Artificial Intelligence for Big Data Systems

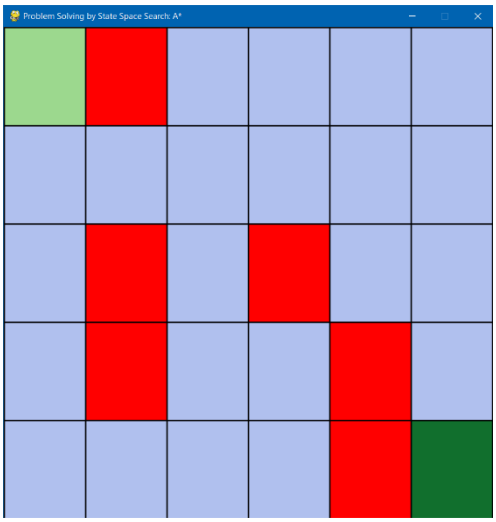
Assignment 1: Problem Solving by State Space Search

(Individual 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: Implement an alternative algorithm for solving the problem using different method or different heuristic function.

We have already seen in the group task how the **A* algorithm** behaved in the situation of the three different heuristics. Now, I will use the same maze problem but implement it with alternative algorithms. I have chosen three other algorithms – **Breadth-First Search (BFS)**, **Depth-First Search (DFS)** and **Greedy Search (GS)**.

Breadth-First Search (BFS) is a classic search algorithm that explores nodes layer by layer making it ideal for finding the shortest path in an unweighted grid. BFS begins with the `start_node` placed in a queue (FIFO). The `visited` set keeps track of the nodes that have been explored in order to avoid revisiting them (prevent cycles). The dictionary `parent_of` maps each node to its parent which will help in reconstructing the path at a later stage. BFS follows a first-in-first-out approach using `queue.popleft()` to explore nodes in layers. For each `current_node` which is dequed, the algorithm does: expansion and neighbour exploration with goal test. The path reconstruction and the animation and rendering are elements which remain the same regardless of the different algorithms.

```
def bfs(draw, start_node, end_node, start_time):
    queue = deque([start_node])
    max_queue_size = 1
    parent_of = {}
    visited = {start_node}

    while queue:
        current_node = queue.popleft()

        if current_node != start_node:
            current_node.expand()

        for neighbour in current_node.neighbours:
            if neighbour == end_node:
                parent_of[neighbour] = current_node
                reconstruct_path(parent_of, end_node, draw, start_time, max_queue_size)
                return True

            if neighbour not in visited and not neighbour.is_obstacle():
                visited.add(neighbour)
                parent_of[neighbour] = current_node
                queue.append(neighbour)
                max_queue_size = max(max_queue_size, len(queue))
                neighbour.visit()

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

    return False
```

Depth-First Search (DFS) is a classic search algorithm that explores as far as possible along each branch before backtracking. It begins by placing the `start_node` on a stack (LIFO). The `visited` set is used again to keep track of nodes that have been explored whereas the `parent_of` dictionary maps each node to its predecessor so that path reconstruction is possible. DFS operates by popping the top node from the stack and marking it as expanded. The neighbour exploration with the goal test is the next step. The path reconstruction and the animation and rendering are elements which remain the same regardless of the different algorithms.

```
def dfs(draw, start_node, end_node, start_time):
    stack = [start_node]
    max_stack_size = 1
    parent_of = {}
    visited = {start_node}

    while stack:
        current_node = stack.pop()

        if current_node != start_node:
            current_node.expand()

        for neighbour in current_node.neighbours:
            if neighbour == end_node:
                parent_of[neighbour] = current_node
                reconstruct_path(parent_of, end_node, draw, start_time, max_stack_size)
                return True

            if neighbour not in visited and not neighbour.is_obstacle():
                visited.add(neighbour)
                parent_of[neighbour] = current_node
                stack.append(neighbour)
                max_stack_size = max(max_stack_size, len(stack))
                neighbour.visit()

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

    return False
```

Greedy Best-First Search prioritises nodes based only on a heuristic (the Manhattan distance) that estimates their proximity to the goal. It chooses the path that seems closest to the target at each step. It is not necessarily optimal in terms of path length. The algorithm pops the node with the smallest heuristic value ($f(n) = h(n)$) from the priority queue and marks it as expanded. The goal test and the neighbour exploration parts follow. The path reconstruction and the animation and rendering are elements which remain the same regardless of the different algorithms.

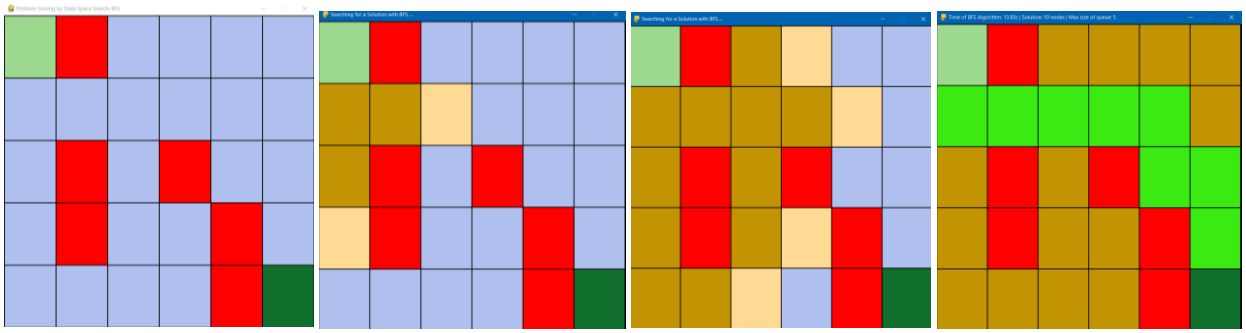
```
def greedy_best_first_search(draw, start_node, end_node, start_time):
    frontier = PriorityQueue()
    insertion_counter = 0
    frontier.put((0, insertion_counter, start_node))
    max_queue_size = 1
    parent_of = {}
    frontier_set = {start_node}
    visited = set()

    while not frontier.empty():
        current_node = frontier.get()[2]
        frontier_set.remove(current_node)
        if current_node == end_node:
            reconstruct_path(parent_of, end_node, draw, start_time, max_queue_size)
            return True
        if current_node != start_node:
            current_node.expand()
        visited.add(current_node)
        for neighbor in current_node.neighbours:
            if neighbor not in frontier_set and neighbor not in visited and not neighbor.is_obstacle():
                parent_of[neighbor] = current_node
                insertion_counter += 1
                heuristic = manhattan_distance(neighbor.get_pos_index(), end_node.get_pos_index())
                frontier.put((heuristic, insertion_counter, neighbor))
                frontier_set.add(neighbor)
                max_queue_size = max(max_queue_size, frontier.qsize())
                neighbor.visit()
        draw()
        pygame.display.update()
        pygame.time.delay(ANIMATION_TIME_MILLISECONDS)
    return False
```

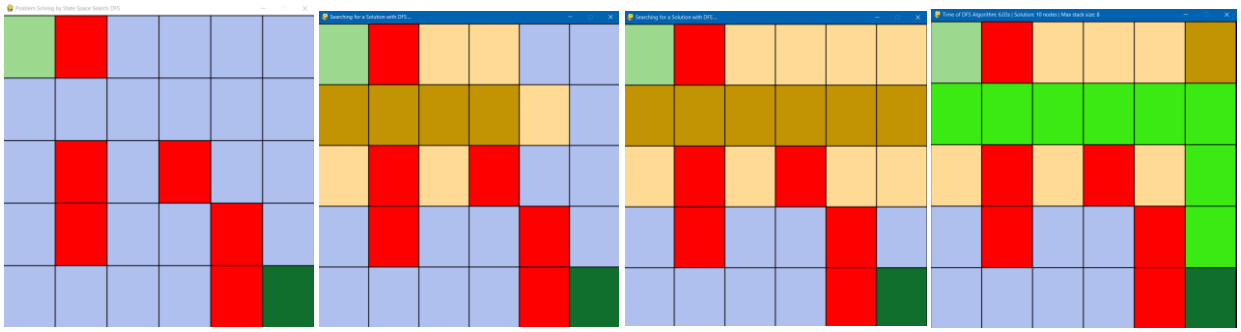
Task 2: Trace the attempted nodes during the search process.

Task 3: Visualise the search paths during the execution in 2D plain in graphics mode.

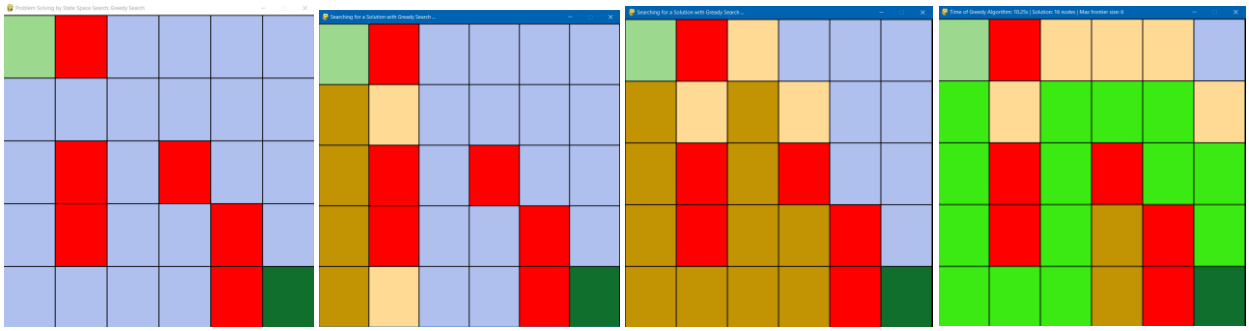
*Screenshots of the GUI of the program using the uninformed **BFS search algorithm**:*



*Screenshots of the GUI of the program using the uninformed **DFS search algorithm**:*



*Screenshots of the GUI of the program using the informed **Greedy Best-First Search algorithm**:*



Task 4: Compare the complexity of the four algorithms in terms of operations and memory.

I will compare the complexity of the four algorithms which I implemented in both parts of the assignment – **A*** (with Manhattan distance as a heuristic), **Breadth-First Search**, **Depth-First Search**, **Greedy Best-First Search** (with Manhattan distance as a heuristic). I will start with the theoretical advantages and disadvantages of each of these.

In general, **BFS** is optimal for unweighted graphs and it guarantees the shortest path if all edges have equal weight. Its main weakness is that it requires more memory (exponential) since it needs to store all the nodes at the current depth level. On the other hand, **DFS** is with low memory usage (linear) but it can be inefficient and non-optimal particularly in large or complex spaces. It does not guarantee to find the optimal path.

Greedy Best-First Search is susceptible to poor heuristics. In addition, it can be so greedy in following only the heuristics that it can fail to find the optimal path. Its advantage is that it is more focused towards the goal. It is best for problems where a good heuristic is available but optimality is not guaranteed at all. **A* Search** combines the best features of BFS and Greedy Search by considering both path cost and goal proximity. As a result, it is optimal and efficient. Its drawback is that it requires much more memory (exponential) and a well-designed heuristic for best performance. It is the most powerful algorithm out of the four.

To compare the complexity of the four algorithms in terms of **operations** and **memory**, I have chosen to measure three metrics of their programs:

- **Execution time** (in seconds) – for visualisation purposes I have left the 600 ms delay of each algorithm iteration in all programs (in that way we can still see the winner in terms of time)
- **Number of nodes in the solution path** – to compare the optimality
- **Max frontier/open set size** – used for measuring the memory complexity of the problem; the maximum number of elements for the specific data structure during the execution

The summarised results can be found in the following table:

	BFS	DFS	Greedy Search	A*
Execution time (with the delay)	13.25 s	5.43 s	10.25 s	11.44 s
№ of nodes in solution path	10	10	16	10
Max frontier/open set size	5	8	6	7

What can we notice from the results? Some contradictions can be seen. For instance, the theory says that **DFS** is with linear memory complexity whereas **BFS** is exponential in terms of memory. Then why **BFS** uses less memory in that problem? The reason is the complexity of the problem – the maze grid is only 5 x 6 nodes. It is too small and we cannot see clearly the difference. When we look at the execution time, **DFS** seems to be the fastest algorithm for that problem. That is not a trend – the reason is the distribution of the obstacles. As we can see in the animation the **DFS** algorithms is lucky to start with the right path and there is only one mistake that it makes. When it comes to the solution path only **Greedy Search** cannot find the optimal one. It makes a mistake on the first turn. The heuristic there is the same for the both directions (8 nodes till the end) of the path. However, the algorithm is not lucky and chooses the wrong one. After that, the algorithm thinks it is on the fastest track until it finds the obstacle at (4, 4).

I have provided other Python programs which make the state space much more complex (20 x 20 nodes). I have put some random obstacles and tried the above algorithms again. In that situation, the power of the A algorithm can be clearly seen. Only the Greedy Search beats it. Moreover, in that case it finds the optimal solution path. However, this is not guaranteed and in that case it is due to the location of the obstacles.*