AY: 2025-26

Class:	**TE**	**Semester:**	**V**
Course Code:	**CSL 502**	**Course Name:**	**Artificial Intelligence**

Name of Student:	**Pranita Kumbhar**
Roll No. :	**70**
Experiment No.:	**04**
Title of the Experiment:	**Implementation of Bidirectional search for problem solving**.
Date of Performance:	**25/07/25**
Date of Submission:	**01/08/25**

Evaluation

Performance Indicator	**Max. Marks**	**Marks Obtained**
Performance	5	
Understanding	5	
Journal work and timely submission	10	
Total	20	

Performance Indicator	**Exceed Expectations (EE)**	**Meet Expectations (ME)**	**Meet Expect Below Expectations (BE)**
Performance	4-5	2-3	1
Understanding	4-5	2-3	1
Journal work and timely submission	8-10	5-8	1-4

Checked by
Name of Faculty : Ms. Rujuta Vartak
Signature :
Date:

Aim: Implementation of Bidirectional search for problem solving.

Objective: To study the Bidirectional searching techniques and its implementation for problem solving.

Theory:

Bidirectional search is a graph search algorithm which find smallest path from source to goal vertex. It runs two simultaneous search –

1. Forward search from source/initial vertex toward goal vertex
2. Backward search from goal/target vertex toward source vertex

Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. The search terminates when two graphs intersect.

Algorithm:

Steps for Bidirectional Search Algorithm

1. **Initialization**:
   - Create two frontiers:
     - One for the forward search starting from the initial node (start).
     - One for the backward search starting from the goal node (goal).
   - Create two sets to keep track of visited nodes for each search direction:
     - visited_start for the forward search.
     - visited_goal for the backward search.
   - Initialize the frontiers by adding the start node to frontier_start and the goal node to frontier_goal.
   - Initialize the visited_start and visited_goal sets with their respective starting nodes.
2. **Search Expansion**:
   - Repeat the following steps until a meeting point is found or one of the frontiers is empty:
     - **Expand Forward Search**:
     - Remove the current node from frontier_start.
     - Expand all neighboring nodes of the current node.

- For each neighbor:
- If the neighbor is not in visited_start:
- Add the neighbor to frontier_start.
- Mark the neighbor as visited in visited_start.
- Check if the neighbor is already in visited_goal:
- If yes, a meeting point is found. Proceed to reconstruct the path.

- **Expand Backward Search**:

- Remove the current node from frontier_goal.
- Expand all neighboring nodes of the current node.
- For each neighbor:
- If the neighbor is not in visited_goal:
- Add the neighbor to frontier_goal.
- Mark the neighbor as visited in visited_goal.
- Check if the neighbor is already in visited_start:
- If yes, a meeting point is found. Proceed to reconstruct the path

3. **Meeting Point**:
   - The search stops when a node from frontier_start is found in visited_goal or a node from frontier_goal is found in visited_start. This node is the meeting point.
4. **Path Reconstruction**:
   - Reconstruct the path from the start node to the goal node by combining the paths from both

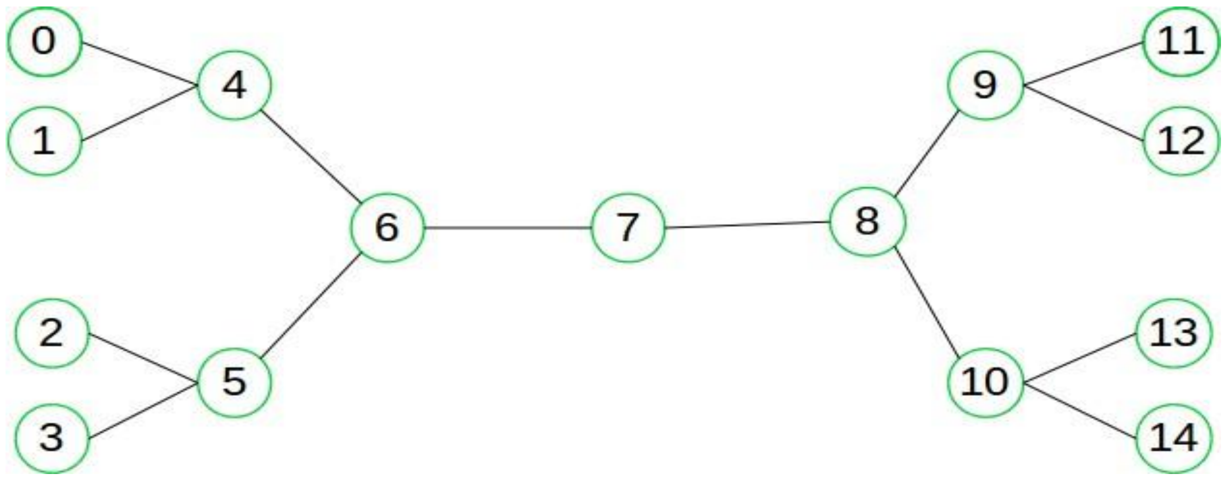
searches at the meeting point.

- Start from the meeting point:
  - Trace back to the start node using the information in visited_start.
  - Trace back to the goal node using the information in visited_goal.
- Concatenate the two paths to form the complete path from start to goal.

5. **Termination**:

- If one of the frontiers is empty and no meeting point is found, it means there is no path from the start node to the goal node

Example:



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.

When to use bidirectional approach?

We can consider bidirectional approach when-

1. Both initial and goal states are unique and completely defined.
2. The branching factor is exactly the same in both directions.

Performance measures

Completeness: Bidirectional Search is complete if we use BFS in both searches. **Time Complexity:** Time complexity of bidirectional search using BFS is **O(b^d)**. **Space Complexity:** Space complexity of bidirectional search is **O(b^d)**.

Optimal: Bidirectional search is Optimal.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

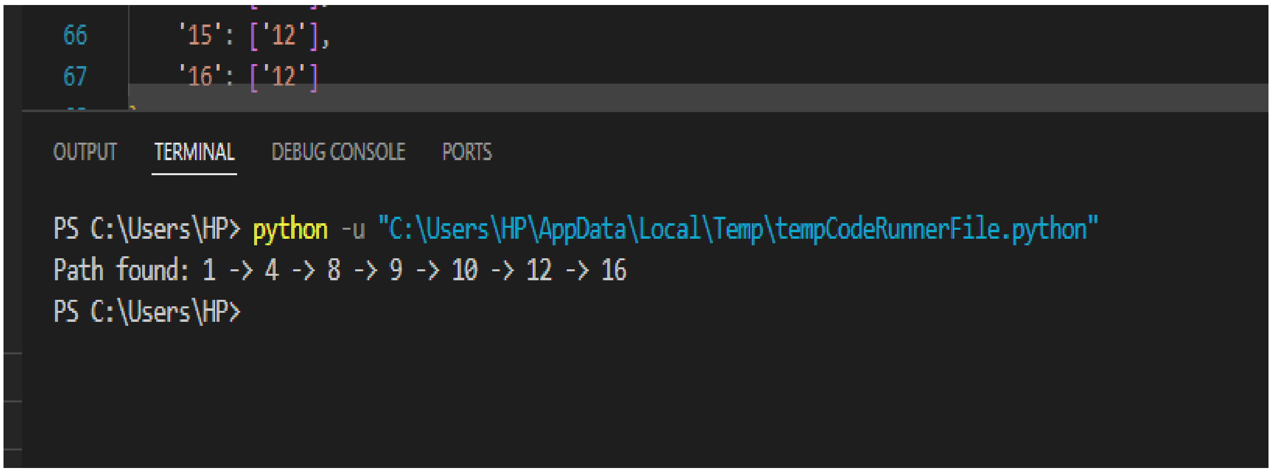
Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

PROGRAM-

```
from collections import deque

def bidirectional_search(start, goal):
    if start == goal:
        return [start]

    front_queue = deque([start])
    back_queue = deque([goal])

    front_visited = {start: None}
    back_visited = {goal: None}
    while front_queue and back_queue:
        # Forward search step
        current_front = front_queue.popleft()
        for neighbor in graph.get(current_front, []):
            if neighbor not in front_visited:
                front_visited[neighbor] = current_front
                front_queue.append(neighbor)
                if neighbor in back_visited:
                    return build_path(neighbor, front_visited, back_visited)

        # Backward search step
        current_back = back_queue.popleft()
        for neighbor in graph.get(current_back, []):
            if neighbor not in back_visited:
                back_visited[neighbor] = current_back
                back_queue.append(neighbor)
                if neighbor in front_visited:
                    return build_path(neighbor, front_visited, back_visited)

    return None

def build_path(meeting_node, front_visited, back_visited):
    # Reconstruct path from start to meeting_node
    path_front = []
    node = meeting_node
    while node is not None:
        path_front.append(node)
        node = front_visited[node]
    path_front.reverse()

    # Reconstruct path from meeting_node to goal
    path_back = []
    node = back_visited[meeting_node]
    while node is not None:
        path_back.append(node)
        node = back_visited[node]

    return path_front + path_back
# Fixed undirected graph
graph = {
    '1': ['4'],
    '2': ['4'],
    '3': ['6'],
    '4': ['1', '2', '8'],
    '5': ['6'],
    '6': ['3', '5', '8'],
    '8': ['4', '6', '9'],
    '9': ['8', '10'],
    '10': ['9', '11', '12'],
    '11': ['10', '13', '14'],
    '12': ['10', '15', '16'],
    '13': ['11'],
    '14': ['11'],
    '15': ['12'],
    '16': ['12']
}

# Test example
start = '1'
goal = '16'
path = bidirectional_search(start, goal)
if path:
    print("Path found:", " -> ".join(path))
else:
    print("No path found.")
```

OUTPUT-



```
    '15': ['12'],
    '16': ['12']
```

```
PS C:\Users\HP> python -u "C:\Users\HP\AppData\Local\Temp\tempCodeRunnerFile.python"
Path found: 1 -> 4 -> 8 -> 9 -> 10 -> 12 -> 16
PS C:\Users\HP>
```

Conclusion:

Bidirectional search is used in real-world applications like GPS navigation for finding the shortest routes, social networks for discovering connections, robot path planning in complex environments, network routing for efficient data transmission, and puzzle solving to find solutions faster. It reduces the search space by exploring from both the start and goal simultaneously, improving efficiency.irectional search techniques.