Uninformed Search

Search

- Searching through a state space involves the following:
  - A set of states
  - Operators and their costs
  - Start state
  - A test to check for goal state

The basic search algorithm

```
Let L be a list containing the initial state
Loop
        if L is empty return failure
        Node ← select (L)
          if Node is a goal
                  then return Node
                       (the path from initial state to Node)
        else generate all successors of Node, and
             merge the newly generated states into L
End Loop
```

- Maintains a list , OPEN which contains all expanded nodes
- Order of placing nodes in OPEN list

Search algo : Key issues

- How to handle loops in search tree?
- Should we return path or a Node?
- Which node to select for expansion?
- Which path to find (Any path/shortest path)

Evaluating Search strategies

- Completeness: Is the strategy guaranteed to find a solution if one exists?
- Optimality: Does the solution have low cost or the minimal cost?
- What is the search cost associated with the time and memory required to find a solution?
  - Time complexity:
  - Space complexity:

different search strategies

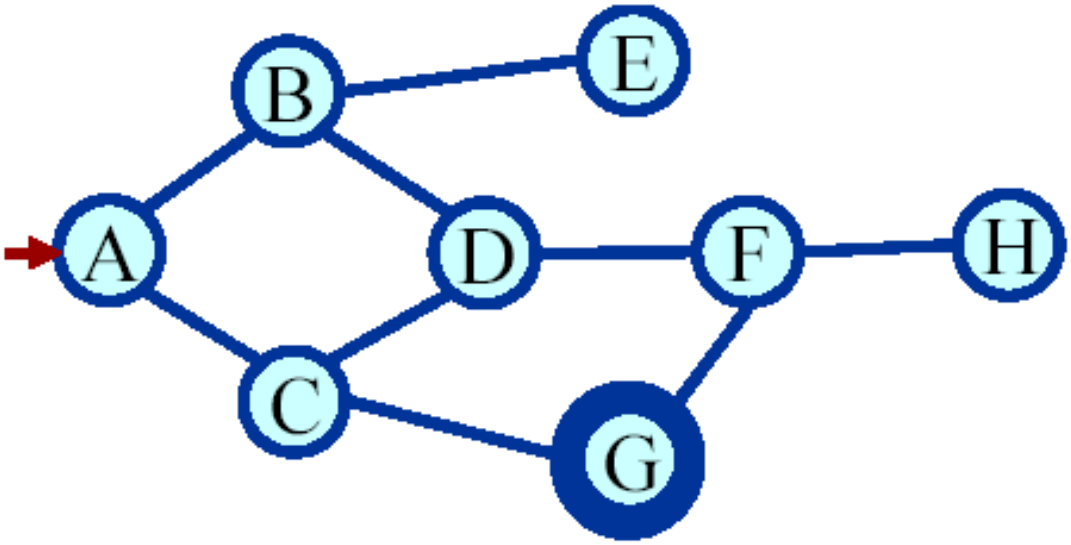
1. Blind Search strategies or Uninformed search
   a. Depth first search
   b. Breadth first search
   c. Iterative deepening search
   d. Iterative broadening search
2. Informed Search
3. Constraint Satisfaction Search
4. Adversary Search

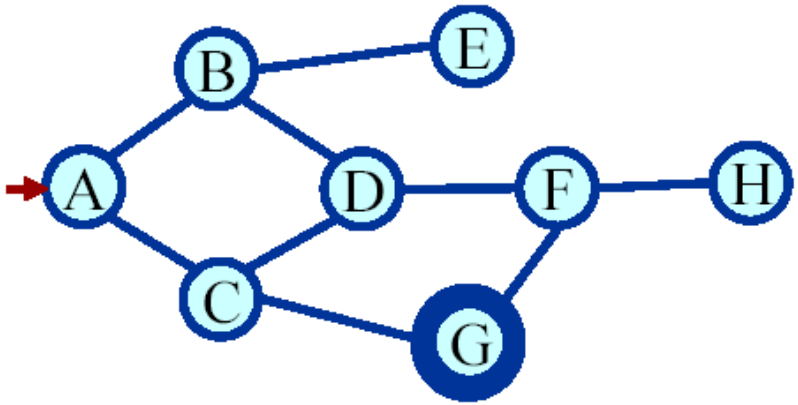
Blind Search

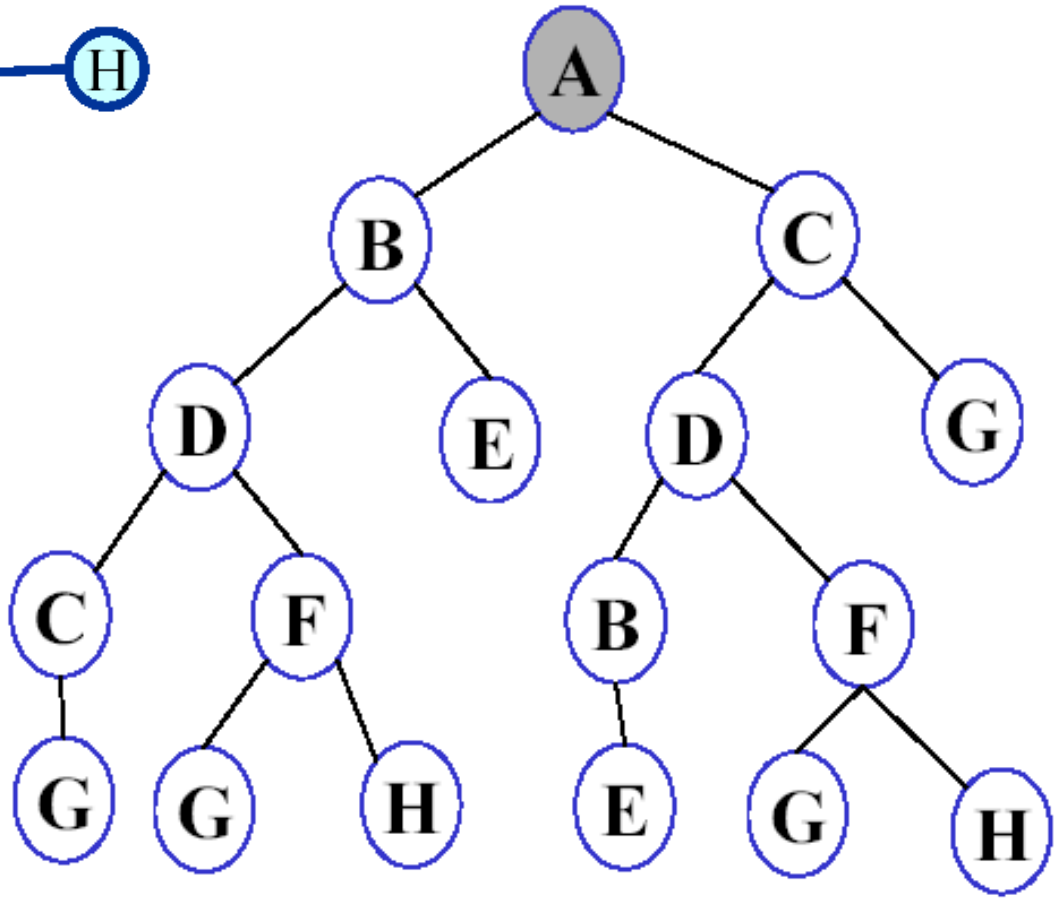
- Does not use any extra information about the problem domain.
  - BFS or Breadth First Search
  - DFS or Depth First Search

Search Tree

- A complete search tree
  - All possible paths excluding cycles







Search tree for state space graph

Search Tree – Terminology

- Root Node: The node from which the search starts.
- Leaf Node: A node in the search tree having no children.
- Ancestor/Descendant: X is an ancestor of Y if either X is Y's parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
- Branching factor: the maximum number of children of a non-leaf node in the search tree
- Path: A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

Node data structure

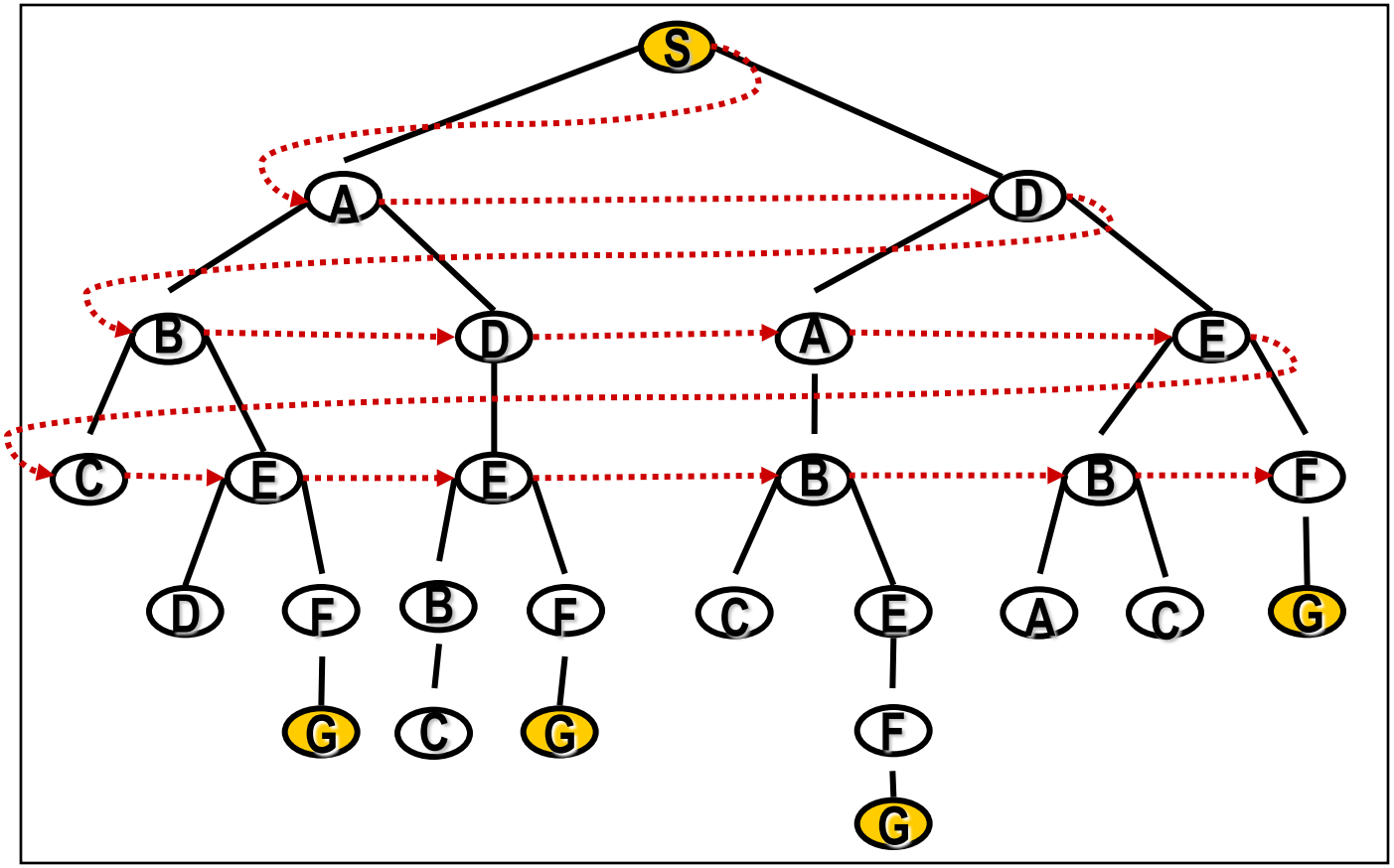
- A node used in the search algorithm is a data structure which contains the following:
  1. A state description
  2. A pointer to the parent of the node
  3. Depth of the node
  4. The operator that generated this node
  5. Cost of this path (sum of operator costs) from the start state
- The nodes that the algorithm has generated are kept in a data structure called OPEN or fringe.
- Initially only the start node is in OPEN.
- Some search algorithms keep track of the closed nodes in a data structure called CLOSED.

Breadth First Search

Breadth first search
Let *fringe* be a list containing the initial state
Loop
    if *fringe* is empty return failure
    Node ← remove-first (fringe)
     if Node is a goal
        then return the path from initial state to Node
    else generate all successors of Node, and
        (merge the newly generated nodes into *fringe*)
        add generated nodes to the back of *fringe*
End Loop

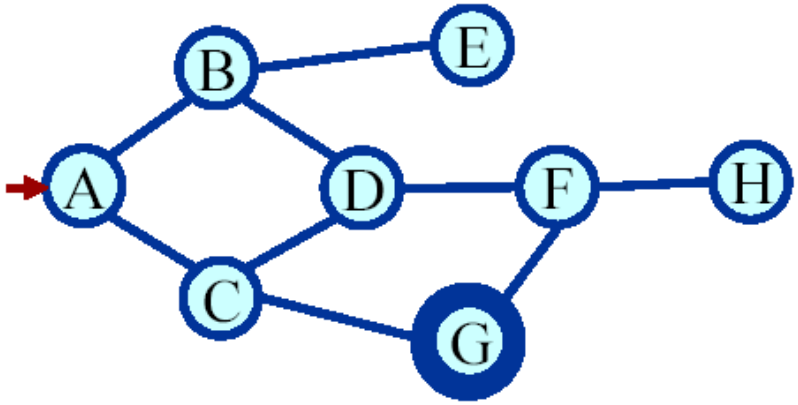
- nodes will be expanded in a FIFO order

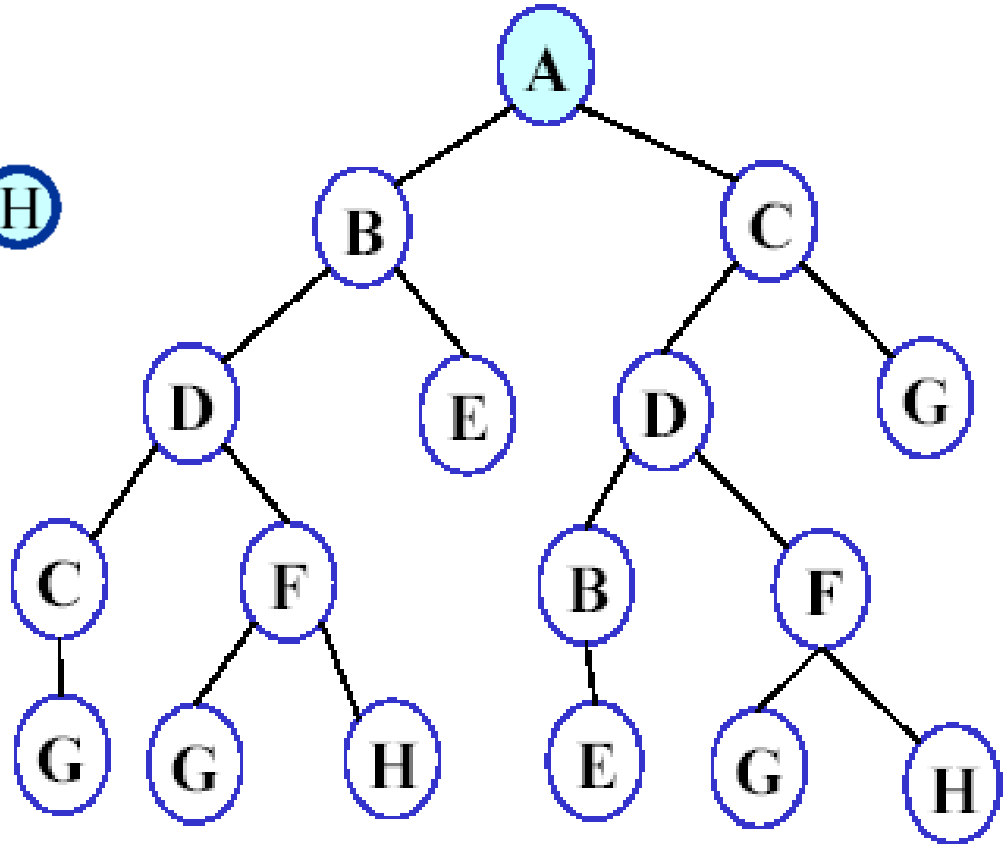
Breadth-first search:



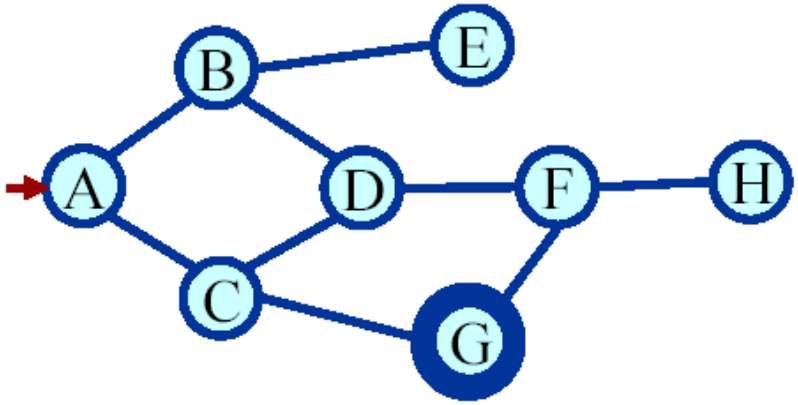
- Move downwards, level by level, until goal is reached.

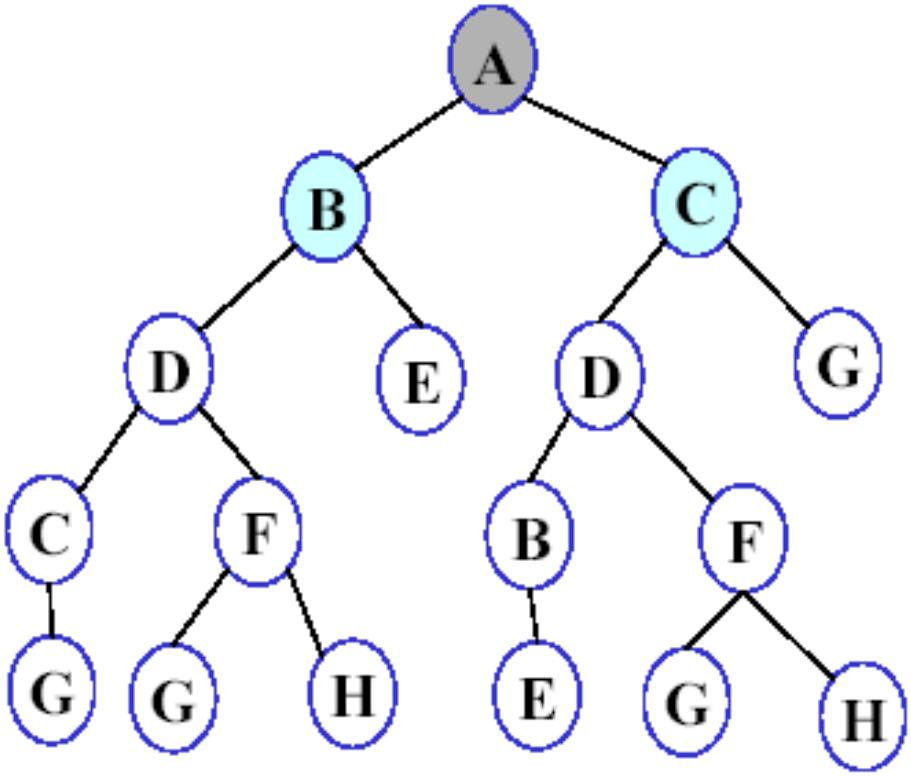
BFS illustrated



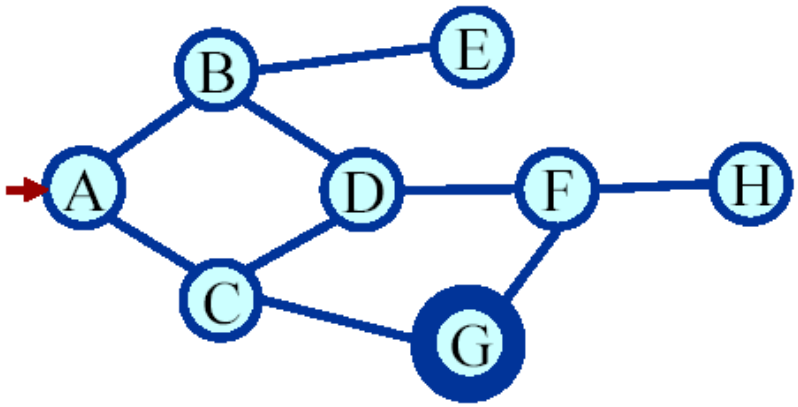


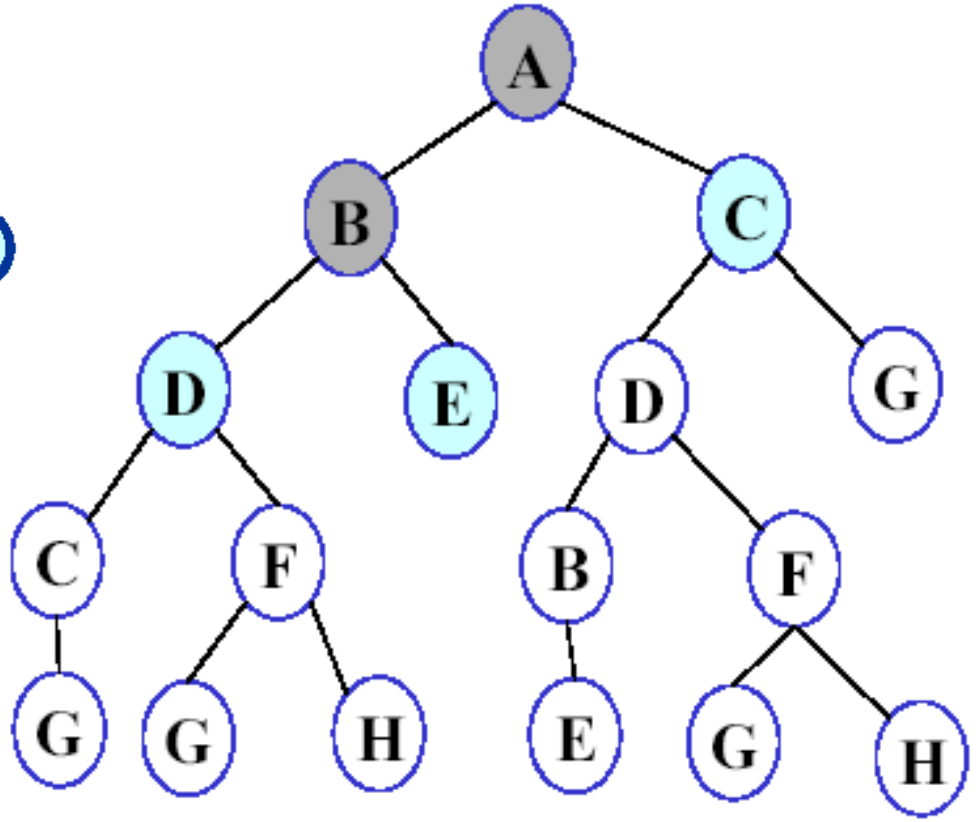
FRINGE: A



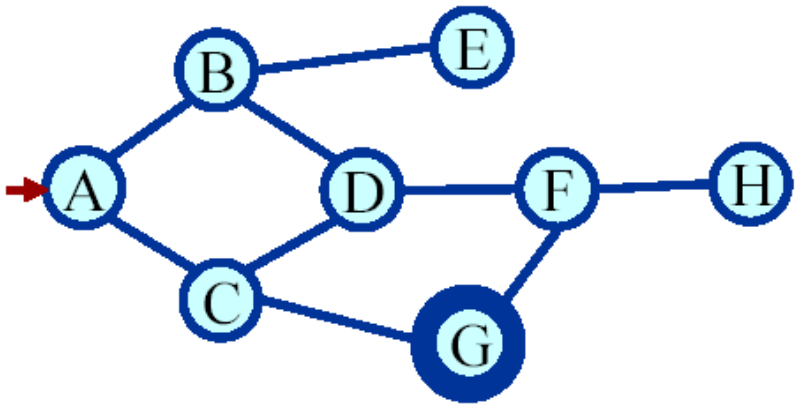


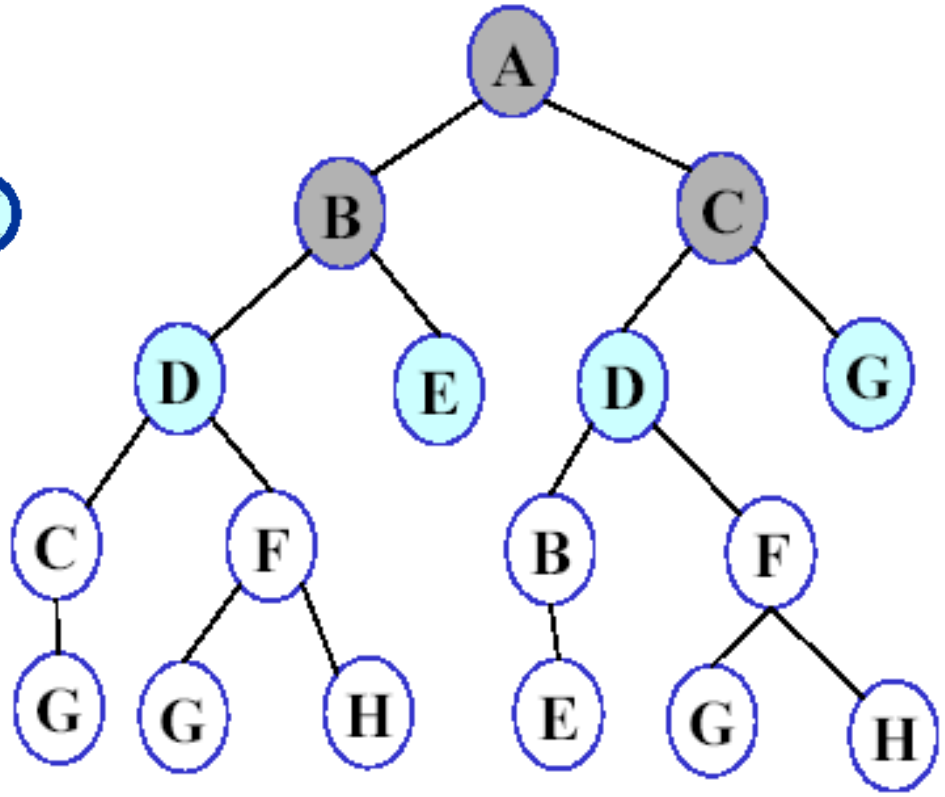
FRINGE: B C



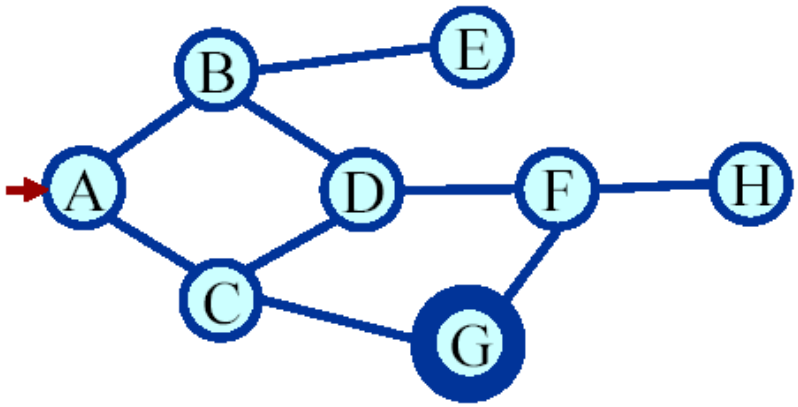


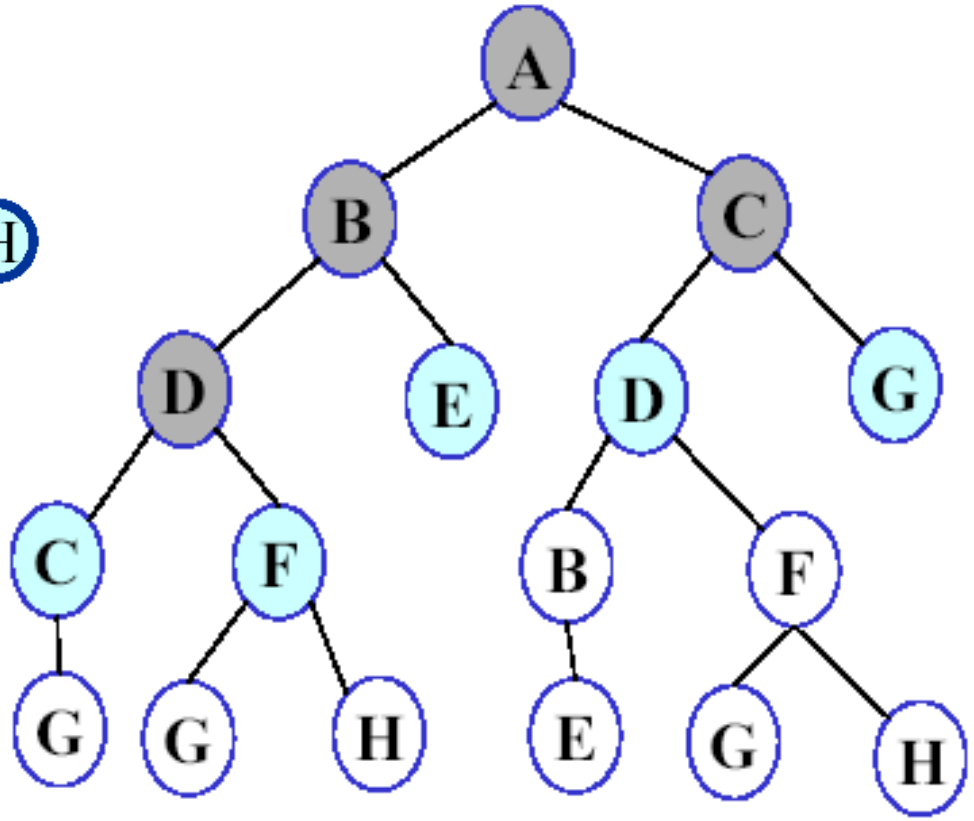
FRINGE: C D E



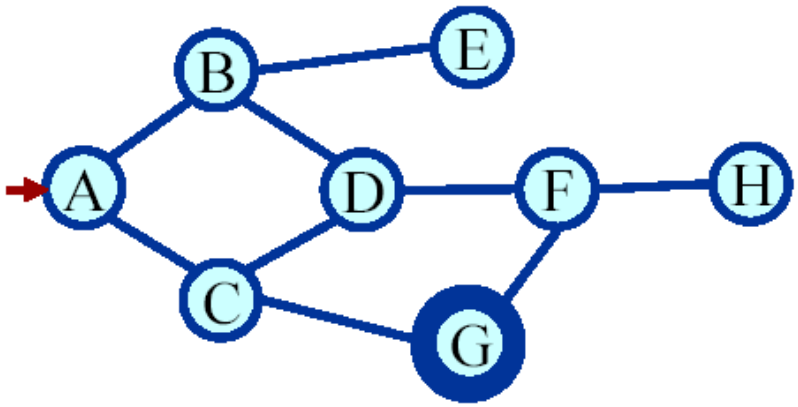


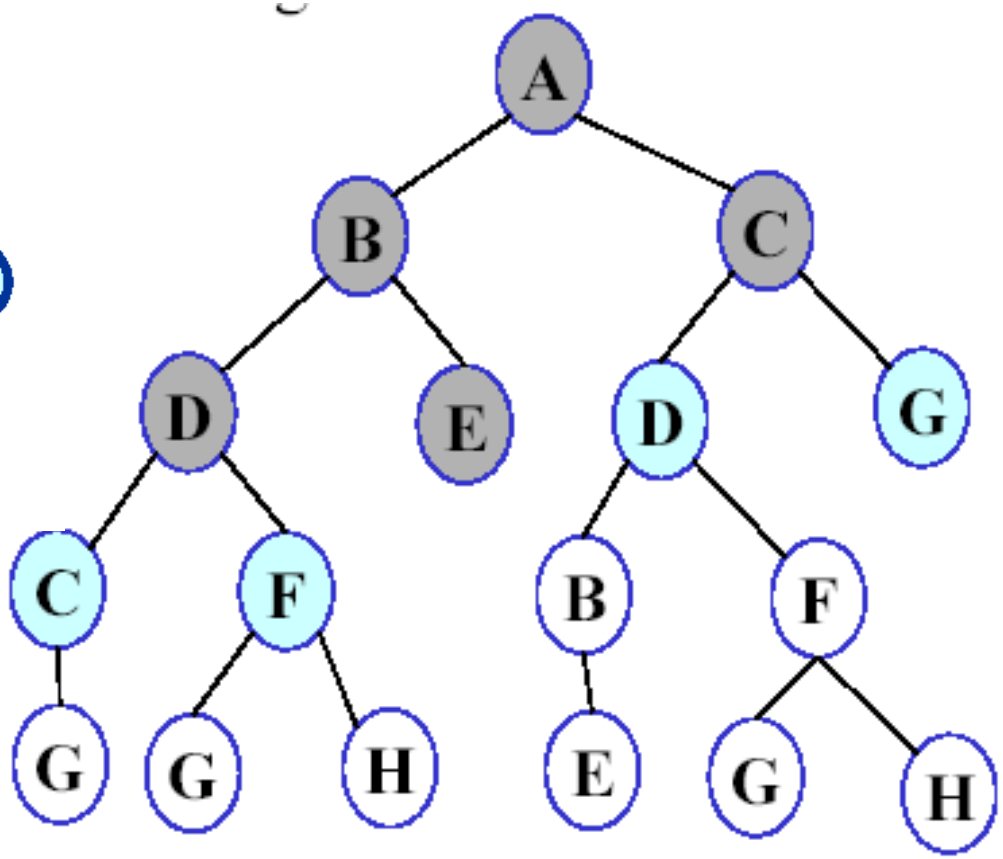
FRINGE: D E D G



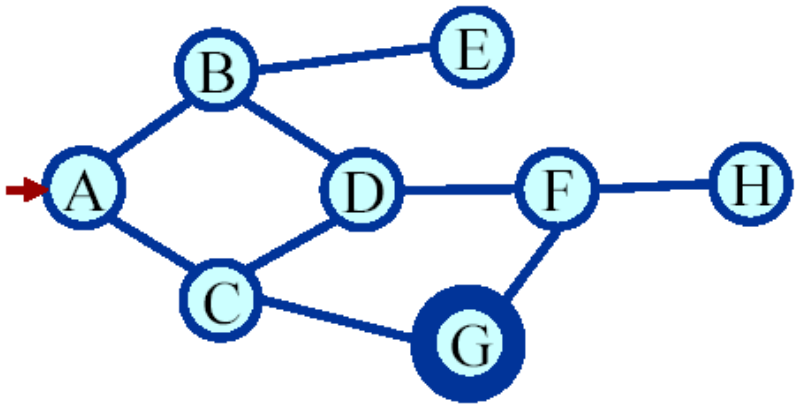


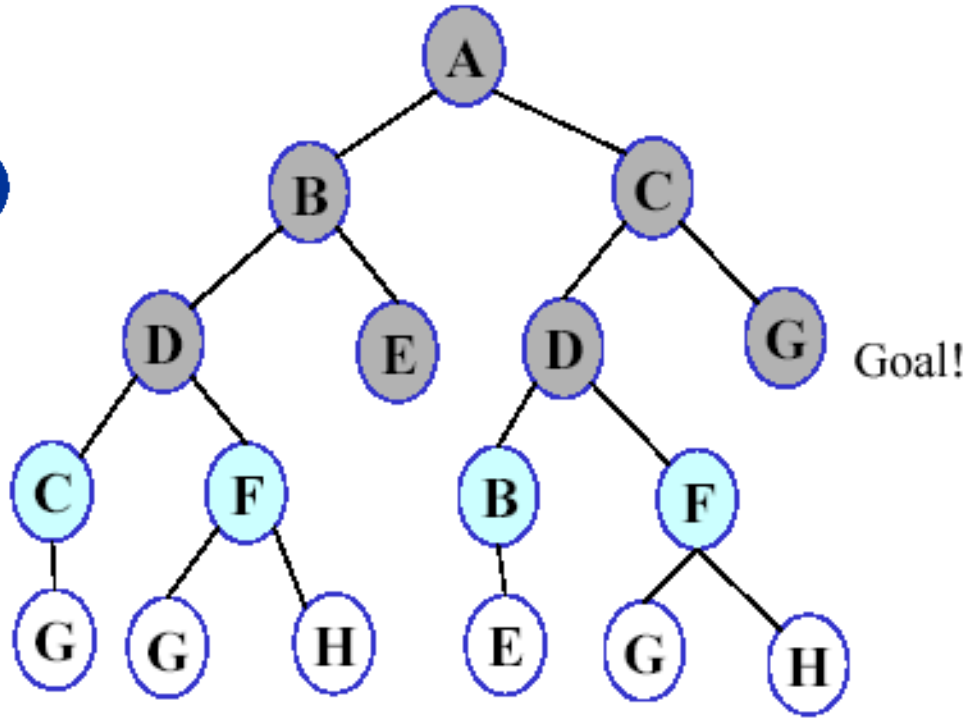
FRINGE: E D G C F





FRINGE: D G C F





FRINGE: G C F B F

Properties of Breadth-First Search

- We assume that every non-leaf node has b children.
- Suppose that d is the depth of the shallowest goal node,
- m is the depth of the node found first.

Properties of Breadth-First Search

- Breadth first search is:
  - Complete.
  - The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, finds a solution with the shortest path length.
  - The algorithm has exponential time and space complexity
  - No. of nodes in a a complete search tree

    $1 + b + b2 + \ldots + bd = (b(d+1) - 1)/(b-1)$ nodes
  - $O(b^d)$

Properties of Breadth-First Search

- Consider a complete search tree of depth 15,
- every node at depths 0 to14 has 10 children and every node at depth 15 is a leaf node.
- The complete search tree in this case will have $O(10^{15})$ nodes.
- If BFS expands 10000 nodes per second and each node uses 100 bytes of storage,

  BFS will take 3500 years to run in the worst case,

  and will use 11100 terabytes of memory.
- Applicable for only limited and small search space.

Advantages of Breadth First Search

- Advantages
  - Finds the path of minimal length to the goal.
- Disadvantages
  - Requires the generation and storage of a tree whose size is exponential to the depth of the shallowest goal node

Uniform-cost search

- The algorithm expands nodes in the order of their cost from the source.
- In uniform cost search the newly generated nodes are put in OPEN according to their path costs.
- This ensures that when a node is selected for expansion it is a node with the cheapest cost among the nodes in OPEN.

Some properties

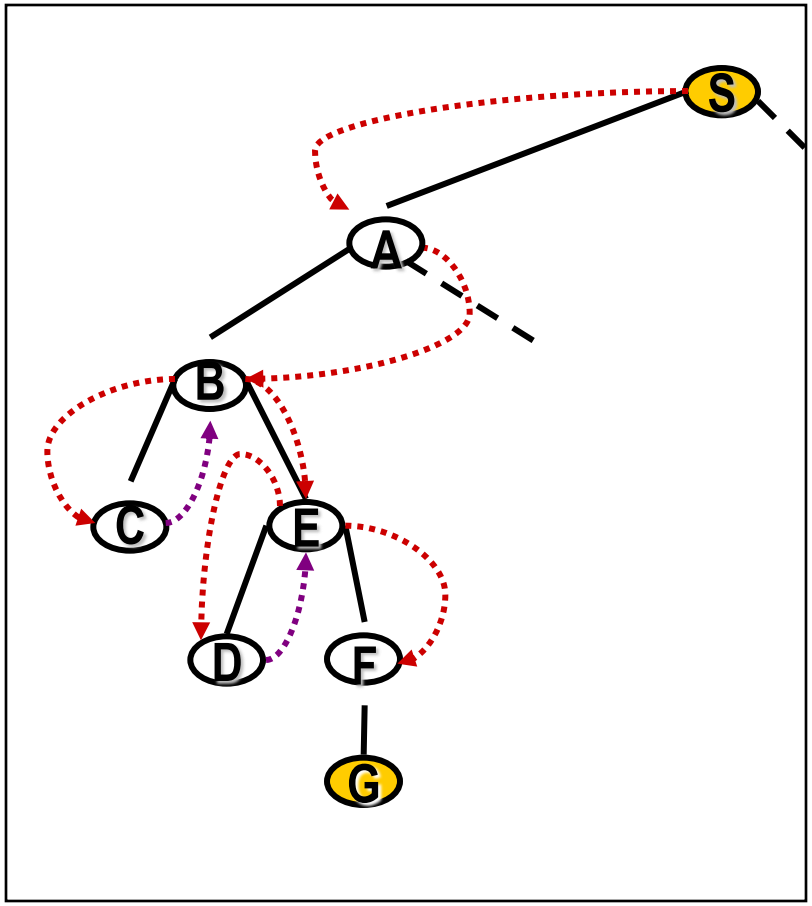
- Complete
- Optimal/Admissible
- Exponential time and space complexity, $O(b^d)$

Depth first Search

Depth First Search
Let *fringe* be a list containing the initial state Loop if *fringe* is empty return failure Node ← remove-first (*fringe*) if Node is a goal then return the path from initial state to Node else generate all successors of Node, and merge the newly generated nodes into *fringe* add generated nodes to the front of *fringe* End Loop

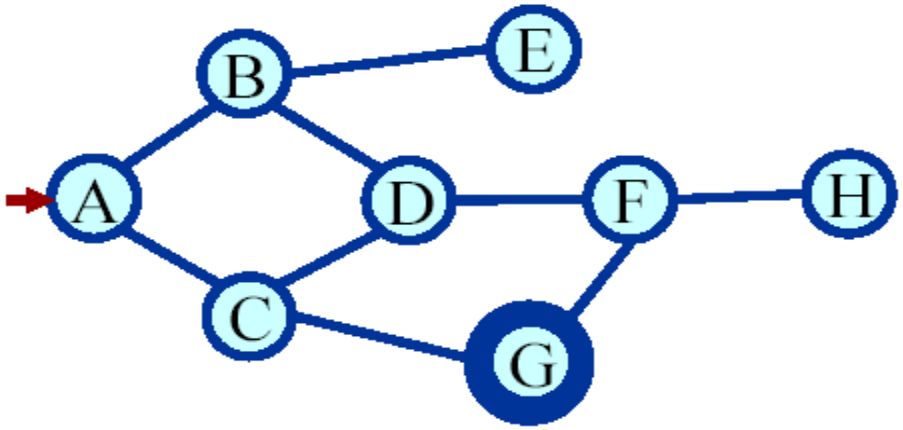
- expands the deepest node first.
- OPEN list follows a LIFO order

Depth-first search



- Select a child
  - convention: left-to-right
- Repeatedly go to next child, as long as possible.
- Return to left-over alternatives (higher-up) only when needed.

DFS illustrated



FRINGE: A

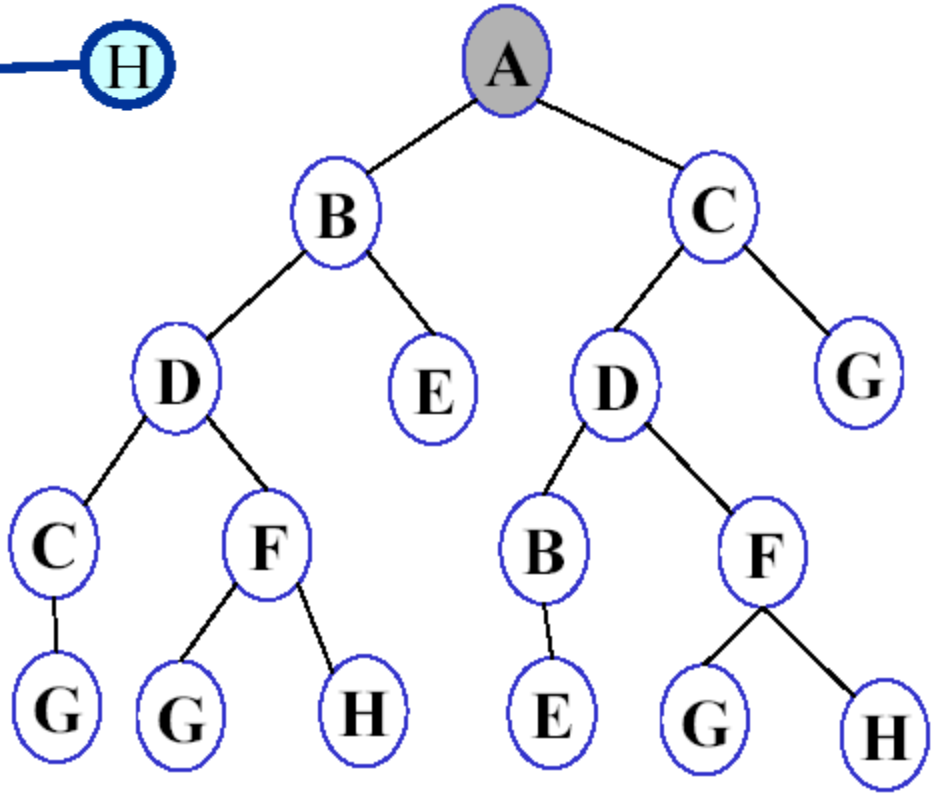
FRINGE: B C

FRINGE: D E C

FRINGE: C F E C

FRINGE: G F E C

FRINGE: G F E C



Properties of Depth First Search

- The algorithm takes exponential time.
- In the worst case the algorithm will take time $O(b^d)$.
- space taken is linear in the depth of the search tree, O(bN).
- If the search tree has infinite depth (due to cycle), the algorithm may not terminate.
- not complete.

Depth Limited Search

- A variation of Depth First Search having a depth bound.
- Nodes are only expanded if they have depth less than the bound.

```
Depth limited search (limit)
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
       else if depth of Node = limit return cutoff
       else add generated nodes to the front of fringe
End Loop
```

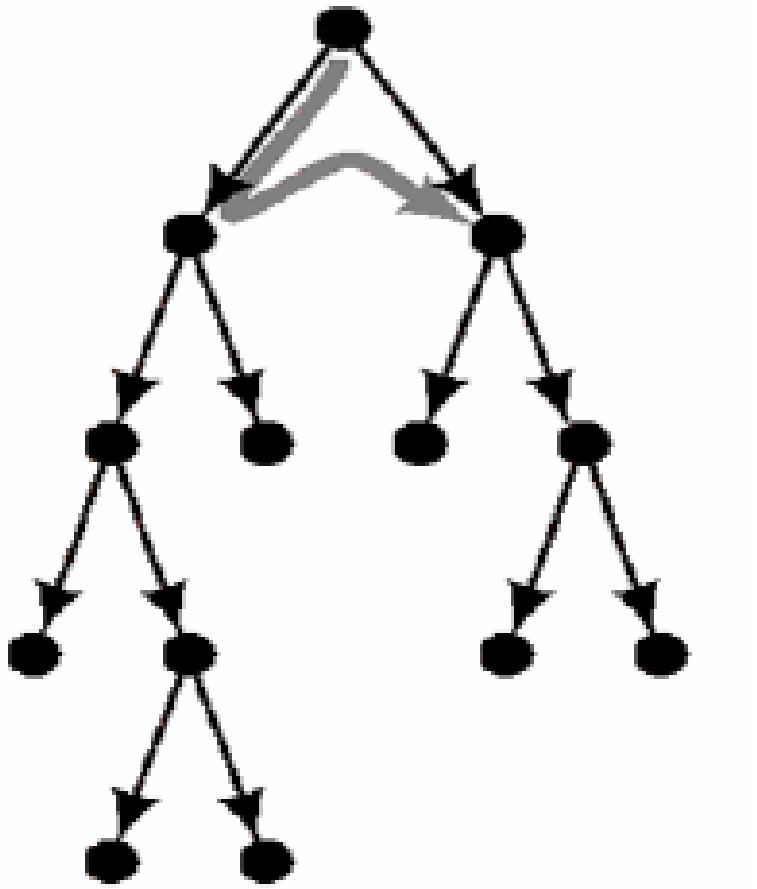
Depth-First Iterative Deepening (DFID)

- First do DFS to depth 0
- then, if no solution found, do DFS to depth 1,
- Carry on above step with depth incremented by 1

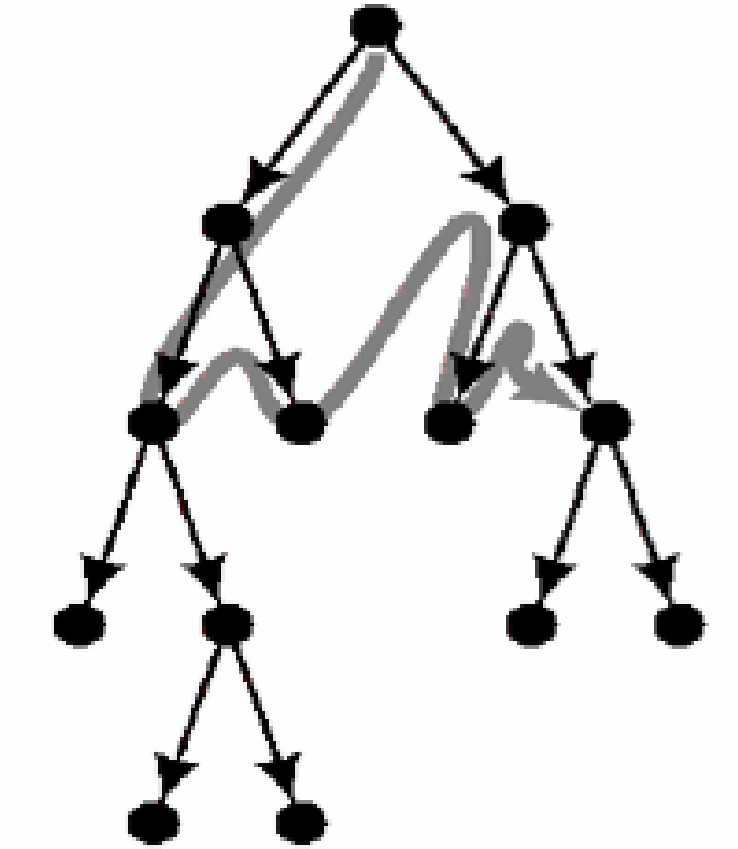
DFID
until solution found do *DFS with depth cutoff c* $c = c+1$

Advantage

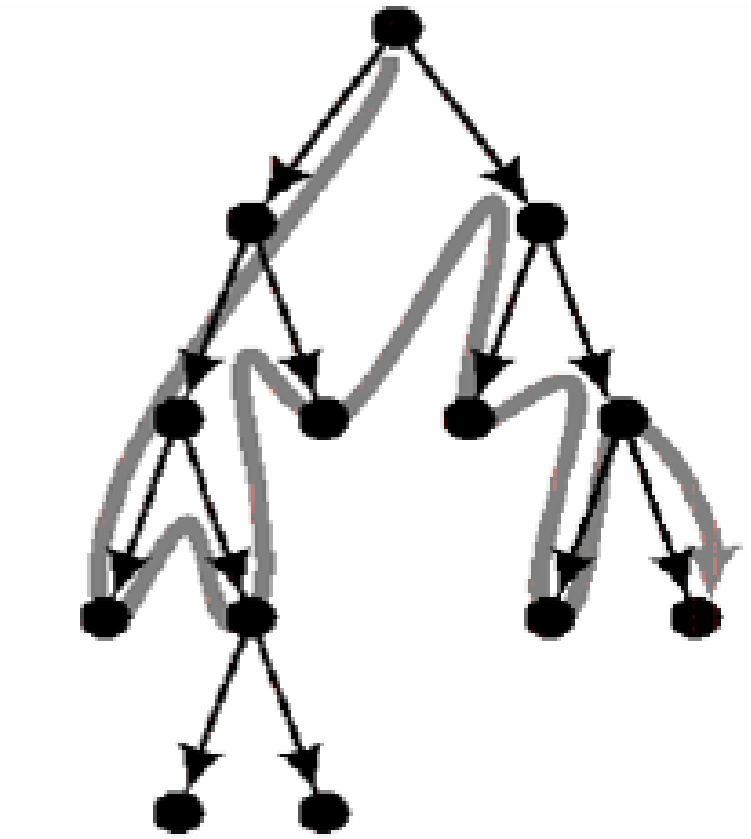
- Linear memory requirements of depth-first search
- Guarantee for goal node of minimal depth



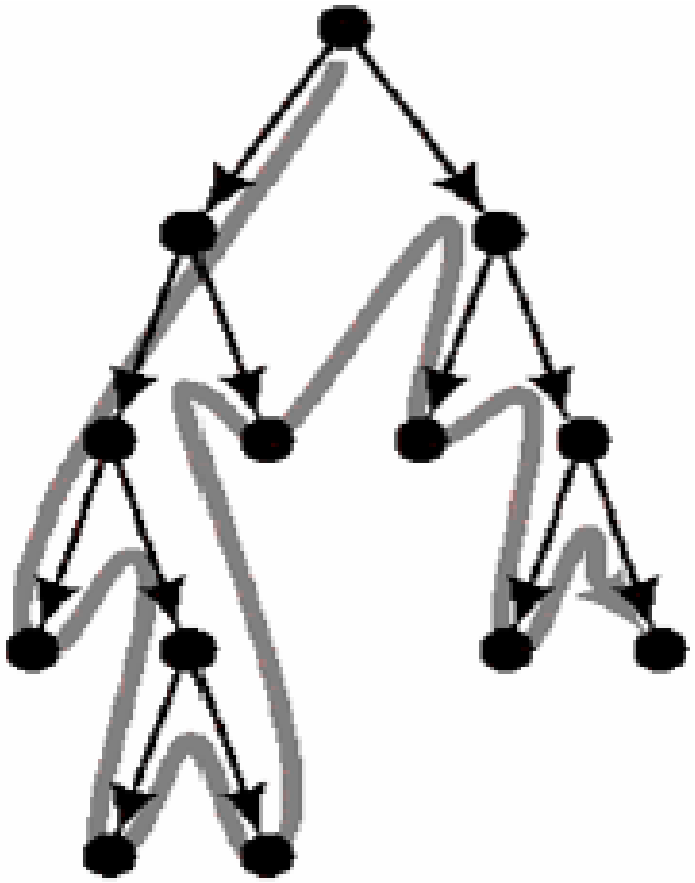
Depth bound = 1



Depth bound = 2



Depth bound = 3



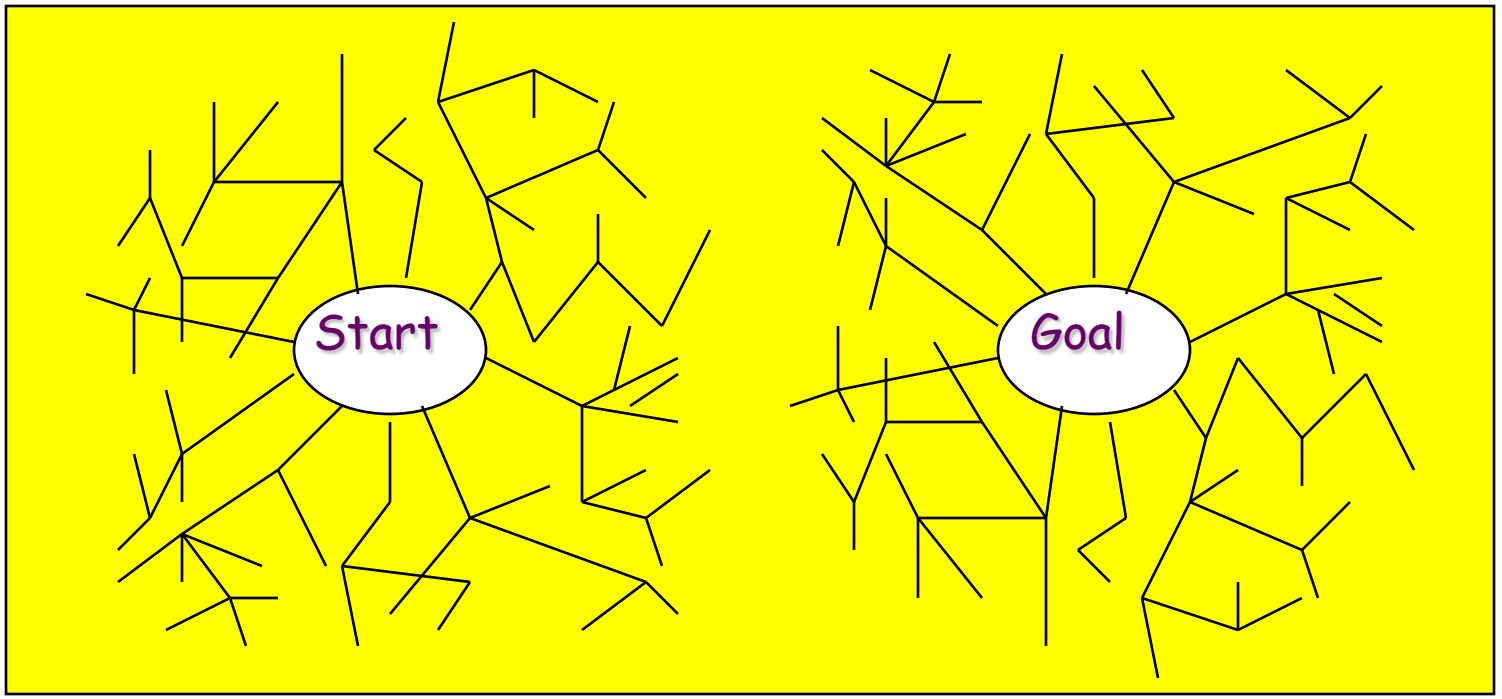
Depth bound = 4

Properties

- Complete
- Optimal/Admissible if all operators have the same cost. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- Time complexity is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times,
  - the worst case time complexity is still exponential, $O(b^d)$
- Depth First Iterative Deepening combines the advantage of BFS (i.e., completeness) with the advantages of DFS (i.e., limited space and finds longer paths more quickly)
- This algorithm is generally preferred for **large state spaces** where the **solution depth is unknown.**

Bi-directional search

- IF you are able to EXPLICITLY describe the GOAL state, AND
- you have BOTH rules for FORWARD reasoning AND BACKWARD reasoning:



Bi-directional search

- Algorithm: Bidirectional search involves alternate searching from the start state toward the goal and from the goal state toward the start. The algorithm stops when they intersect.

Comparing Search Strategies

	Breadth first	Depth first	Iterative deepening	Bidirectional (if applicable)
Time Space Optimum? Complete?				

- d = depth of the tree
- b = (average) branching factor of the tree
- m = depth of the shallowest solution

Search Graphs

- How to avoid repeated state expansions
  - state is already in OPEN
  - But what if the state was in OPEN earlier but has been removed and expanded
  - The newly generated node is checked with the nodes in CLOSED too

```
Graph search algorithm
Let fringe be a list containing the initial state
Let closed be initially empty
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else put Node in closed
                generate all successors of Node S
                for all nodes m in S
                    if m is not in fringe or closed
                        merge m into fringe
End Loop
```

- But this algorithm is quite expensive.
- we can adopt some less computationally intense strategies.
- Such strategies do not stop duplicate states from being generated, but are able to reduce many of such cases.
  - E.g. not return to the state the algorithm just came from (for 15-puzzle like problem).