Nama: Dewa Bagus Putu Arya Dhananjaya

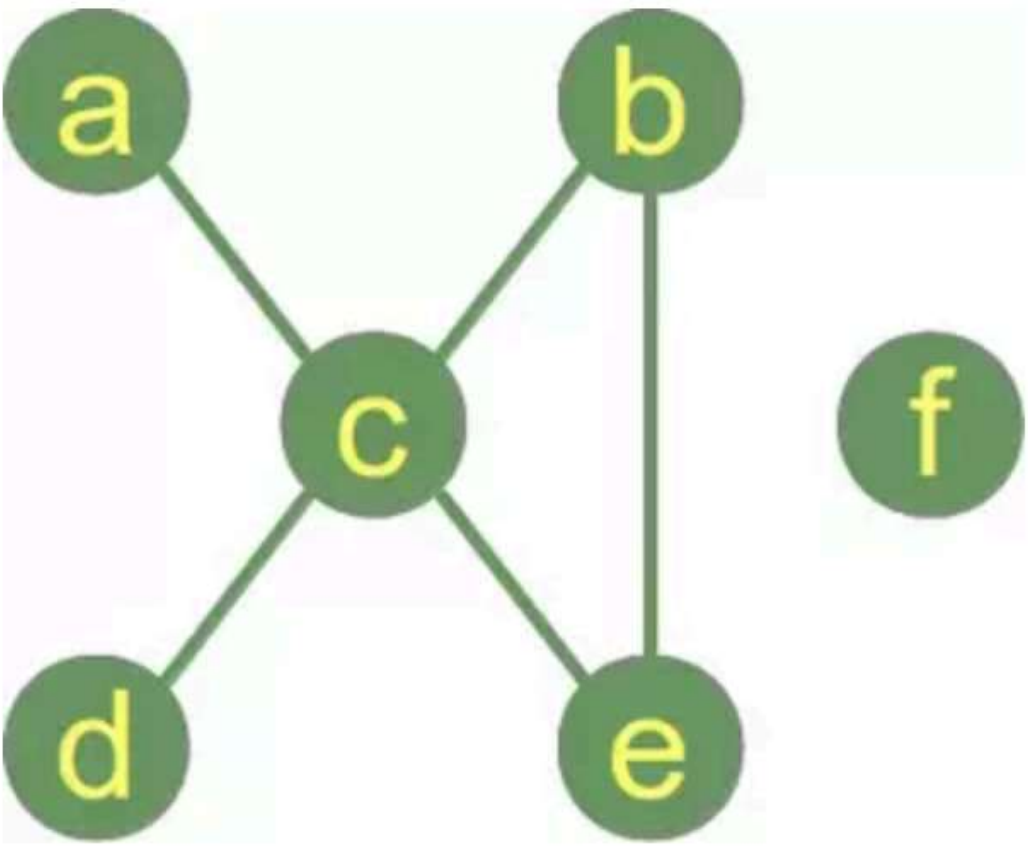
NPM: 10122362

Kelas: 3KA21

Pengenalan Teori Graf

Contoh kasus 1

Dalam contoh kasus pertama kali ini, Kita akan mencoba membangun sebuah graf dengan struktur sebagai berikut.



Dalam merepresentasikan bentuk graf tersebut, kita akan membuat 2 fungsi untuk memudah

Program dimulai dengan mendefinisikan sebuah graf sebagai struktur data berbasis dictionary. Dalam dictionary ini, setiap simpul diwakili oleh sebuah kunci (seperti "a" atau "b"), dan tetangganya (simpul yang terhubung) diwakili oleh set nilai. Misalnya, graf {"a": {"c"}, "b": {"c", "e"}, "c": {"a", "b", "d", "e"}} merepresentasikan hubungan antara simpul-simpul tersebut.

In [1]:
```
## Blok program ini Anda mencoba mendefinisikan node dan hubungan Antara node dalam bentuk obyek.
graph = { "a" : {"c"},
          "b" : {"c", "e"},
          "c" : {"a", "b", "d", "e"},
          "d" : {"c"},
          "e" : {"c", "b"},
          "f" : {}
        }
```

In [2]:
```
## Blok program ini Anda mencoba membuat EDGE dari informasi yang sudah Anda definisikan sebelumnya.
def generate_edges(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append({node, neighbour})

    return edges
```

Fungsi generate_edges(graph) dirancang untuk mengidentifikasi semua edge (sisi) dalam graf. Logika fungsi ini adalah dengan melakukan iterasi pada setiap simpul dan tetangganya, lalu menambahkan pasangan simpul tersebut ke dalam daftar edges. Pasangan simpul ini memastikan semua hubungan antar simpul terdata, meski mungkin muncul duplikasi.

In [3]:
```
print(generate_edges(graph))
```

```
[{'a', 'c'}, {'b', 'e'}, {'b', 'c'}, {'b', 'c'}, {'d', 'c'}, {'a', 'c'}, {'e', 'c'}, {'d', 'c'}, {'b', 'e'}, {'e', 'c'}]
```

In [4]:
```
## Blok program ini Anda mendefinisikan sebuah fungsi untuk mengetahui node mana yang tidak memiliki edge
def find_isolated_nodes(graph):
    """ returns a set of isolated nodes. """
    isolated = set()
    for node in graph:
        if not graph[node]:
            isolated.add(node)
    return isolated
```

Fungsi find_isolated_nodes(graph) bertujuan untuk menemukan simpul yang tidak memiliki koneksi (isolasi). Fungsi ini memeriksa setiap simpul di graf, dan jika sebuah simpul tidak memiliki tetangga, maka simpul tersebut akan dimasukkan ke dalam daftar simpul terisolasi.

In [5]:
```
print(find_isolated_nodes(graph))
```

```
{'f'}
```

Contoh kasus 2

Dalam contoh kasus kedua kali ini, Kita akan mencoba membangun sebuah graf dan mendapatkan informasi lebih detail mengenai graf tersebut. Untuk memudahkan, Kita akan mendefinisikan sebuah Class dalam bahasa pemrograman python, dimana Class ini berisi beberapa fungsi yang bisa kita gunakan untuk mengetahui informasi detail mengenai stuktur dari graf yang kita miliki

Dalam kasus kedua, program memperkenalkan konsep kelas (Graph) untuk meningkatkan fleksibilitas dalam mengelola graf. Kelas ini memiliki fungsi seperti add_vertex untuk menambah simpul baru, add_edge untuk menambah sisi baru, serta fungsi lainnya seperti all_vertices dan all_edges untuk mendapatkan daftar simpul dan sisi yang ada.

In [6]:
```
""" A Python Class
A simple Python graph class, demonstrating the essential
facts and functionalities of graphs.
"""


class Graph(object):

    def __init__(self, graph_dict=None):
        """ initializes a graph object
            If no dictionary or None is given,
            an empty dictionary will be used
        """
        if graph_dict == None:
            graph_dict = {}
        self._graph_dict = graph_dict

    def edges(self, vertice):
        """ returns a list of all the edges of a vertice"""
        return self._graph_dict[vertice]

    def all_vertices(self):
        """ returns the vertices of a graph as a set """
        return set(self._graph_dict.keys())

    def all_edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def add_vertex(self, vertex):
        """ If the vertex "vertex" is not in
            self._graph_dict, a key "vertex" with an empty
            list as a value is added to the dictionary.
            Otherwise nothing has to be done.
        """
        if vertex not in self._graph_dict:
            self._graph_dict[vertex] = []

    def add_edge(self, edge):
        """ assumes that edge is of type set, tuple or list;
            between two vertices can be multiple edges!
        """
        edge = set(edge)
        vertex1, vertex2 = tuple(edge)
        for x, y in [(vertex1, vertex2), (vertex2, vertex1)]:
            if x in self._graph_dict:
                self._graph_dict[x].add(y)
            else:
                self._graph_dict[x] = [y]

    def __generate_edges(self):
        """ A static method generating the edges of the
            graph "graph". Edges are represented as sets
            with one (a loop back to the vertex) or two
            vertices
        """
        edges = []
        for vertex in self._graph_dict:
            for neighbour in self._graph_dict[vertex]:
                if {neighbour, vertex} not in edges:
                    edges.append({vertex, neighbour})
        return edges

    def __iter__(self):
        self._iter_obj = iter(self._graph_dict)
        return self._iter_obj
```

```
    def __next__(self):
        """ allows us to iterate over the vertices """
        return next(self._iter_obj)

    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
```

In [7]:
```
g = { "a" : {"d"},
      "b" : {"c"},
      "c" : {"b", "c", "d", "e"},
      "d" : {"a", "c"},
      "e" : {"c"},
      "f" : {}
    }
```

In [8]:
```
graph = Graph(g)

for vertice in graph:
    print("Edges of vertice {vertice}: ", graph.edges(vertice))
```

```
Edges of vertice {vertice}:  {'d'}
Edges of vertice {vertice}:  {'c'}
Edges of vertice {vertice}:  {'b', 'd', 'c', 'e'}
Edges of vertice {vertice}:  {'a', 'c'}
Edges of vertice {vertice}:  {'c'}
Edges of vertice {vertice}:  {}
```

In [9]:
```
graph.add_edge({"ab", "fg"})
graph.add_edge({"xyz", "bla"})
```

In [10]:
```
print("")
print("Vertices of graph:")
print(graph.all_vertices())

print("Edges of graph:")
print(graph.all_edges())
```

```
Vertices of graph:
{'f', 'd', 'fg', 'b', 'ab', 'e', 'xyz', 'a', 'bla', 'c'}
Edges of graph:
[{'d', 'a'}, {'b', 'c'}, {'d', 'c'}, {'c'}, {'e', 'c'}, {'ab', 'fg'}, {'xyz', 'bla'}]
```

Contoh kasus 3

Dalam contoh kasus ketiga kali ini, Kita akan mencoba membangun sebuah graf dengan menggunakan library networkx dan matplotlib

Pada kasus 3, program menggunakan pustaka NetworkX untuk membuat graf dan menambahkan simpul serta sisi secara dinamis. Kemudian, pustaka Matplotlib digunakan untuk menggambar graf dalam berbagai tata letak (layout), seperti circular, planar, dan andom. Ini memberikan tampilan visual graf yang lebih mudah dipahami.

In [11]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

In [12]:
```
# Creating a Graph
G = nx.Graph() # Right now G is empty
```

In [13]:
```
G.add_node(1)
G.add_nodes_from([2,3])
```

In [14]:
```
G.add_edge(1,2)
```

In [15]:
```
e = (2,3)
G.add_edge(*e) # * unpacks the tuple
G.add_edges_from([(1,2), (1,3)])
```

In [16]:
```
G.nodes()
```

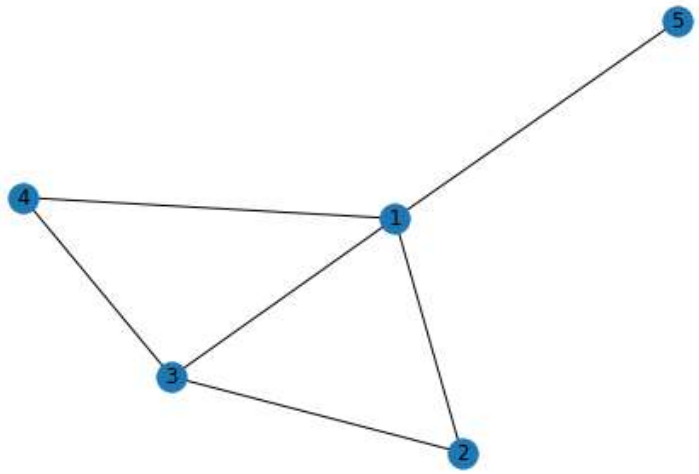
Out[16]:
```
NodeView((1, 2, 3))
```

In [17]:
```
G.edges()
```

Out[17]:
```
EdgeView([(1, 2), (1, 3), (2, 3)])
```

In [18]:
```
G.add_edge(1, 2)
G.add_edge(2, 3)
G.add_edge(3, 4)
G.add_edge(1, 4)
G.add_edge(1, 5)
```

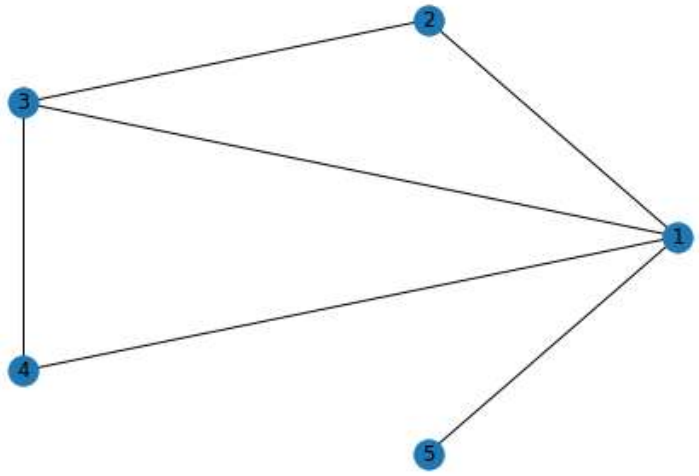
In [19]:
```
nx.draw(G, with_labels = True)
```



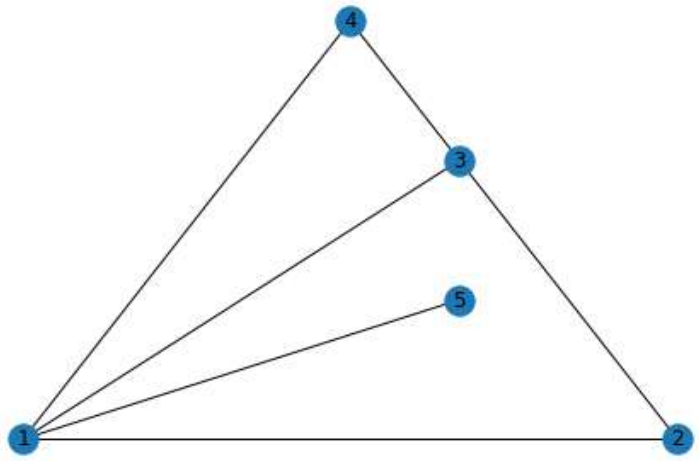
In [20]:
```
plt.savefig("contoh-graf-1.png")
```

```
<Figure size 432x288 with 0 Axes>
```

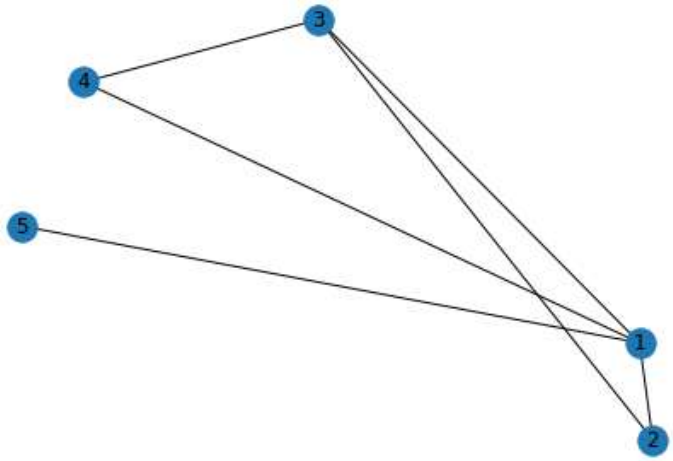
In [21]:
```
# drawing in circular layout
nx.draw_circular(G, with_labels = True)
```



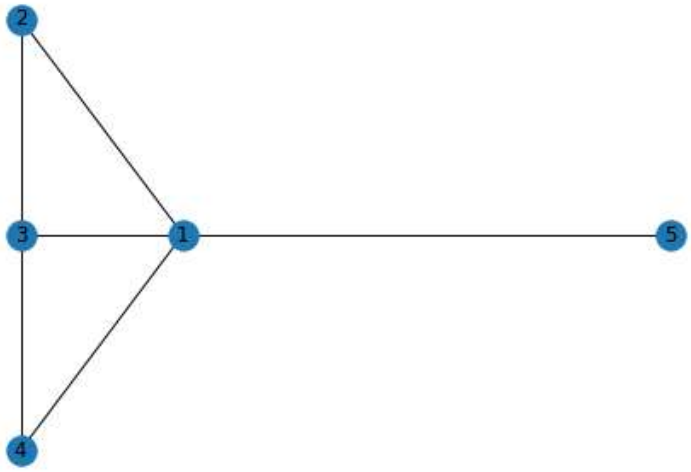
In [22]:
```
# drawing in planar layout
nx.draw_planar(G, with_labels = True)
```



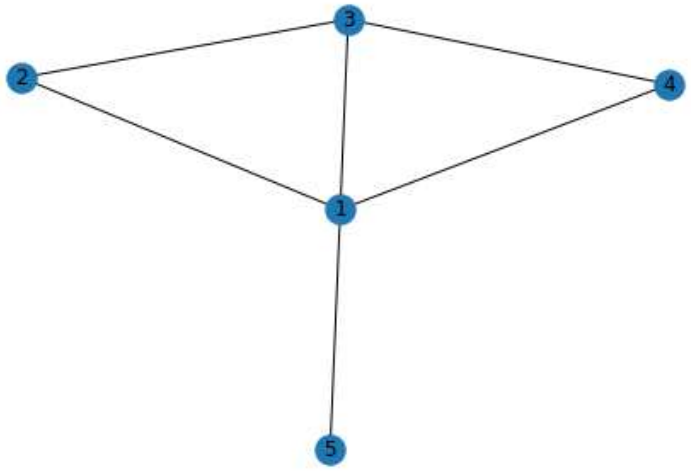
In [23]:
```
# drawing in random layout
nx.draw_random(G, with_labels = True)
```



In [24]:
```
# drawing in spectral layout
nx.draw_spectral(G, with_labels = True)
```



In [25]:
```
# drawing in spring layout
nx.draw_spring(G, with_labels = True)
```



In [26]:
```
# drawing in shell layout
nx.draw_shell(G, with_labels = True)
```