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*—O Grande, Matriz, Algoritmo.**

I. Introducción

Existen muchísimos 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.

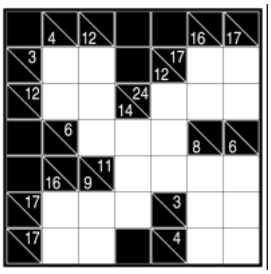


Figura 1. Kakuro

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
```

II. Pseudocódigo

III. Complejidad de los Algoritmos

III-A. Orden de $O(f(n))$ de Función 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:

I

1. 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

Algorithm 2 Solve - Backtracking Algorithm implemented

```
function SOLVEKAKURO(kakuro )
    ▷ kakuro es una matriz, la cual contiene el kakuro sin solución

    if noEmptySpaces(kakuro) then:
        if isKakuroSolved(kakuro) then:
            return True
        else:
            return False

            position=getNextPosition(kakuro)
            row=position[0]
            column=position[1]
            num1 = getNumberLeft(kakuro, position)
            if num1 == 0 then:
                num1 = -25
            end if
            num2 = getNumberUp(kakuro, position)
            if num2 == 0 then:
                num2 = -25
            end if
            deleteRepeatedValues(kakuro,getIntersection
            (getValues(kakuro,position,num1,num2),
            getValuesList(num1,num2,kakuro,position)),position)
            if values==[] then:
                return False
            end if
            for i en el largo de values do
                value = values[i]
                kakuro[row][column] = value
                if solveKakuro(kakuro) then:
                    return True
                else:
                    kakuro[row][column] = BLANK_ SPACE

                    return False

```

Algorithm 3 substract Values Algorithm

```
function SUBSTRACTVAL(k[1..k],p[x, y],sum,bool )
    newSum = 0
    newSpaces = 0
    if left then
        row = p[0]
        col = p[1]
        while col>= 0 do
            if k[row][col] es vector then
                spaces = getSpaces()
                break
            end if
            col−
        end while
        col +1
    end if
    for i en rango de spaces do
        if k[row][col+i] tiene un valor entonces then
            spaces − = 1
            sum − = [el valor que este en esa posición]
        end if
    end for
    if not left then
        row = p[0]
        col = p[1]
        while col>= 0 do
            if k[row][col] es vector then
                spaces = getSpaces()
                break
            end if
            row−1
        end while
        row +1
    end if
    for i en rango de spaces do
        if k[row+i][col] tiene un valor entonces then
            spaces − = 1
            sum − = [el valor que este en esa posición]
        end if
    end for
return sum,spaces
end function=0
```

sumar un 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.

2. 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 obtiene las combinaciones, se agregan todos los valores posibles a una lista. Esto

Algorithm 4 Get Values Algorithm

```
function GETVALUESLIST(hSum vSum,k[1..k],p[x, y] )
    sumUp,spacesUp = substractValues(k,p,vSum,False)
    sumLeft,spacesLeft = substractValues(k,p,hSum,True)
    if si el valor forma una suma vertical then
        if spacesUp = 1 then
            if Si es una intersección de dos sumas then return [sumUp]
            end if
            if spacesLeft =1 then
                if si ambas sumas son iguales then
                    return [sum]
                else
                    return vacío
                end if
            else
                combinaciones = getCombinations()  ▷ retorna un vector de combinacinoes que vienen de un diccionario
                if sumUp esta en combinaciones then
                    return [min de sumLeft y sumUp] si no []
                end if
            end if
        else
            combinaciones = getCombinations()
            for por combinacion en combinaciones] do
                Agregar cada valor a un
                vector llamado valoresUp
            end for
        end if
        if el valor forma una suma horizontal then
            if SpacesLeft = 1 then
                if Si es una intersección de dos sumas then return [sumLeft]
                end if
                if SpacesUp =1 then
                    if si ambas sumas son iguales then
                        return [sum]
                    else
                        return vacío
                    end if
                else
                    combinaciones = getCombinations()  ▷ retorna un vector de combinacinoes que vienen de un diccionario
                    if sumUp esta en combinaciones then
                        return [min de sumLeft y sumUp] si no []
                    end if
                end if
            else
                combinaciones = getCombinations()
                for por combinacion en combinaciones] do
                    Agregar cada valor a un
                    vector llamado valoresLeft
                end for
            end if
        end if
    end if
    return intersección entre valoresLeft y valoresRight
end function
```

Algorithm 5 Repeated Number Algorithm

```
function NUMBERREPEATED(num,k[1..k],p[x, y] )
    row = p[0]
    col = p[1]
    while el espacio sea diferente de vacio o la longitud do
        if si el numero que hay en la posicion es igual al valor recibido then
            return True
        end if
        Pasar a la siguiente posición
    end while
    return False
end function
```

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 función retorna este vector con los posibles valores que pueden ser utilizados y así la función principal pruebe con estos.
Ver Algoritmo 4.

3. La función numberRepeated() tiene un comportamiento lineal. Esta función recibe un valor y toma una desición. Se mueve de forma horizontal y vertical, lo cual es recorrer un vector, buscando todos los numeros que tiene y comparandolos con el valor recibido. Si el valor esta repetido, la función retorna True. La O grande de esta pequeña función es $O(n)$, 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.

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.

III-B. Orden de $O(f(n))$ de Permutaciones:

Para resolver el problema no se utilizo un algoritmo específico que obtuviese todas las permutaciones. Lo implementado fue, ir rellenando espacio por espacio un número a la vez. La función de la poda, analiza y obtiene los posibles valores que pueden colocarse en cada espacio en blanco. Y para la lista total de valores, coloca ese valor en dicha posición y llama a la función principal de backtracking con solo un campo más resuelto. Donde el valor utilizado es prometedor hasta llegar a un punto donde la función de poda determina que para ese lugar con los numeros y espacios que existen hasta el momento ya no hayan opciones, esto resultando a hacer backtrack. Dicho lo anterior se considera que el algoritmo de permutar es una constante, donde puede basarse en una lista de numeros que debe recorrerse nunca mayor a una longitud de nueve.

III-C. *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: hilos*

La utilización de hilos es ampliamente utilizada para mejorar el tiempo de ejecución de programas y algoritmos. Es posible ejecutar programas (o pedazos de ellos) de manera concurrente, en algunos casos, y de manera paralela, en otros, mediante la utilización de hilos.

Un hilo ejecuta cierto pedazo de código de un programa de manera ”separadaz permite realizar múltiples operaciones casi simultáneamente. Muchas veces, las capacidades del sistema en el que se corre el programa (computadora), las características del lenguaje de programación utilizado y algunos otros aspectos dictan la manera en el que los hilos serán ejecutados.

En el caso pertinente a este proyecto, se implementaron hilos utilizando la librería **threading** de Python. Dicha librería permite crear y ejecutar hilos de manera fácil y sencilla, utilizando la función **Thread**.

Python es capaz de ejecutar hilos de manera paralela. Sin embargo, la implementación utilizada de los mismos corresponde a una manera simple, ajena del uso de piscinas de hilos y otras características, la cual corre los hilos, muy probablemente, de manera concurrente.

Hipótesis

A pesar de esto, es de esperar que la utilización de hilos mejore los tiempos de ejecución. Además, se tiene como hipótesis que conforme aumente la cantidad de hilos, el tiempo empeorará y el programa durará más ejecutándose.

Metodología

Para probar (o refutar) la hipótesis, se ejecutó el programa 1000 veces, con el objetivo de resolver un kakuro determinado (ver Figura 2). Para cada ejecución se varió la cantidad de hilos máximos que se podían utilizar. Las cantidades de hilos utilzadas son: dos, cinco, ocho, quince, cincuenta, ciento cincuenta, trescientos, quinientos, setecientos cincuenta y mil. Se guardó el tiempo de ejecución en milisegundos.

Resultados

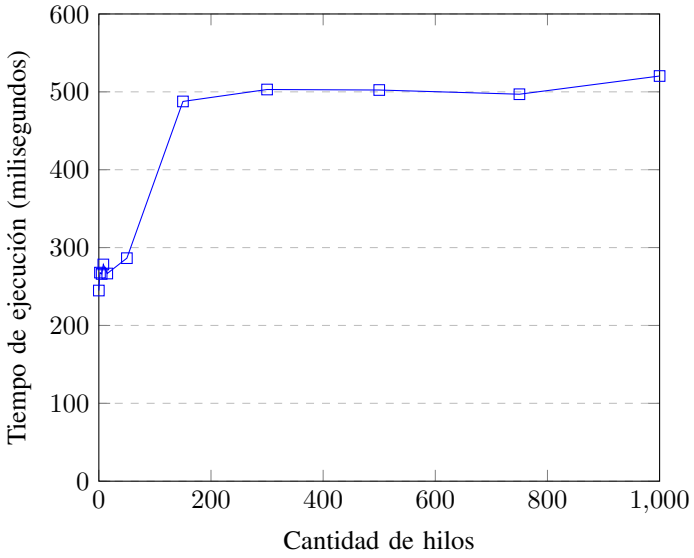
Los resultados demostraron que aumentar la cantidad de hilos empeora el tiempo de ejecución del programa. A pesar de que los tiempos no mostraban un incremento constante (en algunas partes el tiempo fluctuaba), la tendencia se mantenía a la alta conforme la cantidad de hilos aumentaba. El uso de 150 hilos hizo que se duplicara el tiempo de ejecución (244.842 milisegundos en el caso base (sin hilos), 487.662 con 150 hilos).

No se notó mejoría en los tiempos en ningún caso, en relación con la ejecución sin hilos. Las diferencias, sin embargo, entre el uso de 0 a 50 hilos fueron mínimas.

Entre los 350 y 750 hilos, el tiempo de ejecución sufrió una baja en los tiempos. De nuevo, estas diferencias son mínimas, pero demuestran que la utilización de hilos se ve sujeta a fluctuaciones de este tipo.

Gráficos y anexos

Comparación de tiempos de ejecuciónutilizando y sin utilizar hilos



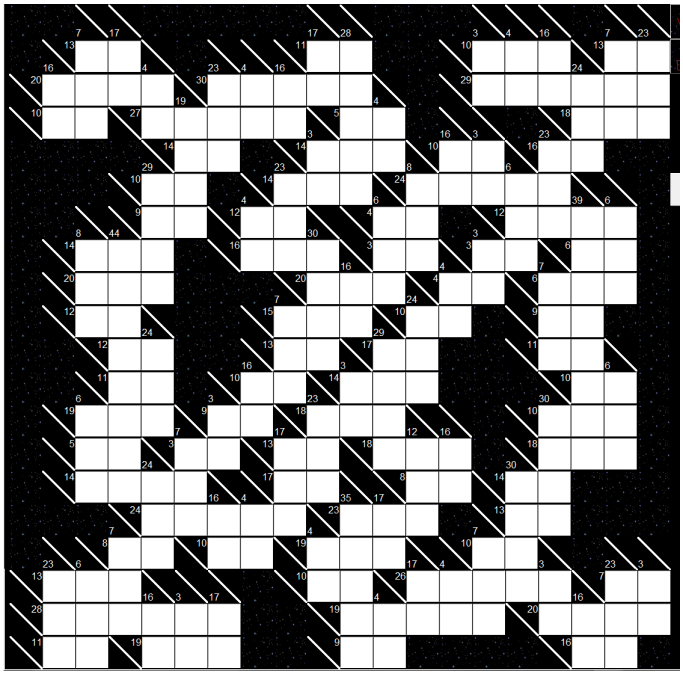


Figura 2. Kakuro utilizado

IV-B. Experimento 2: forks

Como se discutió anteriormente, los hilos permiten paralelizar procesos y código. Un hilo ejecuta cierto pedazo de código. Otra manera de paralelizar un proceso es por medio del uso de **forks**.

Un fork es un proceso que se ejecuta de manera independiente al proceso principal de un programa. El fork es una copia del proceso en el que fue llamado. Todas las variables, funciones y características del proceso padre son heredadas al proceso hijo.

Hipótesis

Como con el experimento anterior, el objetivo de implementar forks es determinar si hay una reducción significativa en el tiempo de ejecución. Sin embargo, es de esperarse que muchos procesos corriendo al mismo tiempo empeoren los tiempos de ejecución.

Metodología

Se ejecutó el programa con un total de cero, uno, dos, cinco y siete forks. Por cada valor de los mencionados, se corrió el programa diez veces y se extrajo un promedio de tiempo en milisegundos. El kakuro utilizado es el mismo que el del experimento anterior (ver Figura 2).

Resultados

Los forks resultaron ser mucho más pesados que los hilos. La generación de más de diez procesos simultáneos provocó que el programa corriera con dificultades. Después de ese punto, el programa por lo general se paralizaba y un reinicio de la computadora era necesario.

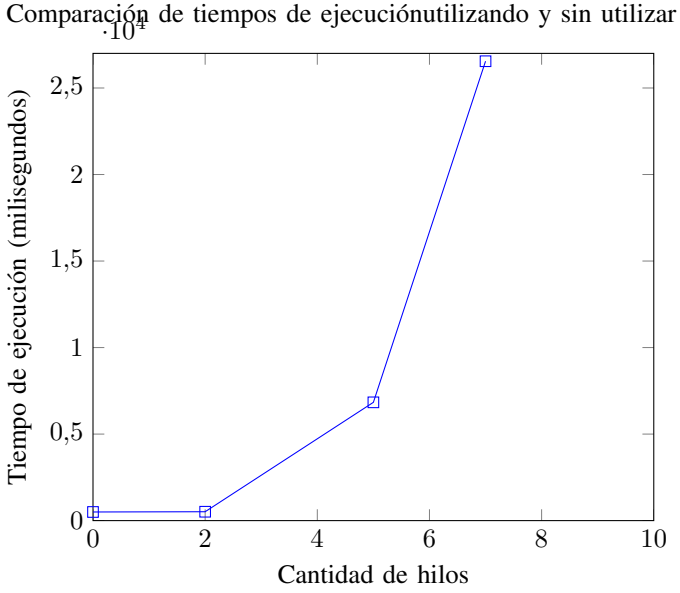
Como era de esperarse, el aumento de forks provocó un crecimiento sustancial del tiempo de ejecución. Entre el uso de dos y cinco forks, se observó un aumento de casi siete veces del tiempo. El mismo comportamiento se observó entre cinco y siete forks.

Aunque entre el uso de cero y un fork hubo una diferencia negativa en el promedio general (el tiempo aumentó al crear un fork), al analizar los tiempos individuales de las corridas se puede notar que, en algunos casos, el uso de un fork mejoró el tiempo de ejecución del programa. Es más, en la mayoría de las corridas el tiempo fue notablemente inferior. Sin embargo, el promedio general de las corridas con un fork, en el final, resultó ser mayor.

Más allá de la ventaja de usar un fork sobre prescindir de su uso, los forks presentaron un comportamiento inestable y de alto consumo computacional. Para un programa como el que corresponde a este proyecto (resolución de kakuros), los forks no presentaron una mejora significativa y las operaciones se vieron, en general, afectadas negativamente por el costo computacional de manejar los procesos simultáneamente.

Gráficos y anexos

Comparación de tiempos de ejecuciónutilizando y sin utilizar forks



IV-C. Experimento 3: tamaños y tiempos

Como se determinó anteriormente, el orden del algoritmo de backtracking para resolver los kakuros es, en el peor de los casos, n^k. Un orden exponencial posee un comportamiento de constante crecimiento (en el caso de los algoritmos). Este crecimiento es acelerado y pronunciado.

¿Se reflejará este comportamiento en la ejecución real del algoritmo? Es de esperarse que las tendencias temporales de las ejecuciones estén sujetas a la teoría. Sin embargo, diversos aspectos pueden influir para que el espacio teórico y el espacio real discrepen en temas de costo y ejecución de los algoritmos.

Además, es común que el tamaño de la entrada de un algoritmo determine el tiempo que durará ejecutándose. Un algoritmo con un comportamiento exponencial, durará más resolviendo un problema entre mayor sea la entrada. Lo anterior es cierto para algoritmos que trabajan con imágenes, por ejemplo, los cuales normalmente poseen una complejidad de n^2: entre más grande sea la imagen, más se tardará en recorrerla y ejecutar el algoritmo.

Los kakuros se parecen a las imágenes: son matrices (en este caso cuadradas) que contienen valores en cada una de sus casillas. ¿Se comportarán de manera similar? A primera instancia, es justo suponer que el tiempo necesario para resolver un kakuro debería incrementar según el tamaño del mismo. ¿Es esto cierto? Los resultados del experimento demuestran que no.

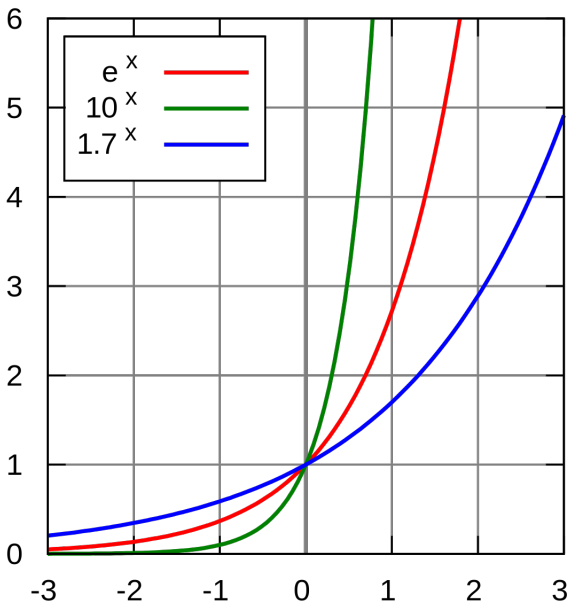


Figura 3. Ejemplo de funciones exponenciales

Metodología

Para comprobar la hipótesis planteada, se ejecutó el algoritmo de resolución de kakuros en tres puzles diferentes, con tamaños distintos: de cinco por cinco; de diez por diez; dos de veinte por veinte y de treinta por treinta.

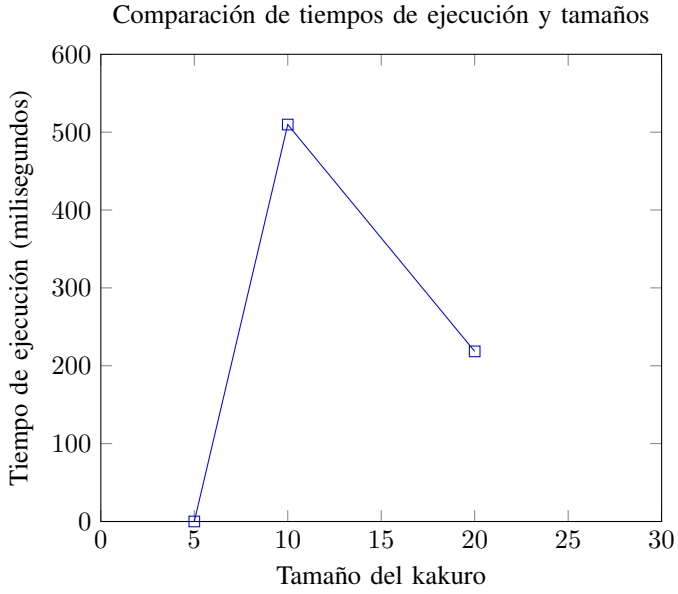
Resultados

Los resultados fueron los siguientes (tiempos de ejecución dados en milisegundos): para el kakuro 5x5, 0.058; para el 10x10, 509.745; para el primero 20x20, 218.498; para el segundo 20x20, 277.822.

Los resultados del experimento muestran que los dos kakuros de mayor tamaño fueron resueltos en la mitad del tiempo que el kakuro de 10x10. Es decir, la mitad del tiempo para el doble de tamaño.

Lo evidenciado por los experimentos se debe al hecho de que la complejidad del algoritmo para los kakuros no depende directamente del tamaño del kakuro. En lugar de esto, la complejidad de ve influenciada por la cantidad de casillas que pueden albergar números, las sumas, las intersecciones, la cantidad de permutaciones que puede poseer un conjunto de casillas, la cantidad de soluciones posibles, entre otros factores.

Gráficos y anexos



El gráfico anterior contrasta con la representación gráfica de una función exponencial (ver Figura 3). Solamente con el tamaño del kakuro no se puede predecir el comportamiento que el algoritmo tendrá ni el tiempo de ejecución. Un cálculo basado solamente en el tamaño del puzle difiere de la proyección teórica sobre el comportamiento del algoritmo. En palabras sencillas, la dificultad de un kakuro no se ve directamente afectada por el tamaño del mismo.

IV-D. Discusión

Sobre hilos

Es interesante notar que el uso de hilos, en los experimentos realizados, no mejoró el tiempo de ejecución del programa. Es posible que lo anterior se deba a la implementación misma de los hilos. Existen diversas manera de implementar y manejar los hilos en un programa. En este proyecto, se utilizaron hilos ”sencillos”, cuya creación se realiza en una línea de código, mediante el uso de la librería **threading** de Python. Los hilos son creados en un ciclo y, después de eso, su ejecución es, de cierto modo, libre.

Existen otros métodos disponibles para implementar hilos. Uno de ellos consiste en utilizar una piscina de hilos (del inglés *thread pool*), la cual crea una estructura en la que los hilos (también llamados *workers*) esperan recibir una orden de ejecución. De esta piscina salen los hilos que se ejecutan en el programa. El objetivo de la piscina es mejorar la concurrencia de procesos. Es posible que una implementación de este tipo genere mejores resultados en la resolución de los kakuros. Lo anterior, sin embargo, se escapa del alcance de este proyecto y queda para futuras revisiones y versiones del programa el utilizar hilos de esa manera.

REFERENCIAS

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