

Generador de Kakuros

Eduard Torres Chaves, y Juan José Solano Quesada,

Abstract—This document discusses the study of the order of functions of pruning, backtracking and the functions in charge of permuting a list. It begins with the study of the large backtracking O and backtracking itself, then follows the study of the or the functions in charge of pruning and then the function in charge of performing permutations.

Index Terms—Backtracking, thread, poda.

I. INTRODUCCIÓN

Un kakuro es un juego matemático que consiste en un conjunto de casillas blancas y negras, en las cuales se deben rellenar las casillas blancas en el tablero con números del 1 al 9 tal que la suma de los valores en las casilla sea igual a la clave que se encuentra a la izquierda para las filas y por encima de las columnas.

Entre sus reglas se distinguen que no se puede repetir el valor de una casilla en la misma fila y columna hasta que haya un espacio negro y, por ende, la cantidad máxima de celdas consecutivas es 9.

Los problemas matemáticos que representan estos enigmas pueden resolverse utilizando técnicas de matriz matemática y backtracking.

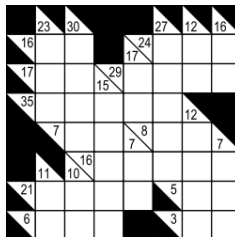


Fig. 1. Ejemplo de kakuro sin resolver

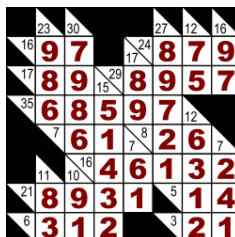


Fig. 2. Ejemplo de kakuro resuelto

II. ANÁLISIS

A. Permutaciones

En la figura 3, se muestra la función utilizada para calcular las permutaciones de una lista de tamaño n , al estudiar la función y las características del mismo, se llega a la conclusión de que posee un orden de $\mathcal{O}(n!)$, ya que en la definición fundamental de una permutación es la igual al orden del factorial de n o $n!$, este representa el número de formas distintas de ordenar n objetos de manera diferente (sin repetir ordenes). El contenido de la lista se vuelve indiferente ya que, no importa el valor de los mismo a la hora de correr el algoritmo, pero si afecta el número de los mismo.

```
def swap(a, i, j):
    a[i], a[j] = a[j], a[i]

def permute(a, i, n):
    if i == n:
        print(a)
        return a
    for j in range(i, n+1):
        swap(a, i, j)
        permute(a, i+1, n)
        swap(a, i, j) # backtrack
```

Fig. 3. Función de permutaciones

B. Poda

Una de las partes más importantes del algoritmo de backtracking es su poda, pues reduce al mínimo la búsqueda de posibles soluciones al problema, lo que permite que la resolución del problema se realiza con mayor rapidez y eficiencia. En este caso la poda consiste en una estructura que contiene las posibles soluciones para que los números de una cantidad determinada de celdas sumen el valor de la clave determinada.

```
def obtenerPosibilidades(goalValue, emptyCells):
    possibilities = {
        3: { 2: [[1,2]] },
        4: { 2: [[1,3]] },
        5: { 2: [[1,4], [2,3]] },
        6: { 2: [[1,5], [2,4]], 3: [[1,2,3]] },
        7: { 2: [[1,6], [2,5], [3,4]], 3: [[1,2,4]] },
```

Fig. 4. Parte de la función de poda

En la figura 4 se puede ver que la cantidad de combinaciones crece conforme el valor de la clave se incrementa, no obstante, cuando la clave se encuentre entre 21 y 24 alcanza el valor máximo de posibles combinaciones, con diferentes valores de casillas consecutivas, las cuales varían en 23 posibles combinaciones. El límite que llega a alcanzar el valor de la clave es de 45, el cual posee una única combinación posible, que es la sumatoria del 1 al 9.

De lo anterior se puede decir que $O(\text{obtenerPosibilidades}(\text{clave}, \text{celdas}))$ se encuentra cuando el valor de la clave es 45 ó cuando el valor de la clave es 24 y la cantidad de celdas es el máximo para generar la suma de la clave, las cuales serían 6 celdas consecutivas. Entonces:

$$O(\text{obtenerPosibilidades}(45, 9)) = 46$$

$$O(\text{obtenerPosibilidades}(24, 6)) = 47$$

Por lo tanto $O(\text{obtenerPosibilidades}) = 47$

C. Backtracking

Al momento de analizar las funciones encargadas del backtracking, se debe de tomar en cuenta que el orden puede ser afectado por otras funciones, como por ejemplo las encargadas de la poda, sin embargo, al ser estudiadas en otra sección, no se contemplará el efecto que tengan en el orden del backtracking. El kakuro tiene la particularidad de solo tener, por numero, 9 casillas vacías, por lo que el rango maximo de interacciones por numero a calcular es de 9. La función backtrack, figura 4, busca la solución con un maximo de 11 veces por el parametro "intentos" (maximo de veces que un mismo numero aparece en una lista de los posibles conjuntos) si no es que no ha encontrado una solución, al analizarla obtenemos un orden $O((n * 11 * k))$ donde k es la cantidad de veces que la recursividad necesiat evaluar por casilla, que a su vez se puede reducir a $O((n * k))$, se tiene que tener en cuenta que backtrack llama a la función meterEnMatriz, figura 5, por lo que es necesario el estudio de la función meterEnMatriz.

La función meterEnMatriz entra en un while que puede tener dos duraciones, en el mejor caso una duración de 9 (en el caso de ser una serie de 9 espacios en blanco sin ninguna intersección) o de 15 como maximo de interacciones antes de que se requiera otro grupo de solución (el cual se encarga la función backtrack), por lo que el orden de la función esta dada por $O(15)$ al ser 15 el maximo entre (9,15).

Con lo anterior, podemos deducir que la O grande del backtracking, como un todo, seria de $O(n * k) * O(C)$, con C siendo la constante 15 de la O grande de la función meterEnMatriz, a lo que termina siendo $O(n * k)$

```
def backtrack(matrizSolo, tuplaActual, listaDeTuplas, intentos):
    if (listaDeTuplas == []):
        return matrizSolo
    else:
        resultado = meterEnMatriz(matrizSolo, tuplaActual, obtenerPosibilidades(tuplaActual[0], tuplaActual[1]))
        if (resultado[0] and intentos < 11):
            backtrack(resultado[1], listaDeTuplas.pop(), listaDeTuplas, 0)
        else:
            intentos += 1
            backtrack(resultado[1], tuplaActual, listaDeTuplas, intentos)
```

Fig. 5. Función de Backtrack

```
def meterEnMatriz(matriz, tupla, posibilidad):
    if (tupla[1] == 1):
        listaDeTuplas = obtenerPosibilidades(tupla[0], tupla[1], tupla[2], matriz)
        posibilidadDeMasFactible = eliminarPresentes(posibilidadDeMasFactible, listaDeTuplas)
        contador = 1
        random.shuffle(posibilidadDeMasFactible)
        metidos = []
        while (len(metidos) < len(posibilidadDeMasFactible) and intentos < 15):
            if (matriz[tupla[0]][tupla[1] + contador] == 0):
                posibilidadDeMasFactible[tupla[1] + contador] = posibilidadDeMasFactible[0]
                if (verificar(tupla[0], tupla[1] + contador, tupla[2], matriz)):
                    metidos.append(tupla[1] + contador)
                    contador += 1
            else:
                print("No se pudo meter porque no se valio el num", posibilidadDeMasFactible[0], " en la posicion ", tupla[0], " ", tupla[1] + contador)
                random.shuffle(posibilidadDeMasFactible)
                intentos += 1
            contador += 1
        if (intentos == 15):
            matriz = limpiarQueQuedo(matriz, matriz, posibilidadDeMasFactible)
            return (False, matriz)
        else:
            return (True, matriz)
    else:
        listaDeTuplas = obtenerPosibilidades(tupla[0], tupla[1], tupla[2], matriz)
        posibilidadDeMasFactible = eliminarPresentes(posibilidadDeMasFactible, listaDeTuplas)
        contador = 1
        random.shuffle(posibilidadDeMasFactible)
        metidos = []
        while (len(metidos) < len(posibilidadDeMasFactible) and intentos < 15):
            if (matriz[tupla[0] + contador][tupla[1]] == 0):
                posibilidadDeMasFactible[tupla[0] + contador] = posibilidadDeMasFactible[0]
                if (verificar(tupla[0] + contador, tupla[1], tupla[2], matriz)):
                    metidos.append(tupla[0] + contador)
                    contador += 1
            else:
                print("No se pudo meter porque no se valio el num", posibilidadDeMasFactible[0], " en la posicion ", tupla[0], " ", tupla[1] + contador)
                random.shuffle(posibilidadDeMasFactible)
                intentos += 1
            contador += 1
        if (intentos == 15):
            matriz = limpiarQueQuedo(matriz, matriz, posibilidadDeMasFactible)
            return (False, matriz)
        else:
            return (True, matriz)
```

Fig. 6. Función de meterEnMatriz

D. Generar un tablero