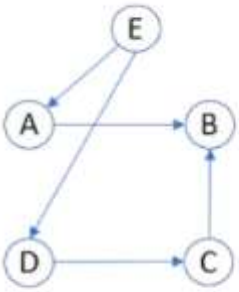
1.Representing Graphs

A graph can be represented using 3 data structures- **adjacency matrix, adjacency list and adjacency set.**

An **adjacency matrix** can be thought of as a table with rows and columns. The row labels and column labels represent the nodes of a graph. An adjacency matrix is a square matrix where the number of rows, columns and nodes are the same. Each cell of the matrix represents an edge or the relationship between two given nodes. For example, adjacency matrix **Aij** represents the number of links from i to j, given two nodes i and j.

	A	**B**	**C**	**D**	**E**
A	0	0	0	0	1
B	0	0	1	0	0
C	0	1	0	0	1
D	1	0	0	1	0
E	0	1	1	0	0

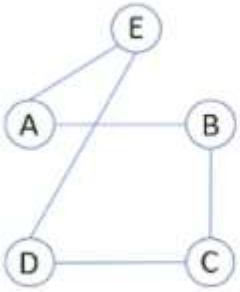
The adjacency matrix for a directed graph is shown in Fig 3. Observe that it is a square matrix in which the number of rows, columns and nodes remain the same (5 in this case). Each row and column correspond to a node or a vertex of a graph. The cells within the matrix represent the connection that exists between nodes. Since, in the given directed graph, no node is connected to itself, all cells lying on the diagonal of the matrix are marked zero. For the rest of the cells, if there exists a directed edge from a given node to another, then the corresponding cell will be marked one else zero.



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	0	0
C	0	1	0	0	0
D	0	0	1	0	0
E	1	0	0	1	0

Fig 3: Adjacency Matrix for a directed graph

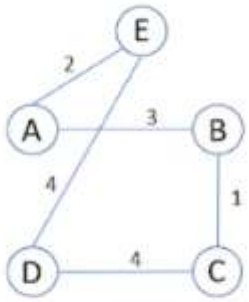
To understand how an undirected graph can be represented using an adjacency matrix, consider a small undirected graph with five vertices (Fig 4). Here, A is connected to B, but B is connected to A as well. Hence, both the cells i.e., the one with source A destination B and the other one with source B destination A are marked one. This suffices the requirement of an undirected edge. Observe that the second entry is at a mirrored location across the main diagonal.



	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Fig 4: Adjacency Matrix for an undirected graph

In case of a weighted graph, the cells are marked with edge weights instead of ones. In Fig 5, the weight assigned to the edge connecting nodes B and D is 3. Hence, the corresponding cells in the adjacency matrix i.e. row B column D and row D column B are marked 3.



	A	B	C	D	E
A	0	3	0	0	2
B	3	0	1	0	0
C	0	1	0	4	0
D	0	0	4	0	4
E	2	0	0	4	0

Fig 5: Adjacency Matrix for a weighted graph

NetworkX library provides an easy method to create adjacency matrices. The following example shows how we can create a basic adjacency matrix using NetworkX.

In **adjacency list** representation of a graph, every vertex is represented as a node object. The node may either contain data or a reference to a linked list. This linked list provides a list of all nodes that are adjacent to the current node. Consider a graph containing an edge connecting node A and node B. Then, the node A will be available in node B's linked list. Fig 6 shows a sample graph of 5 nodes and its corresponding adjacency list.

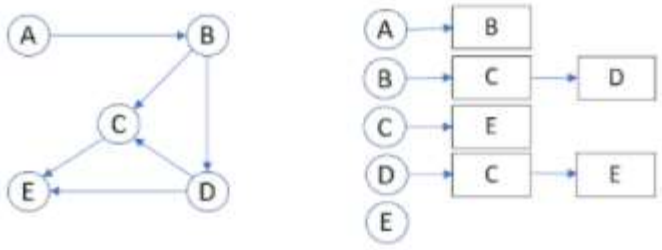


Fig 6: Adjacency list for a directed graph

Note that the list corresponding to node E is empty while lists corresponding to nodes B and D have 2 entries each.

Similarly, adjacency lists for an undirected graph can also be constructed. Fig 7 provides an example of an undirected graph along with its adjacency list for better understanding.

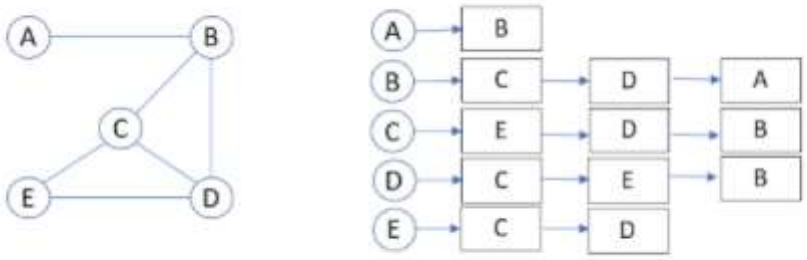


Fig 7: Adjacency list for an undirected graph

Adjacency list enables faster search process in comparison to adjacency matrix. However, it is not the best representation of graphs especially when it comes to adding or removing nodes. For example, deleting a node would involve looking through all the adjacency lists to remove a particular node from all lists. NetworkX library provides a function **adjacency_list ()** to generate the adjacency list of a given graph. The following code demonstrates its use.

The **adjacency set** mitigates a few of the challenges posed by adjacency list. Adjacency set is quite similar to adjacency list except for the difference that instead of a linked list; a set of adjacent vertices is provided. Adjacency list and set are often used for sparse graphs with few connections between nodes. Contrarily, adjacency matrix works well for well-connected graphs comprising many nodes.

This brings us to the end of the blog on graph in data structure. We hope you found this helpful. If you wish to learn more such concepts, you can check out Great Learning Academy's free online courses.

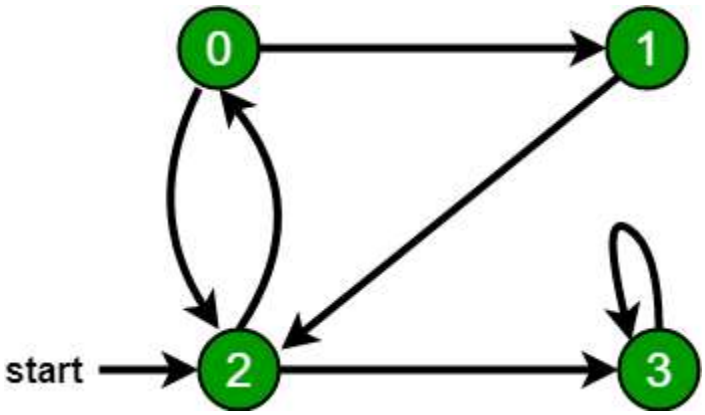
2.Breadth First Search or BFS for a Graph

The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree (See method 2 of this post).
The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.
Example:
In the following graph, we start traversal from vertex 2.



*When we come to **vertex 0**, we look for all adjacent vertices of it.*

- *2 is also an adjacent vertex of 0.*
- *If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.*

There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :

2, 3, 0, 1
2, 0, 3, 1

Implementation of BFS traversal on Graph:

Pseudocode:

Breadth_First_Serach(Graph, X)

Let Q be the queue
Q.enqueue(X)

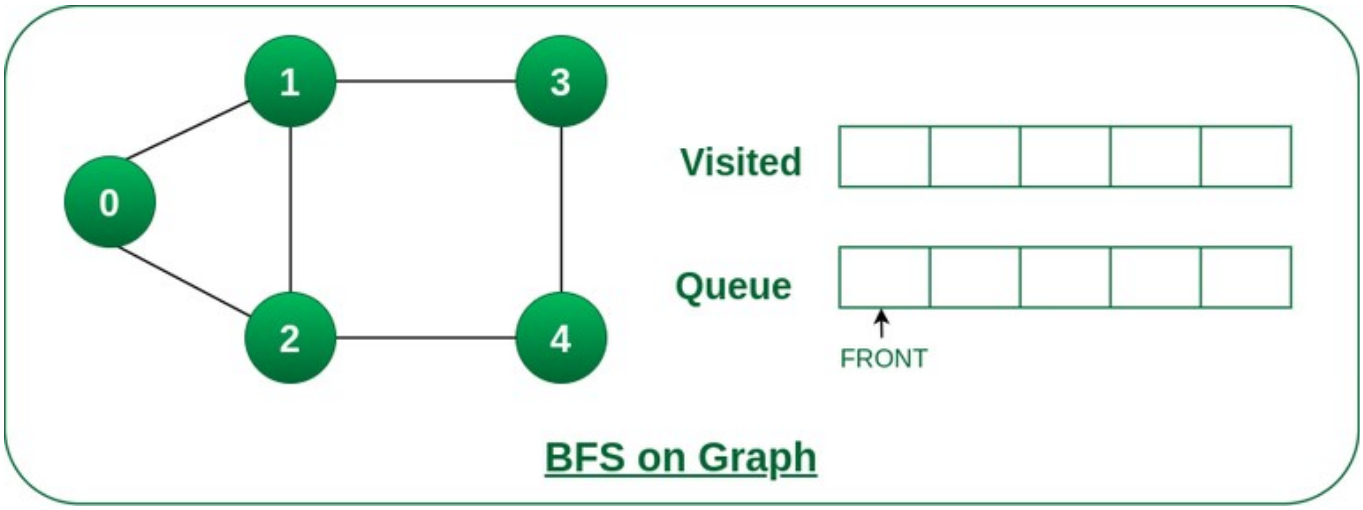
While (Q is not empty)
Y = Q.dequeue( )

Process all the neighbors of Y, For all the neighbors Z of Y
If Z is not visited, Q. enqueue(Z)

Follow the below method to implement BFS traversal.

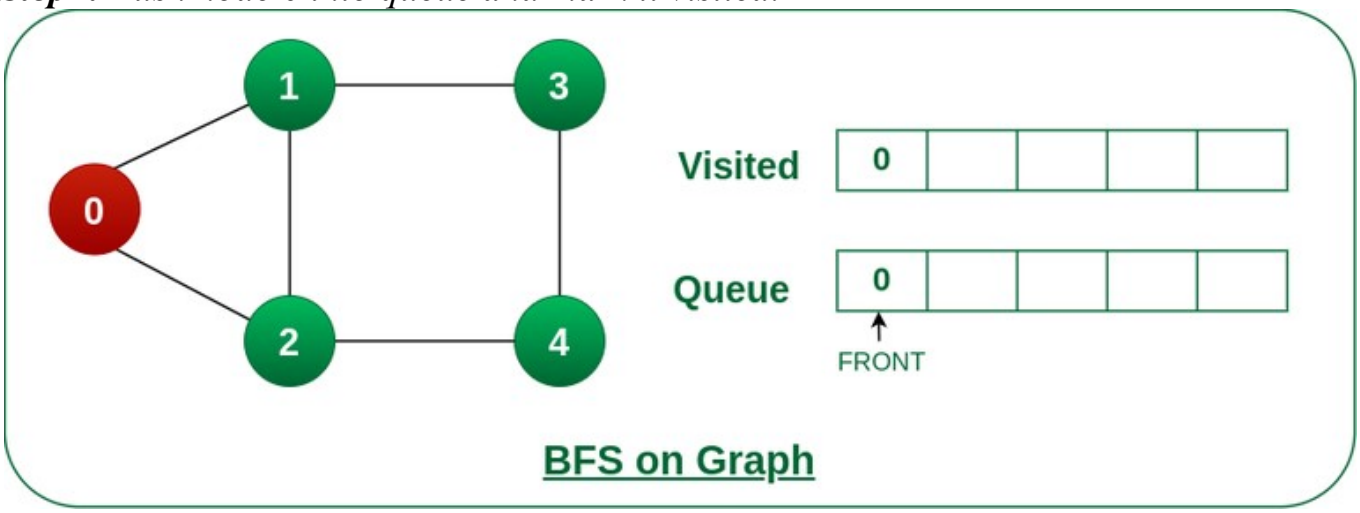
- Declare a queue and insert the starting vertex.
- Initialize a **visited** array and mark the starting vertex as visited.
- Follow the below process till the queue becomes empty:
    - Remove the first vertex of the queue.
    - Mark that vertex as visited.
    - Insert all the unvisited neighbors of the vertex into the queue.

Illustration:
Step1: *Initially queue and visited arrays are empty.*



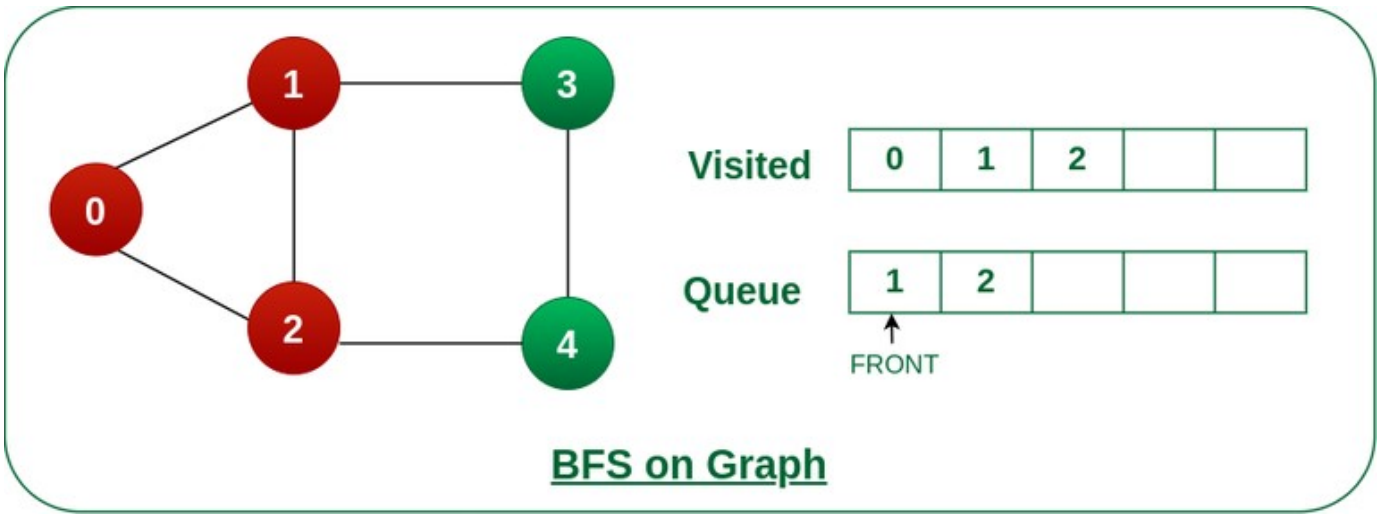
Queue and visited arrays are empty initially.

Step2: *Push node 0 into queue and mark it visited.*



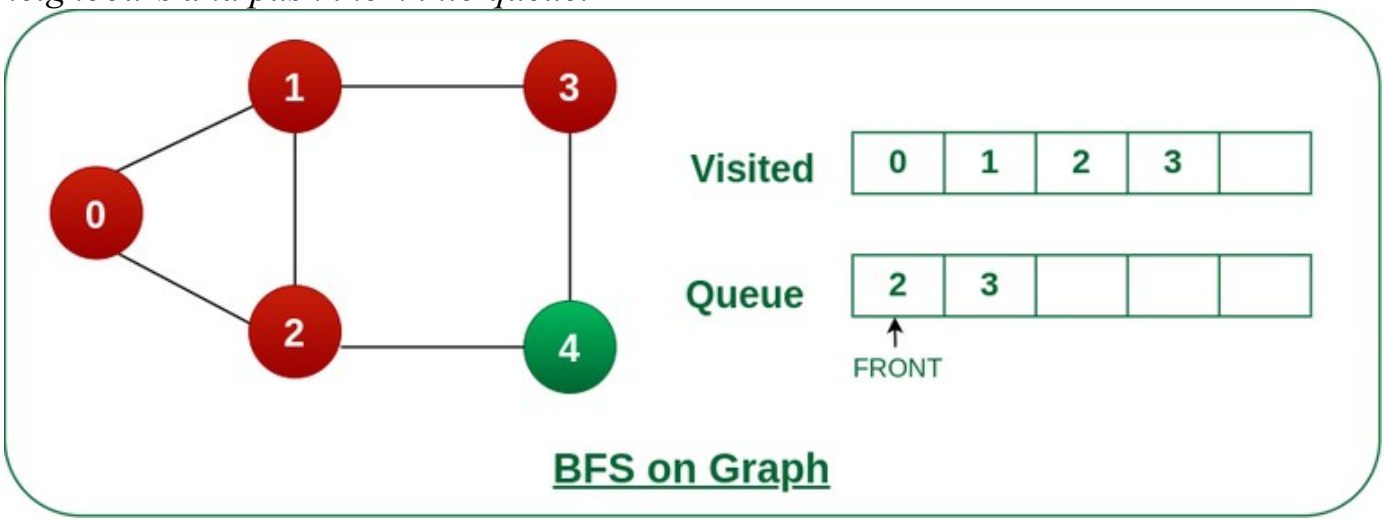
Push node 0 into queue and mark it visited.

Step 3: *Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



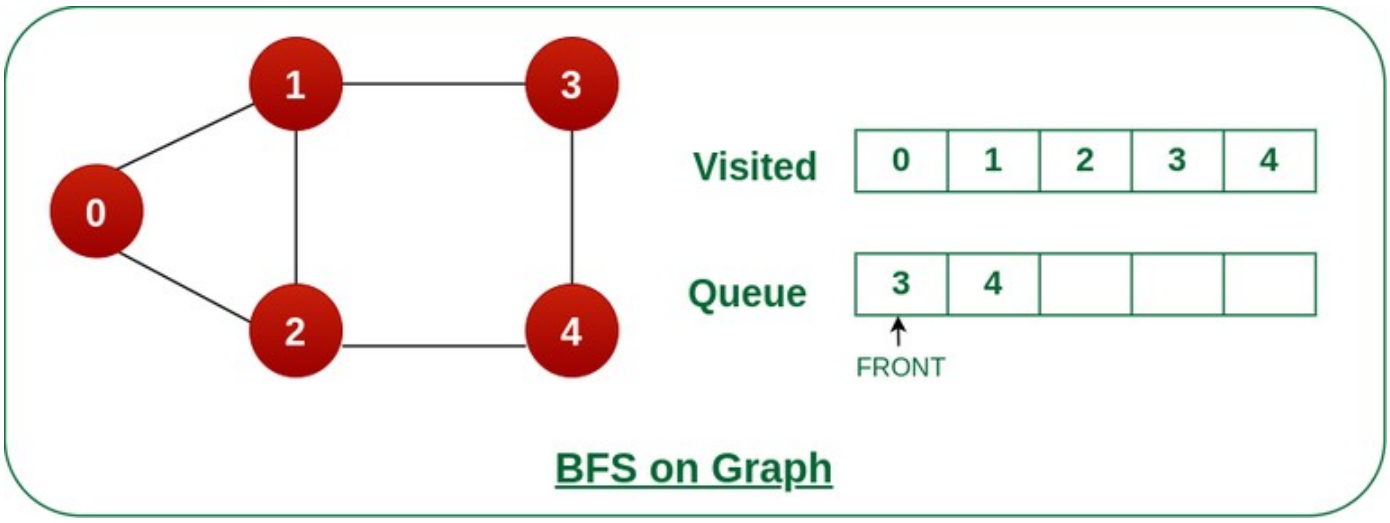
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: *Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



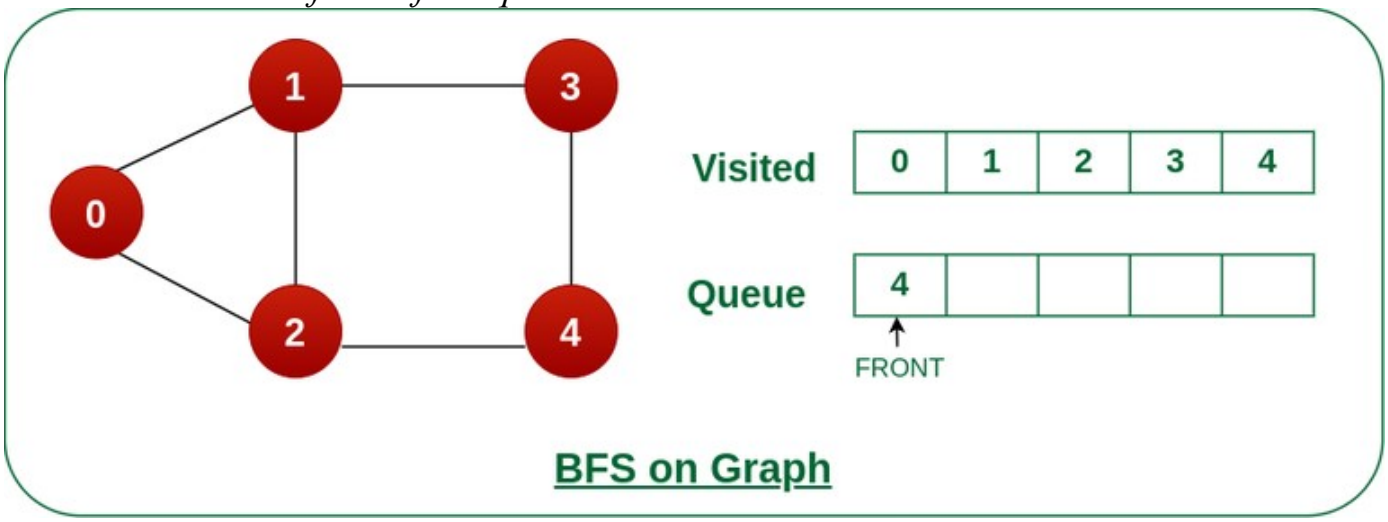
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: *Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



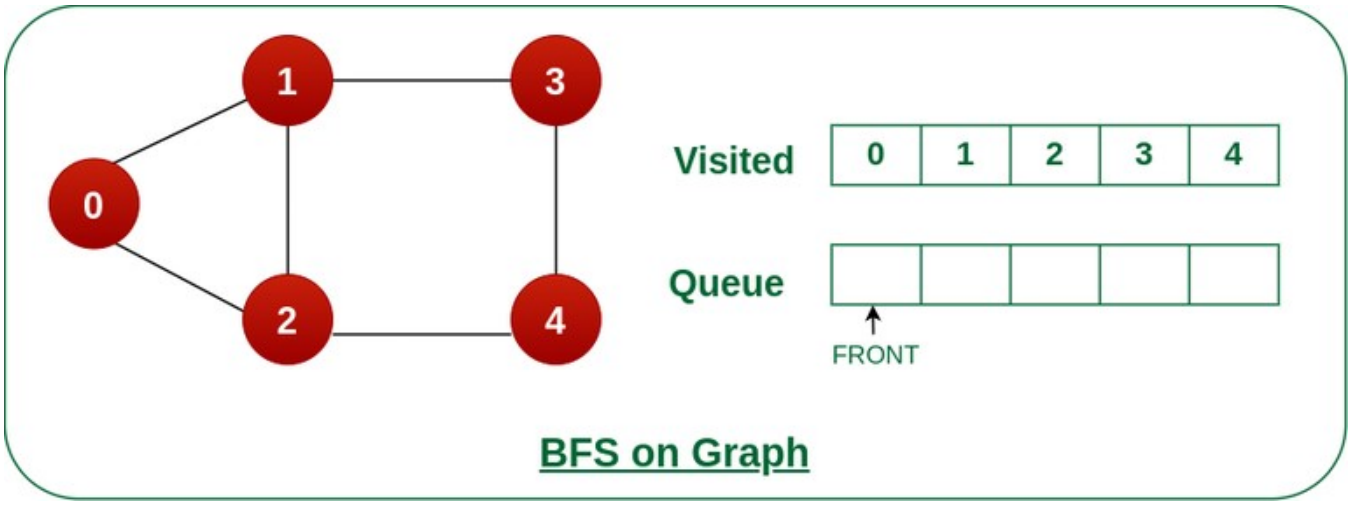
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: *Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: *Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.*
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.

3.Depth First Search

DFS is a recursive traversal algorithm for searching all the vertices of a graph or tree data structure. It starts from the first node of graph G and then goes to further vertices until the goal vertex is reached.

- DFS uses stack as its backend data structure
- edges that lead to an unvisited node are called discovery edges while the edges that lead to an already visited node are called block edges.

DFS procedure

DFS implementation categorizes the vertices in the graphs into two categories:

- Visited
- Not visited

The major objective is to visit each node and keep marking them as “visited” without making any cycle.

Steps for DFS algorithms:

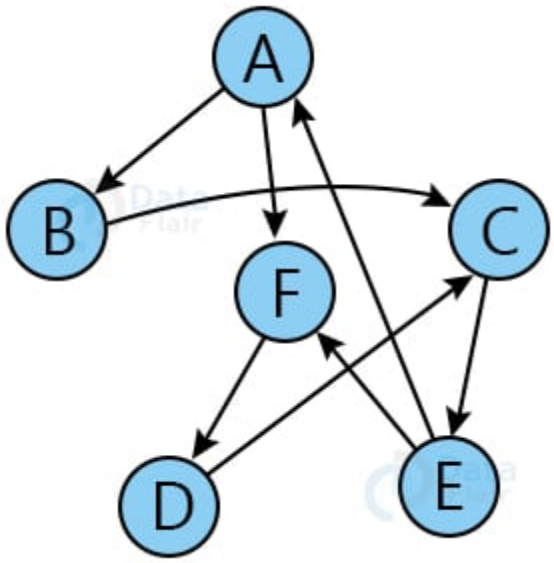
1. Start by pushing starting vertex of the graph into the stack
2. Pop the top item of the stack and add it to the visited list
3. Create the adjacency list for that vertex. Add the non-visited nodes in the list to the top of the stack
4. Keep repeating steps 2 and 3 until the stack is empty

Depth First Search Algorithm

- **Step 1:** STATUS = 1 for each node in Graph G
- **Step 2:** Push the starting node A in the stack. set its STATUS = 2
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N from the stack. Process it and set its STATUS = 3
- **Step 5:** Push all the neighbors of N with STATUS =1 into the stack and set their STATUS = 2
  [END OF LOOP]
- **Step 6:** stop

Working of Depth First Search

Let us consider the graph below:

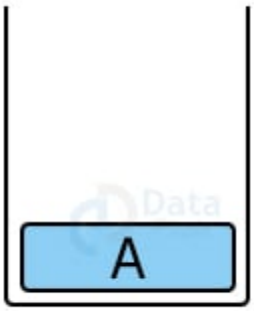


Starting node: A

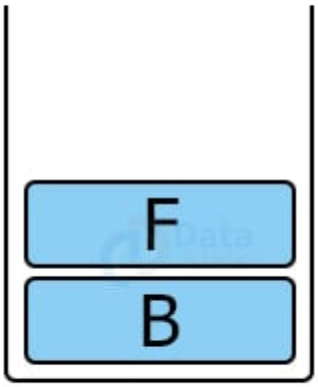
Step 1: Create an adjacency list for the above graph.

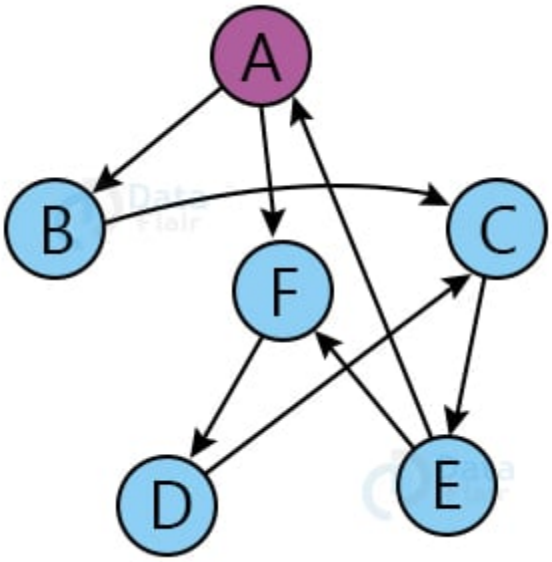
A → B,F
B → C
C → E
D → C
E → A,F
F → D

Step 2: Create an empty stack.
Step 3: Push ‘A’ into the stack

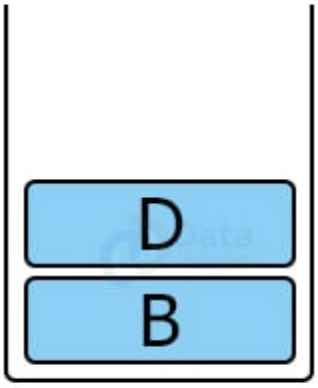


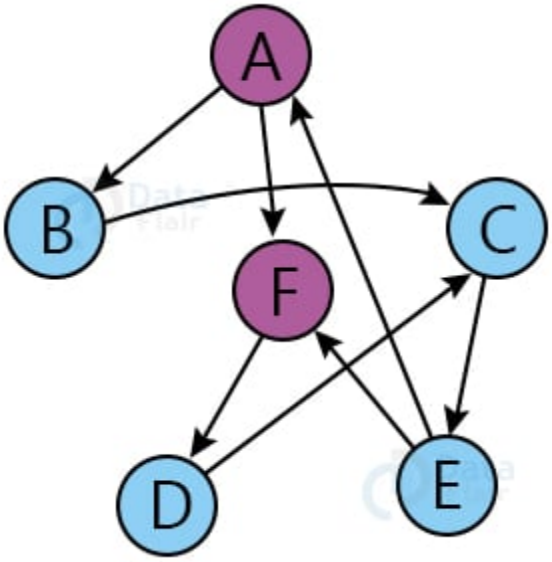
Step 4: Pop ‘A’ and push ‘B’ and ‘F’. Mark node ‘A’ as the visited node



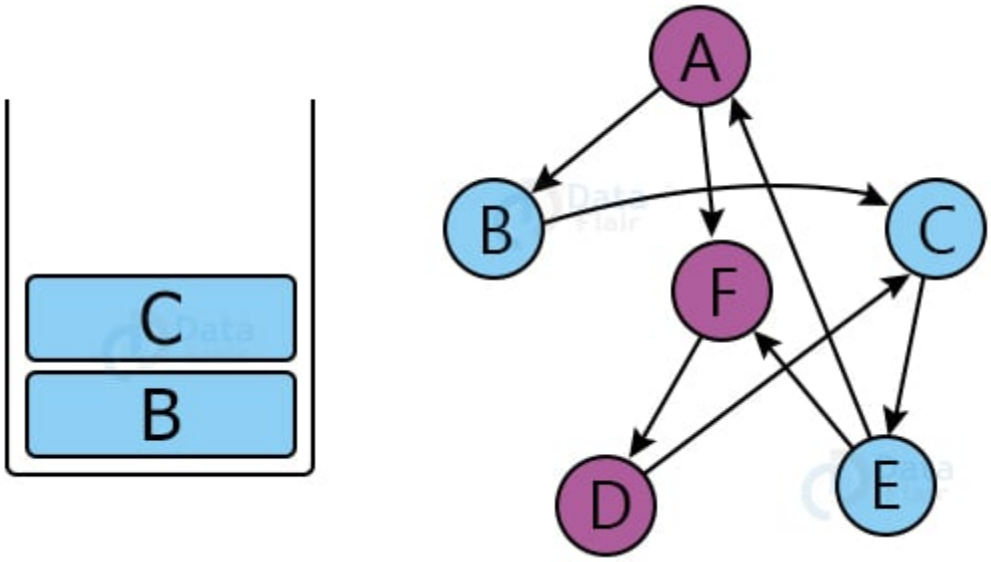


Step 5: pop ‘F’ and push ‘D’. Mark ‘F’ as a visited node.

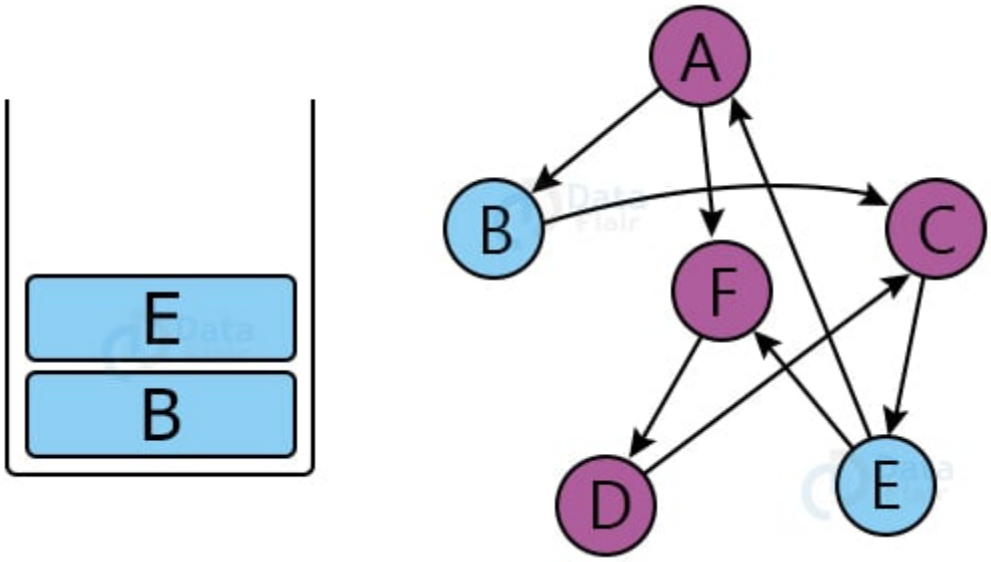




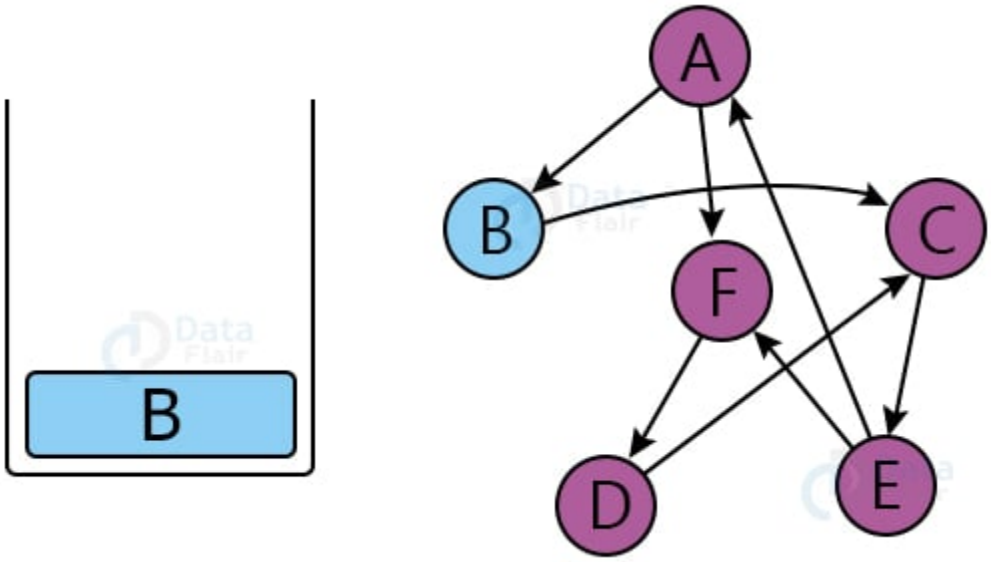
Step 6: pop ‘D’ and push ‘C’. Mark ‘D’ as a visited node.



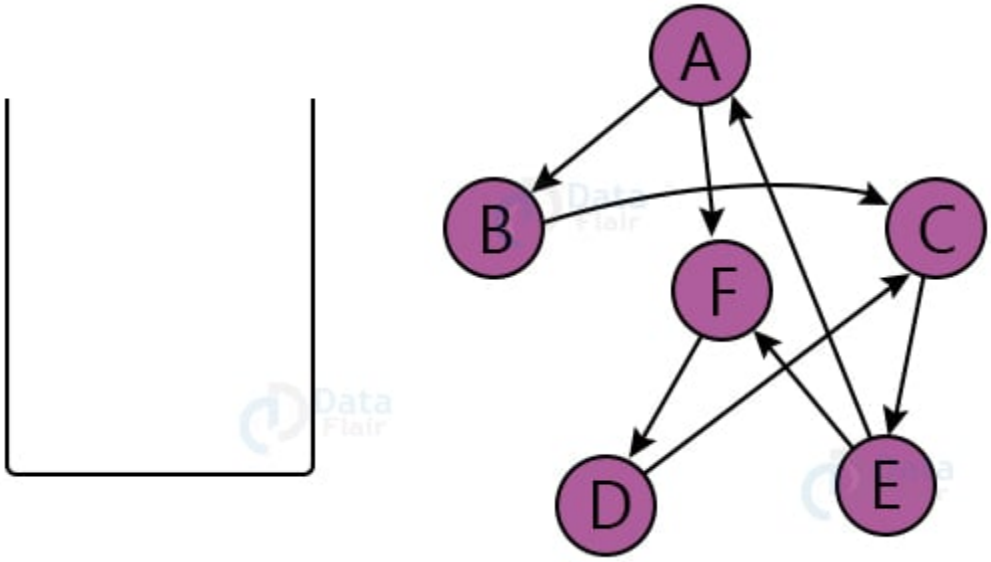
Step 7: pop ‘C’ and push ‘E’. Mark ‘C’ as a visited node.



Step 8: pop ‘E’. Mark ‘E’ as a visited node. No new node is left.

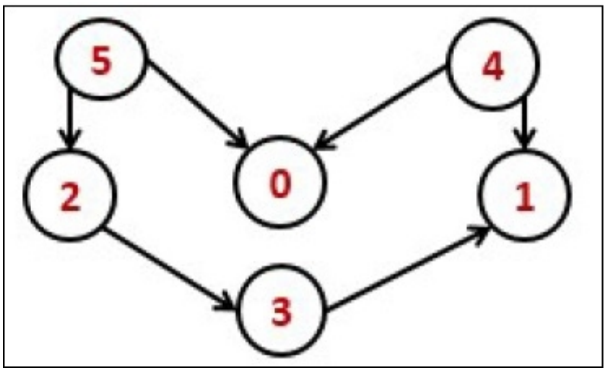


Step 9: pop ‘B’. Mark ‘B’ as visited. All the nodes in the graph are visited now.



4.Topological Sorting

The topological sorting for a directed acyclic graph is the linear ordering of vertices. For every edge U-V of a directed graph, the vertex u will come before vertex v in the ordering.



As we know that the source vertex will come after the destination vertex, so we need to use a stack to store previous elements. After completing all nodes, we can simply display them from the stack.

Input and Output

Input:

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 1 0 0

0 1 0 0 0 0

1 1 0 0 0 0

1 0 1 0 0 0

Output:

Nodes after topological sorted order: 5 4 2 3 1 0

Algorithm

topoSort(u, visited, stack)

Input − The start vertex u, An array to keep track of which node is visited or not. A stack to store nodes.**Output −** Sorting the vertices in topological sequence in the stack.

Begin

mark u as visited

for all vertices v which is adjacent with u, do

```
    if v is not visited, then
      topoSort(c, visited, stack)
  done

  push u into a stack
End
```

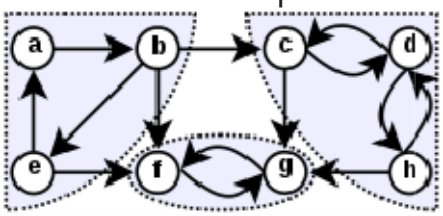
performTopologicalSorting(Graph)

Input − The given directed acyclic graph.**Output −** Sequence of nodes.

```
Begin
  initially mark all nodes as unvisited
  for all nodes v of the graph, do
    if v is not visited, then
      topoSort(i, visited, stack)
  done
  pop and print all elements from the stack
End.
```

5.Strongly connected component

A directed graph is called **strongly connected** if there is a path in each direction between each pair of vertices of the graph. That is, a path exists from the first vertex in the pair to the second, and another path exists from the second vertex to the first. 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.

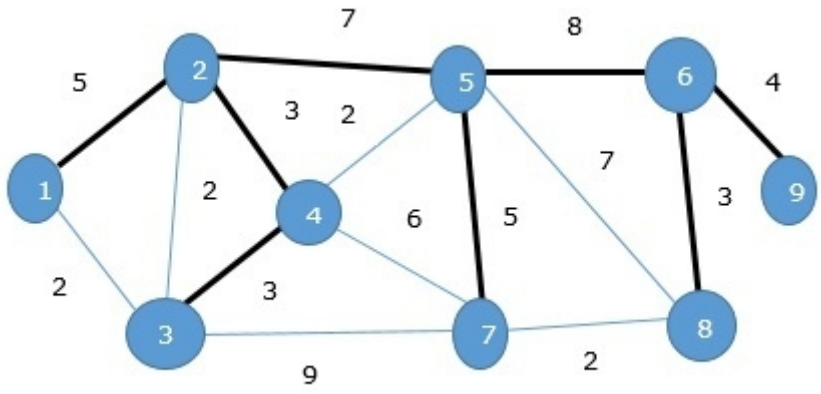


Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are **n** number of vertices, the spanning tree should have **n - 1** number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

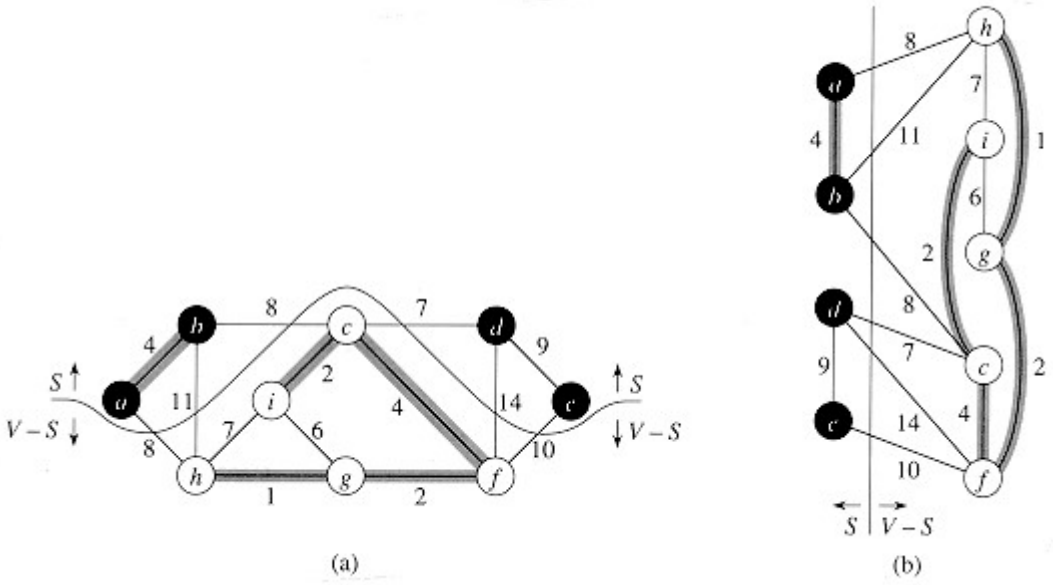
Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.



In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is (5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38.

6.Growing a minimum spanning tree

This greedy strategy is captured by the following "generic" algorithm, which grows the minimum spanning tree one edge at a time. The algorithm manages a set A that is always a subset of some minimum spanning tree. At each step, an edge (u, v) is determined that can be added to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a ***safe edge*** for A, since it can be safely added to A without destroying the invariant.




```
GENERIC-MST(G, w)
1  A ← ∅
2  while A does not form a spanning tree
3      do find an edge (u, v) that is safe for A
4         A ← A ∪ {(u, v)}
5  return A
```

The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section are elaborations of the generic algorithm. They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST. In Kruskal's algorithm, the set A is a forest. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a least-weighted edge connecting the tree to a vertex not in the tree.

Kruskal's Al gorithm

Kruskal's algorithm is based directly on the generic minimum-spanning-tree algorithm given in Section 24.1. It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight. Let C_1 and C_2 denote the two trees that are connected by (u, v). Since (u,v) must be a light edge connecting C_1 to some other tree, Corollary 24.2 implies that (u, v) is a safe

edge for C_1. Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Our implementation of Kruskal's algorithm is like the algorithm to compute connected components from Section 22.1. It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). The combining of trees is accomplished by the UNION procedure.

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3      do MAKE-SET (v)
4  sort the edges of E by nondecreasing weight w
5  for each edge (u, v) ∈ E, in order by nondecreasing weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION (u, v)
9  return A
```

Kruskal's algorithm works as shown in Figure 24.4. Lines 1-3 initialize the set A to the empty set and create $|V|$ trees, one containing each vertex. The edges in E are sorted into order by nondecreasing weight in line 4. The **for** loop in lines 5-8 checks, for each edge (u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A in line 7, and the vertices in the two trees are merged in line 8.

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint-set data structure. We shall assume the disjoint-set-forest implementation of Section 22.3 with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initialization takes time $O(V)$, and the time to sort the edges in line 4 is $O(E \lg E)$. There are $O(E)$ operations on the disjoint-set forest, which in total take $O(E \alpha(E, V))$ time, where α is the functional inverse of Ackermann's function defined in Section 22.4. Since $\alpha(E, V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$.

Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree algorithm from Section 24.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph. (See Section 25.2.) Prim's

algorithm has the property that the edges in the set A always form a single tree. As is illustrated in Figure 24.5, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. At each step, a light edge connecting a vertex in A to a vertex in $V - A$ is added to the tree. By Corollary 24.2, this rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy is "greedy" since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

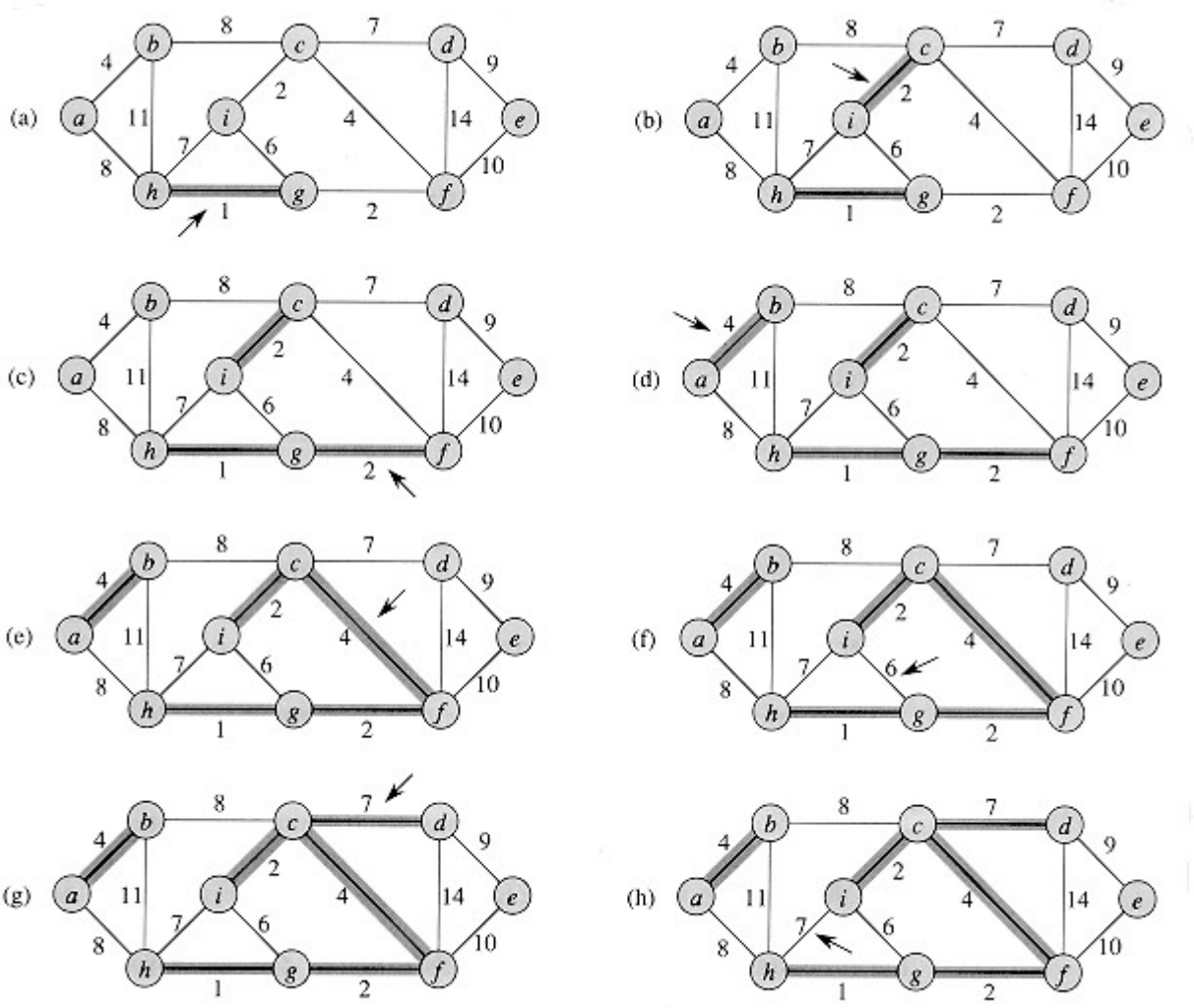
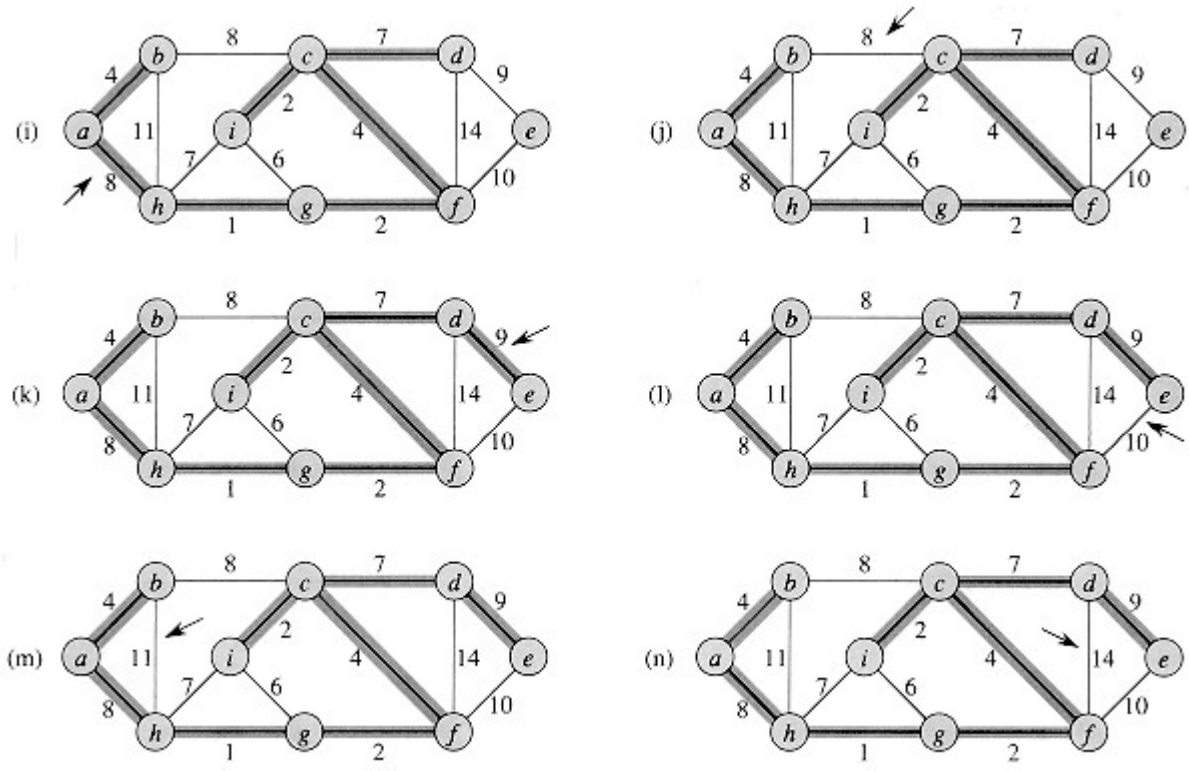


Figure 24.4 The execution of Kruskal's algorithm

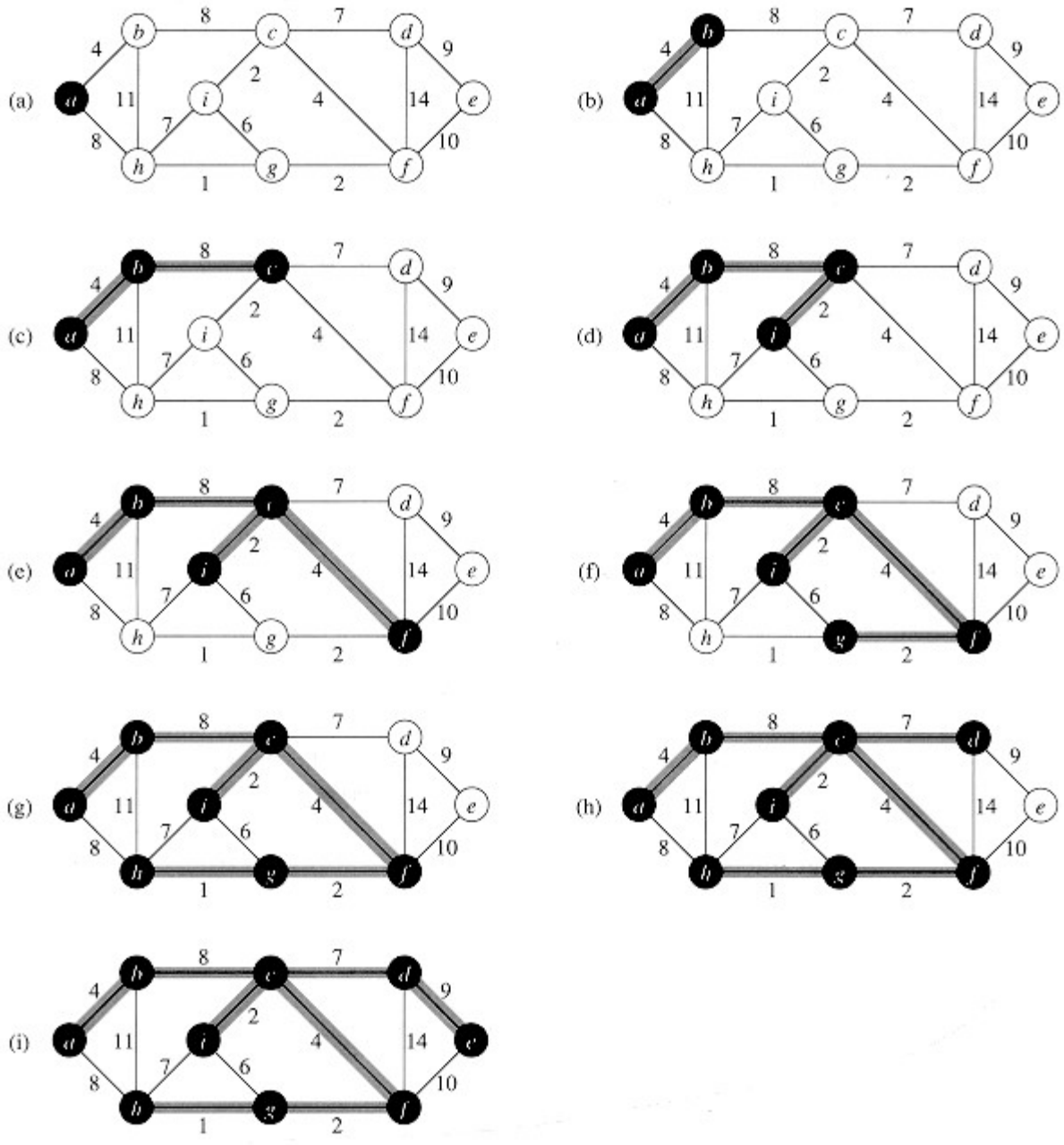


The key to implementing Prim's algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by the edges in A. In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are *not* in the tree reside in a priority queue Q based on a *key* field. For each vertex v, $key[v]$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $key[v] = \infty$ if there is no such edge. The field $\pi[v]$ names the "parent" of v in the tree. During the algorithm, the set A from GENERIC-MST is kept implicitly as

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} .$$

When the algorithm terminates, the priority queue Q is empty; the minimum spanning tree A for G is thus

$$A = \{(v, \pi[v]) : v \in V - \{r\}\} .$$




```
MST-PRIM(G, w, r)
1  Q ← V[G]
2  for each u ∈ Q
3       do key[u] ← ∞
4  key [r] ← 0
5  Π [r] ← NIL
6  while Q ≠ ∅
7       do u ← EXTRACT-MIN(Q)
8          for each v ∈ Adj[u]
```

```
9             do if v ∈ Q and w (u, v) < key[v]
10                  then Π[v] ← u
11                       key[v] ← w(u, v)
```

7.Single Source Shortest Path (SSSP) Problem

Given a directed graph G = (V,E), with non-negative costs on each edge, and a selected source node v in V, for all w in V, find the cost of the least cost path from v to w.

The *cost* of a path is simply the sum of the costs on the edges traversed by the path.

This problem is a general case of the more common subproblem, in which we seek the least cost path from v to a particular w in V. In the general case, this subproblem is no easier to solve than the SSSP problem.

Dijkstra's Algorithm

Dijkstra's algorithm is a *greedy* algorithm for the SSSP problem.

- A "greedy" algorithm always makes the locally optimal choice under the assumption that this will lead to an optimal solution overall.
- Example: in making change using the fewest number of coins, always start with the largest coin possible.

Data structures used by Dijkstra's algorithm include:

- a cost matrix C, where C[i,j] is the weight on the edge connecting node i to node j. If there is no such edge, C[i,j] = infinity.
- a set of nodes S, containing all the nodes whose shortest path from the source node is known. Initially, S contains only the source node.
- a distance vector D, where D[i] contains the cost of the shortest path (so far) from the source node to node i, using only those nodes in S as intermediaries.

The Algorithm

```
DijkstraSSSP (int N, rmatrix &C, vertex v, rvector &D)
{
   set S;
```

```
    int i;
    vertex k, w;
    float cw;

    Insert(S,v);
    for (k = 0; k < N; k++) D[k] = C[v][k];
    for (i = 1; i < N; i++) {
        /* Find w in V-S st. D[w] is minimum */
        cw = INFINITY;
        for (k = 0; k < N; k++)
         if (! Member(S,k) && D[k] < cw) {
            cw = D[k];
            w = k;
         }
        Insert(S, w);
        /* Forall k not in S */
        for (k = 0; k < N; k++)
         if (! Member(S,k))
              /* Shorter path from v to k using w? */
            if (D[w] + C[w][k] < D[k])
              D[k] = D[w] + C[w][k];
    }
} /* DijkstraSSSP */
```

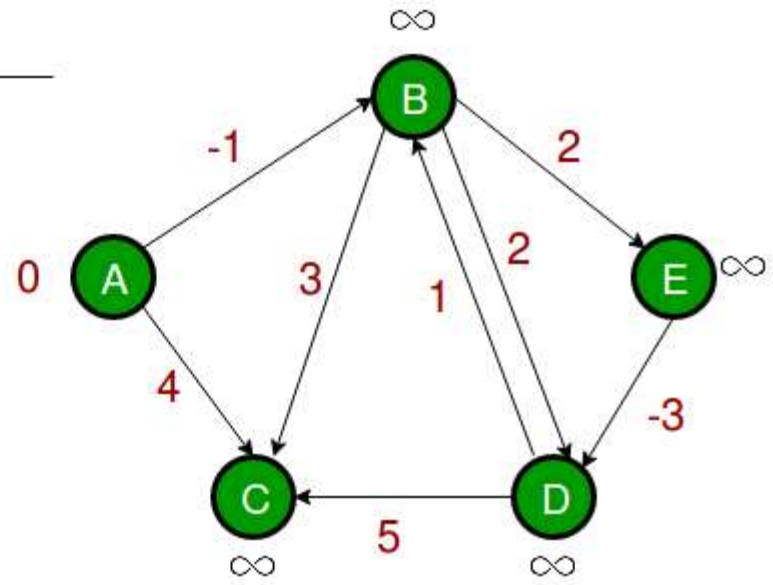
7. Bellman-Ford Algorithm

Given a graph and a source vertex src in the graph, find the shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and the time complexity is O((V+E)LogV) (with the use of the Fibonacci heap). Dijkstra doesn't work for Graphs with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(V * E), which is more than Dijkstra.

Below is the illustration of the above algorithm:

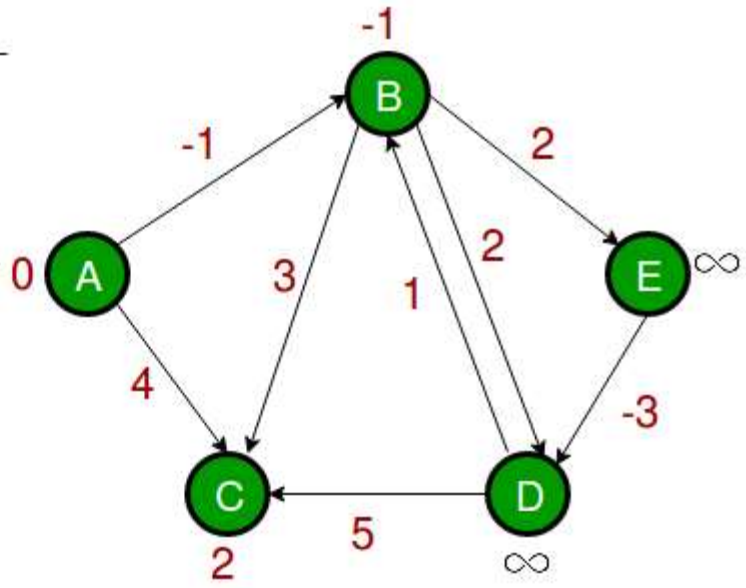
Step 1: *Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.*

A	B	C	D	E
0	∞	∞	∞	∞



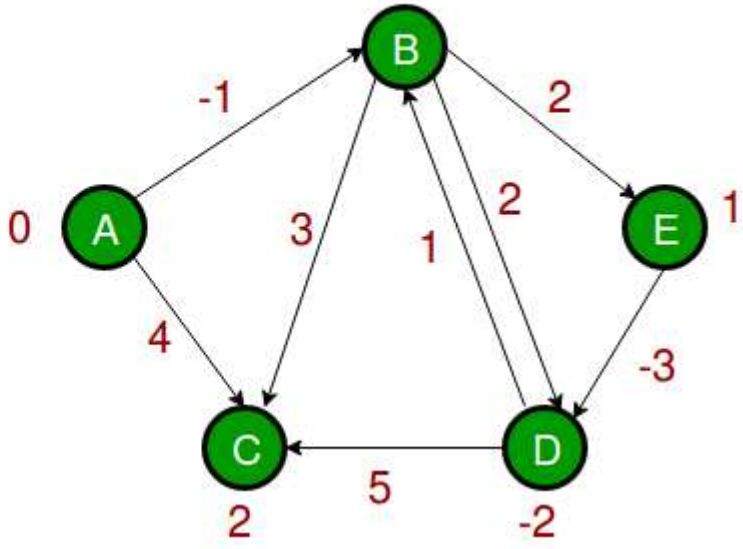
Step 2: *Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.*

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



Step 3: *The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).*

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



Step 4: *The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The*

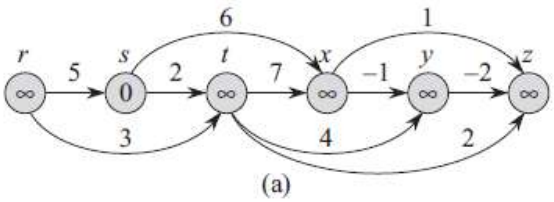
distances are minimized after the second iteration, so third and fourth iterations don't update the distances.
Below i

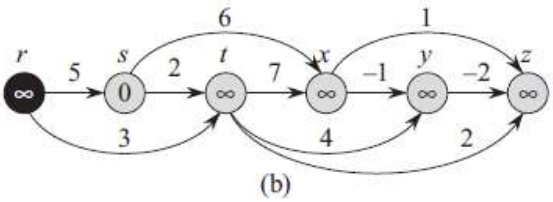
8.Single-source shortest paths in directed acyclic graphs

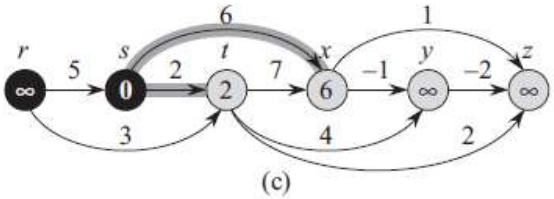
By relaxing the edges of a weighted dag (directed acyclic graph) $G = (V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $2(V + E)$ time. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist. The algorithm starts by topologically sorting the dag (see Section 22.4) to impose a linear ordering on the vertices. If there is a path from vertex u to vertex v, then u precedes v in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process each vertex, we relax each edge that leaves the vertex.

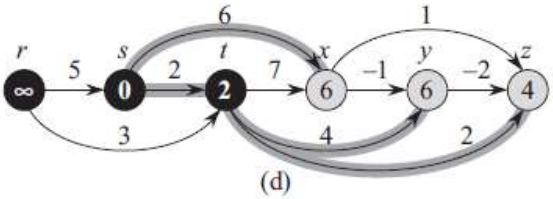
DAG-SHORTEST-PATHS(G,w, s)
1 topologically sort the vertices of G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **for** each vertex u, taken in topologically sorted order
4 **do for** each vertex v □ $Adj[u]$
5 **do** RELAX(u, v,w)

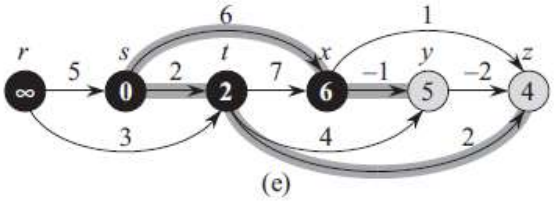
Figure 24.5 shows the execution of this algorithm.
The running time of this algorithm is easy to analyze. As shown in Section 22.4, the topological sort of line 1 can be performed in $2(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $2(V)$ time. There is one iteration per vertex in the **for** loop of lines 3–5. For each vertex, the edges that leave the vertex are each examined exactly once. Thus, there are a total of $|E|$ iterations of the inner **for** loop of lines 4–5. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes 2(1) time, the total running time is $2(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

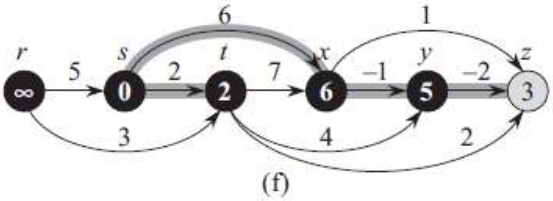

r s t x y z
∞ 0 ∞ ∞ ∞ ∞
5 6 2 7 −1 1 −2
3 4 2
(a)

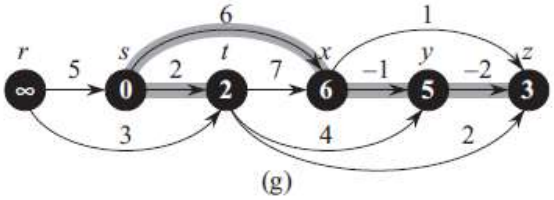

r s t x y z
∞ 0 ∞ ∞ ∞ ∞
5 6 2 7 −1 1 −2
3 4 2
(b)


r s t x y z
∞ 0 2 6 ∞ ∞
5 6 2 7 −1 1 −2
3 4 2
(c)


r s t x y z
∞ 0 2 6 6 4
5 6 2 7 −1 1 −2
3 4 2
(d)


r s t x y z
∞ 0 2 6 5 4
5 6 2 7 −1 1 −2
3 4 2
(e)


r s t x y z
∞ 0 2 6 5 3
5 6 2 7 −1 1 −2
3 4 2
(f)


r s t x y z
∞ 0 2 6 5 3
5 6 2 7 −1 1 −2
3 4 2
(g)