CS 4700:
Foundations of Artificial Intelligence

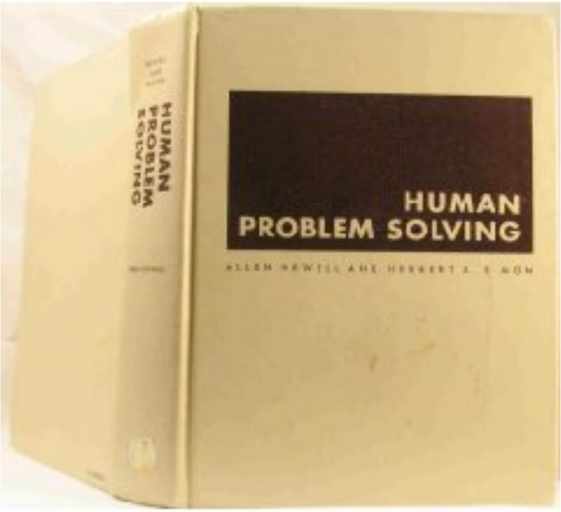
Bart Selman

Problem Solving by Search
R&N: Chapter 3

Introduction

"Search" is one of earliest areas studied in AI. Well-developed and understood.

Originated with Newell and Simon's work on problem solving; Human Problem Solving (1972).



Automated reasoning is a natural search task.

More recently: Given that almost all AI formalisms (planning, learning, etc) are NP-Complete or worse, some form of search (or optimization) is generally unavoidable (i.e., no smarter algorithm available).

Note: ***search and combinatorial optimization are closely related.***

Outline

Problem-solving agents

Problem types

Problem formulation

Example problems

Basic search algorithms (quick; most you already know!)

Problem-solving agents

More details on "states" soon.

Problem solving agents are goal-directed agents:

1. **Goal Formulation: Set of one or more (desirable) world states (e.g. "checkmate opponent in chess" or "reach vacation destination").**
2. **Problem formulation: What actions and states to consider given a goal and an initial state.**
3. **Search for solution: Given the problem, search for a solution --- *a sequence of actions to achieve the goal starting from the initial state*.**
4. **Execution of the solution**

Note: Formulation may feel somewhat "contrived," but is meant to model very general (human/computer) problem solving process.

Example: Path Finding problem

Formulate goal:

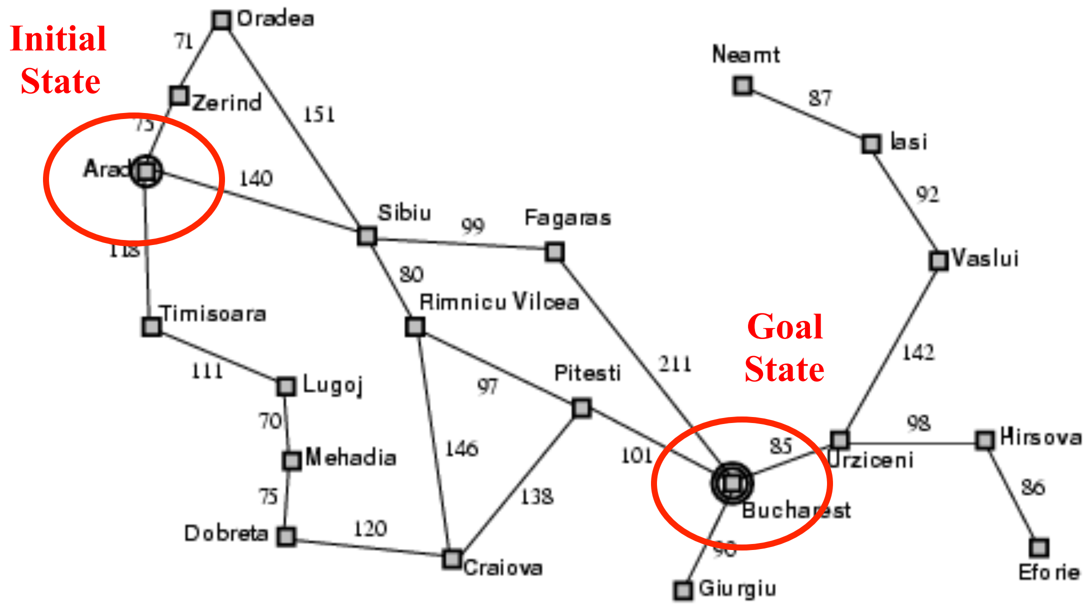
– be in Bucharest (Romania)

Formulate problem:

– **action**: drive between pair of connected cities (direct road)

– **state**: *traveler in a certain city*

(20 world states)



Find solution:

– sequence of cities leading from start to goal state, e.g., Arad, Sibiu, Fagaras, Bucharest

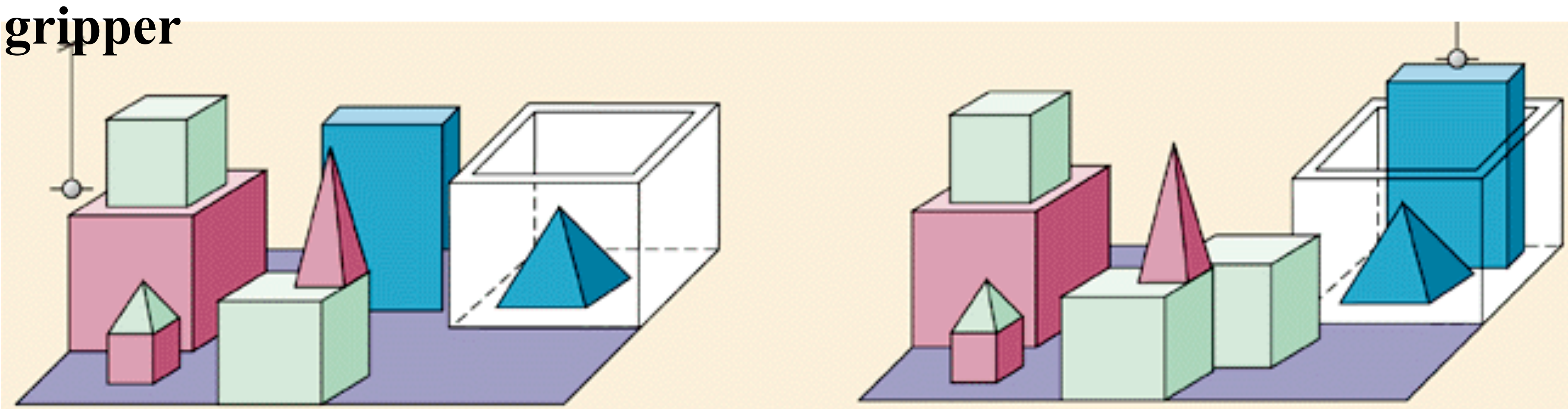
Execution

– drive from Arad to Bucharest according to the solution

Environment: fully observable (map), deterministic, and the agent knows effects of each action.

Note: Map is somewhat of a "toy" example. Our real interest: *Exponentially large spaces*, with e.g. 10^100 or more states. Far beyond full search. Humans can often still handle those! (We need to define a distance measure.) One of the mysteries of cognition.

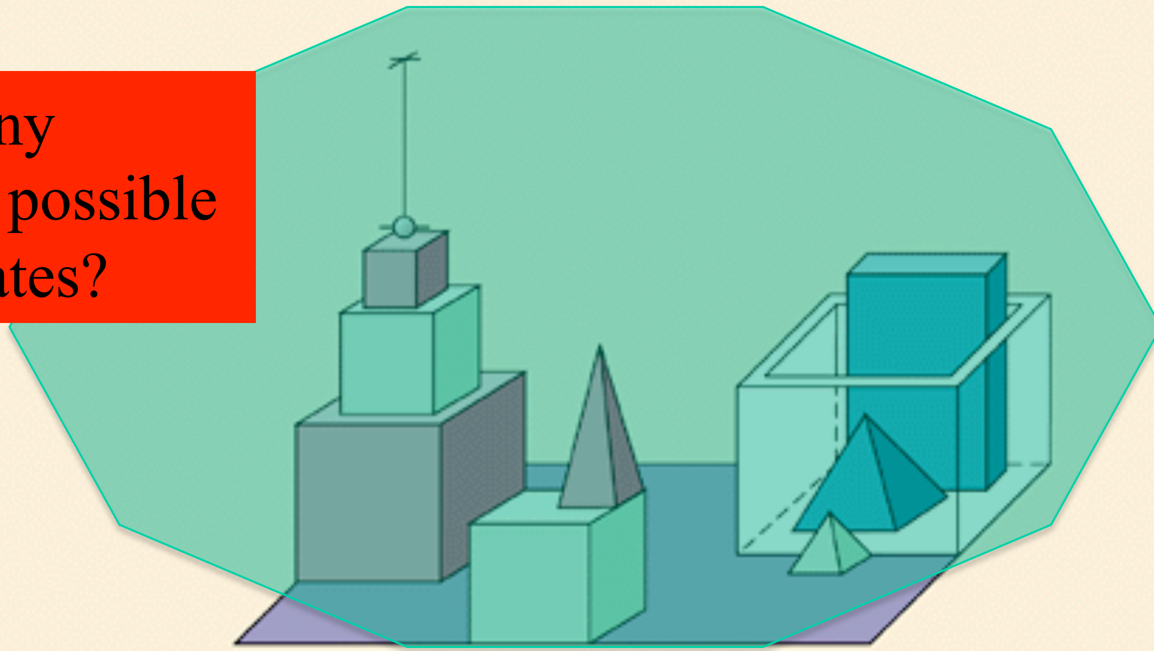
Micro-world: The Blocks World



(a) "Pick up a big red block."

(b) "Find a block which is taller than the one you are holding and put it into the box."

How many different possible world states?



(c) "Will you please stack up both of the red blocks and either a green cube or a pyramid?"

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

Size state space of blocks world example

n = 8 objects, k = 9 locations to build towers, one gripper. (One location in box.)
All objects distinguishable, order matter in towers. (Assume stackable in any order.)

Blocks: Use r-combinations approach from Rosen (section 5.5; CS-2800).

\- - - - - - - - - - - - - - - - **consider 16 = (n + k – 1) "spots"**

Select k – 1 = 8 "dividers" to create locations,

(16 choose 8) ways to do this, e.g.,

| |- - - |- | |- - - | |- | **Allocate n = 8 objs to remaining spots, 8! ways, e.g.,**

| |4 1 8 |5| |6 3 7| |2| **assigns 8 objects to the 9 locations**

a b c d e f g h i **based on dividers**

So, total number of states (ignoring gripper): (16 choose 8) * 8! = 518,918,400
*** 9 for location gripper: > 4.5 billion states even in this toy domain!**
Search spaces grow exponentially with domain. ***Still need to search them, e.g., to find a sequence of states (via gripper moves) leading to a desired goal state.***
How do we represent states? ***[predicates / features]***

Problem types

Increasing complexity

1) Deterministic, fully observable

Agent knows exactly which state it will be in; solution is a sequence of actions.

2) Non-observable --- sensorless problem

- **Agent may have no idea where it is (no sensors); it reasons in terms of belief states; solution is a sequence actions (effects of actions certain).**
- **Cars: drive by "dead reckoning" instead of GPS. Increasing uncertainty in location.**

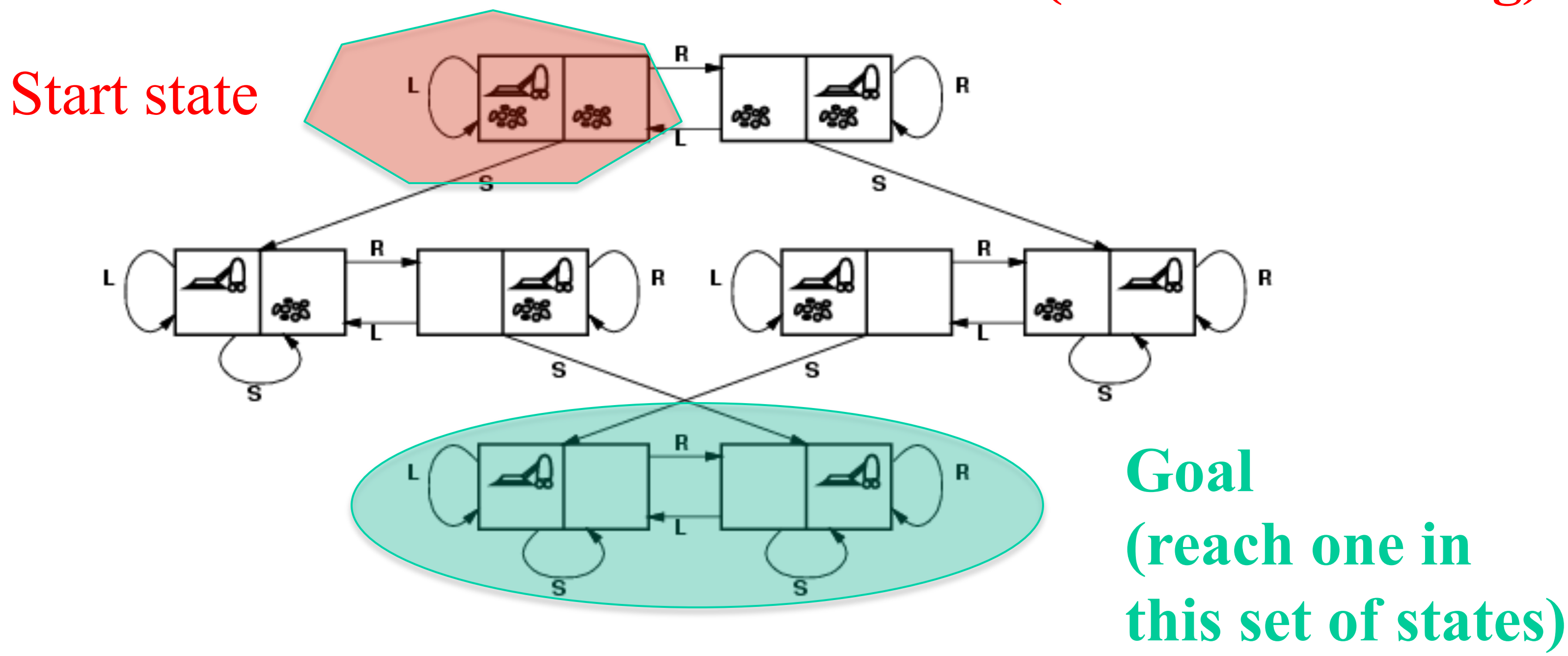
3) Nondeterministic and/or partially observable: contingency problem

- **Actions uncertain, percepts provide new information about current state (adversarial problem if uncertainty comes from other agents).**
- **Solution is a "*strategy*" to reach the goal.**

4) Unknown state space and uncertain action effects: exploration problem

-- Solution is a "strategy" to reach the goal (end explore environment).

Example: Vacuum world state space graph (Russell & Norvig)



states? The agent is in one of 8 possible world states.

actions? *Left*, *Right*, *Suck [simplified: left out No-op]*

goal test? No dirt at all locations (i.e., in one of bottom two states).

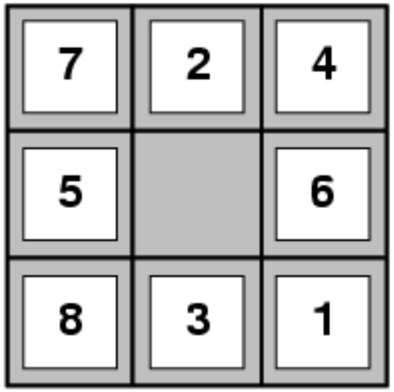
path cost? 1 per action: counts # of actions

Minimum path from Start to Goal state: 3 actions

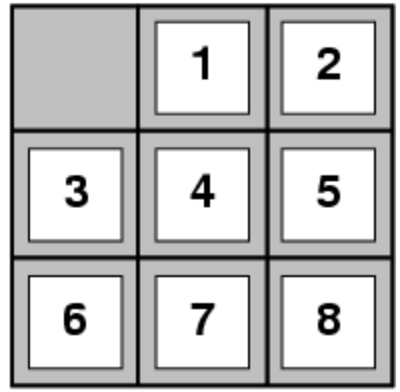
Alternative, longer plan: 4 actions

Note: path with thousands of steps before reaching goal also exists.

Example: The 8-puzzle
“sliding tile puzzle”



Start State



Goal State

Aside:
variations
on goal state.
eg empty square
bottom right or
in middle.

states? the boards, i.e., locations of tiles

actions? move blank left, right, up, down

goal test? goal state (given on right; tiles in order)

path cost? 1 per move

Note: finding optimal solution of *n*-puzzle family is NP-hard!
Also, from certain states you can’t reach the goal.
Total number of states 9! = 362,880 (more interesting space;
not all connected… only half can reach goal state)

15-puzzle

Goal state

Search space:
16!/2 = 1.0461395 e+13, about 10 trillion.
Too large to store in RAM (>= 100 TB). A challenge to search for a path from a given board to goal.

Korf (UCLA): Disk errors become a problem. (cosmic rays)

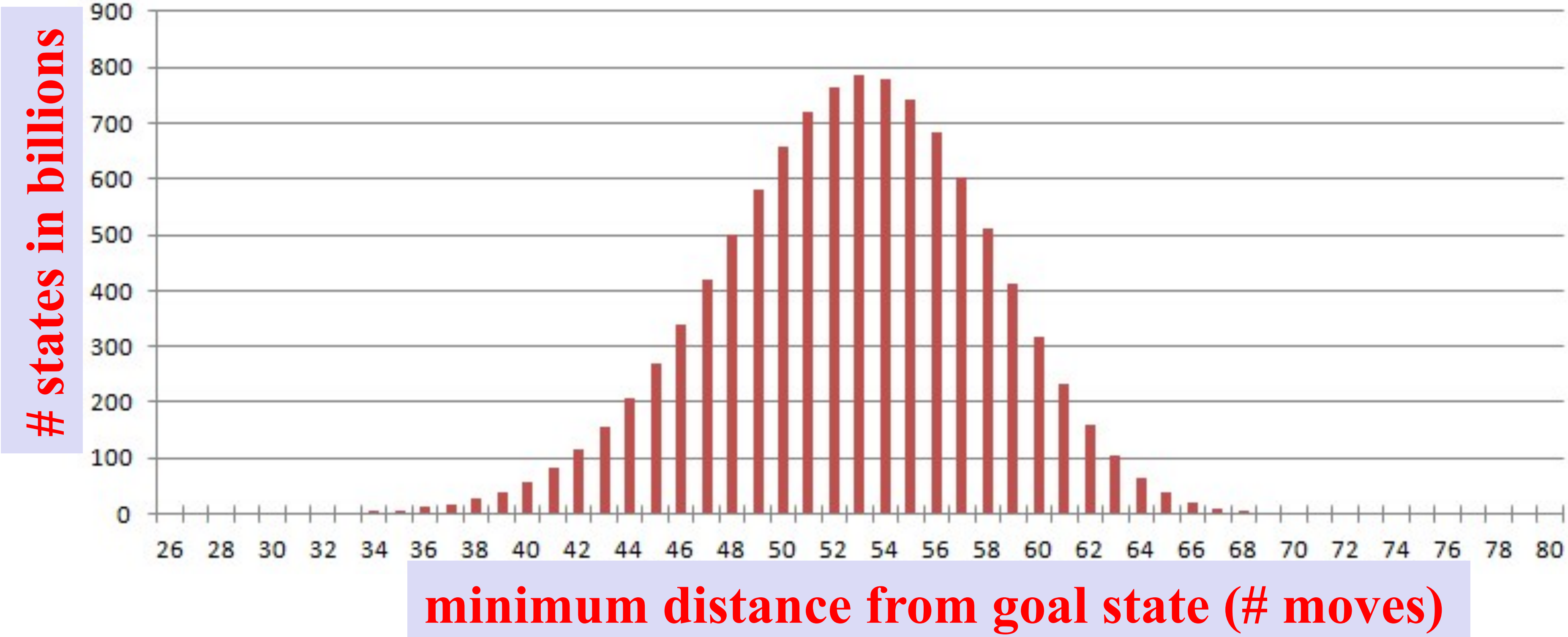
Longest minimum path: 80 moves. Just 17 boards, e.g,

Average minimum soln. length: 53.

People can find solns. But not necessarily minimum length. See solve it! (Gives strategy.)

Korf, R., and Schultze, P. 2005. Large-scale *parallel breadth-first search*. In Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05). See Fifteen Puzzle Optimal Solver. With effective search: opt. solutions in seconds! Average: milliseconds.

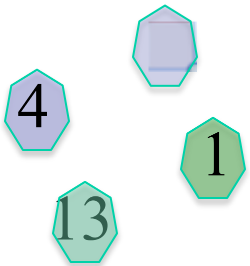
Where are the 10 trillion states?



dist.	# states
0	1
1	2
2	4
3	10
4	24
5	54

etc.

dist.	# states
76	272,198
77	26,638
78	3,406
79	70
80	17



17 boards farthest away from goal state (80 moves)

<2,5,6>

<15,12,11>/
<9,10,14>

What is it about these 17 boards out of over 10 trillion?

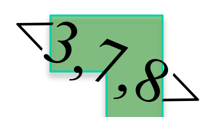
Each require 80 moves to reach:

Intriguing similarities. ***Each number has its own few locations.***

Interesting machine learning task:

Learn to recognize the hardest boards!

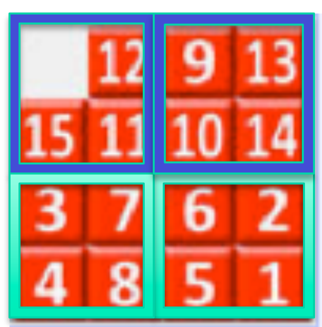
(Extremal Combinatorics, e.g. LeBras, Gomes, and Selman AAAI-12)



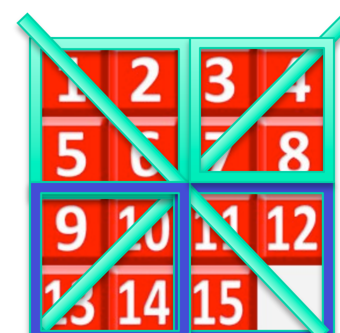
17 boards farthest away from goal state (80 moves)

There is one very special case: Most "regular" extreme case:

Each quadrant **reflected** along diagonal. "move tiles furthest away"



Goal state



Thanks to Jonathan GS

A few urls:

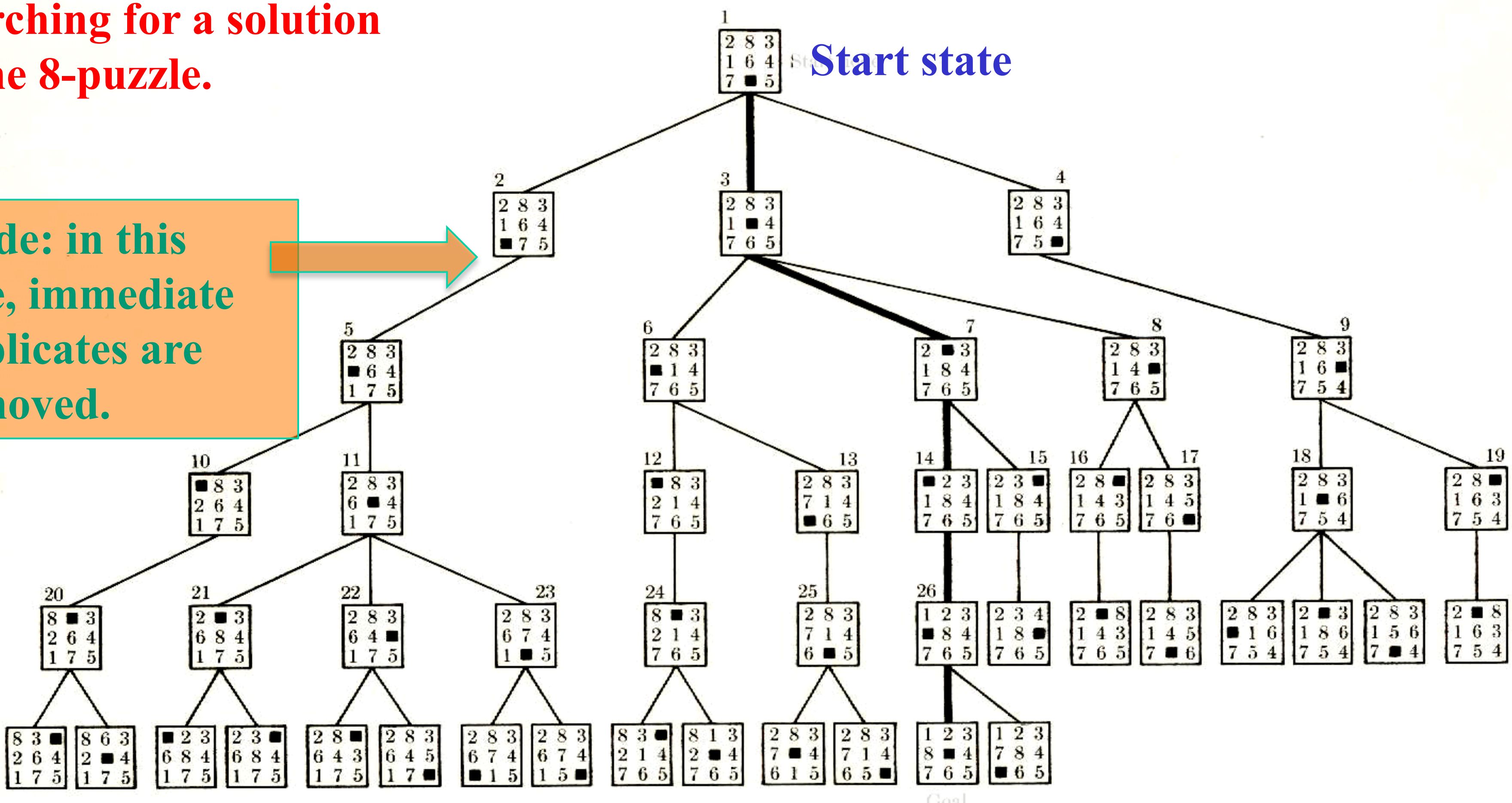
Play the eight puzzle on-line

Play the fifteen puzzle on-line

Let's consider the search for a solution.

Searching for a solution to the 8-puzzle.

Aside: in this tree, immediate duplicates are removed.



A breadth-first search tree

Branching factor 1, 2, or 3 (max). So, approx. 2 --- # nodes roughly doubles at each level. ***Number states of explored nodes grows exponentially with depth.***

For 15-puzzle, hard initial states: 80 levels deep, requires exploring approx. 2^80 ≈ 10^24 states.

If we block all duplicates, we get closer to 10 trillion (the number of distinct states: still a lot!).

Really only barely feasible on compute cluster with lots of memory and compute time. (Raw numbers for *24* puzzle: truly infeasible.)

Can we avoid generating all these boards? Do with much less search?
(Key: bring average branching factor down.)

Gedanken experiment: **Assume that you knew for each state, the minimum number of moves to the final goal state. (Table too big, but assume there is some formula/algorithm based on the board pattern that gives this number for each board and quickly.)**

Using the minimum distance information, is there a clever way to find a minimum length sequence of moves leading from the start state to the goal state? What is the algorithmic strategy?

d = min dist. to goal

Hmm. How do I know?

Note: at least one neighbor with d = 4.

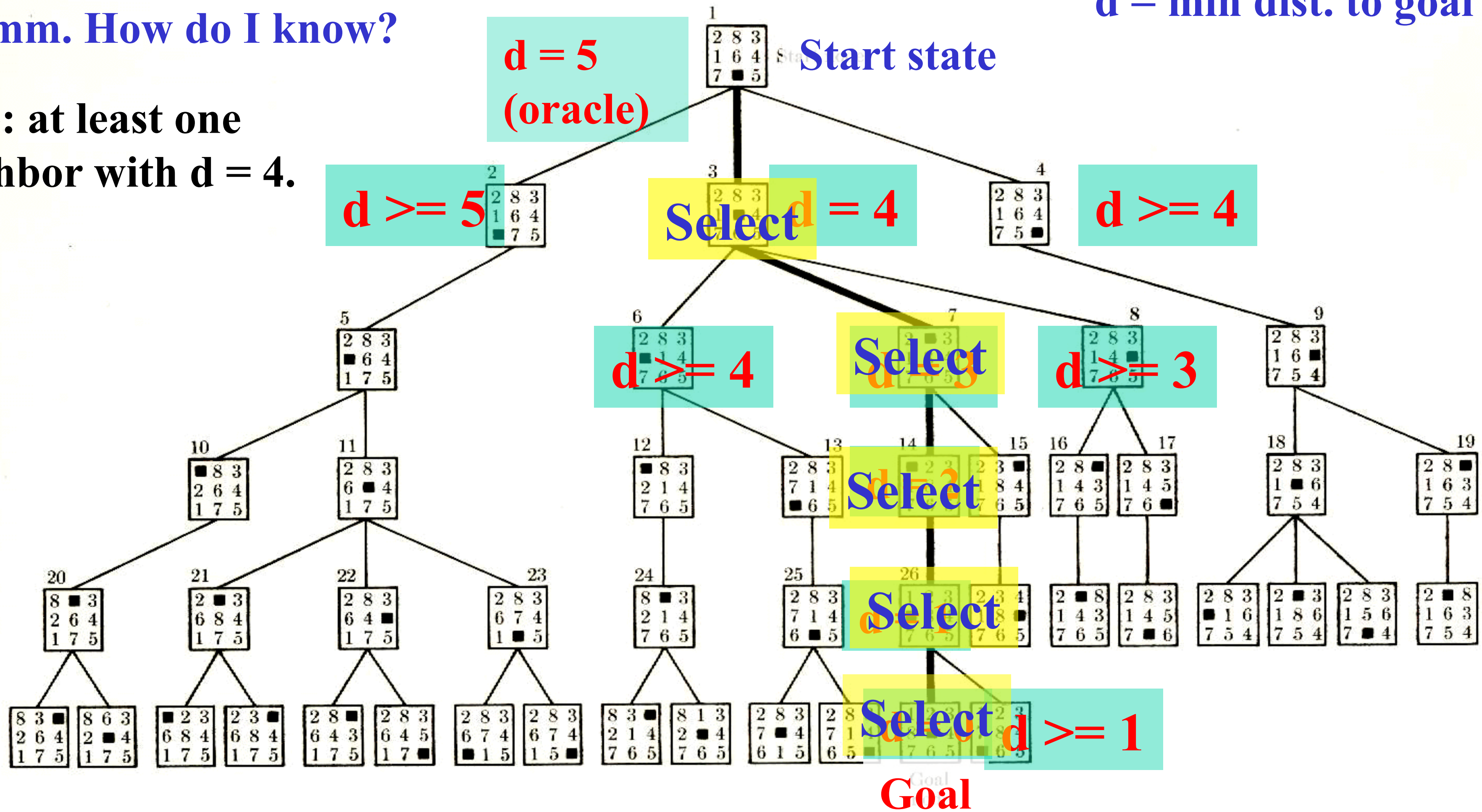


FIG. 2-2 The tree produced by a breadth-first search.

Search tree.

Branching factor approx. 2. So, with "distance oracle" we only need to explore approx. 2 * (min. solution length). (Why 2 times?)

For 15-puzzle, hard initial states: 80 levels deep, requires exploring approx. 2^80 ≈ 10^24 states.

But, with distance oracle, we would only need to explore roughly 80 * 2 = 160 states! (only linear in size of solution length)

We may not have the exact distance function ("perfect heuristics"), but we can still "guide" the search using an approximate distance function.

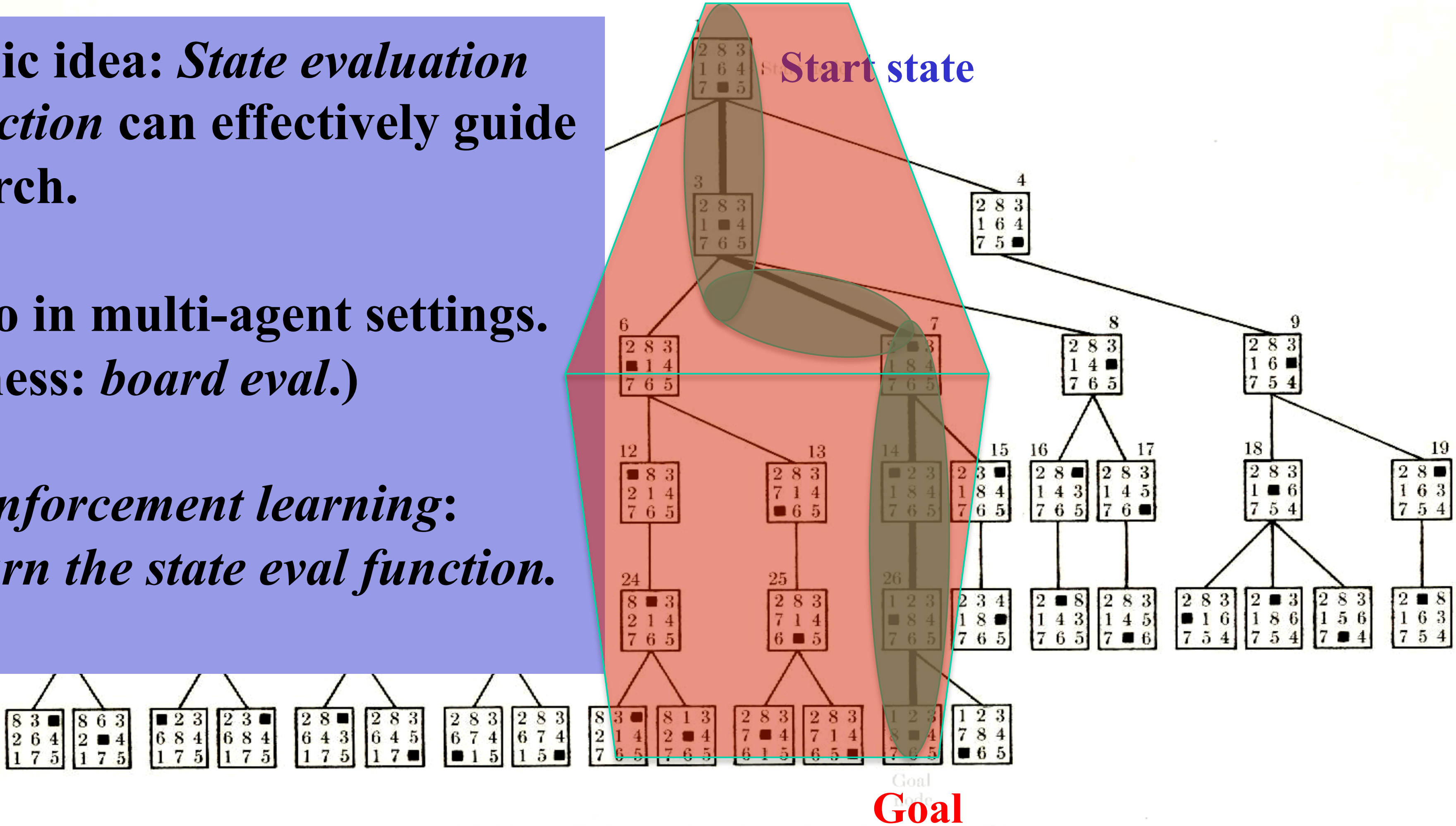
This is the key idea behind "heuristic search" or "knowledge-based search." We use knowledge / heuristic information *about the distance to the g*oal to guide our search process. *We can go from exponential to polynomial or even linear complexity. More common: brings exponent down significantly. E.g. from 2^L to 2^(L/100).*

The measure we considered would be the "perfect" heuristic. Eliminates tree search! Find the right "path" to goal state immediately.

Basic idea: *State evaluation function* can effectively guide search.

Also in multi-agent settings. (Chess: *board eval*.)

Reinforcement learning*: *Learn the state eval function.



Search tree.

Perfect "heuristics," eliminates search.

Approximate heuristics, significantly reduces search.

Best (provably) use of search heuristic info: A* search (soon).

General question: Given a state space,
how good a heuristics can we find?

State evaluation functions or "heuristics"

Provide guidance in terms of what action to take next.

General principle: Consider all neighboring states, reachable via some action. Then select the action that leads to the state with the highest utility (evaluation value). This is a fully greedy approach.

Aside: "Highest utility" was "shortest distance to the goal" in previous example.

Because eval function is often only an estimate of the true state value, greedy search may not find the optimum path to the goal.

By *adding some search* with certain guarantees on the approximation, we can still get optimal behavior (A* search) (i.e. finding the optimal path to the solution). Overall result: generally exponentially less search required.

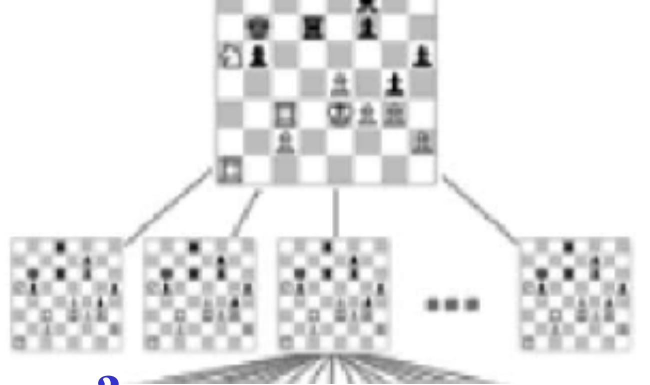
N-puzzle heuristics (“State evaluation function” wrt the goal to be reached):

1) **Manhattan Distance: For each tile the number of grid units between its current location and its goal location are counted and the values for all tiles are summed up. (underestimate; too “loose”; not very powerful)**

2) **Felner, Ariel, Korf, Richard E., Hanan, Sarit, Additive Pattern Database Heuristics, Journal of Artificial Intelligence Research 22 (2004) 279-318. The 78 Pattern Database heuristic takes a lot of memory but solves a random instance of the 15-puzzle within a few milliseconds on average. Finding an optimal solution (80 moves cases) takes a few seconds each. So, thousands of nodes considered instead of many billions.**

Note: many approx. heuristics (“conservative” / underestimates to goal) combined with search can still find optimal solutions.

State evaluation function (or utility value) is a very general and useful idea.



Example:

- In chess, given a board, what would be the perfect evaluation value that you would want to know? (Assume the perspective of White player.)

A: f(board) → {+1, 0, -1}, with +1 for guaranteed win for White,
0 draw under perfect play, and
-1 loss under perfect play.

Perfect play: all powerful opponent.

Given f, how would you play then?

In practice, we only know (so far) of an approximation of f.

f(board) → [-1,+1]   (interval from -1 to +1)

based on "values" of chess pieces, e.g., pawn 1 point, rook 5 points.

Informally, board value gives "probability (?) of winning."

State evaluation function (or utility value) is a very general and useful idea.

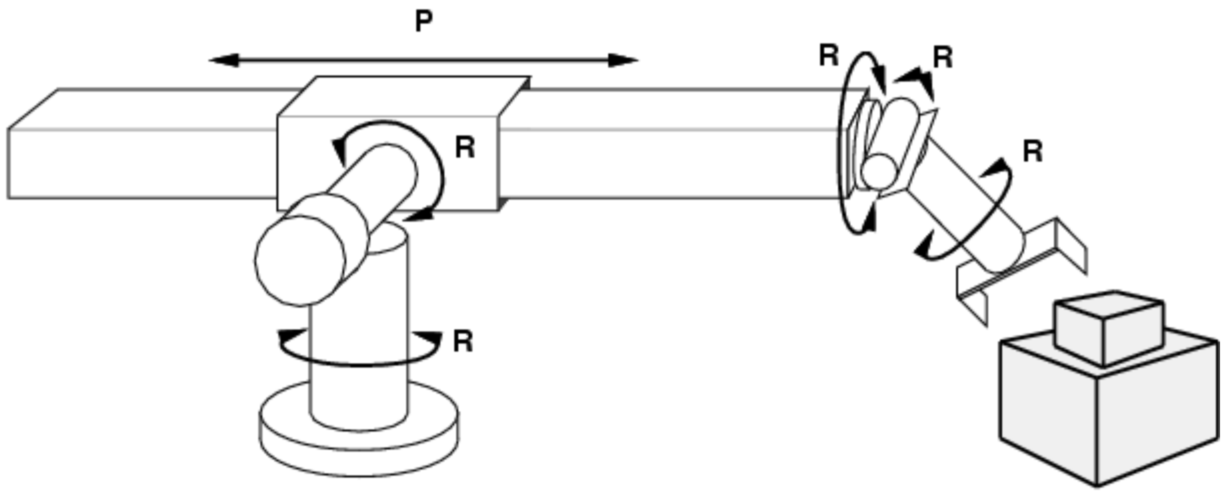
Examples:

- **TD-Gammon backgammon player. Neural net was trained to find approximately optimal state (board) evaluation values (range [-1,+1]). (Tesauro 1995)**

- **"Robocopter" --- automated helicopter control; trained state evaluation function. State given by features, such as, position, orientation, speed, and rotors position and speed. Possible actions: change rotors speed and angle. Evaluation: assigns value in [-1,+1] to capture stability.**

(Abbeel, Coates, and Ng 2008)

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

Other example search tasks

VLSI layout: positioning millions of components and connections on a chip to minimize area, circuit delays, etc.

Robot navigation / planning

Automatic assembly of complex objects

Protein design: sequence of amino acids that will fold into the 3-dimensional protein with the right properties.

Literally thousands of combinatorial search / reasoning / parsing / matching problems can be formulated as search problems in exponential size state spaces.

Any type of task where the solution is "hiding" in an exponential / combinatorial space of possibilities.

Key aspect of intelligence: Our ability to deal with such spaces.

Search Techniques

Searching for a (shortest / least cost) path to goal state(s).

Search through the state space.

We will consider search techniques that use an explicit search tree that is generated by the initial state + successor function.

initialize (initial node)
Loop
    choose a node for expansion
        according to a strategy
    goal node? → done
    expand node with successor function

Tree-search algorithms

Basic idea:

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

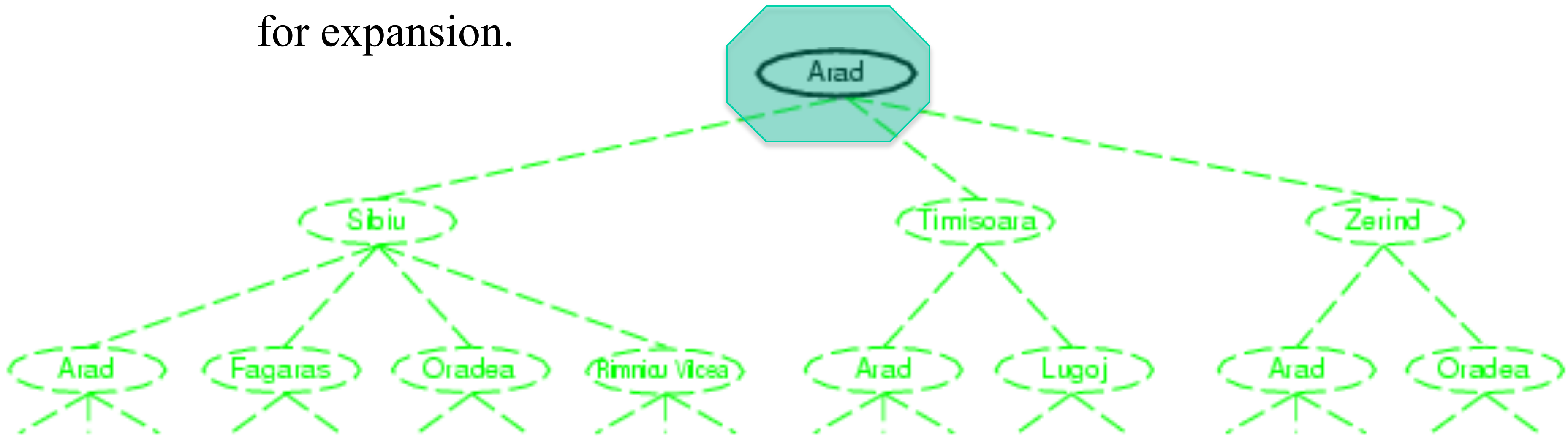
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
```

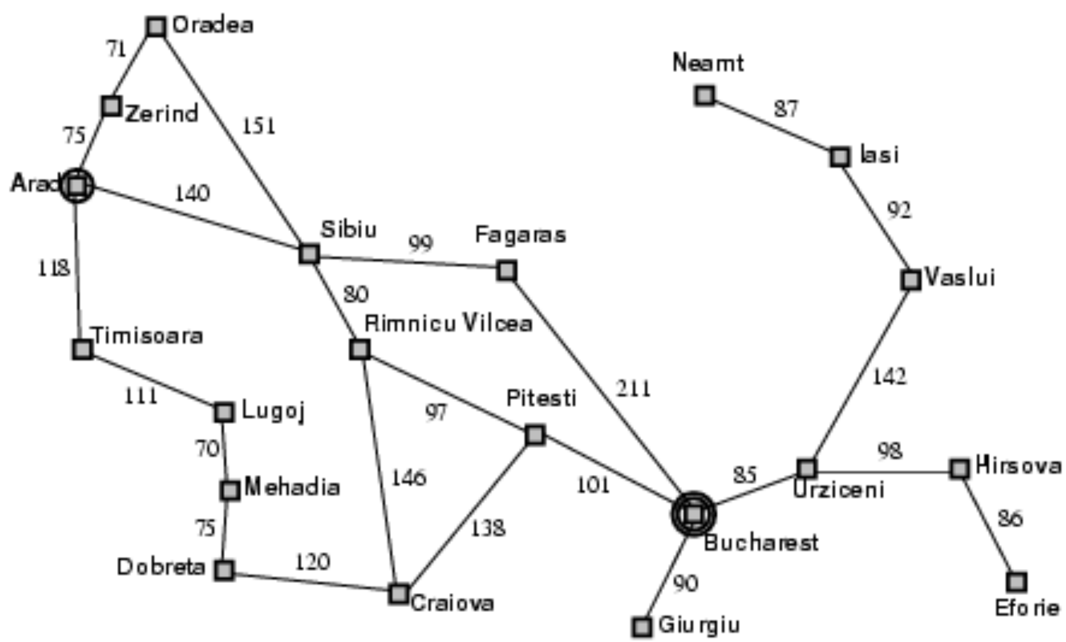
Fig. 3.7 R&N, p. 77

Note: 1) Here we only check a node for possibly being a goal state, *after* we select the node for expansion.
2) A "node" is a data structure containing state + additional info (parent node, etc.

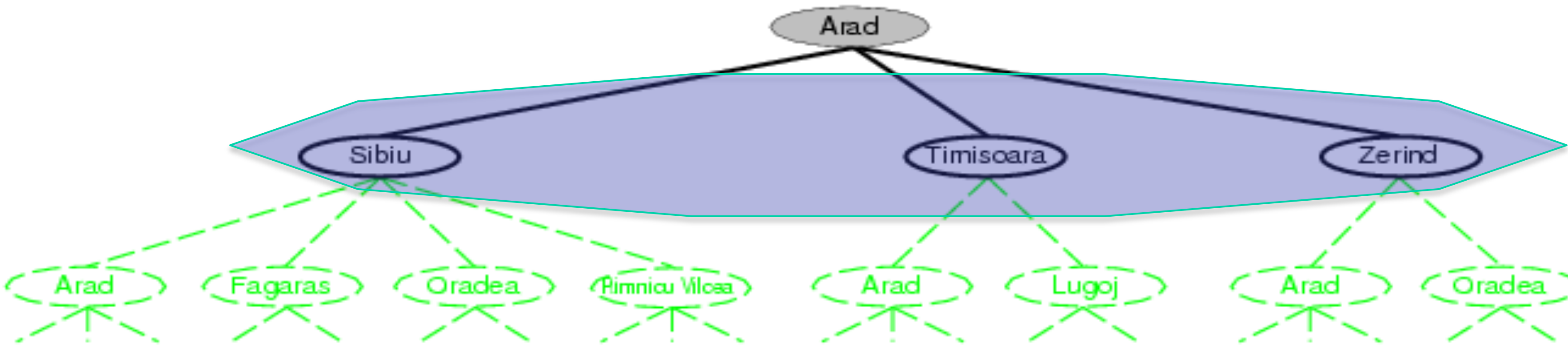
Tree search example

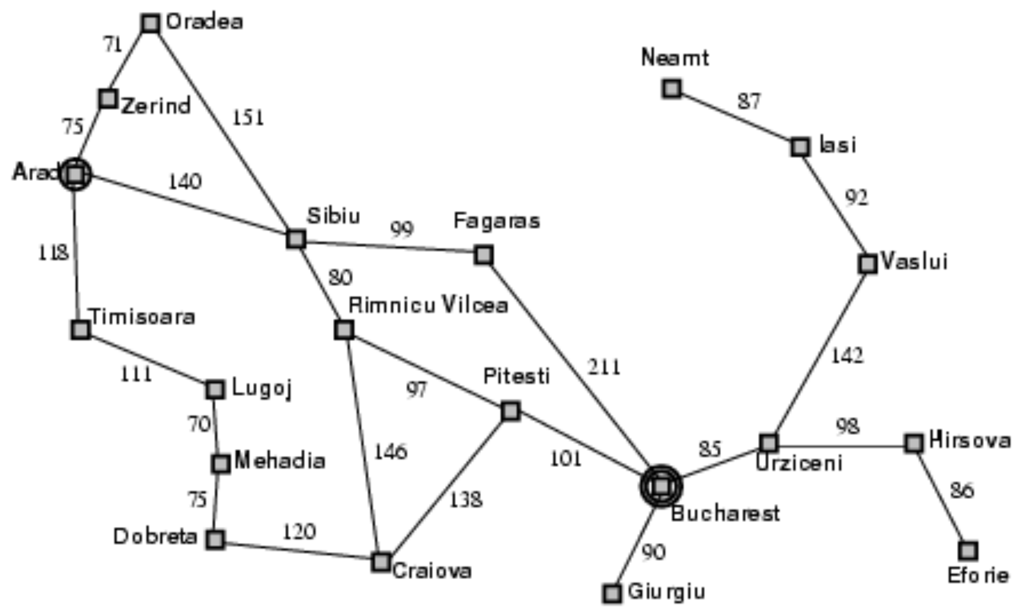
Node selected for expansion.





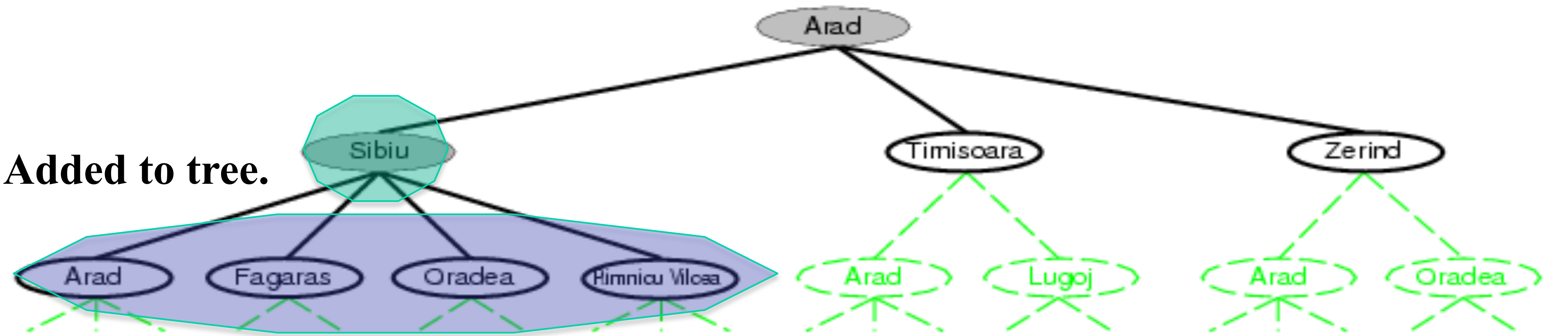
Nodes added to tree.





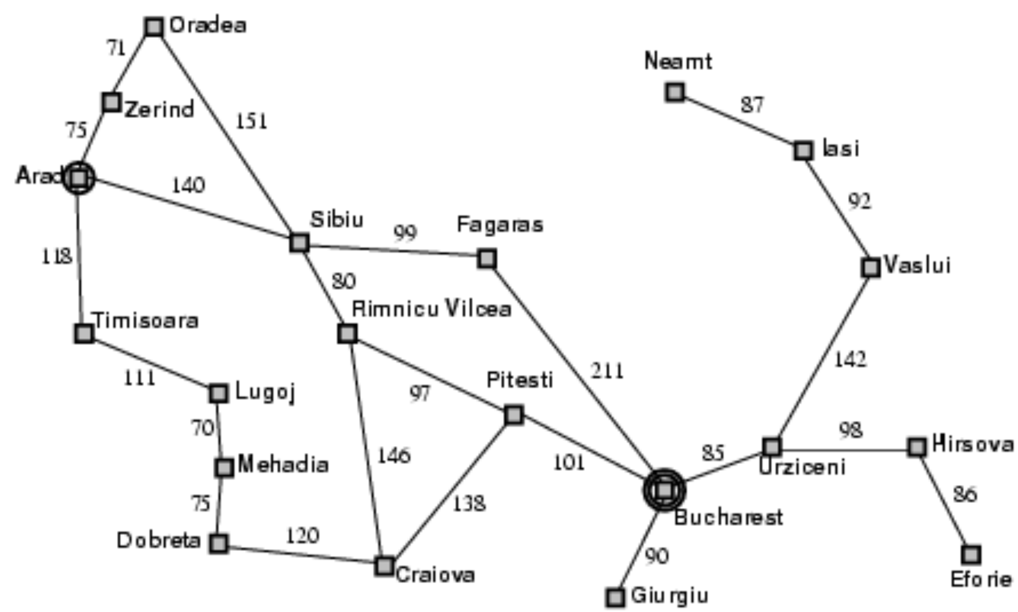
Selected for expansion.

Added to tree.



Note: Arad added (again) to tree! (reachable from Sibiu)

Not necessarily a problem, but in Graph-Search, we will avoid this by maintaining an "explored" list.



Graph-search

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, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Fig. 3.7 R&N, p. 77. See also exercise 3.13.

Note:
1) Uses "explored" set to avoid visiting already explored states.
2) Uses "frontier" set to store states that remain to be explored and expanded.
3) *However, with eg uniform cost search, we need to make a special check when node (i.e. state) is on frontier. Details later.*

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**
- **Bidirectional search**

Key issue: type of queue used for the fringe of the search tree (collection of tree nodes that have been generated but not yet expanded)

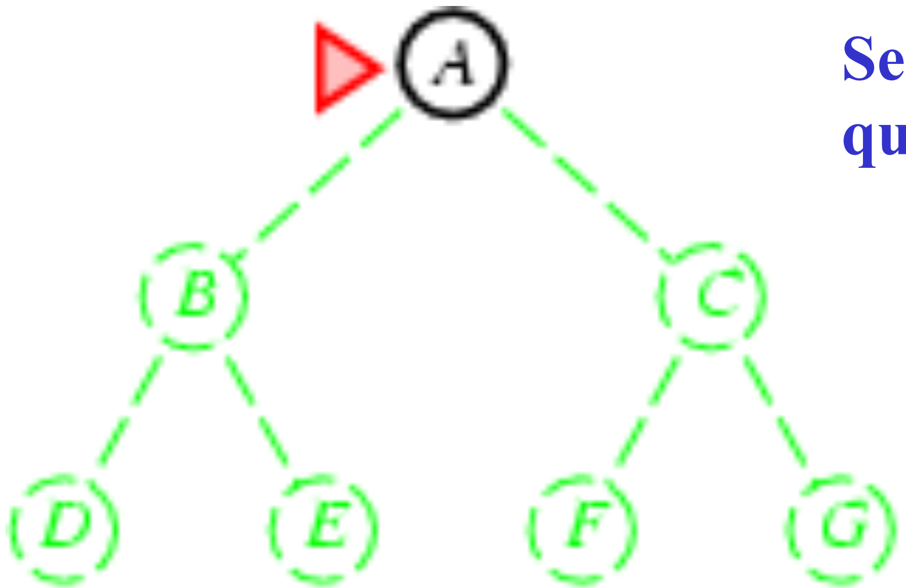
Breadth-first search

Expand shallowest unexpanded node.

Implementation:

- *fringe* **is a FIFO queue, i.e., new nodes go at end**
  (First In First Out queue.)

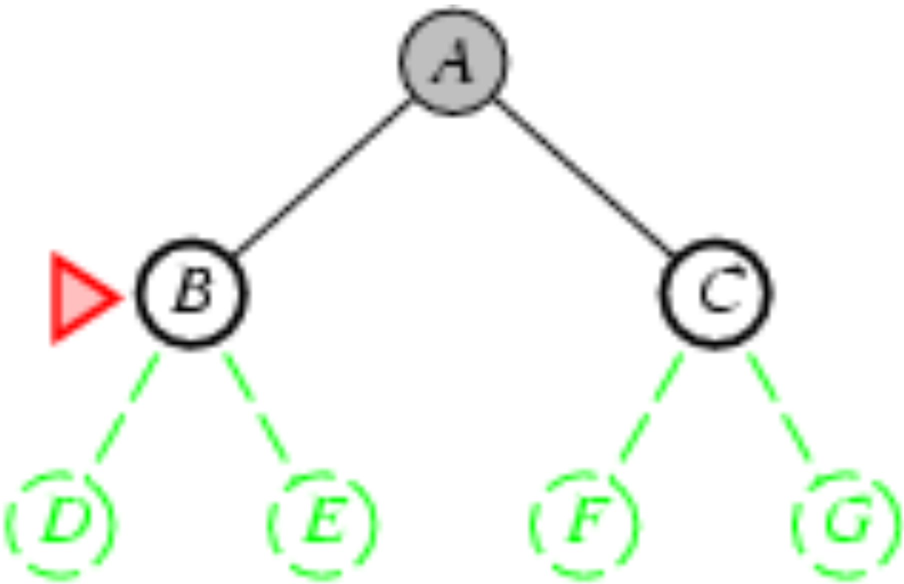
Fringe queue: <A>



Select A from queue and expand.

Gives <B, C>

Queue: <B, C>

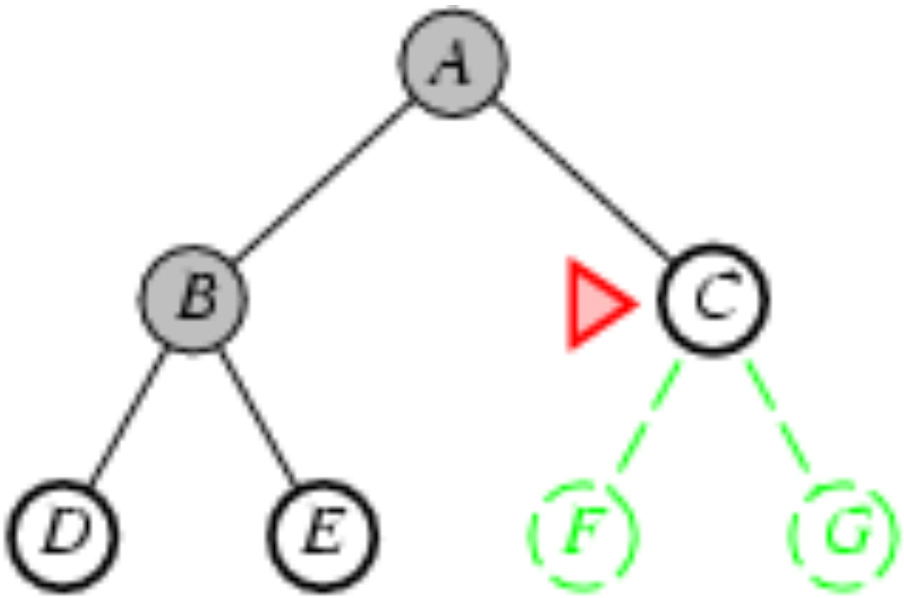


Select B from front, and expand.

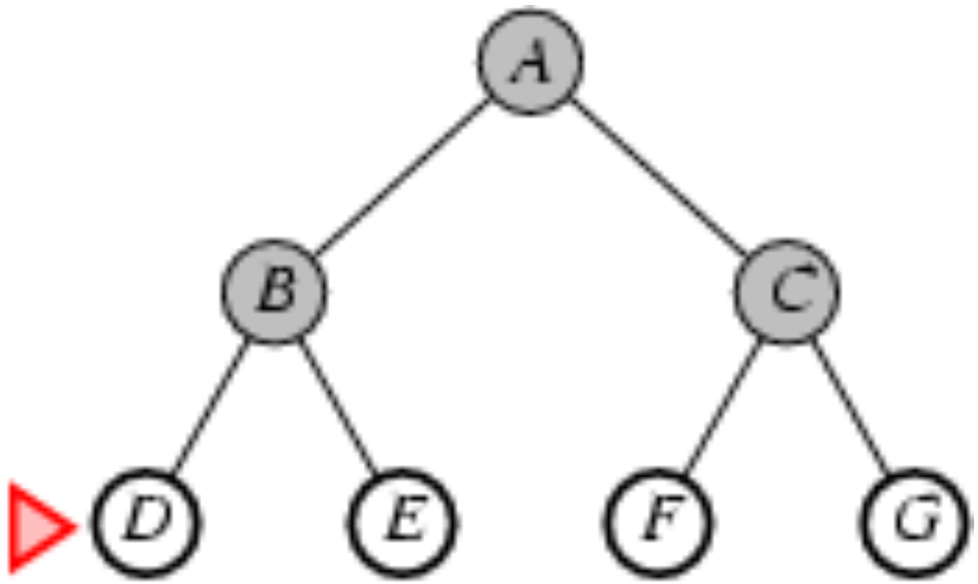
Put children at the end.

Gives <C, D, E>

Fringe queue: <C, D, E>



Fringe queue: <D, E, F, G>



Assuming no further children, queue becomes <E, F, G>, <F, G>, <G>, <>. Each time node checked for goal state.

Properties of breadth-first search

Note: check for goal only when node is expanded.

Complete? Yes (if b is finite)

Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$ Depth d, goal may be last node (only checked when expanded.).

Why?

Space? $O(b^{d+1})$ (keeps every node in memory; needed also to reconstruct soln. path)

Optimal soln. found?

Yes (if all step costs are identical)

Space is the bigger problem (more than time)

b: maximum branching factor of the search tree
d: depth of the least-cost solution

Uniform-cost search

Expand least-cost (of path to) unexpanded node
(e.g. useful for finding shortest path on map)

Implementation:

– ***fringe* = queue *ordered by path cost***

g – cost of reaching a node

Complete? Yes, if step cost $\geq \varepsilon$ **(>0)**

Time? # of nodes with $g \leq$ cost of optimal solution (C*),
$O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$

Space? # of nodes with $g \leq$ cost of optimal solution,
$O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$

Optimal? Yes – nodes expanded in increasing order of *g(n)*

Note: Some subtleties (e.g. checking for goal state).

See p 84 R&N. Also, next slide.

Uniform-cost search

Two subtleties: (bottom p. 83 Norvig)

1) **Do goal state test, only when a node is selected for expansion.**
   (*Reason: Bucharest may occur on frontier with a longer than optimal path. It won't be selected for expansion yet.* Other nodes will be expanded first, leading us to uncover a shorter path to Bucharest. See also point 2).

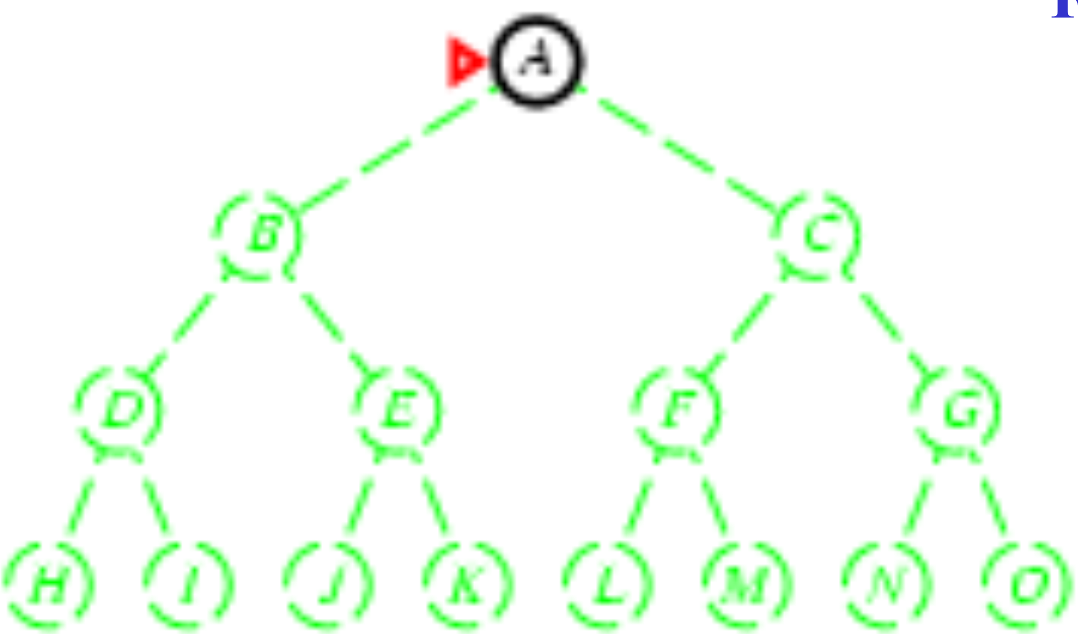
2) **Graph-search alg. says "don't add child node to frontier if already on explored list or already on frontier." BUT, child may give a shorter path to a state already on frontier. Then, we need to modify the existing node on frontier with the shorter path. See fig. 3.14 (else-if part).**

Depth-first search

"Expand deepest unexpanded node"

Implementation:

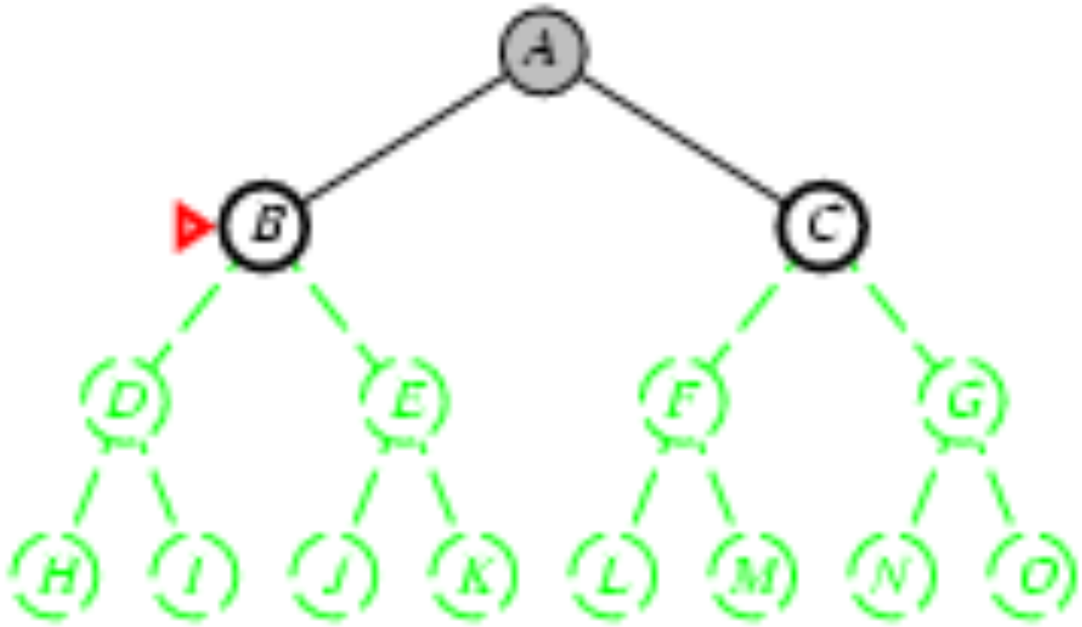
– *fringe* = LIFO queue, i.e., put successors at front ("push on stack")
  Last In First Out



Fringe stack:
A

Expanding A, gives stack:
B
C

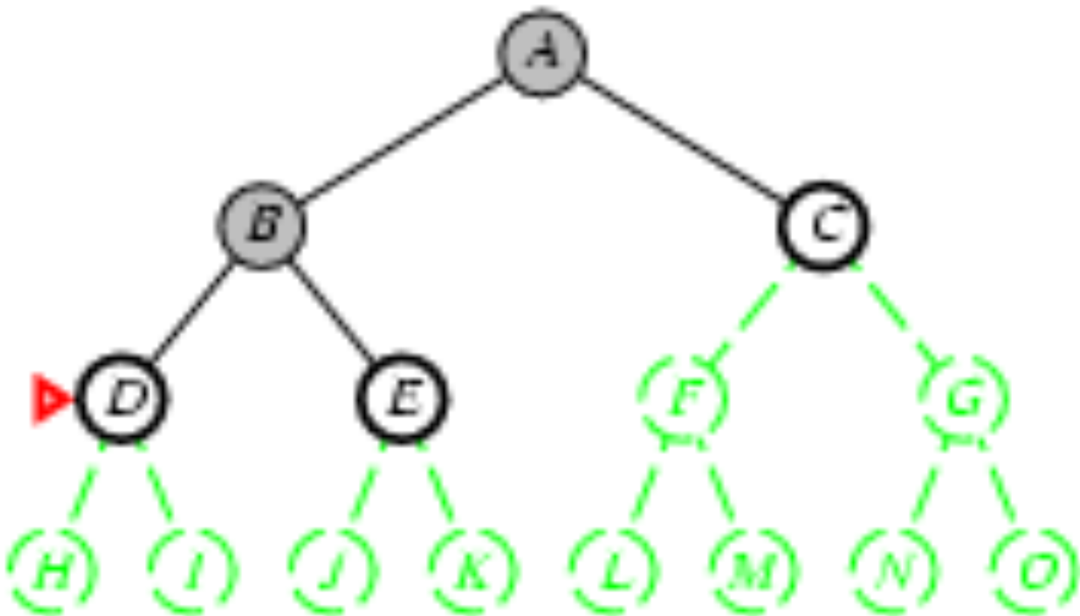
So, B next.



Expanding B,
gives stack:

D
E
C

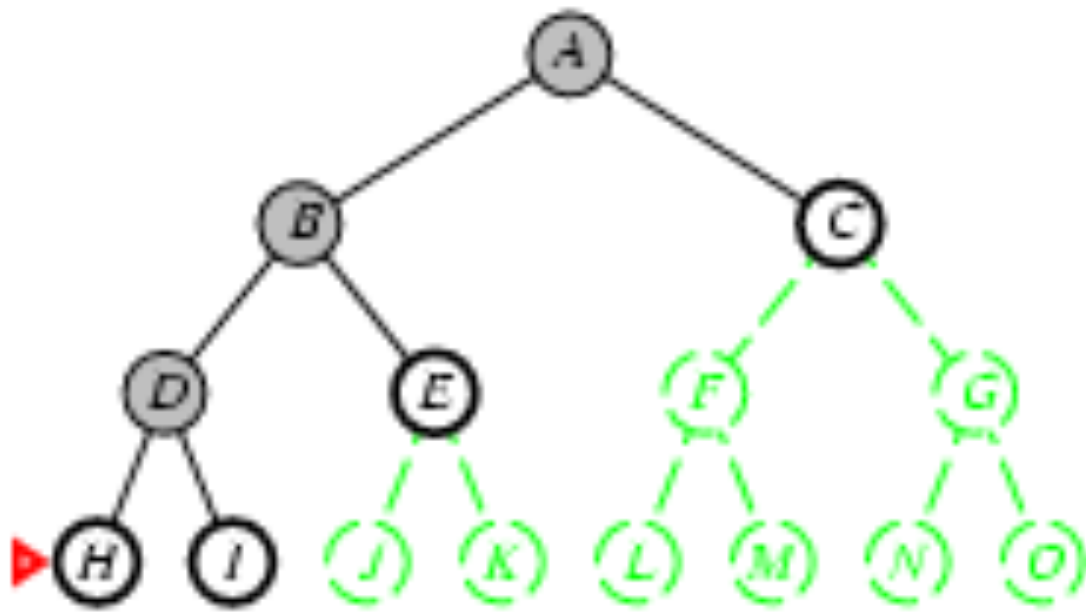
So, D next.

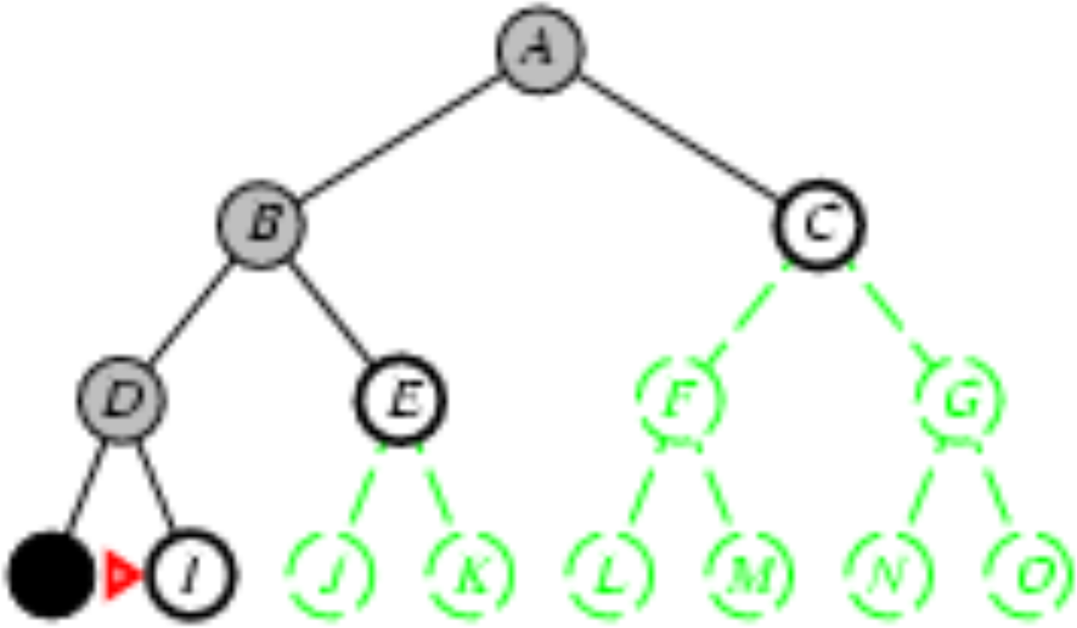


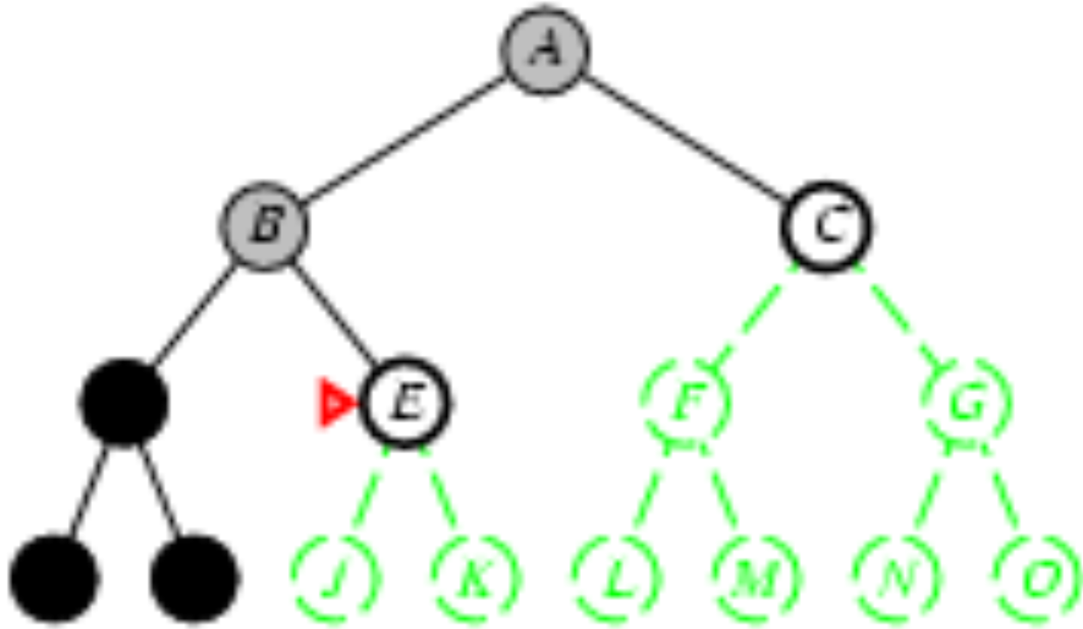
Expanding D, gives stack:

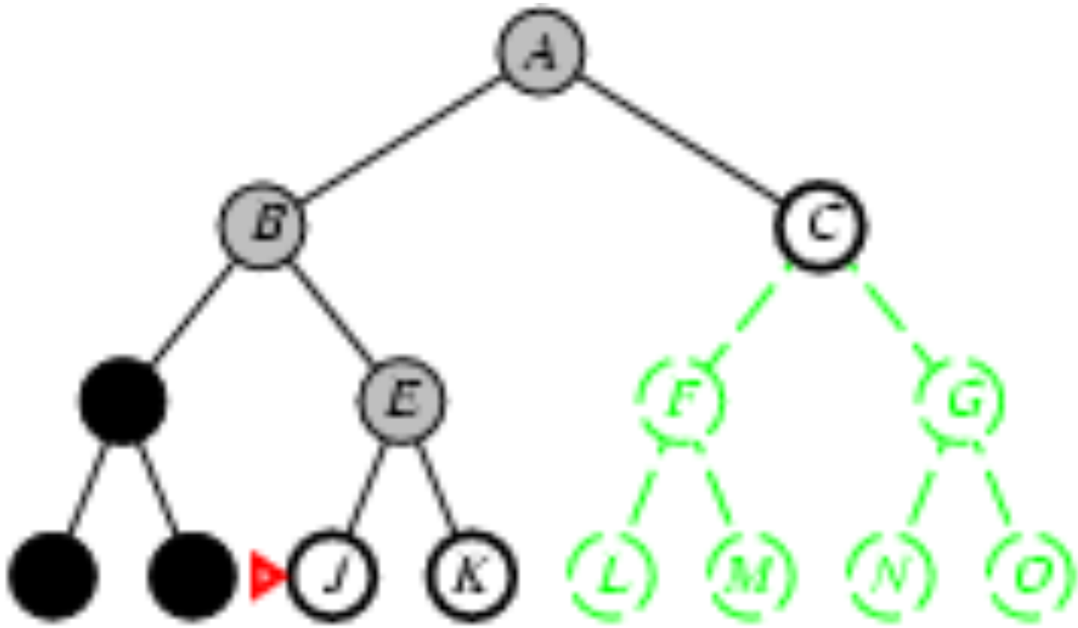
H
I
E
C

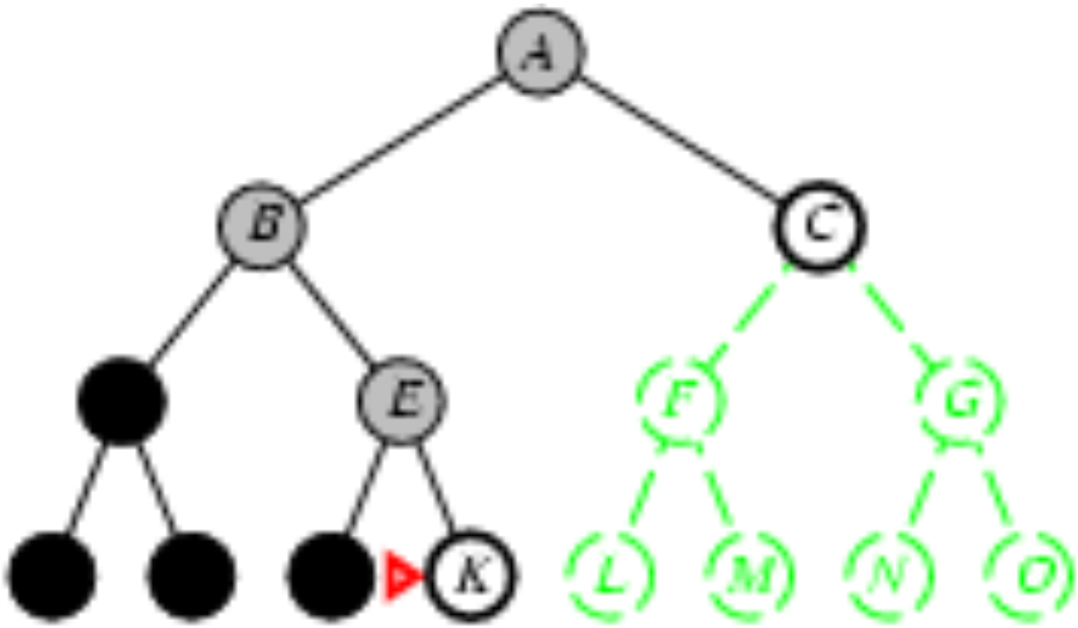
So, H next.
etc.

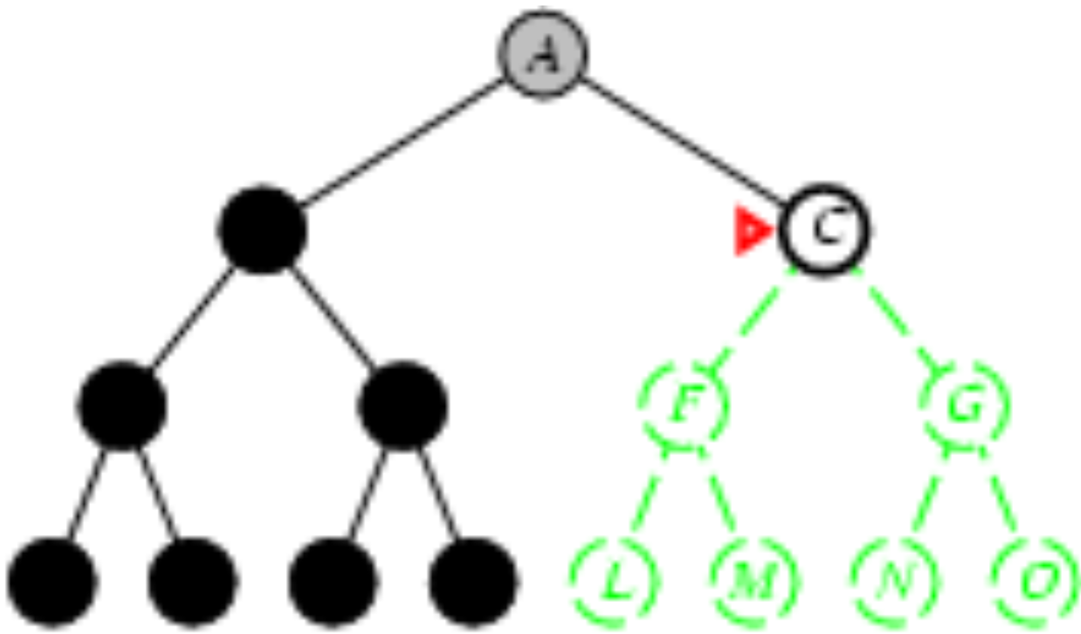

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
I
J
K
L
M
N
O

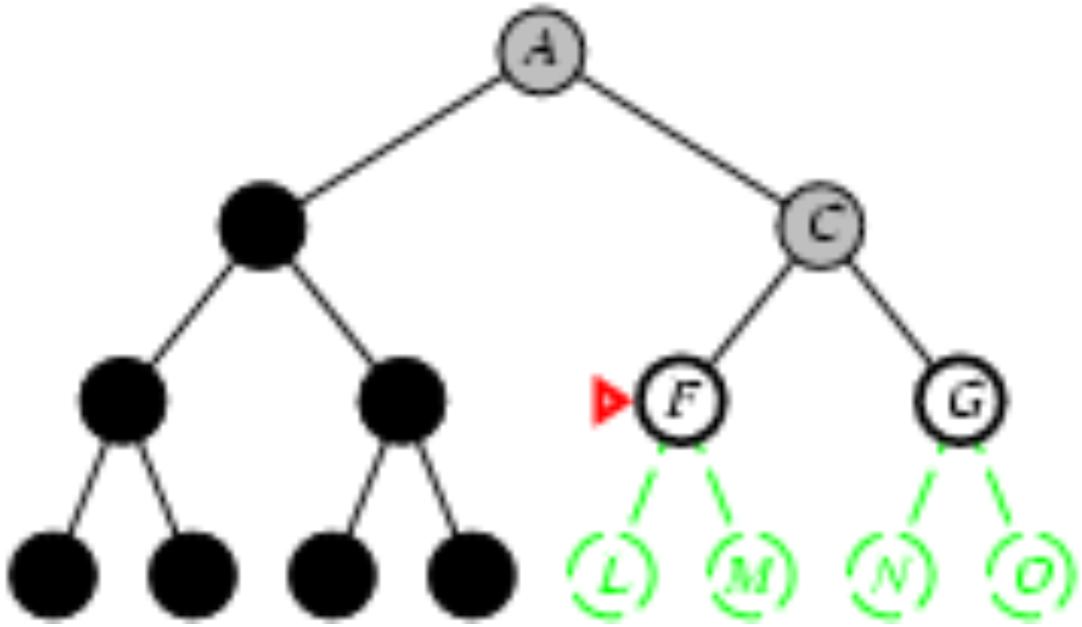

A
B
C
E
F
G
J
K
L
M
N
O

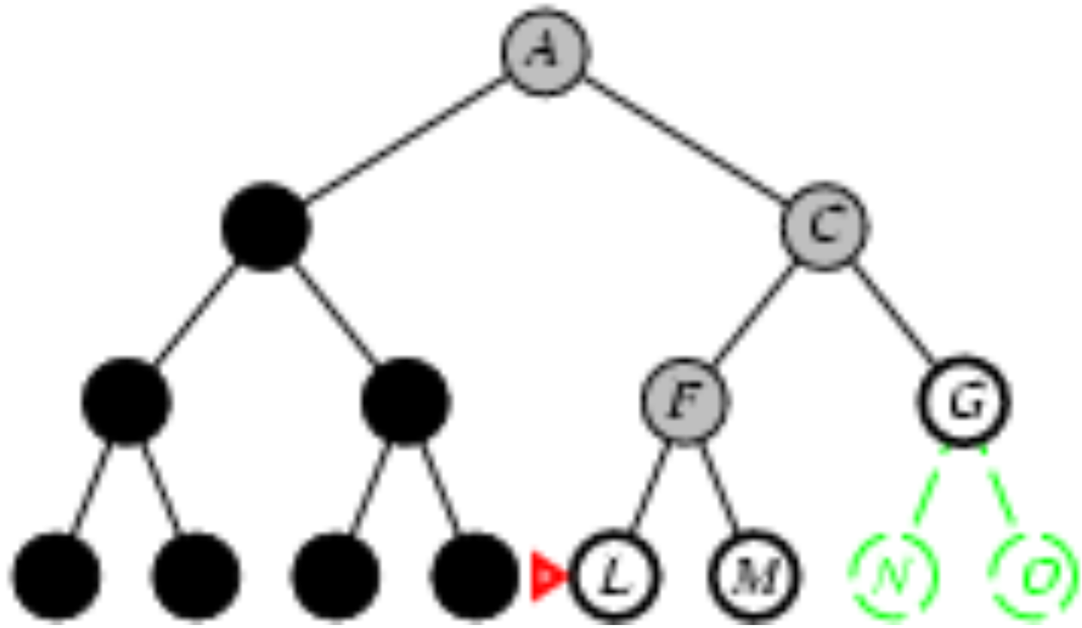

A
B
C
E
F
G
J
K
L
M
N
O

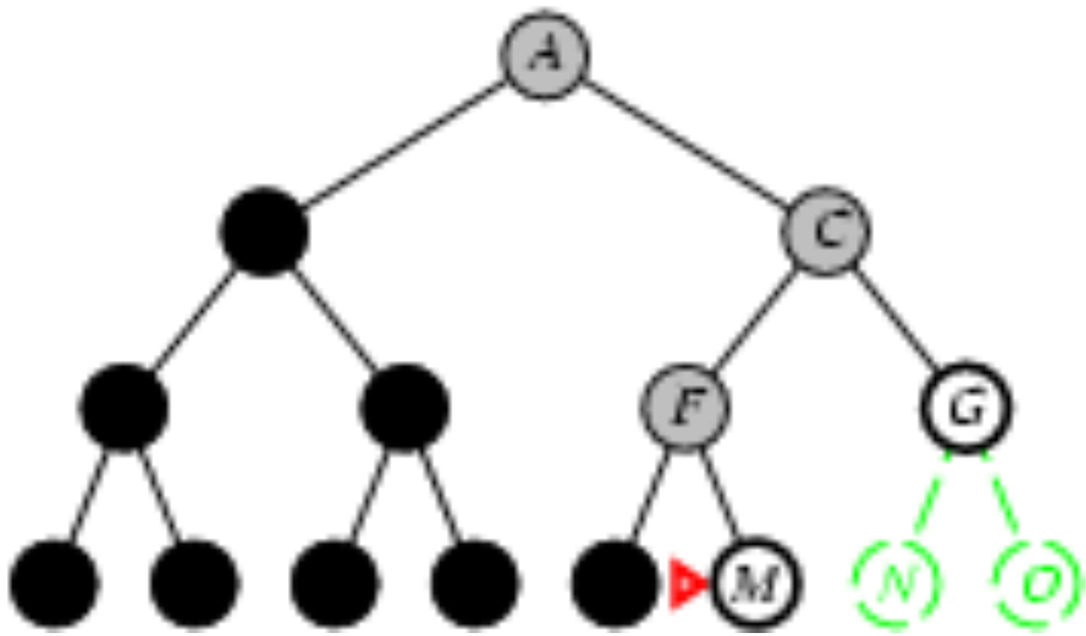

A
B
C
E
F
G
K
L
M
N
O



A
C
F
G
L
M
N
O


A
C
F
G
L
M
N
O


A
C
F
G
L
M
N
O


A
C
F
G
M
N
O

What is main advantage over breadth first search?

What information is stored? How much storage required?

The stack. O(depth x branching).

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? ***$O(b^m)$*: bad 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!**

Note: Can also reconstruct soln. path from single stored branch.

Guarantee that opt. soln. is found? No

***b:* max. branching factor of the search tree**
***d:* depth of the shallowest (least-cost) soln.**
m: maximum depth of state space

Note: In "backtrack search" only one successor is generated
→ only O(m) memory is needed; also successor is modification of the current state, but we have to be able to undo each modification.
More when we talk about Constraint Satisfaction Problems (CSP).

Iterative deepening search

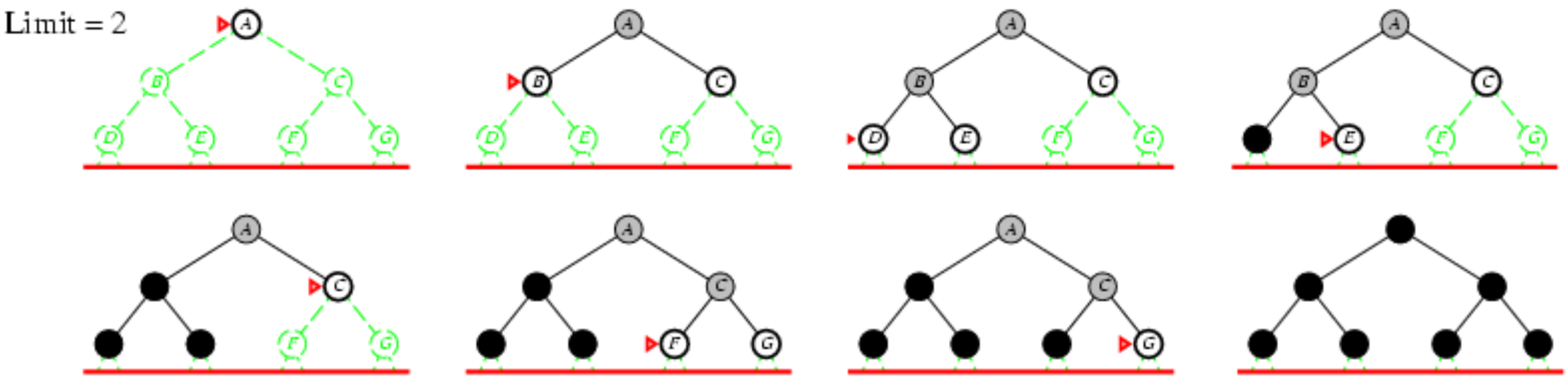
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  inputs: *problem*, a problem
  for *depth* ← 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*

Iterative deepening search l =0

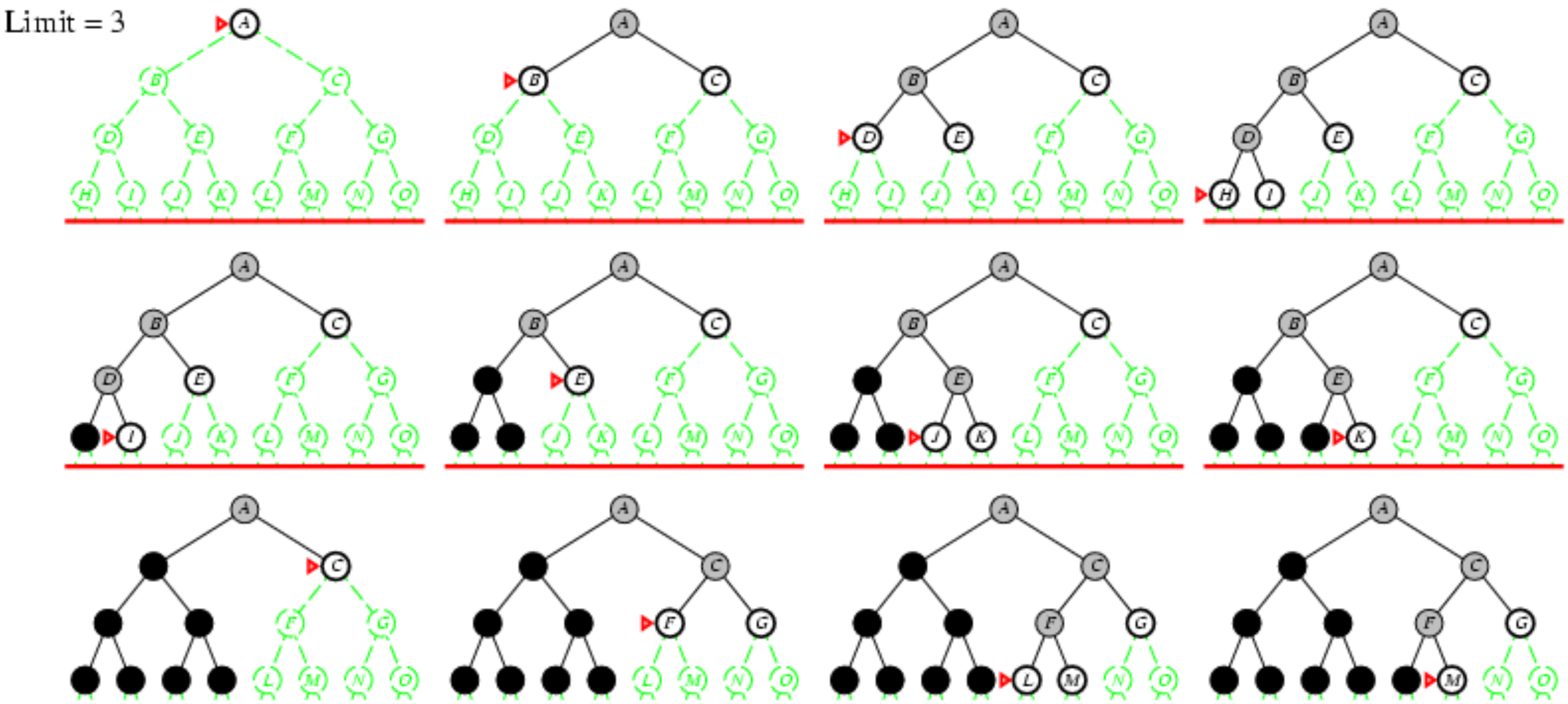
Limit = 0

Iterative deepening search l =1

Iterative deepening search l =2



Iterative deepening search l =3



Why would one do that?

Combine good memory requirements of depth-first with the completeness of breadth-first when branching factor is finite and is optimal when the path cost is a non-decreasing function of the depth of the node.

Idea was a breakthrough in game playing. All game tree search uses iterative deepening nowadays. What's the added advantage in games?

"Anytime" nature.

Iterative deepening search

Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

"One search depth d"

Looks quite wasteful, is it?

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

Nodes generated in a breadth-first search with branching factor *b*:

$$N_{BFS} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

For $b = 10$, $d = 5$,

- $N_{BFS} = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110$
- $N_{IDS} = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$ ☺

Iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Properties of iterative deepening search

Complete? **Yes**

(*b* finite)

Time? $d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

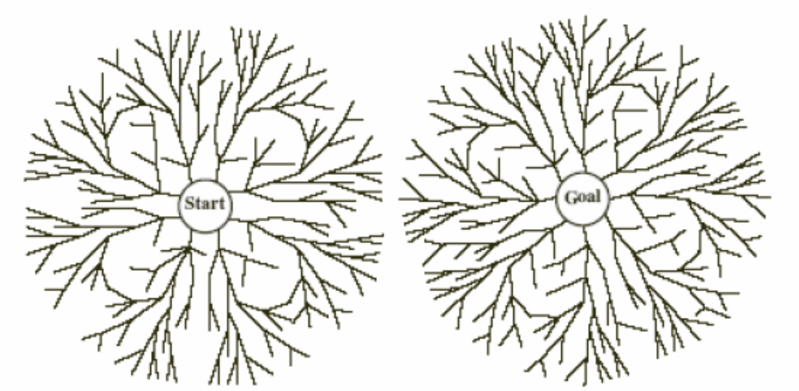
Space? $O(bd)$

Optimal? **Yes, if step costs identical**

Bidirectional Search

- Simultaneously:
  - Search forward from start
  - Search backward from the goal

  Stop when the two searches meet.



- If branching factor = b in each direction, with solution at depth d
  ➔ only $O(2\ b^{d/2}) = O(2\ b^{d/2})$

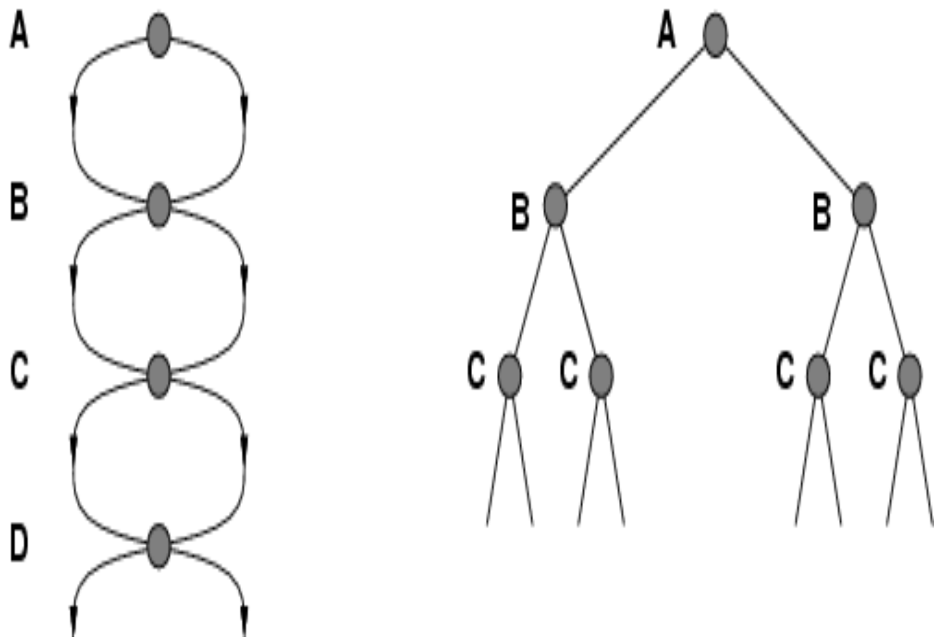
- Checking a node for membership in the other search tree can be done in constant time (hash table)

- Key limitations:
  Space $O(b^{d/2})$
  Also, how to search backwards can be an issue (e.g., in Chess)? What's tricky?
  Problem: lots of states satisfy the goal; don't know which one is relevant.

Aside: The predecessor of a node should be easily computable (i.e., actions are easily reversible).

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

Repeated states



Problems in which actions are reversible (e.g., routing problems or sliding-blocks puzzle). Also, in eg Chess; uses hash tables to check for repeated states. Huge tables 100M+ size but very useful.

See Tree-Search vs. Graph-Search in Fig. 3.7 R&N. But need to be careful to maintain (path) optimality and completeness.

Summary: General, uninformed search

Original search ideas in AI where inspired by studies of human problem solving in, eg, puzzles, math, and games, but a great many AI tasks now require some form of search (e.g. find optimal agent strategy; active learning; constraint reasoning; NP-complete problems require search).

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.**

Avoid repeating states / cycles.