Experiment No.4
Implementation of Bidirectional search for problem solving.
Date of Performance:
Date of Submission:

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.

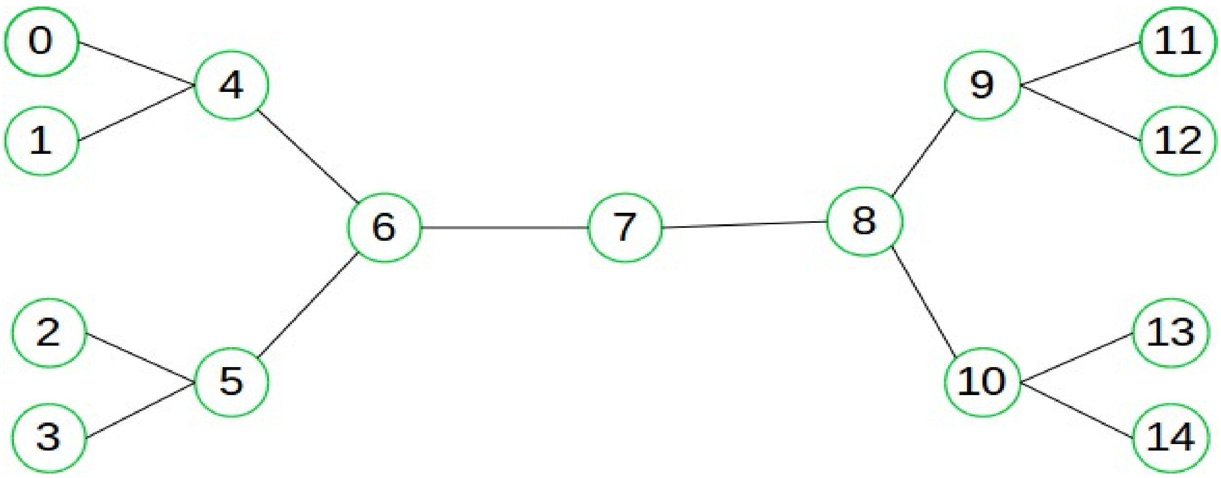
3. **Meeting Point**:
   - The search stops when a node from frontier_start is found in visited_goal or a node from frontier_goal is found in visited_start. This node is the meeting point.
4. **Path Reconstruction**:
   - Reconstruct the path from the start node to the goal node by combining the paths 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 $\mathbf{O(b^d)}$.

Space Complexity: Space complexity of bidirectional search is $\mathbf{O(b^d)}$.

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.**

Code:

```
from collections import deque

# Bidirectional search function
def bidirectional_search(graph, start, goal):
    # Initialize visited sets for both directions
    visited_from_start = {start}
    visited_from_goal = {goal}

    # Queues for BFS from both directions
    queue_start = deque([start])
    queue_goal = deque([goal])

    # Parent dictionaries to reconstruct the path
    parent_start = {start: None}
    parent_goal = {goal: None}

    # Perform BFS from both directions
    while queue_start and queue_goal:
        # Expand from start side
        if queue_start:
            node_start = queue_start.popleft()
            for neighbor in graph[node_start]:
                if neighbor not in visited_from_start:
                    visited_from_start.add(neighbor)
```

```
                parent_start[neighbor] = node_start
                queue_start.append(neighbor)

                # If a connection is found, reconstruct the path
                if neighbor in visited_from_goal:
                    return reconstruct_path(parent_start, parent_goal, neighbor)

        # Expand from goal side
        if queue_goal:
            node_goal = queue_goal.popleft()
            for neighbor in graph[node_goal]:
                if neighbor not in visited_from_goal:
                    visited_from_goal.add(neighbor)
                    parent_goal[neighbor] = node_goal
                    queue_goal.append(neighbor)

                    # If a connection is found, reconstruct the path
                    if neighbor in visited_from_start:
                        return reconstruct_path(parent_start, parent_goal, neighbor)

    return None  # No path found

# Function to reconstruct the path from the meeting point
def reconstruct_path(parent_start, parent_goal, meeting_point):
    # Reconstruct the path from start to meeting point
    path_start = []
    node = meeting_point
    while node is not None:
        path_start.append(node)
        node = parent_start[node]
    path_start.reverse()

    # Reconstruct the path from meeting point to goal
    path_goal = []
    node = parent_goal[meeting_point]
    while node is not None:
        path_goal.append(node)
        node = parent_goal[node]

    # Combine both parts of the path
    return path_start + path_goal

# Example usage
if __name__ == "__main__":
    # Example graph as an adjacency list
    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']
    }

    start_node = 'A'
    goal_node = 'F'
    result = bidirectional_search(graph, start_node, goal_node)
```

```
if result:
    print("Path found:", result)
else:
    print("No path found")
```

Output

```
Path found: ['A', 'C', 'F']
```

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.