

PART A

Experiment No.07

A.1 Aim: Implement Monkey Banana Problem using Python

A.2 Prerequisite: Discrete structure

A.3 Outcome:

After successful completion of this experiment students will be able to

- . Solve real life problems using AI

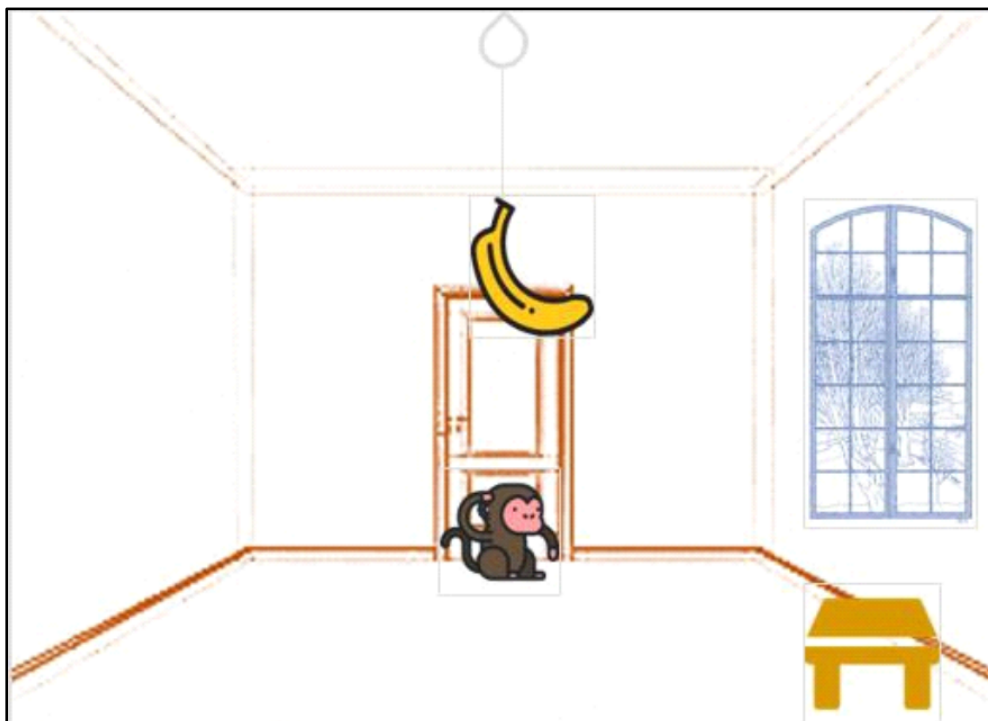
Tools Required: Python / Java

A.4 Theory:

Problem Statement

Suppose the problem is as given below –

- A hungry monkey is in a room, and he is near the door.
- The monkey is on the floor.
- Bananas have been hung from the center of the ceiling of the room.
- There is a block (or chair) present in the room near the window.
- The monkey wants the banana, but cannot reach it.



So how can the monkey get the bananas?

So if the monkey is clever enough, he can come to the block, drag the block

to the center, climb on it, and get the banana. Below are few observations in this case –

- Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.
- If the block position is not at the center, then monkey can drag it to the center.
- If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
- When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

Now, let us see how we can solve this using Prolog. We will create some predicates as follows –

We have some predicates that will move from one state to another state, by performing action.

- When the block is at the middle, and monkey is on top of the block, and monkey does not have the banana (i.e. **has not** state), then using the **grasp** action, it will change from **has not** state to **have** state.
- From the floor, it can move to the top of the block (i.e. **on top** state), by performing the action **climb**.
- The **push** or **drag** operation moves the block from one place to another.
- Monkey can move from one place to another using **walk** or **move** clauses.

Another predicate will be `canget()`. Here we pass a state, so this will perform move predicate from one state to another using different actions, then perform `canget()` on state 2. When we have reached to the state '**has>**', this indicates '**has banana**'. We will stop the execution.

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

| | |
|-------------------------------|----------------------------|
| Roll. No.B32 | Name:Mahesh Suresh Bhosale |
| Class:TE B Comps | Batch:B2 |
| Date of Experiment:08/03/2025 | Date of Submission: |
| Grade: | |

B.1 Software Code written by student:

Python code:

```

from collections import deque

def monkey_banana_problem():
    # Initial state
    initial_state = ('Far-Chair', 'Chair-Not-Under-Banana', 'Off-Chair', 'Empty') # (Monkey's
    Location, Monkey's Position on Chair, Chair's Location, Monkey's Status)
    print(f"\nInitial state is {initial_state}")
    goal_state = ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Holding') # The goal state when
    the monkey has the banana

    # Possible actions and their effects
    actions = {
        "Move to Chair": lambda state: ('Near-Chair', state[1], state[2], state[3]) if state[0] != 'Near-
        Chair' else None,
        "Push Chair under Banana": lambda state: ('Near-Chair', 'Chair-Under-Banana', state[2],
        state[3]) if state[0] == 'Near-Chair' and state[1] != 'Chair-Under-Banana' else None,
        "Climb Chair": lambda state: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', state[3]) if
        state[0] == 'Near-Chair' and state[1] == 'Chair-Under-Banana' and state[2] != 'On-Chair' else
        None,
        "Grasp Banana": lambda state: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Holding') if
        state[0] == 'Near-Chair' and state[1] == 'Chair-Under-Banana' and state[2] == 'On-Chair' and
        state[3] != 'Holding' else None
    }

    # BFS to explore states
    dq = deque([(initial_state, [])]) # Each element is (current_state, actions_taken)
    visited = set()

    while dq:
        current_state, actions_taken = dq.popleft()

        # Check if we've reached the goal
        if current_state == goal_state:
            print("\nSolution Found!")
            print("Actions to achieve goal:")
            for action in actions_taken:
                print(action)
            print(f"Final State: {current_state}")
            return

        # Mark the current state as visited
        if current_state in visited:
            continue
        visited.add(current_state)

        # Try all possible actions
        for action_name, action_func in actions.items():
            next_state = action_func(current_state)
            if next_state and (next_state not in visited):
                dq.append((next_state, actions_taken + [f"Action: {action_name}, Resulting State:
                {next_state}"]))

    print("No solution found.")

# Run the program
monkey_banana_problem()

```

Prolog code:

% Initial state: monkey is at door, on floor, without banana

% Banana is at middle ceiling, block is at window

% Actions: walk, push, climb, grasp

% Define possible moves

move(state(middle, onbox, middle, hasnot), grasp, state(middle, onbox, middle, has)).

move(state(P, onfloor, P, H), climb, state(P, onbox, P, H)).

move(state(P1, onfloor, P1, H), push(P1, P2), state(P2, onfloor, P2, H)).

move(state(P1, onfloor, B, H), walk(P1, P2), state(P2, onfloor, B, H)).

% Define goal state

canget(state(_, _, _, has)).

% Recursive rule to reach goal state

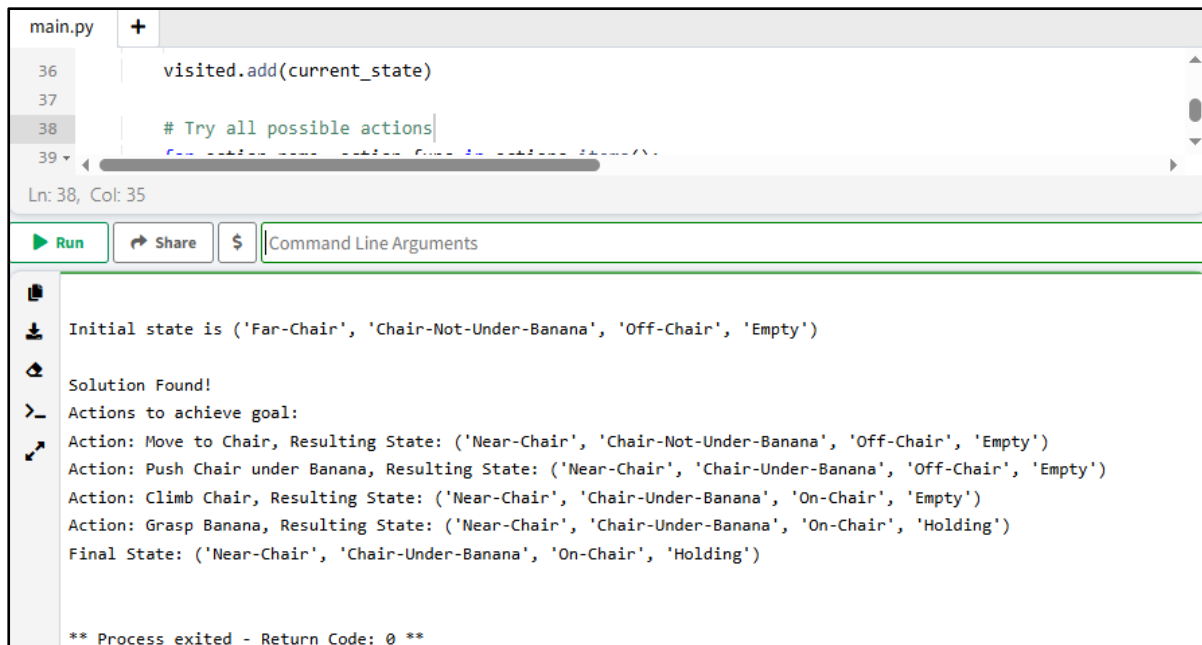
canget(State1) :-

move(State1, _, State2),

canget(State2).

B.2 Input and Output:

Python output:



The screenshot shows a Python IDE with a file named 'main.py'. The code in the editor includes a list 'visited' and a function 'canget' that uses a recursive rule to find a solution. The output window displays the following text:

```
Initial state is ('Far-Chair', 'Chair-Not-Under-Banana', 'Off-Chair', 'Empty')
Solution Found!
Actions to achieve goal:
Action: Move to Chair, Resulting State: ('Near-Chair', 'Chair-Not-Under-Banana', 'Off-Chair', 'Empty')
Action: Push Chair under Banana, Resulting State: ('Near-Chair', 'Chair-Under-Banana', 'Off-Chair', 'Empty')
Action: Climb Chair, Resulting State: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Empty')
Action: Grasp Banana, Resulting State: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Holding')
Final State: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Holding')
** Process exited - Return Code: 0 **
```

Prolog output

```

SWI-Prolog (AMD64, Multi-threaded, version 9.3.21)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.3.21)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit 8;;https://www.swi-prolog.orghttps://www.swi-prolog.org8;;
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/intaz/OneDrive/Documents/Prolog/monkey.pl compiled 0.00 sec, 6 clauses
?-
| [monkey].
true.

?- canget(state(door, onfloor, window, hasnot)).
true.

?- canget(state(middle, onfloor, middle, hasnot)).
true.

?- canget(state(window, onfloor, window, hasnot)).
true.

?- canget(state(door, onfloor, middle, hasnot)).
true.

?- canget(state(middle, onbox, middle, hasnot)).
true.

?- canget(state(door, onfloor, middle, has)).
true.

?- canget(state(door, onbox, window, hasnot)).
false.

?- canget(state(door, onbox, door, hasnot)).
false.

?- ■

```

B.3 Observations and learning:

1. **State Representation:** The problem was modeled using a State class, which tracked the monkey's position, the block's position, and whether the monkey had the banana.
2. **Action-based Transitions:** The problem was solved by defining specific actions (moving the monkey, dragging the block, climbing, and grasping the banana) that transitioned the system between states.
3. **Breadth-First Search (BFS):** BFS was used to explore all possible states and actions. It ensured the shortest sequence of actions to reach the goal (monkey with the banana).
4. **Goal State:** The goal was defined as the state where the monkey has the banana. BFS successfully identified this state by exploring all possible transitions.
5. **Efficiency of BFS:** BFS was effective in solving the problem, as it guarantees the shortest path to the goal, making it suitable for this kind of state-space search problem.

B.4 Conclusion:

In conclusion, this experiment has helped me understand how to model a real-world problem using AI techniques like state representation and search algorithms. Specifically, by using the Monkey Banana Problem, I learned the importance of breaking a complex problem into smaller, manageable actions and using search algorithms (like BFS) to explore different possible solutions efficiently. The problem was solved by systematically transitioning through states, and the BFS approach ensured that we found the shortest path to the goal.

Through this experiment, I also learned that modeling problems in terms of states and actions is a powerful technique in AI, and search algorithms such as BFS are essential tools for solving such problems. The key takeaway is that AI can be used to simulate problem-solving behavior in complex environments and find optimal solutions.

