# 平成30年度
# Fundamental Seminar
# 経路探索アルゴリズム

東京工業大学工学部土木環境工学科
朝倉研究室　長崎滉大

# 目次

# はじめに

- 目的

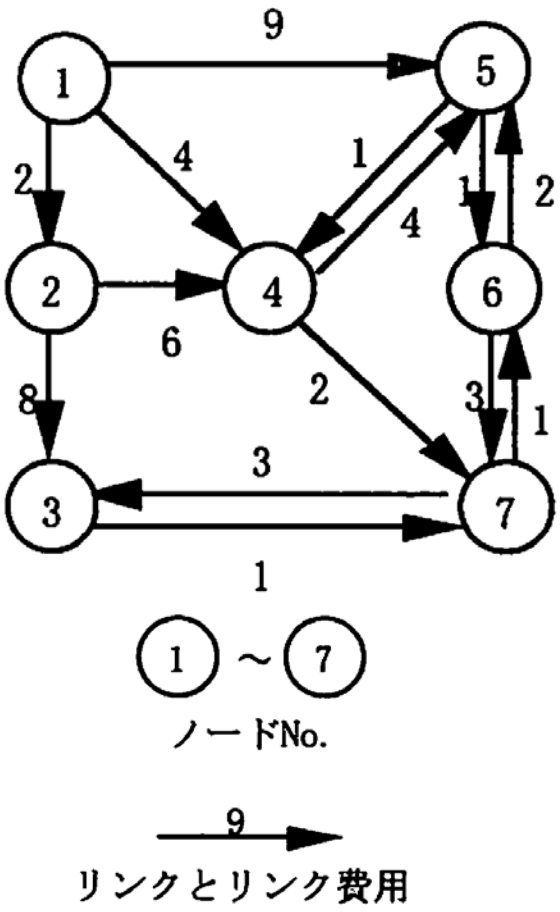
交通ネットワークの問題を解くに当たって最も基本となる最短経路探索について学習し、この先の利用者均衡配分などに応用できるようにする。

# 経路探索

- 例えば、右のようなネットワークがあるとして、例えばノード1から7に行くにはいろいろな経路が考えられる。

1→4→7や1→2→4→7、1→4→5→6→7など

- 特に理由もないのに遠回りする人はあまりいない、すなわち大部分の人間が一番短いルートを用いる。
- したがって最短経路を探索することは交通計画において重要視される。

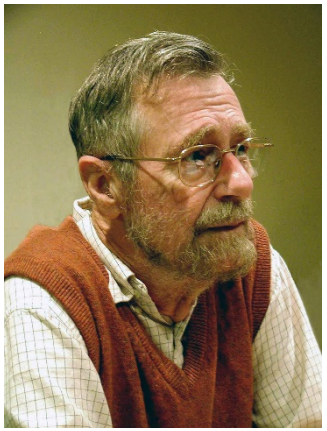


# 最短経路探索アルゴリズム

- 代表的な二つ
- **Dijkstra法(ラベル確定法)**
- **ラベル修正法**

- ある一つの起点からすべての終点までの最短経路とその費用を同時に求められる方法。
- ラベル(後述)を確定させていくためラベル確定法ともよばれる。
- すべてのリンクコストが非負である時にのみ用いることができる。

- おおまかにはDijkstra法と同じ。
- ラベルが修正されうることからラベル修正法と呼ばれる。
- こちらはリンクコストが負であっても使える。



Edsger Dijkstra

# Dijkstera法とは

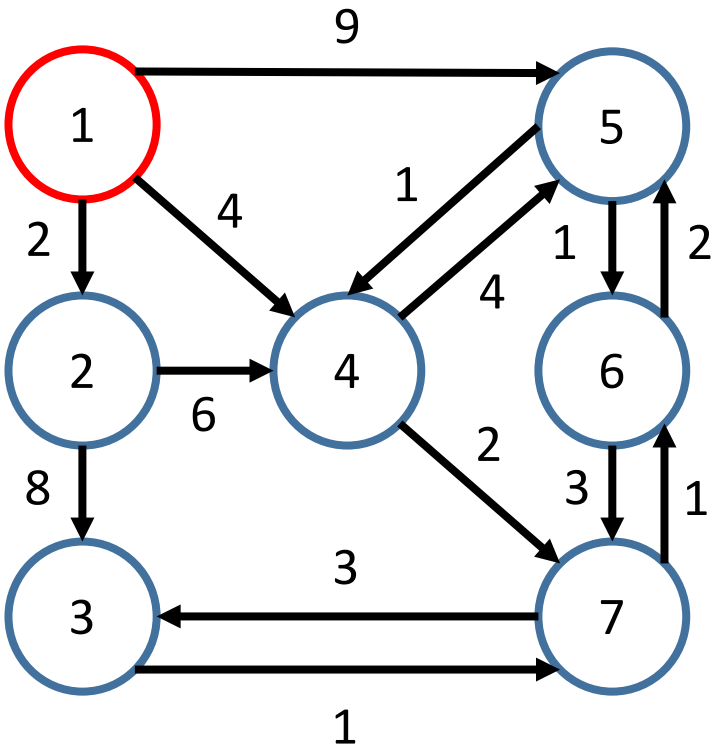
- 前述のように、ある一つの始点から全ての終点への最短経路とその費用が同時に求められる。
- 全てのノードを集合Kと$\overline{K}$に分ける。集合Kは起点oからの最短経路と最小交通費用(これをラベルという)が確定されたものでそれ以外は集合$\overline{K}$に属している。
- 交通ネットワークの均衡分析(土木学会)曰く、

> Dijkstra 法では一回一回の計算ステップで，起点 o からある外周ノード（集合 K の一つ外側にあるノード）までの最小交通費用が一つづつ確定される.その手順は以下のようになる．①前回の計算ステップでノード i の最小交通費用 $c_i$ が確定され，ノード i は集合 K に移されたとする．②そのノード i から出る各リンクの終点ノード{m}のそれぞれについて部分的最小費用 $c'_m=c_i+t_{im}$ ($t_{im}$ は始点 i 終点 m のノードをもつリンクの費用)を計算し， $c'_m$ が現在の $c_m$ よりも小さければ更新（$c_m=c'_m$）し，そうでなければそのまま据え置く．更新された場合は，後の最短経路列挙のための変数 $F_m$(先行ポインタと呼ぶ)を $F_m=i$ とする．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が最小交通費用は確定されていないノード集合$\overline{K}$から，最も小さい部分的最小費用をもつノード j を求める．④ノード j を起点 o からの最小交通費用 $c_j$ をもつノードと確定し，集合 K に移す．このとき $F_j$ に h が入力されているとすると，ノード j を含む最短経路は，起点 o → ・ ・ → h ($=F_j : h\in K$) → j まで生成されたことになる．最小交通費用は $c_j=c_h+t_{hj}$ である．⑤ノード j を i に置き換えて上記の②～⑤のステップを繰り返す.

- この説明でわかる人はいないので、きちんと解説していきます。

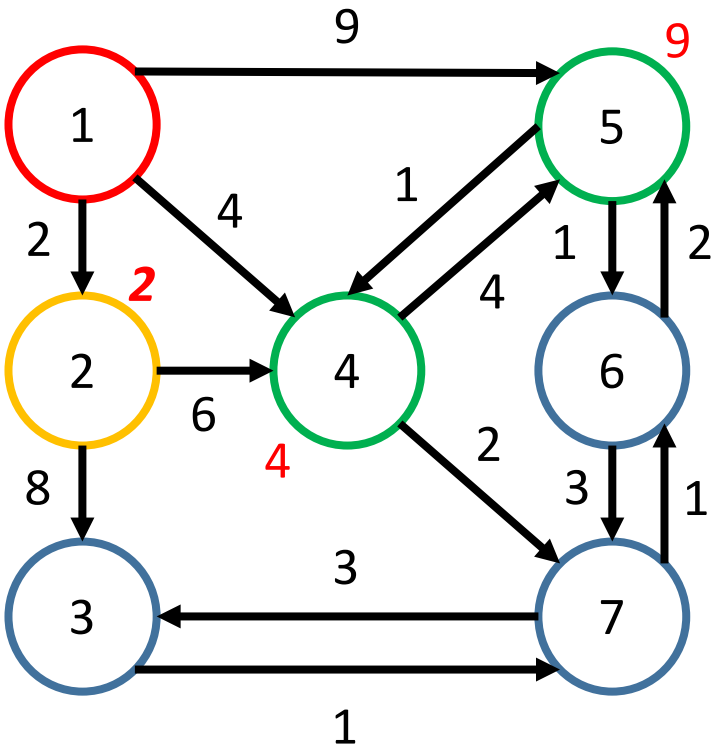
# Dijkstra法　第1ステップ

- 右のようなネットワークを考え、ノード1からその他への最短経路とその費用を計算する
- このとき、始点のノード1はラベル0が確定しており、集合Kに属している。
- 簡単に言うと、ノード1までの一番早い行き方、その費用が移動なしの0と決まって、もうこの先の議論には出てこないということ。
- これから、集合Kに属しているノードは赤くする。



# Dijkstra法　第2ステップ

- まず、1から一発で行けるノードを列挙する。
- その結果、2,4,5が挙げられる。
- 各費用は2,4,9である、一番安いのはノード2に行く費用である2のため、2を集合Kへ移動する。
- それと同時にこの2までの最小費用は2で確定する。
- このとき、ほかの二つのノードまでの費用も据え置く。
- このとき、2に行くときの最短経路で、2の直前にあるのは1であるため、2の先行ポインタ$F_2$について、 $F_2$=1とする。
- ここまでが第二ステップ、これを残りのノードすべてで行う

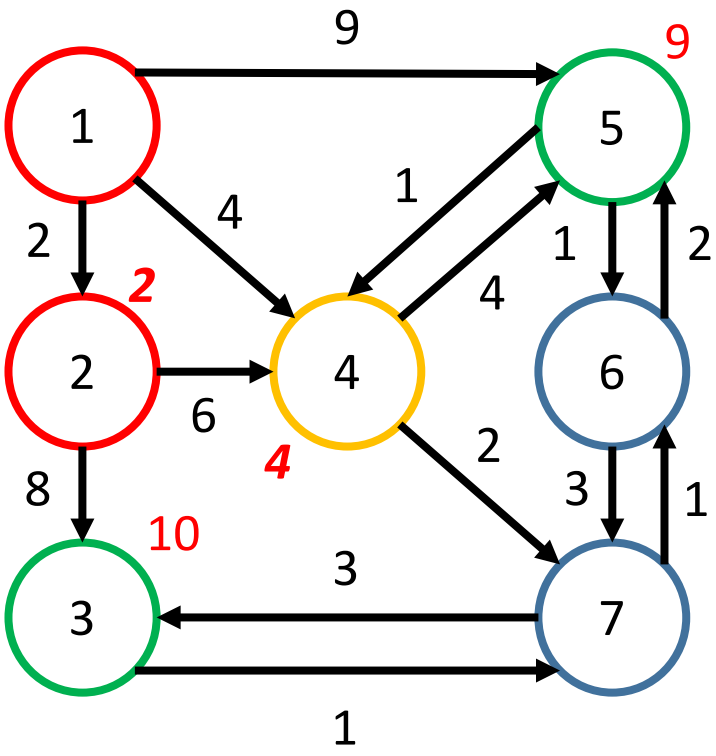


# Dijkstra法　第3ステップ

- 2から一発で行けるノードを列挙する。
- その結果、3,4が挙げられる。
- また、先ほどのノード5も候補に含む。

- このとき、ノード3,4,5に行くための現在の最小費用は10,4,9である、一番安いのはノード4に行く費用である4のため、4を集合Kへ移動する。
- 4に行くには2を経由するよりも1から直接行くほうが安いため、先行ポインタについて、 $F_4=1$とする。

- 各費用を据え置く。

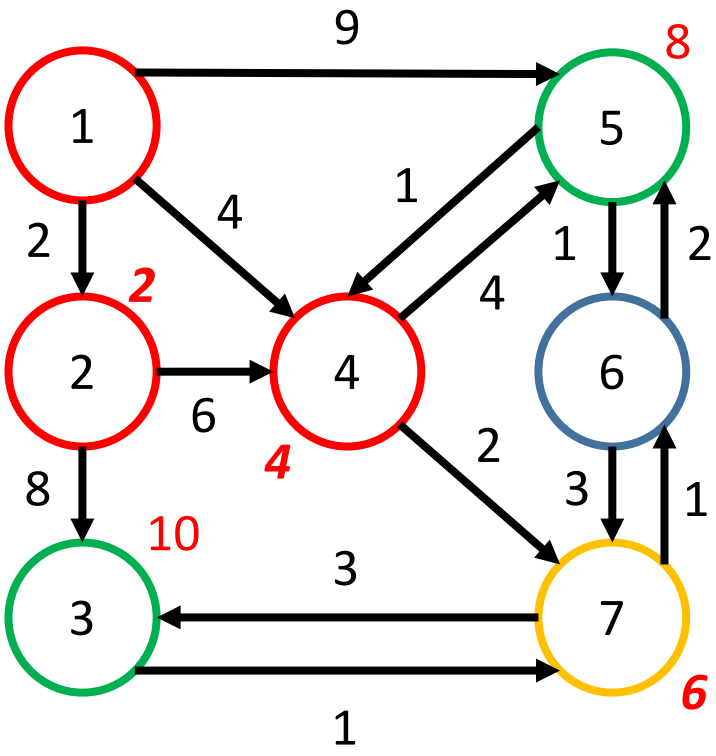


# Dijkstra法　第4ステップ

- 4から一発で行けるノードを列挙する。
- その結果、5,7が挙げられる。
- また、先ほどのノード3も候補に含む。

- このとき、ノード3,5,7に行くための現在の最小費用は10,8,6である、一番安いのはノード7に行く費用である6のため、7を集合Kへ移動する。
- 先行ポインタについて、 $F_7=4$とする。

- ノード5への最小費用がこのステップで9から8に変更された。
- 各費用を据え置く。

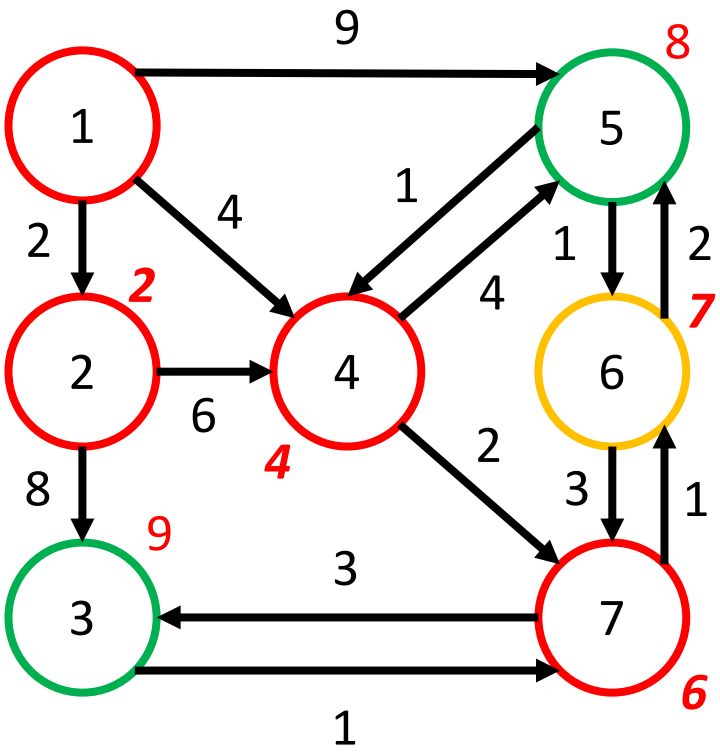


# Dijkstra法　第5ステップ

- 7から一発で行けるノードを列挙する。
- その結果、3,6が挙げられる。
- また、先ほどのノード5も候補に含む。

- このとき、ノード3,5,6に行くための現在の最小費用は9,8,7である、一番安いのはノード6に行く費用である7のため、6を集合Kへ移動する。
- 先行ポインタについて、$F_6$=7とする。

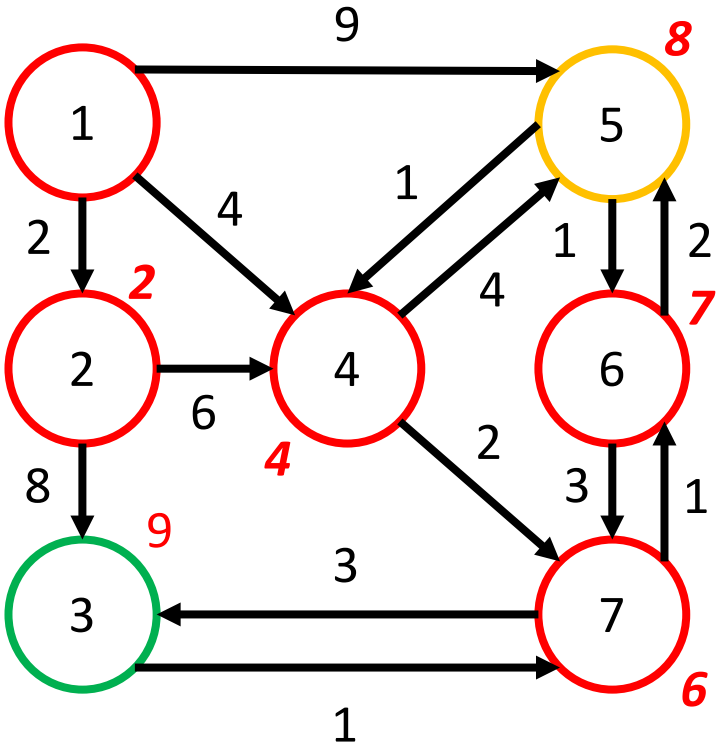
- ノード3への最小費用がこのステップで10から9に変更された。
- 各費用を据え置く。



# Dijkstra法　第6ステップ

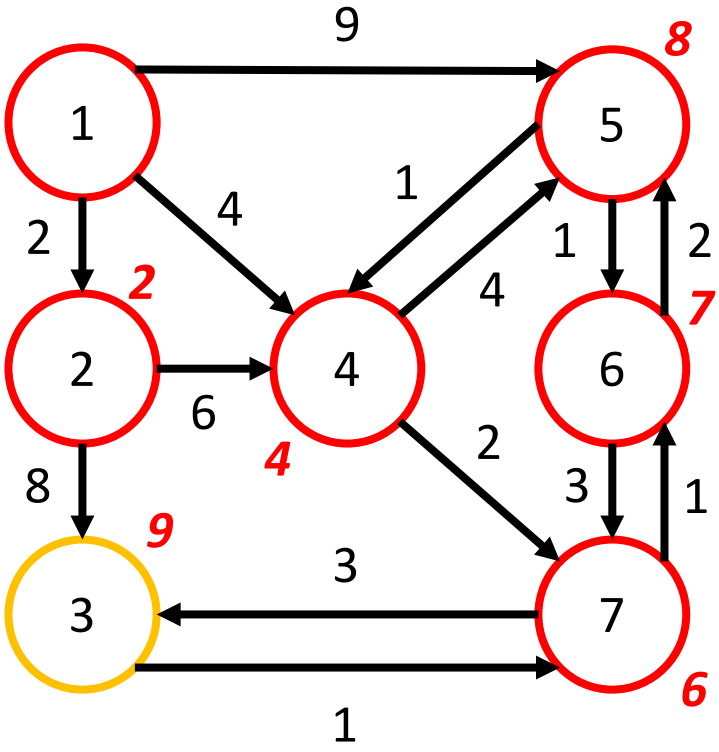
- 6から一発で行けるノードを列挙する。
- その結果、5が挙げられる。
- また、先ほどのノード3も候補に含む。

- このとき、ノード3,5に行くための現在の最小費用は9,8である、一番安いのはノード5に行く費用である8のため、5を集合Kへ移動する。
- このとき、5に行くには6を経由するより、4から直接行くほうが安いため、先行ポインタについて、 $F_5=4$とする。



# Dijkstra法　第7ステップ

- 残りはノード3のみであり、先ほどまでのステップを踏んでも5から一発で行けるノードは全てKに含まれているため、ノード3が次にKに含まれる。
- 先行ノードは、 $F_3$=7となる。
- 全てのノードがKに含まれたため、全てのステップは完了。

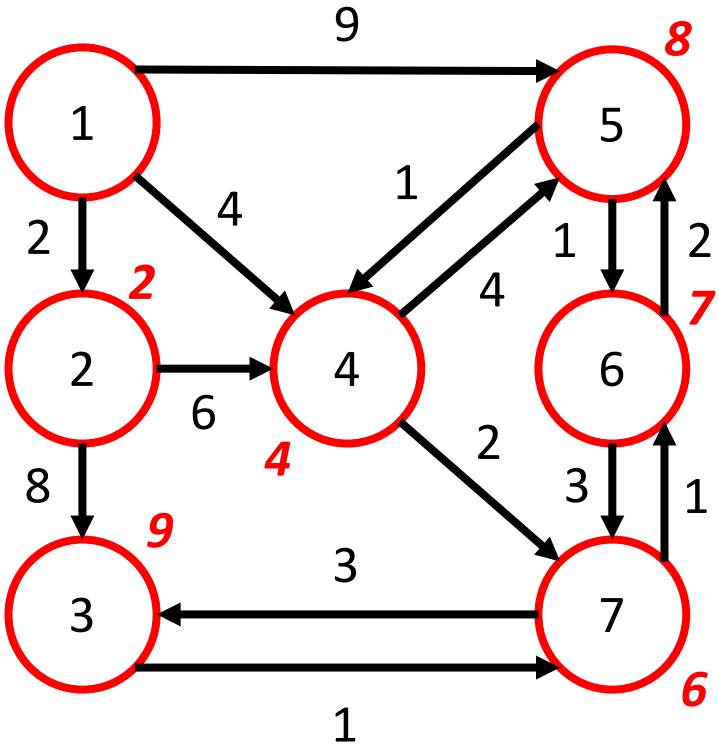


# Dijkstra法　表

- 今までのステップを表にまとめると以下のようになる。

| 計算ステップ | 対象リンク 始点 | 対象リンク 終点 m | 各ノードのラベル $c_m$ {(部分的)最小費用} ノード番号 1 | 2 | 3 | 4 | 5 | 6 | 7 | 先行ポインタ $F_m$ {(部分的)最短経路の先行ノード} ノード番号 1 | 2 | 3 | 4 | 5 | 6 | 7 | 最小費用ノード $\min_p (c_p)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | *0* | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 1 | 2,4,5 | *0* | 2 | ∞ | 4 | 9 | ∞ | ∞ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 2 |
| 3 | 2 | 3,4 | *0* | *2* | 10 | 4 | 9 | ∞ | ∞ | 0 | 1 | 2 | 1 | 1 | 0 | 0 | 4 |
| 4 | 4 | 5,7 | *0* | *2* | 10 | *4* | 8 | ∞ | 6 | 0 | 1 | 2 | 1 | 4 | 0 | 4 | 7 |
| 5 | 7 | 3,6 | *0* | *2* | 9 | *4* | 8 | 7 | *6* | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 6 |
| 6 | 6 | 5,7 | *0* | *2* | 9 | *4* | 8 | *7* | *6* | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 5 |
| 7 | 5 | 4,6 | *0* | *2* | 9 | *4* | *8* | *7* | *6* | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 3 |

注）太字斜体は集合 K に移ったノード



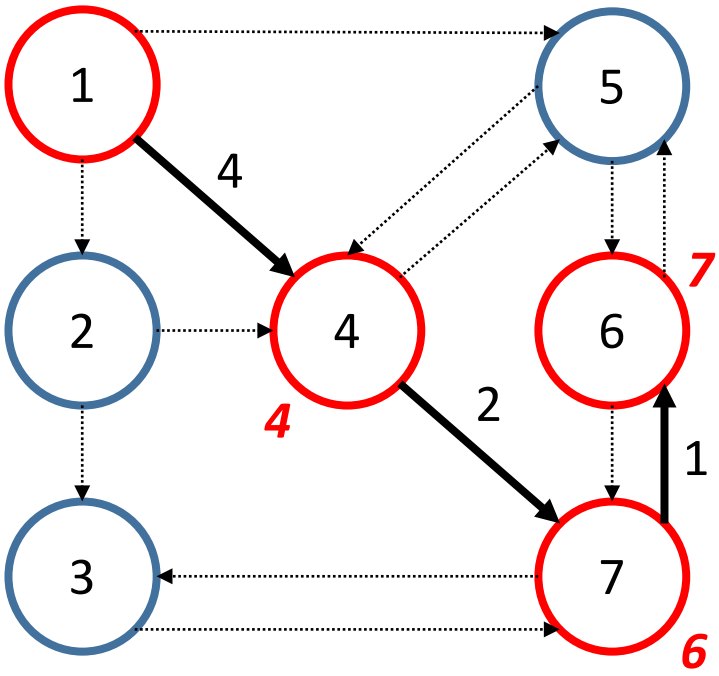

- もし手でDijkstra法をするなら表を書いたほうがいい(そんな奇人いないと思うけど)。

# Dijkstra法　表の見方

- 各ノードへの最短経路は先行ポインタをたどっていけばわかる。

| 計算ステップ | 対象リンク | | 各ノードのラベル $c_m$ {(部分的)最小費用} | | | | | | | 先行ポインタ $F_m$ {(部分的)最短経路の先行ノード} | | | | | | | 最小費用ノード $\min_p (c_p)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | ノード番号 | | | | | | | ノード番号 | | | | | | | |
| | 始点 | 終点 m | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| 1 | | | *0* | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 1 | 2,4,5 | *0* | 2 | ∞ | 4 | 9 | ∞ | ∞ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 2 |
| 3 | 2 | 3,4 | *0* | *2* | 10 | 4 | 9 | ∞ | ∞ | 0 | 1 | 2 | 1 | 1 | 0 | 0 | 4 |
| 4 | 4 | 5,7 | *0* | *2* | 10 | *4* | 8 | ∞ | 6 | 0 | 1 | 2 | 1 | 4 | 0 | 4 | 7 |
| 5 | 7 | 3,6 | *0* | *2* | 9 | *4* | 8 | 7 | *6* | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 6 |
| 6 | 6 | 5,7 | *0* | *2* | 9 | *4* | 8 | *7* | *6* | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 5 |
| 7 | 5 | 4,6 | *0* | *2* | 9 | *4* | *8* | *7* | *6* | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 3 |

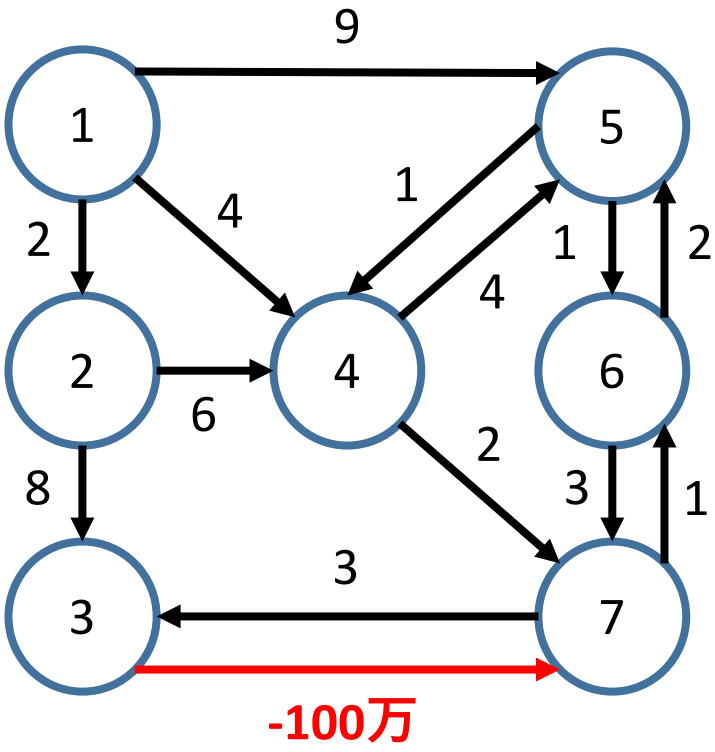
注）太字斜体は集合 K に移ったノード




- 例えば、6の先行ポインタは7、7の先行ポインタは4、4の先行ポインタは1なので、6への最短経路は1→4→7→6

# Dijkstra法の問題点

- もし仮にノード3から7までのコストが-100万だったら？
- 現実的にはほぼ起こりえないが、その道を通ると確実に100万円拾うような道があるとする。このときコストは負であると考えられる。




- 7への最短経路は先ほどのステップ5で1→4→7と確定しており、その時点でノード7はKに属するため経路探索の対象とはならない。
- しかし、3までの最短経路が確定した後に1→2→3→7のように行くと最小費用は更新される。それに伴い他の費用も変更しうる。
- このようにリンクに負の数が存在すると適用できなくなってしまう。

# ラベル修正法とは

- Dijkstra法と同じく、ある一つの始点から全ての終点への最短経路とその費用が同時に求められる。
- 大きな違いがその名の通りラベルが修正されうる点にある。
- 集合Kの代わりにリストという概念を取り入れている。

- もう一度交通ネットワークの均衡分析(土木学会)曰く、

**＜ラベル修正法のアルゴリズム＞**

**Step 1** すべてのノード{ j }についてラベル（交通費用）$c_j = \infty$（十分に大きな値），先行ポインタ $F_j = 0$ とする．

起点を o として，ノード o に関して， $c_o=0$， $i=o$ とする．

{ノードリスト} にノード o のみを登録しておく．

**Step 2** {ノードリスト} の先頭にあるノード i を取り出し，i を {ノードリスト} から削除する．

**Step 3** ノード i から出る全てのリンクの終点{m}について

$c_m > c_i + t_{im}$ ならば 以下の操作を行う．

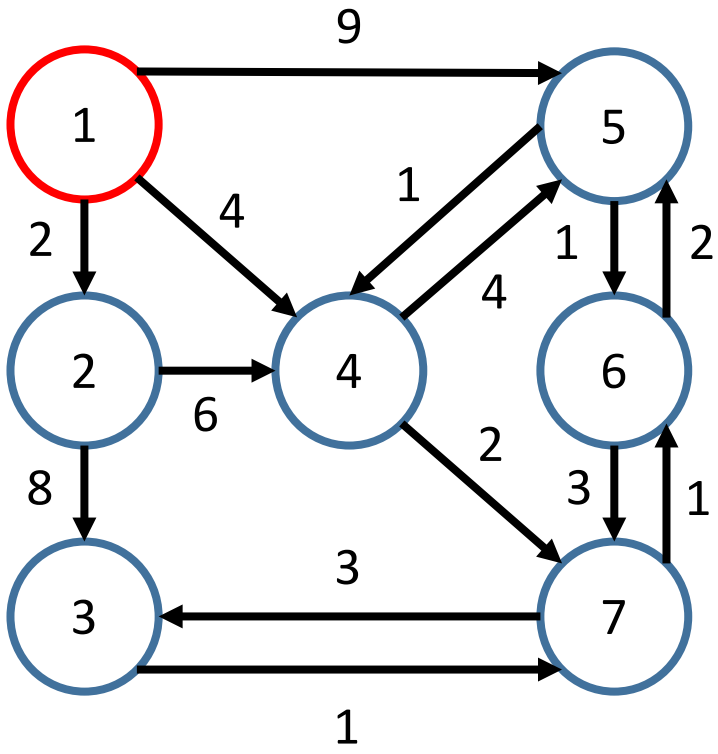
$c_m = c_i + t_{im}$， $F_m=i$ とし，また，ノード m が {ノードリスト} に含まれていなければ m を {ノードリスト} の最後尾に登録する．

**Step 4** {ノードリスト} が空集合ならば終了する．そうでなければ Step 2へ．

- やっぱりわかりづらいので説明します。

# ラベル修正法　第1ステップ

- 先ほどと同様のネットワークを考え、ノード1からその他への最短経路とその費用を計算する
- このとき、始点のノード1はラベル0が確定している。
- リストにノード1を追加する。[1]
- 以下、リストに加えられているノードを[V]で表現し、緑丸で表現する。
- 現在の始点を赤丸で表現する。

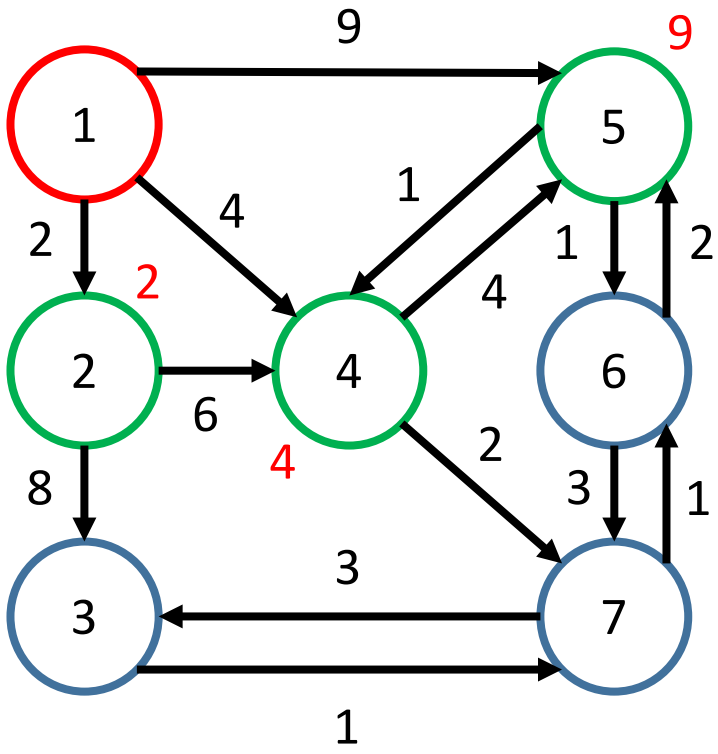


# ラベル修正法　第2,3,4ステップ

2step
- まず、ノード1から2への経路を考え、ノード2にコスト2をラベリングする。
- ノード2をリストに入れる。[2]

3step
- 次にノード1から4を考え、コスト4をラベリングする。
- ノード4をリストに入れる。[2,4]

4step
- 同様にノード5について、コスト9をラベリングする。
- ノード5をリストに入れる。[2,4,5]



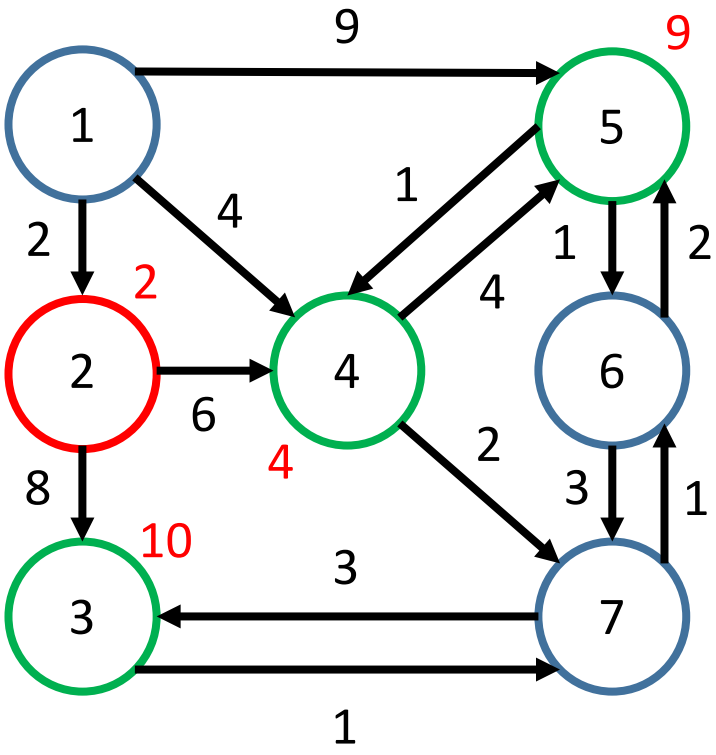
# ラベル修正法　第5,6ステップ

5step

- リストの一番左にあるノード2を始点として考える。
- 先ほどと同様にノード2からノード3を考える、このときコスト10をラベリングする。
- ノード3をリストに入れる。[4,5,3]

6step

- ノード4について考える。このときラベルは変更されない。(4<2+6より)
- ラベルが変更しないためリストの変更も起こらない。[4,5,3]



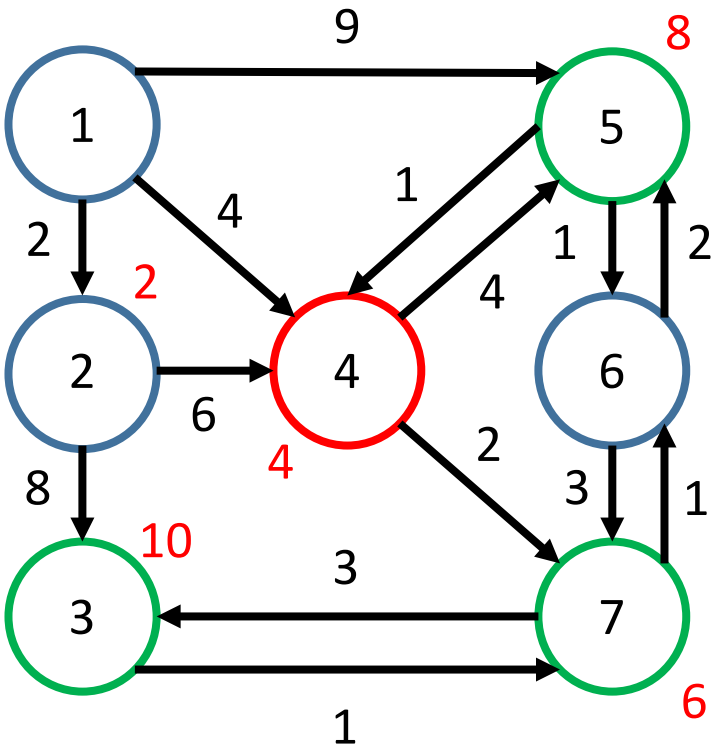
# ラベル修正法　第7,8ステップ

7step

- リストの一番左にあるノード4を始点として考える。
- 先ほどと同様にノード4からノード5を考える、このときコスト8をラベリングする。(9>4+4より)
- 5は元よりリストに含まれているため変更はなし。[5,3]

8step

- ノード7について考える。このときコスト6をラベリングする。
- ノード7をリストに入れる。[5,3,7]



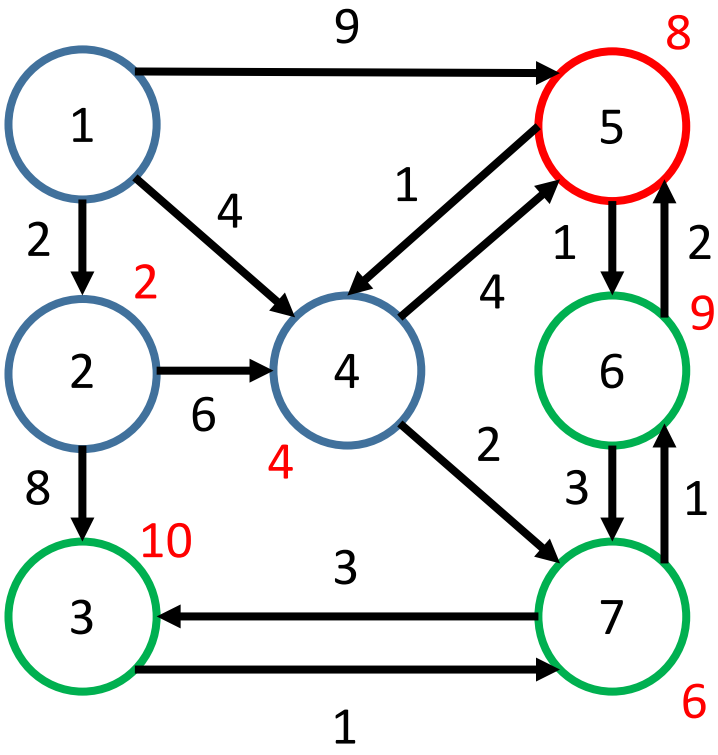
# ラベル修正法　第9,10ステップ

9step

- リストの一番左にあるノード5を始点として考える。
- 先ほどと同様にノード5からノード4を考える、このときラベルは変更されない。(4<9+1より)
- ラベルが変更しないためリストの変更も起こらない。[3,7]

10step

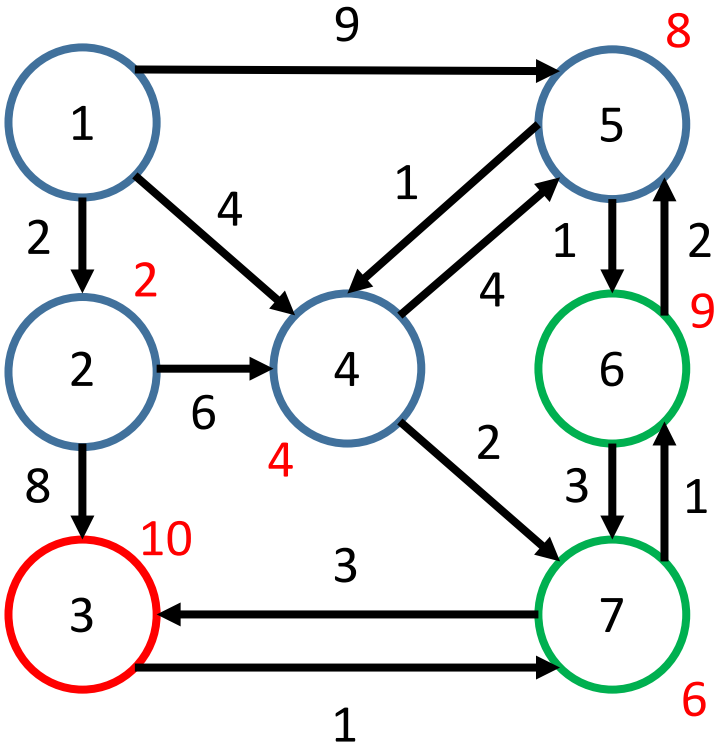
- ノード6について考える。このときコスト9をラベリングする。
- ノード6をリストに入れる。[3,7,6]



# ラベル修正法　第11ステップ

- リストの一番左にあるノード3を始点として考える。
- 先ほどと同様にノード3からノード7を考える、このときラベルは変更されない。(6<10+1より)
- ラベルが変更しないためリストの変更も起こらない。[7,6]

- ここでもしノード3から7までのコストが-100万だったら・・・？



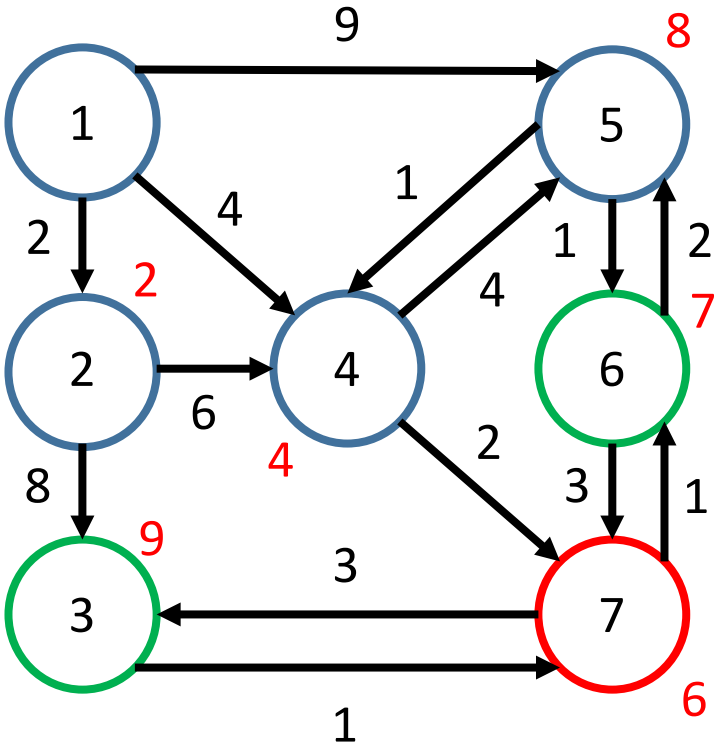
# ラベル修正法　第12,13ステップ

12step

- リストの一番左にあるノード7を始点として考える。
- 先ほどと同様にノード7からノード3を考える、このときコスト9をラベリングする。(10>6+3より)
- ラベルが変更があったため、リストに3を加える。[6,3]

13step

- ノード6について考える。このときコスト7をラベリングする。(7>6+1より)
- 6は元よりリストに含まれているため変更はなし。[6,3]



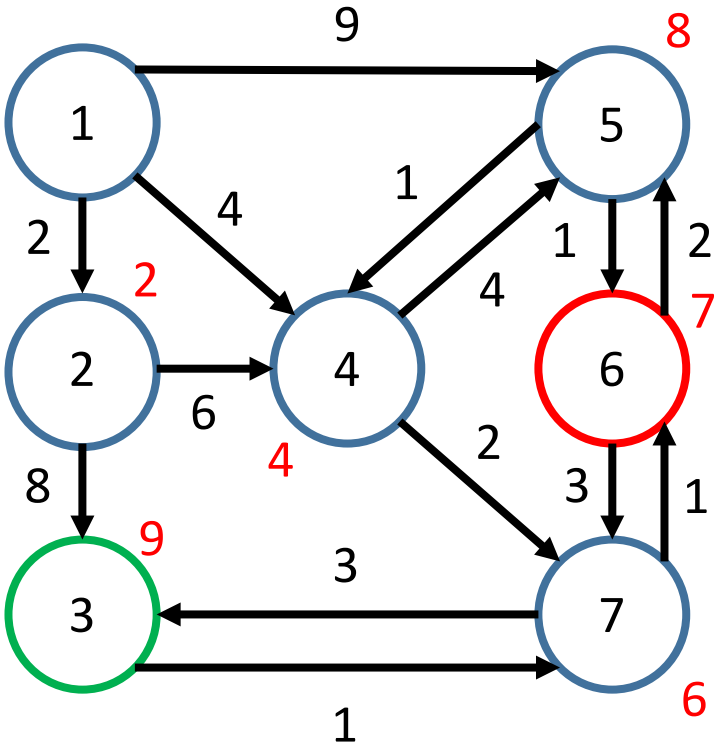
# ラベル修正法　第14,15ステップ

14step

- リストの一番左にあるノード6を始点として考える。
- 先ほどと同様にノード6からノード5を考える、このときラベルは変更されない。(8<7+2より)
- ラベルが変更しないためリストの変更も起こらない。[3]

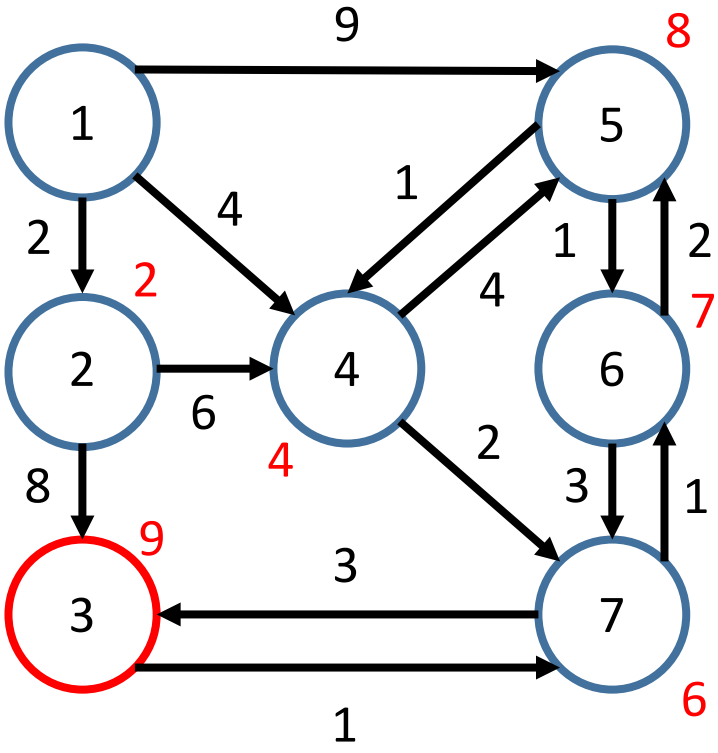
15step

- ノード7について考える。このときラベルは変更されない。(6<7+3より)
- ラベルが変更しないためリストの変更も起こらない。[3]



# ラベル修正法　第16ステップ

- リストの一番左にあるノード3を始点として考える。
- 先ほどと同様にノード3からノード7を考える、このときラベルは変更されない。(6<9+1より)
- ラベルが変更しないためリストの変更も起こらない。[ ]
- リストから全てのノードが消えたため終了する。

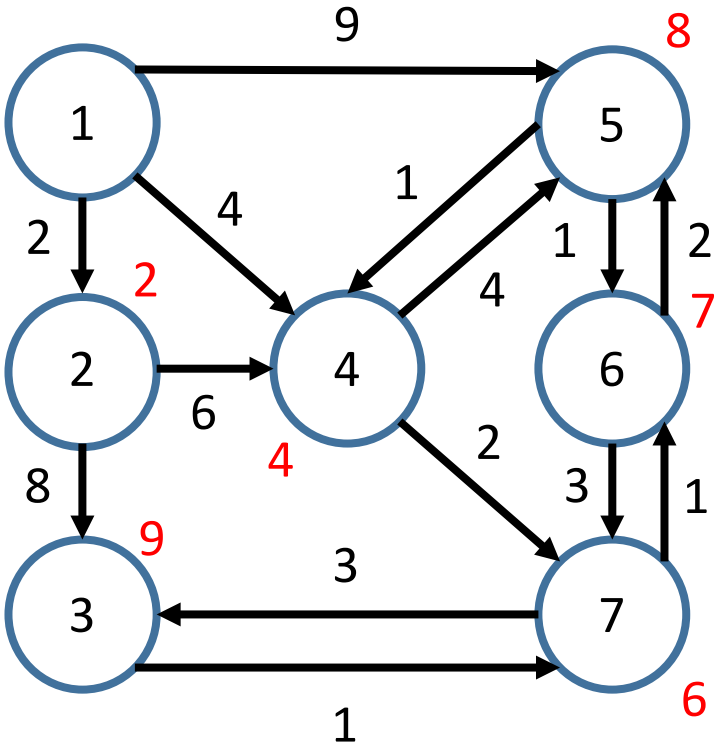


# ラベル修正法　表

- 今までのステップを表にまとめると以下のようになる。

| 計算ステップ | 対象リンクに接続するノード | | 各ノードのラベル $c_m$ {(部分的)最小費用} | | | | | | | 先行ポインタ $F_m$ {(部分的)最短経路の先行ノード} | | | | | | | リスト |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | ノード番号 | | | | | | | ノード番号 | | | | | | | |
| | 始点 | 終点 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| 1 | | | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | *1* | *2* | 0 | *2* | ∞ | ∞ | ∞ | ∞ | ∞ | 0 | *1* | 0 | 0 | 0 | 0 | 0 | *2* |
| 3 | 1 | *4* | 0 | 2 | ∞ | *4* | ∞ | ∞ | ∞ | 0 | 1 | 0 | *1* | 0 | 0 | 0 | *2,4* |
| 4 | 1 | *5* | 0 | 2 | ∞ | 4 | *9* | ∞ | ∞ | 0 | 1 | 0 | 1 | *1* | 0 | 0 | *2,4,5* |
| 5 | *2* | *3* | 0 | 2 | *10* | 4 | 9 | ∞ | ∞ | 0 | 1 | *2* | 1 | 1 | 0 | 0 | *4,5,3* |
| 6 | 2 | *4* | 0 | 2 | 10 | 4 | 9 | ∞ | ∞ | 0 | 1 | 2 | 1 | 1 | 0 | 0 | 4,5,3 |
| 7 | *4* | *5* | 0 | 2 | 10 | 4 | *8* | ∞ | ∞ | 0 | 1 | 2 | 1 | *4* | 0 | 0 | 5,3 |
| 8 | 4 | *7* | 0 | 2 | 10 | 4 | 8 | ∞ | *6* | 0 | 1 | 2 | 1 | 4 | 0 | *4* | *5,3,7* |
| 9 | *5* | *4* | 0 | 2 | 10 | 4 | 8 | ∞ | 6 | 0 | 1 | 2 | 1 | 4 | 0 | 4 | 3,7 |
| 10 | 5 | *6* | 0 | 2 | 10 | 4 | 8 | *9* | 6 | 0 | 1 | 2 | 1 | 4 | *5* | 4 | *3,7,6* |
| 11 | *3* | *7* | 0 | 2 | 10 | 4 | 8 | 9 | 6 | 0 | 1 | 2 | 1 | 4 | 5 | 4 | 7,6 |
| 12 | *7* | *3* | 0 | 2 | *9* | 4 | 8 | 9 | 6 | 0 | 1 | *7* | *1* | 4 | 5 | 4 | *6,3* |
| 13 | 7 | *6* | 0 | 2 | 9 | 4 | 8 | *7* | 6 | 0 | 1 | 7 | 1 | 4 | *7* | 4 | 6,3 |
| 14 | *6* | *5* | 0 | 2 | 9 | 4 | 8 | 7 | 6 | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 3 |
| 15 | 6 | *7* | 0 | 2 | 9 | 4 | 8 | 7 | 6 | 0 | 1 | 7 | 1 | 4 | 7 | 4 | 3 |
| 16 | *3* | 7 | 0 | 2 | 9 | 4 | 8 | 7 | 6 | 0 | 1 | 7 | 1 | 4 | 7 | 4 | |

注）太字斜体は各計算ステップで変化した項目



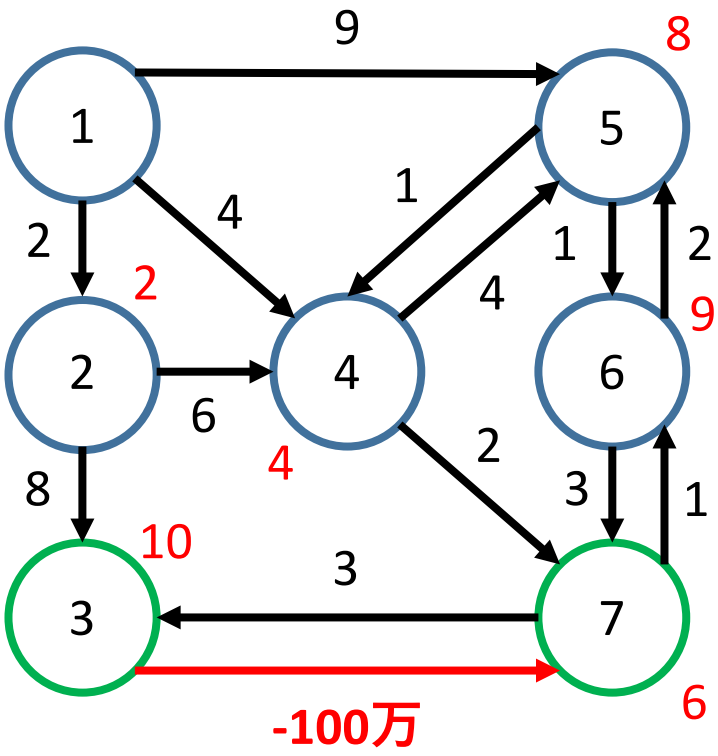

- Dijkstra法では一つのノードからの費用をまとめて計算しており、こちらはそれを分けて書いているためステップ数は増加する。

# ラベル修正法のメリット

- 先ほどのノード3から7までのコストが-100万だったらどうなるかという問題がラベル修正法だと解決できる。

- 第11ステップにおいて3→7を計算するためここで7のラベルが変更でき、それにともない6,5,4とラベルの変更が行える。
- このようにリンクに負の数が存在すると適用できなくなってしまう。

- 実際は交通ネットワークにおいてコストが負になることは稀である。
- 混雑料金を定義することによって遠回りしたほうが”相対的”にコストが負になることはあるが、絶対的に負になることは一般的には無い。



# Dijkstra法とラベル修正法

- ネットワークリンクが負でない一般的な交通ネットワークにおいてはどちらの方が最短経路探索アルゴリズムとして優れているのか？

- Dijkstra法は同じノード間に対しての計算を行わないのに対して、ラベル修正法はその特性上同じノード間に対して計算を行うことがある。

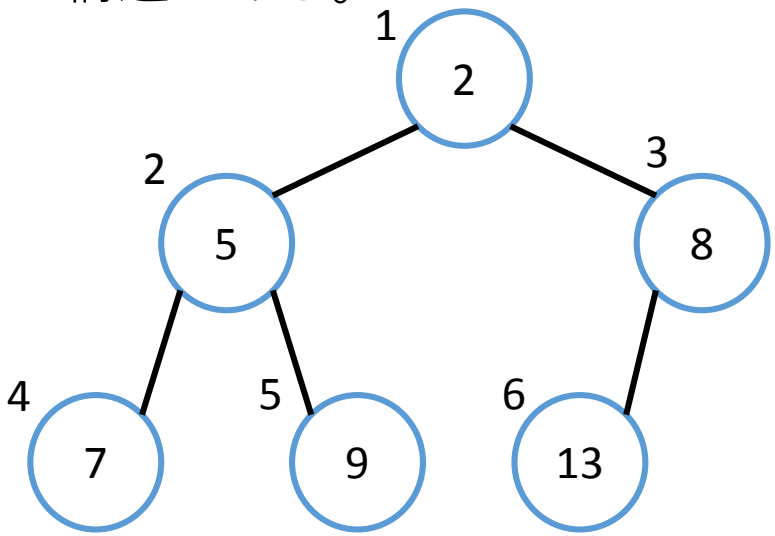
  Ex) ステップ11とステップ16

- そのため計算量が少ないDijkstra法の方が交通ネットワークの最短経路探索アルゴリズムとして優れているといえる。

# ヒープ構造

- ネットワークが複雑になると、一度に取り扱うラベルの数が膨大になる。元々据え置かれていたラベルに加え新しくラベリングされたものが加えられ、大規模になればなるほど最小値を探すのに時間がかかる。

- ヒープ構造とは、Dijkstra法におけるラベルの最小値を効率よく見つけ出すデータ構造である。



ヒープ構造

# ヒープ構造のルール

- ヒープ条件

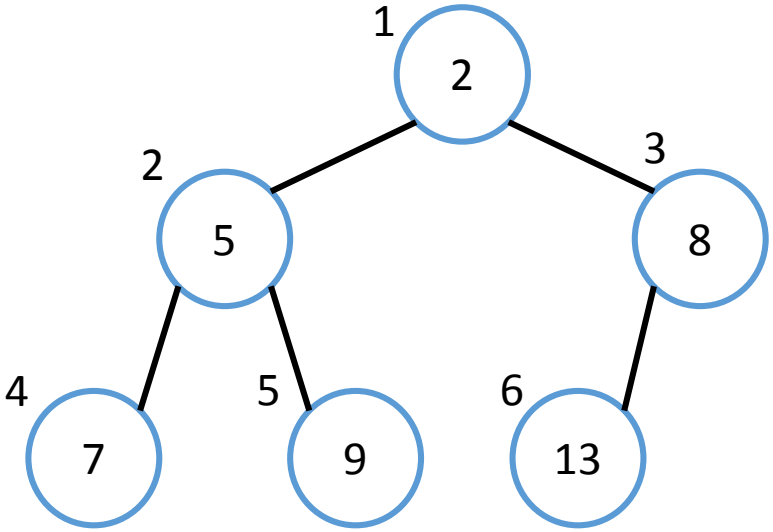
根以外の全てのノードgについて

(gの親ノードのラベル)≦(gのラベル)

- 二分木における性質

親ノードの配列順位nのとき、子の配列順位は2n,2n+1

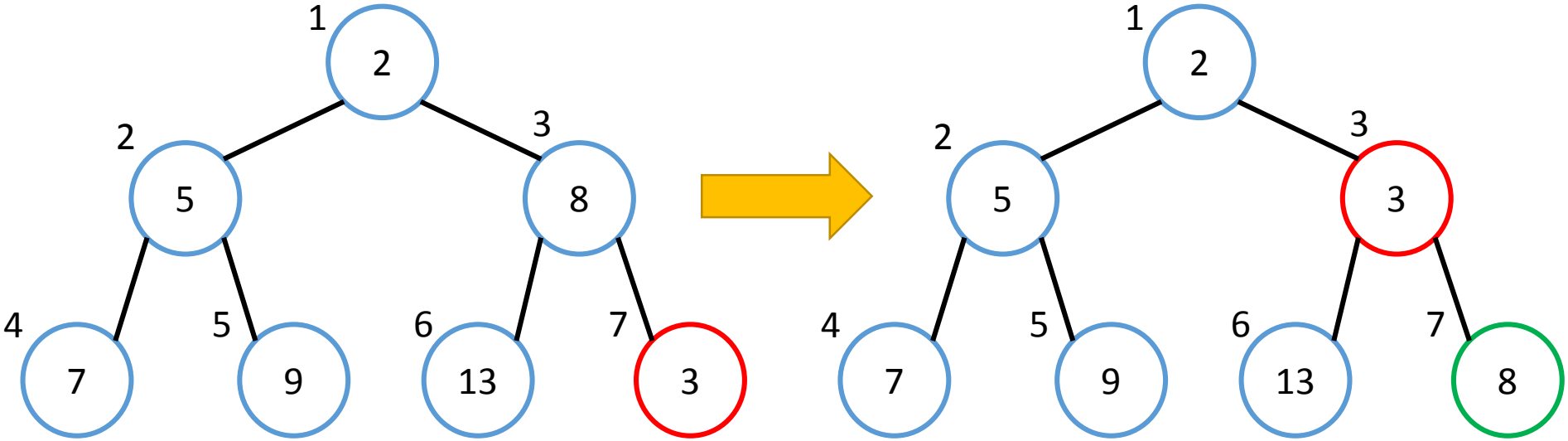
- ヒープ構造を使って何を行うのか？




| 配列順位i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| ラベルX(i) | 2 | 5 | 8 | 7 | 9 | 13 |

# ヒープ構造　新たなデータを挿入

- 新たなデータ3が配列順位7に追加される。
- するとヒープ条件が崩れてしまうため、データの入れ替えを行う。
- 子が親より小さいラベルを持っていたら交換する。

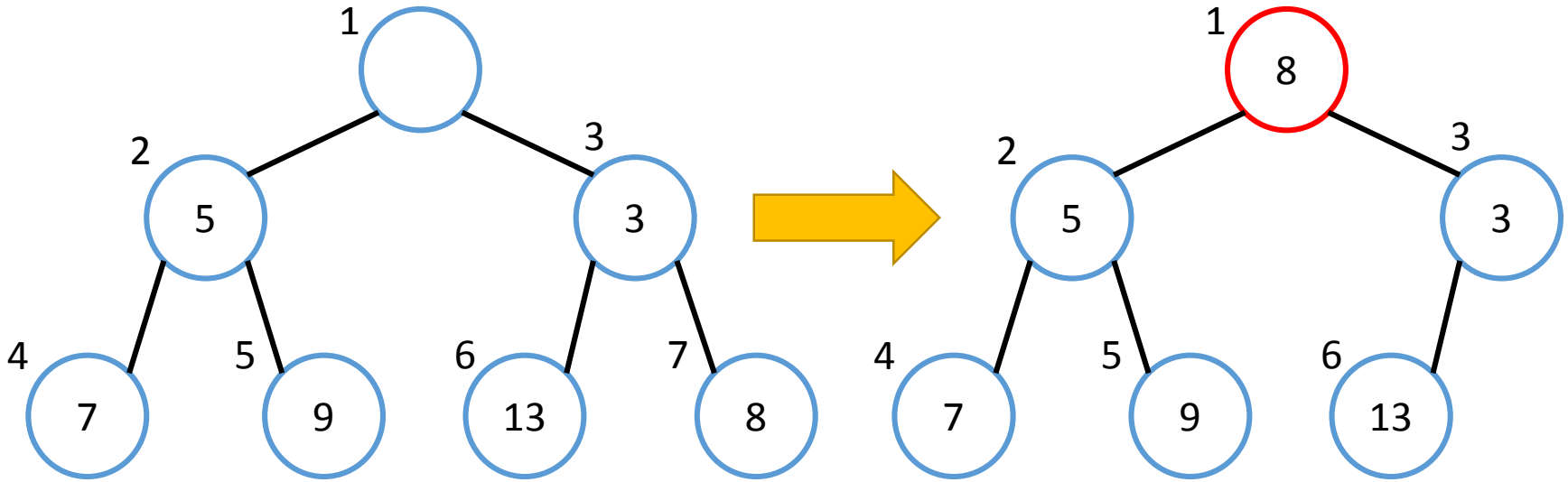



- ヒープ条件が成立するまで繰り返す。
- Dijkstra法において新しいラベルが追加されたときに適用する。

# ヒープ構造　データの削除

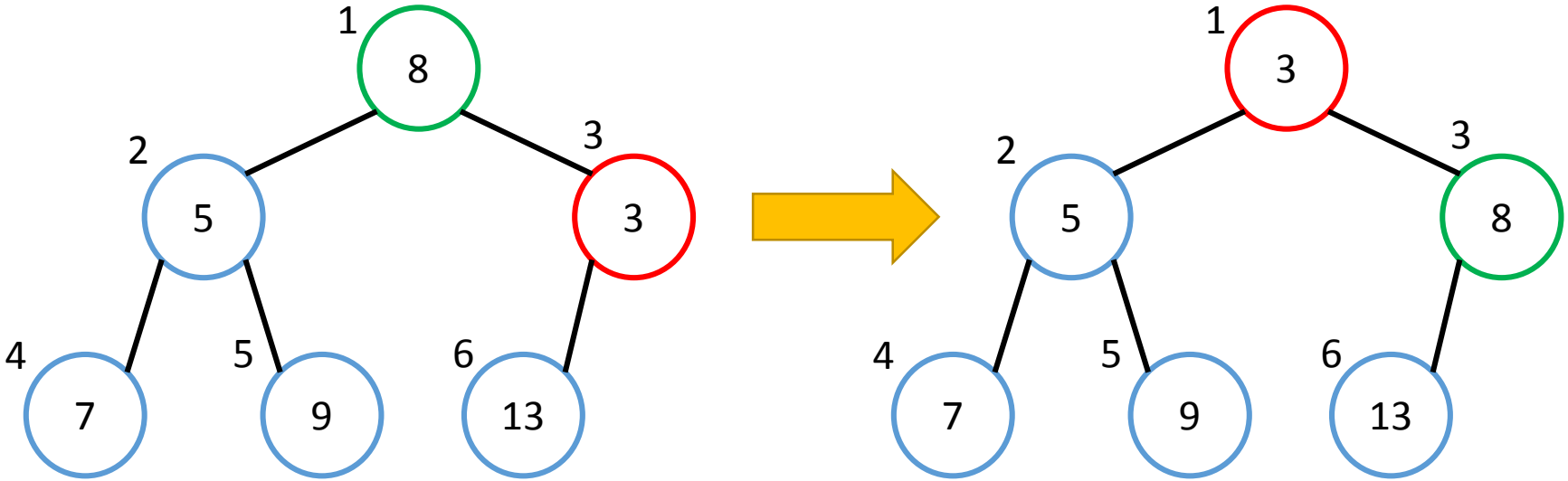
- ヒープ条件より、一番小さいデータの配列順位は1である。
- Dijkstra法において、ノードが集合Kに移動すると最小のデータが一つヒープ構造から削除される。

- まず、配列順位最後尾のデータを根に持っていく。



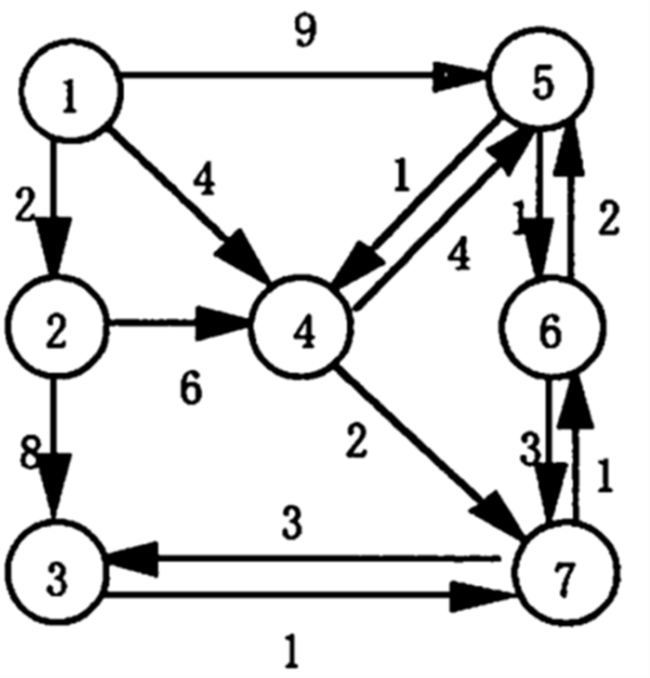
# ヒープ構造　データの削除

- その後、親のラベルと、「小さいほうの子のラベル」を交換する。




- これをヒープ条件が成立するまで繰り返す。
- Dijkstra法において次に確定されるであろうラベルが根に来る。

# R によるDijkstra法




上のネットワークに関してR でDijkstra法を実施し
ノード1から各ノードへの最小リンクコストを算出する

# R によるDijkstra法

- コードの全容

ダイクストラ法 R.R × | ダイクストラ法.R ×

Source on Save

```
#ダイクストラ法

#条件の設定
dijkstra<-function(matrix,startnode){
  L<-matrix    #行列の作成
  L[which(L==0)]<-Inf #すべてのノード間の最小交通費用を無限大にする（未開拓なノード）
  diag(L)<-0 #対角成分はゼロ(自ノード同士)
  n<-nrow(matrix) #ノード数
  d<-rep(Inf,n) #あるノードから全ノードへの最小交通費用を無限大にする（未開拓なノード）
  d[startnode]<-0 #自ノード同士はゼロ
  K<-1:n #最小交通費用が未定のノード集合
  K<-K[-startnode] #起点は除去
  i<-startnode #このノードに関して最小交通費用を探索する

  #ダイクストラ法の実施
  while(length(K)>0){
    for (j in 1:n)
      d[j]<-min(d[j],d[i]+L[i,j]) #最小費用が更新されたら
    i<-K[which(d[K]==min(d[K]))[1]] #Mの中で最小費用が最小のノードについて探索する
    K<-K[-which(K==i)] #今まで起点としていたノードは最小費用が決定したから除去する
  }
  d #起点startnodeから各ノードへの最小費用を算出
}

#先に描写した距離行列で実践
a<-as.matrix(read.csv("matrix.csv",header=F))
dijkstra(a,1)
```

# Rによる Dijkstra法

段階を踏み最小のリンクコストを持つノードを決定していく
まだ調べていないリンクが最小だと認定されないように

前半部分はDijkstra法を実施するための下準備

```
#ダイクストラ法

#条件の設定
dijkstra<-function(matrix,startnode){
  L<-matrix    #行列の作成
  L[which(L==0)]<-Inf #すべてのノード間の最小交通費用を無限大にする（未開拓なノード）
  diag(L)<-0 #対角成分はゼロ(自ノード同士)
  n<-nrow(matrix) #ノード数
  d<-rep(Inf,n) #あるノードから全ノードへの最小交通費用を無限大にする（未開拓なノード）
  d[startnode]<-0 #自ノード同士はゼロ
  K<-1:n #最小交通費用が未定のノード集合
  K<-K[-startnode] #起点は除去
  i<-startnode #このノードに関して最小交通費用を探索する
```

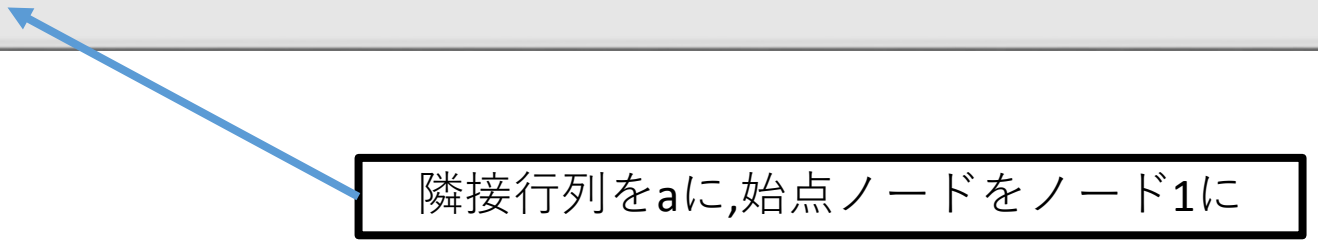
同じノードへの移動はできない

# R によるDijikstra法

後半部分はDijikstra法を実践するコード

```
#ダイクストラ法の実施
 while(length(K)>0){
  for (j in 1:n)
    d[j]<-min(d[j],d[i]+L[i,j]) #最小費用が更新されたら
  i<-K[which(d[K]==min(d[K]))[1]] #Mの中で最小費用が最小のノードについて探索する
  K<-K[-which(K==i)] #今まで起点としていたノードは最小費用が決定したから除去する
 }
 d #起点startnodeから各ノードへの最小費用を算出
}

#先に描写した距離行列で実践
a<-as.matrix(read.csv("matrix.csv",header=F))
dijkstra(a,1)
```



# R によるDijkstra法

- 結果

```
Console C:/Users/k.ogawa/Desktop/R/
+   L<-matrix     #行列の作成
+   L[which(L==0)]<-Inf #すべてのノード間の最小交通費用を無限大にする(未開拓なノード)
+   diag(L)<-0 #対角成分はゼロ(自ノード同士)
+   n<-nrow(matrix) #ノード数
+   d<-rep(Inf,n) #あるノードから全ノードへの最小交通費用を無限大にする(未開拓なノード)
+   d[startnode]<-0 #自ノード同士はゼロ
+   K<-1:n #最小交通費用が未定のノード集合
+   K<-K[-startnode] #起点は除去
+   i<-startnode #このノードに関して最小交通費用を探索する
+
+   #ダイクストラ法の実施
+   while(length(K)>0){
+     for (j in 1:n)
+       d[j]<-min(d[j],d[i]+L[i,j]) #最小費用が更新されたら
+     i<-K[which(d[K]==min(d[K]))[1]] #Kの中で最小費用が最小のノードについて探索する
+     K<-K[-which(K==i)] #今まで起点としていたノードは最小費用が決定したから除去する
+   }
+   d #起点startnodeから各ノードへの最小費用を算出
+ }
>
> #先に描写した距離行列で実践
> a<-as.matrix(read.csv("matrix.csv",header=F))
> dijkstra(a,1)
[1] 0 2 9 4 8 7 6
```

ノード1を始点として

- ノード2への最小リンクコスト→2
- ノード3への最小リンクコスト→9

.

.

- ノード7への最小リンクコスト→6

# プログラミング

- 先ほどの説明のように交通ネットワークにおいてはDijkstra法がメジャーであるため、プログラミングについてはDijkstra法のみを取り扱う。

- ここではpythonにおけるDijkstra法を取り上げる。

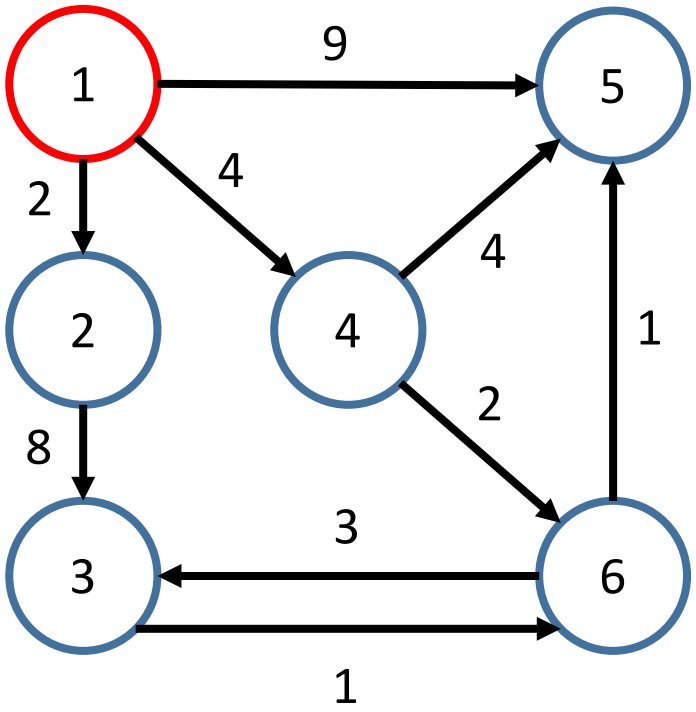
- FS_Dijkstra, Dijkstra_package, tokyo_train_network.csvの三つのファイルの確認をよろしくお願いします。

- スライドは一応作ってあるだけで主な説明はコード上でします。

# Dijkstra法　プログラミング

- 右のようなネットワークを考える。
- まず、隣接行列を定義する。

```
g = [
    [0,2,0,4,9,0],
    [0,0,8,0,0,0],
    [0,0,0,0,0,1],
    [0,0,0,0,4,2],
    [0,0,0,0,0,0],
    [0,0,3,0,1,0],
]
```




- 色々と定義する

```
import math
g_num = len(graph) #ノードの数を定義
unchecked_nodes = list(range(g_num)) # 未探索ノードのリストを作り、全ノードを含める
link_cost = [math.inf] * g_num # ノードごとの最小費用のリストを作り、全ノードのラベルを無限大にする
previous_nodes = [-1] * g_num # 先行ポインタのリスト、[-1]はリストの最後という意味
link_cost[0] = 0 # 最初のノード1~python上では0~の最小費用を0とする
```

# Dijkstra法　プログラミング

- 後々使う未探索ノード集合の中で最小のラベルを持つ要素が何番目にあるのかを取得する関数を定義する。

```
def get_min_rabel_node_rank(min_rabel_node_link, link_cost, unchecked_nodes): # 未探索ノードの中で最小のラベルを持つノードが何番目の要素なのかを
    start = 0
    while True:
        rank = link_cost.index(min_rabel_node_link, start) # 最小費用のリストから最も小さいラベルが何個目の要素なのかを取り出す
        found = rank in unchecked_nodes # そのラベルが未探索ノードのものかどうかを調べる
        if found:
            return rank # もし未探索ノードのものであればrankの値を返す
        else:
            start = rank + 1 # そうでなければ、rankの値に1を足したものを新たなスタート値とし、先ほどのrankより後の最小費用リストから検索しなおす
```

- Dijkstraアルゴリズムの一番肝の部分のコード

```
while(len(unchecked_nodes) > 0): #未探索ノード集合の要素が0より大きい間は繰り返す、つまり未探索ノードがなくなるまで繰り返す
     # まず未探索ノードのうちリンクコストが最小のものを選択する
    possible_min_link_cost = math.inf # 最小のリンクコストを見つけるための一時的なリンクコストを設定。初期値は無限大に設定
    for node_index in unchecked_nodes: # 未探索のノードのなかでループ
         if possible_min_link_cost > link_cost[node_index]:
            possible_min_link_cost = link_cost[node_index] # より小さい値が見つかれば更新
    target_min_index = get_min_rabel_node_rank(possible_min_link_cost, link_cost, unchecked_nodes) # 未探索ノードのうちで最小のラベルを持つ
    unchecked_nodes.remove(target_min_index) # このノードは集合Kに属するため未探索ノード集合から消去
    # ここまでで最小のラベルを持つノードの検索
    # ここからそのノードからのコストを計算し、最小費用を据え置く
    target_link = g[target_min_index] # ターゲットになるノードからのリンクのリスト
    for rank, cost in enumerate(target_link):
        if cost > 0:
            if link_cost[rank] > (link_cost[ target_min_index] + cost):
                link_cost[rank] = link_cost[ target_min_index] + cost # 過去に設定された最小費用よりも小さい場合は更新
                previous_nodes[rank] =  target_min_index #  先行ポインタのリストも更新

# whileからここまでを未探索ノードがなくなるまで繰り返す。
```

# Dijkstra法　プログラミング

- 結果を表示する

```
# 以下で結果の表示

print("-----最短経路-----")
previous_node = check_node - 1 # check_nodeは最短経路、最小費用を調べたいノードを入れる
while previous_node > -1: # 先行ポインタが存在するまで続ける
    if previous_node > 0:
        print(str(previous_node + 1) + " <- ", end='') # 先行ポインタを辿って"<-"でつなげる
    else:
        print(str(previous_node + 1))
    previous_node = previous_nodes[previous_node]

print("-----最小費用-----")
print(link_cost[check_node - 1])
```

- これらを一気にやってくれるやつ

```
def FS_Dijkstra(graph, check_node):
```

- 隣接行列を作りさえすれば(おそらく)これでDijkstraは回ります。
  - 実際通常の隣接行列では大規模ネットワークを表現するとき疎行列(sparse matrix)になることが多いため、工夫して表現する必要がある。

# Dijkstra法　パッケージで

- Dijkstra法はかなり有名なアルゴリズムなのでパッケージがnetworkXに入って出回っている。

- 今回は例として関東近郊のJR駅の路線ネットワークを使用。
- tokyo_train_network.csvに起点、終点、距離(コスト)が入っている。
- まず準備のために色々と定義する

```
import csv
import numpy as np
import networkx as nx

def read_csv(file_name):
    data = [] # データを入れる空のリストを作成
    f = open(file_name, 'r', encoding="utf-8")
    reader = csv.reader(f)
    header = next(reader)
    for row in reader:
        data.append(row) # データの行をすべて一つにまとめる、今回は起点、終点、距離がひとまとめになる
    f.close()
    return data

def create_graph(data):
    network = nx.Graph()
    for x in data:
        network.add_edge(x[0], x[1], cost = float(x[2])) # データをグラフに成形する
```

# Dijkstra法　パッケージで

- Dijkstra法を適用する

```
data = read_csv(file_name) # データを読み込む
graph = create_graph(data) # グラフをつくる
min_path = nx.dijkstra_path(graph, origin, destination, "cost") # 最短経路を求める
min_length = nx.dijkstra_path_length(graph, origin, destination, "cost") # 最小費用を求める
print(min_path)
print(min_length)
```

- これらを一気にやってくれるやつ

```
def dijkstra(file_name, origin, destination)
```

- 試しにいわきから横浜の最短経路を出す

```
dijkstra("tokyo_train_network.csv", 'iwaki', 'yokohama')
```

```
['iwaki', 'tomobe', 'abiko', 'shinmatsudo', 'nippori', 'akihabara', 'kanda', 'tokyo', 'sinagawa', 'kawasaki', 't
urumi', 'higasikanagawa', 'yokohama']
244.00000000000003
```

- なんの面白味もない結果が出た。

- 始点終点をいじれば色々な経路が調べられるのでやってみよう！

# A*アルゴリズム

- Dijkstra法やラベル修正法が全ノードの探索。
- 対してA*は一つの目的地への最短経路を効率よく見つける方法。

- とても簡単に原理を述べると、

あるノード$n$について、

$f(n)$:$n$を経由した最小費用の推定値

$g(n)$:$n$までの最小費用の推定値

$h(n)$:$n$から目的地までの最小費用の推定値

としたとき、

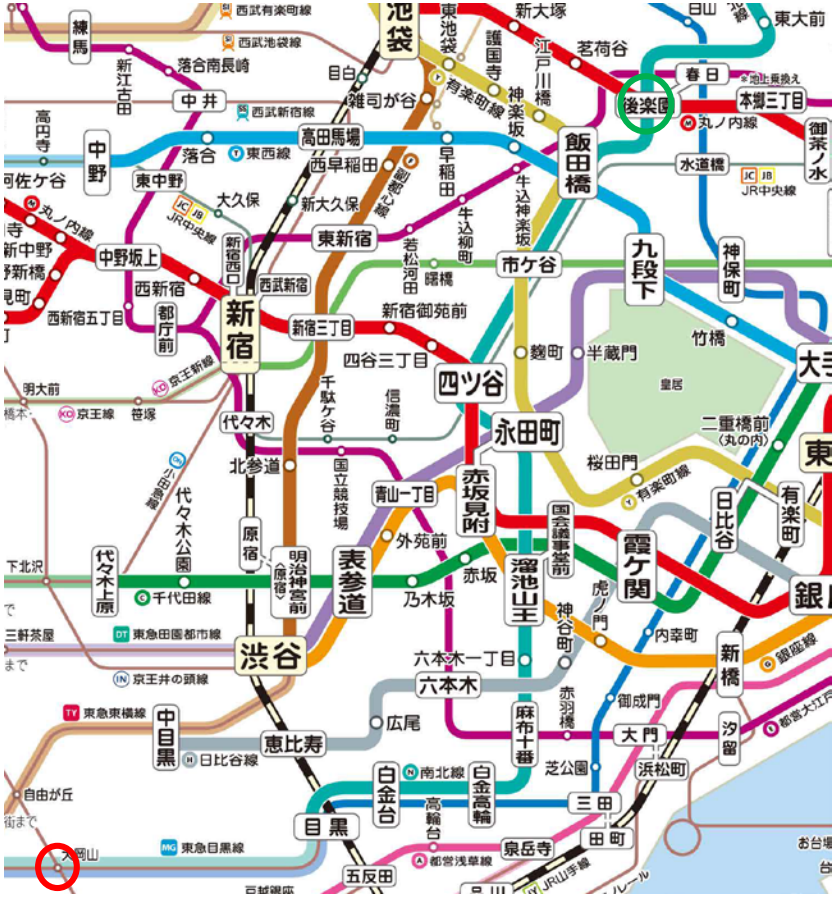
$$f(n) = g(n) + h(n)$$

となり、ある程度適切な$h(n)$を導入していき最短経路を探索する。

- とてもとても簡単に説明すると、近そうな方から調べていくアルゴリズムである。
- 二次元マップなどでは 通常、$h(n)$ にユークリッド距離を適用する。

# A*アルゴリズム

- 例として、東京の地下鉄路線図を用いる。
- 読売巨人軍の本拠地東京ドームがある後楽園(緑丸)から、東工大の所在地大岡山(赤丸)までの最短経路を考える。
- 全ての駅(ノード)は路線図の通りに立地しているとする。
- よって、ユークリッド距離は単純な地図上の距離と等しいと考える。

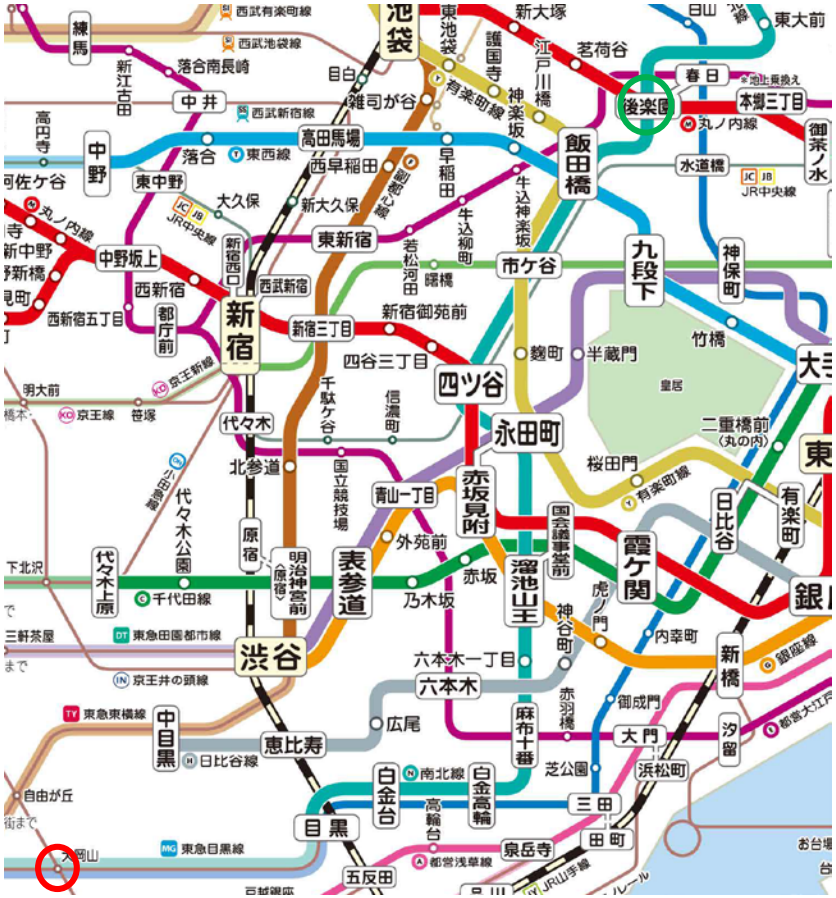


# A*アルゴリズム

- まず、後楽園駅から直接つながっている駅である、**南北線東大前**駅、**飯田橋**駅、**丸ノ内線本郷三丁目**駅、**茗荷谷**駅を中間ノード候補とする。

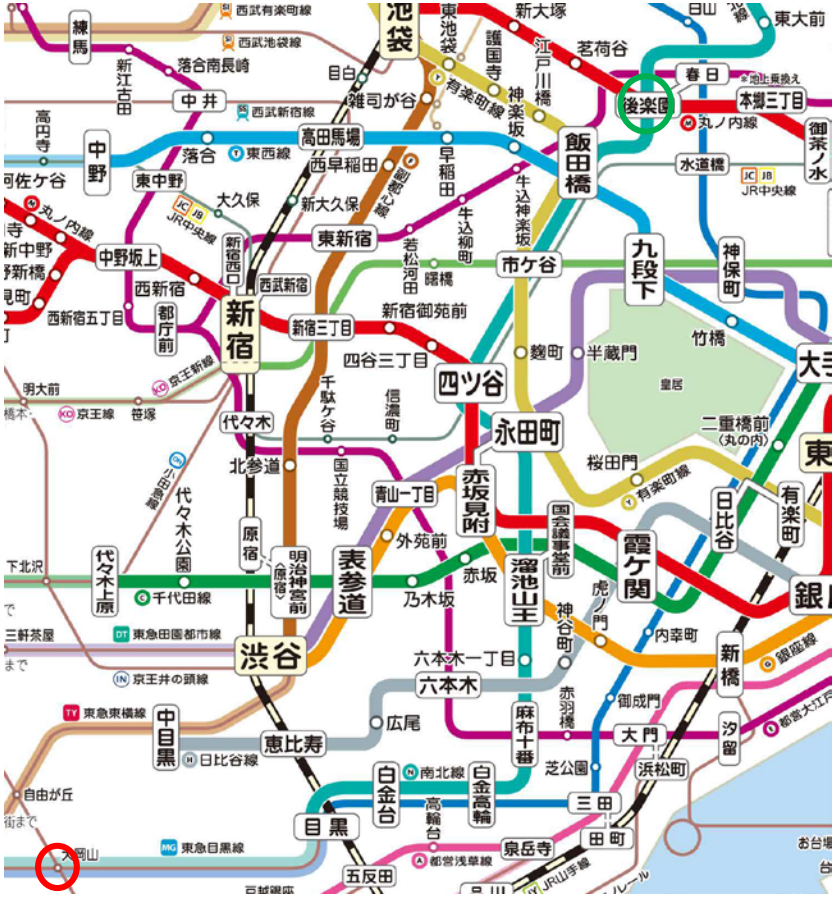
- このとき、$g(n)+h(n)$が小さい、つまり、後楽園から近くてかつ大岡山にも近いノードは**飯田橋**となる。
- よって、**飯田橋**駅を中間ノードnとする。

- これを繰り返していく。



# A*アルゴリズム

- 次に、飯田橋駅から直接つながっている駅である、**南北線市ヶ谷**駅、**東西線九段下駅**、**神楽坂**駅を中間ノード候補とする。
- 先ほどと同様の操作により、次の中間ノードは**市ヶ谷**となる。
- これを繰り返していくと、**四ツ谷**、**赤坂見附**、**青山一丁目**、**表参道**、**渋谷**、**中目黒**、**自由が丘**、**大岡山**という経路が導き出される。
- 実際は乗り換えなどがありこのルートを使う人は存在しないと思われる。



# A*アルゴリズム

- ここで、$n$までの費用の推定値$g(n)$は、探索済みの区間となるためある程度調べると真値$g^*(n)$に近づけることができる。

- しかし、これから先の費用である$h(n)$はどうしてもわからない。
- そのため、概ね正しいであろう値を$h(n)$に導入する。
- ユークリッド距離が縮まればおそらく$h(n)$も縮まるであろうということが言えるため、二次元マップにはユークリッド距離を用いる。

- $h(n)$を、ヒューリスティック(huristic)関数という。
- これは、必ず正しい答えを導けるわけではないが、ある程度の精度の答えを導けるような関数である。

# A*アルゴリズム

- A*はDijkstra法と同様に全てのリンクコストが非負の時適用できる。
- 同様に、$h(n)$も非負でなくてはならない。

- $h(n)$について、真のゴールまでの距離$h^*(n)$を上回らない場合、$h(n)$は許容的(Admissible)であるという。
- 先ほどの条件とあわせて

全てのノード$n$について、

$$0 \leq h(n) \leq h^*(n)$$

- この条件が満たされているならば、求まる経路は最短経路であることが保障されてる。

- もし適当な$h(n)$を導入するといつまで経っても探索が終わらなかったり終わっても不適当な経路が求まってしまう。

# A*アルゴリズム　NetworkX

- 今回は詳細な説明は省くが、NetworkXにDijkstra法と同じようにA*のパッケージが存在する。

- 詳しくは

  https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.shortest_paths.astar.astar_path.html#networkx.algorithms.shortest_paths.astar.astar_path

- Dijkstra法は全てのノードに対して満遍なく探索していくのに対して、　A*は一つのノードに対して近そうな方向から探索していくので、一つのODペアの最短経路を導き出すにはA*が使いやすい。

# ZDD(ゼロサプレス型二分決定グラフ)

- 今までのアルゴリズムは最短経路を効率よく導き出すもの

- ZDDは、一つのODペアの**全経路**を効率よく見つける方法である。
  - ここにおける全経路とは同じ道、同じ地点を二度通らない経路とする。

- たとえば、東京駅から品川駅の最短経路は山手線なりで直接向かうのが一番早いが、東京近郊の範囲で行くルートはとてもたくさんある。

Ex) 東京→千葉→成田→我孫子→新松戸→南浦和→赤羽→新宿→品川

- 全経路数は4152859通りも存在する。

- 全経路が出せるとどこを通らない経路やこの移動距離以下の経路などの融通がとても効く。

# ZDDとは

- 例えば、a,b,cの三つの要素の組み合わせの組み合わせは何通りか？
- まず、a,b,cの組み合わせが$2^3$通り存在する。(λ, a, b, c, ab, ac, bc, abc)
- この$2^3$通りの要素を採用するか採用しないかの2通りずつなので、組み合わせの組み合わせは$2^{2^3}$通り存在する。

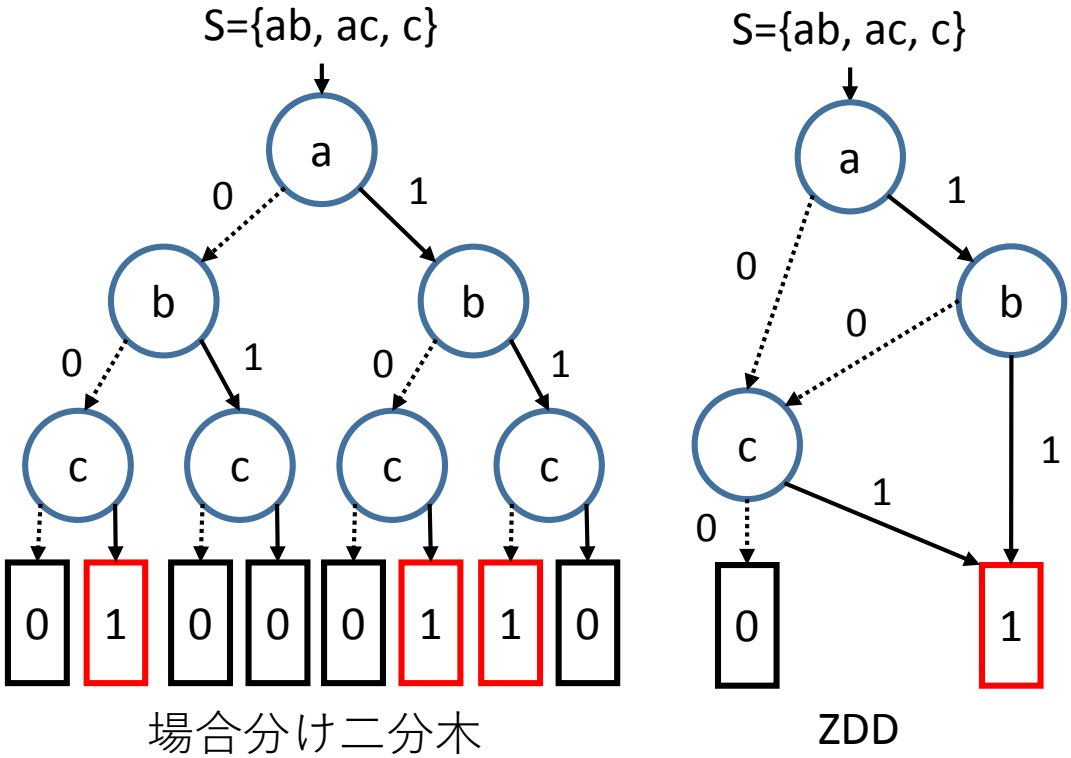
Ex) ϕ, {a,b,bc}, {λ, b, ac, abc}, {λ, a, b, c, ab, ac, bc, abc}

- 三つの要素だと256通りで通常のやり方でも列挙できるが、要素が10個になると$2^{2^{10}}$通りとなり、桁数が300桁以上になってしまい単純な方法では一生かかっても終わらなくなってしまう。
- そこでZDDはこの組み合わせ集合のグラフの簡略化を可能にする。

# ZDDによるグラフ表現

- S = {ab, ac, c}という要素集合を考える。
- 通常の場合分け二分木とZDDによる表現は以下のようになる。

- 圧倒的に簡略化される
- よくよくたどってみるとZDDは三つの要素を表している。

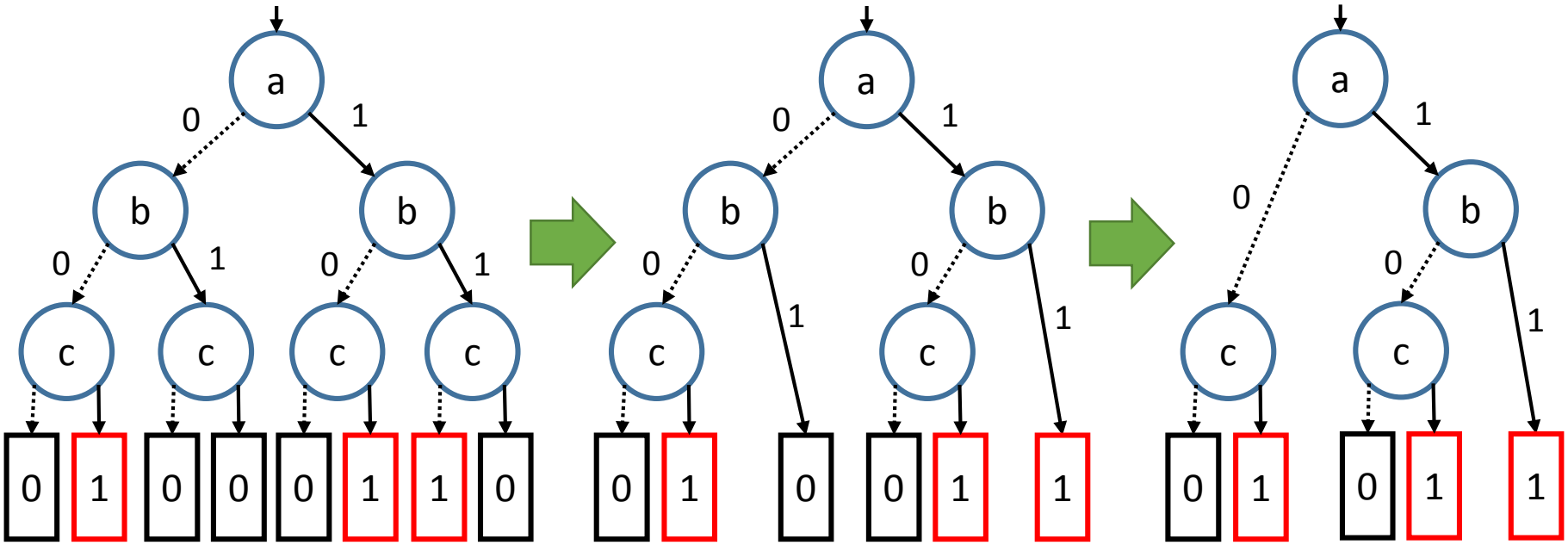


# ZDDの生成法

- 二つ生成法によってZDDを生成する。

(a) 冗長接点の削除

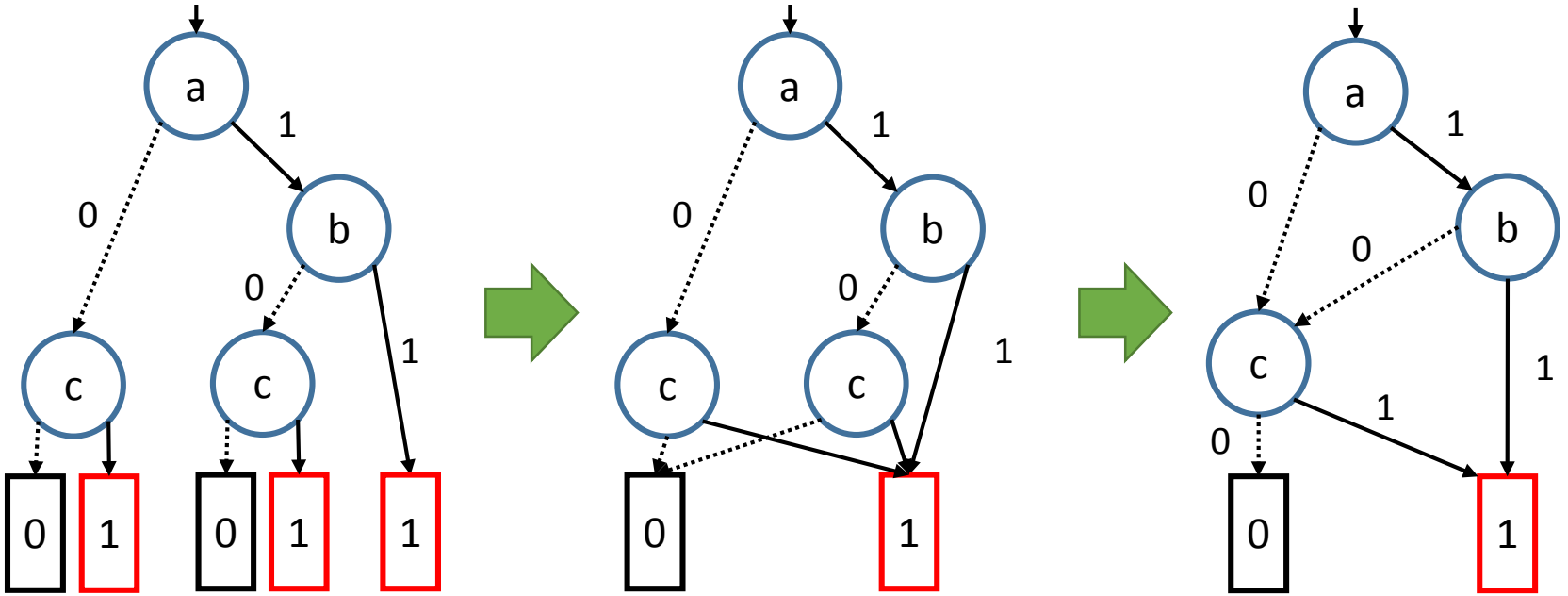
あるノードからの1の矢印が0の終端を指しているときそのノードを消す



# ZDDの生成法

(b) 等価接点の共有

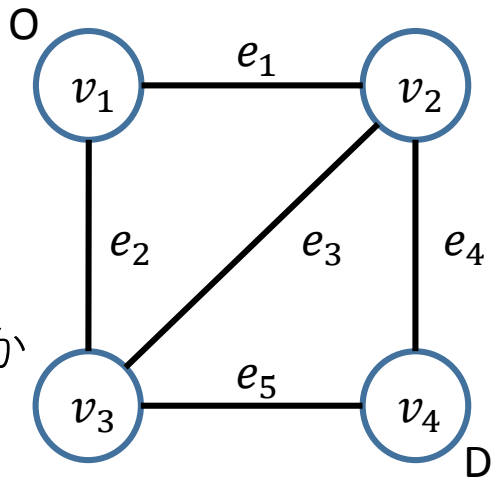
同じ種類のノードからでるリンクが同じ終端を指している時そのノードを共有




- この手順をどの順番で踏んでも同じ形が得られる。
- この最終形をZDDという

# ZDDのOD経路列挙方法

- この方法がどうやって経路列挙に役立つのか？
- 例えば右のようなネットワークがあるとする。
- リンクの組み合わせは$2^5$通り存在する。
- しかし、この中でOD経路になりうるものは4つしかない。
- 集合で表すと$\{e_1e_4, e_1e_3e_5, e_2e_5, e_2e_3e_4\}$となり、ZDDをうまく適用できそうな雰囲気となる。
- では、OD経路を効率よく見つけられれば経路列挙にZDDを役立てることができるのではないか？



# OD経路が満さなければいけない性質

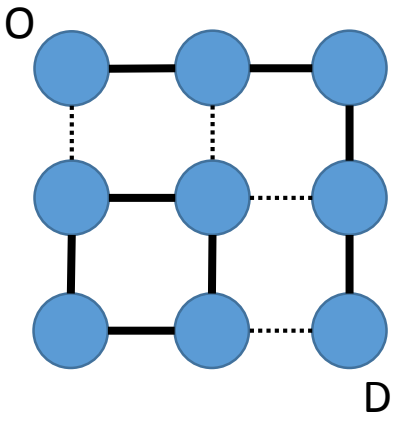
- OD経路になるための必要十分条件は以下の二つである。

Ⅰ. ノードに接続している経路内のリンクの本数を次数とすると、
始点、終点の次数が1、それ以外のノードの次数は0または2となる

Ⅱ. サイクルを含まない

- Ⅰの条件は想像するとわかるが、サイクルとは何か？

- 右のネットワークは条件Ⅰは満たしているが、これはOD経路とはいえない。

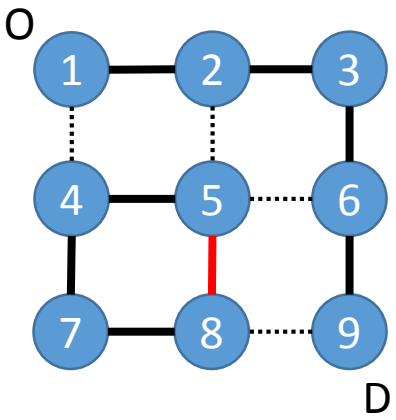



- サイクルがある場合その周辺ノードは条件Ⅰを満たしてしまうので、条件Ⅱが必要であることがわかる。

# 性質の条件式

- 各ノードの次数を記憶する配列変数degを用いる、次数をdeg[$v$]とする。
- 各ノードに対してdeg[$v$]の条件が満たされていれば、条件Ⅰを満たしている。

- また、そのノードが接続している一番小さいノード番号を記憶するcompという配列を用いる。

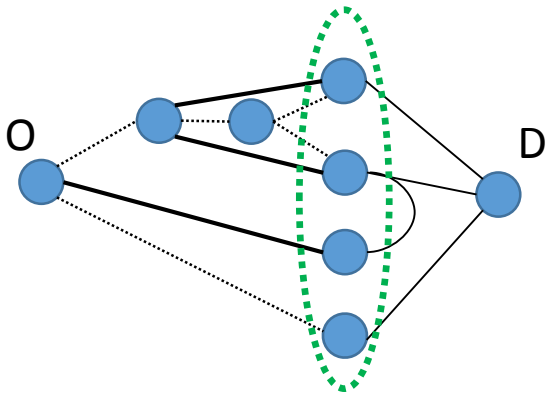



- 右図においてcomp[5],comp[8]は4となる。
- ノード同士を繋ぐ際、サイクルが発生するとcompが等しくなるため、サイクルの有無の判定ができる。

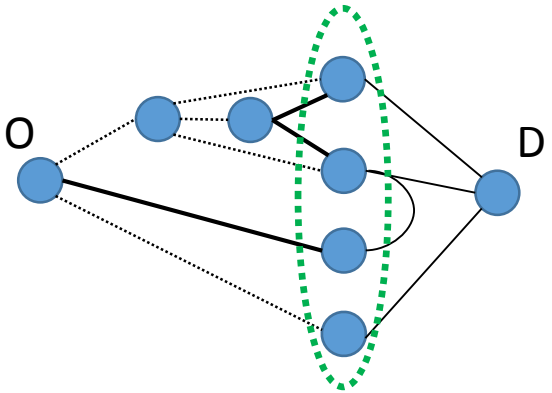
- 各ノードに対してcomp値が等しいノードを繋ぐことを避ければ、条件Ⅱが満たされている。

# フロンティア

- 右の二つのグラフにおいて、点線楕円の左側は処理済みの辺、右側は未処理の辺である。
- このとき、二つのグラフのODを完成させようとすると、楕円の右側は同じ状態となる。




- このような楕円内の頂点集合をフロンティアと呼ぶ。
  - フロンティアの厳密な定義は既に処理したリンクと未処理のリンクが双方接続しているノードである。
- この判定はdegとcompを用いて行う。
  - deg,compが一致するようなときがフロンティアである。
- この場合は同じ計算をすればいいため状態を共有する。

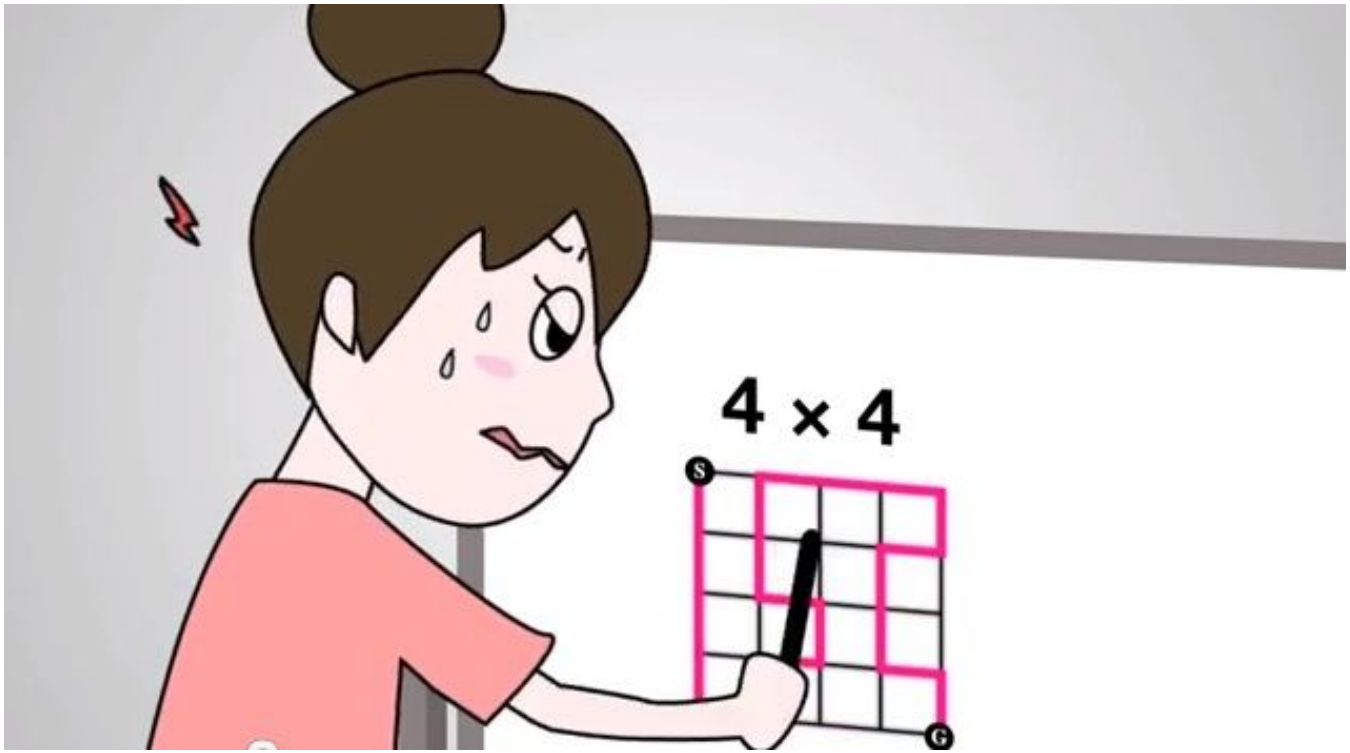


# ZDDのまとめ

- 先ほどのフロンティアについて、フロンティア内に属しているノードのみdeg,compの記憶を行えばいい。
  - 左のノードは再び計算の対象にはならないため
- 以上のような機構を用いて、効率よくODペアを列挙することができる。
- 内容次第では、全ノードを通る方法(巡回セールスマン問題)などへの応用も効く。
- ZDDのコードは本に記載されていたが、古いコードであったため不具合が多発した、興味があればぜひ本を手に取ってほしい。

# まとめ

- まだ紹介していないさまざまな経路探索アルゴリズムが存在する。
- アルゴリズムには一長一短があり、正しい知識を持って最も効率のよいものを使いこなすことが我々には必要なのではないか。



# 参考文献


- Python ダイクストラ法で最短経路を求める
  - https://qiita.com/shizuma/items/e08a76ab26073b21c207
- NetworkX
  - https://networkx.github.io/documentation/networkx-1.10/reference/index.html
- 交通ネットワークの均衡分析(土木学会)
- 駅データ.jp
  - http://www.ekidata.jp/
- 超高速グラフ列挙アルゴリズム(森北出版株式会社)