MINI PROJECT2

CCAI-221

MazeSolver

Student name	Student ID
Mohammed Allagany	
Khalid Nimri	
Aseel Suhail	

Introduction:

This code implements the A* search algorithm in Pygame to find the path between a start and end cell in a 2D grid. The grid is represented by a matrix of "Cell" objects, each with a color that indicates its status (start, end, blocked, unvisited, open, closed, or path). The A* algorithm works by using a priority queue to find the lowest cost path between the start and end cells, taking into consideration the "heuristic function" (Manhattan distance) to estimate the cost of moving to the end cell. The algorithm checks the neighbors of each cell and updates the cost and parent cell until the end cell is reached, at which point the path is traced back using the "came_from" dictionary. The cells are drawn on the Pygame window, showing the progression of the algorithm.

Goal and strategies of:

The main goal is to arrive to the goal in the shortest path available, using heuristic function by measuring the cost of the available options and going through the most valuable and optimized path.

The best algorithm for such cases is the A* search algorithm which we used to find the optimal solution, also it helped with avoiding the obstacles which made randomly by the user.

Code

```python
# ----------initializing window and Constants----------
import pygame
import math
from queue import PriorityQueue

WIDTH = 468  # the width of the window
WIN = pygame.display.set_mode(
    (WIDTH, WIDTH))  # initializing the window with size of WIDTH * WIDTH ; not using height for simplicity
pygame.display.set_caption("A* search algorithms to find the path")

RED = (255, 0, 0)  # is end cell
GREEN = (0, 255, 0)  # is start cell
BLUE = (0, 0, 255)  # is closed cell
WHITE = (255, 255, 255)  # unvisited cell
BLACK = (0, 0, 0)  # blocked or barrier cell
PURPLE = (128, 0, 128)  # is Path cell
GREY = (128, 128, 128)  # grid line color


# ---------------defining Class for cells in the maze----------

class Cell:
    def __init__(self, row, col, width, total_rows):
        self.row = row
        self.col = col
        self.x = row * width
        self.y = col * width
        self.color = WHITE
        self.neighbors = []
        self.width = width
        self.total_rows = total_rows

    def get_coordinate(self):
        return self.row, self.col

    def is_closed(self):
        return self.color == BLUE

    def is_barrier(self):
        return self.color == BLACK

    def is_start(self):
        return self.color == GREEN

    def make_start(self):
        self.color = GREEN

    def is_end(self):
        return self.color == RED

    def reset(self):
        self.color = WHITE

    def make_closed(self):
        self.color = BLUE
```

```python
        self.color = BLUE

    def make_barrier(self):
        self.color = BLACK

    def make_end(self):
        self.color = RED

    def make_path(self):
        self.color = PURPLE

    def draw(self, win):
        pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.width))

    """
    ----------finding cell neighbors while ignoring Blocked ones----------
    example: cell row= 49 , total_rows = 50 , checking down neighbors
        if self.row < self.total_rows - 1  ; the first condition will be False because there is no row under 49 the rows is from 0-49
        and ; if the first condition is True will check the second
        not grid[self.row + 1][self.col].is_barrier(): the second condition will be False if the color of the cell Black meaning it is blocked
    """

    def update_neighbors(self, grid):
        self.neighbors = []
        if self.row < self.total_rows - 1 and not grid[self.row + 1][
            self.col].is_barrier():  # DOWN checking if there is a down neighbor and not blocked
            self.neighbors.append(grid[self.row + 1][self.col])

        if self.row > 0 and not grid[self.row - 1][
            self.col].is_barrier():  # UP checking if there is a up neighbor and not blocked
            self.neighbors.append(grid[self.row - 1][self.col])

        if self.col < self.total_rows - 1 and not grid[self.row][
            self.col + 1].is_barrier():  # RIGHT checking if there is a right neighbor and not blocked
            self.neighbors.append(grid[self.row][self.col + 1])

        if self.col > 0 and not grid[self.row][
            self.col - 1].is_barrier():  # LEFT checking if there is a left neighbor and not blocked
            self.neighbors.append(grid[self.row][self.col - 1])

    def __lt__(self, other):
        return False


"""
defining the heuristic_function which is manhattan distance
    it is h = |x1 - x2| + |y1 - y2|
"""


def heuristic_function(p1, p2):
    x1, y1 = p1  # p1 will be (row , col)
    x2, y2 = p2  # p2 will be (row , col)
    return abs(x1 - x2) + abs(y1 - y2)
```

Explanation

First, the Pygame library is imported and a window with a size of 468x468 is created. The window caption is set to "A* search algorithms to find the path".

Then, various colors used in the maze are defined (RED, GREEN, BLUE, WHITE, BLACK, PURPLE, ORANGE, and GREY) to represent the end cell, start cell, closed cell, unvisited cell, blocked or barrier cell, Path cell, open cell, and grid line color respectively.

Next, a class "Cell" is defined to represent each cell in the maze. The class has various attributes such as the cell's row, column, position (x and y), color, neighbors, width, and total number of rows. It also has various functions such as to check if the cell is closed, open, barrier, start, or end. It also has functions to change the cell's color (make_closed, make_open, make_barrier, make_end, make_path) and to draw the cell (draw). The "update_neighbors" function updates the cell's neighbors list by checking if the cells surrounding it are not barriers.

Finally, a heuristic function called "heuristic_function" is defined, which calculates the manhattan distance between two cells (p1 and p2) by computing |x1 - x2| + |y1 - y2|.

```
    x1, y1 = p1  # p1 will be (row , col)
    x2, y2 = p2  # p2 will be (row , col)
    return abs(x1 - x2) + abs(y1 - y2)



"""
drawing the path from end to start using came_from dictionary
    {start:p5 , p5:p6 , p6:p9 , p9:end}
"""


def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        if current.color != GREEN:
            current.make_path()  # will change the color to purple
        draw()  # will draw the new color in the window


# -------------A* algorithm----------------------

def algorithm(draw, grid, start, end):
    count = 0  # this count can tell us which cell visited first
    open_set = PriorityQueue()  # the open set is Priority Queue will add every cell we visit
    # and take out the one with least h function and least count number
    open_set.put((0, count, start))  # add the start cell with f = 0 , count = 0 , start = starting cell
    came_from = {}  # this dictionary just to save the cell we came from to draw the path at the end
    g_score = {cell: float("inf") for row in grid for cell in
               row}  # set all the g value of the cells to infinity at the beging
    g_score[start] = 0  # the g value of the start coell to 0
    f_score = {cell: float("inf") for row in grid for cell in
               row}  # set all the f value of the cells to infinity at the beging
    f_score[start] = heuristic_function(start.get_coordinate(),
                                        end.get_coordinate())  # calculte the f value of the start cell
    # the distance from the start to the end cell
    # f= g value + heuristic_function

    open_set_hash = {start}  # this open set is the same open set but it is hashed
    # so we can search through the set to see if the cell has been visited or not
    # we can not search through PriorityQueue open set
    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:  # to stop the algorithm from the keyboard
                pygame.quit()

        current = open_set.get()[2]  # to pop the lowest cell with least h function and least count number
        open_set_hash.remove(current)  # to synchronize the hash open set with PriorityQueue open set
        # because they are the same thing

        if current == end:
            reconstruct_path(came_from, end, draw)  # call reconstruct_path method to draw the path at the end
            end.make_end()  # change the color of the cell to RED
            return True  # to stop the algorithm

        for neighbor in current.neighbors:  # loop to visit all cell neighbors , every cell has array of it's neighbors
            temp_g_score = g_score[current] + 1  # calculate the g value for all neighbors
            # the g value is how far the cell from the start cell
            # it is always increasement by 1
            if temp_g_score < g_score[neighbor]:  # to update the g value and f value of the neighbor
                came_from[neighbor] = current  # add the cell to the dictionary example: {neighbor: start}
                g_score[neighbor] = temp_g_score  # update the g value
                f_score[neighbor] = temp_g_score + heuristic_function(neighbor.get_coordinate(), end.get_coordinate())
                # update the f value , f= g value + heuristic_function
                if neighbor not in open_set_hash:  # if the cell is not in the open cell
                    count += 1  # increasented by 1
                    open_set.put(
                        (f_score[neighbor], count, neighbor))  # add the cell to open set with (f , count ,cell)
                    open_set_hash.add(neighbor)  # to synchronize the hash open set with PriorityQueue open set


        draw()  # draw the cell to apply changes to colors in the window

        if current != start:
            current.make_closed()  # change the color of the cell to BLUE
            # the cell has been closed

    return False  # to stop the algorithm


def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            cell = Cell(i, j, gap, rows)
            grid[i].append(cell)

    return grid


def draw_grid_lines(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, GREY, (0, i * gap), (width, i * gap))
    for j in range(rows):
        pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, width))


def draw(win, grid, rows, width):
    win.fill(WHITE)

    for row in grid:
        for cell in row:
            cell.draw(win)

    draw_grid_lines(win, rows, width)
    pygame.display.update()


def get_clicked_pos(mouse_position, rows, width):
```

Cell

Problems Terminal Services

pre-built Python packages shared indexes // Always download // Download once // Don't show again // Configure... (2 minutes ago)

50:1 CRLF UTF-8 4

23°C Clear

Cell

Problems Terminal Services

pre-built Python packages shared indexes // Always download // Download once // Don't show again // Configure... (4 minutes ago)

50:1 CRLF UTF-8 4 spaces Python 3.9 (pythonProject)

23°C Clear ENG 8:00 PM 2/8/2023

Explanation

The code is an implementation of A* algorithm, a pathfinding algorithm used to find the shortest path between two points on a grid.

The code takes a grid and the starting and ending cells as inputs and applies the algorithm to find the path from the start to end.

The reconstruct_path function draws the path by traversing the came_from dictionary, which stores the parent cell of each cell in the path.

The algorithm uses a priority queue to keep track of the cells to be visited, with the priority being the sum of the cost (g-value) and the estimated remaining cost (h-value) to the end.

The code also updates the g-value and f-value of each cell based on the cost to reach that cell and the estimated cost to reach the end from that cell.

The code also visualizes the grid and the path in a window using pygame.

```
    draw_grid_lines(win, rows, width)
    pygame.display.update()


def get_clicked_pos(mouse_position, rows, width):
    gap = width // rows
    x, y = mouse_position

    row = x // gap
    col = y // gap

    return row, col


def main(win, width):
    ROWS = 13
    grid = make_grid(ROWS, width)

    start = None
    end = None

    run = True
    while run:
        draw(win, grid, ROWS, width)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            if pygame.mouse.get_pressed()[0]:  # LEFT
                mouse_position = pygame.mouse.get_pos()
                row, col = get_clicked_pos(mouse_position, ROWS, width)
                cell = grid[row][col]
                if not start and cell != end:
                    start = cell
                    start.make_start()

                elif not end and cell != start:
                    end = cell
                    end.make_end()

                elif cell != end and cell != start:
                    cell.make_barrier()

            elif pygame.mouse.get_pressed()[2]:  # RIGHT
                mouse_position = pygame.mouse.get_pos()
                row, col = get_clicked_pos(mouse_position, ROWS, width)
                cell = grid[row][col]
                cell.reset()
                if cell == start:
                    start = None
                elif cell == end:
                    end = None

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE and start and end:
                    for row in grid:
```

Cell

Problems Terminal Services

pre-built Python packages shared indexes // Always download // Download once // Don't show again // Configure... (4 minutes ago)

```
    start = None
    end = None

    run = True
    while run:
        draw(win, grid, ROWS, width)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            if pygame.mouse.get_pressed()[0]:  # LEFT
                mouse_position = pygame.mouse.get_pos()
                row, col = get_clicked_pos(mouse_position, ROWS, width)
                cell = grid[row][col]
                if not start and cell != end:
                    start = cell
                    start.make_start()

                elif not end and cell != start:
                    end = cell
                    end.make_end()

                elif cell != end and cell != start:
                    cell.make_barrier()

            elif pygame.mouse.get_pressed()[2]:  # RIGHT
                mouse_position = pygame.mouse.get_pos()
                row, col = get_clicked_pos(mouse_position, ROWS, width)
                cell = grid[row][col]
                cell.reset()
                if cell == start:
                    start = None
                elif cell == end:
                    end = None

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE and start and end:
                    for row in grid:
                        for cell in row:
                            cell.update_neighbors(grid)

                    algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)

                if event.key == pygame.K_c:
                    start = None
                    end = None
                    grid = make_grid(ROWS, width)

    pygame.quit()


main(WIN, WIDTH)
```

Cell

Problems Terminal Services

pre-built Python packages shared indexes // Always download // Download once // Don't show again // Configure... (4 minutes ago)

50:1 CRLF UTF-8 4 spaces Python 3.9 (pythonProject)

23°C Clear ENG 8:00 PM 2/8/2023

Explanation

The code is a script written in Python and utilizing the Pygame library.

It creates a grid of cells and allows the user to interact with the grid by clicking and holding the left mouse button to make cells a barrier, clicking and holding the right mouse button to reset cells, and pressing the spacebar to run an algorithm that finds the shortest path between the start and end cells (if both have been designated). The function "main" sets the number of rows in the grid, creates the grid, sets the starting and ending cells, and sets up an event loop to listen for user input. The "run" variable is used to control the event loop, and when it is set to False, the event loop will terminate, and Pygame will be quit.

The event loop listens for quit events, mouse button presses, and key presses.

If the left mouse button is pressed, the position of the mouse is retrieved, and the corresponding cell in the grid is set as the start, end, or barrier cell. If the right mouse button is pressed, the corresponding cell is reset. If the spacebar is pressed, the algorithm is run to find the shortest path. If the "c" key is pressed, the grid is cleared, and the start and end cells are reset.

Output:

User manual

- You can define the starting and goal points by the (Left click).
- You can modify your picks by the (Right click).
- You can start the program by the (Space button).
- You can clear the program by pressing (C key)
- The green color is the starting point.
- The red color is the goal.
- The black color is the boundaries you made.
- The purple color is the shortest path to the goal.
- The blue color is a failed attempt to find the goal.

Conclusion and comments:

Mostly through the design and implementation of the code, we have learnt a lot about the benefit of the search algorithms, and how it specifically works, without forgetting to mention the implementation, and how to seize every feature it provides.

The implementation of the GUI was a lot easier using the Pygame library, because of the functions it provides.

Last but not least, we noticed that the AI implementation in python was easier than the other high-level languages thanks to the libraries provided by the (Python Software Foundation)

References:

- https://www.geeksforgeeks.org/a-search-algorithm/
- https://www.pythonpool.com/a-star-algorithm-python/
- https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/a-star-search-algorithm/
- https://www.geeksforgeeks.org/python-drawing-different-shapes-on-pygame-window/?ref=rp