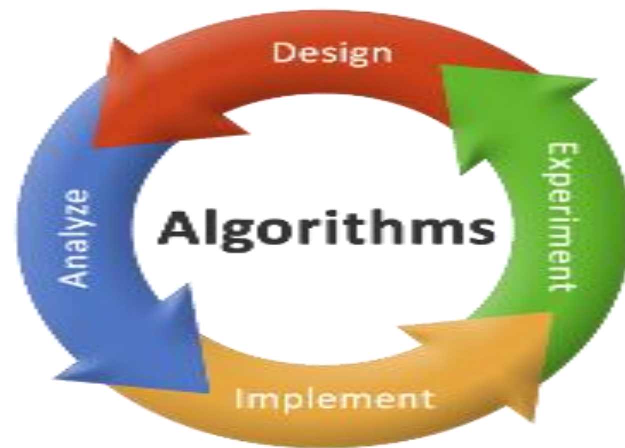
LECTURE NOTES OF ALGORITHMS: DESIGN TECHNIQUES AND ANALYSIS



By

Ass. Prof. Dr. Samaher Al_Janabi

Faculty of Science for Women(SCIW), University of Babylon, Iraq

Samaher@uobabylon.edu.iq

Outlines

GRAPH SEARCH AND CONNECTIVITY

- **Overview**
- **Graph Search - Overview**
- **Depth-First Search (DFS): The Basics**

  (https://www.cs.usfca.edu/~galles/visualization/DFS.html)
- **Breadth-First Search (BFS): The Basics**
- **BFS and Shortest Paths**

(https://www.cs.usfca.edu/~galles/visualization/BFS.html)

- **BFS and Undirected Connectivity**
- **Topological Sort**
- **Computing Strong Components: The Algorithm**
- **Computing Strong Components: The Analysis**

Graph Search - Overview

SUMMARY: This section is all about graph search and its applications. We'll cover a selection of fundamental primitives for reasoning about graphs. One very cool aspect of this material is that all of the algorithms that we'll cover are insanely fast (linear time with small constants); and, it can be quite subtle to understand why they work! The culmination of these lectures --- computing the strongly connected components of a directed graph with just two passes of depth-first search --- we illustrate the point that fast algorithms often require deep insight into the structure of the problem that you're solving.

Dynamic Programming

Many programs in computer science are written to optimize some value; for example, find the shortest path between two points, find the line that best fits a set of points, or find the smallest set of objects that satisfies some criteria. There are many strategies that computer scientists use to solve these problems. One of the goals of this lecture is to expose you to several different problem solving strategies. **Dynamic programming** is one strategy for these types of optimization problems.

Note:

This lecture is start point for sequence of lectures related of optimization problems.

Graph Traversal Methods

In some graph algorithms such as those for finding shortest paths or minimum spanning trees, the vertices and edges are visited in an order that is imposed by their respective algorithms. However, in some other algorithms, the order of visiting the vertices is unimportant; what is important is that the vertices are visited in a systematic order, regardless of the in- put graph. In this lecture, we discuss two methods of graph traversal: depth-first search and breadth-first search.

1. Depth-First Search Algorithm

Depth-first search is a powerful traversal method that aids in the solution of many problems involving graphs. It is essentially a generalization of the preorder traversal of rooted trees. Let G = (V, E) be a directed or undirected graph. A depth-first search traversal of G works as follows. First, all vertices are marked unvisited. Next, a starting vertex is selected, say v ∈ V , and marked visited. Let w be any vertex that is adjacent to v. We mark w as visited and advance to another vertex, say x, that is adjacent to w and is marked unvisited. Again, we mark x as visited and advance to another vertex that is adjacent to x and is marked unvisited. This process of selecting an unvisited vertex adjacent to the current vertex continues as deep as possible until we find a vertex y whose adjacent vertices have all been marked visited. At this point, we back up to the most recently visited vertex, say z, and visit an unvisited vertex that is adjacent to z, if any. Continuing this way, we finally return back to the starting vertex v.

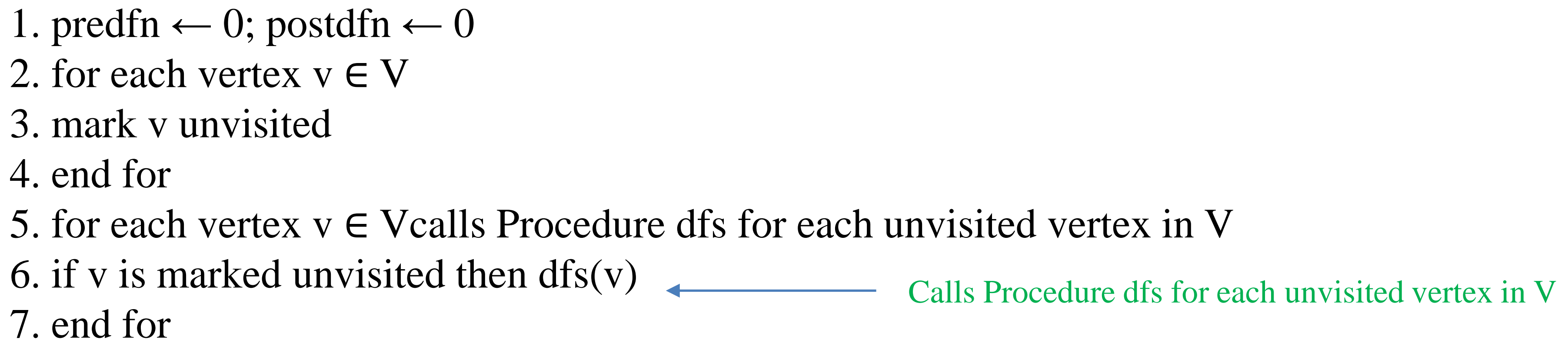
Depth-First Search Algorithm

This method of traversal has been given the name depth- first search, as it continues the search in the forward (deeper) direction. The algorithm for such a traversal can be written using recursion as shown in Algorithm dfs or a stack.

Algorithm DFS

Input: A (directed or undirected) graph G = (V, E).

Output: Preordering and postordering of the vertices in the corresponding depth-first search tree.



1. predfn ← 0; postdfn ← 0
2. for each vertex v ∈ V
3. mark v unvisited
4. end for
5. for each vertex v ∈ Vcalls Procedure dfs for each unvisited vertex in V
6. if v is marked unvisited then dfs(v)
7. end for

Procedure dfs(v)

1. mark v visited
2. predfn ← predfn + 1
3. for each edge (v, w) ∈ E
4. if w is marked unvisited then dfs(w)
5. end for
6. postdfn ← postdfn + 1

Depth-First Search Algorithm

The algorithm starts by marking all vertices unvisited. It also initializes two counters predfn and postdfn to zero. These two counters are not part of the traversal; their importance will be apparent when we later make use of depth-first search to solve some problems. The algorithm then calls Procedure dfs for each unvisited vertex in V . This is because not all the vertices may be reachable from the start vertex. Starting from some vertex v ∊ V , Procedure dfs performs the search on G by visiting v, marking v visited and then recursively visiting its adjacent vertices. When the search is complete, if all vertices are reachable from the start vertex, a spanning tree called the depth-first search spanning tree is constructed whose edges are those inspected in the forward direction, i.e., when exploring unvisited vertices. In other words, let (v, w) be an edge such that w is marked unvisited and suppose the procedure was invoked by the call dfs(v). Then, in this case, that edge will be part of the depth-first search spanning tree.

If not all the vertices are reachable from the start vertex, then the search results in a forest of spanning trees instead. After the search is complete, each vertex is labeled with predfn and postdfn numbers. These two labels impose preorder and postorder numbering on the vertices in the spanning tree (or forest) generated by the depth-first search traversal. They give the order in which visiting a vertex starts and ends. In the following, we say that edge (v, w) is being explored to mean that within the call dfs(v), the procedure is inspecting the edge (v, w) to test whether w has been visited before or not. The edges of the graph are classified differently according to whether the graph is directed or undirected.

The case of undirected graphs

Let G = (V, E) be an undirected graph. As a result of the traversal, the edges of G are classified into the following two types:

• Tree edges: edges in the depth-first search tree. An edge (v, w) is a tree edge if w was first visited when exploring the edge (v, w).

• Back edges: All other edges.

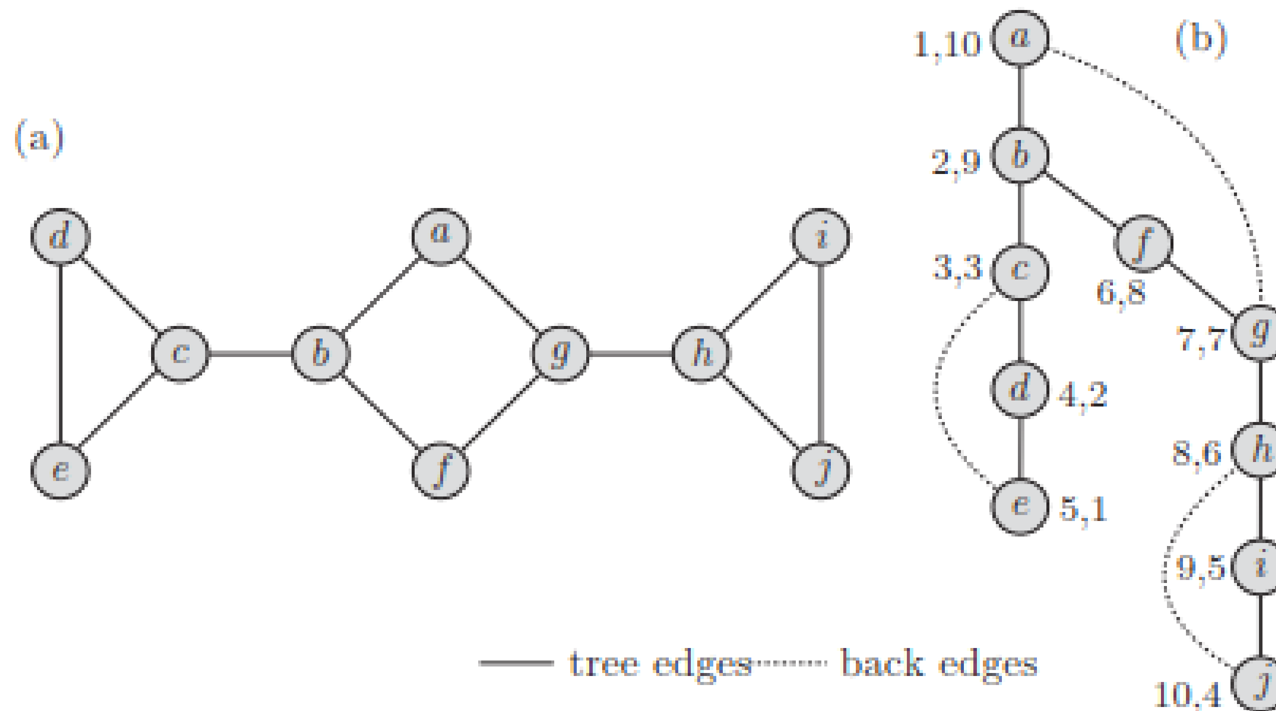


Figure :An example of depth-first search traversal of an undirected graph.

The case of undirected graphs

Figure (b) illustrates the action of depth-first search traversal on the undirected graph shown in Fig. (a). Vertex a has been selected as the start vertex. The depth-first search tree is shown in Fig. (b) with solid lines. Dotted lines represent back edges. Each vertex in the depth-first search tree is labeled with two numbers: predfn and postdfn. Note that since vertex e has postdfn = 1, it is the first vertex whose depth-first search is complete. Note also that since the graph is connected, the start vertex is labeled with predfn = 1 and postdfn = 10, the number of vertices in the graph

The case of directed graphs

Depth-first search in directed graphs results in one or more (directed) span- ning trees whose number depends on the start vertex. If v is the start vertex, depth-first search generates a tree consisting of all vertices reachable from v. If not all vertices are included in that tree, the search resumes from an- other unvisited vertex, say w, and a tree consisting of all unvisited vertices that are reachable from w is constructed. This process continues until all vertices have been visited. In depth-first search traversal of directed graphs, however, the edges of G are classified into four types:

- • Tree edges: edges in the depth-first search tree. An edge (v, w) is a tree edge if w was first visited when exploring the edge (v, w).
- Back edges: edges of the form (v, w) such that w is an ancestor of v in the depth-first search tree (constructed so far) and vertex w was marked visited when (v, w) was explored.
- Forward edges: edges of the form (v, w) such that w is a descendant of v in the depth-first search tree (constructed so far) and vertex w was marked visited when (v, w) was explored.
- Cross edges: All other edges.

The case of directed graphs

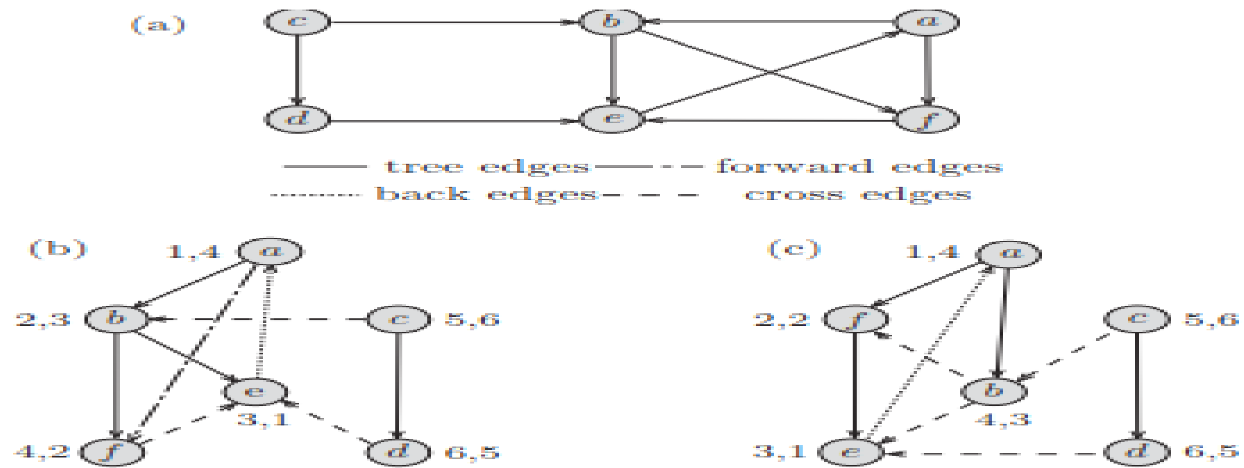
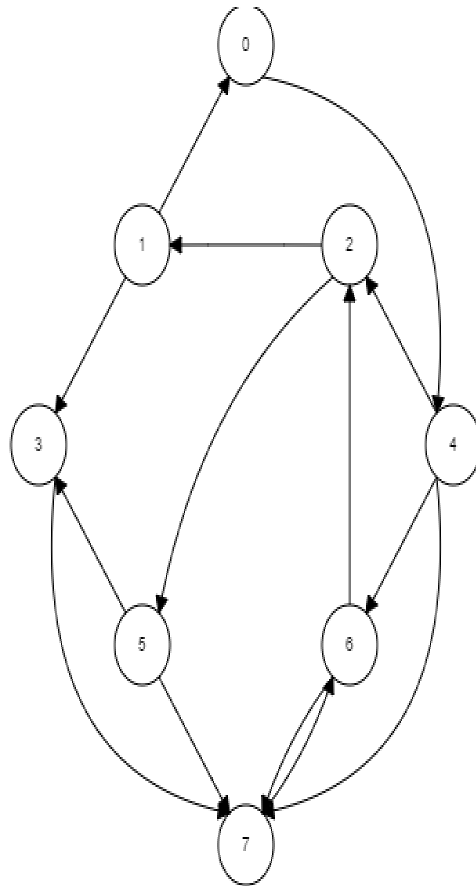


Figure : An example of depth-first search traversal of a directed graph.

Figure (b) illustrates the action of depth-first search traver- sal on the directed graph shown in Fig. (a). Starting at vertex a, the vertices a, b, e and f are visited in this order. When Procedure dfs is initiated again at vertex c, vertex d is visited and the traversal is complete after b is visited from c. We notice that the edge (e, a) is a back edge since e is a descendant of a in the depth-first search tree, and (e, a) is not a tree edge. On the other hand, edge (a, f) is a forward edge since a is an ancestor of f in the depth-first search tree, and (a, f) is not a tree edge. Since neither e nor f is an ancestor of the other in the depth-first search tree, edge (f, e) is a cross edge. The two edges (c, b) and (d, e) are, obviously, cross edges; each edge connects two vertices in two different trees. Note that had we chosen to visit vertex f immediately after a instead of visiting vertex b, both edges (a, b) and (a, f) would have been tree edges. In this case, the result of the depth-first search traversal is shown in Fig. (c). Thus the type of an edge depends on the order in which the vertices are visited.

DFS Applies on directed graphs

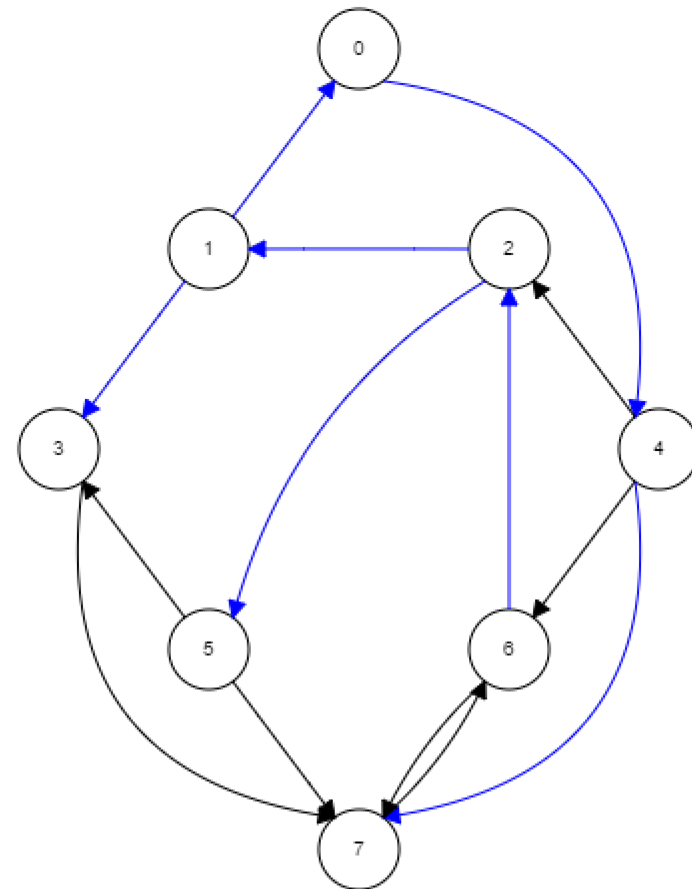
	Parent		Visited
0		0	f
1		1	f
2		2	f
3		3	f
4		4	f
5		5	f
6		6	f
7		7	f



Let Start Vertex: 6
When apply DFS, we get:
Please visit the link:
https://www.cs.usfca.edu/~galles/visualization/DFS.html

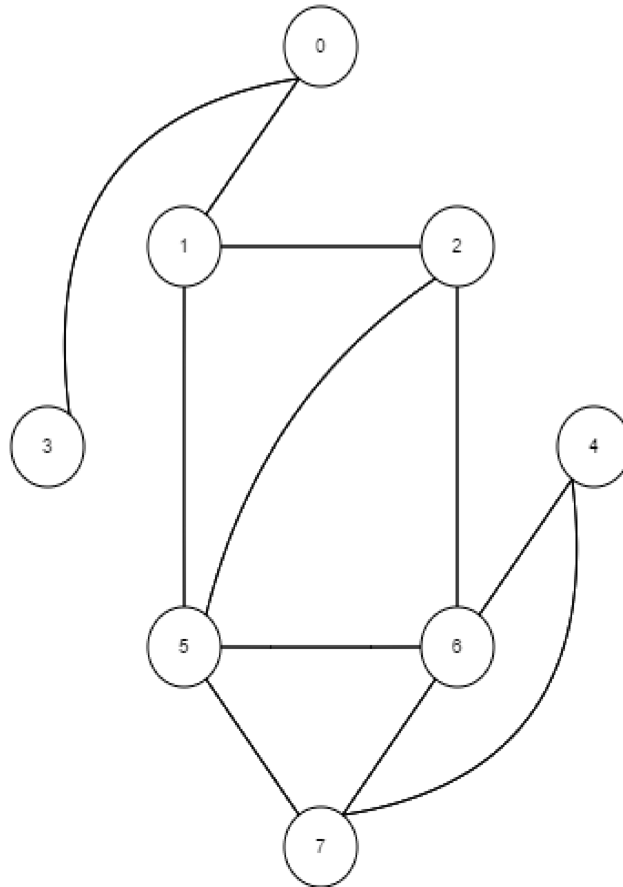
DFS(6)
 DFS(2)
  DFS(1)
   DFS(0)
    DFS(4)
     DFS(7)
   DFS(3)
  DFS(5)

	Parent		Visited
0	1	0	T
1	2	1	T
2	6	2	T
3	1	3	T
4	0	4	T
5	2	5	T
6		6	T
	4	7	T



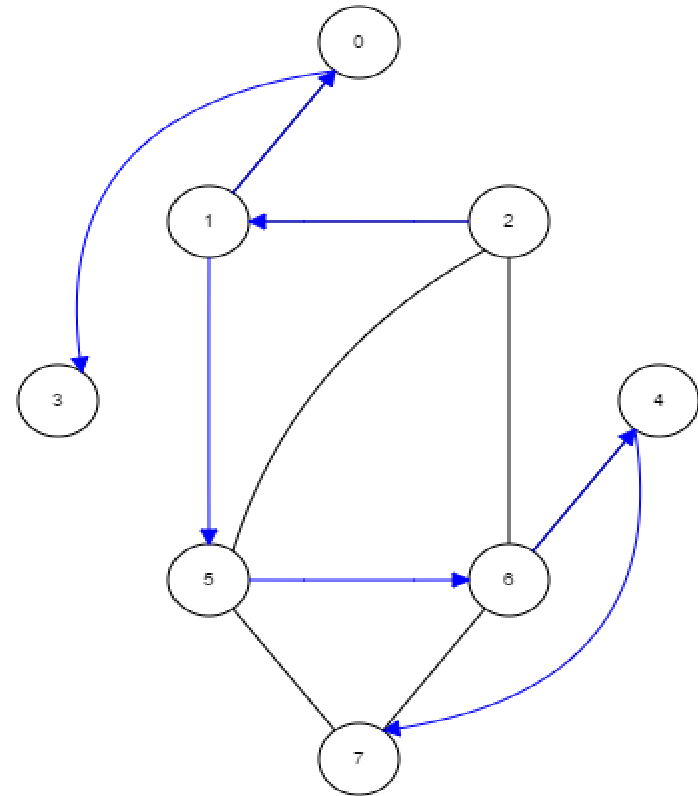
DFS Applies on undirected graphs

	Parent		Visited
0		0	f
1		1	f
2		2	f
3		3	f
4		4	f
5		5	f
6		6	f
7		7	f



DFS(2)
- DFS(1)
  - DFS(0)
    - DFS(3)
  - DFS(5)
    - DFS(6)
      - DFS(4)
        - DFS(7)

	Parent		Visited
0	1	0	T
1	2	1	T
2		2	T
3	0	3	T
4	6	4	T
5	1	5	T
6	5	6	T
7	4	7	T



Let Start Vertex: 2
When apply DFS, we get:
Please visit the link:
https://www.cs.usfca.edu/~galles/visualization/DFS.html

Time complexity of depth-first search

Now we analyze the time complexity of Algorithm dfs when applied to a graph G with n vertices and m edges. The number of procedure calls is exactly n since once the procedure is invoked on vertex v, it will be marked visited and hence no more calls on v will take place. The cost of a procedure call if we exclude the for loop is $\Theta(1)$. It follows that the overall cost of procedure calls excluding the for loop is $\Theta(n)$. Steps 1 and 2 of the algorithm cost $\Theta(1)$ and $\Theta(n)$ time, respectively. The cost of Step 3 of the algorithm to test whether a vertex is marked is $\Theta(n)$. Now, it remains to find the cost of the for loop in Procedure dfs. The number of times this step is executed to test whether a vertex w is marked unvisited is equal to the number of vertices adjacent to vertex v. Hence, the total number of times this step is executed is equal to the number of edges in the case of directed graphs and twice the number of edges in the case of undirected graphs. Consequently, the cost of this step is $\Theta(m)$ in both directed and undirected graphs. It follows that the running time of the algorithm is $\Theta(m + n)$. If the graph is connected or $m \geq n$, then the running time is simply $\Theta(m)$. It should be emphasized, however, that the graph is assumed to be represented by adjacency lists.

Time complexity = $\Theta(m + n)$.

while If the graph is connected or $m \geq n$,

Time complexity = $\Theta(m)$

2. Breadth-First Search

Unlike depth-first search, in breadth-first search when we visit a vertex v we next visit all vertices adjacent to v. The resulting tree is called a breadth-first search tree. This method of traversal can be implemented by a queue to store unexamined vertices. Algorithm bfs for breadth-first search can be applied to directed and undirected graphs. Initially, all vertices are marked unvisited. The counter bfn, which is initialized to zero, represents the order in which the vertices are removed from the queue. In the case of undirected graphs, an edge is either a tree edge or a cross edge. If the graph is directed, an edge is either a tree edge, a back edge or a cross edge: there are no forward edges.

Algorithm BFS
Input: A directed or undirected graph $G = (V, E)$.
Output: Numbering of the vertices in breadth-first search order.

```
1.  bfn ← 0
2.  for each vertex v ∈ V
3.      mark v unvisited
4.  end for
5.  for each vertex v ∈ V
6.      if v is marked unvisited then bfs(v)      ← Calls Procedure BfS for each unvisited vertex in V
7.  end for
```

Procedure bfs(v)

```
1.  Q ← {v}
2.  mark v visited
3.  while Q ≠ {}
4.      v ← Pop(Q)
5.      bfn ← bfn + 1
6.      for each edge (v, w) ∈ E
7.          if w is marked unvisited then
8.              Push(w, Q)
9.              mark w visited
10.         end if
11.     end for
12. end while
```

Breadth-First Search

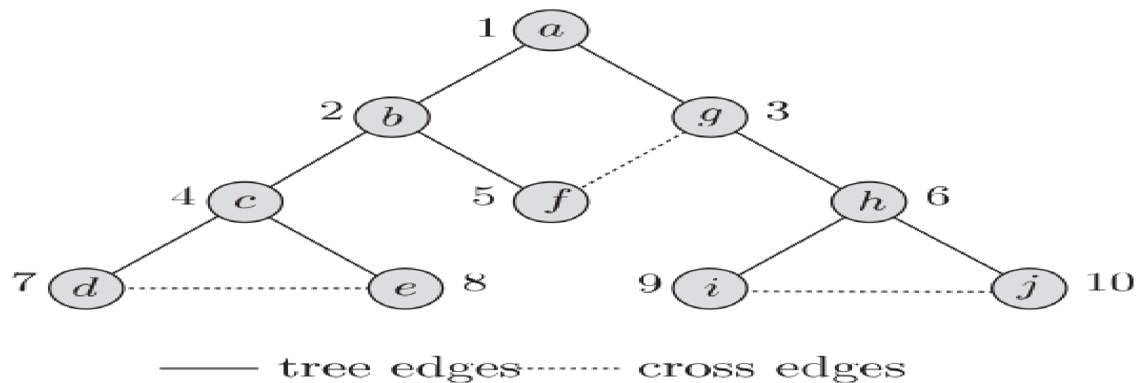


Figure :An example of breadth-first search traversal of an undirected graph

Figure illustrates the action of breadth-first search traversal when applied on the graph shown in Fig. 1(a) starting from vertex a. After popping off vertex a, vertices b and g are pushed into the queue and marked visited. Next, vertex b is removed from the queue, and its adjacent vertices that have not yet been visited, namely c and f, are pushed into the queue and marked visited. This process of pushing vertices into the queue and removing them later on is continued until vertex j is finally removed from the queue. At this point, the queue becomes empty and the breadth-first search traversal is complete. In the figure, each vertex is labeled with its bfn number, the order in which that vertex was removed from the queue. Notice that the edges in the figure are either tree edges or cross edges.

Time complexity

The time complexity of breadth-first search when applied to a graph (directed or undirected) with n vertices and m edges is the same as that of depth-first search, i.e., $\Theta(n + m)$. If the graph is connected or $m \geq n$, then the time complexity is simply $\Theta(m)$.

Time complexity = $\Theta(m + n)$.

While If the graph is connected or $m \geq n$,

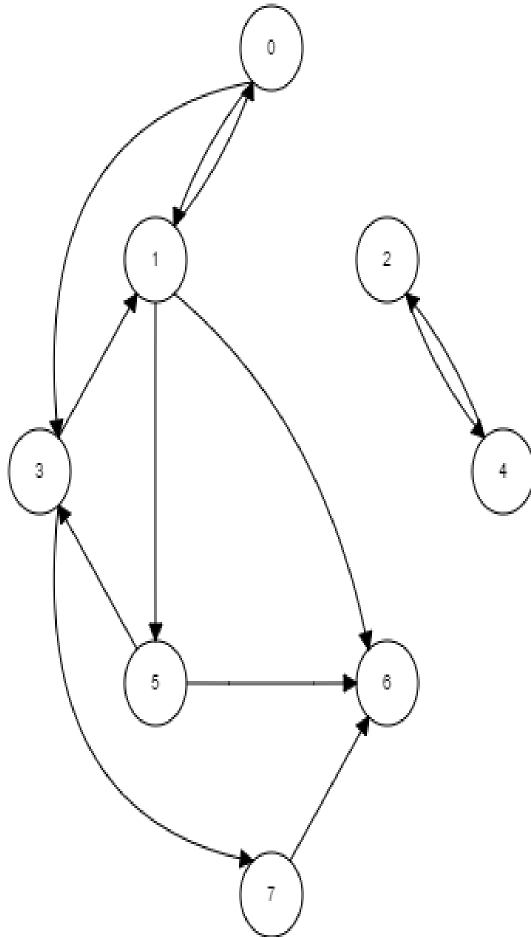
Time complexity = $\Theta(m)$

Applications of Breadth-First Search

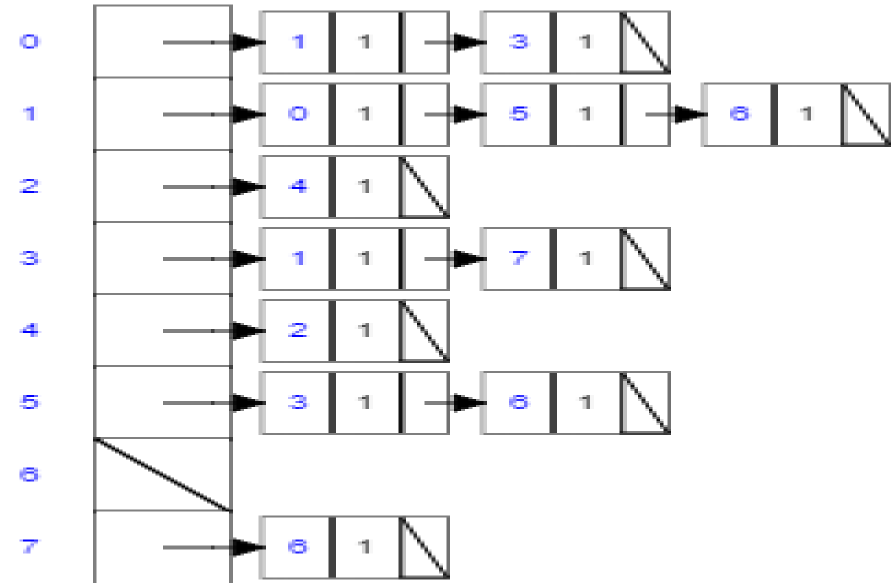
an application of breadth-first search that is important in graph and network algorithms. Let G = (V, E) be a connected undirected graph and s a vertex in V . When Algorithm bfs is applied to G starting at s, the resulting breadth-first search tree is such that the path from s to any other vertex has the least number of edges. Thus, suppose we want to find the distance from s to every other vertex, where the distance from s to a vertex v is defined to be the least number of edges in any path from s to v. This can easily be done by labeling each vertex with its distance prior to pushing it into the queue. Thus, the start vertex will be labeled 0, its adjacent vertices with 1, and so on. Clearly, the label of each vertex is its shortest distance from the start vertex. For instance, in Fig. 9.7, vertex a will be labeled 0, vertices b and g will be labeled 1, vertices c, f and h will be labeled 2, and finally vertices d, e, i and j will be labeled 3. Note that this vertex numbering is not the same as the breadth-first numbering in the algorithm. The minor changes to the breadth-first search algorithm are

BFS applies on Direct Graph

Parent		Visited	
0		0	f
1		1	f
2		2	f
3		3	f
4		4	f
5		5	f
6		6	f
7		7	f



Adjacency List Representation



Adjacency Matrix Representation

	0	1	2	3	4	5	6	7
0		1		1				
1	1					1	1	
2					1			
3		1						1
4			1					
5				1			1	
6								
7							1	

Let Start Vertex: 5
When apply BFS, we get:
Please visit the link:
https://www.cs.usfca.edu/~galles/visualization/BFS.html

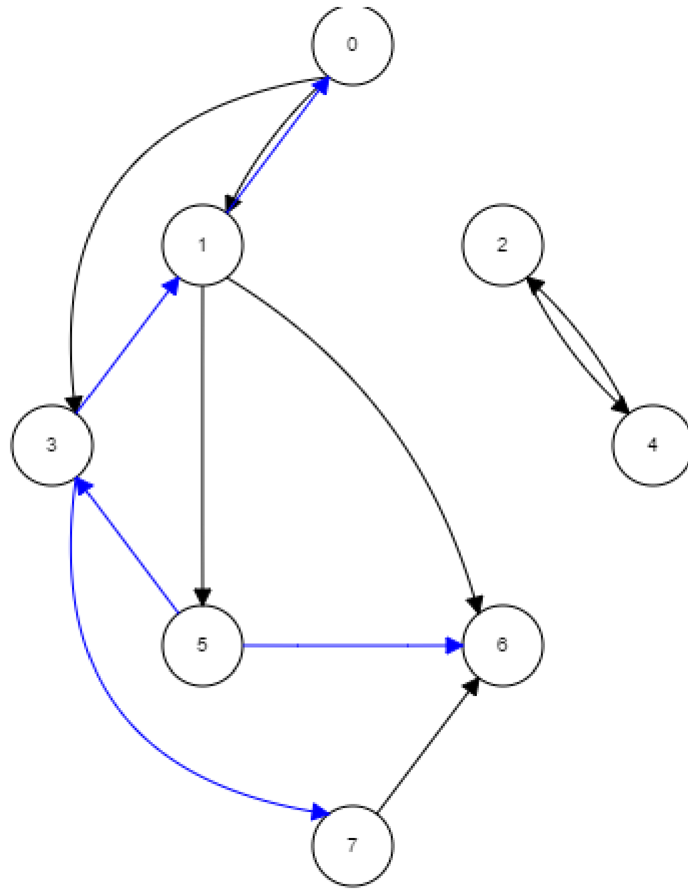
BFS applies on Direct Graph

Parent

0	1
1	3
2	
3	5
4	
5	
6	5
7	3

Visited

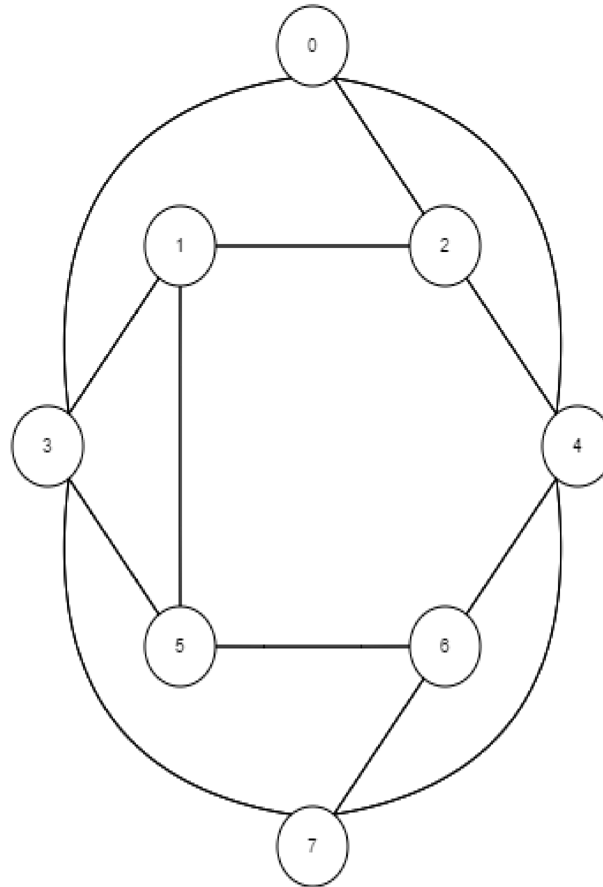
0	T
1	T
2	f
3	T
4	f
5	f
6	T
7	T



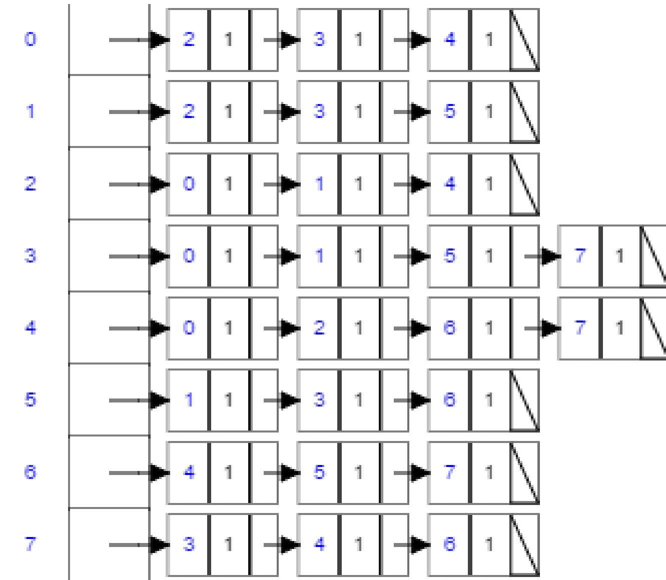
BFS applies on undirected Graph

Parent	
0	
1	
2	
3	
4	
5	
6	
7	

Visited	
0	f
1	f
2	f
3	f
4	f
5	f
6	f
7	f



Adjacency List Representation



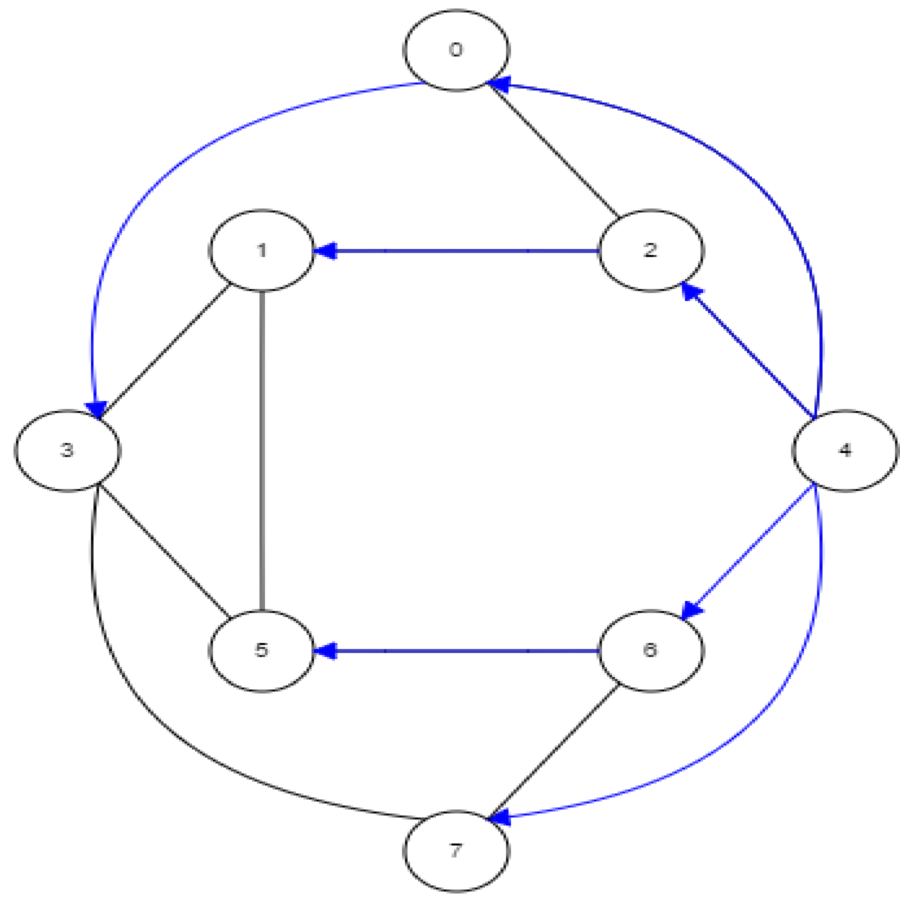
Adjacency Matrix Representation

	0	1	2	3	4	5	6	7
0			1	1	1			
1			1	1		1		
2	1	1			1			
3	1	1				1		1
4	1		1				1	1
5		1		1			1	
6					1	1		1
7				1	1		1	

Let Start Vertex: **4**
When apply BFS, we get:
Please visit the link:
https://www.cs.usfca.edu/~galles/visualization/BFS.html

BFS applies on undirected Graph

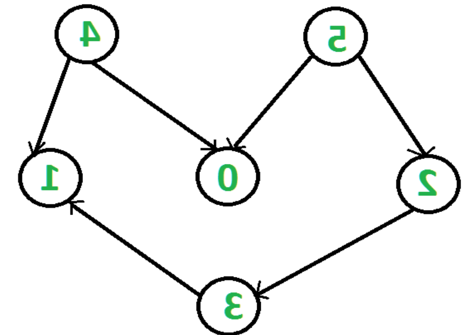
	Parent		Visited
0	4	0	T
1	2	1	T
2	4	2	T
3	0	3	T
4		4	f
5	6	5	T
6	4	6	T
7	4	7	T



Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



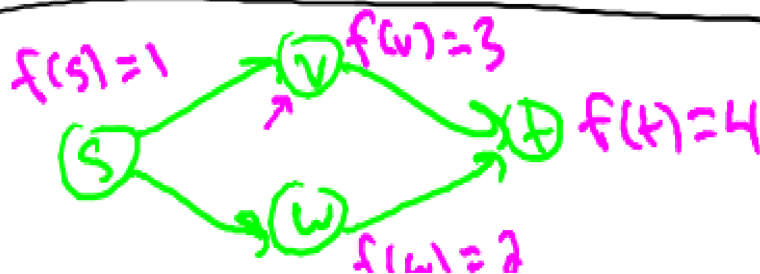
Topological Sorting vs Depth First Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting

Topological Sort via DFS (Slick)

DFS-Loop (graph G)
- mark all nodes unexplored
- current_label = n [to keep track of ordering]
- for each vertex v∈G
  - if v not yet explored [in previous DFS call]
    - DFS(G,v)

DFS (graph G, start vertex s)
- for every edge (s,v)
  - if v not yet explored
    - mark v explored
    - DFS(G,v)
- set f(s) = current_label
- current_label--



Strongly connected component

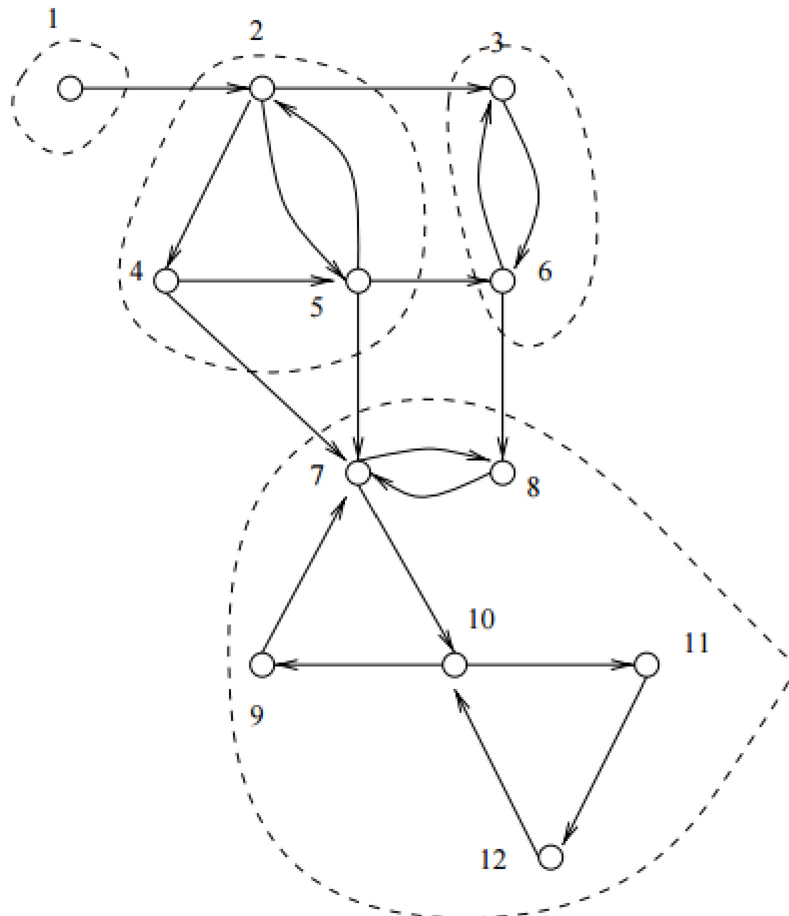
In the mathematical theory of directed graphs, a graph is said to be **strongly connected** or **diconnected** if every vertex is reachable from every other vertex. The **strongly connected components** or **diconnected components** of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.

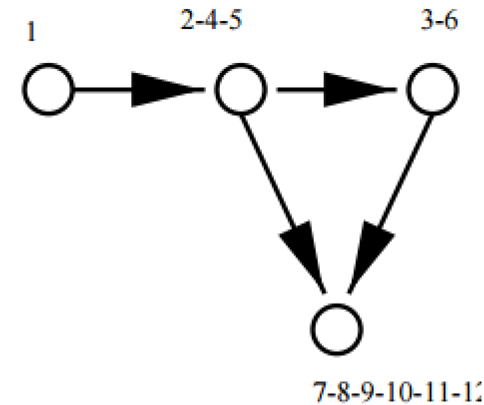
A directed graph is called **strongly connected** if there is a path in each direction between each pair of vertices of the graph. In a directed graph *G* that may not itself be strongly connected, a pair of vertices *u* and *v* are said to be strongly connected to each other if there is a path in each direction between them.

The binary relation of being strongly connected is an equivalence relation, and the induced subgraphs of its equivalence classes are called **strongly connected components**. Equivalently, a **strongly connected component** of a directed graph *G* is a subgraph that is strongly connected, and is maximal with this property: no additional edges or vertices from *G* can be included in the subgraph without breaking its property of being strongly connected. The collection of strongly connected components forms a partition of the set of vertices of *G*.

If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, the **condensation** of *G*. A directed graph is acyclic **if and only if** it has no strongly connected subgraphs with more than one vertex, because a directed cycle is strongly connected and every nontrivial strongly connected component contains at least one directed cycle.

Strongly connected component





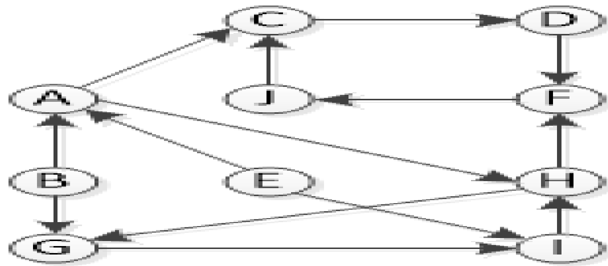
7-8-9-10-11-12

we shrink each of these strongly connected components down to a single node, and draw an edge between two of them if there is an edge from some node in the first to some node in the second, the resulting directed graph has to be a directed acyclic graph (dag) |that is to say, it can have no cycles ((b)). The reason is simple: A cycle containing several strongly connected components would merge them all to a single strongly connected component.

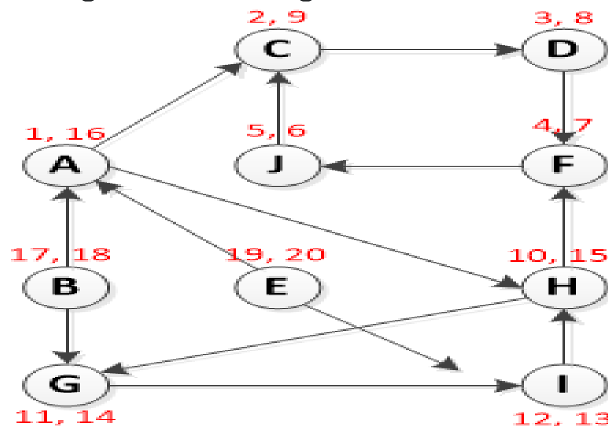
A directed graph and its strongly connected components

How to find Strongly Connected Components (SCC) in a Graph?

1. Call DFS(G) to compute finishing times f[u] for each vertex u
2. Compute Transpose(G)
3. Call DFS(Transpose(G)), but in the main loop of DFS, consider the vertices in order of decreasing f[u] (as computed in step 1)
4. Output the vertices of each tree in the depth-first forest of step 3 as a separate strong connected component: (**Hint: for more details see the below example**)



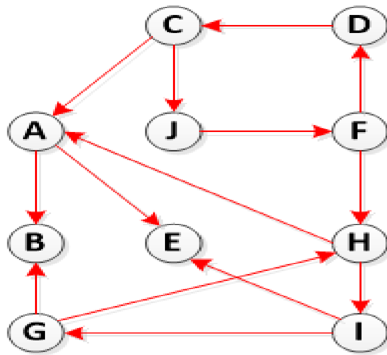
Step 1: Call DFS(G) to compute finishing times f[u] for each vertex u
Running DFS starting on vertex A:



Please notice RED text formatted as [Pre-Vist, Post-Visit]

How to find Strongly Connected Components (SCC) in a Graph?

Step 2: Compute Transpose(G)



Step 3. Call DFS(Transpose(G)), but in the main loop of DFS, consider the vertices in order of decreasing f[u] (as computed in step 1)

Okay, so vertices in order of decreasing post-visit(finishing times) values:

{E, B, A, H, G, I , C, D, F ,J}

So at this step, we run DFS on G^T but start with each vertex from above list:

- ❑ DFS(E): {E}
- ❑ DFS(B): {B}
- ❑ DFS(A): {A}
- ❑ DFS(H): {H, I, G}
- ❑ DFS(G): remove from list since it is already visited
- ❑ DFS(I): remove from list since it is already visited
- ❑ DFS(C): {C, J, F, D}
- ❑ DFS(J): remove from list since it is already visited
- ❑ DFS(F): remove from list since it is already visited
- ❑ DFS(D): remove from list since it is already visited

Step 4: Output the vertices of each tree in the depth-first forest of step 3 as a separate strong connected component.

So we have five strongly connected components: {E}, {B}, {A}, {H, I, G}, {C, J, F, D}

Set of Questions

Q1: Given an adjacency-list representation of a directed graph, where each vertex maintains an array of its outgoing edges (but *not* its incoming edges), how long does it take, in the worst case, to compute the in-degree of a given vertex? As usual, we use n and m to denote the number of vertices and edges, respectively, of the given graph. Also, let k denote the maximum in-degree of a vertex. (Recall that the in-degree of a vertex is the number of edges that enter it.

- **a. Cannot determine from the given information.**
- **b. θ(n)**
- **c. θ(k)**
- **d. θ(m)** ← **Correct answer**

Q2: Consider the following problem: given an undirected graph G with n vertices and m edges, and two vertices s and t, does there exist at least one s-t path?
If G is given in its adjacency list representation, then the above problem can be solved in O(m+n) time, using BFS or DFS. (Make sure you see why this is true.)
Suppose instead that G is given in its adjacency *matrix* representation. What running time is required, in the worst case, to solve the computational problem stated above? (Assume that G has no parallel edges.)

- **a. θ(n2)** ← **Correct answer**
- **b. θ(n∗m)**
- **c. θ(m+n)**
- **d. θ(m+nlogn)**

Set of Questions

Q3: This problem explores the relationship between two definitions about graph distances. In this problem, we consider only graphs that are undirected and connected. The diameter of a graph is the maximum, over all choices of vertices s and t, of the shortest-path distance between s and t. (Recall the shortest-path distance between s and t is the fewest number of edges in an s-t path.) Next, for a vertex s, let l(s) denote the maximum, over all vertices t, of the shortest-path distance between s and t. The radius of a graph is the minimum of l(s) over all choices of the vertex s. Which of the following inequalities always hold (i.e., in every undirected connected graph) for the radius r and the diameter d? [Select all that apply.]

- **a.** $r \geq d/2$ ← **Correct answer**
- **b.** $r \leq d$ ← **Correct answer**
- **c.** $r \geq d$
- **d.** $r \leq d/2$

Q4: Consider our algorithm for computing a topological ordering that is based on depth-first search (i.e., NOT the "straightforward solution"). Suppose we run this algorithm on a graph G that is NOT directed acyclic. Obviously it won't compute a topological order (since none exist).

Does it compute an ordering that minimizes the number of edges that go backward?

For example, consider the four-node graph with the six directed edges (s,v),(s,w),(v,w),(v,t),(w,t),(t,s).

Suppose the vertices are ordered s,v,w,t. Then there is one backwards arc, the (t,s) arc. No ordering of the vertices has zero backwards arcs, and some have more than one.

Set of Questions

a. If and only if the graph is a directed cycle
b. Never
c. Always
d. Sometimes yes, sometimes no ← **Correct answer**

Q5: On adding one extra edge to a directed graph G, the number of strongly connected components...?

a. never decreases by more than 1 (no matter what **G** is)

b. never decreases (no matter what **G** is)

c. will definitely not change (no matter what **G** is)

d. could remain the same (for some graphs **G**) ← **Correct answer**