2017/7/11

# 基礎ゼミ　第15回

# Dialアルゴリズムの実装 -Python 基礎編-

- 土木学会(1998)
  交通ネットワークの均衡分析 第9章 pp.167-171
- Dial, R.B. (1971)
  A Probabilistic Traffic Assignment Algorithm Which Obviates Path Enumeration. Transport Research vol.5 pp.83-111

朝倉研　修士2年
愛甲聡美

# はじめに

今回の目標:

Dialのアルゴリズムを理解し、
Pythonを用いて、ネットワークとOD交通量の入力で
交通量配分を求めるプログラムを書けるようになる

目次

- Dialアルゴリズムとは
- プログラムの流れ
- コーディング

# Dialアルゴリズムとは

- 確率的利用者均衡配分（SUE）の解法のひとつ

**需要固定型利用者均衡配分**
**UE/FD: User Equilibrium assignment with fixed Demand**

- OD交通量所与＆固定
- 最短経路にすべて配分

**需要均衡型利用者均衡配分**
**UE/VD: User Equilibrium assignment with Variable Demand**

- OD交通量が、ネットワークの交通サービスによって変動

**確率的利用者均衡配分**
**SUE: Stochastic User Equilibrium**

- 利用者の経路選択の多様性・不確実性を表現
- どの利用者も経路変更によって旅行時間の短縮を見込めないと思われるときの均衡

**動的利用者均衡配分**
**DUE: Dynamic User Equilibrium assignment**

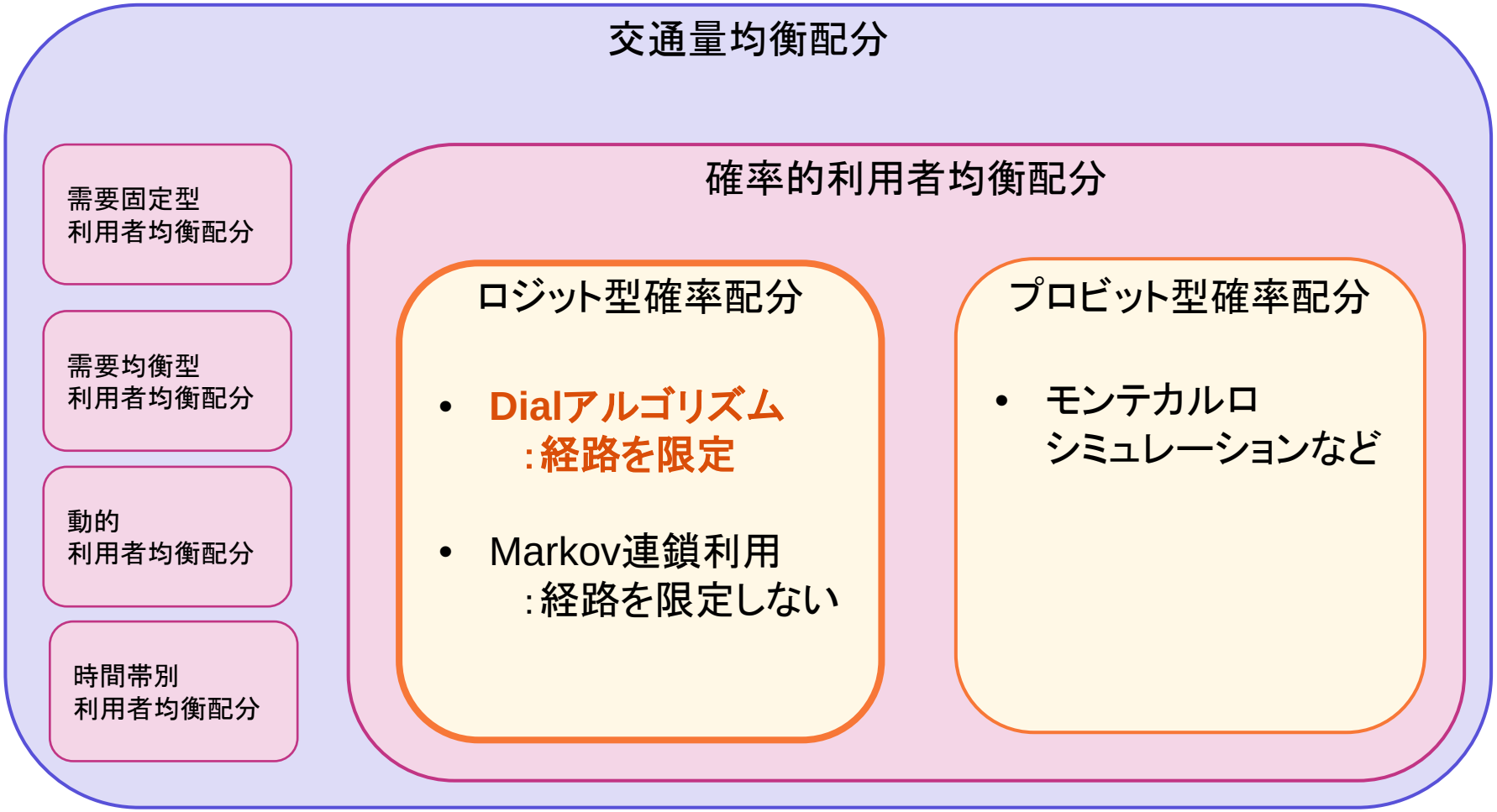
- 出発した車両が到着するまで常にWardropの第一原則が成立

**時間帯別均衡配分**
**TUE: Time-of-day User Equilibrium assignment**

- 時間帯別に利用者均衡が成立
- 前時間帯の残留交通量も加味

# Dialアルゴリズムとは

確率的利用者均衡配分（SUE）の解法のひとつ



# Dialアルゴリズムとは

確率的利用者均衡配分（SUE）の解法のひとつ

- 経路を列挙せずに、ロジット型確率配分計算を実施
- ある基準を満たす「妥当な」経路のみを考慮
- 効率的にロジット型確率配分

ロジット型確率配分　？

- 妥当な経路を選択肢集合としたときの、
  ロジット型配分モデルと同じ交通量パターンを生成する

# Dialアルゴリズムとは

Step 0-A（準備）

- 起点r から他の全ノードへの最小交通費用$c(i)$を計算

$$c(i) \leftarrow C \min[r \rightarrow i]$$

Step 0-B（準備）

- 全リンクについて、リンク尤度$L(i \rightarrow j)$を計算 （$\theta > 0$）

$$L[i \rightarrow j] = \begin{cases} \exp\left[\theta\left\{c(j) - c(i) - t_{ij}\right\}\right] & c(i) < c(j) \\ 0 & otherwise \end{cases}$$

$\theta$:定数。0に近いほど全経路へのランダム配分に、
大きいほど最短経路へのAll-or-nothing配分に。

# 補足

リンク尤度の$\theta$について（Dial（1971）より）

- 縦軸：
  経路の選択確率
- 横軸
  最短経路との差

$\theta$が0に近い

→最短経路も遠回りも
選択確率同じ

$\theta$が大きい

→遠回りの選択確率
低下

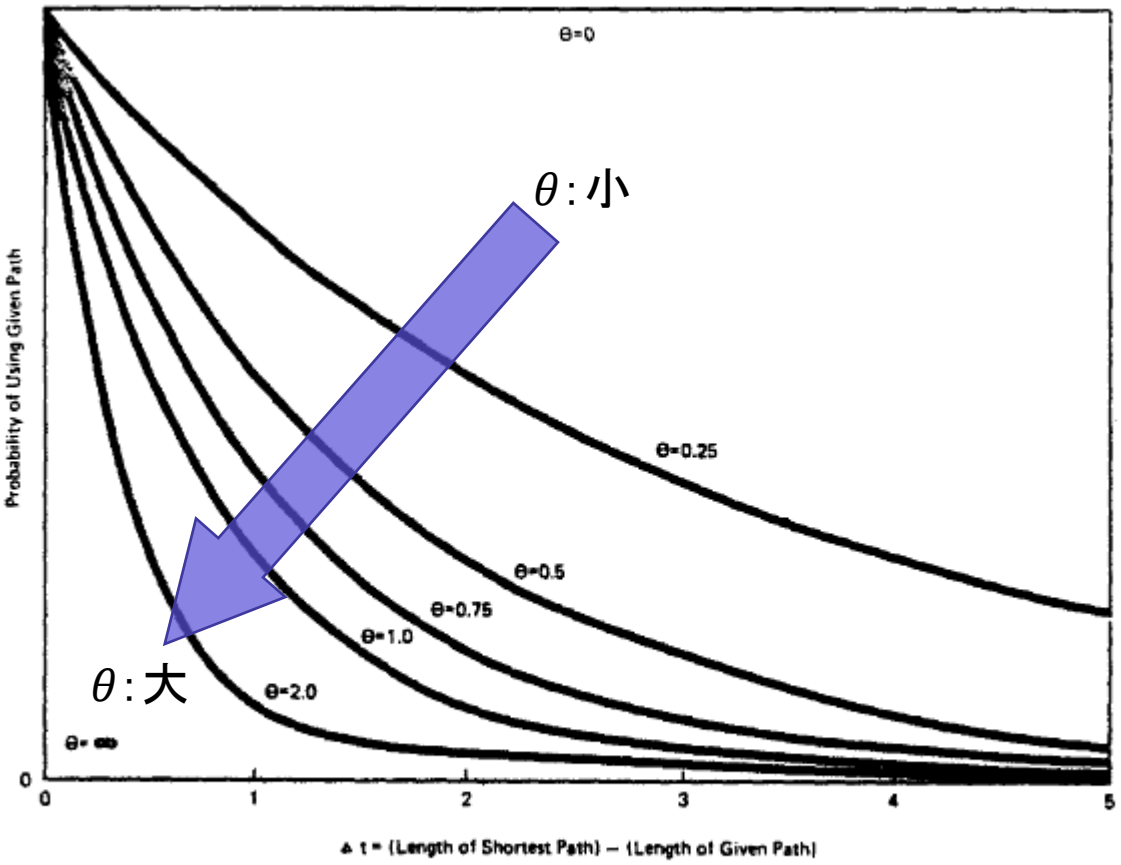



FIG. 9. Probability of using a given path $\Delta t$ longer than shortest path.

Note: The scale of the vertical axis depends on the number and length of competing efficient paths.

# Dialアルゴリズムとは

Step 1（前進処理）

- 起点$r$からの$c(i)$の値の昇順(＝$r$から近い順)にノードを考え、各ノード$i$から流出するリンクのリンク・ウェイトを計算

$$W[i \to j] = \begin{cases} L[i \to j] & for\ i = r \\ L[i \to j] \sum_{m \in I_i} W[m \to i] & otherwise \end{cases}$$

$I_i$:ノード$i$へ流入するリンクの起点集合

# Dialアルゴリズムとは

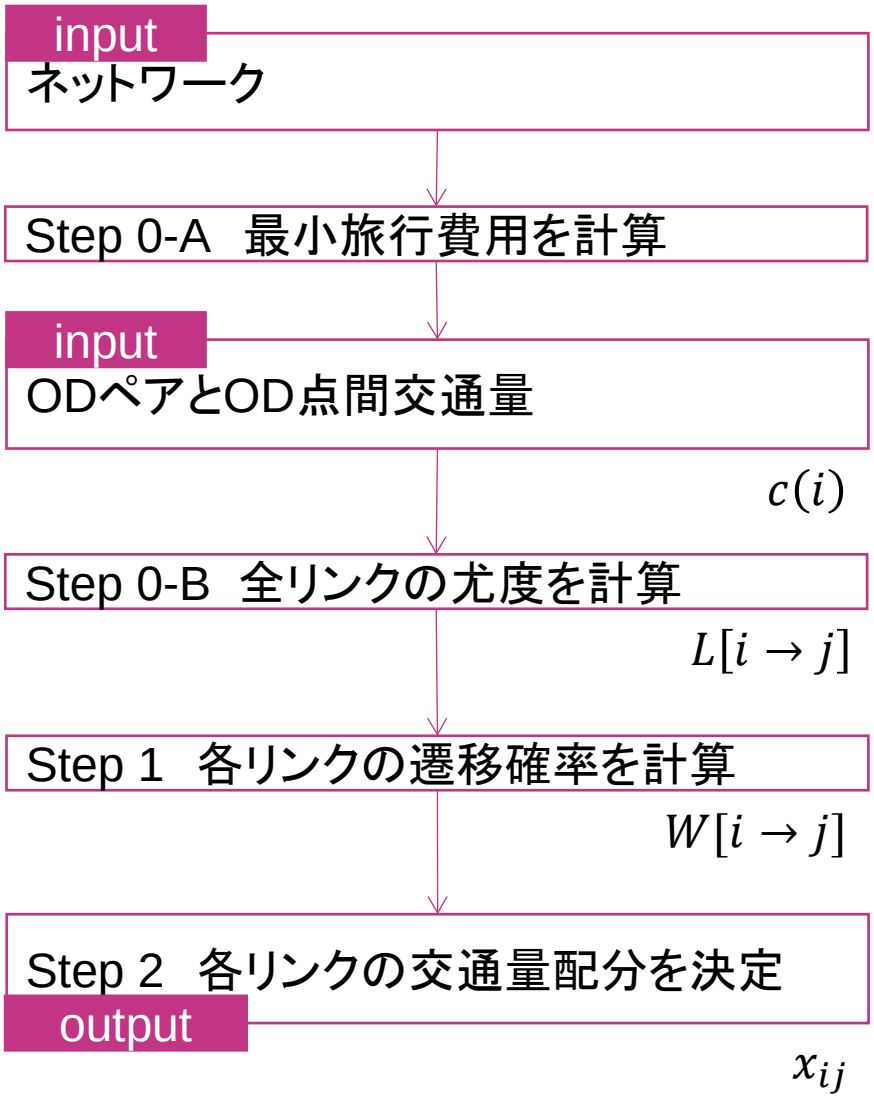
Step 2（後退処理）

- $c(i)$の値の降順(＝$r$から遠い順)にノードを考え、各ノード$j$に流入するリンク交通量$x_{ij}$を次式により計算

$$x_{ij} = (q_{rj} + \sum_{m \in O_j} x_{jm}) \frac{W[i \to j]}{\sum_{m \in I_i} W[m \to j]}$$

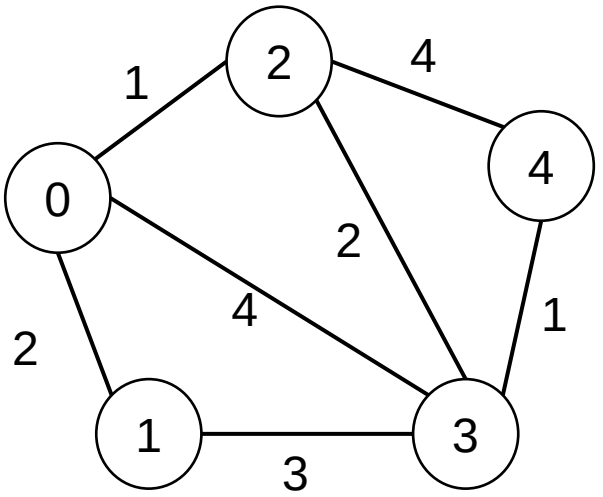
$O_j$:ノード$j$から流出するリンクの終点集合

# プログラムの流れ


input
ネットワーク
Step 0-A 最小旅行費用を計算
input
ODペアとOD点間交通量
$c(i)$
Step 0-B 全リンクの尤度を計算
$L[i \to j]$
Step 1 各リンクの遷移確率を計算
$W[i \to j]$
Step 2 各リンクの交通量配分を決定
output
$x_{ij}$

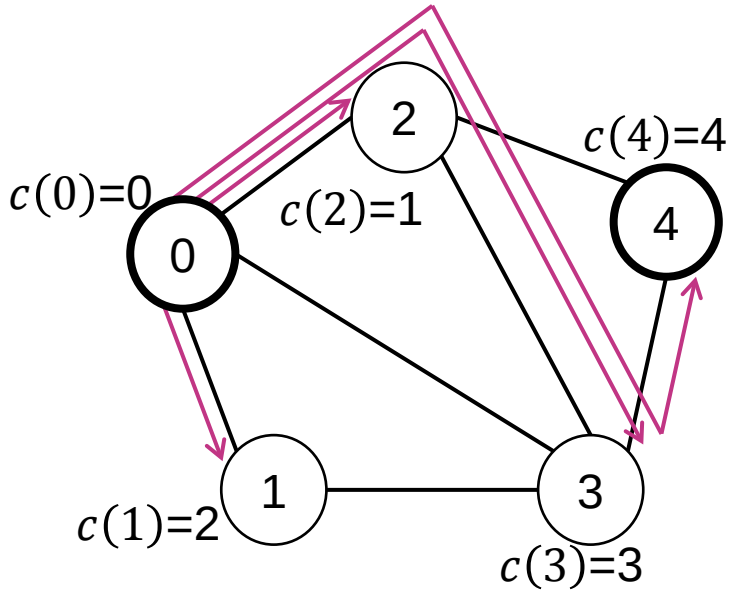
# コーディング

```
###
### Download "Scipy" at
### https://sourceforge.net/projects/scipy/?source=typ_redirect
###

import numpy as np
from math import *
from scipy.stats import rankdata


#####################################################
###step 0-A 起点から全ノードへの最小交通費用を計算######
#####################################################
###0606劉くんのダイクストラ法プログラミングを使います。

#ラベル付き隣接行列:networkを表現
m = [
    [0,2,1,4,0],   # A (0)
    [2,0,0,3,0],   # B (1)
    [1,0,0,2,4],   # C (2)
    [4,3,2,0,1],   # D (3)
    [0,0,4,1,0],   # E (4)
    ]
#初期設定
max_cost = float('inf')       #無限大
m_node = len(m)               #ノード数は隣接行列によって求める
unchecked = [False] * m_node #未確定ノードの集合
cost = [max_cost] * m_node    #起点から各ノードへの最小費用
f_prev = [None] * m_node      #最短経路を列挙するための配列

def search(m,ori,des):      #ネットワークmにおいて起点oriから目的地desまでの最短経路を探す
    cost[ori] = 0           # 起点のノードの距離は０とする
    f_prev[ori] = ori       #起点前のノードは起点とする
    now = ori               #現在地を起点とする

    while True:
        min = max_cost      #変数minは現段階の最小費用を表す（初期は無限大）
        next = -1           #nextは起点から最小費用のあるノードを表す(初期－1)
        unchecked[now] = True
        for i in range(m_node):         #変数iはノード（0~4）
            if unchecked[i]: continue  #ノードが確定しいない場合ループが続く
            if m[now][i]:               #今のノードiと起点(now=ori)が接続しているかどうか
                tmp_cost = m[now][i] + cost[now] #一時的の費用を計算し、最小費用の更新
                if cost[i] > tmp_cost:
                    cost[i] = tmp_cost
                    f_prev[i] = now     #直前のノードに更新
            if min > cost[i]:           #現段階の最小費用と最小費用を持つノードを更新
                min = cost[i]
                next = i
        now = next                      #確定された最小費用持つノードが新らしい"起点"となる
        if next == -1: break            #ノード番号が－1なるまで(すべてノードが確認される)

    print_path(f_prev, cost)            #各接続しているノード間ルートの費用
    return [get_path(ori, des, f_prev), cost[des]]

#結果の出力
def print_path(f_prev, cost):
    for i in range(len(f_prev)):
        print("%d, prev = %d, cost = %d" %  (i, f_prev[i], cost[i]))

def get_path(ori, des, f_prev):
    path = []
    now = des
    path.append(now)
    while True:
        path.append(f_prev[now])      # f_prevには一つ前のノードが入ってる。
        if f_prev[now] == ori: break  #格納されたノードを逆にたどっていけば経路が求められる
        now = f_prev[now]
    path.reverse()
    return path
```



# コーディング

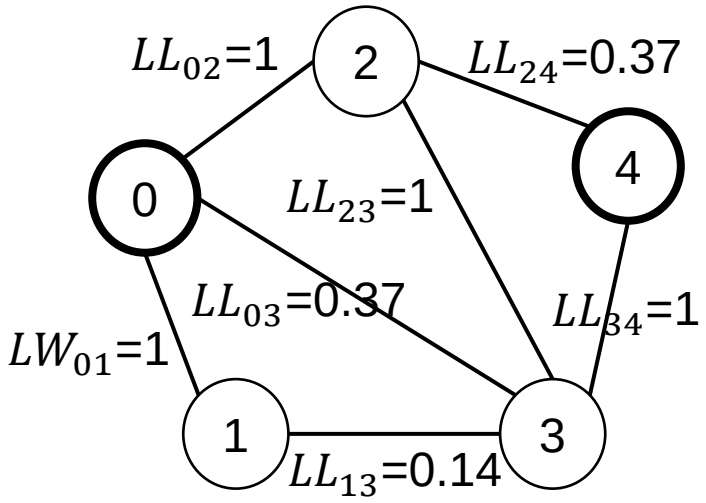
```
###ここからDialのアルゴリズム用

origin=input("Origin?")
destination=input("Destination?")
tf=input("Total flow between O-D?")   #起点-終点の交通量
theta=1

path0, cost0 = search(m, origin, 0)
path1, cost1 = search(m, origin, 1)
path2, cost2 = search(m, origin, 2)
path3, cost3 = search(m, origin, 3)
path4, cost4 = search(m, origin, 4)


print('cost0-0:',cost0)
print('cost0-1:',cost1)
print('cost0-2:',cost2)
print('cost0-3:',cost3)
print('cost0-4:',cost4)
min_cost = [cost0,cost1,cost2,cost3,cost4]
print min_cost

rank = rankdata(min_cost, method='ordinal') #min costを昇順に並べた
print "rank", rank                           時の順位のリストをつくる
```



# コーディング

```
#####################################################
###step 0-B 全リンクの尤度を計算#####################
#####################################################


ll = np.zeros((m_node,m_node))  #リンク尤度の行列：初期値0


for i in range(m_node):
    for j in range(m_node):
        if m[i][j] != 0:        #c(i)<c(j)の場合
            if min_cost[i] < min_cost[j]:
                ll[i][j] = exp(theta*(int(min cost[j])
                               - int(min_cost[i]) - int(m[i][j])))
            else:               #c(i)<c(j)の場合
                ll[i][j] = 0

print "link likelyhood matrix is"
print ll                        #リンク尤度の行列：結果
```

$$L[i \to j] = \begin{cases} \exp\left[\theta\{c(j) - c(i) - t_{ij}\}\right] & c(i) < c(j) \\ 0 & otherwise \end{cases}$$

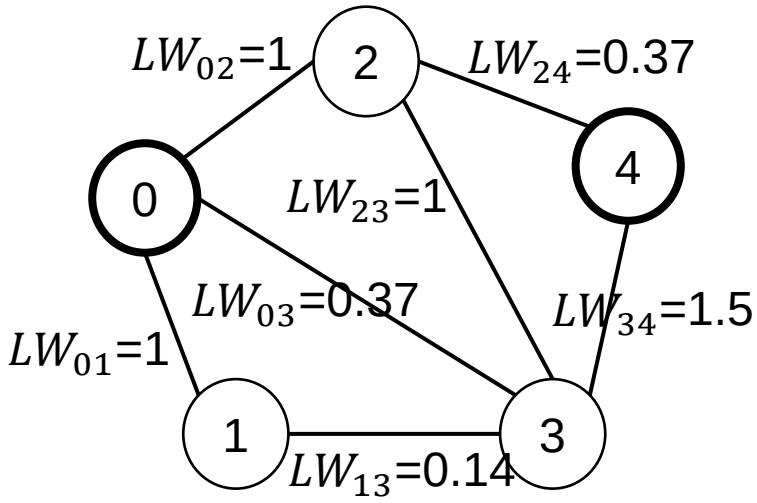



```
link likelyhood matrix is
[[ 0.          1.          1.          0.36787944  0.        ]
 [ 0.          0.          0.          0.13533528  0.        ]
 [ 0.          0.          0.          1.          0.36787944]
 [ 0.          0.          0.          0.          1.        ]
 [ 0.          0.          0.          0.          0.        ]]
```

# コーディング

```
#####################################################
###step 1 各リンクの遷移確率(リンク・ウェイト)を計算####
#####################################################


lw = np.zeros((m_node,m_node))        #リンク遷移確率の行列：初期値0
sum_lw = np.zeros((1,m_node))         #(計算過程用)リンク遷移確率和の行列：初期値0



for r in range(len(rank)):            #起点から近いノードの昇順=rank
    for s in range(len(rank)):
        if rank[s] == r + 1:          #rankに該当するノードsを検索
            for h in range(m_node):
                if m[h][s] != 0:      #該当ノードsへの流入ノード
                    sum_lw[0][s] += lw[h][s]
            for j in range(m_node):
                if s == origin:       #起点の場合
                    lw[s][j] = ll[s][j]
                else:                 #起点以外の場合
                    lw[s][j] = ll[s][j] * sum_lw[0][s]


print "link weight matrix is"
print lw                              #リンク遷移確率の行列：結果
```

$$W[i \to j] = \begin{cases} L[i \to j] & for\ i = r \\ L[i \to j] \sum_{m \in I_i} W[m \to i] & otherwise \end{cases}$$



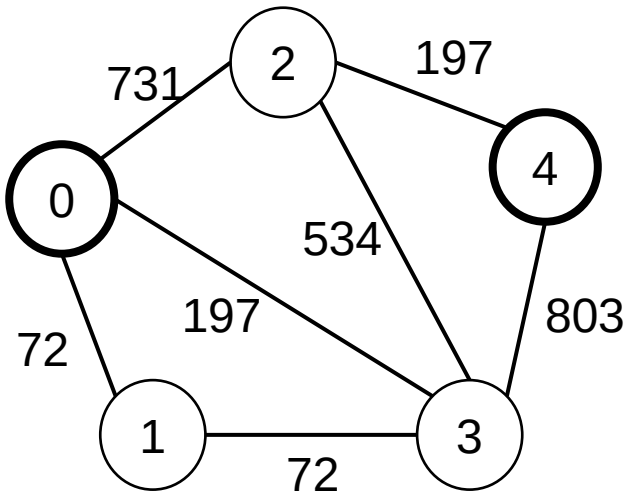

```
link weight matrix is
[[ 0.          1.          1.          0.36787944  0.        ]
 [ 0.          0.          0.          0.13533528  0.        ]
 [ 0.          0.          0.          1.          0.36787944]
 [ 0.          0.          0.          0.          1.50321472]
 [ 0.          0.          0.          0.          0.        ]]
```

# コーディング

```
#####################################################
###step 2 リンクウェイトから、リンク交通量の配分を計算###
#####################################################
```

$$x_{ij} = (q_{rj} + \sum_{m \in O_j} x_{jm}) \frac{W[i \to j]}{\sum_{m \in I_i} W[m \to j]}$$

```
lf=np.zeros((m_node,m_node))                #リンク交通量配分の行列：初期値0
sum_lf = np.zeros((1,m_node))               #(計算過程用)リンク交通量配分和の行列：初期値0

for r in reversed(range(len(rank))):        #起点から近いノードの降順
    for s in reversed(range(len(rank))):
        if rank[s] == r + 1:                #rankに該当するノードsを検索
            for h in range(m_node):
                if m[s][h] != 0:            #該当ノードsからの流出ノード
                    sum_lf[0][s] += lf[s][h]
            for i in range(m_node):
                if sum_lw[0][s] == 0:       #リンク交通量式の分母=0を除外
                    pass
                else:
                    if s == destination:    #終点の場合
                        sum_lf[0][s] = tf
                        lf[i][s] = sum_lf[0][s] * lw[i][s] / sum_lw[0][s]
                    else:                   #終点以外の場合
                        lf[i][s] = sum_lf[0][s] * lw[i][s] / sum_lw[0][s]
```




```
print "link flow matrix is"                 #リンク交通量配分の行列：結果
print lf
```

```
link flow matrix is
[[   0.          72.32948813  731.05857863  196.61193324    0.        ]
 [   0.           0.            0.           72.32948813    0.        ]
 [   0.           0.            0.          534.44664539  196.61193324]
 [   0.           0.            0.            0.          803.38806676]
 [   0.           0.            0.            0.            0.        ]]
>>>
```