

Experiment No. 2

Aim

To implement any one informed search technique using the A* algorithm.

Theory

Informed search is a search technique used in Artificial Intelligence to find a solution efficiently. It uses additional information, called a heuristic, to guide the search toward the goal.

The A* algorithm is a popular informed search algorithm used to find the shortest path between a starting point and a goal in a weighted graph. It considers both the actual cost and the estimated cost to select the best path.

The A* algorithm uses the following formula:

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

Where:

- $g(n)$: Actual cost from the starting node to the current node.
- $h(n)$: Estimated cost from the current node to the goal node.
- $f(n)$: Total estimated cost of the path.

Working of A* Algorithm

1. Start from the initial node.
2. Calculate the $f(n)$ value for the available nodes.
3. Select the node with the lowest $f(n)$ value.
4. Check whether the selected node is the goal node.
5. If the goal is not reached, explore its neighbouring nodes.
6. Repeat the process until the goal is found or no path is available.

Code & Output:

```
import heapq

def astar(start, goal):
    graph = {
        'A': {'B':1, 'C':3},
        'B': {'D':2, 'E':4},
        'C': {'F':2},
        'D': {'G':3},
        'E': {'G':1},
        'F': {'G':2},
        'G': {}
    }
```

```

q = [(0, start)]
parent = {start: None}
cost = {start: 0}

while q:
    _, node = heapq.heappop(q)

    if node == goal:
        path = []
        while node:
            path.append(node)
            node = parent[node]
        return path[::-1], cost[goal]

    for nxt, weight in graph[node].items():
        new_cost = cost[node] + weight

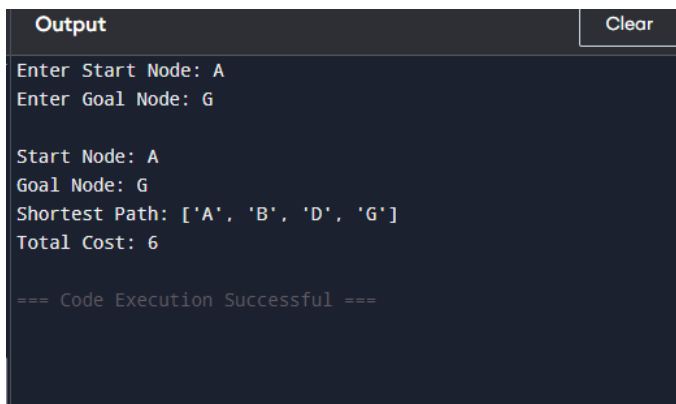
        if nxt not in cost or new_cost < cost[nxt]:
            cost[nxt] = new_cost
            parent[nxt] = node
            heapq.heappush(q, (new_cost, nxt))

start = input("Enter Start Node: ")
goal = input("Enter Goal Node: ")

path, cost = astar(start, goal)

print("\nStart Node:", start)
print("Goal Node:", goal)
print("Shortest Path:", path)
print("Total Cost:", cost)

```



The screenshot shows a dark-themed output window with a 'Clear' button in the top right corner. The text inside the window displays the input and output of the program. The input consists of 'Enter Start Node: A' and 'Enter Goal Node: G'. The output shows 'Start Node: A', 'Goal Node: G', 'Shortest Path: ['A', 'B', 'D', 'G']', and 'Total Cost: 6'. At the bottom, it states '=== Code Execution Successful ==='.

```

Output
Clear

Enter Start Node: A
Enter Goal Node: G

Start Node: A
Goal Node: G
Shortest Path: ['A', 'B', 'D', 'G']
Total Cost: 6

=== Code Execution Successful ===

```

Conclusion

Hence, the informed search technique using the A* algorithm was successfully implemented.

Experiment No. 3

Aim

To implement uninformed search using the Water Jug Problem.

Theory

Uninformed search, also called blind search, is a search technique used in Artificial Intelligence to find a solution without using heuristic information. It uses the initial state, available actions, and goal state to solve a problem.

The Water Jug Problem is a common problem in Artificial Intelligence in which two jugs of different capacities are used to measure an exact amount of water.

For example, we have two jugs: one with a capacity of 4 litres and another with a capacity of 3 litres. The goal is to measure exactly 2 litres of water.

The following operations can be performed:

- Fill: Fill a jug completely with water.
- Empty: Empty all the water from a jug.
- Pour: Transfer water from one jug to another until the first jug is empty or the second jug is full.

Working of the Water Jug Problem

7. Start with both jugs empty.
8. Fill the 4-litre jug completely.
9. Pour water into the 3-litre jug until it becomes full.
10. Empty the 3-litre jug.
11. Transfer the remaining water into the 3-litre jug.
12. Fill the 4-litre jug again.
13. Pour water into the 3-litre jug until it becomes full.
14. The 4-litre jug now contains exactly 2 litres of water.

Code & Output:

```
from collections import deque
```

```
def water_jug(m, n, target):  
    visited = set()  
    queue = deque([(0, 0)])
```

```
    while queue:  
        a, b = queue.popleft()
```

```

if (a, b) in visited:
    continue
visited.add((a, b))

if a == target or b == target or a + b == target:
    return True

next_states = [
    (m, b), # Fill jug1
    (a, n), # Fill jug2
    (0, b), # Empty jug1
    (a, 0), # Empty jug2
    (a - min(a, n - b), b + min(a, n - b)), # Pour jug1 -> jug2
    (a + min(b, m - a), b - min(b, m - a)) # Pour jug2 -> jug1
]

for state in next_states:
    if state not in visited:
        queue.append(state)

return False

```

```

m = int(input("Enter capacity of Jug 1: "))
n = int(input("Enter capacity of Jug 2: "))
target = int(input("Enter target amount: "))

```

```

if water_jug(m, n, target):
    print("Possible")
else:
    print("Not Possible")

```

```

Output
Enter capacity of Jug 1: 2
Enter capacity of Jug 2: 3
Enter target amount: 2
Possible

=== Code Execution Successful ===

```

Conclusion

Hence, the Water Jug Problem was successfully implemented using uninformed search.