ARTIFICIAL INTELLIGENCE

Russell & Norvig

Chapter 3: Solving Problems by Searching, part 2

Problem definition components

1. Initial State
   - For example, In(Arad)
2. Possible Actions
   - For state s, Action(s) returns actions that can be executed in s
   - Actions(In(Arad)) = {Go(Sibiu), Go(Timisoara), Go(Zerind)}
3. Transition Model
   - Successor function, like delta (δ) transitions in finite state machines
   - Together, initial state, actions and transition model define the **state space**
4. Goal Test
   - Similar to “final state”, e.g. {In(Bucharest)}, or abstract property (checkmate)
5. Path Cost
   - Agent’s cost function used as internal performance measure. Usually sum of cost of actions along path from initial state to goal state

Graph Search

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state 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

Search Strategies

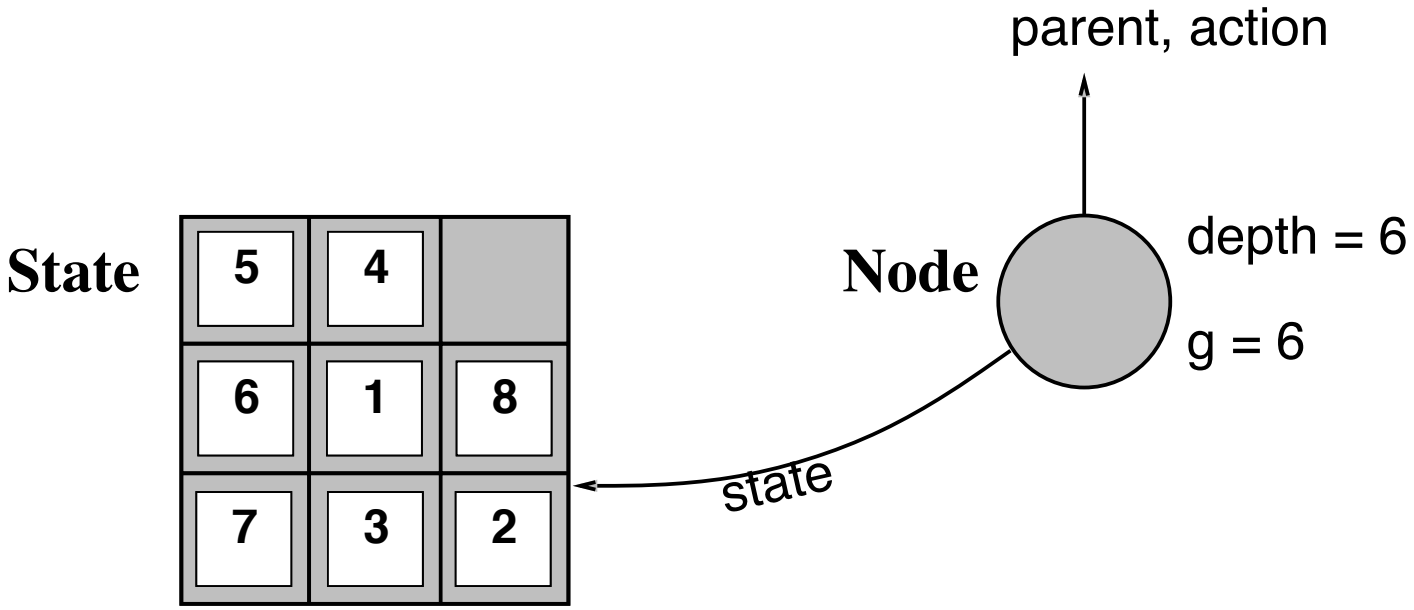
- A search strategy is defined by picking the order of node expansion

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - optimality: does it always find a least-cost (optimal) solution?
  - time complexity: number of nodes generated/expanded
  - space complexity: maximum number of nodes in memory

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

Nodes and States

n.state: state associated with node n
n.parent: node in search tree that generated this node
n.action: action that was applied to parent to generate this node
n.path-cost: cost of path from initial state to this node, denoted by *g(n)*



Informed vs. Uninformed Searches

- Uninformed (or blind) strategies do not exploit any of the information contained in a state
  - Breadth-first search (BFS)
  - Uniform cost search
  - Depth-first search (DFS)
  - Depth-limited search
  - Iterative-deepening search (IDS)
  - Bidirectional search
- Informed (or heuristic) strategies exploit such information to assess that one node is “more promising” than another

Breadth-first search (BFS)

- *Shallowest* unexpanded node is chosen for expansion
- Store frontier of nodes in FIFO queue
- Check if goal when *generated*, since placed on queue and taken off of queue in same order
- Check to avoid repeated states

- Criteria (b is branching factor; d is depth of goal):
  - Complete? Yes (if some goal at finite depth *d, and b is finite*)
  - Space? Not great, size of frontier, so $O(b^d)$ potentially
  - Time? Nodes generated, $b + b^2 + b^3 + \ldots + b^d = O(b^d)$
  - Optimal? Yes, if all actions have same cost

- Space is normally more of a problem with BFS than time

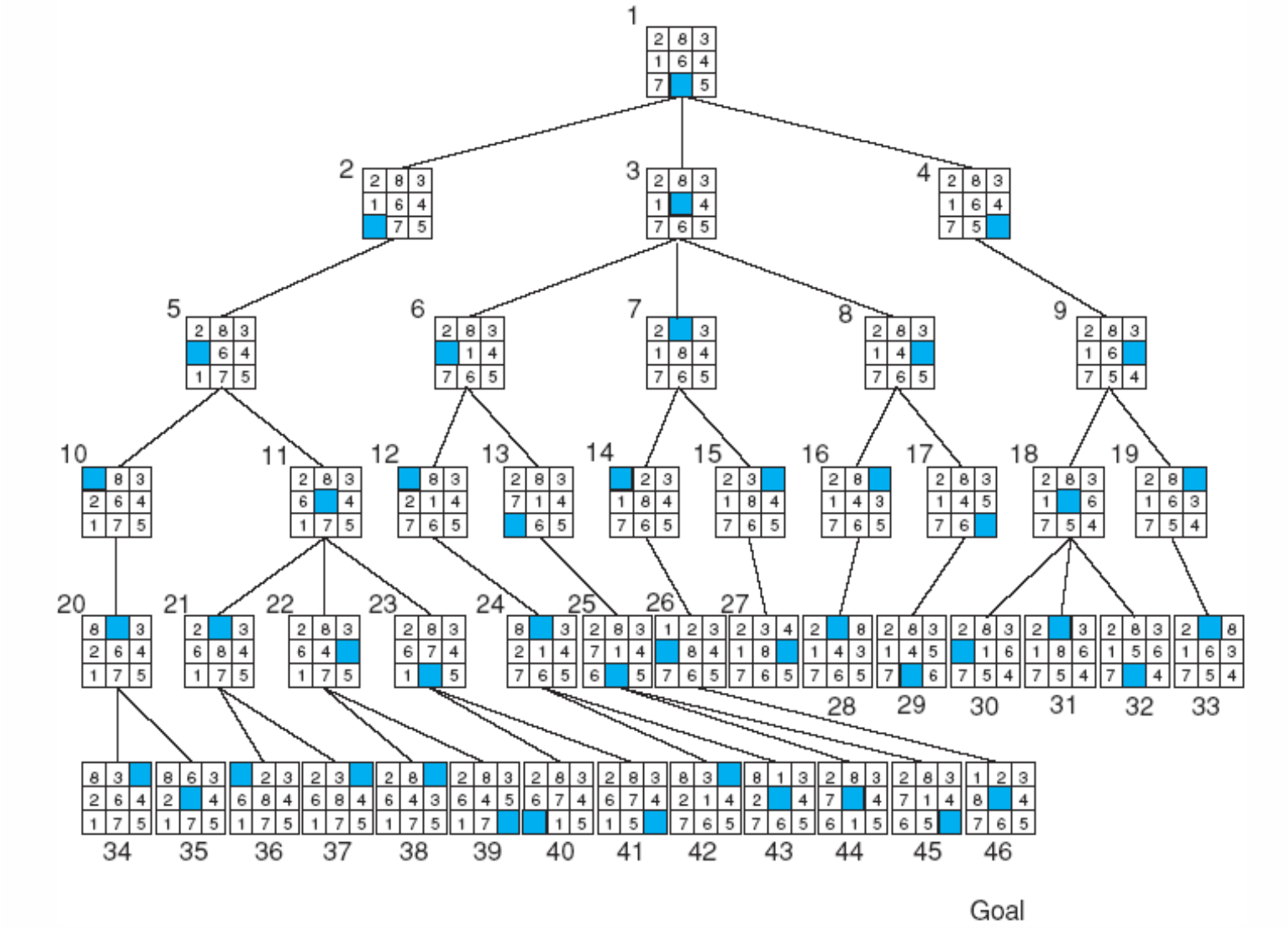
Pseudocode for BFS

```
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-□rst search on a graph.

BFS tree for 8-puzzle



Uniform-cost search

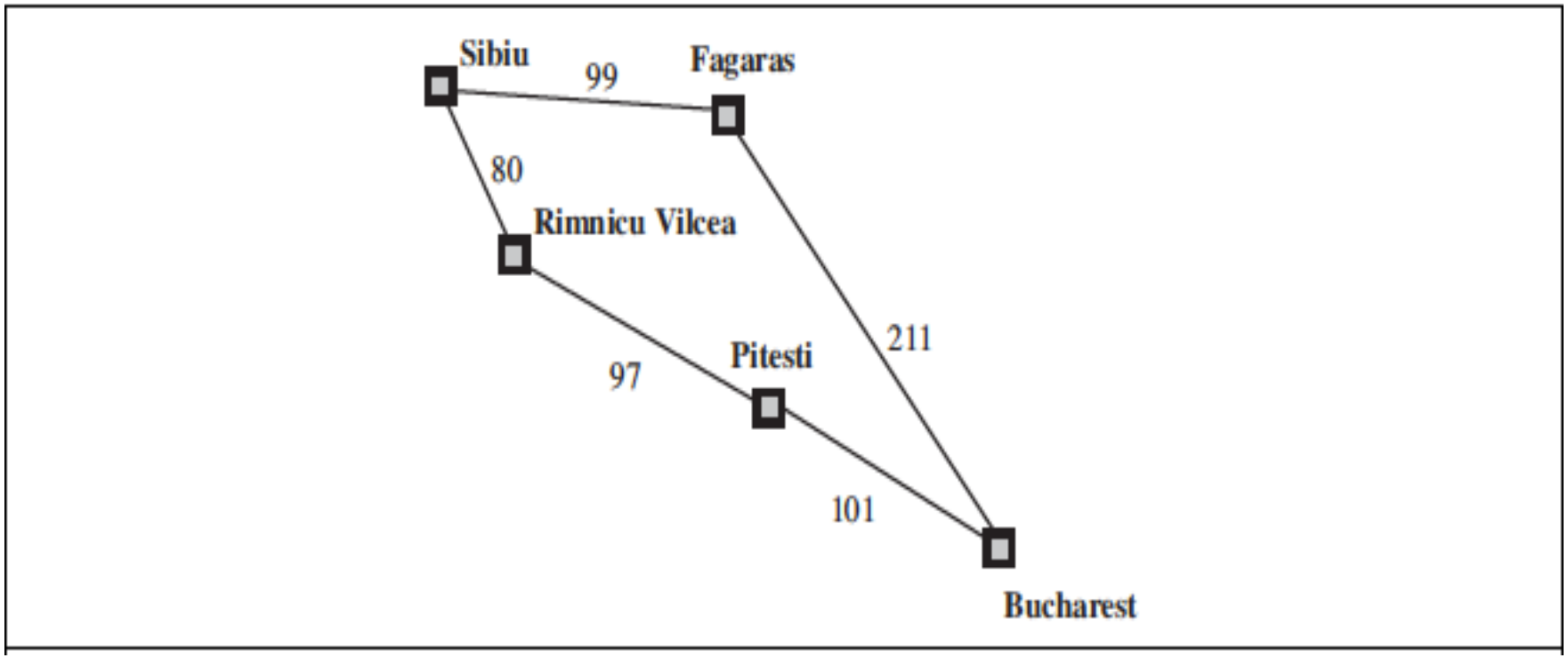
- What about when actions have varying costs?
- For each node *n*, keep track of the "path cost", *g(n)*
- Maintain frontier as a priority queue

- **Uniform-cost search expands the node *n* with the lowest path cost**

- Other differences from BFS:
  - Must check for goal when node chosen for expansion (instead of when generated)
  - Must also check for each state generated that is in frontier, whether this new path has lower path cost

Uniform-cost search example

- Trace with this part of the Romania example



UCS Pseudocode

```
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
```

Uniform cost analysis

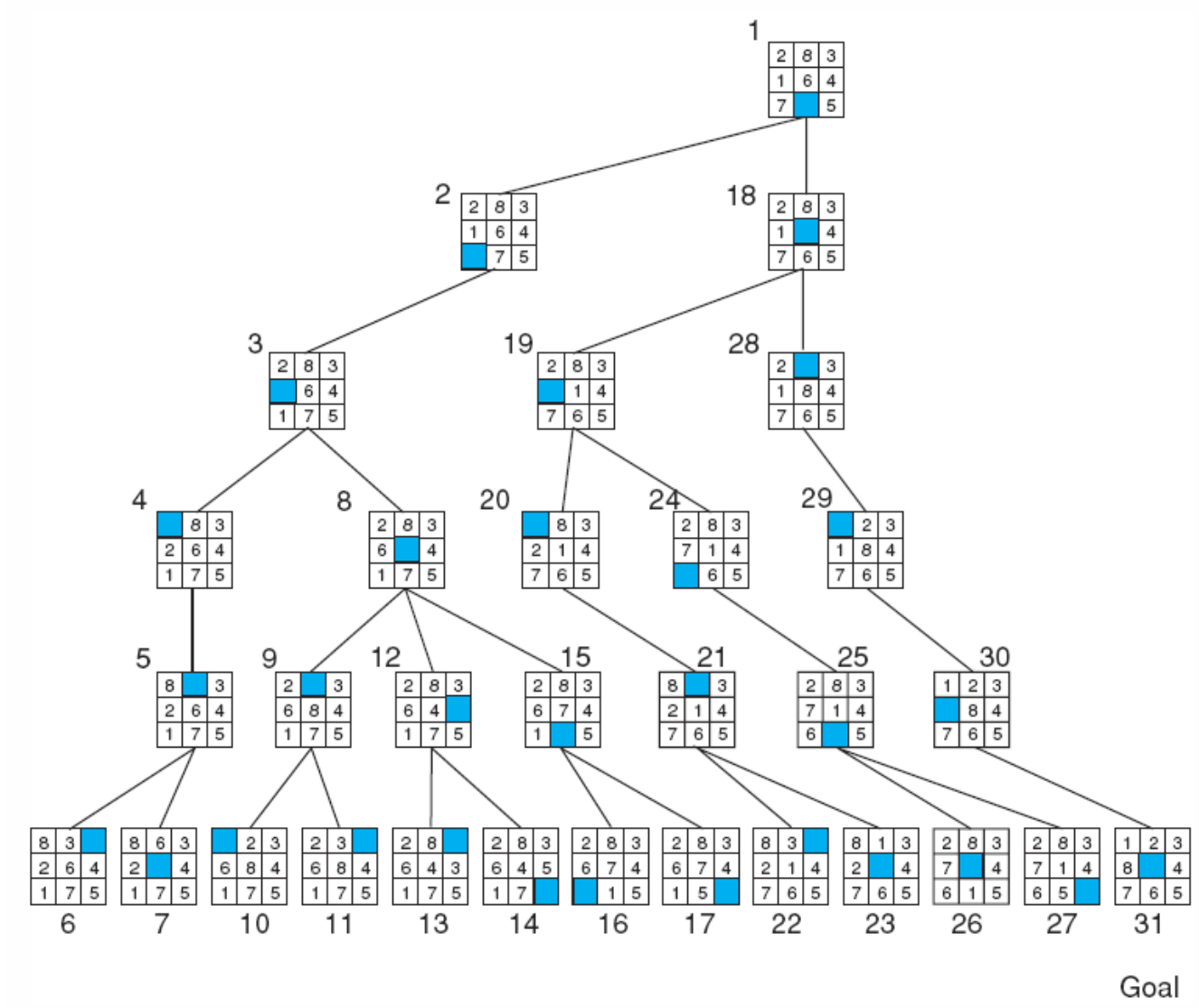
- Assume all actions have positive (non-zero) cost, at least ε

- **Optimal?** Yes, UCS expands nodes in order of optimal path cost
- **Complete?** Yes
- **Time and space** are harder to characterize
- Assume C* is cost of optimal solution, then time and space in worst case is $O(b^{1+\text{floor}(C^*/\varepsilon)})$, which can be worse than $O(b^d)$.

Depth-first search

- Always expand the *deepest* node in the current frontier
- Uses a LIFO queue (aka stack)
- Commonly implemented with recursion
- Criteria
  - Complete? No: fails in infinite-depth spaces with loops, but is complete in finite spaces (when avoiding repeated states)
  - Optimal? No.
  - Time? $O(b^m)$, where *m* is maximum depth of any node. Bad if *m* is much larger than *d*
  - Space (only good thing!): Need only store path from root of search tree and siblings of those nodes, so O(bm)

DFS tree for 8-puzzle



Depth-limited search

- Consider DFS with depth limit l
  - Nodes at depth l are treated as if they have no successors
  - Solves the infinite-path problem
  - If $l < d$ then incomplete
  - If $l > d$ then not optimal

- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

Iterative deepening search

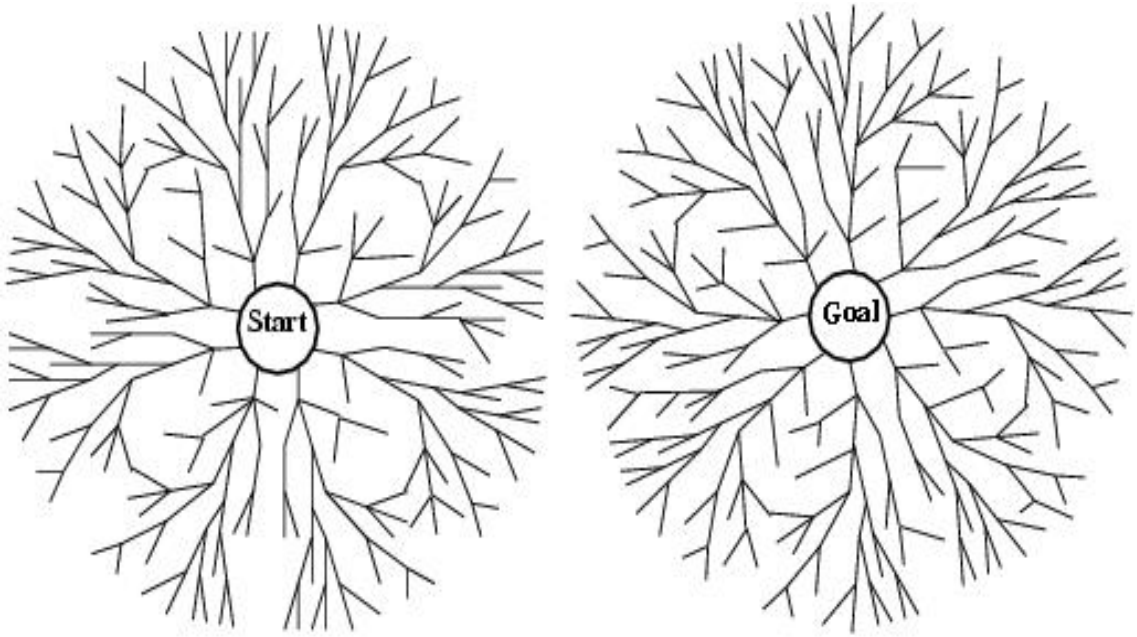
- Best of both BFS and DFS

- BFS is complete but has bad memory usage; DFS has nice memory behavior but doesn't guarantee completeness

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Bidirectional search

- Two simultaneous searches from start an goal.
  - Motivation: $b^{d/2} + b^{d/2} \neq b^{d}$
- Check whether the node belongs to other fringe before expansion.
- Space complexity is the most significant weakness.
- Complete and optimal if both searches are breadth-first.



Comparison of uninformed searches

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes