目 次

第1章　はじめに

生体分子ネットワーク構造や, プログラムのフローチャート図など, データ間の関連性がある構造を表現するために, しばしばグラフ構造が用いられる. グラフとは有限のノード集合と, ノード間を結合するエッジ集合からなるデータ構造である. グラフ表現されたデータから人間が情報を得るには, グラフを適切に視覚化し, ノード間の関係やノードの情報が分かりやすく見えていることが望ましい. しかし, グラフを分かりやすく視覚化することは難しく, グラフが大規模になると人間が手作業で視覚化することは困難である. 任意のグラフからグラフの二次元描画を生成する問題は, グラフ描画問題と呼ばれ, 今日までに多くの手法が提案されている.

グラフ描画は, 人間が理解しやすいことが望ましいが, グラフごとにノードやエッジに与えられる意味は異なり, 人によって見易さの好みが異なるため, グラフ描画に最も良いという基準を与えることは出来ない. そのため, グラフの特徴などを考慮した適切なグラフ描画を考えていく必要がある.

現在, グラフ描画には力学モデルによる方法などが用いられている. グラフを力学的モデルとみなしてシミュレーションを行い, 系が安定するまで計算することで, グラフノード間が均衡した状態になるようなグラフ描画を求めることができる.

[ここで他のグラフ描画アルゴリズムについて述べる]

しかし, ノードの情報に着目したい場合, グラフ描画上にノード情報を表示させるために, ノード間に一定以上の空間を確保したいケースが考えられる. 従来のグラフ描画手法では, グラフ描画にノード間に一定以上の空間を保障することはできない.

この目的のために, 吉田英聡は [1] において, グラフノードをパターン, グリッド状の格子点をテキストとみなした二次元点集合の近似照合によって, 多項式時間で動作する格子状配置アルゴリズムが提案された. この手法により, 任意のグラフのグラフ描画において, 元々のグラフ描画の位置関係を維持しながら, グラフのノード間にグリッド幅以上の空白を確保したグラフ描画の生成が可能となった.

しかし, 二次元点集合の近似照合は時間においても領域においても計算量の次数が高く, 現実的なサイズのグラフ描画に対して直接適用することができなかった. これに対し [1] では, kd-木分割やメッシュ状分割などでグラフを分割統治し, 大規模なグラフに実行できるようにした. しかし, ユーザが対話的に実行し, グラフのグリッド配置を実行するには, ユーザがストレスを感じない程度の素早い応答時間を要求し, 依然として実行時間の高速化が求められている.

そこで本研究では, 二次元点集合近似照合における格子点集合 (テキスト) が, 格

子状であることに着目し, 二次元点集合近似照合を, グラフノード点集合 (パターン) の部分集合を包含する格子点集合に対して軸平行な最小の矩形を部分点集合の境界矩形とし, 空集合からパターン全体になるまで, サイズが 1 ずつ増加するような真に包含関係にある点集合列に対応する矩形拡張の列を考え, 矩形拡張列を最小の平行移動によって格子点集合上に格子配置する問題にみなせることを利用する.

以下では, 第 2 章において, 任意の点集合に対する境界矩形および境界矩形拡張列と, 境界矩形拡張列の格子配置における編集距離の定義を行い, 境界矩形の編集距離の再帰的定義と, 時間・空間ともに $O(n^2)$ の計算量で実行可能な動的計画法 (DP) による解法を紹介する. 第 3 章では, 境界矩形拡張列の格子配置によるグリッドグラフ描画アルゴリズムについて示した後, アルゴリズムの実行時間と, いくつかのグリッドグラフ描画の修正法を示し, グリッドグラフ描画の結果を示す. 第 4 章では結論と, 今後の課題について述べる.

第2章　境界矩形拡張列の格子配置問題とその解法

本章では, 二次元点集合のグリッド近似照合と, 境界矩形拡張列の格子配置について述べる. まずはじめに, これまで行われてきた研究について紹介する.

2.1　これまでの研究

二次元点集合の近似照合の定義として, 吉田 [1] は次のように定義した.

定義 1 n 個の点集合をそれぞれ P, Q とする. このとき, $1 \geq i, j, k, l \geq n$ について $P[i,j]$ と $Q[k,l]$ の編集距離 $d(i,j;k,l)$ を

1. $|P[i,j]| = |Q[k,l]| = 1$ なら $d(i,j;k,l) = 0$,
2. $|P[i,j]| \neq |Q[k,l]|$ なら $d(i,j;k,l) = \infty$,
3. それ以外なら $d(i,j;k,l) =$
$$\min \left\{ \begin{array}{l} d([i,j]_-;[k,l]_-) + |p([i,j])_R - p([i,j]_-)_R - (q([k,l])_R - q([k,l]_-)_R)|, \\ d([i,j]^-;[k,l]^-) + |p([i,j])_T - p([i,j]^-)_T - (q[k,l])_T - q([k,l])^-)_T)| \end{array} \right\}$$

と定義する.　$|\cdot|$ は L_1 のノルムである.

2.2　境界矩形拡張列とその性質

格子配置のための二次元点集合では, パターンとなるグラフノードを表す二次元点集合に対して, テキストは格子点集合は次のように定義できる.

$$\mathrm{Grid}(d) = \{(dx, dy) | x, y \in \mathbb{N}\} (d \in \mathbb{Z})$$

任意の二次元点集合 $S = \{(x,y) | x, y \in \mathbb{Z}^{+2}\}$ に対して, S の点を全て包含する格子点集合に対して平行な境界矩形 R を与えることができる. R は, 対応する S の点によって矩形の左下と右上を定める点または点対によって, 矩形を表現することができる. すなわち S の部分境界矩形は, S から矩形の上下左右を定める点の組み合わせの数だけ存在する.

境界矩形拡張列とは, 任意の二次元点集合には境界矩形拡張列が存在する.

2.3　境界矩形拡張列の格子状配置における編集距離

境界矩形が格子配置されているとは, 境界矩形に含まれる任意の点が重複せずに格子点上にあるということである. ある境界矩形が格子配置されるまでには, 含まれるすべての点を格子点集合に重複せずに対応する平行移動をする必要がある.

しかし, 任意の点集合を包含する境界矩形拡張列における各々の境界矩形は, サイズの小さい包含矩形を含んでおり, その格子配置における編集距離の計算は, 部分問題として, サイズが下位の境界矩形の格子配置における編集距離の計算を含んでいる.

この時, ある境界矩形拡張列がその編集距離を求める計算の部分問題には, 境界矩形拡張列が境界矩形を拡張する方向が影響する. 例えば, 任意の方向に拡張して良い場合, 境界矩形を表現する左下右上点のいずれか一つを取り除いた場合, すなわち部分問題は 4 方向から選択することになる. 部分問題の個数は, 計算量に大きな影響を与えるため, ここでは計算量の次数を下げるために, 部分問題の選択は右上の 2 方向に限定することにする.

つまり, 各境界矩形の編集距離は次の再帰式で与えられる.

2.4　解法

境界矩形拡張列の編集距離を計算する動的計画法によるアルゴリズムを定義する.

[ここでアルゴリズムの説明]

このアルゴリズムは, 時間・計算ともに入力の点集合のサイズ n に対して, 最悪計算量 $O(n^2)$ で動作する.

第3章　実験結果

この章では, 境界矩形拡張列によるグラフの格子配置アルゴリズムを実装し, アルゴリズムの性能を調べる実験を行う. プログラムは C++(MinGW32-gcc-4.6.2) で実装し, Intel Core i7-3770 CPU (3.40GHz) のマシン上で実行した. マシンの RAM は 8.00GB だが, プログラムは 32bit 環境向けにコンパイルしたため, プログラムが利用可能なメモリアドレス領域は 4GB 未満であった.

3.1　グラフの格子配置アルゴリズム

境界矩形拡張列の DP 表をバックトレースしながら, 各々の点に対して, 境界矩形拡張に合わせるように点を平行移動させることで, $O(n)$ でグラフ配置させることができる.

[格子配置アルゴリズムを書く]

3.2　グラフに入力される重複した点の対策

アルゴリズムでは考慮しないことにしたが, 実用上では入力される二次元点集合に重複点が与えられることが考えられる. X 軸, Y 軸列を安定ソートするなどして, 重複点に一意な順序関係を与えてやることで, 格子状配置アルゴリズムを適用させることは容易であるが, 冗長な形状に展開されることもある. 元々, 重複点が与えられるということは, グラフノード間に密接な関係があると考えることが人間にとって自然であるので, 隣接関係を維持しつつグリッド状に展開する方法をアルゴリズムの拡張として組み込んだ.

重複点は一つの点とみなしてアルゴリズムの計算を行い, 重複点を境界矩形に追加して拡張を行う際, 重複点を最小の境界矩形に展開し, そのサイズを境界矩形拡張の編集距離に追加することで処理した.

[重複点展開アルゴリズム]

3.3 伸長したグラフの畳み込み処理

アルゴリズムでは, 計算量の次数を下げるために右上方向にのみ拡張をするように提案した. そのため, グリッド幅に対して密な入力が与えられた場合, グリッドグラフ描画は大きく右上方向に伸長した形状になる. 過度に伸長したグラフは線形に近づき, 通常の矩形ディスプレイに表示するには形状が不適切である. また, 元々のグラフ描画において非常に隣接している点は, そのため, 出力が矩形に近づくようにグラフノード間の不必要な空白を畳み込み, グリッドグラフ描画の幅と高さを小さくする処理を後処理として行った.

[伸長の畳み込み処理アルゴリズム]

3.4 実験結果

実装したプログラムに実際に点集合を入力として与え, プログラムの実行時間と, 点集合のグリッド配置によるグラフの伸長を計測した. 図 3.1, 図 3.2 は, サイズ 100 程度の点集合に対して, アルゴリズムを実行した結果である. 実行時間は 1msec 以下で動作した. 図 3.2 において, グラフ描画全体を包含する境界矩形のサイズは, ○○だけ伸長した.

次に, サイズ 10000 までの点集合に対して実行速度を計測するため, プログラムを実行した. 表 3.1 にその結果をまとめる.

表 3.1 入力サイズと実行時間

サイズ (点)	実行時間 (sec)
1000	0.093
2000	0.343
3000	0.765
4000	1.435
5000	2.231
6000	3.260
7000	4.571
8000	6.035
9000	bad-alloc

実行速度は $O(n^2)$ におさまる範囲で遷移しており, サイズが 3000 程度の点集合に対しても, 十分に対話的に利用できる速度で実行された.

サイズは 8000 以上の入力に対しては, 実験環境では DP 表を生成するための連続した大きなヒープ領域を確保することができなかったので, アルゴリズムを検証することができなかった.

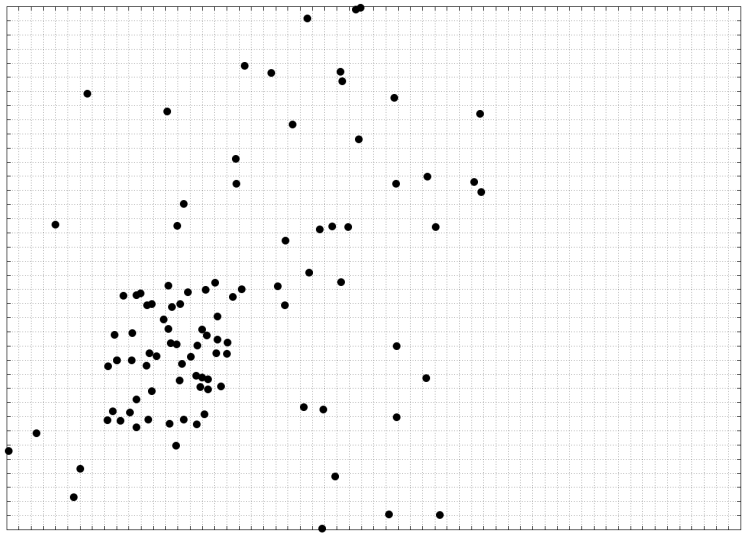

図 3.1 Size100 のグラフ

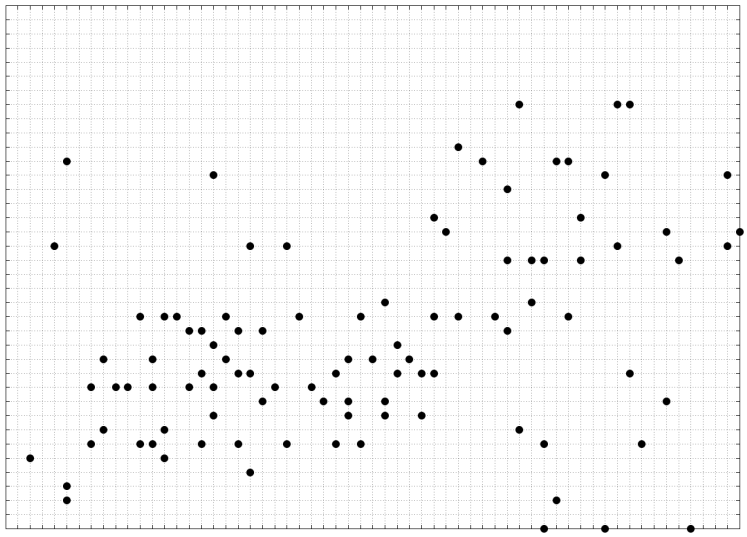

図 3.2 矩形拡張列による格子配置

第4章　終わりに

本研究では, まず点集合に対する境界矩形と境界矩形拡張列を定義し, 境界矩形拡張列の格子配置における編集距離を再帰的に定義した. 次に, 動的計画法による境界矩形拡張列の格子配置における編集距離を, 空間および時間ともに最悪計算量が $O(n^2)$ で実行できるアルゴリズムを提案した.

そして, 境界矩形拡張列によるグラフの格子配置アルゴリズムを提案し, 実行時間とグラフの伸長に関する実験を行った. さらに, 重複点の展開や, グラフ展開後の畳み込み処理の追加により, 格子配置の伸長を抑え, 元のグラフ描画における点の局所性を, 格子配置後のグラフに反映させることができた.

実装したプログラムは, サイズ 3000 程度まで十分に対話的に利用できるだけの速度で動作し, アルゴリズムが実用的な速度で利用可能な実装ができることを示した.

しかし, グラフのサイズに対して疎なグリッド幅で展開する場合, グリッド配置は右上方向に伸長し, グラフの全体像は線形に近づいてしまう. 2 次元グラフ描画をディスプレイするモニターが一般的に矩形であることを考えると, 一度にグラフの可能な限りの範囲をディスプレイに収めるには不適当な形状である.

よって, グリッド配置全体が矩形に近づくように整形されることが望ましい. 本研究では, グリッド配置後の畳み込み処理の実装により対応したが, 局所的な畳み込みでは, グラフ全体の配置を大きく破壊してしまう恐れがあるため, グラフの全体像を踏まえたうえでの適切な整形を行う手法が必要である.

謝辞

本研究を進めるにあたり, 細部に至るまでご指導頂いた下薗真一准教授に深く感謝いたします. また, 日々の研究でお世話になった研究室の皆様に深く感謝の意を表します.

参考文献

[1] 吉田英聡, 二次元点集合近似照合によるグラフの格子状配置アルゴリズム, 電子情報通信学会総合大会講演論文集 2008 年 情報・システム (1), ”S-17”-”S-18”, 2008-03-05