データベースシステム 114－2
（1998. 1. 19）

グラムベース全文検索システムの高速化と応用

佐藤 隆士 杉山 良輔 三木田 哲也 遠藤 央 須山 剛 野田 十悟

大阪教育大学

大阪府柏原市旭ケ丘4-698-1

あらまし　本稿では，二次記憶に格納された大容量テキスト中から任意の文字列を高速に検索するため，グラム (一定長の連続する文字列) に基づいたインデックスと検索アルゴリズムを説明し，その応用について述べる．基本的アイデアは，一定長 (L) の全グラムについて，その文字列が出現するテキスト中の位置を検索部としてコンパクトなデータ構造に記憶する点である．この長さ L 用の検索部を用いて，キー長 L の文字列のみならず，L 未満および L を超える文字列についても，効率的に文字列検索できる．テキスト中の最初のパターンを見つけるために要する二次記憶へのアクセス回数は，キー長 l_k が L 以下のとき2回，そうでない場合は $2(l_k - L + 1)$ 回である．また，全マッチパターンを見つけるまでの時間は，マッチ数のみに比例する最小限の増加率になっている．

Gram Based Full Text Search System and Its Applications

Takashi Sato　Ryosuke Sugiyama　Tetsuya Mikita
Akira Endoh　Tsuyoshi Suyama　Jugo Noda

Department of Atrs and Sciences
Osaka Kyoiku University

4-698-1 Asahigaoka, Kashiwara, 582 Japan

Abstract　In this paper, the authors propose a new gram based index in order to make a fast search for an arbitrary string in a large full text stored in secondary storage. The index stores the location of every string of length L in the text. Using this, we can efficiently search for, not only strings of length L but also those shorter than or longer than L. From an analysis of search cost, the number of accesses to secondary storage in order to find the first match to a key is two when the key length l_k is shorter than or equal to L, and $2(l_k - L + 1)$ otherwise. And the time required to find all matching patterns is proportional to the number of matches, which is the lowest rate of increase for these kind of searches.

1 はじめに

近年，要求される情報処理内容の多様化により，電子的に格納された記事，文書，文献，書誌データ，マニュアル，ニュース，辞書などのオンライン情報から，予め設定されたキーワードに束縛されない検索が必要とされている．文書のオンライン化が進んでいる現在，このようなフリーワードによる全文検索の必要性が増している．大容量テキストに対しては，直接テキストをサーチする方法[1, 2]は効率的でない．このため多様な補助データ構造を使用し，高速化する方法が提案されている．近年，グラムと呼ばれる，テキスト中から切り出された部分文字列に基づくインデックスを用いた検索の研究が活発である[3, 4, 5, 6]．本稿では，テキスト中から任意の文字列を特に高速に検索するためグラムに基づく新しいインデックス構造と，それを用いた検索アルゴリズムを提案する．

日本語を対象とした全文検索システムで1ないし2グラムの転置ファイルを利用したシステムの稼働例が上げられている．日本語の場合，文字数が多いため検索部が大きくなる問題が指摘されており[7]，文字種に応じた複数の検索構造を作成する工夫がなされている[3, 5, 6, 8]．各グラムについて，それを含む文書番号のみならず，フォルスドロップを抑えるため，文書内での位置情報をもつことが多い[4, 5]．また，出現頻度の大きい一部のグラムについては3文字以上のインデックスも含める例がある[3, 4]．

従来の研究に比べ，本稿で提案の方法は，比較的長いグラム長を(英文テキスト6ないし12; 和文テキスト4ないし6程度)を用いている点に特徴を有する．これにより，位置情報がなくとも実用上フォルスドロップの発生を防ぐことができる．この結果，インデックス容量が削減されたばかりでなく，同一グラム値に含まれる文書番号情報が減少し，CPU処理コストを押さえることができた．従来の手法ではグラム長以下の文字列の検索はできないため，部分的に3以上のグラムを用いていても，短い文字列を検索するためテキスト中のすべての1および2グラムのインデックスを切り出す必要があった．本稿での方法では，長いグラムのインデックスのみでグラム長未満の短い文字列も高速に検索可能としている．

また，本稿で提案するインデックス作成手法は，対象テキストの言語上の性質に依存しないため，広範囲な言語，ビット列や遺伝子情報などのあらゆるストリングマッチに適用可能である．

理論的解析によると，テキスト中の最初のパターンを見つけるために要する二次記憶へのアクセス回数は，キー長l_kがグラム長L以下のとき2回，そうでない場合は$2(l_k - L + 1)$回である．また，全マッチパターン検出のために要する時間は，マッチ数のみに比例する最小限の増加率になっている．更に，(1) データ構造を木構造にする，(2) データ間の差分を記憶する，ことによりインデックスをコンパクト化する方法を説明する．

アクセス回数によるコストに現れないCPUコストについても，マシンワード幅を生かしたグラムコーディングによるグラム値比較の高速化，不必要なデコーディングを押さえるデータ構造の採用，デコーディング容易な圧縮方法の採用などにより，ファーストマッチ50msecを達成しており，大容量テキストを対象とした，実用上重要な比較的短い任意文字列の検索方法としてはきわめて高速である．

ワークステーションによるネットワークニュース(100,300,500MB)を被検索テキストとした実験例では，検索時間はマッチ数が多く，しかもキー長がグラムに比べかなり長い場合でも数100msec以下，それ以外の場合は数10から200msec程度であった．また，テキストサイズに対する記憶オーバーヘッドは，グラム長が$L = 4$で47%，$L = 12$で212%であり，実用的なサイズに圧縮可能なことが確認された．

従来，グラムベースの検索構造の作成コストに関してはあまり議論されることがなかったが，大量データを対象とした索引構造の作成のためには膨大が時間と記憶容量を必要としていたものと思われる．本稿の後半では，インデックス作成の高速化，更新処理および多様な検索と文書管理への応用について述べる．

2 諸定義

記号集合Σは，一般的にはテキスト中に現れる全ての文字からなる集合を言うが，本稿では検索部データ構造作成のため文字写像を行った後の文字集合を言うことにする．記号集合の大きさは$\sigma(= |\Sigma|)$で表す．本稿で提案のデータ構造は，被検索データとなるテキスト中に含まれる一定長の文字列（グラムといいう）についてその全ての出現場所を記憶するものである．この一定長のことをグラム長(L)と言う．検索すべき任意文字列をキー(k)と呼び，長さをl_kとする．キーを構成する文字は，$k = c_1 c_2 \cdots c_{l_k}$ $(c_i \in \Sigma; i = 1, 2, \cdots, l_k)$で表される．

テキスト長はn[文字]で表す．テキストは主記憶サイズに比べ大容量で，二次記憶に格納されている場合を対象としており，主記憶とはサイズB[語]のブロック転送を行うものとする．

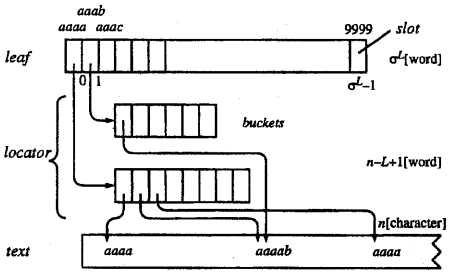


図 1: インデックス構造：基本形 ($L=4$)

記憶の単位は，語および文字とする．典型例は，1 語=4 または 8Byte，1 文字=1 または 2Byte である．

3 インデックス構造と記憶コスト

3.1 基本データ構造

提案のインデックスは，葉部 (leaf) と位置付部 (locator) からなる．長さ L の全ての文字列の組合せ数はσ^Lあることから，辞書順にσ進 L 桁 $(0,1,\cdots,\sigma^L-1)$ のアドレスをもつ葉部を作成する（図1参照）．葉部の各要素をスロットと呼ぶ．スロットには位置付部のバケットへのポインタが入っている．更にバケットには，テキスト中の場所を指すポインタ（複数個）が入っている．あるスロットが指すバケット内には，そのスロットに対応する文字列がテキスト中に現れる場所を記憶している．

ふつうB木の葉にはキーのみならずポインタも格納する．同様に提案のインデックスでもB木のキーをスロット値に対応すると，位置付部に格納しているポインタを葉部に埋め込むことは可能である．敢えて分離したのは，位置情報取出しの際，不必要なスロット値の読込みを防ぐためである．インデックスを圧縮した場合は，不必要なスロット値のデコーディングも防止されることになる．

3.2 記憶コスト

葉部は1語/スロットとしてσ^L[語] となる．長さ L のグラムはテキスト中に $n-L+1$ あるので，ポインタを1語と仮定すると位置付部は $n-L+1 \simeq n$[語] となる．両方合わせて

$$\sigma^L + n \text{ [語]} \tag{1}$$

となる．使用されているすなわち *null* でないスロットが疎な場合は，使用されるスロットのみ集めることができる．この際，各スロットはスロット値も含まなければならないので2語になる．この場合，スロット数は n を超えないので葉部の大きさは $2n$[語] 以下となる．以上を考慮すると，記憶コストは，

$$\min\{2n, \sigma^L\} + n \text{ [語]} \tag{2}$$

となる．

もし，記憶するために情報を語の境界に合わせる必要がなければ，すべての bit に情報を記憶できる．このとき，スロット値は $\lceil L\log_2\sigma\rceil$ [bit] で，ポインターは $\lceil\log_2 n\rceil$ [bit] で表現できるので，上式は bit 数で，

$$\min\{n(\lceil L\log_2\sigma\rceil + \lceil\log_2 n\rceil), \sigma^L\lceil\log_2 n\rceil\} + n\lceil\log_2 n\rceil \ [bit] \tag{3}$$

となる．

4 検索アルゴリズムと検索コスト

本節では，**3.** に提案したインデックスを用いた具体的なキー検索アルゴリズムを示し，ブロック転送回数を見積もり検索コストとする．

4.1 検索アルゴリズム

アルゴリズムはキー長 (l_k) とグラム長 (L) の大小関係により3つの場合に分けて説明する．

4.1.1 $l_k = L$ のとき

(1) キーに対応するスロットを，

$$s = \sum_{i=1}^{L} ord(c_i)\sigma^{L-i} \tag{4}$$

から計算する．但し，ord は各文字に対して一意に $0,1,\cdots,\sigma-1$ の数を対応させる任意関数である．

(2) 葉部のスロット s にあるポインタをたどり，テキスト中の k と同じ文字列へのポインタが入っている位置付部のバケットを知る．

(3) バケット内を順にたどり，テキストでのキーの出現場所をリストする．

4.1.2 $l_k < L$ のとき

(1) 連続する複数個のスロットが該当するので，それらのスロットの上下限 (s_2 および s_1) を

$$s_1 = \sum_{i=1}^{l_k} ord(c_i)\sigma^{L-i} \tag{5}$$

$$s_2 = s_1 + \sigma^{L-l_k} - 1 \tag{6}$$

で求める．

(2) s_1からs_2間のスロット中のポインタが指すs_2-s_1+1個の位置付部のバケットを知る．

(3) バケット内を順にたどり，テキストでのキーkの出現場所を知る．

4.1.3 $l_k > L$のとき

(1) 次式からl_k-L+1個のスロットを求める．

$$s_j = \sum_{i=1}^{L} ord(c_{i+j})\sigma^{L-i}. \qquad (7)$$

但し，$j=0,\cdots,l_k-L$.

(2) $s_j(j=0,\cdots,l_k-L)$のスロットにあるポインタが指す位置付部のバケットを知る．

(3) バケット内を順にたどりテキスト中のキー出現場所の候補を知る．この候補は，得られたバケット内にあるオフセット値から$-j$したものである．

(4) 全てのjについて共通の候補を真の出現場所とする．

$l_k < L$のときは複数個のスロットが指すバケット内の出現場所の和集合をとり，$l_k > L$のときは出現場所の積集合をとることになる．

4.2 検索コスト

4.1 で説明したアルゴリズムを実行するのに要する時間を見積もる．インデックスは二次記憶上に置くため，検索コストを二次記憶から主記憶へのデータのブロック転送（フェッチ）回数とする．

4.2.1 $l_k = L$のとき

キーから式（4）で計算したスロット値をもつ葉部のスロットの読込みに1フェッチを要する．そのスロットのマッチ数をMとすると，すべてのマッチパターンの位置付部の読込みに$\lceil M/B \rceil$フェッチを要する．合わせて，

$$f^a_{eq} = 1 + \lceil M/B \rceil \qquad (8)$$

となる．一方，最初のパターンを見つけるためには位置付部の最初の1ブロックのみ読めばよいので，フェッチ数は$f^1_{eq}=2$である．

4.2.2 $l_k < L$のとき

連続するs_1からs_2までのスロットおよび，各スロットが指す位置付部のバケットをフェッチする．スロットのみならず，これら連続するスロットに対応するバケットも二次記憶上の隣接する領域に格納することができるので，すべてのマッチパターンの出現場所を知るためのフェッチ数は，

$$f^a_{lt} = \lceil \sigma^{L-l_k}/B \rceil + \lceil \sum_{i=s_1}^{s_2} M_i/B \rceil \qquad (9)$$

となる．ここで，M_iは，スロットiのマッチ数である．一方，最初のマッチパターンはスロットs_1の文字列に含まれるものを$f^1_{lt}=2$で見つけることができる．

4.2.3 $l_k > L$のとき

(l_k-L+1)個のスロットとそれに対応する位置付部をフェッチするが，それらは二次記憶上で必ずしも隣接しないため，すべてのマッチパターンの出現場所を知るためのフェッチ数は，

$$f^a_{gt} = l_k - L + 1 + \sum_{j=1}^{l_k-L+1} \lceil M_j/B \rceil \qquad (10)$$

となる．一方，最初のマッチパターンは$f^1_{gt}=2(l_k-L+1)$で求まる．

以上，l_kとLの大小関係から3つの場合について解析したが，マッチ数が固定ならいずれの場合も検索時間はテキストの大きさnに無関係である．また，$l_k < L$の場合は対象となる葉部のスロットおよび位置付部を順アクセスするが，$l_k > L$の場合はランダムアクセスとなる点に注意する必要がある．

4.3 CPU処理時間の低減

4.2 では，解析を容易にするため，検索コストを二次記憶へのアクセス回数で見積もった．しかし，処理方法によっては，CPUコストも無視できない．従来の日本語検索のためのグラム長は，1ないし2を基本としており短い．このため，グラムあたりの位置情報は多数となる．グラムが短いとキー検索のため調べるべきグラム数も増大する．相乗効果により調べるべき位置情報はかなり多くなる．このため位置情報を得てからの処理に時間を要していた．本稿で提案のインデックスは比較的長いグラムを用るためこの問題を回避できる．

近年コンピュータのワード幅が広くなる傾向にある．アルゴリズム上は現れないが，実際のプログラミングではワード幅を生かしたプログラミングを行い，実質的に文字列比較の並列処理を行っている．更に，圧縮は，もっとも効果のある一段のみとしている．多段に圧縮した方が圧縮率は向上するが，解凍にCPUを消費し検索が遅くなることを確認している．

5 検索部の圧縮

4. までで説明した方法で，高速に任意文字列検索可能となるが，基本データ構造は大記憶容量を必要とし使用困難となることがある．そこで本節では，検索の高速性を損わず，提案のインデックスを圧縮する方法を説明する．

5.1 位置付部の圧縮

5.1.1 ブロック番号の使用

本研究では，SGML などの構造を利用したテキスト検索は対象としていない．しかし，大容量テキスト検索においては，構造をもたないことは稀で，少なくともテキストが複数の論理ブロックからなる構造をもつことが通例である．そこで，テキスト中の文字列の出現場所をポインタ（実際は，テキストを格納するファイルの先頭からの位置（オフセット））で答える代わりに，論理ブロック毎に番号を対応付け，その番号 (文書番号と呼ぶ) を検索結果として出力する方法を採用する．1 つの論理ブロックに位置は，(論理ブロック中の文字数 $-L+1$) 個存在するので，位置の代わりに文書番号を用いると，表現すべき数が大幅に減少する．この方法により，テキスト中における文字列の出現位置を表すポインタが 1 語要する場合でも，文書番号で表現すると半語で済む可能性が高い．更に，同一論理ブロック中に同じ部分文字列がしばしば重複して出現するので，同じ文字列に対して重複して文書番号を記憶しないようにする．

但し，出現場所を文書番号で記憶すると，$l_k > L$ の場合，検索時に，$(l_k - L + 1)$ 個の長さ L の文字列が $(L-1)$ 文字ずつ重なり合いながら隣接していることを確認できない．このため，**4.1.3** の (3) で行った $-j$ の補正は不要になり，検索結果は存在の可能性を答えるのみとなる．この場合，真に検索文字列がその論理ブロックに含まれているかどうかは，テキスト中のその論理ブロックに直接アクセスして確認しなければならなくなる．しかし，L が適当に大きければ（例えば $L \geq 6$），実用上フォルスドロップは無視できることを実験で確認している．

5.1.2 ランレングス法による圧縮

5.1.1の方法により，位置付部の各バケットには，同じ部分文字列を含む文書番号の組が記憶されることになる．この番号を整列し，隣合う番号差が小さい部分はその差分で記憶すると，より少ない容量で記憶することができる．これをランレングス法といい，文献[9]に詳しい．ここでは，圧縮および解凍に要する CPU コストを軽減するため比較的単純な方法を採用した．

例えば，7bit で 128 未満の差分を表現でき，次に続くデータが差分であるかどうかを表すために 1bit 用いると，隣合う番号が 128 未満の場合は，1Byte で表現できる．2Byte で表現する場合に比べ 1/2 に位置付部を圧縮できる．

5.2 葉部の圧縮

5.2.1 木構造による圧縮

3. では，L 文字組全てについてスロットを準備したが，大容量文書においてもスロットの利用率はあまり高くないことが **8.** に示す実験結果から分かっている．この傾向は，L が大きくなるはど顕著になる．我々は比較的大きな L を用いるのがよいと考えているので，使用されていないスロットを圧縮の対象とすべきである．単にスロットを詰め合わせただけでは，スロット値から直接該当するポインタが格納される位置を知ることはできなくなるので，葉部のスロットに高速にアクセスするためには補助手段を要する．まず，各スロットに位置付部へのポインタだけでなく，各スロット値を格納する．スロット値が 1 語で表現できる範囲内で，各スロットの大きさは 2 語に増加するが，未使用スロットに格納されていた *null* ポインタが不要となり，結果的にかなり圧縮効果がある．更に，高速性を保証するため一定スロット数毎のスロット値を格納した根部 (root) を作成し，葉部と合わせて木構造とする．

5.2.2 スロットの圧縮

5.2.1で述べた葉部の構造を取ることにすると，各スロットはスロット値と位置付部のバケットへのポインタから構成される．このうち前者は，**5.1.2**と同様ランレングス法により，隣合うスロット値の差分が 128 未満のデータは 1Byte に圧縮できる．

また，各スロット値に対するバケットが順に連続領域に格納されるため，バケットの大きさが 128Byte 未満の場合，それらを指すポインタもランレングス法で圧縮可能である．

6 インデックス作成コストの軽減と更新処理

6.1 インデックス作成コストの軽減

グラムベースのインデックスを用いた検索は，高速である反面，グラムベースのインデックス作成にかかる時間とスペースコストが大きい．まず，直接的な作成方法を示し，次いで改良法を説明する．

直接的な作成方法は以下のとおりである．

(1) テキストを文書番号 (dn) 順に読み，グラム値 ($slot$) と dn の対を作成する．

(2) (1) の出力を 1 次キー *slot*, 2 次キーを *dn* として昇順にソートする．ソートの最終パスで同一 *slot* と *dn* をもつ対を削除する (ユニーク化).

(3) (2) の出力から，圧縮しながら root,leaf,locator を作成する．

(1) の *slot* と *dn* の対は，文書中のすべての文字を先頭としてつくられる．たとえば英語でグラム長を 12 とし，*slot* を 8Byte でコーディングし，*dn* に 2Byte を割り当てたとする．この場合，文書中の各文字 (1Byte) に対して 10 倍の 10Byte を要したことになる．最終的に圧縮されて作成されるインデックス (root,leaf,locator) は 2 倍程度としても，そこに至るまでの中間状態を格納するため 10 倍程度の記憶領域を要することになる．近年コンピュータに接続されるディスク容量が増大しているとしても，ギガバイトオーダーのオンライン文書の索引付けをする際には，かなり負担になる．その上，中間状態の膨張に比例してアクセスコストも増大する．

そこで，中間状態の膨張を押さえインデックス作成のコストを低減する以下のようなバッチ法を考案し評価している．

(1) 入力テキストを主記憶で作業できる分量 (バッチ) だけ読込み，そのバッチについて主記憶上で，*slot* と *dn* の対を作成，ソート，ユニーク化，圧縮を経て，バッチインデックス (bleaf,blocator) を二次記憶上に出力する．

(2) (1) で作られた全てのバッチインデックスを解凍しながら読み込み，マージおよび再圧縮し目的とするテキスト全体に対するインデックスを構成する．

ディスク上におかれる中間結果は，バッチインデックスのみなので，ユニーク化率を $u(0 < u \leq 1)$，圧縮率を $\alpha(0 < \alpha \leq 1)$ とすると，直接的な方法に比べ，$u\alpha$ 倍で済む．詳細は省略するが，二次記憶へのアクセスに関して，バッチ法のコスト (C_B) 対直接法のコスト (C_D) は，

$$C_B/C_D \simeq 2(\alpha e u + 1)/e(4 + 2u + \alpha u) \qquad (11)$$

となる．但し，e は直接法における膨張倍率である．アクセスコストには現れないが，実際には，直接法にはない，(1) での圧縮と (2) での解凍は CPU コストを要するため，上式とおりに高速化されるわけではないことが確認されている．

6.2 更新処理

索引の更新処理はテキストに更新が行われる度にリアルタイムに行うのではなく，ある一定間隔 (例えば一日一回)，前回の索引更新時以降に行われた更新記録を元にバッチ的に行う．

(1) 文書の追加があった場合
まず追加された文書に文書番号を割り当てる．次に追加文書からグラムの切出しを行い，グラム値と文書番号を対にしてグラム値の昇順に書き出したファイルを作成し，これを現行の索引とマージすることにより更新を行う．

(2) 文書の削除があった場合
削除された文書の番号を記録し，位置付部から削除する．削除された文書にのみ含まれていたグラムが存在する場合は葉部からグラム値を削除する．

7 応用

7.1 多様な検索

提案のグラムに基づくインデックスの高速性を生かし，多様な検索に対応可能である．

(1) 複数パターンによる AND/OR 検索
インデックスを用いて各パターンを含む文書番号集合を求め，それらの間で集合演算を行う．

(2) 正規表現
ワイルドカード文字を含む検索は，それで分けられてできる複数個の文字列で検索し，得られた文書番号集合の AND 処理後，もとテキストにアクセスし確認する．

(3) あいまい検索
検索対象文字列から長音の付加削除などの規則を用いて，あいまいとなる可能性のある複数の検索パターンを作成し，これらの検索で得られた文書番号集合の OR 処理を行う．

(4) 出現頻度 (ランキング)
出現頻度を知るためには，インデックスに各グラムに対してその文書中での出現回数の情報を加えておく必要がある．この情報をもとに文書のランキングを行う．

7.2 分散文書検索への応用

WWW など分散したオンライン文書の検索への応用である．

7.2.1 動的に変化する分散文書への対応

構成要素間で以下のような通信を行いながら時間的に変化する文書を検索対象とする (図 2 参照).

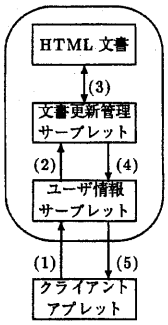


図 2: 更新対応の検索

(1) 検索条件などを送信
- 一般文書:更新された情報量
- 特別文書:指定した文書の監視
- 時間間隔:確認する時間の間隔
- ユーザ情報:文書の閲覧権限など

(2) 更新確認
(1) により指定された時間間隔で通信

(3) 更新情報の監視

(4) 更新情報送信
情報があれば送信

(5) 結果送信
(2) で得られた更新情報を (1) による各条件でフィルタリングした後送信

7.2.2 検索の可視化

検索結果を可視化し，それをヒントに繰り返し分散オンライン文書の検索を行う．可視化にはWWWのページをノード，リンクを枝に例えた一般グラフとしている．リンク構造の描画には，一般グラフの描画方法の1つであるスプリングモデルに基づく手法を使う[10]．この場合，検索にヒットしたページは一箇所に固まりやすい．この集まりを1つのグループとして階層表示を行う．図3に最適化を施した略図を示す．

クライアントの表示部分にはJavaを用いて，ユーザーが見やすいように表示データの加工(グループ化)ができるようにした．ノードを移動し，重なり合ったリンクをほぐしたり，必要なノードの情報を閲覧するなどの対話的操作を可能としている．

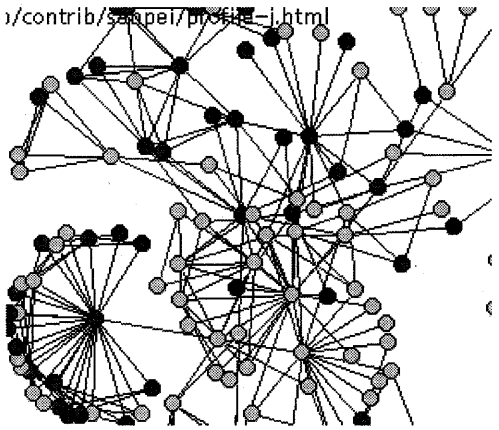


図 3: 最適化をしたサンプル

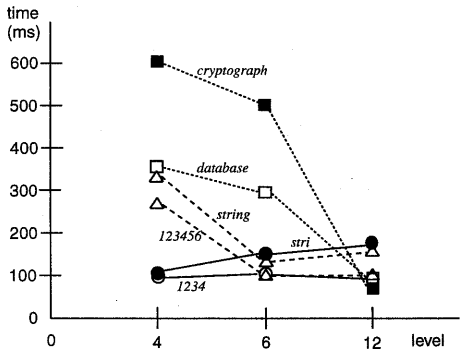
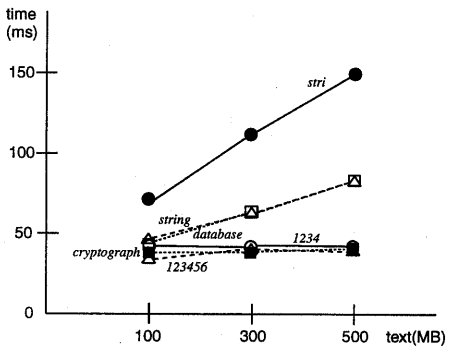


図 4: 実験結果 1, グラム長 – 時間 (100MB Text)

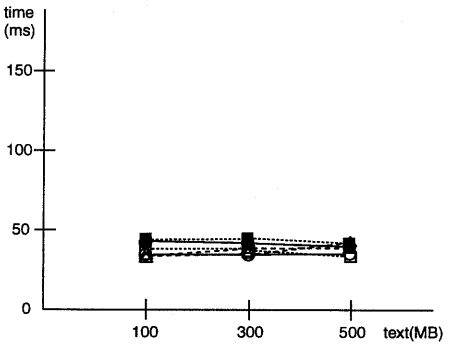
8 実験結果

被検索テキストとして100,300,500MByte分のネットワークニュースを用いた場合の結果を示す．検索時間は，主記憶に根部のみを入れた場合である．即ち，葉部と位置付部は部分的にもキャッシュされない環境で測定した．

まず，100MByte の場合を示す．使用コンピュータは，サンマイクロシステム社 Sparc Server 630(28.5MIPS) である．グラム長 (L) が 4,6,12 について実験した．検索文字列は，低選択度の例として'1234','12345','123456','cryptograph' を，高選択度の例として 'stri','strin','string'.'database' を用いた．出現回数は表1に示す．フォルスドロップは，検索文字列が 'stri' で $L = 4$ の時のみ 17.5%発生したが，他の例では全く生じなかった．テキストに対するインデックスの記憶オーバーヘッドは，$L = 4$



(a) all match



(b) first match

図 5: 実験結果 2, サイズ – 時間 (グラム長: 12)

のとき半分以下 (47.4%), $L = 12$ の場合でも約 2 倍 (212%) であった. 検索時間は図 4 に示すとおりである.

$L = 12$ については, 100MByte のテキストに加え, 300 および 500MByte についても実験した. 使用計算機は, インデックス作成には SGI 社 Power Challenge, 検索時間の測定には, 同じく SGI 社の Indy R4600 を使用した. 測定は, 全マッチ (all match) に要する時間だけでなく, 最初に見つかるマッチ (first match) までの時間についても行った (図 5 参照). 検索時間は, テキストサイズによらずマッチ数にのみ依存することが確認された.

9 おわりに

本稿では, 高速な任意文字列検索を実現するためグラムに基づいたデータ構造を提案し, それを用いた具体的検索アルゴリズムを与え評価した. 提案の手法は, 検索用のデータ構造のみで, 当該文字列を含むテキスト中の位置を知ることができるので, 署名ファイルなど存在の可能性のみを知る手法に比べ, 精度の高い方法と言える. 実験により, 実用上頻繁に行われる比較的短い部分文字列が非常に高速に検索されることを確認した. 記憶容量は, 圧縮により検索の高速性を損なうことなく実用的なサイズにまで減少できた. インデックスの具体的作成および更新方法を説明し処理コストを減少する方法を述べた. 更に, 提案のインデックスを用いた応用にも触れた. 今後の課題は, SGML など構造化文書への対応等である.

表 1: 出現回数

	news (MB)		
	100	300	500
1234	125	516	771
12345	51	136	223
123456	24	87	137
stri	9798	15184	21817
strin	867	3288	4993
string	867	3285	4988
database	1082	4540	6706
cryptograph	9	75	156

参考文献

[1] Boyer, R. S. and Moore, J. S.: A Fast String Searching Algorithm, *Comm. ACM*, Vol. 20, No. 10, pp. 762–772 (1977).

[2] Aho, A. V. and Corasick, M. J.: Efficient String Matching : An Aid to Bibliographic Search, *Comm. ACM*, Vol. 18, No. 6, pp. 333–340 (1975).

[3] Ogawa, Y. and Iwasaki M.: A New Character-based Indexing Method Using Frequency Data for Japanese Documents, *In Proc. of 18th ACM SIGIR Conf.*, pp. 121–129 (1995).

[4] 菅谷他: n-gram 型大規模全文検索方式の開発, 情処 53 全大:5T–2,3 (1996).

[5] 赤峯, 福島: 高速全文検索のためのフレキシブル文字列インバージョン法, アドバンスト・データベース・シンポジウム (1996).

[6] 松井, 難波, 井形:大容量情報全文検索システム, 信学総大:D–4–6 (1997).

[7] 菊地芳秀他: 全文検索の技術動向とシステム事例, 情報処理学会情報学基礎研究報告, 92–FI–25 (1992).

[8] 菊地忠一: 日本語文書用高速全文検索の一手法, 電子情報通信学会論文誌 (D–I), Vol. J75–D–I, No. 9, pp. 836–846 (1992).

[9] Zobel, J., Moffat, A. and Sacks-Davis, R.: An Efficient Indexing Technique for Full-text Database, *Proc. 18th Int. Conf. on VLDB*, 1992, pp. 352–362.

[10] 鎌田 富久: "グラフ描画アルゴリズム", bit, Vol.23, No.3, pp. 284–290 (1991).