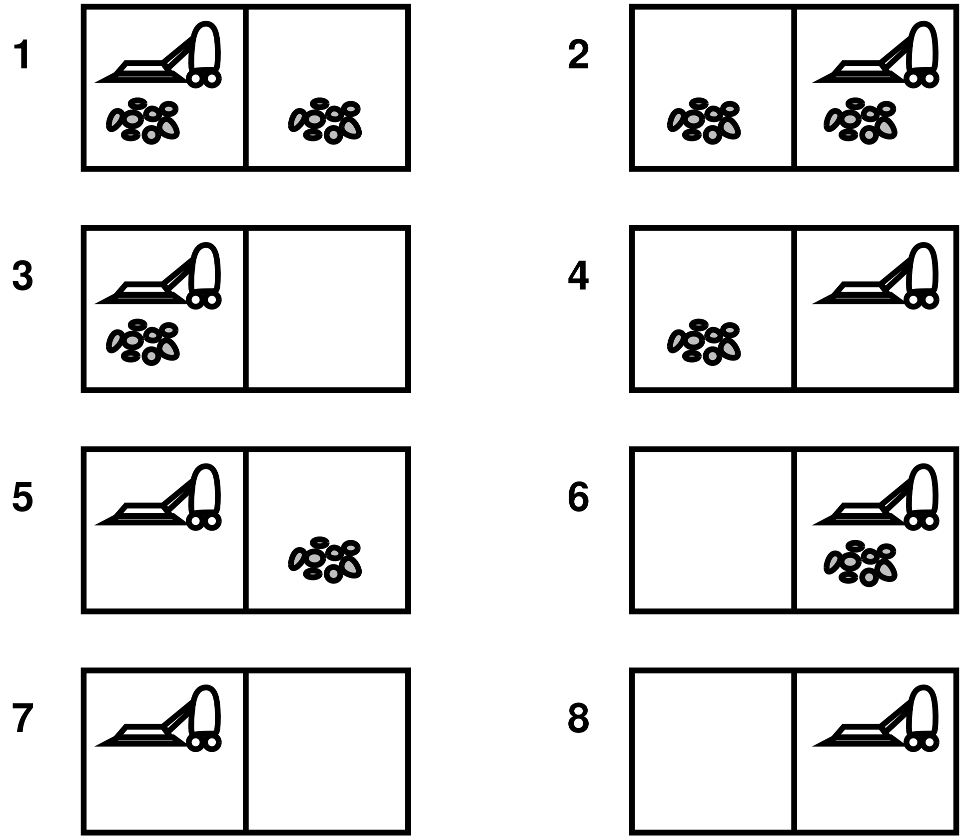
Simplified Vacuum Cleaner Problem

- **World States:** 1, 2, 3, 4, 5, 6, 7, 8
- **Actions:** left, right, suck (deterministic, perfect)
- **Sensors:** give complete information about world state
- **Performance Criterion:** clean up all dirt (world should be in states 7 and 8)



Defining a Search Problem

A Search Problem is a Computational Problem (inside the agent)

- **State Space:** Space of mental states that the agent will consider.
- **Initial State:** The starting mental state
- **Operators:** Simulated actions that take the agent from one mental state to another
- **Goal Test:** A predicate that tests the mental state to see if it is a goal state.

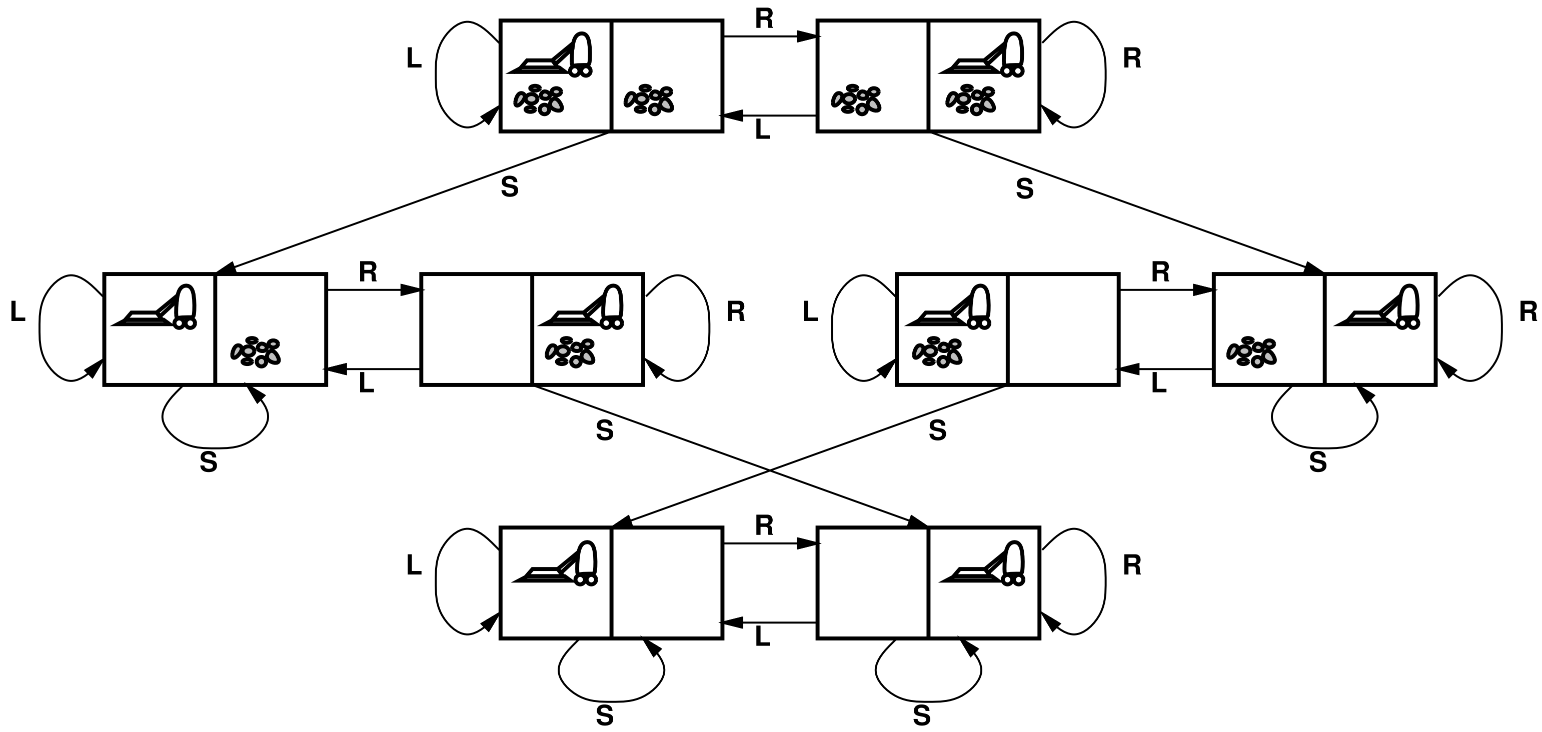
For our simple vacuum cleaner world, there is a direct correspondence:

- **State Space:** the 8 possible world states
- **Operators:** the 3 world actions
- **Initial state:** whatever state the sensors indicate
- **Goal test:** is mental state either 7 or 8?

Mental State Graph

The problem space can be visualized as a graph

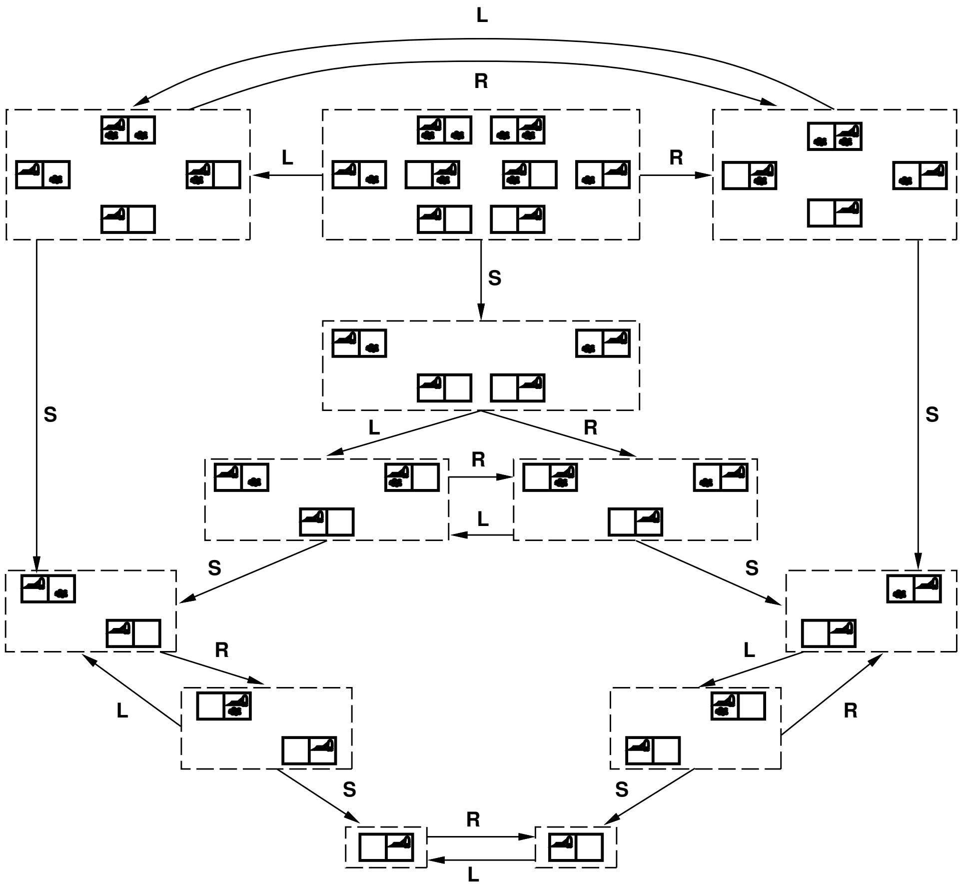
- **Nodes:** mental states
- **Arcs:** operators



Vacuum Environment #2

Same as above, but robot has NO SENSORS

- **Mental States:** Powerset of world states
- **Operators:** Add or remove possible world states
- **Goal test:** Mental state is either {7} or {8}.



Vacuum Environment #3: Murphy's Law

Same as #2, but SUCK action works as follows:

> If carpet is clean and agent SUCKs, then carpet will be dirty
>
> If carpet is dirty and agent SUCKs, then carpet will be clean

This is also a directed graph, but it is disconnected.

There is no path from the starting mental state to the goal mental state.

Vacuum Environment #4: Murphy's Law + Dirt Sensor

Sensor can detect dirt in current square, but it does not tell us which square we are located in.

New Problem Space:

- **Mental States:** same as before.
- **Operators:** New operator: SENSE. This operator can have two different outcomes ("dirt" or "no dirt").
- **Goal test:** same as before.

The graph now becomes a kind of hypergraph, because SENSE can lead to two different resulting states. A solution is a TREE that branches at each SENSE action and where each leaf of the tree is a goal mental state.

LEFT
SENSE
if dirt SUCK
RIGHT
SENSE
if dirt SUCK

The 8-Puzzle

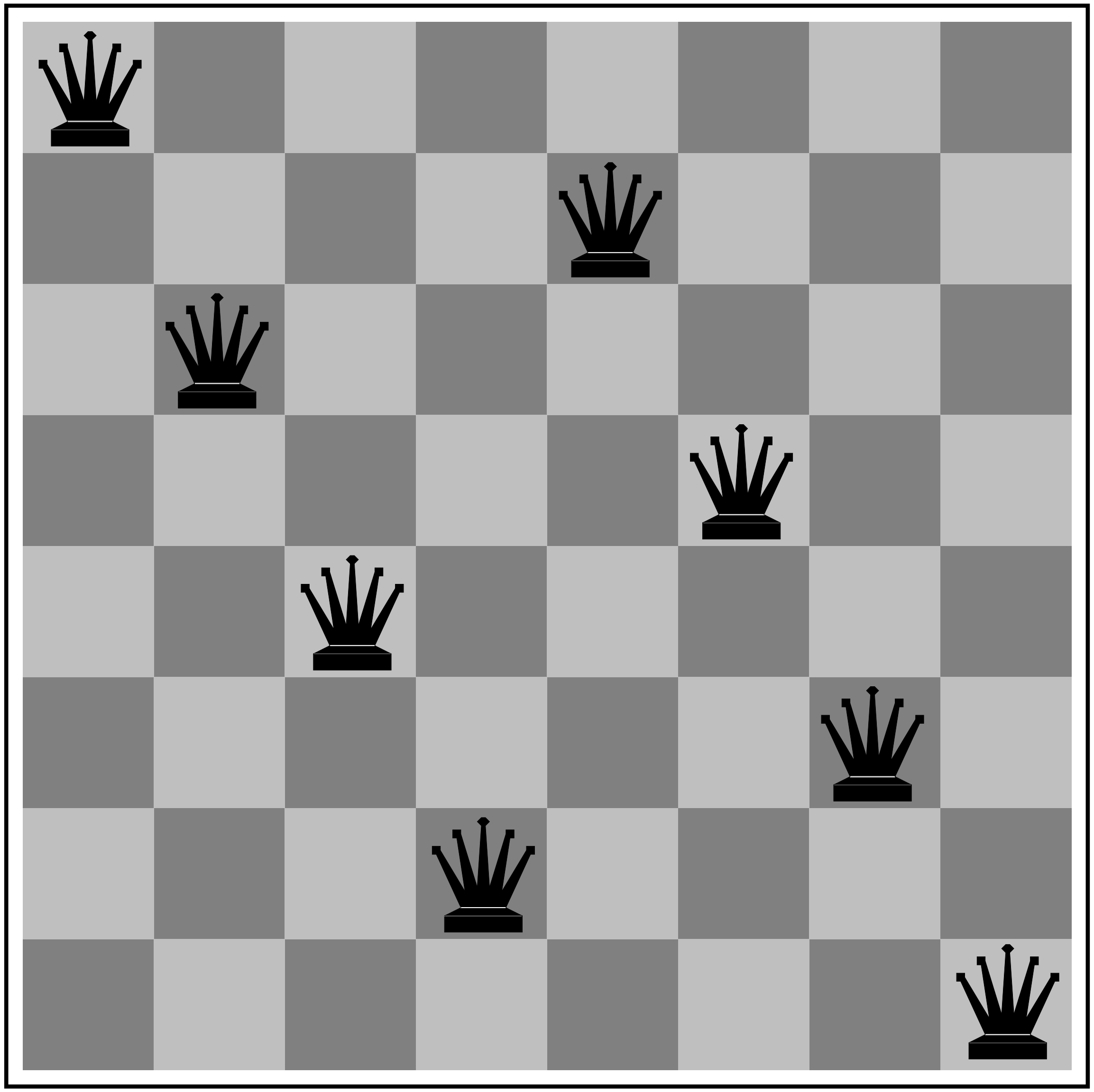
5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

The Eight Queens Problem



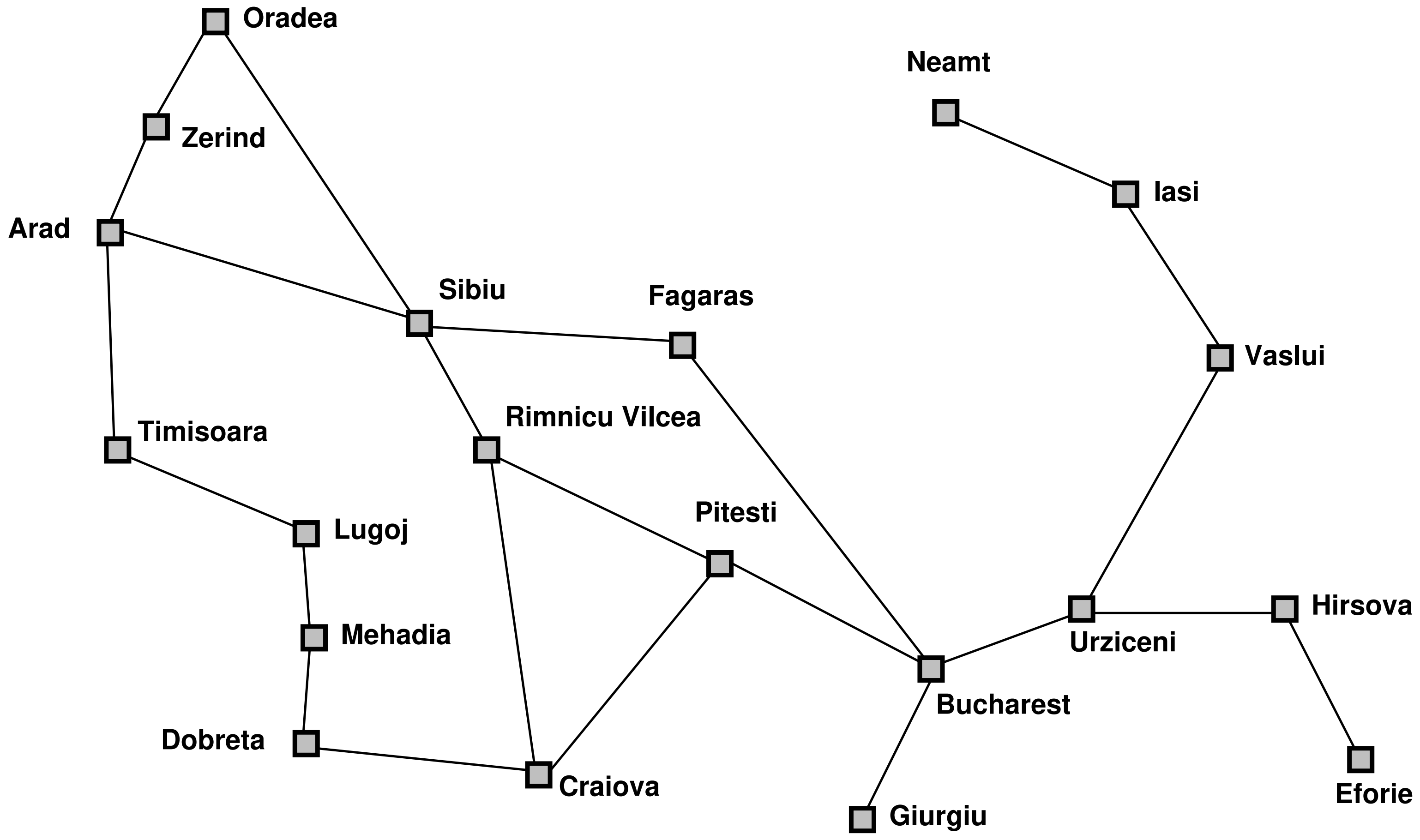
Real-world Problems

- **Route Finding**: network routing, automobiles, airline travel planning
- **Travelling Salesperson Problem (TSP):** Find the shortest tour (Hamiltonian circuit) that visits all nodes in a graph.
- **VLSI Layout:** cell layout, channel routing.
- **Robot Navigation:** continuous motion in 2 or 3 dimensions.
- **Assembly Sequencing:** plan for assembling a mechanical system, for loading and unloading a ship.

AI Search Algorithms

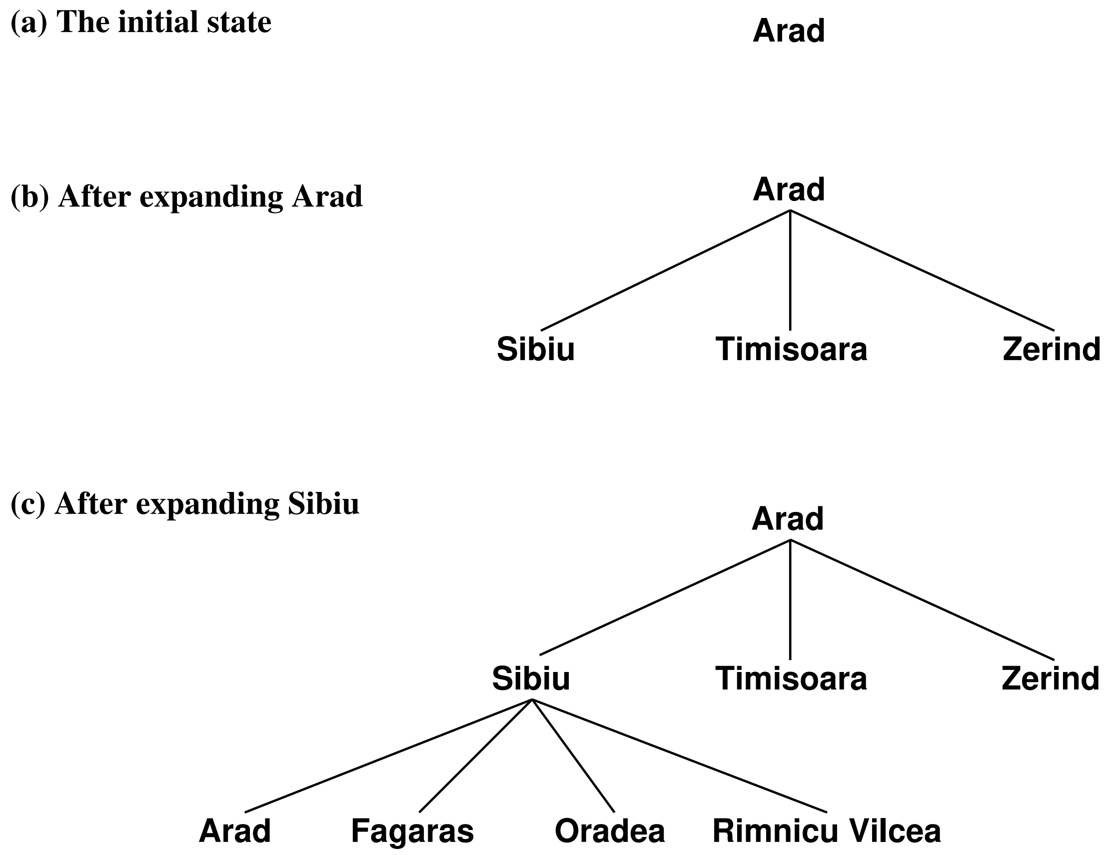
- Search-Based Agents search through problem space graphs to find solutions
- Problem space graphs are typically too large to store explicitly
- Search algorithms incrementally "grow" a subtree of the search graph

Example Problem: Navigating in Romania



Start state: Arad; Goal: Bucharest; Actions: Drive from one town to the next.

Search Algorithms Grow a Subtree of the Graph Dynamically



Informal Definition of GeneralSearch

```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
       if there are no candidates for expansion then return failure
       choose a leaf node for expansion according to strategy
       if the node contains a goal state then return the corresponding solution
       else expand the node and add the resulting nodes to the search tree
   end
```

Definition of GeneralSearch

Node data structure: state, parent-node, operator, depth, path-cost

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

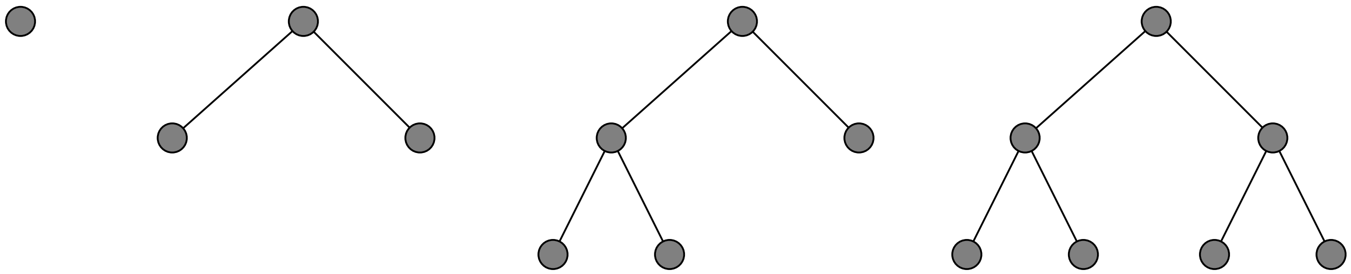
Evaluating a Search Algorithm

- **Completeness**: Is the strategy guaranteed to find a solution when there is one?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory does it need to find a solution?
- **Optimality**: Does the strategy find the highest-quality solution when there are several different solutions?

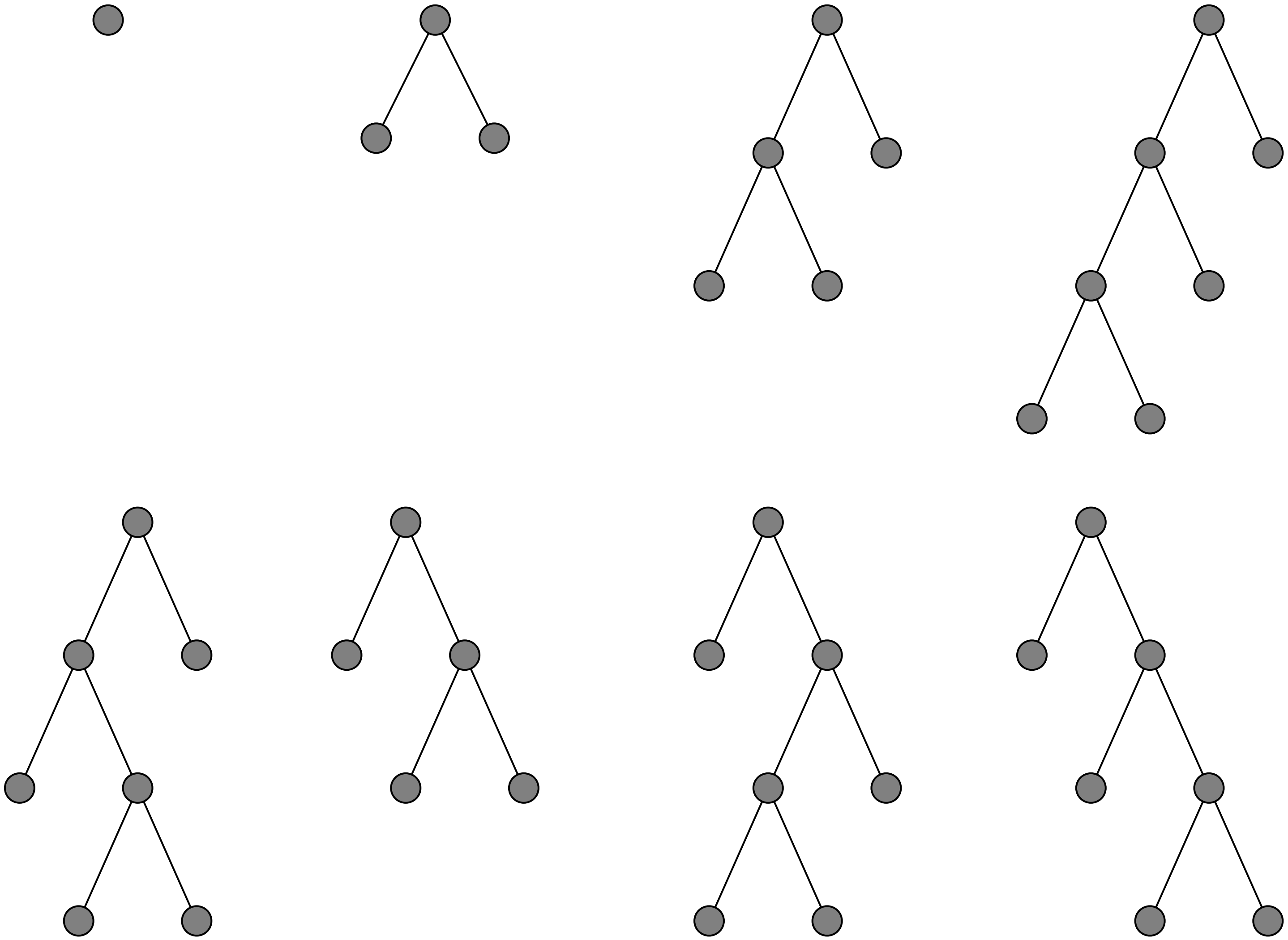
Search Strategies

- **Breadth-First Search**: Always expand the unexpanded node nearest to the root (i.e., push new nodes onto the end of the priority queue = FIFO queue).
- **Depth-First Search**: Always expand the unexpanded node at the deepest level (i.e., push new nodes onto the front of the queue = LIFO queue).
- **Uniform Cost Search**: Always expand the unexpanded node having to lowest path cost (so far). (i.e., use path cost as a priority = PRIORITY queue)

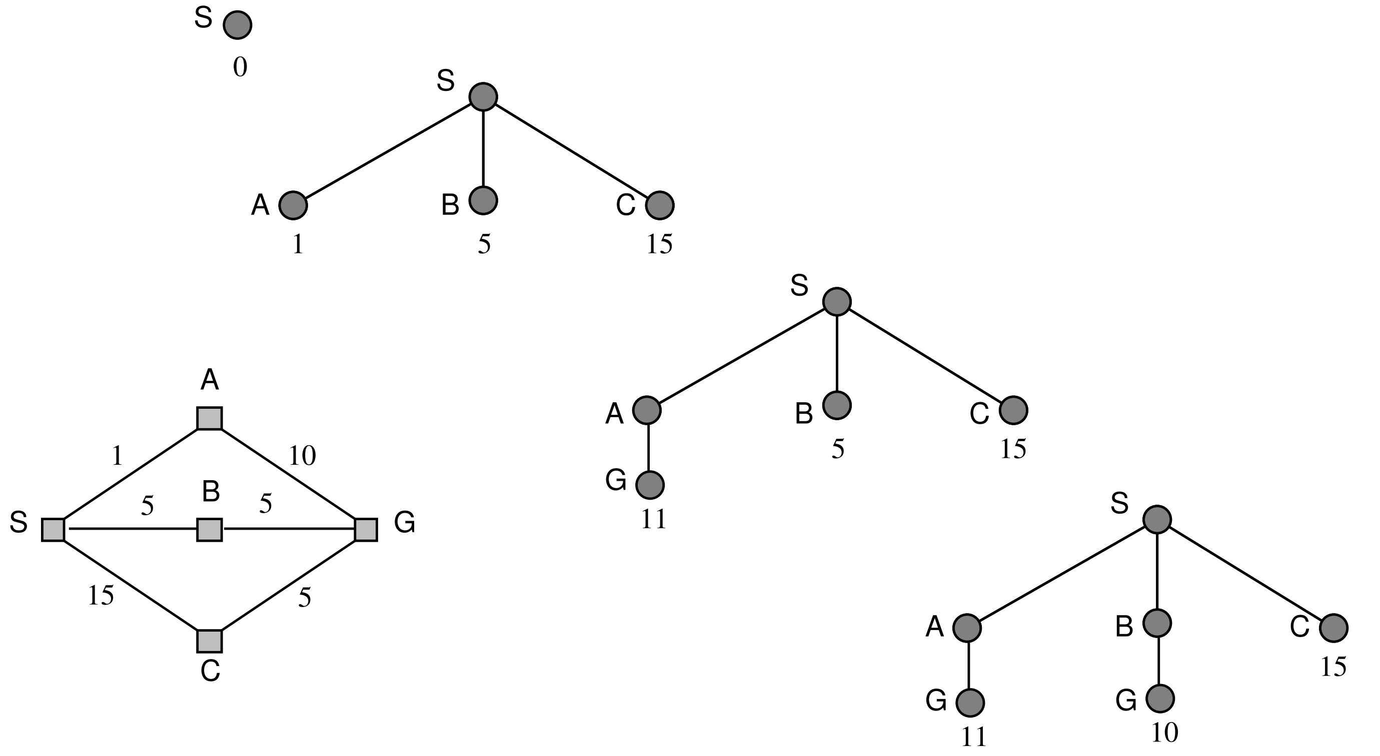
Breadth-First Search Behavior



Depth-First Search Behavior



Uniform Cost Search Behavior



(a) (b)

Space is the Real Problem

Two key parameters describe the search space:

- **Branching Factor:** b
- **Solution Depth:** d

Depth	Nodes	Time		Memory	
0	1	1	millisecond	100	bytes
2	111	.1	seconds	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

Comparing These Three Methods

Criterion	Breadth-First	Depth-First	Uniform-Cost
Time	b^d	b^m	b^d
Space	b^d	bm	b^d
Optimal?	Yes*	No	Yes**
Complete?	Yes	No	Yes

where

- b = branching factor
- d = depth of solution
- m = maximum depth of search tree (possibly infinite)

*BFS is optimal when all operators have equal cost.

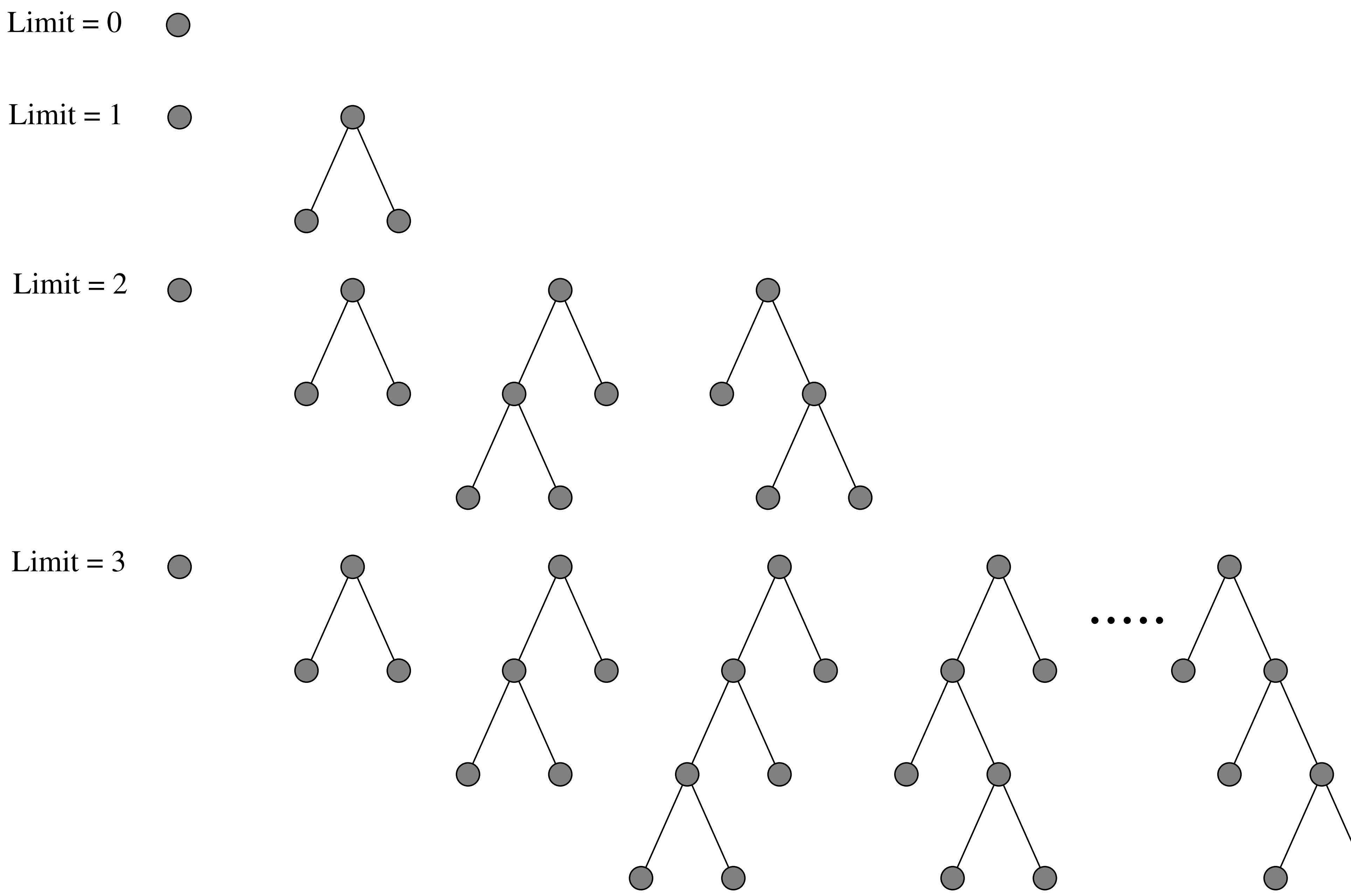
*UCS is optimal when all operators have non-negative cost.

Avoiding Infinite Paths in DFS via Iterative Deepening

Key Idea: Do repeated depth-first-search (DFS) to fixed depths. Increase the maximum depth at each iteration.

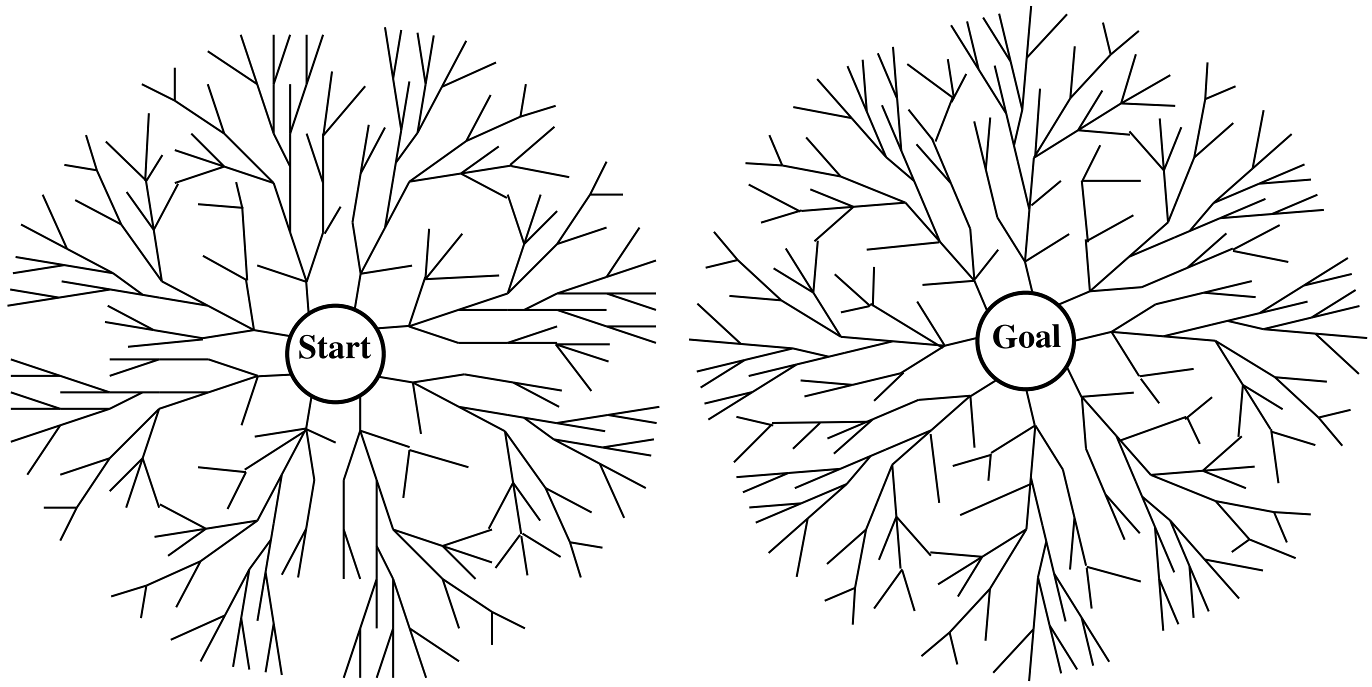
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
   inputs: problem, a problem

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


Limit = 0
Limit = 1
Limit = 2
Limit = 3

Bi-Directional Search

- Unique goal state
- Inverse operators are available
- Easy to compare generated states (hashing is possible)



Comparing Blind Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

DFID is the method of choice in a large search space with unknown solution depth (and no available heuristic information).