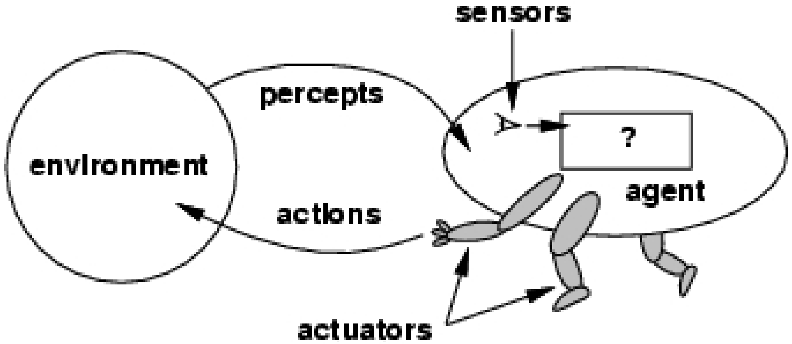
Review

- **Agents**
- **agent function vs agent program**
- A **perfectly rational agent**
- **PEAS**
- **Environments**
  - are categorized along several dimensions:

  observable? deterministic? episodic? static? discrete? single-agent?

  - **Real world vs Virtual world**
- Several basic agent architectures exist:

  reex, reex with state, goal-based, utility-based



- **Questions:**

  1.*Why design of a rational intelligent agent or agents is usually difficult? Where the difficulties come from?*

  2.Why design of a human-like intelligent agent or agents is more challenging？

Lecture 4: Problem-solving in AI

LI Xiaoan (Dustin) 李孝安, Dr. & Associate Prof.
MARS-lab
School of Computer Science and Engineering
Northwestern Polytechnical University

E-mail: dustinli@nwpu.edu.cn
WeChat group: AI-2021-Li
Cell phone: 18629662731

Outline

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

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
```

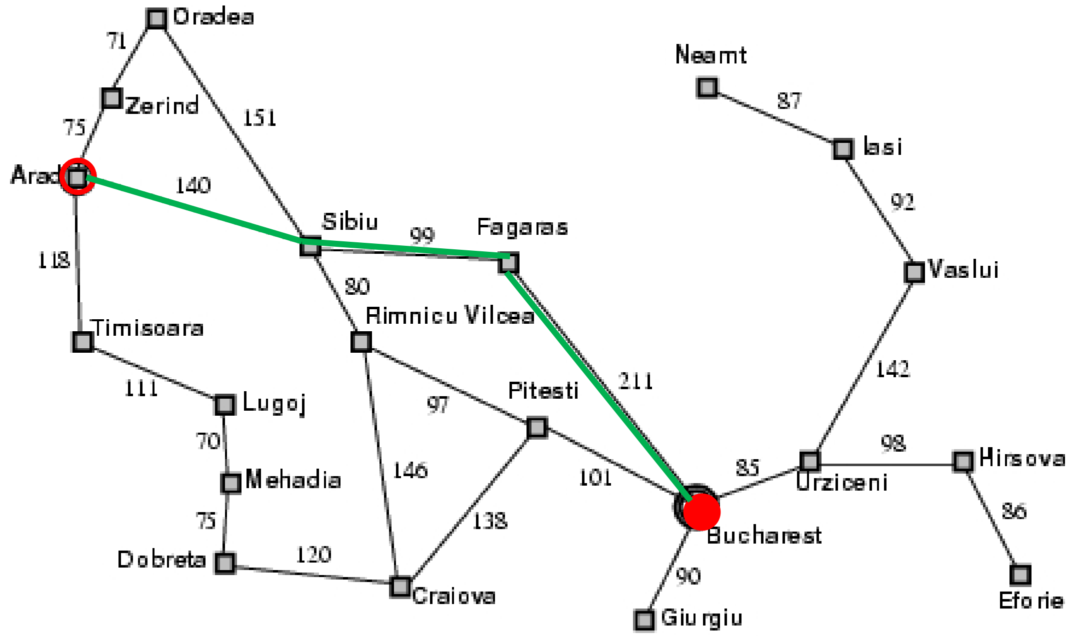
Goal formulation

Problem formulation

Problem-solving process based on algorithms

Example 1: Road planning in Romania

- Agent: touring in Romania on holiday
  - currently in Arad.
  - Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution: *Search(problem)?*
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest



Challenge questions:

Q1: If the map is not labelled, what will happen for this problem-solving?

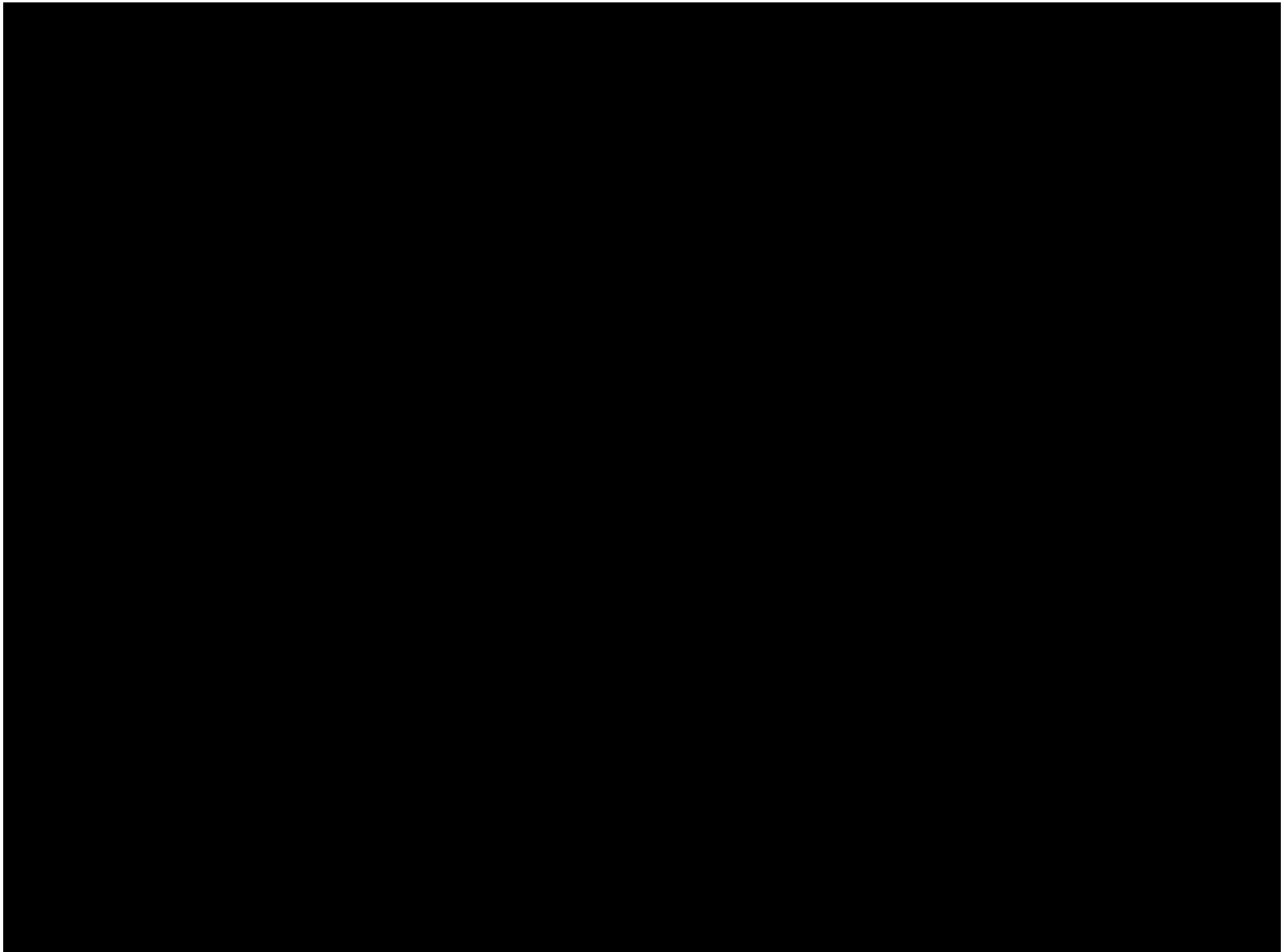
Q2: How to label the map?

Q3: Is the path found the best solution in all practical cases?

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, execution
- Unknown state space → **exploration** problem e.g ADRobotAir
- Known/labelled state space → **exploitation** problem

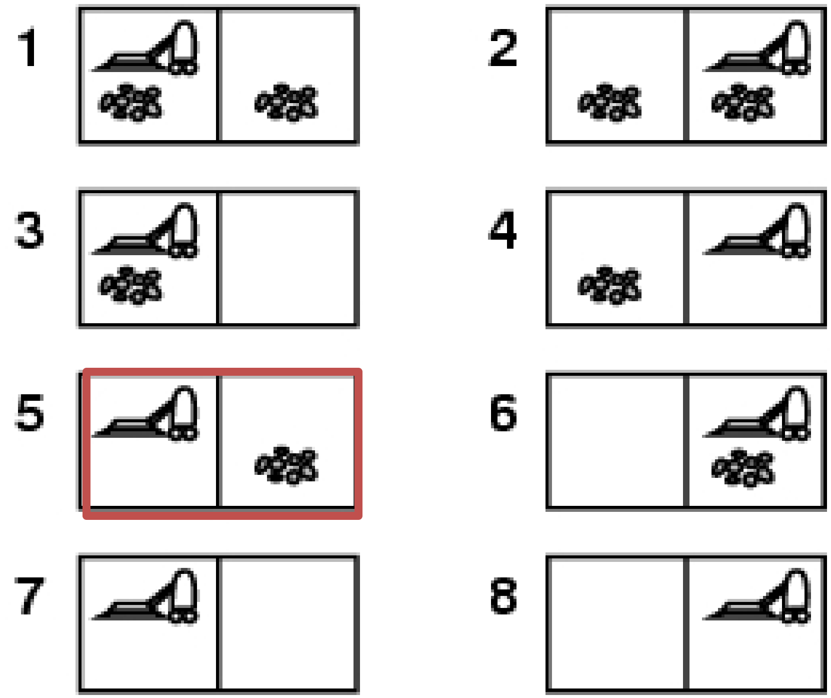
Example 1: ADRobot



Example 2: vacuum world

- P1: Single-state, start in #5. Solution?

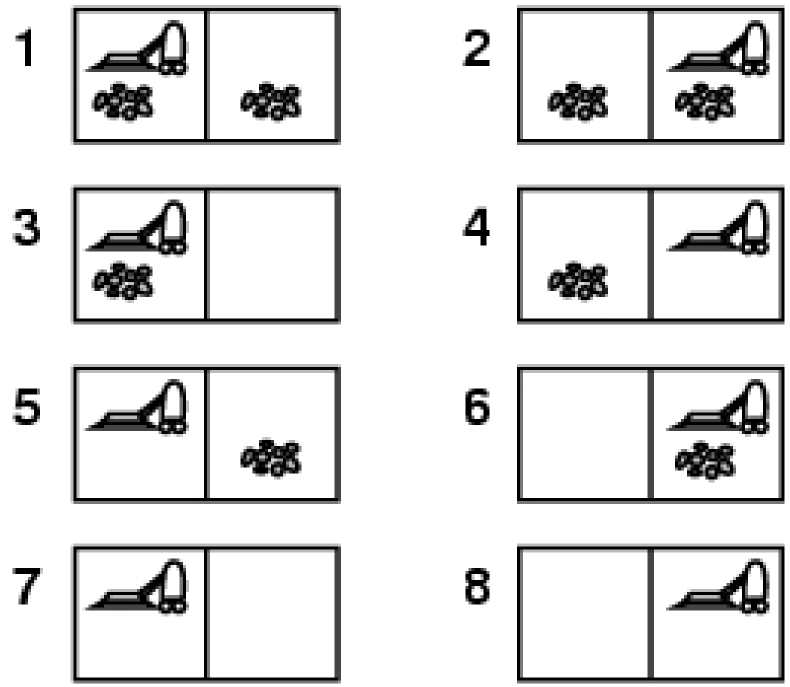
  [Right, Suck]



Example: vacuum world

- P2: 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]

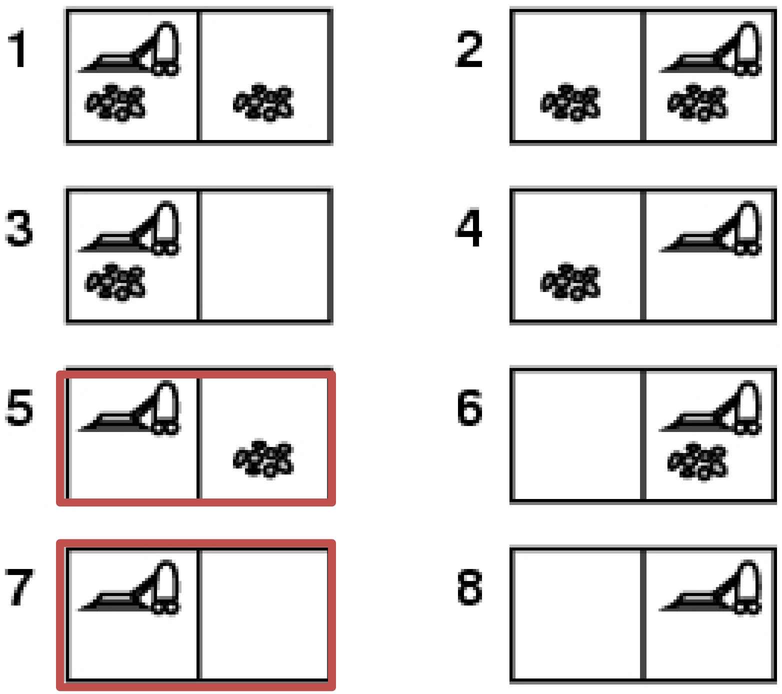


Example: vacuum world

- P3: 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]



Single-state problem formulation

A problem is defined by four items:

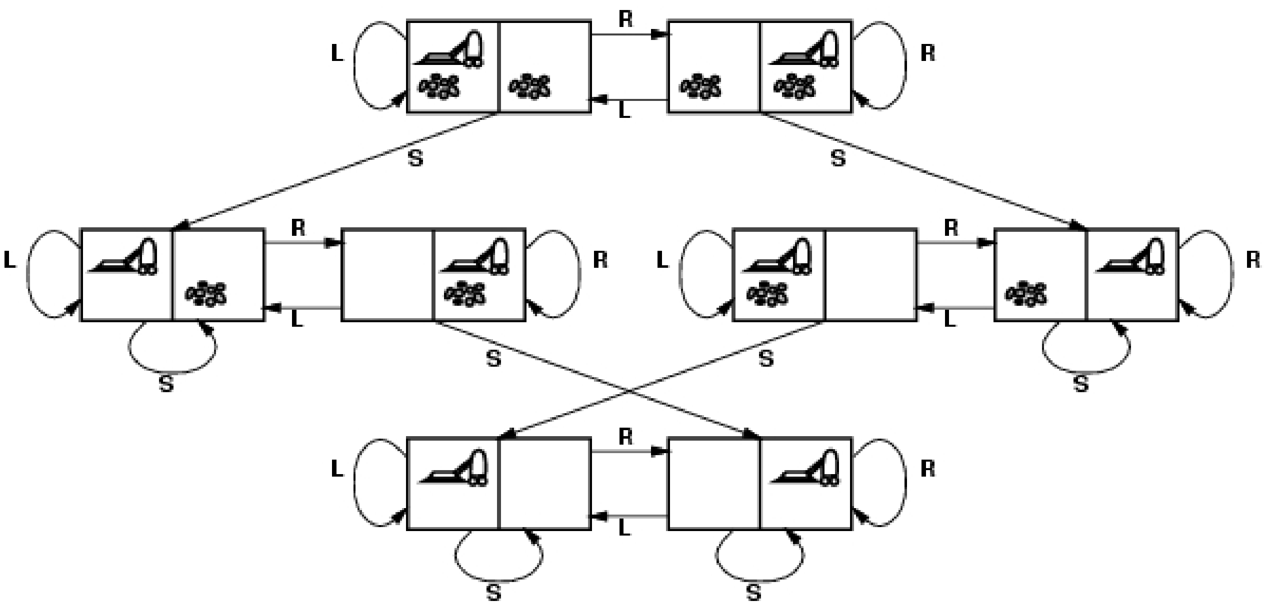
1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   – e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   – explicit, e.g., *x* = "at Bucharest"
   – implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   – e.g., sum of distances,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

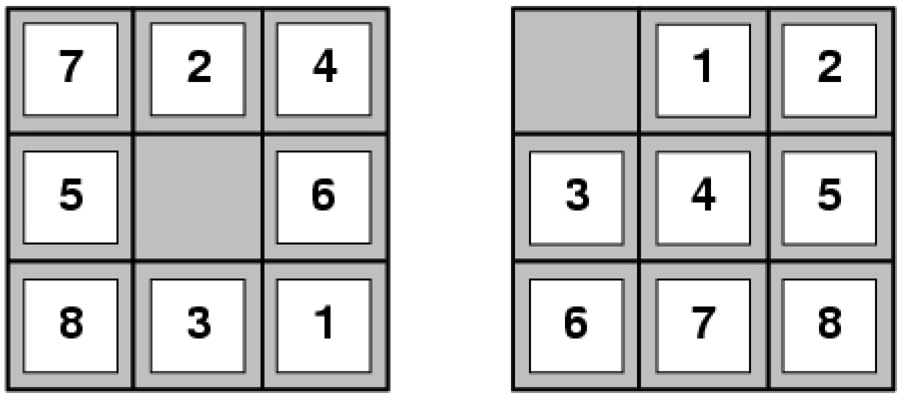
- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability,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

State space for the Vacuum world



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

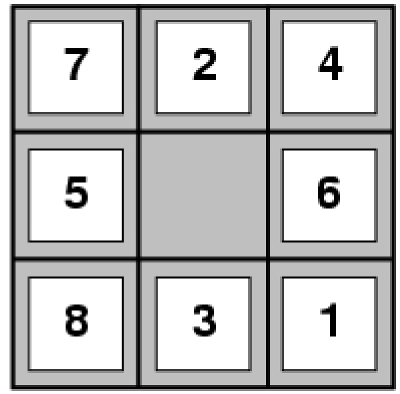
Example 3: The 8-puzzle



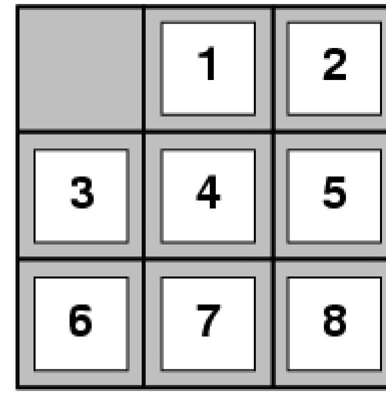
Start State Goal State

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

Example 3: The 8-puzzle



Start State

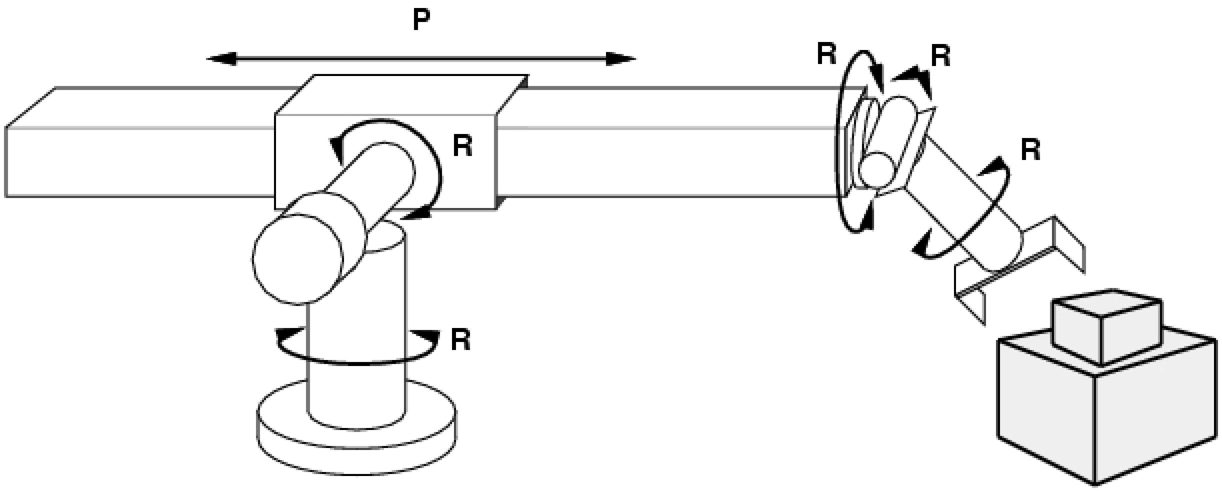


Goal State

- 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 4: 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

Search algorithms

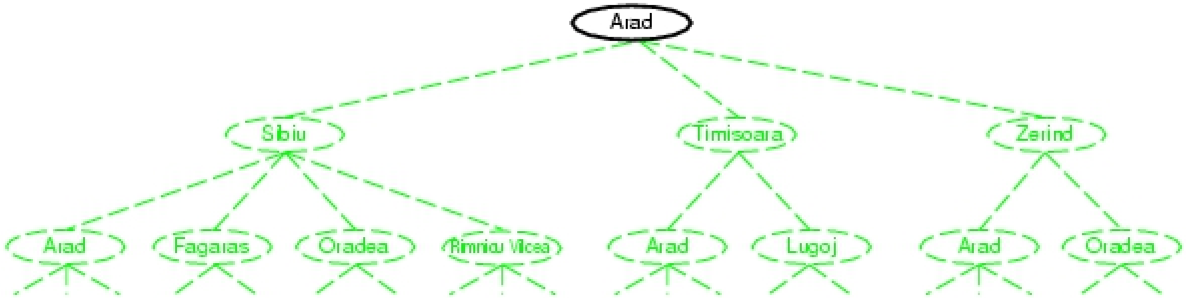
- Strategies
- Algorithms

Tree search algorithms

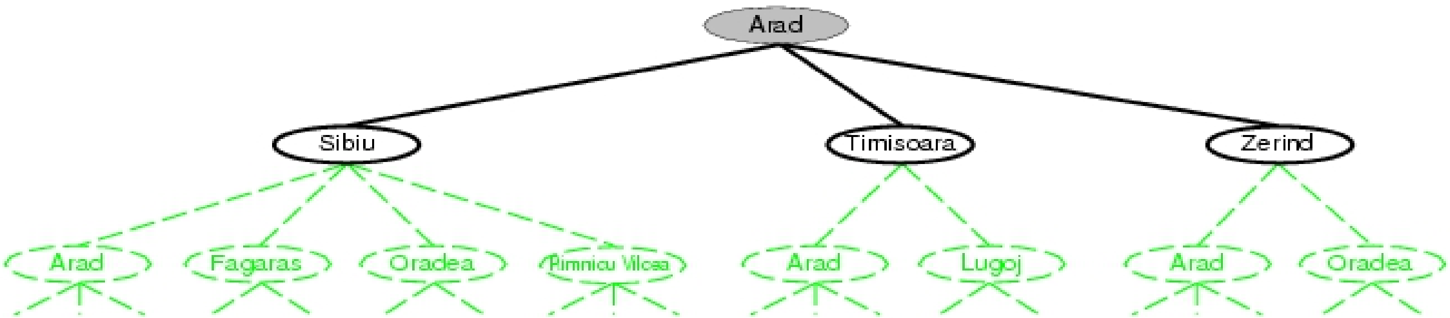
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
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
```

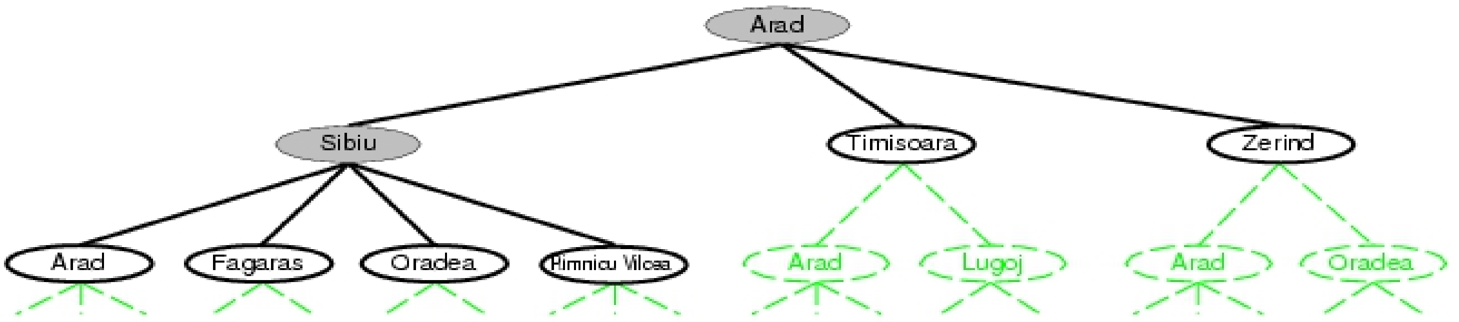
Tree search example



Tree search example



Tree search example



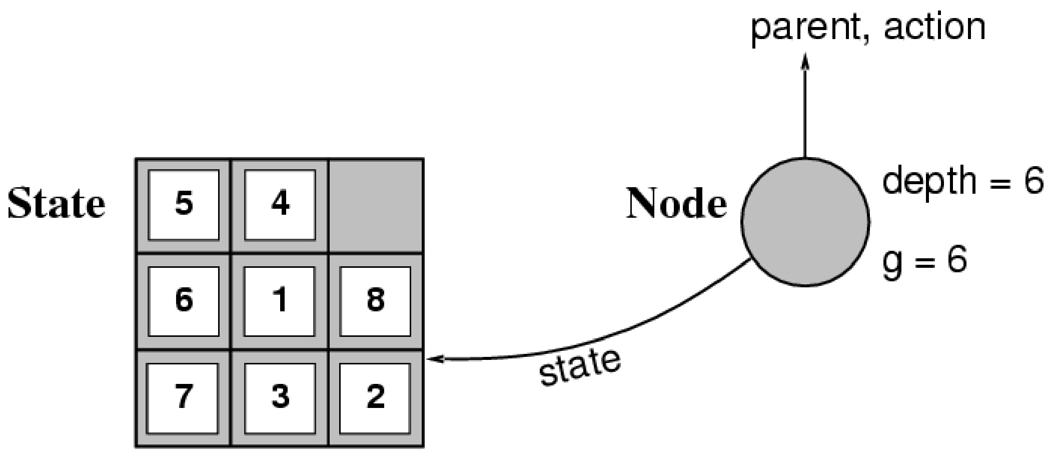
Implementation: general tree search

```
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

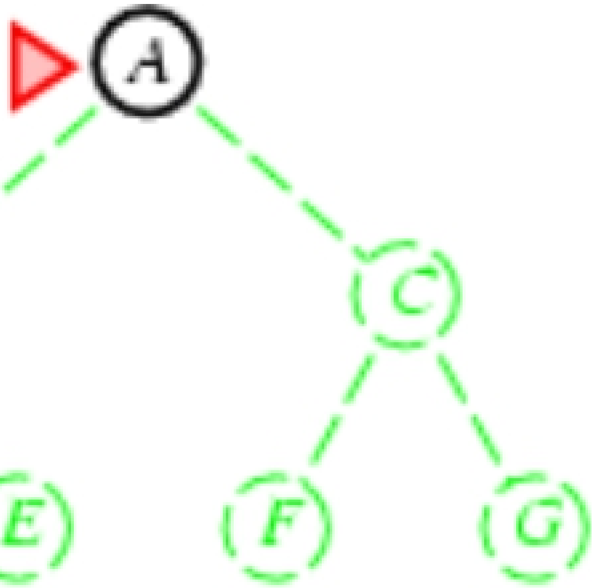
- Uninformed search strategies use only the information available in the problem definition

Strategies:

- Breadth-first search
  Uniform-cost search
  Depth-first search
  Depth-limited search
  Iterative deepening search

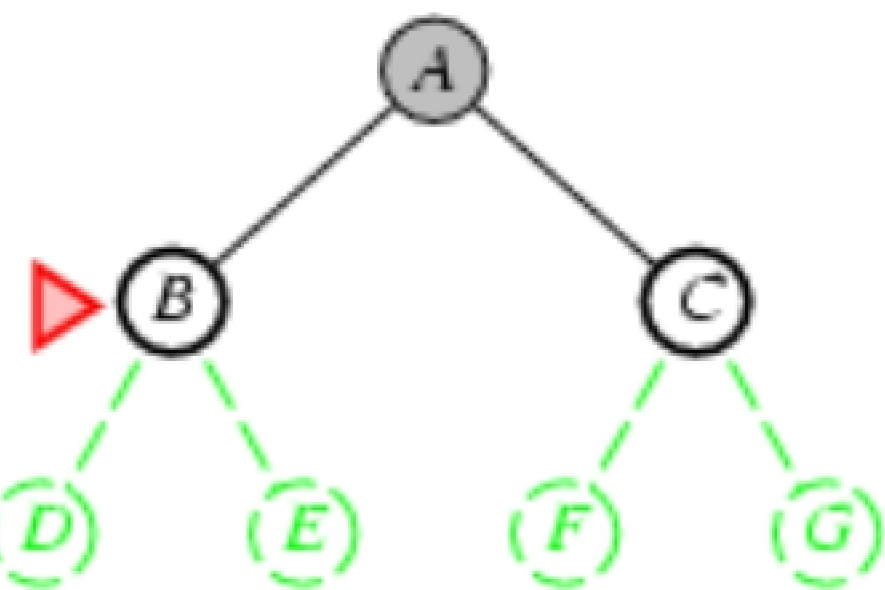
Breadth-first search

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



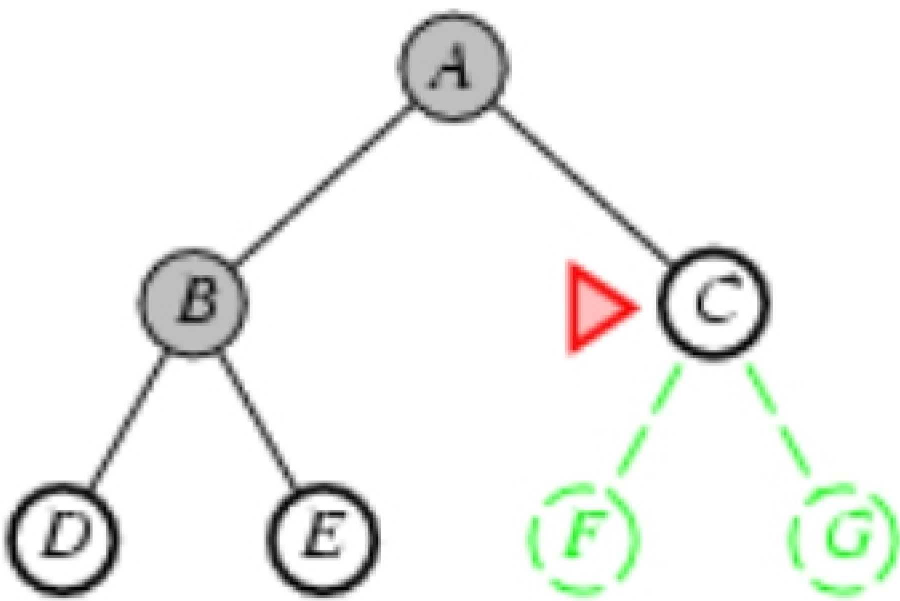
Breadth-first search

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



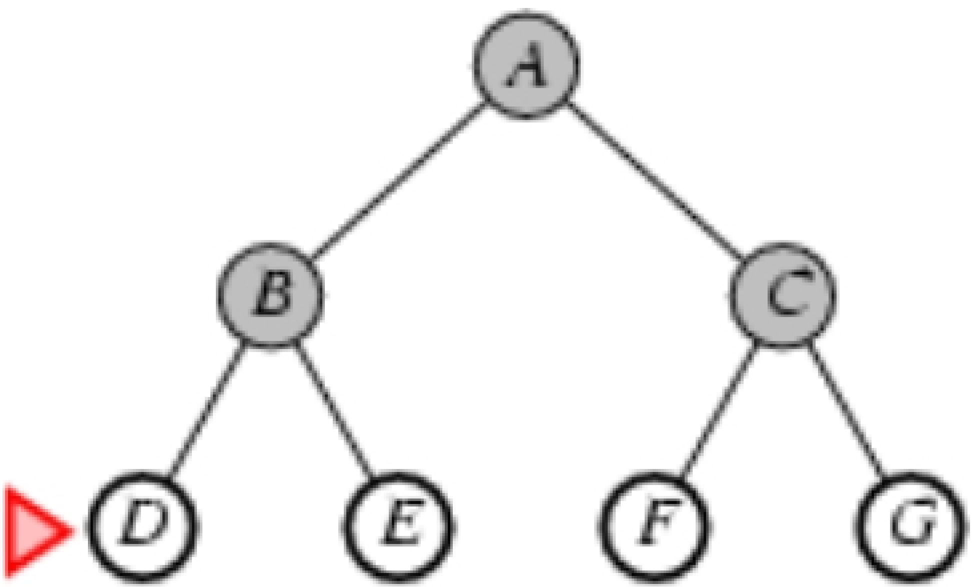
Breadth-first search

- 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)

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 ∞)

Uniform-cost search

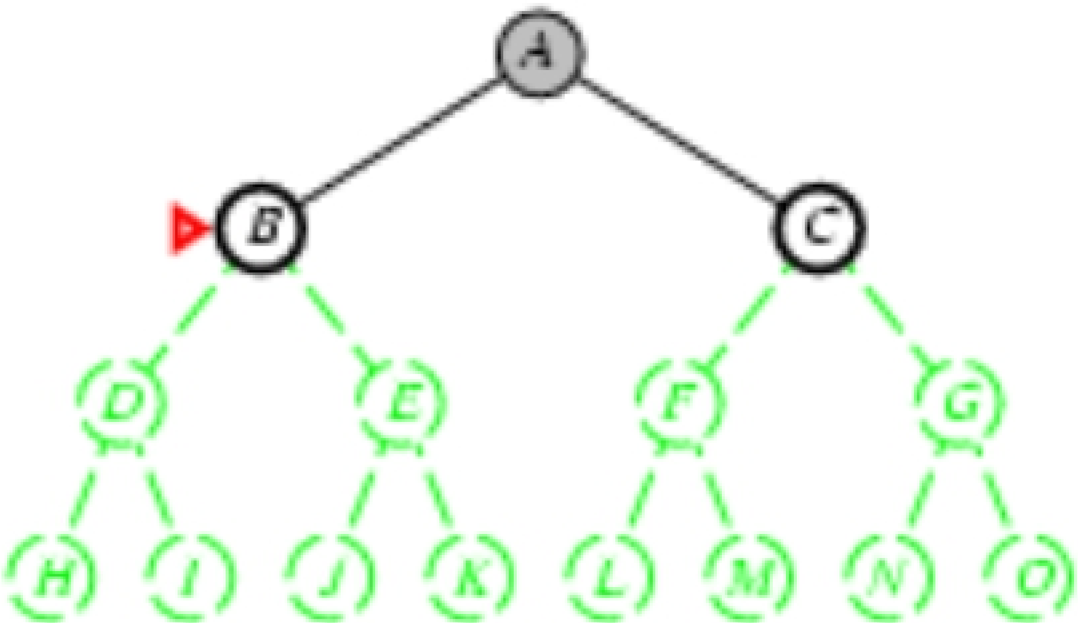
- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with g ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the 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, i.e., put successors at front

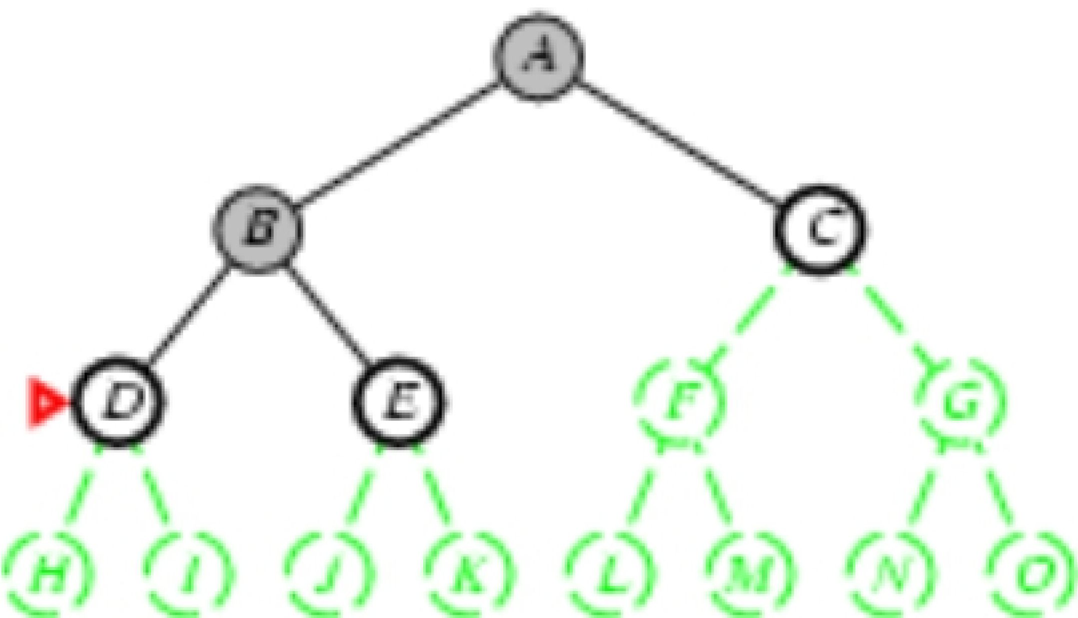
Depth-first search

- 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

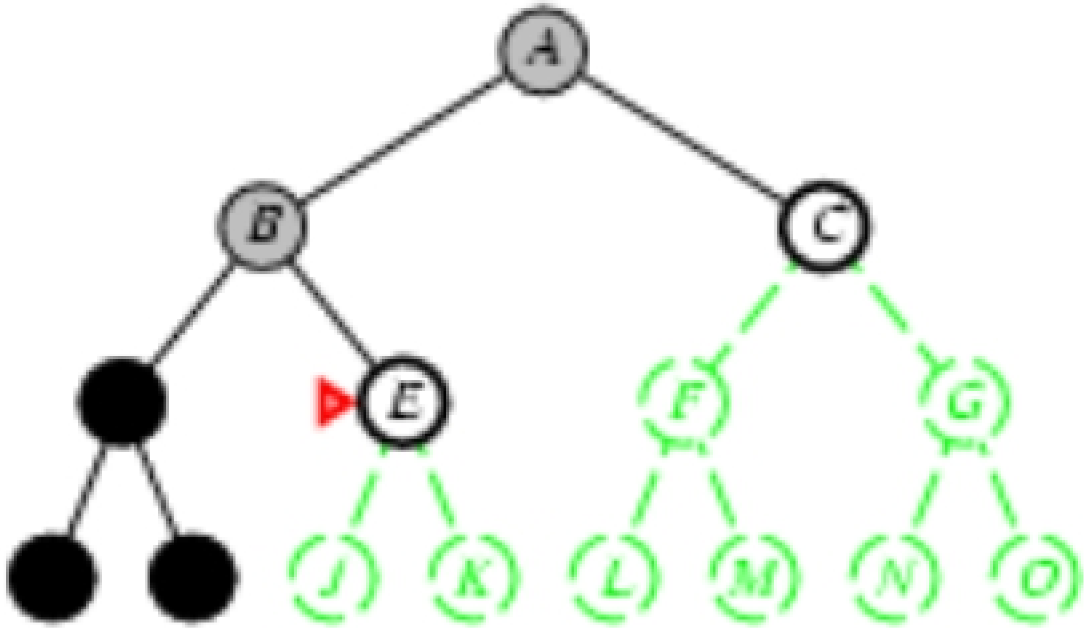
- Expand deepest unexpanded node
  Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

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

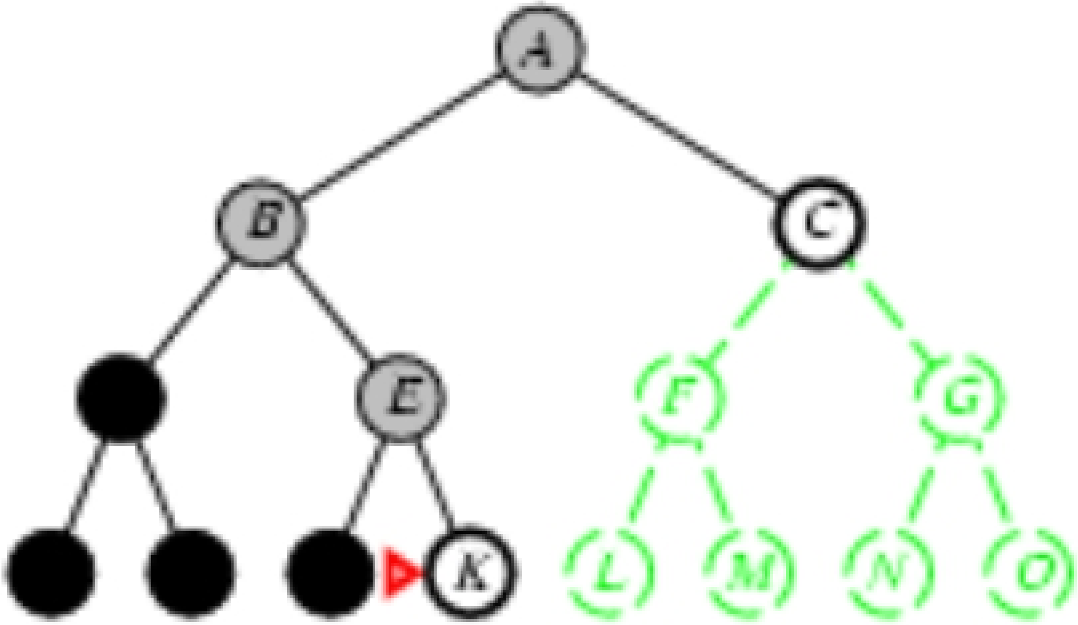


Depth-first search

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

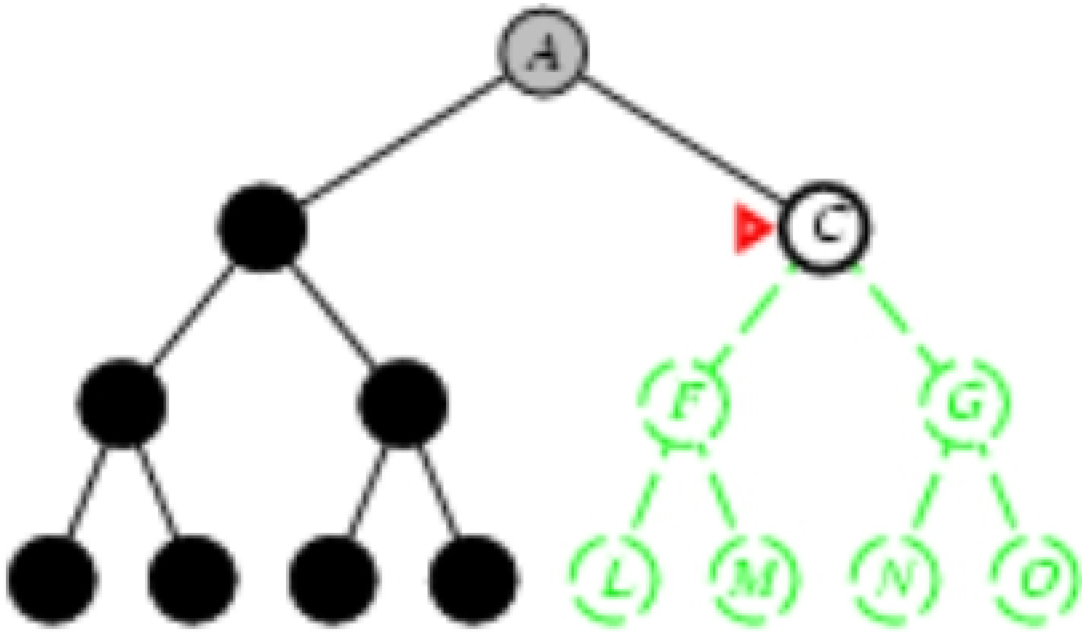
Depth-first search

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



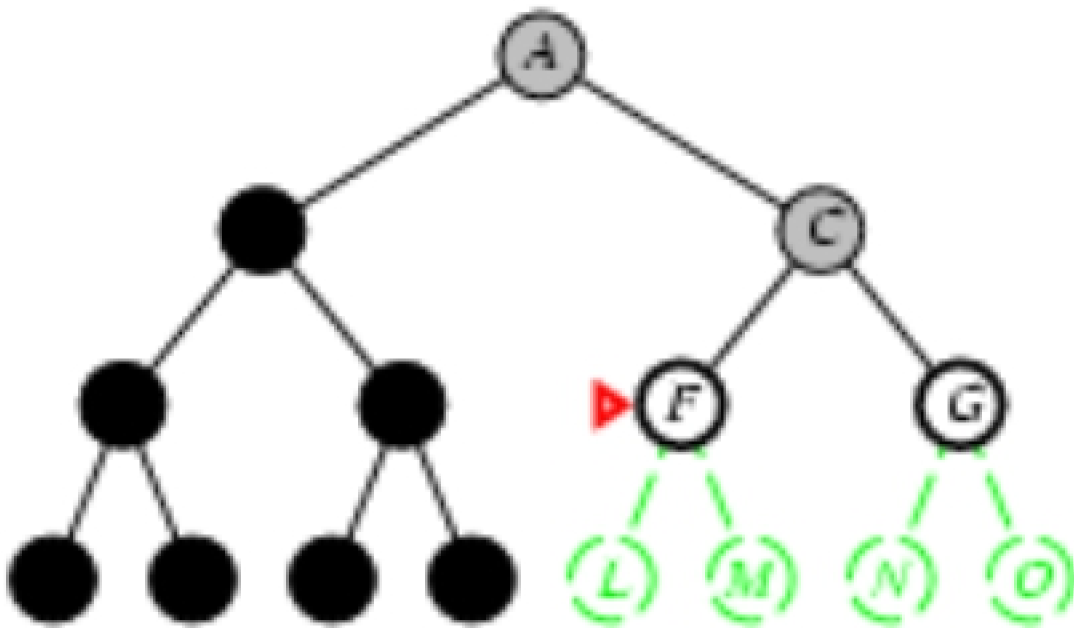
Depth-first search

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



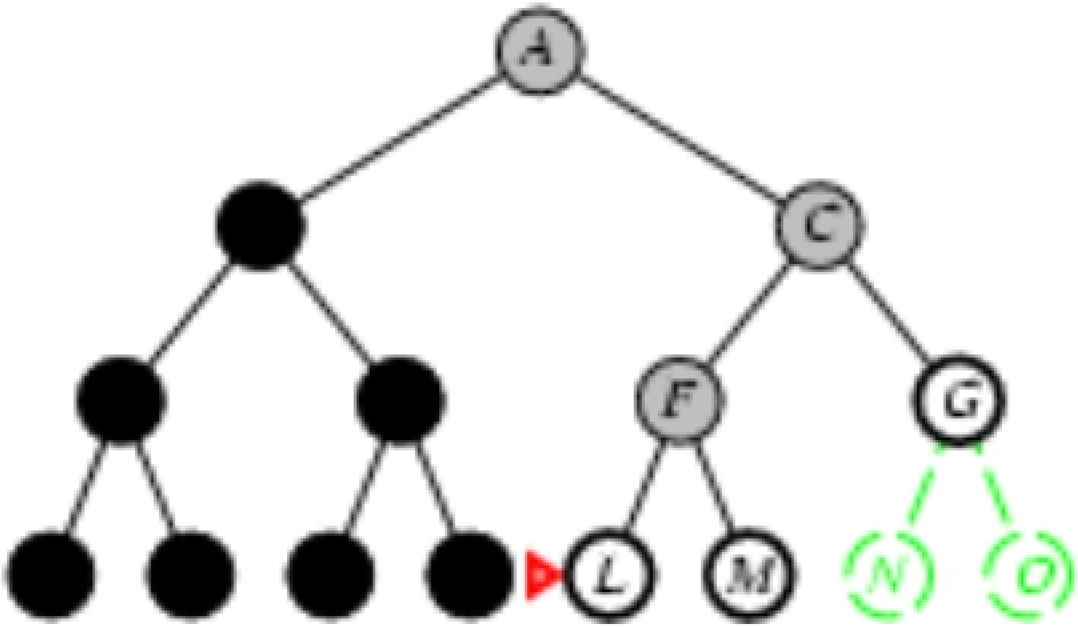
Depth-first search

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



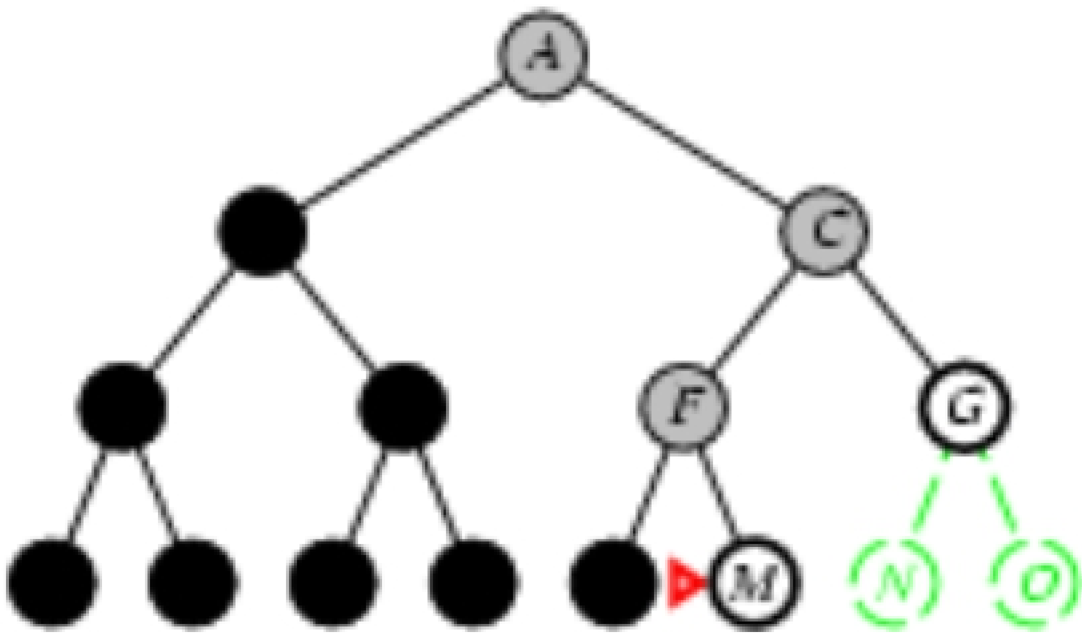
Depth-first search

- 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

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

    → complete in finite spaces

- <u>Time?</u> $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- <u>Space?</u> $O(bm)$, i.e., linear space!
- <u>Optimal?</u> No

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

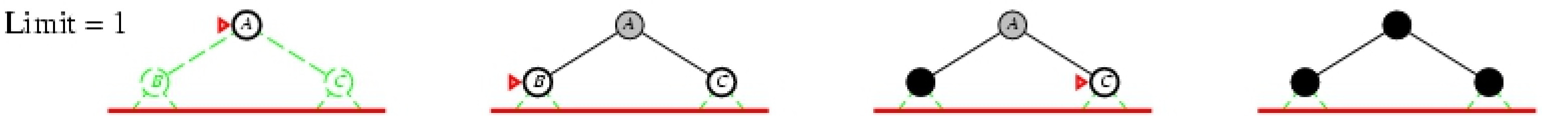
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

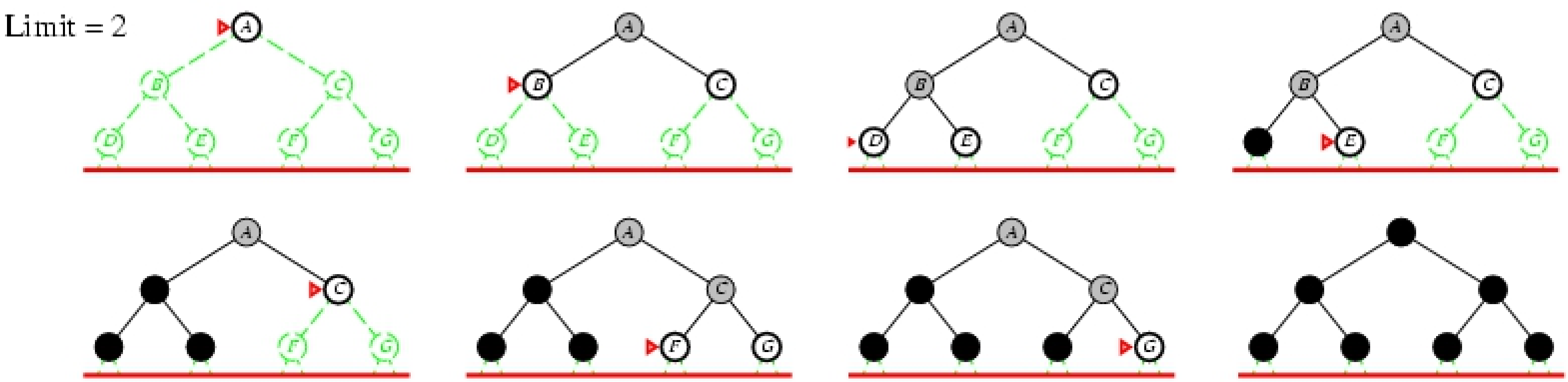
Iterative deepening search l =0

Limit = 0

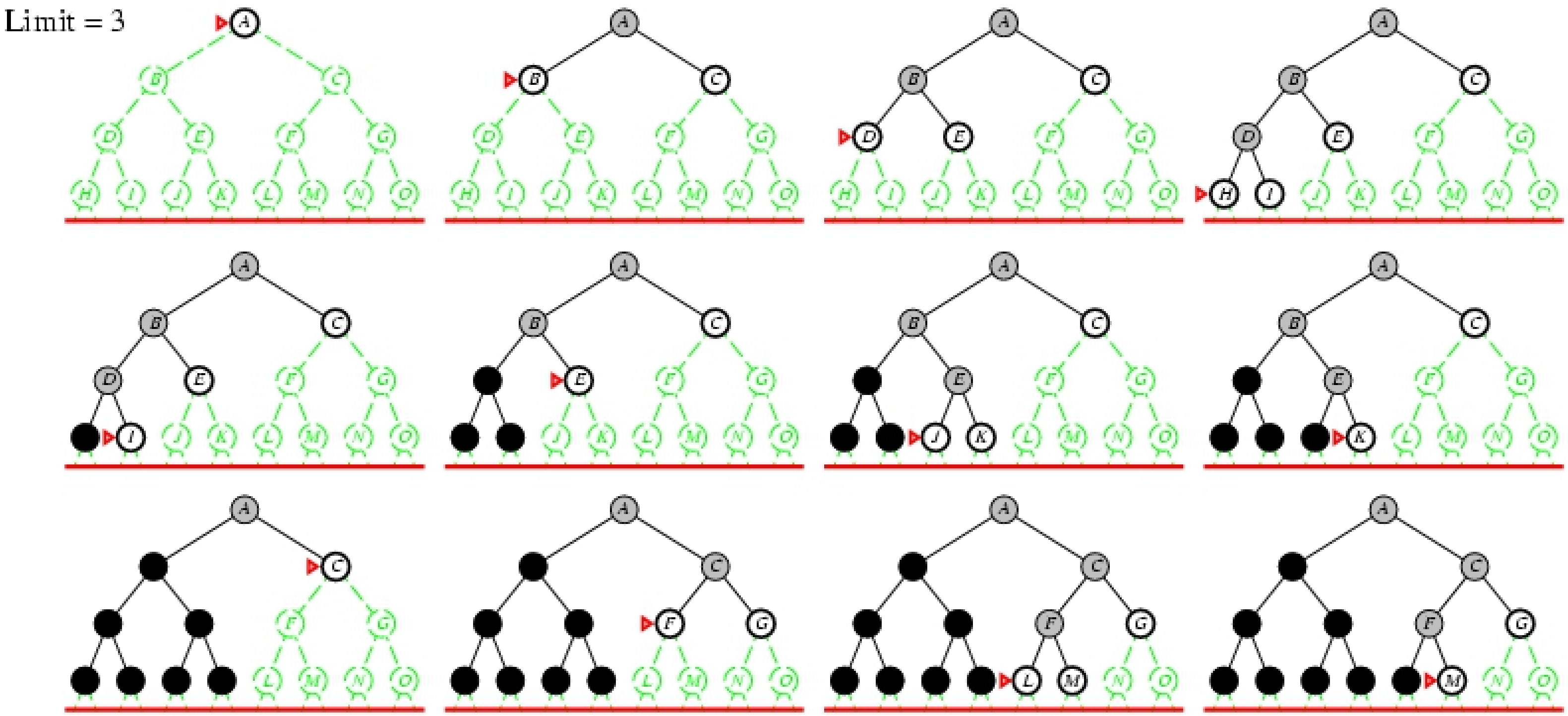
Iterative deepening search l =1



Iterative deepening search *l* =2



Iterative deepening search *l* =3



Iterative deepening search

- Number of nodes generated in 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*:

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

- 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

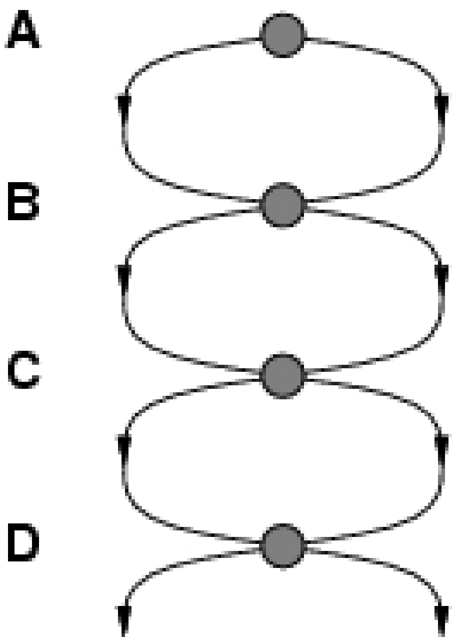
- <u>Complete?</u> Yes
- <u>Time?</u> $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- <u>Space?</u> $O(bd)$
- <u>Optimal?</u> Yes, if step cost = 1

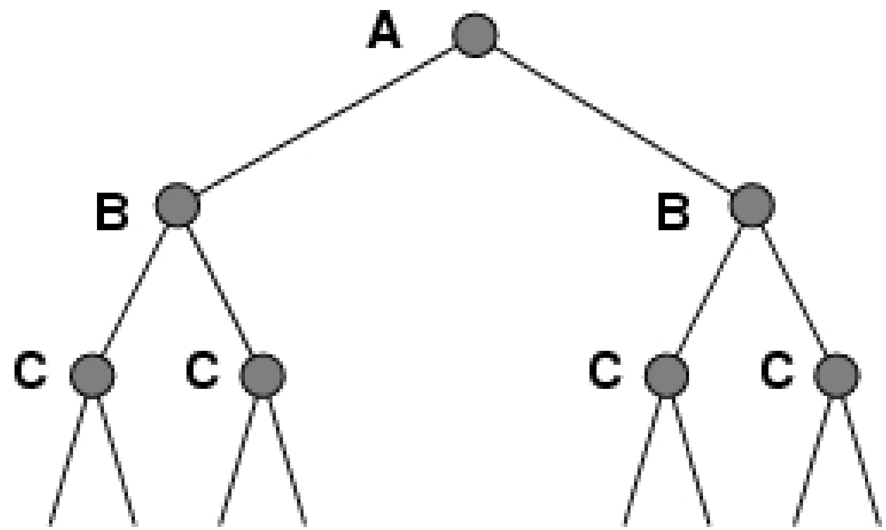
Summary of algorithms*

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

- **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
- **Challenging problem** for **uninformed search** strategies
  - Complexity in space and in time

Assignment

– Chap 3: exercise 3.1, 3.6

*Handed in next Tuesday