Algorithms
Chapter 2: Brute Force

GATE CS Lectures
by Monalisa

- Section 5: Algorithms
- Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques : greedy, dynamic programming and divide-and-conquer . Graph traversals, minimum spanning trees, shortest paths
- Chapter 1:Algorithim Analysis:-Algorithm intro , Order of growth ,Asymptotic notation, Time complexity, space complexity, Analysis of Recursive & non recursive program, Master theorem]
- Chapter 2:Brute Force:-Sequential search, Selection Sort and Bubble Sort , Radix sort, Depth first Search and Breadth First Search.
- Chapter 3: Decrease and Conquer :- Insertion Sort, Topological sort,Binary Search .
- Chapter 4: Divide and conquer:-Min max problem , matrix multiplication ,Merge sort ,Quick Sort , Binary Tree Traversals and Related Properties .
- Chapter 5: Transform and conquer:- Heaps and Heap sort, Balanced Search Trees.
- Chapter 6: Greedy Method:-knapsack problem , Job Assignment problem, Optimal merge, Hoffman Coding, minimum spanning trees, Dijkstra's Algorithm.
- Chapter 7: Dynamic Programming:-The Bellman-Ford algorithm ,Warshall's and Floyd's Algorithm ,Rod cutting, Matrix-chain multiplication ,Longest common subsequence ,Optimal binary search trees
- Chapter 8: Hashing.
- Reference : Introduction to Algorithms by Thomas H. Cormen
- Introduction to the Design and Analysis of Algorithms, by Anany Levitin
- My Note

- Chapter 2:Brute Force
- Sequential search
- Selection Sort and Bubble Sort , Radix sort
- Depth first Search and Breadth First Search.

Brute Force

- ***Brute force*** is a straightforward approach to solving a problem,usually directly based on the problem statement and definitions of the concepts involved.
- The "force" implied by the strategy's definition is that of a computer and not that of one's intellect.
- "Just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.
- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set

Sequential Search:

5	3	8	1	6
0	1	2	3	4

ALGORITHM *SequentialSearch* (*A[0…,n-1],K*)
//Input : An array *A[0..n-1]* and a search key *K*
//Output : The index of the first element in *A* that matches *K or -1* if no matching founds.
i = 0
while *i<n and A[i]≠ K* ***do***
    i = i+1
if *i<n* ***return*** *i*
else return *-1*

- Let K=8
- i=0 ,0<5 and A[0] ≠ K
- i=1, 1<5 and A[1] ≠ K
- i=2, 2<5 and A[2] = K
- 2<5 return 2

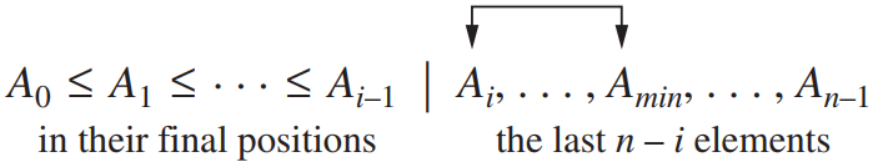
- Worst & Avg-case running time=O(n)
- Best case running time=O(1)
- The average number of key comparisons made by Sequential search is (n + 1)/2
- The ***sorting problem*** is to rearrange the items of a given list in nondecreasing order .

❖ **Selection Sort and Bubble Sort:**

- Application of the brute-force approach to the problem of sorting are Selection Sort and Bubble Sort : given a list of *n* orderable items rearrange them in nondecreasing order.

➢ **Selection Sort**

- We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.
- Then we scan the list, starting with the second element,to find the smallest among the last *n* - 1 elements and exchange it with the second element, putting the second smallest element in its final position.
- After *n* - 1 passes, the list is sorted



- **ALGORITHM** *SelectionSort(A[0..n - 1])*
- //Input: An array *A*[0..*n* - 1] of orderable elements
- //Output: Array *A*[0..*n* - 1] sorted in nondecreasing order
- **for** *i* ← 0 **to** *n* - 2 **do**
- *min* ← *i*
- **for** *j* ← *i* + 1 **to** *n* - 1 **do**
- **if** *A*[*j*] < *A*[*min*] *min* ← *j*
- swap *A*[*i*] and *A*[*min*]

8	**4**	**6**	**9**	**2**	**3**	**1**
1	**4**	**6**	**9**	**2**	**3**	**8**
1	**2**	**6**	**9**	**4**	**3**	**8**
1	**2**	**3**	**9**	**4**	**6**	**8**
1	**2**	**3**	**4**	**9**	**6**	**8**
1	**2**	**3**	**4**	**6**	**9**	**8**
1	**2**	**3**	**4**	**6**	**8**	**9**

- The input size is number of elements *n*;basic operation is the key comparison *A*[*j*] < *A*[*min*].
- The number of times it is executed depends only on the array size and is given by the sum:
- Time Complexity $C(n) = \sum_{i=0}^{n-2} * \sum_{j=i+1}^{n-1} 1$
- $= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$
- $= \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2} \in \Theta(n^2)$
- Thus, selection sort is a $\Theta(n^2)$ algorithm on best case & worst case inputs.
- The number of key swaps is only $\Theta(n)$, or, more precisely, $n - 1$.
- This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

- Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order.
- By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list.
- The next pass bubbles up the second largest element, and so on, until after n - 1 passes the list is sorted.

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \leq \cdots \leq A_{n-1}}_{\text{in their final positions}}$$

- **ALGORITHM** *BubbleSort(A[0..n - 1])*
- //Input: An array *A*[0..*n* - 1] of orderable elements
- //Output: Array *A*[0..*n* - 1] sorted in nondecreasing order
- **for** $i \leftarrow 0$ **to** $n - 2$ **do**
- **for** $j \leftarrow 0$ **to** $n - 2 - i$ **do**
- **if** $A[j] > A[j+1]$ swap $A[j]$ and $A[j + 1]$
- Time Complexity $C(n) = \sum_{i=0}^{n-2} * \sum_{j=0}^{n-2-i} 1$
- $= \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$
- $= \sum_{i=0}^{n-2} (n - 1 - i)$
- $= \frac{n(n-1)}{2} \in \Theta(n^2)$

- The number of key swaps, however, depends on the input.
- In the worst case of decreasing arrays, it is the same as the number of key comparisons: $\Theta(n^2)$

ALGORITHM *BubbleSort*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j] > A[j+1]$ swap $A[j]$ and $A[j+1]$

8 >? 4	6	9	2	3	1	
4	8 >? 6	9	2	3	1	
4	6	8 > ? 9	2	3	1	
4	6	8	9 >? 2	3	1	
4	6	8	2	9 >? 3	1	
4	6	8	2	3	9 >? 1	
4	6	8	2	3	1	9

4 > ? 6	8	2	3	1	9	
4	6 > ? 8	2	3	1	9	
4	6	8 >? 2	3	1	9	
4	6	2	8 > ? 3	1	9	
4	6	2	3	8 > ? 1	9	
4	6	2	3	1	8	9

4 >? 6	2	3	1	8	9	
4	6 >? 2	3	1	8	9	
4	2	6 > ? 3	1	8	9	
4	2	3	6 > ? 1	8	9	
4	2	3	1	6	8	9
⋮						
1	2	3	4	6	8	9

Radix Sort:

- Sorting algorithms called ***radix sorts*** are linear but in terms of the total number of input bits.
- These algorithms work by comparing individual bits or pieces of keys rather than keys.
- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

101	22	18	64	77	99	59	132	564	580	339	7

- There are maximum 3 digit so 3 passes starting from LSB to MSB
- Pass 1:

580	101	22, 132		64, 564			7, 77	18	59, 99, 339
0	1	2	3	4	5	6	7	8	9

- Pass 2:

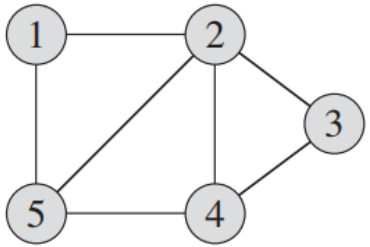
7, 101	18	22	132, 339		59	64, 564	77	580	99
0	1	2	3	4	5	6	7	8	9

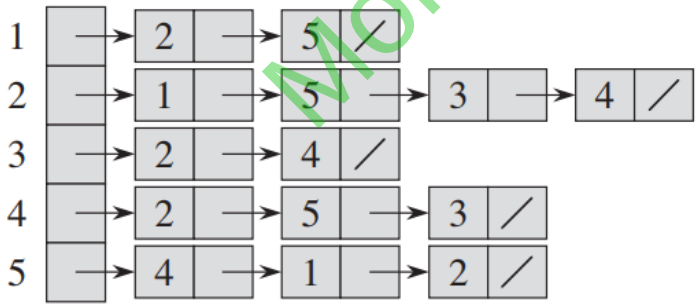
- Pass 3:

7,18,22,59, 64,77,99	101, 132		339		564, 580				
0	1	2	3	4	5	6	7	8	9

- Time Complexity O(n+k)
- n=number of element
- k=digit size of maximum number

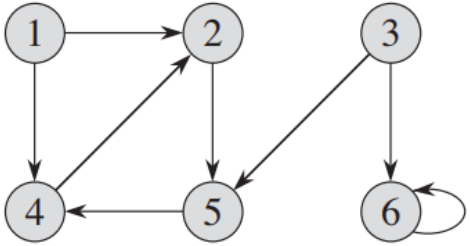
- **Representations of graphs**
- Two standard ways to represent a graph G=(V, E) as an adjacency lists or as an adjacency matrix.
- Where V=set of all vertices in G. E=set of all edges in G.
- Either way applies to both directed and undirected graphs.
- Undirected graph: Edges without direction.Directed graph: Edges with direction.
- |V|=Number of vertices in G / Order of graph
- |E|=Number of edges in G/Size pf graph
- Adjacency-list representation provides a compact way to represent ***sparse*** graphs—those for which |E| is much less than $|V|^2$.
- Adjacency-matrix representation, when the graph is ***dense***—|E| is close to $|V|^2$.
- Weighted Graph: Edges with weight.

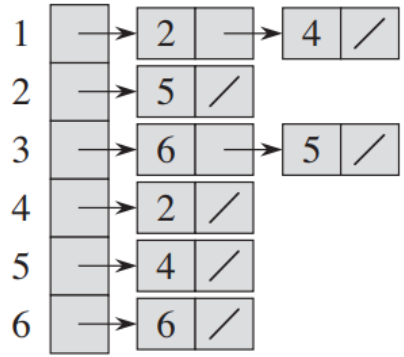




	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(a) Undirected graph , **(b)**Adjacency-list representation, **(c)**Adjacency-matrix representation





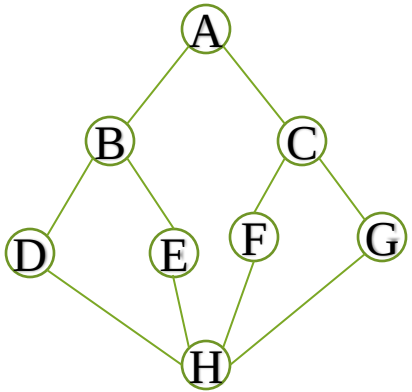
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

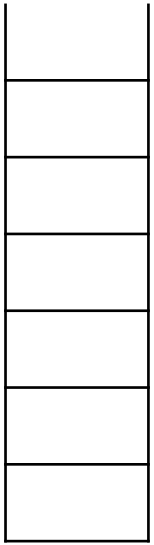
- **(a)D**irected graph **(b)A**djacency-list representation **(c)A**djacency-matrix representation .
- The ***adjacency-list representation*** of a graph G=(V,E) consists of an array *Adj* of |V| lists, one for each vertex in V .
- For each u∈ V , *Adj* [u] consists of all the vertices adjacent to u in G.
- **Adjacency-matrix representation** of a graph G consists of a |V| ×|V| matrix
- A =(a_{ij}) such that
- a_{ij} =1 if (i,j) ∈ E , a_{ij} = 0 otherwise.
- **Depth-first search**
- The strategy followed by depth-first search is, to search “deeper”.
- Depth-first search explores edges out of the most recently discovered vertex *v* that still has unexplored edges leaving it.

- Once all of *v*'s edges 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 vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this entire process until it has discovered every vertex
- Depth-first search also ***timestamps*** each vertex.
- Each vertex *v* has two timestamps: the first timestamp *v.d* records when *v* is first discovered , and the second timestamp *v.f* records when the search finishes examining v's adjacency list.
- These timestamps are integers between 1 and 2|V|, since there is one discovery event and one finishing event for each of the |V| vertices. For every vertex u, ***u.d*** < ***u.f***
- Every vertex u has been assigned a discovery time *u.d* and a finishing time *u.f* .
- It is convenient to use a **stack** to trace the operation of depth-first search.
- We push a vertex onto the stack when the vertex is reached for the first time called **discovery time.**
- We pop a vertex off the stack when it becomes a dead end called **finishing time.**
- DFS traversal is similar to **preorder** traversal in case of Tree.

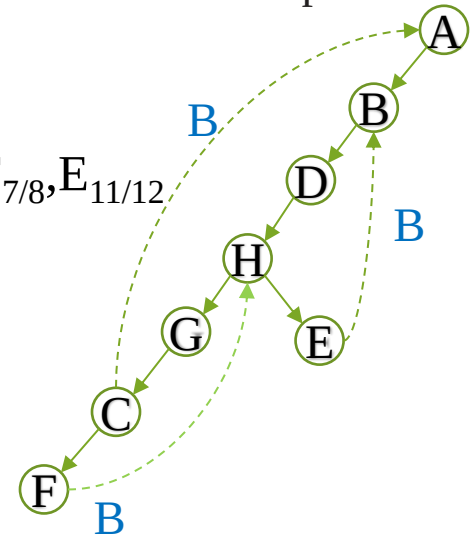
- **ALGORITHM** *DFS(G)*
- //Input: Graph $G = (V,E)$
- //Output: Graph G with its vertices marked with consecutive integers in the order they are first
  //encountered by the DFS traversal
- mark each vertex in V with 0 as a mark of being "unvisited"
- *count* ← 0
- **for** each vertex v in V **do**
- **if** v is marked with 0
- *dfs(v)*
- *dfs(v)*
- //visits recursively all the unvisited vertices connected to vertex v by a path and numbers them
  //in the order they are encountered via global variable *count*
- *count* ← *count* + 1; mark v with *count*
- **for** each vertex w in V adjacent to v **do**
- **if** w is marked with 0; *dfs(w)*
- For the *adjacency matrix representation*, the traversal time is in $\Theta(|V|^2)$,
  and for the *adjacency list representation*, it is in $\Theta(|V| + |E|)$ where $|V|$
  and $|E|$ are the number of the graph's vertices and edges, respectively .

- DFS forms a ***depth-first forest*** comprising several ***depth-first trees***.
- We can define four edge types in **depth-first forest** .
- 1. ***Tree edges*** are edges in the depth-first forest. Edge *(u,v)* is a tree edge if *v* was first discovered by exploring edge (u,v).
- 2. ***Back edges*** are those edges *(u,v)* connecting a vertex u to an ancestor in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
- 3. ***Forward edges*** are those nontree edges *(u,v)* connecting a vertex u to a descendant in a depth-first tree.
- 4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.
- Example 1:



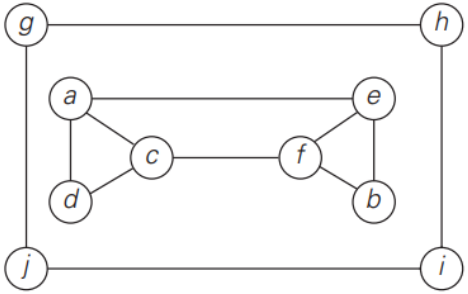


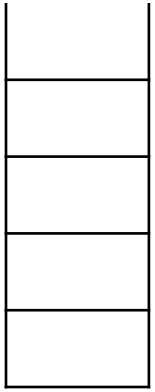
- DFS traversal
- $A_{1/16}$,$B_{2/15}$, $D_{3/14}$, $H_{4/13}$, $G_{5/10}$, $C_{6/9}$, $F_{7/8}$,$E_{11/12}$
- 1:A,B,D,H,G,C,F,E
- Some more DFS sequence are
- 2:A,B,E,H,D,F,C,G,
- 3:H,G,C,F,A,B,D,E
- 4:A,C,F,H,G,E,B,D

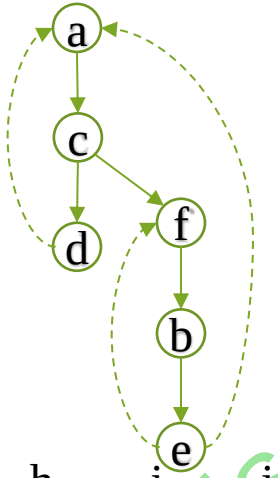


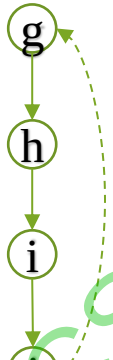
(a) Graph , **(b)** Traversal's stack , **(c)** DFS sequence **(d)** DFS tree

- Example 2:

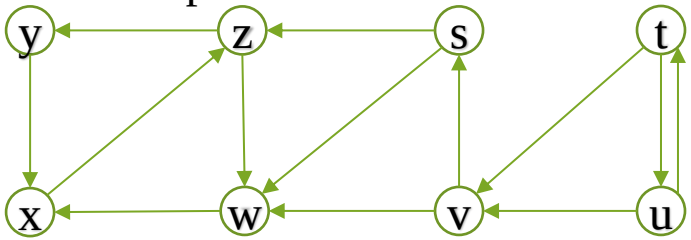






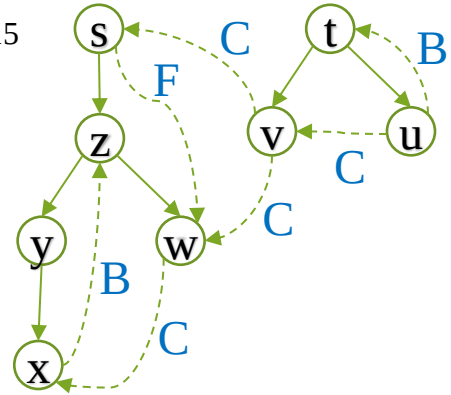


- Example 3:



- DFS traversal with time stamp
- $s_{1/10}$, $z_{2/9}$, $y_{3/6}$,$x_{4/5}$, $w_{7/8}$, $t_{11/16}$,$v_{12/13}$, $u_{14/15}$

- DFS traversal with time stamp
- $a_{1/12}$,$c_{2/11}$, $d_{3/4}$, $f_{5/10}$, $b_{6/9}$, $e_{7/8}$, $g_{13/20}$,$h_{14/19}$, $i_{15/18}$, $j_{16/17}$



- ***Theorem (Parenthesis theorem)***
- In any depth-first search of a (directed or undirected) graph G=(V,E) , for any two vertices u and v, exactly one of the following three conditions holds:
- the intervals [*u.d*,*u.f*] and [*v.d*,*v.f*] are entirely disjoint, and neither *u* nor *v* is a descendant of the other in the depth-first forest,
- the interval [*u.d*,*u.f*] is contained entirely within the interval [*v.d*,*v.f*], and *u* is a descendant of *v* in a depth-first tree, or
- the interval [*v.d*,*v.f*] is contained entirely within the interval [*u.d*,*u.f*] , and *v* is a descendant of *u* in a depth-first tree.

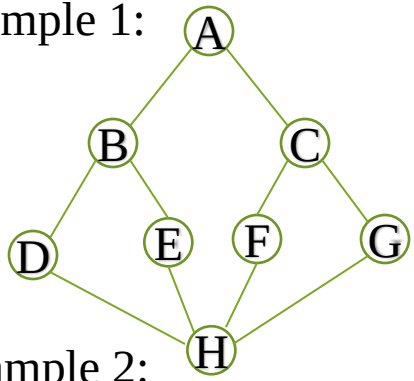
- **Applications of DFS** include checking connectivity and checking acyclicity of a graph.
- Checking a graph's **connectivity** can be done as follows.
- Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the vertices of the graph will have been visited.
- If they have, the graph is connected; otherwise, it is not connected.
- We can use DFS for identifying connected components of a graph .
- As for checking for a **cycle** presence in a graph, we can take advantage of the graph's representation in the form of a DFS forest. If the latter does not have back edges, the graph is clearly acyclic.
- If there is a back edge from some vertex *u* to its ancestor *v* , the graph has a cycle that comprises the path from *v* to *u* via a sequence of tree edges in the DFS forest followed by the back edge from *u* to *v.*
- Other applications of DFS are finding articulation points of a graph,(A vertex of a connected graph is said to be its ***articulation point*** if its removal with all edges incident to it breaks the graph into disjoint pieces.)
- Ex: Consider Discovery & finishing time of a 4 vertices graph . check connectivity
- A)1/6 , 2/5 , 3/4 ,7/8
- Disconnected, 2 component
- B)1/8 ,2/7 ,3/6 ,4/5
- Connected
- C)1/4 ,2/3 ,5/8 , 6/7
- Disconnected, 2 component
- D)1/2 ,3/4 ,5/6 ,7/8
- Disconnected, 4 component

Breadth-first search

- Given a graph G = (V,E) and a distinguished ***source*** vertex *s*, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from *s*.
- It computes the distance (smallest number of edges) from *s* to each reachable vertex.
- It also produces a "breadth-first tree" with root s that contains all reachable vertices.
- For any vertex reachable from s, the simple path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.
- It is convenient to use a **queue** to trace the operation of breadth-first search.
- The queue is initialized with the traversal's starting vertex & marked as visited.
- On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.
- BFS forest of graph can also have two kinds of edges: ***tree edges*** and ***cross edges***.
- ***Tree edges*** are the ones used to reach previously unvisited vertices.
- ***Cross edges*** connect vertices to those visited before, they connect vertices either on the same or adjacent levels of a BFS tree
- BFS uses just one timestamp discovery time/finishing time.

ALGORITHM *BFS(G)*
//Input: Graph $G = (V , E)$
//Output: Graph G with its vertices marked with consecutive integers in the order they are //visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
  count ← 0
    for each vertex v in V **do**
      if v is marked with 0
   bfs(v)
bfs(v)
//visits all the unvisited vertices connected to vertex v by a path and numbers them in the order //they are visited via global variable *count*
  count ← *count* + 1; mark v with *count* and initialize a queue with v
    while the queue is not empty **do**
      for each vertex w in V adjacent to the front vertex **do**
       if w is marked with 0
       count ← *count* + 1; mark w with *count*
       add w to the queue
       remove the front vertex from the queue
Breadth-first search has the same efficiency as depth-first search:
It is in $\Theta(|V|^2)$ for the adjacency matrix representation and in $\Theta(|V| + |E|)$ for the adjacency list representation.

- Example 1:



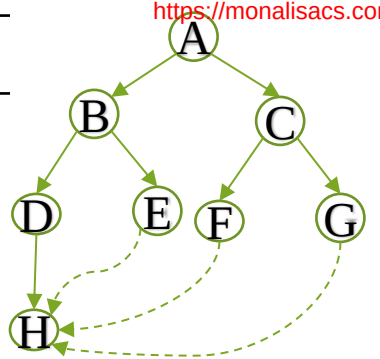
Ex 1 BFS traversal

A_1 ,B_2, C_3, D_4, E_5 , F_6, G_7, H_8

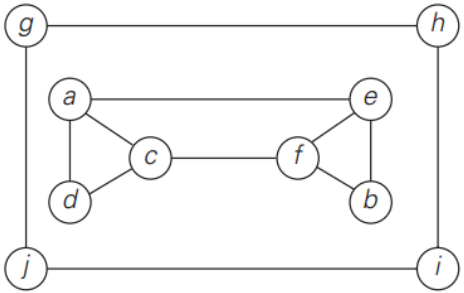
1:A,B,C,D,E,F,G,H

Some more BFS sequence are

2:A,C,B,G,F,D,E,H , 3:H,D,E,F,G,B,C,A , 4:H,G,F,E,D,C,B,A

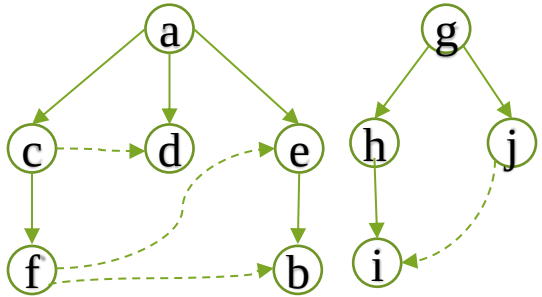


- Example 2:



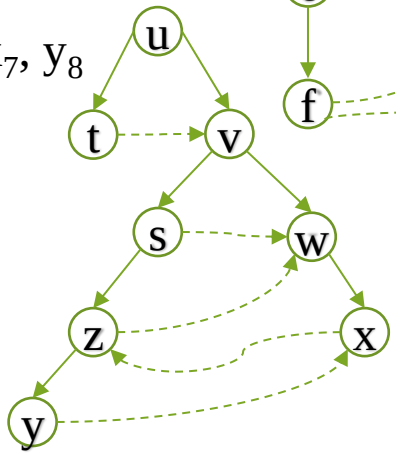
Ex 2 BFS traversal

a_1 ,c_2, d_3, e_4, f_5 , b_6, g_7, h_8 ,j_9,i_{10}

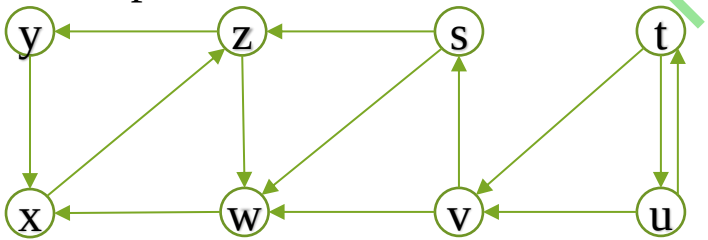


Ex 3 BFS traversal

u_1 ,t_2, v_3, s_4, w_5 , z_6, x_7, y_8



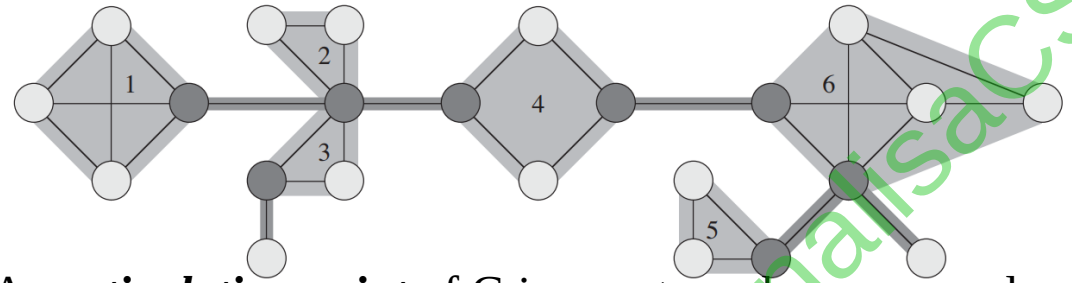
- Example 3:

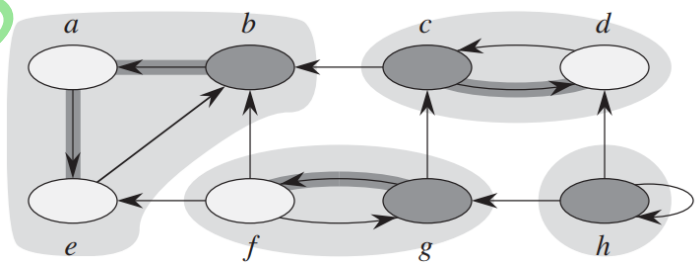


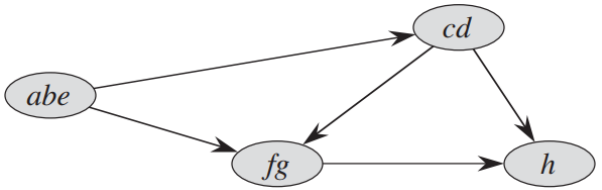
- BFS is similar to **level order** traversal of Tree
- BFS can be used to check connectivity and acyclicity of a graph.
- ***shortest-path distance*** :from s to v as the minimum number of edges in any path from vertex s to vertex v;
- BFS can be used for finding a shortest path with the fewest number of edges between two given vertices.
- **Main facts about depth-first search (DFS) and breadth-first search (BFS)**

	DFS	BFS
Applications	Connectivity ,Acyclicity , Articulation point	Connectivity ,Acyclicity ,shortest path
Algorithm technique	Back tracking	Branch &bound
Number of vertex orderings	Two ordering	One ordering
Edge types(Directed graph)	Tree , back ,forward , cross	Tree , cross edges
Traversal in Tree	Same as Pre Order	Same as level order
Data Structure	stack	queue
Efficiency for adjacency list	adjacency matrix :$\Theta(\|V\|^2)$,adjacency list: $\Theta(\|V\| + \|E\|)$	
Find component of graph	Connected component ,Strongly connected component, Biconnected component.	

- **Graph Component**
- Connected Component :An undirected graph is connected if there is a path between any two vertices in the graph .
- If the graph is not connected then maximal connected subgraph known as connected graph .
- Strongly connected component: of a directed graph G=(V,E) is a maximal set of vertices such that for every pair of vertices *u* and *v* are reachable from each other.
- Every Directed graph is a DAG of its strongly connected component .

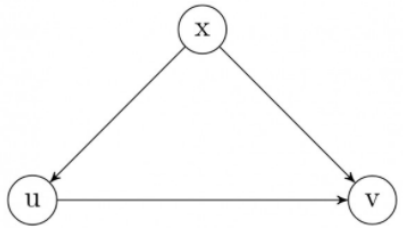






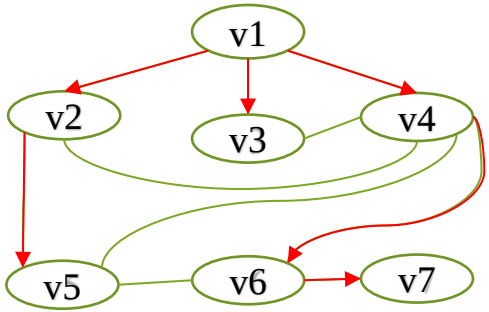
- An ***articulation point*** of G is a vertex whose removal disconnects G.
- A ***bridge*** of G is an edge whose removal disconnects G.
- A ***biconnected component*** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle
- The articulation points are the heavily shaded vertices , the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions.

- Q 1:Consider a BFS of a undirected graph from a node r ,let D(r,u) and D(r,v) represent lengths of shortest path from r to u and v respectively . If u is visited before v then what is valid relation between them.
- Ans : $D(r,u) \leq D(r,v)$
- Q 2:G be a graph with n vertex and k component .If a vertex is removed from G .Then number of component in resultant graph most lie between
- A) k , n B)k-1, k+1 C)k-1, n-1 D)k-1 ,n-k
- Ans : C)k-1, n-1
- GATE IT 2007 | Q 24:Consider a DFS of DAG which of the following is true for all edges $u \rightarrow v$.
- A)d(u) <d(v) B)d(u)<f(v) C)f(u)<f(v) D)f(u)>f(v)



- DFS 1 :$x_{1/6}$,$u_{2/5}$,$v_{3/4}$,
- DFS 2 :$x_{1/6}$,$v_{2/3}$,$u_{4/5}$
- Ans : D)f(u)>f(v)
- GATE IT 2005 | Q 14:In a depth-first traversal of a Graph G with n vertices, k edges are marked as tree edges. The number of connected components in G is A)k B)k+1 C)n-k-1 D)n-k
- Ans: D)n-k

- ISRO2020-32 : G is an undirected graph with vertex set {*v1, v2, v3, v4, v5, v6, v7*}and edge set {*v1v2, v1v3, v1v4 ,v2v4, v2v5, v3v4, v4v5, v4v6, v5v6, v6v7* }. A breadth first search of the graph is performed with v1 as the root node. Which of the following is a tree edge?
- A)*v2v4* B)*v1v4* C)*v4v5* D)*v3v4*



- Ans : B)*v1v4*
- GATE IT 2006 | Q 47: Consider the depth-first-search of an undirected graph with 3 vertices P, Q, and R. Given that
- d(P)=5 units d(Q)=6 units d(R)=14 unit
- f(P)=12 units f(Q)=10 units f(R)=18 units
- Which one of the following statements is TRUE about the graph?
- A) There is only one connected component
- B)There are two connected components, and P and R are connected
- C)There are two connected components, and Q and R are connected
- D)There are two connected components, and P and Q are connected
- $P_{5/12}$,$Q_{6/10}$,$R_{14/18}$
- Two component (P,Q),(R).
- Ans: (D)