creación en:

-El “dibujado” del kakuro, en el que hay la importante restricción de que no puede haber ninguna casilla negra con una tira de tamaño uno. Decimos al principio del proyecto que nuestros kakuros no tratan con casillas negras unitarias. Hay dos dibujos, el cubo (que no puede ser más grande que 10 por razones evidentes), y la esclera diagonal, que pone las casillas de la diagonal principal y sus vecinas como blancas, el resto negras (este dibujo no tiene límite de tamaño). Por motivos de tiempo no hemos hecho más dibujos, además de que este es un algoritmo provisional.

-La creación de una “solución” a ese kakuro, en que se asignan números a las casillas blancas, y luego, si no hay repetidos en las tiras, se calculan las sumas que corresponden a esa solución

-Finalmente enviamos esa posible solución del kakuro, borrando los números de las casillas blancas, para verificar si tiene realmente una sola solución. Eventualmente encontraremos por brute force una solución. No hemos estudiado si ciertos dibujos (que cumplen con nuestra restricción) no tienen ninguna solución unitaria, pero tenemos fuertes indicios (no solo empíricos) de que es así. No adjuntamos la demostración.

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. 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.

Estructuras de datos

Para implementar las distintas funcionalidades requeridas para realizar este proyecto, se ha creado el siguiente conjunto de clases con sus drivers y stubs correspondientes.

Clases y drivers:

Tal y como muestra el diagrama de clases de diseño, nuestro proyecto de Java dispone de un conjunto de clases para implementar las funcionalidades requeridas por la práctica, además de algunas posibles funcionalidades adicionales propuestas por nuestro grupo (a implementar más adelante).

Clases principales

- Kakuro.
- Casilla, casilla negra y casilla blanca.
- Algoritmo.
- Partida.
- Repositorio de kakuros, Repositorio de partidas.

Clases adicionales

- Record.
- Ranking.
- Modo, nivel ayuda.

Algunas clases auxiliares

- Pair.
- Secuencia y combinación.
- Read

Implementación de las clases

En esta sección se ofrece una breve descripción de cada una de las clases, resumiendo alguna de la información proporcionada previamente y entrando más en detalle en cómo están implementadas.

Kakuro

Una de las clases principales de este proyecto, la clase kakuro consiste de un tablero (matriz bidimensional) de las casillas blancas, negras y vacías que lo conforman. También dispone de un ArrayList de sus casillas negras para poder acceder a ellas con facilidad.

Casilla

Casilla es una clase abstracta. De ella heredan los dos tipos de casilla, las blancas y las negras.

Casilla negra

Las casillas negras del kakuro representan aquellas casillas que no rellena el usuario, ya sean casillas vacías, o casillas que dan información sobre la suma de un tramo.

Si son casillas que encabezan un tramo, disponen del número de casillas que forman parte del tramo, además de la suma de sus valores. Disponen también de ArrayLists con los posibles candidatos del tramo en cada momento, además de la unión de todos estos en un ArrayList de EnumSet ya que, tal como se ha comentado previamente, presentan mejoras en eficiencia frente al HashSet. Por último, también disponen de un EnumSet con los valores de las casillas blancas que ya han convergido.

Por otro lado, si se trata de casillas negras vacías, se inicializan sus sumas en -1 para identificarlas, pero más allá sus atributos no son usados.

Casilla blanca

Casilla blanca contiene los valores a rellenar para resolver el kakuro. Esta clase contiene un booleano que indica si su valor es modificable o no, ya que, según el nivel de ayuda, es posible que se muestre el valor de la solución de esa casilla y por lo tanto no es de interés que el usuario lo pueda modificar accidentalmente. Además de esto, contiene dos atributos de tipo int que indican el valor elegido por el usuario en cualquier momento, y el valor de la solución obtenida durante la creación y validación de ese kakuro. Por último, casilla blanca dispone de dos pairs indicando las coordenadas de sus casillas negras “padre” en el tablero para facilitar su acceso a ellas.

Partida

La clase partida representa una instancia en la que un usuario inicia la resolución de un nuevo kakuro. Dispone del atributo id, que la identifica, de una variable con la hora en la que se creó, y otra variable que almacena el tiempo que transcurre mientras el usuario resuelve el kakuro, teniendo en cuenta las posibles pausas y reanudaciones que el usuario pueda llevar a cabo con las llamadas a *guardarPartida* y *cargarPartida*. Para esto, se guarda para cada llamada a *cargarPartida* la hora en timpoIncioSesion, y se almacena el tiempoTranscurrido al *guardarPartida* o *acabarPartida*. Se dispone además de dos booleanos que indican si la partida se ha acabado, y de ser así de si el usuario se ha rendido o no. Para una partida acabada se puede calcular el atributo puntuación.

Repositorio de partidas

El repositorio de partidas es la clase que engloba en un HashMap todas las partidas realizadas, manteniendo en todo momento el número de partidas que figuran en él.

Repositorio de kakuros

El repositorio de kakuros es la clase que engloba en un HashMap todos los kakuros generados, manteniendo en todo momento el número de kakuros que figuran en él.

Record

Record es la clase que almacena para un determinado usuario la mejor partida que ha realizado (mejor puntuación).

Ranking

Ranking es la clase que almacena para un determinado kakuro un listado de partidas ordenados según las puntuaciones obtenidas en cada una de ellas.

Modo

Modo es una clase abstracta que engloba las dos posibles modalidades de juego: normal y ranking. Cada una de estas modalidades aportan información sobre el tipo de partida y el sistema de puntuación.

Nivel de ayuda

Nivel de ayuda es también una clase abstracta que engloba el tipo de ayuda que se ofrece al usuario (a definir más adelante), además su efecto sobre los sistemas de puntuación.

Pair

Pair es la típica clase que contiene dos ints. Se usa mayoritariamente para acceder a posiciones del tablero.

Secuencia

Secuencia es una clase auxiliar especial, que almacena una vez los valores representados en la figura 1 de esta documentación: las posibles combinaciones de números del 1 al 9 que pueden dar una suma dada. Esta implementado en un ArrayList cuatridimensional. Su funcionamiento es el siguiente: dada una suma a alcanzar y un número de casillas, devuélveme el array con todas las posibles combinaciones que pueden dar ese resultado. Esta es una clase muy útil, pues nos facilita de manera directa información que empleamos para la resolución de kakuros.

Combinación

Combinación es una clase del mismo carácter que secuencia. Como se ha explicado anteriormente, devuelve las posibles sumas que se pueden alcanzar dado un número de casillas y uno de los integrantes. Está también implementada con ArrayLists de integers.

Es posible que esta clase se vea modificada en un futuro según avance el desarrollo del algoritmo de creación, e incluso puede que se elimine por completo.

Read

Read es una clase auxiliar que interactúa con los canales de entrada y salida empleando la clase Scanner y devuelve excepciones en caso que no se encuentren los ficheros deseados.

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://www.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