Last update: September 6, 2012

Problem solving by searching

CMSC 421: Chapter 3, Sections 1–4

Motivation and Outline

◇ Lots of AI problem-solving requires trial-and-error search
Chapter 3 describes some algorithms for this

- Types of problems and agents
- Problem formulation
- Example problems
- Basic search algorithms

Problem types

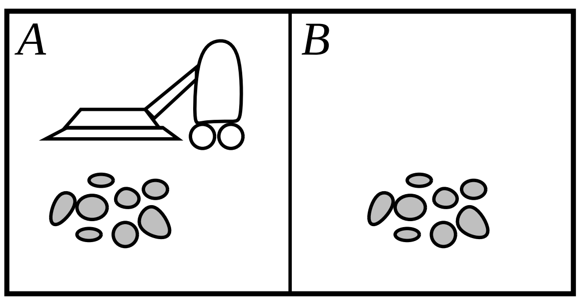
Deterministic, fully observable $\Longrightarrow$ *classical search problem*

- agent knows exactly which state it starts in, what each action does
- no exogenous events (or else they're encoded into the actions' effects)

◇ Solution is a sequence, can predict future states exactly

◇ Example: Vacuum World with **no** exogenous events

- ⋄ Rooms won't spontaneously get dirty again

- Initial state: 

- Goal: have both rooms clean
- Solution: $[Suck, Right, Suck]$

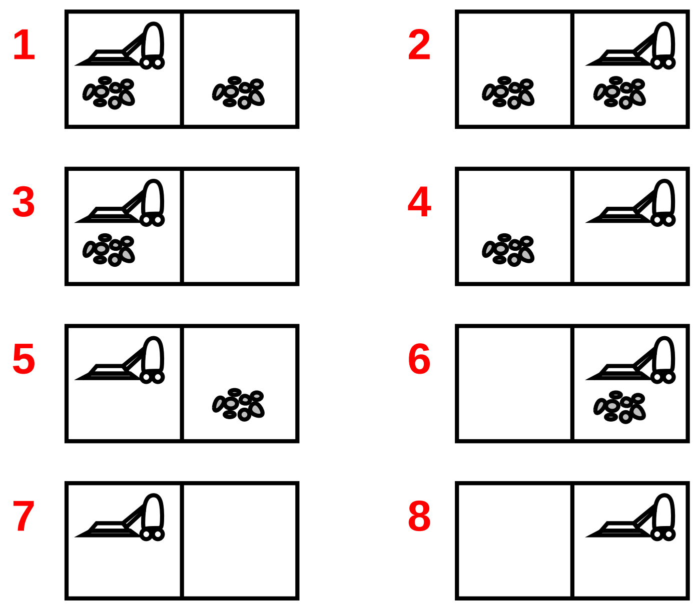
Problem types

Non-observable:

- Agent may have no idea where it is
- Solution (if any) must be a sequence that is *conformant*
  - ◇ Guaranteed to work under all conditions

◇ Example:

- Vacuum World, no exogenous events, and no sensors
- Initial state: could be any, agent has no way to know which
- Goal: both rooms clean
- Assume it's OK to hit the wall
- Solution: $[Right, Suck, Left, Suck]$



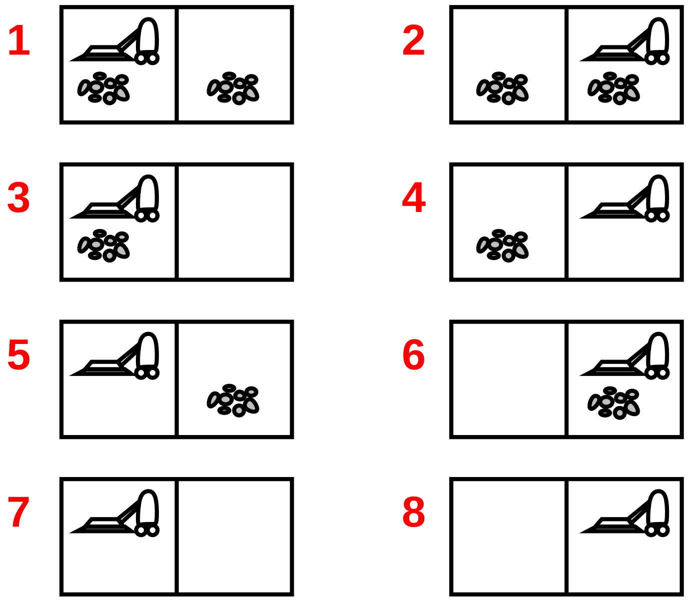
Problem types

Nondeterministic and/or **partially observable**:

- percepts provide new information about current state
- solution is a *contingent plan* or a *policy*
- often **interleave** search, execution

◇ Example:

- Vacuum World, no exogenous events, and *local sensing*:
  - ◇ which room the agent's in
  - ◇ whether that room is dirty
- Initial state: any of $\{5, 6, 7, 8\}$
- Goal: have both rooms clean
- Solution: $[Right,$ **if** $dirt$ **then** $Suck]$

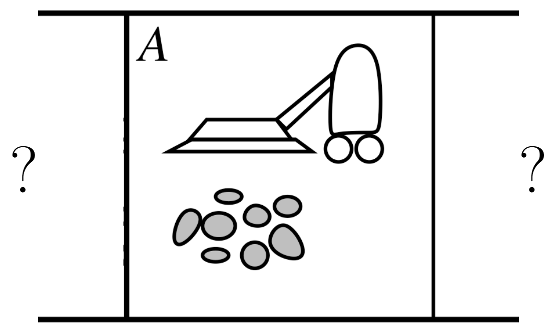


Problem types

◇ **Unknown state space** ⇒ *exploration problem*

◇ Example:

- Vacuum agent with local sensing
  - ◇ Initially, agent sees current location,
    but doesn't know what other rooms there are, or what's in them



Problem-solving agents

- *Online* problem solving: gather knowledge as you go
  - Necessary for exploration problems
  - Can be useful in nondeterministic and partially observable problems

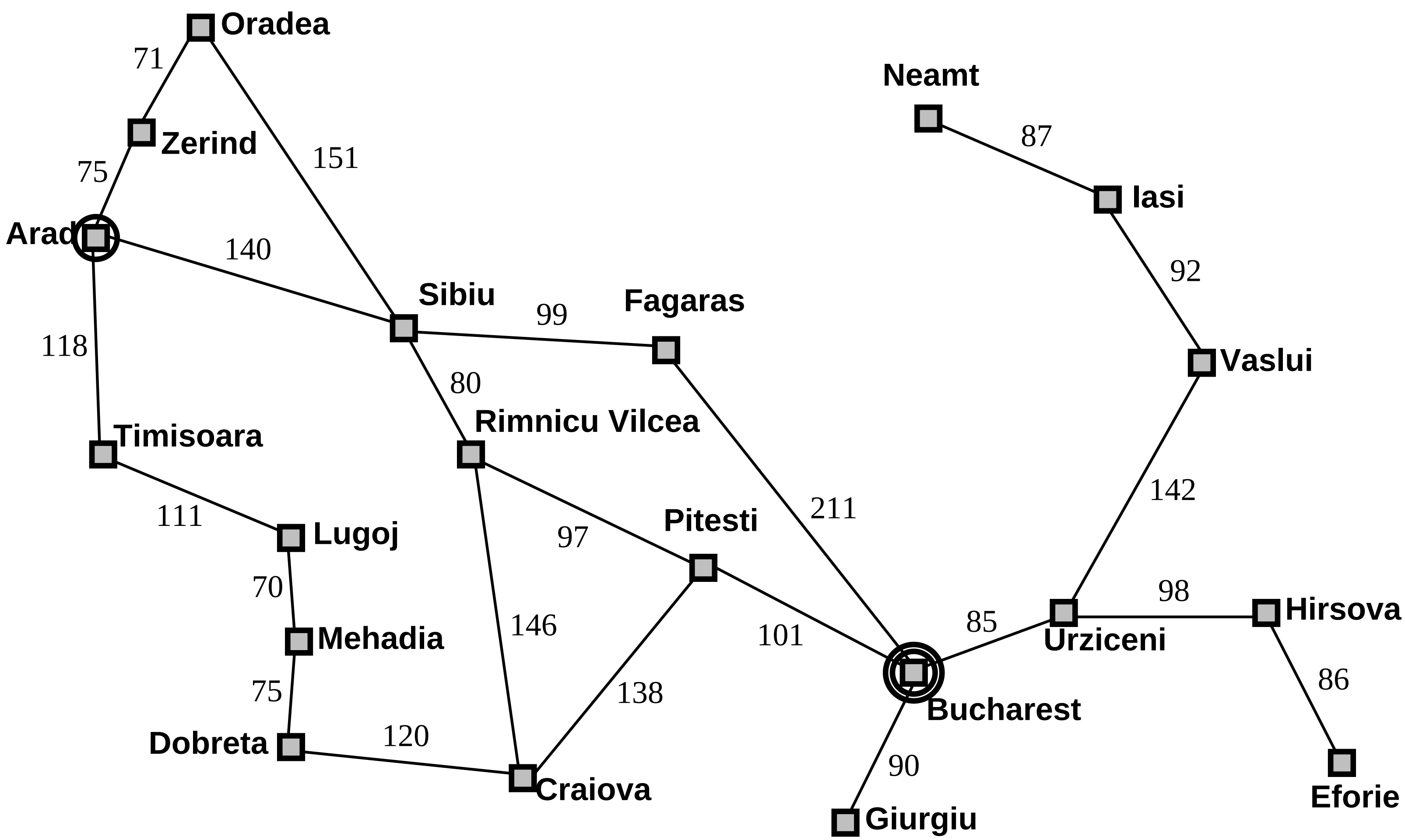
- *Offline* problem solving: develop the entire solution at the start, before you ever start to execute it
  - e.g., the Vacuum World examples on the last three slides

- **Focus of this chapter:** *offline* problem solving for *classical search problems* (i.e., deterministic, fully observable)

Example: Romania

Currently in Arad, Romania; flight leaves tomorrow from Bucharest

states = cities; *actions* = drive between cities; *goal* = be in Bucharest



Selecting a state space

◇ Real world is absurdly complex

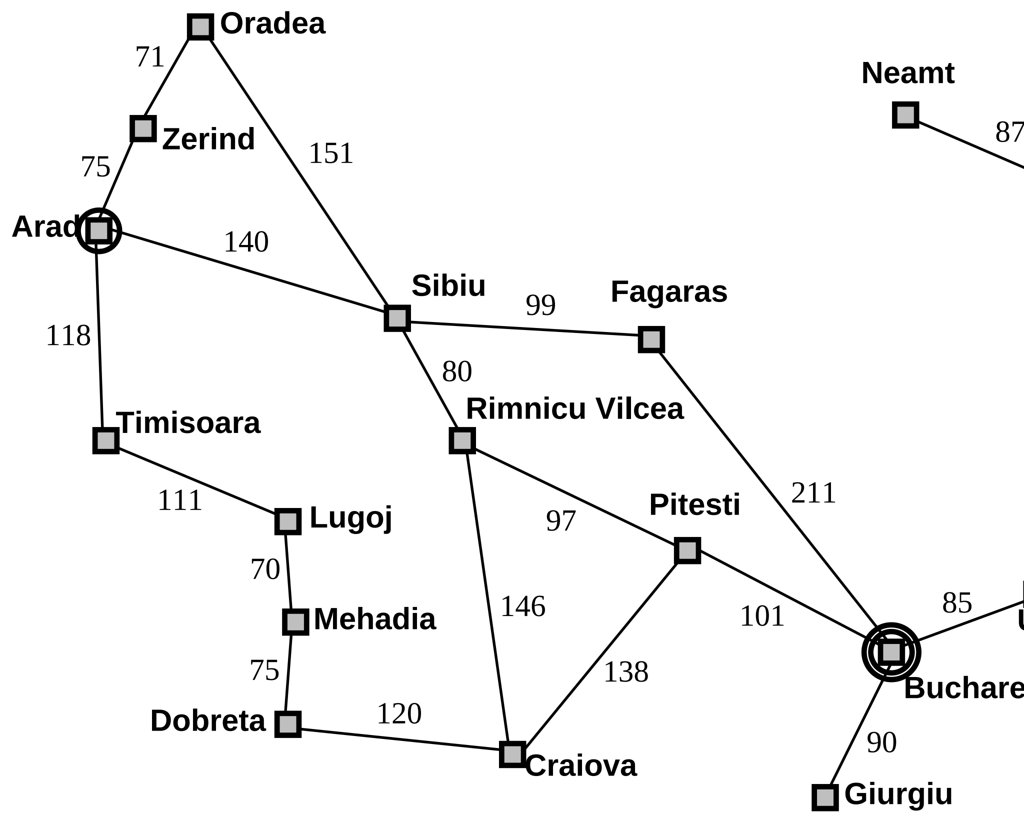
- state space is an **abstraction**

◇ *Abstract state* = set of real states

- E.g., in-Arad includes many locations

◇ *Abstract action* = complex combination of real actions

- E.g., goto-Zerind may include routes, detours, rest stops, etc.
- For guaranteed realizability, it must get you to Zerind no matter where you are in Arad

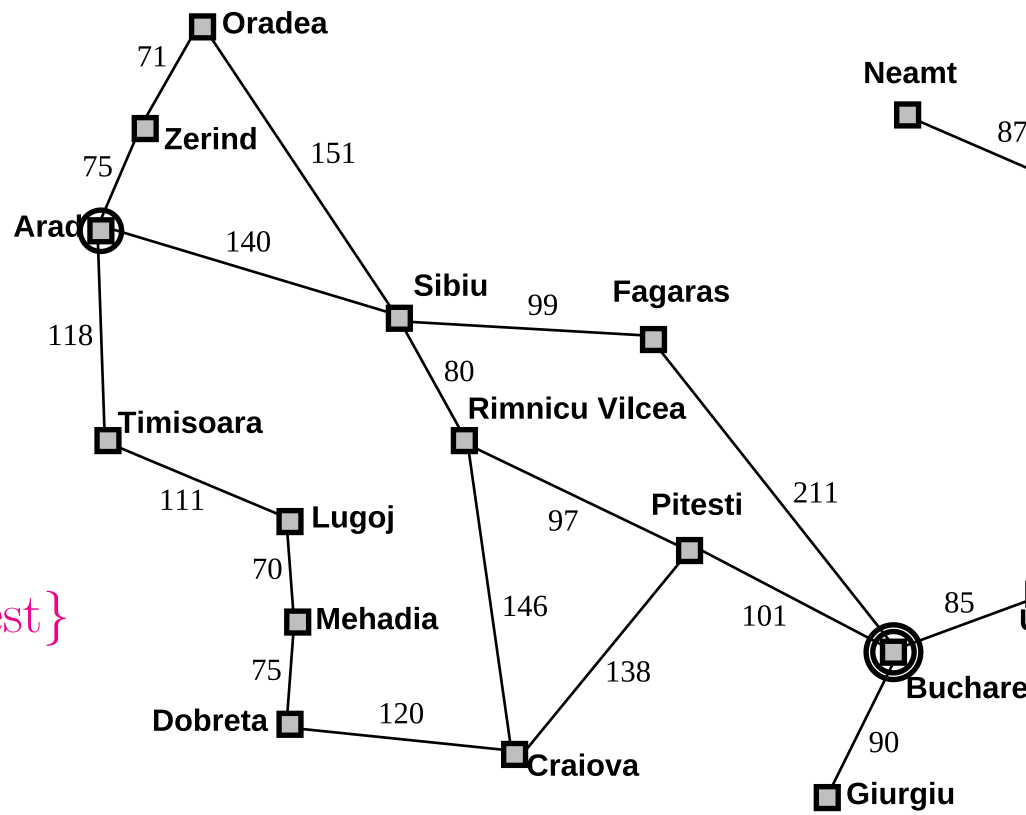


◇ *Abstract solution* = sequence of abstract actions
It represents a set of real paths that are solutions in the real world

Formulation of classical search problems

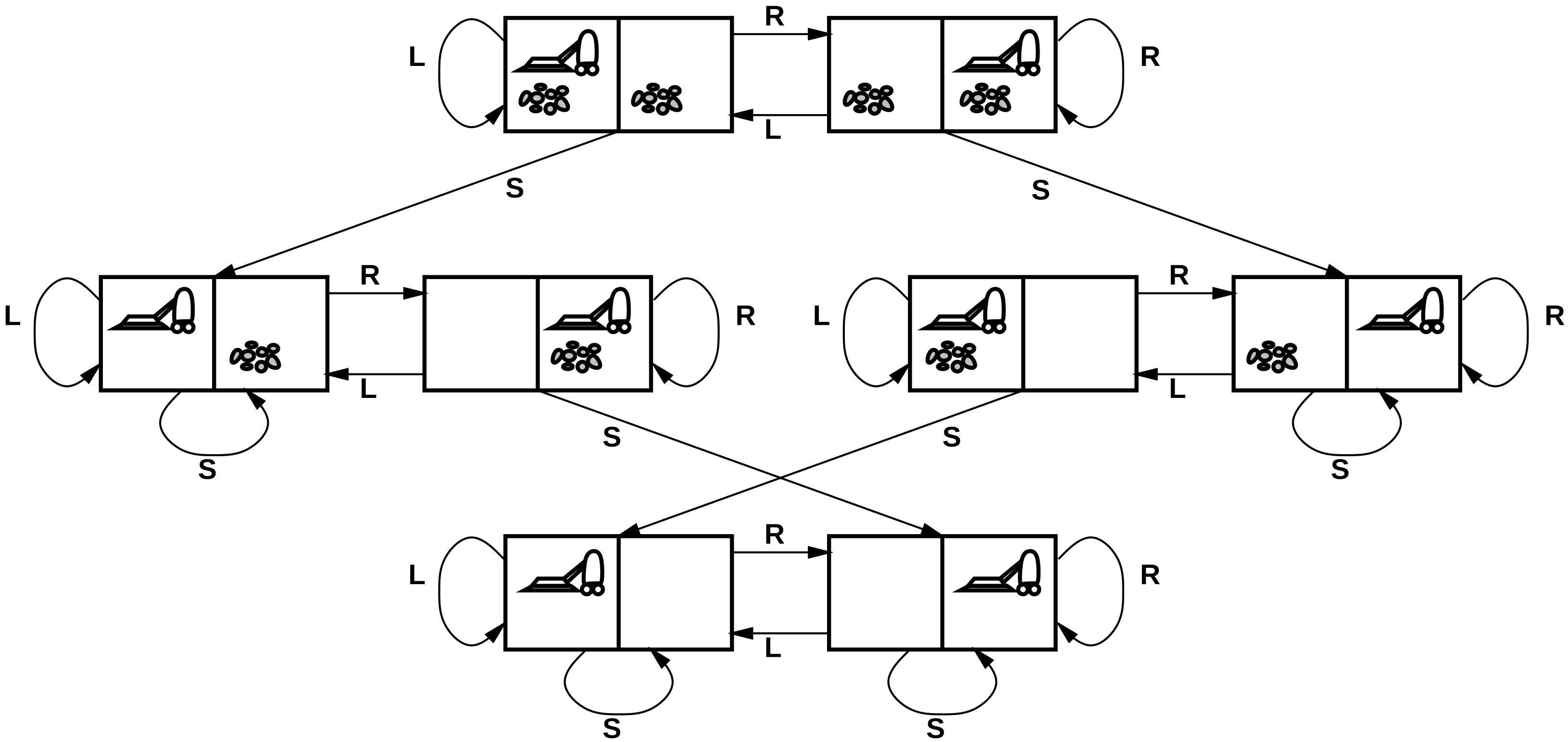
◇ A search problem includes:

- *initial state* s_0, e.g., at-Arad
- *set of actions*, e.g.,
$A = \{\text{goto-Zerind}, \ldots\}$
- *state-transition function* $\gamma(s, a)$
e.g., γ(at-Arad, goto-Zerind)
 = at-Zerind
- *goal test*: either *explicit*, e.g.,
 set of goal states = {at-Bucharest}
 or *implicit*, e.g., $NoDirt(s)$
- *path cost*
  - ◇ additive, e.g., sum of distances, number of actions, etc.
  - ◇ $c(s, a)$ is the *step cost*, assumed to be ≥ 0



◇ *solution*: sequence of actions from the initial state to a goal state

Example: vacuum world, no exogenous events



states: dirt and robot locations (ignore dirt *amounts*, etc.)
actions: $Left$, $Right$, $Suck$, $NoOp$
goal test: no dirt
path cost: 1 per action (0 for $NoOp$)

Example: sliding-tile puzzles

$n \times n$ frame, $n^2 - 1$ movable tiles. Slide the tiles to change their positions.

$n = 3$: the 8-puzzle

7	2	4
5		6
8	3	1

a starting state

1	2	3
4	5	6
7	8	

goal state

$n = 4$: the 15-puzzle

a starting state

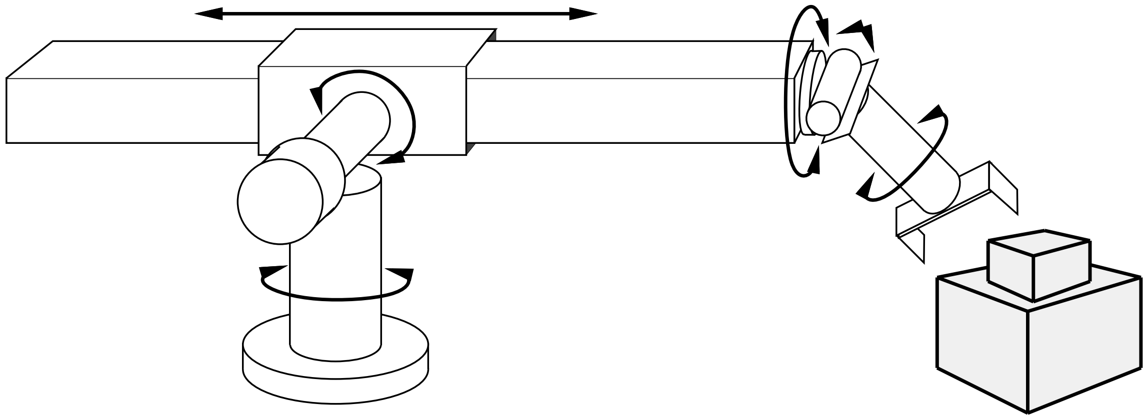
goal state

- *states*: integer locations of tiles (ignore intermediate positions)
- *actions*: move tiles left, right, up, down (ignore unjamming etc.)
- *goal test* = goal state (shown)
- *step cost* = 1 per move, so *path cost* = number of moves

◇ In this family of puzzles, finding **optimal** solutions is NP-hard

- Much easier if we don't care whether the solution is optimal

Example: robotic assembly

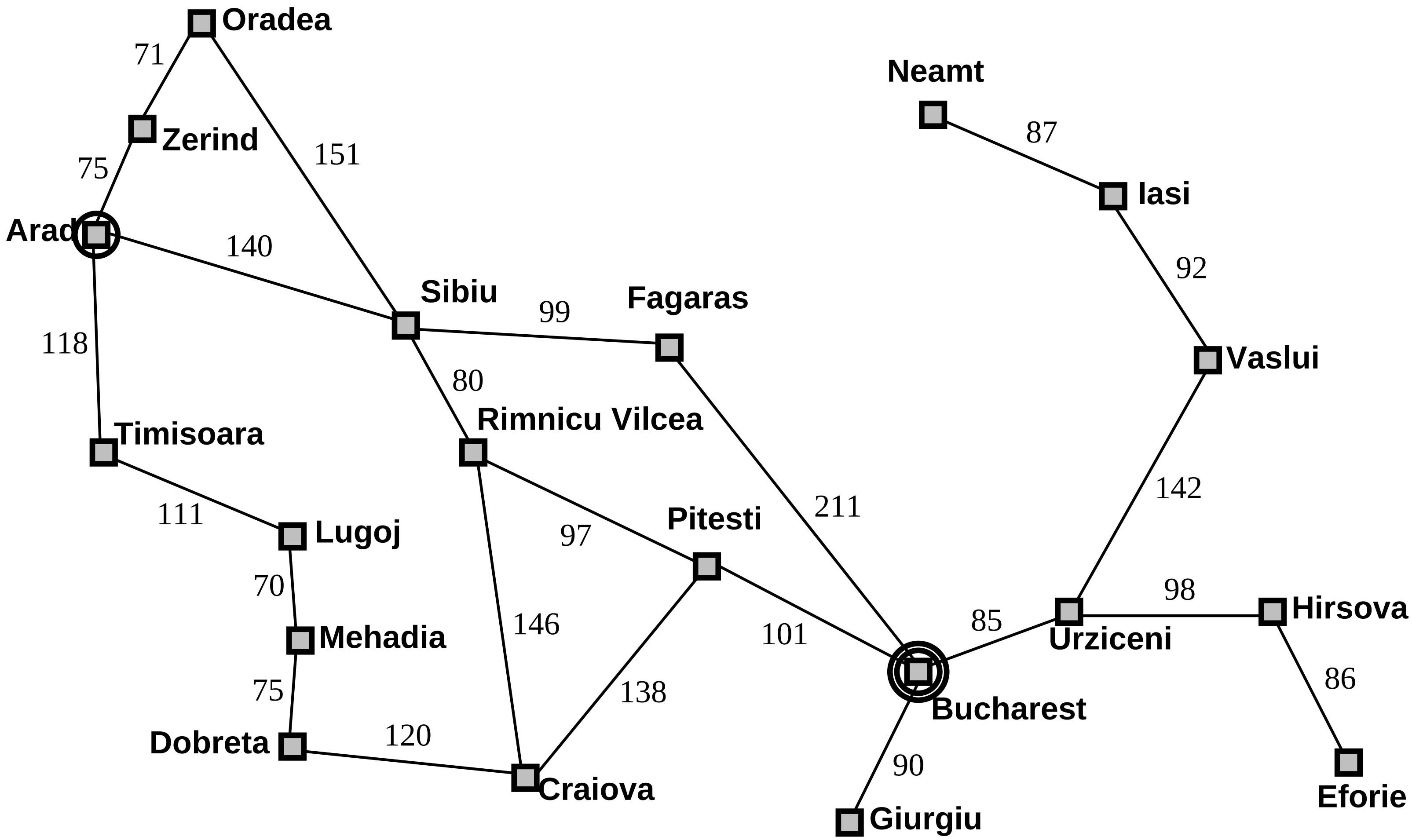


- *states*: real-valued coordinates of robot joint angles, and parts of the object to be assembled
- *actions*: continuous motions of robot joints
- *goal test*: complete assembly
- *path cost*: time to execute

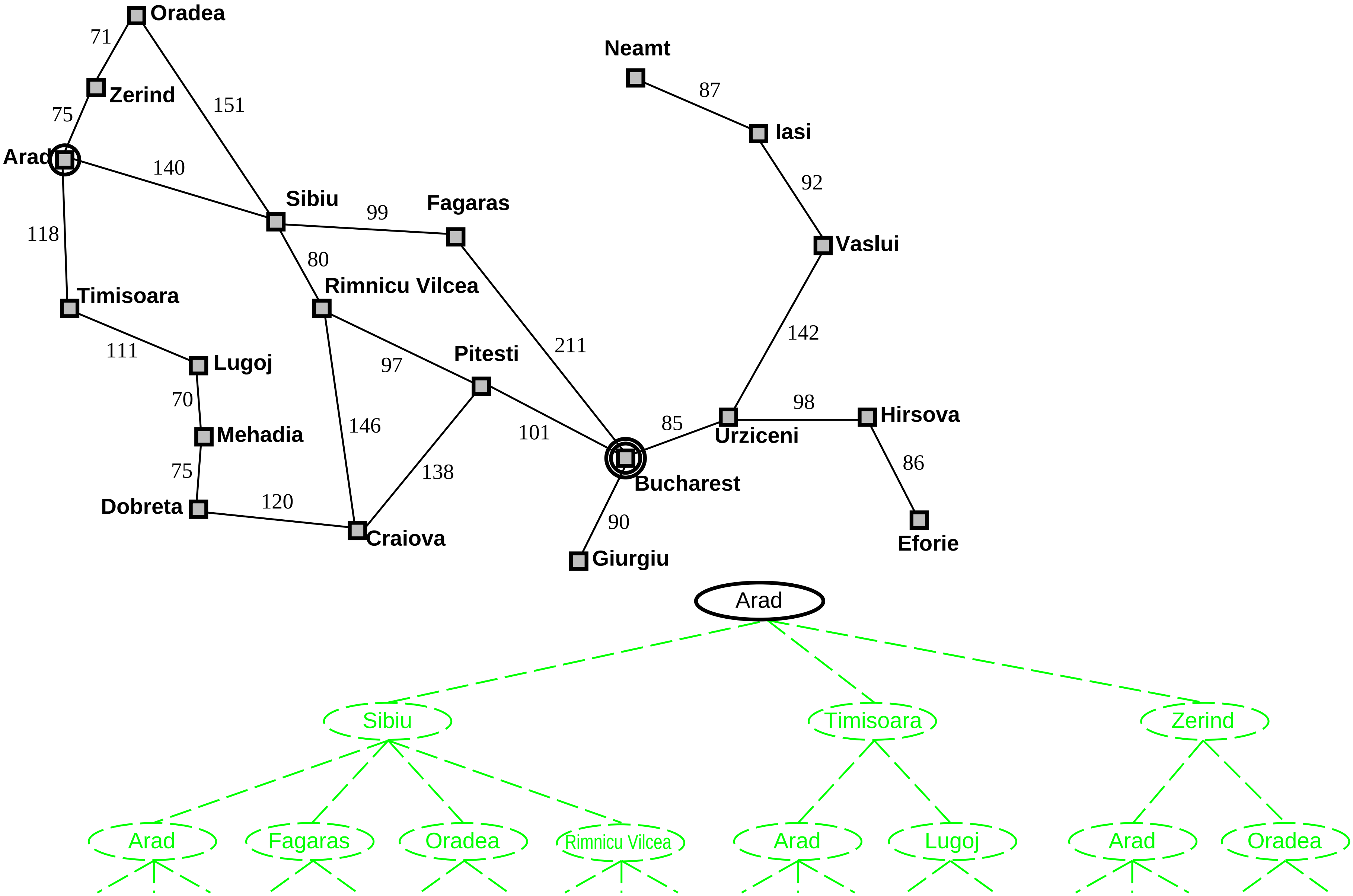
Tree search example

Currently in Arad, Romania; flight leaves tomorrow from Bucharest

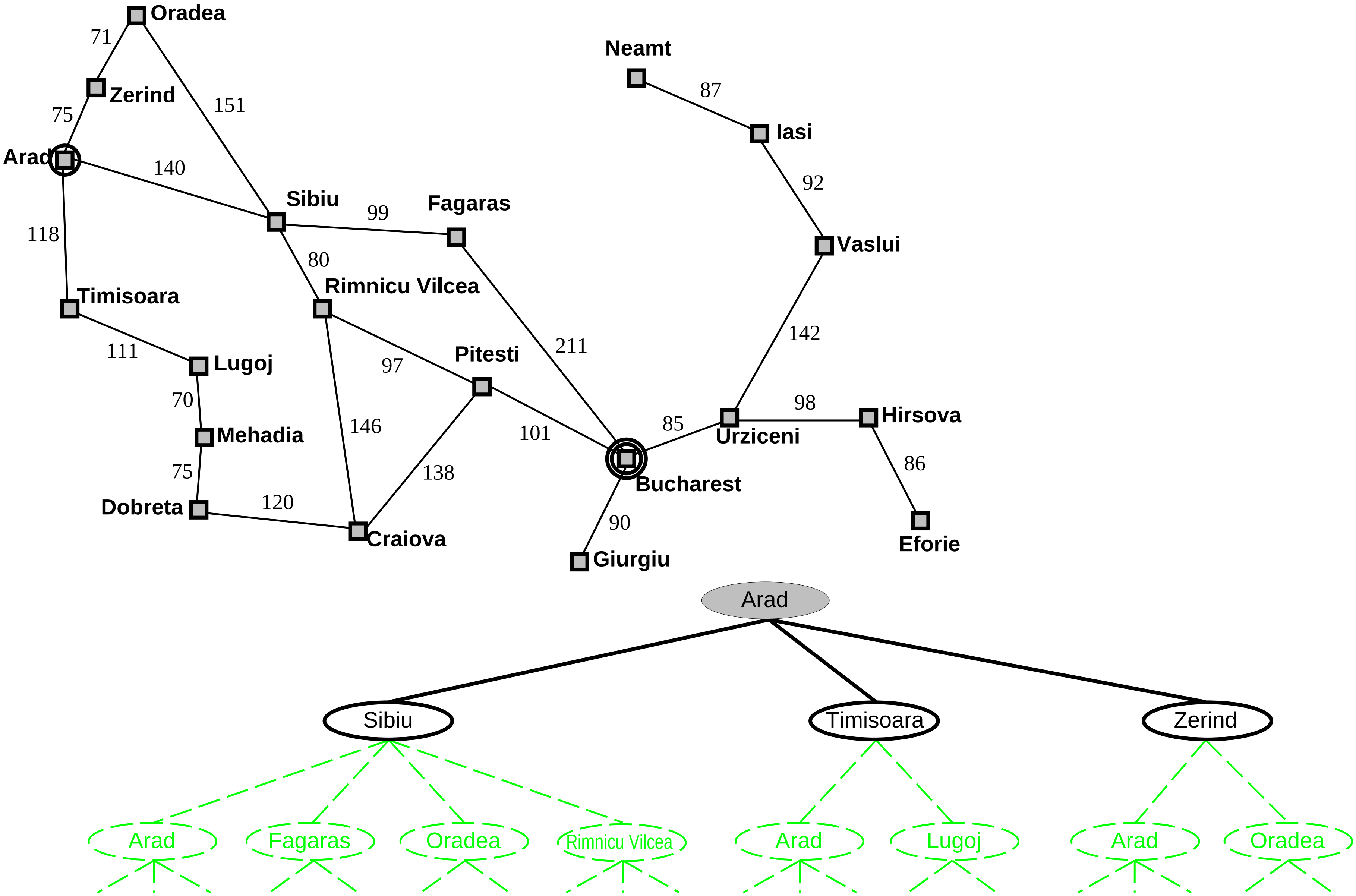
states = cities; *actions* = drive between cities; *goal* = be in Bucharest



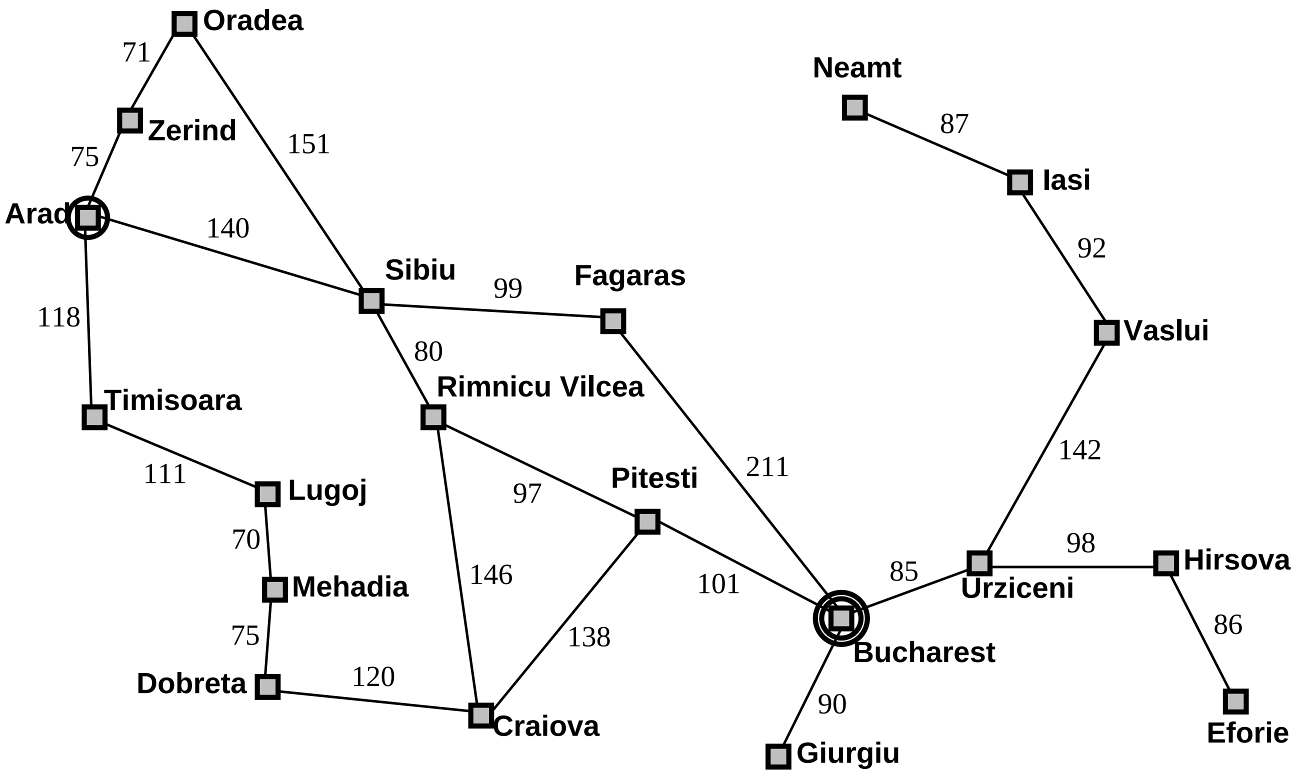
Tree search example

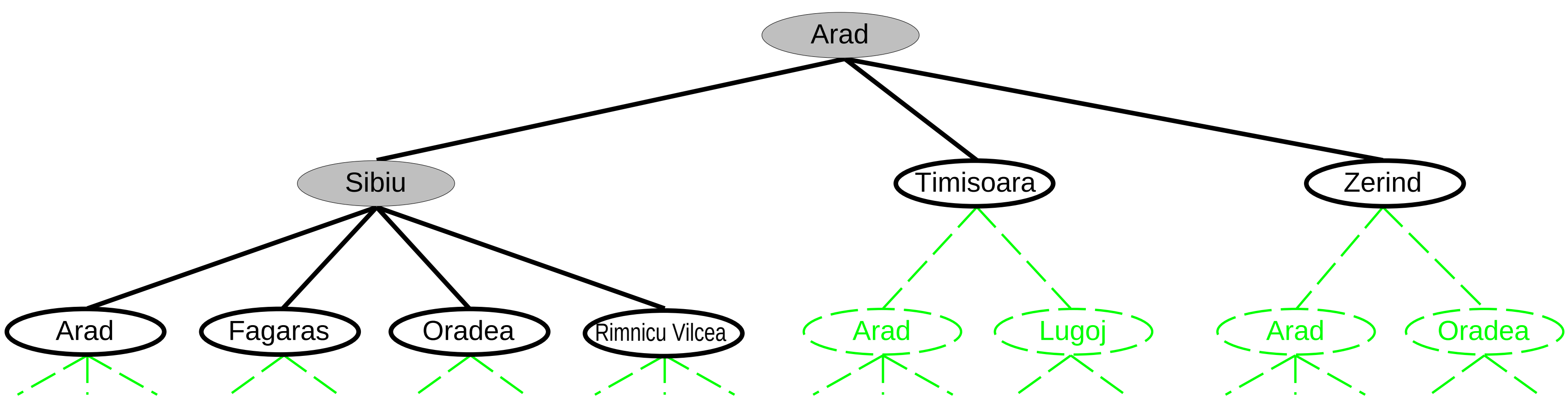


Tree search example



Tree search example

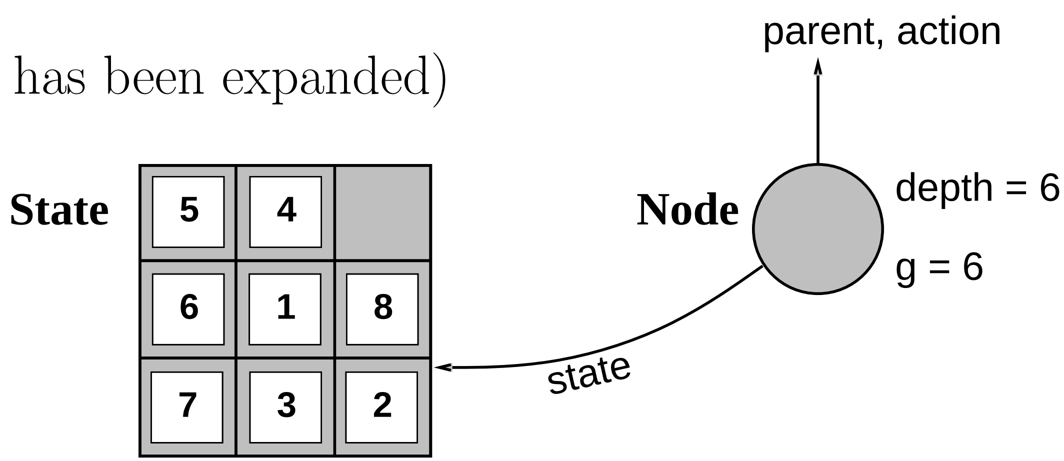




Implementation: states vs. nodes

◇ *Node*: a data structure that's part of a search tree. Includes

- a *state*
- a *parent*
- *children* (if the node has been expanded)
- a *depth*
- a *path cost*



◇ *State*: representation of a physical configuration

- doesn't have parents, children, depth, or path cost

◇ *Expanding* a node x:

- For each of x's children, create a new node and fill in the fields

Eager vs. cautious tree search

```
function EAGER-TREE-SEARCH(problem)          # my version
   frontier ← list that contains a node for problem's initial state
   loop
      if frontier is empty then return Failure
      choose and remove a node x from frontier
      for each node y in x's expansion
         if STATE[y] is a goal then return the corresponding solution
         else add y to frontier

function CAUTIOUS-TREE-SEARCH(problem)       # like TREE-SEARCH in the book
   frontier ← list that contains a node for problem's initial state
   loop
      if frontier is empty then return Failure
      choose and remove a node x from frontier
      if x contains a goal state then return the corresponding solution
      else expand x and add the new nodes to frontier
```

◇ Similarities and differences?

Eager vs. cautious tree search

```
function EAGER-TREE-SEARCH(problem)          # my version
   frontier ← list that contains a node for problem's initial state
   loop
      if frontier is empty then return Failure
      choose and remove a node x from frontier
      for each node y in x's expansion
         if STATE[y] is a goal then return the corresponding solution
         else add y to frontier

function CAUTIOUS-TREE-SEARCH(problem)       # like TREE-SEARCH in the book
   frontier ← list that contains a node for problem's initial state
   loop
      if frontier is empty then return Failure
      choose and remove a node x from frontier
      if x contains a goal state then return the corresponding solution
      else expand x and add the new nodes to frontier
```

◇ EAGER returns solution immediately – generates fewer nodes

◇ CAUTIOUS waits until node is chosen – necessary to find optimal solution

Search strategies

- ◇ A search strategy is defined by picking the **order of node expansion**
- ◇ Ways to evaluate a strategy:
  - *completeness*: does it always find a solution if one exists?
  - *optimality*: does it always find a least-cost solution?
  - *time complexity*: number of nodes generated/expanded
  - *space complexity*: maximum number of nodes in memory
- ◇ Time and space complexity are measured in terms of
  - $b =$ maximum branching factor of the search tree
    - ◇ We'll assume b is finite
  - $d =$ depth of the least-cost solution (or ∞ if there's no solution)
  - $m =$ maximum depth of the state space (may be ∞)

Uninformed search strategies

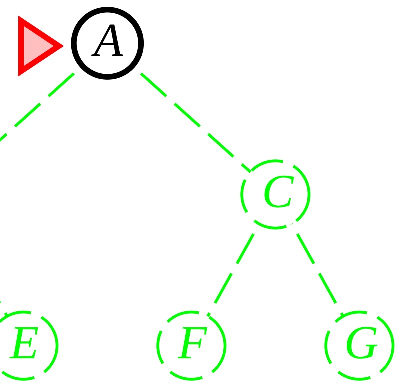
♢ *Uninformed* strategies

  ⋄ use only the information available in the problem definition

- Breadth-first search
- Depth-first search
- Uniform-cost search
- Limited-depth search
- Iterative deepening search

Breadth-first search

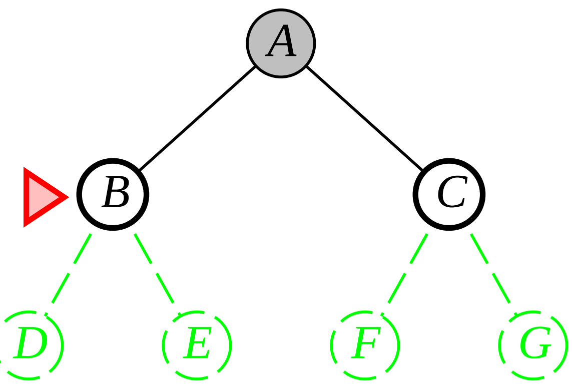
- Expand shallowest unexpanded node
- **Implementation**:
  frontier is a FIFO queue, i.e., new successors go at end



Breadth-first search

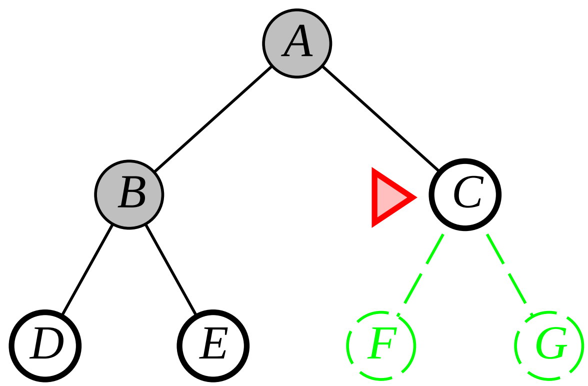
◊ Expand shallowest unexpanded node

◊ **Implementation**:
frontier is a FIFO queue, i.e., new successors go at end



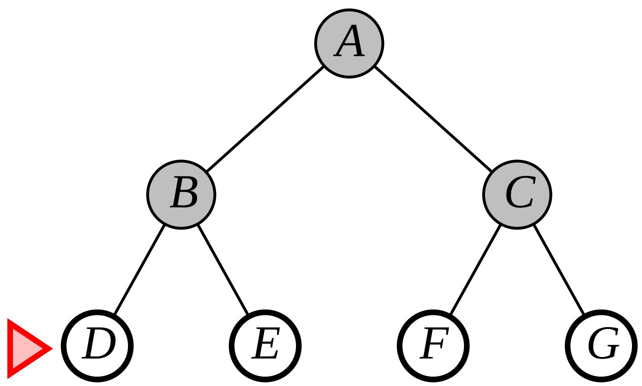
Breadth-first search

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



Breadth-first search

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



Properties of breadth-first search

◇ *Complete??*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of breadth-first search

◇ *Complete?* Yes

◇ *Time?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of breadth-first search

◊ *Complete?* Yes

◊ *Time?* $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^d)$

◊ *Space?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of breadth-first search

◊ *Complete?* Yes

◊ *Time?* $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^d)$

◊ *Space?* $O(b^d)$ (keeps every node in memory)

- If we run for 12 hours and generate nodes at 200 MB/sec, the space requirement is 8.64 TB

◊ *Optimal solutions?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of breadth-first search

◊ *Complete?* Yes

◊ *Time?* $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^d)$

◊ *Space?* $O(b^d)$ (keeps every node in memory)

- If we run for 12 hours and generate nodes at 200 MB/sec, the space requirement is 8.64 TB

◊ *Optimal solutions?*

- Yes if cost $= k$ per step where k is constant; otherwise no

---

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Breadth-first search

```
function EAGER-TREE-SEARCH(problem)          # my version
    frontier ← list that contains a node for problem's initial state
    loop
        if frontier is empty then return Failure
        choose and remove a node x from frontier
        for each node y in x's expansion
            if STATE[y] is a goal then return the corresponding solution
            else add y to frontier

function CAUTIOUS-TREE-SEARCH(problem)       # like TREE-SEARCH in the book
    frontier ← list that contains a node for problem's initial state
    loop
        if frontier is empty then return Failure
        choose and remove a node x from frontier
        if x contains a goal state then return the corresponding solution
        else expand x and add the new nodes to frontier
```

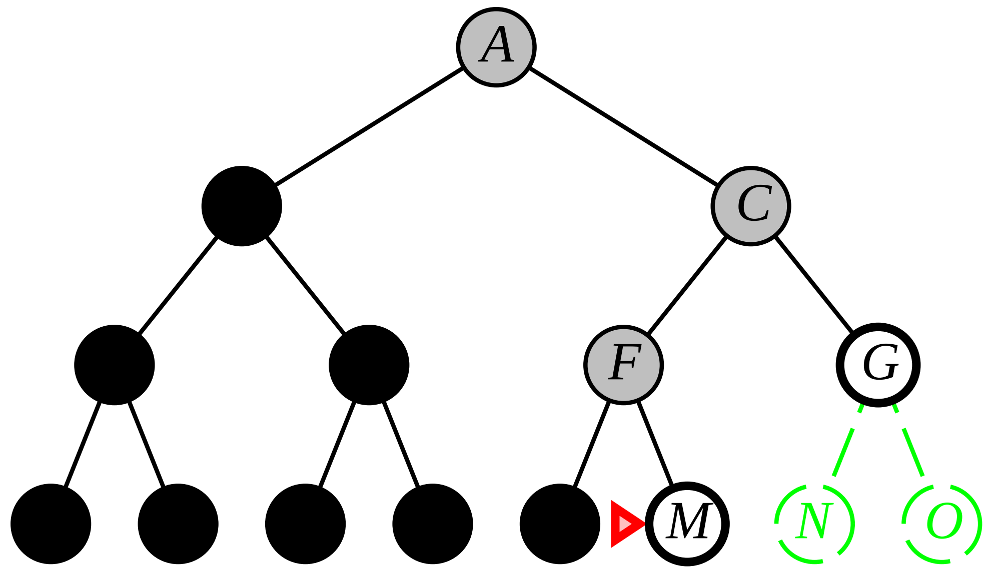
◇ Which is better for breadth-first search?

Comparison

◇ Every edge has cost 10, except for the following two:

  ⋄ (G, N) and (G, O) both cost 5

- M is a goal node of cost 30
- N is a goal node of cost 25

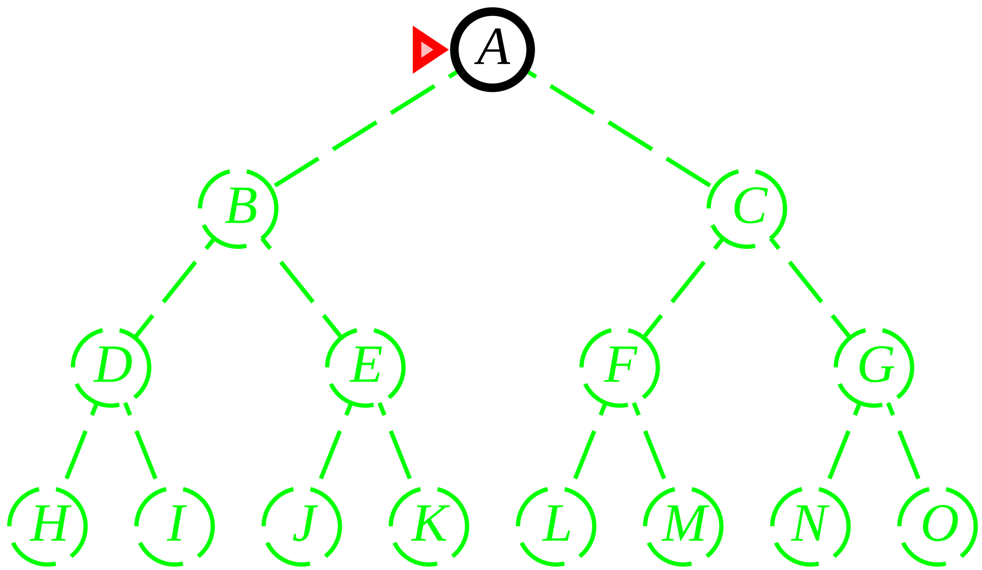


◇ For breadth-first search

- What solutions do EAGER and CAUTIOUS return?
- How many nodes do they generate?

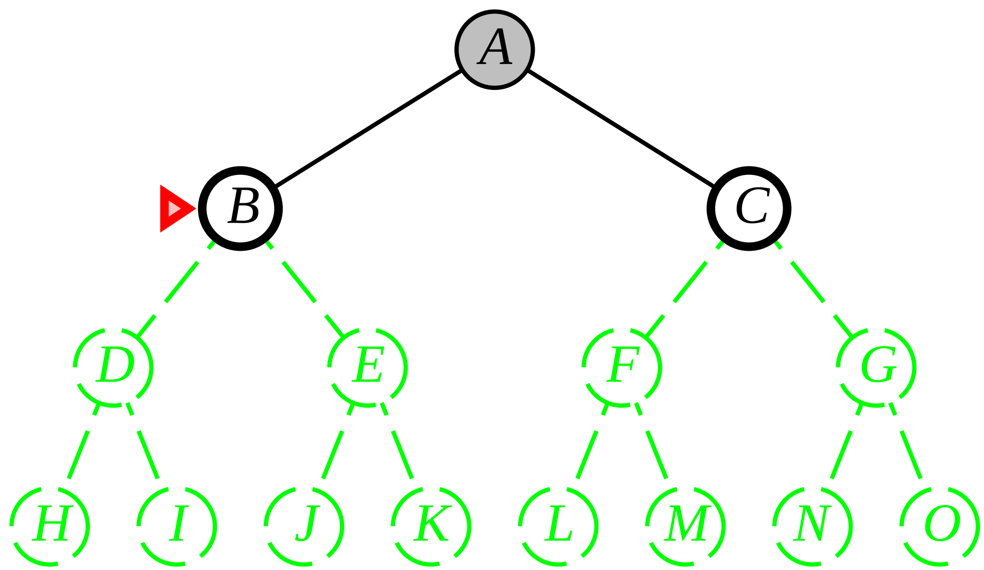
Depth-first search

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



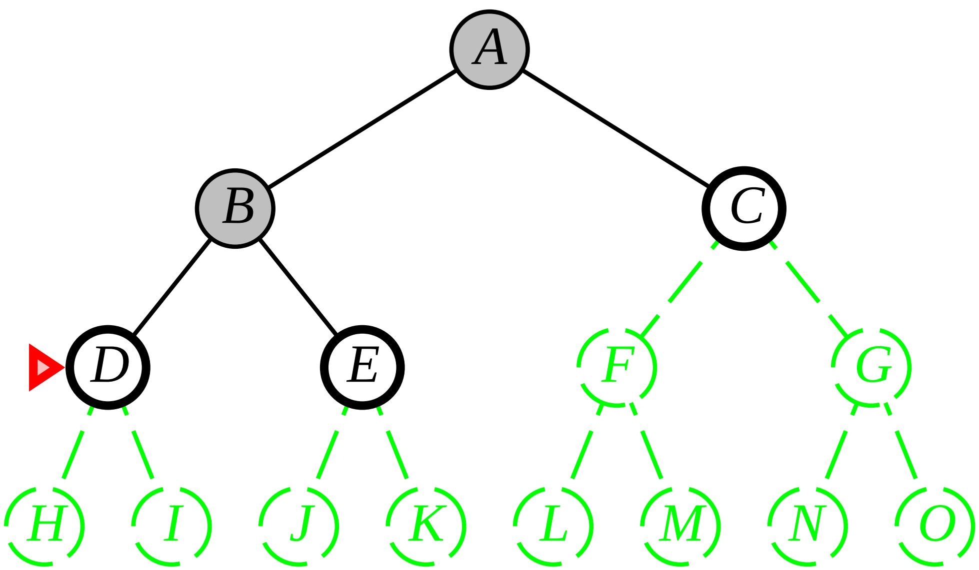
Depth-first search

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



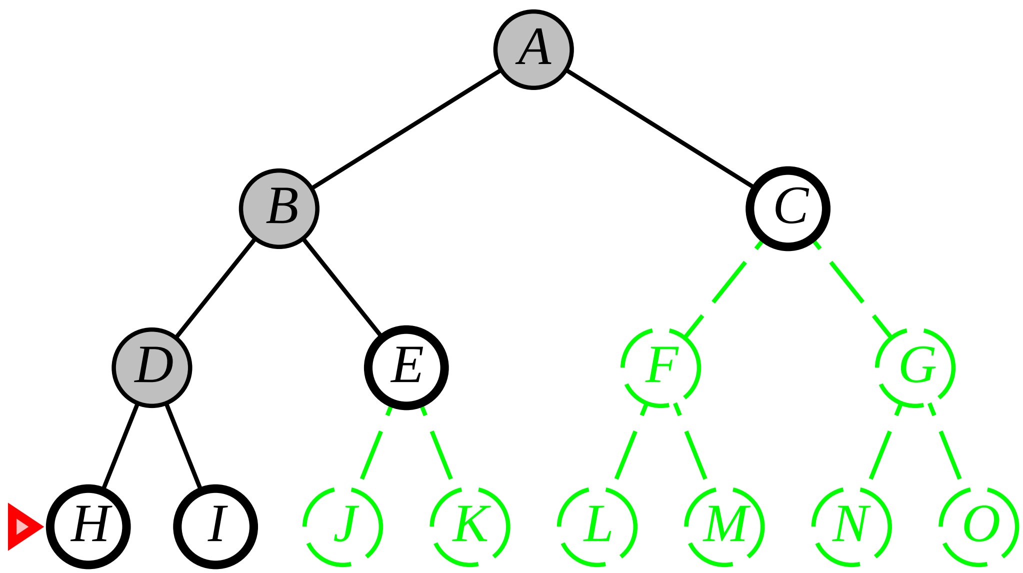
Depth-first search

- ♢ Expand deepest unexpanded node
- ♢ **Implementation**:
  frontier = LIFO queue, i.e., put successors at front



Depth-first search

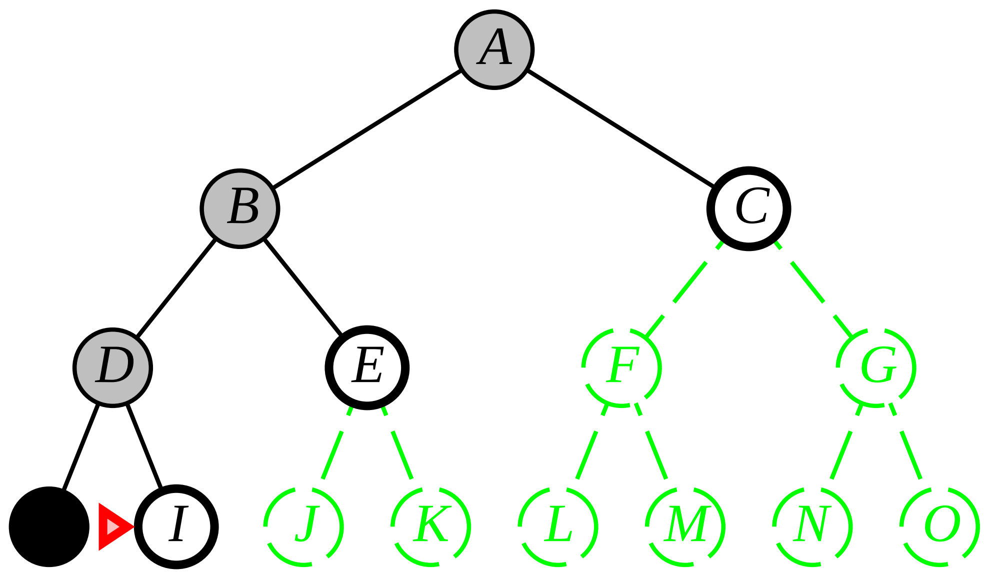
- ♢ Expand deepest unexpanded node
- ♢ **Implementation**:
  frontier = LIFO queue, i.e., put successors at front



Depth-first search

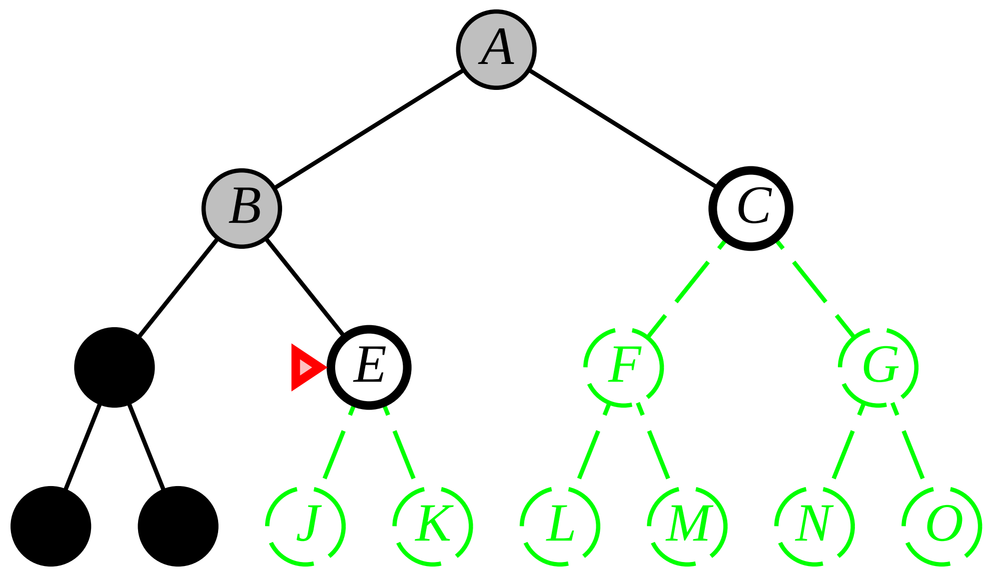
◊ Expand deepest unexpanded node

◊ **Implementation**:
frontier = LIFO queue, i.e., put successors at front



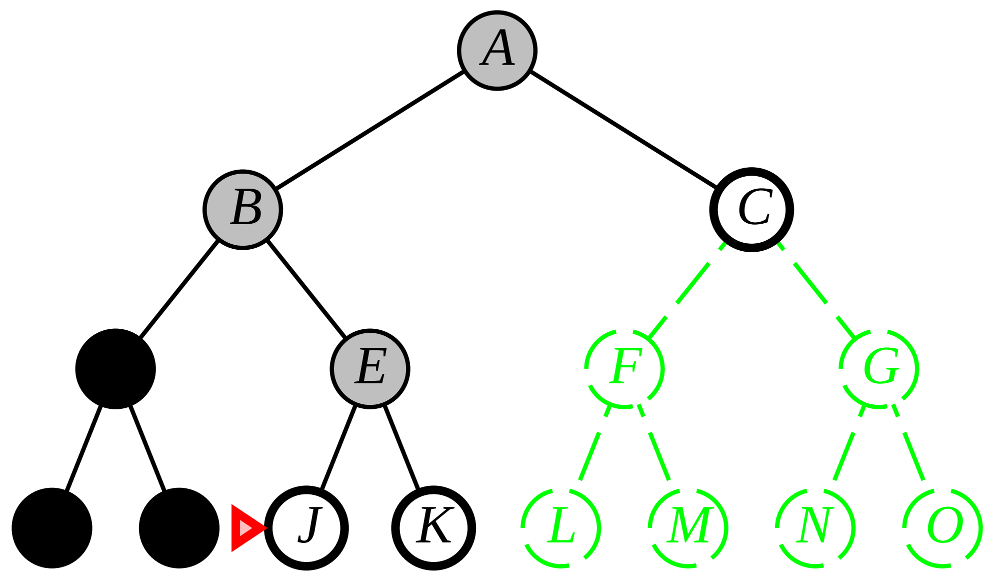
Depth-first search

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



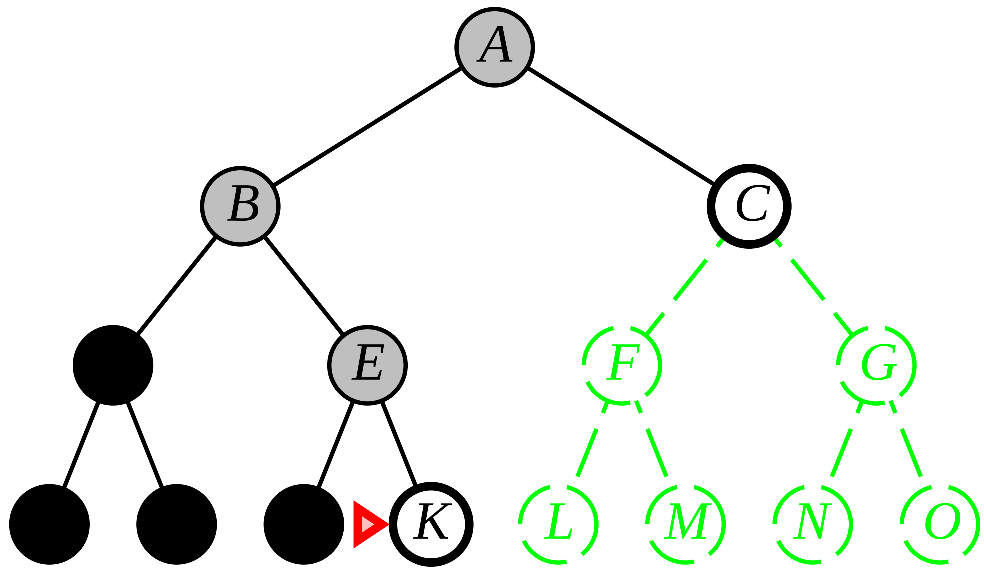
Depth-first search

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



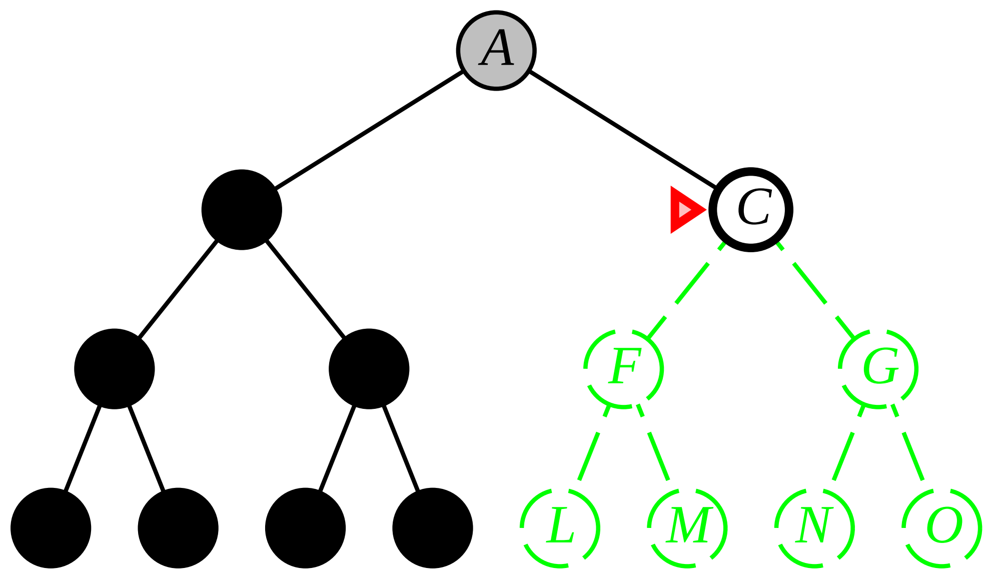
Depth-first search

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



Depth-first search

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



Properties of depth-first search

◇ *Complete?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of depth-first search

◇ *Complete?*

- No in infinite-depth spaces
- Yes in finite spaces, if we do loop-checking:
  - ◇ Don't generate states that are already on the current path

◇ *Time?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of depth-first search

◊ *Complete?*

- No in infinite-depth spaces
- Yes in finite spaces, if we do loop-checking:
  - Don't generate states that are already on the current path

◊ *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?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of depth-first search

◇ *Complete?*
- No in infinite-depth spaces
- Yes in finite spaces, if we do loop-checking:
  - ◇ Don't generate states that are already on the current path

◇ *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 solutions?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Properties of depth-first search

◇ *Complete?*

- No in infinite-depth spaces
- Yes in finite spaces, if we do loop-checking:
  - ◇ Don't generate states that are already on the current path

◇ *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 solutions?* Not unless it's lucky

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Eager vs. cautious tree search

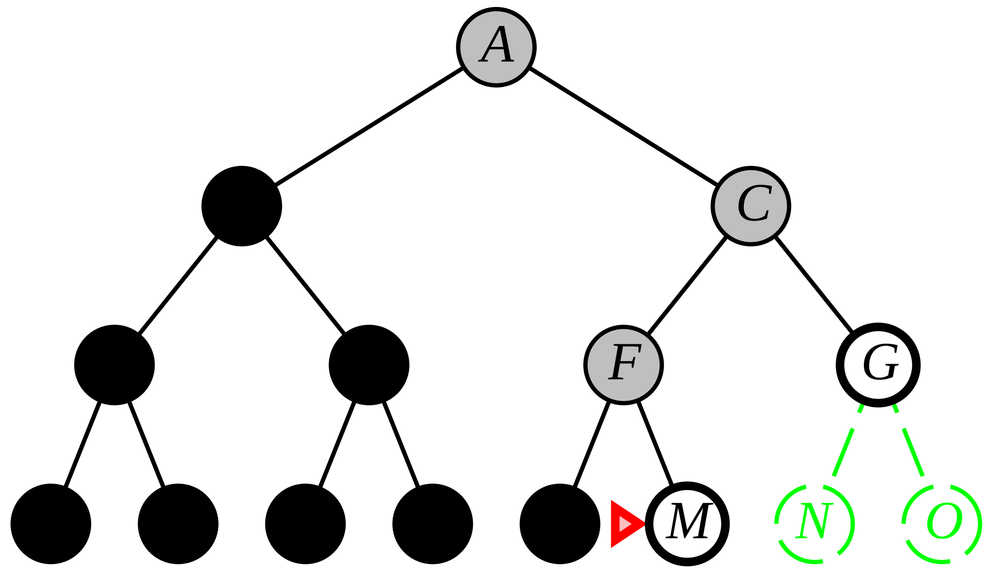
function EAGER-TREE-SEARCH(*problem*) # my version
    frontier ← list that contains a node for *problem*'s initial state
    loop
        if *frontier* is empty **then return** Failure
        choose and remove a node x from *frontier*
        for each node y in x's expansion
            if STATE[y] is a goal **then return** the corresponding solution
            else add y to *frontier*

function CAUTIOUS-TREE-SEARCH(*problem*) # like TREE-SEARCH in the book
    frontier ← list that contains a node for *problem*'s initial state
    loop
        if *frontier* is empty **then return** Failure
        choose and remove a node x from *frontier*
        if x contains a goal state **then return** the corresponding solution
        else expand x and add the new nodes to *frontier*

◇ Which is better for depth-first search?

Comparison

- Every edge has cost 10, except for the following two:
  - (G, N) and (G, O) both cost 5
  - M is a goal node; path cost = 30
  - N is a goal node; path cost = 25



- For breadth-first search
  - What solutions do EAGER and CAUTIOUS tree search return?
  - How many nodes do they generate?

Eager vs. cautious tree search

```
function EAGER-TREE-SEARCH(problem)        # my version
   frontier ← list that contains a node for problem's initial state
   loop
      if frontier is empty then return Failure
      choose and remove a node x from frontier
      for each node y in x's expansion
         if STATE[y] is a goal then return the corresponding solution
         else add y to frontier

function CAUTIOUS-TREE-SEARCH(problem)     # like TREE-SEARCH in the book
   frontier ← list that contains a node for problem's initial state
   loop
      if frontier is empty then return Failure
      choose and remove a node x from frontier
      if x contains a goal state then return the corresponding solution
      else expand x and add the new nodes to frontier
```

◇ Where would we put loop-checking?

Uniform-cost search

◇ Expand least-cost unexpanded node

◇ **Implementation**: *frontier* = queue ordered by path cost, lowest first
Equivalent to breadth-first if step costs all equal

◇ *Complete?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Uniform-cost search

- Expand least-cost unexpanded node
- **Implementation**: *frontier* = queue ordered by path cost, lowest first
  Equivalent to breadth-first if step costs all equal
- *Complete?* Yes, if $\exists\ \epsilon > 0$ such that step cost $\geq \epsilon$
- *Time?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Uniform-cost search

◇ Expand least-cost unexpanded node

◇ **Implementation**: *frontier* = queue ordered by path cost, lowest first
Equivalent to breadth-first if step costs all equal

◇ *Complete?* Yes, if $\exists\ \epsilon > 0$ such that step cost $\geq \epsilon$

◇ *Time?* $|\{$nodes with $g \leq C^*\}| = O(b^{\lceil C^*/\epsilon \rceil})$, where

- C^* = cost of the optimal solution

◇ *Space?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Uniform-cost search

◊ Expand least-cost unexpanded node

◊ **Implementation**: *frontier* = queue ordered by path cost, lowest first
Equivalent to breadth-first if step costs all equal

◊ *Complete?* Yes, if $\exists\ \epsilon > 0$ such that step cost $\geq \epsilon$

◊ *Time?* $|\{\text{nodes with } g \leq C^*\}| = O(b^{\lceil C^*/\epsilon \rceil})$, where

- C^* = cost of the optimal solution

◊ *Space?* $|\{\text{nodes with } g \leq C^*\}| = O(b^{\lceil C^*/\epsilon \rceil})$

◊ *Optimal solutions?*

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Uniform-cost search

◇ Expand least-cost unexpanded node

◇ **Implementation**: *frontier* = queue ordered by path cost, lowest first
Equivalent to breadth-first if step costs all equal

◇ *Complete?* Yes, if $\exists\ \epsilon > 0$ such that step cost $\geq \epsilon$

◇ *Time?* $|\{$nodes with $g \leq C^*\}| = O(b^{\lceil C^*/\epsilon \rceil})$, where

- C^* = cost of the optimal solution

◇ *Space?* $|\{$nodes with $g \leq C^*\}| = O(b^{\lceil C^*/\epsilon \rceil})$

◇ *Optimal solutions?* Yes, if we use Cautious-Tree-Search

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

Eager vs. cautious tree search

```
function EAGER-TREE-SEARCH(problem)          # my version
   frontier ← list that contains a node for problem's initial state
   loop
       if frontier is empty then return Failure
       choose and remove a node x from frontier
       for each node y in x's expansion
           if STATE[y] is a goal then return the corresponding solution
           else add y to frontier

function CAUTIOUS-TREE-SEARCH(problem)       # like TREE-SEARCH in the book
   frontier ← list that contains a node for problem's initial state
   loop
       if frontier is empty then return Failure
       choose and remove a node x from frontier
       if x contains a goal state then return the corresponding solution
       else expand x and add the new nodes to frontier
```

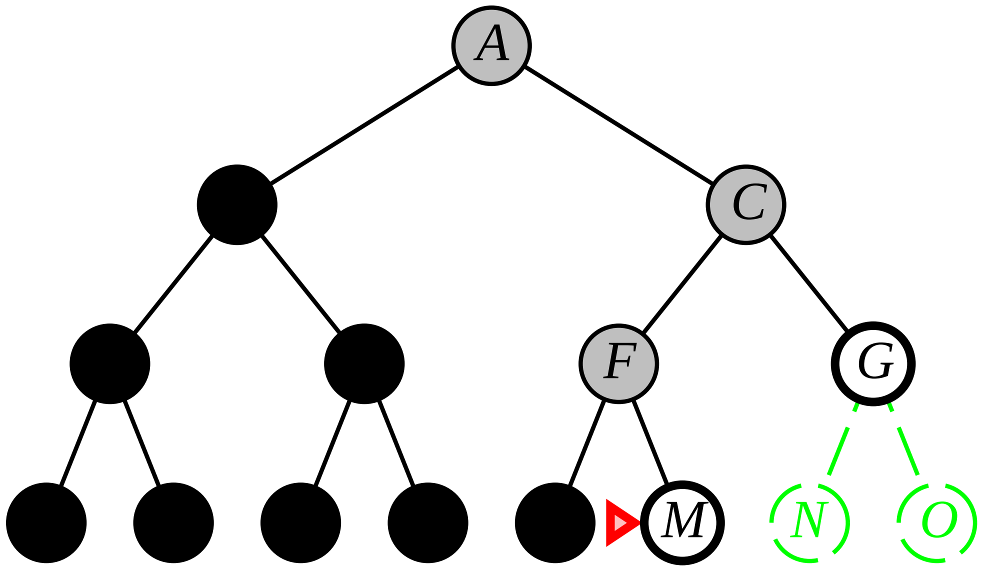
◊ Which is better for uniform-cost search?

Comparison

◇ Every edge has cost 10, except for the following two:

- ⋄ (G, N) and (G, O) both cost 5

- M is a goal node of cost 30
- N is a goal node of cost 25



◇ For uniform-cost search

- What solutions do EAGER and CAUTIOUS return?
- How many nodes do they generate?

Limited-depth search

- Depth-first search, backtrack at each node of depth = *limit* unless it's a solution
- Recursive implementation:

```
function LIMITED-DEPTH-SEARCH(node, problem, limit)
  if node contains a goal state then return the corresponding solution
  else if limit = 0 then return Cutoff
  else
      notfound ← Failure      /* what to return if we don't find a solution */
      for each y in EXPAND(node) do
          result ← LIMITED-DEPTH-SEARCH(y, problem, limit − 1)
          if result is a solution then return result
          else if result = Cutoff then notfound ← Cutoff
      return notfound
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem)
    node ← node for problem's initial state
    for limit ← 0 to ∞ do
        result ← LIMITED-DEPTH-SEARCH(node, problem, limit)
        if result ≠ Cutoff then return result
```

- Limited-depth search to depth 0,
- Limited-depth search to depth 1,
- Limited-depth search to depth 2,

  . . .
- Stop when you find a solution

function ITERATIVE-DEEPENING-SEARCH(*problem*)
  node $\leftarrow$ node for *problem*'s initial state
  for *limit* $\leftarrow$ 0 **to** ∞ **do**
    result $\leftarrow$ LIMITED-DEPTH-SEARCH(*node*, *problem*, *limit*)
    if *result* $\neq$ Cutoff **then return** *result*

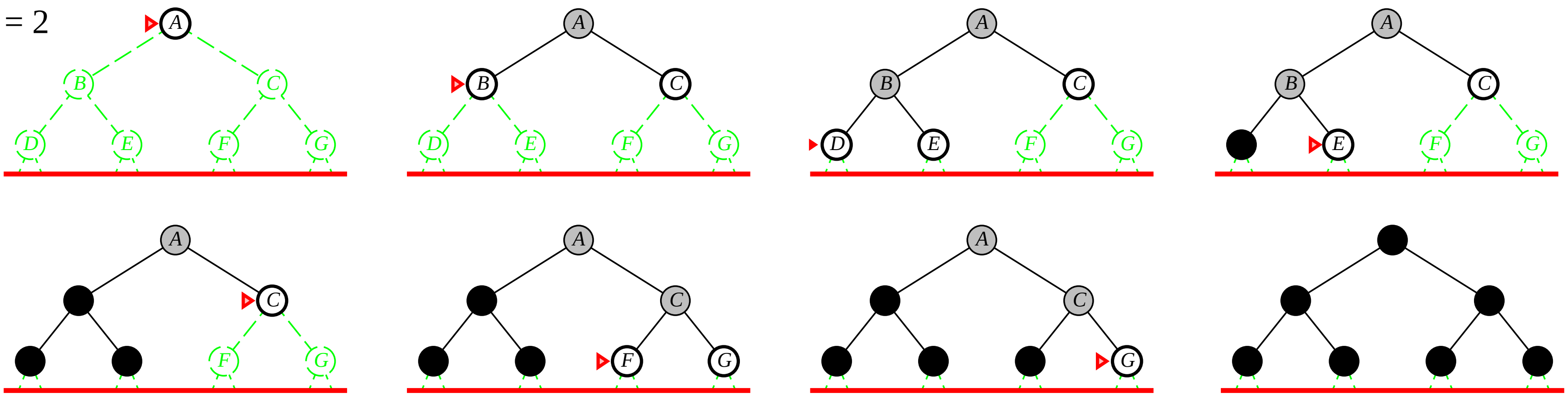
Limit = 0

```
function ITERATIVE-DEEPENING-SEARCH( problem)
    node ← node for problem's initial state
    for limit ← 0 to ∞ do
        result ← LIMITED-DEPTH-SEARCH(node, problem, limit)
        if result ≠ Cutoff then return result
```

Limit = 1

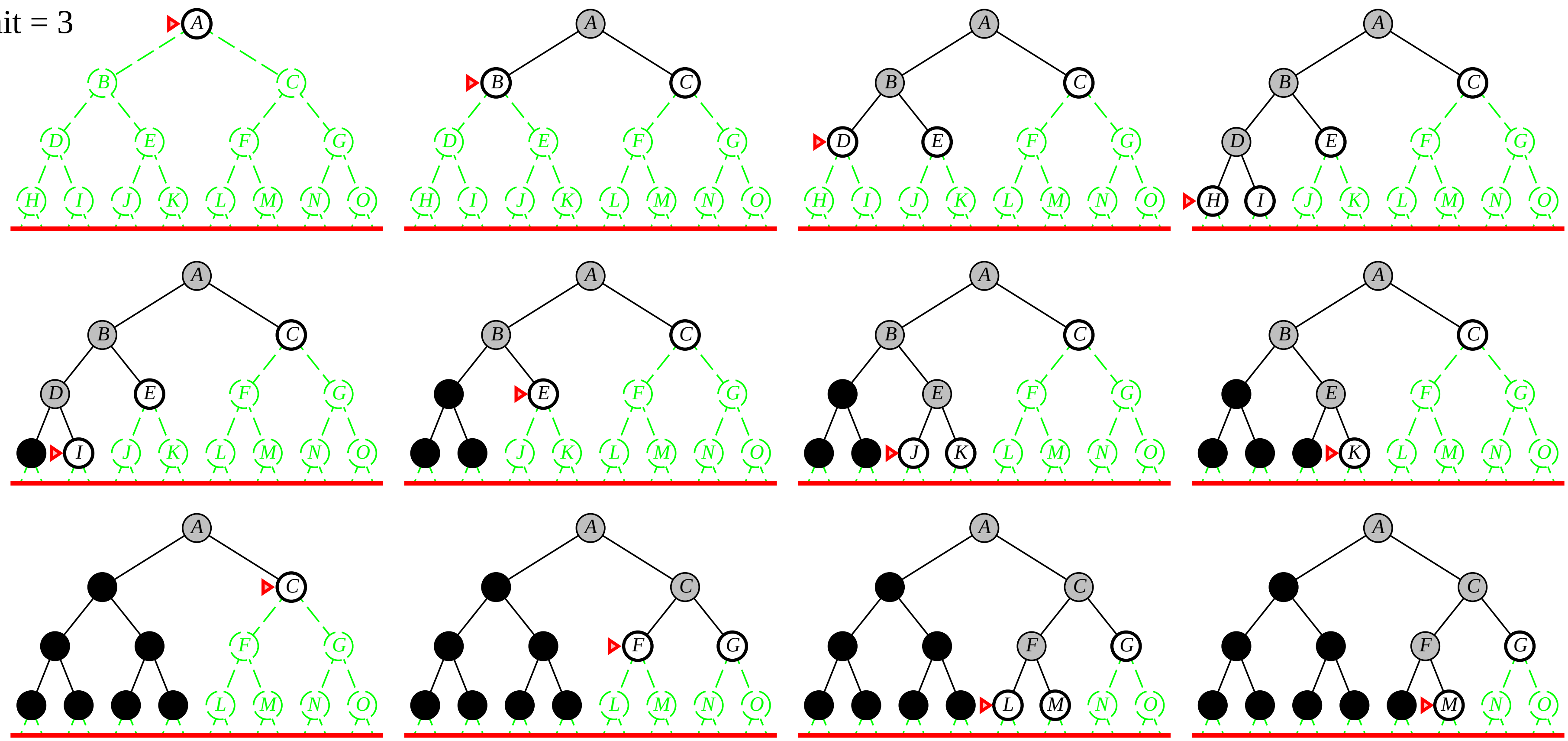
function ITERATIVE-DEEPENING-SEARCH(*problem*)
  node ← node for *problem*'s initial state
  for *limit* ← 0 **to** ∞ **do**
    result ← LIMITED-DEPTH-SEARCH(*node*, *problem*, *limit*)
    if *result* ≠ Cutoff **then return** *result*

Limit = 2



function Iterative-Deepening-Search(*problem*)
    node ← node for *problem*'s initial state
    for *limit* ← 0 **to** ∞ **do**
        result ← Limited-Depth-Search(*node*, *problem*, *limit*)
        if *result* $\neq$ Cutoff **then return** *result*

Limit = 3



Properties of iterative deepening search

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

◊ *Complete?*

Properties of iterative deepening search

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

◇ *Complete?* Yes

◇ *Time?*

Properties of iterative deepening search

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

◊ *Complete?* Yes

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

◊ *Space?*

Properties of iterative deepening search

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

◊ *Complete?* Yes

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

◊ *Space?* $O(bd)$

◊ *Optimal solutions?*

Properties of iterative deepening search

- $b =$ maximum branching factor of the search tree
- $d =$ depth of the least-cost solution
- $m =$ maximum depth of the state space (may be ∞)

◊ *Complete?* Yes

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

◊ *Space?* $O(bd)$

◊ *Optimal solutions?* Yes, if step cost $= 1$

- Can be modified to behave like uniform-cost search

Summary of algorithms

b = branching factor

C^* = cost of optimal solution, or ∞ if there's no solution

d = depth of shallowest solution, or ∞ if there's no solution

ϵ = smallest cost of each edge

l = cutoff depth for limited-depth search

m = depth of deepest node (may be ∞)

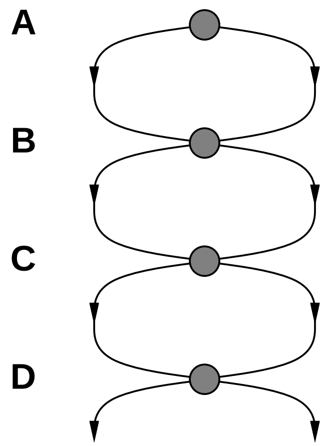
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes$^{(2)}$	No	Yes, if $l \geq d$	Yes
Time	b^d	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^d	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes$^{(1)}$	Yes	No	No	Yes$^{(1)}$

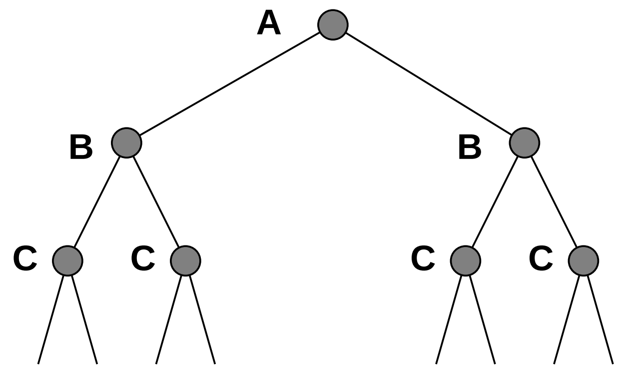
$^{(1)}$ if step costs are equal

$^{(2)}$ if $\epsilon > 0$

Repeated states

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





Graph search

```
function GRAPH-SEARCH(problem)
    frontier ← list that contains a node for problem's initial state
    explored ← empty set
    loop
        if frontier is empty then return Failure
        choose and remove a node x from frontier
        if STATE[x] is a goal then return the corresponding solution
        if STATE[x] is not in explored then
            add STATE[x] to explored
            expand x and add the new nodes to frontier
```

◇ Search strategy is implemented by the INSERTALL function

- breadth-first: insert new nodes at end of queue
- depth-first: insert new nodes at front of queue
- uniform-cost: keep queue ordered by cost

Summary

- ♢ 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
  - When $b \geq 2$, same big-O time as other uninformed algorithms
- ♢ Graph search can take exponentially less time than tree search
  - when the number of paths to a node is exponential in its depth
- ♢ Graph search can take exponentially more space than tree search
  - when the search space is treelike

Homework 1

- ◇ Due in one week
- ◇ 5 problems, 10 points per problem, 50 points total
  - 2.10
  - 3.6(a,b)
  - 3.9(a,c)
  - 3.15
  - 3.18

Python resources

- Documentation: http://docs.python.org
  - **Important**: in the left-hand column, click on Python 3.2 (stable)
- If you don't know Python already, read the Tutorial
- To find out how a function or method works, use these:
  - Library Reference
  - General Index
  - These are less useful
    - Quick search and Search page produce too many irrelevant results
    - Language reference talks about syntax, not what the functions *do*
- If you know Python 2 but not Python 3, this might be useful:
  - http://wiki.python.org/moin/Python2orPython3

Eager tree search

```
class Node():
    """Class for nodes in the search tree"""
    def __init__(self,state,parent,cost):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.children = []

def expand(x,successors):
    """Return a list of node x's children"""
    print('{:14} '.format(x.state),end='')
    # Python's sets have avg lookup time O(1)
    path = set(getpath(x))
    for (state,cost) in successors(x.state):
        if state in path:
            print ("{0} x,  ".format(state), end='')
        else:
            y = Node(state, x, x.cost + cost)
            x.children.append(y)
            status = y.cost
            print ("{0} {1},    ".format(state, status), end='')
    print('')
    return x.children
```

```
def getpath(y):
    """
    Return the path from y.state
    back to the initial state
    """
    path = [y.state]
    while y.parent != False:
        y = y.parent
        path.append(y.state)
    path.reverse()
    return path
```

Eager tree search (continued)

```
def search(state, successors, goal, strategy='bf'):
    """
    Do a tree search starting at state.
    Look for a state x that satisfies goal(x).
    strategy may be either 'bf' (breadth-first) or 'df' (depth-first).
    """
    frontier = [Node(state,False,0)] # "False" means there's no parent
    print('\n{:14}  {}'.format('__Node__', '__Expansion__  . . .'))
    while frontier != []:
        if strategy == 'bf':
            x = frontier.pop(0)   # oldest node; this is inefficient
        elif strategy == 'df':
            x = frontier.pop()    # youngest node; does rightmost branch 1st
        else:
            raise RuntimeError("'" + strategy + "' is not a strategy")
        for y in expand(x,successors):
            if goal(y.state):
                print('');
                return getpath(y)
            frontier.append(y)
    return False
```

Romanian map problem

```
map ={
    'Arad':           {'Sibiu':140,'Timisoara':118,'Zerind':75},
    'Bucharest':      {'Fagaras':211,'Giurgiu':90,'Pitesti':101,'Urziceni':85},
    'Craiova':        {'Dobreta':120,'Pitesti':138,'Rimnicu Vilcea':146},
    'Dobreta':        {'Craiova':120,'Mehadia':75},
    'Eforie':         {'Hirsova':86},
    'Fagaras':        {'Bucharest':211,'Sibiu':99},
    'Giurgiu':        {'Bucharest':90},
    'Hirsova':        {'Eforie':86,'Urziceni':98},
    'Iasi':           {'Neamt':87,'Vaslui':92},
    'Lugoj':          {'Mehadia':70,'Timisoara':111},
    'Mehadia':        {'Dobreta':75,'Lugoj':70},
    'Neamt':          {'Iasi':87},
    'Oradea':         {'Sibiu':151,'Zerind':71},
    'Pitesti':        {'Bucharest':101,'Craiova':138,'Rimnicu Vilcea':97},
    'Rimnicu Vilcea':{'Craiova':146,'Pitesti':97,'Sibiu':80},
    'Sibiu':          {'Arad':140,'Fagaras':99,'Oradea':151,'Rimnicu Vilcea':80},
    'Timisoara':      {'Arad':118,'Lugoj':111},
    'Urziceni':       {'Bucharest':85,'Hirsova':98,'Vaslui':142},
    'Vaslui':         {'Iasi':92,'Urziceni':142},
    'Zerind':         {'Arad':75,'Oradea':71}}
```

Romanian map problem (continued)

```python
def neighbors(state):
    """
    Use this as the successors function. It returns state's
    neighbors on the map, as a sequence of (state,cost) pairs"""
    return map[state].items()

def is_bucharest(state):
    """
    Use this as the goal predicate.
    It returns True if state = Bucharest, else False
    """
    return state == 'Bucharest'
```