Este proyecto consta de un algoritmo que genera un laberinto aleatorio de un número impar de casillas y, además, genera y grafica ( con una interfaz gráfica) uno de sus posibles caminos utilizando un algoritmo de búsqueda profunda (DFS). En sus orígenes, mi intención era que el algoritmo encontrase no uno, sino todos los caminos posibles al laberinto que él mismo había generado, lo que pasa que al existir la posibilidad de generar laberinto de dimensiones muy grandes, se volvía un cálculo muy muy costoso hasta para el ordenador.

Como ya he mencionado anteriormente, el algoritmo utilizado para encontrar uno de sus caminos posibles es el DFS, cuya esencial principal es explorar por completo una rama hasta llegar a un estado de completitud (finalización) o de rechazo, momento en el cual o bien se almacenará la solución o se retrocederá (backtracking). El algoritmo DFS no resulta el más útil de todos a la hora de encontrar un camino, es decir, no encuentra el camino más corto (como lo haría un Dijkstra en grafos ponderados) sino que devuelve el primero que encuentra, lo que en ocasiones nos deja con un camino que da varias vueltas o se complica mucho antes de terminar. A pesar de esto, existe una pequeña solución a este problema y es implementar lo que se conoce como la “Distancia Manhattan”. Se trata de un mecanismo para medir la distancia entre dos puntos en un sistema de cuadrícula ( como por ejemplo las calles de Manhattan, de ahí su nombre). Es similar a la distancia euclidiana, lo que pasa que, en diferencia a esta, la distancia Manhattan no se mueve en líneas rectas arbitrariamente, sino que lo hace con respecto a unos ejes verticales y horizontales, sin moverse diagonalmente. Lo que hace la distancia Manhattan decirte los movimientos mínimos necesario para llegar desde un punto a otro. En definitiva, para hacer mi algortimos DFS mas eficiente podría implementar una funcionalidad extra que priorizase moverse a aquellos vecinos cuya distancia Manhattan hasta la salida fuesen menor, de tal forma que nos acercaríamos cada vez más a la salida.

En muchas ocasiones el programa imprimía un laberinto que no tenía solución (dada la aleatoriedad ) ya que cabía la posibilidad de colocar por ejemplo un muro enfrente de la salida y ya no se podía salir obviamente. Para ello simplemente basta con utilizar un while que compruebe si existe una solución, en cuyo caso contrario pues seguirán barajándose todos los demás parámetros (dimensiones, entrada,salida, …) de forma aleatoria para garantizar que si se imprime un laberinto, este tenga solución.

A demás, se ha implementado, como dije antes, una interfaz gráfica (GUI) y su biblioteca Tkinter, con el objetivo de hacer más agradable a la vista la estructura con la que se muestran ambos laberintos, el enunciado y el resuelto ( uno a la izquierda y el otro a la derecha, en rojo). Además para cada uno de los laberintos, está escrito abajo en el centro las dimensiones aleatorias que han salido en ese laberinto y las coordenadas de la entrada y la salida.

**ALGORITMOS UTILIZADOS Y ANALISIS DE COMPLEJIDAD:**

Para el desarrollo del programa se ha utilizado como mencionamos anteriormente el algoritmo principal del DFS (Deph-First Search) como técnicas algorítmicas hemos usado la selección aleatoria y el Backtracking)

**Análisis de los métodos:**

**1. Método: crear\_laberinto():**

**Genera un laberinto representado por una matriz N x N, donde cada celda se define aleatoriamente como muro o camino. Además, asegura que los bordes estén completamente cerrados con muros.**

* **Complejidad Temporal: O(N^2)**
  + **Se recorre cada celda de la matriz una vez para asignar valores aleatorios.**
  + **Los bordes se establecen con un segundo recorrido lineal, que es despreciable comparado con el recorrido principal.**
* **Complejidad Espacial: O(N^2)**
  + **La matriz ocupa un espacio cuadrático en memoria.**

**Resumen:  
Temporal: O(N^2)  
Espacial: O(N^2)**

**2. Método: crear\_entrada\_salida()**

**Selecciona aleatoriamente una entrada y una salida en los bordes del laberinto, asegurando que no estén en el mismo borde.**

* **Complejidad Temporal: O(1)**
  + **La selección de coordenadas es aleatoria y no depende del tamaño de la matriz.**
* **Complejidad Espacial: O(1)**
  + **Solo se almacenan dos pares de coordenadas.**

**Resumen:  
Temporal: O(1)   
Espacial: O(1)**

**3. Método: encontrar\_un\_camino()**

**Utiliza el algoritmo DFS (Depth-First Search) para encontrar un camino desde la entrada hasta la salida. Marca las celdas visitadas para evitar ciclos y usa Backtracking para retroceder cuando es necesario.**

* **Complejidad Temporal: (N^2)**
  + **En el peor caso, el DFS puede visitar todas las celdas del laberinto antes de encontrar una solución.**
* **Complejidad Espacial: O(N^2)**
  + **Se necesita una matriz adicional para registrar las celdas visitadas.**
  + **La pila de recursión puede crecer hasta N x N en el peor caso.**

**Resumen:  
Temporal: O(N^2)   
Espacial: O(N^2)**

**4. Método: visualizar\_laberinto()**

**Descripción:  
Dibuja cada celda del laberinto en un lienzo gráfico (tk.Canvas), asignándole un color según su tipo (muro, camino, entrada, salida, camino resuelto).**

* **Complejidad Temporal: O(N^2)**
  + **Se recorre la matriz completa para representar cada celda.**
* **Complejidad Espacial: O(1)**
  + **El lienzo ya está predefinido y no escala con el tamaño del laberinto.**

**Resumen:  
Temporal: O(N^2)   
Espacial: O(1)**

**5. Método: generar\_y\_mostrar()**

**Descripción:  
Orquesta los demás métodos: genera el laberinto, selecciona entrada y salida, resuelve el laberinto con DFS y actualiza la visualización.**

* **Complejidad Temporal: O(N^2)**
  + **Incluye la generación de la matriz O(N^2) el DFS O(N^2) y la visualización O(N^2).**
  + **Aunque las tareas se ejecutan secuencialmente, el término dominante sigue siendo cuadrático.**
* **Complejidad Espacial: O(N^2)**
  + **La matriz principal del laberinto y la matriz de celdas visitadas son los elementos más pesados en memoria.**

**Resumen:  
Temporal: O(N^2)   
Espacial: O(N^2)**

**6. Interfaz Gráfica (Tkinter)**

**Descripción:  
La GUI utiliza widgets (Canvas, Label, Button) y organiza su disposición en una estructura jerárquica. Los eventos son manejados mediante callbacks.**

* **Complejidad Temporal O(1)**
  + **Los eventos y acciones (como el clic en el botón) no dependen del tamaño del laberinto.**
* **Complejidad Espacial: O(1)O(1)O(1)**
  + **Los widgets ocupan un espacio fijo en memoria.**

**Resumen:  
Temporal: O(1)  
Espacial: O(1)**

**COMPLEJIDAD FINAL DEL PROGRAMA:**

* Dado que ninguno de los métodos supero en complejidad espacial O(n^2), la complejidad espacial final del programa será de O(n^2) y lo mismo ocurre con la complejidad temporal, por lo que :

**Complejidad espacial** O(n^2)

**Complejidad temporal** O(n^2)

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**