データ構造とアルゴリズム (第11回)

グラフのアルゴリズム(2)

最短経路(Shortest Path)問題

- 入力: 重み付き有向グラフ G=(V,E,c)
- 出力: V の2頂点間の距離とその距離を与える経路

易

2頂点対最短経路問題: s,t ∈ V が与えられる

難しさ

単一始点最短経路問題: s∈Vが与えられる

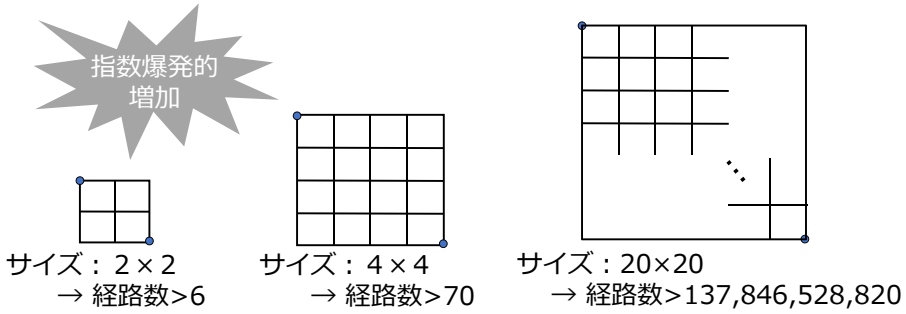
全頂点対最短経路問題

難

辺に重みがない（すべて等しい）場合は，幅優先探索でOK
重みがあるときは？

総当り法 (Brute-force)

- 出発地から目的地までの全経路を調べ，最短の経路を選択する
  - 利点：ほとんどの問題に適用可能
  - 欠点：膨大な計算時間
    (最大$(n-2)!$，下のような碁盤目でも$\binom{2\sqrt{n}}{\sqrt{n}}$)



サイズ：2×2
→ 経路数>6

サイズ：4×4
→ 経路数>70

サイズ：20×20
→ 経路数>137,846,528,820

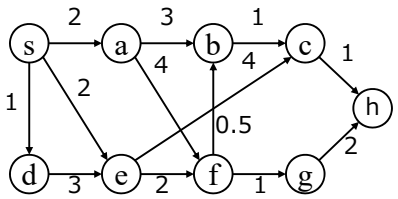
ダイクストラ法

- 基本戦略
  - 出発点に近い地点から順に「出発点からの最短経路」を求めていく
    - 結果的に，出発点から他の全ての点への最短経路が求まる

E.W.Dijkstra (1930-2002) オランダの計算機科学者
1972年チューリング賞受賞
「明快さと数学的厳密さのモデルとなった高レベルの
プログラミング言語 ALGOL の開発への貢献、および
プログラミング言語全般の構造・表現・実装の理解への多大な
貢献に対して。」

ダイクストラのアルゴリズム（アイデア）

X



重みつきグラフ G=(V,E,c)

初期状態:
X={}，D[s]=0，D[s以外]=∞

主な変数

X: 既に最短経路を求めた頂点集合

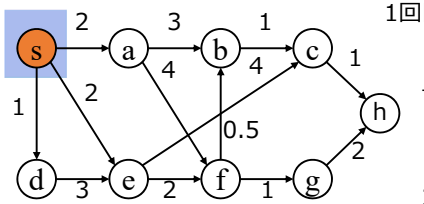
D[u]: 始点 s から頂点 u までの
X に含まれる頂点のみを
使った場合の最短距離

集合 X に含まれる頂点だけを通る経路に限定して，始点 s からの最短距離を求め，D[u] を更新

ダイクストラのアルゴリズム（実行例1）

初期状態: X={}，D[s]=0，D[s以外]=∞

X



1回目繰り返し

1. V-X 中で D[] が最小の頂点を X へ追加
→ X = {s}

2. s につながる全ての頂点(a, d, e)の D[] を更新
→ D[a]=2，D[d]=1，D[e]=2

辺(s,a)の重み

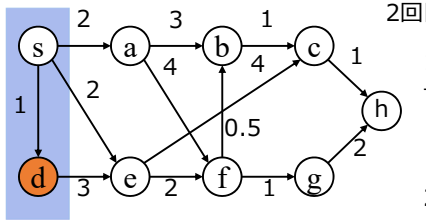
計算方法 **D[a] = min(D[a], D[s]+c(s,a))**
= min(∞, 0+2)=2

現在の距離(D[a])と，X中の頂点を使った経路による距離のうち，小さいほうを新しい距離とする.

ダイクストラのアルゴリズム（実行例2）

状態: X={s}, D[s]=0, D[a]=D[e]=2, D[d]=1, D[それ以外]=∞

X



2回目繰り返し

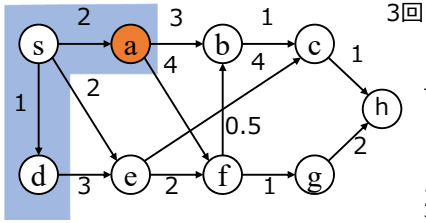
1. V-X 中で D[] が最小の頂点を X へ追加
→ X = {s, d}

2. d につながる全ての頂点 e の D[] を更新
→ D[e]=min(D[e], D[d]+c(d,e))
= min(2, 1+3=4) = 2

ダイクストラのアルゴリズム（実行例3）

状態: X={s, d}, D[s]=0, D[a]=D[e]=2, D[d]=1, D[それ以外]=∞

X



3回目繰り返し

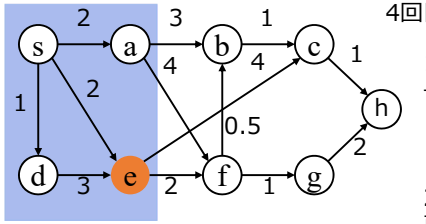
1. V-X 中で D[] が最小の頂点を X へ追加
→ X = {s, a, d}

2. a につながる全ての頂点 (b, f) の D[] を更新
→ D[b]=min(D[b], D[a]+c(a,b))
= min(∞,2+3=5) = 5
D[f] =min(D[f], D[a]+c(a,f))
=min(∞, 2+4=6) = 6

ダイクストラのアルゴリズム（実行例4）

状態: X={s, a, d}, D[s]=0, D[a]=2, D[d]=1, D[e]=2, D[b]=5, D[f]=6,
D[それ以外]=∞

X



4回目繰り返し

1. V-X 中で D[] が最小の頂点を X へ追加
→ X = {s, a, d, e}

2. e につながる全ての頂点 (c, f) の D[] を更新
→ D[c]=min(D[c], D[e]+c(e,c))
= min(∞,2+4=6) = 6
D[f] =min(D[f], D[e]+c(e,f))
=min(6, 2+2=4) = 4

ダイクストラのアルゴリズム（実行例5）

以下同様に X への追加と D[] の更新を繰り返すと…c

5回目終了時

X={s, a, d, e, f}, D[s]=0, D[a]=2, D[b]=4.5, D[c]=6, D[d]=1, D[e]=2, D[f]=4, D[g]=5, D[h]=∞

6回目終了時

X={s, a, b, d, e, f}, D[s]=0, D[a]=2, D[b]=4.5, D[c]=5.5, D[d]=1, D[e]=2, D[f]=4, D[g]=5, D[h]=∞

7回目終了時

X={s, a, b, d, e, f, g}, D[s]=0, D[a]=2, D[b]=4.5, D[c]=5.5, D[d]=1, D[e]=2, D[f]=4, D[g]=5, D[h]=7

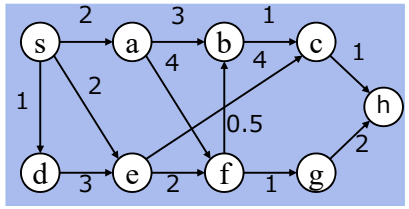
8回目終了時

状態: X={s, a, b, c, d, e, f, g}, D[s]=0, D[a]=2, D[b]=4.5, D[c]=5.5, D[d]=1, D[e]=2, D[f]=4, D[g]=5, D[h]=6.5

ダイクストラのアルゴリズム（実行例6）

状態: X={s, a, b, c, d, e, f, g}, D[s]=0, D[a]=2, D[b]=4.5, D[c]=5.5, D[d]=1, D[e]=2, D[f]=4, D[g]=5, D[h]=6.5

X



9回目繰り返し

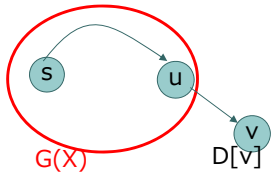
1. V-X 中で D[] が最小の頂点を X へ追加
→ X = {s, a, b, c, d, e, f, g,h}

2. h につながる全ての頂点 の D[] を更新
→ 更新なし

アルゴリズム終了

ダイクストラのアルゴリズム

```
ダイクストラのアルゴリズム：
  入力：G=(V,E,c)と始点s
  for (各頂点v∈V) D[v]=∞
  X= ∅ ; D[s]=0 ;
 while (X ≠ V) {
       V-Xの中でD[ ] が最小の頂点をuとする
       X = X ∪{u}
     for (頂点uに接続する各辺(u,v))
        D[v] = min (D[v], D[u]+c(u,v))
}
```



Xの頂点による
誘導部分グラフ

定理(ダイクストラ法の正当性)

uをXの頂点とし，v (vはXに入っていてもよい)をuに隣接する頂点とする．このとき，D[v]にはG(X)における(Xの頂点のみを用いた)sからvまでの最短経路の長さが格納されている．

→X=Vのとき D[ ]にはGにおけるsからの最短距離が入る

ダイクストラのアルゴリズム（時間計算量1）

```
ダイクストラのアルゴリズム：
  入力：G=(V,E,c)と始点s
  for (各頂点v∈V) D[v]=∞
  X= ∅ ;   D[s]=0
 while (X ≠V) {
       V-Xの中でD[ ]が最小の頂点をuとする
       X = X ∪{u}
     for (頂点uに接続する各辺(u,v))
         D[v] = min (D[v], D[u]+c(u,v))
}
```

Dの初期化：$O(n)$
Whileループ：n 回
forループ：全部で$O(m)$
(隣接リスト使用の場合)

Min(n)：(あとで)
V-Xの中でD[u]の
最小値を見つける時間

- 考えないといけないこと
  - Xの管理
  - 最小の頂点uの発見方法

ダイクストラのアルゴリズム（時間計算量1）

```
ダイクストラのアルゴリズム：
  入力：G=(V,E,c)と始点s
  for (各頂点v∈V) D[v]=∞
  X= ∅ ;   D[s]=0
 while (X ≠V) {
       V-Xの中でD[ ]が最小の頂点をuとする
       X = X ∪{u}
     for (頂点uに接続する各辺(u,v))
         D[v] = min (D[v], D[u]+c(u,v))
}
```

Dの初期化：$O(n)$
Whileループ：n 回
forループ：全部で$O(m)$
(隣接リスト使用の場合)

Min(n)：(あとで)
V-Xの中でD[u]の
最小値を見つける時間

- 考えないといけないこと
  - Xの管理
  - 最小の頂点uの発見方法

計算時間：$O(\mathrm{Min}(n)\cdot n+m)$

ダイクストラ法の実現

最小値の高速な発見

- V-X中で，D[u]の最小値を効率よく求めたい!!

プライオリティキュー(ヒープ)の利用!

最大値と言っているが最小値でも原理は同じ

PQを用いたダイクストラ法

```
PQ を用いた最短経路アルゴリズム
  入力：G=(V,E,c)と始点s
   PQ Hを初期化
  for (各頂点v∈V-{s}) {
      D[v]=∞;
      PushHeap(H, (v, D[v])) // 優先度はD[v]の値で決まるとする
   }
  D[s]=0;  PushHeap(H, (s, D[s]));
   while (Hが空でない) {
      (u, d) = DeleteMin(H);
      for (頂点uに接続する各辺(u,v) ) {
         if (D[v] > D[u]+c(u,v)) {
            D[v] = D[u]+c(u,v);
            Change(H, (v, *), (v, D[v]));
                                  // もし (v, *)がHになければなにもしない
         }
      }
   }
```

HがXを兼ねている
(Hに入っているのが
V-Xの要素)

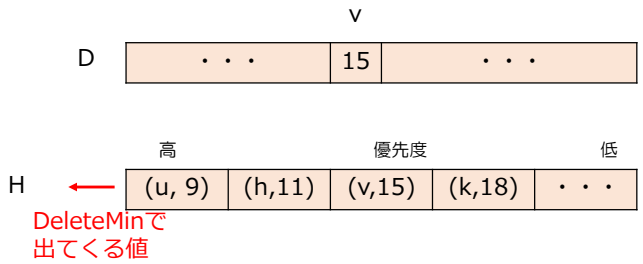
PQを用いたダイクストラ法

```
PQ を用いた最短経路アルゴリズム
  入力：G=(V,E,c)と始点s
   PQ Hを初期化
  for (各頂点v∈V-{s}) {
      D[v]=∞;
      PushHeap(H, (v, D[v])) // 優先度はD[v]の値で決まるとする
   }
  D
   v
                  if (D[v] > D[u]+c(u,v)) {
                     D[v] = D[u]+c(u,v);
                     Change(H, (v, *), (v, D[v]));
                                  // もし (v, *)がHになければなにもしない
         }
      }
   }
```

問題：どうやってH中のvの場所を探す？
(Hがヒープで実現されている場合，vの格納位置が
わかれば$O(\log n)$時間で優先度の変更は可能だが，
格納位置の発見は線形探索を用いるしかない→$\Omega(n)$時間かかる)

解決策

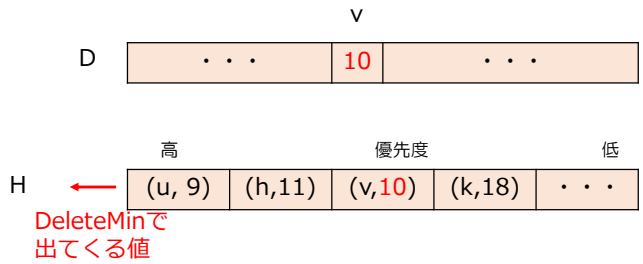
- ダイクストラ法では，優先度(D[v])の変更は必ず優先度が上がる方向での変更になる
  - Changeで優先度が変わるのは距離が短くなった時(D[v]が小さくなる時)のみ



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

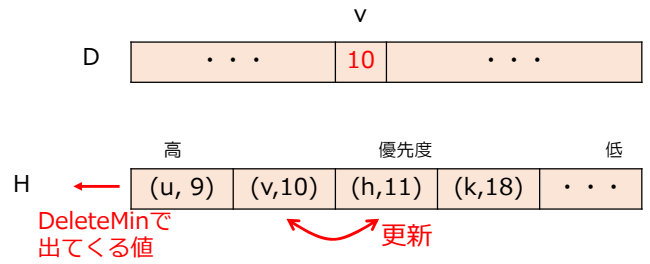
- ダイクストラ法では，優先度(D[v])の変更は必ず優先度が上がる方向での変更になる
  - Changeで優先度が変わるのは距離が短くなった時(D[v]が小さくなる時)のみ



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

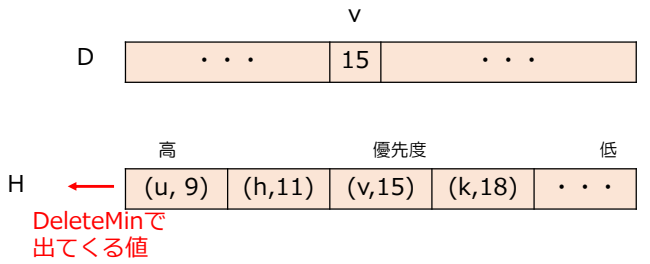
- ダイクストラ法では，優先度(D[v])の変更は必ず優先度が上がる方向での変更になる
  - Changeで優先度が変わるのは距離が短くなった時(D[v]が小さくなる時)のみ



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

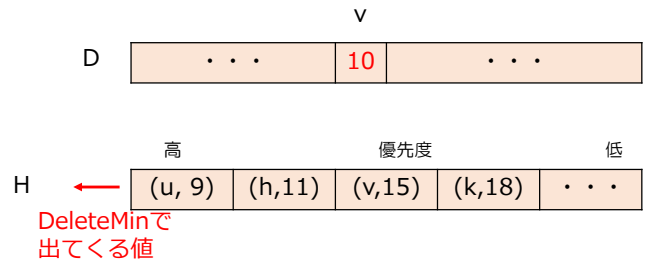
- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

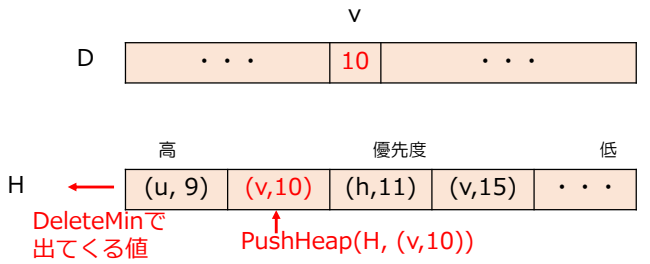
- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

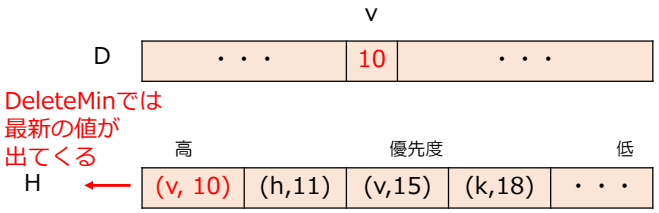
- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

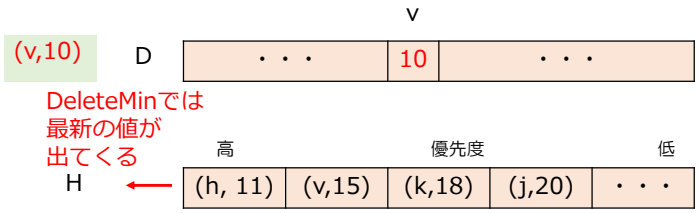
- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

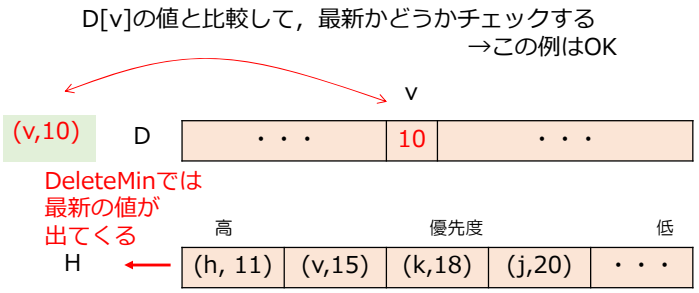
- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

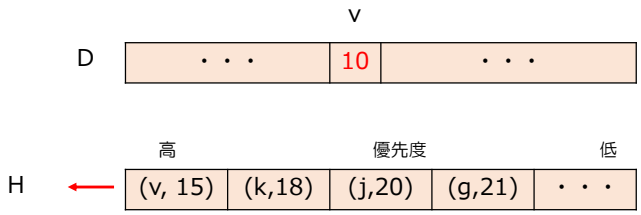
- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

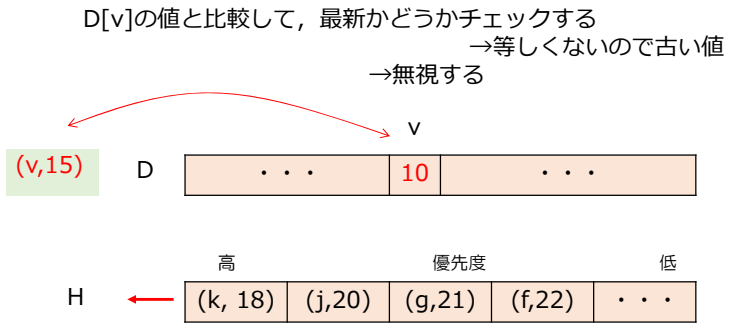
- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

解決策

- 優先度が下がらない場合ChangeをPushHeapで代用可能



(Hは実際の配列中の並びを表しているわけではないので注意)

PQを用いたダイクストラ法：完全版

```
PQ を用いた最短経路アルゴリズム
  入力：G=(V,E,c)と始点s
   PQ Hを初期化
  for (各頂点v∈V-{s}) {
      D[v]=∞;
      PushHeap(H, (v, D[v])) // 優先度はD[v]の値で決まるとする
   }
  D[s]=0;  PushHeap(H, (s, D[s]));
   while (Hが空でない) {
      (u, d) = DeleteMin(H);
      if (d =D[u]) {       //  取り出した値が古い値でないかどうか確認
         for (頂点uに接続する各辺(u,v) ) {
            if (D[v] > D[u]+c(u,v)) {
               D[v] = D[u]+c(u,v);
               PushHeap(H, (v, D[v]));
            }
         }
      }
}
```

PQを用いたダイクストラ法：計算時間

```
PQ を用いた最短経路アルゴリズム
  入力：G=(V,E,c)と始点s
   PQ Hを初期化
  for (各頂点v∈V-{s}) {                                  } O(n log n)
      D[v]=∞;
      PushHeap(H, (v, D[v])) // 優先度はD[v]の値で決まるとする
   }
  D[s]=0;  PushHeap(H, (s, D[s]));
   while (Hが空でない) {
      (u, d) = DeleteMin(H);
      if (d =D[u]) {       //  取り出した値が古い値でないかどうか確認
         for (頂点uに接続する各辺(u,v) ) {
            if (D[v] > D[u]+c(u,v)) {                    } 各頂点uに対して
               D[v] = D[u]+c(u,v);                       1回のみ実行
               PushHeap(H, (v, D[v]));                   PushHeapはO(log n)時間
            }                                            → 計O(m log n)時間
         }
      }
}
```

PQを用いたダイクストラ法：計算時間

```
PQ を用いた最短経路アルゴリズム
  入力：G=(V,E,c)と始点s
   PQ Hを初期化
  for (各頂点v∈V-{s}) {                                  } O(n log n)
      D[v]=∞;
      PushHeap(H, (v, D[v])) // 優先度はD[v]の値で決まるとする
   }
  D[s]=0;  PushHeap(H, (s, D[s]));
   while (Hが空でない) {
      (u, d) = DeleteMin(H);
      if (d =D[u]) {       //  取り出した値が古い値でないかどうか確認
         for (頂点uに接続する各辺(u,v) ) {
            if (D[v] > D[u]+c(u,v)) {                    } 各頂点uに対して
               D[v] = D[u]+c(u,v);                       1回のみ実行
               PushHeap(H, (v, D[v]));                   PushHeapはO(log n)時間
            }                                            → 計O(m log n)時間
         }
      }
}
```

Hには高々m回PushHeap
→ DeleteMin m回で空
→ $O(m \log n)$時間

ダイクストラ法：余禄

- PQを用いたダイクストラ法では，以下の操作が高速に実行できることが本質であった
  - 値の追加(PushHeap)
  - 最小値の削除(DeleteMin)
  - キー値の減少(Change (DecreaseKey))
- さらなる高速化：フィボナッチヒープの利用
  - PushHeap：$O(\log n)$
  - DeleteMin：$O(\log n)$
  - DecreaseKey：ならし(amortized)で$O(1)$
    - 個別に$O(1)$時間は保証されないが，全体で合計$O(m)$時間は保証

ダイクストラ法：余禄

- フィボナッチヒープを利用したダイクストラ法
  - $O(n \log n + m)$時間を達成可能 (cf. 通常版$O((n+m)\log n)$)
- ただし，実際にはそこまで実用的ではない
  - フィボナッチヒープの漸近的計算量に隠れた係数が大きい
  - 入力グラフが疎な場合は差がない

ダイクストラ法：余禄

- ダイクストラ法は辺に負の重みがあると動かない
- 一般に「負の重みの閉路」がある場合，最短経路長は不定(マイナス無限大)である
  - 負閉路があると，そこを何周もすることで経路長をいくらでも短くできる
- 負重みをもつグラフの最短経路問題の解法としてはベルマン=フォード法がよく知られている
  - 負閉路があるときはそれを検知もできる
  - ダイクストラ法よりかはだいぶ遅い($O(nm)$時間)

全点対最短経路問題

全頂点対最短経路問題

- 2頂点対最短経路問題，単一始点最短経路問題
  - ダイクストラのアルゴリズム
    - フィボナッチヒープを利用　$O(n \log n + m)$時間
- 全頂点対最短経路問題
  - ダイクストラのアルゴリズムを各頂点を始点として適用
    - $O(n^2 \log n + mn)$時間（$=O(n^3)$）
  - フロイド(Floyd)のアルゴリズム
    (Warshall-Floyd / Floyd-Warshallのアルゴリズムと呼ばれることもある)
    - $\boldsymbol{O(n^3)}$ 時間（ダイクストラ×nと差なしか，疎なグラフでは遅い）
      - 操作が簡単なので隠れた定数係数が小さい
        →密なグラフでは実用的には高速
      - 負の重みの辺があっても適用可能

フロイドのアルゴリズム

- 方針：再帰的構成
  - 頂点集合$V = [1, \mathrm{n}]$として，$V_k = [1, k]$とする
  - $a_k[i,j]$ = 頂点 $1,2,\ldots,k$ だけを経由する
    頂点 i から頂点 j への最短路の長さ
  - $a_0[i,j]$ = 辺 (i,j) の重み
  - $a_n[i,j]$ = 頂点 i から頂点 j への最短路の長さ

補題(Floydのアルゴリズムの再帰式)

$$a_k[i,j] = \min\{a_{k-1}[i,j], a_{k-1}[i,k] + a_{k-1}[k,j]\}$$

- 既に分かっている最短路と新たに追加された頂点を
  経由する最短路を比べて，よい方を選ぶ

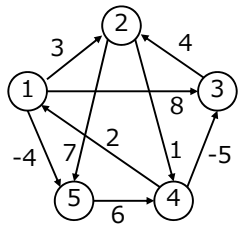
フロイドのアルゴリズム

- kの小さいところから始めて，順次再帰式を使って値を
  定めていく
  - k回目の反復終了時の配列aの値がa_kに相当
    - $c[i,j]$ =辺(i,j)の重み
      - 辺がないときは∞，$c[i,i] = 0$
        (任意の値x(負のxでも)に対して，$\infty + x = \infty$ と定める)

```
フロイドのアルゴリズム：
for i:= 1 to n do
    for j:= 1 to n do
      a[i,j]:= c[i,j];
  for k:= 1 to n do
    for i:= 1 to n do
      for j:= 1 to n do
        if a[i,k] + a[k,j] < a [i,j] then
          a[i,j]:= a[i,k] + a[k,j]
```


実行例

```
フロイドのアルゴリズム：
for i:= 1 to n do
    for j:= 1 to n do
      a[i,j]:= c[i,j];
  for k:= 1 to n do
    for i:= 1 to n do
      for j:= 1 to n do
        if a[i,k] + a[k,j] < a [i,j] then
          a[i,j]:= a[i,k] + a[k,j]
```



配列a

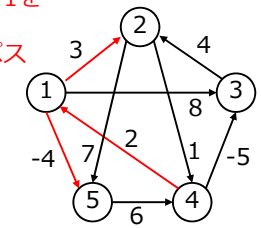
	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

実行例

フロイドのアルゴリズム：

```
for i:= 1 to n do
    for j:= 1 to n do
        a[i,j]:= c[i,j];
    for k:= 1 to n do
        for i:= 1 to n do
            for j:= 1 to n do
                if a[i,k] + a[k,j] < a [i,j] then
                    a[i,j]:= a[i,k] + a[k,j]
```

中継ノードとして1を
考慮することで
新たに発見したパス



配列a

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

配列a(1反復後)

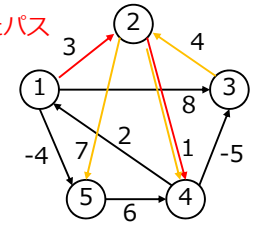
	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

実行例

フロイドのアルゴリズム：

```
for i:= 1 to n do
    for j:= 1 to n do
        a[i,j]:= c[i,j];
    for k:= 1 to n do
        for i:= 1 to n do
            for j:= 1 to n do
                if a[i,k] + a[k,j] < a [i,j] then
                    a[i,j]:= a[i,k] + a[k,j]
```

2を考慮して
新たに発見したパス



配列a(1反復後)

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

配列a(2反復後)

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

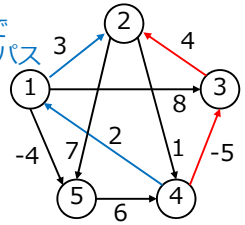
実行例

フロイドのアルゴリズム：

```
for i:= 1 to n do
    for j:= 1 to n do
        a[i,j]:= c[i,j];
    for k:= 1 to n do
        for i:= 1 to n do
            for j:= 1 to n do
                if a[i,k] + a[k,j] < a [i,j] then
                    a[i,j]:= a[i,k] + a[k,j]
```

3を考慮して
新たに発見したパス

2反復目までで
わかっていたパス



配列a(2反復後)

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

配列a(3反復後)

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

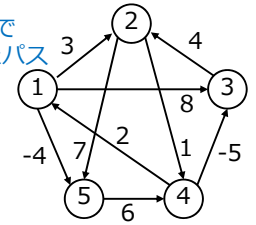
実行例

フロイドのアルゴリズム：

```
for i:= 1 to n do
    for j:= 1 to n do
        a[i,j]:= c[i,j];
    for k:= 1 to n do
        for i:= 1 to n do
            for j:= 1 to n do
                if a[i,k] + a[k,j] < a [i,j] then
                    a[i,j]:= a[i,k] + a[k,j]
```

3を考慮して
新たに発見したパス

2反復目までで
わかっていたパス



配列a(3反復後)

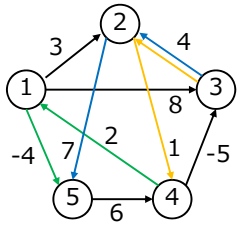
	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

配列a(4反復後)

	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

実行例

フロイドのアルゴリズム：
for $i := 1$ to n do
  for $j := 1$ to n do
    $a[i,j] := c[i,j]$;
  for $k := 1$ to n do
    for $i := 1$ to n do
      for $j := 1$ to n do
        if $a[i,k] + a[k,j] < a[i,j]$ then
          $a[i,j] := a[i,k] + a[k,j]$



配列a(3反復後)

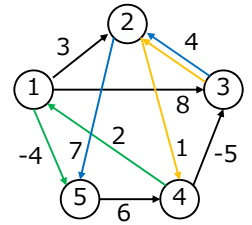
	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

配列a(4反復後)

	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

実行例

フロイドのアルゴリズム：
for $i := 1$ to n do
  for $j := 1$ to n do
    $a[i,j] := c[i,j]$;
  for $k := 1$ to n do
    for $i := 1$ to n do
      for $j := 1$ to n do
        if $a[i,k] + a[k,j] < a[i,j]$ then
          $a[i,j] := a[i,k] + a[k,j]$



配列a(4反復後)

	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

配列a(5反復後．アルゴリズム終了)

	1	2	3	4	5
1	0	3	-1	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

負閉路の検出

- Floydのアルゴリズムは負閉路が存在すればそれを検出可能
  - アイデア：計算終了後の配列$a[i,i]$はiからiへの単純な最短経路長が格納
    - もし負閉路が存在すれば，閉路上の頂点iは自分から自分への長さ負の単純経路を持つ → $a[i,i]$が負になる
  - aの対角成分が非ゼロのとき，元の入力は負閉路を持つ

補足

- フロイドのアルゴリズムは基本隣接行列を用いる
  - アルゴリズムの構造的に，明らかにそのほうが高速
  - 全店対最短経路問題は，そもそも出力サイズが$\Theta(n^2)$(words)なので，隣接行列のスペースの問題はデメリットにならないことに注意
- フロイドのアルゴリズムより大幅に早いアルゴリズムは知られていない
  - 知られている最速アルゴリズム：$O\left(\frac{n^3}{2^{\sqrt{\Omega(\log n)}}}\right)$時間
  - $O(n^{2.99})$時間のアルゴリズムを作れたら歴史に名が残る
    (存在しないという意見もあり)