AI PROJECT

21052136- Amlan prashad

21052349- ritoja poddar

import pygame

import random

import heapq

import sys

# Constants

WIDTH, HEIGHT = 400, 400

GRID\_SIZE = 20

GRID\_WIDTH = WIDTH // GRID\_SIZE

GRID\_HEIGHT = HEIGHT // GRID\_SIZE

SNAKE\_SPEED = 3

# Colors

WHITE = (255, 255, 255)

GREEN = (0, 255, 0)

RED = (255, 0, 0)

BLACK = (0, 0, 0)

# Initialize Pygame

pygame.init()

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption('Snake Game')

# Load the food image

food\_image = pygame.image.load('food.png')  # Make sure to place your food image file in the same directory as your script.

# Snake and Food

snake = [(5, 5)]

food = (15, 15)

# Obstacle Positions

# Define the positions of obstacles in a predetermined pattern

obstacle\_positions = [

    (9, 7),

    (10, 8),

    (11, 9),

    (12, 10),

    (13, 11),

    (5, 8),

    (5, 9),

    (5, 10),

    (5, 11),

    (5, 12),

    (5, 13),

    (5, 14),

    (6, 14),

    (7, 14),

    (8, 14),

    (9, 14),

    (10, 14),

    (11, 14),

    (12, 14),

    # Add more obstacle positions as needed

]

def heuristic(pos1, pos2):

    x1, y1 = pos1

    x2, y2 = pos2

    return abs(x1 - x2) + abs(y1 - y2)  # Manhattan distance

# Define a function for A\* search

def a\_star\_search(start, goal, obstacles):

    # Convert start, goal, and obstacle positions to tuples

    start = tuple(start)

    goal = tuple(goal)

    obstacles = set(tuple(pos) for pos in obstacles)

    open\_list = []

    closed\_list = set()

    came\_from = {}

    g\_score = {start: 0}

    f\_score = {start: heuristic(start, goal)}

    heapq.heappush(open\_list, (f\_score[start], start))

    while open\_list:

        \_, current = heapq.heappop(open\_list)

        if current == goal:

            path = []

            while current in came\_from:

                path.append(current)

                current = came\_from[current]

            path.reverse()

            return path

        closed\_list.add(current)

        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

            neighbor = (current[0] + dx, current[1] + dy)

            if neighbor in closed\_list or neighbor in obstacles:

                continue

            tentative\_g\_score = g\_score[current] + 1

            if neighbor not in [n[1] for n in open\_list] or tentative\_g\_score < g\_score[neighbor]:

                came\_from[neighbor] = current

                g\_score[neighbor] = tentative\_g\_score

                f\_score[neighbor] = g\_score[neighbor] + heuristic(neighbor, goal)

                heapq.heappush(open\_list, (f\_score[neighbor], neighbor))

    return None  # No path found

# Define a function for DFS search

def dfs\_search(start, goal, obstacles):

    stack = [(start, [])]  # Initialize a stack with the start position and an empty path

    visited = set()  # Keep track of visited positions to avoid loops

    while stack:

        current\_position, path = stack.pop()  # Get the current position and path

        if current\_position == goal:

            # Found a path to the goal

            return path

        visited.add(current\_position)

        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

            neighbor = (current\_position[0] + dx, current\_position[1] + dy)

            if (

                neighbor not in visited and

                neighbor not in obstacles and

                0 <= neighbor[0] < GRID\_WIDTH and 0 <= neighbor[1] < GRID\_HEIGHT

            ):

                # Add the neighbor to the stack with the updated path

                stack.append((neighbor, path + [current\_position]))

    # If no path is found, return an empty path

    return []

# Define a function for BFS search

def bfs\_search(start, goal, obstacles):

    queue = [(start, [])]  # Initialize a queue with the start position and an empty path

    visited = set()  # Keep track of visited positions to avoid loops

    while queue:

        current\_position, path = queue.pop(0)  # Get the current position and path

        if current\_position == goal:

            # Found a path to the goal

            return path

        visited.add(current\_position)

        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

            neighbor = (current\_position[0] + dx, current\_position[1] + dy)

            if (

                neighbor not in visited and

                neighbor not in obstacles and

                0 <= neighbor[0] < GRID\_WIDTH and 0 <= neighbor[1] < GRID\_HEIGHT

            ):

                # Add the neighbor to the queue with the updated path

                queue.append((neighbor, path + [current\_position]))

    # If no path is found, return an empty path

    return []

# Define a function for Greedy Best First Search

def greedy\_best\_first\_search(start, goal, obstacles):

    open\_list = []

    closed\_list = set()

    came\_from = {}

    def greedy\_heuristic(pos):

        # Define your heuristic here; for GBFS, it's an estimate of the remaining cost (e.g., Manhattan distance)

        return abs(pos[0] - goal[0]) + abs(pos[1] - goal[1])

    open\_list.append((greedy\_heuristic(start), start))

    while open\_list:

        open\_list.sort()  # Sort the open list based on the heuristic value

        current = open\_list.pop(0)[1]

        if current == goal:

            path = []

            while current in came\_from:

                path.append(current)

                current = came\_from[current]

            path.reverse()

            return path

        closed\_list.add(current)

        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

            neighbor = (current[0] + dx, current[1] + dy)

            if neighbor in closed\_list or neighbor in obstacles:

                continue

            if neighbor not in [n[1] for n in open\_list]:

                came\_from[neighbor] = current

                open\_list.append((greedy\_heuristic(neighbor), neighbor))

    return None  # No path found

# Main game loop

running = True

clock = pygame.time.Clock()

# Set up boundaries around the screen

boundary = [(x, 0) for x in range(GRID\_WIDTH)] + [(x, GRID\_HEIGHT - 1) for x in range(GRID\_WIDTH)] + \

           [(0, y) for y in range(1, GRID\_HEIGHT - 1)] + [(GRID\_WIDTH - 1, y) for y in range(1, GRID\_HEIGHT - 1)]

def check\_collision(new\_head):

    if new\_head in snake[1:] or new\_head in boundary:

        return True

    return False

def generate\_random\_food(obstacle\_positions):

    while True:

        food = (random.randint(1, GRID\_WIDTH - 2), random.randint(1, GRID\_HEIGHT - 2))

        if food not in obstacle\_positions:

            return food

# Initialize the food position

food = generate\_random\_food(obstacle\_positions)

# eaten\_food\_count = False  # Variable to track if the snake has eaten the food

eaten\_food\_count = 0  # Variable to track how many times the snake has eaten food

path = []  # Initialize a list to store the path coordinates

total\_path\_length = 0  # Initialize total path length

# Initialize a variable to track the type of search algorithm

search\_algorithm = "A\*"  # Start with A\* search

# Add a new variable for the second food

food2 = None

# Add a new variable for the third food

food3 = None

# Add a new variable for the fourth food

food4 = None

while running and eaten\_food\_count < 4:  # Run the game until the snake eats food three times

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            running = False

    # # Snake Movement (manual control)

    # keys = pygame.key.get\_pressed()

    # if keys[pygame.K\_UP] and (len(snake) < 2 or (snake[0][0], snake[0][1] - 1) != snake[1]):

    #     direction = (0, -1)

    # elif keys[pygame.K\_DOWN] and (len(snake) < 2 or (snake[0][0], snake[0][1] + 1) != snake[1]):

    #     direction = (0, 1)

    # elif keys[pygame.K\_LEFT] and (len(snake) < 2 or (snake[0][0] - 1, snake[0][1]) != snake[1]):

    #     direction = (-1, 0)

    # elif keys[pygame.K\_RIGHT] and (len(snake) < 2 or (snake[0][0] + 1, snake[0][1]) != snake[1]):

    #     direction = (1, 0)

    # Check if the snake has eaten both foods to end the game

    if eaten\_food\_count >= 4:

        running = False

    if food2 is None and eaten\_food\_count == 1:

        # Generate the second food after eating the first

        food2 = generate\_random\_food(obstacle\_positions)

    if food3 is None and eaten\_food\_count == 2:

        # Generate the third food after eating the second

        food3 = generate\_random\_food(obstacle\_positions)

    if food4 is None and eaten\_food\_count == 3:

        # Generate the fourth food after eating the third

        food4 = generate\_random\_food(obstacle\_positions)

    if eaten\_food\_count == 0:

        if search\_algorithm == "A\*":

            direction = a\_star\_search(snake[0], food, obstacle\_positions)

        elif search\_algorithm == "DFS":

            direction = dfs\_search(snake[0], food2, obstacle\_positions)

        elif search\_algorithm == "BFS":

            direction = bfs\_search(snake[0], food3, obstacle\_positions)

        else:

            direction = greedy\_best\_first\_search(snake[0], food4, obstacle\_positions)

    elif eaten\_food\_count == 1:

        if search\_algorithm == "A\*":

            direction = a\_star\_search(snake[0], food, obstacle\_positions)

        elif search\_algorithm == "DFS":

            direction = dfs\_search(snake[0], food2, obstacle\_positions)

        elif search\_algorithm == "BFS":

            direction = bfs\_search(snake[0], food3, obstacle\_positions)

        else:

            direction = greedy\_best\_first\_search(snake[0], food4, obstacle\_positions)

    elif eaten\_food\_count == 2:

        if search\_algorithm == "A\*":

            direction = a\_star\_search(snake[0], food, obstacle\_positions)

        elif search\_algorithm == "DFS":

            direction = dfs\_search(snake[0], food2, obstacle\_positions)

        elif search\_algorithm == "BFS":

            direction = bfs\_search(snake[0], food3, obstacle\_positions)

        else:

            direction = greedy\_best\_first\_search(snake[0], food4, obstacle\_positions)

    elif eaten\_food\_count == 3:

        if search\_algorithm == "A\*":

            direction = a\_star\_search(snake[0], food, obstacle\_positions)

        elif search\_algorithm == "DFS":

            direction = dfs\_search(snake[0], food2, obstacle\_positions)

        elif search\_algorithm == "BFS":

            direction = bfs\_search(snake[0], food3, obstacle\_positions)

        else:

            direction = greedy\_best\_first\_search(snake[0], food4, obstacle\_positions)

    # Check if direction is not None

    if direction is not None:

        # if direction:  # Check if direction is not empty

        next\_position = direction[0]

        dx, dy = next\_position[0] - snake[0][0], next\_position[1] - snake[0][1]  # Calculate the direction

        # Add the new head position to the path list

        path.append((snake[0][0] + dx, snake[0][1] + dy))

        # Calculate the length of the path list as the snake moves

        total\_path\_length = len(path)

        # Calculate the Manhattan distance between the snake's head and the food

        manhattan\_distance = abs(snake[0][0] - food[0]) + abs(snake[0][1] - food[1])

        # Move snake

        new\_head = (snake[0][0] + dx, snake[0][1] + dy)

        # new\_head = (snake[0][0] + direction[0], snake[0][1] + direction[1])

        if new\_head == food:

            if not eaten\_food\_count:

                eaten\_food\_count += 1

                food = generate\_random\_food(obstacle\_positions)  # Generate a new food position

            else:

                # Update the direction to move towards the second food

                direction = a\_star\_search(snake[0], food, obstacle\_positions)

                # Optionally, you can also check if there's no possible path to the second food and handle that case.

        else:

            # If food is not eaten, continue moving the snake

            snake.pop()

        # Check for collision with obstacles

        if new\_head in obstacle\_positions:

            running = False

        if check\_collision(new\_head):

            running = False

        snake.insert(0, new\_head)

    # Check if the snake's head has reached the first food

    if snake[0] == food:

        food = generate\_random\_food(obstacle\_positions)  # Generate the next food position

        eaten\_food\_count += 1  # Increase the count of eaten food

        if eaten\_food\_count == 1:

            search\_algorithm = "A\*"  # Display A\* algorithm info for the first food

    # Check if the snake's head has reached the second food

    if snake[0] == food2:

        food2 = generate\_random\_food(obstacle\_positions)  # Generate the next food position

        eaten\_food\_count += 1  # Increase the count of eaten food

        if eaten\_food\_count == 2:

            search\_algorithm = "DFS"  # Display DFS algorithm info for the second food

    # Check if the snake's head has reached the third food

    if snake[0] == food3:

        food3 = generate\_random\_food(obstacle\_positions)  # Generate the next food position

        eaten\_food\_count += 1  # Increase the count of eaten food

        if eaten\_food\_count == 3:

            search\_algorithm = "BFS"  # Display DFS algorithm info for the third food

    # Check if the snake's head has reached the fourth food

    if snake[0] == food4:

        food4 = None  # Remove the fourth food

        eaten\_food\_count += 1  # Increase the count of eaten food

    # Draw everything

    screen.fill(BLACK)

    # Draw obstacles

    for obs in obstacle\_positions:

        pygame.draw.rect(screen, GREEN, (obs[0] \* GRID\_SIZE, obs[1] \* GRID\_SIZE, GRID\_SIZE, GRID\_SIZE))

    # Draw the snake

    for segment in snake:

        pygame.draw.rect(screen, GREEN, (segment[0] \* GRID\_SIZE, segment[1] \* GRID\_SIZE, GRID\_SIZE, GRID\_SIZE))

    # Draw the head

    pygame.draw.rect(screen, RED, (snake[0][0] \* GRID\_SIZE, snake[0][1] \* GRID\_SIZE, GRID\_SIZE, GRID\_SIZE))

    # Draw the food image

    screen.blit(food\_image, (food[0] \* GRID\_SIZE, food[1] \* GRID\_SIZE))

    # Draw the boundary

    for point in boundary:

        pygame.draw.rect(screen, GREEN, (point[0] \* GRID\_SIZE, point[1] \* GRID\_SIZE, GRID\_SIZE, GRID\_SIZE))

    # Display the information when approaching the first food

    if eaten\_food\_count == 0:

        # Calculate the Manhattan distance

        manhattan\_distance = abs(snake[0][0] - food[0]) + abs(snake[0][1] - food[1])

        # Display A\* algorithm info

        font = pygame.font.Font(None, 24)

        algorithm\_info = font.render(f'   A\* Algorithm', True, WHITE)

        screen.blit(algorithm\_info, (10, 20))

        # Display Manhattan distance

        distance\_text = font.render(f'   Manhattan Distance: {manhattan\_distance}', True, WHITE)

        screen.blit(distance\_text, (10, 40))

        # Display Total Path Length

        path\_length\_text = font.render(f'   Total Path Length: {total\_path\_length}', True, WHITE)

        screen.blit(path\_length\_text, (10, 60))

    # Display the information when approaching the second food

    if eaten\_food\_count == 1:

        # Calculate the Manhattan distance

        manhattan\_distance = abs(snake[0][0] - food[0]) + abs(snake[0][1] - food[1])

        # Display DFS algorithm info

        font = pygame.font.Font(None, 24)

        algorithm\_info = font.render(f'   DFS Algorithm', True, WHITE)

        screen.blit(algorithm\_info, (10, 20))

        # Display Manhattan distance

        distance\_text = font.render(f'   Manhattan Distance: {manhattan\_distance}', True, WHITE)

        screen.blit(distance\_text, (10, 40))

        # Display Total Path Length

        path\_length\_text = font.render(f'   Total Path Length: {total\_path\_length}', True, WHITE)

        screen.blit(path\_length\_text, (10, 60))

    # Display the information when approaching the third food

    if eaten\_food\_count == 2:

        # Calculate the Manhattan distance

        manhattan\_distance = abs(snake[0][0] - food[0]) + abs(snake[0][1] - food[1])

        # Display BFS algorithm info

        font = pygame.font.Font(None, 24)

        algorithm\_info = font.render(f'   BFS Algorithm', True, WHITE)

        screen.blit(algorithm\_info, (10, 20))

        # Display Manhattan distance

        distance\_text = font.render(f'   Manhattan Distance: {manhattan\_distance}', True, WHITE)

        screen.blit(distance\_text, (10, 40))

        # Display Total Path Length

        path\_length\_text = font.render(f'   Total Path Length: {total\_path\_length}', True, WHITE)

        screen.blit(path\_length\_text, (10, 60))

    # Display the information when approaching the fourth food

    if eaten\_food\_count == 3:

        # Calculate the Manhattan distance

        manhattan\_distance = abs(snake[0][0] - food[0]) + abs(snake[0][1] - food[1])

        # Display GBFS algorithm info

        font = pygame.font.Font(None, 24)

        algorithm\_info = font.render(f'   GBFS Algorithm', True, WHITE)

        screen.blit(algorithm\_info, (10, 20))

        # Display Manhattan distance

        distance\_text = font.render(f'   Manhattan Distance: {manhattan\_distance}', True, WHITE)

        screen.blit(distance\_text, (10, 40))

        # Display Total Path Length

        path\_length\_text = font.render(f'   Total Path Length: {total\_path\_length}', True, WHITE)

        screen.blit(path\_length\_text, (10, 60))

    pygame.display.flip()

    clock.tick(SNAKE\_SPEED)

# "Game Over" message

font = pygame.font.Font(None, 36)

if eaten\_food\_count:

    game\_over\_text = font.render("      Game Over", True, RED)

else:

    game\_over\_text = font.render("      Game Over", True, RED)

screen.blit(game\_over\_text, (WIDTH // 2 - 120, HEIGHT // 2 - 20))

pygame.display.flip()

# Keep the game running even after game over

while True:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()

            sys.exit()