離散数学

# 最短経路問題

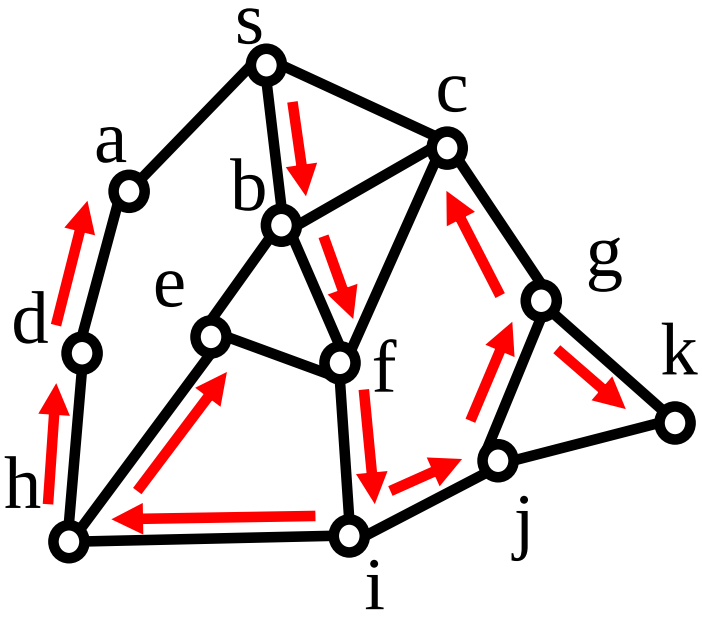
落合 秀也

その前に・・・前回の話

# 深さ優先探索アルゴリズム

開始点sから 深さ優先探索を行うアルゴリズム

```
S.push(s)
While S is not empty
  v := S.pop()
  If F[v] = false Then,
    F[v] := true
    Foreach node u in Adj[v]
      S.push(u)
    EndFor
  EndIf
EndWhile
```




時間計算量：O(n+m)

(*) 厳密には初期化処理が必要だが省略している

その前に・・・前回の話

# 深さ優先探索アルゴリズム2
# 再帰呼び出しによる方法

深さ優先探索を行うアルゴリズム（再帰呼び出し版）

```
Function DFS(v)
  If F[v] = false Then,
    F[v] := true
    Foreach node u in Adj[v]
        DFS(u)
    EndFor
  EndIf
EndFunction
```

頂点sから探索を行う場合：

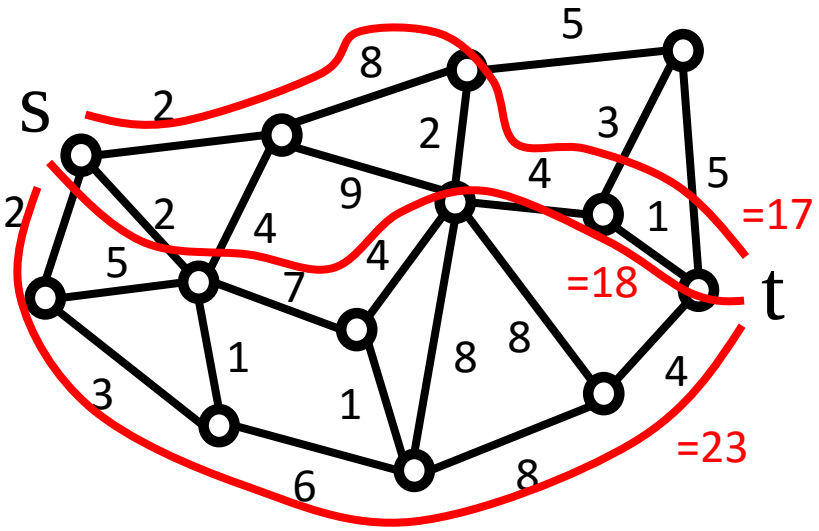
DFS(s)

を実行すれば良い

再帰呼び出しは、実際には、裏ではスタックを使って実行される

# 最短経路問題を考える

- ラベル付(重み付)グラフ G=(V,E)が与えられたとき、頂点sから頂点tへの最短経路 s-t を求めたい
- 辺のラベルの意味:
  - 地点間の道のり
  - 地点間の移動時間
  - 地点間の移動に要する燃料の量
  - 状態の遷移で失う資金



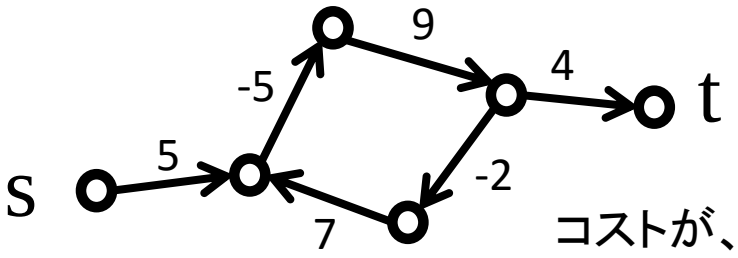

(*) 実際はコスト16の道が存在する

- 最小コストでの移動経路を発見したい

# 最短経路問題の解法

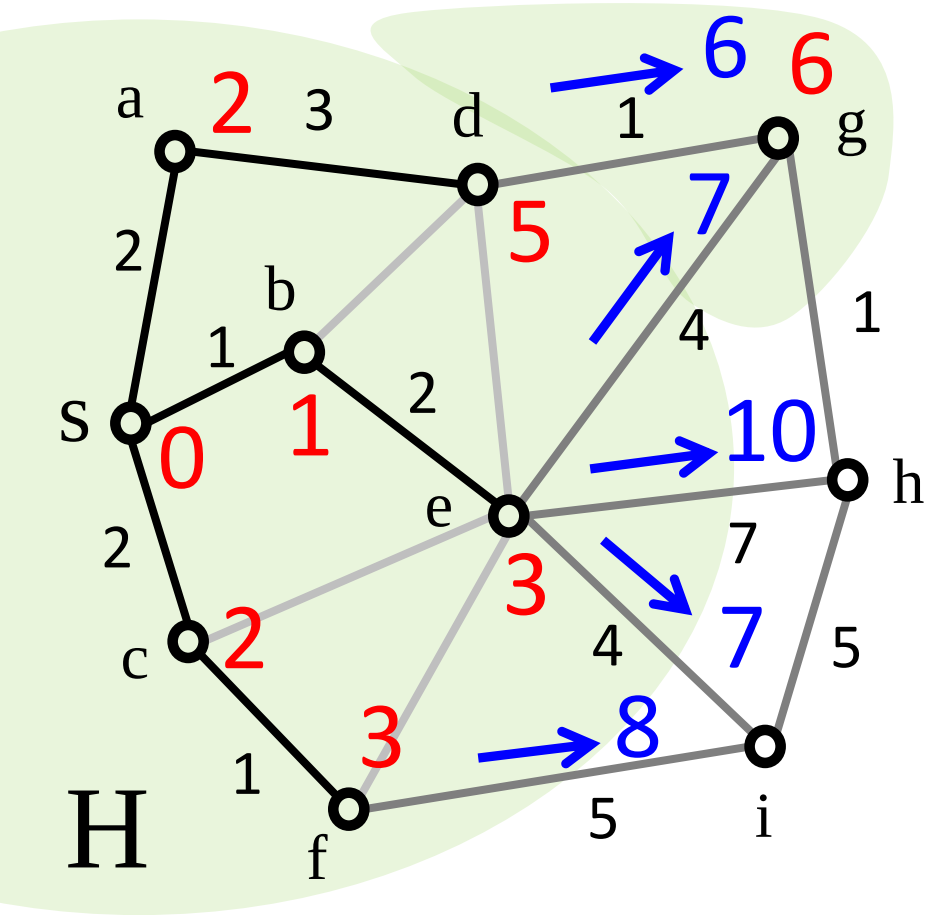
- ダイクストラ法
  - すべての辺の重みが非負の場合に適用可能
  - 貪欲法(Greedy Algorithm)の一種

- ベルマン・フォード法
  - 辺の重みに負があっても良いケースがある（有向グラフで、有向閉路の和が負でない場合）
  - 動的計画法(Dynamic Programming)の一種




コストが、資金の支出の場合、
マイナスは、収入を意味する

# ダイクストラ法(Dijkstra's Algorithm)
## 1959年：Edsger Dijkstraによって考案された




- 考え方 -- G(V,E)に対して
  - 開始点 s∈Vから各頂点への最短経路を求める問題
  - いま、すでに部分Hに関して、最短経路が導出されているとする
    - sからの距離が判明している
  - Hと接する v ∈ V-H に対し、sからの距離を考える
  - その距離の最小値を与えるvを、Hに追加する

赤字: 判明しているsからの最短距離

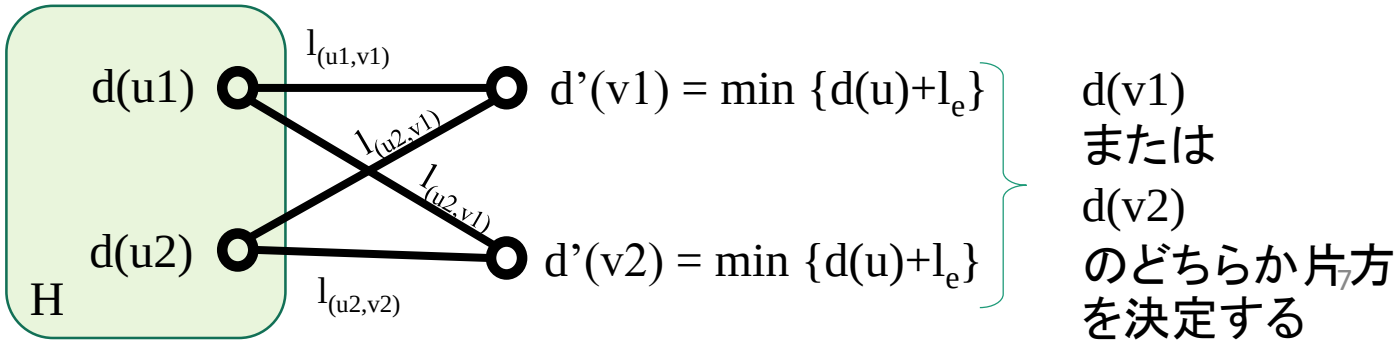
# ダイクストラ法(もう少し正確な定義)

- 用語の定義:
  - グラフG(V,E)において、u, v ∈V, e=(u,v) ∈ Eとする。
  - 辺e=(u,v)の長さを$l_e$あるいは$l_{(u,v)}$で表すとする。
  - 開始点をsとする。d(u)で、sからuまでの最短距離を表すとする。
  - prev(v)で、sからvへの最短経路におけるvの直前の頂点を表す。
- 挙動の定義: Vの部分集合 H を以下のように作成する。
  - 初期状態: H = {s}, d(s)=0, d(v)=∞ (v≠s), prev(v)=null (v∈V) とする。
  - u∈ H で辺e=(u, v)でつながっている v (∈ V-H) を考え、

$$d'(v) = \min_{e=(u,v):u\in H} \{ d(u) + l_e \}$$

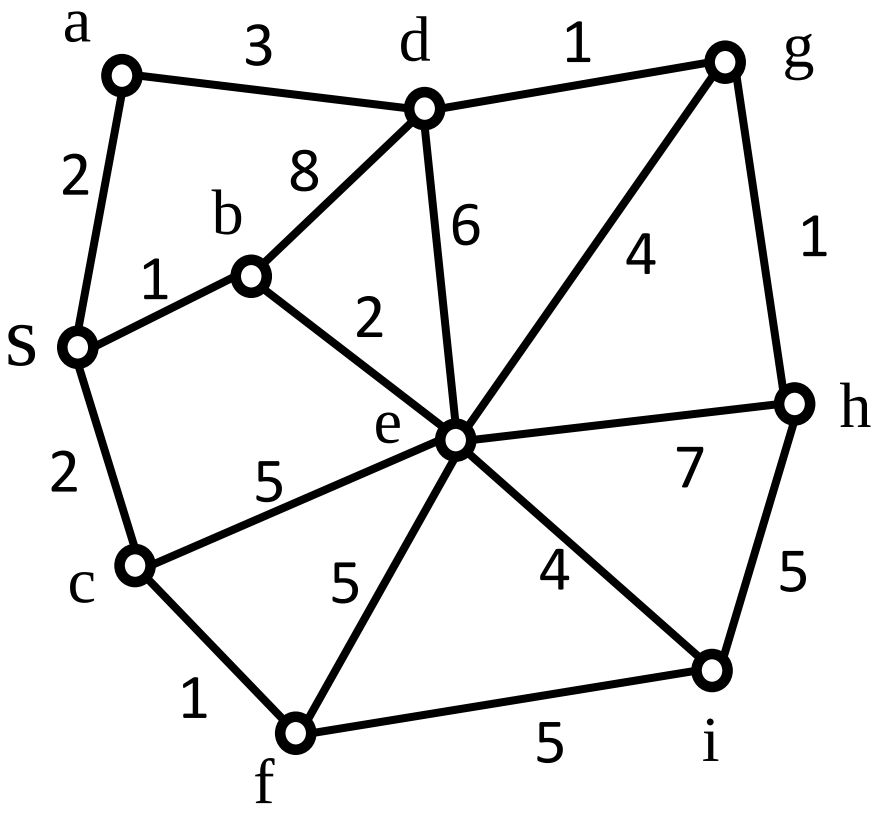
を計算する。

  - v ∈ V-Hにおいて、上記をさらに最小化するvを選定し、そのvをHに追加し、その距離をd(v)とおき、その時のuを用いて、prev(v)=uとする。

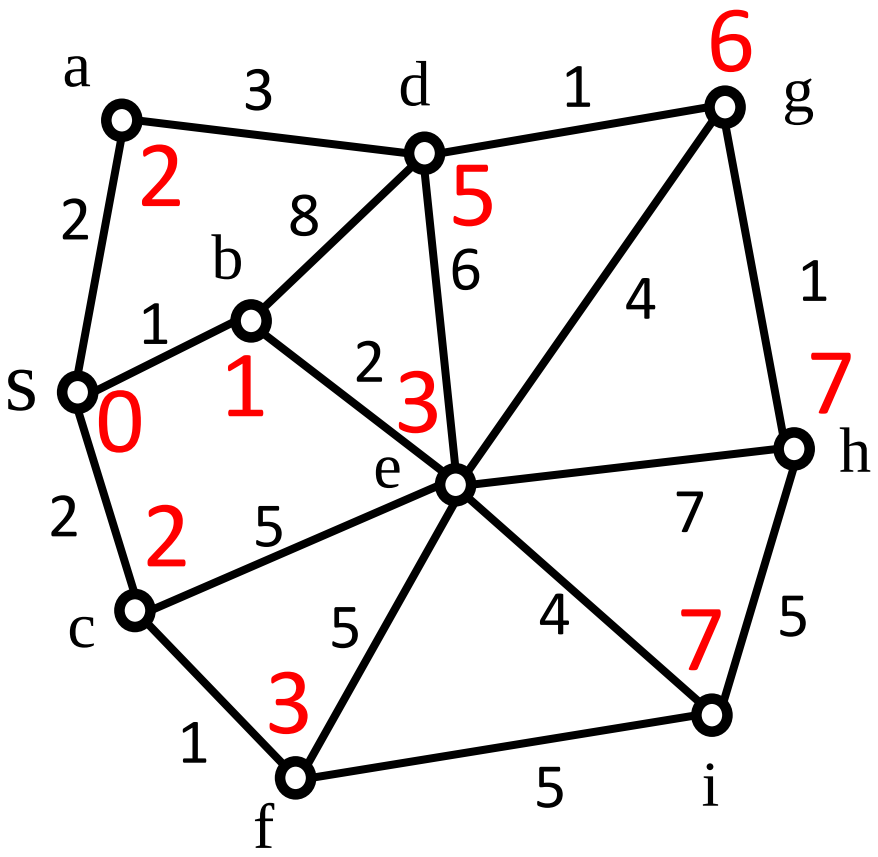


# 練習

- 以下のグラフに対し、ダイクストラ法により、頂点sから各頂点への最短経路とその距離を求めよ。



# 解答:




| 頂点 | 最短経路 | 距離 |
|---|---|---|
| a | s→a | 2 |
| b | s→b | 1 |
| c | s→c | 2 |
| d | s→a→d | 5 |
| e | s→b→e | 3 |
| f | s→c→f | 3 |
| g | s→a→d→g | 6 |
| h | s→a→d→g→h | 7 |
| i | s→c→f→i | 7 |

# ダイクストラ法の実現：アルゴリズムの設計
# 素直に上述の定義に従って考えると・・・

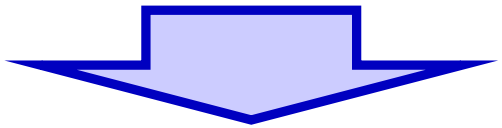
- 準備するもの
  - sからuまでの最小距離を格納する配列: d[u]
    - 初期条件: d[s]=0, (u≠sに対して) d[u]=∞
  - sからvへの最短経路で、vの直前を格納する配列: prev[v]
    - 初期条件: prev[v]=null
  - 辺の長さを与える関数: length(u,v)
  - 最短経路が確定した頂点のリスト: H
    - 初期条件: H = {s}
- 方針

$$d'(v) = \min_{e=(u,v):u \in H} \{ d(u) + l_e \}$$

  - d’[v] を最小にする u∈H, v∈V-H の辺(u,v) を見つける
  - 見つかったら、prev[v]=u, d[v]=d’[v]とおき、vをHに追加する
  - 上記を、H=Vになるまで繰り返す

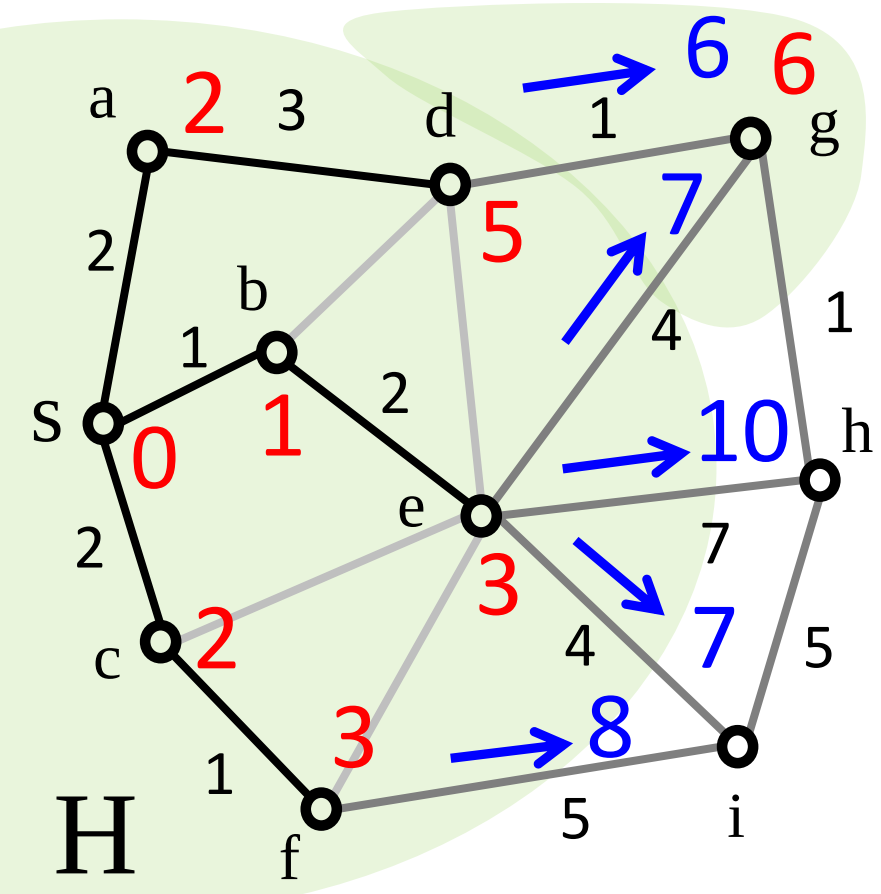
# ダイクストラ法の実現：アルゴリズムの設計
# 工夫をしてみよう

- 最短経路が確定した頂点のリスト(H)を考え、sからの距離d(v)が最小になるv (∈V-H)を、Hに追加していた（そして、これをH=Vになるまで繰り返した）。

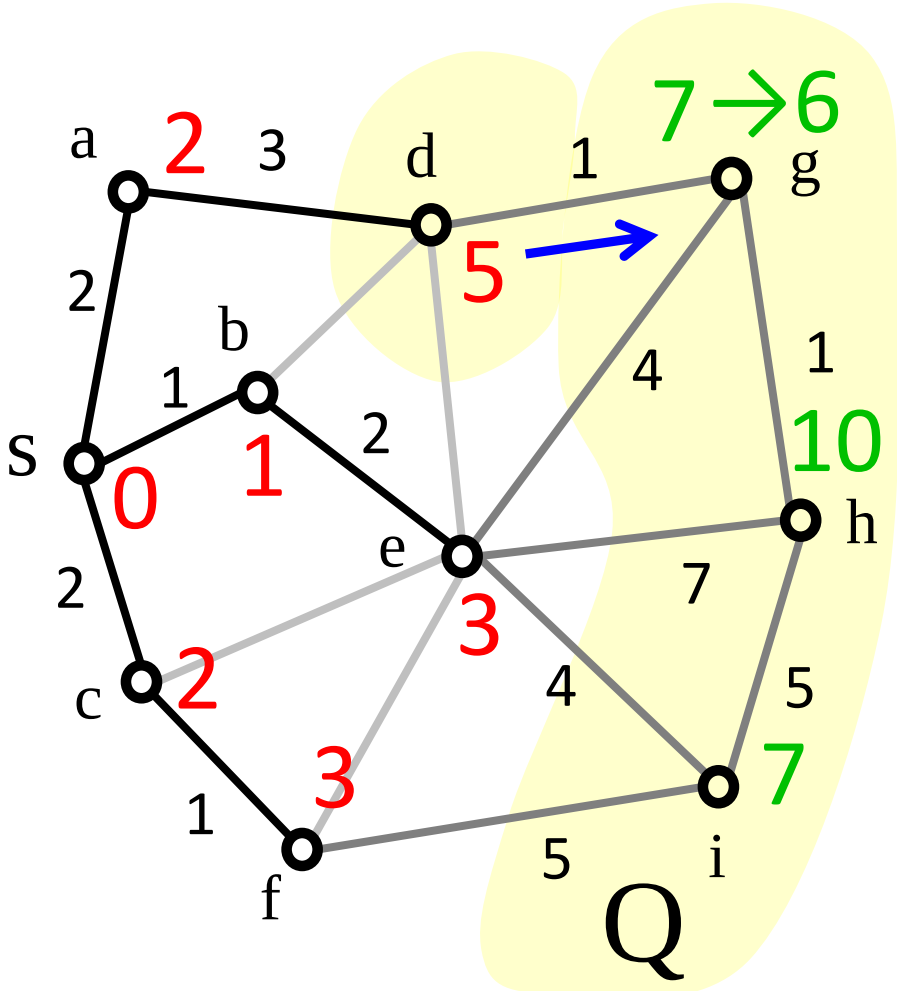


- 最短経路が確定していない頂点のリスト(Q)を考え、sからの距離d(v)を最小にするv(∈Q)を、Qから削除する(そして、これをQが空になるまで繰り返す)。
- 各v(∈Q)において、d(v)の候補を常に計算できていれば、Qから削除される段階で d(v) が最小であれば、そこから新たにd(v)を計算をする必要はない。

# ダイクストラ法の実現: 二つのアプローチ



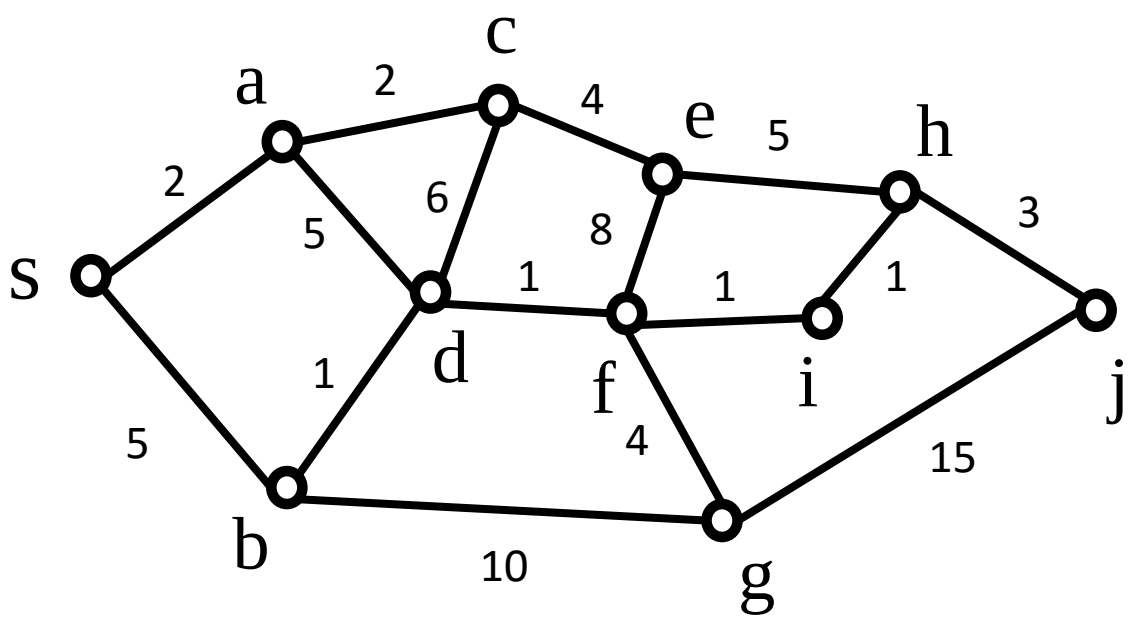

その時点の周辺について計算
最小値を与える頂点を取り込む




最小値を与える頂点を除外する
除外された周辺の頂点までの距離を計算する

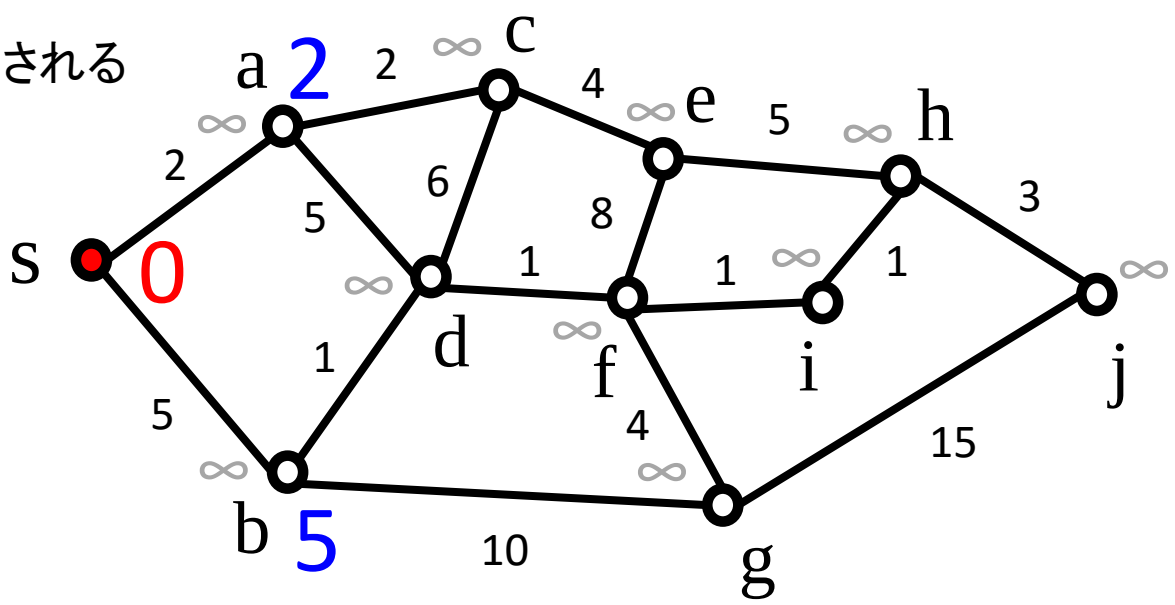
# 練習

- もう一つのダイキストラ法(Qの中から最小値を与える頂点を取り除く方法)で、以下の頂点sから各頂点までの最短経路を求めよ。



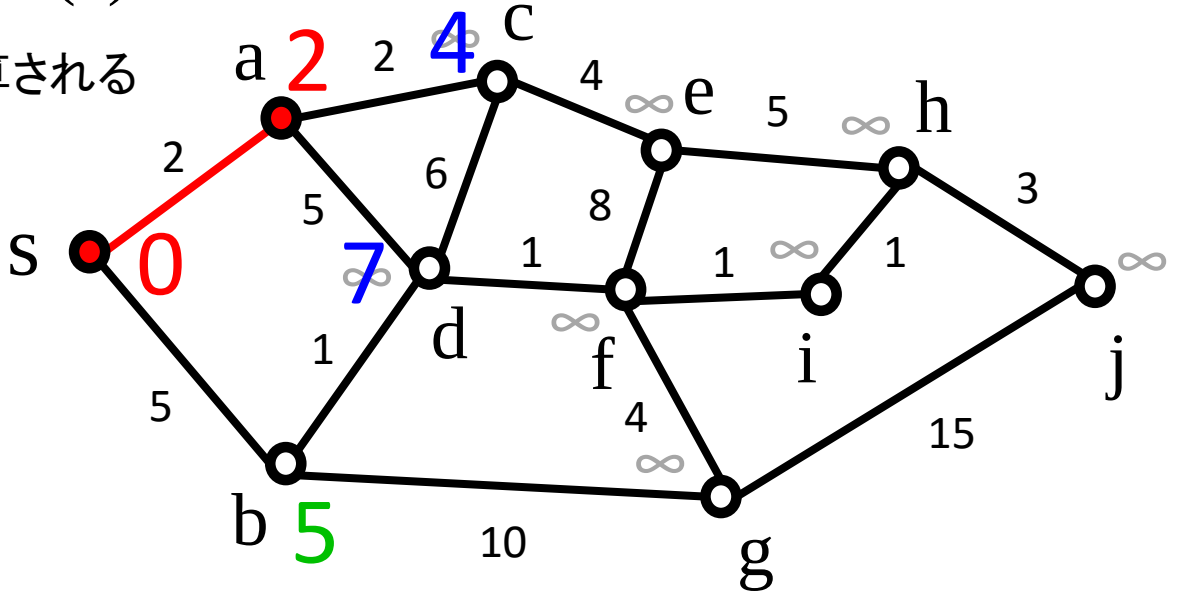
1. d[s]=0と置く。Qからはsが取り出される

sに隣接する a, bの距離が計算される



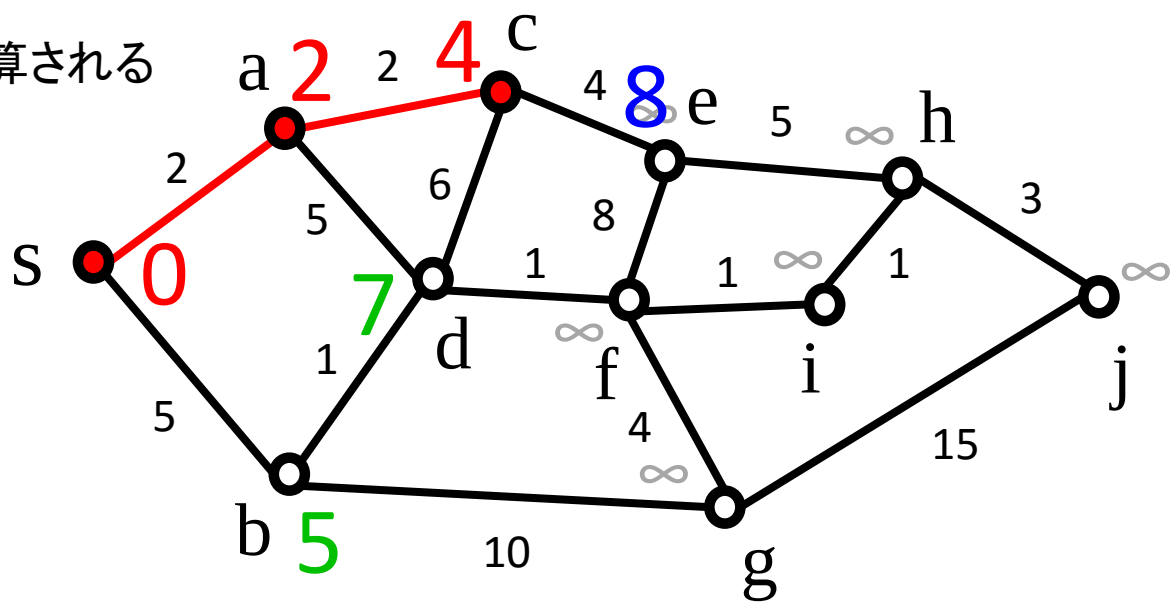

2. Qから、Qの中で最小のd(u)=2を与える a が取り出される

a に隣接する c, dの距離が計算される



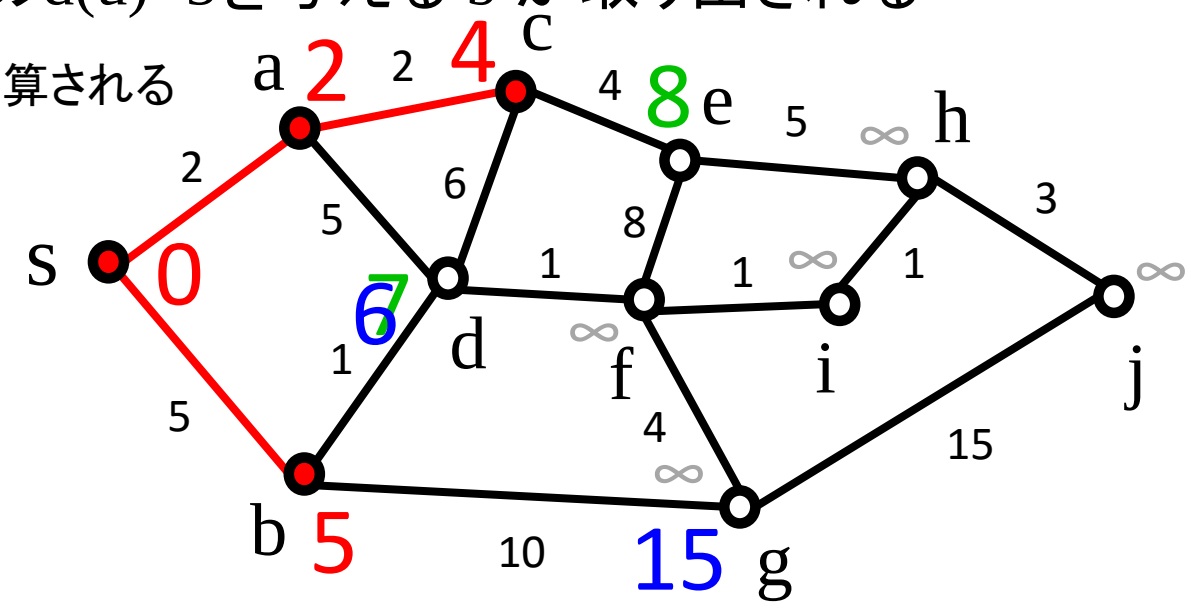
## 3. Qから、Qの中で最小のd(u)=4を与えるcが取り出される

cに隣接するd, eの距離が計算される



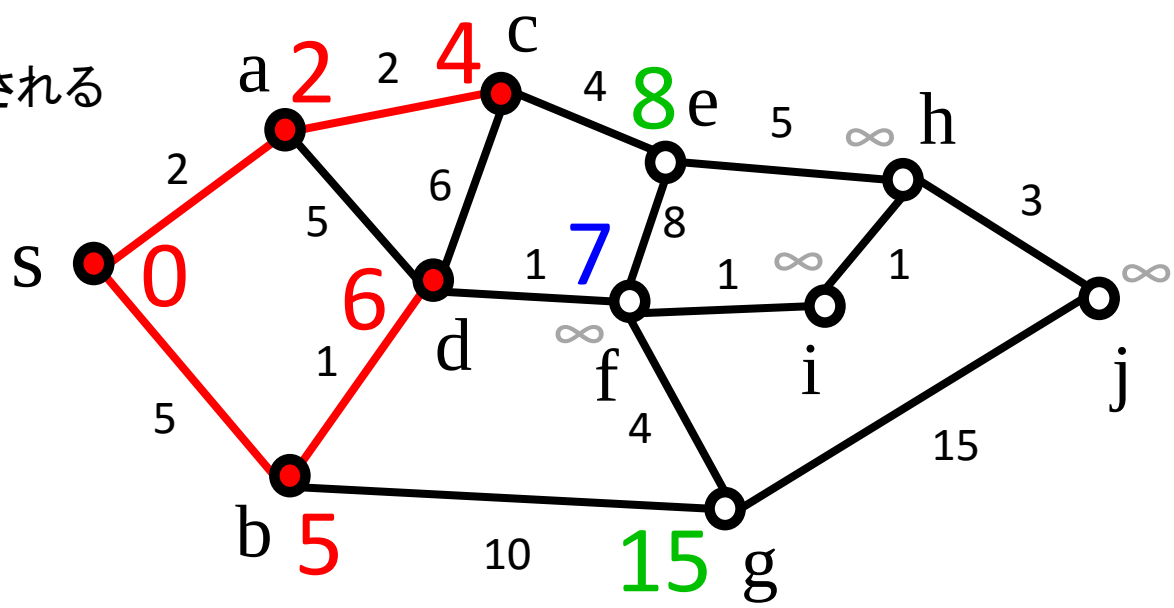

## 4. Qから、Qの中で最小のd(u)=5を与えるbが取り出される

bに隣接するd, gの距離が計算される



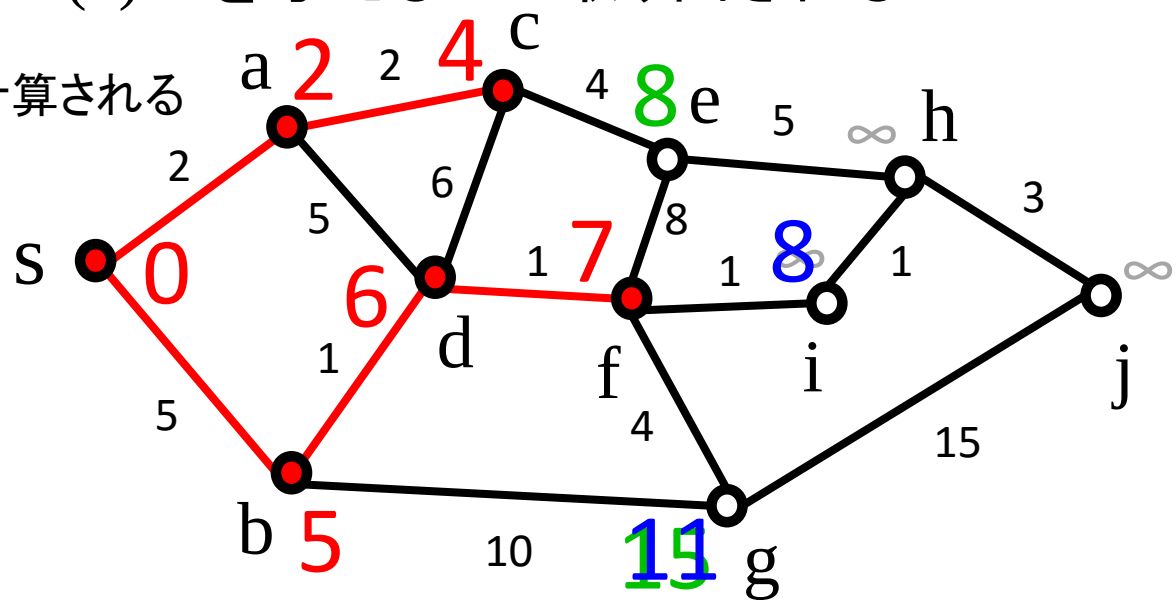
## 5. Qから、Qの中で最小のd(u)=6を与えるdが取り出される

dに隣接するfの距離が計算される



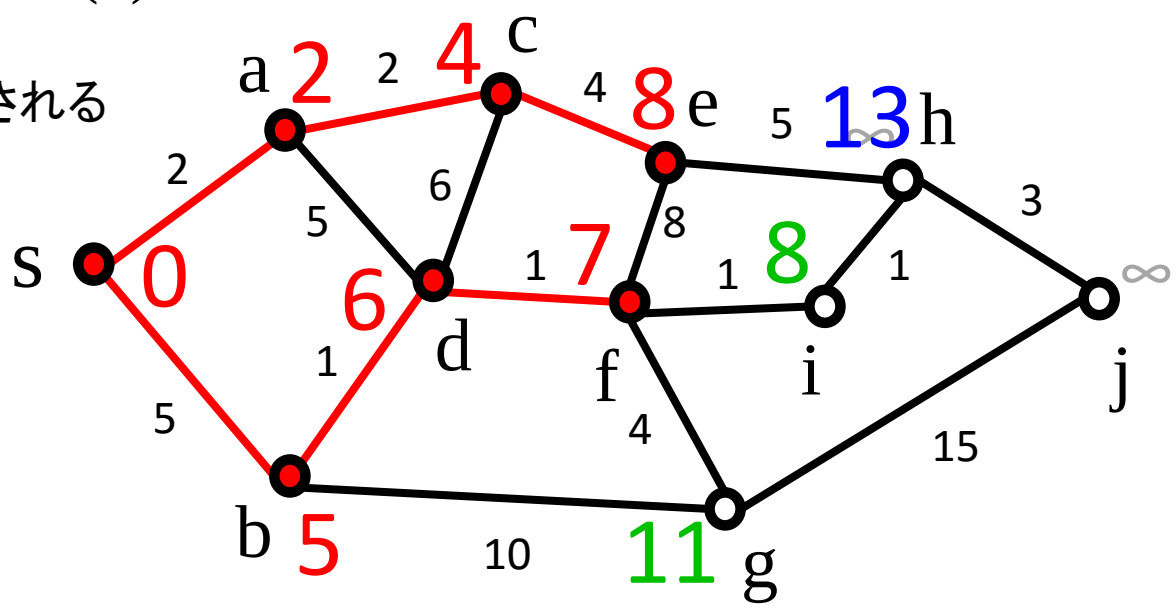

## 6. Qから、Qの中で最小のd(u)=7を与えるfが取り出される

fに隣接するe, i, gの距離が計算される



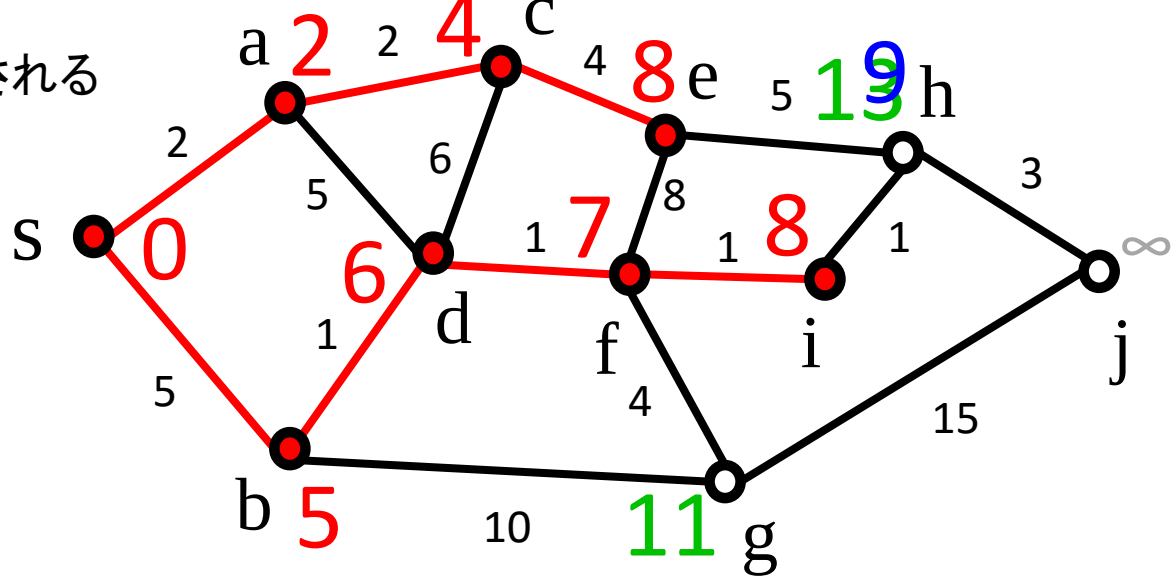
## 7. Qから、Qの中で最小の d(u)=8 を与える e が取り出される

e に隣接する h の距離が計算される



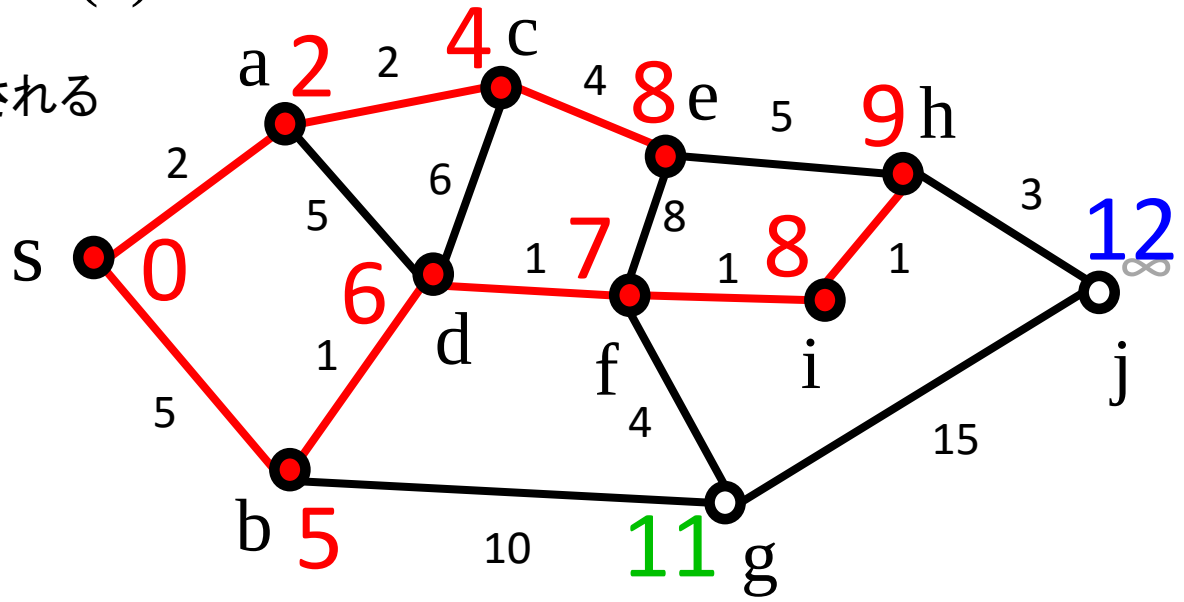

## 8. Qから、Qの中で最小のd(u)=8を与える i が取り出される

i に隣接する h の距離が計算される



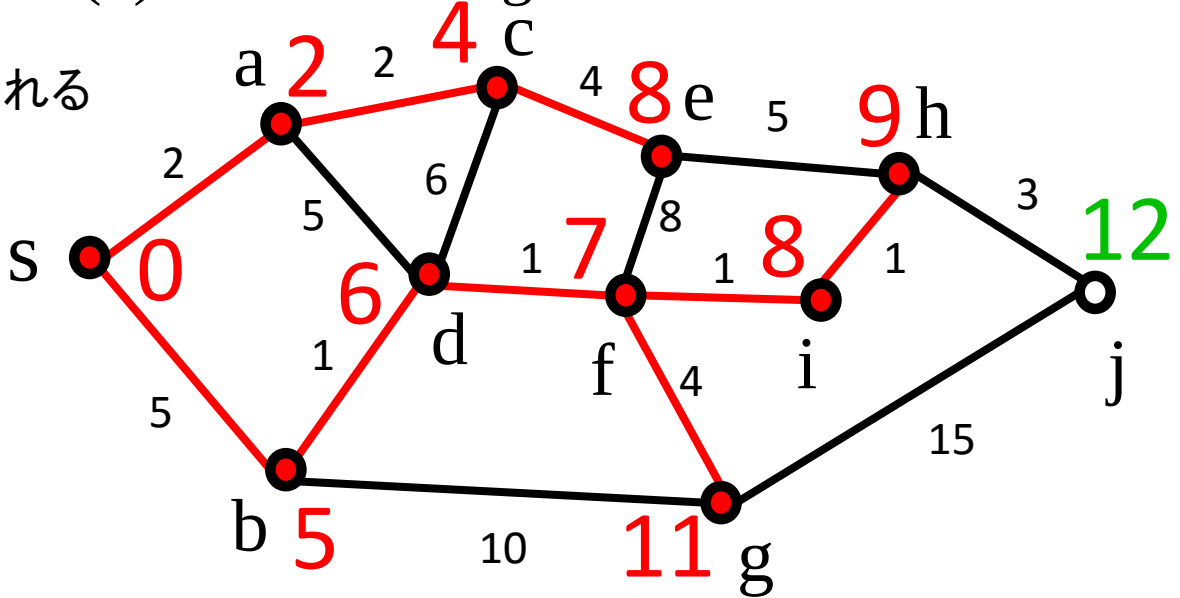
9. Qから、Qの中で最小の d(u)=9 を与える h が取り出される

h に隣接する j の距離が計算される



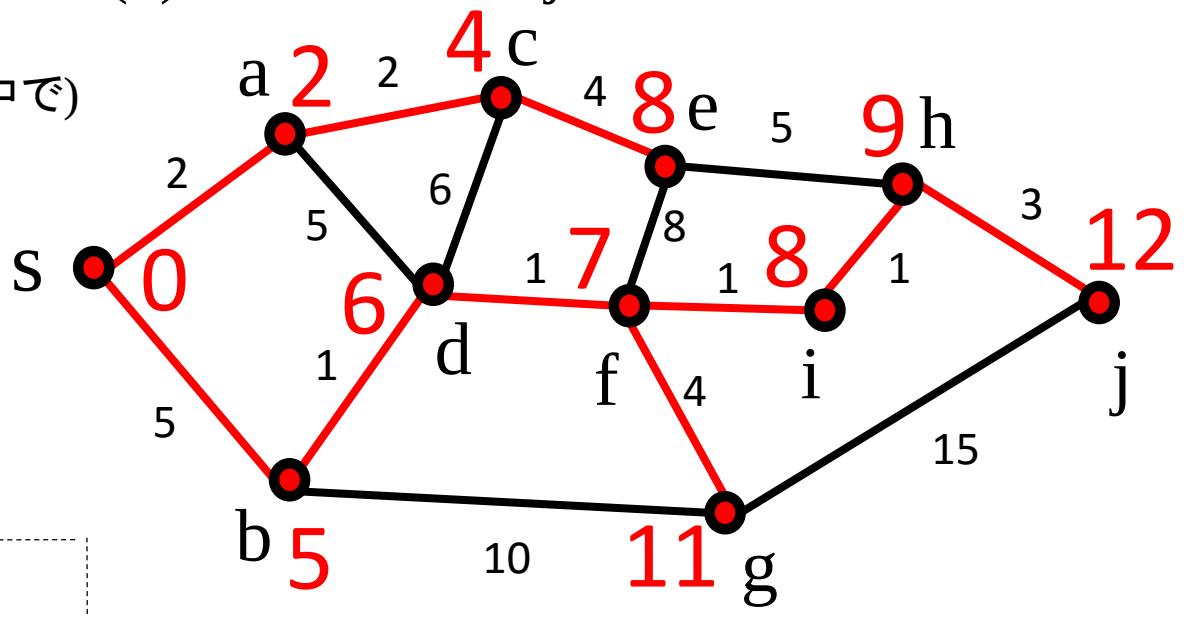

10. Qから、Qの中で最小のd(u)=11を与える g が取り出される

g に隣接する j の距離が計算される



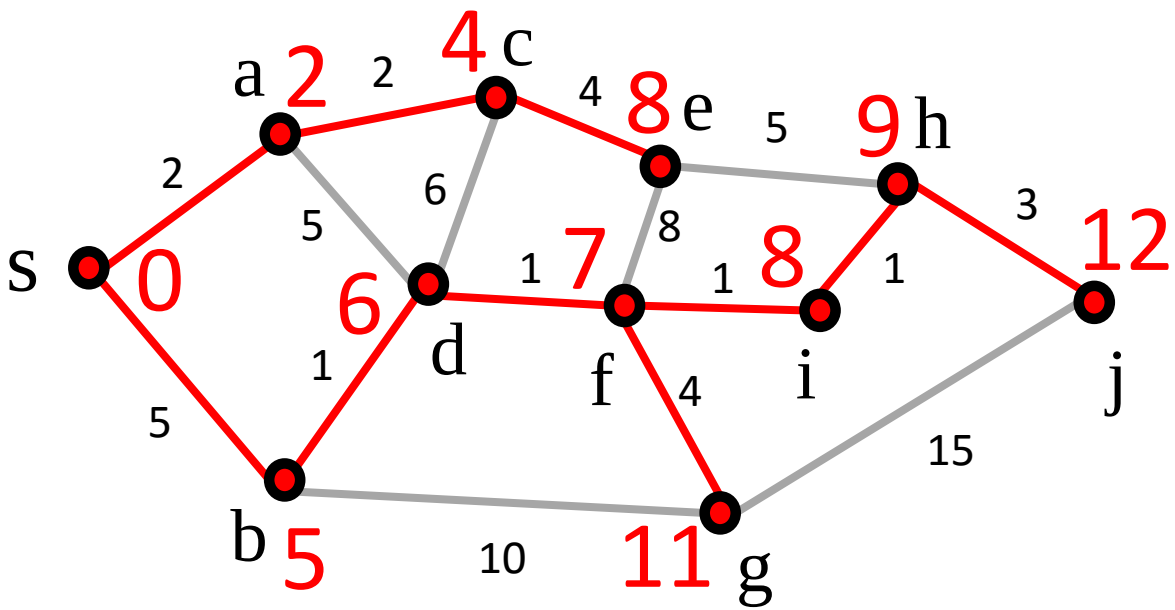
# 11. Qから、Qの中で最小の d(u)=12 を与える j が取り出される

j に隣接する 頂点はなし(Qの中で)



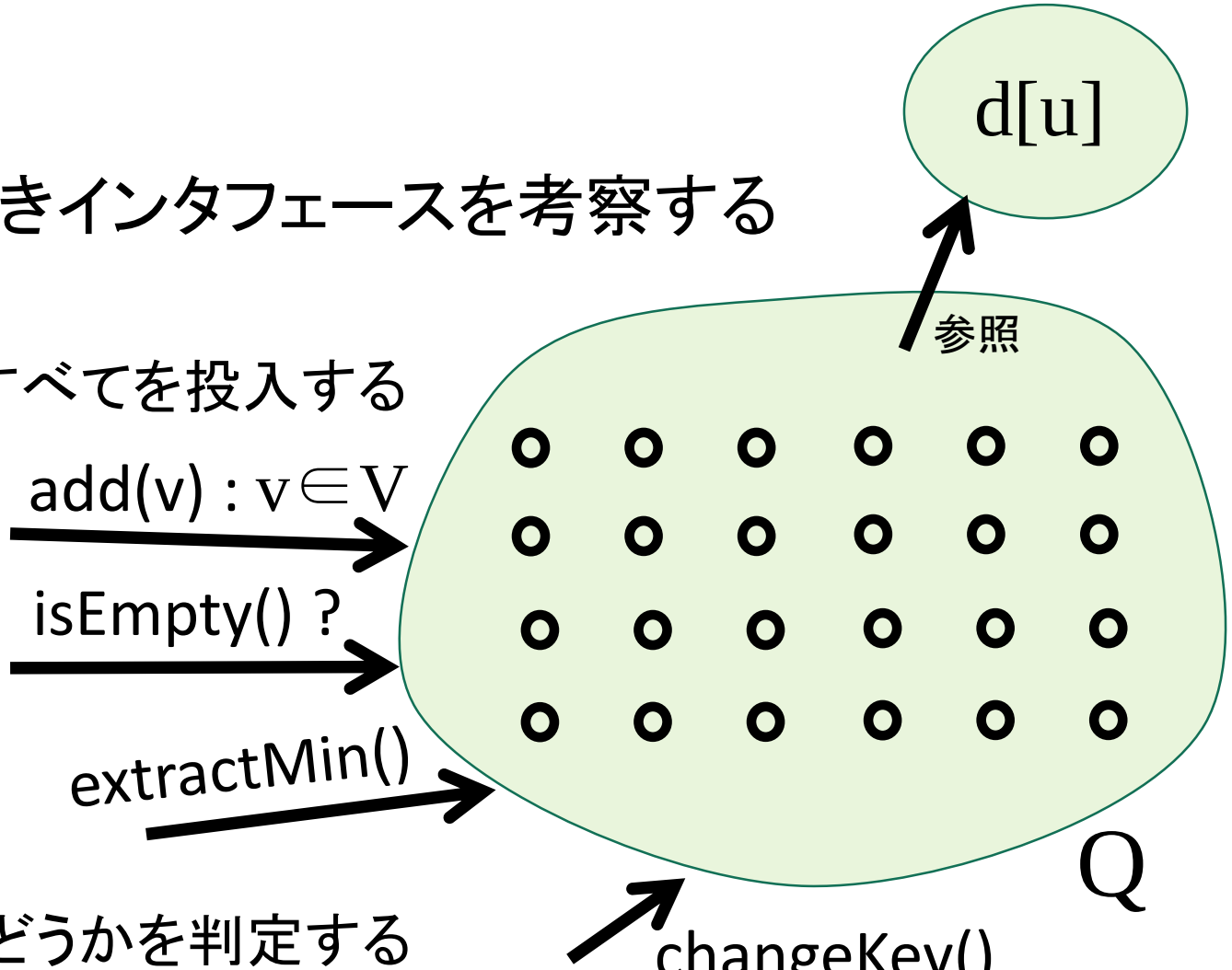

# 12. Qが空なので、終了

| 頂点 | 最短経路 | 距離 |
|---|---|---|
| a | s→a | 2 |
| b | s→b | 5 |
| c | s→a→c | 4 |
| d | s→b→d | 6 |
| e | s→a→c→e | 8 |
| f | s→b→d→f | 7 |
| g | s→b→d→f→g | 11 |
| h | s→b→d→f→i→h | 9 |
| i | s→b→d→f→i | 8 |
| j | s→b→d→f→i→h→j | 12 |



# 装置 Q

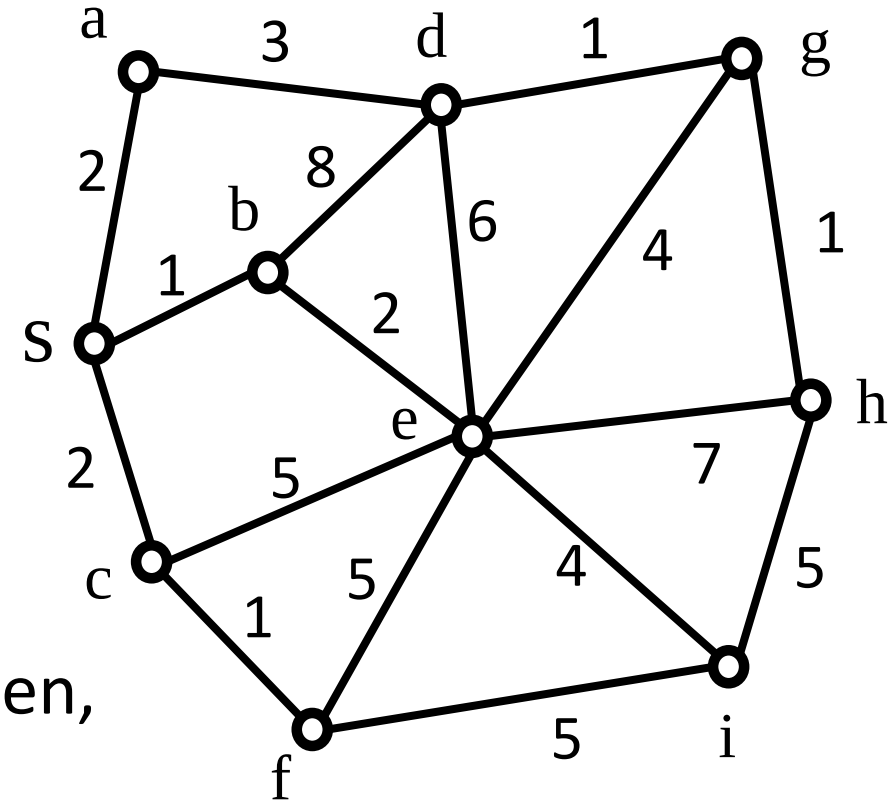
- Qの持つべきインタフェースを考察する
- 初期化時:
  - QにVのすべてを投入する
- 実行時:
  - Qが空かどうかを判定する
  - Qから、d(u)を最小とするuを取り出す
  - Qの内部をdで整理する (Qがヒープの場合)



# ダイクストラのアルゴリズム

```
Foreach v∈V
   d[v] := ∞, prev[v] :=null
   Q.add(v)
EndFor
d[s] :=0

While Q is not empty
  u := Q.extractMin()
  Foreach v in Adj[u]
    If d[v] > d[u] + length(u, v) Then,
      d[v] := d[u] + length(u, v)
      prev[v]=u
    ( Q.changeKey()  --  if the implementation of Q is a heap )
    EndIf
  EndFor
EndWhile
```



# ダイクストラ法: 計算量の考察

```
While Q is not empty
  u := Q.extractMin()
  Foreach v in Adj[u]
    If d[v] > d[u] + length(u, v) Then,
      d[v] := d[u] + length(u, v)
      prev[v]=u
  (    Q.changeKey()    )
    EndIf
  EndFor
EndWhile
```

- While文の内部は n回実行される。
  - Foreach文の内部は nm回実行される。

➔ 実装によっては O(nm) となりえる。

- Q に リストや配列を使う場合
  - Q.extractMin()に O(n) の時間を要する
  - Q.extractMin()は n回 呼び出される

➔ $O(n^2)$

- Q に 2分ヒープを使う場合
  - Q.extractMin()に O(log n)の時間を要する
  - Q.extractMin()の呼び出しはn回ある → O(n log n)
  - d[v]の更新に伴うヒープの組換え処理に、O(log n)の時間を要する
  - d[v]の更新は、m回発生する → O(m log n)

➔ O( (n+m) log n )

# 装置Qの実装1：リストを使った場合

- Q.extractMin()

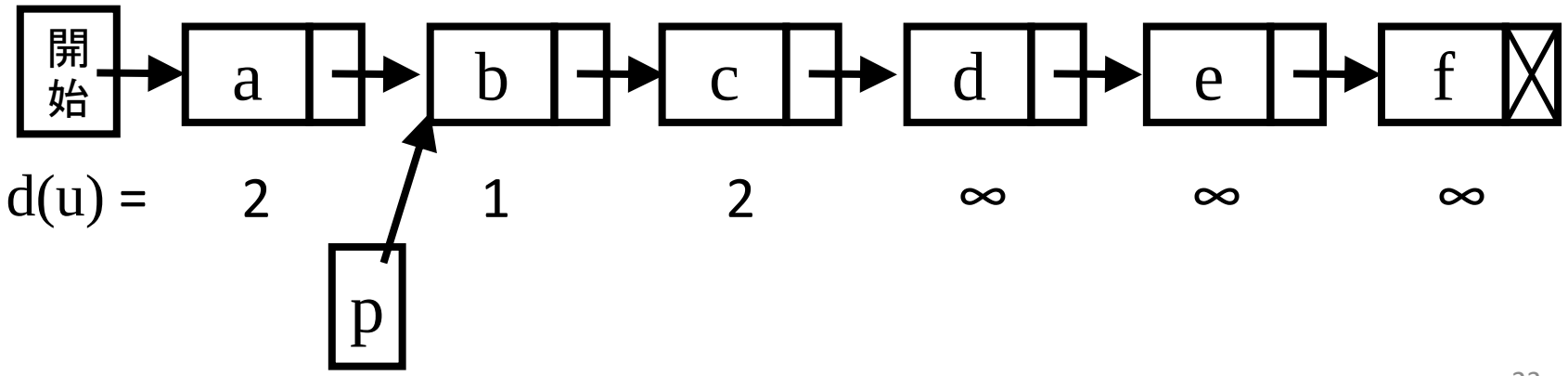
  ポインタ p を用意

  リストのすべての要素に対して、順にd(u)を計算

  より小さいd(u)が発見されたら p に u へのポインタを設定する

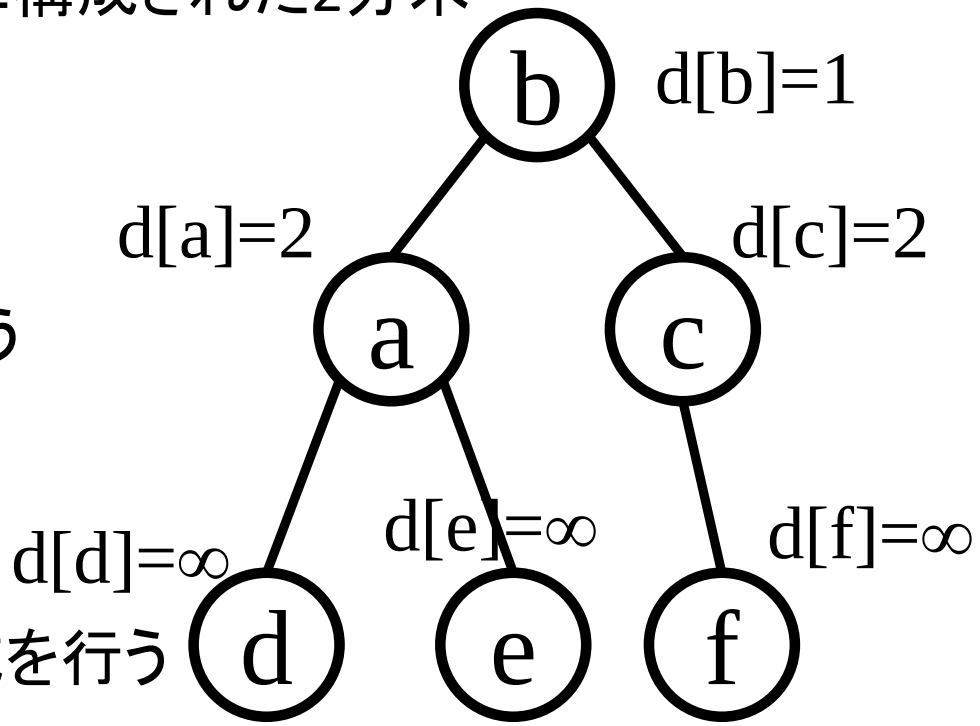
  すべての要素を確認したら、

  p の先を、リストから除外し、その値を返す

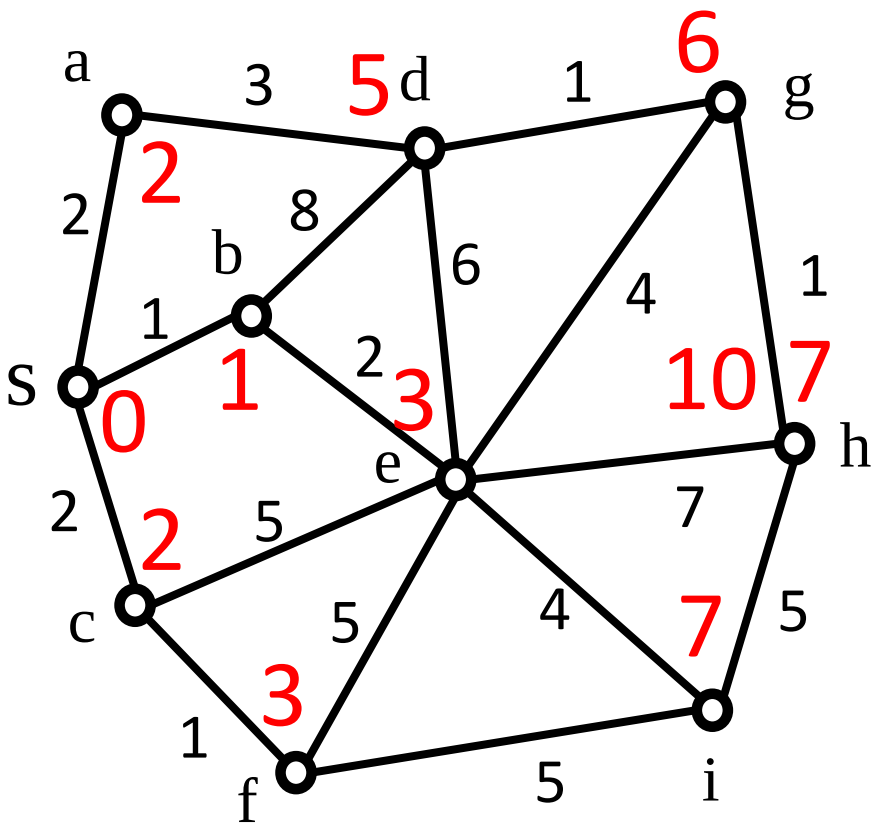


# 装置Qの実装2:ヒープを使った場合

- 2分ヒープ
  - 親≦ 子の関係になるように構成された2分木
  - 根は最小となる
- Q.extractMin()
  - 根を取り出し、再構成を行う
    ➔ 計算量 O(log n)
- Q.changeKey()
  - dの変化に合わせて再構成を行う
    ➔ 計算量 O(log n)



# ベルマン・フォード法
# Bellman Ford Algorithm




## 基本的な考え方

- 開始点sからsへの距離を0とする
- 各頂点の開始点sからの距離は、辺の重みを加算しながら、sから拡散するように伝搬させる。
- 各頂点では距離の最小値を採用する。
- 上記を、頂点数-1回行う

# ベルマン・フォード法

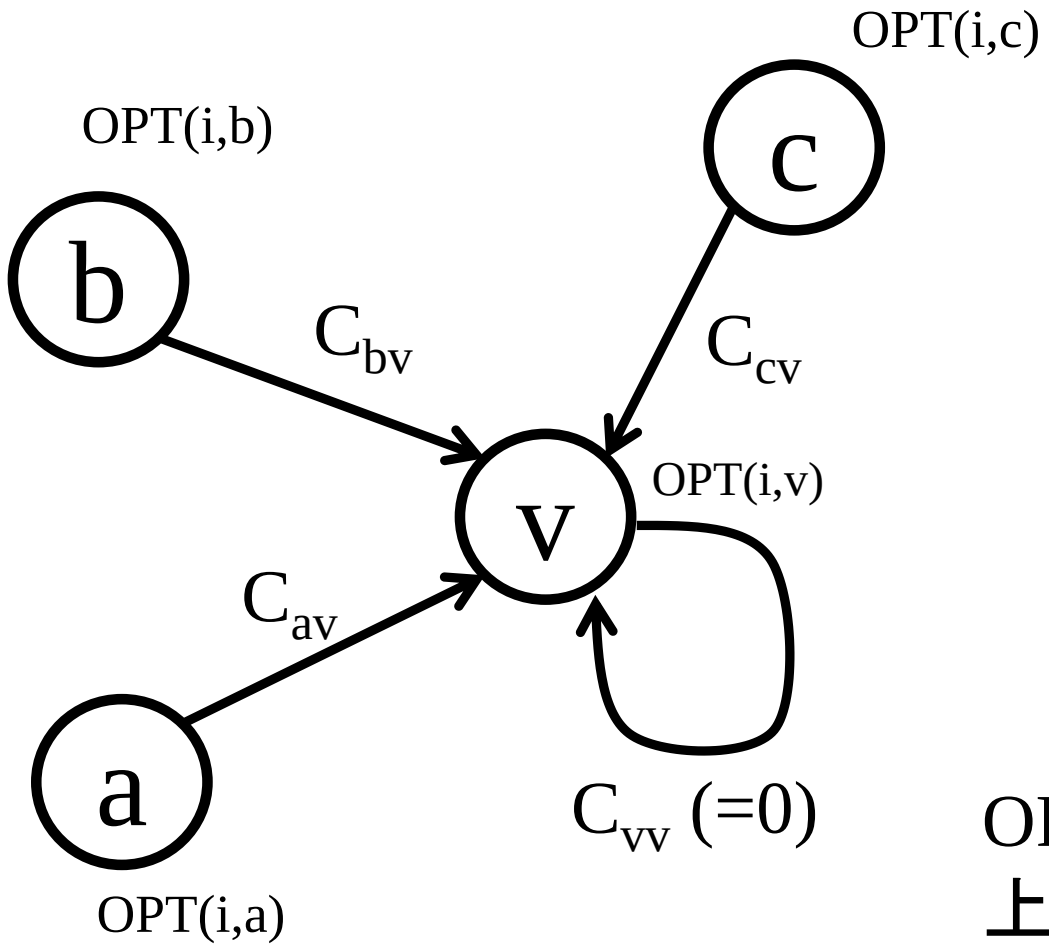
- i回目から、i+1回目への計算(漸化式)を考える
  - ここで、iは、拡散量(sから伝搬した辺の数)に相当する
- i回目(i個の辺の伝搬を考えたとき)における、頂点sから頂点vまでの最短距離をOPT(i,v)とする。このとき、以下が言える。

$$OPT(i+1, v) = \min_{u \in nbr(v)} \{OPT(i,u)+C_{uv}\}$$

(*) ここでnbr(v)は、vの近隣頂点(自分自身も含む)の集合とする
またC$_{uv}$は辺(u,v)のコストとし、便宜上C$_{vv}$=0 とする。

- 初期条件 (i=0のとき)
  - OPT(0,s)=0, OPT(0,v)=∞ (v∈V, v≠s)

# $OPT(i+1, v) = \min_{u \in nbr(v)} \{OPT(i,u)+C_{uv}\}$ の意味

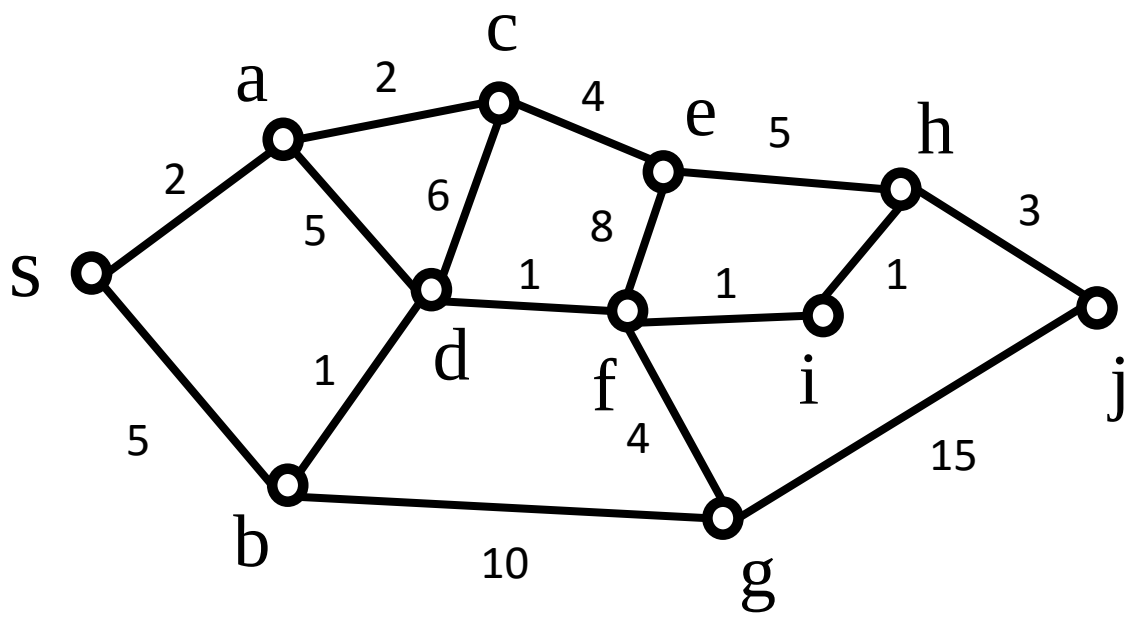



$OPT(i,v)$
$OPT(i,a)+C_{av}$
$OPT(i,b)+C_{bv}$
$OPT(i,c)+C_{cv}$

OPT(i+1,v)は、
上記で最小のものとする

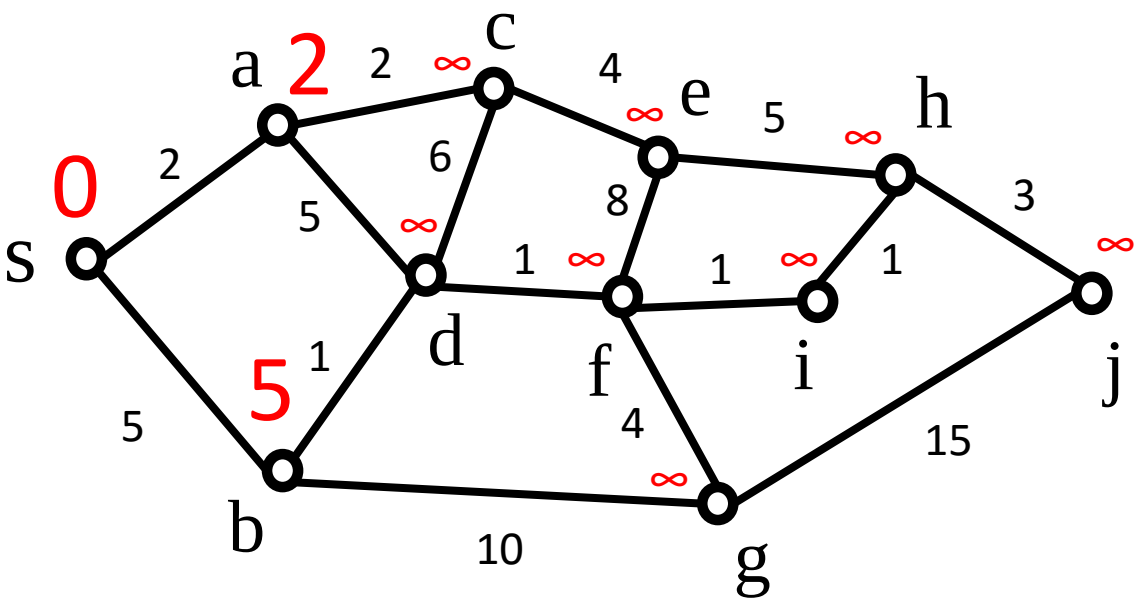
# 練習

- 以下のグラフに対し、ベルマンフォード法により、頂点sから各頂点までの最短距離を求めよ。

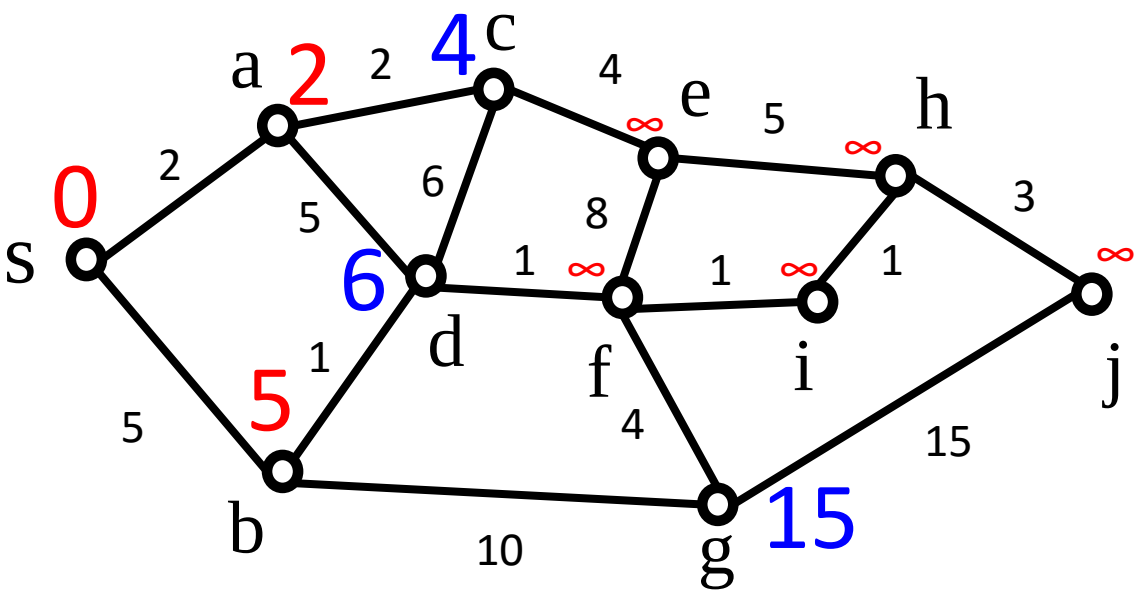



(*) ダイクストラ法との違いも考えてみよ

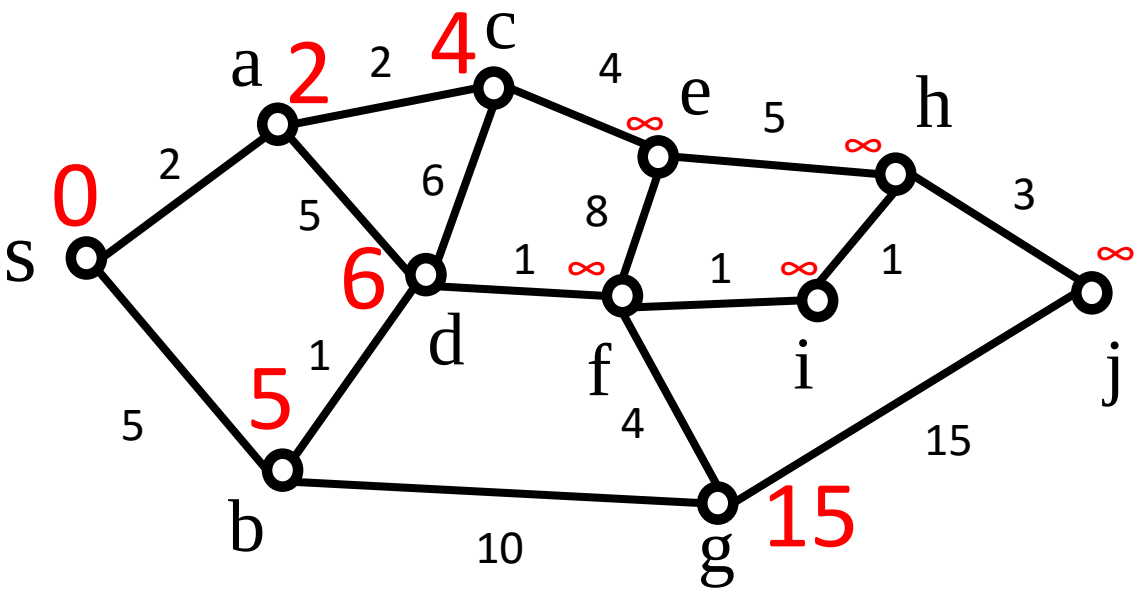
1回目


s
a
b
c
d
e
f
g
h
i
j
0
2
5
∞
∞
∞
∞
∞
∞
∞
2
5
6
1
8
1
1
3
5
1
4
10
15

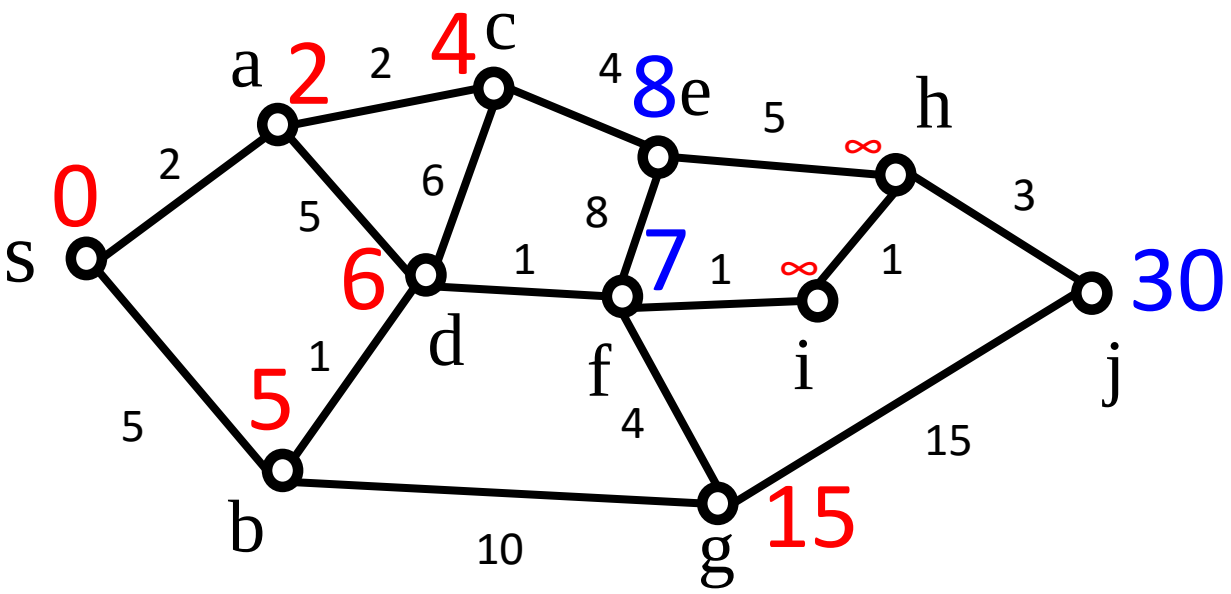
2回目


s
a
b
c
d
e
f
g
h
i
j
0
2
5
4
6
∞
∞
∞
∞
∞
∞
2
5
6
1
8
1
1
3
5
1
4
10
15

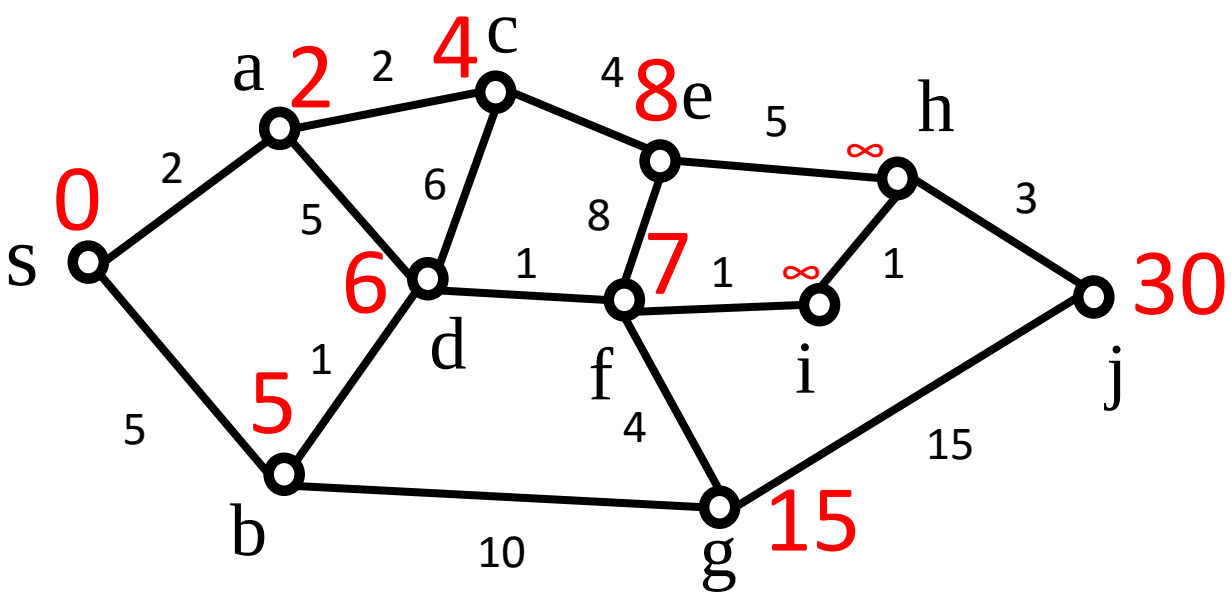
## 2回目


s
a
b
c
d
e
f
g
h
i
j
0
2
4
6
5
15
∞
∞
∞
∞
2
2
4
5
6
8
1
1
1
3
∞
5
1
4
15
10

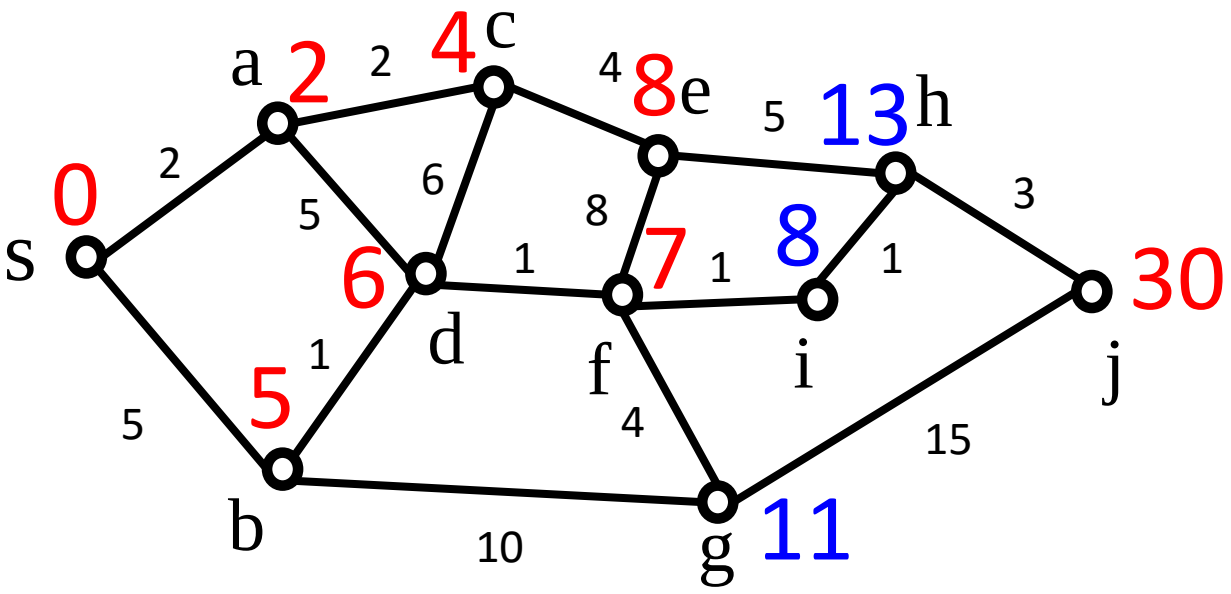
## 3回目


s
a
b
c
d
e
f
g
h
i
j
0
2
4
6
5
8
7
15
30
∞
2
2
4
5
6
8
1
1
1
3
5
1
4
15
10

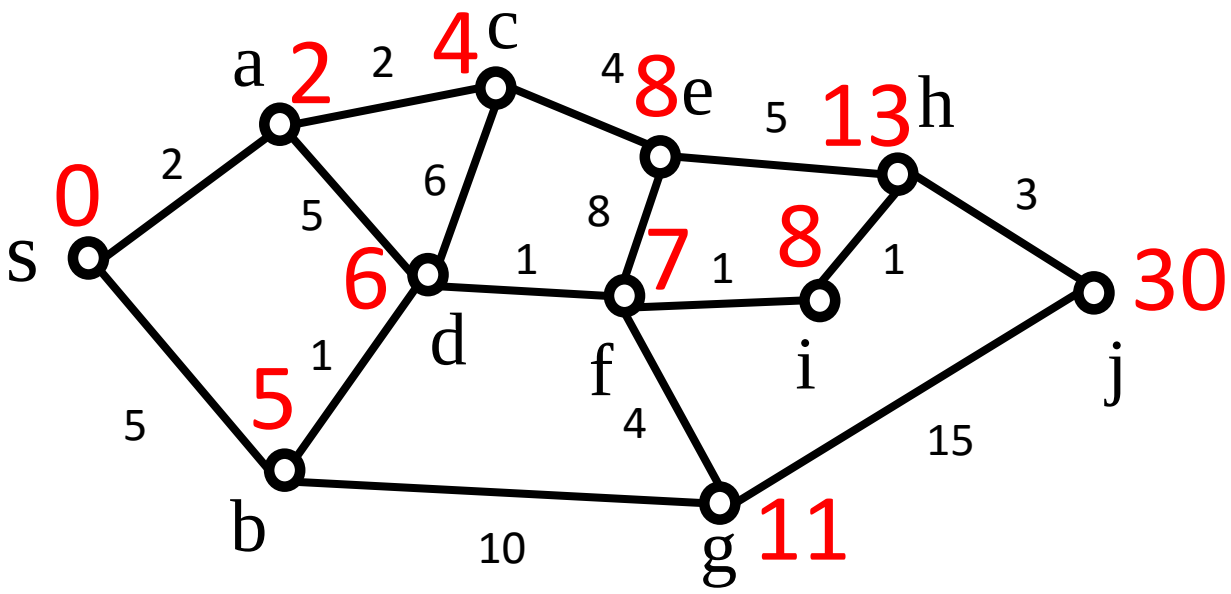
## 3回目


s
0
a
2
c
4
e
8
h
∞
d
6
f
7
i
∞
j
30
b
5
g
15
2
2
4
5
6
8
3
1
1
1
1
5
4
15
10

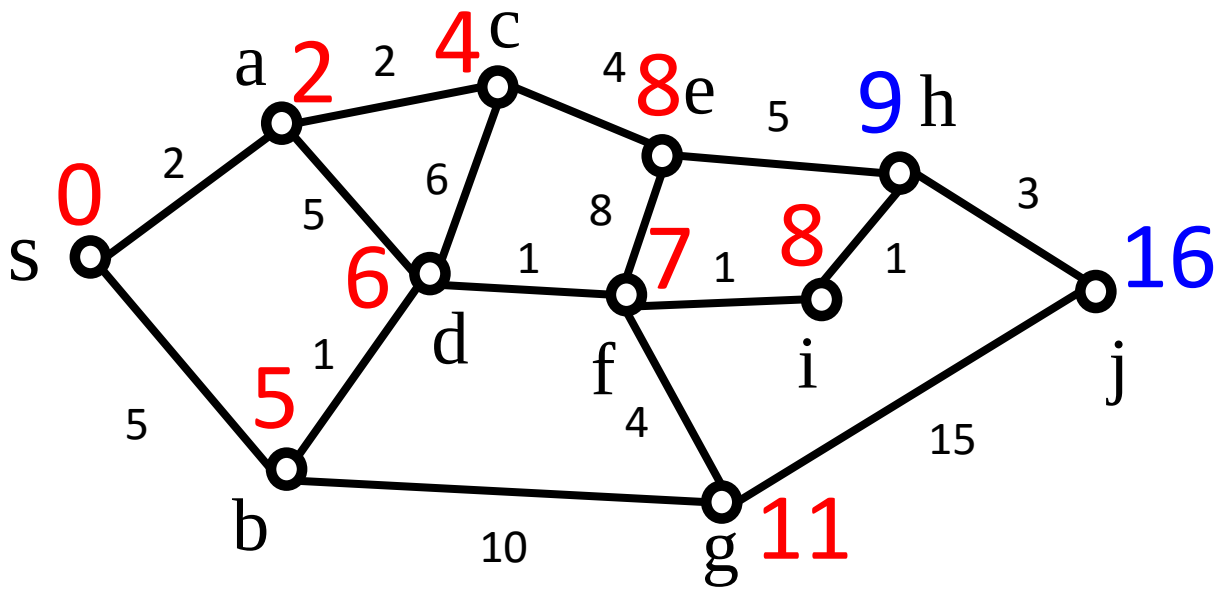
## 4回目


s
0
a
2
c
4
e
8
h
13
d
6
f
7
i
8
j
30
b
5
g
11
2
2
4
5
6
8
3
1
1
1
1
5
4
15
10

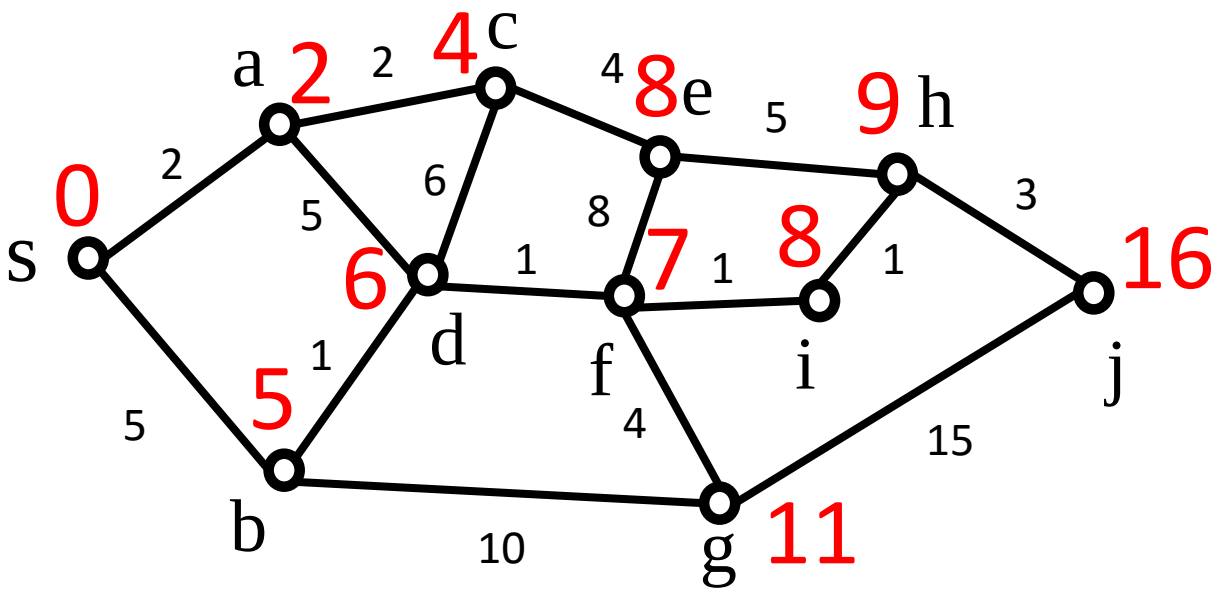
4回目


s
a
b
c
d
e
f
g
h
i
j
0
2
4
6
8
5
13
7
8
30
11
2
2
4
5
6
1
8
1
1
3
5
1
5
4
15
10

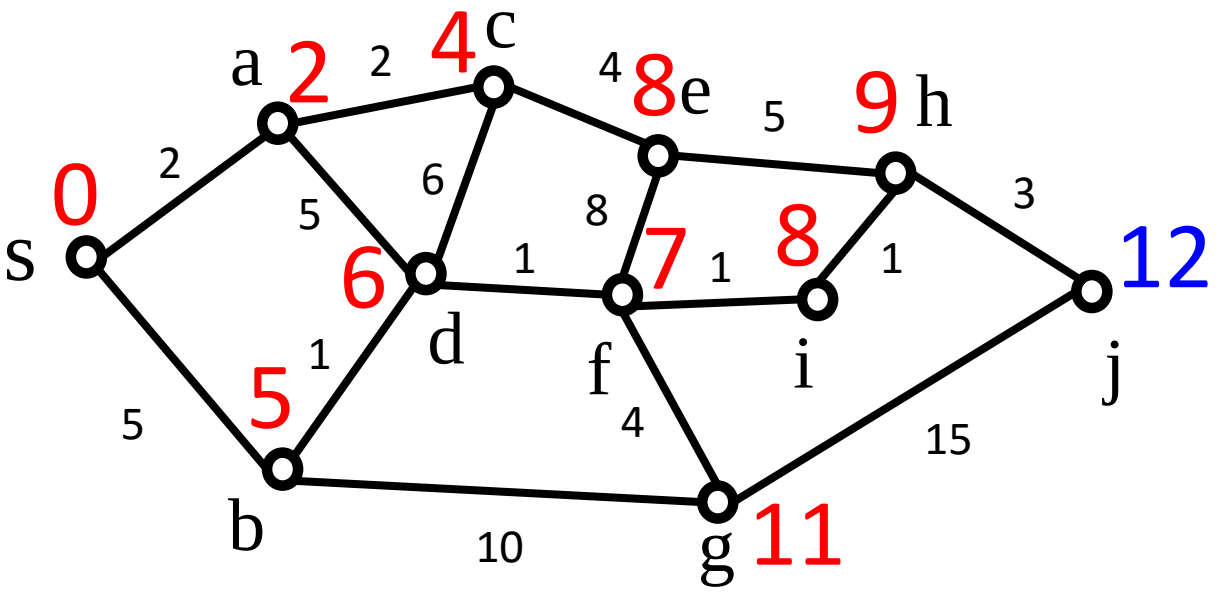
5回目


s
a
b
c
d
e
f
g
h
i
j
0
2
4
6
8
5
9
7
8
16
11
2
2
4
5
6
1
8
1
1
3
5
1
5
4
15
10

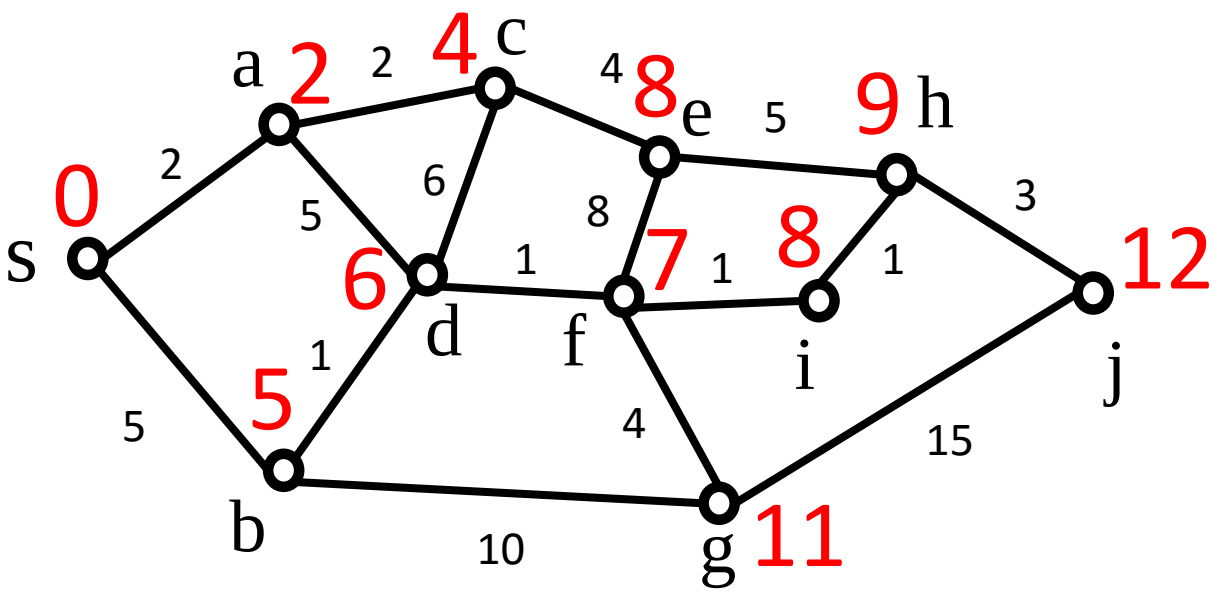
5回目


s
a
b
c
d
e
f
g
h
i
j
0
2
6
5
4
8
9
7
8
16
11
2
2
4
5
6
1
8
1
3
1
5
1
5
4
15
10

6回目


s
a
b
c
d
e
f
g
h
i
j
0
2
6
5
4
8
9
7
8
12
11
2
2
4
5
6
1
8
1
3
1
5
1
5
4
15
10

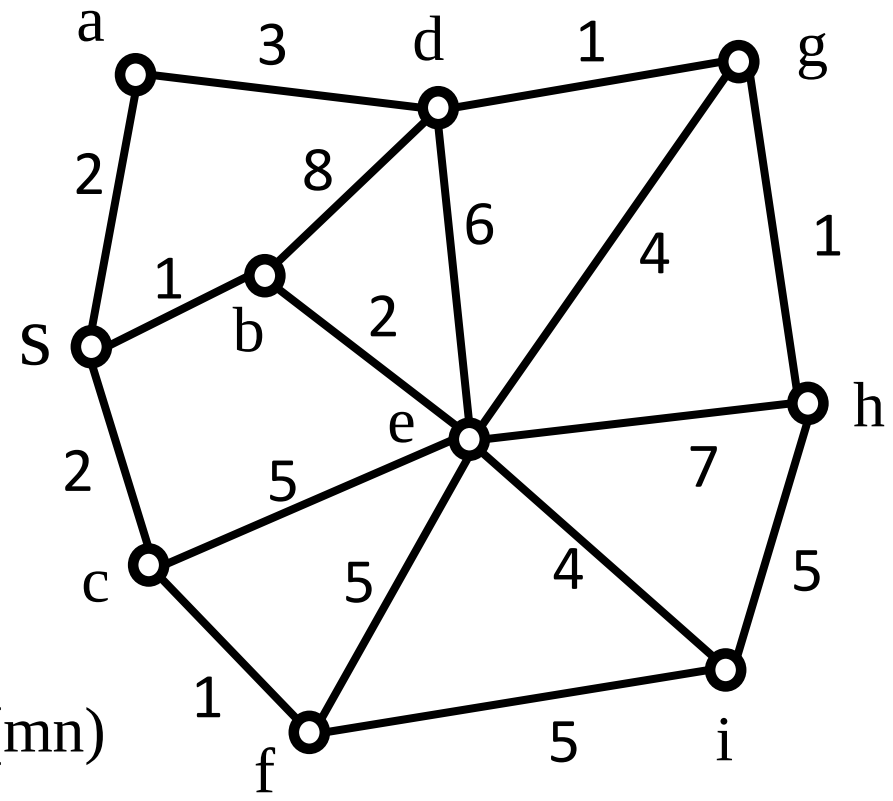
# 6回目以降




| 頂点 | 最短距離 |
|---|---|
| a | 2 |
| b | 5 |
| c | 4 |
| d | 6 |
| e | 8 |
| f | 7 |
| g | 11 |
| h | 9 |
| i | 8 |
| j | 12 |

# ベルマンフォードのアルゴリズム

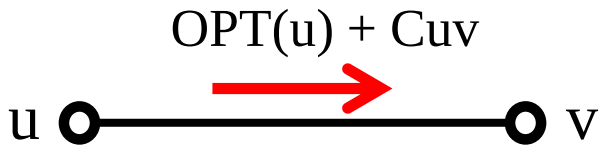
```
M[s]=0
M[v]=∞, prev[v]=null (v∈V, v≠s)
For i=1, … , n-1       //  n=count(V)
  Foreach e=(u, v) in E
    If M[v] > M[u]+Ce
      M[v]=M[u]+Ce
      prev[v]=u
    EndIf
  EndFor
EndFor
```




時間計算量: O(mn)

# ベルマンフォード法の分散化

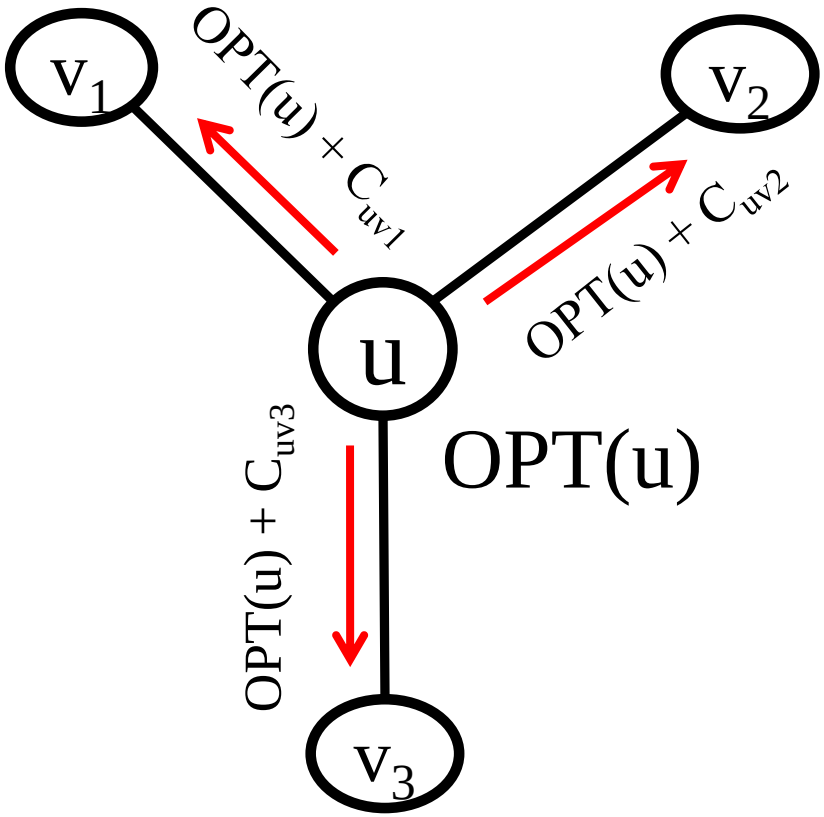
- グラフ G(V,E)において、各頂点は、辺で接続するもう片方の頂点と通信を行うものとする。
- このとき、



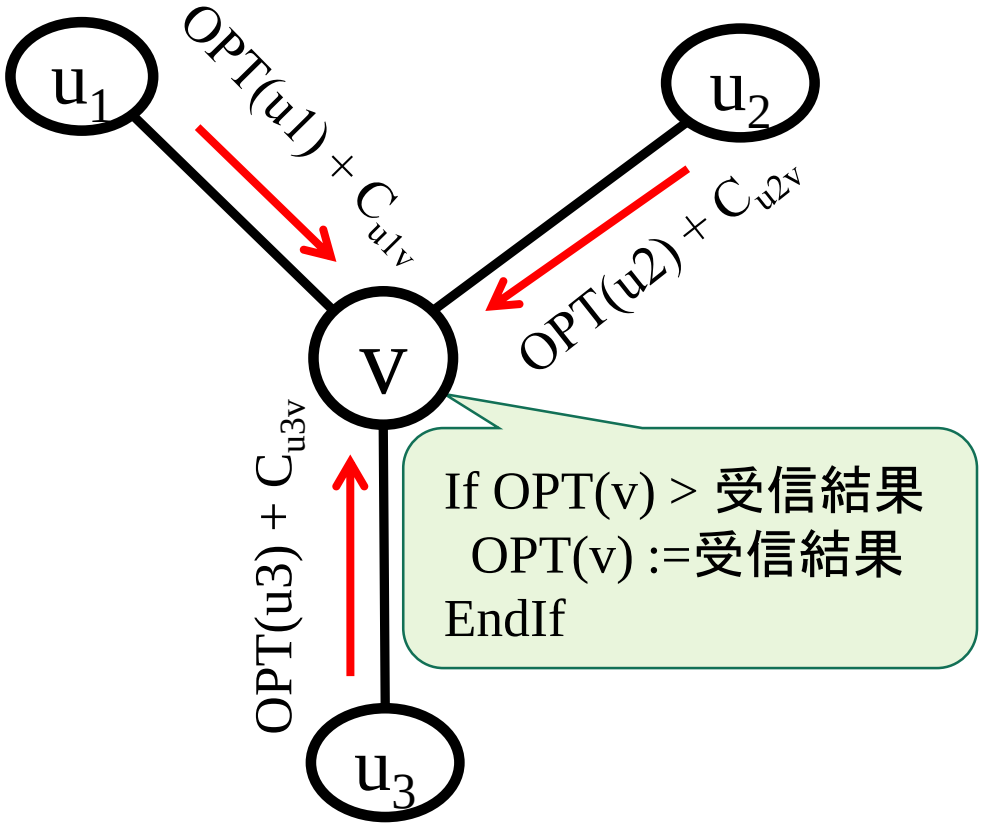

  - 各頂点uから、頂点v (∈nbr(u))に向けて、OPT(u)+Cuvを発信する
  - 受信側の頂点vでは、受信した値の中(他者からのものも含む)で、最小のものをOPT(v)として選ぶ
  - 開始点sのOPT(s)は 0 に固定する

という処理を行うと、ベルマンフォード法を分散的に実装できる

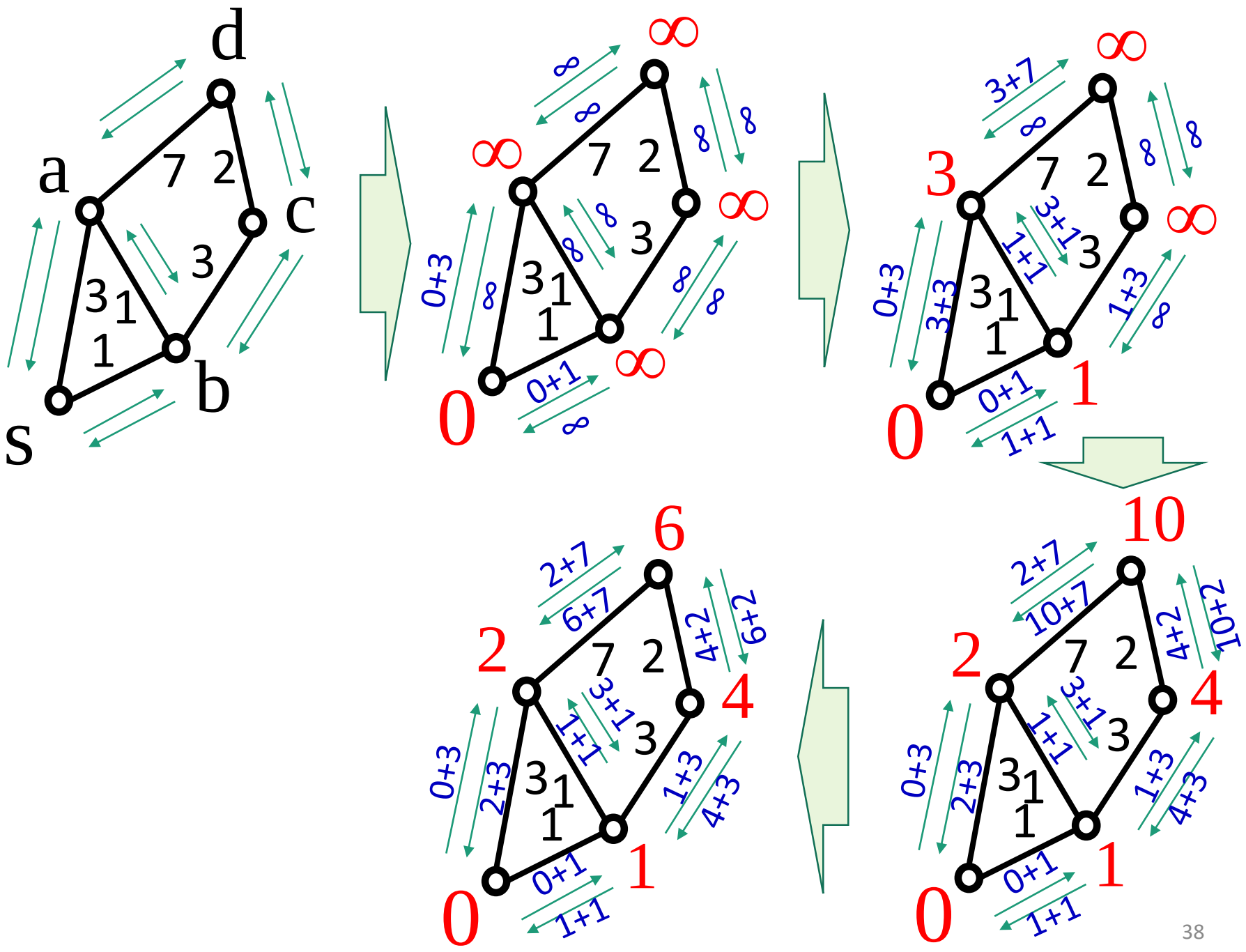
# ベルマンフォード法の分散化：各頂点の動作




頂点uからの送信
(定期的に行う)




頂点vでの受信と処理


d
a
c
s
b
7
2
3
3
1
1
∞
0
6
10
4
0+3
2+3
3+1
1+3
4+3
0+1
1+1
2+7
6+7
10+7
4+2
6+2
10+2
3+7
3+3

# 今後の予定

6月10日 : 休講 (レポートで代替)

6月17日: 最小全域木

6月24日: 最大流問題