

ネットワーク上の軌跡データに対する時間制約付き二点間経路の列挙

小出 智士[†] 吉村 貴克[†] 肖 川^{††} 石川 佳治^{††}

^{††} 名古屋大学 情報科学研究科 〒464-8603 愛知県名古屋市千種区不老町

[†] 豊田中央研究所 〒444-0416 愛知県長久手市横道 41-1

E-mail: [†]{koide,yoshihlmura}@mosk.tytlabs.co.jp, ^{††}chuanx@nagoya-u.jp, ^{††}tishikawa@is.nagoya-u.ac.jp

あらまし 本研究では与えられたネットワーク上の二点間の経路に関する系列マイニング問題を考察する．具体的には軌跡データベース中において、 $u-v$ を通過する部分軌跡として観測され、かつその軌跡の発生時刻が与えられた時間幅に含まれるような経路のみを列挙する．この問題は最頻経路問題 [9] を低頻度経路にまで一般化した問題とみなすことができ、実用上は抜け道案内などの応用が考えられる．この問題に対して、我々は接尾辞配列を応用したデータ構造を考え、これによって対象となる経路の列挙を効率的に枝刈りしながら列挙できることを示す．提案手法を実データに適用し、既存の系列マイニングアルゴリズム (Traj-PrefixSpan [11]) などと比較して数百倍の高速化が可能であることを示す．

キーワード 時空間データ, 系列マイニング, データ駆動型ナビゲーション

1. はじめに

自動車などの移動オブジェクトから得られる軌跡データは年々増加している．軌跡データの代表的な活用方法の一つとしてデータ駆動型のナビゲーションがある．従来の典型的なナビゲーションではコスト最小化する経路を選択する．この場合、平均所要時間や距離、料金などの事前に与えられたコスト関数が用いられる．一方、データ駆動型のナビゲーションにおいては与えられた経路の所要時間分布の予測 [3] や二点間の最も利用頻度の高い経路の抽出 [2], [9] など、より高度な処理を行うことを目的としており、このために大量の軌跡データを効率的に処理することが不可欠である．

本研究ではこのようなデータ駆動型のナビゲーションに関する問題を取り扱う．より具体的には、与えられた二点間の経路の列挙問題を取り扱う (具体的な問題設定は次章で行う)．一般に、ネットワーク上の二点間の経路の場合の数は組合せ論的な膨大な数になってしまう．これらを数え上げるような研究も存在するが、我々は軌跡データベース中に実際に発生した経路のみを列挙する問題を考える．これによって現実的に発生し得る経路のみを列挙することができ、ナビゲーションとしても有用な経路が抽出できる．

データベース分野においては軌跡データからナビゲーションとして有用な経路を抽出する研究が数多く存在する．Chen et al. [2] では most popular route (MPR) 問題を考えた．これはグラフのノード間の遷移確率^(注1)をデータから推定し、そのグラフ上でダイクストラ法に類似の経路探索アルゴリズムを用いることで、二点間の経路のうちで確率が最大になるような経路を見つける、というものである．Luo et al. [9] では time-period-based most frequent path (TPMFP) 問題を

考えた．MPR が時間の要素を考慮していなかったのに対して、TPMFP では対象とする時間幅を問い合わせ時に指定することができる．このような拡張によって、事前に遷移確率などの統計量を計算することができないため、オンライン処理の割合が増加し、より高度な索引構造やアルゴリズムが必要とされる．

これらの研究はいずれも最も高頻度の (もしくは人気のある) 経路を見つける、という問題設定になっており、複数の経路を抽出することはできない．特に、我々は低頻度の経路の検索に着目している．このような低頻度の経路の中にはエキスパートドライバー (タクシー運転手) などが用いる抜け道が含まれている可能性があり、そのような経路を提示することでより高度なナビゲーションが可能になることがその理由である．複数の経路を見つけるためには、例えば MPR における経路探索アルゴリズムを top- k に拡張するということも考えられる．しかしながらこの方法ではたとえ k を大きく取ったとしても、低頻度の経路を見つけることが困難である^(注2)．一方、我々のアプローチでは、実際に発生した経路のみを列挙する．列挙される経路の数は高々データベースに格納されている軌跡の数程度であるため、低頻度の経路も問題なく列挙することができる．

本研究では低頻度経路を含む複数経路をデータベースから抽出する問題を考える．我々は上記で述べた従来研究 (MPR, TPMFP) の課題を解決するために、新しい問題 (時間制約付き二点間経路の列挙問題) を定義し、その問題のためのアルゴリズムを提案し、実データを用いてその有効性を示す．我々の問題設定は一種の系列マイニング問題であると考えられるため、PrefixSpan を修正したアルゴリズムをベースラインとし、提案アルゴリズムがそれよりも数桁高速に経路を列挙できること

(注1)：厳密には確率ではないため、[2] では popularity と呼んでいるが、ここでは説明の簡略化のために「確率」と記述した．

(注2)：その理由としては MPR および TPMFP はデータ量が少ないという仮定に基づいて、高頻度経路を見つけるためのある種のスムージング処理を行っているが、この処理によって最高頻度に近い経路が数多く存在してしまうためである (これはちょうど top- k 最短経路を見つけるためのダイクストラ法で遠回り経路を見つけるのが困難であることと類似している)

を示す.

2. 問題定義

2.1 軌跡データモデル

本研究では移動オブジェクトは有向グラフ $G = (V, E)$ 上を移動するものとする. その軌跡は $T = (p, t)$ で表現され, $p = p_1 p_2 \cdots p_n$ と $t = t_1 t_2 \cdots t_n$ はそれぞれ経路および時刻を表す長さ n の配列である. ここで $p_i \in E$ は道路リンクであり, t_i は道路リンク p_i を通過し終えたときの時刻である. また, $p_{i:j}$ は添字 i から j までを含む p の部分経路であると定義する. $T.p$ および $T.t$ は軌跡 T の経路および時刻の配列を意味するものとする. このような軌跡の集合 $\mathbb{T} = \{T_k\}$ を軌跡データベースと呼ぶ.

2.2 時間制約付き二点間経路の列挙問題

軌跡データベース $\mathbb{T}$ が与えられているとする. 前章で述べたように我々の問題は, 定性的には $\mathbb{T}$ に現れるような二点間の経路を列挙する問題であり, 形式的には以下のように定義される.

軌跡データベース $\mathbb{T}$ の軌跡のうちで, ある時間幅 I の中で, ある経路 P を部分経路として走行したような軌跡の集合 $S(P, I) \in \mathbb{T}$ を考える^(注3). すなわち,

$$S(P, I) = \{T \in \mathbb{T} \mid \exists i, j \text{ s.t.}$$

$$T.p_{i:j} = P \text{ and } T.t_i \in I \text{ and } T.t_j \in I\} \quad (1)$$

である. したがって, $|S(P, I)| > 0$ であるということは少なくともデータベース中の一つの軌跡が経路 P を時間幅 I の中で通過した, という意味である. 本研究ではこの軌跡数をサポートと呼び, $\text{supp}(P, I) = |S(P, I)|$ と定義する. さらに, 二つの道路リンク $u, v \in E$ 間の可能なすべての経路の集合を $\Pi(u, v)$ と書く. これらの記号の準備のもとで, 時間制約付き二点間経路の列挙問題は以下のように定義される.

$$W(u, v, I, \xi) = \{P \in \Pi(u, v) \mid \text{supp}(P, I) > \xi\}. \quad (2)$$

ここで ξ はサポートの最小値を定める定数であり, この値が大きいほど, 高頻度の経路のみが列挙されることになる. なお, 式 (2) の定義 $T.p_{i:j} = P$ より, 軌跡 T は経路 P を部分経路として走行していればよいことを強調しておく. また, 現実的な仮定として $\Pi(u, v)$ は u, v が二回以上出現しない (すなわち u および v は P の最初と最後のみ現れる) ものとする. これはアルゴリズムの記述をシンプルに保つための技術的な制限でもあるが, ループをもつ経路を発見することは応用のシーンでは意味が無いと考えられるため, 現実的な観点からも妥当である.

式 (2) を定義通りに求めようとすると $\Pi(u, v)$ に属するすべての経路 P に対して $\text{supp}(P, I) > \xi$ かどうかをチェックすればよいことになる. 一般に $\text{supp}(P, I)$ は高速に求めることができることが知られているが [15], $\Pi(u, v)$ のサイズは極めて

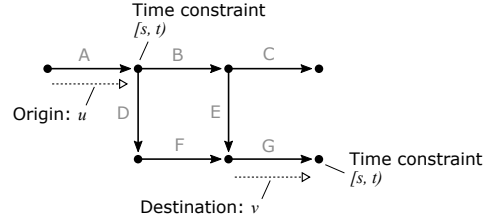


図1 本研究の問題設定: $W(u, v, I, \xi)$. 軌跡データベース $\mathbb{T}$ において, 時間幅 $I = [s, t]$ の間に $\xi + 1$ 回以上走行された, $u-v$ 経路を列挙する.

大きい^(注4), すべての $P \in \Pi(u, v)$ をチェックすることは現実的ではなく, より効率的なアルゴリズムが必要になる.

図1に例を示す. $u = A$ かつ $v = G$ とする. 与えられた時間幅 $I = [s, t]$ において, 軌跡データベース中に経路 $P_1 = ABEG$ が10回, 経路 $P_2 = ADFG$ が3回, 部分経路として出現したとする. $\xi = 5$ のケースを考えると, $\text{supp}(P_1, I) = 10 > \xi$ があるので $P_1 \in W(u, v, I, \xi)$ であるが, $\text{supp}(P_2, I) = 3 \leq \xi$ があるので, $P_2 \notin W(u, v, I, \xi)$ である. すなわち, この例においては $W(u, v, I, \xi) = \{P_1\}$ となる. また, より小さい最小サポート $\xi = 2$ を用いれば, $W(u, v, I, \xi) = \{P_1, P_2\}$ となる.

3. 方法

本章では $W(u, v, I, \xi)$ を求めるためのアルゴリズムとして, ベースラインである PrefixSpan ベースのアルゴリズムおよび提案アルゴリズムについて記す. はじめにこれらのアルゴリズムで共通に用いるデータ構造 (時間情報付き転置索引) を示し, 続いてそれぞれのアルゴリズムの詳細を示す.

3.1 時間情報付き転置索引

ネットワーク上の軌跡データを索引化する方法としては FNR-tree [6], MON-tree [4], NETTRA [8] などが知られている. これら方法では, 軌跡を文書と考え, 転置索引と類似の方法でデータを格納する. 具体的には軌跡データ T_k を軌跡 ID, 道路リンク ID, 時刻の三つ組 (k, p_i, t_i) の列であると考え. そして道路リンク ID p_i を単語とみなして転置索引 Φ_{p_i} に (k, t_i) を格納していく. すなわち転置索引 Φ_{p_i} には道路リンク p_i を通過したすべての軌跡の軌跡 ID およびその通過時間が記録されている. そしてこれらの転置索引を時刻 t_i に関して B^+ -tree や R-tree で索引化する^(注5). このような索引化によって, 以下の性質が得られる.

性質1 道路リンク e を時間幅 I に通過した軌跡のレコードの集合 $\{(k, t)\}$ を, 転置索引 Φ_e から高速に取り出すことができる (B^+ -tree の範囲問い合わせを用いる). この操作を $\text{RangeQuery}(e, I)$ と書く.

(注4): 例えばサイズ 32 のグリッドネットワークを考えると, その右上から左下への可能な経路の数は (最短経路だけを数え上げたとしても) 2^{60} 以上の経路が存在する.

(注5): 実際には FNR-tree や MON-tree では時刻の索引化に R-tree を用いている. これは道路リンク p_i に流入した時刻とそこから流出した時刻の幅を索引化するためである. 本研究では流出した時刻のみを B^+ -tree で索引化することが, 必要に応じて R-tree を用いてもよい.

(注3): この $S(P, I)$ は完全経路クエリと呼ばれる問い合わせの結果と一致する (例えば [15]).

NETTRA [8] では上記の転置索引に以下のように定義されるハッシュ値 h を付与し、 (k, t, h) という型の転置索引を用いる。軌跡 T における経路 $p = p_1 p_2 \dots p_n$ に対して、ハッシュ値の列 $h = h_1 h_2 \dots h_n$ が計算されるが、各ハッシュ値 h_i は $p_1 \dots p_{i-1}$ までの走行距離、すなわち道路リンク e の物理的な長さ $l(e)$ を用いて

$$h_i := \sum_{j=1}^{i-1} l(p_j) \quad (3)$$

と定義される (ただし $i = 1$ のときは $h_1 = 0$ とする)。このハッシュ値は以下のような性質を持つ:

性質 2 $p_{i:j-1} = P$ ならば $h_i - h_j = \sum_{e \in P} l(e)$ 。ここで右辺は経路 P のトータル長さを意味している。

性質 3 また、 (p_i, t_i, h_i) が (p_j, t_j, h_j) の直後に出現する、すなわち $i - 1 = j$ であることを $h_i - h_j = l(p_j)$ でチェックすることができる。

我々はデータサイズが大きい場合を考え、時間情報付き転置索引をディスク上に格納することにする。したがって時間情報付き転置索引は Postgresql などの RDBMS として実装することが可能である。

3.2 ベースライン手法: PrefixSpan

式 (2) は頻度が ξ より大きな経路パターンを見つける問題であり、系列マイニング問題の一種とみなすことができる。系列マイニング問題の代表的なアルゴリズムの一つとして PrefixSpan がある。PrefixSpan アルゴリズムには多くの変種が存在するが、我々は Traj-PrefixSpan [11] を修正したアルゴリズムを考える。Traj-PrefixSpan は PrefixSpan と異なり、発見される頻出系列を部分文字列に限定する、すなわちギャップのあるパターンは禁止される。これによって空間的に連続した道路リンクからなるパターンのみが発見される。

オリジナルの Traj-PrefixSpan をそのまま適用した場合、 $W(u, v, I, \xi)$ をを見つけることはできない。これはオリジナルの Traj-PrefixSpan が (1) 時間幅の条件を考慮しない (2) 開始道路リンク u および v を限定していない、ことによる。そこで我々はこれらの要素を考慮可能とした、修正 Traj-PrefixSpan アルゴリズムを考え、これをベースラインアルゴリズムとした。

修正 Traj-PrefixSpan アルゴリズムを Algorithm 1 に示した。 $W(u, v, I, \xi)$ の計算のためには $PrefixSpan(u, \mathbb{T}|_u)$ を呼び出す。ここで $\mathbb{T}|_u = RangeQuery(u, I)$ である。このアルゴリズムの 1 行目では射影データベース $\mathbb{T}|_P$ のサイズが最小サポートサイズ ξ よりも小さい場合、もしくは P の経路長が θ よりも長い場合に探索を打ち切っている。この θ に関する条件は式 (2) では定められていないが、この条件を付与しない場合、アルゴリズムの終了が極端に遅くなる場合があるため、追加的に付与している ($\theta = \infty$ とおけば定義通りになる)。3, 4 行目では、経路 P の最後の道路リンクが目的地 v であった場合に、経路を出力している。そうでない場合は現在の経路 P を伸ばして探索を続ける。5 行目の $OutGoingEdges(P.last)$ は経路 P の最後の道路リンクに隣接した道路リンクの集合を意味している。7 行目では Pe -射影データベースを求めるために

Algorithm 2 (*Projection*) を呼び出している。 Pe -射影データベースを求める操作においては、*RangeQuery* を用いて道路リンク e を時間幅 I の間に通過した軌跡を転置索引から検索し、すでに求められている P -射影データベースとの join 操作を行うことで求めている。ここで、 Δ -join は join 操作の際に、 P と e が正しい順序でかつ連続して出現することを保証するための join 操作である。このために前節で示した性質 3 を用いる。すなわち Pe というパターンが出現するためには $\mathbb{T}|_P$ のハッシュ値と、 $\mathbb{T}|_e$ のハッシュ値の差が道路リンク e の長さ $l(e)$ と一致すれば十分である。そして、Algorithm 1 の最後 8 行目では PrefixSpan を再帰的に実行することで、より長いパターンの探索を行っている。

このアルゴリズムは最初の道路リンク u から深さ優先探索を行い、到達したネットワークの各エッジ e において *RangeQuery* および join 操作を行う。したがって非常に多数回の *RangeQuery* が呼び出されることになり、非効率である。

Algorithm 1: ベースラインアルゴリズム (修正 Traj-PrefixSpan): $PrefixSpan(P, \mathbb{T}|_P, I)$

```

1 if  $|\mathbb{T}|_P| < \xi$  or  $dist(P) > \theta$  then
2   return //  $\mathbb{T}|_P$  のサイズが小さい /  $P$  が長過ぎる場合に終了
3 if  $P.last = v$  then
4   yield  $P$  and return //  $v$  に到達した場合、パターンを出力する
5 for  $\forall e \in OutGoingEdges(P.last)$  do
6    $P' \leftarrow Pe$  //  $P$  に  $e$  を追加した経路
7    $\mathbb{T}|_{P'} \leftarrow Projection(e, \mathbb{T}|_P, I)$  // 射影 DB を求める
8    $PrefixSpan(P', \mathbb{T}|_{P'}, I)$  // 再帰呼び出し

```

Algorithm 2: Pe -射影データベースを求める: $Projection(e, \mathbb{T}|_P, I)$

```

1  $\mathbb{T}|_e \leftarrow RangeQuery(e, I)$ 
2 return  $\Delta$ -join( $\mathbb{T}|_P, \mathbb{T}|_e, l(e)$ )

```

3.3 提案手法

本節では我々の提案アルゴリズムについて述べる。前節で述べた修正 Traj-PrefixSpan アルゴリズムは、 $W(u, v, I, \xi)$ を系列マイニング問題として捉えたときには自然な考え方であるが、多数回の *RangeQuery* を伴うため、非効率であった。提案手法ではこれを回避するためにいくつかの工夫を行うが、その基本的な考え方はシンプルであり、以下のように要約される 2 フェーズのアルゴリズムである。

(1) (候補生成) u および v を時間幅 I の間に両方通過したような軌跡を抽出し、これを候補軌跡集合 Z とする。

(2) (経路復元) 各候補軌跡に対して、 u から v までの部分軌跡を実際に復元し、復元された経路集合 $\{P_i\}$ の中から出現頻度が ξ より大きい経路の集合を $W(u, v, I, \xi)$ とする。

以下ではこれらの 2 フェーズの詳細について述べた後、高速化手法を提案する。最終的なアルゴリズムについては Algorithm 5 に示す。

a) 候補生成

上記の方針において、候補の生成フェーズでは、

(1) 二つの時間情報付き転置索引 Φ_u および Φ_v から時間幅 I に含まれる軌跡を *RangeQuery* によって検索し、

(2) 得られた候補集合を軌跡 ID (k) をキーとして join する

ことで得られる。より具体的には、ステップ 1 で得られる二つのリレーション $X = \text{RangeQuery}(u, I)$ および $Y = \text{RangeQuery}(v, I)$ に対して、ステップ 2 では

```
SELECT X.*, Y.* FROM X JOIN Y ON
```

```
X.k = Y.k AND X.t < Y.t
```

なる操作を実行することで、 u および v を時間幅 I の間に両方通過したような軌跡の集合 Z を抽出する。なお、条件 $X.t < Y.t$ は、 u と v の通過順序が正しいことを保証するための条件である。

b) 経路復元

上記で得られた u および v を時間幅 I の間に両方通過したような軌跡は u と v をどのような経路で走行したかについては不明であるため、その各々について具体的に経路を復元する。これは 3.1 で述べた時間情報付き転置索引だけでは困難であり、何らかの追加的なデータ構造が必要である。最もシンプルな解としては時間情報付き転置索引の各レコードに次のレコードへのポインタを埋め込む、というものである。これは TB-tree [12] などで用いられている戦略である。しかしながら、このポインタに沿って経路を復元しようとすると、大量のディスク I/O が発生し、先の修正 Traj-PrefixSpan と比較した際には高速化は望めない。

この復元の問題に対する我々の解は、

- 経路情報のみを配列としてメモリ内に圧縮して保持する、というものである。具体的には、時間情報付き転置索引のレコードを (k, t_i, h_i) から (k, t_i, h_i, ptr_i) に変更する。ここで ptr_i はメモリ内に圧縮して保持されている経路配列上の対応する位置へのポインタである (具体的には後述するように逆接尾辞配列をポインタとして用いる)。上で述べた join 操作の際に、このポインタも取り出すことで、圧縮されたインメモリデータ構造上で高速な復元操作を行うことができる。

以下では、(1) このインメモリデータ構造をどのように実現するか、(2) インメモリデータ構造上の復元をどのように枝刈りするか、という二点について議論する。

3.4 経路復元のためのインメモリデータ構造

記号列をインメモリで圧縮しながらさまざまな操作を可能にするデータ構造として FM-index が知られている。以下では、時間情報付き転置索引に対して FM-index を追加的に用いるためのデータ構造 [15] について概観し、その後本研究で対象とする問い合わせでの利用について述べる。

a) FM-index + 逆接尾辞配列による軌跡索引化

我々が [15] で提案したデータ構造では、まずはじめに、軌跡

を含む軌跡データベース $\mathbb{T}$ に含まれる全経路 $\pi_k = T_k.p(k = 1, \dots, K)$ を結合した 軌跡文字列 T を構築する:

$$T = \pi_1^{\text{rev}} \$ \pi_2^{\text{rev}} \$ \dots \$ \pi_K^{\text{rev}} \$ \# . \quad (4)$$

ここで、 π_k^{rev} は経路 π_k を反転させた文字列であり、 $\$$ および $\#$ は連結の境界および終端を識別するための特殊文字であり、辞書順は $\# < \$ < \forall e \in E$ と仮定する。

例えば図 1 の道路ネットワーク上の 4 つの軌跡 $\pi_1 = ABC, \pi_2 = ABEG, \pi_3 = ABEG, \pi_4 = ADFG$ に対しては、 $T = CBA\$GEBA\$GEBA\$GFDA\$ \#$ となる。

次に、軌跡文字列 T の接尾辞配列を考える (以下で概説する文字列索引技術に関しては例えば [14] に詳しい)。図 2 に上述の例で用いた T に対する接尾辞配列の構築を図示した。まず、 T の接尾辞 $S_i = T[i..|T|]$ を考える (図 2 の左)。次に接尾辞 S_i を辞書順にソートする (図 2 の中央)。接尾辞配列 SA は T と同じ長さを持つ整数値の配列であり、 $SA[j] = i$ であるとは S_i が j 番目に小さな接尾辞である、ということを意味する。例えば $S_{19} = \#$ は最も辞書順の小さな接尾辞であるため、 $SA[0] = 19$ となっている。また、逆接尾辞配列 ISA は接尾辞配列の逆関数として定義されるような配列である (接尾辞配列は全単射の写像であるので、これは well-defined である)。

T の Burrows-Wheeler 変換 T_{bwt} は T と同じ長さをもつ文字列であり、接尾辞配列 SA を用いて、 $T_{bwt}[j] = T[SA[j] - 1]$ と定義される [1]。これは可逆変換であることが知られている。Burrows-Wheeler 変換をウェーブレット木と呼ばれるデータ構造に格納することで FM-index が得られる。ウェーブレット木は文字列をメモリ上にコンパクトに保持しつつ、様々な操作を可能とするような簡潔データ構造である。FM-index を用いることで、ウェーブレット木に格納された T_{bwt} から T の復元やパターンマッチングといった操作が可能になる (これらについては後述する)。

次に、これらの情報を時刻情報付き転置索引へどのように格納するのか、について述べる。既に述べたとおり、軌跡データは三つ組 (k, p_i, t_i) にハッシュ値 h_i を加えた (k, p_i, t_i, h_i) で表現される。すべてのデータを軌跡文字列の順序でソートしたテーブルを図 3 に示している。道路リンクの列 p は軌跡文字列 T に対応している。また ISA の列は T の逆接尾辞配列である。このようにして軌跡データと逆接尾辞配列の対応付け $(k, p_i, t_i, h_i, isa_i)$ が得られ、時刻情報付き転置索引 $\Phi_{p_i} \mapsto (k, t_i, h_i, isa_i)$ を格納する、というのが [15] の主旨である。以下では、このデータ構造を用いてどのように効率的に $W(u, v, I, \xi)$ を求めるか、について述べる。

b) FM-index による経路復元

FM-index では任意の接尾辞配列上の位置 (すなわち逆接尾辞配列の値) isa から文字列の復元が可能であることが知られている [5]。これは例えば、図 3 において、 $isa = 8$ としたとき (12 行目)、その後の文字列、すなわち $BEG\dots$ (11 行目よりも上) を任意の長さ復元できるということを意味する。FM-index には軌跡文字列 T そのものは格納されておらず、 T_{bwt} のみが格納されているため、これは自明ではないが、Algorithm 3 に

i	Suffixes S_i	Sorted Suffixes $S_{SA[j]}$	j	$SA[j]$	i	$ISA[i]$
0	CBA\$GEBA\$GEBA\$GFDA\$#	#	0	19	0	12
1	BA\$GEBA\$GEBA\$GFDA\$#	\$#	1	18	1	9
2	A\$GEBA\$GEBA\$GFDA\$#	\$GEBA\$GEBA\$GFDA\$#	2	3	2	6
3	\$GEBA\$GEBA\$GFDA\$#	\$GEBA\$GFDA\$#	3	8	3	2
4	GEBA\$GEBA\$GFDA\$#	\$GFDA\$#	4	13	4	17
5	EBA\$GEBA\$GFDA\$#	A\$#	5	17	5	14
6	BA\$GEBA\$GFDA\$#	A\$GEBA\$GEBA\$GFDA\$#	6	2	6	10
7	A\$GEBA\$GFDA\$#	A\$GEBA\$GFDA\$#	7	7	7	7
8	\$GEBA\$GFDA\$#	A\$GFDA\$#	8	12	8	3
9	GEBA\$GFDA\$#	BA\$GEBA\$GEBA\$GFDA\$#	9	1	9	18
10	EBA\$GFDA\$#	BA\$GEBA\$GFDA\$#	10	6	10	15
11	BA\$GFDA\$#	BA\$GFDA\$#	11	11	11	11
12	A\$GFDA\$#	CBA\$GEBA\$GEBA\$GFDA\$#	12	0	12	8
13	\$GFDA\$#	DA\$#	13	16	13	4
14	GFDA\$#	EBA\$GEBA\$GFDA\$#	14	5	14	19
15	FDA\$#	EBA\$GFDA\$#	15	10	15	16
16	DA\$#	FDA\$#	16	15	16	13
17	A\$#	GEBA\$GEBA\$GFDA\$#	17	4	17	5
18	\$#	GEBA\$GFDA\$#	18	9	18	1
19	#	GFDA\$#	19	14	19	0

図 2 軌跡文字列 $T = \text{CBA\$GEBA\$GEBA\$GFDA\$#}$ に対する接尾辞配列および逆接尾辞配列の構築

	k	p	t	isa	h
0	1	C	33	12	8
1	1	B	28	9	3
2	1	A	23	6	0
3		\$		2	
4	2	G	84	17	10
5	2	E	76	14	5
6	2	B	71	10	3
7	2	A	59	7	0
8		\$		3	
9	3	G	68	18	10
10	3	E	59	15	5
11	3	B	53	11	3
12	3	A	46	8	0
13		\$		4	
14	4	G	52	19	9
15	4	F	47	16	7
16	4	D	41	13	3
17	4	A	32	5	0
18		\$		1	
19		#		0	

軌跡文字列 T T の逆接尾辞配列

図 3 軌跡データ (k, p_i, t_i, h_i) と逆接尾辞配列 ISA の関連付け。道路リンクの列 (p) は軌跡文字列 T に対応しており、 ISA の列はその逆接尾辞配列 ISA に対応している。 ISA の値は経路復元時におけるポイントとしての機能を有する。

よってこれが実現できる。これによって任意の接尾辞配列上の位置 isa から指定した文字 v が出現するまでの部分文字列復元することができる。なお、Algorithm 3 における $rank_w(T_{bwt}, j)$ は接頭辞 $T_{bwt}[0, j)$ における文字 w の出現回数であり、 $C[w]$ は T_{bwt} 内において w よりも辞書順の小さな文字が出現した回数である (事前に計算しておく)。

これを用いて、候補生成フェーズで得られる候補集合 Z に含まれる各レコードの u の逆接尾辞配列の値 (これを isa_u と書く) に対して $extract_until(isa_u, v)$ と実行することで $u-v$ 間の経路を復元することができる。

3.5 経路復元操作の枝刈り

前節の方法によって、インメモリの FM-index と時間情報付き転置索引を組み合わせることができた。経路の復元操作はインメモリで行われるため比較的高速であるが、それでも候補集

Algorithm 3: $extract_until(T_{bwt}, j, v)$: BWT 文字列

T_{bwt} 上の任意の位置 j (逆接尾辞配列の値) から文字 v が見つかるまで文字列をデコードする。

```

1  $S \leftarrow$  empty string
2 repeat
3    $w \leftarrow access(T_{bwt}, j)$  //  $T_{bwt}[j]$ 
4    $S \leftarrow wS$  // concatenate  $w$ 
5    $j \leftarrow C[w] + rank_w(T_{bwt}, j)$ 
6 until  $w \neq v$ 
7 return  $S$ 

```

合 Z が大きい場合、この復元操作がボトルネックになり得る。ここでは経路復元操作を枝刈りし、計算を効率化するための二つのフィルタリング手法について議論する。

- 最小サポート条件を満たすかどうかをハッシュ値に基づいて推定する。
- 経路復元の前にすでに一度復元された経路かどうかを逆接尾辞配列の値に基づいてチェックする。

3.5.1 ハッシュ値に基づく最小サポート条件のチェック

$W(u, v, I, \xi)$ の定義より、軌跡データベース $\mathbb{T}$ 中に ξ 以下しか発生しないような部分経路は $W(u, v, I, \xi)$ には含まれない。したがって、復元する前に出現回数を推定することができれば、それを元に出現回数の少ない経路の復元を回避できる。そこで候補集合 Z だけから、 Z の各要素の $supp(P, I)$ の上限を見積もり、それによって Z をフィルタリングするための方法を提案する。

具体的には Z の各レコードは元の u と v の転置索引に含まれるハッシュ値 h_u および h_v が含まれている。このハッシュ値の差 $\Delta = h_v - h_u$ は $u-v$ 間の経路が同じである限り同じ値をとる。したがって、 Z の各レコード i に対してハッシュ値の差 Δ_i を計算しておき、この値のヒストグラムを構築する。このとき頻度 $count(\Delta_i)$ が ξ 以下になるようなハッシュ値を持つレコード $z \in Z$ は $W(u, v, I, \xi)$ の対象になりえないため、

復元を行う必要がない。なお、この $count(\Delta_i)$ は対応する経路の頻度の厳密な値ではなく、上限である。これはハッシュ値が衝突する (すなわち異なる $u-v$ 間の経路に対して同じハッシュ値を持つ) 可能性があることに起因する。

3.5.2 逆接尾辞配列による重複チェック

上記のハッシュ値に基づくフィルタリングにおいて、フィルタリングされなかった Z の各レコードに対応する $u-v$ 間の経路は多くの重複がある。あるレコード $(k, t_u, h_u, isa_u, t_v, h_v, isa_v) \in Z$ を見たときに、経路を復元する前に既に見つかった経路かどうかをチェックするにはどうすればよいだろうか。ハッシュ値の差 $h_v - h_u$ が同じであっても、同じ経路であるとは言い切れないため、ハッシュ値を用いることはできない。ここでは isa_v を用いることで重複チェックが可能であることを示す。

具体的には、経路 P が FM-index から復元された段階で、FM-index に経路 P のパターンマッチング問い合わせを行う。このパターンマッチング問い合わせは以下のようなものである。例えば、図 2 において $P = GE$ というパターンを考える。各接尾辞 S_i は辞書順でソートされているため、パターン P はある範囲に連続して接頭辞として出現する。この例の場合は 17 番目および 18 番目である (この範囲を図 2 では $P(GE)$ と示している)。一般に任意のパターン P に対してこのように接頭辞が P となるような唯一の範囲 $R(P) := [sp, ep]$ が存在する。この範囲を求める問い合わせを本研究では **パターンマッチング問い合わせ** と呼ぶ。

性質 4 パターンマッチング問い合わせは FM-index を用いた Algorithm 4 によって高速に処理することができる。

性質 5 [15] $R(P) = [sp, ep]$ とする。このとき $sp \leq ISA[i] < ep$ であることと、 $T[i..i + |P|] = P$ は同値である。ここで i は図 3 における行番号を意味している。

Algorithm 4: $search(P, T_{bwt})$: 長さ m のパターン P に対応する接尾辞配列上の範囲を求める

```

1  $w \leftarrow P[m-1]$ 
2  $sp \leftarrow C[w]; ep \leftarrow C[w+1]$ 
3 for  $i = 2, \dots, m$  do
4    $w \leftarrow P[m-i]$ 
5    $sp \leftarrow C[w] + rank_w(T_{bwt}, sp)$ 
6    $ep \leftarrow C[w] + rank_w(T_{bwt}, ep)$ 
7 return  $[sp, ep]$ 
```

これらの性質を用いることで以下のように重複チェックを行うことができる。あるレコード z が $u-v$ 間を経路 P で走行したと $z.isa_v$ が $R(P)$ に含まれることは同値であるため、これを用いて重複チェックを行うことができる。extract-until によって部分軌跡 P が復元されたと同時に、性質 4 によって、範囲 $R(P)$ を求めておく。この範囲を区間木 $ITree$ に格納しておく。性質 5 によって、あるレコード z の逆接尾辞配列の値 $z.isa_v$ をこの区間木に問い合わせることで、対応する区間が登録されている場合にはすでに対応する経路がすでに見つかっている、ということがわかるため、復元を行わず、 P の出現回数

カウントのみを行う。

3.6 提案アルゴリズムのまとめ

Algorithm 5 に提案アルゴリズムをまとめた。まず 1 行目では §3.5.2 で用いる区間木を初期化している。2-4 行目は候補生成フェーズである。5 行目は候補集合 Z に §3.5.1 で述べた、各レコード z のハッシュ差 Δ_z のヒストグラムを生成している。6-16 行目は経路復元フェーズである。7-9 行目は §3.5.1 のハッシュ値によるフィルタリングである。ここで $count(\Delta) \leq \xi$ となるレコードに対応する経路は $\xi + 1$ 回以上出現し得ないので、復元を行わない。10 行目では §3.5.2 の重複チェックである。区間木に $z.isa_v$ を含むような区間が登録されていれば、11-15 行目の経路復元を行わない。経路復元を行う場合、§3.5.2 で述べたように 12 行目で経路復元を行い、復元された経路 P に対応する接尾辞配列上の範囲 R を求め (13 行目)、その範囲を新たに区間木に登録し (14 行目)、連想配列 A に経路を登録する。16 行目では接尾辞配列の範囲 R をキーとしてカウントを一つ増加させている。17 行目ではカウントが ξ よりも大きな経路を $W(u, v, I, \xi)$ として返している。

Algorithm 5: $W(u, v, I, \xi)$ を求めるための提案アルゴリズム

```

1  $ITree \leftarrow \text{Empty interval tree}$ 
2  $X \leftarrow \text{RangeQuery}(u, I)$ 
3  $Y \leftarrow \text{RangeQuery}(v, I)$ 
4  $Z \leftarrow \text{RestrictToAdj}(ST\text{-}join2(X, Y))$  // 時間幅  $I$  の間に  $u \rightarrow v$  を通過した軌跡
5  $count \leftarrow \text{HashCount}(Z)$  // ハッシュ差のヒストグラム
6 for  $z \in Z$  do
7    $\Delta \leftarrow z.h_v - z.h_u$ 
8   if  $count(\Delta) \leq \xi$  then
9     continue
10   $R \leftarrow ITree.find(z.isa_v)$  //  $R$  は  $isa_v$  を含む接尾辞配列上の範囲
11  if  $R$  is null then
12     $P \leftarrow \text{extract-until}(v, z.isa_u, T_{bwt})$  //  $u-v$  経路の復元
13     $R \leftarrow search(P, T_{bwt})$  // 復元された経路  $P$  の接尾辞配列上の範囲  $R$  を求める
14     $ITree.insert(R)$  //  $R$  を区間木に登録しておき、次回以降の復元処理の省略に用いる
15     $A[R] \leftarrow (\text{path} = P, \text{count} = 0)$ 
16     $A[R].count \leftarrow A[R].count + 1$ 
17 return  $[(A[R].\text{path}, A[R].\text{count}) \text{ for } R \text{ in } A.\text{keys} \text{ if } A[R].\text{count} \geq \xi]$ 
```

4. 評価実験

本章では実データを用いて提案アルゴリズムおよびベースラインアルゴリズム (修正 Traj-PrefixSpan) の比較を行う。時間情報付き転置索引は Postgresql (ver. 9.6) を用いて実装した。アルゴリズムは c++ を用いて実装し、FM-index の実装

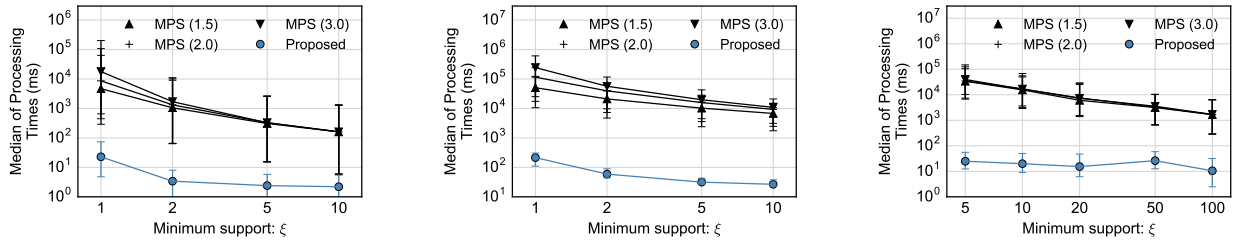


図 4 最小サポート ξ を変化させたときの処理時間。エラーバーは 25%-75% 分位点を示す。

MPS(θ) は修正 Traj-PrefixSpan の略。

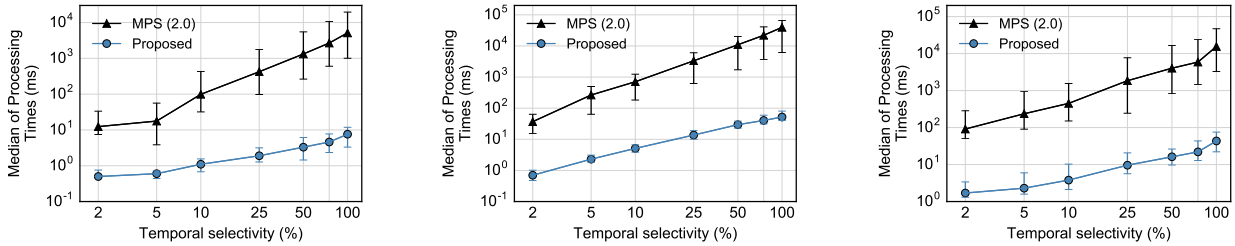


図 5 時間幅 $|I|$ を変化させたときの処理時間の変化 (2% – 100%)。エラーバーは 25%-75% 分位点を示す。

には `sdsl-lite` を用いた。評価には以下の三種類のデータを用いた。

- Singapore: シンガポールのタクシーデータセット。
- Roma: ローマのタクシーデータセット。
- RGSx5: 上記 Singapore データセットを 5 次マルコフ連鎖を用いてランダムサンプリングすることで 5 倍のサイズに増やしたもの。

これらのデータセットの統計を表 1 および FM-index のメモリ使用量に示す。今回は FM-index の実装として、平衡型ウェーブレット木 + 無圧縮ビットベクトルを用いた。圧縮型のウェーブレット木やビットベクトルを用いることでこのサイズをかなり小さくすることができる (この点は関連研究で議論する)。

表 1 データセットの詳細

Dataset	#traj	$ T $	$ E $	FM-index size (GB)
Singapore	340K	98M	55,890	0.209
Roma	130K	12M	56,651	0.033
RGSx5	1.7M	500M	55,890	1.066

図 4 に評価結果を示す。この評価では top-100 頻度の道路リンクをランダムに組み合わせて (u, v) ペアを作成し、時間幅 I は全てのデータが含まれるような大きな幅をとった。最小サポート ξ の影響を見るために ξ を変化させている。修正 Traj-PrefixSpan では、探索経路の長さを最短距離 d_{uv} の θ 倍程度に打ち切っているが、この θ については $\{1.5, 2.0, 3.0\}$ の間で変化させている。

この結果は提案手法が数百倍程度高速であることを示している。これは PrefixSpan が探索する経路が伸びるごとに RangeQuery を行うのに対して、提案手法では RangeQuery は二つの道路リンク、 u および v でのみ実行すればよく、途中の経路

は圧縮されたインメモリ索引が肩代わりするためである。

また、提案手法において、 $\xi = 1$ と $\xi = 5$ の場合を比較すると、10 倍程度 $\xi = 5$ のほうが高速である (Singapore, Roma データセット)。これは Algorithm 5 において、 $\xi = 5$ では $\text{count}(\Delta) \leq \xi$ となりやすく、枝刈りが頻繁に生じたことによる。つまり、この高速化分はハッシュ値を用いた枝刈りの効果によって生じたものである。

次に、図 5 は問い合わせ時間幅 I を変化させた際の処理時間を比較している。すべての時間幅に対して提案手法は高速であるが、問い合わせ時間幅が小さくなるとその高速化の度合い (倍率) は小さくなる。これは時間幅が小さくなると PrefixSpan において早めに枝刈りが行われるようになり、結果として RangeQuery の回数が減少するためであると考えられる。

図 6 は $u-v$ 間の最短距離 d_{uv} と処理時間の関係である。距離が大きくなると処理時間が増加するのは、PrefixSpan と提案手法で共通しているが、その増加率は提案手法のほうがかなり穏やかである。これは PrefixSpan では距離が長くなると、より広い範囲の道路リンクに対する RangeQuery を行う必要がある、ということに起因している (注6)。

最後に図 7 は Singapore データセットにおいて発見された経路の例を示している。この例では $\xi = 2$ を用いており、三つの経路 A, B, C が見つかった。これらの頻度はそれぞれ 4, 29, 3 であり、経路 B が最頻経路であり、それ以外の二つが迂回路であることが示唆される。なお、この例で $\xi = 0$ を用いると、一回だけ出現した経路が全て抽出されるが、これらは大幅な迂回をしているなど、ノイズを含んだ経路が多く見られた。実際

(注6) : その範囲は距離 d_{uv} の二乗程度で増加すると考えられ、実際の結果もそれに近い増加率を示している

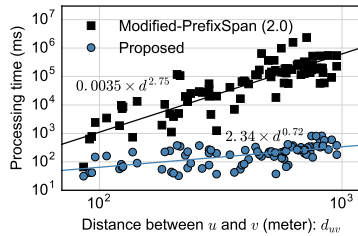


図 6 $u-v$ 間の距離 d_{uv} と所要時間 (Roma データセット, $\xi = 2$)

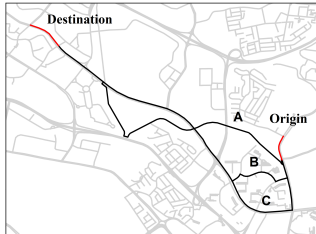


図 7 発見された経路の例

のアプリケーションにおいては適切な ξ を設定する必要があると思われるが、これについては本研究のスコープではないため割愛する。

5. 関連手法

本研究で取り扱った問題と最も関連があるのが、時間制約付き二点間の最頻経路 (Time-period-based Most Frequent Path; TPMFP) 問題である。我々の問題設定は TPMFP と同様、二点間のデータ駆動型ナビゲーションを目的としており、そのモチベーションは非常に近い。本研究では低頻度の経路も発見可能であるため、TPMFP のある種の一般化とみなすことができる。TPMFP に類似の問題設定としては MPR (Most popular route) [2] などがあるが、これは時間制約を考慮していない。

提案手法におけるデータ構造は我々の先行研究 [15] と類似している (先行研究ではハッシュを用いていない)。先行研究 [15] では本研究で扱った問い合わせとは異なる完全経路クエリという問い合わせを扱った。これらより、時間情報付きの転置索引と FM-index の組み合わせによって軌跡データに対する柔軟な時空間問い合わせが可能になる、ということが明らかになりつつあり、今後のさらなる研究が期待される。

今回の実験結果で示したように、現状で入手可能なオープンデータはデータ量がそれほど大きくないため、得られる経路の頻度はそれほど高くない。今後、データ量が増加した際には提案手法の有用性が増してくると期待できるが、その際にはインメモリ索引 (FM-index) のサイズが問題になってくるかもしれない。今回は圧縮手法を適用していないが、今後は FM-index の圧縮手法 ([10] など) が必要になってくると思われる。

ネットワーク上の軌跡データの索引構造は多くの研究がある [4], [6], [7], [8], [13]。しかしながら、これらの研究では本研究で扱ったような二点間の経路の列挙問い合わせは扱われていない。

6. まとめ

本研究では与えられたネットワーク上の二点間の経路に関する系列マイニング問題を考察した。具体的には軌跡データベース中において、 $u-v$ を通過する部分軌跡として観測され、かつその軌跡の発生時刻が与えられた時間幅に含まれるような経路のみを列挙する問題を考え、この問題に対する効率的なデータ構造およびアルゴリズムを提案した。提案手法を実データに適用し、既存の系列マイニングアルゴリズム (Traj-PrefixSpan [11]) と比較して数百倍の高速化が可能であることを見た。提案手法で用いた逆接尾辞配列を軌跡データ検索に用いる、というアイデアは我々の先行研究 [15] でも有効性が示されている (この研究では完全経路クエリと呼ばれる別の問い合わせを扱っていた)。このように、文字列アルゴリズムは軌跡データ検索における様々な問い合わせに対応可能であると考えられ、今後のさらなる研究が期待される。

文献

- [1] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. In *Technical Report 124*. Digital Equipment Corporation, 1994.
- [2] Zaiben Chen, Heng Tao Shen, and Xiaofang Zhou. Discovering popular routes from trajectories. In *Proc. ICDE'11*, pp. 900–911, April 2011.
- [3] Jian Dai, Bin Yang, and Christian S Jensen. Path Cost Distribution Estimation Using Trajectory Data. *Proc. VLDB'16*, Vol. 1, No. 3, pp. 85–96, 2016.
- [4] Victor Teixeira de Almeida and Ralf H. Güting. Indexing the trajectories of moving objects in networks. *Geoinformatica*, Vol. 9, No. 1, pp. 33–60, 2005.
- [5] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pp. 390–398, 2000.
- [6] Elias Frentzos. Indexing objects moving on fixed networks. In *Proc. SSTD'03*, pp. 289–305, 2003.
- [7] Satoshi Koide, Yukihiro Tadokoro, and Takayoshi Yoshimura. SNT-index: Spatio-temporal index for vehicular trajectories on a road network based on substring matching. In *Proc. SIGSPATIAL UrbanGIS Workshop'15*, 2015.
- [8] Benjamin Krogh, Nikos Pelekis, Yannis Theodoridis, and Kristian Torp. Path-based queries on trajectory data. In *Proc. GIS'14*, pp. 341–350, 2014.
- [9] Wuman Luo, Haoyu Tan, Lei Chen, and Lionel M. Ni. Finding time period-based most frequent path in big trajectory data. In *Proc. SIGMOD'13*, pp. 713–724, 2013.
- [10] Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE'07*, pp. 229–241, 2007.
- [11] Mikolaj Morzy. Mining frequent trajectories of moving objects for location prediction. In *Proc. MLDM'07*, pp. 667–680, 2007.
- [12] Dieter Pfoser, Jensen S. Christian, and Yannis Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. VLDB'00*, pp. 395–406, 2000.
- [13] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. Indexing in-network trajectory flows. *The VLDB Journal*, Vol. 20, No. 5, pp. 643–669, October 2011.
- [14] 岡野原大輔. 高速文字列解析の世界: データ圧縮・全文検索・テキストマイニング. 岩波書店, 2012.
- [15] 小出智士, 田所幸浩, 吉村貴克. 文字列索引によるネットワーク制約下の車両軌跡の索引化. In *DEIM*, 2016.