

# 数理最適化実験 レポート03

情報メディア創成学類

201911419 土佐凜斗

# トピック：経路最適化

## グラフ理論入門

グラフの数式による表現

グラフ全体： $G = (V, E)$ ,

点の集合： $V = \{1, 2, 3, 4\}$ ,

辺の集合： $E = \{a, b, c, d, e, f, g\}$

辺の始点と終点：

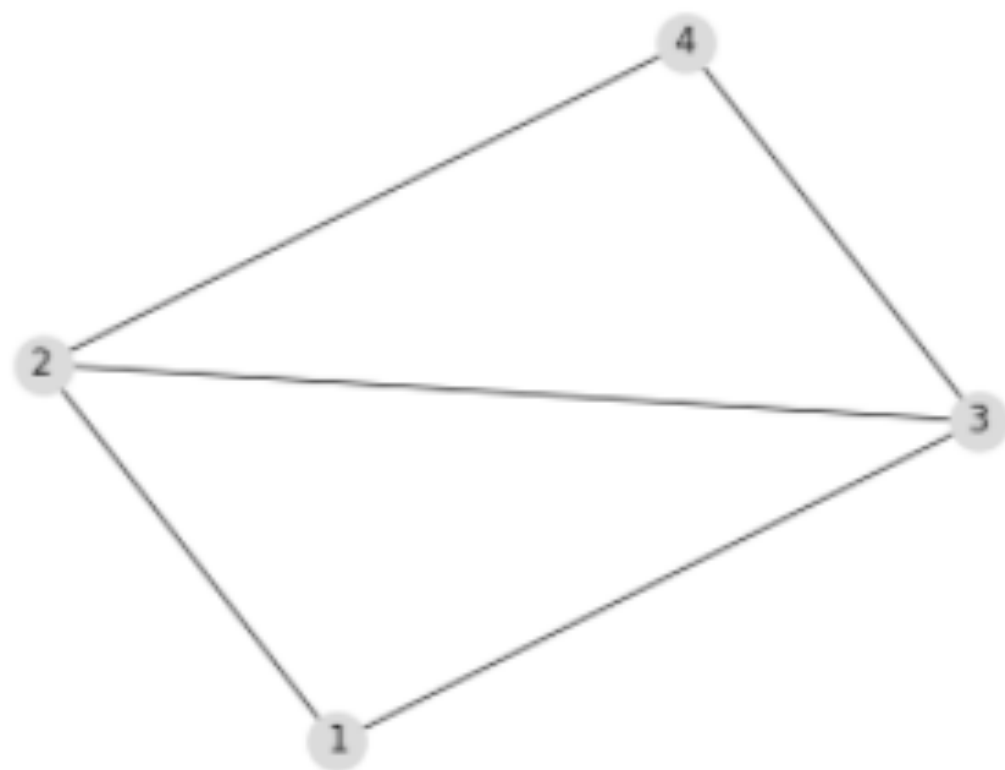
$a = (1, 2), b = (1, 2), c = (1, 3), d = (2, 3), e = (2, 4), f = (3, 3), g = (3, 4)$

## pythonでのグラフ描画

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
vlist = [1,2,3,4]
elist = [(1,1), (1,1), (2,2), (1,2), (1,3), (2,3), (2,4), (3,4)]
G.add_nodes_from(vlist)
G.add_edges_from(elist)
nx.draw_networkx(G, node_color='lightgray', node_size=400)
plt.axis('off')
plt.show()
```

Output



# 用語の整理

## 隣接行列 (Aで表す)

行も列も頂点集合 $V$ によって添字付けされた行列で,  $A(G)$ の成分を $m[u, v]$  ( $u, v$ は $V$ の要素) とすると ( $u, v$ ) が枝のとき1, それ以外は0として表す

## 接続行列 (Mで表す)

行が $V$ , 列が $E$ で添字付けされた行列で,  $M(G)$ の成分を $m[v, e]$  ( $v$ は $V$ の要素,  $e$ は $E$ の要素) とすると  $v$ と $e$ が接続していれば1, それ以外は0とする.

有向グラフの場合,  $v$ が $e$ の始点なら-1, 終点なら+1.

```
# 無向グラフの隣接行列, 接続行列
G = nx.MultiGraph()
G.add_edges_from([(1,2), (1,3), (3,1), (2,3), (2,2)])
A = nx.adjacency_matrix(G)
M = nx.incidence_matrix(G)
print('A = ', A.toarray())
print('M = ', M.toarray())
```

```
# 有向グラフの隣接行列, 接続行列
G2 = nx.MultiDiGraph()
G2.add_edges_from([(1,2), (1,3), (3,1), (2,3), (2,2)])
A2 = nx.adjacency_matrix(G2)
M2 = nx.incidence_matrix(G2, oriented=True)
print('A = ', A2.toarray())
print('M = ', M2.toarray())
```

```
ringringnoMacBook-Pro:work3
A = [[0 1 2]
     [1 1 1]
     [2 1 0]]
M = [[1. 1. 1. 0. 0.]
     [1. 0. 0. 1. 0.]
     [0. 1. 1. 1. 0.]]
```

```
ringringnoMacBook-Pro:work3
A = [[0 1 1]
     [0 1 1]
     [1 0 0]]
M = [[-1. -1. 0. 0. 1.]
     [ 1. 0. -1. 0. 0.]
     [ 0. 1. 1. 0. -1.]]
```

## 単純グラフ

自己ループや多重辺を持たないグラフをいう

## 完全グラフ

- ∴ 単純グラフの中で、どの頂点間も枝で隣接しているグラフを完全グラフという

```
# 完全グラフの描画
```

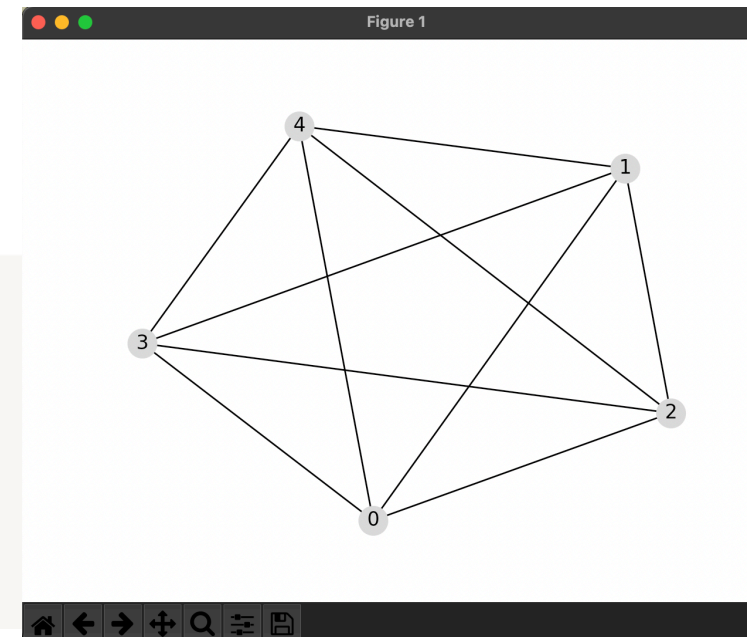
```
G = nx.complete_graph(5)
```

```
p = nx.spring_layout(G, iterations=100)
```

```
nx.draw_networkx(G, pos=p, node_color='lightgray', node_size=300)
```

```
plt.axis('off')
```

```
plt.show()
```



## 2部グラフ

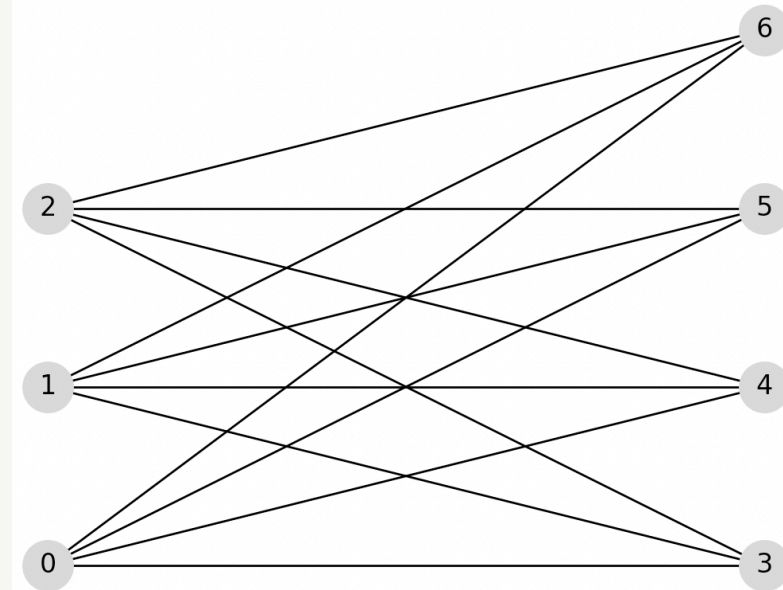
グラフの頂点集合 $V$ が2つの部分集合 $X$ と $Y$ に分割され、どの枝も $X$ と $Y$ の両方に接続しているとき、そのグラフを2部グラフという。さらに、 $X$ と $Y$ のどの頂点間にも枝が存在するとき、完全2部グラフという。

このとき、 $X$ の頂点数を $m$ 、 $Y$ の頂点数を $n$ とすると、 $V$ の頂点数は $m+n$ 、枝数は $mn$ である。

ここで、 $V$ が $X$ と $Y$ に分割されるというのは $V = X \cup Y, X \cap Y = \phi$ であることを指す

```
# 2部グラフ
m = 3
n = 4
G = nx.complete_bipartite_graph(m,n)
p = {}
for i in range(m):
    p[i] = (0,i)
for j in range(n):
    p[m+j] = (1,j)
nx.draw_networkx(G, pos=p, node_color='lightgray', node_size=500)
plt.axis('off')
plt.show()
```

Figure 1





## 次数

グラフ  $G = (V, E)$  において頂点  $v \in V$  に接続している枝の数を  $v$  の次数をいい  $\deg(v)$  で表す (自己ループの場合  $\deg(v)$  は辺一つで +2)

- 握手の定理

$G = (V, E)$  において,  $\sum_{v \in V} \deg(v) = 2|E|$  が成り立つ

※ 点と点をつなぐ辺は一つで, それぞれから一本と見られるから

- 次数が奇数の頂点を奇点, 次数が偶数の頂点を偶点と言う  
任意のグラフにおいて, 奇点は偶数個である.

## 同型性

$G = (V, E)$ と $G' = (V', E')$ において、頂点集合 $V$ と $V'$ の要素間に1対1の対応がつけられて、その対応関係において、枝集合 $E$ と $E'$ に過不足が無いとき、 $G$ と $G'$ は同型であるという。

```
# 同型性のチェック
G1 = nx.Graph()
G1.add_edges_from([(1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6)])

G2 = nx.Graph()
G2.add_edges_from([('a','b'), ('a','d'), ('a','f'), ('b','c'), ('a','e'), ('c','f')])

G3 = nx.Graph()
G3.add_edges_from([('x','z'), ('x','y'), ('y','z'), ('x','u'), ('y','v')])

print(nx.is_isomorphic(G1,G2))
# >>> True
print(nx.is_isomorphic(G1,G3))
# >>> False
```

## 歩道

頂点から頂点へ、ジャンプせずに枝のみを経由して行き来するような経路を歩道という。

## パス

どの頂点も2度以上現れない歩道を路、あるいはパスという

## 閉路

$v_0$ を始点、 $v_n$ を終点とするとき始点と終点が同じである歩道を、閉路という

## サイクル

始点、終点以外の頂点を2度通らない閉路をサイクルという

## 連結なグラフ

任意の点の間にパスが存在するとき、そのグラフを連結なグラフという、連結でないグラフを非連結なグラフという。

非連結なグラフGは非連結な部分グラフに分割することができ、それらをGの連結成分という。非連結なグラフを連結成分に分割することを連結成分分解という。

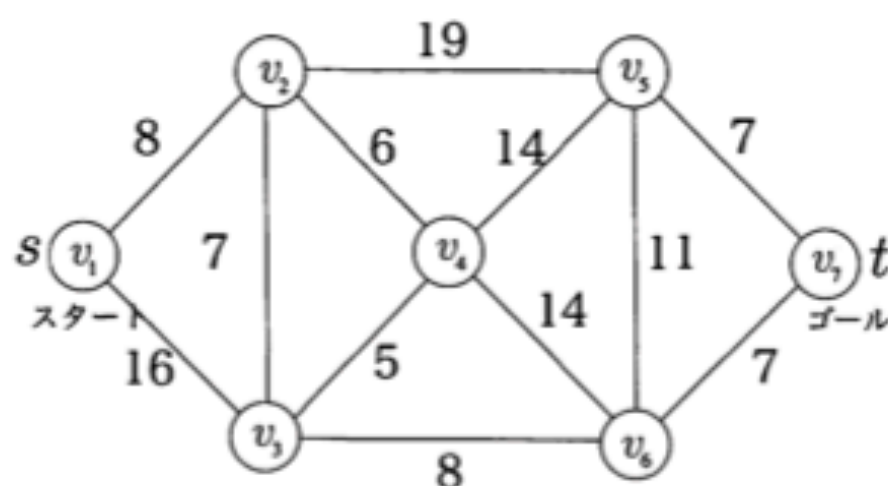
Python ▾

```
# 連結性のチェック
G = nx.path_graph(4)
nx.add_path(G, [10, 11, 12])
print(nx.is_connected(G))
# >>> False
for c in nx.connected_components(G):
    print(c)
# >>> {0, 1, 2, 3}
# >>> {10, 11, 12}
```

# 最短経路問題

スタートからゴールまでのパスが何通りかある中で、重み（長さ）の合計が最小となるパスを見つける問題

カーナビやWebページの経路探索ソフトなどで利用される最適化問題



## ダイクストラ法




1. スタート地点 $s$ からゴールだけでなく全ての点への最短経路とその長さを求めることができる
2. 集合 $W$ として最短経路とその長さが確定した点の集合,  $\bar{W} = V - W$ として最短経路とその長さが確定していない点の集合を用意する.  $W = \phi, \bar{W} = V$ を初期値として $s$ から近い順に $W$ に点を加えていき,  $W = V$ になったら終了.
3.  $d(s, v)$ に,  $s \rightarrow v$ への暫定の最短経路長を格納する. 初期値は $d(s, s) = 0$ ,  $d(s, v) = +\infty, \forall v \in V - \{s\}$ である. 繰り返し途中で $d(s, v)$ を更新する

ダイクストラ法では, ある一点を始点とした拡張木への最短経路を考えるので, その過程で木を構成する.

これを最短経路木といい, ダイクストラ法は最短経路木を求めるアルゴリズムでもあるとも言える.

## NetworkXを用いた解法

Python ▾

 Copy Caption ...

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

weighted_elist = [('v1', 'v2', 8), ('v1', 'v3', 16), ('v2', 'v3', 7),
                  ('v2', 'v4', 6), ('v2', 'v5', 19), ('v3', 'v4', 5),
                  ('v3', 'v6', 8), ('v4', 'v5', 14), ('v4', 'v6', 14),
                  ('v5', 'v6', 11), ('v5', 'v7', 7), ('v6', 'v7', 7)]

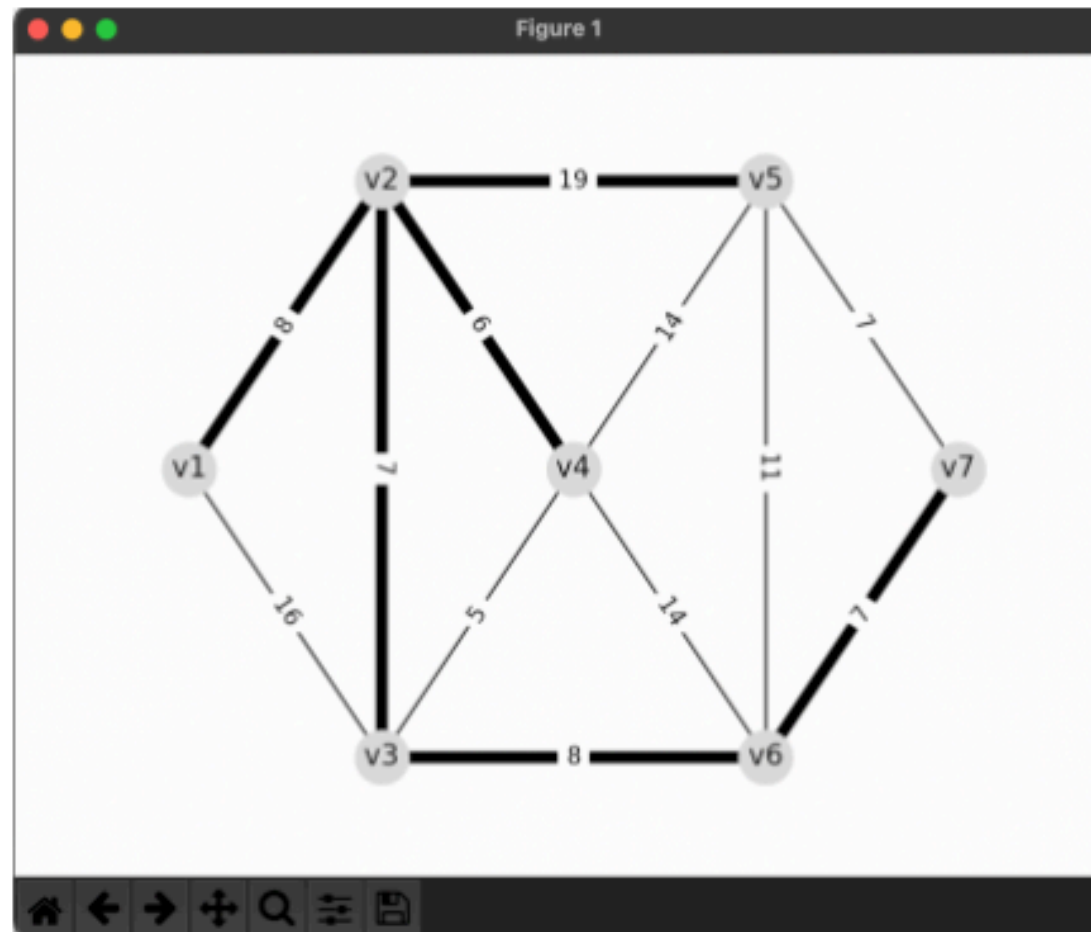
p = {'v1':(0,1), 'v2':(1,2), 'v3':(1,0), 'v4':(2,1), 'v5':(3,2), 'v6':(3,

G = nx.Graph()
G.add_weighted_edges_from(weighted_elist)
elbs = {(u,v):G[u][v]['weight'] for (u,v) in G.edges()}

s = 'v1'
nodes = set(G.nodes()) - {s}
T = set({})
for v in nodes:
    sp = nx.dijkstra_path(G, s, v)
    T = T.union({tuple(x) for x in np.array([sp[:-1], sp[1:]]).T})
T = list(T)

nx.draw_networkx(G, pos=p, node_color='lightgrey', node_size=500, width=5)
nx.draw_networkx_edges(G, pos=p, edgelist=T, width=5)
nx.draw_networkx_edge_labels(G, pos=p, edge_labels=elbs)
plt.axis('off')
plt.show()
```

output

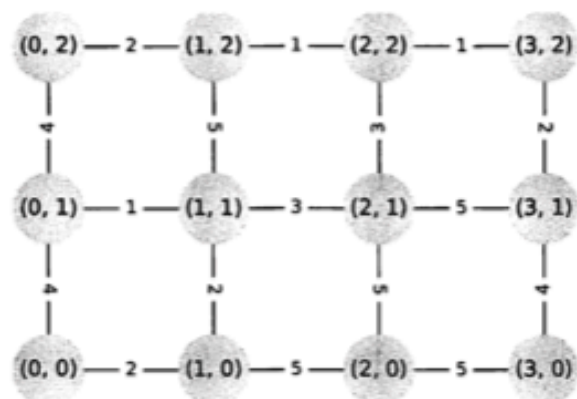




# (中国人)郵便配達人問題



下のような道路の路線図がある。道路上にはその道路の長さ（距離）が書いてある。道路の両脇には満遍なく家が立っており、郵便配達人はそれらの家に郵便物を配達するため全ての道路を少なくとも一回は通過しなければならない。このとき移動距離が最小となる配達人の巡回路を求める。



グラフ論理的に言うと、このようになる。

与えられた重み付きグラフ  $G = (E, V)$  に対して、グラフの枝をいくつか重複させて枝の重みの合計をが最小となるようなオイラーグラフを作れ。

## アルゴリズム

1. 与えられたグラフがオイラーグラフならば、そのグラフに対するオイラー閉路を求める。
2. 与えられたグラフが準オイラーグラフならば、オイラーグラフの判定条件より、奇点が2つあるはずなので、その2つの奇点間の、与えられた重みによる最短路を求め、その最短路に沿って枝を重複させたグラフが答えとなる。
3. 上記以外（奇点が4つ以上）の場合、まず全ての奇点間の最短路とその長さを求める。そのために、奇点を点集合とする完全グラフを作り、枝の重みは最短路の長さとする、その重み付き完全グラフに対して、重み最小の完全マッチングを解く。（これは必ず最適解が存在する）

このマッチングで使われた奇点のペア間に、そのペアの最短路に沿って枝を重複させる。

### ※ 重み最小マッチング

重みの最大値を $\text{maxweight}$ とし、各枝の重みを $w(e)$ から $\text{maxweight} - w(e) + 1$ に変換すると、全ての重みが1より大きくなり、ここで重み最大のマッチングを求めることで完全グラフとなり、この重みを元に戻せば最小重みの完全マッチングが得られる。

```

# 問題のグラフ生成と表示
def MakeGraph(ARGV):
    if len(ARGV) < 1:
        print('グリッドのサイズを正しく入力してください')
        sys.exit()
    elif len(ARGV) == 2:
        print('重みをランダムで設定します')
        G = nx.grid_2d_graph(int(ARGV[0]), int(ARGV[1]))
        for (u,v) in G.edges():
            G[u][v]['weight'] = np.random.randint(1,6)
    else:
        G = nx.grid_2d_graph(int(ARGV[0]), int(ARGV[1]))
        n = 2
        for (u,v) in G.edges():
            G[u][v]['weight'] = ARGV[n]
            n += 1

    nx.draw_networkx(G,
                     pos={v:v for v in G.nodes()},
                     node_color='lightgray',
                     node_size=1500,
                     width=1)
    nx.draw_networkx_edge_labels(G,
                                edge_labels={(u,v):G[u][v]['weight']
                                              for (u,v) in G.edges()},
                                pos={v:v for v in G.nodes()},)

    plt.axis('off')
    plt.show()
    return G

```

```

# 全ての奇点間の最短路の長さを計算
def MakeOddGraph(G):
    # 奇点の点集合作成
    Vodd = [v for v in G.nodes() if G.degree(v)%2 == 1]
    if len(Vodd) == 0:
        return 0
    else:
        #dist[vodd1][vodd2]に計算結果を格納
        dist = dict(nx.all_pairs_dijkstra_path_length(G))

        # 頂点がVoddの完全グラフを作成（重みは最短路長）
        K = nx.Graph()
        K.add_weighted_edges_from([(u,v,dist[u][v])
                                   for (u,v) in combinations(Vodd, 2)])

        nx.draw_networkx(K,
                        pos={v:v for v in K.nodes()},
                        node_color='lightgray',
                        node_size=1500,
                        width=1)
        nx.draw_networkx_edge_labels(K,
                                   pos={v:v for v in K.nodes()},
                                   edge_labels={(u,v):K[u][v]['weight']
                                                for (u,v) in K.edges()})

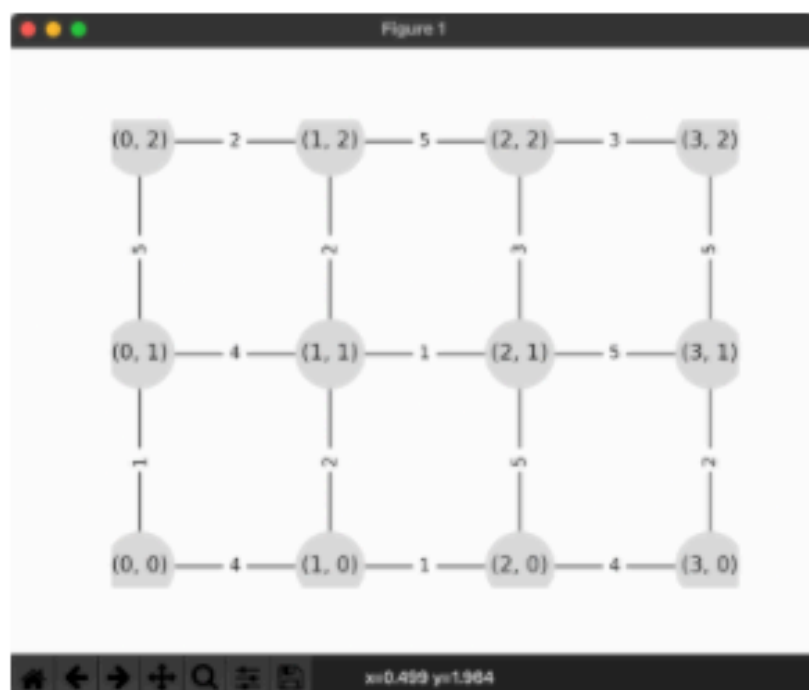
        plt.axis('off')
        plt.show()
        return K

```

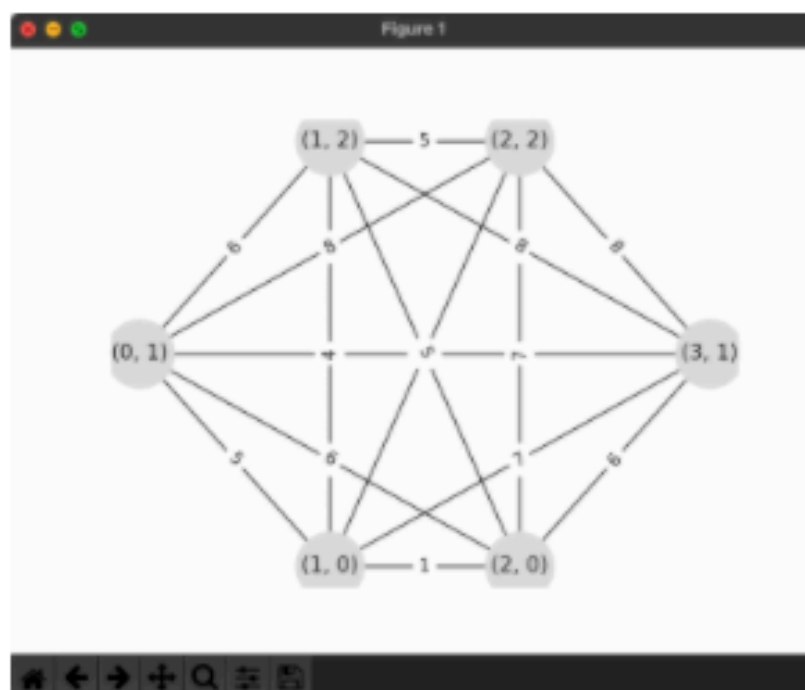
## Code (入力は4×3の重みランダム)

▶ code

Output 1 問題となるグラフ



Output 2 奇点を点集合とする完全グラフ



```

# 重み最小の完全マッチングを求める（偶数個の頂点からなる完全グラフは必ず最適解を持つ）
# 重みを汎化して重み最大マッチングを求めることで重み最小マッチングを得る
def WeightMatching(K):
    CK = K.copy()
    wm = max(CK[u][v]['weight'] for (u,v) in CK.edges())
    for (u,v) in K.edges():
        CK[u][v]['weight'] = wm - CK[u][v]['weight'] + 1

    m = nx.max_weight_matching(CK, maxcardinality=True)
    md = dict(m)
    mm = []
    for (u,v) in md.items():
        if (u,v) not in mm and (v,u) not in mm:
            mm.append((u,v))

    nx.draw_networkx(CK,
                     pos={v:v for v in CK.nodes()},
                     node_color='lightgray',
                     node_size=1500,
                     width=1)
    nx.draw_networkx_edge_labels(CK,
                                pos={v:v for v in CK.nodes()},
                                edge_labels={(u,v):CK[u][v]['weight']
                                              for (u,v) in CK.edges()})

    nx.draw_networkx_edges(CK,
                           pos={v:v for v in CK.nodes()},
                           edgelist=mm,
                           width=5)

    plt.axis('off')
    plt.show()
    return mm

```

```

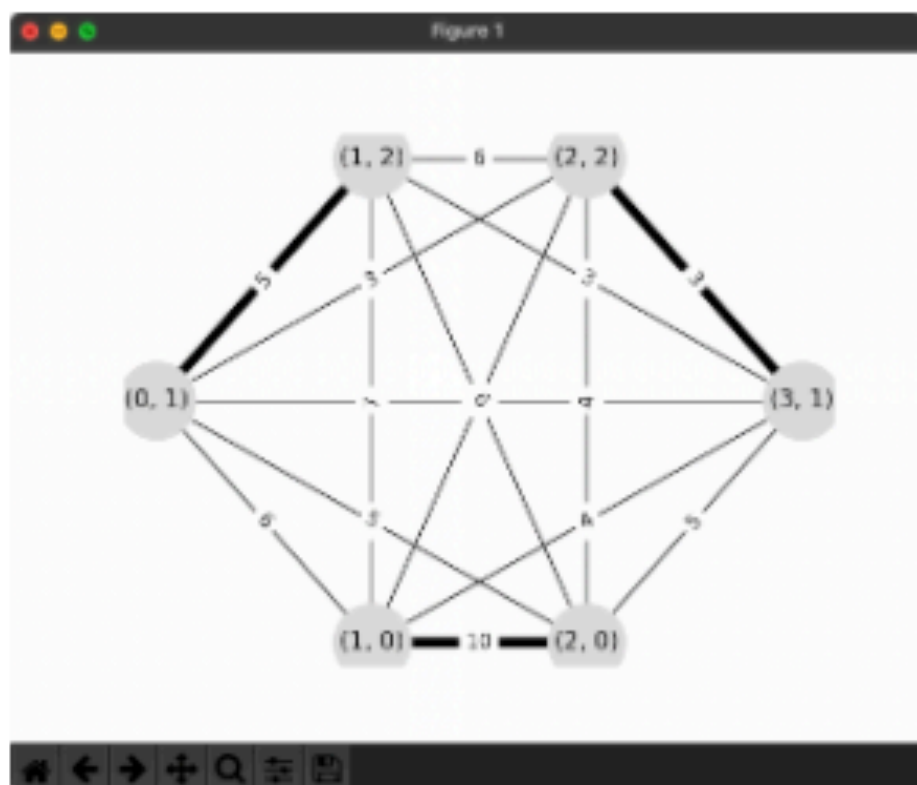
# マッチング（最短路）に沿って、枝を重複させてオイラー閉路を求める
def EulerGraph(G, mm):
    CG = G.copy()
    for (u,v) in mm:
        dp = nx.dijkstra_path(G, u, v)
        # NetworkXのGraphでは多重辺を扱えないので中間点を作成する
        for i in range(len(dp)-1):
            (ux, uy) = dp[i]
            (vx, vy) = dp[i+1]
            if ux == vx:
                wx = ux + 0.3
                wy = (uy + vy) / 2.0
            else:
                wx = (ux + vx) / 2.0
                wy = uy + 0.3
            CG.add_edges_from([(ux,uy), (wx,wy)], [(wx,wy), (vx,vy)]))

    nx.draw_networkx(CG,
                      pos={v:v for v in CG.nodes()},
                      node_color='lightgray',
                      node_size=1500,
                      width=1)

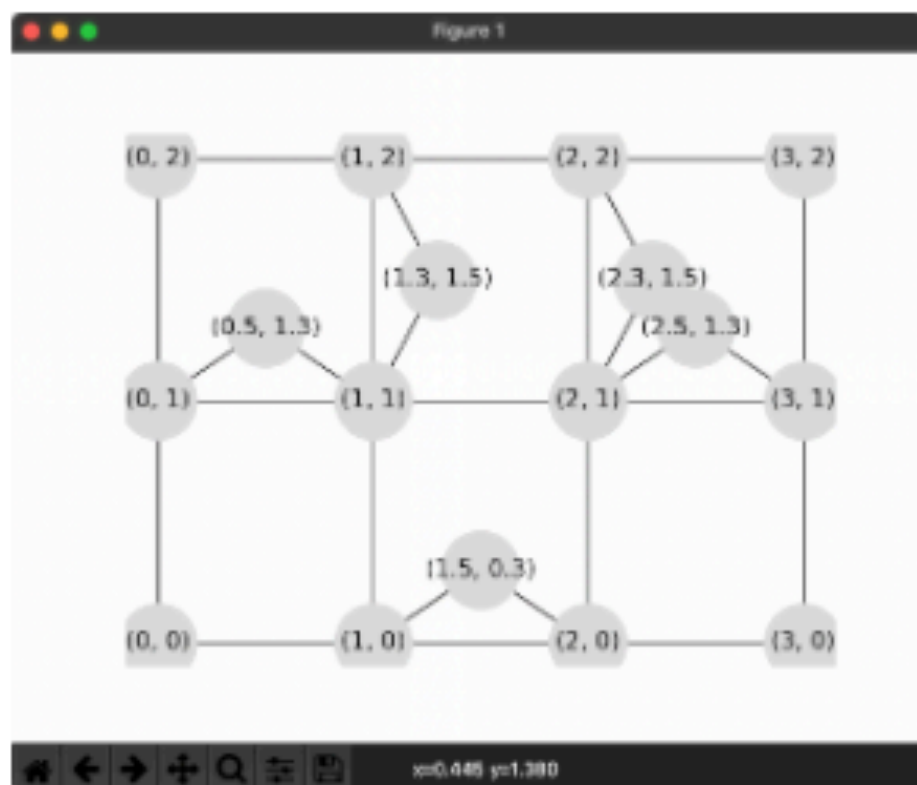
    plt.axis('off')
    plt.show()
    return CG

```

### Output 3 重み最小の完全マッチング



#### Output 4 オイラー閉路を作成





```

# できたグラフからオイラー閉路を作成
def EulerCircuit(CG):
    ec = nx.eulerian_circuit(CG)
    for (i,j) in ec:
        print(i, end='->')
    print('end')
    return 0

```

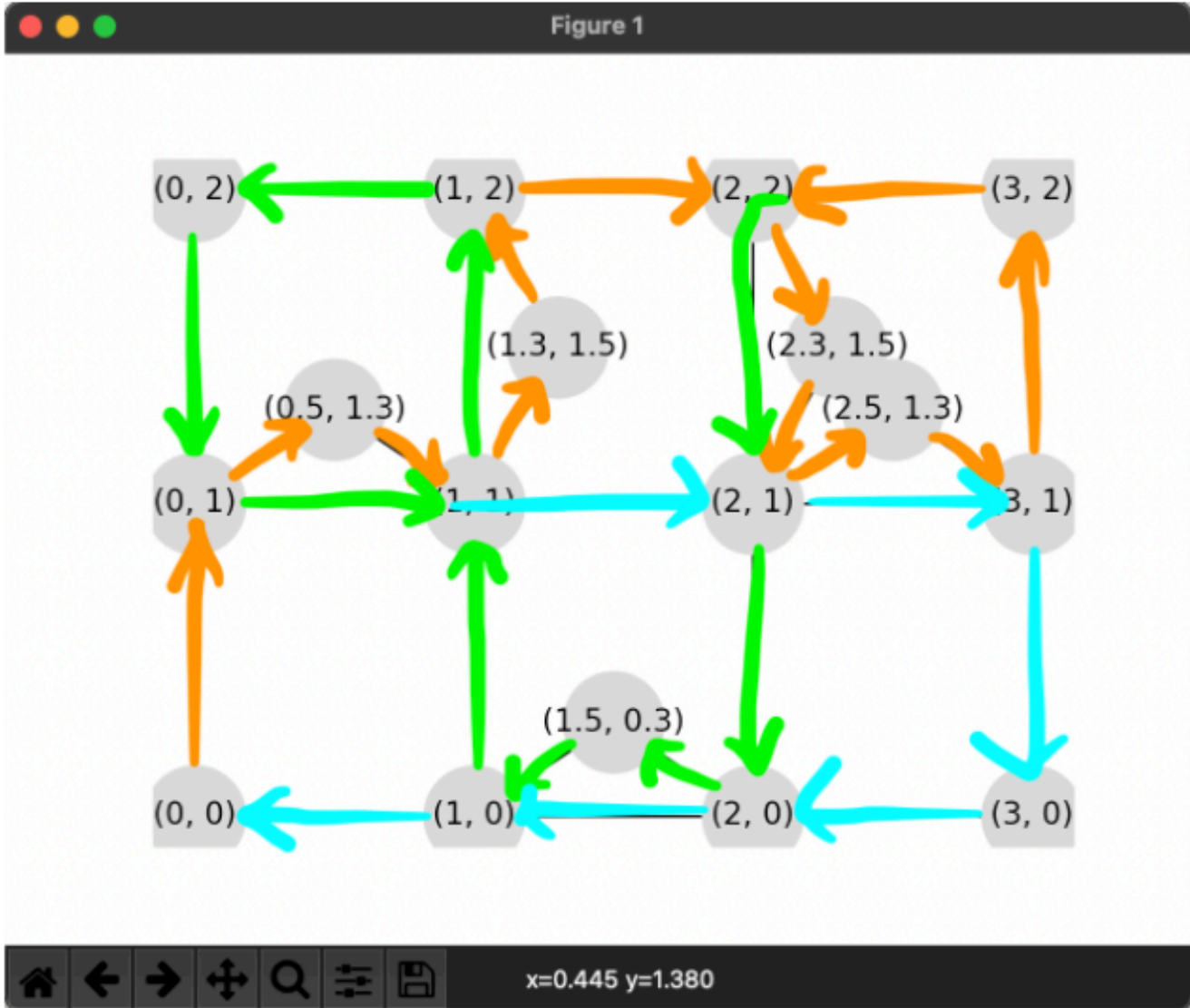
Output 5 出力された最適な経路

```

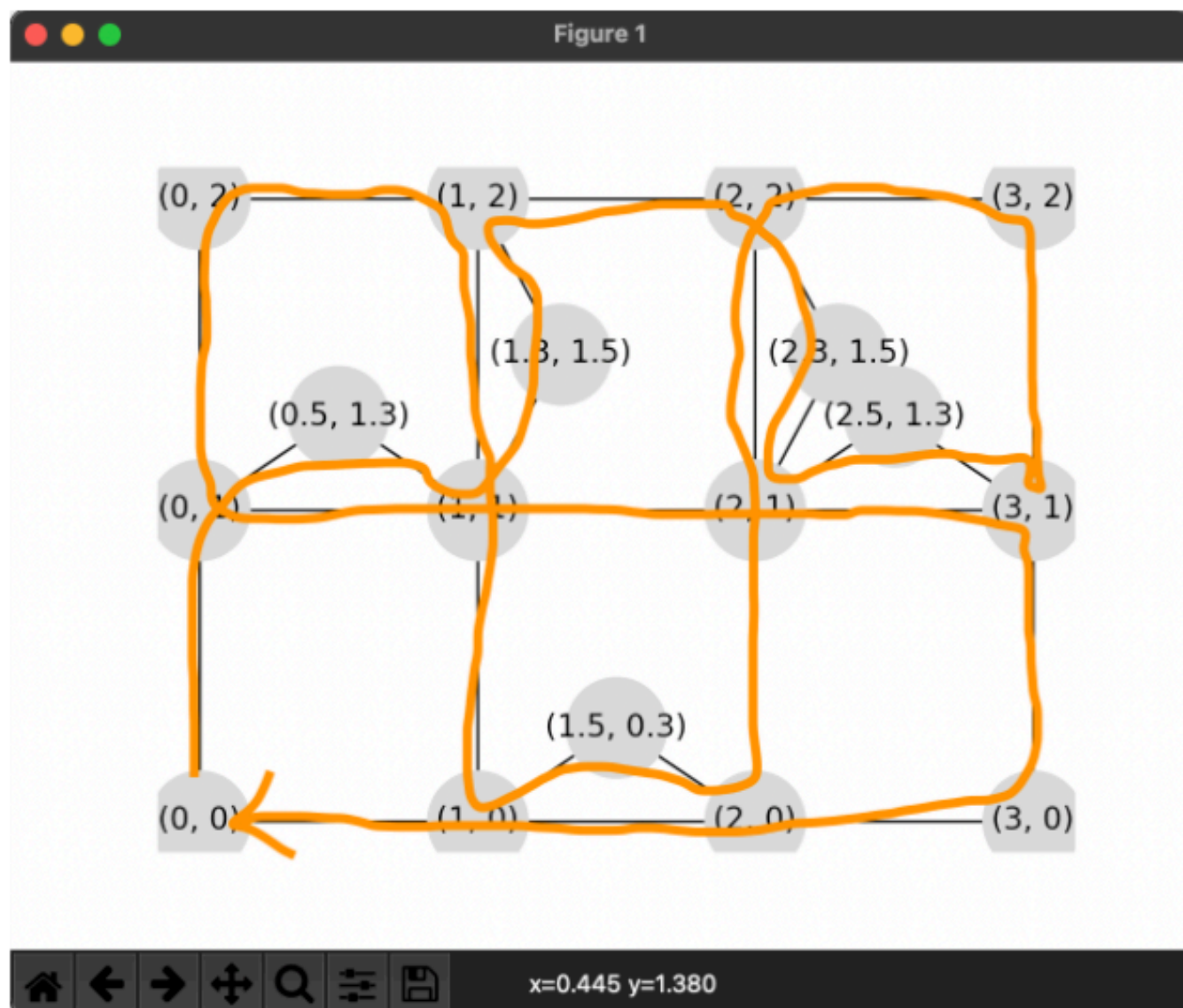
(0, 0)->(0, 1)->(0.5, 1.3)->(1, 1)->(1.3, 1.5)->(1, 2)->(2, 2)->(2.3, 1.5)->(2, 1)->(2.5, 1.3)->(3, 1)->(3, 2)->(2, 2)->(2, 1)->(2, 0)->(1.5, 0.3)->(1, 0)->(1, 1)->(1, 2)->(0, 2)->(0, 1)->(1, 1)->(2, 1)->(3, 1)->(3, 0)->(2, 0)->(1, 0)->end

```

最適解 (オレンジ→緑→青)



一筆書き



# 工夫した点

- ・ 奇点の数で場合分けし，与えられたグラフがオイラーグラフでも，準オイラーグラフでも，それ以外でも解けるようにした
- ・ 各工程を関数化してそれぞれ独立させた
- ・ 入力をコマンドライン引数でできるようにすることで簡単に色々なパターンを試せるようにした
- ・ ランダムで重みをつけるパターンも用意した

# まとめと課題

- ・巡回セールスマン問題が、各頂点を少なくとも一回ずつ回る問題なのに対してこの郵便配達人問題は各辺を少なくとも一回ずつ回る問題である.
- ・この単純な条件の違いで計算量大きく変わり、郵便配達人問題では巡回セールスマン問題ほどの計算は必要とならなかった.
- ・計算量の具体的な比較前にはできなかったものでどれほどのグラフの大きさとどれほどの差が出てくるのかは、課題として残っている
- ・また重み最小の完全マッチングおよび重み最大の完全マッチングは理論にまで詳しく調べられなかったのも課題である.