APCSP_findpath.py

```
import math
import sys
import time
from heapq import heapify, heappop, heappush
from typing import List
import pygame # a python library for creating a grid game

class Point:
    def __init__(self, x: int, y: int, dist_to_start: int = 0, dist_to_end: int = 0,
visited: bool = False, parent: int = None):
        self.x = x
        self.y = y
        self.dist_to_start = dist_to_start
        self.dist_to_end = dist_to_end
        self.visited = visited
        self.parent = parent

    # Compare the total distance of the Point with the total distance of another Point
    def __lt__(self, other):
        return (self.dist_to_start + self.dist_to_end) < (self.dist_to_start +
other.dist_to_end)


def neighbors(p: Point, obstacles: set, N: int, M: int) -> list:
    '''
    neighbors finds the coordinates of all eligable points next to p.
    :param p: Input point
    :param obstacles: Set of points that can not be visited
    :param N: the number of rows of the grid
    :param M: the number of columns of the grid
    :return: List of neighbor points that can be visited.
    '''

    # direction is a list of all possible moving directions from current position
    # each element is a tuple of two integers: (i, j), where i and j can be -1, 0, or 1
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (1, -1), (-1, 1)]

    good_neighbors = []  # a list of accessible points next to p

    # iterate through eight directions
    for d in directions:
        # calculate neighbor's coordinates
        x, y = p.x + d[0], p.y + d[1]
        # Skip if the neighbor in an obstacle or out of boundary
        if (x, y) in obstacles or x <= 0 or y <= 0 or x >= N or y >= M:
            continue
        # otherwise append to the list
        good_neighbors.append((x, y))

    return good_neighbors


def retrace_steps(p: Point) -> list:
```

```
    '''
    retrace_steps starts from the end and finds the path back to the start.
    :param p: The end point of the path.
    :return: The path from the start point to the end point.
    '''

    current, path = p, []
    while current:
        path.append((current.x, current.y))
        current = current.parent
    return path[::-1]


def navigate(start: tuple, end: tuple, obstacles: set, N: int, M: int) -> list:
    '''
    navigate implements the A* Algorithm to find the shortest path between two points.
    :param start: coordinates of the start point of the path
    :param end: coordinates of the end point of the path
    :param obstacles: Set of points that cannot be used in the path
    :param N: the number of rows of the grid
    :param M: the number of columns of the grid
    :return: The path from the start point to the end point or [] if not path can be
found.
    '''

    start_point = Point(start[0], start[1], dist_to_start = 0, dist_to_end =
math.dist(start, end))
    # Candidates for checking the next stop. We use a heap to optimize finding the point
with the least distance.
    candidates = [start_point]
    # Dictionary of (x, y) tuple to a Point record in path finding
    point_dict = {start: start_point}

    while candidates:
        current = heappop(candidates)
        current.visited = True

        # If end is successfully reached, then return the path
        if current.x == end[0] and current.y == end[1]:
            return retrace_steps(current)

        my_neighbors = neighbors(current, obstacles, N, M)
        for n in my_neighbors:
            # Neighbors that have been seen
            if n in point_dict:
                p = point_dict[n]
                # Ignores neighbors that have been considered
                if p.visited:
                    continue

                # Update the neighbor point if using the current point as a stop offers
a better dist_to_start
                offered_dist_to_start = current.dist_to_start + math.dist(n, (current.x,
current.y))
                if offered_dist_to_start >= p.dist_to_start:
                    continue
                p.dist_to_start = offered_dist_to_start
```

```
                p.dist_to_end = math.dist(n, end)
                p.parent = current

                # Reorganizes the candidates heap because this neighbor point has a new
total distance
                heapify(candidates)

            # Brand new neighbors
            else:
                p = Point(
                    n[0],
                    n[1],
                    dist_to_start = math.dist((current.x, current.y), n),
                    dist_to_end = math.dist(n, end),
                    parent = current
                )

                point_dict[(p.x, p.y)] = p
                heappush(candidates, p)

    # Return an empty list if no path is found
    return []


def find_path(waypoints: list, obstacles: list, N: int, M: int) -> List[list]:
    '''
    Calls navigate to get shortest path segments between every two waypoints
    :param waypoints: list of Points that need to be visited in order
    :param obstacles: list of Points that are obstacles and can not be visited
    :param N: the number of rows of the grid
    :param M: the number of columns of the grid
    :returns: A list of lists of the shortest path between every two waypoints in order
              If no path found between two waypoints, stop and returns the list of
shortest
              paths to the last accessible waypoint
    '''

    final_path = [] # a list of lists of path points

    for i in range(len(waypoints) - 1):

        # call navigate to find the shortest path between the current and the next
waypoint
        path_segment = navigate(waypoints[i], waypoints[i + 1], obstacles, N, M)

        # if path found, appends to the final path
        if path_segment:
            final_path.append(path_segment)

        # appends an empty path and return final_path if no path can be found for this
segment
        else:
            final_path.append([])
            break

    return final_path
```

```python
def grid_to_obstacles(grid: List[list], N: int, M: int) -> list:
    '''
    scans every point on the grid, puts obstacles into a list and returns the list
    :param grid: a matrix (a list of lists of integers) of size N * M,
                where each cell is 0 - empty, 1 - waypoint, or 2 - obstacle
    :param N: number of rows of the grid
    :param M: number of columns of the grid
    :returns: obstacles: a list of every point on the grid that is an obstacle
    '''
    obstacles = []
    for r in range(N):
        for c in range(M):
            if grid[r][c] == 2:
                obstacles.append((r, c))
    return obstacles


def update_grid(grid: List[list], waypoints: list, row: int, column: int, last_key: str)
-> None:
    '''
    draw sets a point on the grid to a waypoint, an obstacle, or an empty grid space
    :param grid: the matrix of values on grid from gui
    :param waypoints: the list of ordered waypoints
    :param row: The row the point is on
    :param column: The column the point is on
    :param last_key: The last key that was pressed and what determines what the point
should be
    :returns: None
    '''
    if last_key == "a":  # add waypoint
        waypoints.append((row, column))
        grid[row][column] = 1
    elif last_key == "d":  # delete waypoint or obstacle
        if grid[row][column] == 1:
            waypoints.remove((row, column))
        elif grid[row][column] == 2:
            # reset all waypoints
            for (x, y) in waypoints:
                grid[x][y] = 1
        grid[row][column] = 0
    elif last_key == "s":  # add obstacle
        if grid[row][column] == 1:
            waypoints.remove((row, column))
        grid[row][column] = 2


def update_path_on_grid(grid: List[list], path_list: List[list], waypoints: list) ->
None:
    ''' Update points in the path on the grid '''

    # iterate through path for each segment
    for i in range(len(waypoints) - 1):
        path = path_list[i]

        # mark destination point as display no route for this segment and stop
        if len(path) == 0:
```

```
            (x, y) = waypoints[i+1]
            grid[x][y] = 4
            return

        # otherwise iterate through each point in the path to mark as route if not a
waypoint
        for p in path:
            if p not in waypoints:
                (x, y) = p
                grid[x][y] = 3


def draw_grid(screen, grid: List[list], waypoints: list, N: int, M: int, width: int,
height: int, margin: int) -> None:
    ''' Draw grid using pygame on the screen  '''

    # Define point types in a list, use element index as grid value
    point_types = ['border', 'waypoint', 'obstacle', 'route', 'route_unavailable']

    # Define colors
    colors = {
        'background' : (10, 10, 40),
        'border' : (30, 30, 60),       # 0
        'waypoint' : (0, 255, 0),      # 1
        'obstacle' : (255, 255, 255),  # 2
        'route' : (143, 212, 242),     # 3
        'route_unavailable' : (232, 66, 51)  # 4
    }

    # Fill the background color
    screen.fill(colors['background'])

    # Draw grid cells
    for row in range(N):
        for column in range(M):
            cell = grid[row][column]
            color = colors[point_types[cell]]
            rect = [(margin + width) * column + margin, (margin + height) * row +
margin, width, height]
            pygame.draw.rect(screen, color, rect)


'''
The code below used to make the grid referenced the stack overflow forem
https://stackoverflow.com/questions/33963361/how-to-make-a-grid-in-pygame
'''

def main(N: int, M: int) -> None:
    '''
    Main procedure utilizes python pygame library to create the finding path game
    The waypoints and obstacles are obtained from user inputs using pygame.
    :param N: the number of rows of the grid
    :param M: the number of columns of the grid
    '''

    # Initialize variables
    width, height, margin = 20, 20, 1 # cell and margin size for drawing
```

```
    WINDOW_SIZE = [M * (width + margin), N * (height + margin)]  # set window size
    grid = [[0 for i in range(M)] for j in range(N)]  # init grid with all zeros
    waypoints = []  # init the list of waypoints to empty
    last_grid_changed = (-1, -1) # init last grid changed
    last_key = "a"  # init key to "a" for waypoints

    # Initialize pygame
    pygame.init()
    screen = pygame.display.set_mode(WINDOW_SIZE)
    key_str = "'a' = waypoint, 's' = obstacle, 'd' = erase, 'f' = start, 'backspace' = clear"
    pygame.display.set_caption(f"Path finding: {key_str}")
    clock = pygame.time.Clock()

    # loop through user events until user quits
    while True:
        for event in pygame.event.get():
            # Handle key presses
            if event.type == pygame.KEYDOWN:

                # Change tool based on key press
                if event.key == pygame.K_a:
                    last_key = "a"
                elif event.key == pygame.K_s:
                    last_key = "s"
                elif event.key == pygame.K_d:
                    last_key = "d"

               # Find path when 'f' is pressed
                elif event.key == pygame.K_f:
                    if len(waypoints) <= 1:
                        print("Can't do that, you need more than one waypoint!")
                    else:
                        # get obstacles list from the grid
                        obstacles = grid_to_obstacles(grid, N, M)

                        # call find_path to find the shortest path
                        path = find_path(waypoints, obstacles, N, M)
                        print (f"waypoints = {waypoints}\nobstacles = {obstacles}\npath = {path}")

                        # update each path points to the grid
                        for i in range(len(path) - 1):
                            update_path_on_grid(grid, path[:i+1], waypoints[:i+2])
                            draw_grid(screen, grid, waypoints[:i+2], N, M, width, height, margin)
                            pygame.display.flip()
                            time.sleep(.09)

                        update_path_on_grid(grid, path, waypoints)

                # Clear grid when 'backspace' is pressed
                elif event.key == pygame.K_BACKSPACE:
                    grid = [[0 for i in range(M)] for j in range(N)] # reset grid to all zeros
                    waypoints.clear()
```

```
            # Handle mouse clicks
            elif pygame.mouse.get_pressed()[0]:
                pos = pygame.mouse.get_pos()
                column = pos[0] // (width + margin)
                row = pos[1] // (height + margin)

                # update grid if the grid cell is changed
                if (row, column) != last_grid_changed:
                    last_grid_changed = (row, column)
                    update_grid(grid, waypoints, row, column, last_key)

            # End the game when user quits
            elif event.type == pygame.QUIT:
                pygame.quit()
                print("Thanks for Playing!")
                sys.exit(0)

        draw_grid(screen, grid, waypoints, N, M, width, height, margin)

        clock.tick(60)
        pygame.display.flip() # Update the display

if __name__ == "__main__":
    main(N=40, M=60)
```