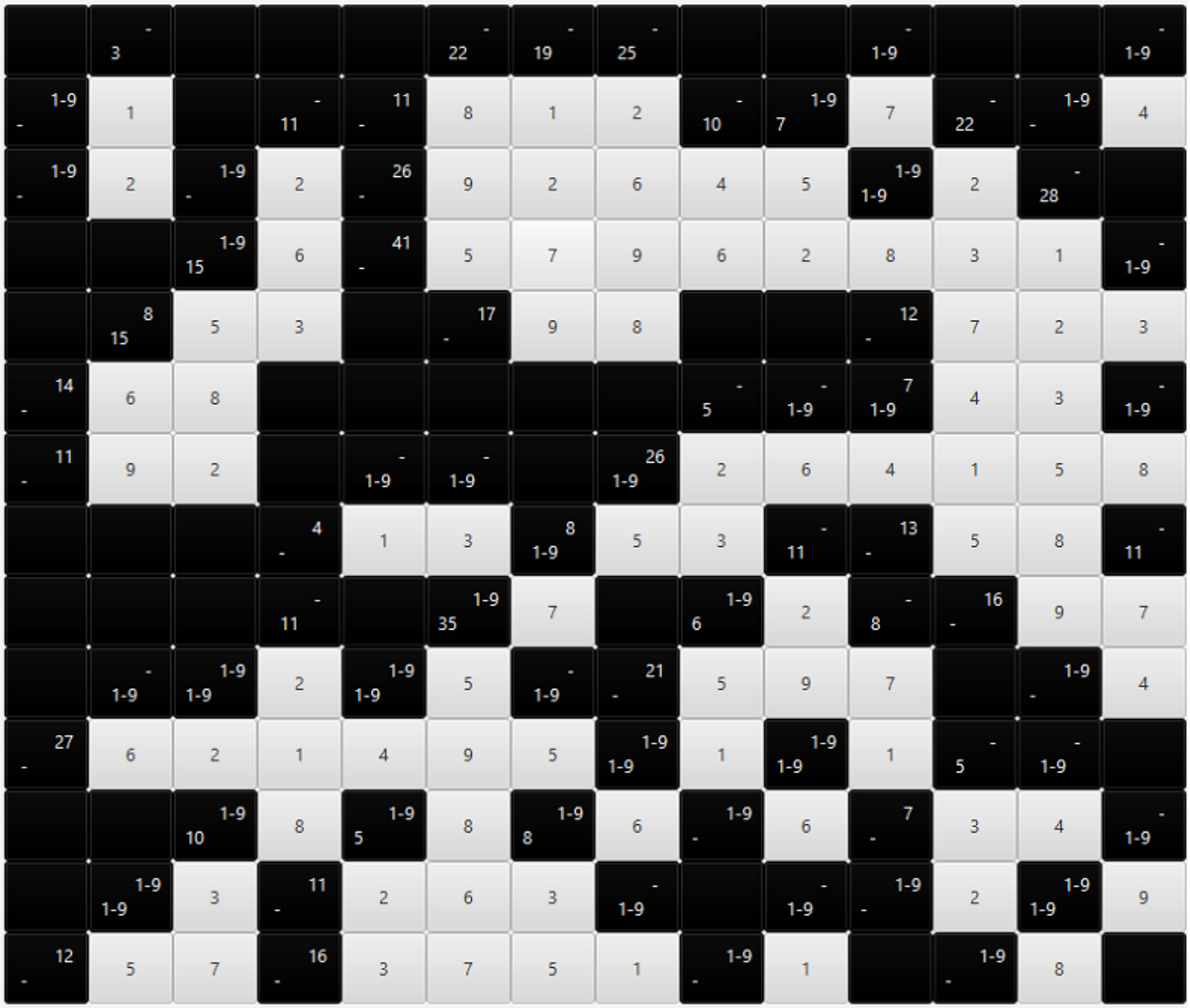
Proyecto BackTracking
Análisis de Algoritmos

Javier Contreras Muñoz
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
Email: javier.contrerasm16@gmail.com

Bryan Mena Villalobos
Instituto Tecnológico de Costa Rica
Heredia, Costa Rica
Email: mena97villalobos@gmail.com

Kakuro



***Abstract*—El algoritmo de backtracking se puede definir como la estrategia utilizada para encontrar soluciones a problemas con restricciones. En dicho algoritmo existen distintas soluciones las cuales pueden ser soluciones parciales. Dichas soluciones se recorren y en caso de no cumplir cierta condición se procede a retroceder en la búsqueda para escoger nuevos caminos. El proyecto consistirá en la generación de un tablero de Kakuro mediante el uso de backtracking y de igual manera resolverlo utilizando el algoritmo de backtracking. Se podrá dar la opción de escoger si resolverlo con hilos, con forks o de forma secuencial. Asimismo se dará una breve explicación sobre cada uno de los distintos algoritmos utilizados en el proyecto así como el análisis en O Grande de las funciones de Permutación, Poda y BackTracking. Como punto final se realizarán distintos experimentos usando hilos, forks y ninguno con el fin de poder determinar ventajas y caracteristicas de usar cada uno de ellos. A continuación se adjunta una imagen del resultado final del proyecto en la parte superior de este abstract.**

I. Introducción

El algoritmo de backtracking se puede definir como la estrategia utilizada para encontrar soluciones a problemas con restricciones. En dicho algoritmo existen distintas soluciones las cuales pueden ser soluciones parciales. Dichas soluciones se recorren y en caso de no cumplir cierta condición se procede a retroceder en la búsqueda para escoger nuevos caminos los cuales siguen siendo posibles soluciones. En caso de recorrrer todas las posibles soluciones y no encontrar una solución satisfactoria se dice que el problema no tiene solución.

La técnica de backtracking va creando todas las posibles combinaciones de elementos para obtener una solución. Su principal virtud es que en la mayoría de las implementaciones se puede evitar combinaciones, estableciendo funciones de acotación (o poda) reduciendo el tiempo de ejecución.

La idea del backtracking está acuñada por la recursion ya que se trabaja con un árbol implícito, no necesariamente con un árbol cargado a memoria ya que hacer dicho proceso puede implicar un costo bastante alto. En dicho árbol cada nodo corresponde con una solución parcial y por lo general se utilizan distintas heurísticas con el fin de no recorrer todo el árbol y evitar recorrer nodos los cuales no conducirán a ninguna solución.

En el siguiente documento se hablará en detalle acerca de los distintos algoritmos utilizados para la resolución del Kakuro así como los algoritmos de permutaciones, poda del árbol, generación del tablero y el uso de los hilos y los forks.

II. Algoritmos

A. BackTracking Generación del Kakuro

El algoritmo que genera el kakuro se subdivide en distintos algoritmos. El primero de ellos utiliza la aleatoriedad para poder colocar los cuadros negros dentro del tablero y a la vez verifica que un cuadro negro hacia la derecha o hacia abajo posea más de 9 casillas.

El proceso de establecer los números claves varía dentro del tablero. El algoritmo que realiza esto primero verifica que columnas o que filas se encuentras solitarias, es decir rodeadas por negros. Una vez establecido dicho filtro se procede a buscar el rango de valores posibles([min,max]) dependiendo del número de casillas blancas disponibles. Luego de establecer dichas claves se procede a verificar el resto de casillas negras las cuales dependen unas de otras. Para esto se comienza seleccionando todas las columnas

B. Backtracking Resolución del Kakuro

En el algoritmo de resolución del Kakuro se comienza resolviendo las columnas y filas que se encuentren solas, es decir, rodeadas por negros. Se utiliza una de las multiples permutaciones generadas respecto a la clave colocada y se completan las filas y columnas correspondientes. Posteriormente para el resto del tablero se van evaluando permutaciones posibles y válidas y mediante la solución de poda(la cual se explicará posteriormente) se verifica si una de esas posibles soluciones es una solución prometedora. En caso de que no sea prometedora se verificará otro camino mediante otra permutación prometedora. La resolución del Kakuro se realiza mediante bloques o islas en 3 distintas modalidades: secuencial, hilos de ejecución y forks. En las últimas 2 se creó un hilo de ejecución y un fork por cada bloque generado.

C. Permutaciones

El algoritmo de permutaciones está implementado en forma recursiva de manera que no genere repetidos. El algoritmo recibe como parámetros una lista con los valores a trabajar para realizar las permutaciones, un String el cual servirá como base para ir formando la permutación, un n el cual indicará el tamaño de la permutación, un r el cual es la cantidad de elementos a usar, un arreglo de Strings para ir guardando las permutaciones generadas y la clave. La clave o valor, es esencial ya que dicho número permitirá filtrar las permutaciones no deseadas con el fin de usar solamente las necesarias. El algoritmo itera sobre cada uno elementos a usar y se llama recursivamente con cada uno de esos elementos para poder ir generando permutaciones simultáneamente. En caso de que se intente colocar un número repetido en la permutación el algoritmo lo omitirá.

D. Poda

Podemos dividir la poda en dos secciones, esto debido a que la manera de generar el Kakuro es diferente a la de resolverlo; para crearlo combinamos la aleatoriedad con el backtraking, primeramente configuramos la clave para las filas o columnas solas (No dependen de otras casillas, es por esto que podemos seleccionar cualquier clave que se pueda escribir en el número de casillas disponibles) luego hacemos un ”sort”, dividimos nuestro tablero en filas y columnas, como antes se mencionó, configuramos la clave de las columnas y luego proseguimos buscando una combinación para las filas en donde calcen, es aqui donde viene el backtraking dado que cortamos parte de las permutaciones posibles en base a las casillas ya configuradas. Para solucionarlo utilizamos dos discriminantes, primeramente

establecemos las permutaciones de cada uno de los botones, luego procedemos a descartar permutaciones con base en otras permutaciones donde existen intersecciones, luego, mediante una función para validar el tablero, establecemos si vale la pena seguir por el camino en el que vamos o es necesario devolernos, más adelante mostraremos un análisis de nuestra función de poda.

III. Hilos y Forks

En el mundo de la computación los hilos o mejor conocidos como threads, son definidos como una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo. Dentro de un mismo proceso pueden existir distintos hilos de ejecución los cuales tienen como objetivo ejecutar distintas tareas a la vez con el fin de poder aumentar la eficiencia del procesador. Dichos hilos se pueden comunicar entre ellos. De igual manera los hilos que comparten recursos son conocidos en conjunto como un proceso.

Los forks o procesos se pueden definir como una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados.

Los procesos son independientes uno de los otros por lo cual se les asigna sectores de memoria diferentes. Estos solo pueden comunicarse a traves de mecanismos de comunicación dados por el sistema. Los hilos tienen mayor facilidad de comunicación ya que existen dentro de un mismo proceso por lo cual comparten mismas direcciones de memoria y datos.

A. Hilos en el Kakuro

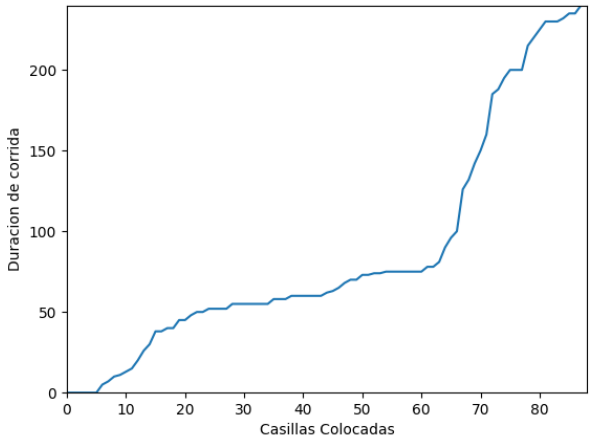
En la imagen presentada anteriormente se puede observar que el Kakuro tiende a dividirse en distintos slots. Con el fin de poder agilizar el proceso de resolución se decidió crear por cada uno de esos slots un hilo de ejecución con la finalidad de que el Kakuro se pueda ir resolviendo por bloques.

B. Forks en el Kakuro

C. Experimentos

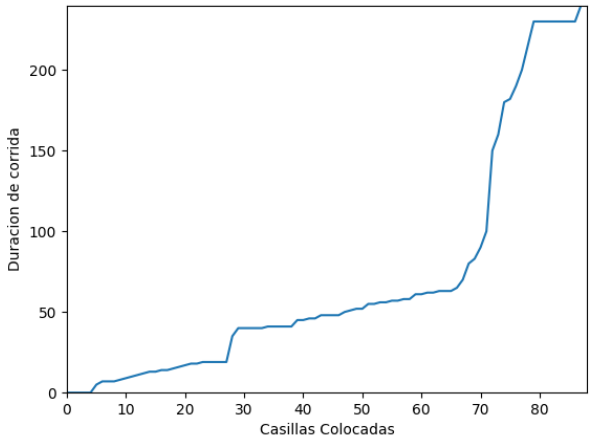
En la parte de experimentos, realizamos varios experimentos con diferentes tipos de kakuro, para empezar notamos un factor común, los hilos duran menos resolviendo que los forks, pensamos que eso es debido a que la cantidad de hilos es mayor a la cantidad de forks y según investigamos el join de los forks es un proceso más complejo al join de los hilos, seguidamente notamos que el tamaño de la ”isla” que resolvemos influye mucho en la duración de nuestros algoritmos, esto debido a la gran cantidad de intersecciones que existen y la cantidad de permutaciones posibles aún despues de pasar por nuestra función de poda, además hicimos pruebas haciendo mayor el número de hilos que utilizamos para resolver el kakuro, notamos si nos pasamos de la cantidad de islas existentes la eficiencia decrece esto debido a que cada hilo se debe comunicar con los otros a diferencia de nuestros hilos por isla que solo se deben preocupar por ellos mismo. Seguidamente cambiamos la cantidad de casillas negras existentes (esto con el fin de reducir el tamaño de las isla) el resultado fue un tiempo menor en la resolución del kakuro esto debido a lo que antes mencionabamos de la cantidad de intersecciones en la isla. En cuanto a nuestra función de solución secuencial

Forks.png



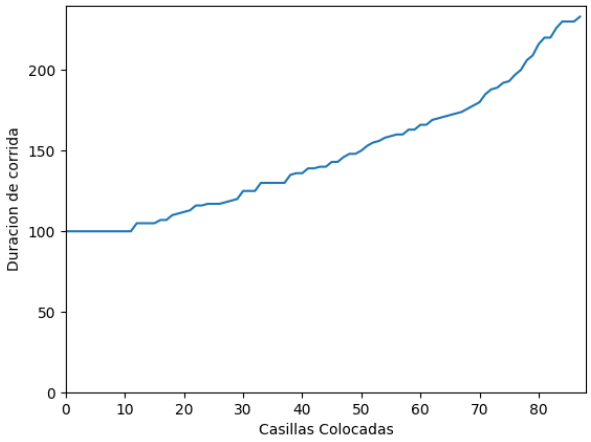
Cabe destacar que solo nos fue posible utilizar un fork para generar el gráfico anterior, despues de realizar varios intentos nuestro computador no podía ejecutar más de dos forks, es por esto que tuvimos que recurrir a un solo fork por isla. Además el grafico esta basado en varias corridadas con diversos kakuros

Hilos.png



Como se aprecia en el gráfico hay momentos en los cuales el crecimiento es ”lineal” mientras que en otros momentos el crecimiento se dispara esto se debe, posiblemente, a una isla de tamaño considerable. De igual manera este grafico no se adaptará a la totalidad de kakuros debido a que son muy variantes

Secuencial.png



A diferencia de las graficas anteriores, nuestra función secuencial necesita ir isla por isla, esto se ve reflejado en un aumento exponencial en el tiempo que necesita para resolver un kakuro

IV. Análisis O Grande Permutaciones

```
if (n == 0)[
    int suma=0;
    String[] array = restricciones.split(",");
    for(int i=0; i<array.length; i++)
        suma+=Integer.parseInt(array[i]);
 if(suma==valor)
    permutaciones.add(restricciones);
  ]
else
    for (int i = 0; i ¡ r; i++)
        if (!restricciones.contains(elem[i]))
            Permutaciones(elem, restricciones +
elem[i] +"," , n - 1, r,permutaciones, valor);
```

Consideramos que esta fucones es de orden $O(r^n)$ donde n es el tamaño de la permutación requerida y r es la cantidad de elementos disponibles para realizar la permutación, descartamos el costo de la recursión debido a que es menor a r^n, el orden lo tomamos así debido a la recursion que se hace para ooptener todas las permutaciones

V. Análisis O Grande BackTracking

Para el backtraking consideramos la siguiente función:

```
for (int[] coor : isla.negros) { int tam = 0;
tam += controlador.verificarNoRepetidosFila(coor[0], coor[1], 1).size();
tam += controlador.verificarNoRepetidosFila(coor[0], coor[1], 2).size();
if (tam != 0) {return false; }
```

}consideramos esta funcion como $O(m * n)$ ya que el for se ejecuta m veces y al sacar la complejidad algoritmica de verificar no repetidos fila pudimos concluir que es de orden n ya que estamos recorriendo un vector

VI. Análisis O Grande Poda

Seleccionamos el siguiente codigo para el analisis de la poda debido a que es el que más se utiliza, como una breve descripción del codigo tenemos una clase que contiene un botón las permutaciones asociadas a él y las intersecciones con otros botones, recorremos este arreglo y a la vez buscamos las intersecciones que posee, cada intersección contiene una lista de permutaciones, estas permutaciones son seleccionadas con base en la permutación del boton original, aqui esta nuestra función de poda, eliminamos una gran cantidad de permutaciones y con esto evitamos recorrer todo el arbol de soluciones. A continuacion presentamos la función:

```
for (PermutacionPorBoton porBoton : permutacionPorBoton){
    for (int[] intersecciones : porBoton.intersecciones){
        Button boton = (Button)
controlador.buscarNodo(intersecciones[0], intersecciones[1]);
        PermutacionPorBoton x = null;
for(PermutacionPorBoton permutacionPorBoton1:permutacionPorBoton){
if(permutacionPorBoton1.boton.equals(boton) && !porBoton.clave){
            x = permutacionPorBoton1;
                break;
                    }
                }
    if(x!=null)
        porBoton.compararPerm(x.permutaciones);
            }
        }
```

Elegimos esta función debido a que es la que más se ejecuta en nuestro codigo de poda, consideramos que esta función es de orden $O(n^3)$ debido a los tres for's anidados, que siempre se ejecutan

References

[1] Wikipedia. (1 de Mayo de 2017). Wikipedia. Obtenido de https://es.wikipedia.org/wiki/Vuelta-atr

[2] EMEZETA. (1 de Mayo de 2017). EMEZETA. Obtenido de https://www.emezeta.com/articulos/backtracking-volviendo-atras

[3] SlideShare. (1 de Mayo de 2017). SlideShare. Obtenido de https://es.slideshare.net/victorrgonzalez357/algoritmo-de-backtracking

[4] Lineas, P. (1 de Mayo de 2017). Puras Lineas. Obtenido de http://puraslineas.com/2011/01/19/combinaciones-permutaciones-y-agrupaciones-en-java/

[5] JAVA. (1 de Mayo de 2017). Oracle Docs. Obtenido de https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html