EARIN MINI-PROJECT 1

solves a maze using A* algorithm

Authors: Das Devansh

Xin Yang

Supervisor: Dr. Marczak Daniel

Date: 2023/3/19

Introduction:

The A* algorithm could be used for finding the shortest path between two points in a graph. In this lab, we implemented the A* algorithm to solve mazes. The algorithm uses a heuristic function to guide its search, which makes it more efficient than other search algorithms, which is characterized by the definition of valuation function. For general heuristic graph search, the node with the lowest f value of the evaluation function is always selected as the extension node. Therefore, f estimates nodes from the point of view of finding a path of minimum cost. Therefore, the value of the evaluation function of each node n can be considered as two components: the actual cost from the start node to node n and the estimated cost from node n to the target node.

Method:

The general process of A* algorithm is described as follows:
step1: Set the starting point S, the end point E, the mapsize, and the obstacle set Blocklist
step2: Add starting point S to the open list Openlist
step3: Calculate the objective function f(x) of S
step4: Find the node with minimum f(x) in Openlist, denoted as N_{min}
step5: Remove N_{min} from Openlist and add Nmin to closed list Closelist
step6: Find all neighbor nodes of Nmin to get the set Nlist
step7: For each element N in Nlist, loop as follows:
step8: if N belongs to Closelist or N belongs to Blocklist:
step9: Skip the node
step10: if N does not belong to Openlist:
step11: Add N to Openlist
step12: Set the parent of node N to N_{min}
step13: Calculate the objective function f(x) of node N
step14: else if N belongs to Openlist:
step15: Node N has computed f(x) before, denote the original f(x) as f(x)_old
step16: Compute the new objective function f(x)_new for node N with Nmin as parent
step17: if f(x)_new < f(x)_old:
step18: Set the parent of node N to Nmin
step19: And update the objective function f(x) = f(x)_new of node N
step20: Turn to step7, loop through all neighbor nodes in the list Nlist
step21: Proceed to step4 until Openlist is empty or Nmin is equal to the end point E

We have implemented the A* algorithm in Python and used it to solve the maze problem.

First, we read the maze from a text file and stored it in a 2D list. We then defined the start and end positions in the maze using a position function.

We use a heuristic function to estimate the cost of reaching the goal from the current position.

At this point, we have two different heuristic functions to choose from to calculate the distance between two points: the Manhattan distance or the Euclidean distance. We then use the A* algorithm to find the shortest path between the start and end positions.

We also define a print_maze function to print the maze and the path to the goal. This function takes the maze and a set of positions as input (the icon for the start position is ✘ and the icon until the end is ✓) and prints the maze and the path to the goal marked with the "x" character, the explored area with o, the available space with □ and the wall marked with ■

The main components are as follows：

1. Import the necessary libraries,
    - the code uses the sleep function from the time library to control the display time of the maze
    - the ndarray class from the numpy library to store a two-dimensional array of the maze
    - the heap data structure from the heapq library to manage the open set of the A* algorithm.
    - The clear_output function from the IPython.display library is also used to clear the output and achieve a real-time refresh of the maze state.
2. 2. Define the wall and space symbols in the maze file and set the display time and the different heuristic algorithm [H(n)]. We also define a number of different mazes, including the coordinates of the start and end points.
3. Define node classes that store the coordinates, g-values(the cost of the path from the start node to the current node), h-values(the estimated cost of the path from the current node to the goal node), f-values(the sum of the g-value and the h-value), and parent nodes of a maze lattice.
4. Define a function to read the maze file, read the maze information from the file, and check whether it is legal.
5. Defines a position, converts a coordinate to a node object and checks that the node is valid (i.e. in a maze and not a wall). The function first

extracts the x and y coordinates from the coordinates argument and then checks that these coordinates are within the maze boundary and that the node pointed to is not a wall.

6. Define two different computational heuristic functions which calculate the heuristic estimate between two nodes using either the Manhattan distance or the Euclidean distance.
7. Define a function to display the maze, which is used to dynamically display the direction of the maze with different symbol

```
    START_SYMBOL = "✘"
    END_SYMBOL = "✔"
    PATH_SYMBOL = "x"
    CHECKED_SYMBOL = "o"
    OPEN_SPACE_SYMBOL = "□"
    WALL_SYMBOL = "■"
```

8. main part

   Define the main algorithm to find the shortest path from the start to the end point in a maze.

   It takes as input a maze (represented as a NumPy array), a start point, and an end point, and returns a path from the start point to the end point (if one exists).

- start and goal are Node objects for the start and end points, respectively.
- display_time controls how quickly the maze is displayed while the algorithm is running.
- h_type specifies the type of the heuristic function, which can be a Manhattan distance or a Euclidean distance.

The specific algorithm is implemented as follows:

1. First, the starting point is added to the open list, which represents the node to explore. Then, loop through the following steps:
2. Pop the node with the lowest F value from the open list, mark it as explored, and add it to the checked set.
3. If the current node is the end point, it is marked as part of the path, and the final maze matrix and path are output.
4. Otherwise, the children of the current node are generated and added to the children list.
5. Loop over the children in the children list and skip them if they have already been explored. Otherwise, calculate the value of the evaluation function of the child to the end point, update the F value of the child, add it to the open list, and add it to the checked set.
6. Until the open list is empty, or an end point is found, the algorithm stops and returns. The final output path is a sequence of node coordinates in the form of tuples, and the correct path order needs to be output in reverse order.

Steps:

1. Initialize the start and end nodes
2. Create an empty checked set and a to-check list containing only the starting point.
3. The node with the smallest F-value was selected as the current node, and it was removed from the to-be-checked list and added to the checked set.
4. If the current node is the destination, then the path is returned.
5. Otherwise, the neighbor nodes in the four directions of the current node are traversed, if the neighbor node has not been checked, its g value, h value and f value are calculated, and it is added to the list to be checked.
6. Read the maze file, get the starting point and the ending point, and call the A* algorithm function to solve the maze problem.

Differences between two heuristic functions:

- Using the Manhattan distance as a heuristic function, we traversed fewer search nodes, but when we used the Euclidean distance as a heuristic function, it explored far more nodes than we expected, the reason being that the Euclidean distance needs to be combined with an diagonal move when used as an A* algorithm
- We also tried increasing the value of H and concluded that increasing this value would speed up the search

Examples are given below:

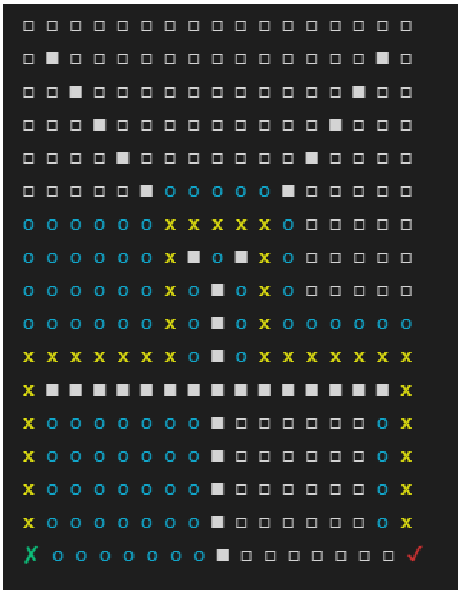

Figure 1. Manhattan for maze5

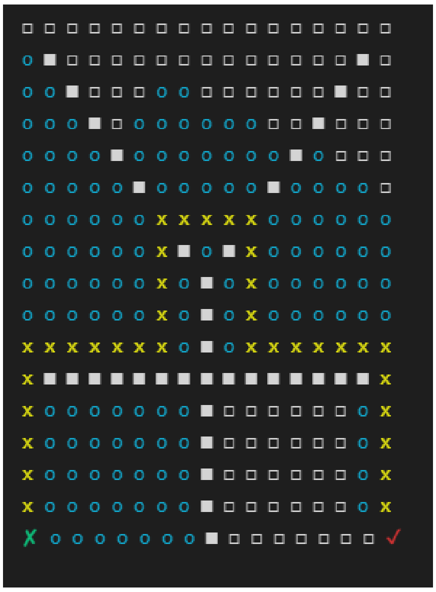

Figure 2. Euclidean for maze5

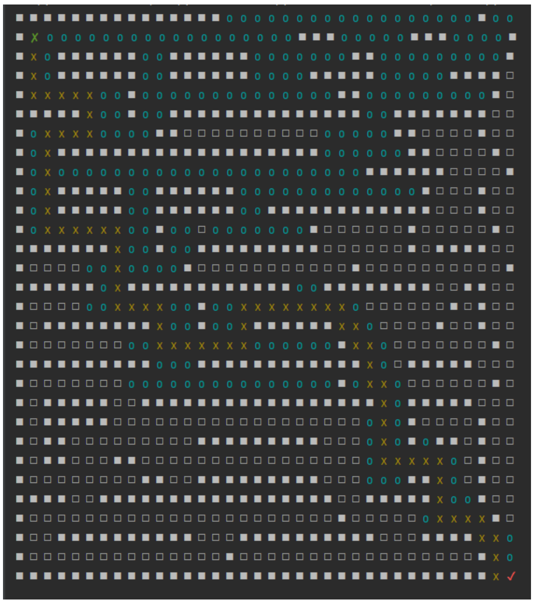

Figure 3 Manhattan for maze6

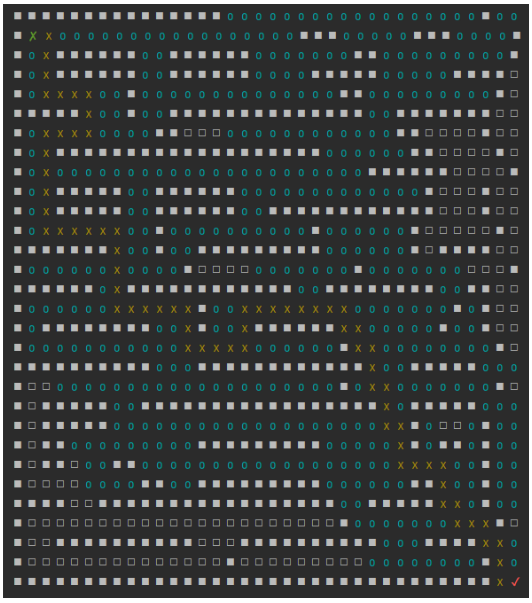

Figure 4 Euclidean for maze6

Reflecting on what went well

- In the modification process, the value of H needs to be greater than or equal to the true distance because it is ignored
  In previous versions, the Manhattan distance was equal to the true distance, which is already the minimum. It can't get any smaller
- The modified Manhattan heuristics have much fewer nodes than the previous extensions

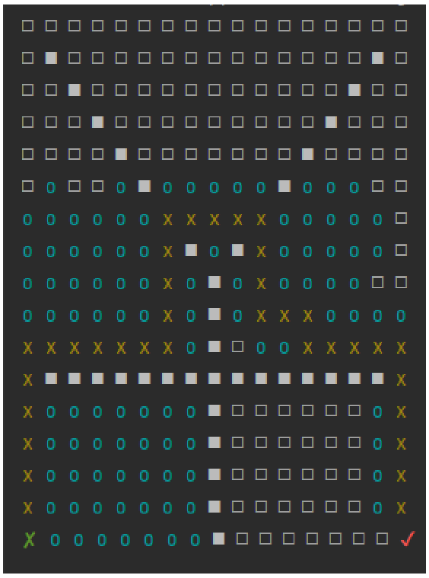

Figure 5Unmodified

Figure 6 modified

what could be improved:

Need to improve the Euclidean distance distance as a heuristic function for the A* function in the case of supporting only straight-line moves

Appendix

```
from time import sleep
from IPython.display import clear_output
import numpy as np
import heapq

WALL = "#"
OPEN = "-"
MANHATTAN = True
EUCLIDEAN = False

DISPLAY_TIME = 0.1 # seconds

HEURISTIC_TYPE = MANHATTAN
# HEURISTIC_TYPE = EUCLIDEAN
# FILENAME = "maze1"
# START = (0, 1)
# GOAL = (2, 4)

# FILENAME = "maze2"
# START = (0, 0)
# GOAL = (3, 3)

# FILENAME = "maze3"
# START = (0, 0)
# GOAL = (4, 7)

# FILENAME = "maze4"
# START = (0, 0)
# GOAL = (2, 0)

FILENAME = "maze5"
START = (16, 0)
GOAL = (16, 16)

# FILENAME = "maze6"
# START = (1, 1)
# GOAL = (29, 35)

class PriorityQueue:
    def __init__(self):
        self.queue = []
        self._index = 0
    def push(self, item, priority):
        # heappush inserts the first element on the queue _queue
        heapq.heappush(self.queue, (priority, self._index, item))
        self._index += 1
```

```python
    def pop(self):
        # heappop removes the first element from the queue _queue
        return heapq.heappop(self.queue)[-1]


class Node:
    def __init__(self, x: int, y: int, g=0, h=0):
        self.x = x
        self.y = y
        self.g = g
        self.h = h
        self.f = g + h
        self.parent = None

    def __lt__(self, other):
        return self.f < other.f

    def coordinates(self):
        return (self.x, self.y)

def read_maze_file(filename: str):
    with open(filename + ".txt", "r") as file:
        maze = np.array([list(line.strip()) for line in file])

    if len(maze) == 0:
        raise Exception("Maze file is empty")

    for row in maze:
        if len(row) != len(maze[0]):
            raise Exception("Maze file is malformed")

    return maze

def position(coordinates: tuple[int, int], maze: np.ndarray):
    x = coordinates[0]
    y = coordinates[1]

    if x < 0 or x >= len(maze) or y < 0 or y >= len(maze[0]) or
maze[x][y] == WALL:
        raise ValueError("Invalid position")
    else:
        return Node(x, y)


def heuristic(a: Node, b: Node, type=MANHATTAN):
    if type == MANHATTAN:
        return abs(a.x - b.x) + abs(a.y - b.y)
    elif type == EUCLIDEAN:
        return ((a.x - b.x) ** 2 + (a.y - b.y) ** 2) ** 0.5
```

```
def print_maze(maze: np.ndarray, checked: set[tuple[int, int]], path:
list[tuple[int, int]], start: Node, end: Node):
    clear_output(wait=False if DISPLAY_TIME > 0 else True)

    START_SYMBOL = "▶"
    END_SYMBOL = "★"
    PATH_SYMBOL = "x"
    CHECKED_SYMBOL = "o"
    OPEN_SPACE_SYMBOL = "□"
    WALL_SYMBOL = "■"

    output = ""
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if (i, j) == start.coordinates():
                output += "\x1b[32m" + START_SYMBOL + "\x1b[0m "
            elif (i, j) == end.coordinates():
                output += "\x1b[31m" + END_SYMBOL + "\x1b[0m "
            elif (i, j) in path:
                output += "\x1b[33m" + PATH_SYMBOL + "\x1b[0m "
            elif (i, j) in checked:
                output += "\x1b[36m" + CHECKED_SYMBOL + "\x1b[0m "
            elif maze[i][j] == OPEN:
                output += OPEN_SPACE_SYMBOL + " "
            elif maze[i][j] == WALL:
                output += WALL_SYMBOL + " "
        output += "\n"

    print(output[:-1])



def a_star(maze: np.ndarray, start: Node, goal: Node, display_time=0.25,
h_type=MANHATTAN):
    checked = set()
    open = PriorityQueue()
    open.push(start,[0,0])
    # open = [start]
    while len(open.queue) > 0:
        #heapq.heapify(open.queue)
        current = open.pop()
        #current = heapq.heappop(open)
        checked.add((current.x, current.y))

        # Print maze at current step
        if display_time > 0:
            print_maze(maze, checked, [], start, goal)
            sleep(display_time)
```

```
        # If goal is reached
        if current.x == goal.x and current.y == goal.y:
            path = []
            while current is not None:
                path.append((current.x, current.y))
                current = current.parent

            # Print final maze with path
            print_maze(maze, checked, path[::-1], start, goal)
            return

        children = []

        # Generate children
        for dx, dy in [(-1, 0), (1, 0),(0, -1), (0, 1)]:
            node_position = (current.x + dx, current.y + dy)

            # If new position is out of bounds or a wall or already in
checked set
            if (
                node_position[0] < 0 or node_position[0] >= len(maze)
or
                 node_position[1] < 0 or node_position[1] >= len(maze[0])
or
                maze[node_position[0]][node_position[1]] == WALL or
                node_position in checked
            ):
                continue

            heuristic_value = heuristic(
                Node(node_position[0], node_position[1]), goal, h_type)
            new_node = Node(
                node_position[0], node_position[1], current.g + 1,
heuristic_value)
            new_node.parent = current
            children.append(new_node)

        # Loop through children
        for child in children:
            if (child.x, child.y) in checked:
                continue
            open.push(child,[child.f,child.h])
            # heapq.heappush(open, child)
            checked.add((child.x, child.y))

maze = read_maze_file(FILENAME)
start = position(START, maze)
```

```
goal = position(GOAL, maze)

a_star(maze, start, goal, DISPLAY_TIME, HEURISTIC_TYPE)
```