CSE 473

Chapter 3
Problem Solving using Search

© The New Yorker collection. All rights reserved.
From The New Yorker Book of Technology Cartoons.

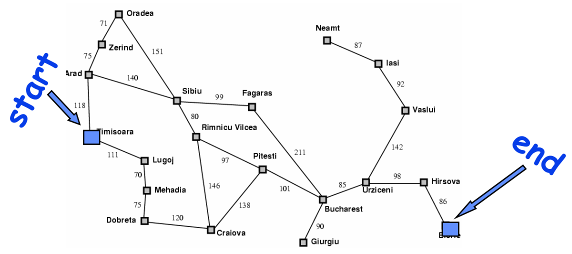
"First, they do an on-line search"

Example: The 8-puzzle

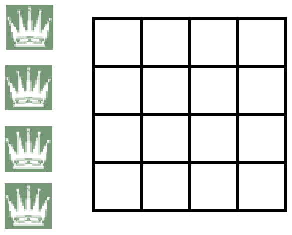
1	2	3
8		4
7	6	5

1	2	3
4	5	6
7	8	

Example: Route Planning

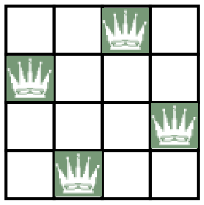


Example: N Queens



4 Queens

Example: N Queens



4 Queens

5

State-Space Search Problems

General problem:

Given a *start state*, find a path to a *goal state*

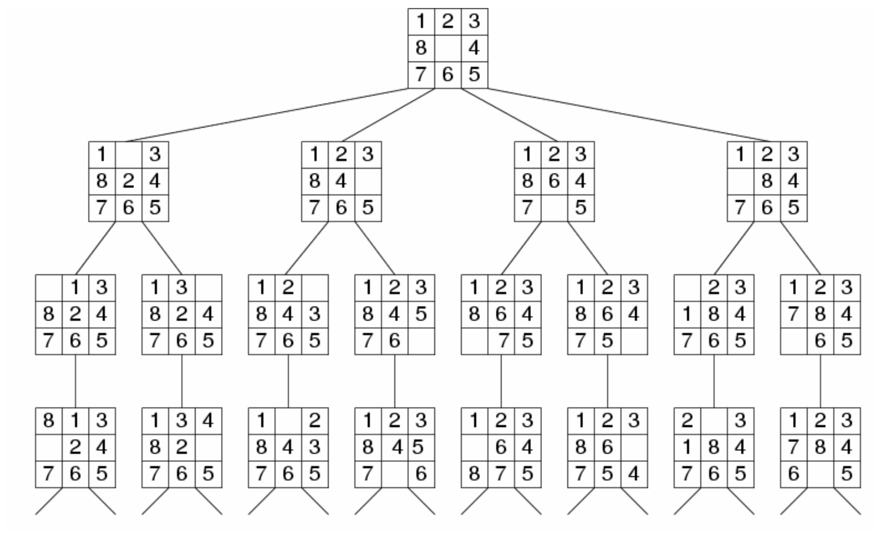
- **Can test if a state is a goal**
- **Given a state, can generate its *successor* states**

Variants:

- **Find any path *vs.* a least-cost path**
- **Goal is completely specified, task is just to find the path**
  - **Route planning**
- **Path doesn't matter, only finding the goal state**
  - **8 puzzle**

6

Tree Representation of 8-Puzzle Problem Space



Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

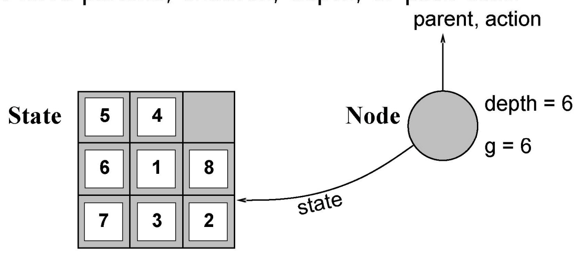
Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem] applied to STATE(node) succeeds return 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
```

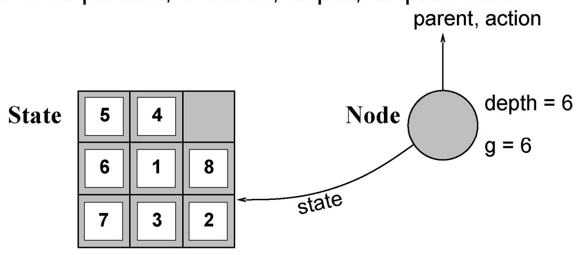
Implementation: states vs. nodes

A *state* is a (representation of) a physical configuration
A *node* is a data structure constituting part of a search tree
includes *parent*, *children*, *depth*, *path cost* $g(x)$
States do not have parents, children, depth, or path cost!



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 *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.

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 ∞)

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

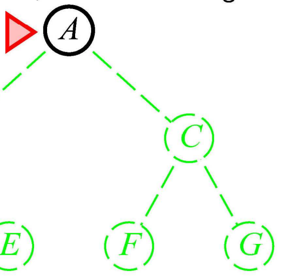
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

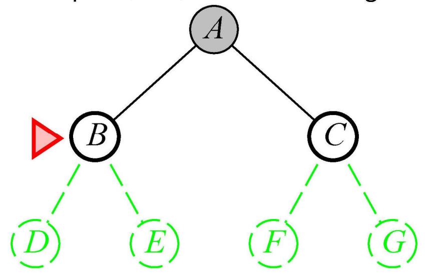
$fringe$ is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

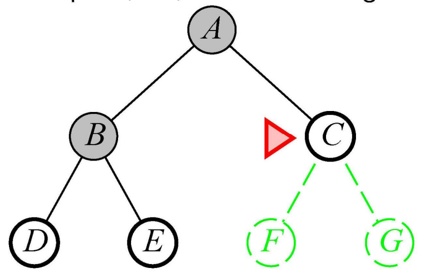
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

Implementation:
fringe is a FIFO queue, i.e., new successors go at end

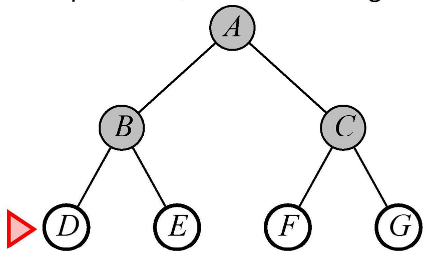


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Space is the big problem for BFS.

Example: b = 10, 10,000 nodes/sec, 1KB/node

d = 2 Ł 1100 nodes, 0.11 secs, 1MB

d = 4 Ł 111,100 nodes, 11 secs, 106 MB

d = 8 Ł 10^9 nodes, 31 hours, 1 TB

Uniform-cost search

Expand least-cost unexpanded node

Implementation:
    fringe = queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

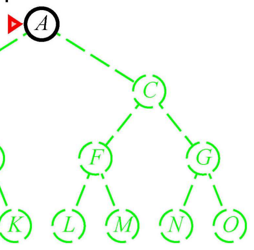
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

23

Depth-first search

Expand deepest unexpanded node

Implementation:
    fringe = LIFO queue, i.e., put successors at front



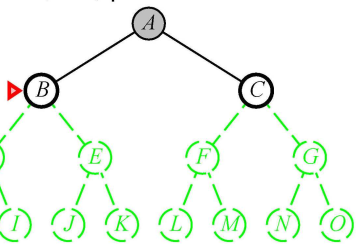
24

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



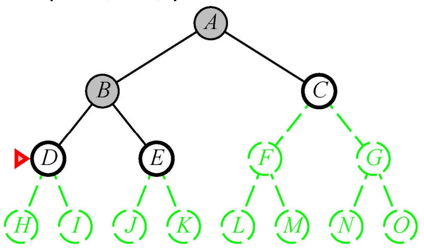
25

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

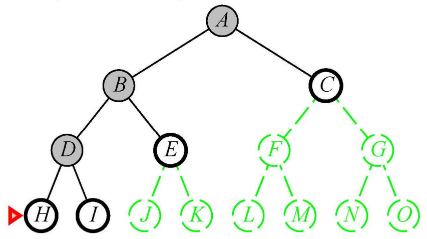


26

Depth-first search

Expand deepest unexpanded node

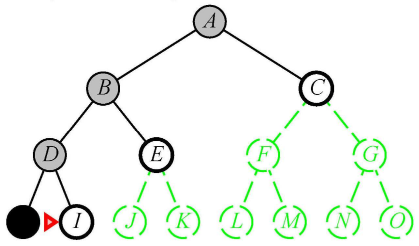
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front

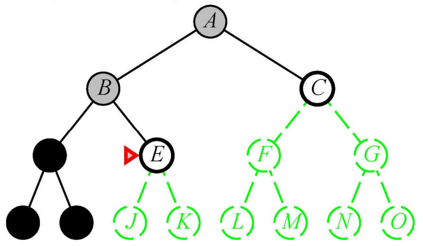


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front



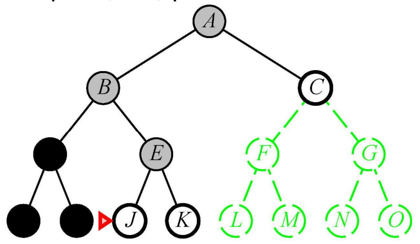
29

Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front



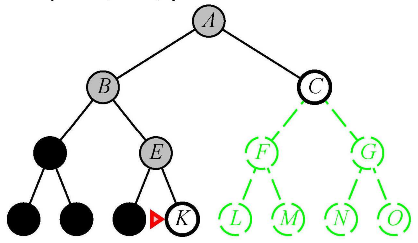
30

Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

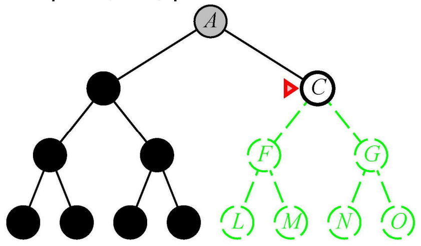


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

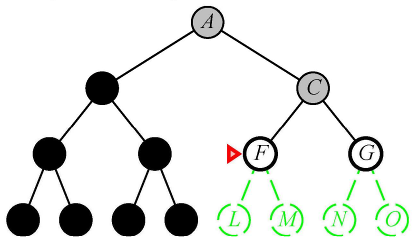


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

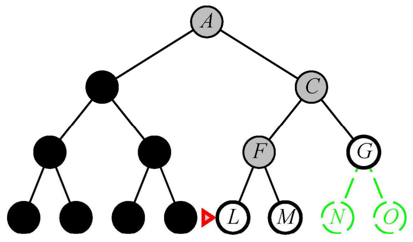


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

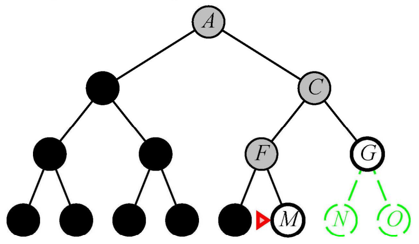


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



35

Properties of depth-first search

Complete??

36

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time??

37

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ 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??

38

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ 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??

39

Properties of depth-first search

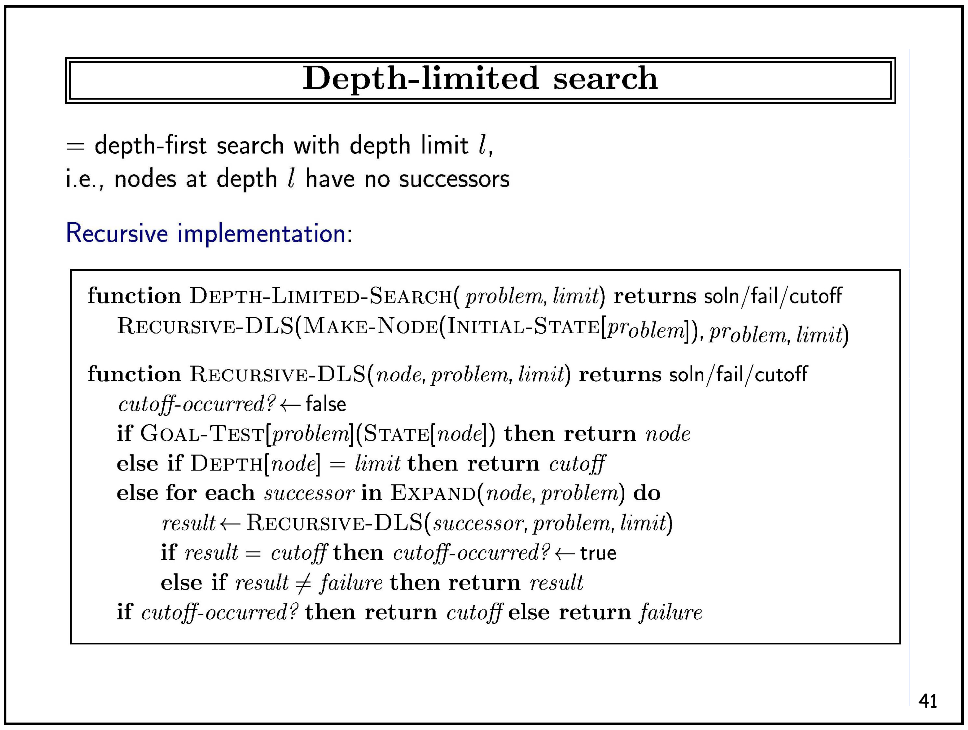
Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

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

40

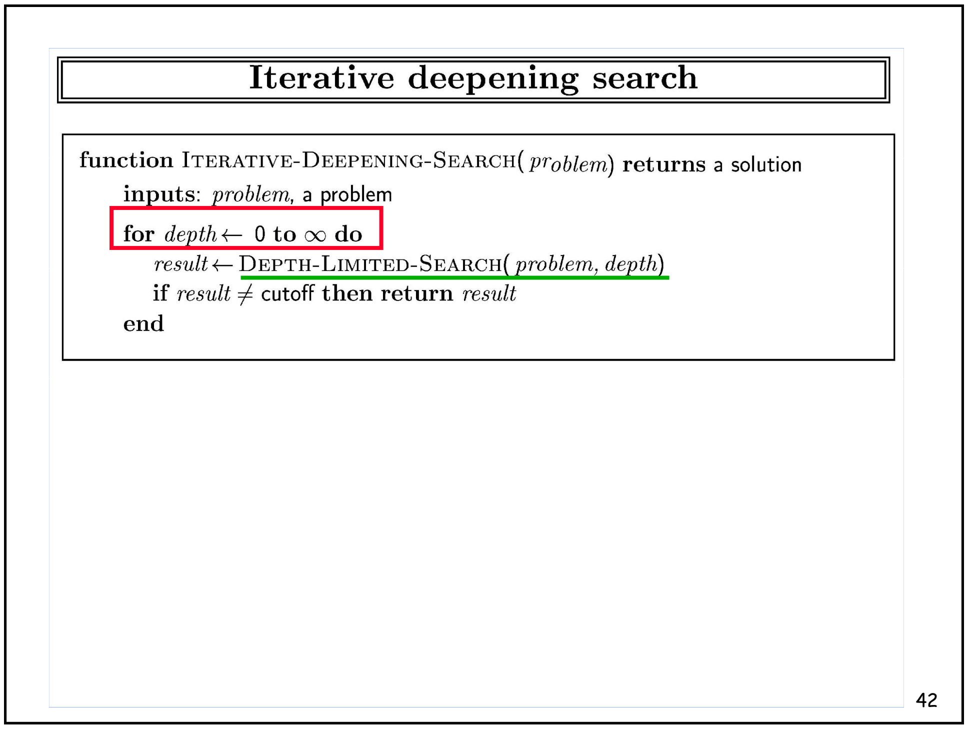


Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

Recursive implementation:

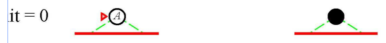
```
function Depth-Limited-Search( problem, limit) returns soln/fail/cutoff
   Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if Goal-Test[problem](State[node]) then return 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

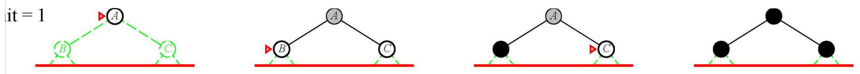
```
function Iterative-Deepening-Search( problem) returns a solution
   inputs: problem, a problem
   for depth ← 0 to ∞ do
      result ← Depth-Limited-Search( problem, depth)
      if result ≠ cutoff then return result
   end
```

Iterative deepening search $l = 0$



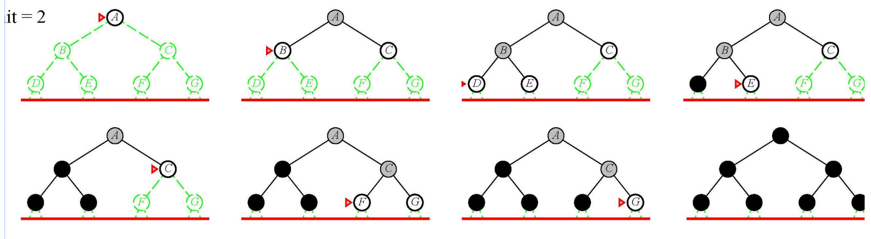
43

Iterative deepening search $l = 1$

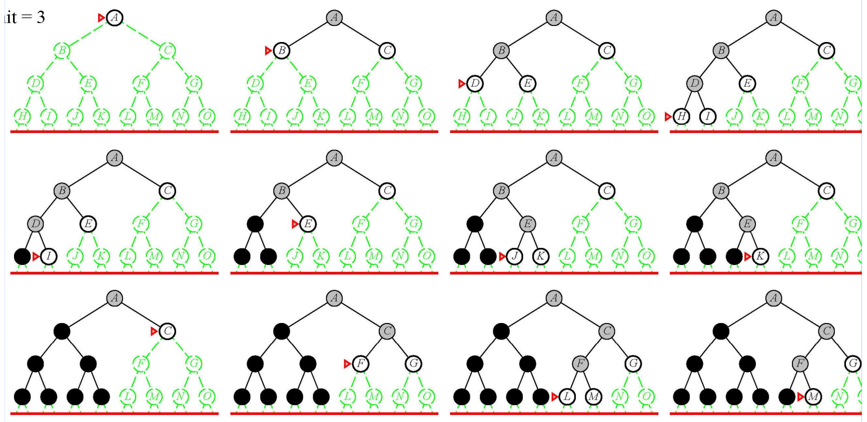


44

Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete??

47

Properties of iterative deepening search

Complete?? Yes

Time??

48

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space??

49

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

50

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree

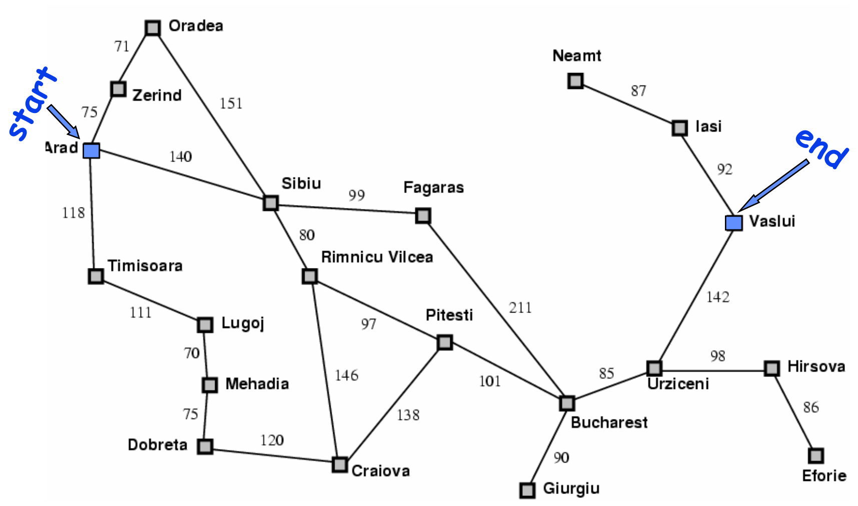
Increasing path-cost limits instead of depth limits

This is called Iterative lengthening search (exercise 3.11)

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes*	No	No	Yes

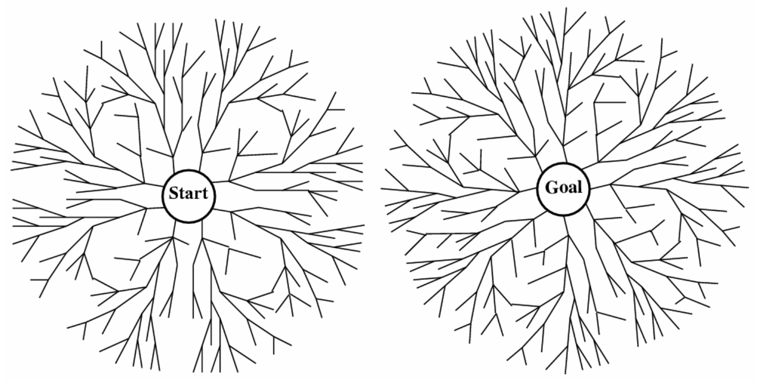
Forwards *vs.* Backwards



Problem: Find the shortest route

53

Bidirectional Search



Motivation: $b^{d/2} + b^{d/2} << b^d$

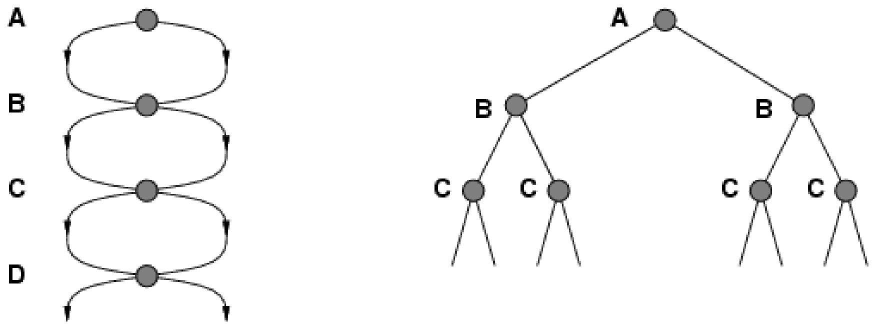
Can use breadth-first search or uniform-cost search

Hard for implicit goals e.g., goal = “checkmate” in chess

54

Repeated States

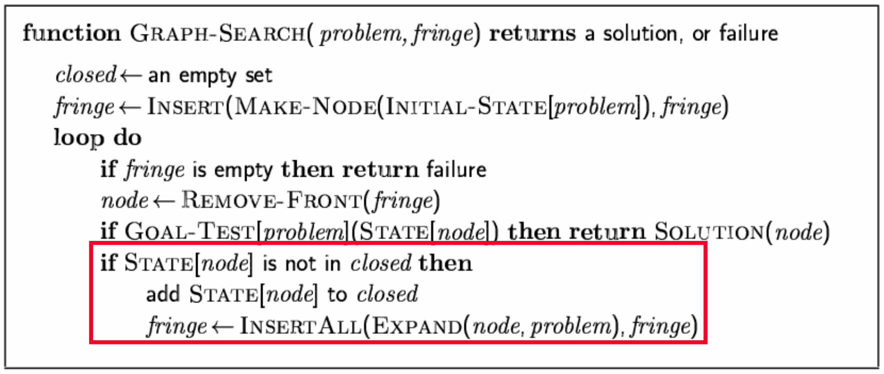
Failure to detect repeated states can turn a linear problem into an exponential one! (e.g., repeated states in 8 puzzle)



Graph search algorithm: Store expanded nodes in a set called *closed* and only add new nodes to the fringe

55

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)
```

56

Can we do better?

All these methods are slow (blind)

Solution use problem-specific knowledge to guide search ("heuristic function")
"informed search" (next lecture)

57