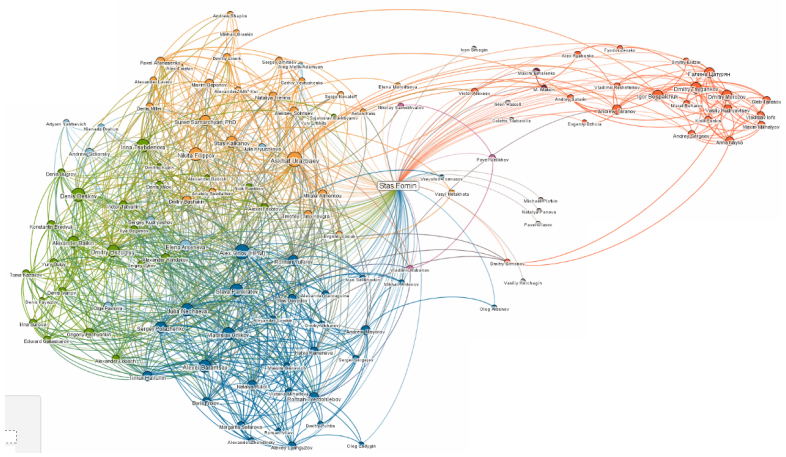
Graph Theory



(source:Daniel A. Spielman,Yale Unviersity)

Author: Vardhan Dongre, Graduate Student, University of Illinois, Urbana-Champaign

This notebook is a result of my personal interest in graph theory and the algorithms involved in solving the complex graph problems.
This work is being developed along side some coursework in graph theory that I have been taking, it includes:

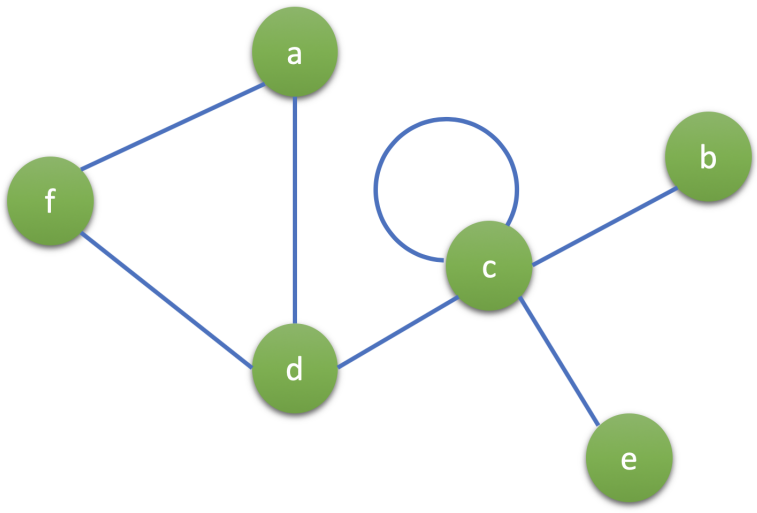
- Graph Theory Algorithms (https://www.udemy.com/course/graph-theory-algorithms/)
- Probabilistic Graphical Models (Reviewed) (https://relate.cs.illinois.edu/course/cs598ook-sp20/)

This notebook mainly contains algorithm implementations and pseudo codes. The objective of this notebook is

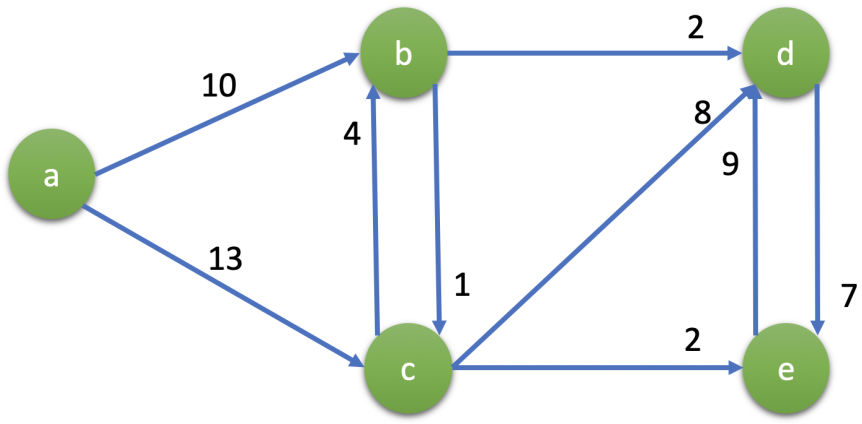
1. To develop knowledge of graph theory algorithms from a computer science perspective
2. To be used as a go to reference for problem solving
3. To help me stay organized while studying this area
4. A tool for my personal and academic development

Some of the earlier cells in this notebook contain work that is part of online/university provided resources. I have re-implemented/re-wrote some of these works after understanding from original source.

In [8]:
```
# Graph

# unweighted
graph_1 = {
    "a":["c"],
    "b":["c","e"],
    "c":["a","b","d","e"],
    "d":["c"],
    "e":["c","b"],
    "f":[]
}

# weighted and directed
graph_2 = {
    "a":{"b":10,"c":3},
    "b":{"d":2,"c":1},
    "c":{"b":4,"d":8,"e":2},
    "d":{"e":7},
    "e":{"d":9}
}

# Examples
#
# Example: 1
# g = { "a" : ["d", "f"],
#       "b" : ["c"],
#       "c" : ["b", "c", "d", "e"],
#       "d" : ["a", "c"],
#       "e" : ["c"],
#       "f" : ["d"]
#     }
```



Unweighted Graph



Weighted Graph

In [45]:
```
# generate edges
def generate_edges(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append((node,neighbour))
    return edges
print(generate_edges(graph_1))
```

```
[('a', 'c'), ('b', 'c'), ('b', 'e'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('c', 'e'), ('d', 'c'), ('e', 'c'),
('e', 'b')]
```

In [46]:
```
# Identify isolated nodes
def isolated_nodes(graph):
    isolated = []
    for node in graph:
        if not graph[node]:
            isolated.append(node)
    return isolated
```

In [47]:
```
isolated_nodes(graph_1)
```

Out[47]:
```
['f']
```

In [53]:

```
# Defining Graph Class

class Graph(object):
    def __init__(self, graph_dict=None):
        if graph_dict == None:
            graph_dict = {}
        self.__graph_dict = graph_dict

    def vertices(self):
        return list(self.__graph_dict.keys())

    def edges(self):
        return self.__generate_edges()

    def add_vertex(self, vertex):
        if vertex not in self.__graph_dict:
            self.__graph_dict[vertex] = []

    def add_edge(self, edge):
        edge = set(edge)
        (vertex1, vertex2) = tuple(edge)
        if vertex1 in self.__graph_dict:
            self.__graph_dict[vertex1].append(vertex2)
        else:
            self.__graph_dict[vertex1] = [vertex2]

    def __generate_edges(self):
        edges = []
        for node in self.__graph_dict:
            for neighbour in self.__graph_dict[node]:
                if {neighbour, node} not in edges:
                    edges.append({node,neighbour})
        return edges

    def __str__(self):
        res = "vertices: "
        for k in self.__graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res

if __name__ == "__main__":

    g = {
        "a":["d"],
        "b":["c"],
        "c":["b","c","d","e"],
        "d":["a","c"],
        "e":["c"],
        "f":[]
    }

    graph = Graph(g)

    print("Vertices of the graph: ")
    print(graph.vertices())

    print("Edges of graph: ")
    print(graph.edges())

    print("Add vertex: ")
    graph.add_vertex("z")

    print("Vertices of graph: ")
    print(graph.vertices())

    print("Add an edge")
    graph.add_edge({"a", "z"})

    print("Vertices of graph: ")
    print(graph.vertices())

    print("Edges of graph: ")
    print(graph.edges())


```

```
Vertices of the graph:
['a', 'b', 'c', 'd', 'e', 'f']
Edges of graph:
[{'a', 'd'}, {'b', 'c'}, {'c'}, {'d', 'c'}, {'e', 'c'}]
Add vertex:
Vertices of graph:
['a', 'b', 'c', 'd', 'e', 'f', 'z']
Add an edge
```

```
Vertices of graph:
['a', 'b', 'c', 'd', 'e', 'f', 'z']
Edges of graph:
[{'a', 'd'}, {'a', 'z'}, {'b', 'c'}, {'c'}, {'d', 'c'}, {'e', 'c'}]
```

Degree of a graph: The degree is an important information associated with a graph. The maximum degree of a graph G is denoted as $\Delta(G)$ and the minimum degree of the graph is denoted as $\delta(G)$. The **Handshaking Theorem** gives us the degree sum formula for a graph: $\sum_{v \in V} deg(v) = 2|E|$ which implies that the total sum of the degrees of each vertex in a graph is equal to twice the number of edges.

In [30]:

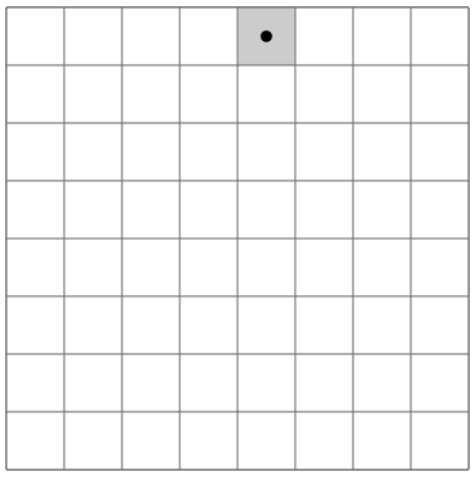
```python
# Degree of a vertex
# Degree is defined as number of edges connecting a vertex with loops counted twice

def degree(graph,vertex):
    neighbours = graph[vertex]
    return len(neighbours)+neighbours.count(vertex)

# Finding Big delta (maximum degree of a graph)

def big_delta(graph):
    max = 0
    for node in graph:
        deg = degree(graph,node)
        if deg > max:
            max = deg
    return max

def small_delta(graph):
    min = float('inf')
    for node in graph:
        deg = degree(graph,node)
        if deg < min:
            min = deg
    return min

# Degree Sequence : The sequence of vertex degrees sorted in an non-increasing order

def degree_sequence(graph):
    seq = []
    for node in graph:
        deg = degree(graph,node)
        seq.append(deg)
    seq.sort(reverse=True)
    return (seq)

# Example
example_graph = { "a" : ["d", "f"],
      "b" : ["c"],
      "c" : ["b", "c", "d", "e"],
      "d" : ["a", "c"],
      "e" : ["c"],
      "f" : ["d"]
    }
vertex = 'c'
print('The degree of vertex',vertex,'in the given graph is: ',degree(example_graph,vertex),'\n')

print('Delta(G) =', big_delta(example_graph),'\n')

print('delta(G) =',small_delta(example_graph),'\n')

print('The degree sequence is:', degree_sequence(example_graph))
```

```
The degree of vertex c in the given graph is:  5

Delta(G) = 5

delta(G) = 1

The degree sequence is: [5, 2, 2, 1, 1, 1]
```

Icosian Game (Hamiltonian Game)

In Graph Theory, a Hamiltonian path is a path in an undirecte or directe graph that visits each vertex exactly once. A graph that contains a Hamiltonian path is called a traceable graph.

Hamiltonian cycle in Dodecahedron



Kinght's tour of Chessboard

The Icosian game involves finding a Hamiltonian cycle in the edge graph of a dodecahedron. The earliest references for this problem can be traced back to 9th century in India, where Hamiltonian cycles in the knight's graph of chessboard were studied. The knight's tour is an instance of the Hamiltonian cycle. The problem of finding a Hamiltonian Path in a given graph is called Hamiltonian Path Problem and is proven to be **NP-complete**

In [ ]:
```
# Under Progress .......
```

Shortest Path Problem

In graph theory, the shortest path problem involves finding a path between two vertices of the graph such that the sum of the weights of the constituent edges is minimized. The most important algorithms for solving this problms include:

- Dijkstra's algorithms
- Bellman-Ford algorithm
- A* earch algorithm
- Floyd-Warshall algorithms
- Johnson's algorithms
- Viterbi algorithm

Each one of these algorithms solves a unique case of the shortest path problem and with vaying complexity. These algorithms are also extremely significant in some other applications such as Machine Learning models that are based on probabilistic graphical models like Hidden Markov Models, Bayesian Networks etc.

In [9]:
```
# Finding a path in a given unweighted graph

def find_path(graph, start, end, path=None):
    if path == None:
        path = []
    path = path + [start]
    if start == end:
        return path
    if start not in graph:
        return []
    for node in graph[start]:
        if node not in path:
            extended_path = find_path(graph,node,end,path)
            if extended_path:
                return extended_path
    return None

# Example
graph_a = { "a" : ["d"],
      "b" : ["c"],
      "c" : ["b", "c", "d", "e"],
      "d" : ["a", "c"],
      "e" : ["c"],
      "f" : []
    }

find_path(graph_a, 'a','b')
```

Out[9]:
```
['a', 'd', 'c', 'b']
```

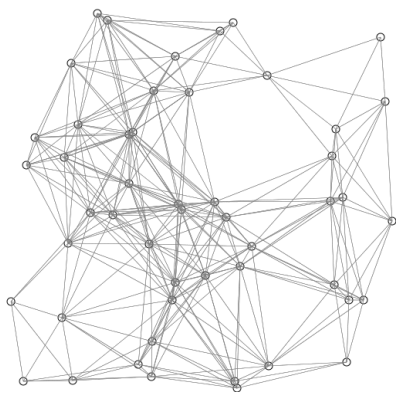
In [17]:
```python
# Finding all the possible paths between two nodes

def find_possible_paths(graph, start, end, path=None):
    if path == None:
        path = []
    path = path + [start]
    if start == end:
        return [path]
    if start not in graph:
        return []
    paths = []
    for node in graph[start]:
        if node not in path:
            extended_paths = find_possible_paths(graph, node, end, path)

            for p in extended_paths:
                paths.append(p)
    return paths


graph_b = { "a" : ["d", "f"],
      "b" : ["c"],
      "c" : ["b", "c", "d", "e"],
      "d" : ["a", "c"],
      "e" : ["c"],
      "f" : ["d"]
    }

start = 'a'
end = 'b'
print("The possible paths between",start,"&",end,'are: ')
find_possible_paths(graph_b,start,end)
```

```
The possible paths between a & b are:
```

Out[17]:
```
[['a', 'd', 'c', 'b'], ['a', 'f', 'd', 'c', 'b']]
```

Dijkstra's Algorithm

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

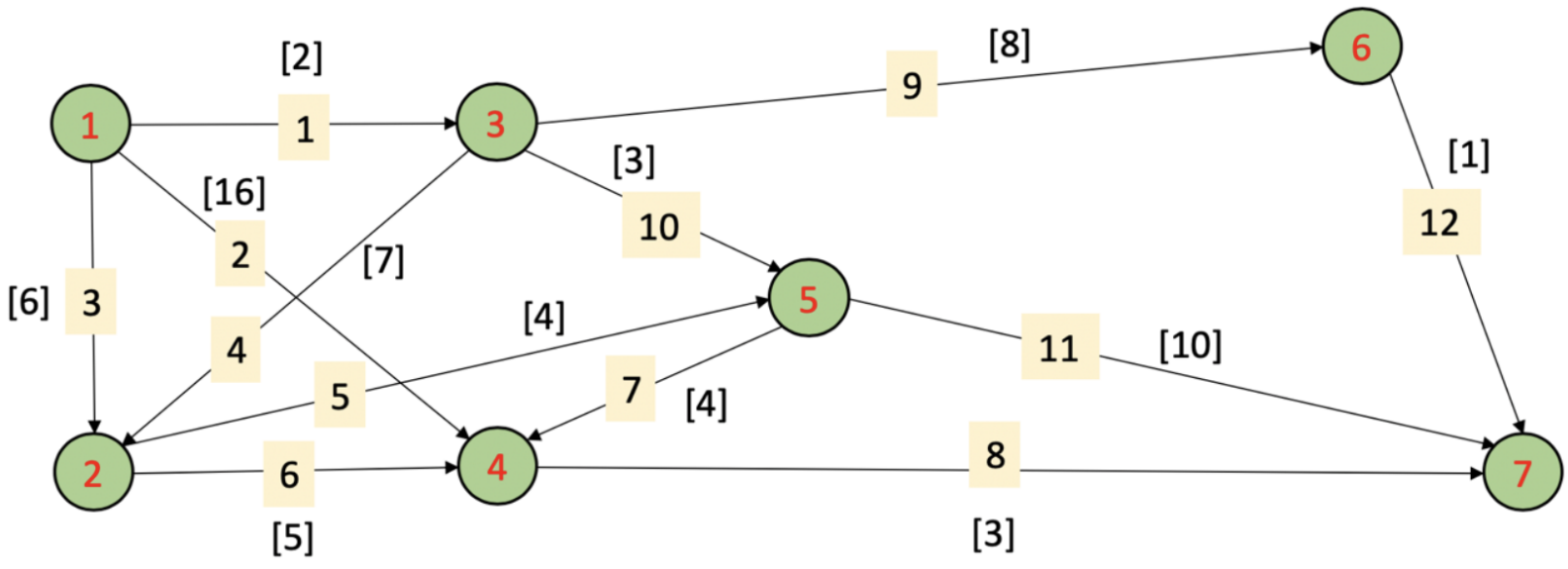


(source: Wikipedia)

In [28]:
```
# Implementing Dijkstra's Algorithm

# graph_3 = {
#     "a":{"b":10,"c":3},
#     "b":{"d":2,"c":1},
#     "c":{"b":4,"d":8,"e":2},
#     "d":{"e":7},
#     "e":{"d":9}
# }

def dijkstra(graph, source, destination):
    shortest_distance = {}
    predecessor = {}
    unseen = graph
    infinity = float('inf')
    path = []

    for vertex in unseen:
        shortest_distance[vertex] = infinity
    shortest_distance[source] = 0

    while unseen:
        minNode = None
        for node in unseen:
            if minNode is None:
                minNode = node
            elif shortest_distance[node]<shortest_distance[minNode]:
                minNode = node

        for child, weight in graph[minNode].items():
            if weight+shortest_distance[minNode]<shortest_distance[child]:
                shortest_distance[child] = weight+shortest_distance[minNode]
                predecessor[child] = minNode

        unseen.pop(minNode)

    currentNode = destination
    while currentNode != source:
        try:
            path.insert(0,currentNode)
            currentNode = predecessor[currentNode]
        except:
            print('Path cannot be formed')
            break
    path.insert(0,source)
    if shortest_distance[destination] != infinity:
        print('The shortest distance from source:',source,'to destination:',
              destination,'is:',shortest_distance[destination])
        print('The shortest path is:', path)

# Shortest Path Solution for Graph

#dijkstra(graph_3,'a','b')
```

Example

The following figure shows a network with source = 1 and destination = 7. The parameters enclosed in the parenthesis represent the length and numbers in yellow box are the link labels. Formulate the shortest path problem.



In [58]:
```
# Define the graph
g1 = {
    '1':{'2':6,'3':2,'4':16},
    '2':{'4':5},
    '3':{'2':7,'5':3,'6':8},
    '4':{'7':3},
    '5':{'4':4,'7':10},
    '6':{'7':1},
    '7':{}
}

# example_graph = Graph(g1)
# print(example_graph.vertices())
# print(example_graph.edges())

# Shortest Path and Distances
nodes = ['1','2','3','4','5','6','7']
source = nodes[0]
destination = nodes[6]
dijkstra(g1,source,destination)
```

```
The shortest distance from source: 1 to destination: 7 is: 11
The shortest path is: ['1', '3', '6', '7']
```

In [ ]:
```

```

Graph Algorithms are often complicated but very intriguing. Their visualizations often result in very astonishing and remarkable patterns. While understanding several algorithms I have observed that visulazation of a graph algorithm has helped me understand them better. Some useful resources for this are:

1. Visual Algo (https://visualgo.net/en)

In [31]:
```
# Mardown Dependencies
from IPython.display import HTML, display
```