Chapter 3:
Solving Problems by Searching
Uninformed Search

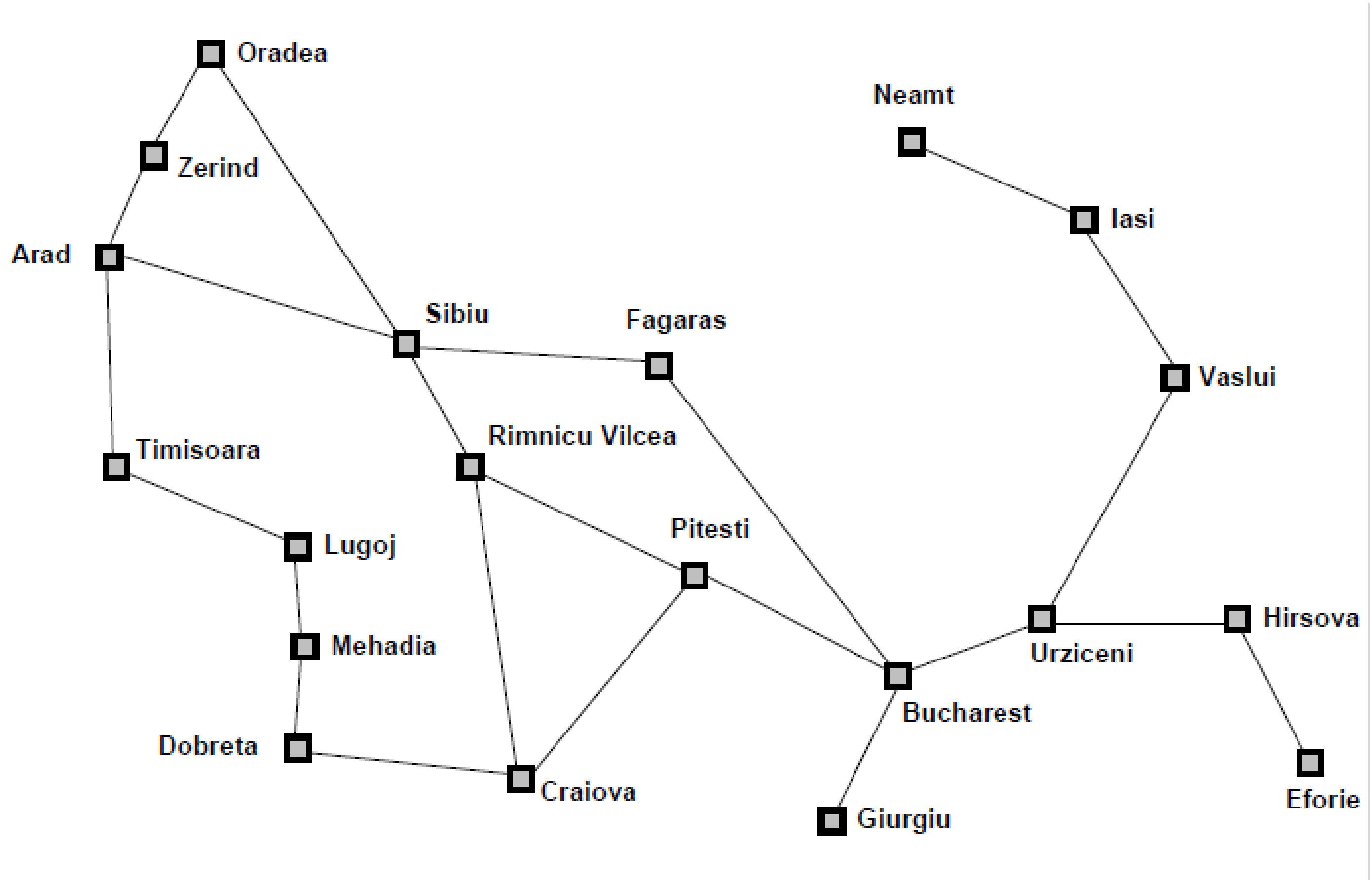
Edited by Neng-Fa Zhou

- *How does an agent can find a sequence of actions that achieves its goals with minimum cost?*
- This is the general task of **problem solving** and is typically performed by **searching** through an internally modeled space of world states.

- States
- The initial state
- ACTIONS(s): the set of actions that can be executed in state s
- The transition model: s’ = RESULT(s,a)
- The goal test
- A path cost function

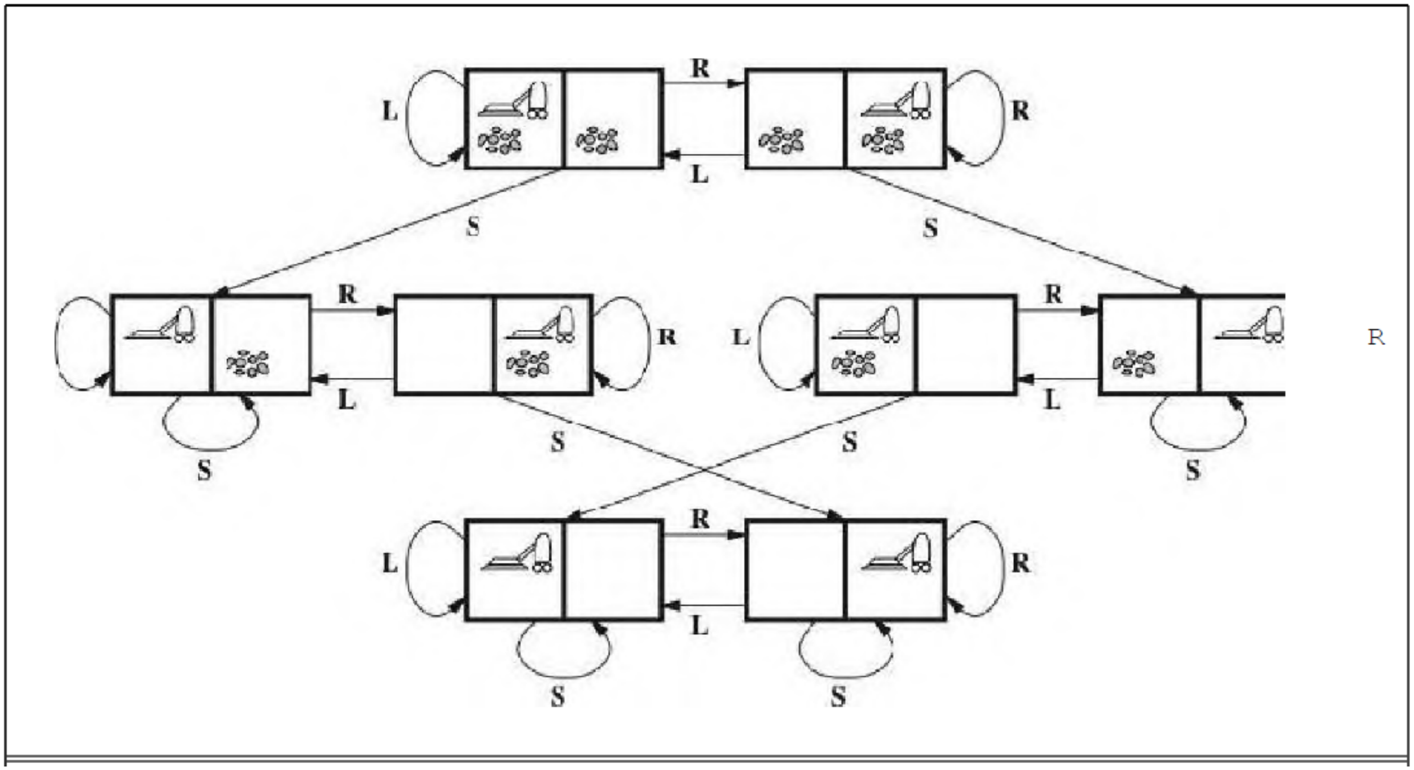
- Given:
  - An **initial state** of the world
  - A set of possible possible actions or **operators** that can be performed.
  - A **goal test** that can be applied to a single state of the world to determine if it is a goal state.
- Find:
  - A **solution** stated as a **path** of states and operators that shows how to transform the initial state into one that satisfies the goal test.
- State space
  - The initial state and set of operators implicitly

- **Step Cost**
  - Each individual action has an associated cost.
- **Path cost**
  - A function that assigns a cost to a path, typically by summing the cost of the individual operators in the path.
- **Optimal Solution**
  - Find a lowest-cost path.



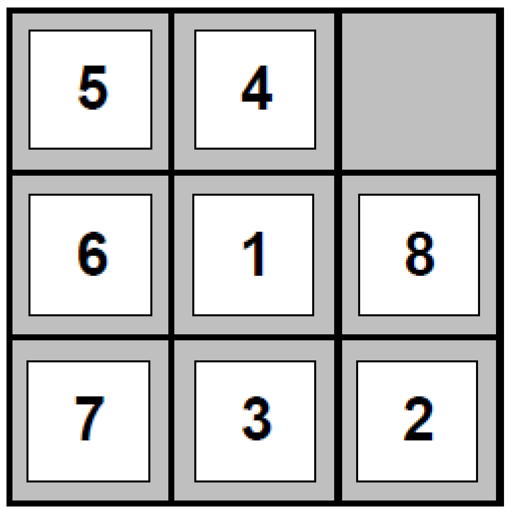
- Find a route from Arad to Bucharest

Example Problems: The Vacuum World

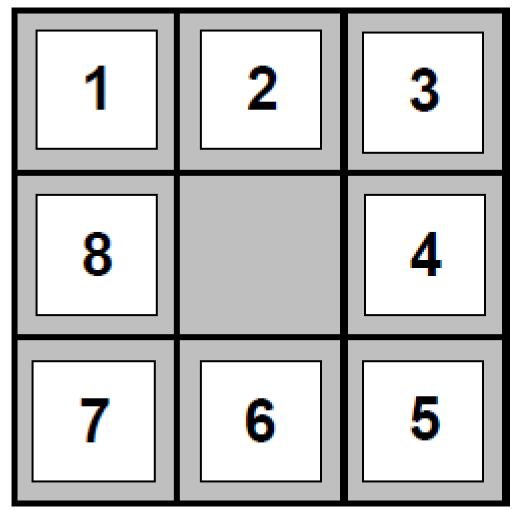


- States: An environment with n locations has nx2^n states.
- Initial state, actions, transition model, goal test, path cost

Example Problems: 8-Puzzle

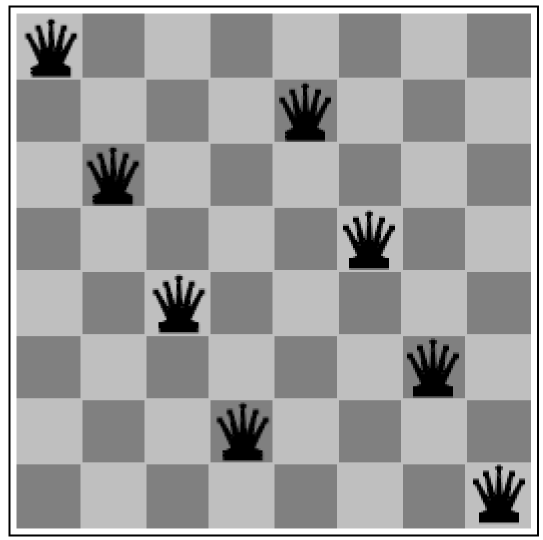


Start State



Goal State

Example Problems: 8-Queens Problem



- Knuth's Conjecture
  - Starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.
- Cryptarithmetic
  - SEND + MORE = MONEY
- Water Jugs Problem
- Missionaries and Cannibals Problem

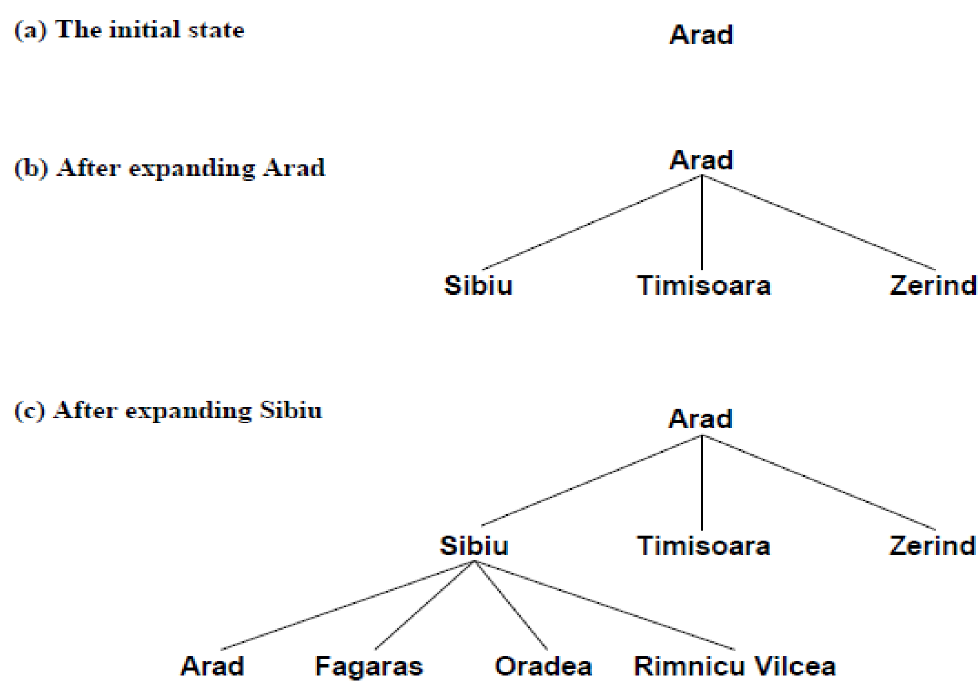
Real-World Problems

- Route finding
- Travelling salesman problem
- VLSI layout
- Robot navigation
- Automatic assembly sequencing
- Protein design

- A state can be **expanded** by generating all states that can be reached by applying a legal operator to the state.
- State space can also be defined by a **successor function** that returns all states produced by applying a single legal operator.
- A **search tree** is generated by generating search nodes by successively expanding states starting from the initial state as the root.

- A **search node** in the tree can contain
  - n.STATE: Corresponding state
  - n.PARENT: Parent node
  - n.ACTION: Operator applied to reach this node
  - n.PATH-COST: g(n), path cost of path from initial state to node

Expanding Nodes and Search



- Search algorithms all share this basic tructure; they vary in how they choose which state to expand next—the so-called **search strategy.**

Tree-Search

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

Graph-Search

```
function GRAPH-SEARCH(problem) returns a solution. or failure
  initialize the frontier using the initial stale of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

Evaluate Search Algorithm's Performance

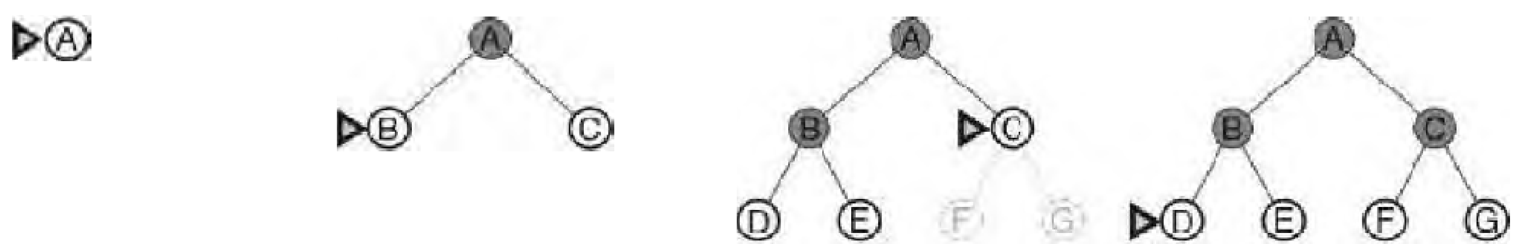
- Completeness (systematic search vs local search)
- Time Complexity
- Space Complexity
- Optimality

- **Blind, exhaustive, brute force,** do not guide the search with any additional information about the problem
  - Breadth-first search
  - Uniform Cost Search
  - Depth-First Search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search

- **Heuristic, intelligent,** use information about the problem (estimated distance from a state to the goal) to guide the search.
  - Greedy best-first search
  - A* search
  - Iterative deepening A* (IDA*)

Breadth-first search

```
Inaction 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)
```

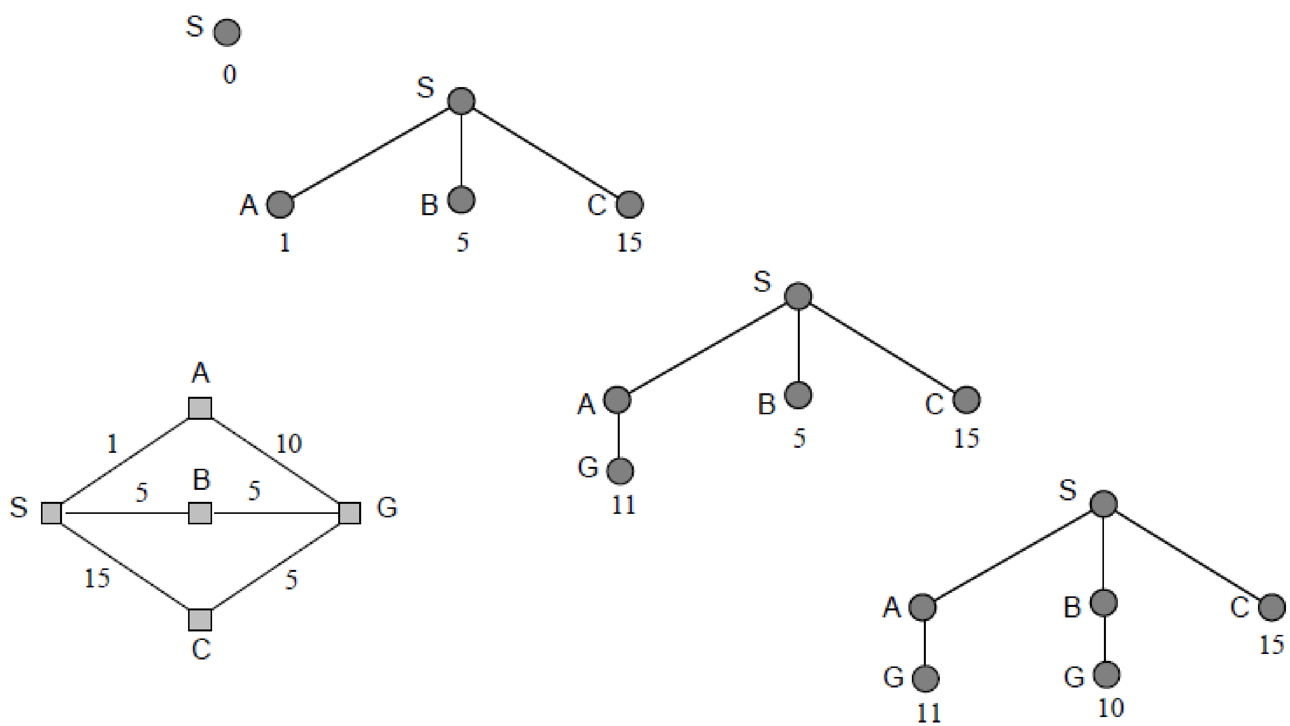


- Assume there are an average of *b* successors to each node, called the **branching factor**.
- Therefore, to find a solution path of length *d* must explore $1 + b + b^2 + b^3 + \ldots + b^d$ nodes.
- Plus need *b^d* nodes in memory to store leaves in queue.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10[illegible]	1.1 seconds	1 gigabyte
8	10[illegible]	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10[illegible]	350 years	10 exabytes

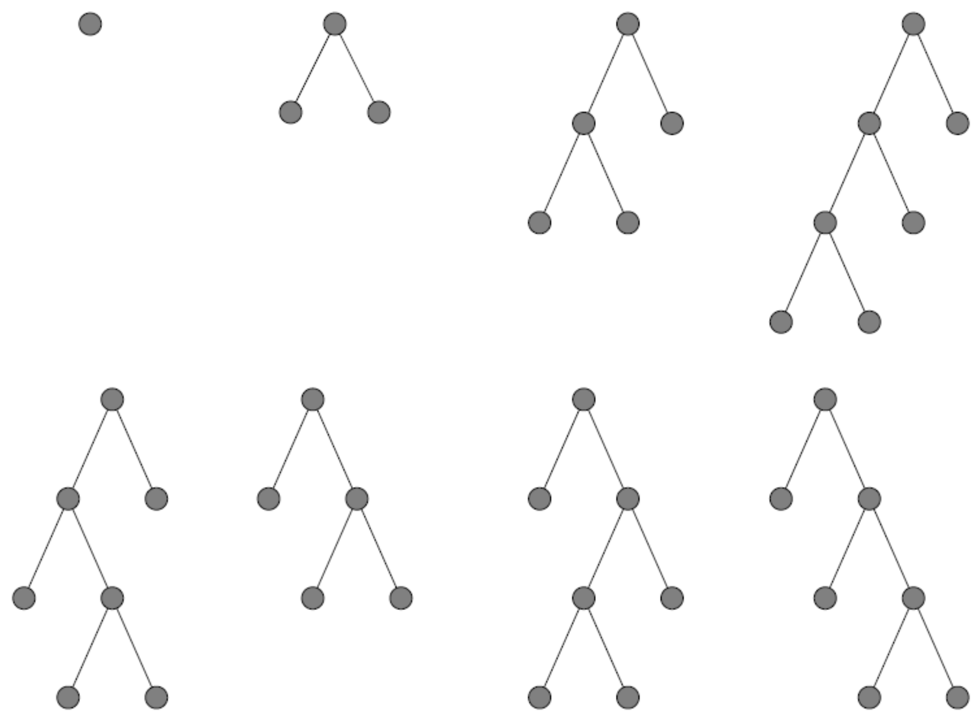
Uniform Cost Search

- Like breadth-first except expands the node with lowest path cost, g(n).



(a) (b)

- DFS expands the deepest unexpanded node first. Implemented using a stack (LIFO).



DFS

```
Inaction 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(n ode. 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)
```

BREADTH-FIRST-SEARCH -> DEPTH-FIRST-SEARCH

FIFO -> LIFO

Depth-First Properties

- Not guaranteed to be complete
- Not guaranteed optimal
- Time complexity in worst case is still O(b^d)
- Space complexity is only O(bm) where m is maximum depth of the tree.

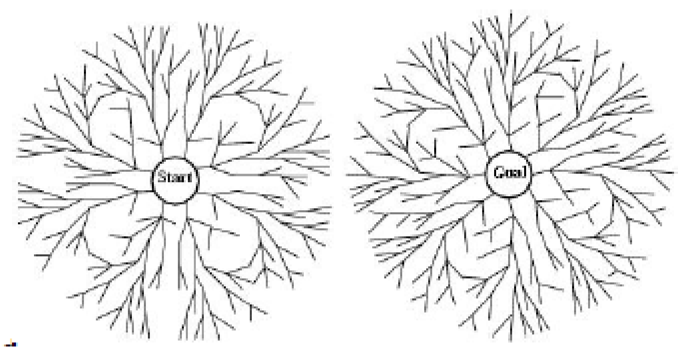
- Calls depth-first search with increasing depth limits until a goal is found.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
   end
   return failure
```

- **completeness** (when the branching factor is finite)
- **optimal** (when the path cost is a non-decreasing function of the depth of the node)
- low **memory** consumption **O(bd)**
- What about **time complexity**?
  - $d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d = \mathbf{O(b^d)}$

- We can 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).



- Rational?

$$b^{d/2} + b^{d/2} << b^{d}$$