Artificial Intelligence for Big Data Systems

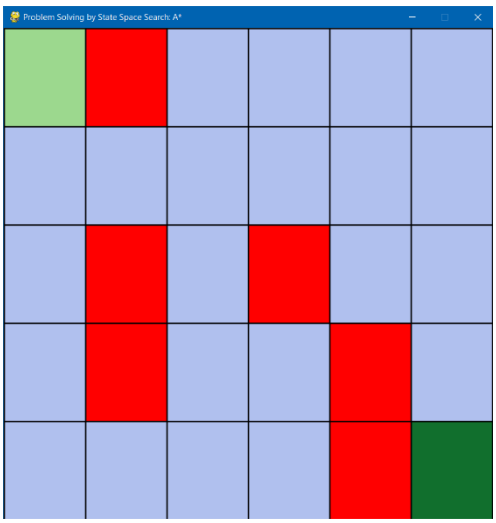
Assignment 1: Problem Solving by State Space Search

(Group Report)

Stefan Velev, 0MI3400521

Introduction

We are given the problem of finding a path between the starting node (0,0) and the end node (4,5) of a maze. The permitted moves are four: Left, Right, Up and Down. The red nodes contain obstacles and no movements are possible through them. The boundaries are walls which are blocking the movements.



My solution to the problem was implemented in ***Python 3.11***. For those purposes, I have used ***PyCharm 2024.3*** as an Integrated Development Environment (IDE). The only external module that I used was ***Pygame***. It is a popular library for building games and multimedia applications which handles graphics, animations, sounds, user input, etc. The rationale behind utilizing it was to create a visual representation of a maze in a 2D plain in graphics mode and employ the **A* algorithm** to find an optimal path from the starting node to the end node. ***Pygame*** serves as the rendering engine that displays the maze grid, nodes based on their type and traces the pathfinding process in real-time.

The solution creates a grid with specific start, end, boundary and empty nodes. The **A* algorithm** then explores possible routes, prioritising those that minimize the estimated cost to the goal. This approach allows the visualisation of the pathfinding steps and the final path efficiently and intuitively.

```
COLOUR_EMPTY_NODE = (176, 192, 238)
COLOUR_BORDERS = (0, 0, 0)
COLOUR_START_NODE = (156, 216, 142)
COLOUR_END_NODE = (17, 111, 45)
COLOUR_OBSTACLE = (255, 0, 0)
COLOUR_VISITED_PATH = (255, 218, 149)
COLOUR_EXPANDED_PATH = (194, 147, 3)
COLOUR_SOLUTION_PATH = (59, 234, 19)
WIDTH = 700

pygame.display.set_caption("Problem Solving by State Space Search: A*")
win = pygame.display.set_mode((WIDTH, WIDTH))
```

Task 1: Choose a suitable representation of the maze nodes using the available Python data structures

Each `Node` represents a cell in the grid and its properties. It defines the position and the size of the cell on the grid. Each `Node` has a colour attribute that changes according to the node's type (obstacle, start, end, visited, expanded, etc.). This structure allows flexibility in assigning properties and methods to each grid cell.

```
class Node:
    def __init__(self, row, col, width, height, total_rows, total_cols):
        self.row = row
        self.col = col

        self.width = width
        self.height = height

        self.total_rows = total_rows
        self.total_cols = total_cols

        self.x = row * width
        self.y = col * height

        self.colour = COLOUR_EMPTY_NODE

        self.neighbours = []
```

The separate `Node` objects contain a list of their neighbours as a property. This is a possible strategy when it comes to the moment of finding the neighbours. We have a method in the `Node` class which is responsible for the correct assignment of the neighbouring nodes:

```
    def update_neighbours(self, grid):
        self.neighbours = []

        if self.col < self.total_cols - 1 and not grid[self.row][self.col + 1].is_obstacle():
            self.neighbours.append(grid[self.row][self.col + 1])

        if self.col > 0 and not grid[self.row][self.col - 1].is_obstacle():
            self.neighbours.append(grid[self.row][self.col - 1])

        if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_obstacle():
            self.neighbours.append(grid[self.row + 1][self.col])

        if self.row > 0 and not grid[self.row - 1][self.col].is_obstacle():
            self.neighbours.append(grid[self.row - 1][self.col])
```

The maze is represented as a 2D grid using nested lists of `Node` objects stored in the `grid` variable. This is an appropriate option because mazes are matrix problems and as such, they are stored in the computer memory as two-dimensional arrays (list of lists).

```
def make_grid(rows, cols, total_width):
    grid = []
    height = total_width // rows
    width = total_width // cols
    for i in range(rows):
        grid.append([])
        for j in range(cols):
            current_node = Node(i, j, height, width, rows, cols)
            grid[i].append(current_node)

    return grid
```

Task 2: Define the success criteria for reaching the end node.

The success criteria for this kind of problem are when the **A* algorithm**'s current node reaches the end node, marking the successful completion of the pathfinding process. In the presented solution of mine that is when the current node (popped from the frontier based on specific criteria) equals the end node. Therefore, the algorithm calls the `reconstruct_path(…)` function to display the path.

```
# Success criteria
if current_node == end_node:
    reconstruct_path(parent_of, end_node, draw, start_time, max_frontier_size)
    return True
```

The function `reconstruct_path(…)` is used to restore the path starting from the end node to the start node. This is done thanks to the `parent_of` dictionary which holds as values the parents of the nodes (stored as keys).

Task 3: Define the test function which accumulates all possibles moves for each node.

In this implementation, the `update_neighbours(self, grid)` method of the `Node` class acts as the test function by accumulating all valid moves (neighbours) for each node that are not obstacles and are not out of the maze board. Such a function is also possible to be outside the `Node` class in a separate data structure. It identifies valid moves by checking if neighbouring nodes are within the grid bounds and are not marked as obstacles.

```
def update_neighbours(self, grid):
    self.neighbours = []

    if self.col < self.total_cols - 1 and not grid[self.row][self.col + 1].is_obstacle():
        self.neighbours.append(grid[self.row][self.col + 1])

    if self.col > 0 and not grid[self.row][self.col - 1].is_obstacle():
        self.neighbours.append(grid[self.row][self.col - 1])

    if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_obstacle():
        self.neighbours.append(grid[self.row + 1][self.col])

    if self.row > 0 and not grid[self.row - 1][self.col].is_obstacle():
        self.neighbours.append(grid[self.row - 1][self.col])
```

The answer whether a node (cell) is free, or it is an obstacle can be understood using the `colour` field of the object. The obstacles are created manually with the help of the `make_obstacle(self)` method. They are fixed on the maze board with the help of the function `create_obstacles(grid)`.

```
    def is_obstacle(self):
        return self.colour == COLOUR_OBSTACLE

    def make_obstacle(self):
        self.colour = COLOUR_OBSTACLE


……

def create_obstacles(grid):
    ……
    ……
    obstacle_6 = grid[4][4]
    obstacle_6.make_obstacle()
```

Task 4: Define the estimation function to calculate the cost for each move.

In the maze problem the cost of each step is 1. The heuristic function estimates the cost of the cheapest path from the given node to the goal (end node). For solving the problem, I have used three heuristic functions – calculating **Manhattan distance**, **Euclidean distance** and **Chebyshev distance**.

```
def manhattan_distance(start, goal):
    x1, y1 = start
    x2, y2 = goal
    return abs(x1 - x2) + abs(y1 - y2)
.....
def euclidean_distance(start, goal):
    x1, y1 = start
    x2, y2 = goal
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
.....
def chebyshev_distance(start, goal):
    x1, y1 = start
    x2, y2 = goal
    return max(abs(x2 - x1), abs(y2 - y1))
```

The **Manhattan distance** is ideal for grid-based mazes where movement is restricted to up, down, left and right. The **Euclidean distance** and the **Chebyshev (diagonal) distance** are useful when diagonal movements are allowed. Nevertheless, all the functions provide a direct estimate of the cost from the current node to the goal.

Task 5: Define the path function which accumulates the past nodes into a path.

The `reconstruct_path(…)` function constructs the path starting from the end node until the start node by backtracking through the `parent_of` dictionary which records each node's predecessor. In that way each node along the path (excluding the start node and the end node) is coloured appropriately as a part of the solution.

```
def reconstruct_path(parent_of, current, draw, start_time, max_frontier_size):
    path_count = 0

    while current in parent_of:
        current = parent_of[current]
        current.make_solution_path()
        path_count += 1
        draw()
    end_time = timer()
    working_time = end_time - start_time
    pygame.display.set_caption(f'Time of A* Algorithm: {format(working_time, ".2f")}s | '
                               f'Solution: {path_count + 1} nodes | '
                               f'Max size of frontier: {max_frontier_size}')
```

Task 6: Define the cost function which accumulates the total cost of the path.

As we know the total cost is calculated using the following formula: $f(n) = g(n) + h(n)$. Two dictionaries in the algorithm are responsible for calculating these aspects – `g_score` and `f_score`. The `g_score` dictionary keeps track of the accumulated cost of the path from the start node to the current node. The `f_score` dictionary estimates the total path cost (accumulated cost so far + heuristic estimate).

```
g_score = {current_neighbour: float("inf") for row in grid for current_neighbour in row}
f_score = {current_neighbour: float("inf") for row in grid for current_neighbour in row}
g_score[start_node] = 0
f_score[start_node] = manhattan_distance(start_node.get_pos_index(), end_node.get_pos_index())
```

Task 7: Create an algorithm for search considering the success criteria.

I have implemented the informed **A* search algorithm**. It utilizes a priority queue to explore nodes based on their `f_score` values and updates the neighbour of each node. This function continues while the frontier is not empty or the end node is not expanded. When that happens the optimal path is reconstructed and displayed.

```
def a_star(draw, grid, start_node, end_node, start_time):
    frontier = PriorityQueue()
    insertion_counter = 0
    frontier.put((0, insertion_counter, start_node))
    max_frontier_size = 1
    parent_of = {}
    g_score = {current_neighbour: float("inf") for row in grid for current_neighbour in row}
    g_score[start_node] = 0

    f_score = {current_neighbour: float("inf") for row in grid for current_neighbour in row}
    f_score[start_node] = manhattan_distance(start_node.get_pos_index(), end_node.get_pos_index())


    frontier_set = {start_node}
    while not frontier.empty():
        current_node = frontier.get()[2]
        frontier_set.remove(current_node)

        if current_node != start_node:
            current_node.expand()

        if current_node == end_node:
            reconstruct_path(parent_of, end_node, draw, start_time, max_frontier_size)
            return True

        for neighbour in current_node.neighbours:
            current_g_score = g_score[current_node] + 1

            if current_g_score < g_score[neighbour]:
                parent_of[neighbour] = current_node
                g_score[neighbour] = current_g_score
                f_score[neighbour] = current_g_score + manhattan_distance(neighbour.get_pos_index(),
                                                                          end_node.get_pos_index())
                if neighbour not in frontier_set:
                    insertion_counter += 1
                    frontier.put((f_score[neighbour], insertion_counter, neighbour))
                    frontier_set.add(neighbour)
                    max_frontier_size = max(max_frontier_size, frontier.qsize())
                    neighbour.visit()

        draw()
        pygame.display.update()
        pygame.time.delay(ANIMATION_TIME_MILLISECONDS)

    return False
```

The goal test is located after popping a node, which ensures that **A* algorithm** stops only when the goal node is dequeued from the priority queue with the lowest `f_score` among remaining nodes, which is essential for optimality. The use of auxiliary structures like `parent_of`, `frontier_set` and the efficient tracking of `g_score` and `f_score` align with **A* algorithm**'s principles, ensure accuracy, optimality, and overall robustness in complex grid environments.

The choice of a heuristic greatly impacts **A* algorithm**'s performance. Here, the Manhattan distance works well for a grid but could be less effective in weighted or diagonal grids. The **A* algorithm** allows flexibility in changing the heuristic, making it adaptable for different grid setups and requirements. For example, Euclidean distance could be used for continuous 2D spaces, enhancing the algorithm's applicability across different scenarios.

Task 8: Run the program and output the optimal path found by the algorithm.

Running the `main(…)` function initiates the program, setting up the grid and calling the **A* algorithm** when a key is pressed. It visually displays the steps in finding the optimal path. The *pale-yellow cells* are the ones which are visited (pushed in the frontier) by the algorithm. The *dark yellow cells* are those which are expanded (popped from the frontier). The optimal path appears in *green colour* on the grid. On the top of the ***Pygame*** window, I have displayed some statistics such as the execution time, the number of nodes in the solution path and the maximum frontier size. For better visualisation, I have made separate plots of each iteration of the algorithm and displayed them as an animation with 600 milliseconds between each so that we can track and assess the results of the program.