CS2109S: Introduction to AI and Machine Learning

Lecture 2: Solving Problems by Searching

19 Aug 2022

Recap

- Introduction
- PEAS
- Environment
- Agent Models

Environment types

	Chess with a clock	Chess without a clock	Autonomous driving
Fully observable	Yes	Yes	No
Deterministic	Strategic	Strategic	No
Episodic	Yes	Yes	No
Static	Semi	Yes	No
Discrete	Yes	Yes	No
Single agent	No	No	No

The environment type largely determines the agent design
The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

Episodic (vs. sequential): The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.

Consider a task in which an agent plays rock-paper-scissors against a human multiple times.

Episodic or sequential?

“Memorylessness”

Depends on the human?

Agenda

- 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
```

Learning to frame the problem

Example: Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:

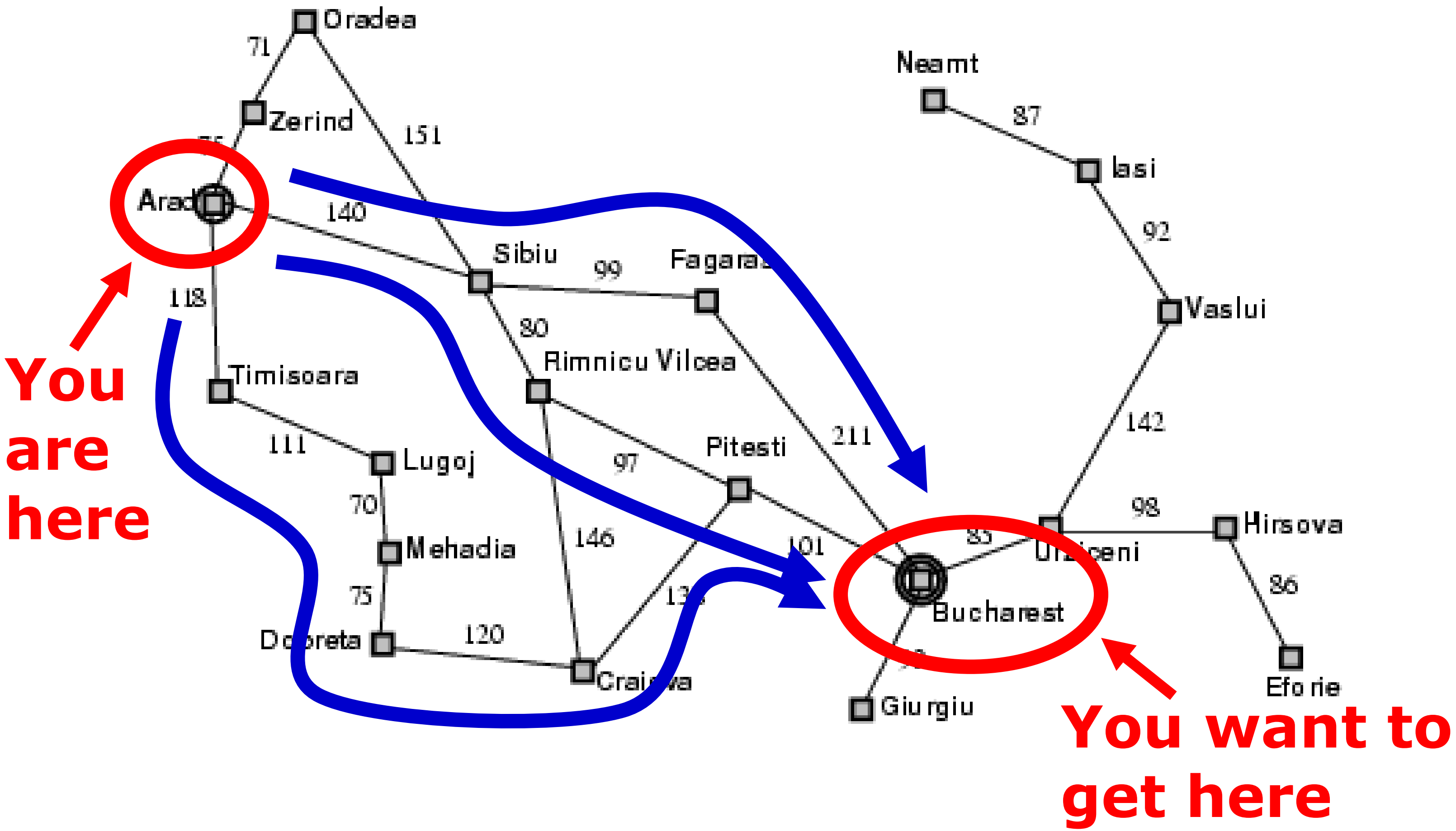
- be in Bucharest

Formulate problem:

- states: various cities
- actions: drive between cities

Find solution:

- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest



Oradea
71
Zerind
151
140
118
Sibiu
99
80
Rimnicu Vilcea
Timisoara
111
Lugoj
70
Mehadia
75
120
146
97
Pitesti
211
101
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
86
Eforie
Bucharest
Giurgiu
You are here
You want to get here

Different 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

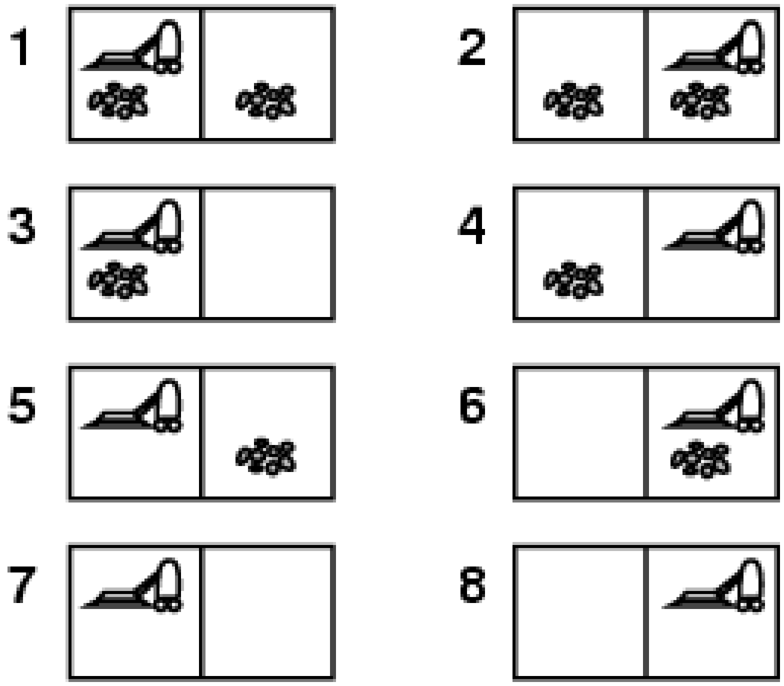
Example: vacuum world

- 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?

  [Right,Suck,Left,Suck]

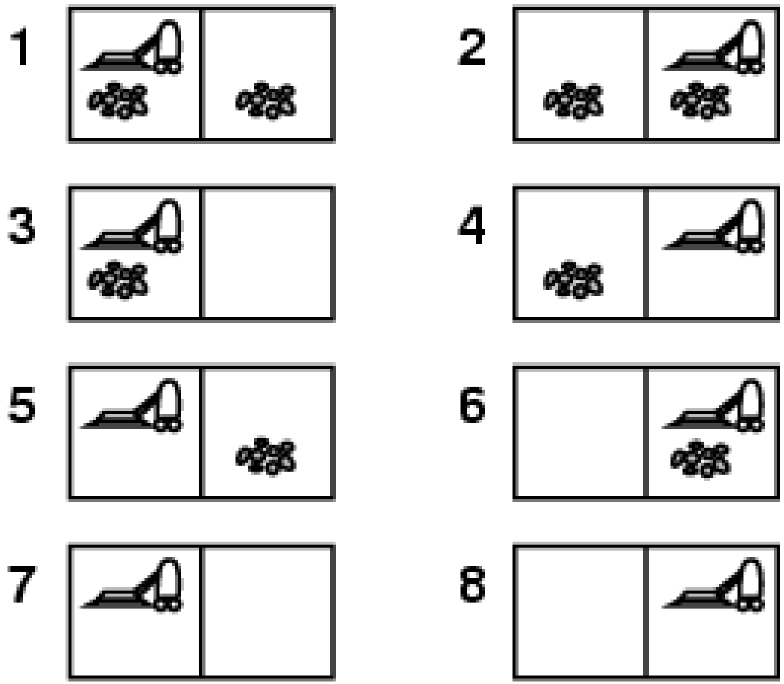


Example: vacuum world

- 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]*



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

Graph

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"

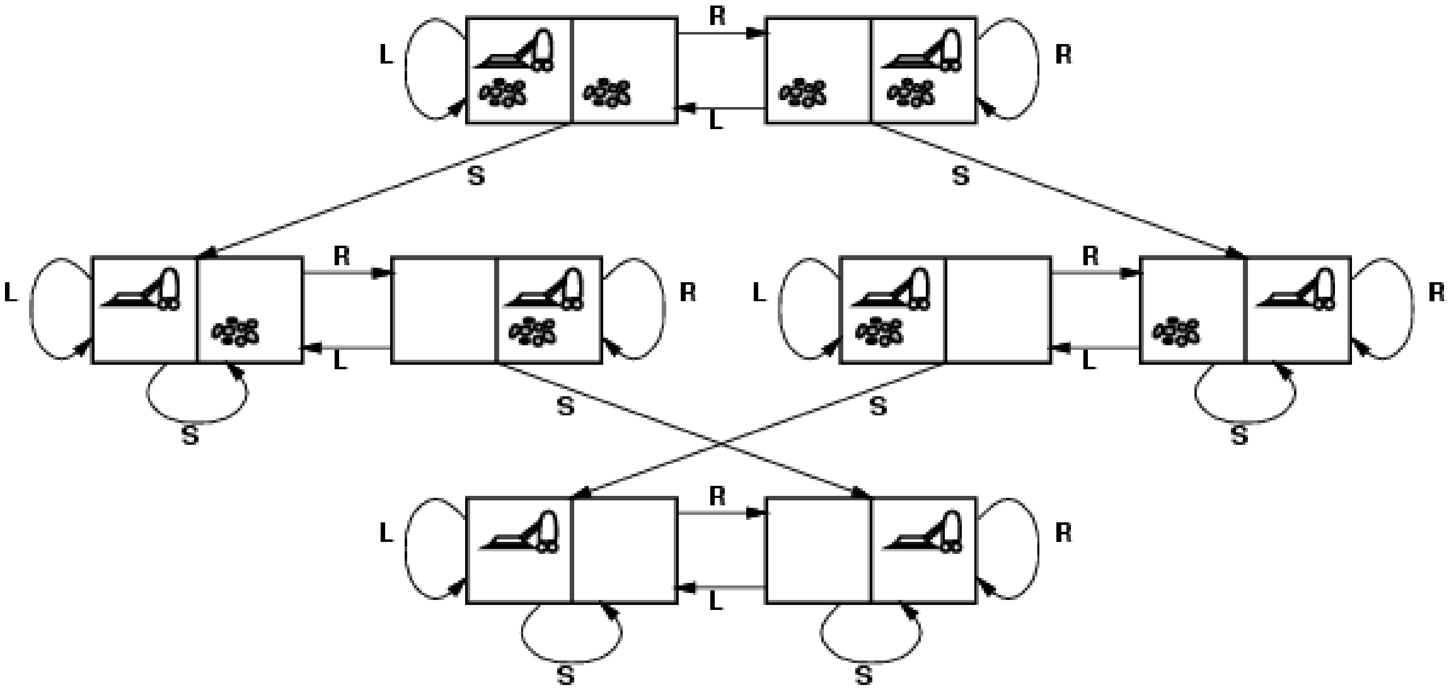
(Abstract) solution =
set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem

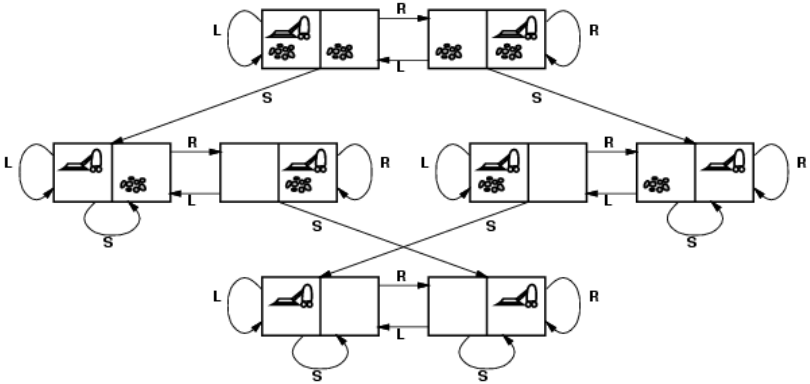
Computer Science is the study of abstractions

Managing Complexity

Vacuum world state space graph



Vacuum world state space graph



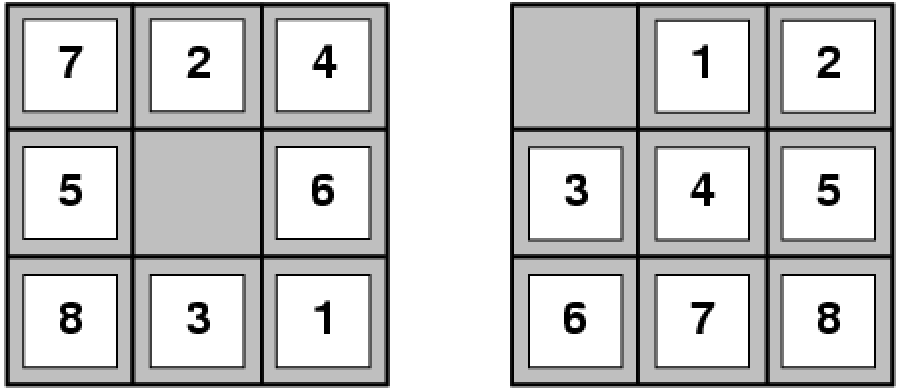
states? integer: dirt and robot location

actions? *Left*, *Right*, *Suck*

goal test? no dirt at all locations

path cost? 1 per action

Example: The 8-puzzle



Start State Goal State

states? locations of tiles

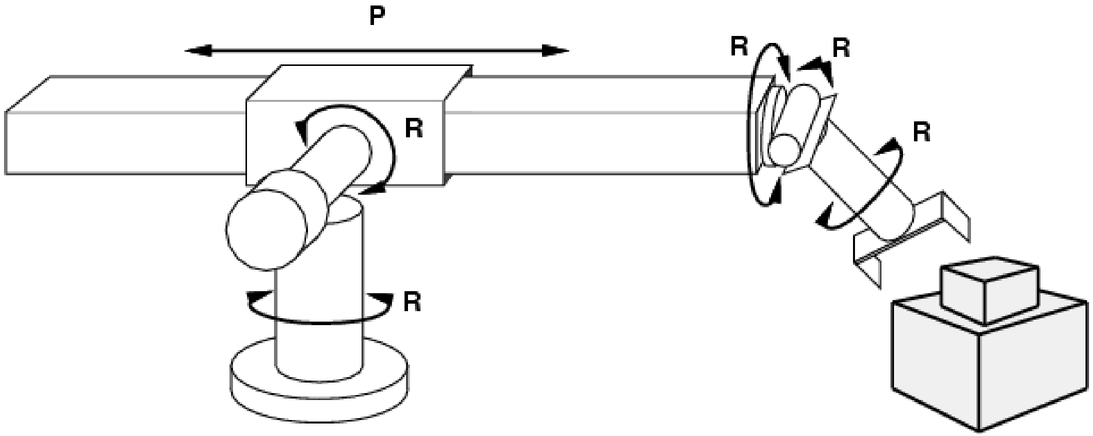
actions? move blank left, right, up, down

goal test? = goal state (given)

path cost? 1 per move

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

Example: 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

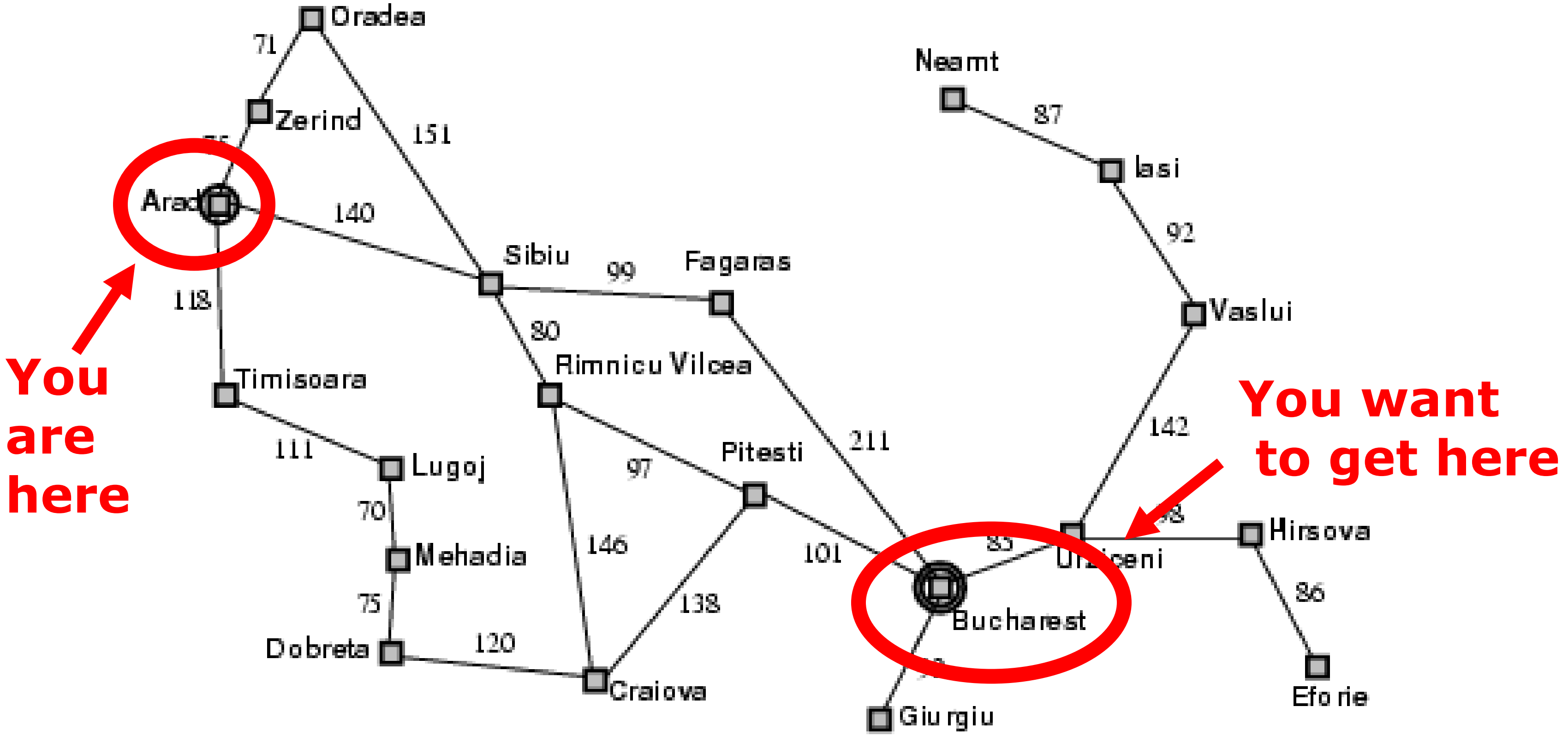
Tree search algorithms

Basic idea:

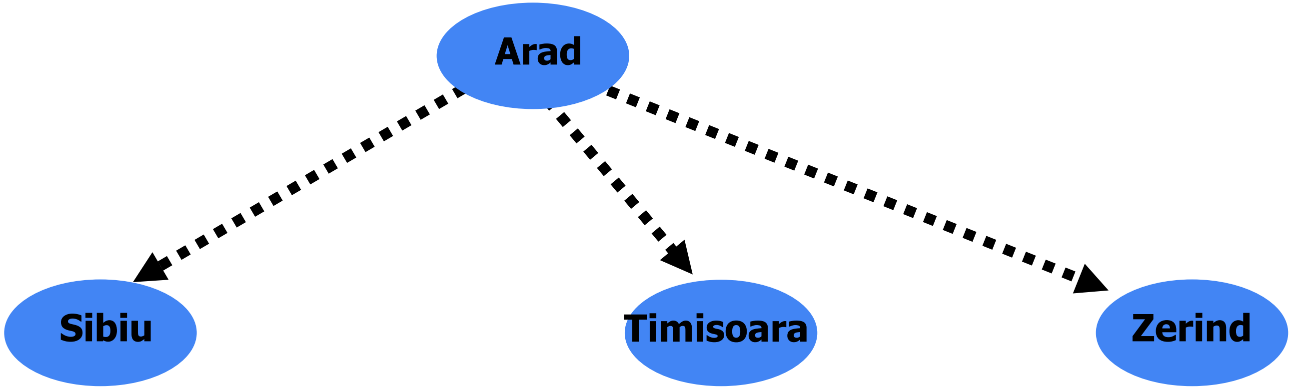
- offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
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
```

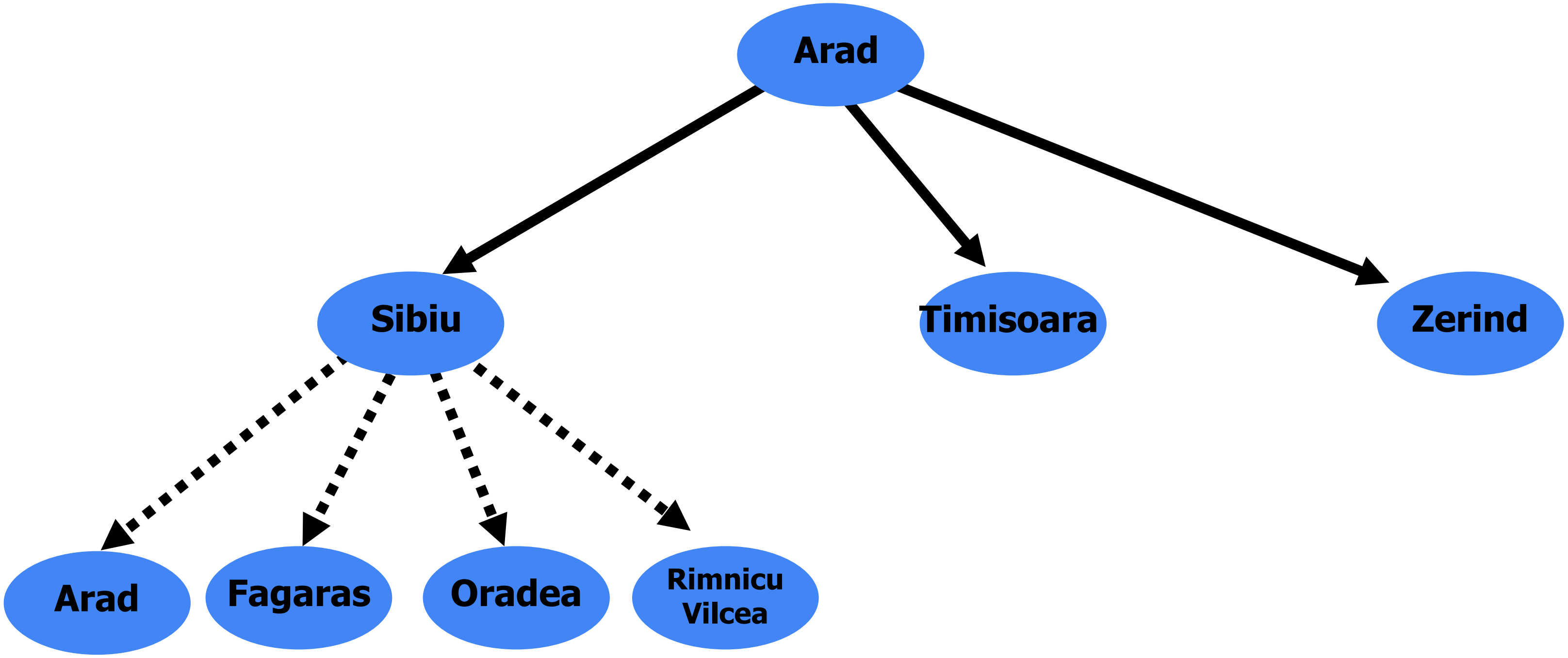
Example: Romania



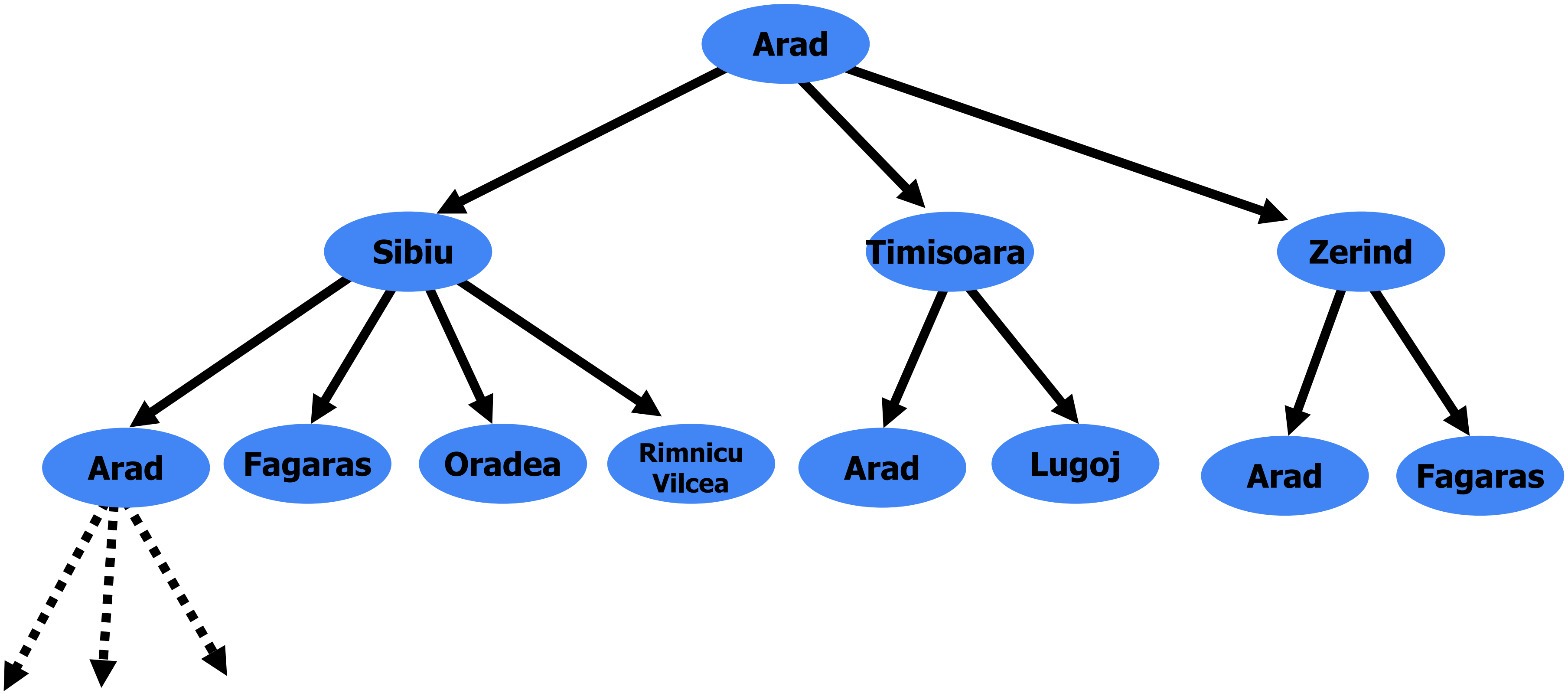
Tree search example



Tree search example



Tree search example



Implementation: general tree search

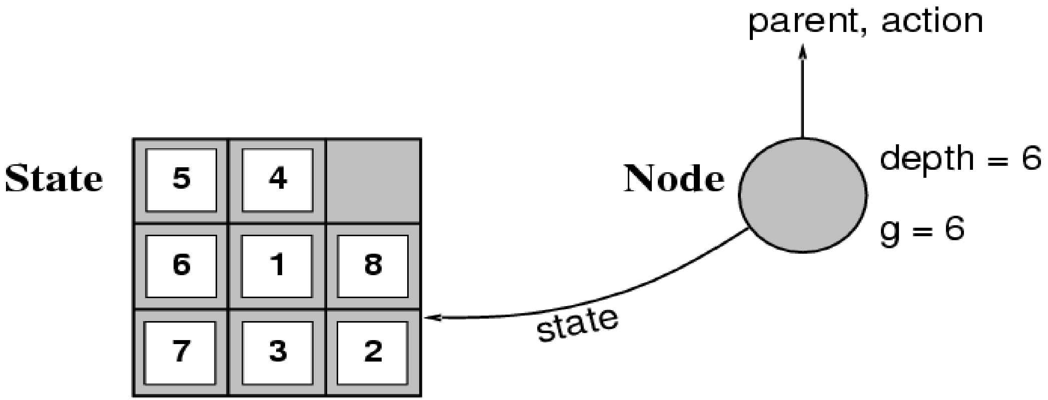
frontier

```
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

1. Breadth-first search (BFS)
2. Uniform-cost search
3. Depth-first search (DFS)
4. Depth-limited search
5. Iterative deepening search

Didn’t we already learn this in CS2040S?

Recap: CS2040S

- Focus was on SSSP and APSP

Condition	Algorithm	Time Complexity
No Negative Weight Cycles	Bellman-Ford Algorithm	$O(VE)$
On Unweighted Graph (or equal weights)	BFS	$O(V+E)$
No Negative Weights	Dijkstra's Algorithm	$O((V+E) log\, V)$
On Tree	BFS / DFS	$O(V)$
On DAG	Topological Sort	$O(V+E)$

Here, in CS2109S

- Might not care about shortest path
- Might not care about the path
- Goal might not be reachable (but we care about how close we can get)
- Implementation details might differ depending on what we care about. You should have learnt enough in CS2040S to deal with it.

Focus: Problem formulation + key ideas on how we can try to solve some classes of “AI” problems

Trade offs!

Breadth-first search

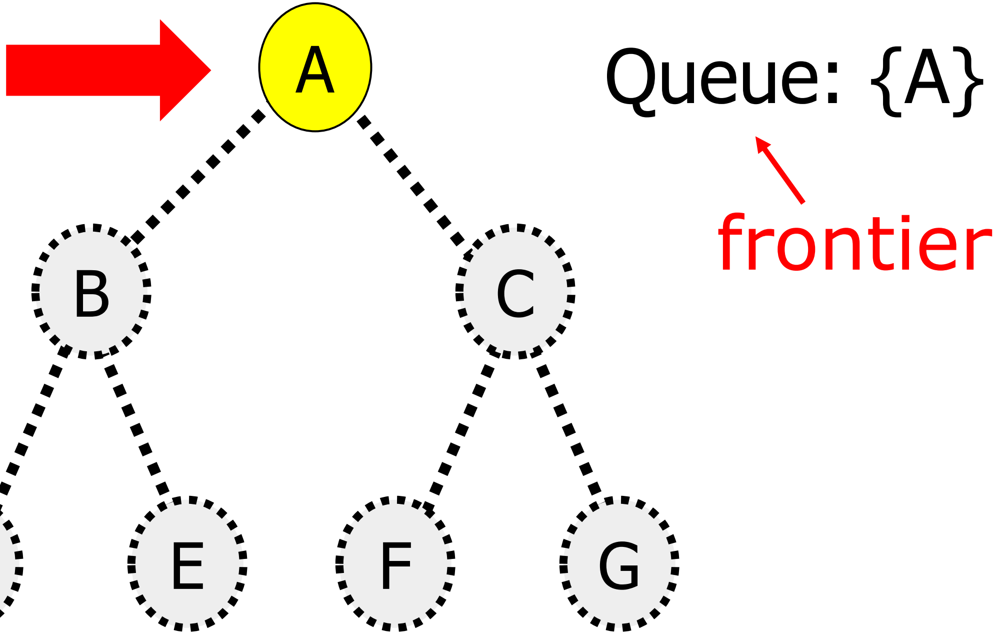
Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

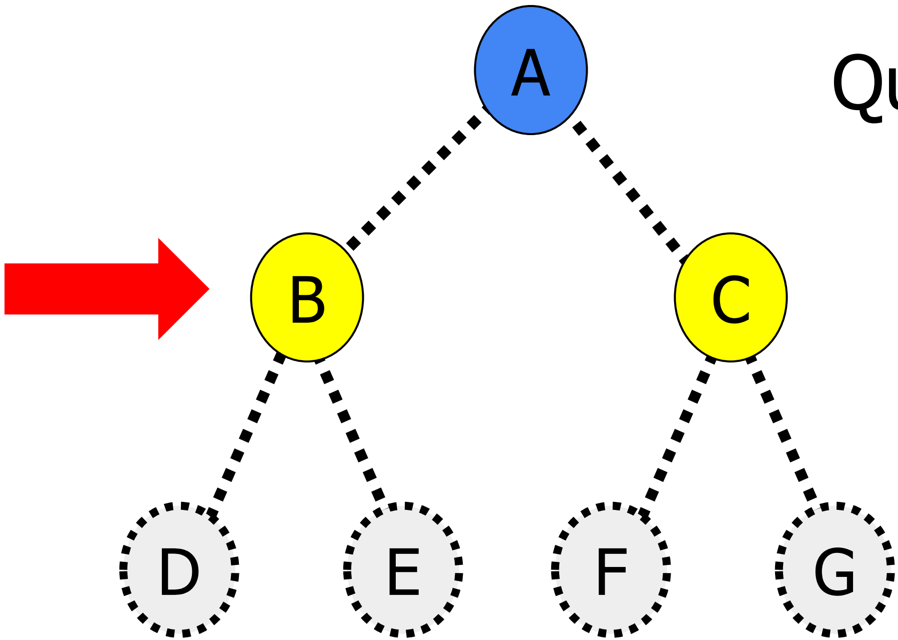
Breadth-first search

Expand shallowest unexpanded node



Breadth-first search

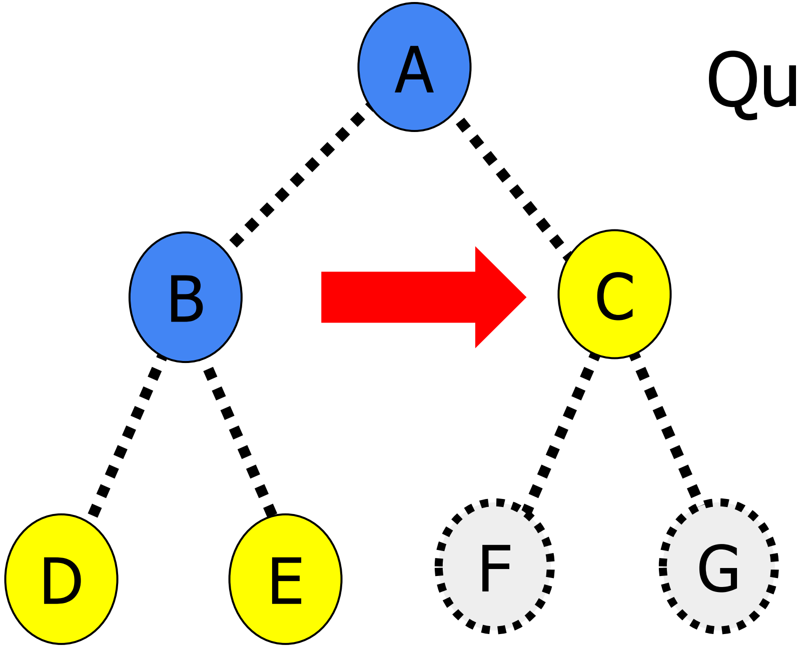
Expand shallowest unexpanded node



Queue: {B,C}

Breadth-first search

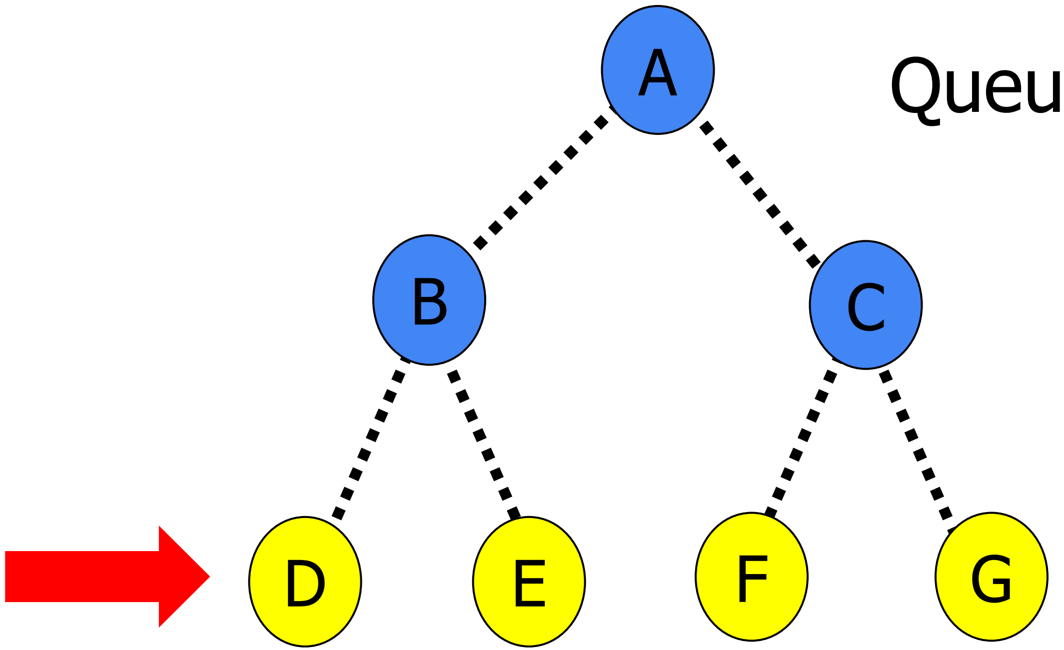
Expand shallowest unexpanded node



Queue: {C,D,E}

Breadth-first search

Expand shallowest unexpanded node



Queue: {D,E,F,G}

Properties of breadth-first search

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

frontier = queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Properties of Uniform-cost search

- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with *g* ≤ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with *g* ≤ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of *g(n)*

Recap: CS2040S

Condition	Algorithm	Time Complexity
No Negative Weight Cycles	Bellman-Ford Algorithm	$O(VE)$
On Unweighted Graph (or equal weights)	BFS	$O(V+E)$
No Negative Weights	Dijkstra's Algorithm	$O((V+E) log\, V)$
On Tree	BFS / DFS	$O(V)$
On DAG	Topological Sort	$O(V+E)$

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

Coming up in problem set

Depth-first search

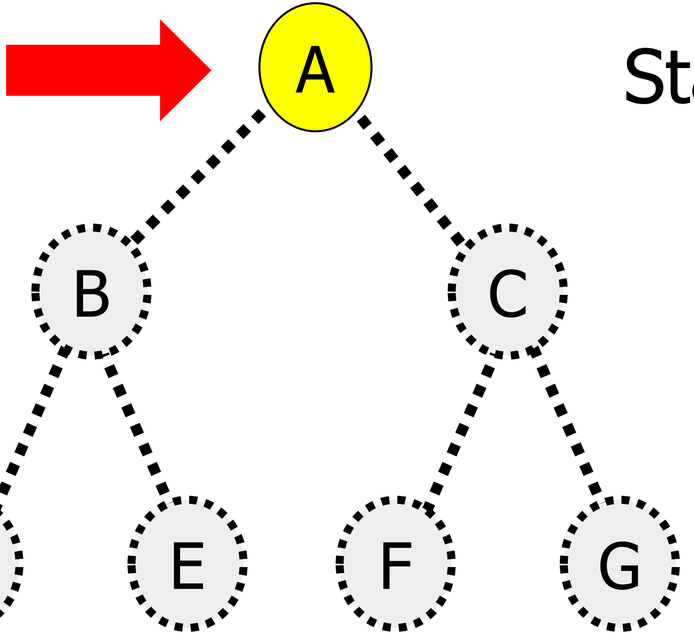
Expand deepest unexpanded node

Implementation:

frontier is a stack, i.e., new successors go to the front

Depth-first search

Expand deepest unexpanded node

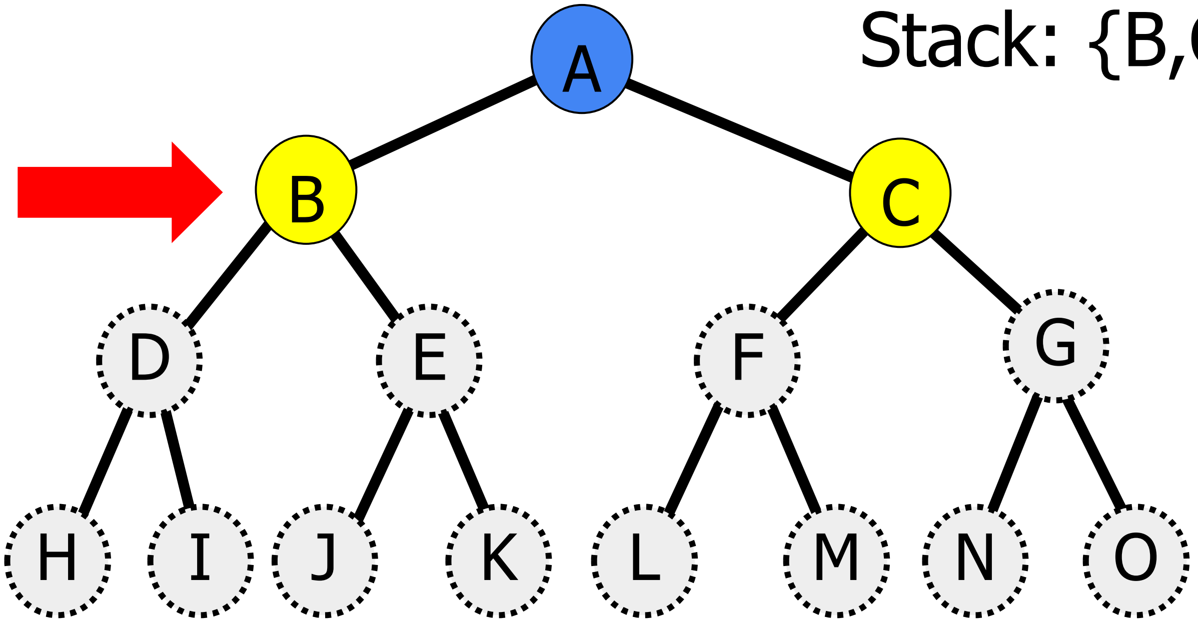


Stack: {A}

Depth-first search

Expand deepest unexpanded node

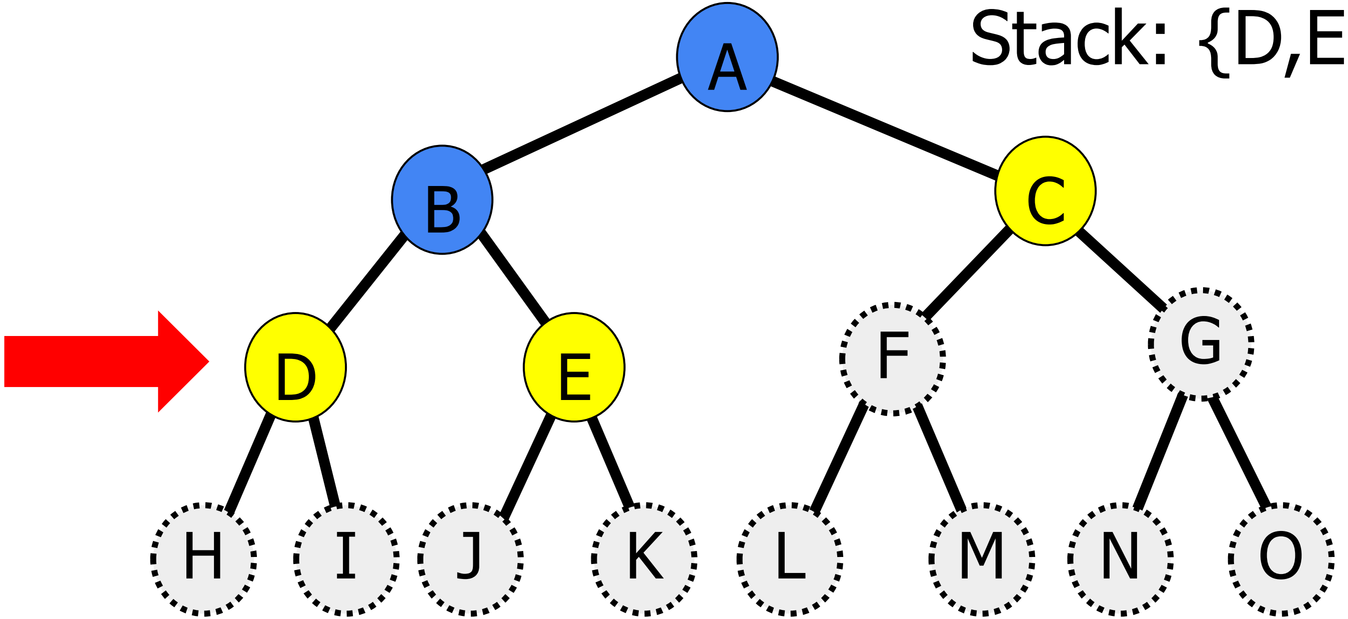
Stack: {B,C}



Depth-first search

Expand deepest unexpanded node

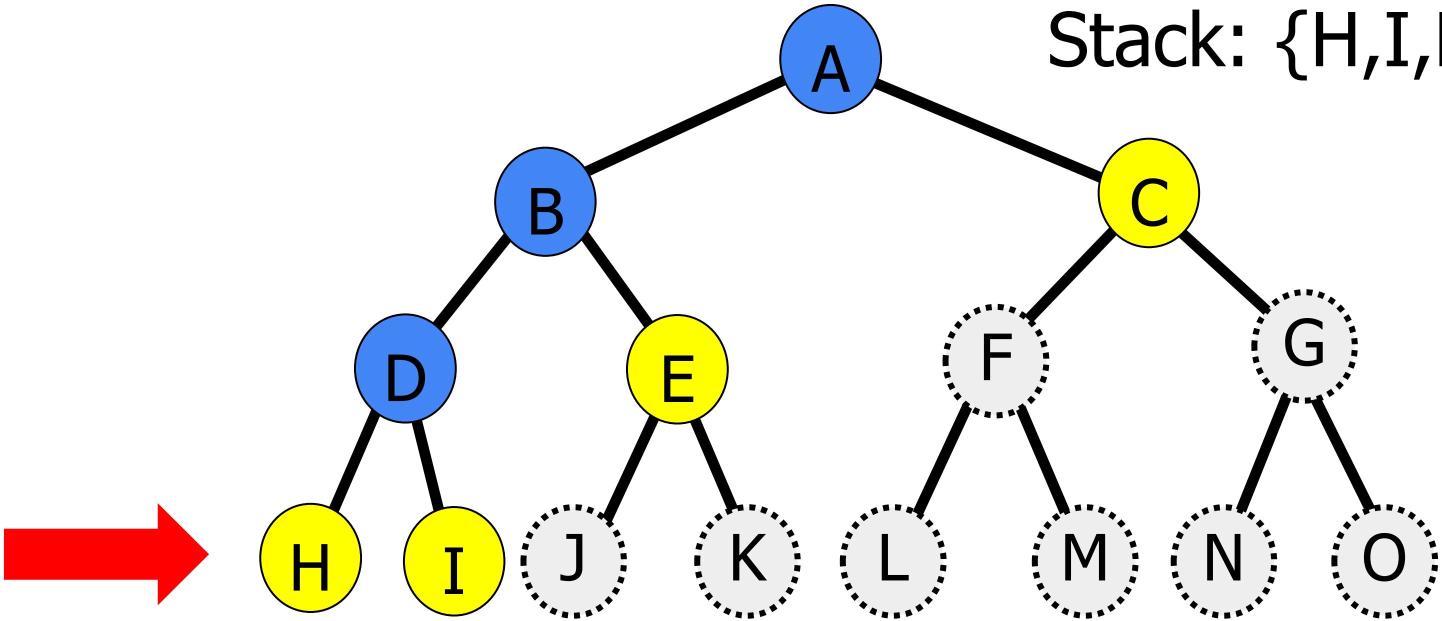
Stack: {D,E,C}



Depth-first search

Expand deepest unexpanded node

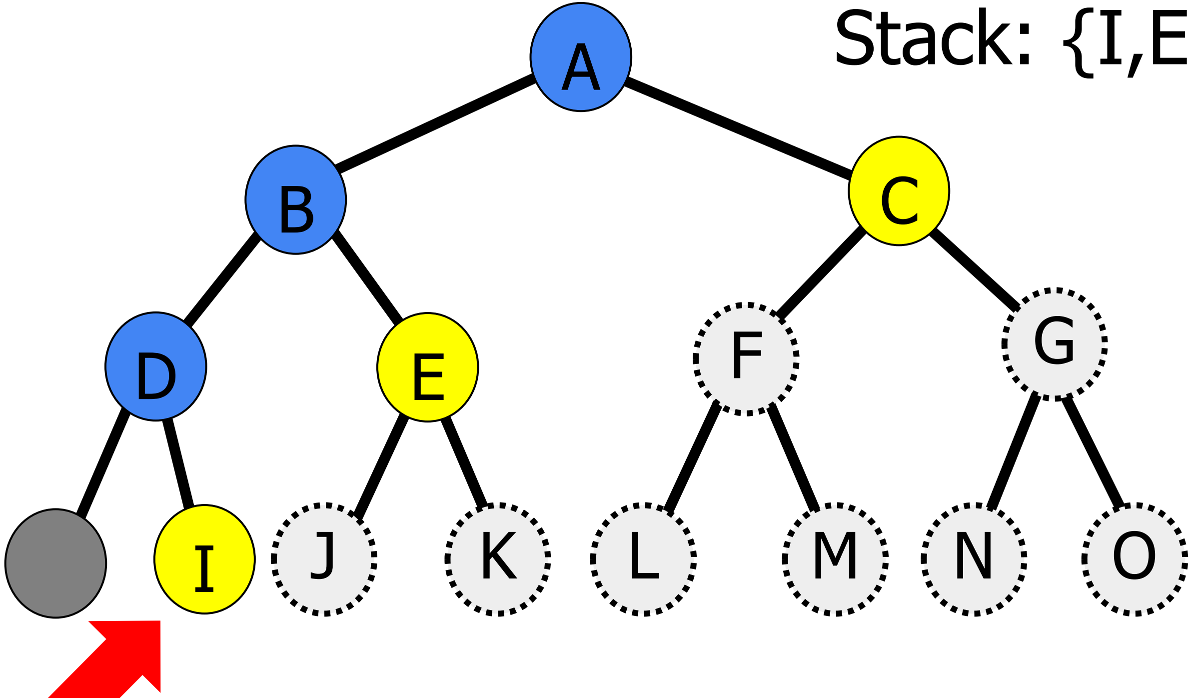
Stack: {H,I,E,C}



Depth-first search

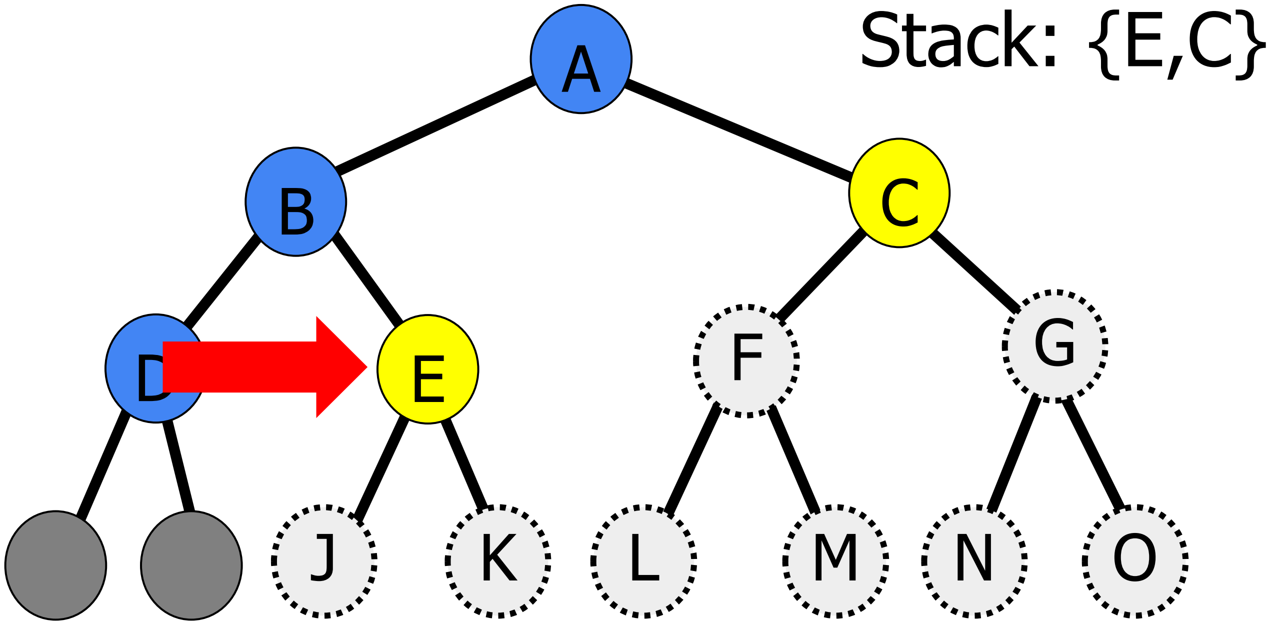
Expand deepest unexpanded node

Stack: {I,E,C}



Depth-first search

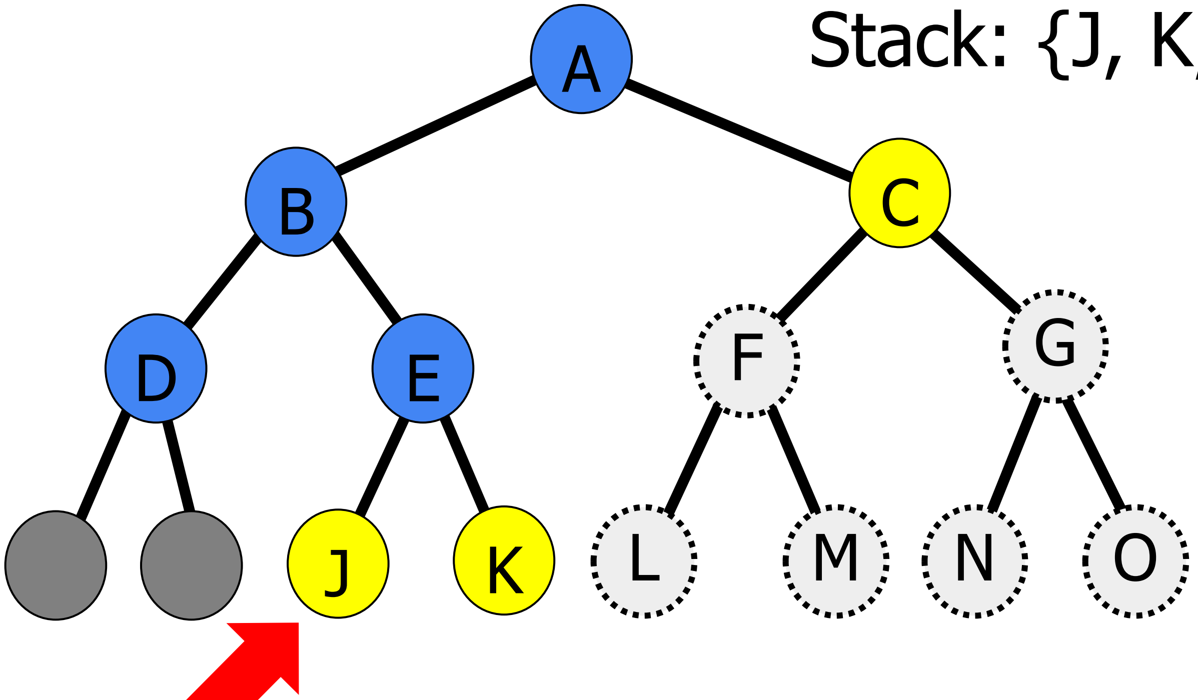
Expand deepest unexpanded node



Depth-first search

Expand deepest unexpanded node

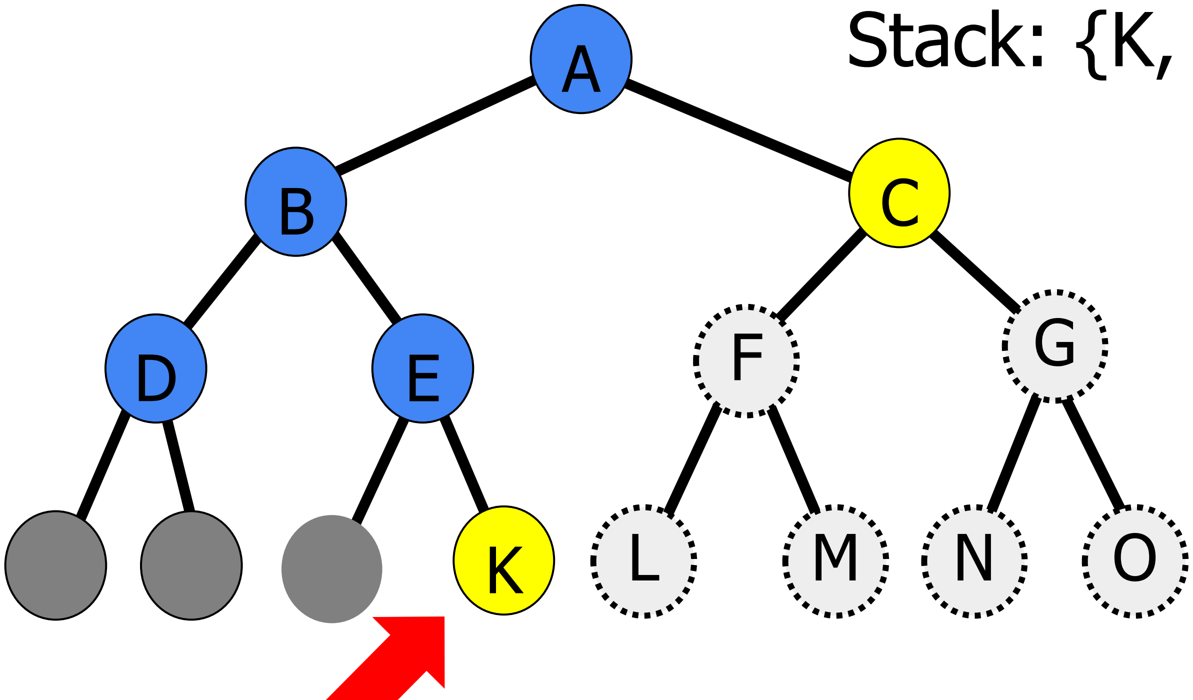
Stack: {J, K, C}



Depth-first search

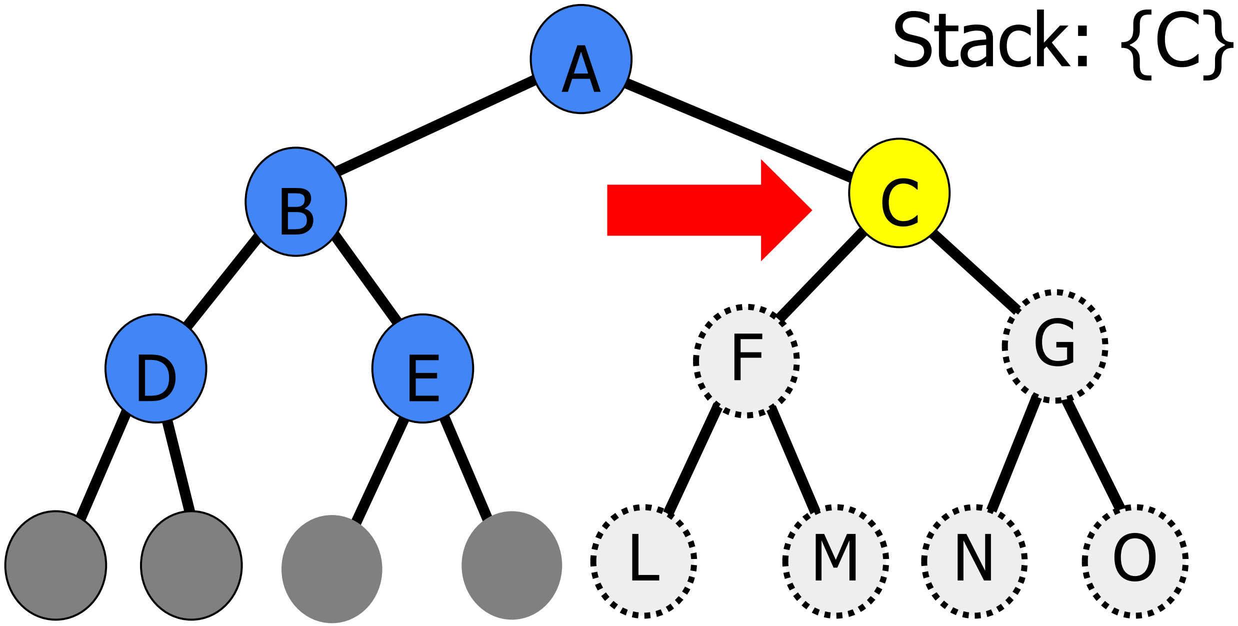
Expand deepest unexpanded node

Stack: {K, C}



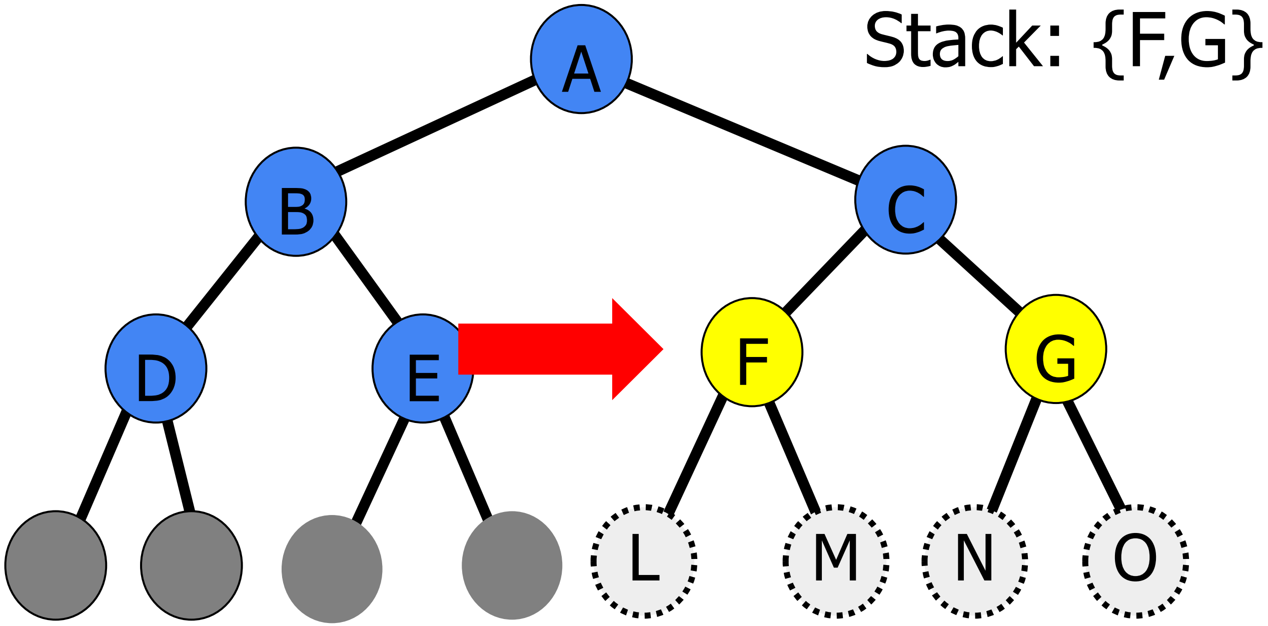
Depth-first search

Expand deepest unexpanded node



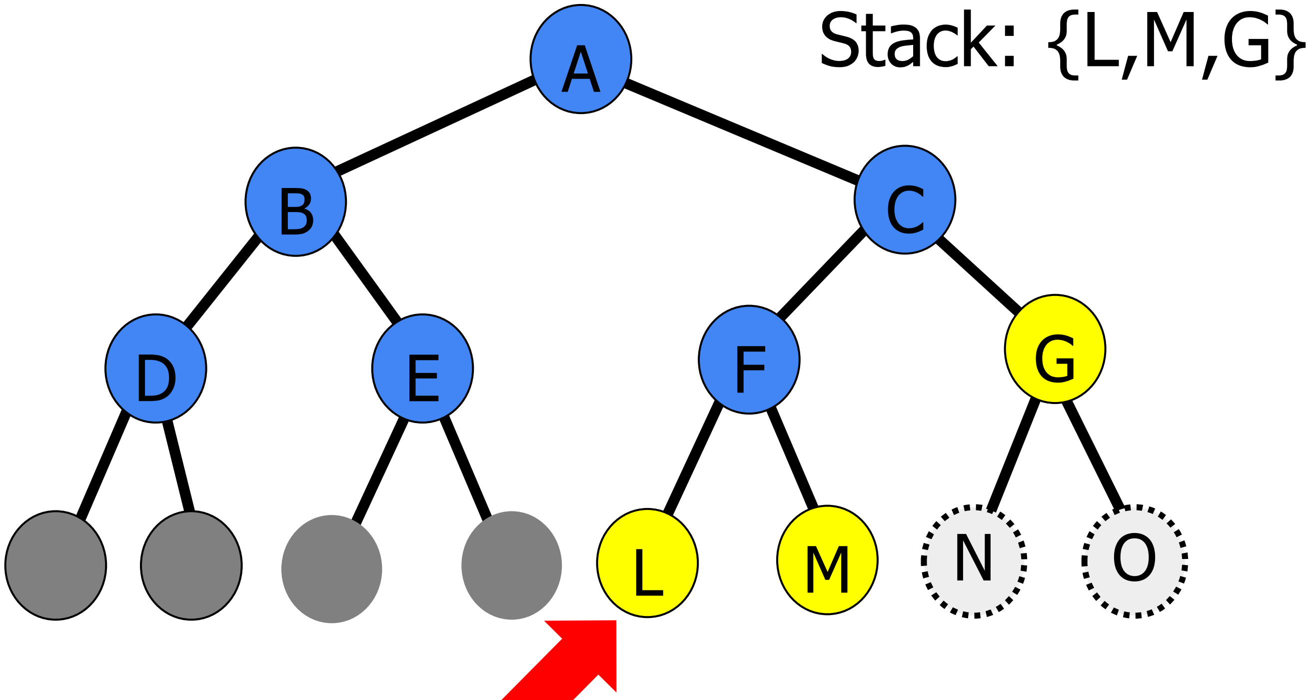
Depth-first search

Expand deepest unexpanded node



Depth-first search

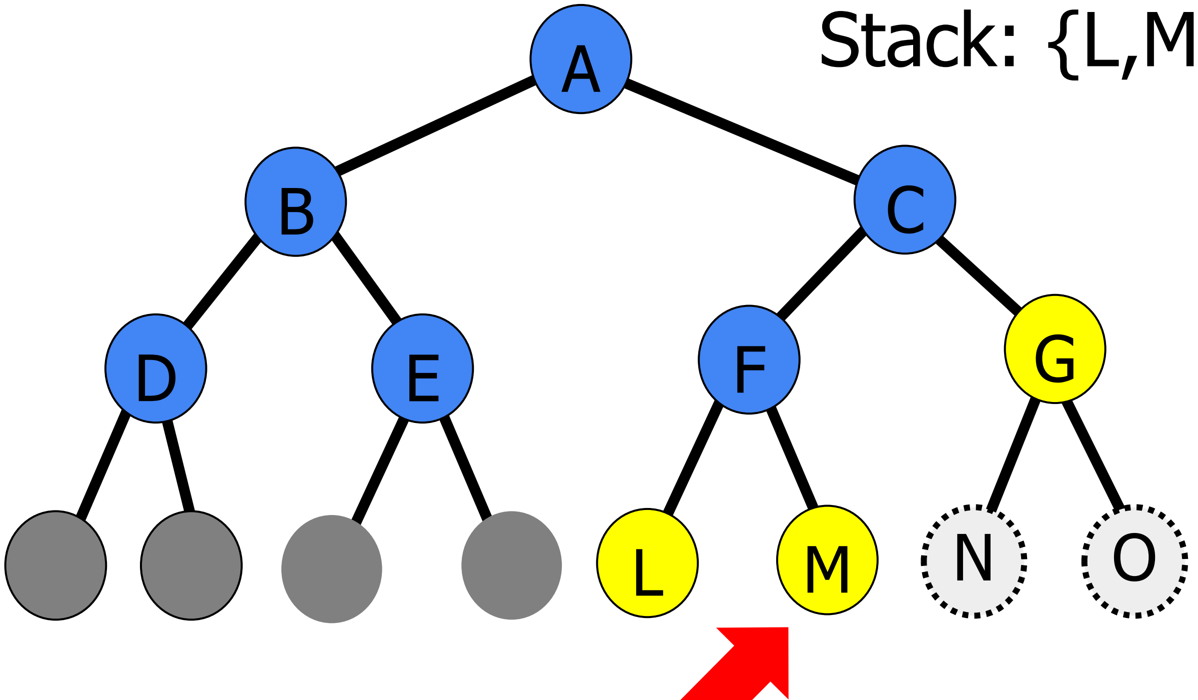
Expand deepest unexpanded node



Depth-first search

Expand deepest unexpanded node

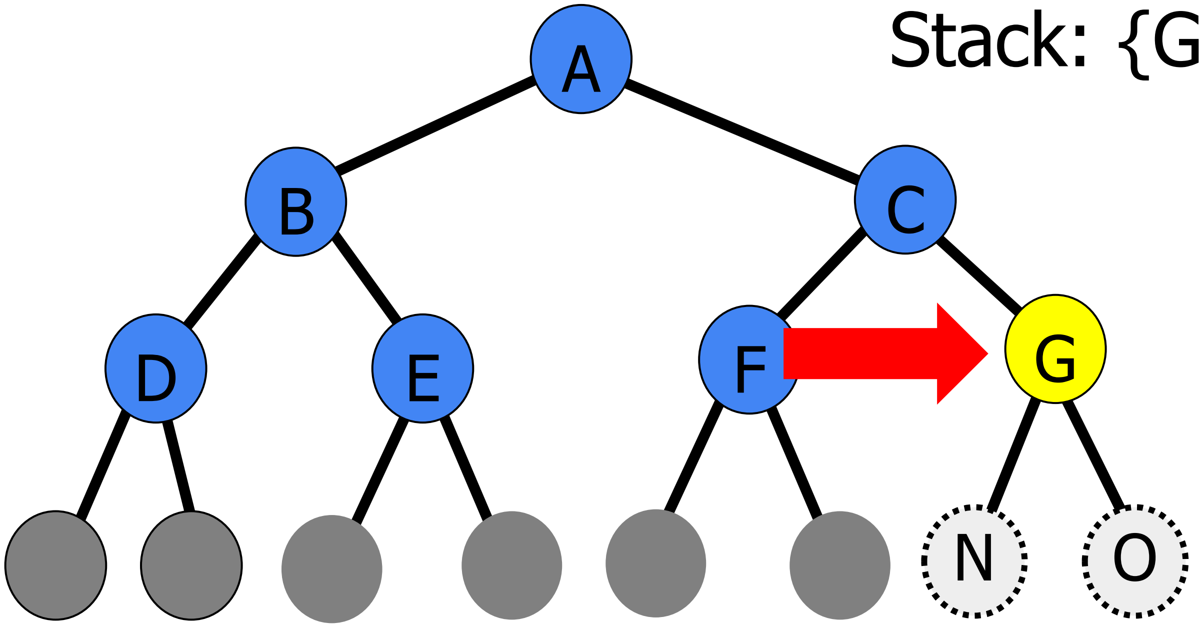
Stack: {L,M,G}



Depth-first search

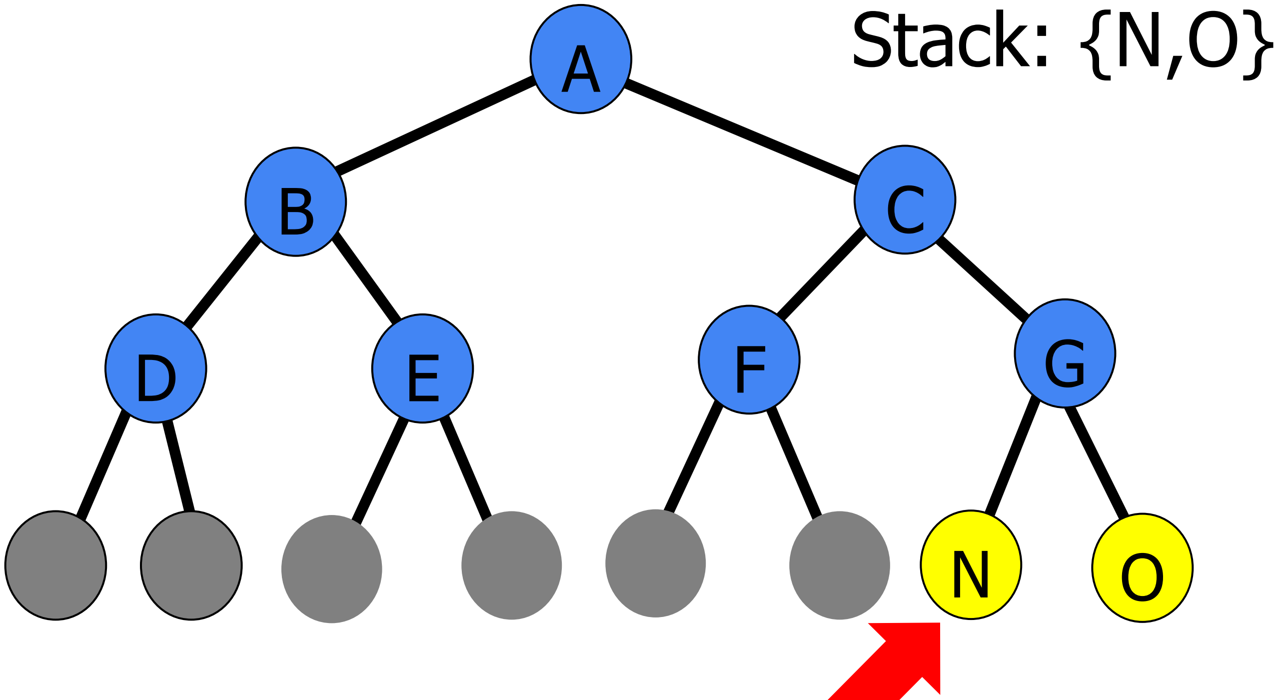
Expand deepest unexpanded node

Stack: {G}



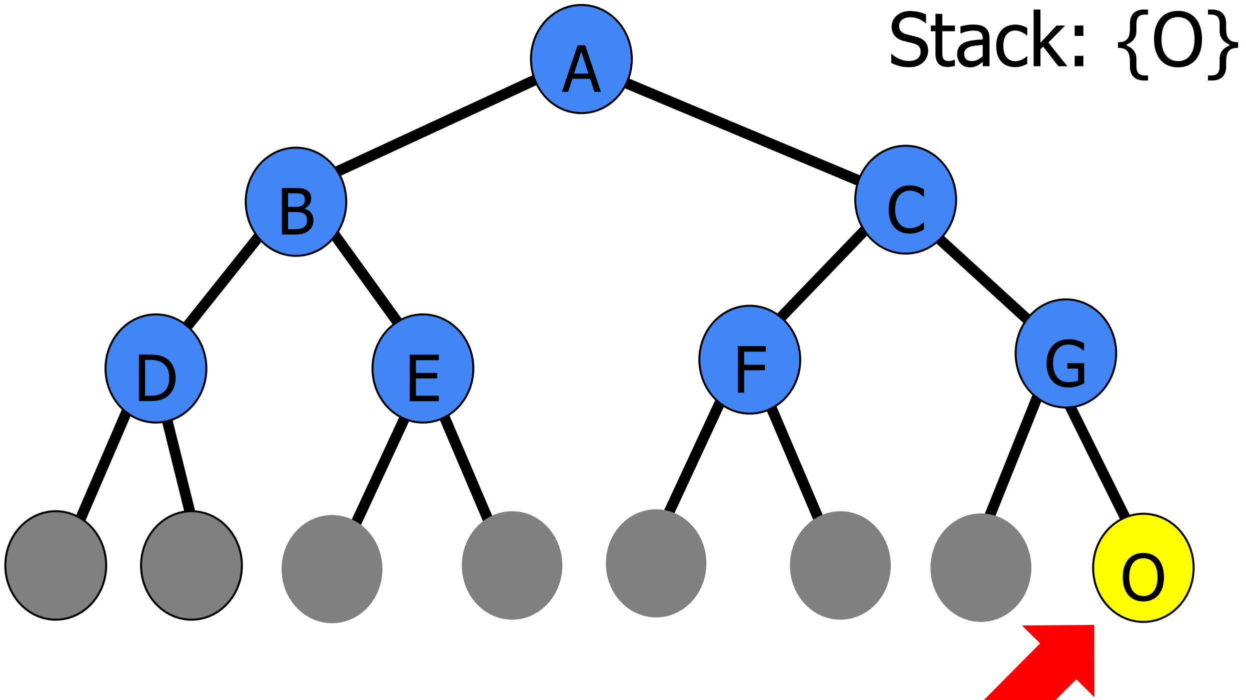
Depth-first search

Expand deepest unexpanded node



Depth-first search

Expand deepest unexpanded node



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

What happens if search space is of infinite depth?

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

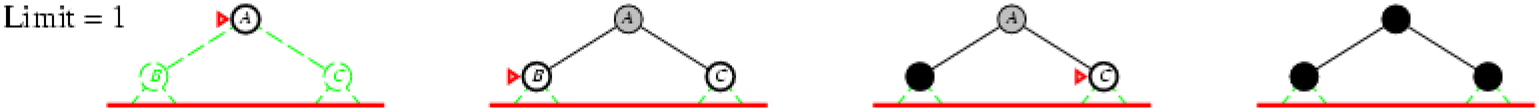
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or fail-
ure
    inputs: *problem*, a problem

    for $depth \leftarrow 0$ **to** ∞ **do**
        $result \leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
        if $result \neq$ cutoff **then return** *result*

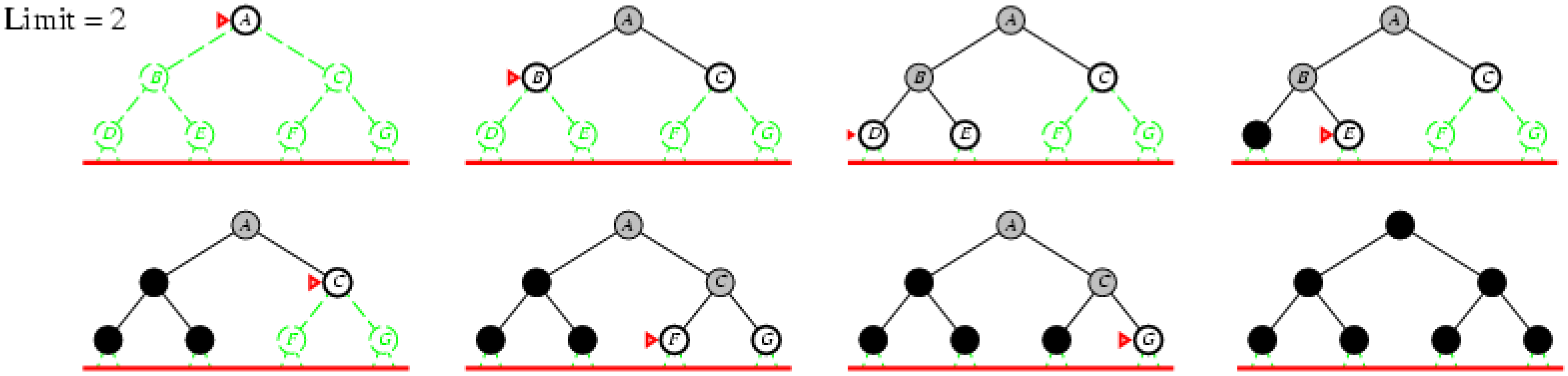
Iterative deepening search / =0

Limit = 0

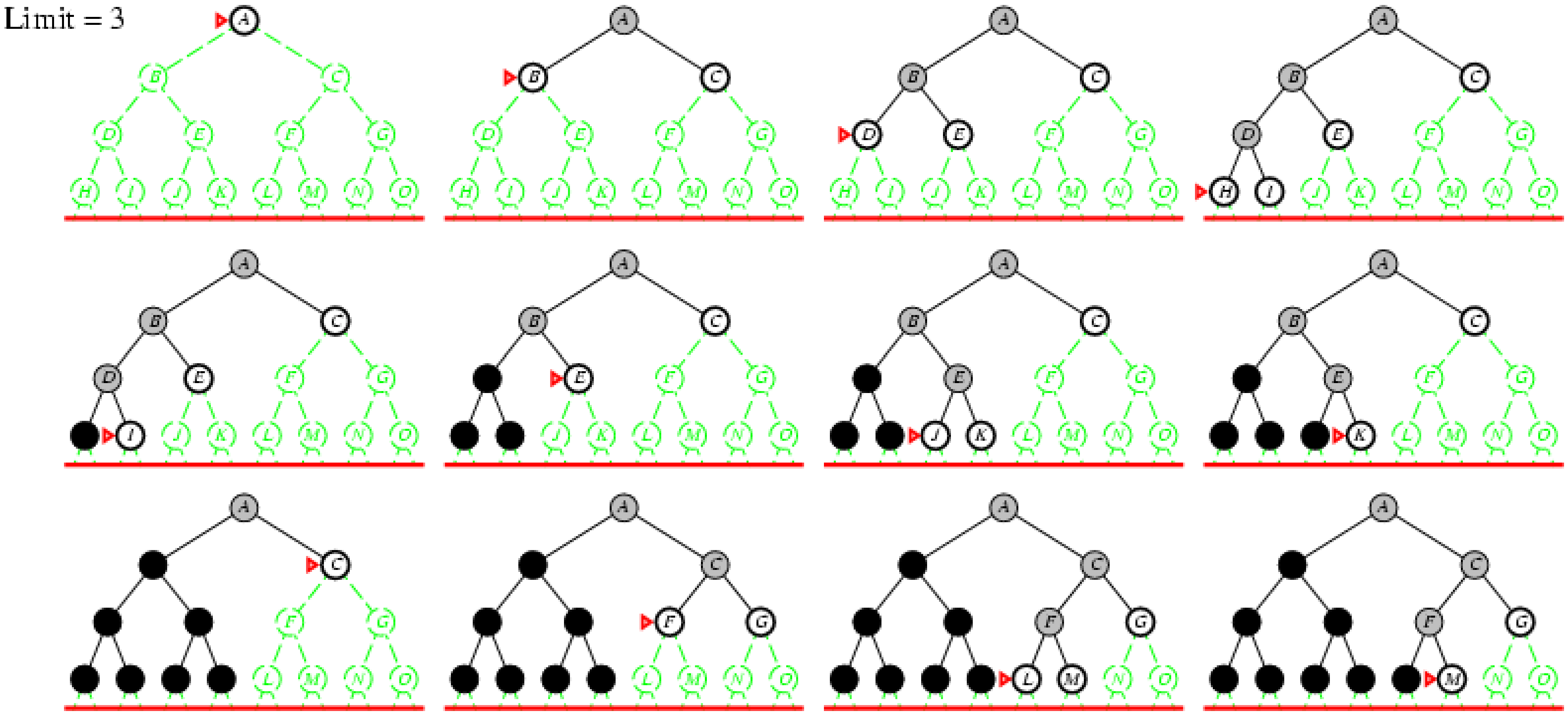
Iterative deepening search / =1



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*:

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

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

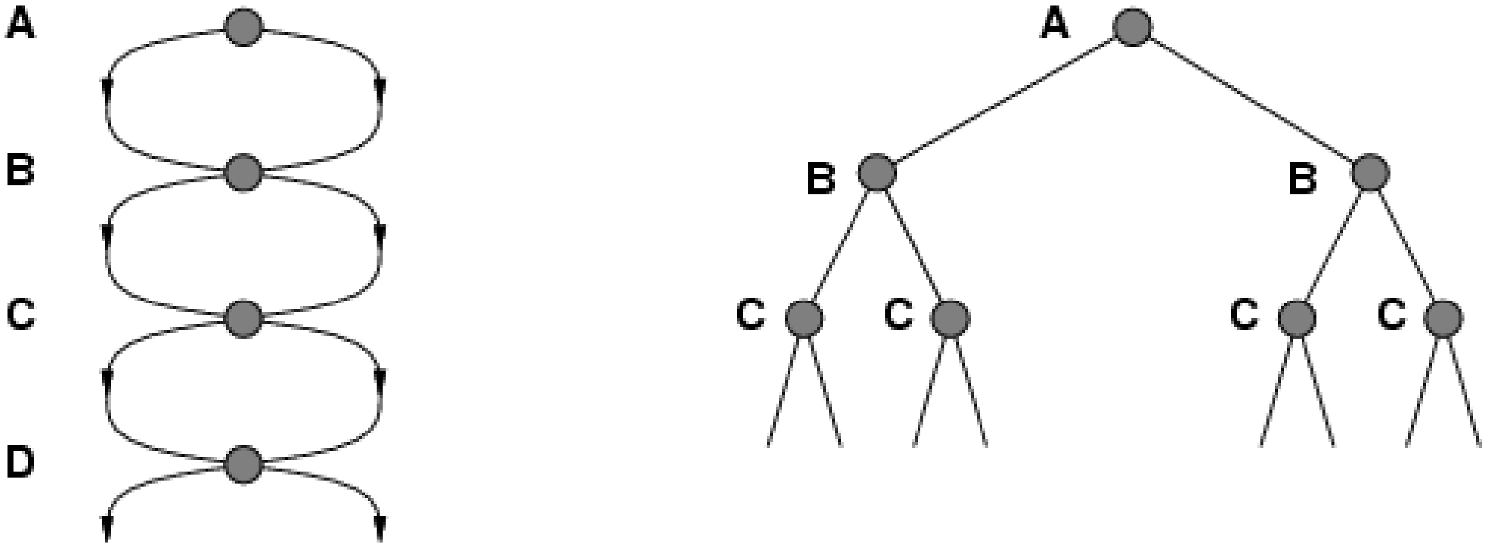
Summary of algorithms

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

b: Branching factor
d: Depth of optimal solution
C^*: Cost of optimal solution
l/m: Depth of search tree

Repeated states

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



Dealing with Repeated states

How do we avoid repeats/loops?

Remember the nodes that we have already visited! ☺

Memoization!

Graph search

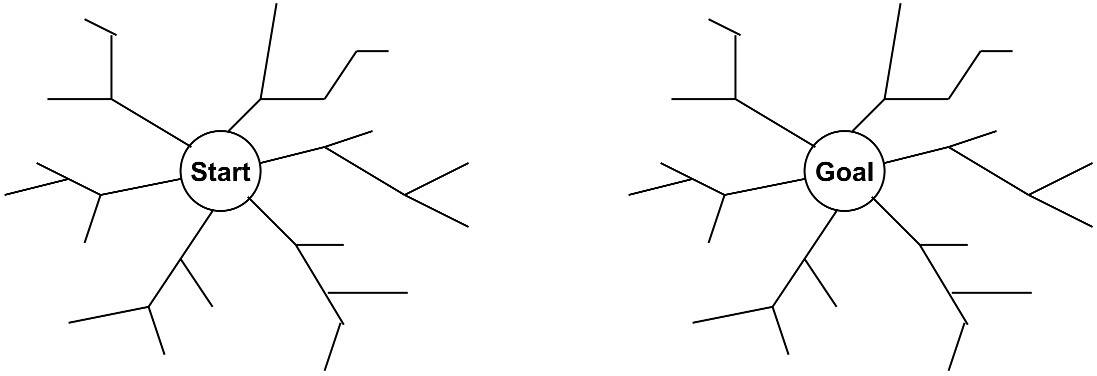
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem](STATE[node]) then return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

Bidirectional Search

Simultaneously search both forward (from the initial state) and backward (from the goal state)

Stop when the two searches meet.

Intuition = 2 x $O(b^{d/2})$ is smaller than $O(b^d)$



Bidirectional Search Discussion

- Numerical Example (b=10, l = 5)
  - Bi-directional search finds solution at d=3 for both forward and backward search. Assuming BFS in each half 2222 nodes are expanded.
- Implementation issues:
  - Operators are reversible, e.g., Pred(Succ(n)) = Pred(Succ(n))
  - There may be many possible goal states.
    - Construct a goal state containing the superset of all goal states.
  - Check if a node appears in the “other” search tree.
  - Using different search strategies for each half.

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 available

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening search

Summary: Search problem formulation

1. initial state

2. actions or successor function

3. goal test/goal state

4. path cost

Unique? Many?

Does it always exist?

Do we care about the path?

Don’t memorize anything

Focus on concepts & understanding!

Application + analysis!

Context matters

Answer to every question:

It depends

Question(s) of the Day

- Which is the best search strategy among the following uninformed search methods?
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
- What does “uninformed” mean?
- When should we use each method of search?

Question(s) of the Day

- What happens if instead of using a stack or queue for the fridge, we insert new nodes at random positions?
- What happens if we don’t know if we can reach the goal state and the search space is too large?

Tutorials Start Next Week!

Your allocated tutorials can be found in Coursemology Workbin.

Stragglers/problems, contact TA Austen to sort it out.

+600 EXP

Do not appeal through EduRec