

# **ME 620: Fundamentals of Artificial Intelligence**

## **Lecture 5: Uninformed Search - I**



**Shyamanta M Hazarika**

Biomimetic Robotics and Artificial Intelligence Lab  
Mechanical Engineering and M F School of Data Sc. & AI  
IIT Guwahati

# Uninformed Search

---

- Review Problem Solving as Search
- State Spaces
- Graph Searching
- Generic Searching Algorithm
- Uninformed Search Strategies
  - Breadth-First Search
  - Depth-First Search
  - Iterative Deepening
  - Uniform-Cost Search
  - Bidirectional Search

# Formal Description of a Problem

---

- State Space** Define a state space that contains **all the possible configurations** of the relevant objects.
- Initial States** Specify one or more states within that space as **possible situations** from which the problem-solving can start.
- Goal States** Specify one or more states that would be **acceptable as solutions** to the problem
- Operators** Specify a **set of rules** that describes the actions (operators) available; information on what must be true, for the action can take place.

# State Space

---

The **set of all possible configurations** of the relevant objects is the space of problem states or the **problem space**. This is also called the **state space**.

For Example – 8-Puzzle

Each tile configuration is a problem state  
8-puzzle have a relatively small space  
362,880 i.e.,  $9!$  different configurations.

# Playing 8-Puzzle

## Problem Solving as State Space Search

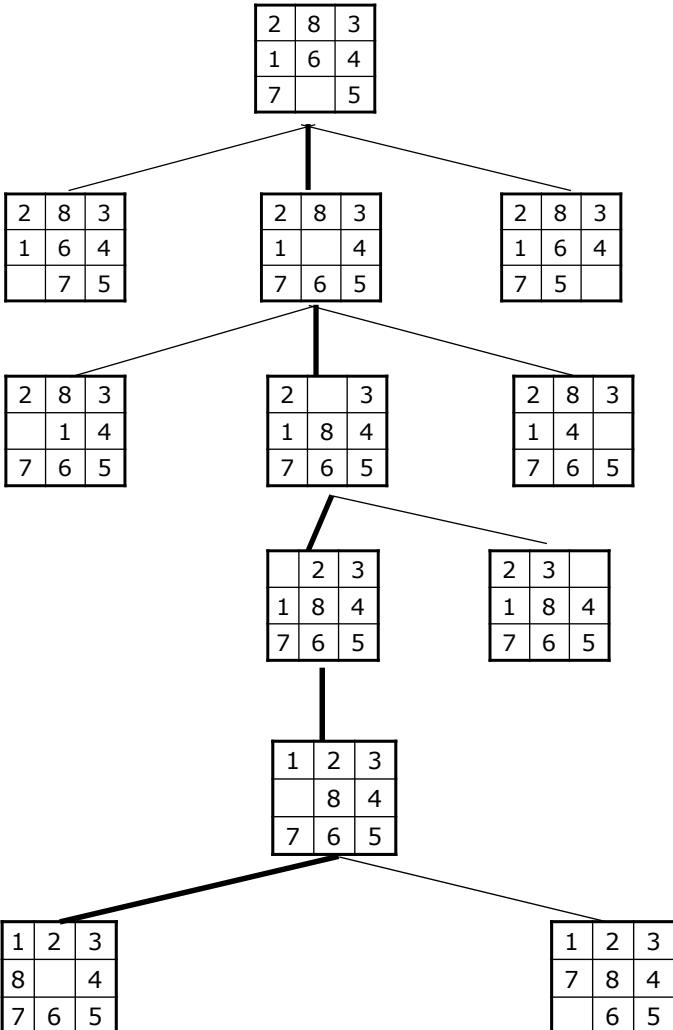
**s** ; Start Node

2	8	3
1	6	4
7		5

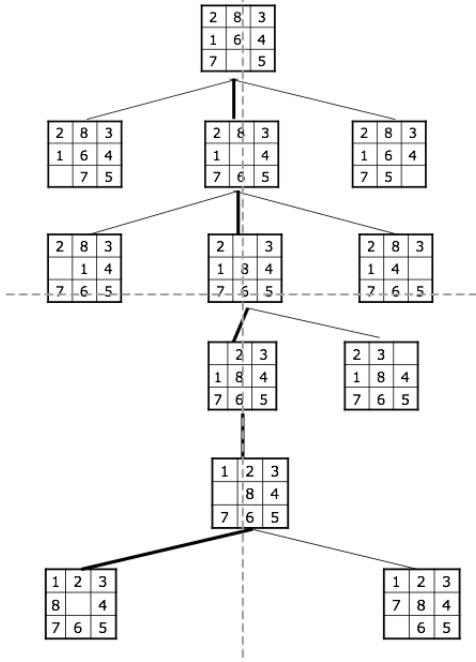
**g** ; Goal Node

1	2	3
8		4
7	6	5

# Problem Solving as State Space Search



# Problem Solving as State Space Search



Can be **abstracted** to the mathematical problem of **finding a path** from the **start node** to a **goal node** in a **directed graph**.

# Graphs: Basic Definitions

---

- A *graph*,  $G$ , comprises a **set  $V$**  of vertices and a **set  $E$**  of edges.
  - The vertex set can be anything, but is most commonly a collection of letters or numbers.
  - The set of edges is a set of doubleton subsets of  $V$ . That is  $E \subseteq \{\{a,b\} : a,b \in V \text{ and } a \neq b\}$ .
  - We denote the graph by  $G(V,E)$ .
- If  $G(V,E)$  is a graph and  $\{a,b\} \in E$ , then we say vertices  $a$  and  $b$  are *adjacent* and the edge  $\{a,b\}$  **joins** them or **connects** them or is *incident on* them. We call  $a$  and  $b$  the *endpoints* of the edge.
  - Two edges that share one vertex, such as  $\{a,b\}$  and  $\{b,c\}$  with  $a \neq c$ , are *adjacent* to each other.

# Graphs: Paths

---

- The notion of a path in a graph is intuitively clear but a little hard to pin down formally.
  - Suppose  $G(V, E)$  is a graph with vertices  $v_0, v_1, \dots, v_k$  (not necessarily distinct) and edges  $e_1, e_2, \dots, e_k$  (not necessarily distinct) in which edge  $e_i = \{v_{i-1}, v_i\}$  for  $i=1, \dots, k$ .
  - The alternating sequence of vertices and edges  $v_0 e_1 v_1 e_2 v_2 \dots e_k v_k$  is a *path from  $v_0$  to  $v_k$  of length  $k$*  (The length is the number of edges, not vertices).

# Directed Graphs (Digraphs)

---

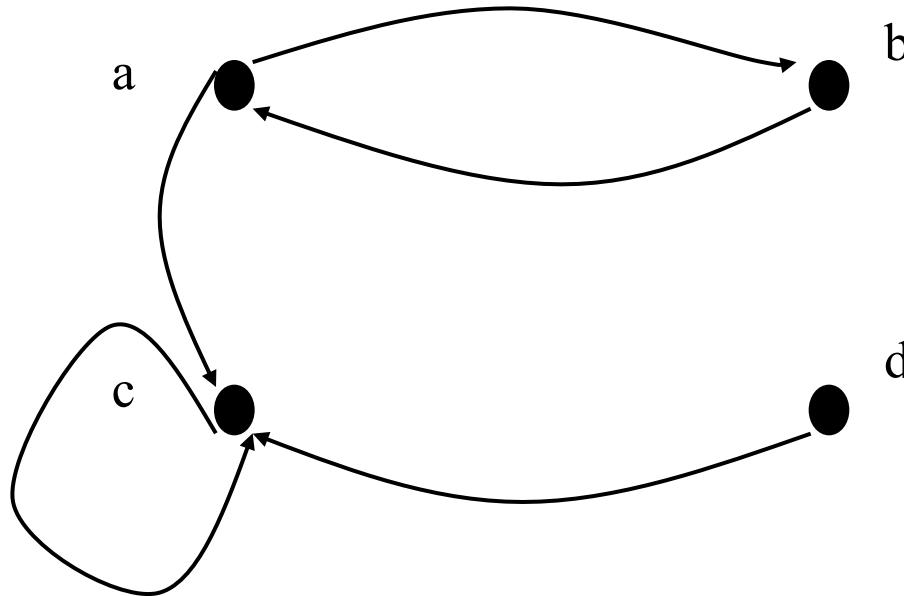
- A *directed graph* (*digraph*, for short)  $G$  consists of a set  $V$  of vertices and a set  $E$  of *directed edges*.
  - A directed edge is an ordered pair of elements of  $V$ . Put another way, the pair  $G(V,E)$  with  $E \subseteq V \times V$  is a digraph
  - Digraphs allow for loops of the form  $(a,a)$ , It is also possible to have two edges  $(a,b)$  and  $(b,a)$  between vertices  $a$  and  $b$ .
  - We take the ordering of the pairs in  $E$  to give each edge a direction: namely the edge  $(a,b)$  goes from  $a$  to  $b$ .
  - When we draw the digraph, we draw the edge  $(a,b)$  as an arrow from  $a$  to  $b$ .

# Digraphs: Example

---

## □ Example:

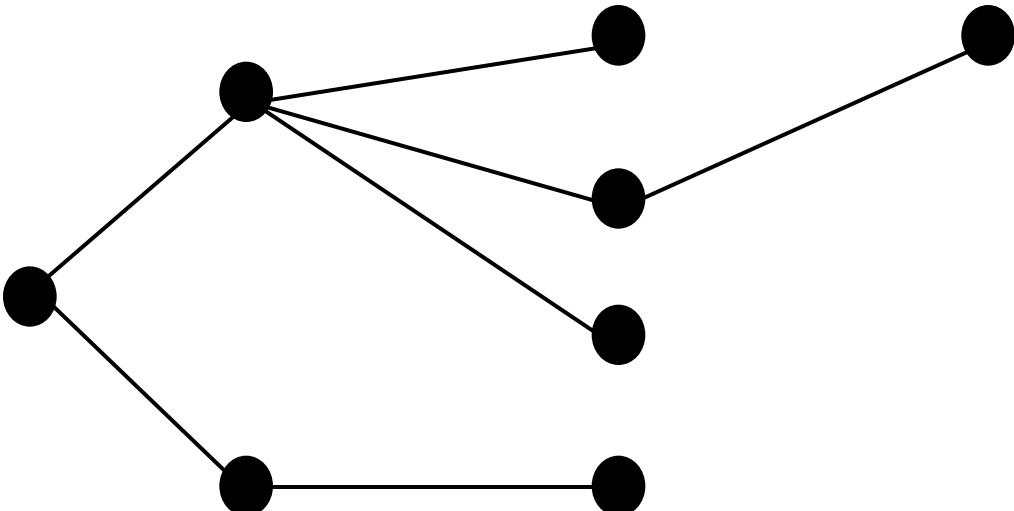
- $V = \{a, b, c, d\}$
- $E = \{(a, b), (a, c), (b, a), (c, c), (d, c)\}$



# Trees

---

- A *tree* is a connected graph with no cycles.
- Trees come up in many contexts: tournament brackets, family trees, organizational charts, and decision trees, being a few examples.



# Graph Searching

---

- The problem of finding a **sequence of actions** to achieve a goal is abstracted as **searching for paths in directed graphs**.
- To solve a problem, define the underlying **search space** and apply a search algorithm to that search space.
- **Searching in graphs** provides an appropriate abstract model of problem solving independent of a particular domain.

# Formalizing search in a state space

---

- A state space is a **graph**  $(V, E)$  where
  - $V$  is a **set of nodes** and
  - $E$  is a **set of arcs**, and each arc is directed from a node to another node
- Each **node** is a **data structure** that contains a **state description** plus other information such as the parent of the node, operator that generated the node from that parent, and other bookkeeping data
- Each **arc** corresponds to an instance of one of the operators.

# Formalizing search in a state space

---

- Each arc has a fixed, **positive cost** associated with it corresponding to the cost of the operator.
- Each node has a set of **successor nodes** corresponding to all of the legal operators that can be applied at the source node's state.
  - The process of expanding a node means to generate all of the successor nodes and add them and their associated arcs to the state-space graph
- One or more nodes are designated as **start nodes**.
- A **goal test** predicate is applied to a state to determine if its associated node is a goal node.

# Formalizing search in a state space

---

- A **solution** is a sequence of operators that is associated with a path in a state space from a start node to a goal node.
- The **cost of a solution** is the sum of the arc costs on the solution path.
  - If all arcs have the same (unit) cost, then the solution cost is just the length of the solution (number of steps / state transitions)

# Formalizing search in a state space

---

- **State-space search** is the process of searching through a state space for a solution by **making explicit** a sufficient portion of an **implicit** state-space graph to find a goal node.
  - For large state spaces, it isn't practical to represent the whole space.
  - Initially  $V=\{S\}$ , where  $S$  is the start node;
    - when  $S$  is expanded, its successors are generated and those nodes are added to  $V$  and the associated arcs are added to  $E$ .
    - This process continues until a goal node is found.

# Formalizing search in a state space

---

- **State-space search** is the process of searching through a state space for a solution by **making explicit** a sufficient portion of an **implicit** state-space graph to find a goal node.
- Each node implicitly or explicitly represents a partial solution path (and cost of the partial solution path) from the start node to the given node.
  - In general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix.

# 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?

# Important Parameters

---

- Maximum number of successors of any state of the search tree
  - branching factor  $b$  of the search tree
- Minimal length of a path in the state space between the initial and a goal state
- depth  $d$  of the shallowest goal node in the search tree
- Maximum depth  $m$  of the state space.

# Graph Search

---

## Procedure **GRAPHSEARCH**

Create a search graph **G** consisting solely of the start node **s**.

Put **s** on a list called **OPEN**

Create a list called **CLOSED** that is initially empty.

Loop: If **OPEN** is empty, exit with failure

Select first node on **OPEN**, remove it from **OPEN** and put it on **CLOSED**. Call this node **n**.

If **n** is goal node, exit successfully with the solution obtained by tracing a path along the pointers from **n** to **s** in **G**

# Graph Search

Expand node  $n$  generating the set  $M$  of its successors and install them as successors of  $n$  in  $G$ .

Establish a pointer to  $n$  from those members of  $M$  that were not already in  $G$ . Add these members of  $M$  to OPEN.

For each member of  $M$  already in  $G$  decide whether or not to redirect its pointer to  $n$ .

For each member of  $M$  already on CLOSED, decide for each of its descendants in  $G$  whether or not to redirect its pointer.

Reorder the list OPEN, either according to some arbitrary scheme or according to heuristic merit.

Go LOOP.

# Uninformed vs. informed search

---

## □ Uninformed search strategies

- Also known as “blind search,” **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\*

# Uninformed Search Strategies

---

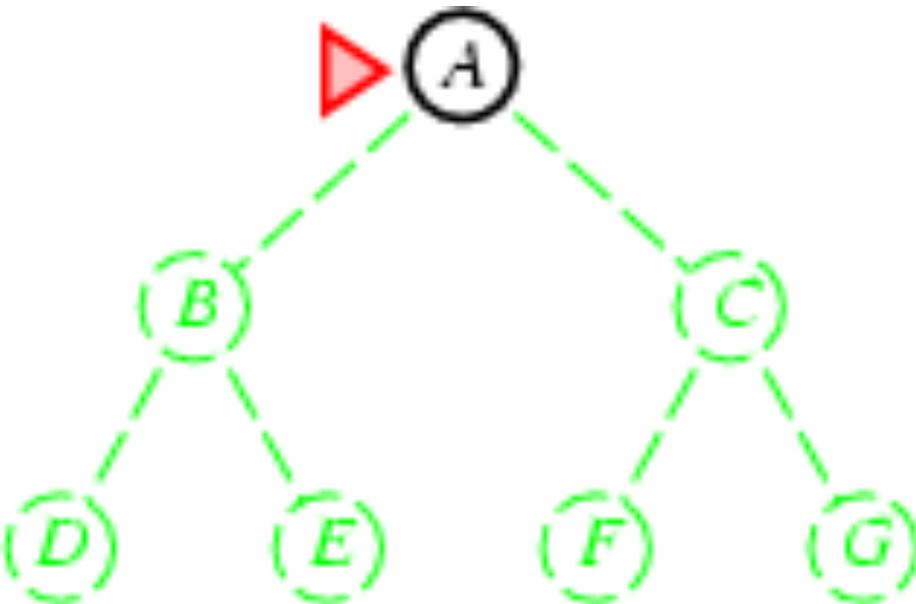
No information from the problem domain is used in ordering the nodes on OPEN, some arbitrary scheme is used.

- Breadth-First Search
- Depth-First Search
- Iterative Deepening
- Uniform-Cost Search
- Bidirectional Search

# Breadth-first search

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

Is A a goal state?



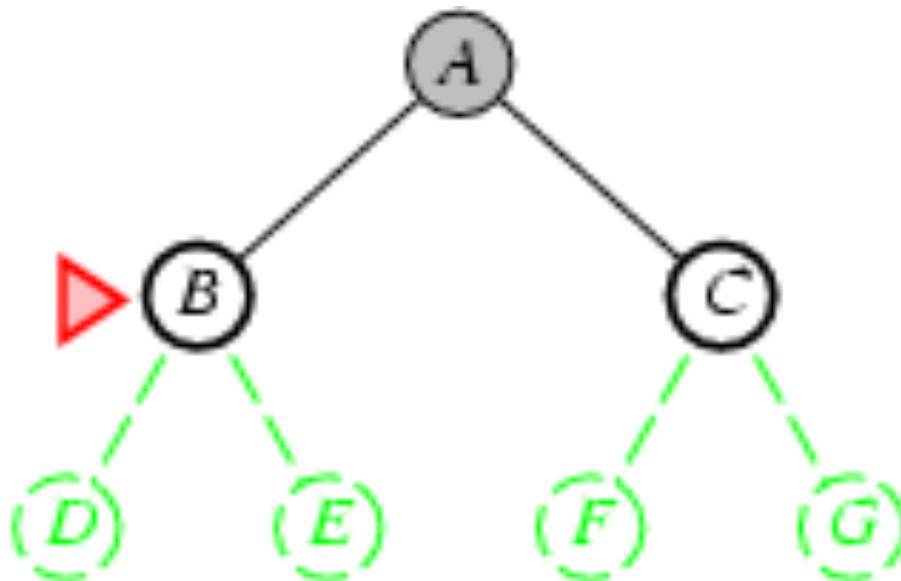
# Breadth-first search

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

Expand:

fringe = [B,C]

Is B a goal state?



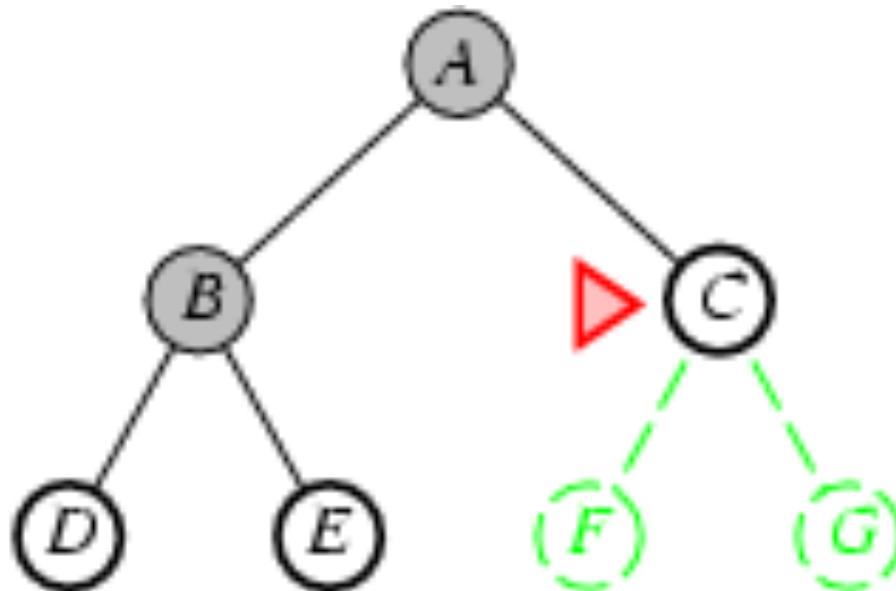
# Breadth-first search

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

Expand:

fringe=[C,D,E]

Is C a goal state?



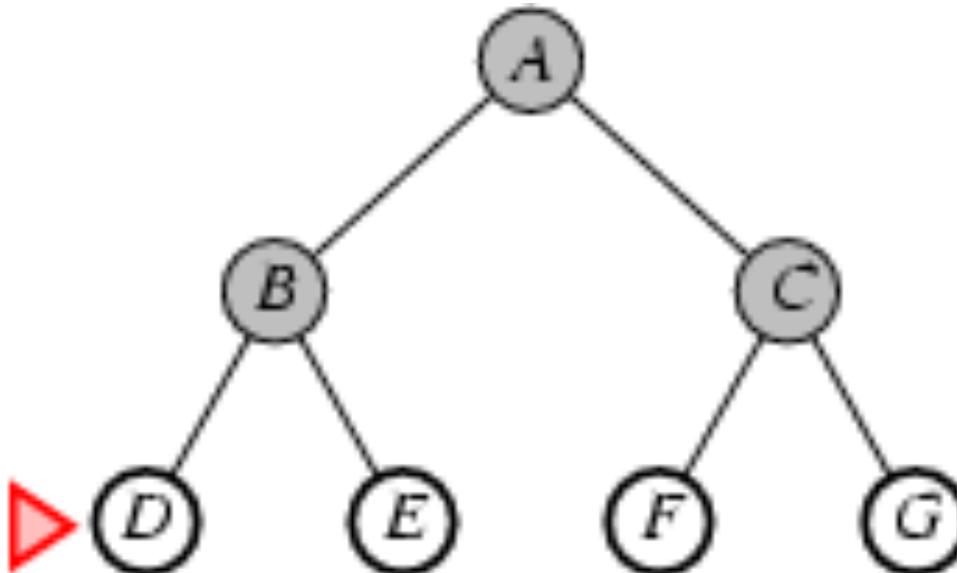
# Breadth-first search

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

Expand:

fringe=[D,E,F,G]

Is D a goal state?



# Properties of Breadth-first search

---

## □ Complete?

- Yes it always reaches goal (if  $b$  is finite)

## □ Time?

- $1+b+b^2+b^3+\dots +b^d = O(b^d)$
- This is the number of nodes we generate

## □ Space?

- $O(b^d)$
- keeps every node in memory, either in fringe or on a path to fringe.

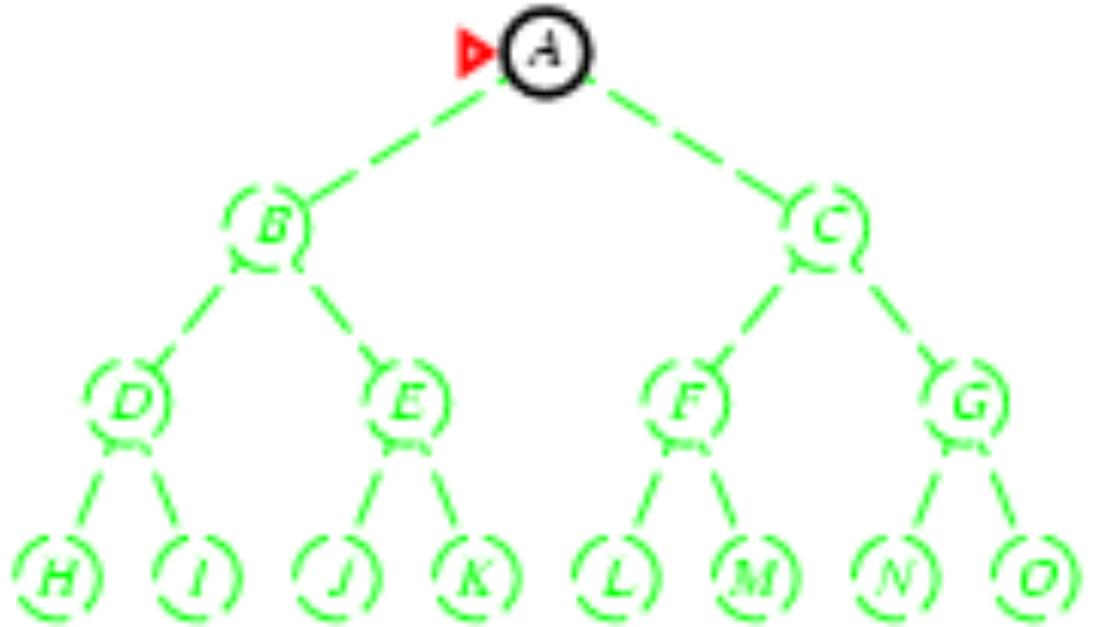
## □ Optimal?

- Yes (if we guarantee that deeper solutions are less optimal, e.g. step-cost=1).

# Depth-first search

- Expand *deepest* unexpanded node
- Implementation:
  - *fringe* = Last In First Out (LIPO) queue, i.e., put successors at front

Is A a goal state?

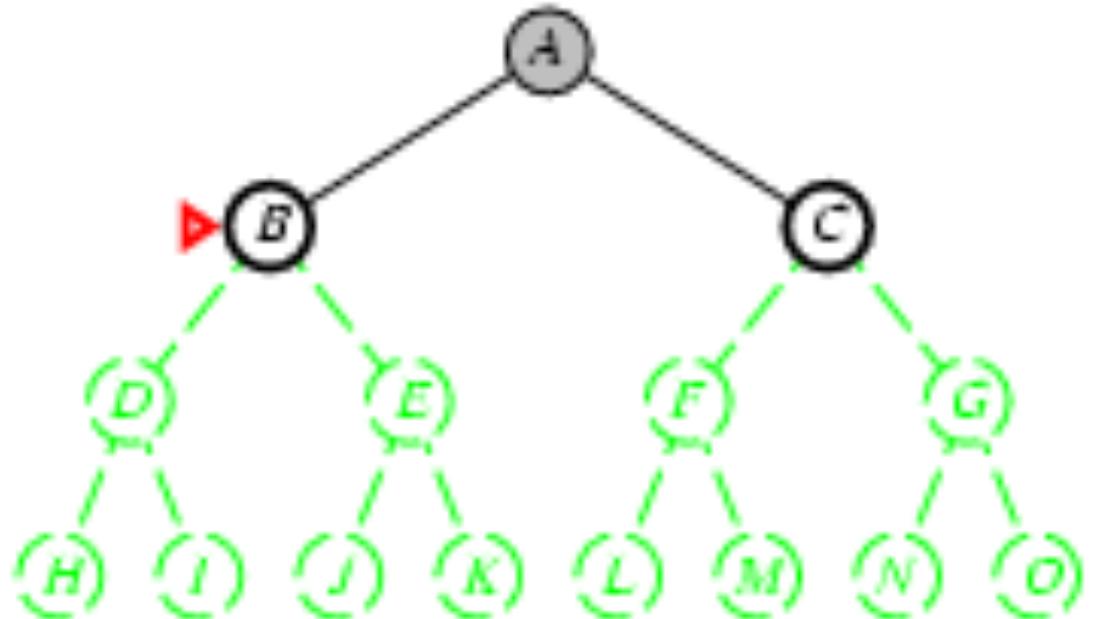


# Depth-first search

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

Expand:  
queue=[B,C]

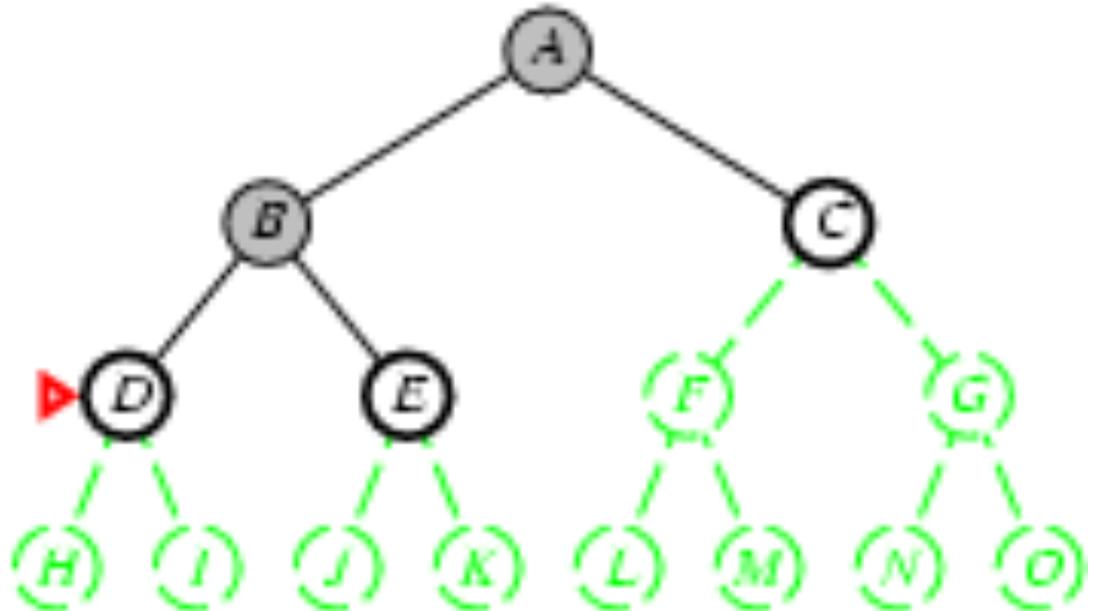
Is B a goal state?



# Depth-first search

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

Expand:  
queue=[D,E,C]  
  
Is D = goal state?



# Depth-first search

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

Expand:

queue=[H,I,E,C]

Is H = goal state?

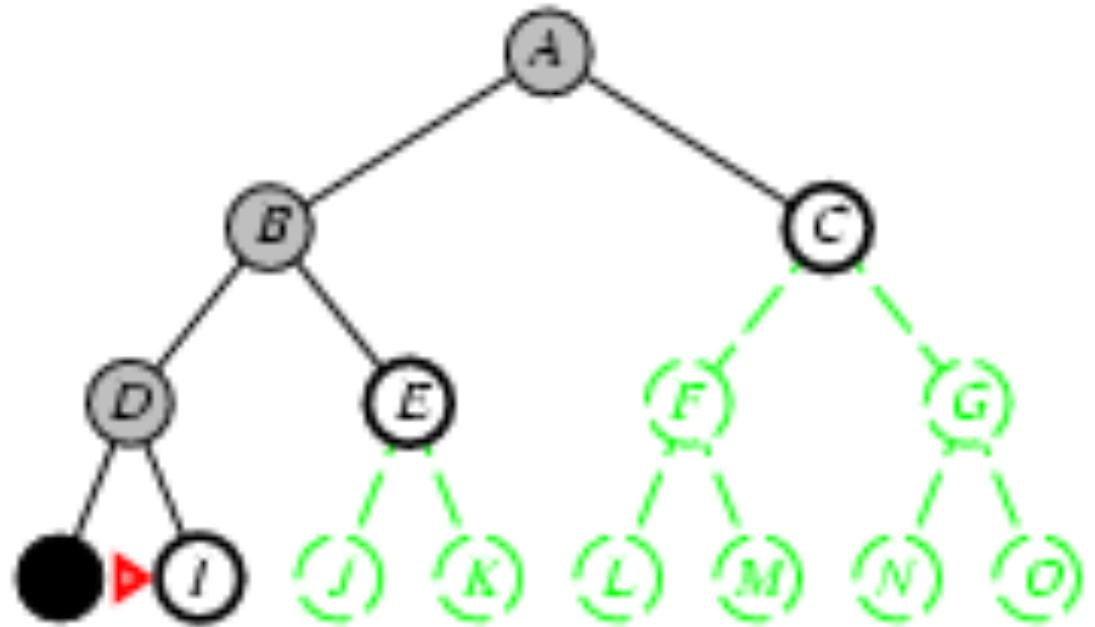


# Depth-first search

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

Expand:  
queue=[I,E,C]

Is I = goal state?

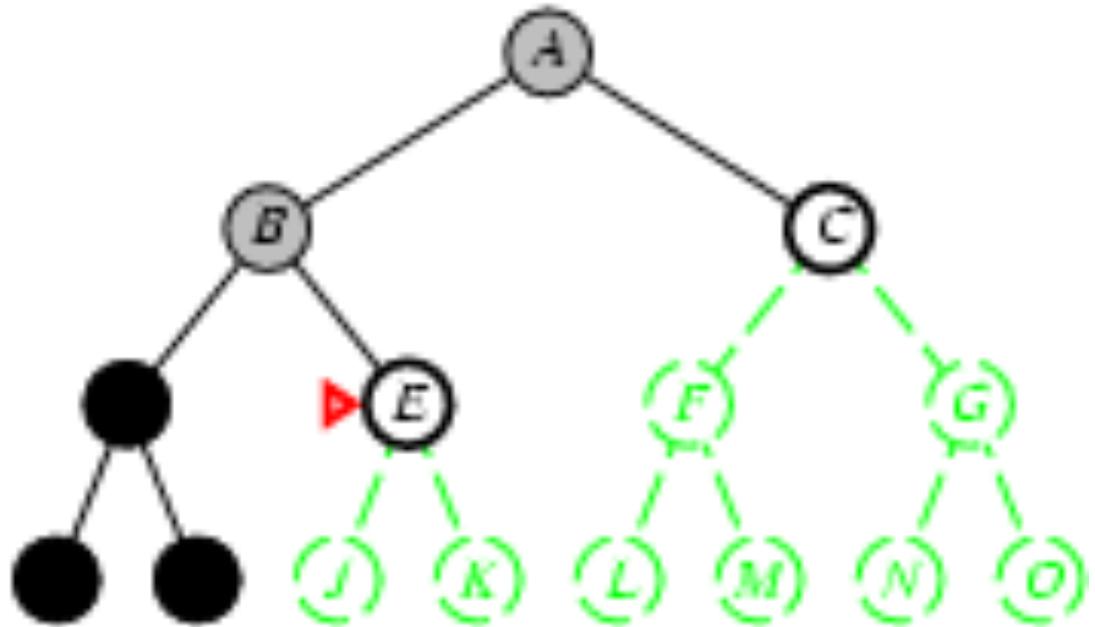


# Depth-first search

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

Expand:  
queue=[E,C]

Is E = goal state?



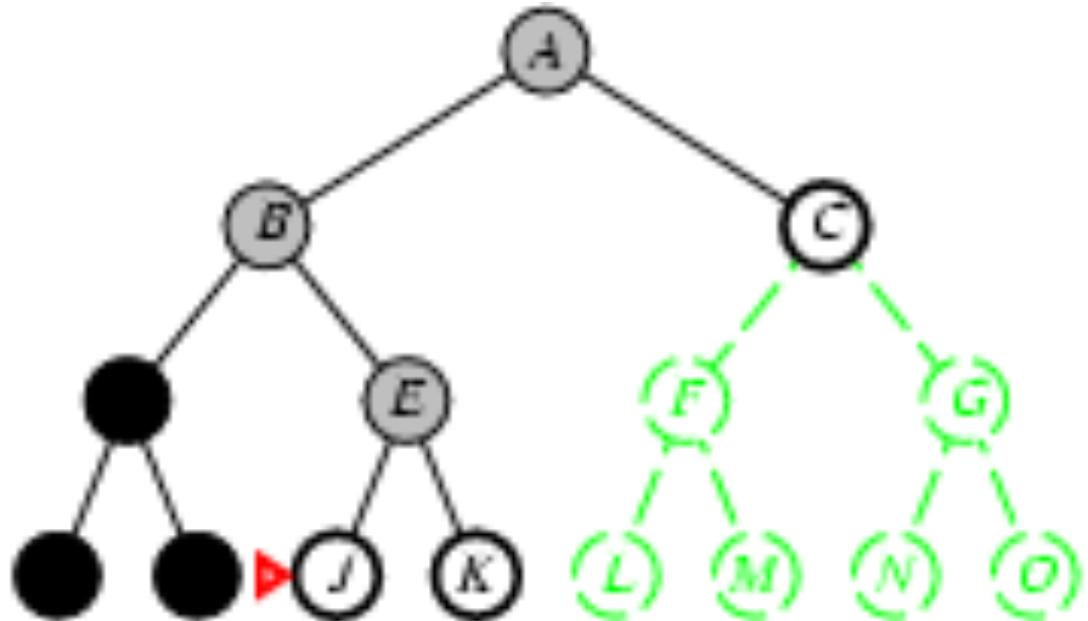
# Depth-first search

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

Expand:

queue=[J,K,C]

Is J = goal state?

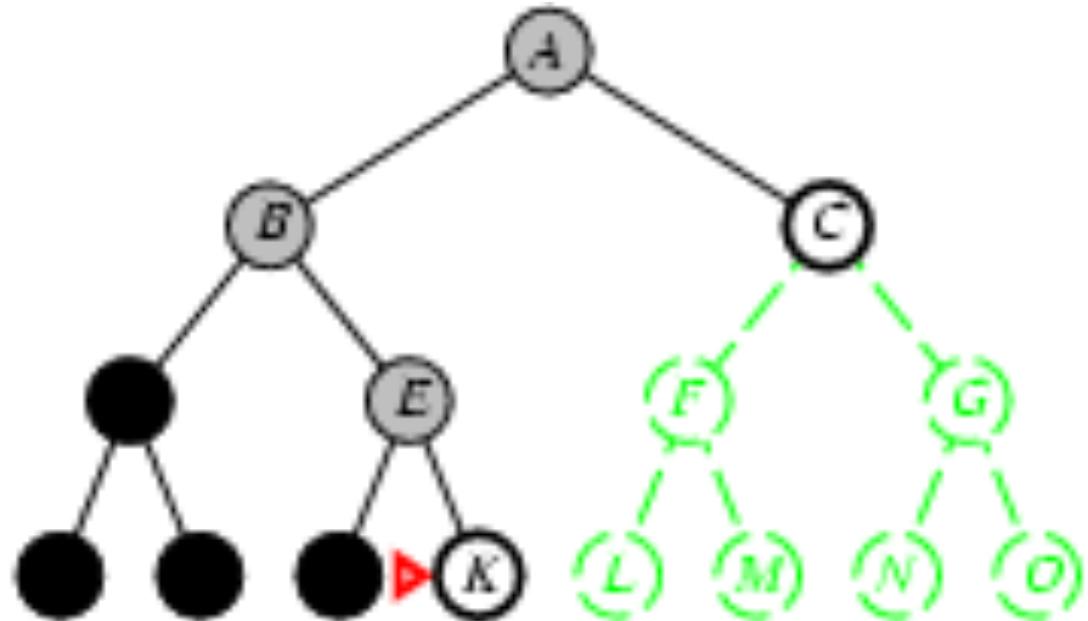


# Depth-first search

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

Expand:  
queue=[K,C]

Is K = goal state?

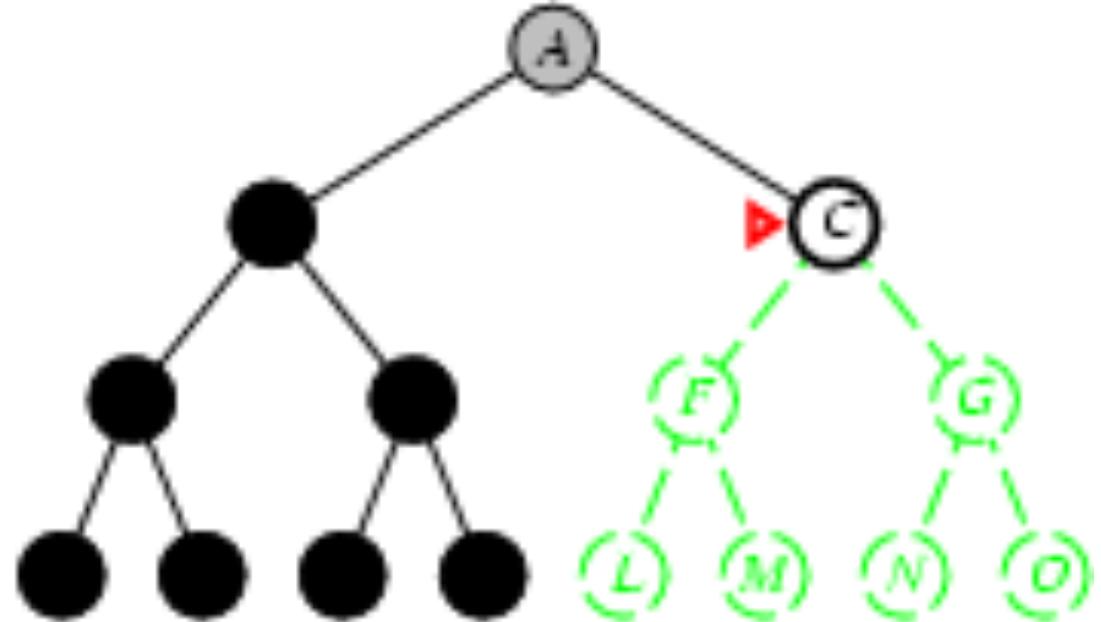


# Depth-first search

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

Expand:  
queue=[C]

Is C = goal state?

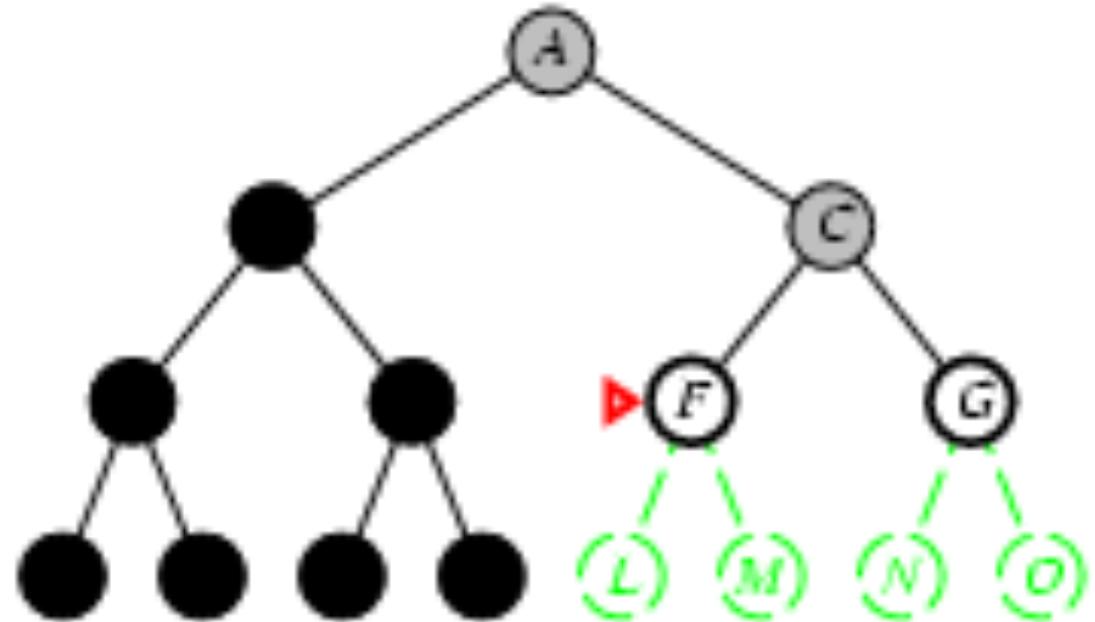


# Depth-first search

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

Expand:  
queue=[F,G]

Is F = goal state?



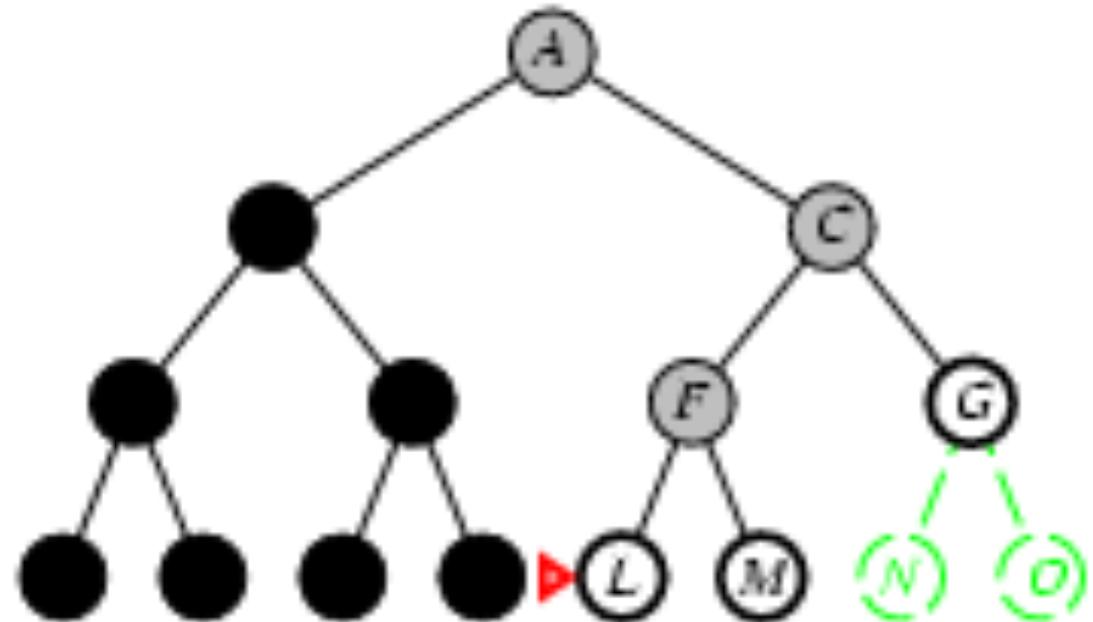
# Depth-first search

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

Expand:

queue=[L,M,G]

Is L = goal state?

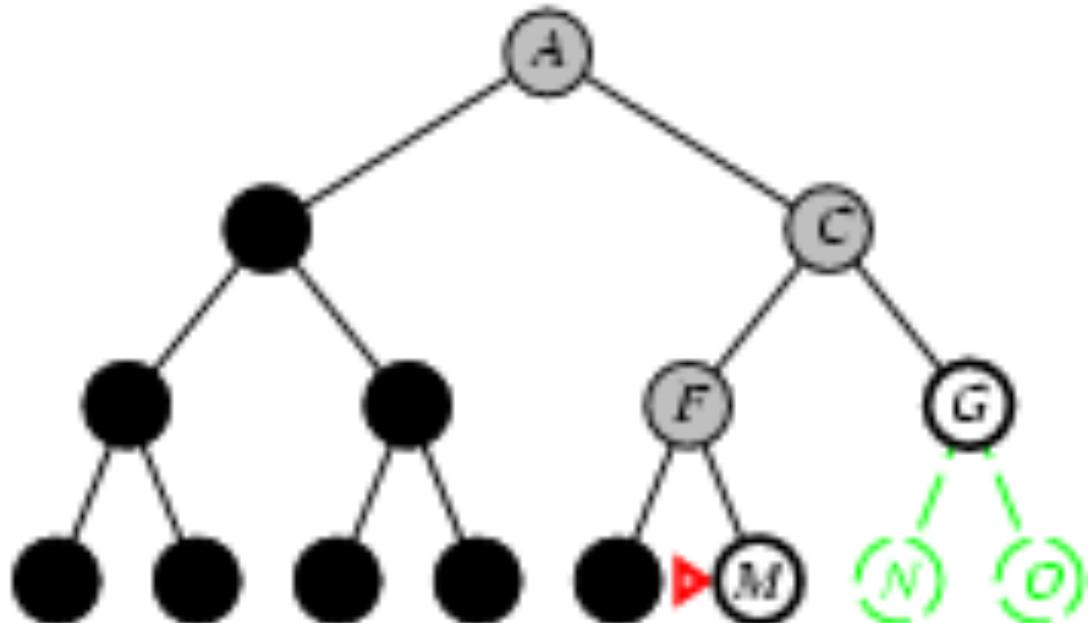


# Depth-first search

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

Expand:  
queue=[M,G]

Is M = goal state?



# Properties of Depth-first search

---

## □ Complete?

- No: fails in infinite-depth spaces
- Can modify to avoid repeated states along path

## □ Time?

- $O(b^m)$  with  $m$ =maximum depth
- 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! (we only need to remember a single path + expanded unexplored nodes)

## □ Optimal?

- No (It may find a non-optimal goal first)