Search (I)

Search

- Search plays a key role in many parts of AI.

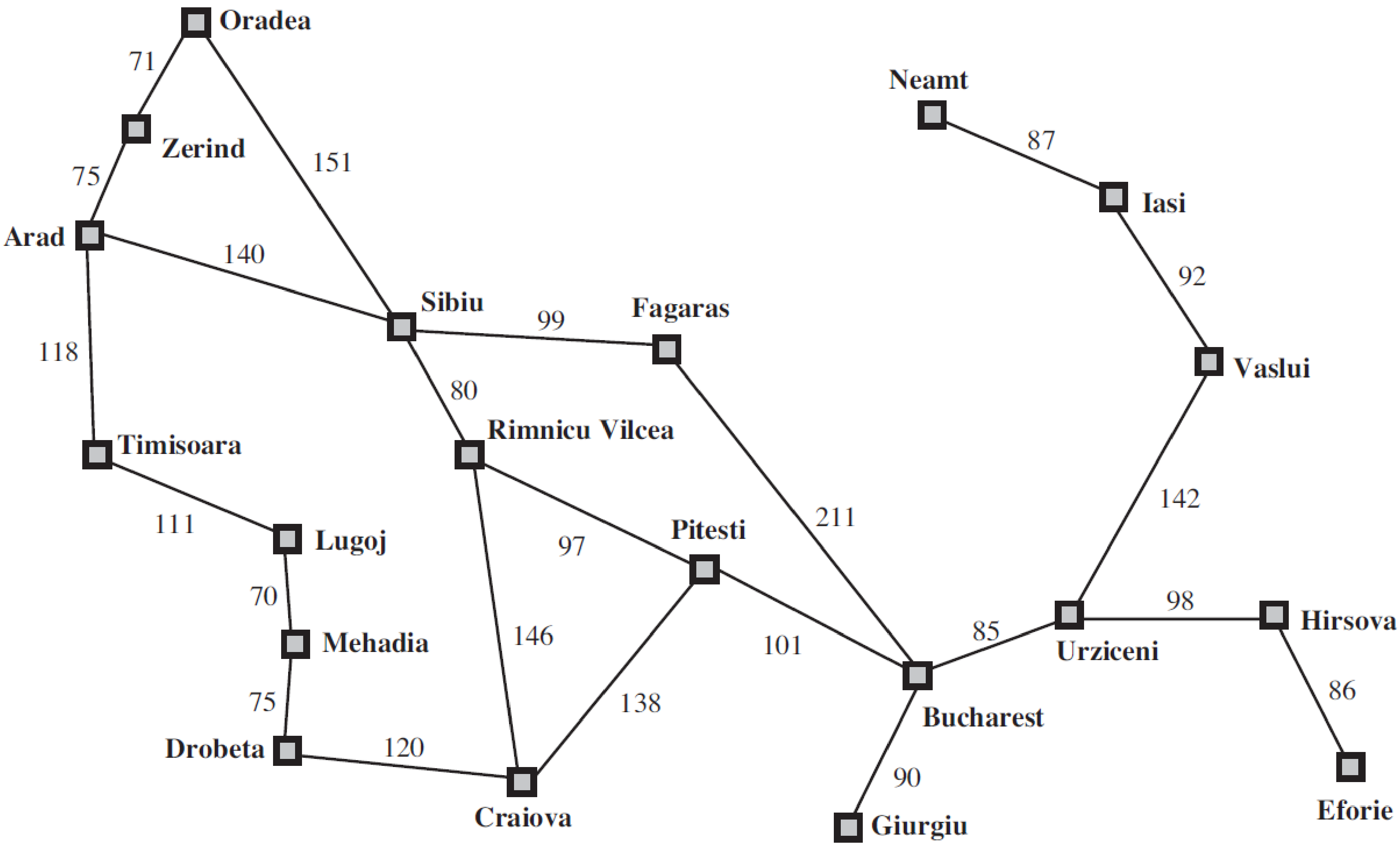
- These algorithms provide the **conceptual backbone** of almost every approach to the **systematic exploration of alternatives**.

- Idea: reduce the problem to be solved to one of searching a graph.

Classes of search

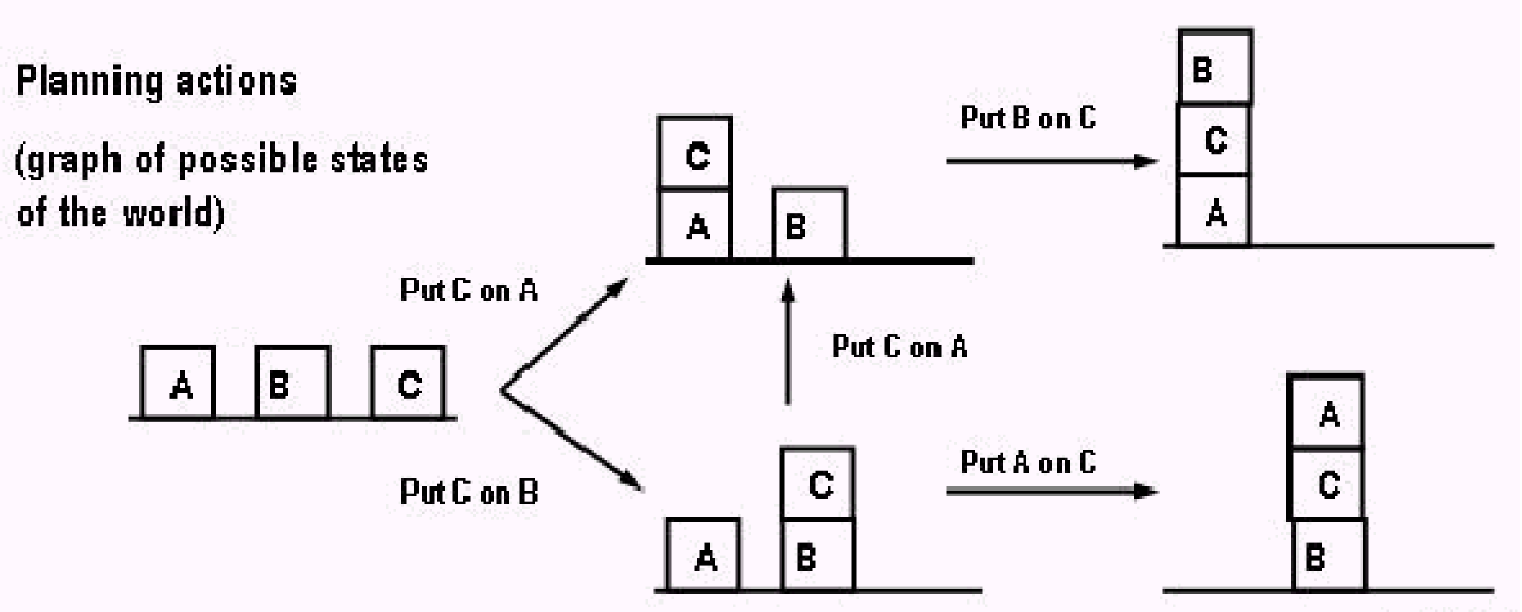
Uninformed Don’t use any background knowledge of the domain	Informed Use background knowledge (heuristics) of the domain to make search faster
Any solution	Optimal solution

Romania graph



Another graph example

- However, graphs can also be much more **abstract**.



Formally: State-space graph

- The **state space** forms a directed **graph**
  - graph nodes ≡ states
  - graph edges ≡ actions

Defining a problem formally

A problem is defined by four items:

1. initial state e.g., "at Arad"

2. actions and successor function *S:* = set of **action-state** tuples
   - e.g., *S(Arad) = {(goZerind, Zerind), (goTimisoara, Timisoara), (goSilbiu, Silbiu)}*
   - You can safely ignore actions in most problems (just successor states would do)

3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*

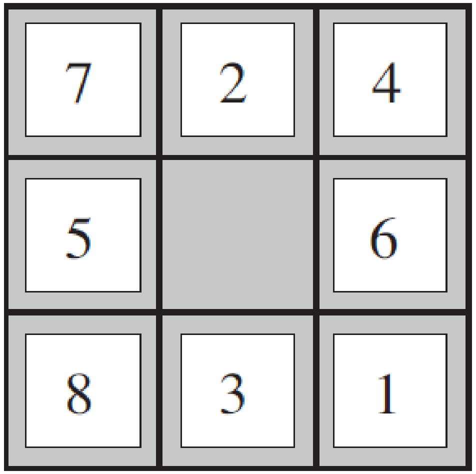
4. step cost
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

- **A solution is a sequence of actions leading from the initial state to a goal state**

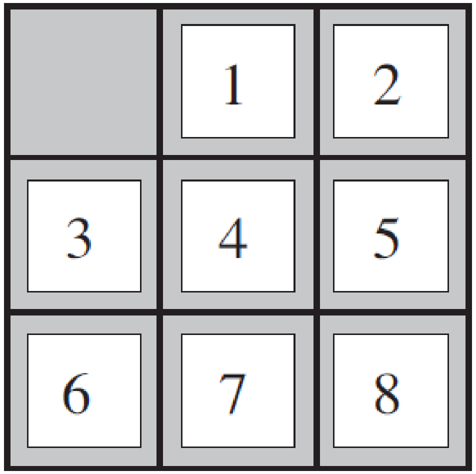
Problem in Java

```
public abstract class Problem {
        public Object initialState;
        abstract boolean goal_test(Object state);
        abstract Set<Object> getSuccessors(Object state);
        abstract double step_cost(Object fromState, Object toState);
}
```

Example: The 8-puzzle



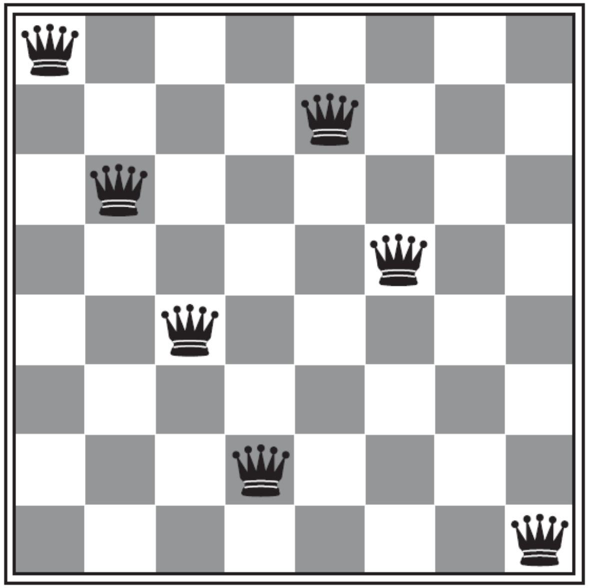
Start State



Goal State

- states? locations of tiles (i.e. board configurations)
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

Example: The 8-queens problem

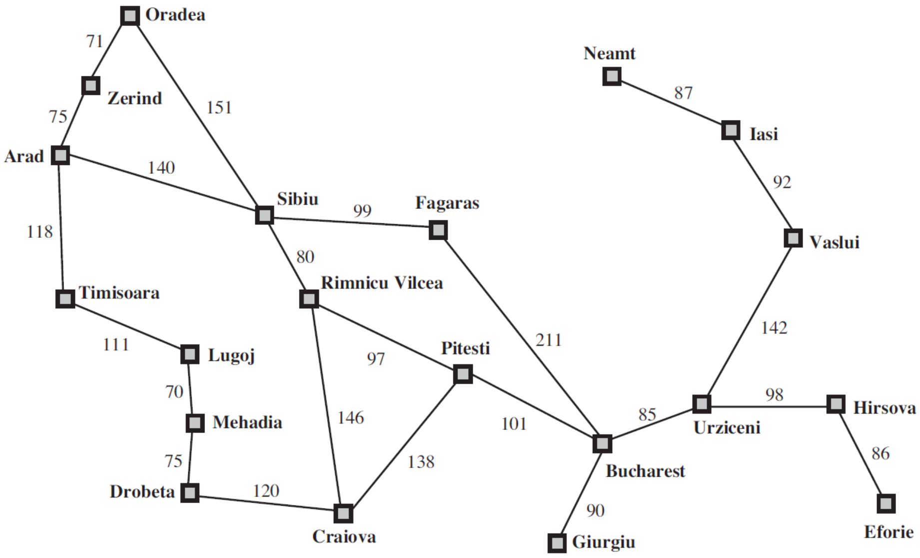


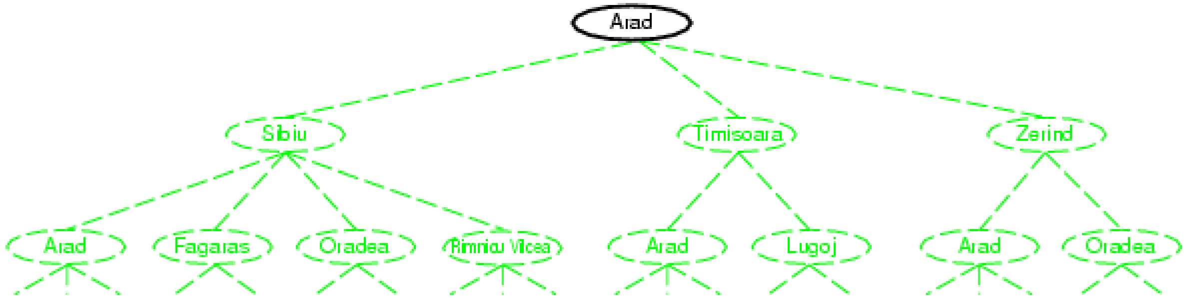
Is this figure really a valid state?

Why not have as states board configurations where all the 8 queens are in?

- states? All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another
- actions? Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen
- goal test? = 8 queens are on the board, none attacked
- path cost? 1 per move

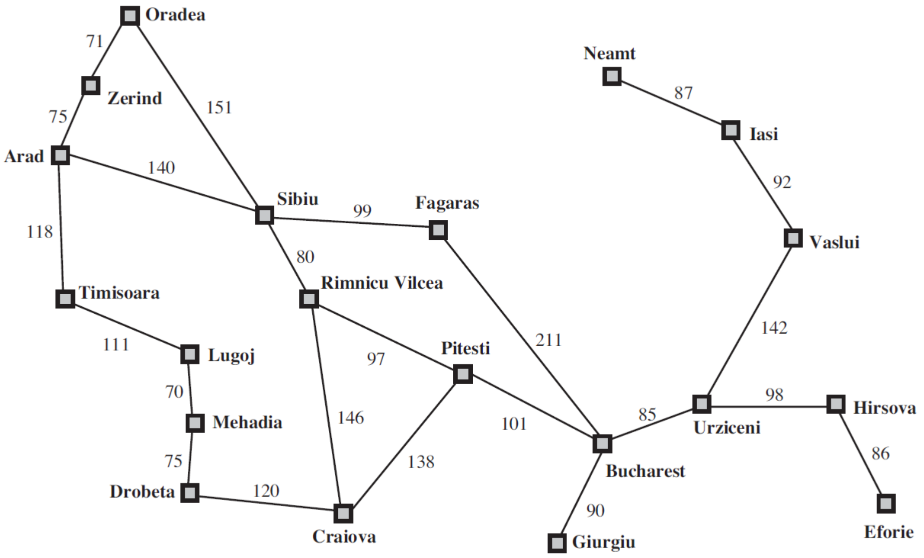
Tree search example

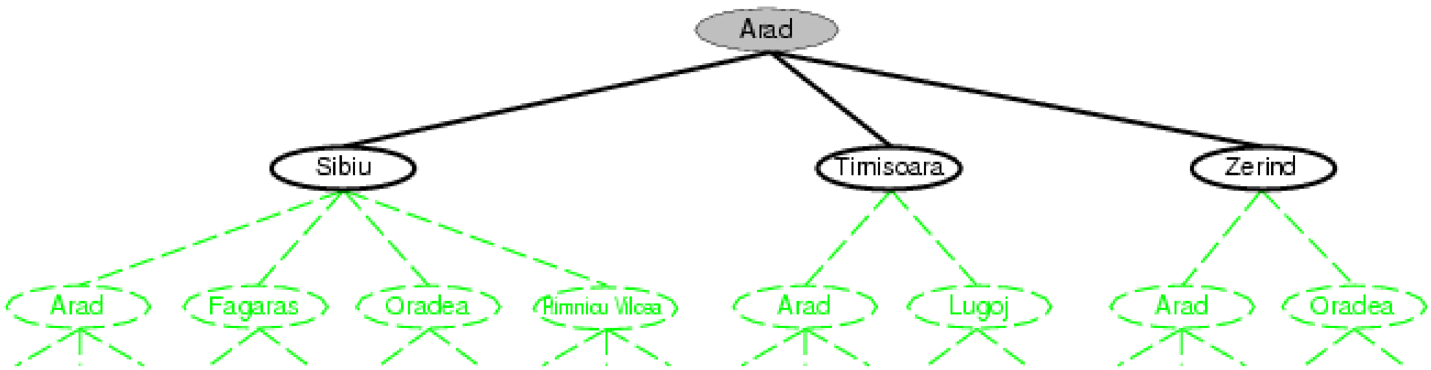




Notion: “tree node expansion”

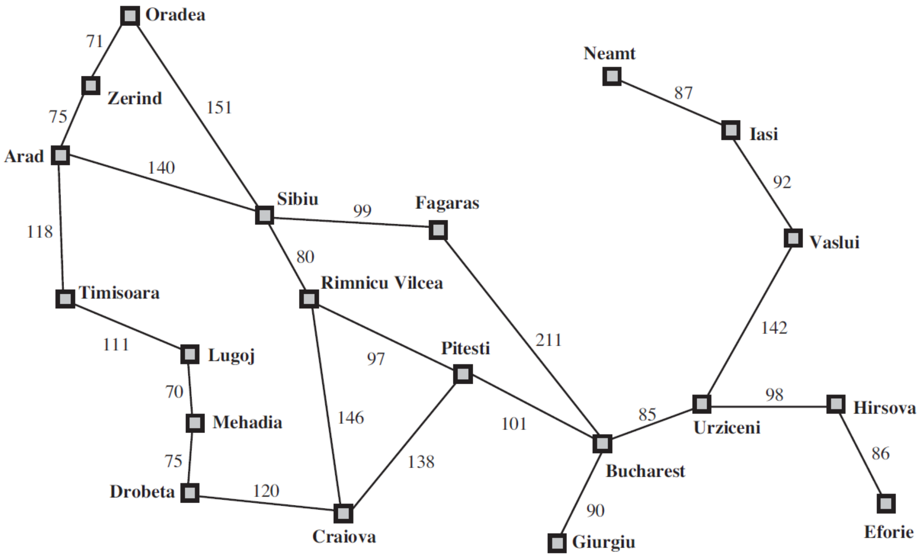
Tree search example

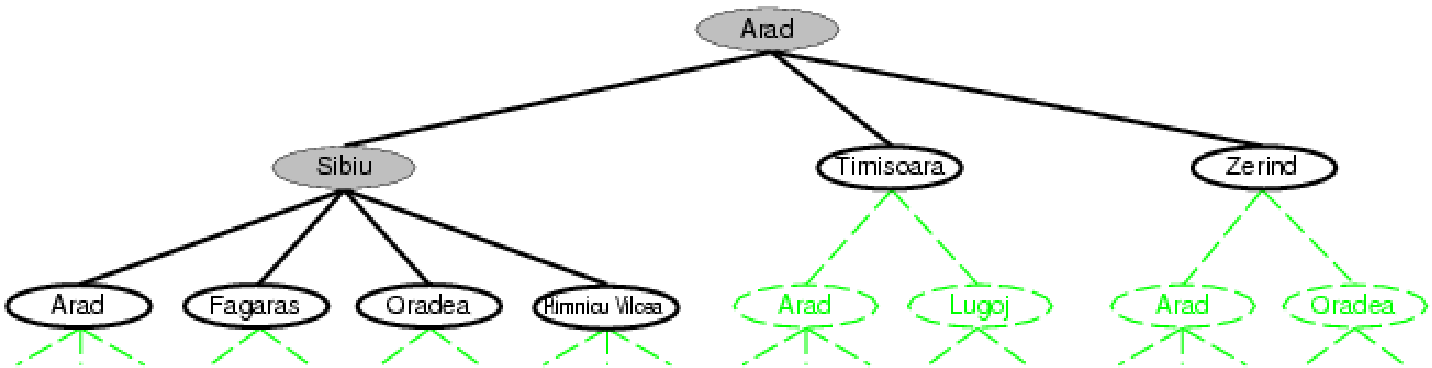




Notion: “frontier”

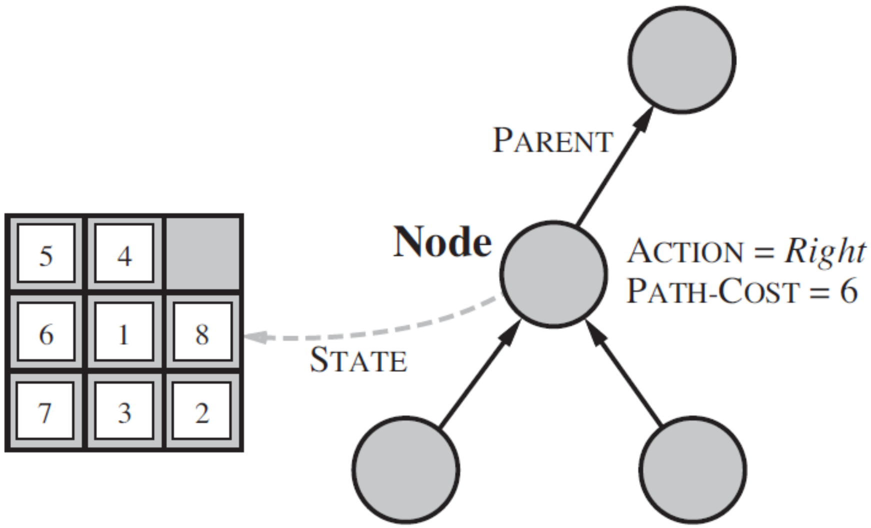
Tree search example





Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a bookkeeping data structure constituting of state, parent node, action, path cost (g), depth



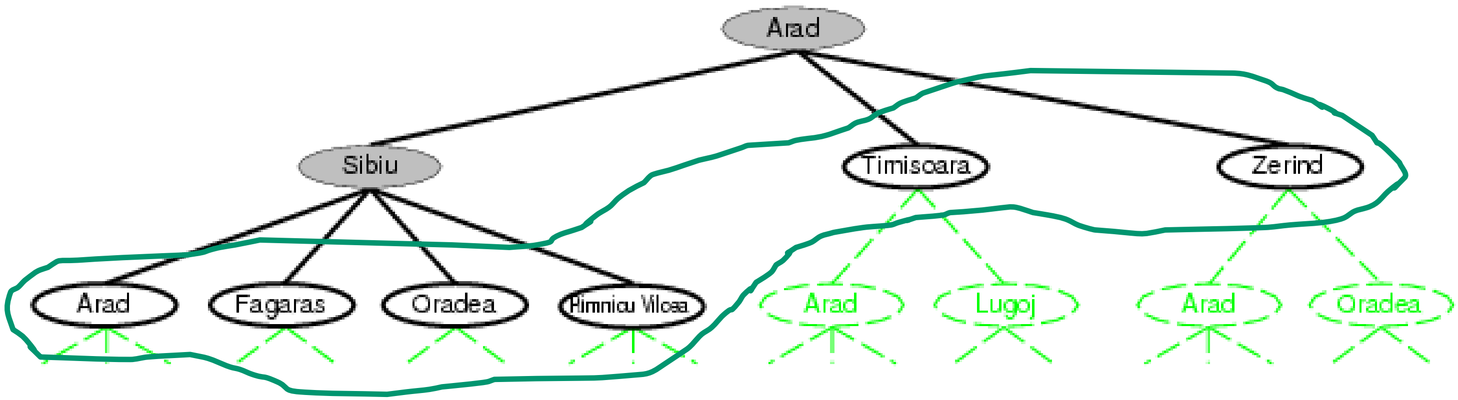
Node in Java

```
public class Node {
        Object state;
        Node parent_node;
        double path_cost;
        int depth;
        int order = -1; //order of expansion; default of -1 means not expanded
}
```

The **frontier** (or **fringe**)

- The collection of nodes that have been generated but not yet expanded is called **frontier**.

- We will implement collection of nodes as **queues** of a certain strategy (e.g. FIFO, LIFO, Priority).



Frontier in Java

```
public interface Frontier {
        boolean isEmpty();
        Node remove();
        void insert(Node n);
        void insertAll(Set<Node> set_of_nodes);
}
```

Example: FrontierFIFO

```
public class FrontierFIFO implements Frontier {
        Deque<Node> queue = new ArrayDeque<Node>();

        public boolean isEmpty() { return queue.isEmpty(); }

        public Node remove() { return queue.remove(); }

        public void insert(Node n) { queue.add(n); }

        public void insertAll(Set<Node> set_of_nodes) {
                for(Node n : set_of_nodes)
                        queue.add(n);
        }
}
```

TreeSearch vs GraphSearch

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
    remove node *n* from the frontier
    if *n* contains a goal state **then return** corresponding solution
    expand *n* adding the resulting nodes to the frontier

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
    remove node *n* from the frontier
    if *n* contains a goal state **then return** corresponding solution
    if the state of *n* is not in **explored**
      add the state of *n* to **explored**
      expand *n* adding the resulting nodes to the frontier

Basic idea: Exploration of state space by generating successors of already-explored states (i.e. expanding states)

Both algorithms search graphs of states.

The difference is that Graph-Search checks to see if a state has been explored before and if so it does not expand it again.

Graph-Search is slightly different from the book.

Creating an Initial Node and Expanding a Node

```
function MakeNode(problem, state) returns a node
    let node be a new node
    node.state = state; node.parent_node = null; node.path_cost = 0; node.depth = 0
    return node
```

```
function Expand(problem, node) returns a set of nodes
   successor_states = problem. getSuccessors(node.state)
   intialize successors to be the empty set
   for each s in successor_states
      let n be a new node
      n.state = s
      n.parent = node
      n.path_cost = node.path_cost + problem.step_cost(node.state, s)
      n.depth = node.depth + 1
      add n to successors
   return successors
```

Search strategies

- A search strategy is defined by picking the order of node expansion (i.e. the queue strategy)

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

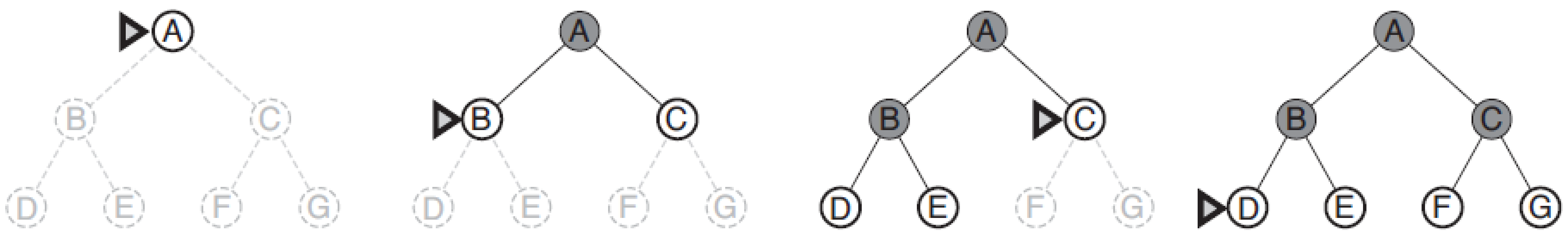
- Time and space complexity are measured in terms of
  - ***b****:* maximum branching factor of the search tree
  - ***d****:* depth of the least-cost solution
  - ***m***: maximum depth of the state space

Search strategies

- Breadth-first search (BFS)
- Uniform-cost search (UCS)
- Depth-first search (DFS)
- Depth-limited search (DLS)
- Iterative deepening search (IDS)

BFS

- FIFO queue used.
- Puts all newly generated successors at the end of the queue, which means that *shallow nodes are expanded before deeper nodes*.
  - i.e. **pick from the frontier to expand the shallowest unexpanded node**



Properties of breadth-first search

- Complete?
  - Yes (if b is finite)
- Time?
  - $O(b^{d+1})$ is the total number of nodes generated
- Space?
  - $O(b^{d+1})$ (keeps every node in memory)
- Optimal?
  - Yes (if cost is a non-decreasing function of depth, e.g. when we have 1 cost per step)

Suppose b=10, 1 million nodes/sec, 1000 bytes/node

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	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^{16}	350 years	10 exabytes

Uniform-cost search

- Expand least-cost unexpanded node.
- The algorithm expands nodes in order of increasing path cost.
- Therefore, the first goal node selected for expansion is the optimal solution.

- Implementation:
  - *frontier* = queue ordered by path cost (priority queue)
- Equivalent to breadth-first if step costs all equal

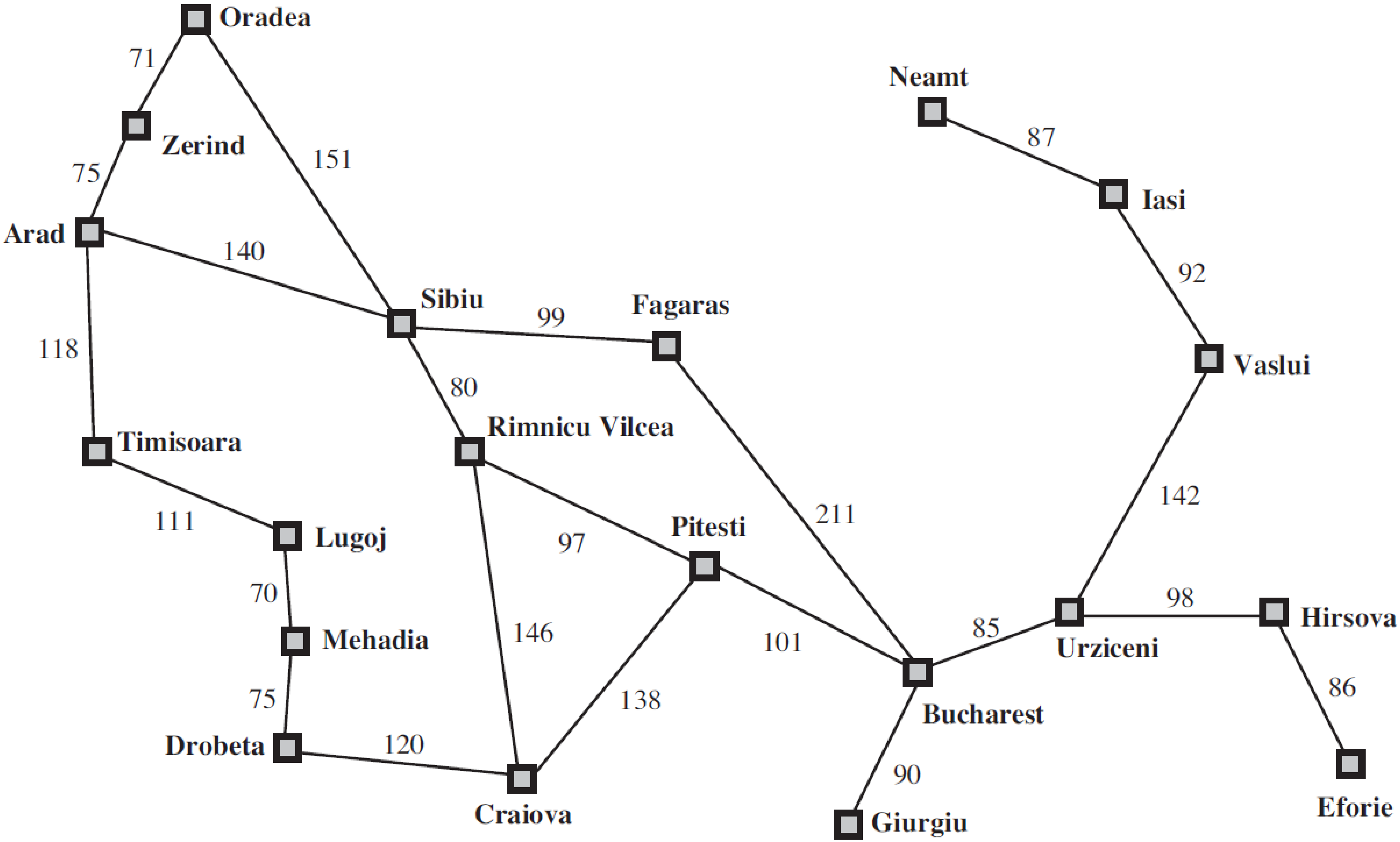
- Complete? Yes, if step *cost* $\geq \varepsilon$ (i.e. not zero)

- Time? number of nodes with $g \leq cost$ of optimal solution, $\boldsymbol{O(b^{C^*/\varepsilon})}$ where C^* is the cost of the optimal solution, and some of their children

- Space? Number of nodes with $g \leq cost$ of optimal solution, $\boldsymbol{O(b^{C^*/\varepsilon})}$, and some of their children

- Optimal? Yes – nodes expanded in increasing order of $g(n)$

Try it here


Oradea
71
Neamt
Zerind
151
87
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
146
85
Hirsova
Mehadia
101
Urziceni
86
75
138
Bucharest
Drobeta
120
90
Craiova
Eforie
Giurgiu

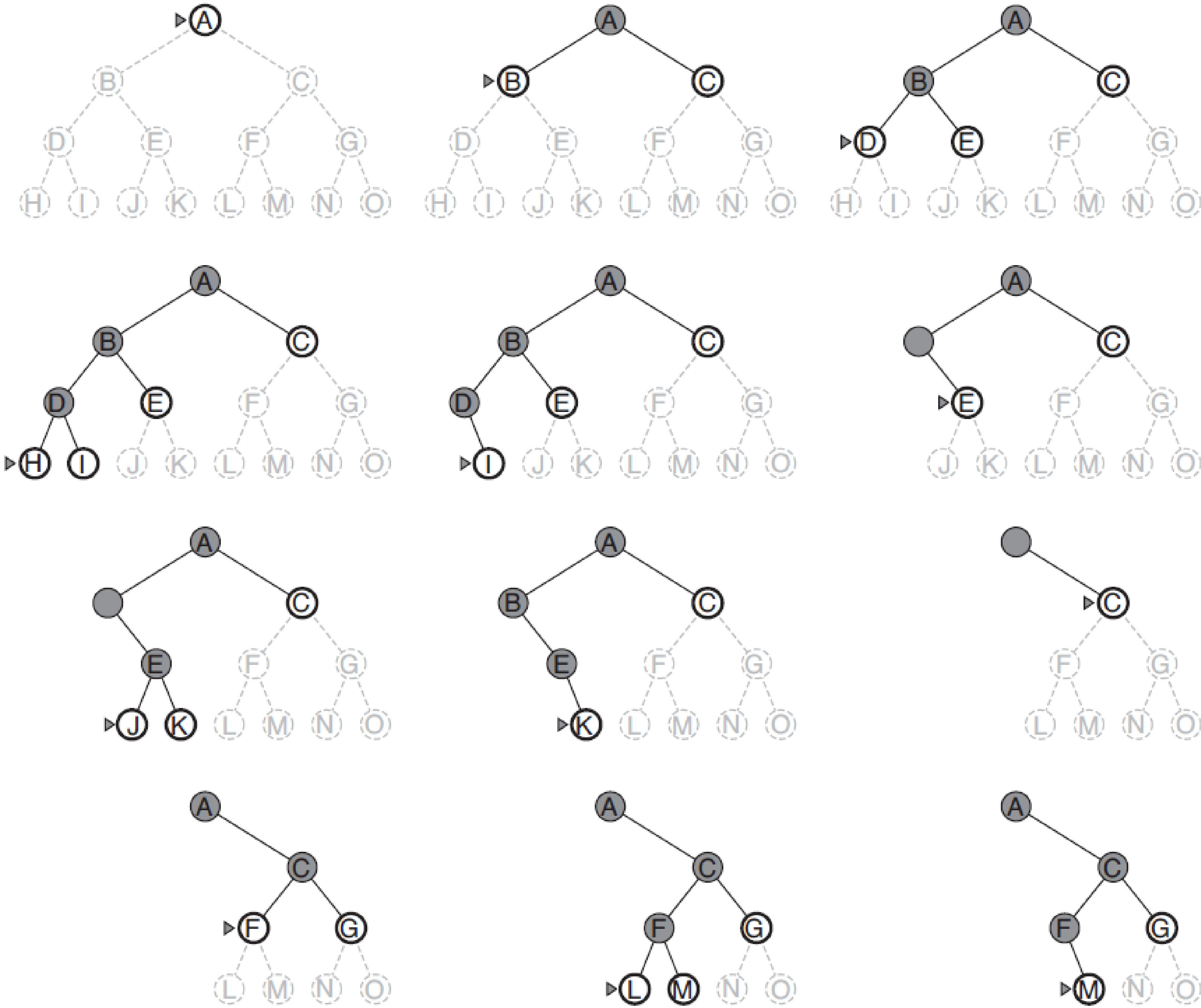
Remark

- Book has an extra operation that is not strictly needed:
  - "**if** child.STATE is in frontier with higher PATH-COST **then** replace that frontier node with child"

Depth-first search

- Expand deepest unexpanded node

- Implementation: *frontier* = LIFO queue, i.e., put successors at the front



A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G I J K L M N O
A C E F G J K L M N O
A C E F G J K L M N O
A B C E F G K L M N O
C F G L M N O
A C F G L M N O
A C F G L M N O
A C F G M N O

Properties of depth-first **tree-search**

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path (GraphSearch)
    - → 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
  - e.g. in the previous figure, if both C and J are goal states, it will output J even though C is a better goal (shallower goal)

- This is the only algorithm running in linear space so far.
  - On the other hand, depth-first graph-search doesn't have linear space.

Depth-limited search (DFS, LIFO Frontier)

```
function DepthLimited-TREE-SEARCH(problem, limit) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     remove n from the frontier
     if n contains a goal state then return the corresponding solution
     if the depth of n is less than limit then
          expand n adding the resulting nodes to the frontier
```

```
function DepthLimited-GRAPH-SEARCH(problem, limit) 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
     remove n from the frontier
     if n contains a goal state then return the corresponding solution
     if the state of n is not in explored and the depth of n is less than limit
        add the state of n to explored
        expand n adding the resulting nodes to the frontier
```

The book has a recursive version.

Iterative deepening search

```
function IterativeDeepening-TREE-SEARCH(problem) returns a solution, or failure
  for limit=0 to infinity
     result = DepthLimited-TREE-SEARCH(problem, limit)
     if result is a solution return result
```

```
function IterativeDeepening-GRAPH-SEARCH(problem) returns a solution, or failure
  for limit=0 to infinity
     result = DepthLimited-GRAPH-SEARCH(problem, limit)
     if result is a solution return result
```

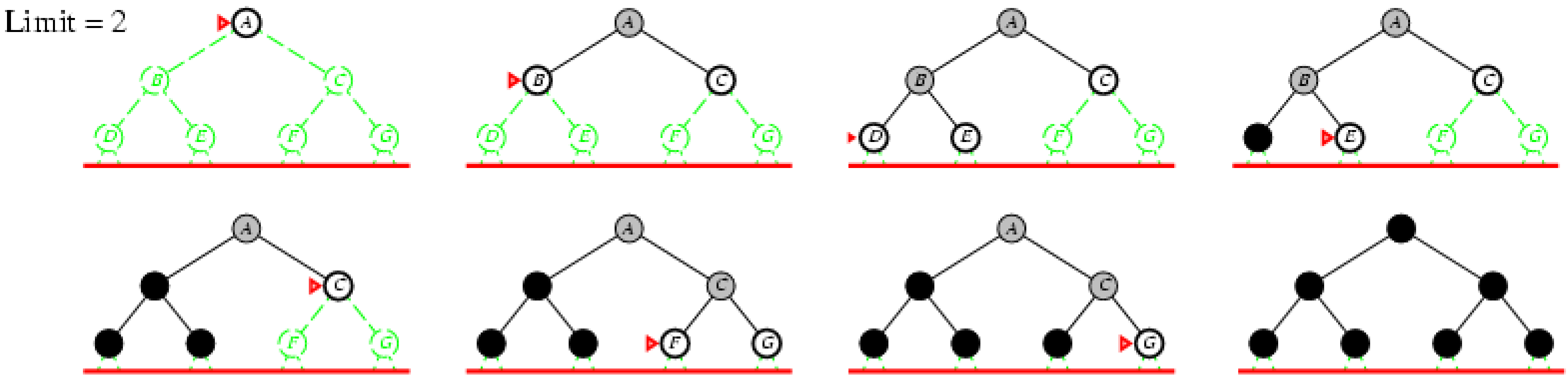
Iterative deepening search l =0

Limit = 0

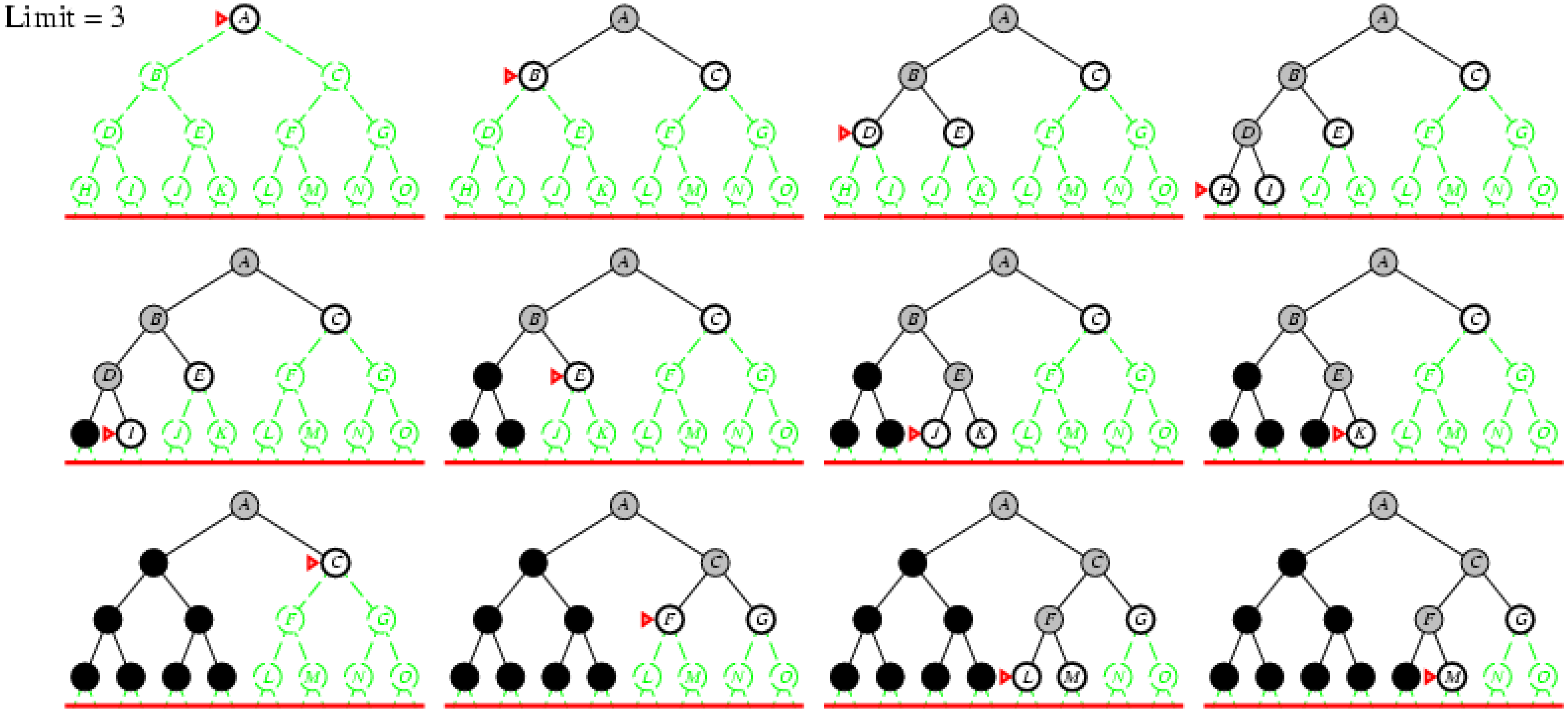
Iterative deepening search l =1

Limit = 1

Iterative deepening search l =2



Iterative deepening search l =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\ b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Summary of algorithms

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

Class problem

- You have three jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet.
- You can fill the jugs up, or empty them out from one another or onto the ground.
- You need to measure out exactly one gallon.