CSE 473

Chapter 3
Problem Solving using Search

"First, they do an on-line search"

Example: The 8-puzzle

Start

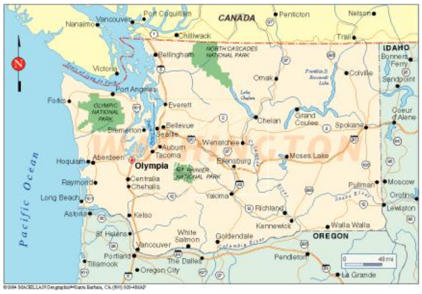
1	2	3
8		4
7	6	5

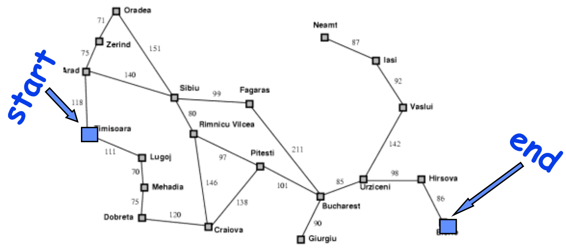
→

Goal

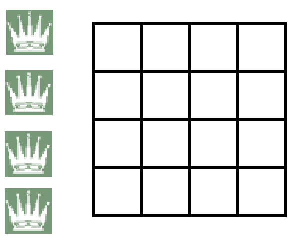
1	2	3
4	5	6
7	8	

Example: Route Planning





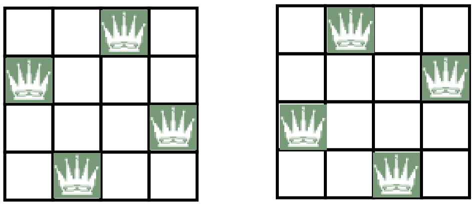
Example: N Queens



4 Queens problem

(Place queens such that no queen attacks any other)

Example: N Queens



4 Queens

State-Space Search Problems

General problem:

Find a path from a *start state* to a *goal state* given:

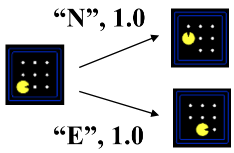
- **A goal test: Tests if a given state is a goal state**
- **A successor function (transition model): Given a state, generates its *successor* states**

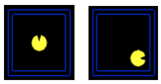
Variants:

- **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, N queens, Rubik's cube**

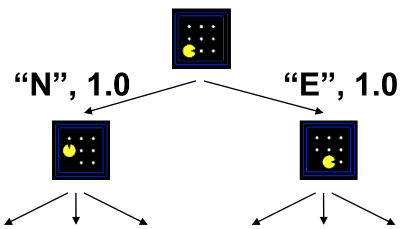
Example: Simplified Pac-Man

Input:

- State space 
- Successor function 

- Start state 
- Goal test 

Search Trees



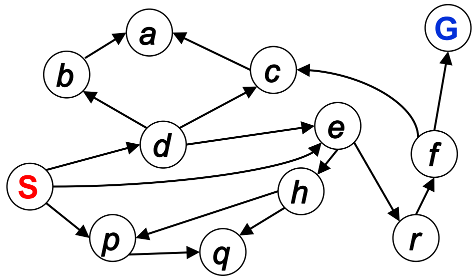
A search tree:

- Root = Start state
- Children = successor states
- Edges = actions and costs
- Path from Start to a node is a "plan" to get to that state
- For most problems, we can never actually build the whole tree (why?)

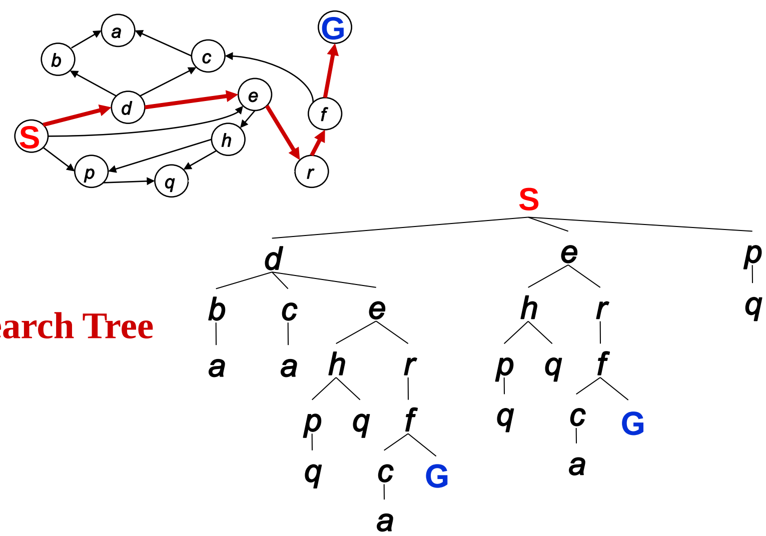
State Space Graph versus Search Trees

State Space Graph

(graph of states with arrows pointing to successors)

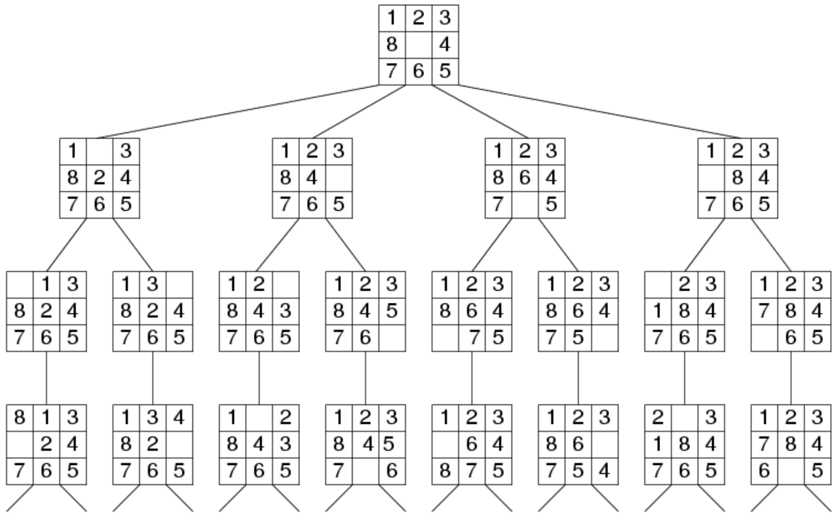


State Space Graph versus Search Trees



Search Tree

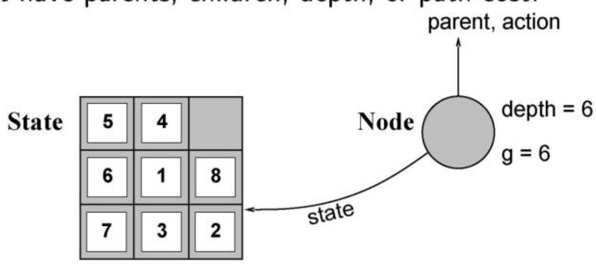
Search Tree for 8-Puzzle



11

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!



12

Searching with Search Trees

Search:

- Expand out possible nodes
- Maintain a **fringe** of as yet unexpanded nodes
- Try to expand as few tree nodes as possible

Implementation: general tree search

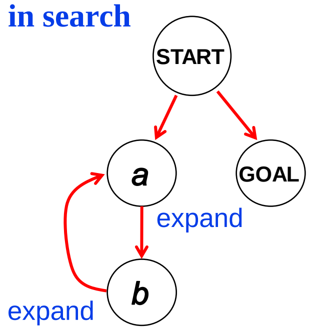
function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
  loop do
    if *fringe* is empty **then return** failure
    node ← REMOVE-FRONT(*fringe*)
    if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **return**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


Handling Repeated States

Failure to detect repeated states (e.g., in 8 puzzle) can cause infinite loops in search



Graph Search algorithm: Augment Tree-Search to store expanded nodes in a set called *explored set (or closed set)* and only add *new* nodes not in the explored set to the fringe

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

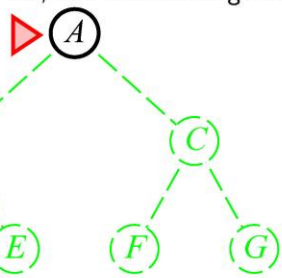
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

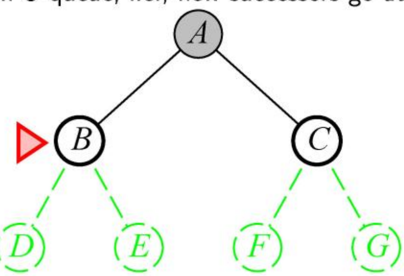


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

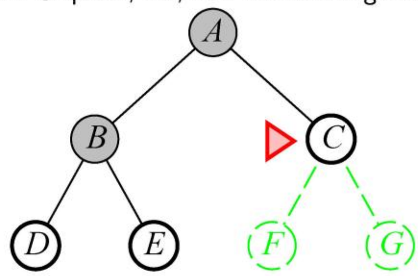


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

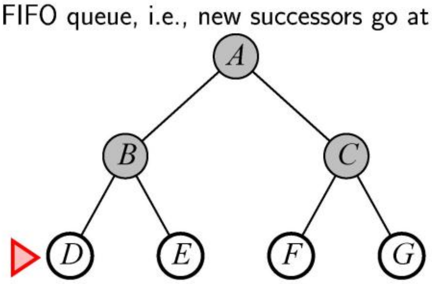


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Properties of breadth-first search

Complete??

23

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

24

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $b + b^2 + b^3 + \cdots + b^d = O(b^d)$ i.e. exp in d

Space??

25

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $b + b^2 + b^3 + \cdots + b^d = O(b^d)$ i.e. exp in d

Space?? $O(b^d)$

Optimal??

26

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $b + b^2 + b^3 + \cdots + b^d = O(b^d)$ i.e. exp in d

Space?? $O(b^d)$

Optimal?? Yes if all step costs are equal. Not optimal in general.

Space and time are big problems for BFS.

Example: b = 10, 1000,000 nodes/sec, 1000 Bytes/node

d = 2 → 110 nodes, 0.11 millisecs, 107KB

d = 4 → 11,110 nodes, 11 millisecs, 10.6 MB

d = 8 → 10^8 nodes, 2 minutes, 103 GB

d = 16 → 10^{16} nodes, 350 years, 10 EB (1 billion GB)

What if the step costs are not equal?

Can we modify BFS to handle any step cost function?

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

$fringe$ = queue ordered by path cost $g(n)$ (Use priority queue)

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, $O(b^{\lfloor C^*/\varepsilon \rfloor + 1})$
where C^* is the cost of the optimal solution

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

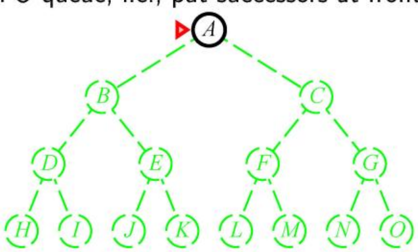
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

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

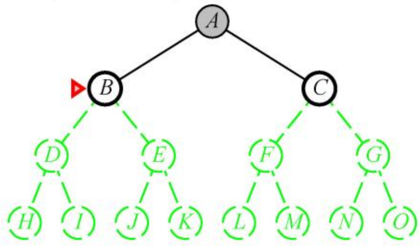


Depth-first search

Expand deepest unexpanded node

Implementation:

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

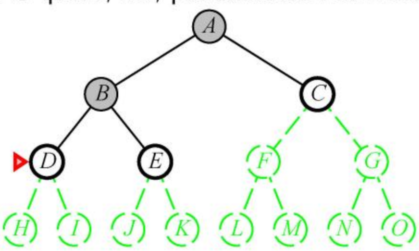


Depth-first search

Expand deepest unexpanded node

Implementation:

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

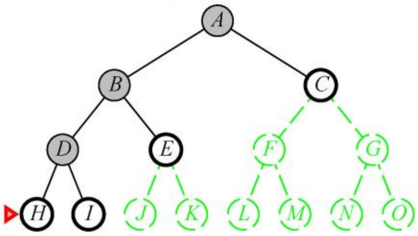


Depth-first search

Expand deepest unexpanded node

Implementation:

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

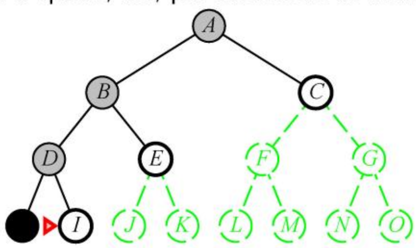


Depth-first search

Expand deepest unexpanded node

Implementation:

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

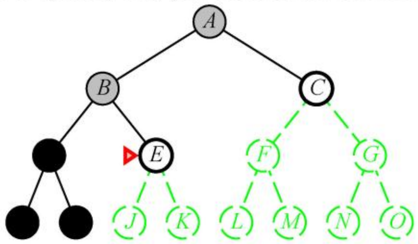


Depth-first search

Expand deepest unexpanded node

Implementation:

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



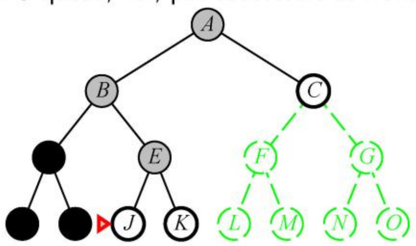
35

Depth-first search

Expand deepest unexpanded node

Implementation:

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



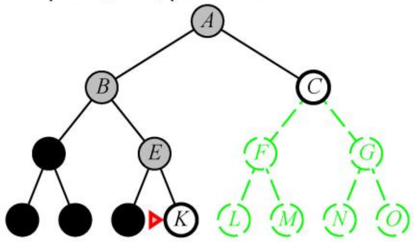
36

Depth-first search

Expand deepest unexpanded node

Implementation:

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



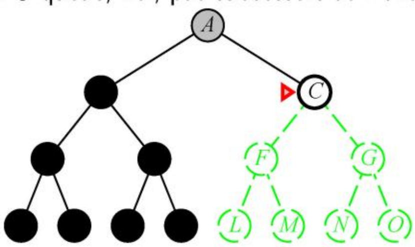
37

Depth-first search

Expand deepest unexpanded node

Implementation:

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



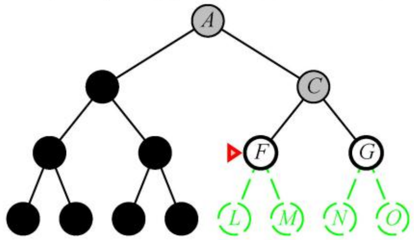
38

Depth-first search

Expand deepest unexpanded node

Implementation:

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

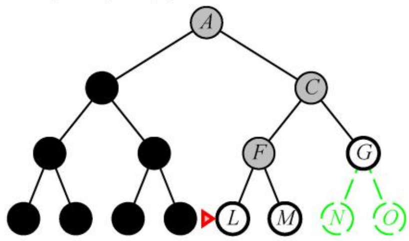


Depth-first search

Expand deepest unexpanded node

Implementation:

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

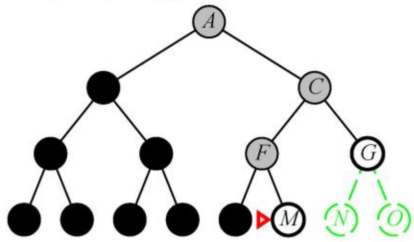


Depth-first search

Expand deepest unexpanded node

Implementation:

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



Properties of depth-first search

Complete??

Properties of depth-first search

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

Time??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path (using "explored" set)
$\Rightarrow$ 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??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path (using "explored" set)
$\Rightarrow$ 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??

45

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path (using "explored" set)
$\Rightarrow$ 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

Space cost is a big advantage of DFS over BFS.

Example: b = 10, 1000 Bytes/node

d = 16 → 156 KB instead of 10 EB (1 billion GB)

46

Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors (can handle infinite state spaces)

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
```

47

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
```

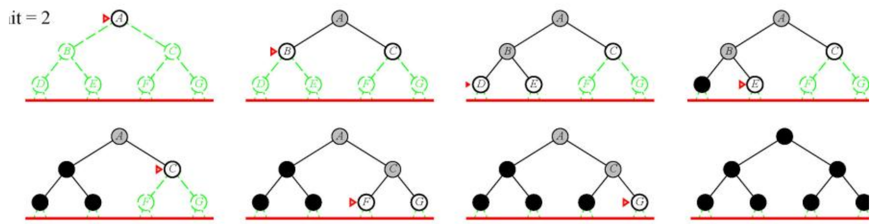
- **DFS with increasing depth limit**
- **Finds the best depth limit**
- **Combines the benefits of DFS and BFS**

48

Iterative deepening search $l = 0$

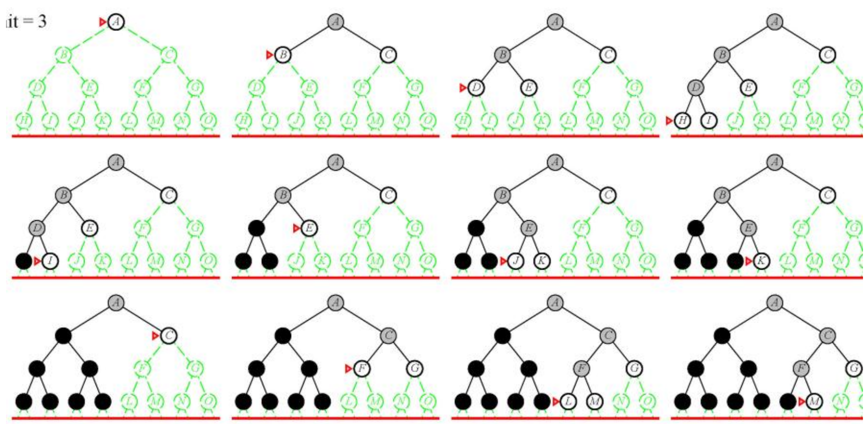
Iterative deepening search $l = 1$

Iterative deepening search $l = 2$



51

Iterative deepening search $l = 3$



52

Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

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

Space??

Properties of iterative deepening search

Complete?? Yes

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

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

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

Space?? $O(bd)$

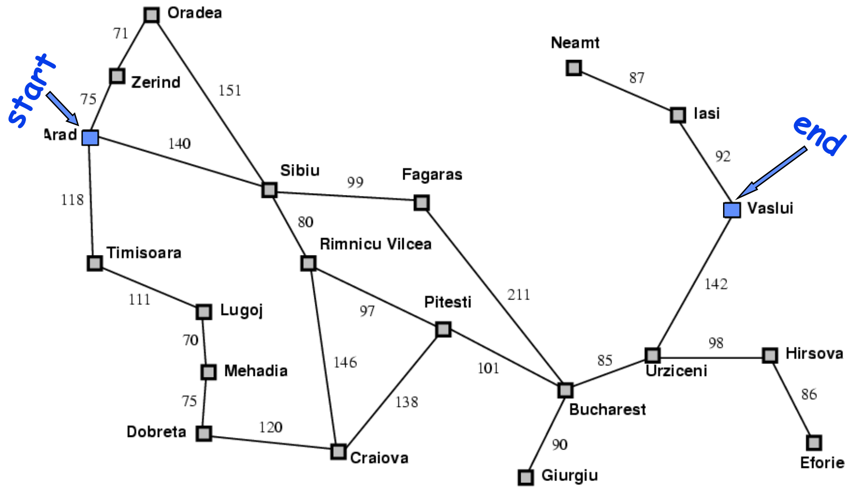
Optimal?? Yes if all step costs are equal. Not optimal in general.

Can be modified to explore uniform-cost tree

Increasing path-cost limits instead of depth limits

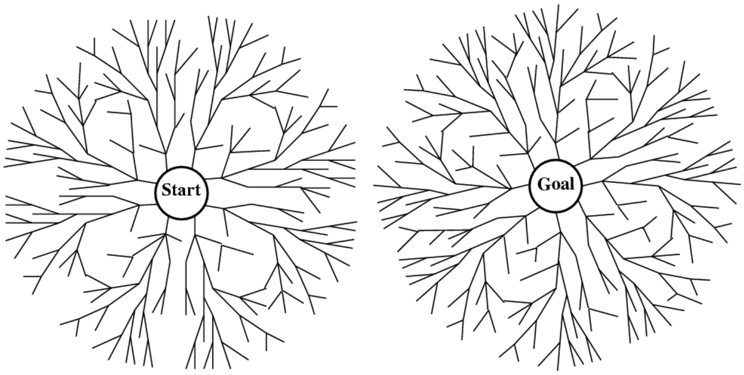
This is called Iterative lengthening search (exercise 3.17)

Forwards *vs.* Backwards



Problem: Find the shortest route

Bidirectional Search



Motivation: $b^{d/2} + b^{d/2} << b^d$ (E.g., $10^8+10^8 = 2 \cdot 10^8 << 10^{16}$)

Can use breadth-first search or uniform-cost search

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

Can we do better?

All these methods are slow (because they are "blind")

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

To Do

- Start Project #1
- Read Chapter 3