

# Chapter 3 Solving Problems By Searching

## 3.1 –3.4 Uninformed search strategies

CS4811 - Artificial Intelligence

Nilufer Onder  
Department of Computer Science  
Michigan Technological University

# Outline

Problem-solving agents

Problem formulation

Basic search algorithms

- Tree search

- Graph search

Evaluating search strategies

Uninformed search strategies

- Breadth-first search

- Uniform-cost search

- Depth-first search

- Depth-limited search

- Iterative deepening search

- Bidirectional search

# Problem-solving agents

**function** SIMPLE-PROBLEM-SOLVING-AGENT (*percept*)

**returns** an action

**inputs:** *percept*, a percept

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

*state*, some description of the current world state

*goal*, a goal, initially null

*problem*, a problem formulation

*state*  $\leftarrow$  UPDATE-STATE (*state*, *percept*)

**if** *seq* is empty **then**

*goal*  $\leftarrow$  FORMULATE-GOAL (*state*)

*problem*  $\leftarrow$  FORMULATE-PROBLEM (*state*, *goal*)

*seq*  $\leftarrow$  SEARCH (*problem*)

**if** *seq* = *failure* **then return** a null action

*action*  $\leftarrow$  FIRST (*seq*)

*seq*  $\leftarrow$  REST (*seq*)

**return** *action*

# Assumptions

- ▶ *Static*: The world does not change unless the agent changes it.
- ▶ *Observable*: Every aspect of the world state can be seen.
- ▶ *Discrete*: Has distinct states as opposed to continuously flowing time.
- ▶ *Deterministic*: There is no element of chance.

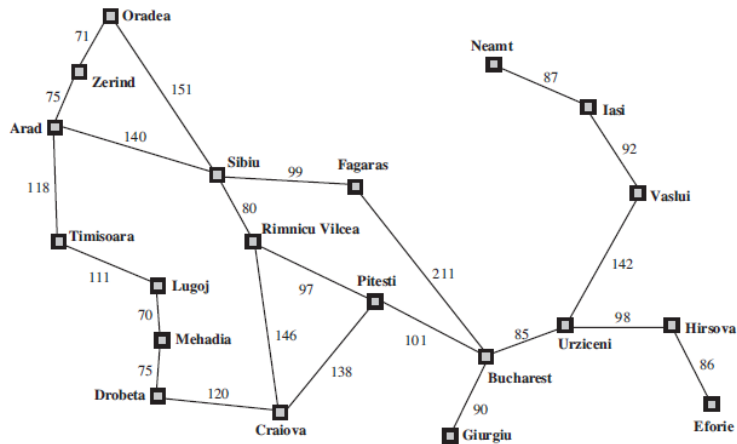
This is a restricted form of a general agent called *offline* problem solving. The solution is executed “eyes closed.”

*Online* problem solving involves acting without complete knowledge

## Example: Traveling in 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  
(any solution or optimal solution?)

# Distances between cities in Romania



# Infrastructure for search algorithms

- ▶ A *problem* is defined by five components:
  - ▶ *initial state* e.g., “In(Arad)”
  - ▶ *actions*,  $ACTIONS(s)$  returns the actions applicable in  $s$ .  
e.g, In Arad, the applicable actions are  
{Go(Sibiu), Go(Timisoara), Go(Zerind)}
  - ▶ *transition model*,  $RESULT(s, a)$  returns the state that results from executing action  $a$  in state  $s$   
e.g.,  $RESULT(In(Arad), Go(Zerind)) = In(Zerind)$ .
  - ▶ *goal test*, can be  
*explicit*, e.g.,  $x = \text{“In Bucharest”}$   
*implicit*, e.g.,  $x = \text{“In a city with an international airport”}$
  - ▶ *path cost* (additive)  
e.g., sum of distances, number of actions executed, etc.  
 $c(x, a, y)$  is the *step cost* of executing action  $a$  in state  $x$  and arriving at state  $y$ , assumed to be  $\geq 0$
- ▶ A *solution* is a sequence of actions leading from the initial state to a goal state

## Selecting a state space

- ▶ The 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!
- ▶ Find an abstraction that is *valid* and *useful*.



## Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

## Example: The 8-puzzle (cont'd)

- ▶ **states:** integer locations of tiles  
(ignore intermediate positions)
- ▶ **actions:** move blank left, right, up, down  
(ignore unjamming etc.)
- ▶ **goal test:** = goal state (given)
- ▶ **path cost:** 1 per move
- ▶ Note that the optimal solution of  $n$ -Puzzle family is NP-hard

# Tree search algorithms

Basic idea:

offline, simulated exploration of state space

by generating successors of the states that haven't been explored  
(a.k.a. *expanding* states)

## Tree search algorithms (cont'd)

**function** TREE-SEARCH (*problem*, *strategy*)

**returns** a solution, or failure

initialize the frontier using the initial state of *problem*

**loop do**

**if** the frontier is empty **then return** failure

    choose a leaf node and remove it from the frontier

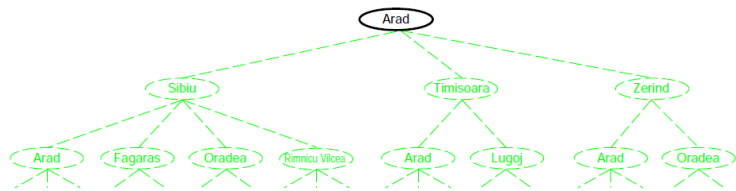
**if** the node contains a goal state

**then return** the corresponding solution

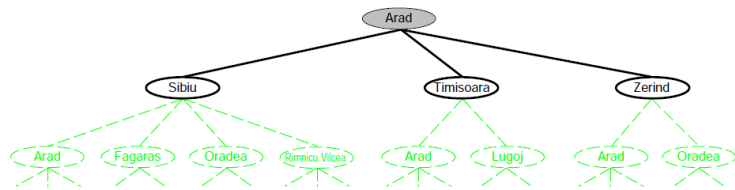
    expand the chosen node and add the resulting nodes to the frontier

**end**

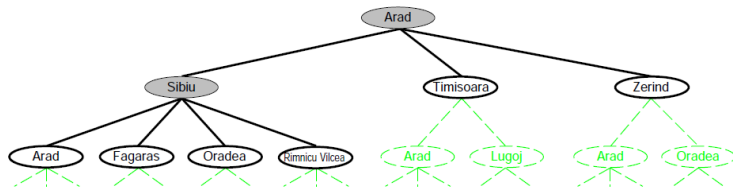
# Tree search example



# Tree search example

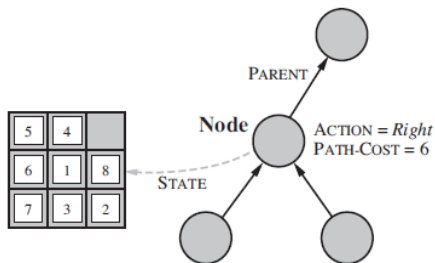


# Tree search example



## Implementation: states vs. nodes

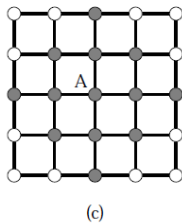
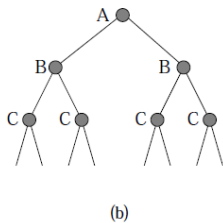
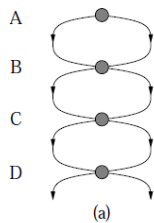
- ▶ A *state* is a (representation of) a physical configuration.
- ▶ A *node* is a data structure constituting part of a search tree
- ▶ A node includes: *parent*, *children*, *depth*, *path cost*  $g(x)$ .
- ▶ States do not have parents, children, depth, or path cost!
- ▶ The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.





## Repeated states

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



# Graph search algorithms

Basic idea:

similar to tree-search

keep a separate list of “explored” states

## Graph search algorithms (cont'd)

**function** GRAPH-SEARCH (*problem*)

**returns** a solution, or failure

initialize the frontier using the initial state of *problem*

→ initialize the explored set to be empty

**loop do**

**if** the frontier is empty **then return** failure

    choose a leaf node and remove it from the frontier

**if** the node contains a goal state

**then return** the corresponding solution

→ add the node to the explored set

expand the chosen node and add the resulting nodes to the frontier

→ only if not in the frontier or explored set

**end**

Note: A → shows the lines that are added to the tree search algorithm.

# Evaluating search strategies

- ▶ A 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/expanded
  - ▶ *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* 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
- ▶ Bidirectional search

# Breadth-first search

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

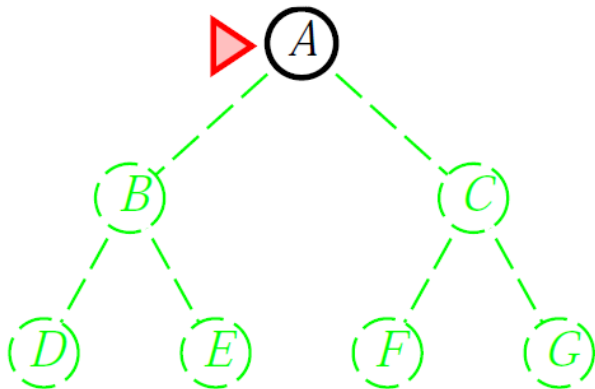
## Progress of breadth-first search

Breadth-first search on a simple binary tree.

At each stage, the node to be expanded next is indicated by a marker.

The nodes that are already explored are gray.

The nodes with dashed lines are not generated yet.



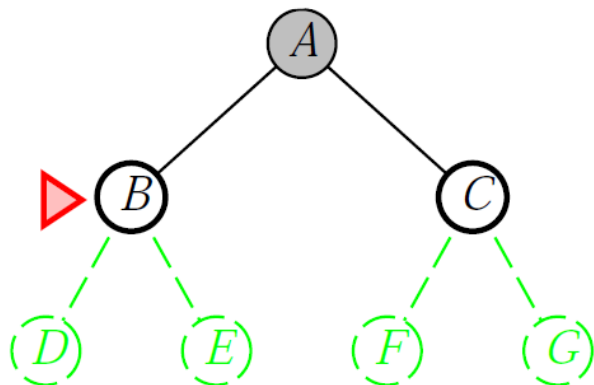
## Progress of breadth-first search

Breadth-first search on a simple binary tree.

At each stage, the node to be expanded next is indicated by a marker.

The nodes that are already explored are gray.

The nodes with dashed lines are not generated yet.





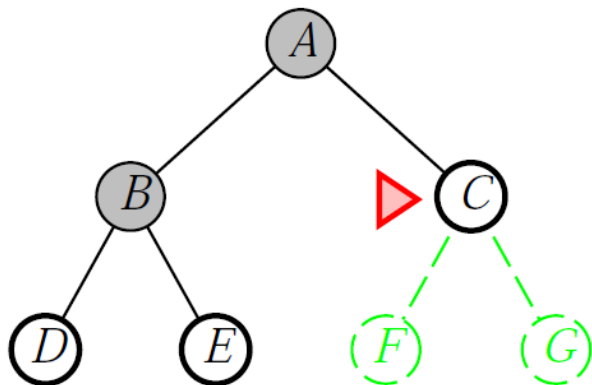
## Progress of breadth-first search

Breadth-first search on a simple binary tree.

At each stage, the node to be expanded next is indicated by a marker.

The nodes that are already explored are gray.

The nodes with dashed lines are not generated yet.



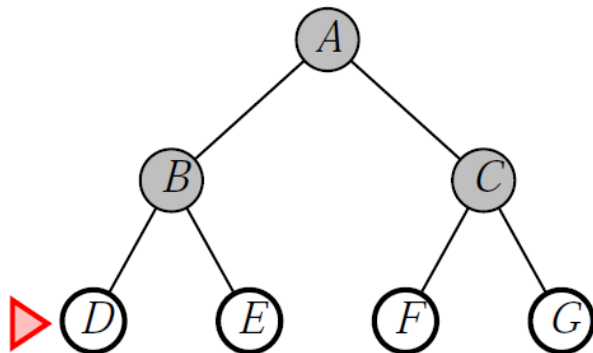
## Progress of breadth-first search

Breadth-first search on a simple binary tree.

At each stage, the node to be expanded next is indicated by a marker.

The nodes that are already explored are gray.

The nodes with dashed lines are not generated yet.



# Properties of breadth-first search

- ▶ *Complete*: Yes (if  $b$  is finite)
- ▶ *Time*:  $b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$ ,  
i.e., number of nodes generated is exponential in  $d$
- ▶ *Space*:  $O(b^{d+1})$  (keeps every node in memory)
- ▶ *Optimal*: Yes (if cost = 1 per step)

Space is the big problem; can easily generate nodes at 100MB/sec  
so 24hrs = 8604GB.

# Breadth-first search algorithm

**function** BREADTH-FIRST-SEARCH (*problem*)

**returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE=*problem*.INITIAL-STATE,  
PATH-COST = 0

**if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

*frontier*  $\leftarrow$  a FIFO queue with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node*  $\leftarrow$  POP(*frontier*) /\* chooses the shallowest node in *frontier* \*/

add *node*.STATE to *explored*

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

*child*  $\leftarrow$  CHILD-NODE (*problem*,*node*, *action*)

**if** *child*.STATE is not in *explored* or *frontier* **then**

**if** *problem*.GOAL-TEST (*child*.STATE) **then**

**return** SOLUTION(*child*)

*frontier*  $\leftarrow$  INSERT (*child*, *frontier*)

# Uniform-cost search

- ▶ Expand the least-cost unexpanded node
- ▶ Implementation: *frontier* is a queue ordered by path cost
- ▶ Equivalent to breadth-first if step costs are all equal

# Properties of uniform-cost search

- ▶ **Complete:** Yes, if step cost  $\geq \epsilon$
- ▶ **Time:** # of nodes with  $g \leq$  cost of optimal solution,  
 $O(b^{1+\lfloor C^*/\epsilon \rfloor})$   
where  $C^*$  is the cost of the optimal solution
- ▶ **Space:** # of nodes with  $g \leq$  cost of optimal solution,  
 $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- ▶ **Optimal:** Yes—nodes expanded in increasing order of  $g(n)$

# Uniform-cost search algorithm

**function** UNIFORM-COST-SEARCH (*problem*)

**returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE=*problem*.INITIAL-STATE,  
PATH-COST = 0

**if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

*frontier*  $\leftarrow$  a priority ordered by PATH-COST, with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node*  $\leftarrow$  POP(*frontier*) /\* chooses the lowest-cost node in *frontier* \*/  
add *node*.STATE to *explored*

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

*child*  $\leftarrow$  CHILD-NODE (*problem*,*node*, *action*)

**if** *child*.STATE is not in *explored* or *frontier* **then**

*frontier*  $\leftarrow$  INSERT (*child*, *frontier*)

**else if** *child*.STATE is in *frontier* with higher PATH-COST **then**

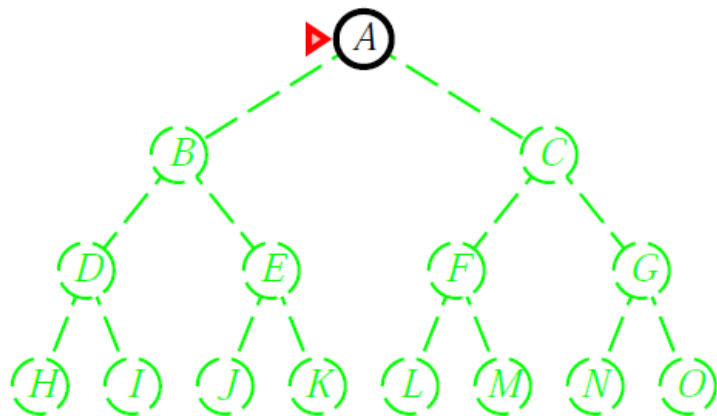
replace that *frontier* node with *child*

# Depth-first search

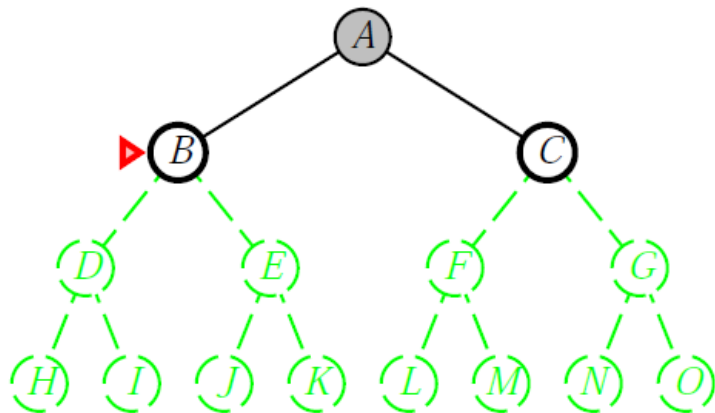
- ▶ Expand deepest unexpanded node
- ▶ Implementation: *frontier* is a LIFO queue, i.e., put successors at front



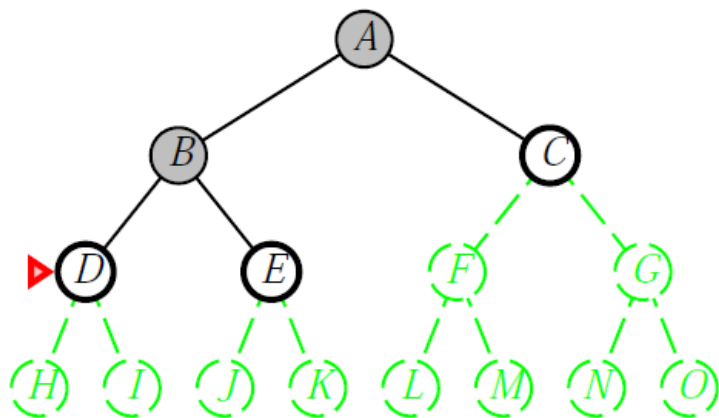
## Progress of depth-first search



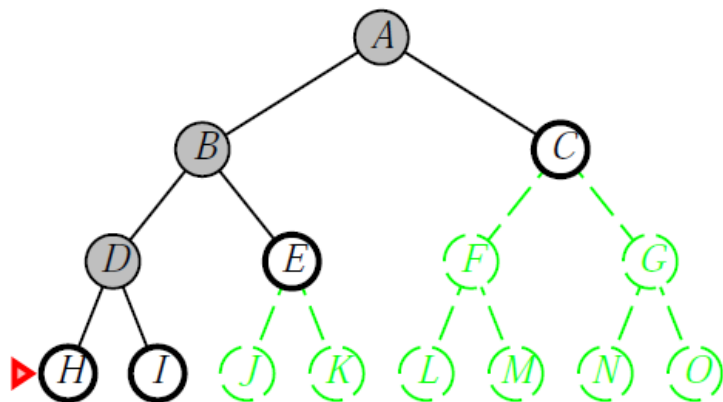
## Progress of depth-first search



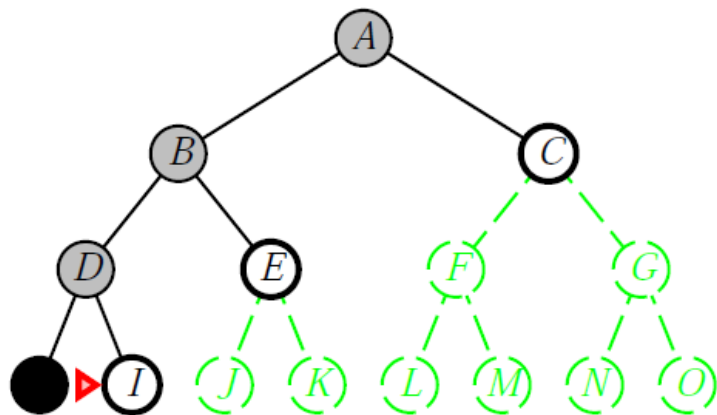
## Progress of depth-first search



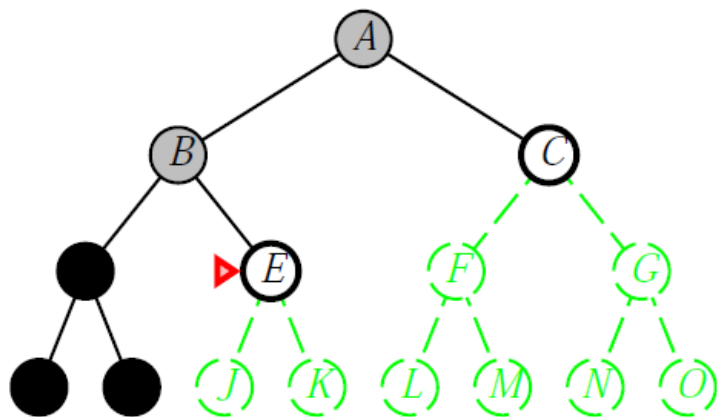
## Progress of depth-first search



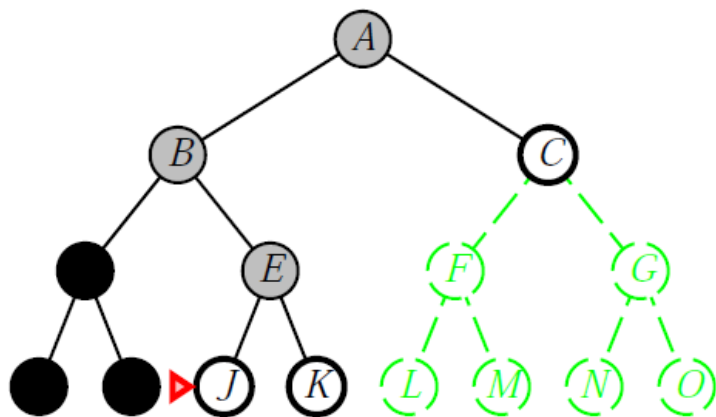
## Progress of depth-first search



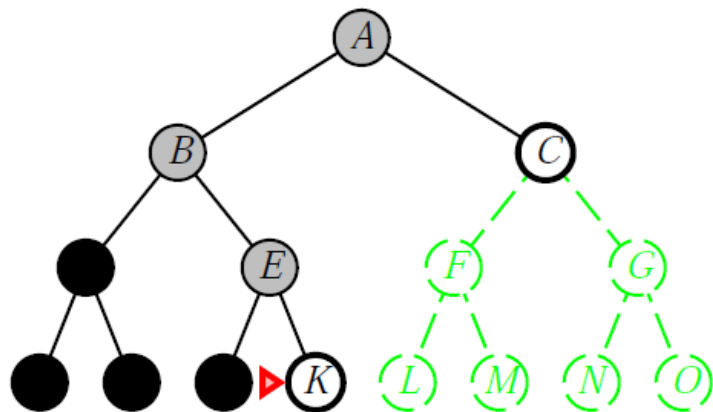
## Progress of depth-first search



## Progress of depth-first search

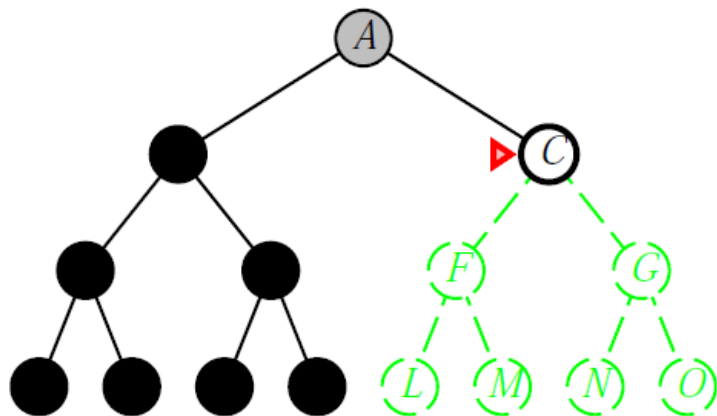


## Progress of depth-first search

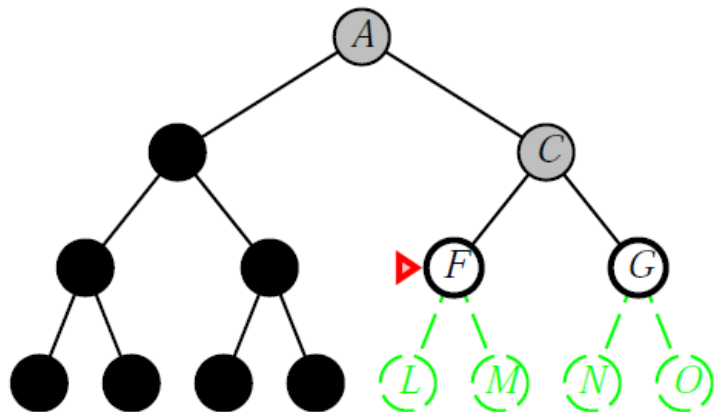




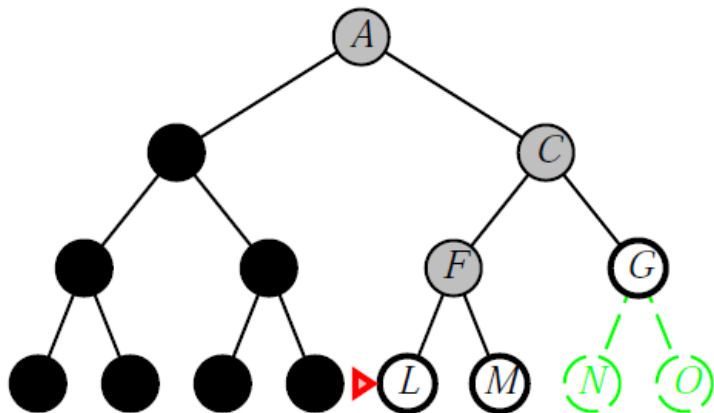
## Progress of depth-first search



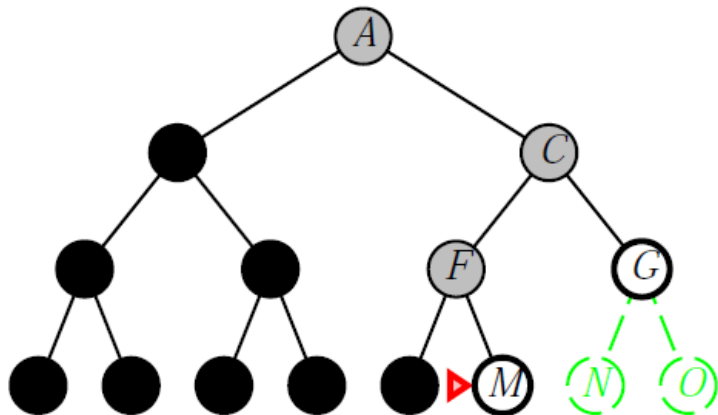
## Progress of depth-first search



## Progress of depth-first search



## Progress of depth-first search



# 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

- ▶ It is equivalent to depth-first search with depth limit  $l$ , i.e., nodes at depth  $l$  have no successors
- ▶ implementation: a recursive implementation is shown on the next page

# Properties of depth-limited search

- ▶ *Complete*: No (similar to DFS)
- ▶ *Time*:  $O(b^l)$ , where  $l$  is the depth-limit
- ▶ *Space*:  $O(bl)$ , i.e., linear space (similar to DFS)
- ▶ *Optimal*: No

## Depth-limited search

**function** DEPTH-LIMITED-SEARCH (*problem*, *limit*)  
**returns** a solution, or failure/cutoff  
**return** RECURSIVE-DLS(MAKE-NODE( *problem*.INITIAL-STATE),  
                  *problem*, *limit*)

**function** RECURSIVE-DLS (*node*, *problem*, *limit*)  
**returns** a solution, or failure/cutoff  
  **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)  
  **else if** *limit* = 0 **then return** *cutoff*  
  **else**  
    *cutoff-occurred?*  $\leftarrow$  false  
    **for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**  
      *child*  $\leftarrow$  CHILD-NODE (*problem*,*node*, *action*)  
      *result*  $\leftarrow$  RECURSIVE-DLS (*child*, *problem*,*limit*-1)  
      **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

- ▶ Do iterations of depth-limited search starting with a limit of 0. If you fail to find a goal with a particular depth limit, increment it and continue with the iterations.
- ▶ Terminate when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.
- ▶ Combines the linear space complexity of DFS with the completeness property of BFS.

# Iterative deepening search ( $l = 0$ )

Limit = 0



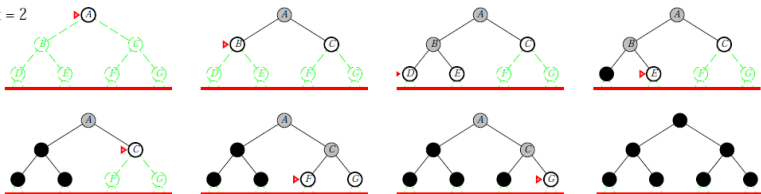
# Iterative deepening search ( $l = 1$ )

Limit = 1



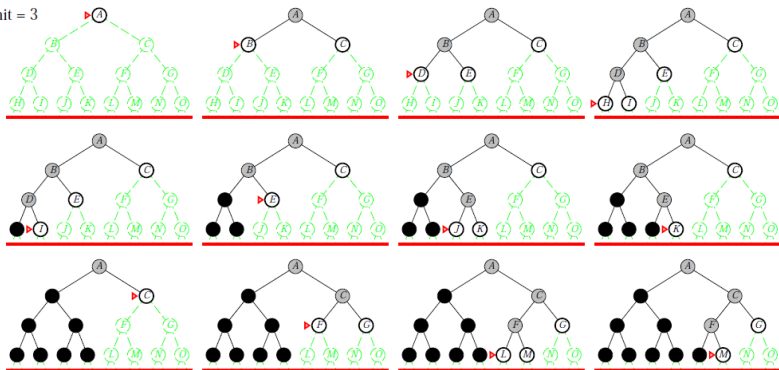
# Iterative deepening search ( $l = 2$ )

Limit = 2



# Iterative deepening search ( $l = 3$ )

Limit = 3



# Properties of iterative deepening search

- ▶ *Complete:* Yes
- ▶ *Time:*  $db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$
- ▶ *Space:*  $O(bd)$
- ▶ *Optimal:* Yes, if step cost = 1  
Can be modified to explore uniform-cost tree

# Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem)  
returns a solution, or failure  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH (problem, depth)  
    if result  $\neq$  cutoff then return result
```

## Compare IDS and BFS

Numerical comparison of the number of nodes generated for  $b = 10$  and  $d = 5$ , solution at the far right leaf:

$$\begin{aligned}N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 \\ &= 123,450\end{aligned}$$

$$\begin{aligned}N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 \\ &= 1,111,100\end{aligned}$$

IDS does better because other nodes at depth  $d$  are not expanded. BFS can be modified to apply the goal test when a node is generated (rather than expanded).

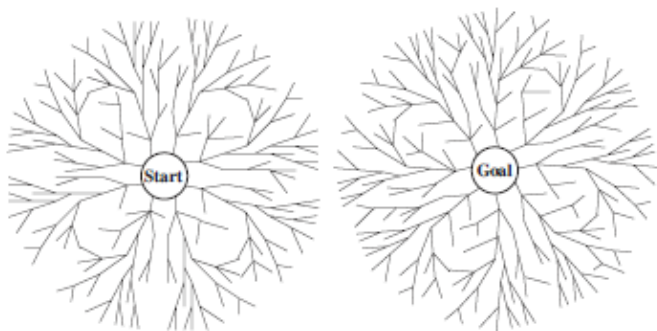


# Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iter. Deep.
Complete?	Yes	Yes	No	Yes	Yes
Time	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes*	Yes*	No	No	Yes

# Bidirectional search

- ▶ Run two simultaneous states:
  - one forward from the initial state
  - one backward from the goal state
- ▶ Motivation:  $b^{(\frac{d}{2})} + b^{\frac{d}{2}}$  is much less than  $b^d$
- ▶ Implementation: Replace the goal check with a check to see whether the frontiers of the searches intersect



# Summary

- ▶ Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- ▶ There are a variety of uninformed search strategies available.
- ▶ Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

## Sources for the slides

- ▶ AIMA textbook (3<sup>rd</sup> edition)
- ▶ AIMA slides (<http://aima.cs.berkeley.edu/>)