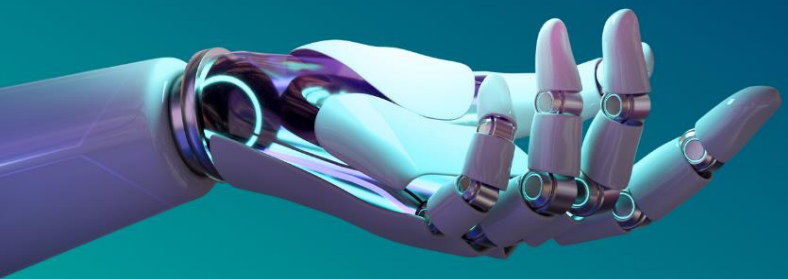
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

UNIVERSIDAD DE GUADALAJARA

PRÁCTICA 2

LABERINTO - PROFUNDIDAD Y PROFUNDIDAD ITERATIVA



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 - Profundidad y Profundidad Iterativa

Instrucciones

Problema 2 del documento Practica 2_SI. pdf utilizando **la búsqueda profundidad y profundidad iterativa.**

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 dfs(grid, window, start, end):
    stack = [start]
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    while stack:
        current_node = stack.pop()

        if grid[current_node.x][current_node.y] == RED:
            return current_node

        if grid[current_node.x][current_node.y] != YELLOW:
            grid[current_node.x][current_node.y] = YELLOW
            draw_grid(window, grid)
            pygame.display.update()
            check_pygame_events()
            pygame.time.delay(50)

            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)
                    stack.append(new_node)

    return None
```

Esta es la función que realiza la búsqueda en profundidad (DFS). Al igual que en BFS, toma como entrada el **grid**, la **window**, el nodo de inicio **start** y el nodo de destino **end**.

El algoritmo DFS utiliza una pila para realizar la búsqueda en profundidad. Se inicia con el nodo de inicio, se apila en la pila y se comprueba si es el nodo de destino. Si no es el nodo de destino, se extrae un nodo de la pila y se examinan sus vecinos. Si el vecino es un movimiento válido, se crea un nuevo nodo para el vecino y se agrega a la pila. Si no es un movimiento válido, se ignora.

Si un nodo es examinado por primera vez, se marca como amarillo en el grid y se actualiza la ventana. Si el nodo es vecino del nodo de destino, se devuelve el nodo. Si no se encuentra el nodo de destino, el algoritmo devuelve **None**.

```
def iddfs(grid, window, start, end):
    def dls(grid, window, current_node, end, depth):
        if depth == 0 and grid[current_node.x][current_node.y] == RED:
            return current_node

        if depth > 0:
            directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
            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):
                    prev_color = grid[new_x][new_y]
                    grid[new_x][new_y] = YELLOW
                    draw_grid(window, grid)
                    pygame.display.update()
                    check_pygame_events()
                    pygame.time.delay(50)

                    result = dls(grid, window, Node(new_x, new_y,
current_node), end, depth - 1)

                    if result is None:
                        grid[new_x][new_y] = prev_color
                        draw_grid(window, grid)
                        pygame.display.update()
                        pygame.time.delay(50)
                    else:
                        return result
```

```
        return None

    depth = 0
    max_depth = ROWS * COLS
    while depth < max_depth:
        result = dls(grid, window, start, end, depth)
        if result is not None:
            return result
        depth += 1

    print("No se encontró un camino")
    return None
```

Esta función implementa el algoritmo de búsqueda en profundidad iterada (IDDFS, por sus siglas en inglés) para encontrar un camino entre un nodo de inicio y un nodo de destino en un grafo. El IDDFS es similar a la búsqueda en profundidad (DFS) pero en lugar de buscar hasta encontrar la solución o llegar a una hoja, busca iterativamente aumentando la profundidad máxima permitida en cada iteración.

En esta implementación, la función **dls** realiza una búsqueda en profundidad limitada (DLS, por sus siglas en inglés) desde el nodo actual hasta una profundidad máxima **depth**. Si se encuentra un nodo objetivo, se devuelve ese nodo. De lo contrario, se retrocede y se continúa la búsqueda desde otro nodo. El algoritmo principal de IDDFS llama a la función **dls** con incrementos de profundidad y devuelve el nodo objetivo si se encuentra, o **None** si no se encuentra ningún camino.

```
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.

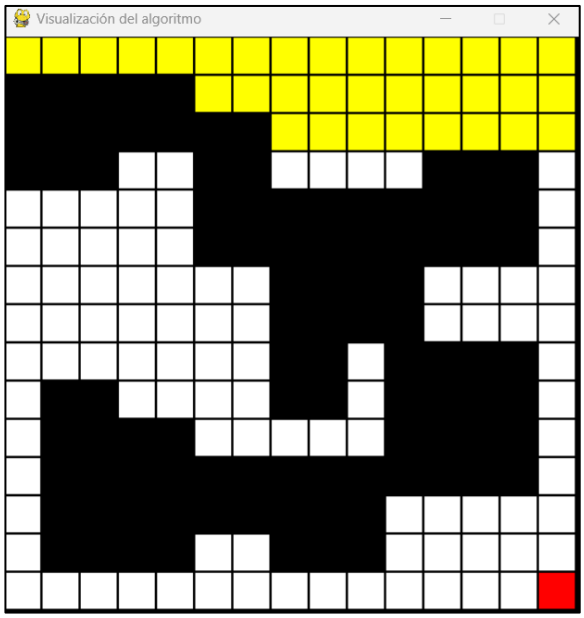
La función **check_pygame_events** se encarga de verificar si ocurre algún evento en la ventana de Pygame, y si el evento es **QUIT**, se cierra la ventana y se sale del programa mediante las funciones **pygame.quit()** y **sys.exit()**, respectivamente.

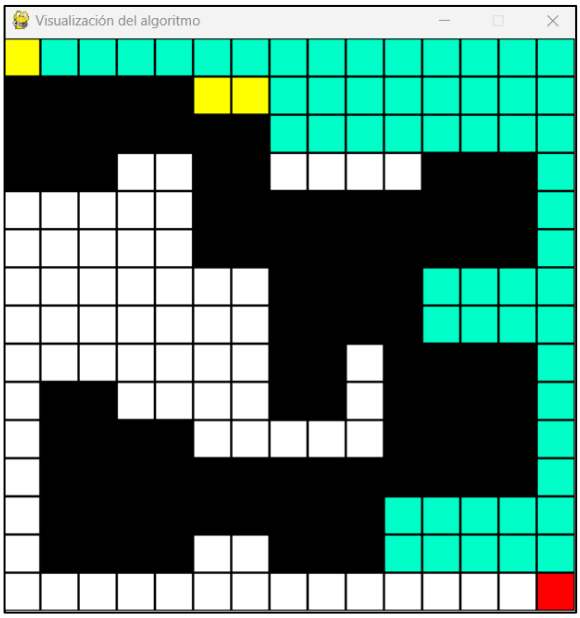
```
def main():
    pygame.init()
    window = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Visualización del algoritmo")
    grid = create_grid()
    end_node = search(grid, window)
    if end_node is not None:
        reconstruct_path(end_node, grid)
    running = True

    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.

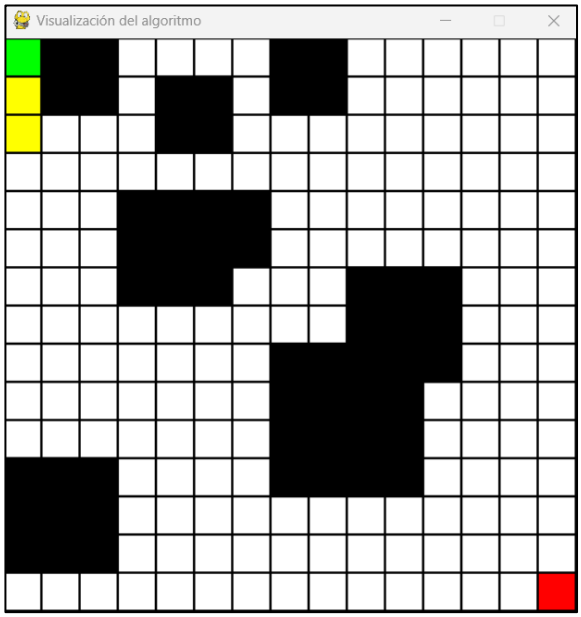
Ejecución

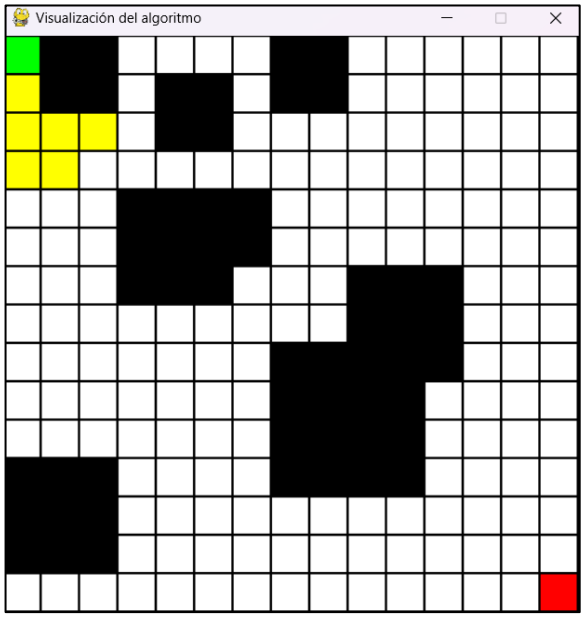
Búsqueda en Profundidad:





Búsqueda en Profundidad Iterativa:





Justificación algoritmo IDDFS

El algoritmo IDDFS, aunque teóricamente puede encontrar la solución en espacios de búsqueda más grandes que el algoritmo de búsqueda en anchura, en la práctica puede ser muy costoso en términos de tiempo de ejecución y no siempre garantiza encontrar la solución óptima. En el caso de la implementación en esta práctica, la combinación de un laberinto complejo y el retraso en la actualización de la ventana de pygame hace que la ejecución del algoritmo IDDFS sea muy lenta y en algunos casos nunca termine de encontrar una solución.

Una solución alternativa sería implementar un algoritmo de búsqueda A* con una heurística bien diseñada. El algoritmo de búsqueda A* es conocido por su eficiencia y garantiza encontrar la solución óptima. Una buena heurística podría ser la distancia Manhattan desde el nodo actual hasta el nodo final. La implementación de un algoritmo de búsqueda A* puede requerir un poco más de trabajo en la definición de la heurística y la implementación del cálculo de los valores de costo y heurística, pero en términos de eficiencia y garantía de encontrar la solución óptima, sería una opción más factible.

Conclusiones

La práctica del laberinto demostró la utilidad de los algoritmos de búsqueda en la resolución de problemas complejos. En particular, los algoritmos de profundidad y profundidad iterativa resultaron efectivos para encontrar una solución en el laberinto. La función de estos algoritmos es explorar el espacio de búsqueda, en este caso el laberinto, para encontrar el camino más corto desde el punto de inicio al punto final.

La ventaja del algoritmo de profundidad es su simplicidad y facilidad de implementación. Sin embargo, su desventaja es que puede quedar atrapado en bucles infinitos si no se controla adecuadamente la exploración del espacio de búsqueda. Por otro lado, la ventaja de la profundidad iterativa es que controla el tamaño de la búsqueda para evitar bucles infinitos y garantiza que encuentre la solución óptima en un espacio de búsqueda finito. Sin embargo, su desventaja es que puede tardar mucho tiempo en encontrar una solución si el laberinto es muy grande.

En cuanto al problema con el algoritmo IDDFS, se demostró que no es factible para laberintos grandes debido a su alto costo computacional. Por lo tanto, se propone

utilizar otros algoritmos de búsqueda, como el algoritmo A*, que utiliza una heurística para determinar la dirección más prometedora de la búsqueda y así reducir el costo computacional.

En resumen, los algoritmos de profundidad y profundidad iterativa son útiles en la resolución de problemas de búsqueda como el laberinto, aunque tienen sus ventajas y desventajas. Además, se debe considerar el tamaño del espacio de búsqueda para seleccionar el algoritmo de búsqueda más adecuado.