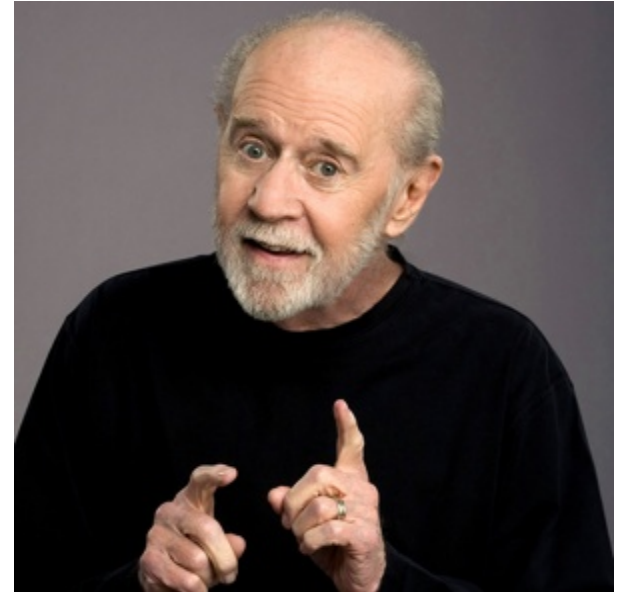




Solving problems by searching

Chapter 3

George Carlin quote of the day



Here's a bumper sticker I'd like to see: "We are the proud parents of a child who's self-esteem is sufficient that he doesn't need us promoting his minor scholastic achievements on the back of our car."



Outline

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



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

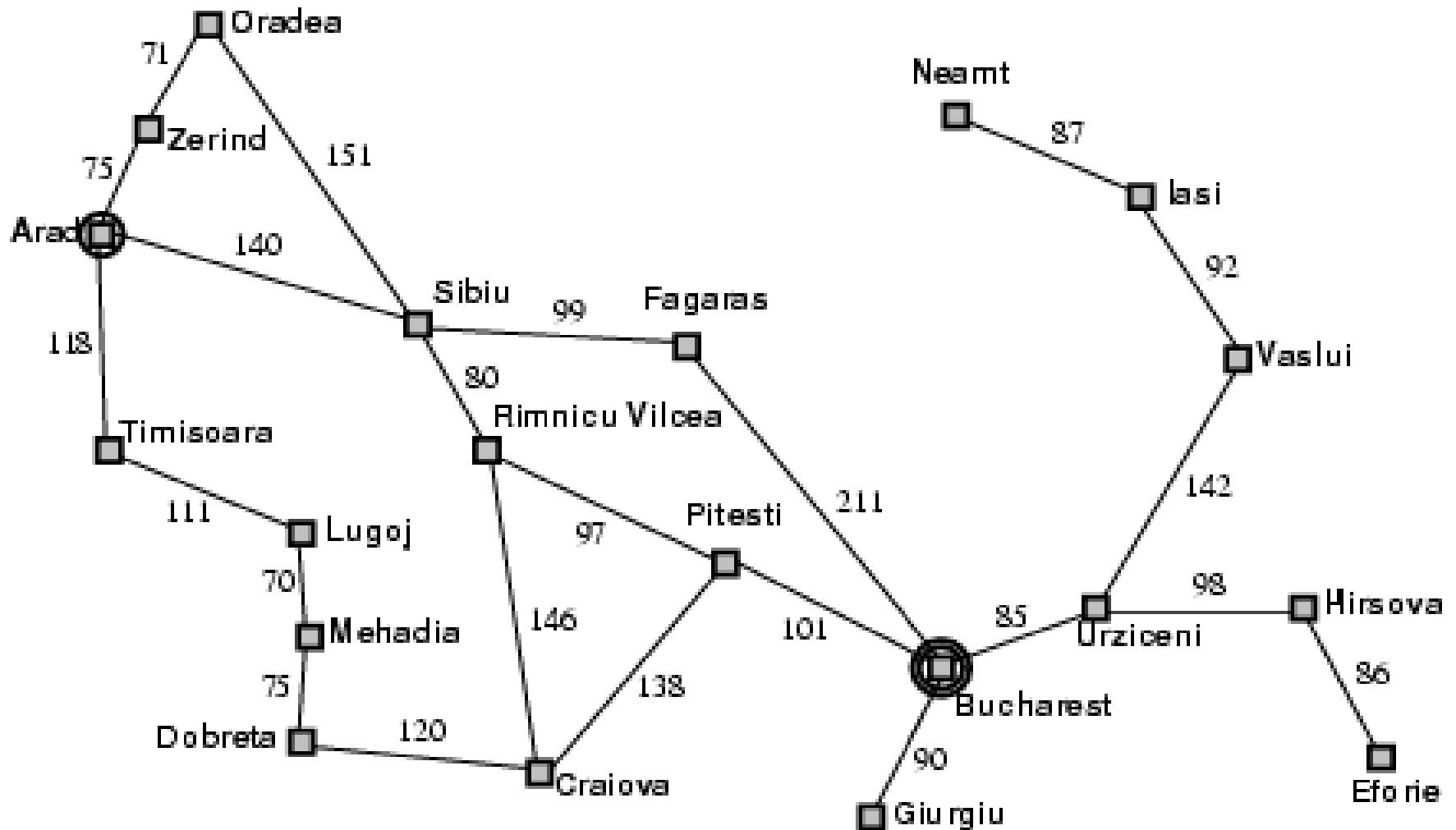
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```



Example: Romania

- On holiday in Romania; currently in Arad
 - (Arad = provincial town)
- Flight leaves tomorrow from Bucharest
 - (that's the capital; basically your best escape route)
- Formulate goal:
 - be in Bucharest (by tomorrow morning)
- Formulate problem:
 - **states**: various cities
 - **actions**: drive between cities (assuming there's a road...)
- Find solution:
 - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romanian roads (see AIMA)



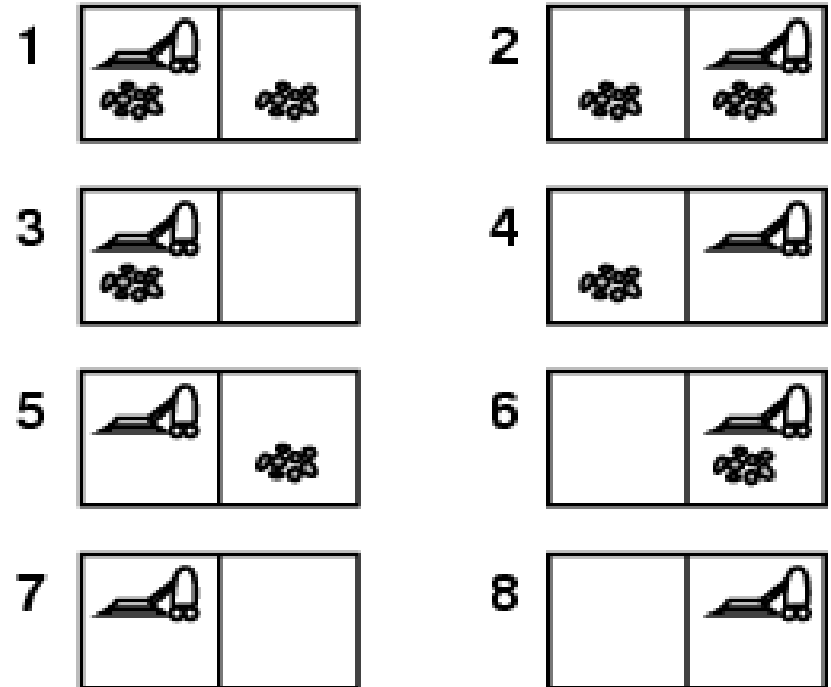


Problem types

- Deterministic, fully observable → single-state problem
 - Agent knows exactly which state it will be in (after each step); 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, execution
- Unknown state space → exploration problem

Example: vacuum world

- Single-state, start in #5.
Solution?



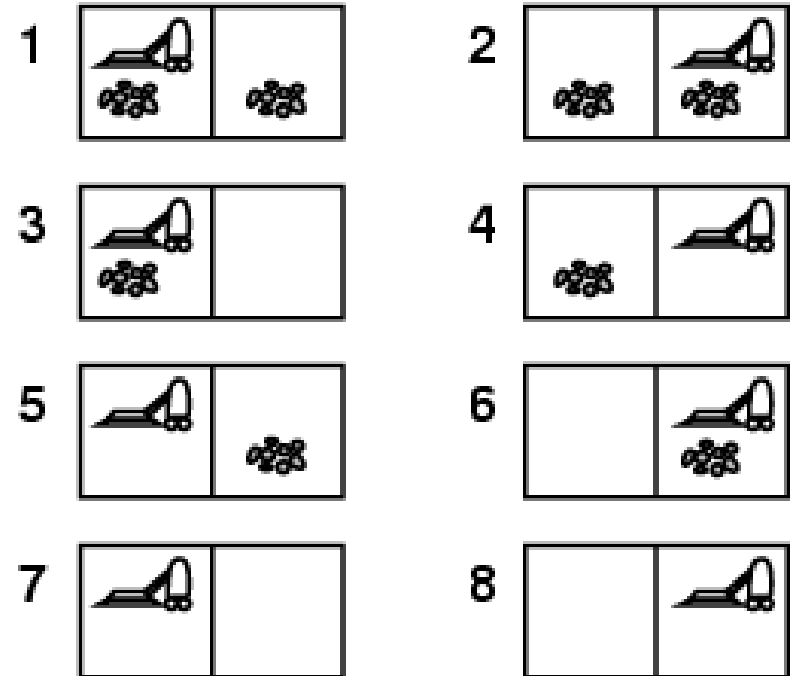
Example: vacuum world

- Single-state: start in #5.

Solution? [*Right, Suck*]

- Sensorless: 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 $\{2, 4, 6, 8\}$

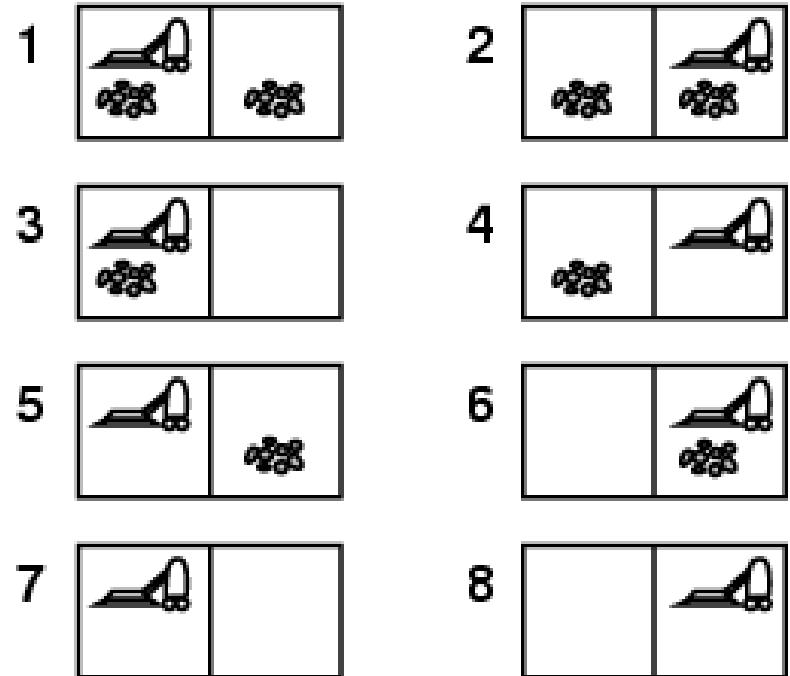
Solution?

[Right, Suck, Left, Suck]

- Contingency (planning)

- 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

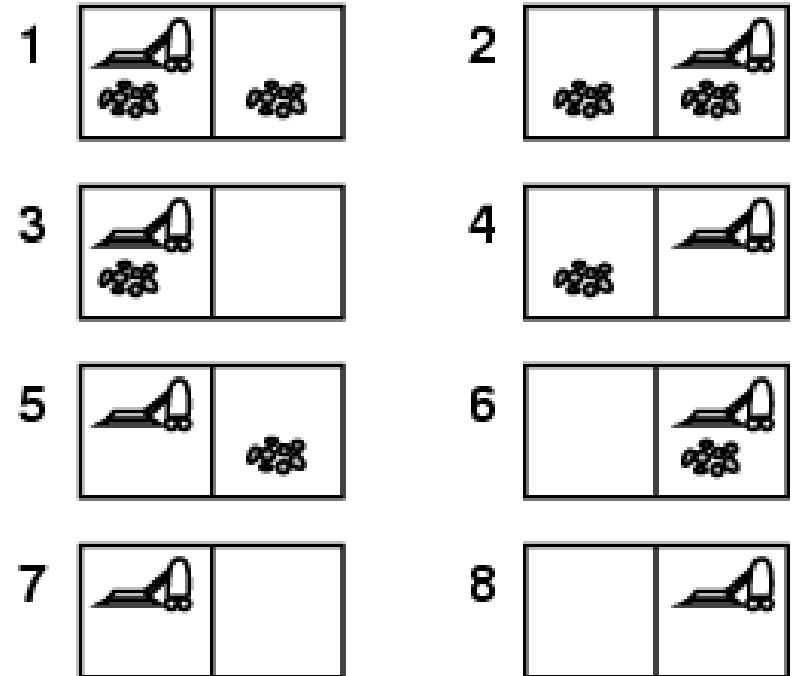
- Sensorless, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$ e.g., *Right* goes to $\{2, 4, 6, 8\}$

- Solution?

[Right, Vacuum, Left, Vacuum]

- Contingency

- Nondeterministic:
- Partially observable:
location, dirt at current location.
- Percept: *[L, Clean]*, i.e., start in #5 or #7



Solution? *[Right, if Dirt then Vacuum]*



Single-state problem formulation

A **problem** is defined by four items:

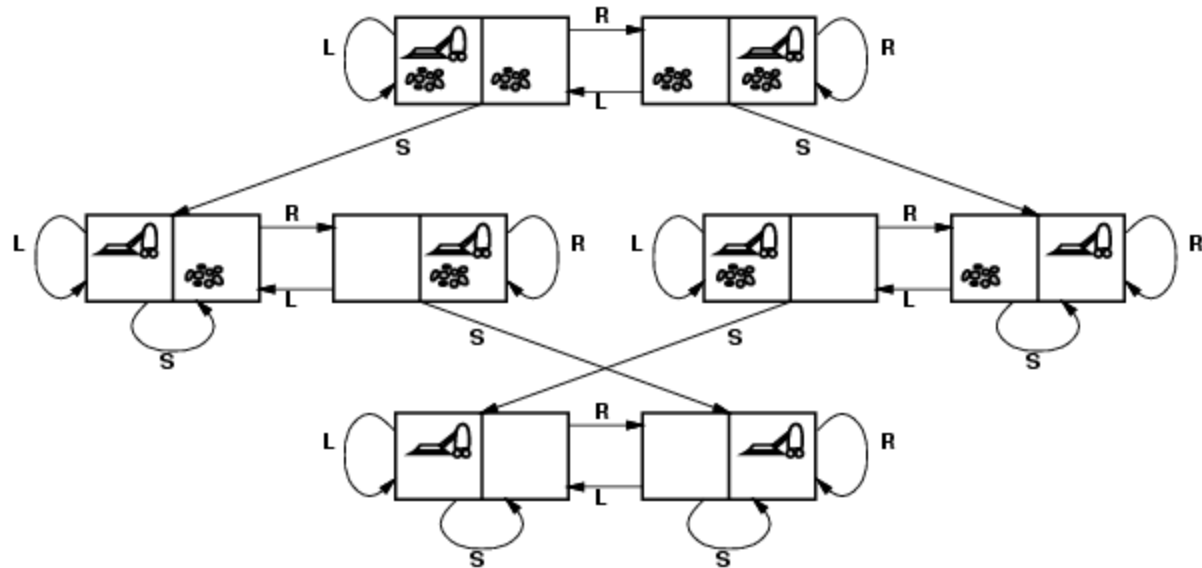
1. **initial state** e.g., "at Arad"
 2. **actions** or **successor function** $S(x)$ = set of action–state pairs
 - e.g., $S(Arad) = \{ \langle Arad \rightarrow Zerind, Zerind \rangle, \dots \}$
 3. **goal test**, can be
 - **explicit**, e.g., $x = \text{"at Bucharest"}$
 - **implicit**, e.g., $Checkmate(x)$
 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

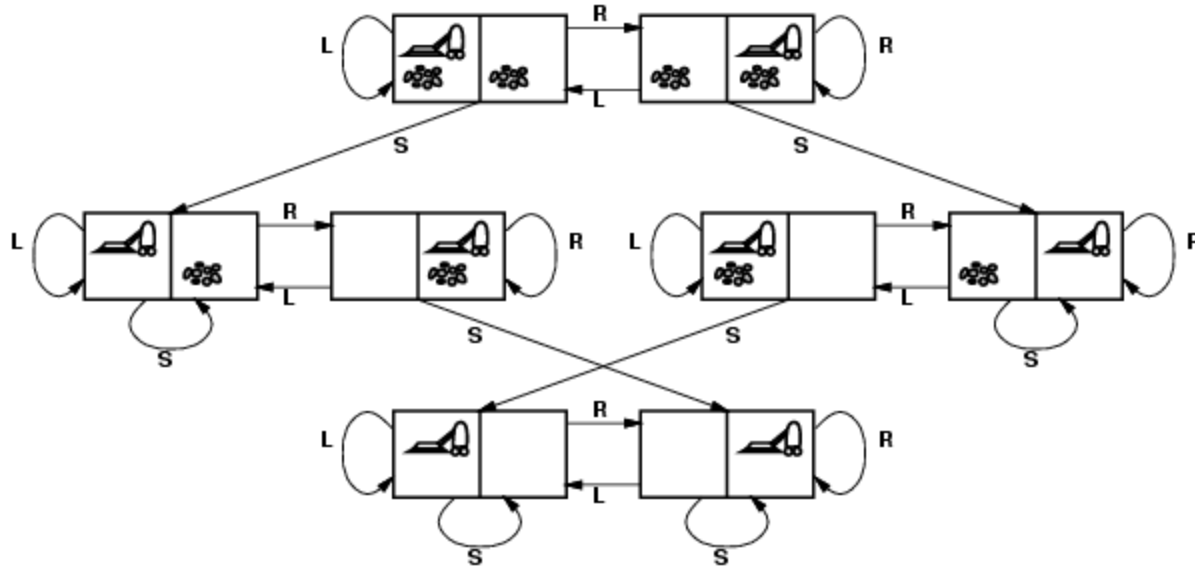
- Real world is absurdly complex
 - state space must be **abstracted** for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
 - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed 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

Vacuum world state space graph



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

Vacuum world state space graph



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

Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

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

Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

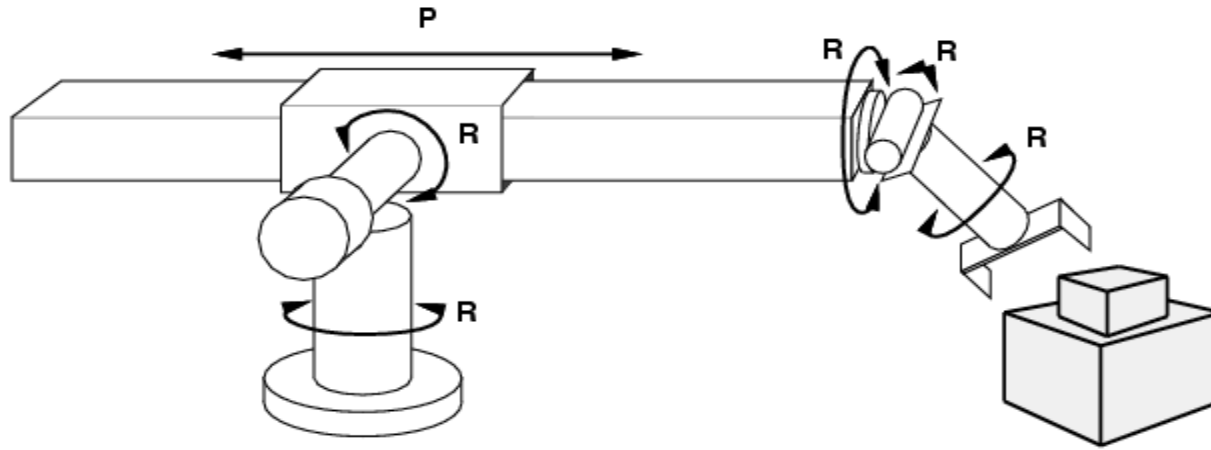
	1	2
3	4	5
6	7	8

Goal State

- states? locations of tiles
- 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

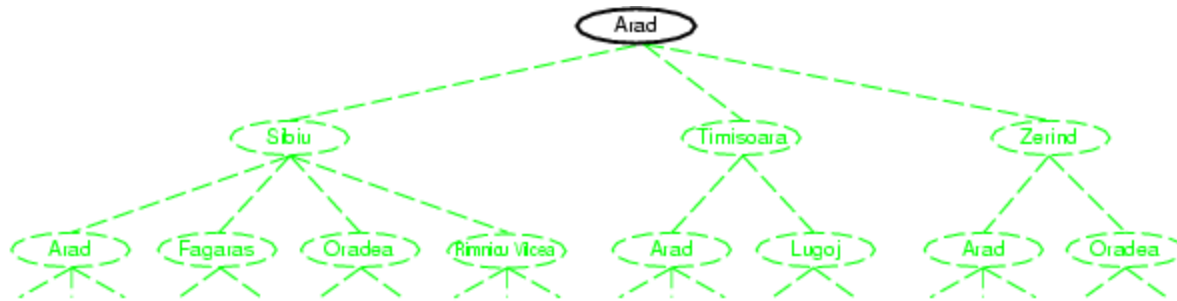


Tree search algorithms

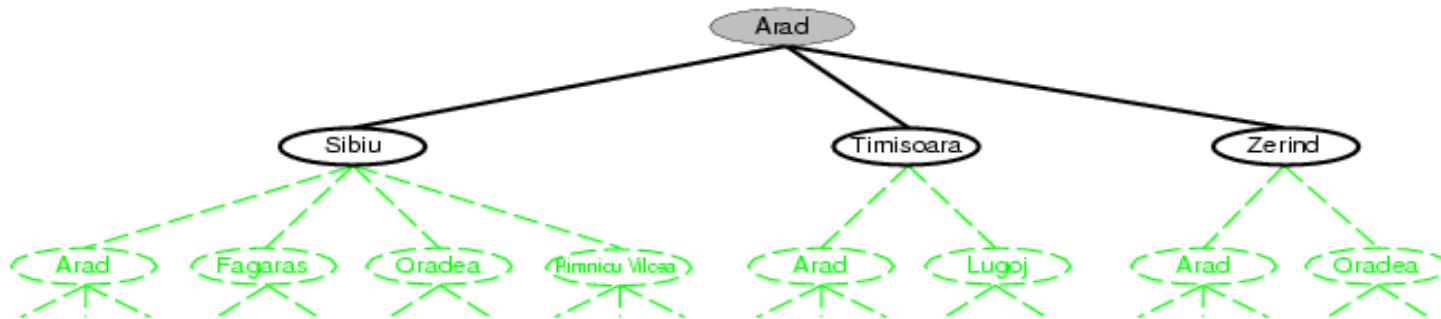
- Basic idea:
 - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. ~expanding states)

```
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
```

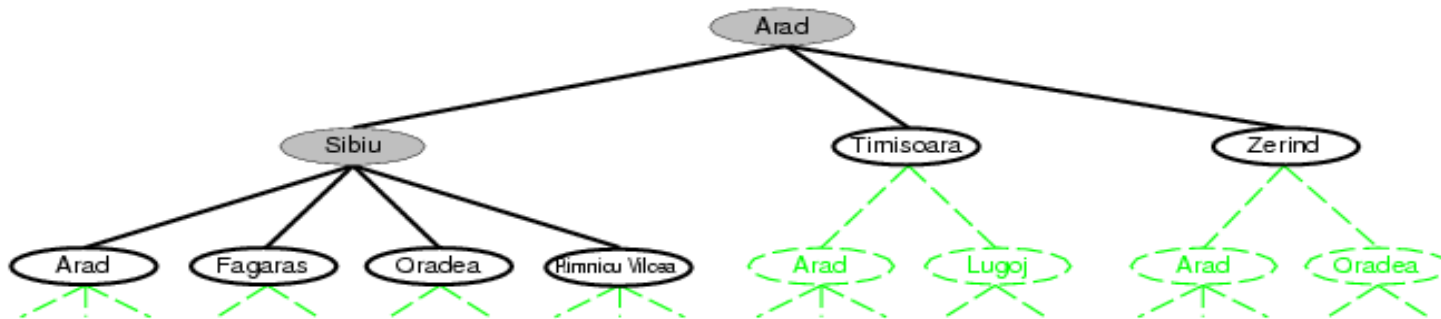
Tree search example



Tree search example



Tree search example





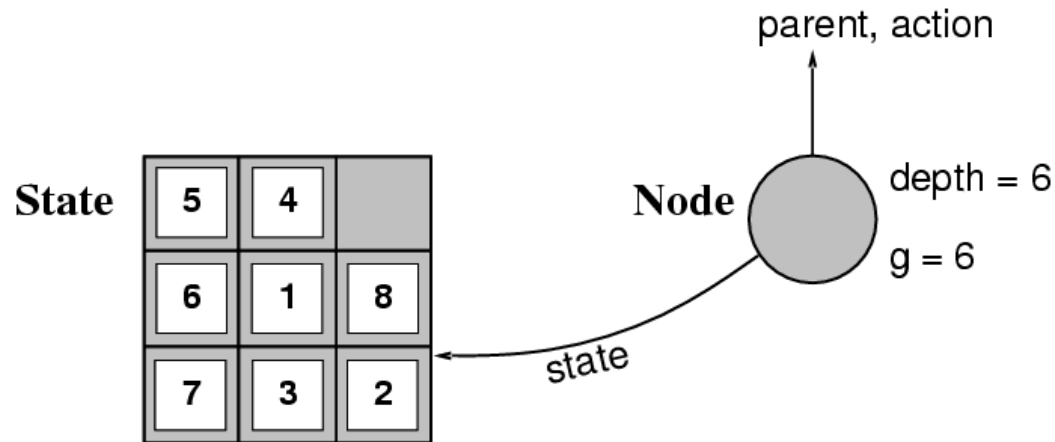
Implementation: general tree search

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

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

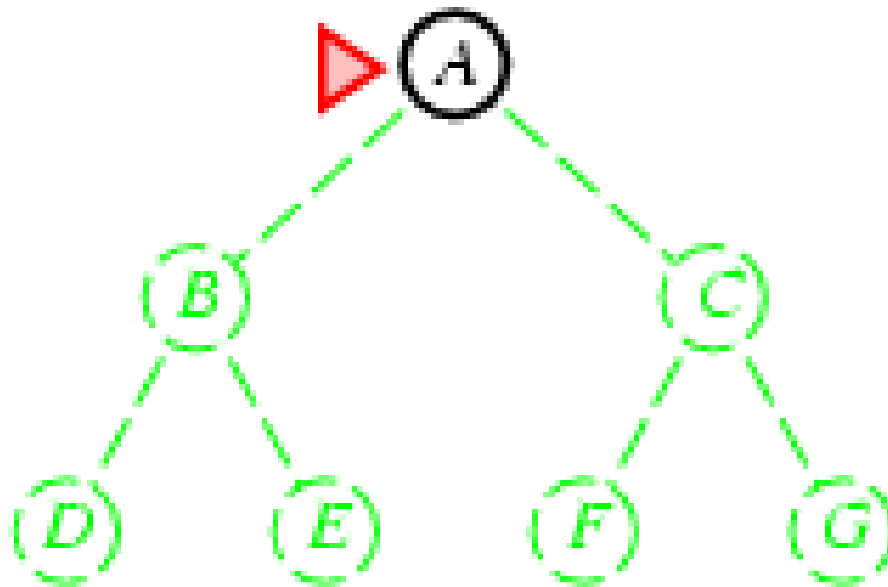


Uninformed search strategies

- **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

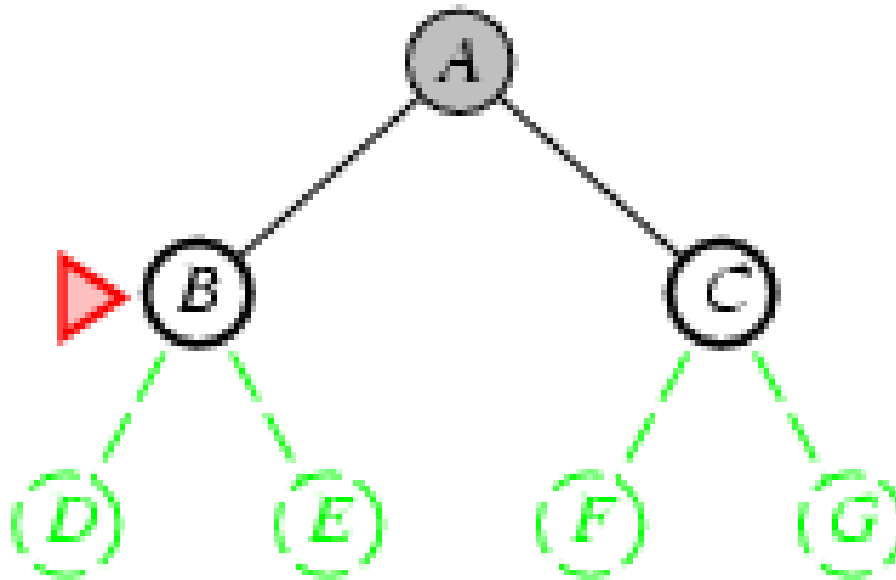
Breadth-first search

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



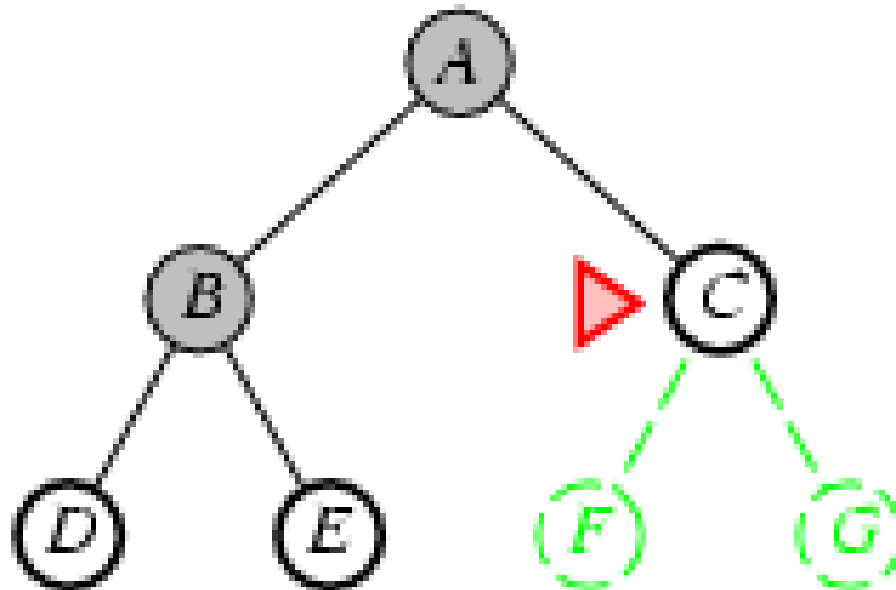
Breadth-first search

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



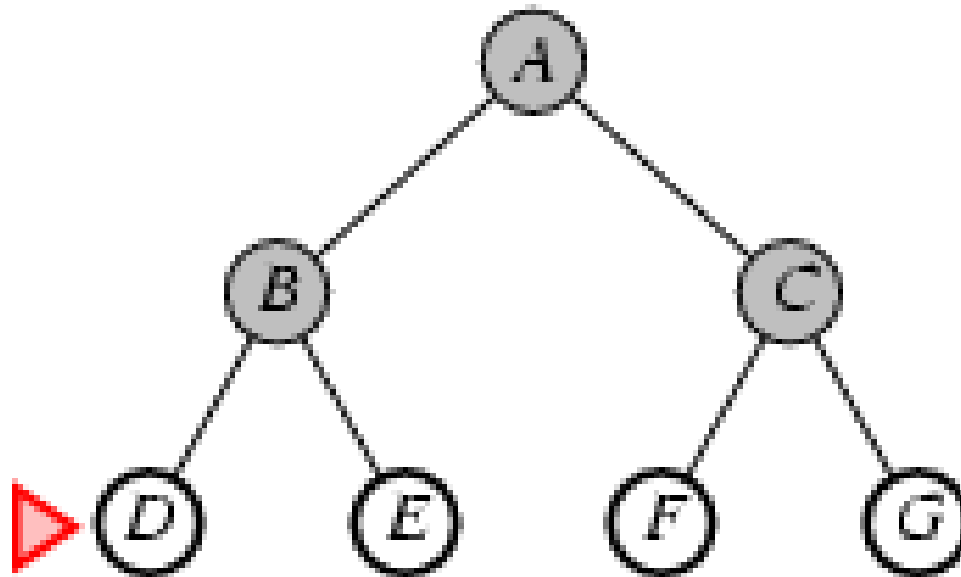
Breadth-first search

- 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





Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- **Space** is the bigger problem (more than time)

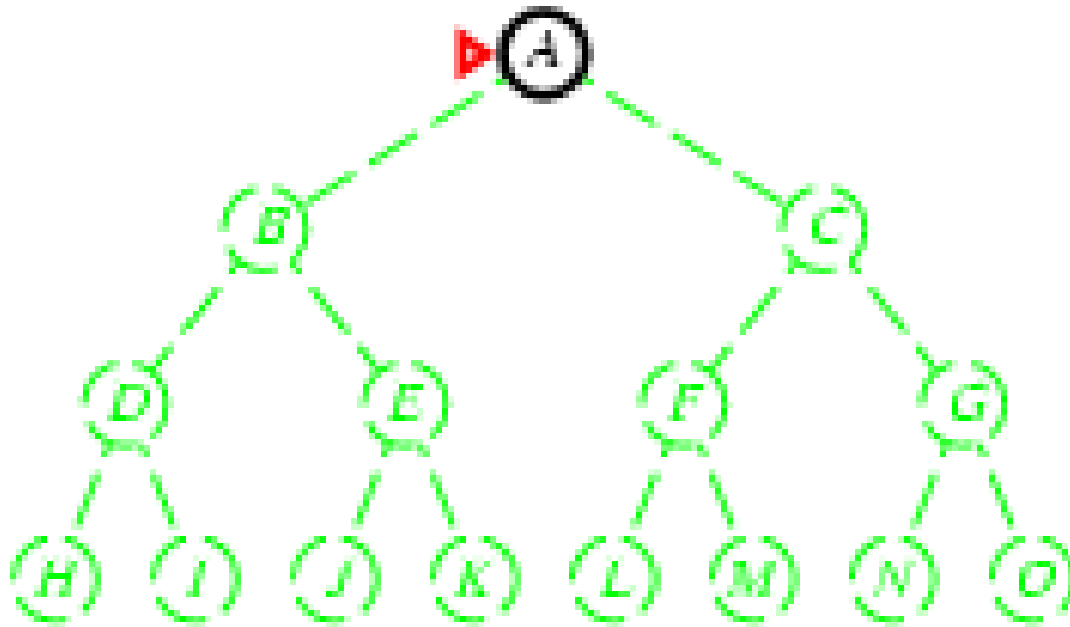


Uniform-cost search

- 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 $\geq \epsilon$
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\epsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\epsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

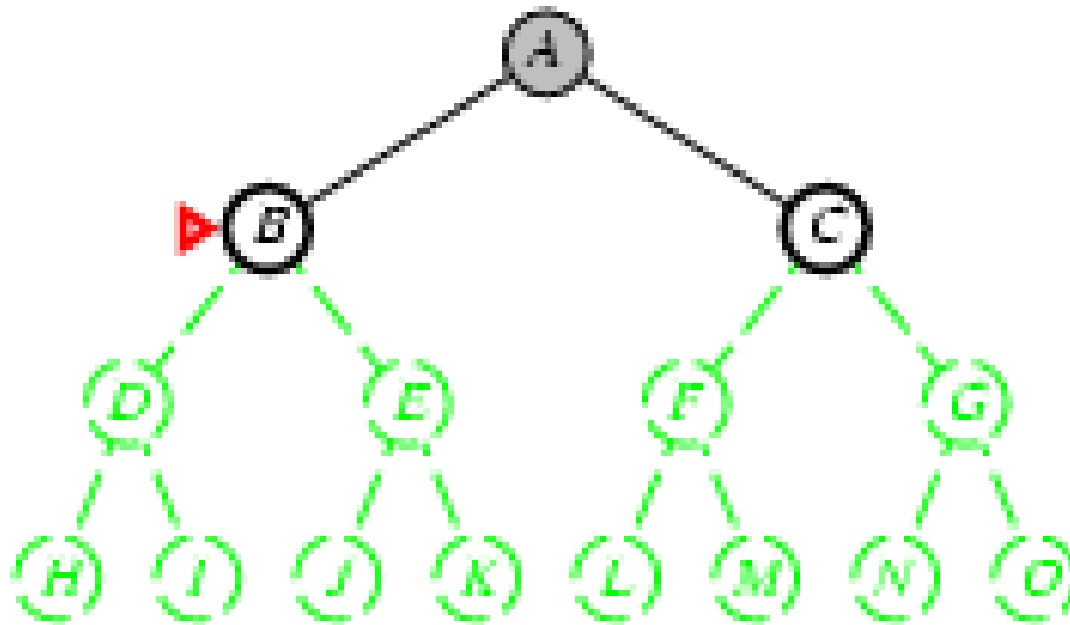
Depth-first search

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



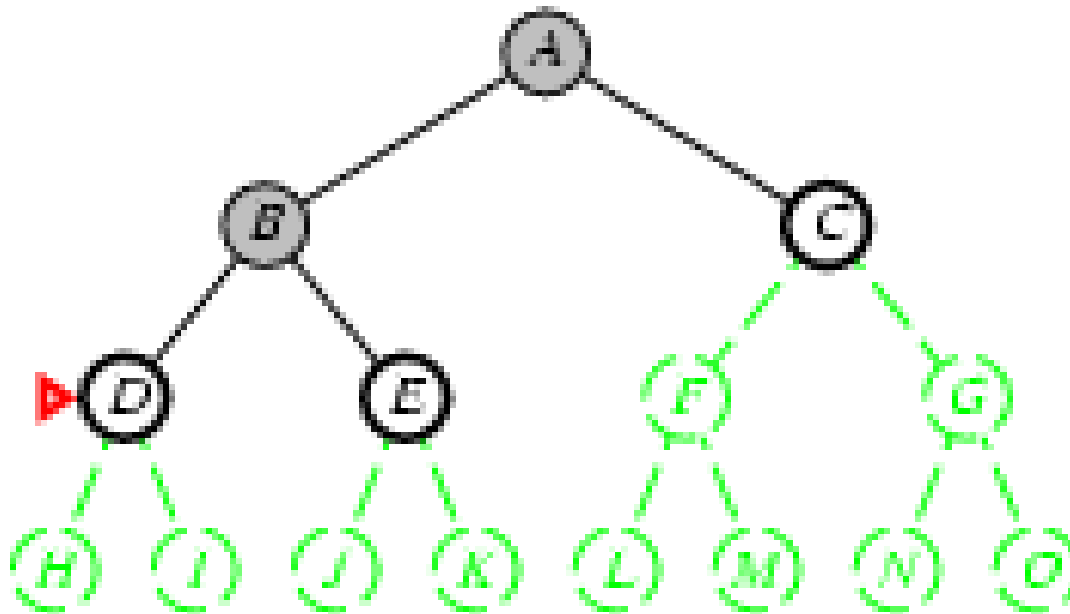
Depth-first search

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



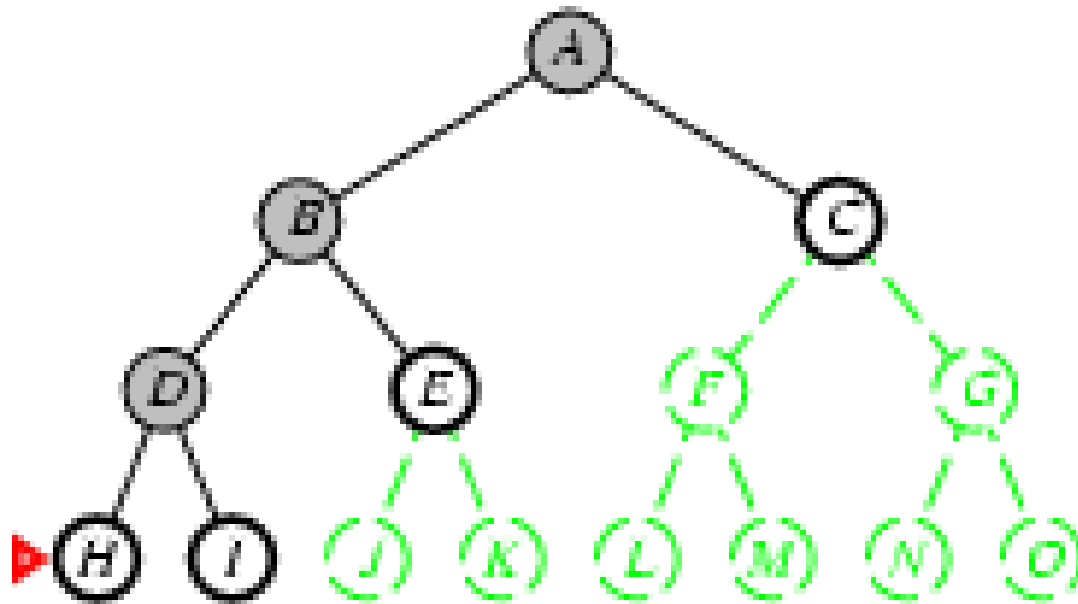
Depth-first search

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



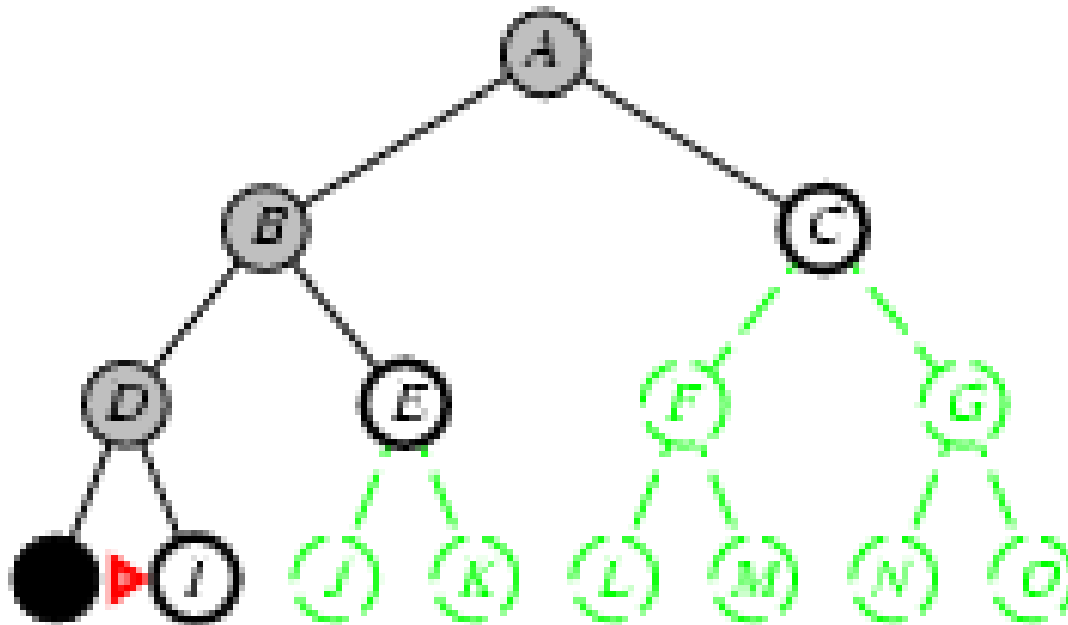
Depth-first search

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



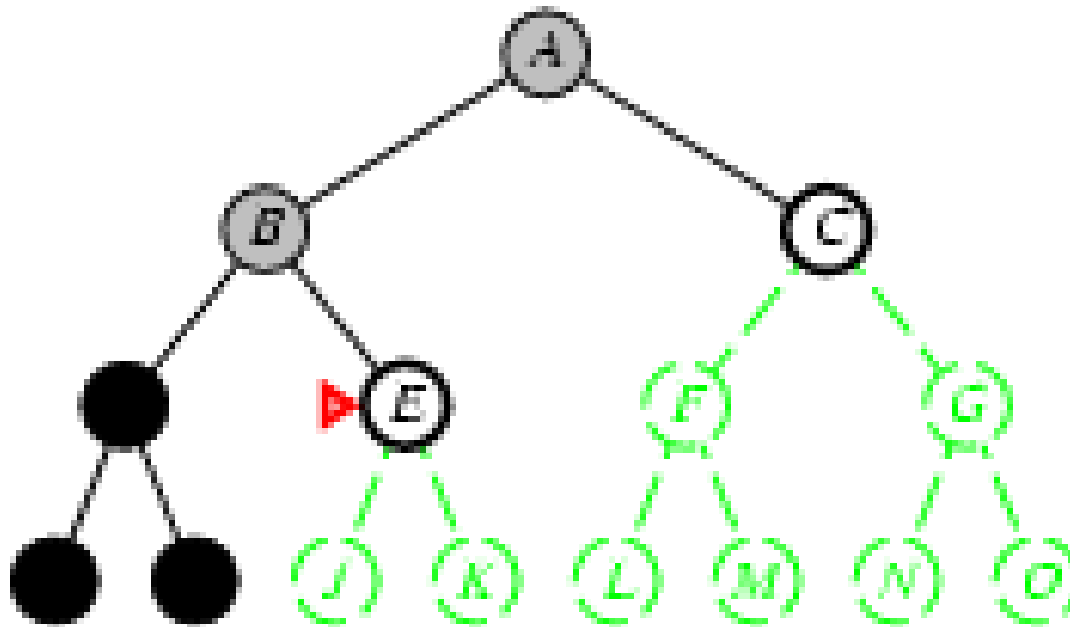
Depth-first search

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



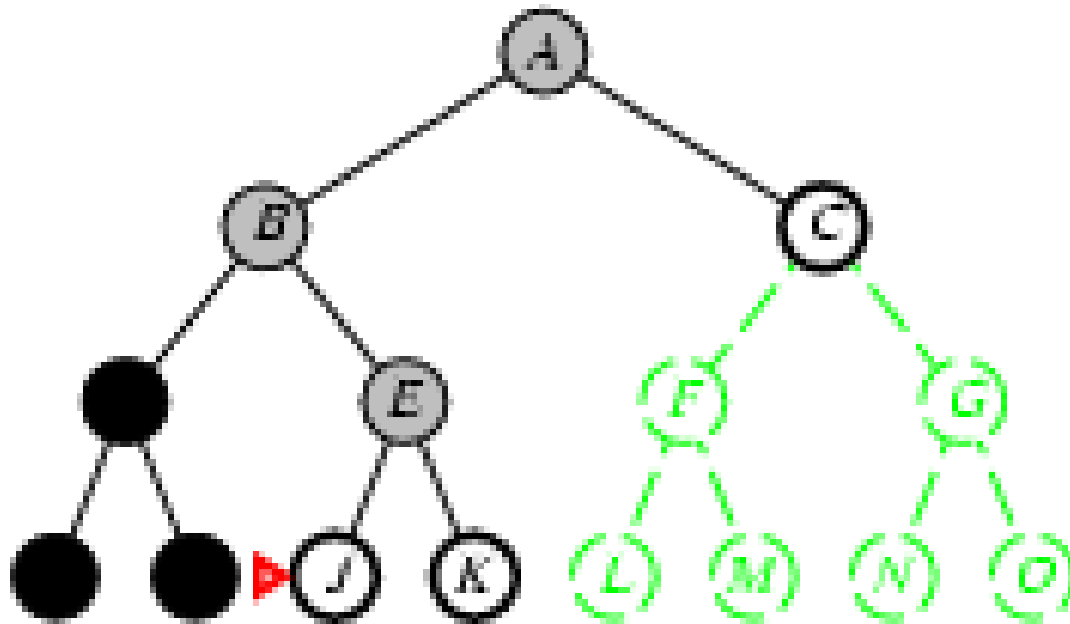
Depth-first search

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



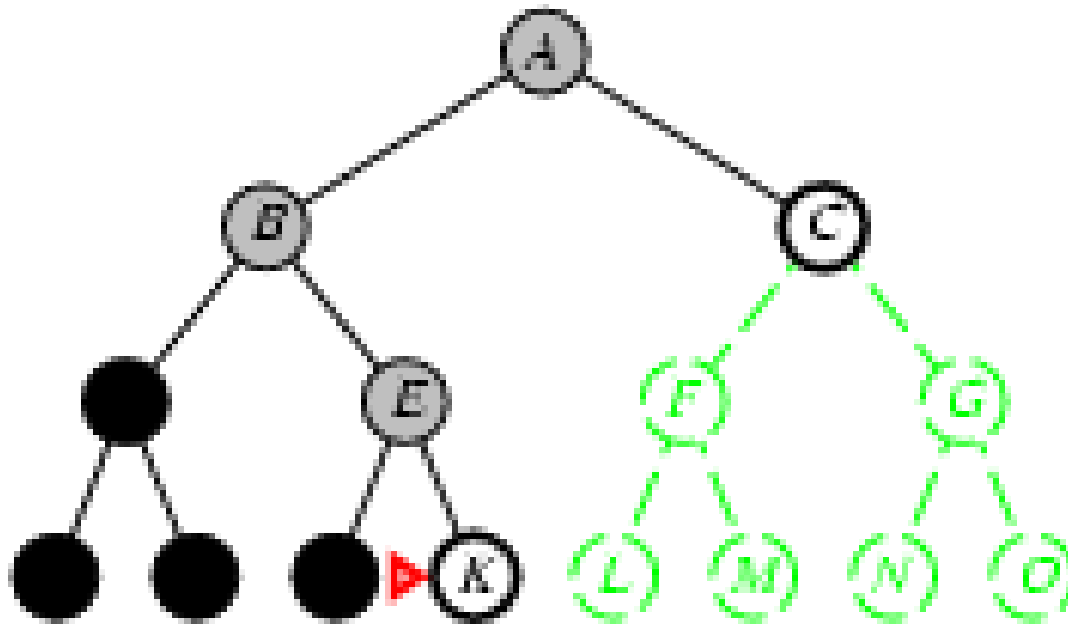
Depth-first search

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



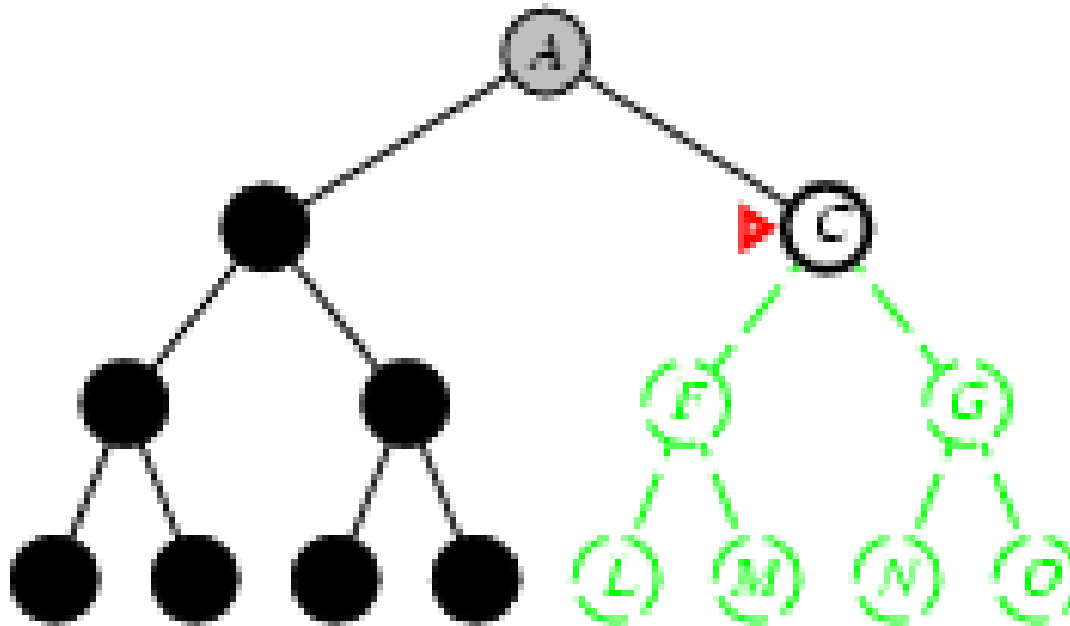
Depth-first search

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



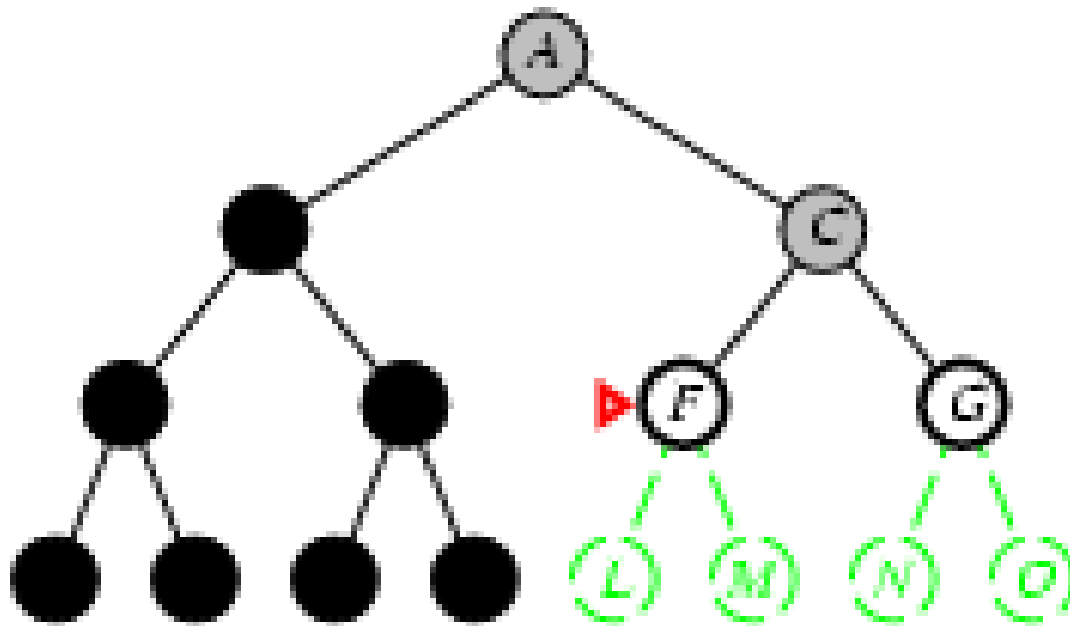
Depth-first search

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



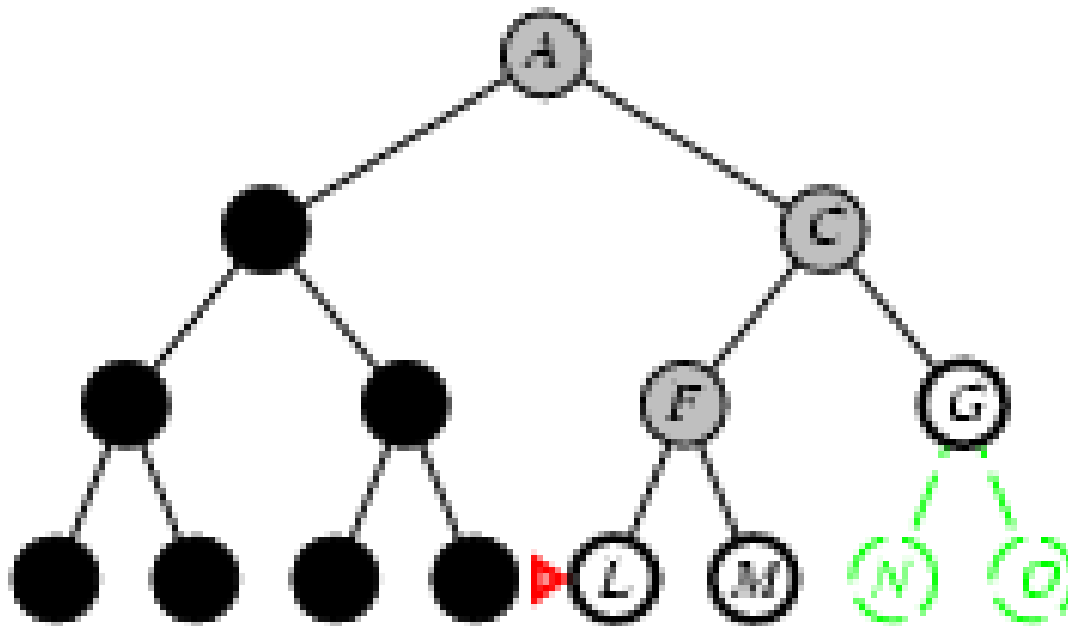
Depth-first search

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



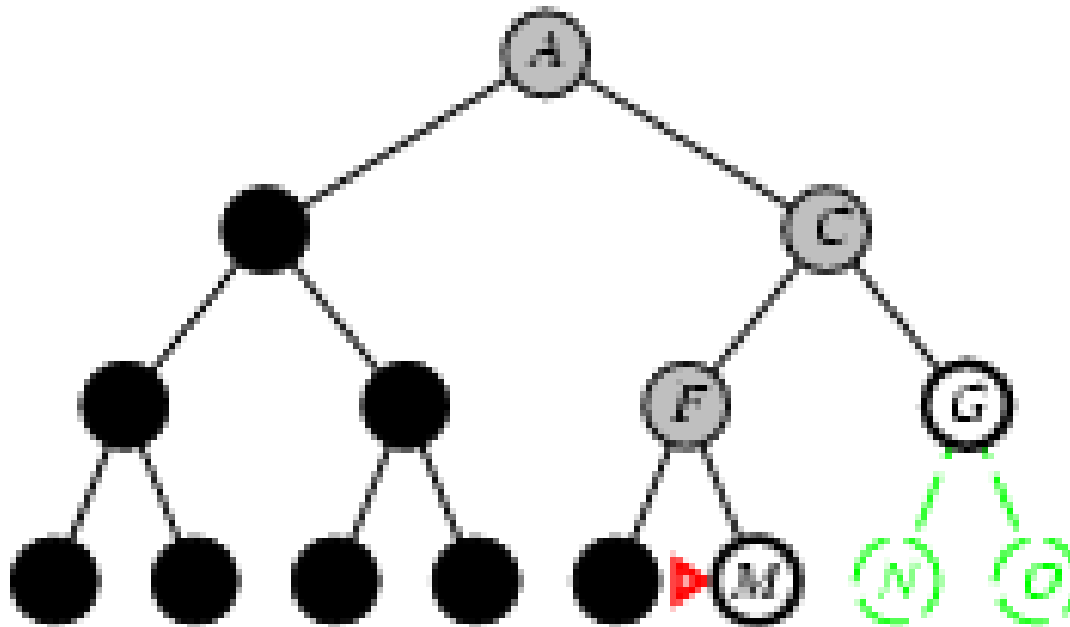
Depth-first search

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



Depth-first search

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





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-limited search

- = **depth-first search** with depth limit (i.e., nodes at depth l have no successors)
- In practice, one iteratively increases $l = 1..d$ up to some “final depth” d
 - 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?  $\leftarrow$  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  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search / =0

Limit = 0



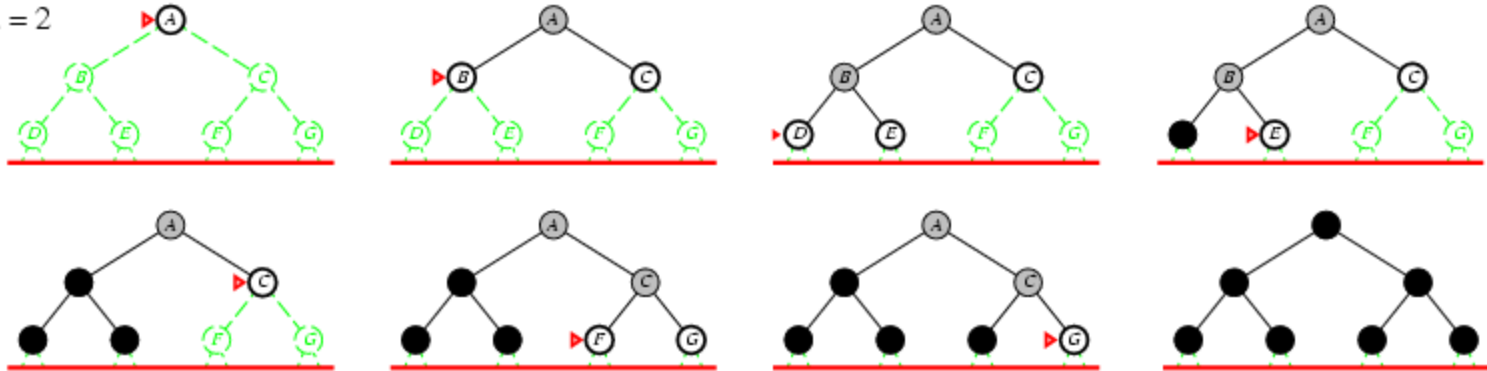
Iterative deepening search / = 1

Limit = 1



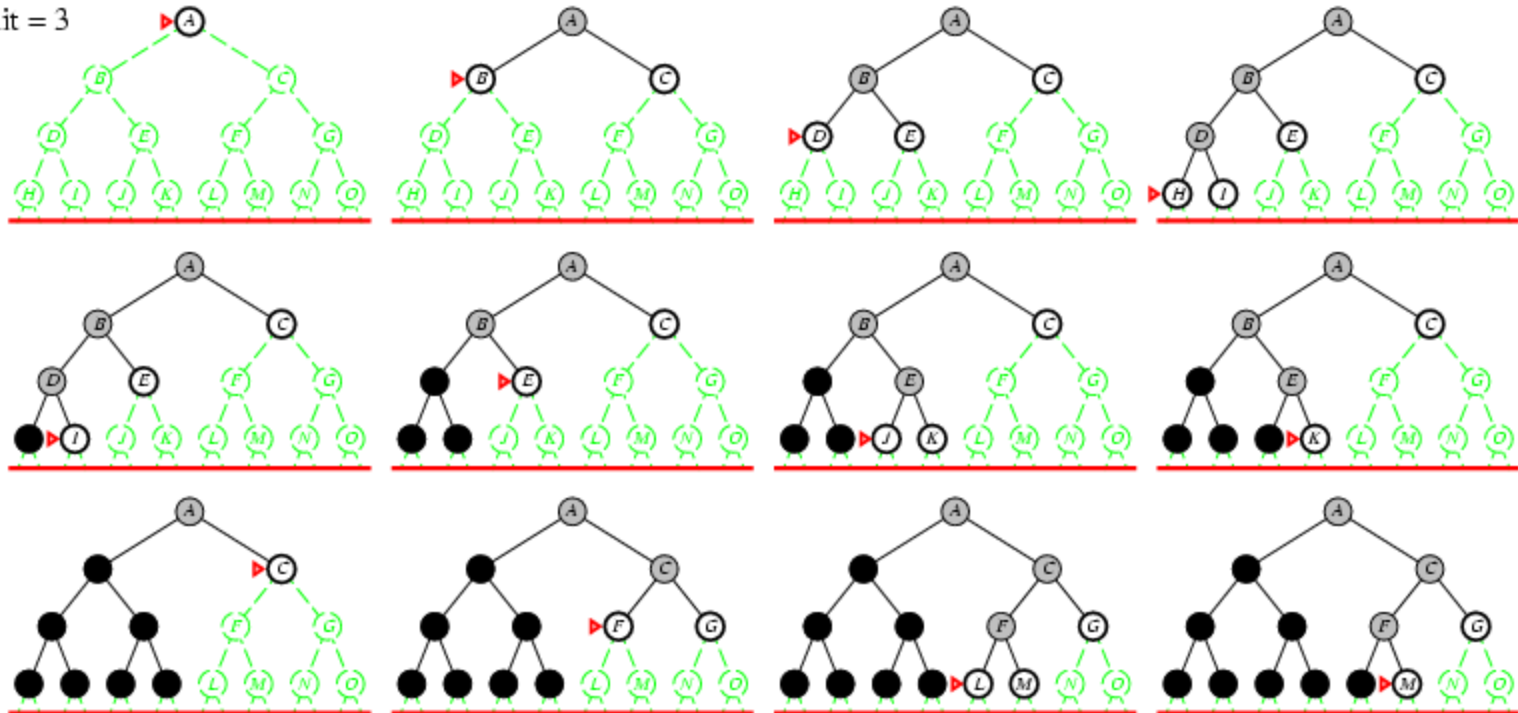
Iterative deepening search / =2

Limit = 2



Iterative deepening search / =3

Limit = 3



Iterative deepening search

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

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

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

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

- 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

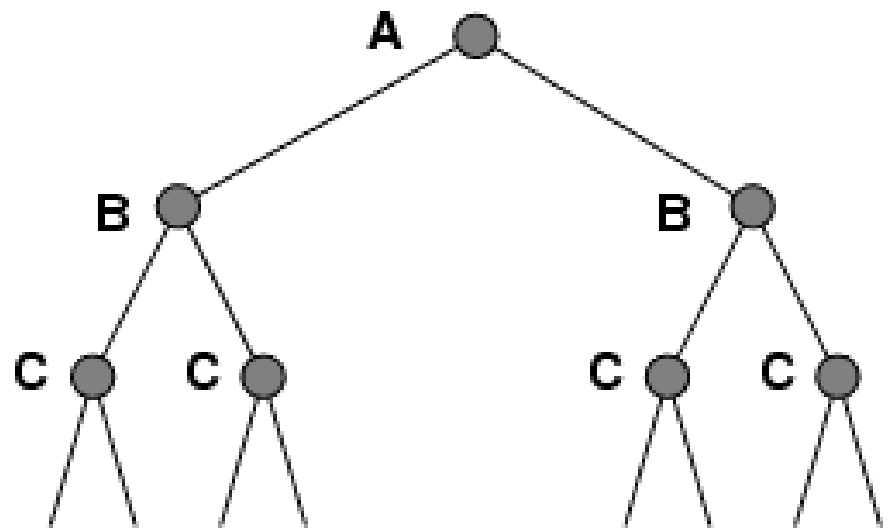
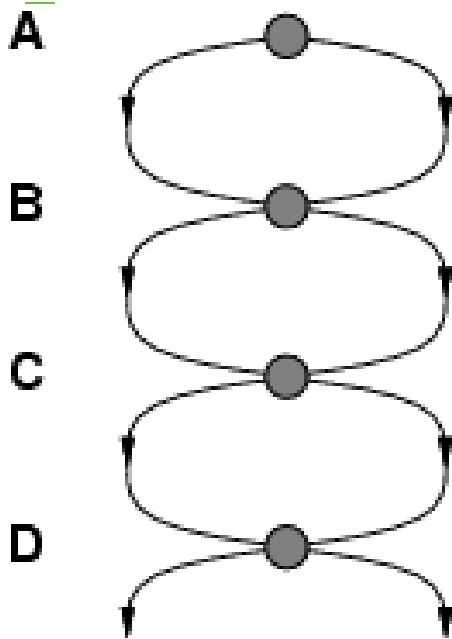
- Complete? Yes
- Time? $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

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 \leftarrow an empty set

fringe \leftarrow INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)

loop do

if *fringe* is empty **then return** failure

node \leftarrow 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 \leftarrow INSERTALL(EXPAND(*node*, *problem*), *fringe*)



Summary

- 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 (an example of a good space-time tradeoff)