

## 第 10 章

# 重み付きグラフ

重み付きグラフの最短経路を求めるアルゴリズムであるダイクストラ法とフロイド法を説明する。

辺に「重み」がついているグラフを重み付きグラフ<sup>\*1</sup>という。経路が含む辺の重みを合計したものが経路の重みである。経路の重みが最小となる経路を最短の経路という。

- ダイクストラ法

ダイクストラ法(10.1節)は、重みを考慮して幅優先探索を行うことで、指定された頂点から各頂点への(重みの合計が)最短の経路を求めるアルゴリズムである。重みが正の場合に適用可能である。

- フロイド法

フロイド法(10.2節)は、行列の積を使って(重みの合計が)最短の経路を求めるアルゴリズムである。負の重みがあっても適用可能である。

### 10.1 ダイクストラ法

この節のサンプルプログラムは <https://lecture.ecc.u-tokyo.ac.jp/~yamaguchi/pc/dijkstra.py> にある。(Jupyter notebook 形式のものは <https://lecture.ecc.u-tokyo.ac.jp/~yamaguchi/pc/dijkstra.ipynb> にある。)

ダイクストラ法では優先度付き待ち行列を使う。優先度付き待ち行列の実現にはヒープ木が使われる所以その意味でも興味深い。

- $s$ : 経路を調べるスタートの頂点
- $d_i$ : 頂点  $s$  から頂点  $i$  に至る最短の経路の重み(になる予定の配列)
- $p_i$ : 頂点  $s$  から頂点  $i$  に至る最短の経路での頂点  $i$  の前の頂点
- $q$ : 優先度付き待ち行列

```

1: procedure DIJKSTRA( $s$ )
2:    $d_i \leftarrow \infty$                                 ▷ 初期値 ( $s$  以外の頂点に対して)
3:    $d_s \leftarrow 0$                                  ▷ 最初の頂点  $s$  には経路(空)の重み = 0 で到達できる。
4:    $p_i \leftarrow \text{未定}$                            ▷ 初期値
5:    $s$  を  $q$  に追加する。
6:   while  $q$  が空でない do
7:      $u \leftarrow q$  の頂点のなかで  $d_u$  が最小の頂点
8:     for  $u$  の隣の頂点  $v$  全てに対して do
9:       if  $d_v > d_u + \text{'u} \rightarrow v \text{ の重み'}$  then
10:         $d_v \leftarrow d_u + \text{'u} \rightarrow v \text{ の重み'}$ 
```

---

<sup>\*1</sup> weighted graph

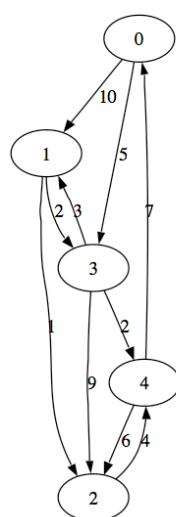
```

11:            $p_v \leftarrow u$ 
12:            $v$  を  $q$  に追加する。
13:       end if
14:   end for
15: end while
16: end procedure

```

辺の重みが全て 1 の場合は幅優先探索と同じになる。

例として以下のグラフを考える。



これを以下のように表現する。

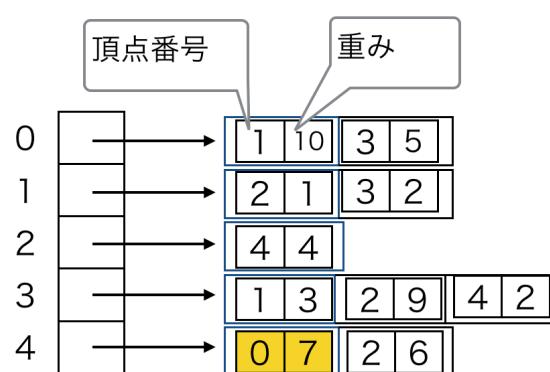
```

1 5
2 0 1 10 # 重みを3列目に追加した
3 0 3 5
4 1 2 1
5 1 3 2
6 2 4 4
7 3 1 3
8 3 2 9
9 3 4 2
10 4 0 7
11 4 2 6

```

### 10.1.1 重み付きグラフの表現

隣接リスト表現を使って重み付きグラフを表現する。



### 10.1.2 Python のプログラム

以下の Python のプログラムで重み付きグラフを読み込む。

```

1 import heapq
2 class Dijkstra:

```

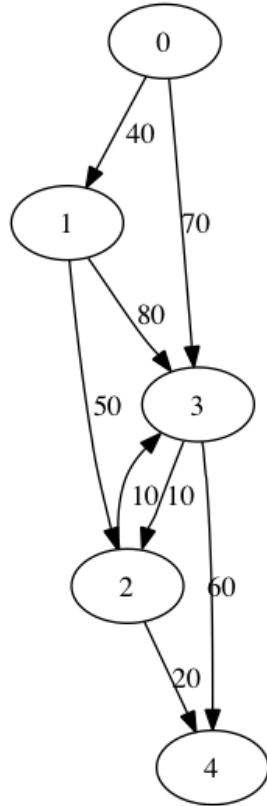
```

3  def __init__(self,g):
4      lines = g.split("\n")
5      self.size = int(lines[0]) # 頂点数。ここでは0からself.size-1まで使う。
6      self.list = [[] for i in range(self.size)]
7      for s in lines[1:]:
8          d1,d2,w = s.split(" ")
9          self.connect(int(d1),int(d2),int(w)) # 重みも登録する
10     def connect(self,x,y,w):
11         self.list[x].append([y,w]) # 重みも登録する
12     def dump(self):
13         for i in range(len(self.list)):
14             for u in self.list[i]:
15                 print(i," ",u[0]," ",u[1])
16     def trav(self,s):
17         d = [float('inf') for i in range(self.size)] # 距離の配列は最初∞に初期化しておく
18         d[s] = 0
19         fmap = [None for i in range(self.size)]
20         pq = [] # 優先度付き待ち行列
21         u = s # sから始める
22         while True:
23             for n in self.list[u]: #隣には(頂点番号、重み)が登録されている
24                 v,weight = n # vが頂点番号、weightが重み
25                 if d[v] > d[u]+weight: # uからvに行った方が重みが少なければ、
26                     d[v] = d[u]+weight # 重みを更新し、
27                     heapq.heappush(pq,[d[v],v]) # (重み、頂点番号)を優先度付き待ち行列に登録
28                     fmap[v] = u # 直前の頂点を登録
29             if len(pq) == 0: # 空になったらwhileから抜ける
30                 break
31             weight,u = heapq.heappop(pq) # (重みと頂点)を取り出す。(重みが小さい順)
32         return [d,fmap] # 重みと直前を返す。
33     def test():
34         g='''5
35         0 1 10
36         0 3 5
37         1 2 1
38         1 3 2
39         2 4 4
40         3 1 3
41         3 2 9
42         3 4 2
43         4 0 7
44         4 2 6''''
45         dk = Dijkstra(g) # 読み込み
46         dk.dump() # 表示
47         d,fmap = dk.trav(0) # Dijkstra法を実行し重みと直前を返す。
48         for i in range(len(d)):
49             print(i,d[i],fmap[i]) # 表示
50     test()

```

優先度付き待ち行列として heapq の heappush と heapq.heappop を使った。heapq では、優先度付き待ち行列として特別の構造を生成する必要はなく、配列を用意しておけばよい。優先度付き待ち行列 pq へのデータ [a,b] の追加は heappush(pq,[a,b]) と行う。この時、a の値が(小さい方が)優先される(heapq の時、先に取り出される)。優先度付き待ち行列 pq からのデータの取り出しは heappop(pq) と行う。取り出されるのは [a,b] である。

問70 以下のグラフで0から4に至る最短経路を求めよ。



問 71 できるだけ更新が多くなるような重み付きグラフを作り、プログラムを適用してみよ。

問 72 重みが自然数の重み付きグラフを考える。頂点  $i$  から頂点  $j$  への辺の重さが  $w$  のとき、頂点  $i$  と頂点  $j$  の間に  $w - 1$  個のダミーの頂点をはさんで、頂点  $i \rightarrow$  ダミー頂点 1  $\rightarrow$  ダミー頂点 2  $\rightarrow \dots \rightarrow$  ダミー頂点  $w-1 \rightarrow$  頂点  $j$  とした(重みの付いてない)有向グラフを作る。このようにして作った重みの付いてない有向グラフでは、通常の待ち行列を使って幅優先探索することで、最短経路と等価なものを求めることができる。優先度付き待ち行列が必要ないのはなぜか。

### 10.1.3 計算量

頂点の数を  $V$ 、辺の数を  $E$  とする。優先度付き待ち行列には最大  $V$  個要素が入る<sup>\*2</sup>。heappush は最大  $V$  回呼ばれる。heappop を二分ヒープで実現していると計算量は  $O(\log V)$  である。1回の heappush の処理は最悪  $O(\log V)$  である。heappush は最大  $E$  回呼ばれる。二分ヒープではなく、フィボナッチヒープを使うと計算量を  $O(V \log V + E)$  にまで下げられる。

## 10.2 フロイド法

この章のサンプルプログラムは <https://lecture.ecc.u-tokyo.ac.jp/~yamaguchi/pc/floyd.py> にある。(Jupyter notebook 形式のものは <https://lecture.ecc.u-tokyo.ac.jp/~yamaguchi/pc/floyd.ipynb> にある。)

重み付きグラフの最短経路<sup>\*3</sup>を求めるアルゴリズムであるワーシャル-フロイド法を説明する。このアルゴリズムでは重みの行列を使うという点で興味深い。

- 頂点  $i$  と頂点  $j$  の間の重み  $w$  を隣接行列の  $i$  行  $j$  列の成分として重み付きグラフを表す。
- 頂点  $i$  と頂点  $j$  の間に辺がないときは隣接行列の  $i$  行  $j$  列の成分を無限大<sup>\*4</sup>とする。

<sup>\*2</sup> 以上の実装では、同一の頂点  $v$  に対して(重み  $w$  の異なる)複数の  $[w, v]$  が優先度付き待ち行列に登録されることがある。 $[w, v]$  を登録する時に、優先度付き待ち行列に同じ  $v$  で既に登録されている場合はその重みを更新すればそれを防ぐことができる。

<sup>\*3</sup> 重みの和が最小のパス

<sup>\*4</sup> python では float['inf'] で無限大を表せる。

```

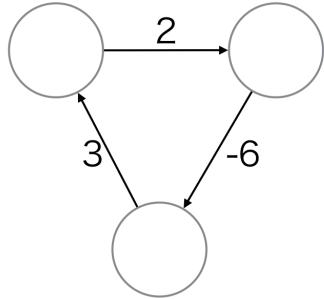
1: procedure FLOYD
2:    $d \leftarrow$  隣接行列
3:   for  $k =$  頂点 1 から頂点  $n$  に対して do
4:     for 各頂点  $i, j$  に対して do
5:       if  $d_{ij} > d_{ik} + d_{kj}$  then
6:          $d'_{ij} \leftarrow d_{ik} + d_{kj}$ 
7:       else
8:          $d'_{ij} \leftarrow d_{ij}$ 
9:       end if
10:    end for
11:     $d \leftarrow d'$ 
12:  end for
13: end procedure

```

各  $k$  に対して、 $d_{ij}$  には、頂点 1 から頂点  $k$  までを経由して<sup>\*5</sup>、頂点  $i$  から頂点  $j$  に至る最短経路がもとめられている。

### 10.2.1 負の重み

ダイクストラ法は重みに負の値があると適用できないが、フロイド法は重みに負の値があっても適用できる。負の重みがある場合は沢山通ればいくらでも重みが減ることがあるので、重みの最小値は  $-\infty$  となるが、そのような状況かが判定できる。



### 10.2.2 Python のプログラム

データはダイクストラ法(10.1節)で使ったものと同じ形式でよい。隣接行列の要素として重みを表現する。

```

1 import numpy as np
2 class AdjDiGraph:
3     def __init__(self,g):
4         lines = g.split("\n")
5         self.size = int(lines[0])
6         self.adjMatrix = np.full((self.size,self.size),float('inf'))
7         for i in range(self.size):
8             self.adjMatrix[i][i] = 0 # 対角成分は0とする。
9         for s in lines[1:]:
10            d1,d2,w = s.split(" ")
11            self.connect1(int(d1),int(d2),int(w))
12    def connect1(self,x, y, w):
13        self.adjMatrix[x][y] = w
14    def floyd(self):
15        d = self.adjMatrix          # 距離行列  $d$  の初期値は隣接行列
16        for k in range(self.size):  # 頂点  $k$  を経由したら短くいけるか調べる
17            newd = np.copy(d)
18            for i in range(self.size):
19                for j in range(self.size):
20                    if d[i][j] > d[i][k]+d[k][j]: # 頂点
21                        i から頂点 j に行くとき頂点 k を経由したら短くいけるなら、
22                        newd[i][j] = d[i][k]+d[k][j] # それを新しい値とする。

```

<sup>\*5</sup> 経由してもよい、という意味で、必要がなければ経由しなくてもよい。

```

22         d = newd # 新しい距離行列を次の距離行列とする。
23     return d
24 def test():
25     wg = '''5
26 0 1 10
27 0 3 5
28 1 2 1
29 1 3 2
30 2 4 4
31 3 1 3
32 3 2 9
33 3 4 2
34 4 0 7
35 4 2 6'''
36     g = AdjDiGraph(wg)
37     print(g.adjMatrix)
38     dm = g.floyd()
39     print("distance matrix")
40     for d in dm:
41         print(d)
42 test()

```

`np.full(d,v)` では、大きさ `d`、各要素の値が `v` の行列を作る。2次元行列の場合、大きさ `d` には幅 `w` と高さ `h` の両方を指定する必要があるので、`d` の所に `(w,h)` と指定する。`(w,h)` はリストの `[w,h]` とほぼ同じ意味であるが、リストと違い後から値が変更されない (`immutable`) もので、タプルと呼んでいる。

`float('inf')` は実数 (浮動小数) の無限大<sup>\*6</sup>となる。

`np.copy` は行列をコピーする。

このプログラムでは行列のかけ算を使うわけではないので、numpy の行列で値を表現するメリットはそれほどないが、`np.full` や `np.copy` などが使えるのはメリットであろう。print するだけで行列らしい表示がされるのも便利である。

### 10.2.3 フロイド法の計算量

フロイド法のプログラムは以下のように三重の繰り返しがあるので、頂点の数  $V$  に対して  $O(V^3)$  の計算量となる。

```

1 for k in range(size)
2 ...
3     for i in range(size)
4         for j in range(size)
5 ...
6         end
7     end
8 end

```

行列の積は  $O(V^3)$ <sup>\*7</sup>であり、それを  $V$  回繰り返すと  $O(V^4)$  となるので、フロイド法は行列の積を  $V$  回繰り返すわけではないことに注意する。

それで正しく求まるのかを考えてみよう。頂点  $k$  を経由するのを  $k = 0, 1, \dots, N - 1$  の順に 1回だけ調べているが足りているのだろうか？例えば、 $1 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 4$  がこれが最短経路のケースを考える。 $k=2$  の時に、5 から 4 に行くとき 2 を経由するのが最短とわかる。 $k=3$  の時に、1 から 5 に行くとき 3 を経由するのが最短とわかる。 $k=5$  の時に、1 から 5、5 から 4 に行くのが 1 から 4 に行くときの最短とわかる。

問 73 全ての場合に最短経路がもとまることを証明せよ。

### 10.2.4 経路を求める

次に経路を求めるために、頂点  $i$  から頂点  $j$  に最短経路で行くときの頂点  $j$  の直前の頂点を  $\text{pi}[i][j]$  に記録することにする。 $\text{pi}$  では、例えば、 $1 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 4$  という最短経路は以下のように表現されることになる。

- $\text{pi}[1][4] \dots 2$
- $\text{pi}[1][2] \dots 5$

\*6 浮動小数の表現として一般的に使われている IEEE754 で定義されている。

\*7 高速化の技法は幾つかあるが単純に書いた場合。

- $\text{pi}[1][5] \dots 3$
- $\text{pi}[1][3] \dots 1$

経路も求めるフロイド法のPythonのプログラムは以下のようになる。

```

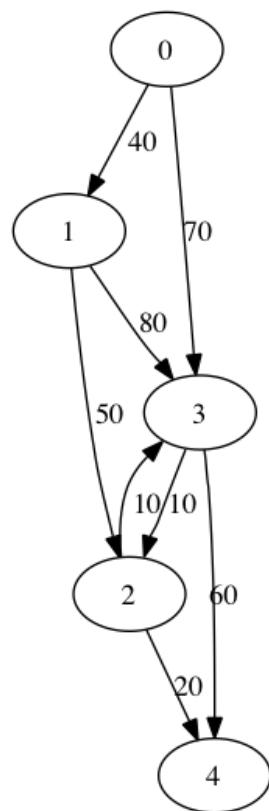
1 import numpy as np
2 import copy
3 class AdjDiGraph:
4     def __init__(self,g):
5         lines = g.split("\n")
6         self.size = int(lines[0])
7         self.adjMatrix = np.full((self.size,self.size),float('inf'))
8         for i in range(self.size):
9             self.adjMatrix[i][i] = 0 # 対角成分は$0$とする。
10        for s in lines[1:]:
11            d1,d2,w = s.split(" ")
12            self.connect1(int(d1),int(d2),int(w))
13    def connect1(self,x, y, w):
14        self.adjMatrix[x][y] = w
15    def floyd(self):
16        pi = np.full((self.size,self.size),-1)
17        for i in range(self.size):
18            for j in range(self.size):
19                if self.adjMatrix[i][j] != float('inf'):
20                    pi[i][j] = i # iとjが直接つながっていたら jの直前は iとして良い
21        d = self.adjMatrix
22        for k in range(self.size):
23            newd = np.copy(d)
24            newpi = np.copy(pi)
25            for i in range(self.size):
26                for j in range(self.size):
27                    if d[i][j] > d[i][k]+d[k][j]:
28                        newd[i][j] = d[i][k]+d[k][j]
29                        newpi[i][j] = pi[k][j] #
                           kを経由していく場合は jの直前を kから jに行く場合の直前とする。
30                        # kとjは直接つながってないことがあるので、 jの直前を kとしてはいけない。
31        d = newd
32        pi = newpi
33        return [d,pi]
34    def test():
35        wg = '''5
36        0 1 10
37        0 3 5
38        1 2 1
39        1 3 2
40        2 4 4
41        3 1 3
42        3 2 9
43        3 4 2
44        4 0 7
45        4 2 6'''
46        g = AdjDiGraph(wg)
47        print(g.adjMatrix)
48        dm,pi = g.floyd()
49        print("distance matrix")
50        print(dm)
51        print("pi")
52        print(pi)
53    test()

```

$\text{pi}$ も行列で表現しているが、初期値を  $\text{np.full}$  で-1としているため、整数の配列となっていることに注意する。

フロイド法の動作は <http://lecture.ecc.u-tokyo.ac.jp/~yamaguch/graph/floyd-sample.pdf> のようになる。

問74 フロイド法で以下のグラフで0から4に至る最短経路を求めよ。



問 75 負の重みがある場合、どのようになるかを調べよ。

### 10.3 計算量

#### 10.3.1 1 頂点-全頂点間

ダイクストラ法では2頂点間だけでなく、ある頂点から出発して他の頂点すべてへの最短経路が求められる。二分ヒープによるダイクストラ法の計算量は  $O((V + E) \log V)$  である。フィボナッチヒープによるダイクストラ法の計算量は  $O((V \log V) + E)$  となる。

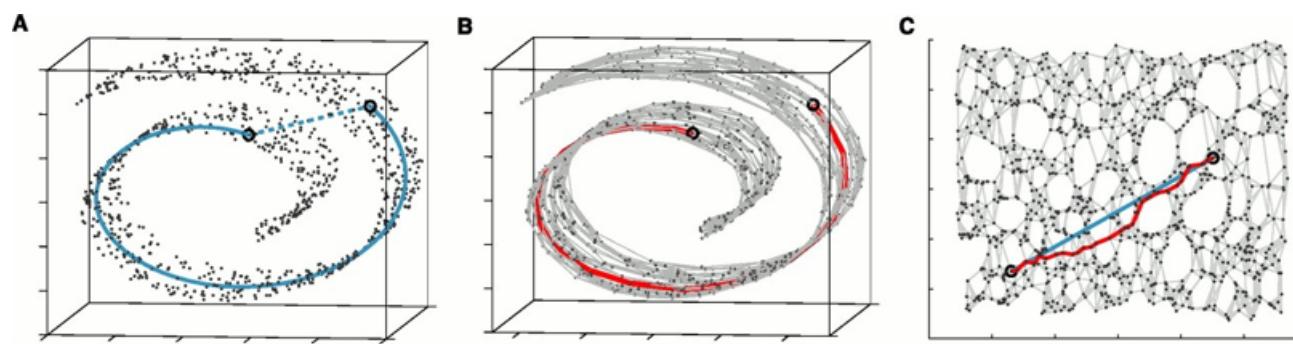
#### 10.3.2 全頂点-全頂点間

フロイド法では全頂点間の最短経路が求められる。フロイド法の計算量は  $O(V^3)$  である。あらゆる頂点から出発してダイクストラ法を繰り返すと全頂点間の最短経路が求められる。二分ヒープによるダイクストラ法を使った場合の計算量は  $O(V(V + E) \log V)$ 、フィボナッチヒープによるダイクストラ法を使った場合の計算量は  $O(V^2 \log V + VE)$  である。 $E$  の大きさは最大では  $V^2$  なので、 $O(V^2 \log V + VE) = O(V^2 \log V + VV^2) = O(V^3)$  となり、フロイド法と同じ計算量となるが、疎なグラフ(辺の数が少ない)ではフロイド法より有利である。

##### 10.3.2.1 Isomap

全頂点-全頂点間の最短経路を求める応用としては **Isomap**<sup>\*8</sup>がある。Isomap は高次元空間内の低次元多様体を低次元に射影する方法である。

<sup>\*8</sup> A Global Geometric Framework for Nonlinear Dimensionality Reduction, Joshua B. Tenenbaum, Vin de Silva, and John C. Langford, Science 22 December 2000: 290 (5500), 2319-2323. [DOI:10.1126/science.290.5500.2319]



A : 高次元中の多様体上の点

B : 近傍の点<sup>\*9</sup>を辺で結ぶとグラフができる。そのグラフの最短経路で 2 点間の大圈距離<sup>\*10</sup>をダイクストラ法を使って求める。

C : MDS を使って大圈距離を保存するように低次元に射影する。

<sup>\*9</sup> 近い方から  $k$  点を選ぶ。 $k$  は指定する。

<sup>\*10</sup> geodesic distance