UNINFORMED SEARCHING IN AI

SEARCH ALGORITHMS

- In AI, the problems are represented as state space or search space, operators/rules, and goal/final.
- So, in AI problems, certain searching algorithms are required to search for a sequence of steps to be taken to find the solution.
- **Hence search is a characteristic of all AI problems.**

SEARCH PROBLEM REPRESENTATION

- Search problem for a state space $\{S, s_0, A, G, P\}$ is to
  - Plan a sequence of actions

    $$P = \{a_0, a_1, \ldots\ldots a_n\}$$

    which leads to traversing a number of states

    $$\{s_0, s_1, s_2, \ldots\ldots s_{n+1} = g\}$$

    such that the total path cost is optimal.

Where S denote set of states,

s_0 denote initial state,

A denote set of actions (operators),

G is the set of goal state(s) and

P is the path cost.

Basic Search Algorithm 1- State Space Tree Search Algorithm

- OPEN = List of generated but unexplored states; initially containing initial state
- CLOSED = List of explored nodes; initially empty
- **Algorithm:**

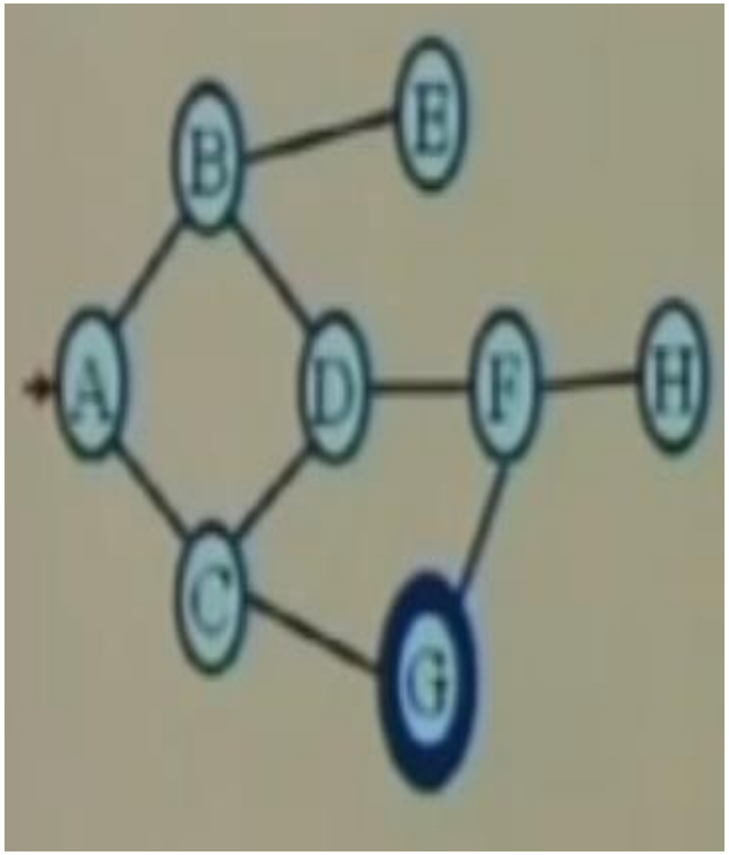
```
Loop until OPEN is empty or success is returned.
  (N,P)→remove-head(OPEN) and add it to CLOSED
      If N is goal node then return success and construct
      path from initial state to N.
      Else generate successors of node N and add it to
      open  i.e. OPEN→OPEN ∪ {MOVEGEN(N)}
      End if
End Loop
```

Basic Search Algorithm 1- State Space Tree Search Algorithm Contd….

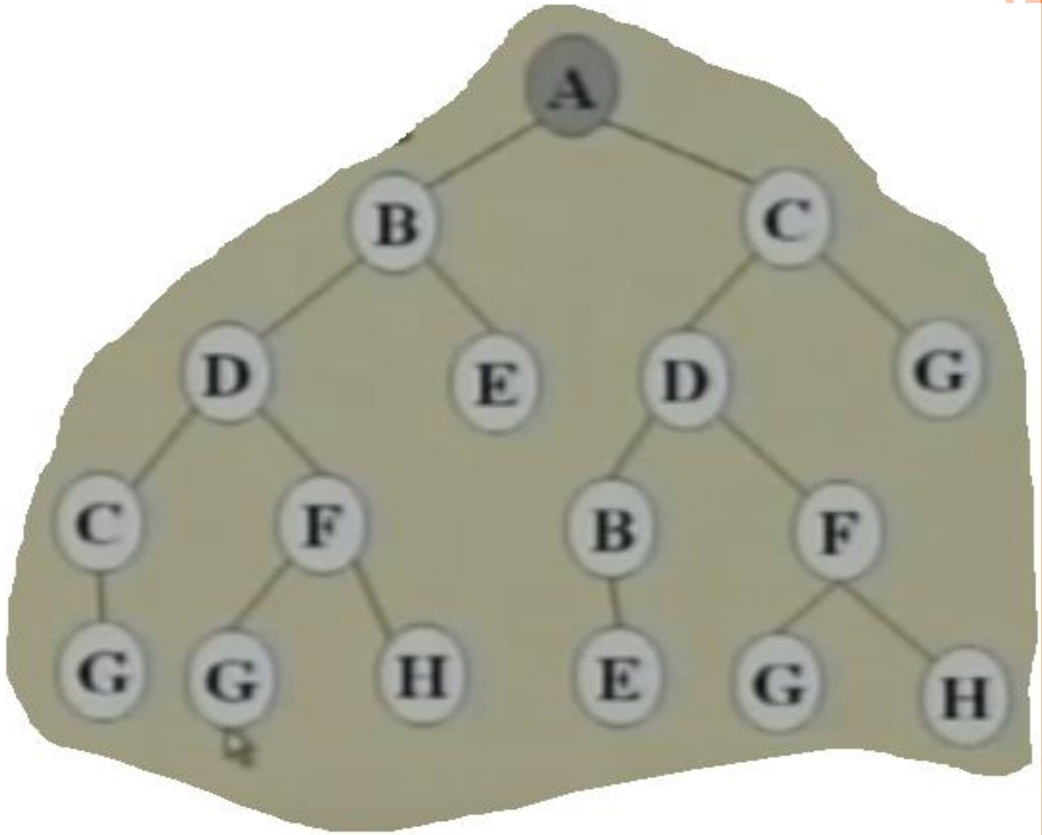
- MOVEGEN(N) function generates successors of node n according to the operators/rules of the problem.
- OPEN is the list that contains all the nodes that have been generated so far but not yet been expanded i.e. the successor function is not applied to the nodes and the children are not generated.
- CLOSED is the list of nodes that have been generated as well as expanded.
- Both the OPEN and CLOSED maintains nodes as node pairs i.e. (N,P) where N is the node and P is the parent node of the tree. This is done to backtrack to the path from the goal node to initial state (if goal node is found).
- The repeated states in the path can be ignored in order to avoid cycles or infinite loops.

- The search algorithm 1 would create a search tree for the problem (even if search space is a graph) with many nodes repeated in the path (shown in the figures in the next slide).
- This is due to the reason that the algorithms adds newly generated children of a node to OPEN without checking that the node has already been generated or expanded.
- Therefore, state space search tree algorithm is not efficient.

Basic Search Algorithm 1- State Space Tree Search Algorithm Contd....



Search Space of problem



Search tree generated using algorithm 1 (with repeated states)

Basic Search Algorithm 2- State Space Graph Search Algorithm

- OPEN = List of generated but unexplored states; initially containing initial state
- CLOSED = List of explored nodes; initially empty
- **Algorithm:**

```
Loop until OPEN is empty or success is returned.
    (N,P)←remove-head(OPEN) and add it to CLOSED
    If N is goal node then return success and construct path
    from initial state to N.
    Else generate successors of node N and add the nodes to
    OPEN that are not already generated or expanded
     i.e. OPEN→OPEN ∪ {MOVEGEN(N)\{OPEN ∪ CLOSED}}
     End if
End Loop
```

- The algorithm does not add any node that has already been generated or explored. Hence it is a better algorithm as it can deal with all kinds of problems.

Classification of Search Algorithms

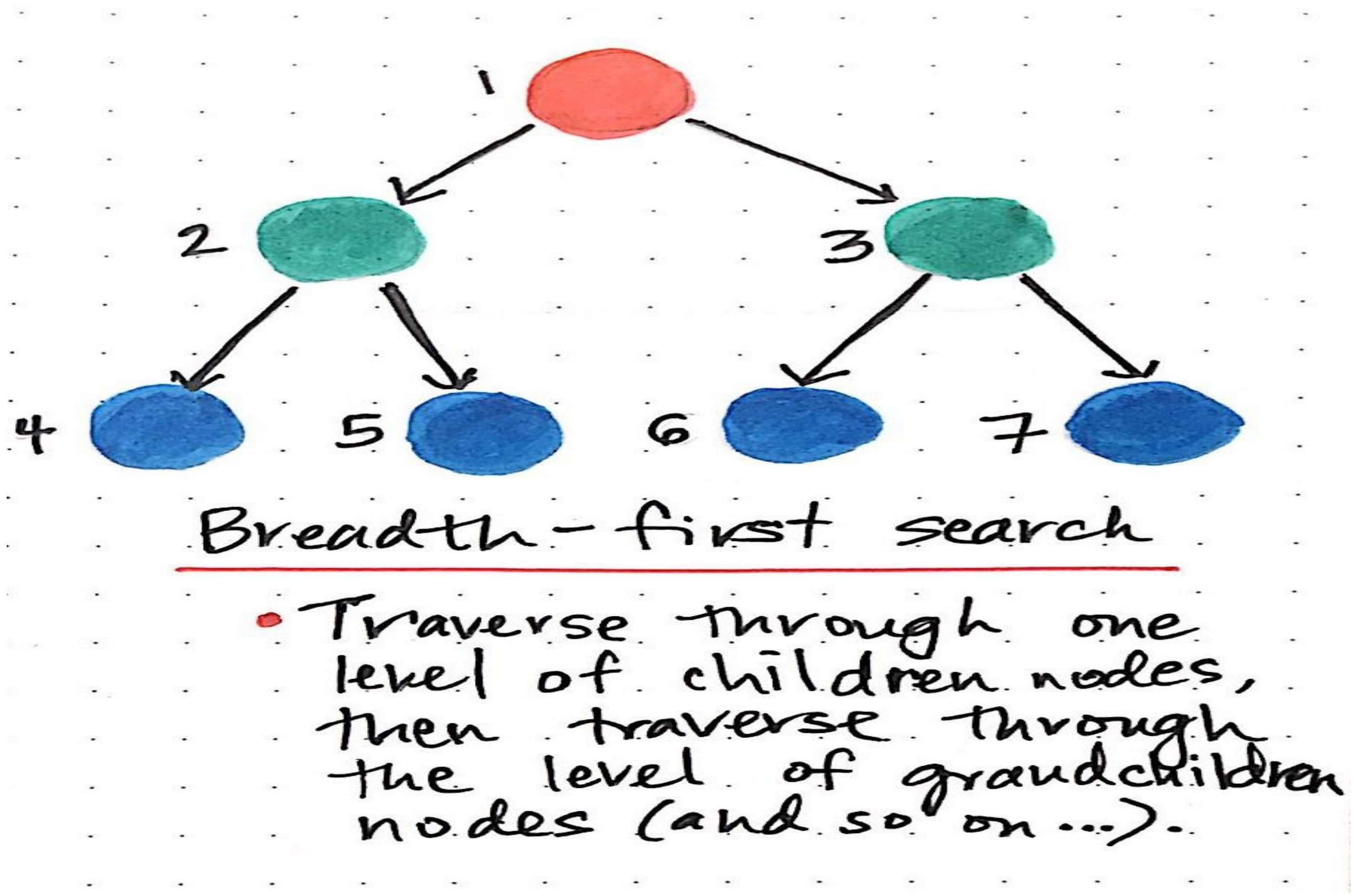
- Search algorithms are broadly classified as:
  - Uninformed (blind) searching techniques.
  - Informed (guided) search techniques.
  - Constraint Satisfaction Search techniques
  - Adversary Search Techniques
  - Stochastic Search Techniques
- These search algorithms differ according to following parameters:
  - The way in which elements are added to the OPEN list (how OPEN is implemented i.e. as a stack, queue, priority queue, etc.).
  - Whether or not elements in the CLOSED list are used for backtracking or not.
  - Whether or not some domain knowledge (in the form of heuristics) is used by the search algorithm or not.
  - Whether or not some random/probabilistic measures are used during the search process or not.

Uninformed / Blind Searching Methods

- Blind search occurs when we do not have any information about the direction in which we should go while searching.
- In these types of algorithms , the nodes in the state space are explored until a goal is found or failure occurs or the space is exhausted or time is limited.
- Various types of uniformed search algorithms include:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Depth-Limited Search
  - DFS with iterative deepening
  - Uniform Cost Search
  - Bidirectional Search

Breadth-First Searching

- BFS is performed by exploring all the nodes at a given depth before moving to the next level.


1
2
3
4
5
6
7
Breadth-first search
• Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

Breadth-First Searching

- The space require for BFS is high if the branching factor (the average number of children of a node) is large.
- Still it is used as we are exploring all the nodes at one level before going to the next level. So it is guaranteed that goal is not left at the lower level.
- **BFS is implemented with the help of queue.**
- **The newly generated children are placed at the end of the queue.**

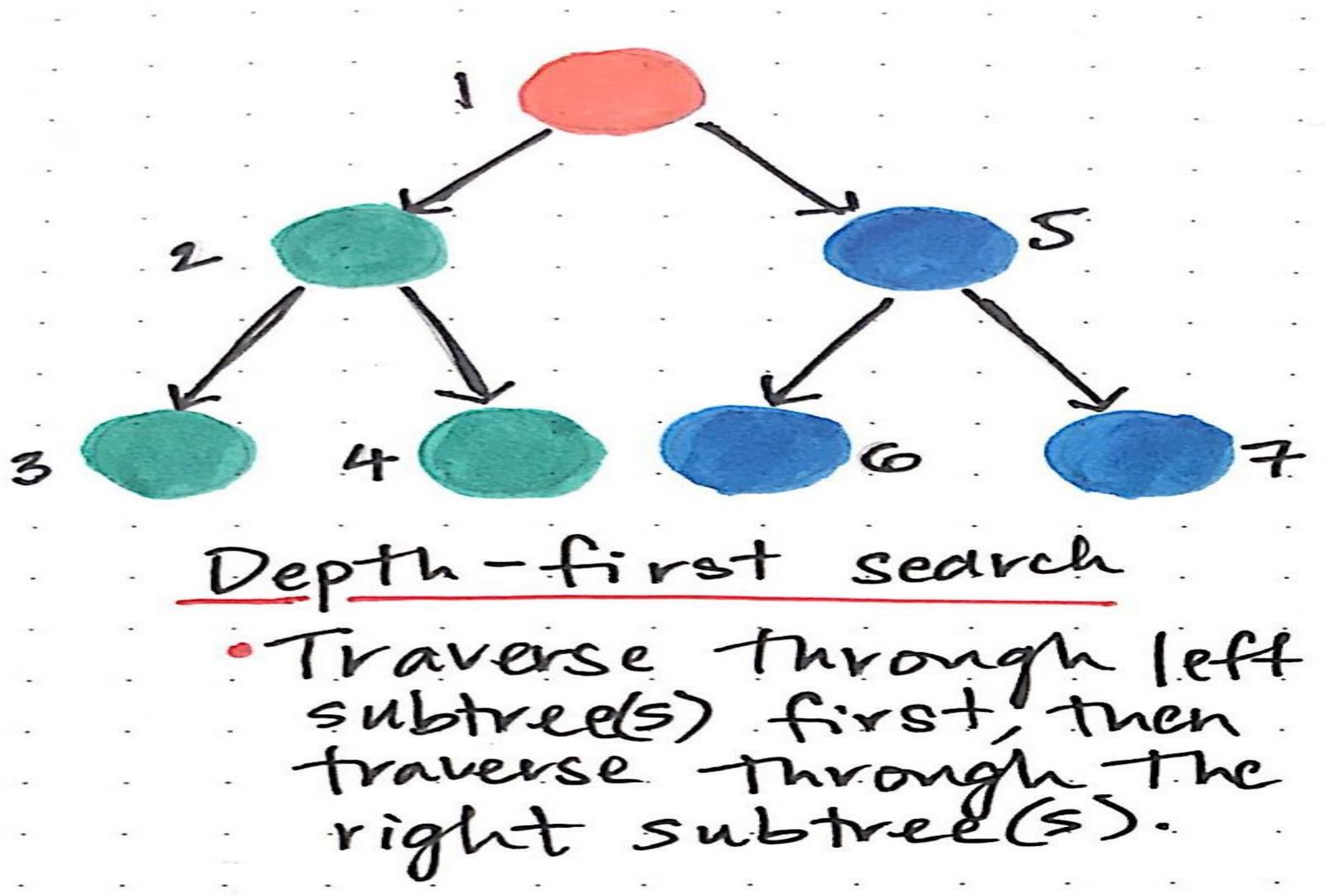
Breadth-First Search Algorithm

- OPEN = List of generated but unexplored states; initially containing initial state (OPEN is used as a QUEUE in BFS).
- CLOSED = List of explored nodes; initially empty
- **Algorithm:**

```
Loop until OPEN is empty or success is returned.
    (N,P)←remove-head(OPEN) and add it to CLOSED
    If N is goal node then return success and construct path from initial state to N.
    Else generate successors of node N and add the nodes to end of OPEN that are not already generated or expanded.
      i.e. OPEN→OPEN ∪ {MOVEGEN(N)\{OPEN ∪ CLOSED}} (to end)
    End if
End Loop
```

Depth-First Search

- Depth-first search is performed by moving downward in the tree in depth-first fashion.


1
2
5
3
4
6
7
Depth-first search
• Traverse through left subtree(s) first, then traverse through the right subtree(s).

Depth-First Search

- The leftmost branch is explored till either the goal node is found or leaf node is reached.
- If the goal is found, then search terminates by giving the solution from root node to goal node.
- Otherwise, backtracking is done for the elements from the right side of the leaf node till the root node.
- If still the solution is not found, then the next branch is explored in the similar way.
- **DFS is implemented with the help of stack.**
- **The newly generated children are placed at the head of the OPEN (stack) so that they are processed first.**

Depth-First Search Algorithm

- OPEN = List of generated but unexplored states; initially containing initial state (OPEN is used as a STACK in DFS)
- CLOSED = List of explored nodes; initially empty
- **Algorithm:**

```
Loop until OPEN is empty or success is returned.
    (N,P)←remove-head(OPEN) and add it to CLOSED
    If N is goal node then return success and construct path
    from initial state to N.
    Else generate successors of node N and add the nodes **to
    front of OPEN** that are not already generated or
    expanded.
     i.e. OPEN→OPEN ∪ {MOVEGEN(N)\{OPEN ∪ CLOSED}} (to front)
    End if
End Loop
```

PROBLEM-I

- Find the solution to the following 8- puzzle problem using DFS, BFS algorithms:

Initial State

2		3
1	8	4
7	6	5

Final State

1	2	3
8		4
7	6	5

Solution Problem-I

- The MOVEGEN function for the 8 –puzzle problem is that we can move the blank tile left, right, up or down. Let the order in which the operators are applied be:

  MOVE BLANK LEFT

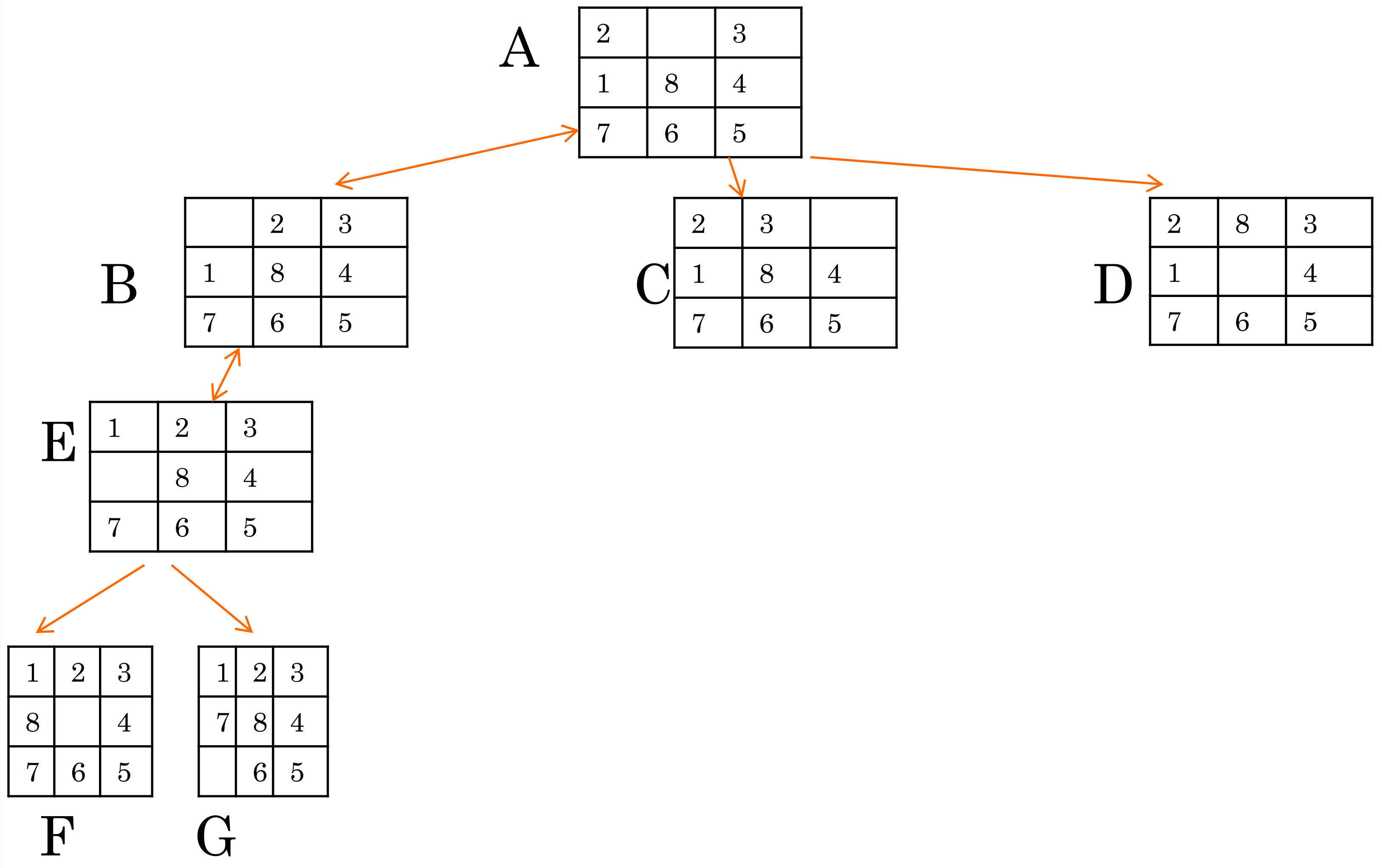
  MOVE BLANK RIGHT

  MOVE BLANK UP

  MOVE BLANK DOWN

Solution- Problem I

- The search space for DFS:



Solution- Problem I Contd……

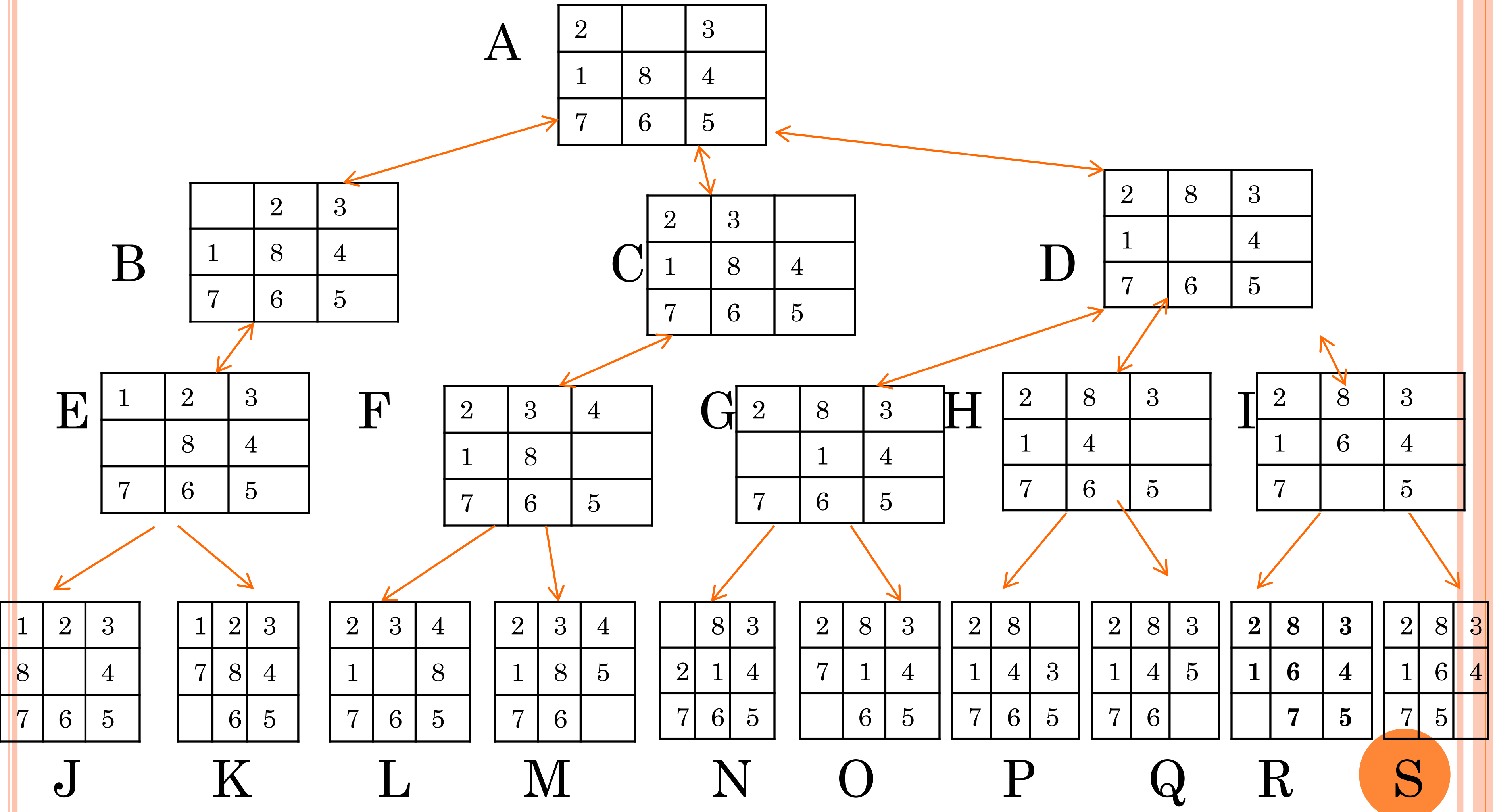
DFS : Initial State – A

Step	OPEN	CLOSED
1	{(A,NIL)}	{ }
2	{(B,A),(C,A),(D,A)}	{(A,NIL)}
3	{(E,B)(C,A),(D,A)}	{(A,NIL),(B,A)}
4	{(F,E)(G,E)(C,A),(D,A)}	{(A,NIL),(B,A),(E,B)}
5	{(G,E)(C,A),(D,A)}	{(A,NIL),(B,A),(E,B)(F,E)}

Therefore DFS solution is $A \rightarrow B \rightarrow E \rightarrow F$

Solution- Problem I

- The search space for BFS is:



Solution- Problem I Contd……

BFS : Initial State – A

Step	OPEN	CLOSED
1	{(A,NIL)}	{ }
2	((B,A)(C,A)(D,A)}	{(A,NIL)}
3	{(C,A)(D,A)(E,B)}	{(A,NIL),(B,A)}
4	{(D,A)(E,B)(F,C)}	{(A,NIL),(B,A)(C,A)}
5	{(E,B)(F,C)(G,D)(H,D)(I,D)}	{(A,NIL),(B,A)(C,A)(D,A)}
6	{(F,C)(G,D)(H,D)(I,D)(J,E)(K,E)}	{(A,NIL),(B,A)(C,A)(D,A)(E,B)}
7	{(G,D)(H,D)(I,D)(J,E)(K,E)(L,F)(M, F)}	{(A,NIL),(B,A)(C,A)(D,A)(E,B)(F,C)}
8	{(H,D)(I,D)(J,E)(K,E)(L,F)(M,F)(N, G)(O,G)}	{(A,NIL),(B,A)(C,A)(D,A)(E,B)(F,C)(G,D)}
9	{(I,D)(J,E)(K,E)(L,F)(M,F)(N,G)(O, G)(P,H)(Q,H)}	{(A,NIL),(B,A)(C,A)(D,A)(E,B)(F,C)(G,D)(H,D)}
10	{(J,E)(K,E)(L,F)(M,F)(N,G)(O,G)(P, H)(Q,H)(R,I)(S,I)}	{(A,NIL),(B,A)(C,A)(D,A)(E,B)(F,C)(G,D)(H,D)(I,D)}
11	{(K,E)(L,F)(M,F)(N,G)(O,G)(P,H)(Q ,H)(R,I)(S,I)}	{(A,NIL),(B,A)(C,A)(D,A)(E,B)(F,C)(G,D)(H,D)(I,D)(J,E)}

SOLUTION- PROBLEM I

- Therefore BFS traversal is A→B →C →D →E →F →G →H →I →J
- Path A → B → E → J

Performance Evaluation of BFS and DFS

- **Completeness:**

  In case of finite search tree with fixed branching factor (number of children a node have), both DFS and BFS find the solution and are hence complete.

  In case of infinite search trees, a DFS algorithm is caught in blind alley i.e. searching at a greater depth in the left path of the search tree while the solution may lie on the right part.

  BFS can find a solution even in infinite graph but take lot of time and memory.

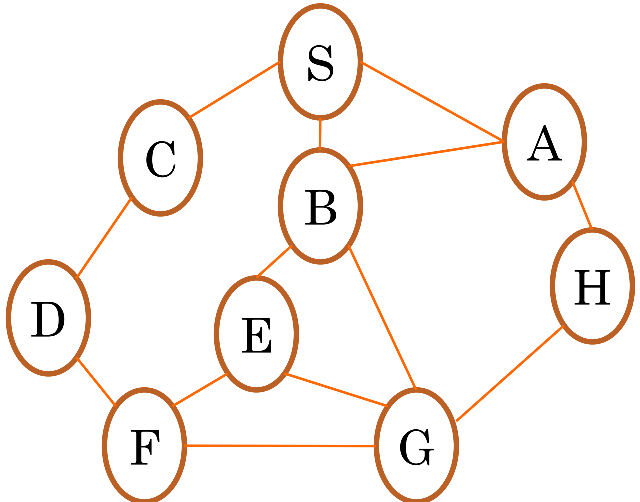
Performance Evaluation of BFS and DFS Contd…

- **Optimality:**

  Since BFS algorithm traverse level by level and finds the goal node which is nearest to the node. Hence the solution of BFS is always optimal (provided the path cost is constant).

  DFS, on the other hand, may not give the optimal solution (with least path cost).

  For instance, in the state space search shown below DFS would return path from start state S to goal state G of 4 unit cost (S → C → D → F → G) where as BFS would return the optimal path of 2 units (S→B →G).

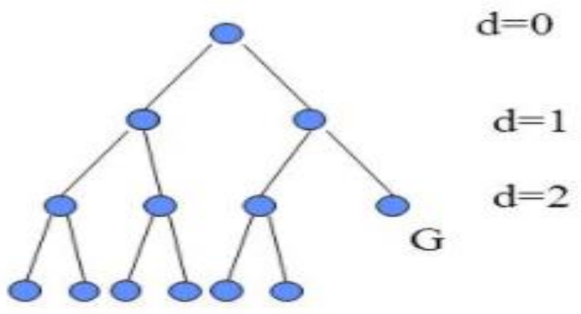


Performance Evaluation of BFS and DFS Contd...

- **Time Complexity**: Consider a search space which grows exponentially with fixed branching factor b (worse case scenario).
- If the solution lies at depth d at the right most node , then the number of nodes generated would be same for both DFS and BFS and are given by:

  $1+b+b^2+b^3+.............(b^{d+1}-b) = O(b^{d+1})$

  (1 node at level 0, b nodes at level 1, b^2 nodes at level 2,….b^d nodes at level d, and $(b^{d+1}-b)$ nodes at level d+1 (as the goal node at depth d would not be expanded)



Performance Evaluation of BFS and DFS Contd…

- In case, the time complexity is measured as the number of nodes expanded, then time complexity is given by:

  $1+b+b^2+b^3+\ldots\ldots\ldots\ldots+b^d=O(b^d)$

- In average case, DFS would expand less number of nodes. The time taken by BFS to DFS is (b+1) to b. Thus BFS takes slightly more time than DFS.

Performance Evaluation of BFS and DFS Contd...

- **Space Complexity**: In case of BFS, the OPEN grows by a factor of b at each level.

  Therefore, if the solution lies at depth d, then space complexity is O(b^{d+1}) as all the nodes of the current level needs to be stored.

  In case of DFS, the OPEN grows linearly. At each level the DFS algorithm adds b nodes. It only stores current nodes of the current path. Therefore space complexity is O(bd).

  Depth-first search would require **12 kilobytes instead of 111 terabytes memory in case of BFS** at **depth *d=12, b=10***,*a factor of* ***10 billion times less space.***

Difference between DFS and BFS

Depth First Search	Breadth First Search
1. Explores all the nodes in the depth first fashion.	1. Explores all the nodes in the breadth first fashion
2. May caught in blind alley i.e. searching at a greater depth in the left path of the search tree while the solution may lie on the right part.	2. Never caught at blind alley because it explores all the nodes at the lower depth before moving to the next node.
3. It uses less space because only current nodes at the current path need to be stored.	3. It uses more space as all the nodes of the current level needs to be stored. The space increases with increase in branching factor.
4. It may find the first solution which may not be optimal.	4. It always finds the optimal solution (if the path cost is uniform).

DEPTH-LIMITED SEARCH

- Depth Limited Search is a variant of DFS.
- It solves the problem of blind alley of depth-first search **by imposing a cut-off** on the maximum depth of a path i.e. it does not apply DFS algorithm beyond a limit (cut-off).
- **Depth-limited search is not complete because solution may lie beyond the cut-off depth.**
- **It does not guarantee** to find the shortest solution first: **depth-limited search is also not optimal.**
- The time and space complexity of depth-limited search is similar to depth-first search. It takes ***$O(b^l)$ time and $O(bl)$ space***, *where l* is the depth limit.

DEPTH- FIRST SEARCH WITH ITERATIVE DEEPENING (DFS-ID)

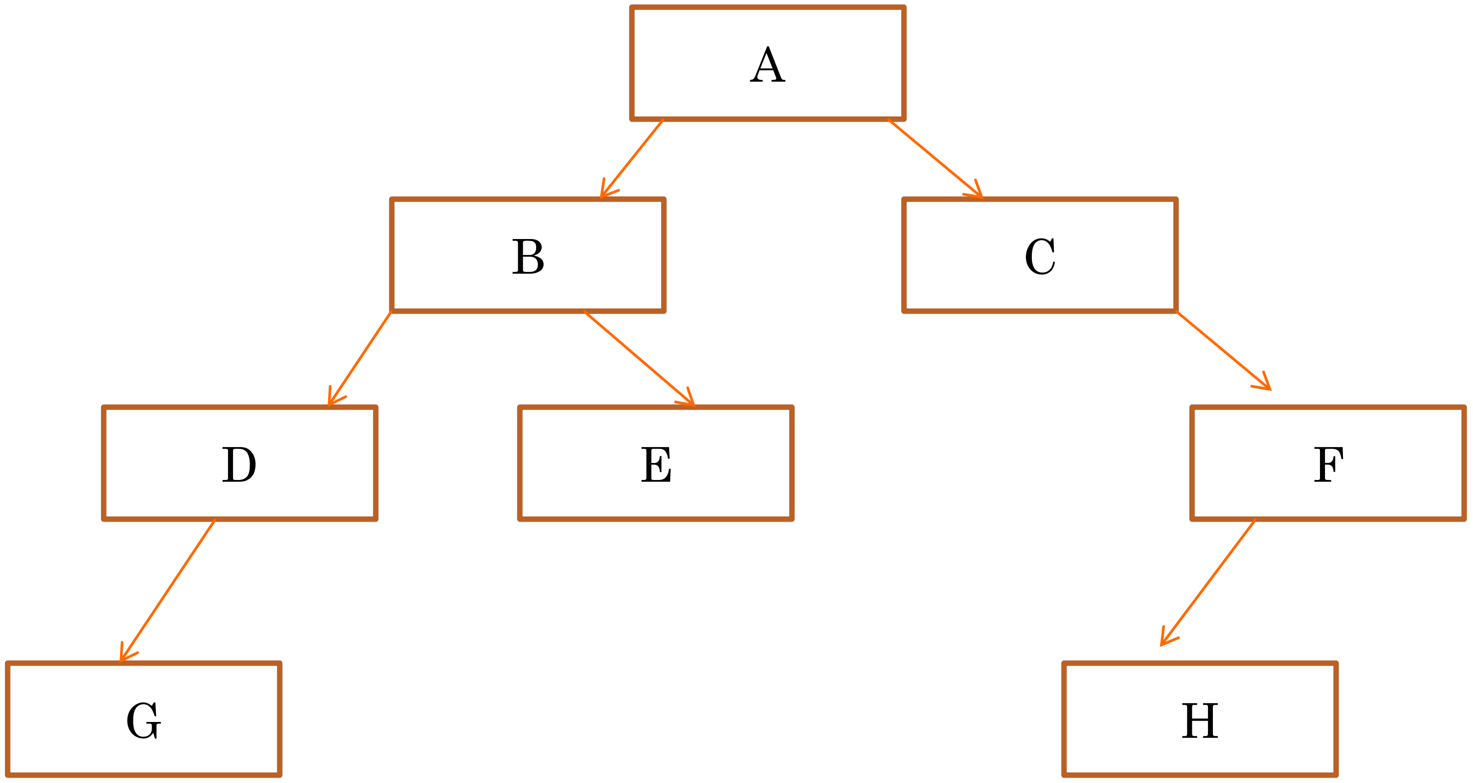
- The major advantage of BFS algorithm is its optimality and that of DFS is the space complexity.
- DFS-ID combines depth-first search's space-efficiency and breadth-first search's optimality.
- DFS-ID calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically it performs DFS in a BFS fashion.
- The search is started with depth one. If the solution is not found, then the level of depth is increased by one and then the search is performed again in depth first fashion.
- Thus, deepening of depth continues till a solution is found.

DFS-ID ALGORITHM

```
DFS-ID (problem) returns success or failure
for depth= 0 to ∞
        result = DEPTH-LIMITED-SEARCH(problem,depth) return
        result
    end for
```

Example- DFS-ID

- Consider the following search tree with initial state A and goal node G.



Example- DFS-ID Contd…..

- **At Depth =1**

  The starting node is A. The initial OPEN is {(A,NIL)}. Since it is not a goal node, the queue will be empty and the DFS-ID will stop.

- **At Depth=2**

Step	OPEN	CLOSED
1	{(A,NIL))	{ }
2	{(B,A),(C,A)}	{(A,NIL))
3	{(C,A)}	{(A,NIL),B,A)}
4	{ }	{(A,NIL),B,A), (C,A)}

Example- DFS-ID Contd…..

- **At Depth =3**

Step	OPEN	CLOSED
1	{(A,NIL)}	{ }
2	{(B,A),(C,A)}	{(A,NIL)}
3	{(D,B),(E,B),(C,A)}	{(A,NIL),(B,A)}
4	{(E,B),(C,A)}	{(A,NIL),(B,A)(D,B)}
5	{(C,A)}	{(A,NIL),(B,A)(D,B),(E,B)}
6	{(F,C) }	{(A,NIL),(B,A)(D,B),(E,B),(C,A)}
7	{ }	{(A,NIL),(B,A)(D,B),(E,B),(C,A),(F,C)}

Example- DFS-ID Contd…..

- **At Depth =4**

Step	OPEN	CLOSED
1	{(A,NIL)}	{ }
2	{(B,A),(C,A)}	{(A,NIL)}
3	{(D,B),(E,,B),(C,A)}	{(A,NIL),(B,A)}
4	{(G,D),(E,B)(C,A)}	{(A,NIL),(B,A)(D,B)}
5	{(E,B)(C,A)}	{(A,NIL),(B,A)(D,B)(G,D)}

- So, the depth first traversal with iterative deepening is:

  A→B → D →G

Performance of DFS-ID

- **Completeness:** The DFS-ID is complete for finite search space. For infinite search spaces also, it will never stuck into blind alley as it checks all the nodes of the previous depth before moving to next depth.
- **Optimality:** Since DFS-ID iteratively deepens into depth after checking all the nodes at the lower levels. Therefore, the solution is optimal.
- **Space Complexity:** DFS-ID at each level traverse nodes in DFS fashion, therefore, the OPEN grows linearly by adding b nodes at each level.

  Thus, space complexity of DFS-ID is same as that of DFS i.e. O(bd)

Performance of DFS-ID Contd…

- **Time Complexity:** Consider a search space which grows exponentially with fixed branching factor b (worse case scenario).
- If the solution lies at depth d at the right most node , then the root node is generated d times, b nodes at level 1 are generated (d-1) times, b^2 nodes are generated (d-2) times and so on. b^d nodes at level d are generated 1 time only.

  Time Complexity:

  $(d)1+(d-1)b+(d-2)b^2+\ldots\ldots\ldots\ldots(1)b^d=O(b^d)$

  BFS also generates nodes at level d+1 with time complexity $O(b^{d+1})$ where as DFS-ID do not.

  Hence in worse case scenario, DFS-ID is faster than BFS.

PROBLEM-I

- Find the solution to the following 8- puzzle problem using DFS-ID algorithms:

Initial State

2		3
1	8	4
7	6	5

Final State

1	2	3
8		4
7	6	5

Solution Problem-I

- The MOVEGEN function for the 8 –puzzle problem is that we can move the blank tile left, right, up or down. Let the order in which the operators are applied be:

  MOVE BLANK LEFT

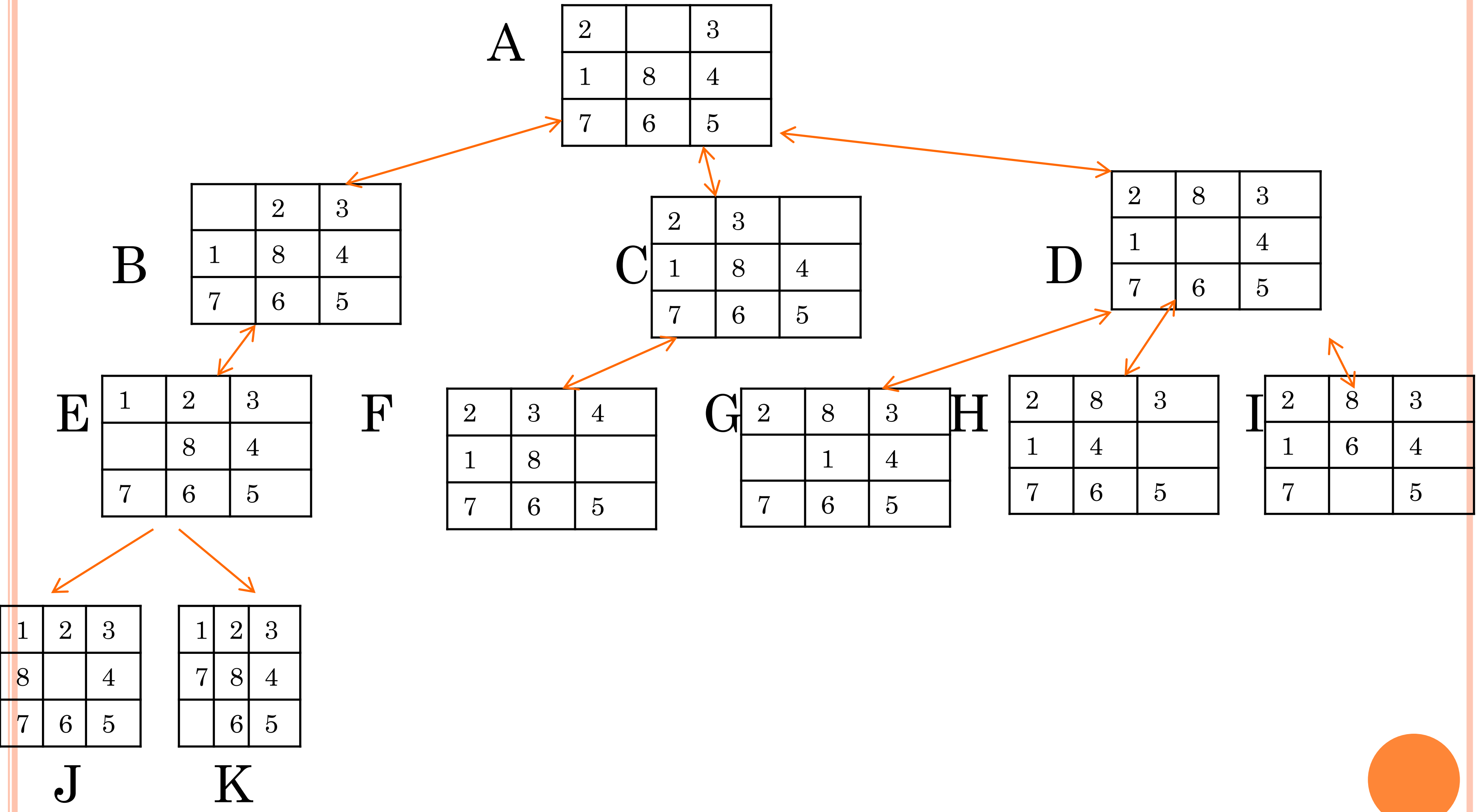
  MOVE BLANK RIGHT

  MOVE BLANK UP

  MOVE BLANK DOWN

Solution- Problem I

- The search space for DFS-ID is:



Solution- Problem I Contd……

- **DFS-ID** Initial State- A,
- **At Depth=1**

  The starting node is A. The initial OPEN is {(A,NIL)}. Since it is not a goal node, the queue will be empty and the DFS-ID will stop.
- **At Depth=2**

Step	OPEN	CLOSED
1	{(A,NIL)}	{ }
2	{(B,A)(C,A)(D,A)}	{(A,NIL)}
3	{(C,A)(D,A)}	{(A,NIL)(B,A)}
4	{(D,A)}	{(A,NIL)(B,A)(C,A)}
5	{ }	{(A,NIL)(B,A)(C,A)(D,A)}

Solution- Problem I Contd……

- **At Depth =3**

Step	OPEN	CLOSED
1	{(A,NIL)}	{ }
2	{(B,A),(C,A),(D,A)}	{(A,NIL)}
3	{(E,B),(C,A),(D,A)}	{(A,NIL)(B,A)}
4	{(C,A)(D,A)}	{(A,NIL)(B,A)(E,B)}
5	{(F,C)(D,A)}	{(A,NIL)(B,A)(E,B)(C,A)}
6	{(D,A)}	{(A,NIL)(B,A)(E,B)(C,A)(F,C)}
7	{(G,D)(H,D)(I,D)}	{(A,NIL)(B,A)(E,B)(C,A)(F,C)(D,A)}
8	{(H,D)(I,D)}	{(A,NIL)(B,A)(E,B)(C,A)(F,C)(D,A)(G,D)}
9	{(I,D)}	{(A,NIL)(B,A)(E,B)(C,A)(F,C)(D,A)(G,D)(H,D)}
10	{ }	{(A,NIL)(B,A)(E,B)(C,A)(F,C)(D,A)(G,D)(H,D)(I,D)}

Solution- Problem I Contd……

- **At Depth =4**

Step	OPEN	CLOSED
1	{(A,NIL)}	{ }
2	{(B,A),(C,A),(D,A)}	{(A,NIL)}
3	{(E,B),(C,A),(D,A)}	{(A,NIL)(B,A)}
4	{(J,E),(K,E),(C,A),(D,A)}	{(A,NIL)(B,A)(E,B)}
5	{(K,E),(C,A),(D,A)}	{(A,NIL)(B,A)(E,B)(J,E)}

- Therefore DFS -ID traversal and path is A→B →E →J →K

Uniform Cost Search (UCS)

- Instead of expanding nodes in order of their depth from the root, uniform-cost search expands nodes in order of their cost from the root.
- At each step, the next node n to be expanded is one whose cost g(n) is lowest where g(n) is the sum of the edge costs from the root to node n.
- OPEN is thus maintained as a priority queue i.e. the nodes in OPEN are stored in the priority of their cost from the root node (g(n)).
- If all the edges in the search graph do not have the same cost then breadth-first search generalizes to uniform-cost search.

Uniform Cost Search (UCS)

- In UCS, whenever a node is expanded, it is checked whether the node is a newly generated node or it is already in OPEN or CLOSED.
- If it is a newly generated node then it is added in OPEN.
- If it is already in OPEN/CLOSED, then it is checked that the new path cost to reach this node is less then earlier or not. If new cost is more then the node is rejected, else this new path is updated.
- In case, the node is in CLOSED list and the new path cost is better then the cost is updated and the same improvement is propagated.

UCS ALGORITHM

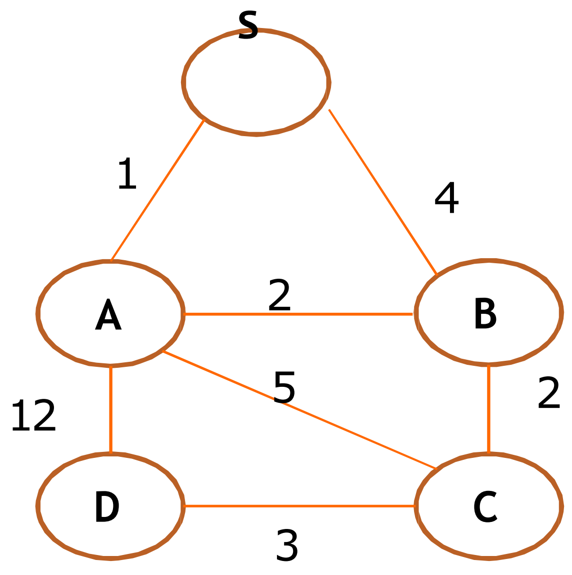
```
Insert start node (S,Φ,gS)) to OPEN
Loop until OPEN is empty or success is returned
        (n,p,g(n)) ← remove-head(OPEN) and add it CLOSED
        If n is a goal node return success and path to n.
        else
            successors← MOVEGEN(n)
            for each successor m do switch
                case
                    case 1: if m ∉ OPEN and m ∉ CLOSED
                            compute g(m) =g(n) +cost (n,m)
                            add (m,n,g(m)) to OPEN according to priority of g(m)
                    case 2: if m ε OPEN
                                compute newg(m)=g(n)+cost(n,m)
                            if newg(m) < g(m) then
                                    update g(m) =newg(m)
                                    update (m, n, g(m)) to OPEN according to priority of g(m)
                            end if
                    case 3: if m ε CLOSED
                            compute newg(m)=g(n)+cost(n,m)
                            if newg(m) < g(m) then
                                    update g(m) =newg(m)
                                    update (m, n, g(m)) to CLOSED according to priority of g(m)
                                    propogateimprovement(m)
                            end if
            end for
        end if
end loop
```

UCS Algorithm Contd….

```
propagateimprovement(m)
for each successor s of m
        compute newg(s)=g(m) +cost(m,s)
        if newg(s) < g(s) then
                update g(s)=newg(s)
                update (s,m,g(s)) in OPEN or CLOSED
                if s ε CLOSED then
                        propogateimprovement(s)
                end if
        end if
end for
```

UCS EXAMPLE

- For the search space shown below, find the optimal path from S to D using UCS algorithm



UCS EXAMPLE

g(S)=0

Add (S,ϕ,0) to OPEN and CLOSED is empty

Iteration	OPEN	CLOSED
0	{(S,ϕ,0)}	{}

Iteration I

Remove head node (S,ϕ,0) from OPEN and add to CLOSED.

Since S is not goal node, therefore successors of S i.e. A and B are produced (case 1: both are new not in OPEN and CLOSED)

Successors of S:

A :g(A)=g(S)+cost(S,A) =0+1 =1

B :g(B)=g(S)+cost(S,B) =0+4 =4

Iteration	OPEN	CLOSED
0	{(S,ϕ,0)}	{}
1	{(A,S,1) (B,S,4)}	{(S,ϕ,0)}

UCS EXAMPLE

Iteration II

Remove head node (A,S,1) from OPEN and add to CLOSED.

Since A is not goal node, therefore successors of A i.e. S, B , C, and D are produced (for S it is case 3 as it is already in CLOSED, for B it is case 2 as it is already in OPEN and for C and D it is case 1 which is not in OPEN and CLOSED)

Successors of A:

S :newg(S)=g(A)+cost(A,S) =1+1=2

since newg(S) is not less than g(S), so this successor is ignored

B :newg(B)=g(A)+cost(A,B) = 1+2=3

since newg(B) is less than g(B), so update (B,A,3) in OPEN

C: g(C)=g(A) +cost(A,C) =1 +5=6,

Add (C,A,6) to OPEN

D: g(D)= g(A) +cost(A,D)=1+12=13

Add (D,A,13) to OPEN

Iteration	OPEN	CLOSED
0	{(S,ϕ,0)}	{}
1	{ (A,S,1) (B,S,4)}	{(S,ϕ,0)}
2	{(B,A,3) (C,A,6) (D,A,13)}	{(S,ϕ,0) (A,S,1)}

UCS EXAMPLE

Iteration III

Remove head node (B,A,3) from OPEN and add to CLOSED.

Since B is not goal node, therefore successors of B i.e. S, A , C are produced (for S and A it is case 3 as it is already in CLOSED, for C it is case 2 as it is already in OPEN)

Successors of B:

S :newg(S)=g(B)+cost(B,S) =3+4=7

since newg(S) is not less than g(S), so this successor is ignored

A :newg(A)=g(B)+cost(B,A) = 3+2=5

since newg(A) is not less than g(A) , so this successor is ignored.

C: newg(B)=g(B) +cost(B,C) =3+2=5

since newg(C) is less than g(C) , so update (C,B,5) in OPEN

Iteration	OPEN	CLOSED
0	{(S,ϕ,0)}	{}
1	{ (A,S,1) (B,S,4)}	{(S,ϕ,0)}
2	{(B,A,3) (C,A,6) (D,A,13)}	{(S,ϕ,0) (A,S,1)}
3	{(C,B,5) (D,A,13)}	{(S,ϕ,0) (A,S,1) (B,A,3)}

UCS EXAMPLE

Iteration IV

Remove head node (C,B,5) from OPEN and add to CLOSED.

Since C is not goal node, therefore successors of C i.e. B,A and D are produced (for B and A it is case 3 as it is already in CLOSED, for D it is case 2 as it is already in OPEN)

Successors of C:

B :newg(B)=g(C)+cost(C,B) =5+4=9

since newg(B) is not less than g(B), so this successor is ignored

A :newg(A)=g(C)+cost(C,A) = 5+5=5

since newg(A) is not less than g(A) , so this successor is ignored.

D: newg(D)=g(C) +cost(C,D) =5+ 3=8

since newg(D) is not less than g(D) , so update (D,C,8) in OPEN

Iteration	OPEN	CLOSED
0	{(S,ϕ,0)}	{}
1	{ (A,S,1) (B,S,4)}	{(S,ϕ,0)}
2	{(B,A,3) (C,A,6) (D,A,13)}	{(S,ϕ,0) (A,S,1)}
3	{(C,B,5) (D,A,13)}	{(S,ϕ,0) (A,S,1) (B,A,3)}
4	{(D,C,8)}	{(S,ϕ,0) (A,S,1) (B,A,3)(C,B,5)}

UCS EXAMPLE

Iteration V

Remove head node (D,C,8) from OPEN and add to CLOSED.

Since D is goal node, therefore algorithm will terminate

Path is S→A → B→C → D with path cost 8

Iteration	OPEN	CLOSED
0	{(S,ϕ,0)}	{}
1	{ (A,S,1) (B,S,4)}	{(S,ϕ,0)}
2	{(B,A,3) (C,A,6) (D,A,13)}	{(S,ϕ,0) (A,S,1)}
3	{(C,B,5) (D,A,13)}	{(S,ϕ,0) (A,S,1) (B,A,3)}
4	{(D,C,8)}	{(S,ϕ,0) (A,S,1) (B,A,3)(C,B,5)}
5	{ }	{(S,ϕ,0) (A,S,1) (B,A,3)(C,A,6)(D,C,8)}

PERFORMANCE OF UCS

- **Completeness:** UCS is complete as it traverse level by level in the search tree as like BFS.
- **Optimality:** UCS is optimal as it chooses the minimum cost path at each step.
- **Time Complexity:** The time complexity of UCS is of the order of $O(b^{c}/m)$ where c is the cost of the optimal path and m is the minimum edge length.
- **Space Complexity:** The space complexity of UCS is same as that of BFS i.e. $O(b^{d+1})$

BIDIRECTIONAL SEARCH

- Bidirectional search simultaneously finds both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle (as shown in figure below).

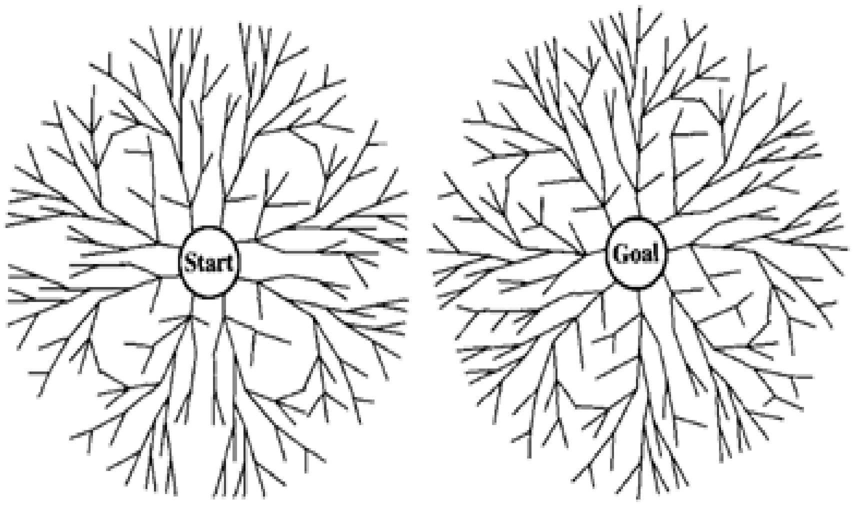


Figure A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Bidirectional Search

- Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. **The search terminates when two graphs intersect.**
- Bidirectional search would be complete and optimal if BFS is used during both forward and backward search.
- It reduces the amount of required exploration dramatically.

  Suppose if branching factor of tree is **b** and distance of goal vertex from source is **d**, then the normal BFS/DFS searching complexity is $O(b^d)$.

  On the other hand, if we execute two search operation then the complexity would be $O(2b^{d/2}) = O(b^{d/2})$ which is far less than $O(b^d)$.

Issues in Bidirectional Search

- Several issues need to be addressed before the algorithm can be implemented.
  - The main question is, what does it mean to search backwards from the goal?

    We need to define the **predecessors** of a node *n to be all those nodes that have n as a successor*.

    When all operators are reversible, the predecessor and successor sets are identical; some problems, however, calculating predecessors can be very difficult.
  - What if there are many possible goal states?
  - There must be an efficient way to check each new node to see if it already appears in the search tree of the other half of the search.
  - We need to decide what kind of search is going to take place in each half.