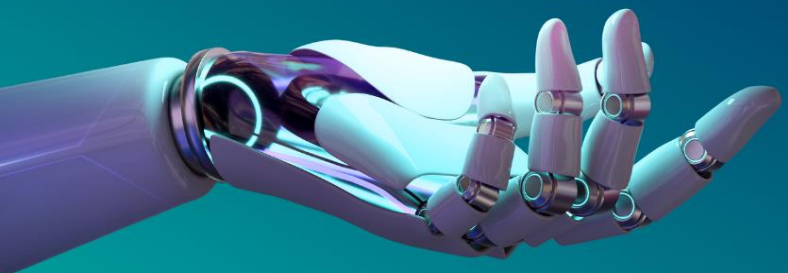
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

UNIVERSIDAD DE GUADALAJARA

PRÁCTICA 2

LABERINTO - AMPLITUD



Alumnos:

Carbajal Armenta Yessenia Paola

Leon Estrada Rafael

Medina Bolaños Danna Paola

Códigos:

220286482

220286121

220286938

Maestro:

Oliva Navarro Diego Alberto

Materia:

Inteligencia Artificial

Sección:

D05

Guadalajara, Jal. a 21 de abril del 2023

Práctica 2 – Laberinto - Amplitud

Instrucciones

Problema 2 del documento Practica 2_SI. pdf utilizando **la búsqueda AMPLITUD**.

Código

A continuación, se explicará el código de manera seccionada para una mejor comprensión.

```
import pygame
import random
from collections import deque
import sys


WIDTH, HEIGHT = 500, 500
ROWS, COLS = 15, 15
SQUARE_SIZE = WIDTH // COLS

WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
YELLOW = (255, 255, 0)
BLUE = (0, 255, 200)

BUSQUEDA_AMPLITUD = 0
BUSQUEDA_PROFUNDIDAD = 1
IDDFS = 2
ALGORITMO = BUSQUEDA_AMPLITUD
```

En esta sección se importan algunos módulos necesarios para el funcionamiento del código. **pygame** es una biblioteca que permite la creación de videojuegos, y en este caso se utilizará para dibujar la cuadrícula del laberinto. **random** es una biblioteca que se utiliza para generar números aleatorios, lo que se utilizará para colocar obstáculos aleatorios en el laberinto. **collections** se utiliza para crear la cola que se utilizará en el algoritmo de búsqueda en anchura.

También se definen algunas constantes, como el tamaño de la ventana y el número de filas y columnas en la cuadrícula. **SQUARE_SIZE** se define como el ancho de la ventana dividido por el número de columnas. Luego, se definen algunos colores que se utilizarán para dibujar la cuadrícula. Por último, se definen algunas constantes que se utilizarán para elegir el algoritmo de búsqueda que se utilizará. En este caso, el algoritmo de búsqueda en amplitud se utilizará de forma predeterminada.

```
class Node:
    def __init__(self, x, y, parent=None):
        self.x = x
        self.y = y
        self.parent = parent
```

Aquí se define la clase **Node**, que se utilizará para representar los nodos en el árbol de búsqueda. Cada nodo tiene una posición en la cuadrícula (**x** e **y**) y una referencia a su nodo padre (**parent**).

```
def create_grid():
    grid = [[WHITE] * COLS for _ in range(ROWS)]

    num_obstaculos = random.randint(5, 15)
    for _ in range(num_obstaculos):
        size = random.randint(2, 4)
        x = random.randint(0, ROWS - size - 1)
        y = random.randint(0, COLS - size - 1)

        for i in range(x, x + size):
            for j in range(y, y + size):
                grid[i][j] = BLACK

    grid[0][0] = GREEN
    grid[ROWS - 1][COLS - 1] = RED
    return grid
```

Esta parte del código define una función llamada **create_grid()**, que crea una matriz bidimensional de filas y columnas con un color de fondo blanco. A continuación, se generan obstáculos de tamaño aleatorio en posiciones aleatorias dentro de la matriz. Estos obstáculos se marcan como negro en la matriz.

Luego, se establecen los puntos de inicio y final para el algoritmo de búsqueda de camino. El punto de inicio se marca como verde en la esquina superior izquierda de la matriz y el punto final se marca como rojo en la esquina inferior derecha de la matriz. Finalmente, la función devuelve la matriz con los obstáculos y los puntos de inicio y final.

```
def draw_grid(window, grid):
    for i in range(ROWS):
        for j in range(COLS):
            pygame.draw.rect(window, grid[i][j], (j * SQUARE_SIZE, i *
SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE))
            pygame.draw.rect(window, BLACK, (j * SQUARE_SIZE, i *
SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE), 1)
def valid_move(x, y, grid):
    if x >= 0 and x < ROWS and y >= 0 and y < COLS:
        if grid[x][y] == WHITE or grid[x][y] == RED or (ALGORITMO == IDDFS
and (grid[x][y] == YELLOW or grid[x][y] == GREEN)):
            return True
    return False
```

La función **draw_grid** es responsable de dibujar la cuadrícula en la ventana de Pygame, utilizando los colores de cada celda especificados en la matriz **grid**. Cada celda se representa como un rectángulo del tamaño definido por la constante **SQUARE_SIZE**.

La función **valid_move** comprueba si el movimiento de la posición **(x, y)** es válido en la cuadrícula **grid**. Devuelve **True** si la posición está dentro de los límites de la cuadrícula, y si la celda en esa posición es blanca, roja o amarilla (en el caso de la búsqueda de IDDFS, que utiliza el amarillo para representar las celdas que se han visitado en iteraciones anteriores), o verde (en el caso de la búsqueda de A*) para representar la celda de inicio. Si la celda es negra, el movimiento no es válido.

```
def search(grid, window):
    start = Node(0, 0)
    end = Node(ROWS - 1, COLS - 1)

    if ALGORITMO == BUSQUEDA_AMPLITUD:
        return bfs(grid, window, start, end)
    elif ALGORITMO == BUSQUEDA_PROFUNDIDAD:
        return dfs(grid, window, start, end)
```

```
    elif ALGORITMO == IDDFS:
        return iddfs(grid, window, start, end)
    else:
        print("Algoritmo no válido")
        return None
```

Esta función ejecuta una búsqueda en el grid utilizando el algoritmo especificado en la variable global **ALGORITMO**. Primero se crean los nodos de inicio y fin, con coordenadas (0,0) y (ROWS-1, COLS-1) respectivamente. Luego se utiliza una estructura condicional para ejecutar la función correspondiente a cada algoritmo de búsqueda. Si **ALGORITMO** no es un valor válido, la función imprime un mensaje y devuelve **None**.

```
def bfs(grid, window, start, end):
    queue = deque([start])
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    while queue:
        current_node = queue.popleft()

        for dx, dy in directions:
            new_x, new_y = current_node.x + dx, current_node.y + dy
            if valid_move(new_x, new_y, grid):
                new_node = Node(new_x, new_y, current_node)
                queue.append(new_node)

                if grid[new_x][new_y] == RED:
                    return new_node

                if grid[new_x][new_y] == WHITE:  # Add this line
                    grid[new_x][new_y] = YELLOW  # Indent this line
                    draw_grid(window, grid)
                    pygame.display.update()
                    check_pygame_events()
                    pygame.time.delay(50)
    return None
```

Esta función implementa el algoritmo de búsqueda en amplitud (BFS) para encontrar el camino más corto desde el nodo de inicio hasta el nodo de destino en una cuadrícula.

Primero se inicializa una cola con el nodo de inicio y se define una lista de direcciones permitidas. Luego, mientras la cola no esté vacía, se extrae el primer elemento y se

comprueba si hay algún vecino al que se pueda mover. Si se puede mover, se crea un nuevo nodo y se agrega a la cola.

Si el nodo recién creado corresponde al nodo de destino, entonces se devuelve este nodo, ya que se ha encontrado el camino más corto. Si no, se marca la celda visitada en la cuadrícula con el color amarillo para indicar que se ha explorado pero no se ha llegado al destino.

Finalmente, si no se puede encontrar un camino al nodo de destino, la función devuelve **None**.

```
def reconstruct_path(end_node, grid):
    current_node = end_node.parent
    while current_node.parent is not None:
        if grid[current_node.x][current_node.y] != GREEN and
grid[current_node.x][current_node.y] != RED:
            grid[current_node.x][current_node.y] = BLUE
        current_node = current_node.parent

def check_pygame_events():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

La función **reconstruct_path** se encarga de reconstruir el camino desde el nodo final hasta el inicio, marcando los nodos del camino con el color azul en la cuadrícula.

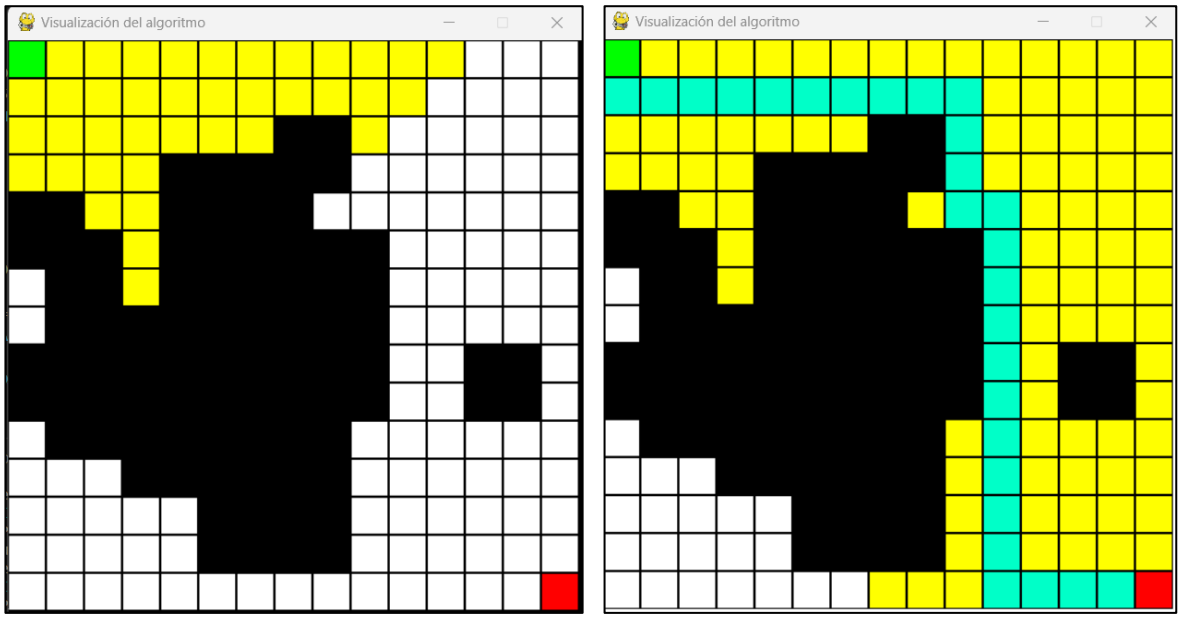
La función **check_pygame_events** se encarga de verificar si ocurre algún evento en la ventana de Pygame, y si el evento es **QUIT**, se cierra la ventana y se sale del programa mediante las funciones **pygame.quit()** y **sys.exit()**, respectivamente.

```
def main():
    pygame.init()
    window = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Visualización del algoritmo")
    grid = create_grid()
    end_node = search(grid, window)
    if end_node is not None:
        reconstruct_path(end_node, grid)
```

```
    running = True

    while running:
        window.fill(WHITE)
        draw_grid(window, grid)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        pygame.display.update()

    pygame.quit()


main()
```

Esta función es la función principal del programa. Primero inicializa Pygame y crea la ventana de visualización. Luego crea la cuadrícula para el algoritmo y llama a la función de búsqueda para encontrar el camino desde el inicio hasta el final. Si se encuentra un camino, llama a la función reconstruct_path para marcar los nodos del camino en azul. Después, entra en un bucle que dibuja la cuadrícula y maneja los eventos de Pygame, permitiendo que el usuario cierre la ventana y salga del programa. Finalmente, cuando el usuario cierra la ventana, se llama a la función quit de Pygame para detener la aplicación.

Ejecución



Conclusiones

El algoritmo de búsqueda por amplitud es un algoritmo de búsqueda en grafos que comienza en el nodo raíz y explora todos los vecinos del nodo actual antes de pasar a los vecinos del siguiente nivel. Esto lo hace mediante colas.

La principal ventaja del algoritmo de búsqueda por amplitud es que siempre encuentra el camino más corto en un grafo no ponderado, lo que lo convierte en una buena opción para aplicaciones en las que el tiempo no es una preocupación y se requiere el camino más corto.

Por otro lado, una desventaja del algoritmo de búsqueda por amplitud es que puede ser muy ineficiente en términos de memoria y tiempo de ejecución si el grafo es muy grande y no está bien estructurado. Además, a medida que el grafo se hace más grande, el algoritmo se vuelve menos práctico y puede ser necesario utilizar otros algoritmos de búsqueda.

En esta práctica de laberinto, el algoritmo de búsqueda por amplitud se utiliza para encontrar el camino más corto desde el inicio hasta el final del laberinto. A medida que se exploran los nodos, se va marcando en el laberinto los nodos visitados y los nodos que forman parte del camino.