**Método de la ingeniería**

**Integrantes:**

-Pablo Pineda - A00395831

-Daniela Londoño - A00392917

-Isabella Huila - A00394751

**Fase 1: Identificación del problema**

El problema consiste en desarrollar un sistema que pueda generar el legendario Laberinto de Creta y resolverlo utilizando algoritmos de búsqueda adecuados. El sistema debe permitir al usuario elegir un punto inicial y un punto final dentro del laberinto para que el agente pueda comenzar su recorrido. Si el jugador no puede completar la misión, el sistema deberá mostrar los caminos posibles.

| Client | Empresa de videojuegos |
| --- | --- |
| User | Personas que interactúan con el juego |
| Functional requirements | **R1**: Generar el laberinto  **R2**: Permitir que el jugador se mueva dentro del laberinto  **R3:** Buscar el camino con el algoritmo BFS  **R4:**  Buscar el camino con el algoritmo DFS  **R5:** Mostrar los caminos posibles dentro del laberinto |
| Problem context | ● El problema consiste en desarrollar un sistema que pueda generar el legendario Laberinto de Creta y resolverlo utilizando algoritmos de búsqueda adecuados. El sistema debe permitir al usuario elegir un punto inicial y un punto final dentro del laberinto para que el agente pueda comenzar su recorrido. Si el jugador no puede completar la misión, el sistema deberá mostrar los caminos posibles. |
| non-functional requirements | R1: The system needs to be efficient using the minimum resources necessary  R2: The system needs to be functional in such a way that it fulfills the intended functions |

| Name or identifier | **R1**: Generar el Laberinto | | |
| --- | --- | --- | --- |
| Resume | El sistema debe generar un laberinto con pasillos y paredes, el cual tendrá un punto inicial y un punto final. Dentro del laberinto existirán varios caminos que llegarán al punto final . | | |
| Inputs | Input name | Datatype | Selection or repetition condition |
| archivoDeTexto | txt |  |
| General activities necessary to obtain the results | 1. El sistema debe generar un grafo el cual representa el laberinto 2. El sistema debe de definir el punto inicial y el punto final 3. El sistema deberá desplegar la interfaz donde estará el laberinto | | |
| Result or postcondition | 1. El laberinto (Grafo) generado. | | |
| Outputs | Output name | Datatype | Selection or repetition condition |
| Laberinto | Grafo | Que haya algún error y no se pueda generar el grafo correctamente |

| Name or identifier | **R2:** Permitir que el jugador se mueva dentro del laberinto. | | |
| --- | --- | --- | --- |
| Resume | El sistema deberá permitir que el jugador se mueva dentro del laberinto, donde no podrá sobrepasar las paredes. Además si el jugador llega al punto final, presentará un interfaz con el fin del juego y que diga has ganado, también se le presentarán los otros posibles caminos para llegar al final del laberinto. | | |
| Inputs | Input name | Datatype | Selection or repetition condition |
| coordenada X | int |  |
| Coordenada Y | int |  |
| General activities necessary to obtain the results | 1. El sistema deberá de desplegar la interfaz con el laberinto y colocar al jugador en el punto inicial del mismo. 2. El sistema deberá permitir que con unas teclas designadas mover el jugador sin sobrepasar las paredes del laberinto. 3. El sistema permite que un jugador llegue a la meta, desplegando una interfaz indicando que ha ganado el juego. 4. Si el jugador llega a la meta se mostrarán los caminos por los cuales el jugador podría haber ganado. | | |
| Result or postcondition | 1. El movimiento del jugador con las condiciones dadas. | | |
| Outputs | Output name | Datatype | Selection or repetition condition |
| posicionJugador | Vector |  |

| Name or identifier | **R3**: Buscar el camino con el algoritmo BFS | | |
| --- | --- | --- | --- |
| Resume | El sistema permite utilizar este método para recorrer todo los nodos del laberinto y buscar las posibles salidas. | | |
| Inputs | Input name | Datatype | Selection or repetition condition |
| puntoInicial | int |  |
| puntoFinal | int |  |
| General activities necessary to obtain the results | 1. Recorrer el laberinto con le método BFS 2. Obtener el camino hacia la salida | | |
| Result or postcondition | 1. El camino de salida del laberinto | | |
| Outputs | Output name | Datatype | Selection or repetition condition |
| caminoSalida | Graph |  |

| Name or identifier | **R4**: Buscar el camino con el algoritmo DFS | | |
| --- | --- | --- | --- |
| Resume | El sistema permite utilizar este método para recorrer todos los nodos del laberinto y buscar las posibles salidas. | | |
| Inputs | Input name | Datatype | Selection or repetition condition |
| puntoInicial | int |  |
| puntoFinal | int |  |
| General activities necessary to obtain the results | 1. Recorrer el laberinto con le método DFS 2. Obtener el camino hacia la salida | | |
| Result or postcondition | 1. El camino de salida del laberinto | | |
| Outputs | Output name | Datatype | Selection or repetition condition |
| caminoSalida | Grafo |  |

| Name or identifier | **R5**: Mostrar los caminos posibles dentro del laberinto | | |
| --- | --- | --- | --- |
| Resume | El sistema debe permitir buscar el camino por donde pasó para encontrar la salida, ya sea con el DFS y BFS. Esto se deberá mostrar en un interfaz de manera que el camino de solución para laberinto se marque con un color. | | |
| Inputs |  |  |  |
| General activities necessary to obtain the results | 1. Obtener el camino recorrido 2. Mostrar el camino y la solución al laberinto. | | |
| Result or postcondition | 1. El camino de solución del laberinto | | |
| Outputs | Output name | Datatype | Selection or repetition condition |
| Camino | Grafo |  |

**Fase 2: Recopilación de Información**

Para la solución de este problema, abordaremos los conceptos y herramientas de programación que nos van a permitir generar el laberinto y recorrerlo. A continuación, abordaremos estos para la implementación del sistema:

**Teoría de grafos:**

Es una rama de la matemática y las ciencias de la computación que estudia las propiedades de los grafos**.** Además, analiza y resuelve los problema relacionados con los grafos.

**Grafo:**

Un gráfico G = (V, E) es una estructura matemática que consta de dos conjuntos V y E. Los elementos de V se llaman vértices (o nodos), y los elementos de E se llaman aristas. Cada arista tiene asociado un conjunto de uno o dos vértices, que son llamados sus extremos.

**terminología**: Se dice que una arista une sus extremos. Un vértice unido por una arista a un vértice V, Se dice que el vértice v es vecino de v.

**self-loop:** es un artista que une un solo punto final a sí mismo.

**multi-edge:** es una colección de dos o más aristas que tienen extremos idénticos.La multiplicidad de aristas es el número de aristas dentro de la multi-arista.

**Grafo simple:** no tiene bucles propios ni aristas múltiples.

**Gráfico sin bucles (o multi-gráfico):** puede tener varias aristas pero no bucles automáticos.

**Gráfico (general):** puede tener bucles automáticos y/o múltiples aristas.

**Algoritmos de recorridos:**

Son técnicas utilizadas para visitar todos los nodos o aristas de un grafo. Estos algoritmos permiten explorar y buscar información dentro de la estructura del grafo

**Breadth-first search:** La búsqueda por amplitud es uno de los algoritmos más simples para buscar un gráfico y el arquetipo de muchos algoritmos de gráficos importantes. La búsqueda en amplitud se denomina así porque expande la frontera entre los vértices descubiertos y no descubiertos de manera uniforme a lo largo de la anchura de la frontera. Puede pensar en ello como descubrir vértices en ondas que emanan del vértice de origen. Es decir, a partir de s, el algoritmo primero descubre todos los vecinos de s, que tienen distancia 1. Luego descubre todos los vértices con distancia 2, luego todos los vértices con distancia 3, y así sucesivamente, hasta que haya descubierto todos los vértices alcanzables desde s

**Depth First Search:** Como su nombre lo indica, la búsqueda en profundidad busca <más profundamente en el gráfico siempre que sea posible. La búsqueda en profundidad explora los bordes fuera del vértice v descubierto más recientemente que todavía tiene bordes sin explorar que lo abandonan. Una vez que se han explorado todas las aristas de v, la búsqueda <backtracks= para explorar las aristas que salen del vértice desde el cual se descubrió v. Este proceso continúa hasta que se han descubierto todos los vértices a los que se puede acceder desde el vértice de origen original.Si quedan vértices no descubiertos, la primera búsqueda en profundidad selecciona uno de ellos como una nueva fuente, repitiendo la búsqueda desde esa fuente. El algoritmo repite todo este proceso hasta que ha descubierto cada vértice.

**Shortest path algorithms:**

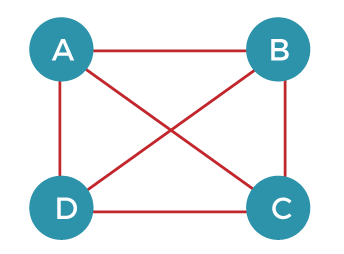
Es el problema que consiste en encontrar un camino entre dos vértices o nodos, de tal manera que la suma de los pesos de las aristas que lo constituyen sea mínima. Al camino más corto entre dos vértices también se le conoce como geodésica.

**The Floyd-Warshall algorithm:** El algoritmo de Floyd es muy similar, pero trabaja con grafos ponderados. Es decir, el valor de la “flecha” que representamos en la matriz puede ser cualquier número real o infinito. Infinito marca que no existe unión entre los nodos. Esta vez, el resultado será una matriz donde estarán representadas las distancias mínimas entre nodos, seleccionando los caminos más convenientes según su ponderación (“peso”).

**El algoritmo de Dijkstra:**

El algoritmo de Dijkstra es utilizado en la teoría de grafos para encontrar el camino más corto entre un nodo de origen y todos los demás nodo dentro del grafo ponderado. La funcionalidad principal del algoritmo es determinar el camino más corto desde un nodo de origen dado hacia todos los demás nodos de un grafo ponderado no dirigido. Aunque también se puede utilizar en un grafo dirigido pero realmente no es tan común encontrarlo. El algoritmo de Dijkstra no admite aristas con pesos negativos, sin embargo, funciona eficientemente en grafos con pesos no negativos y es especialmente útil cuando se trata de encontrar la ruta más corta en redes de comunicación o sistemas de navegación, como este es el caso lo sería en el laberinto.

**Minimum Spanning Trees:**

****

El gráfico anterior se puede representar como G(V, E), donde 'V' es el número de vértices y 'E' es el número de aristas. El árbol de expansión del gráfico anterior se representaría como G`(V`, E`). En este caso, V` = V significa que el número de vértices en el árbol de expansión sería el mismo que el número de vértices en el gráfico, pero el número de aristas sería diferente. El número de aristas en el árbol de expansión es el subconjunto del número de aristas en el gráfico original.

**El algoritmo de Kruskal:**

El algoritmo de Kruskal es un algoritmo utilizado en la teoría de grafos para encontrar un árbol de expansión mínimo en un grafo ponderado no dirigido. Se utiliza ampliamente en los problemas de optimización y diseño de redes.

La funcionalidad principal del algoritmo de Kruskal es encontrar un árbol de expansión mínimo en un grafo ponderado. Un árbol de expansión mínimo es un subconjunto de aristas del grafo que conecta todos los nodos y tiene el menor peso total posible. Este algoritmo funciona de la siguiente manera.

El algoritmo de Kruskal funciona de la siguiente manera:Ordena todas las aristas del grafo por peso, de menor a mayor, luego inicia un árbol vacío como resultado.

como siguiente paso, para cada arista en orden ascendente de peso, si agregarla al árbol no forma un ciclo, se agrega al árbol resultante. Repite el paso 3 hasta que se hayan considerado todas las aristas o se haya construido un árbol de expansión mínimo.

**El algoritmo de Prim:**

El algoritmo es utilizado para encontrar el árbol de expansión mínima en un grafo ponderado no dirigido. El objetivo del algoritmo es construir un subconjunto de aristas que conectan todos los vértices del grafo minimizando la suma de los pesos de las artistas seleccionadas. El algoritmo de Prim funciona de la siguiente manera.

Primero selecciona un vértice inicial arbitrario y lo agrega al árbol de expansión mínima, luego en cada iteración se elige una arista de peso mínimo que conecte un vértice en el árbol con un vértice fuera del árbol; Después agrega la arista seleccionada al árbol de expansión mínima y marca el vértice agregado como visitado. Los dos pasos anteriores se repiten hasta que todos los vértices estén en el árbol de expansión mínima. Por último el árbol de expiación mínima resultante contiene todas las aristas necesarias para conectar todos los vértices del grafo original minimizando el costo total. En pocas palabras, este algoritmo se basa en la idea de ir creciendo el árbol de forma incremental, seleccionando siempre la arista de menor peso que conecte un vértice en el árbol con vértice fuera del árbol.

Tomado de:

* [**Graph Theory and Its Applications, third edition - DOKUMEN.PUB**](https://dokumen.pub/graph-theory-and-its-applications-third-edition.html)
* [**https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf**](https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf)
* [**https://www.javatpoint.com/minimum-spanning-tree-introduction**](https://www.javatpoint.com/minimum-spanning-tree-introduction)

**Fase 3: Búsqueda de Soluciones Creativas**

La generación del laberinto lo vamos a realizar nosotros, por tanto no va a ver un algoritmo específico para generarlo de forma aleatoria, si no que solo vamos agregar vértices al grafos para representar este como nuestro laberinto definido.

Para la búsqueda del camino del laberinto, tenemos las siguientes alternativas:

**Alternativa 1: Búsqueda por BFS**

El algoritmo de búsqueda en anchura es un enfoque común para resolver laberintos. Lo podemos implementar para encontrar la ruta más corta desde el punto inicial hasta el punto final en el laberinto. El algoritmo BFS explorará el laberinto nivel por nivel, asegurándose de que se encuentre la solución más corta. Podemos implementarlo de la siguiente manera:

* Cola de nodos: El algoritmo BFS utiliza una estructura de datos llamada "cola" para almacenar los nodos que se explorarán. Cada nodo representa una celda del laberinto. Comienza en el nodo correspondiente al punto inicial y agrégalo a la cola.
* Expansión de nodos: Mientras la cola no esté vacía, extrae el nodo frontal de la cola y examina sus vecinos. En el contexto del laberinto, los vecinos son las celdas adyacentes (arriba, abajo, izquierda, derecha) que no son paredes. Si se encuentra el punto final, has encontrado la solución y podemos detener la búsqueda.
* Marcar y encolar: A medida que examinas los vecinos de un nodo, marca cada vecino como visitado y agrégalo a la cola si no ha sido visitado antes. Esto asegurará que el algoritmo explore el laberinto en anchura, nivel por nivel.
* Ruta más corta: Durante la expansión de nodos, podemos mantener un registro de los nodos padres para cada nodo visitado. Esto nos permite reconstruir la ruta más corta desde el punto inicial hasta el punto final una vez que lo encuentres.
* Visualización: Mientras el algoritmo BFS está en progreso, el usuario puede visualizar el progreso del agente a medida que explora el laberinto. Se mostrará el laberinto en una interfaz gráfica y resaltar las celdas visitadas y el camino actualmente explorado.

**Alternativa 2: Búsqueda por DFS**

También el algoritmo de búsqueda en profundidad es adecuado para explorar laberintos y encontrar soluciones. Se puede utilizar para que el agente explore el laberinto. Donde el algoritmo para que el agente avance a través de los pasillos y retroceda cuando llegue a un punto muerto. Es decir, este algoritmo lo que hace es buscar por profundidad. También, como el algoritmo BFS, muestra visualmente el progreso del agente en el laberinto.

* Este algoritmo irá buscando los vértices, cuando haya llegado a un vértice, lo que hace es buscar su vecino, este proceso lo seguirá hasta que se encuentre con una pared, si lo hace, este se devuelve el vértice que ha sido visitado completamente y seguirá con otro no haya sido visitado. Cuando encuentras el punto final, se encuentra la solución y se detiene la búsqueda.
* Marcar vértice: A medida que examinas los vecinos de un vértice , marca cada vecino como visitado. Como habíamos dicho anteriormente, esto asegurará que el algoritmo explora el laberinto en profundidad, avanzando lo más lejos posible a lo largo de un camino antes de retroceder.

**Alternativa 3: Búsqueda por The Floyd-Warshall algorithm**

Otra forma de solucionar el laberinto es implementar el algoritmo de Floyd-Warshall, en donde se encuentran los caminos más cortos entre todos los pares de nodos en un grafo ponderado. Aunque su aplicación principal es en la resolución de problemas de rutas más cortas en grafos. Se podría solucionar de la siguiente manera:

* Construcción del grafo: Utilizando la representación del laberinto, se construirá un grafo ponderado donde los vértices representan las celdas del laberinto y las aristas tienen un peso asociado. Las aristas representarán las conexiones entre las celdas del laberinto y su peso dependerá de si hay un pasillo o una pared entre ellas. Por ejemplo, se tendría que asignar un peso bajo a los pasillos y un peso alto a las paredes.
* Aplicación del algoritmo de Floyd-Warshall: Este algoritmo encontrará las rutas más cortas entre todos los pares de vértices en el grafo. Después de ejecutar el algoritmo, se obtendrá una matriz de distancias que indica la longitud de la ruta más corta entre cada par de vértices.
* Búsqueda de la ruta entre los puntos inicial y final: Utilizando la matriz de distancias obtenida del paso anterior, se determinaría la ruta más corta entre los puntos inicial y final.

**Alternativa 4: Búsqueda por el algoritmo de Dijkstra**

Otra solución a nuestro problema, sería usar este algoritmo, en donde se encontrará la ruta más corta desde el punto inicial donde ese será el vértice de origen a todos los demás vértices en un grafo ponderado, hasta llegar al punto final.

* Construcción del grafo: Utilizando la representación del laberinto, se construirá un grafo ponderado donde los vértices representan las celdas del laberinto y las aristas tienen un peso asociado. Las aristas representarán las conexiones entre las celdas del laberinto y su peso dependerá de si hay un pasillo o una pared entre ellas. Se asignará un peso bajo a los pasillos y un peso alto a las paredes.
* Aplicación del algoritmo de Dijkstra: Aplica el algoritmo de Dijkstra al grafo construido, tomando como vértice de origen el punto inicial del laberinto. El algoritmo de Dijkstra calculará la ruta más corta desde el punto inicial a todos los demás vértices del grafo. Durante la ejecución del algoritmo, se mantendrá un registro de los vértices padres para cada nodo visitado.
* Búsqueda de la ruta entre los puntos inicial y final: Utiliza los vértices padres registrados durante la ejecución del algoritmo de Dijkstra para determinar la ruta más corta desde el punto inicial hasta el punto final. Comienza en el punto final y sigue los vértices padres hasta llegar al punto inicial, construyendo la ruta en sentido inverso.

La solución es similar a la anterior, solo que el primero es aplicado con respecto a una matriz mientras que el algoritmo de Dijkstra está en una lista de adyacencia.

**Alternativa 5: Algoritmo de búsqueda en profundidad limitada (Iterative Deepening Depth-First Search, IDDFS)**

Investigando un poco más encontramos este algoritmo que también es de búsqueda que es una variante del algoritmo de búsqueda en profundidad (DFS) que evita algunos de los problemas asociados con DFS, como la posibilidad de caer en bucles infinitos en grafos con ciclos.

El IDDFS utiliza una estrategia iterativa para realizar una búsqueda en profundidad con una profundidad máxima cada vez mayor en cada iteración. Comienza con una profundidad límite de 0 y se incrementa en cada iteración hasta encontrar el objetivo o explorar todo el espacio de búsqueda.

* Algoritmo IDDFS:Implementa el algoritmo de búsqueda en profundidad limitada (IDDFS) para resolver el laberinto. Inicializa una profundidad límite en 0 y establece una bandera para indicar si se ha encontrado el objetivo. Realiza un bucle principal para iterar sobre las diferentes profundidades límite.
* En cada iteración: El algoritmo realiza una búsqueda en profundidad limitada dentro del laberinto utilizando el algoritmo de búsqueda en profundidad. Durante esa búsqueda, lleva un registro de los Vértices visitados y la ruta seguida hasta el momento.Si se encuentra el punto final, establece la bandera de objetivo encontrado en verdadero y almacena la ruta como solución.Incrementa la profundidad límite en 1 en cada iteración hasta que se encuentre el objetivo o se haya explorado todo el espacio de búsqueda.
* Resultados:Si se encuentra el objetivo, muestra la ruta encontrada al usuario como solución.Si no se encuentra el objetivo, muestra los caminos posibles o un mensaje indicando que no se pudo completar la misión.Esta solución utilizando el algoritmo IDDFS permitirá generar el Laberinto de Creta y resolverlo utilizando una búsqueda en profundidad limitada. El algoritmo explorará gradualmente el laberinto en busca de la solución, evitando problemas como bucles infinitos y utilizando una cantidad razonable de memoria.

**Fase 4: Transición de la formulación de ideas a los diseños preliminares.**

Como nuestro objetivo principal es solo encontrar los caminos de salida del laberinto comparando los algoritmos para ver cual es el más eficiente, no incluiremos pesos en las aristas. Por lo tanto descartamos las alternativas 3 y 4 ; ya que estas solo funcionan con pesos. Eventualmente la alternativa 5 podría servirnos pero es un algoritmo que aún no hemos visto en clase por lo tanto sería una investigación aparte que se podría realizar pero nos llevaría un poco de tiempo, así que colocaremos las razones de por qué no deberíamos utilizarlo pero la pasaremos a la siguiente fase para evaluarla más a profundidad , debido a lo cual estas son nuestras razones:

1. Los algoritmos de Floyd-Warshall y Dijkstra están diseñados para resolver problemas en grafos ponderados, donde las aristas tienen pesos asociados. Sin embargo, si no estamos considerando pesos en las aristas del laberinto, aplicar estos algoritmos puede ser innecesario y agregar complejidad adicional a la solución.
2. Ambos algoritmos son más complejos en términos de tiempo de ejecución y uso de recursos en comparación con otros enfoques más simples como la búsqueda en anchura (BFS) o la búsqueda en profundidad (DFS).
3. El IDDFS tiene una complejidad temporal alta, ya que realiza múltiples iteraciones de búsqueda en profundidad limitada con incrementos graduales en la profundidad límite. Esto puede llevar a un tiempo de ejecución largo, sí hiciéramos un código el cual generará laberintos al azar para los laberintos especialmente de gran tamaño. Debido a que el IDDFS realiza múltiples iteraciones de búsqueda en profundidad limitada, requiere un uso significativo de recursos, como memoria y capacidad de procesamiento. Esto puede ser problemático en entornos con recursos limitados o cuando se trabaja con laberintos muy grandes.

Aunque el IDDFS garantiza encontrar una solución si existe dentro de la profundidad límite, no garantiza encontrar la solución más corta u óptima. Es posible que encuentre una solución, pero podría haber una ruta más corta que no se exploró debido a las limitaciones de la profundidad límite.

De acuerdo al objetivo puede ser más eficiente utilizar un enfoque más simple como el BFS y DFS en lugar de las alternativas que estamos descartando.

**Fase 5: Evaluación de la mejor solución.**

De acuerdo a nuestro planteamiento la idea es buscar la salida del laberinto comparando la eficiencia de los dos algoritmos los cuales encuentran un camino solo que de maneras distintas.

Las razones por las cuales elegimos estos dos algoritmo son los siguientes:

* Encontrar una ruta factible: Para el Laberinto de Creta sin pesos en las aristas, el objetivo principal es encontrar una ruta factible desde el punto inicial hasta el punto final. BFS y DFS están diseñados para encontrar una ruta válida en un grafo sin considerar los pesos de las aristas. Estos enfoques simples pueden garantizar que se encuentre una ruta si existe, sin tener en cuenta la longitud de la ruta o si es la ruta más corta.
* Tiempo de ejecución: Tanto BFS como DFS tienen un tiempo de ejecución lineal en relación con el número de celdas del laberinto. Esto significa que a medida que el laberinto se vuelve más grande, el tiempo requerido para encontrar una ruta desde el punto inicial al punto final aumenta de manera predecible. En comparación, algoritmos como Floyd-Warshall y Dijkstra tienen una mayor complejidad temporal, lo que puede resultar en un mayor tiempo de ejecución cuando se aplican a laberintos grandes.
* Uso de recursos: BFS y DFS son algoritmos que no requieren tanto espacio de almacenamiento adicional. Mientras que Floyd-Warshall y Dijkstra necesitan estructuras de datos como matrices y listas adicionales para almacenar información sobre las distancias y los nodos visitados, BFS y DFS solo requieren una estructura de datos básica para almacenar la información del recorrido del laberinto.
* Implementación y simplicidad: BFS y DFS son algoritmos más sencillos de implementar y comprender en comparación con Floyd-Warshall y Dijkstra. No requieren conceptos adicionales como matrices de distancias o cálculos de pesos de rutas más cortas.

Por tanto, para solucionar nuestro problema, utilizaremos los enfoques más simples como BFS y DFS para que el Laberinto de Creta sin pesos en las aristas pueda ser más eficiente en términos de tiempo de ejecución y uso de recursos y cumpliría con nuestro objetivo.

Sin embargo aún evaluamos una opción posible que es la del algoritmo El algoritmo de búsqueda en profundidad limitada (IDDFS), por las siguientes razones:

* Compleción garantizada: El IDDFS garantiza encontrar una solución si existe dentro de la profundidad límite establecida. Esto significa que, a medida que se incrementa gradualmente la profundidad límite en cada iteración, eventualmente se encontrará una solución si existe en el laberinto. Esto ofrece la seguridad de que se encontrará una solución, lo cual es especialmente importante en casos donde la existencia de una solución es crucial.
* Exploración en profundidad: El IDDFS sigue una estrategia de exploración en profundidad, lo que significa que se adentra en una rama del laberinto lo más lejos posible antes de retroceder. Esto nos puede ayudar en laberintos donde las soluciones se encuentran en ramas más profundas, ya que se explorarán primero estas ramas antes de retroceder y explorar otras opciones.
* Implementación sencilla: El IDDFS es relativamente fácil de implementar en comparación con otros algoritmos más complejos. Utiliza las estructuras de datos básicas, como pilas o recursión, lo que facilita su implementación y comprensión..