Universidad de Costa Rica - Escuela de Ingeniería eléctrica

IE0117 Programación bajo plataformas abiertas

Proyecto Final

Estudiante:
| Sixto Loría Villagra C04417 |

Profesor:
Juan Carlos Coto

Primer ciclo de 2022

15 de julio de 2022

Índice

1. Introducción

En este informe se encuentra información acerca de los planteamientos, procedimientos y desarrollo de algoritmos realizados para cumplir con la solución solicitada. También se encontrara información acerca de los principales retos que se obtuvieron a lo largo del desarrollo del programa.

Se seguirán una serie de pasos en donde expondré el planteamiento y razonamiento hecho a la hora de resolver el problema propuesto.

La composición del código se puede dividir en 5 partes:

- Lectura del archivo.
- Soporte para matrices de laberintos de cualquier tamaño
- Búsqueda de los puntos iniciales en los bordes.
- Evaluación de estos puntos iniciales en la función que realiza la solución.
- Función recursiva que realiza la solución.

2. Diseño general

El código se compone en varios archivos, los cuales contienen las funciones necesarias para llegar a la respuesta. Esto se hizo tal y como lo solicito el enunciado, con el propósito de que el código sea mas entendible y fácil a la hora de encontrar errores, lo cual funciono y es un método que tengo pensado seguir aplicando a los próximos proyectos que se me presenten en los cursos futuros.

El diseño del programa fue hecho en un principio en un solo archivo del cual el código se fue pasando a funciones y luego se separo en varios archivos, que contienen grupos de funciones que funcionan como pequeñas librerías las cuales son llamadas desde la función principal, la cual es la que llama a todas y las une ejecuta.

2.1. Lectura del archivo

Luego de llamar a las bibliotecas necesarias el archivo donde se encuentra la función principal lo primero que hace es llamar a la función que lee y valida el archivo.

```
// Funcion que lee y valida el archivo.
void validar_archivo(){


    FILE *laberinto;
    laberinto = fopen("laberinto.txt", "r");
    // Comprobando si hay informacion en el archivo.

    if(laberinto == NULL){
        printf ("El archivo esta vacio");
        exit(1);
    }
    fclose(laberinto);
}
```

Figura 1: Función que lee y valida el documento

Esta función como su nombre lo indica, se encarga de tomar un fichero, para abrir el archivo y verificar si se encuentra vació. Esta función también verifica si el archivo que debe llevar nombre de laberinto.txt se encuentra dentro de la misma carpeta que el código.

2.2. Soporte para matrices de laberintos de cualquier tamaño

Mi código debía de estar preparado para soportar cualquier tamaño de laberinto que el profesor pudiera introducir mediante archivo de texto, por lo que para solucionar esto, lo primero fue calcular las dimensiones de la matriz que esta siendo ingresada, para ello volví a crear otra función que leía el

archivo con ayuda de un ciclo while cuya condición es parar hasta llegar al EOF (end of document) y la función **fgetc** la cual recorre el documento carácter por carácter. Preguntando por los saltos de linea se obtienen la cantidad de filas cuyo valor inicial del contador fue 1 debido a que por lo general no se cuenta la ultima fila al no haber salto de linea. y para las columnas se uso una función que no tomaba en cuenta los ceros sino solo los dígitos (**isdigit**) para de esta forma saber cuantas columnas hay en el documento.

Una vez cuento con los valores de las dimensiones de la matriz, puedo reservar la memoria dinámica correspondiente, para posteriormente tomar los datos del archivo nuevamente y con ellos reconstruirlos en una matriz con la cual poder trabajar.

Para ello hago uso de calloc mediante una función a la cual llame Reservar_memoria, para de una vez iniciar los valores en cero y dejarlos listos para ingresar valores. El ingreso de valores lo hice con otra función llamada ingresar_valores, la cual utiliza un ciclo while para recorrer la matriz de forma similar a la función de contar las filas y columnas, dentro del ciclo utilizo la función atoi para obtener el valor numérico del archivo y de ahí los paso a la matriz.

```c
// Funcion que reserva la memoria dinamica necesaria para la asignacion de valores, necesarios para contruir la matriz.
int **reservar_memoria (int filas, int colum){

    int i;
    int **matriz;

    matriz = (int**)calloc(filas,sizeof(int*));
    if (matriz == NULL){
        printf("No se ha podido revervar memoria \n");
        exit(1);
    }
    for(i = 0; i< filas; i++){
        matriz[i] = (int*)calloc(colum,sizeof(int));
        if (matriz == NULL){
        printf("No se ha podido revervar memoria \n");
        exit(1);
        }
    }
    return matriz;
}
```

Figura 2: Función que reserva memoria dinámica para la matriz.

```c
void introduce_valores(int filas, int colum, int **matriz){
    FILE *laberinto;

    laberinto = fopen("laberinto.txt", "r");
    int i = 0,j = 0;
    char recorrido;
    recorrido = ' ';
    while (recorrido != EOF){
        recorrido = fgetc(laberinto);

        if (i >= filas ){
            break;
        }
        if (j >= colum ){
            i++;
            j = 0;
        }

        if (isdigit(recorrido) ){
                matriz[i][j] = atoi(&recorrido);
                j++;
        }
    }
    fclose(laberinto);
}
```

Figura 3: Función que asigna valores a la matriz creada.

2.3. Búsqueda de los puntos iniciales en los bordes.

Muy bien de momento en el main llevamos esta serie de instrucciones, las cuales corresponden a las funciones antes mencionadas.

```c
// ------------------------------------------------ Contruccion de la matriz -----------------------------------
    // Abriendo y validando el archivo.
    validar_archivo();

    // Contando el numero de filas y columnas que tiene el laberinto.
    contador ( &m, &n);

    // Creacion de la martiz de trabajo
    int **matriz;

    // memoria dinamica para la matriz
    matriz = reservar_memoria(m, n);

    // asignacion de valores.
    introduce_valores(m, n, matriz);


    int i,j;
    mostrar(matriz,  m,  n);


// ------------------------------------------------ Busqueda de valores en los bordes ---------------------------
```

Figura 4: Creación de la matriz para la resolución

Ahora lo que se procede es a encontrar los puntos de inicio validos que del laberinto, estos serán guardados en 2 vectores(uno para cada eje) para luego ser introducidos en el algoritmo de resolución.

Para ello cree una función que mediante ciclos for recorre los bordes de la matriz, como no se sabe el tamaño de la matriz en un inicio el tamaño de esos vectores también ocupan memoria dinámica, por lo que en el primer recorrido lo que se hace es contar la cantidad de puntos encontrados en los bordes.

```
// Funcion que cuenta el numero de entradas, retorna dicho numero.
int num_entradas(int m, int n, char **matriz){
    int i;
    int entrada = 0;

    for (i = 0; i < n ; i++){
        if ( matriz[0][i] == 1 || matriz[0][i] == 2){
                entrada ++;
        }
    }
    for (i = 0; i< m ; i++){
        if ( matriz[i][0] == 1 || matriz[i][0] == 2){
                entrada ++;
        }
    }
    for (i= 0; i < n ; i++){
        if(matriz[m-1][i] == 1 || matriz[m-1][i] == 2){
                entrada ++;
        }
    }
    for (i = 0 ; i < m; i++){
        if (matriz[i][n-1] == 1 || matriz[i][n-1] == 2){
                entrada ++;
        }
    }
    return entrada;
}
```

Figura 5: Funcion que cuenta numero de entradas en bordes

Luego de esto se crea la memoria dinámica para los vectores en el main y con la misma lógica para recorrer los bordes se usa una función que en esta ocasión lo que hace es guardar la posición en los vectores creados anteriormente. De esta forma ya cuento con la posición de cada una de las entradas para ser evaluadas en mi algoritmo de solución del laberinto.

2.4. Evaluación de estos puntos iniciales en la función que realiza la solución

Esta fue de las partes mas difíciles y tediosas de todo el diseño, se tuvo el problema que el compilador de C no podía hacer la comparación hacia atrás de los bordes, se intento solucionar de 2 formas, la primera fue dando mas condiciones, como por ejemplo que no se pasen de los extremos, sin embargo el tan solo hecho que la comparación exista ya genera un error, por lo que se tuvo que acudir a planificar

una resolución diferente para cada posicon en los extremos del borde. Esto lo soluciono siempre y cuando el laberinto no comience en uno de los bordes.

2.5. Función recursiva que realiza la solución

Para la resolución del laberinto, después de investigar vi que la mejor forma era haciéndolo de manera recursiva, el gran resto estuvo en los bordes como mencione anteriormente, Comenzar desde los bordes y que el laberinto en si no este delimitado hace que el algoritmo a crear tenga que ser mas complejo, para proveer todo esto y al no tener un punto de inicio fijo, hace tener que crear una copia de la matriz para que esta sea la que vaya probando todas las posibles soluciones.

Como se especifico que solo una solución es la valida, se creo una variable auxiliar que detiene el bucle cuando se encuentra una solución valida.

3. Principales retos

3.1. La composición del código en diversos archivos

Durante el diseño de la solución, mi código originalmente estaba hecho todo dentro de la función principal súper desordenado, al ser uno de los objetivos separarlo en otros archivos que se compartan como bibliotecas los cuales están compuestas de funciones fue todo un reto adaptar mi código de ese modo pues nunca lo había hecho y para hacerlo es necesario tener buen manejo del tema de punteros.

3.2. Adaptación del código para cualquier tamaño de matriz

En un principio fue difícil pensar y plantar el método para poder hacer que mi código se adaptara en a cualquier matriz, fue conforme el tiempo paso que me ocurrió la idea de meter las dimensiones para poder reconstruirla, usar memoria dinámica para ello también fue todo un reto pues nunca la había usado a ese nivel.

3.3. Manejo del solucionado en los bordes

El mayor reto del código estuvo en comenzar en los bordes pues hacer comparaciones desde ahí origina errores y en una función recursiva que uno tiene que estar comparando en todas las direcciones lo que hace que que ambos puntos colacionen entre si y se tenga que pasar un buen rato pensando una solución.

3.4. Numero indefinido de posibles puntos iniciales y no conocer la posición del mismo

Otro de los principales retos fue el de tener que diseñar que el código pruebe varias soluciones para los distintos y posibles puntos de entrada al laberinto hizo que tuviera que trabajar con una copia de la matriz para poder y a hacer el procedimiento en repetidas ocasiones.

4. Conclusiones:

El proyecto honestamente me corto bastante debido a mi falta de experiencia en poner en practica los temas vistos en el curso. Se llego al resultado de un solucionador que solo falla en los bordes por lo que creo que objetivo prácticamente se cumplió. Si siento que fue de gran aprovechamiento el proyecto pues en el hice uso de prácticamente todo lo aprendido respecto al apartado de programación en C.