データ構造とアルゴリズム (第14-15回)

アルゴリズム設計手法

この章の学習目標

- さまざまなアルゴリズム設計手法を学ぶ
  - 分割統治法
  - 動的計画法
  - 貪欲法
  - 計算困難問題への対処
    (近似率保証・局所探索・分枝限定法)

Part1: 分割統治法

分割統治法

- 再帰アルゴリズムの一種
  - 問題を小さな部分問題に分割(分割フェーズ)
    - 部分問題は再帰的に解を求める
  - 部分問題の解から元の問題の解を構成(統治フェーズ)

分割統治法の代表例：マージソート

58	12	39	90	49	26	68	47	15	39

分割

58	12	39	90	49

26	68	47	15	39

分割された問題は再帰的に独立に解かれる

~~12~~	~~39~~	~~49~~	~~58~~	~~90~~

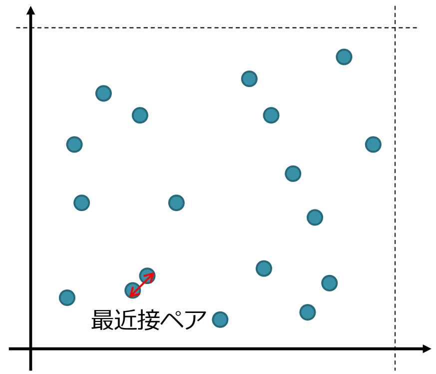
~~15~~	~~26~~	~~39~~	~~47~~	~~68~~

それぞれの問題の解からもとの問題の解を構成する(統治)

12	15	26	39	39	47	49	58	68	90

最近接ペア問題

- 入力：2次元平面上のn点(座標)
- 距離が最も近い点対(の距離)を求めよ



自明な解法

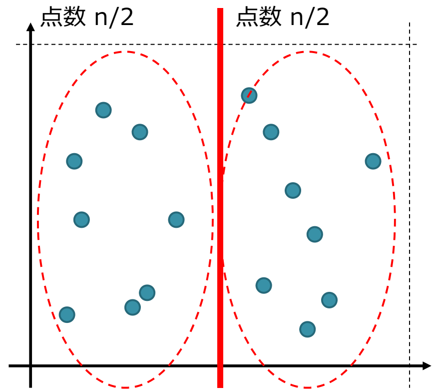
- 全点対の総当たり探索(Brute-Force法)

$$\#\text{全点対} = \binom{n}{2} = O(n^2)$$

  - 距離の計算は$O(1)$ 時間→全体で$O(n^2)$時間
- 分割統治法を用いると$O(n \log n)$時間で解ける！

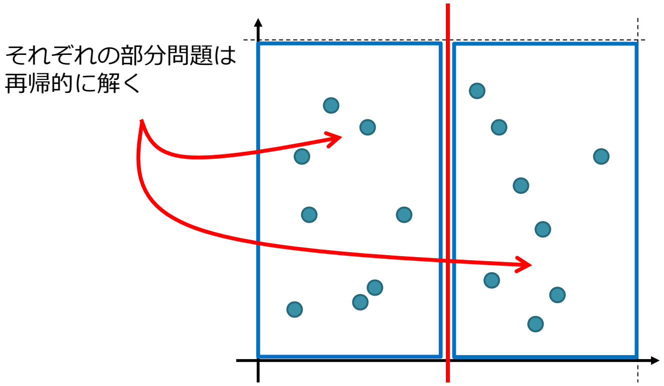
アイデア

- 平面を分割する(点数が均等に分かれるように)
  - 分割されたそれぞれの問題の点数は$n/2$



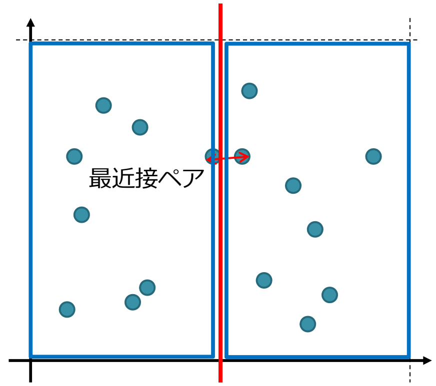
アイデア

- 平面を分割する(点数が均等に分かれるように)
  - 分割されたそれぞれの問題の点数は$n/2$



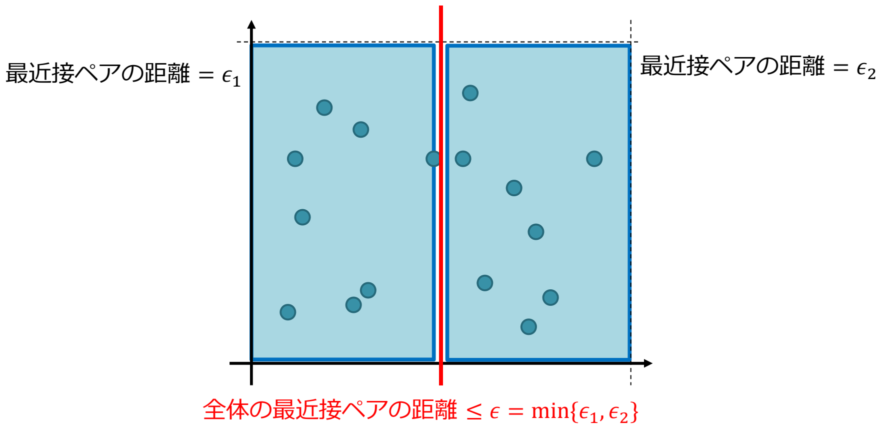
統治

- 最近接ペアが分けた領域にまたがるときが問題
  →統治フェーズでカバーする



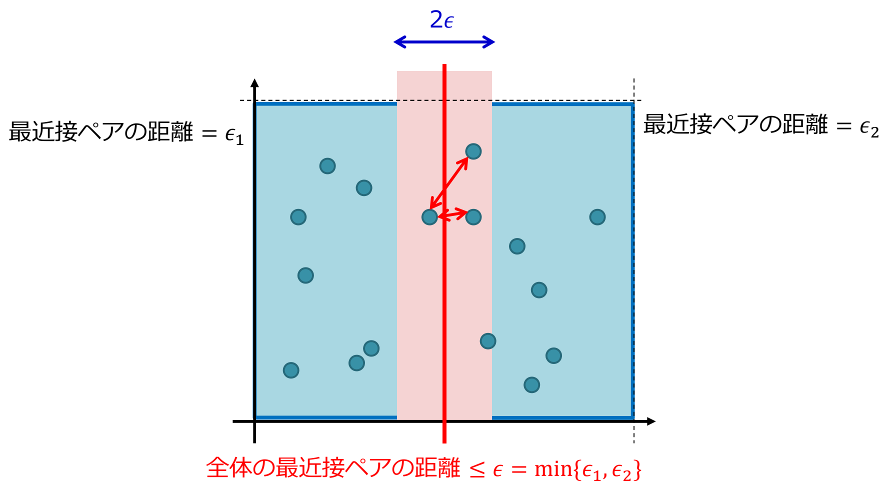
統治

- 分割した部分問題の解は，真の解の上限を与える



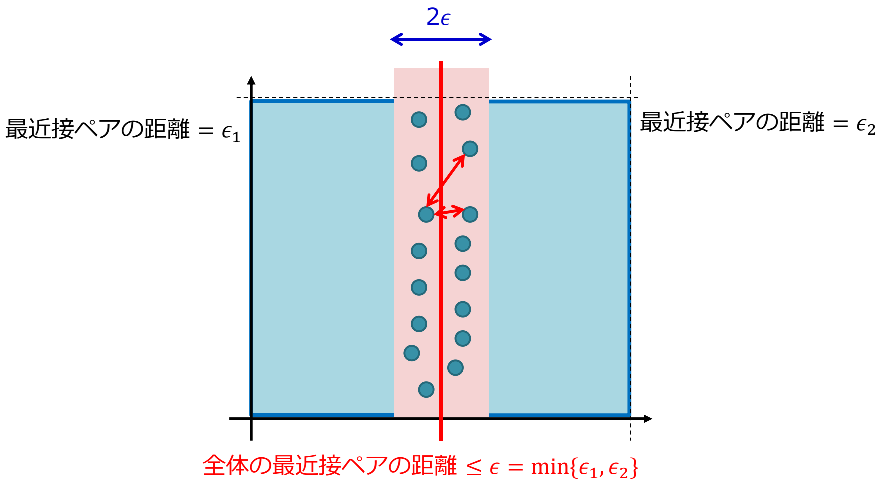
統治

- 高速な統治のアイデア：上界を用いて計算対象を削減する



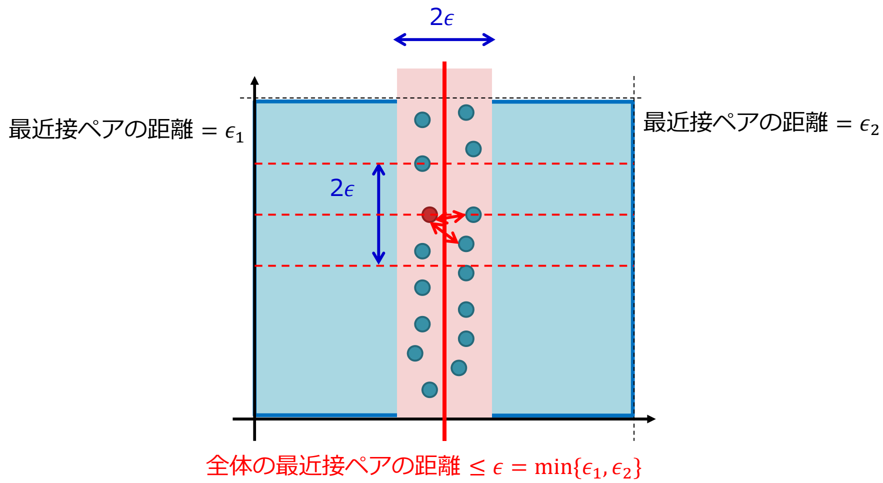
統治

- 問題：候補が削減されないことがある



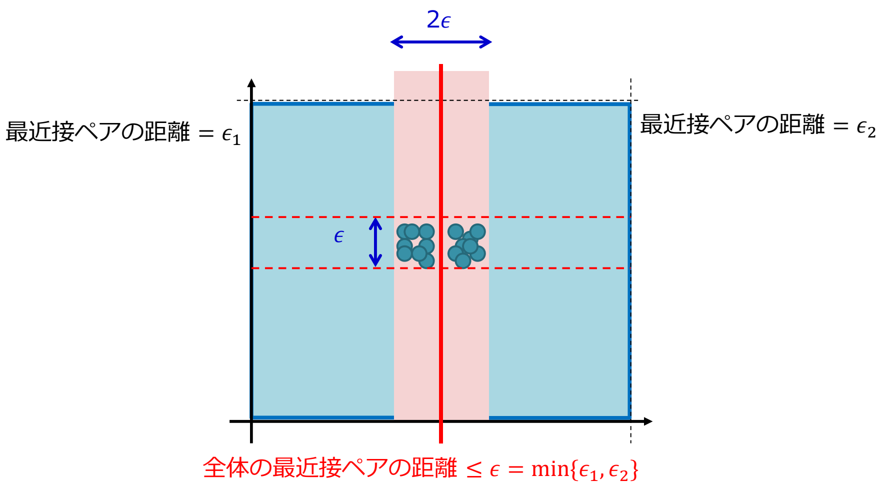
統治

- さらなる削減:y方向でも範囲を限定する



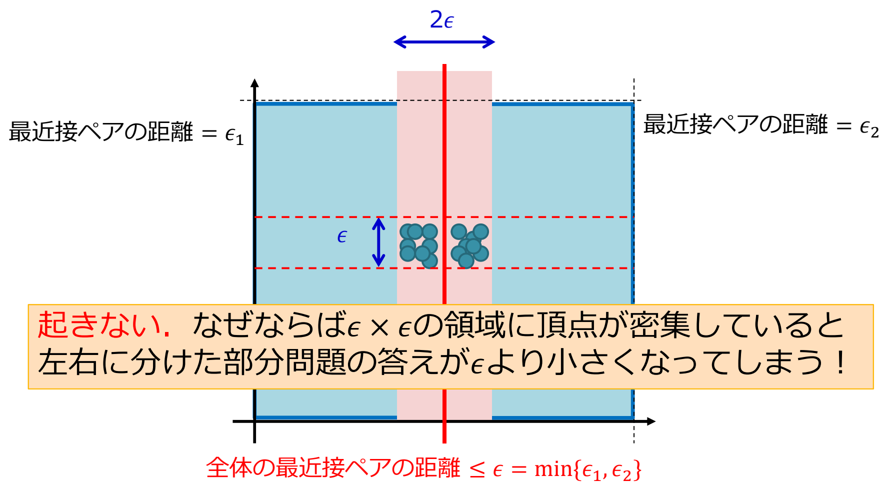
統治

- 疑問再び：同じことがおきないか？



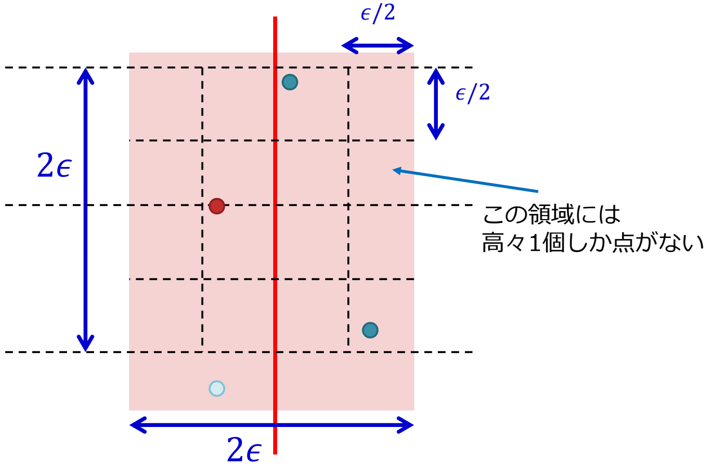
統治

- 疑問再び：同じことがおきないか？



統治

- 実際に比較が必要なのは高々16個しか存在しない！
  →分割線周辺で計算が必要な点対は高々$16n = O(n)$個



統治フェーズの実現

- いくつかの考慮しなければいけないこと
  - 分割線をどう見つけるか？
  - 分割線周辺(x座標±ϵ)の点集合をどう見つけるか？
  - 分割線周辺の各点について，y座標が前後±ϵの候補をどう見つけるか？

統治フェーズの実現

- いくつかの考慮しなければいけないこと
  - 分割線をどう見つけるか？
  - 分割線周辺(x座標±ϵ)の点集合をどう見つけるか？
  - 分割線周辺の各点について，y座標が前後±ϵの候補をどう見つけるか？

解決策：x,y座標の両方でソート

アルゴリズムの設計

- ソートは$O(n \log n)$時間かかるので，再帰の外で行う
- 再帰的な関数は以下のように定義する

closest_pair(xsort, ysort)：
点集合をx座標でソートしたもの(xsort)
点集合をy座標でソートしたもの(ysort)
から最近接ペア(の距離)を返す

簡単のために距離のみ
返しているが，ペア自体を
返すようには容易に変更可能

- 動作の詳細

```
closest_pair(xsort,ysort) {
  (1) xsortを半分に分ける分割線を見つける;
  (2) 分割線で(xsort, ysort)を左側(lxsort, lysort) と (rxsort, rysort)に分ける;
  (3) ϵ = min(closest_pair(lxsort, lysort), closest_pair(rxsort,rysort));
  (4) ysort’ = ysortからx座標分割線±ϵの座標のみを抽出する
  (5) ysort’ 中の各点について(ソート列中で)前後±8個とのみ比較
      ϵより距離の小さいペアがあればそれでϵを更新
}
```

実行例：(1)-(2) (分割フェーズ)

- 入力がx,y座標それぞれでソートされているとする

by x	(0,5)	(1,6)	(2,7)	(3,3)	(4,2)	(4,9)	(5,1)	(8,3)
by y	(5,1)	(4,2)	(8,3)	(3,3)	(0,5)	(1,6)	(2,7)	(4,9)

- 分割：xソート列を中央で半分に割る

(0,5)	(1,6)	(2,7)	(3,3)	x=3	(4,2)	(4,9)	(5,1)	(8,3)

実行例：(1)-(2) (分割フェーズ)

- 入力がx,y座標それぞれでソートされているとする ($O(n \log n)$時間)

by x	(0,5)	(1,6)	(2,7)	(3,3)	(4,2)	(4,9)	(5,1)	(8,3)
by y	(5,1)	(4,2)	(8,3)	(3,3)	(0,5)	(1,6)	(2,7)	(4,9)

- 分割に応じて，y座標のソート列を左右に振り分ける
  - 分割線のx座標が分かっていれば$O(n)$時間

(0,5)	(1,6)	(2,7)	(3,3)	x=3	(4,2)	(4,9)	(5,1)	(8,3)

実行例：(1)-(2) (分割フェーズ)

- 入力がx,y座標それぞれでソートされているとする ($O(n \log n)$時間)

by x	(0,5)	(1,6)	(2,7)	(3,3)	(4,2)	(4,9)	(5,1)	(8,3)
by y	(5,1)	(4,2)	(8,3)	(3,3)	(0,5)	(1,6)	(2,7)	(4,9)

- 分割に応じて，y座標のソート列を左右に振り分ける
  - 分割線のx座標が分かっていれば$O(n)$時間

(0,5)	(1,6)	(2,7)	(3,3)	x=3	(4,2)	(4,9)	(5,1)	(8,3)
(3,3)	(0,5)	(1,6)	(2,7)		(5,1)	(4,2)	(8,3)	(4,9)

実行例：(1)-(2) (分割フェーズ)

- 入力がx,y座標それぞれでソートされているとする ($O(n \log n)$時間)

by x	(0,5)	(1,6)	(2,7)	(3,3)	(4,2)	(4,9)	(5,1)	(8,3)
by y	(5,1)	(4,2)	(8,3)	(3,3)	(0,5)	(1,6)	(2,7)	(4,9)

- 分割に応じて，y座標のソート列を左右に振り分ける
  - 分割線のx座標が分かっていれば$O(n)$時間

(0,5)	(1,6)	(2,7)	(3,3)	x=3	(4,2)	(4,9)	(5,1)	(8,3)
(3,3)	(0,5)	(1,6)	(2,7)		(5,1)	(4,2)	(8,3)	(4,9)

(lxsort, lysort)　(3) (再帰呼び出し)を実行　(rxsort, rysort)

実行例：(4)-(5) (統治フェーズ)

- $\epsilon = \sqrt{2}$, 分割線のx座標= 3 とする

by x	(0,5)	(1,6)	(2,7)	(3,3)	(4,2)	(4,9)	(5,1)	(8,3)
by y	(5,1)	(4,2)	(8,3)	(3,3)	(0,5)	(1,6)	(2,7)	(4,9)

  - yソート列に対して，x座標 $\notin [3-\sqrt{2}, 3+\sqrt{2}]$の座標を削除した列を作成する

実行例：(4)-(5) (統治フェーズ)

- $\epsilon = \sqrt{2}$, 分割線のx座標= 3 とする

by x	(0,5)	(1,6)	(2,7)	(3,3)	(4,2)	(4,9)	(5,1)	(8,3)
by y	~~(5,1)~~	(4,2)	~~(8,3)~~	(3,3)	~~(0,5)~~	~~(1,6)~~	(2,7)	(4,9)

  - yソート列に対して，x座標 $\notin [3-\sqrt{2}, 3+\sqrt{2}]$の座標を削除した列を作成する

(4,2)	(3,3)	(2,7)	(4,9)

実行例：(4)-(5) (統治フェーズ)

- $\epsilon = \sqrt{2}$, 分割線のx座標= 3 とする

by x	(0,5)	(1,6)	(2,7)	(3,3)	(4,2)	(4,9)	(5,1)	(8,3)
by y	~~(5,1)~~	(4,2)	~~(8,3)~~	(3,3)	~~(0,5)~~	~~(1,6)~~	(2,7)	(4,9)

  - yソート列に対して，x座標 $\notin [3-\sqrt{2}, 3+\sqrt{2}]$の座標を削除した列を作成する($O(n)$時間)

(4,2)	(3,3)	(2,7)	(4,9)

  - この配列の各要素について，前後±8個との距離を計算($O(n)$時間)

(このケースは点数が少なすぎて結局全部比較されるが．．．)

実行時間

- $T(n)$：n頂点の最近接ペア発見問題の最悪時実行時間
  - アルゴリズム全体の実行時間は事前のソートがあるので$O(n \log n) + T(n)$時間
- $T(n)$の評価の漸化式：分割，統治のフェーズがいずれも$O(n)$時間なので

$$T(n) \le 2T\left(\frac{n}{2}\right) + O(n)$$

  - これを解いて $T(n) = O(n \log n)$　全体で$O(n \log n)$時間

Part2: 動的計画法

動的計画法：フィボナッチ数列の計算

- フィボナッチ数列：定義

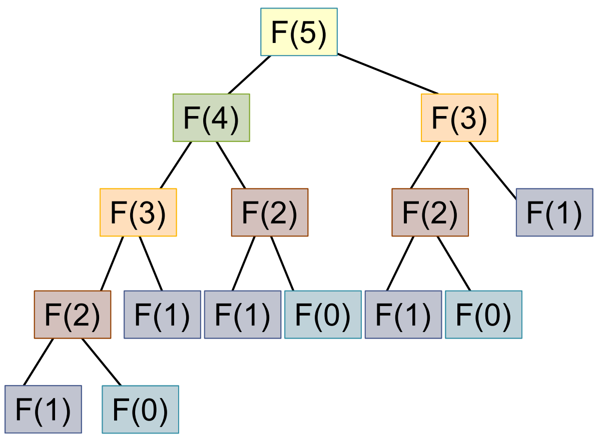
$$\begin{cases} F(0) = F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \quad (n > 1) \end{cases}$$

1, 1, 2, 3, 5, 8, 13, 21, ・・・

- ストレートな再帰アルゴリズム

```
int F(int n) {
    if(n==0 || n==1) return 1;
    else return(F(n-1)+F(n-2));
}
```

F(n)の再帰プログラムの計算過程



同じ関数呼び出しが何度も生じる！

計算時間は指数時間($O(1.62^n)$くらい)

高速化：メモ化

過去の履歴を記録

履歴があるものは再計算しない

```
int Fmemo[0 .. ∞];
for all i do Fmemo[i] = -1


int F(int n) {
    if (Fmemo[n] ≠ -1) return Fmemo[n];
    if(n==0 || n==1) return 1;
    else {
        Fmemo[n] = F(n – 1) + F(n -2);
        return(Fmemo[n]);
}
```

高速化：表を埋める

```
int Fibonacci(int n){
    int F[n] , i ;
    F[0] = F[1] = 1;
    for (i = 2 to n)
        F[i] = F[i-1]+F[i-2];
    return F[n];
}
```

小さいところから
配列を埋めていく

F	1	1	2	3					

いずれの高速化も計算時間はO(n)！

ナップサック問題(KP)

- 入力: (b, w)
  - b: ナップサックの容量
  - $(w_1, w_2, \cdots w_n)$: w_iはi番目のアイテムの重さ
  - $(c_1, c_2, \cdots c_n)$: c_iはi番目のアイテムの価値
- 出力
  - 以下の条件を満たすアイテムの集合$T \subseteq \{1,2, \cdots n\}$ で価値の総和が最大のもの

$$\sum_{i \in T} w_i \leq b \quad \text{(容量制約という)}$$

「ナップサックに詰め込める」

容量制約を満たすT：許容解
最も価値の個数の多い許容解：最適解

KPに対する素朴なアルゴリズム

- すべてのアイテムの組み合わせを試す
  - 1回の組み合わせに対して価値の計算は$O(n)$時間
  - 組み合わせの総数は2^n通り
  - 計$O(n2^n)$時間→これは遅すぎて無理！
- 素朴な戦略：「価値/重さ」が大きい順に詰めて行く(グリーディ法)
  - 上手く最適解が求まらないような例を作ることができる

例えば　アイテム重さ：(9, 2, 2, 2, 2, 2)
アイテム価値：(14, 3, 3, 3, 3, 3)
ナップサック容量：10

(14/9 = 1.5555...)

動的計画法による解法

- 動的計画法を用いたアルゴリズム
  1. $O(nb)$　(疑似多項式時間アルゴリズム)

nと入力の最大値の多項式で
計算時間を上から抑えられる

例えば $c = (4,\ 1.5^n,\ 32,\ 1.1^n \cdots, 1.33^n)$, $b = 1.8^n$
のような場合は多項式時間では終わらない
....が，現実にはそんな問題を解くことが
あまりないので，実用的には気にしなくて
良いことが多い

アルゴリズムの設計（１）

- 再帰（漸化式）設計の基本
  - より小さい問題(部分問題)へと帰着する
- 最初にやること：何を部分問題とするかを決める
- 今回のアプローチ
  - nアイテム($\{1,2,\cdots n\}$)の問題を$n-1$アイテムの問題($\{1,2,\cdots n-1\}$)に帰着

もとの問題P_n
b, $(w_1, w_2 \cdots w_n)$,
$(c_1, c_2, \cdots c_n)$

⇒

部分問題P_{n-1}
b, $(w_1, w_2 \cdots w_{n-1})$,
$(c_1, c_2, \cdots c_{n-1})$

アルゴリズムの設計（２）

- 部分問題を決めたら．．．
  - 帰着の方法（漸化式）を考える
- 最初に考えてみること
  - 部分問題について何らかの情報が分かっていると仮定して，もとの問題の解を計算する方法を考える

最適解だけでは不十分な
ことが多い

アルゴリズムの設計（３）

- 今回の仮定：P_{n-1}に対し，重みの総和がkで価値の総和が最大の解がすべてのkについて分かっているとする
- $T[i][k]$：部分問題P_iに対する「重みの総和がkの許容解で最大の価値」を記録
  - そのような解がなければ-∞ (解なし，を意味する値)

KPの漸化式：

$$T[i+1][k] = \max\{T[i][k], T[i][k-w_{i+1}] + c_{i+1}\}$$

P_iの重みkの解で
一番価値が高いものに対応

$i+1$番目のアイテムを加えることで
重みkになるもののうち，一番
価値が高いものに対応

アルゴリズム DPKP (表を埋める版)

-∞は「解なし」　0は「アイテム数0個の解」

```
Tの各要素←-∞
T[1][w1] ← c1; T[1][0]←0;
for (i = 2 to n) {
    for (k = 0 to b) {
        if (k < wi) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - wi]+ci)
    }
}
答えを出力(後述)
```

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

表T(空欄は-∞とする)

```
Tの各要素←-∞
T[1][w1] ← c1; T[1][0]←0;
for (i = 2 to n) {
    for (k = 0 to b) {
        if (k < wi) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - wi]+ci)
    }
}
```

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

表T(空欄は-∞とする)

```
Tの各要素←-∞
T[1][w1] ← ci; T[1][0]←0;
for (i = 2 to n) {
    for (k = 0  to b) {
        if (k < wi) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - wi]+ ci)
    }
}
```

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

表T(空欄は-∞とする)

```
Tの各要素←-∞
T[1][w1] ← c1; T[1][0]←0;
for (i = 2 to n) {      (i = 2)
    for (k = 0  to b) {
        if (k < wi) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - wi]+ ci)
    }
}
```

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

表T(空欄は-∞とする)

```
Tの各要素←-∞
T[1][w1] ← c1; T[1][0]←0;
for (i = 2 to n) {      (i = 2)
    for (k = 0  to b) {
        if (k < wi) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - wi]+ ci)
    }
}
```

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

```
Tの各要素←-∞
T[1][w_1] ← c_1; T[1][0]←0;
for (i = 2 to n) {     (i = 2)
    for (k = 0  to b) {
        if (k < w_i) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - w_i]+ c_i)
    }
}
```

表T(空欄は-∞とする)

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

```
Tの各要素←-∞
T[1][w_1] ← c_1; T[1][0]←0;
for (i = 2 to n) {     (i = 3)
    for (k = 0  to b) {
        if (k < w_i) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - w_i]+1)
    }
}
```

表T(空欄は-∞とする)

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

```
Tの各要素←-∞
T[1][w_1] ← c_1; T[1][0]←0;
for (i = 2 to n) {     (i = 3)
    for (k = 0  to b) {
        if (k < w_i) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - w_i]+ c_i)
    }
}
```

表T(空欄は-∞とする)

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

```
Tの各要素←-∞
T[1][w_1] ← c_1; T[1][0]←0;
for (i = 2 to n) {     (i = 3)
    for (k = 0  to b) {
        if (k < w_i) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - w_i]+ c_i)
    }
}
```

表T(空欄は-∞とする)

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

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

```
Tの各要素←-∞
T[1][w_1] ← c_1; T[1][0]←0;
for (i = 2 to n) {     (i = 3)
    for (k = 0  to b) {
        if (k < w_i) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - w_i]+ c_i)
    }
}
```

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4									

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

```
Tの各要素←-∞
T[1][w_1] ← c_1; T[1][0]←0;
for (i = 2 to n) {     (i = 4)
    for (k = 0  to b) {
        if (k < w_i) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - w_i]+ c_i)
    }
}
```

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

DPKPの動作例

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

```
Tの各要素←-∞
T[1][w_1] ← c_1; T[1][0]←0;
for (i = 2 to n) {     (i = 4)
    for (k = 0  to b) {
        if (k < w_i) T[i][k] = T[i-1][k]
        else T[i][k] = max (T[i-1][k], T[i-1][k - w_i]+ c_i)
    }
}
```

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

ここの最大値が答え（の価値総和）

解集合の構成

- 問題例
  - $b = 8$,
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{}

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	$\boxed{16}$

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{}

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

負でないほうをたどる

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{}

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

アイテム4を加えて解を
作った→解に4を加える

負でないほうをたどる

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{4}

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

アイテム3は加えなかった

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{4}

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8					
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

アイテム3は含まない
→解に加えない

アイテム3は加えなかった

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{4}

表T(空欄は-∞とする)

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{4,2}

表T(空欄は-∞とする)

アイテム2を加えて解を
作った→解に2を加える

	0	1					6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{4,2}

表T(空欄は-∞とする)

アイテム2を加えて解を
作った→解に2を加える

	0	1					6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

この解を構成したいとする

解集合の構成

- 問題例
  - $b = 8,$
  - 重さ(1,3,2,4)
  - 価値(3,8,4,5)

{4,2,1}

表T(空欄は-∞

アイテム1を加えて解を
作った→解に1を加える

	0	1	2	3	4	5	6	7	8
1	0	3							
2	0	3		8	11				
3	0	3	4	8	11	12	15		
4	0	3		8	11	12	15	13	16

この解を構成したいとする

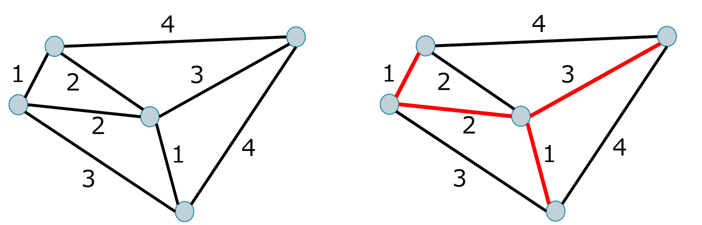
Part3: 貪欲法

貪欲法(greedy法)

- 各時点での最良のものを選択しながら解を構成（グリーディ選択）
  これを繰り返す → 全体として最適になる
  - いつでも使えるわけではない
  - 数学的には，マトロイドという構造を持っている問題に適用可能
    - 闇雲にやってるわけではない

最小全域木問題

- 複数の都市がある長さの線路でつながっている
- いくつかの路線を新幹線に置き換えたい
  - 任意の都市間が新幹線のみで行き来できるようにする
  - 建設コスト（総路線長）はできるだけ短く



最小全域木問題：定義

- 入力：
  - 連結なグラフ$G = (V, E)$と辺への重み付け関数$w: E \to \mathbb{R}^+$

このセットを「重み付きグラフ(weighted graph)」と呼ぶ

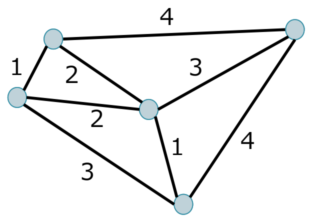
- 出力
  - 以下の値を最小にするGの全域木$T = (V, E')$

$$w(E') = \sum_{e \in E'} w(e)$$

最小全域木問題のコスト関数と呼んだりする

グリーディ法による解法

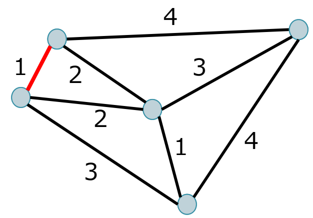
- 直感的には，重みの小さい辺からとっていくのが
  よさそう(同じ重みの辺は好きなほうをとる)
  - ただし，閉路ができてしまうと木にならないので
    そのようなケースは除外する



戦略：
1. すべての辺を重みの小さい順に並べる
2. 小さい重みの辺から順に木の構成辺として加えていく．ただし，加えると閉路ができてしまうときはスキップする
3. 取れる辺がなくなるまで続ける

グリーディ法による解法

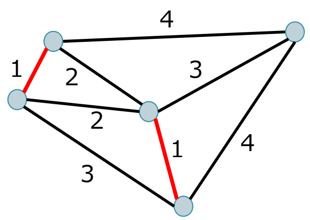
- 直感的には，重みの小さい辺からとっていくのが
  よさそう(同じ重みの辺は好きなほうをとる)
  - ただし，閉路ができてしまうと木にならないので
    そのようなケースは除外する



戦略：
1. すべての辺を重みの小さい順に並べる
2. 小さい重みの辺から順に木の構成辺として加えていく．ただし，加えると閉路ができてしまうときはスキップする
3. 取れる辺がなくなるまで続ける

グリーディ法による解法

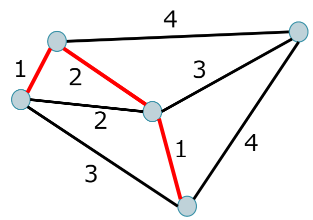
- 直感的には，重みの小さい辺からとっていくのが
  よさそう(同じ重みの辺は好きなほうをとる)
  - ただし，閉路ができてしまうと木にならないので
    そのようなケースは除外する



戦略：
1. すべての辺を重みの小さい順に並べる
2. 小さい重みの辺から順に木の構成辺として加えていく．ただし，加えると閉路ができてしまうときはスキップする
3. 取れる辺がなくなるまで続ける

グリーディ法による解法

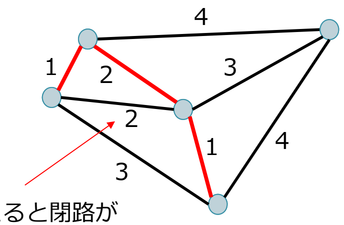
- 直感的には，重みの小さい辺からとっていくのが
  よさそう(同じ重みの辺は好きなほうをとる)
  - ただし，閉路ができてしまうと木にならないので
    そのようなケースは除外する



戦略：
1. すべての辺を重みの小さい順に並べる
2. 小さい重みの辺から順に木の構成辺として加えていく．ただし，加えると閉路ができてしまうときはスキップする
3. 取れる辺がなくなるまで続ける

グリーディ法による解法

- 直感的には，重みの小さい辺からとっていくのがよさそう(同じ重みの辺は好きなほうをとる)
  - ただし，閉路ができてしまうと木にならないのでそのようなケースは除外する



これをとると閉路ができるのでスキップする

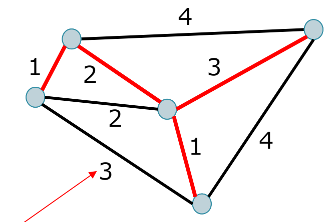
戦略：
1. すべての辺を重みの小さい順に並べる
2. 小さい重みの辺から順に木の構成辺として加えていく．ただし，加えると閉路ができてしまうときはスキップする
3. 取れる辺がなくなるまで続ける

グリーディ法による解法

- 直感的には，重みの小さい辺からとっていくのがよさそう(同じ重みの辺は好きなほうをとる)
  - ただし，閉路ができてしまうと木にならないのでそのようなケースは除外する

完成！

このアルゴリズムは「クラスカルのアルゴリズム(クラスカル法)」と呼ばれる



これをとると閉路ができるのでこちらは選択肢にならない

戦略：
1. すべての辺を重みの小さい順に並べる
2. 小さい重みの辺から順に木の構成辺として加えていく．ただし，加えると閉路ができてしまうときはスキップする
3. 取れる辺がなくなるまで続ける

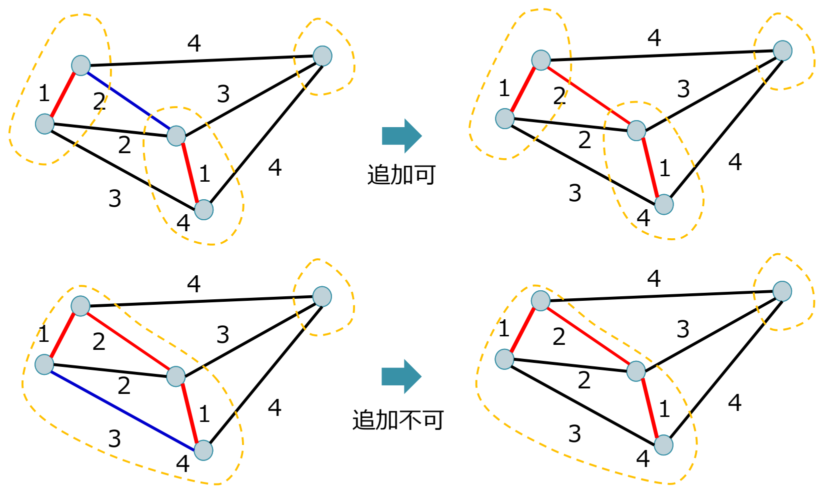
クラスカルのアルゴリズム

```
 入力：G=(V,E={e1, e2, …, em},c)
重みを昇順にソート
  c(e1)≦c(e2)≦…≦c(em) とする
 T=∅；
 for ( i = 1 to m)
     if (Tにeiを加えても閉路ができない)  T = T ∪ {ei}
```

問題：「**ei を加えても閉路ができない**」をどうやって判定？
→ 人間が目で見て確認するのは簡単だが…

連結成分の管理による閉路の判定

- eを追加しても閉路ができない
  =構成中の木で両端点が別の連結成分に属する



union-find データ構造

- 互いに素な集合の集まり(族)を効率的に管理するデータ構造
  - **make-set(v)**: (頂点) vだけからなる集合を作る
  - **find(v)** : vを含む集合(の名前(=代表頂点のID))を返す
  - **union(u,v)**: uを含む集合とvを含む集合の和集合を作る
  - これを用いて「同じ連結成分に属する頂点の集合」を管理する

最小全域木アルゴリズム（詳細）

```
Min-Span-Tree(G=(V,E,c)){
    T = φ ; // 全域木を構成する辺集合
    for (each v ) make-set(v);
    sort(E); // c(e1)≦c(e2)≦…≦c(em)
    for ( i = 1 to m){
      (x,y) = ei ; // eiの端点をx,yとする
      if (find(x) ≠ find(y)) {
           T = T ∪{(x,y)};
           union(x,y);
      }
    }
}
```

「同じ集合(=連結成分)に属する頂点でなければ」
=
「サイクルができなければ」

x,yが属する連結成分をマージする

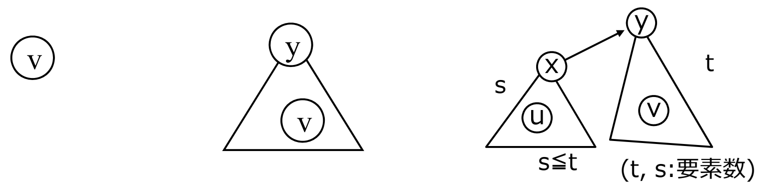
union-find の実現

- 一つの集合=木となるよう内部でデータ管理
  - データ構造の実態は，各要素について，その親のIDを格納する配列parent[]
    - と，集合のサイズを管理する配列size[]

make-set
= 1頂点からなる
木を作る

find(v)
=vから親を辿っていき，
根のIDを返す

union(u, v)
=u,vが属する木の
根をつなげる

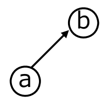


(木の高さを抑えるため，必ずサイズの小さいほうを子側にする)

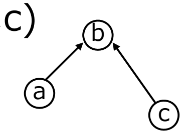
union-find による操作例

頂点集合 V={a, b, c, d, e, f, g} に対する操作例

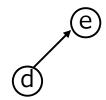
make-set(a) a
make-set(b) b
union(b,a)



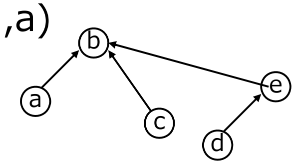
find(a) = b
make-set(c) c
union(a,c)



find(c) = b
make-set(d) d
make-set(e) e
union(e,d)



union(d,a)



union-find の擬似プログラム例

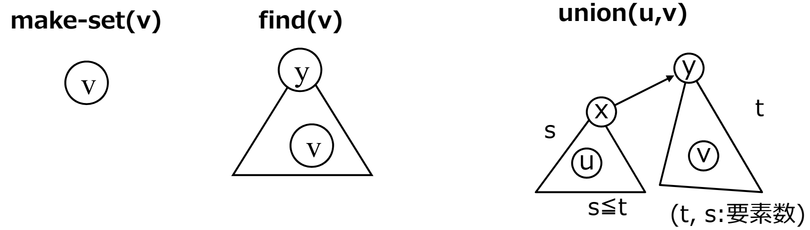
make-set(v): vだけからなる集合を作る
find(v) : vを含む集合(の名前=根の値)を返す
union(u,v): uを含む集合とvを含む集合の和集合を作る(<u>要素数の大きくない方の根（集合）を小さくないほうの根の子（集合の要素）にする</u>)

```
void make-set(i){
  parent[i]=i; size[i]=1;
}

int find(i){
  while (parent[i] != i)
    i = parent[i];
  return i;
}

void union(i,j){
  int p, q ;
  p = find(i); q = find(j);
  if (size[p]>=size[q]){
    parent[q]=p; size[p]=size[p]+size[q];
  } else {
    parent[p]=q; size[q]=size[p]+size[q];
  }
}
```

頂点集合を管理するデータ構造（計算時間）



make-set(v)	find(v)	union(u,v)
O(1)	O(log t) (O(log n)で抑えられる)	O(log s + log t) (O(log n)で抑えられる)

- 実際には，findを実行しながら木の高さを縮めていくことで，もっと高速にできる

最小全域木アルゴリズム（時間）

```
Min-Span-Tree(G=(V,E,c)){
    T = ∅ ; // （全域木を構成する辺集合）
    for (each v ) make-set(v);
    sort(E);// c(e1)≦c(e2)≦…≦c(em)
    for ( i = 1 to m){
      (x,y) = ei  ; // eiの端点をx,yとする
      if (find(x) ≠   find(y)) {
           T = T ∪{(x,y)};
           union(x,y);
      }
    }
}
```

O(n)
O(mlog m)
O(mlog n)

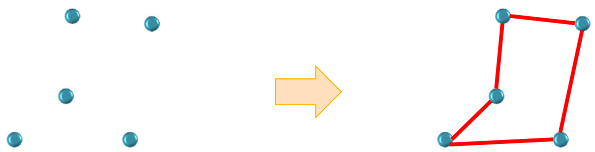
union(x,y)の実行時間
= O(log |x|+log |y|)
≦O(log n) これが m 回実行.

合計 O(mlog m)

Part4 : 近似解法

問題：TSP

- 巡回セールスパーソン問題(Traveling Salesperson Problem)
  - 入力:n頂点の辺重み付き完全(無向)グラフ
  - 出力:全頂点を巡回して（もとに戻ってくる)閉路
  - コスト関数：閉路の長さ
- ユークリッドTSP (Euc-TSP)
  - 平面上(より一般的にはk次元空間）の点を
    最も短い距離で巡回する経路を求める問題



TSP vs KP vs MAXFLOW

- （離散）最適化問題のナイーブな解法
  - 最小全域木問題：すべての木をチェック
  - ナップサック問題：すべての詰め方をチェック
  - ＴＳＰ：全ての巡回路をチェック

いずれも探索には膨大な時間
((超)指数時間）が必要！

- 一方で，最小全域木問題やナップサック問題はより高速な解法があることを既に見てきた
  - 高速=(強/弱)多項式時間

多項式 vs 超多項式

- 多項式：$\Theta(n\log n)$とか$\Theta(n^2)$とか$\Theta(n^{1000})$とか
- 超多項式：$\Theta(\mathrm{n}^{\log n})$とか$\Theta(2^n)$とか$\Theta(n!)$とか

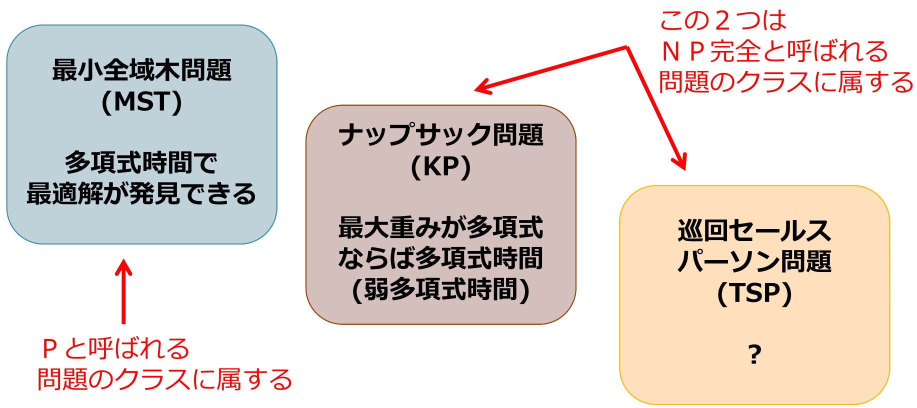
アルゴリズム理論における合意事項

多項式時間で問題を解く→ 実用的なアルゴリズム
超多項式時間で問題を解く→非実用的なアルゴリズム

  - この合意事項に基づく分類が微妙になるような場合もある(e.g., $\Theta(n^{1000})$ vs $\Theta(n^{\log\log\log n})$)

TSP vs KP vs MST

- 3つとも最適化問題の一種
  - 実は難しさ（問題の解きにくさ）が全然違う



TSP vs KP vs MST

注) 「多項式時間」と言ったときは普通は強多項式時間を指す

- 3

ＮＰ完全： 最適解を多項式時間で求めることが無理であると信じられているクラス(正式な定義は別)

Ｐ：多項式時間で最適解を求める方法があるクラス

最小全域木問題 (MST)

多項式時間で最適解が発見できる

Ｐと呼ばれる問題のクラスに属する

ナップサック問題 (KP)

最大重みが多項式ならば多項式時間 (弱多項式時間)

巡回セールスパーソン問題 (TSP)

?

この２つはＮＰ完全と呼ばれる問題のクラスに属する

NP完全問題へのアプローチ

- NP完全な問題はたくさんある
  - 現実的な問題はほとんどそう→ どう対処する？
- 2つのアプローチ
  - 近似解法
    - ベストな解でなく「まあまあ良い」解を現実的な時間内に見つける
  - 厳密解法
    - 最適解をできるだけ効率よく探索
    - それでも時間がかかりすぎるときはあきらめる. (高速にみつかればラッキー)

近似率保証と発見的手法

- 近似解法における２つのスタンス
  - 近似率保証
    - アルゴリズムの出力する解は最適解の○○倍以内
    - 数学的な性能の保証
  - 発見的手法（ヒューリスティック）
    - アルゴリズムの出力する解は経験則的に良い
    - 実験的な性能の保証（運が悪いと非常に悪いかも）

近似率：定義

- あるアルゴリズムが近似率αを達成する
  - 最大化問題の場合
    - 任意の問題例(インスタンス) pとその最適解$opt(p)$に対して, アルゴリズムの返す解のコスト$c(p)$が以下を満たす

$$\alpha \geq \frac{opt(p)}{c(p)}$$

注) $\alpha \leq \frac{c(p)}{opt(p)}$ と定義する向きもある

  - 最小化問題の場合
    - 任意の問題例pとその最適解$opt(p)$に対して, アルゴリズムの返す解のコスト$c(p)$が以下を満たす

$$\alpha \geq \frac{c(p)}{opt(p)}$$

αは定数とは限らない (nに依存することもある)

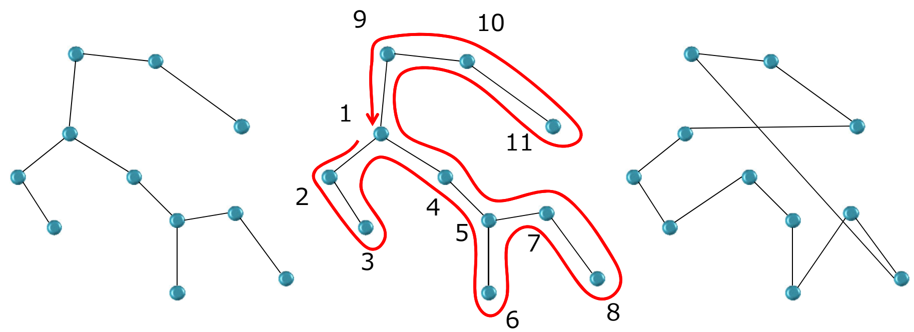
近似アルゴリズムの実際

- TSP
  - 一般の場合は近似率保証のある解を返すことすらNP完全 (「与えられた入力に対してα-近似の解を返せ」という問題自体がおそらく多項式時間解法を持たない)

- Euc-TSPは近似アルゴリズムを作ることができる
  - 2-近似のアルゴリズムを構成できる
    - もっと良いアルゴリズムもある

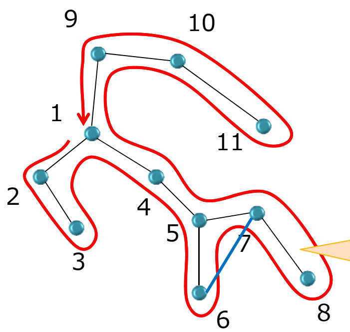
Euc-TSPに対する2-近似アルゴリズム

- 基本的なアイデア
  - 最小全域木Tを張り，Tに沿って巡回していく
    - MSTの構成は$O(n^2 \log n)$時間（＝多項式時間）



2-近似性の証明(1)

- T上の巡回は，各辺を正確に2回通る
  - よって赤線の長さはTのコストc(T)の2倍
- 赤線を元に構成される巡回路は赤線より短い
  - 三角不等式より

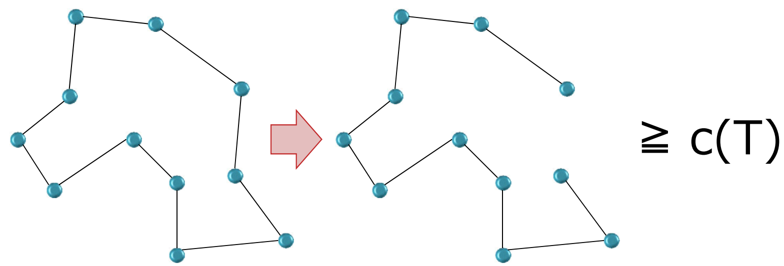


求めた解のコストC≦2・c(T)

赤線に沿って6-5-7と回るよりも
青線で直接移動するほうがコスト低

2-近似性の証明(2)

- 最適な巡回路Optがc(T)以上であることを示す
  - そのとき，C ≦ 2・c(T) ≦ 2Opt となり，2近似性が証明できる
- 最適な巡回路から辺を一本抜くと生成木になる
  - これは必ずc(T)以上（∵TはMSTなので）



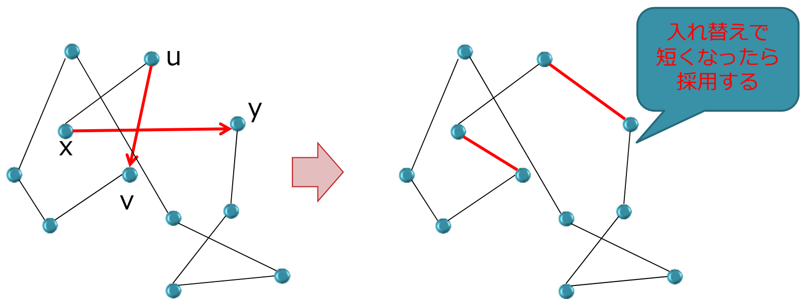
証明からわかること

- 近似アルゴリズムの設計に必要なこと
  - 求めた解を上から押さえる
  - 最適解を下から押さえる
- 最適解が求まらないことを前提としているので「最適解は少なくとも○○以上」という情報が必要

Part5: 発見的手法と厳密解法

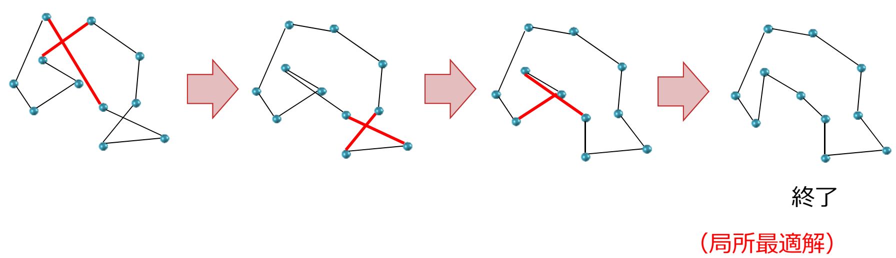
TSPに対する発見的手法

- 2-Opt法
  - 局所探索法(Local search)の一種
  - はじめに一つ許容解を作り，少しづつ改良
    - 2つの辺(u,v), (x,y)を選んで端点を入れ替えてみる
      - (u,y) (x,v) とすると巡回路はつながったまま



TSPに対する発見的手法

- 2-Opt法
  - 入れ替えによる改良を繰り返す
  - もう改良出来なくなったら終了



2-Opt法の計算時間&解の質

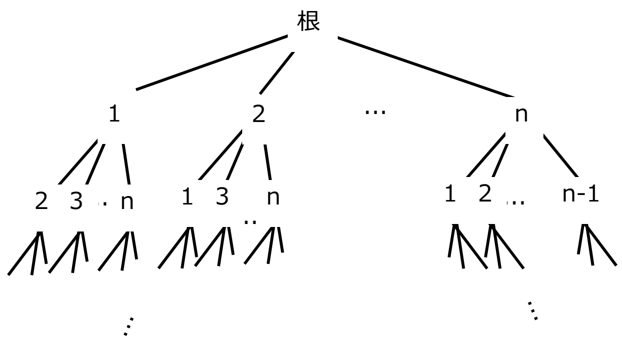
- 計算時間
  - 一回の改良は$O(n^2)$
    - 入れ替え候補が$O(n^2)$個
    - コストの計算は工夫すると$O(1)$時間でできる（差分のみ計算）
    - 解の更新は$O(n)$時間 (1回の改良で1回だけ行う)
  - 改良が指数回起こるような例は存在する
    - 実用的には$O(n^2)$回くらいで終わることが多い
- 最適性，近似率の保証はない
  - 近似率がいくらでも大きくなる入力例がある
  - 実用的には最適解+10～20%ぐらい

分枝限定法

- 全探索の効率を上げる手法の一つ
  - 必ず最適解を出力する
  - 時間はかかる（指数時間かかることもある）
- 基本的な戦略
  - 既に見つけた解よりもよくなる望みがない場合，探索を中止する

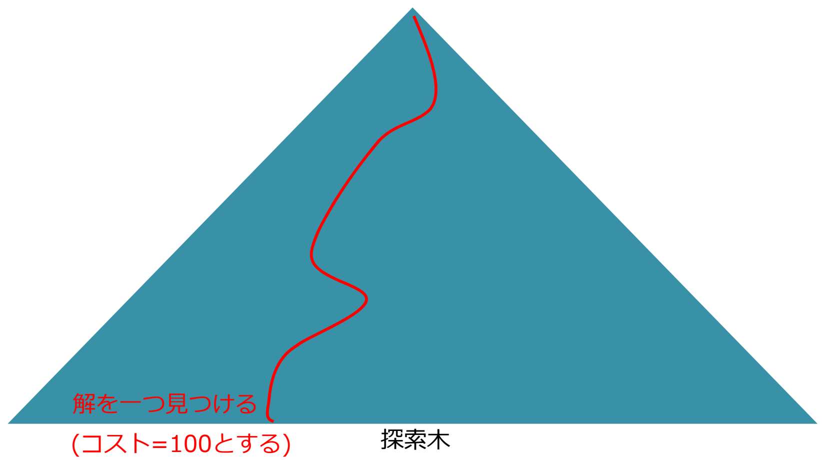
分枝限定法

- 探索木(TSPの例)
  - 可能な解の候補を木状に表現したもの
    - 葉へのパスが一つの許容解に対応する



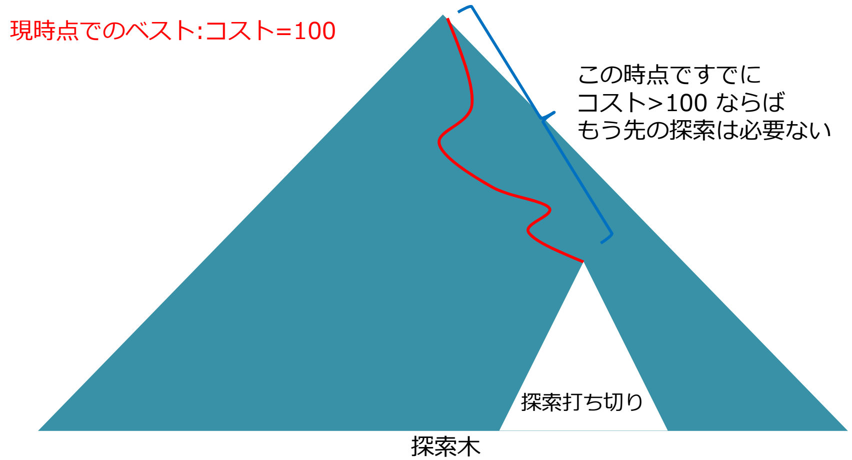
分枝限定法

- アイデア：探索木の深さ優先探索+打ち切り



分枝限定法

- アイデア：探索木の深さ優先探索+打ち切り



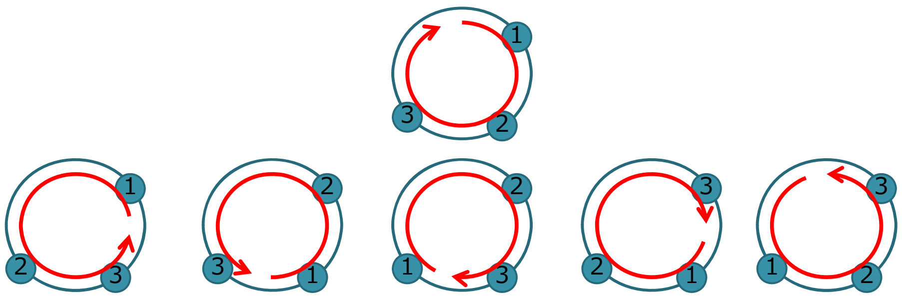
分枝限定法

- アイデアは至極簡単
  - ただし，打ち切りのための条件の設計は工夫必要
    - 単なるコスト比較だけでは速くならないことも多い
    - 木の上のほうで打ち切りできることが重要

  - 探索木の構成そのものにも工夫の余地がある

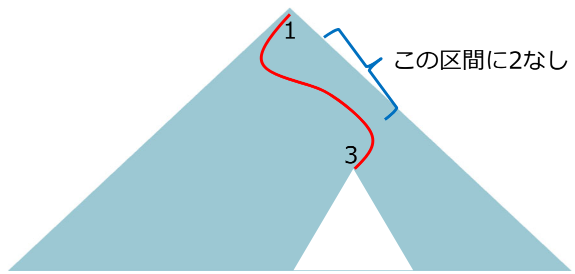
TSP：探索木構成の工夫

- 解が巡回路であることに注意する
  - 任意の3点を選んで，その出現順序を固定できる
    - 例：必ず1→2→3と現れるとできる



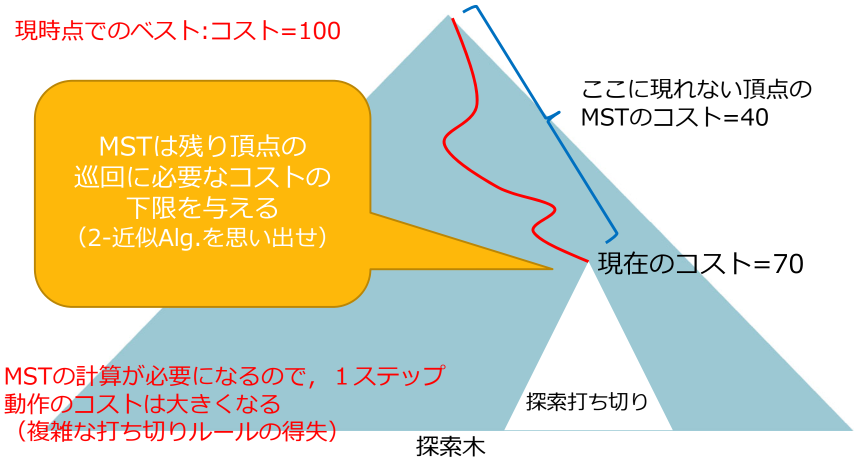
TSP：探索木構成の工夫

- 解が巡回路であることに注意する
  - 任意の3点を選んで，その出現順序を固定できる
    - 例：必ず1→2→3と現れるとできる
  - 1→2→3と現れない枝はそもそも探索木に入れなくてOK



TSP：枝刈の工夫

- 残った頂点のMSTを足す



MR

- ナップサック問題では「価値/重さ」の大きい順で
  取っていくとうまく行かないケースがあることを説明した．

前述の例　アイテム重さ：(9, 2, 2, 2, 2, 2)
アイテム価値：(14, 3, 3, 3, 3, 3)
ナップサック容量：10

(1) この問題例(インスタンス)に対して，授業で説明した
アルゴリズムの実行例(計算完了時のDP表の内容)を書け

(2) 「価値/重さ」の大きい順に取っていったときの解を上述の
インスタンスについて計算し，それが最適解でないことを
確認せよ

付録：クラスカル法の正当性

証明

- アルゴリズムが出力する答えが全域木であることはほぼ明らか
- なぜならば全域木でないとすると，以下のいずれかが成立するが，いずれも矛盾する
  - 閉路がある
    →アルゴリズムの動作に矛盾
  - 連結でない
    → 2つの連結成分をつなぐ辺がまだ取れる
    → アルゴリズムの終了条件に矛盾

よって，証明のメインディッシュは「最小」であることを示すこと

証明

- 背理法による．以下のT, T'について$T \neq T'$を仮定する
  - T:貪欲法により得られる全域木
  - T':最小全域木(複数ある場合，任意の一つ)
- T, T'の構成辺を重さの小さい順に並べる

$$
\begin{array}{llllllll}
T: & e_1, & e_2, & \ldots & e_{i-1}, & e_i, & \ldots & e_{n-2}, & e_{n-1} \\
T': & e'_1, & e'_2, & \ldots & e'_{i-1}, & e'_i, & \ldots & e'_{n-2}, & e'_{n-1}
\end{array}
$$

証明

- 背理法による．以下のT, T'について$T \neq T'$を仮定する
  - T:貪欲法により得られる全域木
  - T':最小全域木(複数ある場合，任意の一つ)
- T, T'の構成辺を重さの小さい順に並べる

$$
\begin{array}{llllllll}
T: & e_1, & e_2, & \ldots & e_{i-1}, & e_i, & \ldots & e_{n-2}, & e_{n-1} \\
T': & e'_1, & e'_2, & \ldots & e'_{i-1}, & e'_i, & \ldots & e'_{n-2}, & e'_{n-1}
\end{array}
$$

  - iを初めて$e_i \neq e'_i$となる辺とする
    ($T \neq T'$なので必ず存在)

証明

- 背理法による．以下のT, T'について$T \neq T'$を仮定する
  - T:貪欲法により得られる全域木
  - T':最小全域木(複数ある場合，任意の一つ)
- T, T'の構成辺を重さの小さい順に並べる

$$
\begin{array}{llllllll}
T: & e_1, & e_2, & \ldots & e_{i-1}, & e_i, & \ldots & e_{n-2}, & e_{n-1} \\
T': & e_1, & e_2, & \ldots & e_{i-1}, & e'_i, & \ldots & e'_{n-2}, & e'_{n-1}
\end{array}
$$

  - iを初めて$e_i \neq e'_i$となる辺とする
    ($T \neq T'$なので必ず存在) →上のように書き換え可能

(★1)

証明

T:貪欲法により得られる全域木
T':最小全域木(複数ある場合，任意の一つ)
i:初めて$e_i \neq e'_i$となる辺

貪欲法で選択される順番

T: $\boxed{e_1,\ e_2,\ \cdots\ e_{i-1},\ e_i,}\ \cdots\ e_{n-2},\ e_{n-1}$

T': $\boxed{e_1,\ e_2,\ \cdots\ e_{i-1},\ e'_i,}\ \cdots\ e'_{n-2}\ ,e'_{n-1}$

これらの辺集合は木の一部分なので，閉路を含まない

- e_1～e_{i-1}を選択した時点において，次の辺の選択がe_i, e'_iどちらであっても閉路は生じない(★2)
  ⇒貪欲法で選ばれたものがより軽いので$w(e_i) < w(e'_i)$
- このとき重みについて以下の関係が成立する

T: $e_1,\ e_2,\ \cdots\ e_{i-1},\ e_i,\ \cdots\ e_{n-2},\ e_{n-1}$

T': $e_1,\ e_2,\ \cdots\ e_{i-1},\ e'_i, < \cdots < e'_{n-2} < e'_{n-1}$

証明

T:貪欲法により得られる全域木
T':最小全域木(複数ある場合，任意の一つ)
i:初めて$e_i \neq e'_i$となる辺

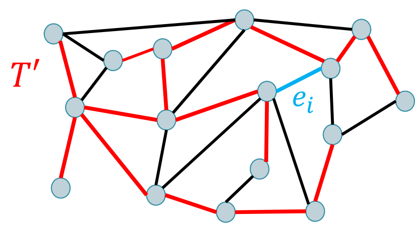
- ここまででわかっていること：まとめ
  - $w(e_i) < w(e'_i) < w(e'_{i+1}) \cdots < w(e'_{n-1})$
  - $\{e_1, e_2, \ldots, e_{i-1}, e_i\}$は閉路を含まない
    すなわち，$\{e'_1, e'_2, \ldots, e'_{i-1}, e_i\}$は閉路を含まない
- また，$e_i \notin T'$であることもわかる
  ($\{e'_1, e'_2, \ldots, e'_{i-1}\} = \{e_1, e_2, \ldots, e_{i-1}\}$はすべて$e_i$と異なり，
  $\{e'_i, e'_{i+1}, \ldots, e'_{n-1}\}$は重さが異なるのですべて$e_i$とは異なる)

証明

- T:貪欲法により得られる全域木
- T':最小全域木(複数ある場合，任意の一つ)
- i:初めて$e_i \neq e'_i$となる辺

1. $w(e_i) < w(e'_i) < w(e'_{i+1}) \cdots < w(e'_{n-1})$
2. $\{e'_1, e'_2, \ldots, e'_{i-1}, e_i\}$は閉路を含まない
3. $e_i \notin T_i$

- いま，T'に辺e_iを追加してみる

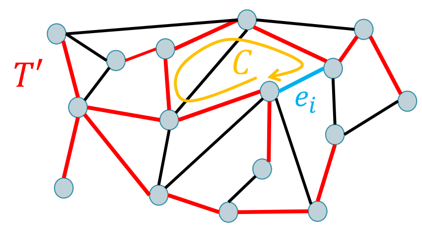


証明

- T:貪欲法により得られる全域木
- T':最小全域木(複数ある場合，任意の一つ)
- i:初めて$e_i \neq e'_i$となる辺

1. $w(e_i) < w(e'_i) < w(e'_{i+1}) \cdots < w(e'_{n-1})$
2. $\{e'_1, e'_2, \ldots, e'_{i-1}, e_i\}$は閉路を含まない
3. $e_i \notin T_i$

- いま，T'に辺e_iを追加してみる
  →定理3-1(5)より，閉路が一つできる(Cとする)

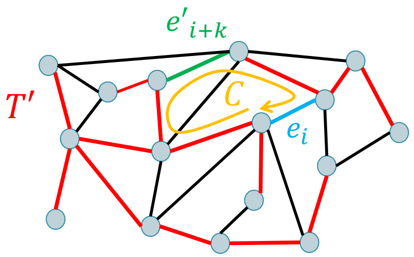


証明

- T:貪欲法により得られる全域木
- T':最小全域木(複数ある場合，任意の一つ)
- i:初めて$e_i \neq e'_i$となる辺

1. $w(e_i) < w(e_i') < w(e'_{i+1}) \cdots < w(e'_{n-1})$
2. $\{e_1', e'_2, \ldots, e_{i-1}', e_i\}$は閉路を含まない
3. $e_i \notin T_i$

- いま，T'に辺e_iを追加してみる
  →定理3-1(5)より，閉路が一つできる(Cとする)
  →$e_1', e_2', \ldots e_{i-1}', e_i$だけでは閉路は構成できないので，
  Cはあるe'_{i+k} $(k \geq 0)$を含む



証明

- T:貪欲法により得られる全域木
- T':最小全域木(複数ある場合，任意の一つ)
- i:初めて$e_i \neq e'_i$となる辺

1. $w(e_i) < w(e_i') < w(e'_{i+1}) \cdots < w(e'_{n-1})$
2. $\{e_1', e'_2, \ldots, e_{i-1}', e_i\}$は閉路を含まない
3. $e_i \notin T_i$

- →$T' - e_{i+k}' + e_i$は全域木になる
  →しかも$w(e_i) < w(e_{i+k}')$なので，最小全域木T'よりも
  軽い全域木が見つかったことになる
  →最小性に矛盾！