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 las 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 tecnicas de matriz matemática y backtracking.

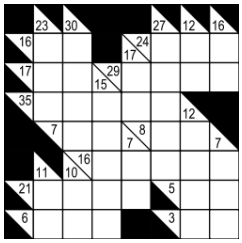

Fig. 1. Ejemplo de kakuro sin resolver

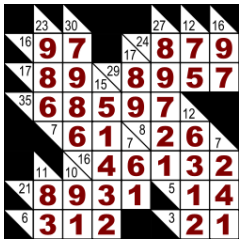

Fig. 2. Ejemplo de kakuro resuelto

E. Torres Chaves estudiante de Ingeniería en computación, Instituto Tecnológico de Costa Rica
J.J. Solano estudiante de Ingeniería en computación, Instituto Tecnológico de Costa Rica

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 caracteristicas del mismo, se llega a la comclusion 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 numero 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 numeros de una cantidad determinada de celdas sumen el valor de la clave determinada.

```
def obtenerPosibilidades(goalValue, emptyCells):
    posibilities = {
          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 enctuentre 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 $O(obtenerPosibilidades(clave, celdas))$ 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:

$$O(obtenerPosibilidades(45, 9)) = 46$$

$$O(obtenerPosibilidades(24, 6)) = 47$$

Por lo tanto $O(obtenerPosibilidades) = 47$

C. *Backtracking*

Al momento de analizar las funcioónes encargadas del backtraking, se debe de tomar en cuenta que el orden puede ser afectado por otras funciónes, como por ejemplo las encargadas de la poda, sin embargo, al ser estudiadas en otra sección, no se contemplara el efecto que tengan en el orden del backtracking. El kakuro tiene la particularidad de solo tener, por numero, 9 casillas vaciás, por lo que el rango maximo de interaciones por numero a calcular es de 9. La función backtrack, figura 4, busca la solucion 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 interaciones 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(matrizSolu,tuplaActual,listaDeTuplas,intentos):
    if (listaDeTuplas==[]):
        return matrizSolu
    else:
        resultado=meterEnMatriz(matrizSolu,tuplaActual,obtenerPosibilidades(tuplaActual[3],tuplaActual[2]))
        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(pmatrix,tupla,posibilidades):
    if(tupla[4]==1):#para las horizontales
        listaToques=sacarToquesHorizontales(tupla[0],tupla[1],tupla[2],pmatrix)#para ver si pega en algun lado con una vertical
        posibilidadMasFactible=escojerMejorPosibilidad(posibilidades,listaToques)
        posibilidadMasFactible=eliminarPresentes(posibilidadMasFactible,listaToques)
        contador=0
        random.shuffle(posibilidadMasFactible)
        metidos=0
        intentos=0
        while ((metidos!=len(posibilidadMasFactible))and intentos<15):
            if(pmatrix[tupla[0]][tupla[1]+contador]=="|"):
                pmatrix[tupla[0]][tupla[1]+contador]=posibilidadMasFactible[0]
                if (verificar(tupla[0],tupla[1]+contador,tupla[2],pmatrix)):
                    print("dio true")
                    metidos+=1
                    contador+=1
                else:
                    print("va a hacer cambio porque no es valido el num",posibilidadMasFactible[0]," en la posicion ",tupla[0]," ",tupla[1]+contador )
                    pmatrix[tupla[0]][tupla[1]+contador]="|"
                    random.shuffle(posibilidadMasFactible)
                    intentos+=1

            else:
                contador+=1
        if(intentos>=15):
            pmatrix=limpiarLoQueHizo(tupla,pmatrix,posibilidadMasFactible)
            return (False,pmatrix)
        else:
            return (True,pmatrix)
    else:
        listaToques=sacarToquesVerticales(tupla[0],tupla[1],tupla[2],pmatrix)
        posibilidadMasFactible=escojerMejorPosibilidad(posibilidades,listaToques)
        posibilidadMasFactible=eliminarPresentes(posibilidadMasFactible,listaToques)
        contador=0
        random.shuffle(posibilidadMasFactible)
        metidos=0
        intentos=0
        while ((metidos!=len(posibilidadMasFactible))and intentos<15):
            if(pmatrix[tupla[0]+contador][tupla[1]]=="|"):
                pmatrix[tupla[0]+contador][tupla[1]]=posibilidadMasFactible[0]
                #print(pmatrix[tupla[0]][tupla[1]+contador])
                if (verificar(tupla[0]+contador,tupla[1],tupla[2],pmatrix)):
                    print("dio true")
                    metidos+=1
                    contador+=1
                else:
                    print("va a hacer cambio porque no es valido el num",posibilidadMasFactible[0]," en la posicion ",tupla[0]," ",tupla[1]+contador )
                    pmatrix[tupla[0]+contador][tupla[1]]="|"
                    random.shuffle(posibilidadMasFactible)
                    intentos+=1

            else:
                contador+=1
        if(intentos>=15):
            pmatrix=limpiarLoQueHizo(tupla,pmatrix,posibilidadMasFactible)
            return (False,pmatrix)
        else:
            return (True,pmatrix)
```

Fig. 6. Función de meterEnMatriz

III. Experimentos y discusión

A. *Hilos*

La utilización de los hilos en la busqueda de la solución de un kakuro en la teoría, parece fiable, pero a la hora de aplicarlos en la solución crea problemas en la solución, usualmente crea resultados invalidos en el kakuro, como lo son numeros repetidos en filas y columnas, o bien , sumas que dan respuesta correcta en una dirección pero no es su opuesta inmediata.En la figura 7 se muestra el codigo utilizado en la aplicación de los hilos(deamon). El poder paralelizar el calculo de las posibilidades de cada grupo de casillas aumenta el tiempo de calculo, pero al multiples estar trabajando sobre una misma matriz, provoca las inconsistencias anteriormente mencionadas.

```
def daemon(tupla,tipo):
    #si es tipo 1 es hor, tipo es ver
    #print (getPossibleNumbers(tupla[3],tupla[2]))
    posibilidades=obtenerPosibilidades(tupla[3],tupla[2])
    if (tipo==1):
        with ingresarResult_lock:
            const=0
            for x in range(tupla[2]):
                if((posibilidades[0][x] in gastados[tupla[1]+x]) and (result[tupla[0]][tupla[1]+const] == "|" or  result[tupla[0]][tupla[1]+const] == "O")):
                    result[tupla[0]][tupla[1]+const]=posibilidades[0][x]
                    const+=1
                    with quitarGastados:
                        gastados[tupla[1]+x].remove(posibilidades[0][x])
                        print("la columa ",(tupla[1]+ x)," le quedan por gastar: ",gastados[tupla[1]+x])
                else:
                    if (result[tupla[0]][tupla[1]+const] == "|"):
                        result[tupla[0]][tupla[1]+const]="O"
                    const+=1

    else:
        with ingresarResult_lock:
            const=0
            for x in range(tupla[2]):
                if((posibilidades[0][x] in gastados[tupla[1]]) and (result[tupla[0]+const][tupla[1]] == "|" or result[tupla[0]+const][tupla[1]] == "O")):
                    result[tupla[0]+const][tupla[1]]=posibilidades[0][x]
                    const+=1
                    with quitarGastados:
                        gastados[tupla[1]].remove(posibilidades[0][x])
                        print("la columa ",(tupla[1])," le quedan por gastar: ",gastados[tupla[1]])
                else:
                    if(result[tupla[0]+const][tupla[1]] == "|"):
                        result[tupla[0]+const][tupla[1]]="O"
                    const+=1

    print("----------------------------------------")
    for x in result:
        print (x)
```

Fig. 7. Función de deamon (Hilos)

B. *Forks*

Los forks son una herramienta que ayuda a mejorar considerablemente el rendimiento del programa. Su funcionamiento consiste en la copia idéntica de un proceso que se está llevando a cabo, el cual será el proceso hijo, incluyendo todas las variables que se declararon antes de la bifurcación del proceso, a excepción de la que guardará el resultado una vez que termine el proceso hijo y su ejecución continua en la línea inmediata después de la duplicación del proceso.
La implementación de este método no pudo ser implementada en este proyecto, sin embargo pudo haber sido de mucha utilidad a la hora de realizar los procesos, mejorando los tiempos de respuestas realizando múltiples tareas a la vez.