プログラミング応用 第6回

河瀬 康志

2017年7月24日

授業スケジュール

	日程	内容
第 1 回	6/12	ガイダンス・復習
第 2 回	6/19	文字列操作（文字列整形，パターンマッチ，正規表現） 平面幾何（線分の交差判定，点と直線の距離，凸包）
—	6/26	出張のためお休み
第 3 回	7/3	乱数（一様分布，正規分布への変換，乱数生成） 統計（データ処理，フィッティング）
第 4 回	7/10	計算量（オーダー表記） スタックとキュー（幅優先探索，深さ優先探索）
第 5 回	7/17	ソートアルゴリズム バックトラック（N クイーン問題，数独）
第 6 回	7/24	動的計画法（ナップサック問題） 最短経路探索（Warshall-Floyd, Bellman-Ford, Dijkstra）
期末試験	7/31	—

期末試験について

- 7/31(月) 13:20 から
- 試験時間は 80 分
- 持ち込み可
  - 授業資料はパソコンにダウンロードしておいてください.
  - 宿題の解答を手に入れておくことをおすすめします.
    (おまけ問題は試験には出しません)
  - 本などの持ち込みも OK
  - 人の持ち込みは不可
- インターネットの使用は不可
  (携帯，スマホ禁止．パソコンの wifi はオフ)

アウトライン

分割統治法と動的計画法

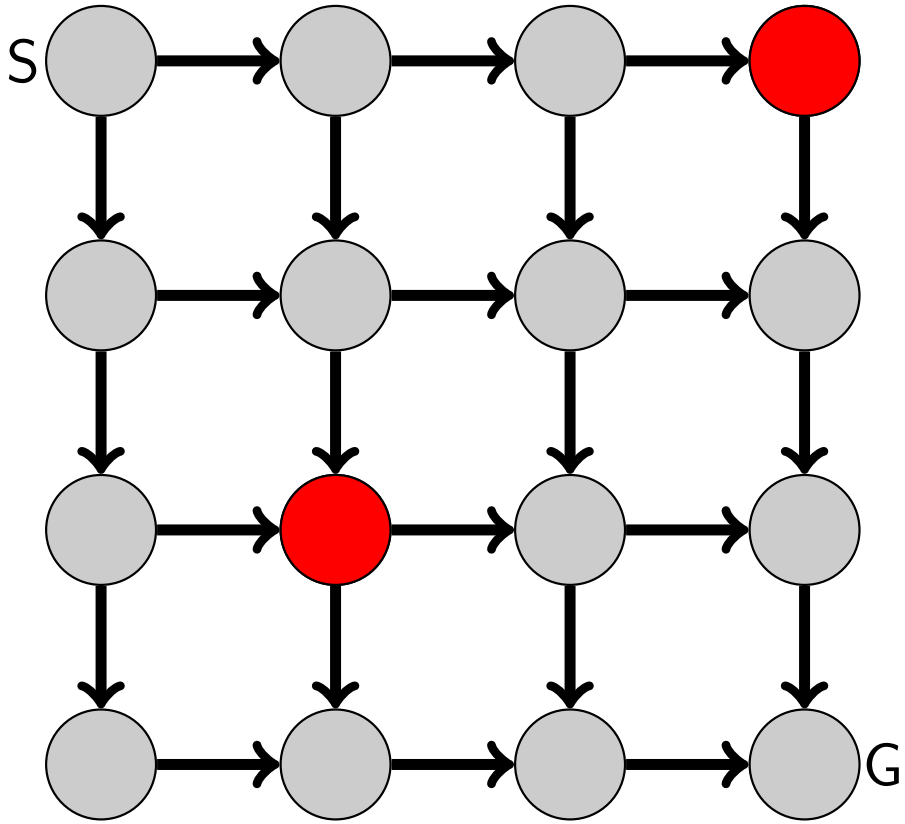
分割統治法 (Divide-and-Conquer)

- 分割：問題を小さな問題 (部分問題) に分割しそれぞれを解く
- 統治：部分問題の解を合わせることで元問題の解を求める
- 分割と統治は再帰的に行う

動的計画法 (Dynamic Programming, DP)

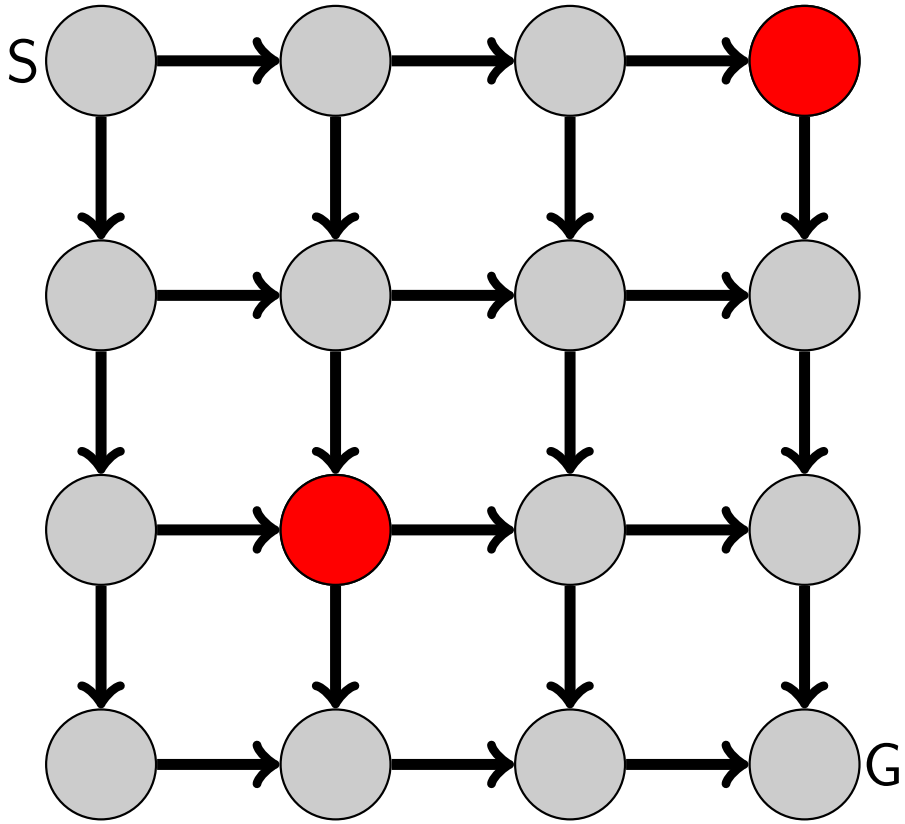
- 分割統治法では同じ問題を部分問題として何度も解くことがある
- どの問題を解いたか覚えておくことで無駄をなくす
- 部分問題の全体を表にし，ボトムアップに表を埋めていく

例：道の数 (1/3)



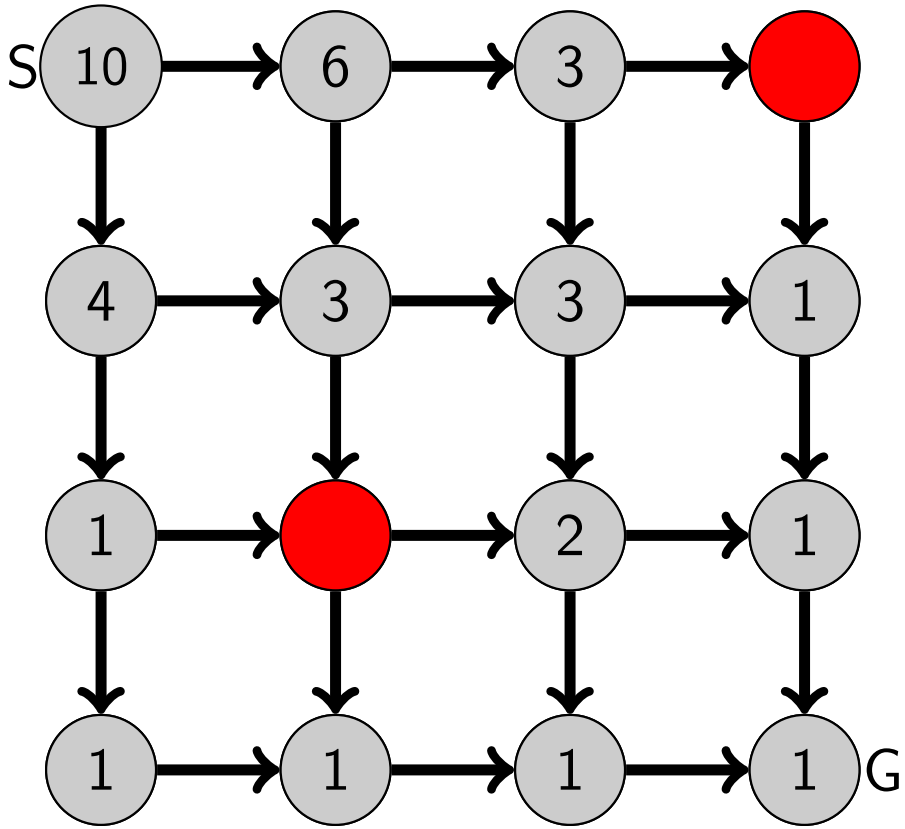
- 点 S から点 G まで後戻りせず移動する道の数何通りか？
- ただし，赤い点は通れないとする

例：道の数 (2/3)



- 右に進んだ場合と上に進んだ場合に分けて考える（分割）
  部分問題の解の和が元問題の道の数である（統治）
- どの道から来たかに依らず，各点から点 G までの道の数は一定
  ⇒ 動的計画法

例：道の数 (3/3)



- 右に進んだ場合と上に進んだ場合に分けて考える（分割）
  部分問題の解の和が元問題の道の数である（統治）
- どの道から来たかに依らず，各点から点 G までの道の数は一定
  $\Rightarrow$ 動的計画法 ($O(h \times w)$)

道の数 — Python では

- 結果の分かっている場所から逆にたどる
- 今回はゴールからゴールまでの道は 1 通りであることから始める

```
def solve(network):
    h = len(network)
    w = len(network[0])
    table = [[0]*h for i in range(w)]
    table[h-1][w-1] = 1
    for i in reversed(range(h)):
        for j in reversed(range(w)):
            if network[i][j]:
                if i+1<h: table[i][j]+=table[i+1][j]
                if j+1<w: table[i][j]+=table[i][j+1]
    return table[0][0]

network = [[1,1,1,0],
           [1,1,1,1],
           [1,0,1,1],
           [1,1,1,1]]

print(solve(network))
```

参考：メモ化再帰

- 再帰関数中で，解いた問題の答えをメモする方法
- 部分問題を解く順番をあまり考えなくてよい

```
def solve2(network):
    memo = {}
    memo[(len(network)-1,len(network[0])-1)]=1
    return _solve(network,0,0,memo)
def _solve(network,i,j,memo):
    if (i,j) in memo: return memo[(i,j)]
    if not network[i][j]: return 0
    res = 0
    if i+1<len(network): res+=_solve(network,i+1,j,memo)
    if j+1<len(network[0]): res+=_solve(network,i,j+1,memo)
    memo[(i,j)]=res
    return res

network = [[1,1,1,0],
           [1,1,1,1],
           [1,0,1,1],
           [1,1,1,1]]

print(solve2(network))
```

部分和問題

部分和問題

与えられた n 個の整数 $a_0, \ldots, a_{n-1}$ からいくつかを選び，
和をちょうど l にすることはできるか？

例：

- $a = [1, 3, 7, 8, 11]$, $l = 20$
  -
- $a = [2, 4, 8, 10, 12]$, $l = 19$
  -

部分和問題

部分和問題

与えられた n 個の整数 $a_0, \ldots, a_{n-1}$ からいくつかを選び，
和をちょうど l にすることはできるか？

例：

- $a = [1, 3, 7, 8, 11]$, $l = 20$
  - できる $[1, 8, 11]$
- $a = [2, 4, 8, 10, 12]$, $l = 19$
  -

部分和問題

部分和問題

与えられた n 個の整数 $a_0, \ldots, a_{n-1}$ からいくつかを選び，
和をちょうど l にすることはできるか？

例：

- $a = [1, 3, 7, 8, 11]$, $l = 20$
  - できる $[1, 8, 11]$
- $a = [2, 4, 8, 10, 12]$, $l = 19$
  - できない (偶数の和は偶数)

部分和問題 — ブルートフォース

方針: 各整数を取るか取らないか全部考える

- 組合せは 2^n 通りで，条件を満たすかのチェックに $O(n)$ $\Rightarrow O(2^n \cdot n)$

```
import itertools
def bruteforce(numbers,target):
    for i in range(len(numbers)):
        for l in itertools.combinations(numbers,i):
            if sum(l)==target:
                return list(l)
```

部分和問題 — バックトラック

方針: 各整数を取るか取らないかで場合分けし，
取り過ぎたり取らなさすぎたら戻る

- 計算量の評価は難しく，どの整数について場合分けするかにも依存

```
def backtrack(numbers,target):
    return _backtrack(numbers,target,sum(numbers))

def _backtrack(numbers,target,total):
    if target==0: return []
    if target==total: return numbers[:]
    if len(numbers)==0: return None
    if target<0 or total<target: return None

    n = numbers.pop()
    p =  _backtrack(numbers,target,total-n)
    q =  _backtrack(numbers,target-n,total-n)
    numbers.append(n)

    if p != None: return p
    if q != None: return q+[n]
```

部分和問題 — 動的計画法

方針: $a_0, \ldots, a_k$ $(k = 0, 1, \ldots, n-1)$ の組合せで h $(0 \leq h \leq l)$ を作れるかを部分問題とし，できるかどうかの表 (table[k][h]) を作る

- table[k][h]=(table[$k-1$][h] or table[$k-1$][$h-a[k]$]) が成立
- table[n][l] の値は $O(n \cdot l)$ で計算できる
- l は入力サイズではないので多項式時間ではない (擬多項式時間)
- 多項式時間では解けないと信じられている (NP 困難)

例: $a = [2, 4, 3]$, $l = 5$

$k \backslash h$	0	1	2	3	4	5
0 $(a_0 = 2)$	—	—	—	—	—	—
1 $(a_1 = 4)$	—	—	—	—	—	—
2 $(a_2 = 3)$	—	—	—	—	—	—

部分和問題 — 動的計画法

方針: $a_0, \ldots, a_k$ $(k = 0, 1, \ldots, n-1)$ の組合せで h $(0 \leq h \leq l)$ を作れるかを部分問題とし，できるかどうかの表 (table[k][h]) を作る

- table[k][h]=(table[$k-1$][h] or table[$k-1$][$h-a[k]$]) が成立
- table[n][l] の値は $O(n \cdot l)$ で計算できる
- l は入力サイズではないので多項式時間ではない (擬多項式時間)
- 多項式時間では解けないと信じられている (NP 困難)

例: $a = [2, 4, 3]$, $l = 5$

$k \backslash h$	0	1	2	3	4	5
0 ($a_0 = 2$)	T	F	T	F	F	F
1 ($a_1 = 4$)	—	—	—	—	—	—
2 ($a_2 = 3$)	—	—	—	—	—	—

部分和問題 — 動的計画法

方針: $a_0, \ldots, a_k$ $(k = 0, 1, \ldots, n-1)$ の組合せで h $(0 \leq h \leq l)$ を作れるかを部分問題とし，できるかどうかの表 (table[k][h]) を作る

- table[k][h]=(table[$k-1$][h] or table[$k-1$][$h-a[k]$]) が成立
- table[n][l] の値は $O(n \cdot l)$ で計算できる
- l は入力サイズではないので多項式時間ではない (擬多項式時間)
- 多項式時間では解けないと信じられている (NP 困難)

例: $a = [2, 4, 3]$, $l = 5$

$k \backslash h$	0	1	2	3	4	5
0 $(a_0 = 2)$	T	F	T	F	F	F
1 $(a_1 = 4)$	T	F	T	F	T	F
2 $(a_2 = 3)$	—	—	—	—	—	—

部分和問題 — 動的計画法

方針: $a_0, \ldots, a_k$ $(k = 0, 1, \ldots, n-1)$ の組合せで h $(0 \leq h \leq l)$ を作れるかを部分問題とし，できるかどうかの表 (table[k][h]) を作る

- table[k][h]=(table[$k-1$][h] or table[$k-1$][$h-a[k]$]) が成立
- table[n][l] の値は $O(n \cdot l)$ で計算できる
- l は入力サイズではないので多項式時間ではない (擬多項式時間)
- 多項式時間では解けないと信じられている (NP 困難)

例: $a = [2, 4, 3]$, $l = 5$

$k \backslash h$	0	1	2	3	4	5
0 $(a_0 = 2)$	T	F	T	F	F	F
1 $(a_1 = 4)$	T	F	T	F	T	F
2 $(a_2 = 3)$	T	F	T	T	T	T

部分和問題 — 動的計画法 in Python

```
def dp(numbers,target):
    table = [[False]*(target+1) for i in range(len(numbers))]
    table[0][0]=True
    if numbers[0]<=target: table[0][numbers[0]]=True
    for k in range(1,len(numbers)):
        for h in range(target+1):
            if table[k-1][h]: table[k][h]=True
            if h>=numbers[k] and table[k-1][h-numbers[k]]: table[k][h]=True
    if not table[len(numbers)-1][target]: return None

    # output
    result = []
    h = target
    for k in reversed(range(len(numbers))):
        if k==0:
            if h>0: result.append(numbers[0])
        elif not table[k-1][h]:
            result.append(numbers[k])
            h -= numbers[k]
    return result
```

ナップサック問題

ナップサック問題

重さが w_i, 価値が p_i の品物 $i = 0, \ldots, n-1$ からいくつか選んで，重さの合計が B 以下の中で価値の合計をなるべく大きくしたい．どのような商品集合を選べばいいか？

$$\max \sum_{i=0}^{n-1} p_i x_i \quad \text{s.t.} \quad \sum_{i=0}^{n-1} w_i x_i \leq B, \ x_i \in \{0, 1\}$$

ナップサック問題に対する動的計画法 1

価値をテーブルとする DP

- 以下の条件を満たす最大価値をテーブルとする
  - 品物は $\{0, \ldots, k\}$ の部分集合 $(k = 0, \ldots, n-1)$
  - 重さの合計は b 以下 $(b = 0, 1, \ldots, B)$
- $\mathrm{table}[k][b] = \max(\mathrm{table}[k-1][b], \mathrm{table}[k-1][b-w_k] + p_k)$ が成立
- 計算量は $O(n \cdot B)$

```
def weight_dp(items,B):
    n = len(items)
    table = [[0]*(B+1) for i in range(n)]
    for k in range(n):
        (w,p)=items[k]
        for b in range(B+1):
            if k==0:
                if b>=w: table[k][b]=p
            else:
                if b>=w: table[k][b]=max(table[k-1][b],table[k-1][b-w]+p)
                else: table[k][b]=table[k-1][b]
    return table[n-1][B]
```

ナップサック問題に対する動的計画法2

重さをテーブルとするDP

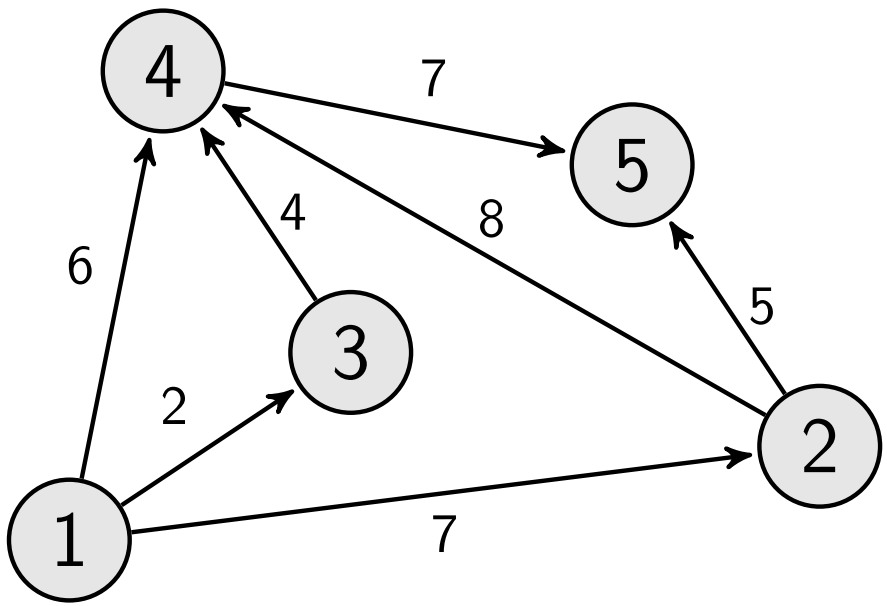
- 以下の条件を満たす最小重みをテーブルとする
  - 品物は $\{0, \ldots, k\}$ の部分集合 $(k = 0, \ldots, n-1)$
  - 価値の合計は q 以上 $(q = 0, 1, \ldots, \sum_{i=0}^{n-1} p_i)$
- $\mathrm{table}[k][q] = \min(\mathrm{table}[k-1][q], \mathrm{table}[k-1][q-p_k] + w_k)$ が成立
- 計算量は $O(n \cdot \sum_{i=0}^{n-1} p_i)$

```
def price_dp(items,B):
    n = len(items)
    P = sum([p for (w,p) in items])
    table = [[B+1]*(P+1) for i in range(n)]
    for k in range(len(items)):
        (w,p)=items[k]
        for q in range(P+1):
            if k==0:
                if q<=p: table[k][q]=w
            else:
                if q<=p: table[k][q]=min(table[k-1][q],w)
                else: table[k][q]=min(table[k-1][q],table[k-1][q-p]+w)
    return max([q for (q,b) in enumerate(table[n-1]) if b<=B])
```

アウトライン

グラフ

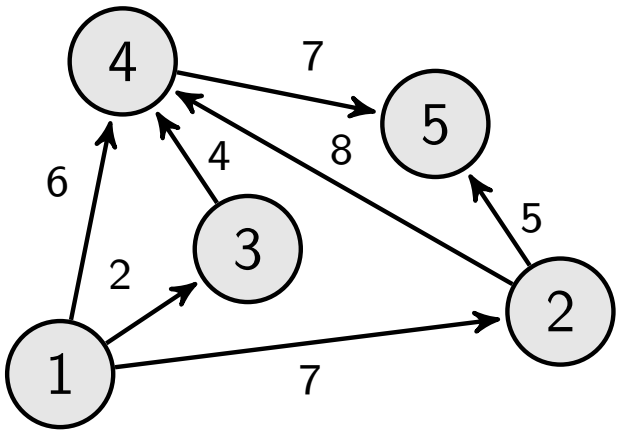
- 辺 (枝) に長さのあるグラフにおける最短経路を扱う
- 幅優先探索は使えない
  - 幅優先探索がつかえるのは，辺の長さがすべて等しいときだけ



グラフの表現

隣接リストとして次のように表現されているとする

```
adj = {1:[(2,7),(3,2),(4,6)],
       2:[(4,8),(5,5)],
       3:[(4,4)],
       4:[(5,7)],
       5:[]
```



Dijkstra 法

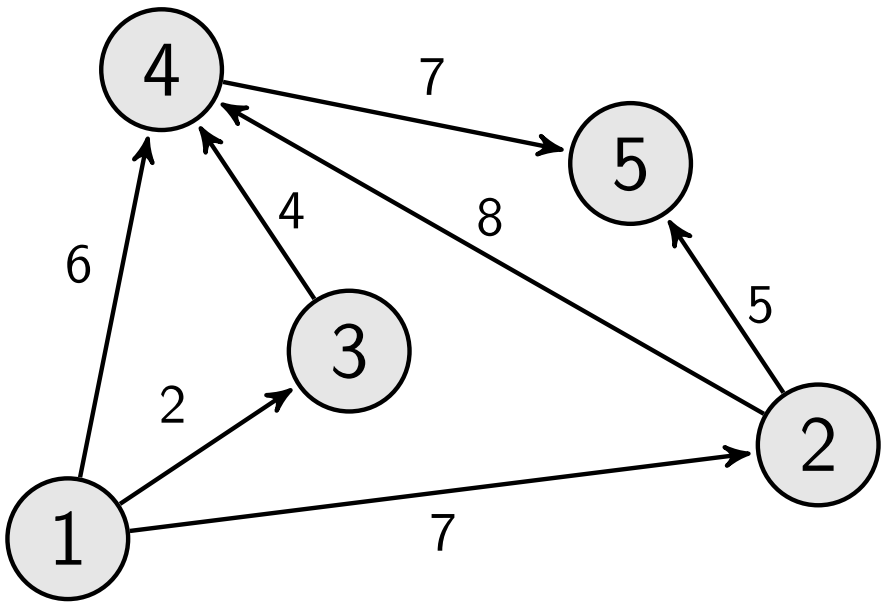
- 単一始点 s からの最短経路を求める方法
- すべての辺の長さは非負と仮定
- 方針：
  - 最短距離の近い頂点から順に確定させていく
  - 新たに確定される頂点は，確定している頂点から 1 ステップなので，下の暫定距離が最小の確定されていない頂点が次に確定される頂点

$$\min_{v \in \text{最小距離が確定した点}} (v \text{ までの距離} + (v, u) \text{ 間の長さ})$$

  - 最初は，s までの距離は 0，それ以外は ∞ とする
- 計算量は以下の通り
  - 単純な実装 $O(|V|^2)$
  - ヒープを使った実装 $O(|E| \log |E|)$

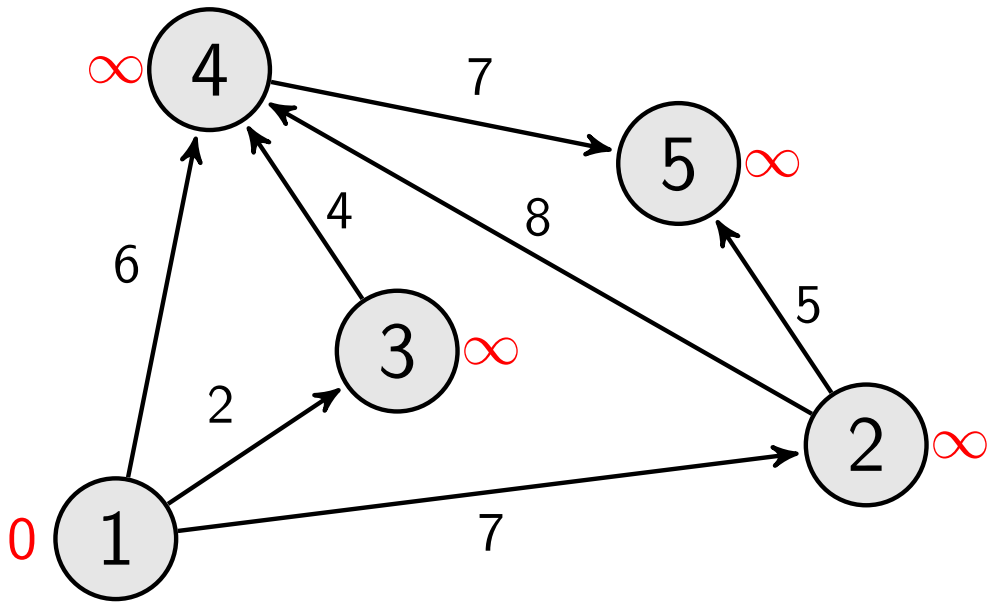
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理：



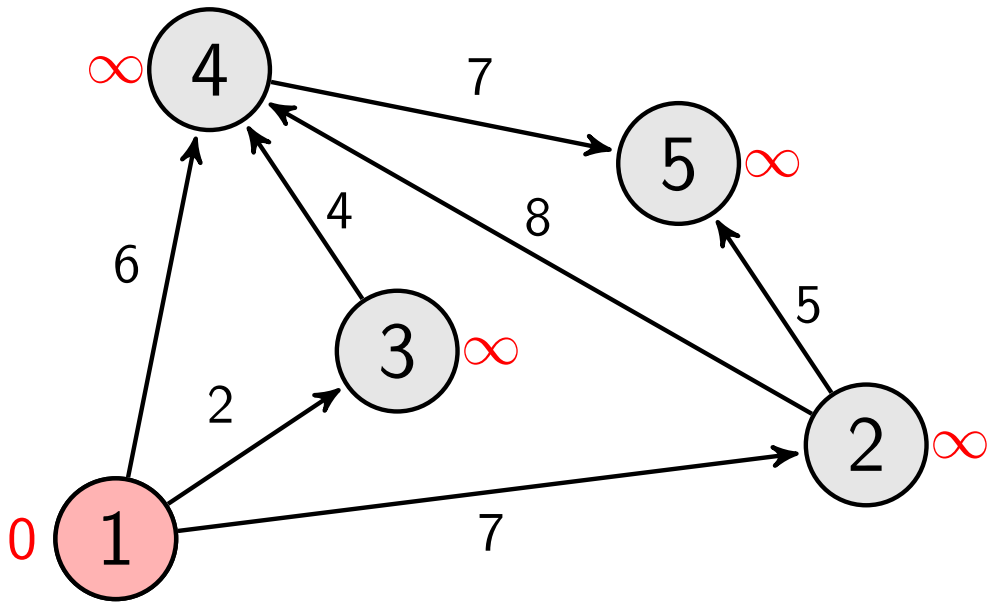
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 初期化



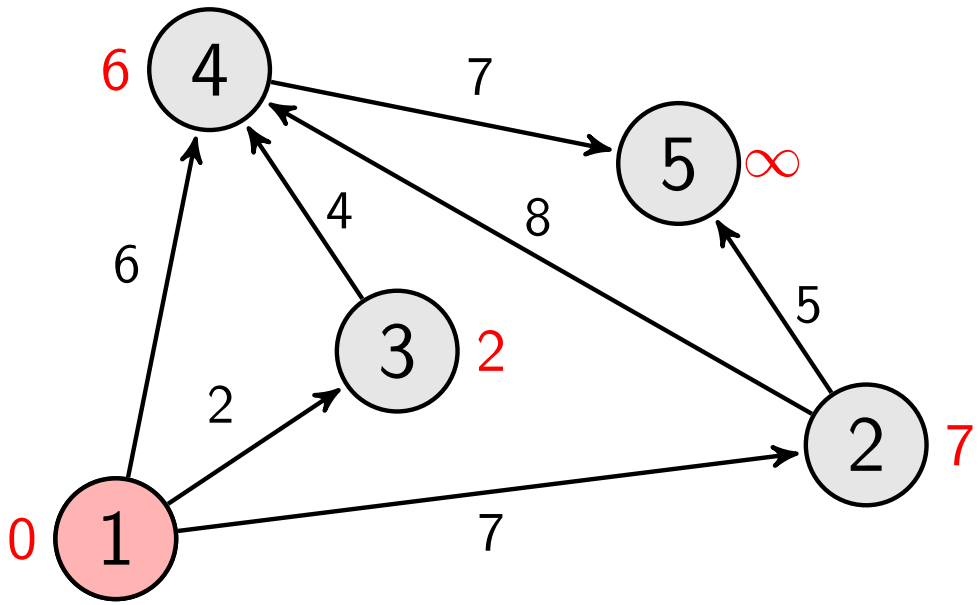
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 1 を確定



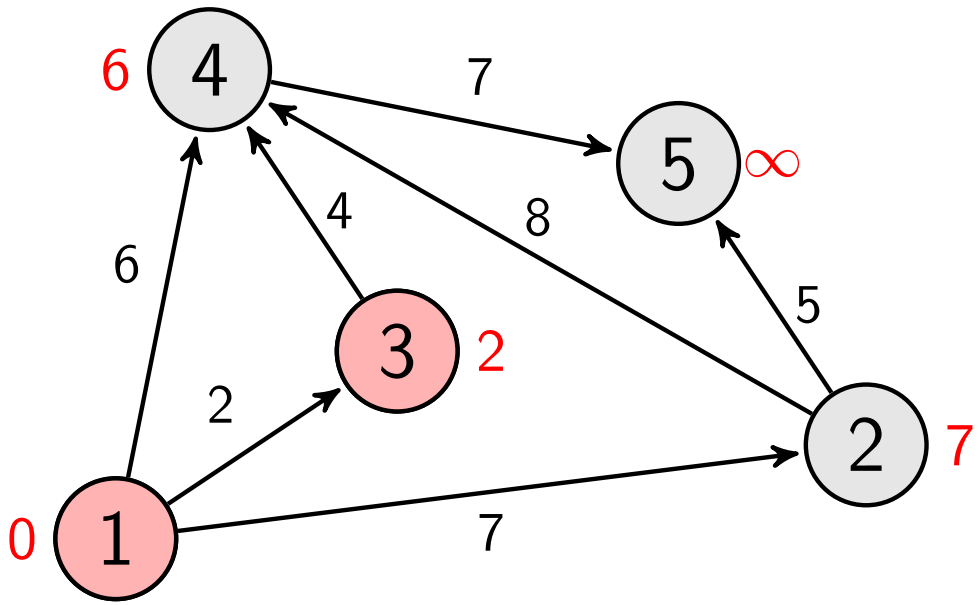
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 暫定距離を更新



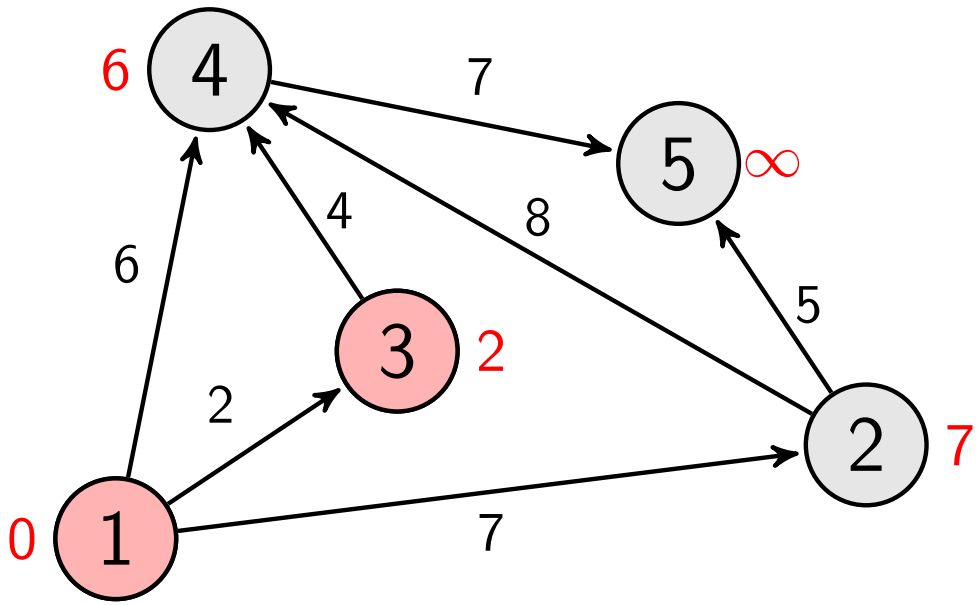
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 3 を確定



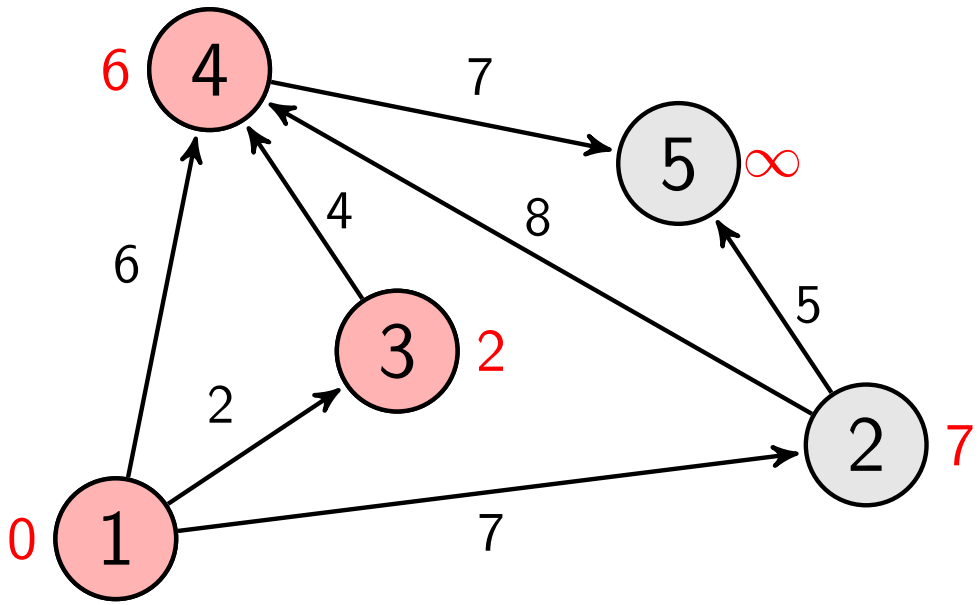
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 暫定距離を更新



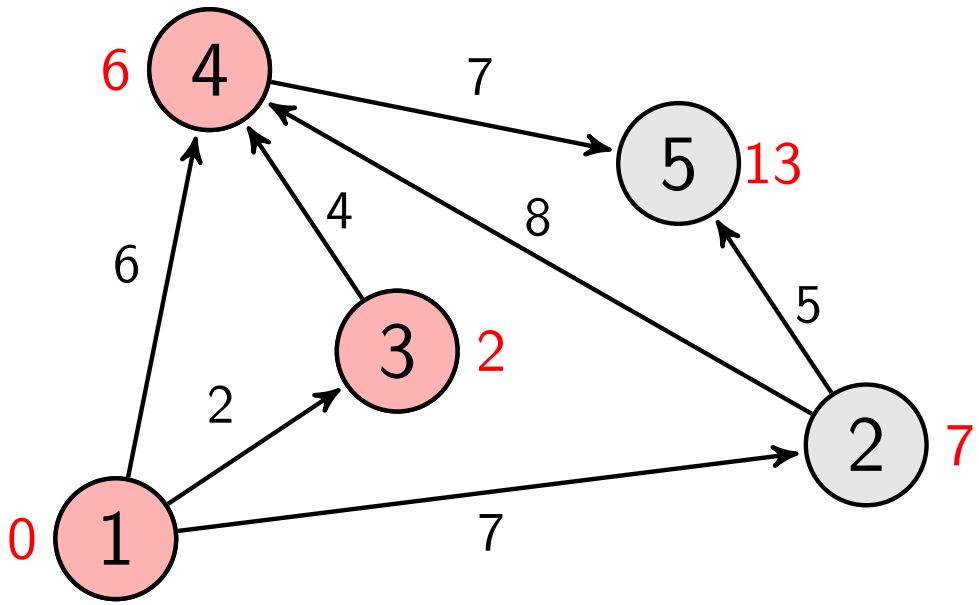
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 4 を確定



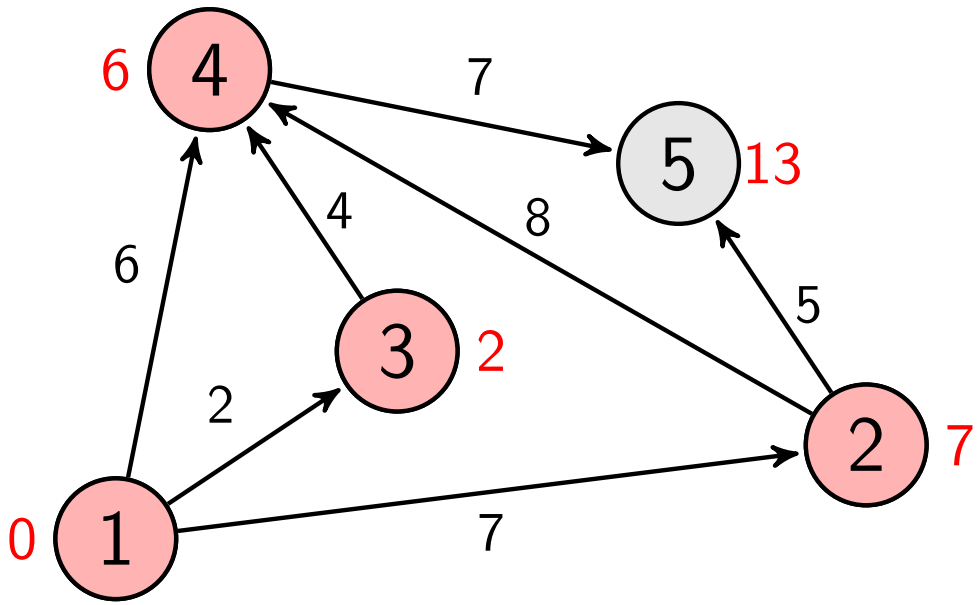
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 暫定距離を更新



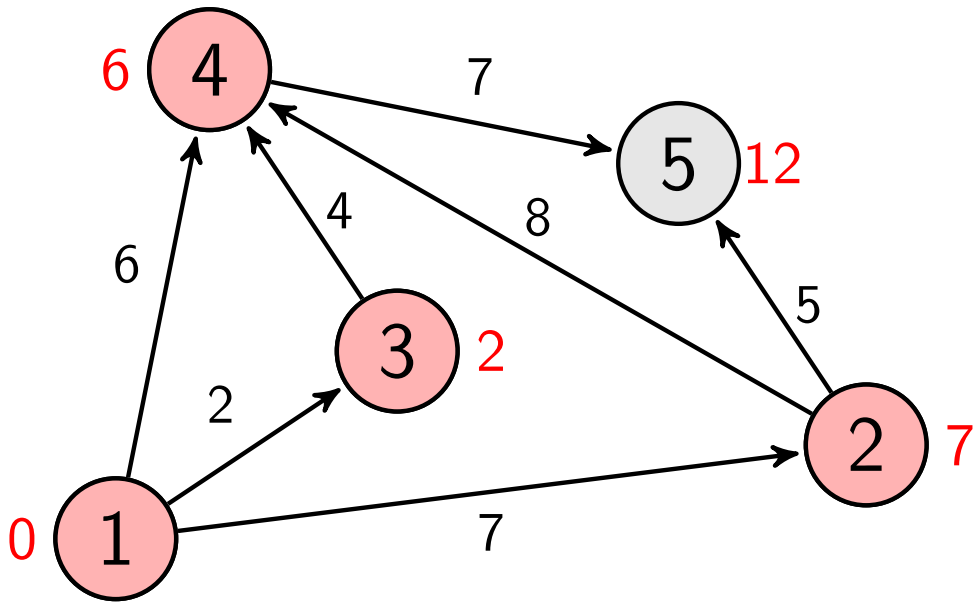
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 2 を確定



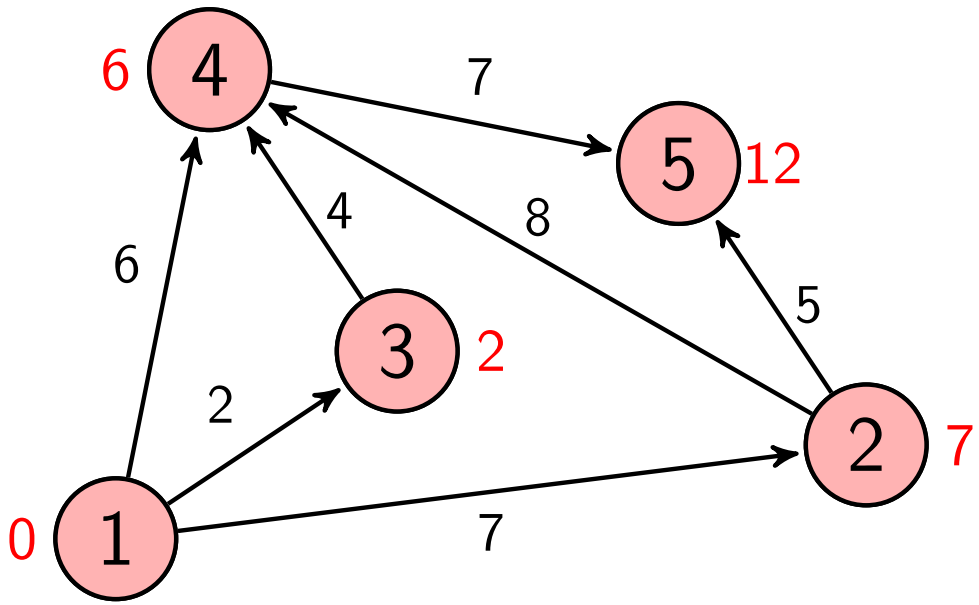
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 暫定距離を更新



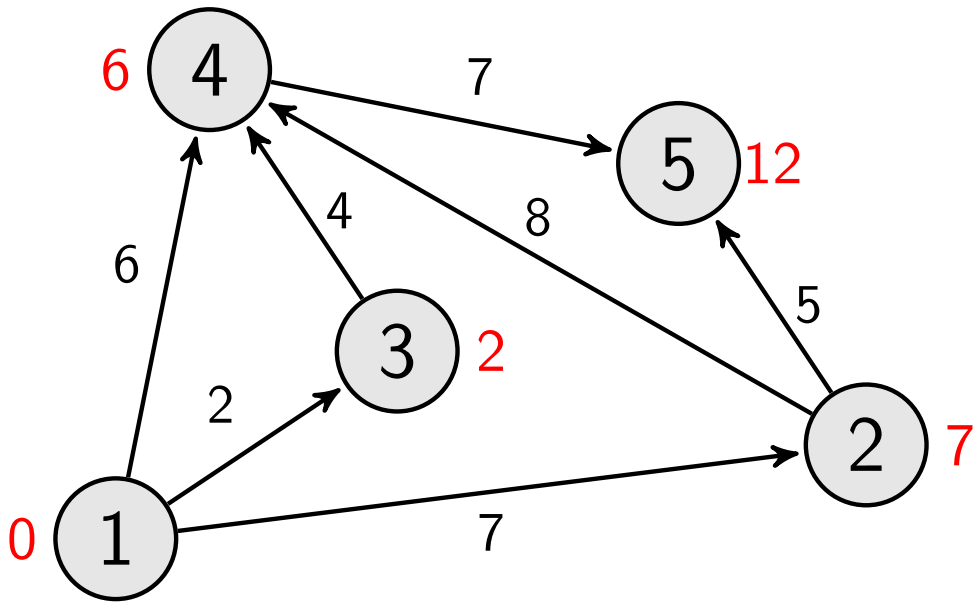
Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 5 を確定



Dijkstra 法の動き

- 始点 $s = 1$ とする
- 処理： 計算終わり



Dijkstra 法 — Python による実装

```
def dijkstra_simple(adj,s):
    vs = adj.keys()
    dist = {v:float('inf') for v in vs}
    dist[s] = 0 # s までの距離は 0
    fixed = set() # 最短距離が確定した点集合
    while True:
        # まだ確定しない中で暫定距離最小のものを探す
        min_dist = float('inf')
        for v in vs:
            if (v not in fixed) and min_dist>dist[v]:
                min_dist=dist[v]
                min_v = v
        if min_dist==float('inf'): break
        fixed.add(min_v)
        for (u,d) in adj[min_v]:
            dist[u] = min(dist[u],dist[min_v]+d) # 暫定距離を更新
    print(dist)
```

- float('inf') で無限大を表す浮動小数点数を得られる

ヒープ

動的な最小値の更新を効率よく行える

- 最小値の取得： $O(1)$
- 最小値の要素を削除： $O(\log n)$
- 要素を挿入： $O(\log n)$

Python では

- import heapq でヒープが使えるようになる
- リスト上の操作として定義されている

```
>>> import heapq
>>> h = [20,30,10]
>>> heapq.heapify(h) # リストをヒープに変換
>>> h
[10, 30, 20]
>>> heapq.heappush(h,5) # h に 5 を追加
>>> heapq.heappush(h,12) # h に 12 を追加
>>> h
[5, 10, 20, 30, 12]
>>> h[0] # 最小値
5
>>> heapq.heappop(h)
5
>>> h
[10, 12, 20, 30]
```

Dijkstra 法 — ヒープを使った実装

```
import heapq
def dijkstra_heap(adj,s):
    vs = adj.keys()
    dist = {}
    heap = [(0,s)]
    while len(heap)>0:
        (d,u) = heapq.heappop(heap)
        if u not in dist:
            dist[u] = d
            for (w,l) in adj[u]:
                heapq.heappush(heap,(d+l,w))
    print(dist)
```

- ヒープは辞書式順序で最小のものを取り出せる
- タプルの最初の要素を暫定距離にしてヒープに入れることで, 次に確定すべき点が分かる
- (辺の少ないグラフに対しては) 効率がよくなる

Bellman-Ford 法

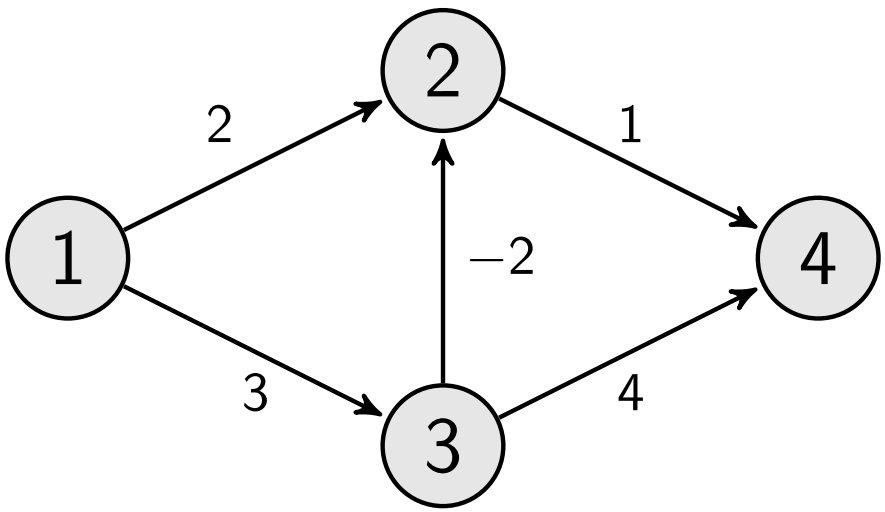
- 単一始点 s からの最短経路を求める方法
- 長さが負の枝があってもよい（負閉路は駄目）
- 計算量は $O(|V| \cdot |E|)$
- 方針:
  - 各点 v までの距離は次の等式を満たす

$$v \text{ までの距離} = \min_{(u,v) \in \text{枝}} (u \text{ までの距離} + (u, v) \text{ 間の長さ})$$

  - 等式に従い，各枝について更新
  - 負閉路がなければ，$|V|$ 回までに更新は止まり，最短距離がでる

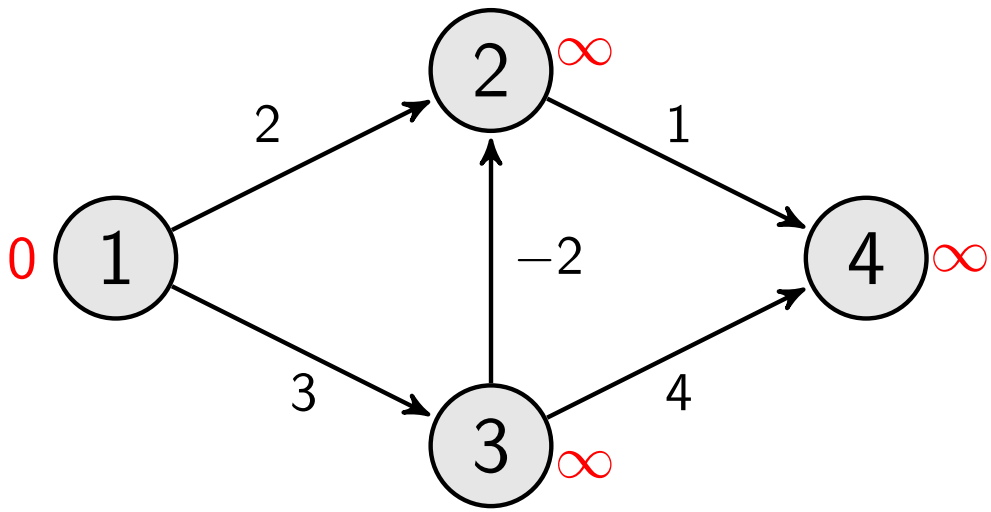
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理:



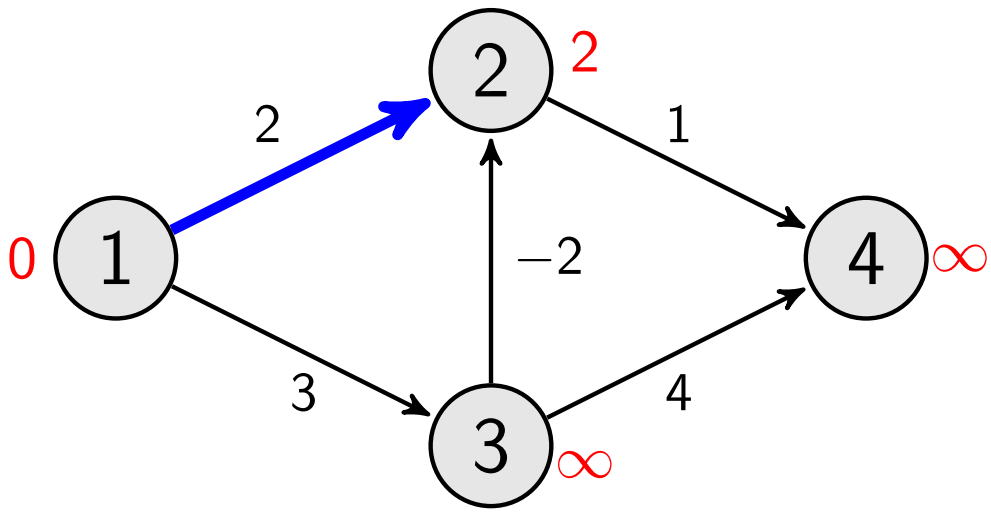
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 初期化



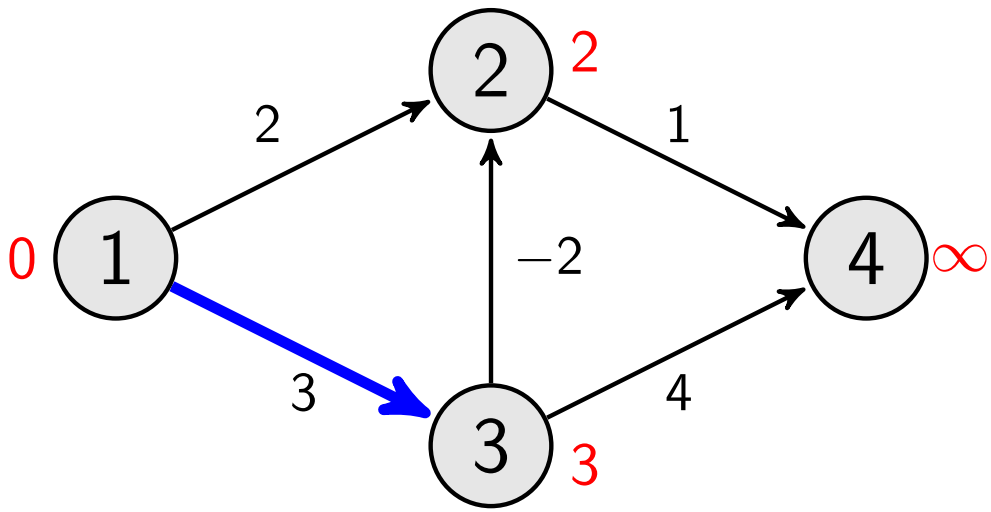
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(1, 2)$　(1 周目)



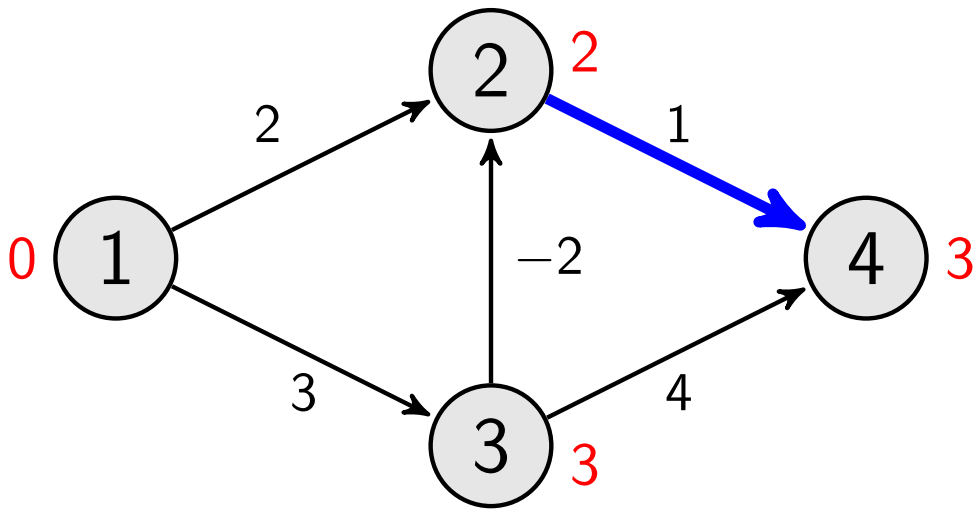
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(1, 3)$　(1 周目)



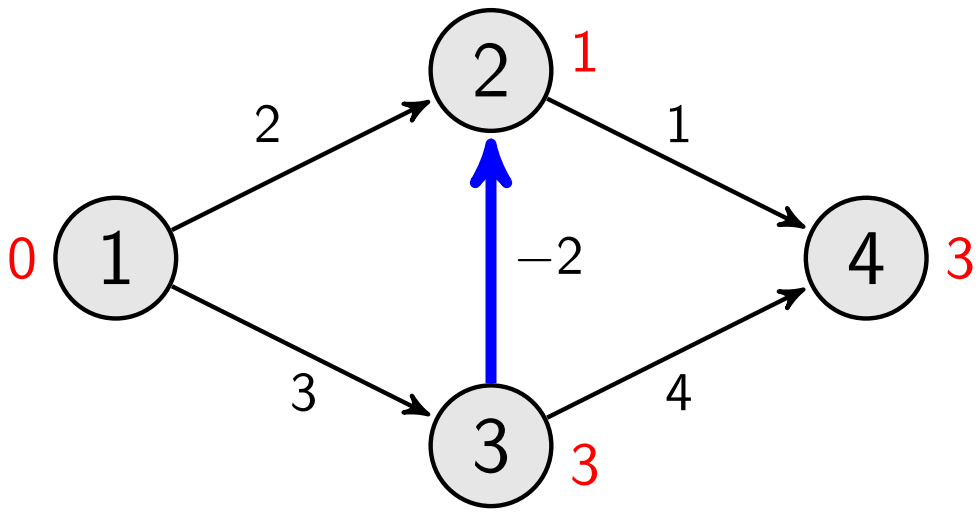
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(2, 4)$　(1 周目)



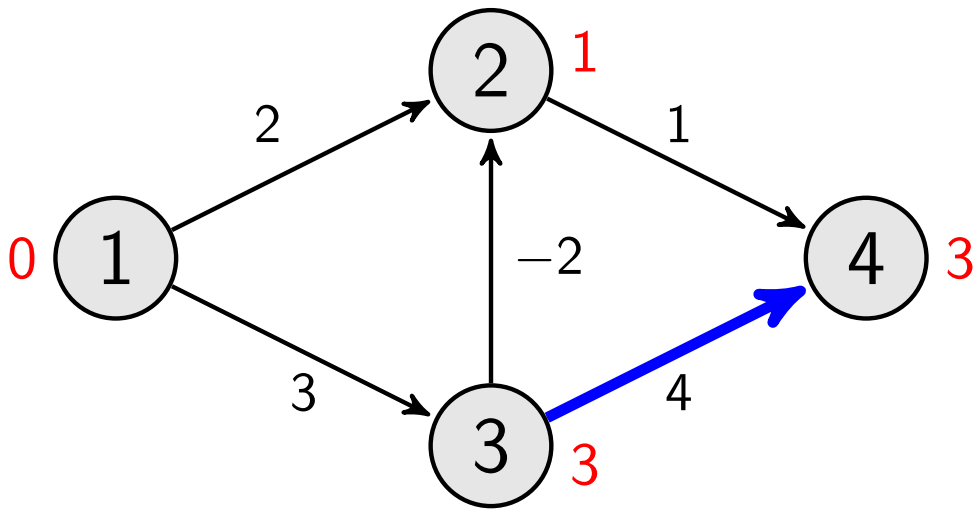
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(3, 2)$　(1 周目)



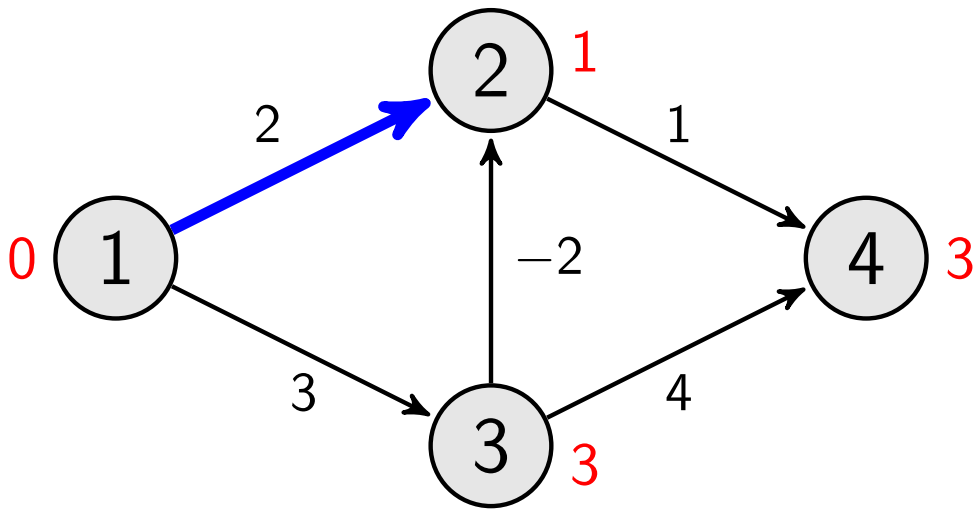
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(3, 4)$　(1 周目)



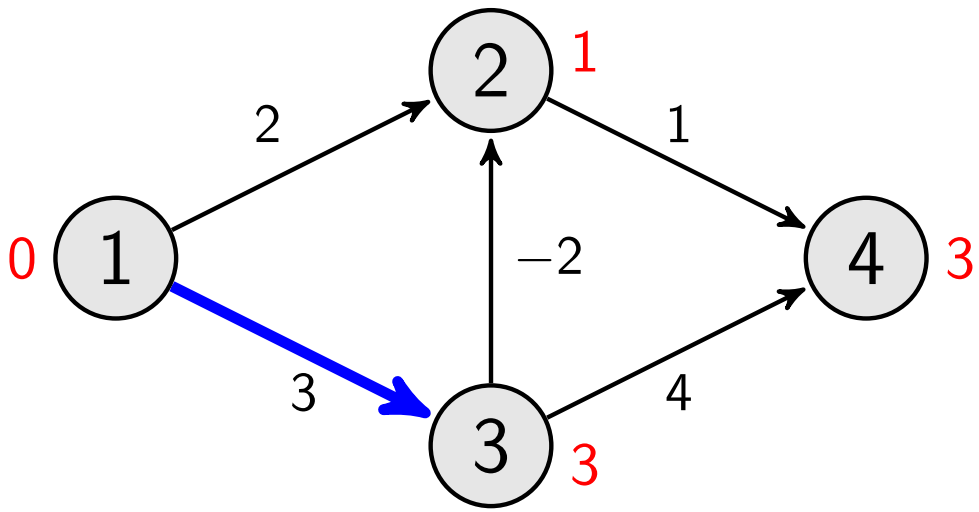
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(1, 2)$　　$(2$ 周目$)$



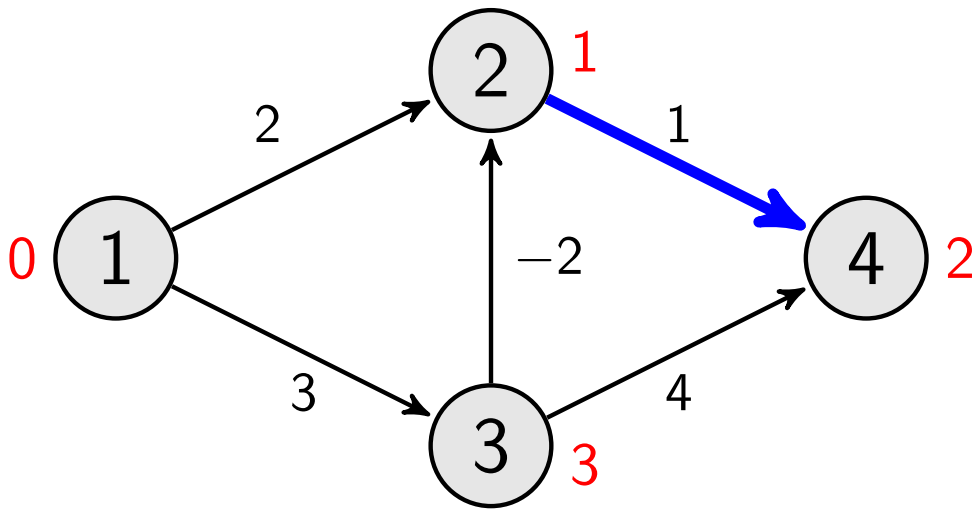
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(1, 3)$　　$(2$ 周目$)$



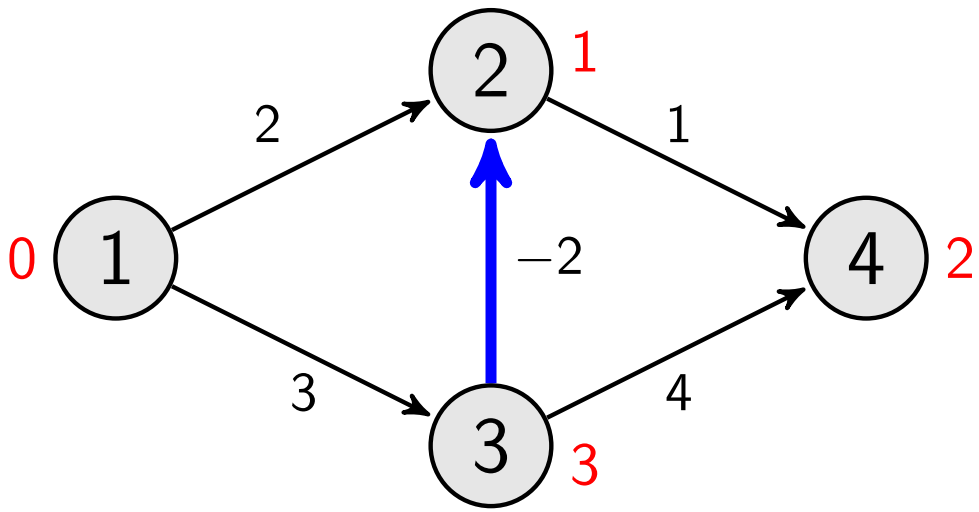
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(2, 4)$　　(2 周目)



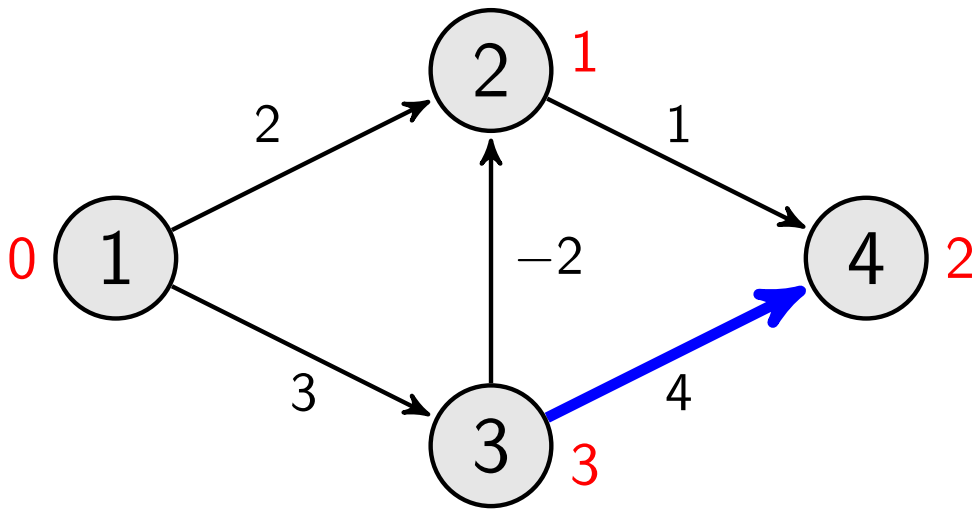
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(3, 2)$　　$(2$ 周目$)$



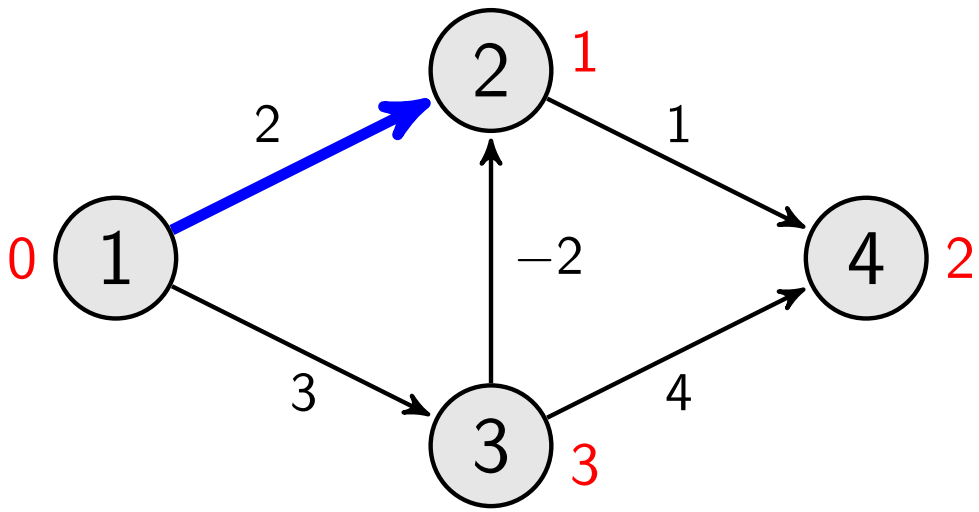
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(3, 4)$　　$(2$ 周目$)$



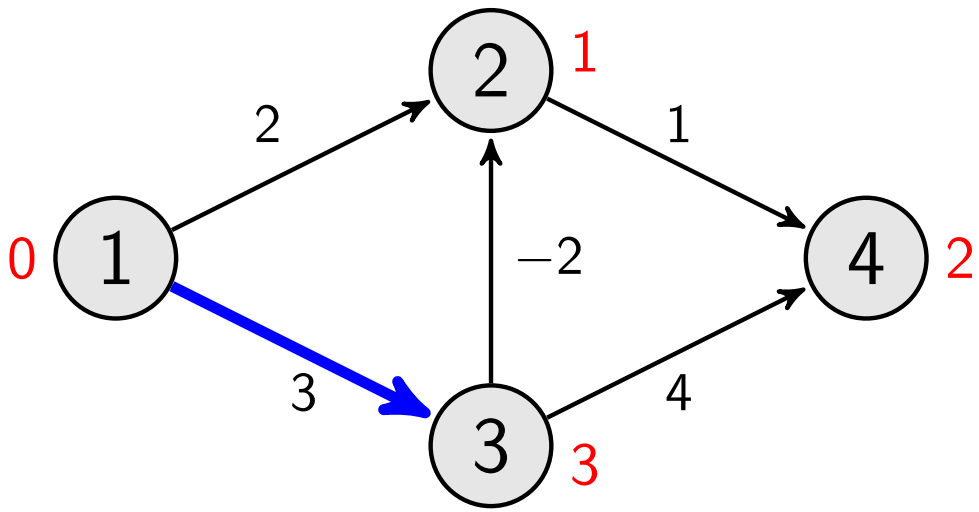
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(1, 2)$　　(3 周目)



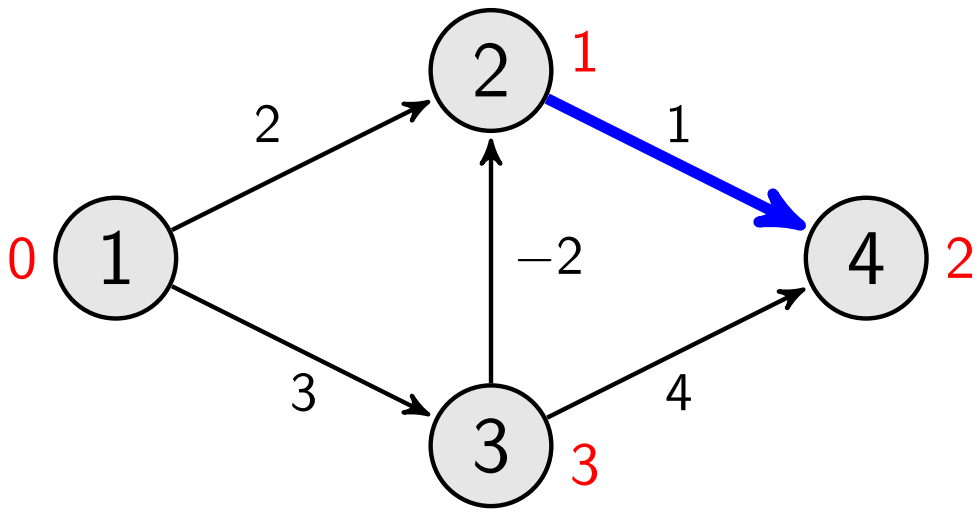
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(1, 3)$　　(3 周目)



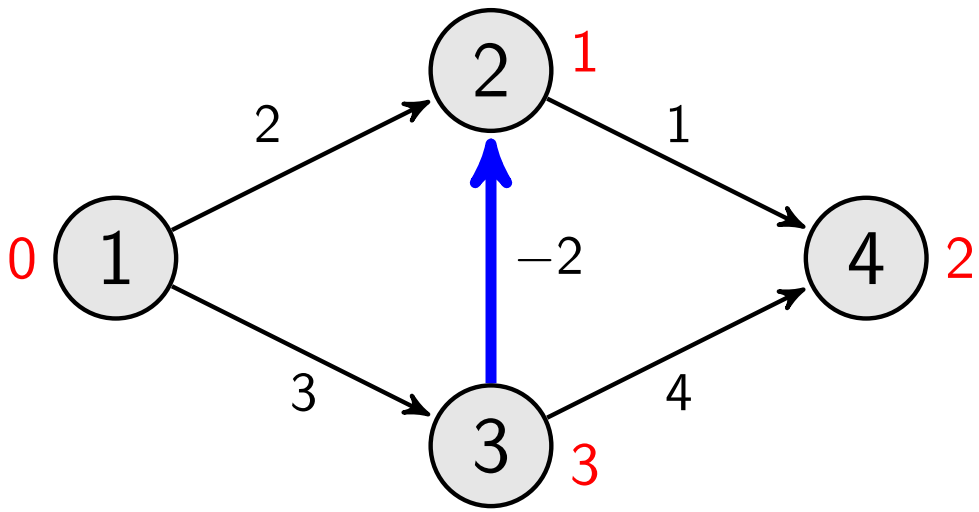
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(2, 4)$　　(3 周目)



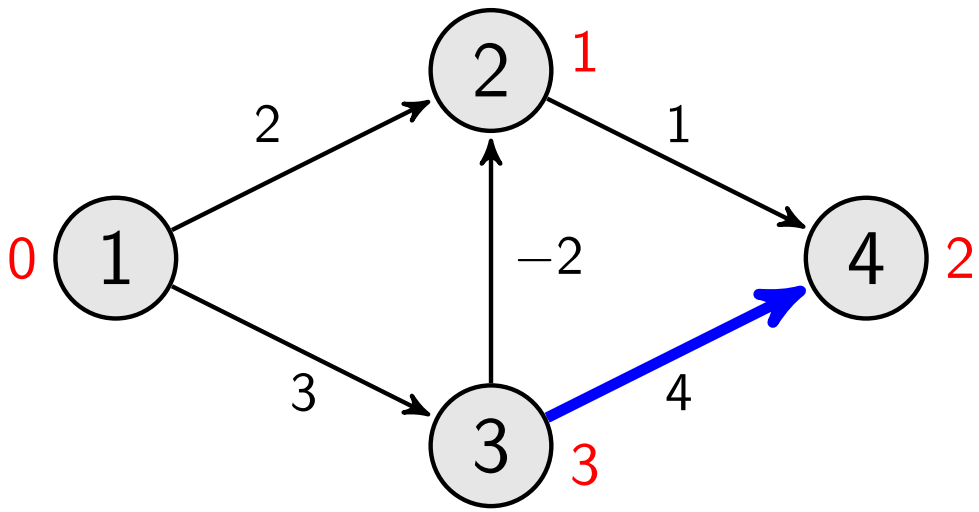
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(3, 2)$　　$(3$ 周目$)$



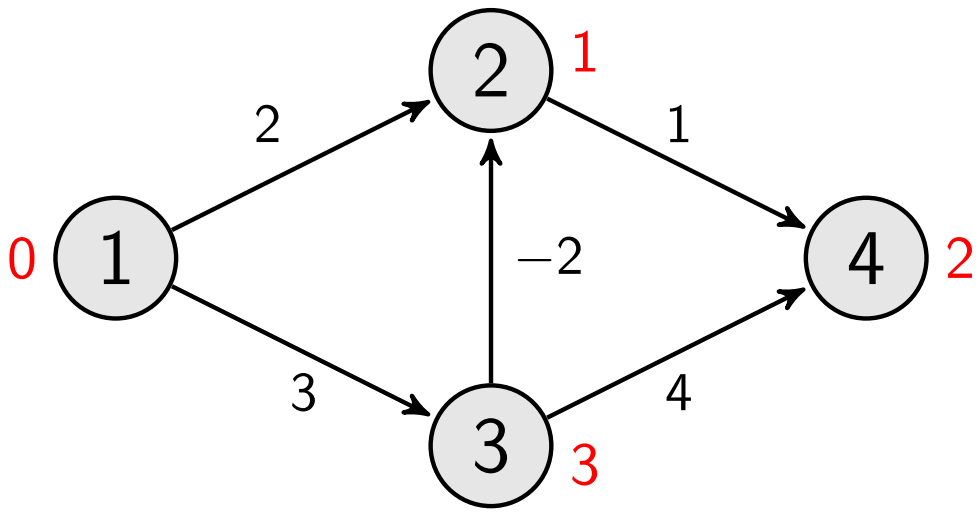
Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理: 枝 $(3, 4)$　　(3 周目)



Bellman-Ford 法の動き

- 始点 $s = 1$ とする
- 処理:



3 周目ではどれも更新されなかったので，これが答えとなる

Bellman-Ford 法 — Python による実装

```
def bellman_ford(adj,s):
    vs = adj.keys()
    dist = {v:float('inf') for v in vs} # 無限大に初期化
    dist[s] = 0
    update = True
    while update:
        update = False
        for v in vs:
            for (u,d) in adj[v]:
                if dist[u]>dist[v]+d:
                    dist[u] = dist[v]+d
                    update = True
    print(dist)
```

Warshall-Floyd 法

- 全点対の最短経路を求める方法
- 負の枝があってもよい（負閉路は駄目）
- 計算量は $O(|V|^3)$
- 方針：
  - 動的計画法を用いる
  - 点 $1, \ldots, k$ のみを用いた場合の，点 i から点 j の最短距離を部分問題とし，その距離を dist[k][i][j] とおく
  - 点 k を利用するかで場合分けすると以下が成立

$$\mathrm{dist}[k][i][j] = \min\{\mathrm{dist}[k-1][i][j], \mathrm{dist}[k-1][i][k] + \mathrm{dist}[k-1][k][j]\}$$

  - 同じ配列を使い回すことで，最初の引数は省略できる

$$\mathrm{dist}[i][j] = \min\{\mathrm{dist}[i][j], \mathrm{dist}[i][k] + \mathrm{dist}[k][j]\}$$

  - 初期値は以下のように設定する

$$\mathrm{dist}[i][j] = \begin{cases} (i,j) \text{ 間の長さ} & ((i,j) \in E), \\ 0 & (i = j), \\ \infty & (\text{otherwise}) \end{cases}$$

Warshall-Floyd 法 — Python による実装

```
def warshall_floyd(adj):
    vs = adj.keys()
    dist = {(v,u):float('inf') for v in vs for u in vs}
    for v in vs:
        dist[(v,v)]=0
        for (u,d) in adj[v]: dist[(v,u)] = d
    for k in vs:
        for i in vs:
            for j in vs:
                dist[(i,j)] = min(dist[(i,j)],dist[(i,k)]+dist[(k,j)])
print(dist)
```

- for 文の順番に注意

まとめ

Dijkstra 法

- 単一始点 (負枝 NG)
- 計算量: $O(|E| \log |E|)$

Bellman-Ford 法

- 単一始点 (負枝 OK, 負閉路 NG)
- 計算量: $O(|E| \cdot |V|)$

Warshall-Floyd 法

- 全点対 (負枝 OK, 負閉路 NG)
- 計算量: $O(|V|^3)$

アウトライン

演習問題提出方法

演習問題は，解答プログラムをまとめたテキストファイル (***.txt) を作成して，OCW-i で提出して下さい．

- 提出期限は次回の授業開始時間です．
- どの演習問題のプログラムかわかるように記述してください．
- 出力結果もつけてください．
- 途中までででもできたところまで提出してください．

演習問題 (1/2)

問 1

容量 100 と 1000 の仮定の下で下記のナップサック問題を解き，最適解での価値と，使われるアイテムの名前のリストを出力せよ．ただし各行は「重さ 価値 名前」が並んでいるものとする．

`http://yambi.jp/lecture/advanced_programming2017/vegetables.txt`

問 2

Dijkstra 法を最短経路で通る頂点も出力するように修正せよ．また，以下のグラフの 1 から 5 までの最短経路を出力せよ．

```
adj = {1:[(2,7),(3,2),(4,6)],
       2:[(4,8),(5,5)],
       3:[(4,4)],
       4:[(5,7)],
       5:[]
       }
```

演習問題 (2/2)

問 3 (おまけ)

数字のリスト a が与えられた時，$i < j$ ならば $a[i] < a[j]$ を満たす部分列で (増加部分列) 最長のものの長さを求めよ．

`http://yambi.jp/lecture/advanced_programming2017/prob6-3.txt`

例: $a = [4, 2, 8, 3, 5, 1, 7]$

問 4 (おまけ)

左上から右下までの道（上下左右のみ移動可能）で，数字の和が最小のものを求めよ．

`http://yambi.jp/lecture/advanced_programming2017/prob6-4.txt`

例：
$$\begin{pmatrix} 131 & 673 & 234 & 103 & 18 \\ 201 & 96 & 342 & 965 & 150 \\ 630 & 803 & 746 & 422 & 111 \\ 537 & 699 & 497 & 121 & 956 \\ 805 & 732 & 524 & 37 & 331 \end{pmatrix}$$