Análisis de algoritmos y resultados Proyecto 2 Kakuro

Nelson Rojas Obando y Jose Luís Rodriguez
Estudiantes de
Ingeniería en Computación
Instituto Tecnológico de Costa Rica
Carnés: 2016062840, 2016093725

***Abstract*—Este proyecto consiste en la generación y solución de kakuros. Un kakuro es tipo de sudoku-crucigrama. Compuesto por casillas donde deben de completarse por filas y columnas hasta obtener los números indicados en los extremos. Además, se presenta el análisis de los algoritmos de generación y solución de kakuros por medio de la técnica de BackTracking; así como, los tiempos de ejecución y el comportamiento cuando se utiliza paralelización.**

I. Introducción

Este documento presenta los resultados teóricos y prácticos del análisis realizado a los algoritmos implementados para la generación y solución de Kakuros.

Los algoritmos implementados se componen de un algoritmo para la generación del tablero de juego, posteriormente un algoritmo para la solución del tabledo del kakuro generado, un algoritmo de permutación de números necesario para la generación de kakuros y por último, el algoritmo de la poda del árbol implícito.

Además, se llevaron a cabo experimentos para medir el tiempo que tarda el algoritmo de backtracking en ser ejecutado por completo en distintos escenarios como; implementanto un número de hilos, y ejecutandolo con el uso de forks y su ejecución normal sin paralelización.

Octubre 19, 2017

II. Análisis teórico de los algoritmos

Para el algoritmo de generación de tableros de juego se requiere una entreada "n" la cual corresponde al tamaño del tablero, se dispone de distintos tamaños como:

- 10x10
- 11x11
- 12x12
- 13x13
- 14x14
- 15x15
- 16x16
- 17x17
- 19x19
- 20x20

Por lo que el N está dado la dimensión de columnas y filas del tablero seleccionado.

A. Algoritmo de genenración

Algoritmo de generación del tablero de juego:

```
from random import randint
class Tablero:
    def __init__(self, tamanio, x):
                #Complejidad
        global tablero, matrizTablero,
            ventanaKakuro, matrizF

        fila=[]
        columna=[]
        #n
        for i in range(tamanio):
            #n
            for j in range(tamanio):
                #c*n^2
                fila+=[0]
            #c*n
            matrizTablero+=[fila]
            #c*n
            fila=[]
            #c*n
            tablero[0][i].set_background(
                "black")
            #c*n
            matrizTablero[0][i]="&"
            #c*n
            tablero[i][0].set_background(
                "black")
            #c*n
            matrizTablero[i][0]="&"

        #Porcentaje del tablero que se va
            a bloquear, puede variar
        #c
        cantBloqueadas=tamanio*2
        #2n
        for h in range(cantBloqueadas):
            #c*2n
            rand_x=randint(0,tamanio-1)
            #c*2n
            rand_y=randint(0,tamanio-1)
```

```
        #((n-1)*(n-1))
        while (matrizTablero[rand_x][
            rand_y]=="&"):
            #c*2n((n-1)*(n-1))
            rand_x=randint(0,tamanio
                -1)
            #c*2n((n-1)*(n-1))
            rand_y=randint(0,tamanio
                -1)
        #c*2n
        matrizTablero[rand_x][rand_y
            ]="&"
        #c*2n
        tablero[rand_x][rand_y].
            set_background("black")
    #n
    for i in range(tamanio):
    #n/5
        for j in range(tamanio//5):
            #c*(n^2)/5
            inicio = self.verificar(
                matrizTablero[i],
                tamanio)
            if (inicio != -1):
                #c*(n^2)/5
                x = randint(inicio,
                    tamanio-1)
                #n*(n-1)
                while (matrizTablero[
                    i][x] == "&"):
                    #c*n^2(n-1)
                    x = randint(
                        inicio,
                        tamanio-1)
                #c*(n^2)/5
                matrizTablero[i][x]="
                    &"
                #c*(n^2)/5
                tablero[i][x].
                    set_background("
                    black")
    #n
    for i in range(tamanio):
        #n
        for j in range(tamanio):
            #c*n^2
            columna+=[matrizTablero[j
                ][i]]
        #n/5
        for h in range(tamanio//5):
            #c*(n^2)/5
            inicio = self.verificar(
                columna, tamanio)
            if (inicio!=-1):
                #c*(n^2)/5
                x = randint(inicio,
                    tamanio-1)
                #n*(n-1)
                while (matrizTablero[
                    x][i]=="&"):
                    #c*n^2(n*(n-1))/5
                    x = randint(
                        inicio,tamanio
                        -1)
                #c*(n^2)/5
                matrizTablero[x][i]="
                    &"
                #c*(n^2)/5
                tablero[x][i].
                    set_background("
                    black")
        #c*n
        columna=[]
    #n
    for i in range(tamanio):
        #n
        for j in range(tamanio):
            try:
                if (matrizTablero[i-
                    1][j]=="&" and
                    matrizTablero[i+
                    1][j]=="&"
                    and matrizTablero
                        [i][j-1]=="&
                        " and
                        matrizTablero[
                        i][j+1]=="&"
                        ):
                    #c*n^2
                    matrizTablero[i][
                        j]="&"
                    #c*n^2
                    tablero[i][j].
                        set_background
                        ("black")
            except:
                #c*n^2
                matrizTablero[i][j]="
                    &"
                #c*n^2
                tablero[i][j].
                    set_background("
                    black")
def verificar(self, fila, tamanio):
    cont=0
    inicio=-1
    #n
    for i in range(tamanio):
        if (fila[i]==0):
            if (cont==0):
                #c*n
```

```
                inicio=i
            #c*n
            cont += 1
        if (fila[i]!=0 and cont<=9):
            #c*n
            inicio=-1
            #c*n
            cont=0
    if cont > 9:
        #c
        return inicio
    else:
        #c
        return -1
```

Barómetro:
Se tomó como barómetro la expresión

$$\frac{c*n^2(n*(n-1))}{5}$$

de donde

$$\frac{c*n^2(n^2-n))}{5}$$

$$\frac{c*n^4-c*n^3}{5}$$

Por lo que O(n) está dado por:

$$O(n) = n^4$$

B. Algoritmo de permutación de números

La entrada ”n” de este algorigmo está dada por el tamaño de la lista de valores que se buscan permutar. Su producto generado es una combinación de los elementos de la lista en las posibles combinaciones que se generan.

Algoritmo de permutación de números:

```
def inserta(x, lst, i):
    #c
    return lst[:i] + [x] + lst[i:]

def inserta_multiple(x, lst):
    #c (n + 1)
    return [inserta(x, lst, i) for i in
        range(len(lst) + 1)]

def permuta(c):
    if len(c) == 0:
        #c
        return [[]]
    #c * n (n + 1)
    return sum([inserta_multiple(c[0], s)
        for s in permuta(c[1:])],[])

def potencia(c):
    if len(c) == 0:
        #c
        return [[]]
    #c * n ^ 2
    r = potencia(c[:-1])
    #c * n * (n ^ 2)
    return r + [s + [c[-1]] for s in r]

def combinaciones(c, n):
    #n * (n ^ 2)
    return [s for s in potencia(c) if len
        (s) == n]

def generarCombinaciones (c, n):
    #c * n ^ 2 (n + 1) * (n ^ 2)
    return sum([permuta(s) for s in
        combinaciones(c, n)], [])

def imprime_ordenado(c):
    #n
    for e in sorted(c, key=lambda s: (len
        (s), s)):
        #c * n
        print(e)
```

Barómetro:
Se tomó como barómetro la expresión

$$c*n^2(n+1)*(n^2)$$

De donde se obtiene

$$c*n^5+c*n^4$$

Por lo que el orden de este algoritmo sería:

$$0(n) = n^5$$

C. Podas del algoritmo de backtracking

Se implementaron podas para verificar la validez y determinar si un camino o combinación de elementos es prometedor
Podas de bactracking:
La entrada de las podas está dado por el tamaño del tablero visto como una matriz cuadrada.

```
from Combinaciones import
    listaDcombinaciones
def is_valid(brd, tamanio):
    #c
    lst = []
    #n
    for i in range(tamanio):
        #n
        for j in range(tamanio):
            #c * n ^ 2
            if(validarAmp(brd[i][j]) ==
                False):
                if(int(brd[i][j]) == 0):
                    continue
                else:
```

```
                    if(int(brd[i][j]) in
                        lst):
                        return False
                    else:
                        lst.append(int(
                            brd[i][j]))
            else:
                lst = []
    return True

def validarAmp(string):
    #c
    if ("&" in str(string)):
        return True
    else:
        return False

def is_posible(prueba, objetivo, i, j,
    tamanio):
    #c
    global matrizF
    lst = listaDcombinaciones[objetivo]
    cont = 0
    h = j
    lstOptima = []
    #n
    while(h < tamanio and validarAmp(
        matrizF[i][h]) == False):
        #c * n
        cont += 1
        h += 1
    #m
    for opt in lst:
        #c * m
        if(len(str(opt)) == cont):
            lstOptima.append(opt)
    #c
    if(len(lstOptima) == 0):
        lstOptima = copy(lst)
    #m
    for opt2 in lst:
        #c * m
        if(str(prueba) in str(opt2)):
            #c ^ 5 * m (n ^ 4)
            if(str(prueba) == verifica(
                prueba, cont, i, j,
                tamanio)):
                return True
    return False

def verifica(prueba, i, j, tamanio):
    #c
    c = 0
    cantperm = int(prueba)*pow(tamanio,
        2)
    pos = matrizTablero[i][j]
    #c * n ^ 2
    while(c < cantperm):
        #c ^ 2 (n ^ 2) * (n ^ 2)
        if(verificarRepetidosArriba(i, j,
            pos) == True):
            return False
        #c ^ 2 (n ^ 2) * (n ^ 2)
        if(verificarRepetidosAbajo(i, j,
            pos, tamanio) == True):
            return False
        c += 1
    return pos

def verificarRepetidosAbajo(i, j, valor,
    tamanio):
    #n ^ 2
    while (matrizTablero[i][j] != "&"):
        #c * n ^ 2
        if (matrizTablero[i][j] == valor)
            :
            return False
        if (i + 1 < tamanio):
            i += 1
        else:
            return True
    return True

def verificarRepetidosArriba(i, j, valor)
    :
    #n ^ 2
    while (matrizTablero[i][j] != "&"):
        #c * n ^ 2
        if (matrizTablero[i][j] == valor)
            :
            return False
        if (i - 1 >= 0):
            i -= 1
        else:
            return True
    return True
```

El análisis correspondiente de la complejidad de las podas implementadas está dado por:

Función is_valid

Barómetro

$$c * n^2$$

Por lo que

$$O(n) = n^2$$

Función validaAmp

Orden constante

Función is_posible

Barómetro

$$c^5 * m(n^4)$$

Por lo que

$$O(n) = m * n^4$$

Donde m está dado por el tamaño de la lista de combinaciones que se compara

Función verifica
Barómetro

$$c^2(n^2) * (n^2)$$

Por lo que

$$O(n) = n^4$$

Función verificarRepetidosAbajo
Barómetro

$$c * n^2$$

Por lo que

$$O(n) = n^2$$

Función verificarRepetidosArriba
Barómetro

$$c * n^2$$

Por lo que

$$O(n) = n^2$$

D. Algoritmo de backtracking para la solución de kakuros

Este algoritmo utiliza las podas que se indicaron anteriormente. Requiere como entradas el tamaño del tablero de juego que se está utilizando; además, lleva control de la cantidad de espacios vacios restantes y de manera recursiva se ejecuta hasta llegar a no tener más espacios vacios siempre y cuando el camino seleccionado, es decir la combinación tomada sea posible y valida

Algoritmo de bactracking:

```
from itertools import product

def backtracking(brd, tamanio, empties):
    if empties == 0:
        #c * n ^ 2
        return is_valid(brd, tamanio)
    #c
    objetivo = 0
    #n ^ 2
    for row, col in product(range(tamanio
        ), repeat=2):
        #c * n ^ 2
        if(validarAmp(brd[row][col]) ==
            True):
            if(len(brd[row][col]) > 1):
                lst = brd[row][col].split
                    ("&")
                if(lst[0] != ""):
                    objetivo = int(lst
                        [0])
            continue
        #c * n ^ 2
        if (validarAmp(brd[row][col]) ==
            False):
            if(int(brd[row][col]) == 0):
                brd2 = copy(brd)
                #c = 9
                for num in
                    [1,2,3,4,5,6,7,8,9]:
                    #9 * m (n ^ 4)
                    if(is_posible(num,
                        objetivo, row, col
                        , tamanio) == True
                        ):
                        brd2[row][col] =
                            num
                        tablero[row][col
                            ].set_text(
                            False, num)
                        #c * n ^ 2 * m (n
                            ^ 4)
                        if(is_valid(brd2,
                            tamanio) ==
                            True and
                            backtracking(
                            brd, tamanio,
                            empties-1) ==
                            True):
                            return True
                        brd2[row][col] =
                            0
                    else:
                        continue
    return False
```

Barómetro:
Se tomó como barómetro la expresión

$$c * n^2 * m(n^4)$$

De donde

$$O(n) = *m(n^6)$$

1	2	1
2	4	2
1	2	1

E. Resultados luego de la implementación de Threads y Forks

Tanto los hilos como los forks son muy utilizados en Ciencias de la Computación con el fin de mejorar el rendimiento de los programas, esto debido a que paralelizan procesos a cambio de una mayor cantidad de recursos del sistema consumidos. A pesar de que ambos presentan funciones muy similares, existe una clara diferencia, los hilos consumen los recursos que el sistema operativo tiene destinados para la ejecución de ese programa, mientras que los procesos son independientes en este aspecto. Esto permite

que un proceso creado pueda contener hilos ejecutándose de forma paralela, tal y como se ilustra a continuación:

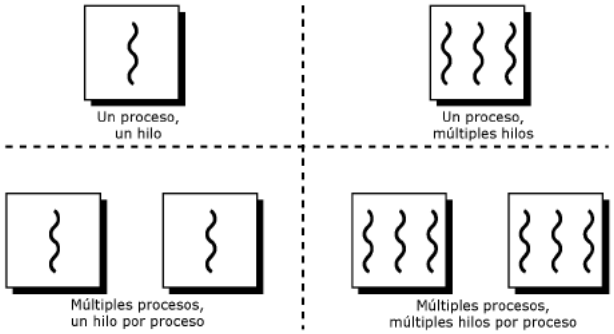

Fig. 1. Funcionamiento de hilos y procesos

Existen diversas variables que ayudan a tomar la decisión correcta sobre cuál escoger al momento de desarrollar un programa. Entre ellas se encuentran: el sistema operativo, el hardware de la computadora y el lenguaje en el que son programados. Es prácticamente imposible controlar las decisiones sobre las prioridades que asigna el SO a sus recursos disponibles, sin embargo las otras dos variables expuestas anteriormente sí, especialmente la última.

Dependiendo que cuál es el lenguaje de programación elegido para desarrollar el proyecto, así como de cuál va a ser la función que se va a ejecutar es más conveniente utilizar un método u otro. Un claro ejemplo de lo anterior es el caso de Python 3x, pues al momento de integrar hilos en el programa su rendimiento disminuyó, contrario a lo que se esperaría en un principio. Luego de hacer exhaustivas pruebas y búsqueda del problema, nos encontramos con que el interprete de Python presenta ciertas restricciones que unicamente permiten que un hilo se ejecute a la vez, esto es llamado ”GIL” por sus siglas en ingles (Global Interpreter Lock). Con estas restricciones no quiere decir que no se puedan implementar hilos, pues el interprete unicamente no permite la ejecución paralela de código (como en nuestro caso), sin embargo otras funciones como la entrada/salida de datos si es permitida. Esto es expuesto en más profundidad en el paper de David Beazley presentado en el PyCon 2010 que se puede encontar en el siguiente link: http://www.dabeaz.com/python/UnderstandingGIL.pdf

Por lo tanto, al menos en Python, los hilos son utilizados para mantener una interfaz funcional y para la entrada/salida tanto de archivos locales así como de red. Pues de otra forma, la implementación de estos empeoraría el rendimiento del programa, tal y como ocurrió en nuestro proyecto al momento de su incorporación y que se muestra en la figura 2

Como se explicó anteriormente, al implementar los hilos el rendimiento fue mucho menor a como lo era antes de estos.

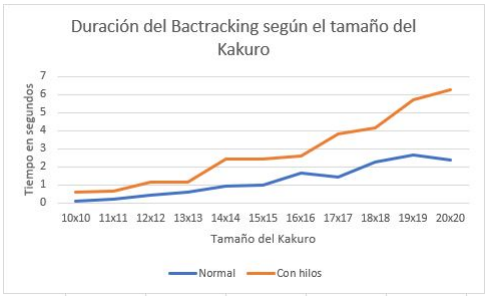

Fig. 2. Rendimiento de hilos en Python

En cuanto a los Forks o procesos, si bien no se puede hacer una comparación como la anterior, debido a que se presentaron ciertas dificultades con la interfaz en Tkinter, en las pruebas que se realizaron quedó demostrado que el algoritmo es más eficiente, aunque no en gran medida. Además, según la cantidad de Forks, el programa muere y por momentos la computadora también lo hacía. Las siguientes son imágenes que demuestran el comportamiento de la máquina al momento de ejecutar una gran cantidad de procesos:

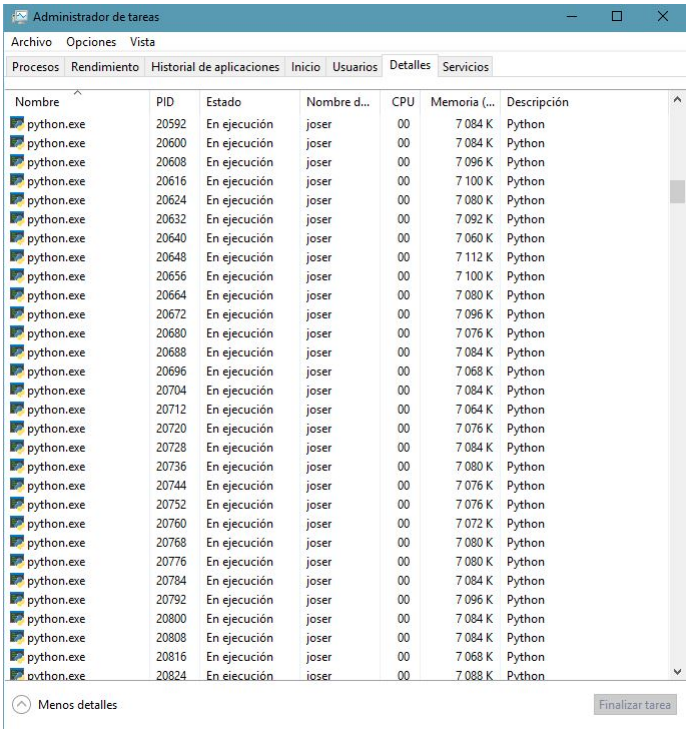

Fig. 3. Parte de los Forks puestos en ejecución

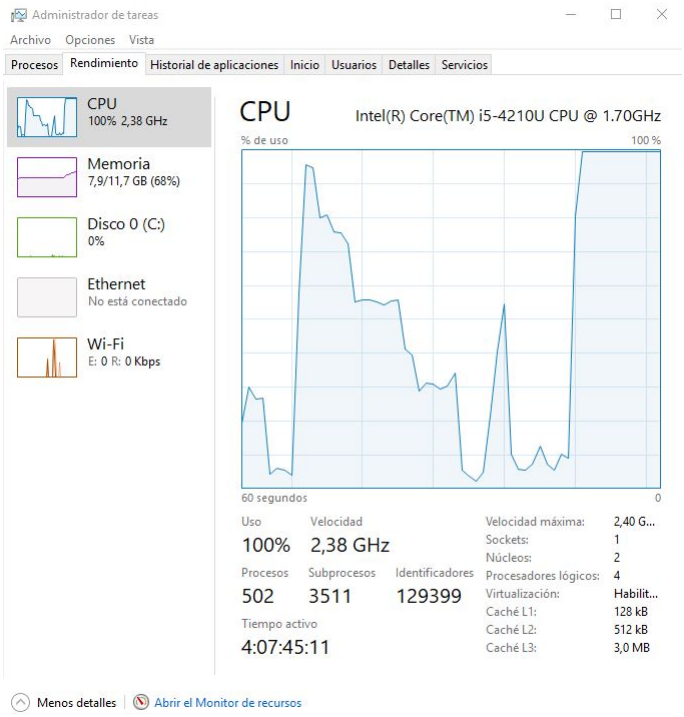

Fig. 4. Gráfico del CPU durante la ejecución de los Forks