Problem solving and search – Chapter 3

August 23, 2021

Outline

- ♢ Problem-solving agents
- ♢ Problem types
- ♢ Problem formulation
- ♢ Example problems
- ♢ Basic search algorithms

Problem-solving agents

- Restricted form of general agent:**offline** problem solving (solution executed "eyes closed.")
- **Online** problem solving involves acting without complete knowledge.

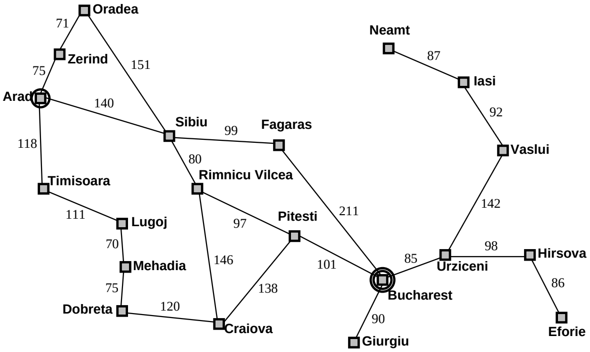
```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
   static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

   state ← UPDATE-STATE(state, percept)
   if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH( problem)
   action ← RECOMMENDATION(seq, state)
   seq ← REMAINDER(seq, state)
   return action
```

Example: Romania

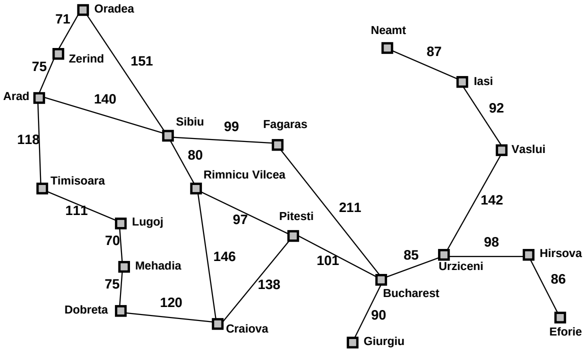
On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

- Formulate goal: be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution: sequence of cities, e.g., Arad, Sibiu, **Fagaras**, Bucharest

Example: Romania



Example: Romania



Straight−line distance
to Bucharest

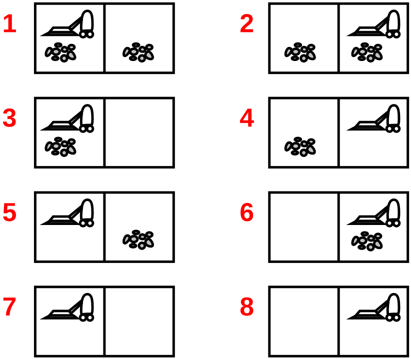
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Problem types

- Deterministic, fully observable $\Longrightarrow$ single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Deterministic, Non-observable $\Longrightarrow$ conformant problem
  - Agent may have no idea where it is; solution (if any) is a sequence
- Nondeterministic and/or partially observable $\Longrightarrow$
  - contingency problem
  - percepts provide **new** information about current state
  - solution is a contingent plan or a policy
  - often **interleave** search, execution
- Unknown state space $\Longrightarrow$ exploration problem ("online")

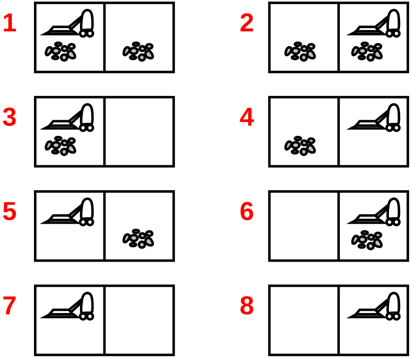
Example: vacuum world - 1

Single-state, start in #5. Solution??



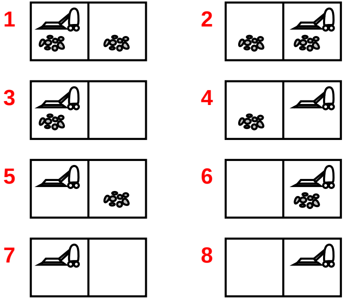
Example: vacuum world - 2

Single-state, start in #5. Solution?? $[Right, Suck]$
Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., *Right* goes to $\{2, 4, 6, 8\}$. Solution??



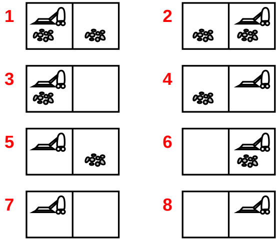
Example: vacuum world - 3

Single-state, start in #5. Solution??
$[Right, Suck]$
Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., *Right* goes to $\{2, 4, 6, 8\}$. Solution??
$[Right, Suck, Left, Suck]$
Contingency, start in #5
Murphy's Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.
Solution??



Example: vacuum world - 4

- Single-state, start in #5. Solution ?? $[Right, Suck]$
- Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
  e.g., *Right* goes to $\{2, 4, 6, 8\}$. Solution ??
  $[Right, Suck, Left, Suck]$
- Contingency, start in #5
  Murphy's Law: *Suck* can dirty a clean carpet
  Local sensing: dirt, location only.
  Solution?? $[Right, \textbf{if}\, dirt\, \textbf{then}\, Suck]$



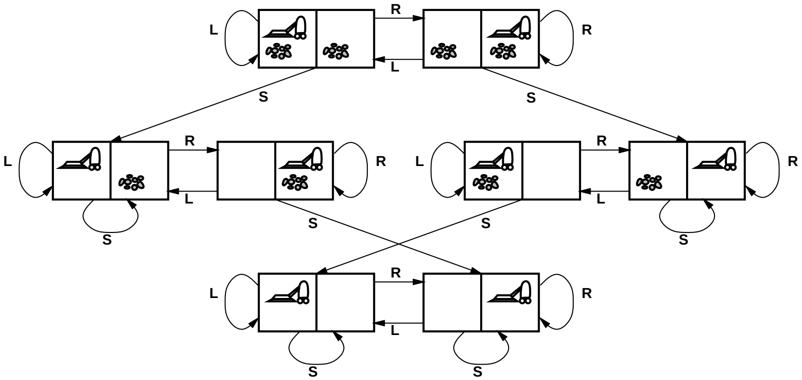
Single-state problem formulation

- A problem is defined by four items:
  1. initial state e.g., "at Arad"
  2. successor function $S(x)$ = set of action–state pairs
     e.g., $S(Arad) = \{< Arad \rightarrow Zerind, Zerind >, \ldots\}$
  3. goal test, can be
     - explicit, e.g., x = "at Bucharest"
     - implicit, e.g., $NoDirt(x)$
  4. path cost (additive)
     e.g., sum of distances, number of actions executed, etc.
     $c(x, a, y)$ is the step cost, assumed to be ≥ 0
- A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

- Real world is (absurdly?) complex $\Rightarrow$ state space must be **abstracted** for problem solving
  - (Abstract) state $=$ set of real states
  - (Abstract) action $=$ complex combination of real actions
    e.g., "Arad $\rightarrow$ Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, **any** real state "in Arad" must get to some real state "in Zerind"
- (Abstract) solution $=$ set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem!

Example: vacuum world state space graph



Example: vacuum world state space graph

states??
actions??
goal test??
path cost??

Example: vacuum world state space graph

states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??
goal test??
path cost??

Example: vacuum world state space graph

states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??
path cost??

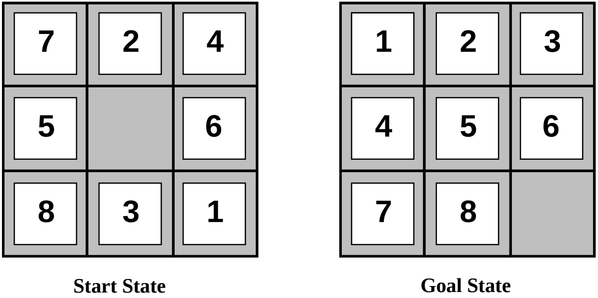
Example: vacuum world state space graph

states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??

Example: vacuum world state space graph

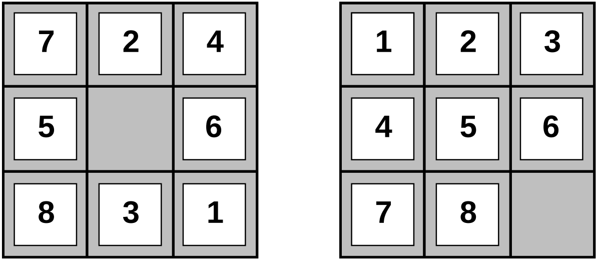
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??: 1 per action (0 for *NoOp*)

Example: The 8-puzzle



states??
actions??
goal test??
path cost??

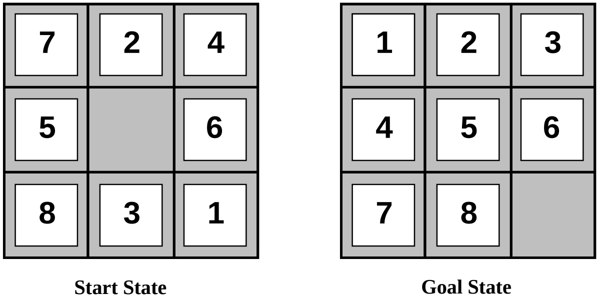
Example: The 8-puzzle



Start State **Goal State**

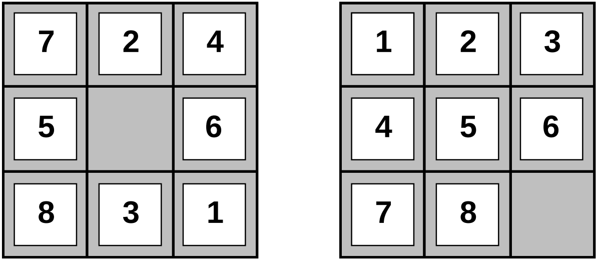
states??: integer locations of tiles (ignore intermediate positions)
actions??
goal test??
path cost??

Example: The 8-puzzle



<u>states??</u>: integer locations of tiles (ignore intermediate positions)
<u>actions??</u>: move blank left, right, up, down (ignore unjamming etc.)
<u>goal test??</u>
<u>path cost??</u>

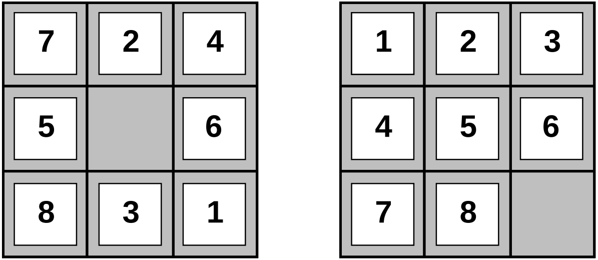
Example: The 8-puzzle



Start State **Goal State**

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??

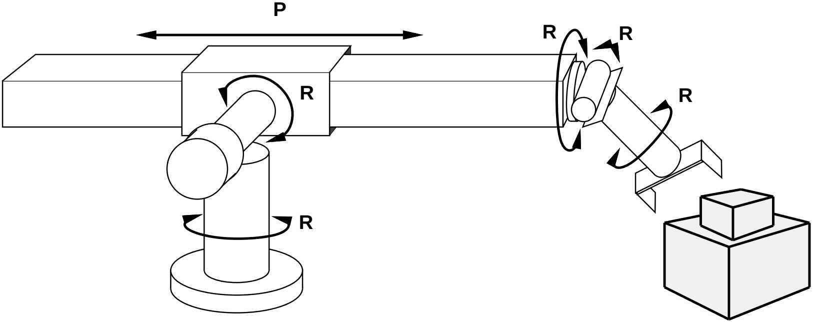
Example: The 8-puzzle



Start State **Goal State**

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move
[Note: optimal solution of *n*-Puzzle family is NP-hard]

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 **with no robot included!**
path cost??: time to execute

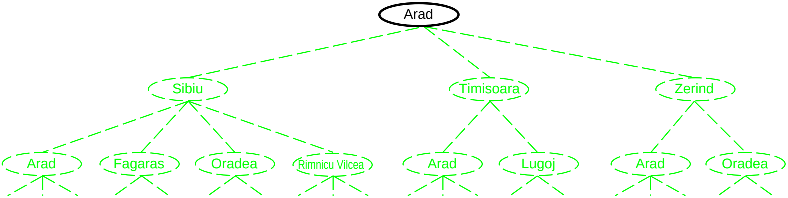
Tree search algorithms

Basic idea:

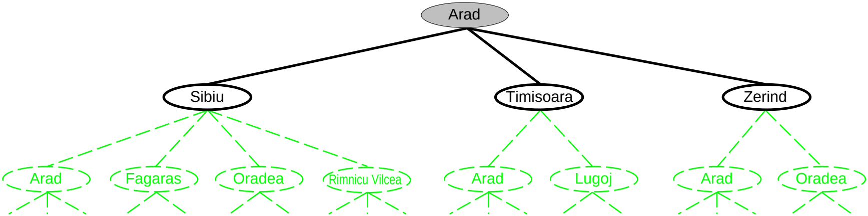
- offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```
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
  end
```

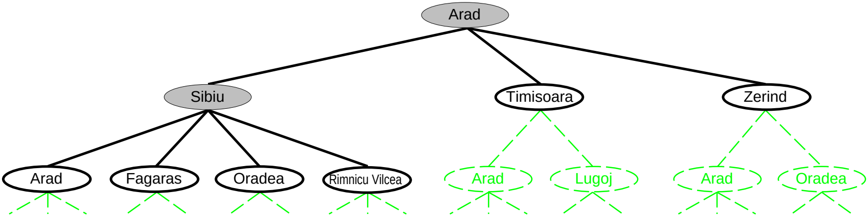
Tree search example



Tree search example

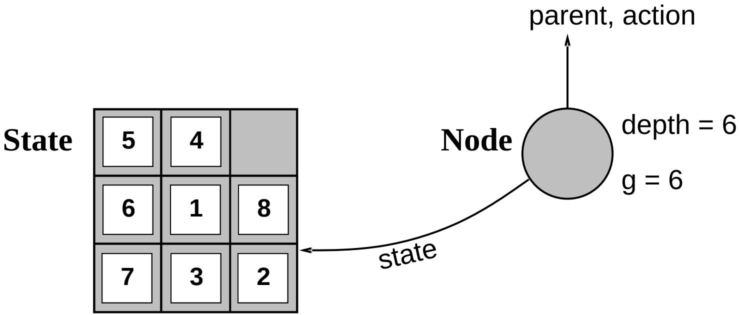


Tree search example



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.

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, STATE(node)) then 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(STATE[node], action, result)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
   return successors
```

Search strategies

- A strategy is defined by picking the **order of node expansion**
- Strategies are evaluated along the following dimensions:
  1. completeness—does it always find a solution if one exists?
  2. time complexity—number of nodes generated/expanded
  3. space complexity—maximum number of nodes in memory
  4. optimality—does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  1. b—maximum branching factor of the search tree
  2. d—depth of the least-cost solution
  3. m—maximum depth of the state space (may be ∞)

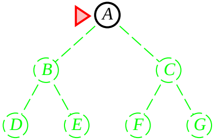
Uninformed search strategies

- Uninformed strategies use only the information available in the problem definition

  1. Breadth-first search
  2. Uniform-cost search
  3. Depth-first search
     - Depth-limited search
     - Iterative deepening search

Breadth-first search

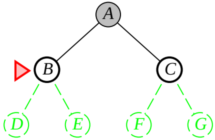
- Expand shallowest unexpanded node
  Implementation: *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

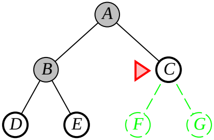
Implementation: *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

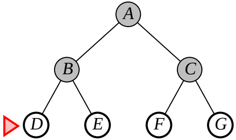
Expand shallowest unexpanded node

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?? 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; can easily generate nodes at 100MB/sec

so 24hrs $=$ 8640GB.

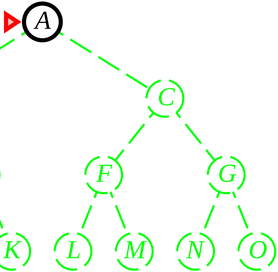
Uniform-cost search

Expand least-cost unexpanded node
Implementation: *fringe* = queue ordered by path cost, lowest first
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 $g(n)$

Depth-first search

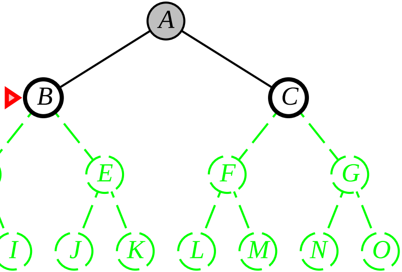
Expand deepest unexpanded node
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

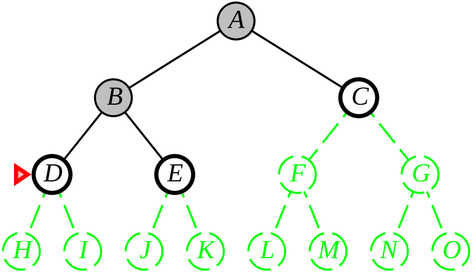
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

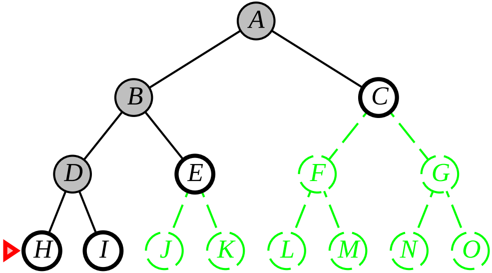
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

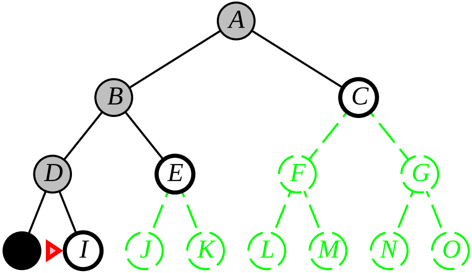
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

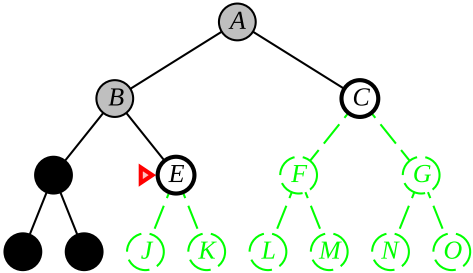
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

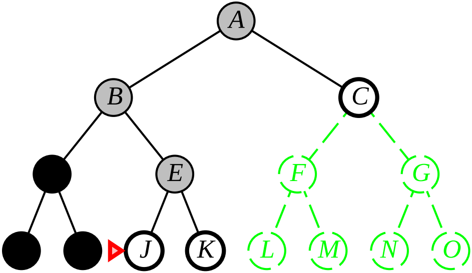
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

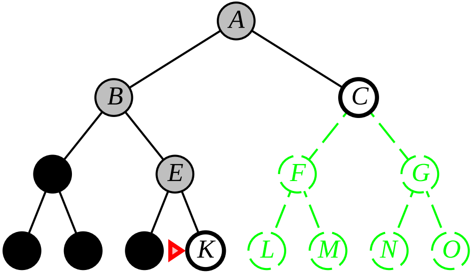
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

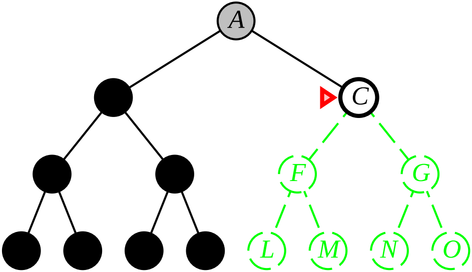
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

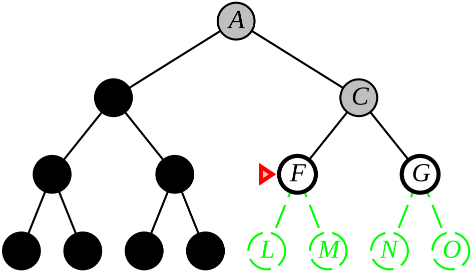
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

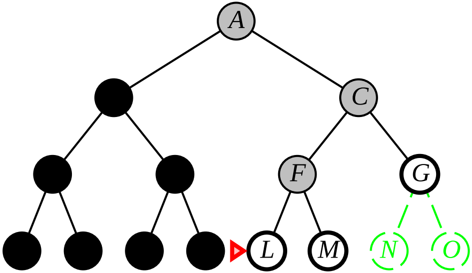
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

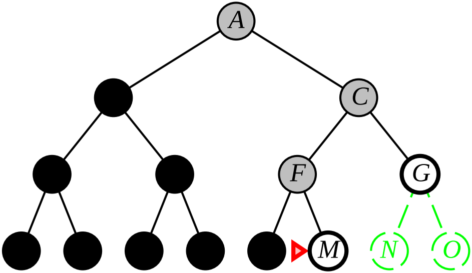
Implementation: *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

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



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

Depth-limited search

Depth-limited search = depth-first search with depth limit l: 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*)

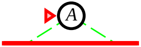
function Recursive-DLS(*node*, *problem*, *limit*) **returns** soln/fail/cutoff
  cutoff-occurred? ← false
  if Goal-Test(*problem*, State[*node*]) **then return** *node*
  else if Depth[*node*] = *limit* **then return** *cutoff*
  else for each *successor* **in** Expand(*node*, *problem*) **do**
    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*

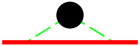
Iterative deepening search

```
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 $l = 0$

Limit = 0

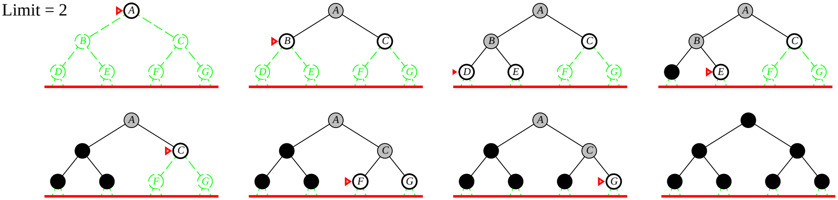




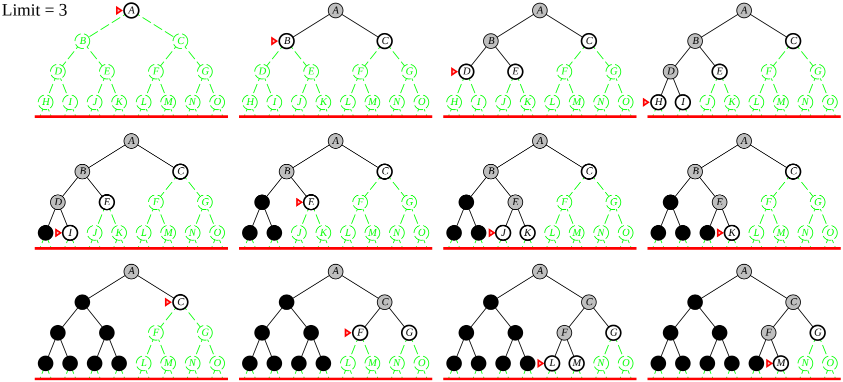
Iterative deepening search $l = 1$

Limit = 1

Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



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
Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$
\begin{aligned}
N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 \\
&= 1,111,100
\end{aligned}
$$

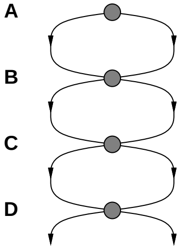
IDS does better because other nodes at depth d are not expanded
BFS can be modified to apply goal test when a node is **generated**

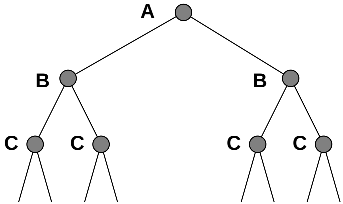
Summary of algorithms

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*

Repeated states

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





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 node
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```

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 and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search