Artificial Intelligence

Information Systems and Machine Learning Lab (ISMLL)
Tomáš Horváth

3rd November, 2011

Solving Problems by Searching

Problem-solving agent

- decides what to do by finding sequences of actions leading to desirable states - <u>goals</u>
  - I. Goal formulation
    - What do we want to “reach”?
  - II. Problem formulation
    - What actions and states to “consider”, given a goal?
    - abstraction
      - the <u>level</u> of states and actions

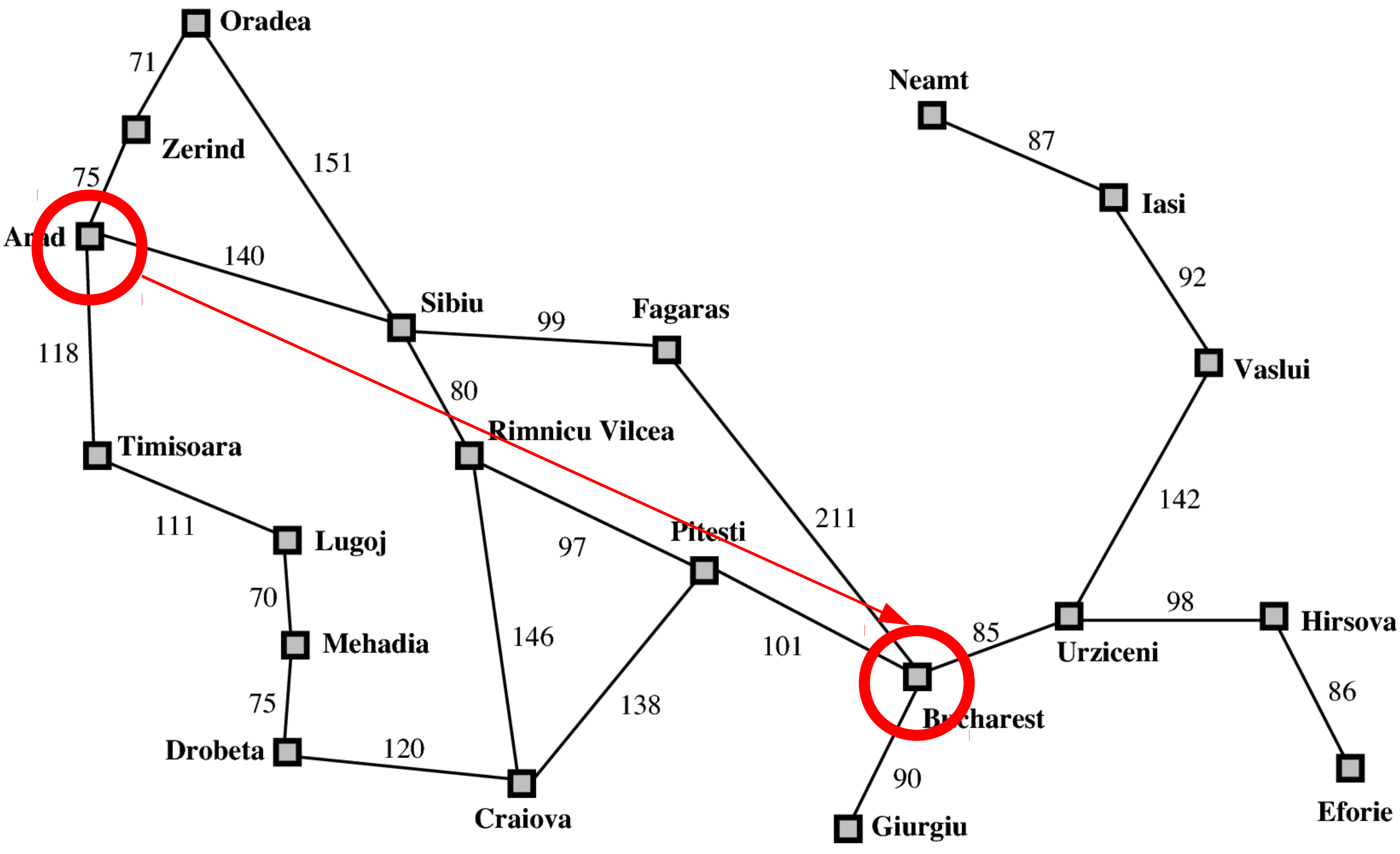
- Looking for such sequences is called SEARCH

Problem-solving agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

- what type of an environment is this agent working in?
  - static? observable? discrete? deterministic?

Example



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

Problem definition

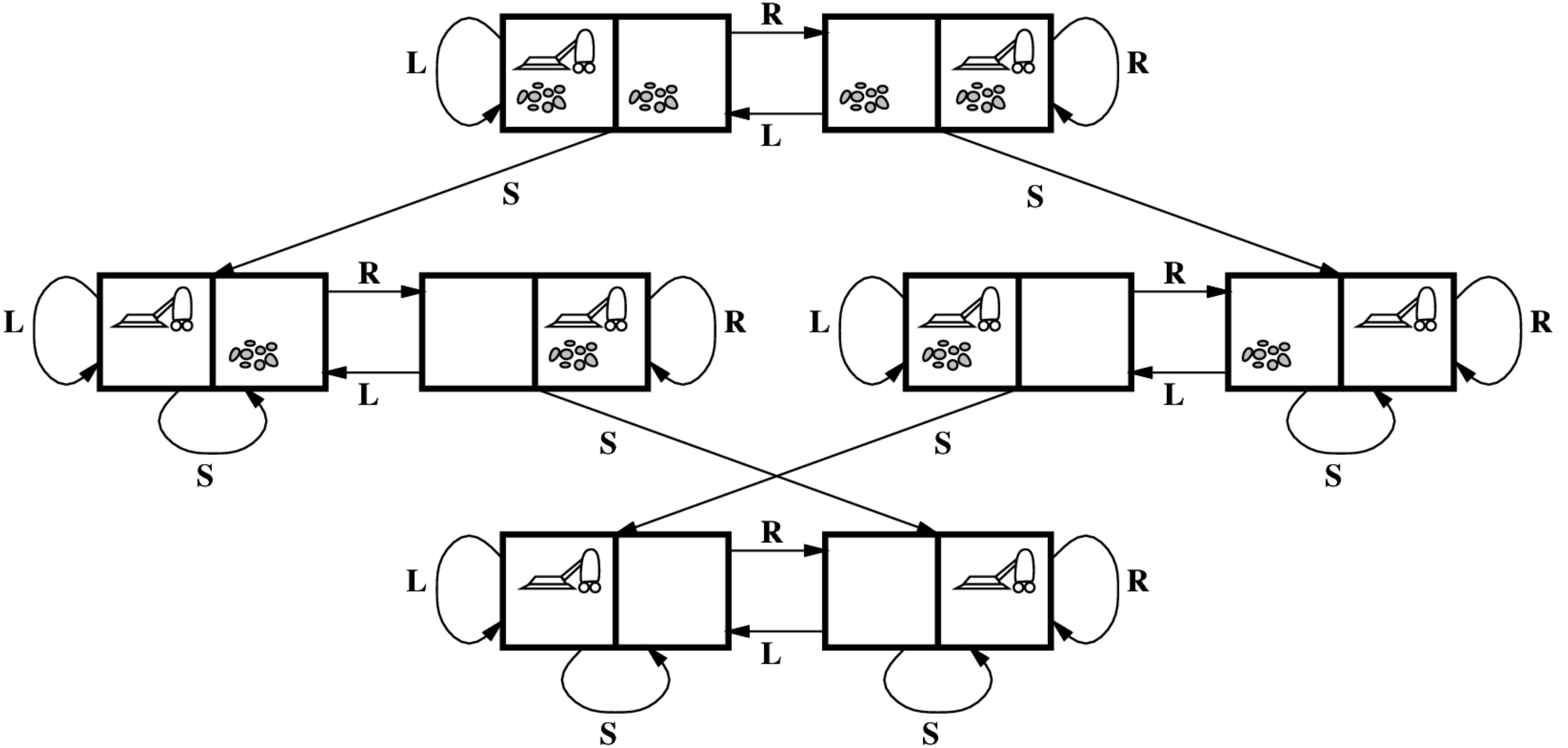
- initial state
- successor function
  - possible actions available from the current state (operators applied to a current state)
- goal test
  - determines whether a given state is a goal state
- path cost
  - assigns a number to each path
  - step costs
- solution
  - a path from the initial state to a goal state

(initial state + successor function) → define a **state space**

Toy examples

- vacuum cleaner
  - states
    - 8 possible world states
      - two locations which could be clean or dirty
  - any state can be initial
  - successor function generates the legal states resulting from trying the three possible actions
    - Left, Right, Suck
  - goal test checks whether all the squares are clean
  - path cost is the number of step
    - the number of steps, each step costs 1

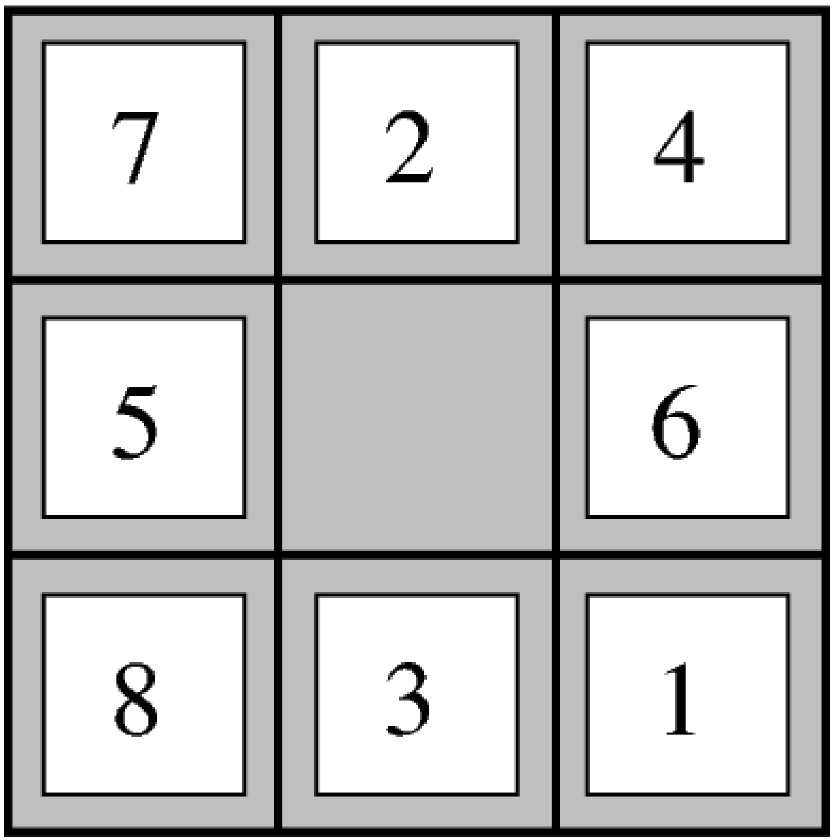
Toy examples



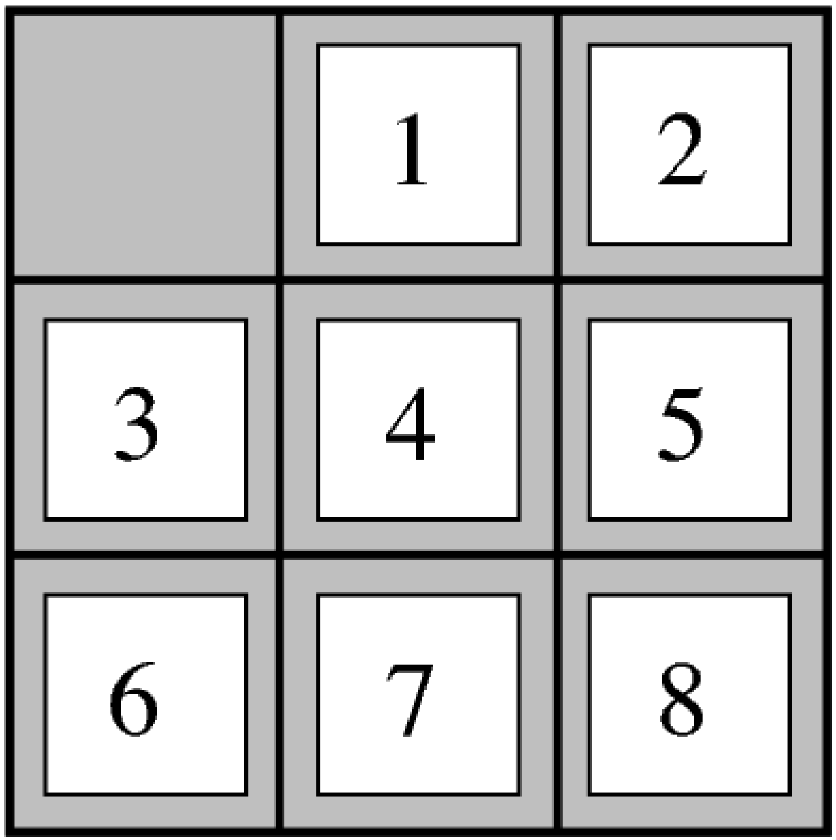
Toy example

- 8 puzzle
  - states
    - the location of each of the 8 tiles and the blank
  - initial state
    - any state
  - successor function
    - Left, Right, Up or Down
  - goal test
    - tests whether the state matches the goal configuration
  - path cost
    - the number of steps, each step costs 1

Toy example



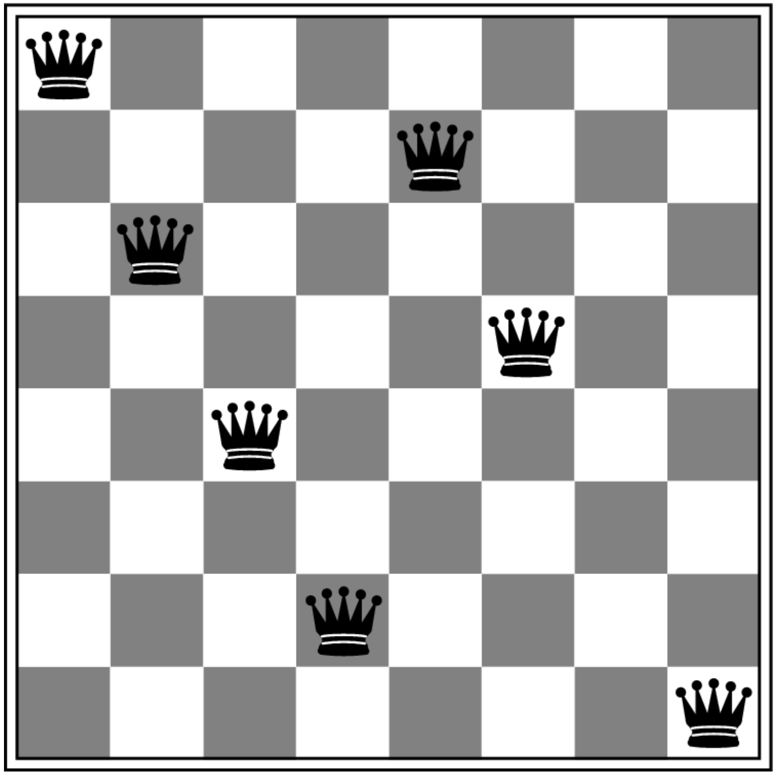
Start State



Goal State

Toy example

- 8-queens problem
  - **How the specification looks like?**



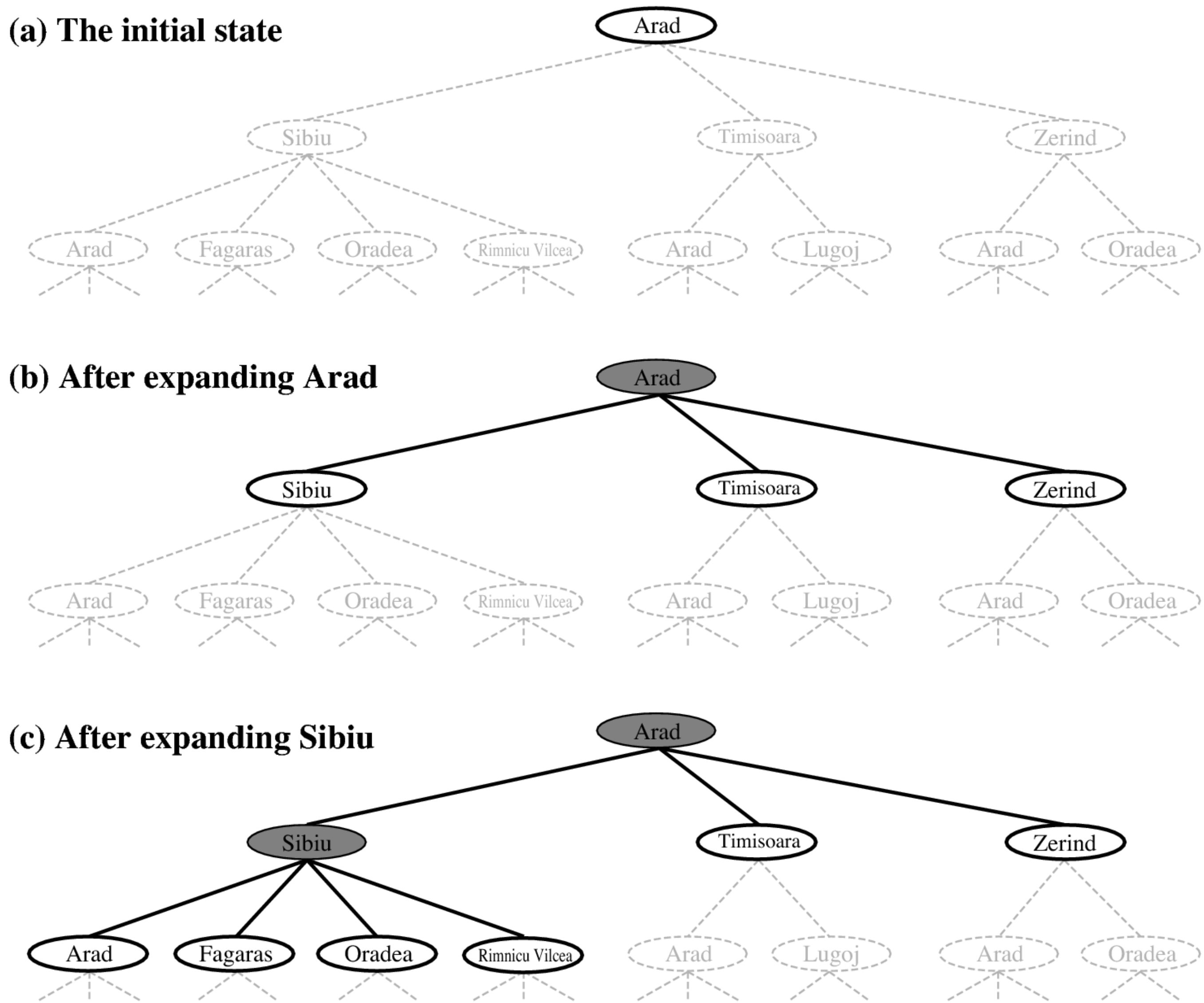
Real-world problems

- route finding, touring, traveling problems
  - get from the location A to the location B
  - visit every city at least once
  - visit every city at least once
- VLSI layout problem
  - positioning millions of components and connections on a chip to minimize area
- Internet searching problem
  - looking for related information by going through the links on the web sites seen

Searching

- search tree, search graph
  - generated by the initial state and the successor function
  - search node
    - an instantiation of a world state
    - main components are
      - state (in the state space to which the node corresponds)
      - parent node
      - action (which was applied to the parent to generate the node)
      - path-cost
      - depth (the number of steps along the path from the initial state)
- search strategy
  - the choice of which state to expand
  - <u>fringe</u>
    - the collection of nodes that have been generated but not yet expanded

Search tree/graph



Search

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
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, 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
    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

Measuring the performance

- completeness
  - Is the algorithm guaranteed to find a solution when there is one?
- optimality
  - Does the strategy find the optimal solution?
- time complexity
  - how long does it take to find a solution?
- space complexity
  - how much memory is needed to perform the search?

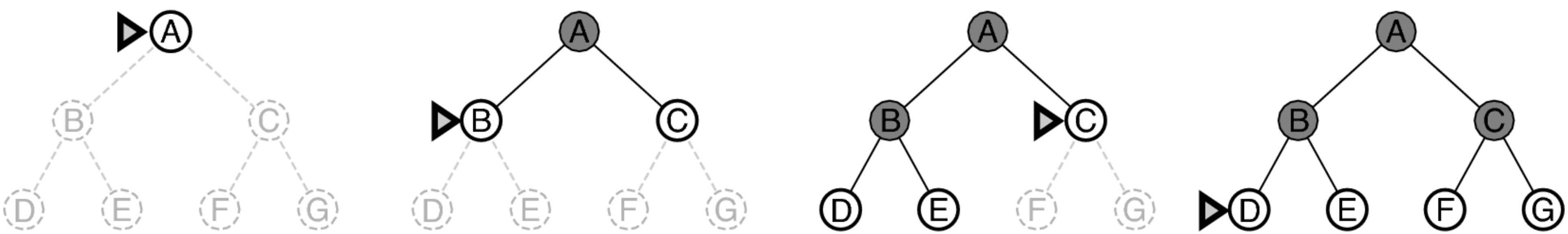
- important factors
  - branching factor (b)
  - depth of the shallowest node (d)
  - the maximum length of any path in the state space (m)

Breadth-first search

```
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)
```

Breadth-first search

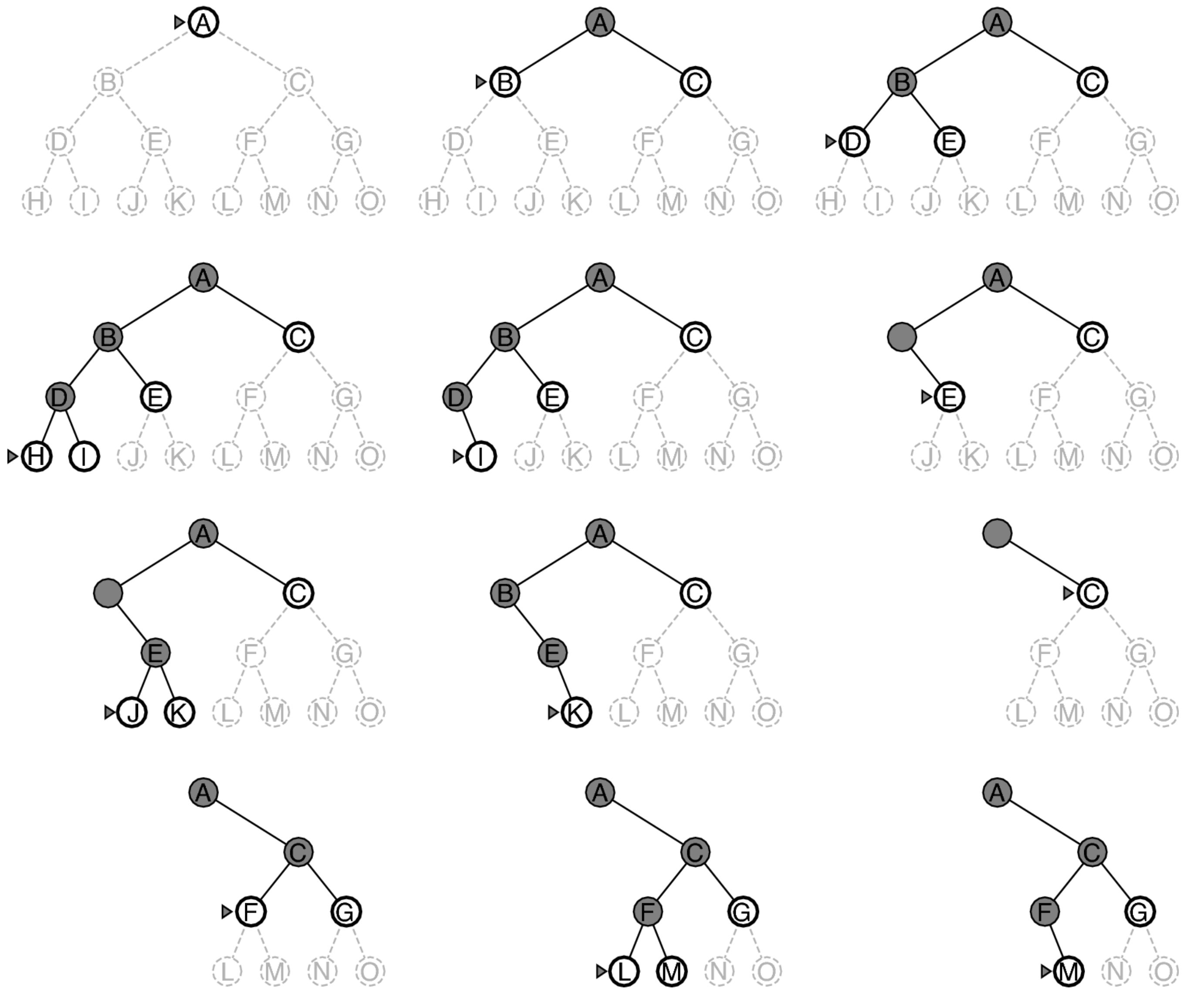
- **what is the total number of nodes generated?**
  - suppose that the solution is at depth d and
  - each node generates b more nodes...



Uniform-cost search

```
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
```

Depth-first search



Depth-first search

- similar to breadth-first search
  - using LIFO

- Backtracking search
  - a variant of depth-first search
  - only one successor is generated at a time
    - nodes should remember which successor to generate next

- **what is the drawback of depth-first search?**

Depth-limited search

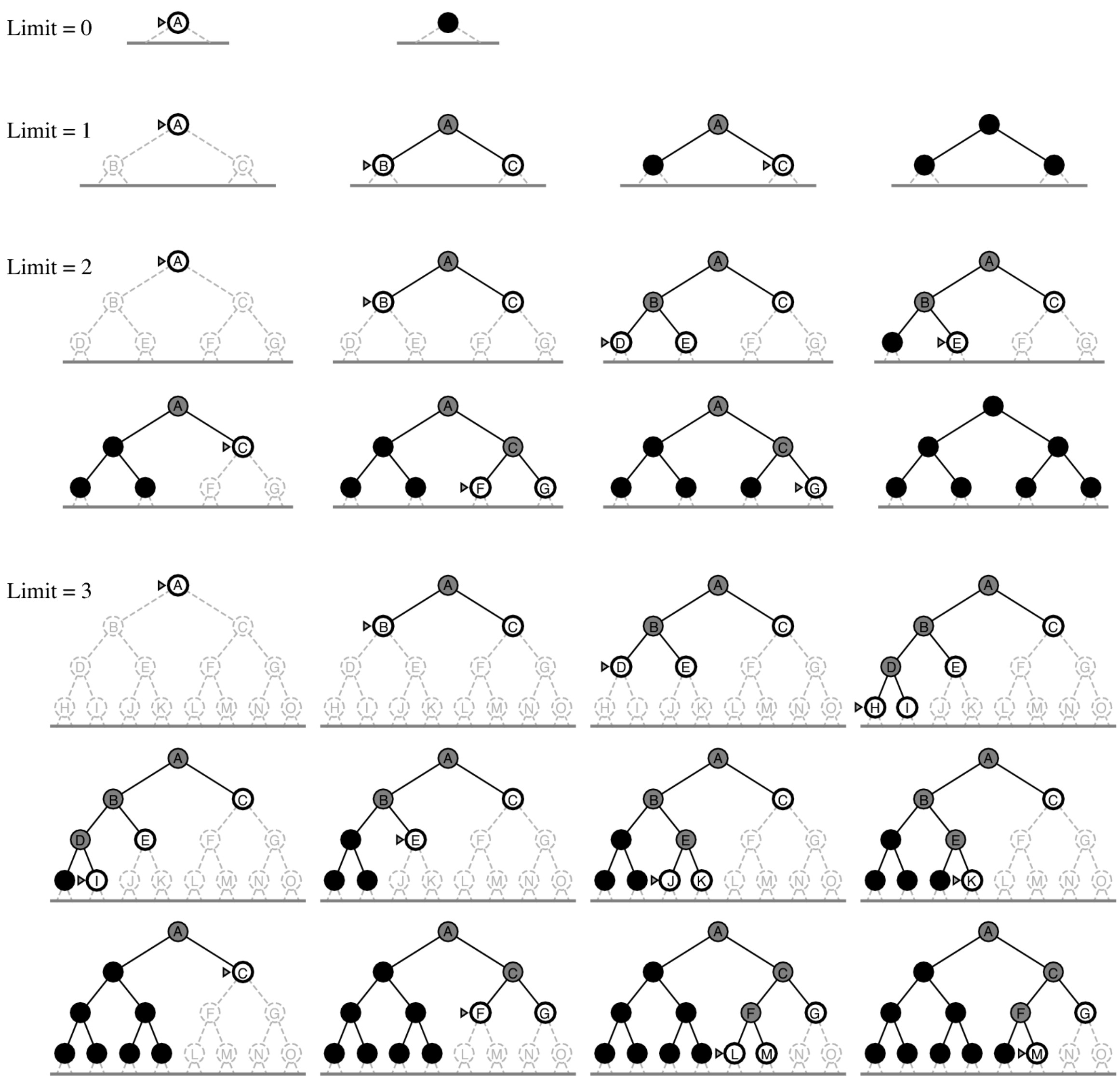
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0 then return cutoff
   else
       cutoff_occurred? ← false
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           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 depth-first search

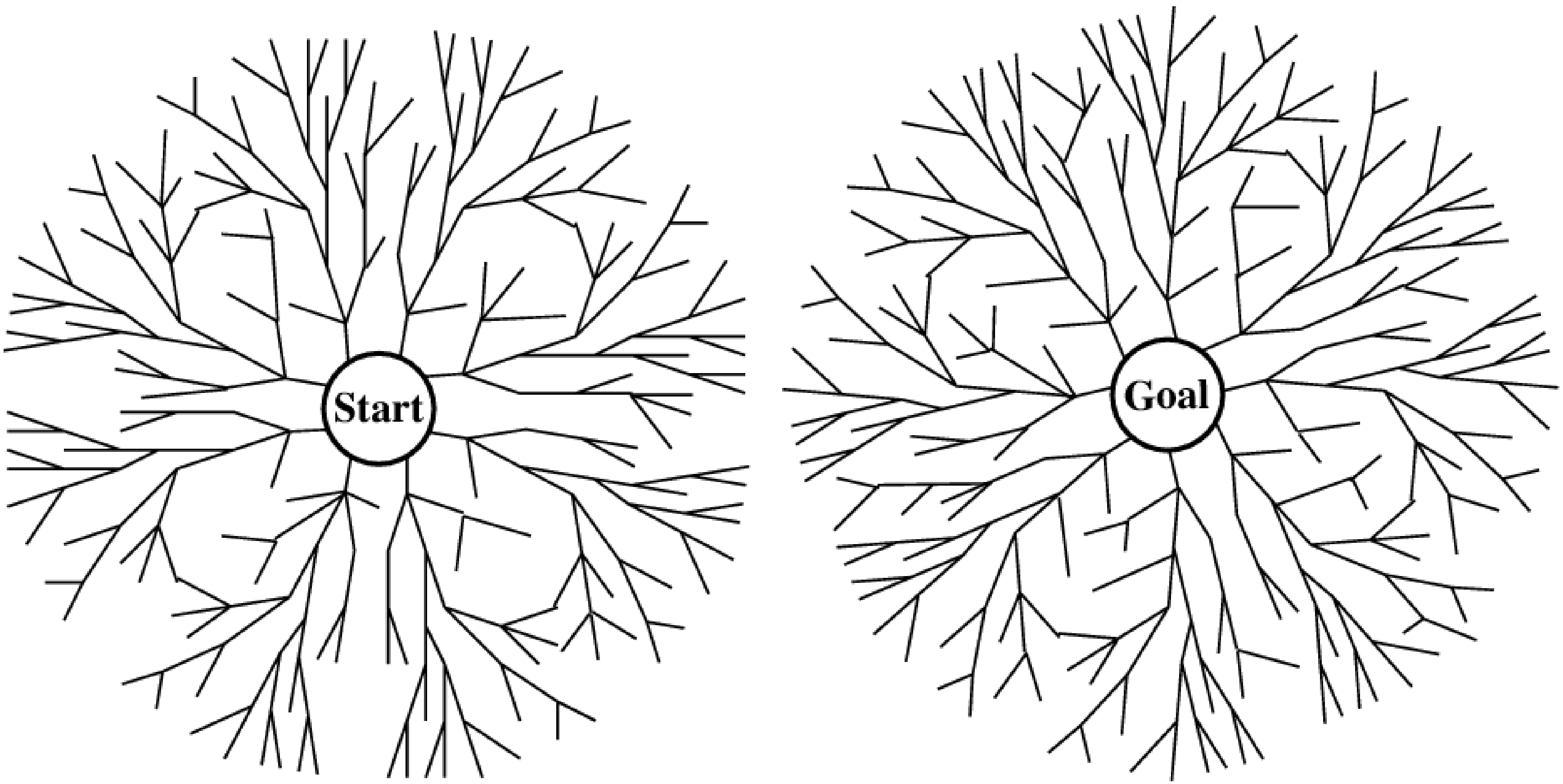
function ITERATIVE-DEEPENING-SEARCH($problem$) **returns** a solution, or failure
  for $depth = 0$ **to** ∞ **do**
    $result \leftarrow$ DEPTH-LIMITED-SEARCH($problem$, $depth$)
    if $result \neq$ cutoff **then return** $result$

- in general, iterative deepening is the preferred uninformed search method when there is a large space and the depth of the solution is not known
  - **why?**
  - **what is the total number of nodes generated?**

Iterative deepening depth-first search



Bidirectional search

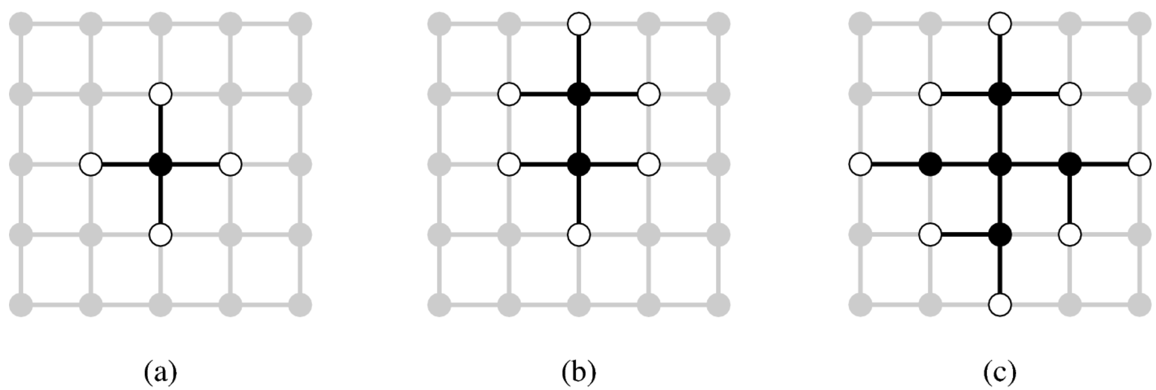


Comparison

Method	Completeness	Time complexity	Space complexity	Optimality
Breadth-first	yes[a]	$O(b^{d+1})$	$O(b^{d+1})$	yes[c]
Uniform-Cost	yes[a,b]	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	yes
Depth-first	no	$O(b^m)$	$O(bm)$	no
Depth-limited	no	$O(b^l)$	$O(bl)$	no
Iterative-deepening	yes[a]	$O(b^d)$	$O(bd)$	yes[c]
Bi-directional (if applicable)	yes[a,d]	$O(b^{d/2})$	$O(b^{d/2})$	yes[c,d]

[a] complete if b is finite
[b] complete if step costs $\geq \epsilon$ for positive ϵ
[c] optimal if step costs are all identical
[d] both directions use breadth-first search

Avoiding repeated states

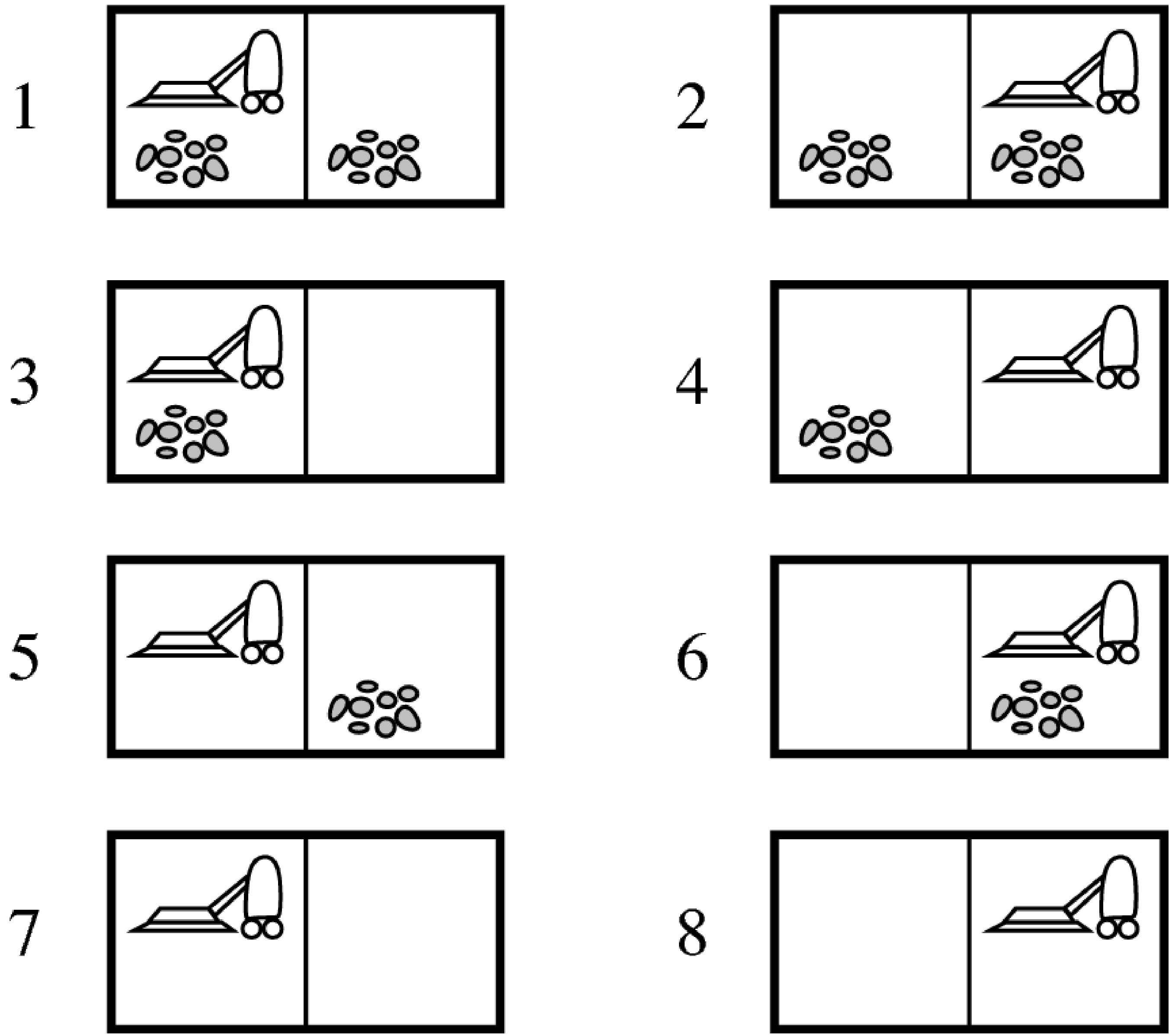


- we can smartly formulate a problem to avoid repeated states
  - **how we can do it in 8-queen problem?**
- cut the search tree - remembering visited states
  - find a trade-off between space and time
  - closed list - expanded nodes
  - open list - not yet expanded nodes

Searching with partial information

- sensorless problem
  - if an agent has no sensors at all, it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states
- contingency problem
  - if the environment is partially observable or if actions are uncertain, then the agent's percepts provide new information after each iteration. Each possible percept defines a contingency that must be planned for

Searching with partial information

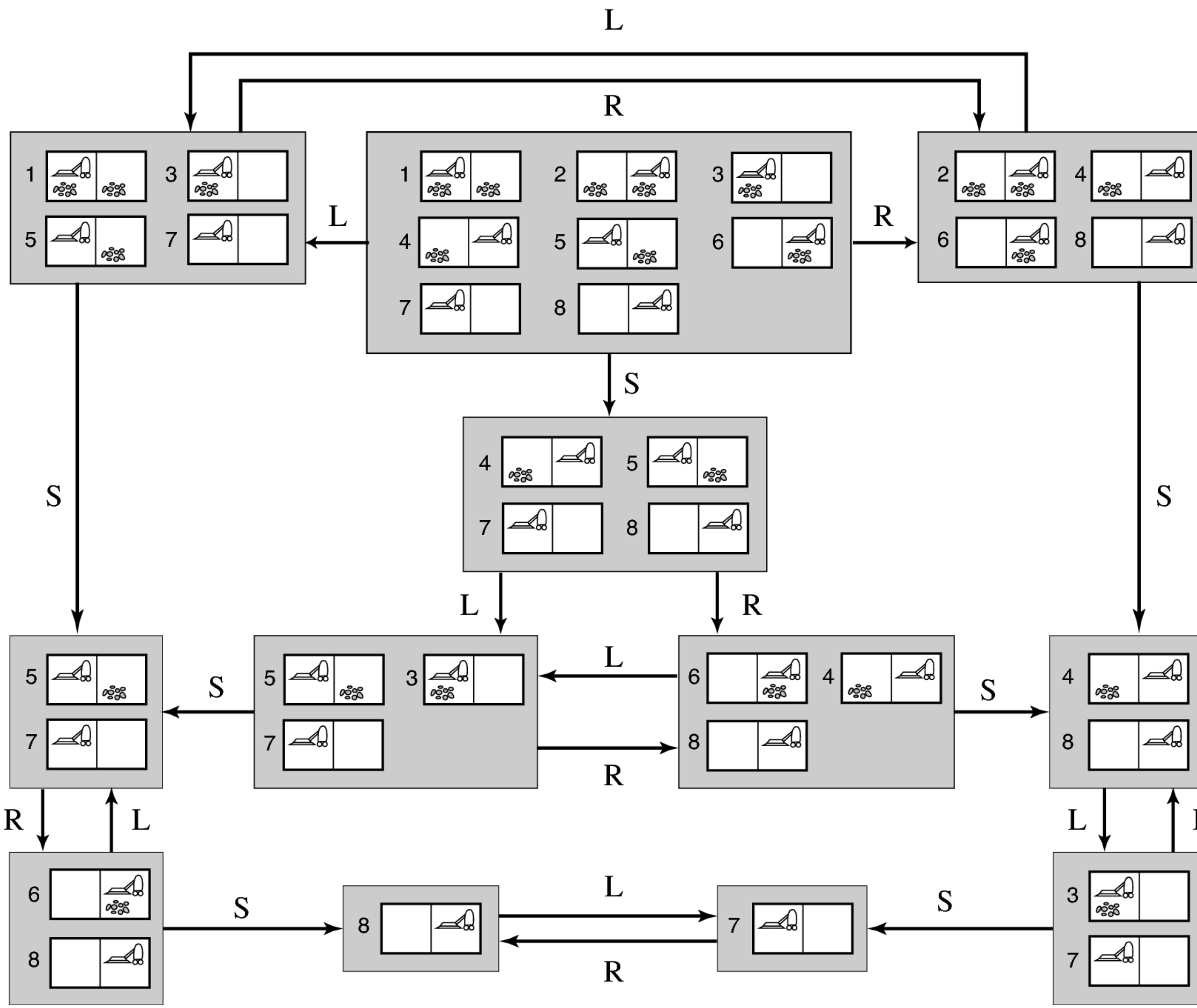


Sensorless problems

- an agent knows all the effects of its actions but has no sensors
  - initial state is one of the set {1,2,3,4,5,6,7,8}
  - Right will cause to be in one of the states {2,4,6,8}
  - [Right, Suck] will cause to be in one of {4,8}
  - [Right, suck, Left, Suck] guarantees to reach the goal state 7

- belief states

Sensorless problems



Contingency problems

- assume Murphy's law
  - Suck sometimes deposits dirt only if there is no dirt
    - percept [L,Dirty] means that an agent is in one of the states {1,3}
    - executing [Suck, Right] will lead to one of the states {6,8}
    - executing the final Suck action in state 6 leads to a goal state but executing Suck in state 8 might take us back to the state 6 (Murphy's law}, in which case the plan fails

  - [Suck, Right, **if** [R,Dirty] **then** Suck] is a solution

Thanks for your attention!
Questions?