**Ai Code Basic Info**

**Code:**

import pygame

import heapq

from collections import deque

import time

# Directions for moving in the grid: Right, Left, Down, Up

DIRECTIONS = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# Colors for visualization

WHITE = (255, 255, 255)  # Grid color

BLACK = (0, 0, 0)        # Unused, but can be for the background

GREEN = (0, 255, 0)      # Final point color and path color

RED = (255, 0, 0)        # Obstacle color

BLUE = (0, 0, 255)       # (Unused, but can be used for visualizing path)

GREY = (169, 169, 169)   # Searching (current cell being explored)

# Grid parameters

CELL\_SIZE = 50

MARGIN = 2

# BFS Algorithm

def bfs(grid, start, goal, draw, clock):

    rows, cols = len(grid), len(grid[0])

    queue = deque([(\*start, [])])

    visited = set()

    visited.add(start)

    path = []

    while queue:

        r, c, current\_path = queue.popleft()

        draw((r, c), GREY)  # Highlight current cell being explored

        path = current\_path + [(r, c)]

        # Wait for space bar press to continue to next step

        waiting\_for\_input = True

        while waiting\_for\_input:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    pygame.quit()

                if event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

                    waiting\_for\_input = False

        if (r, c) == goal:

            return path

        for dr, dc in DIRECTIONS:

            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 0 and (nr, nc) not in visited:

                visited.add((nr, nc))

                queue.append((nr, nc, path))

    return None

# DFS Algorithm

def dfs(grid, start, goal, draw, clock):

    rows, cols = len(grid), len(grid[0])

    stack = [(start, [])]

    visited = set()

    visited.add(start)

    path = []

    while stack:

        (r, c), current\_path = stack.pop()

        draw((r, c), GREY)  # Highlight current cell being explored

        path = current\_path + [(r, c)]

        # Wait for space bar press to continue to next step

        waiting\_for\_input = True

        while waiting\_for\_input:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    pygame.quit()

                if event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

                    waiting\_for\_input = False

        if (r, c) == goal:

            return path

        for dr, dc in DIRECTIONS:

            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 0 and (nr, nc) not in visited:

                visited.add((nr, nc))

                stack.append(((nr, nc), path))

    return None

# Greedy Best-First Search (GBFS) Algorithm

def gbfs(grid, start, goal, draw, clock):

    rows, cols = len(grid), len(grid[0])

    open\_list = [(0, start, [])]

    visited = set()

    visited.add(start)

    def heuristic(a, b):

        return abs(a[0] - b[0]) + abs(a[1] - b[1])  # Manhattan Distance

    path = []

    while open\_list:

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

        draw((r, c), GREY)  # Highlight current cell being explored

        path = current\_path + [(r, c)]

        # Wait for space bar press to continue to next step

        waiting\_for\_input = True

        while waiting\_for\_input:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    pygame.quit()

                if event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

                    waiting\_for\_input = False

        if (r, c) == goal:

            return path

        for dr, dc in DIRECTIONS:

            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 0 and (nr, nc) not in visited:

                visited.add((nr, nc))

                heapq.heappush(open\_list, (heuristic((nr, nc), goal), (nr, nc), path))

    return None

# A\* Algorithm

def a\_star(grid, start, goal, draw, clock):

    rows, cols = len(grid), len(grid[0])

    open\_list = [(0, 0, start, [])]  # f, g, node, path

    visited = set()

    visited.add(start)

    def heuristic(a, b):

        return abs(a[0] - b[0]) + abs(a[1] - b[1])  # Manhattan Distance

    path = []

    while open\_list:

        \_, g, (r, c), current\_path = heapq.heappop(open\_list)

        draw((r, c), GREY)  # Highlight current cell being explored

        path = current\_path + [(r, c)]

        # Wait for space bar press to continue to next step

        waiting\_for\_input = True

        while waiting\_for\_input:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    pygame.quit()

                if event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

                    waiting\_for\_input = False

        if (r, c) == goal:

            return path

        for dr, dc in DIRECTIONS:

            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 0 and (nr, nc) not in visited:

                visited.add((nr, nc))

                heapq.heappush(open\_list, (g + 1 + heuristic((nr, nc), goal), g + 1, (nr, nc), path))

    return None

# UCS Algorithm

def ucs(grid, start, goal, draw, clock):

    rows, cols = len(grid), len(grid[0])

    open\_list = [(0, start, [])]  # cost, node, path

    visited = set()

    visited.add(start)

    path = []

    while open\_list:

        cost, (r, c), current\_path = heapq.heappop(open\_list)

        draw((r, c), GREY)  # Highlight current cell being explored

        path = current\_path + [(r, c)]

        # Wait for space bar press to continue to next step

        waiting\_for\_input = True

        while waiting\_for\_input:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    pygame.quit()

                if event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

                    waiting\_for\_input = False

        if (r, c) == goal:

            return path

        for dr, dc in DIRECTIONS:

            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 0 and (nr, nc) not in visited:

                visited.add((nr, nc))

                heapq.heappush(open\_list, (cost + 1, (nr, nc), path))

    return None

# Beam Search Algorithm

def beam\_search(grid, start, goal, draw, clock, beam\_width=3):

    rows, cols = len(grid), len(grid[0])

    open\_list = [(0, start, [])]

    visited = set()

    visited.add(start)

    path = []

    while open\_list:

        open\_list = sorted(open\_list, key=lambda x: x[0])[:beam\_width]  # Select top `beam\_width` nodes

        next\_open\_list = []

        for \_, (r, c), current\_path in open\_list:

            draw((r, c), GREY)  # Highlight current cell being explored

            path = current\_path + [(r, c)]

            # Wait for space bar press to continue to next step

            waiting\_for\_input = True

            while waiting\_for\_input:

                for event in pygame.event.get():

                    if event.type == pygame.QUIT:

                        pygame.quit()

                    if event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

                        waiting\_for\_input = False

            if (r, c) == goal:

                return path

            for dr, dc in DIRECTIONS:

                nr, nc = r + dr, c + dc

                if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 0 and (nr, nc) not in visited:

                    visited.add((nr, nc))

                    next\_open\_list.append((abs(nr - goal[0]) + abs(nc - goal[1]), (nr, nc), path))

        open\_list = next\_open\_list

    return None

# Function to visualize the grid

def visualize(grid, start, goal, algorithm):

    pygame.init()

    rows, cols = len(grid), len(grid[0])

    width, height = cols \* CELL\_SIZE, rows \* CELL\_SIZE

    screen = pygame.display.set\_mode((width, height))

    pygame.display.set\_caption("Grid-Based Robot Navigation")

    clock = pygame.time.Clock()  # Control visualization speed

    def draw\_cell(pos, color):

        r, c = pos

        pygame.draw.rect(

            screen,

            color,

            [(MARGIN + CELL\_SIZE) \* c + MARGIN, (MARGIN + CELL\_SIZE) \* r + MARGIN, CELL\_SIZE, CELL\_SIZE]

        )

        pygame.display.flip()

    def draw\_grid():

        for row in range(rows):

            for col in range(cols):

                color = WHITE if grid[row][col] == 0 else RED  # Obstacles are red

                draw\_cell((row, col), color)

        draw\_cell(start, GREEN)  # Start position in green

        draw\_cell(goal, GREEN)  # Goal position in green

    # Draw initial grid

    draw\_grid()

    # Run the chosen algorithm

    path = algorithm(grid, start, goal, draw\_cell, clock)

    # Draw the final path

    if path:

        for r, c in path:

            draw\_cell((r, c), GREEN)  # Path is green

            clock.tick(60)

    else:

        print("No path found.")

    # Keep the window open until closed by the user

    running = True

    while running:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                running = False

    pygame.quit()

# Function to get user input for the grid and algorithm

def get\_user\_input():

    print("Welcome to Grid-Based Robot Navigation!")

    # Input grid size

    rows = int(input("Enter the number of rows: "))

    cols = int(input("Enter the number of columns: "))

    # Initialize grid with all free cells (0)

    grid = [[0 for \_ in range(cols)] for \_ in range(rows)]

    # Input number of obstacles

    num\_obstacles = int(input("Enter the number of obstacles: "))

    print("Enter the positions of obstacles (row and column, space-separated):")

    for \_ in range(num\_obstacles):

        r, c = map(int, input("Obstacle: ").split())

        if 0 <= r < rows and 0 <= c < cols:

            grid[r][c] = 1

        else:

            print("Invalid position, skipping!")

    # Input start and goal positions

    print("Enter the start position (row and column, space-separated):")

    start = tuple(map(int, input("Start: ").split()))

    print("Enter the goal position (row and column, space-separated):")

    goal = tuple(map(int, input("Goal: ").split()))

    # Validate inputs

    if not (0 <= start[0] < rows and 0 <= start[1] < cols) or grid[start[0]][start[1]] == 1:

        raise ValueError("Invalid start position!")

    if not (0 <= goal[0] < rows and 0 <= goal[1] < cols) or grid[goal[0]][goal[1]] == 1:

        raise ValueError("Invalid goal position!")

    # Choose algorithm

    print("Choose the algorithm to use:")

    print("1. BFS (Breadth-First Search)")

    print("2. DFS (Depth-First Search)")

    print("3. GBFS (Greedy Best-First Search)")

    print("4. A\* (A Star)")

    print("5. UCS (Uniform Cost Search)")

    print("6. Beam Search")

    choice = int(input("Enter the number corresponding to the algorithm: "))

    algorithms = {

        1: bfs,

        2: dfs,

        3: gbfs,

        4: a\_star,

        5: ucs,

        6: beam\_search

    }

    if choice not in algorithms:

        raise ValueError("Invalid algorithm choice!")

    return grid, start, goal, algorithms[choice]

# Main function

if \_\_name\_\_ == "\_\_main\_\_":

    try:

        grid, start, goal, algorithm = get\_user\_input()

        print(f"Visualizing {algorithm.\_\_name\_\_.replace('\_', ' ').title()} step by step...")

        visualize(grid, start, goal, algorithm)

    except Exception as e:

        print(f"Error: {e}")

**Info of the code :**

**Import Statements:**

import pygame

import heapq

from collections import deque

import time

1. pygame: This is a library used for creating video games and graphical applications. In this case, it's used for visualizing the grid and the search algorithms.
2. heapq: This module provides a way to implement priority queues (using heaps). It's used in algorithms like A\* and Greedy Best-First Search (GBFS).
3. deque: This is a data structure from the collections module that is optimized for fast appends and pops from both ends. It's used in BFS and DFS for storing and retrieving elements.
4. time: This module provides various time-related functions. In your code, it's imported but not used. You might want to remove it if it's unnecessary.

**Directions for Movement:**

DIRECTIONS = [(0, 1), (0, -1), (1, 0), (-1, 0)]

This list contains tuples that represent the four possible directions you can move on the grid:

* (0, 1) → Move right (increment the column)
* (0, -1) → Move left (decrement the column)
* (1, 0) → Move down (increment the row)
* (-1, 0) → Move up (decrement the row)

**Color Definitions:**

WHITE = (255, 255, 255) # Grid color

BLACK = (0, 0, 0) # Unused, but can be for the background

GREEN = (0, 255, 0) # Final point color and path color

RED = (255, 0, 0) # Obstacle color

BLUE = (0, 0, 255) # (Unused, but can be used for visualizing path)

GREY = (169, 169, 169) # Searching (current cell being explored)

These are color definitions in RGB format. Each color is represented as a tuple (R, G, B) where:

* WHITE = (255, 255, 255) is the color for free cells in the grid.
* RED = (255, 0, 0) represents obstacles.
* GREEN = (0, 255, 0) is used for the start and goal cells and also for the path.
* GREY = (169, 169, 169) is used to show the cells that are being explored during the algorithm.
* BLACK = (0, 0, 0) and BLUE = (0, 0, 255) are defined but unused.

**Grid Parameters:**

CELL\_SIZE = 50

MARGIN = 2

* CELL\_SIZE = 50: This determines the size of each cell in the grid (50 pixels by 50 pixels).
* MARGIN = 2: This is the space between cells (2 pixels).

**Search Algorithms:**

The next part of the code defines several search algorithms. Each algorithm performs a search to find a path from the start to the goal, considering obstacles. They use different strategies (BFS, DFS, A\*, etc.).

**BFS (Breadth-First Search) Algorithm:**

def bfs(grid, start, goal, draw, clock):

rows, cols = len(grid), len(grid[0])

queue = deque([(\*start, [])])

visited = set()

visited.add(start)

path = []

1. grid: 2D array representing the map (0 for free space, 1 for obstacles).
2. start and goal: These are tuples representing the start and goal positions on the grid.
3. queue = deque([(\*start, [])]): Initializes the queue with the start position and an empty path list.
4. visited = set(): A set to keep track of visited nodes (to avoid revisiting).
5. visited.add(start): Adds the start position to the visited set.

The rest of the function explores each cell in the grid using BFS:

* while queue:: The main loop continues until all cells have been explored.
* r, c, current\_path = queue.popleft(): Pop the front of the queue. This represents the current cell being explored, and current\_path is the path taken to get there.
* draw((r, c), GREY): Visually highlights the current cell being explored.
* waiting\_for\_input = True: The program pauses, waiting for the user to press the space bar before moving to the next step.
* if (r, c) == goal: If the current cell is the goal, return the path.
* for dr, dc in DIRECTIONS: Explore the neighbors (up, down, left, right) of the current cell.

**DFS (Depth-First Search) Algorithm:**

The DFS algorithm works similarly to BFS, but instead of exploring all neighbors at the current level before moving on, it explores one branch (or path) as deep as possible before backtracking.

def dfs(grid, start, goal, draw, clock):

stack = [(start, [])]

visited = set()

visited.add(start)

path = []

* stack: A list used to explore nodes in a depth-first manner (LIFO: Last-In, First-Out).
* The rest of the logic is similar to BFS but uses a stack instead of a queue.

**Greedy Best-First Search (GBFS) Algorithm:**

def gbfs(grid, start, goal, draw, clock):

open\_list = [(0, start, [])]

visited = set()

visited.add(start)

* In GBFS, the algorithm uses a heuristic function (Manhattan distance) to prioritize exploring the cells closer to the goal.
* open\_list = [(0, start, [])]: A list (priority queue) that stores nodes to be explored. The heuristic value (0 initially) and the current path are stored along with the node.
* heapq.heappop(open\_list): Pops the node with the lowest heuristic value.

**A\* Algorithm:**

def a\_star(grid, start, goal, draw, clock):

open\_list = [(0, 0, start, [])] # f, g, node, path

* This is a more advanced algorithm that combines both the cost to get to the node (g) and the heuristic estimate of the cost to reach the goal (h), which is f = g + h.
* The algorithm chooses nodes with the lowest f value to explore.

**UCS (Uniform Cost Search):**

def ucs(grid, start, goal, draw, clock):

open\_list = [(0, start, [])] # cost, node, path

* UCS is a search algorithm that explores the least-cost nodes first, where the cost is typically the number of steps or distance.

**Beam Search Algorithm:**

def beam\_search(grid, start, goal, draw, clock, beam\_width=3):

* Beam Search explores a fixed number of best nodes at each level, determined by the beam\_width. It’s a heuristic-based search, but only the top beam\_width nodes are expanded in each iteration.

**Grid Visualization:**

def visualize(grid, start, goal, algorithm):

pygame.init()

rows, cols = len(grid), len(grid[0])

width, height = cols \* CELL\_SIZE, rows \* CELL\_SIZE

screen = pygame.display.set\_mode((width, height))

pygame.display.set\_caption("Grid-Based Robot Navigation")

* Initializes the Pygame window with a size based on the grid dimensions (CELL\_SIZE).
* draw\_cell: This function draws a single cell at the specified position with a specific color.

**Drawing the Grid:**

def draw\_grid():

for row in range(rows):

for col in range(cols):

color = WHITE if grid[row][col] == 0 else RED

draw\_cell((row, col), color)

draw\_cell(start, GREEN)

draw\_cell(goal, GREEN)

* This function draws all the cells in the grid: white for free cells and red for obstacles. It also draws the start and goal cells in green.

**Input Handling:**

def get\_user\_input():

print("Welcome to Grid-Based Robot Navigation!")

rows = int(input("Enter the number of rows: "))

cols = int(input("Enter the number of columns: "))

grid = [[0 for \_ in range(cols)] for \_ in range(rows)]

* This function collects user input for the grid dimensions, obstacles, start and goal positions, and the algorithm to use.

**Main Loop:**

if \_\_name\_\_ == "\_\_main\_\_":

try:

grid, start, goal, algorithm = get\_user\_input()

visualize(grid, start, goal, algorithm)

except Exception as e:

print(f"Error: {e}")

* This is the main entry point. It calls get\_user\_input() to get the necessary data, then visualizes the grid and runs the selected algorithm.

**Summary:**

This code creates a grid-based robot navigation simulation where you can choose a search algorithm (BFS, DFS, A\*, UCS, GBFS, Beam Search) and visualize its progress step by step. The grid allows you to place obstacles, set a start and goal position, and run the algorithm to find a path while showing each step of the search process on the screen  
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
  
  
In general, an **open list** is a conceptual structure used in search algorithms to manage and keep track of nodes (or states) that are yet to be explored. It's essentially a **queue** or **list** of possibilities that the algorithm considers for further processing.

**General Definition:**

An **open list** is a collection of "active" nodes or states that:

* **Have been discovered** (or generated).
* **Need further exploration** (or processing).
* Serve as the "frontier" of the search space.

**Key Features of an Open List:**

1. **Dynamic Nature**:
   * Nodes are continually added to and removed from the open list as the algorithm progresses.
2. **Exploration Frontier**:
   * The open list acts as the "boundary" of the area being explored, containing nodes that are connected to the explored region but not yet fully processed.
3. **Priority**:
   * In most cases, nodes in the open list are prioritized based on a specific criterion, such as cost, heuristic value, or depth, depending on the algorithm.
4. **Iterative Process**:
   * The algorithm repeatedly takes nodes from the open list, processes them, and adds new nodes generated from those already explored.

**Practical Examples of Open Lists:**

1. **In Pathfinding**:
   * Nodes in the open list represent locations in a grid that are waiting to be explored.
   * Examples:
     + A\* algorithm: Nodes are prioritized by the sum of their cost to reach and estimated cost to the goal.
     + BFS: Nodes are processed in the order they are discovered (FIFO).
     + UCS: Nodes are processed by their path cost (lowest cost first).
2. **In AI and Search Problems**:
   * Open lists manage the states of a problem space:
     + States that are "open" for further exploration.
     + States can represent partial solutions, configurations, or choices that need to be completed or evaluated.
3. **In Graph Traversal**:
   * Nodes represent vertices in a graph yet to be processed.

**Key Operations on an Open List:**

1. **Add a Node**:
   * When a new node is discovered, it's added to the open list for later processing.
2. **Remove a Node**:
   * The algorithm removes the "most promising" node from the open list based on its specific prioritization logic.
3. **Check Membership**:
   * Sometimes, algorithms check if a node is already in the open list to avoid duplicates.

**Example Analogy:**

Imagine you are navigating through a maze:

* The **open list** is like a collection of paths or intersections that you know exist but haven't explored yet.
* As you explore, you add new intersections (nodes) to your open list and pick the next one to investigate based on your strategy (e.g., nearest to the goal, least traveled, etc.).

In essence, the open list is a fundamental part of any search or traversal process, managing the "to-do" list of the algorithm's exploration efforts.