

計算幾何学的な手法を用いた高速相同性計算手法

松 井 鉄 史^{†1} 宇 野 毅 明^{†1}

編集距離 (レーベンシュタイン距離) は文字列の距離を表すモデルの一つであり、ゲノムの比較をする際に頻繁に使われている。その一方で、素朴な方法は文字列長の 2 乗の計算時間であること、近似なしの高速計算は難しいこと、配列比較で重要視される局所的な類似性を考慮せず、とびとびの対応である程度類似すると判断するという問題点もある。本稿では、局所的な類似性に基づいた新しい類似性の尺度とその高速な計算手法を提案する。直観的には、編集距離はいかに多くの同一な文字を、入れ替えなしで対応できるかであるが、提案する断片編集距離では、ハミング距離が閾値以下の (長さが固定された) 短い文字列を対応させる。これにより局所的な類似性を持つ部分のみを考慮できる。また、多くの場合、このようなハミング距離が閾値以下の短い文字列の組は、配列上に連続して現れることから、連続した出現をひとまとまりにすることで入力データ量を減らすと共に、計算幾何学的な手法を使うことで距離を高速計算するアルゴリズムを提案する。計算実験の結果、既存手法では計算できないような巨大な配列に対しても、本アルゴリズムは短時間で計算が終了することを示す。

Efficient Similarity Computation based on Computational Geometry Techniques

TETSUSHI MATSUI^{†1} and TAKEAKI UNO^{†1}

Edit distance (Levenshtein Distance) is one of popular measure for evaluating the distance of genome sequences, while it has some disadvantages such as straightforward algorithms take square time, decreasing the computation time without approximation is difficult, and it does not consider continuous similarity but considers matches similar even the matches are not continuous but skipping. In this paper, we propose a new similarity measure based on local similarity, and an efficient algorithm for it. Intuitively, edit distance is the maximum matches of the same letters without crossing. Contrary, our “fragment edit distance” matches similar short (fixed length) substrings of Hamming distance at most the given threshold. Usually, such similar substrings appear consequently, thus we reduce the data size by unifying consequent substrings, and propose an efficient algorithm utilizing computational geometry based computational technique. The computational experiments show the efficiency even for large scale data that could not be solved by the existing algorithms.

1. はじめに

近年のゲノムシーケンサの進歩により、ゲノム解読が終了した生物種の数是指数的に増加しつつある。増加するゲノムデータに対して、計算的な技術の進歩が追いつかないという事態が本格化しつつあり、計算機科学の大きな課題となっている。ゲノム配列の比較もそのような問題の一つである。一般にゲノム配列の比較には、編集距離 (レーベンシュタイン距離¹⁰⁾), あるいはある種のコスト関数を用いた編集距離の改良版が使われることが多い。2つの文字列の編集距離は、片方の文字列に対して、文字の挿入・削除・変更の操作を何回行うともう片方の文字に変換することができるかという、その最小回数で定義される。ゲノム配列のように、変化が文字単位で起こることが多いものには有効なモデルである。

編集距離の計算には、素朴な方法だと比較する配列の合計長の2乗に比例する時間がかかるため、染色体のような、100万を超えるような配列には直接的に適用できない。そのため、数々の高速化手法が提案されてきた^{4)-6),9)}。また、モデル自体にも改良が行われ、連続する挿入・削除に低い点数を与える、大域的な入れ替わりを考慮する、というものが考えられてきている。しかしながら、どれも巨大なデータでも計算ができるように、という方向性ではなく、より精緻にゲノムを比較したいという要求から出てきており、計算時間面での改良には結びついていない。

一般に、ゲノムの類似性には、局所的な類似性が大きな意味を持つ。例えば図1のような配列の組2つの類似性を見てみよう。どちらの組でも、文字列の長さは24であり、ハミング距離は12である。しかしながら、右側の組では中央に非常に類似度の高い部分があるため、この部分があるという意味で、2つの配列はより似ていると考えるのが自然であろう。これは、ある程度以上似ていないものは、どの程度似ていなくても同じように似ていないと考えたい、という方針に基づいており、ある程度以上似ていない部分の類似度の差が、似ている部分の類似度に影響を及ぼすのはおかしいだろうという考え方である。

似ている部分の定義として、ここではハミング距離が閾値 d 以下である、固定長 l の部分列を採用する。以下、そのような文字列を単に似た部分列と呼称する。編集距離を採用しないのは、そのような文字列の組が比較的容易に計算できること^{11),12)}、部分列の対応に明解でないところが出ないようにすること、ハミング距離の意味で類似する部分列が幾何的な構造を持つため、アルゴリズムを工夫することで計算の高速化が可能であるという点である。図2の上の図で、文字を結ぶ線は同一の文字 (の位置) の対応を表している。図2の下では、線は $l=3, d=1$ での似た部分列を表しており、編集距離を最小にする対応が太線で表されている (グラフアルゴリズム分野では、このような

対応は非交差的マッチングと呼ばれている)。本稿で提案する断片編集距離は、文字の対応の代わりに似た部分列を用い、効率的な計算のために置換のコストを倍加した距離である。

2つのゲノム配列は、例えば同じような機能を持つ遺伝子があるとき、その配列は局所的に似た構造を持つ。その似た構造の最初の l 文字同士が似た部分列であれば、それらの先頭を1文字ずつずらしていても、やはり似た部分列である、という状況になるであろう。つまり似た部分列は局所的に連続して現れる可能性が高いと考えられる。実際に $l=30, d=2$ としてヒトゲノムで似た文字列を見つけると、多くのものが10から20以上 (とぎれることなく) 連続して現れている。そのため、これら連続した連なりを一つにまとめてデータ化することで、データ量を大幅に削減できる。図2の下では、7対の対応が2つの連なりとして表現できる。ある程度以上の長さを持つ連なりのみを考慮するようにすることで、これらの連なりを列挙する時間も短縮できる^{11),12)}。

連なりをまとめて保持することは、データサイズの減少だけでなく計算時間の高速化にも貢献する。図2の下のように、編集距離に対応する非交差的マッチングの中には、各連なりの中の連続した部分、つまり1区間だけを含むようなものが存在する。この性質に着目し、片方の文字列を頭から尾に向かってスキャンして、連なりをどのようにつなげると良いかを計算するような動的計画法を構築することで、連なりの数 m の2乗に比例する計算時間のアルゴリズムが設計できる。さらにこのアルゴリズムに、2分木を使って各反復を高速化することにより、計算時間を $O(m \log m)$ に減少することができる。このようなテクニックは計算幾何学の分野でよく使われており、例えば線分交差列挙アルゴリズム⁷⁾ がその一例である。

計算実験の結果、人間の染色体のような長大な染色体に対しても、ある程度の長さの連なりのみに注目すれば、実用的な時間で連なりを見つけ、またその数も巨大にならないことを示した。また、断片編集距離の計算自体も短時間で終わることを示した。このような長大な配列に対しても、短時間で編集距離の要因を含む距離が計算できるようになることで、ゲノムの染色体単位での比較がある程度効率良くできるようになったと言えるだろう。

以下、次節で関連研究をまとめ、次に記法を解説する。4節で問題の定義と基礎的な2乗時間アルゴリズムの解説を行い、5節で高速アルゴリズムを解説する。6節で計算実験の結果を示し、最後にまとめる。

2. 関連研究と提案モデル

編集距離の計算は、比較する2つの文字列から作られるメッシュ状のネットワークにおける最短路計算と同値である。そのため、既存の最短路アルゴリズムの高速化技法

string (a):	ATATATATATATATATATATATAT	ATATATATATATATATATATATAT
string (b):	AGAGAGAGAGAGAGAGAGAGAG	GGGGGGATATATATATATATGGGGGG

図 1 全体的に同一文字がある文字列 (左) と局所的に似ている部分を持つ文字列 (右) の比較

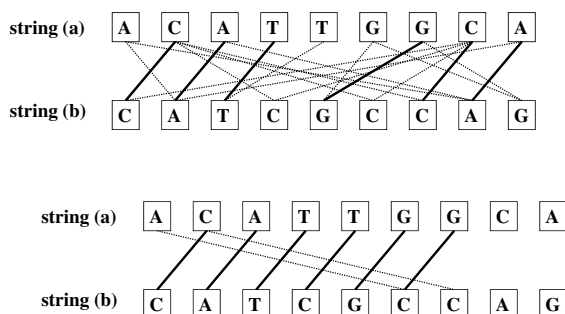


図 2 上:2 つの文字列の同一な文字の位置の対応と編集距離を達成する文字の対応 (太線), 下: $l = 3, d = 1$ のときの似た部分列の先頭位置の対応と最大の対応

が直接的に適用可能である。図 3 にネットワークの一例を示した。このネットワークの左下から右上の最短距離が編集距離になる。編集距離の場合、コストが全て定数であるので、ヒープの代わりにバケットを利用することで、ダイクストラ法の計算時間を $O(n^2 \log n)$ から $O(n^2)$ に落とせる。ここで n は比較する文字列の長さの和である。

これらのネットワークでは、左下から右上の対角線から離れた頂点は、左下からの距離が長くなる。そのため、もし編集距離が短ければ、ダイクストラ法は大きく離れた頂点は探索しない。直観的には、編集距離が文字列の長さの 1 割程度であれば、探索範囲は全体の 2 割程度になる。よって、似たゲノム配列を比較する場合には高速化が期待できる。また、始点と終点から同時に探索を行う両方向探索や、終点までの距離を下限を用いて探索範囲を減少させる A^* アルゴリズムを用いることで、さらなる高速化が行える⁸⁾。しかし、これらの高速化を行っても、実験的な計算時間はやはり $O(n^2)$ であり、巨大な配列では長い計算時間を必要とする。

一方、BLAST をはじめとするアルゴリズム⁴⁾⁻⁶⁾ では、短く完全に同一である部分列 (11 塩基程度) の組を列挙して、それらもとに局所的に高いスコアを持つ部分列の組を見つけ、それらをつないで解を構築している。この手法では、まったく似ていない部分は最初から無視されるため、その分の計算時間が短縮される。しかし、反復配列など、多くの完全一致する短い配列がある部分を持つ配列の比較では、非常に長い時間がかかる。また⁹⁾ のアルゴリズムは、全体を幾つかのブロックに分割し、ブロック同士の類似性をおおまかに計算することで、類似性の高い構造が含まれないようなブロックの組をあらかじめ候補から外し、計算時間を短縮している。こちらも、反復配列など繰り返し構造があると多くのブロック同士の類似性が高くなり、計算時間の短縮が難しくなる。

我々のアプローチでは、似た部分列を局所的に高い類似構造として用いる。似た部分列は^{11),12)} のアルゴリズムを用いて短時間で求められる。部分列の長さを 30 文字程度に設定することで、ハミング距離の閾値を 2 や 3 に設定しても、BLAST などが用いる 11 文字の完全一致する文字列の組よりはるかに数が少なくなり、計算の効率化が見込める。また、ある程度の長さを持つ部分配列が類似している場合、その中の多くの部分が類似部分列となることが多い。図 2 の文字列 (a) の 2 文字目から 9 文字目と文字列 (b) の 1 文字目から 7 文字目が高い類似性を示しており、その結果 5 組の似た部分列が連続している。そのため、これら連続して現れる似た部分列を、最初の似た部分列の位置と長さで表現してひとつにすることにより、データの削減を行える。図 2 の例では、(2,1,長さ 6) と、(1,6,長さ 2) という 2 つの三つ組みで記憶できる。この形式を似た部分列の連続表現と呼ぶ。

この論文で提案する断片編集距離は、非交差的な似た部分列の最大数で定義される。正確には、2 つの文字列の似た部分列の先頭位置 $(x_1, y_1), (x_2, y_2), \dots, (x_h, y_h)$ が与えられたとき、その断片編集距離は

$$N - \max\{|S| \mid S \subseteq \{1, \dots, h\}, (x_i < x_j \text{ and } y_i < y_j) \text{ or } (x_j < x_i \text{ and } y_j < y_i) \text{ for any } i, j\}$$

で定義される。ただし、 N は 2 つの文字列の長さの合計である。直観的には、挿入と削除のコストが置換のコストの半分であると解釈できる。似た部分列のパラメータを $l = 1, d = 0$ と設定した場合、置換のコストを 2 にした編集距離と等価になる。

非交差的な似た部分列の組は、図 2 のグラフでは非交差なマッチングになる。最大の非交差マッチングは $O(h \log h)$ 時間で求めることができるため、断片編集距離を求めるための計算時間は、単純な方法で $O(h \log h)$ となる。本稿では、連続する似た部分列の位置と長さ $(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_m, y_m, z_m)$ で与えられたとき、断片編集距離を $O(m \log m)$ 時間で計算するアルゴリズムを提案する。 h は m より数倍から数十倍大きくなることもあり、単純な手法と比べても大きな計算時間の改善が見込める。このアルゴリズムは各 x_i, y_i, z_i の整数性を仮定していないため、これらの数が実数値を取る場合でも同じ計算で計算が終了する。

3. 記 法

Σ を文字の集合とし、文字列を Σ の文字の列とする。文字列 S の i 番目の文字を $S[i]$ とし、 i を文字 $S[i]$ の位置とよぶ。また、文字列 S の長さを、 S に含まれる文字の数

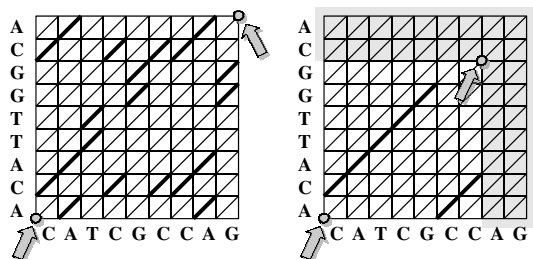


図3 左:2つの文字列の編集距離を求める際に用いるネットワーク. 右:断片編集距離を求める際に用いるネットワーク

とする. 文字列 S のある位置からある位置までの文字がつくる文字列を S の部分文字列とよび, 特に i 番目から j 番目の文字が作る部分文字列を $S[i..j]$ と表記する.

2つの同じ長さの文字列 S_1 と S_2 のハミング距離を, $S_1[i] \neq S_2[i]$ となる位置 i の数で定義する. S_1 と S_2 の編集距離を, S_1 に以下の3つの操作のいずれかを逐次的に行って S_2 に変換するために必要な最小の操作数とする.

- ・ $S[i]$ をある文字 a に変更する
- ・ 位置 i に文字 a を挿入する
- ・ 位置 i の文字を消去する

2次元上の2つの点 (x_1, y_1) と (x_2, y_2) に対し, それらを結ぶ線分を点の集合 $\{(\alpha x_1 + (1 - \alpha)x_2, \alpha y_1 + (1 - \alpha)y_2) | 0 \leq \alpha \leq 1\}$ で定義する. 特に (x, y) と $(x+l, y+l)$ を結ぶ線分を3つ組 (x, y, l) で表記する. 点 p の x 座標を $x(p)$, y 座標を $y(p)$ と表記する. また, 2つの線分の共通部分を, それら両方に含まれる点の集合とする. 線分の集合 $L = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_m, y_m, z_m)\}$ に対して, $x_{\min}(L) = \min\{x_i | (x_i, y_i, z_i) \in L\}$, $x_{\max}(L) = \max\{x_i + z_i | (x_i, y_i, z_i) \in L\}$, と定義する. 同様に $y_{\min}(L), y_{\max}(L)$ を定義する. これらは, L を含む最小の, 各辺が x 軸あるいは y 軸に平行な四角形の座標になる. 2次元平面上的パスを, 点 p_1 と p_2 を結ぶ線分, 点 p_2 と p_3 を結ぶ線分, $\dots$, 点 p_{k-1} と p_k を結ぶ線分の和集合とする. パスは, そのパスを構成する線分の端点の列 $(p_1, p_2, \dots, p_k)$ で表す. パス $P = (p_1, p_2, \dots, p_k)$ が単調であるとは, 任意の $1 \leq i \leq k-1$ に対して, (a) $x(p_i) = x(p_{i+1}), y(p_i) < y(p_{i+1})$, (b) $x(p_i) < x(p_{i+1}), y(p_i) = y(p_{i+1})$, (c) $x(p_i) + l = x(p_{i+1}), y(p_i) + l = y(p_{i+1})$ のどれかが成り立つことである.

4. 問題設定と基本解法

編集距離を計算するときには, 比較する文字列から非巡回的なネットワークを構築し, その有向ネットワーク上の最短路を動的計画法で求めることで計算を行う. ネットワークの構築例を図3の左に示した. 比較する文字列の長さを n_1, n_2 として, 各 $0 \leq i \leq n_1$ と $0 \leq j \leq n_2$

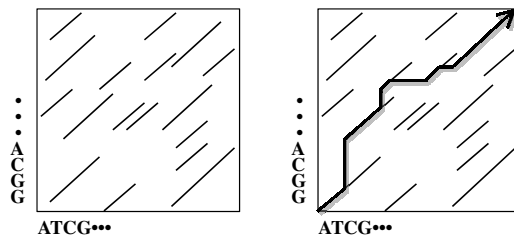


図4 左:斜め方向の線分の集合. 右:単調なパスを重ねたところ

に対応する頂点があり, $(i, j-1)$ から (i, j) への縦の枝, $(i-1, j)$ から (i, j) への横の枝, および $(i-1, j-1)$ から (i, j) への斜めの枝がある. 縦の枝と横の枝のコストは1であり, $(i-1, j-1)$ から (i, j) への斜めの枝のコストは, 1つめの文字列の i 文字目と2つめの文字列の j 文字目が一致するときに0, そうでないときには1となる. このネットワークにおける頂点 $(0, 0)$ から (n_1, n_2) までの距離が編集距離になる.

対して, 断片編集距離を求めるときに用いるネットワークでは, 縦の枝と横の枝のコストは変わらず, $(i-1, j-1)$ から (i, j) への斜めの枝のコストは, 1つめの文字列の i 文字目から始まる l 文字の部分文字列と, 2つめの文字列の j 文字目から始まる l 文字の部分文字列のハミング距離が d 以下の場合に0, そうでないときには2となる. これは, 枝のコストが2のものと1のものは0, 0のものは2に変更したネットワークでの最長パスの距離を, N から引いた値と等しくなる. 図3の右にネットワーク構築の例を示した.

このネットワークの幾何的な理解を考えよう. 上記のネットワークにおいて, 頂点の添え字を座標と見なし, 2次元に埋め込んだネットワークと考える. 連続するコスト0の斜め枝を線分と見なし, その線分の集合 $L = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_m, y_m, z_m)\}$ を問題の入力とする. 単調なパス P に対して, そのスコアを P の各線分と線分集合の各線分の共通部分を取って得られる線分全ての長さの和をとったものとする. 断片編集距離はこのスコアを最大化する P のスコアで定義される. 図4に, 線分集合とパスの重ねる様子を示した. 左が線分集合, 右がパスを重ねたもので, 線分集合の線分の, パスに含まれる部分の長さを足したものがスコアとなる.

単調なパスは無数に存在するが, ある種の標準的なもの限定しても, その中に必ず最大スコアを達成するものがある. 図4左のように, L の線分の途中から始まり, L の線分の途中, あるいは縦横への折れ曲がり点に到着するような縦横の線分が存在した場合, コストを変化させず縦横の線分をずらして, 始まりが L の線分の端点になるようにすることができる(図5参照). よって, P の縦線分の x 座標, および横線分の y 座標は必ず L の線分いずれかの x 座標, y 座標と一致するという条件を付けて良い. 同じ理由から, P の中の斜めの線分はかならず L のいずれか

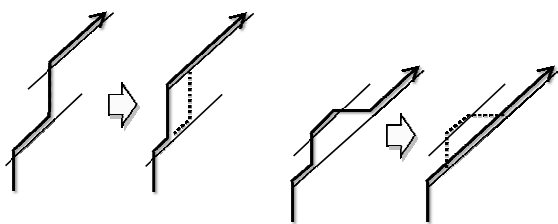


図5 左:斜め線分の途中に始まり途中で終わる縦線は左右にずらしてもスコアは変わらない。右:一度斜め線から離脱してまた戻るパスが最適になることはない。

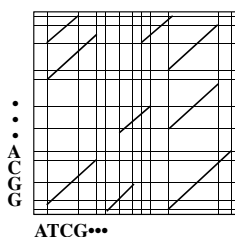


図6 縦横の移動を L の線分の端点に限定して得られるネットワーク

の線分の一部であり、その線分の左下の点から始まるとして良い。また、任意の L の線分 l に対して、 l の部分線分が2回以上現れるようなパスに対しては、その間の線分を取り除いて部分線分をつなげてしまうことでスコアを真に大きくできる(図5の右を参照)。よって、 l の部分線分は一回しか現れないと仮定できる。

これらの観察から、単調なパスを図6のようなネットワーク上に限定しても最適解が得られることがわかる。これにより、単純な動的計画法を用いて $O(|L|^2)$ 時間で最大スコアを求めるアルゴリズムが設計できる。 l がある程度(30文字程度)長ければ、 d/l が1割から2割であっても十分短時間で求められ、かつ L の大きさも小さくなる。そのため高速な計算が可能となる。計算時間と L の詳細については計算実験の節を参照されたい。

5. アルゴリズム

このような線分に対し、平面走査法というアルゴリズムが存在する。これは、 y 軸に平行な走査線を左から右へと動かし、各 x 座標での交わる線分を保持しながらなんらかのデータを更新し、最終的に解を得るという手法である。関数 $f_t(y)$ を、 $(x_{\min}(L), y_{\min}(L))$ を始点とし、 (t, y) を終点とするような単調なパスのスコアの最大値とする。我々は、関数 $f_t(y)$ を保持するデータ構造を構築し、走査線的位置(x 座標) t を $x_{\min}(L)$ から $x_{\max}(L)$ へ動かし、そのデータ構造を更新する。最終的に、 $t = x_{\max}(L)$ となった際に、 $f_t(y_{\max}(L))$ が問題の解となる。

線分 $l_i = (x_i, y_i, z_i)$, $l \in L$ に対して、 l_i のスコア関数 $f_t(l_i, y)$ を、 $(x_{\min}(L), y_{\min}(L))$ を始点とし、 (t, y) を終点とするような単調なパスの中で、 l_i の一部を含み、かつ l_i 以後の線分は全て縦線か横線であるようなもののみを考え、それらの中でのスコアの最大値とする。ただし、 $y < y_i$

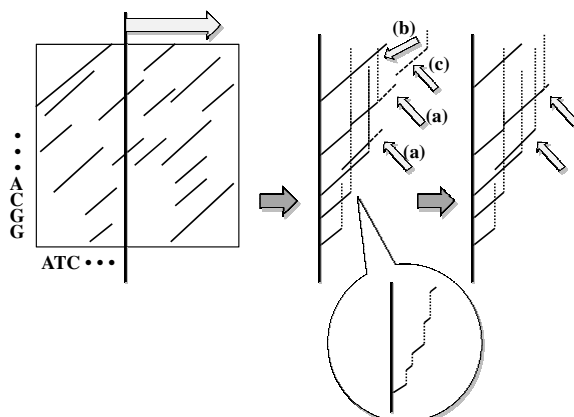


図7 線分集合と走査線(縦の直線): 走査線の左の線分のスコア関数が右にを使って表示されている

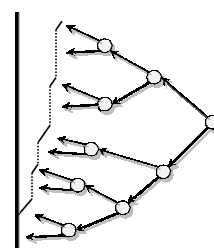


図8 見える線分の並びを2分木を使って保持する様子

か $x < x_i$ が成り立つときは $f_t(l_i, y) = 0$ であるとする。 $f_{x_i}(y_i)$ を用いると、 $t \geq x_i$ に対して、

$$f_t(l_i, y) = \begin{cases} f_{x_i}(y_i) + (y - y_i) & y \leq y_i + \min\{t - x_i, z_i\} \\ f_{x_i}(y_i) + (t - x_i) & y \geq y_i + \min\{t - x_i, z_i\} \end{cases}$$

となり、スコア関数は1本の斜め線分と1本の水平な半直線からなることがわかる。それら2つが交わる y を l_i の角よぶ。また、 $f_t(y) = \max\{f_t(l_i, y)\}$ である。直観的には、 $f_t(y)$ は $f_t(y_i)$ の一番上に出ている部分を集めてできる関数(上側エンベロープ)である。図7に例を示した。左図中央の長い縦線が走査線であり、その周辺、および左側にある線分のスコア関数を右側に図示している。スコアの軸が横向きであることに注意されたい。各スコア関数の、走査線より右側に対応する部分は斜めの点線で表されている。

ある l_i の角 y が $f_t(y) = f_t(l_i, y)$ を満たすとき、 l_i およびそのスコア関数は見えるといい、そうでないときは見えないという。また、 t が $x_i \leq t \leq x_i + z_i$ を満たすとき、 l_i はアクティブであるという。 t が変化すると、アクティブな l_i に関しては関数 $f_t(l_i, y)$ が変化するが、アクティブでなければ変化しない。本稿のアルゴリズムは、 $f_t(y)$ を表すデータ構造を構築し、 t の変化に合わせてそれを更新する。 $f_t(y)$ は t が L のいずれかの線分の端点の x 座標と同じである場合のみ計算すればよいが、それでも関数を直接的に計算したのでは $O(|L|^2)$ の時間がかかってし

まう。そこで本稿では、見える線分とその角の並び方のみを保持することにする。見えるスコア関数のみの上側エンベロープは $f_t(y)$ と一致する。

l_i の角は $y_i + \min\{t - x_i, z_i\}$ で与えられる。また、その角に接続する半直線と線分も、 t が定まれば一意に、定数時間で計算できる。よって、 t と線分の角の並びが定まれば、任意の y に対して、 $f_t(y)$ がどの線分によって達成されているか、つまりどの i に対して $f_t(y) = f_t(l_i, y)$ が達成されているのかを調べることができる。この作業は、見える線分の並びが 2 分木の葉に保持されていれば $O(\log |L|)$ 時間で計算できる。2 分木で並びを保持する構造を図 8 に示した。

t を増加させると、アクティブな線分の角が動く。これにより、線分の見え方が変化する。スコア関数の増大率はどれも同じであるので、アクティブな線分が見えなくなることはなく、またアクティブでない線分が見えるようになることもない。どの線分も、必ずアクティブになったときは見えるため、線分の状態変化のパターンは (1) アクティブでない (2) アクティブかつ見えている (3) アクティブでないが見えている (4) アクティブでなく見えないうものしかない。ただし、(1) と (4) の状態を経ないことはありうる。角の並びの変化は (2) と (4) で起こるため、これらの変化を効率良く扱う方法を考える。

まず、(2) を考える。これは、ある線分 l_i に対して $t = x_i$ となり、線分 l_i が見えるようになる場合である。これは、 y_i が角の並びのどこに挿入されるかを調べれば良く、2 分木により $O(\log |L|)$ 時間でできる。次に (4) を考える。(4) の変化を扱うため、各見える線分に対して、どこまで t が増大すると隣の線分のスコア関数を覆い、見えなくなるかを記憶しておく。そして、線分の並び順が変わったときに、この情報を更新する。並び順が変わる線分は高々 2 つであるので、この更新は定数時間でできる。

(2) の変更が起こる t の位置と (4) で記憶する位置 t の両者をイベント位置と呼ぶ。我々のアルゴリズムはイベント点を x 座標の小さい順にたどり、新たなイベント点の計算を行うことで線分の並び順を更新していく。これにはイベント点を蓄えるヒープを用い、イベント点一つあたり $O(\log |L|)$ 時間で行う。これにより、イベント点を x 軸の小さい順に、全て漏らさずにスキャンすることができる。

定理 1 m 本の平行な線分の集合に対して、その断片編集距離は $O(m \log m)$ 時間で計算できる。

6. 計算実験

この節では、前節にて解説した手法を用いて計算を行った計算機実験の結果を示し、手法の効率性を検証する。まず、比較する文字列から連続する似た部分列を取り出すことができるか、その計算実験の結果を示す。

図 9 に文字列から似た部分列の線分表現を得るため

の時間と得られる線分表現の大きさを計測した結果を示す。文字列としてヒトとマウスの染色体配列を用いた。実験で用いたコンピュータの CPU は Core 2 Duo E8400 (3GHz)、2GB のメモリを搭載したものである。OS は Linux であり、言語は C、コンパイラは gcc を用いた。実験で用いたアルゴリズム SACHICA^{(11),(12)} の実装は著者のホームページからダウンロードできる (<http://research.nii.ac.jp/~uno/index.j.html>)。このプログラム自体は並列計算が可能であるが、今回の実験では並列計算は用いていない。1000 万塩基程度であれば 2-3 分で計算が終了し、線分表現の大きさも、もとの配列の大きさに対して数十倍程度に収まっている。しかし、1 億塩基を超えるような配列では 1 時間以上の計算時間がかかってしまっており（計算は 1 時間で打ち切っている）、線分表現も巨大になると予想される。SACHICA は見つける線分の長さの下限を設定すると高速化できる（見つけ損ないは発生しない）。これを利用して、長さ 30 以上の線分のみを見つけた結果が図 10 である。計算時間が 10 分の 1 以下になり、線分数も減少している。SACHICA が見つけたが長さが足りなくて出力しなかったものも合わせると、線分の 1/4 程度が発見できている。この程度の時間であれば、十分実用に耐用できると考えられる。

次に、今回提案するアルゴリズムのパフォーマンスを検証する。こちらの実験で用いたコンピュータの CPU は Core i5 (2.53 GHz)、8 GB のメモリを搭載したものである。OS は Mac OS X であり、言語は Python を用いた。実験で用いたアルゴリズムの実装は著者のホームページからダウンロードできる (<http://research.nii.ac.jp/~uno/index.j.html>)。アルゴリズムの実装に必要な 2 分木にはいわゆる AA ツリー³⁾を採用した。

入力する問題は、ヒト 1 番とマウス 1 番染色体に対して、 $l = 30, d = 3$ として SACHICA を動かして得られた長さ 30 以上の線分を用いた。この中からランダムに k 個を選んでインスタンスを複数作成し、問題サイズの増加に対するパフォーマンスの変化を調べた。また、比較対象として、2 分木を用いずに見える線分を保持する、最悪 2 乗時間のアルゴリズムを実装した。なお、通常の編集距離の計算は、計算時間が天文学的に長くなるため除外した。問題サイズの増加に関する比較結果を図 11 に示す。

ところで、線分をランダムに選択した場合、線分数 k が少ないところはデータが疎になり、逆に多いところでは比較的密になる。このことは実行時間に影響を与えないはずであるが、念のため線分の分布を反映した選択方法として原点に接する一辺の長さ L の正方形に収まる線分を抜き出した系列を用意し、上と同様に比較したものが図 12 である。この方法で生成したデータは似た構造を持つと思われるが、そのようなデータでも、提案手法が効率良く計算

配列名と長さ	d	計算時間	線分数
ヒト 22 番とマウス 19 番染色体 35,058,832 : 61,342,431	$d = 1$	210 秒	25,831,400
	$d = 2$	472 秒	43,535,796
	$d = 3$	1,942 秒	75,091,394
ヒト 11 番とマウス 11 番染色体 131,246,336 : 121,843,857	$d = 1$	1,200 秒	217,761,847
	$d = 2$	3,032 秒	358,138,558
	$d = 3$	-	-
ヒト 1 番とマウス 1 番染色体 226,213,776 : 197,195,433	$d = 1$	2,979 秒	592,333,718
	$d = 2$	-	-
	$d = 3$	-	-

図 9 $l = 30$ としたときの計算時間と出力された線分数 (線分表現の大きさ)

配列名と長さ (塩基数)	d	計算時間	長さ 30 以上の線分数	全ての発見線分数
ヒト 22 番とマウス 19 番染色体 35,058,832 : 61,342,431	$d = 1$	30 秒	126,933	6,345,507
	$d = 2$	53 秒	333,599	11,391,236
	$d = 3$	162 秒	774,246	19,613,048
ヒト 11 番とマウス 11 番染色体 131,246,336 : 121,843,857	$d = 1$	119 秒	973,468	52,456,915
	$d = 2$	232 秒	2,382,445	94,471,717
	$d = 3$	780 秒	5,203,084	158,582,799
ヒト 1 番とマウス 1 番染色体 226,213,776 : 197,195,433	$d = 1$	250 秒	2,902,687	144,209,935
	$d = 2$	602 秒	6,999,062	257,702,842
	$d = 3$	1817 秒	15,68,298	2,491,360,751

図 10 長さ 30 以上の線分を計算した場合の計算時間と出力された線分数

を行っていることがわかる。

最後にこれらをまとめた両対数グラフを図 13 に示す。グラフ中 proposed と名付けた方の 2 本の線が今回の提案アルゴリズムの実行時間の推移である。ランダムに選択したものの方が若干時間が掛かっているが、いずれのケースも比較アルゴリズムの最悪 2 乗時間の方法より明らかに高速である。回帰直線の傾きを求めると、提案法は 1.13 (ランダムの場合), 1.17 (小さい方から取った場合) となり, 1 より少し大きい値が得られる。このアルゴリズムは $O(m \log m)$ 時間であるから 1 より少し大きい値は予想通りである。一方比較アルゴリズムはいずれの場合も傾きが 2.00 となり 2 乗時間であることが確かめられた。

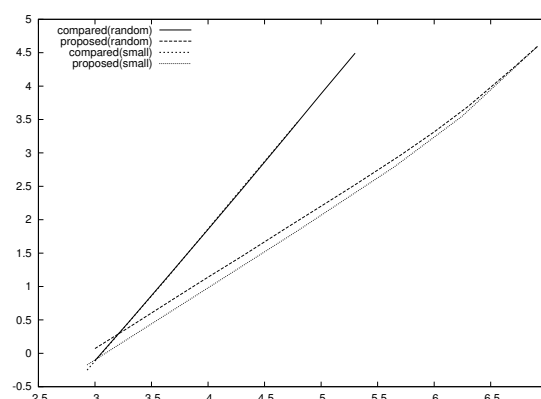


図 13 計算時間の比較 (両対数)

7. ま と め

本稿では、長さが l , ハミング距離が d 以下の部分文字列の組 (似た部分列) を利用した文字列の距離を導入し, また, その距離を $O(m \log m)$ 時間で計算するアルゴリズムの提案を行った。ここで m は, 似た文字列の連続の数である。計算実験により, $l = 30$, $d = 2, 3$ 程度に設定した場合, ある程度連続する似た文字列の組が短時間で全て見つけれられること, その数も莫大にならないことが確認され, また提案したアルゴリズムが短時間で終了することを示した。

今後の拡張として, 連続した似た部分列それぞれに異な

る重みを与えること, 角度の異なる線分を扱えるようにすることが上げられる。前者は似た部分列の精度や重要度の考慮, 後者は音楽データなどでのスピード (テンポ) の異なる似た部分列の類似性を扱えるようになる。今回の解法は傾きや重みが等しいことを深く利用しているため, 新たな計算手法を開発する必要があるだろう。

謝 辞

当研究の一部は科学技術振興機構さきがけによって行われた。

線分数 (k)	提案アルゴリズム (秒)	比較アルゴリズム (秒)
1,000	1.18	0.78
2,000	2.45	2.96
5,000	6.57	18.04
10,000	13.82	71.58
20,000	28.47	287.33
50,000	76.42	1846.84
100,000	159.48	7698.19
200,000	334.36	31163.04
500,000	916.08	-
1,000,000	2064.62	-
2,000,000	4978.10	-
5,000,000	18715.30	-
8,268,769	39842.49	-

図 11 ランダムなデータでの計算時間の比較

一辺の長さ (L)	線分数 (k)	提案アルゴリズム (秒)	比較アルゴリズム (秒)
2,000,000	852	0.67	0.56
5,000,000	4226	3.78	12.66
10,000,000	15367	15.15	174.36
20,000,000	61882	68.11	2886.81
50,000,000	445782	621.57	-
100,000,000	1762491	3535.08	-
226,213,776	8268769	39842.49	-

図 12 小さい方から抜き出したデータでの計算時間

参 考 文 献

- 1) S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman, Basic local alignment search tool, *J. Mol. Biol.* **215**, pp. 403–10, 1990.
- 2) S. F. Altschul, T. L. Madden, A. A. Schaffer and J. Zhang, Z. Zhang Z, W. Miller, D. J. Lipman, Gapped BLAST and PSI-BLAST: a New Generation of Protein Database Search Programs, *Nucleic Acids Res.*, **25**, pp. 3389–3402, 1997.
- 3) A. Anderson, Balanced Search Trees Made Simple, in Proc. Workshop on Algorithms and Data Structures, pp. 60–71, 1993.
- 4) S. Batzoglou, L. Pachter, J. P. Mesirov, B. Berger and E. S. Lander, Human and Mouse Gene Structure: Comparative Analysis and Application to Exon Prediction, *Genome Res* 2000, **10**, pp. 950–958, 2000.
- 5) N. Bray, I. Dubchak, L. Pachter, AVID: A Global Alignment Program, *Genome Res* 2003 **13**, pp. 97–102, 2003.
- 6) M. Brudno, C. B. Do, G. M. Cooper, M. F. Kim, E. Davydov, E. D. Green, A. Sidow and S. Batzoglou, LAGAN and Multi-LAGAN: Efficient Tools for Large-scale Multiple Alignment of Genomic DNA, *Genome Res* 2003 **13**, pp. 721–731, 2003.
- 7) B. Chazelle and H. Edelsbrunner, An Optimal Algorithm for Intersecting Line Segments in the Plane, *Journal of the ACM* **39**, pp. 1–54, 1992.
- 8) T. Ikeda and H. Imai, Fast A* Algorithms for Multiple Sequence Alignment, *Genome Informatics Workshop* **94**, pp. 90–99, 1994.
- 9) R. Nakato and O. Gotoh, Cgaln: Fast and Space-efficient Whole-genome Alignment, *BMC Bioinformatics* **11**, pp. 224, 2010.
- 10) V. Levenshtein, Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones (original in Russian), *Russian Problemy Peredachi Informatsii* **1**, pages 12–25, 1965.
- 11) T. Uno, An Efficient Algorithm for Finding Similar Short Substrings from Large Scale String Data, *PAKDD 2008*, Lecture Notes in Artificial Intelligence 5012, pp. 345–356, 2008.
- 12) T. Uno, Multi-sorting Algorithm for Finding Pairs of Similar Short Substrings from Large-scale String Data, *Knowledge and Information Systems*, 10.1007/s10115-009-0271-6, 2010.