Assignment No 01

Group Members

MUHAMMAD ZEESHAN	210711
UMER ZAKI	211909
MUHAMMAD ABDULLAH	210779

Subject:

ARTIFICIAL INTELLIGENCE

Subject Instructor;

DR. ASHFAQ

Class:

BEEE-8

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

AIR UNIVERSITY, ISLAMABAD

1 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) is an uninformed search algorithm that expands the node with the lowest path cost. It is a special case of Dijkstra's algorithm, where the goal is to find the shortest path from the start node to the goal node, considering the costs of traversing the edges. UCS operates by always exploring the least costly path, prioritizing nodes based on the cumulative cost from the start node.

The algorithm maintains a priority queue to store nodes sorted by their path cost, and it expands the node with the lowest path cost at each step. When a node is expanded, the algorithm explores all its neighbors and updates their costs, ensuring that it only visits the most promising nodes first.

UCS is optimal when the edge costs are non-negative, ensuring that the first time a node is expanded, it has reached its lowest possible cost.

Algorithm 1 Uniform Cost Search (UCS)

Input: Start node *start*, Goal node *goal*, Graph *graph*
Initialize PRIORITY_QUEUE as empty
Push $(0, start)$ into PRIORITY_QUEUE ▷ (cost, node)
Initialize VISITED_SET as empty
Initialize PATH_COST as empty dictionary
while PRIORITY_QUEUE is not empty **do**
    Pop $(cost, current\_node)$ from PRIORITY_QUEUE
    if *current_node* is in VISITED_SET **then**
        continue ▷ Skip already visited nodes
    end if
    Add *current_node* to VISITED_SET
    $PATH\_COST[current\_node] \leftarrow cost$
    if $current\_node = goal$ **then**
        return $PATH\_COST, VISITED\_SET$
    end if
    for each *neighbor* in $graph[current\_node]$ **do**
        if *neighbor* is not in VISITED_SET **then**
            $NEW\_COST \leftarrow cost + EDGE\_COST(current\_node, neighbor)$
            Push $(NEW\_COST, neighbor)$ into PRIORITY_QUEUE sorted by cost
        end if
    end for
end while
return "No Path Found"

2 A* Search Algorithm

A* (A-star) is a search algorithm that combines the advantages of both Uniform Cost Search and Greedy Best-First Search. It uses a heuristic function in addition to the path cost to guide the search. The idea is to use both the actual cost from the start node and the estimated cost to the goal (from the heuristic) to prioritize nodes.

The A* algorithm works by selecting the node with the lowest $f(n) = g(n) + h(n)$, where:

- $g(n)$ is the actual cost from the start node to the current node and in our it is one for every node.
- $h(n)$ is the heuristic estimate of the cost from the current node to the goal and in our case it is found by Manhattan distance.

Algorithm 2 A* Search

```
Input: Start node start, Goal node goal, Graph graph, Heuristic function h
Initialize PRIORITY_QUEUE as empty
Push (h(start), start, 0) into PRIORITY_QUEUE            ▷ (h(n), node, g(n))
Initialize VISITED_SET as empty
Initialize PATH_COST as empty dictionary
while PRIORITY_QUEUE is not empty do
    Pop (h(n), current_node, g(n)) from PRIORITY_QUEUE
    if current_node is in VISITED_SET then
        continue                                          ▷ Skip already visited nodes
    end if
    Add current_node to VISITED_SET
    PATH_COST[current_node] ← g(n)
    if current_node = goal then
        return PATH_COST, VISITED_SET
    end if
    for each neighbor in graph[current_node] do
        if neighbor is not in VISITED_SET then
            NEW_G ← g(n) + EDGE_COST(current_node, neighbor)
            H_NEIGHBOR ← h(neighbor, goal)
            F_NEIGHBOR ← NEW_G + H_NEIGHBOR               ▷ A* cost function
            Push (F_NEIGHBOR, neighbor, NEW_G) into PRIORITY_QUEUE sorted by (F, node)
        end if
    end for
end while
return "No Path Found"
```

2.1 Comparison of UCS and A* Search Algorithm

In our specific case:

- **Uniform Cost Search (UCS)** focuses solely on path cost and does not take into account any heuristic. This makes it a good choice for scenarios where the goal is to find the optimal path based purely on the cost of traversal. UCS guarantees the shortest path in graphs with non-negative edge costs but may explore many unnecessary nodes.
- **A* Search**, on the other hand, utilizes a heuristic to guide the search toward the goal, which can significantly reduce the number of nodes explored. However, the effectiveness of A* depends on the quality of the heuristic function. When the heuristic is perfect, A* is highly efficient, but if the heuristic is poor, A* may end up exploring more nodes than UCS.
- This difference makes UCS more efficient in cases where unnecessary node exploration is costly, while A* is more efficient when there is a good heuristic available to guide the search.

3 Identifying and Utilizing Bottlenecks

C-3PO must escape a cave where a **bottleneck exists at nodes 27 and 35**, forming the **only passage** between two halves of the maze. Instead of a **single A* search**, we will **split the search into two segments** to efficiently navigate this constraint.

3.1 Understanding Bottlenecks in Search Algorithms

3.1.1 What is a Bottleneck?

- A **bottleneck** in search algorithms refers to a **critical point in the search space** that significantly limits movement from one region to another.
- In this problem, **nodes 27 and 35 form the only passage** connecting two parts of the maze.
- If this passage is blocked, or if the search algorithm does not account for it properly, finding an efficient path could become difficult.

3.1.2 Why is Identifying a Bottleneck Important?

- If an algorithm blindly searches the entire space, it may **waste time exploring irrelevant paths**.
- Identifying bottlenecks helps to **prioritize paths** that lead through crucial areas, improving efficiency.

- It ensures **C-3PO focuses on reaching node 27 first, then continues from node 35** to the goal.

3.1.3 How Does Recognizing the Bottleneck Impact Search Strategy?

- Instead of running a **single A* search from start to goal**, we **split the search into two segments**:
  1. **First segment:** From **start (0) to bottleneck (27)**.
  2. **Second segment:** From **bottleneck (35) to goal (61)**.
- This **reduces the search space** and ensures C-3PO reaches the critical path quickly.

3.2 Utilizing a Split Search Strategy

3.2.1 How A* Search Operates in Two Segments

A* is applied **twice**:

1. **From Start (0) to Bottleneck (27)**
   - Uses **Manhattan distance** heuristic from **each node to 27**.
   - Searches the left half of the maze efficiently.
2. **From Bottleneck (35) to Goal (61)**
   - Uses **Manhattan distance** heuristic from **each node to 61**.
   - Focuses only on the right half of the maze, reducing unnecessary exploration.

3.2.2 Considerations When Selecting Heuristics

- **Segment 1 Heuristic** ($h(n)$) $\rightarrow$ Distance from n to 27.
- **Segment 2 Heuristic** ($h(n)$) $\rightarrow$ Distance from n to 61.
- **Accuracy**: The heuristic should be **consistent and admissible**, ensuring optimal search paths.

3.3 Calculating Total Search Time

3.3.1 Step 1: Run A* from (0 $\rightarrow$ 27)

Using the A* function, count nodes visited:

```
visited1, queue_states1, cost1 = a_star_search(0, 27, graph, positions)
```

Total time from 0 to 27 is **14**.

3.3.2 Step 2: Run A* from ($35 \rightarrow 61$)

Restart A* from the second segment:

```
visited2, queue_states2, cost2 = a_star_search(35, 61, graph, positions)
```

Total time from 35 to 61 is **25**.

3.3.3 Step 3: Compute Total Time

Since each node visit takes 1 unit of time, total time is:

```
total_time = cost1 + cost2
```

Total time:

$$14 + 25 = 39\,\text{min}$$

3.3.4 Factors Affecting Search Time

- **Heuristic Accuracy**: If the heuristic underestimates too much, search may expand unnecessary nodes.
- **Maze Complexity**: Obstacles or dead-ends could force the algorithm to explore more nodes.
- **Queue Processing**: Sorting and updating the queue efficiently ensures faster expansion.