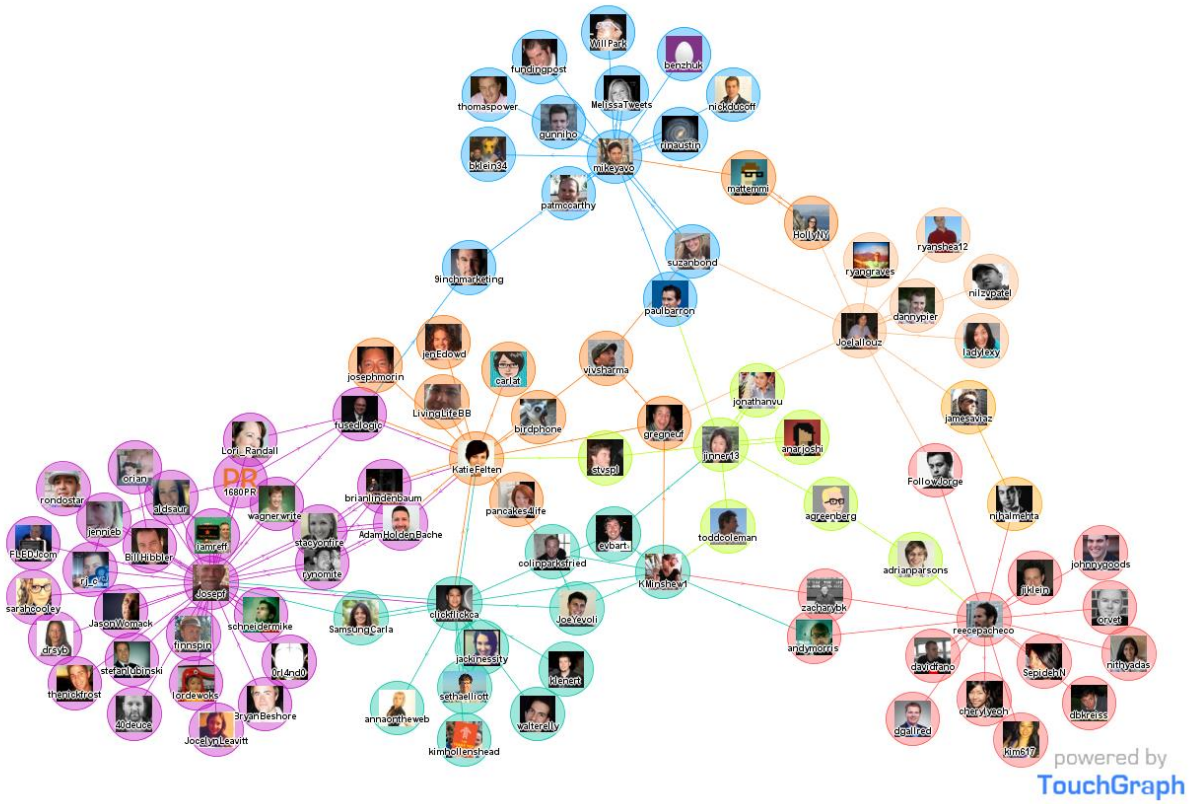
Get started with Graph Theory

Graph Theory, in essence, is the study of properties and applications of graphs or networks. As I mentioned above, this is a huge topic and the goal of this series is to gain an understanding of how to apply graph theory to solve real world problems. If we look out the premise we live in, we could see a number of problems popping out which in turn could be modeled as graphs. For example, choosing a dress combination out of all given categories could be thought of as a perfect scenario.

Photo by Author

If we choose each item from each category like one from caps and another one from T-shirts and so on, we would end up with n possible options. The constraint we have here is, each category has only a limited number of resources. Still, we would have a decent amount of combinations. In this case, we could use graphs to showcase different categories of clothing in each **node** and relationships among them via **edges**. I hope the idea behind nodes and edges is known to everyone. If no, nodes could be thought of as each item in each category, for example, the red cap is a node and the green trouser is also a node, as well. The relationship among different nodes could be depicted using edges i.e. from red cap to blue t-shirt.

Another canonical example of a graph is a social network of friends. Visualizing data into graphs allows us to generate and answer different interesting questions about the data.



Screenshot from TouchGraph

In the case of a social network, we could ask questions like how many friends does John have or how many mutual friends are there between Cathy and Milan.

Types of Graphs

There are different types of graph representations available and we have to make sure that we understand the kind of graph we are working with when programmatically solving a problem which includes graphs.

- **Undirected Graphs**

As the name shows, there won't be any specified directions between nodes. So an edge from node A to B would be **identical** to the edge from B to A.

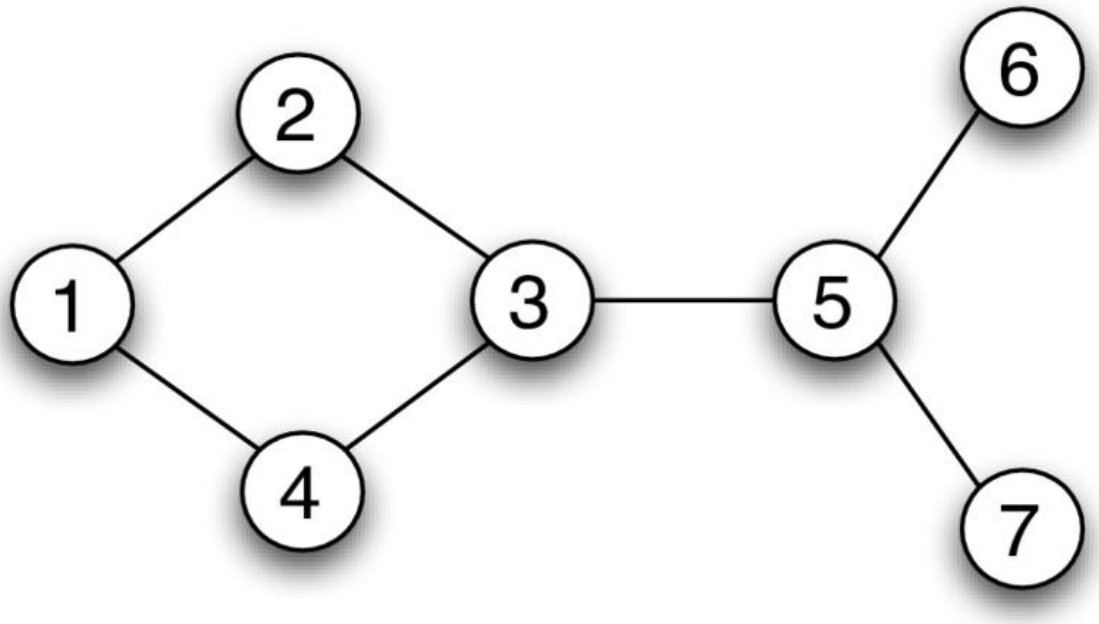
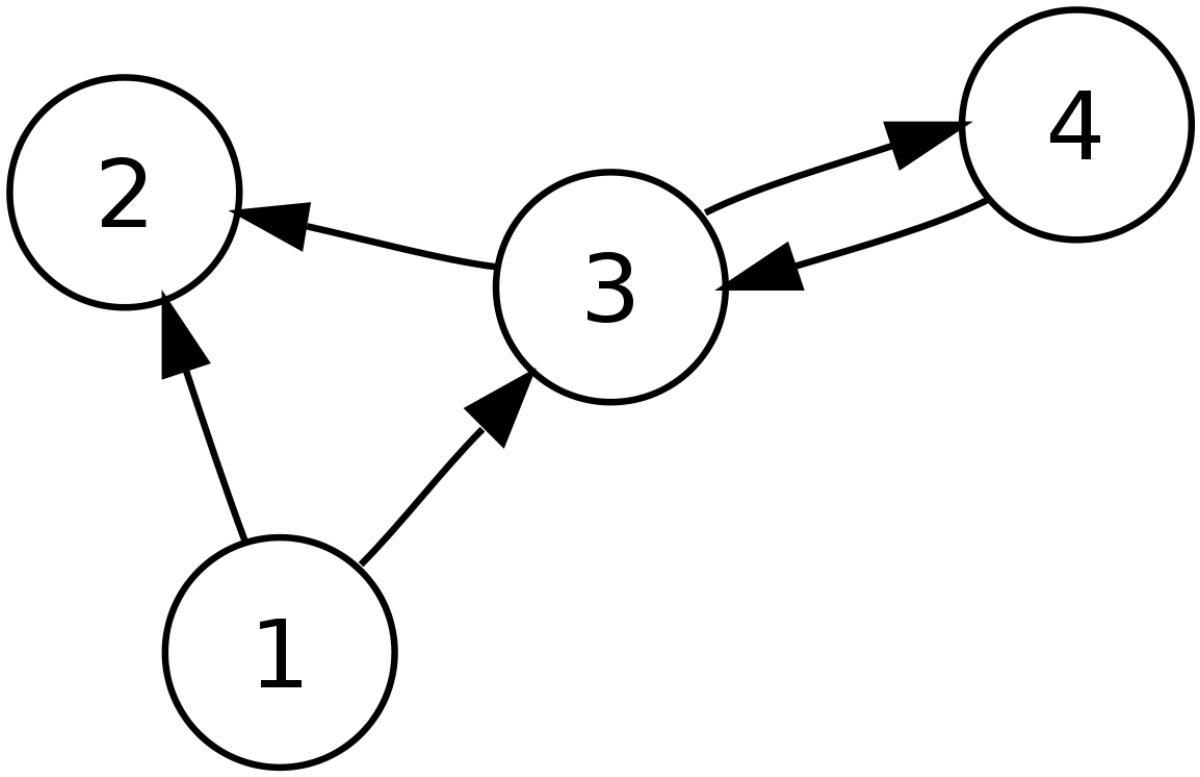


Photo Hakan Teralius on researchgate.net

In the above graph, each node could represent different cities and the edges show the bidirectional roads.

- **Directed Graphs (DiGraphs)**

Unlike undirected graphs, directed graphs have orientation or **direction** among different nodes. That means if you have an edge from node A to B, you can move only from A to B.



Screenshot from WIkiMedia

Like the previous example, if we consider nodes as cities, we have a direction from city 1 to 2. That means, you can drive from city 1

to 2 but not back to city 1, because there is no direction back to city 1 from 2. But if we closely examine the graph, we can see cities with bi-direction. For example cities 3 and 4 have directions to both sides.

- **Weighted Graphs**

Many graphs can have edges containing a weight associated to represent a real world implication such as cost, distance, quantity etc ...

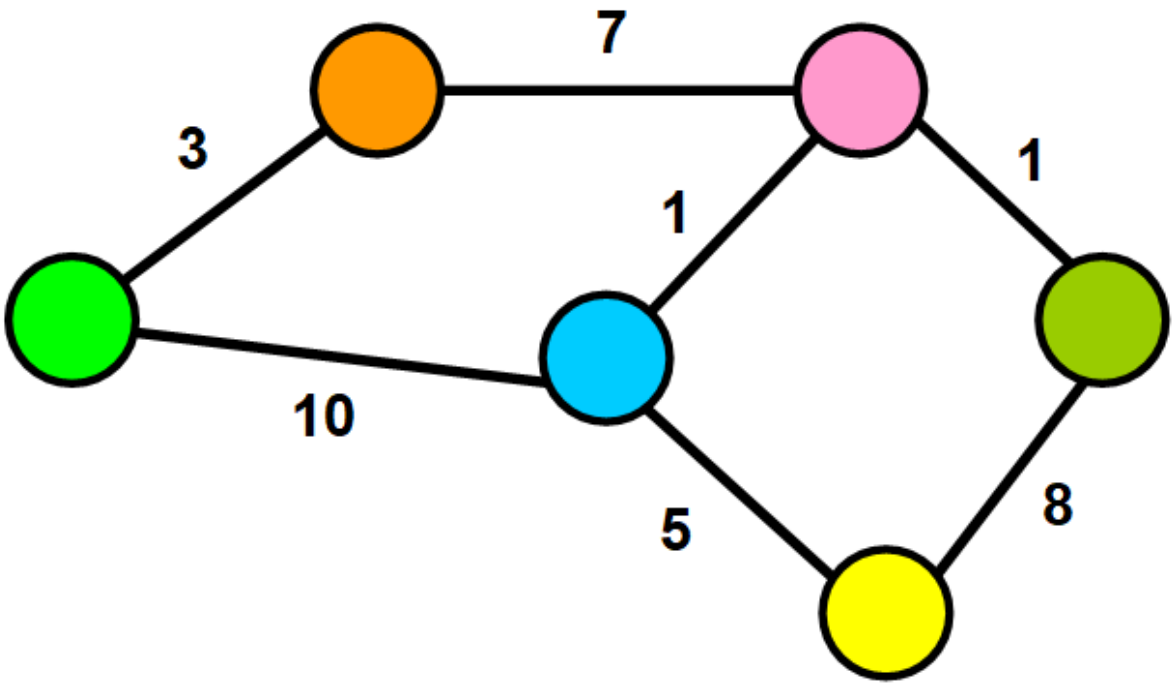


Photo by Estefania Cassingena Navone via freecodecamp.org

Weighted graphs could be either directed or undirected graphs. The one we have in this example is an undirected weighted graph. The cost or distance from node green to orange and vice versa is 3. We could represent this relationship as a triplet like **(u, v, w)** which shows from where the edge is coming in, where it goes and the cost or distance between the two nodes. Like our previous example, if you want to travel between two cities, say city green and orange, we would have to pay off a cost of 3$ or in other terms, we would have to drive 3 miles. These metrics are self-defined and could be changed according to the situations. For a more elaborated example, consider you have to travel to the city pink from green. If you look at the city graph, we can't find any direct roads or edges between the two cities. So what we can do is to travel via another city. The most promising routes would be starting from green to pink via orange and blue. If the weights are costs between cities, we would have to spend 11$ to travel via blue to reach pink but if we take the other route via orange, we would only have to pay 10$ for the trip.

Special Graphs

Apart from the above divisions, we have another set of graphs called special graphs.

- **Tree**

The most important type of special graph is a tree. It's an undirected graph with no cycles. Equivalently, it has **N** nodes and **N — 1** edges.

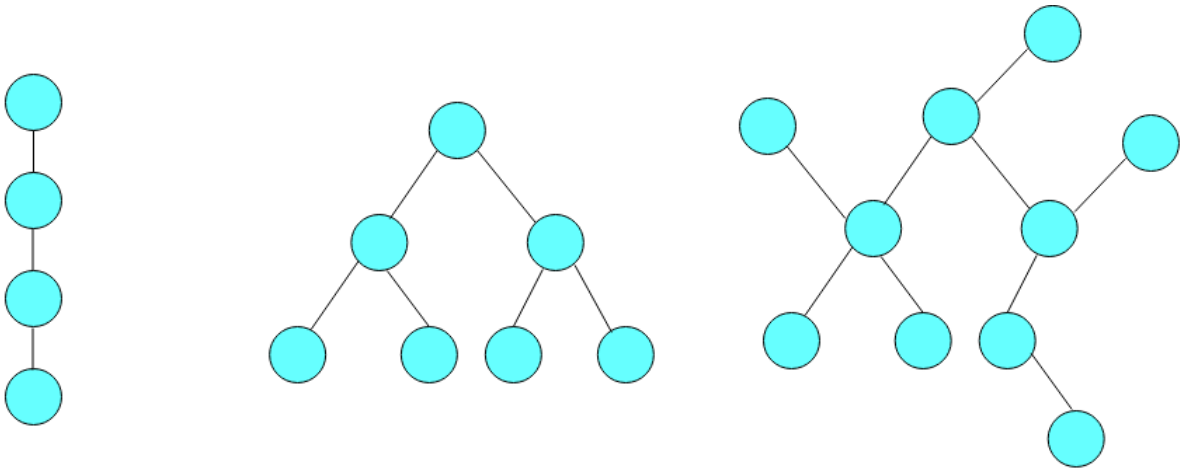


Photo by Author

All the graphs given above are trees, even the leftmost one because it has no cycles in it.

- **Rooted Tree**

A rooted tree is a tree with a designated root node where all other nodes are either coming towards the root or going away from the root.

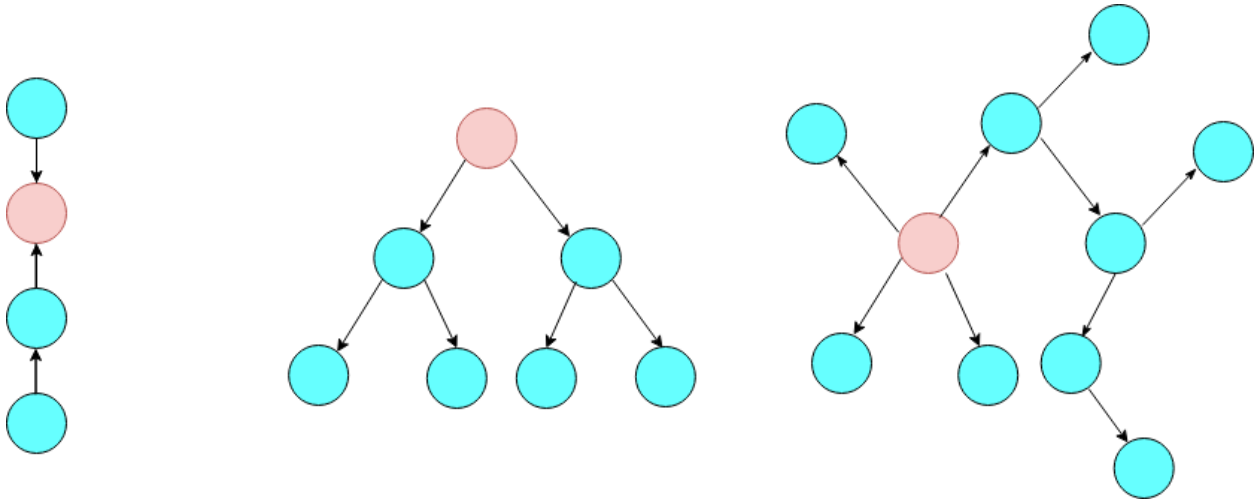


Photo by Author

The node which is in red color is the root node. The leftmost tree is called an **In-tree** because all other nodes are coming towards the root node. The two other trees are **Out-tree**s, because all other nodes are going away from the root.

- **Directed Acyclic Graphs** (**DAG**s)

DAGs are **directed graphs with no cycles**. These graphs play an important role in representing structures with dependencies such as schedulers and compilers.

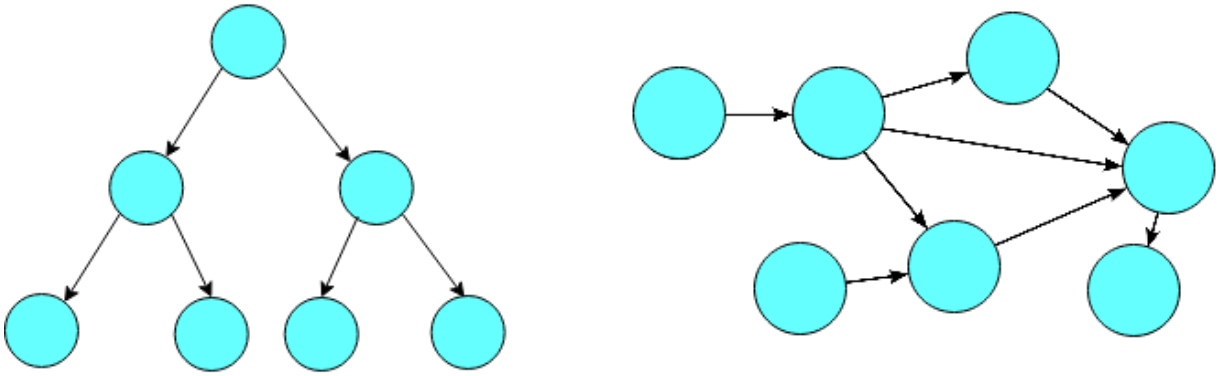
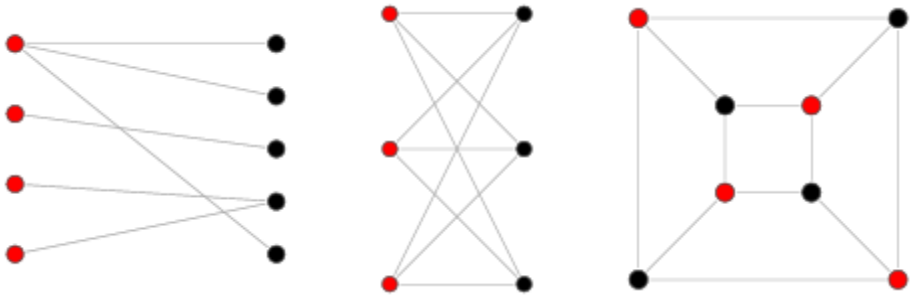


Photo by Author

We can use this graph to show the topological order between something meaningful. For example, if we use DAGs in a process manager, we could say that sub-processes x and y should be completed before proceeding with z.

- **Bi-Partite Graphs**

A Bi-Partite Graph is a one whose vertices could be divided into two disjoint sets, say U and V, where each edge in the graph connects the vertices between U and V. Another similar definition to a Bi-Partite Graph is that the graph would be two colourable.

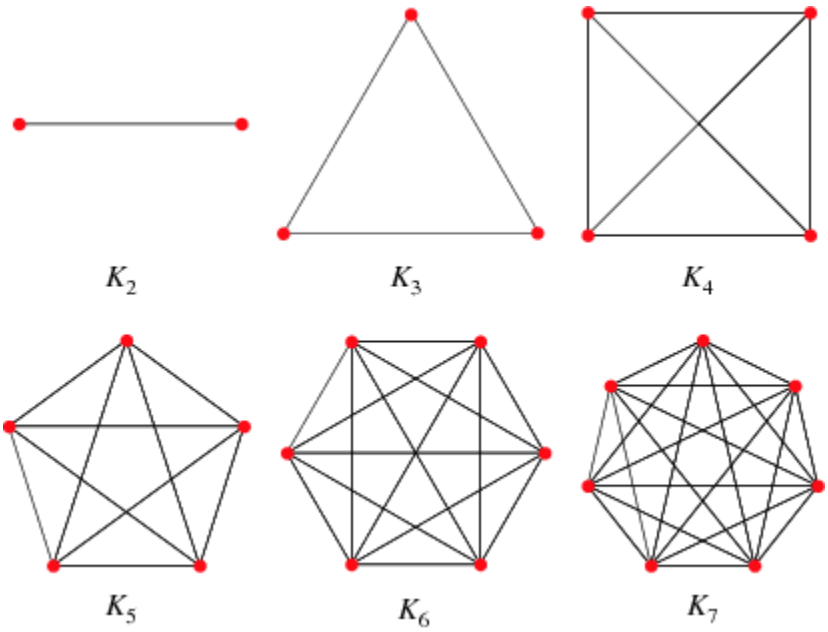


Screenshot from [Wolfram Mathworld]

If we look at the graph, we could see that each graph is divisible into two disjoint sets (U, V) and each edge connects the nodes between U and V.

- **Complete Graph**

We call a graph Complete iff, each pair of vertices has a unique edge connecting between them. A complete graph with n vertices is denoted as **Kn.**



Screenshot from [Wolfram Mathworld]

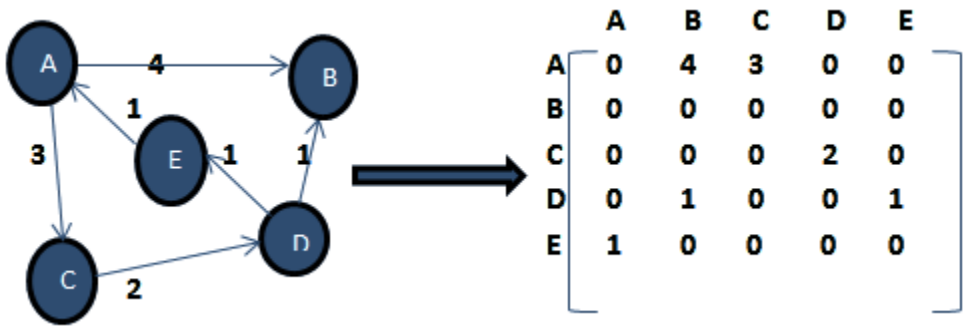
Complete graphs are considered to be the worst case graphs because of the number of edges to be traversed.

Representation of Graphs

Here we discuss how we store a graph in the memory for further processes.

- **Adjacency Matrix**

The efficient way is to use a matrix of size **NxN** where N is the number of nodes. We call this matrix an adjacency matrix.



Screenshot from softwaretestinghelp.com

The above given is of a directed weighted graph and its corresponding adjacency matrix **M**. The matrix is of size 5x5 as there are 5 nodes in total. The cost from node A to B is 4 and it's given in **M[A][B].** Similarly, the cost from one node to itself is zero, so all the diagonal elements would always be zeros. This is a space efficient method to store the graph information for the dense structures and the edge look up would always take a constant time. But, as the number of nodes increases, the space required to keep track of the edge information would also increase exponentially. If most of the edges don't have proper information, we would end up making a sparse matrix.

- **Adjacency Lists**

Another important structure we use to store the node and edge information is an adjacency list. This is a map from nodes to lists

of edges.

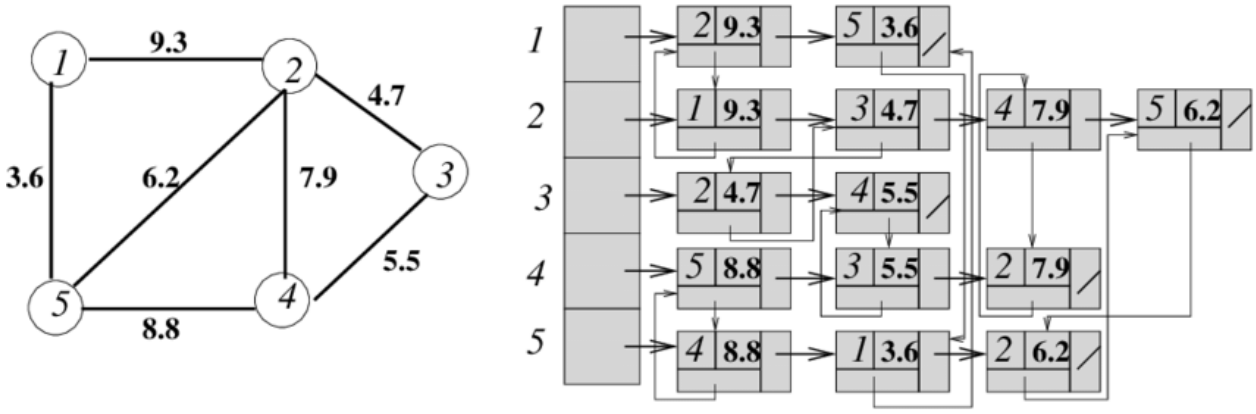
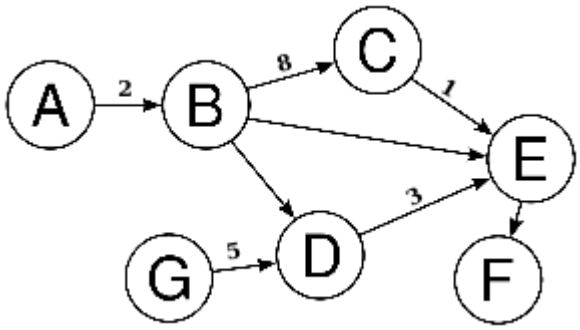


Photo by Surender Baswana via researchgate.net

The given example is of an undirected weighted graph. If we look at the rightmost diagram, we could see a couple of lists starting from each vertex. Node 1 has two outgoing edges namely 2 and 5, we represent this information along with its cost. Each element in the list has the target node and the corresponding cost or weight. Adjacency List is an efficient mechanism to store the graph information if it is sparse i.e. it would take less memory as compared to the adjacency matrix in the same scenario. But still, this is a slight more complex representation rather than the adjacency matrix.

- **Edge Lists**

AN edge list is a way to represent a graph simply as an unordered list of edges. Assume the notation for any triplet (u, v, w) means: " *the cost from u to v is w*".



[(A, B, 2), (B, C, 8), (C, E, 1),
(D, E, 3), (G, D, 5)]

Photo by Author

The edge list is given in the right hand side corresponding to the directed weighted graph on the left hand side. Each pair in the list shows edge information between two nodes and associated weight.