基礎ゼミ第11回

最短経路探査アルゴリズムの実装

朝倉研 修士１年　劉　彬

2017/06/06

OUTLINE

- 最短経路探査アルゴリズムの概要(8章.1)
  - ダイクストラ法（ラベル確定法）
  - ラベル修正法
- 簡単な例を用いてプログラミング
  - ダイクストラ法

最短経路探索アルゴリズム

最短経路探索法：

- 利用者均衡配分の計算アルゴリズムの一部
- 効率が良い：
  ダイクストラ法とラベル修正法

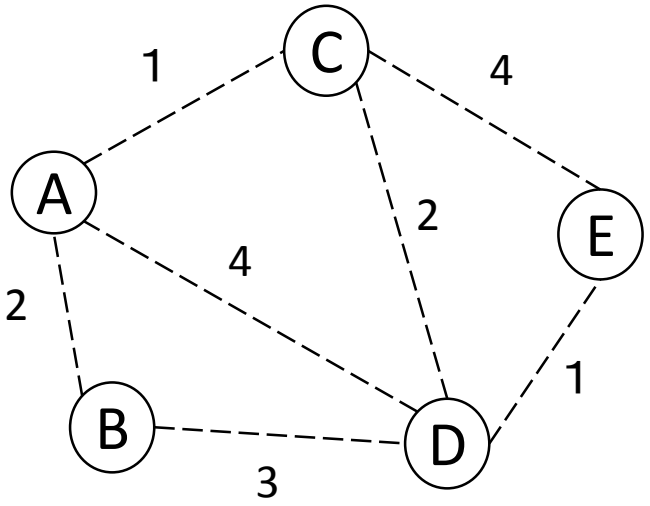
1つの起点からすべての終点までの最短経路と最小費用を１回の計算で同時に求める方法

１ペアの起終点間の最短経路を一回一回求めていくものではない

ダイクストラ法

○　：ノード
3　：リンクと費用

- 起点ノードにより近いノードから順に、全方向に向かって最短経路を列挙していく方法



アルゴリズム

■ノードの集合

K
起点からの最短経路及び最小費用(permanent label)が確定

起点を中心として一つづつ外周部に広がり、拡大していく

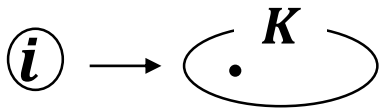
$\bar{K}$
起点からの最小費用(temporary label)が確定されていない

ノードjが持つ部分的最小費用c_jは起点からノードjまでの現段階の最短路費用を表し、計算とともに更新していく

ダイクストラ法

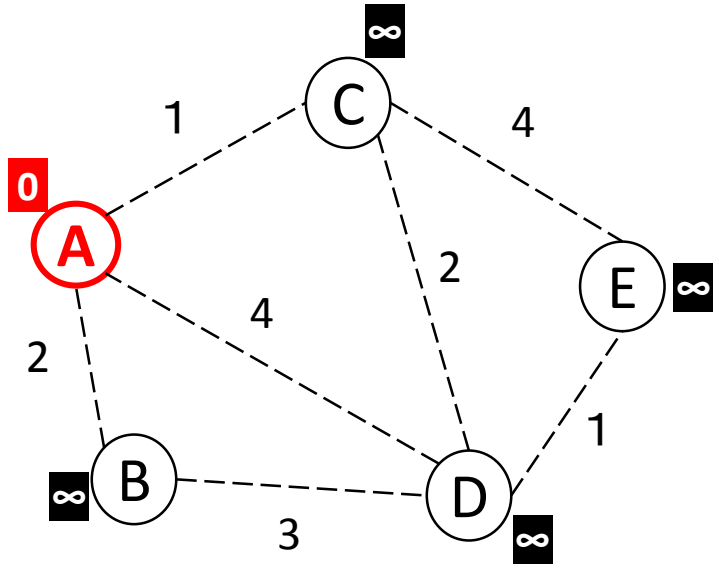
□手順：

① 前回の計算stepでのノードiの最小交通費用c_iが確定された場合，



ステップ１として起点ノードAからAまでの最小費用は０、確定されて集合Kに移す。

起点から各ノードの部分的最小費用（暫定値）c_m=∞とする。



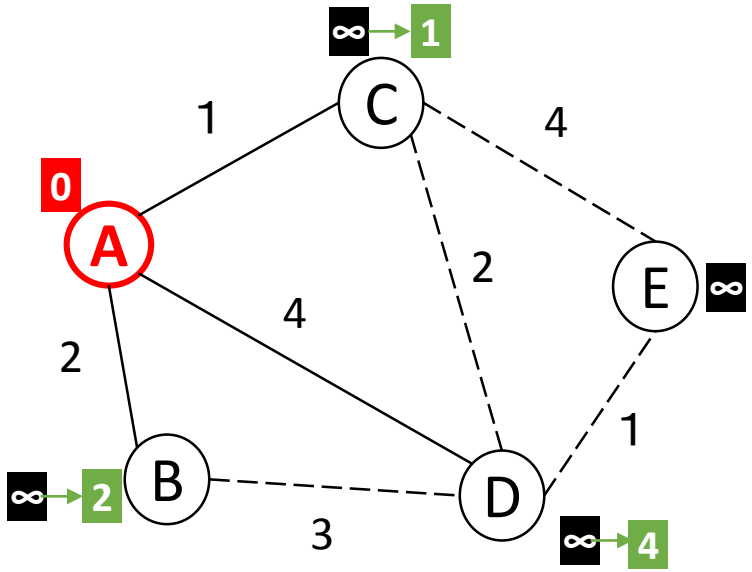
赤色：確定済のノード，
リンク，最小費用
黒：未確定の各項
緑：最小費用の更新

ダイクストラ法

□手順：

② そのノードiから出る各リンクの終点ノード{m}のそれぞれの一時的最小費用$c'_m = c_i + t_{im}$を計算
もし$c'_m < c_m$ の場合$c_m = c_i + t_{im}$で更新.

最短経路列挙のための変数
(先行ポインタ)$F_m = i$



- 確定済みノードAから経路をたどり暫定値(一時的最小費用)を更新：

Aから直接行けるCDBまでの最小費用は1,4,2<∞により更新された.

- 最短経路列挙：

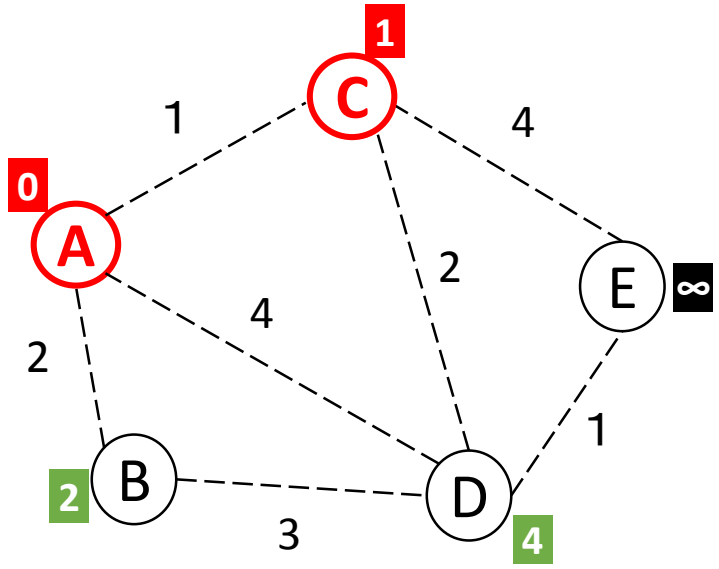
$F_c = A, F_D = A, F_B = A$

ダイクストラ法

□手順：

③ これらのノード{m}を含めて現ステップまでに計算された最小交通費用は集合$\bar{K}$から，現段階で最も小さい一時的最小費用をもつノードjを求める(j∈ $\bar{K}$)

$$c_j = \min_p(c_p) \quad \{p \in \bar{K}\}$$



BCDは未確定の集合$\bar{K}$に所属し，最小費用のノードCを確定する

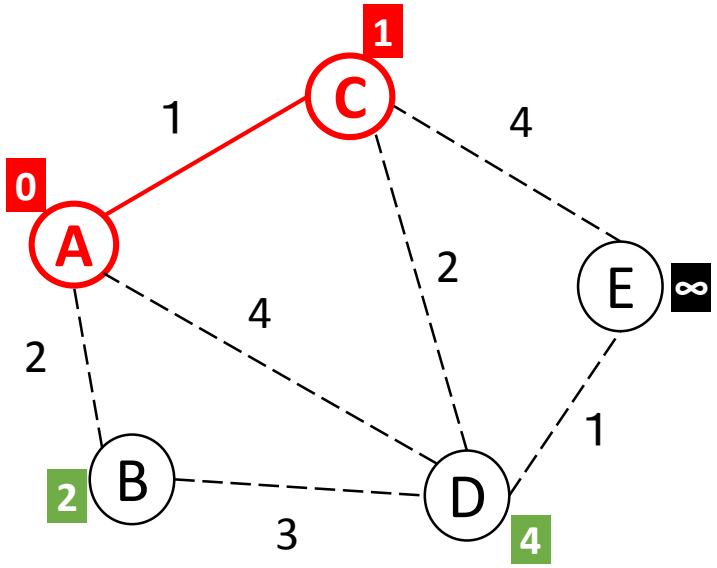
ダイクストラ法

□手順：

④ 起点からの最小交通費用c_iを持つ確定されたノードｊを確定済集合Kに移す

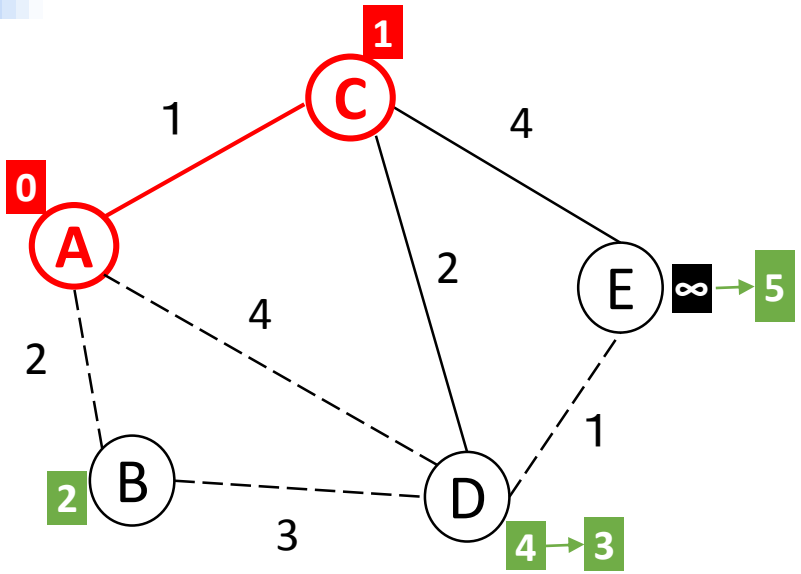
この時最小交通費用はc_j

ノードCが確定済集合Kに移され、
現在最小交通費用は0＋1=1



⑤ ノードｊを*i*に置き換えて②～⑤を繰り返す

ダイクストラ法

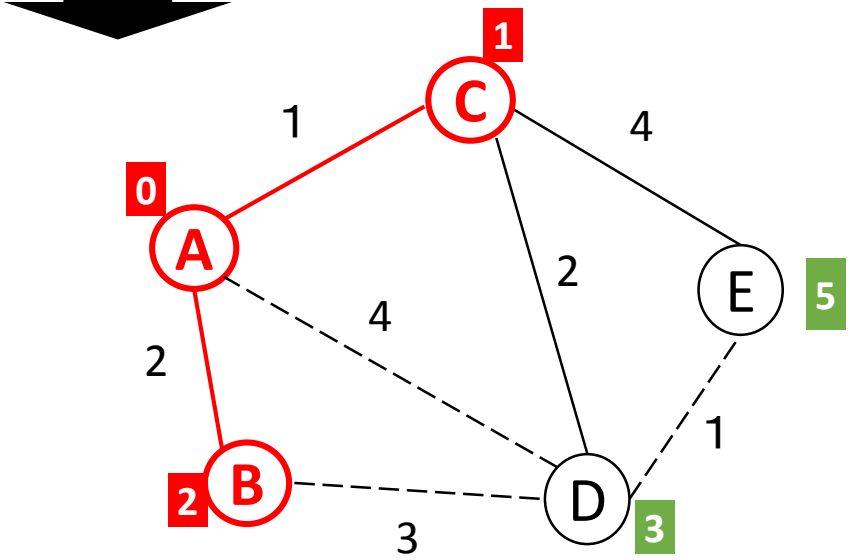


- 確定済のノードCから経路たどり暫定値を更新していく：

A-Dの４よりA-C-Dの３の方が小さいため，３に更新される

- 最短経路列挙：

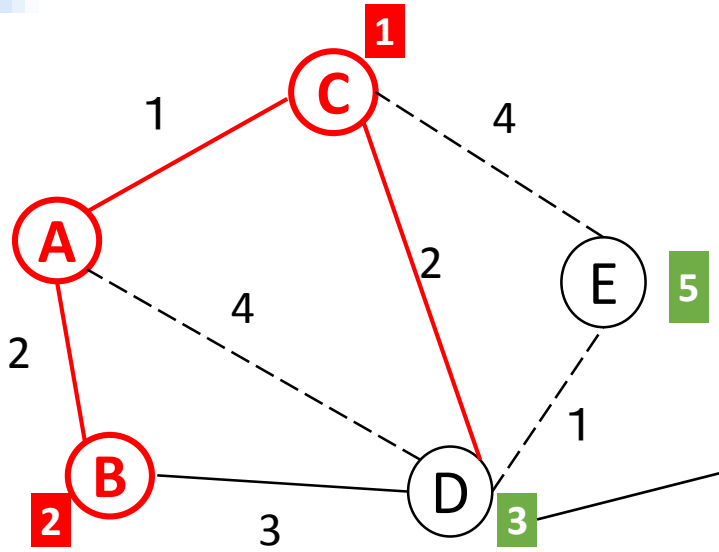
$F_E = C, F_D = C$



未確定BDE中の最小費用をもつノードを確定させる：

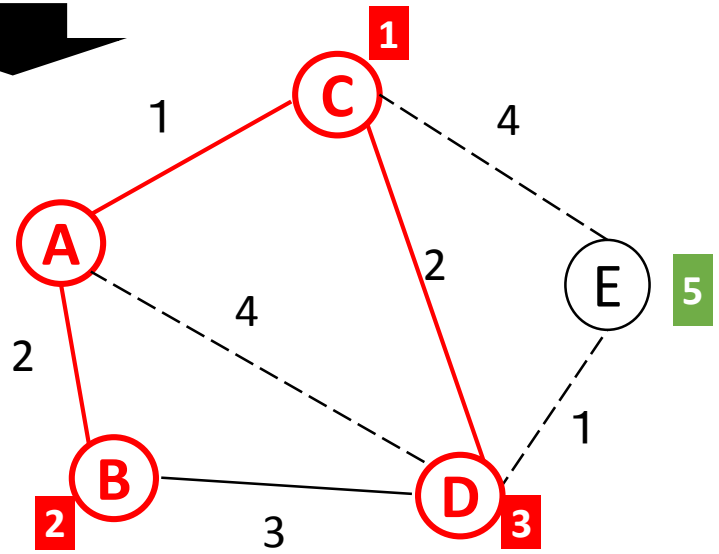
ノードBが最小により確定

ダイクストラ法



確定済のノードBから経路をたどり暫定値を更新：

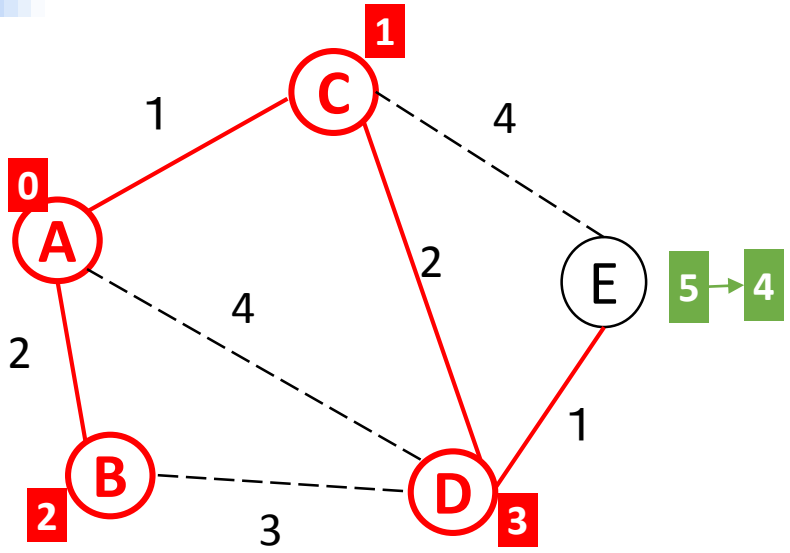
以前のA-C-Dの３よりA-B-Dの５の方が大きいため更新されない

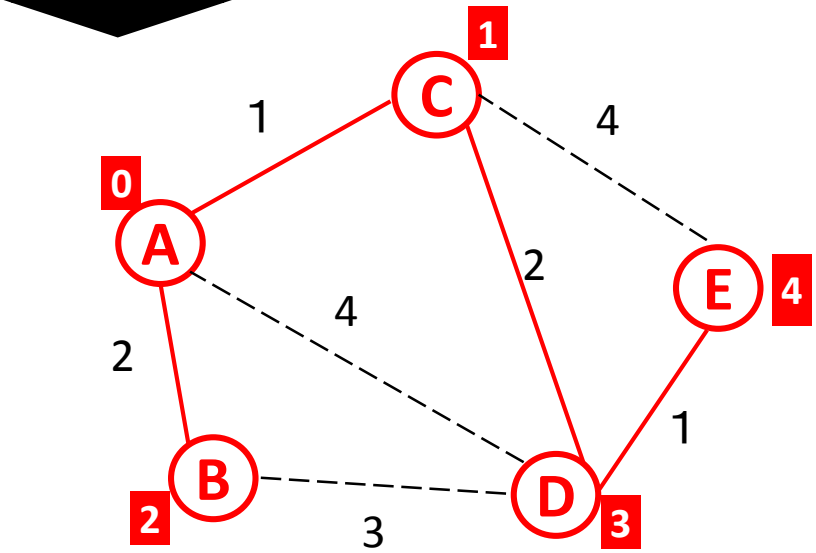


未確定DE中の最小費用をもつノードを確定させる：

ノードDが最小により確定

ダイクストラ法





- 確定済のノードDら経路をたどり暫定値を更新：

以前A-C-Eの５よりA-C-D-Eの４の方が小さいため更新される

- 最短経路列挙：

$F_{\mathrm{E}} = \mathrm{D}$

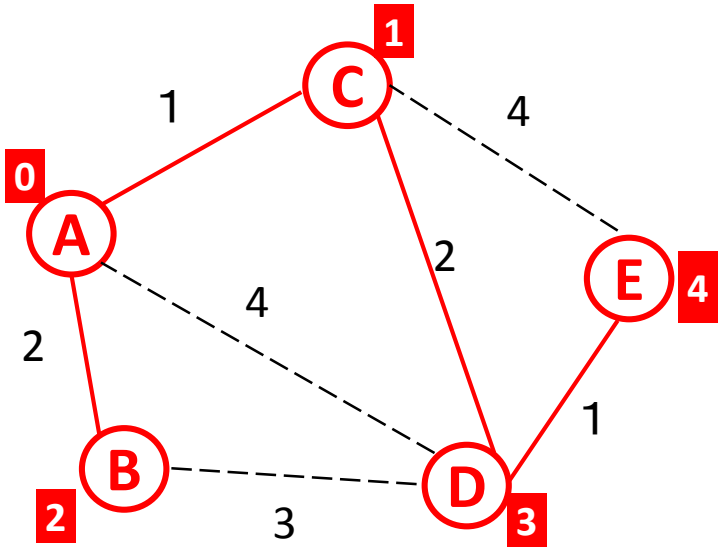
未確定ノードの中最小費用をもつノードを確定させる：

ノードEのみ

全てのノードが確定済集合Kに移した

→**終了**

ダイクストラ法



結果_最短経路：

先行ポインタF_mによる任意のノードdから起点oを逆に戻りながら列挙する

$d \rightarrow (F_d=)h \rightarrow (F_h=)g \rightarrow (F_g=)f \rightarrow \cdots \rightarrow (F_b=)$起点 o

$$\begin{matrix} F_C = A \\ F_D = A \\ F_B = A \end{matrix} \longrightarrow \begin{matrix} F_E = C \\ F_D = C \end{matrix} \longrightarrow F_{\mathrm{E}} = \mathrm{D}$$

・A→Eまでの最短経路：
E→$(F_{\mathrm{E}} =)\mathrm{D} \rightarrow (F_D=)C \rightarrow (F_c=)A$

・最小費用：４

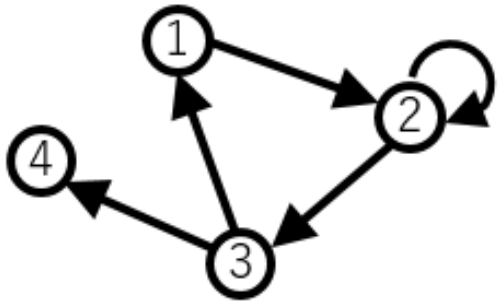
手順のまとめ

1. 起点の最小距離を０，ほかのノードの値を未定義(∞)に設定
2. まだ確定されていないノードのうち，最小値をもとノードを見つけ，確定ノードとする.
3. 確定ノードからの伸びているルートをそれぞれ確認し，「起点から確定ノードまでの費用＋ルートの費用」を計算し，そのノードの現在の最小費用より小さいければ更新していく
4. 全てのノードの起点からの費用が確定していなければ，STEP２に戻る

プログラミング

経路図を表すデータ構造：**隣接行列**

- 各ノード間の隣接関係を行列で表す
- 経路の有無は0と1で表す行列



隣接行列

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

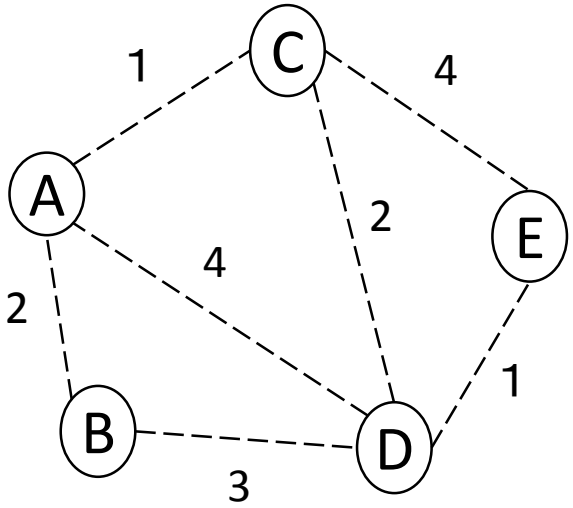
ノード1からノード2へリンクがあるので、12成分（1行目の2列目）が1となる

- 各ルートに対応する費用を行列に格納した行列

  → ラベル付き隣接行列

ノード数：5
5×5のラベル付き隣接行列

$$\begin{pmatrix} 0 & 2 & 1 & 3 & 0 \\ 2 & 0 & 0 & 3 & 0 \\ 1 & 0 & 0 & 2 & 4 \\ 3 & 3 & 2 & 0 & 1 \\ 0 & 0 & 4 & 1 & 0 \end{pmatrix}$$



初期設定

アルゴリズム本体

結果出力設定

```
#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)
    [3,3,2,0,1],   # 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っけば経路が求められる
        now = f_prev[now]
    path.reverse()
    return path


path, cost = search(m, 0, 4)
print('path:',path)
print('cost:',cost)
```

0から数える

先行ポインタ$F_m = i$

temporary label

起点０と終点４を変えてみよう