Solving Problems by Uninformed Searching

Tian-Li Yu

Taiwan Evolutionary Intelligence Laboratory (TEIL)
Department of Electrical Engineering
National Taiwan University
tianliyu@ntu.edu.tw

Readings: AIMA Sections 3.1∼3.4

Outline

Problem-Solving Agents

- A simple problem-solving agent formulates a goal and a problem, searches for a sequence of actions that solves the problem, and then execute the actions one by one.

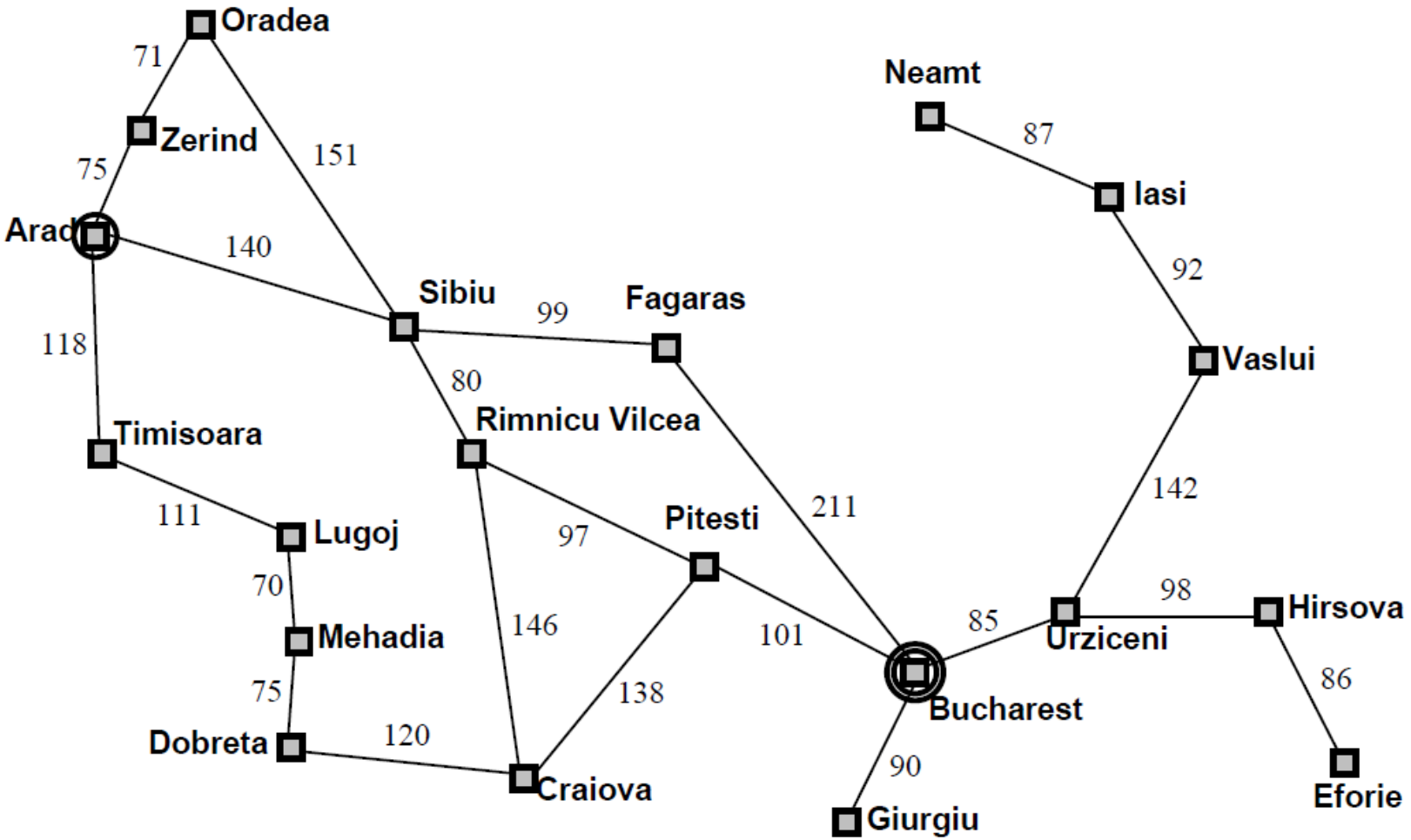
SIMPLE-PROBLEM-SOLVING-AGENT(*percept*)

```
1  state = UPDATE-STATE(state, percept)
2  if seq == empty
3        goal = FORMULATE-GOAL(state)
4        problem = FORMULATE-PROBLEM(state, goal)
5        seq = SEARCH(problem)
6        if seq == failure
7              return NIL
8  action = FIRST(seq)
9  seq = REST(seq)
10 return action
```

Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest.
- Formulate goal
  - Be in Bucharest.
- Formulate problem
  - States: various cities
  - Actions: fly between cities
- Find solution
  - Sequence of cities, *e.g.*, Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem Formulation

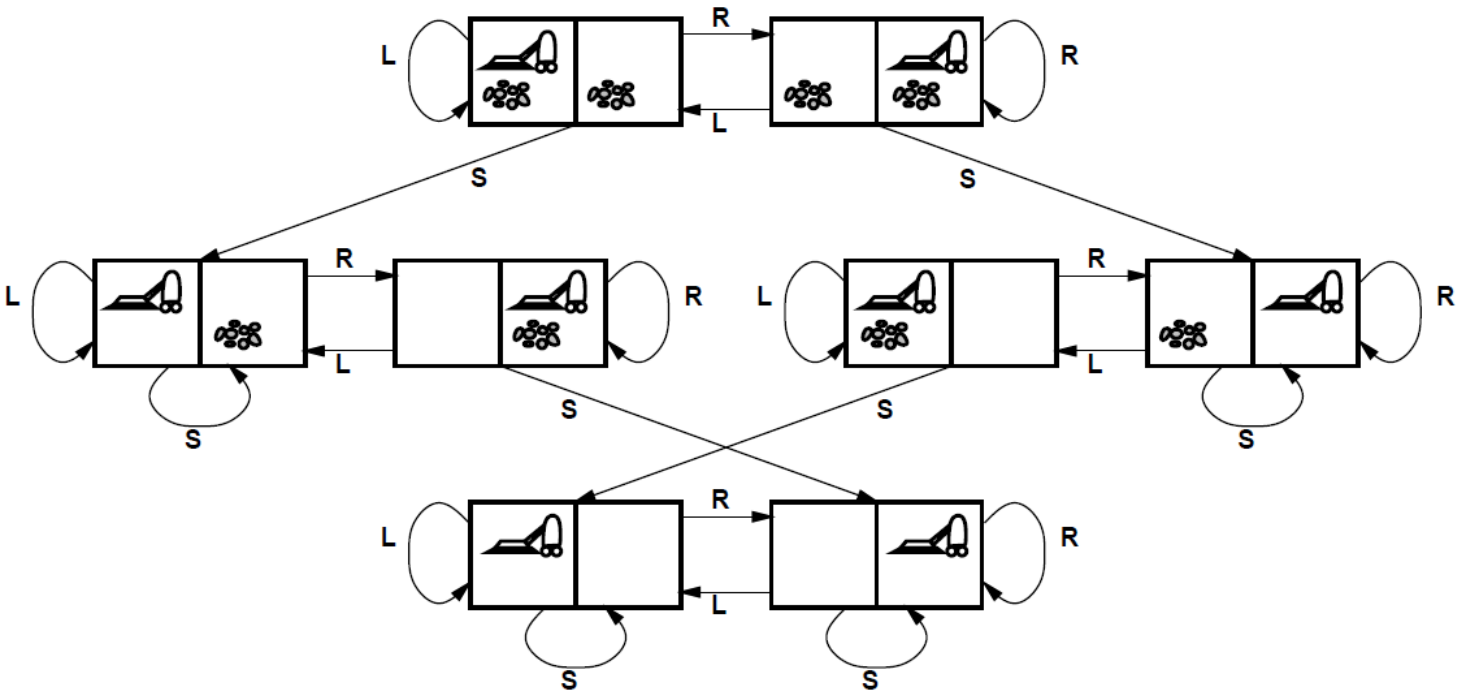
- A problem is defined by five components
  1. Initial state: $In(Arad)$
  2. Actions:
     $\textsc{Action}(In(Arad)) = \{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
  3. Transition model $\textsc{Result}(s, a)$:
     - $\textsc{Result}(In(Arad), Go(Zerind))$=$In(Zerind)$.

     Successor $\mathrm{S}(s)$: states reachable by a single action.
     - $\mathrm{S}(s) = \{s' | \forall a \in \textsc{Action}(s), s' = \textsc{Result}(s, a)\}$
  4. Goal test: $\{In(Bucharest)\}$
  5. Path cost (additive)
     - Sum of distances, number of actions executed, etc.
     - $c(s, a, s')$ is the step cost of taking action a in state s to reach state s', assumed to be ≥ 0
- A solution is a sequence of actions leading from the initial state to a goal state.

Abstraction

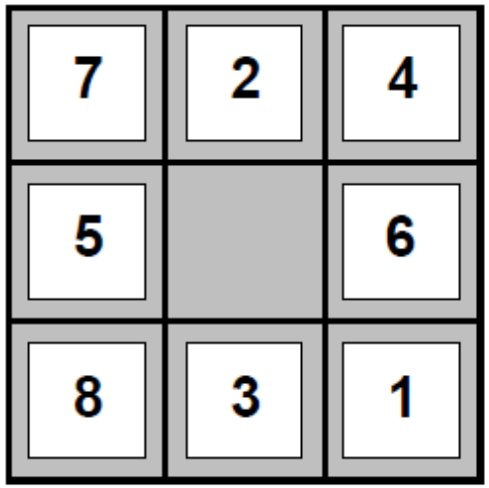
- Real world is absurdly complex
  - State space must be abstracted for problem solving.
- (Abstract) state = subset of real states
- (Abstract) action = complex combination of real actions
  - Go(Zerind) represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (Abstract) solution = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem!

Example: Vacuum World State Space Graph

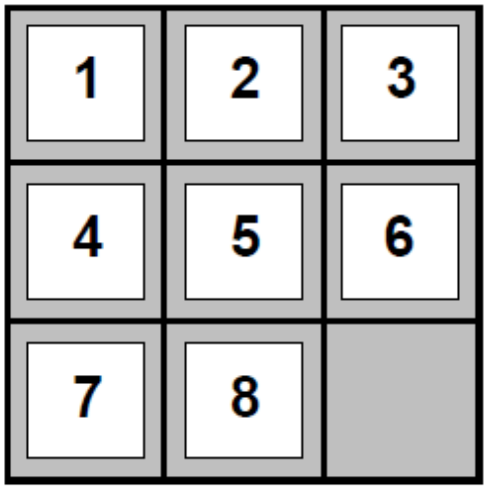


1. Initial state: Any one of the above states. (ignore dirt amounts etc.)
2. Actions: Left, Right, Suck, NoOp
3. Transition model: The above figure.
4. Goal test: no dirt
5. Path cost: 1 per action (0 for NoOp)

Example: The 8-puzzle



Start State

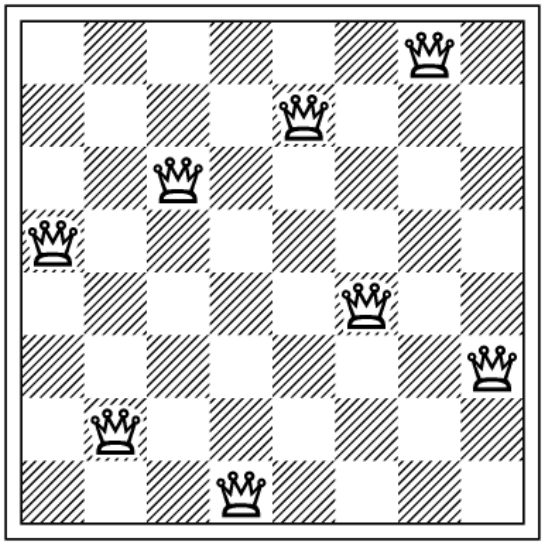


Goal State

1. Initial state: The left figure.
2. Actions: Move blank left, right, up, down.
3. Transition model: Common sense.
4. Goal test: = The right figure.
5. Path cost: One per move.

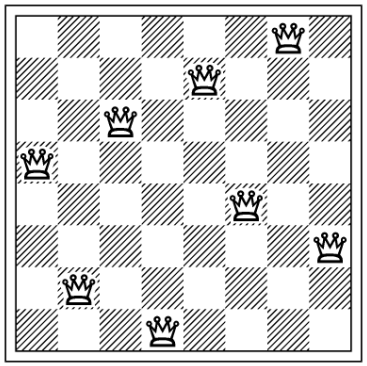
Note: Sliding-block puzzle is $\mathcal{NP}$-hard.

Example: 8-Queen Puzzle



1. Initial state: No queen on the board.
2. Actions: Add a queen on the board where the square is empty.
3. Transition model: Returns the board with a queen added to the specified square.
4. Goal test: 8 queens are on the board, none attacked.
5. Path cost: Number of trials.

Example: 8-Queen Puzzle



- States: Any 0∼8 queens on the board.
  - State space: $C_0^{64} + C_1^{64} + C_2^{64} + \cdots + C_8^{64} \simeq 5.1 \times 10^9$
  - Solution space: $64 \cdot 63 \cdots 57 \simeq 1.8 \times 10^{14}$
- States: One queen per column.
  - State space: $8^0 + 8^1 + 8^2 + \cdots + 8^8 \simeq 1.9 \times 10^7$
  - Solution space: $8^8 \simeq 1.6 \times 10^7$
- States: All possible arrangements of n $(0 \leq n \leq 8)$ queens at leftmost n columns with on queen attacked.
  Actions: Add a queen to the next column with no queen attacked, or backtrack.
  - State space: 2057.

Tree Search Algorithms

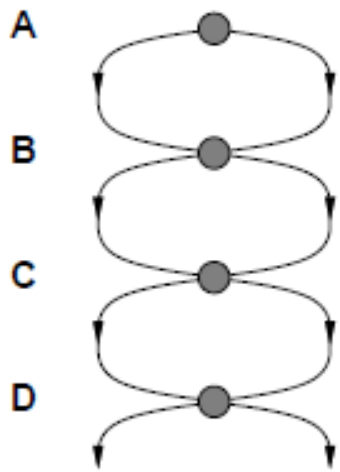
- Offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

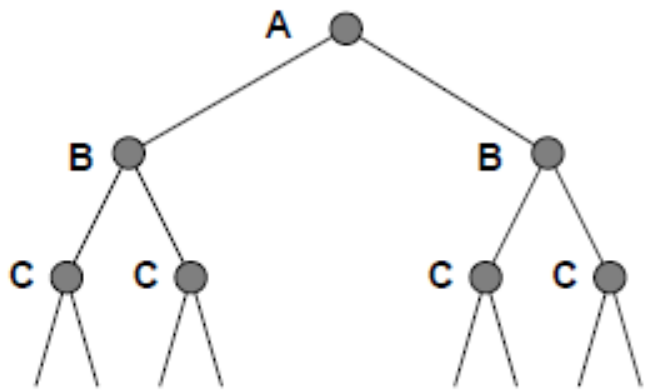
TREE-SEARCH(*problem*)

```
1  initialize the frontier using the initial state of problem
2  repeat
3      if the frontier is empty
4          return failure
5      choose a leaf node and remove it from the frontier.
6      if the node contains a goal state
7          return the corresponding solution
8      expand the chosen node
9      add the resulting nodes to the frontier
```

Repeated States in Graph Search

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Use a queue to record explored states.
- For fast detection of repeated states, hashing techniques are usually adopted.





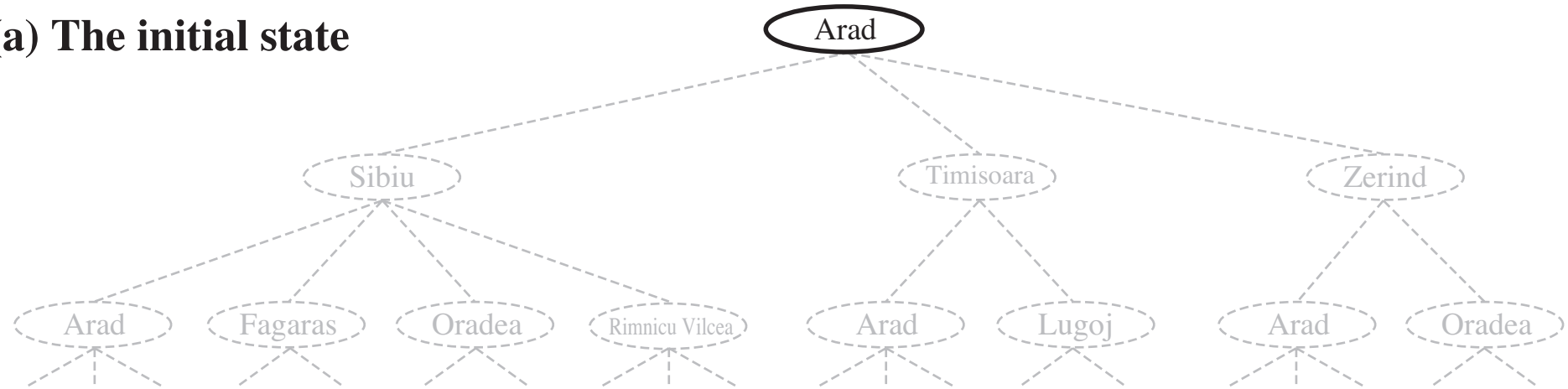
Graph Search Algorithms

GRAPH-SEARCH(*problem*)

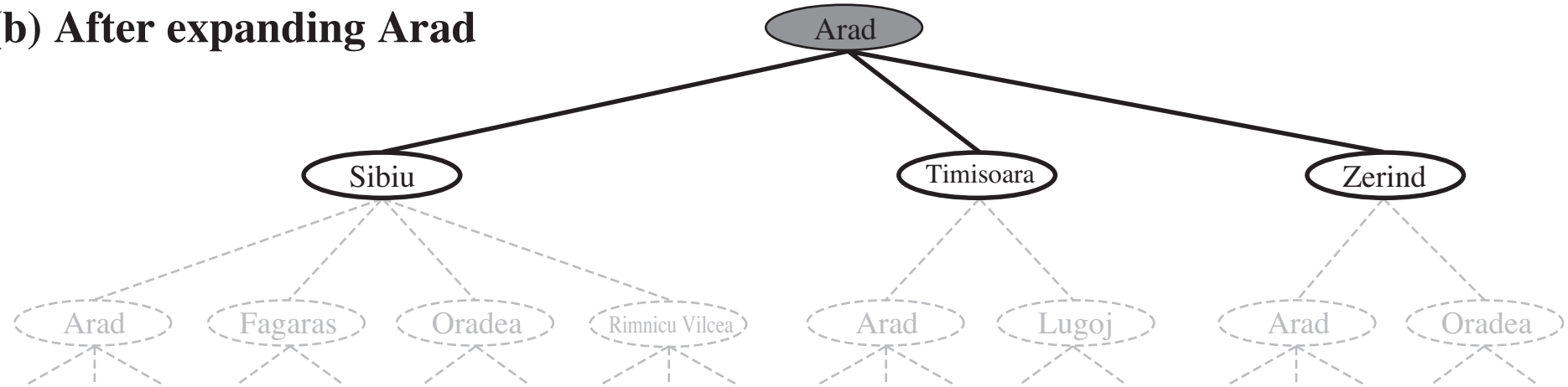
```
1  initialize the frontier using the initial state of problem
2  initialize the explored set to be empty
3  repeat
4      if the frontier is empty
5          return failure
6      choose a leaf node and remove it from the frontier.
7      if the node contains a goal state
8          return the corresponding solution
9      add the node to the explored set
10     expand the chosen node
11     if not in the frontier or explored set
12         add the resulting nodes to the frontier
```

Partial Search Tree

(a) The initial state

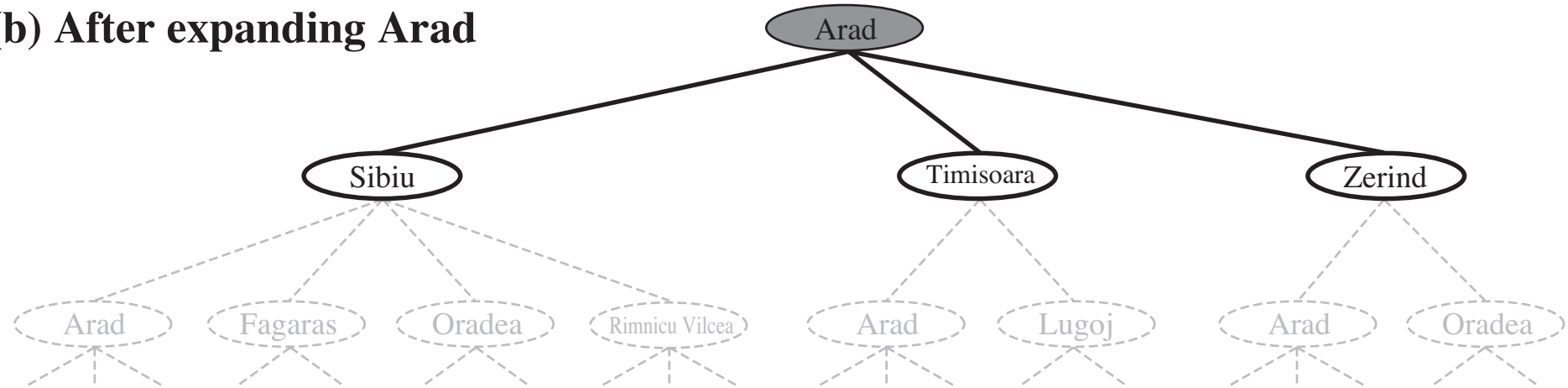


(b) After expanding Arad

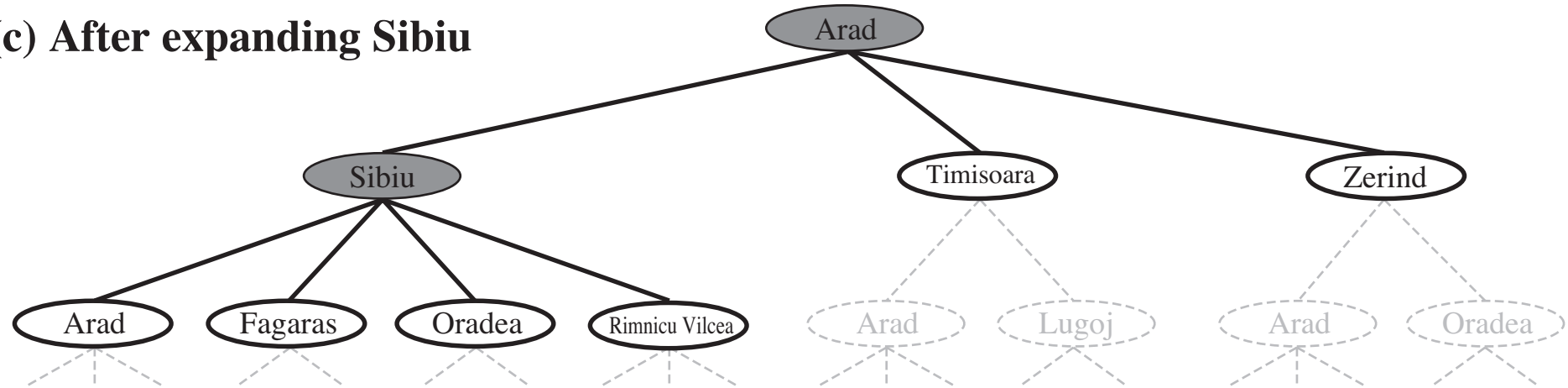


Partial Search Tree

(b) After expanding Arad



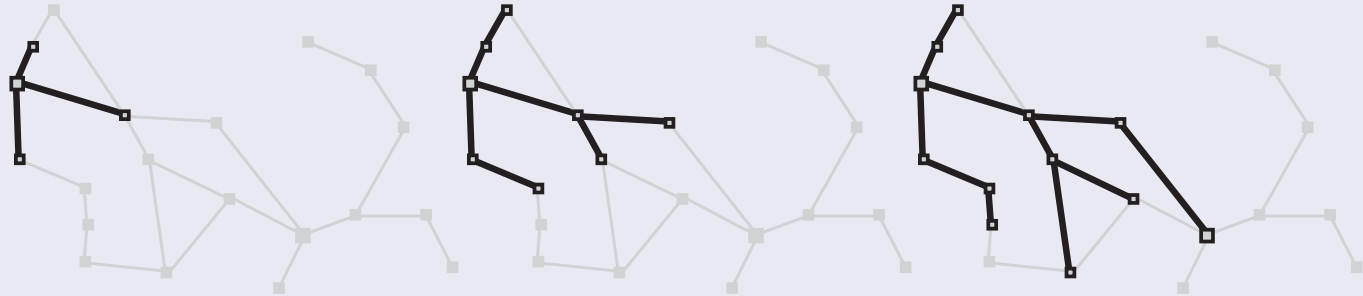
(c) After expanding Sibiu



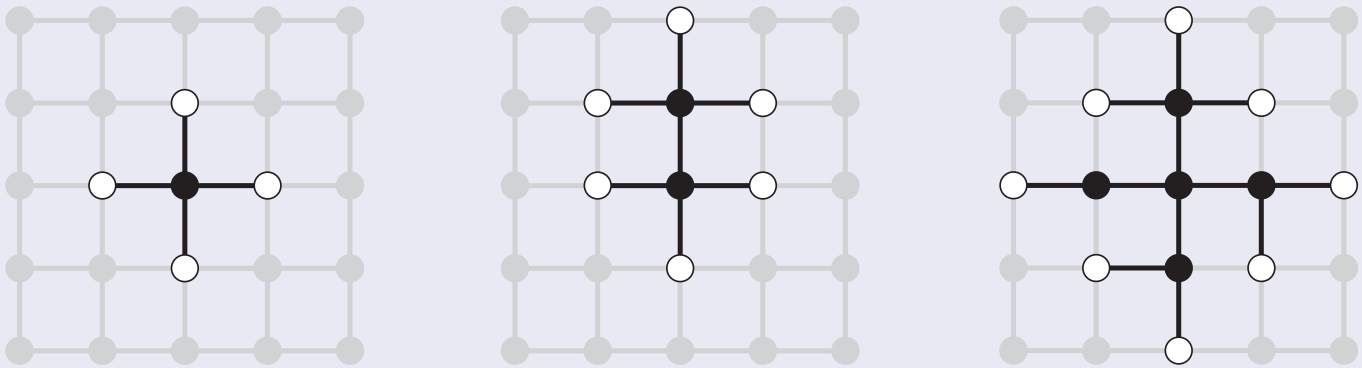
Graph Search, Search Tree, and Frontier Separation

- The frontier separates the state space into explored and unexplored regions (loop invariant proof).

Tree generated by graph search on Romania map

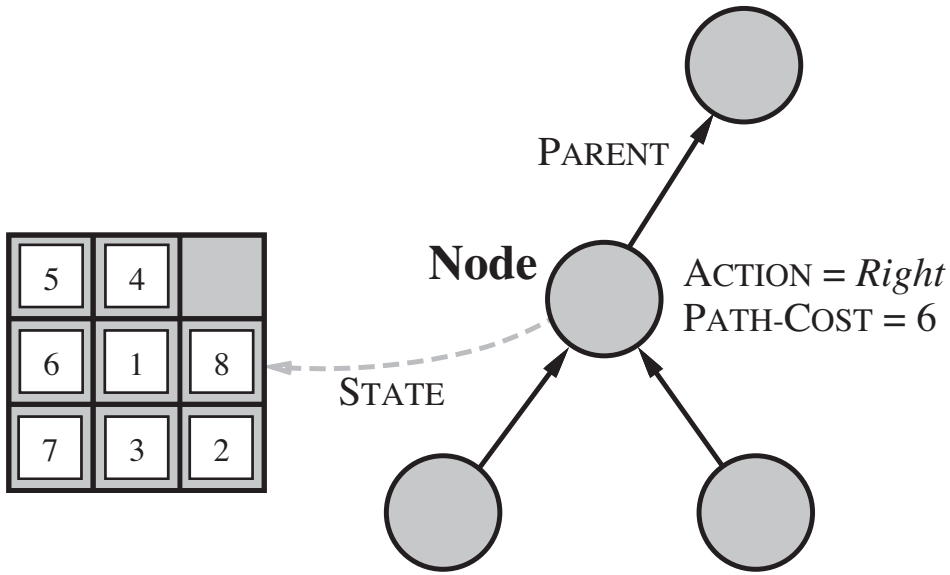


Separation property of GRAPH-SEARCH



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 parent, children, depth, path cost $g(x)$
- States do not have parents, children, depth, or path cost!



Infrastructure for Search Algorithms

CHILD-NODE(*problem*, *parent*, *action*)

return a node *n* with
$n.state = problem.\text{RESULT}(parent.state, action)$
$n.parent = parent$
$n.action = action$
$n.path\_cost = parent.path\_cost + problem.\text{STEP-COST}(parent.state, action)$

- The appropriate data structure to maintain the frontier is a queue.
- Can be
  - FIFO.
  - LIFO (a.k.a. stack)
  - Priority queue

Tree Search Algorithms

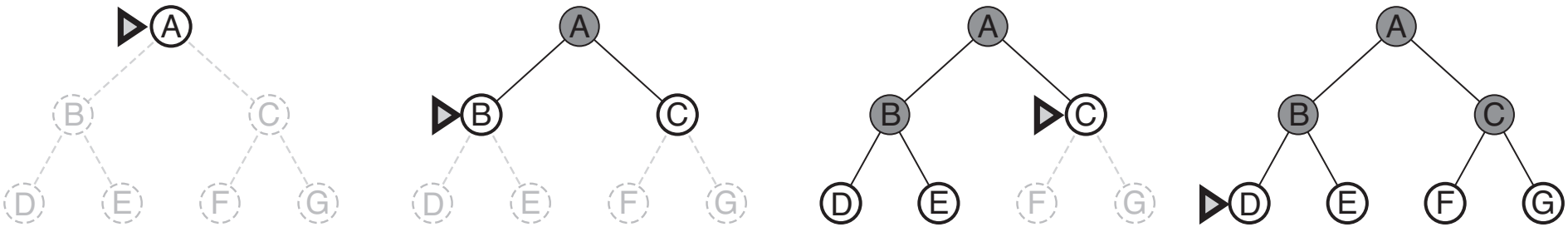
- A strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - Completeness - does it always find a solution if one exists?
  - Optimality - does it always find a least-cost solution?
  - Time complexity - number of nodes generated/expanded
  - 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 ∞)

Uninformed Search Strategies (Blind Search)

- Uninformed strategies use only the information available in the problem definition.
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-First Search (BFS)

- Expand the shallowest unexpanded node.
- FIFO queue.



Properties of BFS

- Completeness: Yes (if b is finite)
- Optimality: No in general; yes when the path cost is a non-decreasing function of the depth of the node.
- Time complexity: $1 + b + b^2 + \cdots + b^d = O(b^d)$.

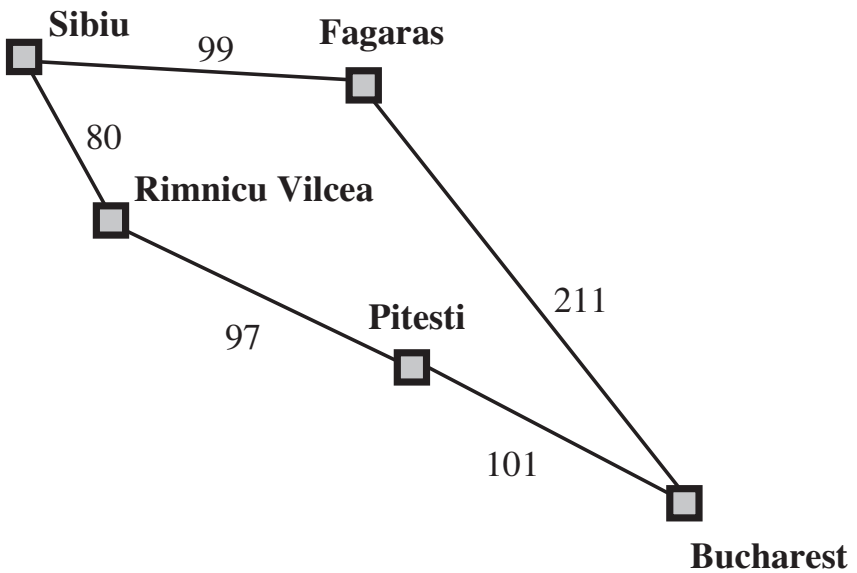
  Or $O(b^{d+1})$ if goal test is applied after expansion.
- Space complexity: $O(b^d)$ (keeps every node in memory)

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Uniform-Cost Search

- Expand the unexpanded node with the lowest path cost.
- Priority queue ordered by $g(n)$.
- Equivalent to BFS if step costs all equal.

- For TREE-SEARCH, priority queue gives the cheapest path first.
- For GRAPH-SEARCH, if the node is already in the frontier, need to find the minimum cost, and call DECREASEKEY as needed.

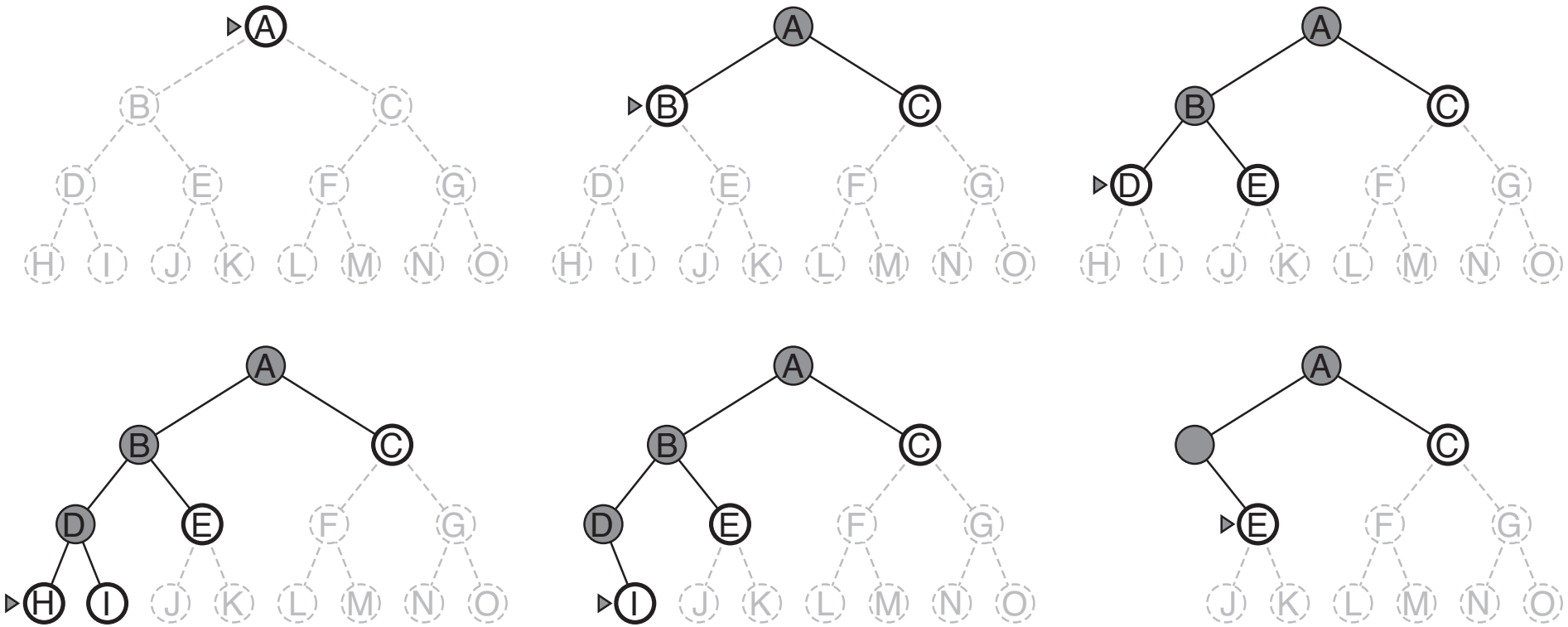


Properties of Uniform-Cost Search

- Completeness: Yes, if step cost $\geq \epsilon > 0$.
- Optimality: Yes - nodes expanded in increasing order of $g(n)$.
- Time complexity: # of nodes with $g \leq$ cost of optimal solution. Maximum depth is given by $1 + \lfloor C^*/\epsilon \rfloor$, where C^* is the cost of the optimal solution.
  $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.
- Space complexity: # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

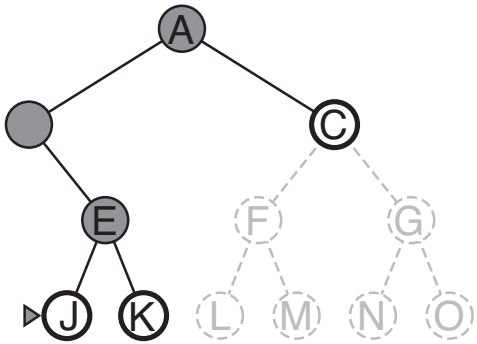
Depth-First Search (DFS)

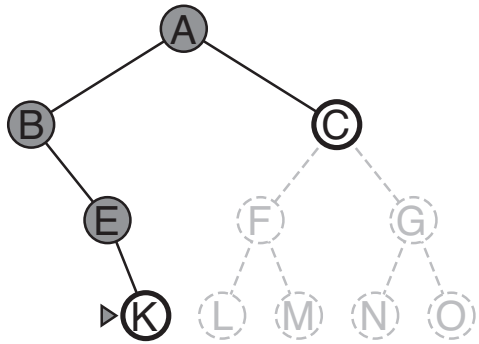
- Expand the deepest unexpanded node.
- LIFO queue, *i.e.*, put successors at front.

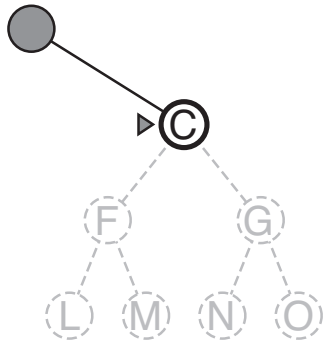


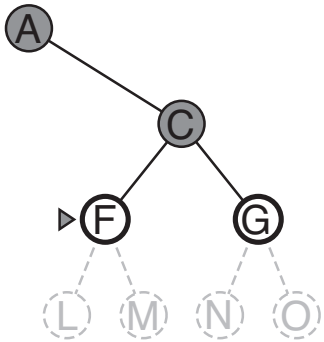
Depth-First Search (DFS)

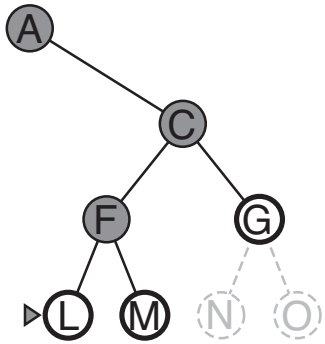
- Expand the deepest unexpanded node.
- LIFO queue, *i.e.*, put successors at front.

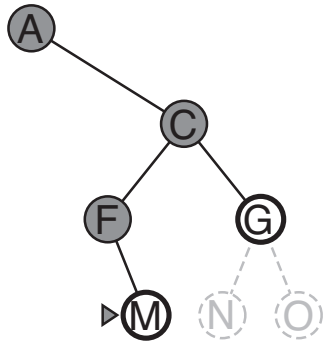












Properties of DFS

- Completeness: No, fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path $\rightarrow$ complete in finite spaces.
- Optimality: No.
- Time complexity: $O(b^m)$, terrible if m is much greater than d.
  - But if solutions are dense, may be much faster than breadth-first
- Space complexity: $O(bm)$, linear space!
  - Backtracking technique only generate one successor instead of all successors $\rightarrow O(m)$.

Depth-Limited Search (DLS)

- DFS never terminates if $m \to \infty$.
- DLS = DFS with depth limit ℓ,
- Nodes at depth ℓ have no successors
- Recursive implementation:

DEPTH-LIMITED-SEARCH(*problem*, *limit*)

return RECURSIVE-DLS(MAKE-NODE(*problem.initial_state*), *problem*, *limit*)

Depth-Limited Search (DLS)

RECURSIVE-DLS(*node*, *problem*, *limit*)

```
1  if problem.GOAL-TEST(node.state)
2      return SOLUTION(node)
3  elseif limit == 0
4      return cutoff
5  else
6      cutoff_occurred = FALSE
7      for each action in problem.ACTIONS(node.state)
8          child = CHILD-NODE(problem, node, action)
9          result = RECURSIVE-DLS(child, problem, limit − 1)
10         if result == cutoff
11             cutoff_occurred = TRUE
12         elseif result ≠ failure
13             return result
14     if cutoff_occurred
15         return cutoff
16     else
17         return failure
```

Properties of DLS

- Completeness: Not complete if $\ell < d$; complete otherwise.
- Optimality: Not optimal in general (even if $\ell > d$).
- Time complexity: $O(b^{\ell})$
- Space complexity: $O(b\ell)$, linear space.
- Two termination conditions:
  - *failure*: no solution.
  - *cutoff*: no solution within the depth limit.

Iterative-Deepening Search (IDS)

- Call DLS iteratively with increasing depth limit.
- Seems to be wasteful, but actually not.
- Combine the benefits of BFS and DFS.

ITERATIVE-DEEPENING-SEARCH(*problem*)

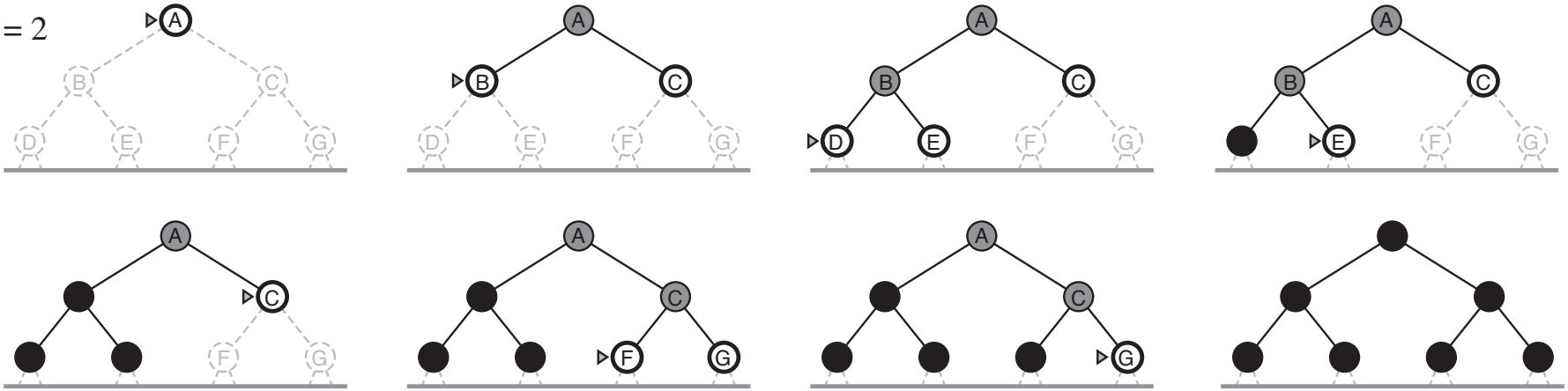
```
1  for depth = 0 to ∞
2      result = DEPTH-LIMITED-SEARCH(problem, depth)
3      if result ≠ cutoff
4          return result
```

Iterative Deepening Search

Limit = 0

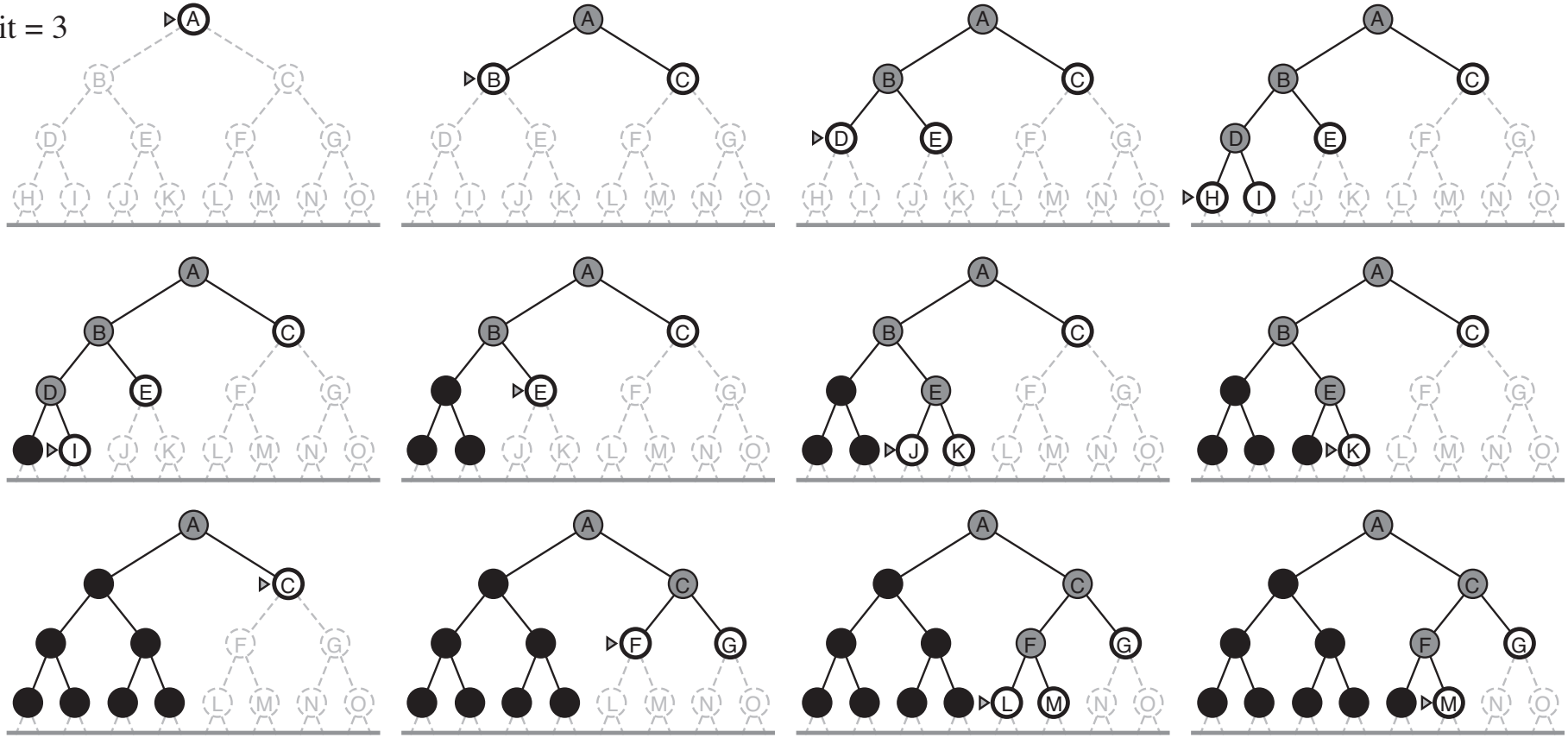
Limit = 1

Limit = 2



Iterative Deepening Search

Limit = 3



Properties of Iterative Deepening Search

- Completeness: Yes
- Optimality: No in general; yes when the path cost is a non-decreasing function of the depth of the node.
  - Can be modified to explore uniform-cost trees (optimal).
- Time complexity: $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space complexity: $O(bd)$
- Numerical comparison for $b = 10$, $d = 5$, solution at far right leaf.
  - N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
  - N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,100
- Repeated search in IDS is not severe.
- In general, IDS is preferred when search space is large and depth is unknown.
- We'll talk about more advantages of IDS in adversarial search.

Summary of Algorithms

Criterion	BFS	Uniform-Cost	DFS	DLS	IDS
Completeness	Yes[a]	Yes[b]	No	No[c]	Yes[a]
Optimality	No[d]	Yes	No	No	No[e]
Time Complexity	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$

[a] if b is finite
[b] if b is finite and step cost $\geq \epsilon$
[c] unless $\ell \geq d$
[d] unless the path cost is a non-decreasing function of the depth of the node

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
  - Initial state.
  - Actions.
  - Transition model.
  - Goal test.
  - Path cost.
- Graph search can be exponentially more efficient than tree search, but usually impractical due to memory requirement.
- Variety of uninformed search strategies judged on the basis of
  - Completeness
  - Optimality
  - Time and space complexity.
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.