Proyecto Análisis de Algoritmos

Jose Fernando Zuluaga[1] Nicolas Daniel Vargas[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{zuluaga_jose, vargaso-ndaniel}@javeriana.edu.co

1. Introduction

Este documento tiene el propósito de documentar, explicar y dar detalles del funcionamiento de la implementación elaborada al proyecto, entrega final, en el curso de Análisis de Algoritmos 2022-03. Este proyecto se basa en el desarrollo de un algoritmo que sea capas de resolver el juego conocido como "FlowFree", el juego presenta acertijos numéricos, cada rompecabezas tiene una cuadrícula de cuadrados con pares de puntos de colores que ocupan algunos de los cuadrados. El objetivo es conectar puntos del mismo color dibujando 'tuberías' entre ellos de modo que toda la cuadrícula esté ocupada por tuberías. Sin embargo, las tuberías no pueden cruzarse. La dificultad está determinada principalmente por el tamaño de la cuadrícula.

2. Detalles Generales

La implementación del juego fue desarrollada en el lenguaje C++ y un entorno Linux, distribución Ubuntu. Cuenta con una programación orientada a objetos. Se utiliza un fichero, archivo de texto, donde se encuentra en formato de texto la información de doce circuitos, mapas, rompecabezas, correspondientes a un nivel del juego FlowFree, cada linea representa un juego. El main en el código se encuentra en el archivo "flowFree.cpp", en el cual se incluye diferentes librerías, objetos y TADs.

3. Análisis del problema

En el presente problema se nos pide que dada una matriz M de tamaño N x N, que representa el tablero de juego de flowfree, se encuentre una solución de forma que todos los colores estén conectados por medio de tuberías con su respectivo destino, que todo el tablero quede lleno de dichas tuberías y que ninguna de estas se cruce o corte a otra. Para el presente problema se supone que todos las matrices ingresadas tienen al menos una solución y que habrá máximo 9 colores por tablero. Dado que la solución final implica decir si la solución presentada es aceptable o no, es decir, si cumple con las reglas del juego, entonces se puede decir que este problema es del tipo de decisión.Por ultimo para la clase diremos que este es un problema de clase np

4. Diseño del problema

4.1. Entradas

Así, el problema de dar una solución acertada a un tablero de tamaño N x N, se define a partir de:

- $i, j \in M \bigwedge 0 \leq i, j \leq N$

- $M = \begin{pmatrix} \mathrm{A}_{1,1}, A_{1,2}, ..., A_{1,N} \\ \mathrm{A}_{2,1}, A_{2,2}, ..., A_{2,N} \\ . \\ . \\ . \\ \mathrm{A}_{N,1}, A_{N,2}, ..., A_{N,N} \end{pmatrix}$, donde $A_{i,j} = \{N, A, V, B, M, P, O, R, T, ''\ ''\} \bigwedge |M| = NXN$
- $visitados = \begin{pmatrix} \mathrm{B}_{1,1}, B_{1,2}, ..., B_{1,N} \\ \mathrm{B}_{2,1}, B_{2,2}, ..., B_{2,N} \\ . \\ . \\ . \\ \mathrm{B}_{N,1}, B_{N,2}, ..., B_{N,N} \end{pmatrix}$, donde $B_{i,j} = \{true, false\} \bigwedge |visitados| = |M|$
- c, donde $c = \{N, A, V, B, M, P, O, R, T, ''\ ''\}$

4.2. Salidas

Con esto en mente y con el análisis realizado previamente entonces podemos decir que nuestra solución nos ofrecerá de salida:

$$M' = \begin{pmatrix} \mathrm{A}_{1,1}, A_{1,2}, ..., A_{1,N} \\ \mathrm{A}_{2,1}, A_{2,2}, ..., A_{2,N} \\ . \\ . \\ . \\ \mathrm{A}_{N,1}, A_{N,2}, ..., A_{N,N} \end{pmatrix}, \text{ donde } A_{i,j} = \{N, A, V, B, M, P, O, R, T\} \bigwedge |M| = NXN$$

5. Descripción del algoritmo

El problema al que nos enfrentamos tiene múltiples soluciones. Para llegar a la solución desarrollada, se realizo un correspondiente análisis al problema donde se llego a a algunas de estas múltiples soluciones. Inicialmente se contemplo la idea de una solución a partir de la fuerza bruta, donde se exploraría en cada casilla, todos los colores, esto nos llevaría a un algoritmo donde encontraría la solución acertada, pero tendría una complejidad elevada y como consecuencia un tiempo de ejecución muy grande, por esta razón seguimos en la búsqueda de otra solución lo cual nos conlleva a la segunda solución la cual se basa en encontrar todos los posibles caminos entre los puntos correspondientes al color, y a partir de esos caminos, buscar la combinación de caminos que satisfagan con las condiciones para la solución adecuada, esta opción es óptima, sin embargo, también presenta una naturaleza de fuerza bruta pero aplicada sobre los caminos, esto se podría implementar mediante el uso de grafos, sin embargo, esto implica una manipulación extra sobre nuestros datos e información para adaptarlas a la solución. Y como consecuencia, llegamos a nuestra tercera solución la cual fue la que se decidió emplear, y se basa en un algoritmo el cual verifique, y explore caminos, durante su marcha, es decir, un algoritmo recursivo que pueda validar el próximo movimiento y su dirección de dicho color, si no es posible hacía dicha dirección, intentar otra, en caso de que no haya una ruta posible, en las cuatro direcciones posibles, retroceder en el camino hasta cierto punto, para esto se usa una matriz auxiliar de booleanos para realizar backtracking sobre sus pasos.

5.1. Pseudocodigo

Para la solución planteada se utilizaron tres nuevas funciones, una de ellas es verificar la cual no se va a estipular acá debido a que es demasiado corta y su lógica se basa en verificar que los valores estén dentro de los margenes aceptados y que no se haya pasado por pedazo previamente, Las otras dos funciones son solucionar y obtenerColor, obtener color busca siguiente color mas cercano por el que no se halla pasado previamente y retorna su ubicación, solucionar es la función principal donde, como mencionamos previamente, el color avanza por las casillas hasta chocar o no avanzar mas y cuando esto ocurre en caso de llegar al final

cambia de color, de lo contrario se devuelve sobre sus propios pasos. Ver *algoritmo*1 *solucionar* y *algoritmo*2 *obtenerColor*.

Algorithm 1 solucionar

Require: M : Matriz cuadrada que representa el mapa del juego y contiene caracteres alfabéticos
Require: i : Valor que representa una posición en la matriz M, este representa la fila.
Require: j : Valor que representa una posición en la matriz M, este representa la columna.
Require: *visitados* : Matriz del mismo tamaño de M, esta contiene valores booleanos que representan si esa posición ya se ocupo.
Require: *color* : color que se esta buscando actualmente.
Ensure: La matriz M' que representa la matriz M solucionada

```
1: procedure SOLUCIONAR(i, j, visitado, color, matriz)
2:     visitado_{i,j} ← true
3:     for p ← 0 to 4 withjumpsof1 do
4:         i1 ← i + dx[p]
5:         j1 ← j + dy[p]
6:         if valid(i1, j1, game.tam, visitado) then
7:             if matriz_{i1,j1} == color then
8:                 visitado_{i1,j1} = true
9:                 pair < int, int > p = obtenerColor(game.tam, visitado)
10:                if p.fist == −1 and p.second == −1 then
11:                    return true
12:                end if
13:                c ← matriz_{p.first,p.second}
14:                if solucionar(p.first, p.second, visitado, c) then
15:                    return true
16:                end if
17:                visitado_{i1,j1} ← false
18:            end if
19:            if matriz_{i1}j1 == 32 then
20:                matriz_{i1}j1 = color
21:                if solucionar(i1, j1, visitados, color) then
22:                    return true
23:                end if
24:            end if
25:        end if
26:    end for
27:    visitado_{i,j} = false
28:    return false
29: end procedure
30:
```

6. Complejidad

Para este algoritmo se tiene una complejidad de:

$$O(P^{n*n} - P)$$

Donde P representa el numero de colores en el mapa de juego seleccionado y
n representa el tamaño del tablero, siendo el numero total de casillas, contando desde 0. Ya que en el peor de los casos, por cada color en su exploración de posibles caminos tenga que pasar por todas las casillas vacías, esto implica casillas no marcadas, diferentes a vació.

Algorithm 2 obtenerColor

Require: Tam : Tamaño del nivel (tamaño de las filas, no de todo el tablero).
Require: $visitado$: Matriz del mismo tamaño de M, esta contiene valores booleanos que representan si esa posición ya se ocupo.
Ensure: posición fila - columna en la matriz donde se encuentra el nuevo color

```
1: procedure OBTENERCOLOR(tam, visitado)
2:     pair < int, int > p
3:     p.fist ← −1 ∧ p.second ← −1
4:     encontro ← false
5:     i ← 0
6:     while!encontradoandi < game.tamthen
7:     j ← 0
8:     while!encontradoandj < game.tamthen
9:     if matriz_{i,j}! = 32and!visitado_{i,j} then
10:        p.first ← i
11:        p.first ← j
12:        encontrado ← true
13:    end if
14:    j+ = 1
15:    EndWhile
16:    i+ = 1
17:    EndWhile
18:    return p
19: end procedure
20:
```

7. Compilación y ejecución del programa

Para la correcta compilación y ejecución del código se requiere un sistema en base Linux de 64bits. Una vez se cuente con este requerimiento basta con ubicarse en la carpeta donde se encuentren los archivos, este va a variar dependiendo de donde se hayan guardado el juego pero en general se debe ver de la forma .../ProyectoAA. Una vez en la carpeta se debe abrir un símbolo del sistema o una terminal, en este se debe ejecutar el siguiente código

g++ -o main flowFree.cpp

Una vez hecho esto, para correr el programa basta con escribir en esa misma terminal o símbolo del sistema el comando

./main

y listo, en la terminal saldrá el juego sin mas requisitos

8. Prueba

Al ejecutar el programa nos preguntara el único dato que requiere, y es definir el tamaño del tablero (6, 7, 8 y 9). Ya ingresado el tamaño el programa elegirá un mapa aleatorio de los cargados en memoria, posteriormente a esto, mostrar el mapa seleccionado en su estado inicial, junto con el mismo mapa en su estado final, ya solucionado, si es posible. Ver imagen *Prueba con mapa* $8x8$, *Prueba con mapa* $9x9$, etc. Viendo los diferentes resultados y su medida en el tiempo de ejecución empleado por cada uno de los casos, se identifica un aumento exponencial del tiempo de ejecución, lo cual es lógico, ya que es directamente proporcional al números de colores en el mapa seleccionado, como consecuencia de elegir un tablero, mapa de mayor tamaño, involucra tener un mayor numero de colores, y esto es lo que afecta en el aumento del tiempo de ejecución del algoritmo, teniendo como resultado en casos con tableros de tamaño 9x9 un tiempo promedio de cuatro minutos para ser solucionado.

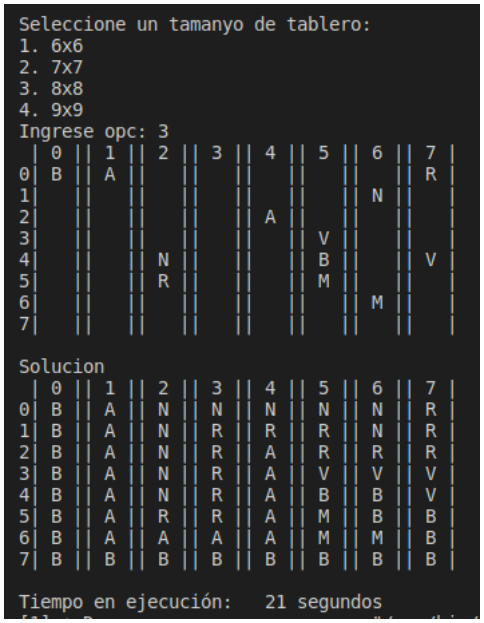

Figura 1: Prueba con mapa 8x8

9. Conclusiones

Se concluye con la elaboración de este proyecto, la relación entre el tiempo de ejecución y la complejidad del algoritmo, donde es importante determinar la herramienta adecuada teniendo en cuenta los datos e información de entrada. La solución planteada que desarrolla el problema se basa es fuerza bruta, utilizando un "backtracking", funcional cuando no se cuenta con un gran numero de datos, en este caso, numero de colores.

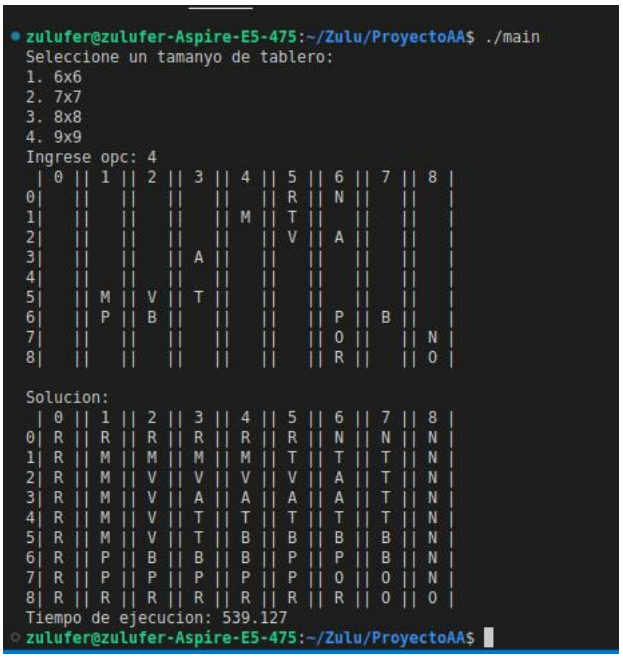

Figura 2: Prueba con mapa 9x9

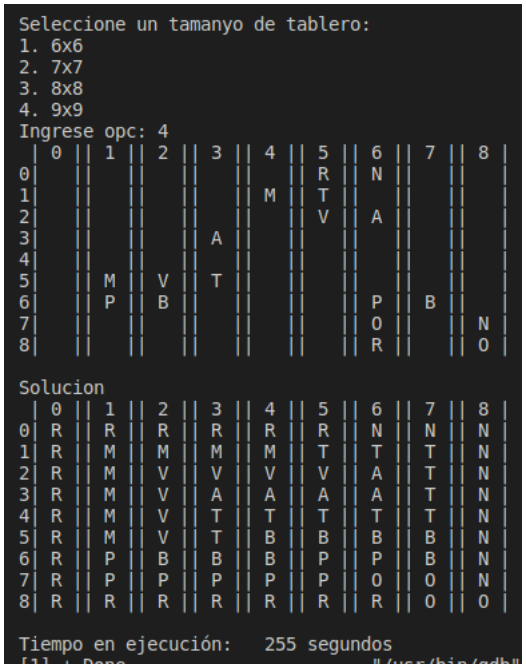

Figura 3: Prueba 2 con mapa 9x9