Disjoint Set Operations

Disjoint Set Operations

Set:

A set is a collection of distinct elements. The Set can be represented, for examples, as S1={1,2,5,10}.

Disjoint Sets:

The disjoints sets are those do not have any common element.
For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

1. Union
2. Find

Disjoint set Union:

If Si and Sj are tow disjoint sets, then their union Si U Sj consists of all the elements x such that x is in Si or Sj.

Example:

S1={1,7,8,9} S2={2,5,10}
S1 U S2={1,2,5,7,8,9,10}

Find:

Given the element I, find the set containing i.

Example:

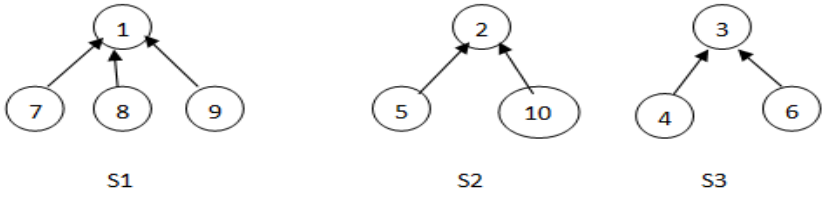
S1={1,7,8,9} S2={2,5,10} s3={3,4,6}
Then,
Find(4)= S3 Find(5)=S2 Find97)=S1

Set Representation:

The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

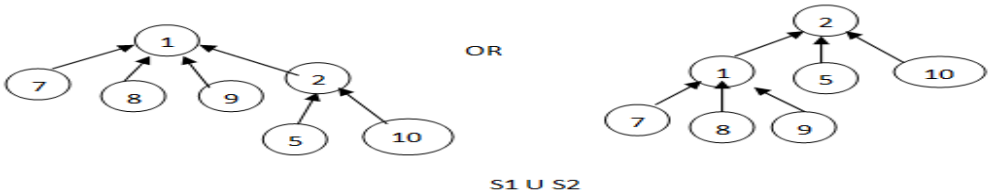
Example:

S1={1,7,8,9} S2={2,5,10} s3={3,4,6}
Then these sets can be represented as



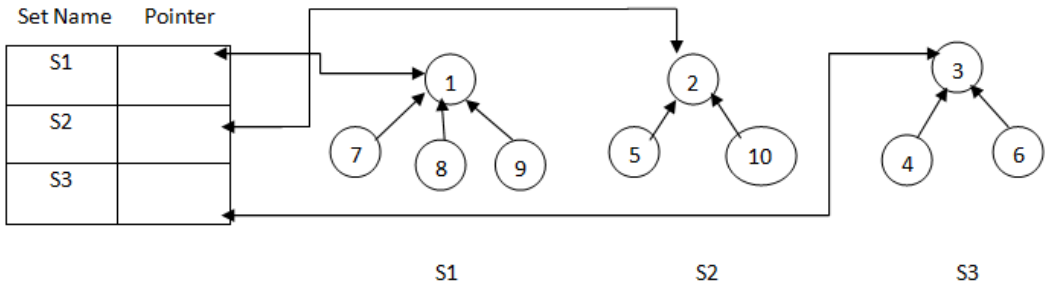
Disjoint Union:

To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.



Find:

To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.

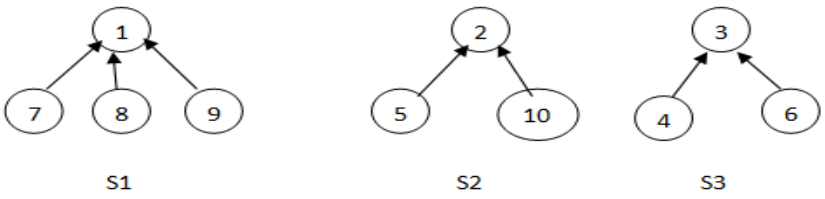


Union and Find Algorithms:

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:

For the following sets the array representation is as shown below.



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

Algorithm SimpleUnion(i,j)

{

```
        P[i]:=j;
```

}

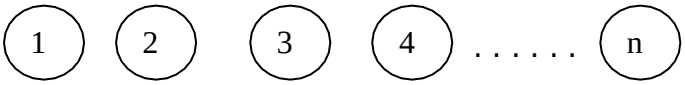
Algorithm for find operation:

The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at I until it reaches a node with parent value -1.

```
Algorithms SimpleFind(i)
{
        while( P[i]≥0) do i:=P[i];
        return i;
}
```

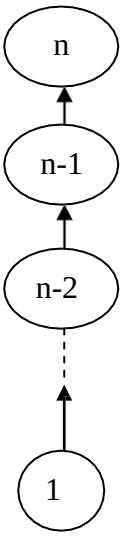
Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets



Then if we want to perform following sequence of operations Union(1,2) , Union(2,3)……. Union(n-1,n) and sequence of Find(1), Find(2)…… Find(n).

The sequence of Union operations results the degenerate tree as below.

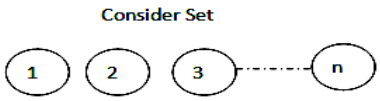


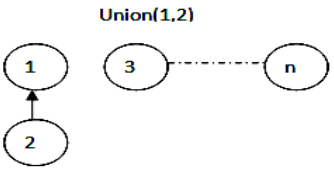
Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time O(n). And for the sequence of Find operations it will take time complexity of $O\left(\sum_{i=1}^{n} i\right) = O(n^2)$.

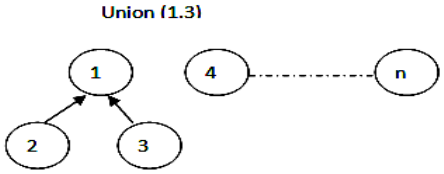
We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

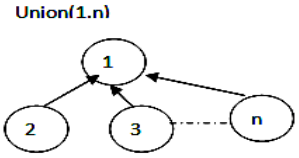
Weighting rule for Union:

If the number of nodes in the tree with root I is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.









To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then count[i] equals to number of nodes in tree with root i.
Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

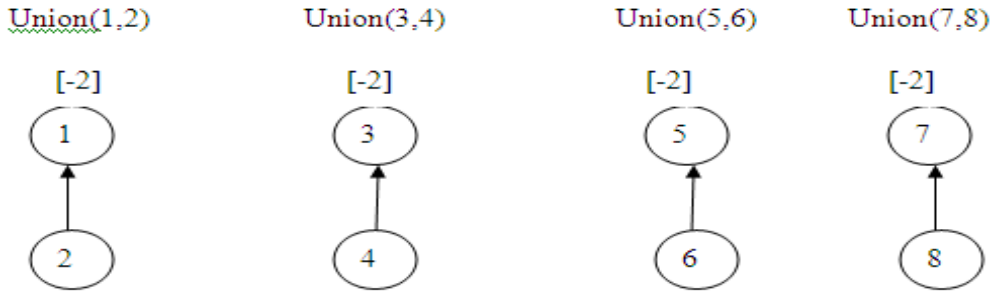
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j , i≠j using the weighted rule
// P[i]=-count[i] and p[j]=-count[j]
{

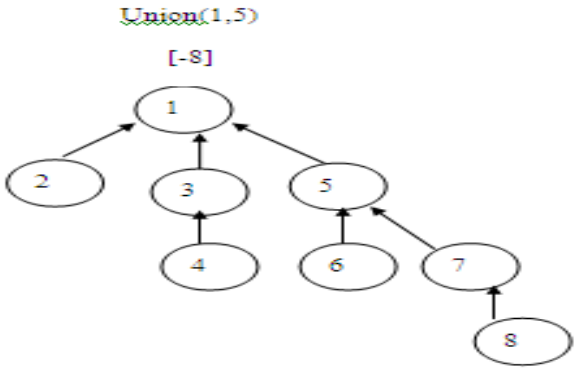
```
        temp:= P[i]+P[j];
        if (P[i]>P[j]) then
        {
          // i has fewer nodes
             P[i]:=j;
             P[j]:=temp;
         }
         else
         {
          // j has fewer nodes
             P[j]:=i;
             P[i]:=temp;
         }
```

}

Collapsing rule for find:

If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i]. Consider the tree created by WeightedUnion() on the sequence of 1≤i≤8. Union(1,2), Union(3,4), Union(5,6) and Union(7,8)





Now process the following eight find operations

Find(8), Find(8)……Find(8)

If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24 moves .
When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i
// use the collapsing rule to collapse all nodes from i to root.
{
        r:=i;
        while(P[r]>0) do r:=P[r]; //Find root
         while(i≠r)
         {
                 //reset the parent node from element i to the root
                 s:=P[i];
                 P[i]:=r;
                 i:=s;
         }
}
```

Chapter 6

BASIC TRAVERSAL AND SEARCH TECHNIQUES

Search means finding a path or traversal between a start node and one of a set of goal nodes. Search is a study of states and their transitions.

Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal.

Techniques for Traversal of a Binary Tree:

A binary tree is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and remaining elements (if any) are partitioned into two binary trees, which are called the left and right subtrees.

There are three common ways to traverse a binary tree:

1. Preorder
2. Inorder
3. postorder

In all the three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among the three orders comes from the difference in the time at which a node is visited.

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for preorder traversal is as follows:

```
treenode = record
{
        Type   data;                                    //Type is the data type of data.
        Treenode *lchild; treenode *rchild;
}
```

```
algorithm inorder (t)
// t is a binary tree. Each node of t has three fields: lchild, data, and rchild.
{
        if t ≠ 0 then
        {
                inorder (t → lchild);
                visit (t);
                inorder (t → rchild);
        }
}
```

Preorder Traversal:

In a preorder traversal, each node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
Algorithm Preorder (t)
// t is a binary tree. Each node of t has three fields; lchild, data, and rchild.
{
        if t ≠ 0 then
        {
                visit (t);
                Preorder (t → lchild);
                Preorder (t → rchild);
        }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

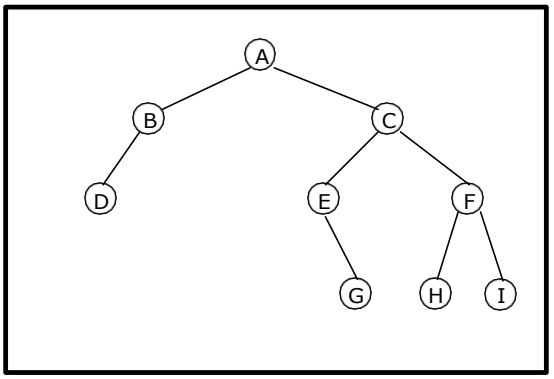
The algorithm for preorder traversal is as follows:

```
Algorithm Postorder (t)
// t is a binary tree. Each node of t has three fields : lchild, data, and rchild.
{
        if t ≠ 0 then
        {
                Postorder (t → lchild);
                Postorder (t → rchild);
                visit(t);
        }
}
```

Examples for binary tree traversal/search technique:

Example 1:

Traverse the following binary tree in pre, post and in-order.



Bin a ry T re e

Preordering of the vertices:
A, B, D, C, E, G, F, H, I.

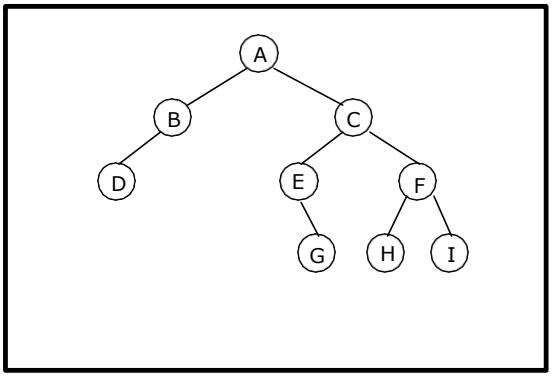
Postordering of the vertices:
D, B, G, E, H, I, F, C, A.

Inordering of the vertices:
D, B, A, E, G, C, H, F, I

Pre, Post and In-order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



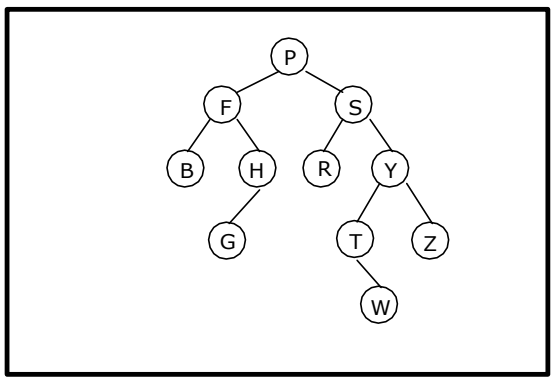
Bin a ry T re e

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



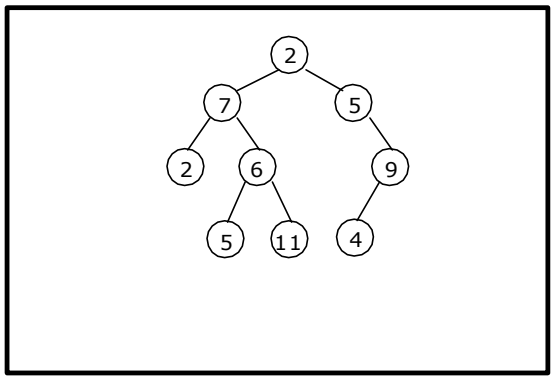
Bin a ry T re e

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
  P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



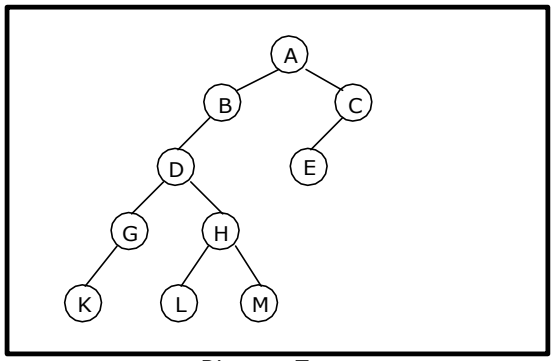
Bin a ry T re e

- Preo rder trav ersal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Posto rder trav arsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Ino rder trav arsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9
- Le vel o rder trav ersal yields:
  2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 5:

Traverse the following binary tree in pre, post, inorder and level order.



Bin a ry T re e

- Preo rder trav ersal yields:
  A, B, D, G , K, H, L, M , C , E
- Posto rder trav arsal yields:
  K, G , L, M , H, D, B, E, C , A
- Ino rder trav arsal yields:
  K, G , D, L, H, M , B, A, E, C
- Le vel o rder trav ersal yields:
  A, B, C , D, E, G , H, K, L, M

Pre, Post, Inorder and level order Traversing

Non Recursive Binary Tree Traversal Algorithms:

At first glance, it appears we would always want to use the flat traversal functions since the use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

The algorithm for inorder Non Recursive traversal is as follows:

```
Algorithm inorder()
{
        stack[1] = 0
        vertex = root
top:    while(vertex ≠ 0)
        {
                push the vertex into the stack
                vertex = leftson(vertex)
        }

        pop the element from the stack and make it as vertex

        while(vertex ≠ 0)
        {
                print the vertex node
                if(rightson(vertex) ≠ 0)
                {
                        vertex = rightson(vertex)
                        goto top
                }
                pop the element from the stack and made it as vertex
        }
}
```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

The algorithm for preorder Non Recursive traversal is as follows:

```
Algorithm preorder( )
{
        stack[1]: = 0
        vertex := root.
        while(vertex ≠ 0)
        {
                print vertex node
                if(rightson(vertex) ≠ 0)
                        push the right son of vertex into the stack.
                if(leftson(vertex) ≠ 0)
                        vertex := leftson(vertex)
                else
                        pop the element from the stack and made it as vertex
        }
}
```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

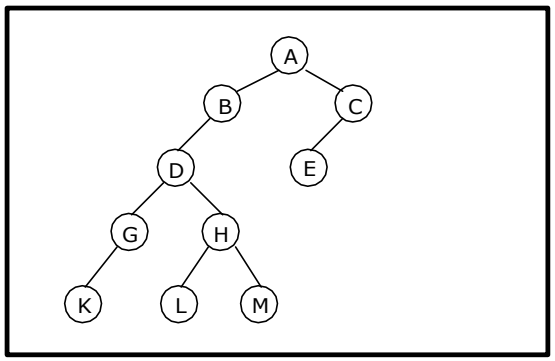
The algorithm for postorder Non Recursive traversal is as follows:

```
Algorithm postorder( )
{
        stack[1] := 0
        vertex := root

 top: while(vertex ≠ 0)
        {
                push vertex onto stack
                if(rightson(vertex) ≠ 0)
                        push -(vertex) onto stack
                vertex := leftson(vertex)
        }
        pop from stack and make it as vertex
        while(vertex > 0)
        {
                print the vertex node
                pop from stack and make it as vertex
        }
        if(vertex < 0)
        {
                vertex := -(vertex)
                goto top
        }
}
```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
H	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A -C B D -H G K		PUSH the left most path of A with a -ve for right sons
	0 A -C B D -H	K G	POP all +ve nodes K and G
H	0 A -C B D	K G	Pop H
	0 A -C B D H -M L	K G	PUSH the left most path of H with a -ve for right sons
	0 A -C B D H -M	K G L	POP all +ve nodes L
M	0 A -C B D H	K G L	Pop M
	0 A -C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A -C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0		Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

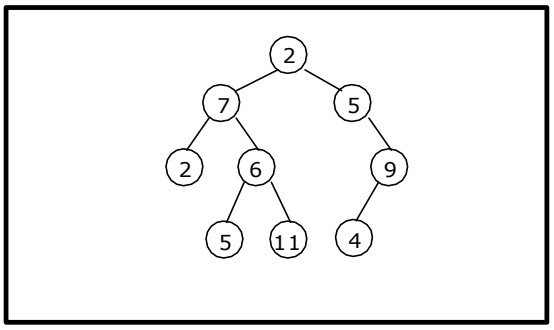
1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields: 2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields: 2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields: 2, 7, 5, 6, 11, 2, 5, 4, 9

Pre, Post and In order Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
11	0 2	2 7 5 6 11	
5	0 5	2 7 5 6 11 2	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	
	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 2 -5 7 -6 2		
2	0 2 -5 7 -6	2	
6	0 2 -5 7	2	
	0 2 -5 7 6 -11 5	2	
5	0 2 -5 7 6 -11	2 5	
11	0 2 -5 7 6 11	2 5	
	0 2 -5	2 5 11 6 7	
5	0 2 5 -9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11	
9	0	2 7 2 6 5 11 5	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty

Techniques for graphs:

Given a graph G = (V, E) and a vertex V in V (G) traversing can be done in two ways.

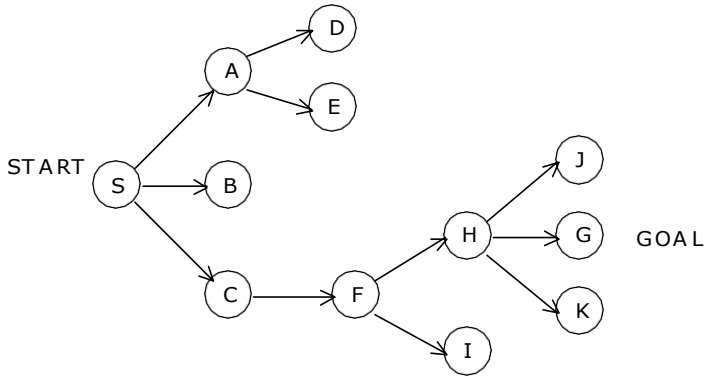
1. Depth first search
2. Breadth first search
3. D-search (Depth Search)

Depth first search:

With depth first search, the start state is chosen to begin, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it "backs up" that means it goes to the immediately previous state or predecessor formally, the state whose successor was 'S' originally.

For example consider the figure. The circled letters are state and arrows are branches.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state. So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.

   To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

2. One more problem is that, the state space tree may be of infinite depth, to prevent consideration of paths that are too long, a maximum is often placed on the depth of nodes to be expanded, and any node at that depth is treated as if it had no successors.

3. We cannot come up with shortest solution to the problem.

Time Complexity:

Let n = |V| and e = |E|. Observe that the initialization portion requires Φ (n) time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n assuming each vertex is reachable from the source). As, each vertex is visited at most once. At each vertex visited, we scan its adjacency list once. Thus, each edge is examined at most twice (once at each endpoint). So the total running time is O (n + e).

Alternatively,

If the average branching factor is assumed as 'b' and the depth of the solution as 'd', and maximum depth m ≥ d.

The worst case time complexity is $O(b^m)$ as we explore b^m nodes. If many solutions exists DFS will be likely to find faster than the BFS.

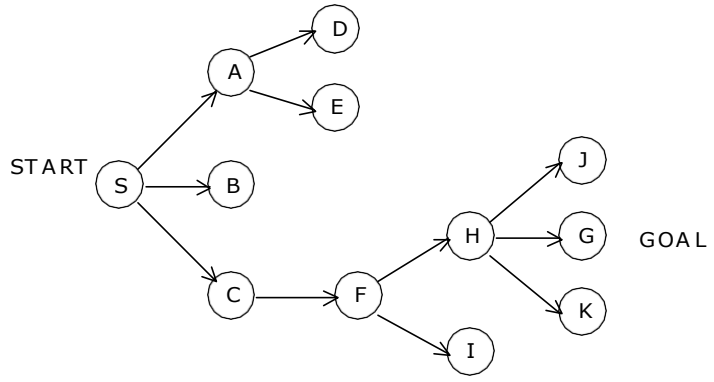
Space Complexity:

We have to store the nodes from root to current leaf and all the unexpanded siblings of each node on path. So, We need to store bm nodes.

Breadth first search:

Given an graph G = (V, E), breadth-first search starts at some source vertex S and "discovers" which vertices are reachable from S. Define the distance between a vertex V and S to be the minimum number of edges on a path from S to V. Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths (where the length of a path = number of edges on the path). Breadth-first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4. So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state. Any time we need a new state, we pick it from the front of the queue and any time we find successors, we put them at the end of the queue. That way we are guaranteed to not try (find successors of) any states at level 'N' until all states at level 'N – 1' have been tried.

Time Complexity:

The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. Let n = |V| and e = |E|. Observe that the initialization portion requires Φ (n) time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n, assuming each vertex is reachable from the source). So, Running time is O (n + e) as in DFS. For a directed graph the analysis is essentially the same.

Alternatively,

If the average branching factor is assumed as 'b' and the depth of the solution as 'd'.

In the worst case we will examine $1 + b + b^2 + b^3 + \ldots + b^d = (b^{d+1} - 1) / (b - 1) = O(b^d)$.

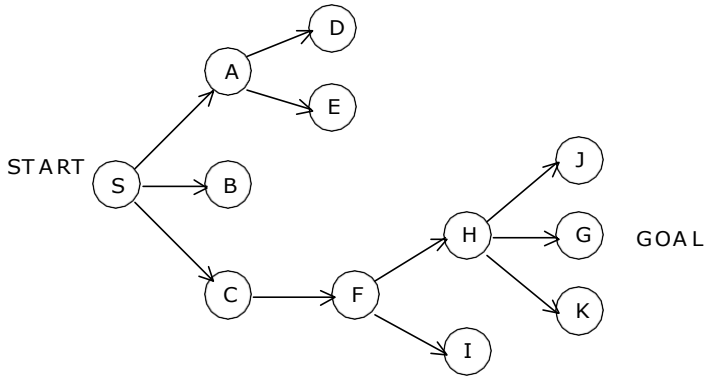
In the average case the last term of the series would be $b^d / 2$. So, the complexity is still $O(b^d)$

Space Complexity:

Before examining any node at depth d, all of its siblings must be expanded and stored. So, space requirement is also $O(b^d)$.

Depth Search (D-Search):

The exploration of a new node cannot begin until the node currently being explored is fully explored. D-search like state space search is called LIFO (Last In First Out) search which uses stack data structure. To illustrate the D-search let us consider the following tree:



The search order for goal node (G) is as follows: S, A, B, C, F, H, I, J, G.

Time Complexity:

The time complexity is same as Breadth first search.

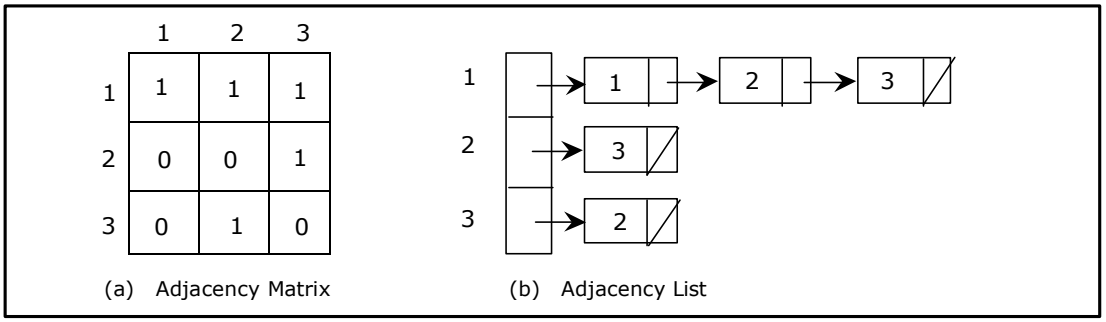
Representation of Graphs and Digraphs by Adjacency List:

We will describe two ways of representing digraphs. We can represent undirected graphs using exactly the same representation, but we will double each edge, representing the undirected edge {v, w} by the two oppositely directed edges (v, w) and (w, v). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

Let G = (V, E) be a digraph with n = |V| and let e = |E|. We will assume that the vertices of G are indexed {1, 2, ... , n}.

$$A[v, w] = \begin{cases} 1 & \text{if } (v,w) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency List: An array Adj [1 . . . . . n] of pointers where for 1 $\leq$ v $\leq$ n, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.



Adjacency matrix and adjacency list

An adjacency matrix requires **Θ** (n^2) storage and an adjacency list requires **Θ** (n + e) storage.

Adjacency matrices allow faster access to edge queries (for example, is (u, v) $\in$ E) and adjacency lists allow faster access to enumeration tasks (for example, find all the vertices adjacent to v).

Depth First and Breadth First Spanning Trees:

BFS and DFS impose a tree (the BFS/DFS tree) along with some auxiliary edges (cross edges) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. Trees are much more structured objects than graphs. For example, trees break up nicely into subtrees, upon which subproblems can be solved recursively. For directed graphs the other edges of the graph can be classified as follows:

Back edges: (u, v) where v is a (not necessarily proper) ancestor of u in the tree. (Thus, a self-loop is considered to be a back edge).

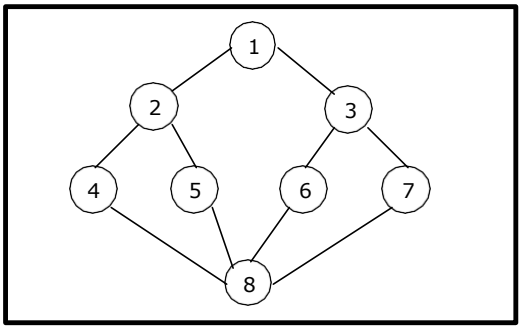
Forward edges: (u, v) where v is a proper descendent of u in the tree.

Cross edges: (u, v) where u and v are not ancestors or descendents of one another (in fact, the edge may go between different trees of the forest).

Depth first search and traversal:

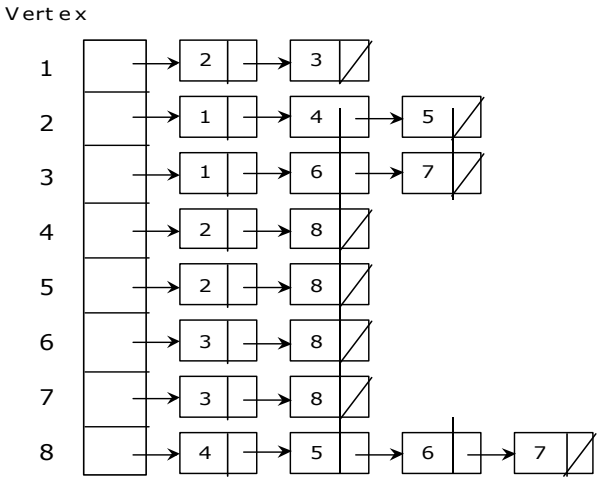
Depth first search of undirected graph proceeds as follows. The start vertex V is visited. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'u' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

Let us consider the following Graph (G):

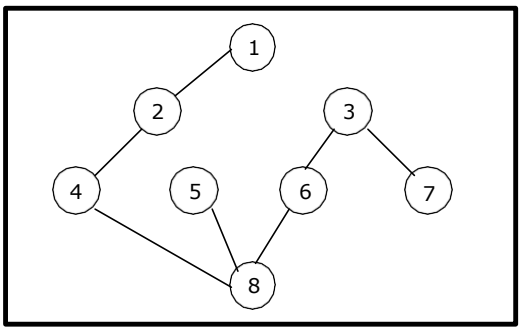


Graph

The adjacency list for G is:



If the depth first is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:

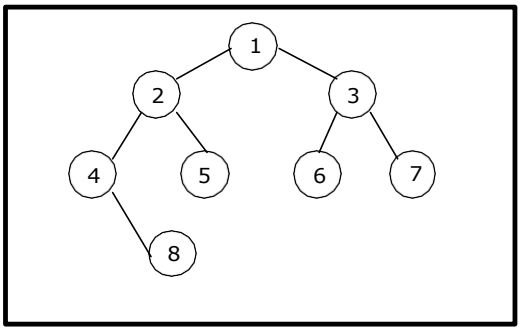


Depth First Spanning Tree

The spanning trees obtained using depth first searches are called depth first spanning trees. The edges rejected in the context of depth first search are called a back edges. Depth first spanning tree has no cross edges.

Breadth first search and traversal:

Starting at vertex 'V' and marking it as visited, BFS differs from DFS in that all unvisited vertices adjacent to V are visited next. Then unvisited vertices adjacent to there vertices are visited and so on. A breadth first search beginning at vertex 1 of the graph would first visit 1 and then 2 and 3.



Breadth First Spanning Tree

Next vertices 4, 5, 6 and 7 will be visited and finally 8. The spanning trees obtained using BFS are called Breadth first spanning trees. The edges that were rejected in the breadth first search are called cross edges.

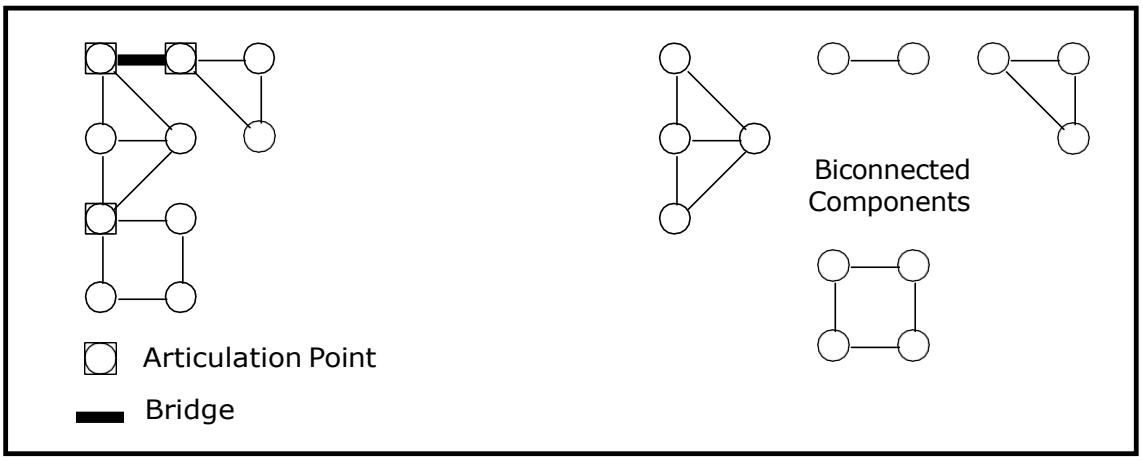
Articulation Points and Biconnected Components:

Let G = (V, E) be a connected undirected graph. Consider the following definitions:

Articulation Point (or Cut Vertex): An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces..

Bridge: Is an edge whose removal results in a disconnected graph.

Biconnected: A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components. This is illustrated in the following figure:



Articulation Points and Bridges

Biconnected graphs and articulation points are of great interest in the design of network algorithms, because these are the "critical" points, whose failure will result in the network becoming disconnected.

Let us consider the typical case of vertex v, where v is not a leaf and v is not the root. Let w_1, w_2, ... w_k be the children of v. For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is no back edge going to a proper ancestor of v, then if we remove v, this subtree becomes disconnected from the rest of the graph, and hence v is an articulation point.

On the other hand, if every one of the subtree rooted at the children of v have back edges to proper ancestors of v, then if v is removed, the graph remains connected (the back edges hold everything together). This leads to the following:

Observation 1: An internal vertex v of the DFS tree (other than the root) is an articulation point if and only if there is a subtree rooted at a child of v such that there is no back edge from any vertex in this subtree to a proper ancestor of v.

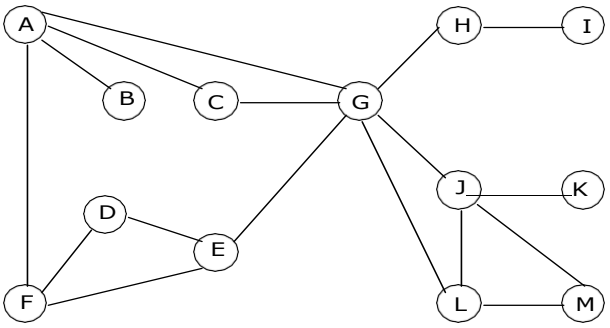
Observation 2: A leaf of the DFS tree is never an articulation point, since a leaf will not have any subtrees in the DFS tree.

Thus, after deletion of a leaf from a tree, the rest of the tree remains connected, thus even ignoring the back edges, the graph is connected after the deletion of a leaf from the DFS tree.

Observation 3: The root of the DFS is an articulation point if and only if it has two or more children. If the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree, and hence cannot disconnect the graph in general.

Articulation Points by Depth First Search:

Determining the articulation turns out to be a simple extension of depth first search. Consider a depth first spanning tree for this graph.

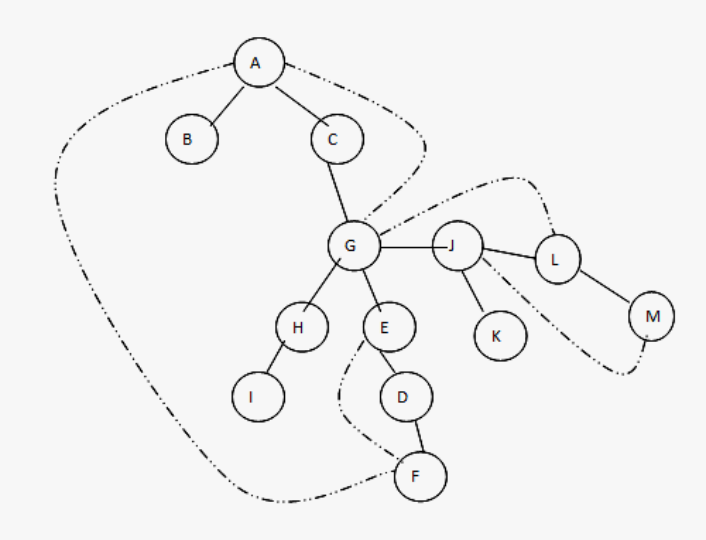


Observations 1, 2, and 3 provide us with a structural characterization of which vertices in the DFS tree are articulation points.

Deleting node E does not disconnect the graph because G and D both have dotted links (back edges) that point above E, giving alternate paths from them to F. On the other hand, deleting G does disconnect the graph because there are no such alternate paths from L or H to E (G's parent).

A vertex 'x' is not an articulation point if every child 'y' has some node lower in the tree connect (via a dotted link) to a node higher in the tree than 'x', thus providing an alternate connection from 'x' to 'y'. This rule will not work for the root node since there are no *nodes higher in the tree*. The root is an articulation point if it has two or more children.

Depth First Spanning Tree for the above graph is:



By using the above observations the articulation points of this graph are:

A : because it connects B to the rest of the graph.
H : because it connects I to the rest of the graph.
J : because it connects K to the rest of the graph.
G : because the graph would fall into three pieces if G is deleted.

Biconnected components are: {A, C, G, D, E, F}, {G, J, L, M}, B, H, I and K

This observation leads to a simple rule to identify articulation points. For each is define L (u) as follows:

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | (u, w) is a back edge}}.

L (u) is the lowest depth first number that can be reached from 'u' using a path of descendents followed by at most one back edge. It follows that, If 'u' is not the root then 'u' is an articulation point iff 'u' has a child 'w' such that:

L (w) ≥ DFN (u)

6.6.2. Algorithm for finding the Articulation points:

Pseudocode to compute DFN and L.

Algorithm Art (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initialized to zero and that // the
global variable num is initialized to 1. n is the number of vertices in G.
{
        dfn [u] := num; L [u] := num; num := num + 1;
        for each vertex w adjacent from u do
        {
                if (dfn [w] = 0) then
                {
                        Art  (w,  u);                   // w is unvisited.
                        L [u] := min (L [u], L [w]);
                }
        else if (w ≠ v) then L [u] := min (L [u], dfn [w]);
        }
}
```

6.6.1. Algorithm for finding the Biconnected Components:

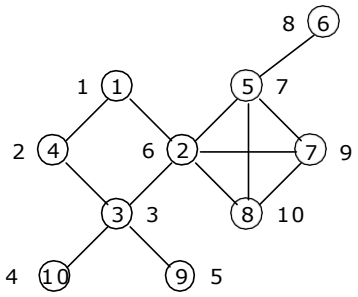
Algorithm BiComp (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initially zero and that the
// global variable num is initialized to 1. n is the number of vertices in G.
{
        dfn [u] := num; L [u] := num; num := num + 1;
        for each vertex w adjacent from u do
        {
                if ((v ≠ w) and (dfn [w] ≤ dfn [u])) then
                        add (u, w) to the top of a stack s;
                if (dfn [w] = 0) then
                {
                        if (L [w] ≥ dfn [u]) then
                        {
                                write ("New bicomponent");
                                repeat
                                {
                                        Delete an edge from the top of stack s;
                                        Let this edge be (x, y);
                                        Write (x, y);
                                } until (((x, y) = (u, w)) or ((x, y) = (w, u)));
                        }
```

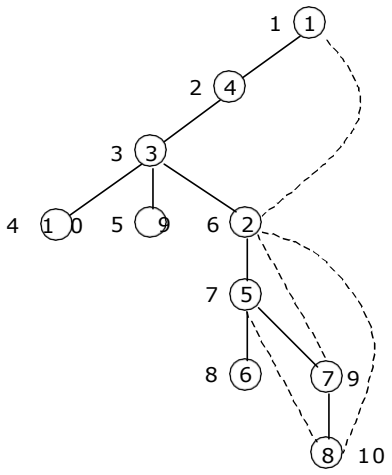
```
                    BiComp  (w,  u);                    // w is unvisited.
                    L [u] := min (L [u], L [w]);
                }
                else if (w ≠ v) then L [u] : = min (L [u], dfn [w]);
        }
}
```

6.7.1. Example:

For the following graph identify the articulation points and Biconnected components:



Graph



Depth First Spanning Tree

To identify the articulation points, we use:

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (4)}} = min {1, L (4)} = min {1, 1} = 1

L (4) = min {DFN (4), min {L (3)}} = min {2, L (3)} = min {2, 1} = 1

L (3) = min {DFN (3), min {L (10), L (9), L (2)}} =
= min {3, min {L (10), L (9), L (2)}} = min {3, min {4, 5, 1}} = 1

L (10) = min {DFN (10)} = 4

L (9) = min {DFN (9)} = 5

L (2) = min {DFN (2), min {L (5)}, min {DFN (1)}}
= min {6, min {L (5)}, 1} = min {6, 6, 1} = 1

L (5) = min {DFN (5), min {L (6), L (7)}} = min {7, 8, 6} = 6

L (6) = min {DFN (6)} = 8
L (7) = min {DFN (7), min {L (8}, min {DFN (2)}}
= min {9, L (8) , 6} = min {9, 6, 6} = 6

L (8) = min {DFN (8), min {DFN (5), DFN (2)}}

= min {10, min (7, 6)} = min {10, 6} = 6

Therefore, L (1: 10) = (1, 1, 1, 1, 6, 8, 6, 6, 5, 4)

Finding the Articulation Points:

Vertex 1: Vertex 1 is not an articulation point. It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as child 5 has L (5) = 6 and DFN (2) = 6, So, the condition L (5) = DFN (2) is true.

Vertex 3: is an articulation point as child 10 has L (10) = 4 and DFN (3) = 3, So, the condition L (10) > DFN (3) is true.

Vertex 4: is not an articulation point as child 3 has L (3) = 1 and DFN (4) = 2, So, the condition L (3) $\geq$ DFN (4) is false.

Vertex 5: is an articulation point as child 6 has L (6) = 8, and DFN (5) = 7, So, the condition L (6) > DFN (5) is true.

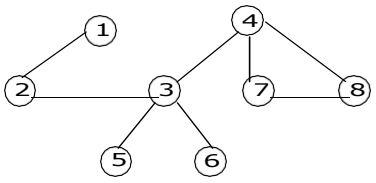
Vertex 7: is not an articulation point as child 8 has L (8) = 6, and DFN (7) = 9, So, the condition L (8) $\geq$ DFN (7) is false.

Vertex 6, Vertex 8, Vertex 9 and Vertex 10 are leaf nodes.

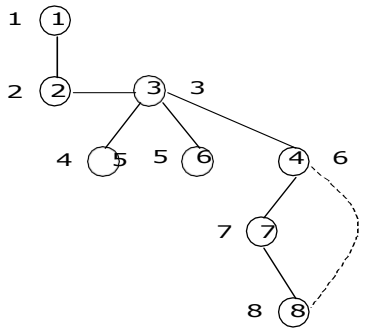
Therefore, the articulation points are {2, 3, 5}.

Example:

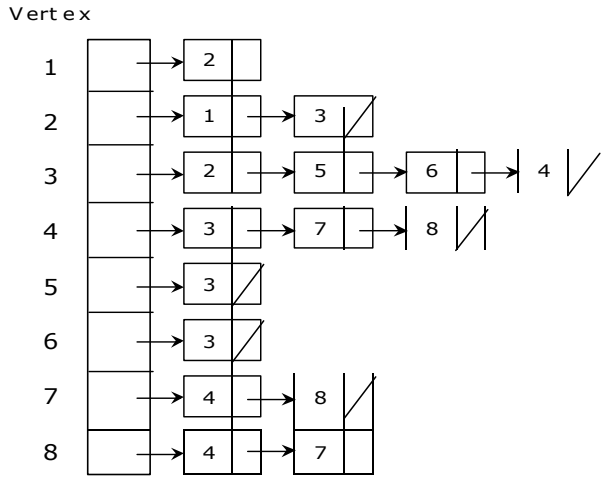
For the following graph identify the articulation points and Biconnected components:



Graph



DFS spanning Tree



Adjacency List

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (2)}} = min {1, L (2)} = min {1, 2} = 1

L (2) = min {DFN (2), min {L (3)}} = min {2, L (3)} = min {2, 3} = 2

L (3) = min {DFN (3), min {L (4), L (5), L (6)}} = min {3, min {6, 4, 5}} = 3

L (4) = min {DFN (4), min {L (7)} = min {6, L (7)} = min {6, 6} = 6

L (5) = min {DFN (5)} = 4

L (6) = min {DFN (6)} = 5

L (7) = min {DFN (7), min {L (8)}} = min {7, 6} = 6

L (8) = min {DFN (8), min {DFN (4)}} = min {8, 6} = 6

Therefore, L (1: 8) = {1, 2, 3, 6, 4, 5, 6, 6}

Finding the Articulation Points:

Check for the condition if L (w) $\geq$ DFN (u) is true, where w is any child of u.

Vertex 1: Vertex 1 is not an articulation point.
It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as L (3) = 3 and DFN (2) = 2.
So, the condition is true

Vertex 3: is an articulation Point as:

I. L (5) = 4 and DFN (3) = 3
II. L (6) = 5 and DFN (3) = 3 and
III. L (4) = 6 and DFN (3) = 3

So, the condition true in above cases

Vertex 4: is an articulation point as L (7) = 6 and DFN (4) = 6.
So, the condition is true

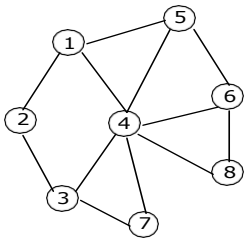
Vertex 7: is not an articulation point as L (8) = 6 and DFN (7) = 7.
So, the condition is False

Vertex 5, Vertex 6 and Vertex 8 are leaf nodes.

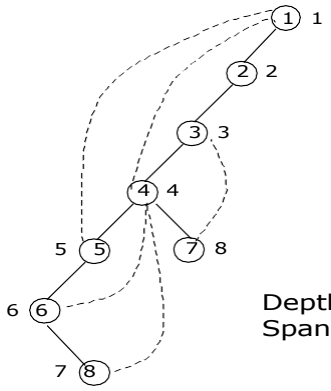
Therefore, the articulation points are {2, 3, 4}.

Example:

For the following graph identify the articulation points and Biconnected components:

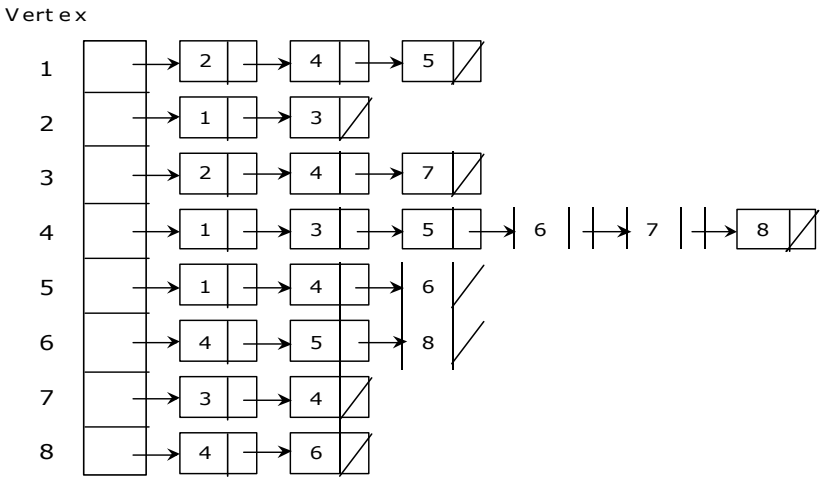


Graph



Depth First
Spanning Tree

DFN (1: 8) = {1, 2, 3, 4, 5, 6, 8, 7}



Adjacency List

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (2)}}
= min {1, L (2)} = 1

L (2) = min {DFN (2), min {L (3)}} = min {2, L (3)} = min{2, 1}= 11

L (3) = min {DFN (3), min {L (4)}} = min {3, L (4)} = min {3, L (4)}
= min {3, 1} = 1

L (4) = min {DFN (4), min {L (5), L (7)}, min {DFN (1)}}
= min {4, min {L (5), L (7)}, 1} = min {4, min {1, 3}, 1}
= min {4, 1, 1} = 1

L (5) = min {DFN (5), min {L (6)}, min {DFN (1)}} = min {5, L (6), 1}
= min {5, 4, 1} = 1

L (6) = min {DFN (6), min {L (8)}, min {DFN (4)}} = min(6, L (8), 4}
= min(6, 4, 4} = 4

L (7) = min {DFN (7), min {DFN (3)}} = min {8, 3} = 3

L (8) = min {DFN (8), min {DFN (4)}} = min {7, 4} = 4

Therefore, L (1: 8) = {1, 1, 1, 1, 1, 4, 3, 4}

Finding the Articulation Points:

Check for the condition if L (w) $\geq$ DFN (u) is true, where w is any child of u.

Vertex 1: is not an articulation point.
It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is not an articulation point. As L (3) = 1 and DFN (2) = 2.
So, the condition is False.

Vertex 3: is not an articulation Point as L (4) = 1 and DFN (3) = 3.
So, the condition is False.

Vertex 4: is not an articulation Point as:
L (3) = 1 and DFN (2) = 2 and
L (7) = 3 and DFN (4) = 4
So, the condition fails in both cases.

Vertex 5: is not an Articulation Point as L (6) = 4 and DFN (5) = 6.
So, the condition is False

Vertex 6: is not an Articulation Point as L (8) = 4 and DFN (6) = 7.
So, the condition is False

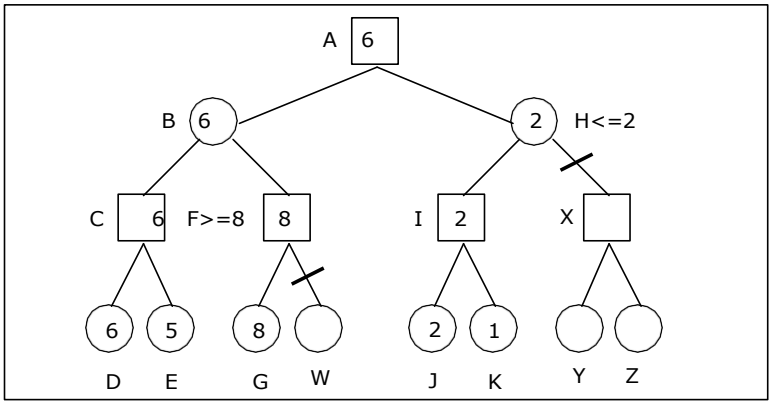
Vertex 7: is a leaf node.

Vertex 8: is a leaf node.

So they are no articulation points.

GAME PLAYING

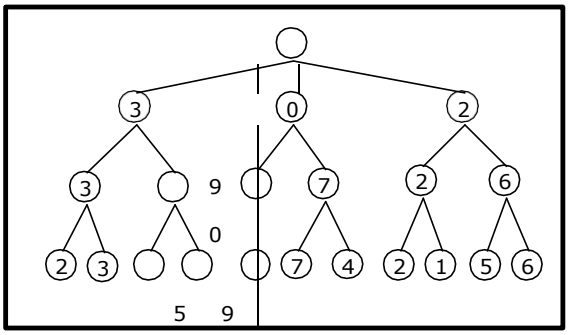
In game-playing literature, the term play is used for a move. The two major components of game-playing program are plausible move generator, and a static evaluation function generator.



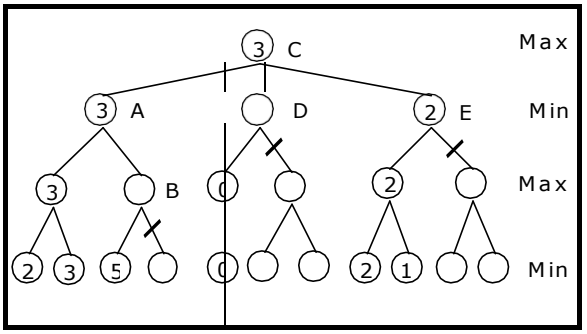
Alpha - Beta Pruning

6.8.1. EXAMPLE 2:

Taking the space of figure as shown below and when alpha-beta pruning applied is on this problem, is as follows:



Hypothetical state space to minimax



Alpha - Beta Pruning

A has β = 3 (A will be no larger than 3).
B is β pruned, since 5 > 3.
C has α = 3 (C will be no smaller than 3).
D is α pruned, since 0 < 3.
E is α pruned, since 2 < 3.
C is 3.

AND/OR GRAPH:

And/or graph is a specialization of hypergraph which connects nodes by sets of arcs rather than by a single arcs. A hypergraph is defined as follows:

A hypergraph consists of:

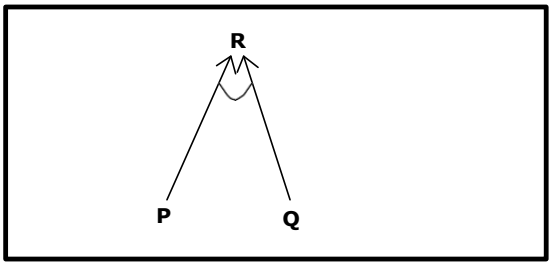
N, a set of nodes,

H, a set of hyperarcs defined by ordered pairs, in which the first implement of the pair is a node of N and the second implement is the subset of N.

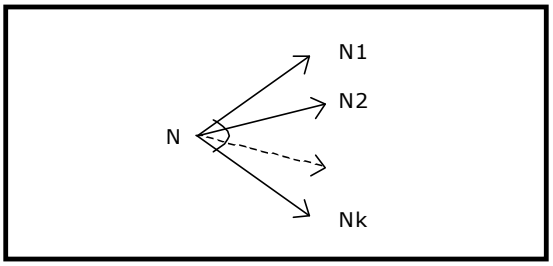
An ordinary graph is a special case of hypergraph in which all the sets of decendent nodes have a cardinality of 1.

Hyperarcs also known as K-connectors, where K is the cardinality of the set of decendent nodes. If K = 1, the descendent may be thought of as an OR nodes. If K > 1, the elements of the set of decendents may be thought of as AND nodes. In this case the connector is drawn with individual edges from the parent node to each of the decendent nodes; these individual edges are then joined with a curved link.

And/or graph for the expression P and Q -> R is follows:



Expression for P and Q -> R



A K-Connector

The K-connector is represented as a fan of arrows with a single tie is shown above.

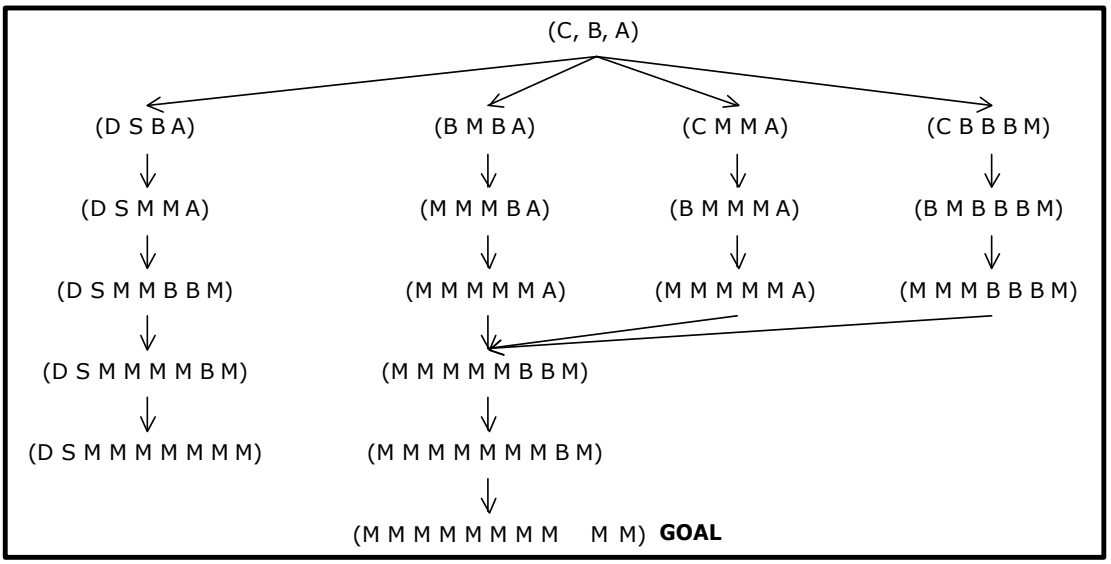
The and/or graphs consists of nodes labelled by global databases. Nodes labelled by compound databases have sets of successor nodes. These successor nodes are called AND nodes, in order to process the compound database to termination, all the compound databases must be processed to termination.

For example consider, consider a boy who collects stamps (M). He has for the purpose of exchange a winning conker (C), a bat (B) and a small toy animal (A). In his class there are friends who are also keen collectors of different items and will make the following exchanges.

1. 1 winning conker (C) for a comic (D) and a bag of sweets (S).
2. 1 winning conker (C) for a bat (B) and a stamp (M).
3. 1 bat (B) for two stamps (M, M).
4. 1 small toy animal (A) for two bats (B, B) and a stamp (M).

The problem is how to carry out the exchanges so that all his exchangable items are converted into stamps (M). This task can be expressed more briefly as:

1. Initial state = (C, B, A)
2. Transformation rules:
   a. If C then (D, S)
   b. If C then (B, M)
   c. If B then (M, M)
   d. If A then (B, B, M)
3. The goal state is to left with only stamps (M, ... , M)



Expansion for the exchange problem using OR connectors only

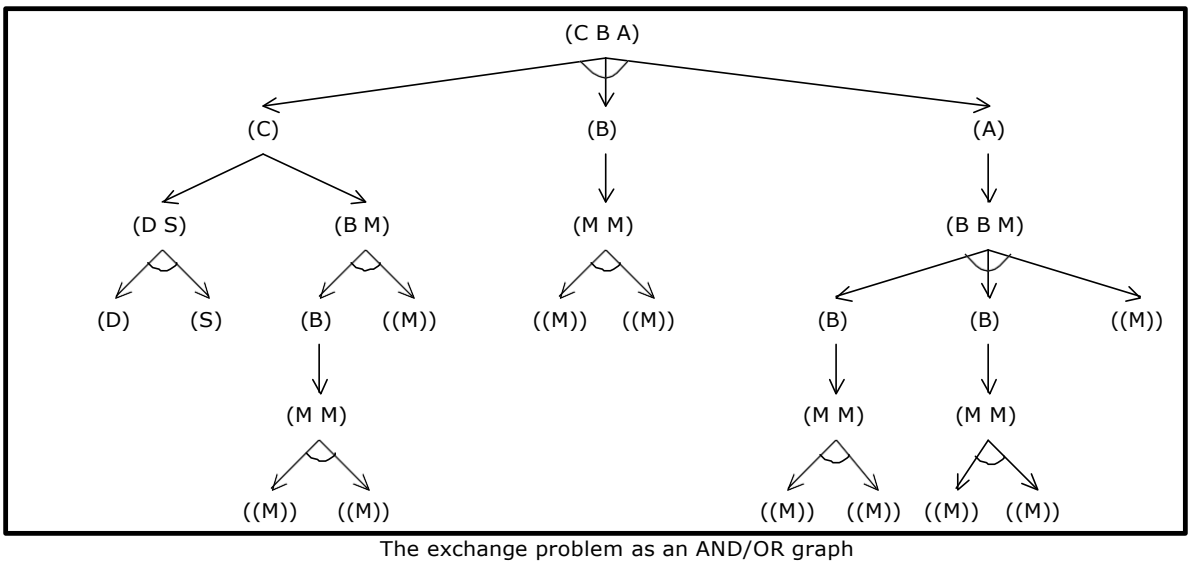
The figure shows that, a lot of extra work is done by redoing many of the transformations. This repetition can be avoided by decomposing the problem into subproblems. There are two major ways to order the components:

1. The components can either be arranged in some fixed order at the time they are generated (or).
2. They can be dynamically reordered during processing.

The more flexible system is to reorder dynamically as the processing unfolds. It can be represented by and/or graph. The solution to the exchange problem will be:

> Swap conker for a bat and a stamp, then exchange this bat for two stamps. Swap his own bat for two more stamps, and finally swap the small toy animal for two bats and a stamp. The two bats can be exchanged for two stamps.

The previous exchange problem, when implemented as an and/or graph looks as follows:

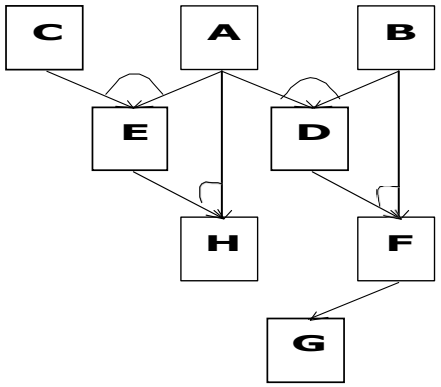


The exchange problem as an AND/OR graph

Example 1:

Draw an And/Or graph for the following prepositions:

1. A
2. B
3. C
4. A ^ B -> D
5. A ^ C -> E
6. B ^ D -> F
7. F -> G
8. A ^ E -> H



Connected components:

Graph:- A graph is collection of sets V and E. where V = finite non-empty set of vertices, E = finite non-empty set of edges.

Denoted by $G = \{V, E\}$.

Vertices : nodes in the graph

Edge : Two adjacent vertices are joined with vertices

Graph traversals:- (i) BFS (ii) DFS.

DFS	BFS
1. Stack data structure used for traversed	1. Queue data structure used for traversed.
2. Efficiency of adjacency matrix $O(V^2)$	2. Efficiency $O(V^2)$
3. Efficiency adjacency list graph $O(\|V\| + \|E\|)$	3. $O(\|V\| + \|E\|)$
4. Applications: Spanning tree connected, Bi connected components articulation point	4. Shortest path, Spanning tree, produce a reverse topological ordering of nodes.

Connected graph:- A graph is said to be connected if there exists a path between any two vertices.

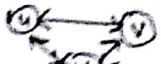
-if graph G is connected, undirected graph, visit all vertices of graph in BFS, DFS.

Connected components:- - It is a collection of vertices such that, there is a path b/w ~~any~~ every pair of vertices in the same component.

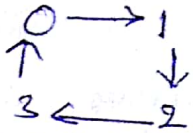
- -obtain path b/w any two vertices, the transitive closure is obtained.

Strongly connected components: A directed graph is strongly connected if there is a path between all pairs of vertices.

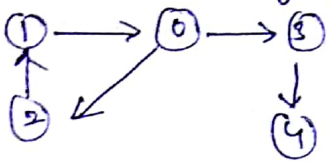
- In directed graphs vertex 'a' is strongly connected to b, if there exists two paths a to b and another from b to a.
- Properties: if u, v are strongly connected vertices satisfies
  1. Reflexive Property: anything equal to itself u=u
  2. Symmetric property path a→b, b→a
  3. Transitive property



1. A directed graph is strongly connected if there is a path b/w all pair of vertices Eg:



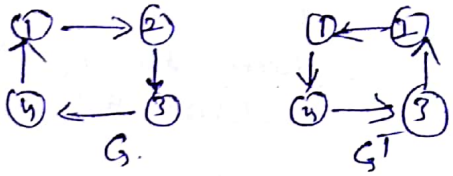
2. A strongly connected component (SCC) of directed graph is a maximal strongly connected subgraph



1-0-2 — component strongly
3 — SCC
4 — SCC

Kosaraju's Idea:

Step1: call DFS(G) to compute finish time for each vertex
Step2: compute G^T
Step3 call DFS(G^T) decreasing order of their finish times
Step4: output

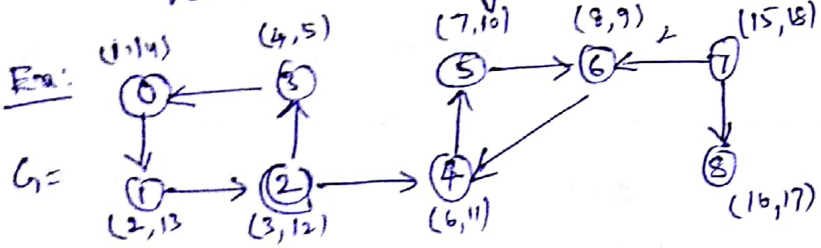


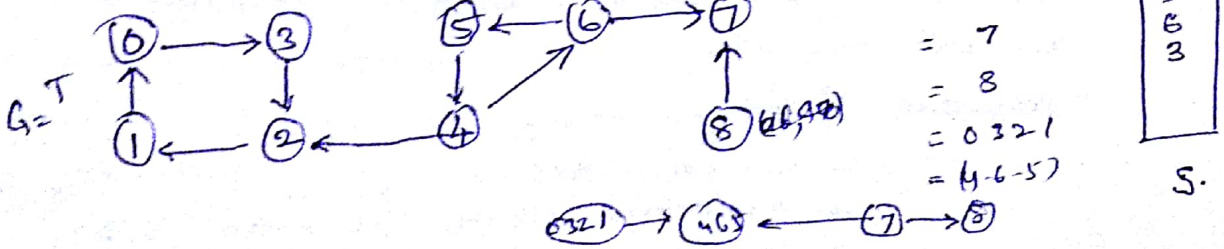
Eg:

Kosaraju's Alg:

1. Create an empty stack
2. Do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex push vertex into stack
3. Reverse directions of all arcs to transpose graph
4. One by one pop a vertex from s while s is not empty. Let the popped vertex be 'v'. Take v as source and do DFS call on v. the DFS starting from v print strongly connected components of v.

Eg:



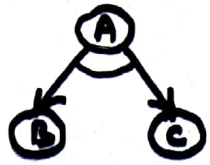


AND/OR GRAPHS: Divide complex problem into several subproblems can be represented as AND/OR Graphs.

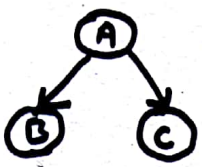
Ex: 1. A is solvable iff both problems B and C are solvable

2. A is solfiable iff atleast one problem of B, C are solvable

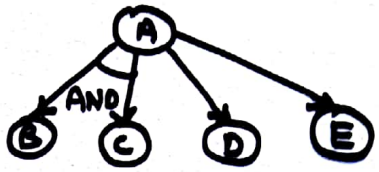
3. Problem A can be solvable either by solving both sub problems B and C or the single problems D or E



1. AND



2. OR



3.

Procedure:- - A structure directed graph in which nodes represent problems and descending of a node represent sub problems

- AND graph nodes will be drawn with an arc across all edges leaving the node.
- Nodes with no decendents are termed as terminal. Terminal node represent primitive problems and are marked either solvable or not solvable. solvable represented by '□' otherwise, ○(circle).

Algorithm to determine if the AND/OR tree T is solvable:-

```
Procedure SOLVE(T)
// T is an AND/OR tree with root T, T≠0
  Alg. returns '1' if problem is solvable otherwise '0' //
case:
  :T is a terminal node: [if T is solvable then return(1) else return(0)
                         endif
  :T is an AND node    : [for each child S of T do
                             if SOLVE(S)=0 then return(0) endif
                         repeat
                         return(1)
  :else:                 [for each child S of T do    /OR node//
                             If SOLVE(S)=1 then return (1) endif
                         repeat
end case
end SOLVE.
```

Ex: