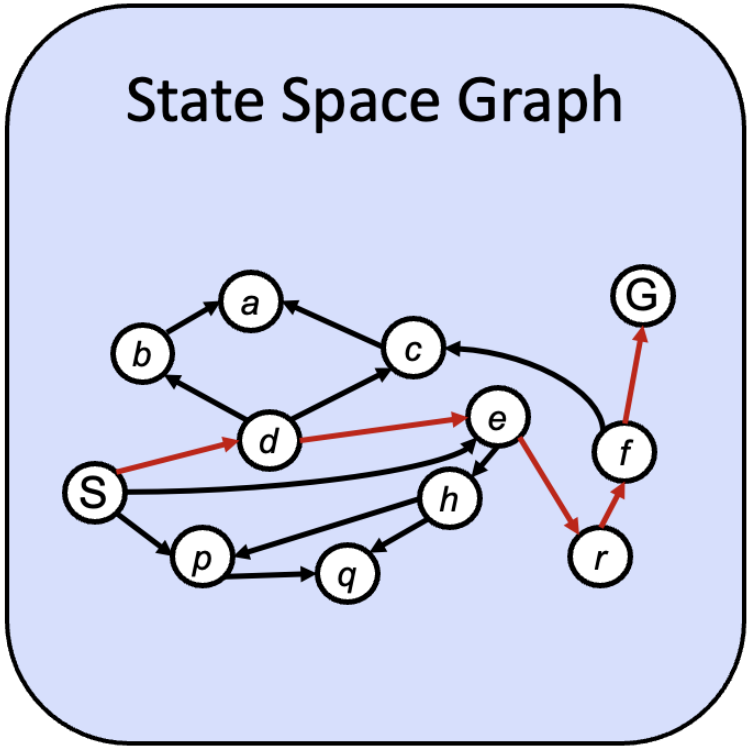
Artificial Intelligence
CS432

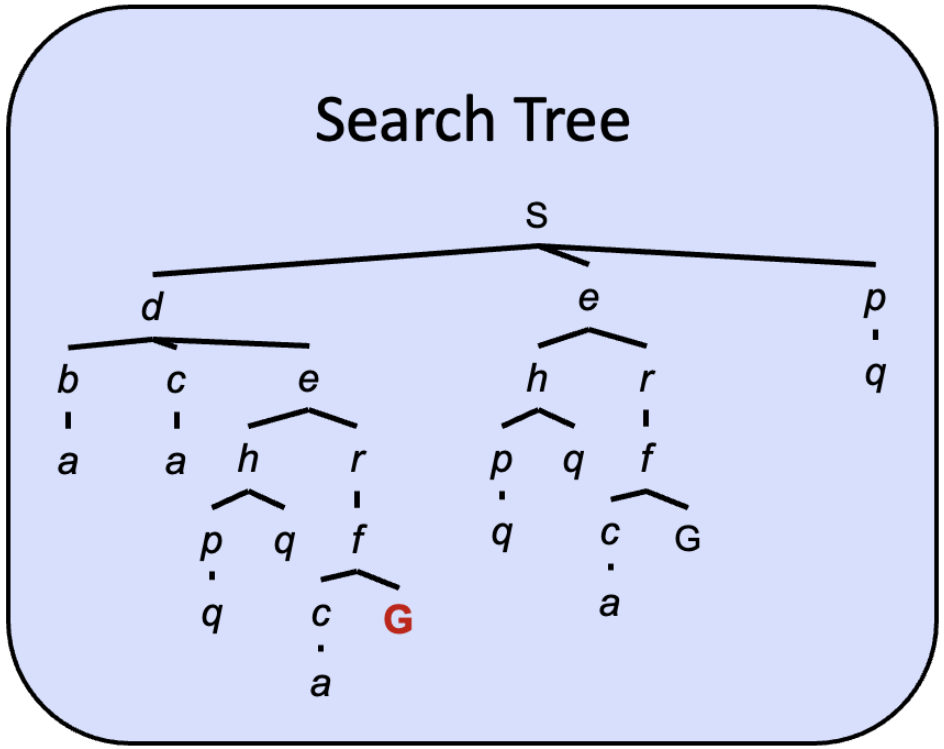
CHAPTER 3 – PART 2
Problem Solving and Search

State Serach Graph Vs Serach Tree



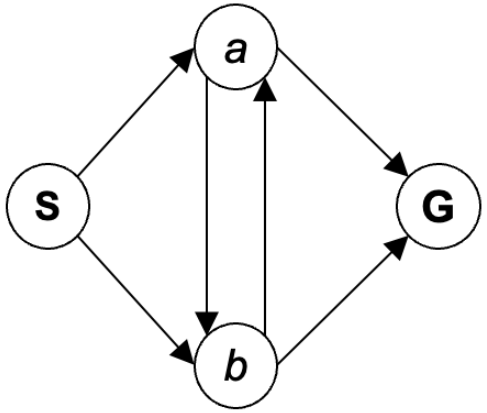
Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.



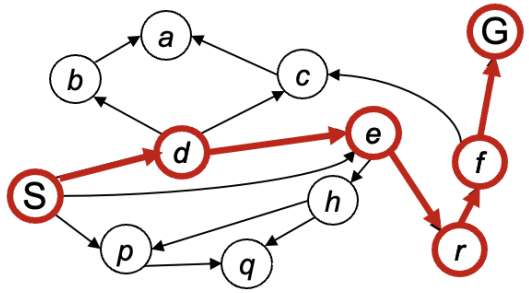
State Serach Graph Vs Serach Tree

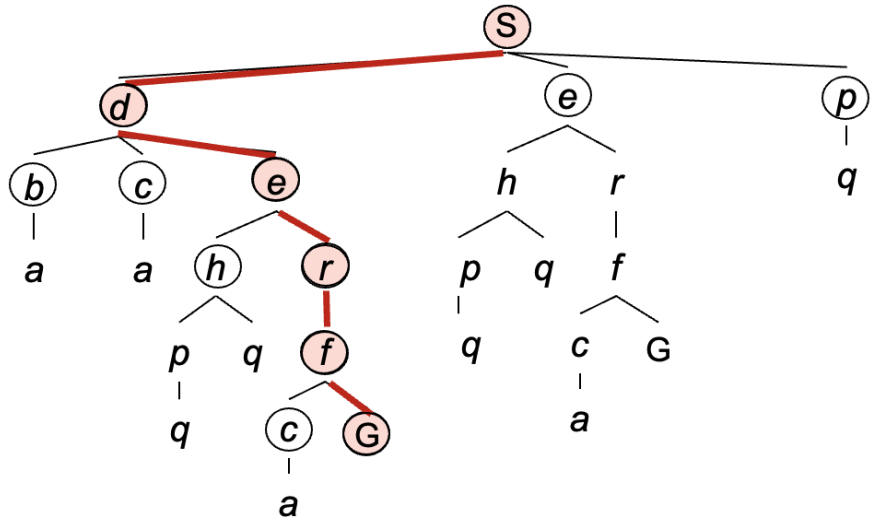
Consider this 4-state graph:



How big is its search tree (from S)?

Tree Serach



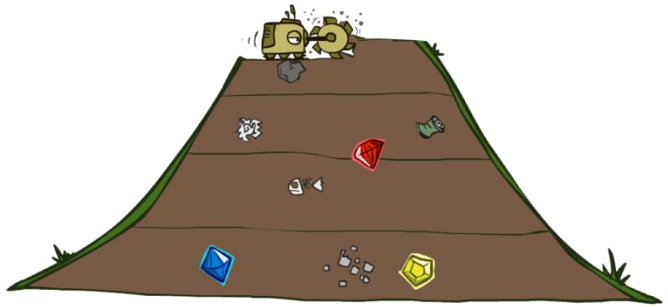


~~s~~
~~s → d~~
s → e
s → p
s → d → b
s → d → c
~~s → d → e~~
s → d → e → h
~~s → d → e → r~~
~~s → d → e → r → f~~
s → d → e → r → f → c
~~s → d → e → r → f → G~~

Uninformed Search Stratigies (Blind Search)

- The strategies have **no additional information** about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- **Examples:**
- **Breadth First Search BFS**
- **Uniform Cost Serach**
- **Depth First Search DFS**
- **Depth Limited Search DLS**
- **Iterative Deeping depth-first Search IDS**
- **Bidirectional Search**

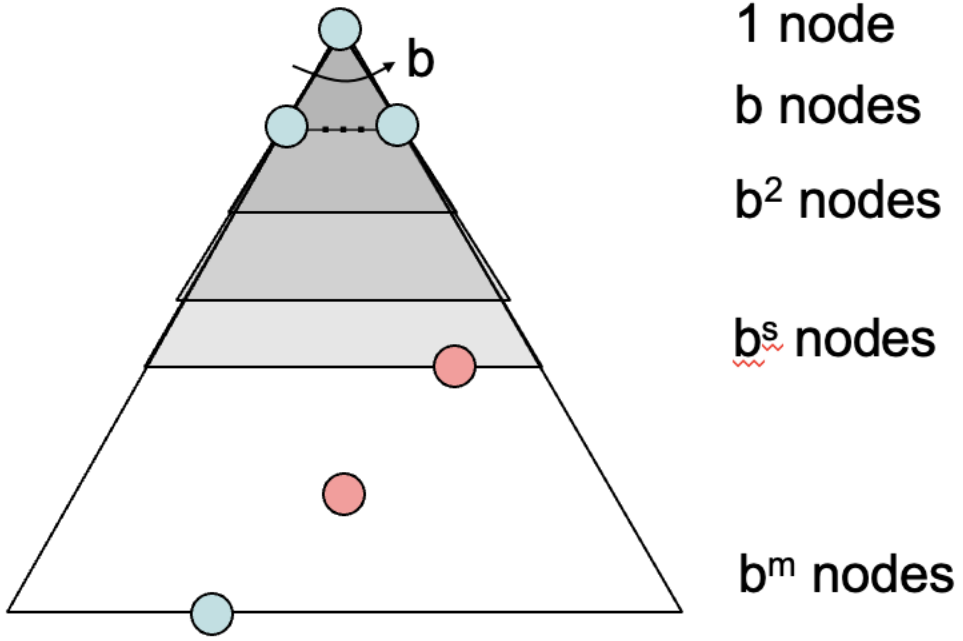
1. Breadth First Search



- Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- This is achieved very simply by using a **FIFO** queue for the frontier.
- Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- Thus, breadth-first search always has the shallowest path to every node on the frontier.

1. Breadth First Search

- **Complete??** Yes if b is finite.
- **Optimal??** if the path cost is a nondecreasing function of the depth of the node.
- **Time Complexity??** Given that b is the branching factor, The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.



1. Breadth First Search

- **Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level.**
- Then the total number of nodes generated is $b + b^2 + b^3 + \cdots + b^d = O(b^d)$.
- (If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $O(b^{d+1})$.)

- **space complexity??**
- For breadth-first graph search in particular, every node generated remains in memory.
- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier
- The space complexity is $O(b^d)$

- The memory requirements are a bigger problem for breadth-first search than is the execution time

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

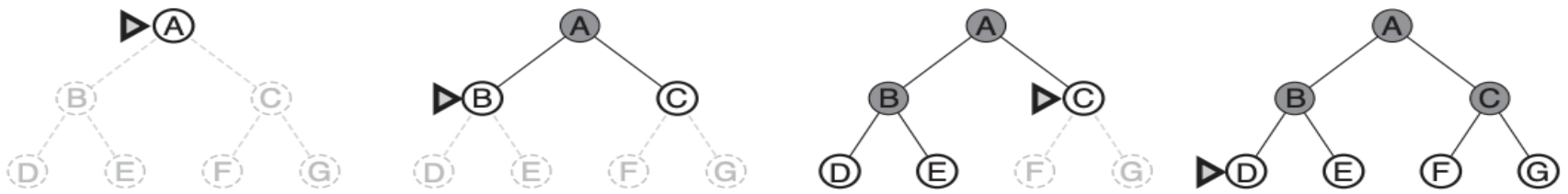


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

2. Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any step-cost function.
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost **g(n).**
- This is done by storing the frontier (fringe) as a priority queue ordered by g.
- There are two other significant differences from breadth-first search.
- The first is that the goal test is applied to a node when it is selected for expansion rather than when it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path.
- The second difference is that a test is added in case a better path is found to a node currently on the frontier

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

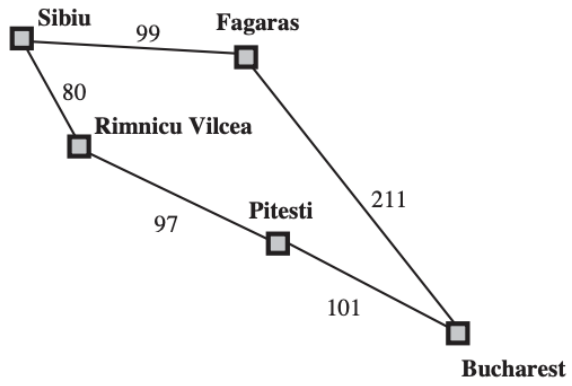


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

2. Uniform-cost search

- **Optimal??** Yes, nodes expanded in increasing order of g(n)
- Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost
- **Complete??** Yes if the cost of every step exceeds some small positive constant ε .
- let $C*$ be the cost of the optimal solution, and assume that every action costs at least ε
- **Time??** # of nodes with g ≤ cost of optimal solution, $O(b^{\lceil C*/\varepsilon \rceil+1})$
  where $C*$ is the cost of the optimal solution
- **Space??** # of nodes with g ≤ cost of optimal solution, $O(b^{\lceil C*/\varepsilon \rceil+1})$

- When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost;

3. Depth-first search

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- The depth-first search algorithm uses a **LIFO** queue.
- A **LIFO** queue means that the most recently generated node is chosen for expansion.

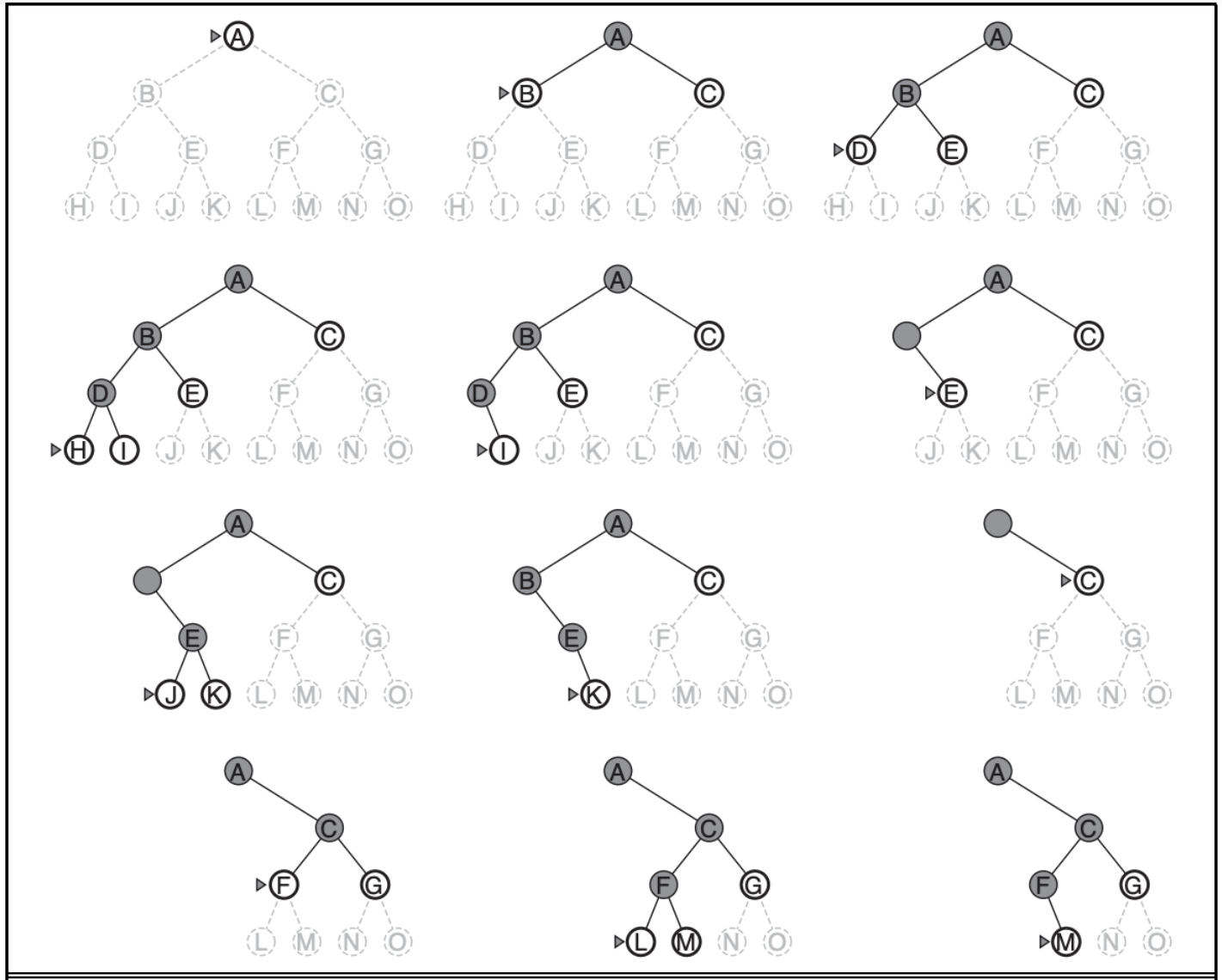


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

3. Depth-first search

- **Complete??** No: fails in infinite-depth spaces, spaces with loops
- complete in finite spaces.
- **Time??** $O(b^m)$: terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first
- **Space??** O(bm), i.e., linear space!
- **Optimal??** No

3. Depth-first search

- A variant of depth-first search called backtracking search uses still less memory.
- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next.
- In this way, only **O(m)** memory is needed rather than O(bm).

4. Depth-limited Search

- The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit *l* .
- Nodes at depth *l* are treated as if they have no successors.
- The depth limit solves the infinite-path problem.
- Unfortunately, it also introduces an additional source of **incompleteness** if we choose *l* < d.
- Depth-limited search will also be **nonoptimal** if we choose *l* >d
- **Its time complexity is O(b^l) and its space complexity is O(b *l*).**
- Depth-first search can be viewed as a special case of depth-limited search with *l* =∞.
- depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

5. Iterative deepening depth-first search

- 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.
- Iterative deepening combines the benefits of depth-first and breadth-first search.
- Like depth-first search, its memory requirements are modest: **O(bd)**.
- Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.

5. Iterative deepening depth-first search

- Iterative deepening search may seem wasteful because states are generated multiple times.
- The total number of nodes generated in the worst case is

$$N(IDS)=(d)b + (d-1)b2 + \cdots + (1)b^d \text{ , which gives a time complexity } O(b^d)$$

- There is some extra cost for generating the upper levels multiple times, but it is not large. For example, if b = 10 and d = 5, the numbers are

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450\ N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 .$$

- Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

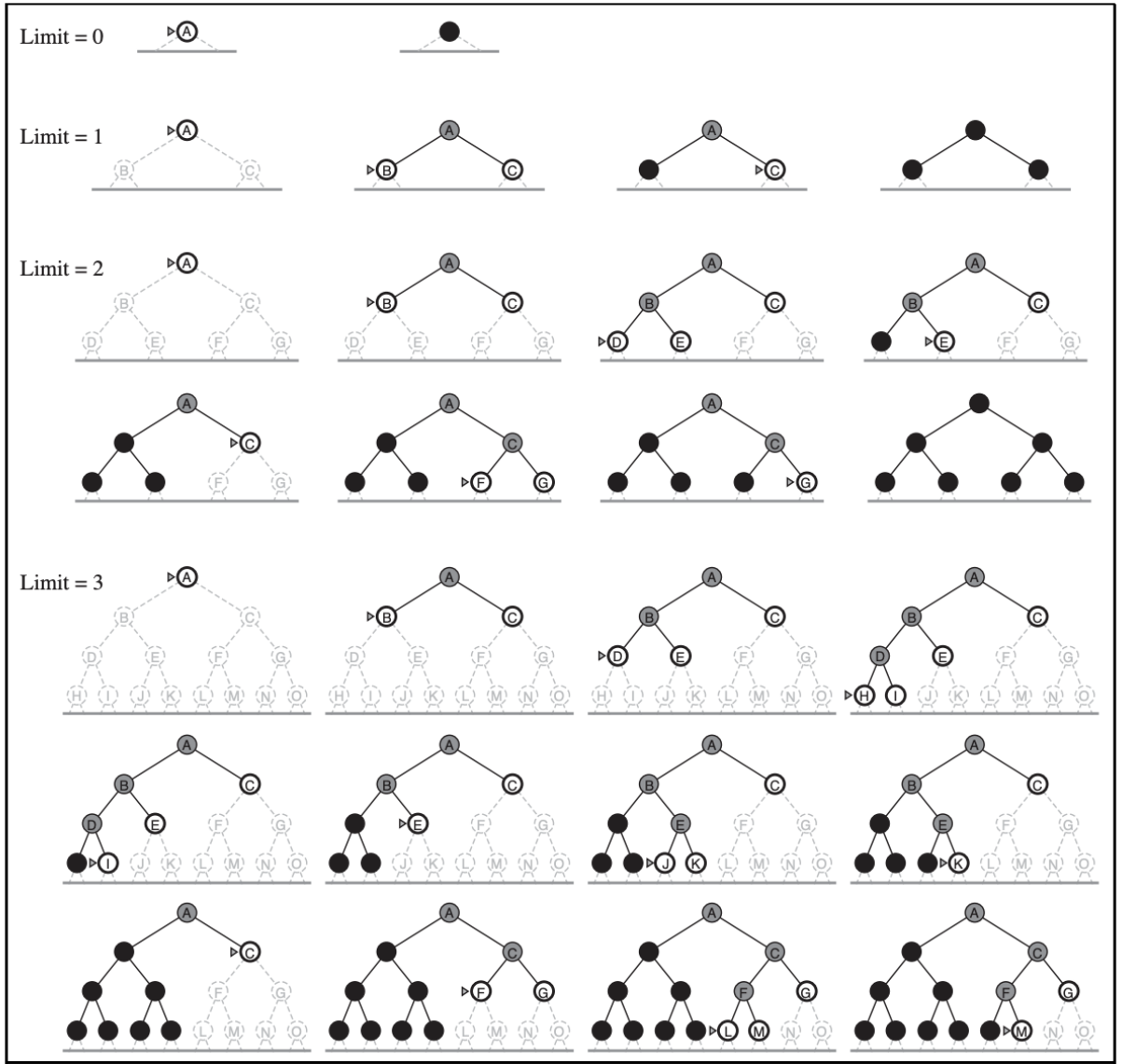


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

6. 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.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than bd, or the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.
- The time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$. The space complexity is also $O(b^{d/2})$.
- This space requirement is the most significant weakness of bidirectional search.

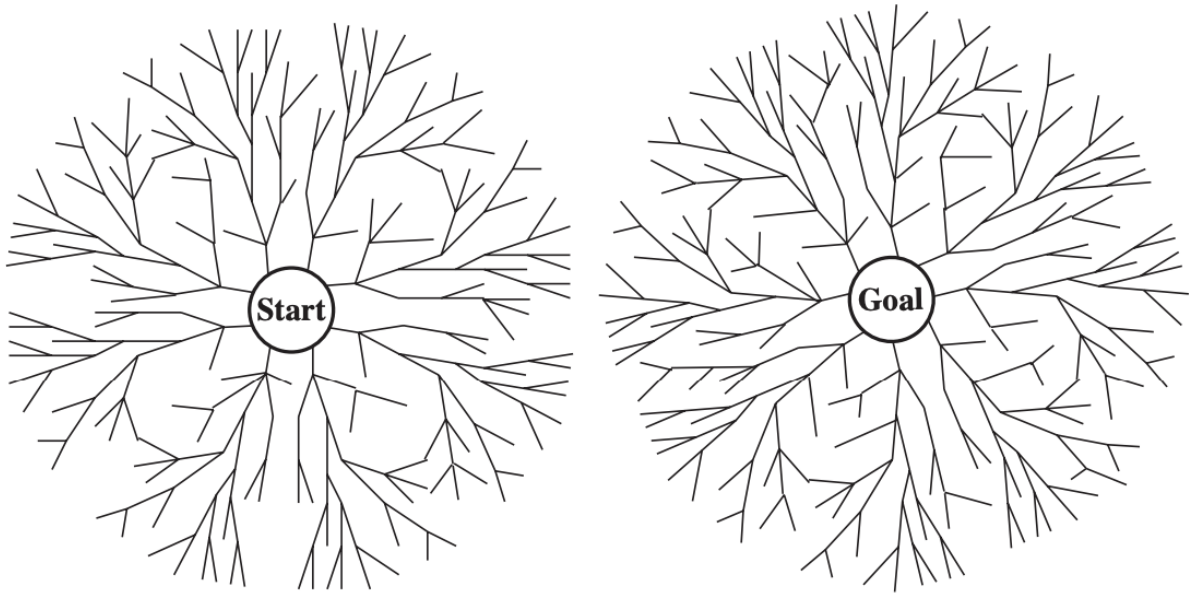


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.