BIDIRECTIONAL SEARCH ALGORITHM

UNDER THE SUPERVISION OF:
DR/ MOHAMED REHAN

BIDIRECTIONAL SEARCH ALGORITHM

Bidirectional search is a graph traversal algorithm designed to efficiently find a path between a start node and a goal node. Unlike traditional approaches that search from the start to the goal in a single direction, bidirectional search performs two simultaneous searches — one forward from the start and one backward from the goal — aiming to meet in the middle. This strategy drastically reduces the number of nodes explored and is particularly useful in large search spaces with high branching factors.

This technique is widely used in AI pathfinding, GPS routing systems, and network optimization, where speed and memory efficiency are crucial.

How Bidirectional Search Works:

1. **Initialization**:
   Create two frontiers:
- The first begins at the start node.
- The second begins at the goal node.

  Each search maintains a set of visited nodes.

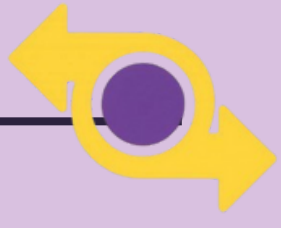
2. **Search Expansion:**
   Alternate expansions from each frontier. For each node:
- Add all unvisited neighbors to the respective frontier.
- Track visited nodes to avoid duplication.

3. **Intersection Check:**

After every expansion step, check whether any node appears in both visited sets. If found, the two searches have intersected a path has been found.

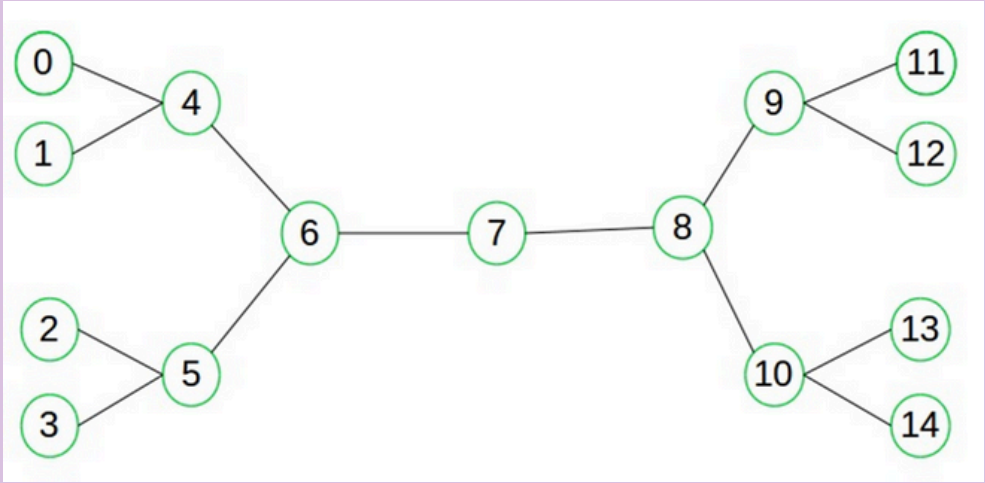
4. **Path Construction:**

Construct the full path by combining the path from the start to the meeting point and from the goal to the meeting point (in reverse).



BIDIRECTIONAL SEARCH ALGORITHM

Example: Find a way from 0 to 14 using bidirectional search



Solution:

Start:0 Goal:14

Initial Setup

- Forward Frontier(from 0): {0}
- Backward Frontier (from 14): {14}
- Visited (forward): {0}
- Visited (backward): {14}

Step 1

- From 0 → neighbor is 4
- Forward Frontier: {4}
- Visited (forward): {0, 4}
- From 14 → neighbor is 10
- Backward Frontier: {10}
- Visited (backward): {14,10}

Step 2

- From 4 → neighbors: {1, 6, 0}
- Add 1 and 6 (0 already visited)
- Forward Frontier: {1, 6}
- Visited (forward): {0, 1, 4, 6}
- From 10 → neighbors: {8, 13, 14}
- Add 8 and 13
- Backward Frontier: {8, 13}
- Visited (backward): {10, 13, 14, 8}

Step 3

- From 1 → neighbor is 4 (already visited)
- From 6 → neighbors: {4, 5, 7}
- Add 5 and 7
- Forward Frontier: {5, 7}
- Visited (forward): {0, 1, 4, 5, 6, 7}
- From 8 → neighbors: {7, 9, 10}
- Add 9 (7 and 10 already visited)
- Backward Frontier: {13, 9, 7}
- Visited (backward): {8, 9, 10, 13, 14, 7}

Meeting Point Found: Node 7
Node 7 is now in both forward and backward visited sets!

BIDIRECTIONAL SEARCH ALGORITHM

Trace the Full Path

- **From start (0):**

0 → 4 → 6 → 7

- **From goal (14):**

14 ← 10 ← 8 ← 7→ reversed becomes 7 → 8 → 10 → 14

Final Path from 0 to 14

0 → 4 → 6 → 7 → 8 → 10 → 14

Total steps: 6

Real-Life Example

A common real-world use case of bidirectional search is in Google Maps or other GPS-based routing systems. When calculating the shortest driving route from your current location to a distant destination, bidirectional search can optimize performance by exploring the road network from both ends (start and goal). This significantly reduces the time and memory needed to find the optimal path, especially in large cities with complex road systems.

Evaluation:

1. Completeness:

Yes, the algorithm is complete if the graph is finite and a path exists.

2. Optimality:

Yes, optimal if both directions use BFS in unweighted graphs or UCS in weighted graphs.

3. The time complexity of bidirectional search is O(b^d/2)

b = branching factor

d = depth of the solution

4. The space complexity of bidirectional search is also O(b^d/2).

Memory usage is higher due to storing two frontiers and visited sets, but still significantly more efficient overall in many scenarios.

Evaluation of Bidirectional Search

- **Pros:**
  - Reduces search time drastically.
  - More efficient in sparse graphs with large branching factors.
  - It often requires less memory than traditional algorithms like BFS, especially in densely connected graphs or large search spaces.
- **Cons:**
  - Requires the ability to reverse actions.
  - Implementation complexity increases with state comparison and path reconstruction.
  - The user should be aware of the goal state