Breadth first Search (BFS)

Given a Graph G=(V,E) with an arbitrary source vertex ‘s’. This Graph traversal scheme starts exploring from any source vertex. Then it explores all of its adjacent vertices. Then it takes one explored vertex in FIFO order and explores all adjacent of this vertex, which are not explored yet. This process is continued till all the vertices are explored.

This algorithm works on both directed and undirected graph. In BFS, the vertices with distance k are explored before the vertices at distance k+1.

Following data structures are used to represent status, parent and distance.

Color [u] : It represents status of a vertex ‘u’.

color[u] = WHITE , means u is unreached

color[u] = GRAY , means u is explored, but all its adjacent are not explored.

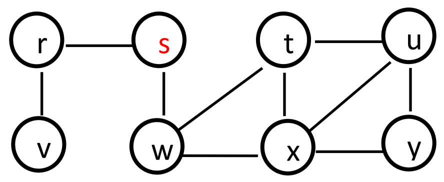
color[u] = BLACK , means u and all the adjacent of u are explored.

p[u]	: It represents the parent vertex of ‘u’.
d[u]	: it represents the distance (number of edges) from source ’s’ to vertex ‘u’

In addition to these data structure, a **Queue data structure** is used to hold the GRAY vertices.

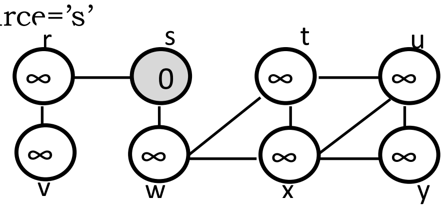
```
Algorithm: BFS(G,s)
//The algorithm makes BFS traversal on a graph G from a start vertex 'S'.
1. FOR each vertex v ∈V[G]-s
2.      color[v] = WHITE
3.      p[u] = nil
4.      d[u] = ∞
5. color[s] = GRAY
8. d[s] = 0
9. ENQUEUE (Q, s)    //insert source vertices in set V in Queue Q
10. WHILE Q ≠ Ø
11.     u= DEQUEUE(Q)   //Delete a vertex from Queue in FIFO order
12.     FOR each vertex v ∈ Adj[u]
13.             IF color [v] = WHITE
14.                 color[v] = GRAY
15.                 p[v] = u
16.                 d[v] = d[u] + 1
17.                 ENQUEUE (Q,v)
18.     color[u] = BLACK
```

Problem: Apply BFS on following graph with start vertex S



Solution:

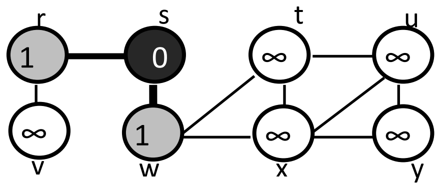
Source='s'



Vertex	r	s	t	u	v	w	x	y
Color	W	**G**	W	W	W	W	W	W
D	**-**	**0**	**-**	**-**	-	-	-	-
P	**-**	**-**	**-**	**-**	-	-	-	-

Q : s

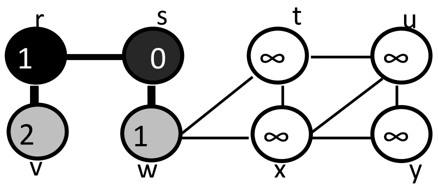
Step-1: Delete s , process its adjacent r,w.



Vertex	r	s	t	u	v	w	x	y
Color	**G**	**B**	W	W	W	**G**	W	W
D	**1**	0	**-**	**-**	-	**1**	-	-
P	**s**	-	**-**	**-**	-	**s**	-	-

Q : r,w

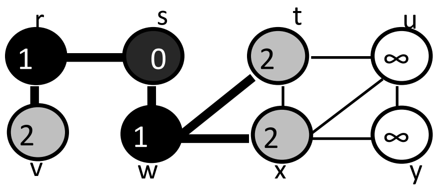
Step-2:Delete r ,Process its adjacent v



Vertex	**r**	s	t	u	v	w	x	y
Color	**B**	B	W	W	**G**	G	W	W
D	1	0	**-**	**-**	**2**	1	-	-
P	s	**-**	**-**	**-**	**r**	s	-	-

Q : w,v

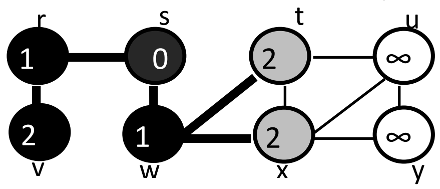
Step-3:Delete w ,Process its adjacent t,x



Vertex	**r**	s	t	u	v	w	x	y
Color	B	B	**G**	W	G	**B**	**G**	W
D	1	0	**2**	**-**	2	1	**2**	-
P	s	**-**	**W**	**-**	r	s	**W**	-

Q : v,t,x

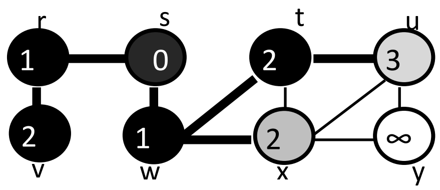
Step-4:Delete v ,v has no white adjacent



Vertex	**r**	s	t	u	v	w	x	y
Color	B	B	G	W	**B**	B	G	W
D	1	0	2	**-**	2	1	2	-
P	s	**-**	W	**-**	r	s	w	-

Q : t,x

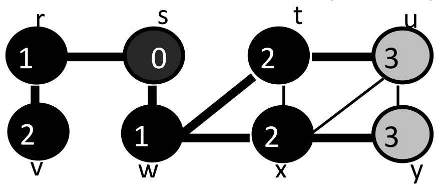
Step-5:Delete t , Process its adjacent u



Vertex	**r**	s	t	u	v	w	x	y
Color	B	B	**B**	**G**	B	B	G	W
D	1	0	2	**3**	2	1	2	-
P	s	**-**	W	**t**	r	s	w	-

Q : x,u

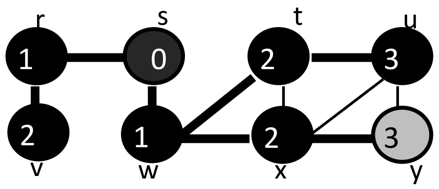
Step-6: Delete x , Process its adjacent y



Vertex	**r**	s	t	u	v	w	x	y
Color	B	B	B	G	B	B	**B**	G
D	1	0	2	3	2	1	2	3
P	s	-	W	t	r	s	w	x

Q : u,y

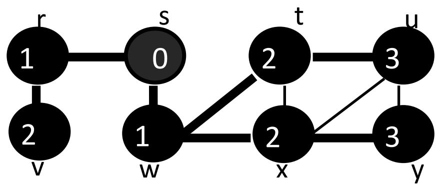
Step-7: Delete u , u has no white adjacent



Vertex	**r**	s	t	u	v	w	x	y
Color	B	B	B	**B**	B	B	**B**	G
D	1	0	2	3	2	1	2	3
P	s	-	W	t	r	s	w	x

Q : y

Step-7: Delete y , y has no white adjacent



Vertex	**r**	s	t	u	v	w	x	y
Color	B	B	B	B	B	B	B	**B**
D	1	0	2	3	2	1	2	3
P	s	-	W	t	r	s	w	x

Q : Nil

The sequence of BFS is the order, the elements are en-queued. ,i.e, **<s,r,w,v,t,x,u,y>**

Analysis:

Let V is the number of vertices and E is the number of edges,

Step 1 to 4 takes O(V) times to initialize status of vertices. Each enqueue and dequeue operation takes O(1) time. As a vertex is enqueued or dequeued once only, so for a total of V Enqueue() operation, takes O(V) times.

The inner loop is executed according to the degree of each vertex, i,e the number of adjacency vertices. In a undirected graph, there are 2*E number of edges, so the inner loop is altogether executed O(E) times.

So the total running time of the algorithm is O(V) + O(E) = **O(V+E).**

Depth First Search (DFS)

In depth first search, edges are explored out of the most recently discovered vertex ‘V” that still has unexplored edges leaving it. When all of the edges of ‘V’ have been explored, the search backtracks to explore edges’ leaving the vertex from which V was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex.

If any undiscovered vertex remains, then one of them is selected as new source vertex and search is repeated until all the vertices are discovered

DFS uses following data structure.

color[u]	: Store the status of vertex ‘u’
p[u]	: It represents the parent vertex of ‘u’.
d[u]	: it represents the time stamp for discovering time
f[u]	: it represents the time stamp for finishing time
stack	: stores the GRAY vertices. **Recursion can be used in place of** stack.

Algorithm: DFS (G, S)
//The algorithm makes DFS traversal on a graph G from s.

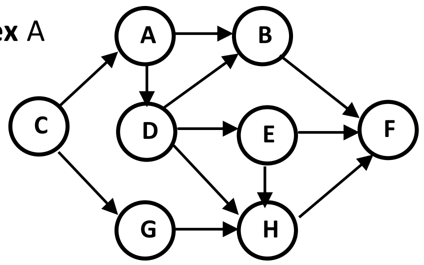
```
1. FOR each vertex v ∈V[G]
2.      color[v] = WHITE
3.      p[u] = nil
4. time = 0
5. FOR each vertex v ∈ V[G]
6.      IF color [v] = WHITE
7.         DFS_Visit(v)

DFS_Visit(u)
1. color[v] = GRAY
2. time = time + 1
3. d[u] = time
4. FOR each vertex v ∈ Adj[u]
5.      IF color [v] = WHITE
6.         p[u] = u
7.         DFS_Visit(v)
8. color[u] = BLACK
9. f[u] = time = time + 1
```

Analysis:

Loop in line-1 to 4 and loop in line 5 to 7 in DFS () take time O(V) exclusive the time to execute DFS_Visit().The DFS_Visit() procedure is called exactly once for each vertex v ϵ V. During the execution of DFS_Visit(), the loop in line 4 to 7 is executed $|Adj[u]|$ times. Since $\sum_{u \in V} |Adj[u]| = O(E)$, the total cost of executing the loop in line 4 to 7 is O(E). So the total running time of the algorithm is O(V) + O(E) = **O(V+E).**

Problem: Apply DFS on following graph with start vertex A

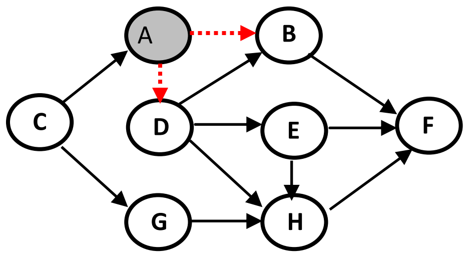


Initialization

time=0

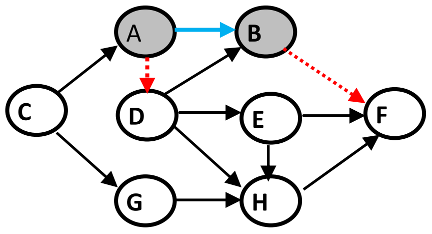
Step-1 Visit A, White adj[A]={B,D},process B

	A	B	C	D	E	F	G	H
p	-	-	-	-	-	-	-	-
d	1							
F								



step-2: Visit B , White adj[B]={F},process F

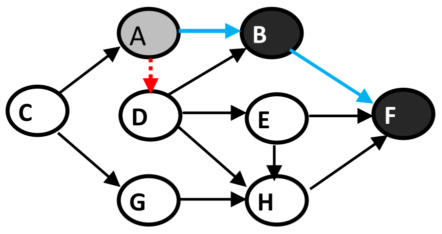
	A	B	C	D	E	F	G	H
p	-	A	-	-	-	-	-	-
d	1	2						
F								



step-3: Visit F ,No Adjacent of F. Backtrack to parent B. B has no adjacent to process, Back to A

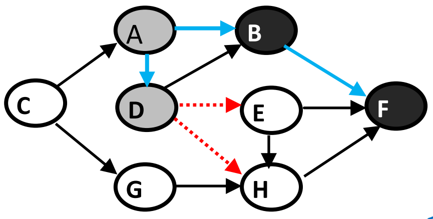
Process D

	A	B	C	D	E	F	G	H
p	-	A	-	-	-	B	-	-
d	1	2				3		
F		5				4		



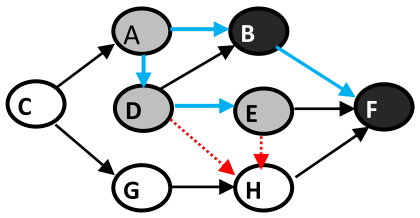
step-4: Visit D , White adj[D]={E,H},process E

	A	B	C	D	E	F	G	H
p	-	A	-	A	-	B	-	-
d	1	2		6		3		
F		5				4		



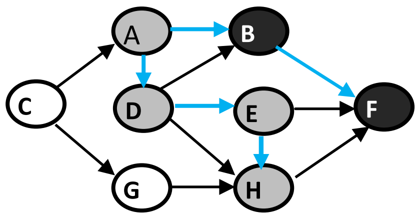
step-5: Visit E , White adj[E]={H},process: H

	A	B	C	D	E	F	G	H
p	-	A	-	A	D	B	-	-
d	1	2		6	7	3		
F		5				4		

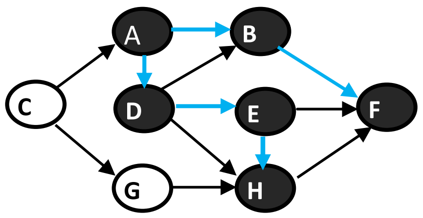


step-6: Visit H , No Adjacent of H.

	A	B	C	D	E	F	G	H
p	-	A	-	A	D	B	-	E
d	1	2		6	7	3		8
F		5				4		9



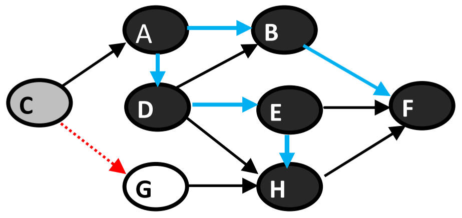
Finish H. Back to E. No adjacent of E remains. Finish E. back to D. No remaining adjacent of D to process. Back to A. Finish A



	A	B	C	D	E	F	G	H
p	-	A	-	A	D	B	-	E
d	1	2		6	7	3		8
F	12	5		11	10	4		9

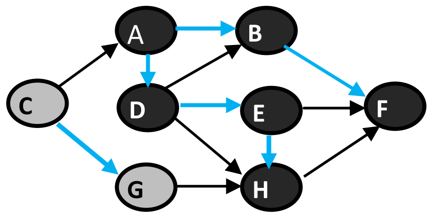
step-7: Visit C , White adj[C]={G},process G.

	A	B	C	D	E	F	G	H
p	-	A	-	A	D	B	-	E
d	1	2	13	6	7	3		8
F	12	5		11	10	4		9



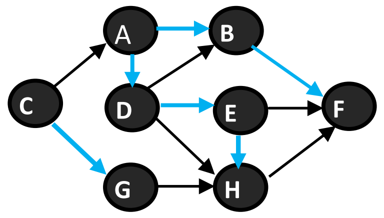
step-8: Visit G. G has no white adjacent.

	A	B	C	D	E	F	G	H
p	-	A	-	A	D	B	-	E
d	1	2	13	6	7	3	14	8
F	12	5		11	10	4		9



Finish G. Back to C. C has no white adjacent. Finish C

	A	B	C	D	E	F	G	H
p	-	A	-	A	D	B	-	E
d	1	2	13	6	7	3	14	8
F	12	5	16	11	10	4	15	9



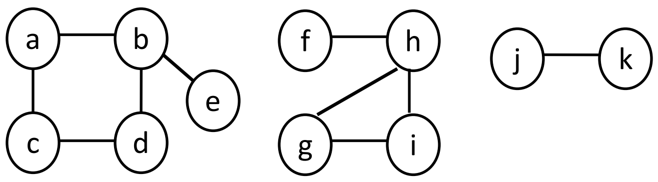
No white vertex remains. **The DFS traversal is : A,B,F,D,E,H,C,G**

It is a collection of disjoint sets, each set is represented by a member of the set, called representative. Member of one set should not be a member of any other sets.

Operations:

Make_Set(x)	: Creates a set containing one member x. x is the representative of the set.
Union(x,y)	: Unites two sets containing x and y into a new set .Representative is any member from new set.
Find_Set(x)	: returns the representative of the set containing x.

Application of disjoint sets: Connected component



S1 : {a,b,c,d,e}

S2: {f,g,h,i}

S3 : { j,k}

Initially all the nodes are dis-joints.

Edge processed	Collection of dis-joint sets										
	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}	{k}
(a,b)	{a,b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}	{k}
(a,c)	{a,b,c}			{d}	{e}	{f}	{g}	{h}	{i}	{j}	{k}
(b,d)	{a,b,c,d}				{e}	{f}	{g}	{h}	{i}	{j}	{k}
(f,h)	{a,b,c,d}				{e}	{f,h}	{g}		{i}	{j}	{k}
(g,i)	{a,b,c,d}				{e}	{f,h}		{g,i}		{j}	{k}
(b,e)	{a,b,c,d,e}					{f,h}		{g,i}		{j}	{k}
(c,d)											
(g,h)	{a,b,c,d,e}					{f,g,h,i}				{j}	{k}
(h,i)											
(j,k)	{a,b,c,d,e}					{f,g,h,i}				{j,k}	

[3-connected component]

Algorithm:

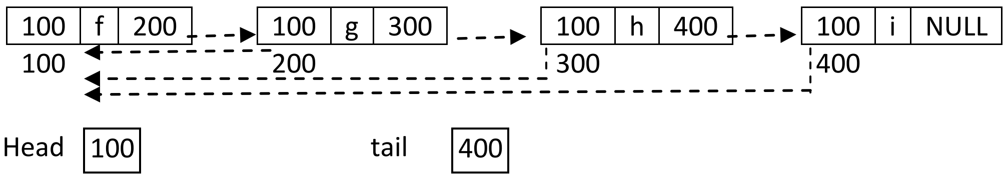
```
Connected_Components(G)
// This algorithm placed each vertices in its own sets.
1. for each vertex v ϵ V[G]
2.       Make_Set(v)
3. for each edge(u,v) ϵ E[G]
4.       If Find_Set(u) ≠ Find_Set(v)
5.             Union(u,v)
```

- Number of Make_Set() operation = n
- Number of Union operation = n-k, k is the number of component.
- Number of find_Set() operation = 2E

Linked List representation:

One linked list is used for each set. First node of every list contains representative. Each node contains (1) Member, (2)Pointer to the node containing next member and (3) pointer to the representative.

Each list maintains two pointers: **head,** points to representative and **tail,** points to the last node. Consider set S2:



Makeset(x) : Creates a node and set member as x and the pointer to representative is self referenced. It takes O (1) time.

Findset(x) : Return pointer to representative . Find_Set() takes O(1) time, as each node contains the address of representative.

Union of Two sets: Let x and y are to lists, union(x,y) appends x's list at the end of the y's list. Follow the steps as below.

- Representative of y’s list become representative of new list.
- pointer of last node of Y now point to the first node of X.
- the pointer to the representative of new list will replace pointer to the representative of each node in x.

Analysis:

Suppose number of nodes in X list is n, finding tail pointer of Y takes O(1) time and Updating each node in X takes O(n) time.

Consider following operations for n disjoint objects x1,x2, .. . xn(worst case scenario)

Operation	**Number of update**
Make_set(x1)	1
Make_set(x2)	1
:	:
Make_set(xn)	1
Union(x1,x2)	1
Union(x2,x3)	2
:	:
Union(xn-1,xn)	n-1
Total	n(n+1)/2

Number of makeset() operation is n . takes O(n) time.

Number of union() operation is n-1 . takes $O(n^2)$ time.

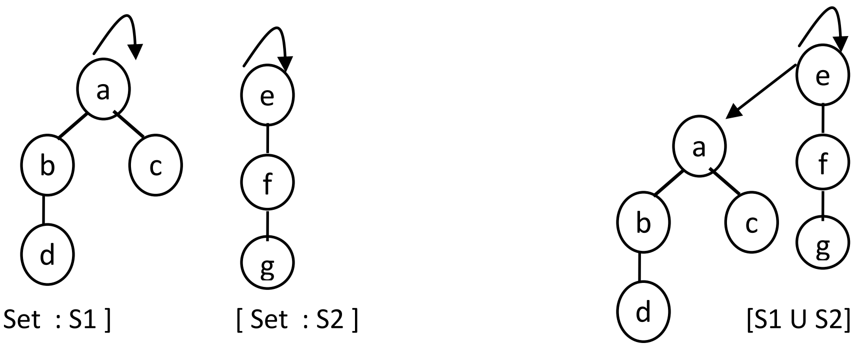
Total of 2n-1 operations takes $O(n) + O(n^2) = O(n^2)$ time.

So average time for each operation = O(n)

Disjoint Set Forest:

This is a faster representation of disjoint sets. Here each set is represented by a tree, with each node containing one member and each member points to its parent. The root of the tree contains the representative and point to itself.

Example: Two sets {a,b,c,d } and {e,f,g} and their union



[Set : S1] [Set : S2] [S1 U S2]

Make_Set(x) : It creates a tree with one node x. requires O(1) time.

Union(x,y) : It causes root of one tree points to root of another tree. Requires O(1) time.

Find_Set(x) : Follows the parent pointer to the root. Requires O(h) . [h-height of tree]

Techniques used to improve running time

- Union by Rank
- Path compression

Union by rank: In this approach, make the root of the tree with fewer nodes point to the root of the tree with more nodes.

For each node, a data structure rank is used that is upper bound on the height of the node. In union operation, the root with smaller rank is to point to the root with higher rank.

Path compression: Generally applied to Find_Set() operation to effectively reduce the running time. This approach makes each node in the find path point directly to the root.

Algorithms: Disjoint operations using disjoint ser forest

```
Make_Set(x)
1. p[x] = x
2. rank[x] = 0

Find_Set(x)
1. if x ≠ p[x]
2.      p[x] = Find_Set(p[x])
3. return p[x]


Union(x,y)
1. x = Find_Set(x)
2. y = Find_Set(y)
3. if rank[x] > rank[y]
4.        p[y] = x
5. else,  p[x] = y
6.        if rank[x] = rank[y]
7.             rank[y] = rank[y] + 1
```