Solving problems by searching

CHAPTER 3

Outline

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

Problem-solving agents

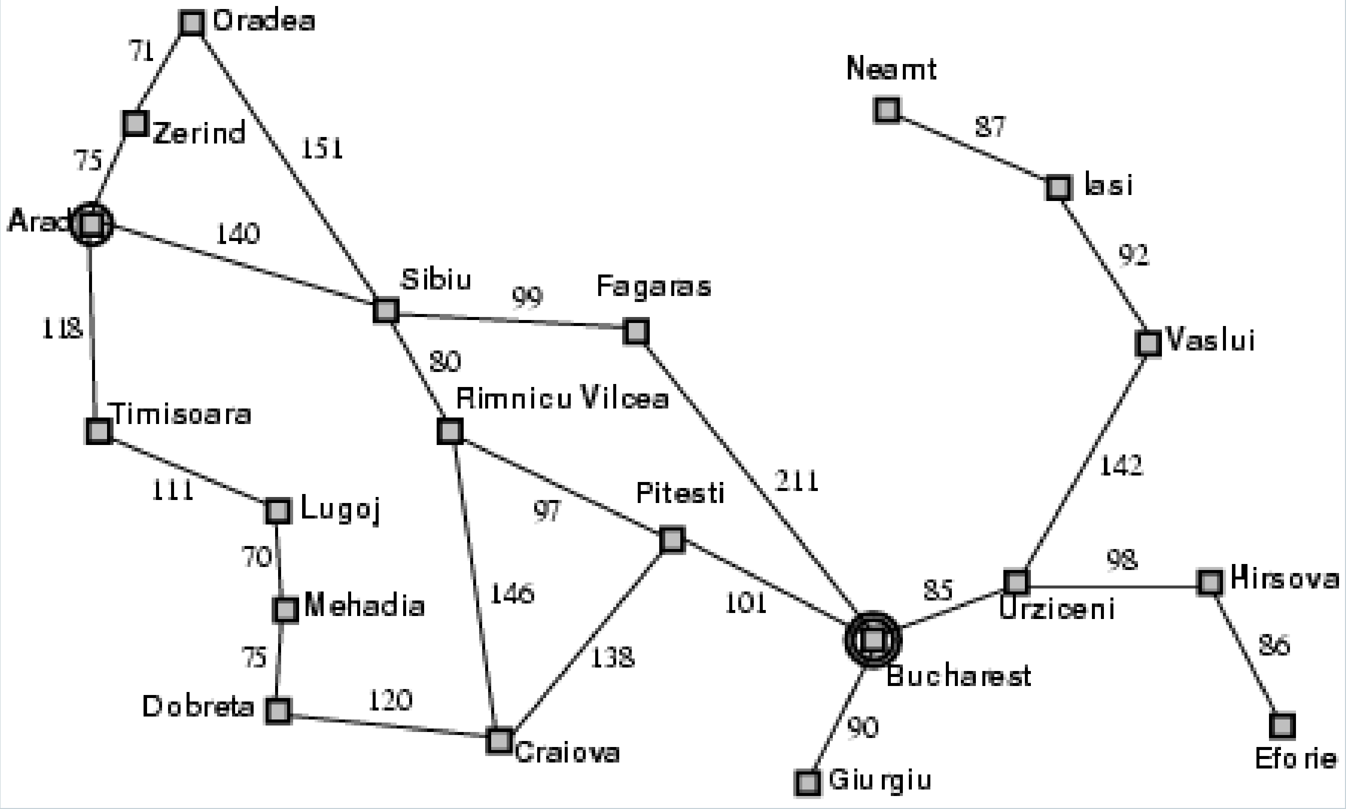

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

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

Note: this is offline problem solving; solution executed “eyes closed.”

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
  -

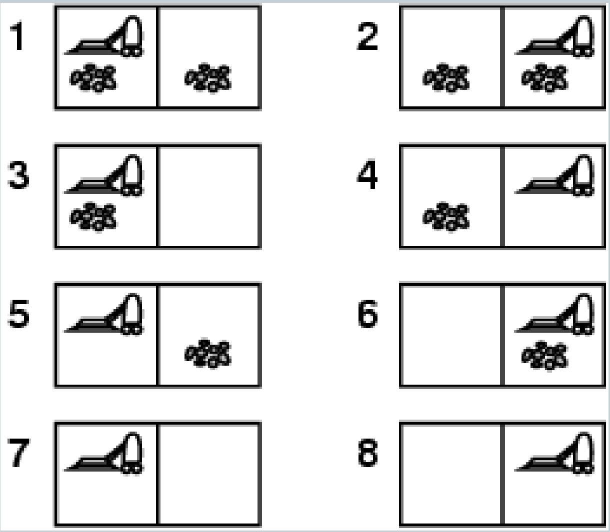
Example: Romania



Abstraction: The process of removing details from a representation
Is the map a good representation of the problem? What is a good replacement?

Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
  - Vacuum world → everything observed
  - Romania → The full map is observed

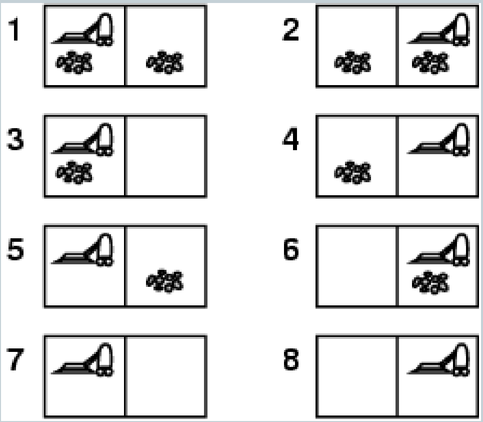


- Single-state: Start in #5. Solution??
  - [Right, Suck]

Problem types

- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
  - Vacuum world → No sensors
  - Romania → No map just know operators(cities you can move to)

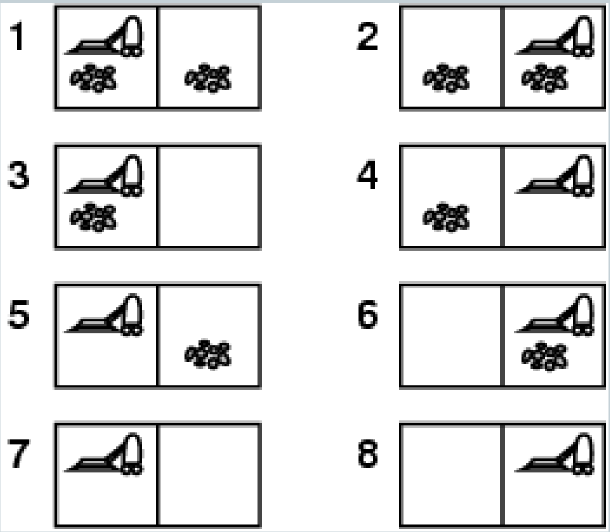
- Conformant: Start in {1, 2, 3, 4, 5, 6, 7, 8}
  - e.g., Right goes to {2, 4, 6, 8}. Solution??
  - [Right, Suck,Left, Suck]



Problem types

- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - Unknown state space → exploration problem
  - Vacuum world → know state of current location
  - Romania → know current location and neighbor cities

- Contingency: [L,clean]
  - Start in #5 or #7
  - Murphy's Law: Suck can dirty a clean carpet
  - Local sensing: dirt, location only.
  - Solution??
  - [Right, if dirt then Suck]



Single-state problem formulation

A problem is defined by four items:

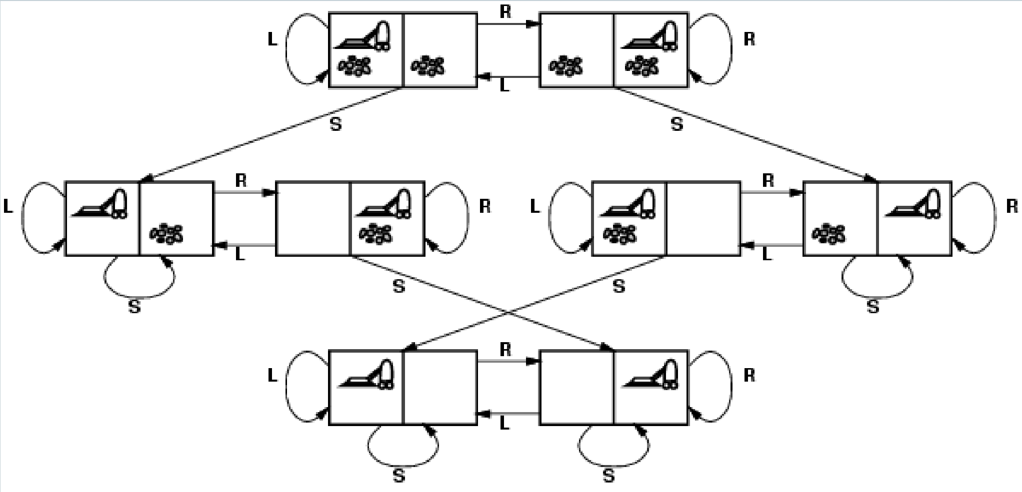
1. initial state e.g., "at Arad“
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, 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

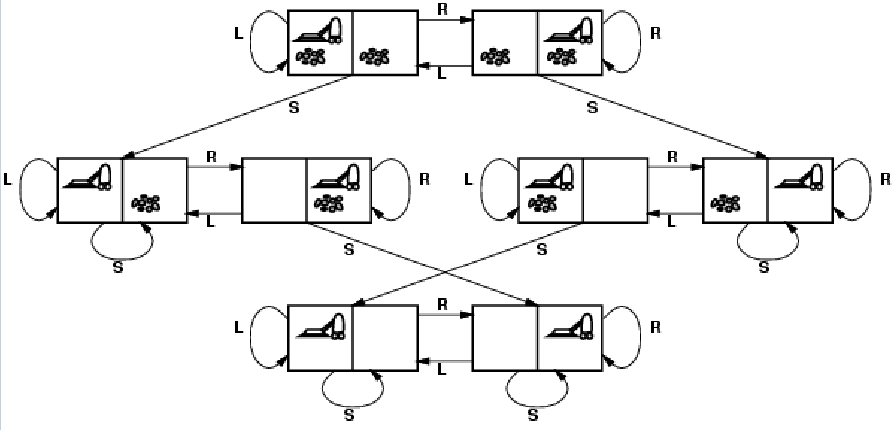
- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
-
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- (Abstract) solution =
  - set of real paths that are solutions in the real world
  -
- Each abstract action should be "easier" than the original problem
-

Vacuum world state space graph



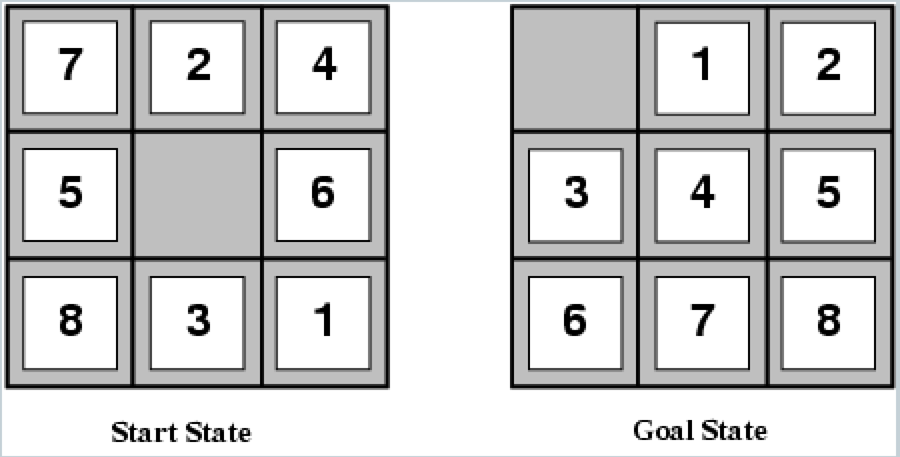
- states?
- actions?
- goal test?
- path cost?
-

Vacuum world state space graph



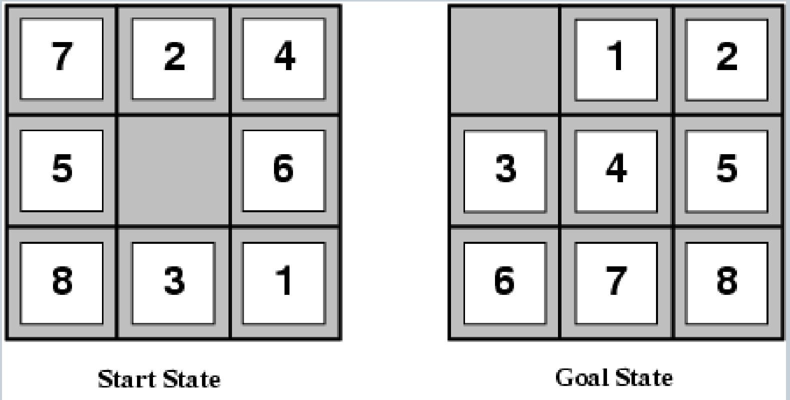
- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



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

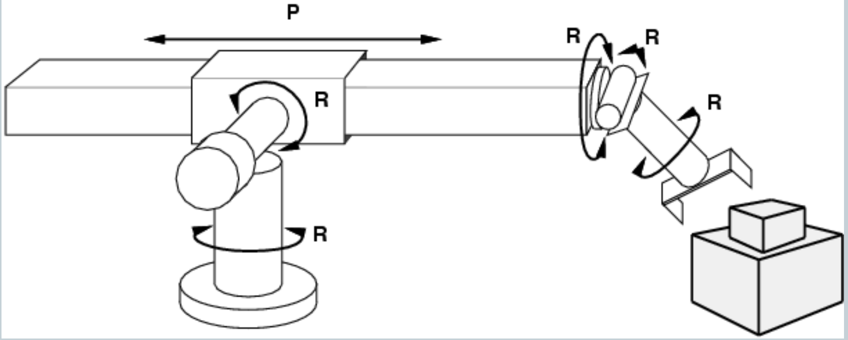
Example: The 8-puzzle



- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: robotic assembly



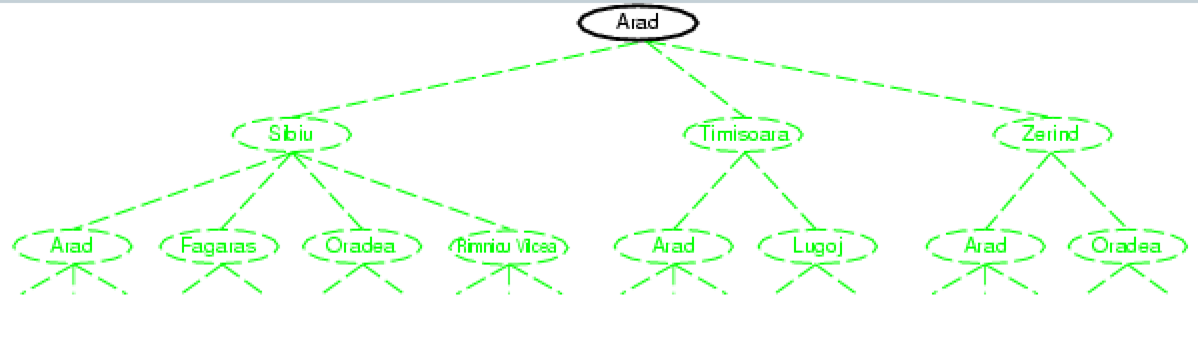
- states?:
  - real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?:
  - continuous motions of robot joints
- goal test?:
  - complete assembly
- path cost?:
  - time to execute

Tree search algorithms

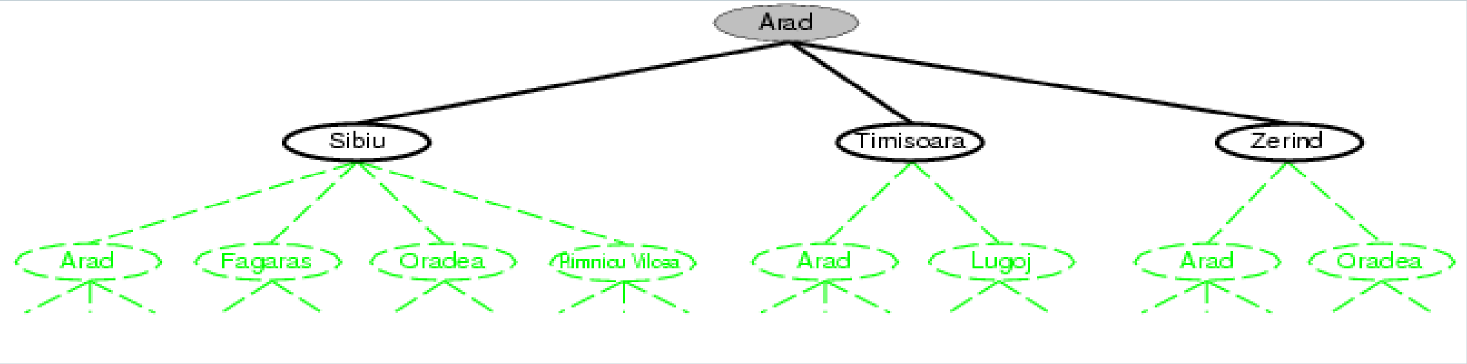
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
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
```

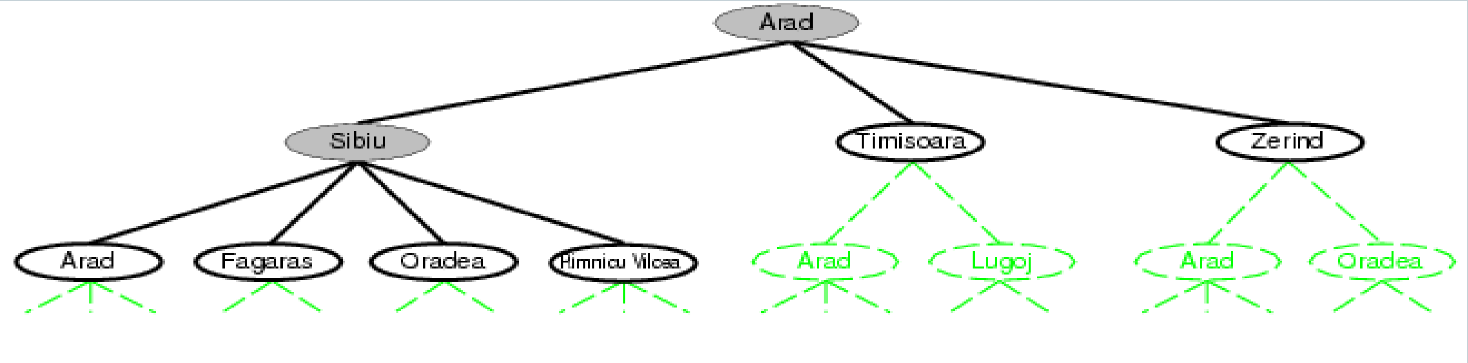
Tree search example



Tree search example



Tree search example



Search strategies

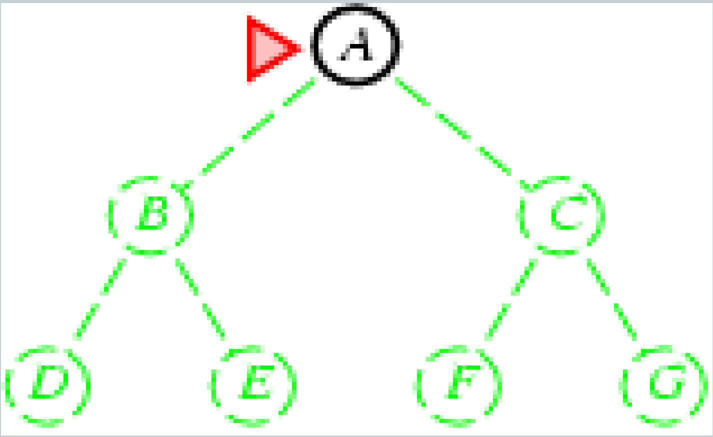
- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: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

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

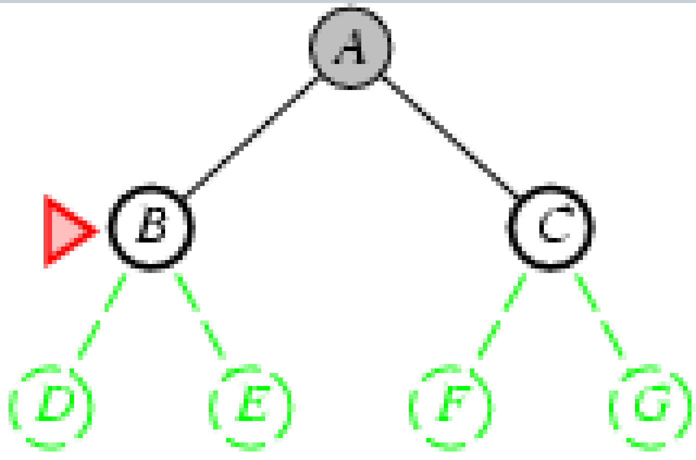
Breadth-first search

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



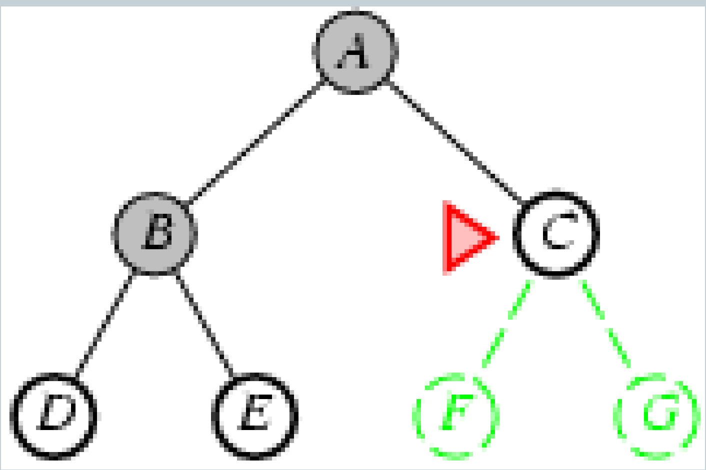
Breadth-first search

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



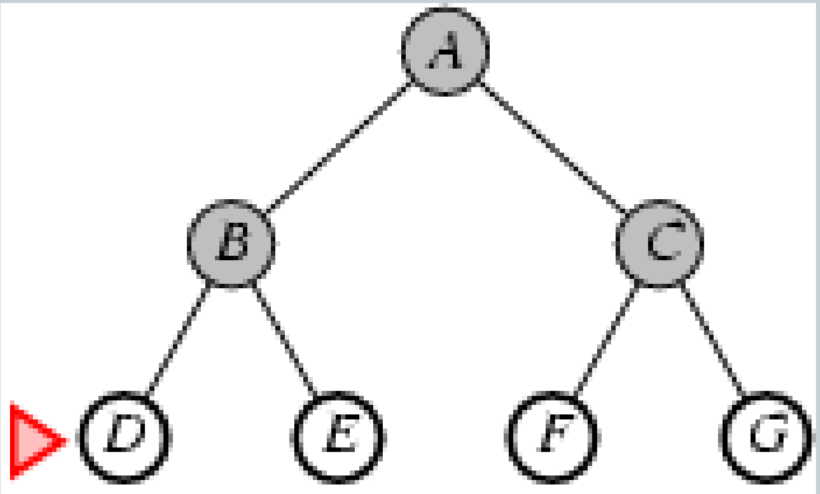
Breadth-first search

- 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

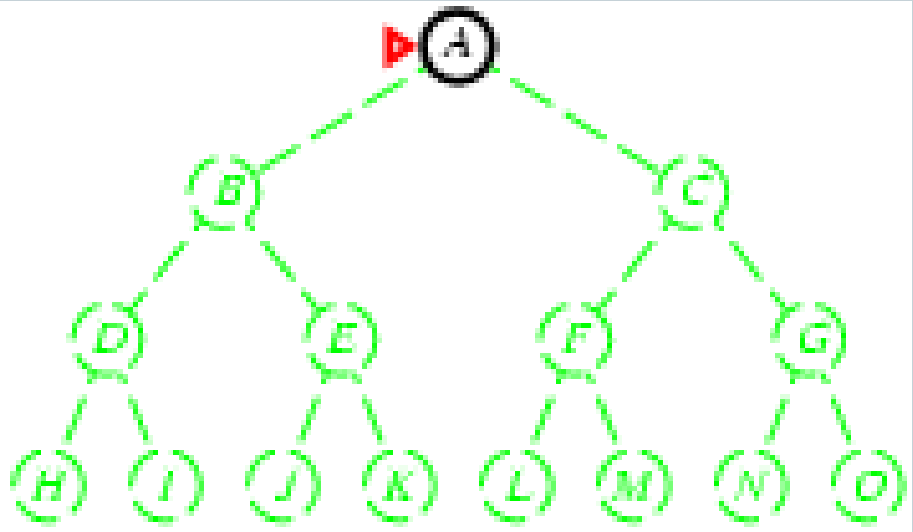
- Complete? Time? Space?Optimal?
- Complete? Yes (if b is finite)
-
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
-
- Space? $O(b^{d+1})$ (keeps every node in memory)
-
- Optimal? Yes (if cost = 1 per step)
-
- Space is the bigger problem (more than time)
-

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 ≥ ε
-
- Time? $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? $O(b^{ceiling(C^*/\varepsilon)})$
-
- Optimal? Yes – nodes expanded in increasing order of $g(n)$
-

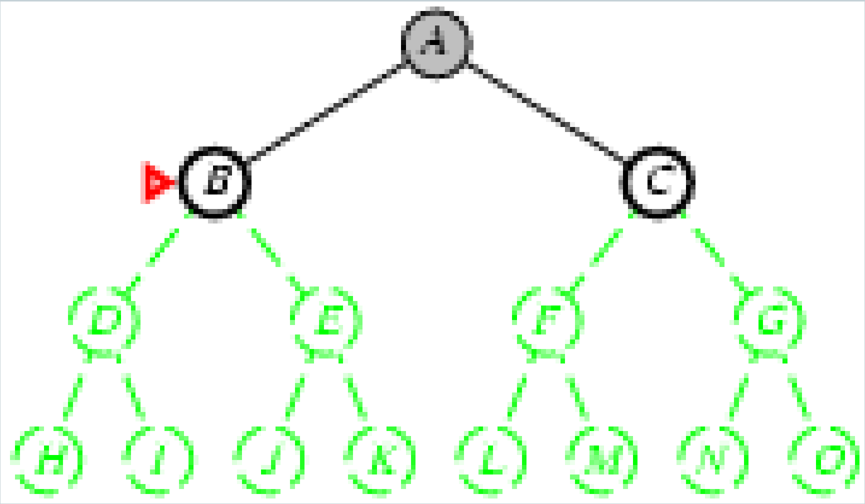
Depth-first search

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



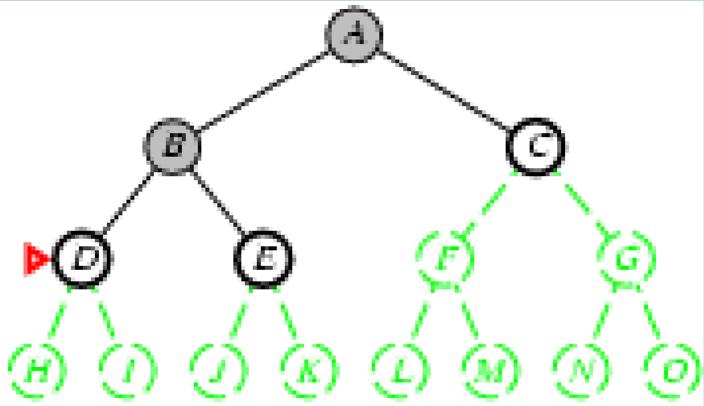
Depth-first search

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



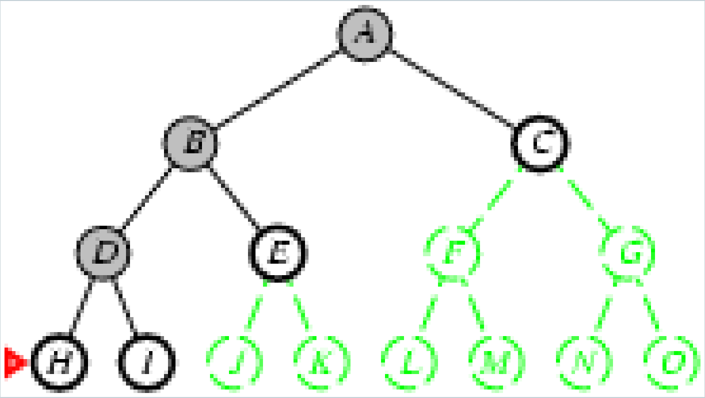
Depth-first search

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



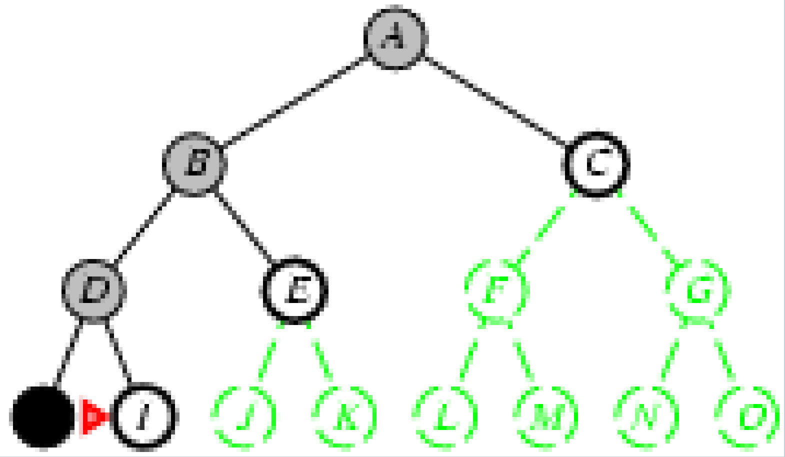
Depth-first search

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



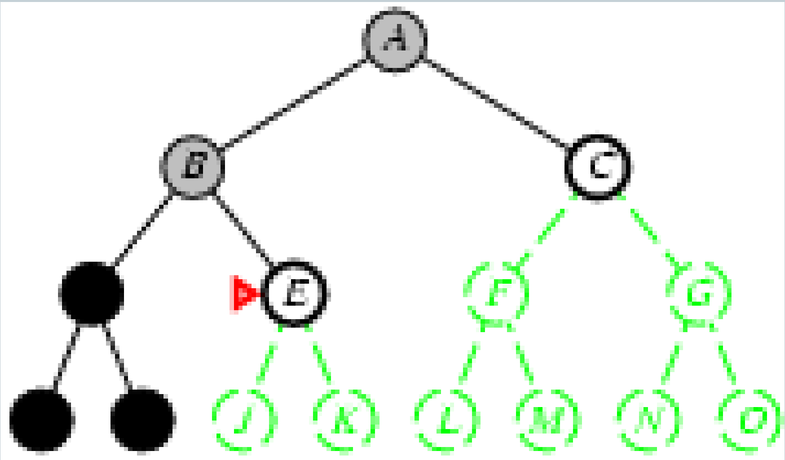
Depth-first search

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



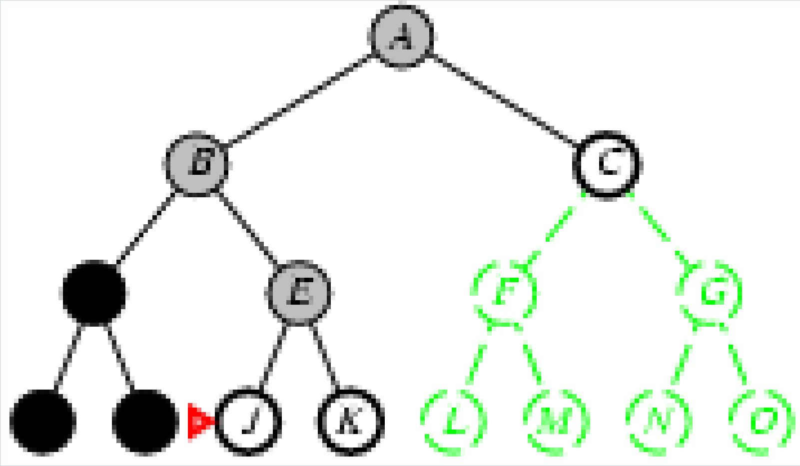
Depth-first search

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

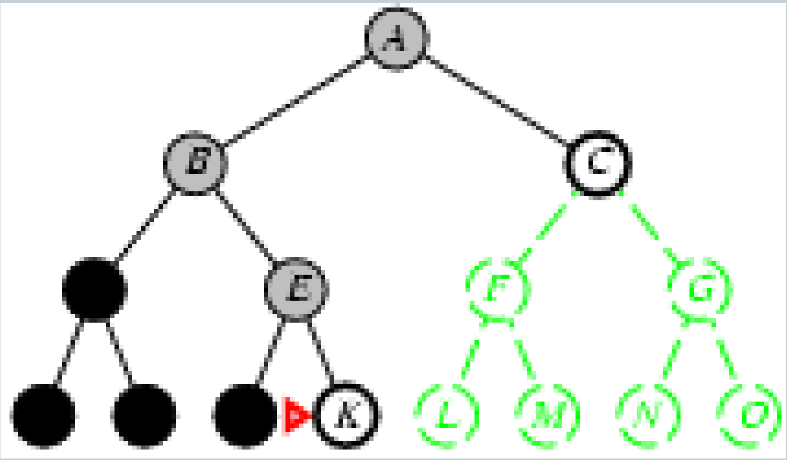


Depth-first search

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

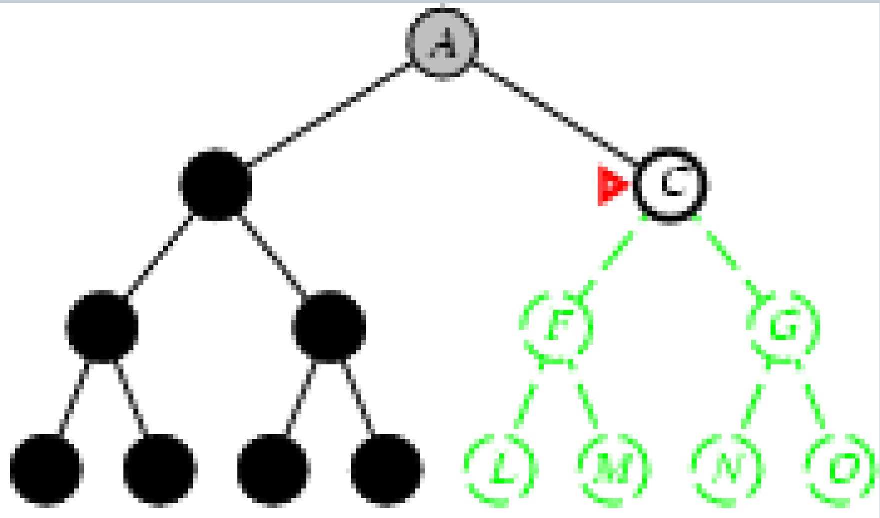


Depth-first search

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

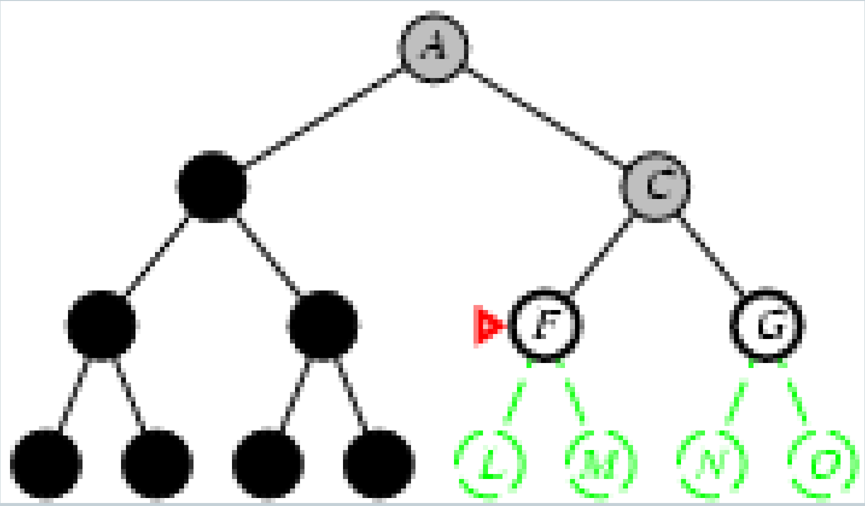
Depth-first search

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



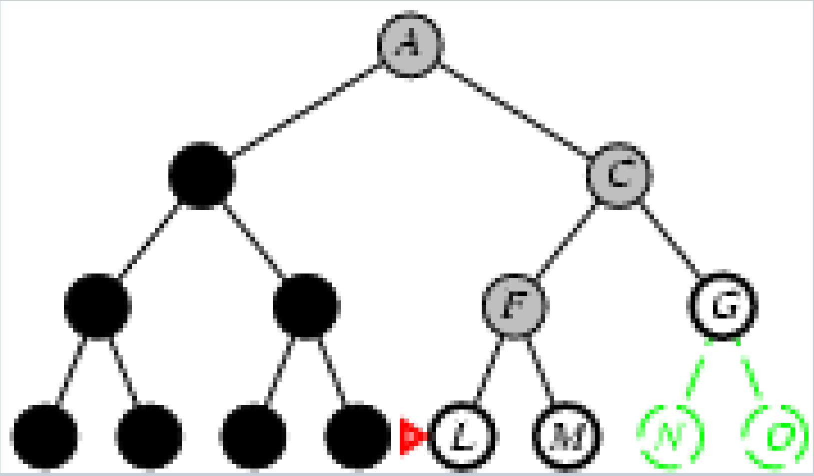
Depth-first search

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



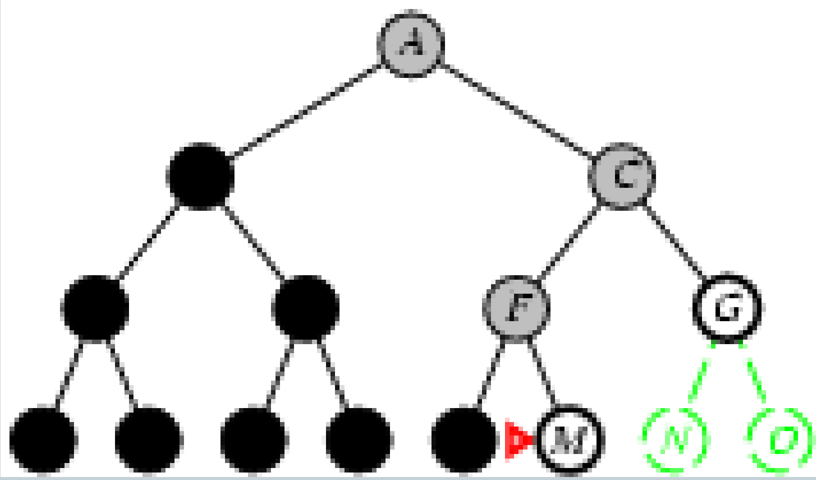
Depth-first search

- 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? Time? Space?Optimal?

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

    → complete in finite spaces

- Time? $O(b^m)$: 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-first search with depth limit *l*,
  - i.e., nodes at depth *l* have no successors
  - Solves infinite loop problem
-
- Complete? No if l < d:
- Time? $O(b^l)$:
- Space? $O(bl)$, i.e., linear space!
- Optimal? No if l > b

Iterative deepening search

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

  inputs: *problem*, a problem

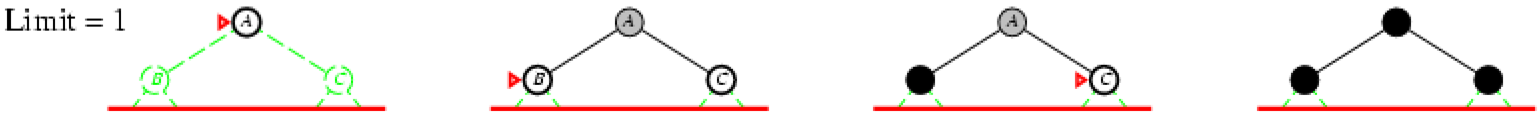
  for *depth* $\leftarrow$ 0 **to** ∞ **do**

    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)

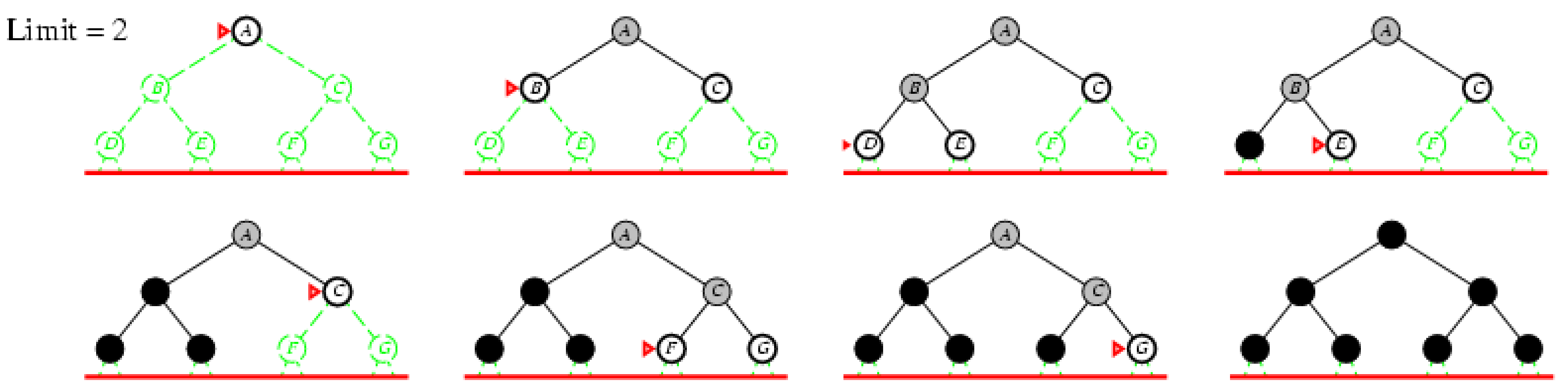
    if *result* $\neq$ cutoff **then return** *result*

Iterative deepening search $l = 0$

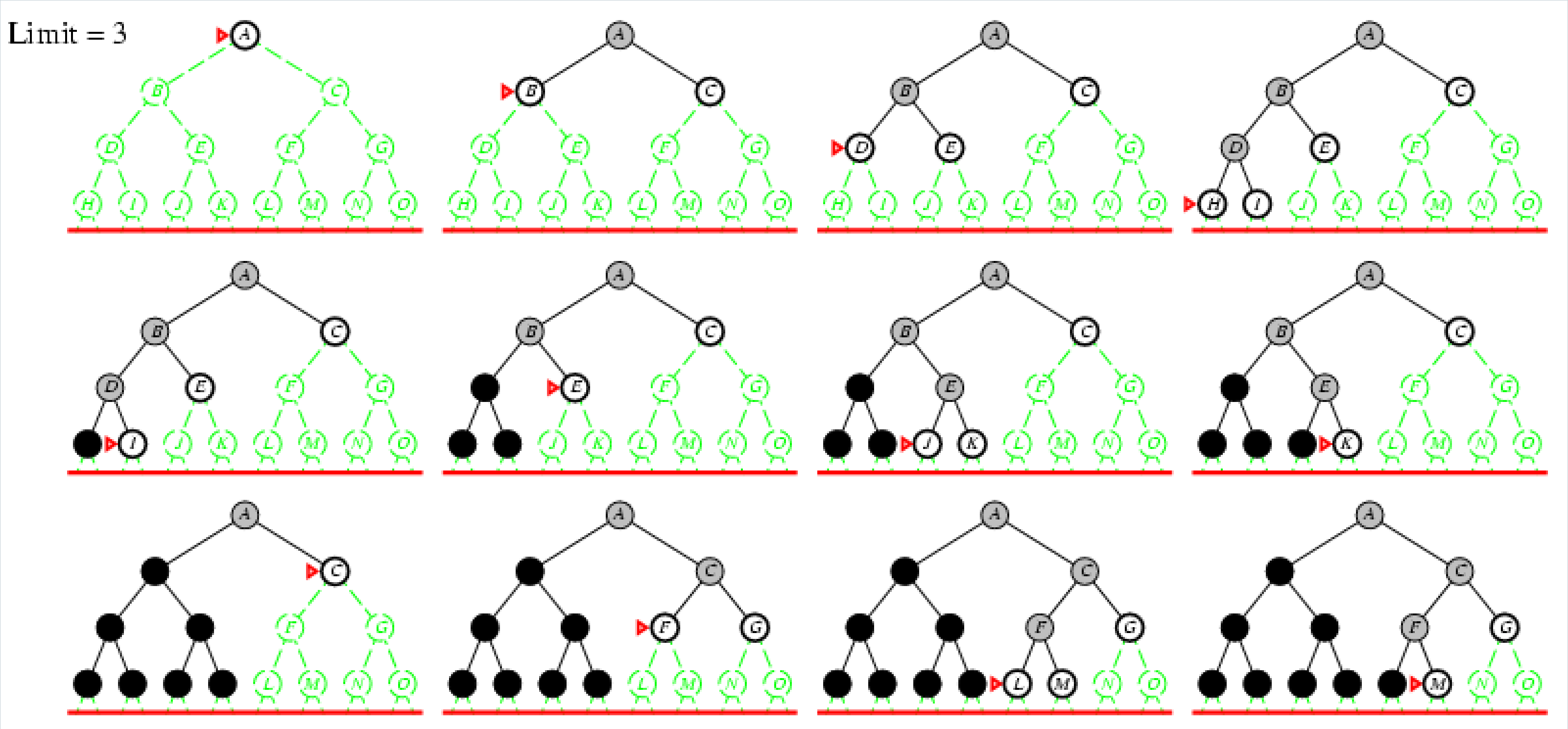
Iterative deepening search l =1



Iterative deepening search $l = 2$



Iterative deepening search l =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

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

$$N_{IDS} = (d+1)b^0 + d\ b\string^^1 + (d-1)b\string^^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

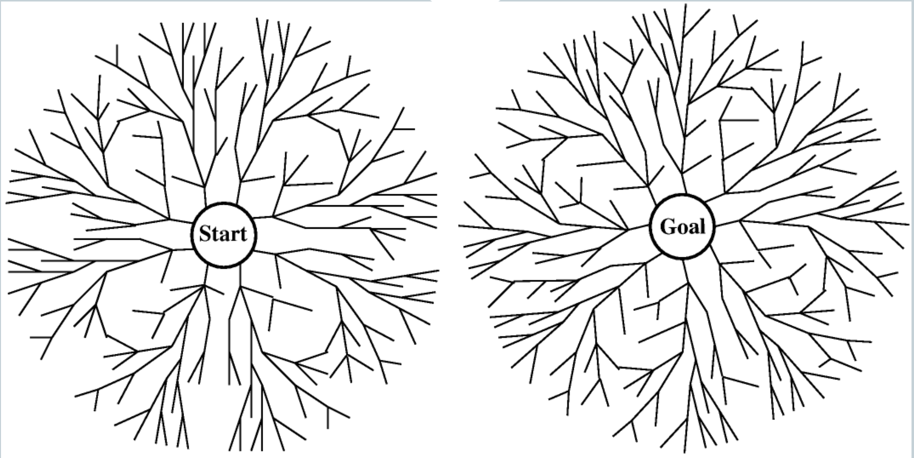
- For $b = 10$, $d = 5$,
-
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  -
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
  -

- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Bidirectional serach



- Motivation : time O(b $^{d/2}$)
- Example d=6, b=10
  - BFS = 11,111,000 nodes
  - Bidirectional = 22 200 nodes

Properties of Bidirectional search

- Complete? Yes
-
- Time? $O(b^{d/2})$
-
- Space? $O(b^{d/2})$
-
- Optimal? Yes

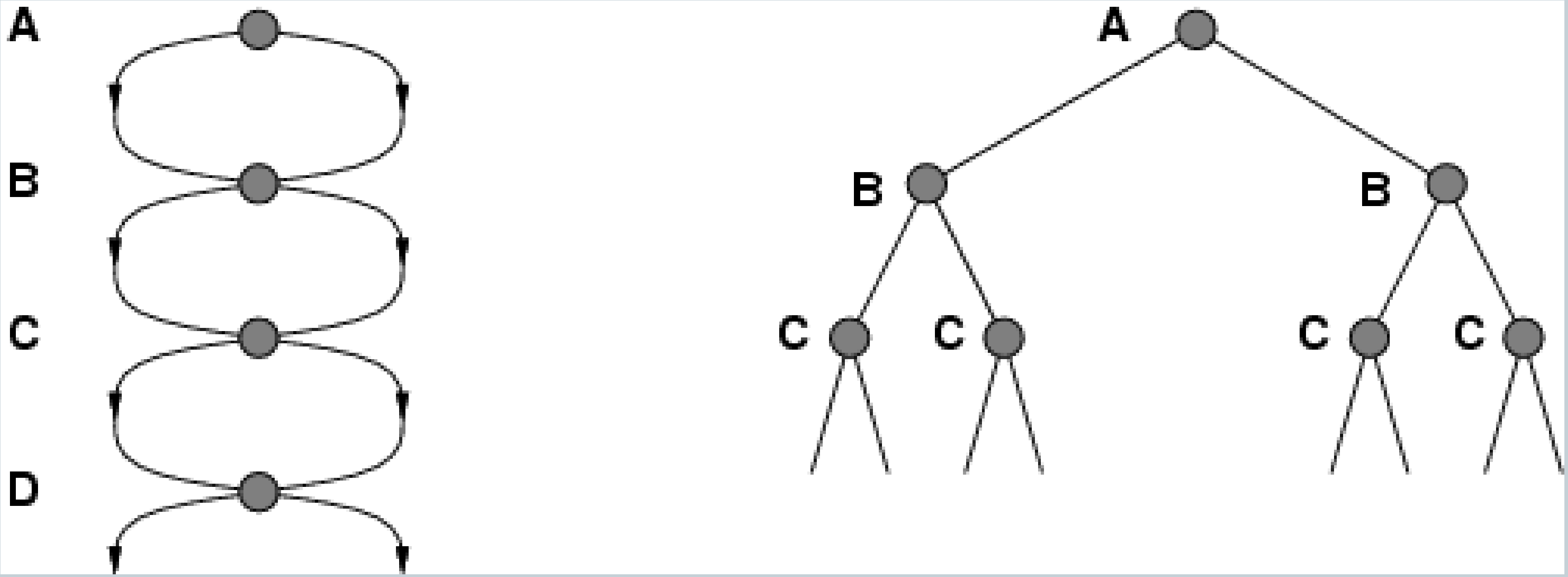
Summary of algorithms

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

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	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 SOLUTION(node)
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

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
-