II Segundo Proyecto Programado Backtracking

Daniel Alvarado Bonilla
Instituto Tecnológico de Costa Rica
Ingeniería en Computación
2014089192
Email: daniel.alvarado.bonilla@gmail.com

Roberto Rojas Segnini
Instituto Tecnológico de Costa Rica
Ingeniería en Computación
2016139072
Email: rojassegniniroberto@gmail.com

***Resumen*—ACA VA EL ABSTRACT**
***Index Terms*—Big O, Matrix, Algorithm, Kakuro, Backtracking, Permutations.**

I. Introducción

Existen muchiísimos tipos de algoritmos y un programador se define en cual tipo de algoritmo se escoge para resolver un determinado problema que tiene una variedad soluciones. Se debe entender el por qué de la escogencia de este tipo. Es decir, en ocaciones no se quiere utilizar el algoritmo mas eficiente, esto no quiere decir que se tiene que utilizar un algoritmo mal fundamento, si no todo lo contrario, poder saber cuando se debe hacer uso de un algoritmo que no se basa en su eficiencia.

Un o de estos tipos de algoritmos es llamado Backtracking o "Vuelta Atras". Este algoritmo se basa en construir soluciones parciales a medida que se progresa en el arbol de opciones de nuestro problema. En otras palabras es una técnica de programación para hacer una búsqueda sistemática a traáves de todas las posibles configuraciones de nuestro problema. Todos los algoritmos de backtracking siguen un mismo patrón, pero varía un poco con el problema.

Se puede observar su forma genérica en la sección de Pseudocódigos.

En el presente trabajo se investigo, trabajo para crear una pequeña aplicación con el fin de que este genere Kakuros. Un kakuro es un enigma lógico que es semejante al conocido Crucigrama. Pero este no usa letras ni palabras, si no, números. Se les conoce también como Suma Cruzada, es un juego bastante popular en Japón. El algoritmo de backtracking explicado anteriormente es utilizado para resolver diferentes Kakuros. Para mejor la eficiencia del mismo, se inventaron diferentes algoritmos mas pequeños de "poda"para asi disminuir de manera significante el árbol de posibles soluciones. Para las diferentes funciones se hizo un análisis de O grande y se realizaron distintos experimentos.

II. Pseudocódigo

III. Complejidad de los Algoritmos

Orden de $O(f(n))$ dde Funcion de Poda:

Este algoritmo tiene en sí bastantes funciones pequeñas. Cada una siendo fundamental para la función de poda. Las más relevantes se explicarán a continuación son:

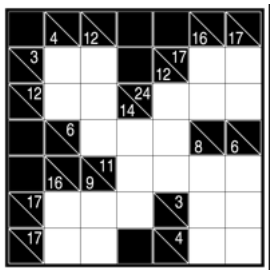

Figura 1. Kakuro

Algorithm 1 Backtracking Generic Algorithm

```
1: function BACKTRACKING(v[1..k] )
2:                                  ▷ v es un vector kprometedor
3:
4:     si v es una solucion entonces escribir v
5:     for  para cada vector(k + 1)−prometedor w do
6:         tal que w[1..k] = v[1..k]
7:         hacer backTracking(w[1..k + 1])
8:     end for
9: end function
```

I SubstractVal() es una función que toma el lugar donde se puede colocar un valor$[1.,9]$, recibe también este valor que se utilizará las sumas a las que pertenece, el mejor caso es que sea una intersección de dos sumas, asi la lista de valores seria menor. Toma esta posición y se mueve ya sea de forma vertical u horizontal, hasta buscar la suma a la que pertenece esa posición. Al encontrar el vector $v[sumaVertical, sumaHorizontal]$ se desplaza contando los espacios disponibles que tiene esta suma. Esto es un comportamiento lineal, es decir $O(n)$ al recorrer una lista. Lo interesante de este algoritmo es que una vez obtenida la cantidad de espacios, se recorre de nuevo desde la posición en la que se encuentra el vector v mencionado anteriormente, y suma los las casillas que ya tienen un valor. Y por cada casilla que tenga un valor, se aumenta un contador llamado newSpaces. Este es otro comportamiento lineal, es decir hasta el momento es un $O(2n)$. Este paso es sumamente importante, por que resta a la suma que se tiene que llegar, lo que se tiene hasta el momento. En otras palabras, si se debe llegar a sumar un

Algorithm 2 Solve - Backtracking Algorithm implemented

```
1: function SOLVEKAKURO(kakuro )
2:     ▷ kakuro es una matriz, la cual contiene el kakuro sin solución
3:
4:     if noEmptySpaces(kakuro) then
5:         if isKakuroSolved(kakuro) then
6:             return True
7:         else
8:         end if
9:         return False
10:    else
11:    end if
12:    position=getNextPosition(kakuro)
13:    row=position[0]
14:    column=position[1]
15:    num1 = getNumberLeft(kakuro, position)
16:    if num1 == 0 then
17:        num1 = -25
18:    end if
19:    num2 = getNumberUp(kakuro, position)
20:    if num2 == 0 then
21:        num2 = -25
22:    end if
23:    deleteRepeatedValues(kakuro,getIntersection
24:    (getValues(kakuro,position,num1,num2),
25:    getValuesList(num1,num2,kakuro,position)),position)
26:    if values==[] then
27:        return False
28:    end if
29:    for i en el largo de values do
30:        value = values[i]
31:        kakuro[row][column] = value
32:        if solveKakuro(kakuro) then
33:            return True
34:        else
35:            kakuro[row][column] = BLANK_ SPACE
36:        end if
37:    end for
38:    return False
39: end function
```

Algorithm 3 substract Values Algorithm

```
1: function SUBSTRACTVAL(k[1..k],p[x, y],sum,bool )
2:     newSum = 0
3:     newSpaces = 0
4:     if left then
5:         row = p[0]
6:         col = p[1]
7:         while col>= 0 do
8:             if k[row][col] es vector then
9:                 spaces = getSpaces()
10:                break
11:            end if
12:            col−
13:        end while
14:        col +1
15:    end if
16:    for i en rango de spaces do
17:        if k[row][col+i] tiene un valor entonces then
18:            spaces − = 1
19:            sum − = [el valor que este en esa posición]
20:        end if
21:    end for
22:    if not left then
23:        row = p[0]
24:        col = p[1]
25:        while col>= 0 do
26:            if k[row][col] es vector then
27:                spaces = getSpaces()
28:                break
29:            end if
30:            row−1
31:        end while
32:        row +1
33:    end if
34:    for i en rango de spaces do
35:        if k[row+i][col] tiene un valor entonces then
36:            spaces − = 1
37:            sum − = [el valor que este en esa posición]
38:        end if
39:    end for
40: return sum,spaces
41: end function
```

total de 27 en cuatro espacios, y ya se tiene 2 espacios utilizados, se toman los valores de estos dos espacios, se suman y luego se le restan al 27. Si se tenía 3 y 7 la nueva suma a la que se tiene que llegar es 17. Por último este numero y los espacios nuevos (2) son retornados. Hasta el momento se tiene $O(2n)$.
Ver Algoritmo 3.

II La función getValuesList es aquella que interpreta los valores retornados por substractVal(). Este primero revisa si dicha posición donde se colocará el valor es intersección de una suma en forma vertical y/o forma horizontal. Si pertenece a alguna de las anteriores, se verifica si tiene y si solo un espacio disponible, se utiliza una pequeña funcion la cuál retorna de un diccionario creado, donde este de llaves la cantidad de espacios disponibles, para cada de estas llaves tiene una llave secundaria la cual es la suma que puede formarse. Por último, tiene un vector de vectores con las posibles combinaciones de valores para obtener esta suma en dicha cantidad de espacios. Al obtenerla, se verifica que la suma, la cual es el unico valor que puede colocarse en la posición dada, si no esta en estas combinaciones, se hace backtracking. Retorna una lista vacía, donde el algoritmo principal ve que no tiene opciones que utilizar y se devuelve. Si la cantidad de espacios es diferente a uno, usando la función que

Algorithm 4 Get Values Algorithm

```
1: function GETVALUESLIST(hSum vSum,k[1..k],p[x, y] )
2:     sumUp,spacesUp = substractValues(k,p,vSum,False)
3:     sumLeft,spacesLeft = substractValues(k,p,hSum,True)
4:     if si el valor forma una suma vertical then
5:         if spacesUp = 1 then
6:             if Si es una intersección de dos sumas then
   return [sumUp]
7:             end if
8:             if spacesLeft =1 then
9:                 if si ambas sumas son iguales then
10:                    return [sum]
11:                else
12:                    return vacío
13:                end if
14:            else
15:                combinaciones = getCombinations()  ▷
   retorna un vector de combinacinoes que vienen de un
   diccionario
16:                if sumUp esta en combinaciones then
17:                    return [min de sumLeft y sumUp] si
   no []
18:                end if
19:            end if
20:        else
21:            combinaciones = getCombinations()
22:            for por combinacion en combinaciones] do
23:                Agregar cada valor a un
24:                vector llamado valoresUp
25:            end for
26:        end if
27:        if el valor forma una suma horizontal then
28:            if SpacesLeft = 1 then
29:                if Si es una intersección de dos sumas then
   return [sumLeft]
30:                end if
31:                if SpacesUp =1 then
32:                    if si ambas sumas son iguales then
33:                        return [sum]
34:                    else
35:                        return vacío
36:                    end if
37:                else
38:                    combinaciones = getCombinations()  ▷
   retorna un vector de combinacinoes que vienen de un
   diccionario
39:                    if sumUp esta en combinaciones then
40:                        return [min de sumLeft y sumUp]
   si no []
41:                    end if
42:                end if
43:            else
44:                combinaciones = getCombinations()
45:                for por combinacion en combinaciones] do
46:                    Agregar cada valor a un
47:                    vector llamado valoresLeft
48:                end for
49:            end if
50:        end if
51:    end if
52:    return intersección entre valoresLeft y valoresRight
53: end function
```

obtiene las combinaciones, se agregan todos los valores posibles a una lista. Esto se hace para cada suma, siempre y cuando la posición en la que se encuentra el algoritmo sea intersecciónTodo lo dicho anteriormente es de un orden lineal, sumando $O(2n + n)$. Esta funci
on retorna este vector con los posibles valores que pueden ser utilizados y así la función principal pruebe con estos. Ver Algoritmo 4.

El orden de la función de poda es de $O(n)$ ya que 3 es una constante que no afecta el comportamiento de dicha función.
Orden de $O(f(n))$ de Permutaciones:
FALTA
Orden de $O(f(n))$ de Backtracking:
El algoritmo de Backtracking implementado se deduce que tiene un orden $O(n^k)$. N es la cantidad de opciones que pueden colocarse en un determinado espacio del Kakuro. Al ser un árbol de soluciones, N el nivel del nodo, donde cada nodo tiene sus propios nodos, esto implica $n(n-1)(n-2)...(n-k+1)$. O grande considera el peor de los casos y este seria recorrer cada nodo hasta su ultimo nivel, es decir recorrer el árbol a profundidad. Siendo un comportamiento de n^k. Existe una función la cual recorre la matriz del Kakuro para encontrar una posición libre, esta parte es $O(n^2)$. En total, $O(n^k + n^2)$ pero al como O Grande utiliza el peor de los casos, se concluye $O(n^k)$

IV. Experimentos

IV-A. Experimento 1

Referencias

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.