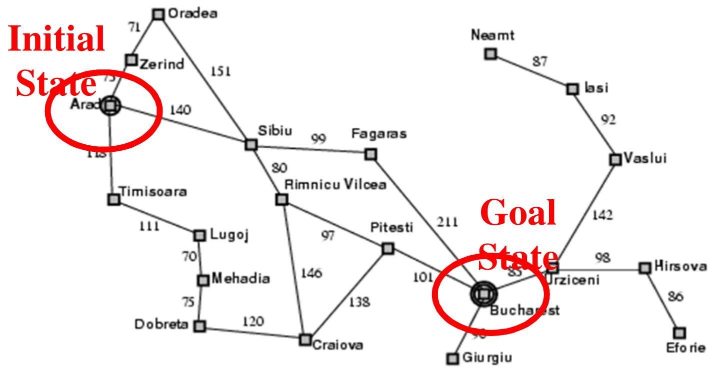
Problem Solving as Search

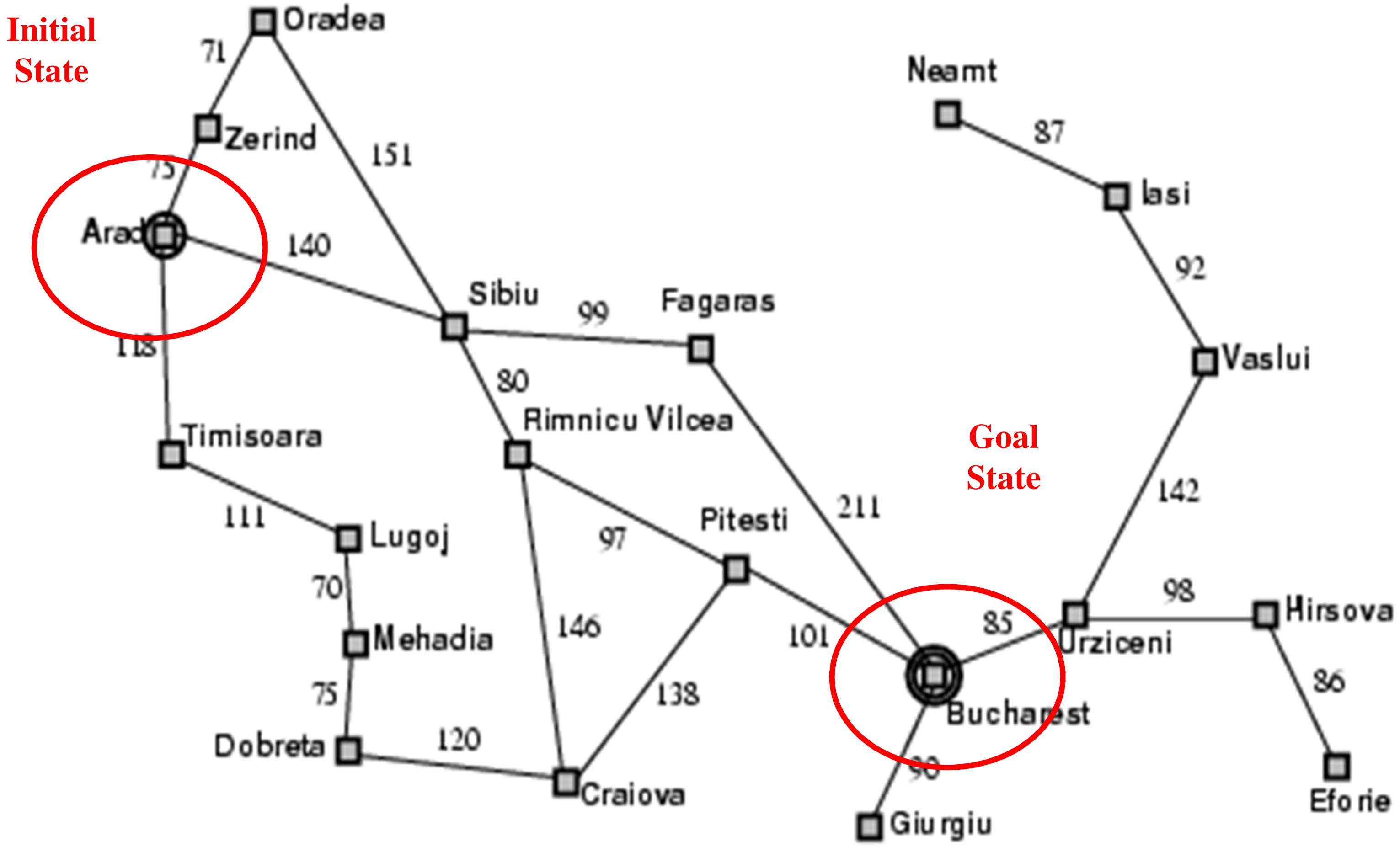
Problem-Solving

- Goal Formulation
  - set of one or more (desirable) world states
  - e.g. checkmate in chess
- Problem formulation
  - what actions and states to consider
  - given a goal and an initial state
- Search for solution
  - given the problem, search for a solution,
  - i.e., a sequence of actions to achieve the goal starting from the initial state
- Execution of the solution

Example: Path Finding problem



- Formulate goal:
  - be in Bucharest (Romania)
- Formulate problem:
  - initial state: be in Arad
  - action: drive between pair of connected cities (direct road)
  - state: be in a city (20 world states)
- Find solution:
  - sequence of cities leading from start to goal state, e.g., Arad, Sibiu, Fagaras, Bucharest
  - Execution
  - drive from Arad to Bucharest according to the solution


Initial
State
Oradea
71
Zerind
75
151
Arad
140
118
Sibiu
99
Fagaras
80
Rimnicu Vilcea
Timisoara
111
Lugoj
70
Mehadia
75
Dobreta
120
146
97
Pitesti
138
Craiova
211
101
Neamt
87
Iasi
92
Vaslui
Goal
State
142
98
85
Urziceni
Hirsova
86
Bucharest
90
Giurgiu
Eforie

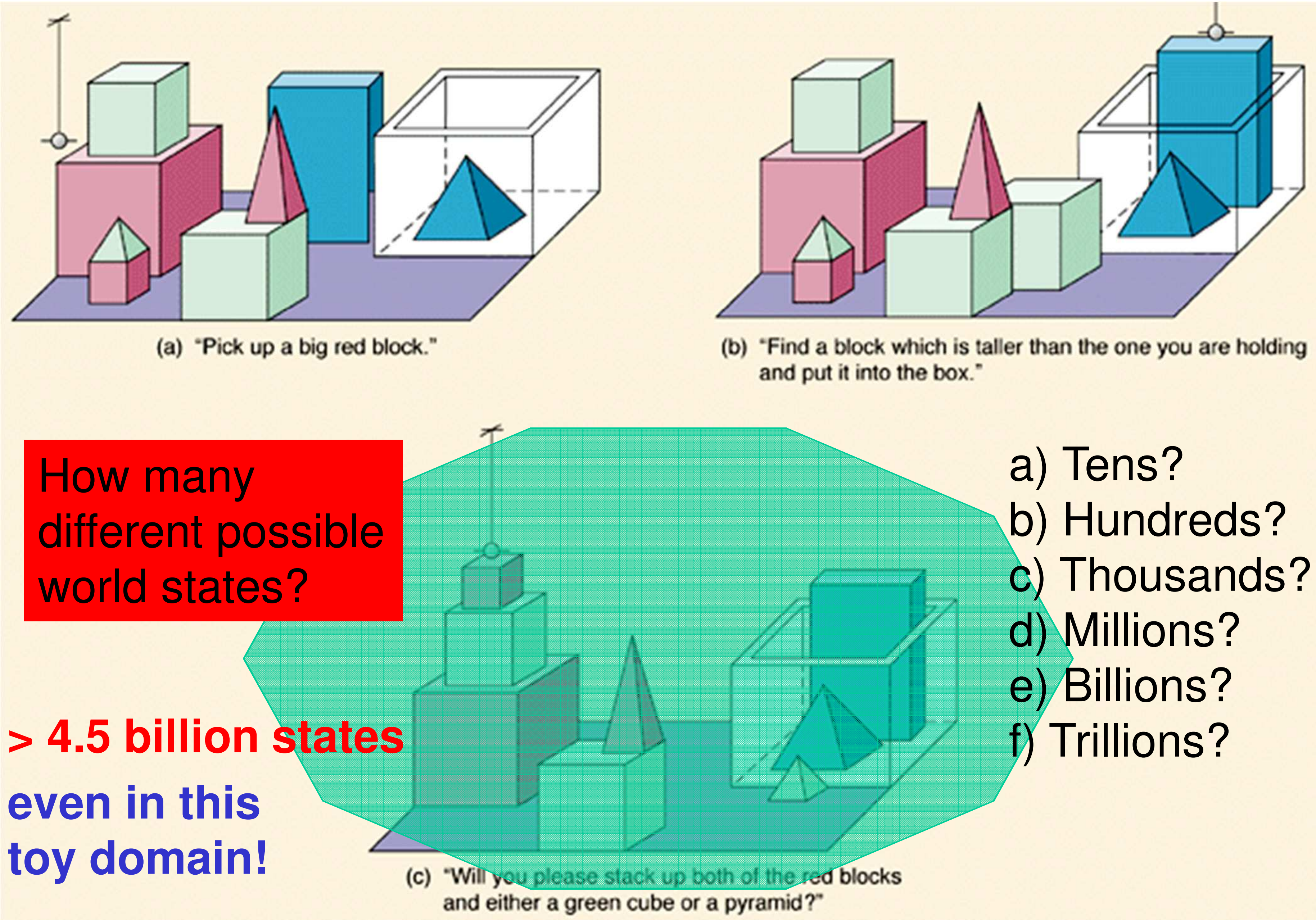
Problem-Solving

- hence conceptually very easy
- but: where‘s the catch?
  (or why haven‘t the robot‘s enslaved us yet?)

the curse of combinatorial explosion

(or our savior ☺)

Example: Blocks World



How many different possible world states?

a) Tens?
b) Hundreds?
c) Thousands?
d) Millions?
e) Billions?
f) Trillions?

> 4.5 billion states

even in this toy domain!

Common trick: state evaluation functions aka “heuristics”

- provide guidance w.r.t. what action to take next
- avoids exploring the whole search space
- e.g.,
  - consider all neighboring states, which are reachable via some action
  - select the action that leads to the state with the highest utility (evaluation value)
  - i.e., greedy approach to action selection

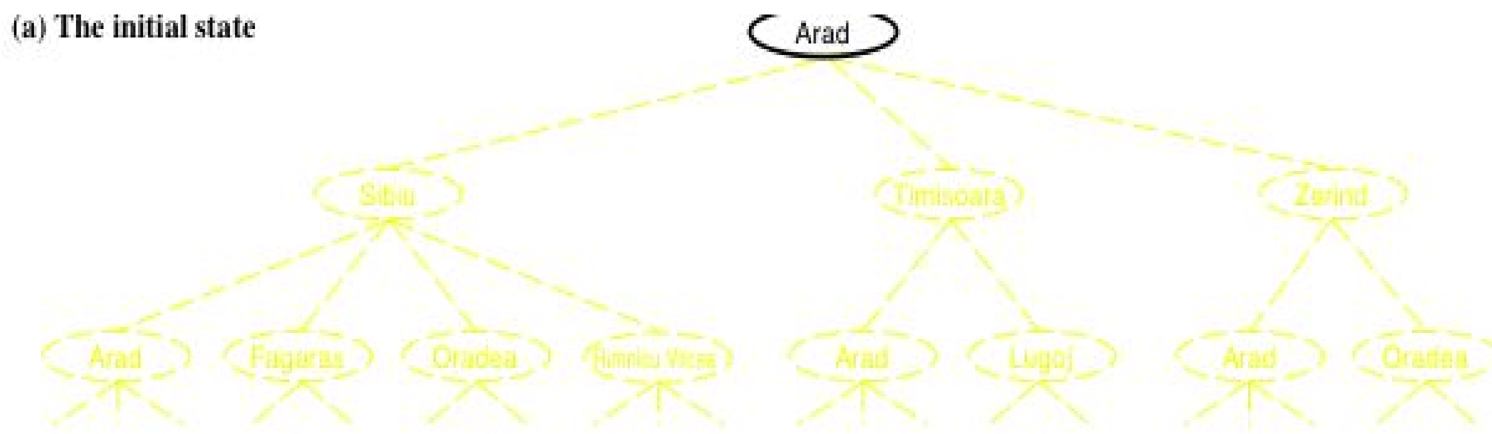
Search Methods

Basic search algorithms

Problem-Solving

- search the state space
- e.g., through *explicit tree generation*
  - ROOT= initial state
  - nodes and leafs generated through successor function

Simple tree search example



function TREE-SEARCH(*problem, strategy*) **return** a solution or failure

    Initialize search tree to the *initial state* of the *problem*

    do

        if no candidates for expansion **then return** *failure*

        choose leaf node for expansion according to *strategy*

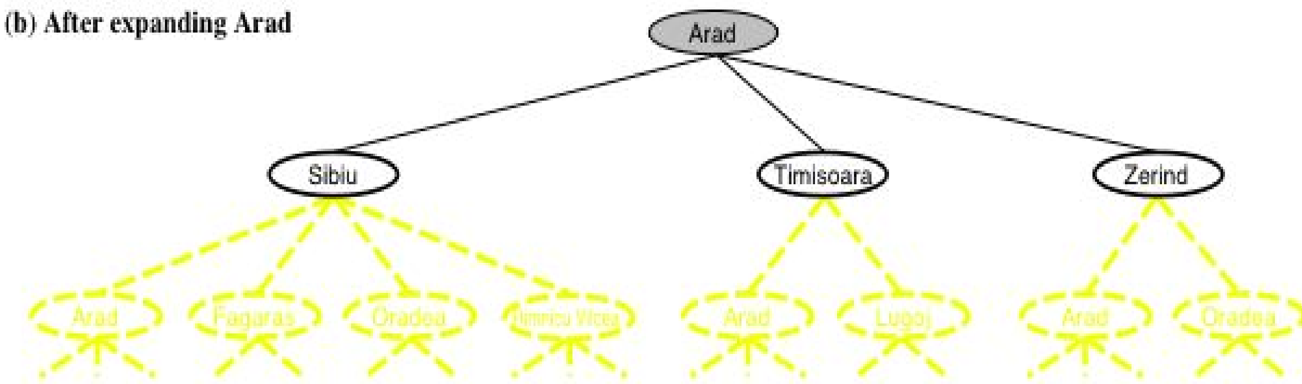
        if node contains goal state **then return** *solution*

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

    enddo

Simple tree search example

(b) After expanding Arad



function TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*

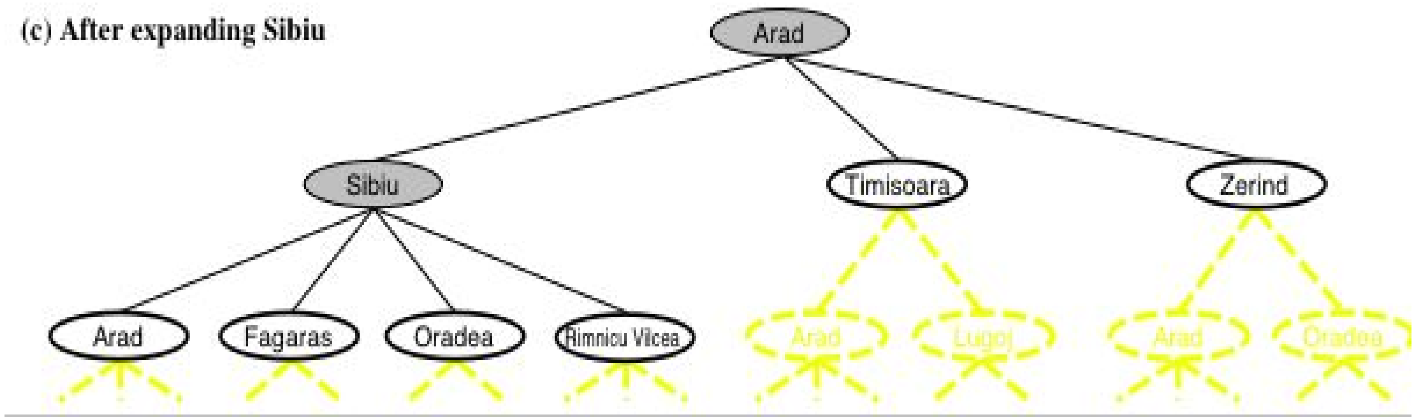
choose leaf node for expansion according to *strategy*

if node contains goal state **then return** *solution*

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

enddo

Simple tree search example



function TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*

choose leaf node for expansion according to *strategy*

if node contains goal state **then return** *solution*

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

enddo

Tree search algorithm

function TREE-SEARCH(*problem,fringe*) **return** a solution or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
    loop do
        if EMPTY?(*fringe*) **then return** failure
        node ← REMOVE-FIRST(*fringe*)
        if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
            then return SOLUTION(*node*)
        fringe ← INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

FRINGE *(aka* ***frontier****)*

= contains generated nodes which are not yet expanded

Tree search algorithm (2)

function EXPAND(*node,problem*) **return** a set of nodes

successors ← the empty set

for each <*action***,** *result***> in** SUCCESSOR-FN**[***problem***]**(STATE[*node*]) **do**

s ← a new NODE

STATE[*s*] ← *result*

PARENT-NODE[*s*] ← *node*

ACTION[*s*] ← *action*

PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(*node, action,s*)

DEPTH[*s*] ← DEPTH[*node*]+1

add *s* to *successors*

return *successors*

Search strategies

search strategy: picking the order of node expansion

- problem-solving performance:
  - **Completeness**: *always find a solution if one exists?*
  - **Optimality**: *always find the least-cost solution?*
  - **Time Complexity**: *#nodes generated/expanded?*
  - **Space Complexity**: *#nodes in memory during search?*
- measured in terms of problem difficulty:
  - *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

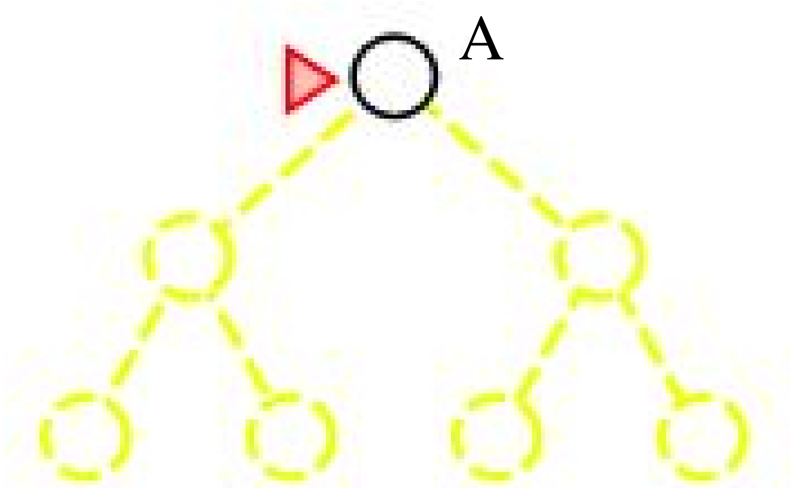
aka blind search

- use only information in problem definition
- example categories of expansion
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Iterative deepening search
  - Bidirectional search

(informed search: strategies can determine whether one state is better than another)

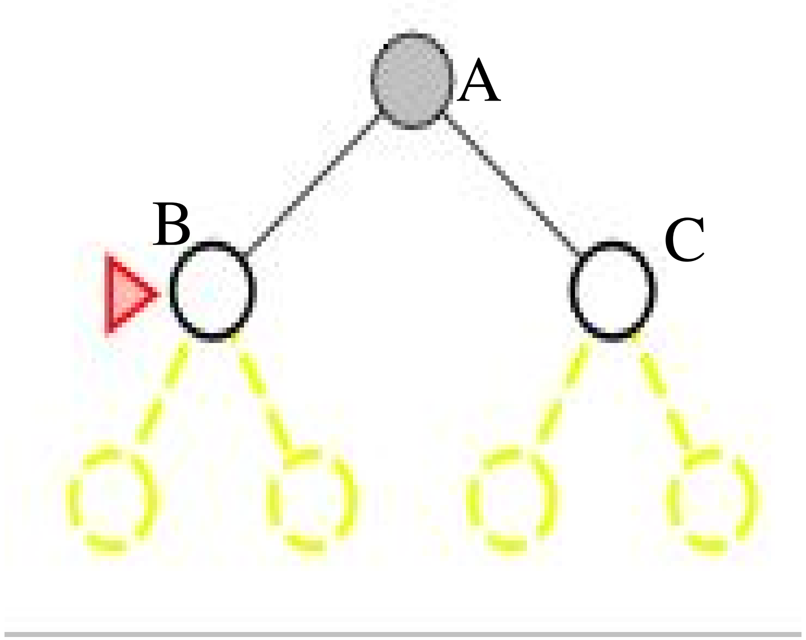
BFS

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a **FIFO** queue (First In, First Out)



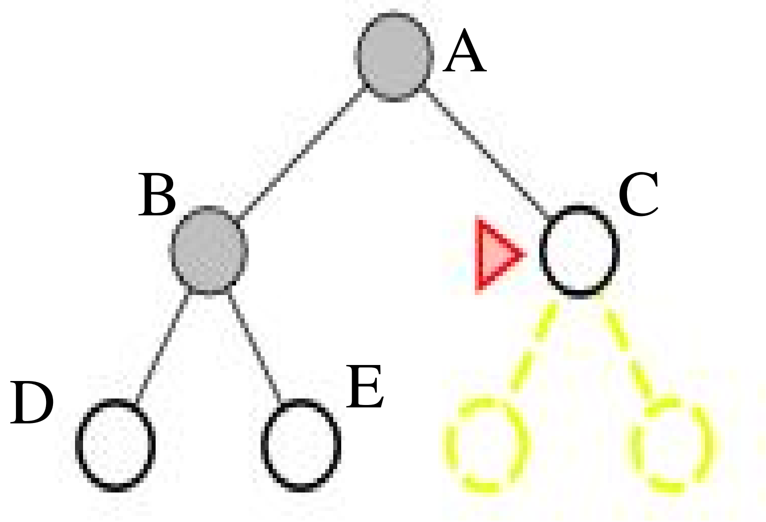
BFS

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



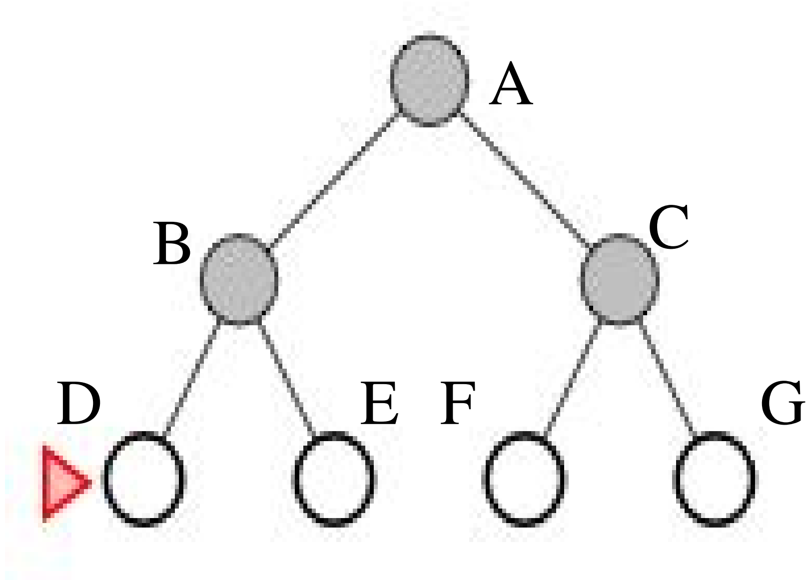
BFS

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



BFS

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



BFS evaluation

Completeness:

- Does it always find a solution if one exists?
- YES
  - If shallowest goal node is at some finite depth d
  - Condition: If *b* is finite, i.e., maximum number of successor nodes is finite

BFS evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - assume branching factor *b* (all nodes have b successors)
  - i.e., root has *b* successors, each node at the next level has again b successors (total b^2), …
  - solution is at depth *d*
  - worst case: expand all but the last node at depth *d*
  - total #nodes expanded:

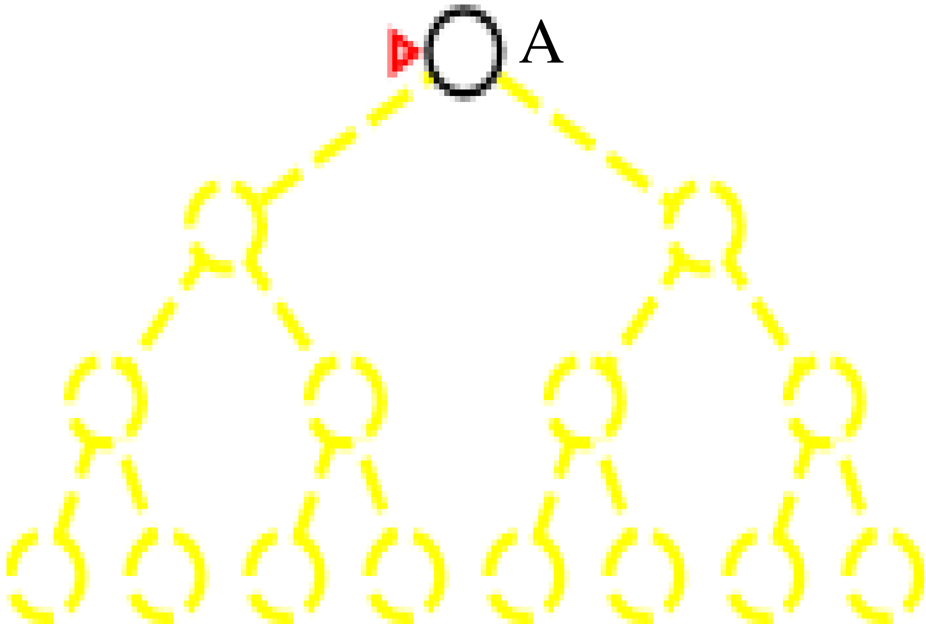
$$b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1})$$

BFS evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total numb. of nodes generated: $O(b^{d+1})$
- Space complexity:
  - like time if each node is retained in memory
- Optimality:
  - *Does it always find the least-cost solution?*
  - In general YES (unless actions have non-uniform cost, i.e., deeper states can be cheaper)

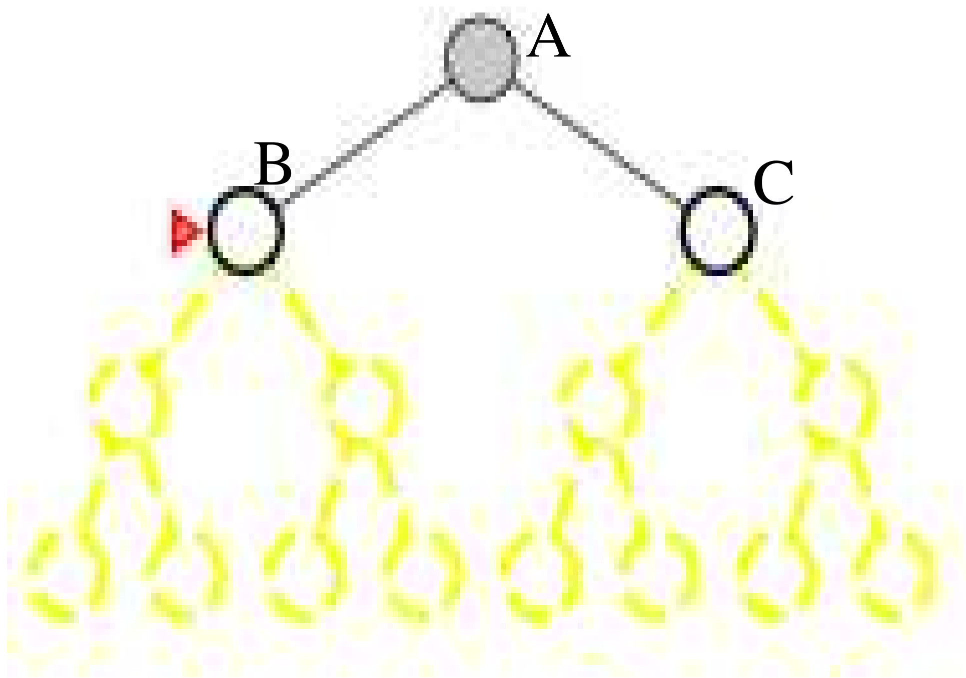
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (i.e., stack)



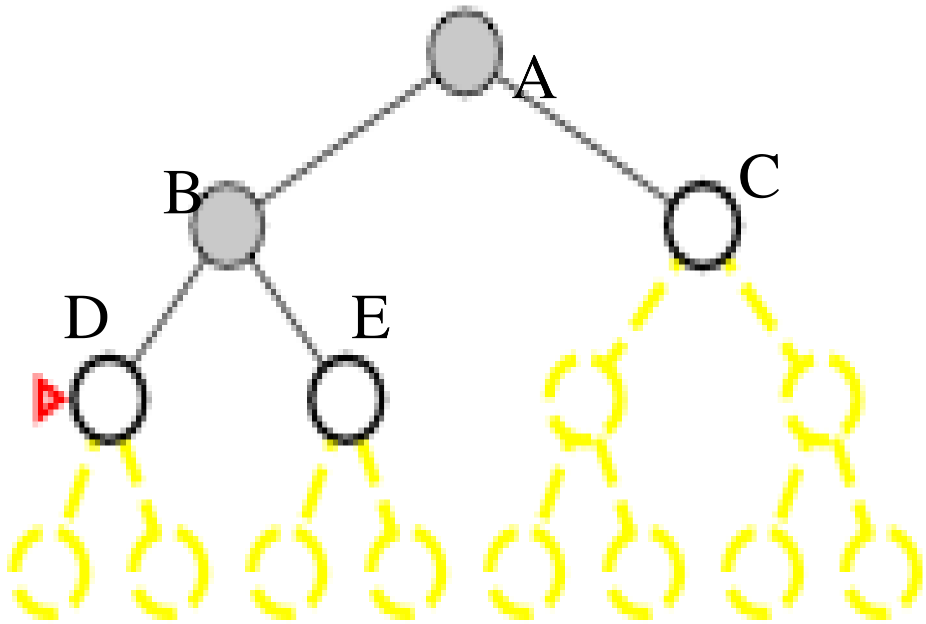
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



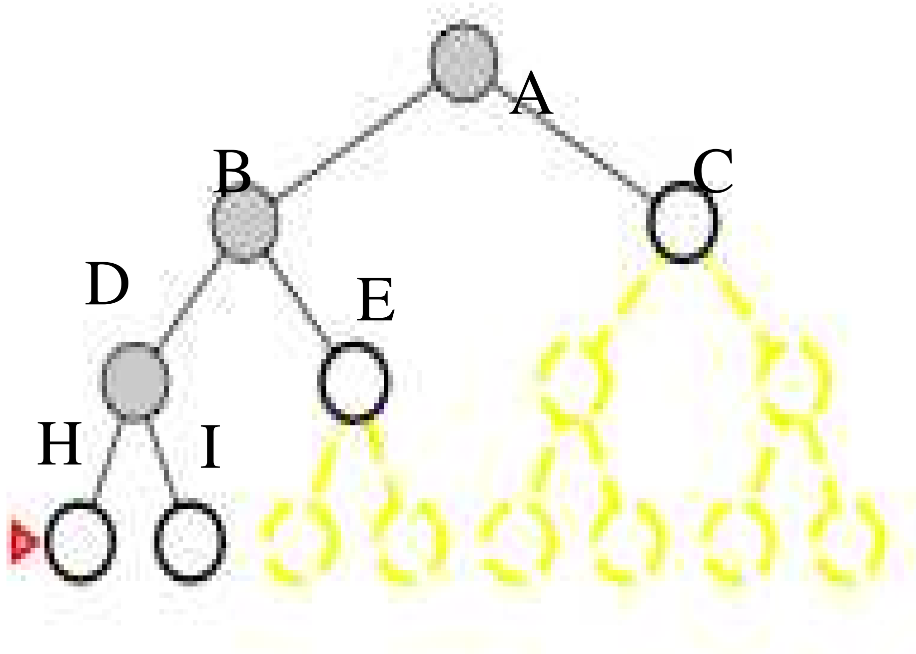
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



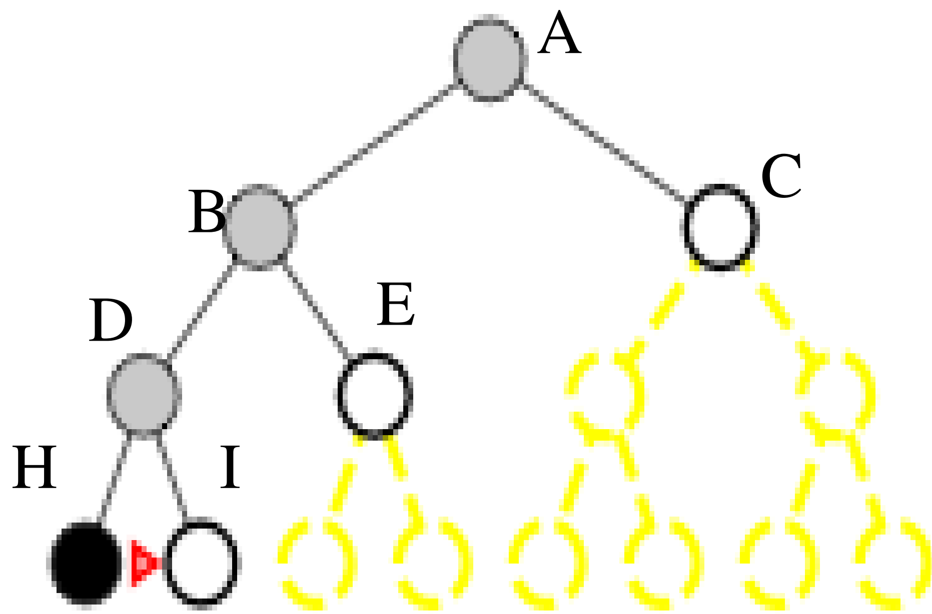
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



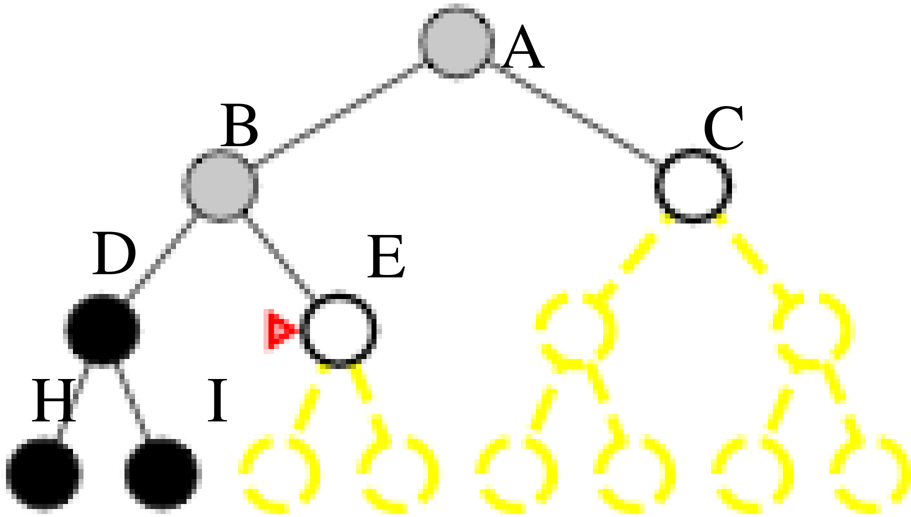
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



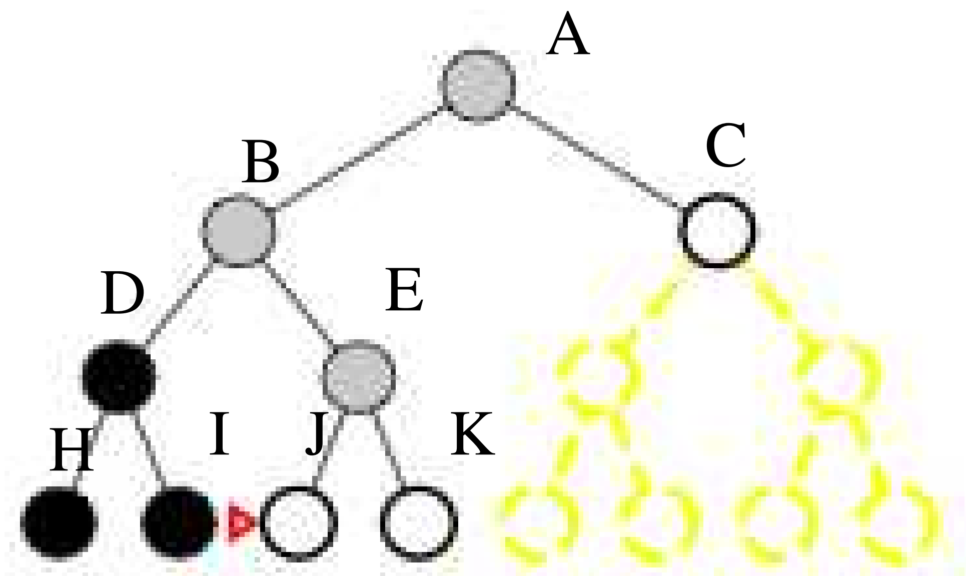
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



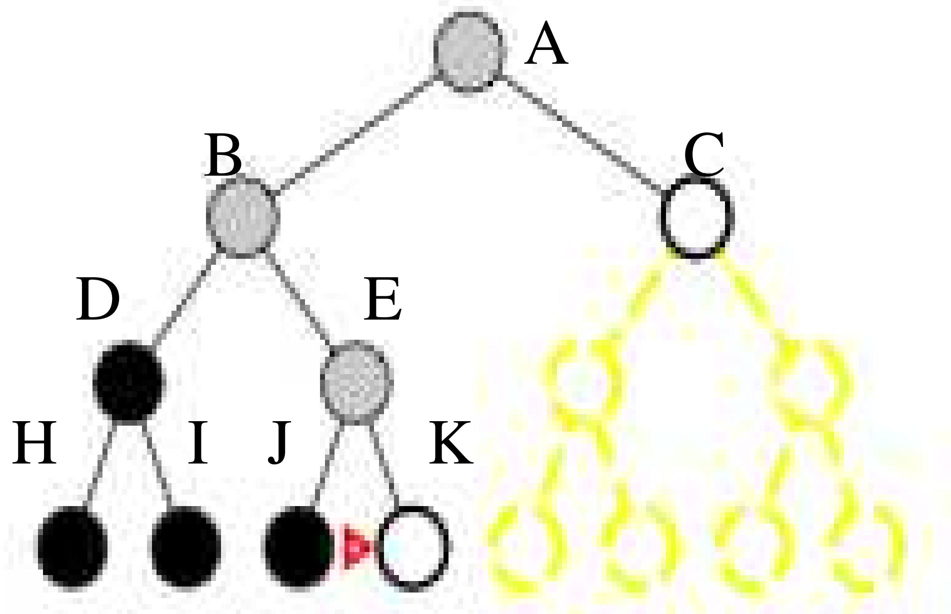
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



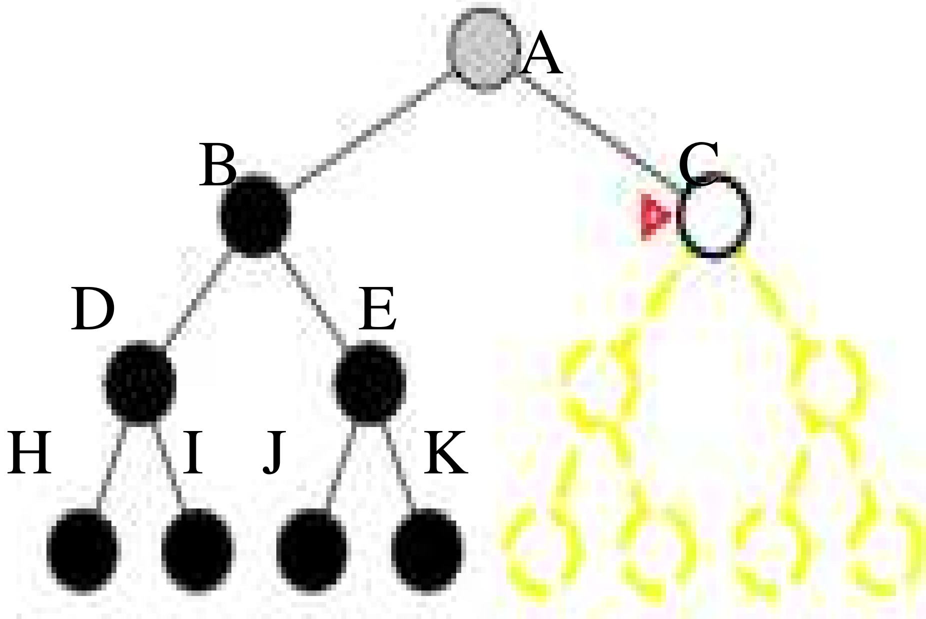
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



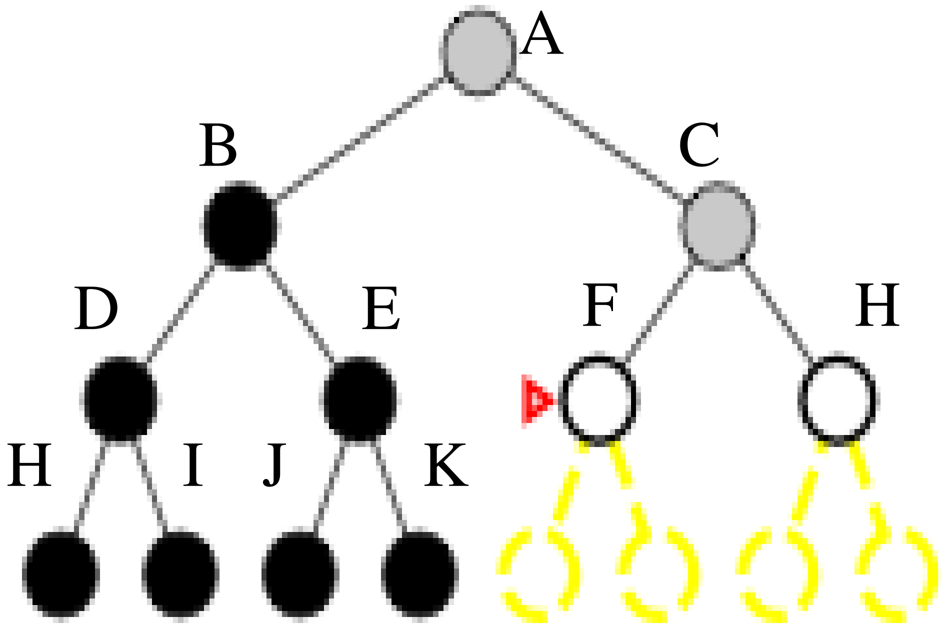
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



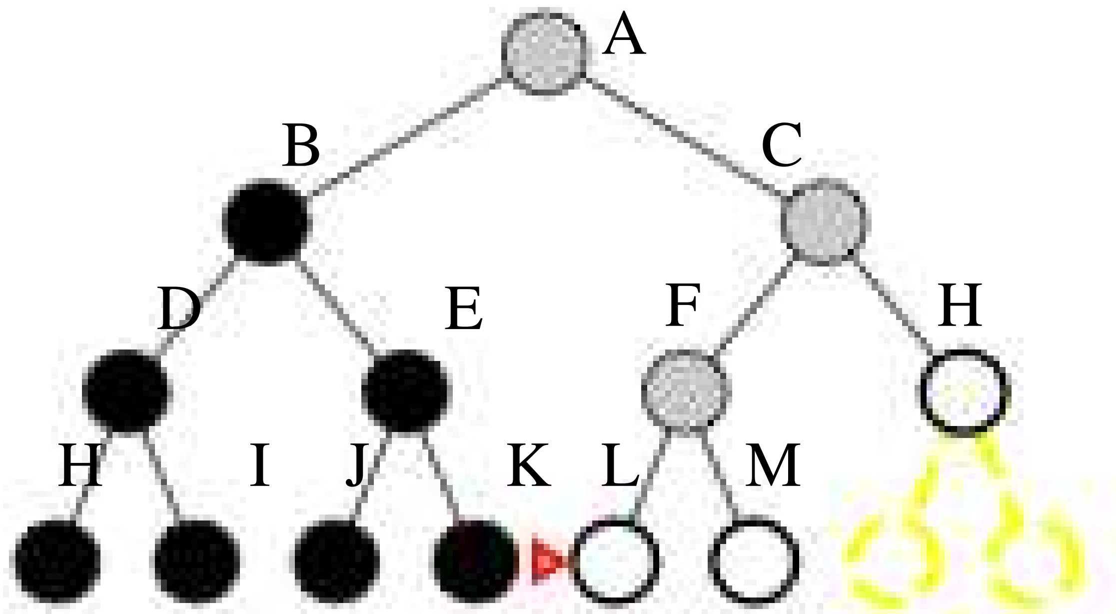
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



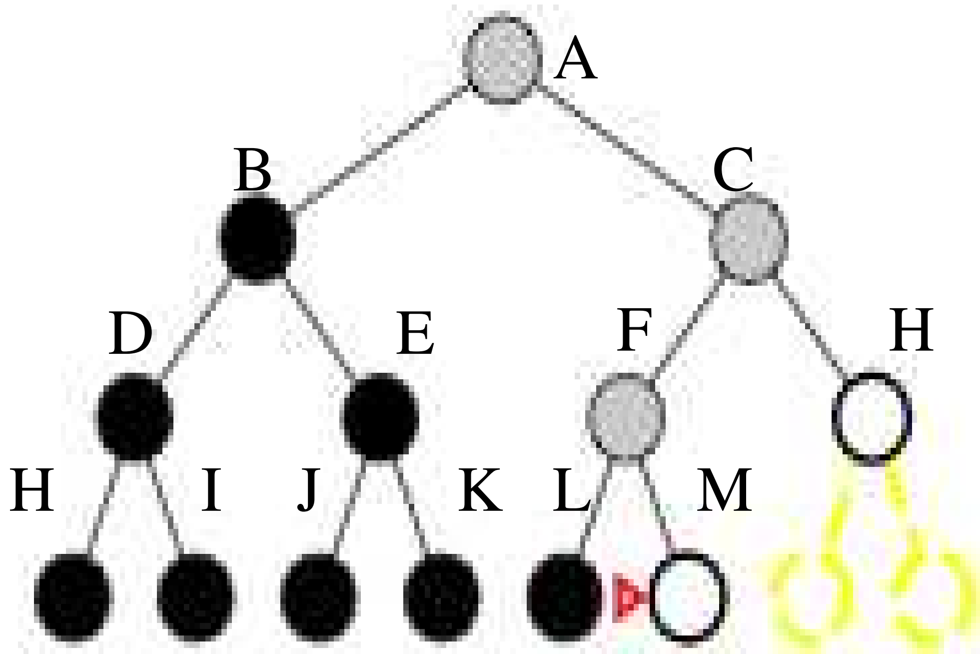
DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



DFS

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO stack



DFS evaluation

Branching factor – b

Maximum depth – m

- Completeness
  - NO unless search space is finite
  - may go infinitely into one branch without solution
- Time complexity $O(b^m)$
  - terrible if *m* is much larger than the depth *d* of the optimal solution
  - but **if there are many solutions**, then it is **typically faster than BFS**

DFS evaluation

- Completeness:
  - NO unless search space is finite
- Time complexity: $O(b^m)$
- Space complexity: $O(bm+1)$
  - backtracking search uses even less memory
  - one successor instead of all *b*
- Optimality:
  - NO unless search space is finite

Iterative deepening search

function ITERATIVE_DEEPENING_SEARCH(*problem*) **return** a solution or failure

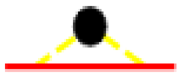
inputs: *problem*

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

- DFS with increasing limits
- combines benefits of DFS and BFS

ID-search, example

- Limit=0

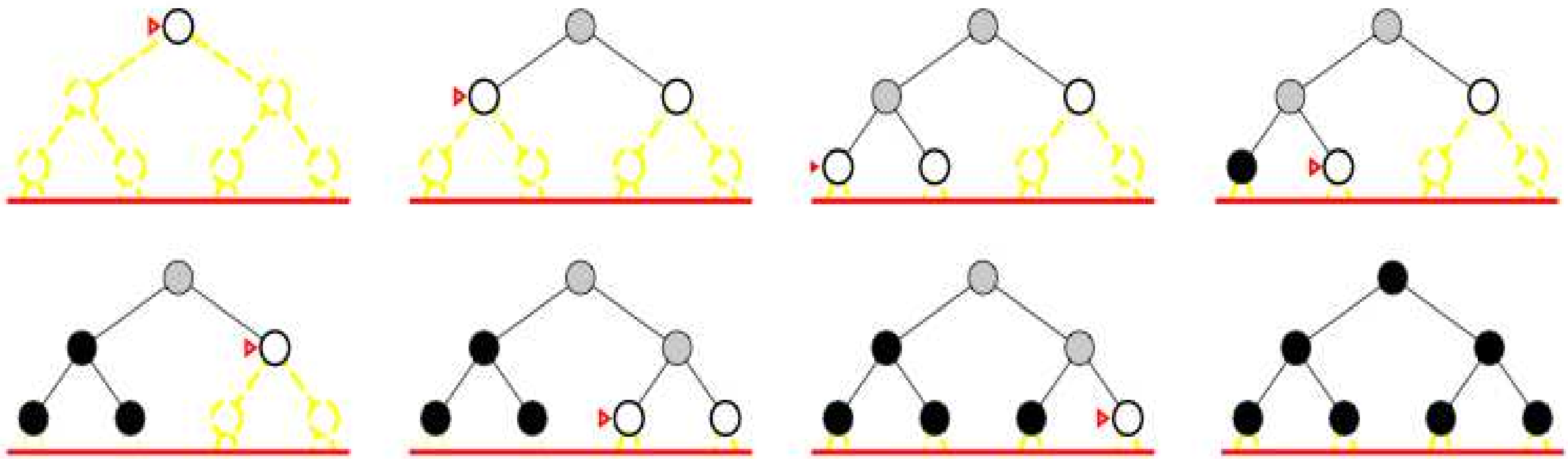


ID-search, example

- Limit=1

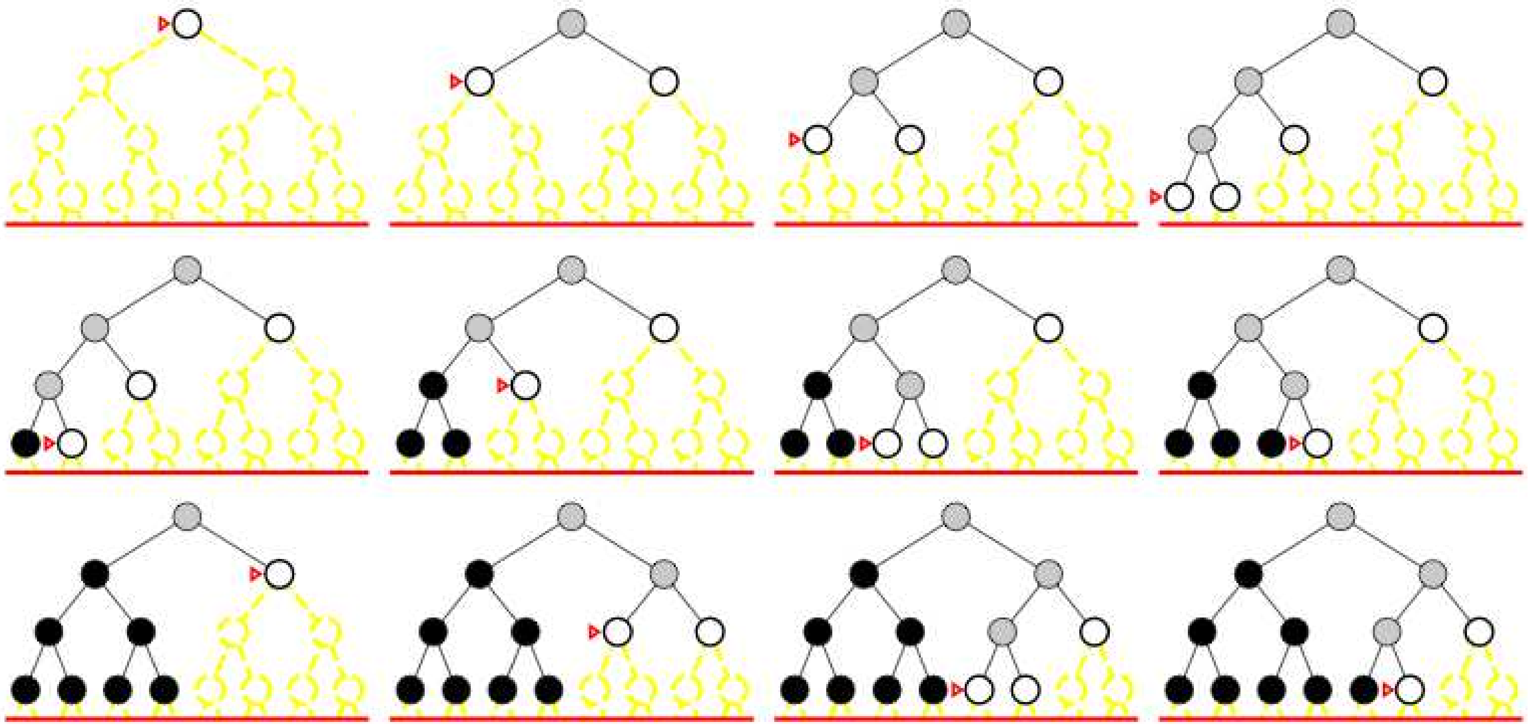
ID-search, example

- Limit=2



ID-search, example

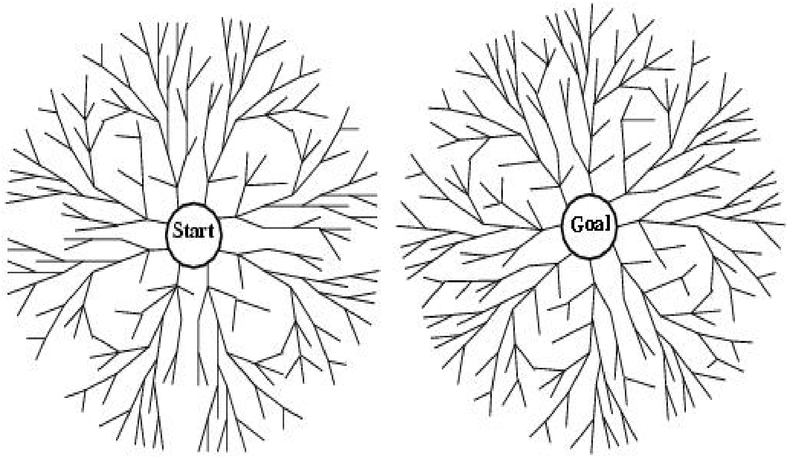
- Limit=3



IDS evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality:
  - YES (if step cost is uniform)

Bidirectional search



- Two simultaneous searches from start and goal
- Check if node belongs to other fringe before expansion

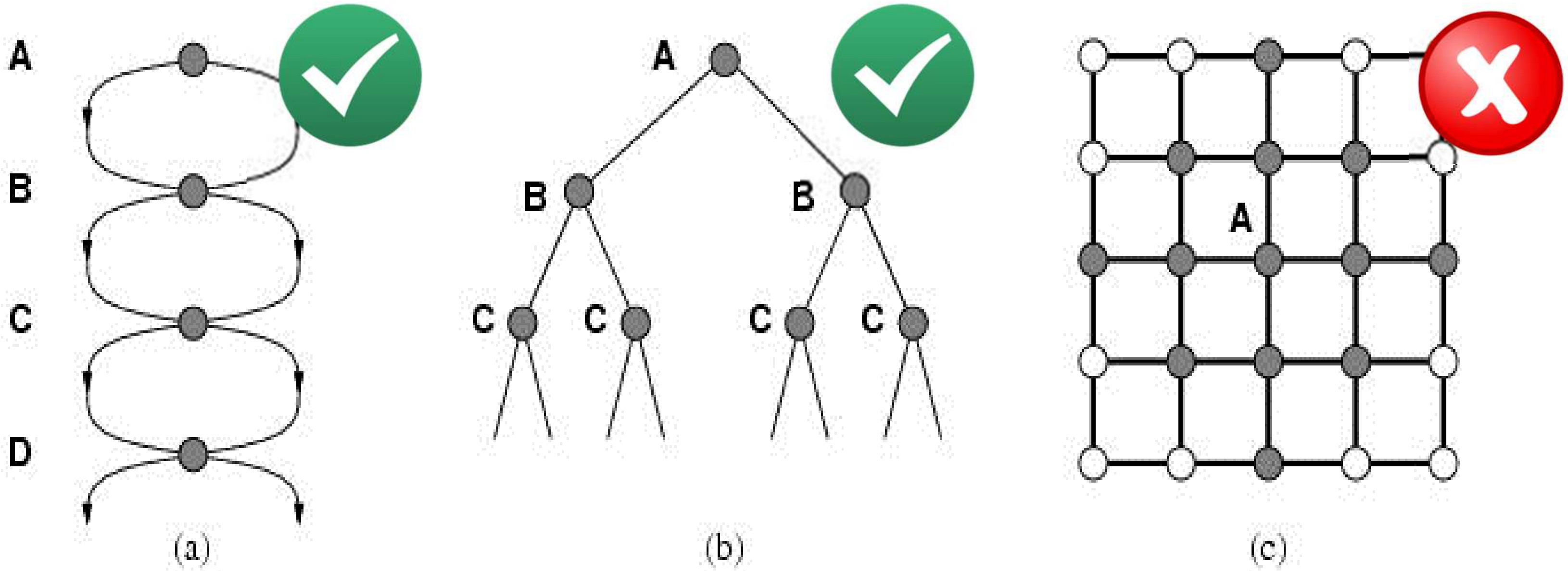
- Motivation: $b^{d/2} + b^{d/2} \neq b^d$
- Complete and optimal if both searches are BFS
- Predecessor of each node should be efficiently computable

Summary of algorithms

Criterion	Breadth-First	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES*	NO	YES, if $l \geq d$	YES	YES*
Time	b^{d+1}	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	NO	NO	YES	YES

Repeated states

Failure to detect repeated states can turn a solvable problems into unsolvable ones.



includes all problems with **reversible** transitions/actions!!!

Solution: Graph Search

- use an “explored” set
  to store already visited states/nodes
- expand only nodes from the frontier
  that are not in the explored set