Laboratory 1

Variant 3, Group 13

By Saumya Shah and Anas Tagui

Introduction

In this laboratory, we aim to solve a 2D maze navigation problem using the A* search algorithm. The maze is represented as a grid of integers, where 1 represents walls and 0 represents walkable paths.
Our goal is to find the shortest path from the start to the end while visualising each step of the search on the console.
The A* algorithm is an efficient pathfinding technique that balances exploration and cost estimation using the formula:

$$f(n) = g(n) + h(n)$$

where g(n) is the cost to reach a node and h(n) is a heuristic estimating the distance to the goal.
We will use the Manhattan Distance heuristic since movement is limited to four directions.
This project will help us understand A*, handle obstacles efficiently, and optimise pathfinding performance in constrained environments.

Advantages of A*

1. **Optimal and complete** – Always finds the shortest path if an admissible heuristic is used.
2. **Efficient compared to brute-force methods** – Uses heuristics to guide the search, reducing unnecessary computations.
3. **Flexible** – Can be adapted with different heuristics for various applications (e.g., robotics, game AI, navigation).

Disadvantages of A*

1. **Heuristic sensitivity** – The efficiency heavily depends on choosing an appropriate heuristic. A poor heuristic can make A* perform like brute-force search.
2. **Memory usage** – A* stores all successor and explored nodes, which can be problematic for very large mazes.
3. **Computation time** – While efficient, it can be slower than simpler algorithms in some cases, especially in uniform-cost environments.

Implementation

Heuristic Functions

```python
# Heuristic 1 - Manhattan Distance
def manhattan_distance(current: Position, finish: Position) -> int:
    return abs(current[0] - finish[0]) + abs(current[1] - finish[1])

# Heuristic 2 - Euclidean Distance
def euclidean_distance(current: Position, finish: Position) -> float:
    return ((current[0] - finish[0]) ** 2 + (current[1] - finish[1]) ** 2) ** 0.5
```

Two heuristic functions are considered: the *Manhattan Distance* between the current position and the end point, and the *Euclidean Distance* between the current position and the end point.

Utilities

Certain utility functions and variables are defined:

```python
# Define valid position function
is_valid_pos = lambda pos: 0 <= pos[0] < len(maze) and 0 <= pos[1] < len(maze[0]) and maze[pos[0]][pos[1]] == 0
```

```python
f_cost = lambda pos, heuristic: pos[1] + heuristic(pos[0], finish)

# Define directions
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
```

`is_valid_pos` is a function to check if a given position is valid (ie. it is within the maze and it is not a wall).

`f_cost` is a function to calculate the *f* cost (heuristic score + path cost)

`directions` defines the set of directions (up, down, left, right) that the player can move to. It is used to generate successor positions.

A* Implementation

```python
if not is_valid_pos(start) or not is_valid_pos(finish):
    return -1, []
```

First, the function checks whether the start and finish positions are valid. If not, a path is impossible, so the function returns immediately with `-1`.

Initialisation:

```python
num_steps: int = -1
path: List[Position] = []
```

```
# Initialise frontier and explored set
frontier: PriorityQueue = PriorityQueue()
explored: Set[Position] = set()

# Initialise visualisation variables
parent: Dict[Position, Position] = {}
visualisation: List[MazeViz] = []
```

Next, the function initialises the variables used for the search and visualisation. A priority queue is used to keep track of the positions in the frontier indexed by the `f_cost`. A set is used to keep track of explored sets. A dictionary is used to keep track of the *parent* of each position so that the final path can be reconstructed for visualisation. A list of `MazeViz` (`List[List[String]]`) is used to store the String representation of each step of the search.

```
# Add start position to frontier
frontier.put((0, (start, 0)))
```

The starting state is then put into the frontier, and the search loop is begun.

<u>Loop:</u>

```
# Loop until frontier is empty
while frontier.qsize() > 0:
```

The loop is run until the frontier is empty.

At each step of the loop, the following happens:

```
_, (curr_pos, curr_cost) = frontier.get()

# Check if current position is the finish
if curr_pos == finish:
    num_steps = curr_cost

    # Reconstruct the path
    path = []
    current = curr_pos
    while current != start:
        path.append(current)
        current = parent[current]
    path.append(start)
    path.reverse()  # Path is from start to finish

    break
```

1. An element is popped from the frontier and set as the current position.
2. If the current position is the end point, then the search is terminated, the number of steps is set to the `g` (path cost) of the finish position, the path is reconstructed using the `parent` dictionary, and the number of steps and visualisation variables are

returned.

```
# Add current position to explored set
explored.add(curr_pos)
```

3. The current position is added to the `explored` set.

```
# Generate successors
for d_x, d_y in directions:
    new_pos = (curr_pos[0] + d_x, curr_pos[1] + d_y)
    new_cost = curr_cost + 1

    # If the new position is valid and not explored, add it to the frontier
    if is_valid_pos(new_pos) and new_pos not in explored:
        frontier.put((f_cost((new_pos, new_cost), heuristic_func), (new_pos, new_cost)))
        parent[new_pos] = curr_pos  # Record the parent of the new position

    # If the new position is in the frontier, update the cost if it is lower
    elif is_valid_pos(new_pos) and new_pos in [pos for _, (pos, _) in frontier.queue]:
        for i, (_, (pos, cost)) in enumerate(frontier.queue):
            if pos == new_pos and new_cost < cost:
                frontier.queue[i] = (f_cost((new_pos, new_cost), heuristic_func), (new_pos, new_cost))
                parent[new_pos] = curr_pos  # Update parent if cost is improved
                break
```

4. Successors for the current position are generated using the four directions.
    a. If the successor is a valid position but not yet explored, then its `f_cost` is calculated and it is added to the frontier.
    b. If the successor is a valid position but already in the frontier, then its cost is compared to the existing entry for that position in the frontier. If the new cost is lower, then it is updated in the frontier.
    c. In both of the above cases, the `parent` for the successor position is appropriately updated.

```
# Visualisation
viz: MazeViz = [[' ' if cell == 0 else '#' for cell in row] for row in maze]
for pos in explored:
    viz[pos[0]][pos[1]] = '.'
for pos in [pos for _, pos in frontier.queue]:
    pos = pos[0]
    viz[pos[0]][pos[1]] = 'i'
viz[start[0]][start[1]] = 'S'
viz[finish[0]][finish[1]] = 'F'
visualisation.append(viz)
```

5. The visualisation is generated by taking the original maze and mapping the positions to appropriate character representations.

```python
# Final visualisation of path (if path found)
if num_steps != -1:
    viz: MazeViz = [[' ' if cell == 0 else '#' for cell in row] for row in maze]
    for pos in explored:
        viz[pos[0]][pos[1]] = '.'
    for i, pos in enumerate(path):
        viz[pos[0]][pos[1]] = str(i)
    viz[start[0]][start[1]] = 'S'
    viz[finish[0]][finish[1]] = 'F'
    visualisation.append(viz)
```

At the end of the function, a final visualisation is generated indicating the final path found. Then, the `num_steps` and `visualisation` variables are returned.

Visualisation

The key for each symbol is as follows:

```
. - explored
i - frontier
S - start
F - finish
# - wall
Number - step number
```

The following is an example of a visualisation of A* search from (0, 0) to (2, 2).

```
S #
i #   #
    F
# # #
      #
```

(1, 0) is added to the frontier first, as indicated by the `i` (indicating 'in the frontier')

```
S #
. #   #
i   F
# # #
      #
```

(2, 0) is added to the frontier next, and (1, 0) is marked with a `.` (indicating that it has been explored).

```
S #
. #   #
. i F
# # #
      #
```

```
S #
. #   #
. . F
# # #
      #
```

After exploration, the finish state is added to the frontier. When it is popped from the frontier and visited, the solution is found.

```
Final path:
S #
1 #   #
2 3 F
# # #
      #
```

The final path is printed to the console.

Discussion

Impact of Different Heuristics

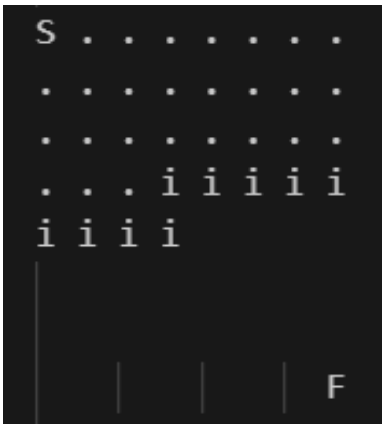
The code implements A* search with two possible heuristic functions:

Manhattan Distance: Sum of absolute differences in x and y coordinates
Euclidean Distance: Straight-line distance between points

When using these different heuristics in the A* algorithm, we can expect several differences:

Exploration Pattern: Manhattan distance will prioritise grid-aligned movements, which results in exploration in a diamond-shaped pattern. Euclidean distance tends to explore in a more circular pattern from the start point.

For example, these are two visualisations from an open 8 x 8 maze with the start and end point at different corners. The first corresponds to the Euclidean distance heuristic, and the second corresponds to the Manhattan distance heuristic.



We notice that the first frontier looks more ‘circular’, while the second looks more ‘square-like’.

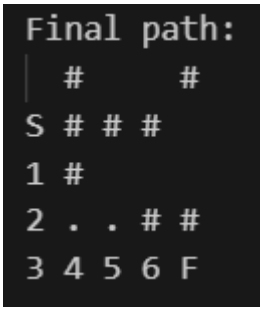
Performance with Movement Constraints: Since the algorithm allows only 4-directional movement (up, down, left, right), Manhattan distance is theoretically more appropriate as it exactly represents the minimum possible path length. On the other hand, Euclidean distance will always underestimate the true path cost in grid-based movement without diagonals, which maintains admissibility but can be less efficient.

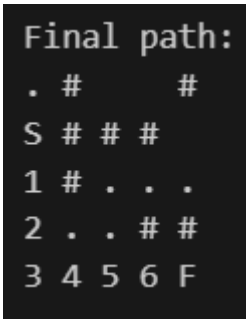
Path Selection: Manhattan may perform better in maze-like environments with perpendicular walls. Euclidean may perform better when the optimal path requires moving diagonally.

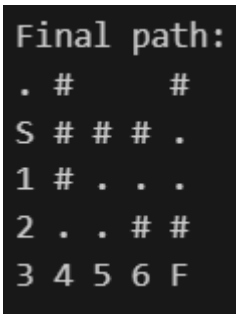
Comparison between A* & other search algorithms

Node Expansion Efficiency:

Using either Manhattan or Euclidean distance heuristic, A* consistently explored less than or equal extra nodes compared to other informed and uninformed search algorithms, such as breadth-first search or greedy best-first search.







For example, in the above maze, the first image corresponds to the final state with A* search using Manhattan distance, the second corresponds to greedy best-first search, and the last corresponds to breadth-first search.

Therefore, with an appropriate admissible heuristic, A* always has performance (in terms of nodes expanded) better than or equal to other search algorithms. This is because A* uses a heuristic to focus on promising paths.

Space Efficiency:

A* has high memory usage compared to some search algorithms, such as depth-first search. This is because A* can have exponential space complexity in the worst case ($O(b^d)$) while DFS in the worst case has polynomial space complexity (O(d)).

Optimality & Completeness

A* search is both complete and optimal (when using an admissible heuristic), meaning it never overestimates the true cost to the goal. It guarantees finding a solution if one exists by expanding nodes in order of increasing estimated `f cost`. The algorithm ensures optimality because it expands the least-cost node first, meaning the first time it reaches the goal, it has found the shortest path. If the heuristic is also consistent (monotonic), A* never needs to re-expand nodes, further improving efficiency. Other methods, such as DFS or GBFS may be incomplete or may not return the optimal path.

Conclusion

A* is optimal (with an admissible heuristic), complete, and generally more efficient than other search algorithms, like BFS, DFS, or GBFS. It can also be tailored to specific problems by designing appropriate heuristics.

However, it has a high memory cost, since it stores multiple paths in the priority queue. It is also sensitive to the heuristic, and a bad heuristic may lead to worse performance. It can also be slow if the heuristic is weak or the search space is large.

Conclusion

Learning Outcomes

This laboratory on the A* search algorithm provided insights into pathfinding techniques, heuristic design, and algorithmic performance in maze navigation scenarios. Specifically, we learnt about the following:

Heuristic Sensitivity: The choice of heuristic function significantly impacts search performance. Our comparative analysis of Manhattan and Euclidean distance heuristics showed that different distance metrics produce distinct exploration patterns which can be leveraged for efficiency in different contexts.

Algorithmic Efficiency: A* consistently demonstrated superior node expansion efficiency compared to alternative search algorithms like breadth-first search and greedy best-first search. This efficiency stems from its intelligent use of cost estimation and exploration guidance.

Challenges

Efficient Implementation: We found it difficult to implement certain parts of the A* algorithm efficiently without increasing the complexity of the code, which would make it more difficult to understand and explain.

Test Case Design: Finding test cases that demonstrated the difference between different heuristics and search algorithms was sometimes difficult due to the often subtle differences between the heuristics.

Improvements

- We can try out more sophisticated heuristic functions that can adapt to specific maze characteristics, such as using Chebyshev distance.
- We can design more efficient implementations, such as using a dictionary to track distances between nodes instead of iterating through the frontier.
- We can develop more robust visualisation methods that make it easier to track the search progression.