PROYECTO RATÓN Y QUESO

Tina María Torres - 2259729

Marlon Astudillo Muñoz - 2259462

Juan José Gallego Calderón - 2259433

Juan José Hernández Arenas - 2259500

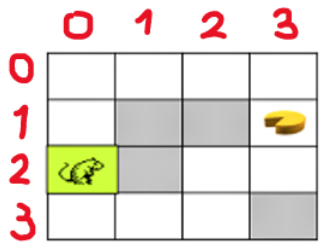
Universidad Del Valle

Ingeniería De Sistemas

Inteligencia Artificial

Tuluá, Valle Del Cauca

2024



Amplitud

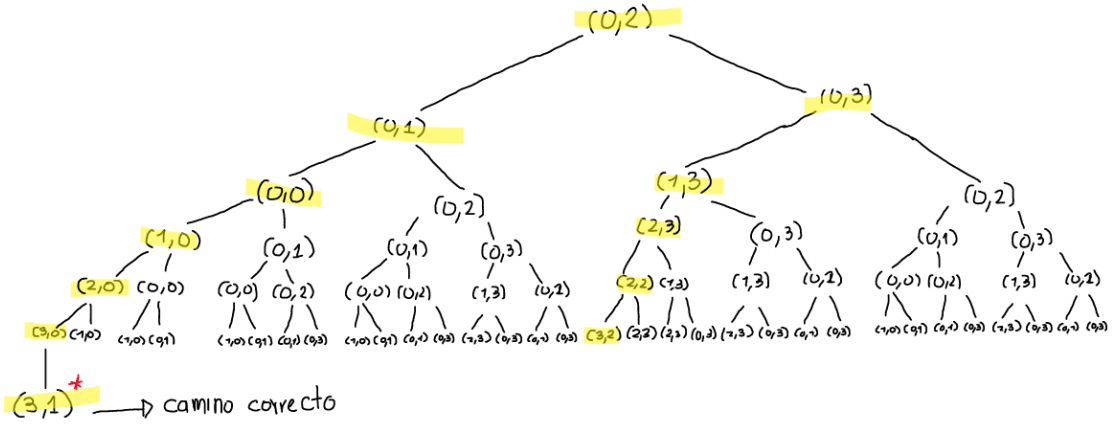
La búsqueda se lleva a cabo expandiendo los nodos del árbol, de manera que explore primero todos los nodos de un nivel antes de pasar al siguiente, es decir, que se recorrería de la siguiente manera…

#1 en (0,2) -> (0,1) -> (0,3)

#2 en (0,1) -> (0,0) -> (0,2) y en (0,3) -> (1,3) -> (0,2)

#3 …

Árbol solucionado



El programa se compone de varias funciones que, en conjunto, permiten:

1. Identificar los movimientos válidos en el laberinto según el ejercicio.
2. Determinar si se ha alcanzado la meta (el queso).
3. Realizar la expansión de nodos utilizando un enfoque de búsqueda por amplitud.

Estructuras y Librerías Utilizadas

- **Deque (from collections import deque)**: Se utiliza una cola (FIFO) para implementar la búsqueda en amplitud, ya que esta estructura es eficiente para insertar y eliminar elementos desde ambos extremos.
- **Clase Nodo (from arbol import Nodo)**: La clase Nodo define la estructura de cada nodo del árbol, con atributos como valor (posición en el laberinto), id, costo, padre, e hijos.
- **Lista bidimensional (laberinto)**: Representa el laberinto, donde un valor de 0 indica un espacio libre y 1 indica un obstáculo.

- **Tuplas y listas**: Se utilizan para manejar las posiciones y los posibles movimientos dentro del laberinto.

Explicación de las funciones

obtener_movimientos_validos(pos_actual):

- ★ Esta función toma como entrada la posición actual (x, y) del ratón.
- ★ Genera una lista de posiciones posibles hacia donde el ratón puede moverse: arriba, derecha, abajo e izquierda.
- ★ Solo se consideran movimientos dentro de los límites del laberinto y hacia espacios libres (0).
- ★ Devuelve una lista con las posiciones válidas a las que se puede mover.

es_meta(pos_actual):

- ★ Verifica si la posición actual coincide con la posición del queso (meta).
- ★ Retorna True si el ratón ha llegado al queso, y False en caso contrario.

amplitud(árbol, idn):

- ★ Esta función implementa el algoritmo de búsqueda por amplitud.
- ★ Utiliza una cola (deque) para manejar los nodos que deben ser expandidos, comenzando por la raíz del árbol.

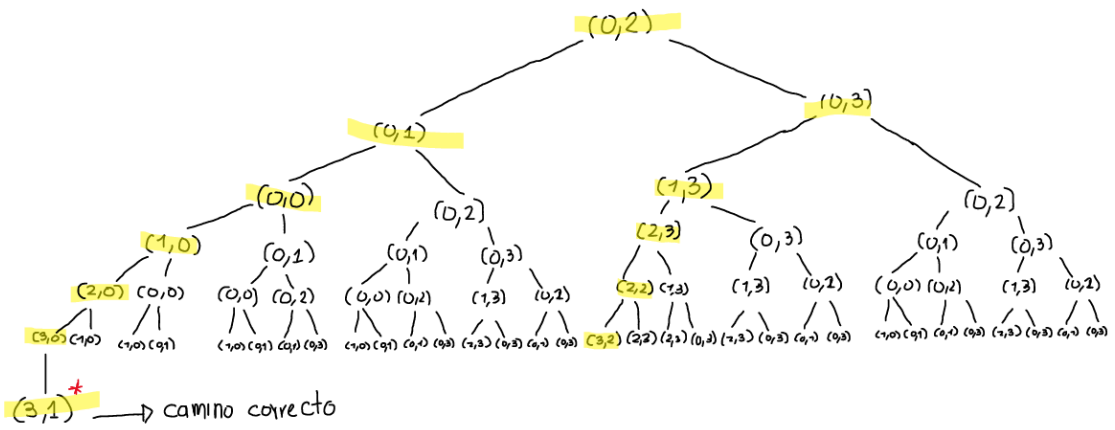
Proceso

- o Extrae el primer nodo de la cola y verifica si es la meta.
- o Si el nodo actual tiene hijos, los agrega a la cola para su futura expansión.
- o Si el nodo no tiene hijos, se obtienen todos los movimientos válidos desde su posición actual.
- o Para cada movimiento válido, se crea un nuevo nodo que representa la nueva posición del ratón y se añade como hijo al nodo expandido.
- o La función retorna el árbol y un valor True o False que indica si se alcanzó la meta.

Costo Uniforme

La búsqueda de costo uniforme se asemeja a un algoritmo de búsqueda en amplitud, pero se enfoca en expandir los nodos de menor costo acumulado primero, como estamos usando costo 1 posee mucha más libertad para resolverse.

Árbol solucionado



El programa está compuesto por varias funciones que permiten:

1. Calcular el costo acumulado de un nodo desde la raíz.
2. Identificar todas las hojas del árbol y sus respectivos costos.
3. Determinar los movimientos válidos en el laberinto.
4. Realizar la expansión de nodos según el enfoque de costo uniforme.

Estructuras y Librerías Utilizadas

- **Clase Nodo (from arbol import Nodo)**: Representa cada nodo del árbol, con atributos como valor (posición en el laberinto), id, costo, padre, e hijos.
- **Lista bidimensional (laberinto)**: Representa el laberinto, donde un 0 indica un espacio libre y un 1 indica un obstáculo.
- **Tuplas y listas**: Se utilizan para gestionar las posiciones y los movimientos dentro del laberinto.

Funciones utilizadas

calcular_costo_acumulado(nodo):

★ Esta función calcula el costo total para llegar a un nodo determinado desde la raíz.
★ Utiliza un enfoque recursivo para sumar el costo del nodo actual al costo acumulado de su padre, hasta llegar a la raíz.
★ Si el nodo es la raíz (no tiene padre), devuelve su propio costo.

obtener_hojas_con_costos(nodo):

★ Esta función identifica todas las hojas del árbol y calcula sus costos acumulados utilizando la función calcular_costo_acumulado.
★ Si un nodo no tiene hijos, se considera una hoja y se agrega a la lista hojas junto con su costo acumulado.
★ Si el nodo tiene hijos, la función se llama recursivamente para cada hijo, acumulando los resultados en una lista.
★ Devuelve una lista de tuplas donde cada tupla contiene un nodo hoja y su costo acumulado.

obtener_movimientos_validos(pos_actual):

- ★ Determina los movimientos válidos que se pueden realizar desde una posición (x, y) dentro del laberinto.
- ★ Verifica que las posiciones propuestas estén dentro de los límites del laberinto y sean espacios libres (0).
- ★ Retorna una lista con las posiciones válidas a las que se puede mover el ratón.

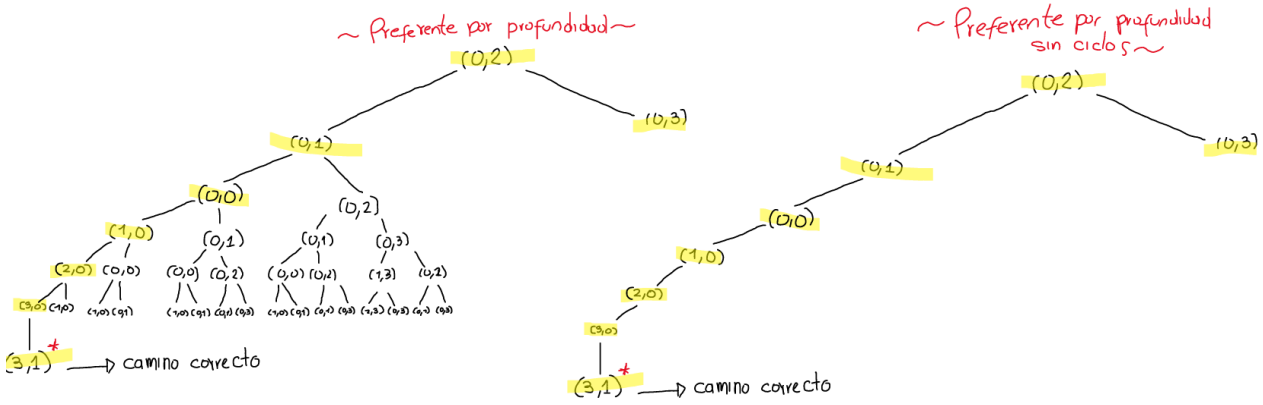
costo_uniforme(arbol, idn)

- ★ Implementa el algoritmo de búsqueda de costo uniforme.
- ★ Primero, obtiene todas las hojas del árbol junto con sus costos acumulados usando obtener_hojas_con_costos.
- ★ Identifica la hoja con el menor costo acumulado utilizando la función min y la expande.
- ★ Si la posición del nodo seleccionado coincide con la del queso, retorna el árbol y True indicando que se alcanzó la meta.
- ★ En caso contrario, genera nodos hijos para cada movimiento válido desde la posición actual, aumentando el costo en 1 unidad por cada movimiento.
- ★ Finalmente, retorna el árbol actualizado y False si aún no se ha encontrado la meta.

Búsqueda por profundidad

El objetivo es encontrar un camino desde una posición inicial hasta una meta (el queso) expandiendo primero los nodos más profundos.

Árbol solucionado



El código se compone de varias funciones que permiten:

1. Identificar el nodo hoja más profundo en un árbol.
2. Determinar los movimientos válidos dentro del laberinto.
3. Implementar la expansión de nodos siguiendo el enfoque de búsqueda preferente por profundidad.

Estructuras y Librerías Utilizadas

- **Clase Nodo (from arbol import Nodo)**: Cada nodo en el árbol representa una posición en el laberinto con atributos como valor (posición), id, costo, padre, e hijos.

- **Lista bidimensional (laberinto)**: Representa el laberinto, donde 0 indica un espacio libre y 1 un obstáculo.
- **Tuplas y listas**: Se utilizan para manejar posiciones y movimientos dentro del laberinto.
- **Pilas**: Se usan para simular el comportamiento del algoritmo DFS.

Explicación de las funciones

obtener_nodo_hoja_mas_profundo(nodo)

- ★ Esta función busca el nodo hoja más profundo en un árbol utilizando un enfoque basado en pilas (simulando DFS).
- ★ La pila contiene tuplas con el nodo actual y su profundidad.
- ★ Si un nodo no tiene hijos, se considera una hoja y se agrega a la lista hojas con su profundidad.
- ★ Si el nodo tiene hijos, estos se añaden a la pila en orden inverso para que se priorice el nodo más a la izquierda.
- ★ Una vez que todas las hojas se han encontrado, se selecciona la hoja con la mayor profundidad usando max.
- ★ Resultado: Devuelve el nodo hoja más profundo encontrado.

obtener_movimientos_validos(pos_actual)

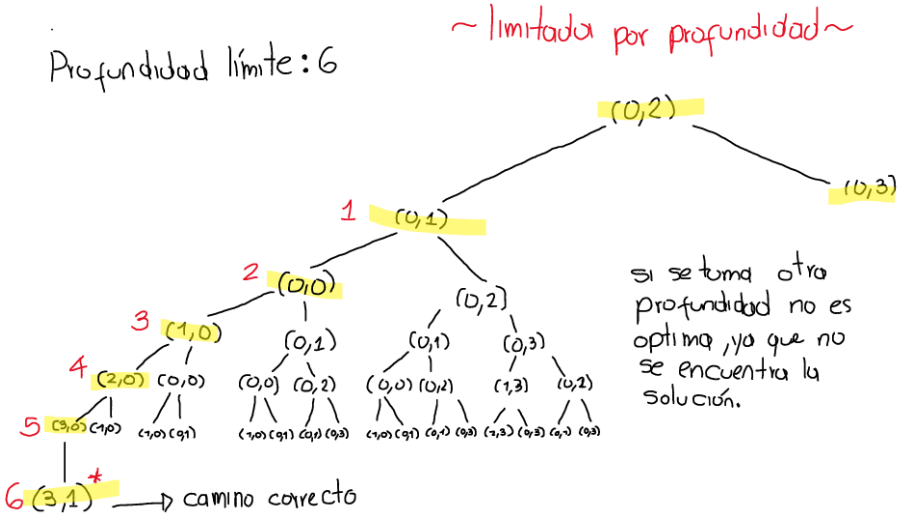
- ★ Esta función determina los movimientos válidos desde una posición (x, y) en el laberinto.
- ★ Se consideran las cuatro direcciones posibles (arriba, derecha, abajo, izquierda).
- ★ Verifica que los movimientos propuestos estén dentro de los límites del laberinto y sean espacios libres (0).
- ★ Resultado: Retorna una lista con las posiciones válidas a las que se puede mover.

preferente_por_profundidad(arbol, idn)

- ★ Implementa el algoritmo de búsqueda preferente por profundidad (DFS).
- ★ Primero, encuentra la hoja más profunda utilizando obtener_nodo_hoja_mas_profundo.
- ★ Si la posición del nodo coincide con la del queso, retorna True indicando que se ha alcanzado la meta.
- ★ Si no se ha encontrado el queso, expande el nodo añadiendo hijos para cada movimiento válido desde la posición actual.
- ★ Cada movimiento tiene un costo fijo de 1.
- ★ Resultado: Retorna el árbol expandido y False si aún no se ha encontrado el queso.

Búsqueda limitada por profundidad

A diferencia del enfoque clásico de búsqueda en profundidad, esta variante establece un límite máximo de profundidad para la expansión de nodos.



El algoritmo incluye:

1. La identificación de nodos hojas dentro de un límite de profundidad.
2. La expansión de nodos en función de los movimientos válidos desde una posición dada en el laberinto.
3. Una verificación para determinar si se ha alcanzado el objetivo (el queso).

Estructuras y Librerías Utilizadas

- **Clase Nodo (from arbol import Nodo)**: Cada nodo en el árbol representa una posición en el laberinto, con atributos como valor, id, costo, padre, e hijos.
- **Lista bidimensional (laberinto)**: Representa el laberinto donde 0 indica un espacio libre y 1 un obstáculo.
- **Tuplas y listas**: Utilizadas para manejar posiciones y movimientos en el laberinto.
- **Pilas**: Empleadas para realizar un recorrido en profundidad.

Explicación de las Funciones

obtener_movimientos_validos(pos_actual)

- ★ Esta función genera todos los movimientos posibles desde una posición dada (x, y) en el laberinto.
- ★ Filtra aquellos movimientos que están dentro de los límites y no se topan con obstáculos.
- ★ Resultado: Retorna una lista con las coordenadas válidas a las que se puede mover.

obtener_nodo_hoja_mas_profundo_limite(nodo, limite)

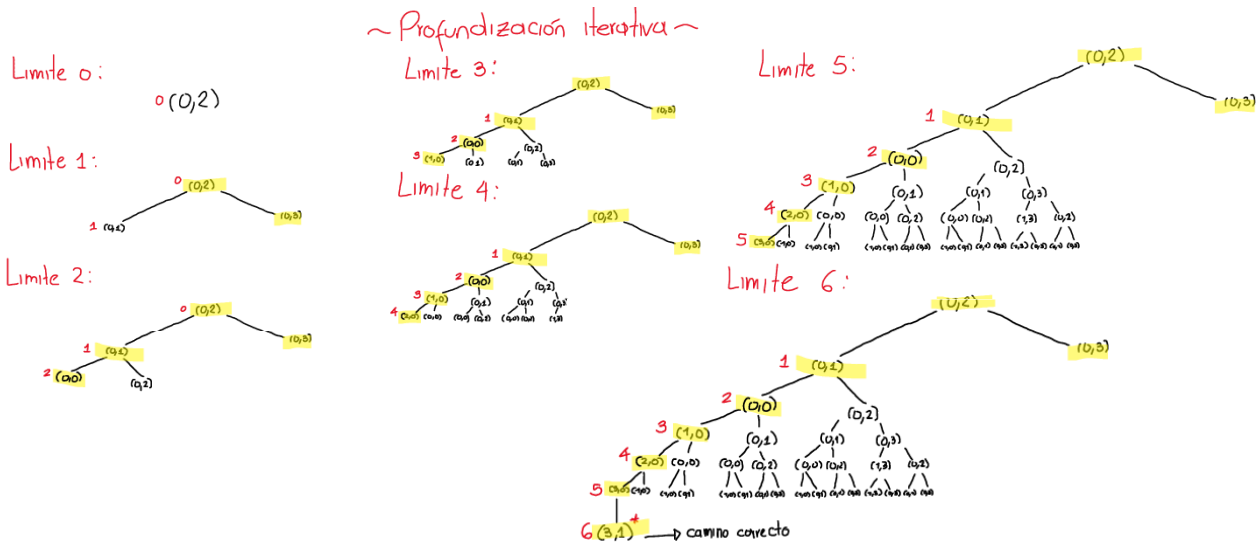
- ★ Utiliza un enfoque basado en pilas para realizar una búsqueda en profundidad, pero con un límite establecido (límite).
- ★ Agrega nodos a la pila para su exploración sólo si su profundidad es menor que el límite.
- ★ Si encuentra nodos hoja, selecciona la hoja más profunda dentro del límite.
- ★ Resultado: Devuelve el nodo hoja más profundo encontrado dentro del límite de profundidad o None si no se encuentra ninguno.

profundidad_limitada(arbol, limite, idn)

- ★ Esta función implementa el algoritmo de búsqueda en profundidad limitada.
- ★ Obtiene la hoja más profunda que aún está dentro del límite de profundidad usando obtener_nodo_hoja_mas_profundo_limite.
- ★ Si la hoja encontrada corresponde al queso, el algoritmo termina y retorna el árbol y True.
- ★ Si no se encuentra el queso, expande el nodo agregando hijos para cada movimiento válido desde la posición actual.
- ★ Resultado: Retorna el árbol expandido y False si aún no se ha encontrado el queso.

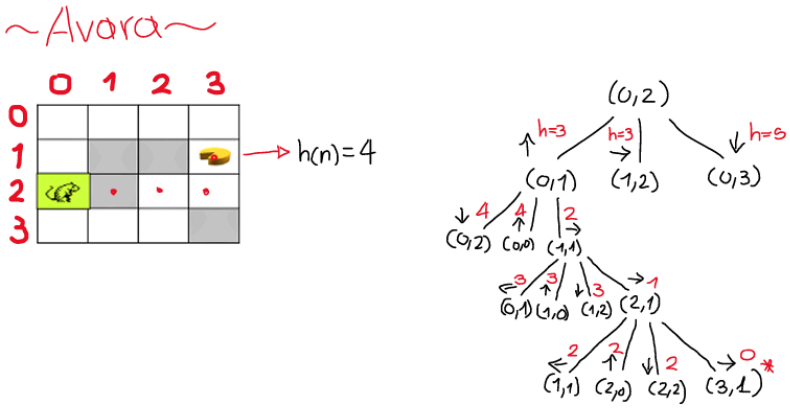
Profundidad iterativa

Es la misma profundidad limitada, solo que aumentamos el límite en +1. Esto se realiza desde la misma main.



Busqueda avara

A diferencia de los enfoques anteriores, este algoritmo utiliza una **función heurística** para priorizar la expansión de nodos, eligiendo aquellos que parecen estar más cerca de la meta (el queso) según una estimación.



Pasos generales:

1. Se obtienen todas las hojas del árbol y se selecciona la hoja con el valor más bajo de la función heurística (en este caso, la distancia de Manhattan al queso).
2. Si la hoja seleccionada corresponde a la posición del queso, se ha encontrado la solución.
3. Si no, se expanden los movimientos válidos desde la posición actual y se calculan los valores heurísticos para cada nuevo nodo hijo.
4. El árbol se actualiza y el proceso continúa hasta encontrar el queso o agotarse las opciones.

Estructuras y Librerías Utilizadas

abs(): Esta función es parte de la librería estándar de Python y se utiliza para calcular la distancia de Manhattan en el cálculo de la heurística entre la posición actual y la meta.

eval(): Esta es una función integrada de Python que se utiliza para evaluar una cadena como una expresión de Python, en este caso, para convertir el valor del nodo (que está representado como una cadena en formato de tupla, como "(x, y)") en una tupla real de Python.

Explicación de las Funciones

obtener_hojas_con_heuristica(nodo)

- ★ Recorre el árbol recursivamente y obtiene todas las hojas junto con sus valores de heurística
- ★ Devuelve una lista de tuplas (nodo, heurística).

obtener_movimientos_validos(pos_actual)

- ★ Genera todos los movimientos posibles (arriba, derecha, abajo, izquierda) desde una posición (x, y) dada.
- ★ Filtra los movimientos que están dentro del laberinto y no colisionan con obstáculos (1).
- ★ Resultado: Devuelve una lista de posiciones válidas a las que se puede mover.

calcular_heuristica(pos_actual, meta)

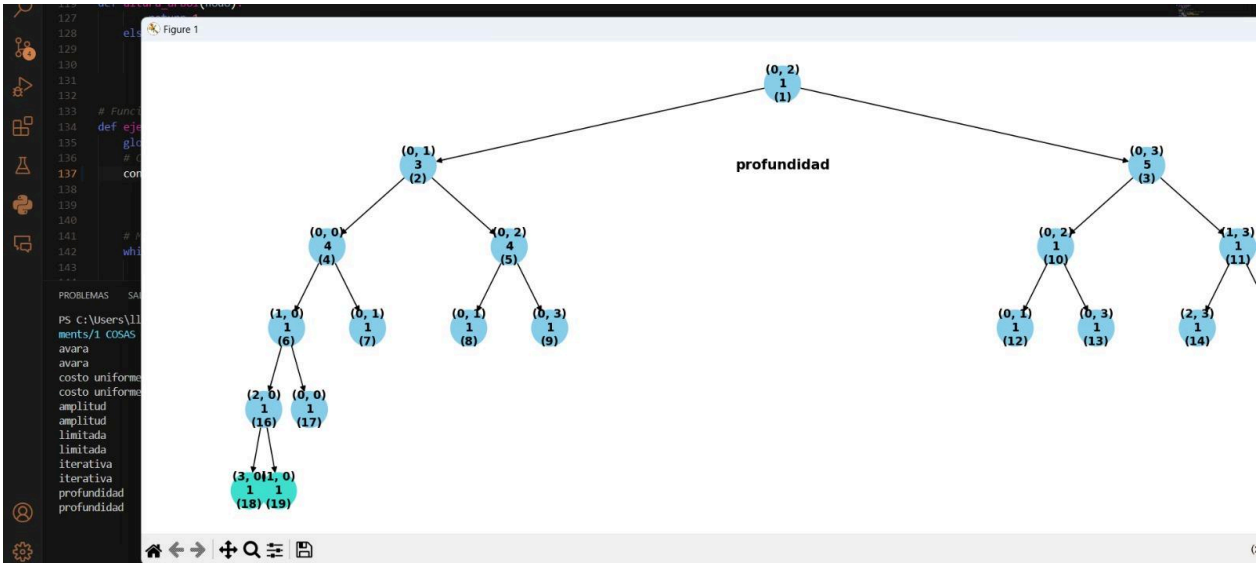
- ★ Calcula la distancia de Manhattan entre la posición actual (x1, y1) y la meta (x2, y2).
- ★ Esta distancia es adecuada para un entorno cuadriculado como el laberinto.
- ★ Resultado: Devuelve un valor entero que representa la distancia estimada.

busqueda_avara(arbol, idn)

- ★ Obtiene todas las hojas del árbol junto con sus valores heurísticos.
- ★ Selecciona la hoja con el valor de heurística más bajo (es decir, la que está más cerca del queso según la estimación).
- ★ Si la posición del nodo a expandir coincide con la posición del queso, se ha alcanzado la meta y se retorna True.
- ★ Si no, expande los movimientos válidos desde la posición actual, creando nuevos nodos y asignándoles sus respectivos valores heurísticos.
- ★ Resultado: Devuelve el árbol actualizado y False si aún no se ha alcanzado el objetivo.

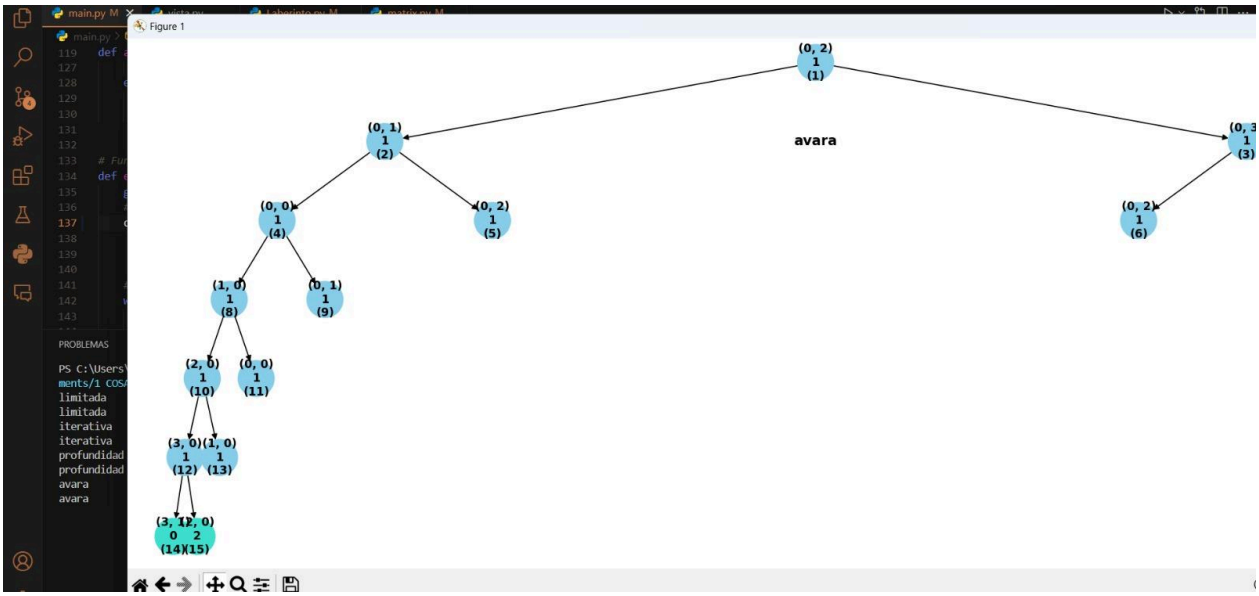
Pruebas

Prueba #1



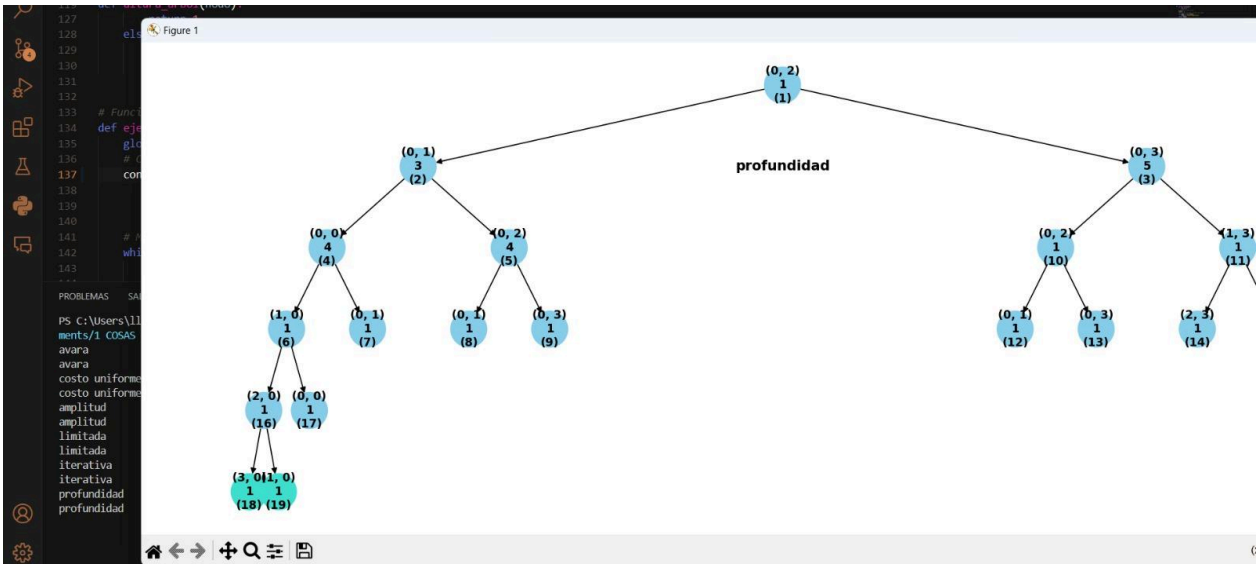
Cuando el número de expansiones es bajo, las diferencias entre los distintos métodos de búsqueda, como profundidad iterativa, profundidad limitada y preferente, son prácticamente imperceptibles. Esto ocurre porque la cantidad de nodos explorados es tan reducida que las particularidades de cada técnica no llegan a influir significativamente en el resultado o en el proceso de búsqueda. En estas condiciones, todos los métodos tienden a comportarse de manera similar, ya que el problema no requiere un análisis exhaustivo del espacio de estados ni una gestión compleja de los nodos. Por lo tanto, las ventajas o desventajas propias de cada enfoque no se manifiestan de manera clara cuando las expansiones necesarias son mínimas.

Prueba #2



Cuando el algoritmo emplea los métodos de búsqueda en profundidad, ya sea iterativa, limitada o preferente, junto con el enfoque de búsqueda avara, logra encontrar la solución de manera eficiente en el caso del laberinto representado en la imagen, siempre y cuando se respete el orden de expansión de los nodos. Esto se debe a que, independientemente del método específico utilizado, las características del laberinto, combinadas con el orden en el que se evalúan los movimientos posibles, permiten que el algoritmo converja rápidamente hacia la meta. La búsqueda avara, en particular, destaca al guiarse por la heurística que prioriza los movimientos más prometedores, mientras que los métodos de profundidad garantizan una exploración estructurada. En este caso específico, la estructura del laberinto favorece que todos los enfoques lleguen a la solución de manera directa, mostrando una convergencia natural y eficaz.

Prueba #3



Cuando el algoritmo de búsqueda avara toma el control, rápidamente selecciona el camino más corto basado en la heurística, guiando al proceso hacia la solución de manera eficiente y ágil. Esto se debe a que, al priorizar los movimientos con la menor heurística, avara enfoca su búsqueda hacia la meta sin realizar exploraciones innecesarias. Sin embargo, cuando el control se transfiere nuevamente a las estrategias de búsqueda no informadas, el proceso puede volverse ineficiente. Al no contar con una heurística que oriente la expansión de los nodos, estos métodos tienden a explorar áreas menos prometedoras del espacio de búsqueda, lo que puede generar una disminución en la eficiencia y llevar a un camino más largo hacia la solución. En este punto, el comportamiento del algoritmo se vuelve menos predecible y más lento, lo que afecta negativamente la rapidez con que se encuentra la solución.

Conclusión pruebas

En términos generales, y observando el rendimiento promedio, las búsquedas implementadas permiten encontrar la solución en una gran cantidad de escenarios, demostrando su efectividad en diferentes contextos. Sin embargo, a pesar de su rendimiento satisfactorio, hay espacio para optimizar la combinación de los métodos de búsqueda utilizados. Una posible mejora sería la eliminación de algunas

de las estrategias menos eficientes, como la búsqueda de profundidad limitada e iterativa, que en ciertos casos no contribuyen significativamente a la eficiencia general del algoritmo. Al refinar la selección y combinación de los métodos de búsqueda, se podría lograr una mayor rapidez y precisión en la obtención de soluciones, optimizando el proceso sin sacrificar la efectividad. Esto podría incluir el enfoque de explorar de manera más eficiente la profundidad o incluso considerar otras estrategias más avanzadas para combinar las búsquedas de manera más efectiva.

Nota: para ejecutar el programa se requiere instalar las librerías: customtkinter, networkx, matplotlib. Y ejecutar el archivo vista.py