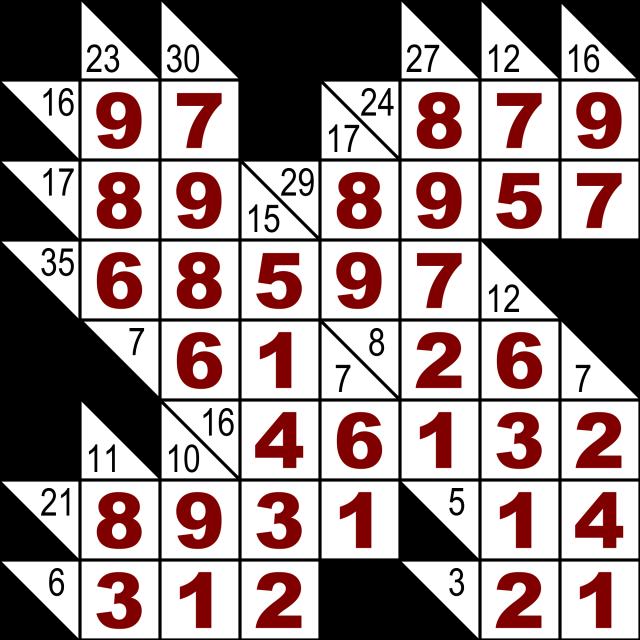
|  |
| --- |
| Representación del conocimiento |
| Sonia Díaz Santos, Jorge González Cabrera |
|  |
| El objetivo del proyecto es la realización de un sistema basado en el conocimiento usando Prolog. En nuestro caso de Kakuro, un puzle japonés a menudo referido como una transcripción matemática del crucigrama.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

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. 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.

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

En primer lugar, cuando llamamos a nuestra función ‘kakuro’ necesitamos tres valores. El primero, ‘VALS’, corresponde a nuestra matriz de números, en los 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.

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.

Para el método de impresión tenemos varias etapas. La función write\_sum es llamada desde la función principal kakuro para imprimir tanto la matriz como el resultado de las sumas horizontales y verticales. Aquí se produce una subdivisión entre la función imprimir que sirve para imprimir todos los elementos de una determinada fila llamada tanto para imprimir el vector de sumas horizontales y verticales como para imprimir fila por fila la matriz, y la función imprimir\_matrix que sirve para imprimir la matriz que es en sí el juego kakuro. Esta función imprime fila por fila llamando a la función imprimir para así imprimir la matriz entera fila por fila.

# 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 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 concreta. 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 que 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.

# 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 para encontrar más soluciones tarda considerablemente debido al gran número de posibilidades que puede generar.

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 escaza para generar una solución en un tiempo razonable.

En primer lugar, 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. Por otro lado, en el caso de requerir una matriz completa no especificando un número de filas ni de columnas por lo que el número de soluciones se hace extremadamente grande, pero en casos pequeños se puede observar la posibilidad de hacerlo.

En otro caso, que sería darle todas las casillas con variables, pero diciéndole el tamaño del array y las casillas que no corresponden a valores, propondría muchas soluciones incluso en tamaño 2x2. Aún así no hemos comprobado todas las capacidades en este sentido.

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |





# 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/>