動的空間データ構造への応用を目的とした
複体の描画点数低減アルゴリズムの改良

島根大学 総合理工学部 数理･情報システム学科

計算機科学講座 田中研究室

S093105 忍足 泰暁

目次

1. 序論

地図上の行政区画を表す地理情報は、平面を多角形に分割した複体[1]によって記述されている。

先行研究で坂口[2]は、行政区画境界線データ[3]に対して、研究の過程で隣接行列表現[2]を用いた複体に対する描画点数低減アルゴリズムの考案を行い、C 言語にてプログラムを実装した。

このアルゴリズムを、システムへの応用の観点から機種依存性の低いプログラミング言語である Java 言語へ移行することにより描画点数低減アルゴリズムの実用性を向上させることが期待出来る。

また、描画点を減らすことでリアルタイムでの処理を行う事を目的とした動的空間データ構造への応用時に、省メモリ化によるデータの取得・更新・挿入・削除を行う際の高速化を図ることも期待出来る。

最終的に坂口[2]は地図グラフ表現の標準的データ構造を利用した方法を用いることで解決したが、隣接行列表現を用いたプログラムには研究の過程で 3 つの問題が存在していた。それは、隣接行列のサイズが大きすぎることからメモリの確保が行えないこと(メモリ確保の問題)・プログラムの実行に時間が掛かること(実行時間の問題)・隣接行列は面についての情報を記憶させることが出来ないこと(面情報の記憶が出来ない問題)である。

これに対して、メモリ確保の問題・実行時間の問題にはスパース行列(圧縮行格納方式)を用い、面情報記憶が出来ない問題にはノード[4]を間引かない・面情報が記憶されていた間引き前のデータに沿ってデータを書き出すことで対処出来ると考えられる。

本研究では、プログラムの C 言語から Java 言語への移行を行なっている際に生じた、プログラムが抱える問題の解決に取り組んだ。

2. 先行研究調査

CiNii と JDreamⅡにて、先行研究の調査を行った。2013 年 01 月 27 日時点での調査結果を表 1 に示す。

検索結果の件数が少ないものについては、タイトルと抄録を確認することで調査した。件数が多いものについては、まずタイトルを確認し、関連のありそうなタイトルについて抄録を確認することで調査を行った。

表 1　CiNii・JDreamⅡでの先行研究調査結果

検索キーワード	件数	
	CiNii	JDreamⅡ
複体 AND（点 OR 描画点）	12	227
複体 AND 描画点	0	0
グラフ AND 描画点	1	5
描画点	5	12
グラフ AND 複体 AND 点 AND 低減	0	0
グラフ AND 点 AND 低減	64	3685
複体 AND 点 AND 低減	0	1

※CiNii では AND は空白文字で表す。

調査した結果、本研究と同様の研究は存在しなかった。従って、本研究は少なくとも日本では新規性が高いものであると考えられる。

3. 描画点数低減アルゴリズムの実装

この章では、描画点数低減アルゴリズムに必要なデータ・理論について述べる。また、1章で述べたメモリ確保の問題・面情報の記憶が出来ない問題への対処についても述べる。

3.1. 境界線データの変換

まず、GML形式の行政区画境界線データ[3]を基盤地図ビューア・コンバーター[3]にてシェープファイル形式へ変換を行う。

そしてこのシェープファイルに対して、点と点の隣接情報を隣接行列へと登録することで変換を行い、描画点数低減アルゴリズムを実行した。

描画点数の低減前と低減後のシェープファイルは、ビューアである ShapeViewer[5]を用いて再生し、描画点が間引かれているかどうかの確認を行った。

3.1.1. シェープファイル

シェープファイルは、ESRI 社が仕様を定めたファイルフォーマットであり、地理情報を扱う際のデータとして世界的に使われている。

シェープファイルは、メイン・ファイル(拡張子：shp)、インデックス・ファイル(拡張子：shx)、属性ファイル(拡張子：dbf)の3つのファイルから構成される[6]。

シェープファイルは、点・線・面のデータを扱うことが出来る[6]ようになっており、シェープ・タイプという形で種類毎に管理方法が異なる。今回は、その中で Polygon と呼ばれるシェープ・タイプと、PolyLine と呼ばれるシェープ・タイプを用いた。

3.1.1.1. シェープ・タイプ：Polygon

Polygon は、面を扱うデータである。今回使用するデータは、行政区画を面で表現したデータとなっている。

図 1 に Polygon の図を示す。ある面に飛び地が存在する場合は、一つの描画点の座標データ中に通常の面の開始位置、飛び地の面の開始位置という順番で開始位置が記録されている。面を構成している描画点については、始点と終点が同一であり、面が閉じるデータとなっている。通常の面は時計回りに描画点が格納されている。内部に飛び地が存在する場合、飛び地の面は反時計回りに描画点が格納されている。このとき、点と点を結んで出来る線分の右側が面の内部となる。

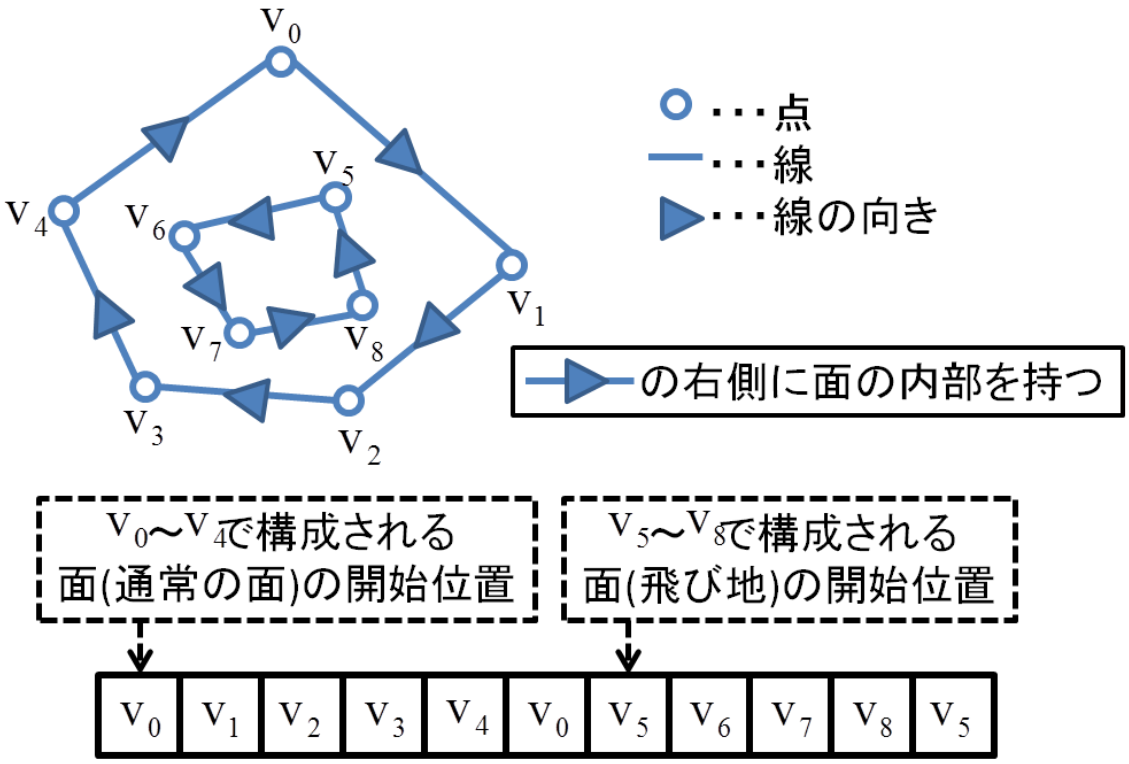

図 1　Polygon の説明図(飛び地を有する場合)

メイン・ファイルと属性ファイルは、図 2 に示す様にそれぞれ格納された順番に一対一で対応している。メイン・ファイルの「レコード・コンテンツ」に各 Polygon の描画点が格納されており、属性ファイルの「データベース・レコード」に各 Polygon の情報が格納されている。

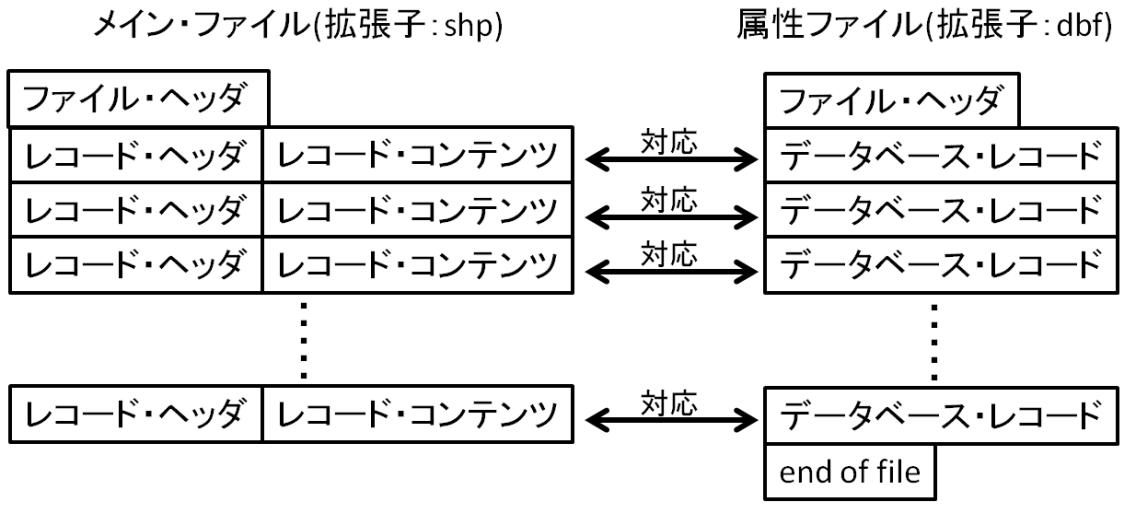

図 2　メイン・ファイル(shp)と属性ファイル(dbf)の関係

Polygon(東京都)の例を図 3 に示す。

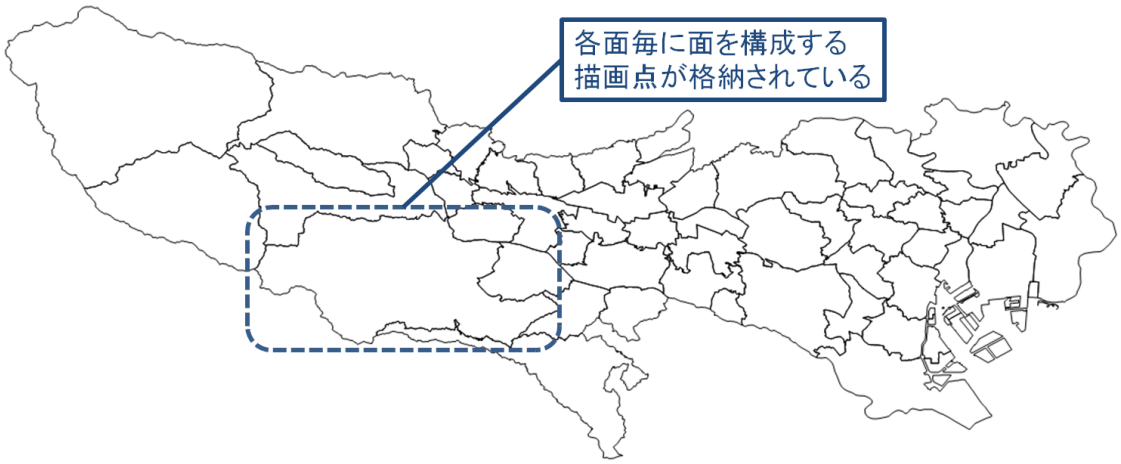

図 3　シェープファイル(Polygon)

3.1.1.2. シェープ・タイプ：PolyLine

PolyLine は、線を扱うデータである。今回使用するデータは、行政区画を線で表現したデータとなっている。

PolyLine の例(東京都)を図 4 に示す。東京都の境界線では無い箇所に線が引かれているものは、所属未確定地域の仮設線である。

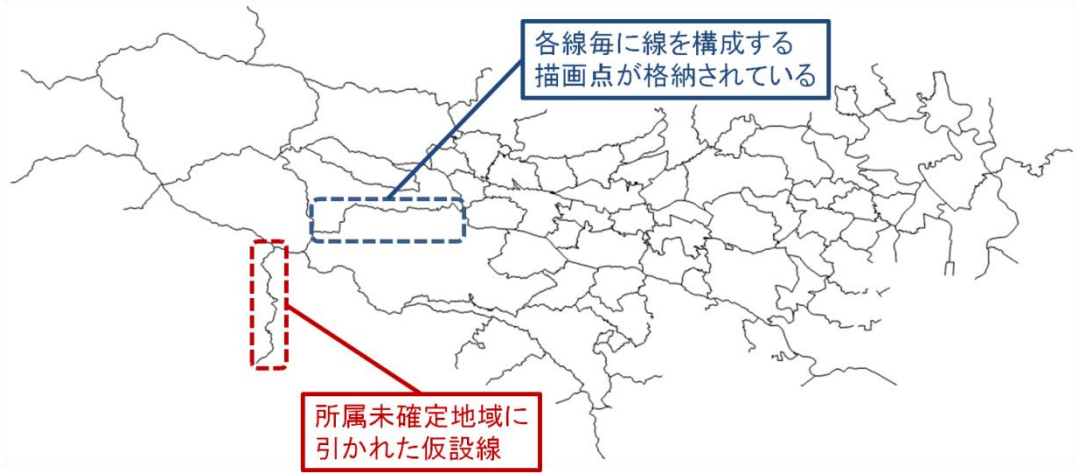

図 4　シェープファイル(PolyLine)

描画点数低減アルゴリズムを実行する際に、間引いてはいけない点をプログラムに記憶させる為に使用した。

3.1.2. 隣接行列表現

隣接行列表現[2]は、点と点を接続した辺を行列で表現する方法である。行と列がそれぞれ点番号に対応しており、どの点とどの点が繋がっているのかが理解しやすい。

しかし、都道府県規模のデータ等で隣接行列を生成しようと試みた場合、(点の重複を含まずに)点の個数を数えても数万の個数になる。その為、数万×数万の大きさの隣接行列をメモリ上に確保することが困難であるという問題点がある。

3.1.3. スパース行列(圧縮行格納方式)

3.1.2 節で述べた問題点に対して、スパース行列(圧縮行格納方式)[7]を用いることでメモリ確保の問題を解決することが出来る。

スパース行列(圧縮行格納方式)を扱うプログラムの作成にあたっては、java 言語で扱うことが可能な、数値計算ライブラリである matrix-toolkits-java[8]を参考に実装を行った。以下、スパース行列(圧縮行格納方式)を扱うプログラムに沿って、スパース行列(圧縮行格納方式)の説明を行う。

スパース行列(圧縮行格納方式)は、行列の行と列の大きさの規模が大きいものであり、かつ行列内の要素の大半が 0 である行列に対して有効な行列である。簡易な例を図 5 に示す。何行目の何列目にどんな要素の値があるのかを、それぞれ配列で管理する。columnIndex には各行毎の非 0 要素がある位置を登録し、各行毎の columnIndex の開始位置を rowPointer へ登録する。rowPointer の 0 番目が 1 行目の columnIndex の開始位置、1 番目が 2 行目の columnIndex の開始位置、…という要領で登録する。要素の値は、該当する columnIndex に対応する位置と同じ配列の添字に data として登録する。

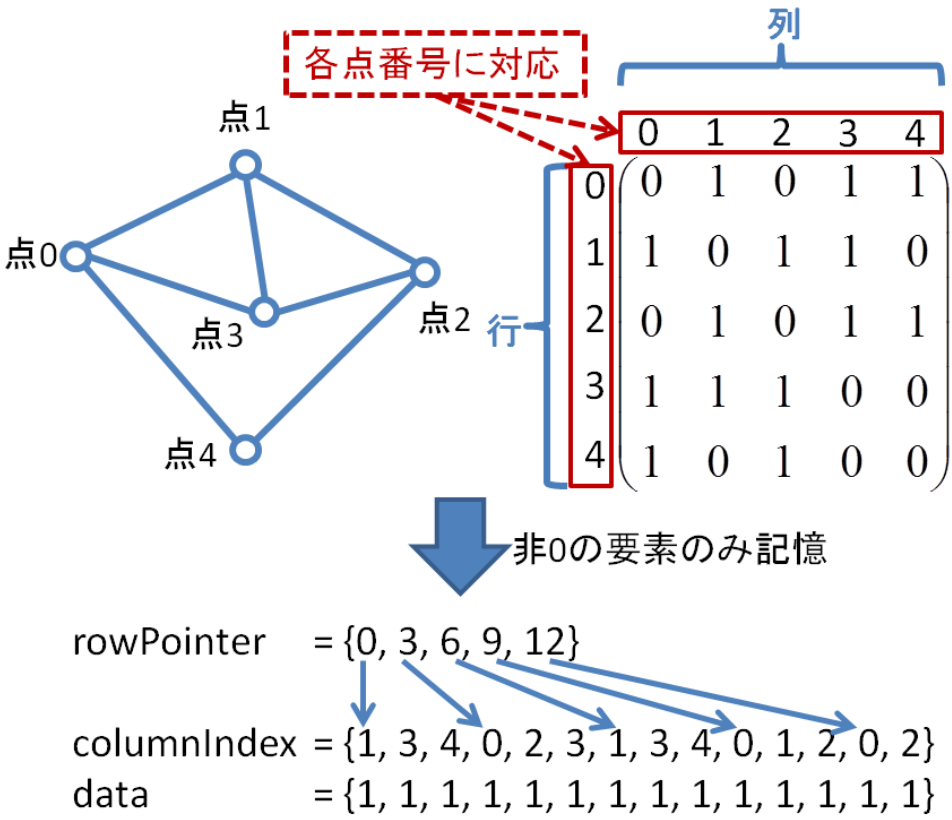

図 5　スパース行列(圧縮行格納方式)

なお図 5 の rowPointer について、図 5 には明示していないが、各行毎の columnIndex の開始値とは別に rowPointer の末尾に

非 0 の要素の数＋1

したものを登録する(9.3 節の 78 行目・90 行目)。

作成したスパース行列(圧縮行格納方式)のプログラムを利用する際は、スパース行列にする前の配列の行と列のサイズ(これは描画点数低減前の点の数にすればよい)、そして図 6 に示す様に、各行毎に非 0 の列の位置が記録された 2 次元配列 nz を作り、これらの 3 つの値をコンストラクタに渡すことで、スパース行列(圧縮行格納方式)を扱うインスタンスを生成する。このコンストラクタは、9.3 節のプログラム 41～62 行目にある。

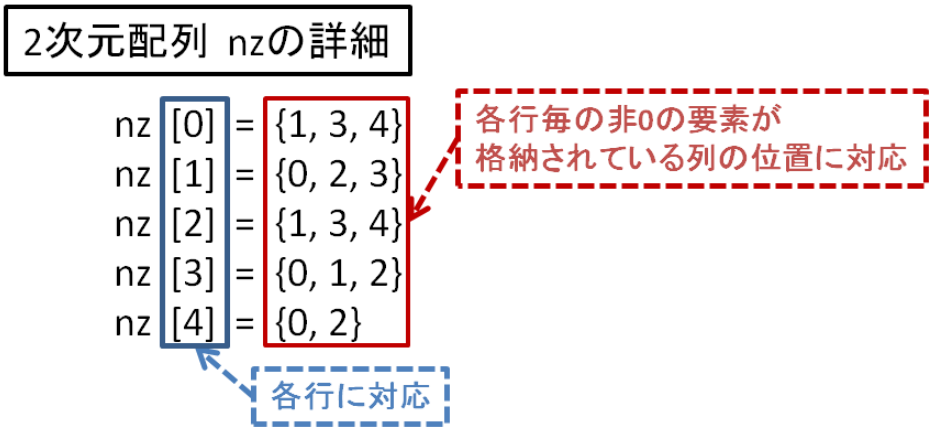

図 6　図 5 の隣接行列に対する配列 nz

スパース行列(圧縮行格納方式)の生成は、描画点数低減前と描画点数低減後で個別のインスタンスを生成することにより行なっている。

描画点数低減前のスパース行列(圧縮行格納方式)は、境界線データの隣接情報を元に生成している。この処理は、9.1 節では 293～590 行目で行なっており、9.2 節では 313～611 行目で行なっている。

描画点数低減後のスパース行列(圧縮行格納方式)は、描画点数低減アルゴリズム実行後の描画点数低減前のスパース行列(圧縮行格納方式)を元に生成している。この処理は、9.1 節では 953～1026 行目で行なっており、9.2 節では 984～1058 行目で行なっている。

3.2.　描画点数低減アルゴリズム

描画点数低減アルゴリズムは、辺の短絡除去を行うことで、描画点の間引きを行なっている。辺の短絡除去は、グラフから辺を取り除き、その辺の両端点を同一視する操作のことである[1]。

坂口[2]は、間引きを行う際の辺の選択方法として、条件 1.～条件 4.を述べている。実際のプログラムでは、以下の順番で描画点数低減アルゴリズムを実行している。

1. 以下の条件を満たした点に対して、新しい接続先の候補となる点を格納する配列 near に辺の長さが短い順に格納する
   (ア)辺の長さが閾値以下
   (イ)辺の両端点が別のある 1 つの点を共有しない
   　　(三角形の形をした面の場合、間引いてしまうと面の数が変化してしまう)

2. 配列 near に格納された点について、最も短い辺から以下の判定を行う
   (ア)新しい辺と既存の辺全てを交差判定する
   (イ)新しい辺と他の新しい辺全てを交差判定する(プログラム内ではカット)
3. 2. での(ア)・(イ) 両方の条件で交差がなければ、隣接情報の更新を行う
4. 1.～3. を全ての描画点に対して行う

ただし、今回はノード[4]の箇所を間引かないことから(3.3 節)、間引く辺は図 7 の様にノードとノードの間にある辺のみになる。その際に、図 8 の様に 2 つの既存の辺が除去され、新しい辺は 1 つしか生成されない。よって、上記の 2.(イ)の処理は不要になる。

図 7　間引きを行う対象となる辺

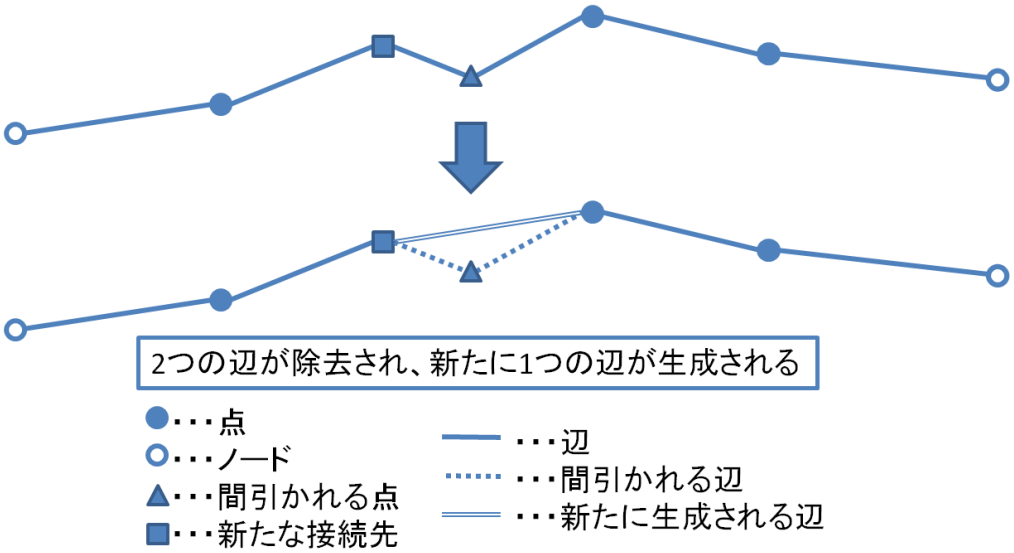

図 8　間引きの様子

また、新しい辺は 1 つしか生成されないことから、2.(ア)の処理が 1 度でも行われたらループから抜けるようにしている。この処理は、9.1 節では 892 行目の break 文で行なっており、9.2 節では 936 行目の break 文で行なっている。

描画点数低減アルゴリズムの処理は、9.1 節では 823～950 行目で行なっており、9.2 節では 860～978 行目で行なっている。また、9.1 節の 896～917 行目は上記の 2.(イ)の処理に相当する箇所であり、坂口[2]のプログラム(9.4 節)との比較目的で、コメントアウトして掲載している。

3.2.1. 交差判定

坂口[2]は、間引く辺を見つける際の交差判定をする方法について、詳しく述べていない。本節では坂口[2]のプログラム(9.4節)で実装されていた交差判定の方法について述べる。

この方法は、初等的な計算幾何学を用いたもので、3つの点の直線の傾きを求めて反時計回り・時計回りを判定する方法を応用して、2線分の交差判定を行うものである[9]。

まず、反時計回り・時計回りを判定する為のメソッドccwのソースコードを図 9に示す。メソッドccwは、9.1節では1790～1837行目に記述しており、9.2節では1875～1922行目に記述している。

```
1  /**
2   * メソッドccwは、点v0,v1,v2をv0→v1→v2の順番に移動するときにどのような順番で移動するかを判定します。
3   * @return 反時計回りなら<code>1</code>、時計回りなら<code>-1</code>、3点が同一直線上にあり点v2がv0とv1の間にあるなら<code>2</code>、いずれにも当てはまらなければ<code>0</code>
4   */
5  static int ccw(Vertex v0, Vertex v1, Vertex v2) {
6     double dx1, dx2, dy1, dy2;// dy1/dx1:(辺(v0,v1)の傾き, dy2/dx2:(辺(v0,v2)の傾き
7     //--- 点が一致するか判定 ---//
8     if ((v0.x==v2.x && v0.y==v2.y) || (v1.x==v2.x && v1.y==v2.y)){
9        return 0;
10    }
11    dx1 = v1.x - v0.x;
12    dy1 = v1.y - v0.y;
13    dx2 = v2.x - v0.x;
14    dy2 = v2.y - v0.y;
15    //--- 辺がX軸に垂直な場合dx=0となってしまうので、両辺の傾きに(dx1*dx2)を掛ける ---//
16    if (dy2*dx1 > dy1*dx2) {
17       return  1;// 反時計回り
18    }
19    if (dy2*dx1 < dy1*dx2) {
20       return -1;// 時計回り
21    }
22    //--- 同一直線上で「v1からv2への方向」が「v1からv0への方向」と同じで、「v1からv2の距離」が
23    「v1からv0の距離」より大きい ---//
24    if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) {
25       return 0;
26    }
27    //--- 同一直線上で「v0からv2への方向」が「v0からv1への方向」と同じで、「v0からv2の距離」が
28    「v0からv1の距離」より大きい ---//
29    if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2)) {
30       return 0;
31    }
32    return 2;  // 3点が同一直線上にあり、v2が「v0とv1の間」にある場合
33 }
```

図 9　メソッドccw(付録から抜粋、一部改編)

メソッドccwは、3つの点(v0・v1・v2)を引数として受け渡し、v0→v1→v2の順番で辿った時に、その順番が、反時計回り・時計回り・3点が同一直線上かつv2がv0とv1の間

にある・それ以外のいずれかを判定する。以下で、図 9 のソースコードの解説を行う。

10～12 行目では、3 つの点が 1 つでも一致する場合を判定している。

13～16 行目では、Δx, Δy(図 9 での dx, dy)を、図 10 の様に定義し、Δx, Δy を求めている。

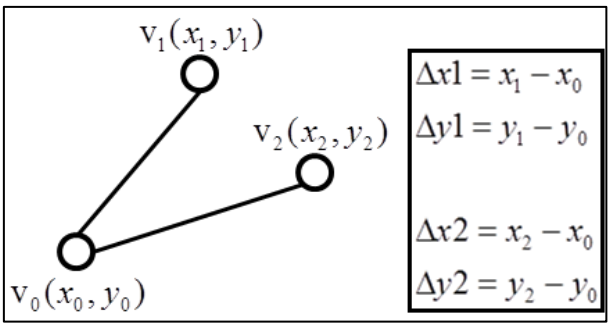

図 10 Δx, Δy の定義

18～20 行目の動作を表したものを図 11 に、21～23 行目の動作を表したものを図 12 に示す。

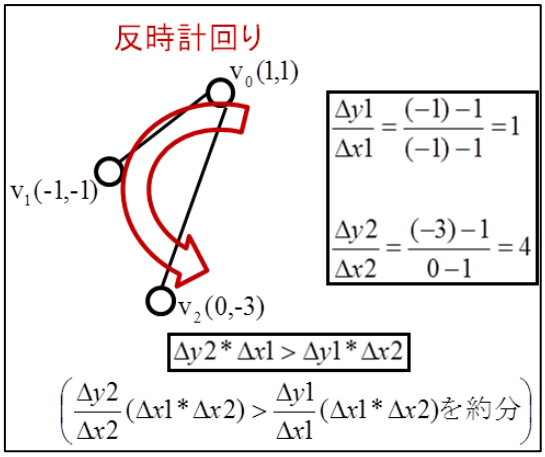

図 11 反時計回りの判定(一例)

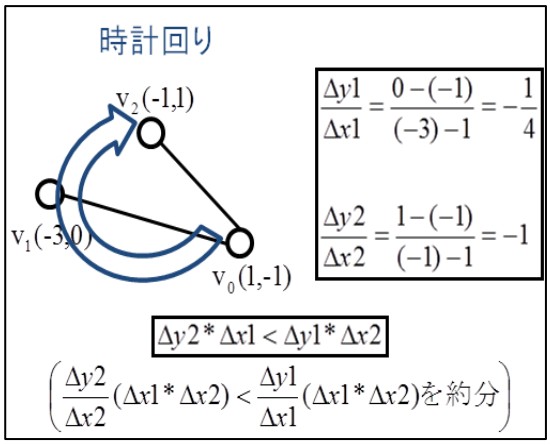

図 12 時計回りの判定(一例)

また、図 11・図 12 の判定式が約分されている理由は dy1/dx1 または dy2/dx2 の一方が図 13 の様に垂直であった場合に対応する為である。

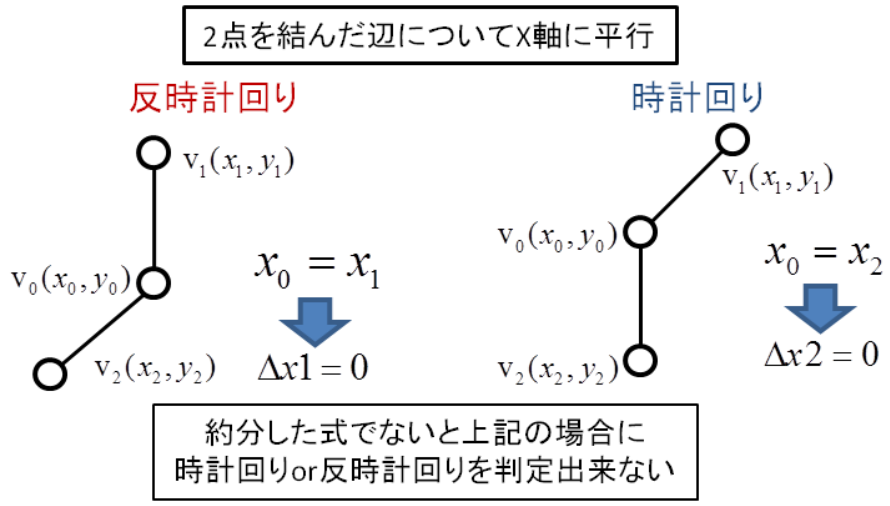

図 13　図 11・図 12 の判定式が約分されている理由

26～28 行目の動作を表したものを図 14 に、31～33 行目の動作を表したものを図 15 に示す。

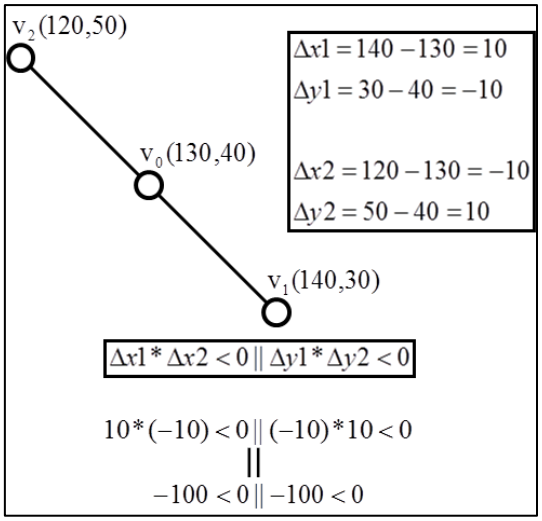

図 14　v2-v0-v1 となる場合(一例)

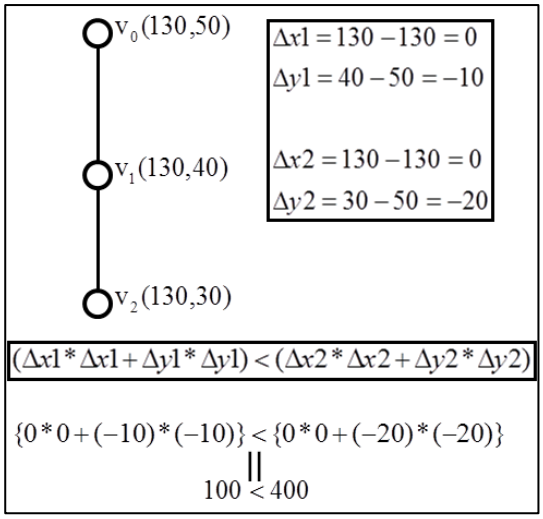

図 15　v0-v1-v2 となる場合(一例)

上記の条件を全て満たさなかった 34 行目では、点 v0 と点 v1 の間に点 v2 が存在する場合が該当する。

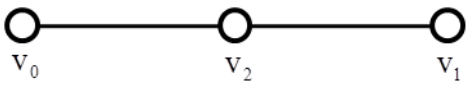

図 16　v0-v2-v1 となる場合(一例)

次に、メソッド ccw を利用して、交差判定を行うメソッド isIntersection を図 17 に示す。メソッド isIntersection は、9.1 節では 1750～1788 行目に記述しており、9.2 節では 1833～1873 行目に記述している。

```
1  /**
2   * メソッドisIntersectionは、辺(v0,v1)と辺(u0,u1)の交差判定をします。
3   * @return 交差するならば<code>true</code>、交差しないならば<code>false</code>
4   */
5  static boolean isIntersection(Vertex v0, Vertex v1, Vertex u0, Vertex u1) {
6     int comm_flag = 0;// 両辺が点を一つだけ共有していれば1、さもなければ0
7     //--- 辺が完全に一致すれば交差すると判定 ---//
8     if (((v0.x==u0.x && v0.y==u0.y) && (v1.x==u1.x && v1.y==u1.y))
9           || ((v0.x==u1.x && v0.y==u1.y) && (v1.x==u0.x && v1.y==u0.y))) {
10       return true; // 交差する
11    }
12    //--- 点を一つだけ共有している ---//
13    if ((v0.x==u0.x && v0.y==u0.y) || (v0.x==u1.x && v0.y==u1.y)
14          || (v1.x==u0.x && v1.y==u0.y) || (v1.x==u1.y && v1.x==u1.y)) {
15       comm_flag = 1;
16    }
17    //--- 他方の辺上に点があれば交差すると判定 ---//
18    if (ccw(v0,v1,u0)==2 || ccw(v0,v1,u1)==2 || ccw(u0,u1,v0)==2 || ccw(u0,u1,v1)==2) {
19       return true; // 交差する
20    }
21    //--- 他方の辺上に点が無く、1つの点を共有しているか、同一直線上に点を持てば交差することはないので
22 comm_flag==1とccw==0は考えない ---//
23    if (comm_flag==0
24    && (ccw(v0,v1,u0)!=0 && ccw(v0,v1,u1)!=0 && ccw(u0,u1,v0)!=0 && ccw(u0,u1,v1)!=0)) {
25       //--- 両辺の両端がもう片方の辺に対して同じ側にない場合は交差すると判定 ---//
26       if ((ccw(v0,v1,u0)!=ccw(v0,v1,u1)) && (ccw(u0,u1,v0)!=ccw(u0,u1,v1))) {
27          return true; // 交差する
28       }
29    }
30    return false;      // 交差しない
31 }
```

図 17 メソッド isIntersection(付録から抜粋、一部改編)

メソッド isIntersection は、辺(v0,v1)と辺(u0,u1)の点を 4 つの点を引数として受け渡し、2 つの辺が交差するか交差しないかを判定する。以下、図 17 のソースコードについての解説を行う。

8～11 行目では、辺が完全に一致する場合に交差すると判定する。

13～16 行目では、2 つの辺が 1 つの点を共有する場合を判定する。

18～20 行目では、2 つの線分が図 18 の様な位置関係である場合に交差すると判定する。

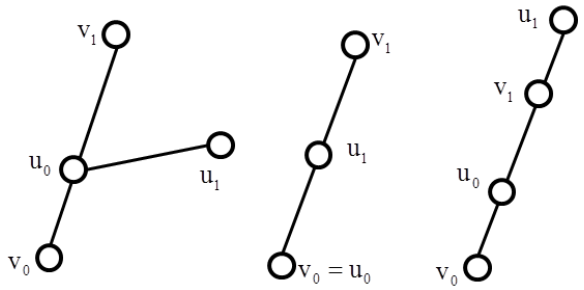

図 18　他方の辺上に点がある場合(一例)

23～24 行目の if 文では、他方の辺上に点が無く、1 つの点を共有する場合(図 19(a))と他方の辺上に点が無く、同一直線上に点を持つ場合(図 19(b))は交差しないと判定している。

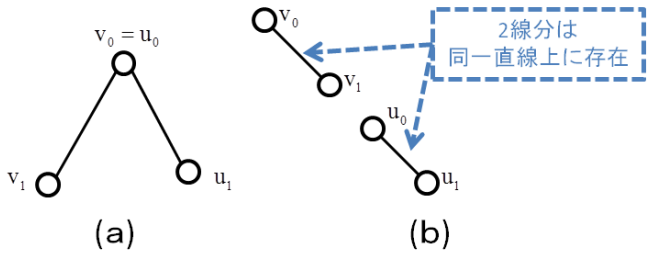

図 19　「他方の辺上に点が無い」かつ
「1 つの点を共有」または「同一直線上に点を持つ」場合(一例)

26～28 行目の動作を表したものを図 20 に、30 行目の動作を表したものを図 21 にそれぞれ示す。この時、呼び出されるメソッド ccw の取り得る値は 1(反時計回り)か-1(時計回り)のみとなる。

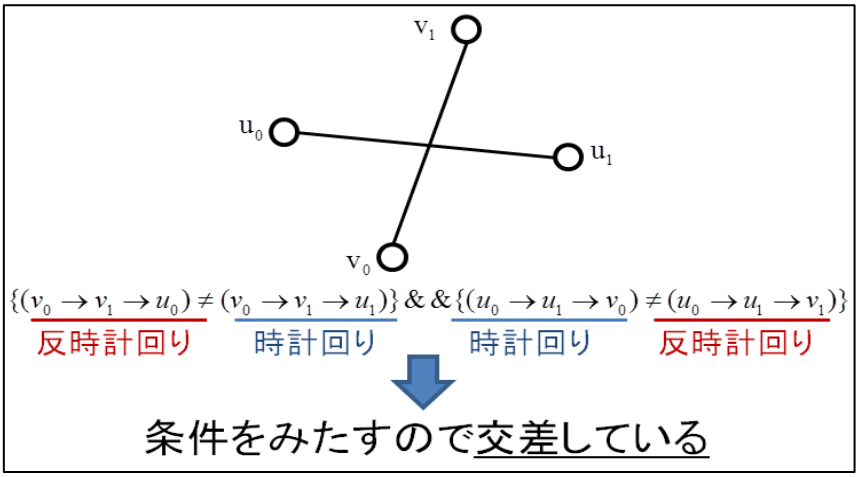

図 20　交差する場合(一例)

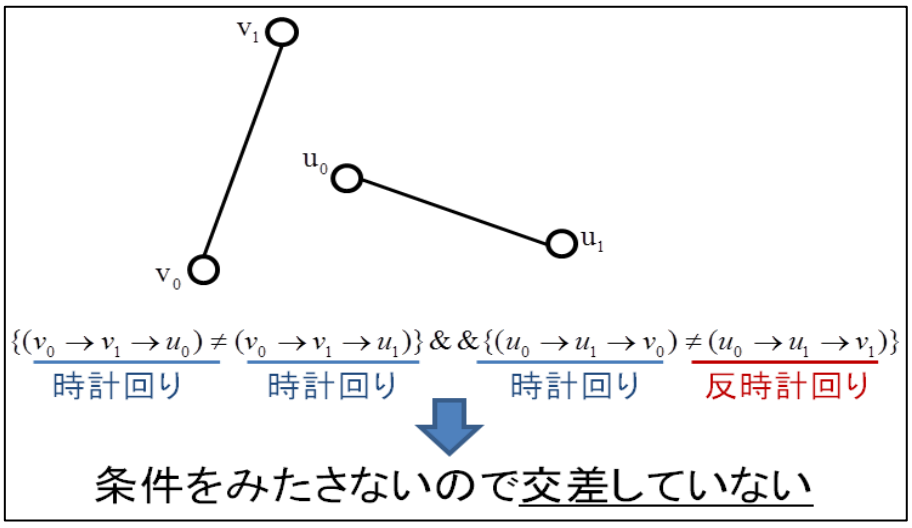

図 21　交差しない場合(一例)

そして、この判定を新しい辺と全ての既存の辺に対して行うことで、隣接情報を更新した際に交差が生じることなく間引くことが出来る。

3.3.　面情報を記憶した隣接データ作成

隣接行列表現[2]では、点と点の隣接関係を記憶する為のものである為、面の情報を記憶することができない。これでは、面の情報をシステムへ登録する際に不便である。

そこで、間引き前のシェープファイル(メイン・ファイル)を再度読み込み、面に沿って間引き後の描画点を新しいシェープファイルへ書き出すようにした。

この方法では、まず図 22 の様に間引き後に 1 つの面を構成する XY 座標を読み込み、点を重複せずに登録する処理箇所で記憶しておいた点番号に変換する。すると、間引かれてない点は XY 座標に対応する点番号が返され、間引いた点については-1 という値が返される。この処理は、9.1 節では 1208～1230 行目で行なっており、9.2 節では 1244～1266 行目で行なっている。

その後、図 23 の様に面を先頭から辿って行き、-1 以外の点と点を結んだ辺が隣接していれば点番号に対応した XY 座標を書きだすという仕組みになっている。この処理は、9.1 節では 1234～1471 行目で行なっており、9.2 節では 1270～1507 行目で行なっている。

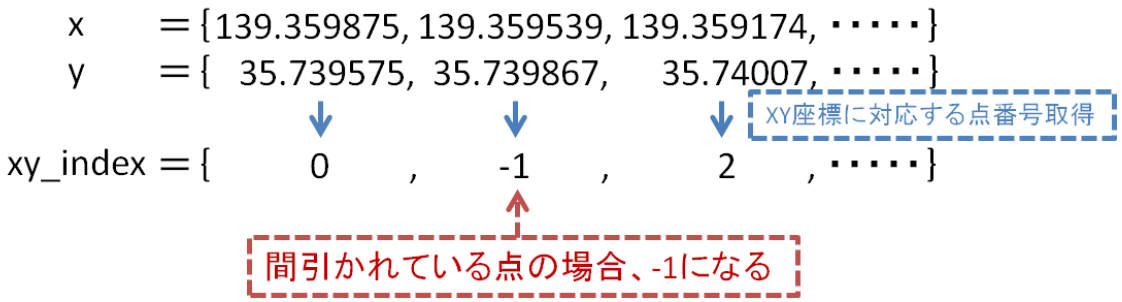

図 22　座標に対応する点番号の取得(間引き後)

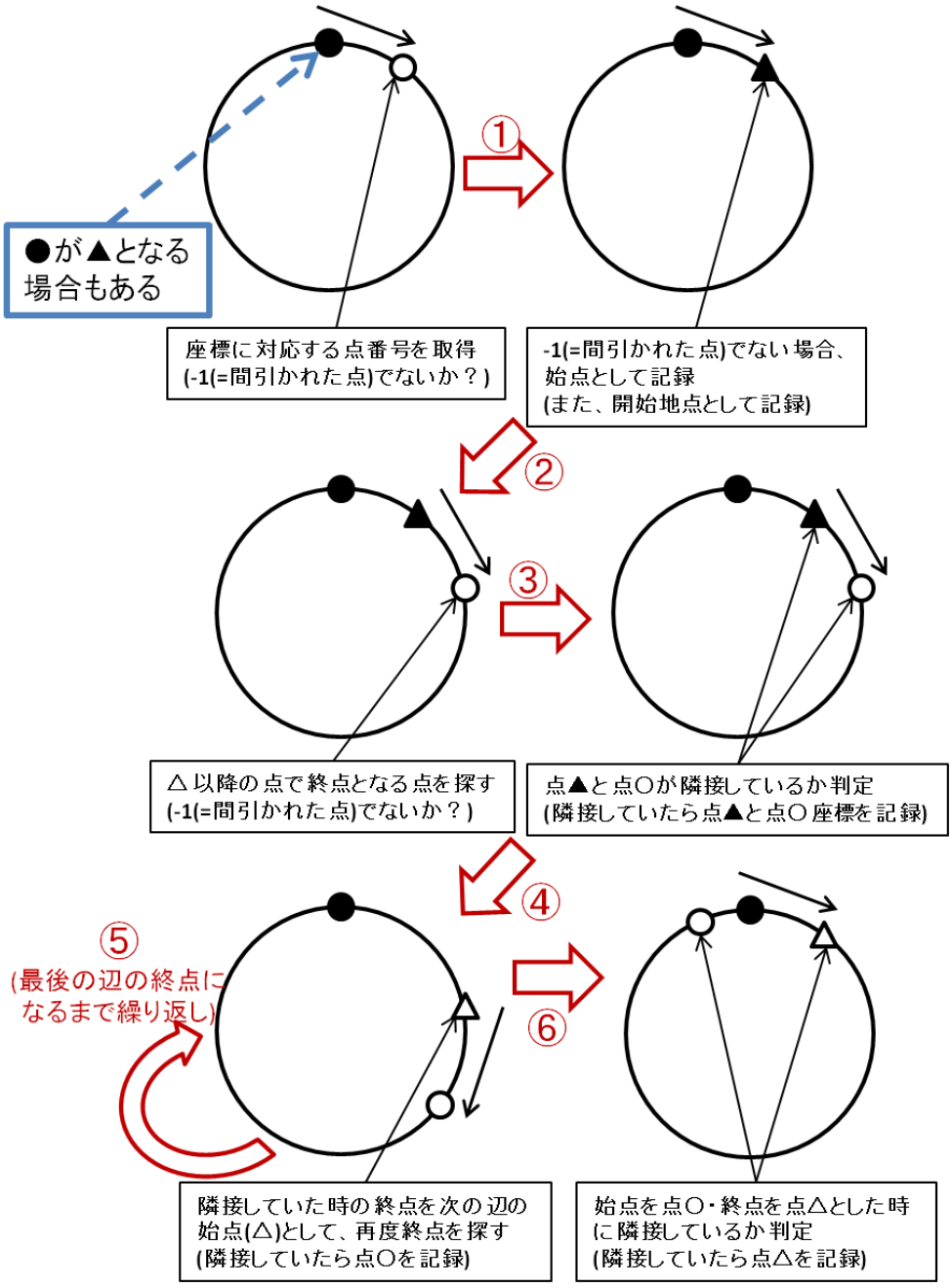

図 23　面情報を書き出す方法

また、上記のことを行う際には、描画点数低減アルゴリズムを実行するときに面同士の曲線が交わる・分岐する点であるノード[4]を間引かないようにする必要がある。これを行わないと別の面の点に描画点が間引かれてしまい、面情報を記憶した状態での書き出しを行うことが出来ない。この処理は、9.1 節では 833 行目の if 文で行い、9.2 節では 874 行目の if 文で行なっている。

ノード[4]の登録は、3.1.1.2 項の PolyLine から、各辺データの始点・終点を取り出すことで行なっている。この時、図 4 の様に所属未確定地域に引かれた仮設線の点が含まれてしまうので、図 3 にある Polygon に属する点のみを登録するようにしている。これらの処理は、9.1 節では 650～820 行目で行い、9.2 節では 684～854 行目で行なっている。

4. 描画点数低減アルゴリズムの実行結果

描画点数低減アルゴリズムの実行結果について、描画点数低減前の結果を図 24 に、描画点数低減後の結果を図 25 に示す。閾値については、見れば明らかに間引かれている事が分かる閾値 0.085[cm]で実行したものを載せている。

尚、閾値・辺の長さの単位である[cm]は、地図上の縮尺 1/25,000 における[cm]である。

図 24　東京都(描画点数低減前)

図 25　東京都(描画点数低減後,閾値 0.085[cm])

読み込んだ東京都のデータに於ける点の数・辺の数・辺の長さの最小値・最大値を示したものを表 2 に示す。このとき、描画点数低減アルゴリズムの閾値は、表 2 の辺の長さの最小値から最大値の間で指定することになる。

表 2　東京都の描画点数低減前の辺の長さの最小値・最大値

辺の長さ(最小値)[cm]	辺の長さ(最大値)[cm]
0.000141	2.173642

描画点数低減前・低減後の点の数・辺の数の変化を表 3 に示す。点の数・辺の数については、重複の無いように数えている。

表 3　東京都の描画点数低減前・低減後の点の数・辺の数

閾値[cm]	点の数	辺の数
-	45788	45845
0.085	4938	4995

実行時間をまとめたものを表 4 に示す。

表 4　図 25 の間引きの閾値実行時間

間引きの閾値[cm]	実行時間
0.085	65 分 21.077528 秒

測定及び開発を行った計算機環境は表 5 の通りである。

表 5　測定・開発を行った計算機環境

OS	Microsoft Windows7 Professional 64[bit]
CPU	Intel® Pentium® CPU G6950 @2.80[GHz]
メモリ	8[GB]
開発環境	eclipse Helios 3.6
JDK	Jdk1.6.0_30

次に、坂口[2]が最終的に用いた地図グラフ表現の標準的データ構造を利用した方法のプログラム(C 言語)に対して、今回使用した東京都のシェープファイルの変換前の GML のデータを用いて、間引きの閾値を表 4 と同じ閾値にして実行した際の実行時間を表 6 に示す。

表 6　地図グラフ表現の標準的データ構造を応用した方法での実行時間

間引きの閾値[cm]	実行時間
0.085	6 分 27.023000 秒

5. 考察

この章では、1 章の序論で述べた実行時間の問題に対する対処と実行時間について述べる。

今回、坂口[2]のプログラムとの比較を目的としたプログラム(9.1 節)と実行時間の高速化を行ったプログラム(9.2 節)の 2 種類のプログラムを作成している。この様にしたのは、坂口[2]のプログラムとの比較を目的としたプログラムは、実行時間が掛かり過ぎる為である。プログラムを実行し、5 日程計算機を放置して様子を見た時点で点の数 45788 個中約 600 個目までしか処理されていなかった。

実行時間が遅い理由を述べる。坂口[2]のプログラムとの比較を目的としたプログラムでは、新しい辺と既存の辺全てに対して交差判定をする箇所(3.2 節の 2.(ア))がある。この箇所で図 26 改良前の様に対角成分を除いた上三角行列に相当する箇所を

$$(45,788\times45,788-45,788)/2=1,048,247,578$$

もなぞって隣接判定・交差判定を行なっており、これを 45,788 個の点全てに対して判定していた為、実行時間が遅くなっていた。

そこで、図 26 改良後の様に新しい辺と既存の辺全てに対して交差判定をする箇所にて、なぞる対象を、対角成分を除いた上三角行列に相当する箇所から、スパース行列で数値を管理している配列そのものを取得し、それをなぞるという形に変更した。こうすることで、なぞる数は辺の数＋新しい辺となることから

$$2\times45845+\alpha$$

のみとなる。

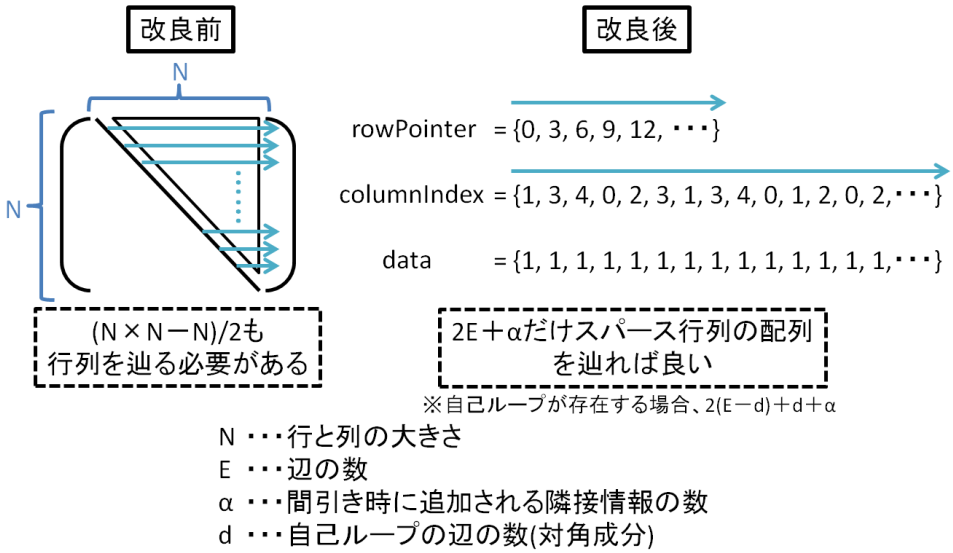

図 26　プログラムの改良前と改良後

表 4 からも分かるように、これまで 5 日間掛かっても 45,788 個中 600 個目までしか間引きが行われていなかったものが約 1 時間程度で済ませることが出来るようになった。

図 26 改良前の処理は、9.1 節の 865～895 行目に相当する。図 26 改良後の処理は、9.2

節の 911～939 行目に相当する。

6. 今後の課題

4 章の描画点数低減前・低減後の東京都行政区画境界線データを示した図 24・図 25 より、明らかに描画点数が低減されたことが確認できる。

しかし、表 4・表 6 より地図グラフ表現の標準的データ構造を利用した方法には、実行時間で劣ってしまっている為、今後は今回理解出来たことを生かして地図グラフ表現の標準的データ構造を利用した方法を Java 言語に移行していく必要がある。

7. 謝辞

本研究にあたり、最後まで熱心なご指導をいただきました田中章司郎教授には心より御礼申し上げます。また、田中研究室の皆様からは本研究に関して数々の御協力と御助言をいただきました。厚く御礼申し上げます。なお、本論文・本研究で作成したプログラム及びデータ、並びに関連する発表資料等の全ての著作権を、本研究の指導教官である田中章司郎教授に譲渡致します。

8. 引用文献

[1]伊理正夫, 腰塚武志(1993), 計算幾何学と地理情報処理 第 2 版, 共立出版

[2]坂口隼(2008), グラフ理論に基づく複体の描画点数低減の検討, 島根大学 数理・情報システム学科 卒業論文

[3]基盤地図情報ダウンロードサービス
http://fgd.gsi.go.jp/download/

[4]今井拓也(2006), 動的に描画点数を考慮した非同期数値地図情報表示システムの設計と実装, 島根大学 数理・情報システム学科 卒業論文

[5]GIS ソフト - Shape Viewer
http://www.geocities.jp/hykgis/shapeviewer.html

[6]シェープファイルの技術情報(日本語版,訳)
http://www.esrij.com/cgi-bin/wp/wp-content/uploads/documents/shapefile_j.pdf

[7]Compressed Row Storage(CRS)
http://netlib.org/linalg/html_templates/node91.html#SECTION00931100000000000000

[8]matrix-toolkits-java(ライブラリの CompRowMatrix.java)
http://code.google.com/p/matrix-toolkits-java/

[9]R.セジウィック(原著), 野下浩平, 星守, 佐藤創, 田口東(邦訳)(1994), アルゴリズム C++, 近代科学社