Universidad de Antioquia

Facultad de Ingeniería

Programas educativos:

Ingeniería de Telecomunicaciones
Ingeniería Electrónica

Docentes:

ANÍBAL GUERRA

AUGUSTO SALAZAR

Parcial 1: Informática II

Desafío 1

Informe para Desafío 1

Estudiantes:

ANTONIO CARLOS MERLANO RICARDO (1010152199)

JUAN ESTEBAN LOPEZ CASTRILLON (1127941113)

08 DE ABRIL DEL 2024

Semestre 2024 – 1

Análisis del Sistema de Cerraduras X de Informa2

El sistema de cerraduras X desarrollado por Informa2 es una solución de seguridad que se basa en una estructura de datos especial, denominada M, capaz de rotar y cambiar de dimensiones. Esta estructura se caracteriza por ser cuadrada y tener un número impar de filas, lo que facilita su alineación central.

Estructura de datos M y Cerradura X:

La cerradura X se compone de varias estructuras M alineadas, utilizando la celda central como referencia. No hay restricciones en cuanto a la cantidad de estructuras M que se pueden alinear ni en cuanto a sus tamaños, lo que permite una flexibilidad en el diseño de las cerraduras. Por ejemplo, una cerradura X (5, 7, 5, 9) podría estar compuesta por cuatro estructuras de tamaños 5x5, 7x7, 5x5 y 9x9 respectivamente.

Regla de Validación K:

Para abrir una cerradura X, se debe validar la rotación de las estructuras M alineadas mediante una regla K específica. Esta regla K consiste en un conjunto de condiciones que deben cumplirse para que la cerradura se abra correctamente. Por ejemplo, la regla K (4,3,1,-1,1) indica que, para la celda ubicada en la fila 4, columna 3 de la primera estructura, su valor debe ser mayor al de la celda correspondiente en la siguiente estructura, y así sucesivamente.

Proceso de Apertura:

El proceso de apertura de la cerradura implica rotar cada una de las estructuras M de forma independiente, alineando las celdas de manera que se cumplan las condiciones establecidas por la regla K. Para ello, es necesario diseñar algoritmos capaces de crear y rotar matrices bidimensionales de manera eficiente, teniendo en cuenta la relación entre filas y columnas durante la rotación.

Pasos para Resolver el Problema:

1. Diseñar un algoritmo para crear arreglos bidimensionales con tamaño variable, utilizando arreglos dinámicos para evitar desbordamientos de memoria.

2. Implementar un algoritmo para rotar las estructuras M, teniendo en cuenta que, al rotar 90 grados, las filas se convierten en columnas y las columnas se invierten en orden.

3. Desarrollar un algoritmo para generar la cerradura X, que pregunte al usuario la cantidad y tamaño de las estructuras M que compondrán la cerradura, y cree las estructuras de manera adecuada.

4. Crear un algoritmo de validación que verifique si se cumplen las condiciones de la regla K para cada celda de la cerradura X, rotando las estructuras M según sea necesario hasta que la cerradura se abra.

NOTA: CONSIDERACIONES PARA LA ALTERNATIVA DE SOLUCIÓN PROPUESTA

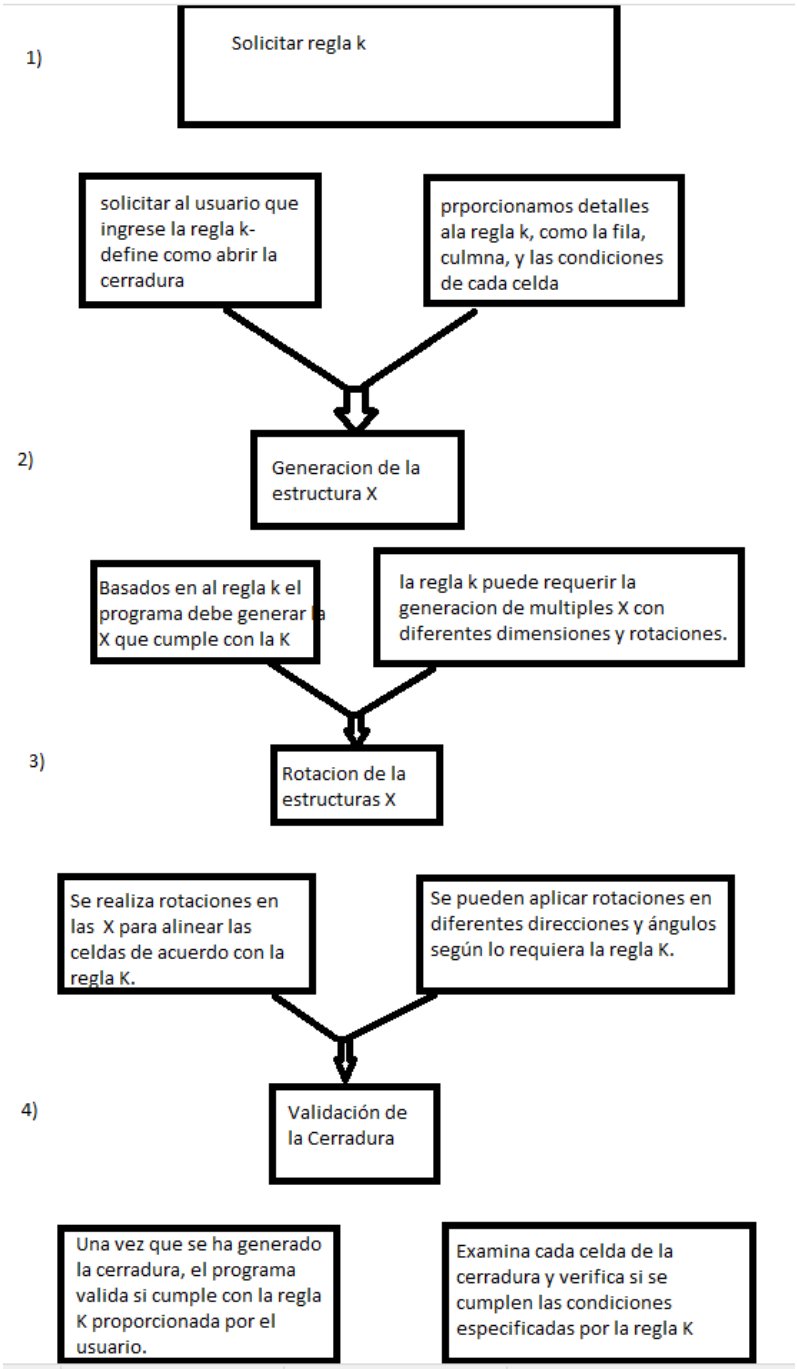
Es muy importante asegurar la eficiencia del algoritmo tanto en tiempo como en memoria, especialmente al diseñar las rutinas de generación y rotación de estructuras M. Esto implica implementar estrategias de afrontamiento óptimas, optimizadas para matrices de diferentes tamaños, evitando sobrecargas innecesarias. De manera similar, la regla K debemos verificar la para garantizar que las condiciones especificadas se cumplan adecuadamente en todas las celdas de la estructura M y la cerradura X.

Es fundamental realizar pruebas para verificar la funcionalidad y confiabilidad del sistema, probar diferentes configuraciones de la cerradura X e incorporar casos extremos para garantizar que el sistema funcione.

Esquema implementado para el desarrollo del problema

Para implementar la serie de pasos que utilizamos para el desarrollo del desafío primeramente fue dividir el programa por etapas, es decir:

Para cada etapa del desarrollo, utilizamos guiados por la guía donde una vez implementados los pasos ya analizados, para esto realizamos una especie de esquema donde este nos permita la comprensión del desarrollo en el programa:



Algoritmos implementados

Generación de Estructura X:

Después de haber encontrado una configuración válida para la cerradura X, esta función se encarga de generar y llenar cada matriz en cerradura X de acuerdo con esta configuración. Utiliza la función llenarEstructura() para llenar cada matriz individualmente**.**

```
void generadorDeEstructuraX()
{

    //sacamos la longitud del arregloK

    int tamaño = tamañoCerradura - 1;
    int *condiciones = new int[tamaño];
    int *configuracion = new int[tamaño];
    //metemos en condiciones los valores de reglak desde el 2 hasta tamañoCerradura+1
    for(int i = 0; i <= tamaño ; i++){
        condiciones[i] = reglaK[i+2];
    }

    encontrar_valores(tamañoCerradura,condiciones,configuracion);
    //aqui ya podemos eliminar la memoria de condiciones
    delete [] condiciones;
    //ahora que ya sabemos una posible configuracion para la cerradura X simplemente reservamos la memoria nece
    for(int j = 0 ; j<tamañoCerradura; j++){
        cerraduraX[j] = new int*[configuracion[j]];

        for(int k = 0 ; k <configuracion[j] ; k++){
            cerraduraX[j][k] = new int[configuracion[j]];
        }
    }

    //ahora llenamos cada matriz de la cerradura
    llenarCerraduraX(configuracion);
    //liberamos la memoria del arreglo configuracion
    delete[] configuracion;

}
```

Encontrar Valores que Cumplen Condiciones:

Este algoritmo encuentra valores que cumplen con ciertas condiciones definidas en la regla K. Utiliza un enfoque de para generar combinaciones de valores que satisfacen las condiciones.

```
// Función para encontrar los valores que cumplen con las condiciones dadas
void encontrar_valores(int num_bucles, int condiciones[],int *configuracion) {
    int* valores = new int[num_bucles]; // Arreglo dinámico para almacenar los valores
    for (int i = 0; i < num_bucles; ++i) {
        valores[i] = 3; // Inicializa todos los valores en 3 que es la minima que podemos hacer
    }

    while (true) {
        // Verifica si las condiciones se cumplen
        bool condiciones_cumplidas = true;
        for (int i = 0; i < num_bucles - 1; ++i) {
            int condicion = condiciones[i];
            if (condicion == 1 && !(valores[i] > valores[i + 1])) {
                condiciones_cumplidas = false;
                break;
            } else if (condicion == -1 && !(valores[i] < valores[i + 1])) {
                condiciones_cumplidas = false;
                break;
            } else if (condicion == 0 && !(valores[i] == valores[i + 1])) {
                condiciones_cumplidas = false;
                break;
            }
        }

        if (condiciones_cumplidas) {
            // Imprime los valores que cumplen las condiciones
            cout << "Valores que cumplen las condiciones: ";
            for (int i = 0; i < num_bucles; ++i) {
                configuracion[i] = valores[i];
                cout << valores[i];
                if (i != num_bucles - 1) {
                    cout << ", ";
                }
            }
            cout << endl;
            delete[] valores; // Libera la memoria del arreglo dinámico
            return; // Finaliza la ejecución después de encontrar una solución
        }

        // Incrementa los valores en el arreglo
        for (int i = 0; i < num_bucles; ++i) {
            valores[i] += 2;
            if (valores[i] >= 10) {
                valores[i] = 3;
            } else {
                break;
            }
        }
    }
}
```

Gestión de Memoria Dinámica

El código que tenemos gestiona la memoria dinámica adecuadamente para evitar las tan llamadas fugas de memoria. New se utiliza para crear matrices bidimensionales en cerraduraX y delete para liberarla. Aquí un ejemplo de como utilizamos el delete para liberar la memoria dinamica

```
void liberarArreglo(int** arreglo,int filas) {


    for (int i = 0; i < filas; ++i) {
        delete[] arreglo[i]; // Liberar cada fila del arreglo
    }
    delete[] arreglo; // Liberar el arreglo en sí
}
```

Función pedirclavek:

se encarga de solicitar al usuario la clave para generar una cerradura basada en esa clave. La clave se refiere a un conjunto de reglas (denominado regla K) que determinan cómo se va a generar la cerradura. La función guía al usuario para ingresar los elementos de la clave, que incluyen la fila, la columna y los valores de las celdas para cada matriz en la cerradura.

```
void pedirClaveK()
{

    int fila; //variable para ubicar la celda
    int columna; //variable para ubicar la celda
    int valor;

    cout << "------  Ingrese la clave para generar una cerradura para esa Clave  ------ \n Cual es el tamano que va a tener la cerradura : ";
    cin >> tamañoCerradura;

    // generamos el arreglo que va a contener la regla K
    //la regla K tiene una posicion mas por eso usamos la variable tamañoCerradura y le sumamos 1
    reglaK = new int[tamañoCerradura + 1];

    // aprovechamos y generamos el arrayX que va a contener las matrices
    cerraduraX = new int **[tamañoCerradura];//OJO todavia no hemos reservado la memoria para lo de adentro de las matrices , eso lo hacemos en la funcion que genera las estructuras

    //hacemos un bucle que se va a repetir el tamaño de la cerradura + 1 para pedir los valores para el arreglo de K al ser <=tamañoCerradura y i = 0 es lo mismo que tamañoCerradura + 1
    for (int i = 0; i <= tamañoCerradura; i++)
    {
        if (i == 0)
        {
            bool filaValida = false;
            int PrimeraVez = true;

            while (!filaValida)
            {
                cout << (PrimeraVez ? "Ingrese la fila : " : "La fila debe ser un numero positivo \n Ingrese nuevamente la fila :  ") << endl;
                cin >> fila;
                if (fila > 0)
                {
                    reglaK[0] = fila - 1;
                    filaValida = true;
                }
                else
                {
                    PrimeraVez = false;
                }
            }
        }

        else if (i == 1)
        {
            bool columnaValida = false;
            int PrimeraVez = true;

            while (!columnaValida)
            {
                cout << (PrimeraVez ? "Ingrese la columna : " : "La columna debe ser un numero positivo \n Ingrese nuevamente la columna : ") << endl;
                cin >> columna;
                if (columna > 0)
                {
                    reglaK[1] = columna - 1;
                    columnaValida = true;
                }
                else
                {
                    PrimeraVez = false;
                }
            }
        }

        else
        {
            bool valorValido = false;
            int PrimeraVez = true;

            while (!valorValido)
            {
                cout << (PrimeraVez ? "Ingrese el valor o condicion de la celda de la matriz " + to_string(i - 1) + " con respecto a la siguiente estructura " + to_string(i) + " (Matriz)  : " : "la condic
                cin >> valor;
                if (valor == 0 || valor == 1 || valor == -1)
                {
                    reglaK[i] = valor;
                    valorValido = true;
                }
                else
                {
                    PrimeraVez = false;
                }
            }
        }
```

Problemas en el desarrollo:

A medida que íbamos desarrollando el programa nos encontramos con varios problemas principalmente era entender y comprender el análisis ya previamente realizado para lograr plasmar la ideas en nuestro programa como por ejemplo el proceso mas complejo es realizar la validación de la regla K ya que esto representa el 60% de nuestro programa, por otro parte tuvimos varios problemas de sintaxis esto dado por algunos conceptos en memoria dinámica, la utilización de punteros pero que a medida íbamos utilizando mas el programa era más fácil de detectar por medio del uso dela depuración que nos simplifico en detectar los errores por ejemplo para el desarrollo de las matrices porque nos permite ubicarnos exactamente donde queremos implementar o modificar parte del código, por ultimo no solo es analizar y entender el desafío planteado si no como plasmar nuestras ideas para la realización del código, por otra parte toco aprender a utilizar correctamente la plataforma git hub y como trabajar en proyectos grupales y la realización y actualización de commit.

Evolución del programa

Para el desarrollo en qt dividimos el proyecto en varios archivos el main.cpp, funciones.cpp y función.h. Como se sobre entiende en el main implementaremos las funciones que desarrollamos en fuciones.cpp y en función.h están declaradas cada función

En lo que se desarrollo el programa se llevo paso a paso por la guía, a continuación, la evolución del programa por serie de pasos:

1. Definición de funciones: primero fueron estas funciones:
    - Void pedirDatos()
    - Void crearEstructuras()
    - Void llenarEsctuctura()

    Estas funciones se utilizaron para la creación de la estructura M es decir para la creación de matriz de tamaño 3x3 en adelante.

2. Implementación de las funciones para rotar dimensión de las matrices
    - void girarIzquierda
    - void liberarArreglo()

    como su nombre lo indica la utilizamos para rotar la matriz 90 grados ala izquierda en un máximo de 3 ocasiones. Y para liberararreglo() eliminar la memoria de arreglos bidimensionales

3. Empezamos a utilizar las distintas reglas para K:
    - void pedirClavek()
    - void validarReglaK()

Estas funciones nos permiten que el usuario ajuste el numero de estructuras filas y columnas para validar la regla k y proceder a utilizar esta información para generar la primera matriz de la cerradura.

4. En este paso nos enfocamos a la aplicación de la validación de la regla k por lo cual hicimos cambios e implementamos y borramos varias funciones que ya no utilizaríamos para concluir con la finalización de nuestro código, en el cual nos enfocamos en cumplir el desarrollo total del programa donde lo mas importante es que por medio de la condición el programa fuera capaz de funcionar correctamente
    - Void mostrarMatrizConLibreria()
        - Void llenarCerraduraX()
        - Void encontrar_valores()

La funcion mostrarMatrizConLibreria() nos permite obtener una función que imprime la matriz de manera más organizada.

Con la función llenarCerradura() lo que hacemos es rellenar las matrices de las cerraduras.

encontrar_valores() nos permite combinaciones de las matrices para que cumplan las condiciones ya dadas.