Lecture 5
Artificial Intelligence

Khola Naseem
khola.naseem@uet.edu.pk

Problem Solving by Searching

Measuring problem-solving performance

- We can evaluate an algorithm's performance in four ways:
  - **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
  - **Optimality:** Does the strategy find the optimal solution?
  - **Time complexity:** How long does it take to find a solution?
  - **Space complexity:** How much memory is needed to perform the search?

Measuring problem-solving performance

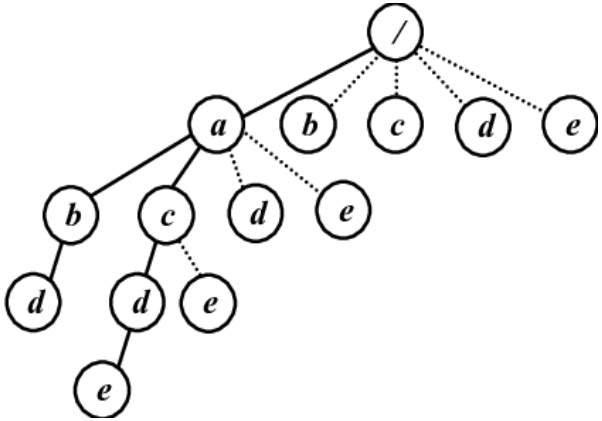
- In AI, complexity is expressed in
  - b, branching factor, maximum number of successors of any node
  - d, the depth of the shallowest goal node. (depth of the least-cost solution)
  - m, the maximum length of any path in the state space
- Time and Space is measured in
  - number of nodes generated during the search
  - maximum number of nodes stored in memory

Search strategies

- Uninformed search
  - no information about the number of steps
  - or the path cost from the current state to the goal
  - search the state space blindly , Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.
  - E.g Breadth-first search, Depth-first search, Uniform cost search, Bidirectional Search
- Informed search, or heuristic search
  - a cleverer strategy that searches toward the goal,
  - based on the information from the current state so far
  - E.g. A*, Heuristic DFS, Best first search

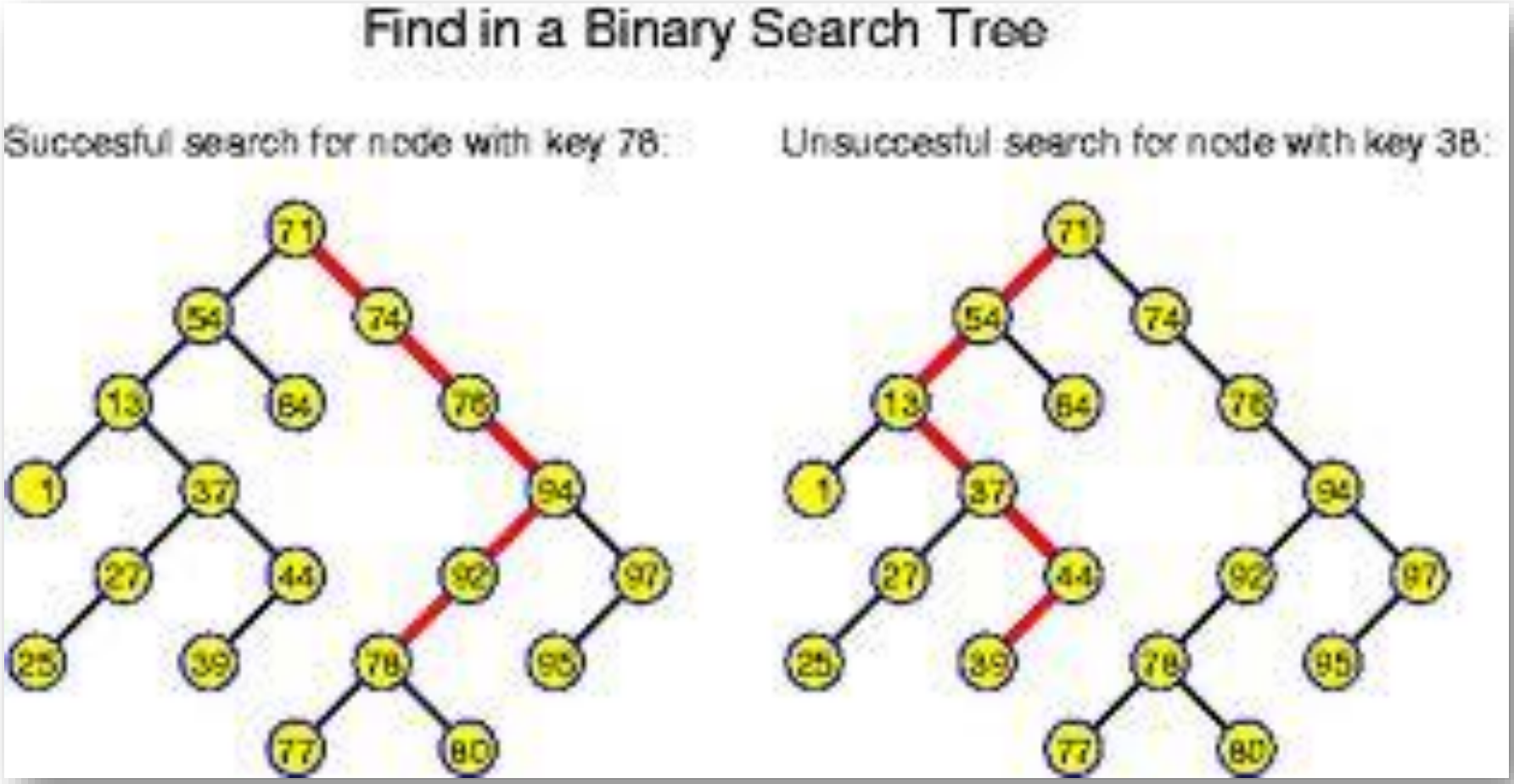
Linking Search to Trees and Graphs

- You can begin to visualize the concept of a graph (or Tree)
- Searching along different paths of the graph until you reach the solution
- The nodes can be the states
- The whole graph can be the state space
- The links can be the actions……



Credit: Khola Naseem

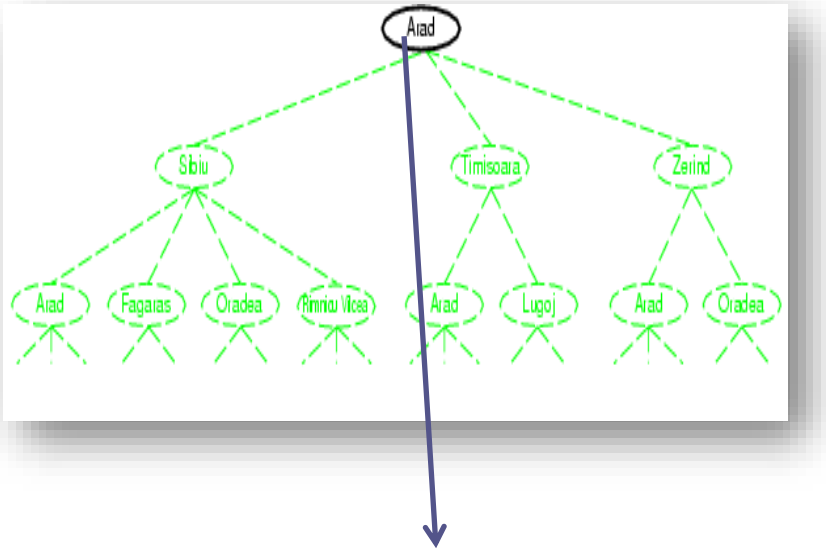
Linking Search to Trees and Graphs

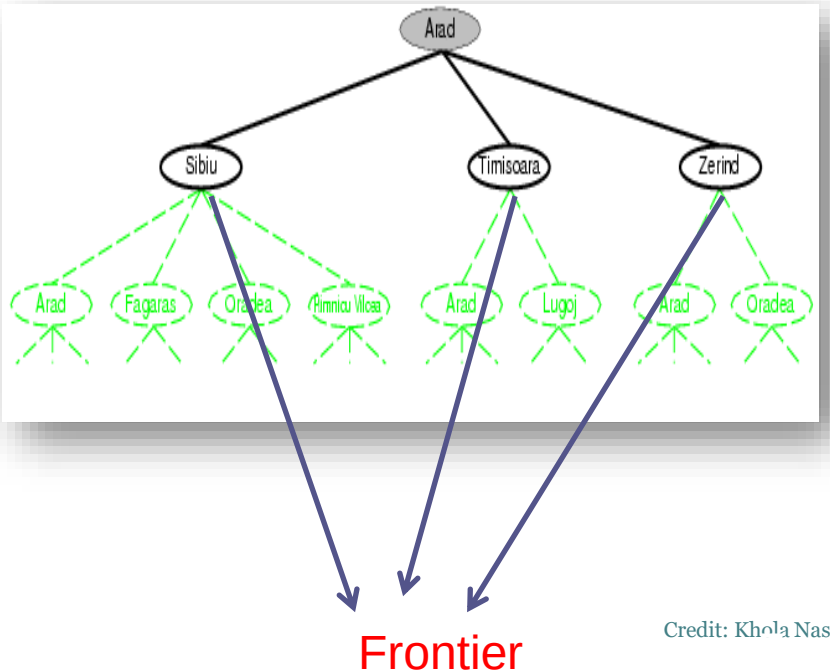


Credit: Khola Naseem

Trees and Graphs

- Fringe: The collection of nodes that have been generated but not yet expanded
- Each element of the fringe is a leaf node, with (currently) no successors in the tree
- The search strategy defines which element to choose from the fringe





Credit: Khola Naseem

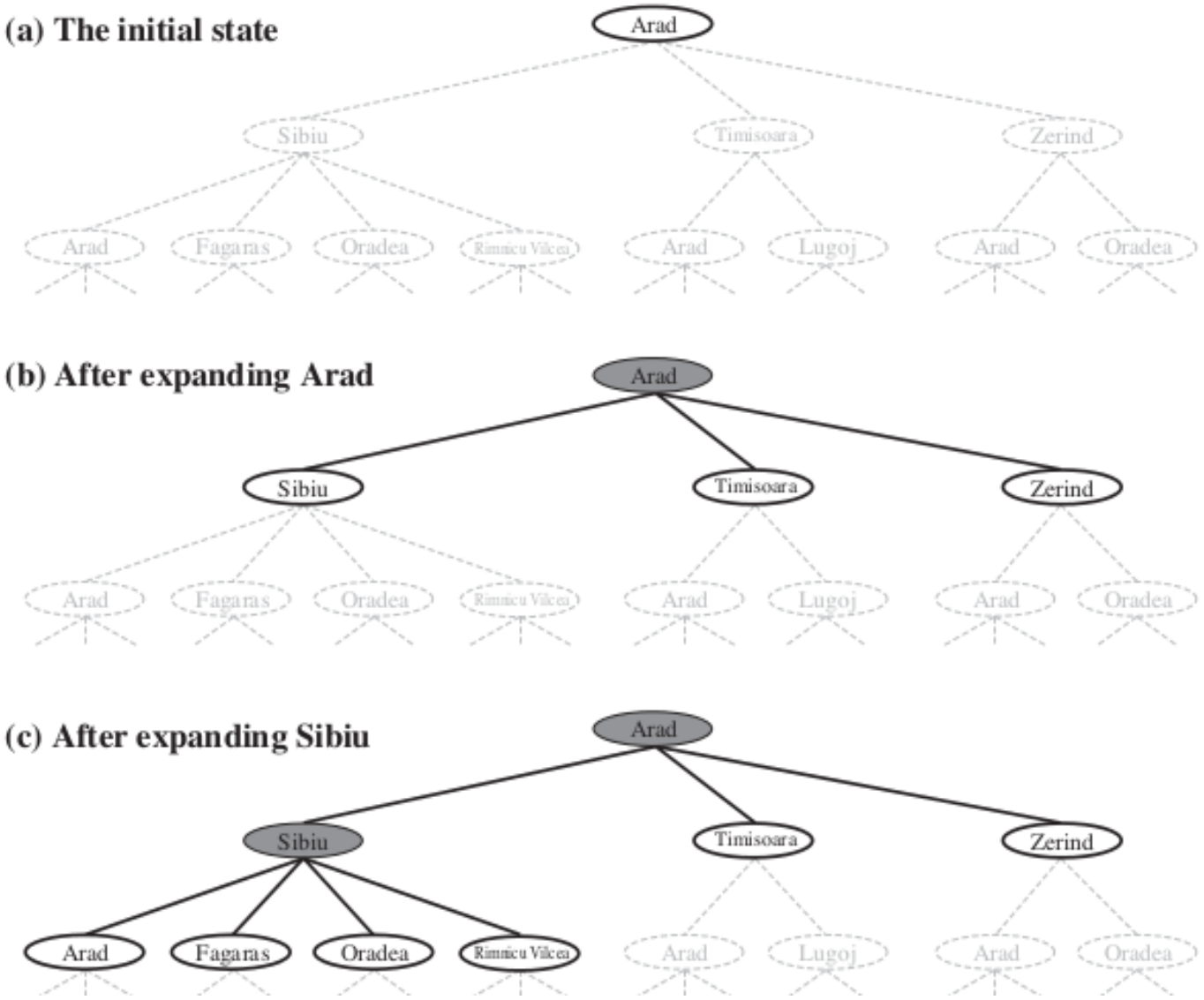
Linking Search to Trees and Graphs

```
function TREE-SEARCH( problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH( problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Tree search Example



Credit: Khola Naseem

Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed.

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

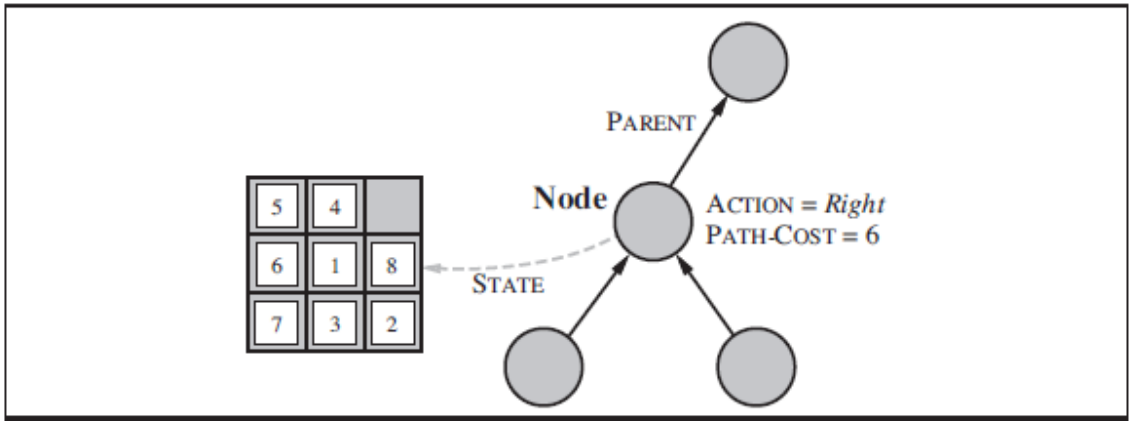


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Credit: Khola Naseem

Infrastructure for search algorithms

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action),
        PARENT = parent, ACTION = action,
        PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Credit: Khola Naseem

Trees and Graphs

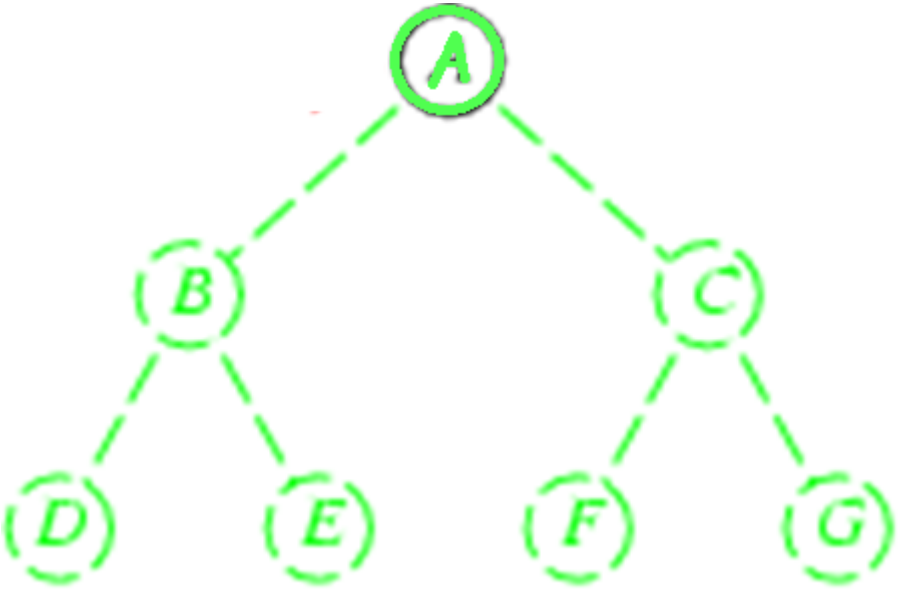
- The fringe is implemented as a queue
  - MAKE_QUEUE(element,…): makes a queue with the given elements
  - EMPTY?(queue): checks whether queue is empty
  - FIRST(queue): returns 1st element of queue
  - REMOVE_FIRST(queue): returns FIRST(queue) and removes it from queue

    known as pop(queue)
  - INSERT(element, queue): add element to queue known as

    push(element,queue)

Breadth-first search

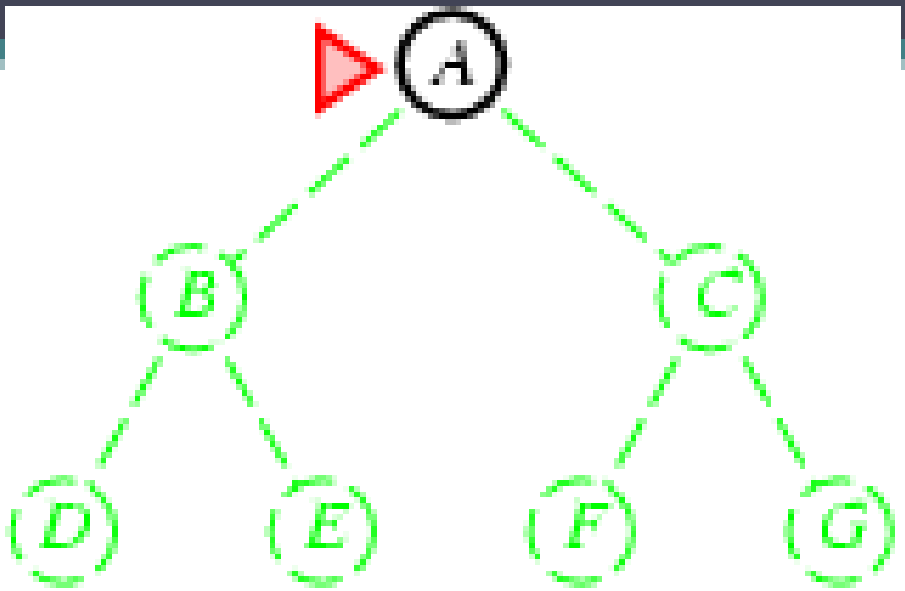
- Implementation:
  - fringe is a FIFO queue, i.e., new successors go at end
  - Goal Test is applied to each node when it is generated rather than when it is selected for expansion



Frontier: [A]

Chose Lead Node: None
Goal state: M
Explored:[]
Expand: None

Credit: Khola Naseem

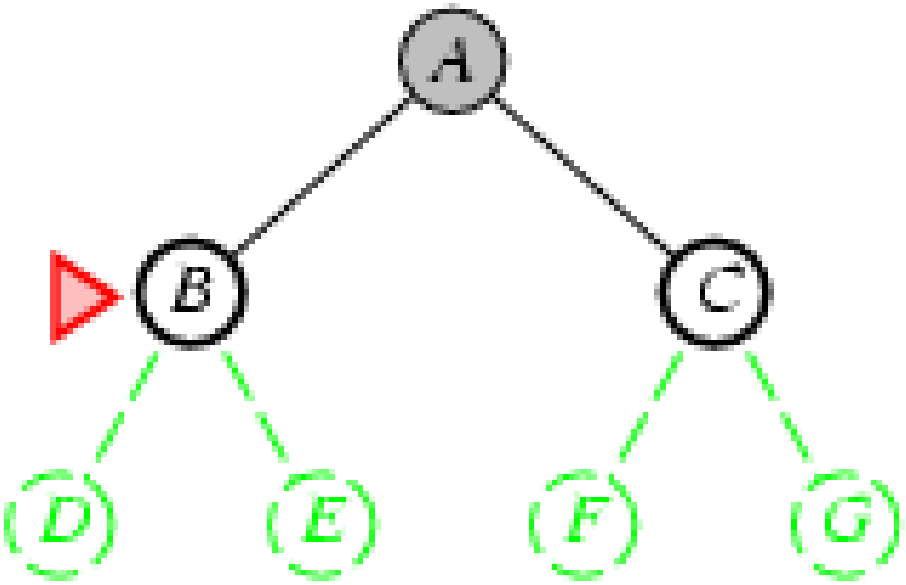


Credit: Khola Naseem

Frontier: [A]

Chose Lead Node: A
Frontier: []
Is '**A**' a goal state? NO

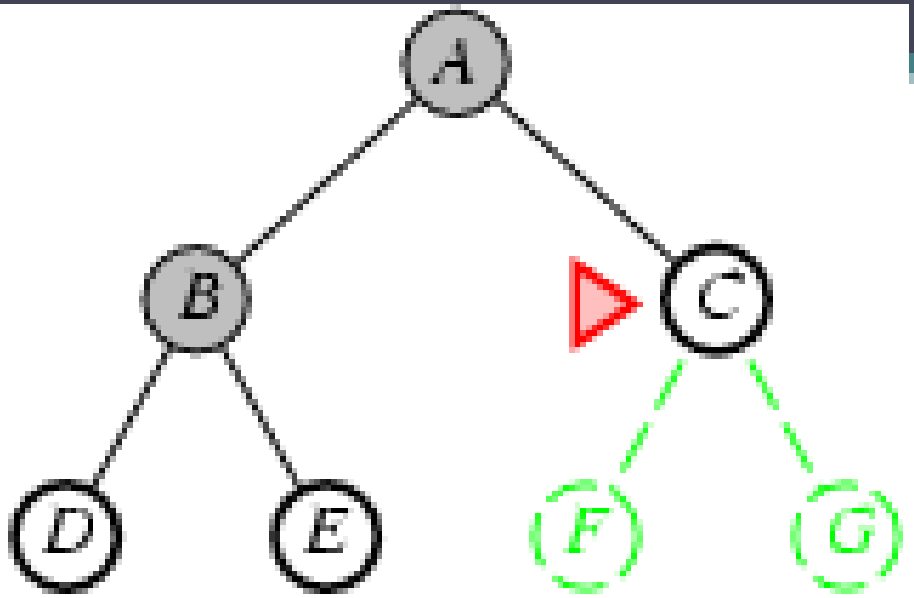
Explored: [A]
Expand: A
Frontier: [B,C]



Frontier: [B,C]

Chose Lead Node: B
Frontier: [C]
Is '**B**' a goal state? NO

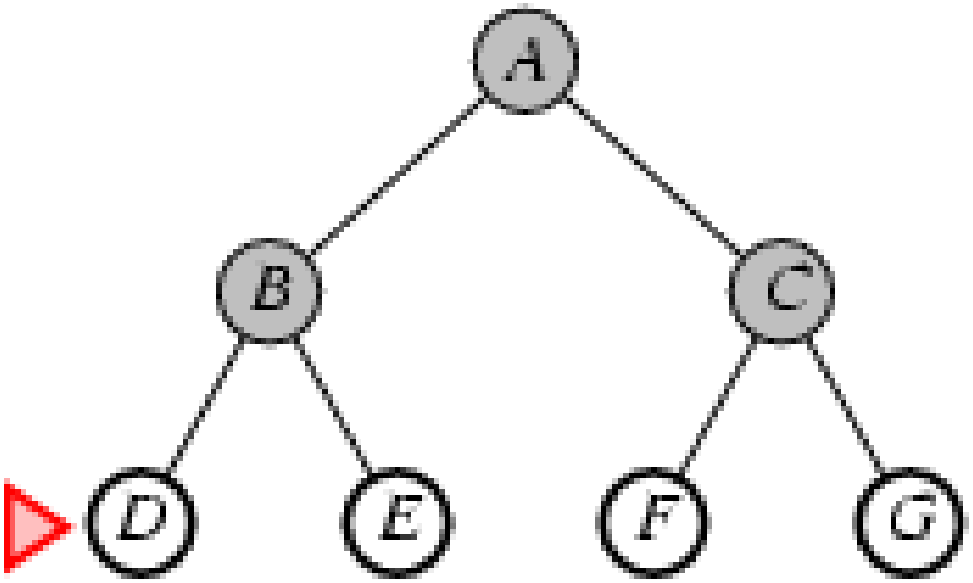
Explored: [A,B]
Expand: B
Frontier: [C,D,E]



Credit: Khola Naseem

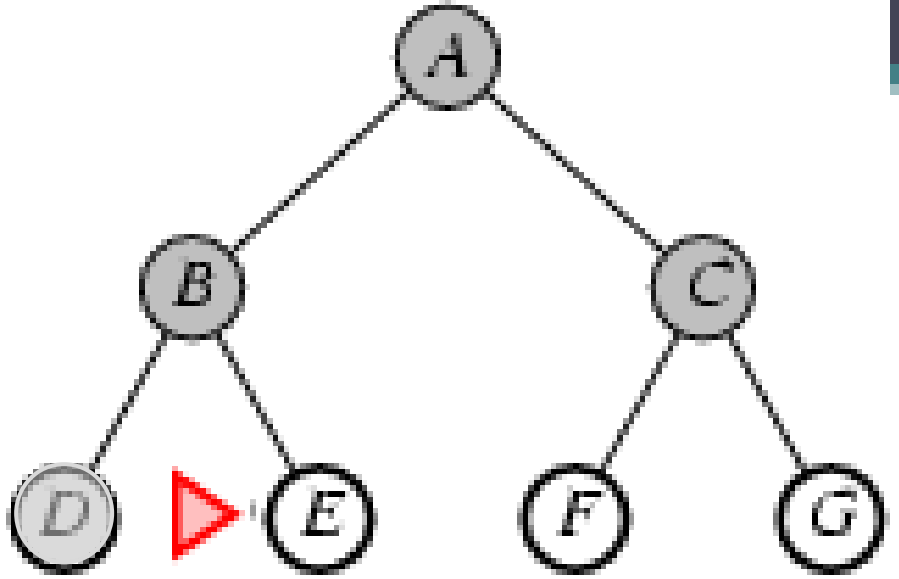
Frontier: [C,D,E]
Chose Leaf Node: C
Frontier: [D,E]
Is 'C' a goal state? NO

Explored: [A,B,C]
Expand: C
Frontier: [D,E,F,G]



Frontier: [D,E,F,G]
Chose Lead Node: D
Frontier: [E,F,G]
Is '**D**' a goal state? NO

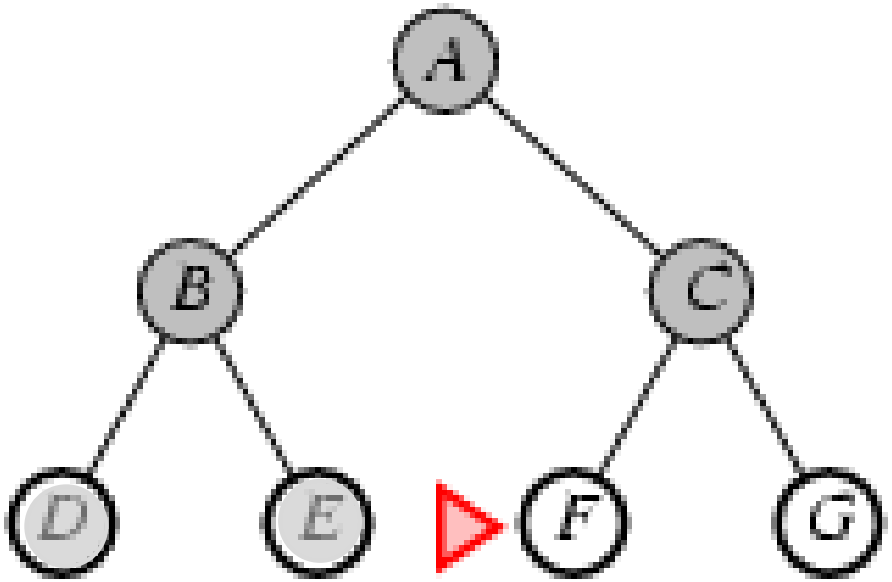
Explored: [A,B,C,D]
Expand: D
Frontier: [E,F,G]



Credit: Khola Naseem

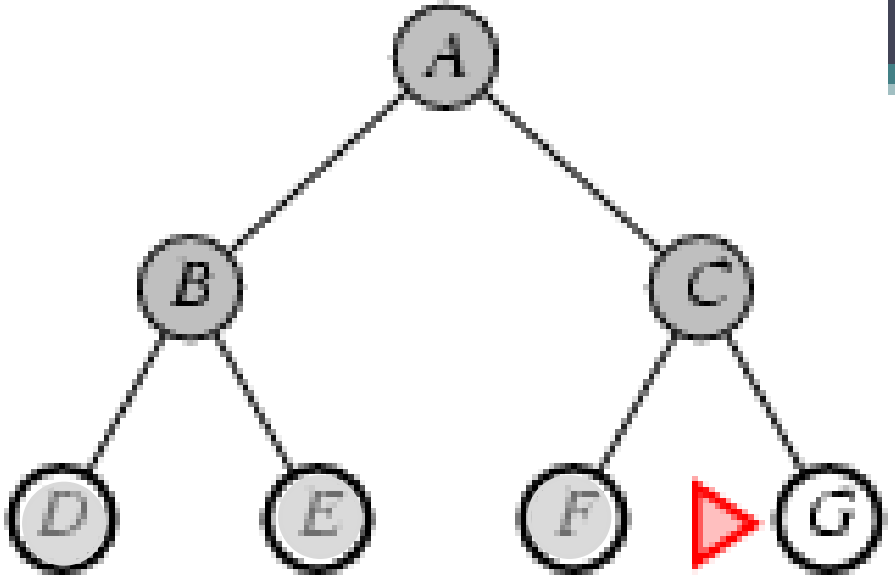
Frontier: [E,F,G]
Chose Lead Node: E
Frontier: [F,G]
Is '**E**' a goal state? NO

Explored: [A,B,C,D,**E**]
Expand: E
Frontier: [F,G]



Frontier: [F,G]
Chose Lead Node: F
Frontier: [G]
Is '**F**' a goal state? NO

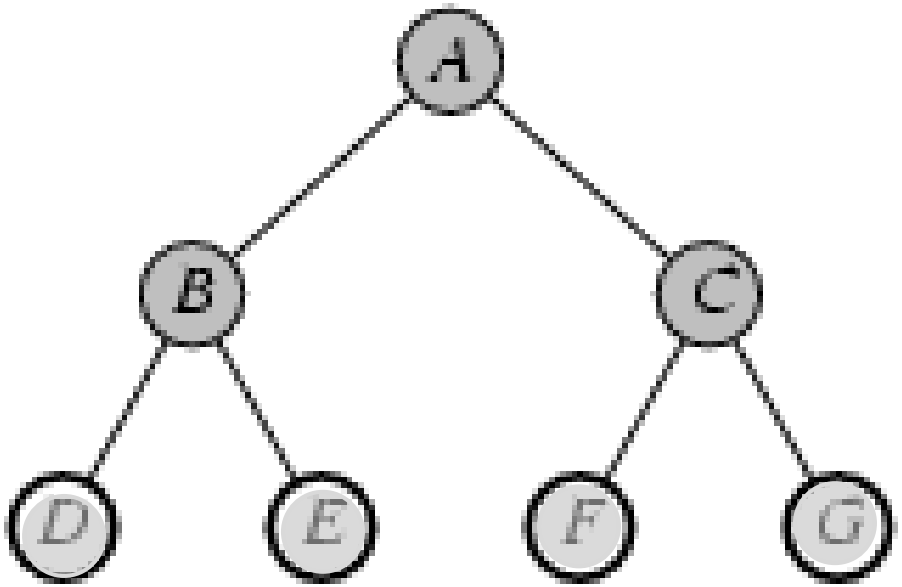
Explored: [A,B,C,D,E,F]
Expand: F
Frontier: [G]



Credit: Khola Naseem

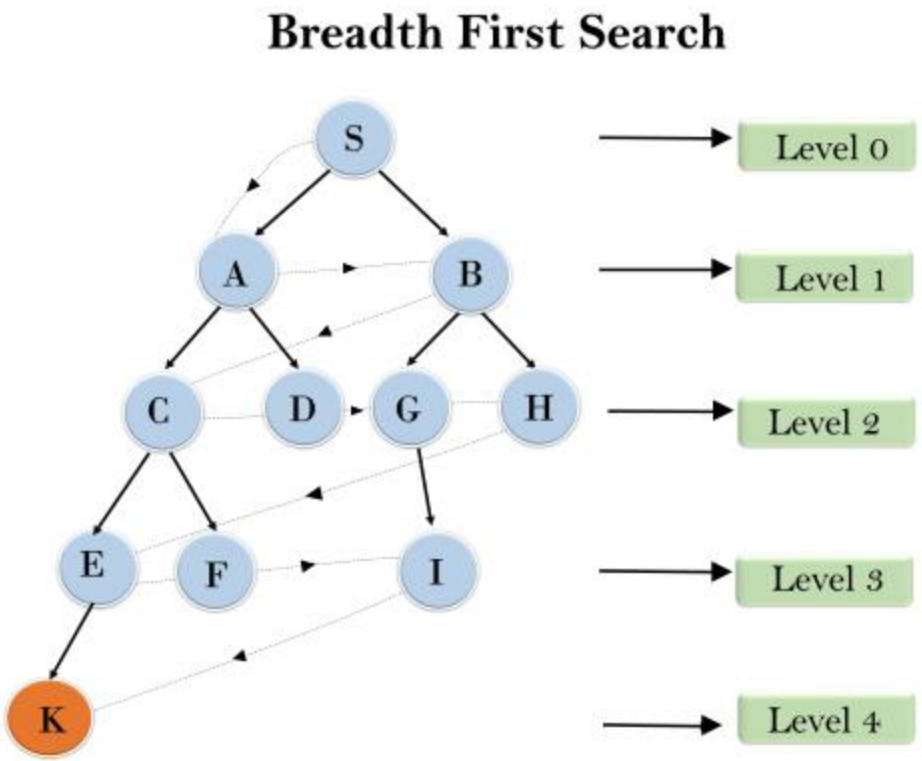
Frontier: [G]
Chose Leaf Node: G
Frontier: []
Is '**G**' a goal state? NO

Explored: [A,B,C,D,E,F,**G**]
Expand: G
Frontier: []



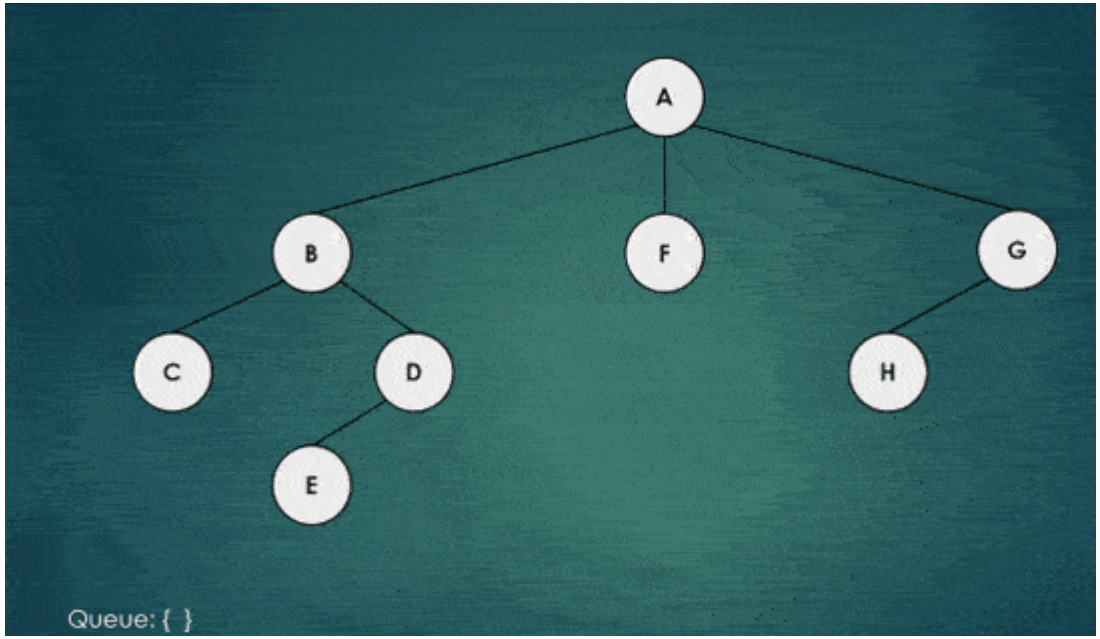
Solution not Found.

BFS



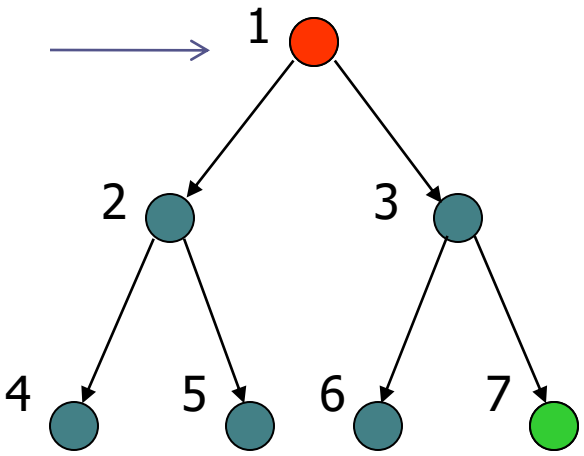
Credit: Khola Naseem

BFS

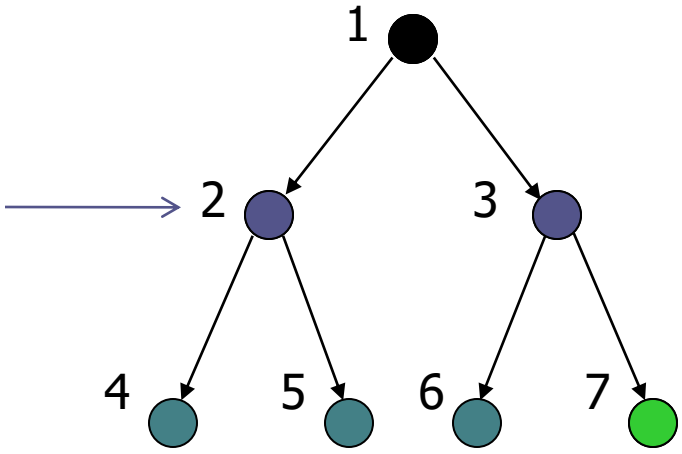


Credit: Khola Naseem

BFS



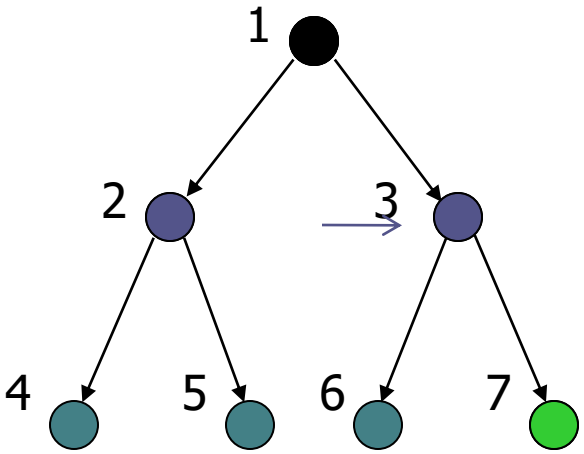
FRINGE = (1)



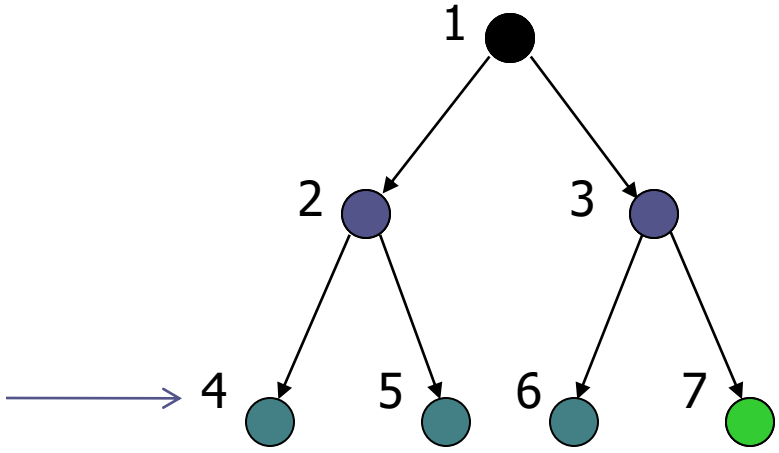
FRINGE = (2,3)

Credit: Khola Naseem

BFS



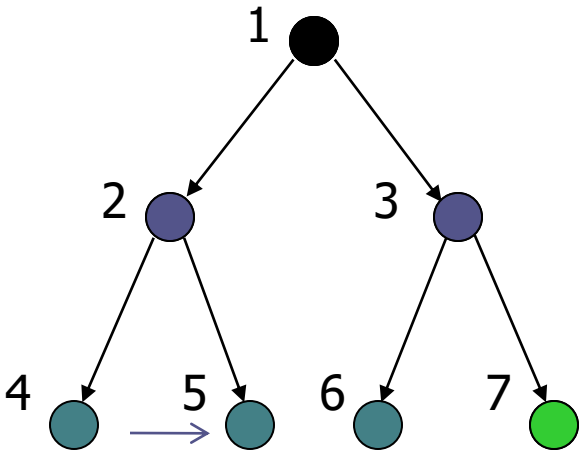
FRINGE = (3,4,5)



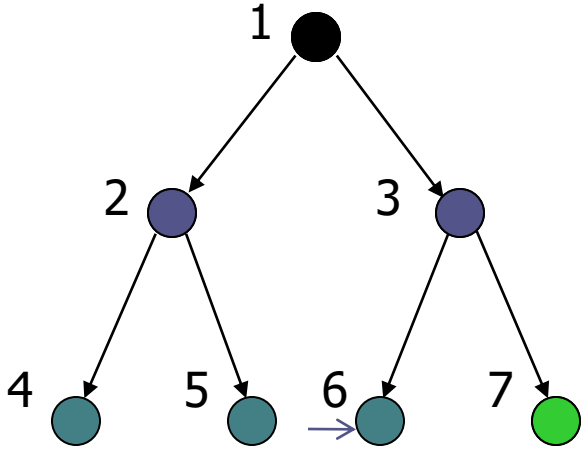
FRINGE = (4,5,6,7)

Credit: Khola Naseem

BFS



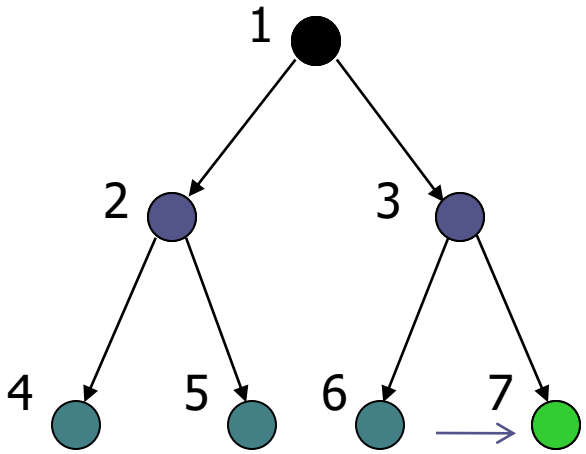
FRINGE = (5,6,7)



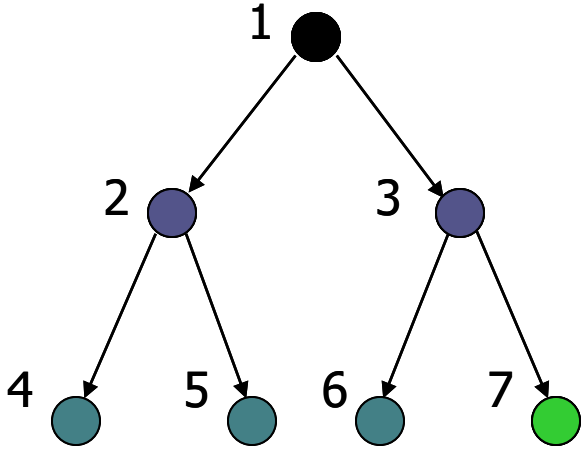
FRINGE = (6,7)

Credit: Khola Naseem

BFS



FRINGE = (7)



FRINGE = ()

Credit: Khola Naseem

BFS:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

BFS:

- Complete? Does it always find a solution if one exists? Yes
- Time? b^0+b^1+b^2+b^3+… +b^d = O(b^d) i.e. exponential in d
- Space? O(b^d) (keeps every node in memory)
  - For any kind of graph/tree search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity.
- Optimal? Does it always find a least-cost solution?
  - Yes (if cost = 1 per step. i.e. identical step costs)
- Space is a big problem: exponential expansion

BFS Problem:

- b how many children a node can have and d is the level to the nearest goal
- If b=2 and d=50 how many nodes are processed --> 2^50
- Space:
  - 2^50 means 1 peta nodes needs to be process, if 1 node takes 1 byte the 2^50 bytes will be required to process this tree so bfs is not good in space
- Time if 1 GB=2^30 nodes process in 1 sec how many days are required to process 2^50 nodes
- Question?

BFS Problem:

- Time if 1 GB=2^30 nodes process in 1 sec how many days are required to process 2^50 nodes
- Question?
- 2^50=2^50/2^30=2^20(Sec)
- 1 min=2^6 sec so 2^20/2^6=2^14 mints
- 1 hour=2^6 mints so 2^14/2^6=2^8 hours
- 1 day= 2^5 hours so 2^8/2^5=2^3= 8 days
- So 8 days are required to process 2^50 nodes

BFS Conclusion:

- BFS is an elegant solution.
- It is complete and optimal.

But

- when it is run on practical nature problem having large state space or depth is vast.
- It perform badly in space and time

Uniform-cost Search

- When all step costs are equal, BFS is optimal
  - It always expands shallowest unexpanded node
- With a simple extension: We find algorithm that is optimal with any step-cost function
- UCS, expands the node n with lowest path cost $g(n)$
- How: use priority queue ordered by $g(n)$

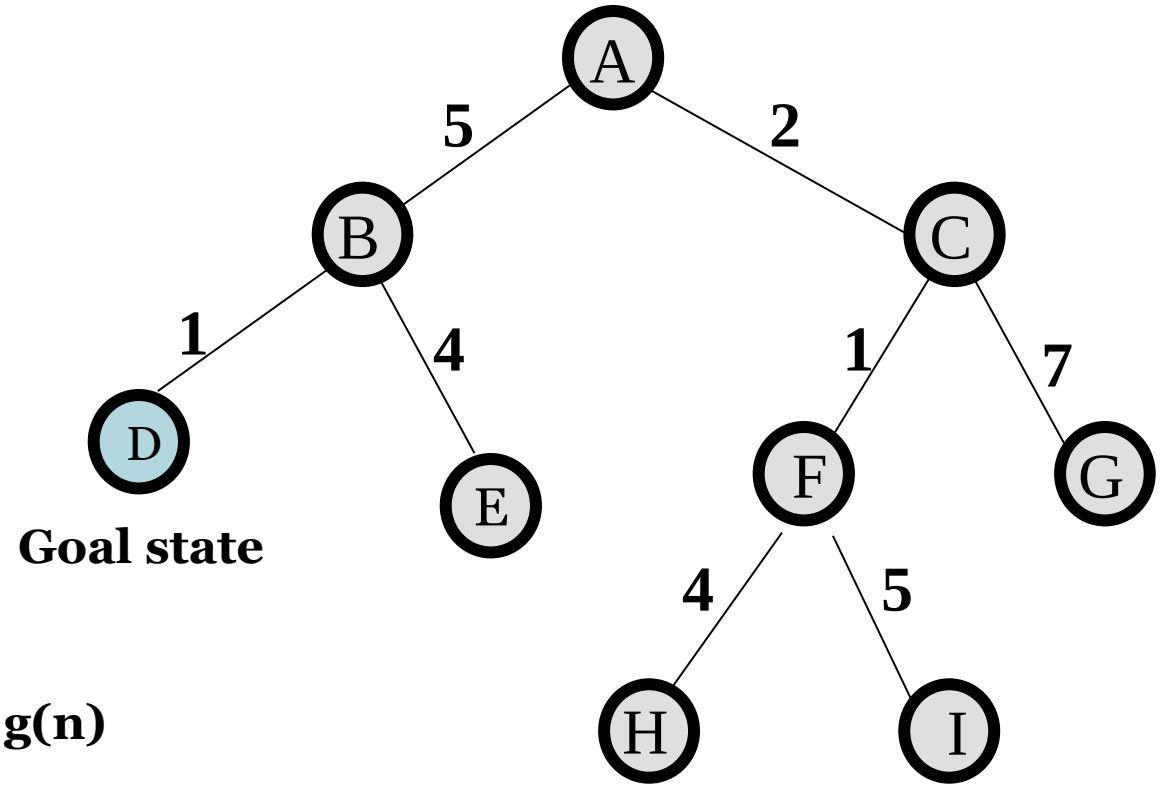
Credit: Khola Naseem

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Credit: Khola Naseem

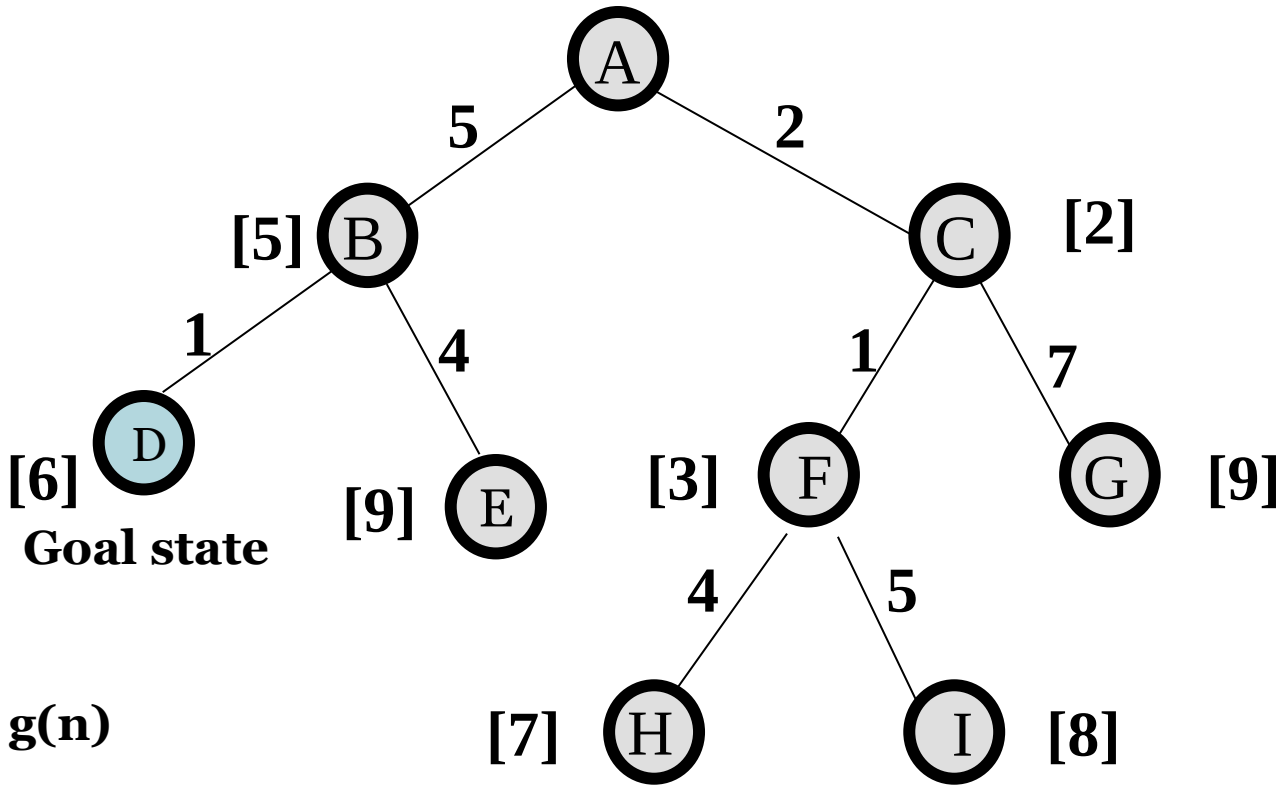
Uniform Cost Search (UCS)



[x] = g(n)

path cost of node n

Uniform Cost Search (UCS)



[x] = g(n)

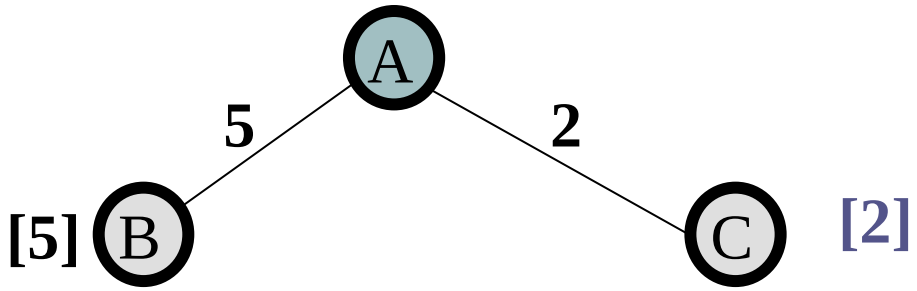
path cost of node n

Uniform Cost Search (UCS)

Explored:[]

Frontier = [{A,0}]

Visit: A
Expand: A
Is A a goal state?



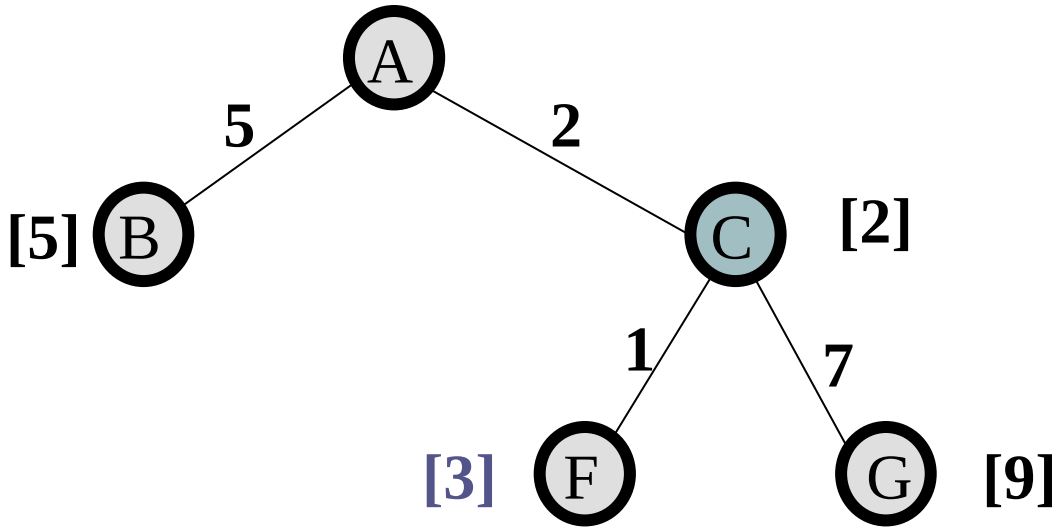
Uniform Cost Search (UCS)

Explored: [A]

Frontier = [(C,2),(B,5)]

Visit: C
Frontier = [(B,5)]

Is C a goal state? NO
Expand : C



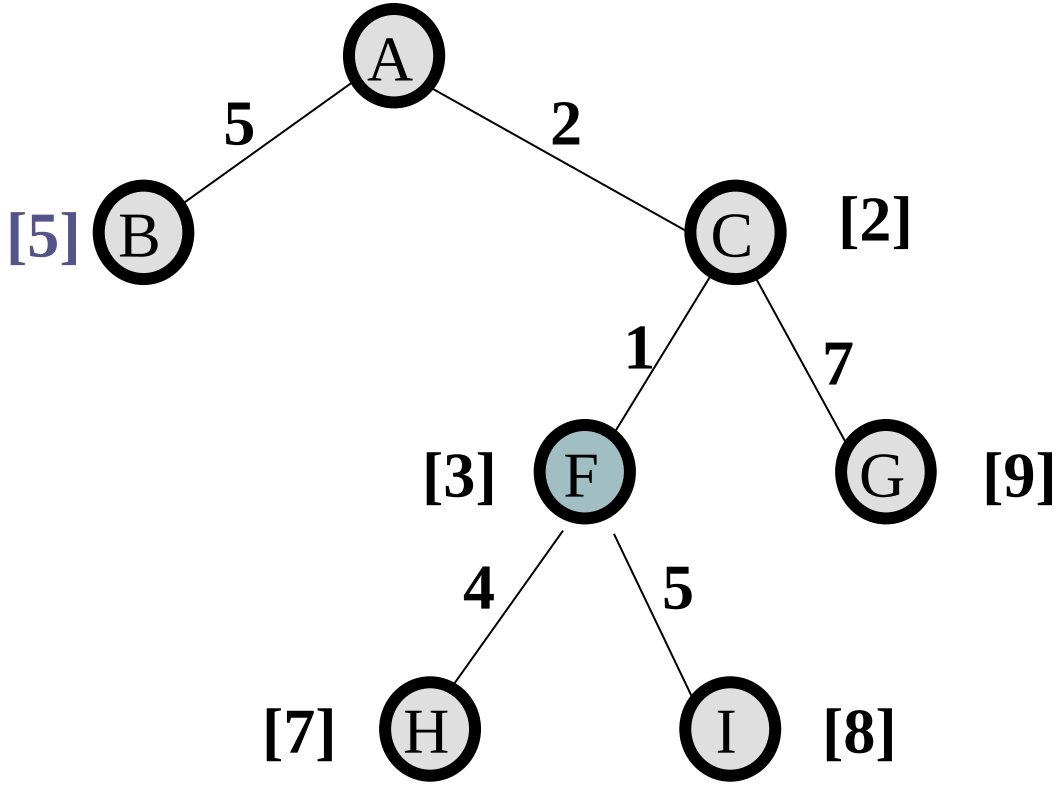
Uniform Cost Search (UCS)

Explored: [A,C]

fringe = [(F,3),(B,5),(G,9)]

Visit: F
fringe = [(B,5),(G,9)]

Is F a goal state?
Expand: F



Credit: Khola Naseem

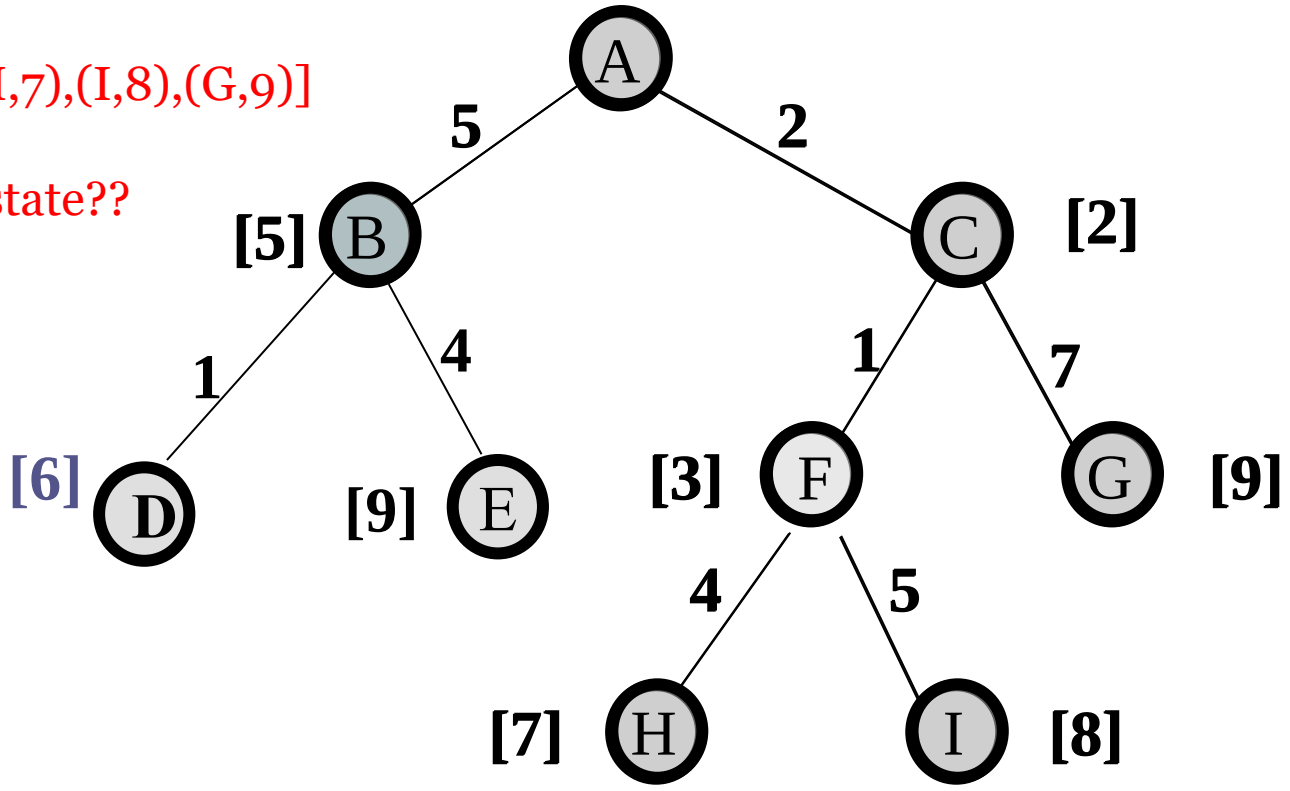
Uniform Cost Search (UCS)

Explored: [A,C,F]

fringe = [(B,5),(H,7),(I,8),(G,9)]

Visit: B
fringe = [(H,7),(I,8),(G,9)]

Is B a goal state??
Expand: B



Credit: Khola Naseem

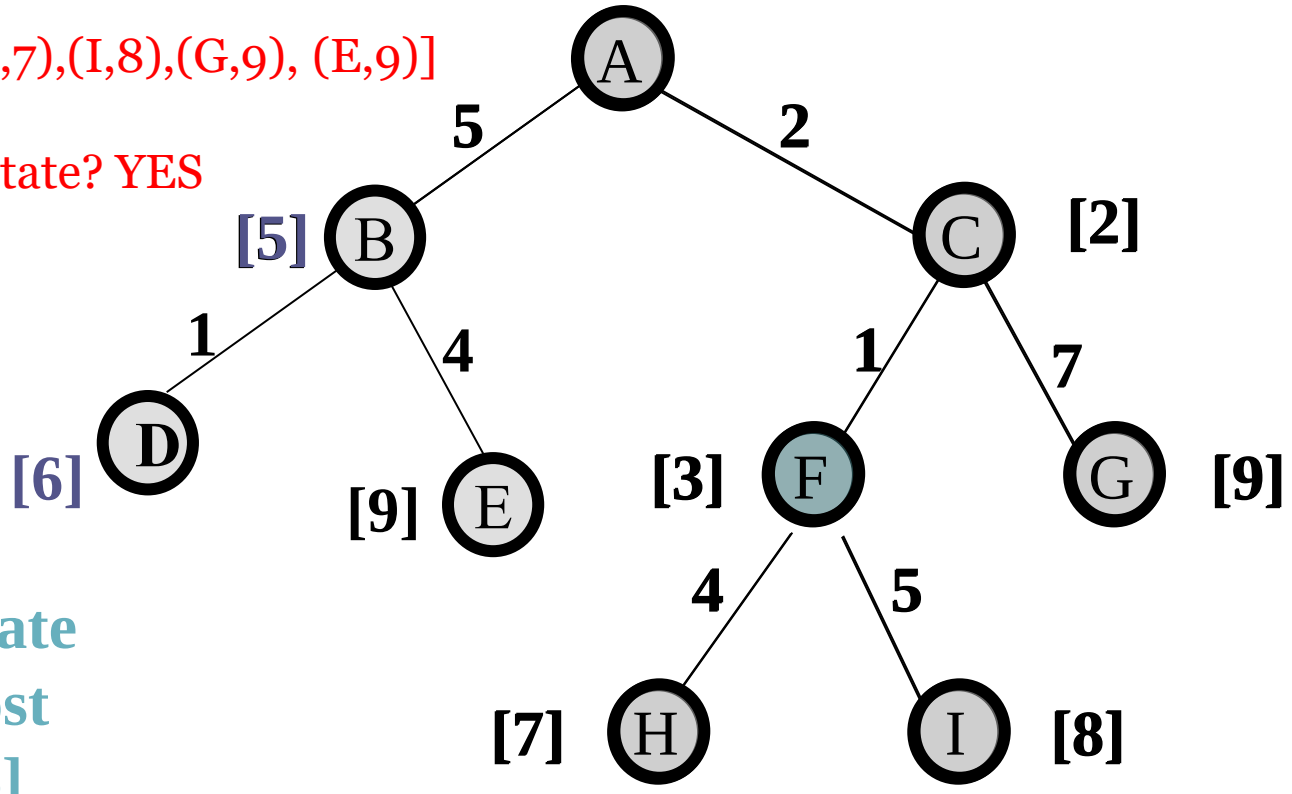
Uniform Cost Search (UCS)

Explored: [A,C,F,B]

fringe = [(D,6),(H,7),(I,8),(G,9), (E,9)]

Visit: D
fringe = [(H,7),(I,8),(G,9), (E,9)]

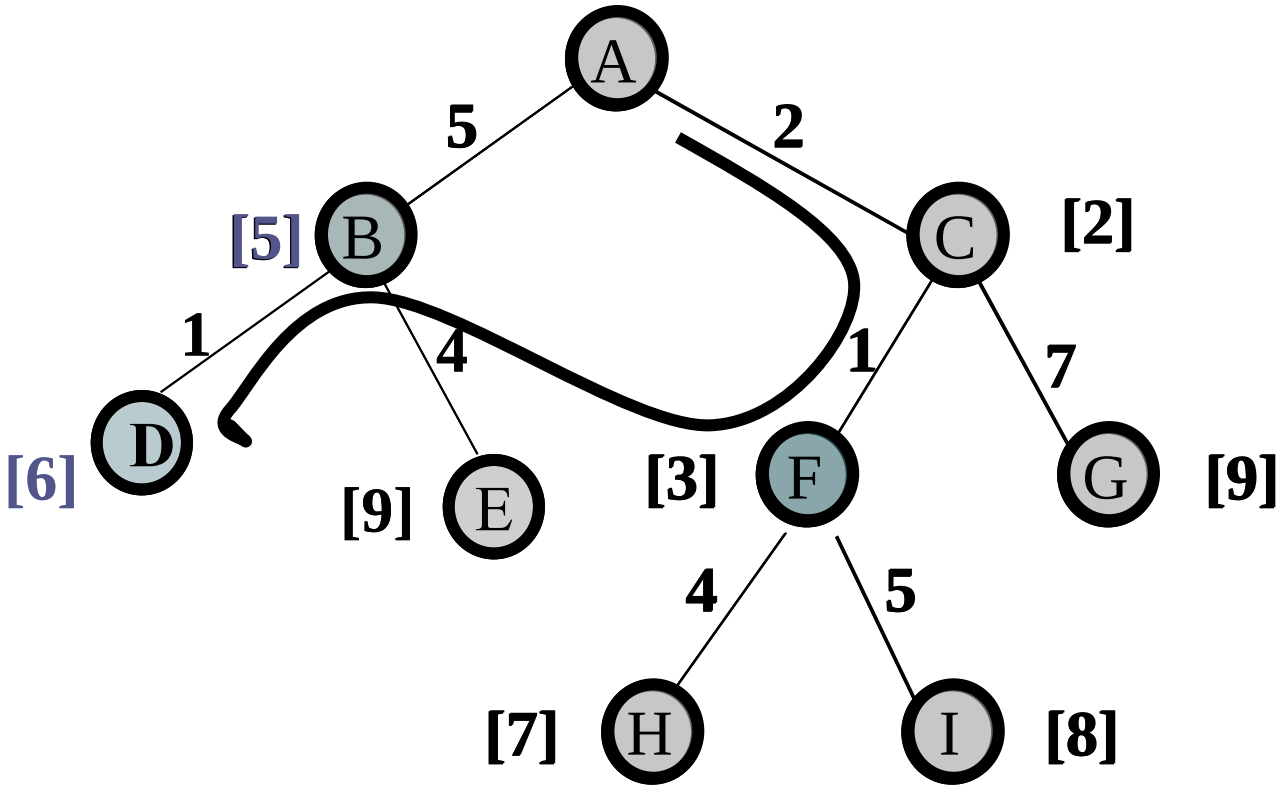
Is D a goal state? YES



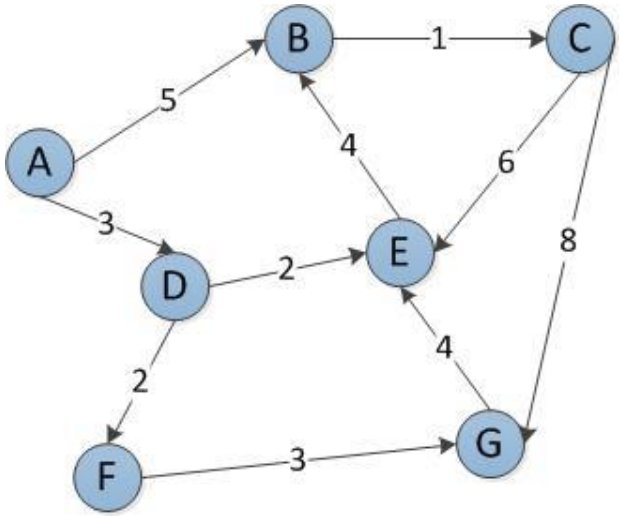
Goal state path cost g(n)=[6]

Credit: Khola Naseem

Uniform Cost Search (UCS)



Credit: Khola Naseem



- Start Node: A
- Goal Node: G

Step	Frontier	Expand[*]	Explored: a set of nodes
1	{(A,0)}	A	∅
2	{(A-D,3),(A-B,5)}	D	{A}
3	{(A-B,5),(A-D-E,5),(A-D-F,5)}	B	{A,D}
4	{(A-D-E,5),(A-D-F,5),(A-B-C,6)}	E	{A,D,B}
5	{(A-D-F,5),(A-B-C,6)}[*]	F	{A,D,B,E}
6	{(A-B-C,6),(A-D-F-G,8)}	C	{A,D,B,E,F}
7	(A-D-F-G,8)}	G	{A,D,B,E,F,C}
8	∅		

- Found the path: A -> D -> F -> G.
- *B is not added to the frontier because it is found in the explored set.

Credit: Khola Naseem

Uniform Cost Search (UCS)

- For any step-cost function, Uniform Cost search expands the node *n* with the lowest path cost.
- Implementation:
  - fringe = queue ordered by path cost, lowest first
- UCS takes into account the total cost: g(n).
- UCS is guided by path costs rather than depths. Nodes are ordered according to their path cost.
- Equivalent to breadth-first if step costs all equal

Credit: Khola Naseem

Uniform Cost Search (UCS)

- **Complete?** *Does it always find a solution if one exists?*
  - Yes
    - If b is finite
    - Completeness is guaranteed provided the cost of every step exceeds some small positive constant epsilon.
    - **Optimal?** *Does it always find a least-cost solution?*
  - Yes,
    - if step cost ≥ small +ve constant epsilon
- **Time Complexity:**
  - $O(b^{\wedge(C^*/epsilon)}) \sim O(b^{C^*})$
    - where C^* is the cost of the optimal solution
- **Space Complexity:**
  - $O(b^{\wedge(C^*/epsilon)}) \sim O(b^{C^*})$
    - where C^* is the cost of the optimal solution

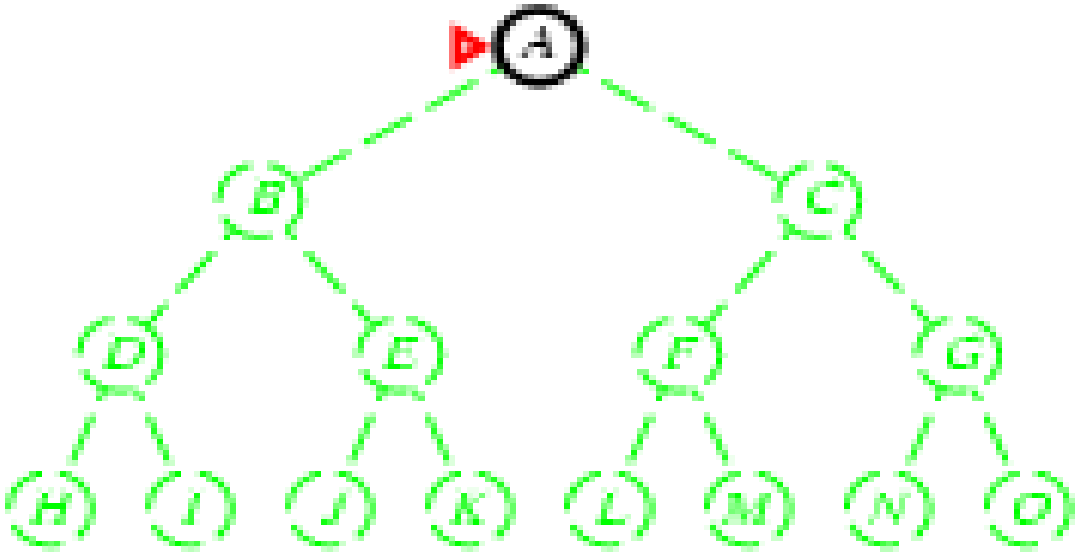
Credit: Khola Naseem

Depth first search:

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end
  - goes back and expands nodes at shallower levels
  - Dead end leaf nodes but not the goal
- Backtracking search
  - only one successor is generated on expansion
  - rather than all successors
- fewer memory

Depth first search:

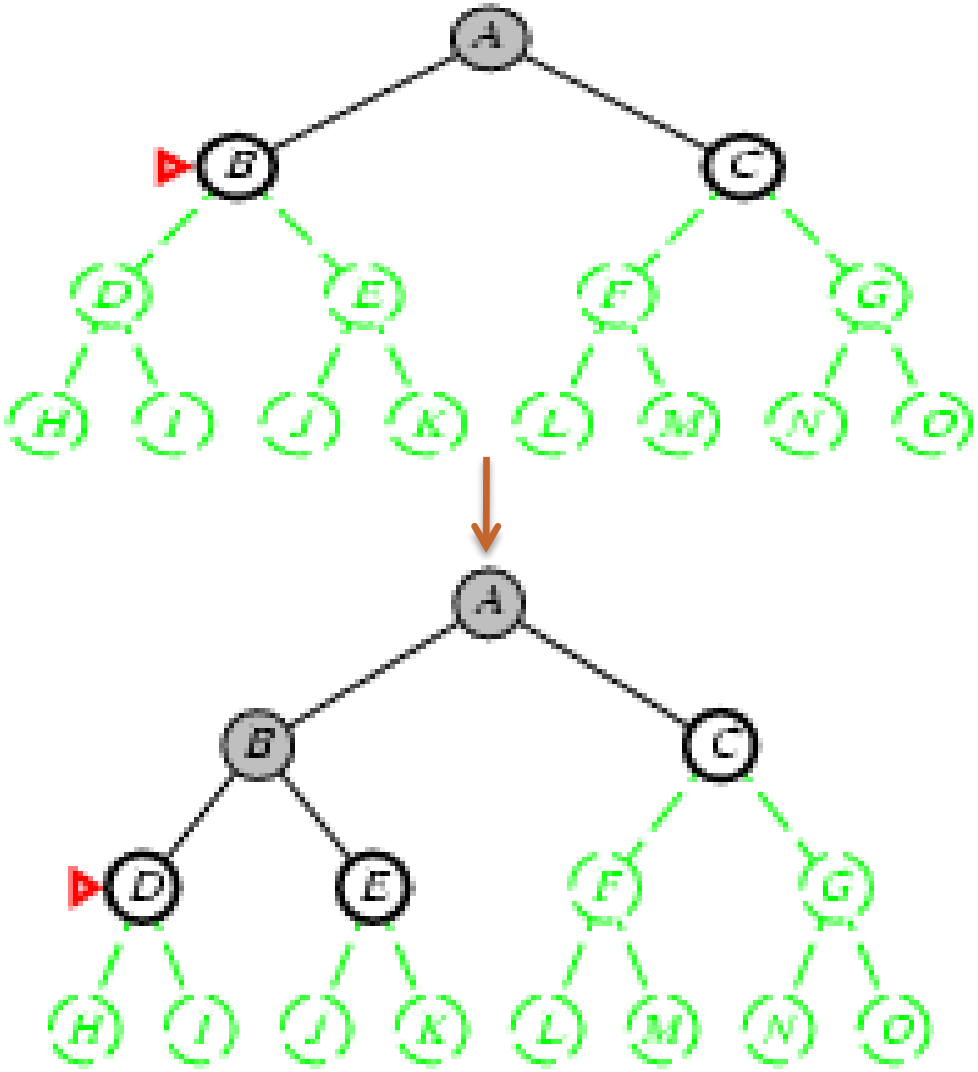
- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front



Credit: Khola Naseem

Depth first search:

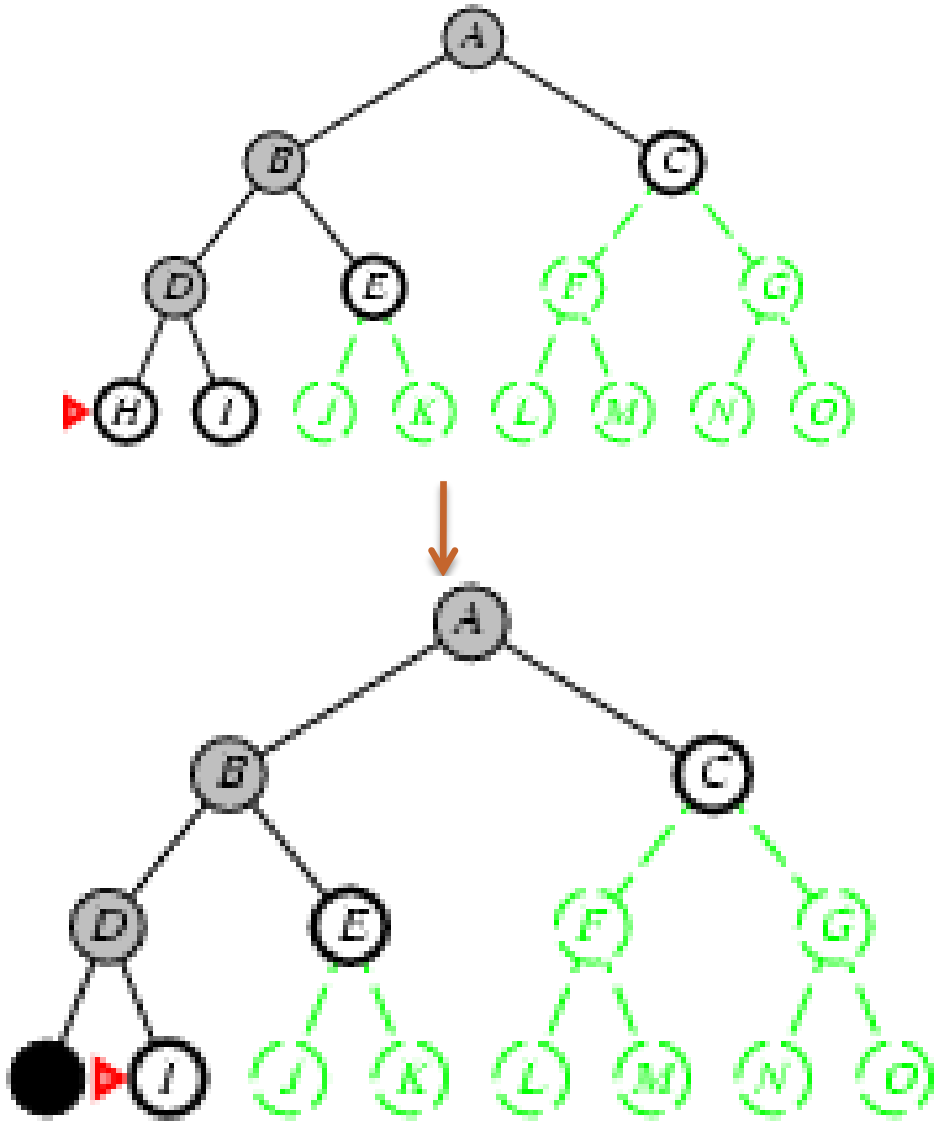
- Expand deepest unexpanded node



Credit: Khola Naseem

Depth first search:

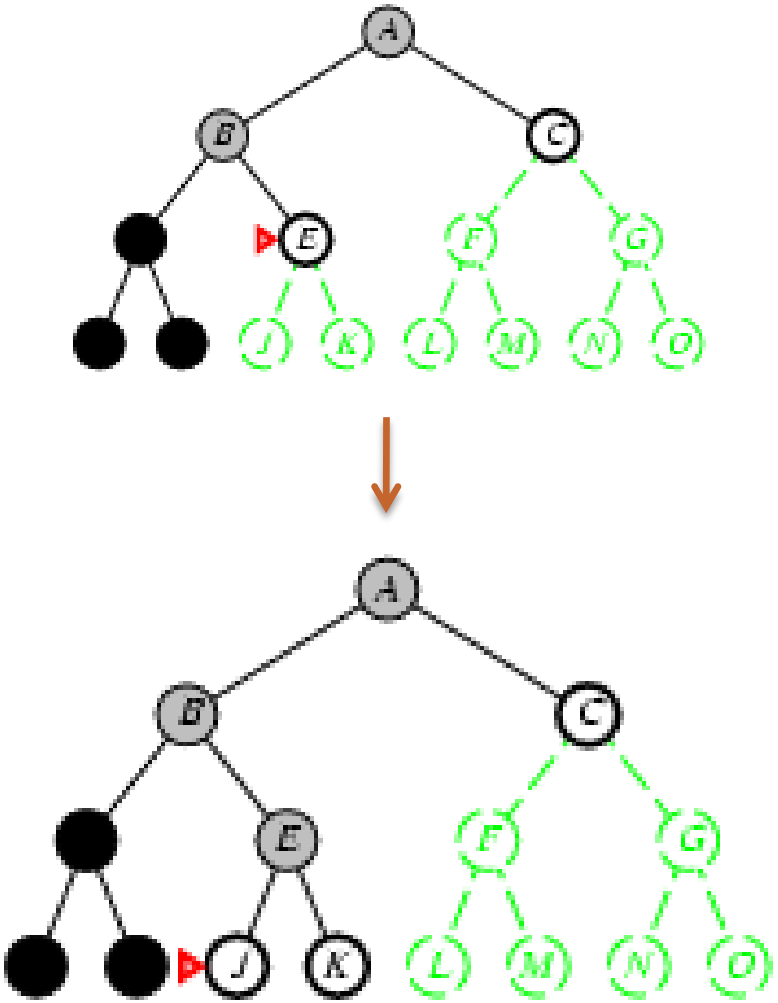
- Expand deepest unexpanded node



Credit: Khola Naseem

Depth first search:

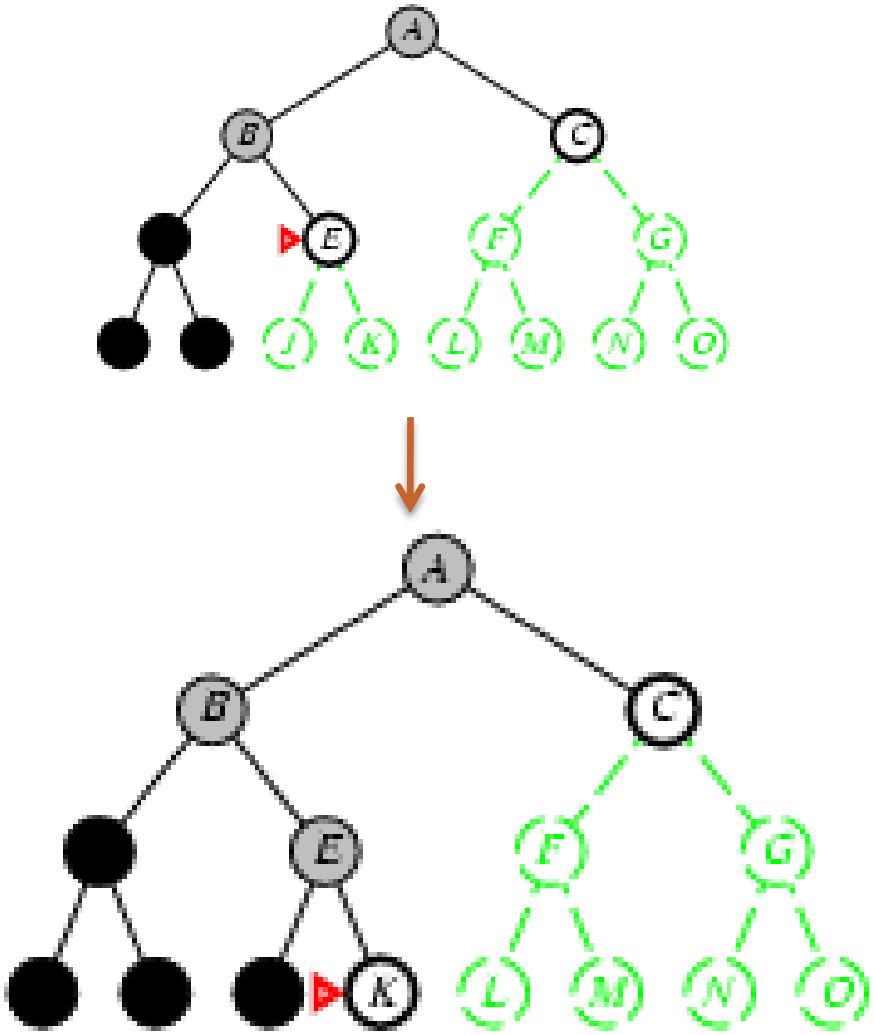
- Expand deepest unexpanded node



Credit: Khola Naseem

Depth first search:

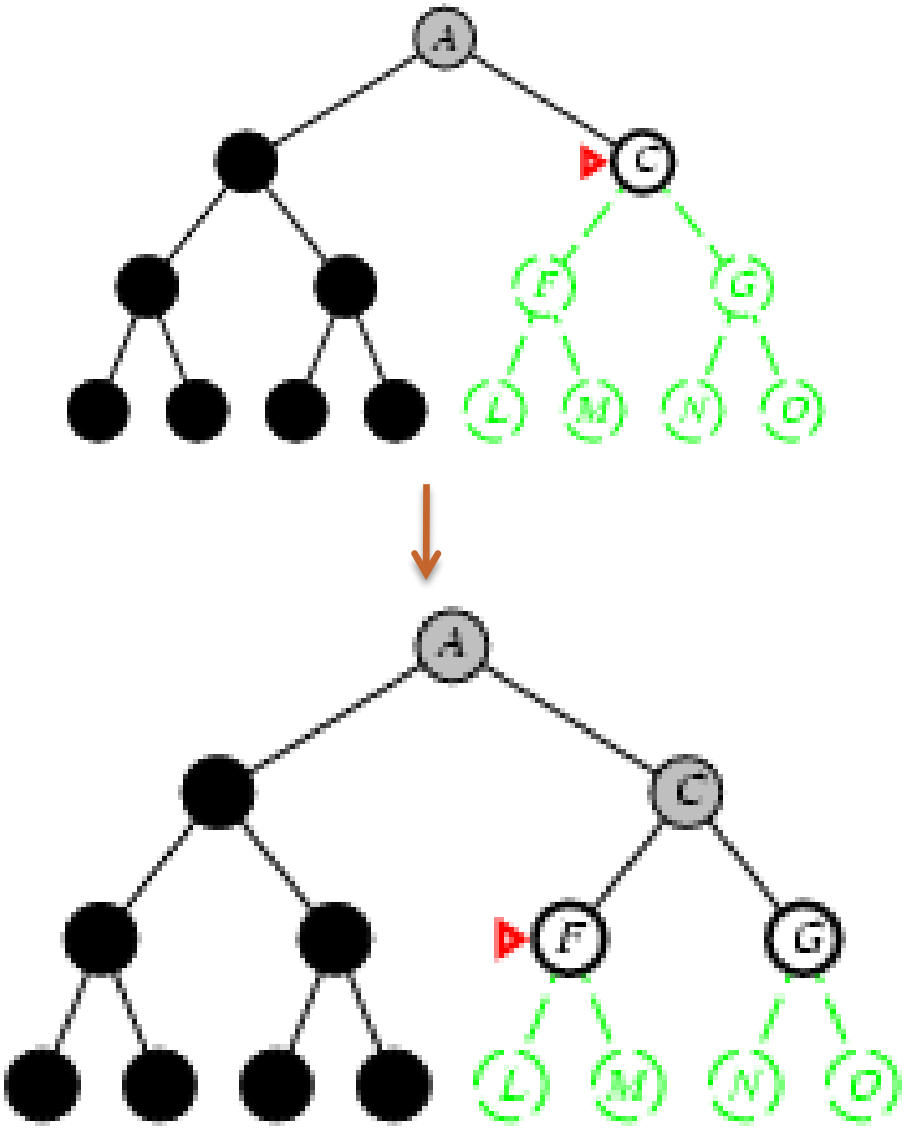
- Expand deepest unexpanded node



Credit: Khola Naseem

Depth first search:

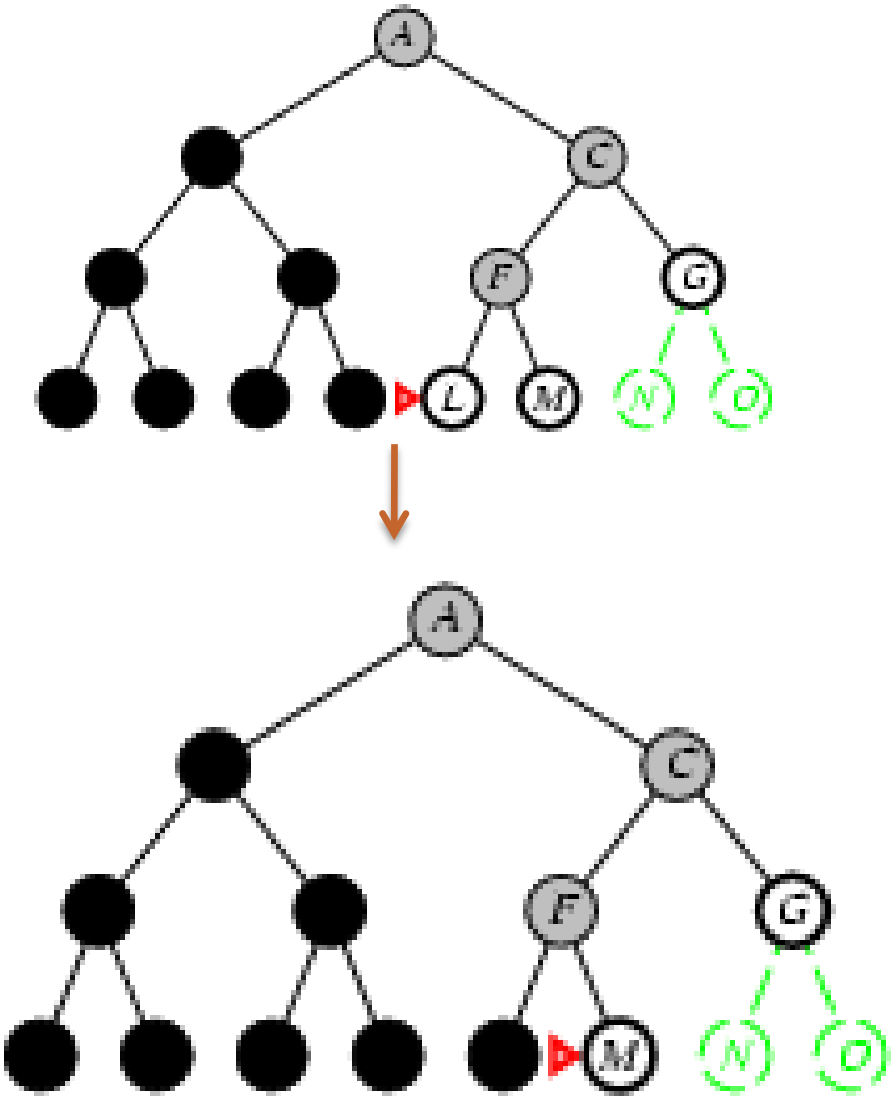
- dfs



Credit: Khola Naseem

Depth first search:

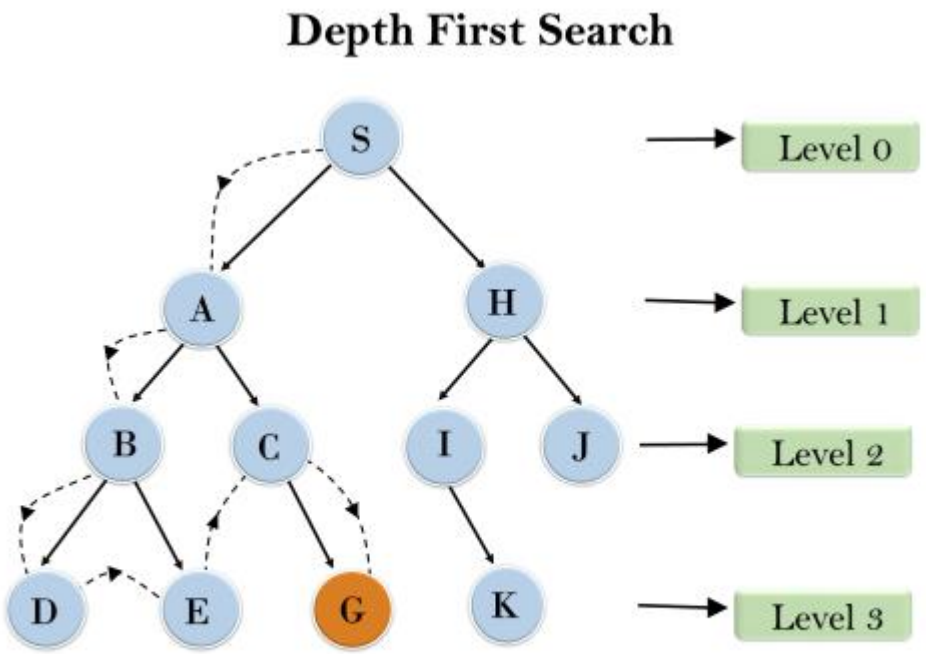
- dfs



Credit: Khola Naseem

Depth first search:

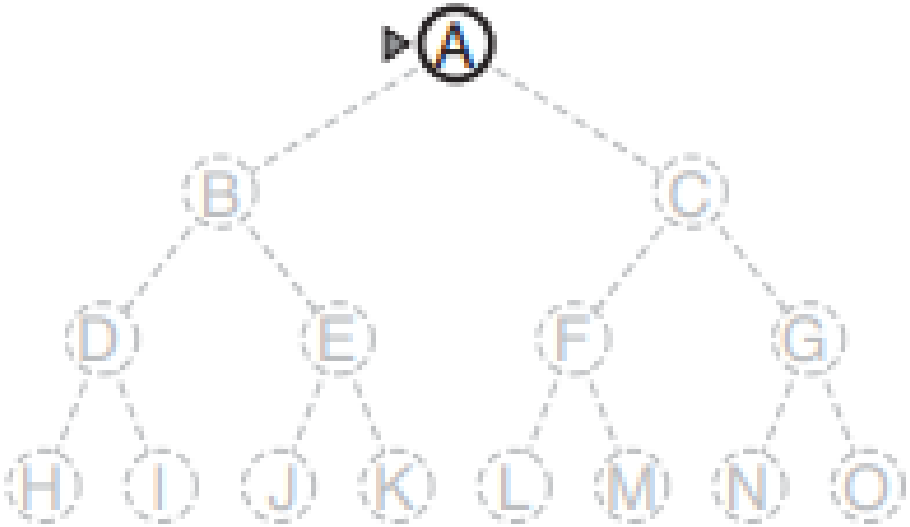
- dfs



Credit: Khola Naseem

Depth first search:

- dfs



Credit: Khola Naseem

Depth first search:

- Implementation: use of a Last-In-First-Out queue or stack(LIFO).

  Enqueue nodes in LIFO (last-in, first-out) order.
- Not complete
  - because a path may be infinite or looping
  - then the path will never fail and go back try another option
- Not optimal
  - it doesn't guarantee the best solution, May find a non-optimal goal first
- It overcomes
  - the time and space complexities

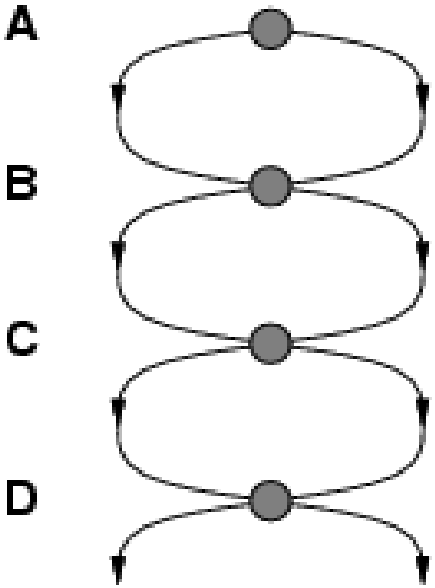
Depth first search:

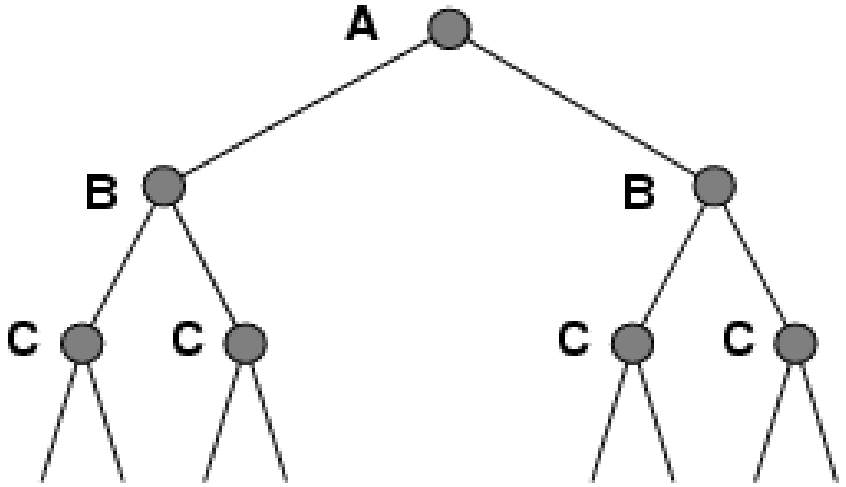
- the time and space complexities
- Time? O(b^m) with m=maximum depth
  - terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? O(bm), i.e., linear space! (we only need to
  - remember a single path + expanded unexplored nodes)

Credit: Khola Naseem

Repeated states:

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





Credit: Khola Naseem

DFS: Optimality :

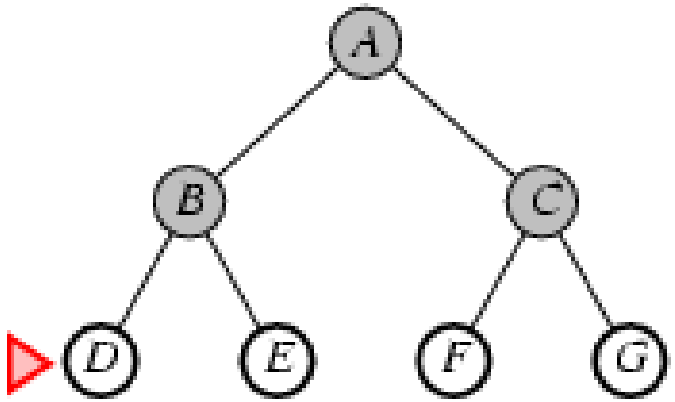
- Time Complexity
  - worst case: there is 1 goal leaf at the RHS
  - so DFS will expand all nodes

    (m is cutoff)

    $=b\hat{}0+b\hat{}1+b\hat{}2+\ldots+b\hat{}m$

    $= O(bm)$
- Space Complexity
  - O(bm) i.e., linear space!
  - we only need to remember a single path + expanded unexplored nodes

$$O(\underbrace{m+m+\cdots+m}_{b})$$

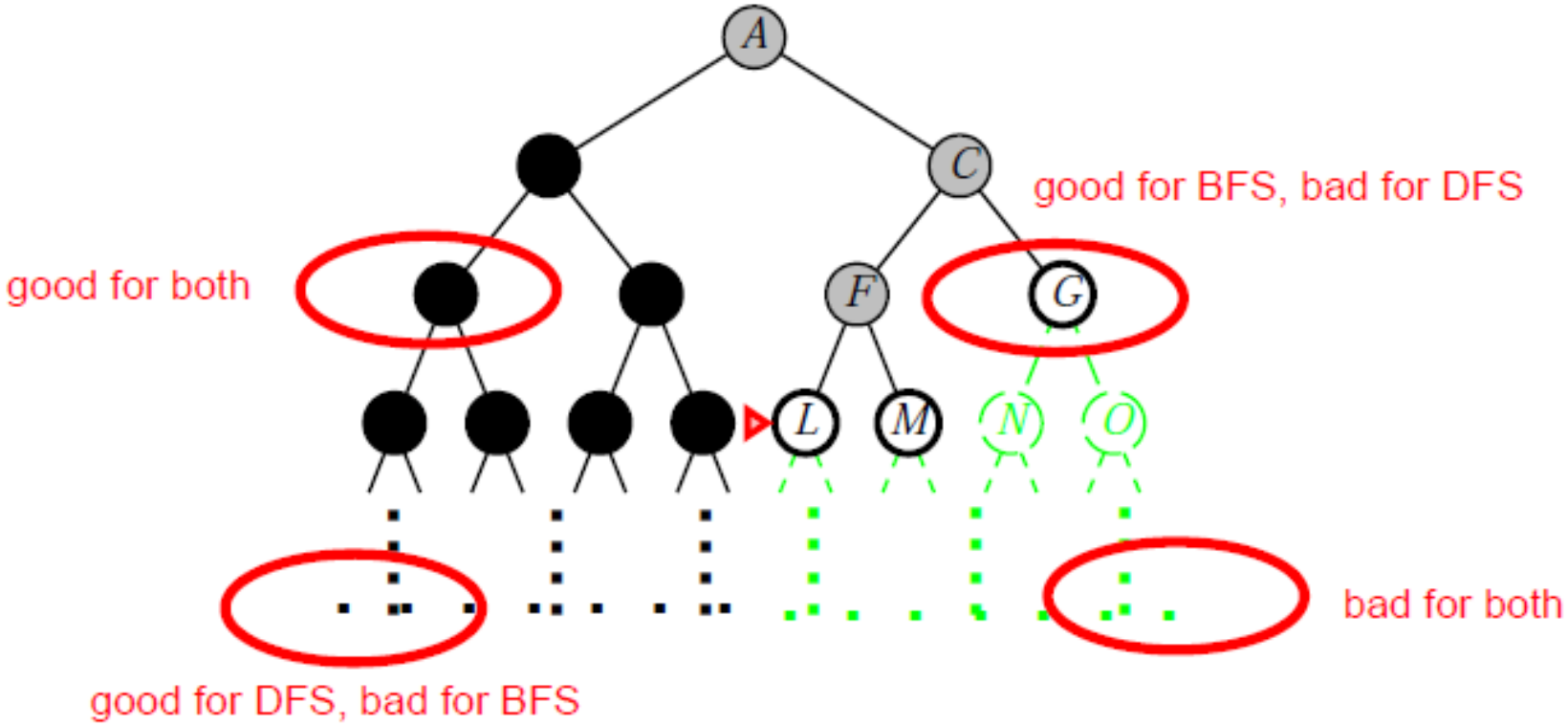


Start Node: A
Goal Node: G

Step	Frontier	Expand[*]	Explored: a set of nodes
1	{A}	A	∅
2	{(A-B),(A-C)}	B	{A}
3	{(A-B-D),(A-B-E),(A-C)}	D	{A,B}
4	{(A-B-E),(A-C)}	E	{A,B,D}
5	{(A-C)}	C	{A,B,D,E}
6	{(A-C-F),(A-C-G)}	F	{A,B,D,E,C}
7	(A-C-G)}	G	{A,B,D,E,C,F,G}
8	∅		

Found the path: A -> C -> G.

BFS or DFS


A
C
F
G
L
M
N
O
good for BFS, bad for DFS
good for both
bad for both
good for DFS, bad for BFS