Solving problems by searching

CS4881 - Artificial Intelligence

Jay Urbain, Ph.D.

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

Reflex & Problem Solving Agents

- Reflex agents
  - Base their actions on a direct mapping from states to actions.
  - Cannot operate well in environments requiring a sequence of operations
    - mapping sequences of state-actions would be too large to store, and would take too long to learn.
- Problem Solving or Goal based agents
  - Can succeed in this environment by considering future actions and the desirability of their outcomes.
  - A Problem solving agent is a goal based-agent that use atomic representations.
    - where states of the world are considered as wholes (no internal structure is visible to the agent).

Problem Solving Agents

- How an agent can find a *sequence of actions that achieves its goals*, when no single action will do.
- Goals help organize behavior by *limiting the courses of action* an agent seeks to achieve.
- *Goal formulation* based on the *current state* and the agent's performance measure is the 1st step in problem solving.
- *Problem formulation* is the process of deciding what actions and states to consider, given a goal.

Travel from Arad to Bucharest

- Map of Romania
  - Agent with several options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.
  - *Environment?*

Travel from Arad to Bucharest

- Environment assumptions:
  - Observable (know current state – *have map*)
  - Discrete (finite number of actions to choose)
  - Known (which states are reached by each action).
  - Deterministic (each action has one outcome).
- Under these assumptions:
  - *Solution to any problem is a fixed sequence of actions.*

Search

- *Search: sequence of actions to achieve goal.*
  - Search algorithm takes a problem as input and returns a solution in the form of a sequence of actions.
  - Once solution is found, actions it recommends are executed.

Formulate -> Search -> Action

- *When executing, the agent is running open loop, i.e., it ignores percepts since it already knows in advance what they will be.*

Formulate Problem

- State space of problem – forms directed graph
  - Initial state
  - Description of the possible actions
  - Description of what each action does (transition model)
- Goal test
- Path cost

Problem-solving agents

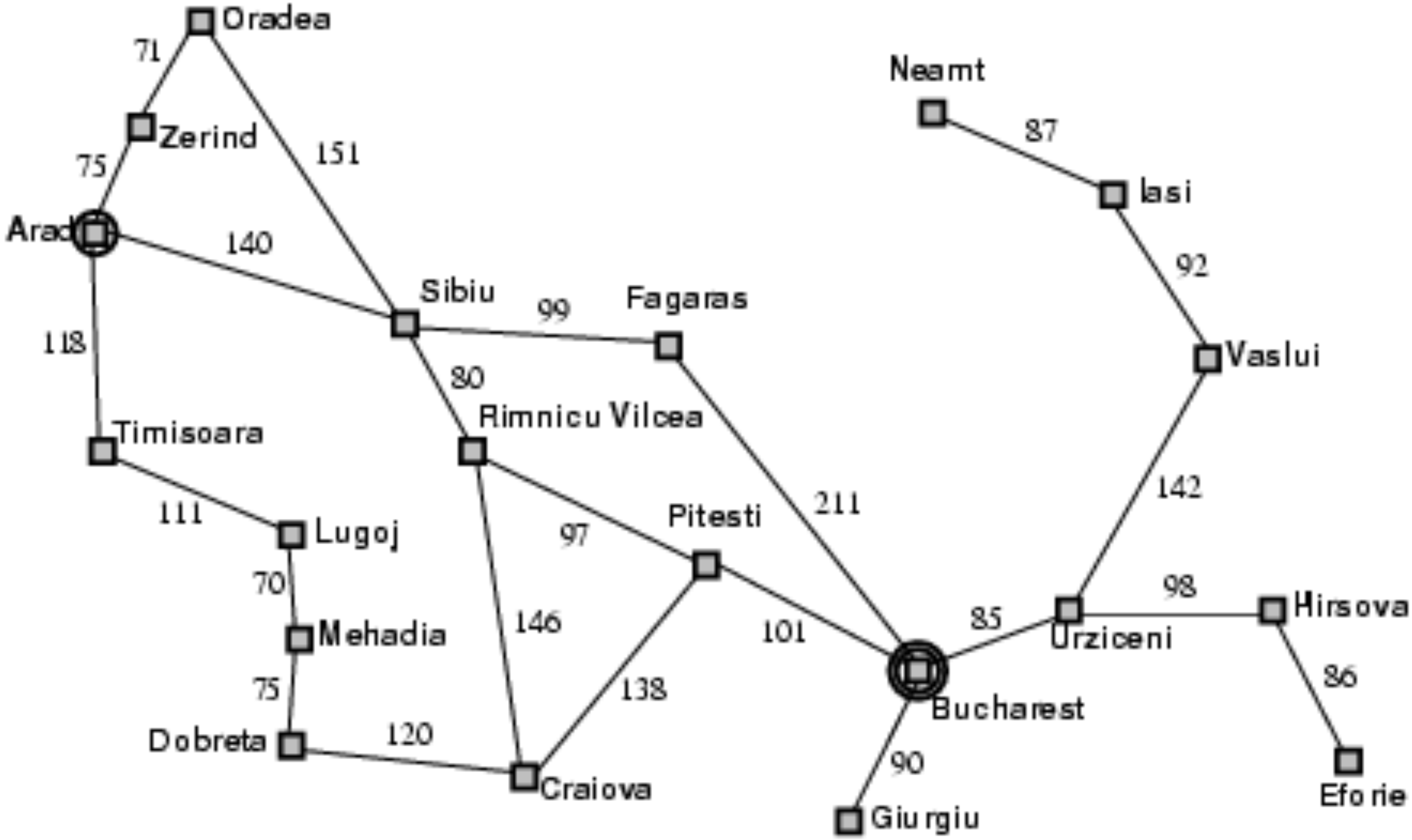
```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
   static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

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

Example: Romania

- On spring break in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest to Milwaukee.
- Formulate goal:
  - be in Bucharest
- Formulate problem (what actions and states to consider, given initial state and a goal):
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania

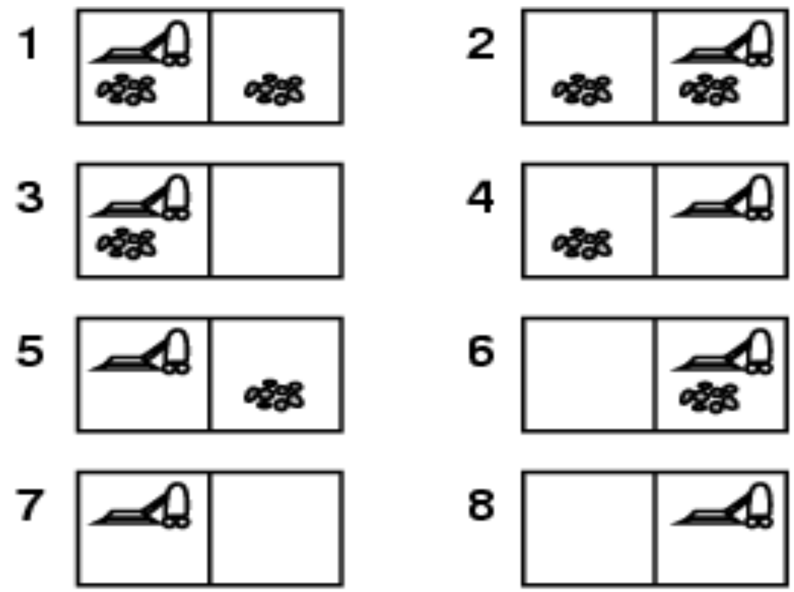


Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; *solution is a sequence*.
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; *solution is a sequence*.
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave search and, execution
- Unknown state space → exploration problem

Example: vacuum world

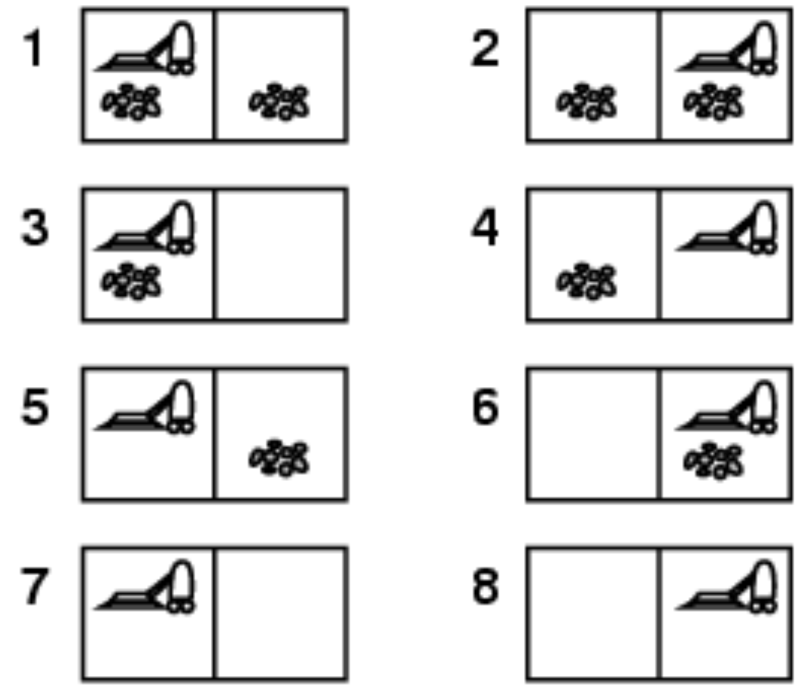
- Single-state (fully observable, deterministic), start in #5. Solution?



Example: vacuum world

- Single-state, start in #5.
  Solution? *[Right, Suck]*

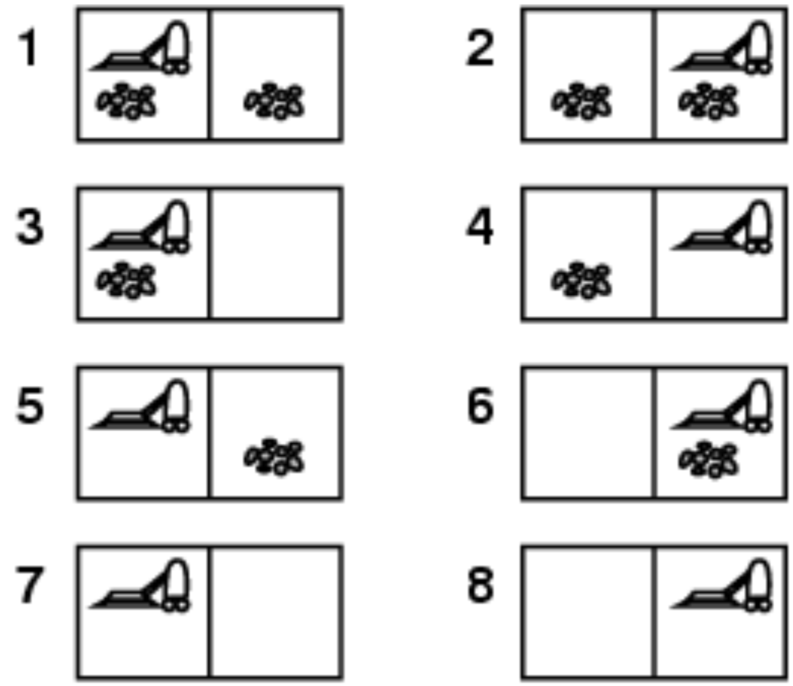
- Sensorless (non-observable), start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?



Example: vacuum world

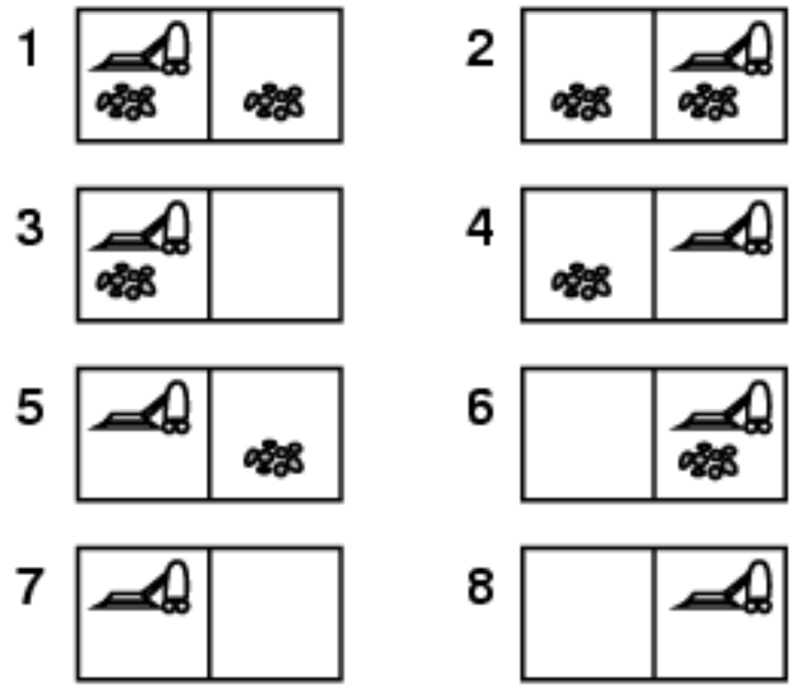
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution? *[Right,Suck,Left,Suck]*

- Contingency (Nondeterministic and/or partially observable)
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7 Solution?



Example: vacuum world

- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution? *[Right,* ***if*** *dirt* ***then*** *Suck]*



State problem formulation

A problem is defined by four items:

1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) = {<Arad → Zerind, Sibiu>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *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

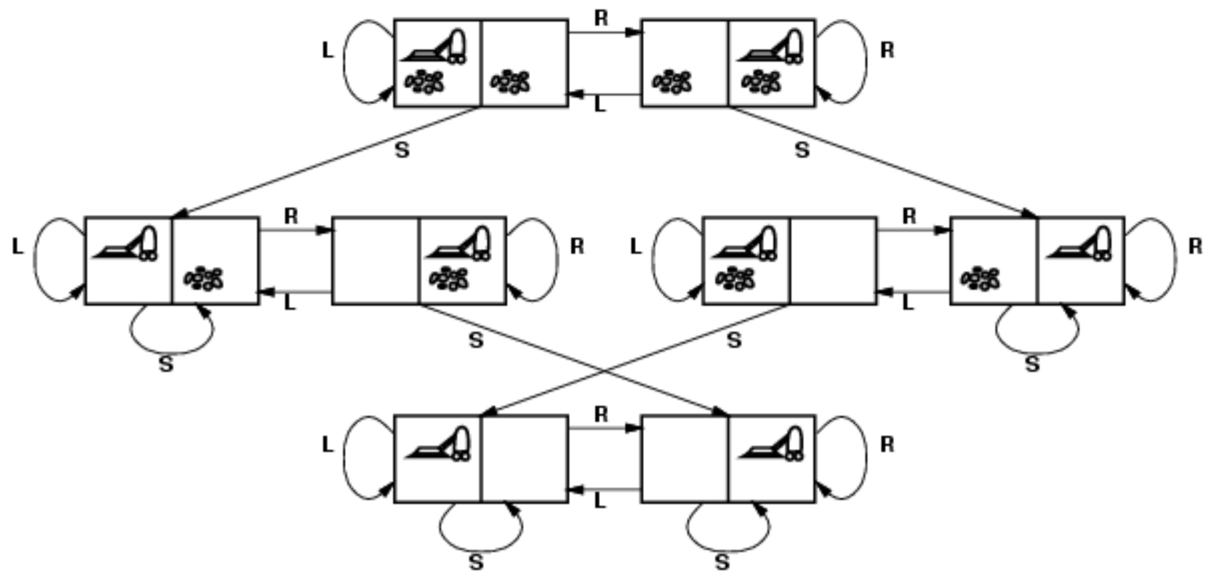
Selecting a state space

- Real world is absurdly complex
  - → state space must be abstracted for problem solving

- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.

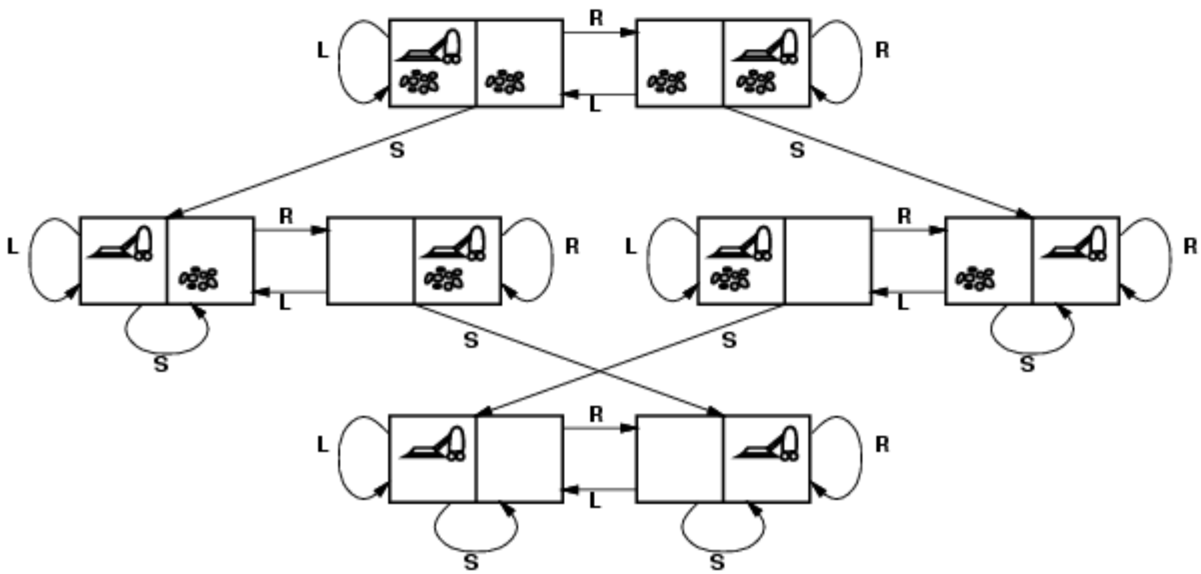
- ***(Abstract) solution =***
  - ***set of real paths that are solutions in the real world***
  - Each abstract action should be "easier" than the original problem

Vacuum world state space graph



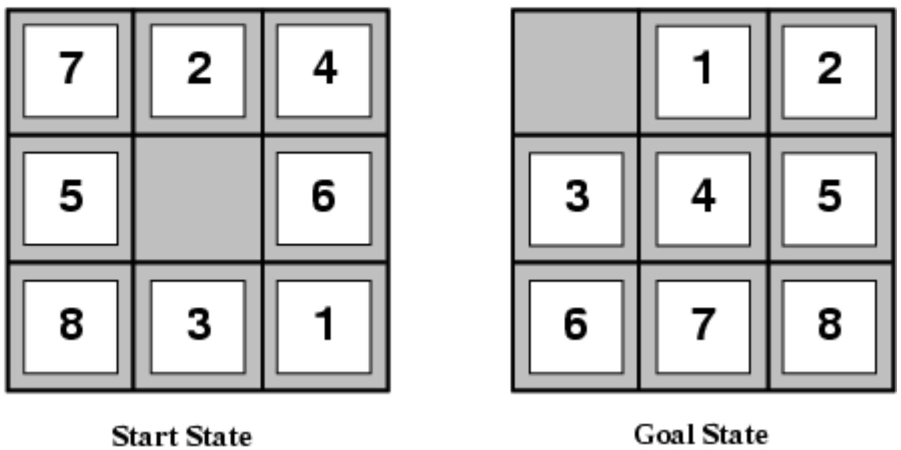
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



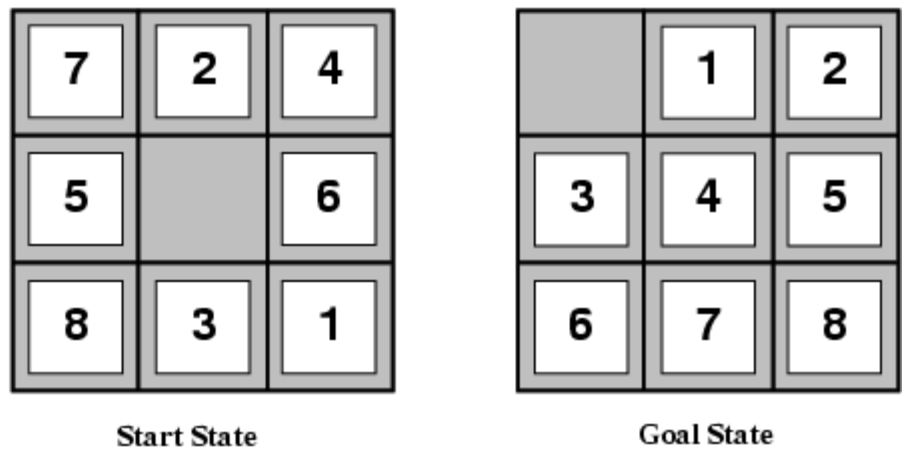
- states? dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



- states?
- actions?
- goal test?
- path cost?

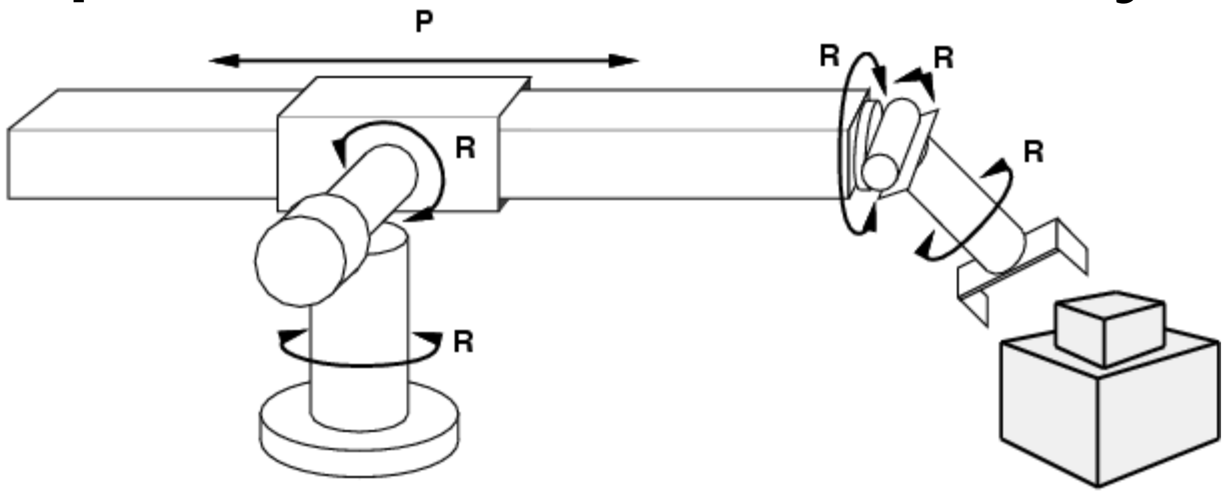
Example: The 8-puzzle



- states? locations of tiles (9! = 362,880)
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[*Note: optimal solution of n-Puzzle family is NP-hard*]

Example: robotic assembly



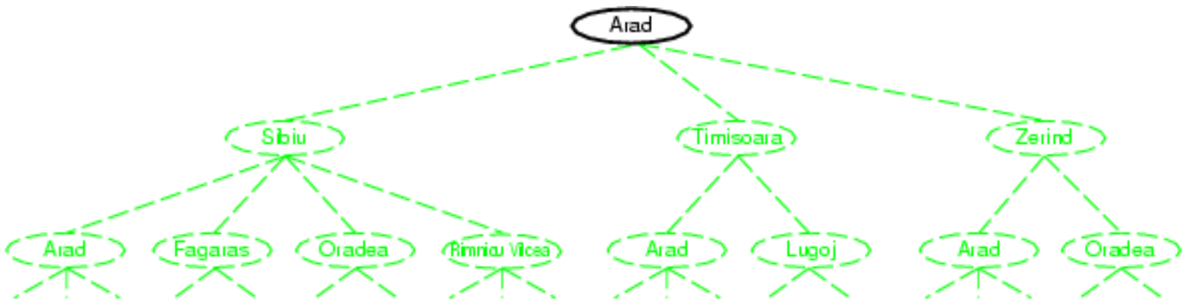
- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

Tree search algorithms

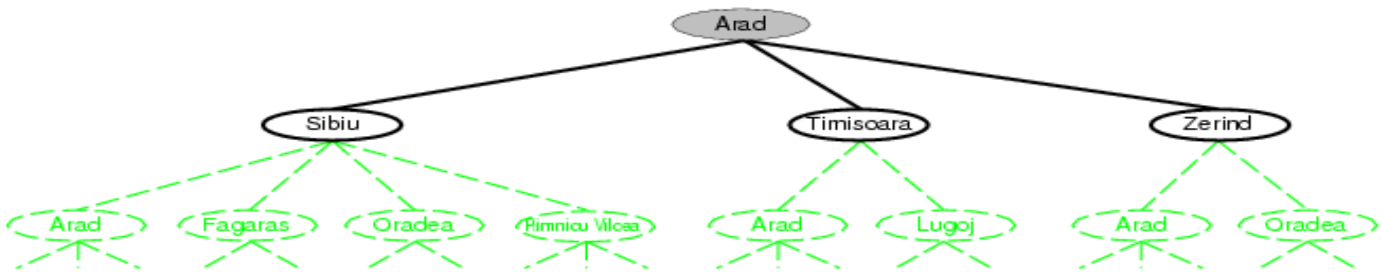
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
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
```

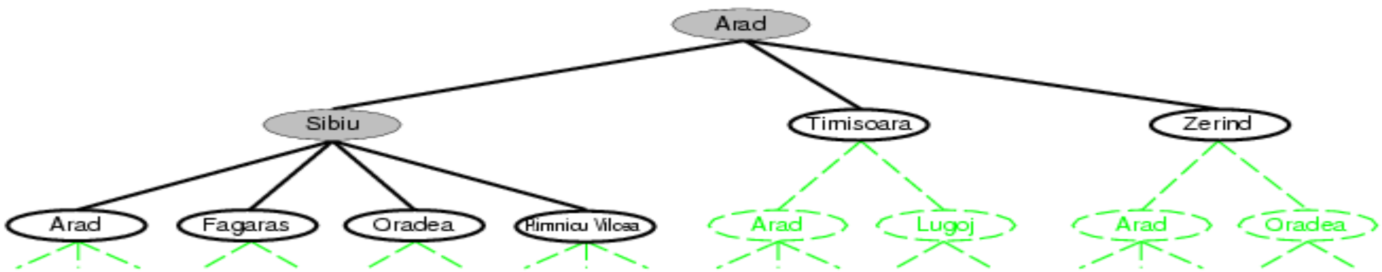
Tree search example



Tree search example



Tree search example



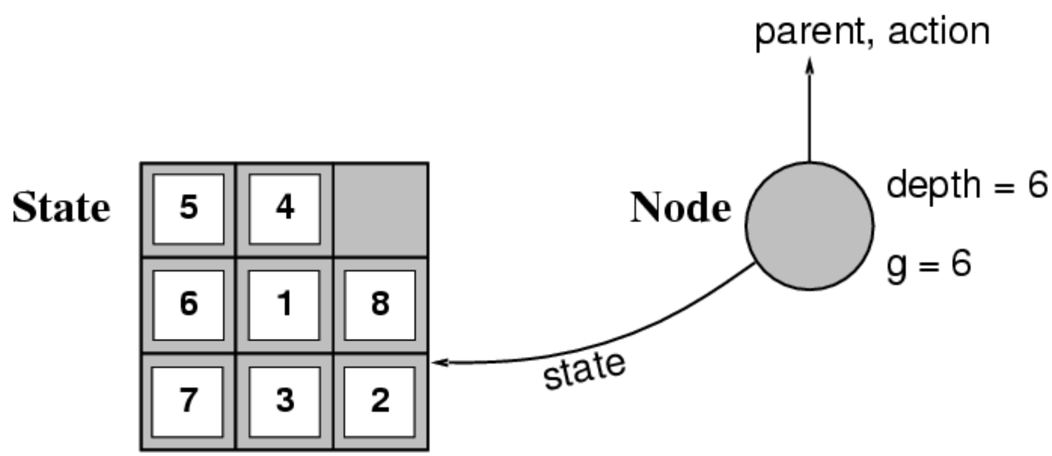
Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if fringe is empty then return failure
       node ← REMOVE-FRONT(fringe)
       if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
       fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
       s ← a new NODE
       PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
       PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
       DEPTH[s] ← DEPTH[node] + 1
       add s to successors
   return successors
```

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 `SuccessorFn` of the problem to create the corresponding states.

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 ∞)

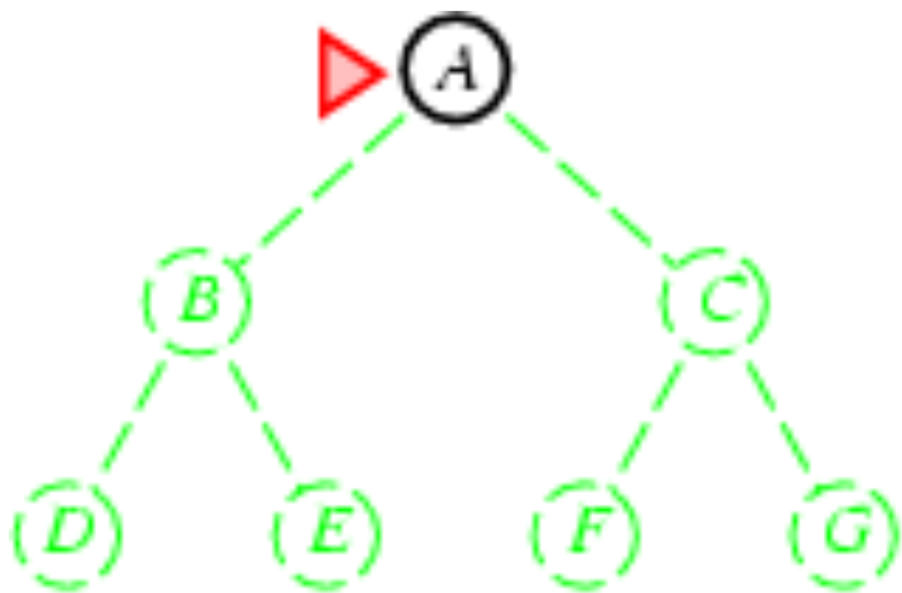
Uninformed search strategies

Uninformed search 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

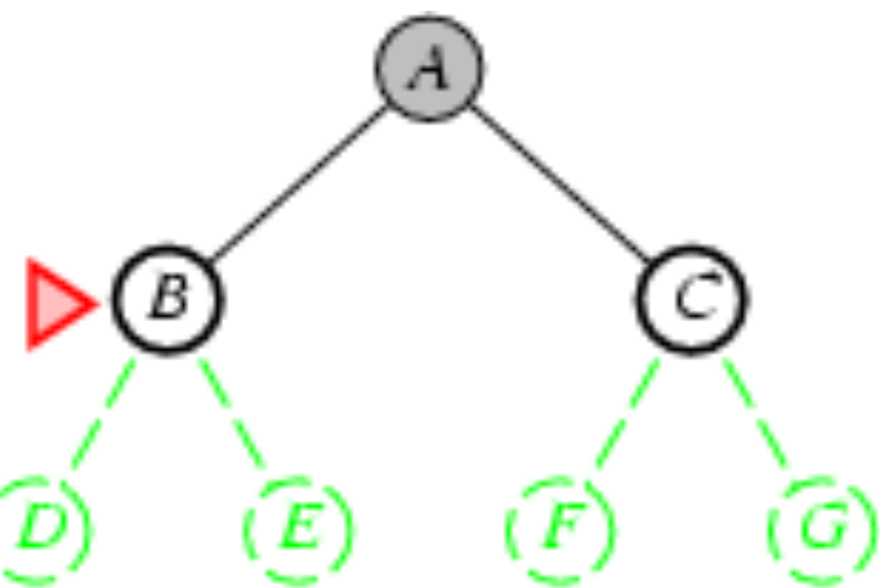
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



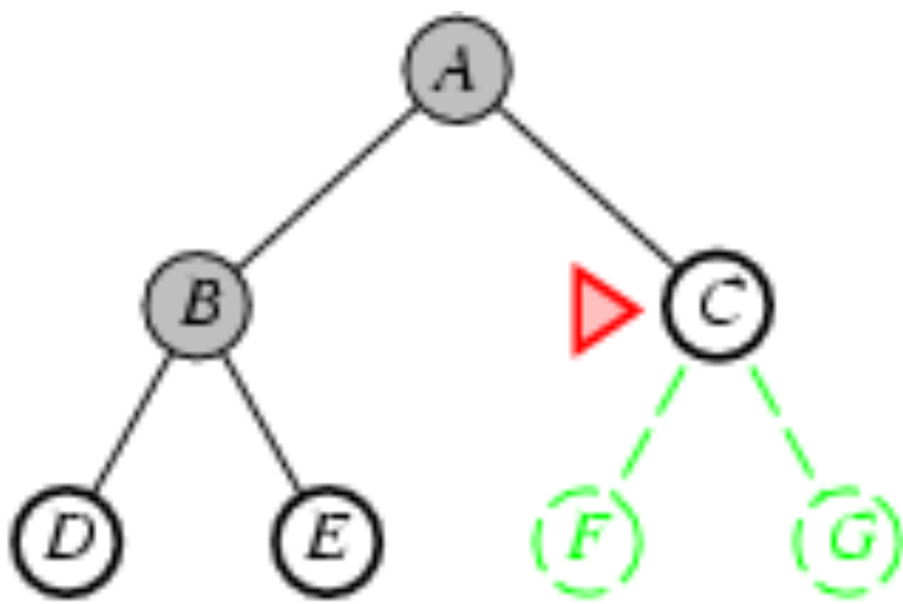
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



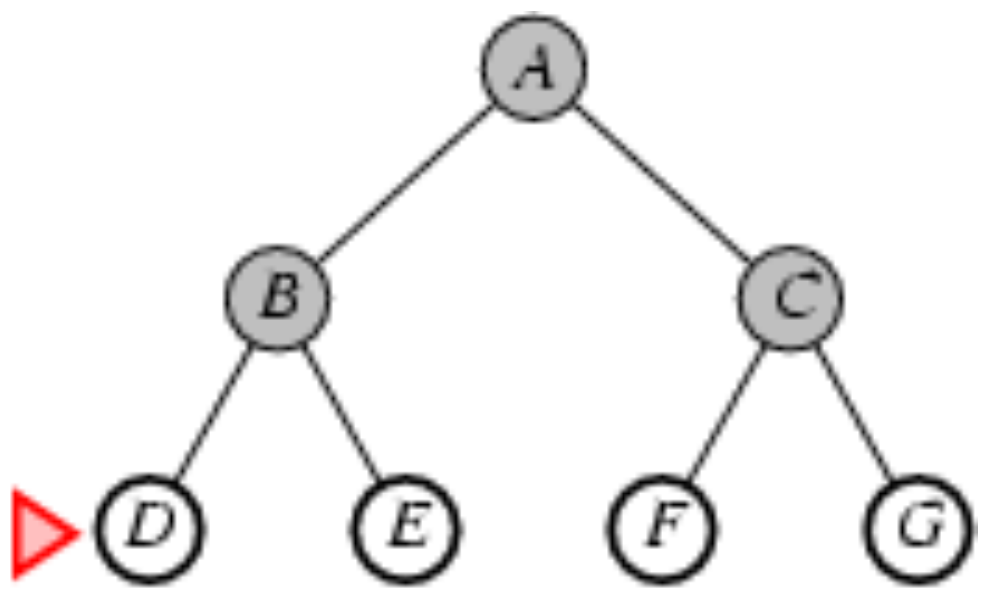
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

```
enqueue the root node // FIFO Queue
while(true)
   dequeue a node and examine it.
   if goal(node)
      return result
   else
      enqueue successor nodes (direct child nodes).
   if queue is empty
      return "not found"
```

Properties of breadth-first search

- Complete?
- Time?
- Space?
- Optimal?

Properties of breadth-first search

- Complete? Yes (if *b* is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$*
- Space? $O(b^d)$* (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

- Space is the bigger problem (more than time)!

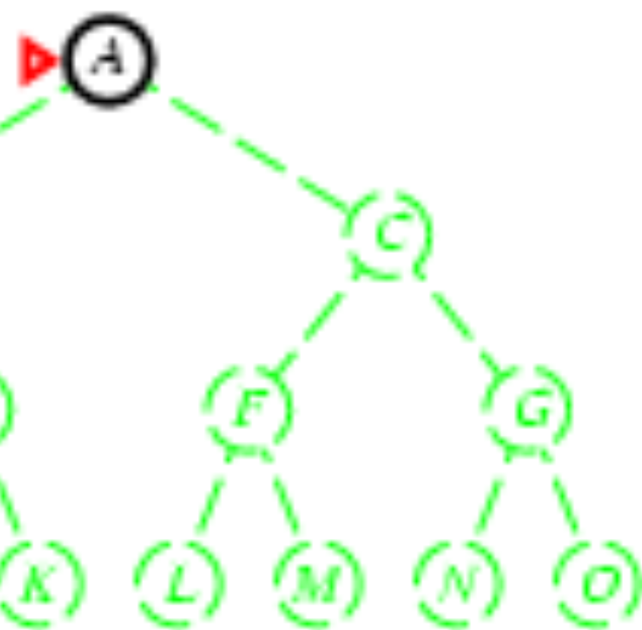
**Note:* $O(b^{d+1})$ *if goal test applied to nodes when selected for expansion rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected.*

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with g ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with g ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

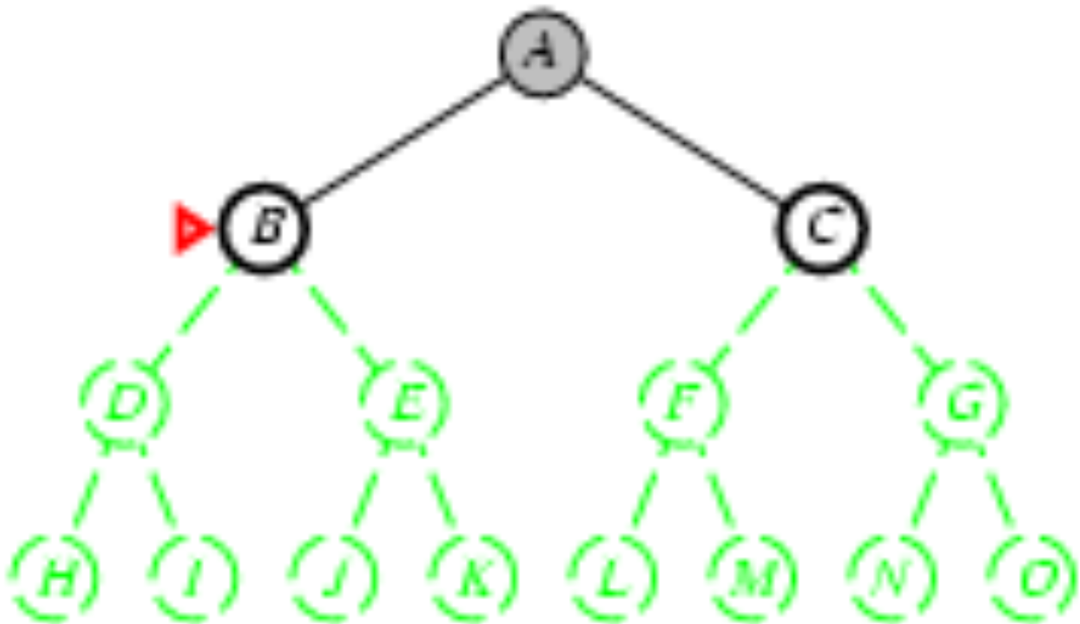
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue (stack), i.e., put successors at front



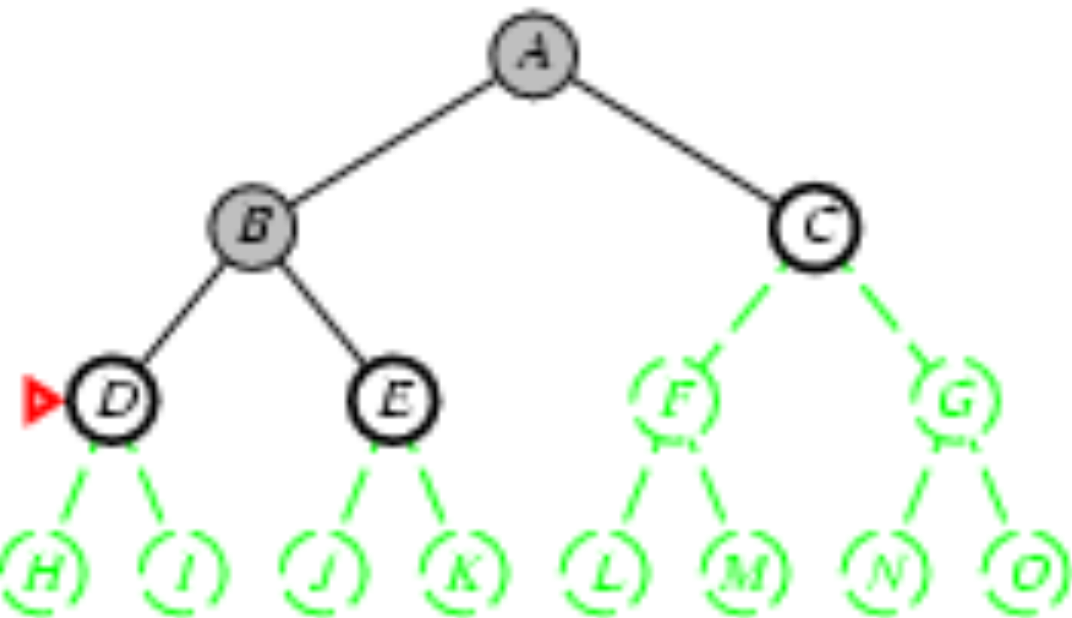
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



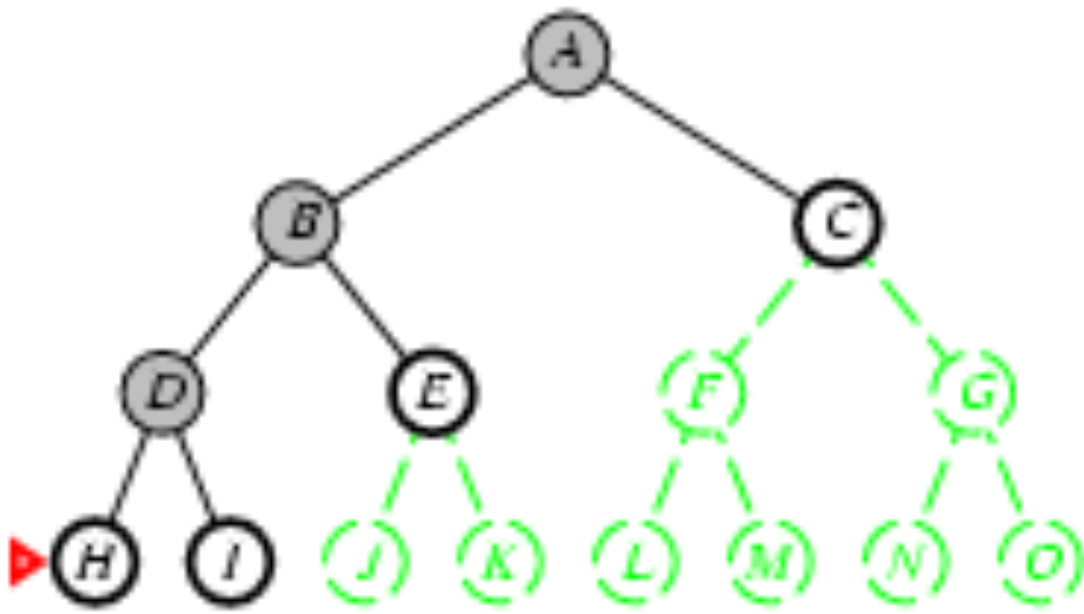
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - ☐ *fringe* = LIFO queue, i.e., put successors at front



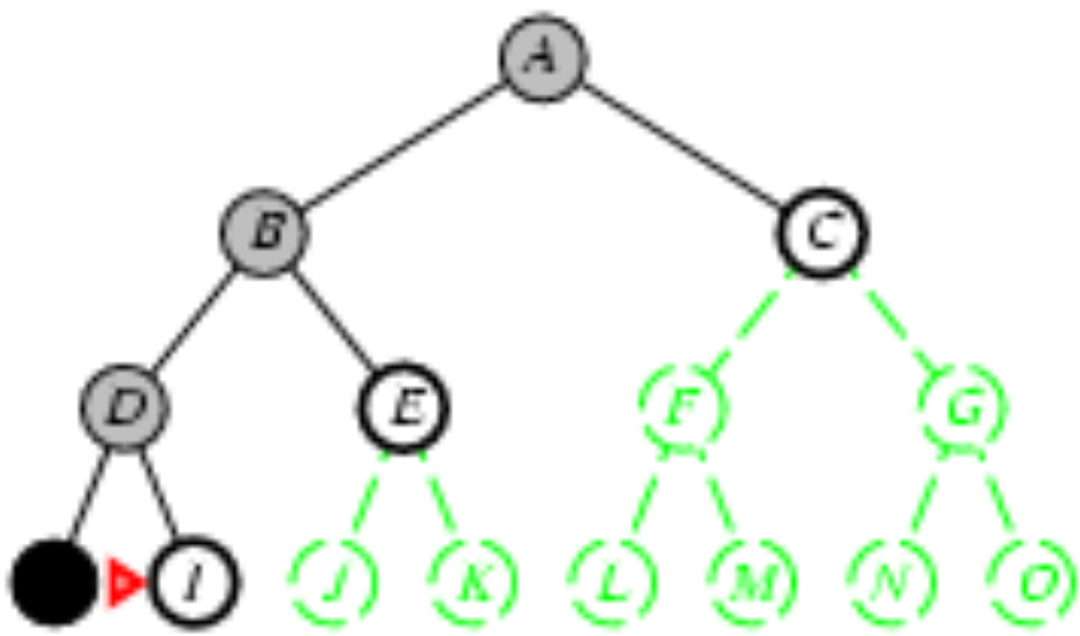
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



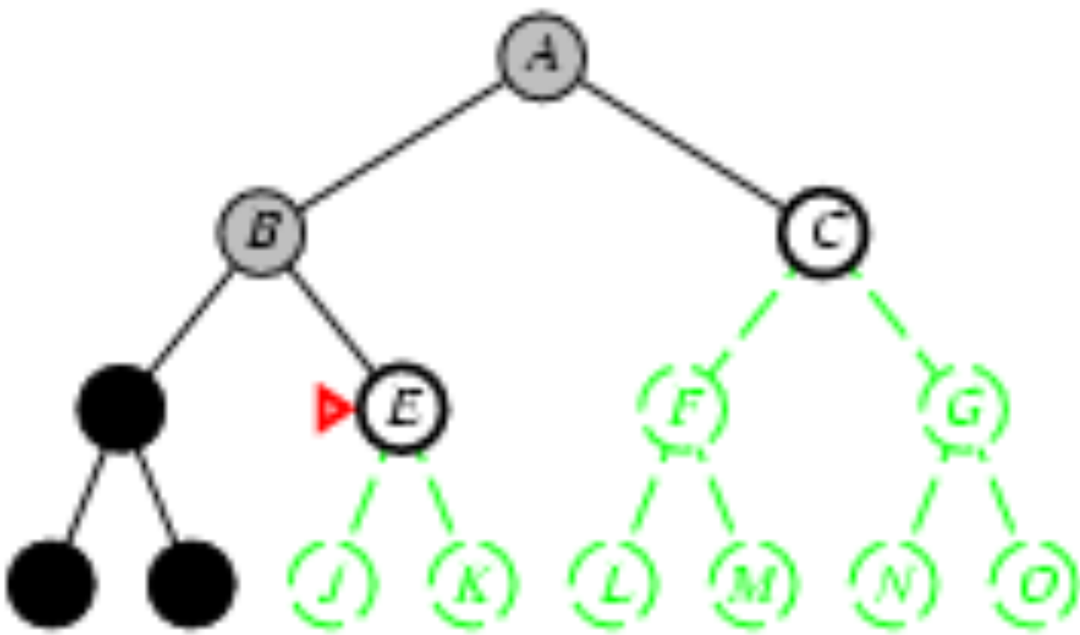
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



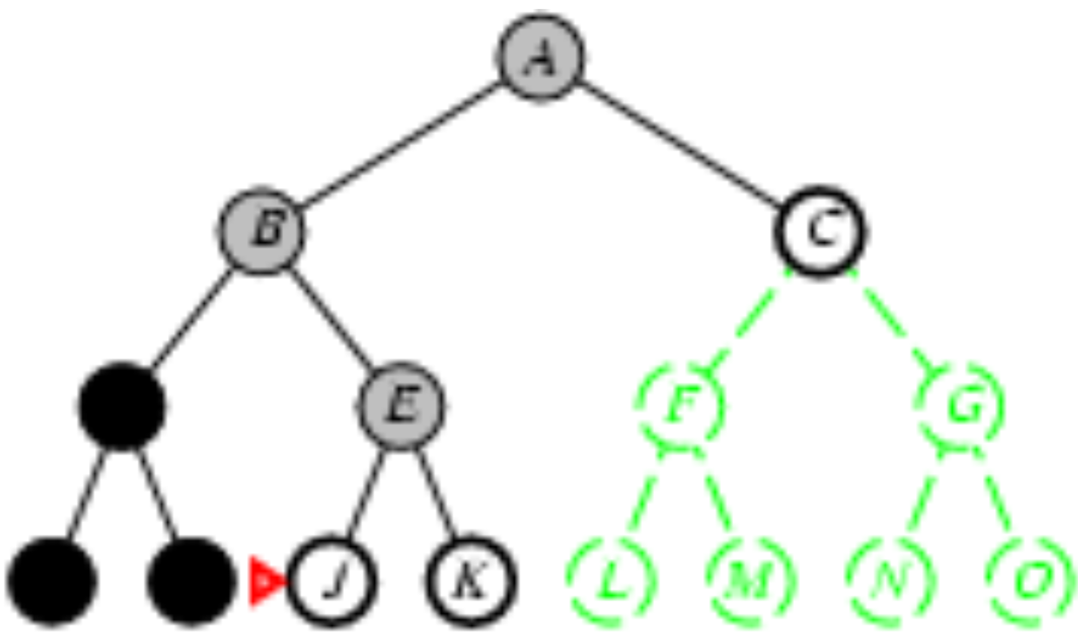
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



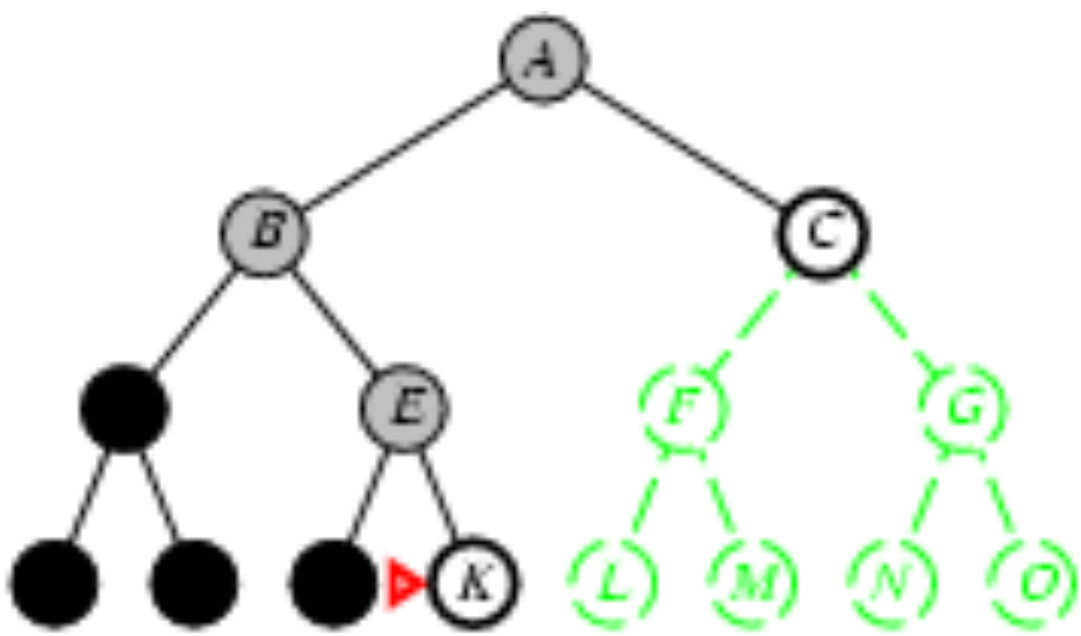
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



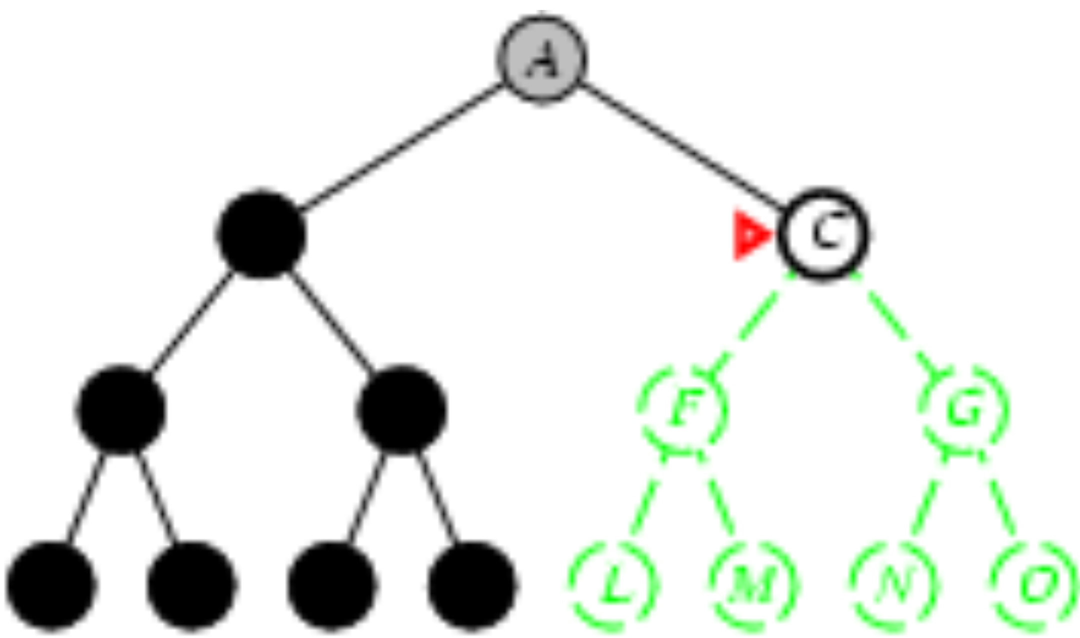
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



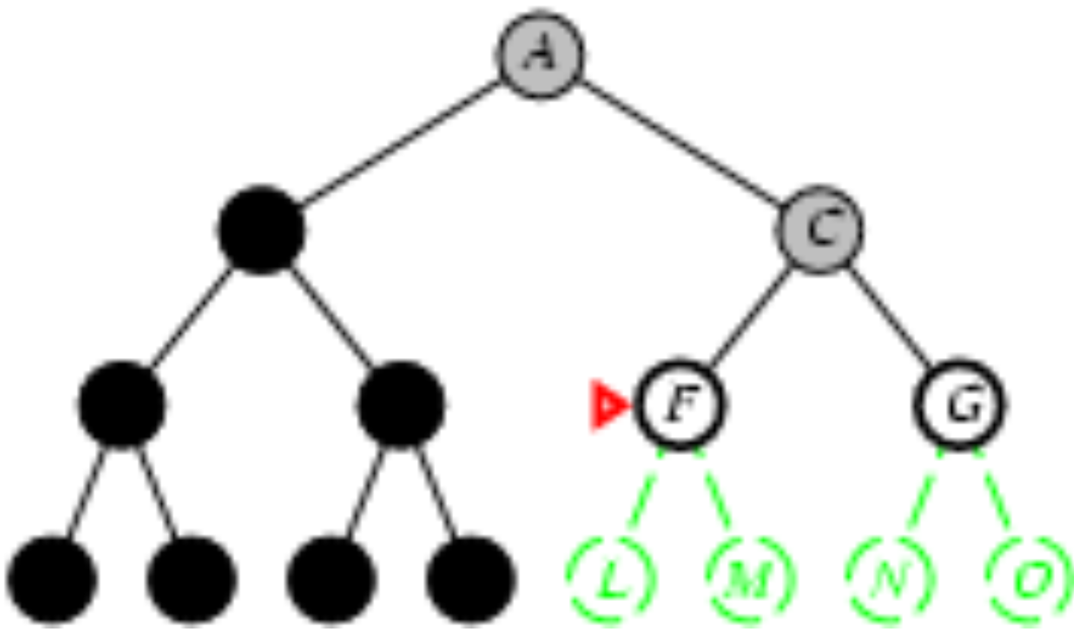
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



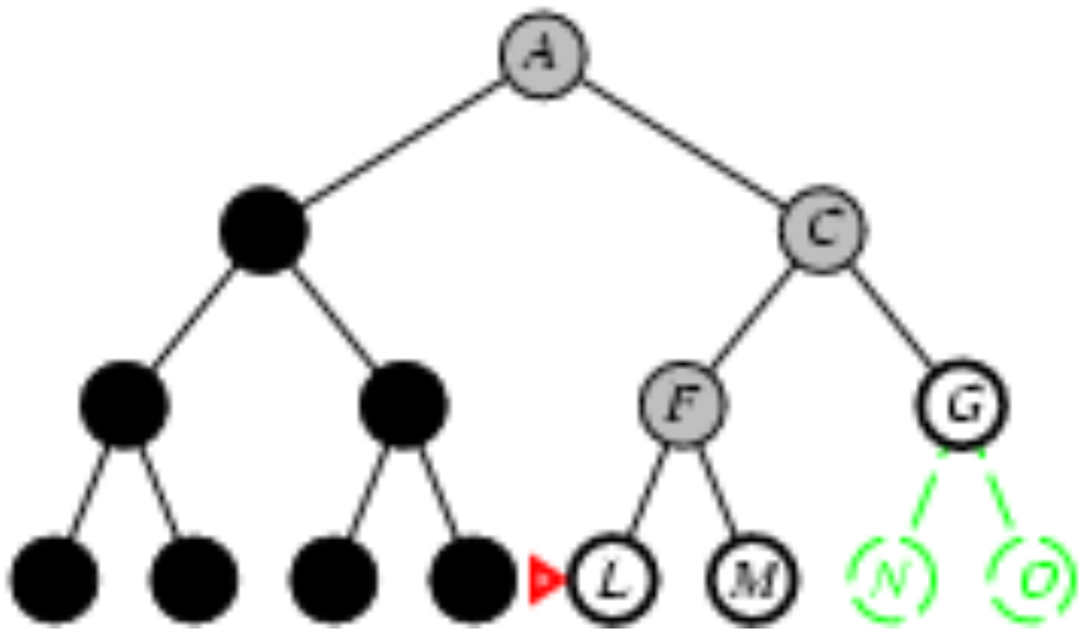
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



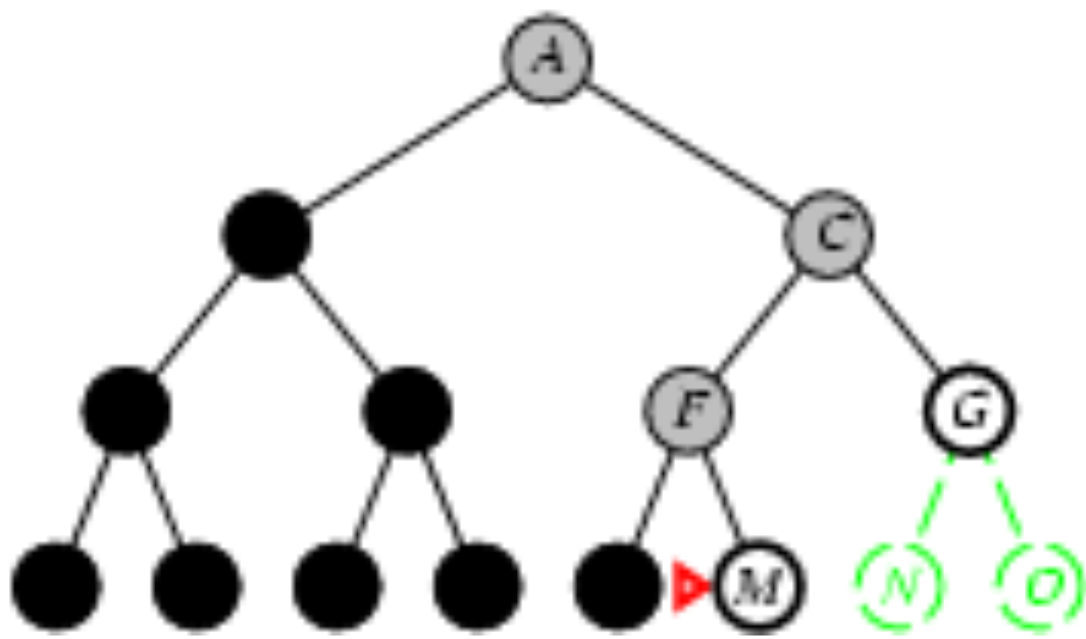
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



Properties of depth-first search

- Complete?
- Time?
- Space?
- Optimal?

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - → 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

Depth-first search (iterative)

```
push the root node // LIFO Queue == stack
while(true)
   pop a node and examine it.
   if goal(node)
      return result
   else
      push successor nodes (direct child nodes).
   if queue is empty
      return "not found"
```

Depth-first search (recursive)

```
dfs(node n)
    if goal(n)
        return n
    for each child node c
        dfs(c);
```

Depth-limited search

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

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

-
```
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

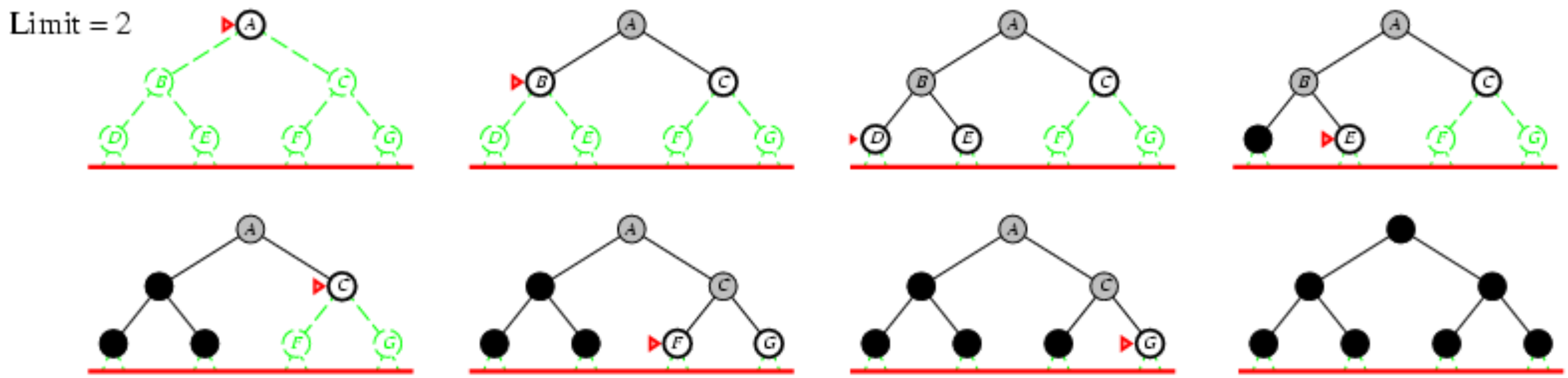
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening search l =0

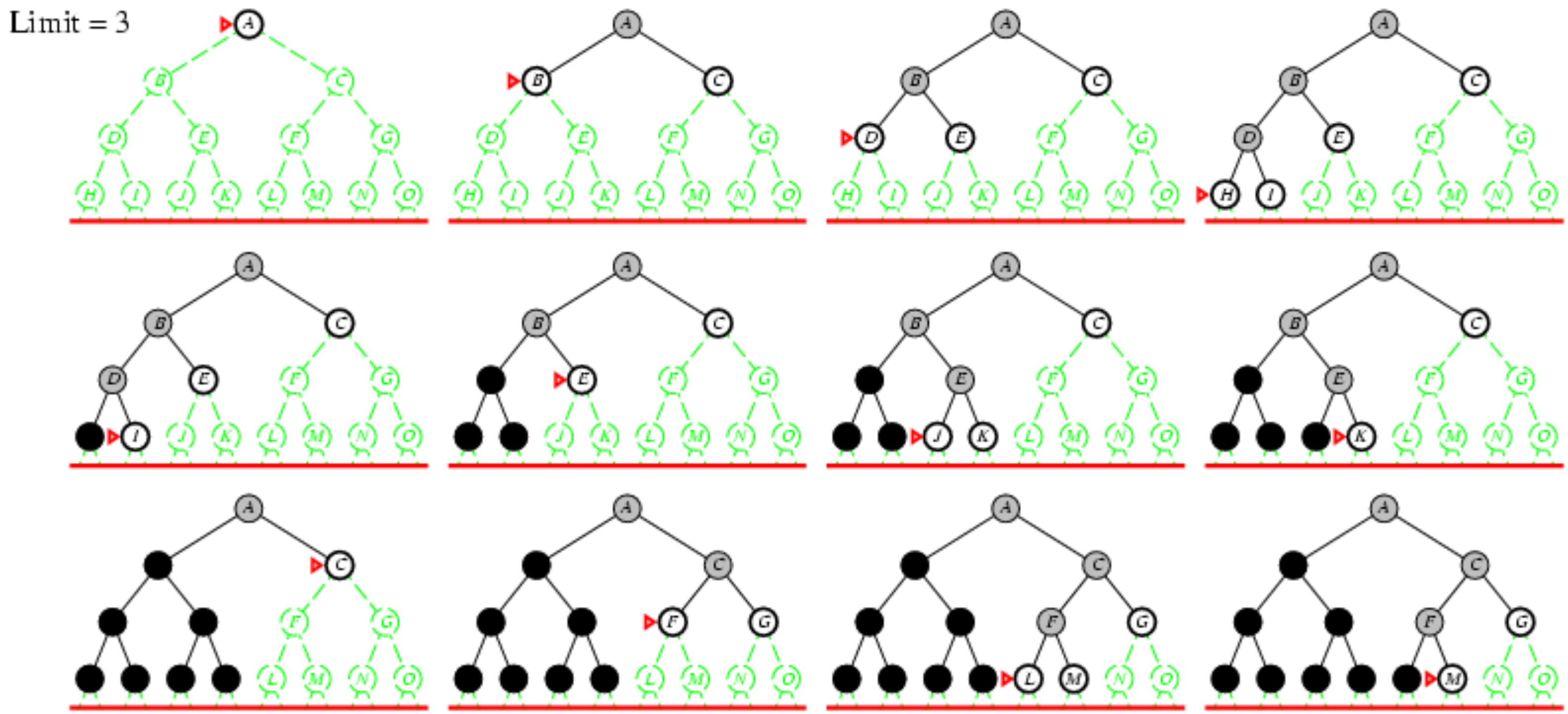
Limit = 0

Iterative deepening search *l* =1

Iterative deepening search l =2



Iterative deepening search *l* =3



Iterative deepening search

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

$$N_{IDS} = d*b\char`^1 + (d-1)b\char`^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

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

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

Properties of iterative deepening search

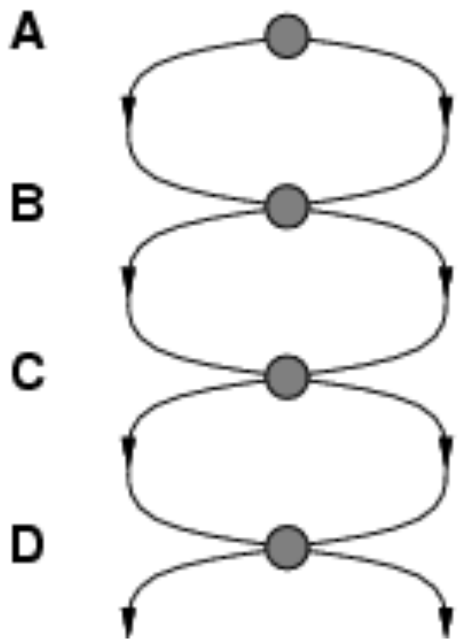
- Complete? Yes
- Time? $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

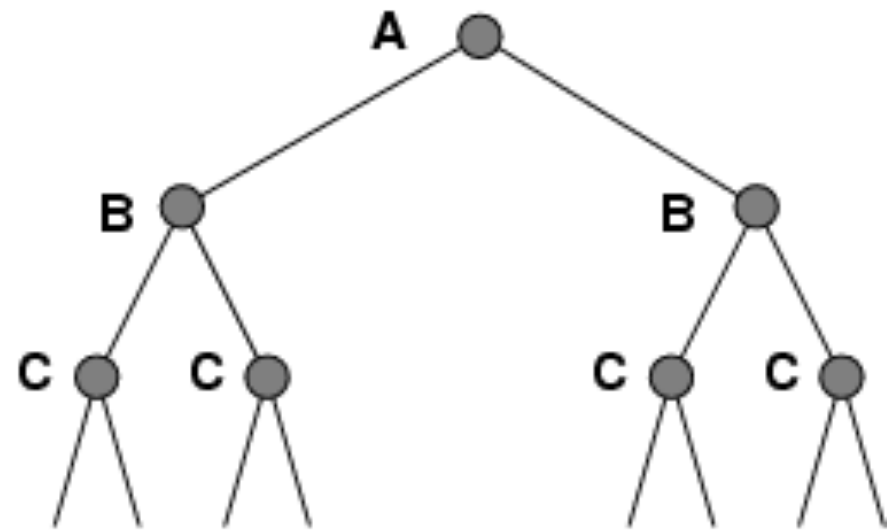
Summary of algorithms

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

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!





Graph search

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if *fringe* is empty **then return** failure
  node ← REMOVE-FRONT(*fringe*)
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

Summary (1)

- Actions in environments that are *determinisitc*, *observable*, *static*, and *completely known*.
- In these cases, an agent can construct sequences of actions that achieve its goals – this process is called *search*.
- Problem formulation usually requires *abstracting* away real-world details to define a state space that can feasibly be explored.
- Variety of uninformed search strategies.
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

Summary (2)

- Before an agent can start searching for solutions, it must formulate a ***goal***, then use the goal to formulate a ***problem***.

- A problem consists of four parts:
  1. Initial state
  2. Set of states
  3. Set of actions
  4. Goal test function
  5. Path cost function

- A ***path*** through the ***state space*** from the ***initial state*** to a ***goal state*** is a ***solution***.

- A single general TREE-SEARCH algorithm can be used to solve any problem; specific variants of the algorithm embody different strategies.

Summary (3)

- Search algorithms are judged on the basis of:
  - Completeness
  - Optimality
  - Time complexity
  - Space complexity

- Breadth-first search – complete, optimal for unit step costs, and has space & time complexity costs of *$O(b^d)$. Where d is depth of shallowest soln.*

- Uniform-cost search – like Breadth-first search, but expands node with lowest cost. It's complete and optimal if cost of each step exceeds some positive έ.

- Depth-first search – selects the deepest unexpanded node in the search tree for expansion. It's neither complete nor optimal has time complexity costs of *$O(b^m)$ and space complexity costs $O(bd)$.* Where *m* is max depth.

- Depth-limited search imposes a fixed limit on depth-first search.

Summary (4)

- Bi-directional search can enormously reduce time complexity, but is not always applicable and may require too much space.
- For graphs it is usually important to check for repeated states. GRAPH-SEARCH algorithm checks for repeated states.
- When the environment is partially observeable, the agent can apply search algorithms in the space of belief states, or sets of states the agent might be in.
- A contingency plan is needed to handle unknown circumstances that may arise.