Solving Problems: Blind Search

Instructor: **B. John Oommen**

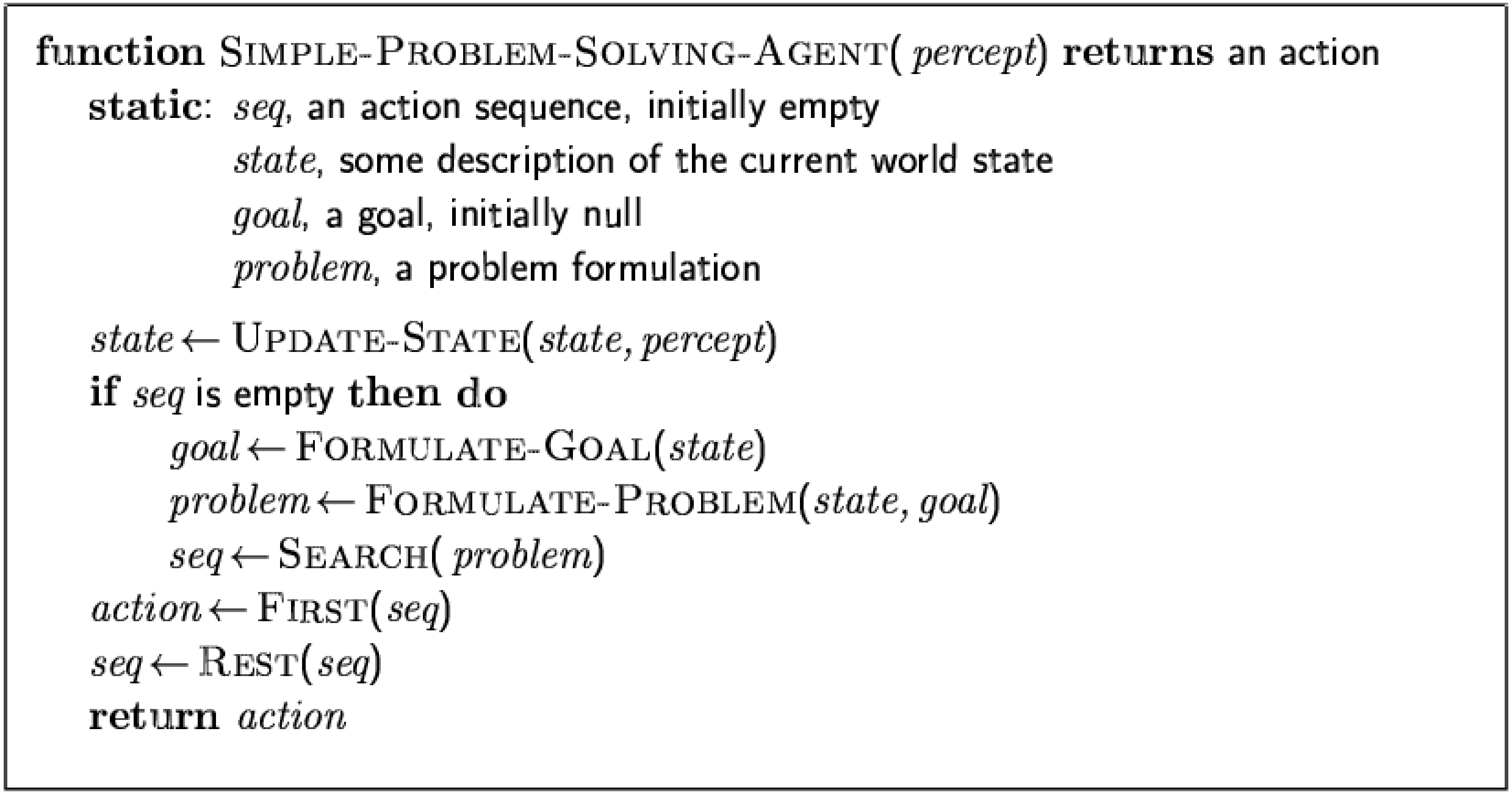
Chancellor’s Professor

Fellow: IEEE ; Fellow: IAPR

School of Computer Science, Carleton University, Canada

The primary source of these notes are the slides of Professor Hwee Tou Ng from Singapore. **I sincerely thank him for this**.

Problem Solving Agents



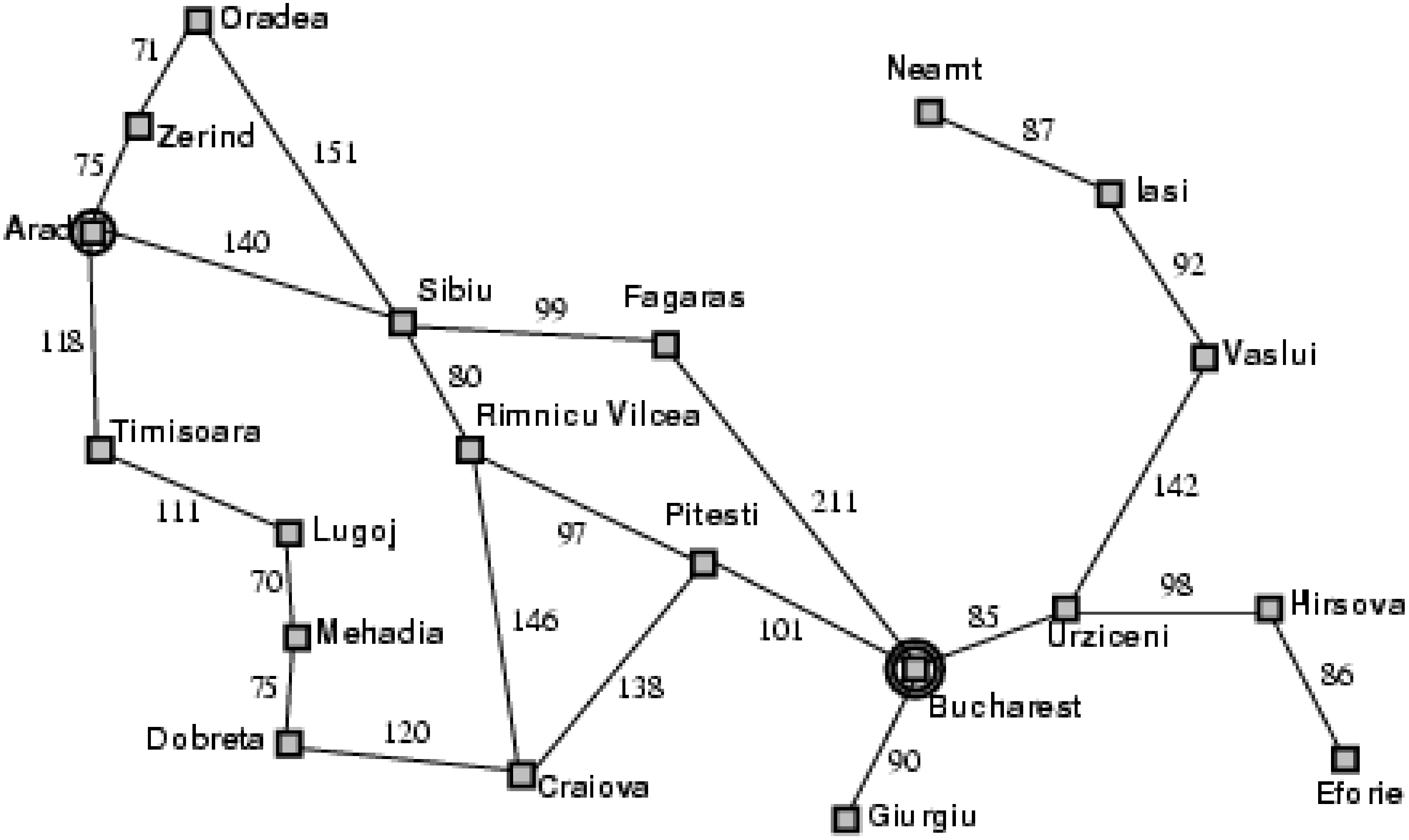
```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

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

Example: Travel in 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: Travel in Romania

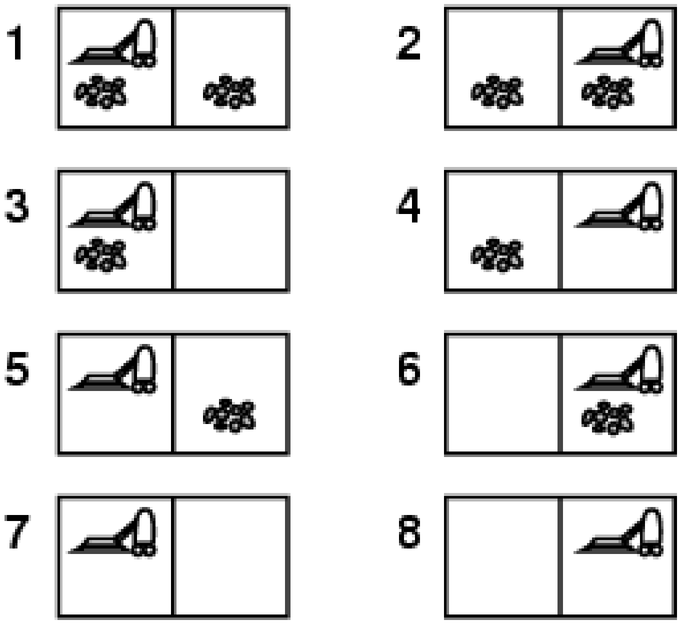


Problem Types

- 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
  - Often interleave: Search, execution
- Unknown state space → Exploration problem

Example: Vacuum World

- Single-state; Start in #5.
  Solution?



Example: Vacuum World

- Single-state

  Start in #5.

  Solution? *[Right, Suck]*

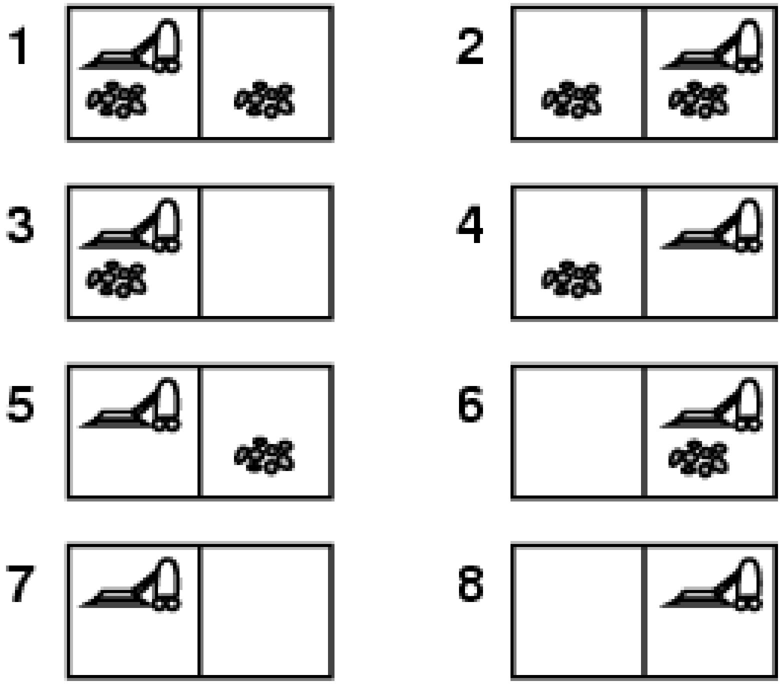
- Sensorless

  Start in {*1,2,3,4,5,6,7,8*}

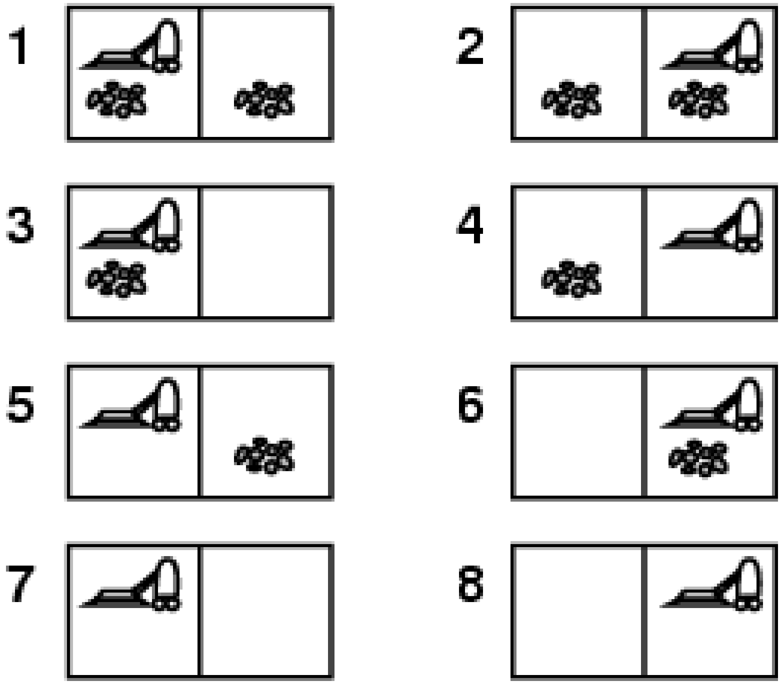
  Right goes to {*2,4,6,8*}

  Solution?

- Now more information



Example: Vacuum World



- Sensorless

  Start in {*1,2,3,4,5,6,7,8*}

  Right goes to {*2,4,6,8*}

  Solution?

  [Right,Suck,Left,Suck]

- Contingency
  - Nondeterministic:

    Suck may dirty a clean carpet
  - Partially observable

    Location, dirt at current location.
  - Percept: *[L, Clean],*

    Start in #5 or #7

  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. This 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

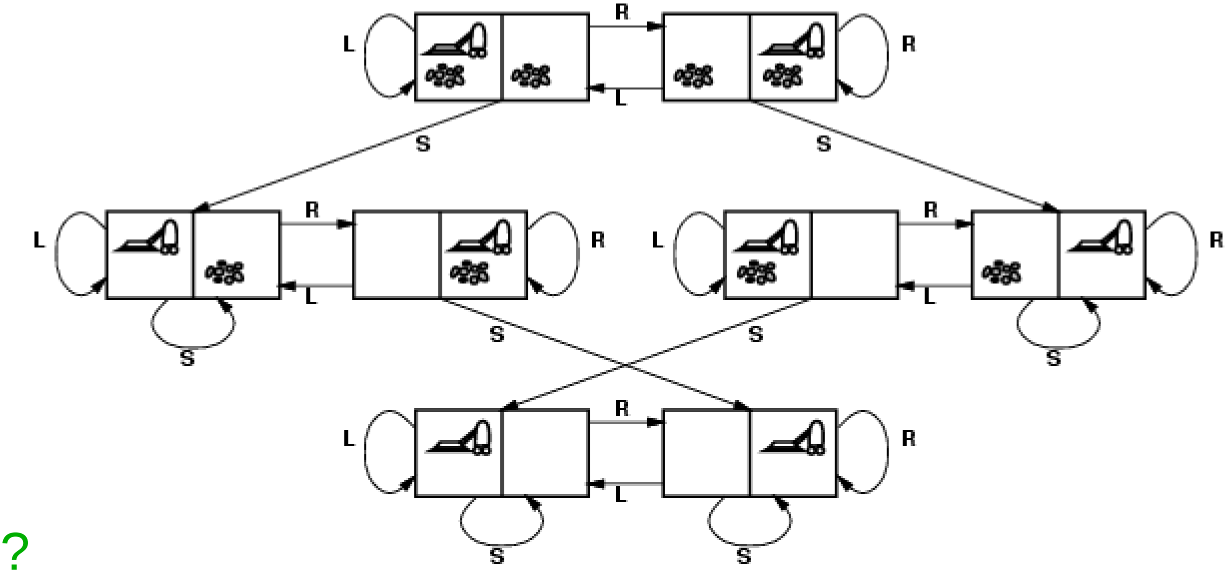
- Solution is a sequence of actions leading from the *initial* to a *goal* state

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”: 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

Vacuum World: State Space Graph



- States?
- Actions?
- Goal test?
- Path cost?

Vacuum World: State Space Graph

- States?

  Integer dirt/robot locations

- Actions?

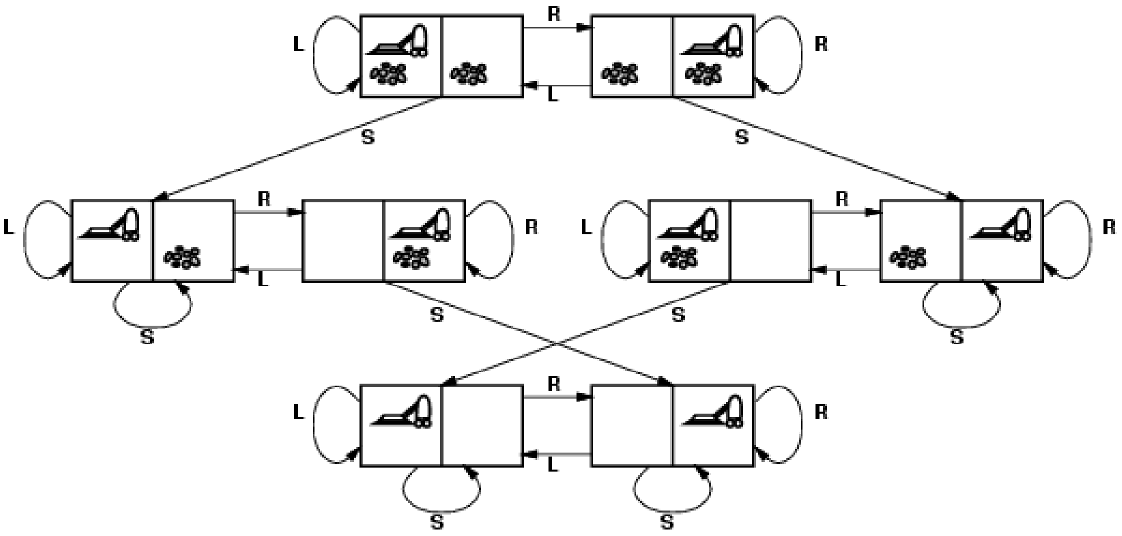
  Left, *Right*, *Suck*

- Goal test?

  No dirt at all locations

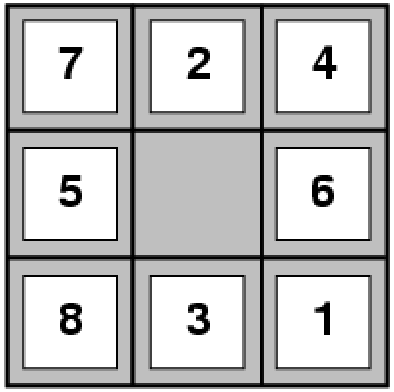
- Path cost?

  1 per action

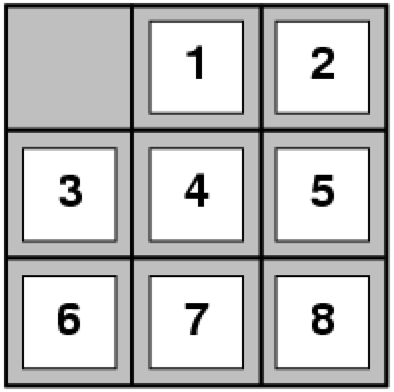


Example: The 8-puzzle

- **States?**
  Locations of tiles
- **Actions?**
  Move blank L/R/U/D
- **Goal test?**
  Goal state (Given: InOrder)
- **Path cost?**
  1 per move; Length of Path
- **Complexity of the problem**
  8-puzzle
    9! = 362,880 different states
  15-puzzle:
    16! =20,922,789,888,000
    10^{13} different states



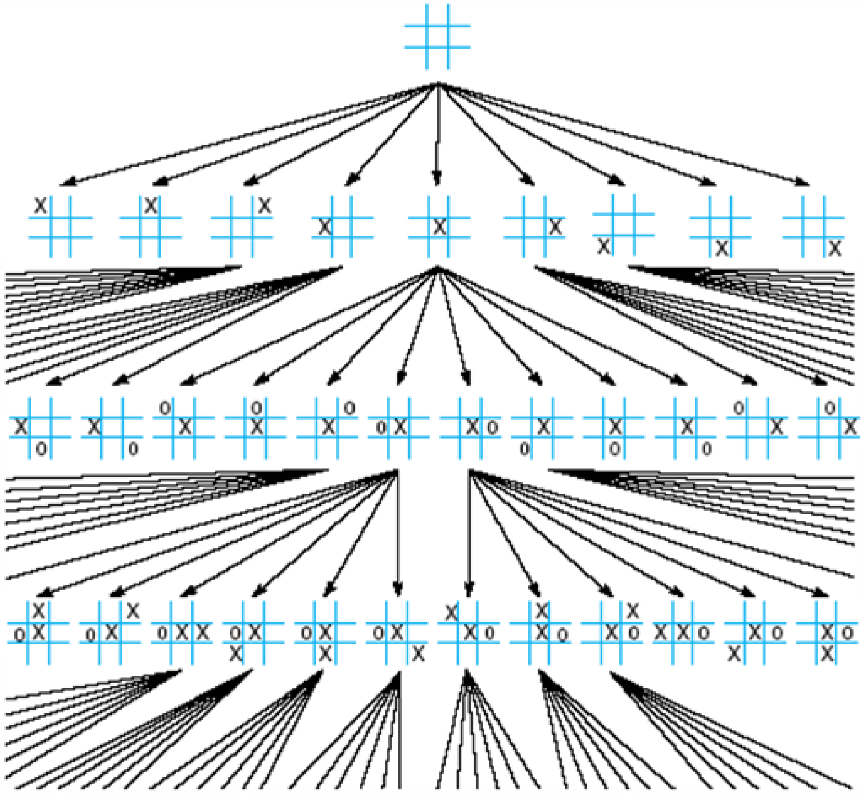
Start State



Goal State

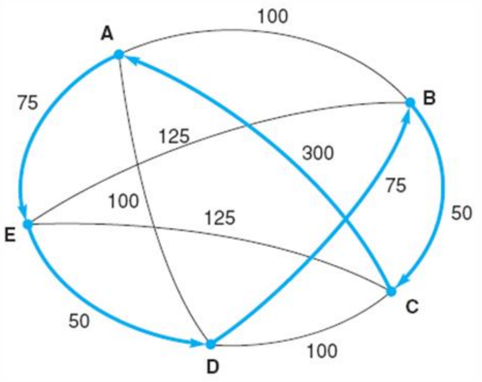
Example: Tic-Tac-Toe

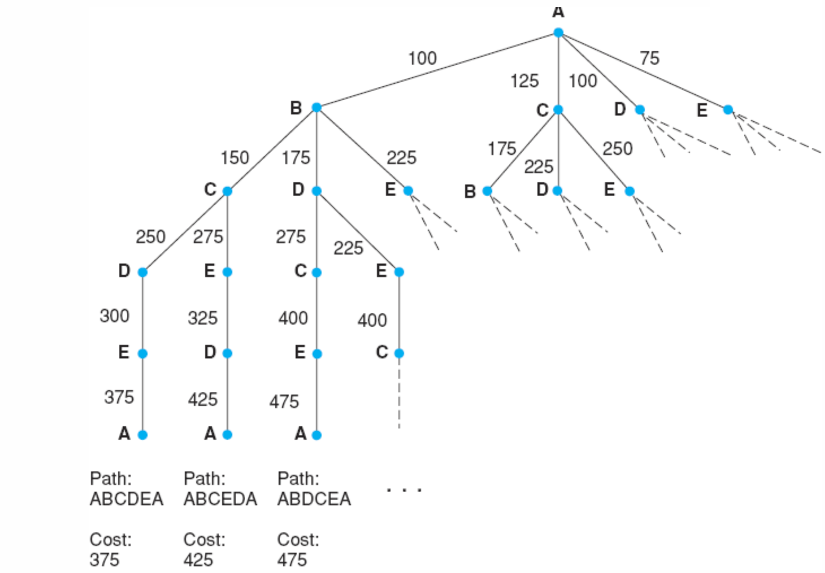
- **States?**
  Locations of tiles
- **Actions?**
  Draw X in the blank state
- **Goal test?**
  Have three X's in a row, column and diagonal
- **Path cost?**
  The path from the Start state to a Goal state gives the series of moves in a winning game
- **Complexity of the problem**
  9! = 362,880 different states
- **Peculiarity of the problem**
  Graph: Directed Acyclic Graph
  Impossible to go back up the structure once a state is reached.



Example: Travelling Salesman

- **Problem**
  Salesperson has to visit 5 cities
  Must return home afterwards
- **States?**
  Possible paths???
- **Actions?**
  Which city to travel next
- **Goal test?**
  Find shortest path for travel
  Minimize cost and/or time of travel
- **Path cost?**
  Nodes represent cities and the
  Weighted arcs represent travel cost
  Simplification
  Lives in city A and will return there.
- **Complexity of the problem**
  (N - 1)! with N the number of cities





State Space

- Many possible ways of representing a problem
- State Space is a natural representation scheme
- A State Space consists of a set of “states”
- Can be thought of as a snapshot of a problem
  - All relevant variables are represented in the state
  - Each variable holds a legal value
- Examples from the Missionary and Cannibals problem (What is missing?)

MMCC | MC

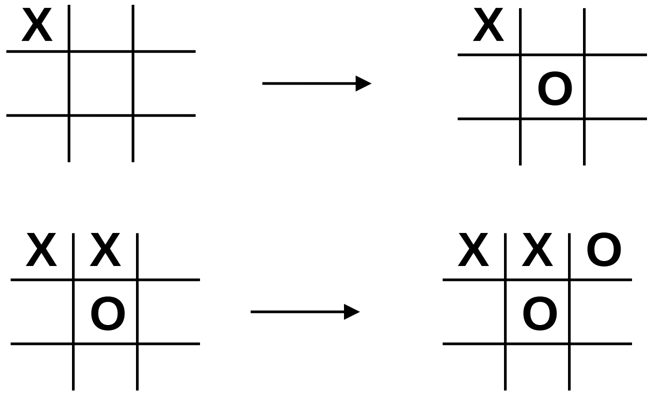
MMC | MCC

MMMCCC |

| MMMCCC

Counter Example: Don’t Use State Space

- Solving Tic Tac Toe using a DB look up for best moves
- e.g. Computer is ‘O’



Each Transition Pair is recorded in DB

Input

Best Move

- Simple but
- Unfortunately most problems have exponential No. of rules

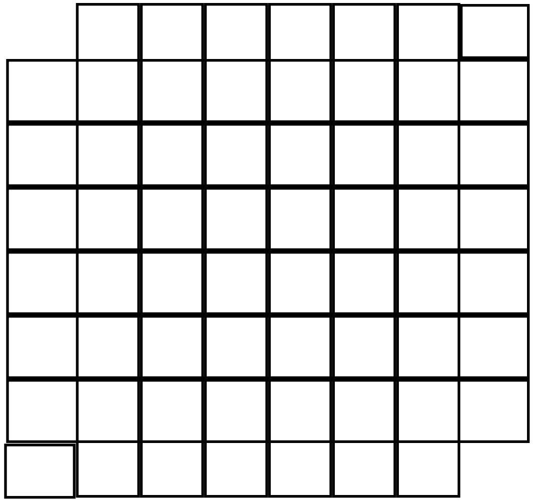
Knowledge in Representation

- Representation of state-space can affect the amount of search needed
- Problem with comparisons between search techniques IF representation not the same
- When comparing search techniques:

  Assume representation is the same

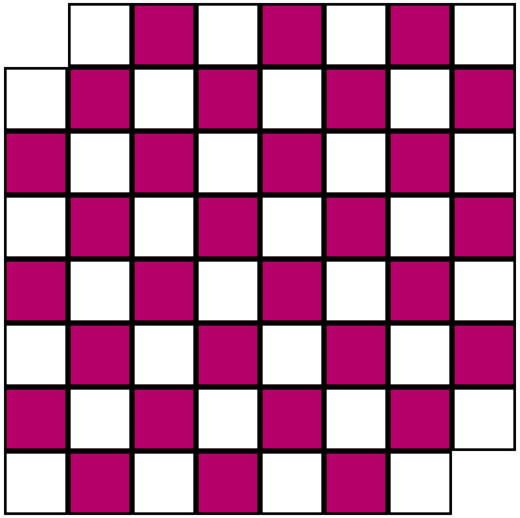
Representation Example

- Mutilated chess board
  - Corners removed
  - From top left and bottom right
- Can you tile this board?
  - With dominoes that cover two squares?



Representation 1

Representation Example: Continued



Number of White Squares= 32

Number of Black Squares= 30

Representation 2

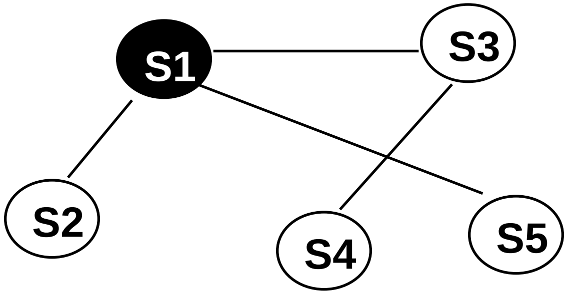
Representation 3

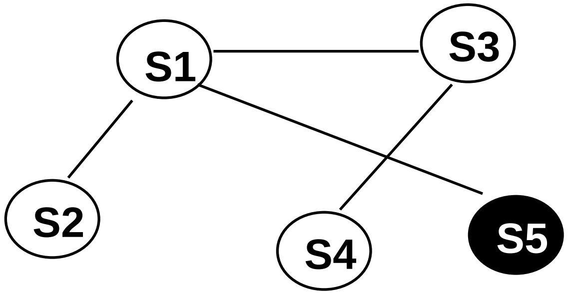
Production Systems

- A set of rules of the form pattern ⟶ action
  - The pattern matches a state
  - The action changes the state to another state
- A task specific DB
  - Of current knowledge about the system (current state)
- A control strategy that
  - Specifies the order in which the rules will be compared to DB
  - What to do for conflict resolution

State Space as a Graph

- Each node in the graph is a possible state
- Each edge is a legal transition
- Transforms the current state into the next state





- Problem solution: A search through the state space

Goal of Search

- Sometimes solution is some final state
- Other times the solution is a path to that end state

Solution as End State:

- Traveling Salesman Problem
- Chess
- Graph Colouring
- Tic-Tac-Toe
- N Queens

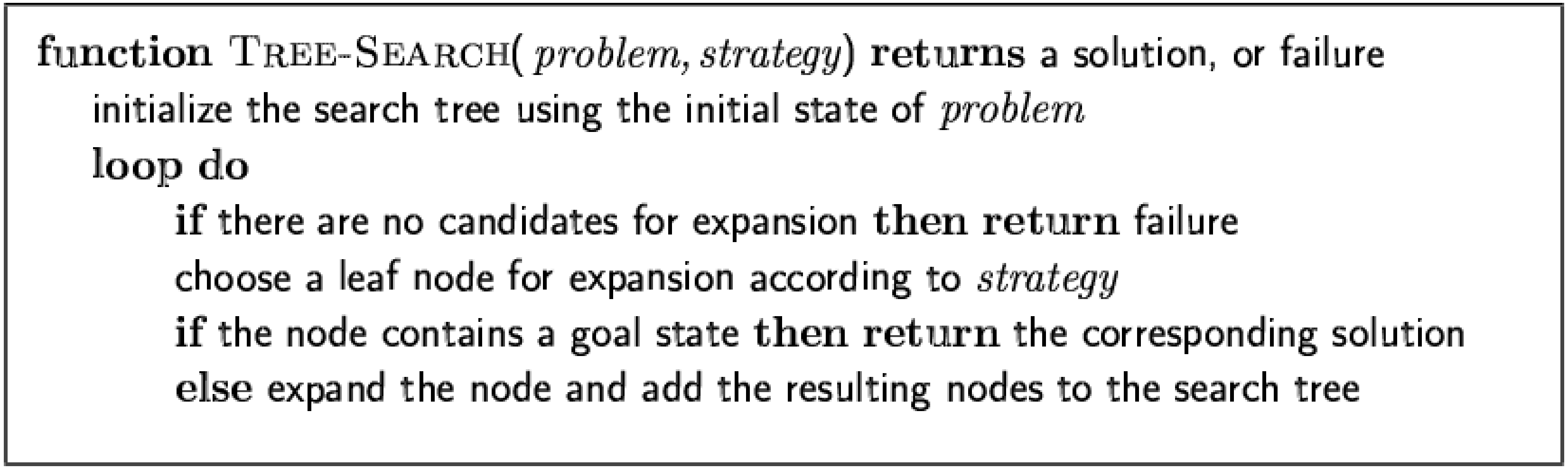
Solution as Path:

- Missionaries and Cannibals
- 8 puzzle
- Towers of Hanoi

Tree Search Algorithms

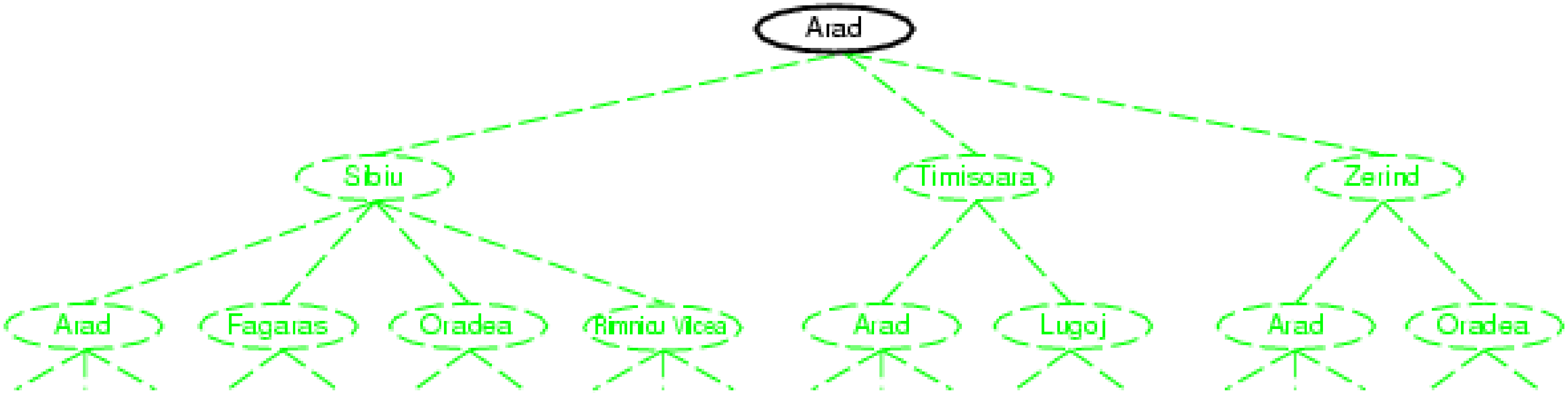
Basic Idea

- Offline, simulated exploration of state space
- Generate successors of already-explored states
- a.k.a. Expanding states

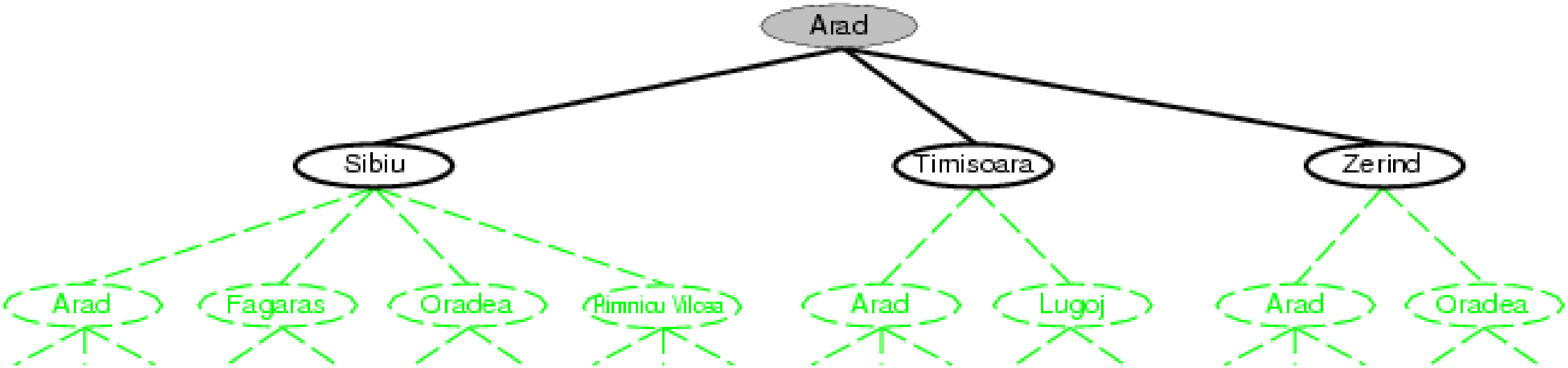


```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
```

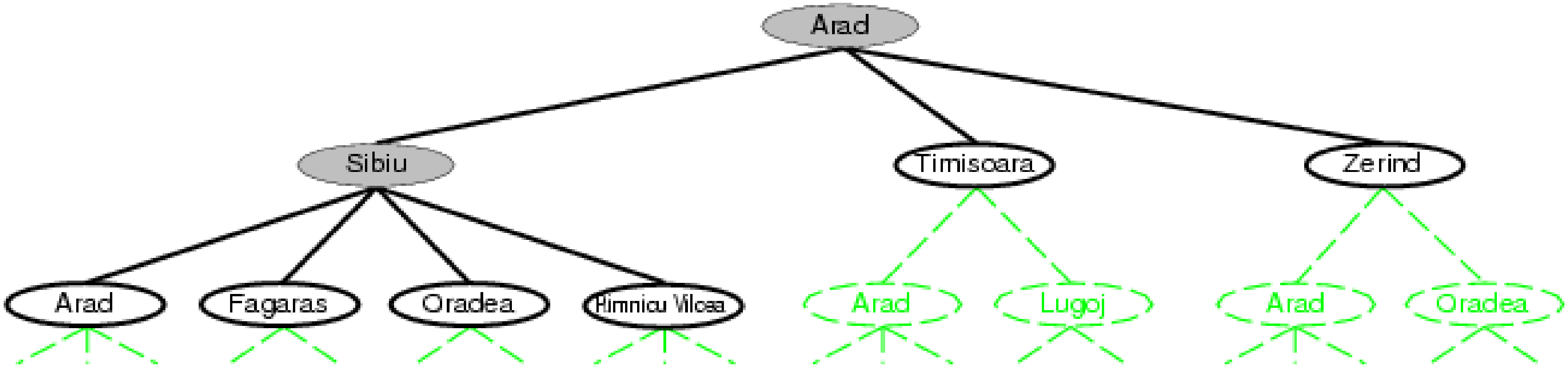
Example: Tree Search



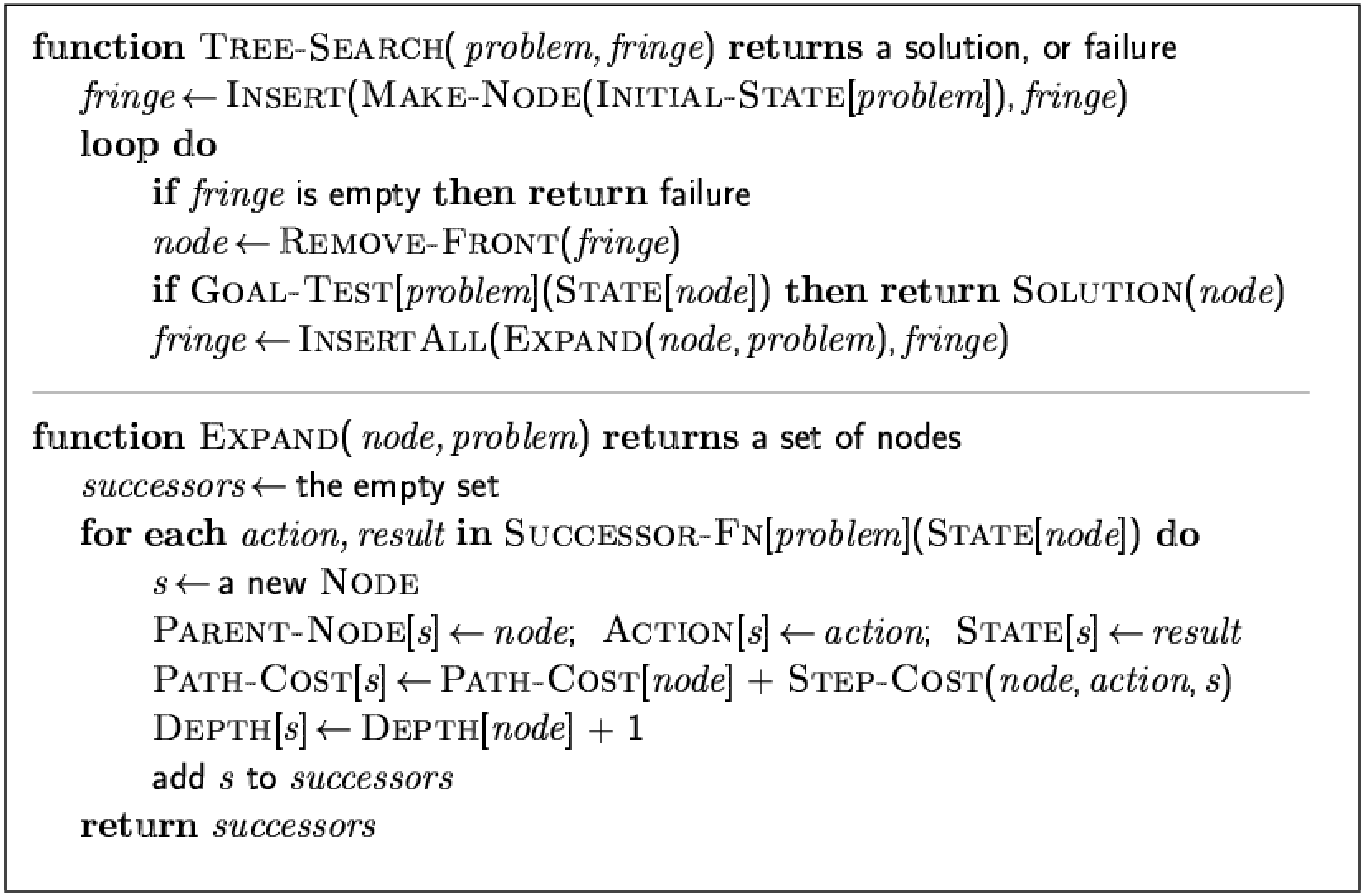
Example: Tree Search



Example: Tree Search



Implementation: General Tree Search

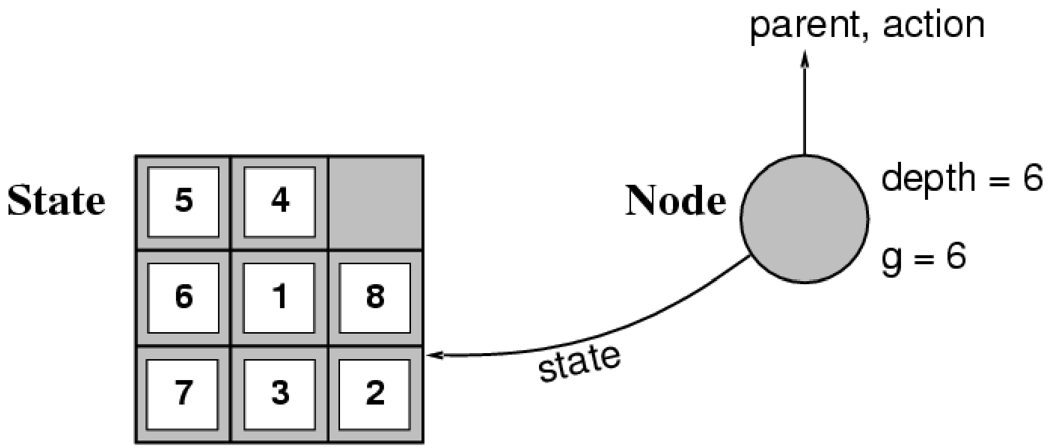


```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        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
- A node is a data structure constituting part of a search tree
- Includes state, parent node, action, path cost *g(x)*, depth



- **`Expand`** function creates new nodes, filling in the various fields
- **`SuccessorFn`** of the problem creates the corresponding states.

Search Strategies

- Search strategy: 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 ∞)

Strategies for State Space Search

- Data-Directed vs. Goal-Directed search
  - Data driven (forward chaining)
  - Goal driven (backward chaining)

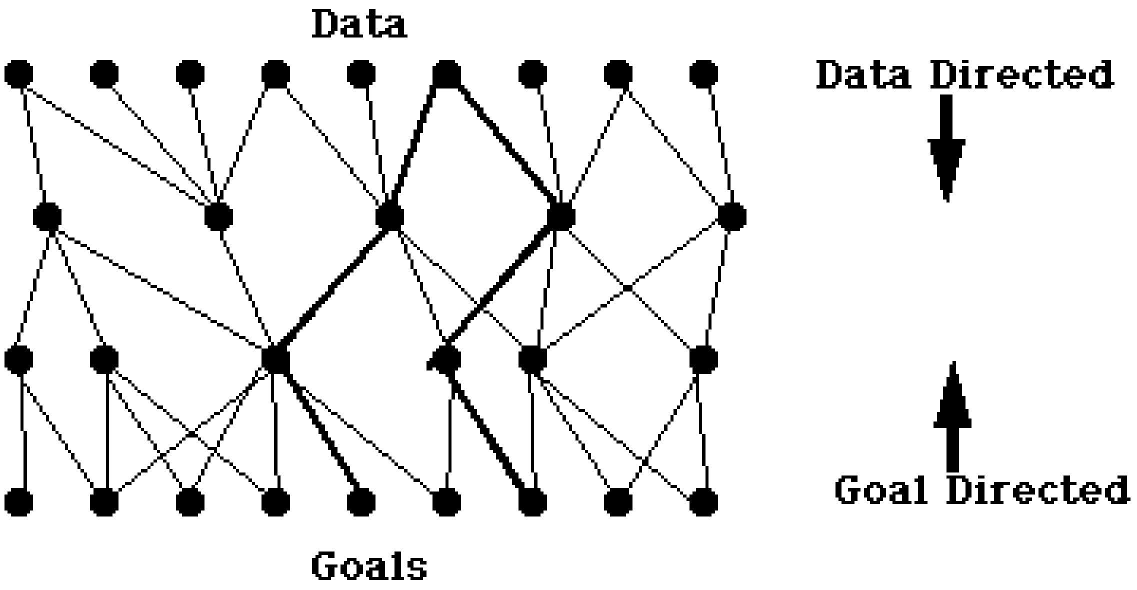
- Data-Directed (Forward Chaining)
  - Start from available data
  - Search for goal

- Goal-Directed (Backward Chaining)
  - Start from goal, generate sub-goals
  - Until arriving at initial state.

- Best strategy depends on problem

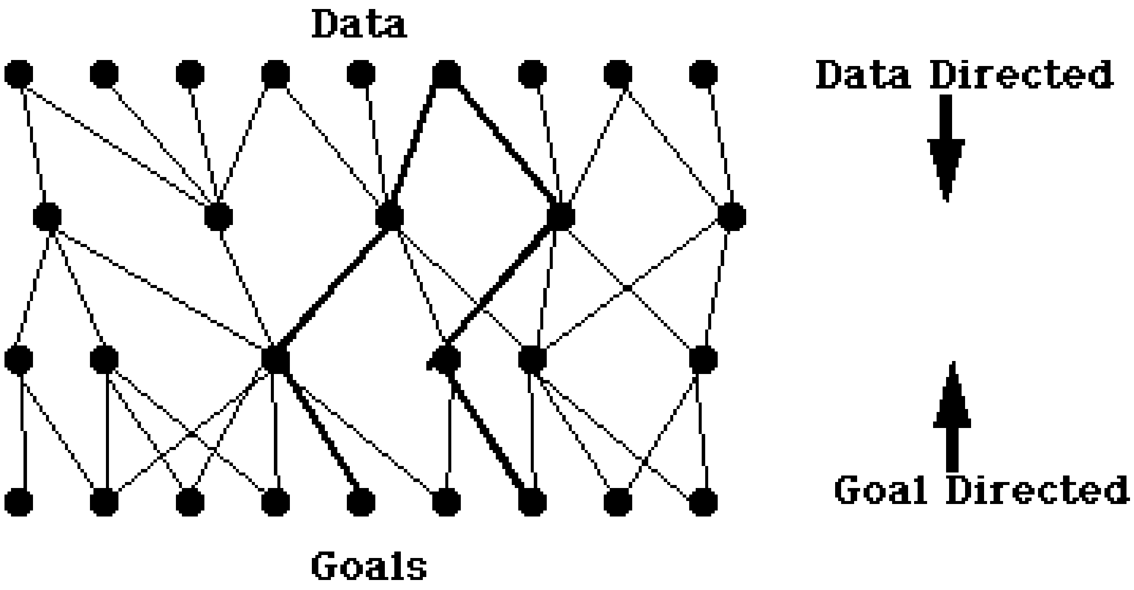
Strategies for State Space Search

- Data-Directed Search (Forward Chaining)
  - Start from available data
  - Search for goal



Strategies for State Space Search

- Goal-Directed (Backward Chaining)
  - Start from goal, generate sub-goals
  - Until you arrive at initial state.



Forward/Backward Chaining

- Verify: I am a descendant of Thomas Jefferson
  - Start with yourself (goal) until Jefferson (data) is reache
  - Start with Jefferson (data) until you reach yourself (goal).
- Assume the following:
  - Jefferson was born 250 years ago.
  - 25 years per generation: Length of path is 10.
- Goal-Directed search space
  - Since each person has 2 parents
  - The search space: Order of 2^{10} ancestors.
- Data-Directed search space
  - If average of 3 children per family
  - The search space: Order of 3^{10} descendents

- So Goal-Directed (backward chaining) is better.
- But both directions yield exponential complexity

Forward/Backward Chaining

- Use the Goal-Directed approach when:
  - Goal or hypothesis is given in the problem statement
  - Or these can easily be formulated
  - There are a large number of rules that match the facts of the problem
  - Thus produce an increasing number of conclusions or goals
  - Problem data are not given but must be acquired by the solver

- Use the Data-Directed approach when:
  - All or most of the data are given in the initial problem statement.
  - There are a large number of potential goals
  - But there are only a few ways to use the facts and given information of a particular problem instance
  - It is difficult to form a goal or hypothesis

Uninformed Search Strategies

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

Backtracking Search

- A method to search the “tree”
- Systematically tries *all* pathes through state space
- In addition: Does not get stuck in cycles

Backtracking Search: Idea

- Principle
  - Keep track of visited nodes
  - Apply recursion to get out of dead ends

- Termination
  - If it finds a goal: *Quit* and return the solution path
  - Also *Quit* if state space is exhausted

- Backtracking
  - If it reaches a dead end, it backtracks
  - It does this to the most recent node on the path having unexamined siblings and continues down one of these branches
  - It requires stack oriented recursive environment

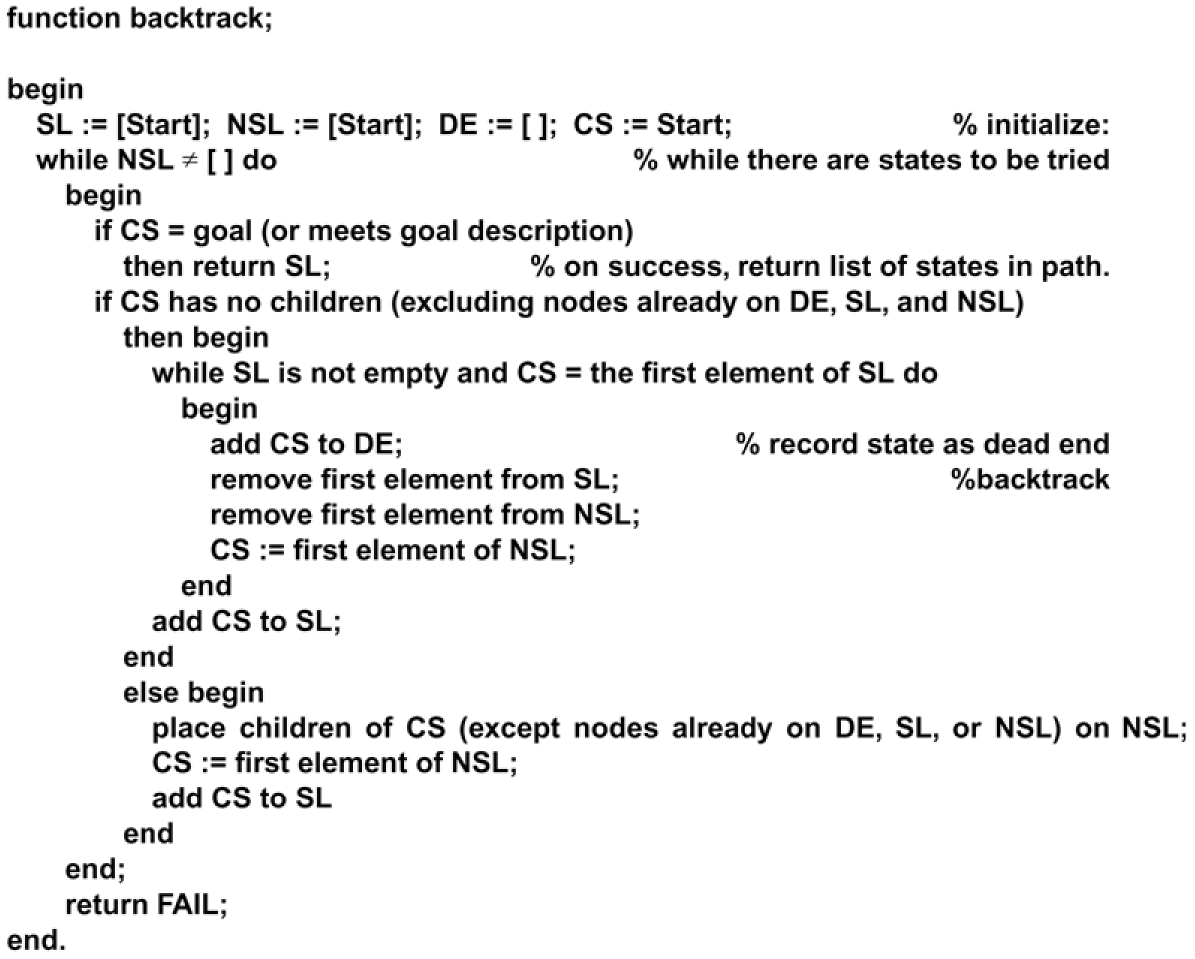
Backtracking Search: Idea

- Details of Backtracking
  - SL (State List):
    - States in current path being tried
    - If Goal is found, SL contains ordered list of states on solution path
  - NSL (New State List)
    - Nodes awaiting evaluation.
    - Nodes: Descendants have not been generated and searched
  - DE (Dead Ends)
    - States whose descendants failed to contain a goal node.
    - If encountered again: Recognized and eliminated from search

Backtracking Search: Idea

- Backtrack is a Data-Directed search
  - Because it starts from the root
  - Then evaluates its descendent children to search for the goal
- Backtrack can be viewed as a Goal-Directed
  - Let the goal be a root of the graph
  - Evaluate descendent back in attempting to find the start (i.e., “root”)
- Backtrack prevents looping by explicit check in NSL

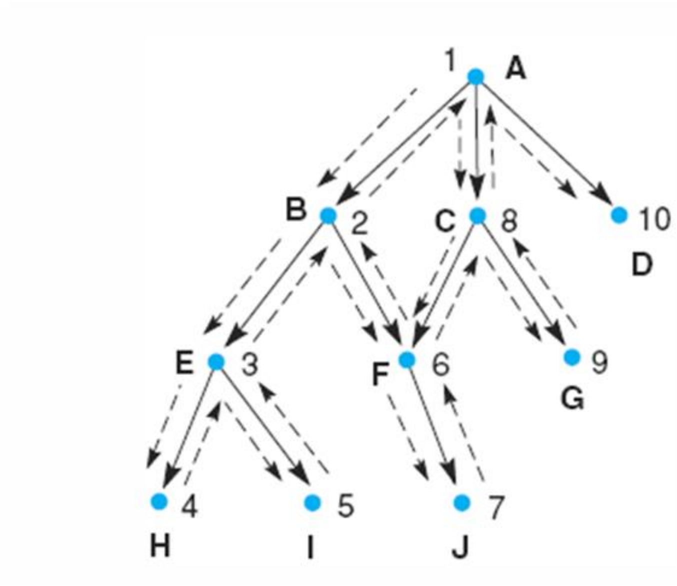
The Backtrack Algorithms



```
function backtrack;

begin
   SL := [Start];  NSL := [Start];  DE := [ ];  CS := Start;                    % initialize:
   while NSL ≠ [ ] do                                     % while there are states to be tried
      begin
         if CS = goal (or meets goal description)
            then return SL;                       % on success, return list of states in path.
         if CS has no children (excluding nodes already on DE, SL, and NSL)
            then begin
               while SL is not empty and CS = the first element of SL do
                  begin
                     add CS to DE;                              % record state as dead end
                     remove first element from SL;                              %backtrack
                     remove first element from NSL;
                     CS := first element of NSL;
                  end
               add CS to SL;
            end
            else begin
               place children of CS (except nodes already on DE, SL, or NSL) on NSL;
               CS := first element of NSL;
               add CS to SL
            end
      end;
      return FAIL;
end.
```

Trace: Backtracking Algorithms

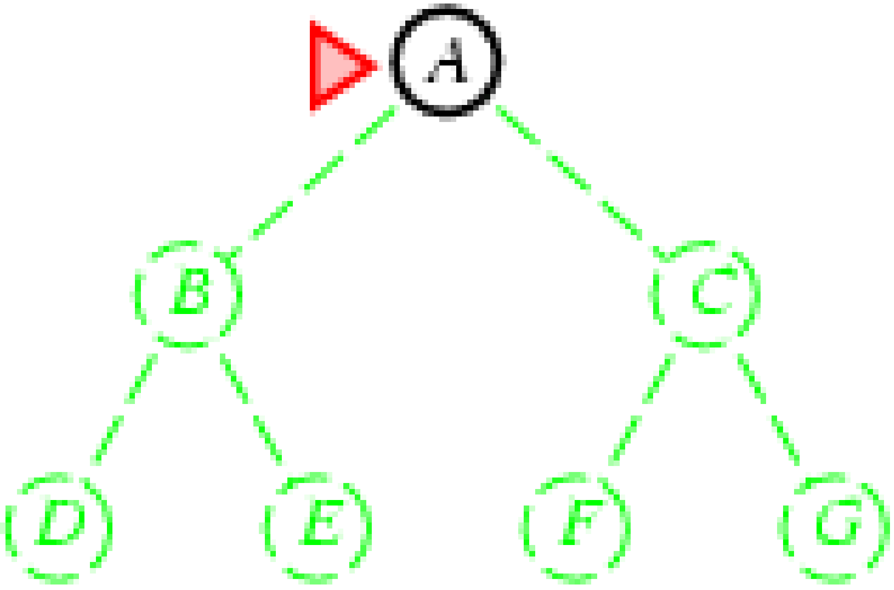


Initialize: SL = [A]; NSL = [A]; DE = [ ]; CS = A;

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[ ]
1	B	[B A]	[B C D A]	[ ]
2	E	[E B A]	[E F B C D A]	[ ]
3	H	[H E B A]	[H I E F B C D A]	[ ]
4	I	[I E B A]	[I E F B C D A]	[H]
5	F	[F B A]	[F B C D A]	[E I H]
6	J	[J F B A]	[J F B C D A]	[E I H]
7	C	[C A]	[C D A]	[B F J E I H]
8	G	[G C A]	[G C D A]	[B F J E I H]

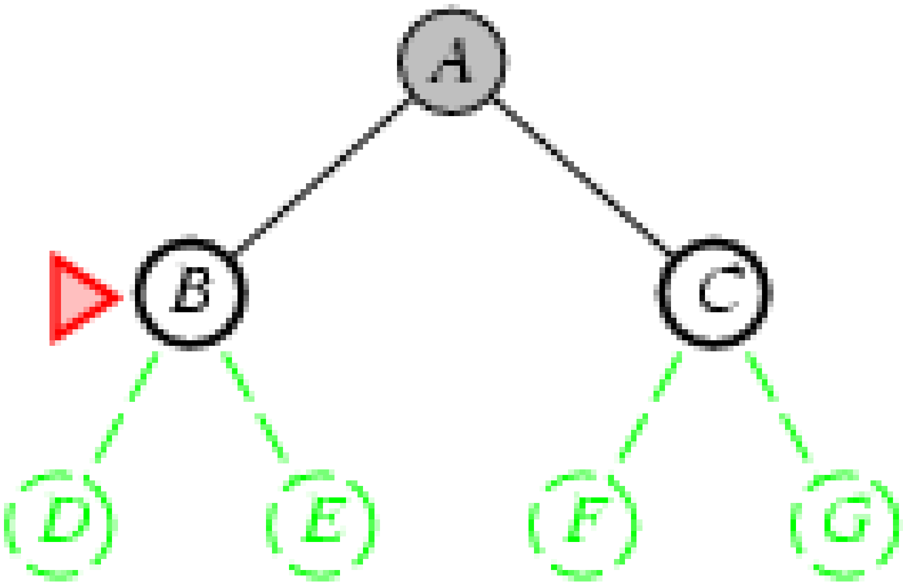
Breadth-first Search

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



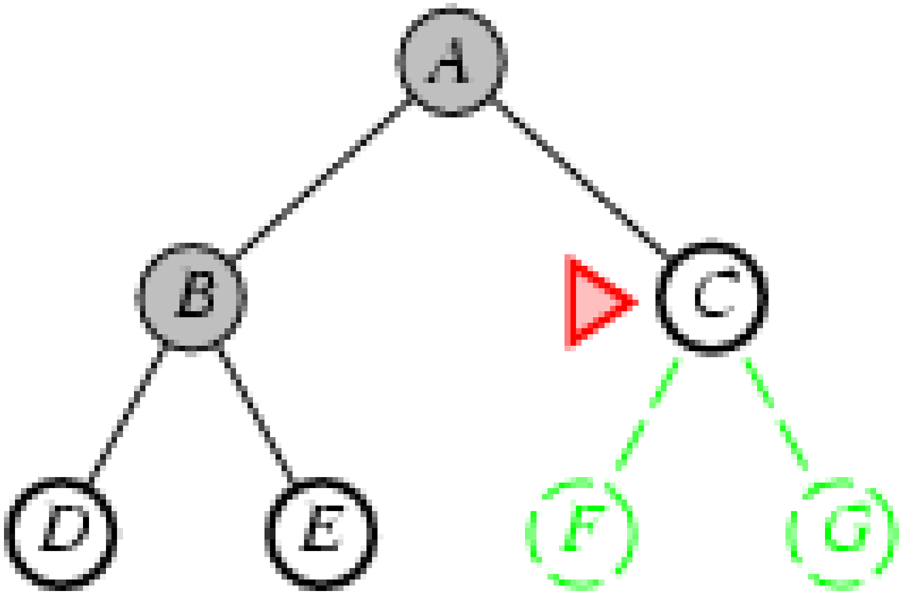
Breadth-first Search

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



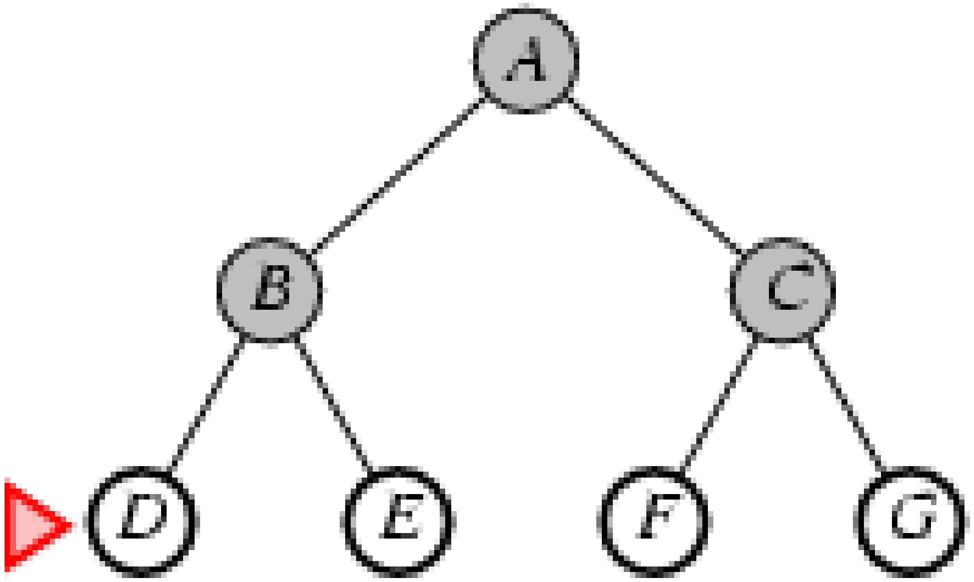
Breadth-first Search

- 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



Breadth-first Search

BFS (S):

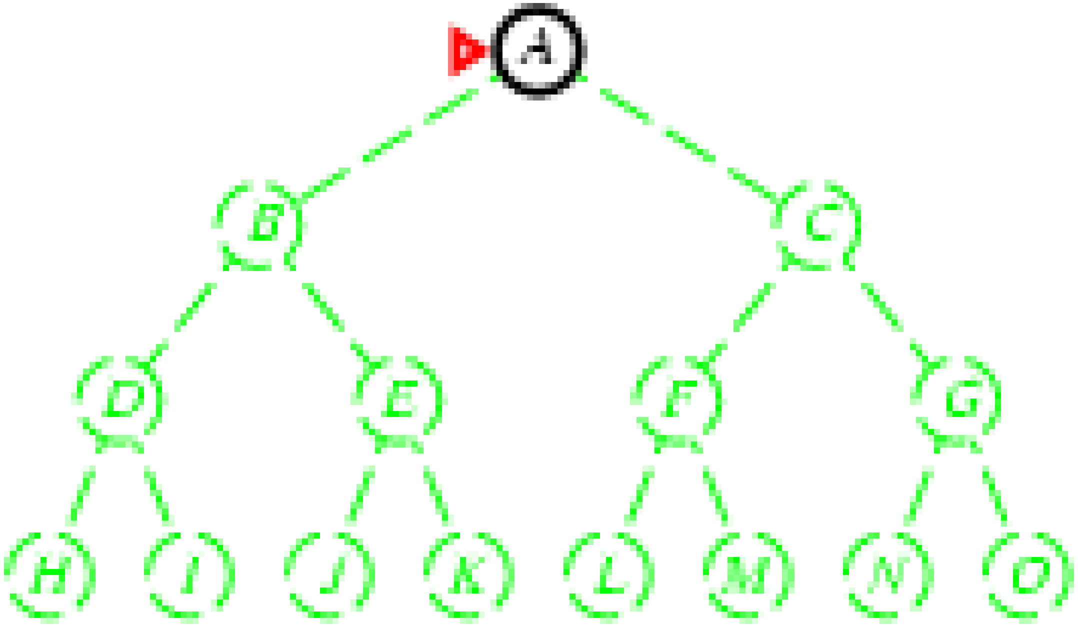
```
1. Create a variable called NODE-LIST and set it to S
2. Until a Goal state is found or NODE-LIST is empty do:
   – Remove the first element from NODE-LIST and call it E;
     If NODE-LIST was empty: Quit
   – For each way that each rule can match the state E do:
       ➢ Apply the rule to generate a new state
       ➢ If new state is a Goal state: Quit and return this state
       ➢ Else add the new state to the end of NODE-LIST
```

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)

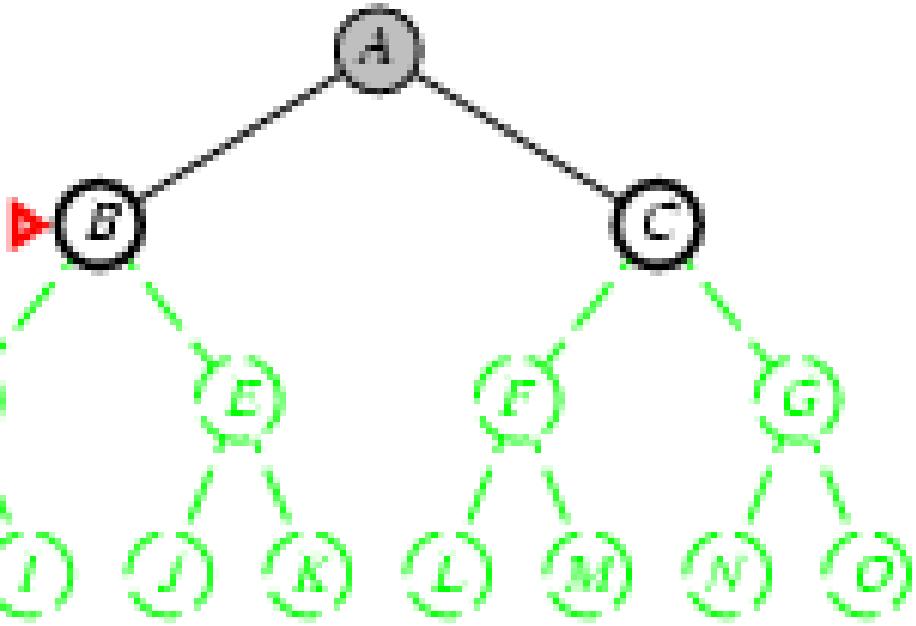
Depth-first Search

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



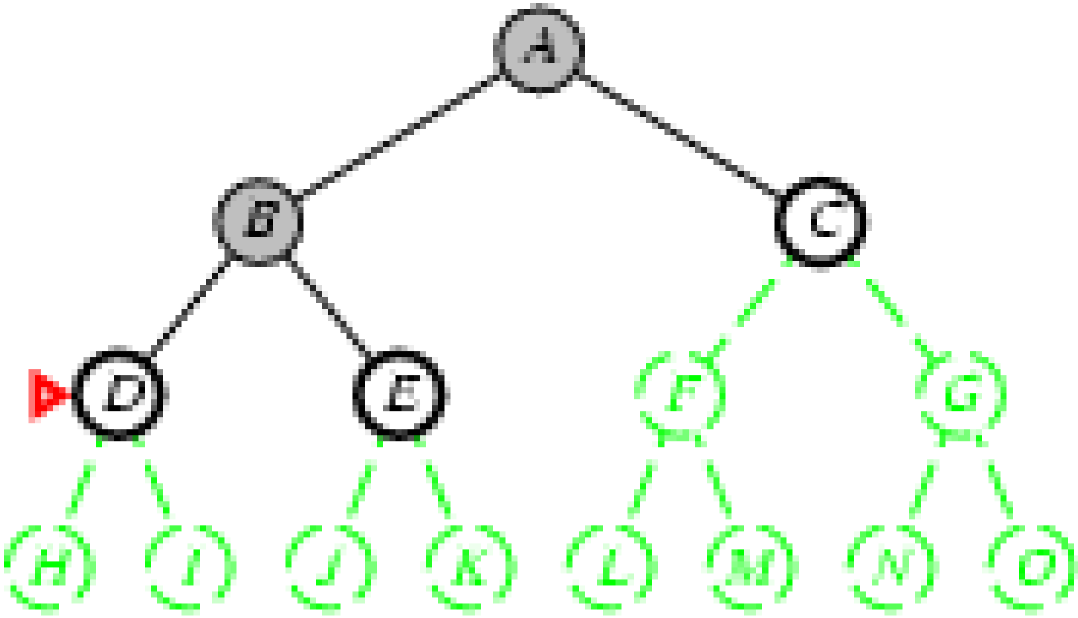
Depth-first Search

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



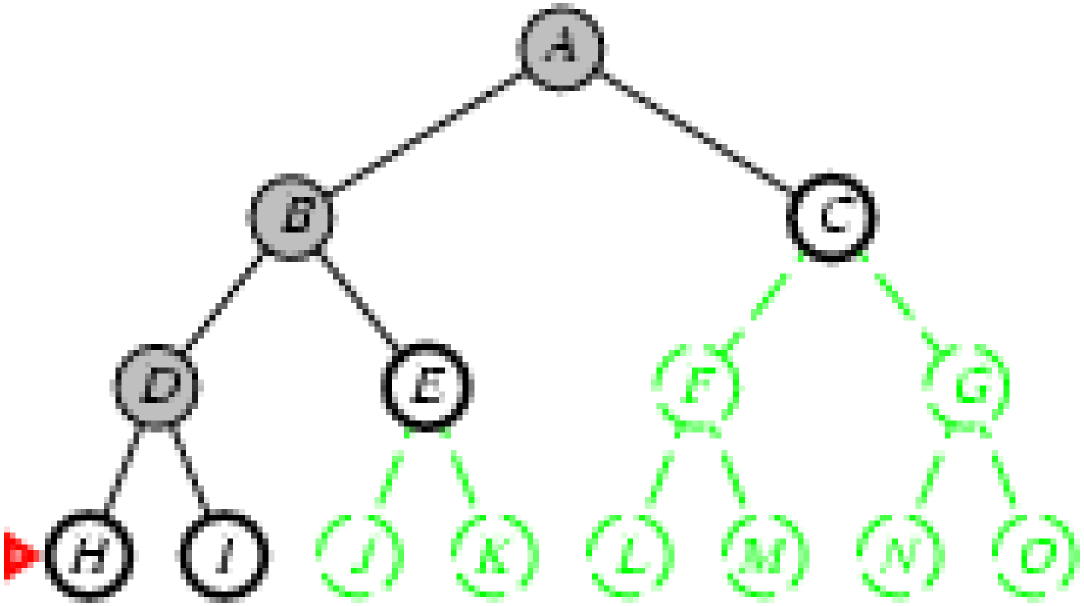
Depth-first Search

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



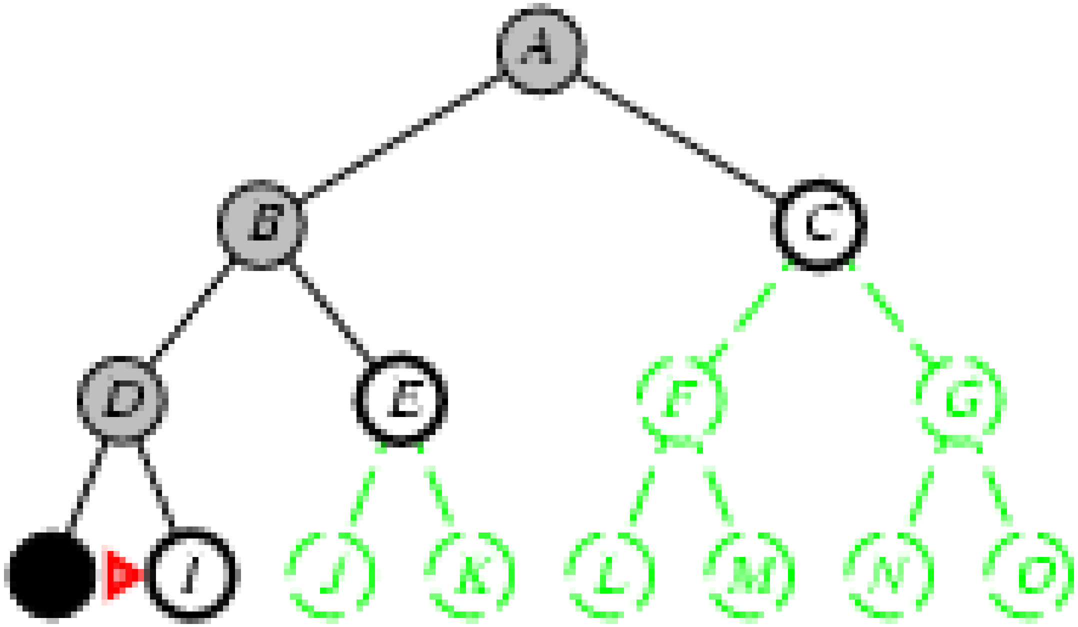
Depth-first Search

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



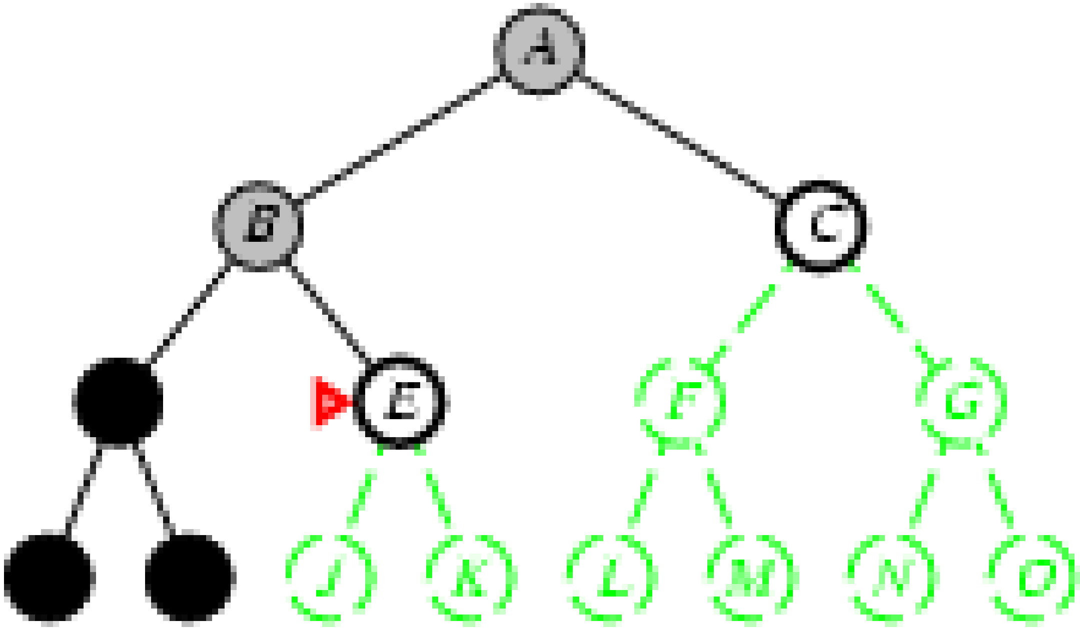
Depth-first Search

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



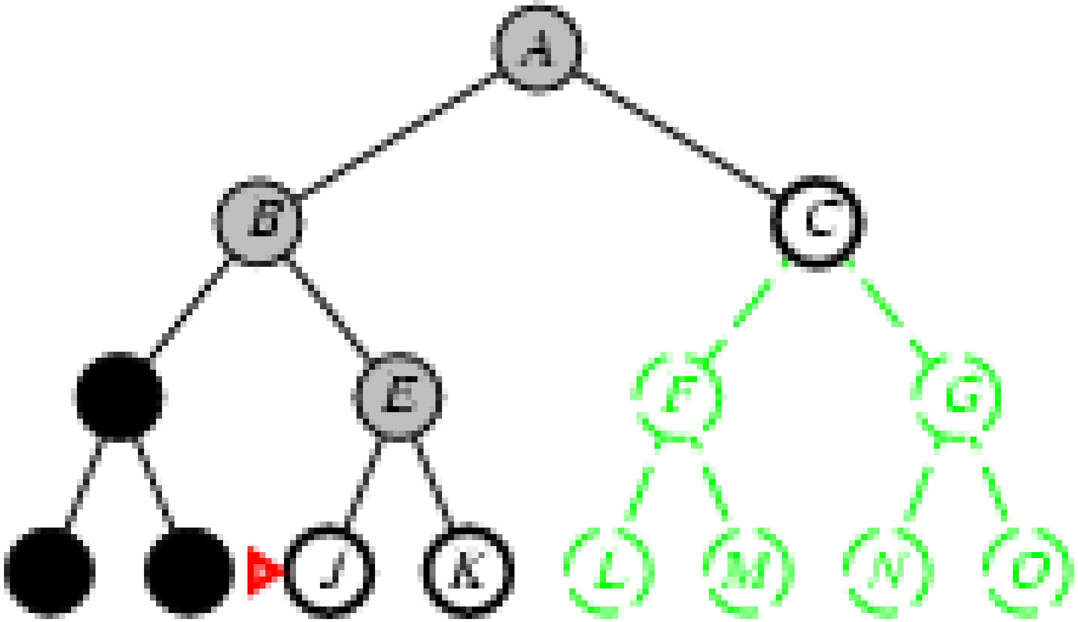
Depth-first Search

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



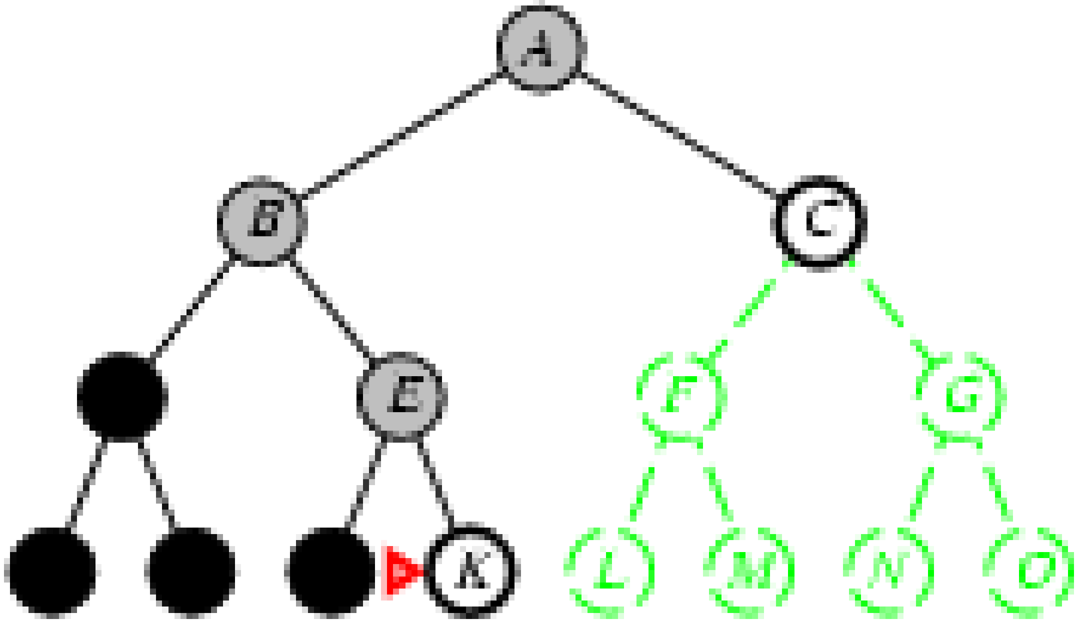
Depth-first Search

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



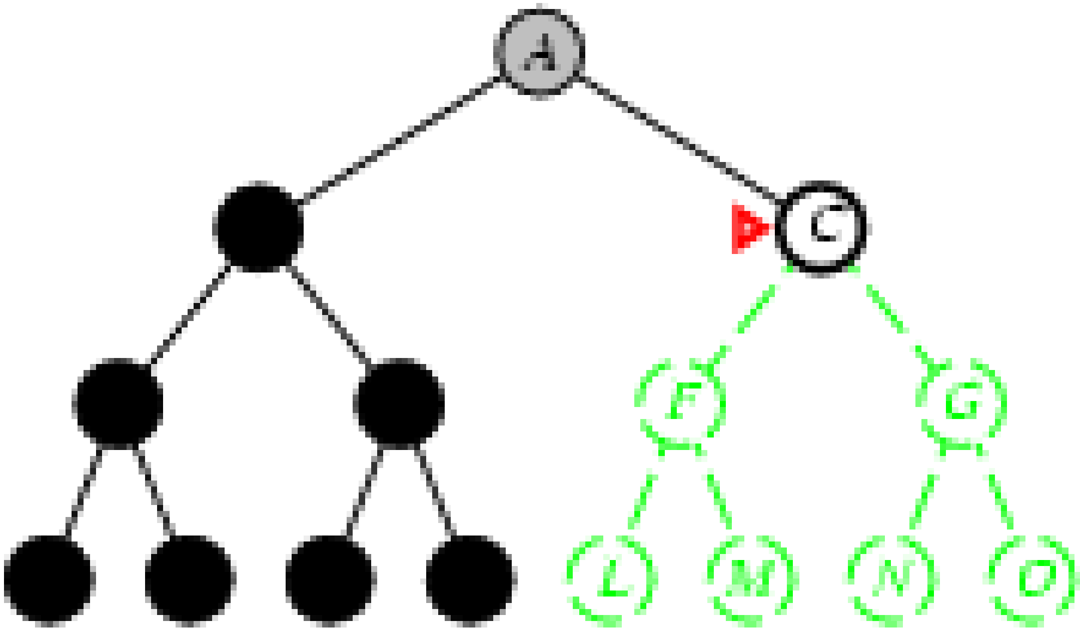
Depth-first Search

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



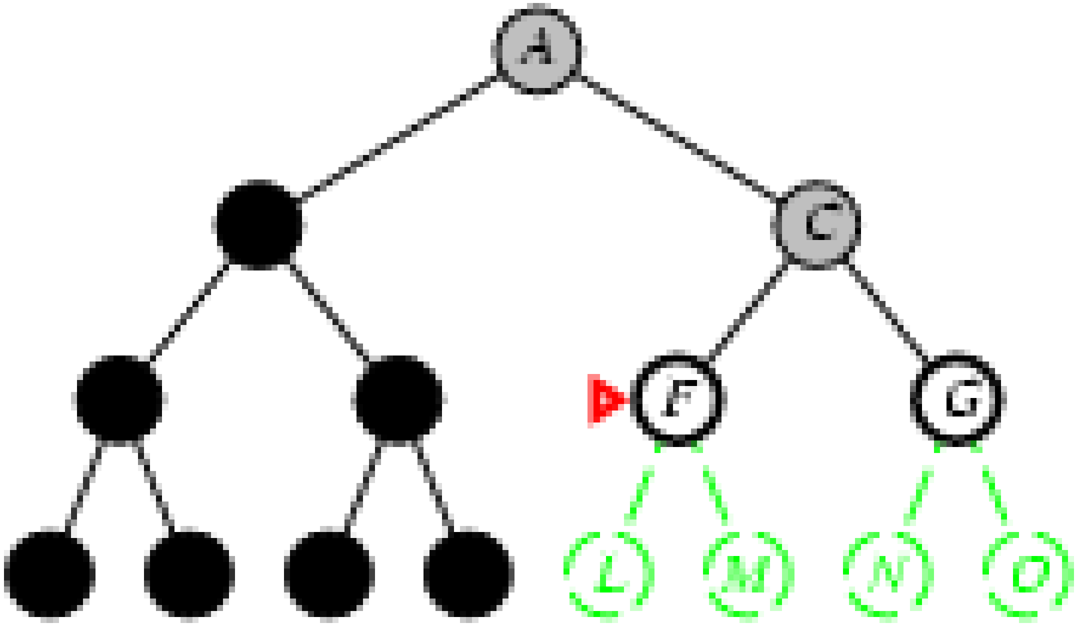
Depth-first Search

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



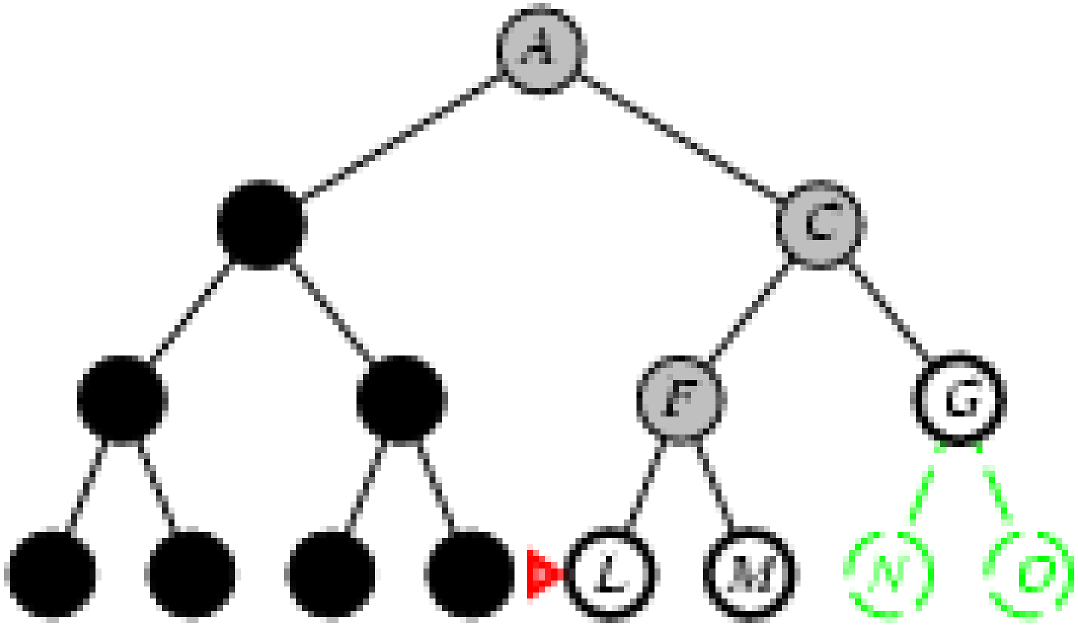
Depth-first Search

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



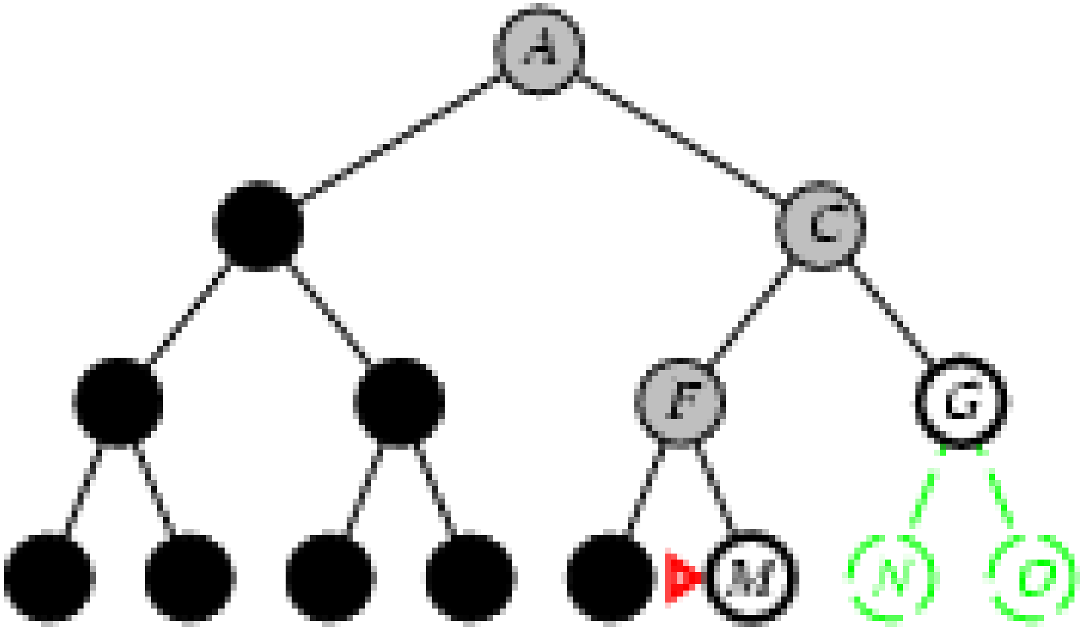
Depth-first Search

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



Depth-first Search

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



Depth-first Search

DFS (S):

1. If S is a Goal state: *Quit* and return *success*
2. Otherwise, do until *success* or *failure* is signaled:
    - Generate state E, a successor of S. If no more successors signal *failure*
    - Call DFS (E)

Depth-first Search

- Almost the same as a depth first tree traversal except
  - All nodes generated on the fly by production system
  - Algorithm halts when solution found

- DFS assumes tree structure of search space; may not be true
  - If not, can get caught in cycles
  - Thus in these cases, DFS must then be modified

    e.g. Each state has a Flag that is raised when node is *visited*

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
  - If solutions are dense, may be much faster than breadth-first

- **Space?**
  - $O(bm)$, i.e., linear space!

- **Optimal?**
  - No

Differences: DFS and BFS

- DFS and BFS wrt ordering nodes in open list:
  - DFS uses a stack: Nodes are added on the top of the list
  - BFS uses a queue: Nodes are added at the end of the list
- DFS and BFS wrt examination process:
  - DFS examines all the node's children and their descendent before the node's siblings
  - BFS examines all the node's siblings and their children
- DFS and BFS wrt completeness:
  - DFS is not complete (it may be stuck in an infinite branch)
  - BFS is complete (it always finds a solution if it exists)

Differences: DFS and BFS

- DFS and BFS wrt optimality:
  - DFS is not optimal: (it will not find the shortest path)
  - BFS is optimal: (it always finds shortest path)
- DFS and BFS wt memory:
  - DFS requires less memory (only memory for states of one path needed)
  - BFS requires exponential space for states required
- DFS and BFS wrt efficiency:
  - DFS is efficient if solution path is known to be long
  - BFS is inefficient if branching factor B is very high

What to Choose: DFS and BFS

- The choice of the *DFS* or *BFS*
  - Depends on the problem being solved
  - Importance of finding the shortest path
  - The branching factor of the space
  - The available compute time and space resources
  - The average length of paths to a goal node
  - Whether we are looking for all solutions or the first one

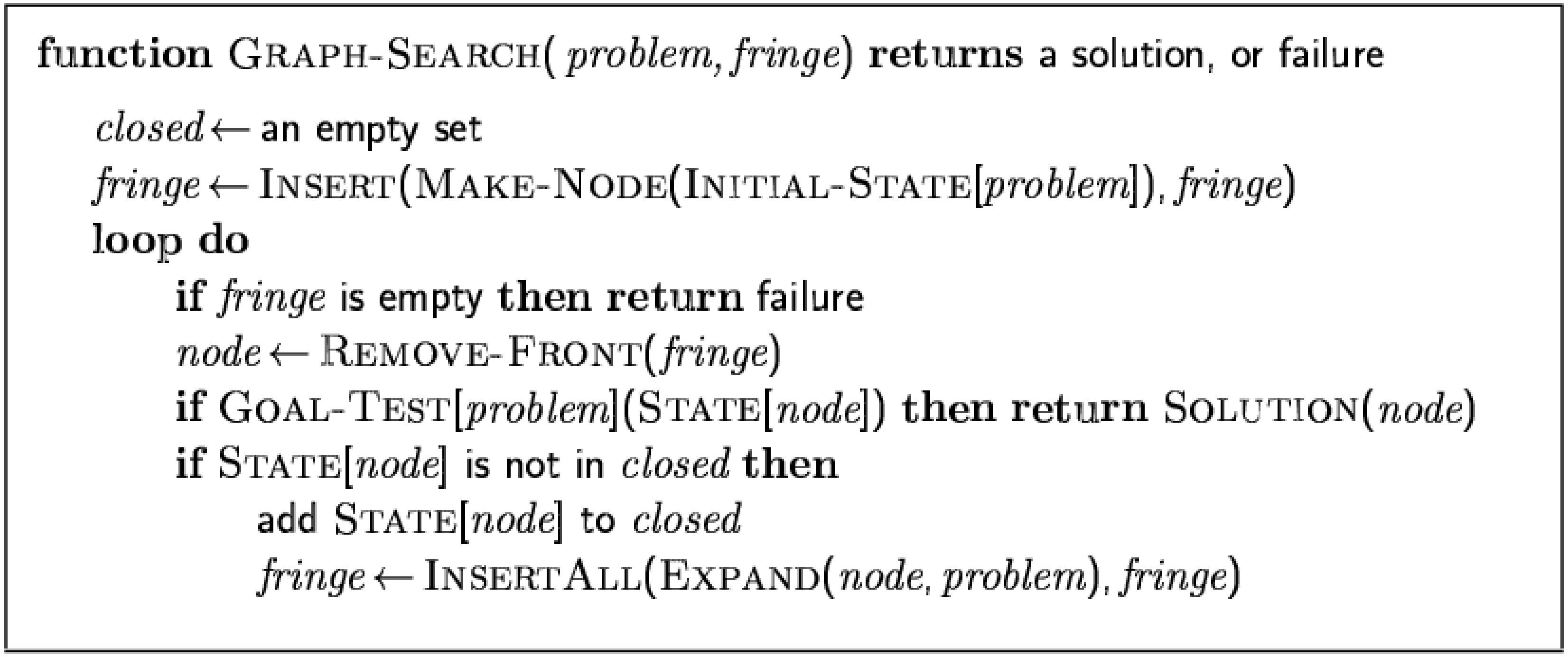
Changing a Cyclic Graph Into a Tree

- Most production systems include cycles
- Cycles must be broken to turn graph into a tree
- Then use the above tree searching techniques
- Can’t “mark” nodes - they are generated dynamically
- Therefore: Keep a list of all visited states (“Closed”)
- Check each state examined if it is in “Closed”
- If it is in “Closed”: Ignore it and examine the next…

Algorithm to Break Cycles

- When a node is examined
  - ; Check node to see if it is in “Closed” list
  - If node is in the “Closed” list
    - ➢ Ignore it
  - Else
    - ➢ Add node to “Closed” list
    - ➢ Process node

Graph Search



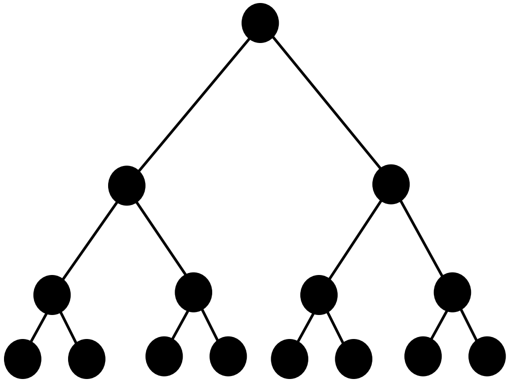
```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

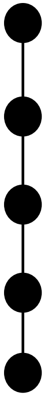
Example: DFS with Cycle Cutting

Initializations: S = first_state, CLOSED = Empty_List

```
DFS (S):
    If S is in CLOSED
        Return Failure
    Else
        Place S in CLOSED
        If S is a Goal state, Return Success
        Loop
            • Generate state E, a successor of S.
                – If no more successors return Failure
            • Result = DFS (E)
            • If Result = Success Return Success
```

BFS vs. DFS





- BFS expensive wrt space
    - Linear in # of nodes
- DFS
    - Only stores a max of log of the No. of nodes

- BFS constant memory needed
- DFS linear in # of nodes

- Time to find sol^n depends on where the sol^n is in the tree
- DFS may find a longer path than BFS when multiple sol^{n}s exist
- BFS guaranteed minimum path solution

Uniform-cost search

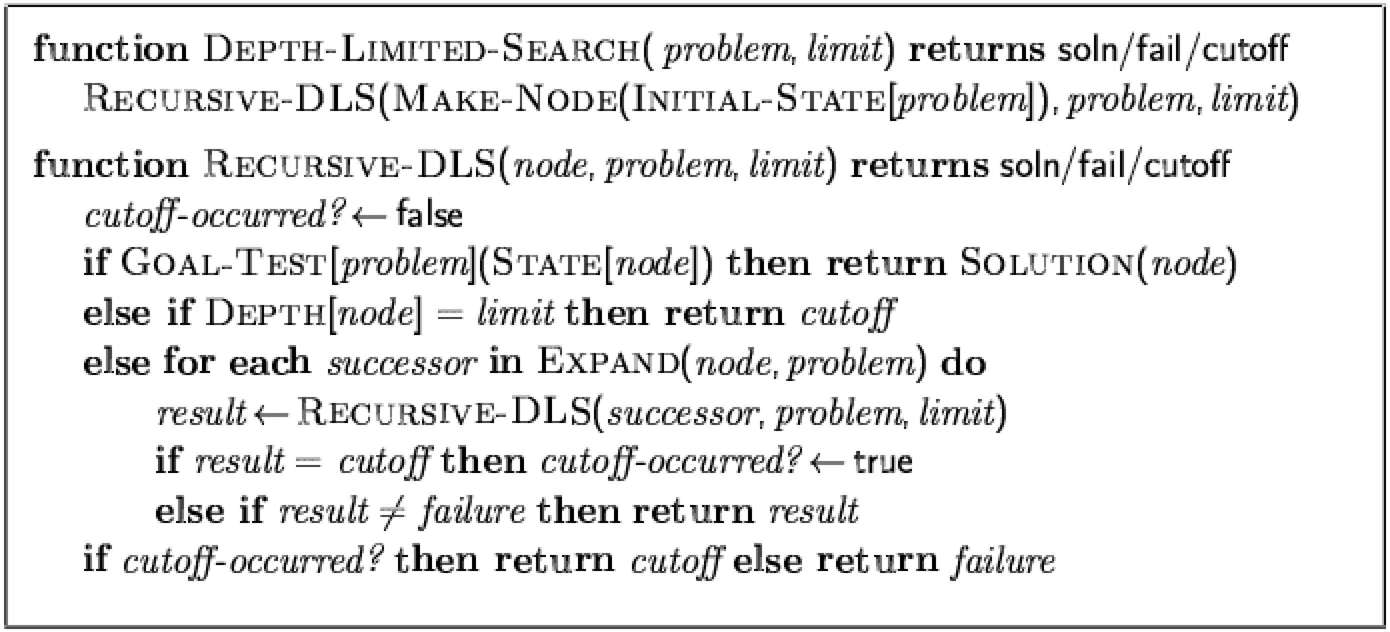
- Expand least-cost unexpanded node
- **Implementation:**
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- **Complete?**
  - Yes, if step cost ≥ ε
- **Time?**
  - No. of nodes with *g* ≤ cost of optimal solution
  - $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- **Space?**
  - No. of nodes with *g* ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- **Optimal?**
  - Yes – nodes expanded in increasing order of *g(n)*

Depth-limited Search

This is the Depth-first search with depth limit *L*,

i.e., nodes at depth *L* have no successors

- Recursive implementation:



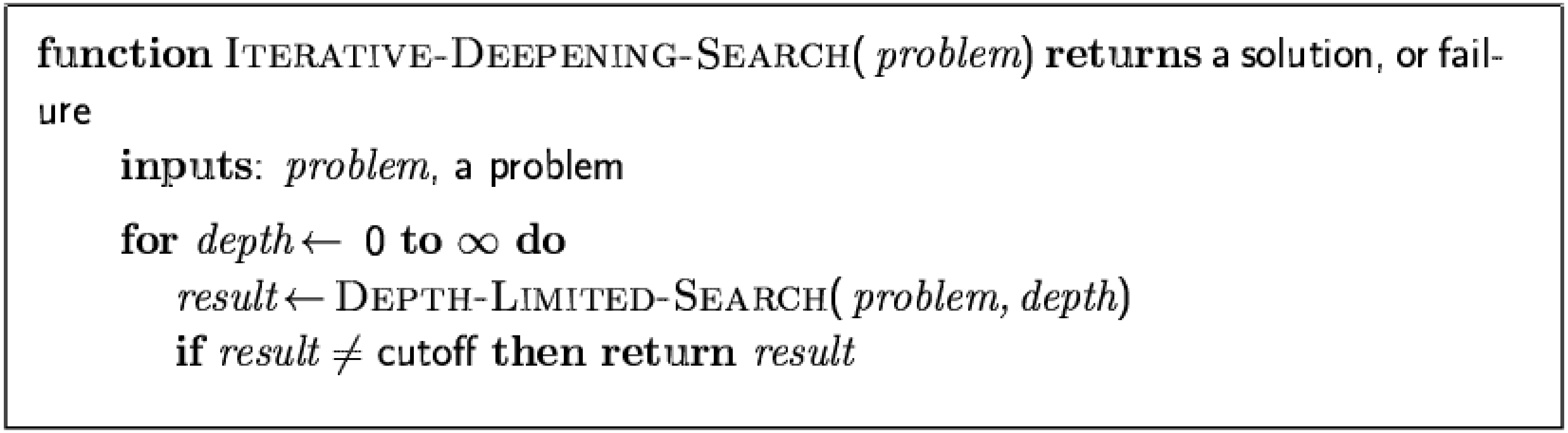
```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return 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 Search

- **Iterative deepening depth-first search** (IDDFS)
- A depth-limited search is run repeatedly,
- Depth limit increased with each iteration until it reaches *d*, the depth of the shallowest goal state.
- On each iteration, IDDFS:
  - Visits the nodes in the search in the same order as the DFS.
  - The cumulative order in which nodes are first visited, with no pruning, is effectively BFS.
  - SO: If there is an optimal solution at a lower depth, it finds it.

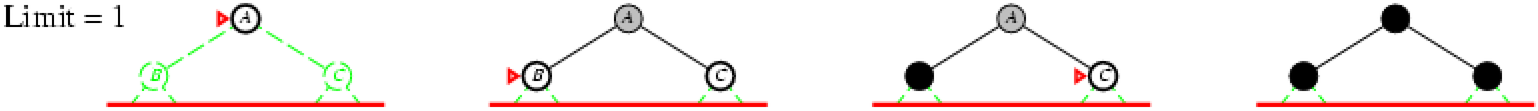
Iterative Deepening Search



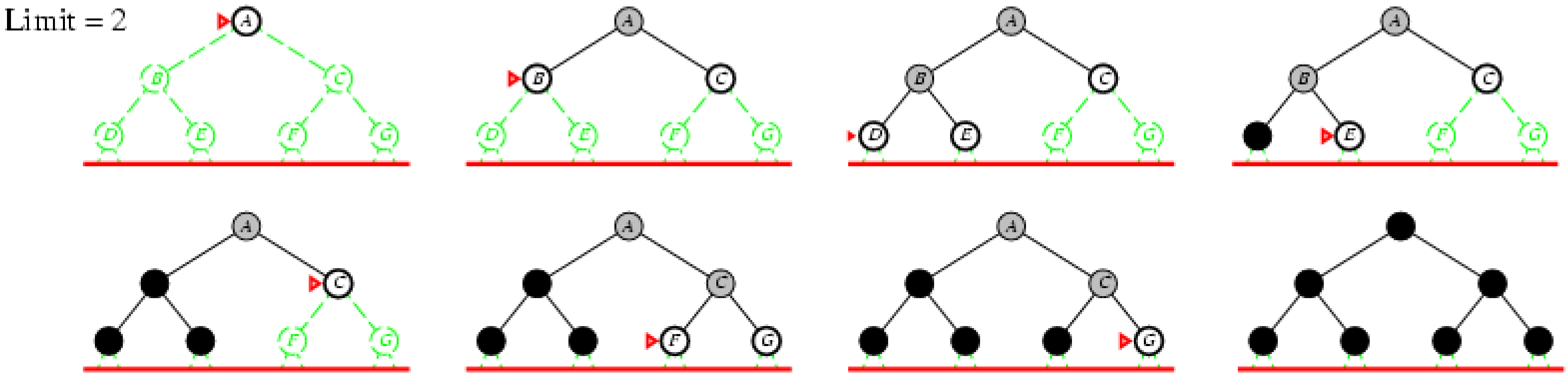
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Iterative Deepening Search *L* =0

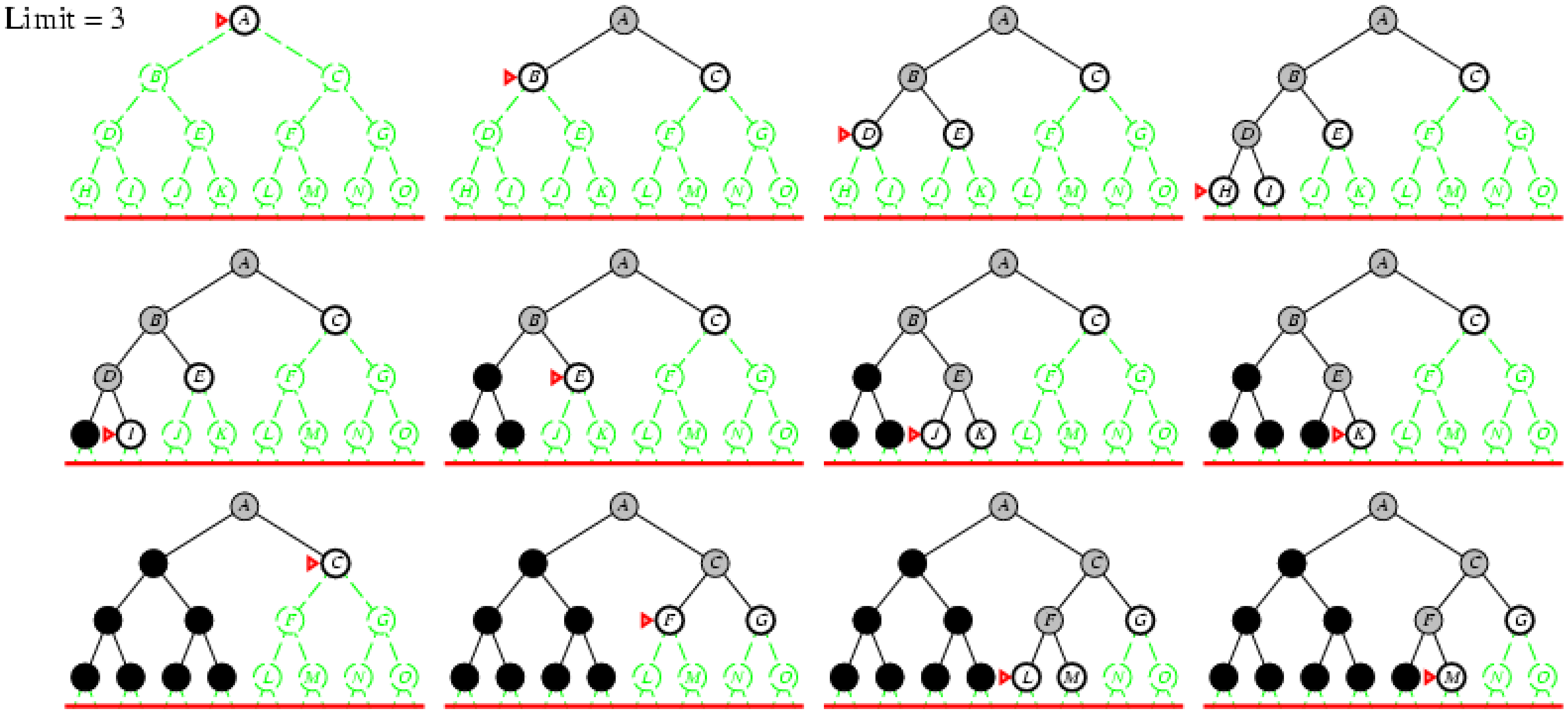
Iterative Deepening Search $L = 1$



Iterative Deepening Search *L* = 2



Iterative Deepening Search *L* = 3



Iterative Deepening Search Properties

- **Complete?**
  - Yes
- **Time?**
  - Nodes on the bottom level are expanded once
  - Those on the next to bottom level are expanded twice, etc.
  - Up to the root of the search tree, which is expanded d + 1 times.
  - $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- **Space?**
  - $O(bd)$
- **Optimal?**
  - Yes, if step cost = 1

Depth-limited vs. Iterative Deepening Search

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

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

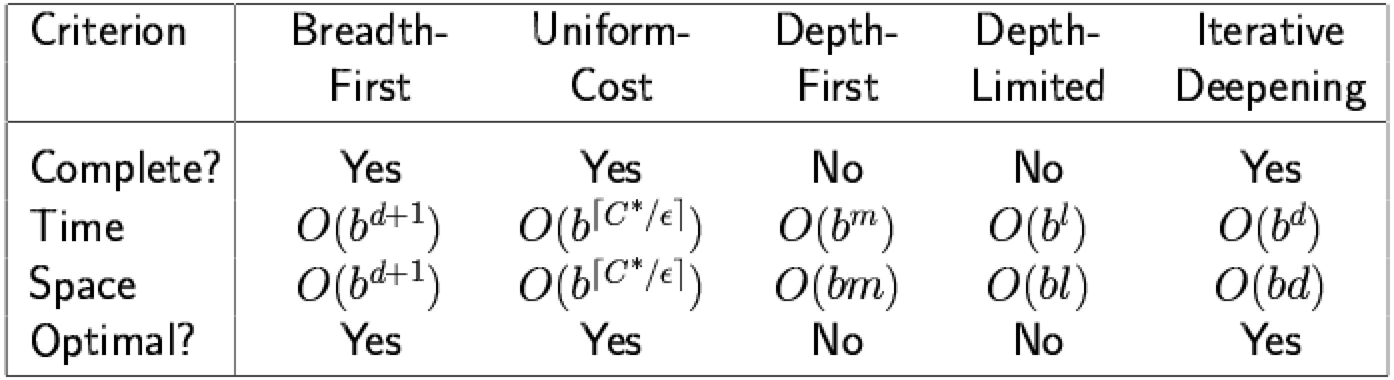
- Number of nodes generated in an **Iterative Deepening Search** to depth *d* with branching factor *b*:

$$\mathrm{N}_{\mathrm{IDS}} = (\mathrm{d}+1)\mathrm{b}^0 + \mathrm{d}\ \mathrm{b}^1 + (\mathrm{d}-1)\mathrm{b}^2 + \ldots + 3\mathrm{b}^{\mathrm{d}-2} + 2\mathrm{b}^{\mathrm{d}-1} + 1\mathrm{b}^{\mathrm{d}}$$

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

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

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