Problems & Search

Search Problem

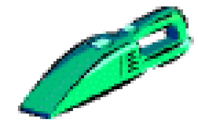
- Problem solving = Searching for a goal state
- **State Space**
- **Actions**
- **Goal test**: applicable to a single state problem to determine if it is the goal state.
- **Path cost**: relevant if more than one path leads to the goal, and we want the shortest path.

State Space

- State spaces can be defined as a tuple [*N*, *A*, *S*, *G*] where:
  - *N* is a set of states
  - *A* is a set of arcs connecting the states
  - *S* is a nonempty subset of *N* that contains start states
  - *G* is a nonempty subset of *N* that contains the goal states.
- A state space has some common properties:
  - complexity, where branching factor is important
  - structure of the space, see also graph theory: directionality of arcs, tree, or Rooted graph

Toy Problems

(1) Vacuum World as a Single-state problem



- **Initial State**: one of the 8 states shown above.
- **Actions:** move Left, move Right, Suck.
- **Goal Test**: no dirt in any square.
- **Path cost**: each action costs 1.

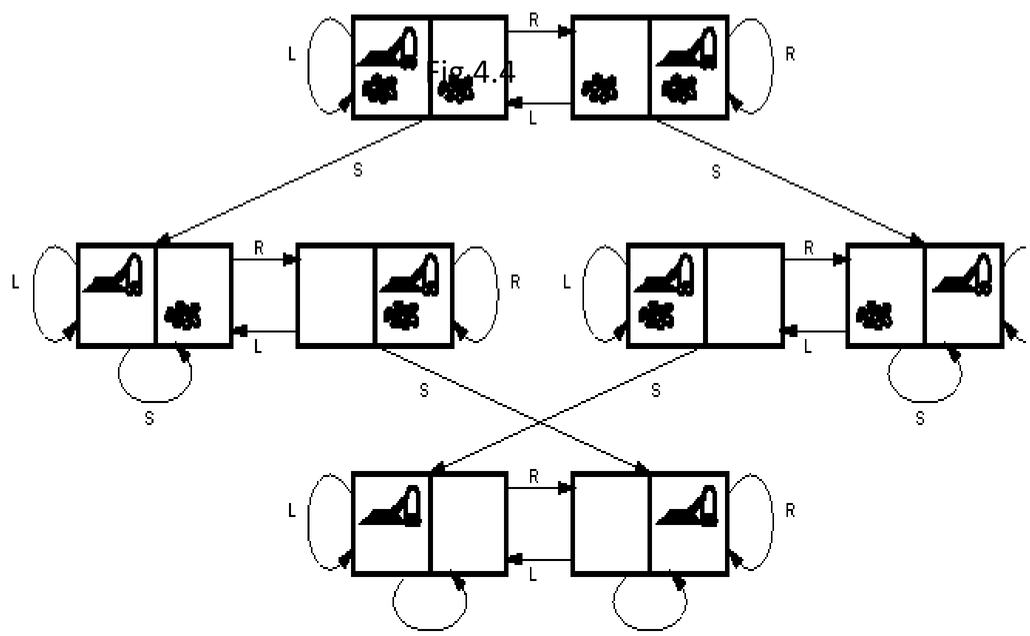
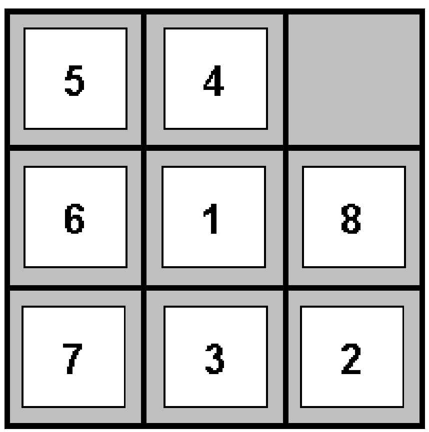


Fig.4.4

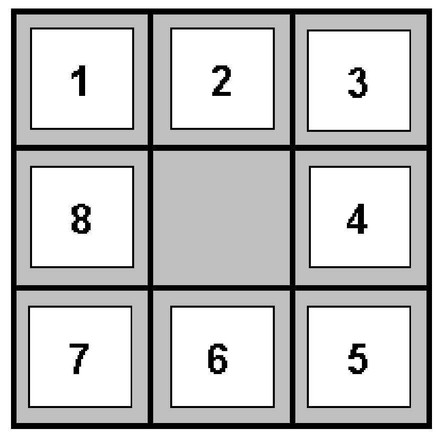
Toy Problems

(3) 8-puzzle problem

- **Initial State**: The location of each of the 8 tiles in one of the nine squares
- **Actions**: blank moves (1) Left (2) Right (3) Up (4) Down
- **Goal Test**: state matches the goal configuration
- **Path cost**: each step costs 1, total path cost = no. of steps



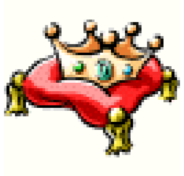
Start State



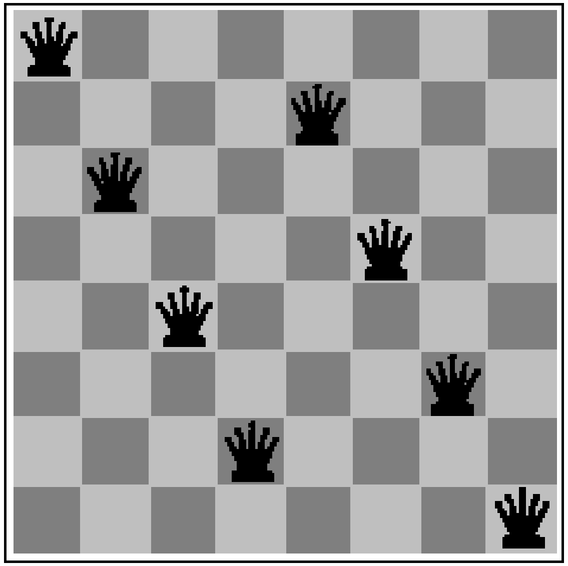
Goal State

Toy Problems

(4) 8-queens problem



- **Initial State**: Any arrangement of 0 to 8 queens on board.
- **Actions**: add a queen to any square.
- **Goal Test**: 8 queens on board, none attacked.
- **Path cost**: not applicable or Zero (because only the final state counts, search cost might be of interest).



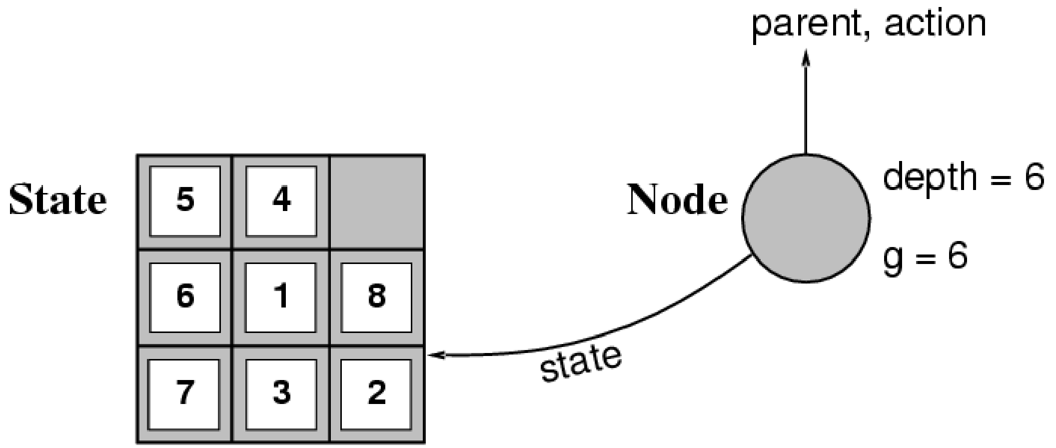
Toy Problems
(5) Cryptarithmetic

- **Initial State**: a cryptarithmetic puzzle with some letters replaced by digits.
- **Actions**: replace all occurrences of a letter with a non-repeating digit.
- **Goal Test**: puzzle contains only digits, and represents a correct sum.
- **Path cost**: not applicable or 0 (because all solutions equally valid).

```
 FORTY          Solution:  29786     F=2, O=9, R=7, etc
+  TEN                       850
+  TEN                       850
 -----                     -----
 SIXTY                     31486
```

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth



- The Expand function creates new nodes, filling in the various fields and using the Successor Fn of the problem to create the corresponding states.

State Space Search: Water Jug Problem

"You are given two jugs, a 4-litre one and a 3-litre one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 litres of water into 4-litre jug."

State Space Search: Water Jug Problem

- State: (x, y)

  $x = 0, 1, 2, 3$, or 4 $\qquad\qquad$ $y = 0, 1, 2, 3$

- Start state: $(0, 0)$.
- Goal state: $(2, n)$ for any n.
- Attempting to end up in a goal state.

State Space Search: Water Jug Problem

1. $(x, y) \rightarrow (4, y)$
   if $x < 4$
2. $(x, y) \rightarrow (x, 3)$
   if $y < 3$
3. $(x, y) \rightarrow (x - d, y)$
   if $x > 0$
4. $(x, y) \rightarrow (x, y - d)$
   if $y > 0$

State Space Search: Water Jug Problem

5. $(x, y) \rightarrow (0, y)$

   if $x > 0$

6. $(x, y) \rightarrow (x, 0)$

   if $y > 0$

7. $(x, y) \rightarrow (4, y - (4 - x))$

   if $x + y \geq 4, y > 0$

8. $(x, y) \rightarrow (x - (3 - y), 3)$

   if $x + y \geq 3, x > 0$

State Space Search: Water Jug Problem

9. $(x, y) \rightarrow (x + y, 0)$

   if $x + y \leq 4, y > 0$

10. $(x, y) \rightarrow (0, x + y)$

    if $x + y \leq 3, x > 0$

11. $(0, 2) \rightarrow (2, 0)$

12. $(2, y) \rightarrow (0, y)$

State Space Search: Water Jug Problem

1. current state = (0, 0)
2. Loop until reaching the goal state (2, 0)
   - Apply a rule whose left side matches the current state
   - Set the new current state to be the resulting state

(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
(0, 2)
(2, 0)

State Space Search: Water Jug Problem

The role of the condition in the left side of a rule
$\Rightarrow$ restrict the application of the rule
$\Rightarrow$ more efficient

1. $(x, y) \rightarrow (4, y)$
   if $x < 4$
2. $(x, y) \rightarrow (x, 3)$
   if $y < 3$

State Space Search: Water Jug Problem

Special-purpose rules to capture special-case knowledge that can be used at some stage in solving a problem

11. $(0, 2) \rightarrow (2, 0)$

12. $(2, y) \rightarrow (0, y)$

State Space Search: Summary

1. Define a state space that contains all the possible configurations of the relevant objects.
2. Specify the initial states.
3. Specify the goal states.
4. Specify a set of rules:
   - What are unstated assumptions?
   - How general should the rules be?
   - How much knowledge for solutions should be in the rules?

Search Strategies

- Blind (un-informed) search strategies
  - Breadth-first search
  - Uniform cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bi-directional search
- Heuristic (informed) search strategies

Search strategies

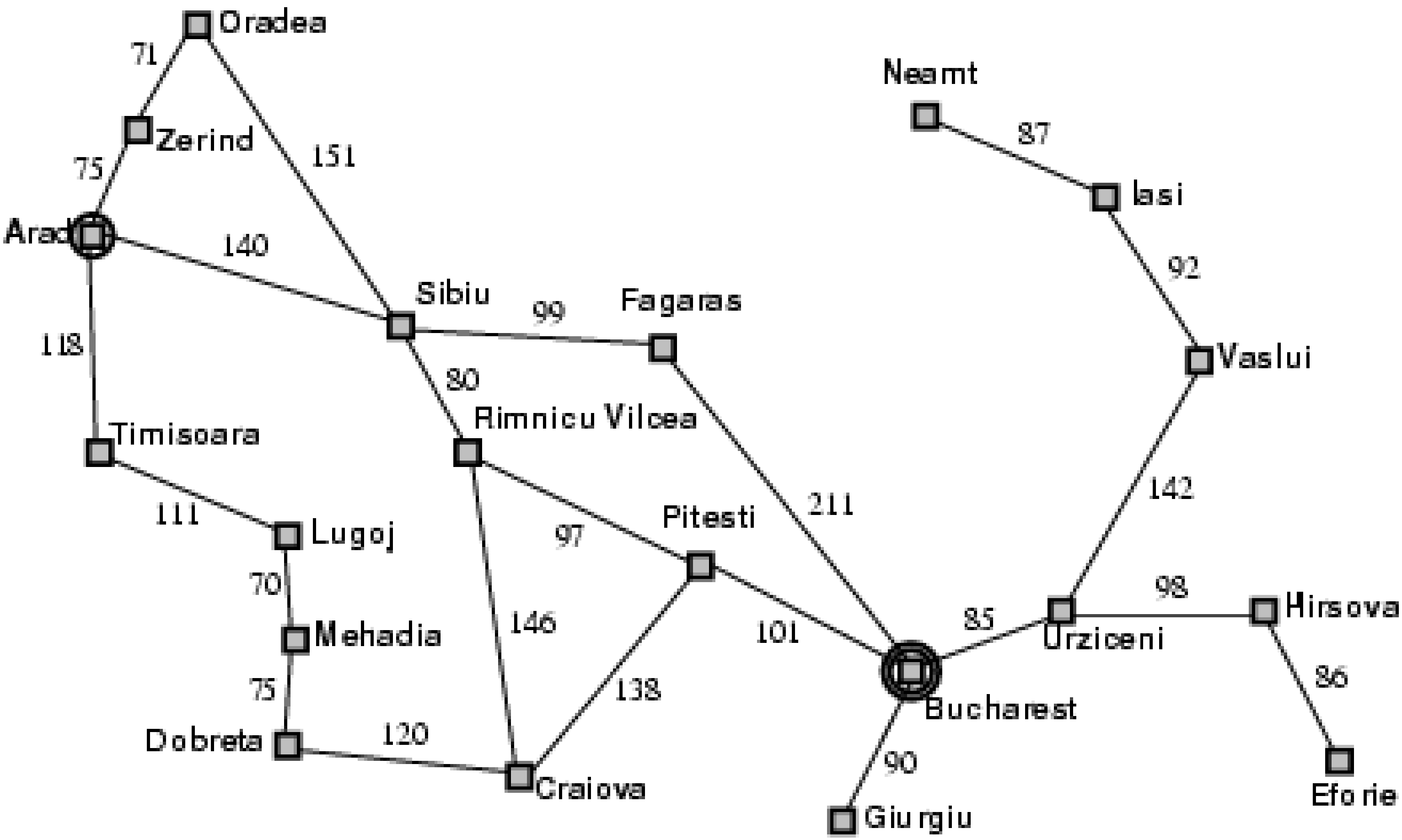
- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: 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 (may be ∞)

Tree Search

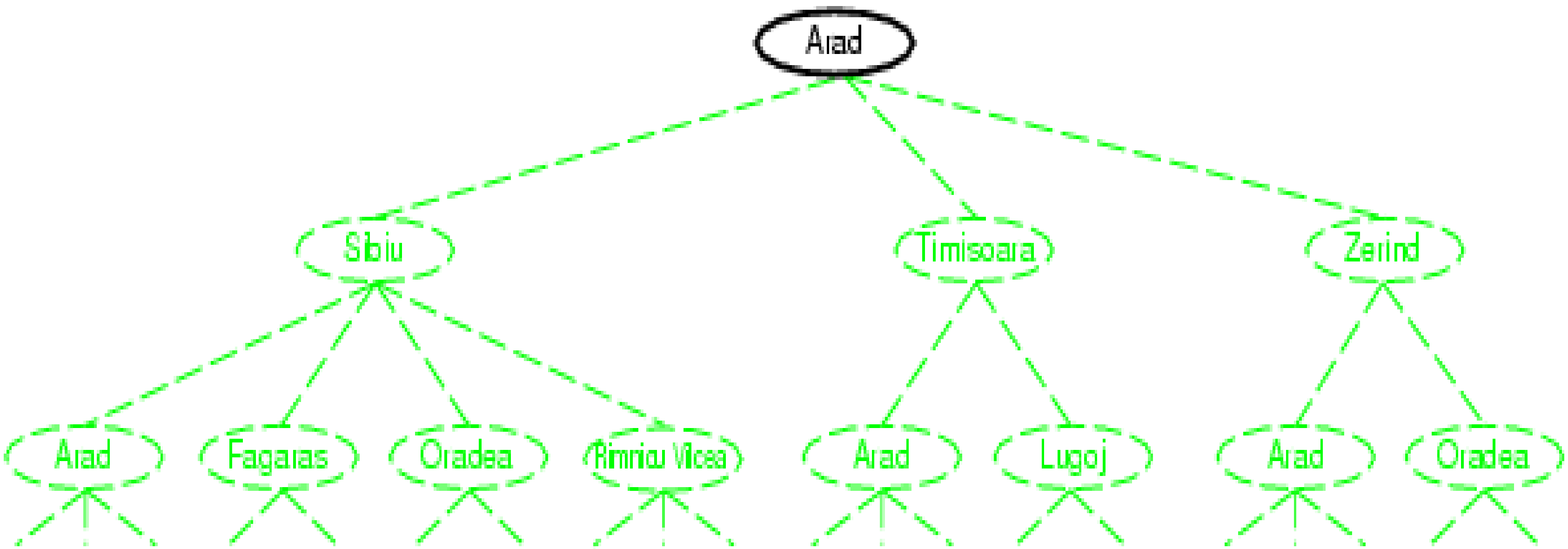
- Exploration of state space by generating successors of already-explored states

```
function TREE-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
```

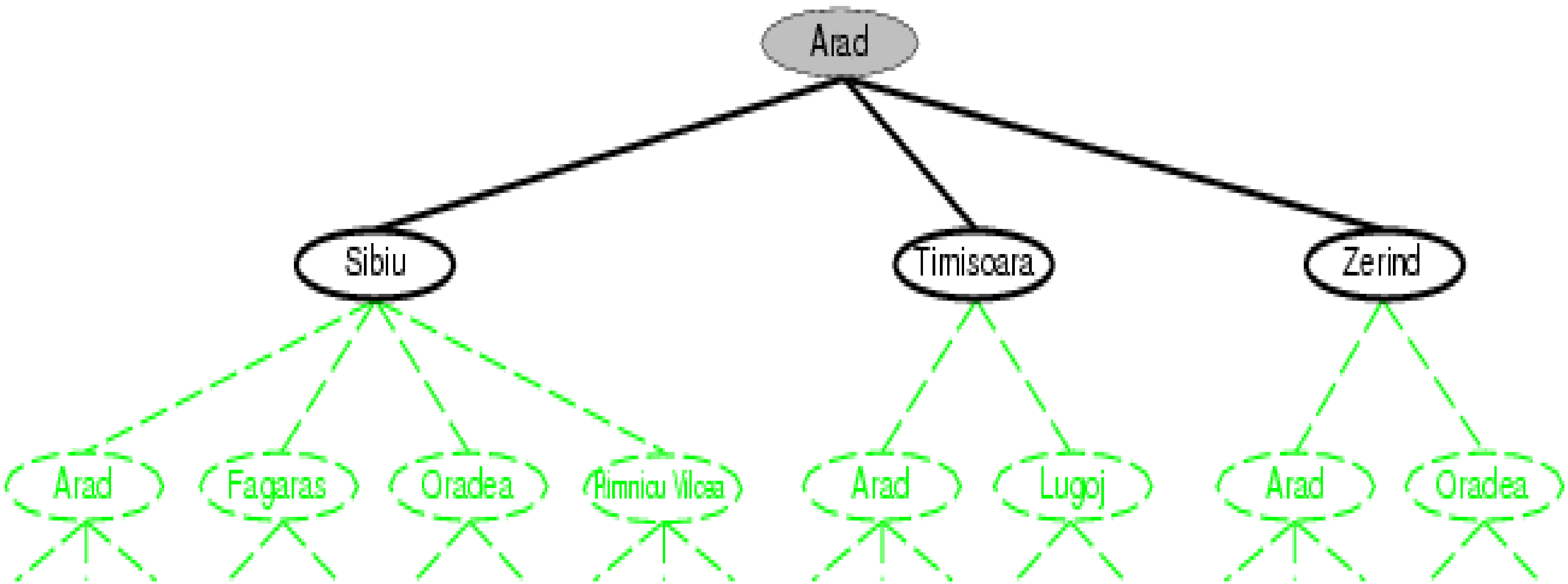
Tree Search: Example



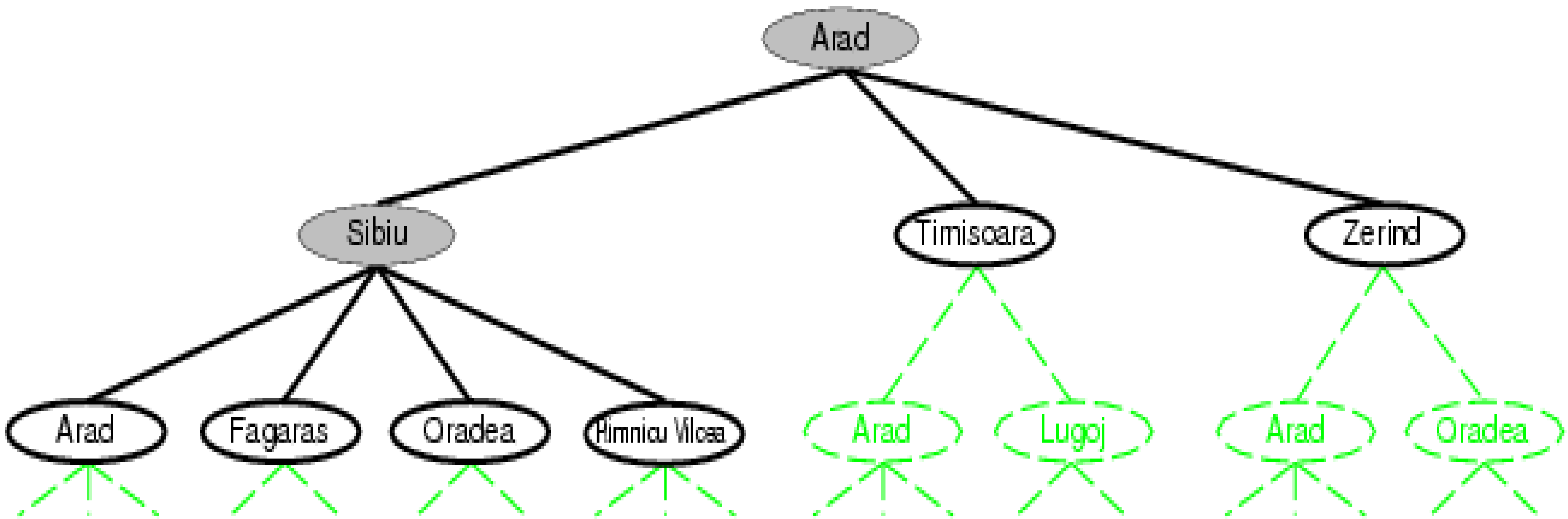
Tree Search: Example



Tree Search: Example



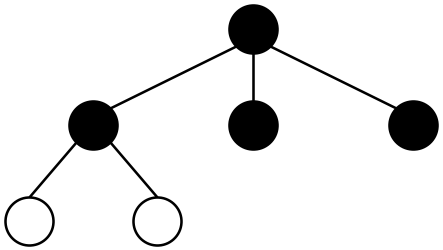
Tree Search: Example



Search Strategies: Blind Search

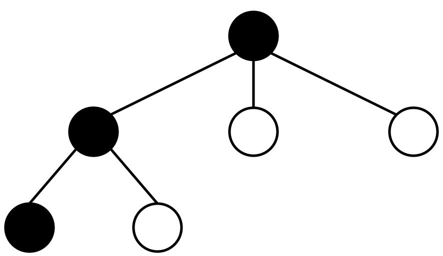
- Breadth-first search

  Expand all the nodes of one level first.



- Depth-first search

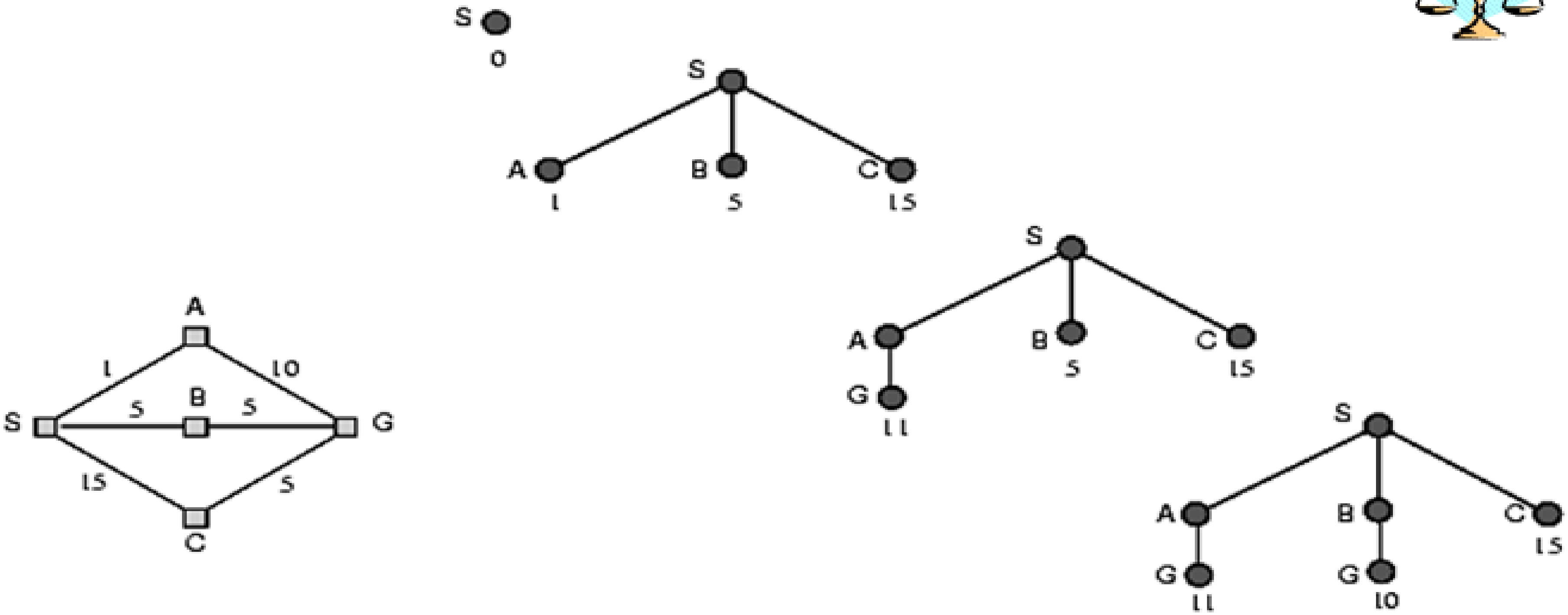
  Expand one of the nodes at the deepest level.



Uniform Cost Search

- BFS finds the shallowest goal state.
- Uniform cost search modifies the BFS by **expanding ONLY the lowest cost node** (as measured by the path cost *g(n)*)
- The **cost of a path** must **never decrease** as we traverse the path, ie. no negative cost should in the problem domain

BS2. Uniform Cost Search (cont)



(a) (b)

- A route finding problem. **(a)** The state space, showing the cost for each operator. **(b)** Progression of the search. Each node is labeled with a numeric path cost *g(n)*. At the final step, the goal node with g=10 is selected

Depth-limited search

= depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

- Recursive implementation:

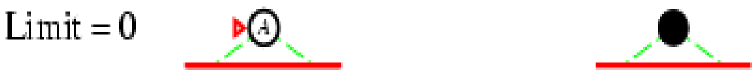
```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

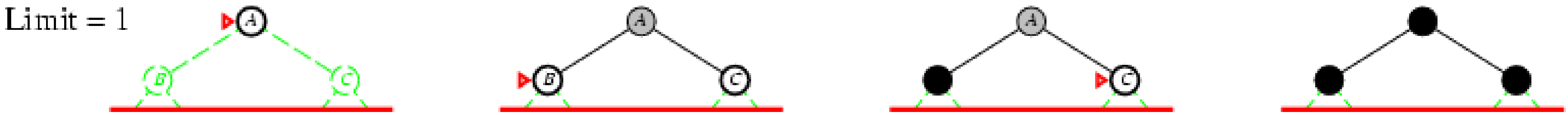
Iterative deepening search

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

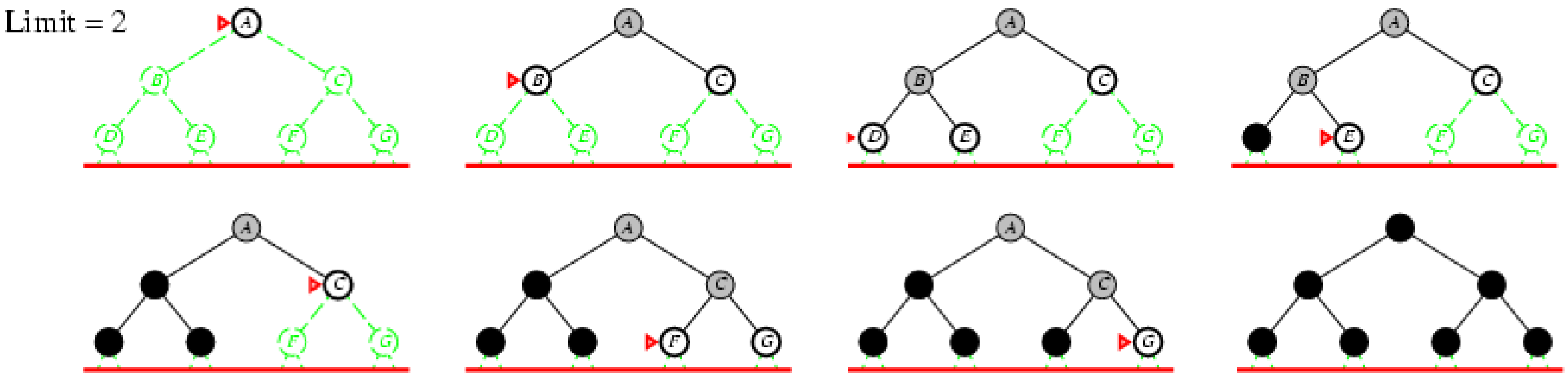
Iterative deepening search *l* =0



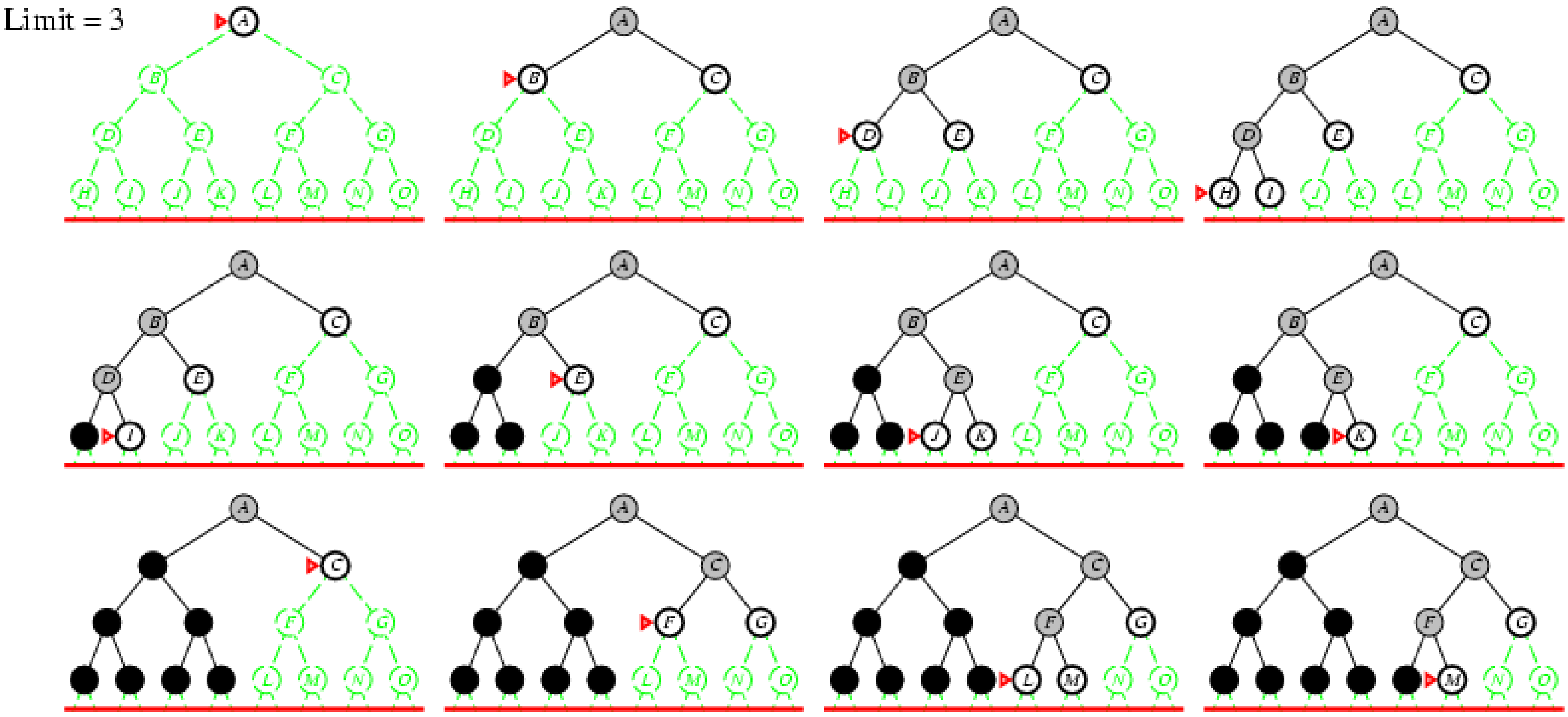
Iterative deepening search *l* =1



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*:

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

- 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