Name	**Hatim Yusuf Sawai**
UID no.	**2021300108**
Experiment No.	**2**

AIM:	**Solve given problem using BFS**
PROBLEM DEFINITION:	**Implement a Triangle Peg game using the Uninformed searching technique BFS and analyze the algorithms with respect to Completeness, Optimality, time, and space Complexity.** **Rules:** If you haven't played it before, the rules are straightforward: 1. There are 14 pegs that are placed in a triangle made of 15 holes. 2. One hole is open at the beginning of every game. 3. The choice of this hole is up to the player. 4. The goal of the game is to eliminate all but 1 peg from play. 5. Pegs can be eliminated by jumping one peg over another. 6. A jump consists of picking up one peg, moving over another peg, and placing it down in an empty hole. The jumped peg is then removed from the board.
THEORY:	**The Triangle Peg Problem:** Imagine you have a triangular board with 15 holes arranged in the shape of a triangle. Initially, there are pegs in all the holes except one. The goal of the game is to jump pegs over one another and remove them from the board until only one peg remains. It's like a puzzle, and it might sound simple, but it's surprisingly tricky! The Triangle Peg Problem is a classic example in the field of artificial intelligence and machine learning, often used to illustrate concepts related to search algorithms, problem-solving, and optimization. This puzzle presents an intriguing challenge that underscores the complexity of certain combinatorial problems and the need for advanced AI and ML techniques to tackle them.

The Setup:

- You have a triangle-shaped board with 15 holes, and you place 14 pegs in these holes at the beginning.
- There's one empty hole to start with, and you can choose its location.
- The goal is to eliminate all but one peg from the board.

Strategy using BFS:

Step 1: Initialization

- Start with the initial state of the game, which is a configuration of the board with one hole vacant and pegs in the remaining holes.
- Create an empty queue to store states.

Step 2: Enqueue the Initial State

- Enqueue the initial state into the queue. This state represents the starting position of the game.

Step 3: Main BFS Loop

- Enter a loop that continues until a solution is found or the queue becomes empty.

Step 4: Dequeue a State

- Dequeue (remove) the front state from the queue. This state represents the current position of the game.

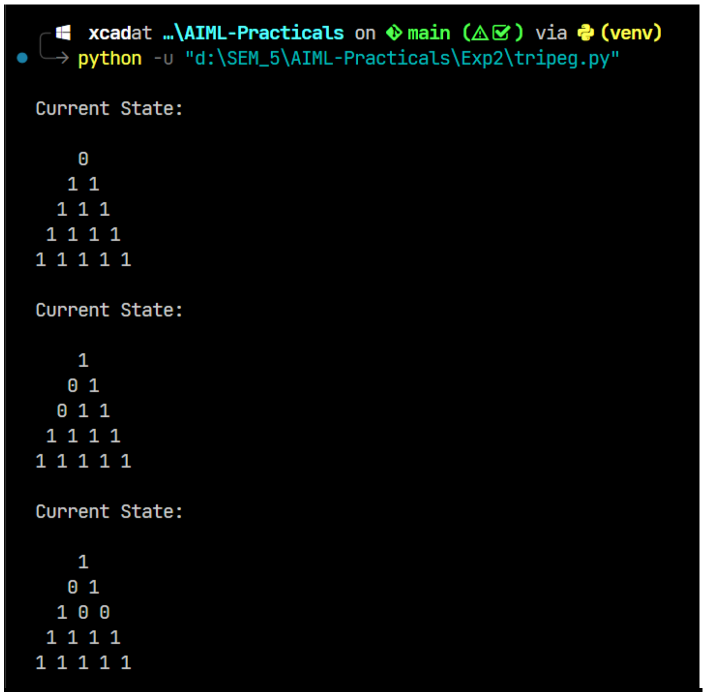
Step 5: Check for the Goal State

- Check if the current state is the goal state, where only one peg remains on the board. If it is, you have found a solution, and you can exit the loop.

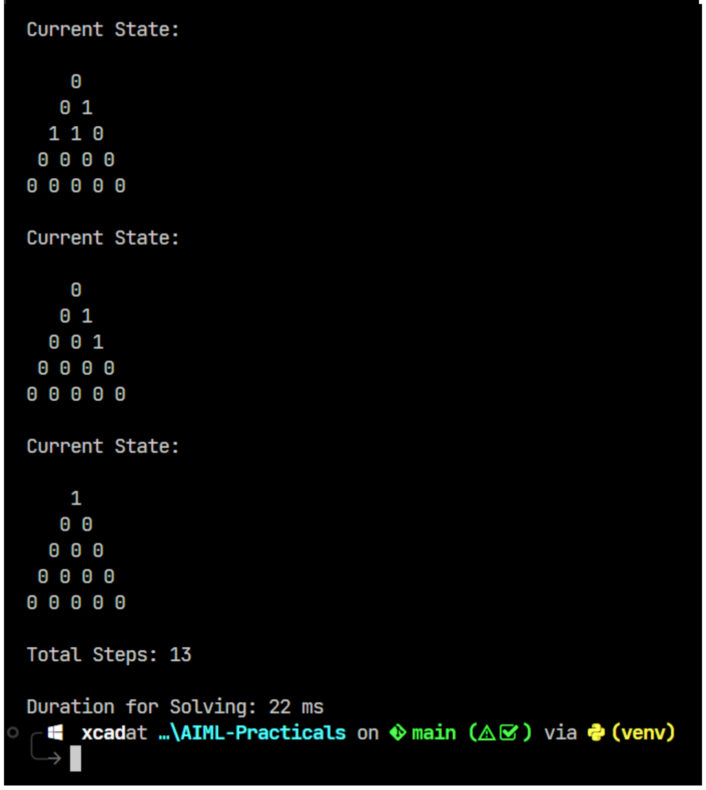
	Step 6: Generate Valid Moves • If the current state is not the goal state, generate all valid moves from the current state. Valid moves involve jumping one peg over another into an empty hole, following the game's rules. **Step 7: Enqueue New States** • For each valid move, create a new state representing the board's configuration after the move. • Enqueue these new states into the queue if they have not been visited before. Keep track of visited states to avoid revisiting them. **Step 8: Repeat** • Repeat steps 4 to 7 until a solution is found or the queue is empty. **Step 9: Backtrack to Find the Solution**
CODE:	see below

```
import time

def make_move(pos, t, sub):
    new_pos = pos.copy()
    for i in range(3):
        new_pos[t[i]] = sub[i]
    return new_pos

def trianglepeg(pos, goal):
    queue, solution = [], []
    path = {tuple(pos): None}
    while pos != goal:
        for next_move in iterate_puzzle(pos):
            c = tuple(next_move)
            if c in path:
                continue
            path[c] = pos
            queue.append(next_move)
        if len(queue) == 0:
            return None
        pos = queue.pop(0)
    while pos:
        solution.insert(0, pos)
        pos = path[tuple(pos)]
    return solution

def iterate_puzzle(pos):
    valid_moves = [[0,1,3], [1,3,6], [3,6,10], [2,4,7], [4,7,11], [5,8,12],
                   [10,11,12], [11,12,13], [12,13,14], [6,7,8], [7,8,9], [3,4,5],
                   [0,2,5], [2,5,9], [5,9,14], [1,4,8], [4,8,13], [3,7,12]]
    for t in valid_moves:
        if pos[t[0]] == 1 and pos[t[1]] == 1 and pos[t[2]] == 0:
            yield make_move(pos, t, [0, 0, 1])
        if pos[t[0]] == 0 and pos[t[1]] == 1 and pos[t[2]] == 1:
            yield make_move(pos, t, [1, 0, 0])

def main():
    initial_board = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    goal = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    t = time.time()
    solution = trianglepeg(initial_board, goal)
    d = time.time() - t
    for s in solution:
        print("\nCurrent State:\n")
        for i in range(5):
            print(' ' * (4 - i), end='')
            row = s[i * (i + 1) // 2:i * (i + 1) // 2 + i + 1]
            print(' '.join(map(str, row)))
    print("\nTotal Steps: %d" % (len(solution) - 1))
    print("\nDuration for Solving: %d ms" % (d * 1000))


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

OUTPUT:



```
 xcadat …\AIML-Practicals on main (⚠✓) via (venv)
● └→ python -u "d:\SEM_5\AIML-Practicals\Exp2\tripeg.py"

Current State:

    0
   1 1
  1 1 1
 1 1 1 1
1 1 1 1 1

Current State:

    1
   0 1
  0 1 1
 1 1 1 1
1 1 1 1 1

Current State:

    1
   0 1
  1 0 0
 1 1 1 1
1 1 1 1 1
```



```
Current State:

    0
   0 1
  1 1 0
 0 0 0 0
0 0 0 0 0

Current State:

    0
   0 1
  0 0 1
 0 0 0 0
0 0 0 0 0

Current State:

    1
   0 0
  0 0 0
 0 0 0 0
0 0 0 0 0

Total Steps: 13

Duration for Solving: 22 ms
○ xcadat …\AIML-Practicals on main (⚠✓) via (venv)
  └→ █
```

CONCLUSION:	In this experiment, we learned how to implement a solution to the triangle peg problem using breadth-first search algorithm.