**Session-5**

**Uninformed Search Strategies and Informed Search Strategies**

1. **Iterative deepening depth-first search (IDFS):**

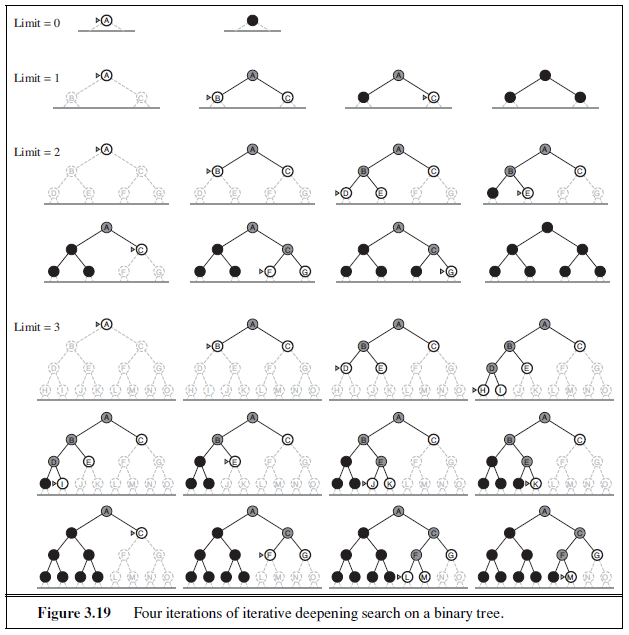
**Iterative deepening search** (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.

This will occur when the depth limit reaches d, the depth of the shallowest goal node. In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

**Algorithm:**

* 1. Repeatedly applies Depth Limited Search with increasing limits.
  2. It terminates when a solution is found or if the Depth Limited Search returns failure, meaning that no solution exists.

**Eg.**



Iterative deepening search is analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer. It would seem worthwhile to develop an iterative analogy to uniform-cost search, inheriting the latter algorithm’s optimality guarantees while avoiding its memory requirements. The idea is to use

increasing path-cost limits instead of increasing depth limits. The resulting algorithm, called **iterative lengthening search.**

**Advantages:**

* 1. It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

**Disadvantages:**

* 1. The main drawback of IDDFS is that it repeats all the work of the previous phase.

**Performance of IDFS:**

**Completeness:**

This algorithm is complete is if the branching factor is finite.

**Optimal:**

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

**Time Complexity:**

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(bd)**.

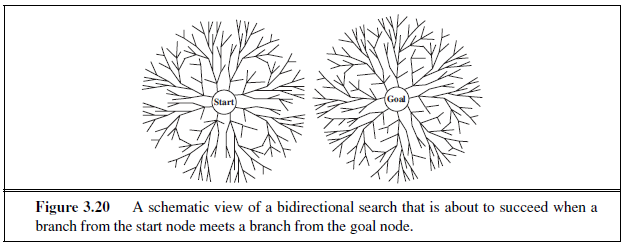
**Space Complexity:**

The space complexity of IDDFS will be **O(bd)**.

1. **Bidirectional Search:** The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle (Figure 3.20). The motivation is that bd/2 + bd/2 is much less than bd, or in the figure, the area of the two small circles is less than the area of one big circle centred on the start and reaching to the goal.

* Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found. It is important to realize that the first such solution found may not be optimal, even if the two searches are both breadth-first; some additional search is required to make sure there isn’t another short-cut across the gap. The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time. Bidirectional search is a [graph search algorithm](https://en.wikipedia.org/wiki/Graph_search_algorithm) that finds a [shortest path](https://en.wikipedia.org/wiki/Shortest_path) from an initial [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) to a goal vertex in a [directed graph](https://en.wikipedia.org/wiki/Directed_graph).

**Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.**



**Algorithm:**

* 1. **Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.**
  2. **Then we run two simultaneous searches**
     + 1. **from initial state called as forward-search**
       2. **other from goal node called as backward-search**

**3. The search stops when these two graphs intersect each other.**

**Eg.** In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.

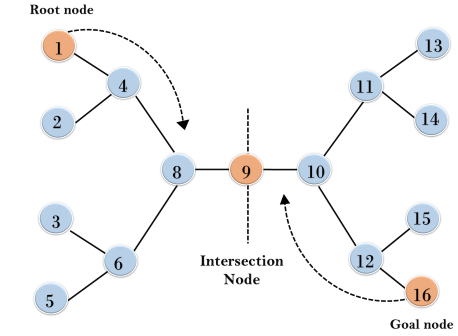


Fig. Illustrating Bidirectional example

**Advantages:**

1. Bidirectional search is fast.
2. Bidirectional search requires less memory

**Disadvantages:**

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

**Performance of Bidirectional Search:**

**Completeness:** Bidirectional Search is complete if we use BFS in both searches.

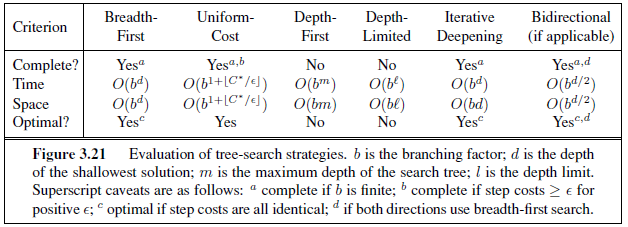
**Optimal:** Bidirectional search is Optimal.

**Time Complexity:** Time complexity of bidirectional search using BFS is **O(bd)**.

**Space Complexity:** Space complexity of bidirectional search is **O(bd)**.

**Comparing uninformed search strategies:**

Figure 3.21 compares search strategies in terms of the four evaluation criteria. This comparison is for tree-search versions. For graph searches, the main differences are that depth-first search is complete for finite state spaces and that the space and time complexities are bounded by the size of the state space.



**Informed Search**

The uninformed search algorithms search the search space for all possible solutions of the problem without having any additional knowledge about search space except the given problem state space. But **informed search algorithm** contains additional information such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge helps agents to explore less to the search space and find more efficiently the goal node.

**Heuristics function:** Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. It is denoted as h.

* + 1. **Greedy Best-First Search (GBFS):**

Greedy best-first search' tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, ***f (n) = h(n).***

The algorithm is called "greedy"—at each step it tries to get as close to the goal as it can.

Greedy Best First search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.

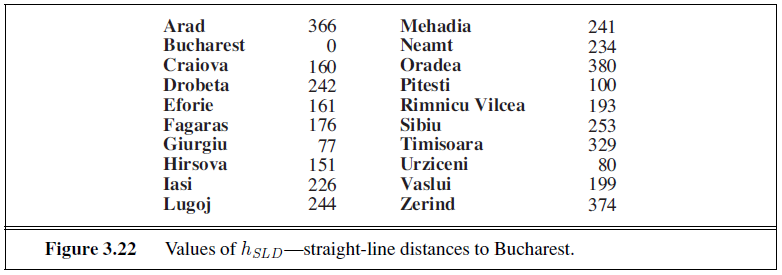
**Algorithm:**

1. Start with OPEN containing just the initial state
2. If the OPEN list is empty, Stop and return failure.
3. Remove the node ‘n’, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
4. Expand the node n, and generate the successors of node n.
5. Check each successor of node n, whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
6. For each successor node, evaluate function f(n)=h(n), and place into OPEN if it is not in either OPEN or CLOSED list.
7. Return to Step 2.

It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function (h) to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor.

**Eg.** Apply Greedy Best First Search algorithm to find path from Arad to Bucharest.

We use the **straight** **line distance** heuristic, which we will call hSLD. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure 3.22.



Explanation for exploring:

1. Initially, OPEN= [Sibiu, Timisoara and Zerind] and CLOSED= [Arad]

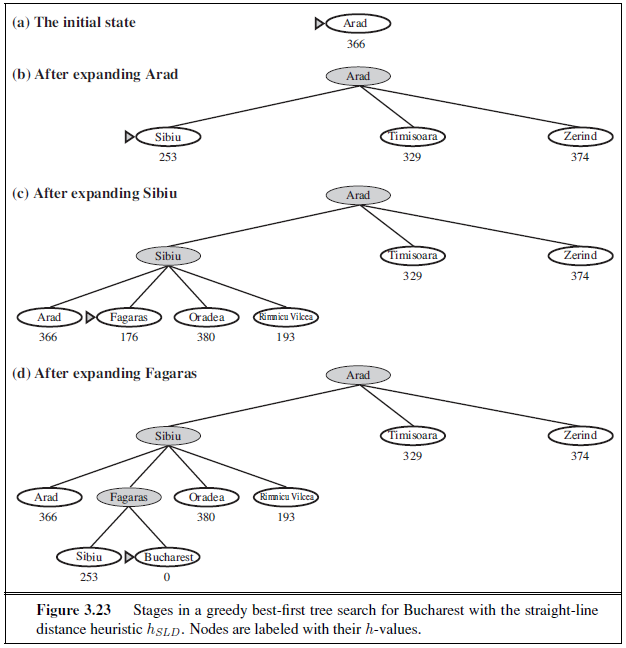
2. The first node to be expanded from Arad will be Sibiu because it is closer (hSLD = 253) to Bucharest than either Zerind or Timisoara.

OPEN = [Zerind, Timisoara, Fagaras, Rimncu Vilcea and Oradea] and CLOSED = [Arad, Sibiu]

3. The next node to be expanded will be Fagaras (hSLD = 176) because it is closest.

OPEN = [Zerind, Timisoara, Fagaras, Rimncu Vilcea, Oradea, Bucharest] and CLOSED = [Arad, Sibiu, Fagaras]

4. Fagaras in turn generates Bucharest (hSLD = 0), which is the goal.



**Advantages:**

1. Greedy Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
2. This algorithm is more efficient than BFS and DFS algorithms.

**Disadvantages:**

1. It can behave as an unguided depth-first search in the worst case scenario.
2. This algorithm is not optimal.

**Performance of Greed Best First Search:**

**Time Complexity:** The worst case time complexity of Greedy best first search is O(bm).

**Space Complexity:** The worst case space complexity of Greedy best first search is O(bm). Where, m is the maximum depth of the search space.

**Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.

**Optimal:** Greedy best first search algorithm is not optimal.