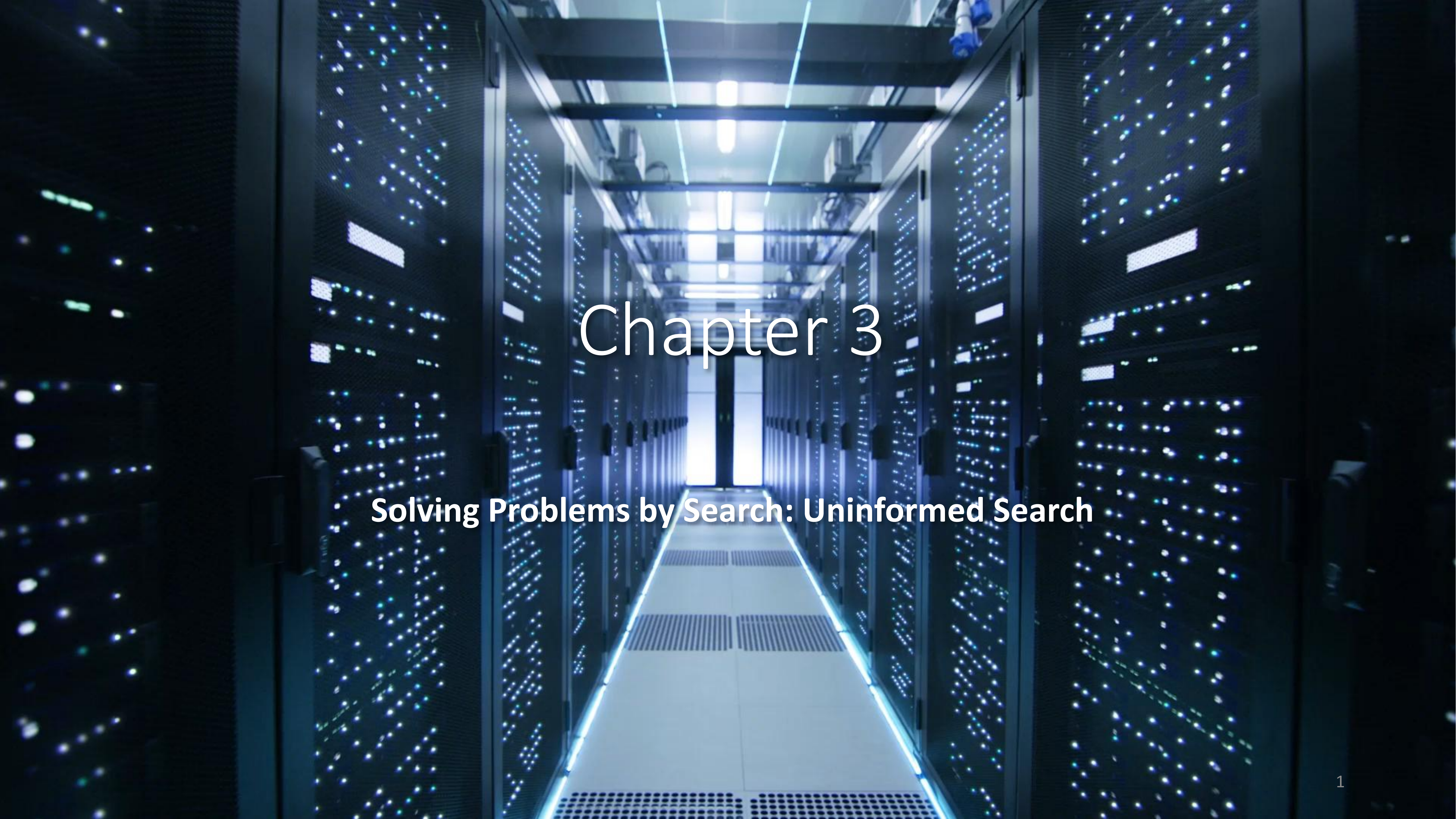


Chapter 3

Solving Problems by Search: Uninformed Search

Uninformed search algorithm

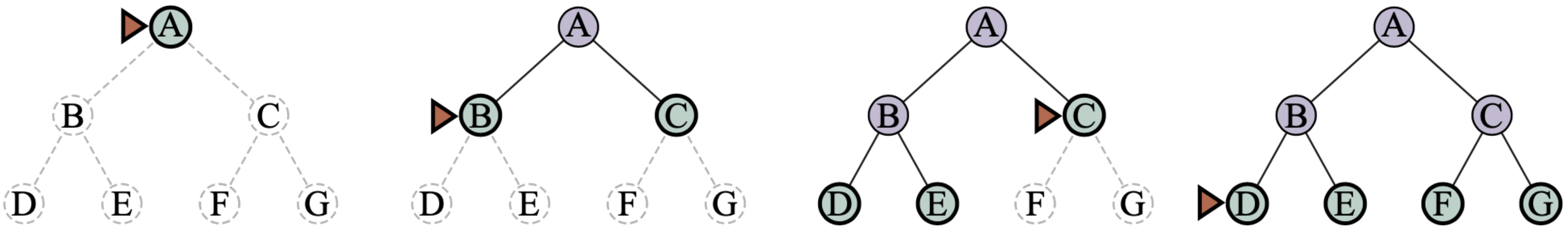
- Uninformed search (blind search) strategies use only the information available in the problem definition

- Strategies that know whether one non-goal state is better than another are called informed search or heuristic search

Uninformed strategies: use only the information available in the problem definition:

1. Breadth First Search (BFS)
2. Uniform Cost Search (UCS)
3. Depth First Search (DFS)
4. Depth Limited Search (DLS)
5. Iterative Deepening Search (IDS)
6. Bidirectional search

1. Breadth First Search (BFS)

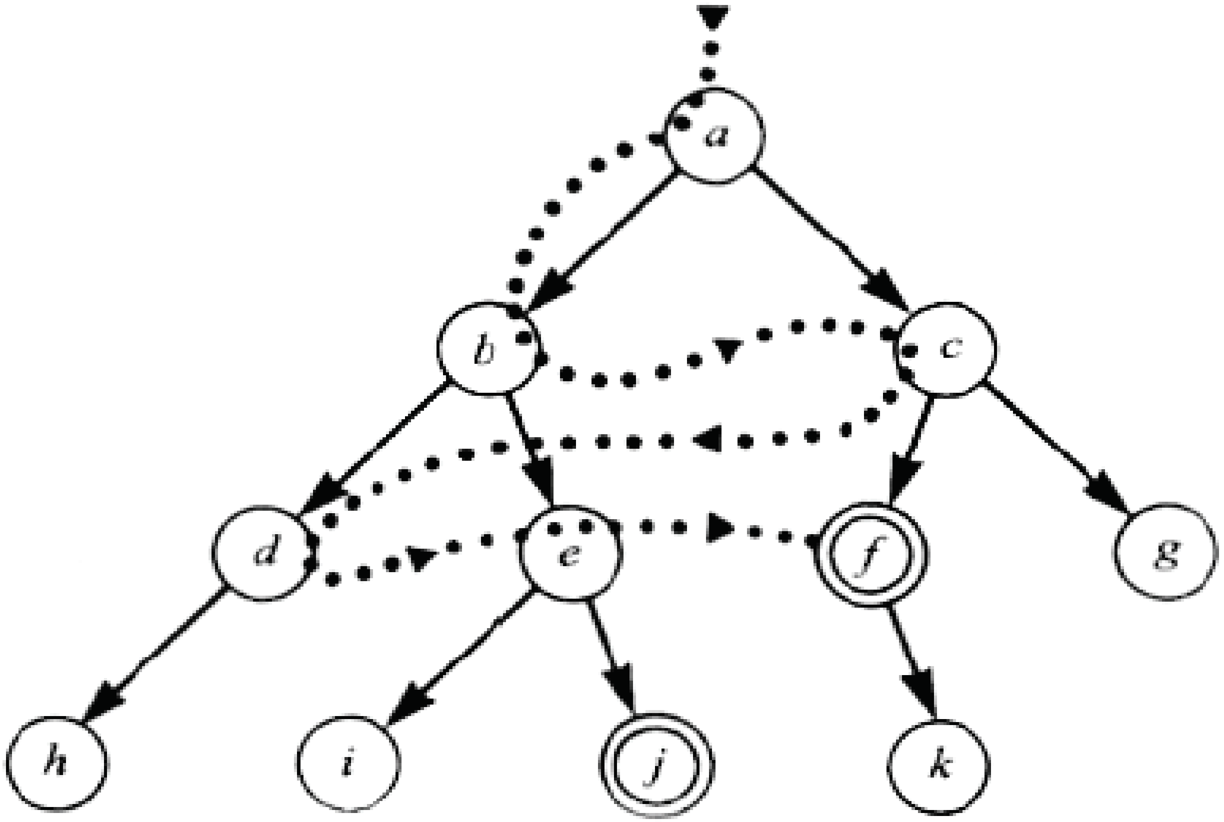


- **Main idea**: Expand all nodes at depth i before expanding nodes at depth $i + 1$. (Shallow nodes are expanded before deeper nodes)
- **Implementation**:
  - The frontier list is a First-In-First-Out queue (FIFO).
  - Test for **goal** before putting in FIFO.

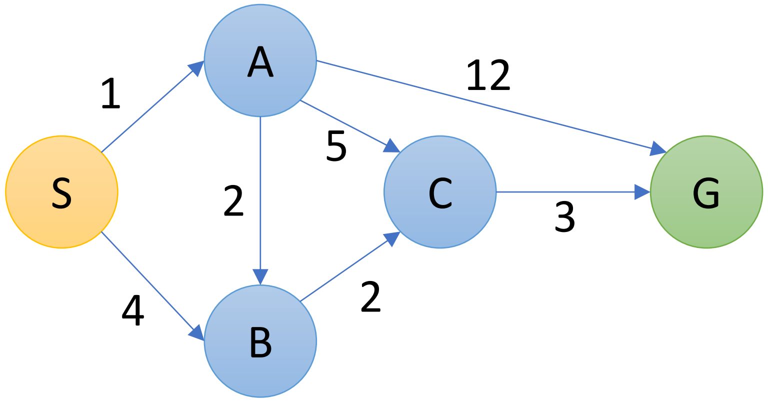
1. Breadth First Search (BFS)

```
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)
```

Breadth First Search (BFS)



1. BFS: Example



- Initial State = S, Path-Cost = 0
- Frontier:

S				

- Explored:

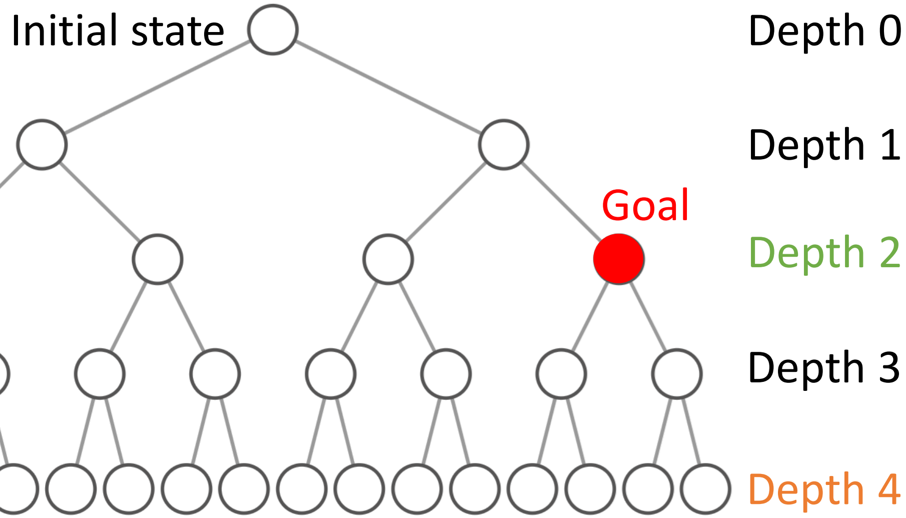
Recall: Evaluating search algorithms

1. **Completeness**: is the algorithm guaranteed to find a solution if one exists?
2. **Time complexity**: number of operations (time) necessary to find a solution.
3. **Space complexity**: memory requirement.
4. **Optimality**: if there are multiple solutions, does the algorithm find the best one (minimum cost)?

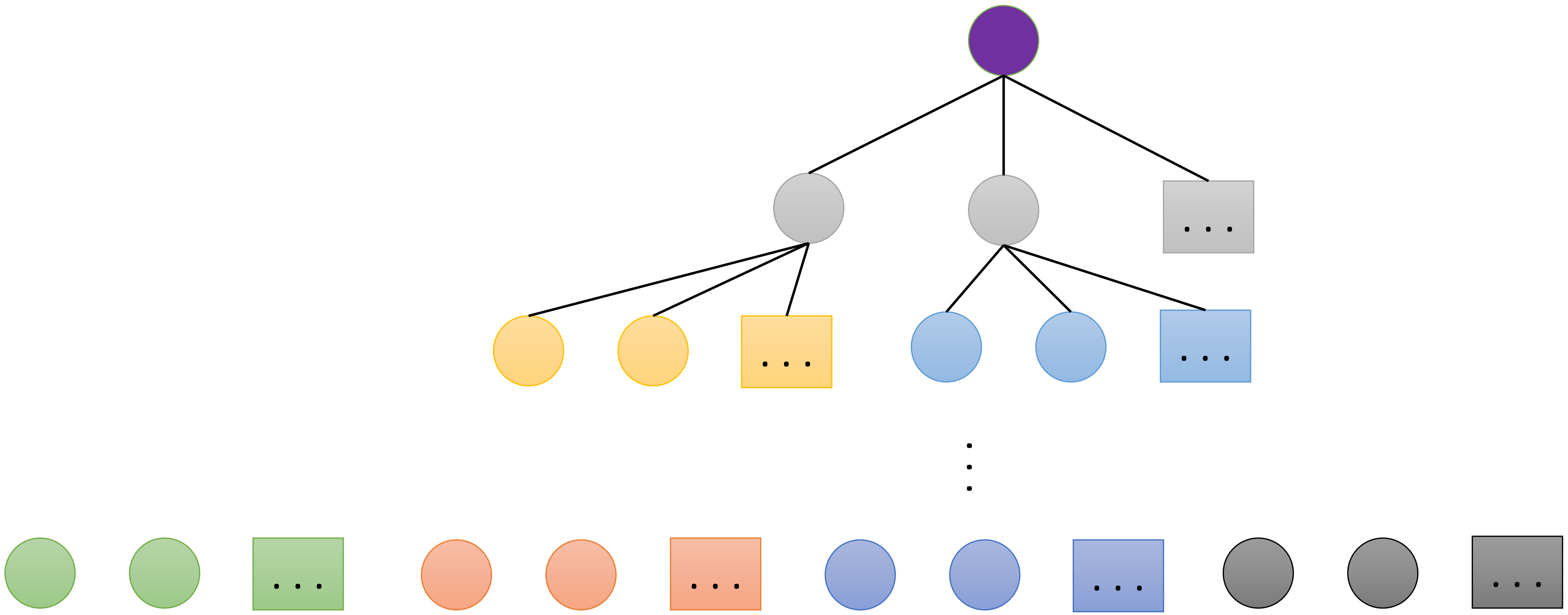
- Time and space complexity are measured in terms of
  - $\boldsymbol{b}$: maximum branching factor of the search tree
  - $\boldsymbol{d}$: depth of the best solution
  - $\boldsymbol{m}$: maximum depth of the state space (may be infinite)

How to find complexity ?

- Take the worst possible example and compute its running time
- Difficult to do with graphs: use trees
- We use a complete tree with branching factor b and depth m. The goal is at depth d.
- Example: $b = 2, m = 4, d = 2$



1. BFS Performance



b successors

$b * b = b^2$ successors

$\underbrace{b * b * \cdots * b}_{d \text{ times}} = b^d$ successors

1. BFS Performance

- Completeness: Guaranteed for finite space. Guaranteed when a solution exists.
- Optimality: Yes, if step-costs are equal, otherwise no.
- Time Complexity: Total number of nodes generated: $O(b^d)$
  - What if the goal test was done when the node was selected for expansion instead of added to the frontier? $O(b^{d+1})$ (This is not BFS)
- Space Complexity: $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier

1. BFS Complexity

Bigger Problem

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

2. Uniform Cost Search (UCS)

- When all step costs are equal, breadth-first search is optimal
  - it always expands the *shallowest* unexpanded node
- With any step cost function, an optimal algorithm expands the *lowest path cost* $g(n)$, instead of expanding the shallowest node

UCS:

- **Main idea**: Expand the cheapest node, where the cost is the path cost $g(n)$
- **Implementation**:
  - The frontier list is a priority queue with $g(n)$ as the priority

2. UCS

```
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
```

2. UCS

```
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
```

2. UCS

```
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
```

Expand node with smallest cost

2. UCS

```
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
```

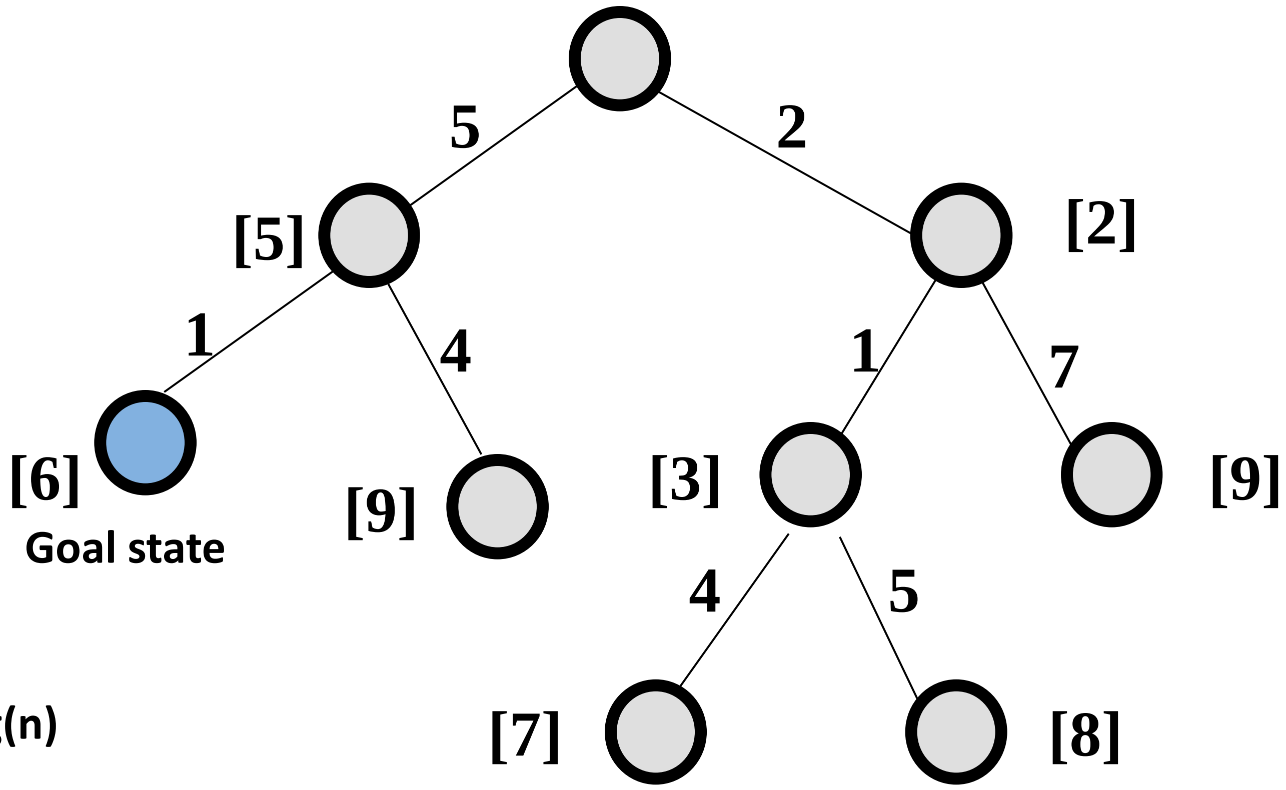
Check when expanded rather than when generated, because what if it is on a suboptimal path?

2. UCS

```
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
```

What if a better path is found?

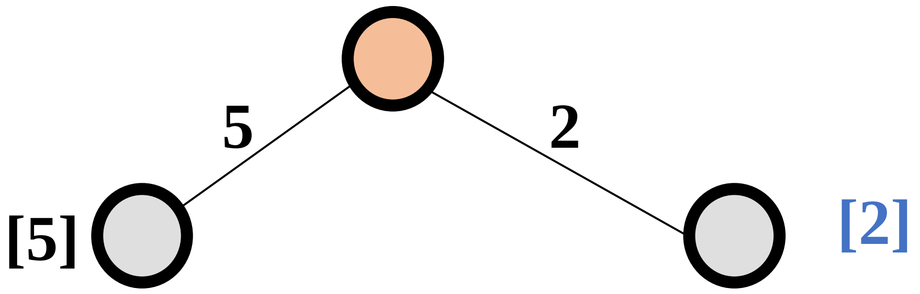
Uniform Cost Search (UCS)



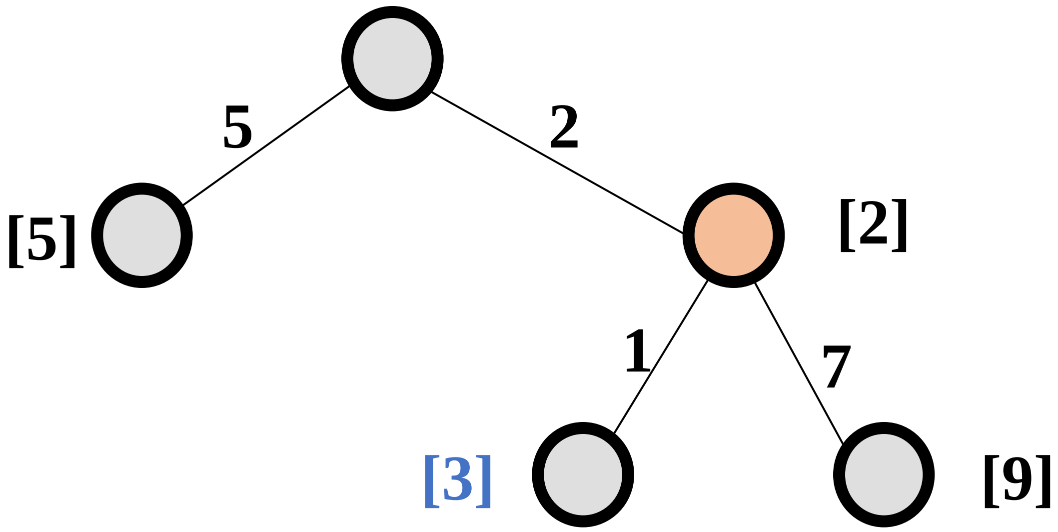
[x] = g(n)

path cost of node n

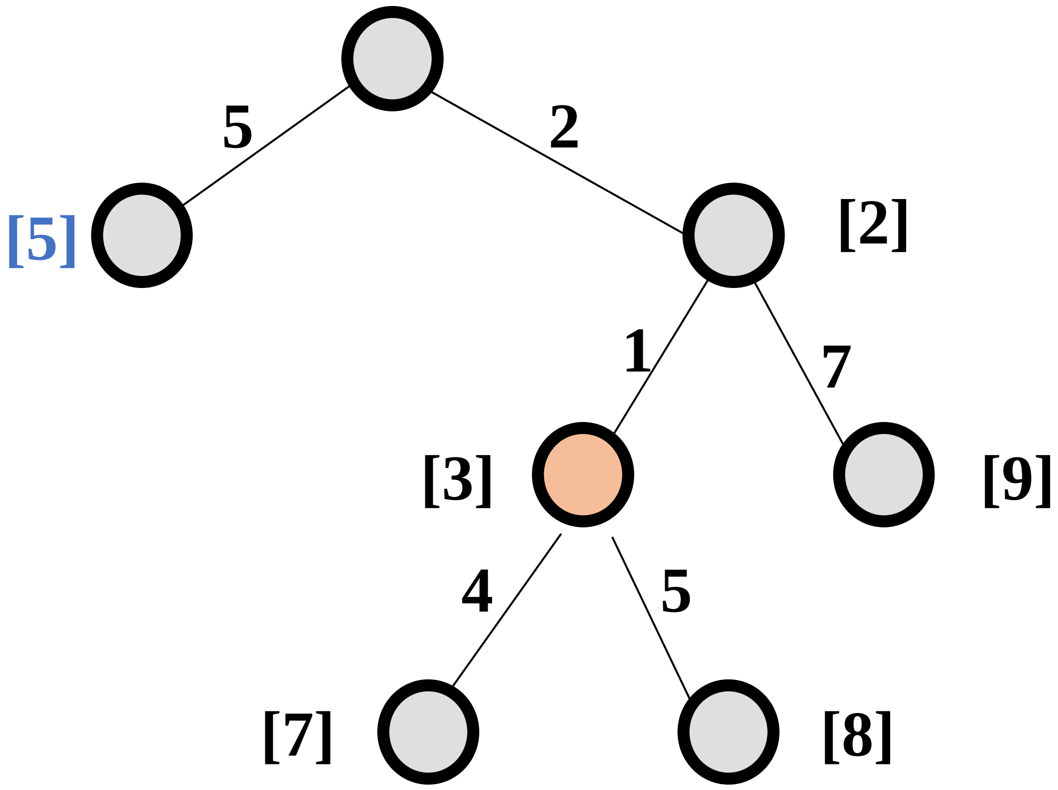
Uniform Cost Search (UCS)



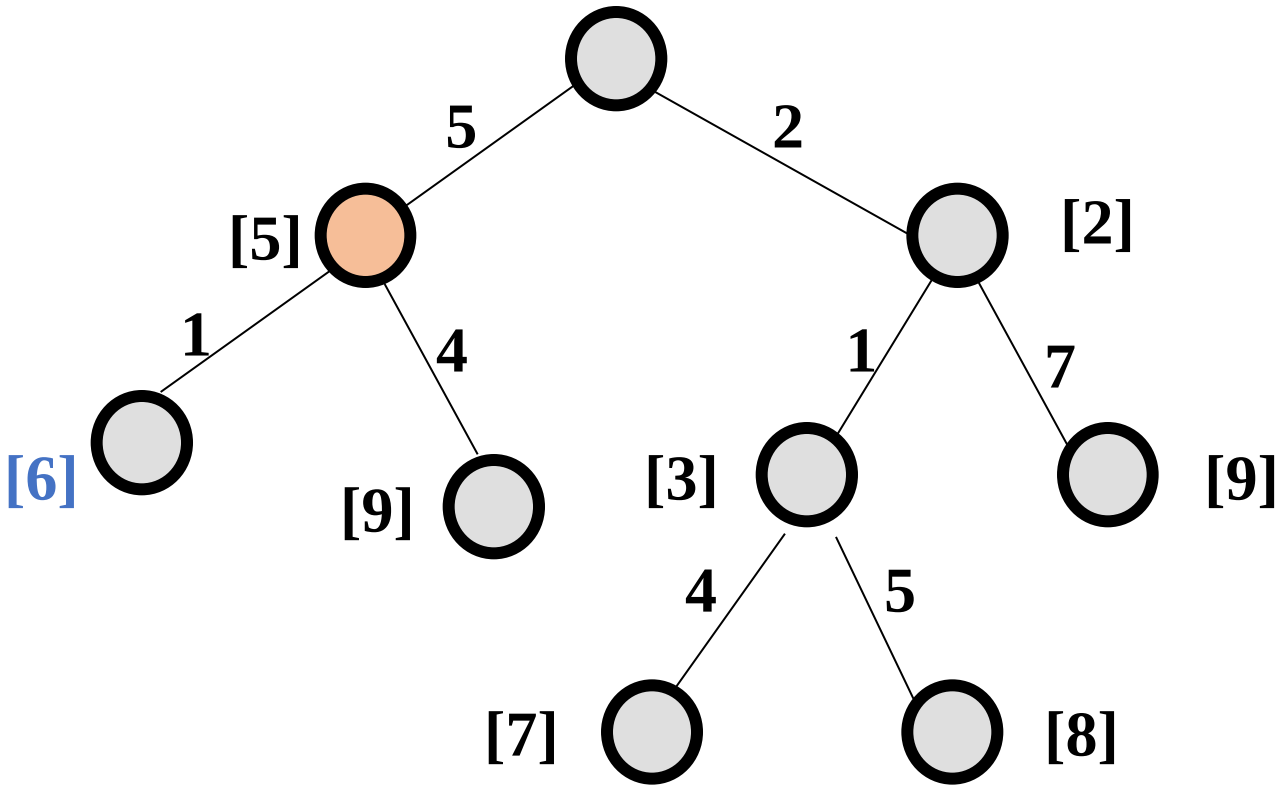
Uniform Cost Search (UCS)



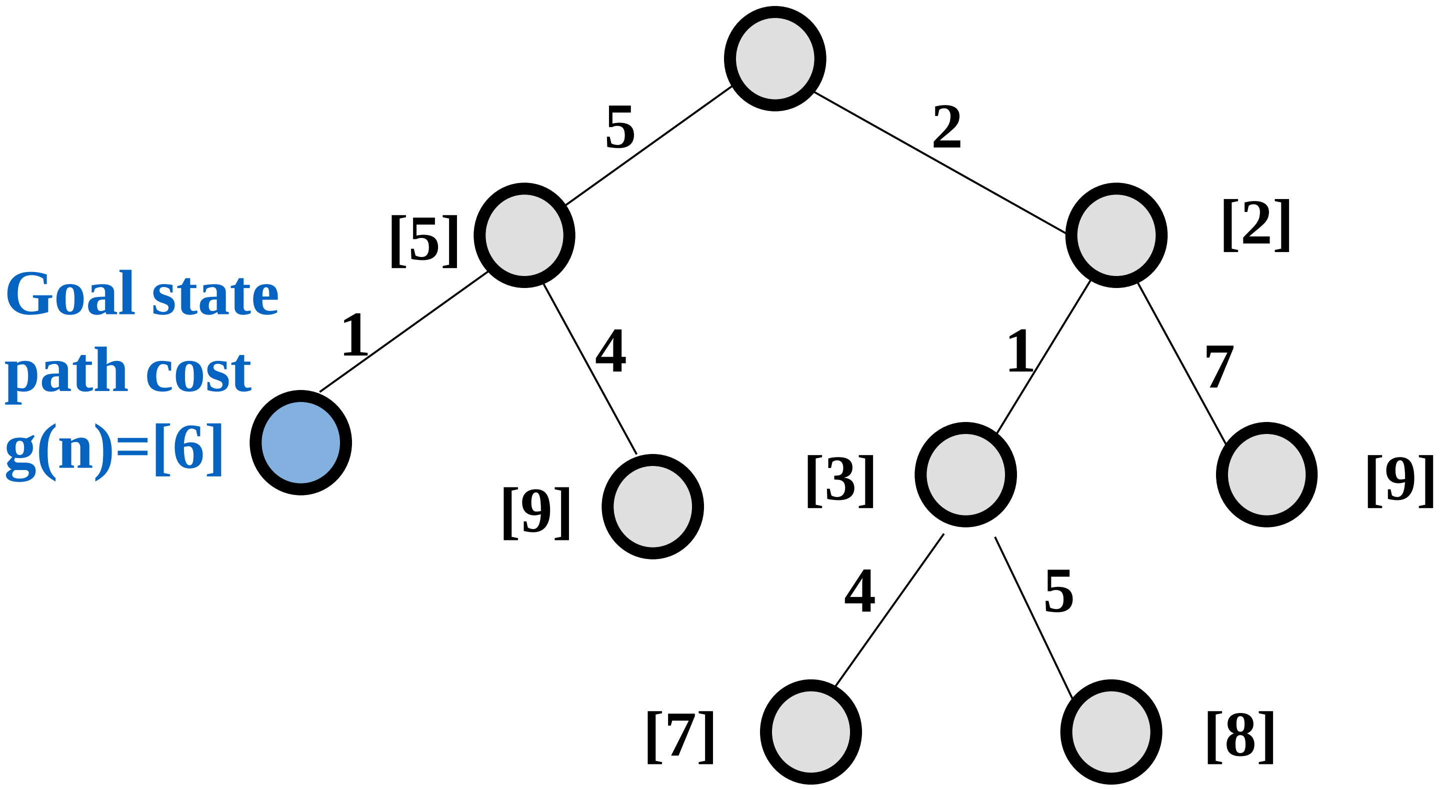
Uniform Cost Search (UCS)



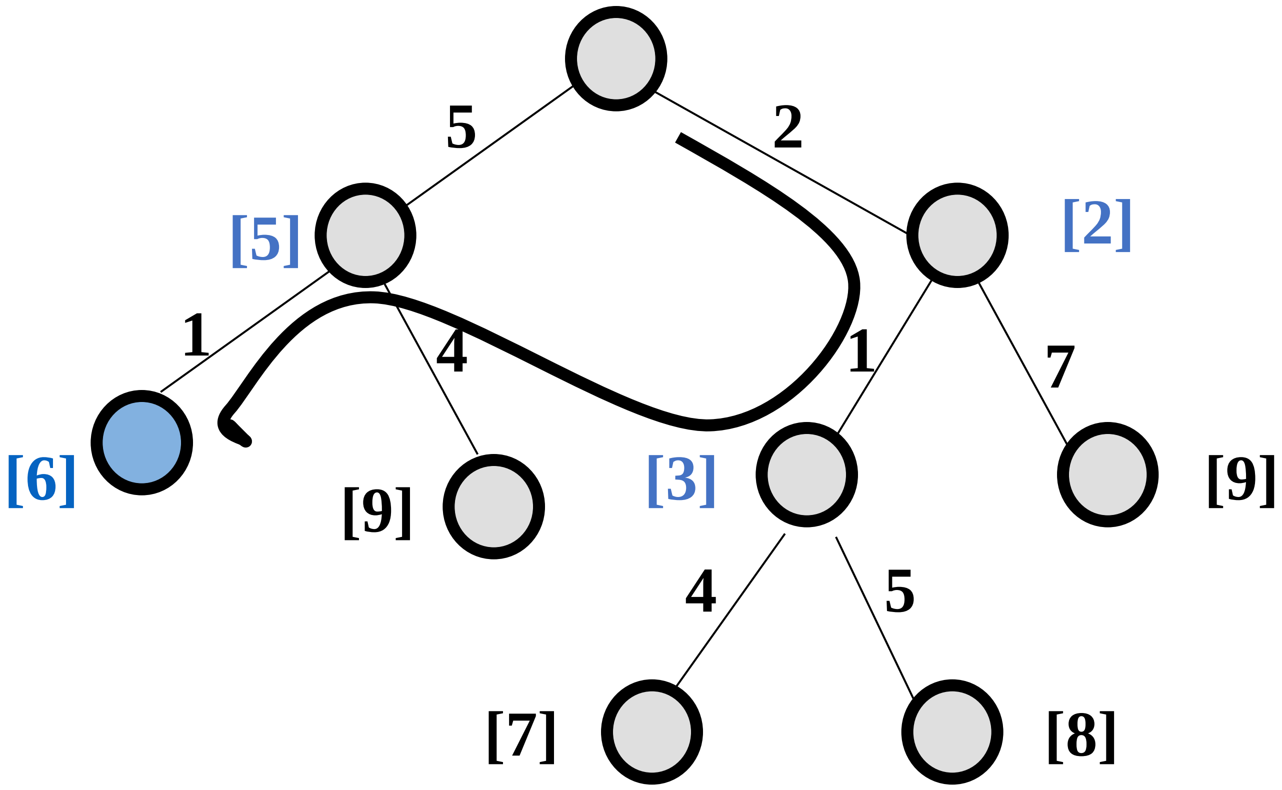
Uniform Cost Search (UCS)



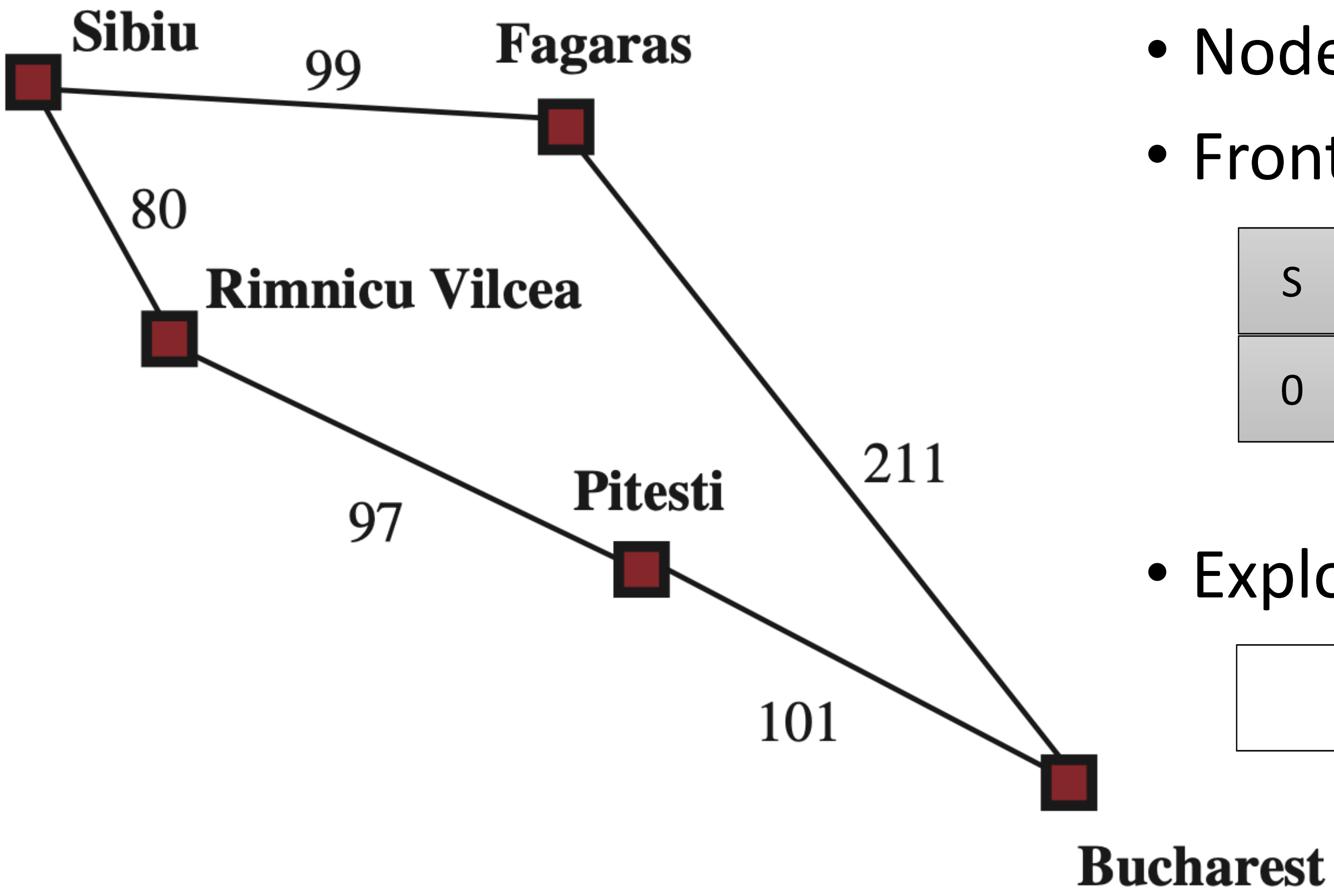
Uniform Cost Search (UCS)



Uniform Cost Search (UCS)



2. UCS: Example

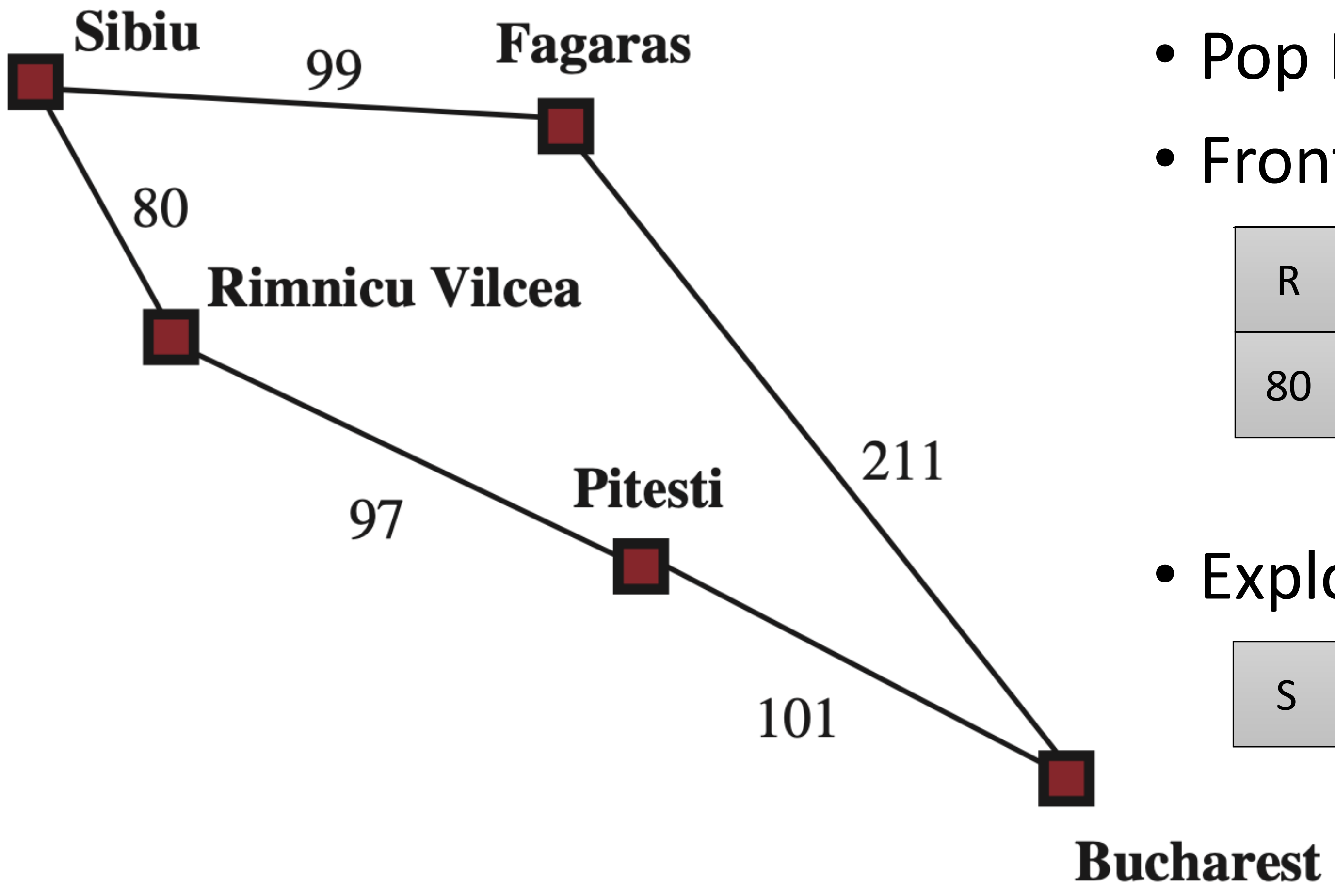


- Node = Sibiu
- Frontier

S				
0				

- Explored

2. UCS: Example



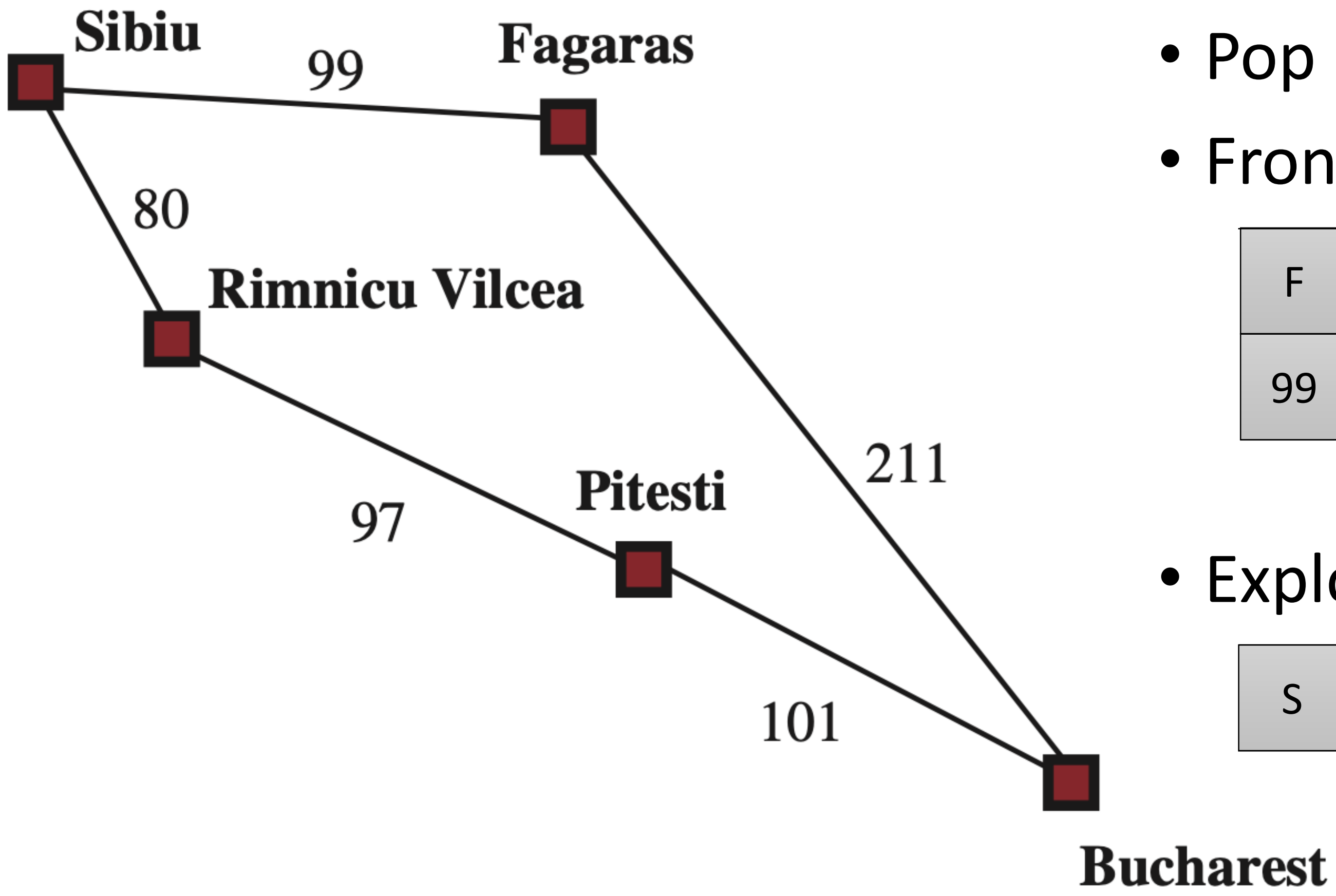
- Pop Node = Sibiu, Goal? No
- Frontier

R	F			
80	99			

- Explored

S				

2. UCS: Example



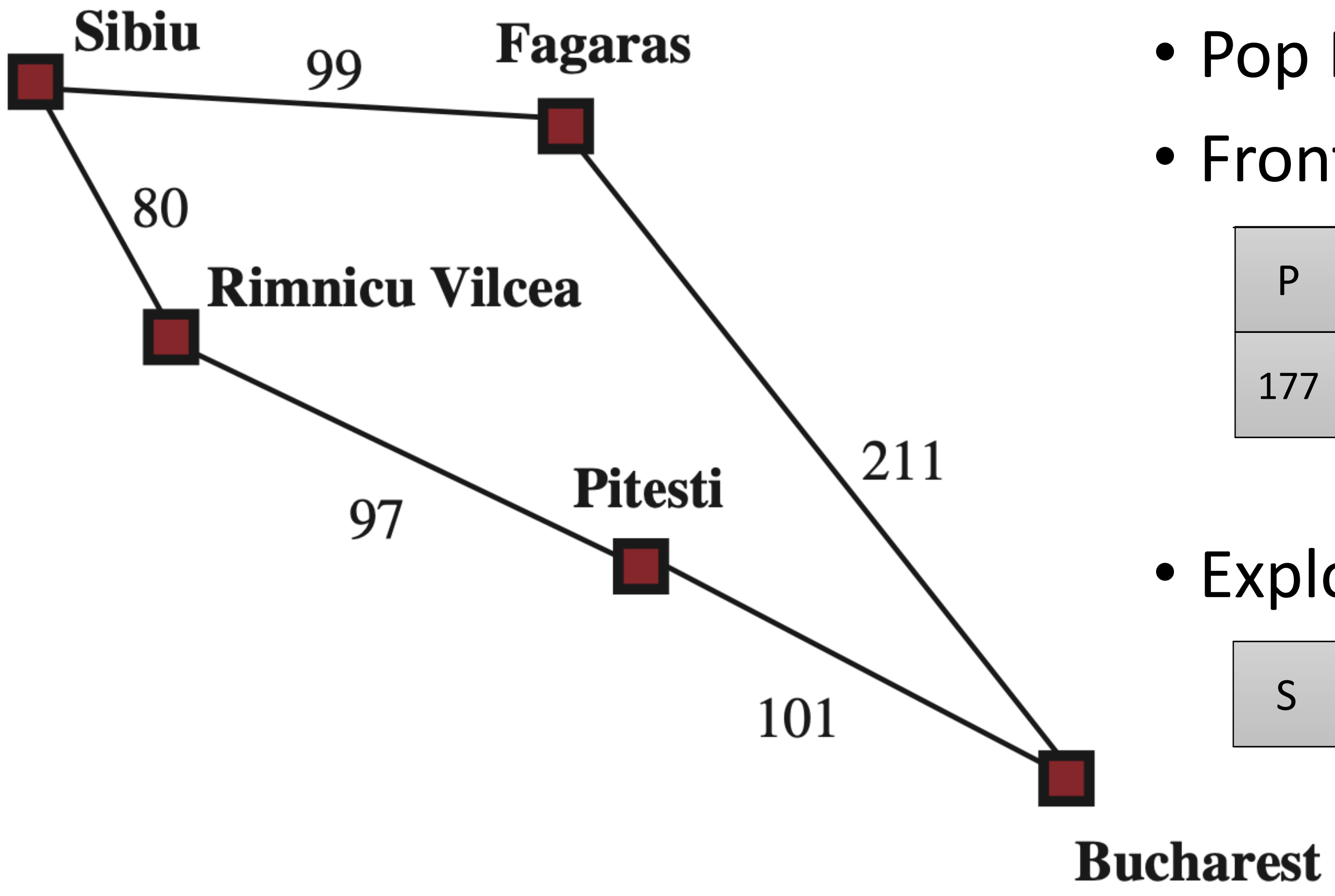
- Pop Node = R, Goal? No
- Frontier

F	P			
99	177			

- Explored

S	R			

2. UCS: Example



- Pop Node = F, Goal? No
- Frontier

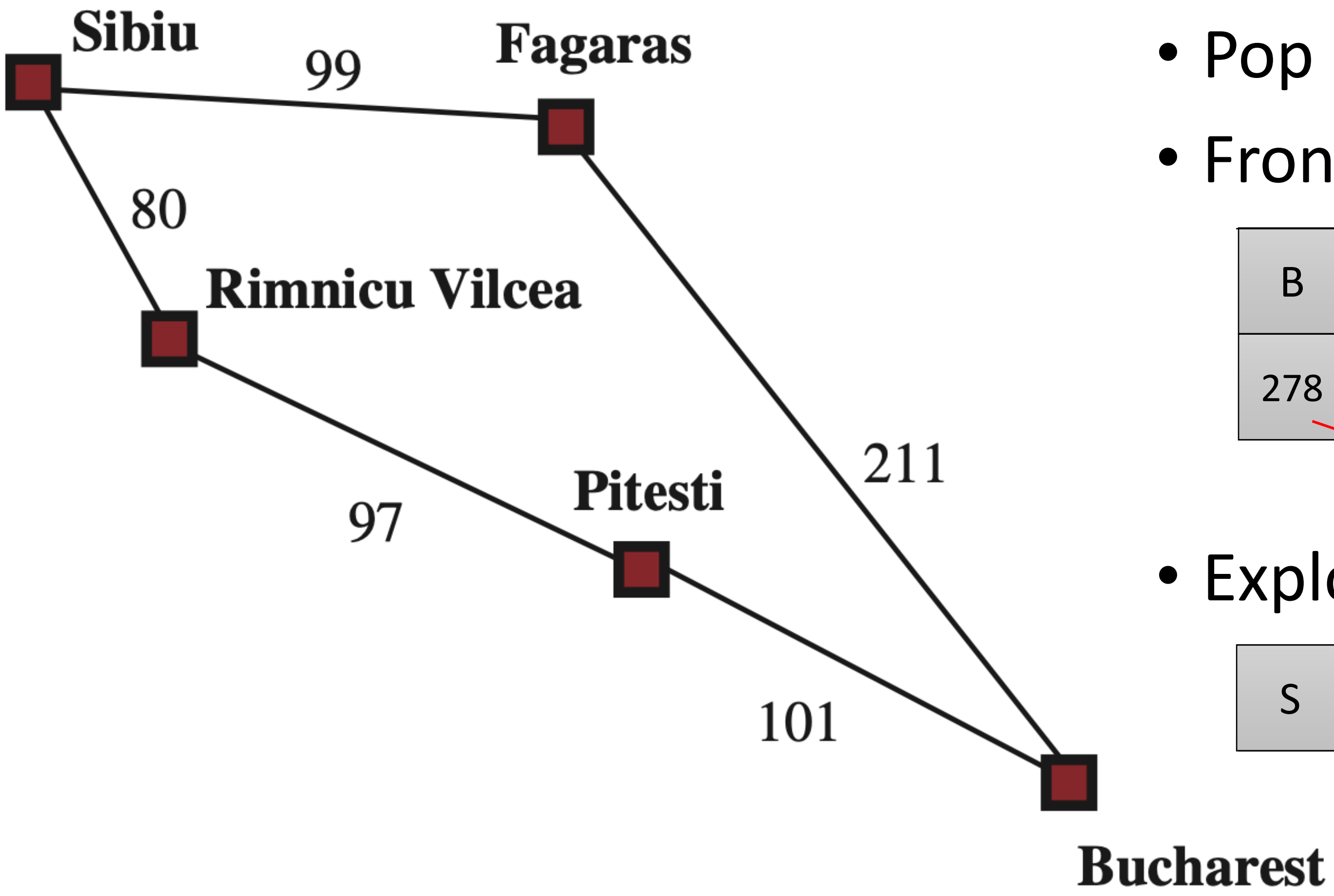
P	B			
177	310			

We added Bucharest, the goal

- Explored

S	R	F		

2. UCS: Example



- Pop Node = P, Goal? No
- Frontier

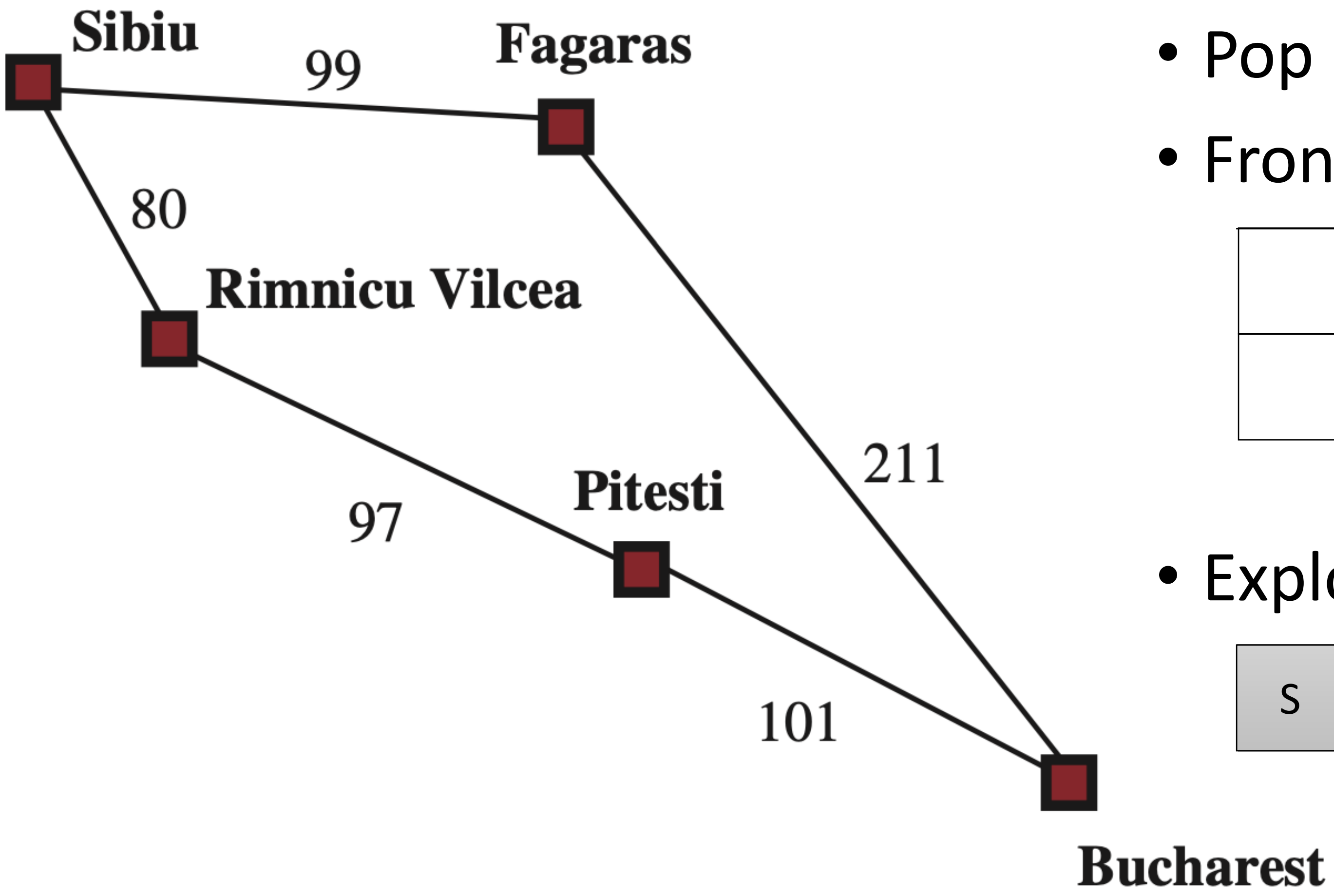
B				
278				

Cost changed

- Explored

S	R	F	P	

2. UCS: Example



- Pop Node = B, Goal? Yes
- Frontier

- Explored

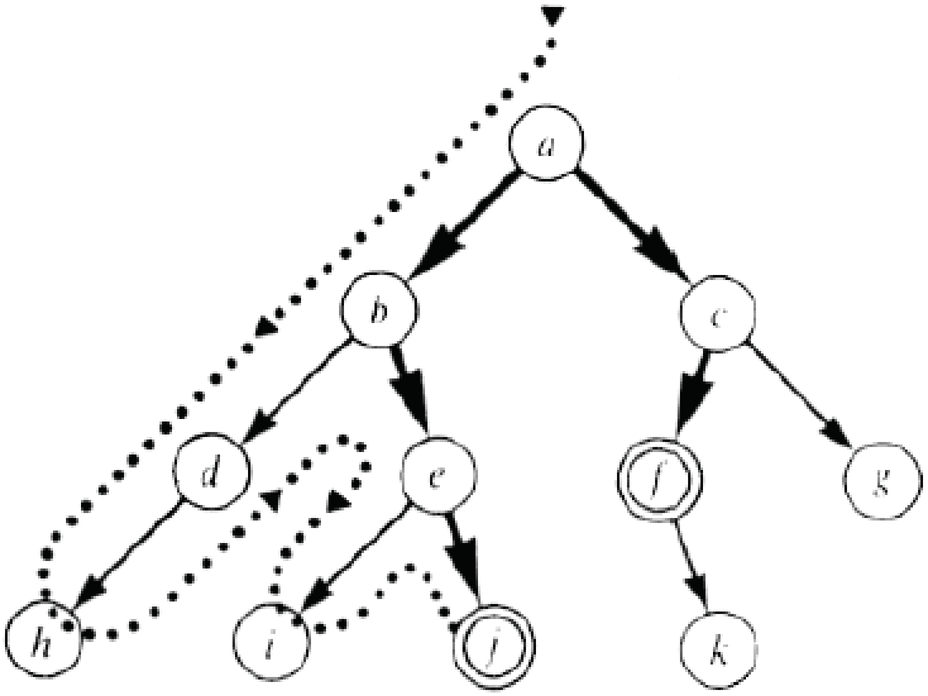
S	R	F	P	

No need to add goal once it is found

2. UCS Performance

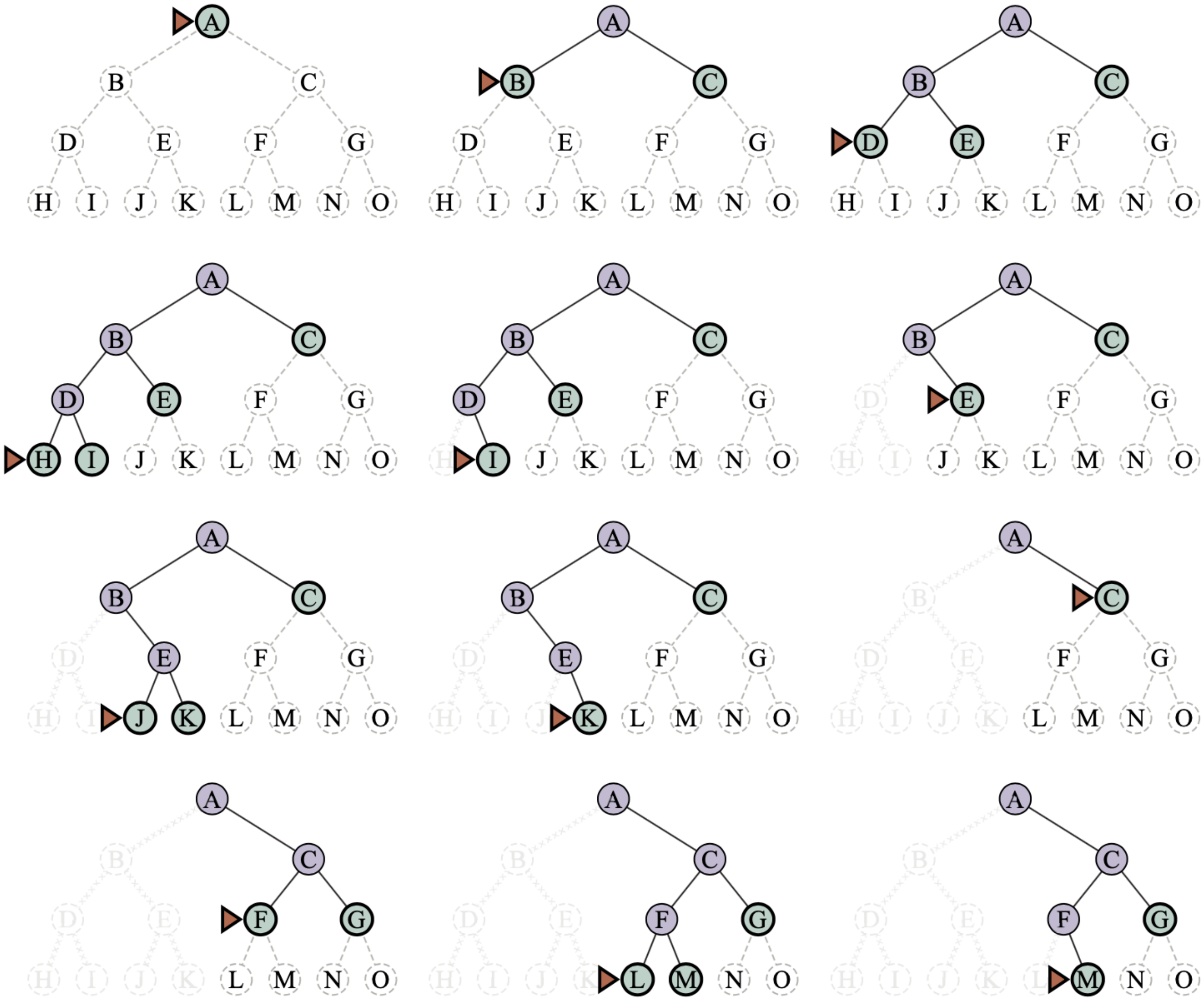
- Optimality: Yes, given that step costs are nonnegative
- Completeness: Yes, if the cost of every step exceeds some small positive constant $\varepsilon > 0$
  - UCS does not care about the number of steps
  - Gets stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions
- Time and Space Complexity:
  - C^* be the cost of the optimal solution
  - Every action costs at least ε
  - $O\left(b^{1+\lfloor C^*/\varepsilon \rfloor} \right)$, if steps are equal: $O(b^{d+1})$

3. Depth First Search (DFS)

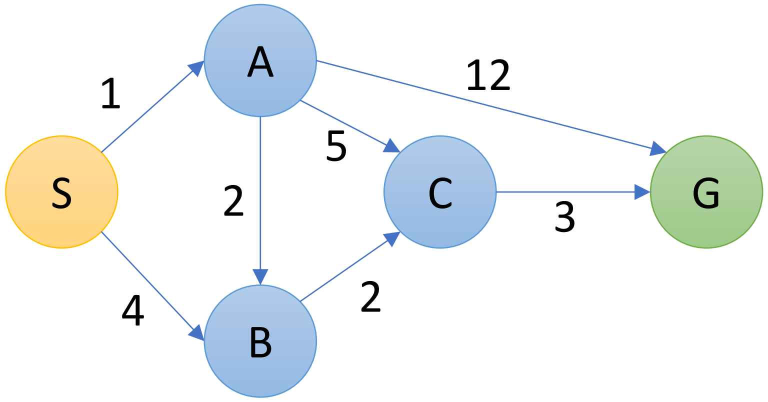


3. Depth First Search (DFS)

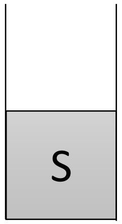
- **Main idea**: Expand node at the deepest level (breaking ties left to right).
- **Implementation**: Same as UCS, but the frontier list is a stack, Last-In-First-Out (LIFO).
- Grayed out means removed from memory



3. DFS: Example



- Initial State = S, Path-Cost = 0
- Frontier:



- Explored:

3. DFS Performance

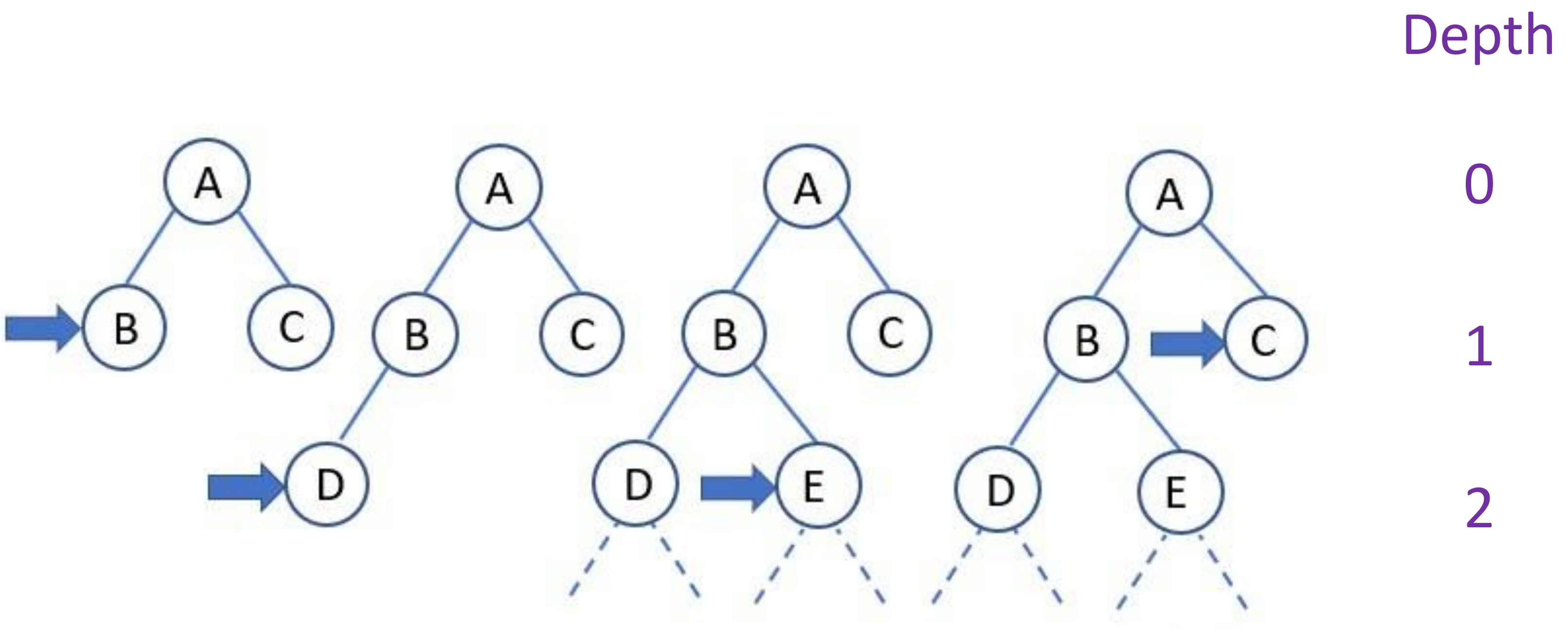
- Optimality: No
- Completeness: Guaranteed for finite space. (Tree search not complete!)
- Time Complexity: generate $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node
  - m itself can be much larger than d
- Space Complexity: $O(b * m)$
  - b branching factor
  - m maximum depth

BFS	16	10^{16}	350 years	10 exabytes

DFS: 156 KB

4. Depth-Limited Search (DLS)

- Assume Depth limit = 2



4. DLS

- It is simply DFS with a depth bound (limit)
  - Searching is not permitted beyond the depth bound
- Works well if we know what the depth of the solution is
- Termination is guaranteed
- If the solution is beneath the depth bound, the search cannot find the goal

4. DLS

```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

4. DLS Performance

- Completeness: No
- Optimality: No
- Time Complexity: $O(b^L)$, where L is the depth limit.
- Space Complexity: $O(b * L)$, where L is the depth limit.

5. Iterative Deepening Search (IDS)

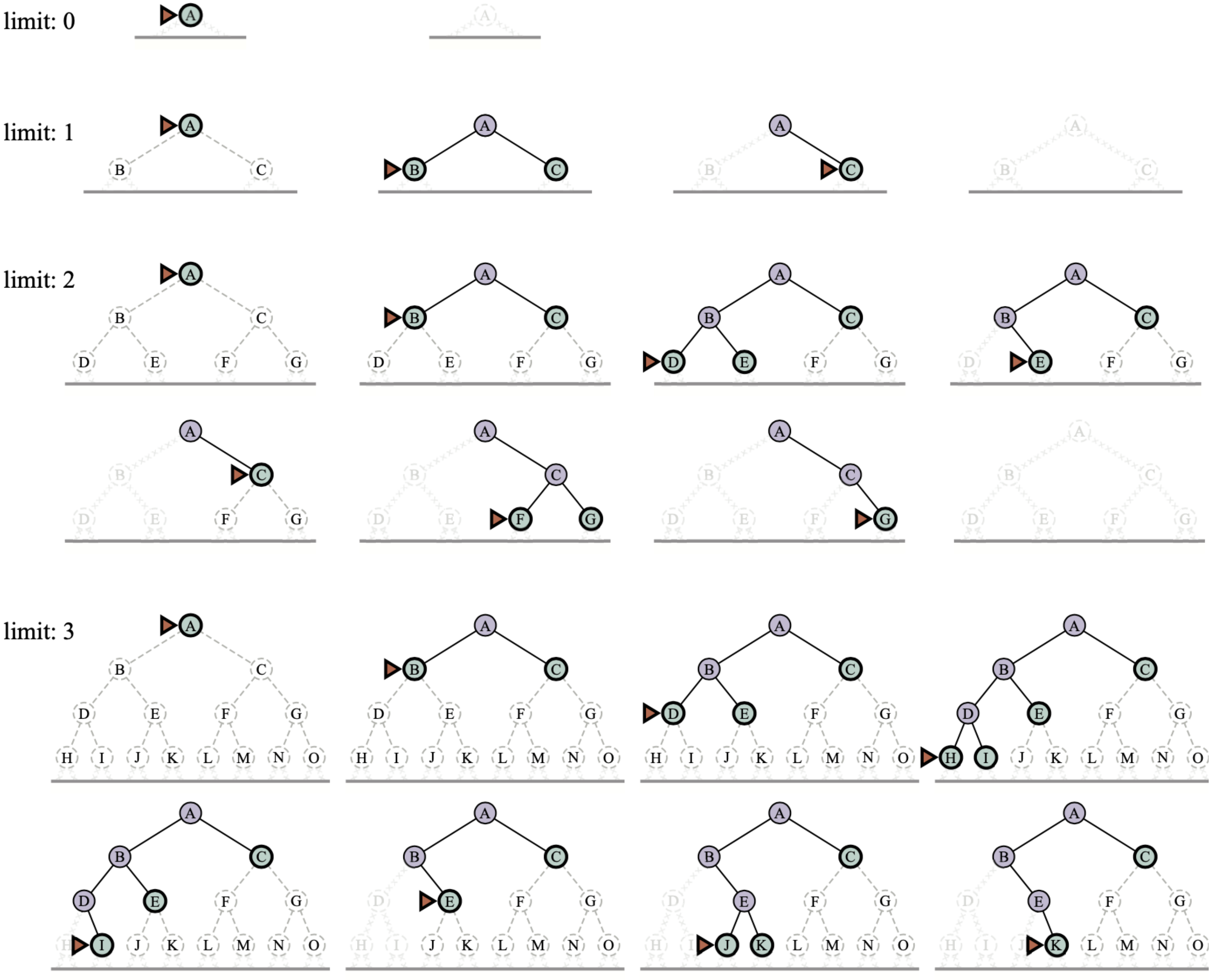
- If the depth is unknown, use Iterative deepening search (IDS)

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*
  for *depth* = 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ *cutoff* **then return** *result*

5. IDS

- **Main idea**: Expand node at depth zero, if not goal, increase level

- Grayed out means removed from memory



Students

5. IDS Performance

- IDS combines the benefits of BFS and DFS:
  - Like DFS the memory requirements are very modest $O(b*d)$
  - Like BFS, it is complete when the branching factor is finite
- The total number of generated nodes:

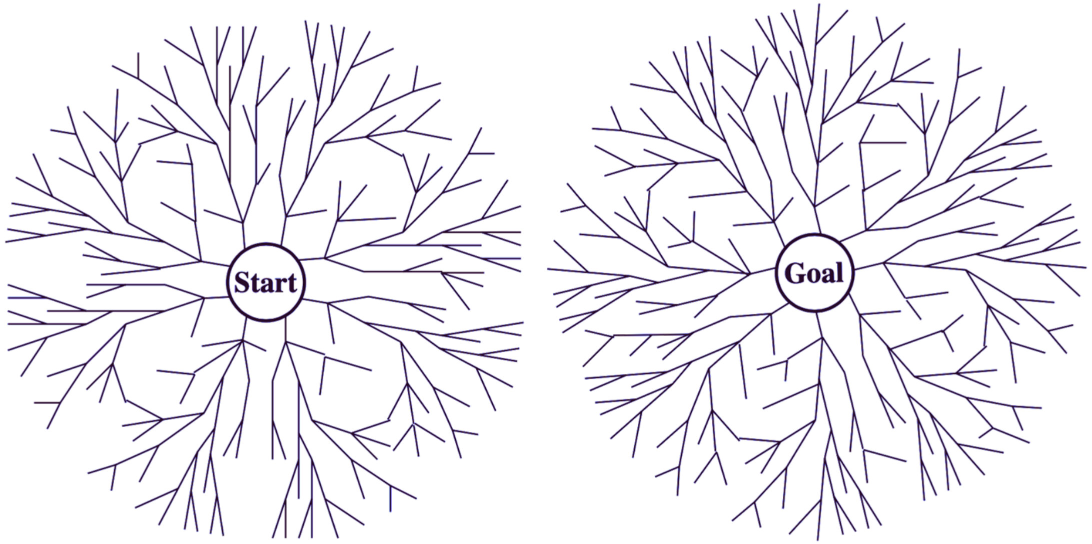
$$N(\mathrm{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

- In general, IDS is the preferred uninformed search method, when search space is large, and depth of solution is unknown
- Completeness: Yes (if b is finite).
- Optimality: Yes when the path cost is a non-decreasing function of depth.
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b*d)$

Students

6. Bidirectional Search (BDS)

- **Main idea**: Start searching from both the initial state and the goal state (if applicable), meet in the middle (the frontiers intersect)
- Why? Because time (and space) complexity is $b^{\frac{d}{2}} + b^{\frac{d}{2}}$ is much less than b^d
  - Area of the two small circles is less than the area of one big circle



6. Bidirectional Search (BDS)

- Use BFS or UCS for search
- Can not be used in implicit goals
- Difficult when the actions are not reversible
- Requires a method for computing predecessors
  - Difficult when e.g. goal is no queen attacks another queen
  - Easy in finding a route from a map

6. BDS Performance 

- Completeness: Yes, if b is finite and use BFS or UCS in both directions
- Optimality: Yes, if UCS is used in both directions or the step costs are all identical and BFS is used in both directions
- Time Complexity: $O(b^{\frac{d}{2}})$
- Space Complexity: $O(b^{\frac{d}{2}})$

Students

Summary

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^{c}	Yes	No	No	Yes^{c}	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Tutorial

Blind Search Algorithms

Tree Search:

BFS, DFS, DLS, IDS

Basic Search Algorithms

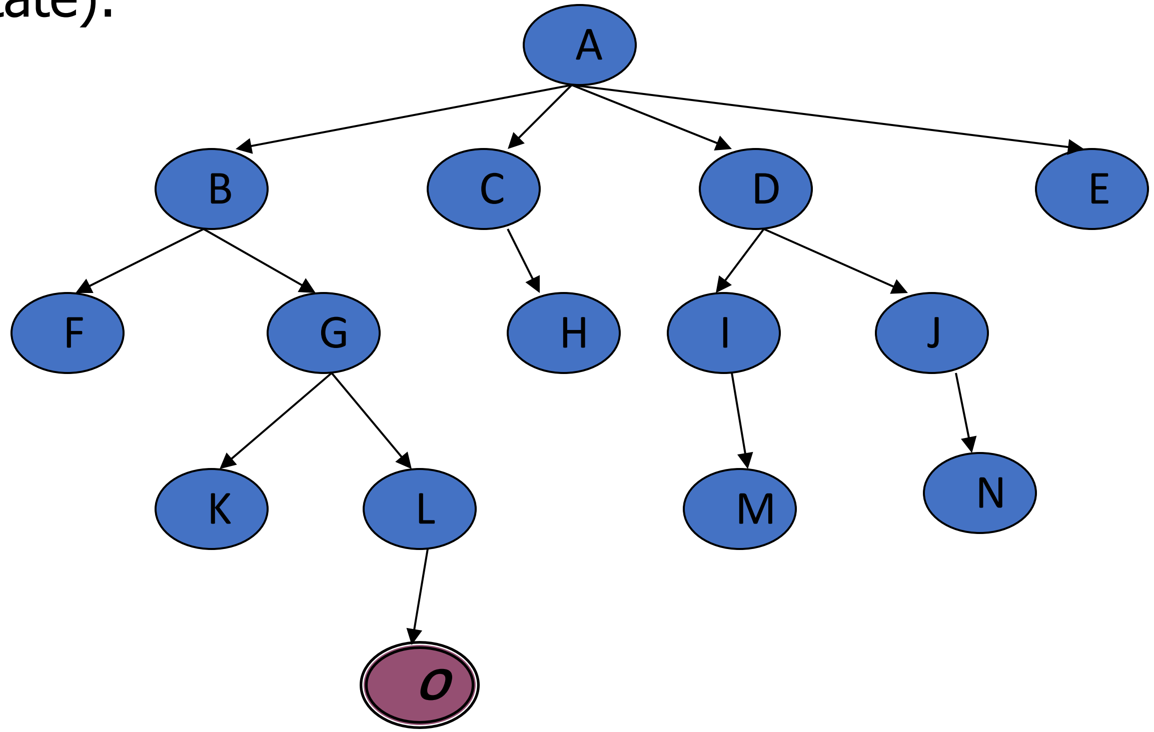
Breadth First Search

BFS

Breadth First Search

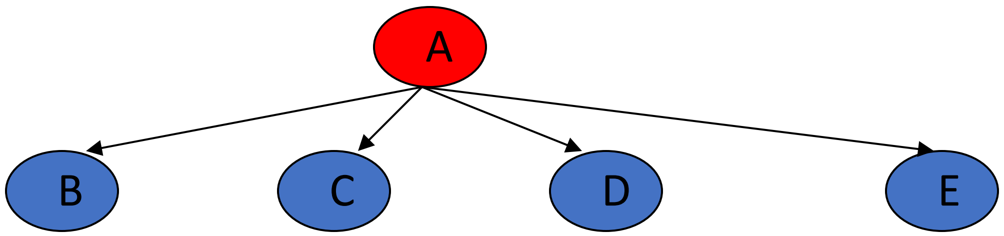
- Application1:

  Given the following state space (tree search), give the sequence of visited nodes when using BFS (assume that the node ***O*** is the goal state):



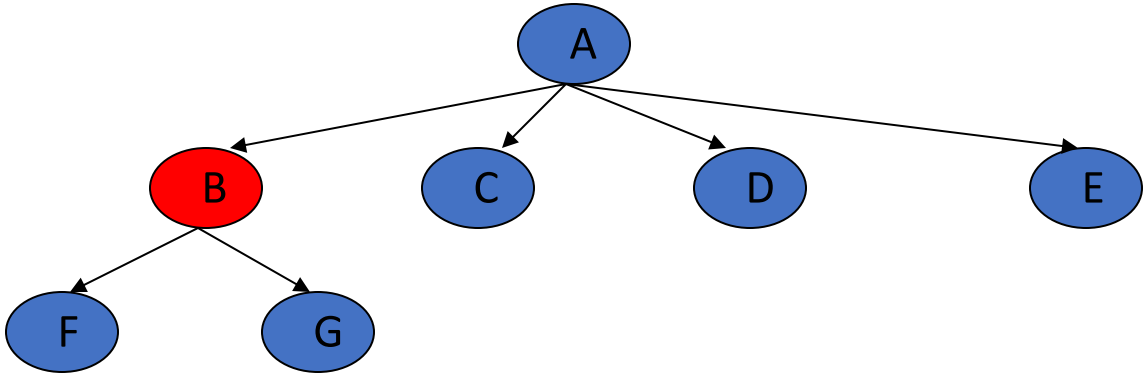
Breadth First Search

- A,



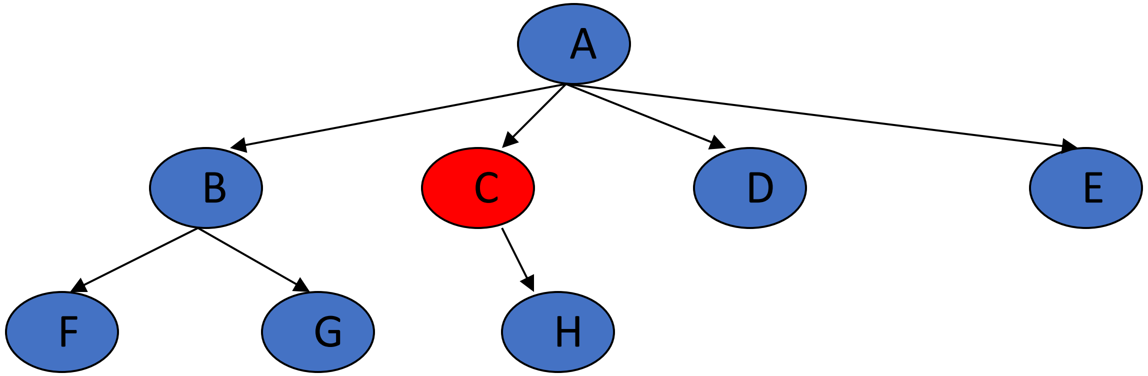
Breadth First Search

- A,
- B,



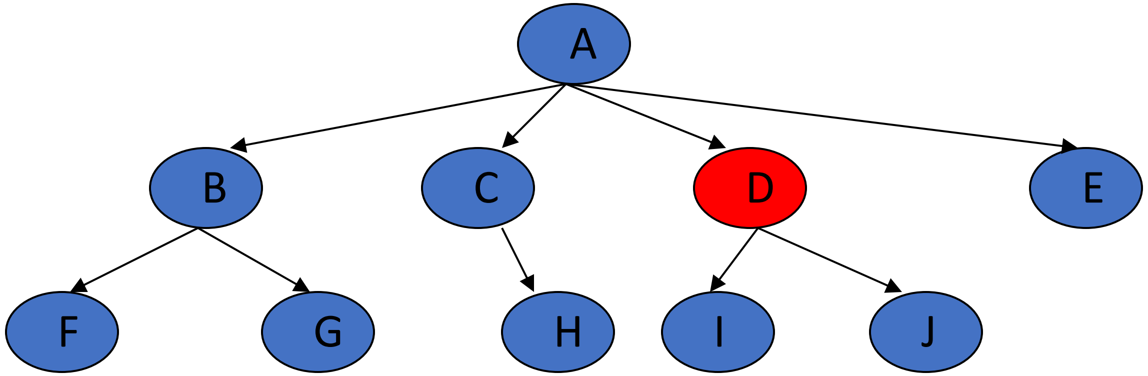
Breadth First Search

- A,
- B,C



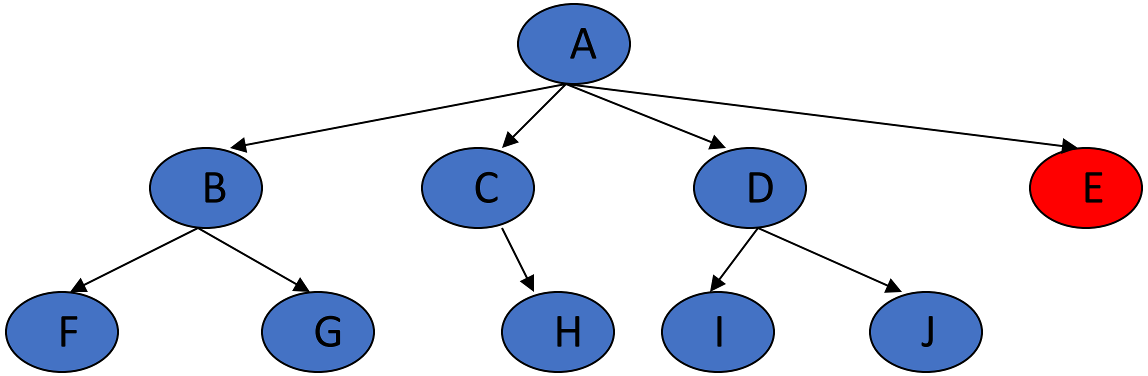
Breadth First Search

- A,
- B,C,D



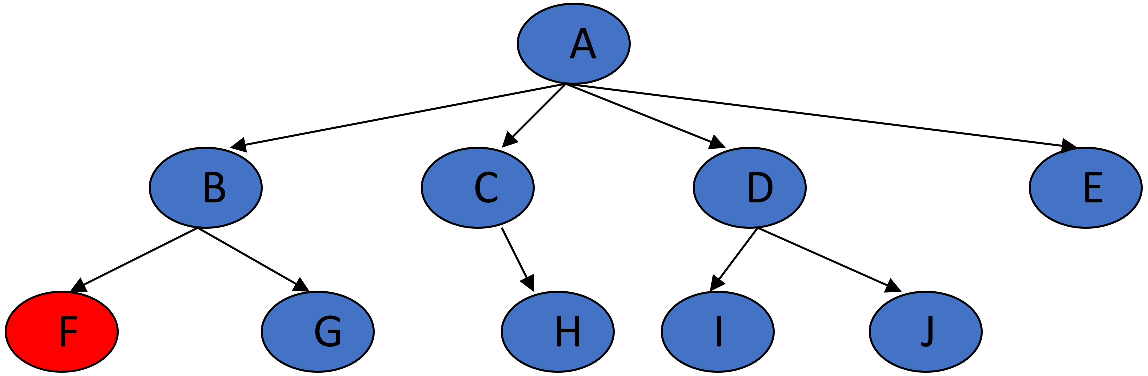
Breadth First Search

- A,
- B,C,D,E



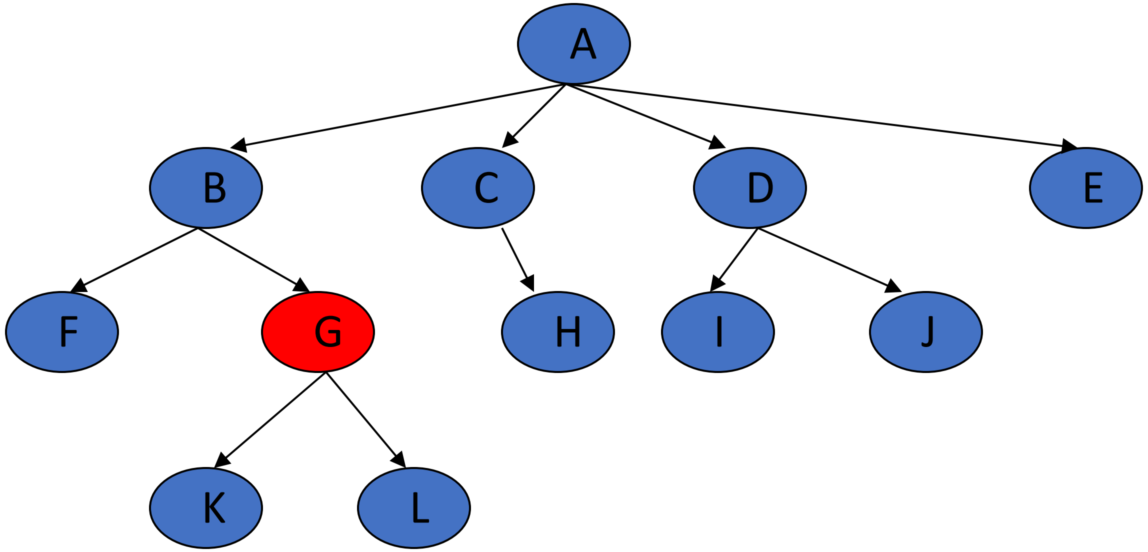
Breadth First Search

- A,
- B,C,D,E,
- F,



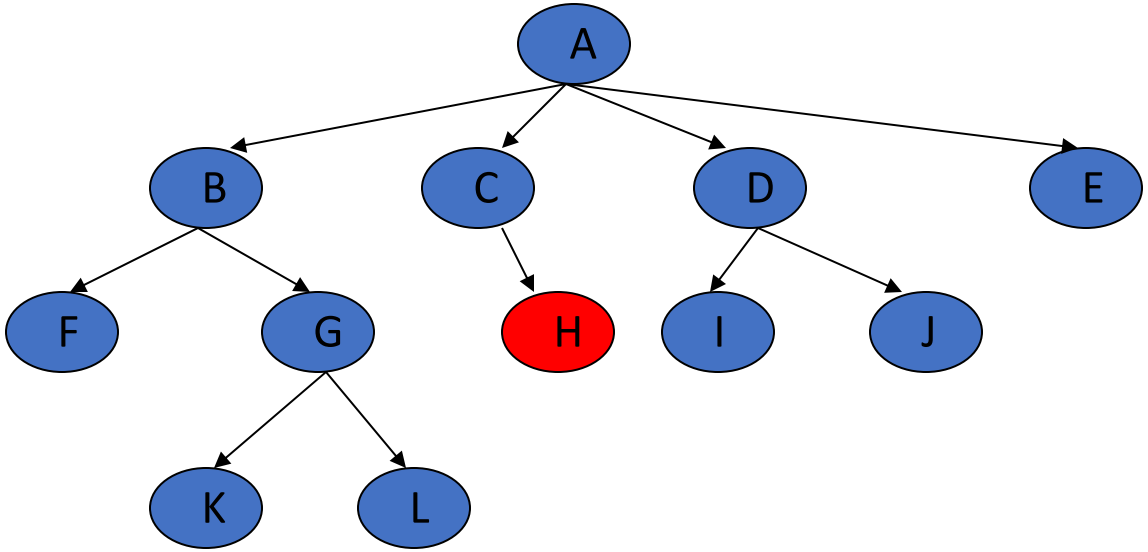
Breadth First Search

- A,
- B,C,D,E,
- F,G



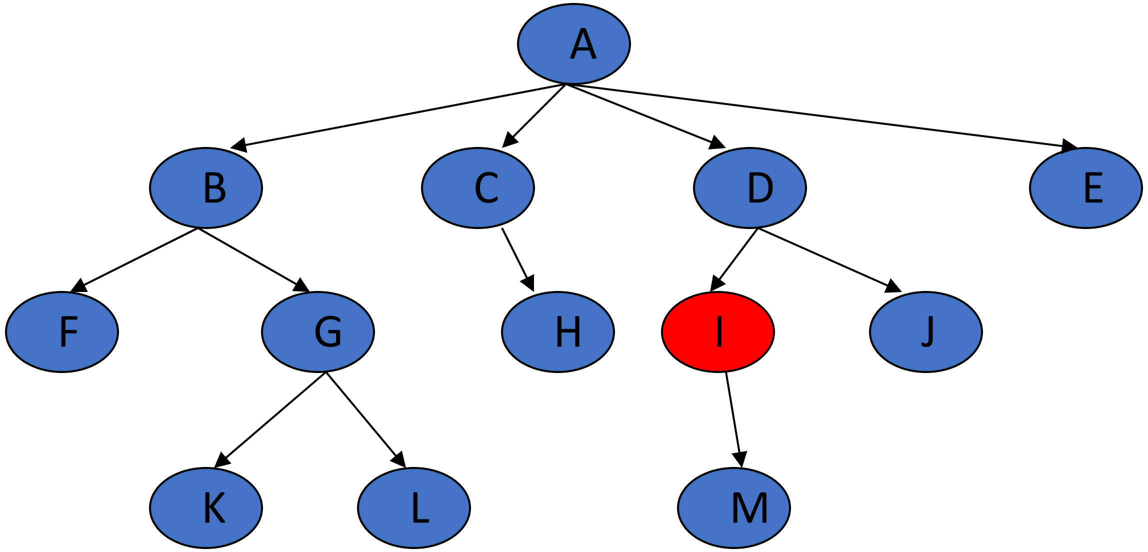
Breadth First Search

- A,
- B,C,D,E,
- F,G,H



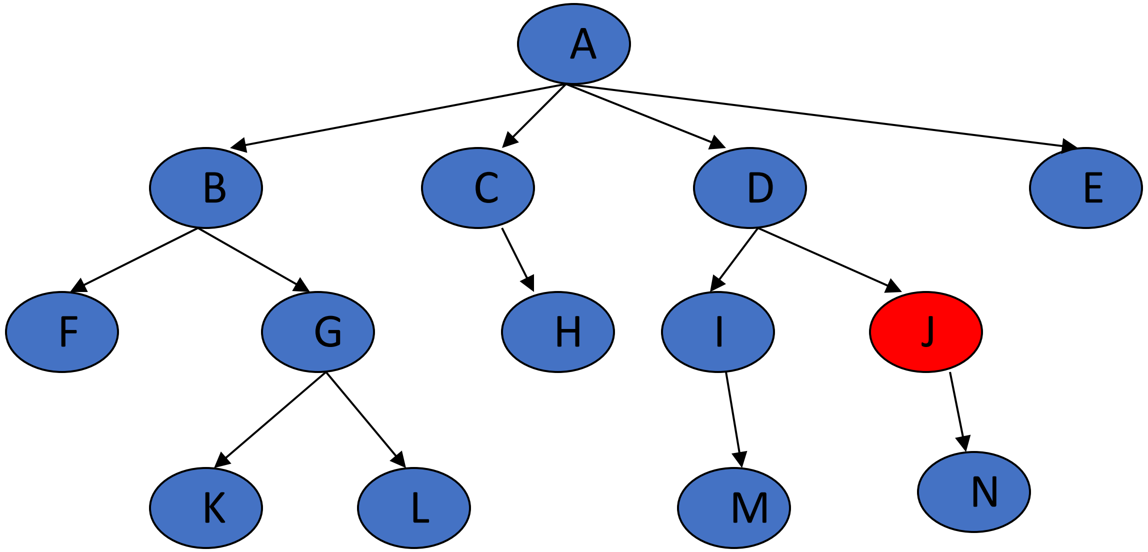
Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I



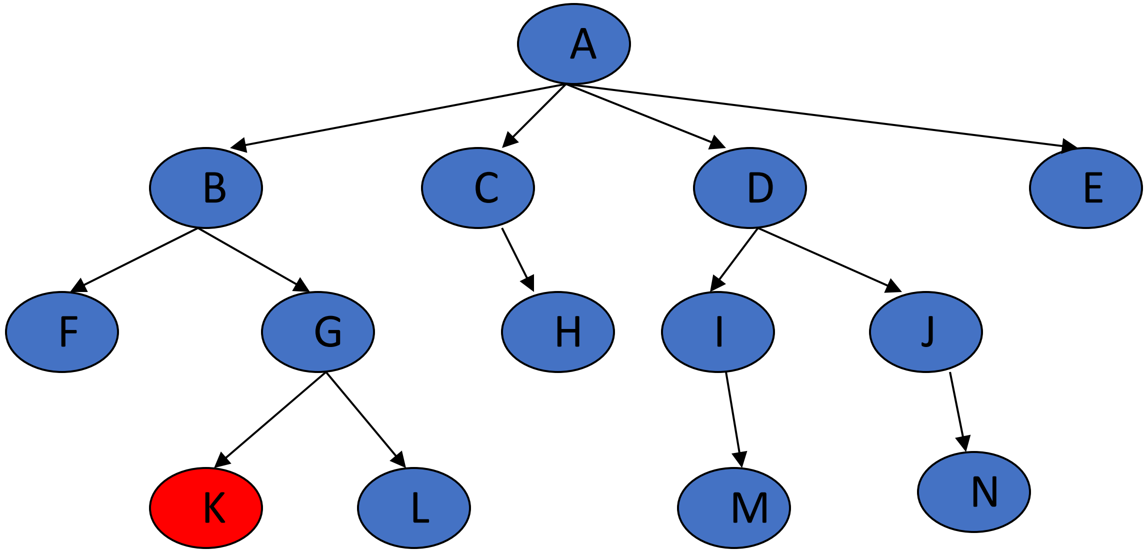
Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,



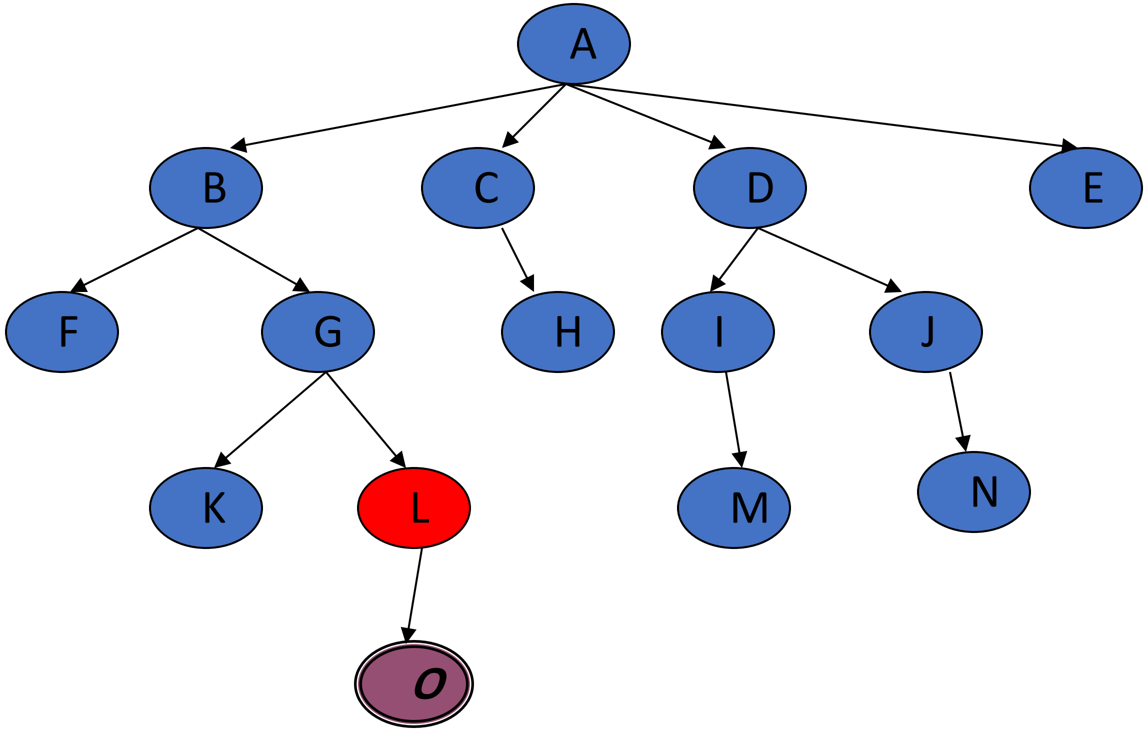
Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,



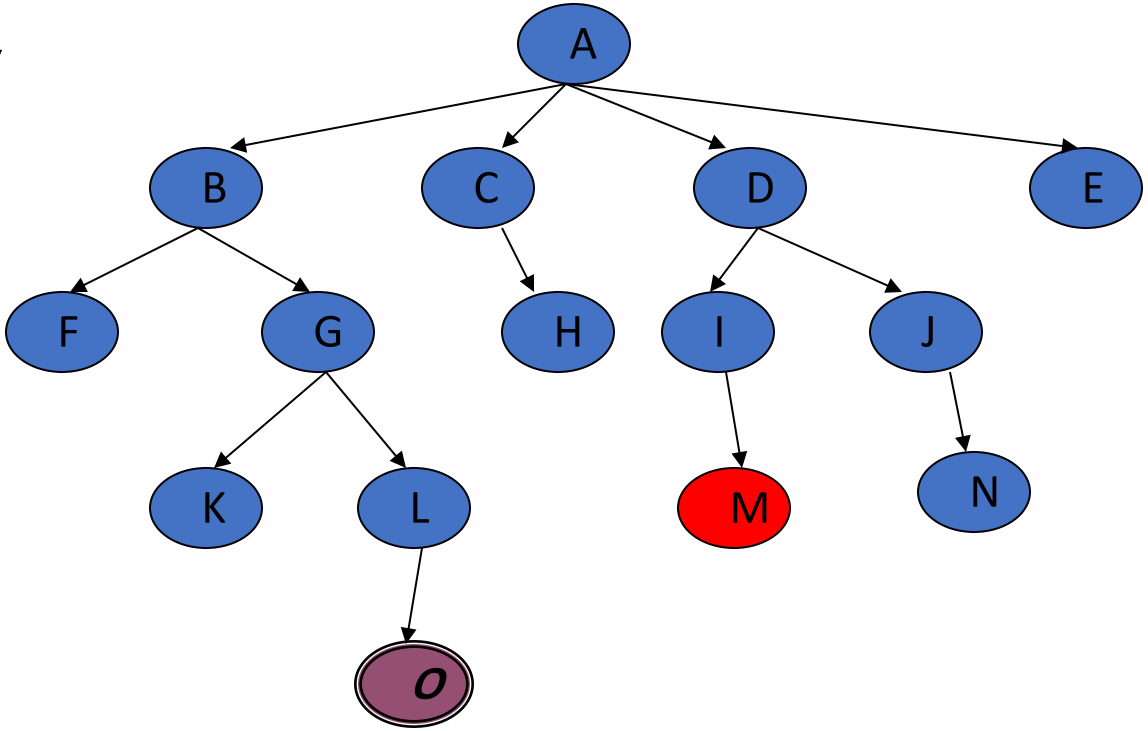
Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L



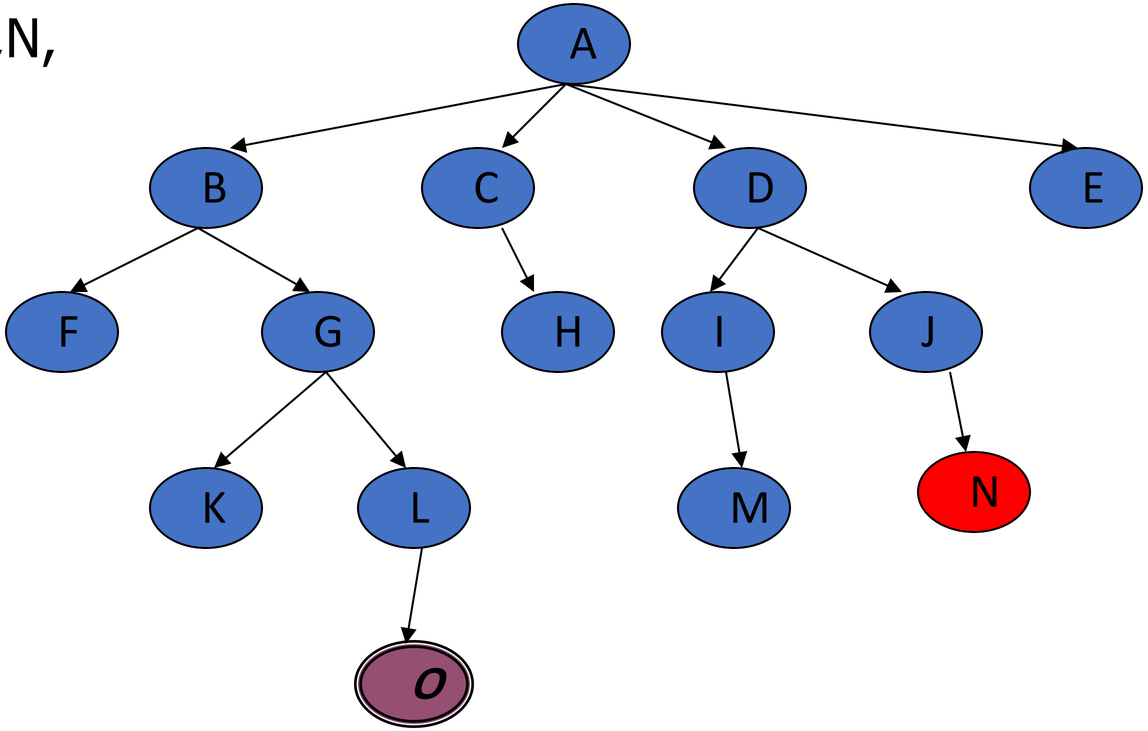
Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L, M,



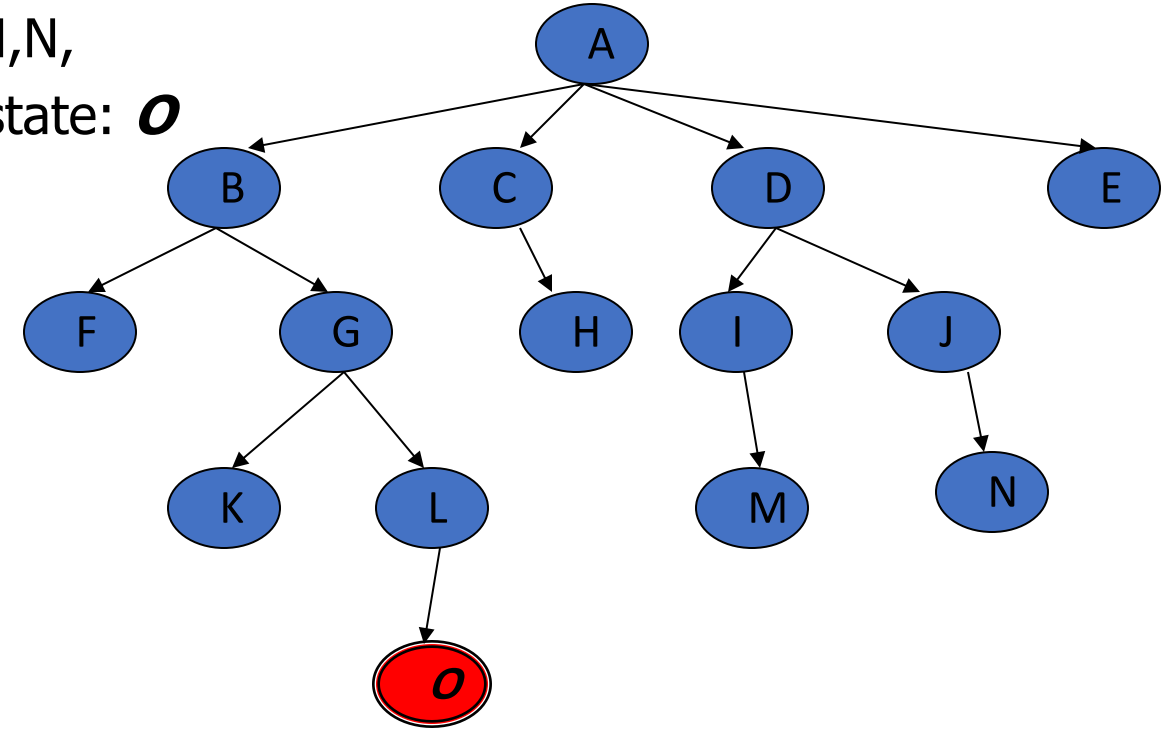
Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L, M,N,



Breadth First Search

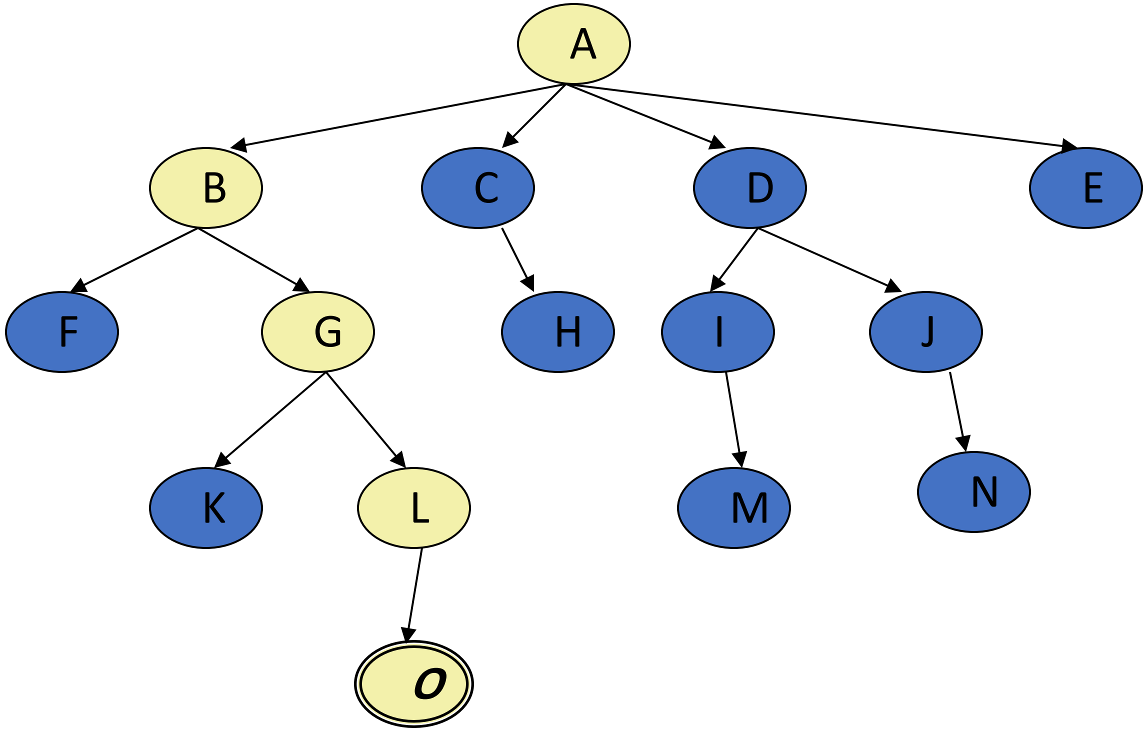
- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L, M,N,
- Goal state: ***O***



Breadth First Search

- The returned solution is the sequence of operators in the path:

  A, B, G, L, O



Basic Search Algorithms

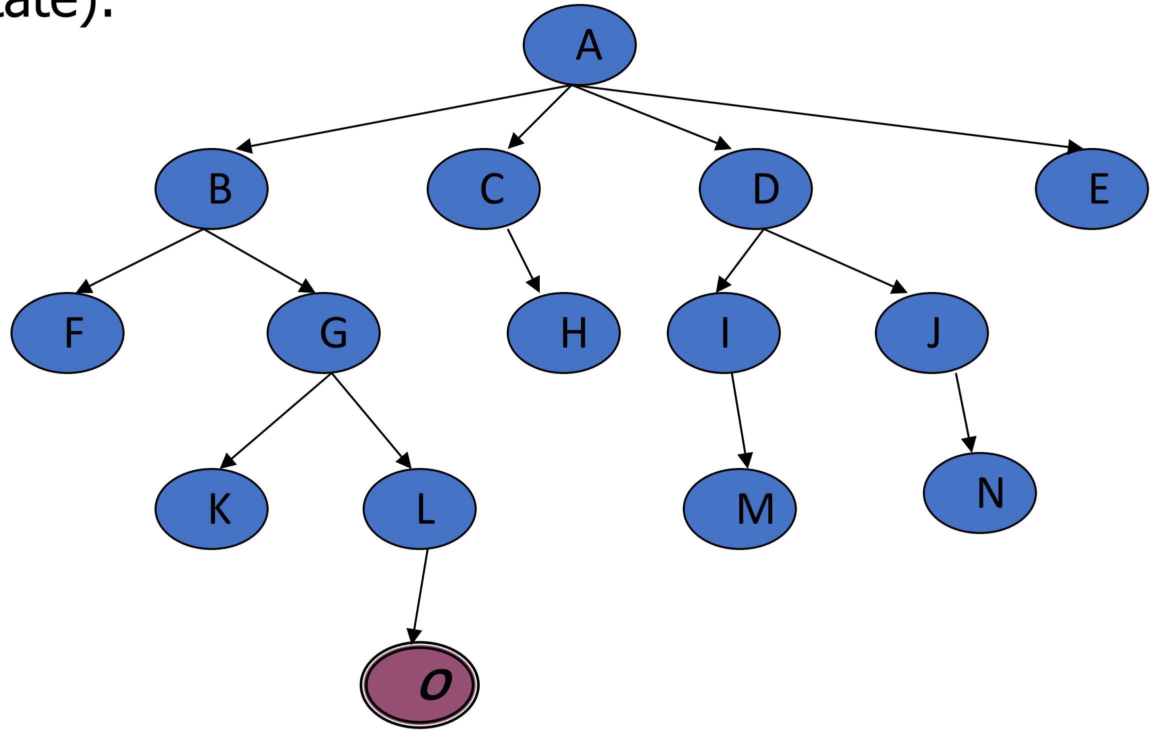
Depth First Search

DFS

Depth First Search (DFS)

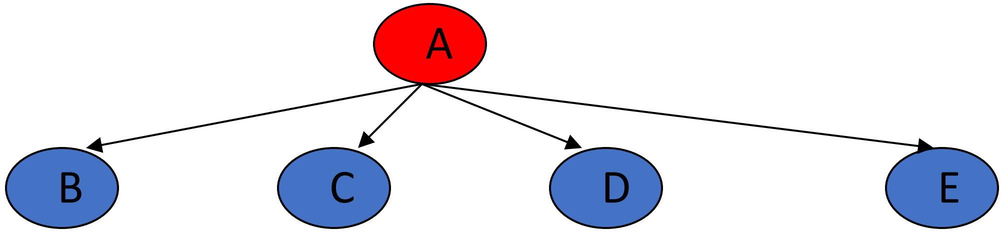
- Application2:

  Given the following state space (tree search), give the sequence of visited nodes when using DFS (assume that the node ***O*** is the goal state):



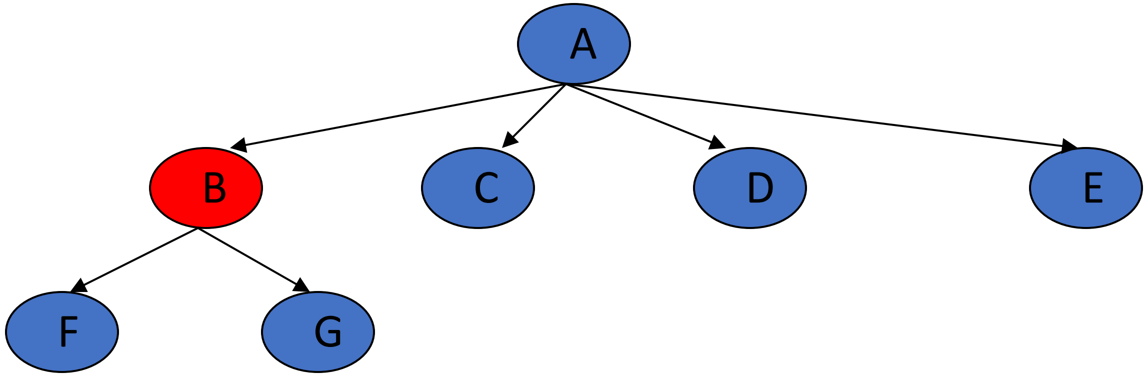
Depth First Search

- A,



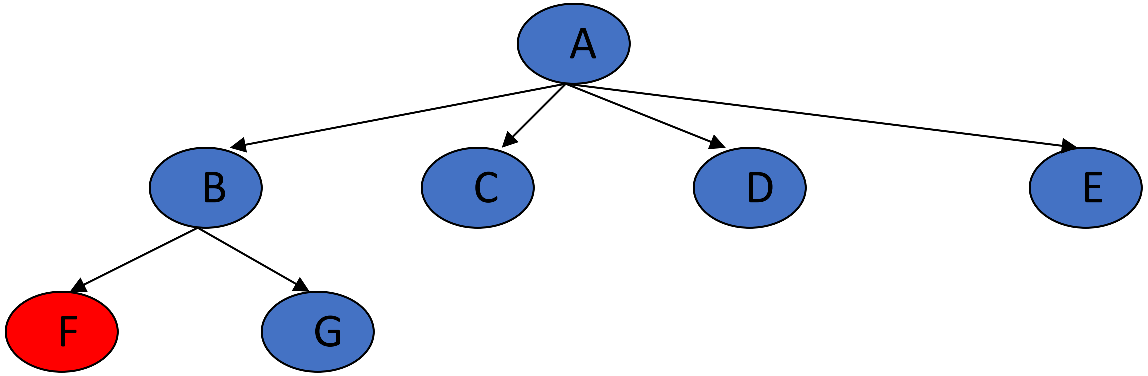
Depth First Search

- A,B,



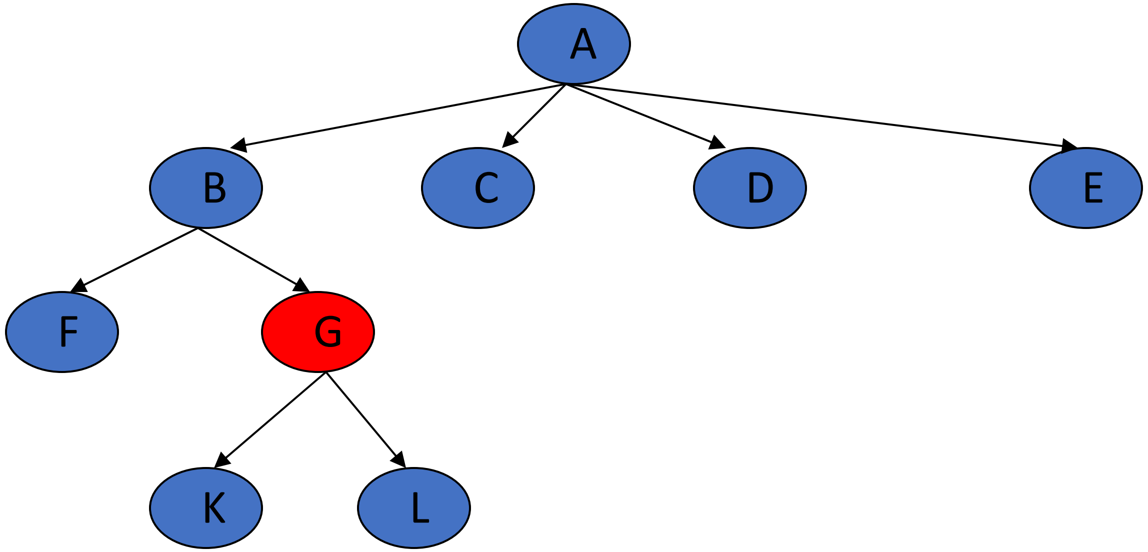
Depth First Search

- A,B,F,



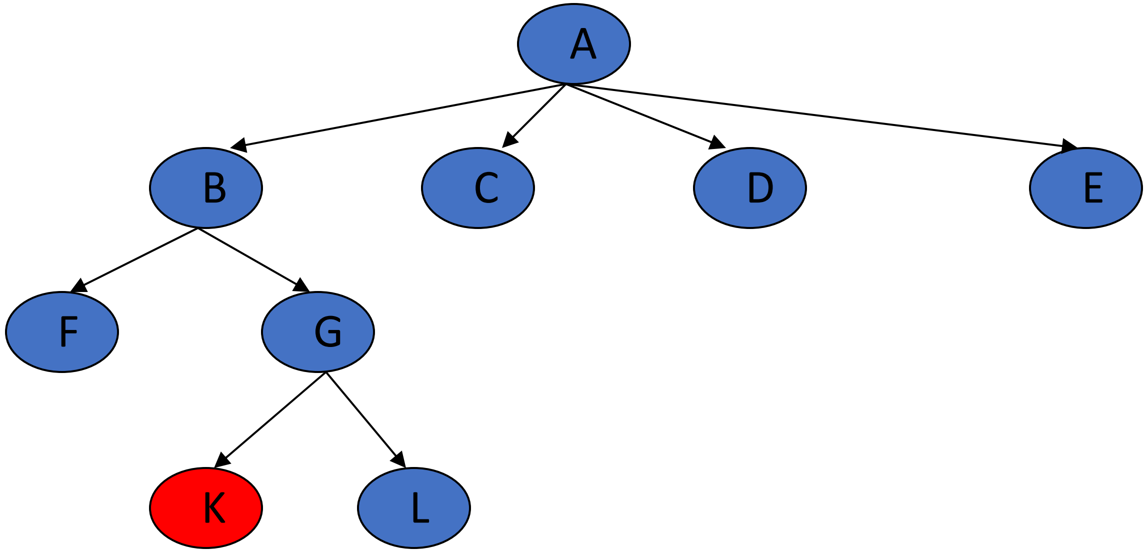
Depth First Search

- A,B,F,
- G,



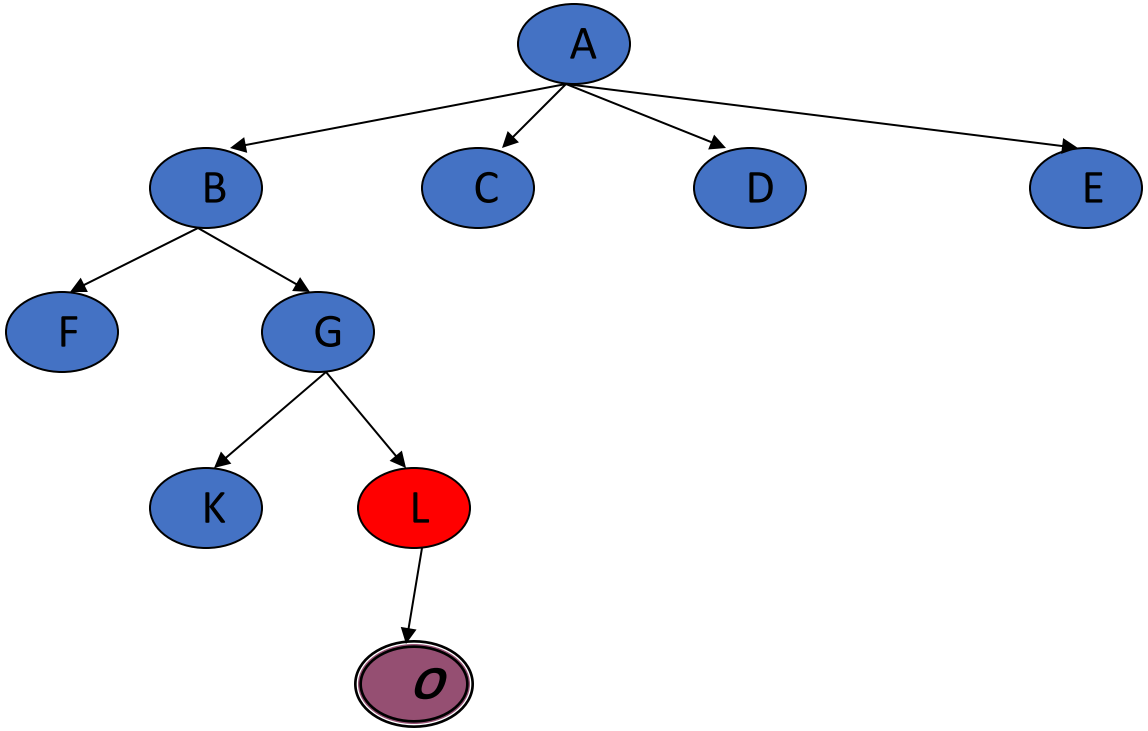
Depth First Search

- A,B,F,
- G,K,



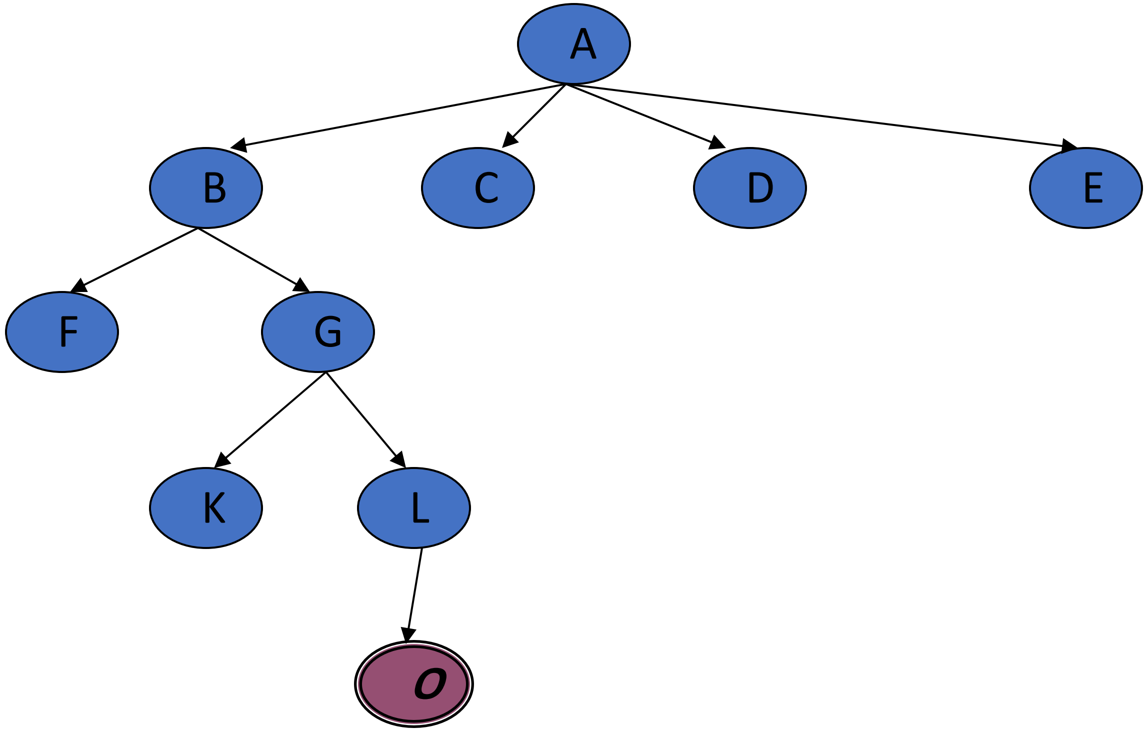
Depth First Search

- A,B,F,
- G,K,
- L,



Depth First Search

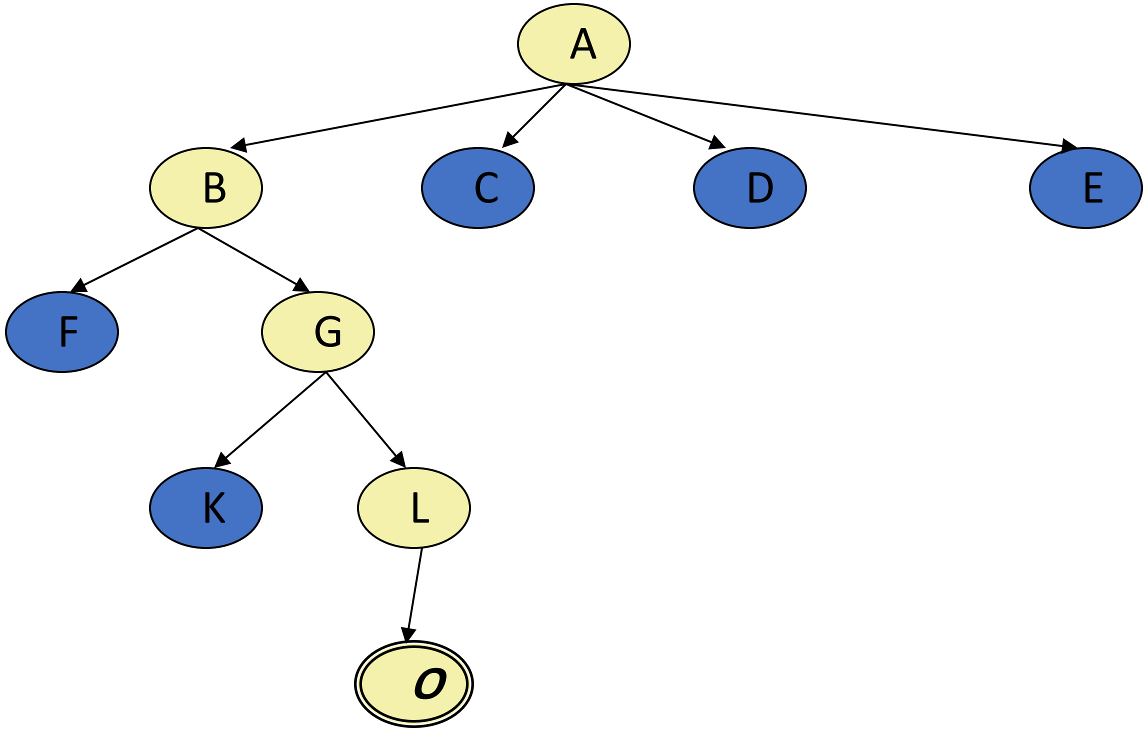
- A,B,F,
- G,K,
- L, *O: Goal State*



Depth First Search

The returned solution is the sequence of operators in the path:

A, B, G, L, O



Basic Search Algorithms

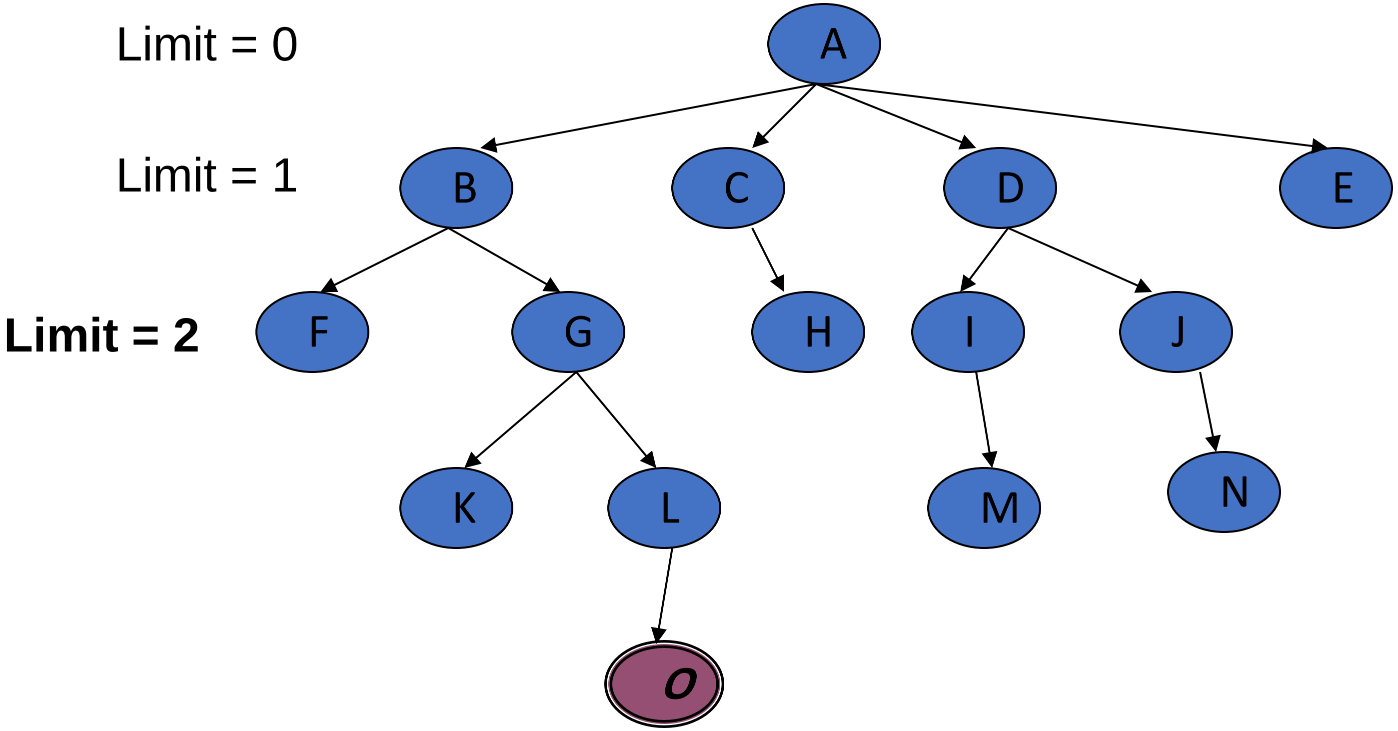
Depth-Limited Search

DLS

Depth-Limited Search (DLS)

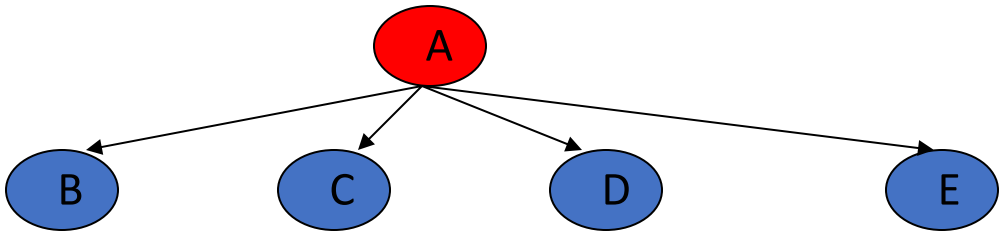
- Application3:

  Given the following state space (tree search), give the sequence of visited nodes when using DLS (Limit = 2):



Depth-Limited Search (DLS)

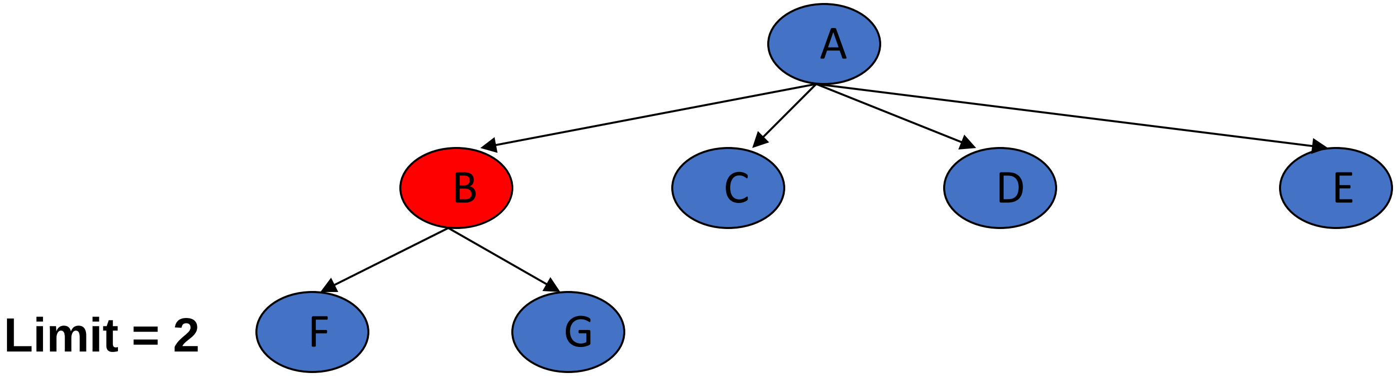
- A,



Limit = 2

Depth-Limited Search (DLS)

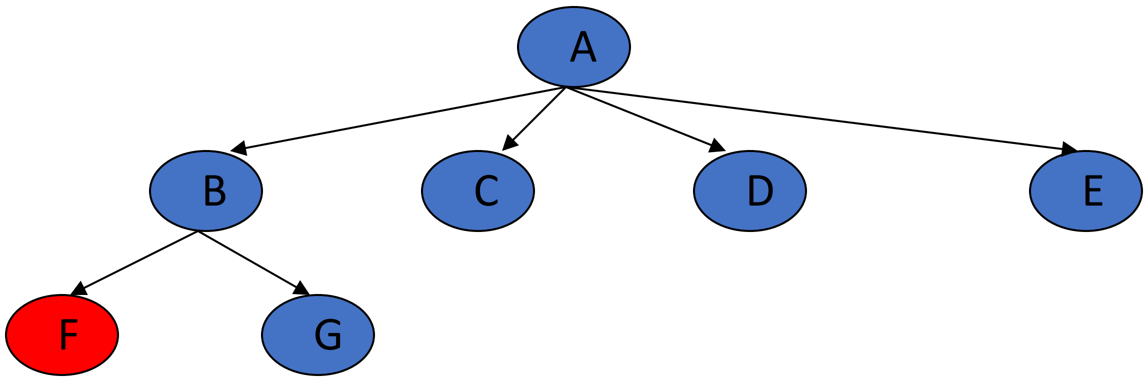
- A,B,



Depth-Limited Search (DLS)

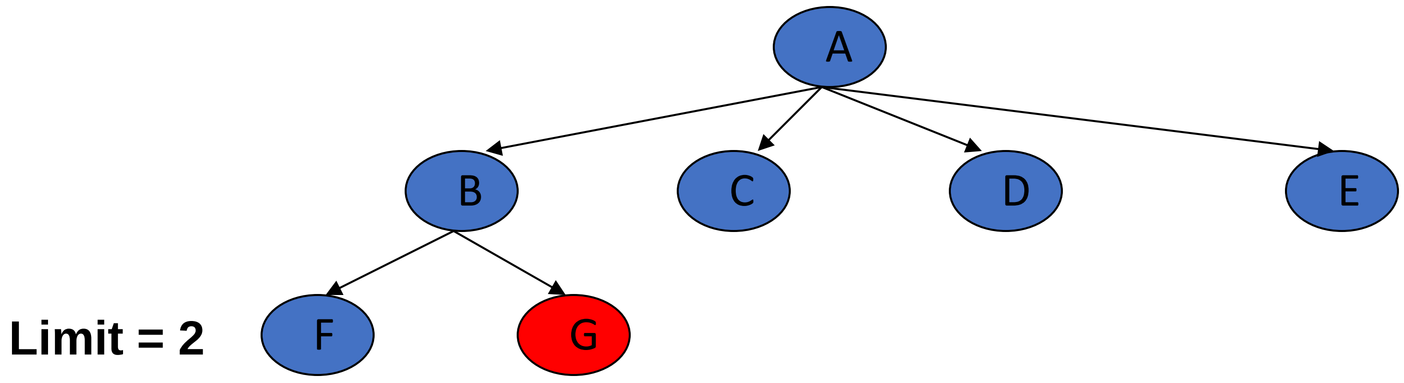
- A,B,F,

Limit = 2



Depth-Limited Search (DLS)

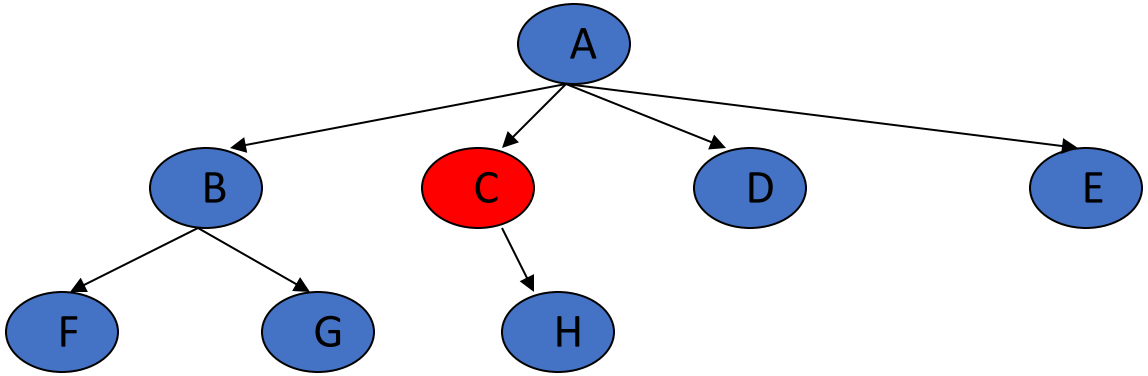
- A,B,F,
- G,



Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,

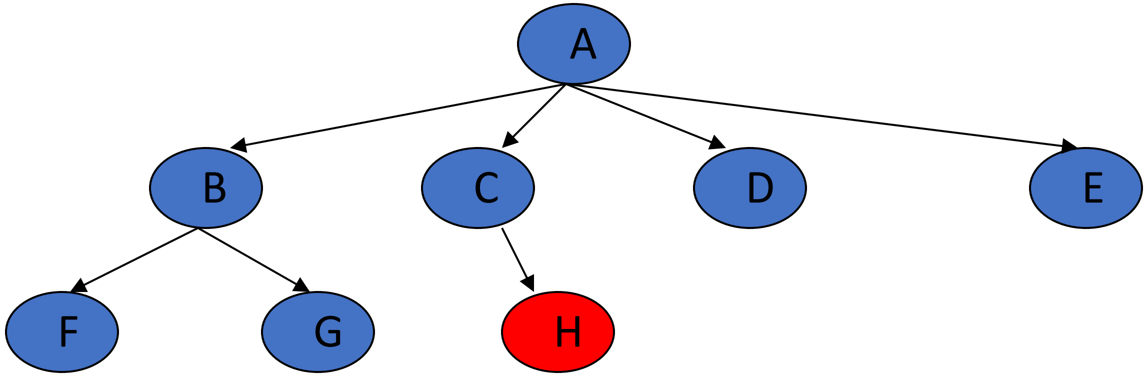
Limit = 2



Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,

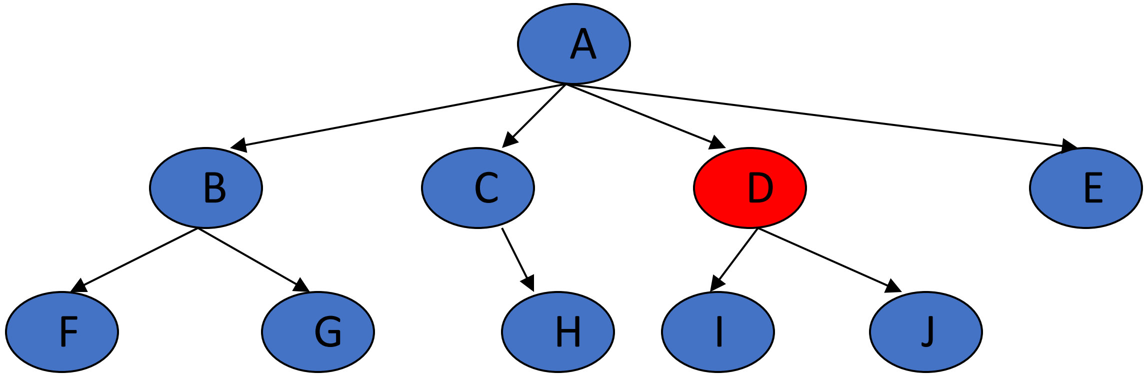
Limit = 2



Depth-Limited Search (DLS)

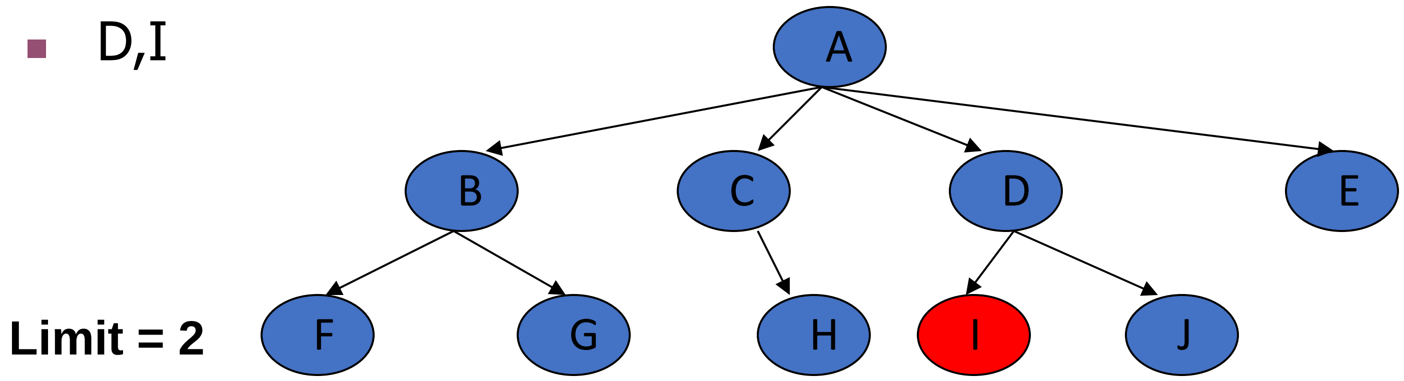
- A,B,F,
- G,
- C,H,
- D,

Limit = 2



Depth-Limited Search (DLS)

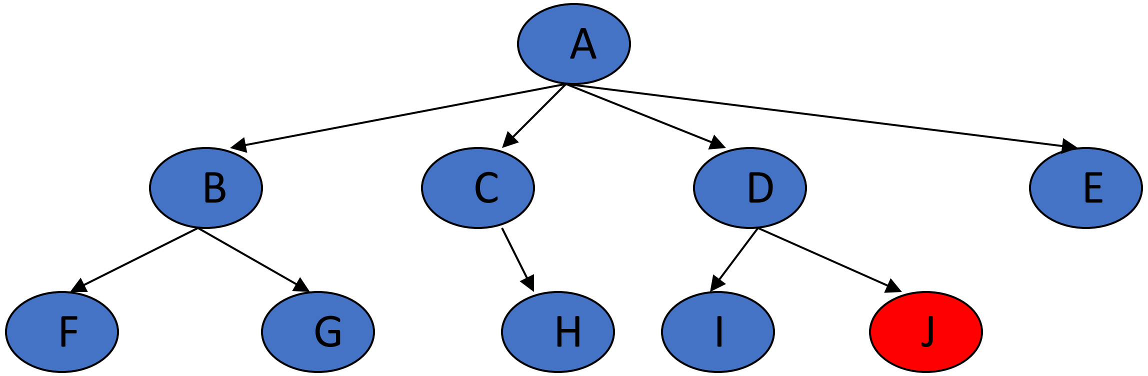
- A,B,F,
- G,
- C,H,
- D,I



Depth-Limited Search (DLS)

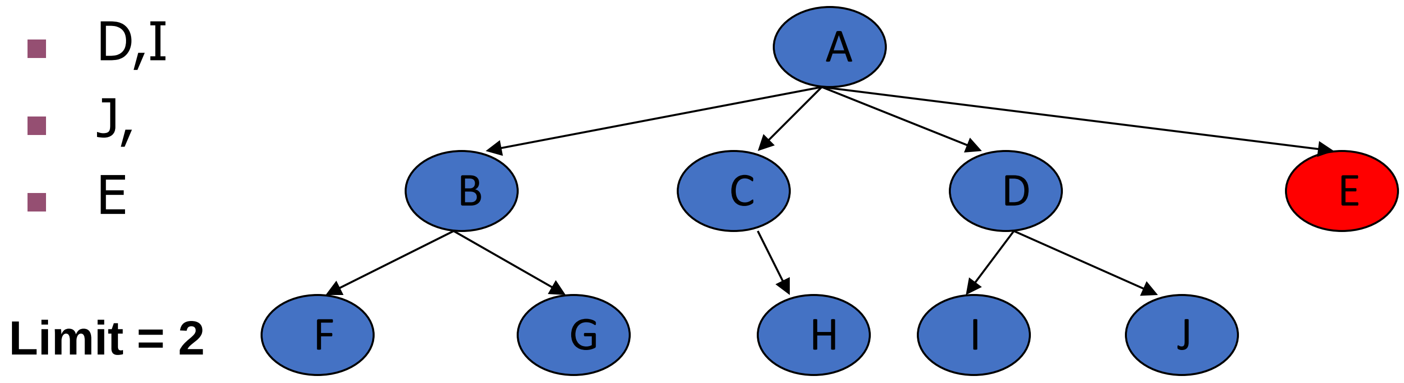
- A,B,F,
- G,
- C,H,
- D,I
- J,

Limit = 2



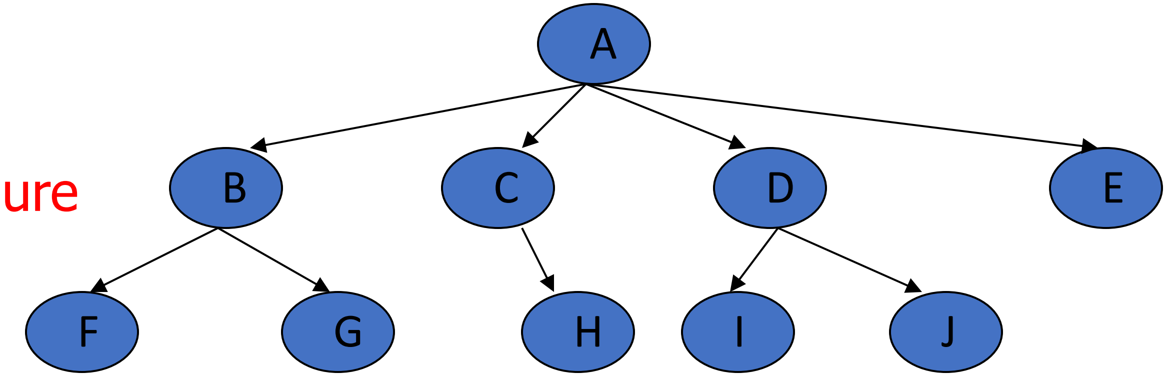
Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,
- D,I
- J,
- E



Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,
- D,I
- J,
- E, Failure

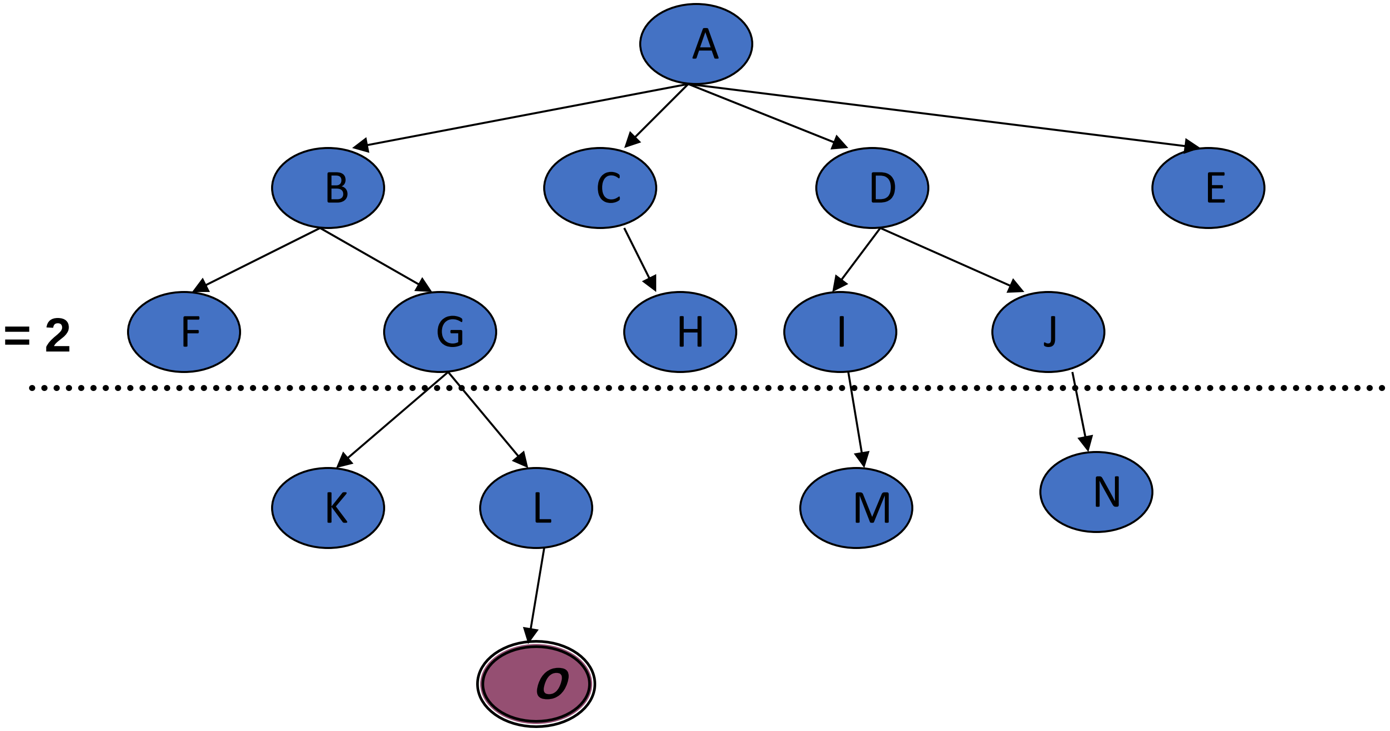


Limit = 2

Depth-Limited Search (DLS)

- DLS algorithm returns Failure (no solution)
- The reason is that the goal is beyond the limit (Limit =2): the goal depth is (d=4)

Limit = 2



Basic Search Algorithms

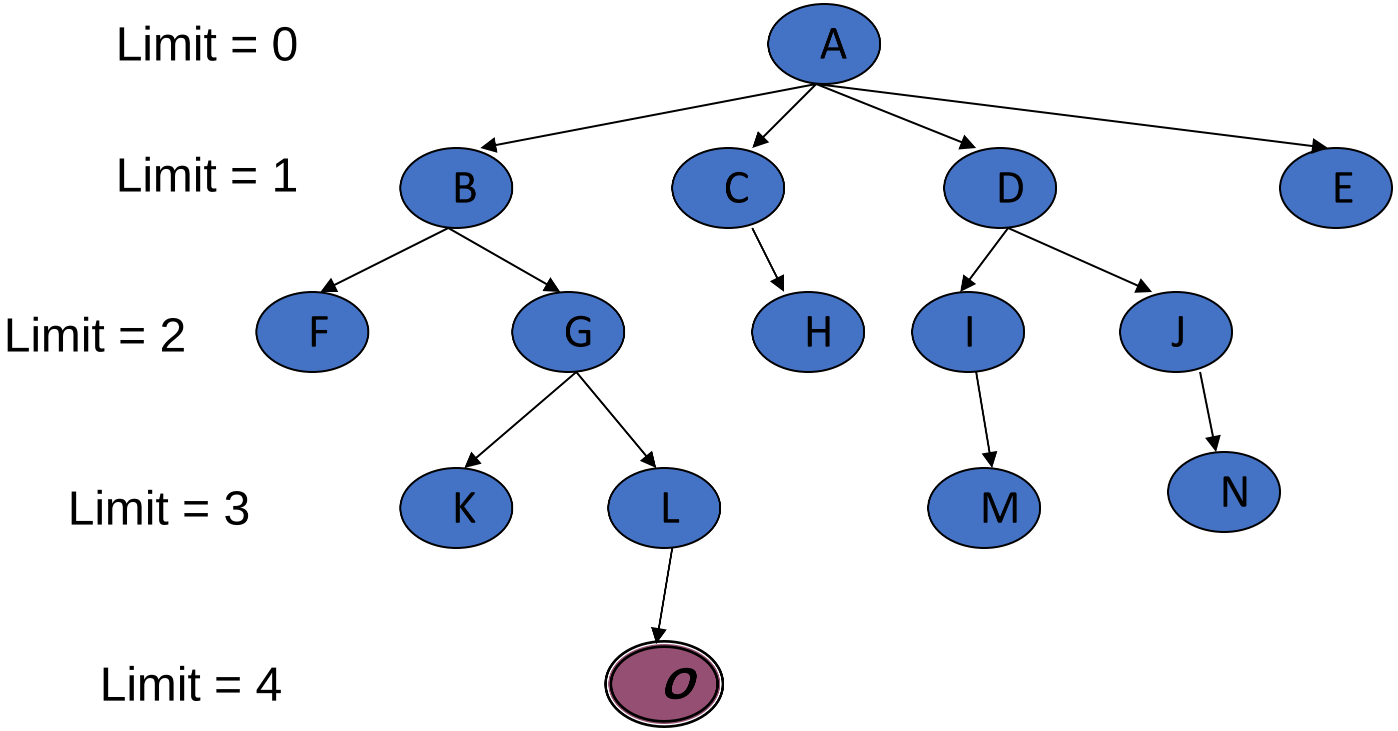
Iterative Deepening Search

IDS

Iterative Deepening Search (IDS)

- Application4:

  Given the following state space (tree search), give the sequence of visited nodes when using IDS:



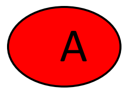
Iterative Deepening Search (IDS)

DLS with bound = 0

Iterative Deepening Search (IDS)

- A,

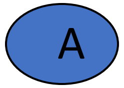
Limit = 0



Iterative Deepening Search (IDS)

- A, Failure

Limit = 0

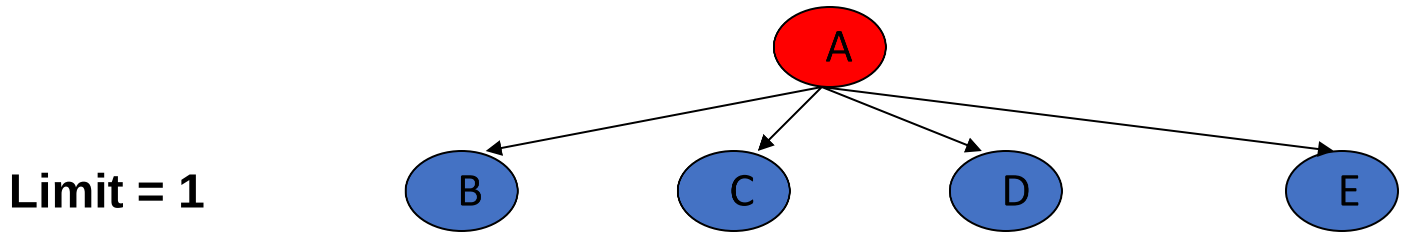


Iterative Deepening Search (IDS)

DLS with bound = 1

Iterative Deepening Search (IDS)

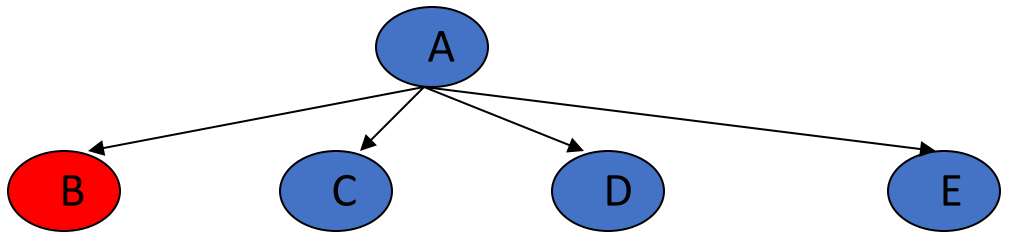
- A,



Iterative Deepening Search (IDS)

- A,B,

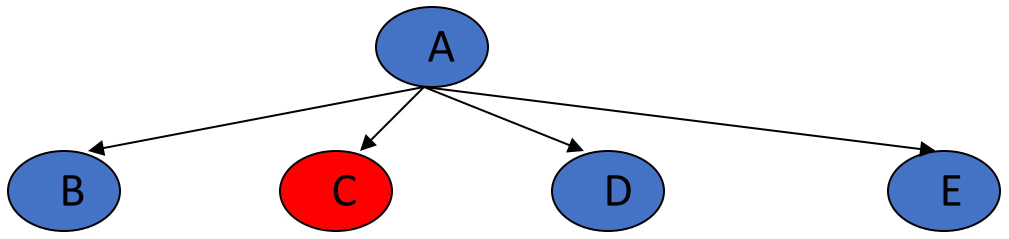
Limit = 1



Iterative Deepening Search (IDS)

- A,B,
- C,

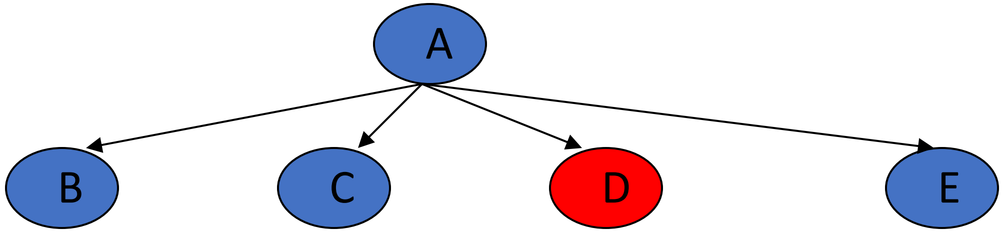
Limit = 1



Iterative Deepening Search (IDS)

- A,B,
- C,
- D,

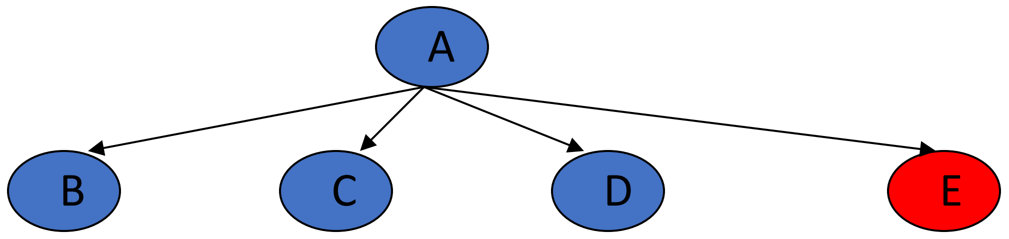
Limit = 1



Iterative Deepening Search (IDS)

- A,B
- C,
- D,
- E,

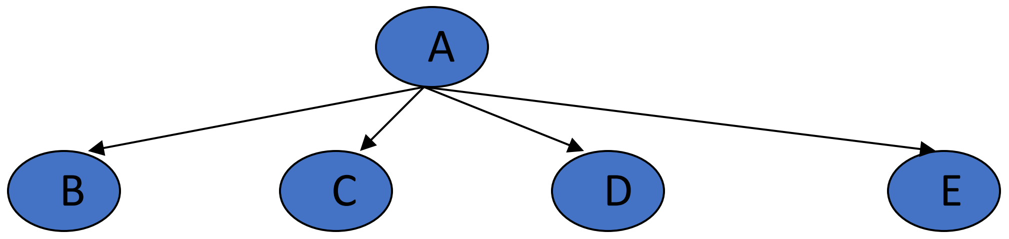
Limit = 1



Iterative Deepening Search (IDS)

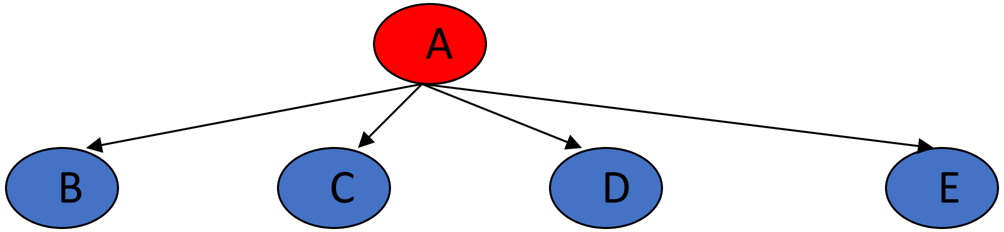
- A,B,
- C,
- D,
- E, Failure

Limit = 1



Iterative Deepening Search (IDS)

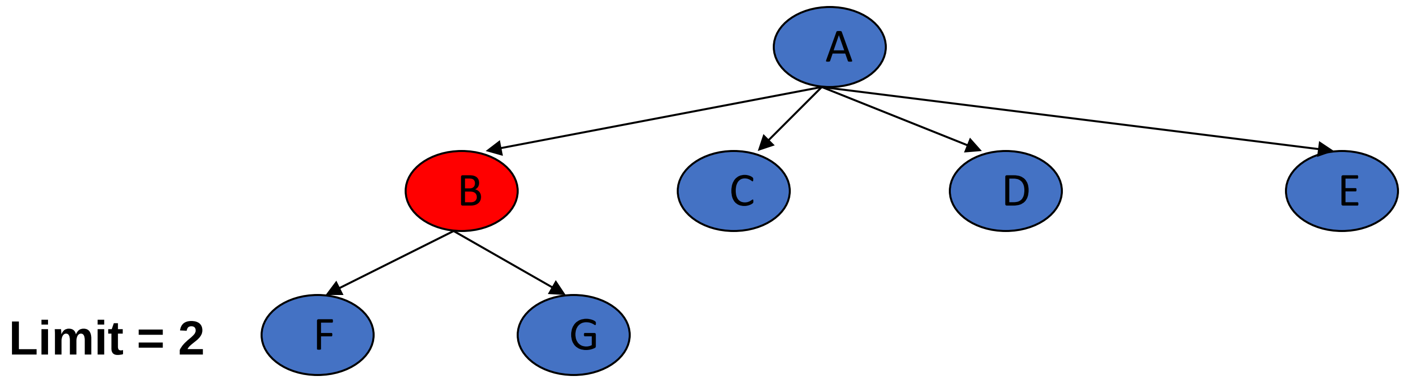
- A,



Limit = 2

Iterative Deepening Search (IDS)

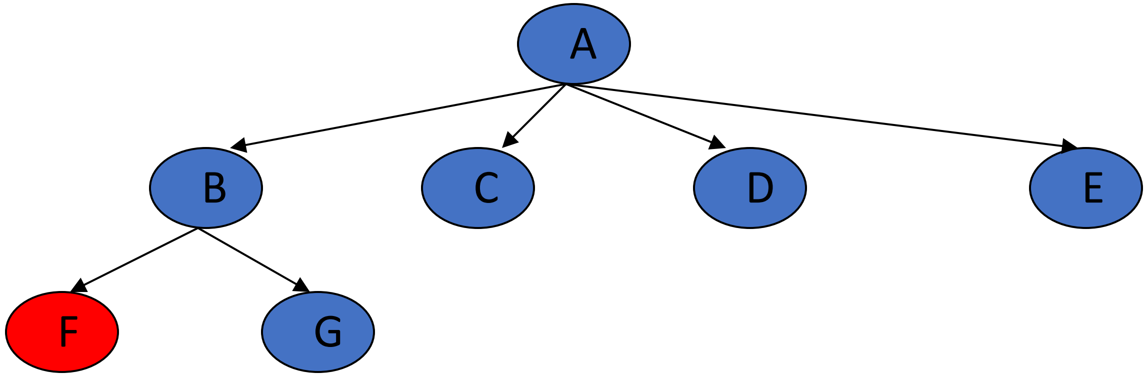
- A,B,



Iterative Deepening Search (IDS)

- A,B,F,

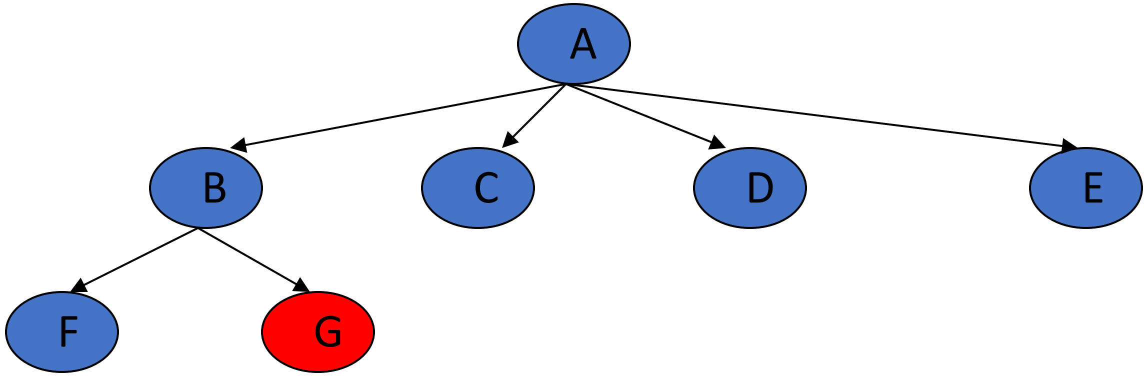
Limit = 2



Iterative Deepening Search (IDS)

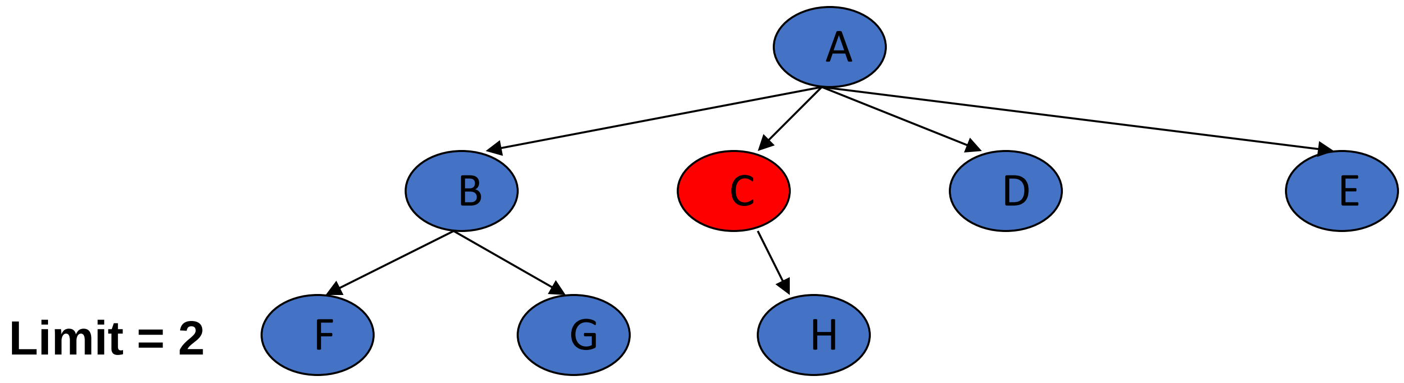
- A,B,F,
- G,

Limit = 2



Iterative Deepening Search (IDS)

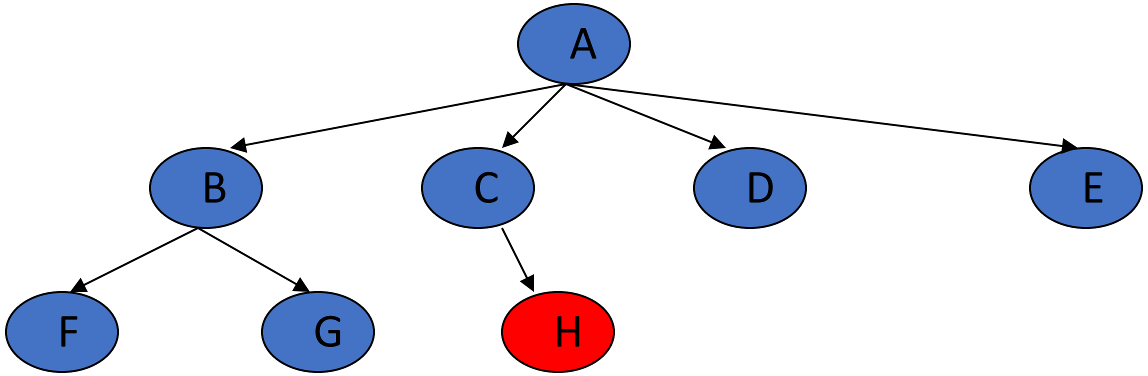
- A,B,F,
- G,
- C,



Iterative Deepening Search (IDS)

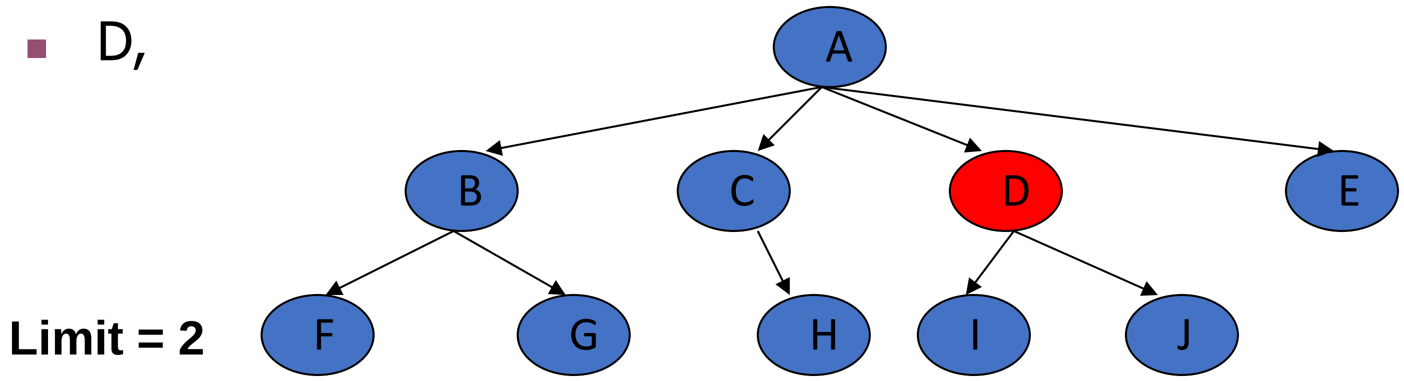
- A,B,F,
- G,
- C,H,

Limit = 2



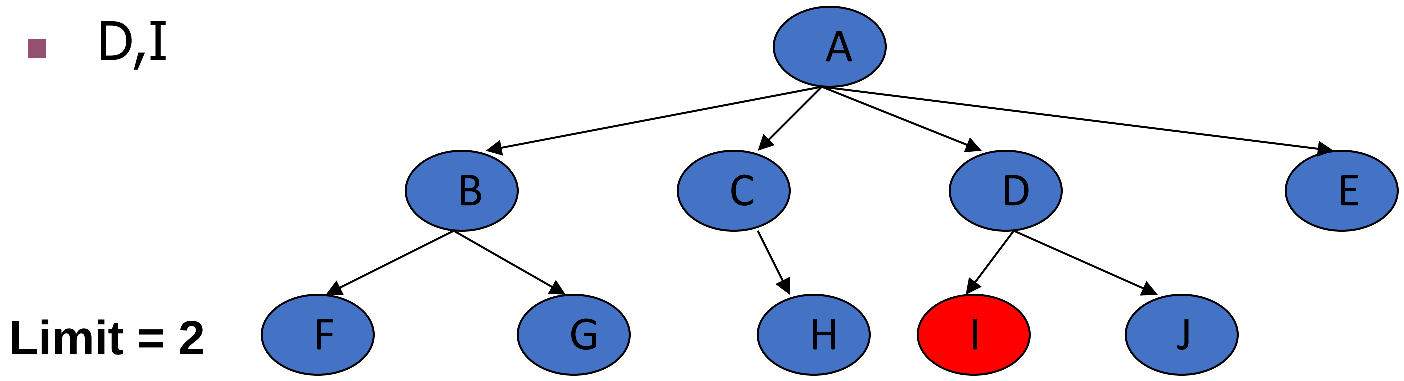
Iterative Deepening Search (IDS)

- A,B,F,
- G,
- C,H,
- D,



Iterative Deepening Search (IDS)

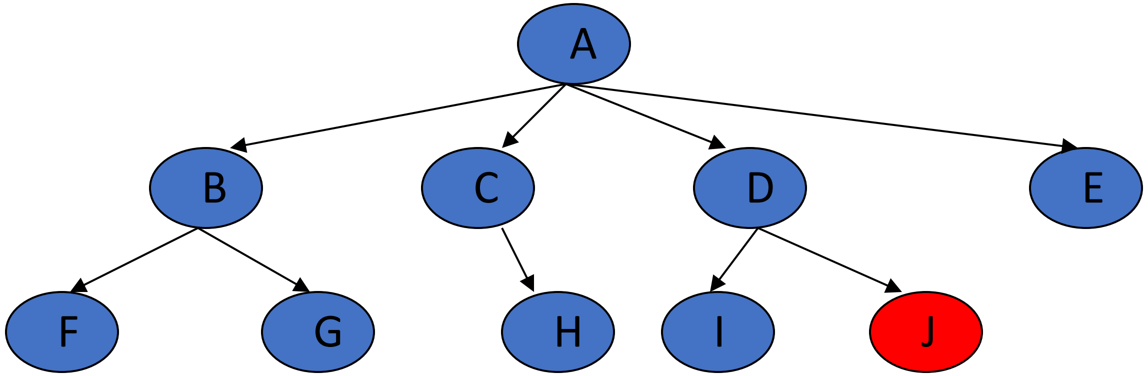
- A,B,F,
- G,
- C,H,
- D,I



Iterative Deepening Search (IDS)

- A,B,F,
- G,
- C,H,
- D,I
- J,

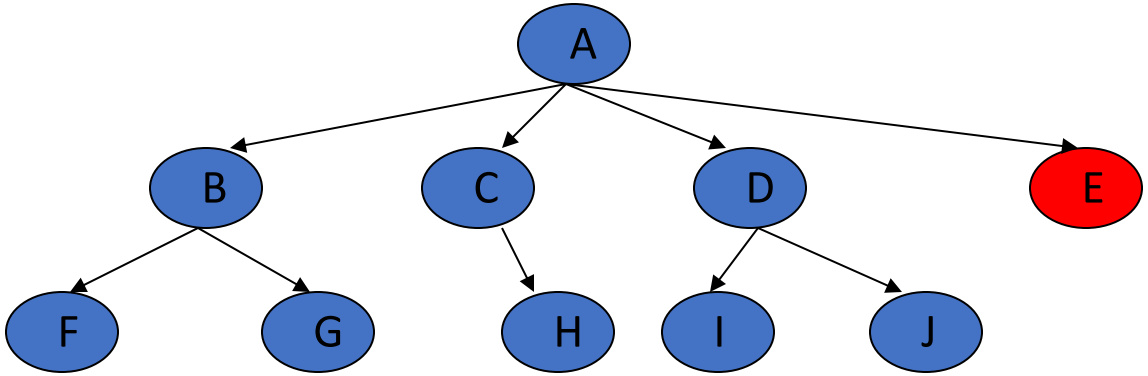
Limit = 2



Iterative Deepening Search (IDS)

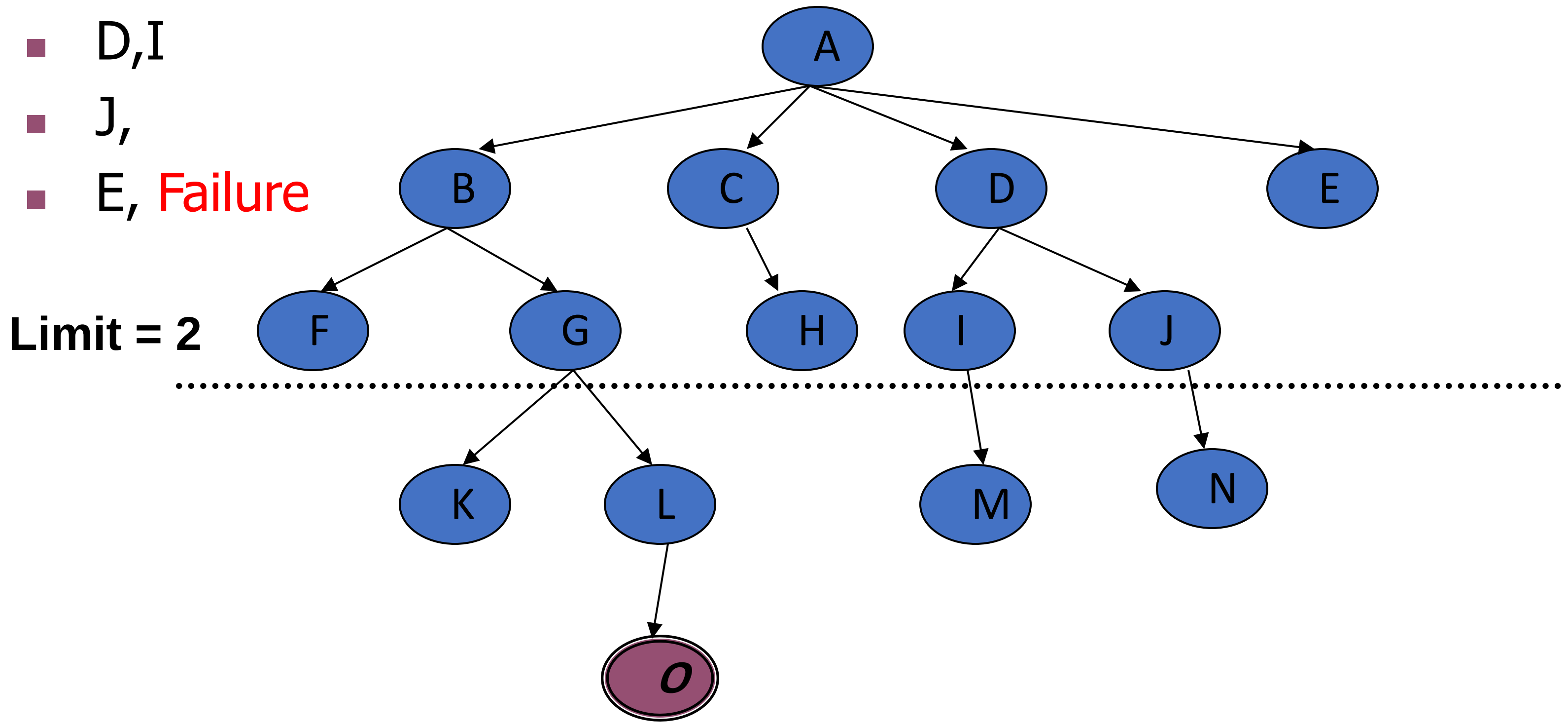
- A,B,F,
- G,
- C,H,
- D,I
- J,
- E

Limit = 2



Iterative Deepening Search (IDS)

- A,B,F,
- G,
- C,H,
- D,I
- J,
- E, Failure

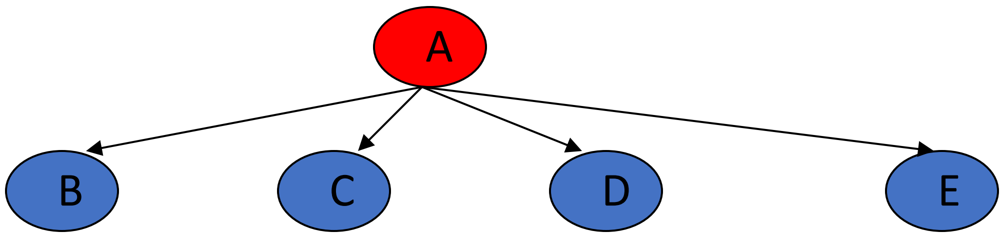


Iterative Deepening Search (IDS)

DLS with bound = 3

Iterative Deepening Search (IDS)

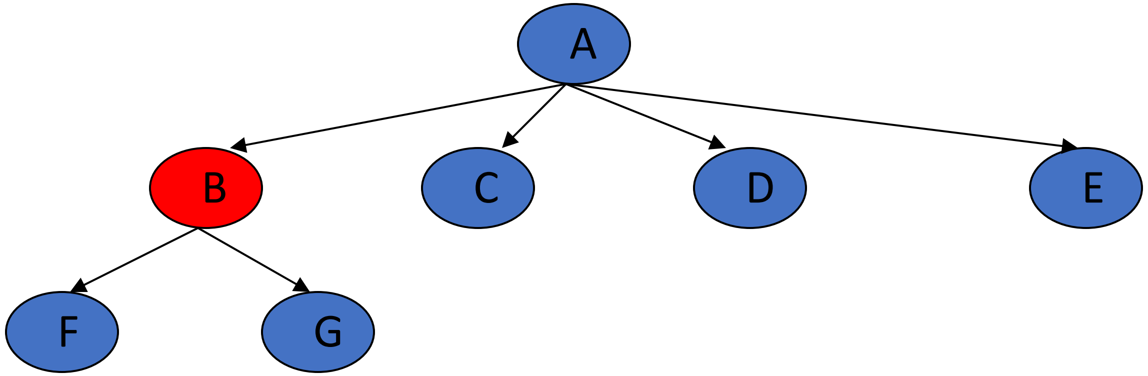
- A,



Limit = 3

Iterative Deepening Search (IDS)

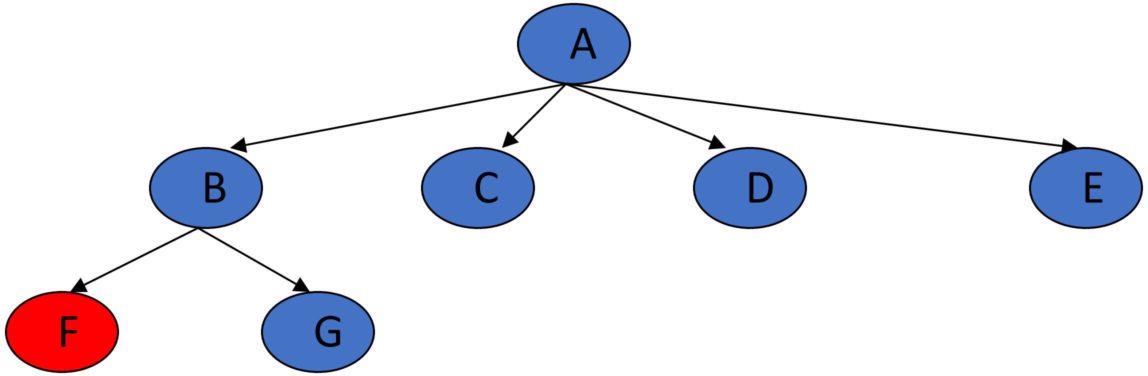
- A,B,



Limit = 3

Iterative Deepening Search (IDS)

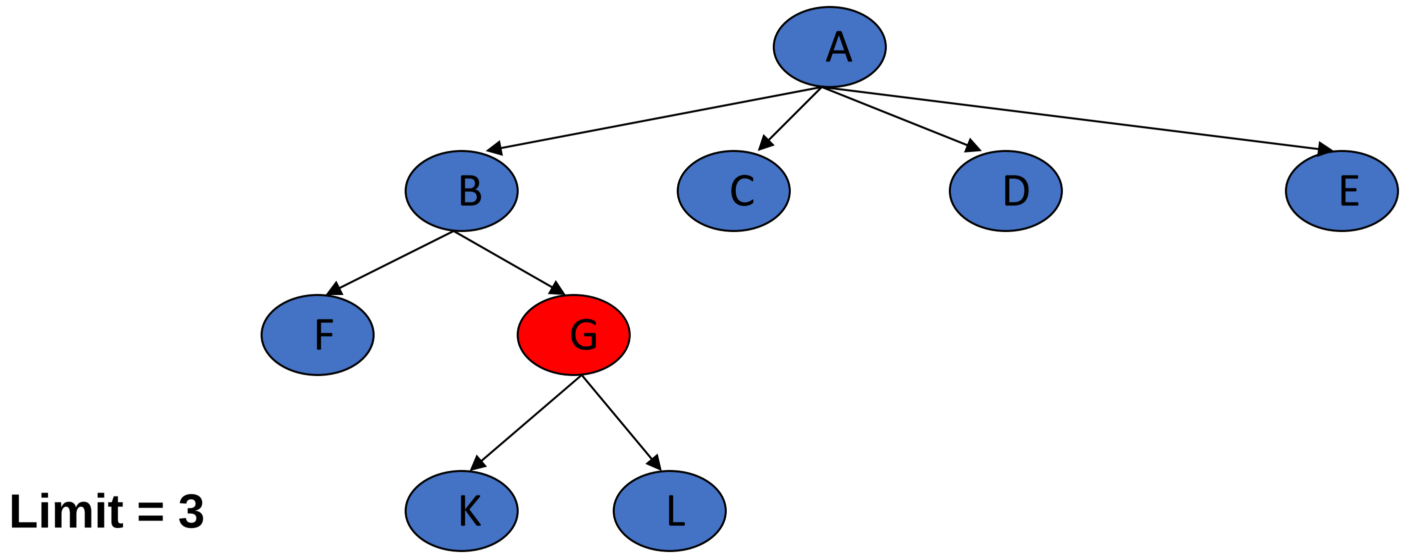
- A,B,F,



Limit = 3

Iterative Deepening Search (IDS)

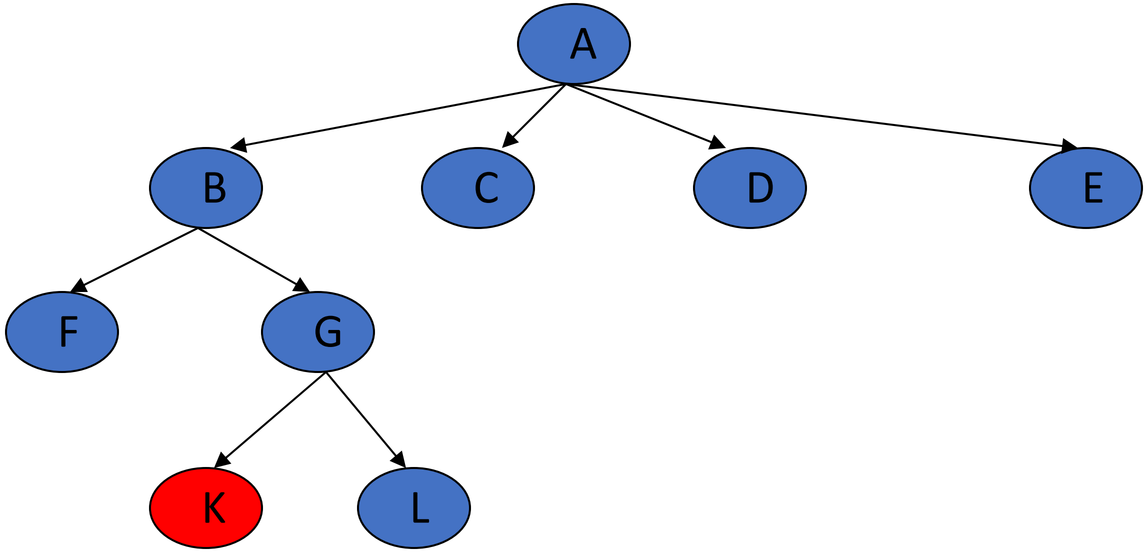
- A,B,F,
- G,



Iterative Deepening Search (IDS)

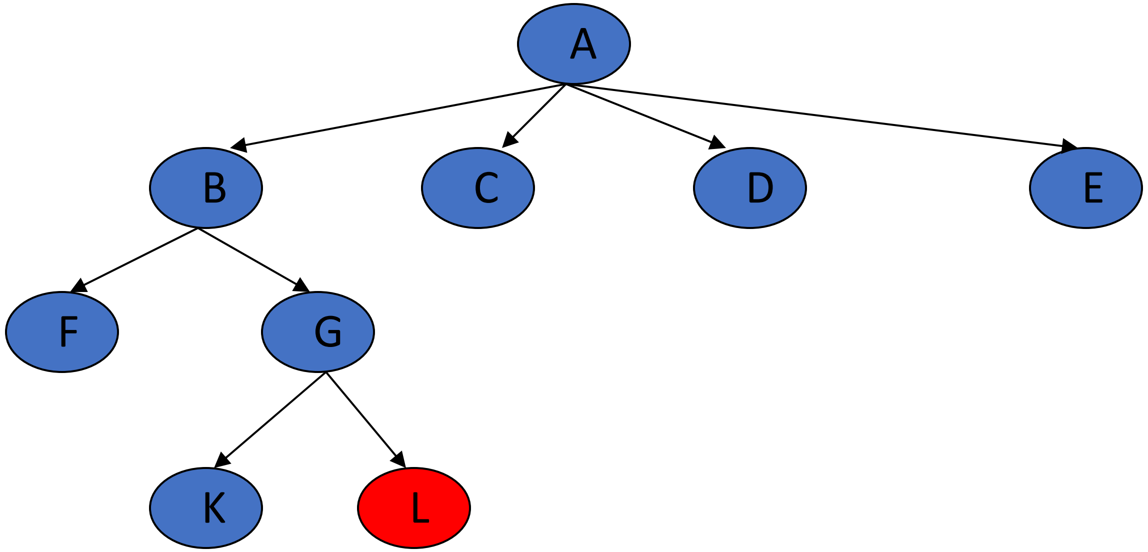
- A,B,F,
- G,K,

Limit = 3



Iterative Deepening Search (IDS)

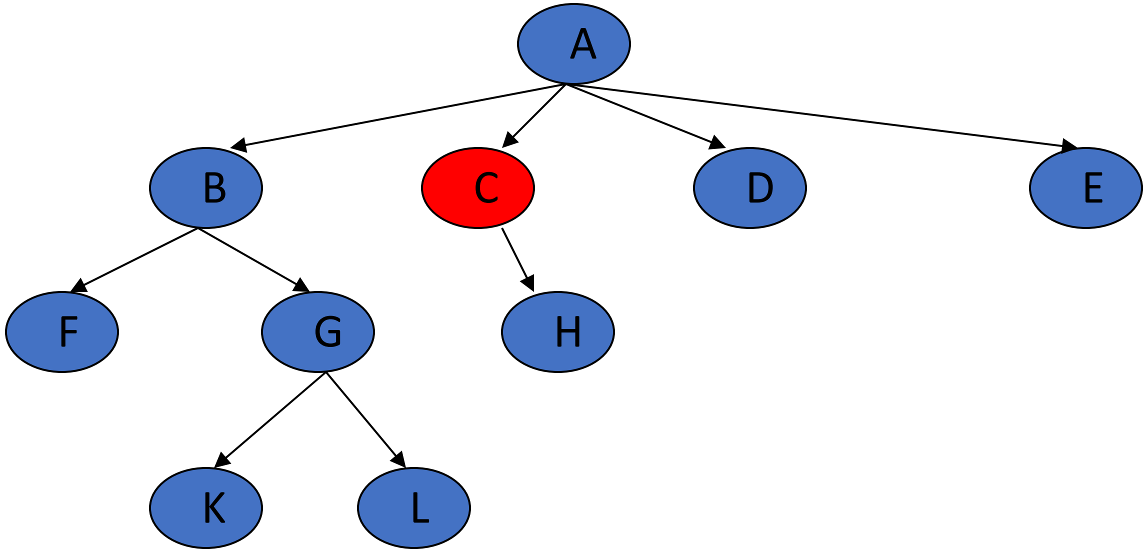
- A,B,F,
- G,K,
- L,



Limit = 3

Iterative Deepening Search (IDS)

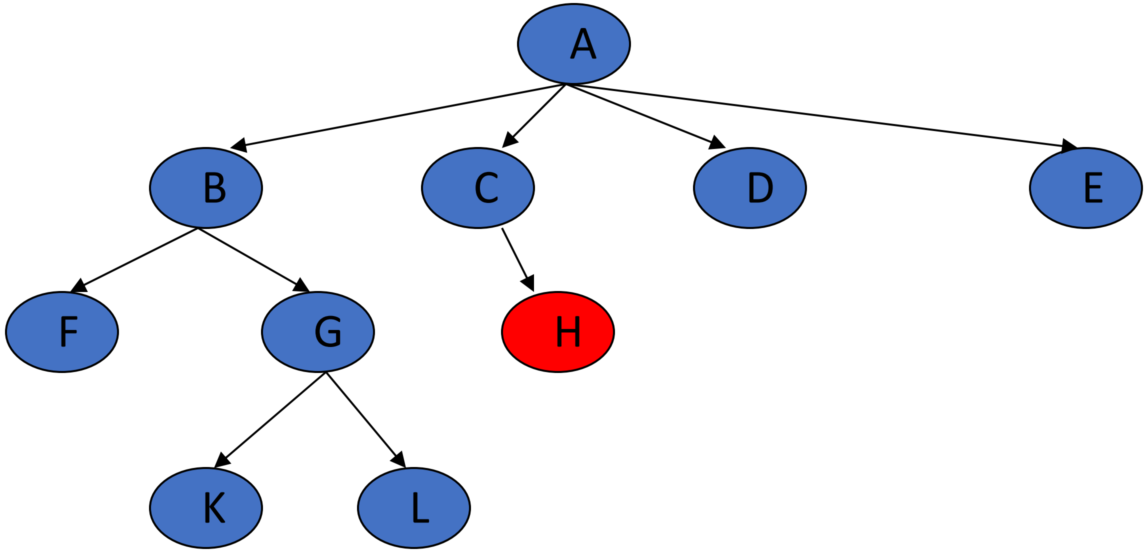
- A,B,F,
- G,K,
- L,
- C,



Limit = 3

Iterative Deepening Search (IDS)

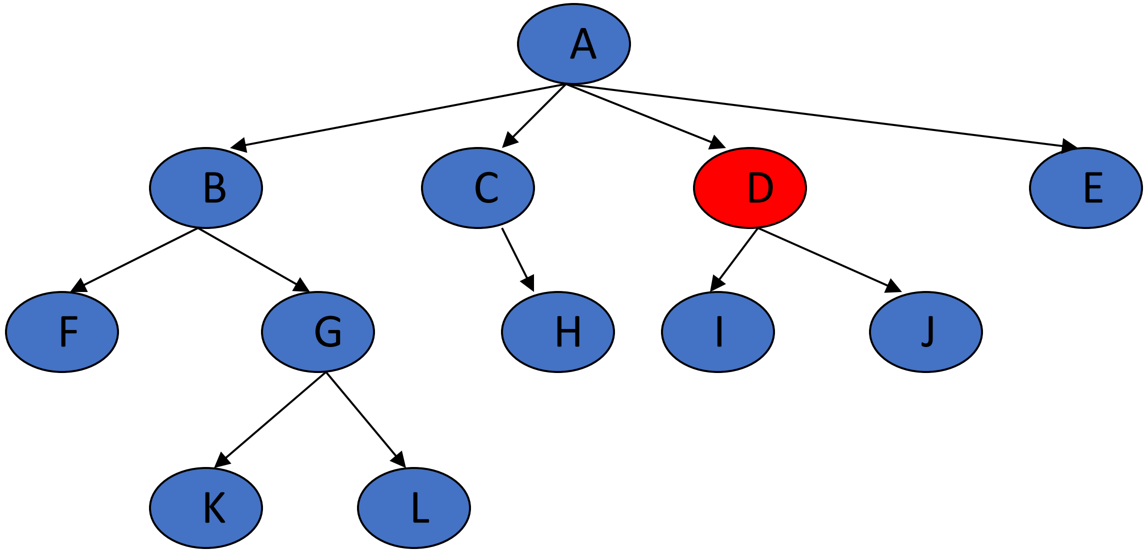
- A,B,F,
- G,K,
- L,
- C,H,



Limit = 3

Iterative Deepening Search (IDS)

- A,B,F,
- G,K,
- L,
- C,H,
- D,

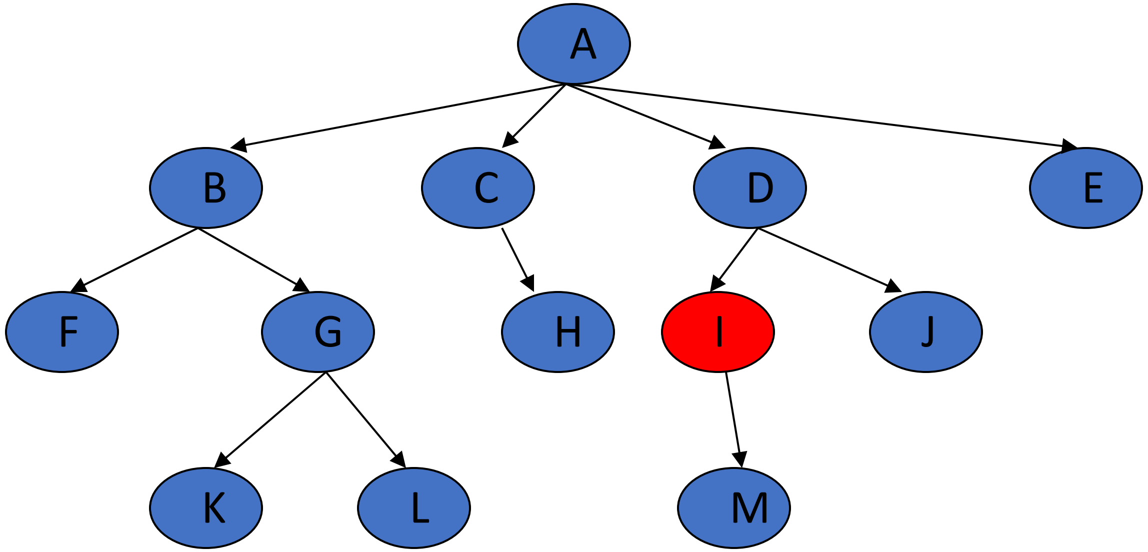


Limit = 3

Iterative Deepening Search (IDS)

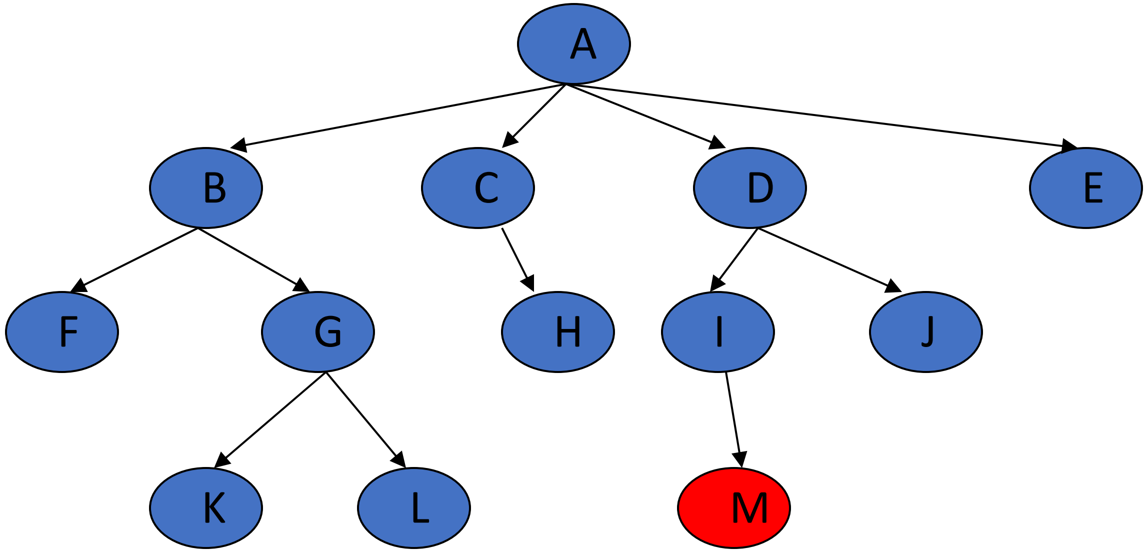
- A,B,F,
- G,K,
- L,
- C,H,
- D,I,

Limit = 3



Iterative Deepening Search (IDS)

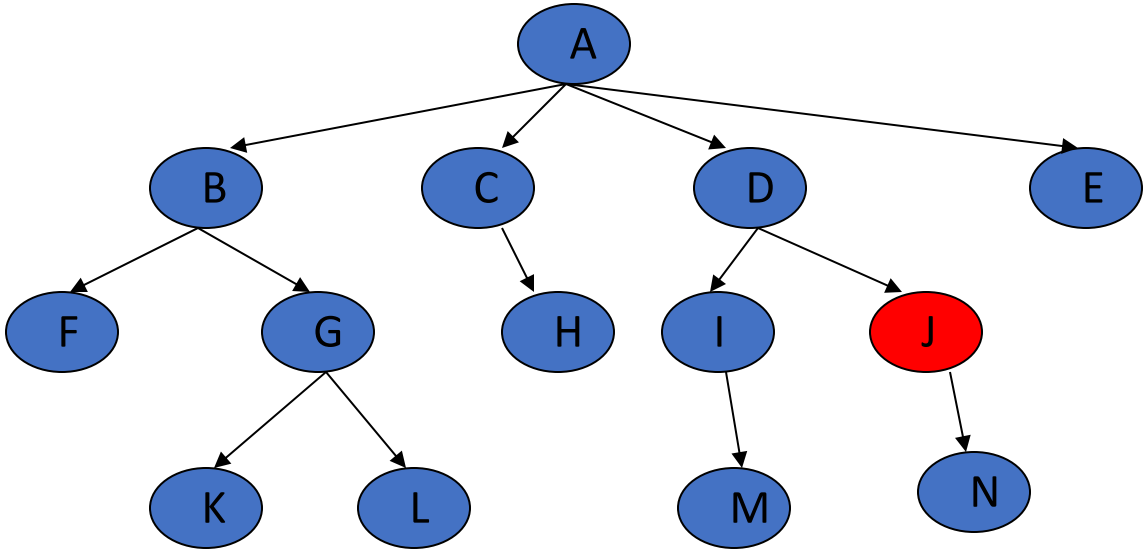
- A,B,F,
- G,K,
- L,
- C,H,
- D,I,M,



Limit = 3

Iterative Deepening Search (IDS)

- A,B,F,
- G,K,
- L,
- C,H,
- D,I,M,
- J,

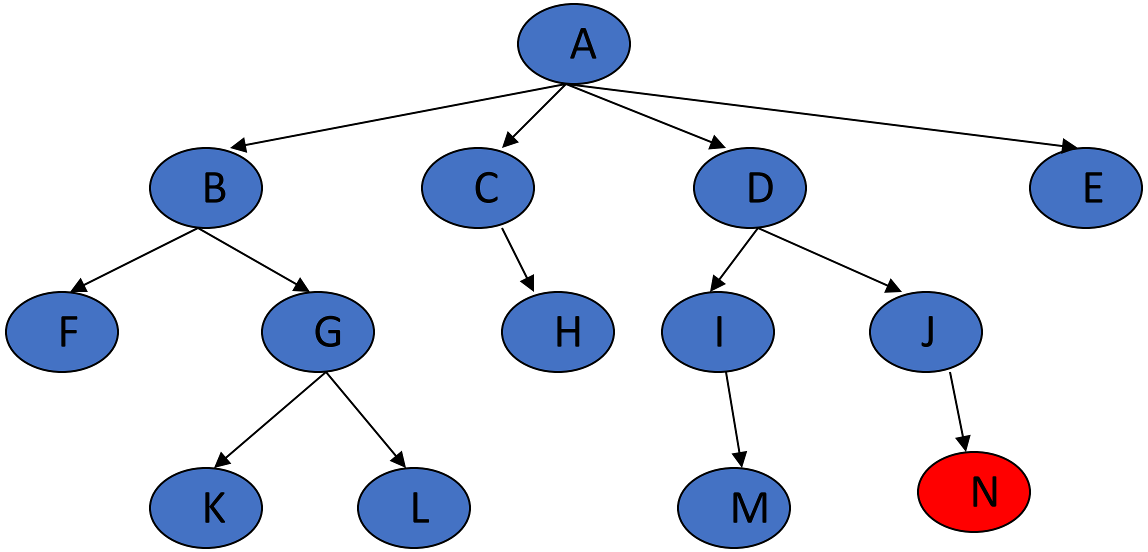


Limit = 3

Iterative Deepening Search (IDS)

- A,B,F,
- G,K,
- L,
- C,H,
- D,I,M,
- J,N,

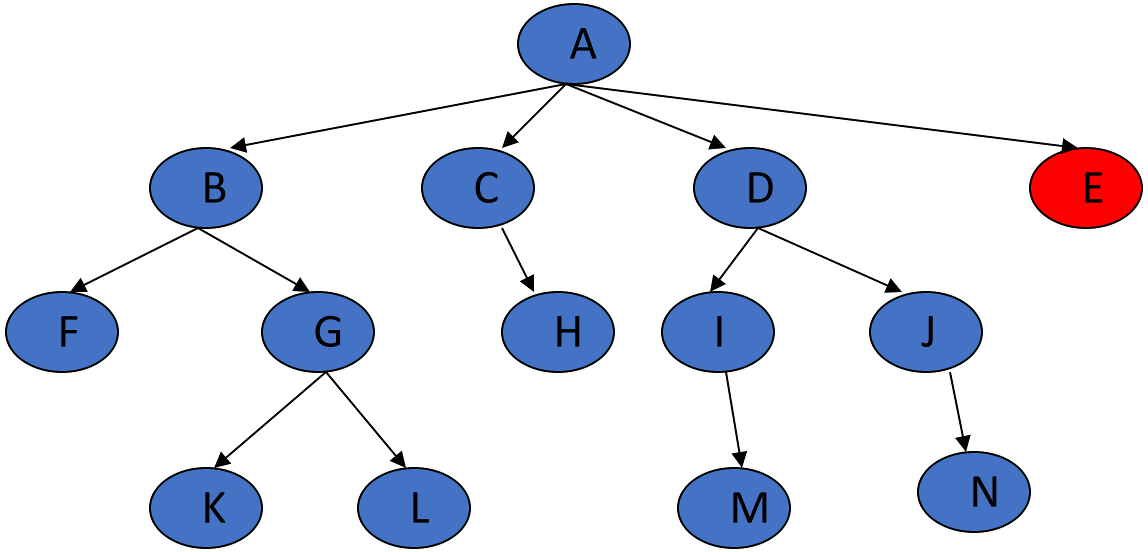
Limit = 3



Iterative Deepening Search (IDS)

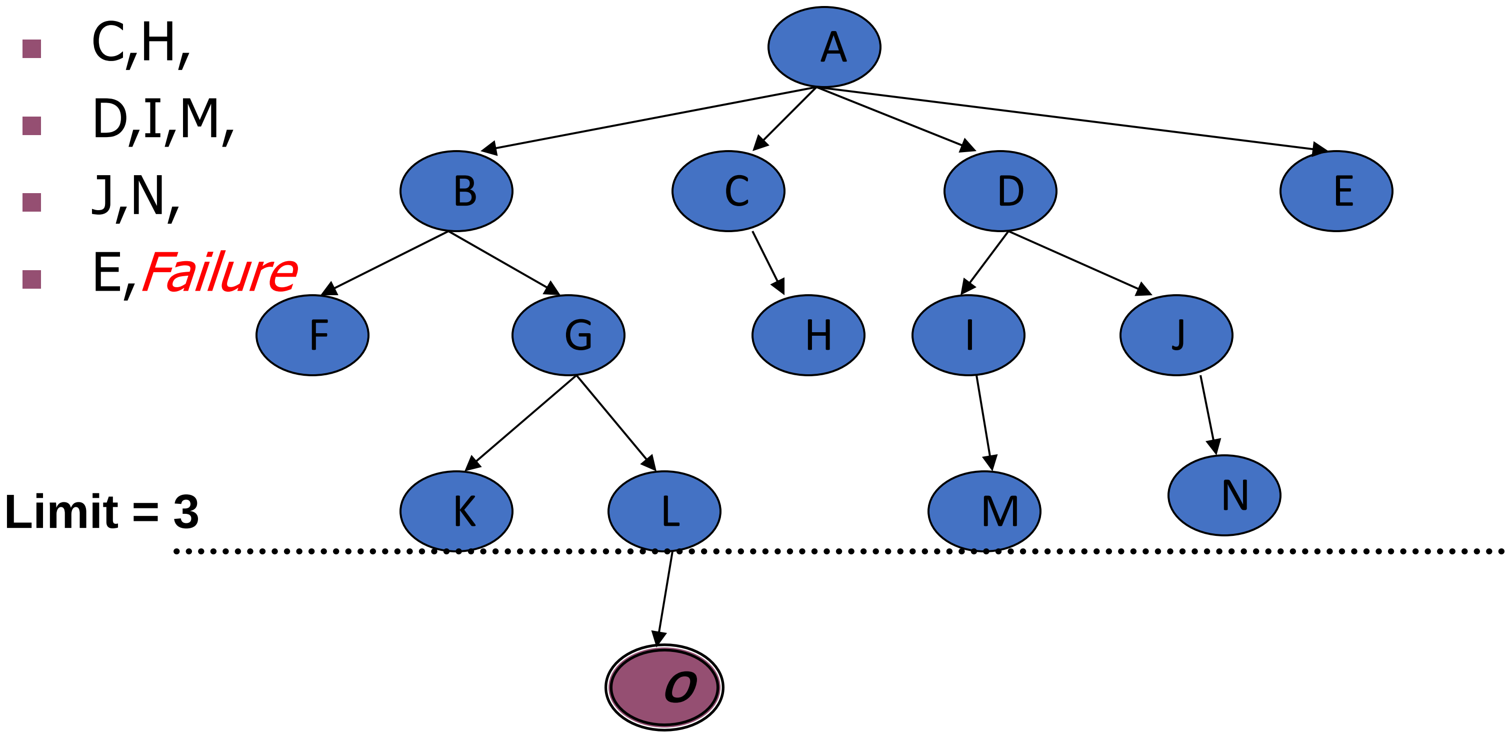
- A,B,F,
- G,K,
- L,
- C,H,
- D,I,M,
- J,N,
- E,

Limit = 3



Iterative Deepening Search (IDS)

- A,B,F,
- G,K,
- L,
- C,H,
- D,I,M,
- J,N,
- E,*Failure*

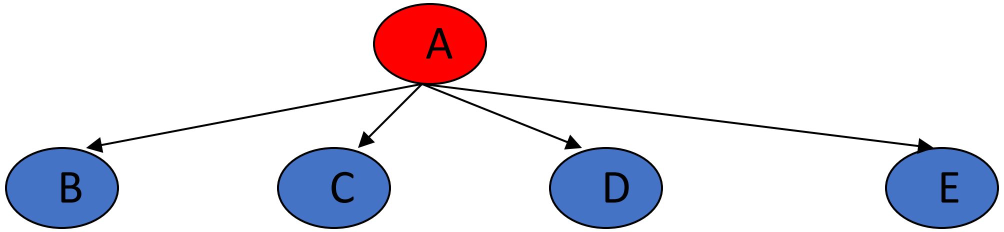


Iterative Deepening Search (IDS)

DLS with bound = 4

Iterative Deepening Search (IDS)

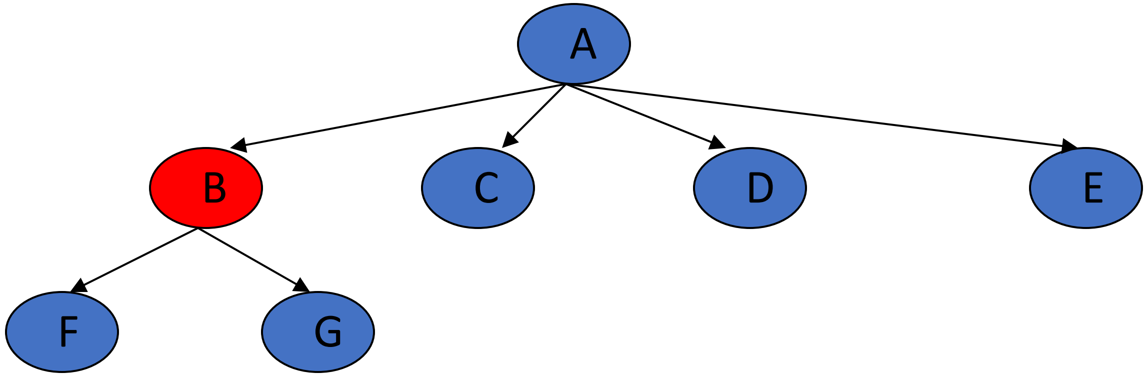
- A,



Limit = 4

Iterative Deepening Search (IDS)

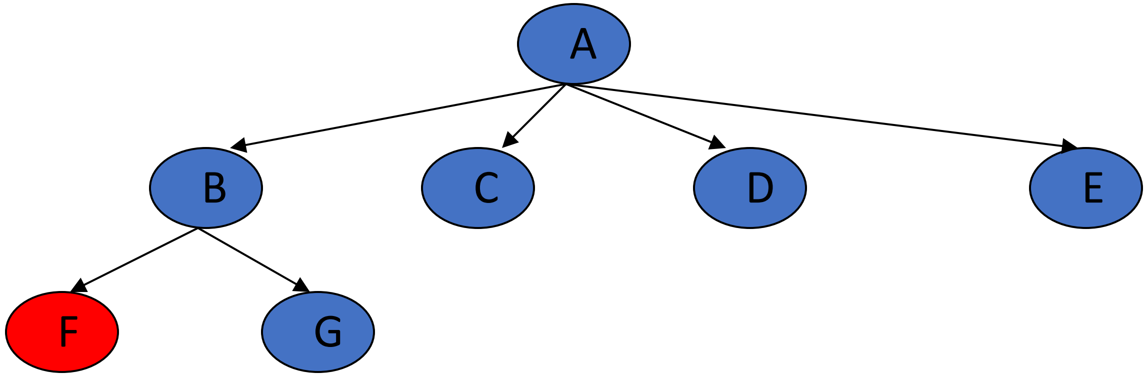
- A,B,



Limit = 4

Iterative Deepening Search (IDS)

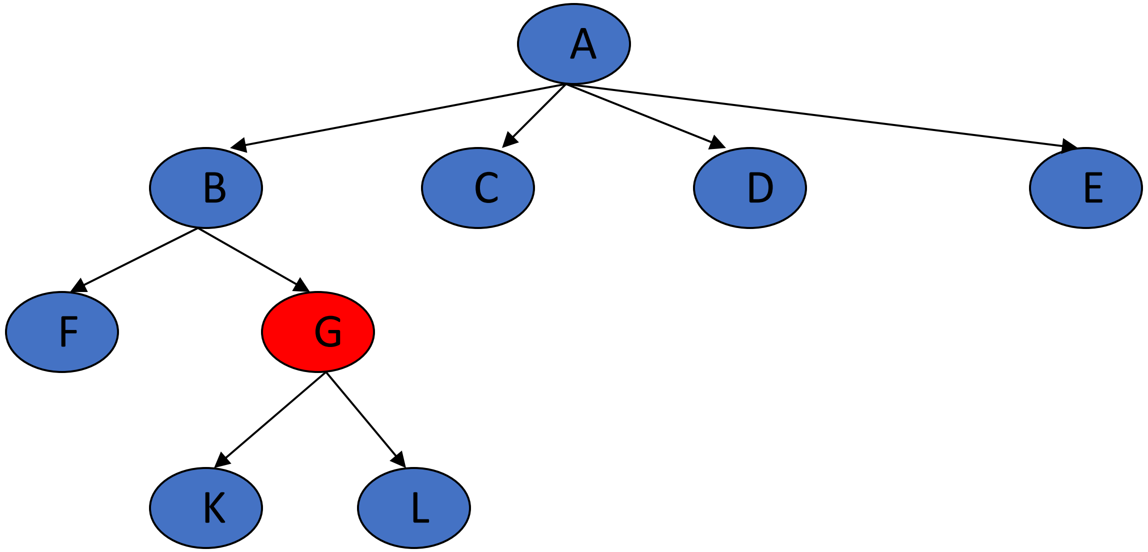
- A,B,F,



Limit = 4

Iterative Deepening Search (IDS)

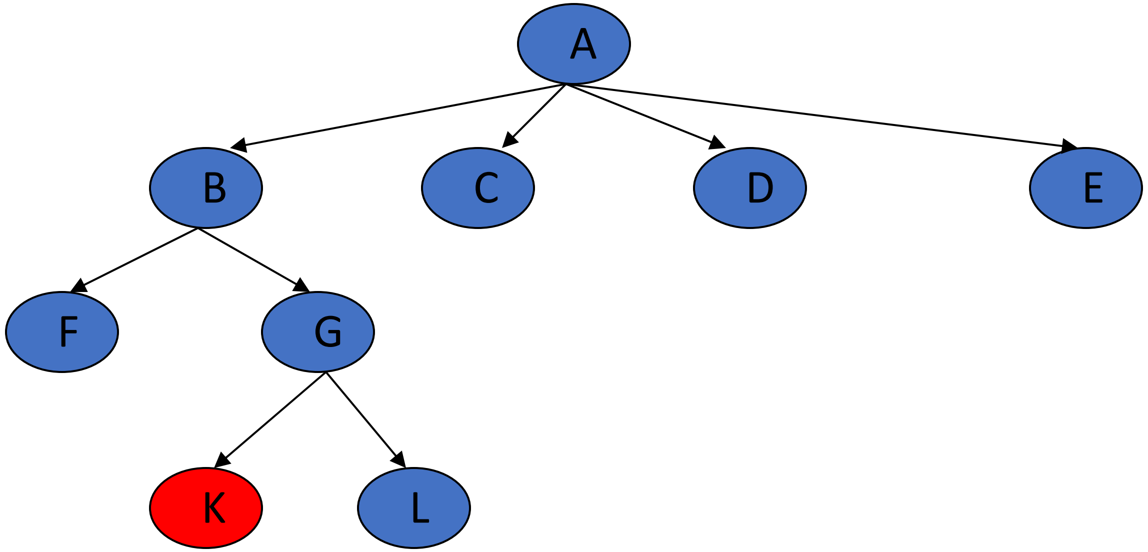
- A,B,F,
- G,



Limit = 4

Iterative Deepening Search (IDS)

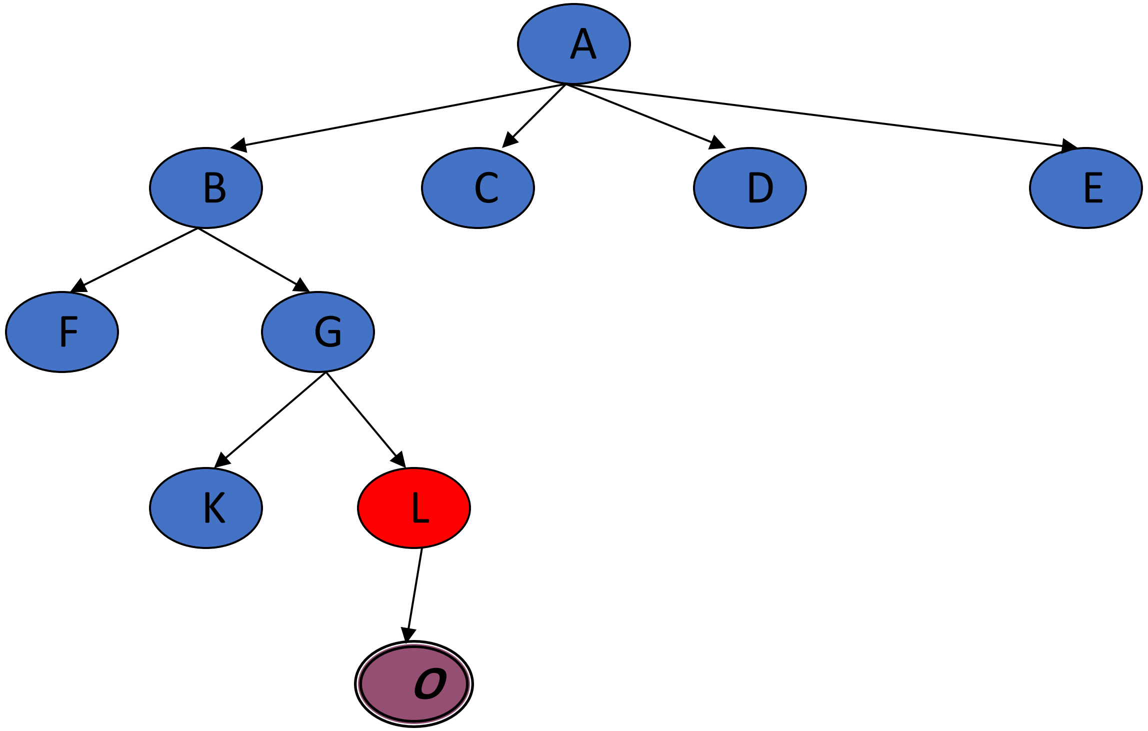
- A,B,F,
- G,K,



Limit = 4

Iterative Deepening Search (IDS)

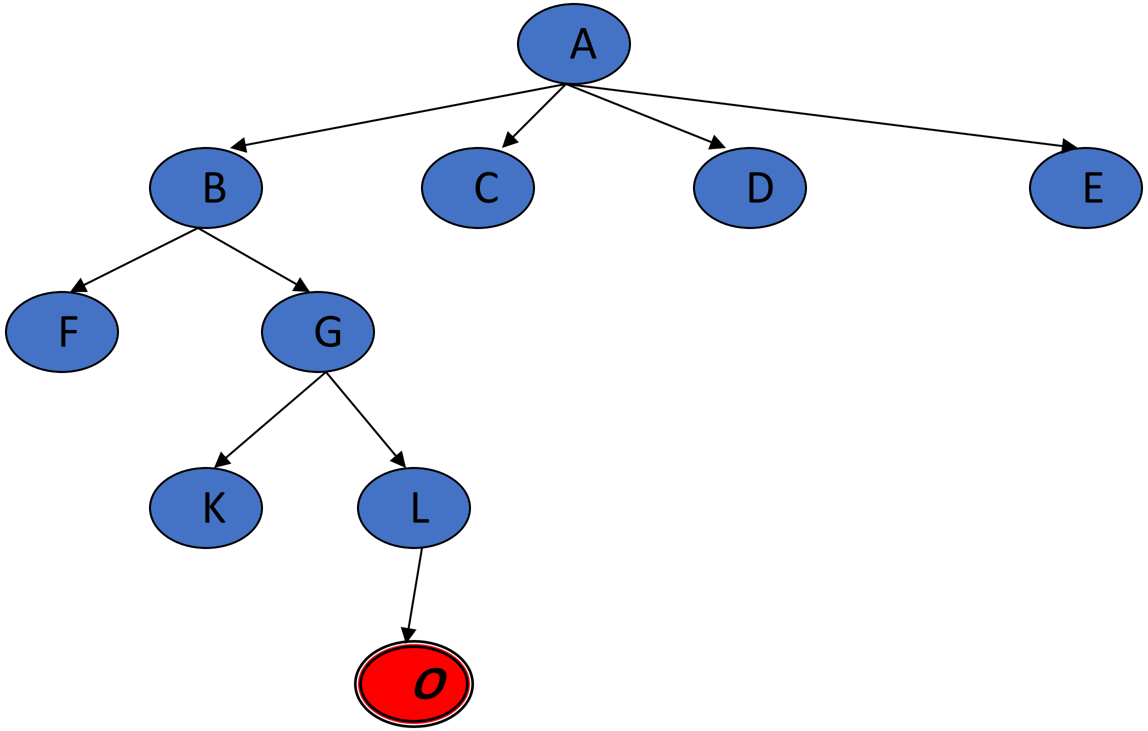
- A,B,F,
- G,K,
- L,



Limit = 4

Iterative Deepening Search (IDS)

- A,B,F,
- G,K,
- L, *O: Goal State*

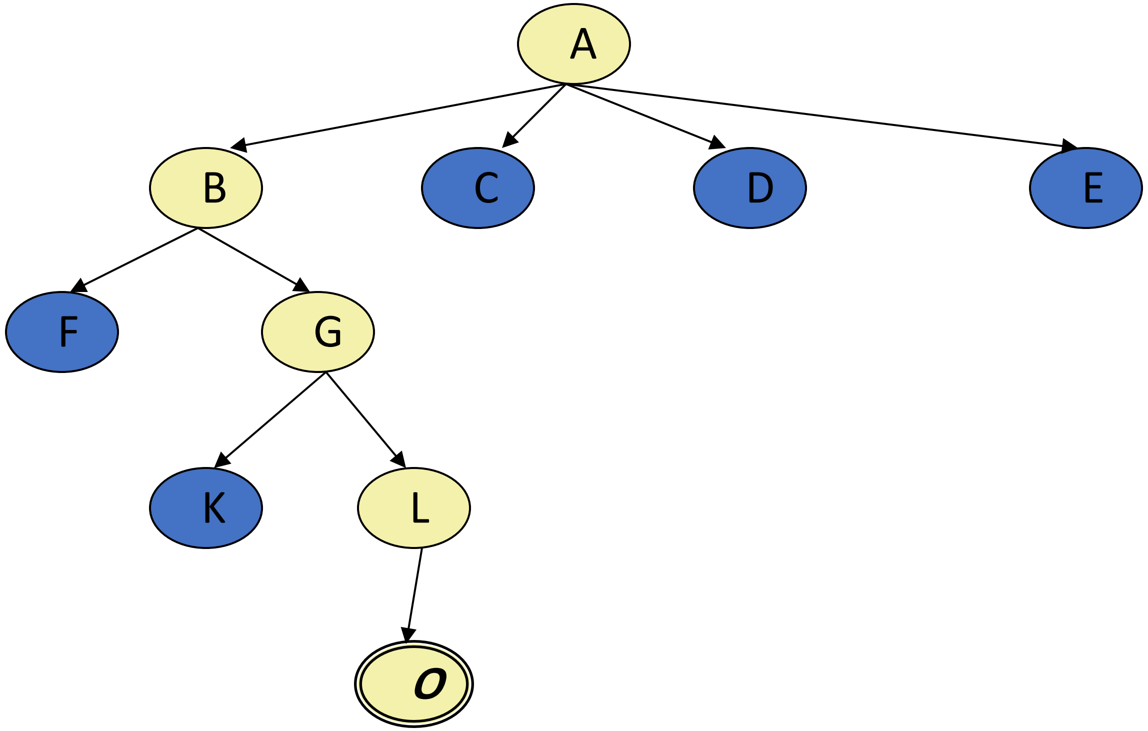


Limit = 4

Iterative Deepening Search (IDS)

The returned solution is the sequence of operators in the path:

A, B, G, L, O



Summary

- ✓ Search: process of constructing sequences of actions that achieve a goal given a problem.
- ✓ The studied methods assume that the environment is observable, deterministic, static and completely known.
- ✓ Goal formulation is the first step in solving problems by searching. It facilitates problem formulation.
- ✓ Formulating a problem requires specifying four components: Initial states, operators, goal test and path cost function. Environment is represented as a state space.
- ✓ A solution is a path from the initial state to a goal state.
- ✓ Search algorithms are judged on the basis of completeness, optimality, time complexity and space complexity.
- ✓ Several search strategies: BFS, DFS, DLS, IDS,…
- ✓ All uninformed searches have an exponential time complexity – hopeless as a viable problem solving mechanism (unless you have a quantum computer!)