Graphs

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of edges which connect a pair of nodes.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.

In this chapter we will discuss some basic graph terminology of graphs and then define two fundamental representations for graphs, the adjacency matrix and adjacency list. Then we will discuss algorithms for finding the minimum-cost spanning tree, useful for determining lowest-cost connectivity in a network. We will end this chapter by presenting the two most commonly used graph traversal algorithms, called depth-first and breadth-first search, with their application.

Basic Terminology of Graphs

A graph G = (V,E) consists of a set of vertices V and a set of edges E, such that each edge in E is a connection between a pair of vertices in V. The number of vertices is written |V|, and the number of edges is written |E|.

Undirected Graph: A graph whose edges are unordered pairs of vertices. That is, each edge connects two vertices where edge (u, v) = edge (v, u).

Directed Graph: A graph whose edges are ordered pairs of vertices. That is, each edge can be followed from one vertex to another vertex where edge (u, v) goes from vertex u to vertex v.

Acyclic G (a) A graph with no path that starts ar ; at the same vertex.

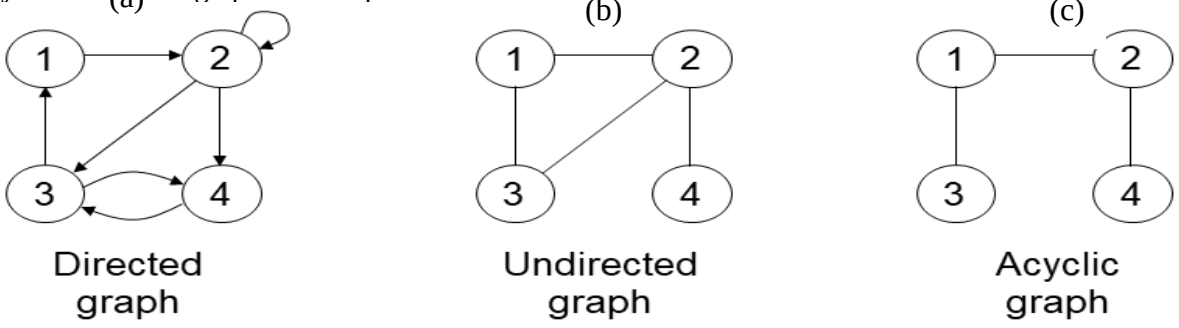


Figure 9.1: An example of (a) directed, (b) undirected and (c) acyclic graph

Another example of undirected graph G=(V,E) in figure 9.2, where V={1,2,3,4,5,6}andE={{1,2},{1,5},{2,5},{3,6}} and the vertex 4 is isolated.

Vertex 1,2,5 has degree 2; 3,6 has degree 1; vertex 4 has degree 0. Vertex 3 is adjacent to vertex 6 and vice versa; {1, 5} is adjacent to 2; 4 is not adjacent to any other vertex.

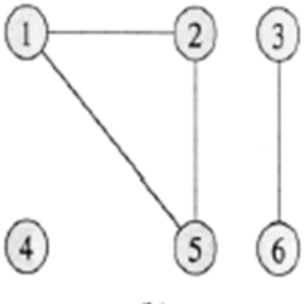


Figure 9.2: An example of undirected graph

Another example of **directed** graph G= (V, E)in figure 9.3, where V = {1, 2, 3, 4, 5, 6} and E={ (1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3) }. The edge (2,2) is **self-loop**. Vertex 5 has **in-degree** 2 and **out-degree** 1. Vertex 4 is **adjacent** to vertex 5; {1, 5} is **adjacent** to 4; 3 is not **adjacent** to any other vertex except 6.

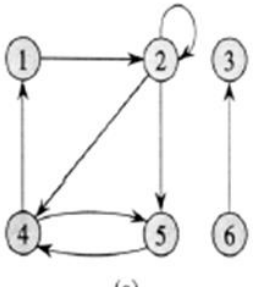


Figure 9.3: An example of directed graph

Complete graph: When every vertex is strictly connected to each other. (The number of edges in the graph is maximum).

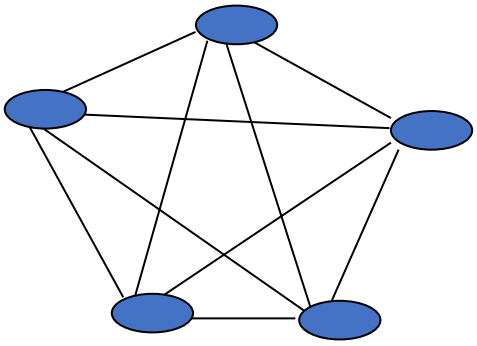


Figure 9.4: An example of Complete Graph

Dense graph: When the number of edges in the graph is close to maximum. (*adjacency matrix* is used to store info for this). A dense graph is one where there are many edges, but not necessarily as many as in a complete graph. This term is intentionally vague and is intended to convey a general sense that the number of edges can be expected to be large with respect to the number of vertices.

Sparse graph: When number of edges in the graph is very few. (*adjacency list* is used to store info for this)

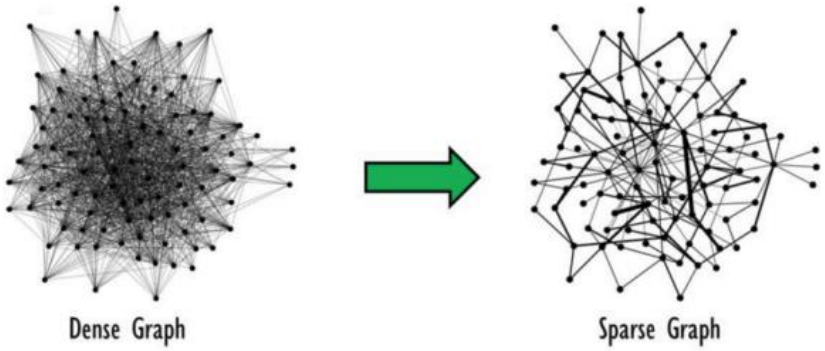


Figure 9.4: An example of dense and sparse graph

Weighted graph: associates weights with either the edges or the vertices
DAG: Directed acyclic graphs
Connected: if every vertex of a graph can *reach* every other vertex, i.e., every pair of vertices is connected by a path
Strongly connected: every 2 vertices are reachable from each other (in a digraph)
Connected Component: equivalence classes of vertices under "is reachable from" relation. Simply put, it is a subgraph in which any two vertices are **connected** to each other by paths, and which is **connected** to no additional vertices in the super graph.

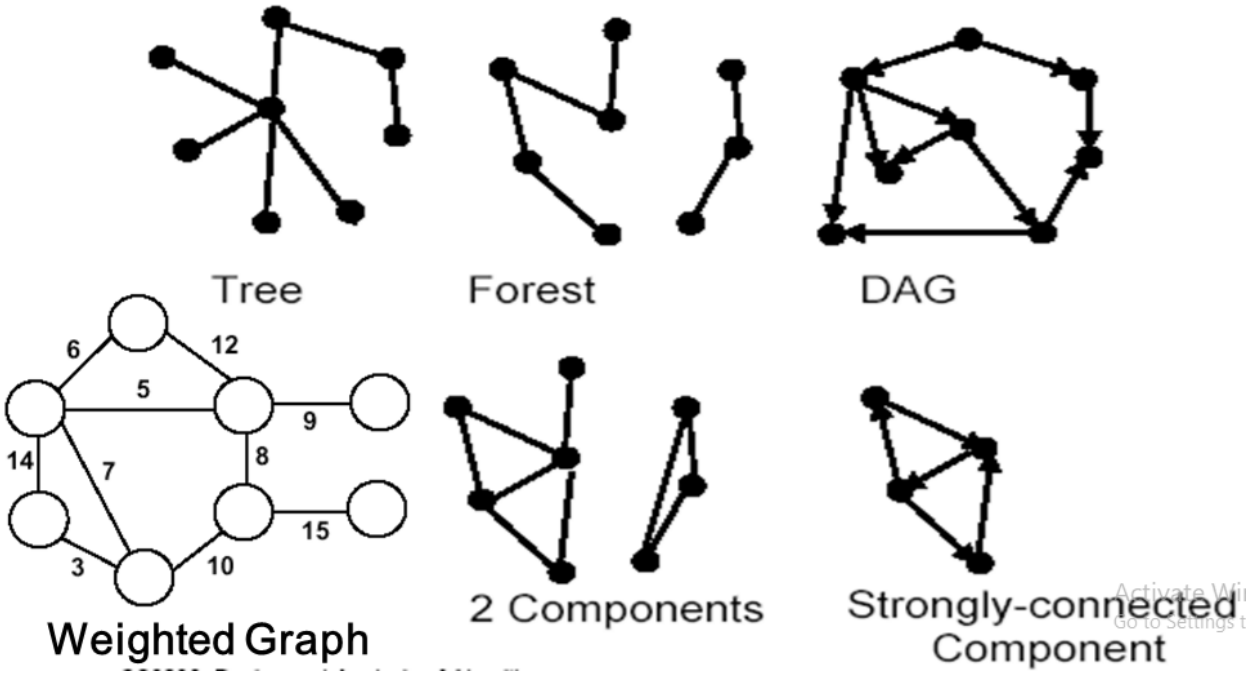


Figure 9.5: An example of tree, forest, DAG, weighted graph, components and strongly connected component

Degree of a vertex v: The degree of vertex v in a graph G, written d (v), is the number of edges incident to v, except that each loop at v counts twice (*in-degree* and *out-degree* for directed graphs). For example, in figure 9.6, d())=0, d(1)=1, d(2)=2, d(3)=3, d(4)= 3, d(5)=5, d(6)=2.

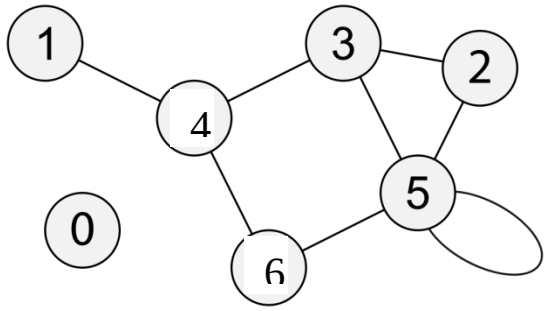


Figure 9.6: An example of undirected graph

1.1.1 Graph Applications

In **Computer science** graphs are used to represent the flow of computation. **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices. In **Facebook**, users are considered to be the vertices and if they are friends then there is

an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**. In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm. In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur. Here we outline just some of the many applications of graphs.

- State-space search in Artificial Intelligence
- Geographical information systems, electronic street directory[Figure 9.7]
- Logistics and supply chain management
- Telecommunications network design[Figure 9.8]
- Many more industry applications
- The graphic representation of world wide web (www)
- Resource allocation graph for processes that are active in the system.
- The graphic representation of a map[Figure 9.9]
- Scene graphs: The contents of a visual scene are also managed by using graph data structure.

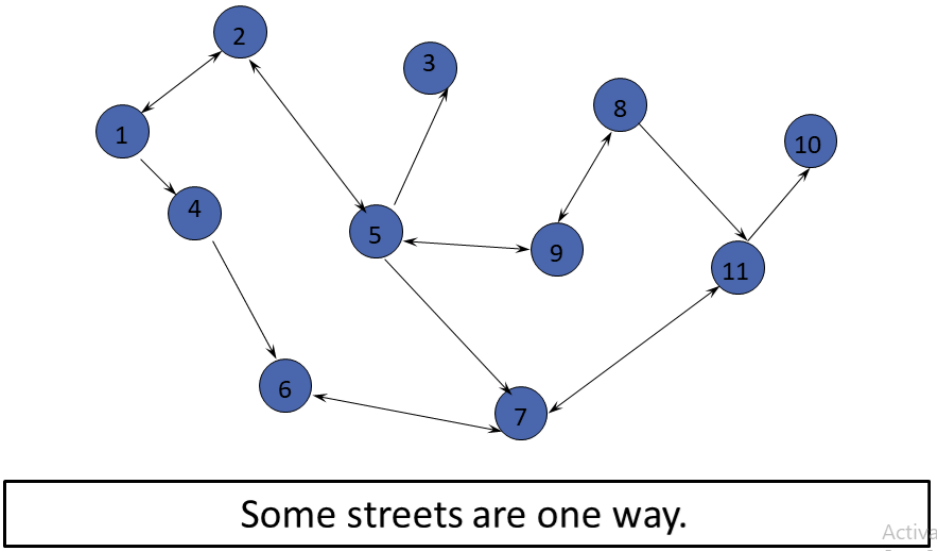


Figure 9.7: An example of directed graph which represents street map

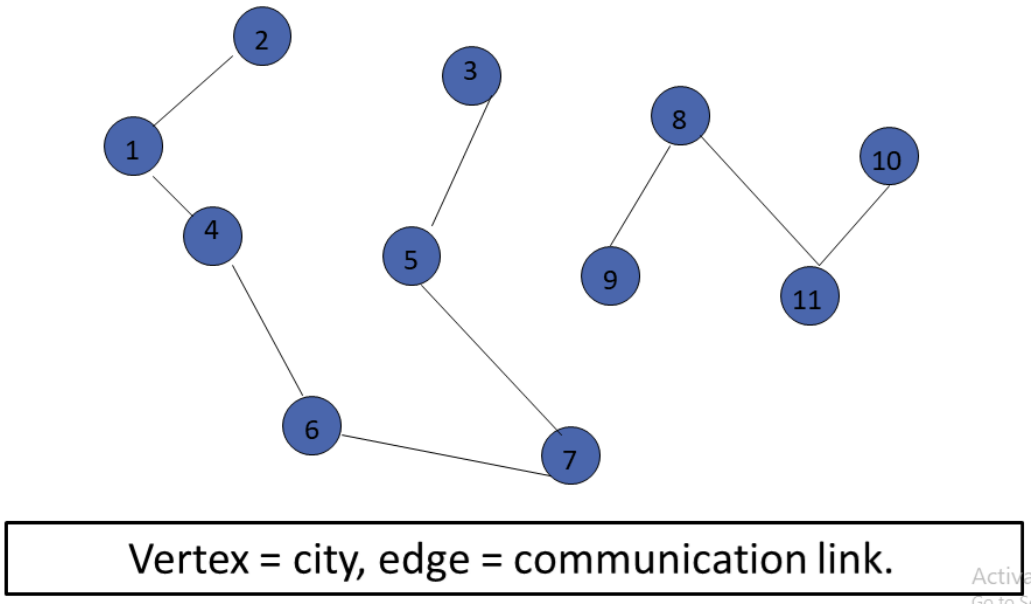


Figure 9.8: An example of undirected graph which represents Telecommunications network design

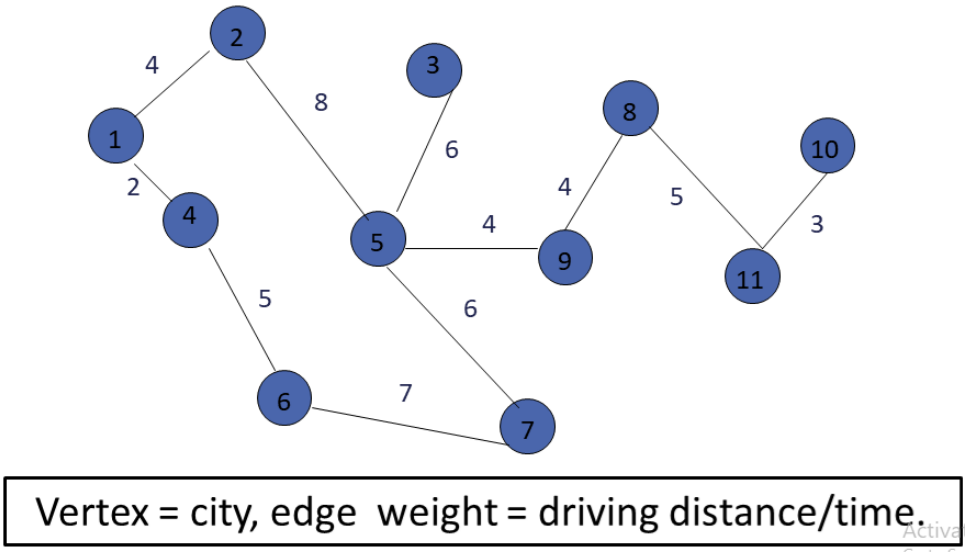


Figure 9.9: An example of undirected weighted graph which represents city map

1.1.2 Graph Representation

There are two commonly used methods for representing graphs. The adjacency matrix is illustrated by Figure 9.10(b). The adjacency matrix for a graph is a |V|x|V| array. Assume that |V| = n and that the vertices are labeled from V_0 through V_{n-1}. Row i of the adjacency matrix contains entries for Vertex V_i. Column j in row i is marked if there is an edge from V_i to V_j and is not marked otherwise. Thus, the adjacency matrix requires one bit at each position. Alternatively, if we wish to associate a number with each edge, such as the weight or distance between two vertices, then each matrix

position must store that number. In either case, the space requirements for the adjacency matrix are $\Theta(|V|^2)$.

The second common representation for graphs is the adjacency list, illustrated by Figure 9.10(c). The adjacency list is an array of linked lists. The array is |V| items long, with position i storing a pointer to the linked list of edges for Vertex V_i. This linked list represents the edges by the vertices that are adjacent to Vertex V_i.

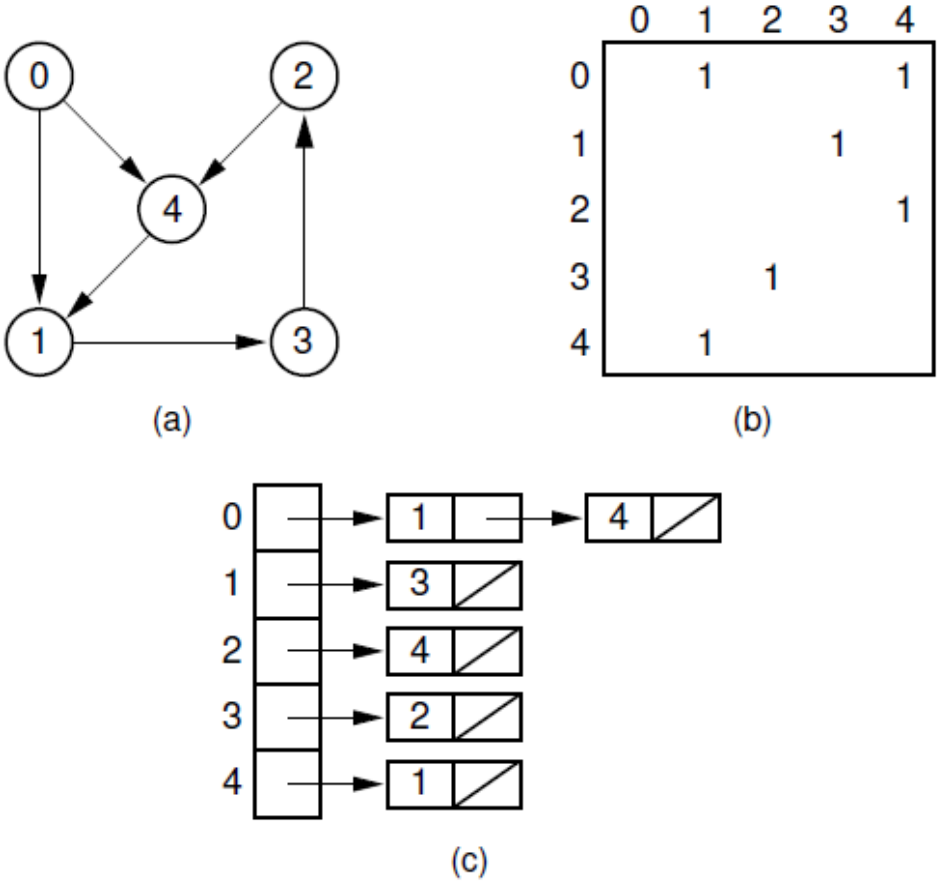


Figure 9.10 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

The storage requirements for the adjacency list depend on both the number of edges and the number of vertices in the graph. There must be an array entry for each vertex (even if the vertex is not adjacent to any other vertex and thus has no elements on its linked list), and each edge must appear on one of the lists. Thus, the cost is $\Theta(|V| + |E|)$.

Both the adjacency matrix and the adjacency list can be used to store directed or undirected graphs. Each edge of an undirected graph connecting Vertices U and V is represented by two directed edges: one from U to V and one from V to U. Figure 9.11 illustrates the use of the adjacency matrix and the adjacency list for undirected graphs.

Which graph representation is more space efficient depends on the number of edges in the graph. The adjacency list stores information only for those edges that actually appear in the graph, while the adjacency matrix requires space for each potential edge, whether it exists or not. However, the adjacency matrix requires no overhead for pointers, which can be a substantial cost, especially if the only information stored for an edge is one bit to indicate its existence. As the graph becomes denser, the adjacency matrix becomes relatively more space efficient. Sparse graphs are likely to have their adjacency list representation be more space efficient.

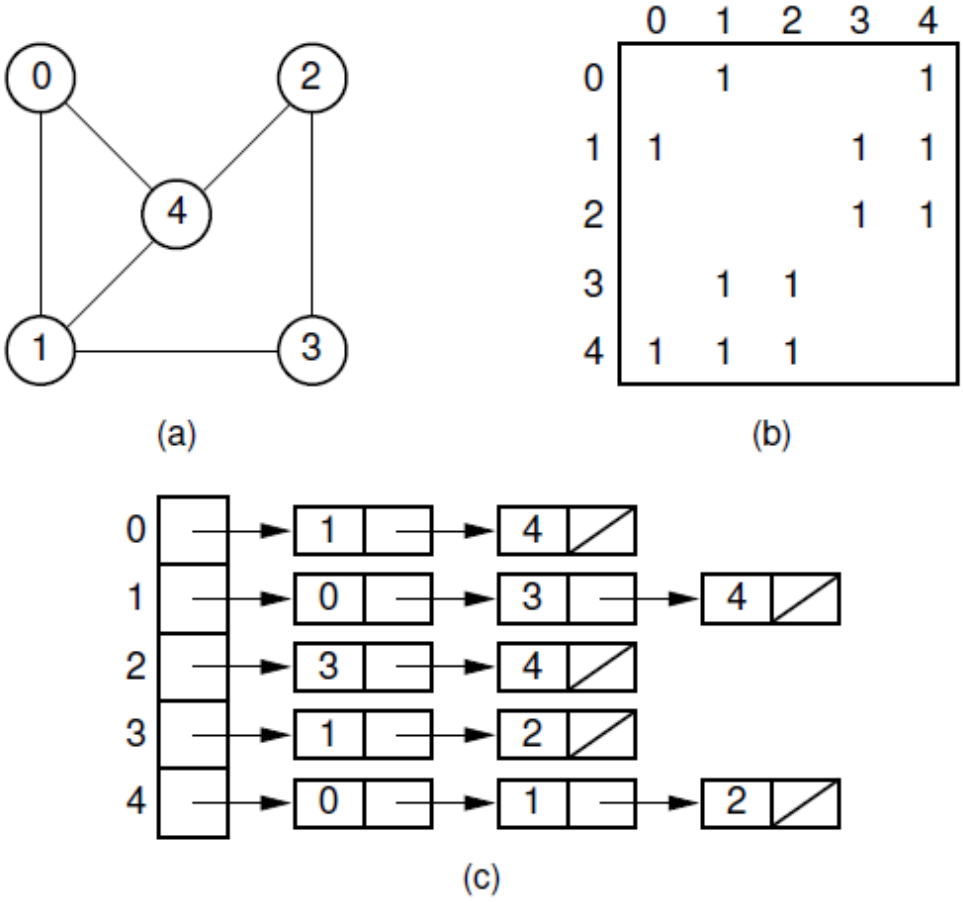


Figure 9.11 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list

1.2 Minimum Spanning Tree

The **minimum-cost spanning tree (MST)** problem takes as input a connected, undirected graph G, where each edge has a distance or weight measure attached. The MST is the graph containing the vertices of G along with the subset of G's edges that (1) has minimum total cost as measured by summing the values for all of the edges in the subset, and (2) keeps the vertices connected. Applications where a solution to this problem is useful include soldering the shortest set of wires needed to connect a set of terminals on a circuit board, and connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

The MST contains no cycles. If a proposed MST did have a cycle, a cheaper MST could be had by removing any one of the edges in the cycle. Thus, the MST is a free tree with |V|-1 edges. The name "minimum-cost spanning tree" comes from the fact that the required set of edges forms a tree, it spans the vertices (i.e., it connects them together), and it has minimum cost. Figure 9.12 shows the MST for an example graph.

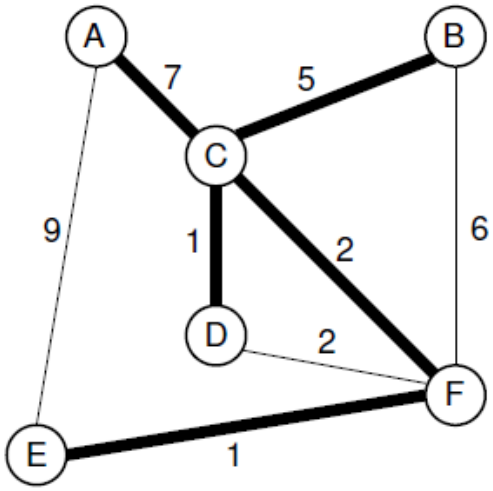


Figure 9.12 A graph and its MST. All edges appear in the original graph. Those edges drawn with heavy lines indicate the subset making up the MST. Note that edge (C, F) could be replaced with edge (D, F) to form a different MST with equal cost.

1.2.1 Applications of Minimum Spanning Tree

Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices. MST is fundamental problem with diverse applications.

1) **Network design (***telephone, electrical, hydraulic, TV cable, computer, road)*

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

2) **Approximation algorithms for NP-hard problems (** *traveling salesperson problem*, *Steiner tree*)

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.
Note that if you have a path visiting all points exactly once, it's a special kind of tree. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it's a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

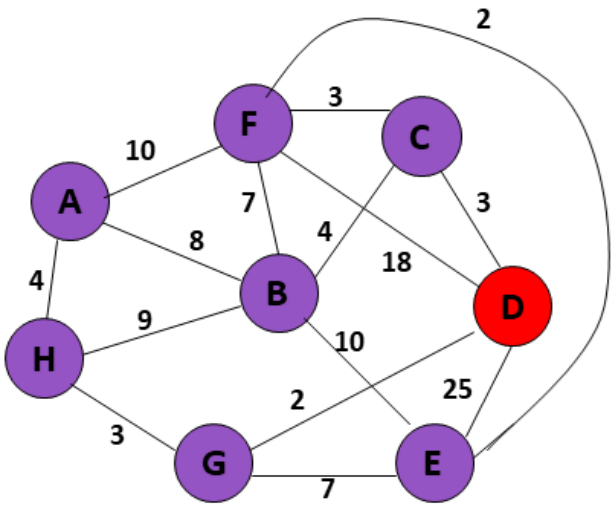
3) **Indirect applications.**
   – max bottleneck paths
   – LDPC codes for error correction
   – image registration with Renyi entropy
   – learning salient features for real-time face verification
   – reducing data storage in sequencing amino acids in a protein
   – model locality of particle interactions in turbulent fluid flows
   – auto configuration protocol for Ethernet bridging to avoid cycles in a network
4) **Cluster analysis**
   k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

1.2.2 Prim's Algorithm

The first of our two algorithms for finding MSTs is commonly referred to as Prim's algorithm. Prim's algorithm is very simple. Start with any Vertex N in the graph, setting the MST to be N initially. Pick the least-cost edge connected to N. This edge connects N to another vertex; call this M. Add Vertex M and Edge (N, M) to the MST. Next, pick the least-cost edge coming from either N or M to any other vertex in the graph. Add this edge and the new vertex it reaches to the MST. This process continues, at each step expanding the MST by selecting the least-cost edge from a vertex currently in the MST to a vertex not currently in the MST.

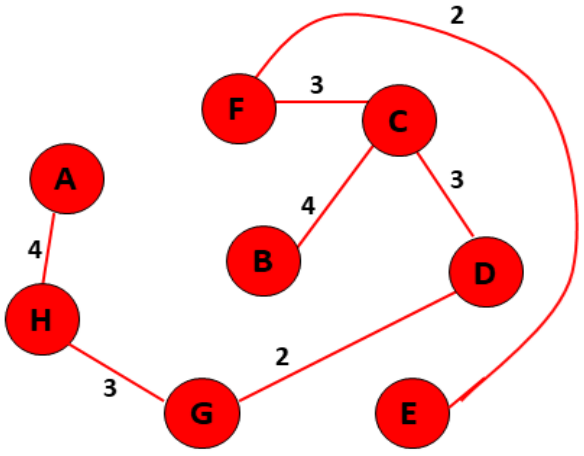
Pseudocode

```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes {
      remove an edge (v, w) of lowest cost from E
      if w is already in T then discard edge (v, w)
      else {
            add edge (v, w) and node w to T
            add to E the edges adjacent to w
      }
```



	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			

Figure 9.13: An example of undirected weighted graph and a table where K represents visit information, d_v represents distance and p_v represents parent for each vertex v.



	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done

Figure 9.14: Final MST with minimum cost 21.

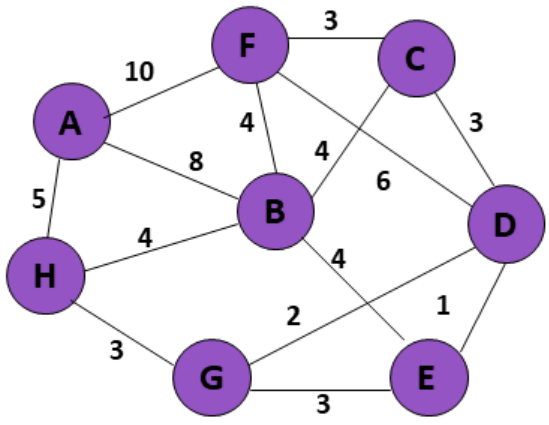
1.2.3 Kruskal's Algorithm

Our next MST algorithm is commonly referred to as Kruskal's algorithm. Kruskal's algorithm is also a simple, greedy algorithm. First partition the set of vertices into |V| equivalence classes, each consisting of one vertex. Then process the edges in order of weight. An edge is added to the MST, and two equivalence classes combined, if the edge connects two vertices in different equivalence classes. This process is repeated until only one equivalence class remains.

Pseudocode :

```
T = empty spanning tree;
E = set of edges;
N = number of nodes in graph;

while T has fewer than N - 1 edges {

     remove an edge (v, w) of lowest cost from E

     if adding (v, w) to T would create a cycle

          then discard (v, w)

          else add (v, w) to T

}
```

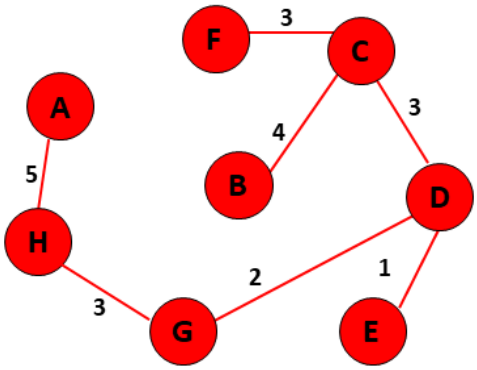
We can find an edge of lowest cost just by sorting the edges. Efficient testing for a cycle requires a fairly complex algorithm (UNION-FIND) which we don't cover in this course. Consider an undirected weighted graph in figure 9.15. The figure 9.16 shows the final MST after applying kruskal's algorithm.



edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Figure 9.15: An example of undirected weighted graph and a table for sorted list of edges.



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

Done

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Figure 9.16: Final MST with minimum cost 21.

1.3 Graph Traversal Algorithms

Often it is useful to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph traversal and is similar in concept to a tree traversal. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, in order, or post order. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an in order traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain may consist of a large collection of states, with connections between various pairs of states. Solving the problem may require getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it may not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph may contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

1.3.1 DFS

The first method of organized graph traversal is called depth-first search (DFS). The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows: Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop. The figure 9.17 shows an illustration of how DFS works.

```
depthFirstSearch(v)
{  Label vertex v as reached.
   for (each unreached vertex u adjacenct from v)
       depthFirstSearch(u);
}
```

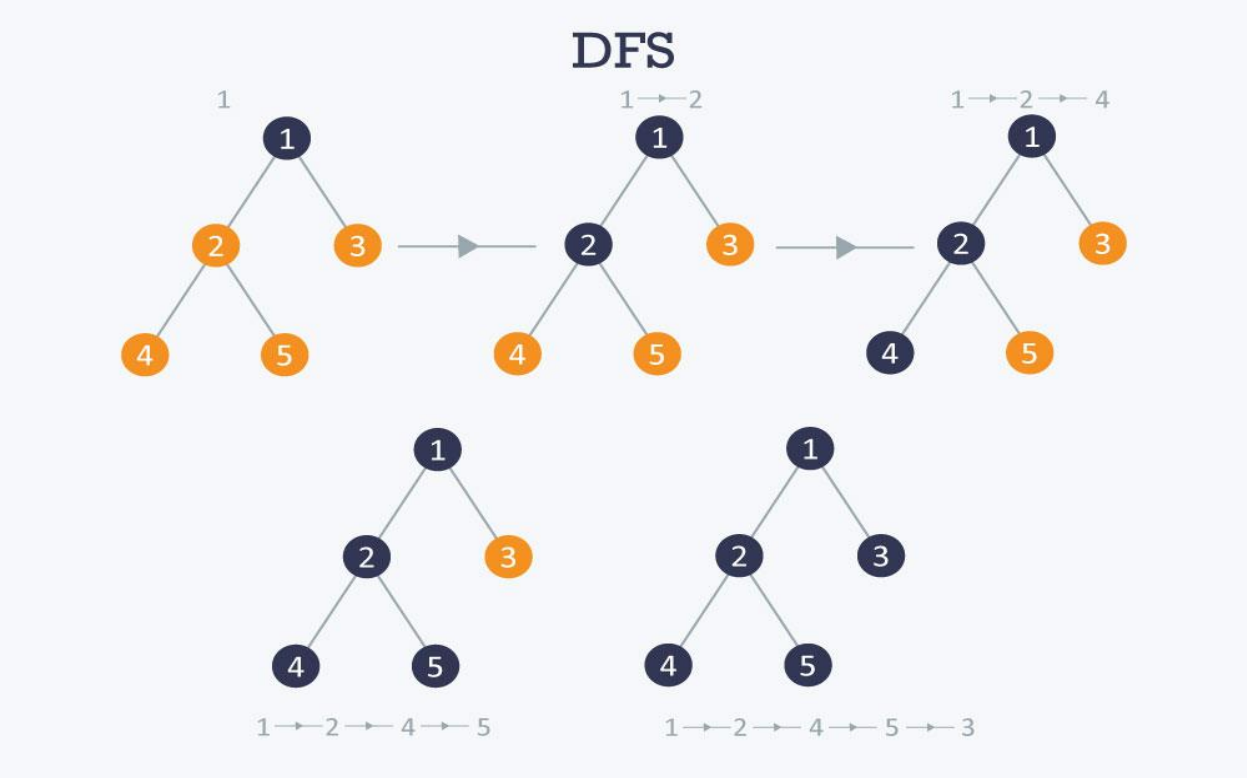


Figure 9.17: An illustration of DFS Algorithm

1.3.2 DFS: Classification of Edges

The traversal algorithm DFS can be used to classify edges of G. Following are the list of edges which can be found by using DFS algorithm.

- **Tree edges**: Edges in the depth-first forest.

- **Back edges**: Edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree (where v is not the parent of u). It also applies for self-loops.
- **Forward edges**: Non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- **Cross edges**: All other edges.

DFS yields valuable information about the structure of a graph. In DFS of an undirected graph we get only tree and back edges; no forward or back-edges. The graphs in figure 9.18 show the classification of edges.

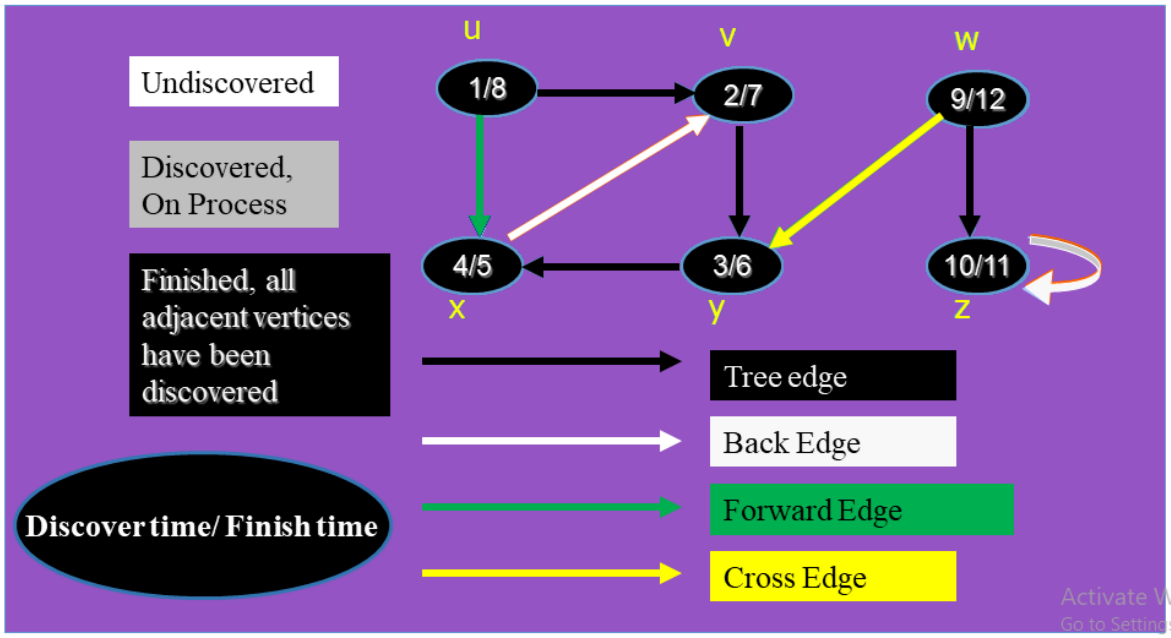


Figure 9.18: An illustration of classifying edge using DFS Algorithm

1.3.3 Applications of DFS

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph. Following are the problems that use DFS as a building block.

1) For a weighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) **Detecting cycle in a graph**
A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

3) **Path Finding**
We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

4) **Topological Sorting**

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when re computing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers.

5) **To test if a graph is bipartite**
We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

6) **Finding Strongly Connected Components of a graph**

A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algorithm for finding Strongly Connected Components)

7) **Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

1.3.4 BFS

Our second graph traversal algorithm is known as a breadth-first search (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom. Figure 9.19 shows a graph and the corresponding breadth-first search tree. Figure 9.20 illustrates the BFS process for the graph of Figure 9.19(a).

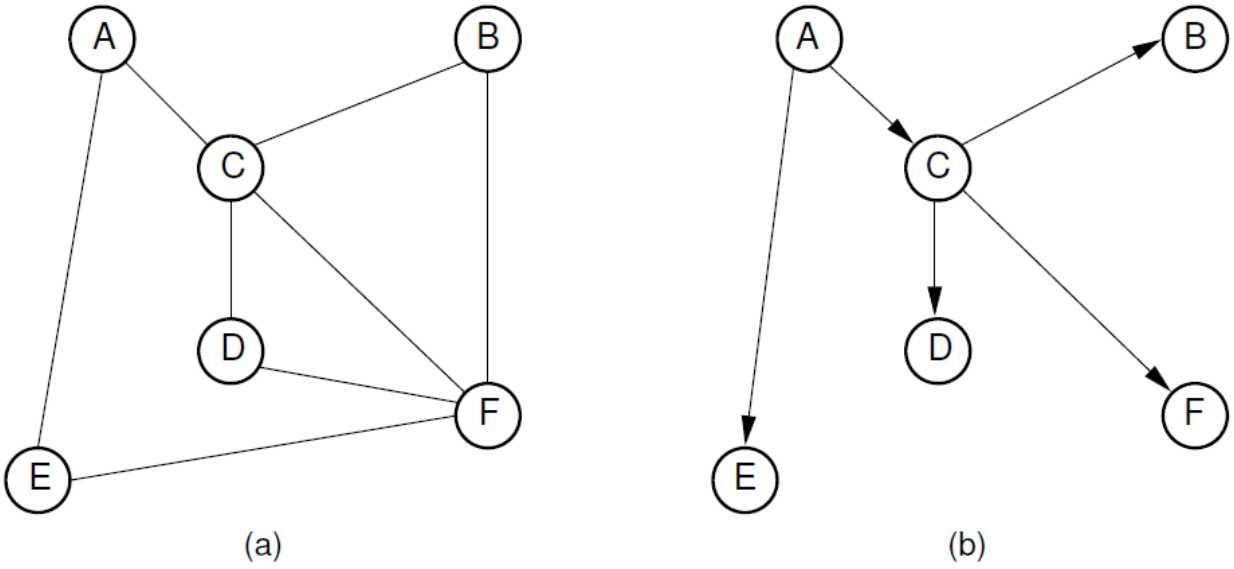


Figure 9.19 (a) A graph. (b) The breadth-first search tree for the graph when starting at Vertex *A*.

A	

Initial call to BFS on A.
Mark A and put on the queue.

C	E	

Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C)
Process (A, E).
Mark and enqueue E. Print(A, E).

E	B	D	F	

Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).

B	D	F	

Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.

D	F	

Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.

F	

Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.

Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

Figure 9.20 A detailed illustration of the BFS process for the graph of Figure 11.11(a) starting at Vertex *A*. The steps leading to each change in the queue are described.

1.3.5 Applications of BFS

We have earlier discussed Breadth First Traversal Algorithm for Graphs. We have also discussed applications of Depth First Traversal. Now we will discuss the applications of Breadth First Search.

1) **Shortest Path and Minimum Spanning Tree for unweighted graph**
   In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) **Peer to Peer Networks**
   In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) **Crawlers in Search Engines**
   Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

4) Social Networking Websites
   In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) **GPS Navigation systems**
   Breadth First Search is used to find all neighboring locations.

6) **Broadcasting in Network**
   In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection
   Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

8) Cycle detection in undirected graph
   In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use BFS to detect cycle in a directed graph also,

9) **Ford–Fulkerson algorithm**
   In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

10) **To test if a graph is Bipartite**

We can either use Breadth First or Depth First Traversal.

11) **Path Finding**
We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component
We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

1.4 Exercises

1) Draw the adjacency matrix representation for the graph of Figure 9.21.
2) Draw the adjacency list representation for the same graph.
3) Show the DFS tree for the graph of Figure 9.21, starting at Vertex 1.
4) Show the BFS tree for the graph of Figure 9.21, starting at Vertex 1.
5) Does either Prim's or Kruskal's algorithm work if there are negative edge weights?
6) List the order in which the edges of the graph in Figure 9.21 are visited when running Prim's MST algorithm starting at Vertex 3. Show the final MST.
7) List the order in which the edges of the graph in Figure 9.21 are visited when running Kruskal's MST algorithm. Each time an edge is added to the MST, show the result on the equivalence array. Show the final MST.

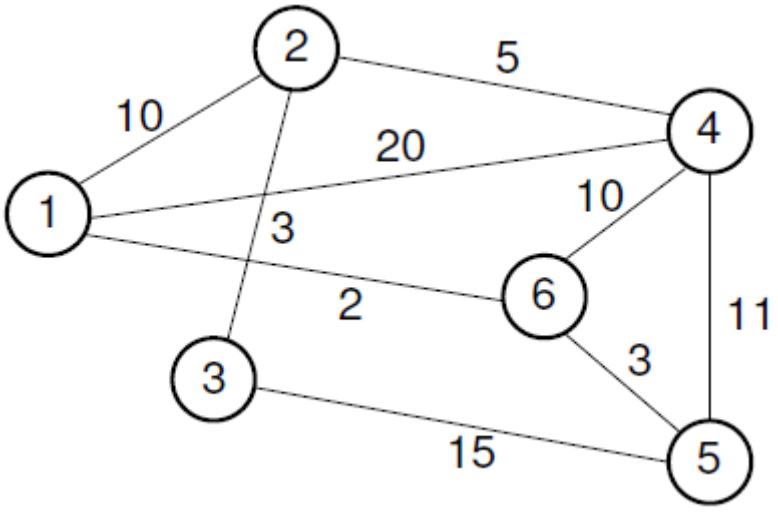


Figure 9.21: Example graph for Chapter exercises.

1.5 References

[1] https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

[2] https://www.geeksforgeeks.org/applications-of-depth-first-search/

[3] https://en.wikipedia.org/wiki/Data_structure

[4] https://visualgo.net/en/dfsbfs?slide=1

[5] "Schaum's Outline of Data Structures with C++". By John R. Hubbard (Can be found in university Library)
[6] "Data Structures and Program Design", Robert L. Kruse, 3rd Edition, 1996.
[7] "Data structures, algorithms and performance", D. Wood, Addison-Wesley, 1993
[8] "Advanced Data Structures", Peter Brass, Cambridge University Press, 2008
[9] "Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), Clifford A. Shaffer, Virginia Tech, Blacksburg, VA 24061 January 2, 2012
[10] "C++ Data Structures", Nell Dale and David Teague, Jones and Bartlett Publishers, 2001.
[11] "Data Structures and Algorithms with Object-Oriented Design Patterns in C++", Bruno R. Preiss
[12] https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/