D_045

大規模文字列ソートのための適応的なデータ分割アルゴリズム
An Adaptive Algorithm for Dividing Large Sets of Strings to Efficiently Sort

浅井達哉[1] 岡本青史[1] 有村博紀[2]
Tatsuya Asai Seishi Okamoto Hiroki Arimura

1 はじめに

1990年代後半から，XMLデータに代表される半構造データ [1] が，急速にネットワーク上を流通するようになってきた．半構造データは，異なるシステム間のデータ交換フォーマットとして使われることが多かったが，最近は金融・流通・医療などの様々な分野で，半構造データをそのまま蓄積し，活用しようとする動きが広がっている．

文字列ソートは，半構造データ処理のためのもっとも基本的な技術の1つであり，半構造データの集計や結合（Join）など，多彩な応用をもつ．ところが，蓄積される半構造データは巨大になる一方であるため，文字列ソートアルゴリズムの規模耐性が問題になりつつある．

クイックソートなどの重複走査型のソートアルゴリズムは，データが大規模になると，計算性能の劣化を引き起こす．最近，トライ [3] を用いた高速なソートアルゴリズム [5] が提案されているが，このアルゴリズムは計算領域を多く使用するため，やはり大規模データへに直接適用することは困難である．

本稿の構成は以下のとおりである．2節では，本稿の議論に必要な準備を行う．3節では，大規模文字列ソートのための適応的なデータ分割アルゴリズムについて述べる．4節では，実験により提案手法を評価する．最後に，5節で本稿をまとめる．

2 準備

本稿では，文字列のソート問題を考察する．定義を準備する．集合 A の要素数を $|A|$ と書く．非負整数 $n \geq 0$ に対して，$[n] = \{1, \ldots, n\}$ と定義する．$\Sigma = \{a, b, \ldots\}$ を文字アルファベットとし，Σ 上の全順序を $\leq_\Sigma$ を仮定する．文字列 $a_1, \ldots, a_m \in \Sigma$ に対して，i 番目の文字を $S[i] = a_i$，その長さを $|S| = n$ と定義する．二項関係 $\leq_\Sigma$ によって定まる Σ^* 上の**辞書式順序**を $\leq_{\mathrm{leq}}$ で表す．

本稿では，**文字列リスト** $S = (w_1, \ldots, w_n)$ が基本的なオブジェクトである．ここに，$n \geq 0$ であり各 $w_i \in \Sigma^*$ $(1 \leq i \leq n)$ は文字列である．**文字列リスト全体の集合**を $SEQ(\Sigma^*)$ で表す．$\|S\| = \sum_{i=1}^{n} |w|$ で S の文字列の**総サイズ**を表す．$S \cdot T$ で二つの文字列リスト S と T の連結を表す．文字列リスト $(w_1, \ldots, w_m)$ と文字列の多重集合 $\{w_1, \ldots, w_m\}$ を同一視する．$S \subseteq T$ で，S が T の部分系列であることを表す．$S = T_1 \cup T_2$ で，リスト S が多重集合としてリスト T_1 と T_2 の和になっていることを表す．

[1]株式会社富士通研究所 ナレッジ研究センター
[2]北海道大学 大学院情報科学研究科

定義 1 **文字列整列問題**とは，入力として長さ n の文字列リスト $S = (w_1, \ldots, w_n) \in SEQ(\Sigma^*)$ を受け取り，整列済の文字列リスト $T = (w_{f(1)}, \ldots, w_{f(n)})$ を計算する問題である．ここに，置換 $f : [n] \to [n]$ は添え字の並べ替えであり，$w_{f(1)} < \cdots < w_{f(n)}$ が成立すると仮定する．

リスト S の文字列数が $K = |S|$ で総サイズが $N = \sum_{w \in S} |w|$ のとき，文字列整列問題は主記憶上で $O(K \log K + N)$ 時間で解くことができる [2]．

ここで，外部整列の計算モデルを導入する．正整数を $M \geq B \geq 0$ とする．**ディスクアクセス機械** (DAM) は，CPUからランダムアクセス可能なサイズ M の**主記憶**と，サイズ B のバケットに分割され，バケット毎に入出力可能な任意に大きな容量の**外部ディスク** $D = B_1 \cdots B_n$ $(n \geq 0)$ をもつ計算機である．

2.1 分配型外部文字列整列

長さ n の文字列リストを $S = (w_1, \ldots, w_n)$ とおく．任意の部分リスト $B \subseteq S$ を S の**バケット**という．バケット $B, C \subseteq S$ に対して，二項関係 $\prec$ を $B \prec C \iff \forall b \in B, \forall c \in C, b \leq_{\mathrm{lex}} c$ と定義し，B は C より**辞書順で小さい**という．正整数を K とする．S のバケットの列 $\mathbf{B}^{(K)} = (B_1, \ldots, B_K)$ が S の**整列分割** (quantile partition) であるとは，多重集合として $S = B_1 \cup \cdots \cup B_K$ が成立し，順序制約 $B_1 \prec \cdots \prec B_K$ を満たすことをいう．通常の K 分位分割と異なり，各バケット $B_i \subseteq S$ の大きさは異なっていても良い．

アルゴリズム: DistStrSort
入力: 文字列リスト S, 正整数 $M \geq B \geq 0$.
出力: 整列済の文字列リスト T.

1. 正整数 $K = n/B$ に対して，$B_1 \prec \cdots \prec B_K$ となる S の整列分割 $S = B_1 \cup \cdots \cup B_K$ で，最大バケットサイズ B 以下のものを求める．
2. 各 $i = 1, \ldots, K$ に対して，バケット B_i を整列して，整列済みバケット C を得る．
3. 全バケットを順に出力ながら，外部ディスク上で連結し，整列済み文字列リスト $T = C_1 \cup \cdots \cup C_K$ を出力する．

図 1: 分配型文字列整列アルゴリズムの主手続き

図1に整列分割を用いた分配型の文字列整列アルゴリズムを与える．アルゴリズムの正当性と計算量を解析する．S の整列分割を $S = B_1 \cup \cdots \cup B_K$ とおく．この最大バケットサイズを $B = \max_{i=1}^{K} |B_i|$ と定義する．次の補助問題を考えよう．

定義 2 **バケットサイズ限定整列分割問題**とは，入力として長さ n の文字列リスト $S = (w_1, \ldots, w_n) \in SEQ(\Sigma^*)$

と，正整数 $B \geq n/K$ を受け取り，最大バケットサイズが B を超えないような S の整列分割 $S = B_1 \cup \cdots \cup B_K$ を求める問題である．

補題 1 $M \geq B \geq 0$ と $K \geq n/B$ を仮定する．S のバケットサイズ限定整列分割問題とサイズ B のバケットに対する文字列整列問題を主記憶サイズ M で解く副手続き $\mathcal{A}_{\mathrm{QP}}$ と $\mathcal{A}_{\mathrm{ST}}$ があると仮定する．このとき，長さ n の文字列リスト $S = (w_1, \ldots, w_n)$ に対して，図 1 の分配型文字列整列アルゴリズムは，主記憶サイズ M で文字列整列問題をとく．アルゴリズムの計算時間は，

$$T(N) = T_{\mathrm{QP}}(N, B) + (N/B) \cdot T_{\mathrm{ST}}(B) + O(N)$$

であり，入出力量は

$$D(N) = D_{\mathrm{QP}}(N, B) + O(N/B)$$

である．ここに，$N = \|S\|$ は総入力サイズであり，$T_{\mathrm{QP}}(N, B)$ と $T_{\mathrm{ST}}(B)$ （$D_{\mathrm{QP}}(N, B)$ と $D_{\mathrm{ST}}(B)$）は副手続き $\mathcal{A}_{\mathrm{QP}}$ と $\mathcal{A}_{\mathrm{ST}}$ の計算時間（入出力量）である．

Proof. アルゴリズムの正当性は，リスト S の整列分割の定義から示される．計算時間は，バケットサイズ限定整列分割ではすべてのバケットは主記憶に収まることと，副手続きに関する仮定から導かれる． ■

定理 2 入力文字列リスト S に対して，図 2 のアルゴリズム DIVIDE が木のサイズが M 以下で，各頂点 v のカウントが $\frac{1}{k}B$ 以下である要約トライ $\mathcal{T}$ を計算したとする．このとき，アルゴリズム DISTSTRSORT は，$T = O(N)$ 時間と $D = N/B$ 入出力で S の整列済みリストを計算する．

Proof. 本節の補題 3 と次節の補題 7 から示される． ■

図 1 の分配型文字列整列アルゴリズムで，ステップ 2 のバケットの整列には任意の内部文字列整列アルゴリズムを利用できる．しかし，後の節でみるように，分配整列アルゴリズムを，トライ整列アルゴリズムおよびテキスト置換アルゴリズムと組合せることで，テキストデータベースのグループ化演算（GROUPBY）や関数型結合（FJOIN）等の高度なデータ処理が外部記憶上で高速に実現可能となる．

3 データ分割アルゴリズム

前節で示した図 1 の分配型文字列整列アルゴリズムでは，ステップ 1 の分割フェーズで，限られた主記憶サイズのもとで，整列分割問題を解く必要がある．とくに，データ分割においては，各バケットの大きさは上限値 B の制約をみたした上で，性能の均等化の観点から，偏りができるだけ小さいことが望ましい．そこで，本節では近似解を許すことで，効率よい整列分割の計算を可能にするアプローチをとる．

アルゴリズム: DIVIDE
入力: 文字列リスト $S = (w_1, \ldots, w_n)$, 正整数 $M \geq B \geq 0$.
出力: S の最大バケットサイズ制限 m 整列分割 $B_1 \cup \cdots \cup B_m$.
ここで，$m \geq 1$ は実行時に決まる正整数.

1. サイズ B の空バケット $B_1, \ldots, B_i, \ldots$ を用意する．成長しきい値 α を計算する．
2. ディスクから S を読み要約トライ $\mathcal{T}$ を構築する．
   $\mathcal{T} :=$ BUILDDIETTRIE(S, α);
3. 要約トライ $\mathcal{T}$ の葉を文字列の辞書順で巡回し，昇順に B 個ずつバケット $B_1, B_2, \ldots,$ に割り当てる．
4. ディスクから S を読み，各文字列を $\mathcal{T}$ の到達した頂点に対応したバケットに出力する．
   SPLITDIETTRIE$(\mathcal{T}, S)$;

図 2: データ分割アルゴリズム

3.1 素朴な分割アルゴリズム

初めにトライソートを直接使って，分割計算が可能か検討する．入力文字列リストを分割するための直接的な方法は，文字列辞書であるトライ構造 [4] を用いるトライソートである．

この方法では，まず，ディスク上の入力データ S を走査して，すべての文字列を主記憶上のトライ $\mathcal{T}$ に挿入する．挿入が終わったら，トライの根から開始して，すべての頂点 v を深さ優先探索で巡回し，各頂点が表す文字列 $\mathtt{label}(v)$ を $\mathtt{count}(v)$ 回ずつ辞書順でディスクに出力する．辞書順で隣接した文字列は，ちょうどサイズ B ずつ同じバケットにまとめて出力する．次は，よく知られた結果である．文字上値が使えるので，比較に基づく整列に関する下限 $O(n \log n)$ は適用されないことに注意されたい．

補題 3 トライソートは，総サイズ M の入力文字列リスト S に対して，主記憶上で $T = O(M)$ 時間（と $D = M/B$ の入出力で）S の整列済みリストを計算する．

しかし，サイズ $N = \|S\|$ の入力文字列リスト S の整列分割を求めるためには，少なくとも $O(N)$ の主記憶サイズを必要とするので，この方法は，主記憶サイズ M が限定された場合には適用できない．

3.2 適応的要約トライ構築アルゴリズム

図 3 に，要約トライを構築するアルゴリズム BUILDDIETTRIE を示す．要約トライ $\mathcal{T}$ は有限状態機械として実装されており，サイズ $s \geq 0$ の文字集合 $\Sigma = \{c_0, \ldots, c_{s-1}\}$ に対して，各状態 v は遷移先の状態をあらわす s 個のポインタの配列 $(\mathtt{goto}(v, 0), \ldots, \mathtt{goto}(v, s-1))$ として実現する．新しい状態 v が生成されたとき，これらのポインタを $NULL$ で初期化する．また，v は生成時に 1 で初期化される整数カウンタ $\mathtt{count}(v)$ を持つ．

要約トライ $\mathcal{T}$ の構築において重要なのが，成長しきい値と呼ぶ正整数 $\alpha > 0$ である．これは構築において，現在の枝の成長に必要な最小頻度を表すしきい値である．

最初に，トライは根だけからなる空トライ $\mathcal{T} = \{\mathtt{root}\}$ として初期化する．文字列 $w = c_1 \cdots c_\ell$ をトライに挿入する場合は，根か出発して先頭から文字を一つずつ読

アルゴリズム: BUILDDIETTRIE
入力: 文字列リスト $S = (w_1, \ldots, w_n)$, 正整数 $M \geq B \geq 0$, 正整数 $\alpha > 0$
出力: S に対する要約トライ $\mathcal{T}_S$.
1. $\mathcal{T} :=$ {root}; (空語のみを表す根だけのトライ).
2. 各 $i := 1, \ldots, n$ に対して以下を実行する：
   (a) 次の文字列 $w_i = c_1 \cdots c_{\ell(i)}$ をディスクから読む; $j = 0$;
   (b) 以下をくりかえし実行する：
   - $y :=$ goto(x, c_j); （c_j 辺をたどる ）
   - もし $y \neq \perp$ が成立するならば, $x := y$;
   - もし $y = \perp$ ならば, 以下を実行する：
     - count$(x) < \alpha$ ならば何もせず, ステップ 2(a) に行って次の文字列を処理する;
     - count$(x) \geq \alpha$ ならば以下を実行する：新しい頂点 y を生成する; $T := T \cup \{y\}$; count$(y) := 1$; goto$(x, c_j) := y$;
   - $j := j + 1$;
3. $\mathcal{T}$ を返す.

図 3: 要約トライの構築アルゴリズム

みながら, goto ポインタで遷移し, 対応する辺をたどる. 要約トライの各状態 v は整数カウンタ count(v) を持っており, ある状態に到着するとカウンタ値を一つ増やす. カウンタは入力文字列のある接頭辞の近似的な出現回数を示している.

もし現在の状態から遷移先がなければ, 新しく状態を生成しようとする. このとき, 現在の状態のカウンタが閾値 α を超えていない限り, 新しい辺は伸ばさない. 超えていれば遷移先の状態を生成して枝を伸ばす. このようにして, データ中の接頭辞の出現回数に合わせて, 適応的に枝を伸ばしていく.

補題 4 図 3 のアルゴリズム BUILDDIETTRIE は, 長さ n の文字列リスト S に対して, $T = O(N)$ 時間と $D = N/B$ 逐次入出力で動作する.

3.3 バケット分割アルゴリズム

アルゴリズム DIVIDE では, 要約接頭辞トライのそれぞれの葉節点を通るデータ件数を予想し, それに基づいて, バケットサイズができるだけ均等になるよう, 葉節点をバケットに割り当てる.

補題 5 図 4 の手続き SPLITDIETTRIE は, 長さ n の文字列リスト S とそのサイズ B 以下のバケット ID 付き要約トライ $\mathcal{T}$ に対して, $T = O(Kn) = O(N)$ 時間と $D = N/B$ 逐次入出力で S の整列分割を計算する. ここに K は $\mathcal{T}$ の深さである.

補題 6 入力文字列リスト S に対して, 図 2 のアルゴリズム DIVIDE が出力した要約トライを $\mathcal{T}$ とし, 整列分割を $S = B_1 \cup \cdots \cup B_m$ $(m \geq 0)$ とする. もし各頂点 v のカウントが $\forall v \in \mathcal{T}$, $0 \leq$ count$(v) \leq \frac{1}{k}B$ を満たすならば, 任意のバケット B_i $(1 \leq i \leq m)$ は $\frac{k-1}{k}B \leq B_i \leq B$ を満たす.

アルゴリズム: SPLITDIETTRIE
入力: S に対する要約トライ $\mathcal{T}_S$, 文字列リスト $S = (w_1, \ldots, w_n)$, 正整数 $M \geq B \geq 0$.
出力: S の整列分割をディスクに出力する.
1. $v :=$ root; (トライの根).
2. 各 $i := 1, \ldots, n$ に対して以下を実行する：
   (a) 次の文字列 $w_i = c_1 \cdots c_{\ell(i)}$ をディスクから読む; $j = 0$;
   (b) 以下をくりかえし実行する：
   - $y :=$ goto(x, c_j); （c_j 辺をたどる ）
   - もし $y \neq \perp$ が成立するならば, $x := y$;
   - もし $y = \perp$ ならば, 以下を実行する：
     - 現在の文字列 w_i を ID $i =$ bid(x) のバケット B_i に出力する. ステップ 2(a) に行って次の文字列を処理する;
   - $j := j + 1$;

図 4: 要約トライによる分配アルゴリズム

補題 7 より一頂点あたりの文字列数を B に比べて小さくできればバケットの充填率は高くなる. そのためには, メモリが許す限りでできるだけ積極的に頂点を分割した方が良い.

補題 7 入力文字列リスト S に対して, 図 2 のアルゴリズム DIVIDE が構築した要約トライ $\mathcal{T}$ は, 木のサイズが M 以下で, 各頂点 v のカウントは $\forall v \in \mathcal{T}$, $0 \leq$ count$(v) \leq \frac{1}{k}B$ を満たすと仮定する. このとき, アルゴリズム DIVIDE は, $T = O(Kn) = O(N)$ 時間と $D = N/B$ 入出力で S の整列分割を計算する. ここに K は $\mathcal{T}$ の深さである.

4 計算機実験

本節では, 実データを用いた実験について報告する. 我々は提案アルゴリズムを C 言語で実装し, PC（Xeon 3.6GHz, 3.25GB RAM, WindowsXP）上で実験を行なった.

実験に用いたデータは, あるサーチエンジンの検索ログから Cookie 項目を抜き出して得られたテキストデータである. データの性質は以下の通り：

- ファイルサイズ：490MB
- レコード件数：11,445,513 件（重複あり）
- レコード件数：1,092,567 件（重複なし）
- レコード長の平均： 44.9
- レコード長の分散： 19.7
- レコード長の最大値：58
- レコード長の最小値：15

4.1 要約トライの節点数

まず最初に, 成長頻度パラメータを $\alpha = 100, 1000, 10000$ と変化させながら, アルゴリズムが構築する要約トライの節点数を測定した. 結果を表 1 に示す.

表 1: 要約トライの節点数

成長頻度 α	100	1,000	10,000
節点数	426,209	92,768	5,030

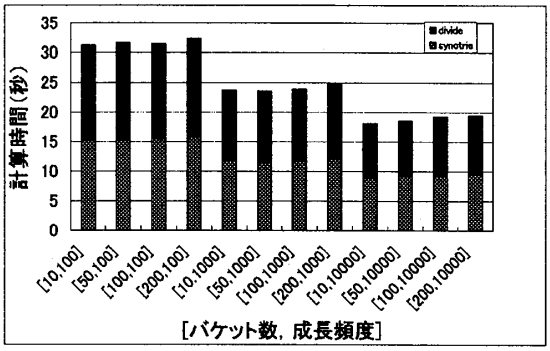

図 5: BUILDDIETTRIE と SPLITDIETTRIE の計算時間

表より，成長頻度を大きくすると，要約トライの大きさが小さくなることが確認できる．次の実験では，要約トライの大きさと計算時間の関係について考察する．

4.2 計算時間

4.2.1 BUILDDIETTRIE と SPLITDIETTRIE

バケット数と成長頻度の 2 つのパラメータを変えながら，アルゴリズム BUILDDIETTRIE とアルゴリズム SPLITDIETTRIE の計算時間を測定した．図 5 に結果を示す．図の横軸は［バケット数，成長頻度］の組であり，縦軸が計算時間である．

結果から，計算時間は，共に成長頻度が大きいほど高速になることが分かる．すなわち，要約トライを小さく構築する方が，BUILDDIETTRIE も SPLITDIETTRIE も計算時間の観点で有利であることが分かった．また，バケット数の値は，計算時間にほとんど影響を与えないことも分かった．

4.2.2 分配型文字列ソートの計算時間

次に，図 1 に示した分配型文字列整列アルゴリズムにおいて，データ分割に DIVIDE を，文字列整列にトライソートを用いた場合の計算時間を測定した．図 6 に結果を示す．図には，比較対象として，GNU の sort コマンドによる整列時間もプロットしている．結果から，我々のアルゴリズムは，データサイズに対して線形時間で動作しており，データサイズが大きくなるほど，GNU-sort に比べて高速に動作することが分かる．

4.3 バケットサイズの分散

最後に，分割されたバケットの大きさの分散を測定した．表 2 に，各パラメータ値における，バケットサイズの標準偏差を示す．この表より，成長頻度が低いほど，すなわち要約トライが大きいほど，分散が小さい値を示しており，より良いデータ分割を達成していることが分かる．

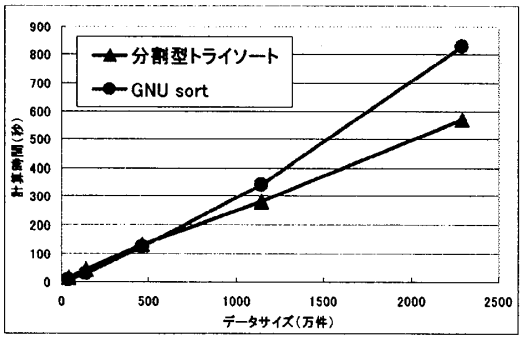

図 6: 分配型文字列ソートの計算時間

表 2: バケットサイズの標準偏差

		バケット数			
		10	50	100	200
成長頻度	100	150.4674	414.065	529.4358	1181.679
	1000	1509.739	2642.023	3701.049	1143.671
	10000	34106.74	15559.69	10068.27	7683.745
	（平均バケットサイズ）	1144551	228910	114455	57228

まとめると，計算時間の高速化とバケットサイズ均一化はトレードオフの関係にあり，目的に応じて成長頻度パラメータの与え方が変わることが分かった．

5 まとめ

本稿では，大規模文字列ソートのためのデータ分割問題を考察し，要約トライを用いた適応的なデータ分割アルゴリズムを提案した．プロトタイプを実装し，簡単な性能実験を通じて，提案アルゴリズムの有効性を示した．

今後の課題としては，数 GB 以上の大規模データを使ったソート実験が挙げられる．他の分割法との比較を行ない，本手法の利点を明らかにしたい．また，グループ化演算や関数型結合など，より高度なデータ処理への適用も今後の課題である．

参考文献

[1] S. Abiteboul, P. Buneman, D. Suciu, *Data on the Web*, Morgan Kaufmann, 2000.

[2] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, On Sorting Strings in External Memory, Proc. the 29th Annual ACM Symposium on Theory of Computing (STOC'97), 540–548, 1997.

[3] Aho, A. V., Hopcroft, J. E., Ullman, J. D., *Data Structures and Algorithms*, Addison-Wesley, 1983.

[4] E. Fredkin, Trie memory, CACM, 3(9), 490–499, 1960.

[5] R. Sinha, J. Zobel, Efficient Trie-Based Sorting of Large Sets of Strings, Proc. of *ACSC'03*, 2003.