# 矩形を用いた1次元と2次元の空間的近接パターン発見

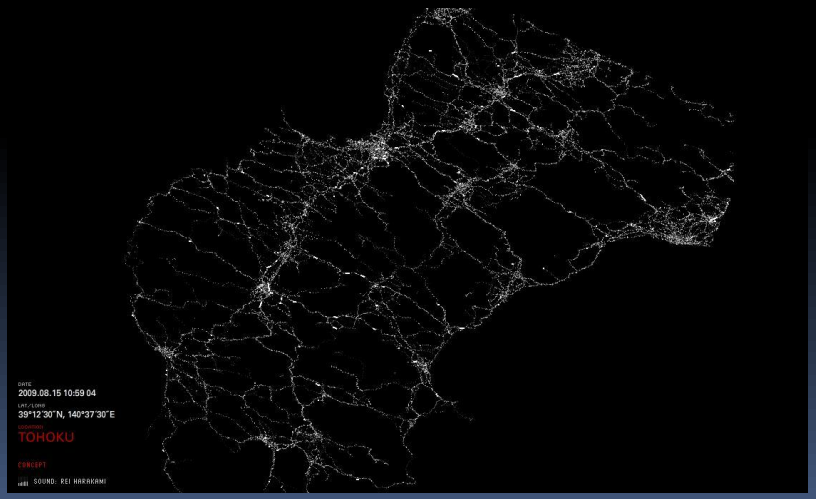
Efficient Discovery of 1-Dim and 2-Dim Spatial Proximity Patterns Using Axis-parallel Rectangles

北海道大学工学部　情報エレクトロニクス学科
コンピュータサイエンスコース
情報知識ネットワーク研究室 4年
関根　渓

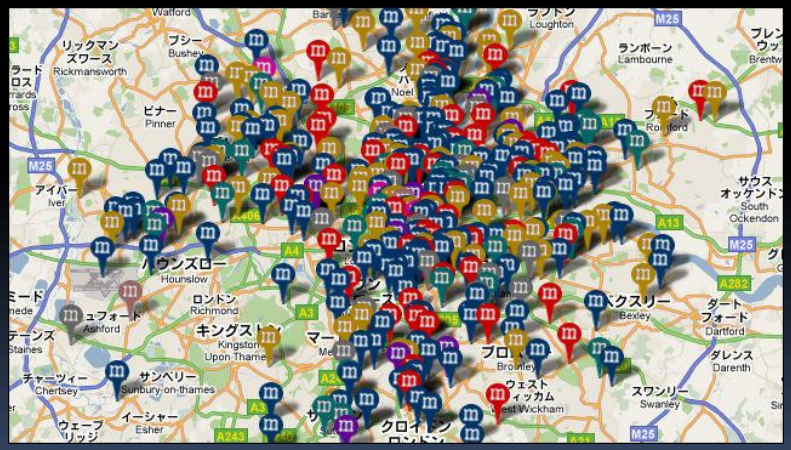
2012.2.10

# はじめに

- 近年，移動体センサーやストリーム技術の発展により，大規模な位置イベントデータから特徴的な規則性やパターンを発見するための効率のよい技術が望まれている



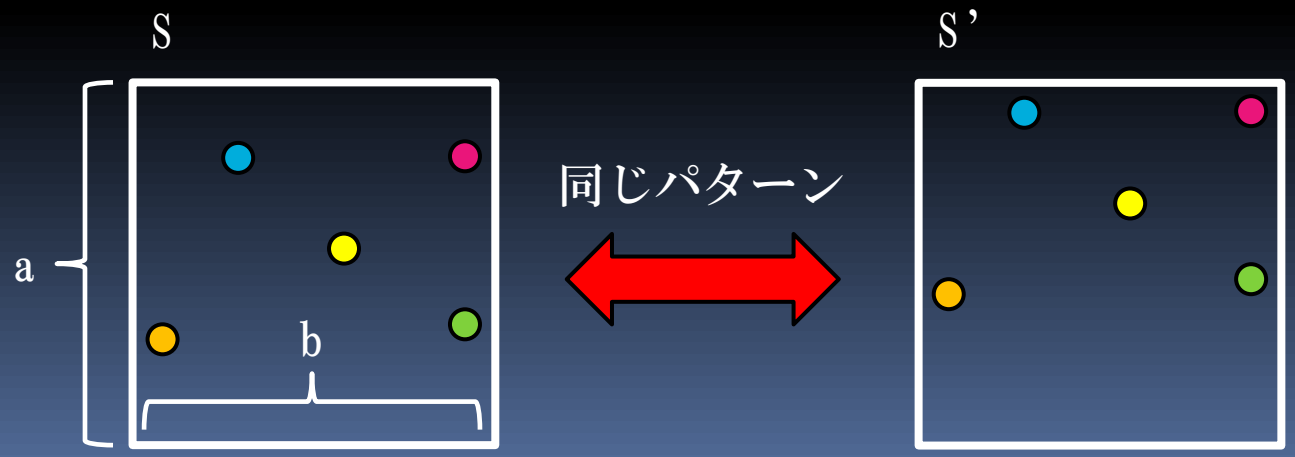

NEURON ROAD 本田技研工業株式会社




Google Maps Google Inc

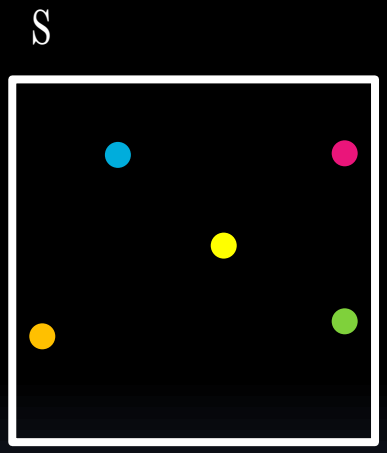
# 矩形を用いた近接パターン

- 近接パターンとは
  - d次元空間上の入力点集合 $P_d = \{p_0, \cdots, p_n\}$
  - 正整数 $c>0$ に対して，色の集合 $C=\{c_0, c_1, \cdots\}$
  - 点への色付けを表す関数 $color:P \rightarrow C$
  - 矩形R(二次元の場合はa×bの長方形)
  - 矩形Rが切り取る点集合Pの任意の部分集合を，PとRの共通部分集合 $S=P \cap R$ と定義し，Sを近接パターンと呼ぶ



# 矩形を用いた近接パターン

- 制約の例：k-多色
  - 非負整数 k>0 に対して，点集合パターンSがk個以上の異なる色を含む

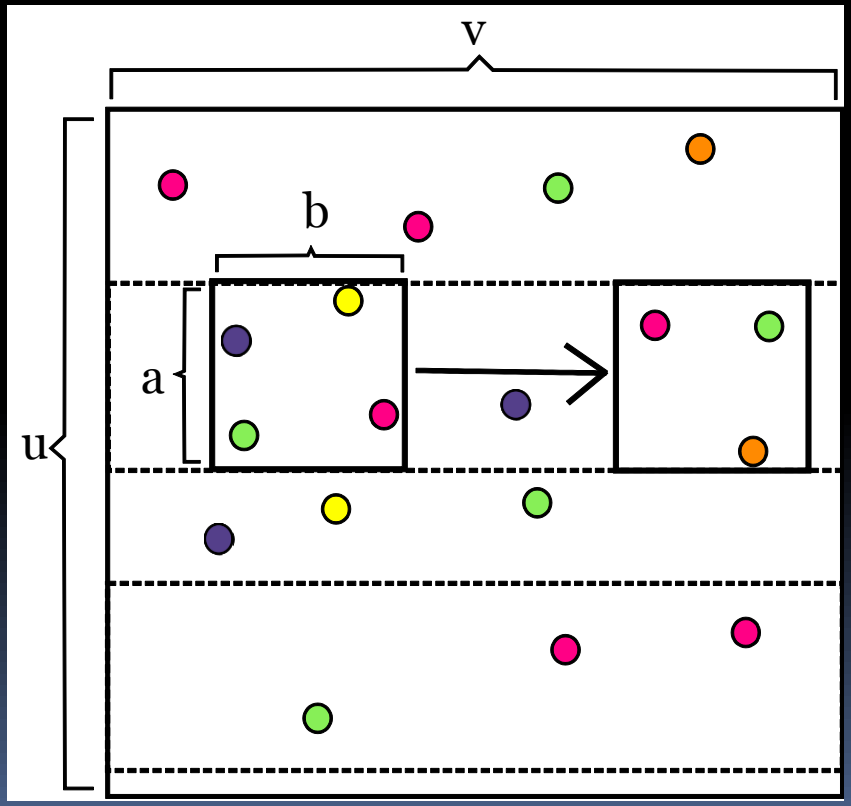



Sはk≤5なら出力される

  - 前処理として，全ての近接パターンを列挙しておけば，効率よく連続して様々な特徴的近接パターンを見つけることができる

# 矩形による全近接パターン列挙問題

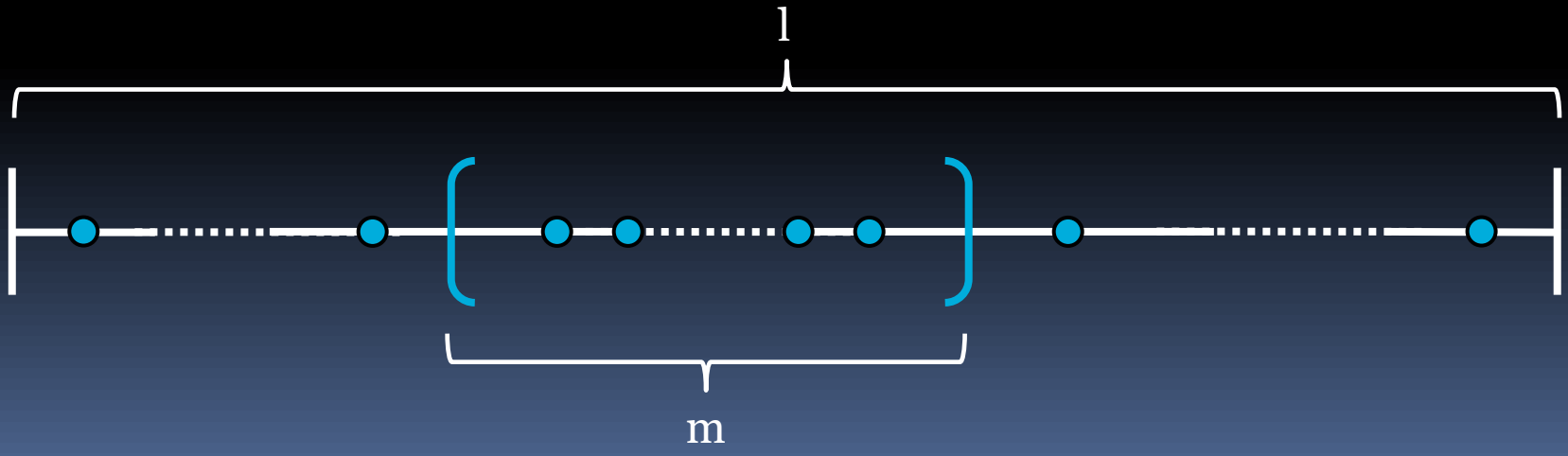
- 二次元空間における問題




- 点集合をy座標に関して昇順にソートして点列Yを作る
- Yを参照しながら，二次元空間上で，a×vの水平な帯(以下ではこれをストライプと呼ぶ)を上から下へ移動させながら，その内部の点をx座標の昇順でソートして点列XSを作る
- 各時点ごとにストライプの内部で，サイズa×bの矩形を動かして，近接パターンを計算する

# 矩形による全近接パターン列挙問題

- 一次元空間における問題
  - 入力：正実数 $m, l>0$ において 矩形の幅 $m$，空間の幅$l$，1次元空間上の入力点集合， $P_1 = \{p_0, \cdots, p_n\}$
  - 出力： $P_1$と任意の$m$-区間の全近接パターン $P_1 \cap [x, x+m]\ (0 \leqq x \leqq l-m)$ を重複なく出力せよ



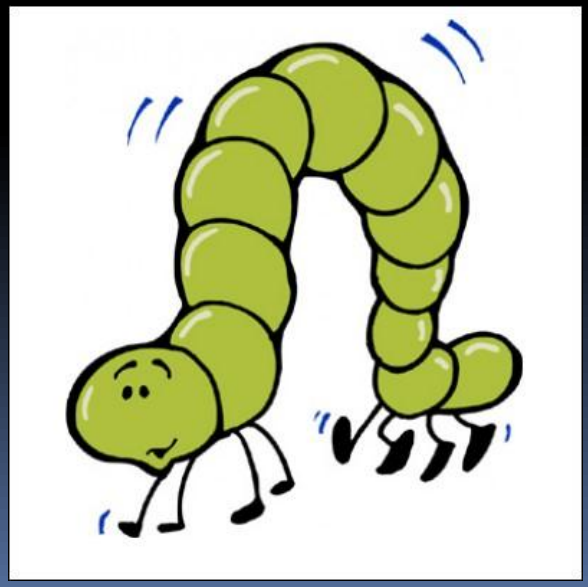
# 本研究の主結果

- 近接パターン発見の基本となる，一次元および二次元の矩形と，一次元，二次元空間上の点集合との全ての近接パターンを列挙するアルゴリズムを考察した

- 一次元空間における全近接パターンを線形時間で列挙アルゴリズム(ε-尺取虫法)を提案，実装し，実験によりその挙動を確認した

# ε-尺取虫法(ε-Inchworm-Method)

- 一次元空間における全近接パターン列挙アルゴリズム
- 入力点数をnとすると，O(n)時間で計算可能
- 極小数εの導入により，実数座標に対応



# ε –尺取虫法( ε –Inchworm–Method)

- アルゴリズムの動作

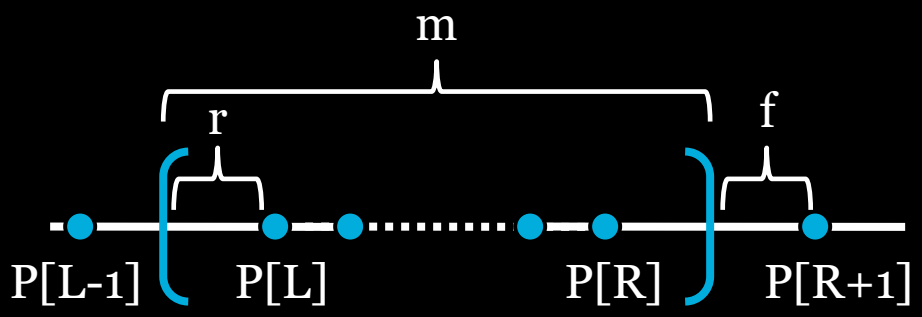
```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:         PrintWindow(P, L, R)
10:     else if r = f then
11:          矩形に点を1つ入れる
12:          r, fの値を更新する
13:          PrintWindow(P, L, R)
14:    else if r < f then
15:         if (R = L and P [R + 1] – P [L] > m then
16:              矩形に点を1つ入れて，1つ出す
17:              r, fの値を更新する
18:              PrintWindow(P, L, R)
19:         else
20:              eps←(f-r)/2
21:              矩形から点を1つ出す
22:              r, fの値を更新する
23:             PrintWindow(P, L, R)
24:         end if
25:     end if
26:  end while
```

# ε -尺取虫法( ε -Inchworm-Method)

- アルゴリズムの動作

```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:         PrintWindow(P, L, R)
10:     else if r = f then
11:         矩形に点を1つ入れる
12:         r, fの値を更新する
13:         PrintWindow(P, L, R)
14:    else if r < f then
15:        if (R = L and P [R + 1] – P [L] > m then
16:            矩形に点を1つ入れて，1つ出す
17:            r, fの値を更新する
18:            PrintWindow(P, L, R)
19:        else
20:            eps←(f-r)/2
21:            矩形から点を1つ出す
22:            r, fの値を更新する
23:            PrintWindow(P, L, R)
24:        end if
25:    end if
26: end while
```

L:矩形内の最左点の添字
R:矩形内の最右点の添字
r:矩形の左端とP[L]の距離
f:矩形の右端と次にP[R+1]の距離

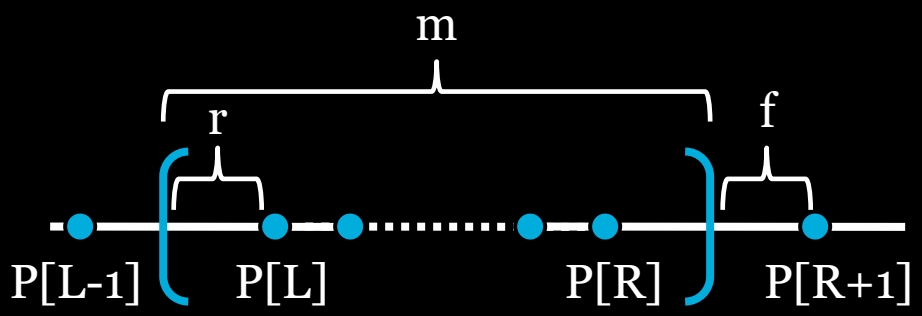


# ε –尺取虫法( ε –Inchworm–Method)

- アルゴリズムの動作

```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:          PrintWindow(P, L, R)
10:     else if r = f then
11:         矩形に点を1つ入れる
12:         r, fの値を更新する
13:         PrintWindow(P, L, R)
14:     else if r < f then
15:         if (R = L and P [R + 1] – P [L] > m then
16:             矩形に点を1つ入れて，1つ出す
17:             r, fの値を更新する
18:             PrintWindow(P, L, R)
19:         else
20:             eps←(f-r)/2
21:             矩形から点を1つ出す
22:             r, fの値を更新する
23:             PrintWindow(P, L, R)
24:         end if
25:     end if
26:  end while
```

手続きSetWindow
空間における矩形の初期位置をO(n)で決定する

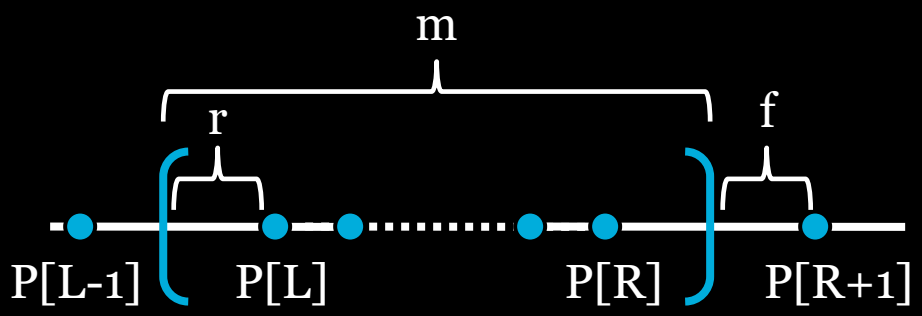


# ε -尺取虫法( ε -Inchworm-Method)

- アルゴリズムの動作

```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:          PrintWindow(P, L, R)
10:     else if r = f then
11:         矩形に点を1つ入れる
12:         r, fの値を更新する
13:         PrintWindow(P, L, R)
14:     else if r < f then
15:         if (R = L and P [R + 1] – P [L] > m then
16:             矩形に点を1つ入れて，1つ出す
17:             r, fの値を更新する
18:             PrintWindow(P, L, R)
19:         else
20:             eps←(f-r)/2
21:             矩形から点を1つ出す
22:             r, fの値を更新する
23:             PrintWindow(P, L, R)
24:         end if
25:     end if
26:  end while
```

手続きPrintWindow
矩形内の点集合を出力する関数

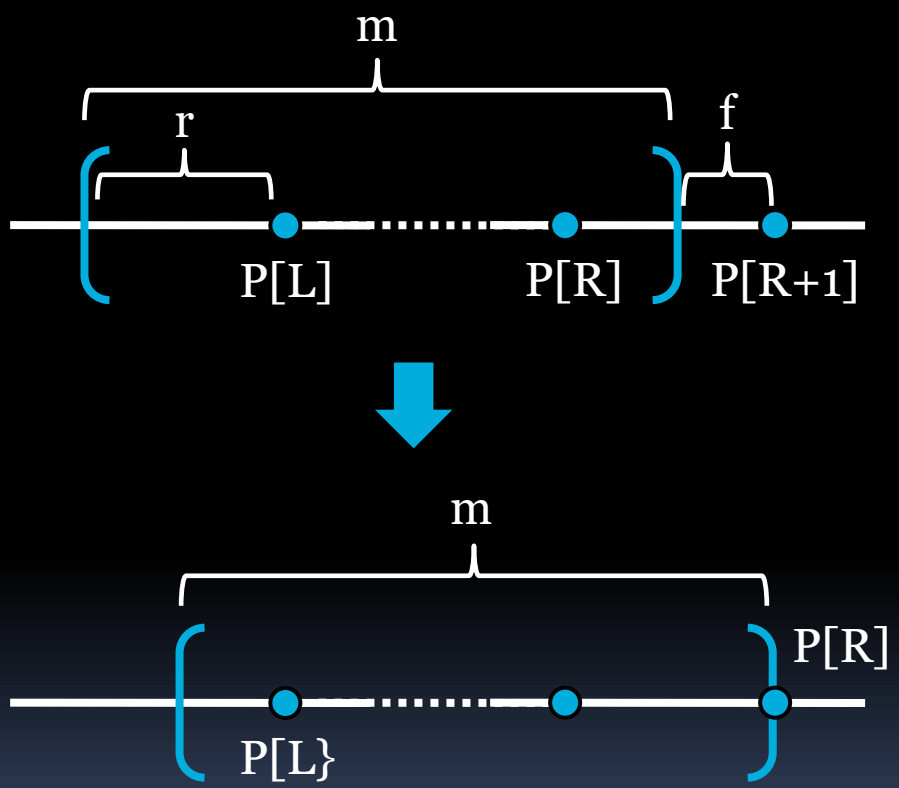


# ε -尺取虫法( ε -Inchworm-Method)

- アルゴリズムの動作

```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:          PrintWindow(P, L, R)
10:     else if r = f then
11:         矩形に点を1つ入れる
12:         r, fの値を更新する
13:         PrintWindow(P, L, R)
14:     else if r < f then
15:         if (R = L and P [R + 1] – P [L] > m then
16:             矩形に点を1つ入れて，1つ出す
17:             r, fの値を更新する
18:             PrintWindow(P, L, R)
19:         else
20:             eps←(f-r)/2
21:             矩形から点を1つ出す
22:             r, fの値を更新する
23:             PrintWindow(P, L, R)
24:         end if
25:     end if
26:  end while
```

(1) r>fのとき

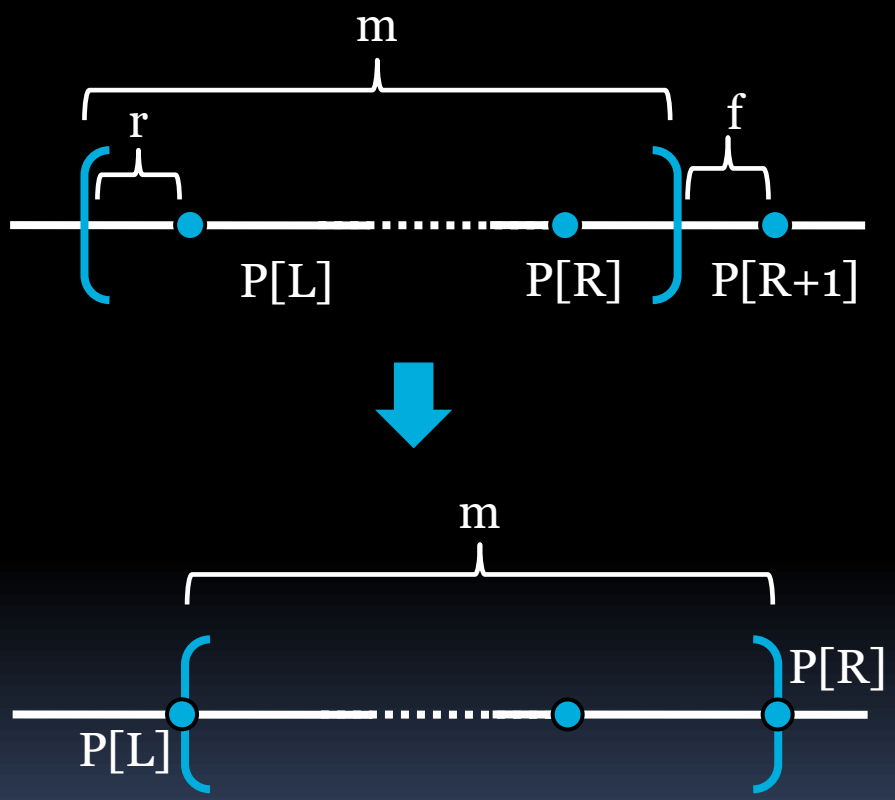


# ε -尺取虫法( ε -Inchworm-Method)

- アルゴリズムの動作

```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:          PrintWindow(P, L, R)
10:     else if r = f then
11:         矩形に点を1つ入れる
12:         r, fの値を更新する
13:         PrintWindow(P, L, R)
14:     else if r < f then
15:         if (R = L and P [R + 1] – P [L] > m then
16:             矩形に点を1つ入れて，1つ出す
17:             r, fの値を更新する
18:             PrintWindow(P, L, R)
19:         else
20:             eps←(f-r)/2
21:             矩形から点を1つ出す
22:             r, fの値を更新する
23:             PrintWindow(P, L, R)
24:         end if
25:     end if
26:  end while
```

(2) r=fのとき

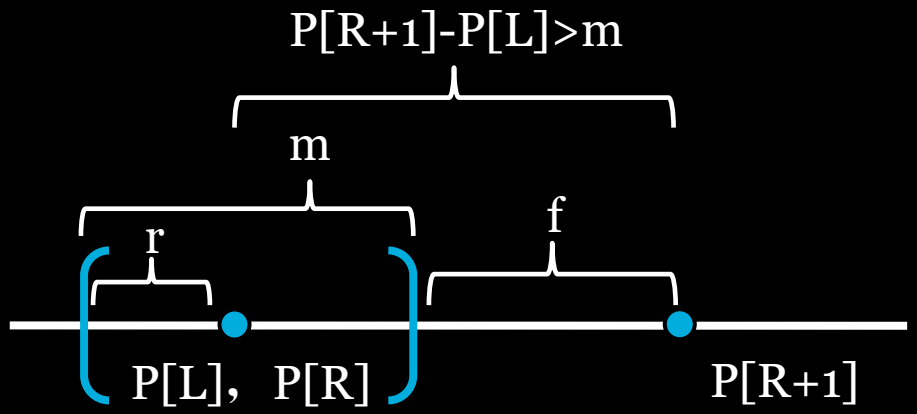


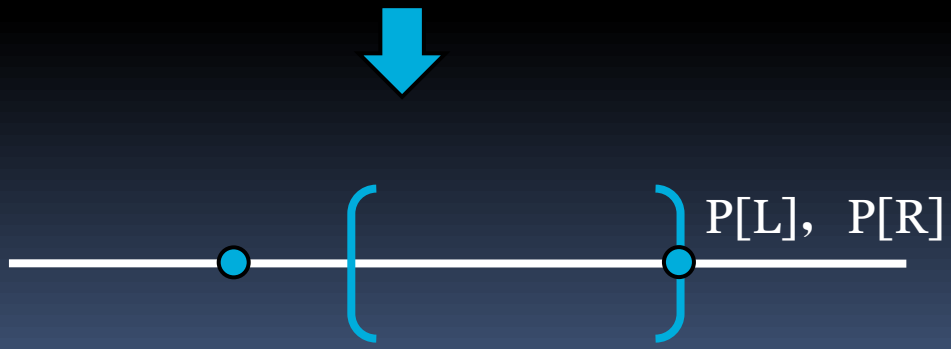
# ε -尺取虫法( ε -Inchworm-Method)

- アルゴリズムの動作

```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:          PrintWindow(P, L, R)
10:     else if r = f then
11:         矩形に点を1つ入れる
12:         r, fの値を更新する
13:         PrintWindow(P, L, R)
14:     else if r < f then
15:         if (R = L and P [R + 1] – P [L] > m then
16:             矩形に点を1つ入れて，1つ出す
17:             r, fの値を更新する
18:             PrintWindow(P, L, R)
19:         else
20:             eps←(f-r)/2
21:             矩形から点を1つ出す
22:             r, fの値を更新する
23:             PrintWindow(P, L, R)
24:         end if
25:     end if
26: end while
```

(3) r<f かつR=L かつ P[R+1]-P[L]>m



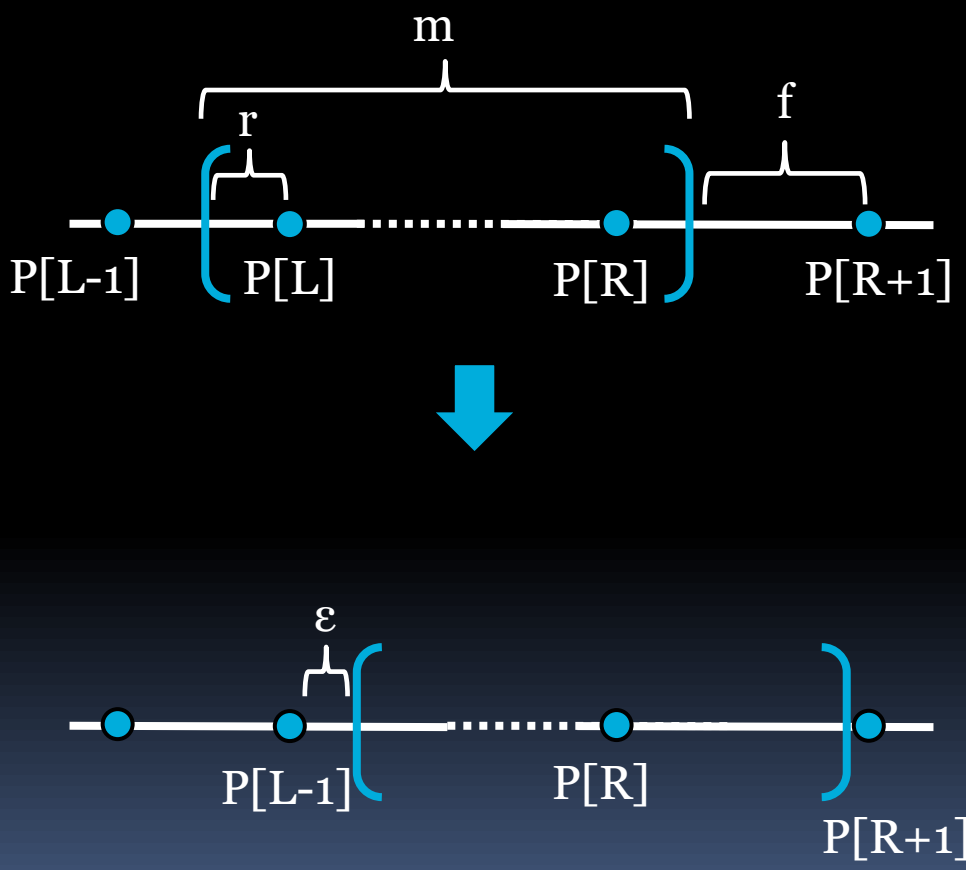



# ε -尺取虫法( ε -Inchworm-Method)

- アルゴリズムの動作

```
手続き ε-Inchworm-Method(l, n, m, P )
1:  R←0 L←0
2:  r←0 f ←0
3:  SetWindow(n, m, L, R, r, f, P)
4:  PrintWindow(P, L, R)
5:  while 矩形の右端が終端に到達していない do
6:      if r > f then
7:          矩形に点を1つ入れる
8:          r, fの値を更新する
9:          PrintWindow(P, L, R)
10:     else if r = f then
11:         矩形に点を1つ入れる
12:         r, fの値を更新する
13:         PrintWindow(P, L, R)
14:     else if r < f then
15:         if (R = L and P [R + 1] − P [L] > m then
16:             矩形に点を1つ入れて，1つ出す
17:             r, fの値を更新する
18:             PrintWindow(P, L, R)
19:         else
20:             eps←(f-r)/2
21:             矩形から点を1つ出す
22:             r, fの値を更新する
23:             PrintWindow(P, L, R)
24:         end if
25:     end if
26: end while
```

(4) r<f かつ(3)以外の場合



# ε-尺取虫法(ε-Inchworm-Method)

- 計算時間
  - 出力は定数時間で行えると仮定する
  - 手続きSetWindowに O(n)時間
  - 全近接パターン数をhとすると，パターン計算には全体で O(h)時間
  - アルゴリズム全体の計算時間はO(n+h)

- 近接パターン数
  - 変数L,Rはアルゴリズムの構造上減少しない
  - L≦n，R≦nである

    → L,R共に最大n回しか増加しない

    → h=O(n)

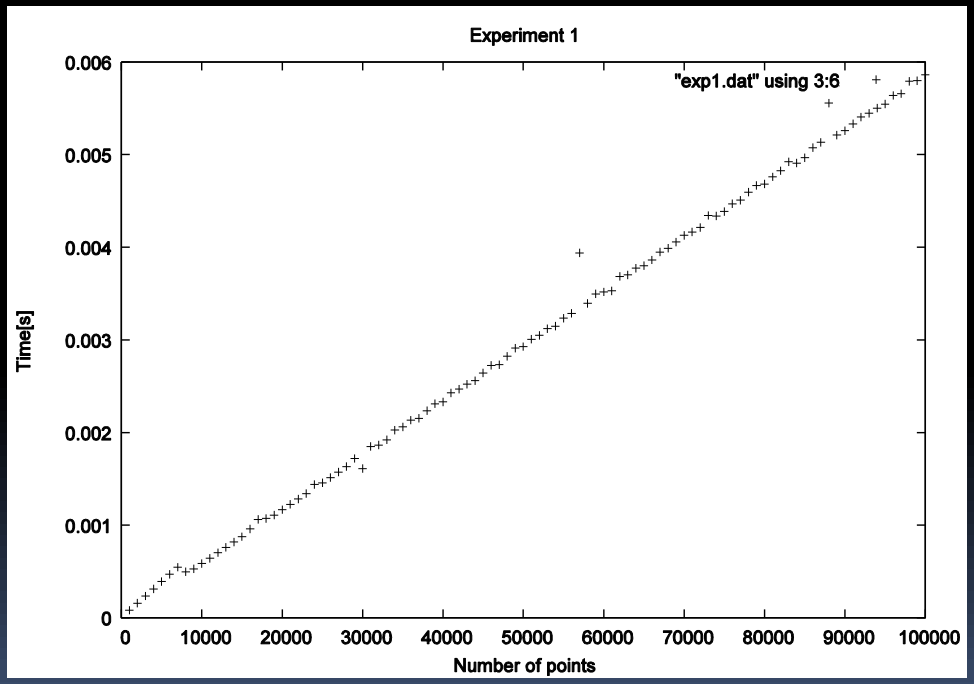
    → アルゴリズム全体の計算時間はO(n)

# 実験

- 実験手法
  - データ
    - 乱数によって生成した重複無しのソート済み整数点集合
  - 実験方法
    - テキストデータとパラメータをプログラムに読み込ませる
    - 手続き SetWindow の直前で時刻$t_1$を計測
    - 全近接パターンの計算が終了した時点で時刻$t_2$を計測
    - 計算時間 T= $t_2$- $t_1$を記録

# 実験

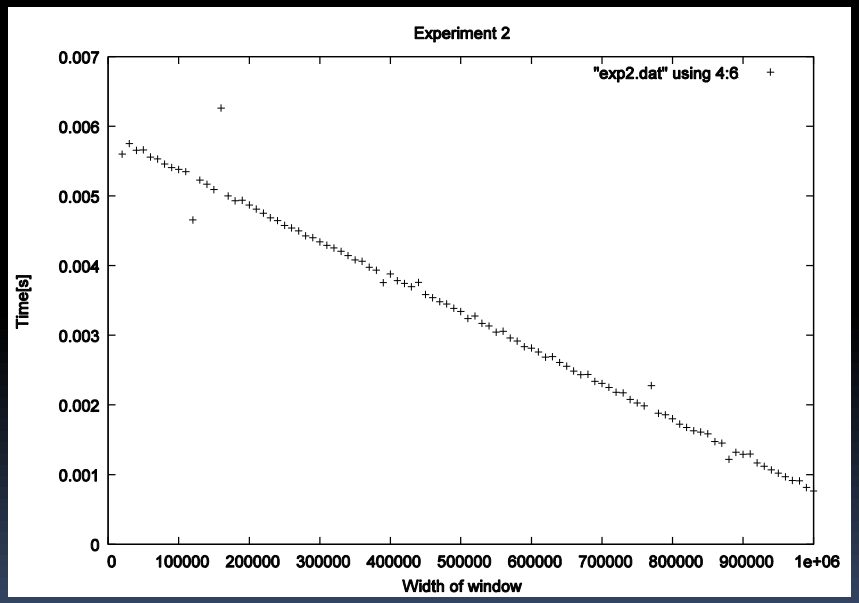
- 実験１：入力データに対する計算時間
  - 矩形の幅 mを固定し，空間の幅 lとデータ数 nを同じ割合で増加させた
  - 入力データが増加すると，解の個数も増えていくので，計算時間も増加していくことが予想される

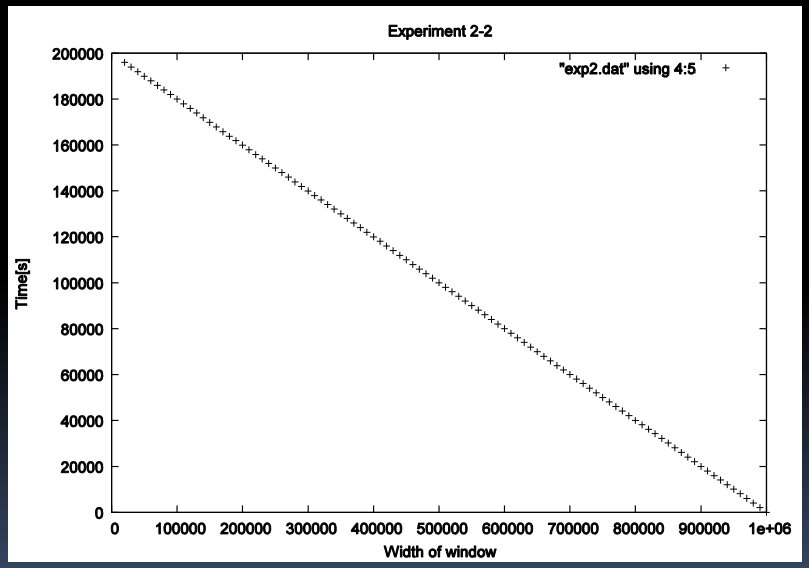



結果より，予想が正しいことが確認できた

# 実験

- 実験２：ウィンドウ幅と計算時間
  - 空間の幅lとデータ数nを固定して，矩形の幅mを増加させた
  - 矩形の幅が大きくなるほど，空間上で矩形を動かせる範囲が制限されていくので，出力されるパターン数が少なくなり，またそれに伴って計算時間も小さくなっていくことが予想される

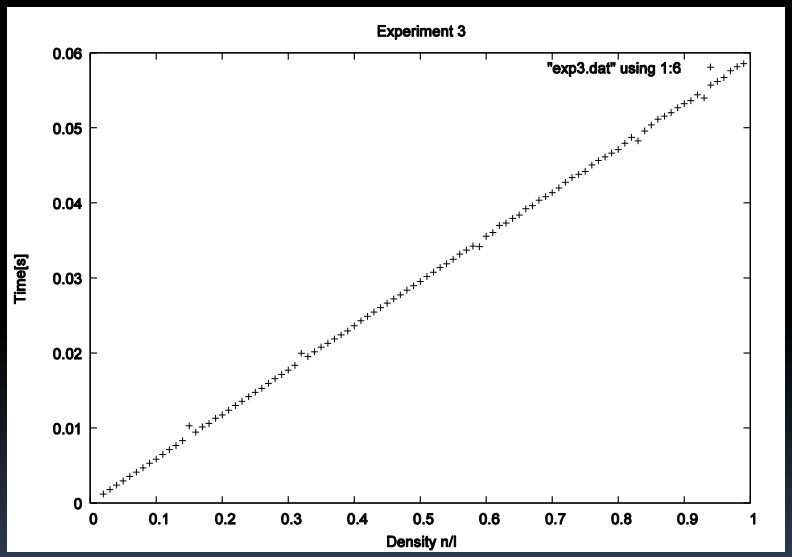


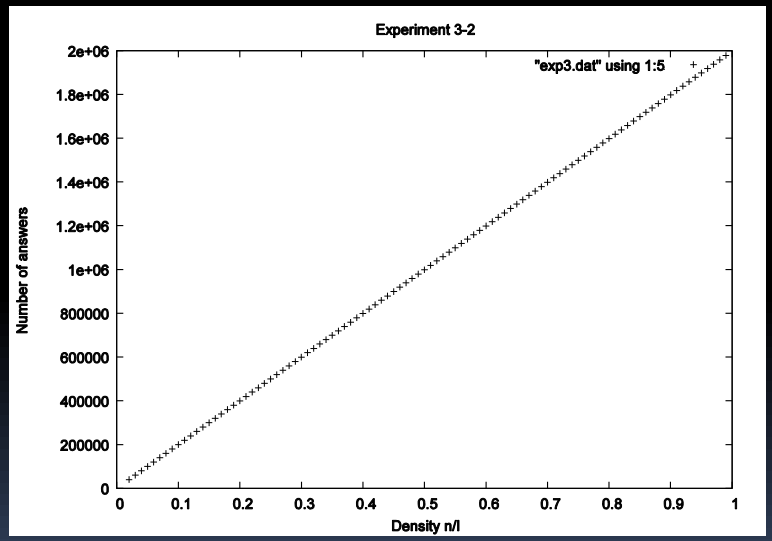




結果より，その予想が正しいことが確認できた

# 実験

- 実験３：点の密度と計算時間
  - 空間の幅lとウィンドウ幅mを固定し，データ数nを増加させた
  - 空間内の点の密度が高くなるにつれて，出力する解の個数が増加すること，またそれに伴って計算時間が増加することが予想される




結果より，その予想が概ね正しいことが確認できた

# おわりに

- 一次元空間におけるアルゴリズムに関しては，実験を行うことで，予め想定していたアルゴリズムの挙動が正しかったことが確認できた
- ２次元空間におけるアルゴリズムに関しては，アイディアは提案できたものの，実装には至らなかった

# 今後の展望

- 実験結果の不明瞭な点を明らかにする
- ナイーブなアルゴリズムを用いて比較実験を行う
- 二次元空間のアルゴリズムを実装し，地理データ等を用いて実験，評価を行う