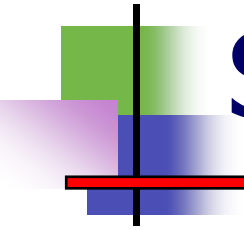


Solving problems by searching

Uninformed or Blind Search

Outline

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

Problem-solving agent

A simple Problem-solving agent is a goal-based agent. It decides what to do by finding different possible sequences of actions that lead to desirable states, and then choosing the best sequence.

This process of looking for such a sequence is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase.

Thus "formulate, search, execute" is the main parts in designing an agent.

Problem-solving agent

A problem can be defined formally by four components:

- Initial state
- A description of the possible actions available to the agent
- Goal test: determines whether a given state is a goal state.
- Path cost: It assigns a numeric cost of each path.

Problem solving algorithm:

- A solution to a problem is a path from initial state to a goal state.
- Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

A simple problem-solving agent

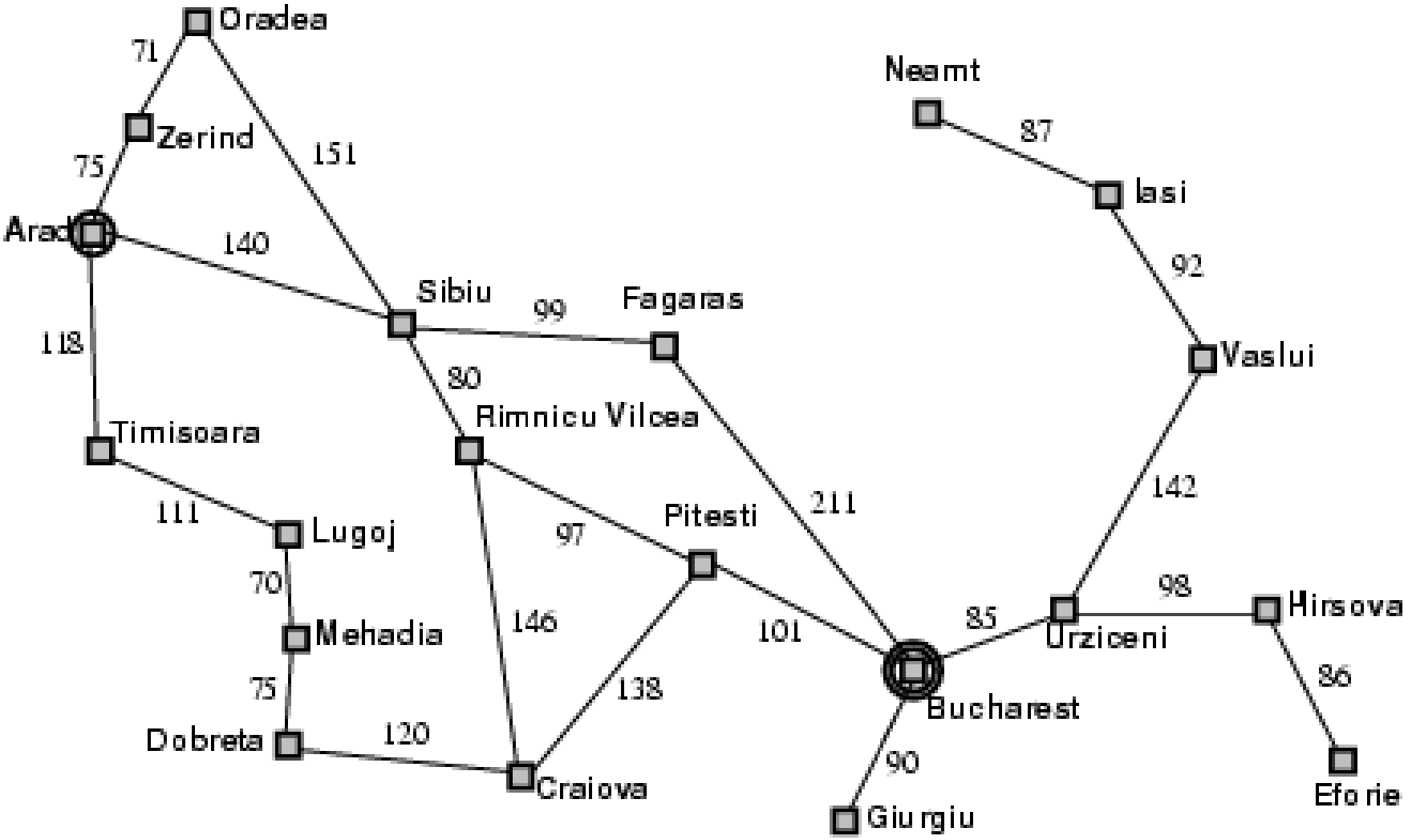
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

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

Example Problem:Touring Holiday in Romania

- The agent on holiday in Romania; currently in Arad
- Have a nonrefundable flight ticket to fly out of Bucharest tomorrow
- Formulate goal:
  - be in Bucharest and catch the flight
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Road Map of Romania



Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
- Unknown state space → exploration problem

Single-state problem formulation

A problem is defined by four items:

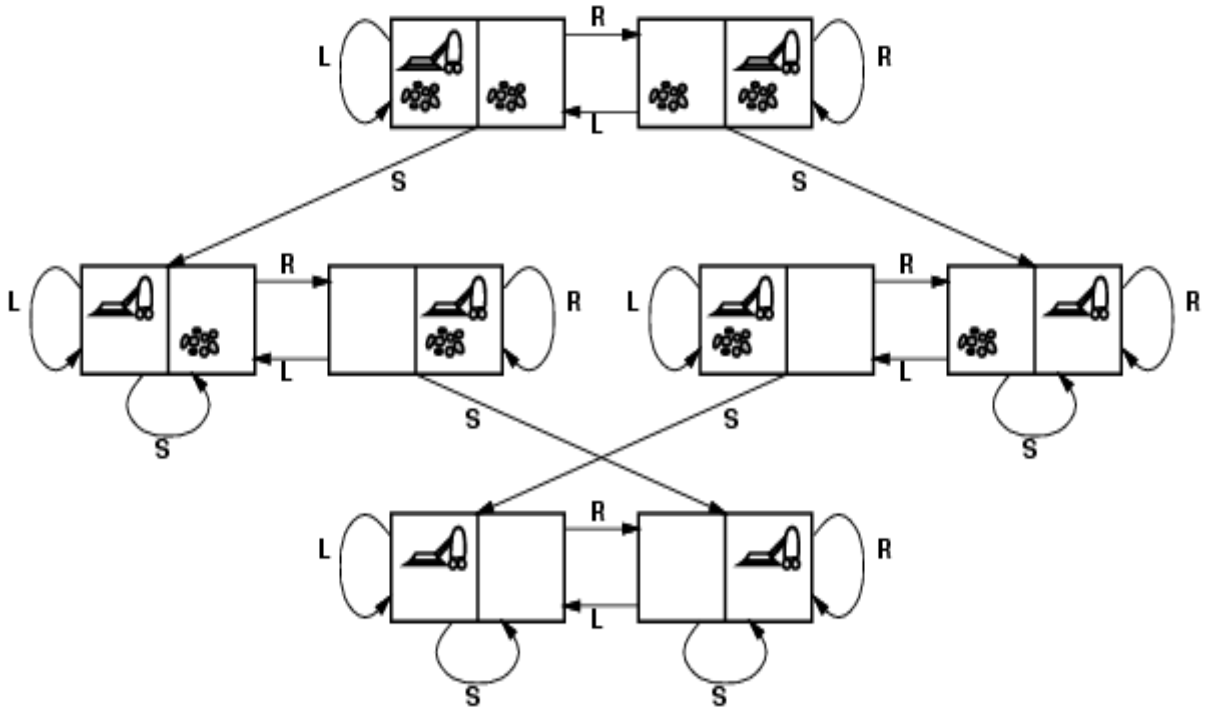
1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost of taking action *a* to go from state *x* to state *y*, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

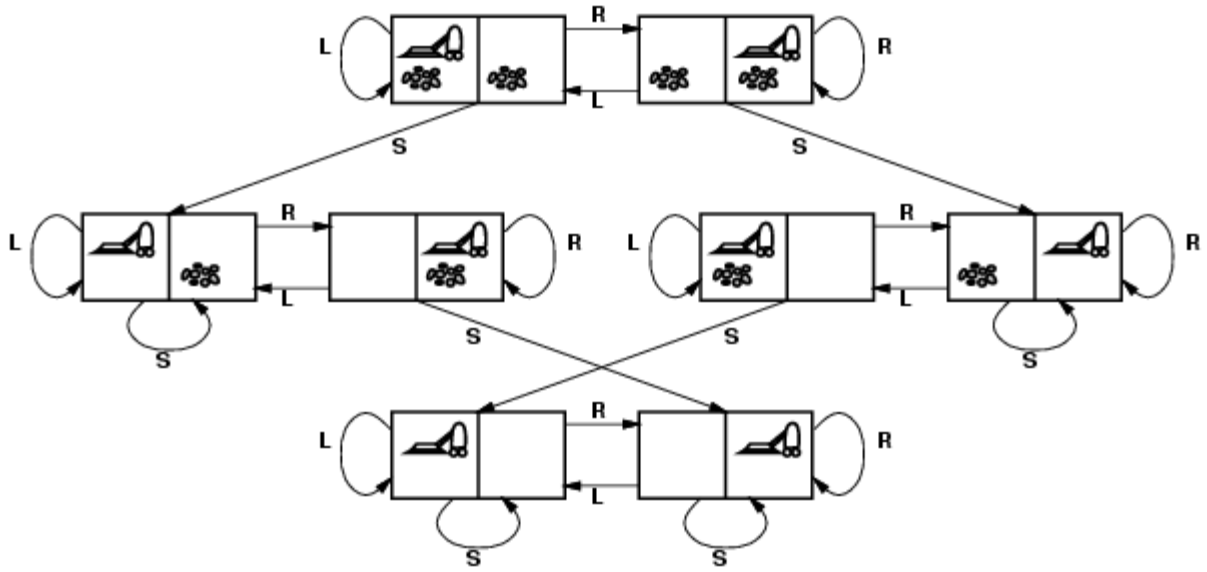
- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
-
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed 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

Vacuum world state space graph



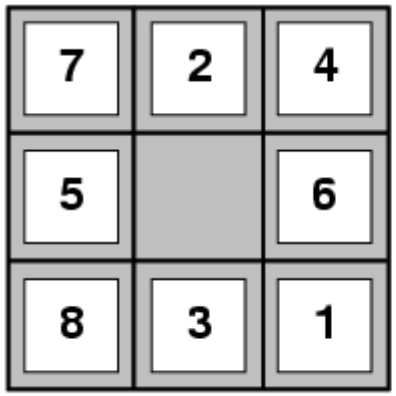
- states?
- actions?
- goal test?
- path cost?
-

Vacuum world state space graph

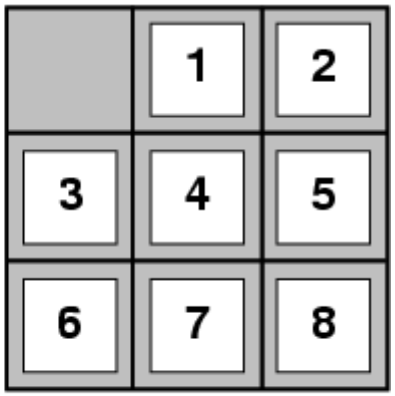


- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



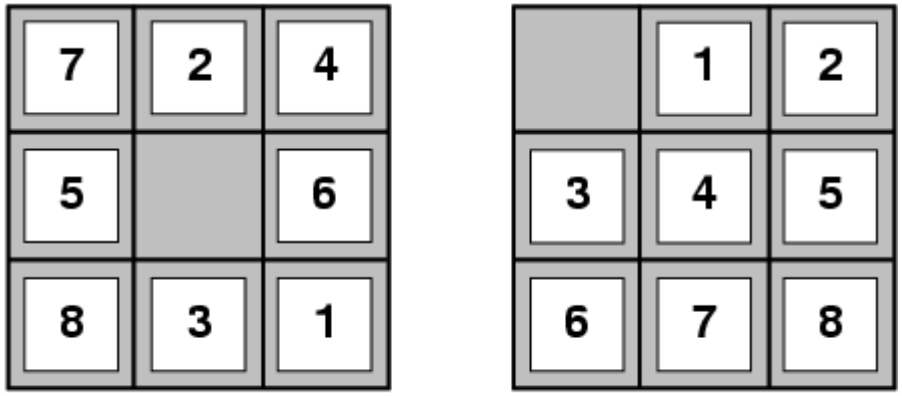
Start State



Goal State

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

Example: The 8-puzzle



Start State Goal State

- 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]

Searching for solutions

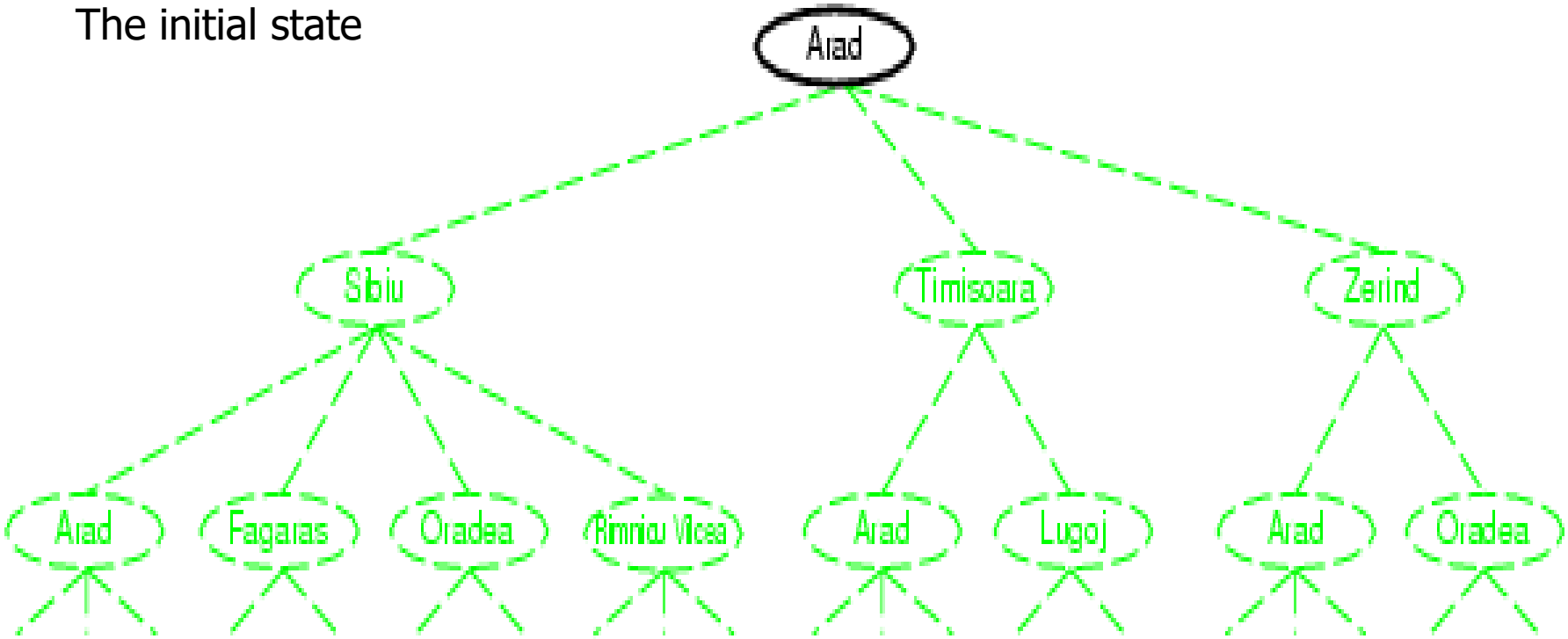
Tree search algorithm

```
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
```

An informal description of the general tree-search algorithm.

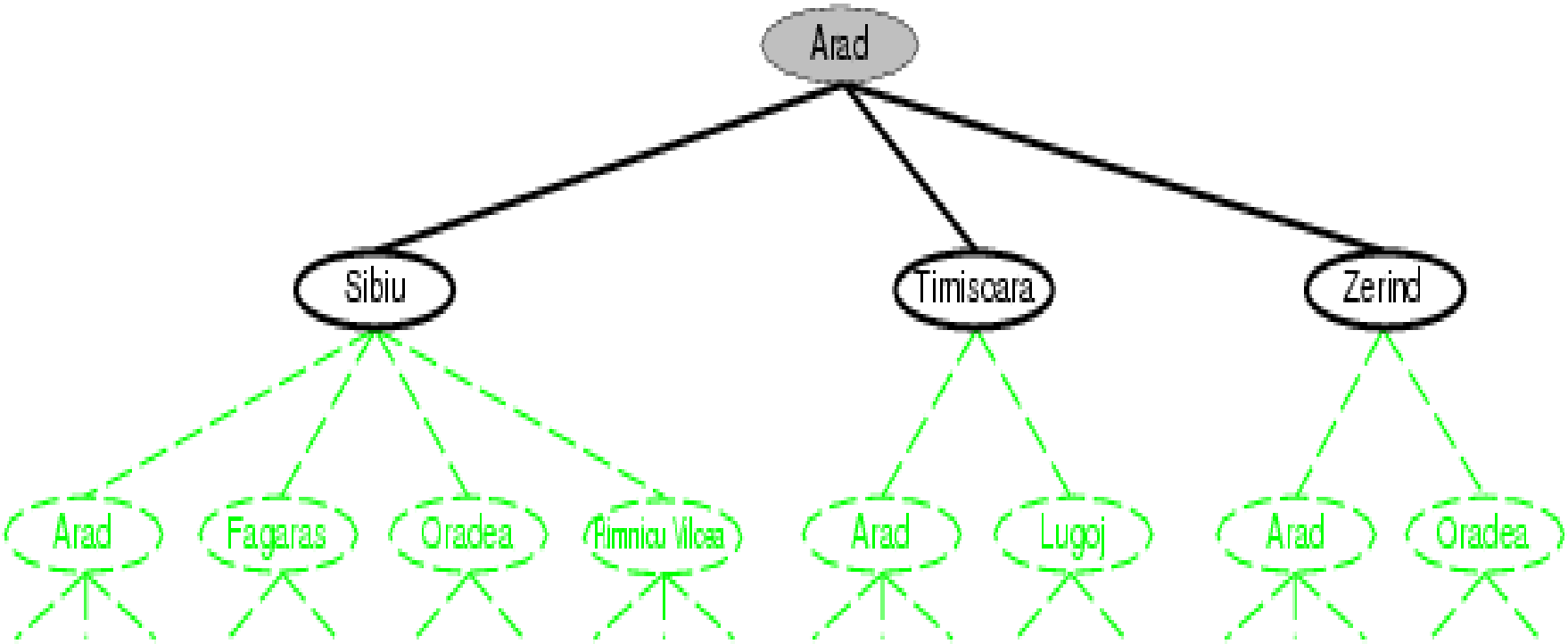
Tree search example (cont.)

The initial state



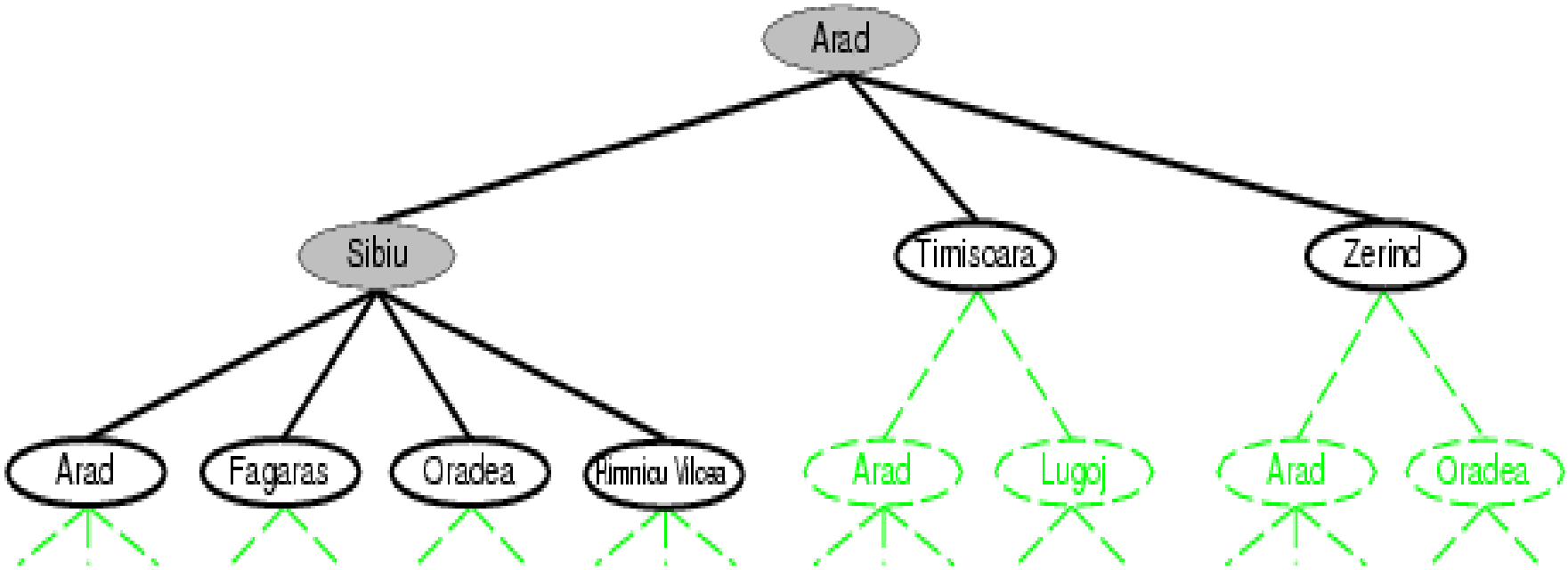
Tree search example (cont.)

After expanding Arad



Tree search example (cont.)

After expanding Sibiu



Implementation: general tree search

General tree search algorithm

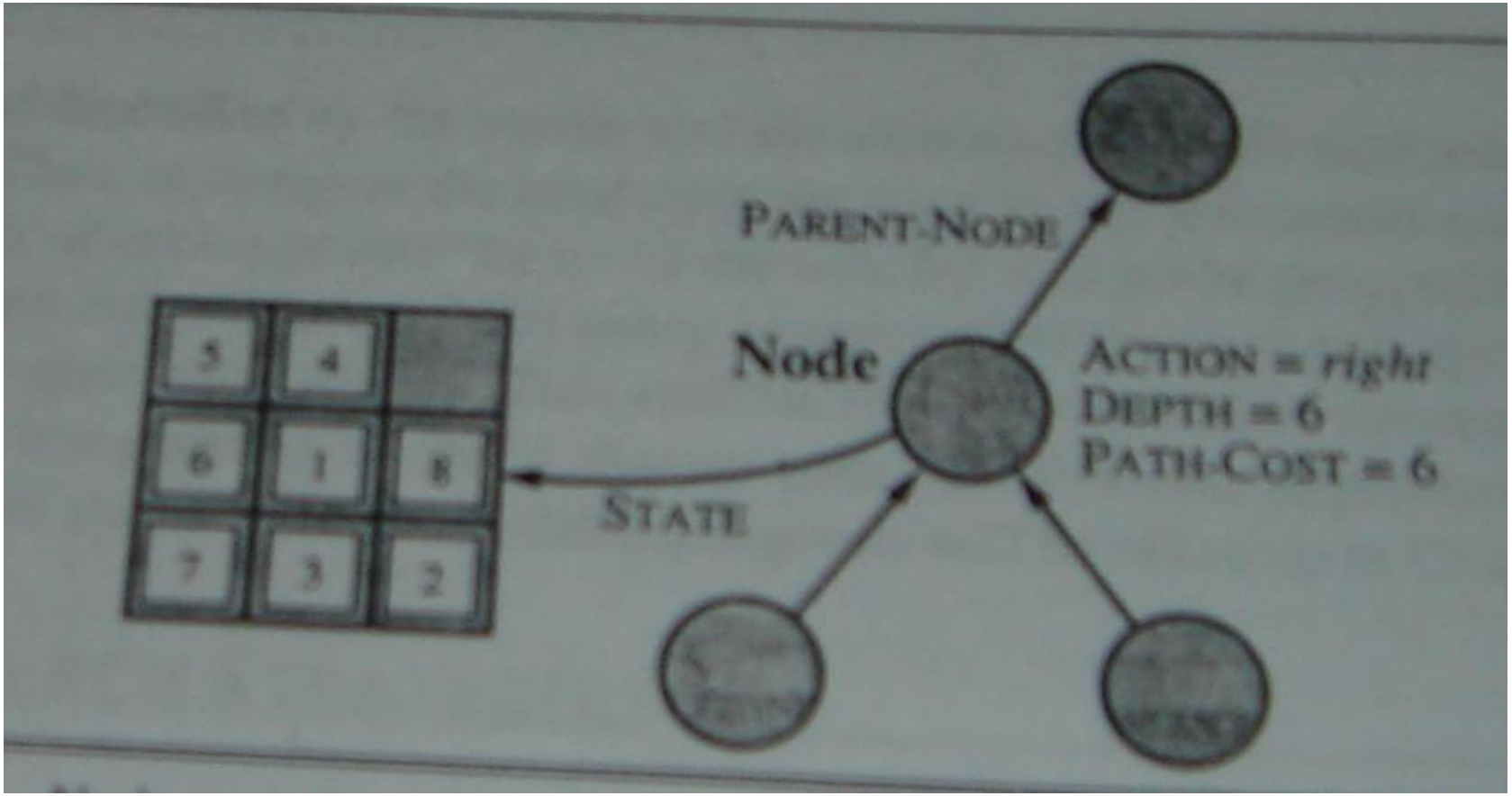
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)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

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- Nodes are the data structures from which the search tree is constructed. Each node has a parent, a state, and various bookkeeping fields, such as action, path cost, depth.
- Depth represents the number of steps along the path from the initial state.
- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.
-

Implementation: states vs. nodes



Nodes are the data structures from which the search tree is constructed. Each node has a parent, a state, and various bookkeeping fields, such as action, path cost, depth. Arrows point from child to parent

Search strategies and their performance evaluation

- A search strategy is defined by picking the order of node expansion

- Search algorithm's performances are evaluated in the following four ways:
  - completeness: Does it always find a solution if one exists?
  - time complexity: How long does it take to find a solution?
  - space complexity: How much memory is needed to perform the search? That means the maximum number of nodes in memory to perform the search.
  - 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

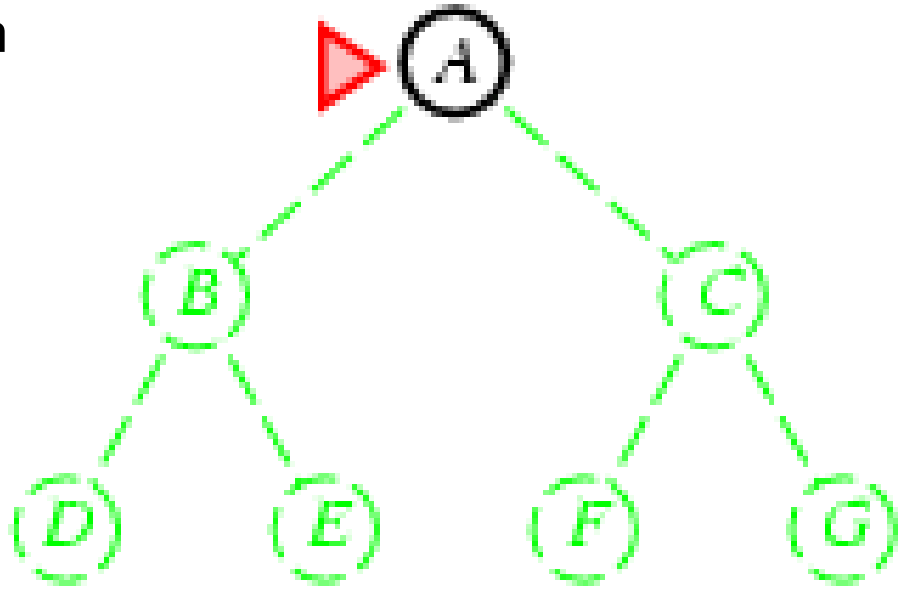
- Uninformed (blind) search strategies use only the information available in the problem definition. All they can do is generate successor and distinguish a goal state from a non-goal state.
- Strategies that know whether one non-goal state is 'more promising' than another are called informed or heuristic search strategies.

Uninformed (blind) search strategies are of following:

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

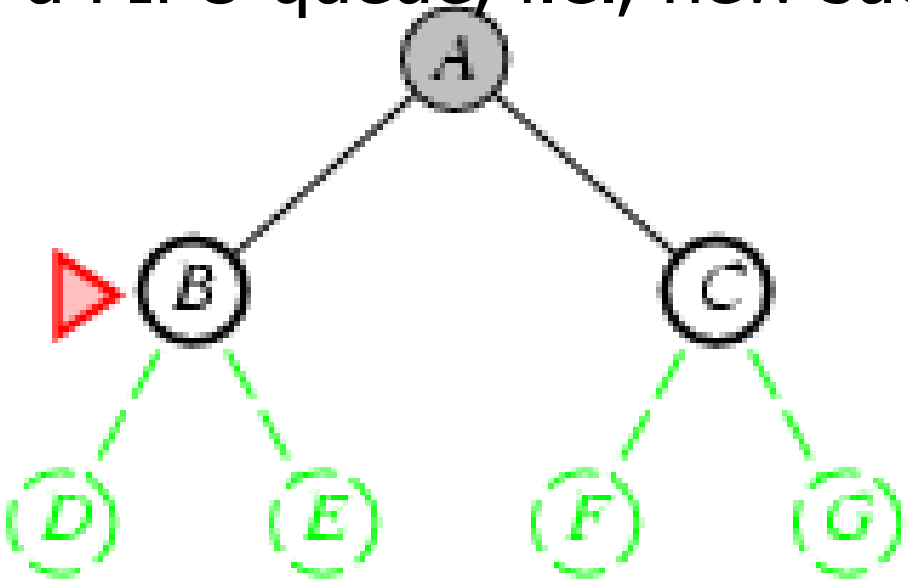
Breadth-first search

- Expand shallowest unexpanded node
-
- Implementation:
  - Root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
  - In general all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
  - fringe is a w successors go at end
  -



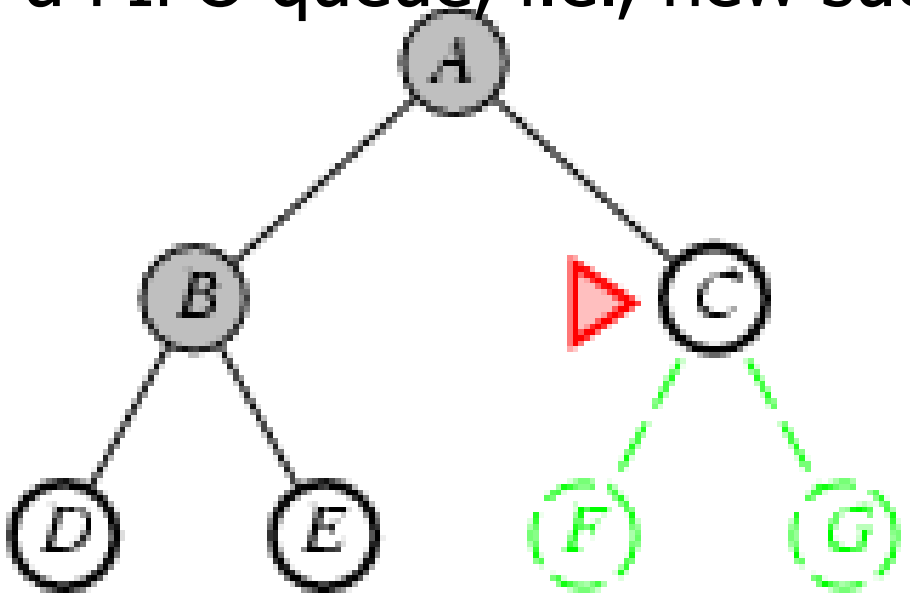
Breadth-first search

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



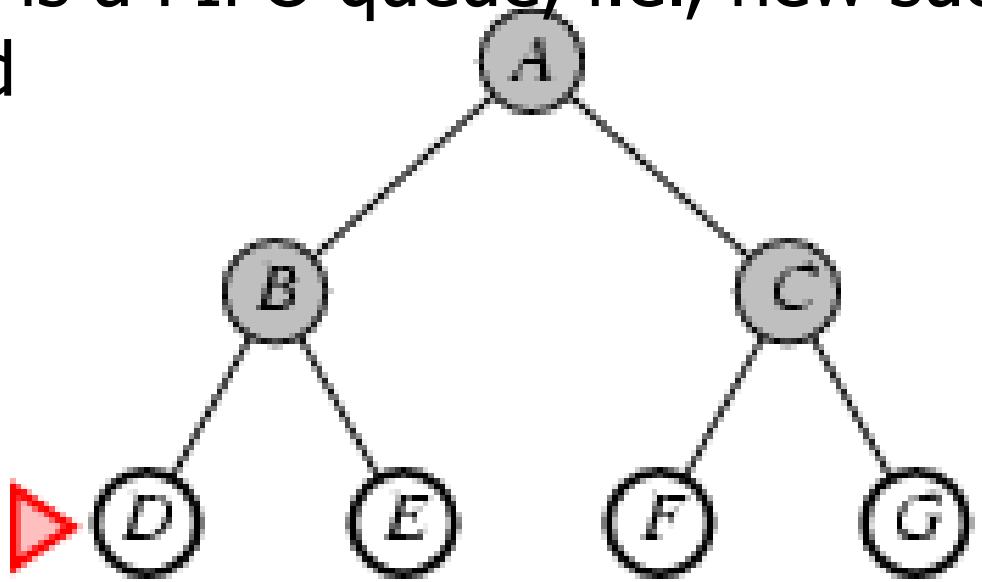
Breadth-first search

- 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

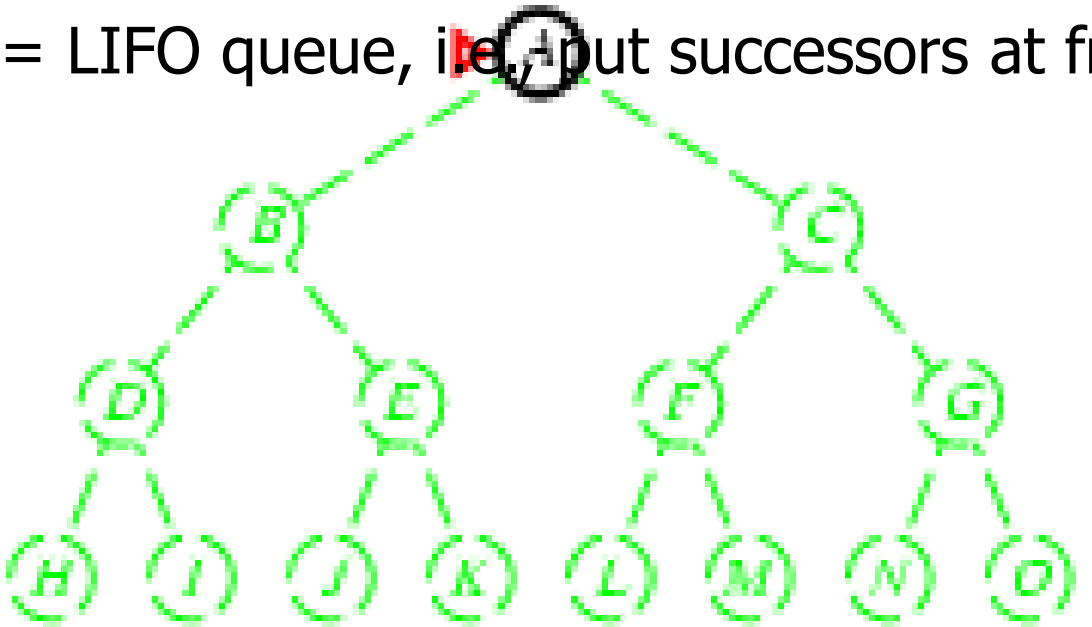
- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

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? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
-

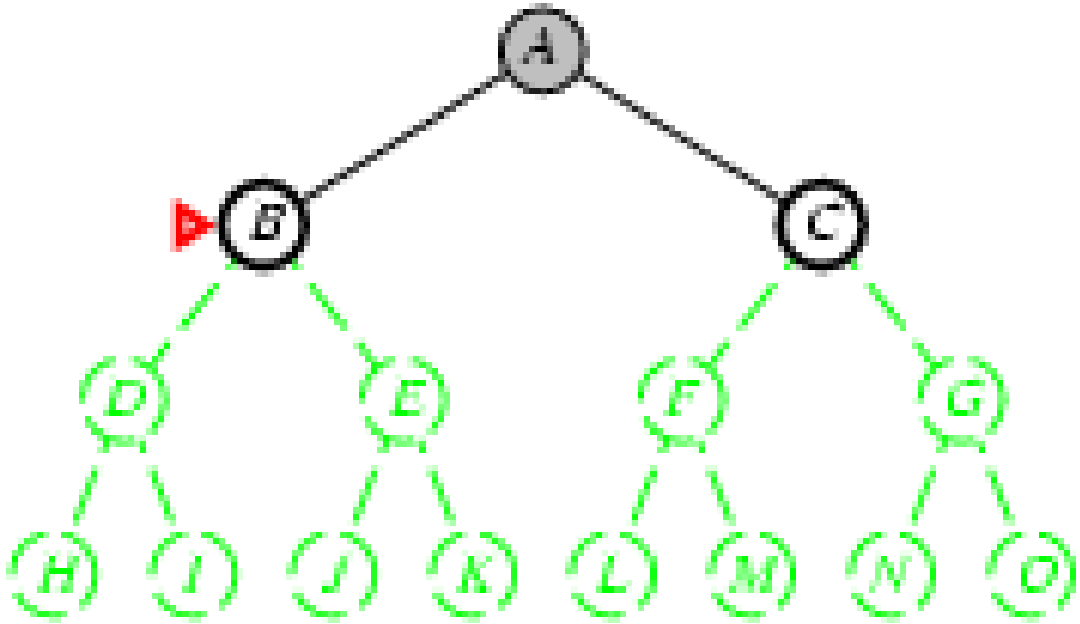
Depth-first search

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



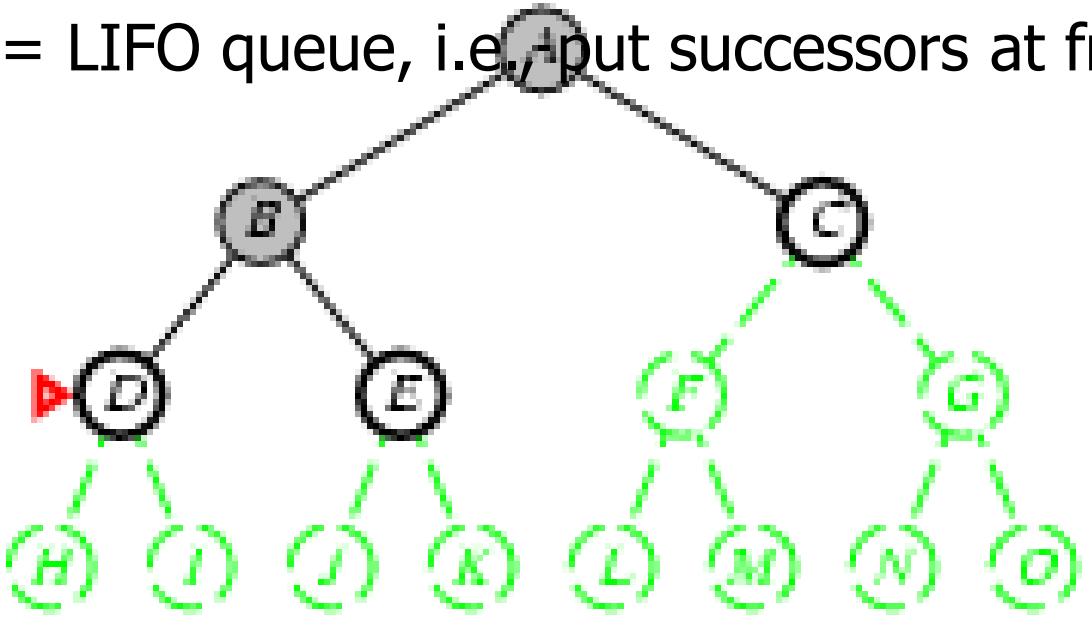
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* front
  -



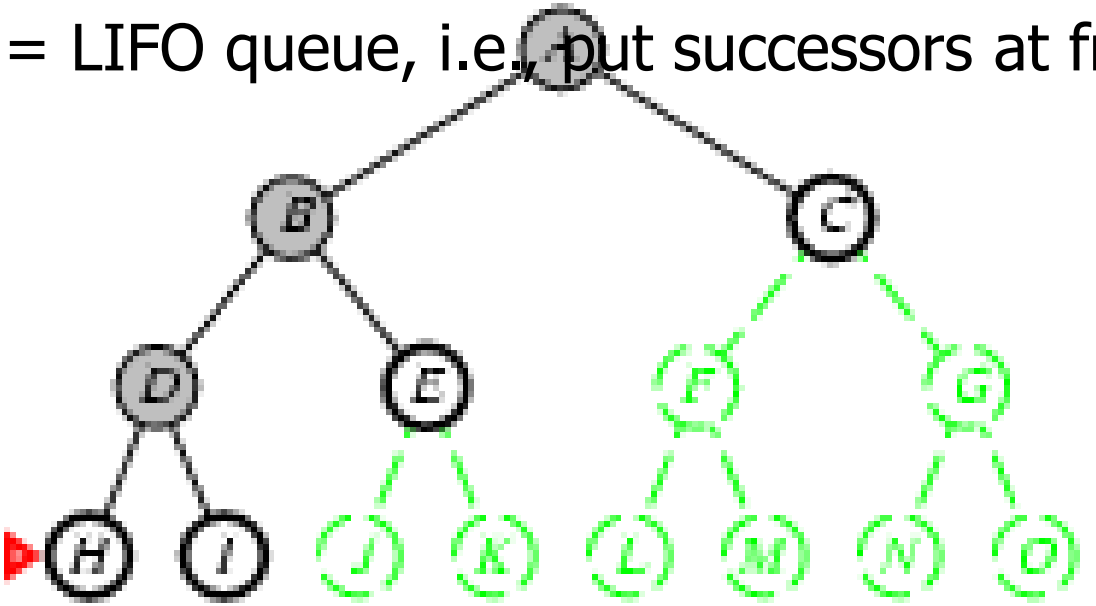
Depth-first search

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



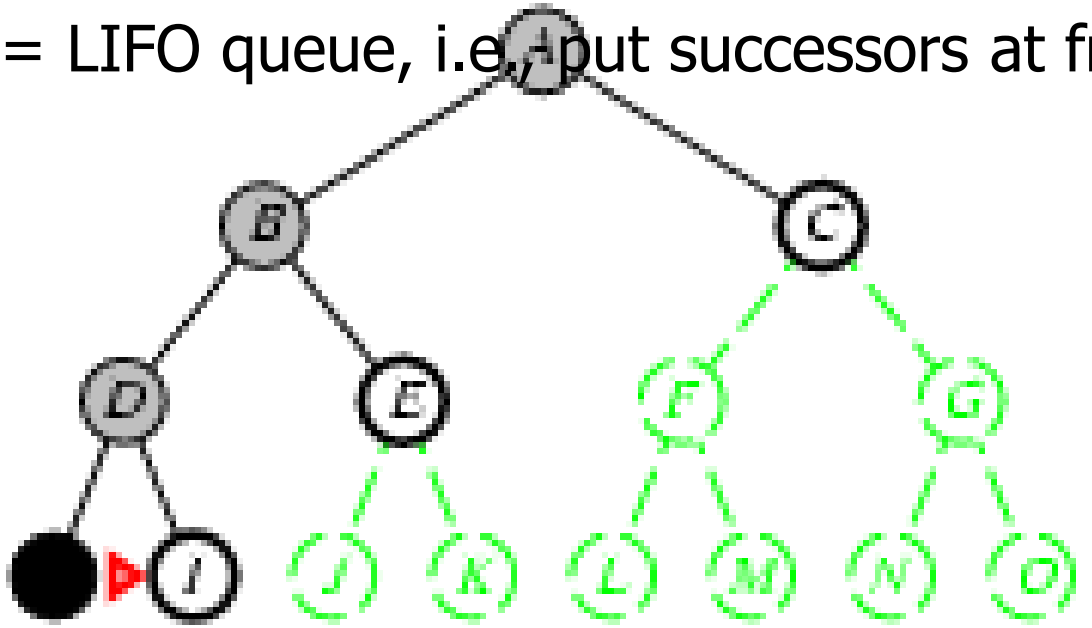
Depth-first search

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



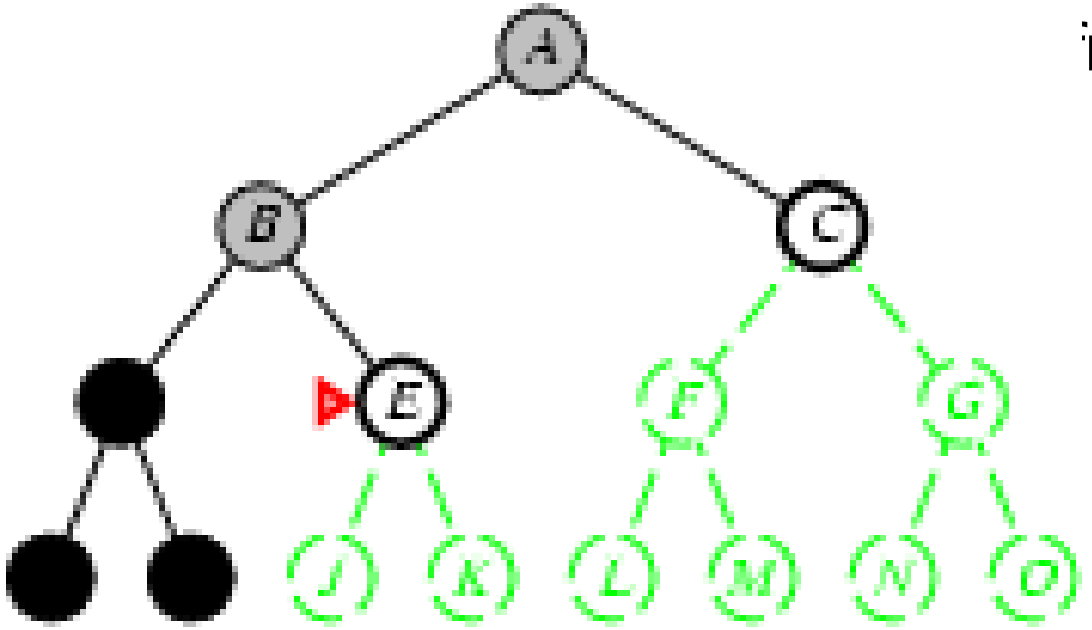
Depth-first search

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



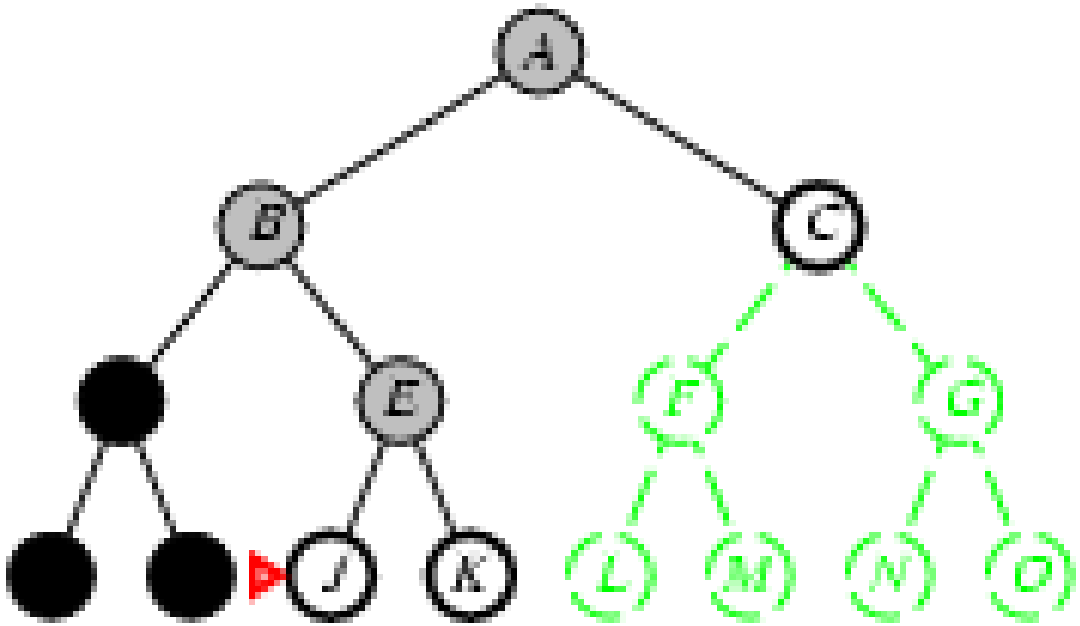
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* front
  -



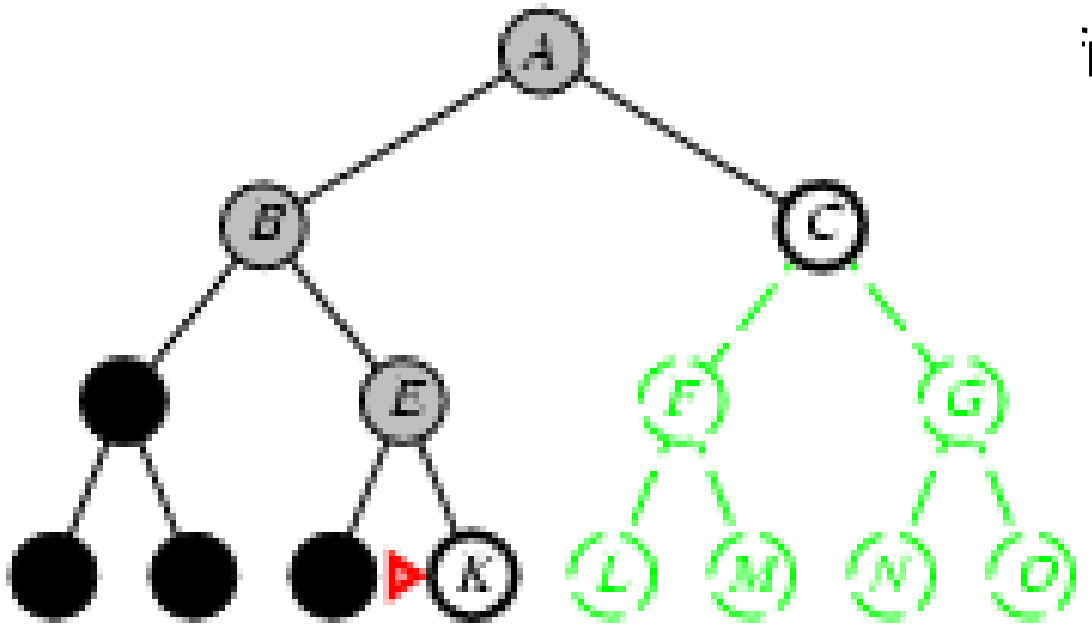
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* front
  -



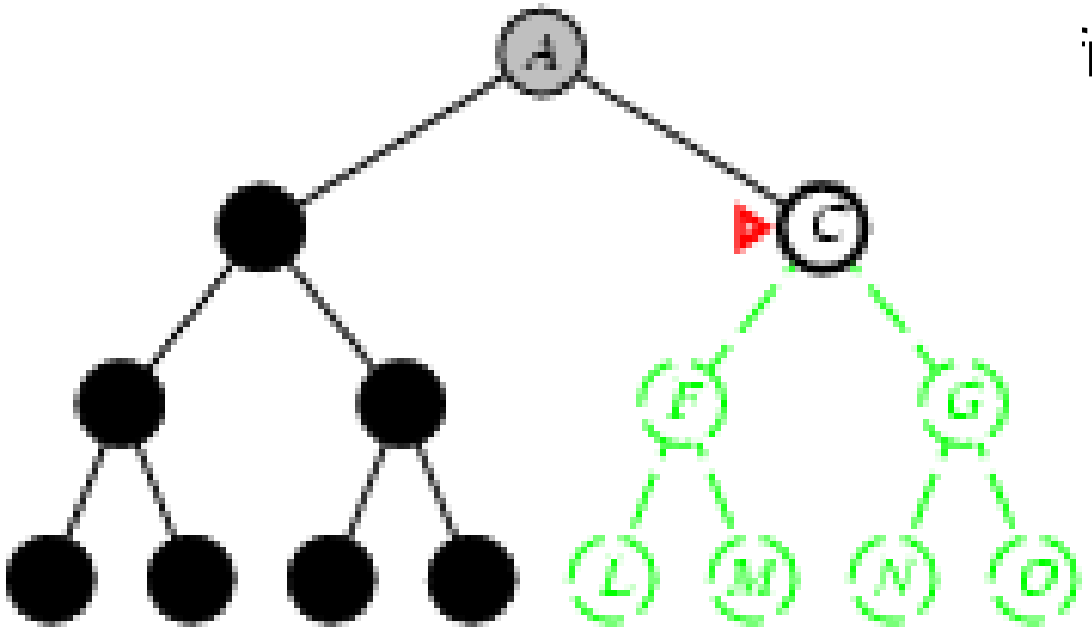
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* front
  -



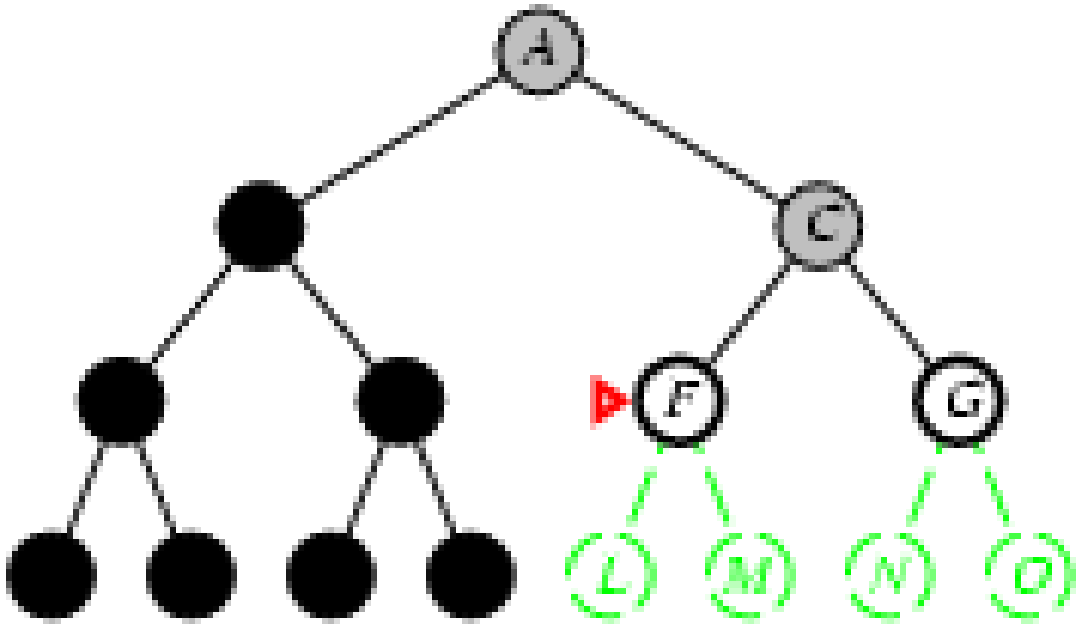
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* front
  -



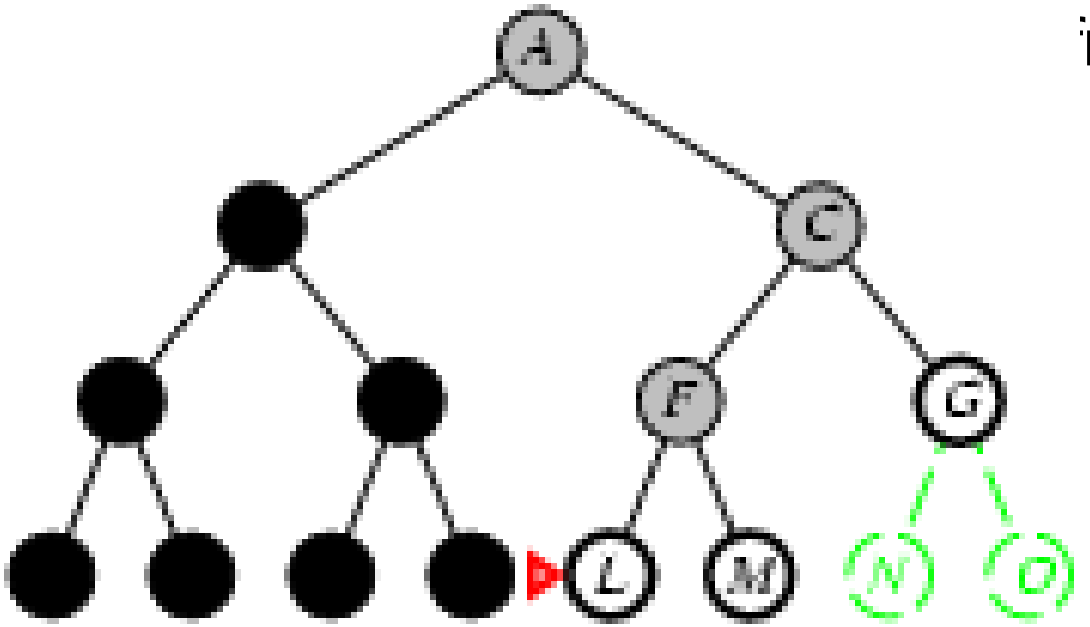
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* ront
  -



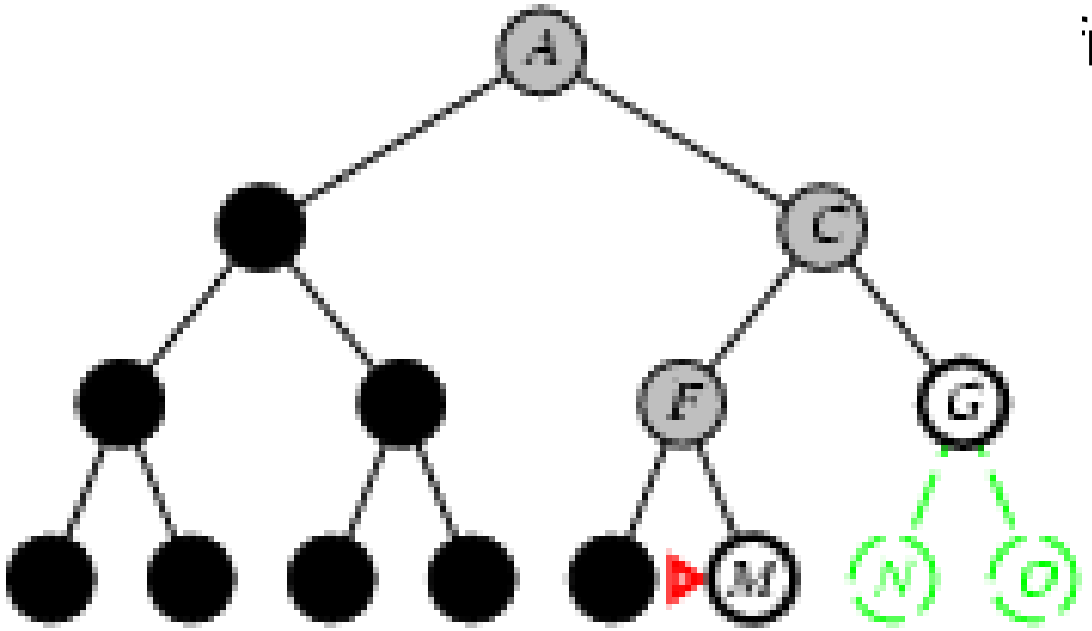
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* front
  -



Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* front
  -



Properties of depth-first search

- 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

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

Depth-limited search

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

- R

```
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 SOLUTION(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 depth-first search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

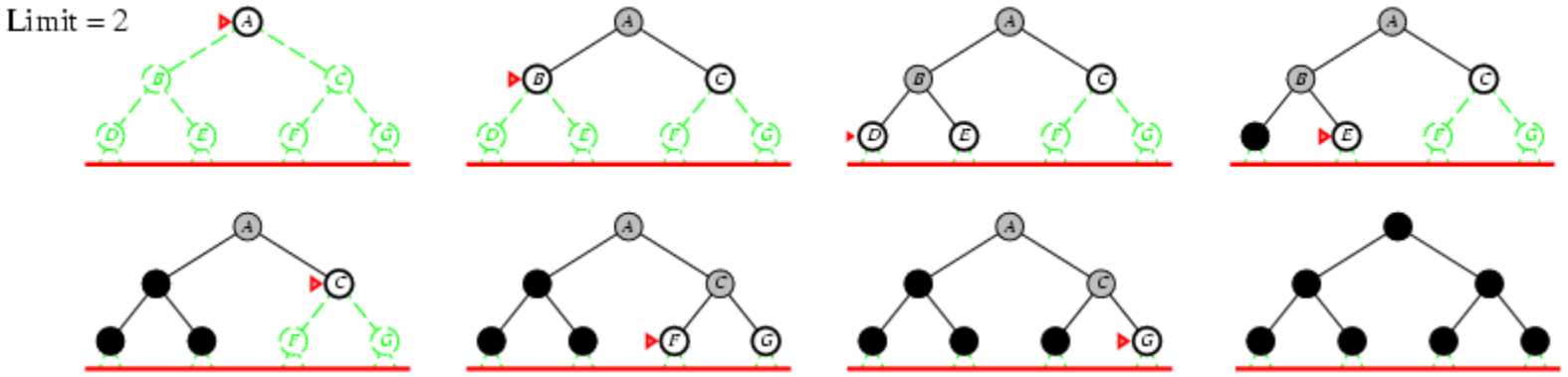
Iterative deepening depth-first search find the best depth limit by gradually increasing the limit.

Iterative deepening search /=0

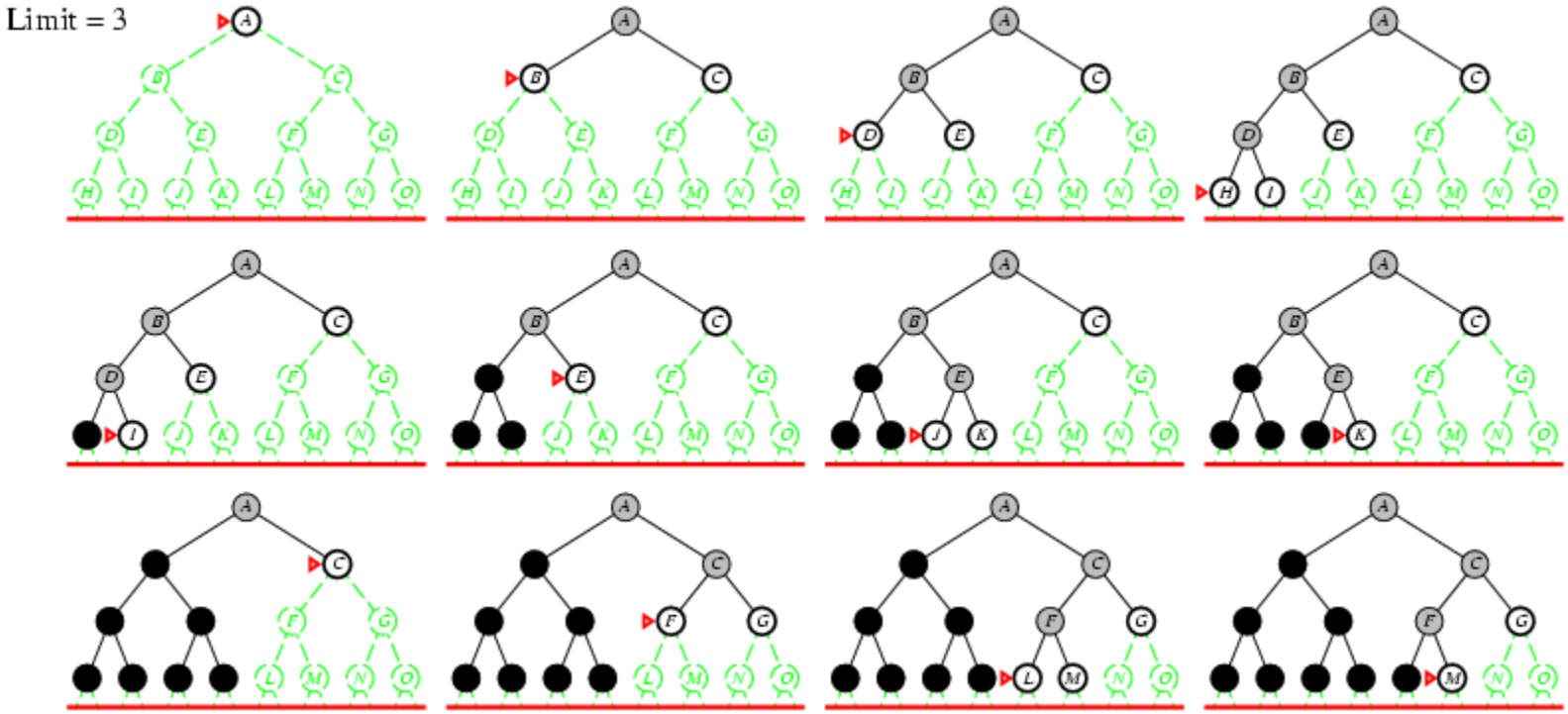
Limit = 0

Iterative deepening search /=1

Iterative deepening search /=2



Iterative deepening search /=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*:

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

- 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

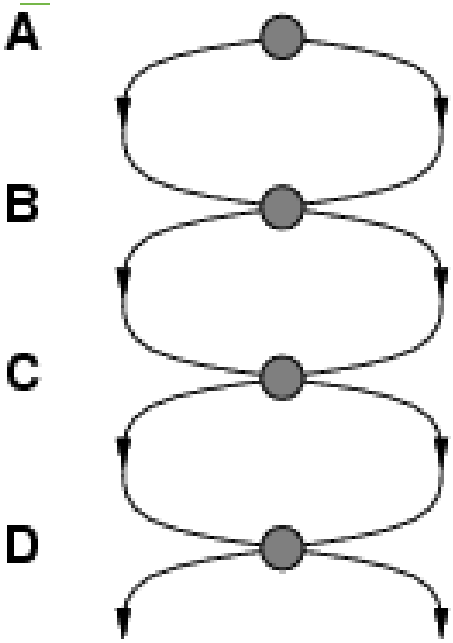
Summary of algorithms

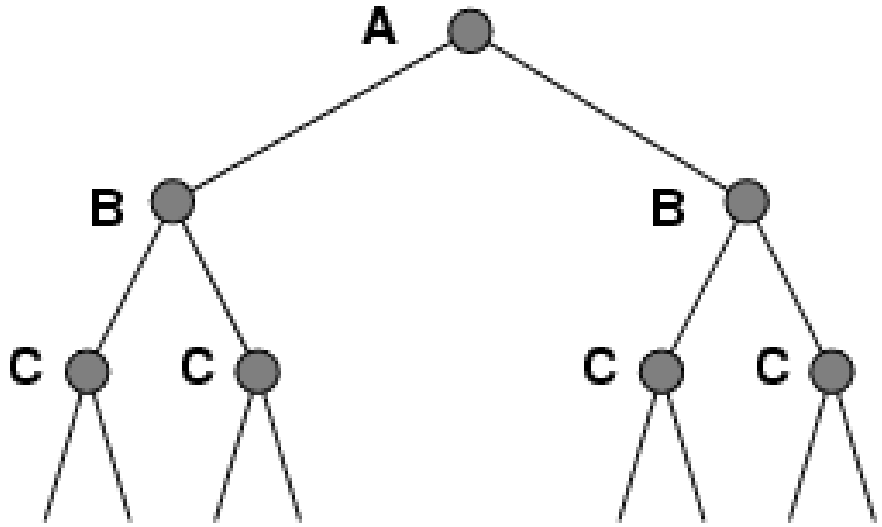
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Evaluation of search strategies. *b* is the branching factor; *d* is the depth of the shallowest solution; *m* is the maximum depth of the search tree; *l* is the depth limit.

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.

Formal Big-O definition

- Let f and g be functions. We say that $f(x)$ is $O(g(x))$ if there are constants c and k such that

  $$|f(x)| \leq C\,|g(x)|$$

  whenever $x > k$

Big- O notation expresses the upper bound of growth function.

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c,k$: $\forall n>k$: $30n+8 \leq cn$.
    - Let $c=31$, $k=8$. Assume $n>k=8$. Then $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$.
- Show that n^2+1 is $O(n^2)$.
  - Show $\exists c,k$: $\forall n>k$: $n^2+1 \leq cn^2$.
    - Let $c=2$, $k=1$. Assume $n>1$. Then $cn^2 = 2n^2 = n^2+n^2 > n^2+1$, or $n^2+1< cn^2$.

Formal Big-Omega definition

- Let *f* and *g* be functions. We say that *f*(*x*) is $\Omega(g(x))$ if there are constants *c* and *k* such that

$$|f(x)| \geq C|g(x)|,$$

$$\text{whenever } x > k.$$

Big- Ω notation expresses the lower bound of growth function. It is introduced by Donald Knuth in the 1970's.

Formal Big-Theta definition

- Let f and g be functions. We say that $f(x)$ is $\Theta(g(x))$ if f(x) is O($g(x)$) and f(x) is $\Omega(g(x))$.

  Big-Θ notation expresses both an upper and a lower bound on the size of $f(x)$ relative to a reference function $g(x)$.

  It is introduced by Donald Knuth in the 1970's.

Names for some orders of growth

- $O(1)$ Constant
- $O(\log_c n)$ Logarithmic (same order $\forall c$)
- $O(\log^c n)$ Polylogarithmic (With c a constant.)
- $O(n)$ Linear
- $O(n^c)$ Polynomial (for any c)
- $O(c^n)$ Exponential (for $c>1$)
- $O(n!)$ Factorial

$f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \le C|g(x)|, \text{ whenever } x > k.$$

Big-O notation expresses the upper bound of growth function.

Function growth rates

Logarithmic scale!

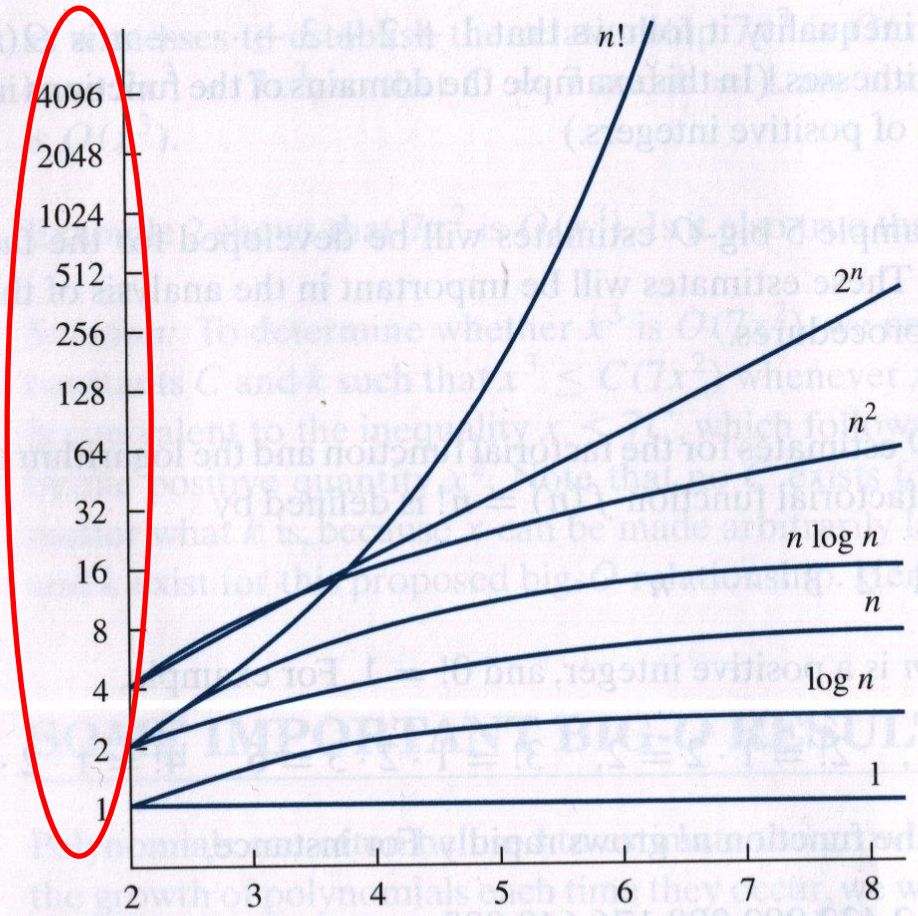


FIGURE 3 **A Display of the Growth of Functions Commonly Used in Big-*O* Estimates.**

Tractable *vs.* intractable

- A problem or algorithm with at most polynomial time complexity is considered *tractable* (or *feasible*). **P** is the set of all tractable problems.
- A problem or algorithm that has complexity greater than polynomial is considered *intractable* (or *infeasible*).
- Note that $n^{1,000,000}$ is *technically* tractable, but really very hard. $n^{\log \log \log n}$ is *technically* intractable, but easy. Such cases are rare though.

P *vs.* NP

- Any term of the form n^c, where c is a constant, is a polynomial
  - Thus, any function that is $O(n^c)$ is a polynomial-time function
  - 2^n, $n!$, n^n are not polynomial (NP) functions

- **NP** is the set of problems for which there exists a tractable algorithm for *checking a proposed solution* to tell if it is correct.

- It is "widely believed" that there is no efficient solution to NP complete problems

- If you could solve an NP complete problem in polynomial time, you would be showing that P = NP

- We know that **P⊆NP**, but the most famous unproven conjecture in computer science is that this inclusion is *proper*.
  - *i.e.*, that **P⊂NP** rather than **P=NP.**