修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　　情報・ネットワーク工学専攻　博士前期課程		
氏　　名	山﨑智博	学籍番号	1631154
論 文 題 目	集合間類似度を用いたストリームデータの top-k 類似検索に対する高速な厳密解アルゴリズム		

要　旨

IoT の隆盛に伴い，ストリームデータ解析の重要性が高まっている．その中でストリームデータを対象とする類似検索は，正常/異常検出や情報推薦の基盤となる技術として注目されている．

ストリームデータに対する類似検索には様々な問題設定が知られているが，本研究では，2016 年に Xu らによって提案された「Continuous Similarity Search for Evolving Queries」(CSPEQ) 問題を取り扱う．この問題は，アルファベットを要素とするストリームデータを取り扱い，ストリームの直近の要素群をクエリ集合として，集合のデータベースからクエリと最も類似した上位 k 個のデータを探すことを目的とし，時間によるユーザの嗜好の変化に適応的に情報推薦を行う状況をモデル化したものである．クエリがユーザの最近の嗜好，検索結果がユーザの嗜好に合わせて推薦されるオブジェクトに相当する．

この問題に対する自明な解法は，毎時刻クエリと全データ間で類似度を計算することである．一方 Xu らは, 過去に計算した類似度の値から現在時刻で上位 k 位に入る可能性があるデータを決定し，それらに対してのみ類似度を計算する枝刈りによる厳密解法 GP(General Pruning algorithm)と Min-Hash を用いた近似解法 MHI(MinHash-based algorithm using inverted indices)を提案した．

これに対して本研究では，前の時刻から類似度が変化する可能性があるデータに対してのみ，現在の類似度を更新することで，類似度の計算回数を大幅に減らし，さらに 1 回の類似度計算も $O(1)$で行うことで高速化する新しい手法を EA-FIL(Exact Algorithm using Frequency-based Inverted Lists)を提案した．そして，実験的にその性能を評価した結果，MHI と同じくらい高速で，GP を 10 倍以上凌駕する性能であることが確認できた．

また，CSPEQ 問題はスライディングウインドウサイズを固定していることから，要素数がスライディングウインドウサイズと大きく違うデータベース内の集合は類似度が低くなって検索結果となりにくいという問題を持っている．本論文ではこれを緩和するため，ウインドウサイズに幅を持たせた新しい問題設定を提唱し，それに対する高速なアルゴリズム VWEA-FIL(Variable Window size EA-FIL)も構築した．これについても評価実験を行った結果，EA-FIL を全てのウインドウサイズに適用する単純手法よりも高速で空間計算量も小さいことが確認できた．

平成29年度　　修士論文

集合間類似度を用いたストリームデータのtop-k類似検索に対する高速な厳密解アルゴリズム

電気通信大学大学院 情報理工学研究科
情報・ネットワーク工学専攻

1631154　山﨑 智博

指導教員
古賀 久志 准教授
南 泰浩 教授

平成30年3月7日

目 次

第1章

序論

1.1 研究背景と研究目的

IoT の隆盛に伴い，ストリームデータ解析の重要性が高まっている．その中でストリームデータを対象とする類似検索は，正常/異常検出や情報推薦の基盤となる技術として注目されている．例えば，ユーザがクリックしたウェブページの履歴はストリームデータであるが，ユーザの嗜好を表しており，ストリームデータに対する類似検索によりユーザの嗜好に適合したウェブページやウェブ広告の推薦が実現できる．

ストリームデータに対する類似検索には様々な問題設定が知られており，2016 年に Xu ら [1] によって新しい問題「Continuous Similarity Search for Evolving Queries」[1] (CSPEQ 問題) が提唱された．この問題は，アルファベットを要素とするストリームデータを取り扱い，ストリームの直近の要素群をクエリ集合として，集合のデータベースからクエリと最も類似した上位 k 個のデータを探すことを目的とする．時刻経過によってストリームに新しい要素が出現するとクエリは変化するが，クエリが変わる度に最も類似した上位 k 個のデータを更新する必要がある．この問題は，時間によるユーザの嗜好の変化に適応的に情報推薦を行う状況をモデル化したもので，クエリがユーザの最近の嗜好，検索結果がユーザの嗜好に合わせて推薦されるオブジェクトに相当する．この問題に対する自明な解法は，時間が経過する度に，クエリと全データ間で類似度を計算することである．しかし，自明な解法では類似度計算の計算回数が多くなるためオーバーヘッドが大きい．そこで，Xu ら [1] は，過去に計算した類似度の値から現在時刻で上位 k 位に入る可能性があるデータを決定し，それらに対してのみ類似度を計算する枝

刈りによる高速化手法と Min-Hash を用いた近似解法を提案した．

これに対して本研究では，クエリに関する以下の性質に着目して，従来手法を凌駕する厳密解法の高速化を実現する．

1. 徐々に変化するクエリに対して，現在時刻と次の時刻で類似度が変わらないデータが多い．

2. 次の時刻の類似度は，現在の類似度を更新することで高速に計算できる．

具体的には，前の時刻から類似度が変化する可能性があるデータに対してのみ，現在の類似度を更新することで，類似度の計算回数を大幅に減らし，さらに 1 回の類似度計算のオーバーヘッドも削減して処理を高速化する新しい手法の提案を目的とした．

また，CSPEQ 問題はスライディングウインドウサイズを固定していることから，要素数がスライディングウインドウサイズと大きく違うデータベース内のデータは類似度が低くなって検索結果となりにくいという問題を持っている．本論文ではこれを緩和するため，ウインドウサイズに幅を持たせた新しい問題設定を提唱し，それに対する高速なアルゴリズムも構築した．

1.2　本論文の構成

以下に本論文の構成を述べる．

第 2 章 CSPEQ 問題について述べる．

第 3 章 従来の高速化手法について述べる．

第 4 章 提案手法について述べる．

第 5 章 拡張した問題設定の定義とその厳密解アルゴリズムを提案する．

第 6 章 提案手法の評価実験について述べる．

第 7 章 関連研究について述べる．

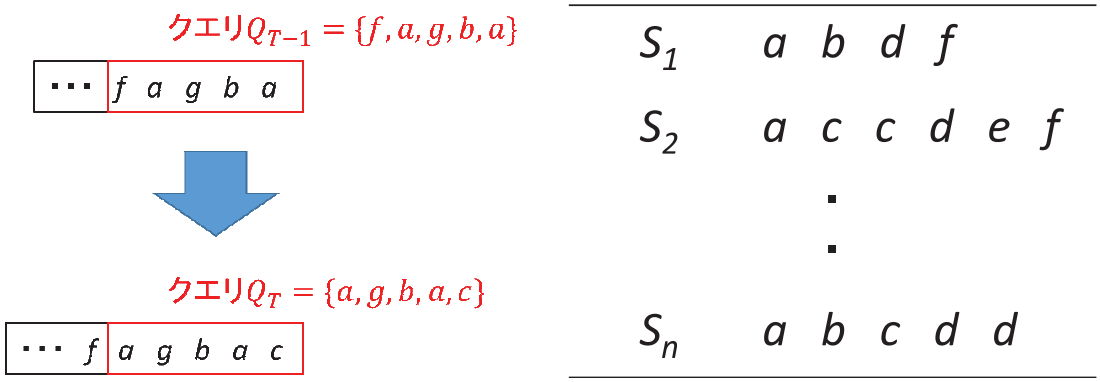
第 8 章 本研究のまとめを述べる．

第2章

CSPEQ問題

本章では,「Continuous Similarity Search for Evolving Queries」[1] 問題 (CSPEQ 問題) の定義を述べる．CSPEQ 問題は 2016 年に Xu ら [1] によって提唱された問題である．この問題は 1 つのストリームデータと n 個の集合データベース D から構成される．ストリームデータの直近の要素群をクエリ集合として，D の中から top-k の類似集合を見つける問題である．

2.1 問題定義



(1) ストリームデータ　　(2) データベース

図 2–1: 問題設定

図 2–1(1) にストリームデータを図示する．$\Phi = \{x_1, x_2, \cdots, x_{|\Phi|}\}$ をアルファベット集合とする．データストリームには毎時刻，新しい要素 $e \in \Phi$ が追加される．時刻 T において，データストリームの直近の W 個の要素群がクエリ Q_T になる．つ

まり，e_T を時刻 T に追加された要素とすると，クエリ Q_T は以下のように表現される．

$$Q_T = \{e_{T-W+1}, e_{T-W+2}, \cdots, e_T\}.$$

図 2–1(1) には，時刻 T において新しい要素 c が追加されクエリが変化している様子が示されている．

図 2–1(2) にデータベースを図示する．一方，データベース D にはアルファベットを要素とする n 個の集合 $\{S_1, S_2, \cdots, S_n\}$ が登録されている．データベースは静的であり時刻によらず固定されている．

本問題は毎時刻において，クエリと最も類似した上位 k 個の集合 (top-k) を D から検索する問題である．

クエリ Q とデータベースの集合 S 間の類似度は Jaccard 係数

$$\mathrm{sim}(S, Q) = \frac{|S \cap Q|}{|S \cup Q|}$$

を利用する．ただし，S，Q が多重集合である場合，Jaccard 係数は

$$|S \cap Q| = \sum_{i=1}^{|\Phi|} \min(s_i, q_i)$$

$$|S \cup Q| = \sum_{i=1}^{|\Phi|} \max(s_i, q_i)$$

と定義される (拡張 Jaccard 係数とも呼ばれる)．ここで，s_i, q_i は集合 S，Q が i 番目のアルファベット x_i を含む数を表している．

2.2　自明な解法 (BFM)

この問題に対する，自明な解法は，毎時刻 T でデータベース D 内の全ての集合 S_i とクエリ Q_T との類似度を計算し，top-k を求める方法である．時刻 T におけるこのアルゴリズムを Algorithm 1 に示す．

この手法では，各時刻で類似度の計算を n 回行うことが処理のボトルネックとなる．1 回の Jaccard 係数の計算の計算量は，集合の要素の和に比例し，

$$O(|Q| + |S|) = O(W + |S|). \tag{2.1}$$

となる．従って各時刻での計算量は $O(\sum_{i=1}^{n}(W + |S_i|)) = O(nW + \sum_{i=1}^{n}|S_i|)$ である．

Algorithm 1 単純な top-k 類似検索アルゴリズム (BFM)

1: **for** $i \leftarrow 1$ to n **do**
2: 　$\mathrm{sim}(S_i, Q_T)$ を計算.
3: **end for**
4: 類似度が大きい順に k 個取り出す.

2.3　この問題の応用領域

この問題は，時間によるユーザの嗜好の変化に適応的に情報推薦を行う状況をモデル化したものである．ストリームデータに毎時刻追加される要素が，ユーザの嗜好を表している．そして，クエリがユーザの最近の嗜好，検索結果がユーザの嗜好に合わせて推薦されるオブジェクトに相当する．これにより，クエリが時間経過に伴い変化し，ユーザの最近の嗜好が変化していく状況を模擬している．

また，コンピュータゲームにも応用できる．ゲームではプレイヤーが状況に応じて，武器や道具を使う．このとき，敵の数などの環境は時間によって変化する．これを CSPEQ 問題に当てはめると，ストリームデータに毎時刻追加される要素がプレイヤーが接近した物体を表し，クエリがプレイヤーの現在の状況を表す．これに似た属性を持つ武器や防具を検索することで，現在の状況に適切な武器や道具を推薦できる．

第3章

従来の高速化手法

本章では，CSPEQ 問題の Xu らによる従来手法 [1] を説明する．Xu らは枝刈りによる厳密解法と Min-Hash を用いた近似解法を提案している．

3.1 枝刈りによる厳密解法 (GP)

2.2 節で述べた単純手法では，過去に計算した類似度の値を再利用して計算量を削減することを一切しない．一方，Xu らは過去に計算した類似度の値を用いて計算量を削減する枝刈りアルゴリズム GP(General Pruning algorithm)[1] を提案した．

この手法の基本アイデアは，次のようになる．

- 過去のクエリに対する類似度の値から現在のクエリに対する類似度の上限値を高速に求める．
- 上限値が小さく，現在の時刻において上位 k 位に入る可能性がない集合に対して類似度計算を省略する．

つまり，GP は各時刻の類似度の計算回数を n から減らすことで処理を高速化する．

まず，上限値の計算方法について説明する．時刻 t における，Q_t と集合 $S \in D$ 間の類似度を $\mathrm{sim}(S, Q_t)$ とする．時刻が u 進み $t+u$ になった時，新しい要素が u 個追加され，古い要素がクエリから u 個除かれる．新しい要素が全て共通要素となり，古い要素が全て元々共通要素でなかった場合，類似度は式 (3.1) により表すことができ，これが上限値となる．

$$\frac{u + |S \cap Q_t|}{-u + |S \cup Q_t|}. \tag{3.1}$$

ここで，積集合の要素数は $\mathrm{sim}(S, Q_t) = \frac{|S \cap Q_t|}{|S \cup Q_t|}$ と $|S \cup Q_t| = |S| + |Q_t| - |S \cap Q_t|$ より，

$$|S \cap Q_t| = \frac{(|S| + |Q_t|) \cdot \mathrm{sim}(S, Q_t)}{\mathrm{sim}(S, Q_t) + 1}$$

と表すことができ，これを式 (3.1) に代入すると，

$$\begin{aligned} \frac{u + |S \cap Q_t|}{-u + |S \cup Q_t|} &= \frac{u + |S \cap Q_t|}{-u + |S| + |Q_t| - |S \cap Q_t|} \\ &= \frac{(|S| + |Q_t| + u) \cdot \mathrm{sim}(S, Q_t) + u}{|S| + |Q_t| - u \cdot \mathrm{sim}(S, Q_t) - u}. \end{aligned} \tag{3.2}$$

となる．式 (3.2) に含まれる変数はクエリと集合のサイズ，進んだ時刻，類似度であり全て既知なので，上限値が過去の類似度 $\mathrm{sim}(S, Q_t)$ から高速に計算できることを示している．

次に，上限値を利用した類似度計算の省略方法について述べる．時刻 T において，枝刈りアルゴリズムはまず一部の集合グループ $R \subset D$ に対してクエリとの類似度を計算し，R の中で top-k を求める．この時，k 番目に類似した集合の類似度は厳密な top-k に対する類似度の下限値 lb となる．R 以外の集合のうち，上限値が lb を下回る集合に対しては類似度の計算をしなくてもよい (枝刈り)．上限値が lb を上回る集合に対しては類似度を計算し，top-k を確定する．どれだけ枝刈りができるかは R に依存し，クエリに類似した集合を多く含むほど枝刈りの効率が良い．Xu らは望ましい R を得るため min_step というタイマーを使用した．min_step は個々の集合 S に対して次のように定義される．

時刻 t に Q_t と S の間で類似度を計算したとする．そして，θ を時刻 t における Q_t と k 番目に最も類似した集合の類似度とする．この時点で，S に対する min_step は，S の類似度の上限が θ を上回るまでの時間に初期化される．つまり，

$$\theta < \frac{(|S| + |Q_t| + min\_step) \cdot \mathrm{sim}(S, Q_t) + min\_step}{|S| + |Q_t| - min\_step \cdot \mathrm{sim}(S, Q_t) - min\_step}$$

$$min\_step = \lceil \frac{(\theta - \mathrm{sim}(S, Q_t) \cdot (|S| + |Q_t|))}{(1 + \theta) \cdot (1 + \mathrm{sim}(S, Q_t))} \rceil$$

となる．

その後，時刻が進むごとに min_step は 1 ずつ減らされる．時刻 T に対する R は，時刻 T に $min\_step = 0$ となった集合のグループである．

3.2　Min-Hash を用いた近似アルゴリズム (MHI)

3.1 節で述べた厳密解法では，1 つの集合 S に対して類似度を 1 回求める計算量は $O(W + |S|)$ であり，この点では自明な解法と変わらない．ここで述べる近似解法 MHI(MinHash-based algorithm using inverted indices) は Min-Hash[3] により，類似度の近似値を $O(1)$ の時間で高速に更新する．Min-Hash は，Jaccard 係数を類似度とした集合の類似検索をハッシュベースで実現する確率的な Locality-Sensitive Hashing (LSH)[2] である．Min-hash による集合 A のハッシュ値 $h(A)$ の計算方法は以下の通りである．

1. ベクトルの列を入れ替える規則 π を決めておく
2. 規則 π に従い，A を並び替える
3. A を並び替えたもののうち最初に現れる要素をハッシュ値とする

2 つの集合 A, B のハッシュ値が一致する確率は，Jaccard 係数と等しいという性質を持つ．すなわち，$P[h(A) = h(B)] = \mathrm{sim}(A, B)$.

この近似解法では複数個のハッシュ関数 $h_1, h_2, \cdots, h_r$ を用意し，クエリ Q_T と各集合との類似度をハッシュ値の一致率によって近似する．このアルゴリズムによる時刻 T での類似度の計算方法について述べる．

まず，あらかじめハッシュ関数ごとにハッシュテーブルを作成しておく．これは，あるハッシュ関数 h_i に対してハッシュ値 x となる集合群を $table_i(x)$ に登録する．時刻 T において，クエリに対するハッシュ値 $h_i(Q_T)$ を計算する $(1 \leq i \leq r)$．このハッシュ値が時刻 $T-1$ におけるハッシュ値 $h_i(Q_{T-1})$ から変化した場合，$table_i(h_i(Q_{T-1}))$ に登録されている集合については h_i についてハッシュ値が新たに一致しなくなった集合群である．よってクエリとの類似度の近似値を $\frac{1}{r}$ だけ小さくする．一方，$table_i(h_i(Q_T))$ に登録されている集合については新たにハッシュ値が一致するようになった集合群のため，クエリとの類似度の近似値を $\frac{1}{r}$ 大きくする．この更新処理の対象は，ハッシュ値が $h_i(Q_{T-1})$ あるいは $h_i(Q_T)$ と一致する集合に限定され，ハッシュ値に対する転置インデックス (つまり，ハッシュテーブル) により必要な集合だけを処理できる．また，ハッシュ値が変化しなかった場合は更新処理が不要である．

この手法は，転置インデックスを用いて処理対象を高速決定すること，及び，$O(1)$ の時刻で類似度の近似値を更新していることが本研究と類似する．しかし，以下の点が相違する．

1. Min-Hash が多重集合に対応していないため，近似解法も多重集合に対応していない．これに対して，我々の提案手法は多重集合も取り扱える．

2. 近似解しか求められない．近似精度はハッシュ関数の数 r とトレードオフの関係にあり，r を増やすほど近似精度があがるが，処理対象の集合も増加し実行速度が低下する．我々の提案手法では厳密解が求まり，このような問題は起きない．

3. 転置インデックスを適用するのが容易な問題設定になっている．

また，複数のハッシュテーブルを使用するため，空間計算量が大きいのもこの近似解法の欠点である．

第4章

EA-FIL

本章では，新しいアルゴリズムEA-FIL(Exact Algorithm using Frequency-based Inverted Lists)を提案する．従来の厳密解法では類似度の上限値を利用して，top-kに入る可能性のない集合に対して類似度計算を省略していた．つまり，類似度の順位に着目して類似度計算を省略する．これに対して，我々は時刻が$T-1$からTに進んだ時に，類似度の値が変化する可能性がない集合に対して，類似度計算を省略するというアプローチを取る．類似度の値が変化する可能性がある集合Sに対しては$\mathrm{sim}(S, Q_T)$を計算するが，これも$\mathrm{sim}(S, Q_{T-1})$の値を再利用して$O(1)$の時間で高速に求める．

4.1 時刻変化に伴う類似度の変化

本節では，時刻が$T-1$からTに進んだ時に類似度がどう変化するかについて考察する．とくに類似度であるJaccard係数の分子，分母をそれぞれ構成するクエリとデータベース内の集合Sの積集合，和集合のサイズの変化に着目する．

時刻$T-1$におけるクエリは$Q_{T-1} = \{e_{T-W}, e_{T-W+1}, \cdots, e_T, e_{T-1}\}$となる．これを$Q_T = \{e_{T-W+1}, e_{T-W+2}, \cdots, e_T\}$と比較すると，時刻$T$において，$e_{T-W}$が離脱し，$e_T$が追加されている．以下では，要素$e_T$を$IN$，要素$e_{T-W}$を$OUT$と記述する．さらに，$Q'_T$を$Q_{T-1}$と$Q_T$の共通部分とする．すなわち，

$$Q'_T = Q_{T-1} \backslash OUT = Q_T \backslash IN.$$

この時，$Q_{T-1} \to Q_T$の変化をOUTの離脱によりQ'_Tになってから，INの追加によりQ_Tになったつまり，$Q_{T-1} \to Q'_T \to Q_T$と考えても正当性を失わない．

後半の $Q'_T \to Q_T$ のステップでは，積集合，和集合の要素数の変化について以下の Theorem 1, 2 がそれぞれ成り立つ.

Theorem 1.

$$\begin{aligned} &\textit{If } IN \in S\backslash Q'_T, |S \cap Q_T| = |S \cap Q'_T| + 1. \\ &\textit{If } IN \notin S\backslash Q'_T, |S \cap Q_T| = |S \cap Q'_T|. \end{aligned}$$

Proof. $IN \in S\backslash Q'_T$ であれば，$IN \in Q_T$ より $IN \in S \cap Q_T$ である．さらに $IN \in S\backslash Q'_T$ より，$IN \in S\backslash S \cap Q'_T$ であり，IN は時刻 T で新たに積集合に加わる要素であるので，$|S \cap Q_T| = |S \cap Q'_T| + 1$.

$IN \notin S\backslash Q'_T$ であれば，(1)$IN \notin S$ あるいは，(2)「$IN \in S$ かつ $IN \notin S\backslash Q'_T$」のどちらかである．(1) の場合は，明らかに $|S \cap Q_T| = |S \cap Q'_T|$.(2) の場合は IN は時刻 T で新たに積集合に加われないのでやはり $|S \cap Q_T| = |S \cap Q'_T|$. □

Theorem 2.

$$\begin{aligned} &\textit{If } IN \in S\backslash Q'_T, |S \cup Q_T| = |S \cup Q'_T|. \\ &\textit{If } IN \notin S\backslash Q'_T, |S \cup Q_T| = |S \cup Q'_T| + 1. \end{aligned}$$

Proof. 集合の性質より，$|S \cup Q_T| = |S| + |Q_T| - |S \cap Q_T|$ が成立する．さらに，$|Q_T| = |Q'_T| + 1$ なので，

$$|S \cup Q_T| = |S| + |Q'_T| + 1 - |S \cap Q_T|.$$

右辺の $|S \cap Q_T|$ に Theorem 1 を代入することで本定理を示せる． □

前半の $Q_{T-1} \to Q'_T$ のステップでは，和集合，積集合の要素数の変化について以下の性質が成り立つ.

Theorem 3.

$$\begin{aligned} &\textit{If } OUT \in S\backslash Q'_T, |S \cap Q'_T| = |S \cap Q_{T-1}| - 1. \\ &\textit{If } OUT \notin S\backslash Q'_T, |S \cap Q'_T| = |S \cap Q_{T-1}|. \end{aligned}$$

Theorem 4.

$$\begin{aligned} &\textit{If } OUT \quad \in \quad S\backslash Q'_T, |S \cup Q'_T| = |S \cup Q_{T-1}|. \\ &\textit{If } OUT \quad \notin \quad S\backslash Q'_T, |S \cup Q'_T| = |S \cup Q_{T-1}| - 1. \end{aligned}$$

Proof. Q'_T を基準として考えると，Q_T も Q_{T-1} もどちらも Q'_T に 1 要素を追加したものである．よって，Theorem 1，Theorem 2 より，以下の 4 つが成り立つ．

1. $OUT \in S\backslash Q'_T \rightarrow |S \cap Q_{T-1}| = |S \cap Q'_T| + 1$
2. $OUT \notin S\backslash Q'_T \rightarrow |S \cap Q_{T-1}| = |S \cap Q'_T|$
3. $OUT \in S\backslash Q'_T \rightarrow |S \cup Q_{T-1}| = |S \cup Q'_T|$
4. $OUT \notin S\backslash Q'_T \rightarrow |S \cup Q_{T-1}| = |S \cup Q'_T| + 1$

これらを $|S \cap Q'_T|$ について解くことで Theorem 3，Theorem 4 が得られる．　□

Theorem 1 から Theorem 4 をルールとして用いると，クエリ Q_{T-1} に対する積集合と和集合のサイズを再利用して，時刻 T の top-k 類似検索を実現できる．その方針に沿った単純なアルゴリズムを Algorithm 2 に示す．このアルゴリズムでは，各 S に対する和積集合サイズを S.intersection, S.union という変数に記憶する．これは時刻 $T+1$ の top-k 類似検索で再利用するためである．

この単純なアルゴリズムでは，Jaccard 係数の更新自体は $O(1)$ で行われるが，D 内の全集合に対して処理が実行され，さらに IN や OUT が $S_i\backslash Q'_T$ に含まれるかの判定に $O(|S_i| + W)$ の時間がかかるため，自明な解法と同程度の速度となる．

後述する我々の提案手法は転置インデックスにより全集合を処理するのを回避し，高速化を実現する．通常，転置インデックスは，処理対象が「特定の要素を含むデータ群」に限定される場合に，それらを高速発見するためのデータ構造である．しかし，上述した単純アルゴリズムでは，IN や OUT を含む集合群だけではなく，IN や OUT を含まない集合群も処理する必要がある．このため，転置インデックスを用いた処理高速化は自明ではない．

以降，4.2 節で類似度更新ルールを，$IN \in (S\backslash Q'_T)$，$OUT \in (S\backslash Q'_T)$ のみを処理対象とするものに書き換える．4.3 節では $IN \in (S\backslash Q'_T)$ を満足する集合群 $\{S \in D | IN \in (S\backslash Q'_T)\}$ に高速アクセスするためのデータ構造として頻度別の転置インデックスを提案する．そして，4.4 節でこのデータ構造を利用した単純アルゴリズムの高速化手法を述べる．この高速化手法が提案手法となる．

Algorithm 2 類似度更新アルゴリズム

1: **for** $i \leftarrow 1$ to n **do**
2:　**if** $OUT \in (S_i \backslash Q'_T)$ **then**
3:　　S_i.intersection を 1 減らす.
4:　**else**
5:　　S_i.union を 1 減らす.
6:　**end if**
7:　**if** $IN \in (S_i \backslash Q'_T)$ **then**
8:　　S_i.intersection を 1 増やす.
9:　**else**
10:　　S_i.union を 1 増やす.
11:　**end if**
12:　$\mathrm{sim}(S_i, Q_T) = \frac{S_i.\mathrm{intersection}}{S_i.\mathrm{union}}$.
13: **end for**
14: 類似度が大きい順に集合を k 個取り出す.

4.2　類似度更新ルールの書き換え

本節では，前節で考察した類似度更新パターンを，転置インデックスを用いた高速化に対応させるために書き換える．具体的には，類似度更新条件に含まれる特定の要素を含まない集合 $(IN \notin S \backslash Q'_T, OUT \notin S \backslash Q'_T)$ を処理対象から取り除く．

Algorithm2 の単純法をよく見ると，$OUT \in (S_i \backslash Q'_T)$ かつ $IN \in (S_i \backslash Q'_T)$ の場合は，intersection が 1 減った後に 1 増えるので，intersection の値は for ループの実行により変化しない．同様に $OUT \notin (S_i \backslash Q'_T)$ かつ $IN \notin (S_i \backslash Q'_T)$ の場合は，union が 1 減った後に 1 増えるので，union の値は不変である．集合 S は，$IN \in S \backslash Q'_T$, $OUT \in S \backslash Q'_T$ を満たすかどうかで，4 パターンに分類される．以下にそのパターンと類似度の変化を示す．

パターン 1). $OUT \in (S_i \backslash Q'_T)$ かつ $IN \in (S_i \backslash Q'_T) \rightarrow$ 変化なし

パターン 2). $OUT \notin (S_i \backslash Q'_T)$ かつ $IN \in (S_i \backslash Q'_T) \rightarrow$intersection が-1，union が+1

パターン 3). $OUT \in (S_i \backslash Q'_T)$ かつ $IN \notin (S_i \backslash Q'_T) \rightarrow$intersection が+1，union が-1

パターン 4). $OUT \notin (S_i \backslash Q'_T)$ かつ $IN \notin (S_i \backslash Q'_T) \rightarrow$ 変化なし

従って，単純法における 4 つの条件文は，以下の 2 条件に集約できる．

条件 1). $OUT \in (S_i \backslash Q'_T)$ かつ $IN \notin (S_i \backslash Q'_T)$ ならば，intersection を 1 減らし，union を 1 増やす．

条件 2). $IN \in (S_i \backslash Q'_T)$ かつ $OUT \notin (S_i \backslash Q'_T)$ ならば，intersection を 1 増やし，union を 1 減らす．

ただし，これでは相変わらず IN や OUT を含まない集合群を処理しており，転置インデックスを用いた高速化に向かない．

そこで上記の 2 条件をさらに次のように書き換えて IN や OUT を含まない集合群の処理を排除する．

条件 1). $OUT \in (S_i \backslash Q'_T)$ ならば，intersection を 1 減らし，union を 1 増やす．

条件 2). $IN \in (S_i \backslash Q'_T)$ ならば，intersection を 1 増やし，union を 1 減らす．

書き換え後の類似度更新ルールが類似度変更パターンを満たしているかを検討する．パターン 1 の場合，条件 1,2 をともに満たすため，intersection を 1 減らし，union を 1 増やした後，intersection を 1 増やし，union を 1 減らし，結局類似度は変化しない．パターン 2 の場合，条件 1 に合致するため，intersection を 1 減らし，union を 1 増やす．パターン 3 の場合，条件 2 に合致するため，intersection を 1 増やし，union を 1 減らす．パターン 4 の場合，類似度の更新処理が行われず，類似度は変化しない．以上より，書き換え前と全く同じ動きになる．

書き換え後の 2 条件は，転置インデックスを利用して高速化可能である．

4.3　頻度別の転置インデックス

標準的な転置インデックスでは，アルファベットの要素 $x \in \Phi$ に対して x を含む集合のグループ $\{S \in D | x \in S\}$ を記憶する．従って，集合 S が x を含むという情報は持つが，x を何個含むかという情報は保持しない．これに対して，頻度別の転置インデックスでは $x \in \Phi$ と自然数 α に対して，2 次元のリスト $l(x, \alpha)$ を用意し，x を α 個含む集合のグループを記憶する．頻度別の転置インデックスを図 4–1 に図示する．例えば，この図において S_2 は x_2 を 2 つ含むため，$l(x_2, 2)$ に登録されている．頻度別の転置インデックスはデータベース D を 1 回スキャンするだけで構築可能で，登録される集合ののべ数も標準的な転置インデックスから増えない．

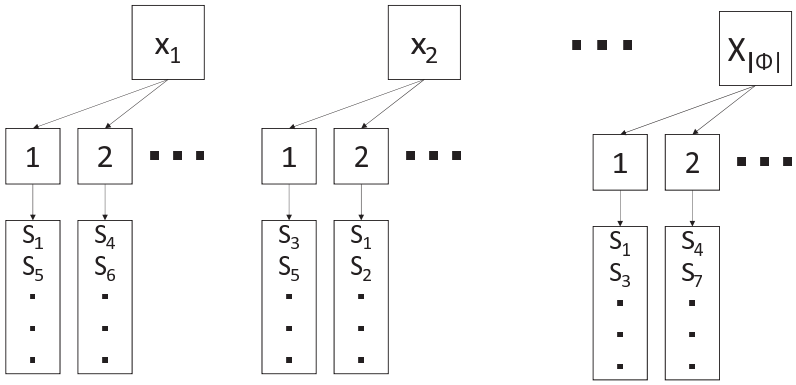


図 4–1: 頻度を考慮した転置インデックス

頻度別の転置インデックスを用いると，4.1 節で述べた $IN \in (S \backslash Q'_T)$ を満足する集合群 $\{S \in D | IN \in (S \backslash Q'_T)\}$ を高速に発見できる．$IN \in (S \backslash Q'_T)$ が成立する必要十分条件は，S が Q'_T より要素 IN を真に多く含むことである．よって，Q'_T が IN を含む回数を β とすると，IN を $\beta + 1$ 回以上含むリスト $\{l(IN, \gamma) | \gamma \geq \beta + 1\}$ を走査すれば，$\{S \in D | IN \in (S \backslash Q'_T)\}$ が発見できる．また，毎時刻 T において，Q'_T がもつ IN の数を高速に求めるため，クエリが各アルファベットをいくつ含んでいるかをヒストグラムで管理しておく．このヒストグラムは，Q_1 のとき，Q_1 に含まれる要素を数えることで作成でき，$T \geq 2$ では，時刻が $T-1$ から T に進むとき，IN を 1 増やし，OUT を 1 減らすことで簡単に更新できる．

同様に，$OUT \in (S \backslash Q'_T)$ を満足する集合群 $\{S \in D | OUT \in (S \backslash Q'_T)\}$ も検出できる．

4.4　高速な厳密解アルゴリズム

本節では，頻度別の転置インデックスを用いて 4.1 節で述べた単純法を高速化する提案手法を述べる．

時刻 T における提案手法での top-k 検索アルゴリズムを Algorithm3 に示す．このアルゴリズムは，各集合 S に対してクエリとの積集合のサイズ，和集合のサイズをそれぞれ S.intersection, S.union という変数に記憶する．時刻 T で処理対象とならなかった集合に対しては時刻 $T-1$ での S.intersection, S.union，類似度の値

Algorithm 3 提案手法

1: **if** $IN \neq OUT$ **then**
2:　転置インデックスで，$OUT \in (S \backslash Q'_T)$ を満たす S を見つける．
3:　**for** S such that $OUT \in (S \backslash Q'_T)$ **do**
4:　　S.intersection を 1 減らし，S.union を 1 増やす．
5:　　$\mathrm{sim}(S, Q_T) = \frac{S.\mathrm{intersection}}{S.\mathrm{union}}$.
6:　**end for**
7:　転置インデックスで，$IN \in (S \backslash Q'_T)$ を満たす S を見つける．
8:　**for** S such that $IN \in (S \backslash Q'_T)$ **do**
9:　　S.intersection を 1 増やし，S.union を 1 減らす．
10:　　$\mathrm{sim}(S, Q_T) = \frac{S.\mathrm{intersection}}{S.\mathrm{union}}$.
11:　**end for**
12: **end if**
13: 類似度が大きい順に集合を k 個取り出す．

がそのまま時刻 T に持ち越される点に注意されたい．

なお，1 行目の if 文では，$IN = OUT$ の時は $Q_{T-1} = Q_T$ なので明らかに類似度は不変なので，その場合は処理を避けている．

提案手法の長所を以下にまとめる．

- 3 行目，8 行目での for 文は頻度別の転置インデックスにより処理対象となる集合が高速に発見される．4.1 節で述べた単純法と異なり，$IN \notin (S \backslash Q'_T)$ かつ $OUT \notin (S \backslash Q'_T)$ となる集合 S が処理されないで済む．この条件を満足する集合の数が多いほど，提案手法の効率は向上する．

- Jaccard 係数の更新処理自体は $O(1)$ で実現される (4-5 行目，9-10 行目)．

以上より，時刻が $T-1$ から T に進んだ時に提案手法では，(1) 類似度の値が変化する可能性がない集合に対して類似度計算を省略し，(2) 類似度の値が変化する可能性がある集合に対しても類似度の更新を $O(1)$ の時間で完了することで，各時刻での処理時間を削減する．

4.5　理論的な計算量

本節では，提案手法 EA-FIL の計算量の理論解析を行う．EA-FIL では，Jaccard 係数の更新を集合サイズによらず $O(1)$ で実現する．そのため，計算量は転置インデックスを走査して何個の集合を処理するかに依存する．そこで，転置インデックスに登録されている集合の数を理論的に解析し，その結果から EA-FIL の計算量について論じる．簡単のためデータベースに登録されている集合について以下を仮定する．

- 集合サイズは全て W である．
- 集合の要素は全て Φ から一様分布に従って選択されている．

4.5.1　1 つの転置インデックスあたりの登録集合数

まず，1 つの集合 S は S が要素として含むアルファベットの転置インデックスに登録される．$|\Phi|$ が大きい場合，1 つの集合に同じ種類の要素が入りにくくなり，マルチセットを作りにくい．S がマルチセットでないとすると W 個の転置インデックスに登録される．よって，集合数 n，アルファベット種類数 $|\Phi|$ より，各アルファベットの転置インデックスには平均で $\frac{nW}{|\Phi|}$ 個の集合が登録される．

次に，$|\Phi|$ が小さい場合，集合に含まれるアルファベットに重複が起こりやすくなる．この重複を考慮した転置インデックス 1 つあたりの登録集合数を考える．

まず，$|\Phi|$ 種類のアルファベットから i 個のアルファベットをランダムに選択したとき，集まったアルファベットの種類数の期待値を X_i とする．その場合，X_W が集合サイズ W の集合に含まれるアルファベットの種類数にあたる．X_i は，i 回アルファベットを選び，X_{i-1} 種類のアルファベットを集めたあと，1 回アルファベットを選択する．i 回目の試行で，新しいアルファベットを選択する確率は，$\frac{|\Phi|-X_{i-1}}{|\Phi|}$ である．よって，X_i は以下の式で表すことができる．

$$\begin{aligned} X_i &= X_{i-1} + \frac{|\Phi| - X_{i-1}}{|\Phi|} \\ &= 1 + \frac{|\Phi| - 1}{|\Phi|} X_{i-1} \end{aligned}$$

そして，X_1 は当然 1 となる．ここで，$\frac{|\Phi|-1}{|\Phi|} = q$ とすると，

$$\begin{aligned}
X_1 &= 1 \\
X_2 &= 1 + q \cdot X_1 \\
&= 1 + q \\
X_3 &= 1 + q \cdot X_2 \\
&= 1 + q(1+q) \\
&= 1 + q + q^2 \\
&\vdots \\
X_i &= 1 + q \cdot X_{i-1} \\
&= 1 + q(1 + q + q^2 \cdots q^{i-2}) \\
&= 1 + q + q^2 + \cdots q^{i-1}
\end{aligned}$$

となる．ここで，等比級数の和の公式を用いて X_W は，以下のようになる．

$$\begin{aligned}
X_W &= 1 + q + q^2 + \cdots q^{W-1} \\
&= \frac{1 - q^W}{1 - q} \\
&= \frac{1 - \left(\frac{|\Phi|-1}{|\Phi|}\right)^W}{1 - \frac{|\Phi|-1}{|\Phi|}} \\
&= |\Phi| \left(1 - \left(\frac{|\Phi| - 1}{|\Phi|}\right)^W\right) \\
&= |\Phi| \left(1 - \left(1 - \frac{1}{|\Phi|}\right)^W\right)
\end{aligned}$$

1 つの集合 S に含まれるアルファベットの種類数は，$|\Phi| \left(1 - \left(1 - \frac{1}{|\Phi|}\right)^W\right)$ であり，この数だけ S は転置インデックスに登録される．データベース D には集合が n 個あり，これが $|\Phi|$ 個の転置インデックスに登録される．よって，転置インデックス 1 つあたりに登録される集合数は，

$$|\Phi| \left(1 - \left(1 - \frac{1}{|\Phi|}\right)^W\right) \times \frac{n}{|\Phi|} = n \left(1 - \left(1 - \frac{1}{|\Phi|}\right)^W\right)$$

となる．

4.5.2　EA-FIL の計算量

EA-FIL では，毎時刻 T において，$IN \in (S \backslash Q'_T)$ および，$OUT \in (S \backslash Q'_T)$ を満たす S について処理をする．ここで，$IN \notin Q'_T$ であれば，IN の転置インデックスに登録されたすべての集合が処理対象となる．また，$IN \in Q'_T$ であれば，IN の転置インデックスに登録されている集合の内一部が処理対象である．つまり，EA-FIL は 1 回の処理において，最大で IN と OUT の転置インデックスに登録されている全集合を $O(1)$ で処理する．

以上より，EA-FIL の計算量は最大で 4.5.1 節で述べた 1 つの転置インデックスあたりの集合の登録数に比例する．

4.6　top-k の求め方

提案手法ではすべての集合に対しての類似度を計算した後，類似度の大きい順に k 個取り出す処理がある．この処理を高速化する方法について説明する．

単純に上位 k 個の集合を求めるには，全集合を類似度順にソートする方法や，Max ヒープを使い上位 k 個を取り出すなどが考えられる．しかし，全データをソートすると $O(n \log n)$，Max ヒープを作る場合でも，全ての集合と類似度のペアをヒープに挿入するのに $O(\log n!)$ の計算量がかかる．

本研究では，Min ヒープを使い，これを $O(n \log k)$ 以下で実現する．まず，最初の k 個の集合は，単に Min ヒープに挿入すればよい．$(k+1)$ 個目以降の集合については，ヒープの根の集合 root と比較し，root よりも類似度が小さければヒープを更新しない．root よりも類似度が大きければ root をヒープから削除し，その集合をヒープに挿入する．これを全集合について行い，最後にヒープに残った k 個の集合を top-k とする．この過程で，ヒープのサイズが k を上回ることはなく，データの削除および追加は $O(\log k)$ で処理ができる．よって，全体の計算量は $O(n \log k)$ になる．$\forall S_i \in D$ に対する $\mathrm{sim}(S_i, Q_T)$ から top-k を求めるアルゴリズムを Algorithm 4 に示す．

Algorithm 4 top-k を求めるアルゴリズム

1: **for** $i \leftarrow 1$ to k **do**
2:　$\mathrm{sim}(S_i, Q_T)$ を Min ヒープに入れる.
3: **end for**
4: **for** $i \leftarrow k+1$ to n **do**
5:　**if** root $< \mathrm{sim}(S_i, Q_T)$ **then**
6:　　root を取り出す.
7:　　$\mathrm{sim}(S_i, Q_T)$ を Min ヒープに入れる.
8:　**end if**
9: **end for**
10: ヒープ内のデータを top-k とする.

第5章

拡張した検索問題

本章では，CSPEQ 問題 [1] を拡張した新たな問題設定を提唱する．CSPEQ 問題は，ウインドウサイズが固定であり，集合サイズが大きく違うデータは検索結果となりにくい．

新しい問題設定では，ウインドウサイズを可変にし，集合サイズの違いによって，類似度が小さくなってしまう問題を緩和し古い要素よりも新しい要素を重視する新しい類似度を導入する．

5.1 CSPEQ 問題の問題点

CSPEQ 問題 [1] は，固定された長さのウインドウサイズとデータベース内の集合のサイズとの差の影響を受けてしまうという欠点がある．これは，Jaccard 係数は集合のサイズが大きく違うと，上限が小さくなってしまうからである．例えば，集合 A，B があり，$|A| \leq |B|$ とすると，

$$\frac{|A \cap B|}{|A \cup B|} \leq \frac{|A|}{|B|}$$

である．

集合サイズの差の影響を受け，検索結果が不適当となる例を示す．集合 $S_1 = \{a, a, b, f, g\}$，集合 $S_2 = \{a, a, a, b, c, d, e, f, f, g\}$ とし，時刻 T においてストリームデータの直近の 10 個の要素が新しい順に $\{a, b, g, a, f, f, f, g, a, b\}$ とする．ここで，$W = 10$ とすると，$\mathrm{sim}(S_1, Q_T) = 0.5$，$\mathrm{sim}(S_2, Q_T) = 0.54$ である．しかし，ストリームデータの直近の 5 つの要素をみると，集合 S_1 と完全に一致している．このような例の場合，最近の嗜好に合致するオブジェクトを推薦する状況を想定すると，S_1 のほうが適当である．

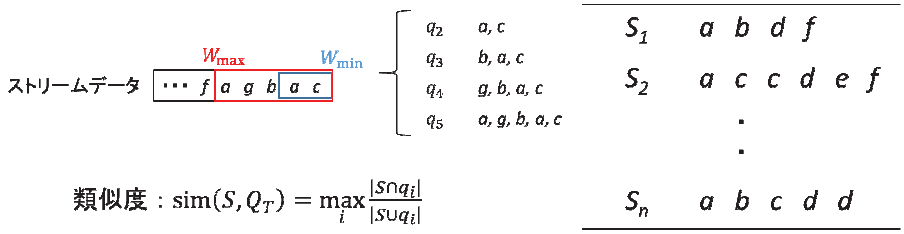


図 5–1: 拡張した問題設定

以上の問題点を踏まえ，CSPEQ 問題を拡張した新しい問題設定を提案する．

5.2　問題定義

$\Phi = \{x_1, x_2, \cdots, x_{|\Phi|}\}$ をアルファベット集合とし，データストリームと，データベース D は，CSPEQ 問題と同様とし，類似する上位 k 個の集合を検索する問題とする．ここで，スライディングウインドウのサイズの最小を $W_{\min}$，最大を $W_{\max}$ とする．時刻 T において，データストリームの直近の $W_{\min}, W_{\min}+1, \cdots, W_{\max}$ 個の要素集合をそれぞれ，小クエリ $q_{\min}, q_{\min+1}, \cdots, q_{\max}$ とする．時刻 T におけるクエリ Q とデータベース D 内の集合 S との類似度を以下のように定義する．

$$\mathrm{sim}(S, Q) = \max_i \frac{|S \cap q_i|}{|S \cup q_i|}$$

新しく導入した類似度であれば，集合サイズが $W_{\min} \sim W_{\max}$ であれば，最大が 1 となり，集合サイズの差による類似度の低下を軽減できる．

5.3　単純な解法

この問題に対して，単純な解法は CSPEQ 問題を解くアルゴリズムを $W_{\min} \sim W_{\max}$ のそれぞれのウインドウサイズについて適用し，集合ごとにどのウインドウが最大値をとるかを決定してから，上位 k を求める方法である．CSPEQ 問題を解くアルゴリズムとして EA-FIL を使った場合の手順を以下に示す．

まず，全集合に対して Jaccard 係数，和集合サイズ，積集合サイズを $W_{\min} \sim W_{\max}$ のそれぞれのウインドウサイズについて保存しておく．また，小クエリが各アルファベットをいくつ含んでいるかを示すヒストグラムも全小クエリについて用意する．時刻 T において，全小クエリ $q_{\min}, q_{\min+1}, \cdots, q_{\max}$ と各集合との Jaccard 係数，和集合サイズ，積集合サイズを EA-FIL によって更新する．各集合

について全小クエリのうち Jaccard 係数が最大のものをその集合とクエリの時刻 T における類似度とする．最後に類似度が大きい上位 k 個の集合を検索結果とする．

このアルゴリズムには，以下の欠点がある．

- 各ウインドウサイズについて Jaccard 係数，和積集合サイズを保存しておく必要があり，空間計算量が大きくなる
- 全てのウインドウサイズにおいて新しく入ってきた要素 IN は共通であり処理対象となる集合もほぼ同じであるが，各ウインドウサイズについて処理が必要である

これらを解消するアルゴリズムを提案する．

5.4　VWEA-FIL

この節では拡張した CSPEQ 問題を高速に解くアルゴリズム VWEA-FIL (Variable Window size EA-FIL) を提案する．VWEA-FIL は，この問題に対して，EA-FIL を適用したものを改良したものである．これにより，前述の EA-FIL を単純に適用した場合に対し，処理対象をおよそ半分にし，空間計算量も削減する．

この手法の時刻 T における類似度更新の流れは以下の通りである．

1. 各集合 S に対して，ウインドウサイズが最小の時の Jaccard 係数を求める
2. ウインドウサイズを 1 を増やし，Jaccard 係数が大きくなる可能性のある集合に対して，Jaccard 係数を計算し類似度を更新する
3. ウインドウサイズが最大になるまで，2. を繰り返す

まず，ウインドウサイズが $W_{\min}$ の時の各集合 S と小クエリ $q_{\min}$ との和集合サイズ，積集合サイズ，Jaccard 係数を EA-FIL を使って更新する．ここで，和集合サイズ，積集合サイズをそれぞれ $S.\mathrm{intersection}, S.\mathrm{union}$ という変数に記憶する．また，類似度 $\mathrm{sim}(S_i, Q_T)$ の初期値として Jaccard 係数を代入する．$q_{\min+1}$ は，$q_{\min}$ に，要素が 1 つ追加されたものであり，ある集合 S に対して，

$$\frac{|S \cap q_{\min}|}{|S \cup q_{\min}|} < \frac{|S \cap q_{\min+1}|}{|S \cup q_{\min+1}|}$$

となるのは，追加された要素が新たに共通要素となったときである．つまり，クエリ q_i で新たに追加された要素 $q_i \backslash q_{i-1}$ を NEW とすると，類似度が大きくなる条件は，

$$NEW \in S \backslash q_{i-1}$$

である．これは，FIL によって高速発見可能である．また，小クエリ $q_{\min+x}$ と集合 S との積集合サイズ S.intersection は 4.1 節の定理 1 より，$NEW \in S \backslash q_{\min+x-1}$ を満たすとき 1 増える．さらに，$q_{\min+x}$ と S との和集合サイズ S.union は以下の式で求められる．

$$S.\text{union} = W_{\min} + x + |S| - S.\text{intersection}$$

S.intersection を S.union で割ることで，$q_{\min+x}$ と S との Jaccard 係数を求めることができる．この値が $q_{\min} \sim q_{\min+x-1}$ までの S との Jaccard 係数の最大値 $\text{sim}(S, Q_T)$ より大きければ類似度を更新する．

このアルゴリズムにおける FIL による処理対象の高速発見でも EA-FIL と同様にクエリが各アルファベットをいくつ含むかをヒストグラムで管理する．まず，$q_{\min}$ に関するヒストグラムは毎時刻更新し，ウインドウサイズが $W_{\min}$ の時の EA-FIL による Jaccard 係数の更新に利用する．そして，ウインドウサイズが $W_{\min+1} \sim W_{\max}$ のときは，毎時刻 T において一時的に使用するヒストグラムを用意し，クエリに含まれる各アルファベットの数を管理する．具体的には，ウインドウサイズが $W_{\min+1}$ の時の処理を行う際に，$q_{\min}$ に関するヒストグラムを一時的なヒストグラムにコピーする．その後，クエリが q_{i-1} から q_i に変わるごとに一時的なヒストグラムの NEW を 1 増やすことで，簡単に更新できる．

このアルゴリズムでは，各集合に対して保存しておく必要があるのは，$W_{\min}$ に対する Jaccard 係数，和積集合サイズ，ヒストグラムであり，一時変数として S.intersection，S.union，$\text{sim}(S, Q_T)$ と一時的なヒストグラムを用意しておけばよい．さらに，NEW は各小クエリにおける OUT に相当し，IN に関しては最初の 1 回しか処理を行っていない．そのため，必要な処理の量は EA-FIL を単純に適用した場合のおよそ半分となる．

VWEA-FIL のアルゴリズムを Algorithm 5 に示す．

Algorithm 5 は，時刻 T における類似度の更新方法である．まず，1 行目でウインドウサイズが最小の時の Jaccard 係数を更新している．次に，2 行目から 6 行目の **for** 文で時刻 T における S に対する類似度と S.intersection，S.union の初期値

Algorithm 5 VWEA-FIL

1: EA-FIL で $\frac{|S\cap q_{\min}|}{|S\cup q_{\min}|}$ を更新
2: **for** $i \leftarrow 1$ to n **do**
3: $S_i.\text{intersection} = |S_i \cap q_{\min}|$
4: $S_i.\text{union} = |S_i \cup q_{\min}|$
5: $\text{sim}(S_i, Q_T) = \frac{S_i.\text{intersection}}{S_i.\text{union}}$
6: **end for**
7: **for** $x \leftarrow 1$ to $\max - \min$ **do**
8: 転置インデックスで，$NEW \in (S \backslash q_{\min+x-1})$ を満たす S を見つける．
9: **for** S such that $NEW \in (S \backslash q_{\min+x-1})$ **do**
10: S.intersection を+1
11: $S.\text{union} = W_{\min} + x + |S| - S.\text{intersection}$
12: **if** $\text{sim}(S, Q_T) < \frac{S.\text{intersection}}{S.\text{union}}$ **then**
13: $\text{sim}(S, Q_T) = \frac{S.\text{intersection}}{S.\text{union}}$
14: **end if**
15: **end for**
16: **end for**
17: 類似度が大きい順に集合を k 個取り出す．

を設定している．ここで，$|S \cap q_{\min}|$，$|S \cup q_{\min}|$ を S.intersection と S.union の変数にコピーしているのは，時刻 $T+1$ において，$|S \cap q_{\min}|$，$|S \cup q_{\min}|$ を再利用するためである．7 行目から 16 行目の **for** 文で，類似度を求めている．まず，7 行目でウインドウサイズが広がったときに類似度が大きくなる可能性のある集合群を転置インデックスを使って見つける．それらの類似度が現在のウインドウサイズのクエリとの Jaccard 係数より小さければ類似度を更新する．これをウインドウサイズ最大まで繰り返す．最後に top-k を求める．

第6章

実験

CSPEQ 問題 [1] に対して EA-FIL と既存手法で検索実験を行い，その時間計算量と空間計算量を比較する．また，拡張した問題に対しても，VWEA-FIL と単純な EA-FIL の適用との性能比較をする．

実験のプラットフォームは，メモリ 8GB，Ubuntu14.04，Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz である．

6.1 CSPEQ 問題

人工データと実データを使用し，EA-FIL の性能を実験的に評価する．

6.1.1 人工データ

人工データを以下の方法で生成し，評価実験のデータベースおよび，ストリームデータとした．

- ランダムに生成
- IBM Quest data generator で生成

まず，ランダムな人工データの生成方法を説明する．スライディングウインドウのサイズを W として，データベース D 内の n 個の集合は，要素数を $0.8W \sim 1.2W$ の範囲でランダムに決定し，その数だけアルファベット集合 Φ から要素をランダムに発生させて作成した．一方，ストリームデータには毎時刻 Φ からランダムに選択された要素が追加される．

IBM Quest data generator によるデータ生成は，先行研究 [1] と同じである．IBM Quest data generator は，集合サイズの平均と集合数を指定することで，集合群を生成するプログラムである．本実験では，ウインドウサイズ W を平均値として設定した．また，ストリームデータは，データベース内で要素数が $0.8W$ 以上から $1.2W$ 以下となる集合をランダムに選択して連結後，各時刻において先頭から 1 要素ずつ入力することで模擬した．

データ生成に関するパラメータをまとめると以下のとおりである．

- スライディングウインドウのサイズ W
- アルファベットの種類 $|\Phi|$
- データベース内の集合数 n

6.1.2　実データ

以下の 2 種類のデータセットをデータベースとして検索実験を行った．

- Market Basket dataset[4]
- Click Stream data set[5]

いずれのデータセットも先行研究 [1] で用いられたものである．Market Basket dataset は，複数の買い物客がそれぞれ購入したものを記録したものである．以下にデータセットのパラメータを示す．

- 商品 (アルファベット) の種類 $|\Phi| = 16470$
- データベース内の集合数 $n = 88162$

また，データベース内の集合の要素数の平均は 10.3 であったため，スライディングウインドウのサイズ $W = 10$ とした．

Click Stream data set は，MSNBC のウェブサイトでユーザーがクリックした URL を順に記録したものであり，各 URL は”news” や”tech” のようにカテゴリに量子化されている．このデータセットのパラメータは以下のとおりである．

- カテゴリ (アルファベット) の種類 $|\Phi| = 17$
- データベース内の集合数 $n = 31790$

データベース内の集合の要素数の平均は 13.33 であったため，スライディングウインドウのサイズ $W = 13$ とした．

実データの検索実験におけるストリームデータの作り方は，IBM Quest data generator による実験と同様である．

6.1.3　時間計算量の評価実験

この人工データ，実データの下で，各時刻 T において，クエリ Q_T と類似度が高い k 個の集合を検索する検索実験を実施した．時刻を $T = 1$ から，$T = 1000$ まで進め，1000 回の top-k 類似検索にかかる合計の処理時間を測定し，以下の 4 つの手法の性能を比較した．

- BFM:自明な解法
- GP:枝刈りベースの従来の厳密解法
- MHI:Min-hash ベースの従来の近似解法
- EA-FIL

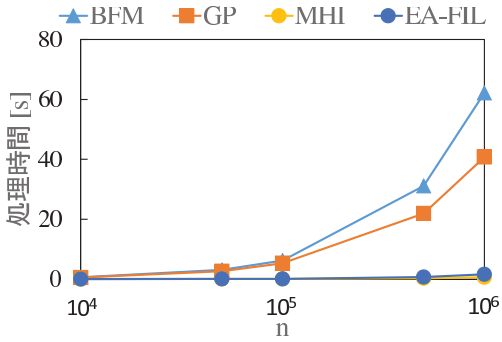
また，本実験では，近似解法のハッシュ関数の数 $r = 100$ とした．

(1) 集合数 n に対する処理時間の変化

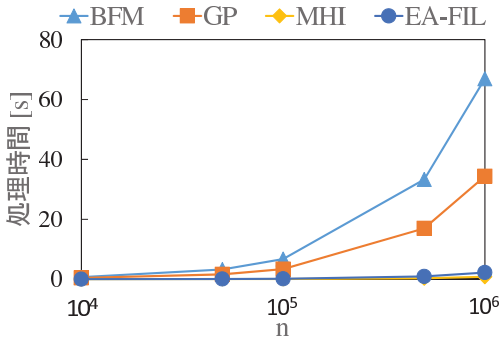
$k = 10$，$W = 10, |\Phi| = 10000$ の条件で集合数 $n = 10000, 50000, 100000, 500000, 1000000$ と変化させた時の実行時間を図 6–1 に示す．

図 6–1 より，人工データをランダムに生成した場合も，IBM Quest data generator で生成した場合も EA-FIL は GP と比べると実行時間が 10 倍以上早くなっており，EA-FIL の有効性が示されている．

また，近似解法と比較しても EA-FIL は同等の速度で処理をできていることがわかる．さらに，MHI は近似解なのに対し，EA-FIL は厳密解を求めることができる．MHI の正解率は，ランダムに生成したデータを使った実験において $n = 10000$ のときが最も低く 33%，$n = 100000$ のときが最も高く 54%であり，IBM Quest data generator を使った実験では，$n = 10000$ のときが最も低く 80%，$n = 500000$ のときが最も高く 85% であった．本実験では，ハッシュ関数の数 $r = 100$ でこれを増やすことで精度向上が見込めるが，実行時間が遅くなってしまう．



(1) ランダムに生成

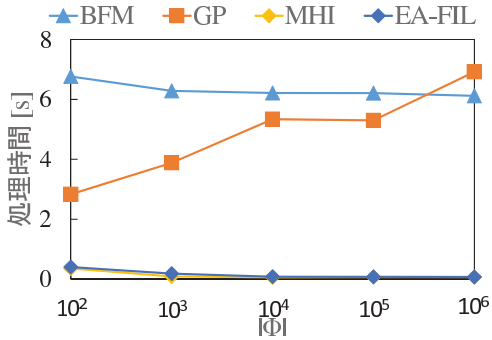


(2)IBM Quest data generator で生成

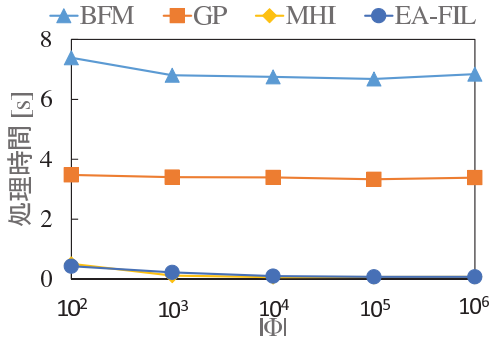
図 6–1: データ数 n を変化させたときの実行時間

(2) 要素の種類数 $|\Phi|$ に対する処理時間の変化

次に，$k = 10$，$W = 10, n = 100000$ の条件で，パラメータ $|\Phi| = 100, 1000, 10000, 100000, 1000000$ に変化させた時の実行時間を図 6–2 に示す．



(1) ランダムに生成



(2)IBM Quest data generator で生成

図 6–2: ラベル種類数 $|\Phi|$ を変化させたときの実行時間

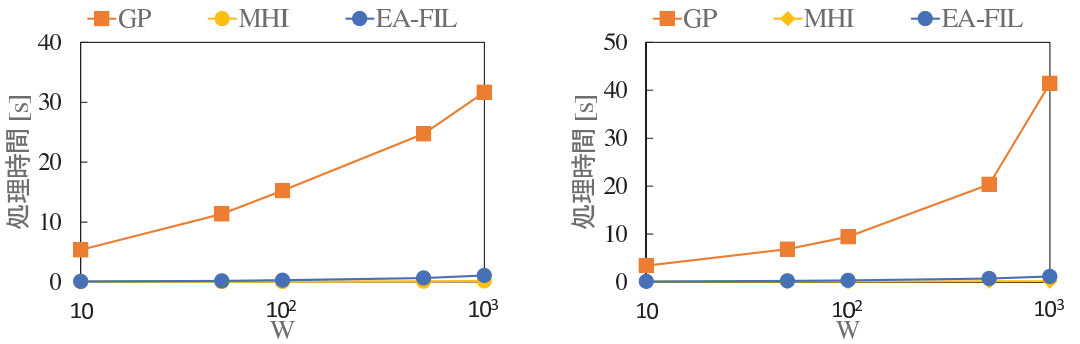
すべての $|\Phi|$ に対して，EA-FIL が GP よりも実行時間が短く，近似解法と匹敵する速度を達成する．ただし，$|\Phi|$ が小さくなると，EA-FIL の速度が低下する傾向が見られる．これは転置インデックスのリスト数が $|\Phi|$ に比例して減少するため，n を固定した条件では転置インデックスの 1 つのリストに登録された集合数が増加して，処理対象の集合が増えるためである．同様の傾向が転置インデックスを利用する MHI にも見られる．

一方，ランダムに人工データを生成した実験において従来手法では $|\Phi|$ が大き

くなるに連れて実行速度が低下し，$|\Phi|$=1000000 の時には単純解法よりも遅くなってしまう．この理由は $|\Phi|$ を大きくしてランダムに集合を生成すると，集合間類似度の期待値が下がり，クエリに対して top-k となる集合の類似度も低下し，枝刈りが困難になるためである．この事は，クエリに対して top-k 類似度の値が低い状況では，枝刈りを適用することが一般に困難になるということを示唆する．

(3) ウインドウサイズ W に対する処理時間の変化

$k = 10$，$|\Phi| = 10000, n = 100000$ の条件で，パラメータ $W = 10, 50, 100, 500, 1000$ に変化させた時の実行時間を図 6–3 に示す．ただし，単純手法は，遅すぎるため，このグラフには掲載していない．



(1) ランダムに生成　　(2)IBM Quest data generator で生成

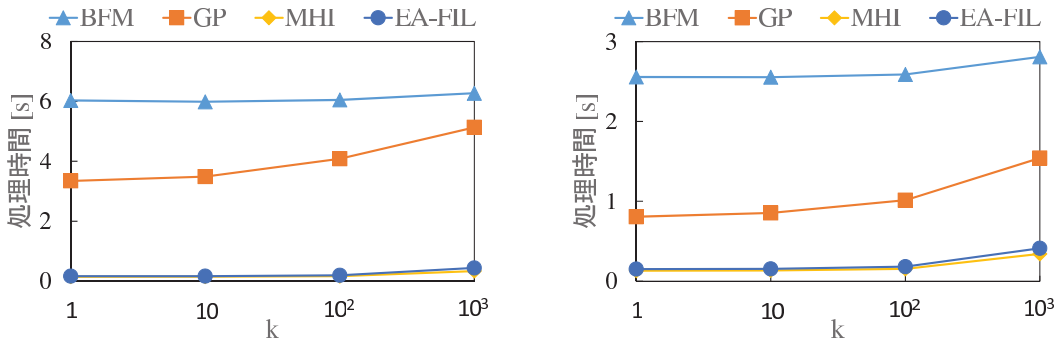
図 6–3: ウインドウサイズ W を変化させたときの実行時間

この実験でも，EA-FIL が GP を圧倒し，MHI に匹敵する速度を達成した．W が大きくなるに連れ，EA-FIL と GP の速度差が拡大する．例えば，ランダムに生成したデータを使った実験で $W = 1000$ の時，GP の実行時間は 31.8 秒で，EA-FIL の実行時間は 1.08 秒であり速度差は約 30 倍になる．この速度差は，EA-FIL では類似度の更新処理が $O(1)$ で実行できるのに対し，GP では類似度の計算量が $O(W)$ となることに起因する．

(4) 実データ

最後に，Market Basket dataset と Click Stream data set の二つの実データに対してパラメータ $k = 1, 10, 100, 1000$ に変化させた時の実行時間を図 6–4 に示す．

人工データに対する実験結果と同様に，EA-FIL は GP を大幅に上回り，MHI に



(1)Market Basket dataset　　(2)Click Stream data set

図 6–4: 実データに対する top-k 類似検索の実行時間

匹敵する実行速度を達成した．特に，Click Stream data set は $|\Phi|$ が人工データに対して設定した値よりも小さく，転置インデックスベースの EA-FIL と MHI には大変不利な状況であるが，それでも GP より圧倒的に高速であり，有効性が示された．

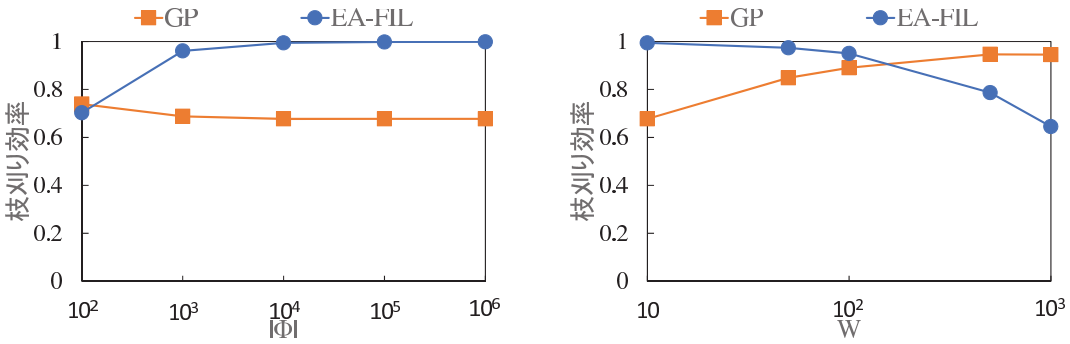
(5) 枝刈り効率について

従来手法 GP も，提案手法 EA-FIL もクエリとの類似度を計算する必要のない集合を見つけ，その計算を省略 (枝刈り) することで高速化をしている．この枝刈り効率を以下のように定義する．

$$\text{枝刈り効率} = \frac{n - \text{時刻 } T \text{ で処理をした集合の数}}{n}$$

図 6–5 に IBM Quest data generator で生成したデータに対する検索実験における GP と EA-FIL の枝刈り効率を示す．

図 6–5 に示すように，EA-FIL は，$|\Phi|$ が大きく，W が小さいときに枝刈り効率が良く，GP はその逆である．また，図に示した 10 条件の内，7 条件で EA-FIL のほうが枝刈り効率が良いことが分かるが，枝刈り効率が GP のほうが良い実験条件においても EA-FIL のほうが高速に動いている．これは，EA-FIL が $O(1)$ 更新を行っているためである．特に W が大きいときに GP の枝刈り効率が大きくなるが $O(1)$ 更新の影響も顕著に表れるため EA-FIL のほうが高速になると考えられる．



(1) ラベル種類数 $|\Phi|$ を変化　　(2) ウインドウサイズ W を変化

図 6–5: 人工データに対する top-k 類似検索の枝刈り効率

6.1.4　空間計算量の評価実験

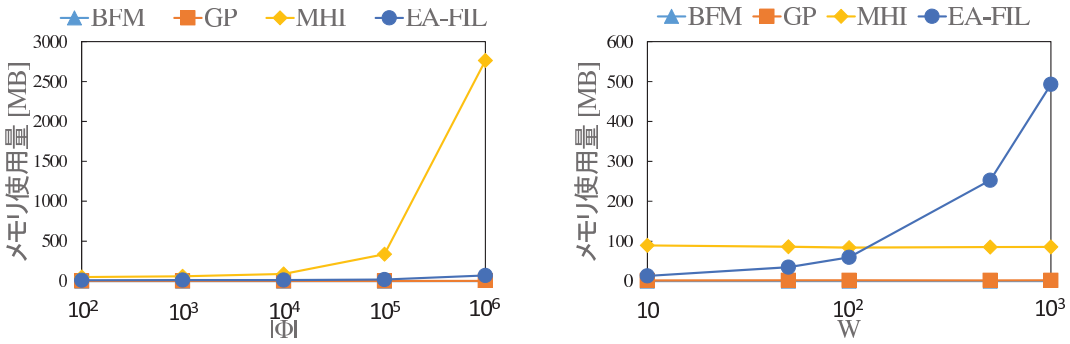
提案アルゴリズムの空間計算量について検索実験中にどれだけのメモリを使用しているか測ることで評価した．なお，このメモリ使用量は，各プログラム実行時の総メモリ使用量から，データベースおよび，クエリのメモリ使用量を引いたものとなっている．詳しい実験方法を以下に示す．

各アルゴリズムに対して，時刻 $T = 1 \sim 10$ まで検索実験を行うプログラムを実行し，getrusage() 関数を使って，プログラム実行時のピークメモリ使用量を測定した．さらに，データベース，クエリを読み込む処理だけを実行するプログラムを走らせて getrusage() 関数でピークメモリ使用量を測定し，各アルゴリズムのメモリ使用量をその分だけ引いた．

図 6–6 に IBM Quest data generator で生成したデータに対する検索実験における各プログラムのメモリ使用量を示す．

図 6–6(1)，(2) に示すように EA-FIL は，GP と比べてメモリを多く使用している．これは，EA-FIL が転置インデックスやクエリのヒストグラムを必要としているのに対し，GP は 3.1 節で述べた min_step と上限値のみを保存しておけばよいためためである．

図 6–6(1) に示すように EA-FIL は，すべての $|\Phi|$ に対して，MHI よりもメモリ使用量が少ないことがわかる．ただし，図 6–6(2) より，ウインドウサイズ W 大きいとメモリ使用量が大きくなってしまう．本実験の環境では，W が増えるときに，データベースの内の集合サイズも比例して増加させている．そのため，EA-FIL の転置リストの登録数が，データベースの総要素数と同じくらいであるためメモリ



(1) ラベル種類数 $|\Phi|$ を変化　　(2) ウインドウサイズ W を変化

図 6–6: 人工データに対する top-k 類似検索の空間計算量

使用量が増加したのに対して，MHI では各データの要素数が転置リストに影響しない．以上より，W が 100 以下であれば，メモリ使用量の点でも EA-FIL が MHI よりも優れていることがわかる．

6.2　拡張した CSPEQ 問題

人工データと実データを使用し，VWEA-FIL の性能を実験的に評価する．

6.2.1　使用したデータ

IBM Quest data generator によって作成した人工データと 6.1.2 節の実データを使用した．人工データと実データによるデータベースとストリームデータの作成方法は 6.1.3 節の実験と同様である．

本節の実験は，前節の実験と全く同じデータを使用しているが，パラメータ W は，本節においてはデータベース内の集合の大きさの平均値を表すものとする．

本実験におけるウインドウサイズの最小 $W_{\min}$，最大 $W_{\max}$ は，$W_{\min} = 0.5W$，$W_{\max} = 1.5W$ とした．

6.2.2　時間計算量の評価実験

各時刻 T において，クエリ Q_T と類似度が高い k 個の集合を検索する検索実験を実施した．時刻を $T = 1$ から，$T = 1000$ まで進め，1000 回の top-k 類似検索に

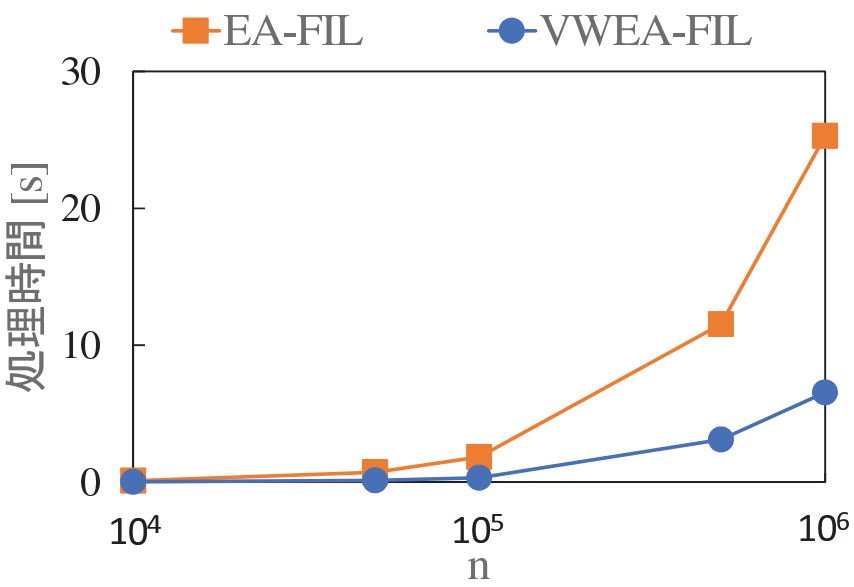


図 6–7: データ数 n を変化させたときの拡張した問題設定に対する各アルゴリズムの実行時間

かかる合計の処理時間を測定し，以下の 2 つの手法の性能を比較した．

- EA-FIL:単純な EA-FIL の適用
- VWEA-FIL:5.4 節で提案した手法

(1) 集合数 n に対する処理時間の変化

$k = 10$，$W = 10, |\Phi| = 10000$ の条件で集合数 $n = 10000, 50000, 100000, 500000, 1000000$ と変化させた時の実行時間を図 6–7 に示す．

図 6–7 より，VWEA-FIL が EA-FIL と比べると実行速度が数倍速くなっており，有効性が示されている．しかし，5.4 節で述べたように，VWEA-FIL は EA-FIL から処理対象を約半分にしたものであるが，例えば $n = 100000$ のとき VWEA-FIL は 0.31 秒であったのに対し，EA-FIL は 1.83 秒でおよそ 6 倍の時間がかかっている．これは，アルゴリズム上の問題ではなく，メモリアクセスに時間がかかっている影響であると考えられる．

(2) 要素の種類数 $|\Phi|$ に対する処理時間の変化

次に，$k = 10$，$W = 10, n = 100000$ の条件で，パラメータ $|\Phi| = 100, 1000, 10000, 100000, 1000000$ に変化させた時の実行時間を図 6–8 に示す．

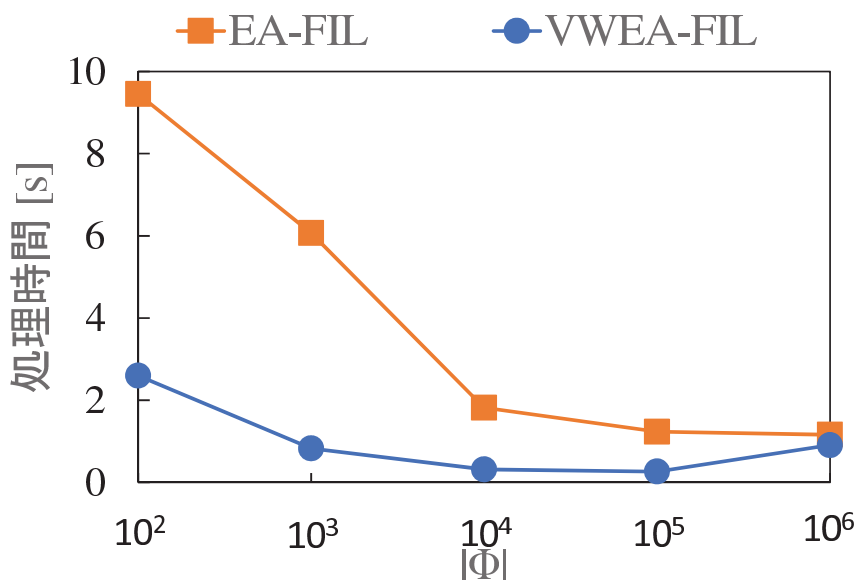


図 6–8: ラベル種類数 $|\Phi|$ を変化させたときの拡張した問題設定に対する各アルゴリズムの実行時間

図 6–8 に示すように，全ての $|\Phi|$ において，VWEA-FIL が EA-FIL よりも高速であった．しかし，$|\Phi| = 10^6$ のとき，VWEA-FIL の処理時間が $|\Phi| = 10^5$ より増加してしまっている．Jaccard 係数計算回数を測定した結果，$|\Phi| = 10^5$ のとき，約 101 万回，$|\Phi| = 10^6$ のとき，約 80 万回 Jaccard 係数の計算を行っており計算回数自体は減少しているためこれも，メモリアクセスに時間がかかっている影響であると考えられる．

(3) 平均集合サイズ W に対する処理時間の変化

$k = 10$，$|\Phi| = 10000, n = 100000$ の条件で，パラメータ $W = 10, 50, 100, 500, 1000$ に変化させた時の実行時間を図 6–9 に示す．

図 6–9 より，この実験でも，VWEA-FIL が EA-FIL を上回る性能であることが分かった．

(4) 実データ

最後に，Market Basket dataset と Click Stream data set の二つの実データに対して，top-10 類似検索を行った時の実行時間を表 6–1 に示す．

人工データに対する実験結果と同様に，実データを用いても VWEA-FIL は EA-

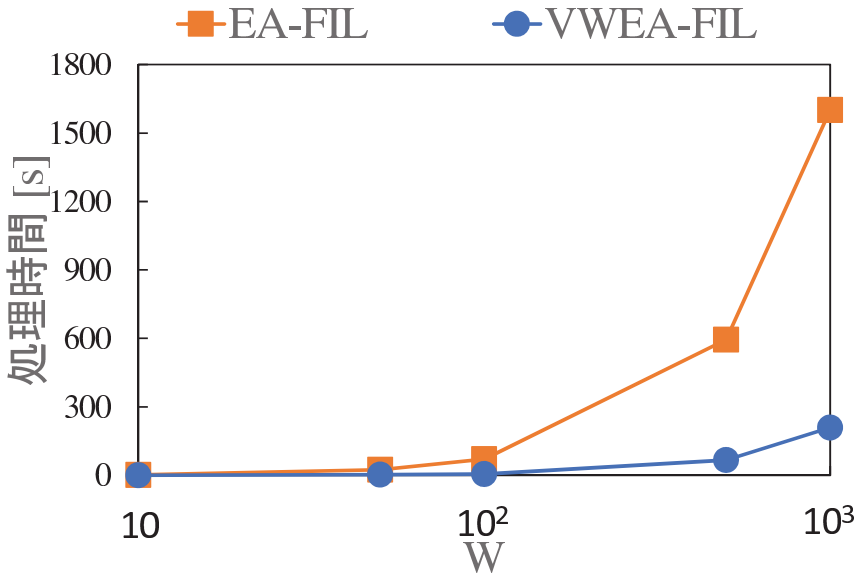


図 6–9: 平均集合サイズ W を変化させたときの拡張した問題設定に対する各アルゴリズムの実行時間

表 6–1: 実データに対する top-10 類似検索の実行時間

手法	Market Basket dataset	Click Stream data set
EA-FIL	3.42[s]	5.15[s]
VWEA-FIL	0.91[s]	1.14[s]

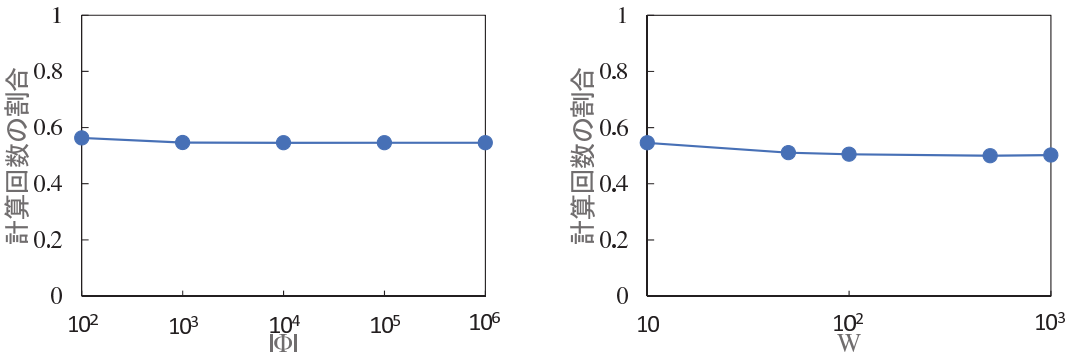
FIL より高速であることが示された.

(5)Jaccard 係数の計算回数

5.4 節で述べたように，VWEA-FIL の Jaccard 係数の計算回数は，EA-FIL のおよそ半分となるはずである．これを実験的に示す.

検索実験中に EA-FIL，VWEA-FIL の Jaccard 係数計算回数をそれぞれ測定した．図 6–10 に，EA-FIL に対する VWEA-FIL の Jaccard 係数計算回数の割合を示す.

図 6–10 に示すように，VWEA-FIL の Jaccard 係数の計算回数は EA-FIL のおよそ半分になっている．この結果から実験的にも VWEA-FIL が EA-FIL の半分の計算量となることが示された.



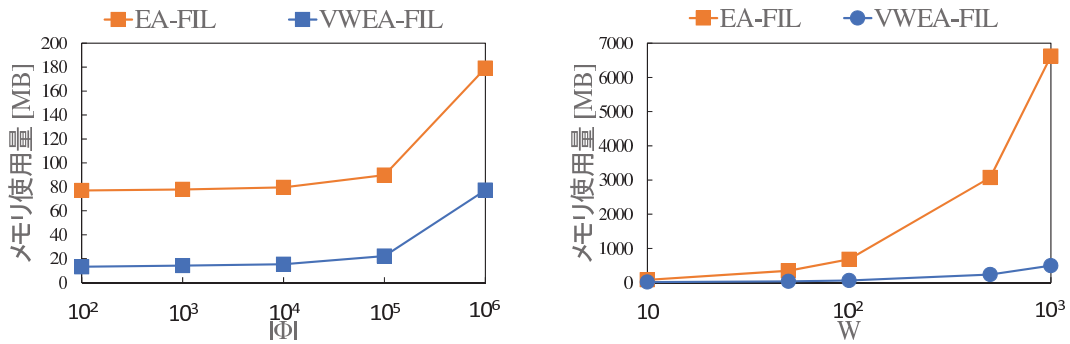
(1) ラベル種類数 $|\Phi|$ を変化　　(2) 平均集合サイズ W を変化

図 6–10: top-k 類似検索における EA-FIL に対する VWEA-FIL の計算回数の割合

6.2.3　空間計算量の評価実験

提案アルゴリズムの空間計算量について検索実験中にどれだけのメモリを使用しているか測ることで評価した．メモリ使用量の測定方法は，6.1.4 節の実験と同様である．

図 6–11 に人工データに対する検索実験における各プログラムのメモリ使用量を示す．



(1) ラベル種類数 $|\Phi|$ を変化　　(2) 平均集合サイズ W を変化

図 6–11: 人工データに対する top-k 類似検索の空間計算量

図 6–11 より，全ての $|\Phi|$ と W において，VWEA-FIL のメモリ使用量が少ないことがわかる．これは，EA-FIL がスライディングウインドウの数だけ Jaccard 係数，和積集合サイズ，クエリのヒストグラムを保存しておく必要があるのに対し，VWEA-FIL はそれらを最小のウインドウサイズのものだけ保存しておけばよいか

らである．これにより，空間計算量においても，VWEA-FIL が EA-FIL を上回る性能であることが示された．

第7章

関連研究

CSPEQ 問題は 2016 年に提唱された新しい問題であり，この問題を扱った先行研究は [1] だけである．本章では，ストリームデータの類似検索に関してまとめる．ストリームデータの類似検索は，時間経過に伴って (1) データベースが変化するタイプ，(2) クエリが変化するタイプ，(3) クエリとデータベース両方が変化するタイプがあり得る．(1) のデータベースが時間経過に伴って変化するタイプがこれまで最も盛んに研究されている．Yang ら [6] はユーザの嗜好性を表すクエリ関数 F に対して，データベースから $F(S)$ の値が最大になるデータを探索する top-k 問題を考えた．データには有効期間が設定されており，top-k 内のデータの有効期間が切れた時にいかに効率よく top-k を更新するかが論点である．Rao ら [7] は複数のクエリ間でデータを評価した結果を共有させることでクエリの処理効率を向上させている．Lian ら [8] は実数値が追加されるデータストリームに対して，W 次元データに対する類似検索を Locality-Sensitive Hashing[2] で実現した．この方法では時間経過と共にハッシュ関数を更新する．

本研究が取り扱う CSPEQ 問題 [1] は (2) のクエリのみが変化する問題である．Kontaki ら [9] は，クエリとデータベースが両方変化する状況での類似検索を考えた．DFT(Discrete Fourier Transform) 係数をストリームの特徴量として使用し，特徴量を効率よく更新する方法を提案した．

Datar ら [10] は本研究と同様に時間と共に変化する集合を取扱い，Min-Hash のハッシュ値を効率的に更新する手法を開発した．この手法は 3.2 節で述べた Min-Hash ベースの近似解法のサブルーチンとして使える技術であるが，具体的な類似検索問題への適用までは議論されていない．また，本研究ではスライディングウィンドウ内のデータを集合として取り扱うが，これを文字列として扱う研究は

古くから多くなされている．近年の研究成果としては，ハミング距離がクエリ文字列と高々k 異なるパターンを発見する k-mismatch problem をストリーミング環境 [11][12] で解いた例がある．

第8章

結論

本論文では，CSPEQ 問題に対する厳密解アルゴリズム EA-FIL とこの問題を拡張した問題とその高速な厳密解法を提案した．

このアルゴリズムは，頻度を考慮した転置インデックスによって，類似度が変化する可能性がある集合のみを処理対象とした．また，類似度の計算を前回の類似度，クエリに追加される要素と除かれる要素から，$O(1)$ で行うことができるようにした．このとき，転置インデックスによる類似度の更新を可能にするため，特定の要素を含まない集合を処理対象から除外するようアルゴリズムを改良した．

そして，従来の厳密解法および，近似解法との比較実験を行った．その結果提案手法は，従来の近似解法と同じくらい高速であり，従来の厳密解法を凌駕する性能であることが確認できた．

本論文では，CSPEQ 問題の欠点を補う新しい問題設定を考案した．この問題設定では，ウインドウサイズの最大値と最小値を決めておき，その中から各集合に対して一番 Jaccard 係数が大きいものを集合とクエリとの類似度とすることで，集合のサイズの差により類似度の上限値が小さくなる問題を緩和した．また，その問題設定に対して単純に EA-FIL を適用するよりも数倍高速な厳密解アルゴリズムを提案した．

参考文献

[1] X. Xu, C. Gao, J. Pei, K. Wang, and A. Al-Barakati, "Continuous similarity search for evolving queries," Knowledge and Information Systems, vol.48(3), pp.649-678, September 2016.

[2] M. Datar, N. Immorlica, P. Indyk and V.S. Mirrokni, "Locality-sensitive Hashing Scheme Based on P-stable Distributions", Proc. Twentieth Symposium on Computational Geometry, pp.253-262, 2004.

[3] A. Z. Broder, M. Charikar, A. M. Frieze and M. Mitzenmacher, "Min-Wise Independent Permutations", Journal of Computer and System Sciences, vol.60(3), pp.630-659, 2000

[4] http://fimi.ua.ac.be/data/

[5] http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php

[6] D. Yang, A. Shastri, E. A. Rundensteiner, and M. O. Ward,"An Optimal Strategy for Monitoring Top-k Queries in Streaming Windows",Proc. 14th International Conference on Extending Database Technology, pp.57-68, 2011.

[7] W Rao, L Chen, S Chen and S Tarkoma, "Evaluating continuous top-k queries over document streams" World Wide Web vol.17(1), pp.59-83, 2012.

[8] X. Lian,L. Chen and B. Wang, "Approximate similarity search over multiple stream time series", Proc. 12th international conference on database systems for advanced applications(DASFAA'07), pp.962-968, 2007.

[9] M. Kontaki,A. N. Papadopoulos and Y. Manolopoulos,"Adaptive similarity search in streaming time series with sliding windows", Data & Knowledge Engineering, vol.63(2), pp.478-502, 2007.

[10] M. Date and, S. Muthukrishnan "Estimating rarity and similarity over data stream windows", Proc. 10th European Symposium on Algorithms(ESA'02), pp.323-335, 2002.

[11] R. Clifford and B. Sach. ”Pseudo-realtime pattern matching: Closing the gap”, Proc. 21st Symp. on Combinatorial Pattern Matching, pp.101-111, 2010.

[12] R. Clifford, and A. Fontaine, E. Porat,B. Sach, and T. Starikovskaya, ”The k-mismatch problem revisited”, Proc. Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pp.2039-2052, 2016.

謝辞

本研究を行うにあたり，研究の場と適切なご指導，ご助言を頂いた古賀久志准教授，南泰浩教授に深く感謝いたします．日頃から研究に関する活発なご意見，ご助言を頂いた戸田貴久助教授と中鹿亘助教授に深く感謝いたします．多忙の中，多くのご助言，ご指導下さった柳生智彦客員准教授と鈴木一哉客員准教授に深く感謝いたします．また，研究室での生活や研究の様々な場面でご助言を頂きました南 · 古賀 · 戸田 · 中鹿研究室の学生の皆さま，すでにご卒業された先輩方に感謝いたします．

平成 30 年 3 月 7 日

図一覧

表一覧