Chandigarh University
Master of Engineeing—Artificial Intelligence

Experiment 2

Student name: *Aditya Kumar (24MAI14003)*

Subject: *Artificial Intelligence Lab (24CSH-621)*
– Instructor: *Dr. Manjit Singh*
Date: *08 Aug, 2024*

Aim: Implement Informed (heuristic) search strategy for optimal shortest path and traffic navigational system.

1 Theory

The A* (A-Star) algorithm is a popular and highly efficient pathfinding and graph traversal algorithm often used in computer science, particularly for solving problems related to navigating through graphs or grids. It finds the shortest path between two nodes by considering both the actual cost from the start node to the current node (known as ”g” cost) and an estimated total cost to reach the destination from the current node (known as ”h” cost).

2 Algorithm

1. Initialization: Create two lists - open list and closed list. The open list contains nodes that need exploring, while the closed list contains nodes that have already been visited or evaluated. Add the start node to the open list with a cost value equal to its ”g” (actual) cost from the starting position.

2. Loop until the open list is empty:

   - Find the node in the open list with the lowest f-cost, where ‘f(n) = g(n) + h(n)‘, and break the loop if it’s a destination node (i.e., its position matches the goal).
   - Move the selected node from the open list to the closed list.

3. Generate neighboring nodes: For each neighbor of the current node, calculate their ”g” cost by adding up the ”g” value of the current node and the edge weight (distance) between them. Also, estimate the ”h” cost using a heuristic function such as Euclidean distance or Manhattan distance.

4. Check each neighboring node:

   - If the neighbor is not in the open list yet, add it there with its calculated values of g-cost and h-cost (f-cost), along with a reference to the current node (”parent”).

- If the neighbor already exists in the open list but has a higher g-cost than currently found one, update its ”g” cost value and set the current node as its new parent. This ensures that we’re always on the best path towards our goal.

5. Repeat until you reach the destination: Keep repeating steps 2 to 4 until the open list is empty or when a final destination has been reached. The algorithm will then return the shortest-path sequence as a stack (or list) of nodes, where removing an element from the end of this structure results in returning the optimal path.

3 Pseudocode

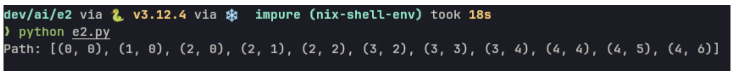
```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path
function A_Star(start, goal, h)
    openSet := {start}
    cameFrom := an empty map
    gScore := map with default value of Infinity
    gScore[start] := 0
    fScore := map with default value of Infinity
    fScore[start] := h(start)
    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)
        openSet.Remove(current)
        for each neighbor of current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)
    return failure
```

4 Code

```python
import heapq


class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0
        self.h = 0
        self.f = 0

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


def heuristic(node_position, goal_position):
    # Manhattan distance heuristic
    return abs(node_position[0] - goal_position[0]) + abs(
        node_position[1] - goal_position[1]
    )


def astar(grid, start, goal):
    # Create start and goal node
    start_node = Node(start)
    goal_node = Node(goal)

    # Initialize both open and closed sets
    open_set = []
    closed_set = set()

    # Add the start node to the open set
    heapq.heappush(open_set, start_node)

    # Main loop
    while open_set:
        # Select the node with the lowest f-score
        current_node = heapq.heappop(open_set)
        closed_set.add(current_node.position)

        # Goal node is reached
        if current_node.position == goal_node.position:
            return reconstruct_path(current_node)

        # Evaluate neighbors
        neighbors = get_neighbors(grid, current_node)
        for neighbor_position in neighbors:
            # Create a neighbor node
            neighbor_node = Node(neighbor_position, current_node)

            # Ignore the neighbor which is already evaluated
            if neighbor_node.position in closed_set:
                continue

            # Calculate the g, h, and f values
            neighbor_node.g = current_node.g + 1
            neighbor_node.h = heuristic(neighbor_node.position, goal_node.
                position)
            neighbor_node.f = neighbor_node.g + neighbor_node.h

            # If a neighbor is in the open set and has a higher f-score,
                skip it
            if not add_to_open(open_set, neighbor_node):
                continue

```

```python
            # Otherwise, add the neighbor to the open set
            heapq.heappush(open_set, neighbor_node)

    # Return None if no path is found
    return None


def get_neighbors(grid, node):
    (x, y) = node.position
    neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]

    # Filter out invalid neighbors
    valid_neighbors = []
    for nx, ny in neighbors:
        if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] == 0:
            valid_neighbors.append((nx, ny))
    return valid_neighbors


def add_to_open(open_set, neighbor):
    for node in open_set:
        if neighbor.position == node.position and neighbor.g >= node.g:
            return False
    return True


def reconstruct_path(current_node):
    path = []
    while current_node:
        path.append(current_node.position)
        current_node = current_node.parent
    return path[::-1]  # Return reversed path


# Example usage
grid = [
    [0, 1, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0],
]

start = (0, 0)
goal = (4, 6)

path = astar(grid, start, goal)
print("Path:", path)
```

5 Output



```
dev/ai/e2 via 🐍 v3.12.4 via ❄  impure (nix-shell-env) took 18s
❯ python e2.py
Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3), (3, 4), (4, 4), (4, 5), (4, 6)]
```

6 Learning Outcomes

1. Understanding of graph theory
2. Implementation of informed search strategy
3. Search strategy evaluation