

Pacman IA

Cabal Correa Andres Felipe - 2160339

Cuestas Parada Daniel Jose - 2067550

Moreno Gil Johan Alejandro - 2160052

Facultad de Ingeniería, Universidad del Valle - Sede Tuluá

Introducción a la Inteligencia Artificial

Mtr. Triana Madrid Joshua David

Octubre 2024

Índice

Resumen	3
Introducción	4
Objetivo General	4
Objetivos específicos:	4
Marco Teórico	5
Agentes de Software	5
Métodos de búsqueda	5
Búsqueda no Informada	5
Búsqueda Informada	6
Metodología	7
Desarrollo	8
Nodos	11
Agentes	13
René	14
Método deep_search():	14
Bucle Principal	15
Construcción de Ruta:	15
Piggy	17
Algoritmos de búsqueda de Piggy	20
1. A*(A Star)	20
2. Amplitud(BFS)	24
Resultados	29
Referencias	34

Resumen

Se desarrolló un programa en el lenguaje de programación Python en el que se aplicaron los conceptos fundamentales de IA vistos en el curso con el objetivo de programar comportamientos específicos a Agentes de software permitiéndoles cumplir tareas tales como la búsqueda de elementos en el entorno ambiente, haciendo uso de las técnicas de búsqueda informada y no informada.

Introducción

Los modelos de inteligencia artificial clásicos establecieron las bases para el desarrollo de tecnologías modernas en el campo de la IA, para la apropiación de estos conceptos se plantea el desarrollo de un juego en el cual mediante del uso de modelos de búsqueda informada y no informada se pretende simular el recorrido de dos agentes dentro del laberinto, cada uno tiene como objetivo encontrar un elemento dentro del laberinto.

Objetivo General

Desarrollar un programa en Python donde la rana Rene (Agente 1) juegue solo y encuentre a su amigo Elmo mientras que Piggy (Agente 2) busca a Rene.

Objetivos específicos:

- Desarrollar un agente para la rana René, de tal forma que siga el método de búsqueda limitada por profundidad hasta encontrar a Elmo.
- Desarrollar un agente para Piggy, de tal forma que siga el método de búsqueda por Amplitud o A* hasta encontrar a René.
- Desarrollar una interfaz gráfica donde se pueda visualizar el movimiento de cada agente.

Marco Teórico

Agentes de Software

Los agentes de software son modelos informáticos que funcionan como entidades autónomas, a los cuales se les asigna un objetivo o tarea específica la cual completan mediante la interacción con su entorno o ambiente, en el cual se pueden encontrar diferentes recursos e incluso a otros agentes cuya interacción es asimilada por sensores que transforman las señales recibidas en datos, los cuales serán recopilados con el fin de tomar decisiones según el caso presentado. (*AGENTES SOFTWARE | Oficina De Transferencia De Resultados De Investigación*, n.d.)

Métodos de búsqueda

Con el objetivo de dar solución a un problema definido surge la necesidad de una secuencia de pasos a seguir para progresivamente recorrer el camino que da solución al problema que acontece, esto se relaciona directamente con el concepto previo de Agente puesto que según lo establecido anteriormente, los agentes tienen como objetivo dar cumplir una tarea especificada. La sinergia entre estos conceptos da como resultado la posibilidad de calcular con anterioridad la acción a ejecutar por el agente para secuencialmente alcanzar su objetivo. (*Métodos De Búsqueda IA*, n.d.)

Los modelos de búsqueda usan la información recopilada por el agente para crear un árbol de estados, cada estado corresponde a la información recopilada por el agente en cada posible movimiento, esto hace que al recorrer el árbol en alguna de sus ramas se llegue a un estado final en el que el agente complete su objetivo, lo que deja la puerta abierta para obtener la secuencia de pasos desde la rama en la que se encontró el estado final o meta.

Búsqueda no Informada

Los modelos que corresponden a esta clasificación se caracterizan porque no poseen información acerca de la cantidad de pasos hasta el objetivo o no cuentan con el valor del costo asociado a ejecutar cada acción disponible dentro del agente (*Métodos De Búsqueda IA*, n.d.). De los métodos de búsqueda no informada encontramos:

- **Preferente por amplitud:** Este método establece analizar secuencialmente los estados (nodos) que se encuentran en una profundidad d antes que los estados de una profundidad $d + 1$. (*Métodos De Búsqueda IA*, n.d.)
- **Preferente por profundidad limitada:** Este método se caracteriza por analizar el primer estado de cada profundidad hasta llegar a la última profundidad o al límite previamente definido. En algunos casos el método puede quedar en un bucle infinito del cual sería imposible salir si no hay un límite de iteraciones que le permita analizar un nodo de una rama diferente fuera del bucle. Al alcanzar el límite se analiza al nodo siguiente en la profundidad que se encuentre o el nodo hermano del nodo previo al que se está analizando si la rama ha sido analizada completamente.

Búsqueda Informada

A diferencia de los métodos de búsqueda no informada, los métodos bajo esta clasificación cuentan con una Heurística la cuál es una función que retorna información adicional que permite el cálculo de una solución más óptima para cumplir el objetivo del agente. Bajo esta clasificación encontramos

- **A*:** Este método utiliza una función de costo f compuesta a su vez por una heurística h y una función g que representa el costo de desplazarse desde el nodo inicial al nodo actual. En este método se analizan los nodos cuyo costo f sea el menor de su altura dentro del árbol revisando previamente nodos de alturas previas con costos menores.

Metodología

Para el desarrollo de este proyecto se identificaron tres módulos principales que componen al proyecto en su totalidad:

1. Módulo gráfico
2. Módulo del Agente Rene y el algoritmo de búsqueda asociado
3. Módulo del Agente Piggy y los algoritmos de búsqueda relacionados.

A cada integrante se le asignó una sección del proyecto, las cuales fueron desarrolladas de forma independiente con el fin de avanzar rápidamente en la creación del mismo. El desarrollo del proyecto fue realizado en el lenguaje de programación Python esto con el fin de satisfacer el requerimiento establecido en el enunciado del proyecto, además se hizo uso de diferentes librerías proveídas por el lenguaje tales como:

- Pygame: para el desarrollo del módulo gráfico del proyecto.
- Collections: para el manejo de la estructuras de datos requeridas tales como Colas y Pilas.
- Time: para controlar el tiempo de refresco de la interfaz gráfica.
- ABC: para el uso de herencia en las clases creadas para el proyecto.

El uso de un repositorio fue indispensable para agilizar el desarrollo, puesto que con el uso del mismo se tenían centralizados los trabajos hechos por cada integrante y facilitó la posterior integración de los módulos desarrollados. Para ello, se usó la plataforma GitHub para alojar el repositorio empleado.

Posterior al trabajo individual de cada persona, se agendó una reunión virtual en la que se discutió acerca de la implementación de cada módulo desarrollado y cómo realizar la unión de estos.

Desarrollo

Módulo Gráfico

El módulo gráfico, desarrollado con Pygame, es el componente encargado de visualizar la interfaz de usuario y el entorno en el que los agentes (René y Piggy) interactúan dentro del laberinto. Esta implementación se enfoca en hacer visible el proceso de búsqueda de objetivos mediante animaciones y elementos interactivos. A continuación, se explican detalladamente sus partes:

1. Inicialización del Entorno Gráfico

Para dar inicio a la interfaz gráfica, se define una ventana de juego con las siguientes configuraciones:

- Dimensiones: La ventana del juego tiene un ancho de 500 píxeles y una altura de 400 píxeles, las cuales son asignadas a constantes (ANCHO y ALTO).
- Colores: Se establecen dos colores principales: NEGRO (0, 0, 0) para el fondo y BLANCO (255, 255, 255) para los textos y botones.
- Ventana y Título: La ventana se crea con pygame.display.set_mode((ANCHO, ALTO)) y se le asigna el título “Pacman versión Univalle” con pygame.display.set_caption().

Estas configuraciones son la base para el resto de las interacciones visuales y facilitan el uso de una paleta de colores y dimensiones fijas en el diseño de la interfaz.



```

1 ANCHO = 500
2 ALTO = 400
3 NEGRO = (0, 0, 0)
4 BLANCO = (255, 255, 255)
5 VENTANA = pygame.display.set_mode((ANCHO, ALTO))
6
7
8 pygame.display.set_caption('Pacman versión Univalle')
9
10 laberinto = Laberinto(ANCHO, ALTO)

```

2. Pantalla de Bienvenida (welcome())

La función welcome() es responsable de mostrar una pantalla inicial, interactiva, que permite al usuario elegir el tipo de laberinto y comenzar el juego. La función se estructura en los siguientes pasos:

- Carga de Imagen de Fondo: Se carga una imagen de bienvenida , la cual es escalada al tamaño de la ventana usando pygame.transform.scale(fondo, (ANCHO, ALTO)).
- Texto de Laberinto Seleccionado: Se define un mensaje predeterminado que mostrará información sobre el laberinto seleccionado, cambiando cuando el usuario presiona las teclas 1 a 4, o 0:
 - Tecla 1: Asigna el laberinto Maze.MAZE_A y actualiza el mensaje a “Laberinto A: el laberinto del Enunciado”.
 - Tecla 2: Asigna el laberinto Maze.MAZE_B y muestra “Laberinto B: Rene encuentra a Elmo”.
 - Tecla 3: Asigna el laberinto Maze.MAZE_C y muestra “Laberinto C: Piggy encuentra a Rene”.
 - Tecla 4: Asigna el laberinto Maze.MAZE_D y muestra “Laberinto D: Rene atascado”.
 - Tecla 0: Establece un laberinto aleatorio y mantiene el mensaje “Laberinto Random”.

Este selector permite que el usuario elija el escenario específico de búsqueda antes de iniciar el juego, y el mensaje se muestra dinámicamente para brindar claridad sobre la elección.

- Botón de "Jugar": Se configura un botón interactivo que cambia de color cuando el puntero del mouse se coloca sobre él y devuelve **True** cuando se hace clic, iniciando así el juego.
- La función welcome(): finaliza al capturar el evento de clic en el botón, iniciando el juego en caso de una selección exitosa, y mantiene el bucle de espera si el usuario no ha seleccionado nada.

```

● ○ ●
1 def welcome():
2     fondo = pygame.image.load('images/fondo_bienvenida.png')
3     fondo = pygame.transform.scale(fondo, (ANCHO, ALTO))
4
5     text_label = "Laberinto Random"
6     play_button = pygame.Rect(ANCHO // 2 - 50, ALTO // 2 - 25, 100, 50)
7
8     while True:
9         keys = pygame.key.get_pressed()
10
11         if keys[pygame.K_1]:
12             laberinto.set_mapa(Maze.MAZE_A)
13             text_label = "Laberinto A: el laberinto del Enunciado"
14         elif keys[pygame.K_2]:
15             laberinto.set_mapa(Maze.MAZE_B)
16             text_label = "Laberinto B: Rene encuentra a Elmo"
17         elif keys[pygame.K_3]:
18             text_label = "Laberinto C: Piggy encuentra a Rene"
19             laberinto.set_mapa(Maze.MAZE_C)
20         elif keys[pygame.K_4]:
21             text_label = "Laberinto D: Rene atascado"
22             laberinto.set_mapa(Maze.MAZE_D)
23         elif keys[pygame.K_0]:
24             text_label = "Laberinto Random"
25             laberinto.set_mapa(None)

```

3. Pantalla de Fin del Juego (end_game())

La función end_game() muestra una pantalla final al terminar el juego, indicando al usuario que el juego ha concluido. Las características clave son:

- Fondo de Pantalla Final: Se carga una imagen que cubre toda la ventana, creando un ambiente de “Game Over”.
- Mensaje Personalizado: Se renderiza un mensaje de texto centrado con el contenido “El juego ha terminado” o un mensaje personalizado, usando una fuente de tamaño 40 en color blanco.

Esta pantalla se mantiene hasta que el usuario decida cerrar la ventana. Esto asegura que el usuario tenga una retroalimentación visual clara al concluir la partida.

```

1 def end_game(text="El juego ha terminado"):
2     fondo = pygame.image.load('images/game-over.jpeg')
3     fondo = pygame.transform.scale(fondo, (ANCHO, ALTO))
4     font = pygame.font.Font(None, 40)
5     text_surface = font.render(text, True, (255, 255, 255))
6
7     text_x = (ANCHO - text_surface.get_width()) // 2
8     text_y = ALTO - text_surface.get_height() - 10
9
10    background_rect = pygame.Rect(
11        text_x - 10, text_y - 5, text_surface.get_width() + 20, text_surface.get_height() + 10)
12
13    while True:
14        for evento in pygame.event.get():
15            if evento.type == pygame.QUIT:
16                pygame.quit()
17                return False
18
19        VENTANA.blit(fondo, (0, 0))
20        pygame.draw.rect(VENTANA, (0, 0, 0), background_rect)
21
22        VENTANA.blit(text_surface, (text_x, text_y))
23
24        pygame.display.flip()

```

Nodos

Los nodos son la representación de la información primordial que es usada por las estrategias de búsqueda en cada punto de iteración del programa, cada nodo está representado por una clase *Node* que recibe como parámetros:

- Position: Una tupla que representa la posición del agente en el momento de la creación del nodo.
- Parent: Un nodo que corresponde a un estado previo al estado actual es decir un nodo padre del nodo actual. Este valor puede ser nulo en caso del nodo inicial o raíz del cual los algoritmos parten para sus respectivos cálculos.
- Costo (g): Valor numérico correspondiente al costo acumulado de los desplazamientos previos del agente para llegar al nodo actual.
- Heurística (h): Valor numérico aproximado correspondiente a la distancia desde la posición del nodo actual al objetivo o meta del agente.

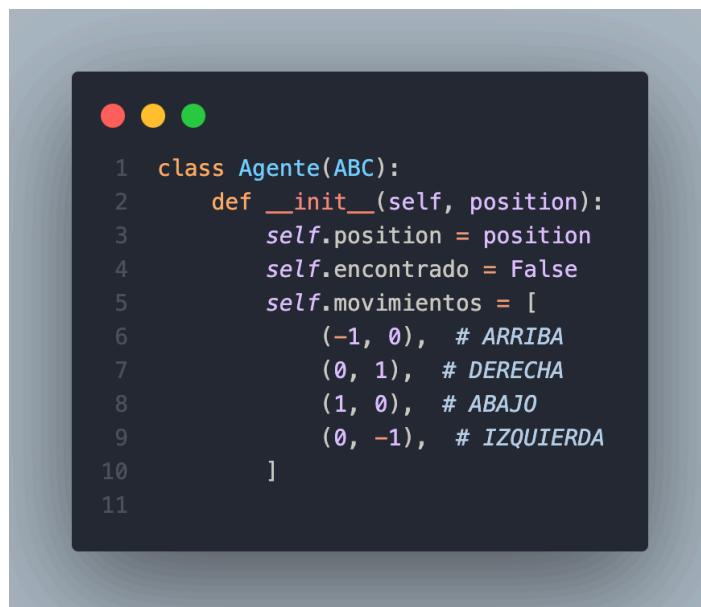
```
● ● ●  
1  class Node:  
2      def __init__(self, position, parent=None, g=0, h=0, nivel=0):  
3          self.position = position  
4          self.parent = parent  
5          self.g = g # Costo desde el inicio  
6          self.h = h # Heurística  
7          self.f = g + h # Costo total  
8          self.nivel = nivel  
9  
10     def __str__(self):  
11         return f"{self.position}"  
12  
13     def __lt__(self, other):  
14         return self.f < other.f  
15  
16     def __repr__(self):  
17         return self.__str__()
```

Agentes

Para el manejo se empleó el concepto de Programación Orientada a Objetos (POO) con el fin de estructurar correctamente los componentes que componen a cada agente e implementar el modelo de reutilización de código propio de la herencia y el polimorfismo.

Como punto de partida se creó una clase abstracta llamada Agente, en esta se definen las propiedades que poseen los mismos las cuales son:

- Posición: representa las coordenadas del agente en el laberinto que está recorriendo.
- Encontrado: Una bandera booleana que se activa cuando el agente encuentra su respectivo objetivo.
- Movimientos: Un arreglo de valores que representan las acciones que puede realizar cada agente, en este caso, los agentes se pueden desplazar en las direcciones: Arriba, Derecha, Abajo, Izquierda.



```

● ● ●

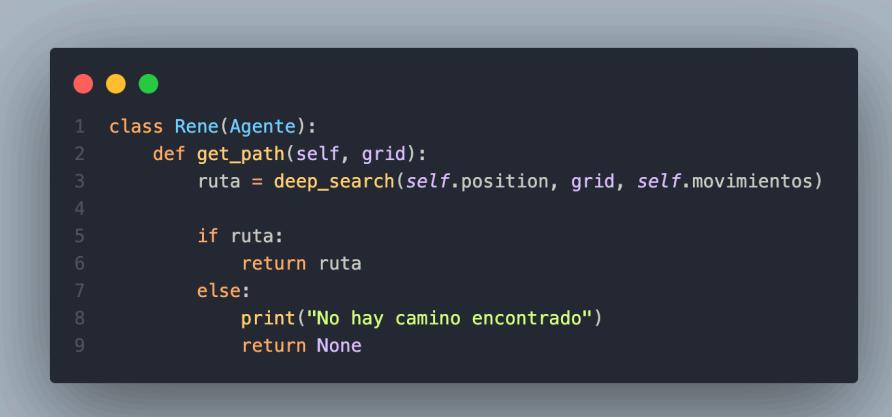
1  class Agente(ABC):
2      def __init__(self, position):
3          self.position = position
4          self.encontrado = False
5          self.movimientos = [
6              (-1, 0), # ARRIBA
7              (0, 1), # DERECHA
8              (1, 0), # ABAJO
9              (0, -1), # IZQUIERDA
10         ]
11

```

A partir de la superclase agente se desarrollaron clases hijas para los agentes especificados en los requisitos.

René

René es el agente que tiene como objetivo encontrar dentro del laberinto a Elmo, para ello se implementó la estrategia de búsqueda por Profundidad Limitada. Para la implementación del mismo se creó una clase que hereda las propiedades especificadas en la superclase Agente y adicionalmente se desarrolló el método *get_path()* el cual retorna la ruta a seguir por el agente dentro del laberinto hasta llegar a su objetivo.



```

1  class Rene(Agente):
2      def get_path(self, grid):
3          ruta = deep_search(self.position, grid, self.movimientos)
4
5          if ruta:
6              return ruta
7          else:
8              print("No hay camino encontrado")
9              return None

```

El método *get_path()* recibe como parámetro el grid, que corresponde a la matriz que representa el laberinto que debe recorrer el agente, este es usado posteriormente por el método *deep_search()* que se encarga de calcular la ruta del agente mediante el algoritmo de búsqueda por Profundidad Limitada. En base al resultado del algoritmo de búsqueda se retorna el camino a seguir o se hace el print indicando que no se encontró la ruta.

Método *deep_search()*:

Como se explicó anteriormente este método corresponde al implementación de la estrategia de búsqueda por Profundidad Limitada, este algoritmo recibe 3 parámetros:

- *position*: Corresponde a un objeto de tipo *tuple* el cual representa la posición desde la que se realizará la búsqueda del objetivo.
- *grid*: Corresponde a una matriz que representa el estado actual del laberinto.
- *movimientos*: Un arreglo de tuplas que sirve como constantes que al operarse en conjunto con la posición del agente producen un desplazamiento en las direcciones Arriba, Derecha, Abajo, Izquierda.

Bucle Principal

Para la realización de la búsqueda se declara un stack que contiene los nodos que deben ser expandidos, esta lista se va a iterar siempre y cuando existan nodos pendientes por analizar. El criterio para agregar un nodo a la lista está dado por dos condiciones

- Movimiento Válido: El agente tiene una lista de movimientos los cuales son ejecutados uno a uno con el objetivo de determinar si a la casilla que se llega es una casilla válida, es decir, si la casilla no corresponde a un obstáculo, en caso de cumplir esta condición se crea una Instancia de la clase nodo asignando la información de la casilla encontrada para ser agregada a la lista de nodos por expandir.
- Límite: La estrategia empleada cuenta con un límite que indica el nivel de profundidad máximo al que se puede llegar mientras se está ejecutando el algoritmo. Para este caso está dado por la cantidad de casillas que contiene el laberinto, esto hace que en caso de encontrarse en un bucle de movimientos, el algoritmo pueda salir del mismo si ya ha realizado una cantidad de movimientos que concuerde con la cantidad de casillas del algoritmo.

En cada iteración del bucle antes de calcular los nodos generados por los posibles movimientos del agente, se verifica si este se encuentra en la casilla que corresponde a su objetivo. En caso de que el agente ya esté en la casilla objetivo se para la ejecución del bucle y se pasa a la construcción de la ruta a seguir por el agente.

Construcción de Ruta:

Para la construcción del camino a seguir por el agente hasta su objetivo se recibe el último nodo expandido que debe corresponder al que representa el objetivo del agente, en base a este agente se define un arreglo al que iterativamente se agrega el nodo padre del nodo final, y a ese nodo se realiza el mismo procedimiento lo que se traduce en agregar los nodos previos que se encuentran en la misma rama del nodo objetivo hasta llegar al nodo inicial o raíz.



```

● ● ●

1 def construir_ruta(nodo: Node):
2     ruta = []
3     while nodo:
4         ruta.append(nodo.position)
5         nodo = nodo.parent
6     return ruta[::-1]

```

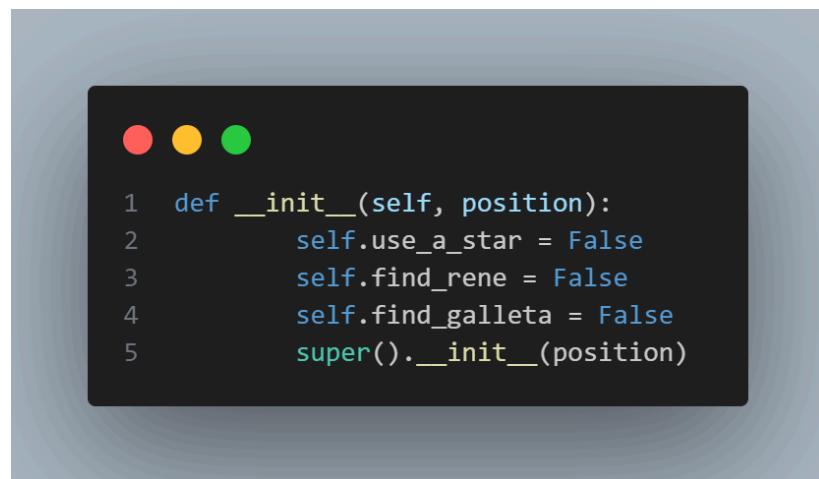
Posterior al respectivo cálculo de ruta, se envía el camino generado a la clase que corresponde al agente para ser aplicado a la interfaz visual del programa.

```
● ● ●
1 def deep_search(position: tuple, grid, movimientos):
2     stack = deque([Node(position)])
3     filas, columnas = len(grid), len(grid[0])
4     limit = filas * columnas - 1
5
6     while stack:
7         current_node: Node = stack.popleft()
8
9         coordenadas = current_node.position
10        iteracion = current_node.nivel
11
12        try:
13            cell = grid[coordenadas[0]][coordenadas[1]]
14        except IndexError:
15            print("Error: Coordenadas fuera de rango")
16
17        if cell == "E":
18            print("¡Encontré a Elmo!, construyendo ruta...")
19            return construir_ruta(current_node)
20
21        if iteracion < limit:
22            for movimiento in movimientos[::-1]:
23                x = coordenadas[0] + movimiento[0]
24                y = coordenadas[1] + movimiento[1]
25
26                # Chequear si las coordenadas son válidas y no son un bloque
27                x_valido = 0 <= x < filas
28                y_valido = 0 <= y < columnas
29
30                if x_valido and y_valido:
31                    nueva_celda = grid[x][y]
32                    if nueva_celda != 1:
33                        # Agregar a la pila para explorar en la siguiente iteración
34                        new_node = Node((x, y), current_node,
35                                         nivel=iteracion + 1)
36                        stack.appendleft(new_node)
37
38    return None
```

Piggy

Se creó una clase representativa para el agente piggy en la cual tenemos definidos los siguientes métodos:

- **`__init__`**: El método init es un constructor de python, este recibe como parámetro una posición. Se le dan los siguientes atributos para la estancia de la clase piggy.



- **`self.use_a_star`**: define un atributo de la instancia llamado use_a_star y se inicializa con el valor False. Este atributo es un indicador booleano que determina si el objeto usará el método A*.
- **`self.find_rene`**: define un atributo de la instancia llamado find_rene y se inicializa con el valor False. Este atributo es un indicador booleano que determina si el objeto encontró a René.
- **`self.find_galleta`**: define un atributo de la instancia llamado find_galleta y se inicializa con el valor False. Este atributo es un indicador booleano que determina si el objeto encontró a la Galleta.
- **`super().__init__(position)`**: Se hace el llamado al método init de la clase padre y así obtener los atributos de esta.

- **move:** este método modela el movimiento de piggy dentro de un grid (el laberinto) hacia una posición “goal_position” (posición de René). El método realiza los siguientes pasos:

```
● ● ●  
1 def move(self, goal_position, grid):  
2     costo = 1  
3     if self.position == goal_position:  
4         self.find_rene = True  
5         print("¡Piggy ha encontrado a René!")  
6         return self.position  
7  
8     self.use_a_star = random.random() < 0.4  
9  
10    if self.use_a_star:  
11        print("Moviendo a Piggy usando A*")  
12        path = a_star_search(  
13            self.position, goal_position, grid, self.movimientos, self)  
14        self.position = path[1][0]  
15        costo = path[1][1]  
16    else:  
17        print("Moviendo a Piggy usando BFS")  
18        path = bfs(self.position, goal_position,  
19                    grid, self.movimientos, self)  
20        self.position = path[1]  
21  
22    if not path:  
23        print("No hay camino encontrado")  
24        return self.position, costo  
25  
26  
27    try:  
28  
29        if self.position == goal_position:  
30            self.find_rene = True  
31            print("¡Piggy ha encontrado a René!")  
32        elif grid[self.position[0]][self.position[1]] == "G":  
33            print("¡Piggy ha encontrado la galleta!")  
34  
35        return self.position, costo  
36  
37    except IndexError:  
38        print("Error en el movimiento de Piggy, manteniendo posición actual.")  
39        costo = 0  
40        return self.position, costo
```

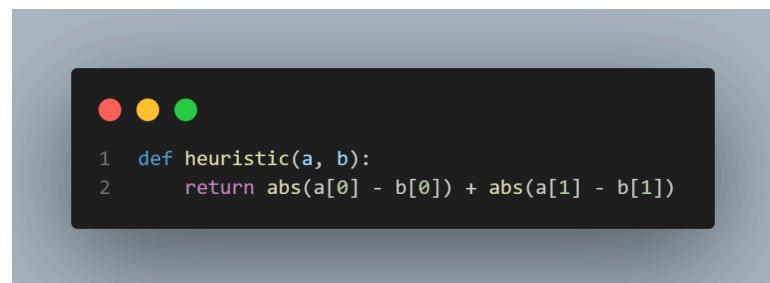
- Válida si Piggy ya encontró a René, si es así imprime el mensaje "¡Piggy ha encontrado a René!" y retorna la posición actual de piggy sin realizar más cálculos.
- Selecciona de forma aleatoria el uso del algoritmo A* donde este tiene una probabilidad de un 40% y el método de amplitud tiene el otro 60%, para esto usa el atributo `use_a_star` cambiando su valor booleano.
- Válida si va a usar o no el algoritmo A* si es así entonces imprime el mensaje "Moviendo a Piggy usando A*" y guarda el camino y el costo que les retorna el método que hace la búsqueda A*. Si no se va a usar el algoritmo A* entonces primero válida si Piggy ya ha encontrado la galleta, si es así el costo del movimiento será 0.5 sino 1 luego imprime el mensaje "Moviendo a Piggy usando BFS" y guarda el camino que le retorna el método que hace la búsqueda por **amplitud**.
- En caso de no hallar algún camino va a imprimir el mensaje "No hay camino encontrado" y va a retornar la posición actual de Piggy y un costo de 0 sin realizar más cálculos.
- si ya ha encontrado la galleta va a darle un costo de 0.5 al movimiento sino el costo del movimiento será de 1.
- Por último intenta actualizar la posición de piggy moviéndose un paso solamente y se valida si en la nueva posición se encuentra René para marcarlo como hallado y retornamos la nueva posición de piggy y el costo del movimiento. Si en el camino de piggy no existe un siguiente paso ocurrirá un IndexError y se imprimirá el mensaje ""Error en el movimiento de Piggy, manteniendo posición actual." y va a retornar la posición actual de Piggy y tendrá un costo de 0.

Algoritmos de búsqueda de Piggy

1. A*(A Star)

Este algoritmo de búsqueda se le aplicó al personaje “Piggy” solamente y se estructuró de la siguiente manera:

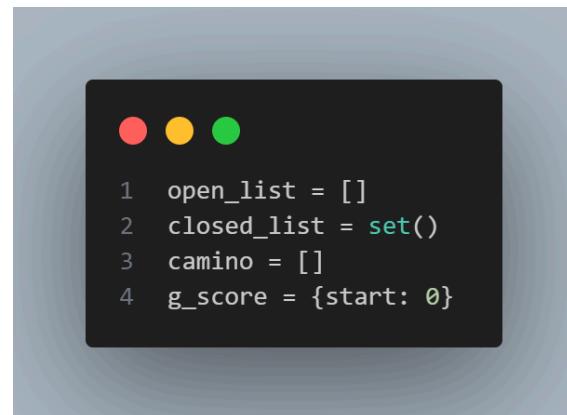
- Se definió primero la heurística a utilizar la cual sería la de Manhattan.



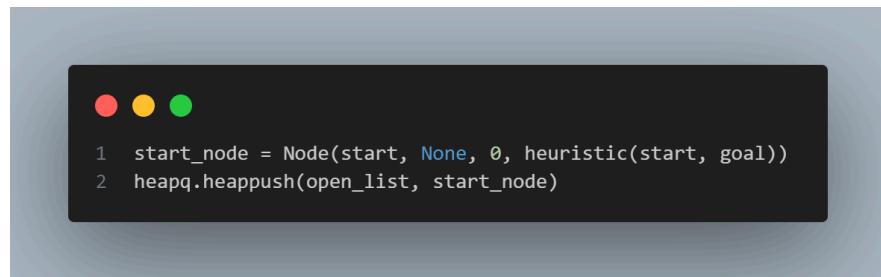
- Los parámetros de la función principal son:
 - **start y goal**: Coordenadas de inicio y objetivo.
 - **Grid**: La cuadrícula de búsqueda.
 - **movimientos**: Lista de movimientos posibles en la cuadrícula (ej., movimientos en las direcciones cardinales).
 - **self**: Referencia al objeto Piggy, usada para actualizar ciertos atributos (como `find_galleta`).



- Se inicializan los elementos a usar
 - **open_list**: Una lista de prioridades que contiene nodos por explorar, ordenados por sus costos estimados ($g + h$).
 - **closed_list**: Un conjunto que mantiene los nodos ya visitados.
 - **g_score**: Diccionario que guarda el costo de los movimientos acumulados desde start a cada nodo alcanzado.



- El nodo de inicio se crea con un costo de movimiento inicial de cero y un valor heurístico calculado hacia el objetivo. Este nodo se agrega a open_list, iniciando el proceso de búsqueda.



Bucle Principal del Algoritmo



```

1  while open_list:
2      current_node = heapq.heappop(open_list)
3      closed_list.add(current_node.position)
4      coordenadas = current_node.position
5
6      try:
7          cell = grid[coordenadas[0]][coordenadas[1]]
8      except IndexError:
9          print("Error: Coordenadas fuera de rango")
10
11     if cell == "R" or current_node.position == goal:
12         path = []
13         while current_node:
14             path.append((current_node.position, current_node.c))
15             current_node = current_node.parent
16
17         camino = path[::-1]
18
19
20     return camino
21
22     if cell == "G":
23         self.find_galleta = 2
24
25     # Explorar vecinos
26     for dx, dy in movimientos:
27
28         neighbor_pos = (
29             current_node.position[0] + dx, current_node.position[1] + dy)
30
31         if neighbor_pos in closed_list:
32             continue
33
34         if (0 <= neighbor_pos[0] < len(grid) and
35             0 <= neighbor_pos[1] < len(grid[0]) and
36             grid[neighbor_pos[0]][neighbor_pos[1]] != 1):
37
38             cell_cost = 1
39             if self.find_galleta > 0:
40                 cell_cost = 0.5
41                 self.find_galleta -= 1
42
43             g = current_node.g + cell_cost
44             h = heuristic(neighbor_pos, goal)
45
46             neighbor_node = Node(neighbor_pos, current_node, cell_cost, g, h)
47
48             if neighbor_pos in g_score and g >= g_score[neighbor_pos]:
49                 continue
50
51             g_score[neighbor_pos] = g
52             heapq.heappush(open_list, neighbor_node)

```

La búsqueda A* opera dentro de un bucle que extrae nodos de `open_list`, comenzando por el de menor costo estimado. Cada nodo extraído se agrega a `closed_list`, lo cual evita que se vuelva a procesar.

Si el nodo actual contiene el objetivo o el símbolo específico (marcado como R en el código), el algoritmo considera el objetivo alcanzado. En este punto, la función construye el camino óptimo desde el nodo actual al de inicio utilizando referencias al nodo padre de cada uno. Este camino se invierte para obtener el orden correcto de los pasos a seguir.

Cálculo del Costo de Movimiento

El código ajusta el costo de movimiento de Piggy en función del estado de ciertos elementos en la cuadrícula:

- Si Piggy encuentra una "galleta", el costo de movimiento se reduce, simulando un beneficio que facilita su desplazamiento. Si no hay beneficios adicionales, se usa el costo normal.

Exploración de Vecinos

El algoritmo evalúa cada movimiento permitido en movimientos para generar posiciones vecinas. Cada vecino potencial debe cumplir las siguientes condiciones:

1. No debe estar en `closed_list`.
2. La posición vecina debe estar dentro de los límites de la cuadrícula y no contener un obstáculo.

Si el vecino cumple estos requisitos, se calcula un nuevo costo de movimiento (g) y una nueva heurística (h). Luego se crea un nodo para el vecino, y se agrega a `open_list` si su costo acumulado es menor que cualquier costo previamente registrado para esa posición.

Retorno de Resultados

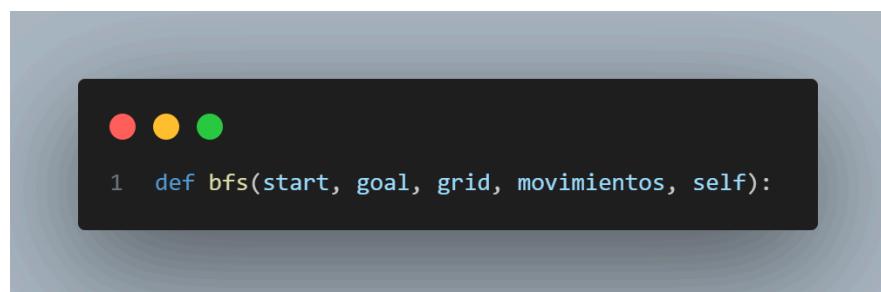
Al final de la búsqueda, la función devuelve el camino y el costo total. Si el camino no existe, el resultado será un camino vacío y un costo predeterminado, lo que indica que no se pudo encontrar una ruta al objetivo.

2. Amplitud(BFS)

El algoritmo de búsqueda en amplitud (BFS, por sus siglas en inglés) es implementado aquí para guiar al personaje "Piggy" hacia un objetivo en una cuadrícula. En este caso, BFS permite a "Piggy" explorar la cuadrícula en orden de cercanía a su posición inicial, asegurando un camino óptimo en términos de pasos necesarios. Este método se adapta a las restricciones de la cuadrícula y permite que "Piggy" identifique elementos clave, como "galletas", que afectan su estado.

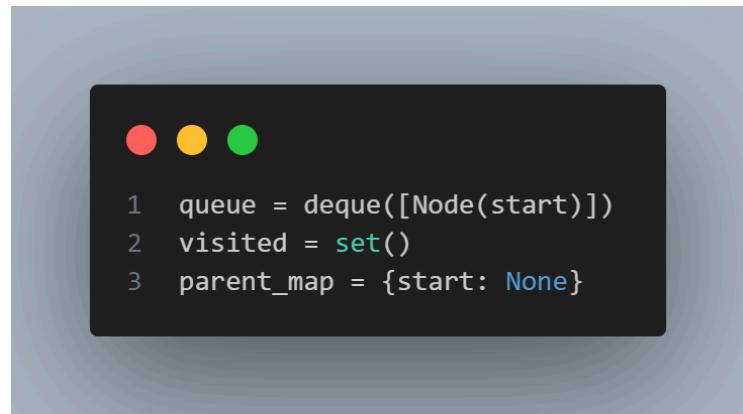
- **Parámetros:**

- **start y goal:** Coordenadas de inicio y objetivo, respectivamente.
- **grid:** Representa la cuadrícula de búsqueda, donde distintos valores indican el tipo de celda (libre, obstáculo o con elementos especiales).
- **movimientos:** Lista de movimientos posibles en las direcciones cardinales, permitiendo a Piggy explorar las posiciones adyacentes.
- **self:** Referencia al objeto Piggy, utilizada para actualizar ciertos atributos, como el descubrimiento de una "galleta" (find_galleta).



- **Elementos Iniciales:**

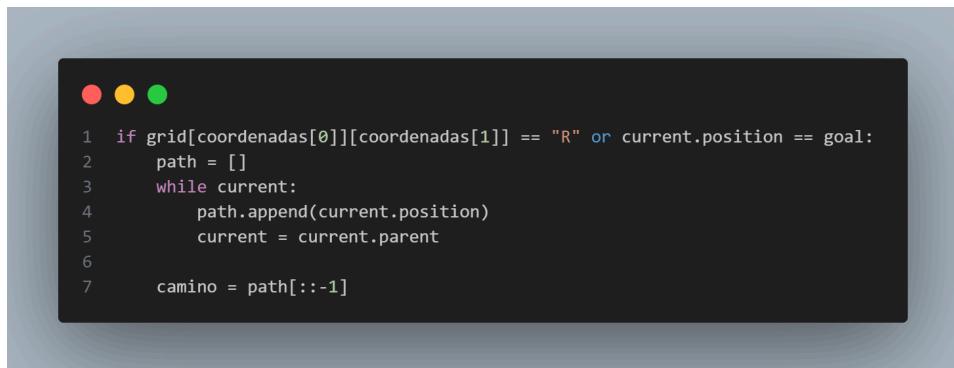
- **queue:** Cola utilizada para mantener nodos pendientes de explorar, lo cual sigue el orden de búsqueda en amplitud.
- **visited:** Conjunto que contiene las posiciones ya exploradas, evitando revisitar nodos y optimizando el proceso.
- **parent_map:** Diccionario que rastrea los nodos predecesores, permitiendo reconstruir el camino hacia el objetivo.



Bucle Principal del Algoritmo

El algoritmo extrae nodos de queue, comenzando por la posición inicial y avanzando en orden de amplitud. En cada iteración:

- 1. Verificación de Meta:** Comprueba si el nodo actual contiene el objetivo o un marcador especial ("R" en el código), en cuyo caso el camino hacia el objetivo se reconstruye usando parent_map y se retorna en orden correcto.
- 2. Construcción del Camino:** Si se alcanza el objetivo, se reconstruye el camino acumulado hasta ese punto, invirtiendo el orden para obtener los pasos desde el inicio hasta el objetivo.



Exploración de Vecinos

Para cada nodo extraído, se evalúan las posiciones vecinas según los movimientos permitidos:

1. **Límites y Obstáculos:** Verifica que cada vecino esté dentro de los límites de la cuadrícula y no sea un obstáculo.
2. **Nodos No Visitados:** Si el vecino cumple las condiciones y no ha sido visitado, se agrega a queue para su futura exploración.
3. **Detección de Galletas:** Si Piggy encuentra una celda con "galleta", self.find_galleta se actualiza y se imprime un mensaje.



```

1
2     next_step = camino[1]
3     if grid[next_step[0]][next_step[1]] == "G":
4         self.find_galleta = True
5         print("Piggy encontro la galleta!!!!")
6     return camino # Devuelve el camino en orden correcto
7
8 # Explorar vecinos
9 for dx, dy in movimientos:
10    x = coordenadas[0] + dx
11    y = coordenadas[1] + dy
12
13    x_valido = 0 <= x < len(grid)
14    y_valido = 0 <= y < len(grid[0])
15
16    if x_valido and y_valido:
17
18        nueva_celda = grid[x][y]
19        vecino_pos = (x, y)
20
21        # Verifica que no sea un obstáculo y no haya sido visitado
22        if nueva_celda != 1 and vecino_pos not in visited:
23
24            # Agregar a la cola para explorar en la siguiente iteración
25            new_node = Node(vecino_pos, current)
26
27            queue.append(new_node)
28            visited.add(vecino_pos)
29
30            # Registro del predecessor
31            parent_map[vecino_pos] = current.position
32

```

Retorno de Resultados

Al finalizar la búsqueda, si se encuentra un camino, se retorna en orden correcto. Si no existe una ruta posible hacia el objetivo, el algoritmo devuelve una lista vacía, indicando que el objetivo es inaccesible desde la posición inicial.

Función Principal

Para la integración de todos los módulos encontramos la función principal, la cual se encarga de instanciar el laberinto y ejecutar las funciones de búsqueda de cada agente de acuerdo a los resultados obtenidos en cada renderización del algoritmo. Para la construcción del laberinto y mostrar los movimientos de cada agente se realizan los siguientes pasos:

1. **Calcular la ruta de Rene a Elmo:** como agente principal se encuentra Rene, este agente tiene un objetivo fijo que no va a cambiar a medida que se realicen los movimientos, por ende su ruta solo se debe calcular una vez iniciado el juego
2. **Renderizar actualizaciones en el Juego:** Se establece un bucle cuya función es actualizar la ventana gráfica en la que se reflejan los movimientos de cada agente.
3. **Validar Turno:** En cada iteración sólo un agente se puede mover, el primer agente en moverse es rene y por ende se irá intercalando con los movimientos de Piggy durante la ejecución.
4. **Mover a Rene:** en caso de que le corresponda el turno, se extrae el movimiento siguiente de la ruta obtenida del algoritmo de Búsqueda.
5. **Mover a Piggy:** después de cada movimiento de Rene se ejecuta el algoritmo de búsqueda asignado a piggy, este retornara el movimiento siguiente a realizar antes que la posición de Rene cambie.

```

● ○ ● ○
1 def juego():
2     def dibujar_mapa():
3         VENTANA.fill(NEGRO)
4         laberinto.dibujar(VENTANA)
5         pygame.display.flip()
6
7     if laberinto.mapa == None:
8         laberinto.set_mapa([[0 for _ in range(5)] for _ in range(5)])
9         laberinto.generar_mapa()
10
11    rene_pos, piggy_pos, elmo_pos, galleta_pos = laberinto.get_positions()
12    dibujar_mapa()
13
14    rene_pos_anterior = None
15    piggy_pos_anterior = None
16
17    rene = Rene(rene_pos)
18    piggy = Piggy(piggy_pos)
19    costo_acumulado_piggy = 0
20
21
22    rene_path = rene.get_path(laberinto.mapa)
23    if rene.has_path:
24        rene_path = deque(rene_path)
25        rene_path.popleft()
26        turno = rene
27    else:
28        turno = piggy
29
30    while True:
31        sleep(1)
32
33        for evento in pygame.event.get():
34            if evento.type == pygame.QUIT:
35                quit()
36
37        if rene.has_path and len(rene_path) == 0:
38            end_game("Rene y Elmo se encontraron")
39            return
40        elif not rene.has_path :
41            turno = piggy
42
43        if piggy.find_rene:
44            end_game("Piggy y Rene se encontraron")
45            return
46
47        print("turno", turno)
48        if turno == piggy:
49            if not piggy.find_rene:
50
51                piggy_pos_anterior = piggy_pos
52                movimiento, costo = piggy.move(rene_pos, laberinto.mapa)
53                costo_acumulado_piggy += costo
54
55                if not rene.has_path and movimiento == piggy_pos_anterior:
56                    end_game("Los agentes no tienen caminos")
57                    return
58
59                piggy_pos = movimiento
60
61                laberinto.mover_agente(
62                    piggy_pos, "P", elmo_pos, piggy_pos_anterior, galleta_pos)
63
64                turno = rene
65            elif turno == rene and rene_path:
66                print("Mueve Rene")
67
68                rene_pos_anterior = rene_pos
69                rene_pos = rene_path.popleft()
70
71                laberinto.mover_agente(
72                    rene_pos, "R", elmo_pos, rene_pos_anterior, galleta_pos)
73
74                turno = piggy
75                if piggy_pos == rene_pos:
76                    piggy.find_rene = True
77
78        print("Costo Acumulado de piggy", costo_acumulado_piggy)
79        dibujar_mapa()
80

```

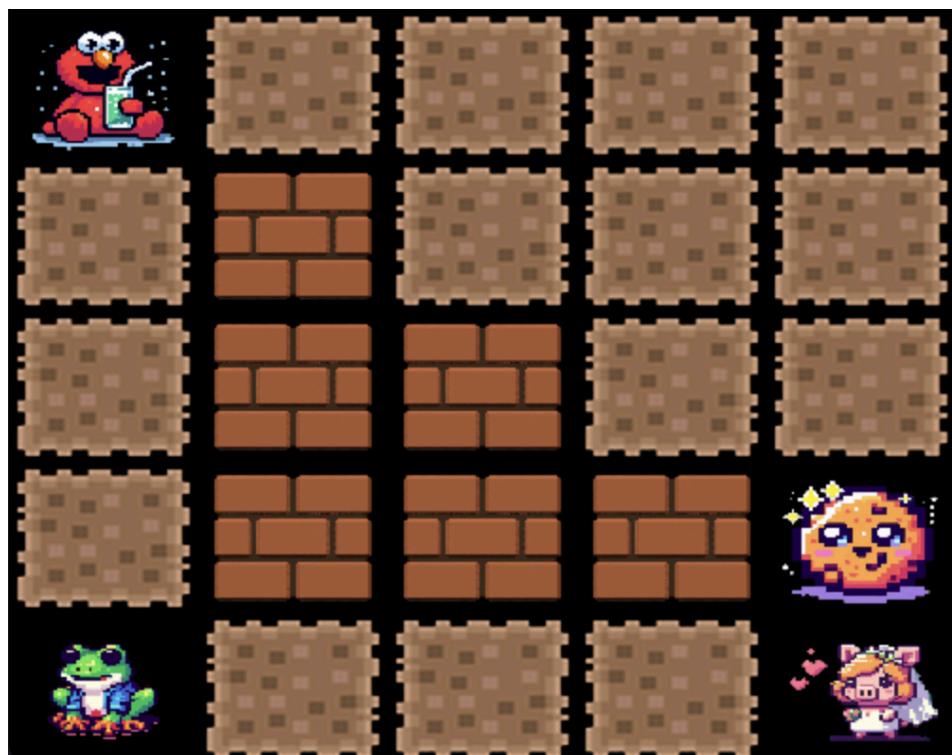
Resultados

Para la verificación del funcionamiento del juego se establecieron 4 escenarios:

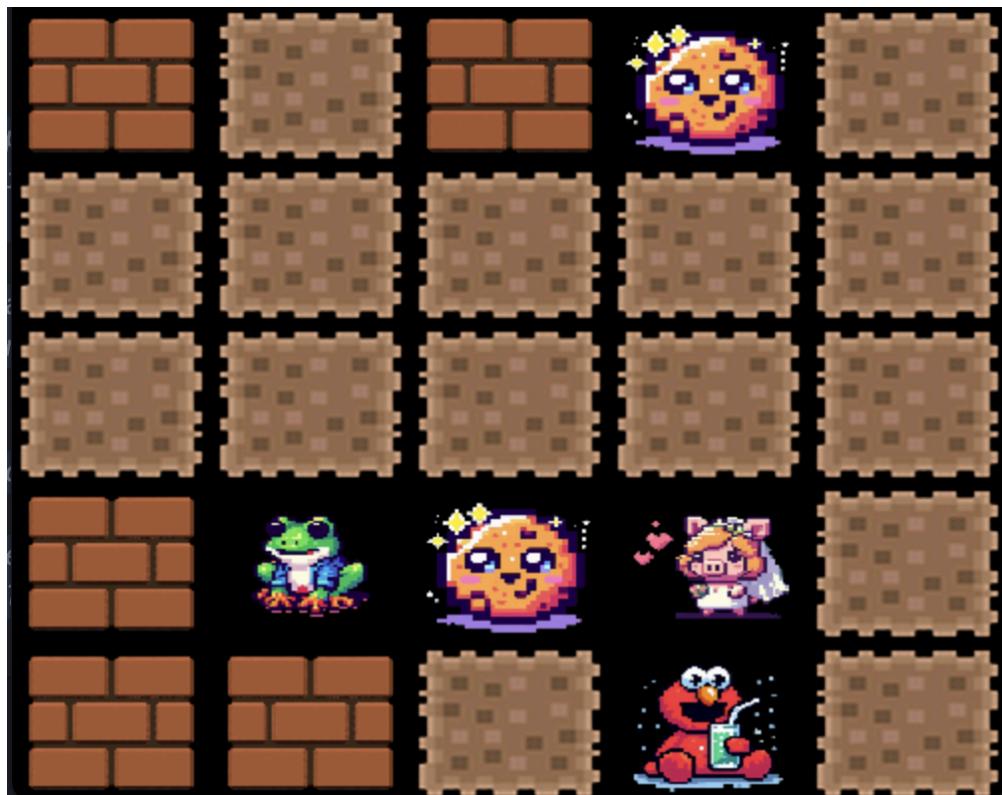
- **Escenario A:** corresponde al laberinto establecido en el enunciado del juego.



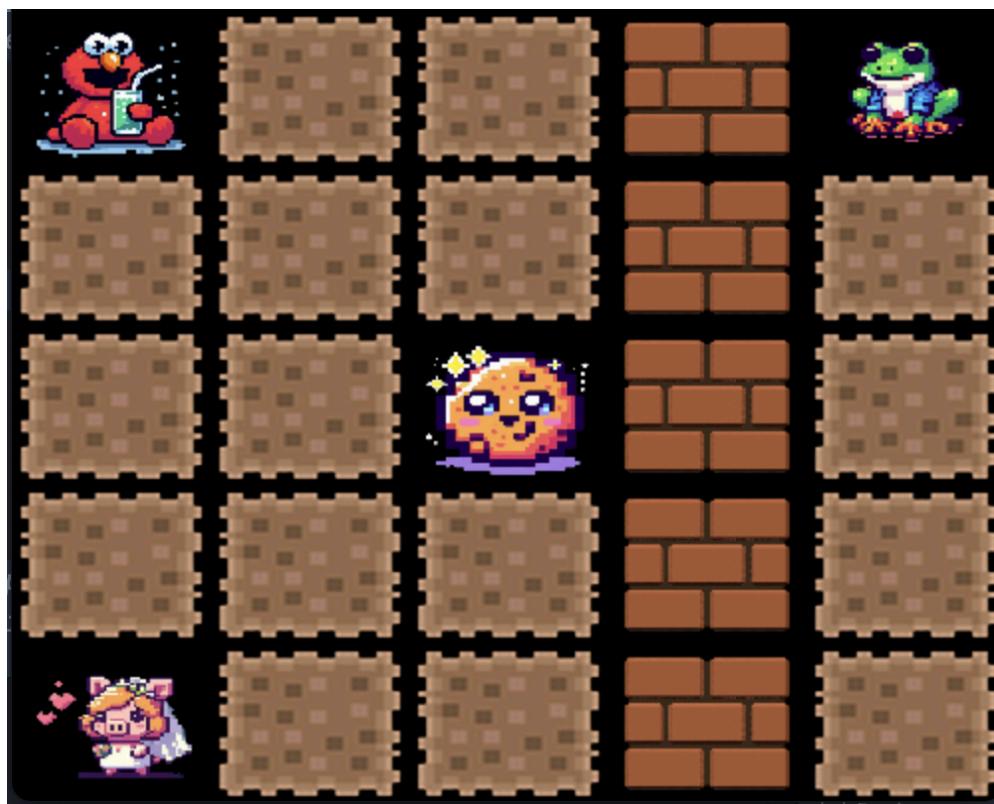
- **Escenario B:** se modeló un laberinto en el que Rene siempre encuentra a Elmo.



- Escenario C: se modeló un laberinto en el que Piggy siempre alcanza a Rene.

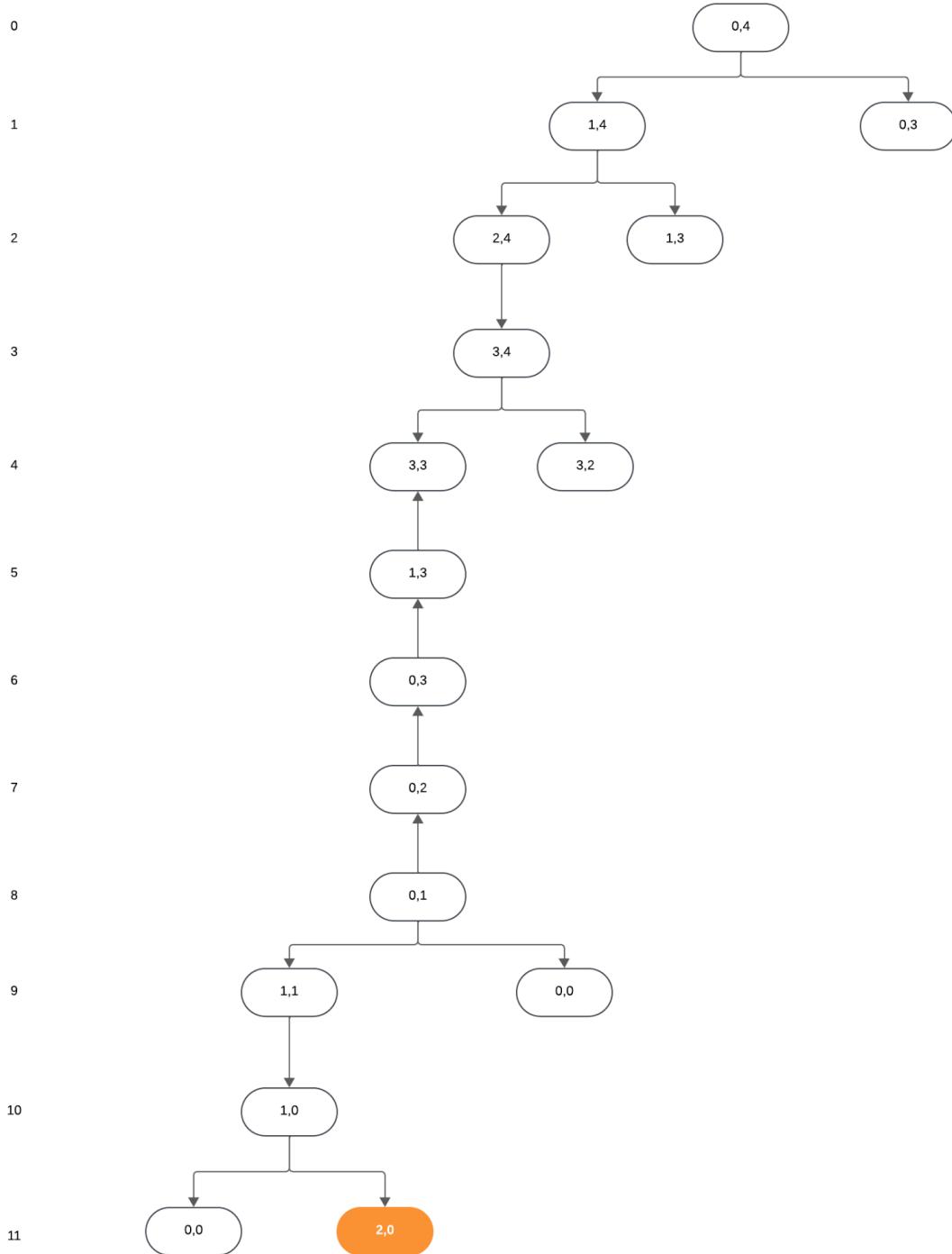


- Escenario D: se modeló un laberinto en el que los agentes no pueden alcanzar su objetivo.



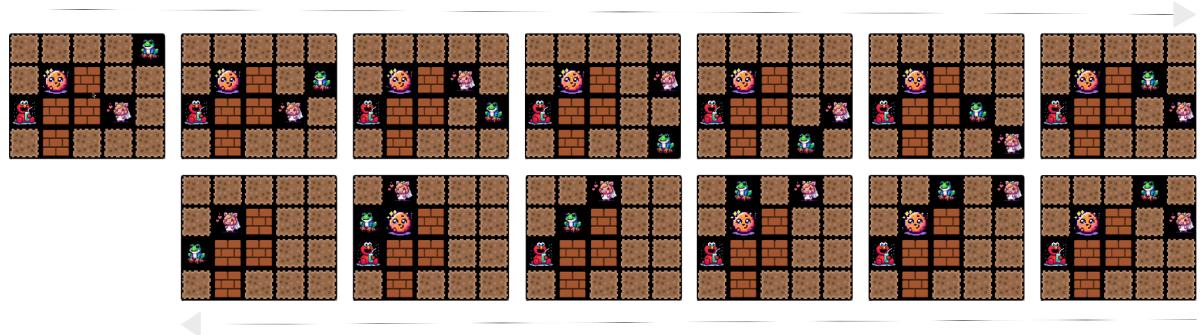
Escenario A

En este caso encontramos que Rene alcanza su objetivo encontrando a Elmo. Al realizar la ejecución teórica del algoritmo de búsqueda encontramos el siguiente árbol



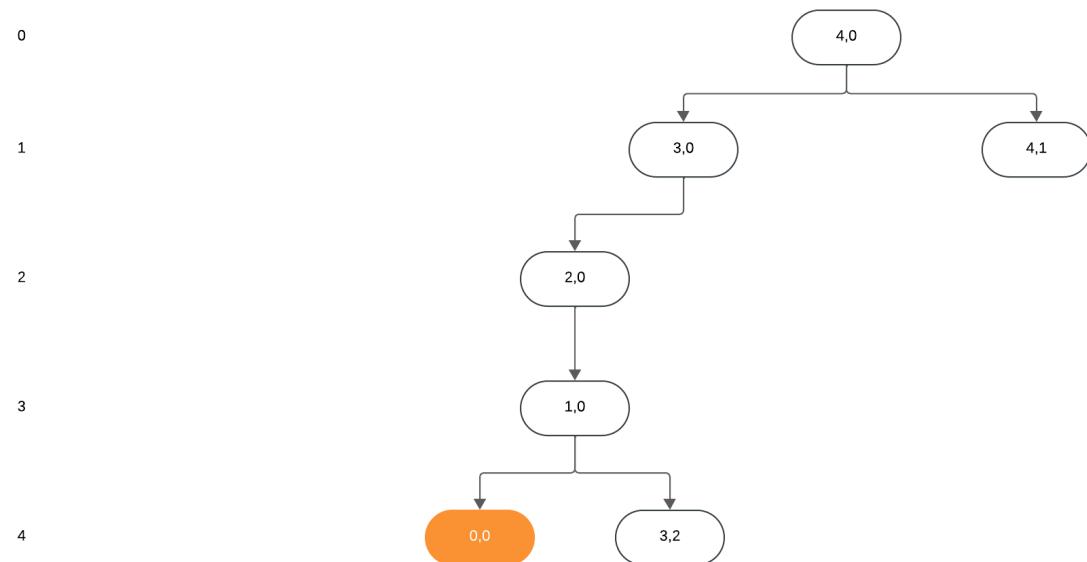
Como se puede observar el agente llega a la solución después de 11 iteraciones, esto debido a que para llegar a la coordenada (1,0) el agente solo tiene un camino posible y al implementar la estrategia de evitar ciclos, al expandir el hijo izquierdo encontramos que no

hay más movimientos posibles que no impliquen regresar a un estado anterior, por lo tanto se expande el nodo siguiente el cual corresponde a la meta u objetivo del agente.



Escenario B

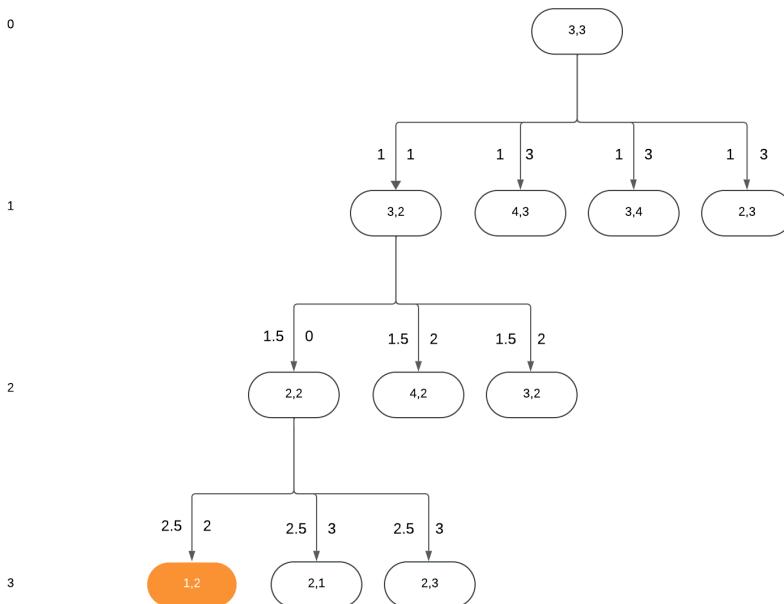
En este caso encontramos un laberinto más simple que el anterior y se repite la situación en la que Rene alcanza su objetivo, la construcción de su ruta encontramos el siguiente árbol de estados:



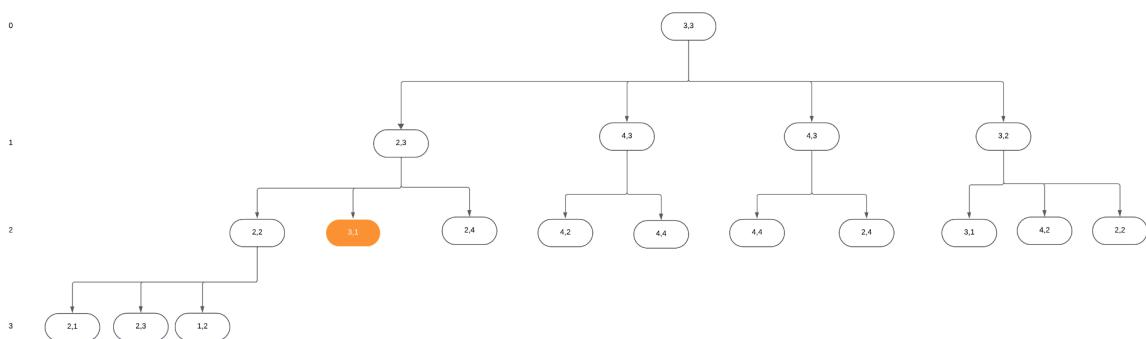
Escenario C

En este caso encontramos que Rene no alcanza su objetivo encontrando a Elmo ya que piggy lo alcanza utilizando la búsqueda A* y búsqueda por amplitud . Al realizar la ejecución teórica del algoritmo de búsqueda encontramos el siguiente árbol:

Búsqueda A*:



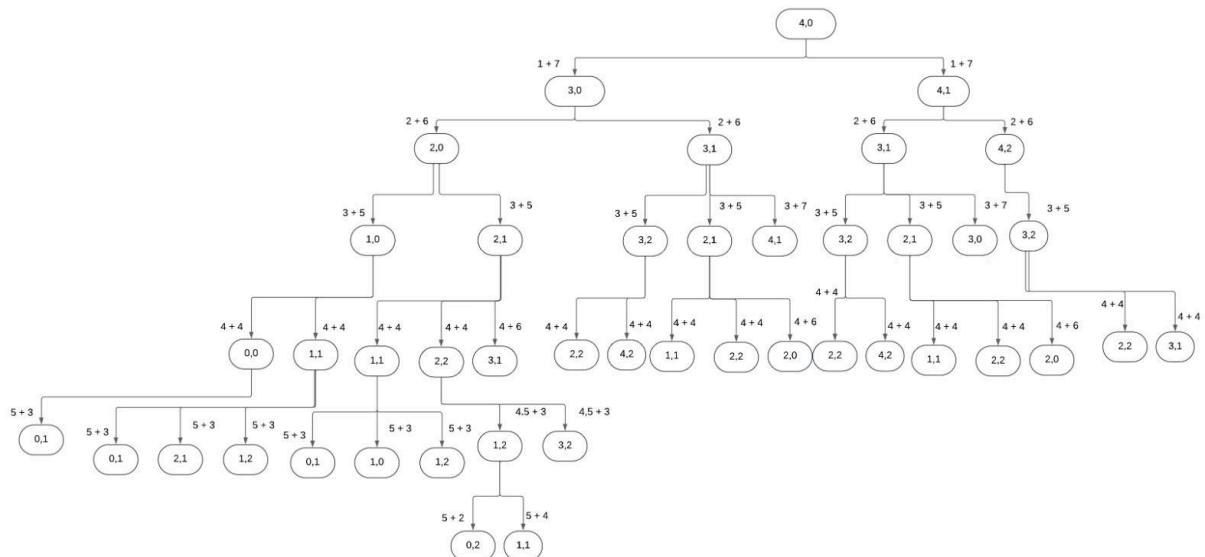
Búsqueda por amplitud:

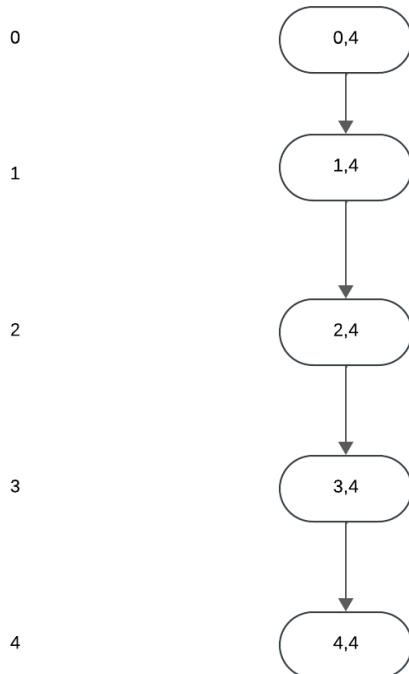


Escenario D

En este caso encontramos que Rene está atascado, y por ende no puede acceder a elmo en el árbol podemos observar que se genera la ruta hasta el último nodo accesible, pero en el programa no se ve reflejado ningún movimiento debido a que la ruta solo se construye si se llega al objetivo.

En el caso del agente Piggy es similar puesto que esta al no acceder a Rene hace el cálculo de todos los nodos accesibles pero no se ve reflejado ningún movimiento puesto que no llega a su objetivo.





Referencias

- Reina, M. (2011, October 27). *Algoritmo de búsqueda A* (PathFinding A*) – XNA*. Escarbando Código. <https://escarbandocodigo.wordpress.com/2011/07/11/1051/>
- Métodos de Búsqueda IA.* (n.d.).
<https://inteligenciaartificialgrupo33.blogspot.com/p/metodos-de-busqueda-y-ejemplos.html>
- AGENTES SOFTWARE | Oficina de Transferencia de Resultados de Investigación.* (n.d.).
<https://www.ucm.es/otri/complutransfer-agentes-software>