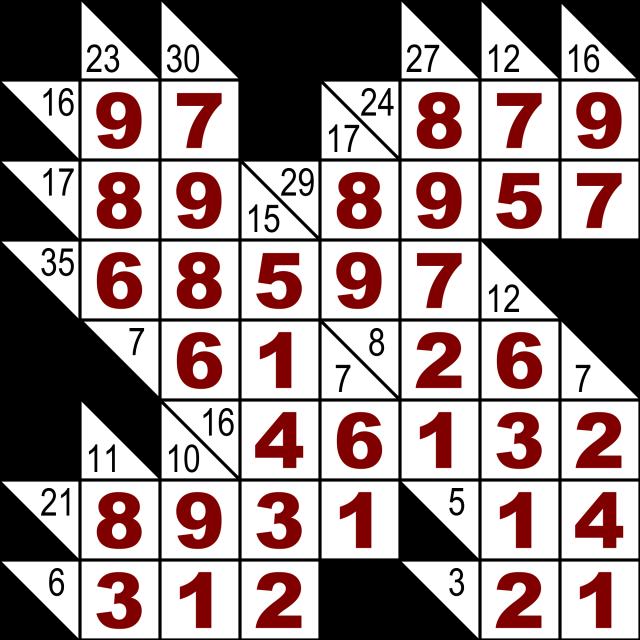
|  |
| --- |
| Representación del conocimiento |
| Sonia Díaz Santos, Jorge González Cabrera |
|  |
| El objetivo del proyecto es la realización de un sistema basado en el conocimiento usando Prolog. En nuestro caso, de Kakuro, un puzle japonés a menudo referido como una transcripción matemática del crucigrama. |
|  |

Kakuro

# Explicación del rompecabezas

El Kakuro es un rompecabezas que consiste en una cuadrícula en la que tendremos que rellenar algunos de los cuadrados con números del 1 al 9.

Aquí tenemos un ejemplo ya resuelto. En este caso los espacios en blanco corresponden a los espacios donde hay que introducir los números. Estos no pueden repetirse por secciones, considerando como tal a las celdas blancas delimitadas por espacios no válidos vertical y horizontalmente. Por ejemplo, en la primera fila nos encontramos con las secciones (9,7) y (8,7,9). Como vemos, hay dos nueve en la primera fila, pero no en la misma sección.

De igual forma, en la primera columna tenemos las secciones (9,8,6) y (8,3).

En el ejemplo, también podemos ver unos números más pequeños en unos triángulos. Esta es la verdadera dificultad del rompecabezas, ya que indican el valor que deben sumar los números que pertenecen a una determinada sección. Por ejemplo, en las secciones mencionadas anteriormente comprobamos que ‘16 = 9 + 7’ y ’24 = 8 + 7 + 9’.

# Explicación del código

kakuro



En primer lugar, cuando llamamos a nuestra función ‘kakuro’ necesitamos tres valores. El primero, ‘VALS’, corresponde a nuestra matriz de números, en la que los espacios en los que no se introducen números se simbolizan con un 0. Las otras dos variables corresponden a la suma de las secciones en horizontal, ‘SUMX’, y en vertical, ‘SUMY’. En el caso que se llame a esta función se calcularán que los valores de la matriz cumplen con las sumas horizontales y posteriormente se comprobará de igual manera las sumas verticales haciendo la traspuesta de la matriz. Con esto en mente, las sumas deberán aparecer ordenadas de izquierda a derecha y de arriba a abajo.

check\_rows



La comprobación de las filas requiere cuatro valores. En primer lugar, la matriz a comprobar seguida de los valores de las sumas (las horizontales en el primer caso, y las verticales con la traspuesta). Los dos siguientes son acumuladores, en el que el primero de ellos corresponde a la suma que se va realizando con los valores que se encuentran de una sección y, en segundo lugar, un array de los números disponibles, para evitar que se repitan en una misma fila y que sólo se puedan usar números del 1 al 9.

Para comprobar la matriz comprobaremos siempre el primer valor de la primera fila. Hay varias opciones a contemplar. En primer lugar, podría haber un número del 1 al 9, o una variable. En nuestro caso obligamos a que si hay una variable tenga que haber un número del 1 al 9, para asegurarnos de saber cuales son las casillas disponibles. Como segunda opción podría haber un 0, lo que indicaría que hay un espacio en el que no se introduce ningún número. Por último, se contempla la posibilidad de que ya esa fila esté vacía, por lo que habría que continuar con el resto.



En la primera opción, utilizamos la función ‘selecciona’ trabajada en prácticas para comprobar si el primer elemento pertenece a uno de los posibles y en tal caso obtener un nuevo array que elimine dicho elemento. Al haber un número, el acumulador debe aumentar, por lo que se le suma el valor del número asignado a dicha casilla. Por último, se vuelve a llamar a la función con el nuevo acumulador, los nuevos números disponibles y quitando el primer valor de la primera fila ya que lo acabamos de comprobar.



En el caso de que el valor a comprobar fuera un 0 habría que contemplar dos posibilidades más. Una de ellas sería que el acumulador valiera 0, lo que significaría que no ha habido números antes, por lo que simplemente quitamos el primer valor de la matriz. Primero comprobamos que el primer valor es una constante, y no una variable. Con esto lo que queremos conseguir es que, si se utiliza una variable para conocer el valor que podría ser, no contemple la posibilidad de que sea una casilla no apta.



Ahora lo que tenemos que tener en cuenta es si cuando encontramos un 0 quiere decir que ha terminado una sección. Para ello, comprobamos que el primer valor de la suma corresponde al acumulador y que este valor no sea cero. Si cumple estas condiciones se volverá a llamar a la función quitando el primer valor de la matriz, el primer valor de la suma porque ya lo hemos comprobado, y el acumulador y los valores disponibles se reinician porque se ha acabado una sección.



De forma similar, cuando lleguemos al final de una fila podrá significar que habrá acabado una sección o no. Por lo tanto, estos casos se comportarán igual que los dos casos anteriores con la salvedad de que ahora en vez de quitar el primer elemento de la primera fila, se quitará la fila completa.



Finalmente, una comprobación acaba correctamente si la matriz y el array de los valores de las sumas se vacían al mismo tiempo.

imprimir\_kakuro



Para el método de impresión tenemos varias etapas. La función ‘imprimir\_kakuro’ es llamada desde la función principal ‘kakuro’ para imprimir tanto la matriz como el resultado de las sumas horizontales y verticales.

Para realizar la impresión de la matriz llamamos a la función ‘imprimir\_matrix’. En esta función, se encuentra la condición de parada cuando la lista esté vacía. Si la lista no está vacía coge la primera fila de la matriz y llama al método ‘imprimir’.

El método ‘imprimir’ se encarga de mostrar los valores de un array. Al imprimir la matriz tenemos que poner la condición ‘[A|VALS] \== []’ para comprobar si la lista se encuentra vacía. En caso contrario, se ejecutarían ambas funciones.

Una vez se ha impreso la primera fila, se vuelve a llamar al método ‘imprimir\_matrix’ para que se impriman las demás filas que componen la matriz.

Cuando se ha impreso toda la matriz, la función ‘imprimir\_kakuro’ imprime los valores de las sumas horizontales y verticales. Para llevar a cabo este cometido, llamamos a la función ‘imprimir’ para mostrar los valores de las sumas.

La función ‘imprimir’ muestra los ceros con guiones, este es el motivo por el cual no imprimimos la matriz directamente y hacemos las funciones descritas anteriormente. Así podemos mostrar un formato separado por espacios y bien estructurado para una mejor visualización.

Para imprimir hemos utilizado otras dos funciones, ‘nl’ que imprime un salto de línea y ‘tab(X)’ que sirve para imprimir espacios, siendo ‘X’ el número de espacios que queremos asignar.

traspuesta



Para hacer la traspuesta vamos a pasarle dos parámetros. En primer lugar, la matriz que queremos trasponer y, en segundo lugar, aquella que dará como resultado. Para cumplir nuestro objetivo se hará la traspuesta columna a columna con la función ‘traspuesta\_columna’. La primera vez que llamemos a dicha función le daremos la matriz original, la primera columna del resultado y un tercer parámetro que devolverá todos los elementos de la matriz que aún no han sido traspuestos. Como comprobación, dejamos constancia de que la primera fila de este tercer parámetro será igual a la primera fila de la matriz original sin el primer elemento.

La función ‘traspuesta\_columna’ sólo comprobará en la primera llamada que el primer valor de cada fila de la matriz original coincidirá con los valores de la primera fila del resultado. Para ello, en cada nivel de recursividad quitará la primera fila de la matriz original y el primer elemento de la primera fila del resultado. Por otro lado, para el correcto funcionamiento del algoritmo necesitamos crear una matriz auxiliar que quite la primera columna de la matriz original. Por ello, comprobaremos en todo momento que la primera fila de este tercer parámetro será igual a la primera fila de la matriz original sin el primer elemento.

Este método acabará correctamente cuando los tres elementos estén vacíos simultáneamente.

Cuando salgamos de este método, dentro de la función ‘traspuesta’ se volverá a llamar a sí misma, pero sin la primera fila de la matriz original y sin la primera fila del resultado. Este método seguirá disminuyendo la matriz hasta que ambos acaben a la vez. Sin embargo, tenemos en cuenta que la matriz tiene el mismo número de columnas en todas las filas (una condición básica de las matrices), pero si se hiciera mal uso de la función podría dar resultados incoherentes.

# Proceso de realización

En nuestro primer planteamiento, queríamos comprobar simultáneamente tanto las filas como las columnas. Esto quizás generaría un árbol más pequeño, pero la verdad es que requeriría tener muchos más datos en cuenta en cada nivel del árbol y el código hubiera sido mucho más complejo y difícil de leer.

Antes de tomar esta decisión el planteamiento era realizar diferentes funciones para las comprobaciones en horizontal y en vertical. Sin embargo, hacer la traspuesta de los valores nos permitió reutilizar el mismo método y mejorar la visibilidad del código.

De igual modo, también nos dimos cuenta de que la separación por secciones y la suma de dichos elementos se podían hacer con la misma función si usáramos acumuladores. Además, necesitábamos evitar repetir los números, que también se solucionó con un acumulador, ya que los posibles casos eran demasiados como para hacer funciones independientes para cada número.

Entre las dificultades, el mayor contratiempo fue darnos cuentas de que utilizar variables podría dificultar orientar el árbol por una rama en particular. En concreto, para simular los espacios en los que no se introducen números utilizamos el valor 0. Cuando queremos detectar cual es el primer valor de la primera fila hace falta diferenciar si es un 0 o si se encuentra entre el 1 y el 9. En el caso de usar la función con números no habría ningún problema. Sin embargo, si le pasáramos una variable a la función podría crear una rama con valor 0. Para solucionarlo recurrimos a la función ‘nonvar’, que nos permite saber si el valor que se le pasa tiene un valor asignado o no, y así permitir el 0 sólo si el usuario lo ha escrito.

En menor medida, hubo un problema cuando pasábamos una variable para recibir el array con las sumas tanto en vertical como en horizontal. Como en el caso anterior, nos dimos cuenta de que le estábamos dando permiso para que fuera cualquier valor. Por lo tanto, cuando le especificábamos que uno de los valores de la suma tenía que ser igual al acumulador, en ocasiones este podía ser cero y cumplía el resto de las normas del rompecabezas. Esto se solucionó simplemente condicionando para que este valor no pudiera ser cero.

Hemos tenido dificultades a la hora de mostrar el resultado del Kakuro por pantalla debido a que no queríamos hacerlo en una sola función. Esto se debe a que queríamos que en el caso en el que un elemento fuera un 0 se imprimiera un ‘-’ para que el formato de salida fuera más limpio y legible. El problema en este caso es que tuvimos que añadir dos condiciones.

La primera es ‘V1 \== 0’ ya que si no se añade esta condición el programa ejecutaría la función ‘imprimir’ dos veces. En primer lugar, ejecutaría la función ‘imprimir([0|A])’ y después ejecutaría la función ‘imprimir([A|VALS])’.

La segunda condición que añadimos es ‘[A|VALS] \== []’. Esto se debe al mismo motivo que la primera condición. Si no la pusiéramos cuando la matriz estuviera vacía se llamaría primero a la función ‘imprimir\_matrix([])’ y después se llamaría a ‘imprimir\_matrix([A|VALS])’ .

# División de roles

En cuanto al trabajo que realizamos cada uno cabe destacar que ambos participaron en la realización de este informe y en el planteamiento inicial del rompecabezas, que llevó a tener la estructura que posee actualmente el código. Es decir, establecer el mismo método para sumas verticales y horizontales e ir comprobando los elementos de uno en uno, además de las estructuras de los atributos.

En el caso de Jorge González Cabrera, continuó con el resto de la realización del rompecabezas, mientras que Sonia Díaz Santos se encargó primordialmente de mostrar el resultado formateado por pantalla y de hacer la traspuesta de una matriz sin utilizar librerías propias de SWI-prolog.

Por supuesto, ambos trabajaron conjuntamente para llegar a las soluciones de cada una de sus tareas.

# Tablas de rendimiento

Aquí mostramos algunos datos sobre las capacidades del código. Como se puede apreciar, en muchos de los casos el resultado es inmediato, pero a partir de un cierto número aumenta considerablemente el tiempo. En los casos que hemos puesto ‘Más de diez minutos’ no ha llegado a conseguir el resultado tras diez minutos, pero sigue trabajando. En otros, el primer resultado es inmediato, pero a la hora de crear el resto del árbol tarda considerablemente debido al gran número de posibilidades que puede generar.

|  |  |  |  |
| --- | --- | --- | --- |
| Tamaño de matriz | Nº de variables | Ubicación de variables | Tiempo aprox. |
| 6x6 | 0 | - | Inmediato |
| 6x6 | 6 | Posiciones concretas de matriz | Inmediato |
| 6x6 | 12 | Posiciones concretas de matriz | Inmediato |
| 6x6 | 18 | Posiciones concretas de matriz | 2 segundos |
| 6x6 | 24 | Posiciones concretas de matriz | Más de diez minutos |
| 6x6 | 4 | Ambos arrays de las sumas | Inmediato |
| 6x6 | 1 | Un array completo de las sumas | Inmediato |
| 6x6 | 2 | Ambos arrays de las sumas | Inmediato |
| 13x13 | 2 | Ambos arrays de las sumas | Primer resultado inmediato |
| 13x13 | 1 | Un array completo de las sumas | Primer resultado inmediato |
| 13x13 | 0 | - | Inmediato |
| 13x13 | 13 | Posiciones concretas de matriz | Inmediato |
| 13x13 | 26 | Posiciones concretas de matriz | Inmediato |
| 13x13 | 52 | Posiciones concretas de matriz | Más de diez minutos |

En esta práctica hemos intentado conseguir el mayor numero de posibilidades en cuanto a que datos queremos averiguar, algunos de los cuales se encuentran en estas pruebas. Hay concretamente dos que no hemos planteado porque dan información escasa para generar una solución en un tiempo razonable.

En primer lugar, en el caso de requerir una matriz completa (no especificando un número de filas ni de columnas), el número de soluciones se hace extremadamente grande por lo que no hemos sido capaces de ver si se podría completar con éxito.

En el caso de darle todas las casillas como variables (excepto las celdas no válidas), se podría deducir el tamaño del array. No hemos comprobado todas las capacidades de estos casos debido a que genera muchas soluciones incluso en tamaño 2x2.

En otro caso, si queremos averiguar una fila concreta de la matriz dándole sólo una variable (en vez de una variable por posición) sólo podrá ejecutarse si ninguno de esos valores puede ser cero.

# Ejemplo

Este ejemplo es uno de los realizados para completar la tabla anterior. En concreto, corresponde a la penúltima fila. En primer lugar, llamamos de esta forma a la función:



Realmente así no fue como la escribimos, ya que fue todo en una línea, pero así es más fácil de visualizar al separar cada fila de la matriz en una línea distinta.

Al ejecutarlo, el resultado será este:



Lo primero que se verá es el resultado de nuestra función ‘imprimir\_kakuro’ con los valores correctos en el caso de que se haya encontrado alguna solución. Posteriormente se verán los valores de las sumas (tanto horizontales como verticales). Dichos valores estarán ordenados por su aparición de arriba abajo y de izquierda a derecha. Por último, como cualquier otra función de prolog, veremos los valores que se han asignado a las variables.

# Enlaces de interés

Repositorio Github con nuestro código e informe:

<https://github.com/alu0101061672/Kakuro>

Repositorio Github que nos dio la idea y que sirvió como primer planteamiento:

<https://github.com/cbarrick/puzzles.pl/tree/master/kakuro>

Página con información sobre el funcionamiento del rompecabezas:

<https://en.wikipedia.org/wiki/Kakuro>

Página que nos proporcionó una gran cantidad de ejemplos para comprobar, además de ayudarnos a comprender su funcionamiento:

<https://www.kakuroconquest.com/>