Fundamental Data Structures and Algorithms 06 - Graphs

Unit 3: Basic Data Structures (continued)

3.7 Graph

A graph is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called vertices, together with a collection of pairwise connections between them, called edges. Graphs have applications in modeling many domains, including mapping, transportation, computer networks, and electrical engineering. By the way, this notion of a "graph" should not be confused with bar charts and function plots, as these kinds of "graphs" are unrelated to the topic of this chapter.

3.7.1 Basic Graph Terminologies

Viewed abstractly, a ***graph*** G is simply a set V of ***vertices*** and a collection E of pairs of vertices from V, called ***edges***. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V. Incidentally, some books use different terminology for graphs and refer to what we call vertices as ***nodes*** and what we call edges as ***arcs***. We use the terms "vertices" and "edges."

Edges in a graph are either ***directed*** or ***undirected***. An edge (u, v) is said to be ***directed*** from u to v if the pair (u, v) is ordered, with u preceding v. An edge (u, v) is said to be ***undirected*** if the pair (u, v) is not ordered. Undirected edges are sometimes denoted with set notation, as u, v, but for simplicity we use the pair notation (u, v), noting that in the undirected case (u, v) is the same as (v, u). Graphs are typically visualized by drawing the vertices as ovals or rectangles and the edges as segments or curves connecting pairs of ovals and rectangles. The following are some examples of directed and undirected graphs:

- We can visualize collaborations among the researchers of a certain discipline by constructing a graph whose vertices are associated with the researchers themselves, and whose edges connect pairs of vertices associated with researchers who have coauthored a paper or book. Such edges are undirected because coauthorship is a ***symmetric*** relation; that is, if *A* has coauthored something with *B*, then *B* necessarily has coauthored something with *A*.
- We can associate with an object-oriented program a graph whose vertices represent the classes defined in the program, and whose edges indicate inheritance between classes. There is an edge from a vertex v to a vertex u if the class for v inherits from the class for u. Such edges are directed because the inheritance relation only goes in one direction (that is, it is ***asymmetric***).

If all the edges in a graph are undirected, then we say the graph is an ***undirected graph***. Likewise, a ***directed graph***, also called a ***digraph***, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a ***mixed graph***. Note that an undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u,v) by the pair of directed edges (u,v) and (v,u). It is often useful, however, to keep undirected and mixed graphs represented as they are, for such graphs have several applications, as in the following examples:

- A city map can be modeled as a graph whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections. This graph has both undirected edges, which correspond to stretches of two-way streets, and directed edges, which correspond to stretches of one-way streets. Thus, in this way, a graph modeling a city map is a mixed graph.
- Physical examples of graphs are present in the electrical wiring and plumbing networks of a building. Such networks can be modeled as graphs, where each connector, fixture, or outlet is viewed as a vertex, and each uninterrupted stretch of wire or pipe is viewed as an edge. Such graphs are actually components of much larger graphs, namely the local power and water distribution networks. Depending on the specific aspects of these graphs that we are interested in, we may consider their edges as undirected or directed, for, in principle, water can flow in a pipe and current can flow in a wire in either direction.

The two vertices joined by an edge are called the ***end vertices*** (or ***endpoints***) of the edge. If an edge is directed, its first endpoint is its ***origin*** and the other is the ***destination*** of the edge. Two vertices u and v are said to be ***adjacent*** if there is an edge whose end vertices are u and v. An edge is said to be ***incident*** to a vertex if the vertex is one of the edge's endpoints. The ***outgoing edges*** of a vertex are the directed edges whose origin is that vertex. The ***incoming edges*** of a vertex are the directed edges whose destination is that vertex. The ***degree*** of a vertex v, denoted $deg(v)$, is the number of incident edges of v. The ***in-degree*** and ***out-degree*** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted $indeg(v)$ and $outdeg(v)$, respectively.

- We can study air transportation by constructing a graph G, called a flight network, whose vertices are associated with airports, and whose edges are associated with flights (See Fig. 3.20). In graph G, the edges are directed because a given flight has a specific travel direction. The endpoints of an edge e in G correspond respectively to the origin and destination of the flight corresponding to e. Two airports are adjacent in G if there is a flight that flies between them, and an edge e is incident to a vertex v in G if the flight for e flies to or from the airport for v. The outgoing edges of a vertex v correspond to the outbound flights from v's airport, and the incoming edges correspond to the inbound flights to v's airport. Finally, the in-degree of a vertex v of G corresponds to the number of inbound flights to v's airport, and the out-degree of a vertex v in G corresponds to the number of outbound flights.

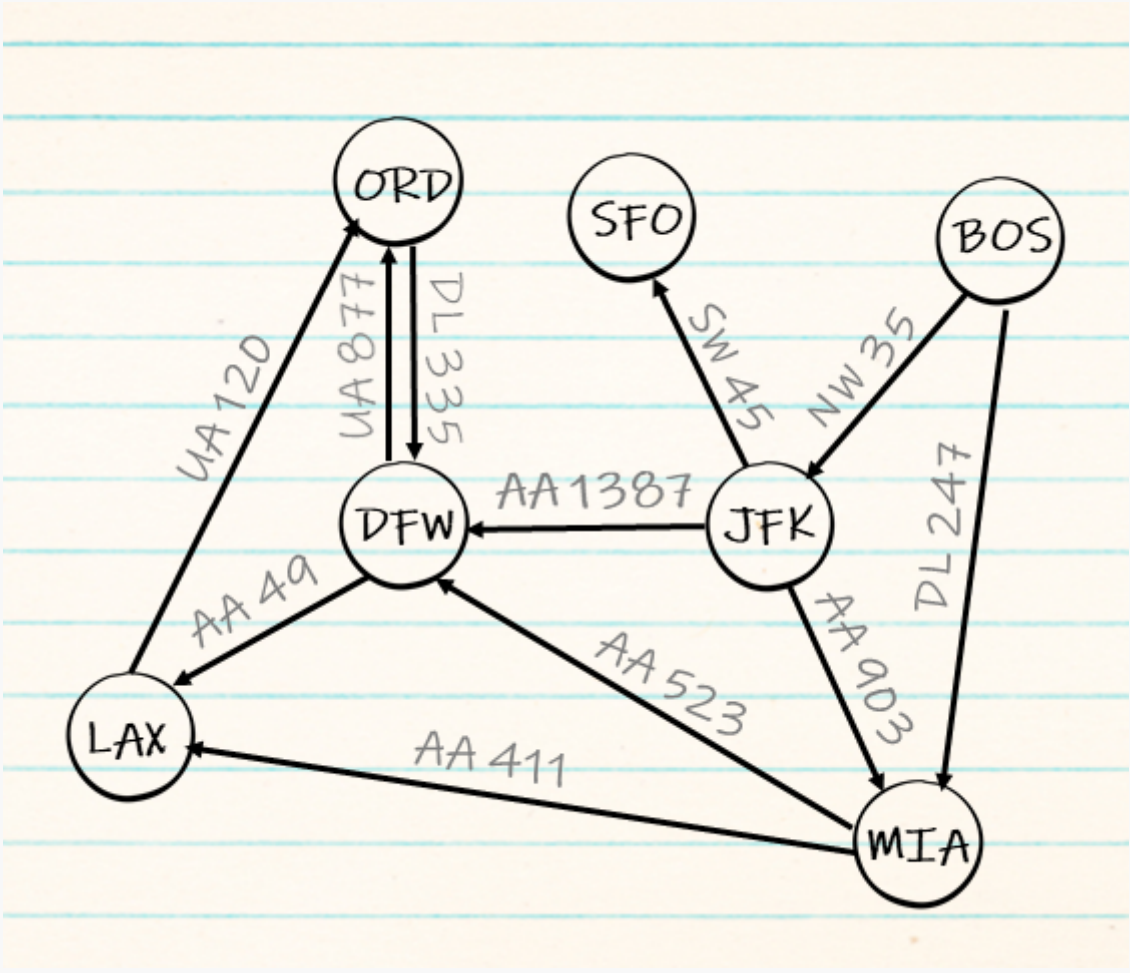

Fig 3.20. Example of a directed graph representing a flight network.

The definition of a graph refers to the group of edges as a ***collection***, not a ***set***, thus allowing two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called ***parallel edges*** or ***multiple edges***. A flight network can contain parallel edges, such that multiple edges between the same pair of vertices could indicate different flights operating on the same route at different times of the day. Another special type of edge is one that connects a vertex to itself. Namely, we say that an edge (undirected or directed) is a self-loop if its two endpoints coincide. A ***self-loop*** may occur in a graph associated with a city map, where it would correspond to a "circle" (a curving street that returns to its starting point).

With few exceptions, graphs do not have parallel edges or self-loops. Such graphs are said to be ***simple***. Thus, we can usually say that the edges of a simple graph are a ***set*** of vertex pairs (and not just a collection). Throughout this chapter, we assume that a graph is simple unless otherwise specified.

A ***path*** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex. A ***cycle*** is a path that starts and ends at the same vertex, and that includes at least one edge. We say that a path is ***simple*** if each vertex in the path is distinct, and we say that a cycle is ***simple*** if each vertex in the cycle is distinct, except for the first and last one. A d***irected path*** is a path such that all edges are directed and are traversed along their direction. A ***directed cycle*** is similarly defined. For example, in Fig 3.20, (BOS, NW35, JFK, AA 1387, DFW) is a directed simple path, and (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) is a directed simple cycle. Note that a directed graph may have a cycle consisting of two edges with opposite direction between the same pair of vertices, for example (ORD, UA 877, DFW, DL 335, ORD) in Fig 3.20. A directed graph is ***acyclic*** if it has no directed cycles. For example, if we were to remove the edge UA 877 from the graph in Fig 3.20, the remaining graph is acyclic. If a graph is simple, we may omit the edges when describing path P or cycle C, as these are well defined, in which case P is a list of adjacent vertices and C is a cycle of adjacent vertices.

Given a graph G representing a city map, we can model a couple driving to dinner at a recommended restaurant as traversing a path though G. If they know the way, and do not accidentally go through the same intersection twice, then they traverse a simple path in G. Likewise, we can model the entire trip the couple takes, from their home to the restaurant and back, as a cycle. If they go home from the restaurant in a completely different way than how they went, not even going through the same intersection twice, then their entire round trip is a simple cycle. Finally, if they travel along one-way streets for their entire trip, we can model their night out as a directed cycle.

Given vertices u and v of a (directed) graph G, we say that u reaches v, and that v is reachable from u, if G has a (directed) path from u to v. In an undirected graph, the notion of reachability is symmetric, that is to say, u reaches v if an only if v reaches u. However, in a directed graph, it is possible that u reaches v but v does not reach u, because a directed path must be traversed according to the respective directions of the edges. A graph is connected if, for any two vertices, there is a path between them. A directed graph G is strongly connected if for any two vertices u and v of G, u reaches v and v reaches u. See Fig 3.21 for some examples.

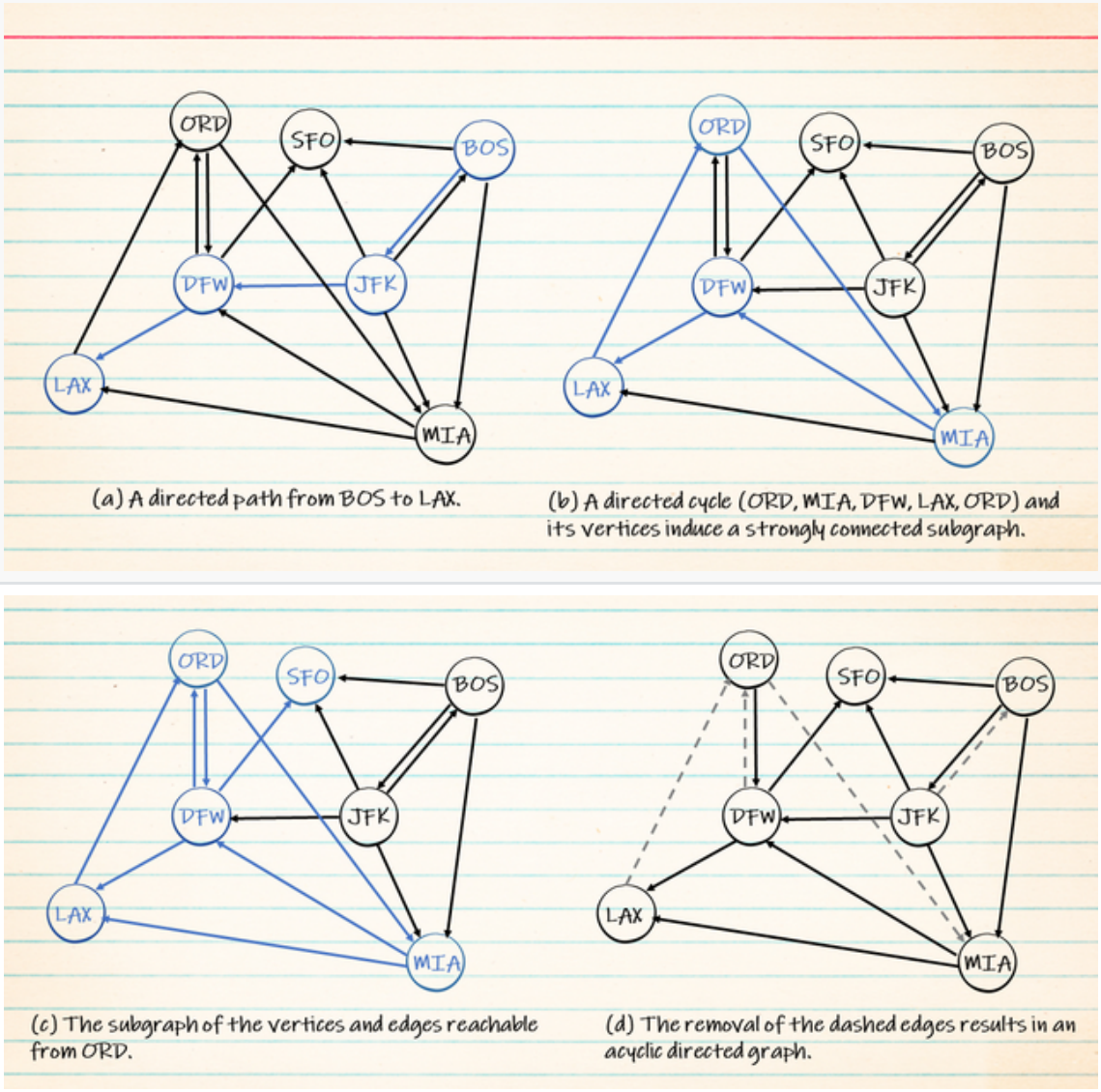

Fig 3.21. Examples of reachability in a directed graph.

A ***subgraph*** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively. A ***spanning subgraph*** of G is a subgraph of G that contains all the vertices of the graph G. If a graph G is not connected, its maximal connected subgraphs are called the connected components of G. A forest is a graph without cycles. A tree is a connected forest, that is, a connected graph without cycles. A spanning tree of a graph is a spanning subgraph that is a tree. (Note that this definition of a tree is somewhat different from the one given in Chapter 3.6, as there is not necessarily a designated root.)

> Perhaps the most talked about graph today is the Internet, which can be viewed as a graph whose vertices are computers and whose (undirected) edges are communication connections between pairs of computers on the Internet. The computers and the connections between them in a single domain, like wiley.com, form a subgraph of the Internet. If this subgraph is connected, then two users on computers in this domain can send email to one another without having their information packets ever leave their domain. Suppose the edges of this subgraph form a spanning tree. This implies that, if even a single connection goes down (for example, because someone pulls a communication cable out of the back of a computer in this domain), then this subgraph will no longer be connected.

3.7.2 Graph Representation

Graphs can be represented in two main forms. One way is to use an ***adjacency list*** and the other is to use an ***adjacency matrix***.

We shall be working with the following figure to develop both types of representation for the following graph:

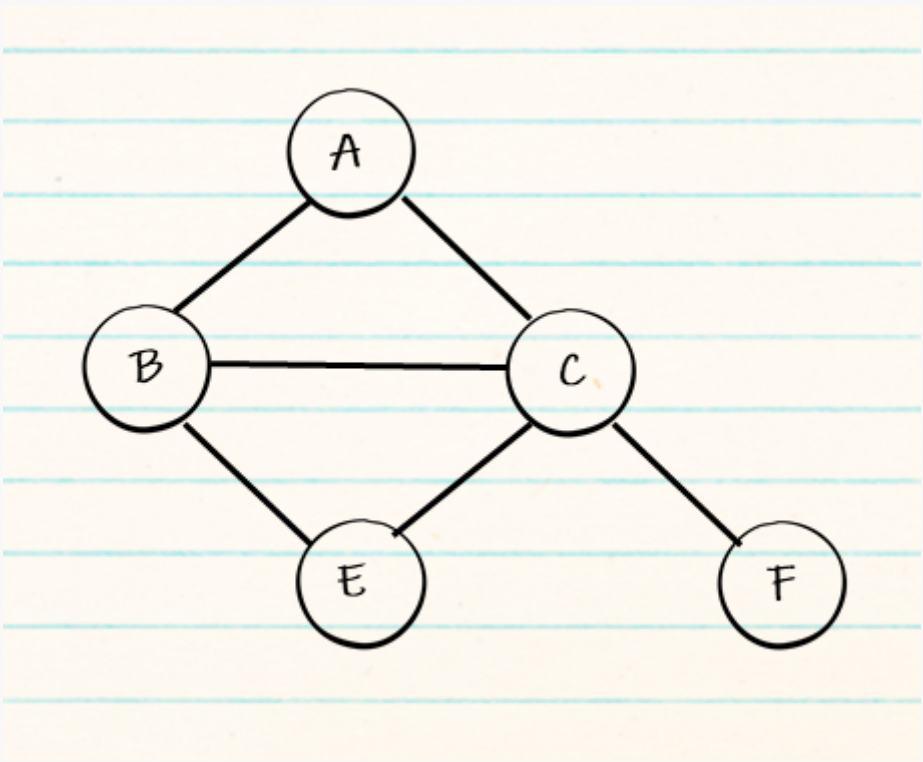

Fig 3.22. Sample graph for representation.

3.7.2.1 Adjacency list

A simple list can be used to present a graph. The indices of the list will represent the nodes or vertices in the graph. At each index, the adjacent nodes to that vertex can be stored.

Index	Vertex	Adjacent Vertices
0	A	B,C
1	B	E, A
2	C	A, B, E, F
3	E	B, C
4	F	C

As shown in the table above, index 0 represents vertex A, with its adjacent vertices being B and C.

Using a list for the representation is quite restrictive because we lack the ability to directly use the vertex labels. A dictionary is therefore more suited. To represent the graph in the diagram, we can use the following statements:

```
graph = dict()
graph['A'] = ['B', 'C']
graph['B'] = ['E','A']
graph['C'] = ['A', 'B', 'E','F']
graph['E'] = ['B', 'C']
graph['F'] = ['C']
```

Now we easy establish that vertex A has the adjacent vertices B and C. Vertex F has vertex C as its only neighbor.

3.7.2.2 Adjacency Matrix

Another approach by which a graph can be represented is by using an adjacency matrix. A matrix is a two-dimensional array. The idea here is to represent the cells with a 1 or 0 depending on whether two vertices are connected by an edge.

Given an adjacency list, it should be possible to create an adjacency matrix. A sorted list of keys of graph is required:

```
matrix_elements = sorted(graph.keys())
cols = rows = len(matrix_elements)
```

The length of the keys is used to provide the dimensions of the matrix which are stored in cols and rows. These values in cols and rows are equal:

```
adjacency_matrix = [[0 for x in range(rows)] for y in range(cols)]
edges_list = []
```

We then set up a cols by rows array, filling it with zeros. The `edges_list` variable will store the tuples that form the edges of in the graph. For example, an edge between node A and B will be stored as (A, B).

The multidimensional array is filled using a nested for loop:

```
for key in matrix_elements:
    for neighbor in graph[key]:
        edges_list.append((key,neighbor))
```

The neighbors of a vertex are obtained by `graph[key]` . The key in combination with the `neighbor` is then used to create the tuple stored in `edges_list`.

The output of the iteration is as follows:

```
[('A', 'B'), ('A', 'C'), ('B', 'E'), ('B', 'A'), ('C', 'A'),
('C', 'B'), ('C', 'E'), ('C', 'F'), ('E', 'B'), ('E', 'C'),
('F', 'C')]
```

What needs to be done now is to fill the our multidimensional array by using 1 to mark the presence of an edge with the line:

```
for edge in edges_list:
    index_of_first_vertex = matrix_elements.index(edge[0])
    index_of_second_vertex = matrix_elements.index(edge[1])
    adjacency_matrix[index_of_first_vertex][index_of_second_vertex] = 1
```

The `matrix_elements` array has its `rows` and `cols` starting from A through to E with the indices 0 through to 5. The `for` loop iterates through our list of tuples and uses the index method to get the corresponding index where an edge is to be stored.

The adjacency matrix produced looks like so:

```
[0, 1, 1, 0, 0]
[1, 0, 0, 1, 0]
[1, 1, 0, 1, 1]
[0, 1, 1, 0, 0]
[0, 0, 1, 0, 0]
```

At column 1 and row 1, the 0 there represents the absence of an edge between A and A. On column 2 and row 3, there is an edge between C and B.

We have seen how the the adjacency list and adjacency matrix of an undirected and unweighted graph can be implemented. However, it is important to note that both methods of representing graphs can also be applied to directed and/or weighted graphs. We will be exploring these graphs in the exercises.

3.7.3 Graph Traversal

Since graphs don't necessarily have an ordered structure, traversing a graph can be more involving. Traversal normally involves keeping track of which nodes or vertices have already been visited and which ones have not. A common strategy is to follow a path until a dead end is reached, then walking back up until there is a point where there is an alternative path. We can also iteratively move from one node to another in order to traverse the full graph or part of it. In the next section, we will discuss breadth and depth-first search algorithms for graph traversal.

3.7.3.1 Breadth-First Search

The breadth-first search algorithm starts at a node, chooses that node or vertex as its root node, and visits the neighboring nodes, after which it explores neighbors on the next level of the graph.

Consider the following graph:

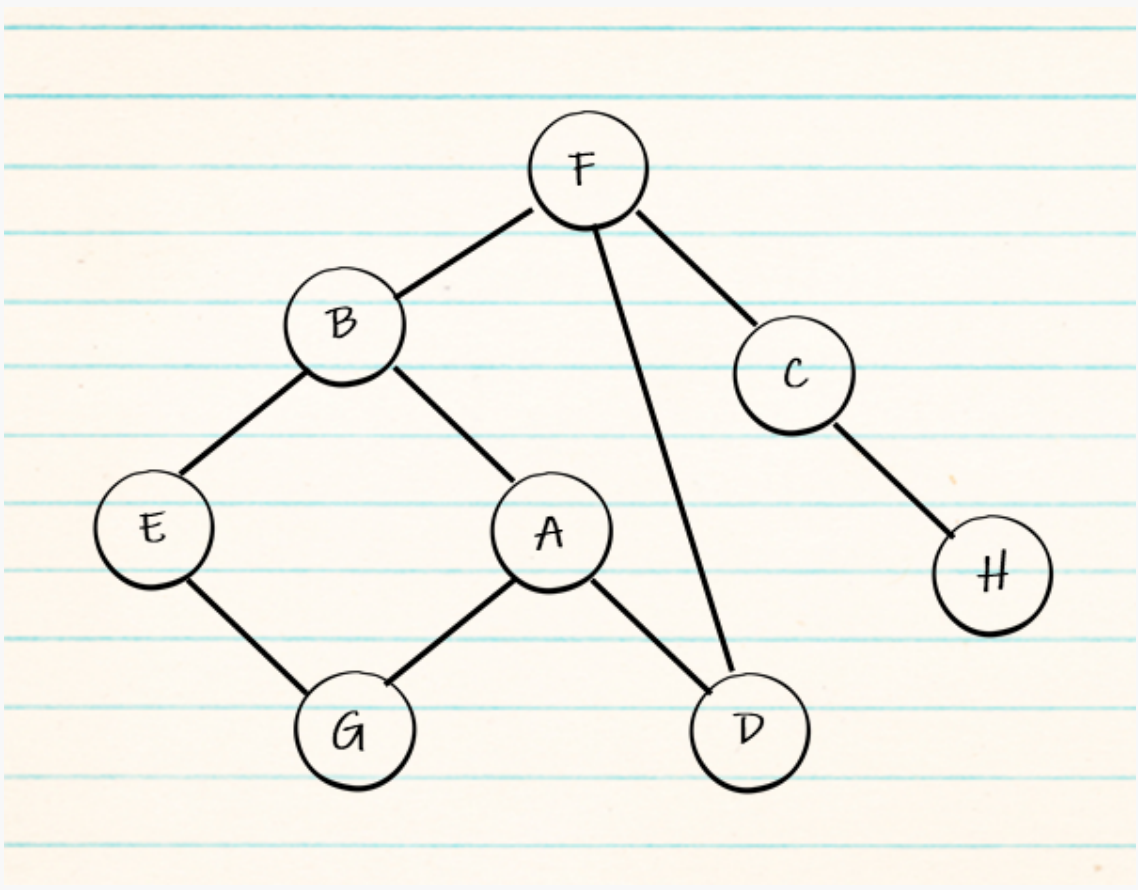

Fig 3.23. Example representation of an undirected graph for traversal.

The diagram is an example of an undirected graph. We continue to use this type of graph to help make explanation easy without being too verbose.

The adjacency list for the graph is as follows:

```
graph = dict()
graph['A'] = ['B', 'G', 'D']
graph['B'] = ['A', 'F', 'E']
graph['C'] = ['F', 'H']
graph['D'] = ['F', 'A']
graph['E'] = ['B', 'G']
graph['F'] = ['B', 'D', 'C']
graph['G'] = ['A', 'E']
graph['H'] = ['C']
```

In trying to traverse this graph breadth first, we will employ the use of a queue. The algorithm creates a list to store the nodes that have been visited as the traversal process proceeds. We shall start our traversal from node A.

Node A is queued and added to the list of visited nodes. Afterward, we use a `while` loop to effect traversal of the graph. In the `while` loop, node A is dequeued. Its unvisited adjacent nodes B, G, and D are sorted in alphabetical order and queued up. The queue will now contain the nodes B, D, and G. These nodes are also added to the list of visited nodes. At this point, we start another iteration of the `while` loop because the queue is not empty, which also means we are not really done with the traversal.

Node B is dequeued. Out of its adjacent nodes A, F, and E, node A has already been visited. Therefore, we only enqueue the nodes E and F in alphabetical order. Nodes E and F are then added to the list of visited nodes. Our queue now holds the following nodes at this point: D, G, E, and F. The list of visited nodes contains A, B, D, G, E, F.

Node D is dequeued but all of its adjacent nodes have been visited so we simply dequeue it. The next node at the front of the queue is G. We dequeue node G but we also find out that all its adjacent nodes have been visited because they are in the list of visited nodes. Node G is also dequeued. We dequeue node E too because all of its nodes have been visited. The only node in the queue now is node F.

Node F is dequeued and we realize that out of its adjacent nodes B, D, and C, only node C has not been visited. We then enqueue node C and add it to the list of visited nodes. Node C is dequeued. Node C has the adjacent nodes F and H but F has already been visited, leaving node H. Node H is enqueued and added to the list of visited nodes.

Finally, the last iteration of the `while` loop will lead to node H being dequeued. Its only adjacent node C has already been visited. Once the queue is completely empty, the loop breaks The output of the traversing the graph in the diagram given by `breadth_first_search(graph, 'A')` is

```
['A', 'B', 'D', 'G', 'E', 'F', 'C', 'H']
```

The code for a breadth-first search is given as follows:

```
from collections import deque

def breadth_first_search(graph, root):
    visited_vertices = list()
    graph_queue = deque([root])
    visited_vertices.append(root)
    node = root

    while len(graph_queue) > 0:
        node = graph_queue.popleft()
        adj_nodes = graph[node]
        remaining_elements = set(adj_nodes).difference(set(visited_vertices))
        if len(remaining_elements) > 0:
            for elem in sorted(remaining_elements):
                visited_vertices.append(elem)
                graph_queue.append(elem)

    return visited_vertices
```

When we want to find out whether a set of nodes are in the list of visited nodes, we use the statement `remaining_elements = set(adj_nodes).difference(set(visited_vertices))`. This uses the set object's difference method to find the nodes that are in `adj_nodes` but not in `visited_vertices`.

In the worst-case scenario, each vertex or node and edge will be traversed, thus the time complexity of the algorithm is $O(|V| + |E|)$, where $|V|$ is the number of vertices or nodes while $|E|$ is the number of edges in the graph.

3.7.3.2 Depth-First Search

As the name suggests, this algorithm traverses the depth of any particular path in the graph before traversing its breadth. As such, child nodes are visited first before sibling nodes. It works on finite graphs and requires the use of a stack to maintain the state of the algorithm:

```
def depth_first_search(graph, root):
    visited_vertices = list()
    graph_stack = list()
    graph_stack.append(root)
    node = root
    # continued below
```

The algorithm begins by creating a list to store the visited nodes. The `graph_stack` variable is used to aid the traversal process. For continuity's sake, we are using a regular Python list as a stack.

The starting node, called `root`, is passed with the graph's adjacency matrix, graph. `root` is pushed onto the stack. `node = root` holds the first node in the stack:

```
    #continued from above
    while len(graph_stack) > 0:

        if node not in visited_vertices:
            visited_vertices.append(node)
        adj_nodes = graph[node]

        # *
        if set(adj_nodes).issubset(set(visited_vertices)):
            graph_stack.pop()
        if len(graph_stack) > 0:
            node = graph_stack[-1]
            continue
        else:
            remaining_elements =
set(adj_nodes).difference(set(visited_vertices))

        # **
        first_adj_node = sorted(remaining_elements)[0]
        graph_stack.append(first_adj_node)
        node = first_adj_node

    return visited_vertices
```

The body of the `while` loop will be executed provided the stack is not empty. If `node` is not in the list of visited nodes, we add it. All adjacent nodes to `node` are collected by `adj_nodes = graph[node]`. If all the adjacent nodes have been visited, we pop that node from the stack and set node to `graph_stack[-1]`. `graph_stack[-1]` is the top node on the stack. The `continue` statement jumps back to the beginning of the `while` loop's test condition.

If, on the other hand, not all the adjacent nodes have been visited, the nodes that are yet to be visited are obtained by finding the difference between the `adj_nodes` and `visited_vertices` with the statement `remaining_elements = set(adj_nodes).difference(set(visited_vertices))`.

The first item within `sorted(remaining_elements)` is assigned to `first_adj_node`, and pushed onto the stack. We then point the top of the stack to this node.

When the `while` loop exists, we will return the `visited_vertices`.

Dry running the algorithm will prove useful. Consider the following graph:

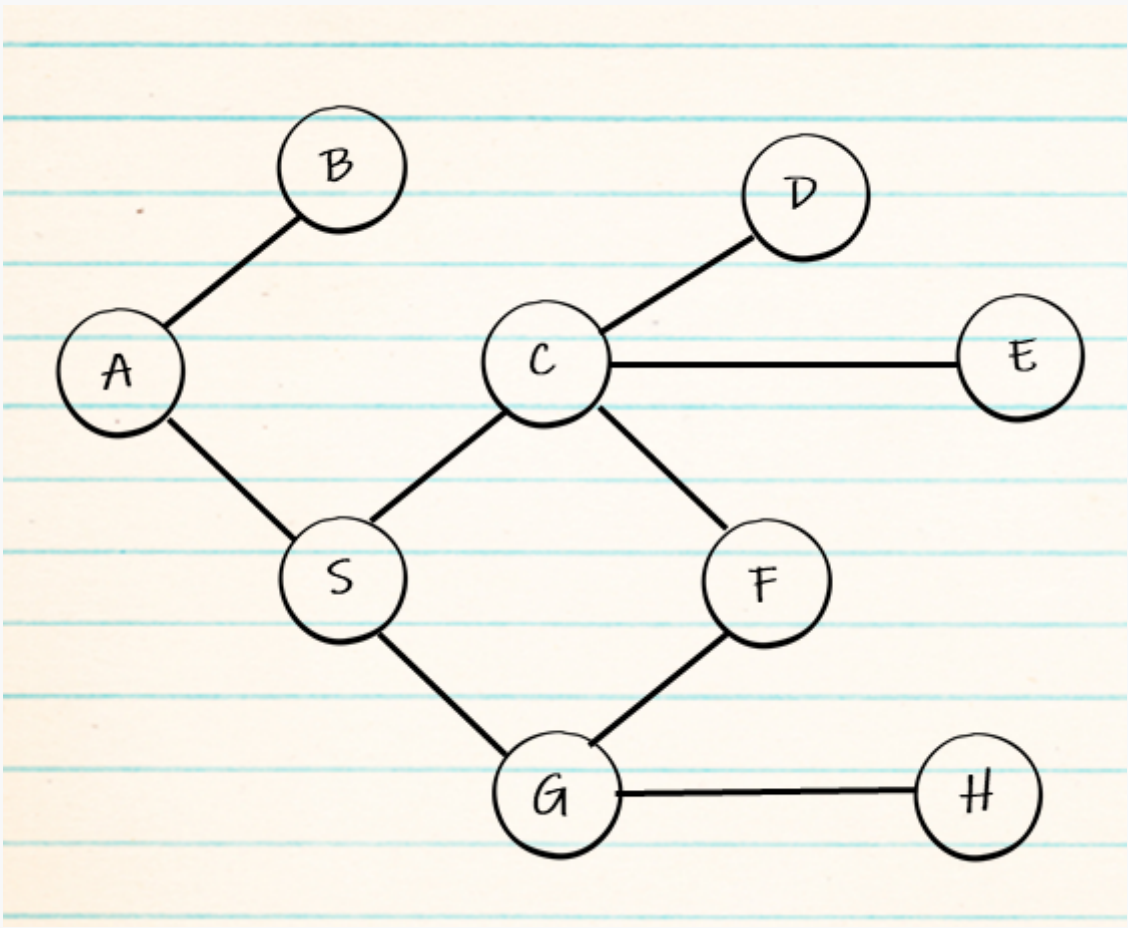

Fig 3.24. Example representation of an undirected graph for depth-first search.

The adjacency list of such a graph is given as follows:

```
graph = dict()
graph['A'] = ['B', 'S']
graph['B'] = ['A']
graph['S'] = ['A','G','C']
graph['D'] = ['C']
graph['G'] = ['S','F','H']
graph['H'] = ['G','E']
graph['E'] = ['C','H']
graph['F'] = ['C','G']
graph['C'] = ['D','S','E','F']
```

Node A is chosen as our beginning node. Node A is pushed onto the stack and added to the `visited_vertices` list. In doing so, we mark it as having been visited. The stack `graph_stack` is implemented with a simple Python list. Our stack now has A as its only element. We examine node A's adjacent nodes B and S. To test whether all the adjacent nodes of A have been visited, we use the `if` statement:

```
# *
if set(adj_nodes).issubset(set(visited_vertices)):
    graph_stack.pop()
if len(graph_stack) > 0:
    node = graph_stack[-1]
continue
```

If all the nodes have been visited, we pop the top of the stack. If the stack `graph_stack` is not empty, we assign the node on top of the stack to `node` and start the beginning of another execution of the body of the `while` loop. The statement `set(adj_nodes).issubset(set(visited_vertices))` will evaluate to `True` if all the nodes in `adj_nodes` are a subset of `visited_vertices`. If the `if` statement fails, it means that some nodes remain to be visited. We obtain that list of nodes with `remaining_elements = set(adj_nodes).difference(set(visited_vertices))`.

From the diagram, nodes B and S will be stored in `remaining_elements`. We will access the list in alphabetical order:

```
# **
first_adj_node = sorted(remaining_elements)[0]
graph_stack.append(first_adj_node)
node = first_adj_node
```

We sort `remaining_elements` and return the first node to `first_adj_node`. This will return B. We push node B onto the stack by appending it to the `graph_stack`. We prepare node B for access by assigning it to `node`.

On the next iteration of the `while` loop, we add node B to the list of visited nodes. We discover that the only adjacent node to B, which is A, has already been visited. Because all the adjacent nodes of B have been visited, we pop it off the stack, leaving node A as the only element on the stack. We return to node A and examine whether all of its adjacent nodes have been visited. The node A now has S as the only unvisited node. We push S to the stack and begin the whole process again.

The output of the traversal `breadth_first_search(graph, 'A')` is:

```
['A', 'B', 'S', 'C', 'G', 'D', 'E', 'F', 'H']
```

Depth-first searches find application in solving maze problems, finding connected components, and finding the bridges of a graph, among others.

3.7.4 Directed Acyclic Graphs

Directed graphs without directed cycles are encountered in many applications. Such a directed graph is often referred to as a ***directed acyclic graph***, or ***DAG***, for short.

One such application of a DAG includes the scheduling constraints between the tasks of a project. In order to manage a large project, it is convenient to break it up into a collection of smaller tasks. The tasks, however, are rarely independent, because scheduling constraints exist between them. For example, in a house building project, the task of ordering nails obviously precedes the task of nailing shingles to the roof deck. Clearly, scheduling constraints cannot have circularities, because they would make the project impossible. The scheduling constraints impose restrictions on the order in which the tasks can be executed. Namely, if a constraint says that task a must be completed before task b is started, then a must precede b in the order of execution of the tasks. Thus, if we model a feasible set of tasks as vertices of a directed graph, and we place a directed edge from u to v whenever the task for u must be executed before the task for v, then we define a *directed acyclic graph*.

Consider the following DAG:

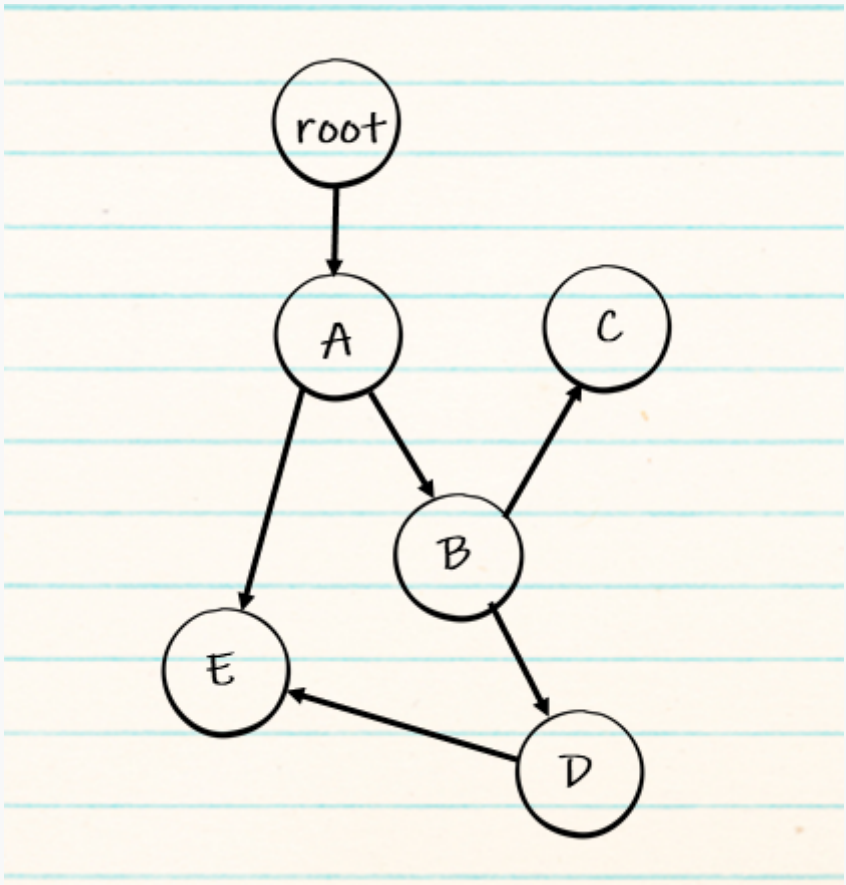

Fig 3.25. Example Representation of a Directed Acyclic Graph

While we can use adjacency list or matrix to represent the DAG in Fig 3.25, we shall use a Python package called NetworkX instead. NetworkX provides a standard programming interface and graph implementation that is suitable for many applications including designing, generating and drawing network models. You may need to install it (refer here).

Here's how we can construct our sample graph with the NetworkX library:

```
import networkx as nx

graph = nx.DiGraph()    # DiGraph is short for "directed graph"
graph.add_edges_from([("root", "a"), ("a", "b"), ("a", "e"), ("b", "c"), ("b",
"d"), ("d", "e")])
```

The directed graph is modeled as a list of tuples that connect the nodes. Remember that these connections are referred to as “edges” in graph nomenclature. Take another look at the graph image and observe how all the arguments to `add_edges_from` match up with the arrows in the graph.

NetworkX is 'smart' enough to infer the nodes from a collection of edges. Calling `graph.nodes()` will output:

```
NodeView(('root', 'a', 'b', 'e', 'c', 'd'))
```

3.7.4.1 Topological Sorting

Topological sorting (or topological ordering) is a form of sorting of a DAG such that for every directed edge from node u to node v, u comes before v in the order. For a directed graph to have a topological ordering, it must be acyclic. We can verify by using `nx.is_directed_acyclic_graph(graph)`.

Our graph has nodes (A, B, C, etc.) and directed edges (AB, BC, BD, DE, etc.). Here’s a couple of requirements that our topological sort need to satisfy:

- for AB, A needs to come before B in the ordering
- for BC, B needs to come before C
- for BD, B needs to come before D
- for DE, D needs to come before E

However, this also means that a DAG may have more than one topological ordering. Fortunately, NetworkX provides a `topological_sort()` method which ensures exactly this. It presents an iterable, that guarantees that when you arrive at a node, you have already visited all the nodes it on which it depends. Let’s run the method `list(nx.topological_sort(graph))` and see if all our requirements are met.

```
['root', 'a', 'b', 'd', 'e', 'c']
```

3.7.5 Minimum Spanning Tree

We now move on to another common graph-based problem. Suppose you have been given a weighted undirected graph such as the following:

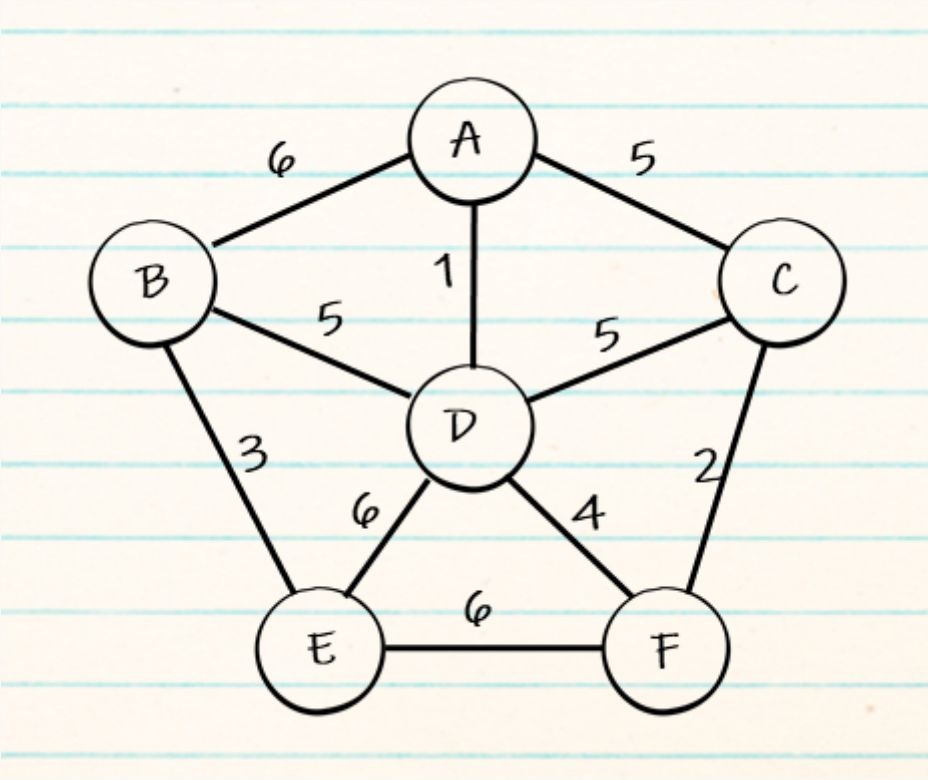

Fig 3.26. Example of a Weighted Spanning Tree

We could think of the vertices as representing houses, and the weights as the distances between them. Now imagine that you are tasked with supplying all these houses with some commodity such as water, gas, or electricity. For obvious reasons, you will want to keep the amount of digging and laying of pipes or cable to1 a minimum. So, what is the best pipe or cable layout that you can find, i.e. what layout has the shortest overall length?

Obviously, we will have to choose some of the edges to dig along, but not all of them. For example, if we have already chosen the edge between A and D, and the one between B and D, then there is no reason to also have the one between A and B. More generally, it is clear that we want to avoid circles. Also, assuming that we have only one feeding-in point (it is of no importance which of the vertices that is), we need the whole layout to be connected. We have seen already that a connected graph without circles is a tree.

Hence, what we are looking for is a ***minimum spanning tree*** of the graph. A *spanning tree* of a graph is a subgraph that is a tree which connects all the vertices together, so it 'spans' the original graph but using fewer edges. Here, minimum refers to the sum of all the weights of the edges contained in that tree, so a minimum spanning tree has total weight less than or equal to the total weight of every other spanning tree. As we shall see, there will not necessarily be a unique minimum spanning tree for a given graph.

In order to come up with some ideas which will allow us to develop an algorithm for the minimal spanning tree problem, we shall need to make some observations about minimum spanning trees. Let us assume, for the time being, that all the weights in the above graph were equal, to give us some idea of what kind of shape a minimum spanning tree might have under those circumstances. Here are some examples:

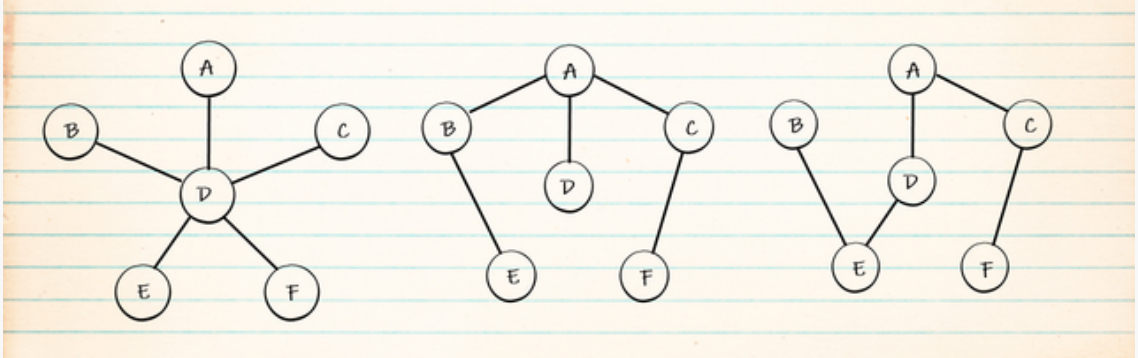

Fig 3.27. Possible shapes of the minimum spanning tree

We can immediately notice that their general shape is such that if we add any of the remaining edges, we would create a circle. Then we can see that going from one spanning tree to another can be achieved by removing an edge and replacing it by another (to the vertex which would otherwise be unconnected) such that no circle is created. These observations are not quite sufficient to lead to an algorithm, but they are good enough to let us prove that the algorithms we find do actually work.

3.7.5.1 Prim's Algorithm

Suppose that we already have a spanning tree connecting some set of vertices S. Then we can consider all the edges which connect a vertex in S to one outside of S, and add to S one of those that has minimum weight. This cannot possibly create a circle, since it must add a vertex not yet in S. This process can be repeated, starting with any vertex to be the sole element of S, which is a trivial minimum spanning tree containing no edges. This approach is known as ***Prim's algorithm***.

When implementing Prim's algorithm, one can use either an array or a list to keep track of the set of vertices S reached so far. One could then maintain another array or list `closest` which, for each vertex i not yet in S, keeps track of the vertex in S closest to i. That is, the vertex in S which has an edge to i with minimal weight. If `closest` also keeps track of the weights of those edges, we could save time, because we would then only have to check the weights mentioned in that array or list.

For the above graph, starting with $S = A$, the tree is built up as follows:

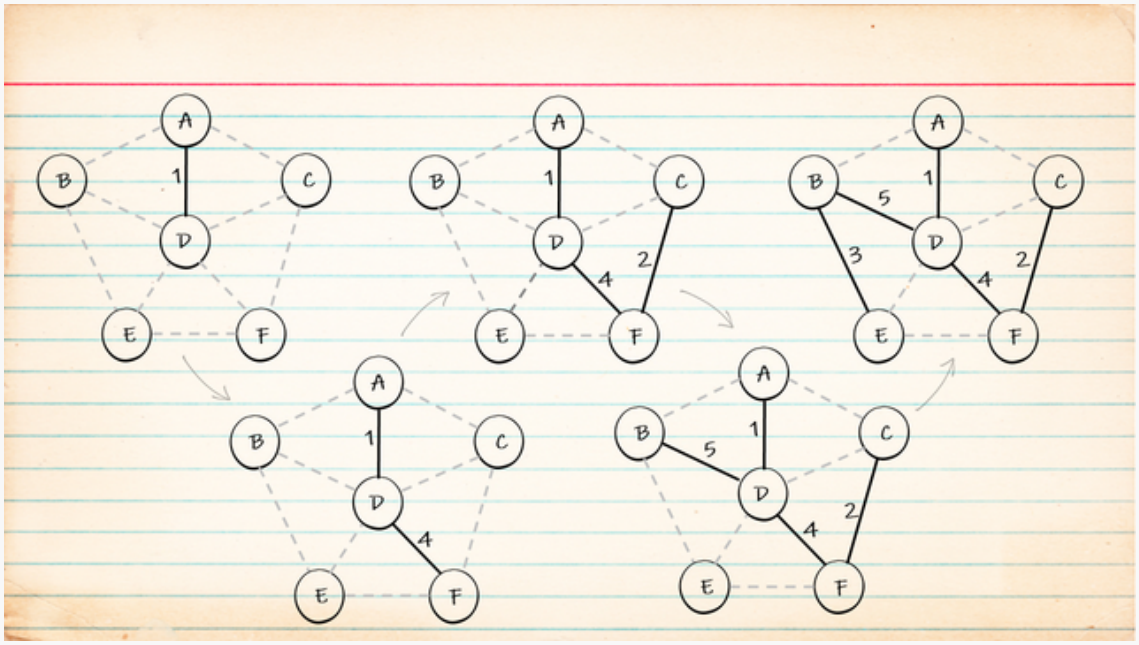

Fig 3.28. Minimum Spanning Tree using Prim's Algorithm

It is slightly more challenging to produce a convincing argument that this algorithm really works. It is clear that Prim's algorithm must result in a spanning tree, because it generates a tree that spans all the vertices, but it is not obvious that it is minimum. There are several possible proofs that it is, but none are straightforward. The simplest works by showing that the set of all possible minimal spanning trees X_i must include the output of Prim's algorithm.

Let Y be the output of Prim's algorithm, and X_1 be any minimum spanning tree. The following illustrates such a situation:

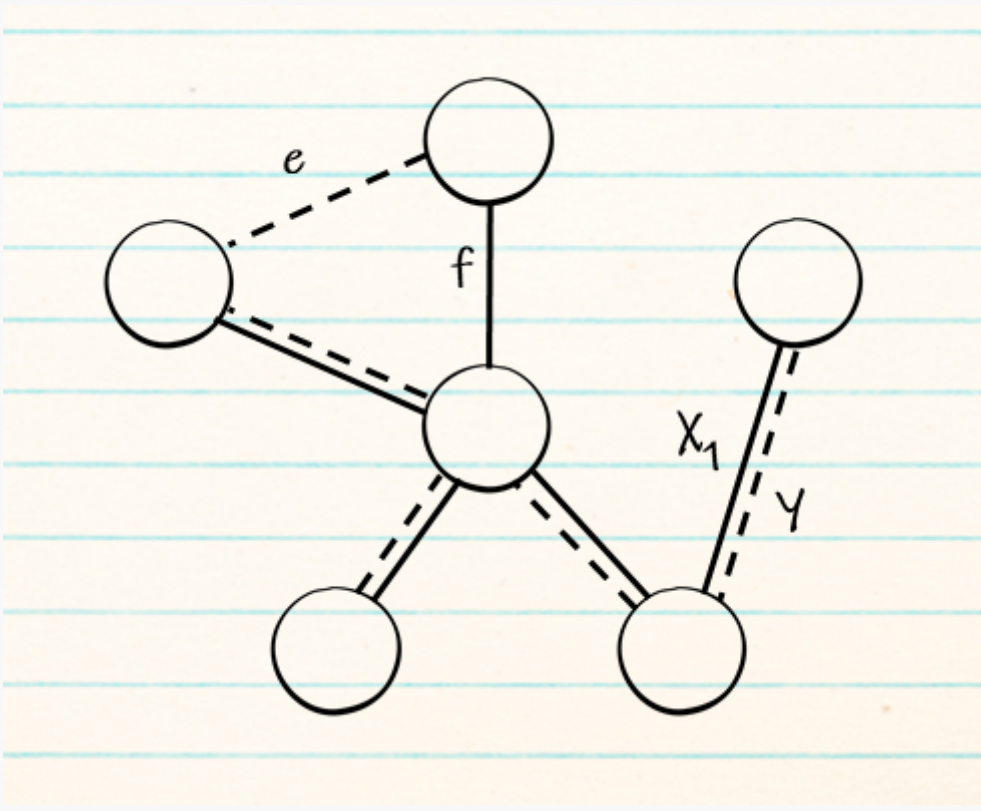

Fig 3.29. Illustration to proof that Prim's algorithm outputs a spanning tree that is minimum.

We don't actually need to know what X_1 is – we just need to know the properties it must satisfy, and then systematically work through all the possibilities, showing that Y is a minimal spanning tree in each case. Clearly, if $X_1 = Y$, then Prim's algorithm has generated a minimum spanning tree. Otherwise, let e be the first edge added to Y that is not in X_1. Then, since X_1 is a spanning tree, it must include a path connecting the two endpoints of e, and because circles are not allowed, there must be an edge in X_1 that is not in Y , which we can call f. Since Prim's algorithm added e rather than f, we know $weight(e) \leq weight(f)$. Then create tree X_2 that is X_1 with f replaced by e. Clearly X_2 is connected, has the same number of edges as X_1, spans all the vertices, and has total weight no greater than X_1, so it must also be a minimum spanning tree. Now we can repeat this process until we have replaced all the edges in X_1 that are not in Y, and we end up with the minimum spanning tree $X_n = Y$, which completes the proof that Y is a minimum spanning tree.

The time complexity of the standard Prim's algorithm is O(n^2) because at each step we need to choose a vertex to add to S, and then update the `closest` array. we should also consider whether it really is necessary to process every vertex at each stage, because it could be sufficient to only check actually existing edges. We therefore now consider an alternative edge-based strategy.

3.7.5.2 Kruskal's Algorithm

This algorithm does not consider the vertices directly at all, but builds a minimal spanning tree by considering and adding edges as follows: Assume that we already have a collection of edges T. Then, from all the edges not yet in T, choose one with minimal weight such that its addition to T does not produce a circle, and add that to T. If we start with T being the empty set, and continue until no more edges can be added, a minimal spanning tree will be produced. This approach is known as ***Kruskal's algorithm***.

For the same graph as used for Prim's algorithm (Fig 3.26), this algorithm proceeds as follows:

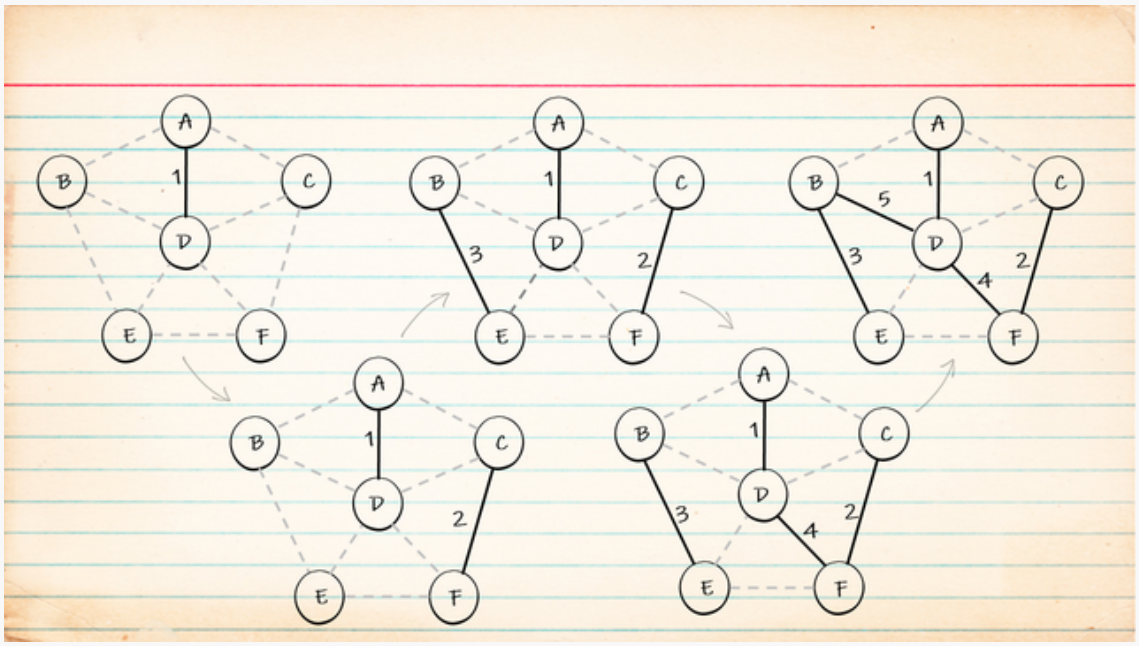

Fig 3.30. Minimum Spanning Tree using Kruskal's Algorithm

In practice, Kruskal's algorithm is implemented in a rather different way to Prim's algorithm. The general idea of the most efficient approaches is to start by sorting the edges according to their weights, and then simply go through that list of edges in order of increasing weight, and either add them to T, or reject them if they would produce a circle. There are implementations of that which can be achieved with overall time complexity O($e \log e$), which is dominated by the O($e \log e$) complexity of sorting the e edges in the first place.

This means that the choice between Prim's algorithm and Kruskal's algorithm depends on the connectivity of the particular graph under consideration. If the graph is sparse, i.e. the number of edges is not much more than the number of vertices, then Kruskal's algorithm will have O($n \log n$) complexity but will be faster than the standard O(n^2) Prim's algorithm. However, if the graph is highly connected, i.e. the number of edges is near the square of the number of vertices, it will have complexity O($n^2 \log n$) and be slower than Prim's.