SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL501 **DATE:15 / 08 / 2024**

COURSE NAME: Artificial Intelligence Laboratory **CLASS: TY-IT**

NAME: Anish Sharma

EXPERIMENT NO.02

CO/LO: Apply various AI approaches to knowledge intensive problem solving, reasoning, planning and uncertainty.

AIM / OBJECTIVE: Implement BFS/DFS search algorithms to reach goal state. (To identify and analyze Uninformed Search Algorithm to solve the problem).

DESCRIPTION OF EXPERIMENT:

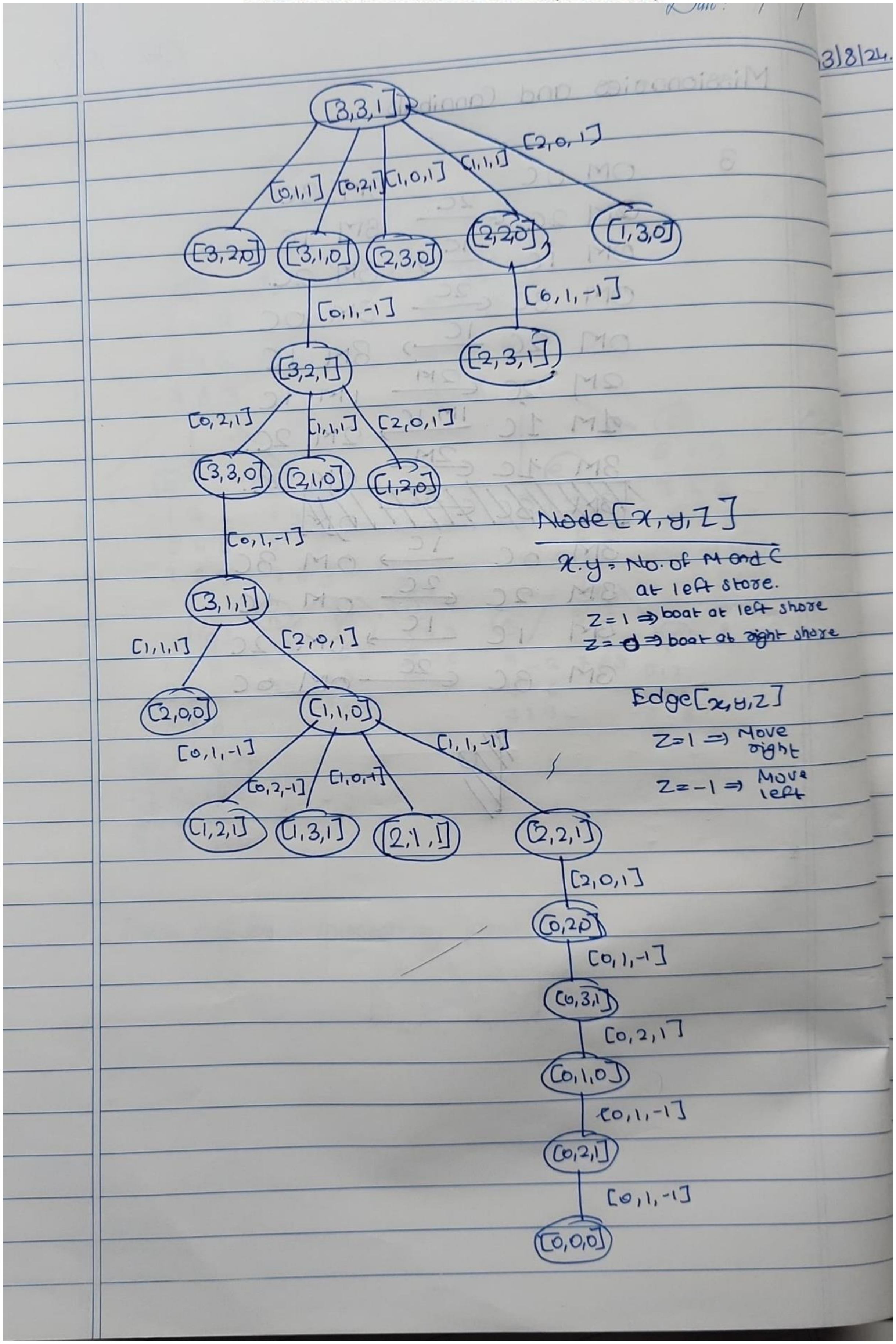
- Students should generate the state space for 8-Puzzle/Tic Tac Toe/Misionaries and Cannibals problem.
- The traversal path for Depth first search and Breadth first search should be displayed.
- Compare and contrast DFS & BFS with respect to time, space complexities and completeness and optimality.
- Generalized solution should be generated for catering any problem.

EXPLANATION / SOLUTIONS (DESIGN):

AI agents solve problems by exploring a search space, which consists of all possible states that can be reached from an initial state through a series of actions. The goal is to find a path from the initial state to a desired goal state. This process begins with problem representation, where the initial state, goal state, state space, and possible actions are defined.

AI agents use search strategies to navigate the search space. These strategies are categorized into uninformed (blind) and informed (heuristic) approaches. Uninformed strategies like Breadth-First Search (BFS) and Depth-First Search (DFS) explore the search space without additional information, while informed strategies like A* Search use heuristics to guide the search, making it more efficient.

By applying these strategies, the agent systematically explores possible paths until it finds a solution or determines that none exists. For example, a vacuum cleaner bot might use these strategies to determine the optimal sequence of moves to clean an entire grid. This method of problem-solving enables AI agents to tackle a wide range of challenges by effectively navigating and optimizing the search space.


3/8/24.
[3,3,1]
[0,1,1]
[0,2,1]
[1,0,1]
[1,1,1]
[2,0,1]
[3,2,0]
[3,1,0]
[2,3,0]
[2,2,0]
[1,3,0]
[0,1,-1]
[0,1,-1]
[3,2,1]
[2,3,1]
[0,2,1]
[1,1,1]
[2,0,1]
[3,3,0]
[2,1,0]
[1,2,0]
[0,1,-1]
[3,1,1]
[1,1,1]
[2,0,1]
[2,0,0]
[1,1,0]
[0,1,-1]
[0,2,-1]
[1,0,-1]
[1,1,-1]
[1,2,1]
[1,3,1]
[2,1,1]
[2,2,1]
[2,0,1]
[0,2,0]
[0,1,-1]
[0,3,1]
[0,2,1]
[0,1,0]
[0,1,-1]
[0,2,1]
[0,1,-1]
[0,0,0]
Node[x,y,Z]
x.y = No. of M and C at left shore.
Z=1 ⇒ boat at left shore
Z=0 ⇒ boat at right shore
Edge[x,y,Z]
Z=1 ⇒ Move right
Z=-1 ⇒ Move left

Code:

```
from collections import deque
from typing import List, Dict, Tuple

# Define a state in the problem
class State:
    def __init__(self, mL: int, cL: int, mR: int, cR: int, boatOnLeft:
bool):
        self.missionariesLeft = mL
        self.cannibalsLeft = cL
        self.missionariesRight = mR
        self.cannibalsRight = cR
        self.boatOnLeft = boatOnLeft

    def __eq__(self, other):
        return (self.missionariesLeft == other.missionariesLeft and
                self.cannibalsLeft == other.cannibalsLeft and
                self.missionariesRight == other.missionariesRight and
                self.cannibalsRight == other.cannibalsRight and
                self.boatOnLeft == other.boatOnLeft)

    def __hash__(self):
        return hash((self.missionariesLeft, self.cannibalsLeft,
                     self.missionariesRight, self.cannibalsRight,
                     self.boatOnLeft))

    def print(self):
        print(f"Left: ({self.missionariesLeft}, {self.cannibalsLeft}) "
              f"Right: ({self.missionariesRight}, {self.cannibalsRight}) "
              f"Boat on {'Left' if self.boatOnLeft else 'Right'}")

def is_valid(state: State) -> bool:
    if (state.missionariesLeft < 0 or state.cannibalsLeft < 0 or
        state.missionariesRight < 0 or state.cannibalsRight < 0 or
        state.missionariesLeft > 3 or state.cannibalsLeft > 3 or
        state.missionariesRight > 3 or state.cannibalsRight > 3):
        return False
    if (state.missionariesLeft < state.cannibalsLeft and
state.missionariesLeft > 0) or \
       (state.missionariesRight < state.cannibalsRight and
state.missionariesRight > 0):
        return False
    return True
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

```
def get_possible_moves(current: State) -> List[State]:
    moves = []
    if current.boatOnLeft:
        if current.missionariesLeft >= 2:
            moves.append(State(current.missionariesLeft - 2,
current.cannibalsLeft,
                               current.missionariesRight + 2,
current.cannibalsRight,
                               not current.boatOnLeft))
        if current.missionariesLeft >= 1:
            moves.append(State(current.missionariesLeft - 1,
current.cannibalsLeft,
                               current.missionariesRight + 1,
current.cannibalsRight,
                               not current.boatOnLeft))
        if current.cannibalsLeft >= 2:
            moves.append(State(current.missionariesLeft,
current.cannibalsLeft - 2,
                               current.missionariesRight,
current.cannibalsRight + 2,
                               not current.boatOnLeft))
        if current.cannibalsLeft >= 1:
            moves.append(State(current.missionariesLeft,
current.cannibalsLeft - 1,
                               current.missionariesRight,
current.cannibalsRight + 1,
                               not current.boatOnLeft))
        if current.missionariesLeft >= 1 and current.cannibalsLeft >= 1:
            moves.append(State(current.missionariesLeft - 1,
current.cannibalsLeft - 1,
                               current.missionariesRight + 1,
current.cannibalsRight + 1,
                               not current.boatOnLeft))
    else:
        if current.missionariesRight >= 2:
            moves.append(State(current.missionariesLeft + 2,
current.cannibalsLeft,
                               current.missionariesRight - 2,
current.cannibalsRight,
                               not current.boatOnLeft))
        if current.missionariesRight >= 1:
            moves.append(State(current.missionariesLeft + 1,
current.cannibalsLeft,
                               current.missionariesRight - 1,
current.cannibalsRight,
                               not current.boatOnLeft))
```

```
        if current.cannibalsRight >= 2:
            moves.append(State(current.missionariesLeft,
current.cannibalsLeft + 2,
                               current.missionariesRight,
current.cannibalsRight - 2,
                               not current.boatOnLeft))
        if current.cannibalsRight >= 1:
            moves.append(State(current.missionariesLeft,
current.cannibalsLeft + 1,
                               current.missionariesRight,
current.cannibalsRight - 1,
                               not current.boatOnLeft))
        if current.missionariesRight >= 1 and current.cannibalsRight >= 1:
            moves.append(State(current.missionariesLeft + 1,
current.cannibalsLeft + 1,
                               current.missionariesRight - 1,
current.cannibalsRight - 1,
                               not current.boatOnLeft))
    return moves

def bfs(start: State) -> bool:
    q = deque([start])
    parent_map: Dict[State, State] = {start: None}
    visited = set([start])
    failed_paths = 0
    total_iterations = 0

    while q:
        total_iterations += 1
        current = q.popleft()

        # Check if we've reached the goal state
        if current.missionariesRight == 3 and current.cannibalsRight == 3:
            # Reconstruct the path
            path = []
            while current is not None:
                path.append(current)
                current = parent_map[current]
            path.reverse()

            # Print the path
            print("BFS Path:")
            for state in path:
                state.print()

            print(f"\nTotal Iterations: {total_iterations}")
```

```
            print(f"Failed Paths: {failed_paths}")
            return True

        for next_state in get_possible_moves(current):
            if is_valid(next_state) and next_state not in visited:
                visited.add(next_state)
                parent_map[next_state] = current
                q.append(next_state)
            else:
                failed_paths += 1
                print("Failed Iteration Node:")
                current.print()  # Print the current node that failed

    print(f"\nTotal Iterations: {total_iterations}")
    print(f"Failed Paths: {failed_paths}")
    return False


def dfs(start: State) -> bool:
    stack = [start]
    parent_map: Dict[State, State] = {start: None}
    visited = set([start])
    failed_paths = 0
    total_iterations = 0

    while stack:
        total_iterations += 1
        current = stack.pop()

        # Check if we've reached the goal state
        if current.missionariesRight == 3 and current.cannibalsRight == 3:
            # Reconstruct the path
            path = []
            while current is not None:
                path.append(current)
                current = parent_map[current]
            path.reverse()

            # Print the path
            print("DFS Path:")
            for state in path:
                state.print()

            print(f"\nTotal Iterations: {total_iterations}")
            print(f"Failed Paths: {failed_paths}")
            return True
```

```
        for next_state in get_possible_moves(current):
            if is_valid(next_state) and next_state not in visited:
                visited.add(next_state)
                parent_map[next_state] = current
                stack.append(next_state)
            else:
                failed_paths += 1
                print("Failed Iteration Node:")
                current.print()  # Print the current node that failed

    print(f"\nTotal Iterations: {total_iterations}")
    print(f"Failed Paths: {failed_paths}")
    return False

def main():
    start = State(3, 3, 0, 0, True)
    print("BFS Result:")
    if not bfs(start):
        print("No solution found using BFS")

    print("\nDFS Result:")
    if not dfs(start):
        print("No solution found using DFS")

if __name__ == "__main__":
    main()
```

Output and traversal path:

```
BFS Result:
Failed Iteration Node:
Left: (3, 3) Right: (0, 0) Boat on Left
Failed Iteration Node:
Left: (3, 3) Right: (0, 0) Boat on Left
Failed Iteration Node:
Left: (3, 1) Right: (0, 2) Boat on Right
Failed Iteration Node:
Left: (3, 2) Right: (0, 1) Boat on Right
Failed Iteration Node:
Left: (2, 2) Right: (1, 1) Boat on Right
Failed Iteration Node:
Left: (2, 2) Right: (1, 1) Boat on Right
Failed Iteration Node:
Left: (2, 2) Right: (1, 1) Boat on Right
Failed Iteration Node:
Left: (3, 2) Right: (0, 1) Boat on Left
Failed Iteration Node:
Left: (3, 2) Right: (0, 1) Boat on Left
Failed Iteration Node:
Left: (3, 2) Right: (0, 1) Boat on Left
Failed Iteration Node:
Left: (3, 2) Right: (0, 1) Boat on Left
Failed Iteration Node:
Left: (3, 0) Right: (0, 3) Boat on Right
Failed Iteration Node:
Left: (3, 1) Right: (0, 2) Boat on Left
Failed Iteration Node:
Left: (3, 1) Right: (0, 2) Boat on Left
Failed Iteration Node:
Left: (3, 1) Right: (0, 2) Boat on Left
Failed Iteration Node:
Left: (1, 1) Right: (2, 2) Boat on Right
Failed Iteration Node:
Left: (1, 1) Right: (2, 2) Boat on Right
Failed Iteration Node:
Left: (1, 1) Right: (2, 2) Boat on Right
Failed Iteration Node:
Left: (1, 1) Right: (2, 2) Boat on Right
```

```
Left: (2, 2) Right: (1, 1) Boat on Left
Failed Iteration Node:
Left: (2, 2) Right: (1, 1) Boat on Left
Failed Iteration Node:
Left: (2, 2) Right: (1, 1) Boat on Left
Failed Iteration Node:
Left: (2, 2) Right: (1, 1) Boat on Left
Failed Iteration Node:
Left: (0, 2) Right: (3, 1) Boat on Right
Failed Iteration Node:
Left: (0, 2) Right: (3, 1) Boat on Right
Failed Iteration Node:
Left: (0, 2) Right: (3, 1) Boat on Right
Failed Iteration Node:
Left: (0, 3) Right: (3, 0) Boat on Left
Failed Iteration Node:
Left: (0, 1) Right: (3, 2) Boat on Right
Failed Iteration Node:
Left: (0, 1) Right: (3, 2) Boat on Right
Failed Iteration Node:
Left: (0, 1) Right: (3, 2) Boat on Right
Failed Iteration Node:
Left: (1, 1) Right: (2, 2) Boat on Left
Failed Iteration Node:
Left: (1, 1) Right: (2, 2) Boat on Left
Failed Iteration Node:
Left: (0, 2) Right: (3, 1) Boat on Left
Failed Iteration Node:
Left: (0, 2) Right: (3, 1) Boat on Left
```

Comparison between BFS and DFS

```
BFS Path:
Left: (3, 3) Right: (0, 0) Boat on Left
Left: (3, 1) Right: (0, 2) Boat on Right
Left: (3, 2) Right: (0, 1) Boat on Left
Left: (3, 0) Right: (0, 3) Boat on Right
Left: (3, 1) Right: (0, 2) Boat on Left
Left: (1, 1) Right: (2, 2) Boat on Right
Left: (2, 2) Right: (1, 1) Boat on Left
Left: (0, 2) Right: (3, 1) Boat on Right
Left: (0, 3) Right: (3, 0) Boat on Left
Left: (0, 1) Right: (3, 2) Boat on Right
Left: (1, 1) Right: (2, 2) Boat on Left
Left: (0, 0) Right: (3, 3) Boat on Right

Total Iterations: 15
Failed Paths: 34
```

```
DFS Path:
Left: (3, 3) Right: (0, 0) Boat on Left
Left: (2, 2) Right: (1, 1) Boat on Right
Left: (3, 2) Right: (0, 1) Boat on Left
Left: (3, 0) Right: (0, 3) Boat on Right
Left: (3, 1) Right: (0, 2) Boat on Left
Left: (1, 1) Right: (2, 2) Boat on Right
Left: (2, 2) Right: (1, 1) Boat on Left
Left: (0, 2) Right: (3, 1) Boat on Right
Left: (0, 3) Right: (3, 0) Boat on Left
Left: (0, 1) Right: (3, 2) Boat on Right
Left: (0, 2) Right: (3, 1) Boat on Left
Left: (0, 0) Right: (3, 3) Boat on Right

Total Iterations: 12
Failed Paths: 28
```

Observation Sheet Questions:

1. **Explain how AI agents solve a problem using search strategies.**
2. **Apply uninformed search strategies for a vacuum cleaner bot and design the state space for the problem.**

CONCLUSION:

In conclusion, AI agents solve problems by systematically exploring a defined search space using various search strategies. By representing the problem in terms of states, actions, and goals, an AI agent can apply uninformed search strategies, such as Breadth-First Search (BFS) or Depth-First Search (DFS), to navigate the search space and find a solution.

REFERENCES:

[1] Stuart Russell and Peter Norvig, “Artificial Intelligence: A Modern Approach”, 2nd Edition, Pearson Education, 2010