グラフ理論

グラフとはデータ構造の一つである。

グラフ理論では、点（節点またはノード）(Vertex or Vertices or Node)と線（辺）(Edge(s))の有限集合で記述する。

辺とノードの組合せでネットワークグラフを作成する

ノードまたは点の集合$V(G)$

辺の集合$E(G)$

In [4]:
```
# 隣接行列を作る関数
def graph_matrix(Edges,nodes=6):
    V,E=nodes,len(Edges)
    adj_matrix=[[0]*V for i in range(V)] #nodes数の全てゼロの行列を作成
    for i in range(E):# 隣接した点だけ、重みwを代入する
        u,v,w=Edges[i] # 各々辺の(a,b,c)のタプルをu,v,wに代入する
        adj_matrix[v-1][u-1]=adj_matrix[u-1][v-1]=w
    return adj_matrix

edges = [(1,2,1),(1,5,1),(2,3,1),(2,5,1),(3,4,1),(4,5,1),
         (4,6,1)]
```

In [5]:
```
%matplotlib inline
```

In [6]:
```
edges
```

Out[6]:
```
[(1, 2, 1), (1, 5, 1), (2, 3, 1), (2, 5, 1), (3, 4, 1), (4, 5, 1), (4, 6, 1)]
```

In [7]:
```
g_matrix=graph_matrix(edges)
print("隣接行列：")
for each in g_matrix:
    print(each)
```

```
隣接行列：
[0, 1, 0, 0, 1, 0]
[1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 0]
[0, 0, 1, 0, 1, 1]
[1, 1, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0]
```

無効グラフの実装

In [8]:
```
simple_edges=[(1,2),(1,5),(2,3),(2,5),(3,4),(4,5),(4,6)]
```

In [9]:
```
# グラフライブラリを使用する
import networkx as nx
```

In [10]:
```
# グラフデータを入力し、gに格納する
g = nx.Graph(simple_edges)
```

In [11]:
```
# グラフの節点（ノード）のリスト（集合）nodes or vertices (vertex)
print(g.nodes())
```

```
[1, 2, 5, 3, 4, 6]
```

In [12]:
```
# グラフの辺のリスト（集合）edges
print(g.edges())
```

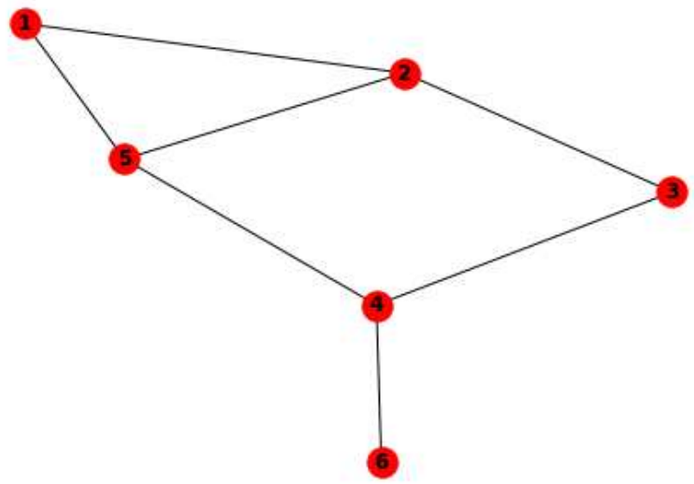
```
[(1, 2), (1, 5), (2, 3), (2, 5), (5, 4), (3, 4), (4, 6)]
```

In [13]:
```
# グラフの隣接行列 adjacent matrix : それ以下がゼロの値になる
print(nx.adjacency_matrix(g))
```

```
  (0, 1)        1
  (0, 2)        1
  (1, 0)        1
  (1, 2)        1
  (1, 3)        1
  (2, 0)        1
  (2, 1)        1
  (2, 4)        1
  (3, 1)        1
  (3, 4)        1
  (4, 2)        1
  (4, 3)        1
  (4, 5)        1
  (5, 4)        1
```

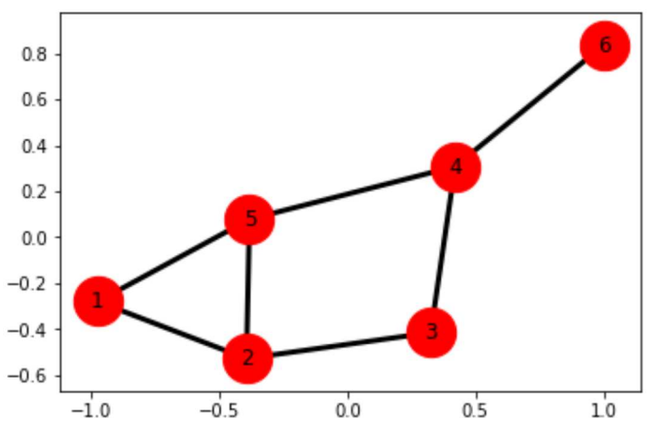
In [14]:
```
# グラフの図
nx.draw(g, with_labels=True, font_weight='bold')
```

```
D:\Anaconda3\lib\site-packages\networkx\drawing\nx_pylab.py:611: MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```



In [15]:
```
# グラフの図を拡張する
options = {
    'node_color': 'red',
    'node_size': 800,
    'width': 3,
    'arrowstyle': '-|>',
    'arrowsize': 12,
}
```

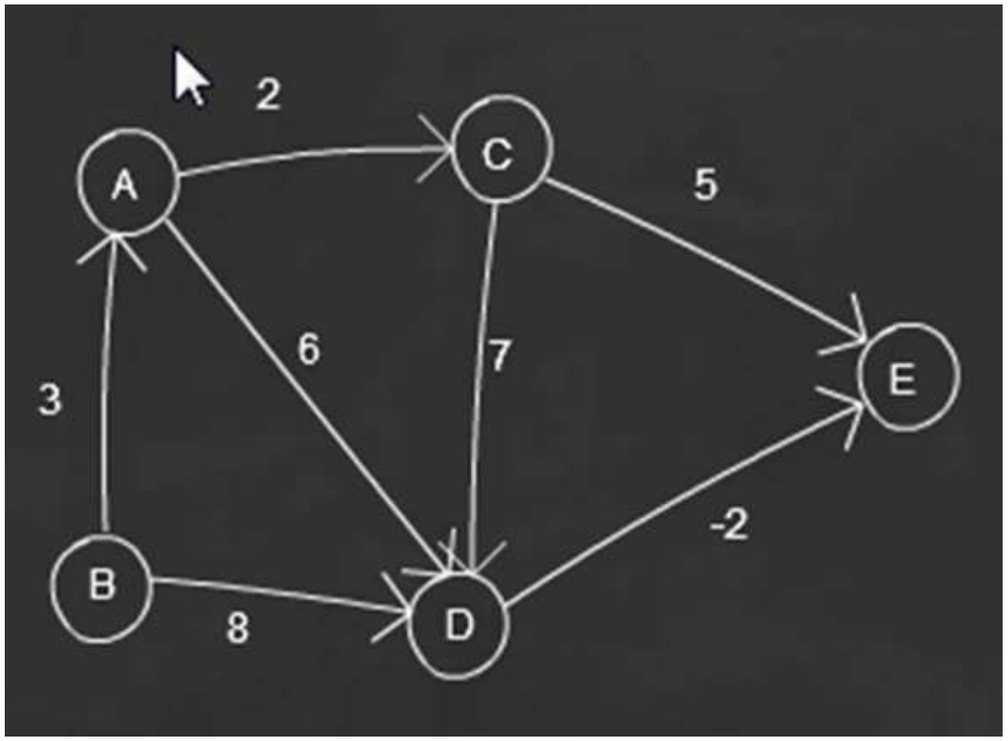
In [16]:
```
nx.draw_networkx(g, arrows=True, **options)
```



重み付け有効グラフの実装

In [17]:
```
from IPython.display import Image
Image('wg.png')
```

Out[17]:



In [18]:
```
L={'A':{'C':2,'D':6},'B':{'D':8,'A':3},
   'C':{'D':7,'E':5},'D':{'E':-2},'E':{}}

class Graph:
    def __init__(self,g):
        self.g=g
    def Vertex(self):
        return self.g.keys()
    def Adj(self,v):
        return self.g[v].keys()
    def w(self,u,v):
        return self.g[u][v]

G=Graph(L)

print("グラフのポインタ：%s" % (G))
print ("グラフの全体：%s"% (G.g))
print ("ノードの集合：%s" % (G.Vertex()))
print ("ノード'D'に隣接するノードのリスト:%s"%(G.Adj('D')))
```

```
グラフのポインタ：<__main__.Graph object at 0x000002290E0A4E80>
グラフの全体：{'A': {'C': 2, 'D': 6}, 'B': {'D': 8, 'A': 3}, 'C': {'D': 7, 'E': 5}, 'D': {'E': -2}, 'E': {}}
ノードの集合：dict_keys(['A', 'B', 'C', 'D', 'E'])
ノード'D'に隣接するノードのリスト:dict_keys(['E'])
```

In [19]:
```
nodes = list(G.Vertex())
```

In [20]:
```
nodes
```

Out[20]:
```
['A', 'B', 'C', 'D', 'E']
```

In [21]:
```
L_edges =[]
for node in nodes:
    adj_nodes = list(G.Adj(node))
    adj_edges = [(node,some_node) for some_node in adj_nodes]
    L_edges += adj_edges
L_edges
```

Out[21]:
```
[('A', 'C'),
 ('A', 'D'),
 ('B', 'D'),
 ('B', 'A'),
 ('C', 'D'),
 ('C', 'E'),
 ('D', 'E')]
```

In [22]:
```
L_edges_w=[]
for each in L_edges:
    u,v = each
    L_edges_w.append((u,v,G.w(u,v)))
    #print(G.w(u,v), end=' ')
L_edges_w
```

Out[22]:
```
[('A', 'C', 2),
 ('A', 'D', 6),
 ('B', 'D', 8),
 ('B', 'A', 3),
 ('C', 'D', 7),
 ('C', 'E', 5),
 ('D', 'E', -2)]
```

In [23]:
```
wg = nx.DiGraph(directed=True)
```

In [24]:
```
for each in L_edges_w:
    u,v,w = each
    edge = [(u,v)]
    wg.add_edges_from(edge,weight=w)
```

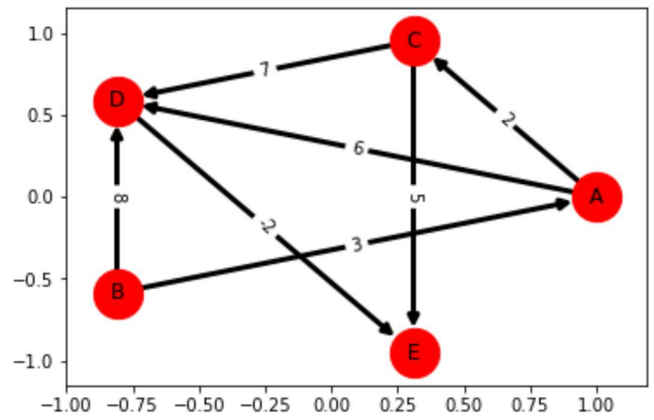
In [25]:
```
import matplotlib.pyplot as plt
limits = plt.axis('off')  # turn of axis
```

In [26]:
```
edge_labels=dict([((u,v,),d['weight'])
                  for u,v,d in wg.edges(data=True)])
```

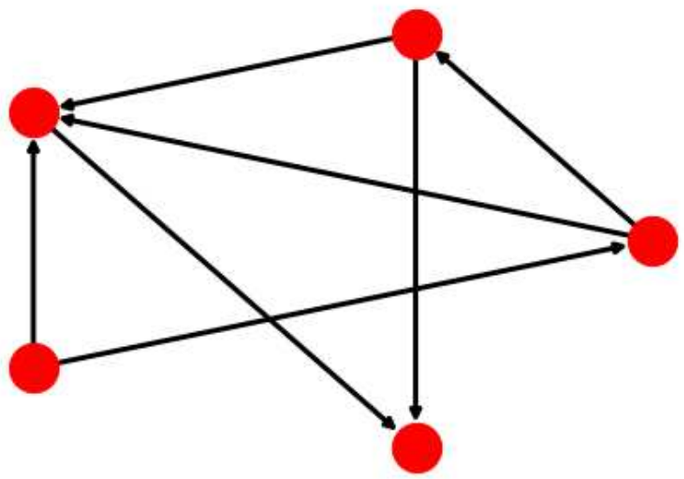
In [27]:
```
edge_labels
```

Out[27]:
```
{('A', 'C'): 2,
 ('A', 'D'): 6,
 ('C', 'D'): 7,
 ('C', 'E'): 5,
 ('D', 'E'): -2,
 ('B', 'D'): 8,
 ('B', 'A'): 3}
```

In [28]:
```
pos=nx.circular_layout(wg)
nx.draw_networkx_edge_labels(wg,pos,edge_labels=edge_labels, arrows=True, **options)
nx.draw_networkx(wg, pos, arrows=True, **options)
```



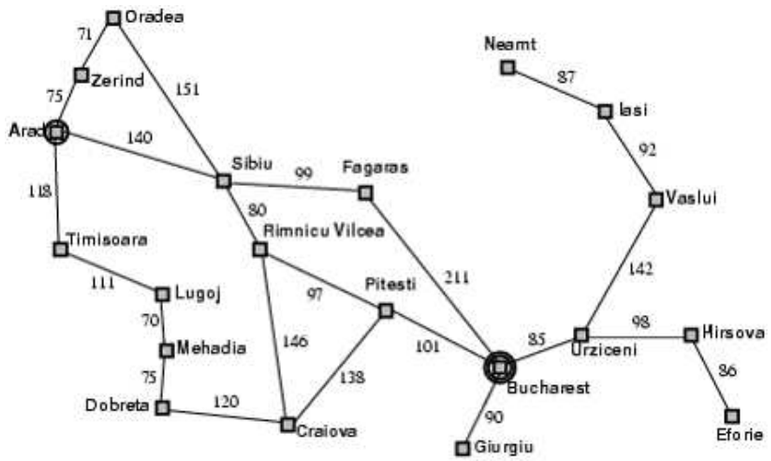
In [29]:
```
nx.draw(wg, pos, arrows=True, **options)
```



In [30]:
```
# ルーマニア（東ヨーロッパ）の路線地図からなるグラフ  (Romania Graph)

#町名とBucharestまでの直線距離 (Heuristics data)
Cities = [('Arad',366),('Zerind',374),('Timisoara',329),
          ('Sibiu',253),('Oradea',380),('Lugoj',244),('Fagaras',176),
          ('Rimniu Vilcea',193),('Mehadia',241),('Pitesti',101),
          ('Dobreta',242),('Craiova',160),('Bucharest',0),
          ('Giurgiu',77),('Urziceni',80),('Hirsova',151),
          ('Eforie',161),('Vasliu',199),('Iasi',226),('Neamt',234)]
```

In [31]:
```
Image('romania.png')
```

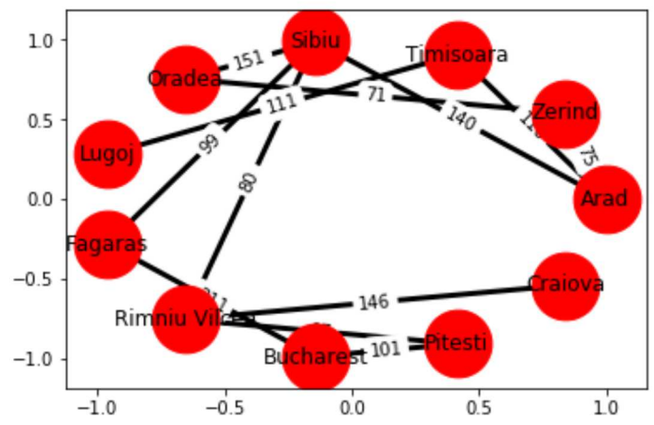
Out[31]:



In [52]:
```
romania_g = nx.Graph(directed=False)

# 町がノードになる
nodes = []
for city,h in Cities:
    nodes.append(city)


#隣接する町は辺を作る
#def city_edges(nodes,romania_g):
romania_g.add_edges_from([(nodes[0],nodes[1])],weight=75)
romania_g.add_edges_from([(nodes[0],nodes[2])],weight=118)
romania_g.add_edges_from([(nodes[0],nodes[3])],weight=140)
romania_g.add_edges_from([(nodes[1],nodes[4])],weight=71)

romania_g.add_edges_from([(nodes[4],nodes[3])],weight=151)
romania_g.add_edges_from([(nodes[2],nodes[5])],weight=111)
romania_g.add_edges_from([(nodes[3],nodes[6])],weight=99)
romania_g.add_edges_from([(nodes[3],nodes[7])],weight=80)

romania_g.add_edges_from([(nodes[6],nodes[12])],weight=211)
romania_g.add_edges_from([(nodes[7],nodes[9])],weight=97)
romania_g.add_edges_from([(nodes[7],nodes[11])],weight=146)
romania_g.add_edges_from([(nodes[9],nodes[12])],weight=101)

# 完全なグラフではない (not complete)
```

In [53]:
```
edge_labels=dict([((u,v,),d['weight'])
                  for u,v,d in romania_g.edges(data=True)])
```

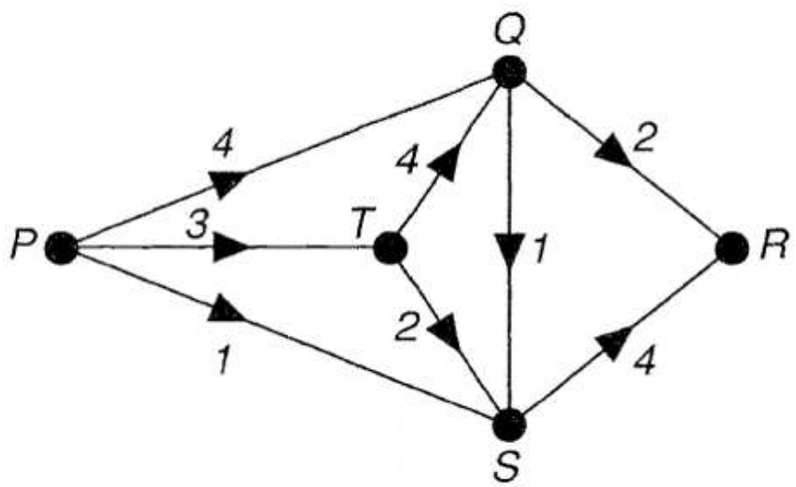
In [56]:
```
options = {
    'node_color': 'red',
    'node_size': 1500,
    'width': 3
}
pos=nx.circular_layout(romania_g)
nx.draw_networkx_edge_labels(romania_g,pos,edge_labels=edge_labels, **options)
nx.draw_networkx(romania_g, pos, **options)
```



重み付きグラフ3

In [35]:
```
Image('g1.png')
```

Out[35]:



In [42]:
```
L={'P':{'Q':4,'S':1,'T':3},'T':{'Q':4,'S':2},
   'Q':{'S':1,'R':2},'S':{'R':4}}
```

In [43]:
```
L
```

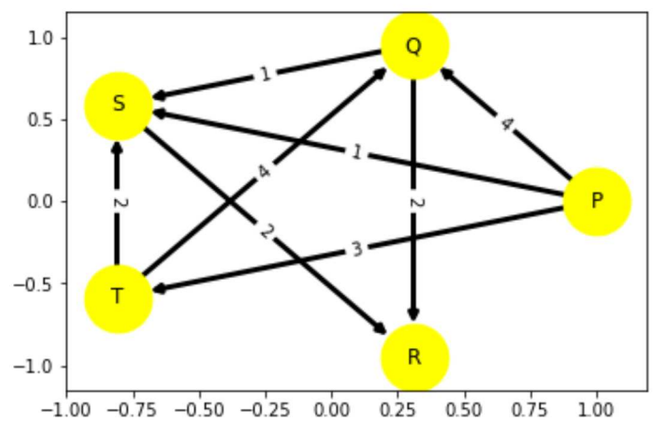
Out[43]:
```
{'P': {'Q': 4, 'S': 1, 'T': 3},
 'T': {'Q': 4, 'S': 2},
 'Q': {'S': 1, 'R': 2},
 'S': {'R': 4}}
```

In [57]:
```
wg = nx.DiGraph(directed=True)
```

In [58]:
```
wg.add_edges_from([('P','Q')],weight=4)
wg.add_edges_from([('P','S')],weight=1)
wg.add_edges_from([('P','T')],weight=3)
wg.add_edges_from([('T','Q')],weight=4)
wg.add_edges_from([('T','S')],weight=2)
wg.add_edges_from([('Q','S')],weight=1)
wg.add_edges_from([('Q','R')],weight=2)
wg.add_edges_from([('S','R')],weight=2)
```

In [59]:
```
edge_labels=dict([((u,v,),d['weight'])
                  for u,v,d in wg.edges(data=True)])
```

In [60]:
```
options = {
    'node_color': 'yellow',
    'node_size': 1500,
    'width': 3
}
pos=nx.circular_layout(wg) #nx.random_layout, nx.spectral_layoutなどもあります
nx.draw_networkx_edge_labels(wg,pos,edge_labels=edge_labels, **options)
nx.draw_networkx(wg, pos, **options)
```



In [ ]: