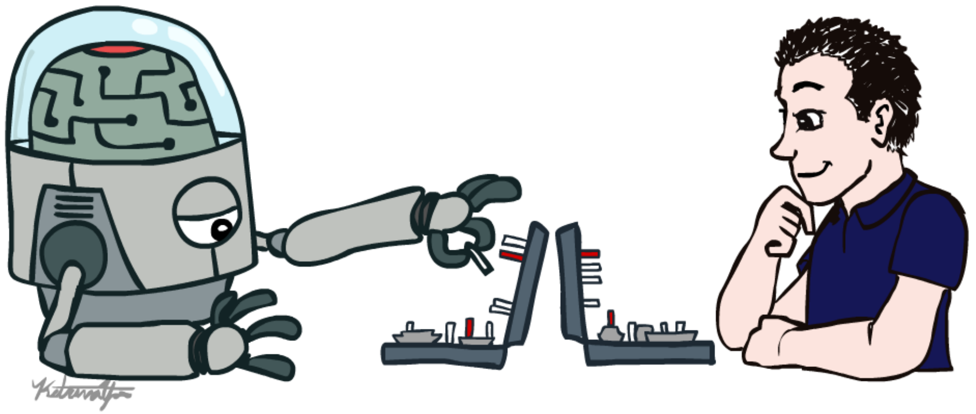
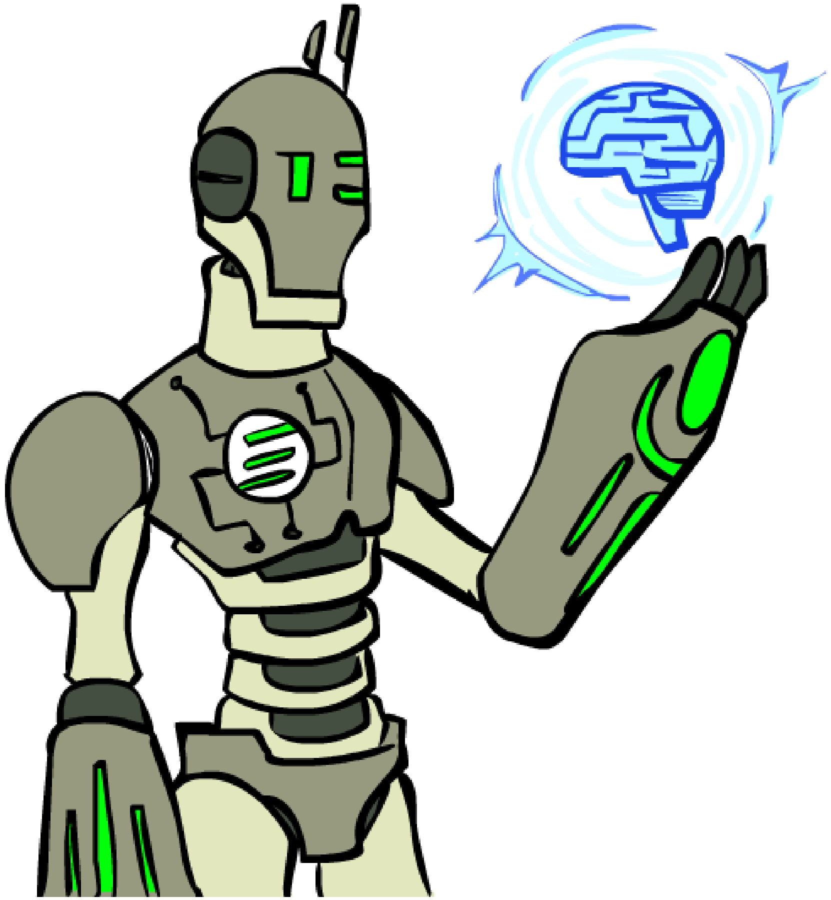
Lecture 02



Ashis Kumar Chanda
chanda@rowan.edu

Rowan University

Today



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

Problem-solving agents

Function Sim_Prob_solv_agent (*percept*) return an *action*

static: *seq*, an action sequence, initially empty

state, some description of the current world state

goal, a goal, initially null (based on current situation and PM)

problem, a problem formulation (what action and state to consider, given a goal)

state ← Update_Sate (*state*, *percept*)

If *seq* is empty then do

goal ← Formulate_Goal (*state*)

problem ← Formulate_problem (*state, goal*)

seq ← Search (*problem*)

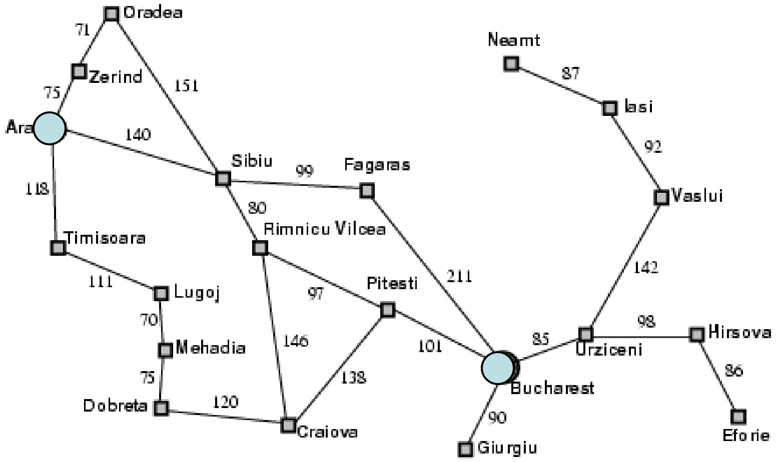
action ← First (*seq*)

seq ← Rest (*seq*)

Return *action*

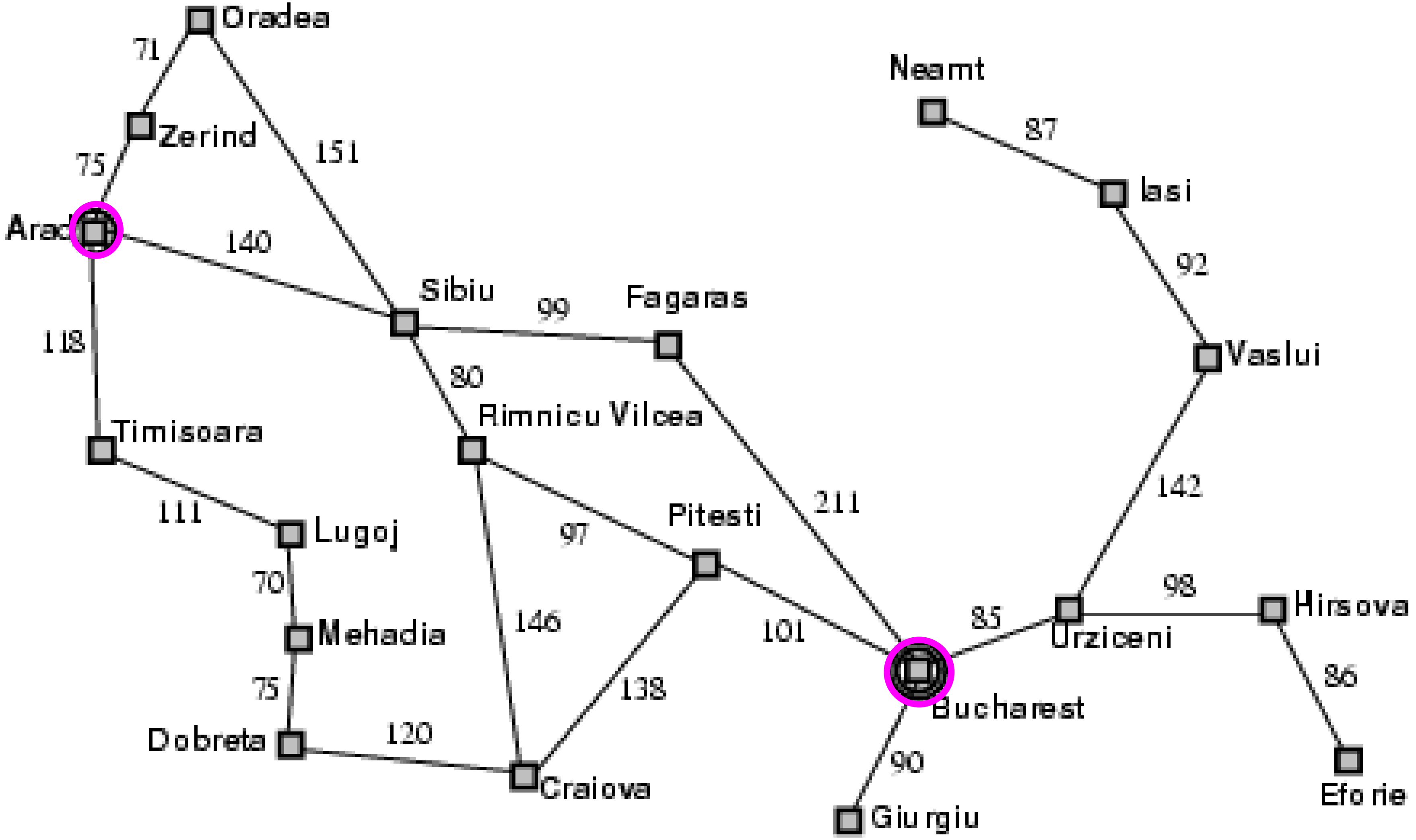
Example: Romania

- Romania is a southeastern European country.
- Suppose, on holiday in Romania; you are currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest



function SEARCH (*problem*) return *solution* (as a sequence)

Example: Romania



Search problem formulation

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

1. initial state e.g., "at Arad“

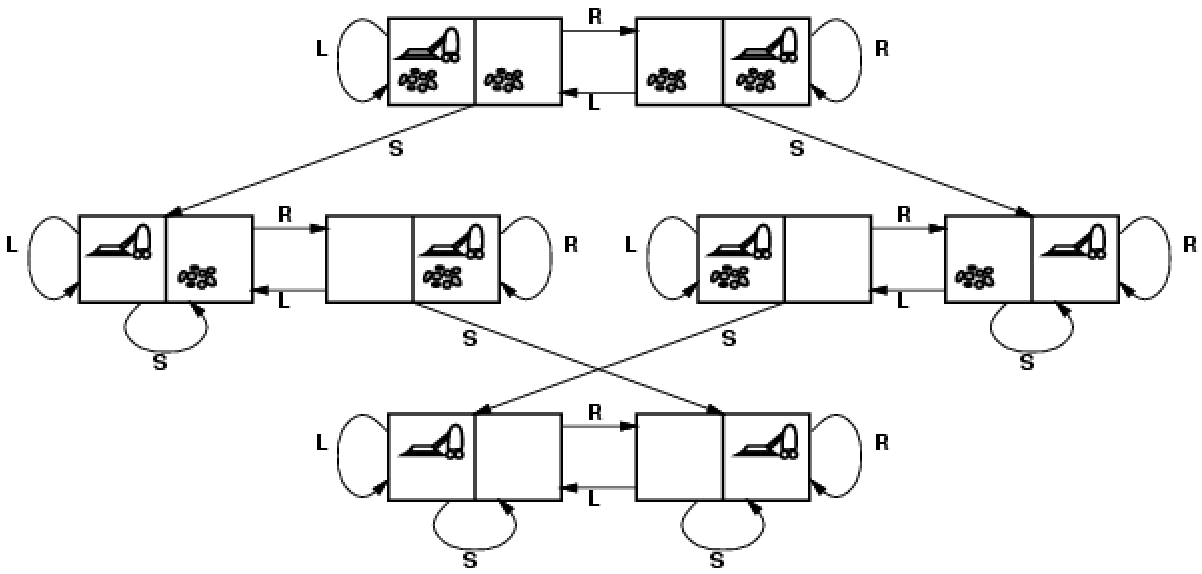
2. **actions** or successor function $S(x)$ = set of action–state pairs
   - e.g., $S(Arad)$ = {<*Arad* → *Zerind*, *Zerind*>, … }

3. **goal test**, can be
   - explicit, e.g., x = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*

4. **path cost**
   - e.g., sum of distances, number of actions executed, etc.
   - $c(x,a,y)$ is the step cost, assumed to be ≥ 0

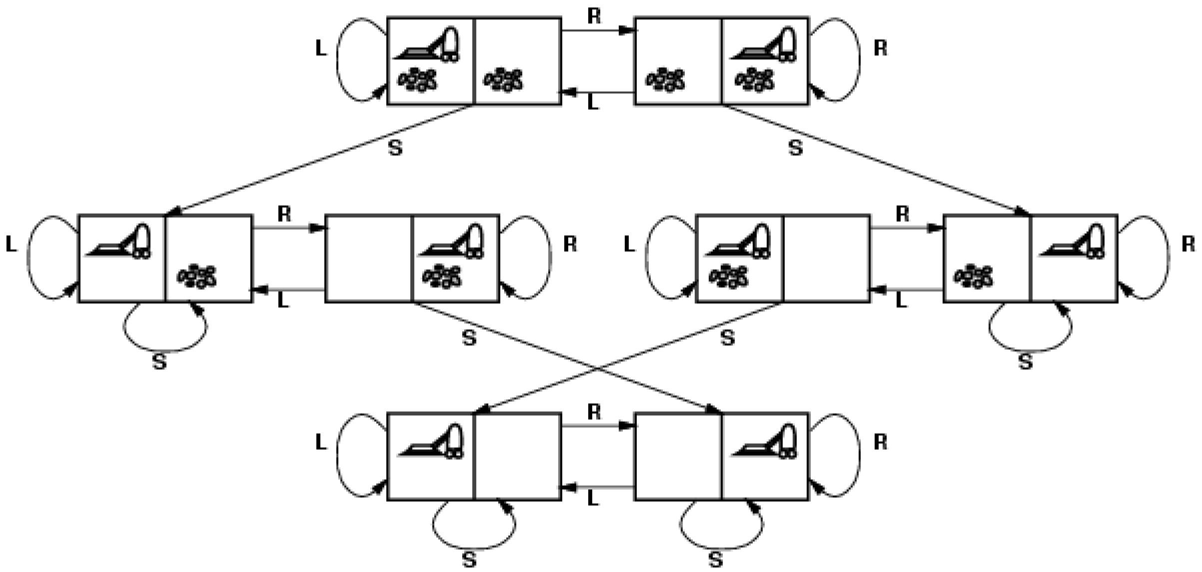
A **solution** is a sequence of actions leading from the initial state to a goal state

Vacuum world state space graph



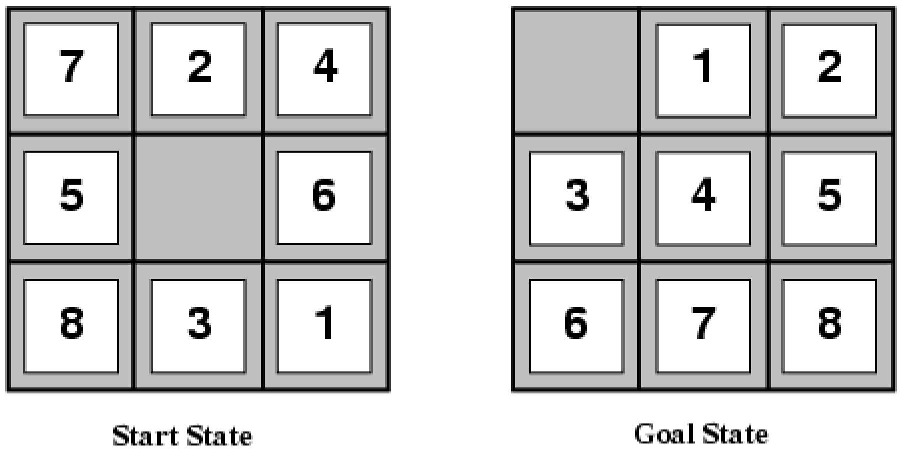
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



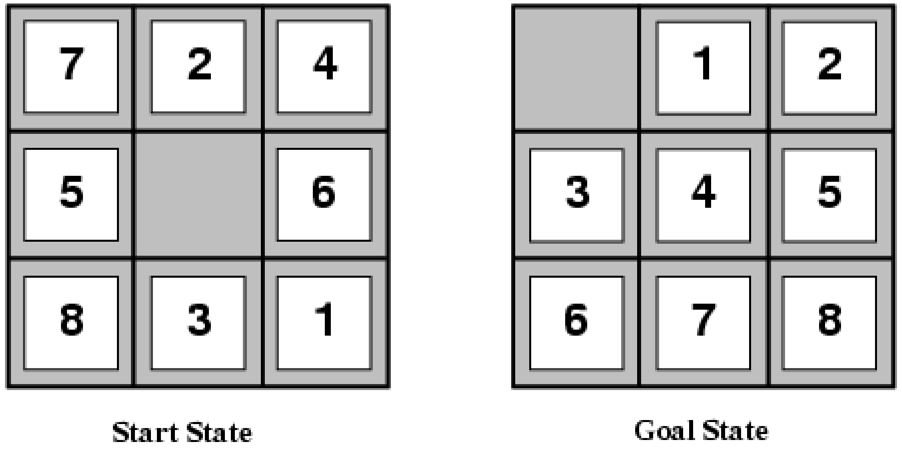
- states? dirt and robot location
- actions? *Left*, *Right*, *Clean*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



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

Example: The 8-puzzle



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

Tree search algorithms

- Basic idea:
  - offline, simulated 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

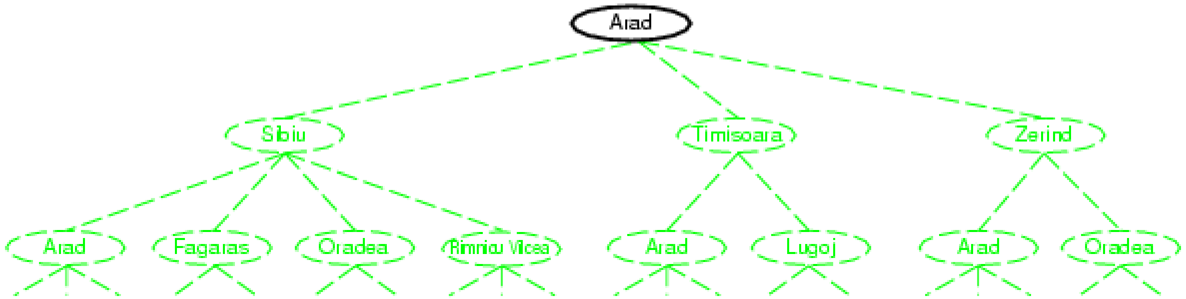
If there are no candidate for expansion then return *failure*

Choose a leaf node for expansion according to *strategy*

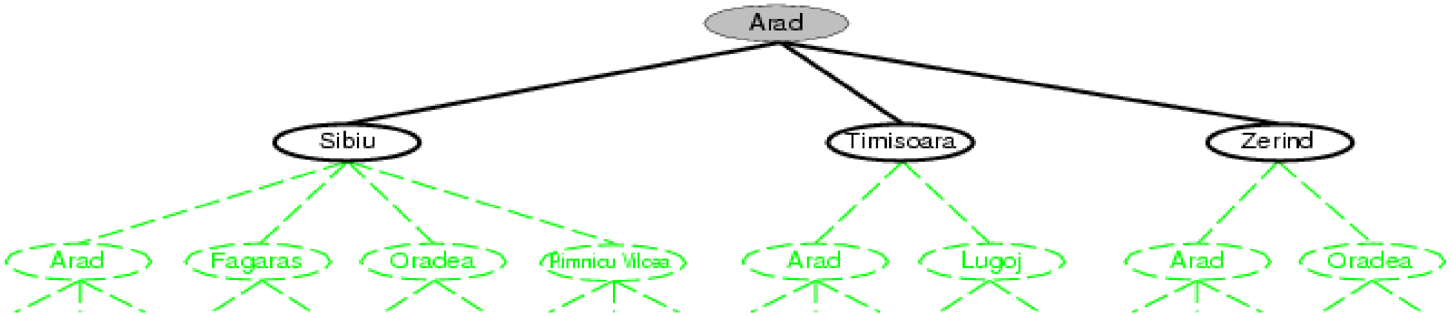
if the node contain a goal state then return the corresponding solution

else expand the node and add the resulting nodes to the search tree

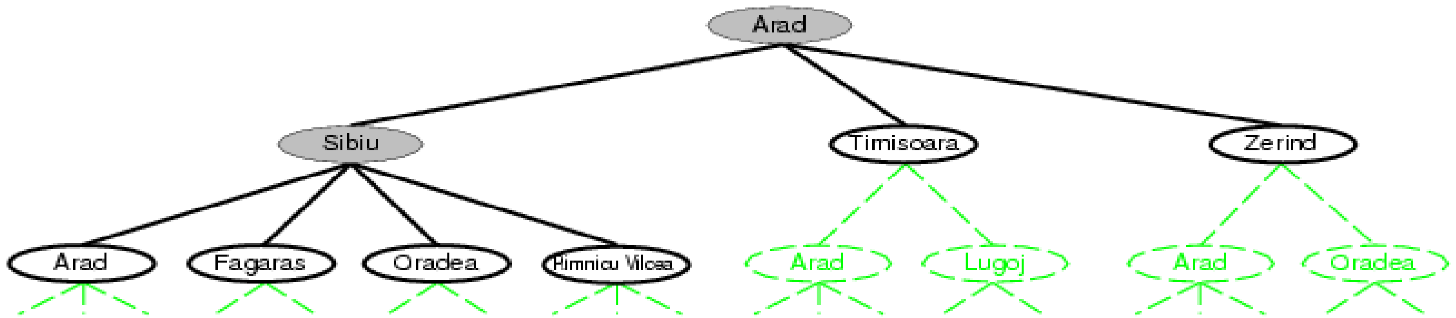
Tree search example



Tree search example

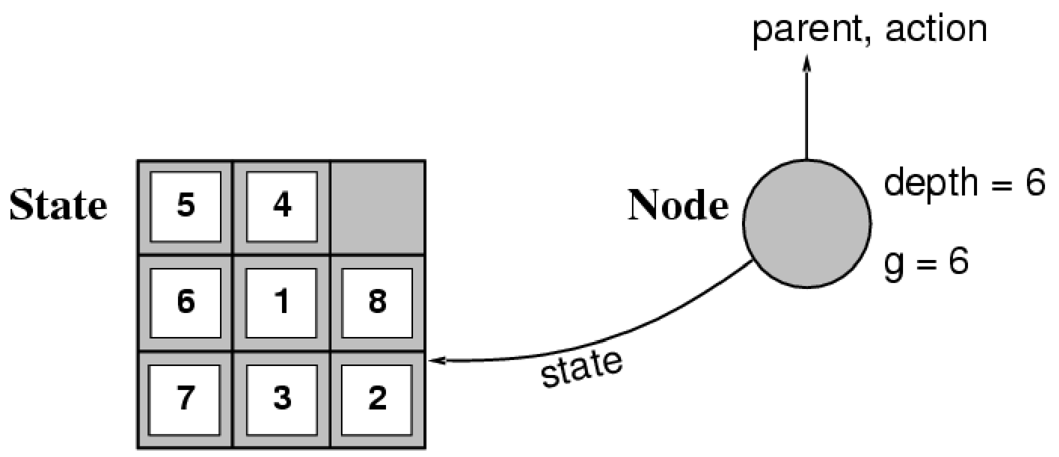


Tree search example



Implementation: 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



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 ∞)

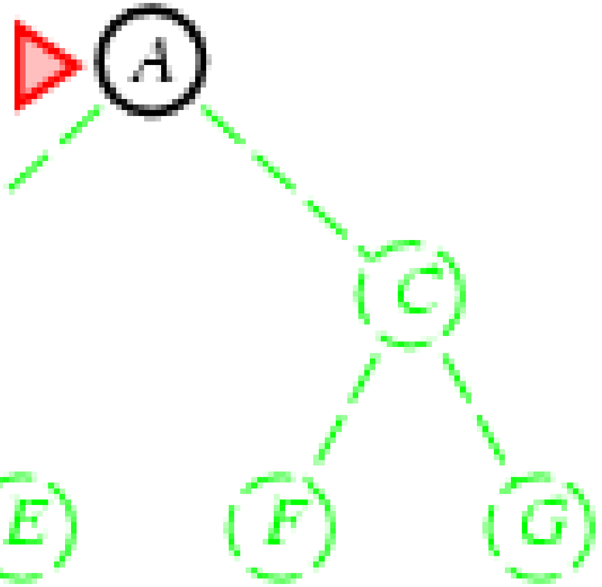
Uninformed search strategies

- Uninformed (blind) 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

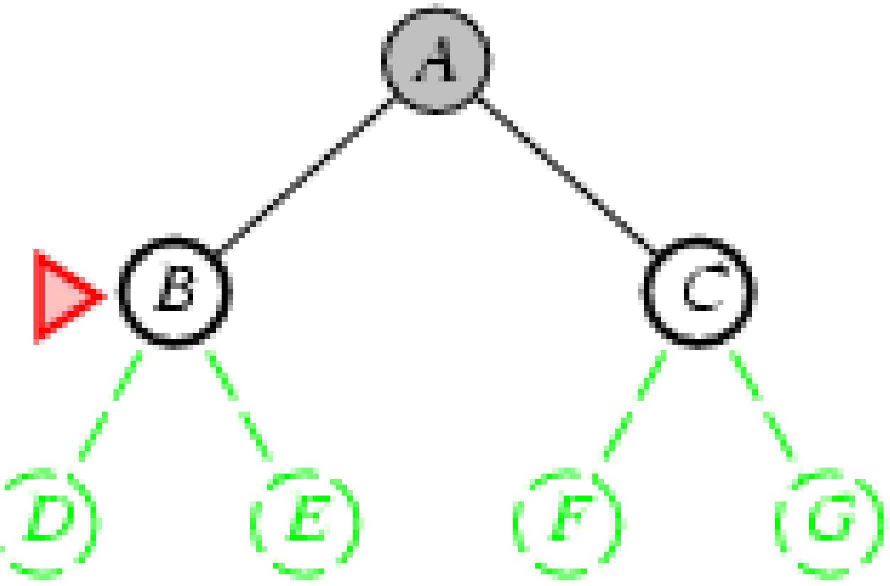
Breadth-first search

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



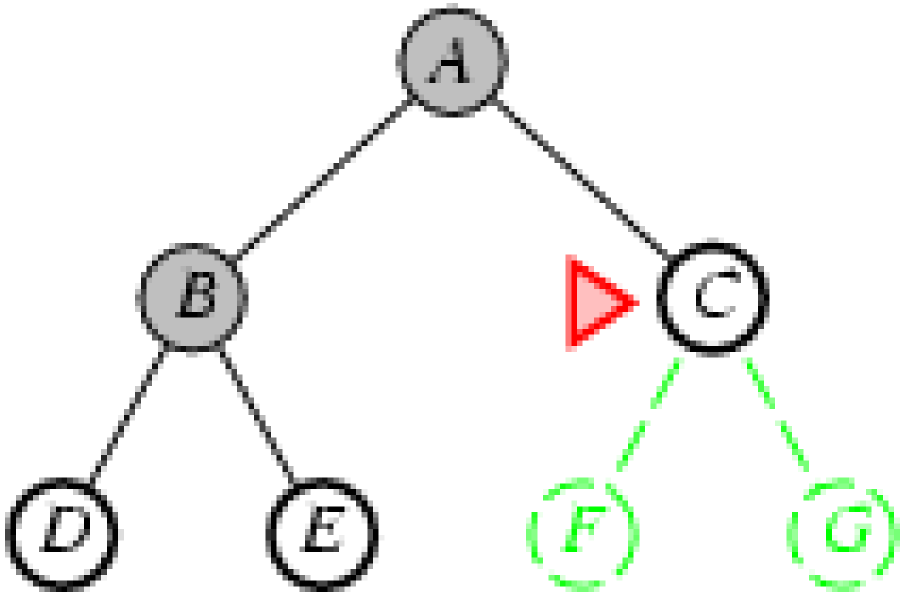
Breadth-first search

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



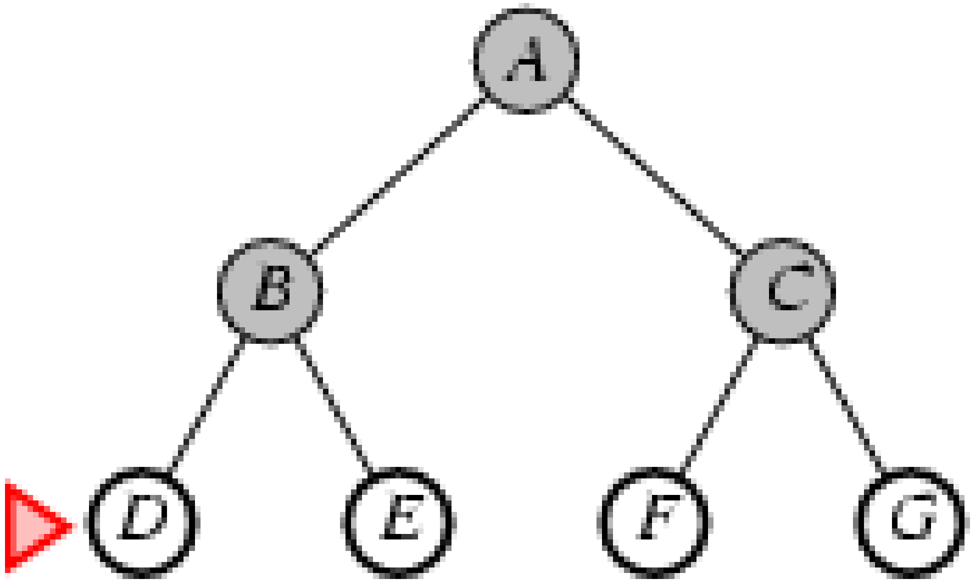
Breadth-first search

- 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



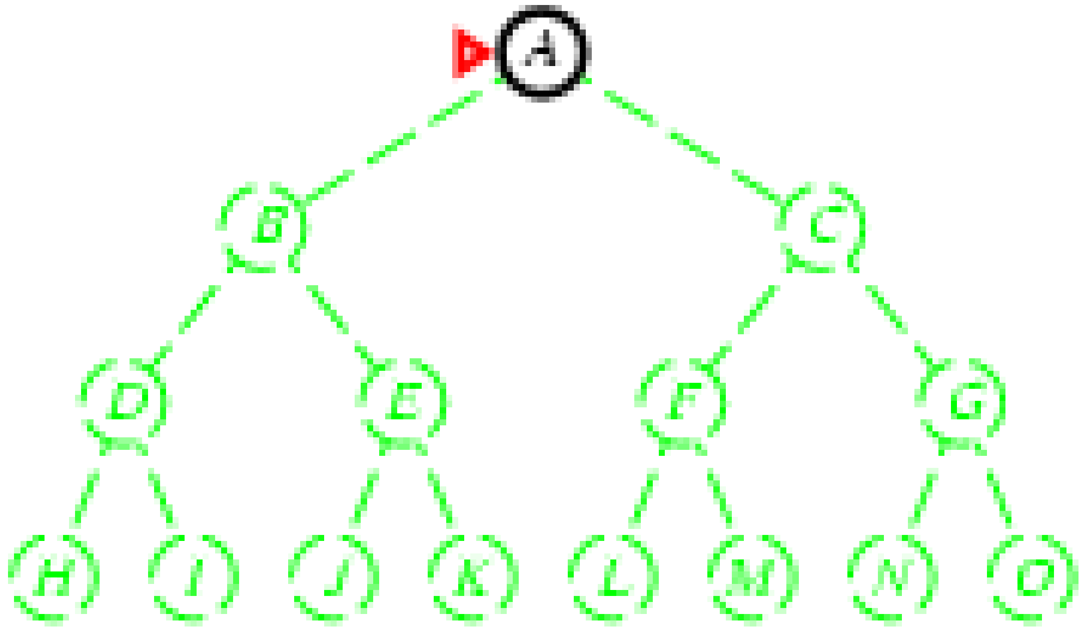
Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots +b^d + b(b^d-1) = O(b^{d+1})$
- 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)

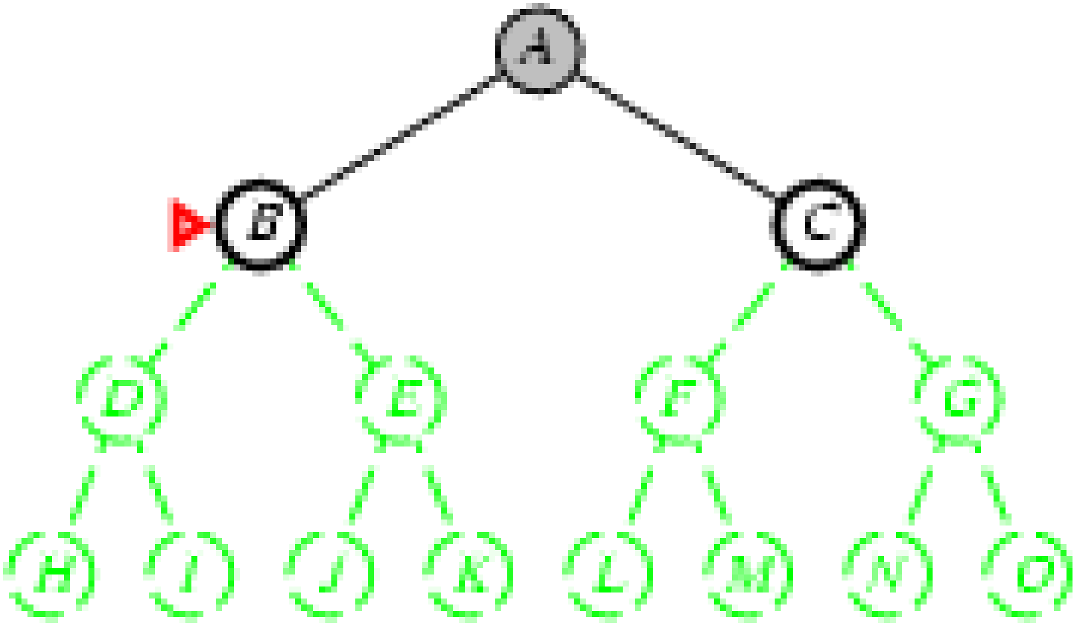
Depth-first search

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



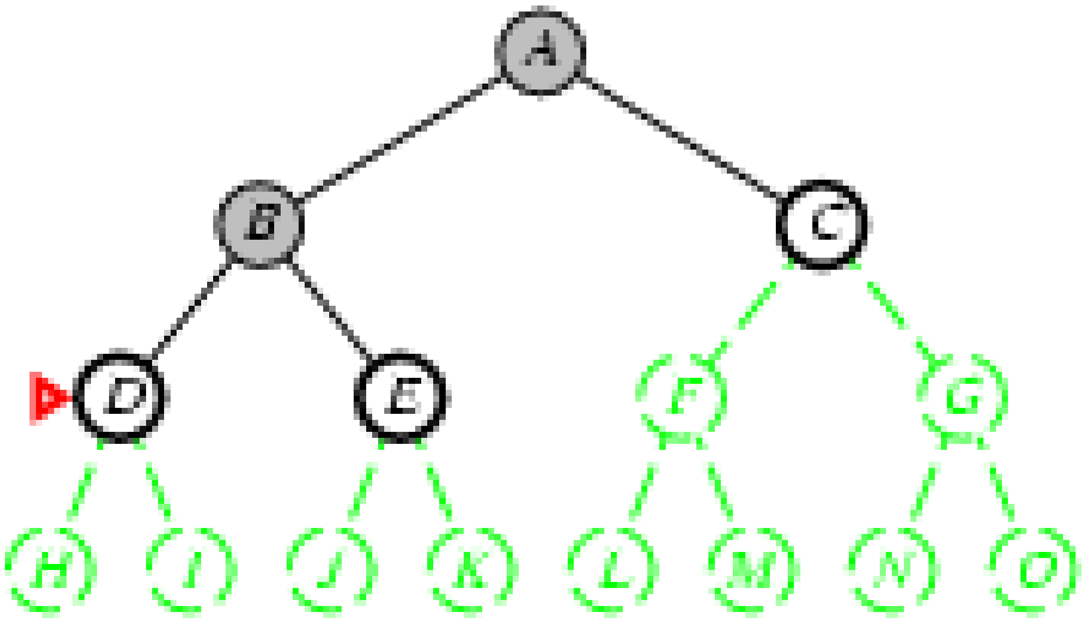
Depth-first search

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



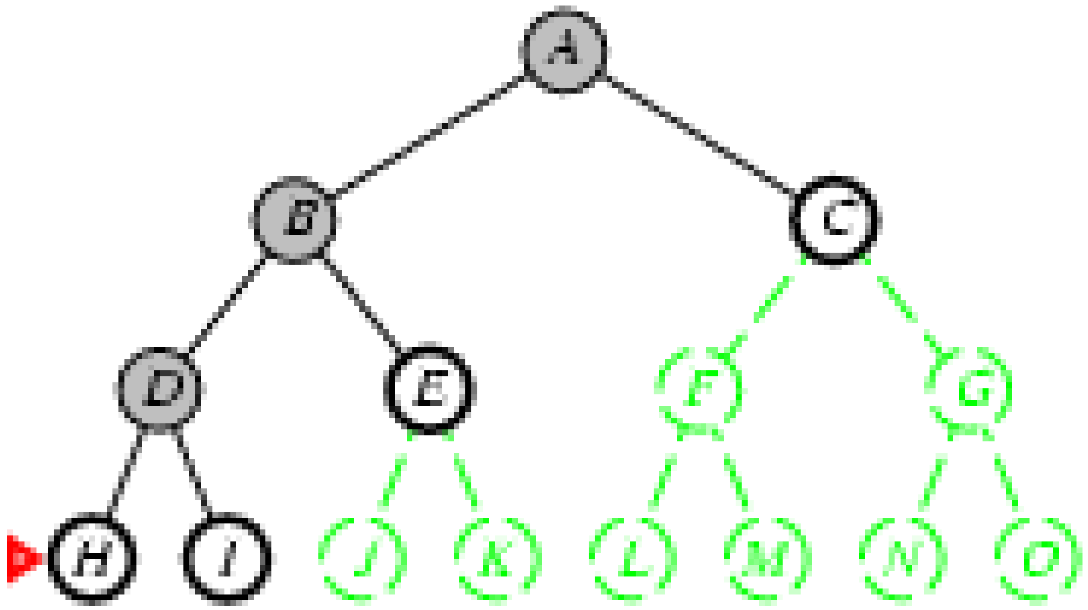
Depth-first search

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



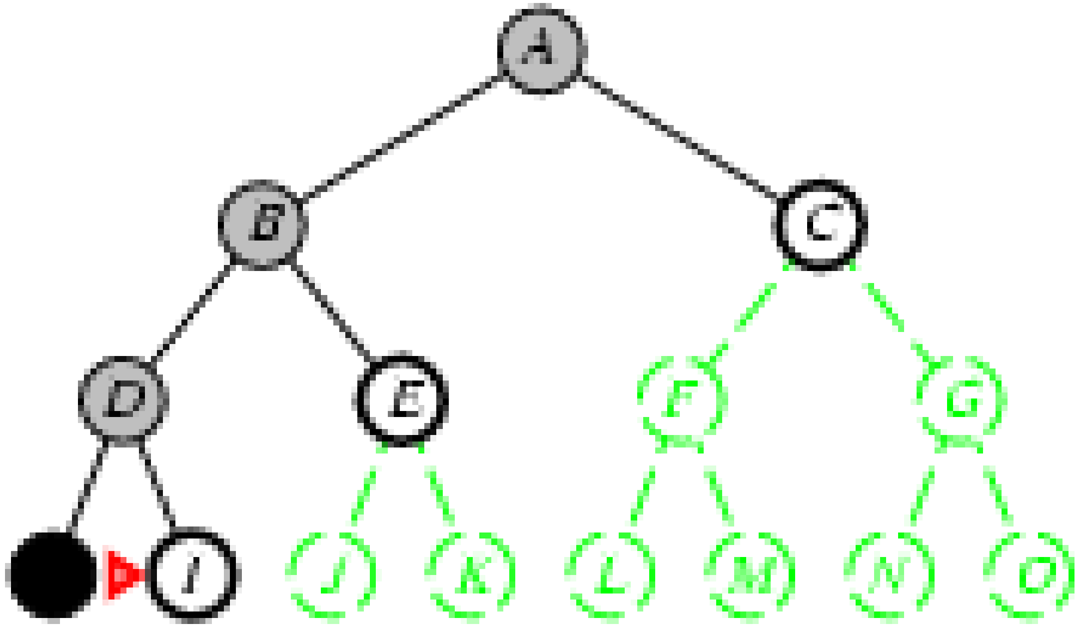
Depth-first search

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



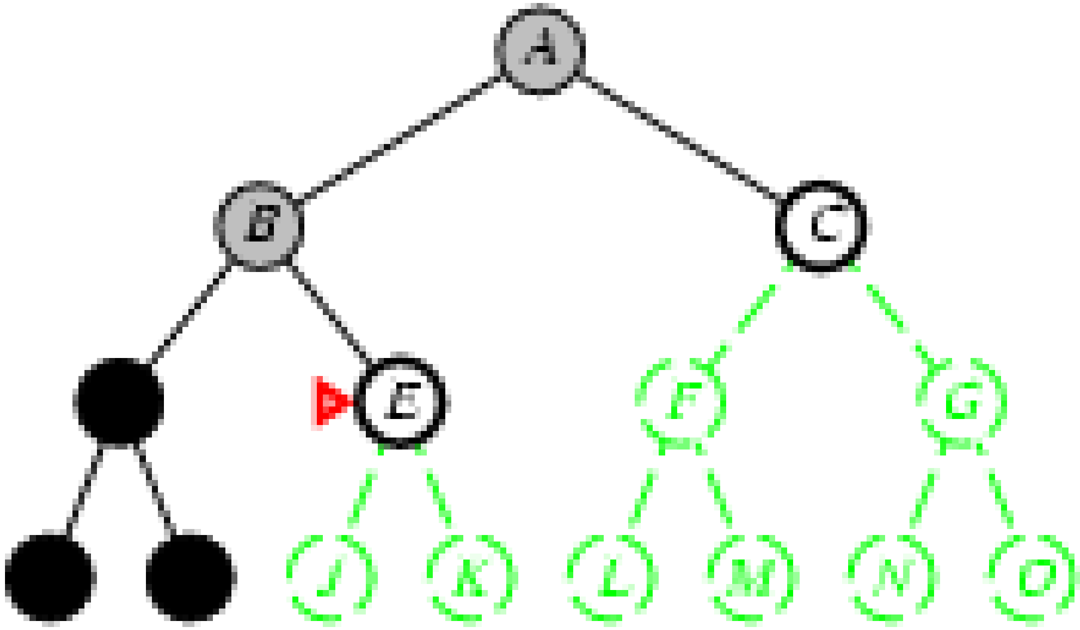
Depth-first search

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



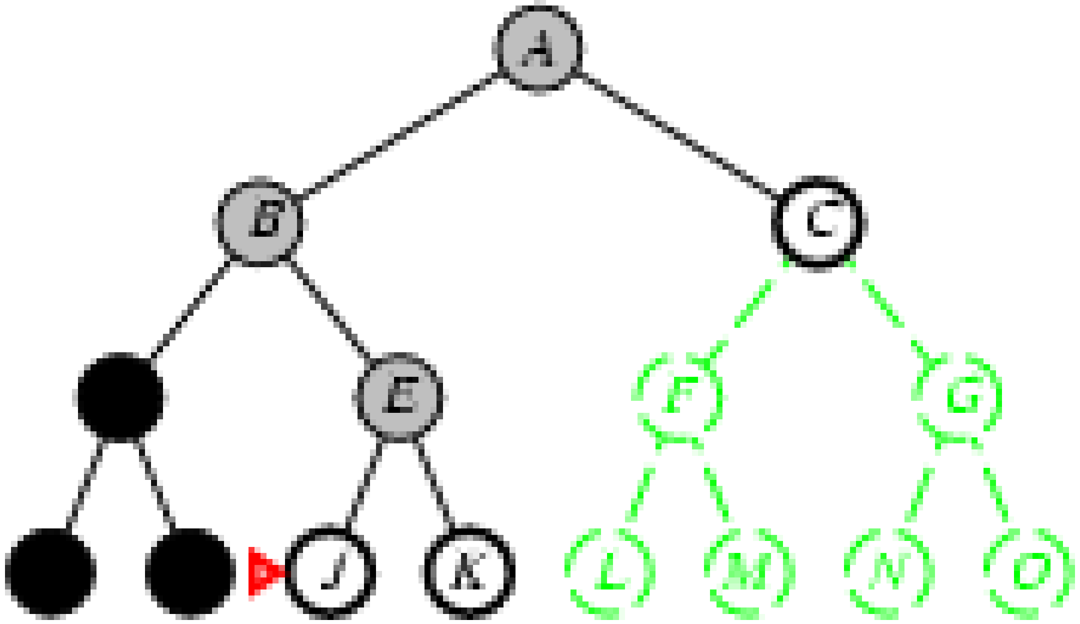
Depth-first search

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



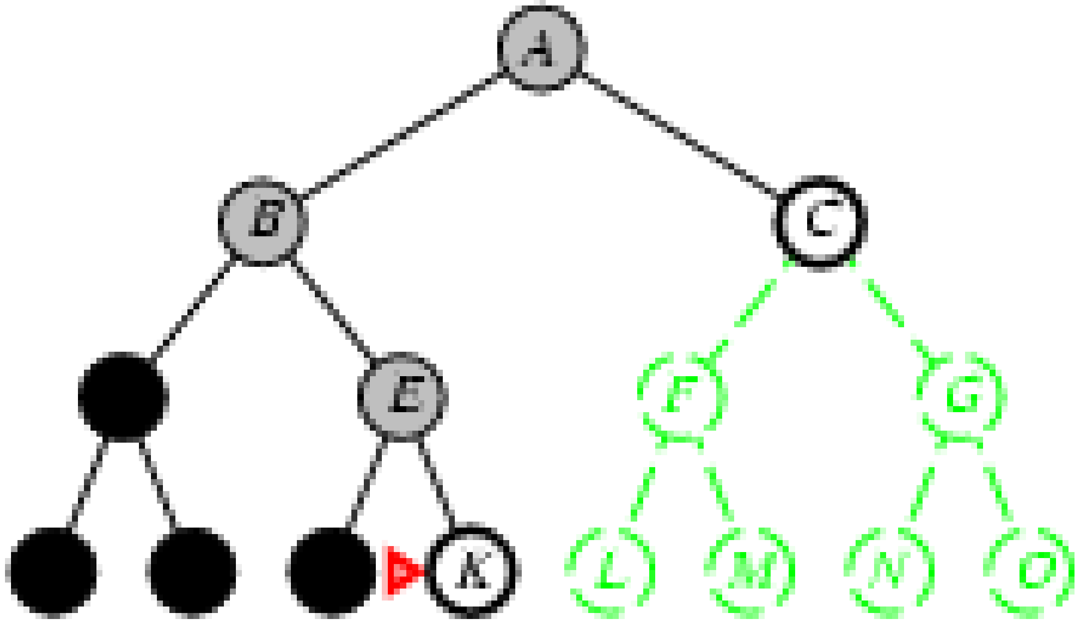
Depth-first search

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



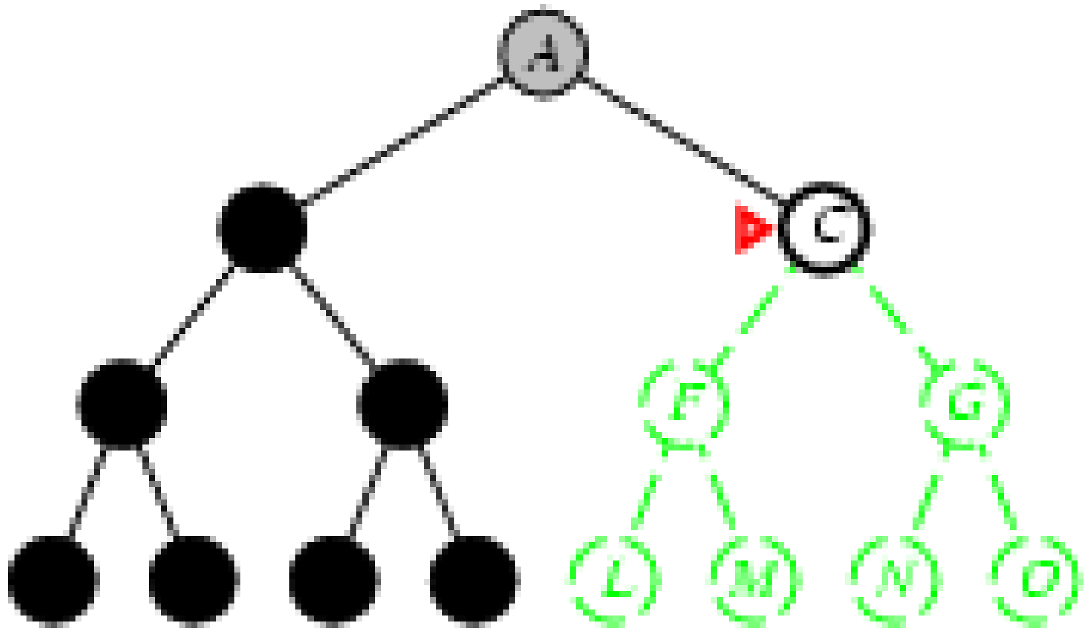
Depth-first search

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



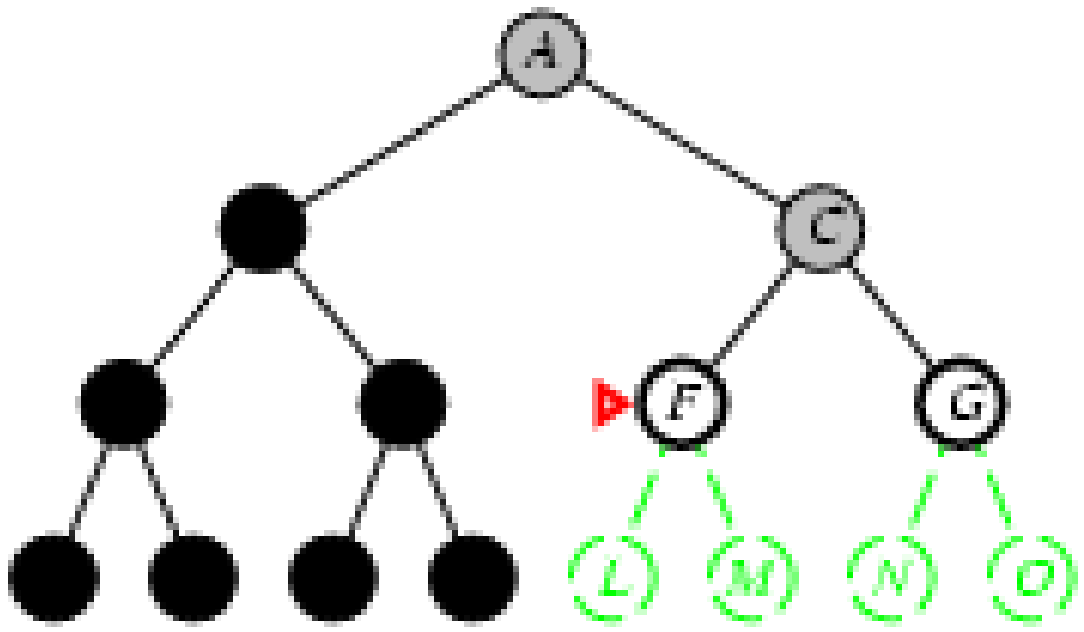
Depth-first search

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



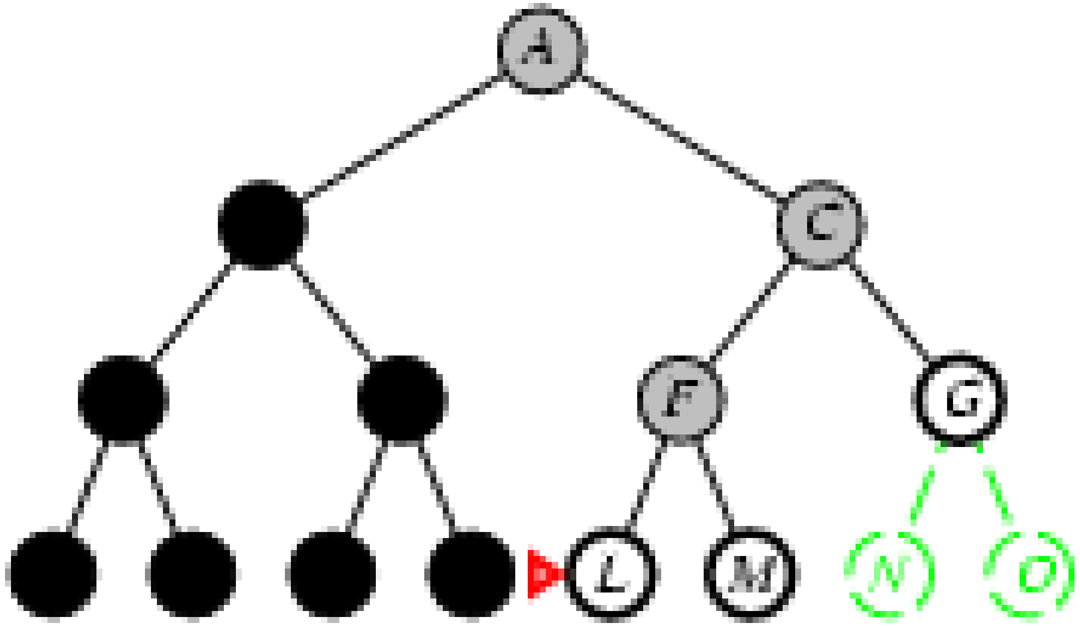
Depth-first search

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



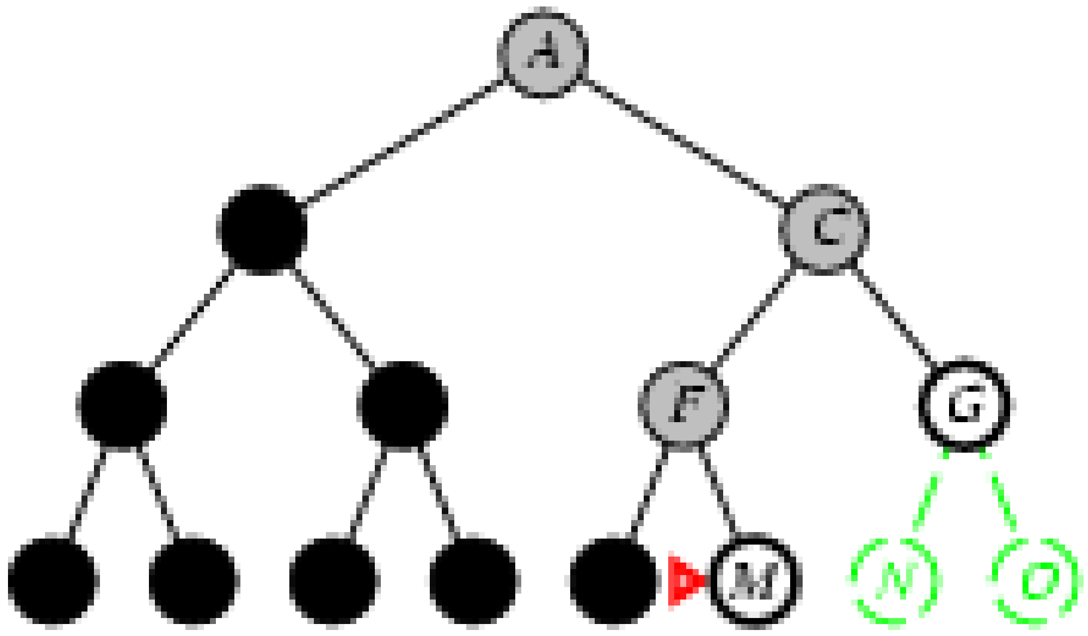
Depth-first search

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

Iterative deepening search

Function Iterative_Deepening_Search(*problem*) return *solution* or *failure*

Inputs: *problem*, a problem

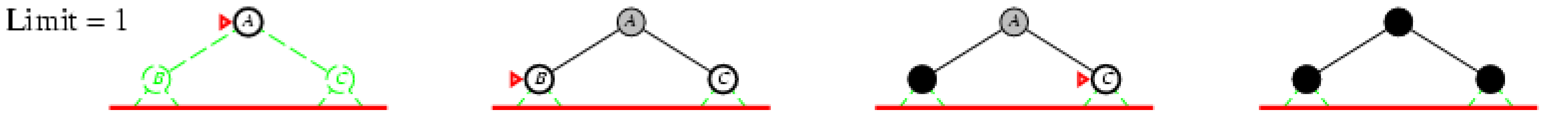
For *depth* ← 0 to ∞ do

    result ← Depth_Limited_Search (*problem, depth*)

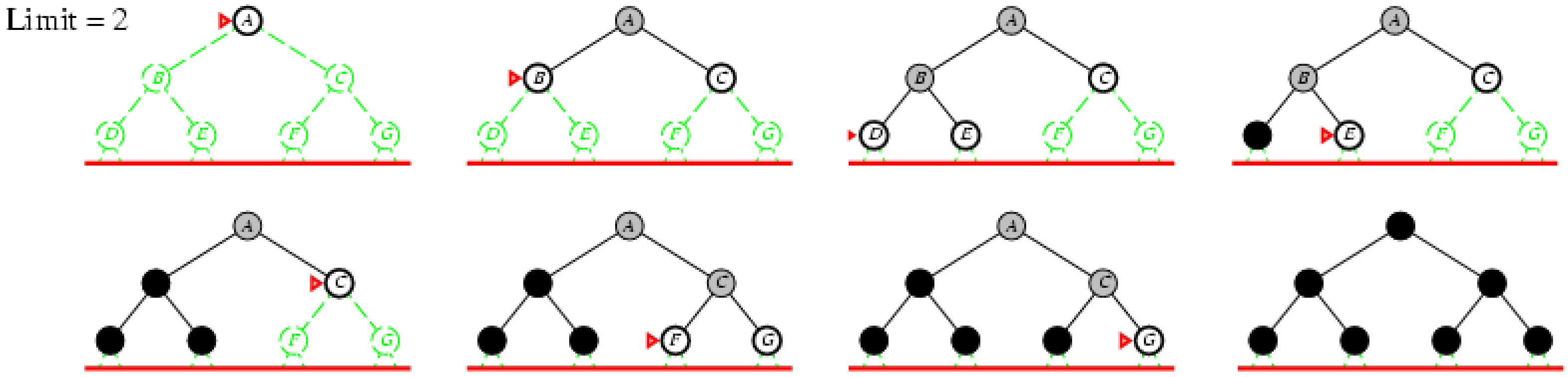
    if *result* ≠ cutoff then return *result*

Iterative deepening search l =0

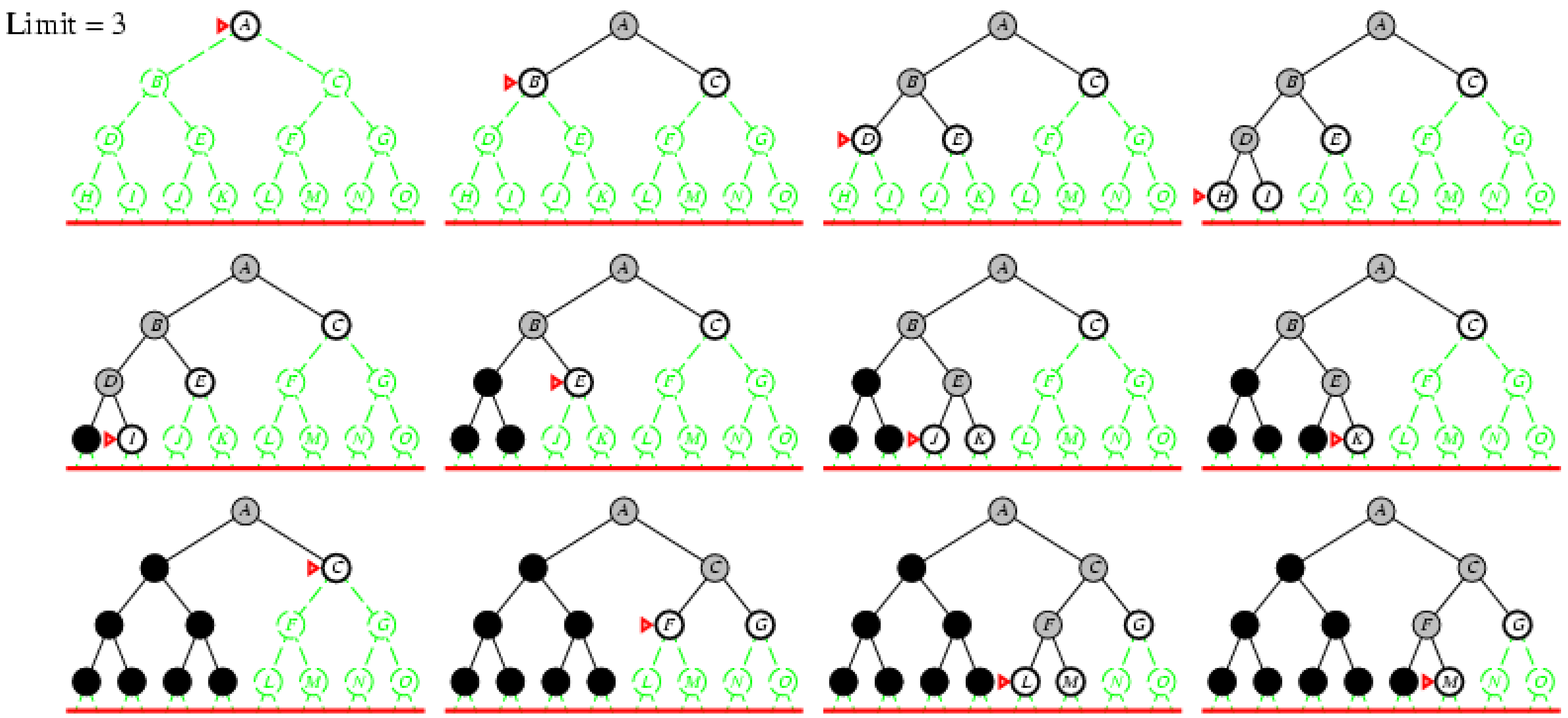
Iterative deepening search $l = 1$



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

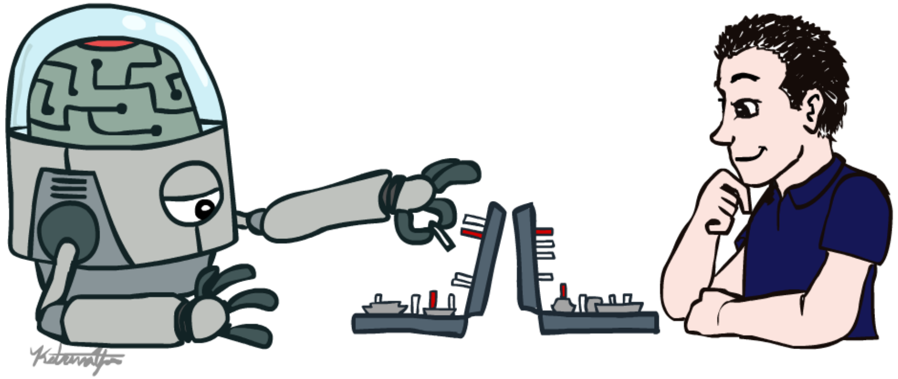
- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Summary of blind search algorithms

Criterion	Breadth First	Depth First	Depth Limited	Iterative Deepening
Complete?	Yes	No	No	Yes
Time	O(b^{d+1})	O(b^m)	O(b^l)	O(b^d)
Space	O(b^{d+1})	O(bm)	O(bl)	O(bd)
Optimal	Yes	No	No	Yes

Next class?

- Informed search
- **Heuristic search**



Thanks!