Solving Problems by Searching

CSE-345: Artificial Intelligence

Introduction

To build a system to solve a particular problem, need to do four things:

- Define the problem precisely.
- Analyze the problem.
- Isolate & represent the task knowledge that is necessary to solve the problem.
- Choose the best problem-solving technique (s) & apply it (them) to the particular problem.

Production Systems

- Search forms the core of many intelligent processes
- A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.
- It is useful to structure AI programs in a way that facilitates describing & performing the search process.
- Production systems provide such structures.

Production Systems

A production system consist of:

- A **set of rules**, each consisting of a left side (a pattern) that determines the applicability of the rule & a right side that describes the operation to be performed if the rule is applied
- One or more **knowledge/databases** that contain whatever information is appropriate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.
- A **control strategy** that specifies the order in which the rules will be compared to the database & a way of resolving the conflicts that arise when several rules match at once
- A **rule applier**

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 ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

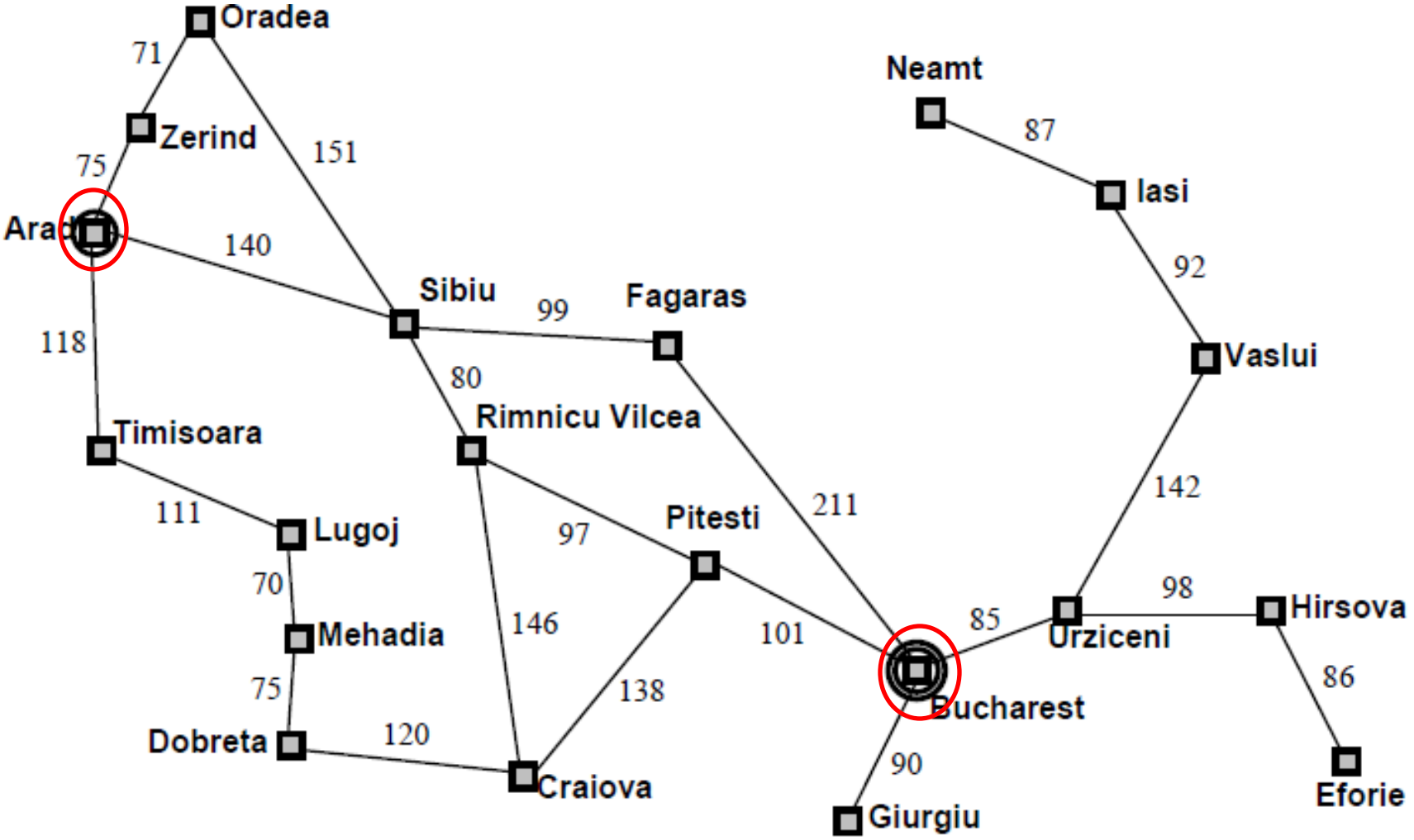
Assumptions

- Static: The world does not change unless the agent changes it.
- Observable: Every aspect of the world state can be seen.
- Discrete: Has distinct states as opposed to continuously flowing time.
- Deterministic: There is no element of chance.
  - This is a restricted form of a general agent called offline problem solving.
  - Online problem solving involves acting without complete knowledge

Why Search?

- To achieve goals or to maximize utility, need to predict what the result of actions in the future will be.
- There are many sequences of actions, each with their own utility.
- To find, or search for, the best one.

Example: Romania



Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

- To be in Bucharest

Formulate problem:

- states: various cities
- actions: drive between cities

Find solution:

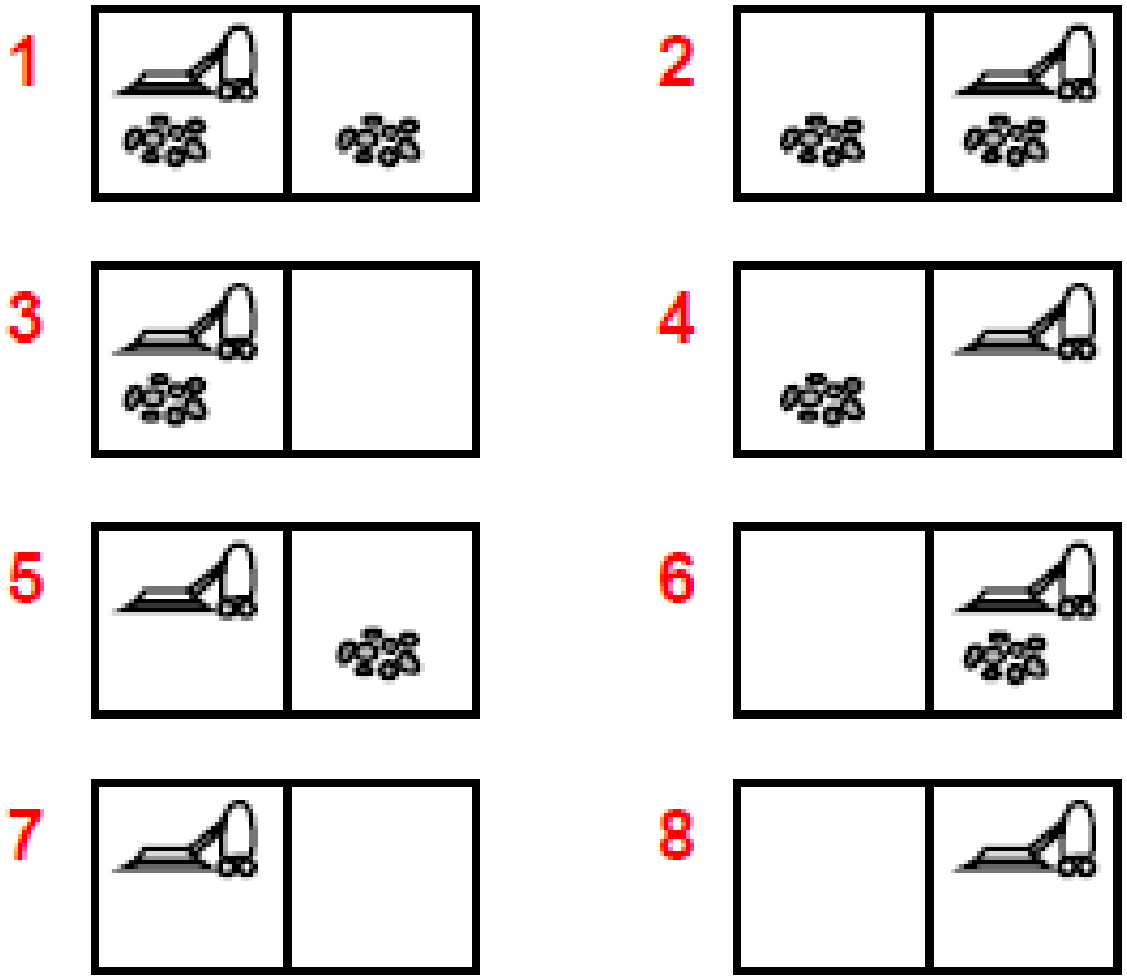
- By searching through states to find a goal
- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Execute states that lead to a solution

Problem Types

- Single-state Problem → Deterministic, fully observable
  - Agent knows exactly which state it will be in;
  - Can calculate what action sequence will gets to a goal state
- Multiple State Problem→ Non-observable
  - Agent may have no idea where it is;
  - Agent must reason about sets of states that it might get to, rather than single states
- Contingency Problem → Nondeterministic and/or partially observable
  - percepts provide new information about current state
  - solution is a contingent plan or a policy
  - often interleave search, execution
- Exploration Problem → Unknown state space

Eight Possible States of Simplified Vacuum World



Example: Vacuum World

- Single-state

  start in #5.

  Solution??[Right; Suck]

- Multi-state

  start in {1; 2; 3; 4; 5; 6; 7; 8}

  e.g., Right goes to {2; 4; 6; 8}.

  Solution?? [Right; Suck; Left; Suck]

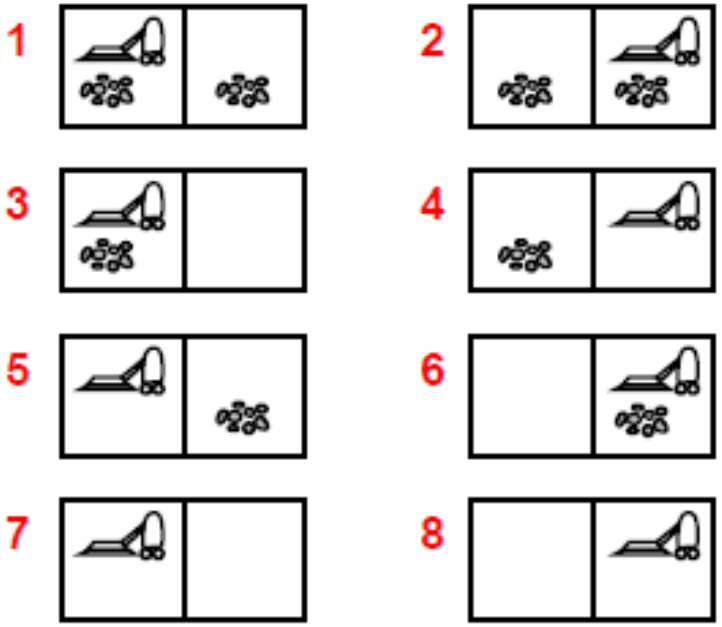
- Contingency

  start in #5

  Murphy's Law: Suck can dirty a clean carpet

  Local sensing: dirt, location only.

  Solution??[Right; if dirt then Suck]



Single-State Problem Formulation

A problem is defined by four items:

- initial state e.g., "at Arad"
- successor function, S- Given a particular state *x*, *S(x)* returns the set of states reachable from *x* by any single action.
  - e.g., S(Arad) = {<Arad → Zerind; Zerind>,…}
- goal state, can be
  - explicit, e.g., x = "at Bucharest"
  - implicit, e.g., NoDirt(x)
- 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 absolutely complex

  →state space must be abstracted for problem solving

  (Abstract) state = set of real states

  (Abstract) action = complex combination of real actions

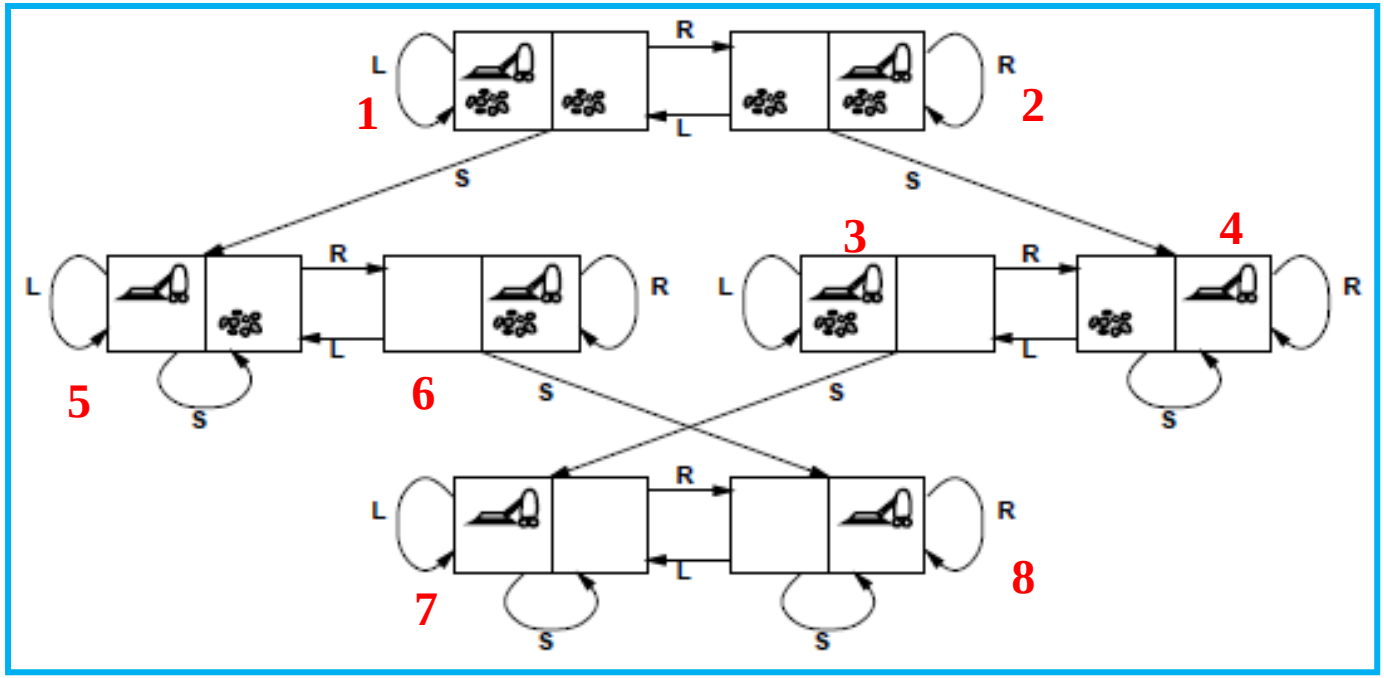
  e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.

  (Abstract) solution =

  set of real paths that are solutions in the real world

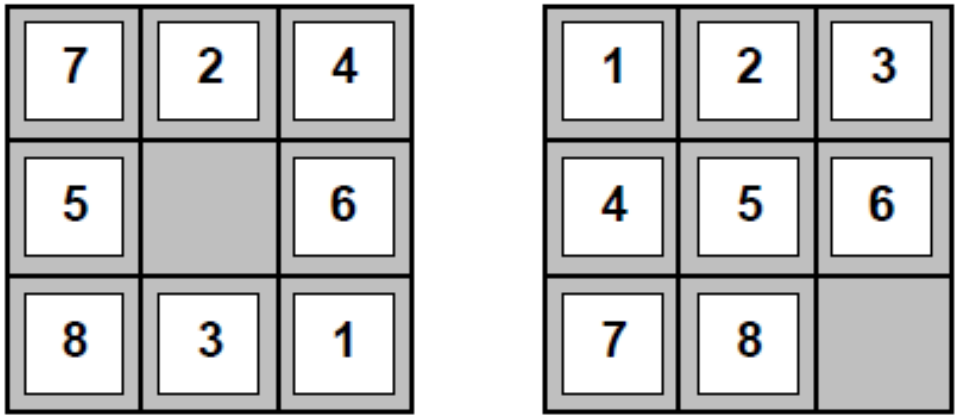
- Each abstract action should be "easier" than the original problem

Vacuum World State Space Graph



- states??: discrete: 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

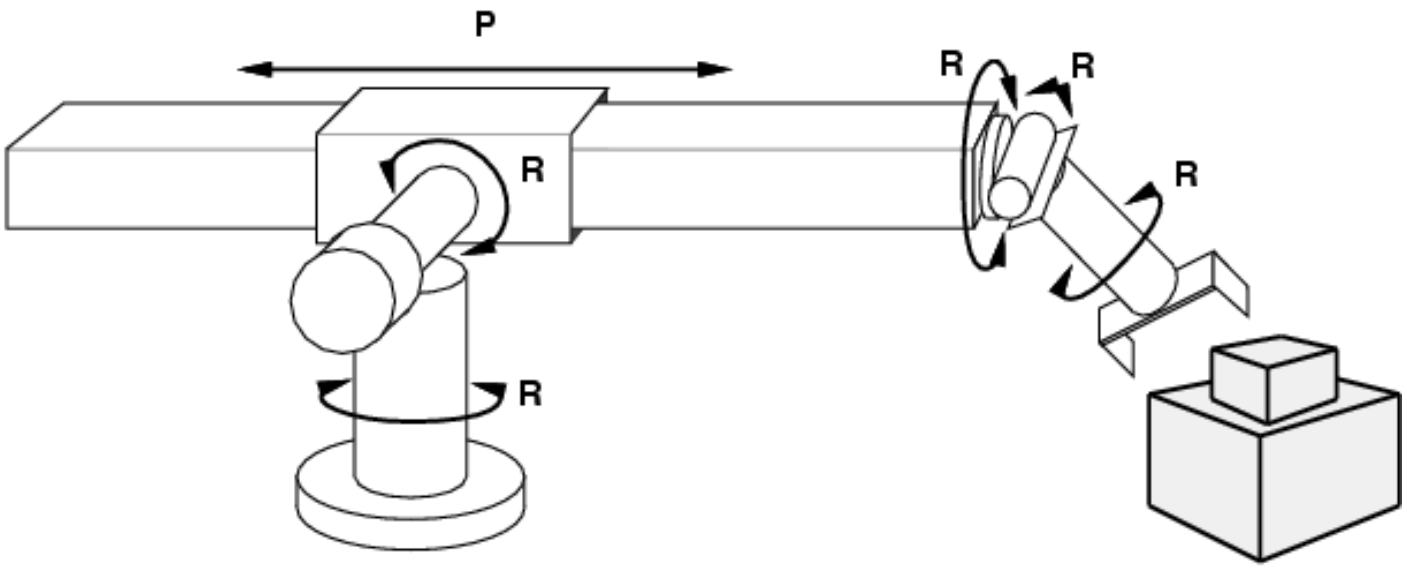


Start State Goal State

- 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
- path cost??: time to execute

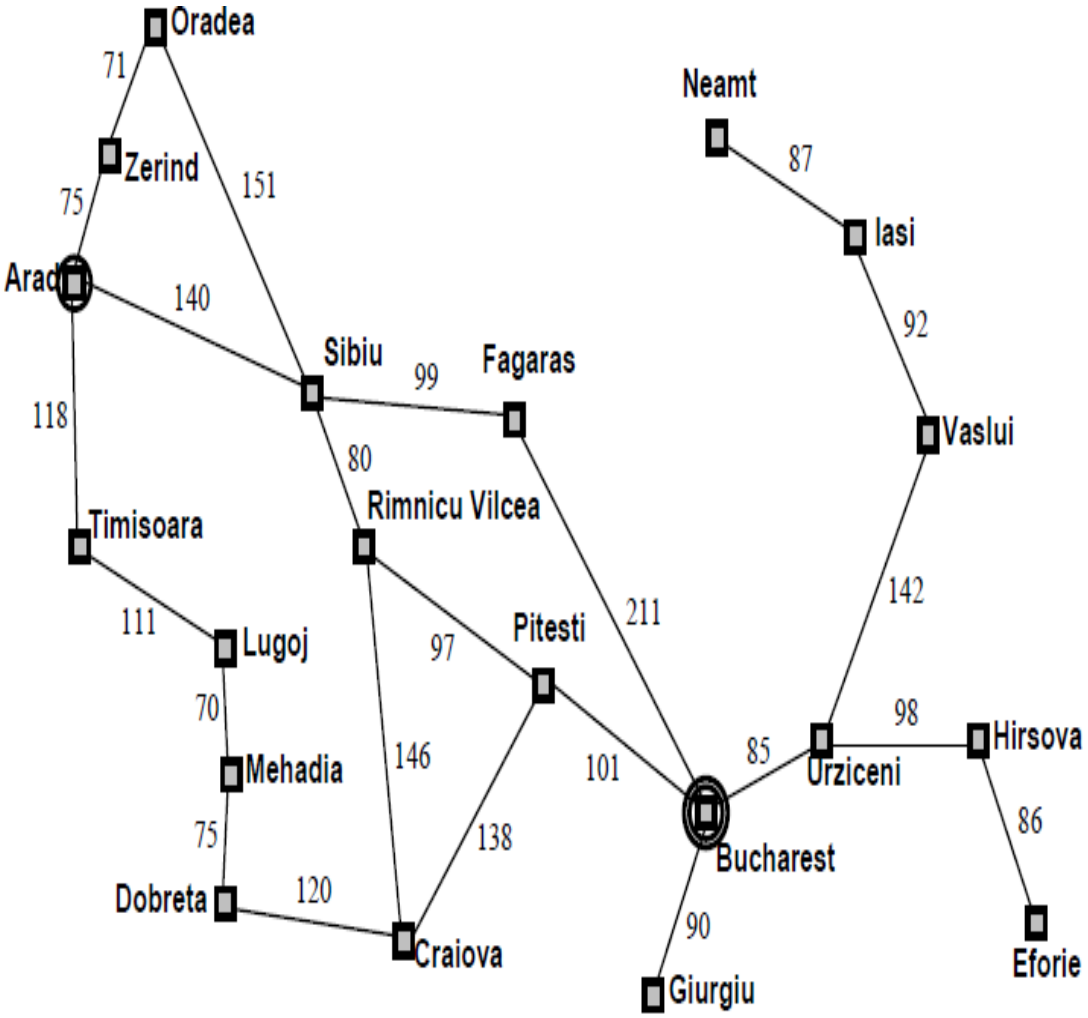
Tree Search Algorithms

Basic idea:

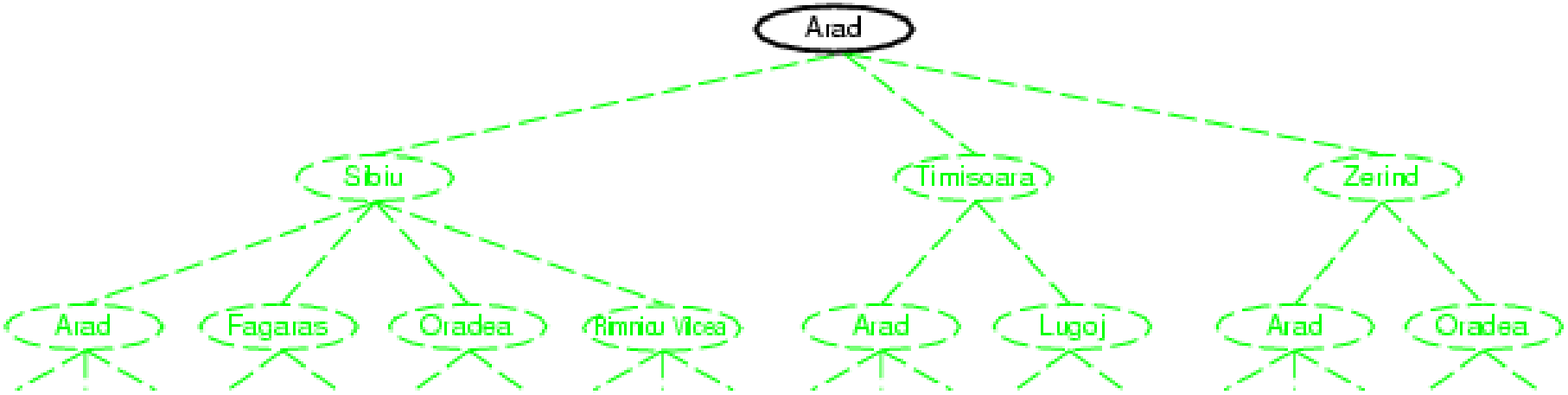
- simulated exploration of state space by generating successors of already-explored states (i.e., expanding states)

```
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
```

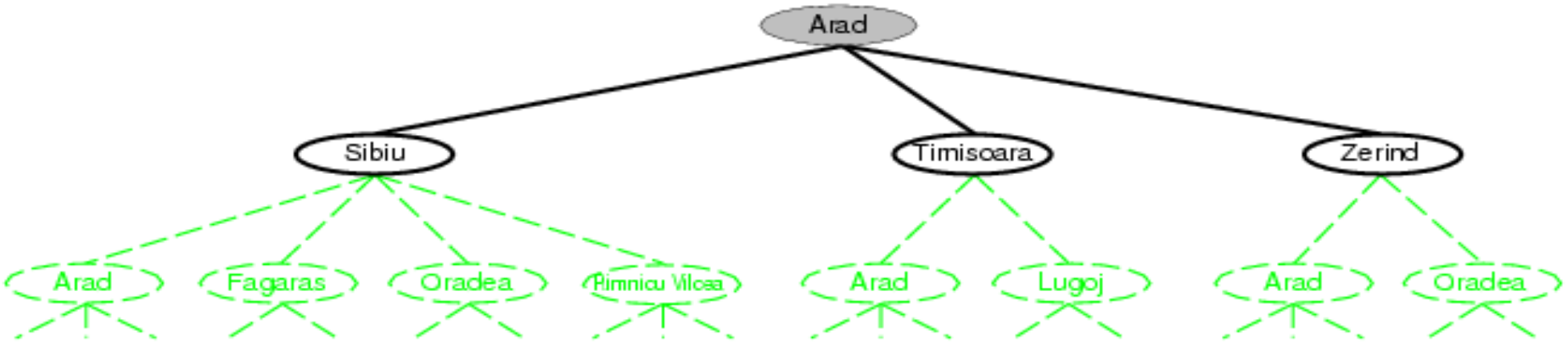
Tree Search Example



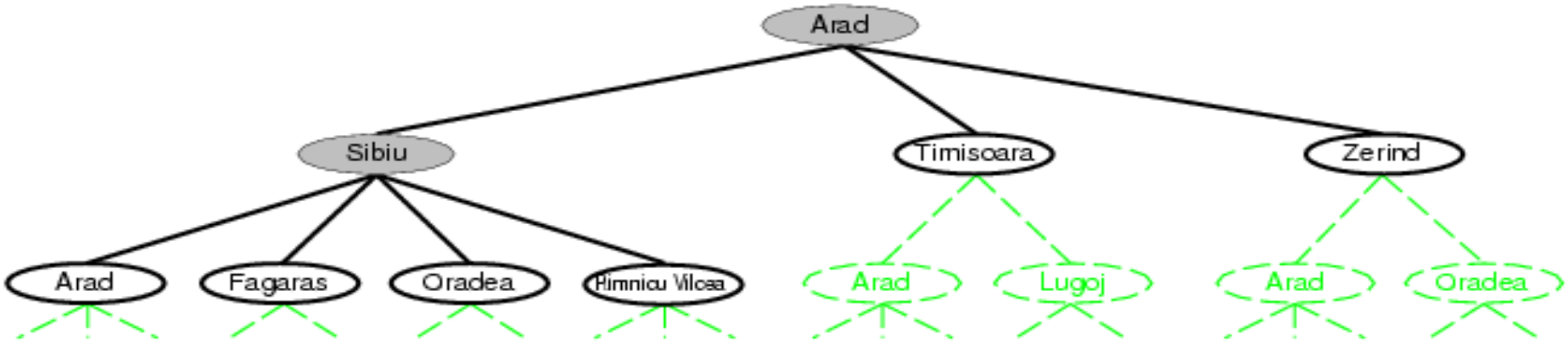
Tree Search Example



Tree Search Example

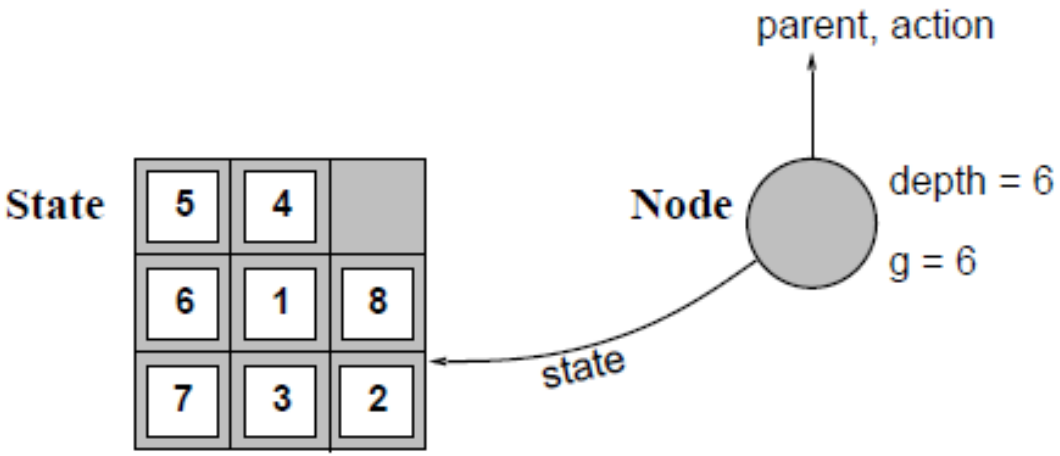


Tree Search Example



Implementation: States vs. Nodes

- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth
- A state is a (representation of) a physical configuration and do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields & 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(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Search Strategies

- **Uninformed search strategies**
  - Also known as "blind search," uninformed search strategies use no information about the likely "direction" of the goal node(s)
  - Uninformed search methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional
- **Informed search strategies**
  - Also known as "heuristic search," informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal node(s)
  - Informed search methods: Hill climbing, best-first, greedy search, beam search, A, A*

Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded
- **Space complexity**
  - How much space is used by the algorithm? Usually measured in terms of the maximum size of the “nodes” list during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? That is, is it the one with minimum cost?

Uninformed Search Methods

Breadth-First Search (BFS)

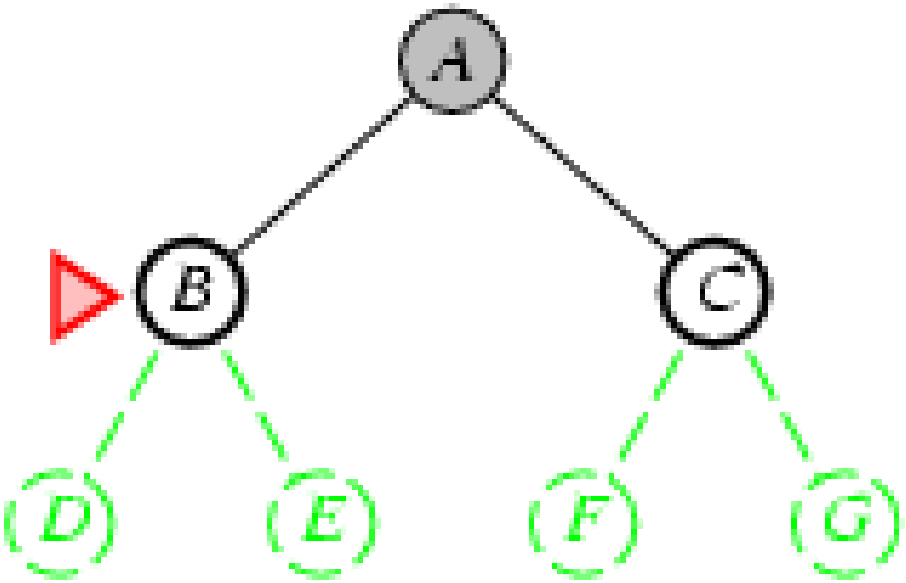
- Expand shallowest unexpanded node
- Fringe: nodes waiting in a queue to be explored
- Implementation:
  - *fringe* is a FIFO queue, i.e. new successors go at end

Example

Expand:

- fringe = [B,C]

Is B a goal state?

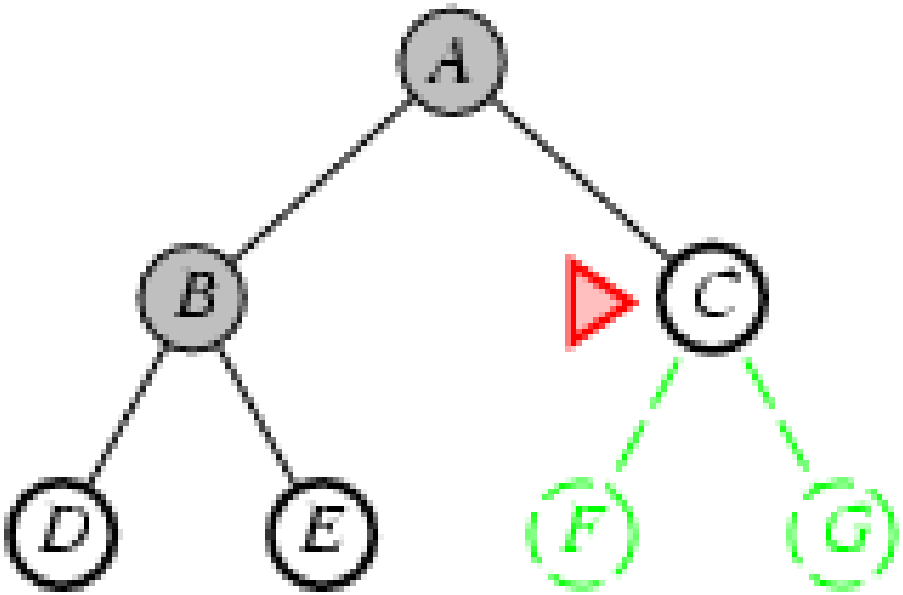


Example

Expand:

- fringe=[C,D,E]

Is C a goal state?

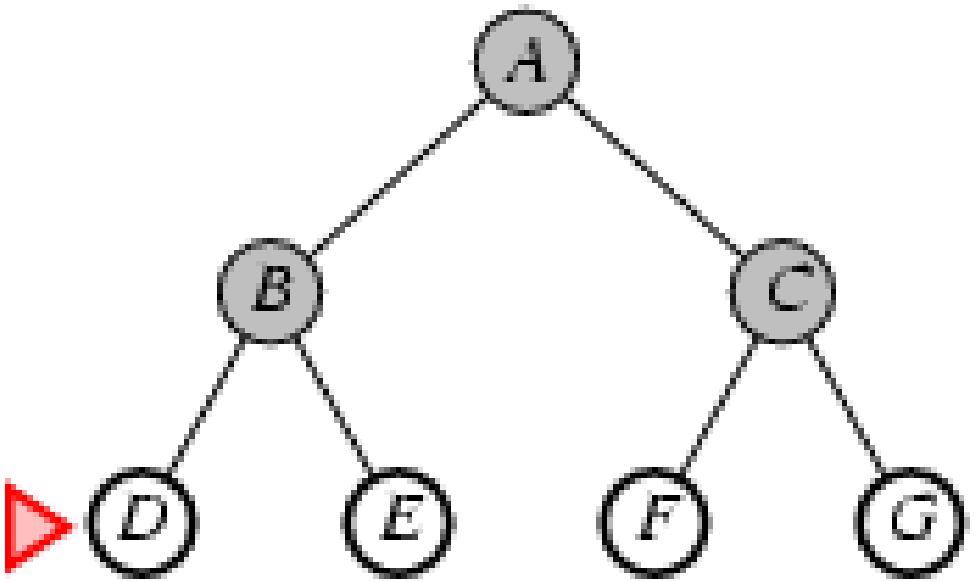


Example

Expand:

- fringe=[D,E,F,G]

Is D a goal state?



Breadth-First Search (BFS)

- BFS traverses a graph in what is sometime called *level order.*
- Intuitively it starts at the source node and visits all the nodes directly connected to the source.
- We call these level 1 nodes. Then it visits all the unvisited nodes connected to level 1 nodes, and calls these level 2 nodes etc.
- Since all of the nodes of a level must be saved until their child nodes in the next level have been generated, the space complexity is proportional to the number of nodes at the deepest level

Properties of BFS

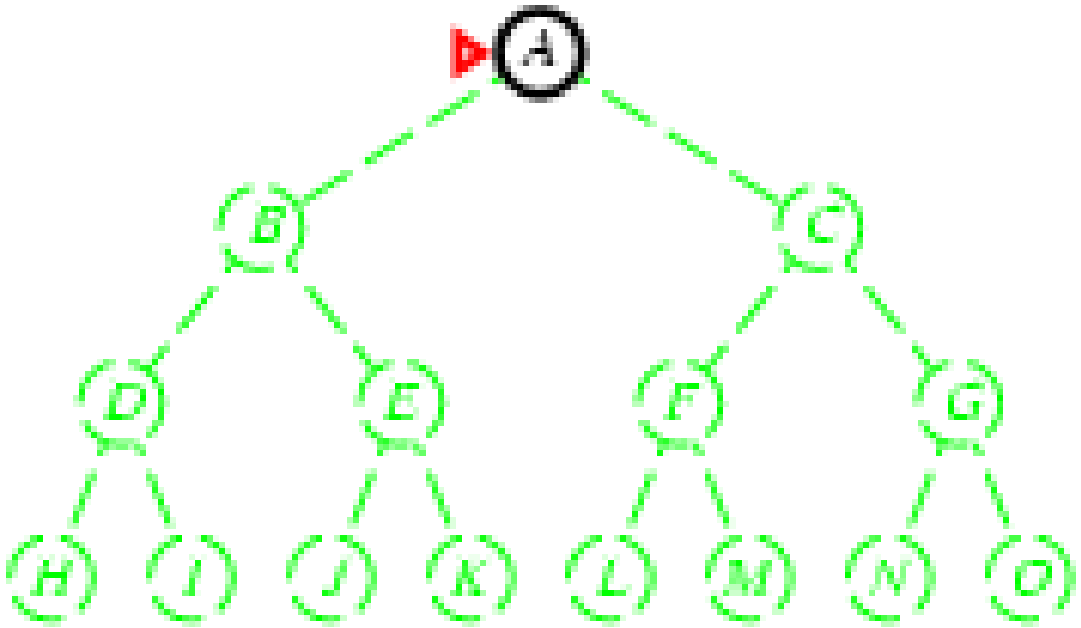
- **Complete ?** Yes
- **Optimal (i.e., admissible)?** if all operators have the same cost. Otherwise, not optimal but finds solution with shortest path length.
- **Exponential time and space complexity?** $O(b^{d+1})$, where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node
  - Will take a **long time to find solutions** with a large number of steps because must look at all shorter length possibilities first
    - A complete search tree of depth d where each non-leaf node has b children, has a total of $1 + b + b^2 + ... + b^d = (b^{(d+1)} - 1)/(b-1)$ nodes
    - For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are $1 + 10 + 100 + 1000 + ... + 10^{12} = (10^{13} - 1)/9 = O(10^{12})$ nodes in the complete search tree. If BFS expands 1000 nodes/sec and each node uses 100 bytes of storage, then BFS will take 35 years to run in the worst case, and it will use 111 terabytes of memory!
- **Space is the bigger problem (more than time)**

Depth-First Search (BFS)

- Expand deepest unexpanded node
- Fringe: nodes waiting in a queue to be explored
- Implementation:
  - *fringe* = LIFO stack, i.e., put successors at front

Example

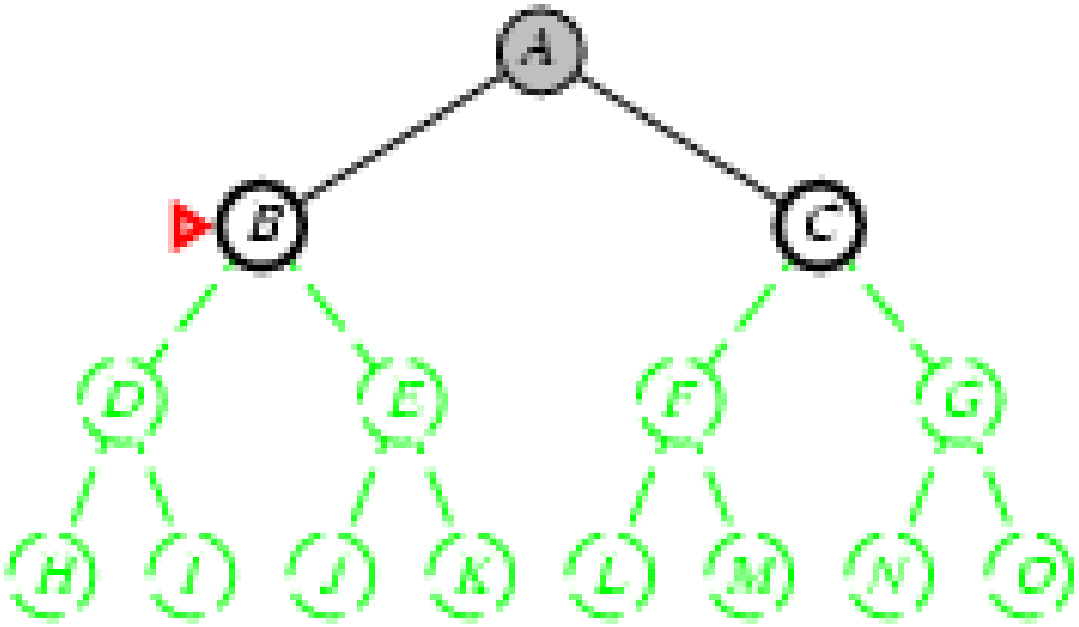
Is A a goal state?



Example

- queue=[B,C]

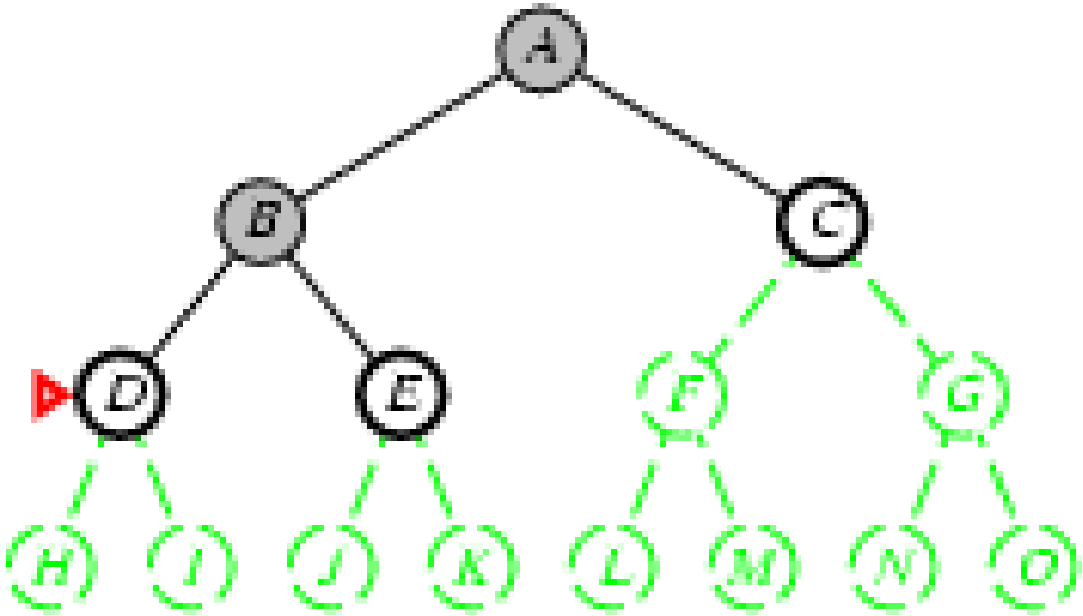
Is B a goal state?



Example

- queue=[D,E,C]

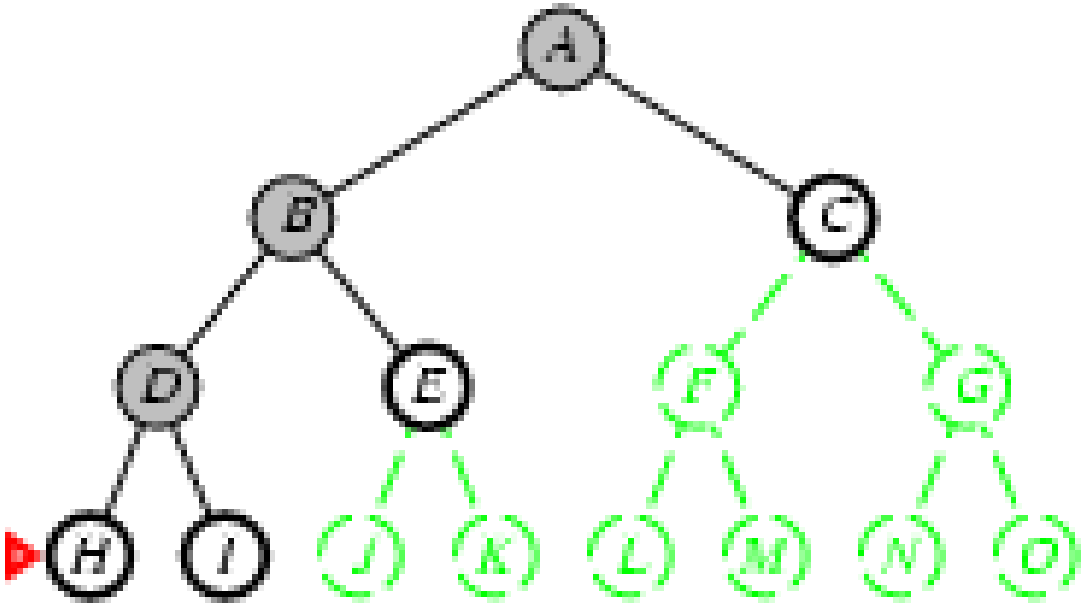
Is D = goal state?



Example

- queue=[H,I,E,C]

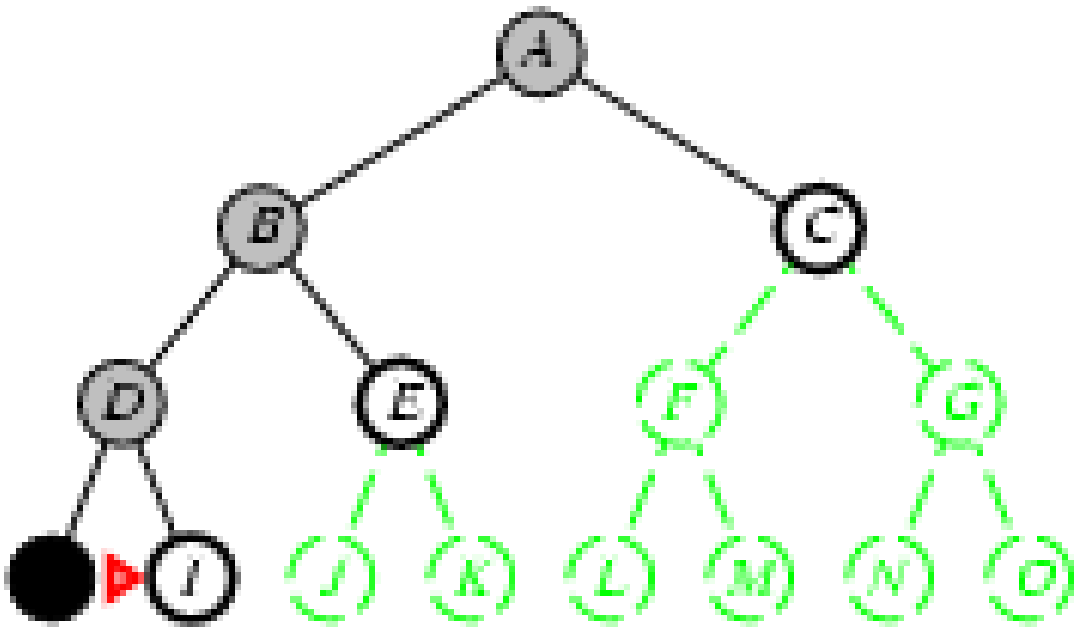
Is H = goal state?



Example

- queue=[I,E,C]

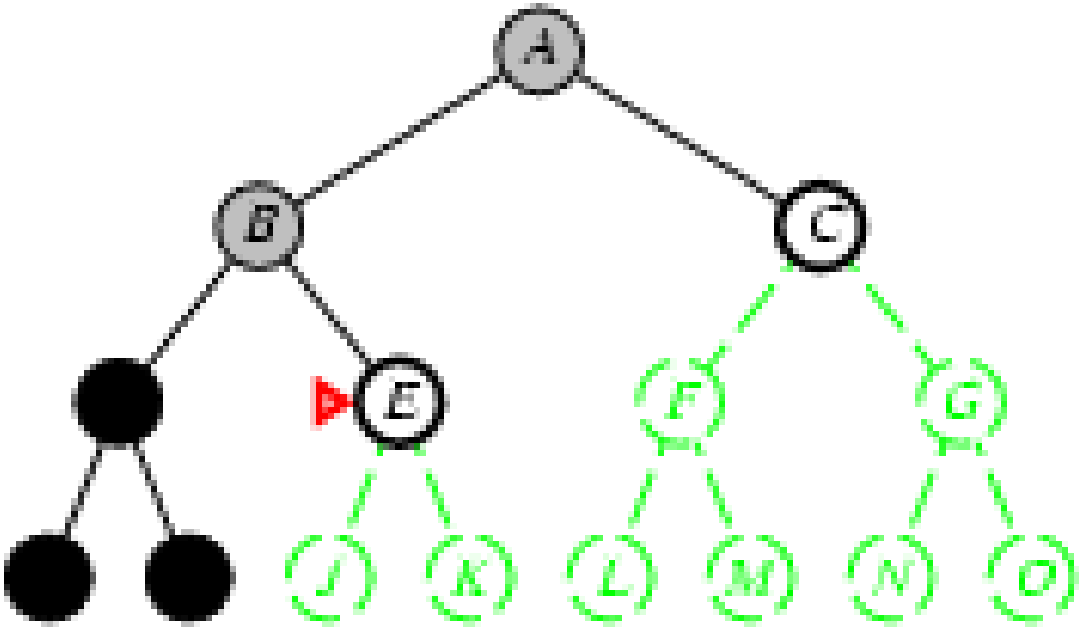
Is I = goal state?



Example

- queue=[E,C]

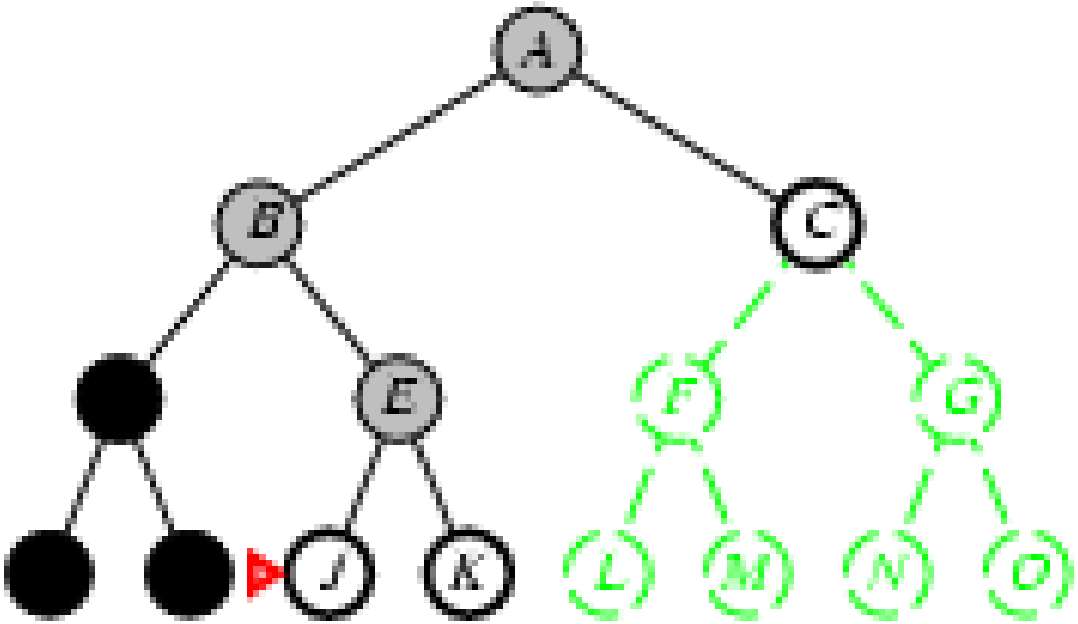
Is E = goal state?



Example

- queue=[J,K,C]

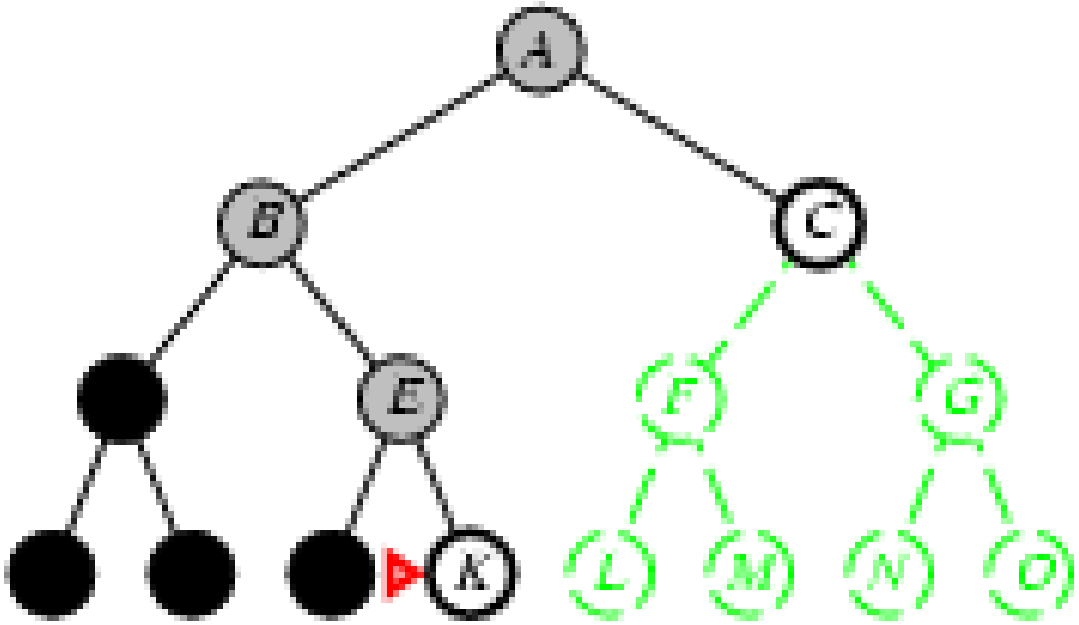
Is J = goal state?



Example

- queue=[K,C]

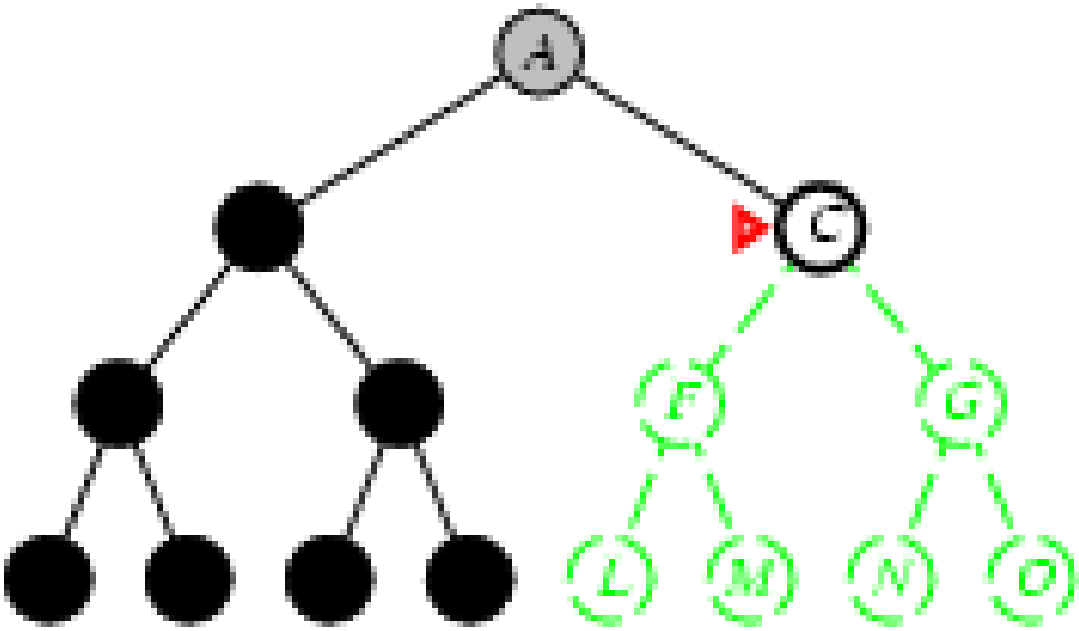
Is K = goal state?



Example

- queue=[C]

Is C = goal state?



Properties of DFS

- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D ("depth-limited search")
- **Not complete** (with or without cycle detection, and with or without a cutoff depth)
- **Exponential time**, $O(b^d)$, but only **linear space**, O(bd)
  - Can find **long solutions quickly** if lucky (and **short solutions slowly** if unlucky!)
  - When search hits a dead-end, can only back up one level at a time even if the "problem" occurs because of a bad operator choice near the top of the tree. Hence, only does "chronological backtracking"

BFS vs. DFS

Pros of BFS

- BFS will not get trapped exploring a blind alley. This contrasts with DFS, which may follow a single, unfruitful path for a very long time, perhaps forever, before the path actually terminates in a state that has no successors
- This is a particular problem of DFS (i.e., a state has a successor that is also of its ancestors) unless special care is expected to test for such a situation.
- If there is a solution, then BDS is guaranteed to find it.
- Furthermore, if there are multiple solutions, then a minimal solution (minimum # of steps) will be found.
- This is guaranteed by the fact that longer paths are never explored until all shorter ones have already been examined
- This contrasts with DFS, which may find a long path to a solution in one part of the tree, when a shorter exists in some other, unexplored part of thee tree

Pros of DFS

- DFS requires less memory since only the nodes on the current path are stored
- This contrast with BFS, where all of the tree that has so far been generated must be stored
- By chance, DFS may find a solution without examine much of the search space at all
- Contrast with BFS, in which all parts of the tree must be examined to level n before any nodes on level n+1 can be examined.
- This is particularly significant if many acceptable solutions exist,
- DFS can stop when one of them is found

Depth-Limited Search (DLS)

- To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don’t expand beyond depth L.
- DFS with depth limit *l*, i.e., nodes at depth *l* have no successors

Iterative Deepening Search (IDS)

- First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

until solution found do

DFS with depth cutoff c

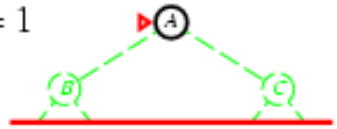
c = c+1

Example

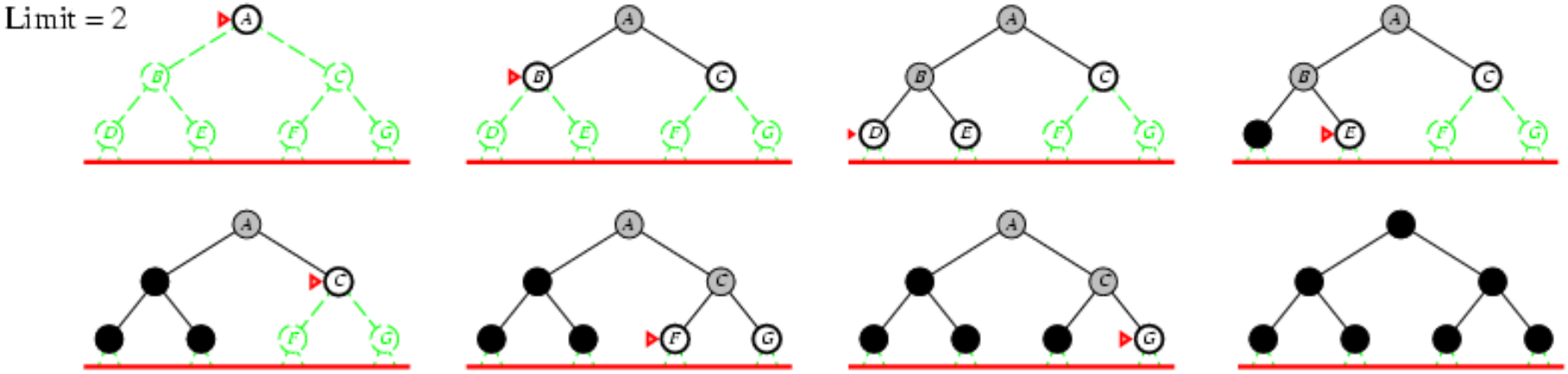
Limit = 0

Example

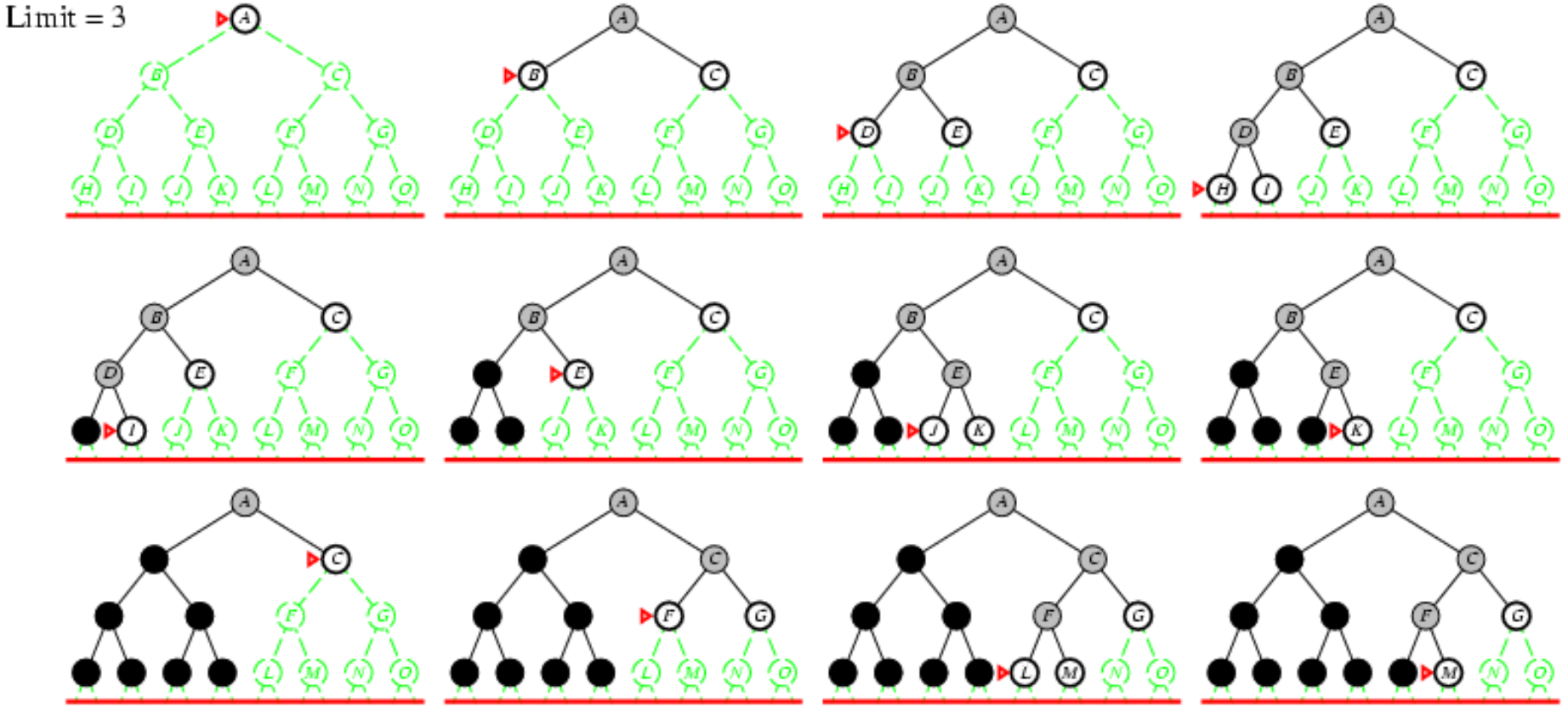
Limit = 1



Example



Example



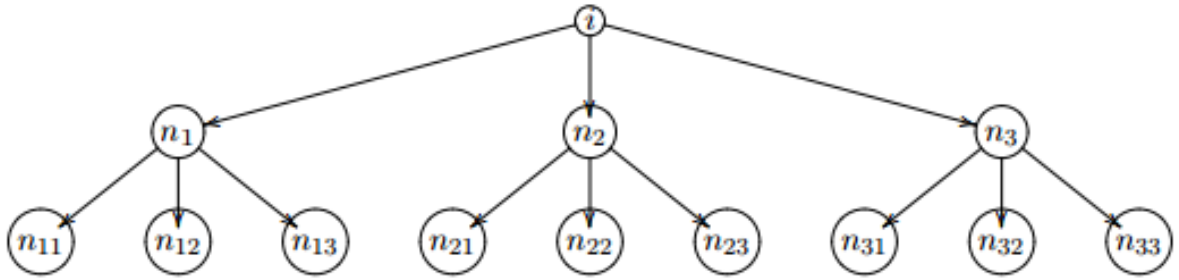
Properties of Iterative IDS

- Complete? Yes
- Exponential time complexity? $O(b^d)$, like BFS
- Linear space complexity? O(bd), like DFS
- Optimal? Yes, if step cost = 1 or increasing function of depth.

✓ Has advantage of BFS (i.e., completeness) and also advantages of DFS (i.e., limited space and finds longer paths more quickly)

✓ Generally preferred for **large state spaces** where **solution depth is unknown**

Comparative Analysis (b = 3,d = 2)



Iterative Deepening search

c	Visited	List
0		i
0	i	empty
1		i
1		n_1 n_2 n_3
1	n_1	n_2 n_3
1	n_2	n_3
1	n_3	empty
2		i
2		n_1 n_2 n_3
2		n_{11} n_{12} n_{13} n_2 n_3
2	n_{11}	n_{12} n_{13} n_2 n_3
2	n_{12}	n_{13} n_2 n_3
2	n_{13}	n_2 n_3
2		n_{21} n_{22} n_{23} n_3
2	n_{21}	n_{22} n_{23} n_3
2	n_{22}	n_{23} n_3
2	n_{23}	n_3
2		n_{31} n_{32} n_{33}
2	n_{31}	n_{32} n_{33}
2	n_{32}	n_{33}
2	n_{33}	empty

Depth-first search

Visited	List
	i
i	n_1 n_2 n_3
n_1	n_{11} n_{12} n_{13} n_2 n_3
n_{11}	n_{12} n_{13} n_2 n_3
n_{12}	n_{13} n_2 n_3
n_{13}	n_2 n_3
n_2	n_{21} n_{22} n_{23} n_3
n_{21}	n_{22} n_{23} n_3
n_{22}	n_{23} n_3
n_{23}	n_3
n_3	n_{31} n_{32} n_{33}
n_{31}	n_{32} n_{33}
n_{32}	n_{33}
n_{33}	empty

Breadth-first search

Visited	List
	i
i	n_1 n_2 n_3
n_1	n_2 n_3 n_{11} n_{12} n_{13}
n_2	n_3 n_{11} n_{12} n_{13} n_{21} n_{22} n_{23}
n_3	n_{11} n_{12} n_{13} n_{21} n_{22} n_{23} n_{31} n_{32} n_{33}
n_{11}	n_{12} n_{13} n_{21} n_{22} n_{23} n_{31} n_{32} n_{33}
n_{12}	n_{13} n_{21} n_{22} n_{23} n_{31} n_{32} n_{33}
n_{13}	n_{21} n_{22} n_{23} n_{31} n_{32} n_{33}
n_{21}	n_{22} n_{23} n_{31} n_{32} n_{33}
n_{22}	n_{23} n_{31} n_{32} n_{33}
n_{23}	n_{31} n_{32} n_{33}
n_{31}	n_{32} n_{33}
n_{32}	n_{33}
n_{33}	empty

Max-list-length = 1 + d(b-1)

Max-list-length = b^d

IDS

- 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*:

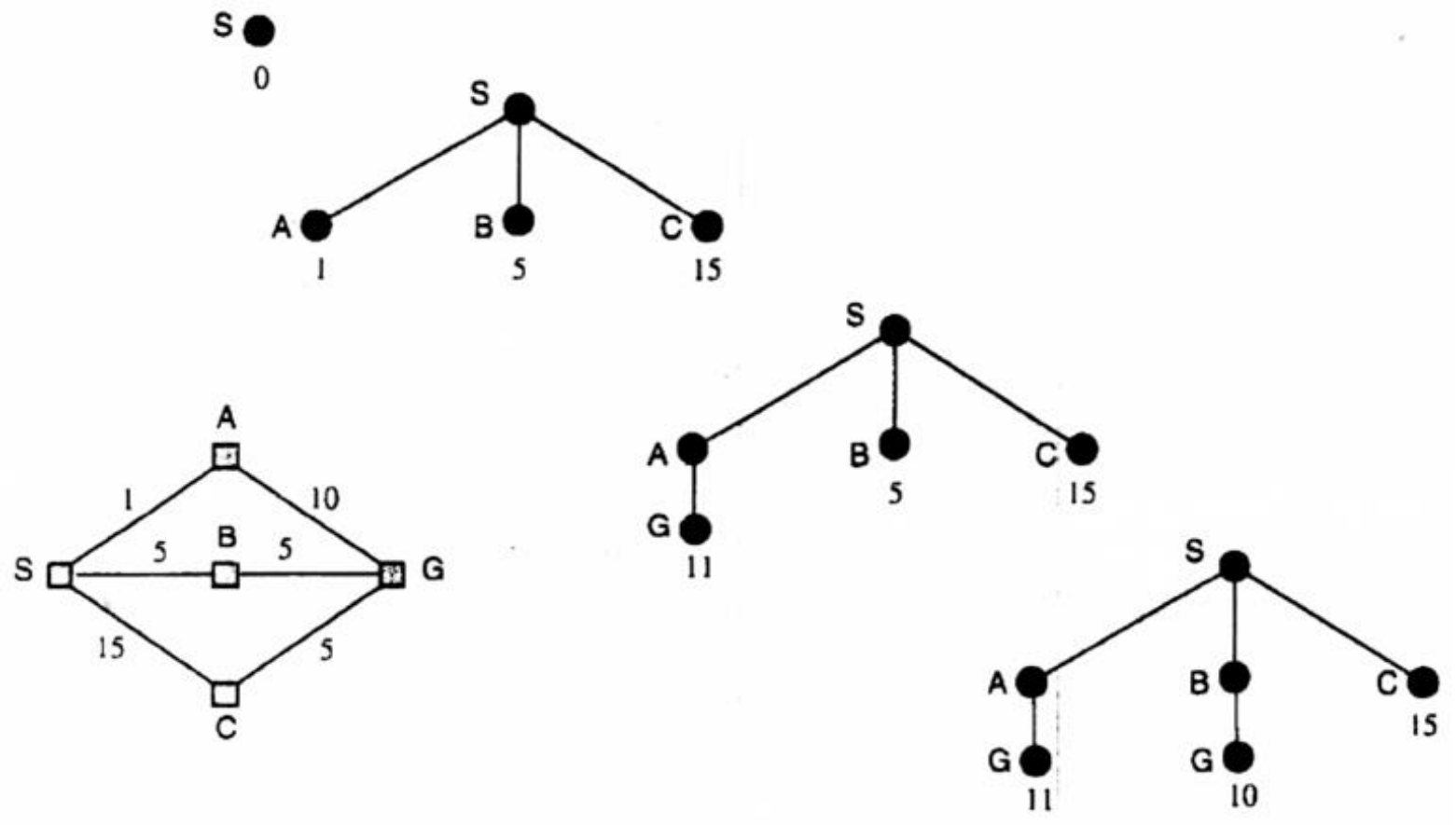
$N_{IDS} = (d+1)b^0 + d\,b\text{^}1 + (d-1)b\text{^}2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- For *b = 10, d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
    - An iterative deepening search from depth 1 all the way down to depth *d* expands only about 11 % more nodes than a single breadth-first or depth-limited search to depth *d*, when *b* = 10.

Uniform-Cost Search

- Enqueue nodes by **path cost**. That is, let g(n) = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.
- Called *"Dijkstra's Algorithm"* in the algorithms literature and similar to *"Branch and Bound Algorithm"* in operations research literature
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if all step costs all equal.

Example



Uniform-Cost Search Properties

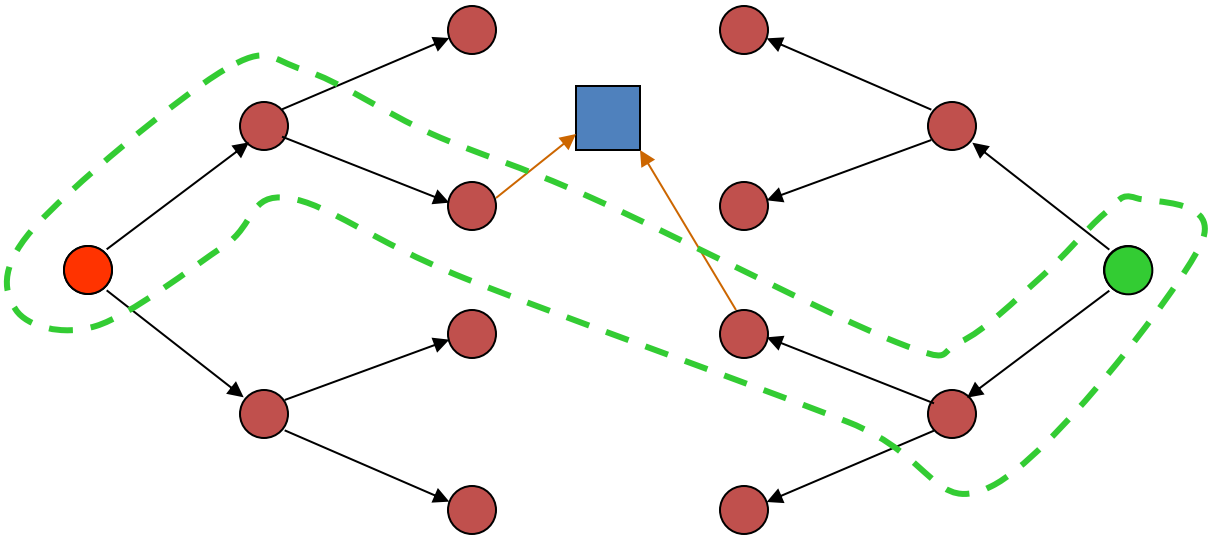
- Complete? Yes, if step cost $\geq \varepsilon$
  (otherwise it can get stuck in infinite loops)
- Time? # of nodes with *path cost* $\leq$ cost of optimal solution.
- Space? # of nodes with path cost $\leq$ cost of optimal solution.
- Optimal? Yes, for any step cost $\geq \varepsilon$

Bidirectional Search

- Run two simultaneous searches
  - one forward from the initial state another backward from the goal
  - stop when the two searches meet
- However, computing backward is difficult
  - A huge amount of goal states
  - At the goal state, which actions are used to compute it?
  - Can the actions be reversible to computer its predecessors?

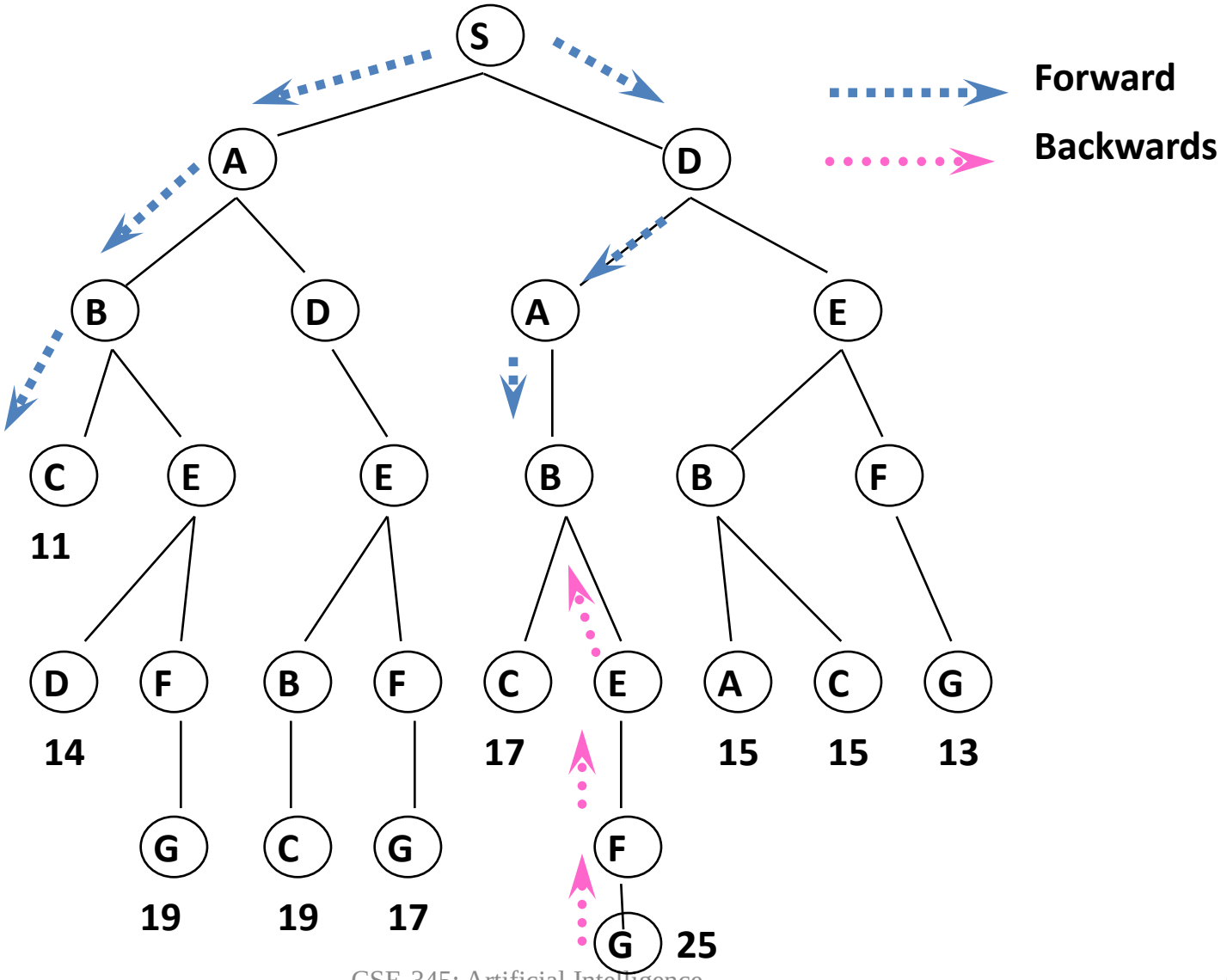
Bidirectional Search

2 fringe queues: FRINGE1 and FRINGE2



Time and space complexity = $O(b^{d/2})$ << $O(b^{d})$

Bidirectional Search



Evaluation of Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

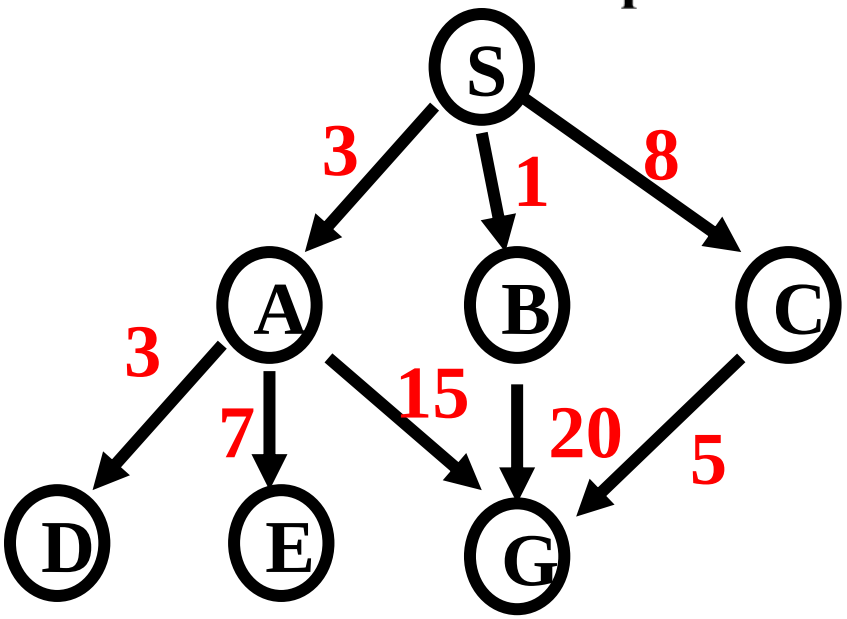
b – branching factor *d* – depth of optimal solution

m – maximum depth *l* – depth limit

C – is the cost of the optimal solution

ε – is a positive constant and it is assumed that every step costs at lest ε.

Example (BFS)

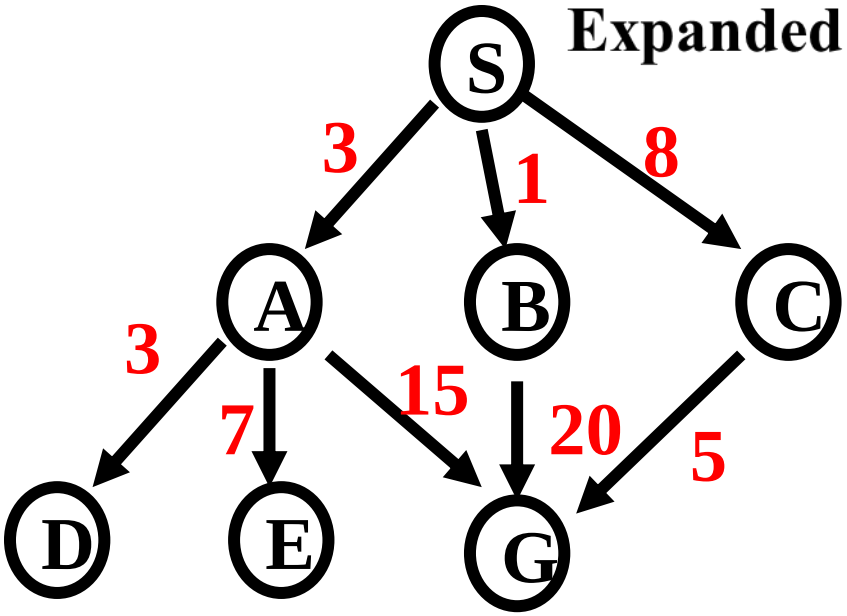


Expanded node	Nodes list
	{ S^0 }
S^0	{ A^3 B^1 C^8 }
A^3	{ B^1 C^8 D^6 E^{10} G^{18} }
B^1	{ C^8 D^6 E^{10} G^{18} G^{21} }
C^8	{ D^6 E^{10} G^{18} G^{21} G^{13} }
D^6	{ E^{10} G^{18} G^{21} G^{13} }
E^{10}	{ G^{18} G^{21} G^{13} }
G^{18}	{ G^{21} G^{13} }

Solution path found is S A G , cost 18

Number of nodes expanded (including goal node) = 7

Example (DFS)

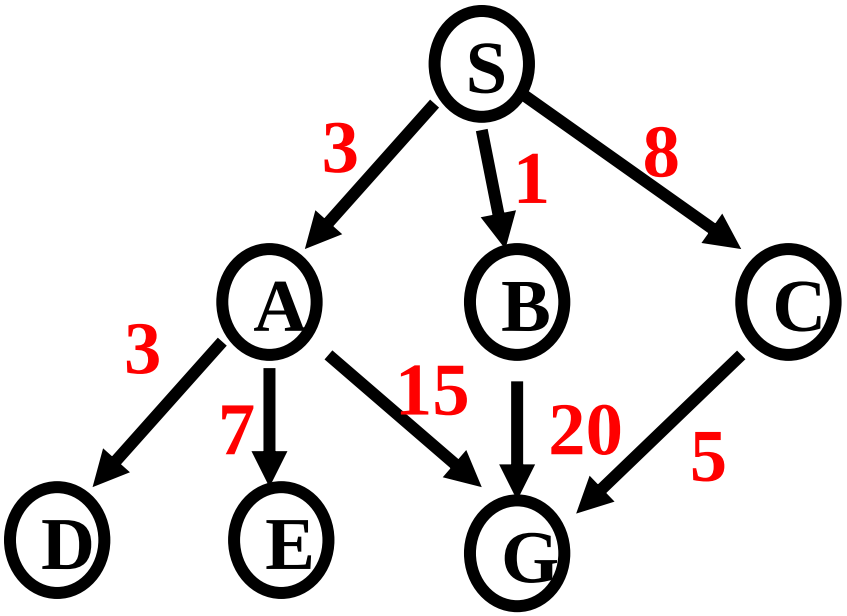


Expanded node	Nodes list
	$\{ S^0 \}$
S^0	$\{ A^3 \; B^1 \; C^8 \}$
A^3	$\{ D^6 \; E^{10} \; G^{18} \; B^1 \; C^8 \}$
D^6	$\{ E^{10} \; G^{18} \; B^1 \; C^8 \}$
E^{10}	$\{ G^{18} \; B^1 \; C^8 \}$
G^{18}	$\{ B^1 \; C^8 \}$

Solution path found is S A G , cost 18

Number of nodes expanded (including goal node) = 5

Example (Uniform-Cost Search)



Expanded node	Nodes list
	{ S^0 }
S^0	{ B^1 A^3 C^8 }
B^1	{ A^3 C^8 G^{21} }
A^3	{ D^6 C^8 E^{10} G^{18} G^{21} }
D^6	{ C^8 E^{10} G^{18} G^1 }
C^8	{ E^{10} G^{13} G^{18} G^{21} }
E^{10}	{ G^{13} G^{18} G^{21} }
G^{13}	{ G^{18} G^{21} }

Solution path found is S A G , cost 13

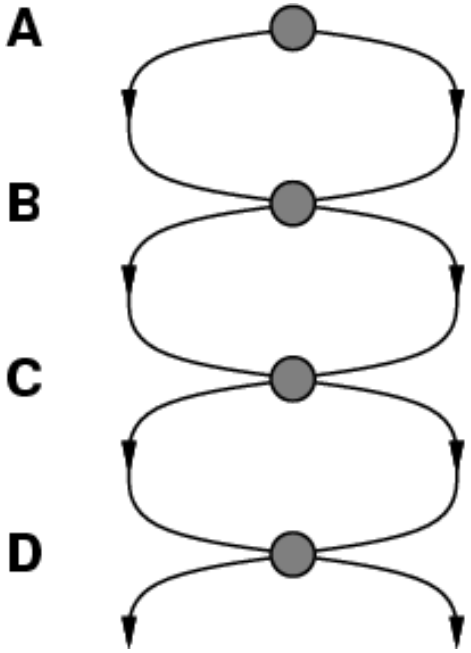
Number of nodes expanded (including goal node) = 7

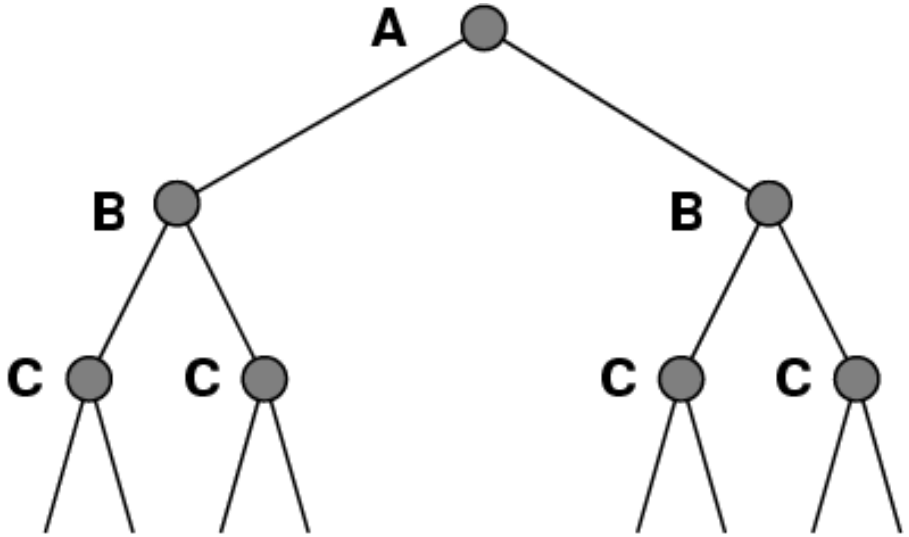
Avoiding Repeated States

- Ignored one of the most important complications to the search process: the possibility of wasting time by expanding states that have already been encountered and expanded before on some other path.
- For some problems, this possibility never comes up; each state can only be reached one way, e.g. 8-queens problem
- For many problems, repeated states are unavoidable, e.g. route-finding problems.
- The search trees for these problems are infinite, but if we prune some of the repeated states, we can cut the search tree down to finite size, generating only the portion of the tree that spans the state space graph.
- Even when the tree is finite, avoiding repeated states can yield an exponential reduction in search cost.

Repeated States

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





Solutions to Repeated States

- In increasing order of effectiveness in reducing size of state space and with increasing computational costs:
  1. Do not return to the state you just came from.
  2. Do not create paths with cycles in them.
  3. Do not generate any state that was ever created before.
- Net effect depends on frequency of "loops" in state space.

Graph Search

```
function graph-search (problem, QUEUEING-FUNCTION)
;; problem describes the start state, operators, goal test, and operator costs
;; queueing-function is a comparator function that ranks two states
;; graph-search returns either a goal node or failure
nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
  closed = {}
  loop
     if EMPTY(nodes) then return "failure"
     node = REMOVE-FRONT(nodes)
     if problem.GOAL-TEST(node.STATE) succeeds
        then return node.SOLUTION
     if node.STATE is not in closed
        then ADD(node, closed)
            nodes = QUEUEING-FUNCTION(nodes,
                EXPAND(node, problem.OPERATORS))
 end
  ;; Note: The goal test is NOT done when nodes are generated
  ;; Note: closed should be implemented as a hash table for efficiency
```

Graph Search

- Breadth-first search and uniform-cost search are optimal graph search strategies.
- Iterative deepening search and depth-first search can follow a non-optimal path to the goal.
- Iterative deepening search can be used with modification:
  - It must check whether a new path to a node is better than the original one
  - If so, IDS must revise the depths and path costs of the node's descendants.

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

The END