**AI LAB:**

**WEEK1**

**Depth first search:**

```
graph = {
  'A': ['B', 'C'],
  'B': ['D', 'E'],
  'C': ['F'],
  'D': [],
  'E': ['F'],
  'F': []
}
visited = set()

def dfs(visited, graph, node):
  if node not in visited:
    print(node)
    visited.add(node)
    for neighbor in graph[node]:
      dfs(visited, graph, neighbor)
dfs(visited, graph, 'A')
```

**o/p:**

```
A
B
D
E
F
C
```

**WEEK-2:**

**Best first search:**

```python
from queue import PriorityQueue


def best_first_search(graph, source, target):
    visited = set()
    pq = PriorityQueue()
    pq.put((0, source))


    while not pq.empty():
        cost, u = pq.get()
        if u in visited:
            continue


        print(u, end=" ")
        visited.add(u)


        if u == target:
            break


        for v, c in graph[u]:
            if v not in visited:
                pq.put((c, v))


    print()


graph = {
    0: [(1, 3), (2, 6), (3, 5)],
    1: [(4, 9), (5, 8)],
    2: [(6, 12), (7, 14)],
    3: [(8, 7)],
    8: [(9, 5), (10, 6)],
    9: [(11, 1), (12, 10), (13, 2)]
```

```
}

source = 0
target = 9
best_first_search(graph, source, target)
```

**o/p:**

0 1 3 2 8 9

**WEEK-3:**

**Depth limit search:**

```
def dls(graph, node, goal, depth_limit):
    if depth_limit == 0 and node != goal:
        return False
    if node == goal:
        return True
    return any(dls(graph, neighbor, goal, depth_limit - 1) for neighbor in graph.get(node, []))

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

start_node, goal_node, depth_limit = 'A', 'F', 3
result = dls(graph, start_node, goal_node, depth_limit)
if result:
    print(f"Goal node '{goal_node}' found within depth limit.")
```

```
else:
    print(f"Goal node '{goal_node}' not found within depth limit.")
```

**o/p:** Goal node 'F' found within depth limit.

**WEEK-4:**

**Heuristic approach**

**WEEK-5:**

**Mini max algorithm:**

```
import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
    if curDepth == targetDepth:
        return scores[nodeIndex]

    if maxTurn:
        return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth))
    else:
        return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = int(math.log2(len(scores)))

print("The optimal value is:", minimax(0, 0, True, scores, treeDepth))
```

**o/p:** The optimal value is: 12

**WEEK-6**

**A* algorithm:**

```
def a_star_algo(start, stop):
    open_set, closed_set, g, parents = {start}, set(), {start: 0}, {start: start}

    while open_set:
        current = min(open_set, key=lambda node: g[node] + heuristic(node))
        if current == stop or not Graph_nodes[current]:
            break

        for neighbor, weight in get_neighbors(current) or []:
            tentative_g = g[current] + weight
            if neighbor not in open_set and neighbor not in closed_set:
                open_set.add(neighbor)
                parents[neighbor] = current
                g[neighbor] = tentative_g
            elif tentative_g < g[neighbor]:
                parents[neighbor] = current
                g[neighbor] = tentative_g

        open_set.remove(current)
        closed_set.add(current)

    path = [current]
    while current != start:
        current = parents[current]
        path.append(current)
    path.reverse()

    return path if stop in path else None

def get_neighbors(v):
    return Graph_nodes.get(v, [])
```

```
def heuristic(n):
    return {'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7, 'G': 0}.get(n, 0)



Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}

result = a_star_algo('A', 'G')
print(f"Path found: {result}" if result else "Path does not exist!")
```

**o/p:** Path found: ['A', 'E', 'D', 'G']

**2----**

```
from queue import PriorityQueue

def heuristic_search(graph, start, goal, heuristic_func):
    open_set = PriorityQueue()
    open_set.put((0, start))
    closed_set = set()
    g = {start: 0}
    parents = {start: start}

    while not open_set.empty():
        _, current = open_set.get()

        if current == goal:
```

```
            path = []
            while current != start:
                path.append(current)
                current = parents[current]
            path.append(start)
            path.reverse()
            return path

        closed_set.add(current)

        for neighbor, cost in graph.get(current, []):
            if neighbor in closed_set:
                continue

            tentative_g = g[current] + cost
            if neighbor not in g or tentative_g < g[neighbor]:
                g[neighbor] = tentative_g
                f_value = tentative_g + heuristic_func(neighbor, goal)
                open_set.put((f_value, neighbor))
                parents[neighbor] = current

    return None

def get_neighbors(v):
    return Graph_nodes.get(v, [])

def manhattan_distance(node, goal):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

# Describe your graph here
Graph_nodes = {
```

```
    (0, 0): [((1, 0), 1), ((0, 1), 1)],
    (1, 0): [((0, 0), 1), ((1, 1), 1)],
    (0, 1): [((0, 0), 1), ((1, 1), 1)],
    (1, 1): [((1, 0), 1), ((0, 1), 1)],
}

start_node = (0, 0)
goal_node = (1, 1)

result = heuristic_search(Graph_nodes, start_node, goal_node, manhattan_distance)

print(f"Path found: {result}" if result else "Path does not exist!")
```

**o/p:** Path found: [(0, 0), (0, 1), (1, 1)]