グラフ理論

ヒューリスティック無しグラフ探索(1)：木の変換から幅優先探索と深さ優先探索、

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$E(G)$

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$E(G)$

```
%matplotlib inline
```

無効グラフの実装

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

```
# グラフライブラリを使用する
import networkx as nx
```

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

```
print(g.nodes())
```

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

```
print(g.edges())
```

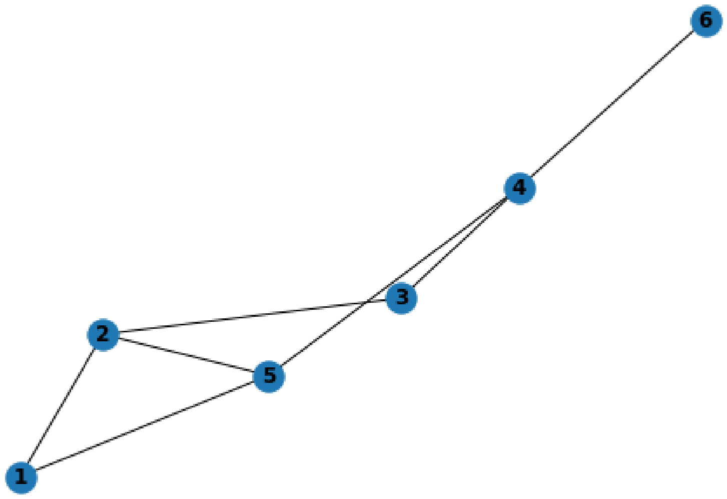
```
    [(1, 2), (1, 5), (2, 3), (2, 5), (5, 4), (3, 4), (4, 6)]
```

```
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
```

```
nx.draw(g, with_labels=True, font_weight='bold')
```



```
nx.is_tree(g)
```

```
False
```

```
for path in nx.all_simple_paths(g, source=1, target=6):
    print(path)
```

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

```
paths = list(nx.shortest_simple_paths(g, 1, 6))
```

```
print(paths)
```

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

```
from itertools import islice
def k_shortest_paths(G, source, target, k, weight=None):
    return list(islice(nx.shortest_simple_paths(G, source, target, weight=weight), k))
for path in k_shortest_paths(g, 1, 6, 2):
     print(path)
```

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

```
Tree_bfs = nx.bfs_edges(g,1)
```

```
print(list(Tree_bfs))
```

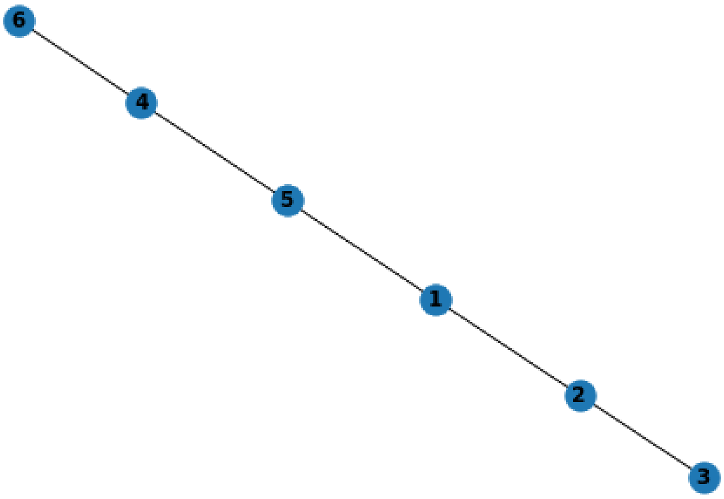
```
[(1, 2), (1, 5), (2, 3), (5, 4), (4, 6)]
```

```
Tree_bfs = nx.bfs_edges(g,1)
t0 = nx.Graph(list(Tree_bfs))
```

```
t0.nodes()
```

```
NodeView((1, 2, 5, 3, 4, 6))
```

```
nx.draw(t0, with_labels=True, font_weight='bold')
```



```
print(list(nx.bfs_successors(g,1)))
```

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

```
Trees = nx.dfs_tree(g,1)
print(Trees.nodes())
```

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

```
print(Trees.edges())
```

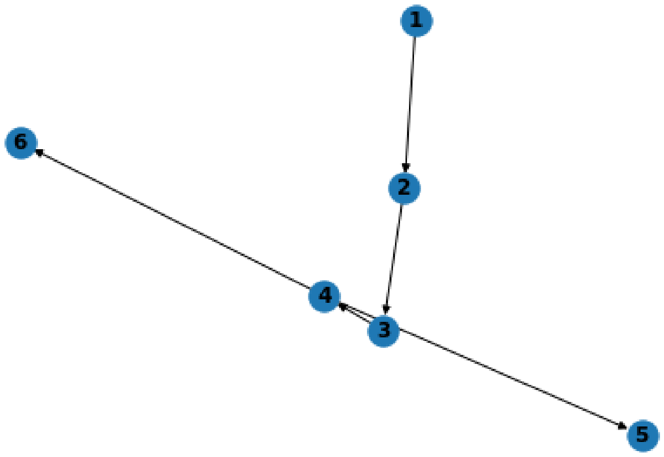
```
[(1, 2), (2, 3), (3, 4), (4, 5), (4, 6)]
```

```
nx.draw(Trees, with_labels=True, font_weight='bold')
```

```
#g.add_path([1,5,2,3,4,6])
Trees1 = nx.dfs_tree(g,1)
print(Trees1.edges())
```

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

```
nx.draw(Trees1, with_labels=True, font_weight='bold')
```



```
#g.add_path([1,5,4,6])
Trees1 = nx.dfs_tree(g,1)
print(Trees1.edges())
```

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

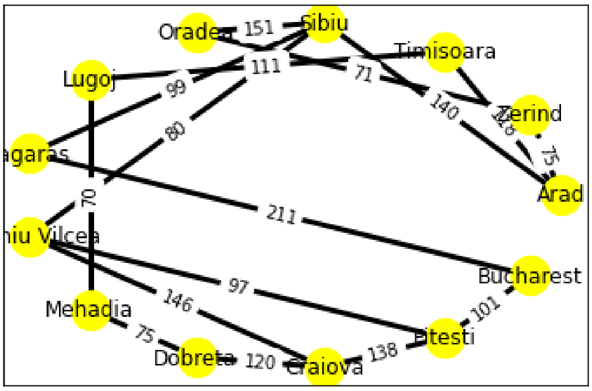
```
#ノード1から
print(list(nx.dfs_preorder_nodes(g,1)))
```

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

```
options = {
    'node_color': 'red',
    'node_size': 800,
    'width': 3,
    'arrowstyle': '-|>',
    'arrowsize': 12,
}


nx.draw_networkx(g, arrows=True, **options)
```

重み付け有効グラフの実装

```
# Romania Graph

#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)]


romania_g = nx.Graph(directed=False)

nodes = []
for city,h in Cities:
    nodes.append(city)

#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)

#Lugoj -> Mehadia -> Dobreda -> Craiova -> Pitesti
romania_g.add_edges_from([(nodes[5],nodes[8])],weight=70)
romania_g.add_edges_from([(nodes[8],nodes[10])],weight=75)
romania_g.add_edges_from([(nodes[10],nodes[11])],weight=120)
romania_g.add_edges_from([(nodes[11],nodes[9])],weight=138)

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)

# not complete



r_weights = romania_g.edges(data=True)


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


options = {
    'node_color': 'yellow',
    'node_size': 500,
    'width': 3
}
pos=nx.circular_layout(romania_g)
nx.draw_networkx_edge_labels(romania_g,pos,edge_labels=edge_labels)
nx.draw_networkx(romania_g, pos, **options)
```



```
for path in nx.all_simple_paths(romania_g, source='Arad', target='Bucharest'):
    print(path)
```

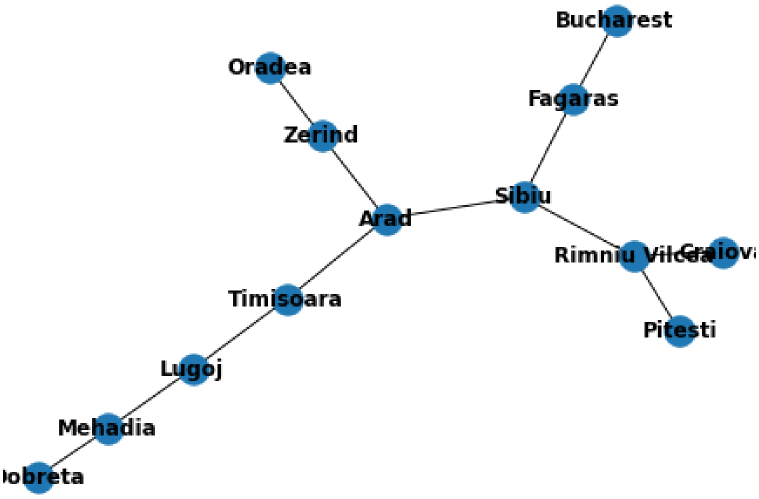
```
['Arad', 'Zerind', 'Oradea', 'Sibiu', 'Fagaras', 'Bucharest']
['Arad', 'Zerind', 'Oradea', 'Sibiu', 'Rimniu Vilcea', 'Pitesti', 'Bucharest']
['Arad', 'Zerind', 'Oradea', 'Sibiu', 'Rimniu Vilcea', 'Craiova', 'Pitesti', 'Bucharest']
['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Dobreta', 'Craiova', 'Pitesti', 'Rimniu Vilcea', 'Sibiu', 'Fagara
['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Dobreta', 'Craiova', 'Pitesti', 'Bucharest']
['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Dobreta', 'Craiova', 'Rimniu Vilcea', 'Sibiu', 'Fagaras', 'Buchar
['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Dobreta', 'Craiova', 'Rimniu Vilcea', 'Pitesti', 'Bucharest']
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
['Arad', 'Sibiu', 'Rimniu Vilcea', 'Pitesti', 'Bucharest']
['Arad', 'Sibiu', 'Rimniu Vilcea', 'Craiova', 'Pitesti', 'Bucharest']
```

```
from itertools import islice
def k_shortest_paths(G, source, target, k, weight=None):
    return list(islice(nx.shortest_simple_paths(G, source, target, weight=weight), k))
for path in k_shortest_paths(romania_g,'Arad','Bucharest',2):
     print(path)
```

```
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
['Arad', 'Sibiu', 'Rimniu Vilcea', 'Pitesti', 'Bucharest']
```

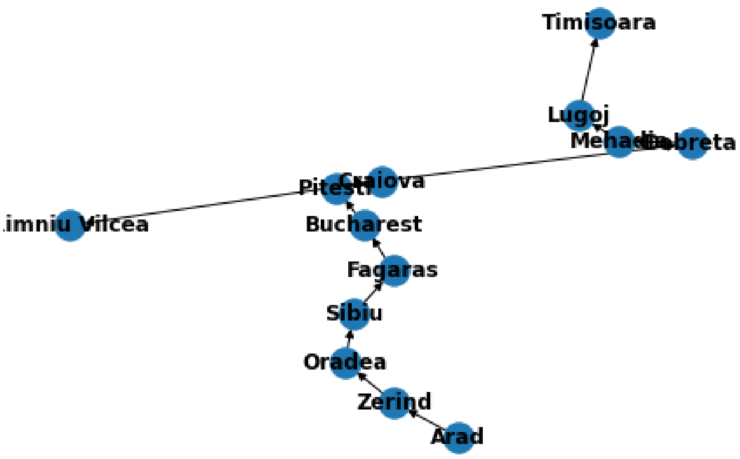
```
Tree_bfs = nx.bfs_edges(romania_g,'Arad')
t0 = nx.Graph(list(Tree_bfs))
```

```
nx.draw(t0, with_labels=True, font_weight='bold')
```

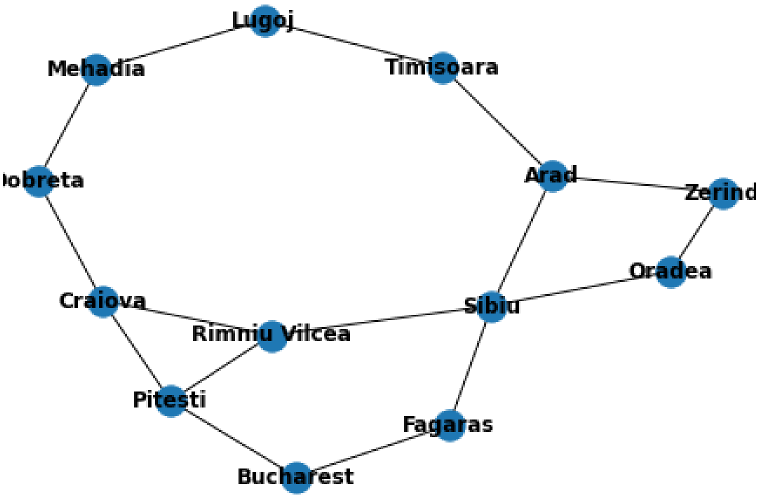


```
Trees = nx.dfs_tree(romania_g,'Arad')
#print(Trees.nodes())
```

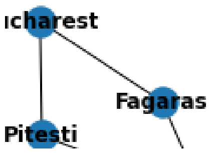
```
#print(Trees.nodes())


#print(Trees.edges())


nx.draw(Trees, with_labels=True, font_weight='bold')
```



```
Tree_dfs = list(nx.edge_dfs(nx.Graph(romania_g.edges()), romania_g.nodes()))
t0 = nx.Graph(list(Tree_dfs))


nx.draw(t0, with_labels=True, font_weight='bold')
```



```
Tree_dfs = list(nx.edge_dfs(romania_g,'Arad'))
t1 = nx.Graph(list(Tree_dfs))
nx.draw(t1, with_labels=True, font_weight='bold')
```



```
#Path length
p_l = nx.shortest_path_length(romania_g, source='Arad', target='Bucharest')
print(p_l)
```

```
3
```

```
#Path length
p_l = nx.shortest_path_length(romania_g, source='Arad', target='Bucharest',weight='weight')
print("AradからBucharestまでの最短経路長さは",p_l,"km")
```

```
AradからBucharestまでの最短経路長さは 418 km
```

```
#bell_ford = nx.bellman_ford(romania_g, source='Arad', weight='weight')
#print("Aradからの最短経路長さは",bell_ford,"km")
```

```
#dijkstra_path
#Path length
path = nx.dijkstra_path(romania_g, source='Arad', target='Bucharest',weight='weight')
print(path)
```

```
['Arad', 'Sibiu', 'Rimniu Vilcea', 'Pitesti', 'Bucharest']
```

```
#dijkstra_path_length
p_l = nx.dijkstra_path_length(romania_g, source='Arad', target='Bucharest',weight='weight')
print(p_l)
```

```
418
```

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

```
L
```

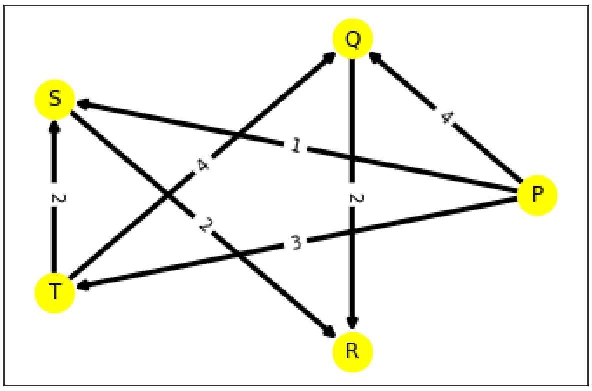
```
{'P': {'Q': 4, 'S': 1, 'T': 3},
 'Q': {'R': 2, 'S': 1},
 'S': {'R': 4},
 'T': {'Q': 4, 'S': 2}}
```

```
wg = nx.DiGraph(directed=True)
```

```
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','R')],weight=2)
wg.add_edges_from([('S','R')],weight=2)
```

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

```
                    for u,v,d in wg.edges(data=True)])


pos=nx.circular_layout(wg)
nx.draw_networkx_edge_labels(wg,pos,edge_labels=edge_labels)
nx.draw_networkx(wg, pos, **options)
```



```
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 0x7fcfc31b7748>
     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’])
```

```
nodes = list(G.Vertex())


nodes
```

```
     [’A’, ’B’, ’C’, ’D’, ’E’]
```

```
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
```

```
     [(’A’, ’C’),
      ('A', 'D'),
      ('B', 'D'),
      ('B', 'A'),
      ('C', 'D'),
      ('C', 'E'),
      ('D', 'E')]
```

```
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
```

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

```
wg = nx.DiGraph(directed=True)
```

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

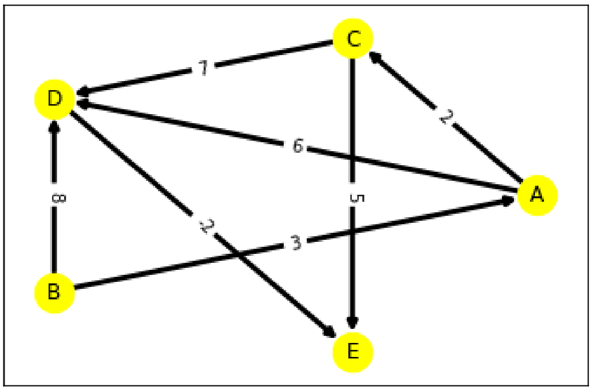
```
import matplotlib.pyplot as plt
limits = plt.axis('off')  # turn of axis
```

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

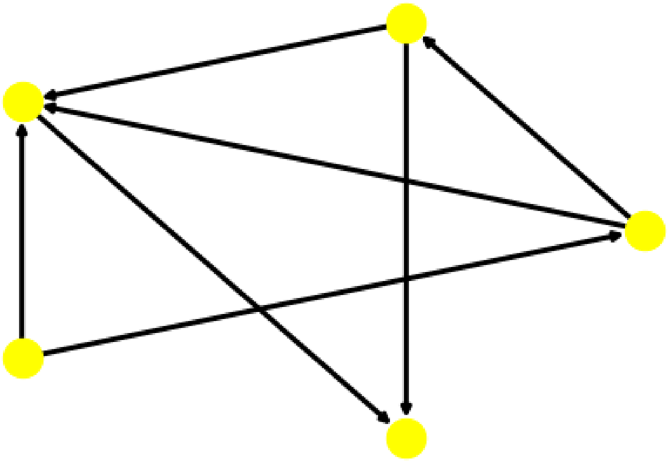
```
edge_labels
```

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

```
pos=nx.circular_layout(wg)
nx.draw_networkx_edge_labels(wg,pos,edge_labels=edge_labels)
nx.draw_networkx(wg, pos, arrows=True, **options)
nx.draw_networkx(wg, pos, arrows=True, **options)
```

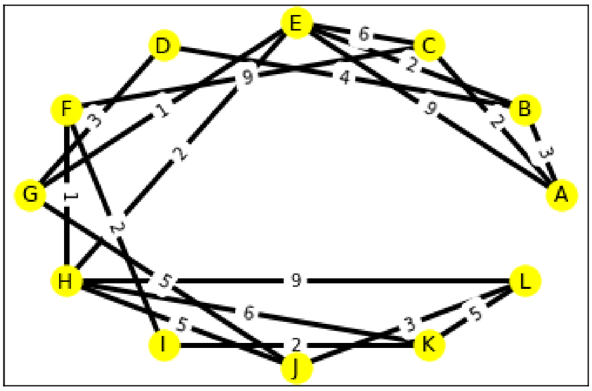


```
nx.draw(wg, pos, arrows=True, **options)
```



```
# Example 2019/11/19
g1 = nx.Graph(directed=False)

g1.add_edges_from([(’A’,’B’)],weight=3)
g1.add_edges_from([(’A’,’C’)],weight=2)
g1.add_edges_from([(’A’,’E’)],weight=9)

g1.add_edges_from([(’B’,’E’)],weight=2)
g1.add_edges_from([(’B’,’D’)],weight=4)

g1.add_edges_from([(’C’,’E’)],weight=6)
g1.add_edges_from([(’C’,’F’)],weight=9)

g1.add_edges_from([(’D’,’G’)],weight=3)

g1.add_edges_from([(’E’,’G’)],weight=1)
g1.add_edges_from([(’E’,’H’)],weight=2)

g1.add_edges_from([(’C’,’F’)],weight=9)

g1.add_edges_from([(’F’,’H’)],weight=1)
g1.add_edges_from([(’F’,’I’)],weight=2)

g1.add_edges_from([(’G’,’J’)],weight=5)

g1.add_edges_from([(’I’,’K’)],weight=2)

g1.add_edges_from([(’H’,’J’)],weight=5)
g1.add_edges_from([(’H’,’I’)],weight=9)
```

```
g1.add_edges_from([('H','L')],weight=9)
g1.add_edges_from([('H','K')],weight=6)

g1.add_edges_from([('K','L')],weight=5)
g1.add_edges_from([('J','L')],weight=3)



edge_labels=dict([((u,v,),d['weight'])
                  for u,v,d in g1.edges(data=True)])


options = {
    'node_color': 'yellow',
    'node_size': 300,
    'width': 3
}
pos=nx.circular_layout(g1)
nx.draw_networkx_edge_labels(g1,pos,edge_labels=edge_labels)
nx.draw_networkx(g1, pos, arrows=True, **options)
```



```
print(g1.edges(data=True))
```

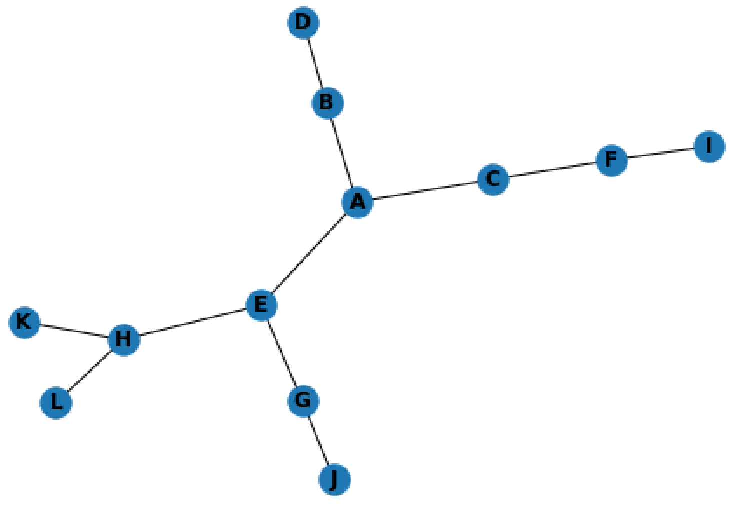
```
[('A', 'B', {'weight': 3}), ('A', 'C', {'weight': 2}), ('A', 'E', {'weight': 9}), ('B', 'E', {'weight': 2}),
```

```
for path in nx.all_simple_paths(g1, source='A', target='L'):
    print(path)
```

```
['A', 'B', 'E', 'H', 'F', 'I', 'K', 'L']
['A', 'B', 'E', 'H', 'J', 'L']
['A', 'B', 'E', 'H', 'L']
['A', 'B', 'E', 'H', 'K', 'L']
['A', 'B', 'D', 'G', 'E', 'C', 'F', 'H', 'J', 'L']
['A', 'B', 'D', 'G', 'E', 'C', 'F', 'H', 'L']
['A', 'B', 'D', 'G', 'E', 'C', 'F', 'H', 'K', 'L']
['A', 'B', 'D', 'G', 'E', 'C', 'F', 'I', 'K', 'H', 'J', 'L']
['A', 'B', 'D', 'G', 'E', 'C', 'F', 'I', 'K', 'H', 'L']
['A', 'B', 'D', 'G', 'E', 'C', 'F', 'I', 'K', 'L']
['A', 'B', 'D', 'G', 'E', 'H', 'F', 'I', 'K', 'L']
['A', 'B', 'D', 'G', 'E', 'H', 'J', 'L']

['A', 'B', 'D', 'G', 'E', 'H', 'L']
['A', 'B', 'D', 'G', 'E', 'H', 'K', 'L']
['A', 'B', 'D', 'G', 'J', 'H', 'E', 'C', 'F', 'I', 'K', 'L']
['A', 'B', 'D', 'G', 'J', 'H', 'F', 'I', 'K', 'L']
['A', 'B', 'D', 'G', 'J', 'H', 'L']
['A', 'B', 'D', 'G', 'J', 'H', 'K', 'L']
['A', 'B', 'D', 'G', 'J', 'L']
['A', 'C', 'E', 'B', 'D', 'G', 'J', 'H', 'F', 'I', 'K', 'L']
['A', 'C', 'E', 'B', 'D', 'G', 'J', 'H', 'L']
['A', 'C', 'E', 'B', 'D', 'G', 'J', 'H', 'K', 'L']
```

```
['A', 'C', 'E', 'B', 'D', 'G', 'J', 'L']
['A', 'C', 'E', 'G', 'J', 'H', 'F', 'I', 'K', 'L']
['A', 'C', 'E', 'G', 'J', 'H', 'L']
['A', 'C', 'E', 'G', 'J', 'H', 'K', 'L']
['A', 'C', 'E', 'G', 'J', 'L']
['A', 'C', 'E', 'H', 'F', 'I', 'K', 'L']
['A', 'C', 'E', 'H', 'J', 'L']
['A', 'C', 'E', 'H', 'L']
['A', 'C', 'E', 'H', 'K', 'L']
['A', 'C', 'F', 'H', 'E', 'B', 'D', 'G', 'J', 'L']
['A', 'C', 'F', 'H', 'E', 'G', 'J', 'L']
['A', 'C', 'F', 'H', 'J', 'L']
['A', 'C', 'F', 'H', 'L']
['A', 'C', 'F', 'H', 'K', 'L']
['A', 'C', 'F', 'I', 'K', 'H', 'E', 'B', 'D', 'G', 'J', 'L']
['A', 'C', 'F', 'I', 'K', 'H', 'E', 'G', 'J', 'L']
['A', 'C', 'F', 'I', 'K', 'H', 'J', 'L']
['A', 'C', 'F', 'I', 'K', 'H', 'L']
['A', 'C', 'F', 'I', 'K', 'L']
['A', 'E', 'B', 'D', 'G', 'J', 'H', 'F', 'I', 'K', 'L']
['A', 'E', 'B', 'D', 'G', 'J', 'H', 'L']
['A', 'E', 'B', 'D', 'G', 'J', 'H', 'K', 'L']
['A', 'E', 'B', 'D', 'G', 'J', 'L']
['A', 'E', 'C', 'F', 'H', 'J', 'L']
['A', 'E', 'C', 'F', 'H', 'L']
['A', 'E', 'C', 'F', 'H', 'K', 'L']
['A', 'E', 'C', 'F', 'I', 'K', 'H', 'J', 'L']
['A', 'E', 'C', 'F', 'I', 'K', 'H', 'L']
['A', 'E', 'C', 'F', 'I', 'K', 'L']
['A', 'E', 'G', 'J', 'H', 'F', 'I', 'K', 'L']
['A', 'E', 'G', 'J', 'H', 'L']
['A', 'E', 'G', 'J', 'H', 'K', 'L']
['A', 'E', 'G', 'J', 'L']
['A', 'E', 'H', 'F', 'I', 'K', 'L']
['A', 'E', 'H', 'J', 'L']
['A', 'E', 'H', 'L']
['A', 'E', 'H', 'K', 'L']
```

```
Tree_bfs = nx.bfs_edges(g1,'A')
t0 = nx.Graph(list(Tree_bfs))
nx.draw(t0, with_labels=True, font_weight='bold')
```

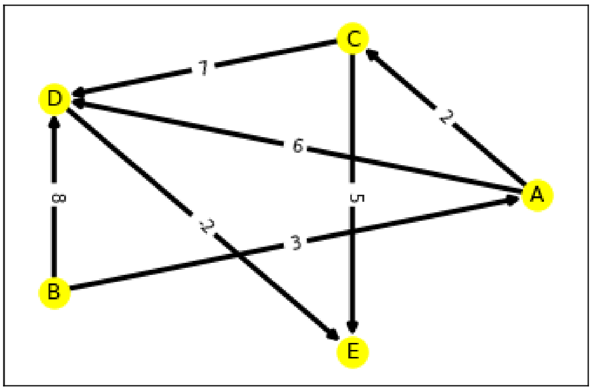


ヒューリスティック無しグラフ探索(2)

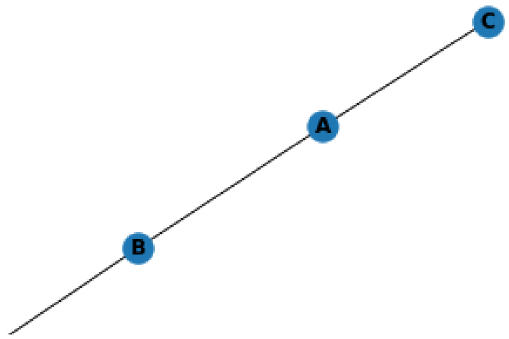
- 均一コスト探索 (ダイクストラ法: djikstra algorithm)

- 双方向探索 (そうほうこう)　同時に2つの方向から探索を行う) (bidirectional search)

```
%matplotlib inline


# グラフライブラリを使用する
import networkx as nx


#L={'A':{'C':2,'D':6},'B':{'D':8,'A':3},
#   'C':{'D':7,'E':5},'D':{'E':-2},'E':{}}


# Example 2019/11/19
g0 = nx.DiGraph(directed=True)

g0.add_edges_from([('A','C')],weight=2)
g0.add_edges_from([('A','D')],weight=6)
g0.add_edges_from([('B','D')],weight=8)
g0.add_edges_from([('B','A')],weight=3)
g0.add_edges_from([('C','D')],weight=7)
g0.add_edges_from([('C','E')],weight=5)
g0.add_edges_from([('D','E')],weight=-2)



edge_labels=dict([((u,v,),d['weight'])
                 for u,v,d in g0.edges(data=True)])


options = {
    'node_color': 'yellow',
    'node_size': 300,
    'width': 3
}
pos=nx.circular_layout(g0)
nx.draw_networkx_edge_labels(g0,pos,edge_labels=edge_labels)
nx.draw_networkx(g0, pos, arrows=True, **options)
```

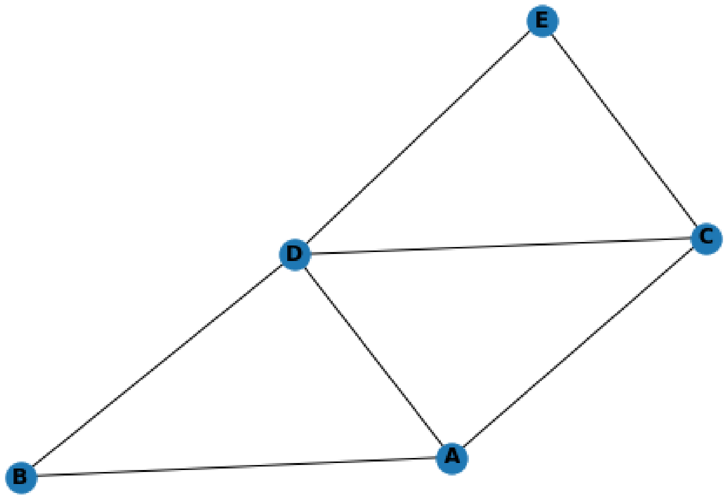


```
Tree_bfs = nx.bfs_edges(g0,'B')
t0 = nx.Graph(list(Tree_bfs))


nx.draw(t0, with_labels=True, font_weight='bold')
```



```
for path in nx.all_simple_paths(g0, source='B', target='E'):
    print(path)

      ['B', 'D', 'E']
      ['B', 'A', 'C', 'D', 'E']
      ['B', 'A', 'C', 'E']
      ['B', 'A', 'D', 'E']
```

```
Tree_dfs = list(nx.edge_dfs(g0,'B'))
t1 = nx.Graph(list(Tree_dfs))
nx.draw(t1, with_labels=True, font_weight='bold')
```



```
list(Tree_dfs)

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

```
#dijkstra_path
path = nx.dijkstra_path(g0, source='B', target='E',weight='weight')
print(path)

      ['B', 'D', 'E']
```

```
nodes,dist = nx.dijkstra_predecessor_and_distance(g0, source='B')
print(nodes)
print(dist)

      {'B': [], 'D': ['B'], 'A': ['B'], 'C': ['A'], 'E': ['D']}
```

```
    {'B': 0, 'A': 3, 'C': 5, 'D': 8, 'E': 6}
```

```
#Path length
length = nx.dijkstra_path_length(g0, source='B', target='E',weight='weight')
print(length)
```

```
    6
```

```
length,path=nx.bidirectional_dijkstra(g0,'B','E')
```

```
print(length)
```

```
    6
```

```
print(path)
```

```
    ['B', 'D', 'E']
```