UANL

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FCFM

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS

Universidad Autónoma de Nuevo León

Facultad De Ciencias Físico Matemáticas

Entrega Final

ESTRUCTURA DE DATOS

PRODUCTO INTEGRAL DE APRENDIZAJE

ALUMNO

JOSE LUIS CALDERON GALARZA – 2132939

GRUPO 31

PROYECTO DE RESOLUCIÓN DE LABERINTOS CON GRAFOS

INTRODUCCION

Los laberintos, con sus enrevesados senderos y desafíos lógicos, han sido una fuente constante de intriga y desafío a lo largo de la historia. Resolver un laberinto puede ser un ejercicio mental agotador, que requiere paciencia, concentración y habilidades estratégicas. En este proyecto, se propone abordar este desafío.

Imagina tener a tu disposición una herramienta que, de manera rápida y sencilla, pueda encontrar la solución para casi cualquier laberinto. Este proyecto se centrará en el desarrollo de un sistema que haga precisamente eso, utilizando el poder de la teoría de grafos y la informática. La creciente necesidad de contar con una solución automatizada y cómo este enfoque puede tener un impacto significativo en diversos campos, desde la inteligencia artificial hasta la vida cotidiana.

No solo se buscará simplificar la resolución de laberintos, sino que también se desbloqueará su potencial en una amplia variedad de aplicaciones. Mientras se explora cómo esta tecnología puede transformar la forma en que se abordan los laberintos y cómo podría ser una herramienta valiosa en la resolución de problemas en sectores tan diversos como la navegación, la planificación de rutas, la optimización de redes, la robótica y más.

MARCO TEORICO

Laberinto: El término laberinto viene del griego labyrinthos, y describe una estructura en forma de nudo espiral con un solo camino de salida, a diferencia de los laberintos modernos, que brindan múltiples vías entrelazadas. Etimológicamente, la palabra nos remite al labrys o doble hacha minoica, el símbolo de la diosa madre de Creta, aunque la palabra tiene su origen en Lidia, y posiblemente llegó a Creta desde Anatolia (Asia Menor) a través del comercio.

Un laberinto se define como una estructura arquitectónica o un patrón espacial compuesto por una serie de pasillos, caminos o senderos entrelazados de manera intrincada. Con ramificaciones y decisiones que pueden llevar a múltiples direcciones.

Los laberintos suelen ser diseñados para desafiar la navegación y la orientación. Creando un entorno laberíntico confuso en el que los individuos pueden perderse o experimentar dificultades para encontrar la salida.

En un laberinto típico, existe un camino correcto que conduce a la salida, mientras que otros caminos pueden llevar a callejones sin salida o bucles que retroceden hacia puntos anteriores del laberinto.

Asimismo, los laberintos pueden ser de diferentes tamaños y complejidades, desde laberintos pequeños utilizados como juegos o desafíos de entretenimiento hasta grandes estructuras arquitectónicas que forman parte de parques, jardines o edificios culturales.

Es importante destacar que los laberintos se diferencian de los laberintos multicursales, donde hay múltiples caminos posibles, y de los laberintos unicursales, que tienen solo un camino lineal hacia una salida.

Búsqueda en anchura (BFS): Una búsqueda en anchura (BFS) es un algoritmo de búsqueda para lo cual recorre los nodos de un grafo, comenzando en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo), para luego explorar todos los vecinos de este nodo. A continuación, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo. Cabe resaltar que si se encuentra el nodo antes de recorrer todos los nodos, concluye la búsqueda.

La búsqueda por anchura se usa para aquellos algoritmos en donde resulta crítico elegir el mejor camino posible en cada momento del recorrido.

Búsqueda en profundidad (DFS): La búsqueda en profundidad (DFS) es un algoritmo para atravesar o buscar estructuras de datos de árboles o gráficos. El algoritmo comienza en el nodo raíz (seleccionando algún nodo arbitrario como nodo raíz en el caso de un gráfico) y explora lo más lejos posible a lo largo de cada rama antes de retroceder. Se necesita memoria adicional, generalmente una pila, para realizar un seguimiento de los nodos descubiertos hasta el momento a lo largo de una rama específica, lo que ayuda a retroceder en el gráfico.

Funciones y librerías que desconocía

_kbhit() es una función en el lenguaje de programación C que se utiliza para determinar si se ha presionado una tecla en el teclado sin bloquear la ejecución del programa. El nombre _kbhit proviene de "keyboard hit" (tecla de teclado presionada). Esta función es específica de algunas implementaciones de la biblioteca de conio en sistemas DOS y Windows.

HANDLE: En el contexto de programación en Windows utilizando el lenguaje C, un handle se refiere a un tipo de dato llamado HANDLE. Este tipo se utiliza para representar manejadores de objetos del sistema operativo, como archivos,

procesos, hilos, semáforos, y otros objetos del sistema. Los manejadores (HANDLEs) son esenciales en la programación de Windows para interactuar con recursos del sistema operativo.

DEFINICION DE PROBLEMÁTICA

Los laberintos son estructuras que constan de múltiples caminos y ramas que desafían nuestra orientación y habilidades lógicas. A menudo nos encontramos con laberintos en juegos, libros, películas y otros entretenimientos. Sin embargo, resolver el laberinto no es tarea fácil. Requiere mucha paciencia, concentración y estrategia. Pero ¿Y si tuviéramos una herramienta que ayudara a resolver casi cualquier laberinto de forma rápida y sencilla? Pues este es el objetivo de este proyecto.

Dificultades de Resolver Laberintos Manualmente:

Resolver un laberinto manualmente implica seguir una serie de pasos que pueden ser tediosos y confusos. Primero, debemos identificar el punto de entrada y el punto de salida del laberinto. Luego, debemos explorar los diferentes caminos posibles, evitando los callejones sin salida. Además, debemos recordar el camino que hemos seguido para poder volver atrás si nos equivocamos o encontrar una ruta alternativa si hay más de una solución. Todo esto requiere de un alto nivel de razonamiento lógico y memoria espacial.

Necesidad de una Solución Automatizada:

Debido a las dificultades involucradas en resolver un laberinto manualmente, existe una clara necesidad de una solución automatizada que pueda hacerlo por nosotros. Una solución automatizada es aquella que utiliza un algoritmo o un programa informático para encontrar el camino más corto o óptimo entre el punto de entrada y el punto de salida de un laberinto. Una solución automatizada nos ahorraría tiempo, esfuerzo y frustración al resolver un laberinto. Además, nos permitiría disfrutar más del proceso y del resultado.

Ejemplo de problemática:

Supongamos que los laberintos seguirán la siguiente simbología:

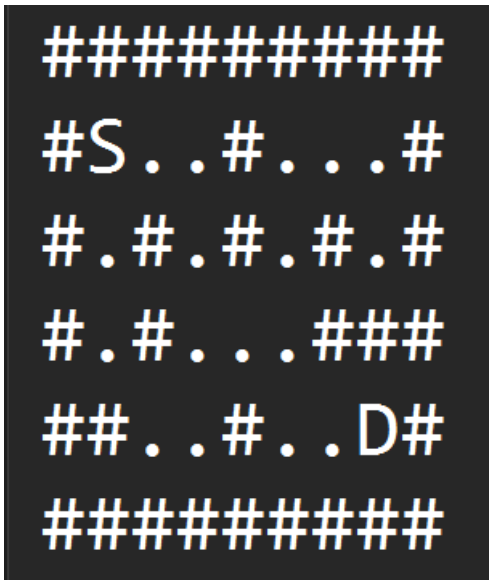
para representar una pared.

. para representar un espacio libre.

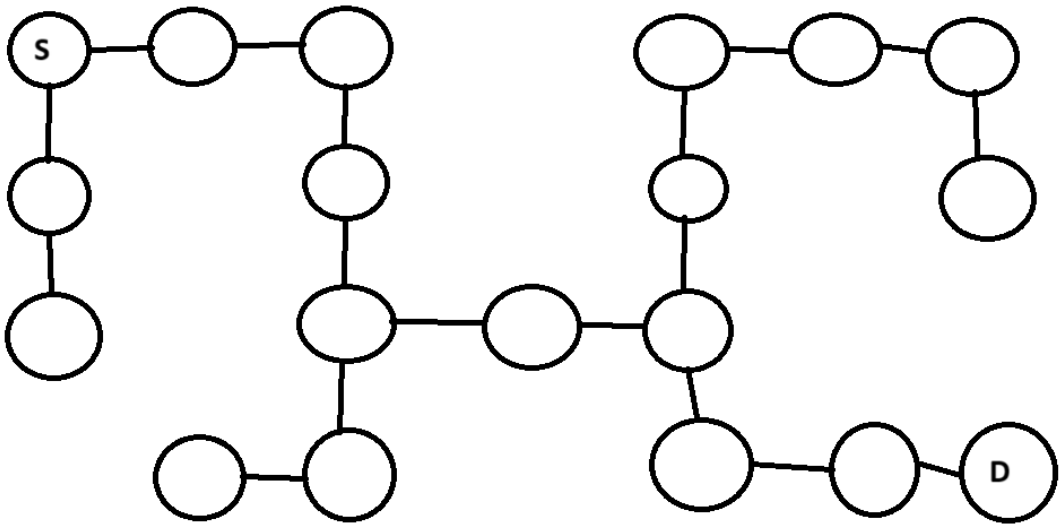
S para representar el punto de inicio.

D para representar el punto de destino.

Un ejemplo de laberinto sería el siguiente:



Para resolverlo podríamos verlo de otra manera para que este sea más simple de observar. Por ejemplo, el laberinto convertido en grafo seria así:



A simple vista, parece que hay varias rutas posibles para llegar desde el punto de inicio al punto de destino. Esto puede hacer que sea más complicado decidir cuál es la ruta óptima. Esto hace que este laberinto sea complicado de resolver y requiere un pensamiento estratégico para encontrar la mejor manera de llegar al punto de destino desde el punto de inicio.

Importancia de la problemática:

Los laberintos son problemas clásicos en el campo de la inteligencia artificial. Resolver laberintos se utiliza para desarrollar y probar algoritmos de búsqueda y planificación, que son fundamentales en aplicaciones de IA, como sistemas de navegación, robótica y juegos. Los problemas de laberintos son versátiles y relevantes en muchas disciplinas y situaciones de la vida cotidiana. Ayudan a

desarrollar habilidades cognitivas, a optimizar procesos y a abordar desafíos del mundo real, lo que los convierte en un área importante de estudio y entretenimiento.

En conclusión, resolver un laberinto es una problemática que requiere de mucha habilidad y destreza. La problemática de resolver los laberintos es un tema relevante y versátil que abarca diversos campos, desde la investigación en inteligencia artificial hasta el entretenimiento y la planificación urbana. Los laberintos representan desafíos que involucran la toma de decisiones, la resolución de problemas y la optimización de rutas, lo que los hace valiosos para el desarrollo de habilidades cognitivas y la aplicación en situaciones del mundo real.

PROPUESTA DE SOLUCION

La propuesta de solución para enfrentar la problemática es crear un programa que resuelva de manera automática laberintos proporcionados por el usuario.

Para crear un solo programa en C que resuelva diferentes laberintos proporcionados por el usuario a través de archivos de texto, se seguirán estos pasos generales:

Leer el Laberinto desde un Archivo de Texto: El primer paso es leer el laberinto desde un archivo de texto. Se abrirá el archivo de texto proporcionado por el usuario y cargará el laberinto en una estructura de datos en memoria. Asegurando el identificar la ubicación de inicio (S) y el destino (D).

Construir un Grafo: Se convertirá el laberinto en una representación de grafo. Cada celda transitable (.) se convierte en un nodo del grafo y se conecta con sus celdas vecinas transversales. Las celdas con # (paredes) no se incluirán en el grafo.

Resolver el Laberinto con Grafos: Se utilizará un algoritmo de búsqueda en grafos, como la búsqueda en profundidad (DFS) o la búsqueda en amplitud (BFS), para encontrar el camino desde el punto de inicio (S) hasta el punto de destino (D) en el grafo. El algoritmo debe mantener un registro de las celdas visitadas para evitar ciclos infinitos.

Marcar la Ruta en el Laberinto: Después de encontrar el camino, se marcará las celdas del laberinto que forman parte de la ruta desde el punto de inicio hasta el punto de destino.

Imprimir el Laberinto Resuelto: Al final se imprimirá el laberinto con la ruta marcada para mostrar el camino desde S hasta D.

Además de los pasos mencionados anteriormente, es importante considerar algunos aspectos adicionales en la implementación de este programa para garantizar su eficacia y usabilidad:

Interacción con el Usuario: Se debe incorporar una interfaz de usuario que permita a los usuarios proporcionar el archivo de texto con el laberinto y recibir la solución de manera clara.

Manejo de Errores: El programa debería ser capaz de manejar posibles errores, como la falta de un punto de inicio (S) o un destino (D) en el laberinto proporcionado, o la presencia de caracteres inválidos en el archivo de texto. Se deben proporcionar mensajes de error informativos para guiar al usuario en la corrección de estos problemas.

Documentación y Comentarios: Para que el programa sea comprensible y mantenible, es fundamental incluir documentación y comentarios en el código fuente. Esto facilitará la comprensión de cómo funciona el programa y permitirá futuras modificaciones o mejoras.

Pruebas y Validación: Antes de poner en funcionamiento el programa, se deben realizar pruebas exhaustivas con diferentes laberintos, incluyendo casos de prueba que cubran diversas situaciones posibles. Esto ayudará a garantizar que el programa funcione de manera correcta y confiable.

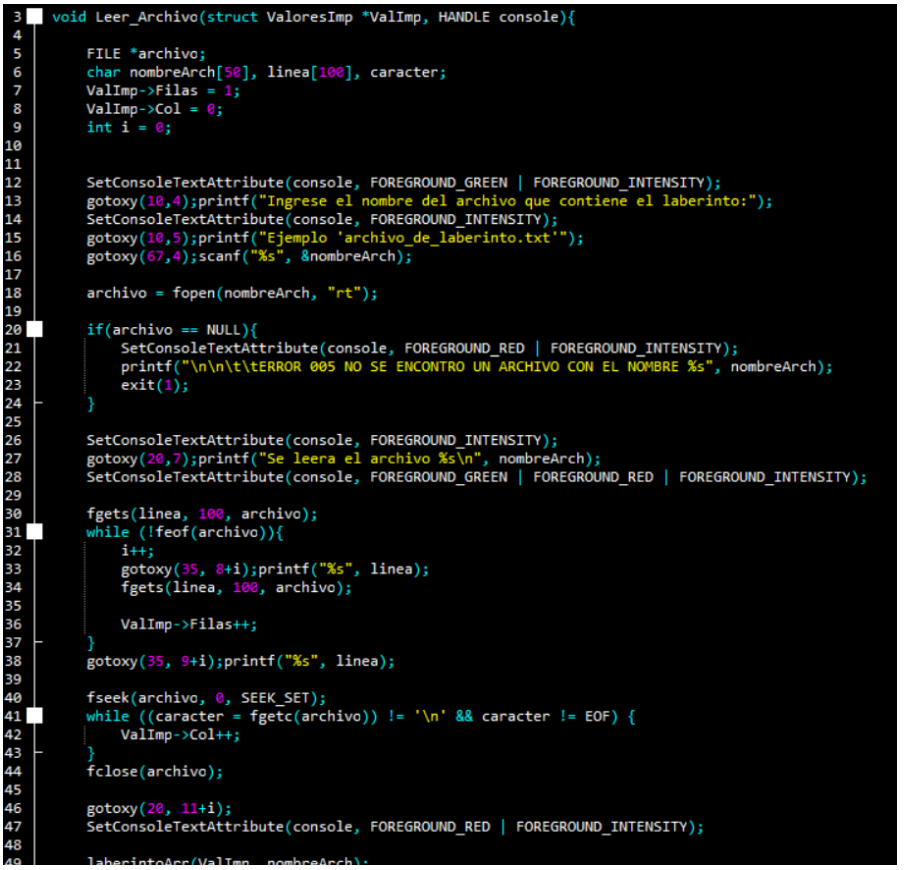
En resumen, la creación de un programa que resuelva laberintos de manera automática a partir de archivos de texto es una tarea que requiere planificación, diseño cuidadoso y pruebas exhaustivas. Al seguir los pasos y consideraciones adicionales mencionados anteriormente, se puede desarrollar un programa efectivo y útil para abordar esta problemática.

Desarrollo

1. **Leer el Laberinto desde un Archivo de Texto:**

Se creo una función en C que puede leer un archivo de texto que contiene la representación del laberinto.

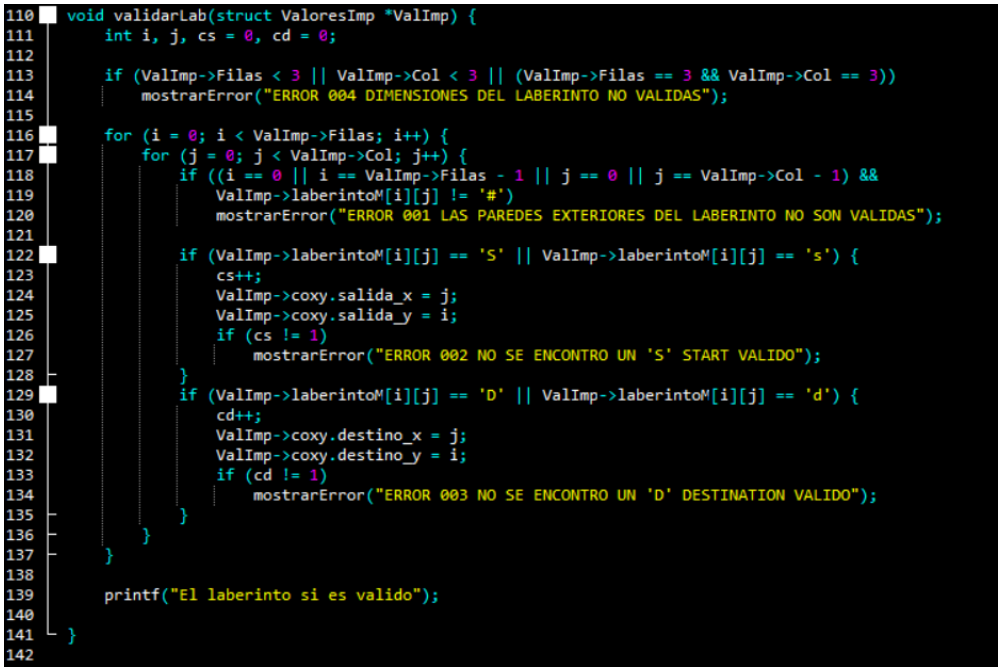
La función Leer_Archivo tiene como objetivo leer un laberinto desde un archivo de texto, obtener información relevante sobre sus dimensiones y coordenadas de inicio y destino, y luego realizar algunas validaciones.



```c
void Leer_Archivo(struct ValoresImp *ValImp, HANDLE console){

    FILE *archivo;
    char nombreArch[50], linea[100], caracter;
    ValImp->Filas = 1;
    ValImp->Col = 0;
    int i = 0;


    SetConsoleTextAttribute(console, FOREGROUND_GREEN | FOREGROUND_INTENSITY);
    gotoxy(10,4);printf("Ingrese el nombre del archivo que contiene el laberinto:");
    SetConsoleTextAttribute(console, FOREGROUND_INTENSITY);
    gotoxy(10,5);printf("Ejemplo 'archivo_de_laberinto.txt'");
    gotoxy(67,4);scanf("%s", &nombreArch);

    archivo = fopen(nombreArch, "rt");

    if(archivo == NULL){
        SetConsoleTextAttribute(console, FOREGROUND_RED | FOREGROUND_INTENSITY);
        printf("\n\n\t\tERROR 005 NO SE ENCONTRO UN ARCHIVO CON EL NOMBRE %s", nombreArch);
        exit(1);
    }

    SetConsoleTextAttribute(console, FOREGROUND_INTENSITY);
    gotoxy(20,7);printf("Se leera el archivo %s\n", nombreArch);
    SetConsoleTextAttribute(console, FOREGROUND_GREEN | FOREGROUND_RED | FOREGROUND_INTENSITY);

    fgets(linea, 100, archivo);
    while (!feof(archivo)){
        i++;
        gotoxy(35, 8+i);printf("%s", linea);
        fgets(linea, 100, archivo);

        ValImp->Filas++;
    }
    gotoxy(35, 9+i);printf("%s", linea);

    fseek(archivo, 0, SEEK_SET);
    while ((caracter = fgetc(archivo)) != '\n' && caracter != EOF) {
        ValImp->Col++;
    }
    fclose(archivo);

    gotoxy(20, 11+i);
    SetConsoleTextAttribute(console, FOREGROUND_RED | FOREGROUND_INTENSITY);
```

Identificar la ubicación de inicio (S) y el destino (D) en el laberinto leído.

La función verifica la validez del laberinto en términos de dimensiones, marcado de paredes exteriores y presencia única de las entradas 'S' y 'D', al encontrar S o D guarda las coordenadas en una estructura. Si alguna condición no se cumple, se muestra un mensaje de error correspondiente.



```c
void validarLab(struct ValoresImp *ValImp) {
    int i, j, cs = 0, cd = 0;

    if (ValImp->Filas < 3 || ValImp->Col < 3 || (ValImp->Filas == 3 && ValImp->Col == 3))
        mostrarError("ERROR 004 DIMENSIONES DEL LABERINTO NO VALIDAS");

    for (i = 0; i < ValImp->Filas; i++) {
        for (j = 0; j < ValImp->Col; j++) {
            if ((i == 0 || i == ValImp->Filas - 1 || j == 0 || j == ValImp->Col - 1) &&
                ValImp->laberintoM[i][j] != '#')
                mostrarError("ERROR 001 LAS PAREDES EXTERIORES DEL LABERINTO NO SON VALIDAS");

            if (ValImp->laberintoM[i][j] == 'S' || ValImp->laberintoM[i][j] == 's') {
                cs++;
                ValImp->coxy.salida_x = j;
                ValImp->coxy.salida_y = i;
                if (cs != 1)
                    mostrarError("ERROR 002 NO SE ENCONTRO UN 'S' START VALIDO");
            }
            if (ValImp->laberintoM[i][j] == 'D' || ValImp->laberintoM[i][j] == 'd') {
                cd++;
                ValImp->coxy.destino_x = j;
                ValImp->coxy.destino_y = i;
                if (cd != 1)
                    mostrarError("ERROR 003 NO SE ENCONTRO UN 'D' DESTINATION VALIDO");
            }
        }
    }

    printf("El laberinto si es valido");

}
```

Cargar el laberinto en una estructura de datos en memoria que facilite la manipulación y búsqueda.

Esta función sirve para almacenar un laberinto, obtenido de un archivo, en una estructura de valores importantes para el programa

```c
void laberintoArr(struct ValoresImp *ValImp, char nombreAr[50]){

    FILE *arch;
    int i, j;
    char caracter;

    ValImp->laberintoM = (char **)malloc(ValImp->Filas * sizeof(char *));
    if (ValImp->laberintoM == NULL) {
        printf("Error: No se pudo asignar memoria para las filas.\n");
        exit(1);
    }

    // Asigna memoria para las columnas de cada fila
    for (i = 0; i < ValImp->Filas; i++) {
        ValImp->laberintoM[i] = (char *)malloc(ValImp->Col * sizeof(char));
        if (ValImp->laberintoM[i] == NULL) {
            printf("Error: No se pudo asignar memoria para las columnas.\n");
            exit(1);
        }
    }

    arch = fopen(nombreAr, "rt");
    if(arch == NULL){
        printf("\n\n\t\tERROR 005 NO SE ENCONTRO UN ARCHIVO CON EL NOMBRE %s", nombreAr);
        exit(1);
    }

    caracter = fgetc(arch);
    for (i = 0; i < ValImp->Filas; i++) {
        for (j = 0; j < ValImp->Col; j++) {
            while (caracter == '\n') {
                caracter = fgetc(arch);
            }
            ValImp->laberintoM[i][j] = caracter;
            caracter = fgetc(arch);
        }
    }
    fclose(arch);

}
```

2. **Construir un Grafo:**

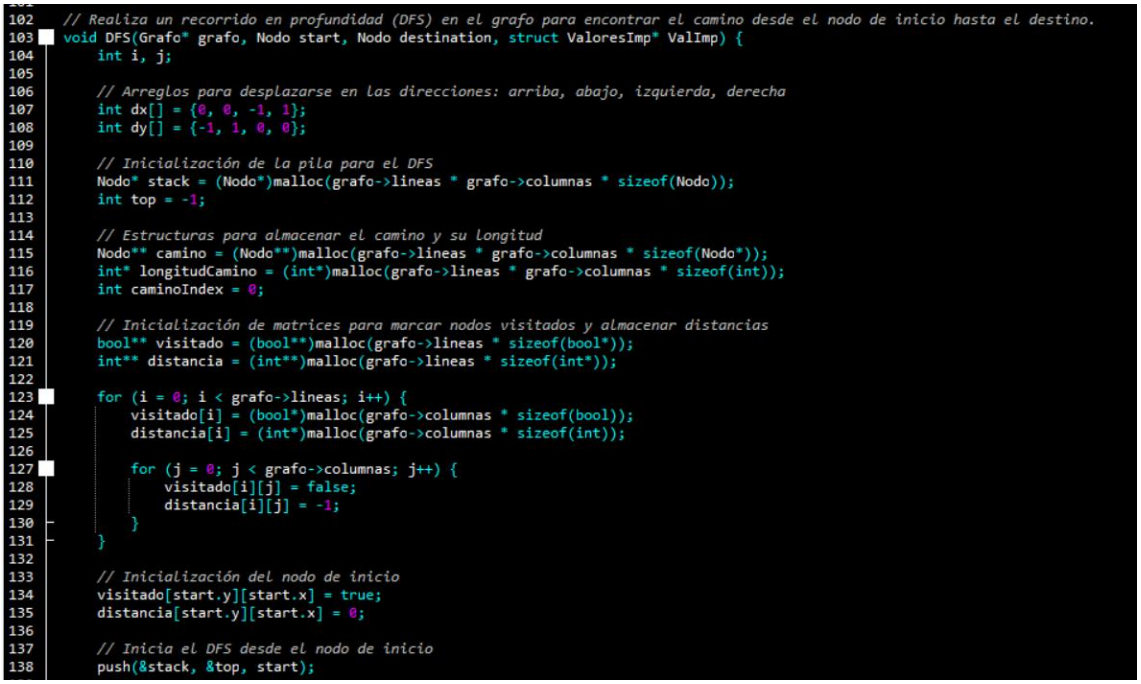
Se creo una función que convierte la representación del laberinto en una estructura de grafo, para ser más exactos, en lista de adyacencias.
Cada celda transitable (.) debe convertirse en un nodo del grafo y conectarse con sus celdas vecinas transversales.
Las celdas con paredes (#) no deben incluirse en el grafo.

```c
// Convierte el laberinto en un grafo y encuentra el camino utilizando DFS
void convertirGrafo(struct ValoresImp* ValImp, HANDLE console) {
    int i, j;
    Grafo* grafo = iniGrafo(ValImp->Filas, ValImp->Col);
    Nodo start = { ValImp->coxy.salida_x, ValImp->coxy.salida_y };
    Nodo destination = { ValImp->coxy.destino_x, ValImp->coxy.destino_y };

    // Identifica nodos y conexiones en el laberinto
    for (i = 0; i < ValImp->Filas; i++) {
        for (j = 0; j < ValImp->Col; j++) {
            if (ValImp->laberintoM[i][j] == '.' || ValImp->laberintoM[i][j] == 'S' || ValImp->laberintoM[i][j] == 'D' || ValImp->laberintoM[i][j] == 's' || ValImp-
                Nodo currentNode = { j, i };

                // Conectar con nodos adyacentes
                if (i > 0 && (ValImp->laberintoM[i - 1][j] == '.' || ValImp->laberintoM[i - 1][j] == 'S' || ValImp->laberintoM[i - 1][j] == 'D' || ValImp->laberint
                    addEdge(grafo, currentNode, (Nodo) { j, i - 1 });
                }
                if (i < ValImp->Filas - 1 && (ValImp->laberintoM[i + 1][j] == '.' || ValImp->laberintoM[i + 1][j] == 'S' || ValImp->laberintoM[i + 1][j] == 'D' ||
                    addEdge(grafo, currentNode, (Nodo) { j, i + 1 });
                }
                if (j > 0 && (ValImp->laberintoM[i][j - 1] == '.' || ValImp->laberintoM[i][j - 1] == 'S' || ValImp->laberintoM[i][j - 1] == 'D' || ValImp->laberint
                    addEdge(grafo, currentNode, (Nodo) { j - 1, i });
                }
                if (j < ValImp->Col - 1 && (ValImp->laberintoM[i][j + 1] == '.' || ValImp->laberintoM[i][j + 1] == 'S' || ValImp->laberintoM[i][j + 1] == 'D' || Va
                    addEdge(grafo, currentNode, (Nodo) { j + 1, i });
                }
            }
        }
    }
```

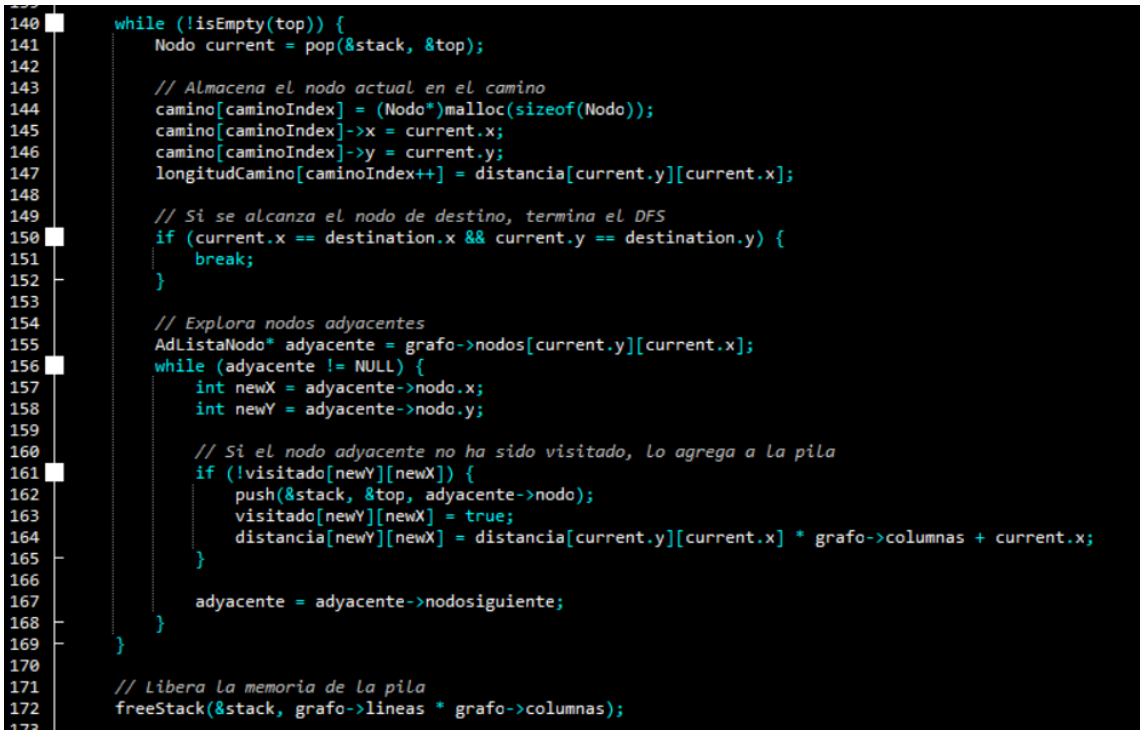
En esta función se construye el grafo, manda a llamar a otras funciones para ayudar a esto, entre ellas una que inicializa el grafo y otra que crea las conexiones entre cada nodo

3. **Resolver el Laberinto con Grafos:**

Se busco implementar un algoritmo de búsqueda en grafos, como la búsqueda en profundidad (DFS) o la búsqueda en amplitud (BFS) para utilizar el algoritmo para encontrar el camino desde el punto de inicio (S) hasta el punto de destino (D) y para mantener un registro de las celdas visitadas para evitar ciclos infinitos. El primer intento fue con el algoritmo BFS el cual, aunque si recorría todos los nodos y llegaba al destino, tenia la dificultad de desmarcar los caminos innecesarios así que se opto utilizar el algoritmo DFS



```c
// Realiza un recorrido en profundidad (DFS) en el grafo para encontrar el camino desde el nodo de inicio hasta el destino.
void DFS(Grafo* grafo, Nodo start, Nodo destination, struct ValoresImp* ValImp) {
    int i, j;

    // Arreglos para desplazarse en las direcciones: arriba, abajo, izquierda, derecha
    int dx[] = {0, 0, -1, 1};
    int dy[] = {-1, 1, 0, 0};

    // Inicialización de la pila para el DFS
    Nodo* stack = (Nodo*)malloc(grafo->lineas * grafo->columnas * sizeof(Nodo));
    int top = -1;

    // Estructuras para almacenar el camino y su longitud
    Nodo** camino = (Nodo**)malloc(grafo->lineas * grafo->columnas * sizeof(Nodo*));
    int* longitudCamino = (int*)malloc(grafo->lineas * grafo->columnas * sizeof(int));
    int caminoIndex = 0;

    // Inicialización de matrices para marcar nodos visitados y almacenar distancias
    bool** visitado = (bool**)malloc(grafo->lineas * sizeof(bool*));
    int** distancia = (int**)malloc(grafo->lineas * sizeof(int*));

    for (i = 0; i < grafo->lineas; i++) {
        visitado[i] = (bool*)malloc(grafo->columnas * sizeof(bool));
        distancia[i] = (int*)malloc(grafo->columnas * sizeof(int));

        for (j = 0; j < grafo->columnas; j++) {
            visitado[i][j] = false;
            distancia[i][j] = -1;
        }
    }

    // Inicialización del nodo de inicio
    visitado[start.y][start.x] = true;
    distancia[start.y][start.x] = 0;

    // Inicia el DFS desde el nodo de inicio
    push(&stack, &top, start);
```



```c
    while (!isEmpty(top)) {
        Nodo current = pop(&stack, &top);

        // Almacena el nodo actual en el camino
        camino[caminoIndex] = (Nodo*)malloc(sizeof(Nodo));
        camino[caminoIndex]->x = current.x;
        camino[caminoIndex]->y = current.y;
        longitudCamino[caminoIndex++] = distancia[current.y][current.x];

        // Si se alcanza el nodo de destino, termina el DFS
        if (current.x == destination.x && current.y == destination.y) {
            break;
        }

        // Explora nodos adyacentes
        AdListaNodo* adyacente = grafo->nodos[current.y][current.x];
        while (adyacente != NULL) {
            int newX = adyacente->nodo.x;
            int newY = adyacente->nodo.y;

            // Si el nodo adyacente no ha sido visitado, lo agrega a la pila
            if (!visitado[newY][newX]) {
                push(&stack, &top, adyacente->nodo);
                visitado[newY][newX] = true;
                distancia[newY][newX] = distancia[current.y][current.x] * grafo->columnas + current.x;
            }

            adyacente = adyacente->nodosiguiente;
        }
    }

    // Libera la memoria de la pila
    freeStack(&stack, grafo->lineas * grafo->columnas);
```

Este algoritmo se basa en recorrer el grafo (laberinto convertido) por profundidad, esto quiere decir que el algoritmo determina porque camino irse y explora lo mas lejos posible y después regresa para tomar otro camino, en el programa se utilizó una pila dinámica para llevar a cabo el algoritmo, y poder visitar todos los nodos adyacentes necesarios hasta llegar al destino, una ves lo encuentra el programa imprime el recorrido por profundidad realizado

4. **Marcar la Ruta en el Laberinto:**
   Después de encontrar el camino, marcar las celdas del laberinto que forman parte de la ruta desde el punto de inicio hasta el punto de destino.

```
185       // Marca el camino en el laberinto con 'X'
186       int auxiliar = camino[caminoIndex - 1]->x + camino[caminoIndex - 1]->y;
187       int a = camino[caminoIndex - 1]->x,b = camino[caminoIndex - 1]->y;
188       for (i = caminoIndex - 1; i >= 0; i--) {
189           int m = camino[i]->x;
190           int n = camino[i]->y;
191           if (ValImp->laberintoM[n][m] == 'S' || ValImp->laberintoM[n][m] == 'D' || ValImp->laberintoM[n][m] == 's' || ValImp->laberintoM[n][m] == 'd') {
192               // No realiza ninguna acción si el nodo es la salida o el destino
193           } else {
194               // Marca el camino con 'X' en el laberinto
195               if ((m + n - auxiliar == 1 || m + n - auxiliar == -1)&& (a == m || b == n)) {
196                   ValImp->laberintoM[n][m] = 'X';
197                   auxiliar = m + n;
198                   a = m;
199                   b = n;
200               }
201           }
202       }
203
```

   Para solucionar el problema de no marcar los caminos de más, se encontró una solución un poco arcaica pero funcional, se recorre el camino encontrado de atrás hacia adelante para así poder crear una condición que ayudara a eliminar a esos nodos innecesarios en la solución del laberinto

5. **Imprimir el Laberinto Resuelto:**

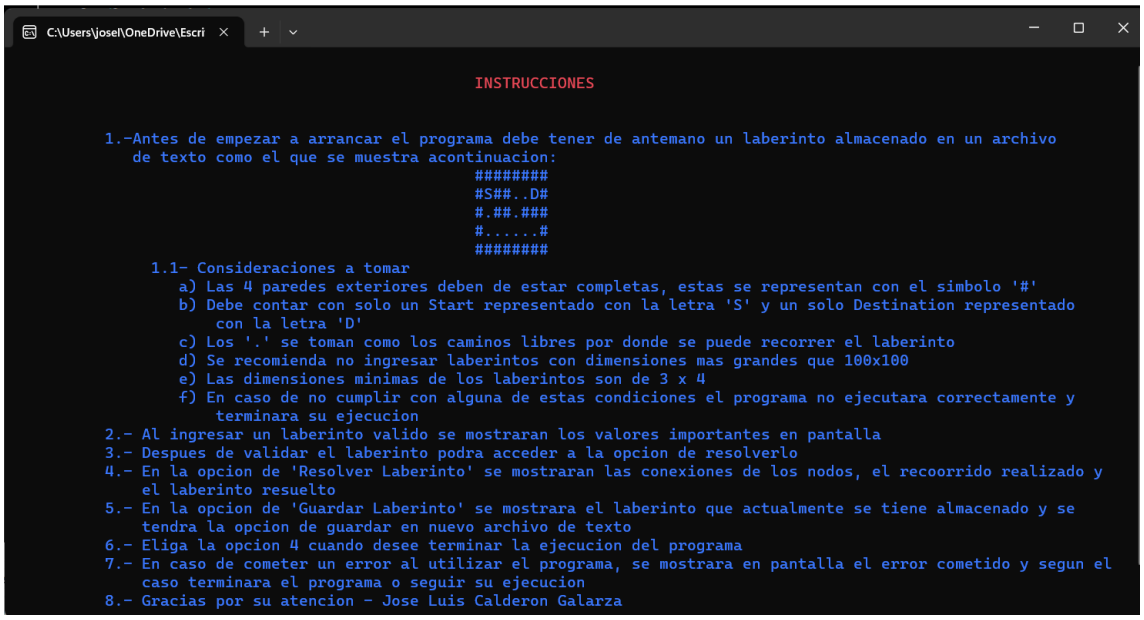
Se creo una función para imprimir el laberinto con la ruta marcada para mostrar el camino desde S hasta D de manera clara para que el usuario pueda visualizar la solución.

```
216   // Imprime el laberinto resaltando la salida (S), destino (D) y el camino (X)
217   void printLab(struct ValoresImp* ValImp, HANDLE console) {
218       int i, j;
219       for (i = 0; i < ValImp->Filas; i++) {
220           for (j = 0; j < ValImp->Col; j++) {
221               if (ValImp->laberintoM[i][j] == 'S' || ValImp->laberintoM[i][j] == 'D' || ValImp->laberintoM[i][j] == 's' || ValImp->laberintoM[i][j] == 'd') {
222                   SetConsoleTextAttribute(console, FOREGROUND_GREEN | FOREGROUND_INTENSITY);
223                   printf("%c", ValImp->laberintoM[i][j]);  // Resalta la salida, destino en verde brillante
224                   SetConsoleTextAttribute(console, FOREGROUND_BLUE | FOREGROUND_INTENSITY);
225               } else {
226                   if (ValImp->laberintoM[i][j] == 'X') {
227                       SetConsoleTextAttribute(console, FOREGROUND_RED | FOREGROUND_INTENSITY);
228                       printf("%c", ValImp->laberintoM[i][j]);  // Resalta el camino en rojo brillante
229                       SetConsoleTextAttribute(console, FOREGROUND_BLUE | FOREGROUND_INTENSITY);
230                   } else {
231                       printf("%c", ValImp->laberintoM[i][j]);  // Imprime los otros caracteres normales
232                   }
233               }
234           }
235           printf("\n");
236       }
237       printf("\n");
238       getch();  // Pausa para visualización
239   }
```

En esta función se imprime el laberinto además que se resalta el camino y los puntos S y D para una mejor visualización para el usuario

6. **Interacción con el Usuario:**

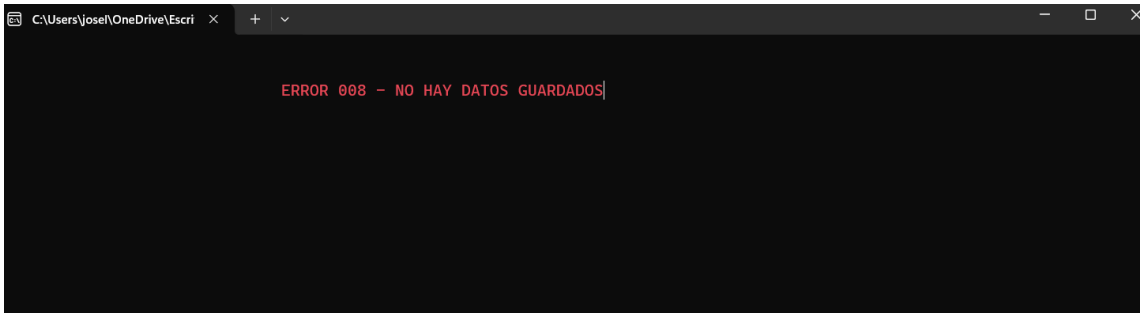
Implementar una interfaz de usuario que permita a los usuarios proporcionar el archivo de texto con el laberinto.
Proporcionar mensajes claros y amigables para guiar al usuario durante la interacción.



Se creo una pantalla de instrucciones antes de iniciar con el programa para ayudar al usuario y pueda comprender el funcionamiento de esto

7. **Manejo de Errores:**

A lo largo del programa se llevaron a cabo manejo de errores para que el programa se ejecute de la mejor manera, además se implementó la función para a la hora de pasar un error aparezca en la pantalla que fue lo que causo el error como se muestra a continuación:

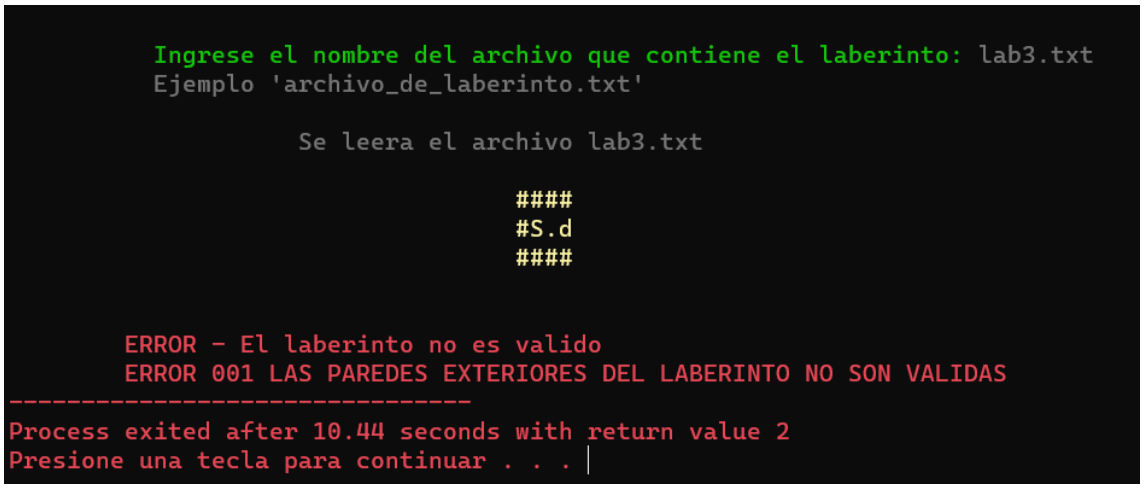


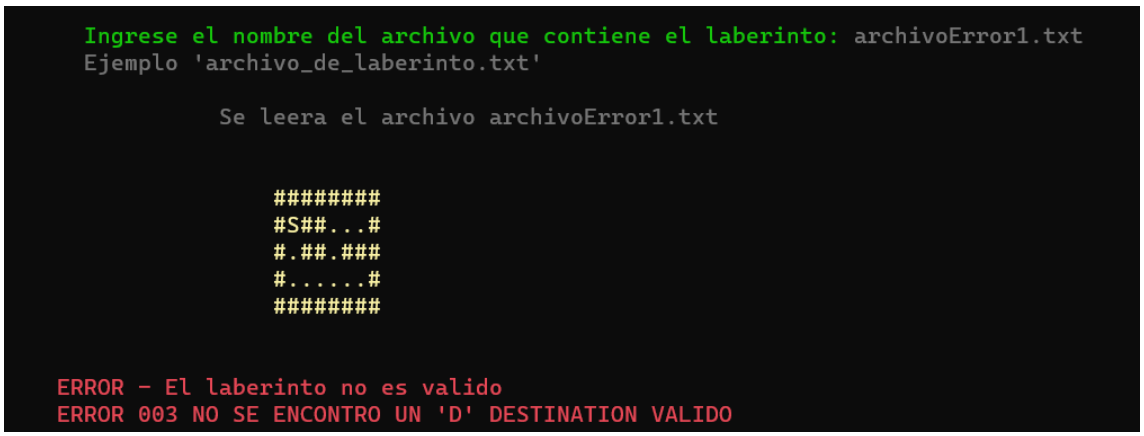
8. **Documentación y Comentarios:**

Esta documentación del código fuente explica el propósito y funcionamiento de cada función. Y además se agregaron comentarios para facilitar la comprensión del código y posibles modificaciones futuras.

9. **Pruebas y Validación:**

Se realizaron varias pruebas utilizando diferentes laberintos, incluyendo casos de prueba que cubran diversas situaciones posibles. A continuación algunos errores posibles por el usuario.





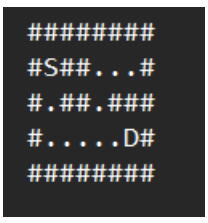
```
          Ingrese el nombre del archivo que contiene el laberinto: archivo
          Ejemplo 'archivo_de_laberinto.txt'

                ERROR 005 NO SE ENCONTRO UN ARCHIVO CON EL NOMBRE archivo
--------------------------------
Process exited after 10.23 seconds with return value 1
Presione una tecla para continuar . . .
```

```
         Posible camino encontrado para resolver el laberinto:
         NOTA - Si no se marca un camino por completo es porque no existe solucion


  ##########################
  #S##XX.......#############
  #XXX.X.####################
  ###XXX#######...###########
  #.##.#######.##############
  #..........##############
  #.###########.##############
  #.##.#######.##############
  #......#####.##############
  ######.......##############
  #.##.#######.##############
  #......#####.##############
  #############...###########
  #.##...#######..........##
  #.##.#########.###.####.##
  #............###.####.##
  ####.##############.#######
  #.##...######...........##
  #.##.####..######.######.##
  #......###.###...######.##
  #####.####.....#.##..#D####
  #.##.........##.....######
  #.##.##################.###
  #.....................###
  ##########################
```

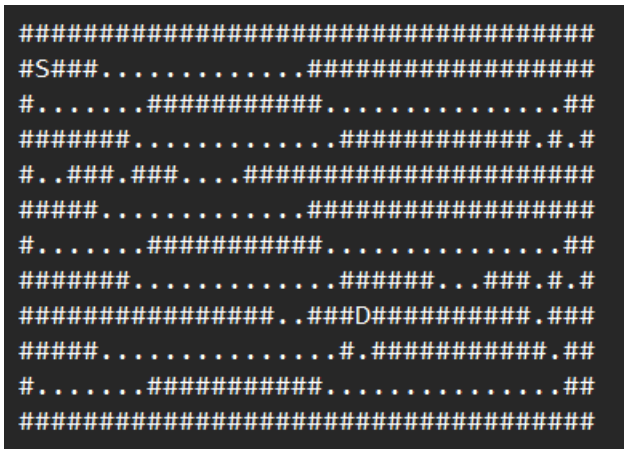
Análisis de resultados

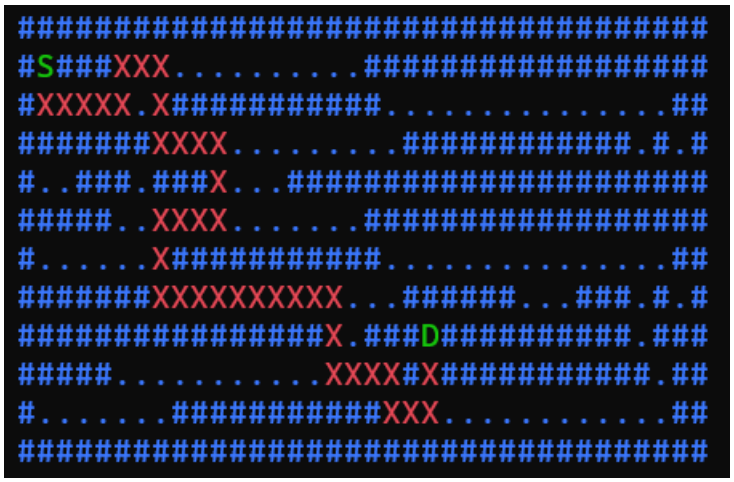
A continuación, se muestra las pruebas realizadas al programa al ingresar laberintos validos

1. archivo_de_laberinto.txt

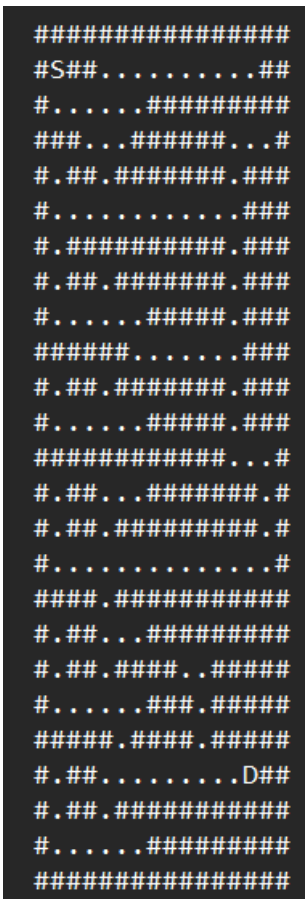


2. archivoLab1.txt

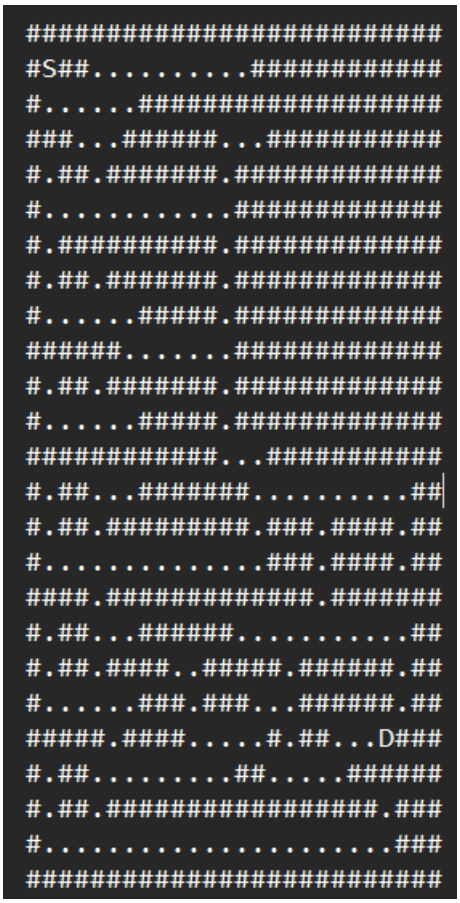


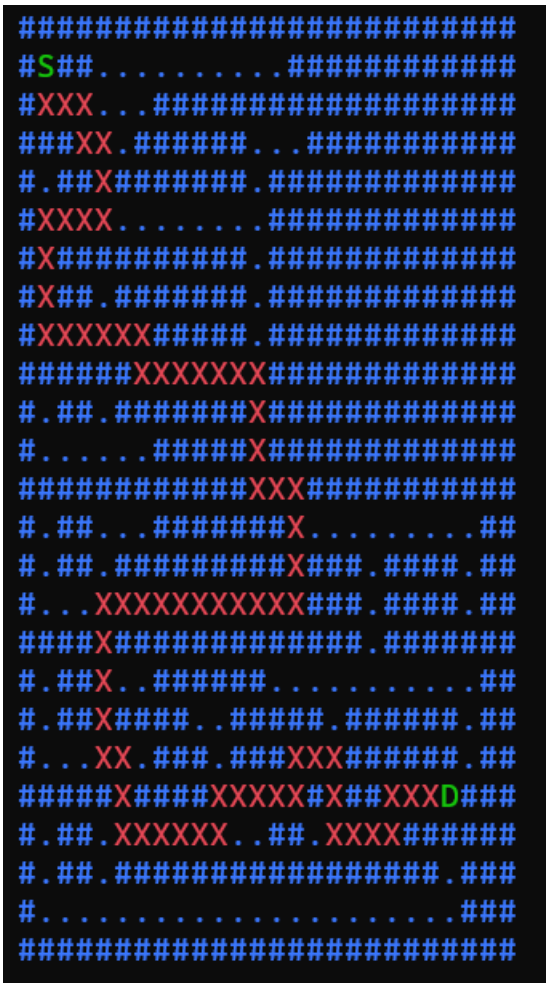


3. archivoLab2.txt



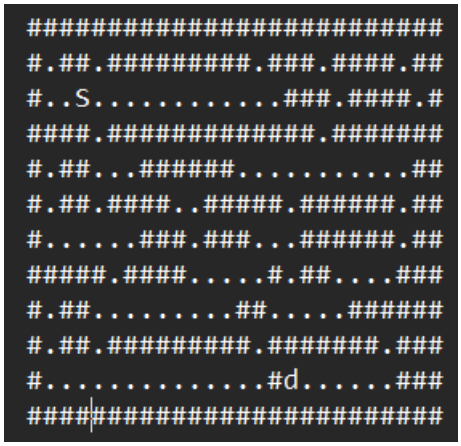
4. archivoLab3.txt

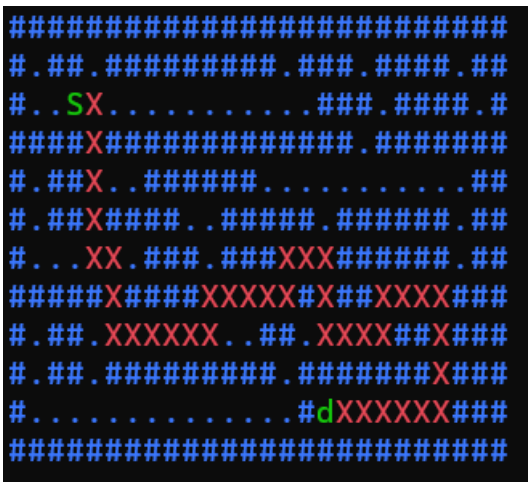

#########################
#S##.....############
#.....##################
###...######...##########
#.##.#######.############
#...........############
#.#########.############
#.##.#######.############
#.....#####.############
######.......############
#.##.#######.############
#.....#####.############
############...##########
#.##...#######.........##
#.##.#########.###.####.##
#.............###.####.##
####.############.#######
#.##...######..........##
#.##.####..#####.######.##
#.....###.###...######.##
#####.####.....#.##...D###
#.##.........##.....######
#.##.################.###
#....................###
#########################


#########################
#S##.....############
#XXX...##################
###XX.######...##########
#.##X#######.############
#XXXX........############
#X#########.############
#X##.#######.############
#XXXXXX#####.############
######XXXXXXX############
#.##.#######X############
#.....#####X############
############XXX###########
#.##...#######X.........##
#.##.#########X###.####.##
#...XXXXXXXXXXX###.####.##
####X##############.#######
#.##X..######..........##
#.##X####..#####.######.##
#...XX.###.###XXX######.##
#####X####XXXXX#X##XXXD###
#.##.XXXXXX..##.XXXX######
#.##.################.###
#....................###
#########################

5. archivoLab4.txt

6. archivoLab5.txt





Gracias a las pruebas realizadas y los resultados obtenidos podemos decir que el programa cumple con sus objetivos principales, aunque es necesario realizar mucha mas variedad de pruebas para determinar los límites que tiene el programa.

Conclusión:

En conclusión, después de muchas horas de trabajo se ha logrado el objetivo principal del proyecto, crear un programa que resuelve laberintos de manera automática. Gracias a la implementación exitosa de algoritmos de búsqueda en grafos, la interfaz de usuario amigable y el manejo de errores hacen que el programa sea efectivo y útil. Además, esta documentación y comentarios facilitan la comprensión para posibles mejoras futuras. Este programa no solo aborda la problemática de resolver laberintos, sino que también puede ser una herramienta valiosa en el campo de la inteligencia artificial, la planificación de rutas, la optimización de redes y otros sectores. En general, el proyecto ha sido exitoso y cumple con los objetivos establecidos.

Referencias

Solano, F. (2023, julio 26). ¿Qué es un laberinto? Definición, Tipos y Características. Encuentratutarea.com; Encuentra Tu Tarea. https://encuentratutarea.com/que-es-un-laberinto-definicion-tipos-y-caracteristicas/

Mark, J. J. (2018). Laberinto. Enciclopedia de la Historia del Mundo. https://www.worldhistory.org/trans/es/1-531/laberinto/

Búsqueda en profundidad. (s/f). Academia-lab.com. Recuperado el 17 de noviembre de 2023, de https://academia-lab.com/enciclopedia/busqueda-en-profundidad/