9

Graphs and Algorithms

Graphs are a non-linear data structure, in which the problem is represented as a network by connecting a set of nodes with edges, like a telephone network or social network. For example, in a graph, nodes can represent different cities while the links between them represent edges. Graphs are one of the most important data structures; they are used to solve many computing problems, especially when the problem is represented in the form of objects and their connection, e.g. to find out the shortest path from one city to another city. Graphs are useful data structures for solving real-world problems in which the problem can be represented as a network-like structure. In this chapter, we will be discussing the most important and popular concepts related to graphs.

In this chapter, we will learn about the following concepts:

- The concept of the graph data structure
- How to represent a graph and traverse it
- Different operations and their implementation on graphs

First, we will be looking into the different types of graphs.

Graphs

A graph is a set of a finite number of vertices (also known as nodes) and edges, in which the edges are the links between vertices, and each edge in a graph joins two distinct nodes. Moreover, a graph is a formal mathematical representation of a network, i.e. a graph **G** is an ordered pair of a set **V** of vertices and a set **E** of edges, given as `G = (V, E)` in formal mathematical notation.

An example of a graph is shown in *Figure 9.1*:

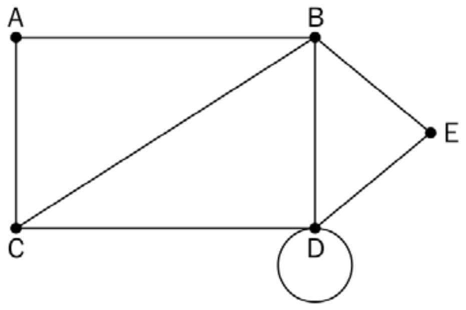


Figure 9.1: An example of a graph

The graph `G = (V, E)` in *Figure 9.1* can be described as below:

- `V = {A, B, C, D, E}`
- `E = {{A, B}, {A, C}, {B, C}, {B, D}, {C, D}, {D, D}, {B, E}, {D, E}}`
- `G = (V, E)`

Let's discuss some of the important definitions of a graph:

- **Node or vertex**: A point or node in a graph is called a vertex. In the preceding diagram, the vertices or nodes are **A**, **B**, **C**, **D**, and **E** and are denoted by a dot.
- **Edge**: This is a connection between two vertices. The line connecting **A** and **B** is an example of an edge.
- **Loop**: When an edge from a node is returned to itself , that edge forms a loop, e.g. **D** node.
- **Degree of a vertex/node**: The total number of edges that are incidental on a given vertex is called the degree of that vertex. For example, the degree of the **B** vertex in the previous diagram is 4.
- **Adjacency**: This refers to the connection(s) between any two nodes; thus, if there is a connection between any two vertices or nodes, then they are said to be adjacent to each other. For example, the C node is adjacent to the A node because there is an edge between them.
- **Path**: A sequence of vertices and edges between any two nodes represents a path. For example, **CABE** represents a path from the **C** node to the **E** node.
- **Leaf vertex** (also called *pendant vertex*): A vertex or node is called a leaf vertex or pendant vertex if it has exactly one degree.

Now, we shall take a look at the different types of graphs.

Directed and undirected graphs

Graphs are represented by the edges between the nodes. The connecting edges can be considered directed or undirected. If the connecting edges in a graph are undirected, then the graph is called an undirected graph, and if the connecting edges in a graph are directed, then it is called a directed graph. An undirected graph simply represents edges as lines between the nodes. There is no additional information about the relationship between the nodes, other than the fact that they are connected. For example, in *Figure 9.2*, we demonstrate an undirected graph of four nodes, **A**, **B**, **C**, and **D**, which are connected using edges:

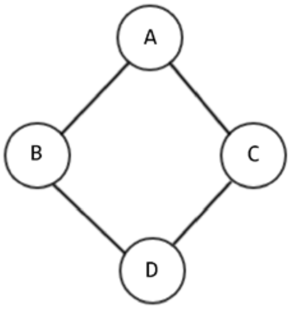


Figure 9.2: An example of an undirected graph

In a directed graph, the edges provide information on the direction of connection between any two nodes in a graph. If an edge from **A** node to **B** is said to be directed, then the edge (**A**, **B**) would not be equal to the edge (**B**, **A**). The directed edges are drawn as lines with arrows, which will point in whichever direction the edge connects the two nodes.

For example, in *Figure 9.3*, we show a directed graph where many nodes are connected using directed edges:

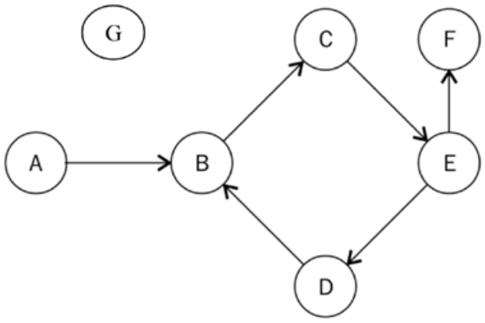


Figure 9.3: An example of a directed graph

The arrow of an edge determines the flow of direction. One can only move from **A** to **B**, as shown in the preceding diagram—not **B** to **A**. In a directed graph, each node (or vertex) has an indegree and an outdegree. Let's have a look at what these are:

- **Indegree**: The total number of edges that come into a vertex in the graph is called the indegree of that vertex. For example, in the previous diagram, the **E** node has 1 indegree, due to edge **CE** coming into the **E** node.
- **Outdegree**: The total number of edges that go out from a vertex in the graph is called the outdegree of that vertex. For example, the **E** node in the previous diagram has an outdegree of 2, as it has two edges, **EF** and **ED**, going out of that node.
- **Isolated vertex**: A node or vertex is called an isolated vertex when it has a degree of zero, as shown as **G** node in *Figure 9.3*.
- **Source vertex**: A vertex is called a source vertex if it has an indegree of zero. For example, in the previous diagram, the **A** node is the source vertex.
- **Sink vertex**: A vertex is a sink vertex if it has an outdegree of zero. For example, in the previous diagram, the **F** node is the sink vertex.

Now that we understand how directed graphs work, we can look into directed acyclic graphs.

Directed acyclic graphs

A **directed acyclic graph** (**DAG**) is a directed graph with no cycles; in a DAG all the edges are directed from one node to another node so that the sequence of edges never forms a closed loop. A cycle in a graph is formed when the starting node of the first edge is equal to the ending node of the last edge in a sequence.

A DAG is shown in *Figure 9.4* in which all the edges in the graph are directed and the graph does not have any cycles:

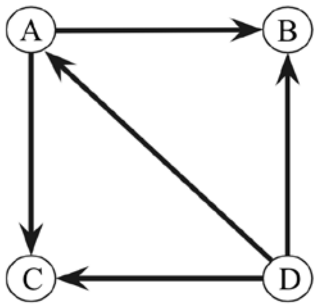


Figure 9.4: An example of a directed acyclic graph

So, in a directed acyclic graph, if we start on any path from a given node, we never find a path that ends on the same node. A DAG has many applications, such as in job scheduling, citation graphs, and data compression.

Next, we will discuss weighted graphs.

Weighted graphs

A weighted graph is a graph that has a numeric weight associated with the edges in the graph. A weighted graph can be either a directed or an undirected graph. The numeric weight can be used to indicate distance or cost, depending on the purpose of the graph:

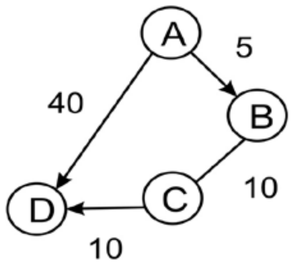


Figure 9.5: An example of a weighted graph

Let's consider an example – *Figure 9.5* indicates different ways to reach from **A** node to **D** node. There are two possible paths, such as from **A** node to **D** node, or it can be nodes **A-B-C-D** through **B** node and **C** node. Now, depending on the weights associated with the edges, any one of the paths can be considered better than the others for the journey – e.g. assume the weights in this graph represent the distance between two nodes, and we want to find out the shortest path between **A-D** nodes; then one possible path **A-D** has an associated cost of 40, and another possible path **A-B-C-D** has an associated cost of 25. In this case, the better path is **A-B-C-D**, which has a lower distance.

Next, we will discuss bipartite graphs.

Bipartite graphs

A bipartite graph (also known as a bigraph) is a special graph in which all the nodes of the graph can be divided into two sets in such a way that edges connect the nodes from one set to the nodes of another set. See *Figure 9.6* for a sample bipartite graph; all the nodes of the graphs are divided into two independent sets, i.e., set U and set V, so that each edge in the graph has one end in set U and another end in set V (e.g. in edge (*A*, *B*), one end or one vertex is from set U, and another end or another vertex is from set V).

In bipartite graphs, no edge will connect to the nodes of the same set:

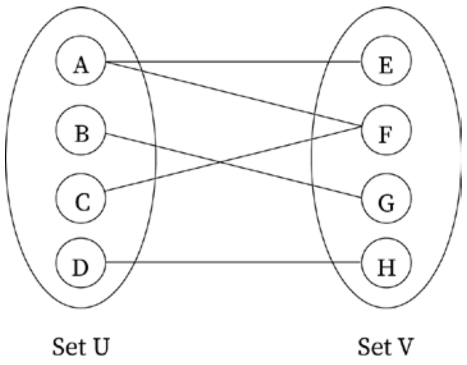


Figure 9.6: An example of a bipartite graph

Bipartite graphs are useful when we need to model a relationship between two different classes of objects, for example, a graph of applicants and jobs, in which we may need to model the relationship between these two different groups; another example may be a bipartite graph of football players and clubs in which we may need to model if a player has played for a particular club or not.

Next, we will discuss different graph representation techniques.

Graph representations

A graph representation technique means how we store the graph in memory, i.e., how we store the vertices, edges, and weights (if the graph is a weighted graph). Graphs can be represented with two methods, i.e. (1) an adjacency list, and (2) an adjacency matrix.

An adjacency list representation is based on a linked list. In this, we represent the graph by maintaining a list of neighbors (also called an adjacent node) for every vertex (or node) of the graph. In an adjacency matrix representation of a graph, we maintain a matrix that represents which node is adjacent to which other node in the graph; i.e., the adjacency matrix has the information of every edge in the graph, which is represented by cells of the matrix.

Either of these two representations can be used; however, our choice depends on the application where we will be using the graph representation. An adjacency list is preferable when we expect that the graph is going to be sparse and we will have a smaller number of edges; e.g. if a graph of 200 nodes has say 100 edges, it is better to store this kind of graph in an adjacency list, because if we use an adjacency matrix, the size of the matrix will be 200x200 with a lot of zero values. The adjacency matrix is preferable when we expect the graph to have a lot of edges, and the matrix will be dense. In the adjacency matrix, the lookup and check for the presence or absence of an edge are very easy compared to adjacency list representation.

We will be discussing adjacency matrices in detail in subsequent sections. First, we will take a look at adjacency lists.

Adjacency lists

In this representation, all the nodes directly connected to a node x are listed in its adjacent list of nodes. The graph is represented by displaying the adjacent list for all the nodes of the graph.

Two nodes, A and B, in the graph shown in *Figure 9.7*, are said to be adjacent if there is a direct connection between them:

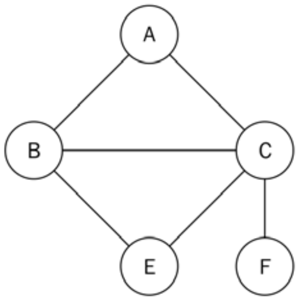


Figure 9.7: A sample graph of five nodes

A linked list can be used to implement the adjacency list. In order to represent the graph, we need the number of linked lists equal to the total number of nodes in the graph. At each index, the adjacent nodes to that vertex are stored. For example, consider the adjacency list shown in *Figure 9.8* corresponding to the sample graph shown in *Figure 9.7:*

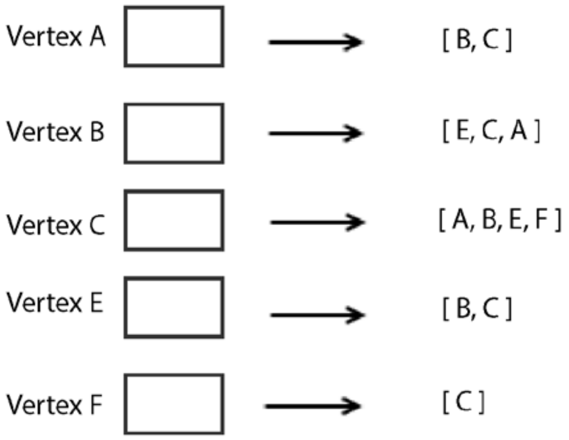


Figure 9.8: Adjacency list for the graph shown in Figure 9.7

Here, the first node represents the `A` vertex of the graph, with its adjacent nodes being `B` and `C`. The second node represents the `B` vertex of the graph, with its adjacent nodes of `E`, `C`, and `A`. Similarly, the other vertices, `C`, `E`, and `F`, of the graph are represented with their adjacent nodes, as shown in the previous *Figure 9.8*.

Using a `list` for the representation is quite restrictive, because we lack the ability to directly use the vertex labels. So, to implement a graph efficiently using Python, a `dictionary` data structure is used since it is more suitable to represent the graph. To implement the same graph using a dictionary data structure, we can use the following code snippet:

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

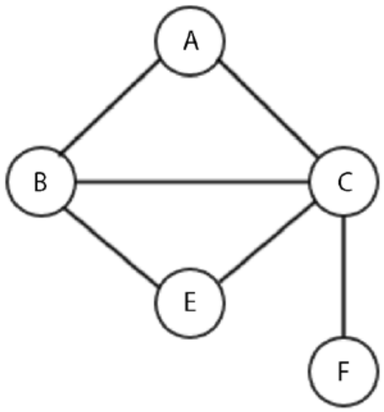
Now we can easily establish that the `A` vertex has the adjacent vertices of `B` and `C`. The `F` vertex has the `C` vertex as its only neighbor. Similarly, the `B` vertex has adjacent vertices of `E`, `C`, and `A`.

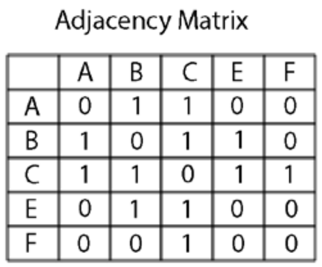
The adjacency list is a preferable graph representation technique when the graph is going to be sparse and we may need to add or delete the nodes in the graph frequently. However, it is very difficult to check whether a given edge is present in the graph or not using this technique.

Next, we will discuss another method of graph representation, i.e., the adjacency matrix.

Adjacency matrix

Another approach to representing a graph is to use an adjacency matrix. In this, the graph is represented by showing the nodes and their interconnections through edges. Using this method, the dimensions (`V x V`) of a matrix are used to represent the graph, where each cell denotes an edge in the graph. A matrix is a two-dimensional array. So, the idea here is to represent the cells of the matrix with a `1` or a `0`, depending on whether two nodes are connected by an edge or not. We show an example graph, along with its corresponding adjacency matrix, in *Figure 9.9*:




Adjacency Matrix

	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

Figure 9.9: Adjacency matrix for a given graph

An adjacency matrix can be implemented using the given adjacency list. To implement the adjacency matrix, let's take the previous dictionary-based implementation of the graph. Firstly, we have to obtain the key elements of the adjacency matrix. It is important to note that these matrix elements are the vertices of the graph. We can get the key elements by sorting the keys of the graph. The code snippet for this is as follows:

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

Next, the length of the keys of the graph will be the dimensions of the adjacency matrix, which are stored in `cols` and `rows`. The values of the `cols` and `rows` are equal.

So, now, we create an empty adjacency matrix of the dimensions `cols` by `rows`, initially filling all the values with zeros. The code snippet to initialize an empty adjacency matrix is as follows:

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

The `edges_list` variable will store the tuples that form the edges in the graph. For example, an edge between the A and B nodes 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))

print(edges_list)
```

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 preceding Python code for storing the edges of the graph is as follows:

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

The next step in implementing the adjacency matrix is to fill it, using `1` to denote the presence of an edge in the graph. This can be done with the `adjacency_matrix[index_of_first_vertex][index_of_second_vertex] = 1` statement. The full code snippet that marks the presence of edges of the graph is as follows:

```
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

print(adjacency_matrix)
```

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

The output of the preceding code is the adjacency matrix for the sample graph shown previously in *Figure 9.9*. The adjacency matrix produced looks like the following:

```
[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 row `1` and column `1`, `0` represents the absence of an edge between **A** and **A**. Similarly, at row `3` and column `2` there is a value of `1` that denotes the edge between the **C** and **B** vertices in the graph.

The use of the adjacency matrix for graph representation is suitable when we have to frequently look up and check the presence or absence of an edge between two nodes in the graph, e.g. in creating routing tables in networks, searching routes in public transport applications and navigation systems, etc. Adjacency matrices are not suitable when nodes are frequently added or deleted within a graph, in those situations, the adjacency list is a better technique.

Next, let us discuss different graph traversal methods in which we visit all the nodes of the given graph.

Graph traversals

A graph traversal means to visit all the vertices of the graph while keeping track of which nodes or vertices have already been visited and which ones have not. A graph traversal algorithm is efficient if it traverses all the nodes of the graph in the minimum possible time. Graph traversal, also known as a graph search algorithm, is quite similar to the tree traversal algorithms like `preorder`, `inorder`, `postorder`, and level order algorithms; similar to them, in a graph search algorithm we start with a node and traverse through edges to all other nodes in the graph.

A common strategy of graph traversal is to follow a path until a dead end is reached, then traverse back up until there is a point where we meet an alternative path. We can also iteratively move from one node to another in order to traverse the full graph or part of it. Graph traversal algorithms are very important in answering many fundamental problems—they can be useful to determine how to get from one vertex to another in a graph, and which path from **A** node to **B** node in a graph is better than other paths. For example, graph traversal algorithms can be useful in finding out the shortest route from one city to another in a network of cities.

In the next section, we will discuss two important graph traversal algorithms: **breadth-first search** (**BFS**) and **depth-first search** (**DFS**).

Breadth-first traversal

Breadth-first search (**BFS**) works very similarly to how a level order traversal algorithm works in a tree data structure. The BFS algorithm also works level by level; it starts by visiting the root node at level 0, and then all the nodes at the first level directly connected to the root node are visited at level 1. The level 1 node has a distance of 1 from the root node. After visiting all the nodes at level 1, the level 2 nodes are visited next. Likewise, all the nodes in the graph are traversed level by level until all the nodes are visited. So, breadth-first traversal algorithms work breadthwise in the graph.

A queue data structure is used to store the information of vertices that are to be visited in a graph. We begin with the starting node. Firstly, we visit that node, and then we look up all of its neighboring, or adjacent, vertices. We first visit these adjacent vertices one by one, while adding their neighbors to the list of vertices that are to be visited. We follow this process until we have visited all the vertices of the graph, ensuring that no vertex is visited twice.

Let's consider an example to better understand the working of the breadth-first traversal for graphs, using the sample shown in *Figure 9.10*:

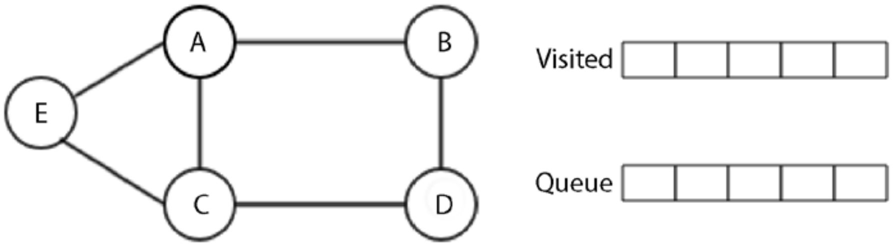


Figure 9.10: A sample graph

In *Figure 9.10*, we have a graph of five nodes on the left, and on the right, a queue data structure to store the vertices to be visited. We start visiting the first node, i.e., **A** node, and then we add all its adjacent vertices, **B**, **C**, and **E**, to the queue. Here, it is important to note that there are multiple ways of adding the adjacent nodes to the queue since there are three nodes, **B**, **C**, and **E**, that can be added to the queue as either **BCE**, **CEB**, **CBE**, **BEC**, or **ECB**, each of which would give us different tree traversal results.

All of these possible solutions to the graph traversal are correct, but in this example, we add the nodes in alphabetical order just to keep things simple in the queue, i.e., **BCE**. The **A** node is visited as shown in *Figure 9.11*:

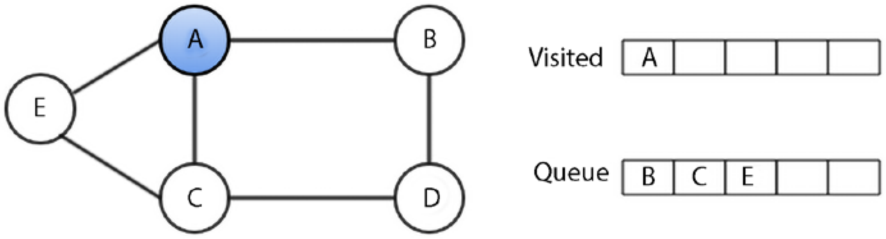


Figure 9.11: Node A is visited in breadth-first traversal

Once we have visited the **A** vertex, next, we visit its first adjacent vertex, **B**, and add those adjacent vertices of vertex **B** that are not already added in the queue or not visited. In this case, we have to add the **D** vertex (since it has two vertices, **A** and **D** nodes, out of which **A** is already visited) to the queue, as shown in *Figure 9.12*:

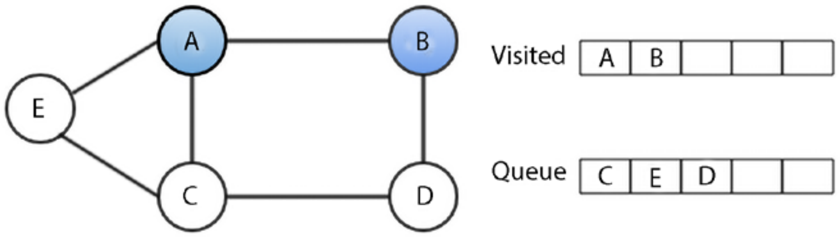


Figure 9.12: Node B is visited in breadth-first traversal

Now, after visiting the **B** vertex, we visit the next vertex from the queue—the **C** vertex. And again, add those adjacent vertices that have not already been added to the queue. In this case, there are no unrecorded vertices left, as shown in *Figure 9.13*:

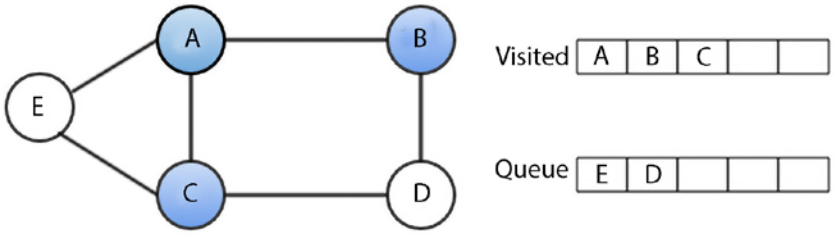


Figure 9.13: Node C is visited in breadth-first traversal

After visiting the **C** vertex, we visit the next vertex from the queue, the **E** vertex, as shown in *Figure 9.14*:

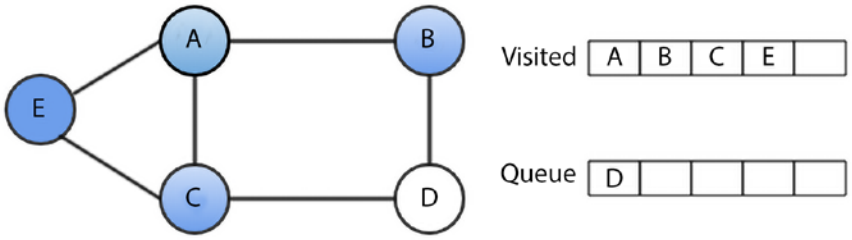


Figure 9.14: Node E is visited in breadth-first traversal

Similarly, after visiting the **E** vertex, we visit the **D** vertex in the last step, as shown in *Figure 9.15*:

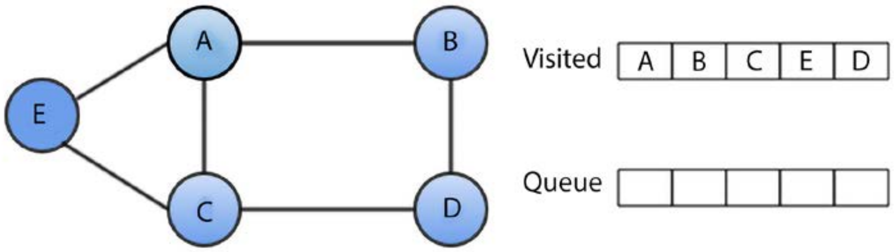


Figure 9.15: D node is visited in breadth-first traversal

Therefore, the BFS algorithm for traversing the preceding graph visits the vertices in the order of **A-B-C-E-D**. This is one of the possible solutions to the BFS traversal for the preceding graph, but we can get many possible solutions, depending on how we add the adjacent nodes to the queue.

To understand the implementation of this algorithm in Python, we will use another example of an undirected graph, as shown in *Figure 9.16*:

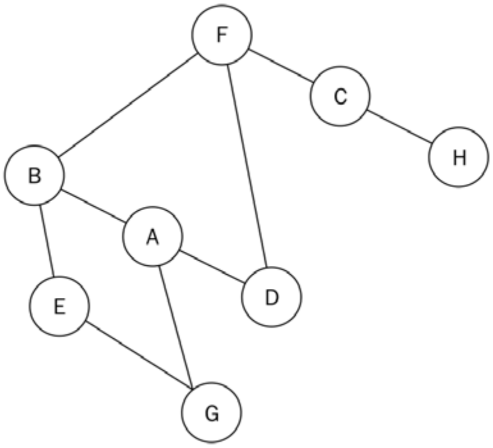


Figure 9.16: An undirected sample graph

The adjacency list for the graph shown in *Figure 9.16* 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']
```

After storing the graph using the adjacency list, the implementation of the BFS algorithm is as follows, which we will discuss with an example in detail:

```
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
```

To traverse this graph using the breadth-first algorithm, we first initialize the queue and the source node. We start traversal from **A** node. Firstly, **A** node is queued and added to the list of visited nodes. Afterward, we use a `while` loop to affect the traversal of the graph. In the first iteration of the `while` loop, node A is dequeued.

Next, all the unvisited adjacent nodes of **A** node, which are **B**, **D**, and **G**, are sorted in alphabetical order and queued up. The queue now contains nodes **B**, **D**, and **G**. This is shown in *Figure 9.17*:

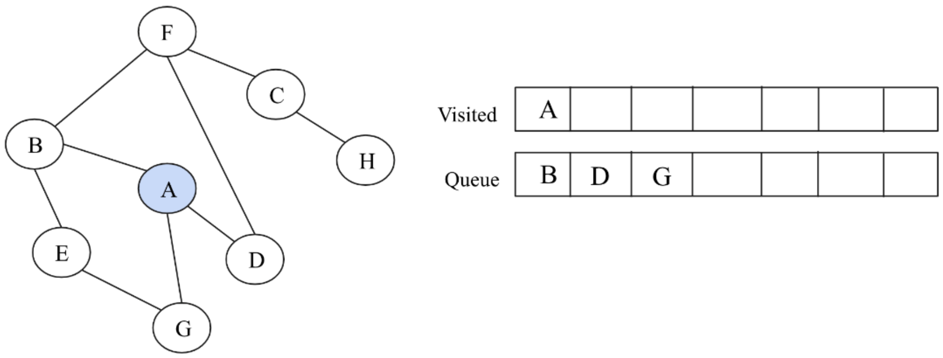


Figure 9.17: Node A is visited using the BFS algorithm

For implementation, we add all these nodes (**B**, **D**, **G**) to the list of visited nodes, and then we add the adjacent/neighboring nodes of these nodes. At this point, we start another iteration of the `while` loop. After visiting **A** node, **B** node is dequeued. Out of its adjacent nodes (**A**, **E**, and **F**), **A** node has already been visited. Therefore, we only queue the **E** and **F** nodes in alphabetical order, as shown in *Figure 9.18*.

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

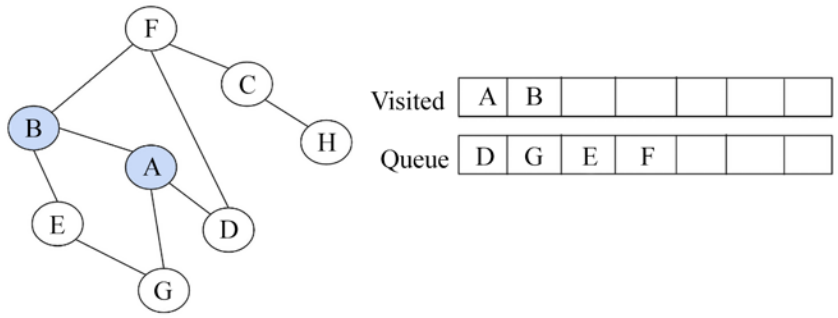


Figure 9.18: Node B is visited using the BFS algorithm

The queue now holds the following nodes at this point—**D**, **G**, **E**, and **F**. The **D** node 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 the **G** node, but we also find out that all its adjacent nodes have been visited because they are in the list of visited nodes. So, the **G** node is also dequeued. We dequeue the **E** node too because all of its adjacent nodes have also been visited. The only node in the queue now is the **F** node; this is shown in *Figure 9.19*:

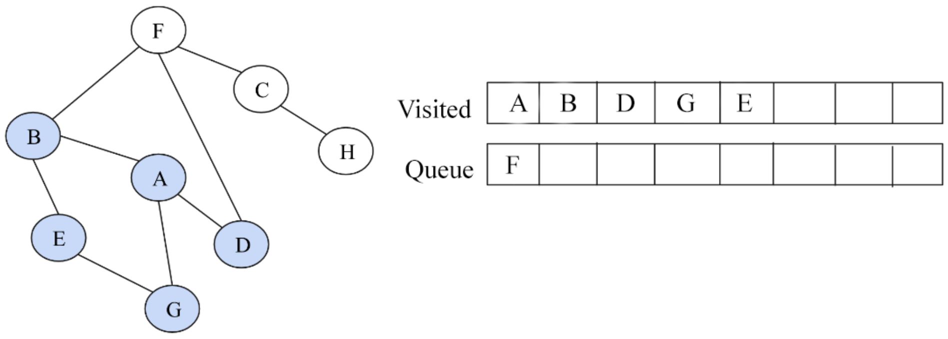


Figure 9.19: Node E is visited using the BFS algorithm

The **F** node is dequeued, and we see that out of its adjacent nodes, **B**, **D**, and **C**, only **C** has not been visited. We then enqueue the **C** node and add it to the list of visited nodes, as shown in *Figure 9.20*:

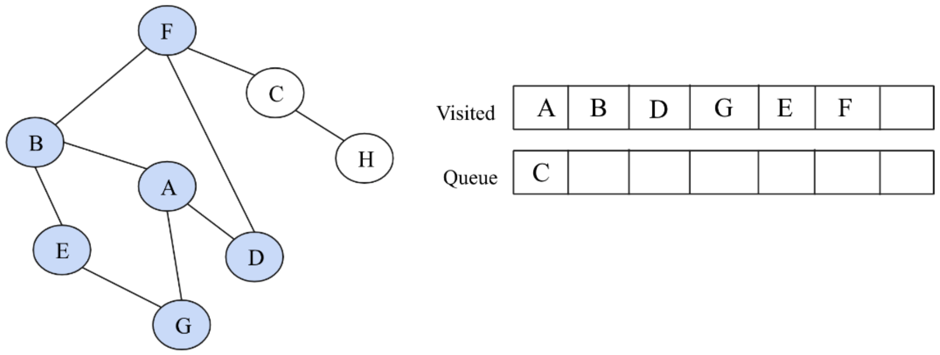


Figure 9.20: Node E is visited using the BFS algorithm

Then, the **C** node is dequeued. **C** has the adjacent nodes of **F** and **H**, but **F** has already been visited, leaving the **H** node. The **H** node is enqueued and added to the list of visited nodes. Finally, the last iteration of the `while` loop will lead to the **H** node being dequeued.

Its only adjacent node, **C**, has already been visited. Once the queue is empty, the loop breaks. This is shown in *Figure 9.21*:

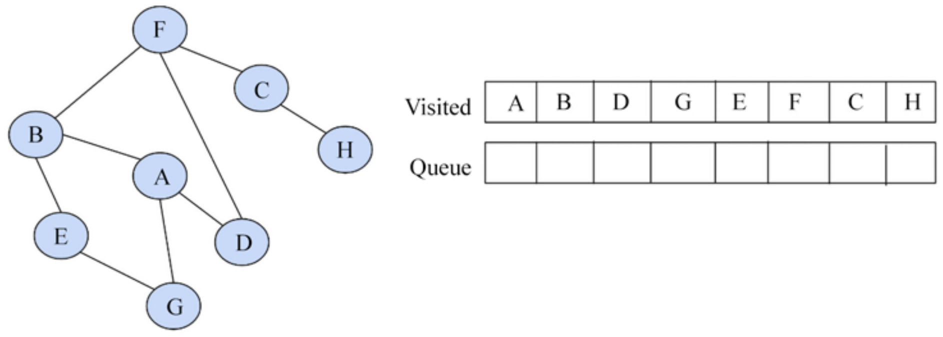


Figure 9.21: Final node H is visited using the BFS algorithm

The output of the traversal of the given graph using the BFS algorithm is **A**, **B**, **D**, **G**, **E**, **F**, **C**, and **H**.

When we run the above BFS code on the graph shown in *Figure 9.16* using the following code:

```python
print(breadth_first_search(graph, 'A'))
```

We get the following sequence of nodes when we traverse the graph shown in *Figure 9.16*:

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

In the worst-case scenario, each node and the edge will need to be traversed, and hence each node will be enqueued and dequeued at least once. The time taken for each enqueue and dequeue operation is O(1), so the total time for this is O(V). Further, the time spent scanning the adjacency list for every vertex is O(E). So, the total time complexity of the BFS algorithm is `O(|V| + |E|)`, where `|V|` is the number of vertices or nodes, while `|E|` is the number of edges in the graph.

The BFS algorithm is very useful for constructing the shortest path traversal in a graph with minimal iterations. As for some of the real-world applications of BFS, it can be used to create an efficient web crawler in which multiple levels of indexes can be maintained for search engines, and it can maintain a list of closed web pages from a source web page. BFS can also be useful for navigation systems in which neighboring locations can be easily retrieved from a graph of different locations.

Next, we will discuss another graph traversal algorithm, i.e., the depth-first search algorithm.

Depth-first search

As the name suggests, the **depth-first search** (**DFS**) or traversal algorithm traverses the graph similar to how the `preorder` traversal algorithm works in trees. In the DFS algorithm, we traverse the tree in the depth of any particular path in the graph. As such, child nodes are visited first before sibling nodes.

In this, we start with the root node; firstly we visit it, and then we see all the adjacent vertices of the current node. We start visiting one of the adjacent nodes. If the edge leads to a visited node, we backtrack to the current node. And, if the edge leads to an unvisited node, then we go to that node and continue processing from that node. We continue the same process until we reach a dead end when there is no unvisited node; in that case, we backtrack to previous nodes, and we stop when we reach the root node while backtracking.

Let's take an example to understand the working of the DFS algorithm using the graph shown in *Figure 9.22*:

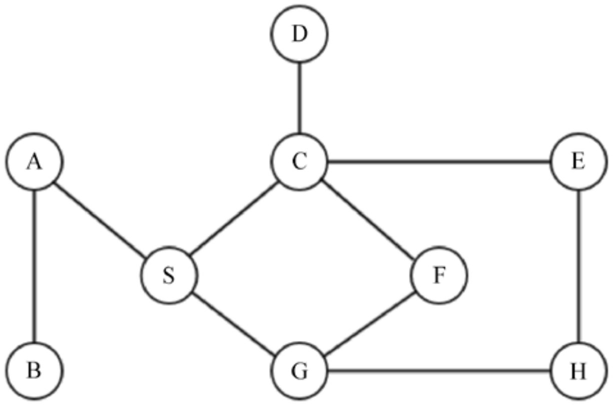


Figure 9.22: An example graph for understanding the DFS algorithm

We start by visiting the **A** node, and then we look at the neighbors of the **A** vertex, then a neighbor of that neighbor, and so on. After visiting the **A** vertex, we visit one of its neighbors, **B** (in our example, we sort alphabetically; however, any neighbor can be added), as shown in *Figure 9.23*:

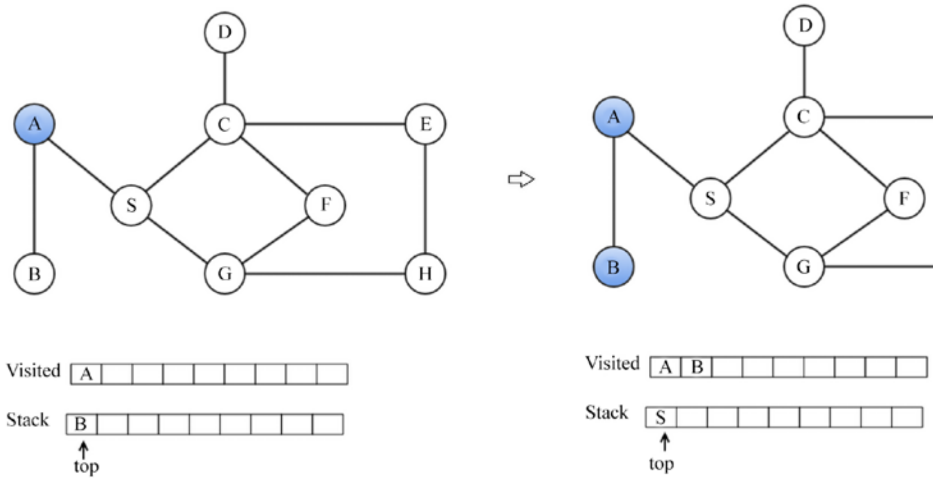


Figure 9.23: Nodes A and B are visited in depth-first traversal

After visiting the **B** vertex, we look at another neighbor of **A**, that is, **S**, as there is no vertex connected to **B** that can be visited. Next, we look for the neighbors of the **S** vertex, which are the **C** and **G** vertices. We visit **C** as shown in *Figure 9.24*:

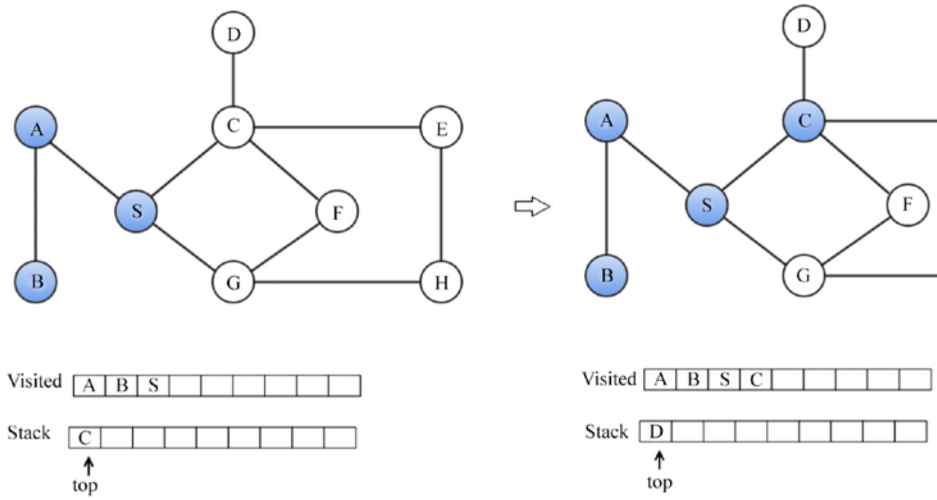


Figure 9.24: Node C is visited in depth-first traversal

After visiting the **C** node, we visit its neighboring vertices, **D** and **E**, as shown in *Figure 9.25*:

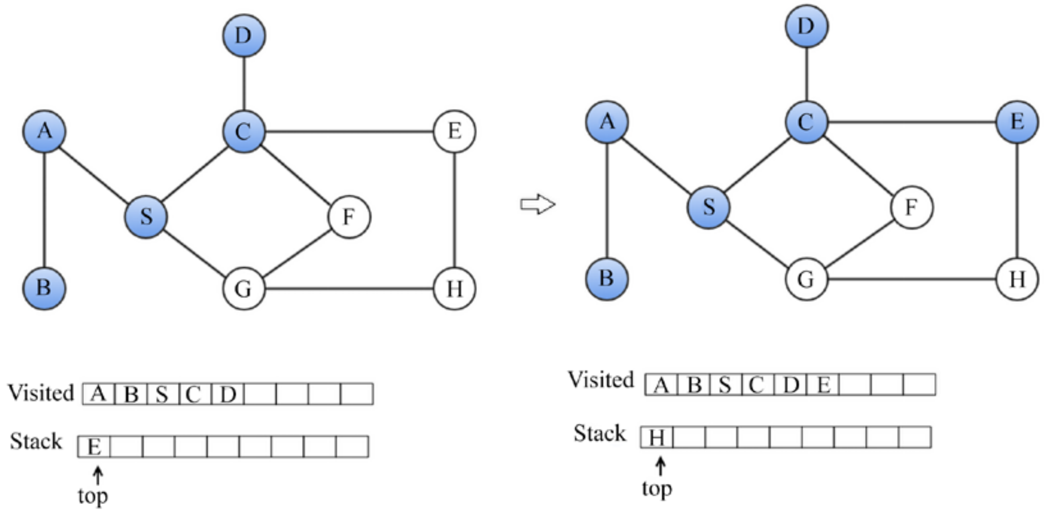


Figure 9.25: D and E nodes are visited in depth-first traversal

Similarly, after visiting the **E** vertex, we visit the **H** and **G** vertices, as shown in *Figure 9.26*:

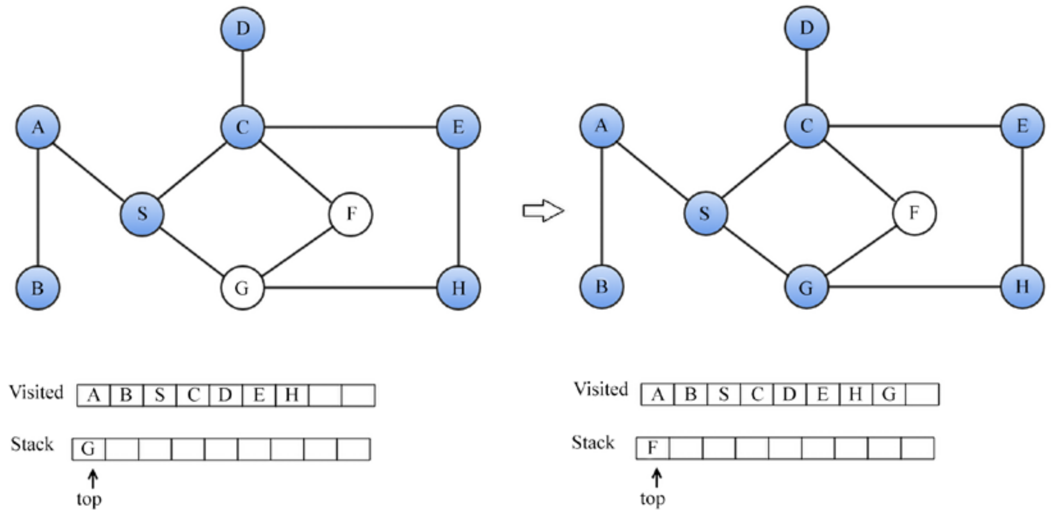


Figure 9.26: H and F nodes are visited in depth-first traversal

Finally, we visit the **F** node, as shown in *Figure 9.27*:

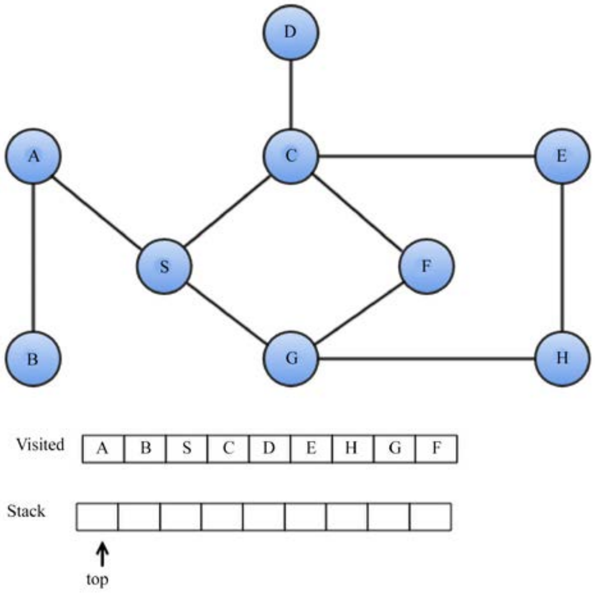


Figure 9.27: F node is visited in depth-first traversal

The output of the DFS traversal is **A-B-S-C-D-E-H-G-F**.

To implement DFS, we start with the adjacency list of the given graph. Here is the adjacency list of the preceding graph:

```
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']
```

The implementation of the DFS algorithm begins with creating a list to store the visited nodes. The `graph_stack` stack variable is used to aid the traversal process. We are using a Python list as a stack.

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

```
def depth_first_search(graph, root):
    visited_vertices = list()
    graph_stack = list()

    graph_stack.append(root)
    node = root
        while graph_stack:
            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 the `node` under consideration is not in the list of visited nodes, we add it. All adjacent nodes of `node` are collected by `adj_nodes = graph[node]`. If all the adjacent nodes have been visited, we pop the top node from the stack and set `node` to `graph_stack[-1]`. Here, `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, then the nodes that are yet to be visited are obtained by finding the difference between the `adj_nodes` and `visited_vertices` with the `remaining_elements = set(adj_nodes).difference(set(visited_vertices))` statement.

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 exits, we will return visited_vertices.

We will now explain the working of the source code by relating it to the previous example. The **A** node is chosen as our starting 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 graph_stack stack is implemented with a simple Python list. Our stack now has **A** as its only element. We examine the **A** node'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 graph_stack 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 set(adj_nodes).issubset(set(visited_vertices)) statement 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)).

Referring to the diagram, the **B** and **S** nodes will be stored in remaining_elements. We will access the list in alphabetical order as follows:

```
        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 the **B** node onto the stack by appending it to the graph_stack. We prepare the **B** node for access by assigning it to node.

On the next iteration of the while loop, we add the **B** node 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 **A** as the only element on the stack. We return to **A** and examine whether all of its adjacent nodes have been visited. The **A** node 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 is `A-B-S-C-D-E-H-G-F`.

The time complexity of DFS is O(V+E) when we use an adjacency list, and O(V^2) when we use an adjacency matrix for graph representation. The time complexity of DFS with the adjacency list is lower because getting the adjacent nodes is easier, whereas it is not efficient with the adjacency matrix.

DFS can be applied to solving maze problems, finding connected components, cycle detection in graphs, and finding the bridges of a graph, among other use cases.

We have discussed very important graph traversal algorithms; now let us discuss some more useful graph-related algorithms for finding the spanning tree from the given graph. Spanning trees are useful for several real-world problems such as the traveling salesman problem.

Other useful graph methods

It is very often that we need to use graphs for finding a path between two nodes. Sometimes, it is necessary to find all the paths between nodes, and in some situations, we might need to find the shortest path between nodes. For example, in routing applications, we generally use various algorithms to determine the shortest path from the source node to the destination node. For an unweighted graph, we would simply determine the path with the lowest number of edges between them. If a weighted graph is given, we have to calculate the total weight of passing through a set of edges.

Thus, in a different situation, we may have to find the longest or shortest path using different algorithms, such as a **Minimum Spanning Tree**, which we look into in the next section.

Minimum Spanning Tree

A **Minimum Spanning Tree** (**MST**) is a subset of the edges of the connected graph with an edge-weighted graph that connects all the nodes of the graph, with the lowest possible total edge weights and no cycle. More formally, given a connected graph G, where G = (V, E) with real-valued edge weights, an MST is a subgraph with a subset of the edges $T \subseteq E$ so that the sum of edge weights is minimum and there is no cycle. There are many possible spanning trees that can connect all the nodes of the graph without any cycle, but the the minimum weight spanning tree is a spanning tree that has the lowest total edge weight (also called cost) among all other possible spanning trees. An example graph is shown in *Figure 9.28* along with its corresponding MST (on the right) in which we can observe that all the nodes are connected and have a subset of edges taken from the original graph (on the left).

The MST has the lowest total weight of all the edges, i.e. (1+4+2+4+5 = 16) among all the other possible spanning trees:

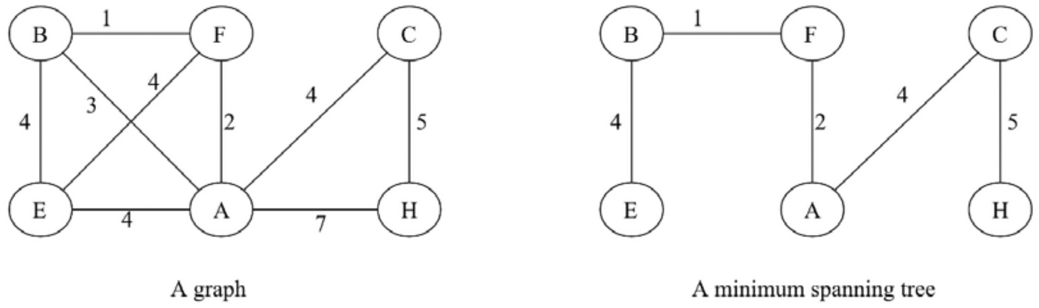


Figure 9.28: A sample graph with the corresponding Minimum Spanning Tree

The MST has diverse real-world applications. They are mainly used in network design for road congestion, hydraulic cables, electric cable networks, and even cluster analysis.

First, let us discuss Kruskal's minimum spanning tree algorithm.

Kruskal's Minimum Spanning Tree algorithm

Kruskal's algorithm is a widely used algorithm for finding the spanning tree from a given weighted, connected, and undirected graph. It is based on the greedy approach, as we firstly find the edge with the lowest weight and add it to the tree, and then in each iteration, we add the edge with the lowest weight to the spanning tree so that we do not form a cycle. In this algorithm, initially, we treat all the vertices of the graph as a separate tree, and then in each iteration we select edge with the lowest weight in such a way that it does not form a cycle. These separate trees are combined, and it grows to form a spanning tree. We repeat this process until all the nodes are processed. The algorithm works as follows:

1. Initialize an empty MST (M) with zero edges
2. Sort all the edges according to their weights
3. For each edge from the sorted list, we add them one by one to the MST (M) in such a way that it does not form a cycle

Let's consider an example.

We start by selecting the edge with the lowest weight (weight 1), as represented by the dotted line shown in *Figure 9.29*:

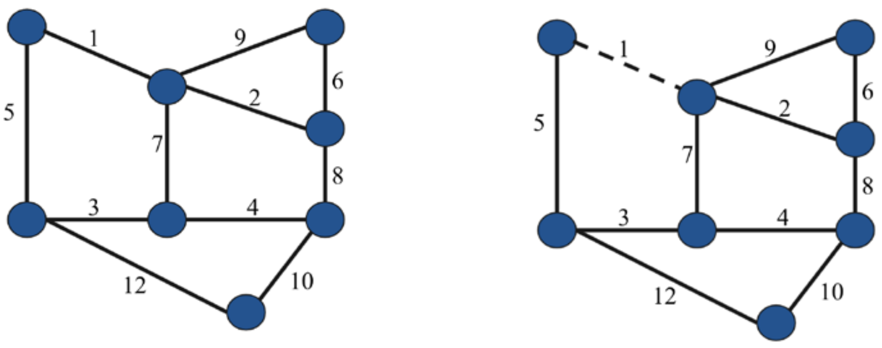


Figure 9.29: Selecting the first edge with the lowest weight in the spanning tree

After selecting the edge with weight 1, we select the edge with weight 2 and then the edge with weight 3, since these are the next lowest weights, as shown in *Figure 9.30*:

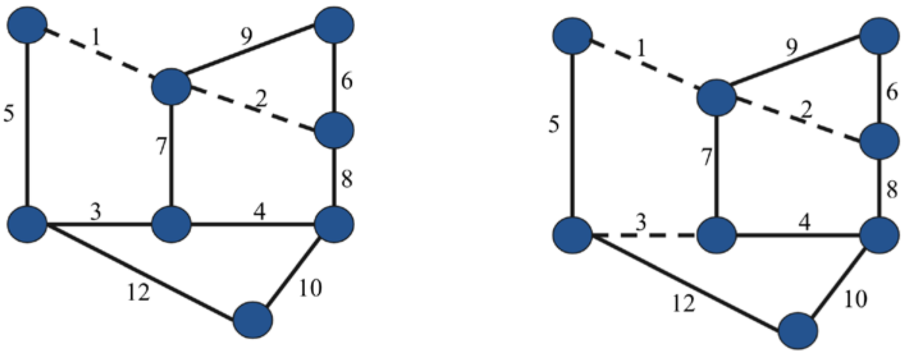


Figure 9.30: Selecting edges with wieghts 2 and 3 in the spanning tree

Similarly, we select the next edges with weights 4 and 5 respectively as shown in *Figure 9.31*:

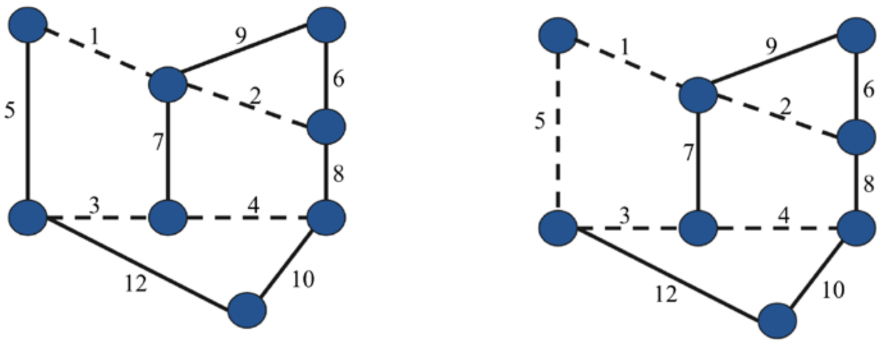


Figure 9.31: Selecting edges with weights 4 and 5 in the spanning tree

Next, we select the next edge with weight 6 and make it a dotted line. After that, we see that the lowest weight is 7 but if we select it, it makes a cycle, so we ignore it. Next, we check the edge with weight 8, and then 9, which are also ignored because they will also form a cycle. So, the next edge with the lowest weight, 10, is selected. This is shown in *Figure 9.32*:

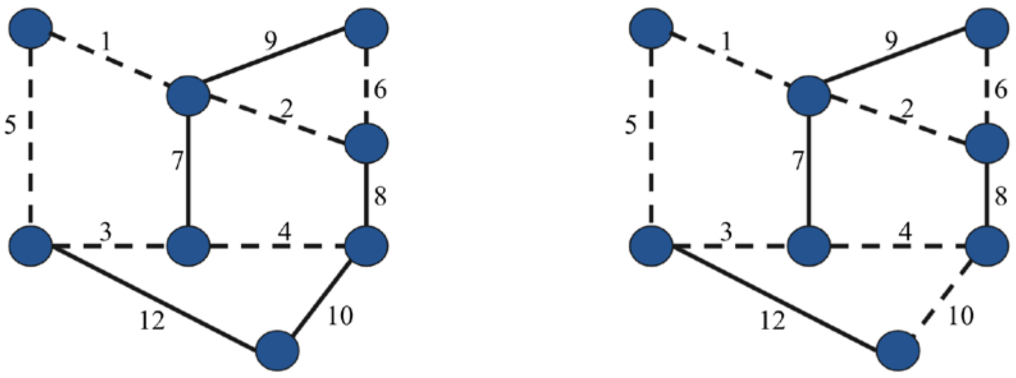


Figure 9.32: Selecting edges with weights 6 and 10 in the spanning tree

Finally, we see the following spanning tree using Kruskal's algorithm, as shown in *Figure 9.33*:

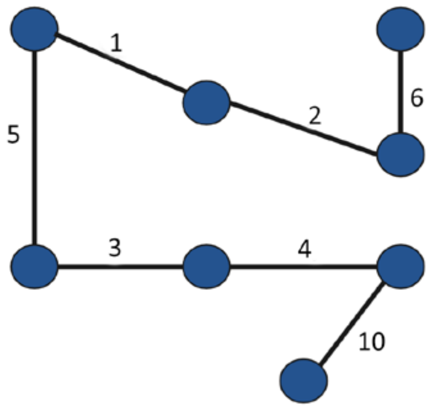


Figure 9.33: The final spanning tree created using Kruskal's algorithm

Kruskal's algorithm has many real-world applications, such as solving the traveling salesman problem (TSP), in which starting from one city, we have to visit all the different cities in a network with the minimum total cost and without visiting the same city twice. There are many other applications, such as TV networks, tour operations, LAN networks, and electric grids.

The time complexity of Kruskal's algorithm is O (E log (E)) or O (E log(V)), where E is the number of edges and V is the number of vertices.

Now, let us discuss one more popular MST algorithm in the next section.

Prim's Minimum Spanning Tree algorithm

Prim's algorithm is also based on a greedy approach to find the minimum cost spanning tree. Prim's algorithm is very similar to the Dijkstra algorithm for finding the shortest path in a graph. In this algorithm, we start with an arbitrary node as a starting point, and then we check the outgoing edges from the selected nodes and traverse through the edge that has the lowest cost (or weights). The terms cost and weight are used interchangeably in this algorithm. So, after starting from the selected node, we grow the tree by selecting the edges, one by one, that have the lowest weight and do not form a cycle. The algorithm works as follows:

1. Create a dictionary that holds all the edges and their weights
2. Get the edges, one by one, that have the lowest cost from the dictionary and grow the tree in such a way that the cycle is not formed
3. Repeat step 2 until all the vertices are visited

Let us consider an example to understand the working of Prim's algorithm. Assuming that we arbitrarily select **A** node, we then check all the outgoing edges from **A**. Here, we have two options, **AB** and **AC**; we select edge **AC** since it has less cost/weight (weight 1), as shown in *Figure 9.34*:

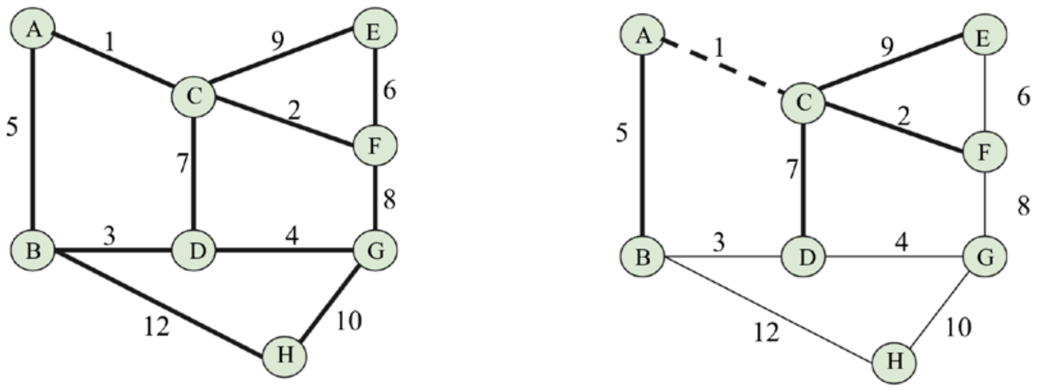


Figure 9.34: Selecting edge AC in constructing the spanning tree using Prim's algorithm

Next, we check the lowest outgoing edges from edge **AC**. We have options **AB**, **CD**, **CE**, **CF**, out of which we select edge **CF**, which has the lowest weight of 2. Likewise, we grow the tree, and next we select the lowest weighted edge, i.e., **AB**, as shown in *Figure 9.35*:

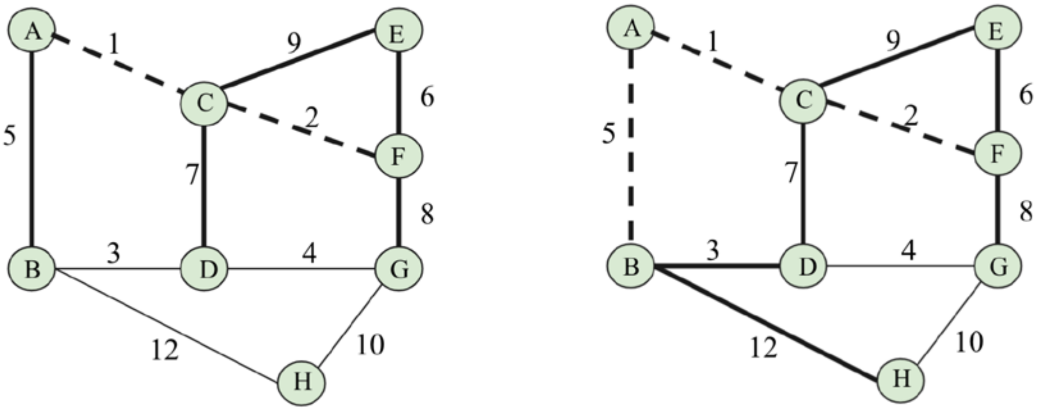


Figure 9.35: Selecting edge AB in constructing the spanning tree using Prim's algorithm

Afterward, we select edge **BD**, which has a weight of 3, and similarly, next, we select edge **DG**, which has the lowest weight of 4. This is shown in *Figure 9.36*:

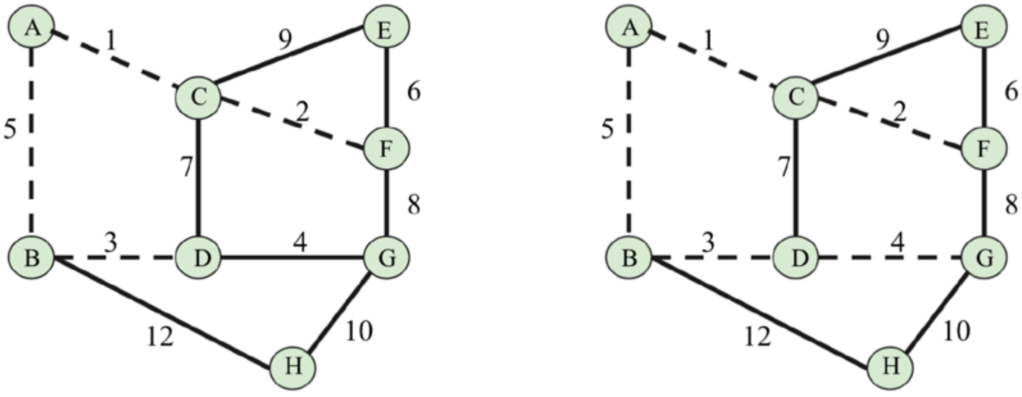


Figure 9.36: Selecting edges BD and DG in constructing the spanning tree using Prim's algorithm

Next, we select edges **FE** and **GH**, which have weights of 6 and 10 respectively, as shown in *Figure 9.37*:

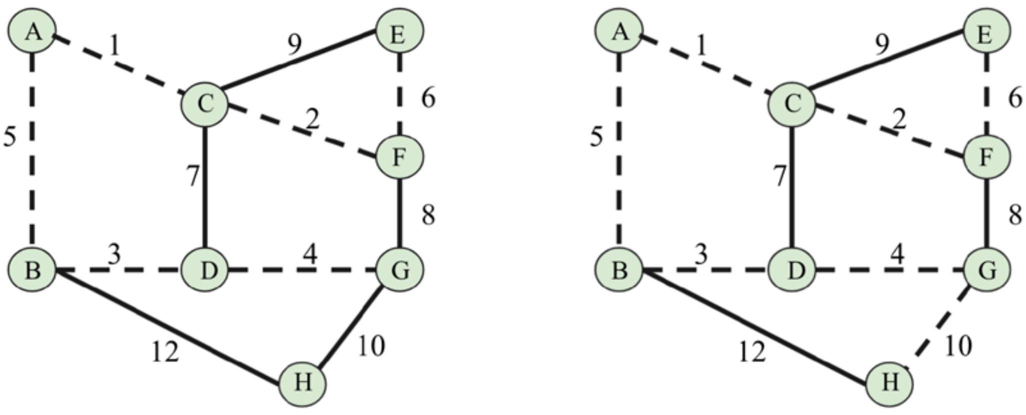


Figure 9.37: Selecting edges FE and GH in constructing the spanning tree using Prim's algorithm

Next, whenever we try to include any more edges, a cycle is formed, so we ignore those edges. Finally, we obtain the spanning tree, which is shown below in *Figure 9.38*:

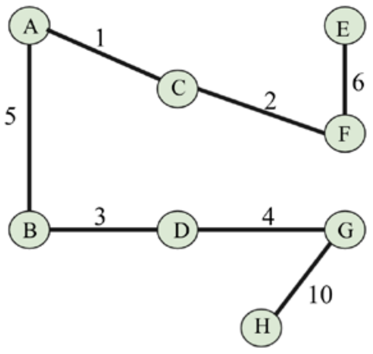


Figure 9.38: The final spanning tree using Prim's algorithm

Prim's algorithm also has many real-world applications. For all the applications where we can use Kruskal's algorithm, we can also use Prim's algorithm. Other applications include road networks, game development, etc.

Since both Kruskal's and Prim's MST algorithms are used for the same purpose, which one should be used? In general, it depends on the structure of the graph. For a graph with **C** vertices and **E** edges, Kruskal's algorithm's worst-case time complexity is O(E logV), and Prim's algorithm has a time complexity of O(E + V logV). So, we can observe that Prim's algorithm works better when we have a dense graph, whereas Kruskal's algorithm is better when we have a sparse graph.

Summary

A graph is a non-linear data structure, which is very important due to the large number of real-world applications it has. In this chapter, we have discussed different ways to represent a graph in Python, using lists and dictionaries. Further, we learned two very important graph traversal algorithms, i.e., depth-first search (DFS) and breadth-first search (BFS). Moreover, we also discussed two very important algorithms for finding an MST, i.e. Kruskal's algorithm and Prim's algorithm.

In the next chapter, we will discuss searching algorithms and the various methods using which we can efficiently search for items in lists.

Exercises

1. What is the maximum number of edges (without self-loops) possible in an undirected simple graph with five nodes?
2. What do we call a graph in which all the nodes have equal degrees?
3. Explain what cut vertices are and identify the cut vertices in the given graph:

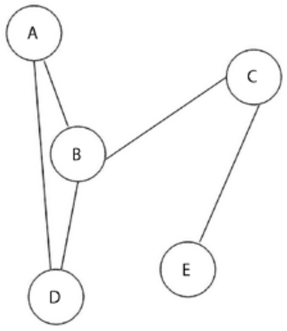


Figure 9.39: A sample graph

4. Assuming a graph G of order n, what will be the maximum number of cut vertices possible in graph G?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/MEvK4`