CF-Suffix Trieを用いた頻出移動パターンマイニング手法

Frequent Moving Pattern Mining Method Using CF-Suffix Trie

稲田泰裕[1*]　池田大輔[2]　鈴木英之進[2]
Yasuhiro Inada[1]　Daisuke Ikeda[2]　Einoshin Suzuki[2]

[1] 九州大学大学院 システム情報科学府 情報理学専攻
[1] Department of Informatics, Graduate School of ISEE, Kyushu University
[2] 九州大学大学院 システム情報科学研究院 情報理学部門
[2] Department of Informatics, Kyushu University

Abstract: In this paper, we propose CF-Suffix Trie for mining frequent moving patterns from spatiotemporal data and an online algorithm for constructing the trie. Our methods can discover patterns and their related spatial regions automatically with only a single scan of the data. We evaluate our methods experimentally using datasets with artificial object trajectories. The performance experiment shows that our method are more than 1000 times fater than naive methods and exhibits more than 95 % of precision.

1 はじめに

近年，位置情報を取得することのできるGPSを搭載した携帯電話や，移動追跡のためのRFIDタグなどの普及はめざましい．携帯電話へのGPS機能搭載が原則として義務化 [1] されたこともあり，GPS機能付き携帯電話は携帯電話所有者の44 %にまで普及している [2]．さらに，2010年度のRFIDタグ市場は2000 %を越える拡大が予測されており，人や物の位置情報を容易に取得できる環境が整いつつあるといえる [3]．そこから得られる位置情報の系列データは，時間的な意味 (タイムスタンプや時間的な前後関係) と空間的な意味 (物体の位置情報) の双方を持つため，しばしば時空間データと呼ばれる．

時空間データを解析することは，都市開発や在庫管理，気象予報，野生動物の生態系解明など，様々な問題解決のために非常に有益である．実際に，ユーザの行動モデルやスケジュールを分析し，現在の時間や位置情報を元に，ユーザにとって有益な情報を配信したり，検索を支援するシステムなどが開発されている [4, 5, 6]．このように，時空間データから有用な知識をマイニングする技術へのニーズが高まっている．

時空間データを扱った研究として，時間的な相関パターンに注目した研究 [7, 8] や，空間的な相関パターンに注目した研究 [9, 10] などが行われている．その後Verheinらによって，双方を統合した時空間相関ルールSTARsを効率的に発見する手法が提唱されている [11]．STARsとは，“時間帯 t_1 において領域 r_1 に出現した物体は，その後 t_2 において r_2 に出現する”という時空間的な相関関係である．彼らはSTARsを効率的に発見するために，いくつかの時空間パターンを定義し，枝刈り法を提案している．ある時間帯 t において領域 r から流出する物体が閾値以上であればsource，領域 r へ流入する物体が閾値以上であればsink，どちらも閾値以上であればthoroughfare，などである．これらは言い換えれば，1つの時間帯に関する時空間パターンである．各時間帯 t_1，t_2 においてこれらの時空間パターンをマイニングし，それらを組み合わせて候補パターンを生成する．最終的に，前述の通り2つの時間帯に関する時空間パターン (移動パターン) を発見する．

複数の時間帯に関する時空間パターン発見についての研究も複数行われている [12, 13, 14]．これらの研究において，物体の移動経路は実数の位置情報の系列で表現されるが，この系列の処理方法によって2種類に大別できる．一方は，空間をあらかじめいくつかの領域に分割しておき，実数の位置情報を物体が存在した領域のIDによって置換し，単次元系列の入力データとする手法である [12]．この入力データからSuffix treeを構築し，頻出な移動パターンを発見する．言い換えれば，多くの物体が通過する空間的領域が，移動パターンの構成要素となる．この手法は，一般的な頻出系列パターン発見問題として扱うことができるが，発見されるパターンはあらかじめ設定された領域分割に依存する．分割が大まかすぎると物体の移動を詳しく記述することができない．逆に分割が詳細すぎると，近く

*連絡先：九州大学大学院システム情報科学府
819-0395 福岡県福岡市西区元岡 744 番地
E-mail:yasuhiro.inada@i.kyushu-u.ac.jp

に存在しているのにも関わらず異なる領域だと判断してしまう．このように領域分割が適切でない場合，データに眠っている興味深いルールを発見できない可能性が高い．そのため，パターンを記述する要素 (空間的領域) を自動で発見することは，とても重要な問題である．

もう一方は，実数の位置情報の系列をそのまま入力データとする手法である [13, 14]．この場合，パターンを記述する要素 (空間的領域) を自動で発見する必要がある．Cao らの研究 [13] では，パターンを記述する要素を自動で発見するために，複数のデータ点を要約して直線にフィットさせるアルゴリズム [15] を用いている．位置情報の部分系列を直線にフィットさせ，この直線をパターンの構成要素として連結し，移動パターンを記述する．Mamoulis らの研究 [14] は，周期的に頻出する移動パターンに注目した研究であり，入力データに一定の周期長を要求する．まず，周期 T ごと (例えば，週や日など) に区切られたセグメント (移動履歴) を考える．次に，各セグメントの同じポジションごとに密度クラスタリングを行う．例えば，各セグメントが T=24 時間の周期を持ち，24 個のデータ点で構成される場合，同じ時刻ごとに密度クラスタリングを行うことになる．クラスタリングにより発見された 1 つの時間帯に関する時空間パターンから，長さ n ($2 \leq n \leq T$) の時空間パターンを発見する手法として，論文中には 2 つのアルゴリズムが提案されている．1 つは，Apriori ベースのボトムアップ的手法，もう 1 つは max-subpattern hit set property [16] を利用したトップダウン的手法である．しかし，どちらの手法も最初から全入力データにアクセスする必要があり，オンラインで構築することが出来ない．

本稿では，頻出時空間パターンの中でも特に物体の移動に注目し，頻出移動パターン発見問題を導入する．この問題に対し，入力された位置情報 (データ点) をオンラインでクラスタリングし，頻出移動パターンおよびパターンを記述する空間的領域を自動で発見するアルゴリズムを提案する．さらに，それを実現するための新しいデータ構造 CF-STrie (CF-Suffix Trie) を提案する．CF-STrie は，Suffix trie [17] の各ノードに CF 木 (clustering feature tree) [18] を持たせたデータ構造で，空間的領域の系列パターンを表現する．CF 木は BIRCH [18] で用いられるデータ構造で，入力されたデータ点を逐次クラスタリングし，CF ベクトルによってデータ点が属する空間的領域を表現する．CF-STrie は，挿入するデータ点に最も近い子ノードを選択して CF 木を降りることで，データ点を自動的にクラスタリングし，CF-STrie のパスによって同じような経路を辿ってきたデータ点のみをクラスタリングできる．さらに，CF-STrie はたった一度のデータスキャンで構築可能である．CF-STrie は，入力データに一定の周期長を要求しないため，様々な時空間データに適用可能である．本稿では，人工的に生成した物体の移動データに対して提案手法を適用し，頻出移動パターンマイニングにおける提案手法の有効性を示す．

手法を具体的に解説する前に，まず第 2 章で形式的に定義を行い，本稿で扱う問題設定について述べる．データ構造およびマイニングアルゴリズムについて第 3 章で具体的に解説する．第 4 章では，人工データを用いて実験を行い，提案手法の評価を行う．今後の展望を含め，結論を第 5 章にまとめる．

2 頻出移動パターン発見問題

2.1 定義

ロケーション l は，二次元の座標 (x, y) で表現される物体の位置情報である．x, y はそれぞれ実数値をとる．**移動シーケンス** s は，一定時間おきに記録された len 個のロケーションの系列であり，物体の移動履歴を表現する．s のロケーションの個数 len を s の長さと呼ぶ．s の末尾には終端記号\$を付与し，$s=l_1 l_2 ... l_{len}$\$と記述する．**データセットシーケンス** D は m 個の移動シーケンスを接続した 1 つの系列データである．D において最長の移動シーケンスの長さを max_len で表す．D 中の i 番目 ($1 \leq i \leq m$) の移動シーケンスを s^i で表し，その長さを len^i ($1 \leq len^i \leq max\_len$) で表す．$s^i$ の j 番目 ($1 \leq j \leq len^i$) のロケーションで始まる部分系列 (i.e. s^i の各接尾辞) を $s^{i,j}$ で表し，その長さを $len^{i,j}$ ($1 \leq len^{i,j} \leq len^i$) で表す．$s^{i,j}$ 中の k 番目 ($1 \leq k \leq len^{i,j}$) のロケーションを $l_k^{i,j}$ で表す．最終的に，データセットシーケンス D は，各移動シーケンスの末尾を示す終端記号\$を付与したロケーションの系列として次のように記述することができる．$D = s^{1,1} s^{2,1} ... s^{m,1} = l_1^{1,1} l_2^{1,1} ... l_{len^{1,1}}^{1,1}\$\, l_1^{2,1} l_2^{2,1} ... l_{len^{2,1}}^{2,1}\$ ...\$\, l_1^{m,1} l_2^{m,1} ... l_{len^{m,1}}^{m,1}\$

提案手法は，**CF ベクトル** [18] を用いて空間的領域を表現する．この空間的領域をクラスタと呼ぶこともある．CF ベクトル CF は，クラスタ内に属するロケーション l_c (c=1, 2, ..., num) の総数 num，それらの線形和 $LS = \sum_{c=1}^{num} l_c$，平方和 $SS = \sum_{c=1}^{num} (l_c)^2$ からなる．このとき，その空間的領域を CF($l_1, l_2, ..., l_{num}$) と記述する．つまり，CF=(num, LS, SS)=CF($l_1, l_2, ..., l_{num}$) である．CF の各要素は，$CF.num$，$CF.LS$，$CF.SS$ で表す．**移動パターン** P は CF ベクトルで表現された空間的領域の系列 $CF_1 CF_2 ... CF_{p\_len}$ ($1 \leq p\_len \leq max\_len$) で表される．$P$ を構成する空間的領域の個数 p_len を P の長さと呼ぶ．P を構成する 1 番目の空間的領域 CF_1 には，移動シーケンスの接尾辞 $s^{i,j}$ の 1 番目のロケーション $l_1^{i,j}$ のみが属している．P を構成する k 番目 ($2 \leq k \leq p\_len$) の空間的領域 CF_k において，$s^{i,j}$ の k 番

目のロケーション $l_k^{i,j}$ が属するとき，CF_{k-1} には $l_{k-1}^{i,j}$ が属していなければならない．このとき，CF_{t+1} に $l_{k+1}^{i,j}$ が属しているとは限らない．P の支持度は，系列の末尾 CF_{p_len} に属するロケーションの総数 $CF_{p\_len}.num$ である．P の支持度が最小支持度 $minsup$ 以上であるとき，P の接頭辞である移動パターンを頻出移動パターンとみなす．

2.2 問題設定

頻出移動パターン発見問題は，データセットシーケンス D が与えられたとき，D 中に存在する移動パターン P のうち，頻出な移動パターンの集合を出力する問題である．Mamoulis らの問題設定 [14] は，シーケンス $s^{i,1}$ におけるインデックス k が等しいロケーション $l_k^{i,1}$ の集合に対し，各 k ごとに独立して密度クラスタリングを行い，パターンを記述する空間的領域を発見する．そのため発見された空間的領域には，パターンに無関係なロケーションを含んでいる場合がある．本稿で導入した問題設定は，$l_k^{i,j}$ のクラスタリングを行う際に，$l_{k-1}^{i,j}$ のクラスタリング結果を利用する．同じような移動経路を辿ってきたロケーションのみでクラスタリングを行うことで，移動パターンの特徴をより正確に表現できる．

例として，図 1 のデータセットシーケンスを考える．図 1 中には 3 つの移動シーケンス，s^1=l_1 l_2 l_3 l_4\$, s^2=l_5 l_6 l_7 l_8\$, s^3=l_9 l_{10} l_{11} l_{12}\$が存在する．l_6 周辺から l_7 周辺へ移動した部分系列は 3 つあり，それらのうち 2 つは l_8 周辺へ移動した．この移動パターンを記述するための空間的領域として，クラスタ CF_1=CF(l_2, l_6, l_{10})，CF_2=CF(l_3, l_7, l_{11})，CF_3=CF(l_4, l_8) を発見し，この移動パターンを P_1=$CF_1CF_2CF_3$ で表す．

一方，l_5 (および l_9) 方面から移動してきて l_6 周辺，l_7 周辺へと移動した部分系列は 2 つあり，そのうち 1 つが l_8 周辺へ移動した．このとき，l_6，l_7 および l_8 はそれぞれ，CF_1，CF_2 および CF_3 に属している．既存の問題設定は，パターンを記述する領域として CF_8=CF(l_5, l_9)，CF_1，CF_2，CF_3 を発見し，この移動パターンを P_2'=$CF_8CF_1CF_2CF_3$ で表す．しかし，これらの空間的領域には l_1 方面から移動してきた部分系列のロケーションを含んでいる．l_5 (および l_9) 方面から移動してきた移動パターンの特徴をより正確に表現するには，l_1 方面から移動してきた部分系列のロケーションを含まない方が好ましい．本稿では前節の定義に従い，クラスタ CF_8=CF(l_5, l_9)，CF_9=CF(l_6, l_{10})，CF_{10}=CF(l_7, l_{11})，CF_{11}=CF(l_8) を発見し，この移動パターンを P_2=$CF_8CF_9CF_{10}CF_{11}$ で表す．同様にして，図 1 からさらにいくつかの移動パターンを発見できるが，ここでは割愛する．

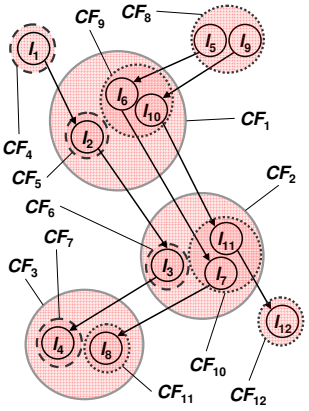

図 1: 問題設定

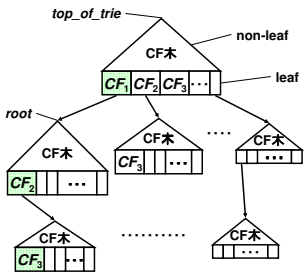

図 2: CF-STrie の概形

最小支持度 $minsup$=2 とするとき，P_1 の接頭辞である移動パターン P_1=$CF_1CF_2CF_3$，P_1'=CF_1CF_2，P_1''=CF_1 は全て頻出移動パターンである．P_2 は頻出移動パターンではないが，P_2 の接頭辞である P_2'=$CF_8CF_9CF_{10}$ は頻出移動パターンである．

3 データ構造とアルゴリズム

3.1 CF-STrie の概要

CF-STrie (CF-Suffix Trie) は，パターンを記述する空間的領域を自動的に発見するために CF ベクトルおよび CF 木 [18] を用いる．CF 木は BIRCH [18] で用いられるデータ構造で，入力されたロケーションを逐次クラスタリングし，CF ベクトルによってロケーションが属する空間的領域を表現する．CF-STrie は，Suffix trie [17] の各ノードに CF 木を持たせたデータ構造で，空間的領域の系列パターンを表現する．混同を避けるため，CF-STrie を構成する各 CF 木をトライノード，各 CF 木内のノードを CF ノードと呼ぶ．さらに，葉でない CF ノードを CF 内部ノード，葉の CF ノードを CF 葉ノードと区別する．図 2 にその概形を示す．一意に定まる要素 (e.g. 記号) によって構成されるパターン (e.g. 文字列) の場合，ノードごとに要素を保持する Suffix trie よりも，繰り返す文字列を配列上で共有する Suffix tree [17] の方がメモリ領域を節約できる．しかし 2.2 節で述べたように，クラスタに属するロケーションはそれまでの移動経路により変化するため，提案手法は Suffix trie を基本構造とし，部分移動パターンごとに独立して CF 木を保持している．

CF-STrie の各 CF 木は，共通の分岐係数 θ_{B} および最大半径 θ_{r} を持つ高さ平衡木である．CF ノードは，自身が葉であるかどうかを示す $node\_type$ および，最大 θ_{B} 個の構造体 [CF_i, $pointer_i$] (i=1, 2, ..., θ_{B}) からなる．CF 葉内部ノードの CF ベクトル CF_i は，ポイン

タ $pointer_i$ で示される子ノードの CF ベクトルを表す．つまり，CF 内部ノードは，自身より下層にあるサブクラスタで構成されるクラスタを代表する．CF 葉ノードの CF ベクトル CF_i は，最も小さいクラスタで，移動パターンの構成要素となる空間的領域を示す．その半径は θ_{r} 以下に収まらなくてはならない．CF 葉ノードの $pointer_i$ は，次のロケーションを挿入すべき CF 木の根を示す．CF-STrie のパスを辿ることで，パス上に存在する CF 葉ノードの CF ベクトルの系列を構成し，移動パターンを表現する．

アルゴリズム 1 に，CF-STrie を用いた頻出移動パターンマイニングの流れを示す．本手法は，大きく分けて 2 つのステップからなる．ステップ 1 は CF-STrie の構築である (アルゴリズム 1, 行 1〜18)．まず，データセットから 1 本の移動シーケンス $s^{i,1}$ を読み込む．読み込まれた $s^{i,1}$ の各接尾辞 $s^{i,j}$ において，$s^{i,j}$ の各ロケーション $l_k^{i,j}$ を CF 木に挿入する (アルゴリズム 1, 行 4〜15)．top_of_trie は CF-STrie の根を示し，ロケーション $l_k^{i,j}$ は k 段目のトライノード (CF 木) に挿入される．`Start_to_Insert()` は，ロケーション $l_k^{i,j}$ を $root$ で示される CF 木に挿入する関数である (アルゴリズム 1, 行 5, 11)．このとき，$l_{k-1}^{i,j}$ が挿入された CF 葉ノードの情報は，$last\_CF$ および $last\_CF\_num$ に退避されている．これらは，前回実行された `Start_to_Insert()` によって，$l_{k-1}^{i,j}$ が CF 葉ノード $last\_CF$ の CF ベクトル $CF_{last\_CF\_num}$ に追加されたことを示す．

`Start_to_Insert()` によって $l_k^{i,j}$ が挿入された CF 葉ノードの情報は，$last\_CF\_back$ および $last\_CF\_num\_back$ に格納されて返される．これらは，$l_k^{i,j}$ が CF 葉ノード $last\_CF\_back$ の CF ベクトル $CF_{last\_CF\_num\_back}$ に追加されたことを示す．同様に，次のロケーション $l_{k+1}^{i,j}$ を挿入すべき CF 木の根のアドレスは，$next\_root$ に格納されて返される．

その後，$l_k^{i,j}$ が挿入された CF 木 (i.e. $root$) へのパスを，$l_{k-1}^{i,j}$ が挿入された CF 葉ノード $last\_CF$ に登録する (アルゴリズム 1, 行 12)．パスの登録後，$last\_CF$ および $last\_CF\_num$ を，$l_k^{i,j}$ が挿入された CF 葉ノードの情報に更新する (アルゴリズム 1, 行 7〜8, 13〜14)．$root$ を $next\_root$ に更新し (アルゴリズム 1, 行 9, 15)，再び `Start_to_Insert()` を呼んで次のロケーションを挿入する．これをデータセット D 中のすべての移動シーケンスに対して繰り返す．CF-STrie の構築方法については次節にて詳しく解説し，詳細なアルゴリズムは APPENDIX に掲載する．

ステップ 2 は頻出パターンの出力である (アルゴリズム 1, 行 19)．`Output_Frequent_Pattern()` は，深さ優先探索で再帰的に CF-STrie を走査し，頻出移動パターンを出力する関数である．ある任意の移動パターンを P，P の任意の接頭辞を P' とするとき，P と P' の重複する部分系列は同じ CF ベクトルで表される．図 1 を例として用いると，P=$CF_1CF_2CF_3$，P'=CF_1CF_2 である．Suffix trie の特徴より，P が頻出であれば，P' も頻出である．P が分かれば P' についても知ることができるため，本手法では最長パターンのみを出力する．同様に，P' が頻出でないならば，P も頻出ではないので，最小支持度 $minsup$ による探索空間の枝刈りが可能である．したがって，CF-STrie の走査は $minsup$ を満たす深さまで行い，$minsup$ を下回った時点でそこまでのパスを頻出移動パターンとして出力する．

Algorithm 1 Mining_Frequent_Pattern()

Input: 移動データセット D=$s^1s^2...s^m$，最小支持度 $minsup$, 分岐係数 θ_{B}, 最大半径 θ_{r}
Output: 頻出移動パターン

```
1:  top_of_trie←NULL
2:  for all s^i such that loaded from D do
3:    for all s^{i,j} such that suffix of s^i do
4:      root←top_of_trie
5:      Start_to_Insert(l_1^{i,j}, &root, &next_root,
            &last_CF_back, &last_CF_num_back)
6:      top_of_trie←root
7:      last_CF←last_CF_back
8:      last_CF_num←last_CF_num_back
9:      root←next_root
10:     for all l_t^{i,j} such that location of s^{i,j} (1<t) do
11:       Start_to_Insert(l_t^{i,j}, &root, &next_root,
              &last_CF_back, &last_CF_num_back)
12:       last_CF->pointer[last_CF_num]←root
13:       last_CF←last_CF_back
14:       last_CF_num←last_CF_num_back
15:       root←next_root
16:     end for
17:   end for
18: end for
19: Output_Frequent_Pattern(top_of_trie, minsup)
```

4 仮想都市を想定した人工データによる実験

4.1 実験データ概要

図 3 に示す仮想都市の通勤，通学時を想定し，人々の移動パターンを生成する．まず初めに，人々の移動パターンの種となるシードシーケンスを生成する．仮想都市の面積は 1000 × 1000 とし，交通網として { 鉄道，一般道 } を考える．移動手段に { 電車，バス，徒歩 } を設ける．それぞれの移動速度は {15，10，3} である．鉄道網に存在する 35 個の駅付近にはベッドタウンを想定し，シードシーケンスの出発点を 1 つずつ配

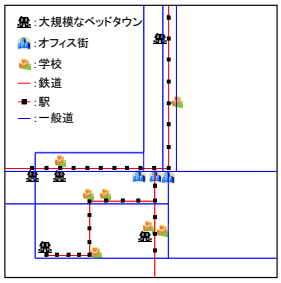

図 3: 仮想都市

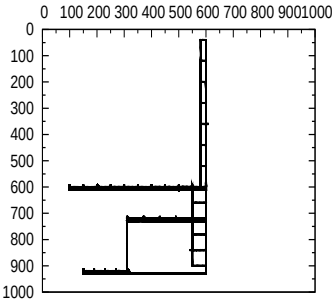

図 4: シードシーケンスの軌跡

置する．そのうち 5 箇所には，さらに 3 つずつ出発点を配置し，大規模なベッドタウンを構成する．よって計 50 個の出発点を配置する．また，オフィス街として 3 箇所，学校として 7 箇所に到着点を配置する．各ベッドタウンから各オフィス街，および最寄りの学校へ 2 本ずつシードシーケンスを生成する．一方は電車による移動，もう一方はバスによる移動である．駅 (バス停) と出発点 (到着点) 間の移動および，電車，バスに乗る必要が無い場合は徒歩とする．最終的に 400 本のシードシーケンス (ロケーション総数 15290 点) を生成する．実験データにはシードシーケンスを増殖させて使用する．また，増殖させて生成した移動シーケンスをパターンシーケンスと呼ぶ．

実験の比較対象として，単純な Suffix trie を用いた naive 手法を実装した．提案手法のトライノードが CF 木であるのに対し，naive 手法では単純に CF ベクトルのリンクリストを持たせた．リストに登録された各 CF ベクトルは，移動パターンの構成要素となる空間的領域を示す．その半径は θ_r 以下に収まらなくてはならない．ロケーション挿入時には，リストに登録された全ての CF ベクトルと比較し，クラスタリングを行う．リストの順番は，挿入されるロケーションの順番に依存する．

実験は Ubuntu8.04，Intel (R) Pentium4 (3.06GHz)，1.5GB のメモリを搭載した PC で行った．計算時間は time コマンドにてユーザ時間を測定し，メモリ使用量は生成されたノードの数から推定した．

4.2 パターンを構成する移動シーケンスのみで構成された実験データ

本実験は，パターンシーケンスのみでデータセットを構成し，ノイズが混入していないデータセットにおける入力シーケンス数の増加に対する性能実験を行った．シードシーケンスの各ロケーション l_i に対し，平均 l_i，標準偏差 σ=0.3 の正規乱数によって，シーケンス数を ρ 倍に増殖させて実験データとした．増殖係数

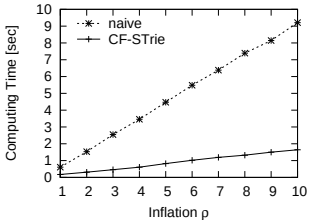

図 5: 計算時間 (ϵ=0)

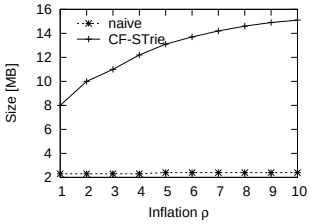

図 6: メモリ使用量 (ϵ=0)

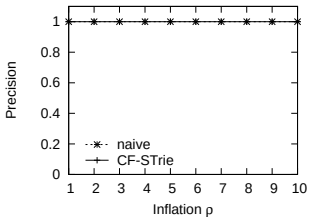

図 7: Precision (ϵ=0)

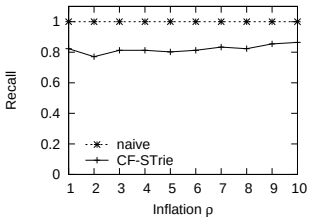

図 8: Recall (ϵ=0)

は ρ=[1, 10] で 1 刻みで変化させた．実験データを構成するシーケンスの総数は最大で 4,000 本，ロケーションの総数は最大で約 15 万点であった．CF，および CF 木のパラメータは，分岐係数 θ_B=3，最大半径 θ_r=2 で一定とした．*minsup* はパターンシーケンス数の 10% (つまり，40ρ) に設定し，頻出移動パターンのマイニングを行った．

図 5 に計算時間，図 6 にメモリ使用量の比較結果を示す．naive 手法のメモリ使用量は約 2MB でほぼ一定であったが，提案手法のメモリ使用量はシーケンス数の増加と共に増加した．これは CF 木の性質上，ロケーションを異なる CF へ挿入してしまうことがあり，本来は不要な CF ノードが生成されてしまうためと考えられる．しかし，最大でも 15MB 程度のメモリ使用量であり，今回の実験環境においては十分動作可能であることが分かった．一方，提案手法の計算時間は naive 手法よりも平均して 5.3 倍早いという結果が得られた．

ρ=1，ϵ=0 において naive 手法によってマイニングされた頻出移動パターンを基準とし，図 7 に適合率，図 8 に再現率を示す．提案手法の再現率は naive 手法よりも低い結果となったが，存在する頻出移動パターンの約 8 割をマイニングすることに成功した．再現率の低下は上述の CF 木の性質によるものと考えられる．一方，頻出パターンでないパターンを誤検出することは全く無く，100 %という非常に高い精度を示した．

4.3 パターンとは無関係な移動シーケンスを加えた実験データ

本実験は，パターンシーケンスとノイズシーケンスでデータセットを構成し，ノイズが混入したデータセッ

トにおける入力シーケンス数の増加に対する性能評価実験を行った．前節で使用した実験データに，ϵ 倍のノイズシーケンスを加え実験データとした．ノイズシーケンスは最大 30 個のロケーションで構成され，速度 10 の直線移動である (出発点と移動方向はノイズシーケンスごとにランダムで決定する)．1000 × 1000 のエリア外に達したロケーションはカットする．最終的に，実験データを構成するシーケンスの総数は，$400\rho(1+\epsilon)$ 本となる．増殖係数は前節同様 ρ=[1, 10] の範囲で変化させ，ノイズ係数は ϵ=[0, 10] で変化させた ([0, 1] の範囲では 0.1 刻み，[1, 10] の範囲では 1 刻み)．実験データを構成するシーケンスの総数は最大で 44,000 本，ロケーションの総数は最大で約 120 万点であった．CF，および CF 木のパラメータは，分岐係数 θ_{B}=3，最大半径 θ_{r}=2 で一定とした．*minsup* はパターンシーケンス数の 10% (つまり，40ρ) に設定し，頻出移動パターンのマイニングを行った．

図 9 は，naive 手法に対する提案手法の速度比を示している．シーケンス数が増加した場合でも，提案手法は naive 手法に比べ非常に高速であり，最大 1800 倍程度の速度で動作した．

図 10〜13 に，代表として ρ=5 におけるノイズの増加に対する naive 手法と提案手法の性能比較を示す．図 10 は計算時間，図 11 はメモリ使用量の比較結果である．提案手法は naive 手法と比較し，2 倍強のメモリを使用したが，メモリ使用量は最大でも 850MB 程度であり，今回の実験環境においては十分に動作可能であることが分かった．一方，naive 手法はノイズの増加とともに計算時間が指数関数的に増加するのに対し，提案手法はほぼ線形時間で動作した．さらに，naive 手法が最大 10,000sec 程度の計算時間を要したのに対し，提案手法は約 8 秒で動作可能であり，計算時間を大幅に短縮することができた．

ρ=5，ϵ=0 において naive 手法によってマイニングされた頻出移動パターンを基準とし，図 12 に適合率，図 13 に再現率を示す．提案手法の再現率は naive 手法よりも低い結果となったが，入力するシーケンスの 9 割がノイズという場合においても，存在する頻出移動パターンの約 6 割をマイニングすることに成功した．頻出パターンでないパターンを誤検出することはほとんど無く，95 %以上という非常に高い精度を維持した．

4.4 データセットのノイズ含有率が提案手法の性能に及ぼす影響

データセットのノイズ含有率が提案手法の性能に及ぼす影響に関して性能実験を行った．初期データセットを ρ=1，ϵ=0 とし，パターンシーケンスのみで増殖させた場合 (ρ=[1, 10]，ϵ=0) と，ノイズシーケンスによって増殖させた場合 (ρ=1，ϵ=[0, 9]) で提案手法の性能を比較する．一方は入力シーケンス数の増加に対し，パターンシーケンスの割合が 100 %で一定であるが，もう一方はパターンシーケンスの割合が低くなる (ノイズ含有率が高くなる)．図 14〜17 に比較結果を示す．図 14 は計算時間，図 15 はメモリ使用量の比較結果である．入力シーケンス数が同じであれば，計算時間にはほとんど差が無く，最大でも 0.4[sec] 程度であった．この差は，ノイズシーケンスのロケーション数の方が少なかったため生じたと考えられる．計算時間に関して提案手法は，データセットのノイズ含有率の影響を受けることなく動作可能であるといえる．一方，ノイズ含有率が増加するとともに，メモリ使用量はほぼ線形に増加した．これは，ノイズシーケンスにより大量の CF ノードが生成されるためであるが，メモリ使用量は最大でも 160MB 程度であり，今回の実験環境においては十分に動作可能であった．提案手法は，データセットに見られる傾向が強いほどメモリを節約することができるといえる．

ρ=1，ϵ=0 において naive 手法によってマイニングさ

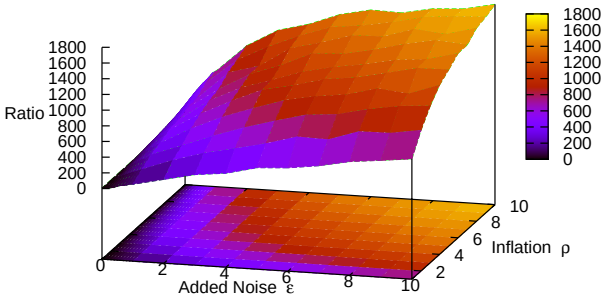

図 9: 計算時間 (ratio to naive)

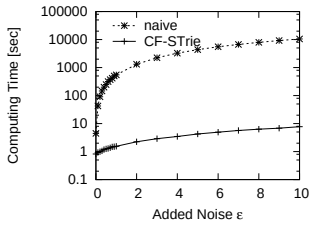

図 10: 計算時間 (ρ=5)

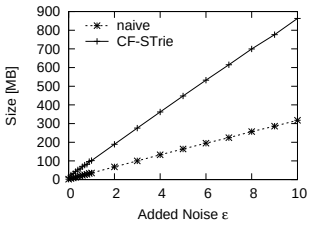

図 11: メモリ使用量 (ρ=5)

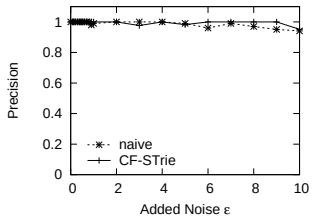

図 12: Precision (ρ=5)

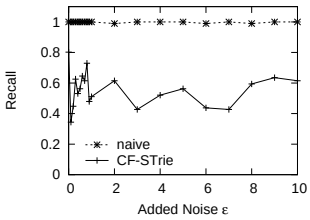

図 13: Recall (ρ=5)

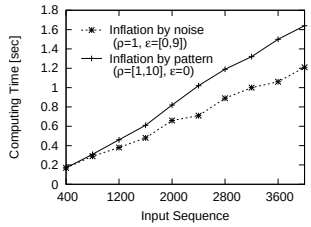

図 14: 計算時間

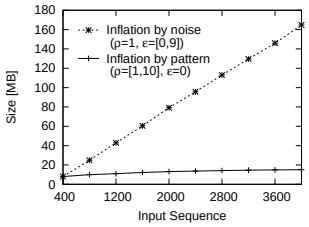

図 15: メモリ使用量

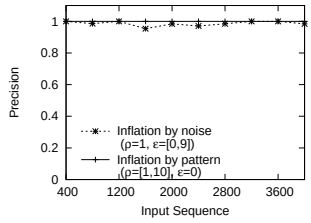

図 16: Precision

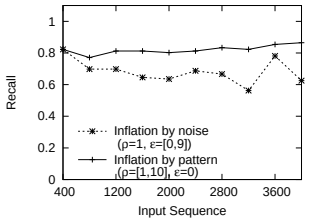

図 17: Recall

れた頻出移動パターンを基準とし，図 16 に適合率，図 17 に再現率を示す．ノイズ含有率が高くなると再現率は低下したが，入力するシーケンスの 9 割がノイズという場合においても，存在する頻出移動パターンの約 6 割をマイニングすることに成功した．頻出パターンでないパターンを誤検出することはほとんど無く，95 %以上という非常に高い精度を維持した．

5 まとめ

頻出時空間パターンマイニングにおいて，頻出時空間パターンを記述する空間的領域を自動的に発見することは重要な問題であった．従来の手法において，その問題を扱った論文はいくつか存在するが，最初から全入力データに対するアクセスが必要であったり，クラスタリングの際に複数回にわたり入力データを走査したりする必要があった．本稿では，頻出時空間パターンの中でも特に物体の移動に注目し，頻出移動パターンマイニング問題を導入した．この問題に対して我々は，位置情報のクラスタリングおよび頻出移動パターンのマイニングを，オンラインで行うためのデータ構造およびその構築アルゴリズムを提案した．さらに，人工的に生成した物体の移動データに対して提案手法を適用し，頻出移動パターンマイニングにおける提案手法の有効性を示した．

提案手法は，メモリ使用量および頻出移動パターンの再現率に関して naive 手法に劣るものの，本稿の実験環境においては十分に動作可能であり，ノイズ含有率が 90 %というデータセットに対しても存在する頻出移動パターンの約 6 割をマイニングすることに成功した．さらに提案手法は，95 %以上の高い精度で頻出移動パターンをマイニングでき，naive 手法と比べ最大約 1800 倍という高速な動作が可能となった．

今後の研究の方向性として，誤った葉へのデータ挿入による再現率低下の改善が挙げられる．具体策として，BIRCH には Phase4 としてクラスタの精錬フェーズがあり，このフェーズを CF-STrie に応用することが考えられる．

謝辞

本研究の一部は，科学技術振興機構　戦略的国際科学技術協力推進事業の援助を受けている．

参考文献

[1] 総務省，「携帯電話からの緊急通報における発信者位置情報通知機能に係る技術的条件」についての報告書案. http://www.soumu.go.jp/s-news/2004/pdf/040517_3_b1.pdf (2004).

[2] Web マーケティングガイド，【自主リサーチ調査結果】モバイル GPS 機能に関する調査 (上). http://www.e-research.biz/profile/pro_3/002351.html (2007).

[3] 矢野経済研究所, 2007 年版　非接触 IC カード・RF-ID (無線 IC タグ) 市場に関する調査結果. http://www.yano.co.jp/press/pdf/236.pdf (2007).

[4] NTT ドコモ, 行動支援型レコメンドシステムの開発. http://www.nttdocomo.co.jp/info/news_release/page/070928_00.html (2007).

[5] NTT レゾナント，goo スティック for Firefox. http://stick.goo.ne.jp/ff/ (2008).

[6] NTT レゾナント，まち goo キセキ. http://lifelog.machi.goo.ne.jp/ (2008).

[7] J. M. Ale and G. Rossi, An Approach to Discovering Temporal Association Rules. Proc. 2000 ACM Symposium on Applied Computing, pp.294-300 (2000).

[8] Y. Li, P. Ning, X. S. Wang and S. Jajodia, Discovering Calendar-Based Temporal Association Rules. Proc. Data Knowledge Eng., 44, pp.193-218 (2003).

[9] S. Shekhar and Y. Huang, Discovering Spatial Co-location Patterns, A Summary of Results.

Proc. the 7th International Symposium on Spatial and Temporal Databases (SSTD 01), pp.236-256 (2001).

[10] Y. Huang, H. Xiong, S. Shekhar and J. Pei, Mining Confident Colocation Rules without A Support Threshold. Proc. SAC 2003, pp.497-501 (2003).

[11] F. Verhein and S. Chawla, Mining Spatio-Temporal Patterns in Object Mobility Databases. Data Mining and Knowledge Discovery, Online First (2007).

[12] R. Hamid, S. Maddi, A. Bobick and M. Essa, Structure from Statistics - Unsupervised Activity Analysis using Suffix Trees. Proc. ICCV2007, pp.1-8 (2007).

[13] H. Cao, N. Mamoulis and D. W. Cheung, Mining Frequent Spatio-Temporal Sequential Patterns. Proc. ICDM 2005, pp.82-89 (2005).

[14] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao and D. W. Cheung, Mining, Indexing, and Querying Historical Spatiotemporal Data. Proc. KDD 2004, pp.236-245 (2004).

[15] D. H. Douglas and T. K. Peucker, Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. In The Canadian Cartographer, Vol.10, No.2, pp.112-122 (1973).

[16] J. Han, G. Dong and Y. Yin, Efficient Mining of Partial Periodic Patterns in Time Series Database. Proc. ICDE 1999, pp.106-115 (1999).

[17] E. Ukkonen, On-line Construction of Suffix Trees. Algorithmica, Vol.14, No.3, pp.249-260 (1995).

[18] T. Zhang, R. Ramakrishnan and M. Livny, BIRCH: An Efficient Data Clustering Method for Very Large Databases. Proc. SIGMOD Conference 1996, pp.103-114 (1996).

APPENDIX

CF-STrie の構築

CF-STrie はロケーションが挿入されることで動的に構築される．挿入するロケーションに最も近い子ノードを選択して CF 木を降りることで，各ロケーションは自動的にクラスタリングされる．同時に，辿ってきたトライノードのパスによって，移動経路に関してもクラスタリングされる．

CF-STrie を構築するアルゴリズムについて解説する．`Start_to_Insert()` では，各 CF 木構築の初期段階における例外的な処理を行う．最終的に `Start_to_Insert()` は，ロケーションを挿入した CF 葉ノードに関する情報を，*last_CF_back* および *last_CF_num_back* に格納して返す．同様に，次のロケーションを挿入すべき CF 木の根のアドレスを，*next_root* に格納して返す (アルゴリズム 2, 行 4〜6, 14〜16, 19〜21)．

root が NULL の場合，CF 葉ノードを *root* に生成し，CF_0 に l を追加する (アルゴリズム 2，行 1〜6)．`new()` は，引数が LEAF ならば CF 葉ノード，INTERIOR ならば CF 内部ノードを生成する関数である．`Add_Location_to_CF()` は，指定した CF 葉ノードの CF ベクトルにロケーションを追加する関数である．

root が葉の場合は，l が CF_0 の中心から半径 θ_r 内にあればそのまま CF_0 に l を追加する (アルゴリズム 2，行 17〜21)．l を CF_0 に追加できない場合は，*root* の内容を *CF_buf* へ退避し，新たに CF 内部ノードを *root* に生成する．さらに，*CF_back* に CF 葉ノードを生成し，その CF_0 に l を追加する．その後，*CF_buf* と *CF_back* を *root* の子ノードとして登録する (アルゴリズム 2，行 8〜13)．`Add_Children_to_Parent()` は子ノードを親ノードに登録する関数である．

root が CF 内部ノードの場合は， `Recursively_Insert()` を呼ぶ (アルゴリズム 2，行 24)．`Nearest_CF_Num()` は，指定した CF ノードの CF ベクトルのうち，最も l に近い CF ベクトルのインデックスを返す関数である (アルゴリズム 3，行 2)．`Recursively_Insert()` はロケーション l に最も近い子ノードを選択して，再帰的に CF 木を葉まで降りて行く (アルゴリズム 3，行 21〜36)．葉に達するとロケーションの追加を行う (アルゴリズム 3，行 3〜20)．ロケーション l が，最も近いクラスタの中心から半径 θ_r 内にあれば，l を追加し CF ベクトルを更新する (アルゴリズム 3，行 15〜20)．最も近いクラスタに追加できなければ，空いている CF ベクトルに l を追加する (アルゴリズム 3，行 9〜13)．もし空きがなければ，CF 葉ノードを分割する必要がある (アルゴリズム 3，行 5〜9)．`Split_Leaf()` は，CF 葉ノードの分割を行う関数で，最も離れた 2 つのクラスタを選択し，残りのクラスタを近い方に再分配する．再分配により，ロケーションを追加した CF ノードのアドレスが変更された場合には，*last_CF_back* および *last_CF_num_back*，*next_root* を更新する必要がある．

l の追加が終わったら，再帰的に戻りながら CF 内部ノードのパスおよび CF ベクトルを更新する (アル

ゴリズム 3，行 24〜35）．`Cal_CF()` は，指定した CF ノードの CF ベクトルを再計算し更新する関数である．子ノードで CF ノードの分割が起きた場合は，新たに生成された CF ノードが CF_back に格納されて返される．CF_back が NULL でない場合は，親ノードに CF_back へのパスおよび CF ベクトルを登録する（アルゴリズム 3，行 31〜34）．親ノードに空きがなければ，親ノードを分割する（アルゴリズム 3，行 26〜31）．`Split_Interior()` は，CF 内部ノードの分割を行う関数で，最も離れた 2 つのクラスタを選択し，残りのクラスタを近い方に再分配する．根が分割される場合には，CF 木の高さが 1 つ高くなる．

Algorithm 2 Start_to_Insert()

Input: ロケーション l, $**root$, $**next\_root$, $**last\_CF\_back$, $*last\_CF\_num\_back$
Output: none

```
1: if *root==NULL then
2:   new(root, LEAF)
3:   Add_Location_to_CF(l, root, 0)
4:   *next_root←*root->pointer[0]
5:   *last_CF_back←*root
6:   *last_CF_num_back←0
7: else if *root->node_type==LEAF then
8:   if l がクラスタ*root->CF[0] の中心から θr 外にある then
9:     CF_buf←*root
10:    new(root, INTERIOR)
11:    new(&CF_back, LEAF)
12:    Add_Location_to_CF(l, &CF_back, 0)
13:    Add_Children_to_Parent(&CF_buf, &CF_back,root)
14:    *next_root←CF_back->pointer[0]
15:    *last_CF_back←CF_back
16:    *last_CF_num_back←0
17:  else
18:    Add_Location_to_CF(l, root, 0)
19:    *next_root←*root->pointer[0]
20:    *last_CF_back←*root
21:    *last_CF_num_back←0
22:  end if
23: else
24:   Recursively_Insert(l, root, &CF_back, next_root, last_CF_back, last_CF_num_back)
25:   if CF_back != NULL then
26:     CF_buf←*root
27:     new(root, INTERIOR)
28:     Add_Children_to_Parent(&CF_buf, &CF_back,root)
29:   end if
30: end if
```

Algorithm 3 Recursively_Insert()

Input: l, $**node$, $**CF\_new$, $**next\_root$, $**last\_CF\_back$, $*last\_CF\_num\_back$
Output: none

```
1: *CF_new←NULL
2: nearest_CF_num←Nearest_CF_Num(l, node)
3: if *node->node_type==LEAF then
4:   if l がクラスタ*node->CF[nearest_CF_num] の中心から θr 外にある then
5:     if node には既に θB 個のクラスタが存在する then
6:       new(CF_new, LEAF)
7:       Add_Location_to_CF(l, CF_new, 0)
8:       Split_Leaf(node, CF_new, next_root, last_CF_back, last_CF_num_back)
9:     else
10:      Add_Location_to_CF(l, node, empty_CF_num)
11:      *next_root ←*node->pointer[empty_CF_num]
12:      *last_CF_back←*node
13:      *last_CF_num_back←empty_CF_num
14:    end if
15:  else
16:    Add_Location_to_CF(l, node, nearest_CF_num)
17:    *next_root ←*node->pointer[nearest_CF_num]
18:    *last_CF_back←*node
19:    *last_CF_num_back←nearest_CF_num
20:  end if
21: else
22:   CF_buf←*node->pointer[nearest_CF_num]
23:   Recursively_Insert(l, &CF_buf, &CF_back, next_root, last_CF_back, last_CF_num_back)
24:   *node->CF[nearest_CF_num] ←Cal_CF(&CF_buf)
25:   if CF_back != NULL then
26:     if node には既に θB 個のクラスタが存在する then
27:       new(CF_new, INTERIOR)
28:       *CF_new->CF[0]←Cal_CF(&CF_back)
29:       *CF_new->pointer[0]←CF_back
30:       Split_Interior(node, CF_new)
31:     else
32:       *node->CF[empty_CF_num] ←Cal_CF(&CF_back)
33:       *node->pointer[empty_CF_num] ←CF_back
34:     end if
35:   end if
36: end if
```