RD-002

重複レコードの多い大規模トライ辞書の圧縮

Compression of Large-Scale Trie Dictionaries with Biased Records

矢田 晋 † 森田 和宏 泓田 正雄 青江 順一
Susumu Yata Kazuhiro Morita Masao Fuketa Jun-ichi Aoe

徳島大学工学部 Faculty of Engineering, University of Tokushima
† 日本学術振興会特別研究員 PD

1. はじめに

トライ [1] は文字列の格納に適した木構造であり，形態素解析器 [2] や侵入検知システム（IDS）[3]，組み合わせ素性による分類器 [4] など，幅広い用途を持つ．そして，トライに対する圧縮構造の一つとして，Directed Acyclic Word Graph (DAWG) の有効性が知られている [5]．DAWG はトライの共通部分木を併合して得られるグラフであり，キー集合の格納に用いると，少ないノードで多くのキーを格納することができる．

キー集合としての DAWG に関する従来研究では，動的に更新する手法 [6] やダブル配列による実装 [7] が提案されている．しかし，構築時間や作業領域の制限により，大規模なキー集合への適用は難しいという欠点がある．また，レコードを必要としない用途を想定しているため，応用範囲は極めて限定的である．

従来手法がレコードに対する制約を設けている理由は，キーに対するレコードを一意に割り当てると，トライに共通部分木が存在しなくなるためである．一方で，頻度や重みをレコードとする辞書については，Zipf の法則にあてはまるデータが多いことから，重複レコードが出現しやすいという特徴がある．

そこで，本研究では，重複レコードの多いトライ辞書を DAWG 辞書へと圧縮する効率的な手法を提案する．提案手法では，キーを辞書順に登録していくことにより，トライを介することなく DAWG を構築する．そのため，より大規模な DAWG の構築が可能となる．また，構築時の索引構造にハッシュ表を採用することにより，入力データのサイズに対して線形時間で DAWG を構築できる．

Google により公開されている日本語 n-gram データ [8] を用いた実験では，トライ辞書から DAWG 辞書への圧縮により，同数のノードで 3 倍以上のキーを登録できることが示された．また，4 億のノードで構成される大規模な DAWG を 10–20 分程度の実時間で構築できることが確認された．

以下，2 章でトライ辞書の概要と構築アルゴリズムについて述べ，3 章では，キー集合に対する DAWG の概要と構築アルゴリズムを説明する．そして，4 章で DAWG 辞書の効率的な構築手法を提案し，5 章では，評価実験の設定と結果を示す．最後に，6 章で本研究の成果をまとめる．

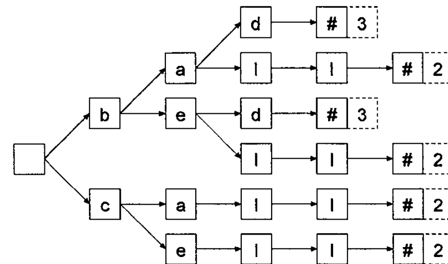

図 1 トライ辞書

2. トライ辞書

2.1 トライ辞書の概要

トライ [1] はキーの構成文字をラベルとする木構造であり，入力文字列に含まれるキーを高速に検出できるという特徴を持つ．また，文字列の終端記号をラベルとして用いることにより，キー同士の識別やレコードの格納を容易に実現できる．そのため，トライは自然言語処理を中心に広く用いられている [6]．

トライ辞書の例を**図 1** に示す．登録されているキーとレコードは，‘*bad*’ = 3, ‘*ball*’ = 2, ‘*bed*’ = 3, ‘*bell*’ = 2, ‘*call*’ = 2, ‘*cell*’ = 2 の 6 組である．図 1 において，‘#’ は終端記号を表しており，直後の数値はレコードを表している．各キーは根から葉への経路上にラベルとして格納されるため，終端記号の存在により，キーと葉は一意に対応することになる．

2.2 トライ辞書の二分木表現

本研究では，大規模なトライ辞書における共通部分木の併合を目的としているため，トライの表現方法に対して，空間効率の高さだけでなく，共通部分木を容易に判定できることが要求される．そのため，一般的に効率的とされているダブル配列 [9] や Ternary Search Tree (TST)[10] ではなく，二分木表現を採用する*.

*二分木表現によるトライ辞書や DAWG 辞書をダブル配列や TST に変換することは可能である [7].

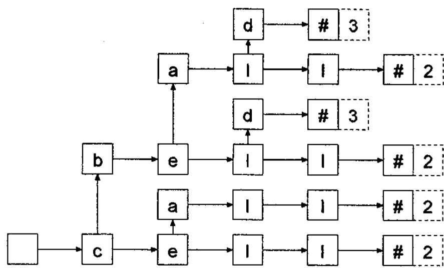

図 2 トライ辞書の二分木表現

トライ辞書の二分木表現では，子ノードへの参照 *child*，兄弟ノードへの参照 *sib*，ラベル *label*，レコード *record* を各ノードのメンバとする．ただし，*child* が無効な参照 nil になる場合にのみ *record* が必要となるため，*child* と *record* には同じ領域を割り当てることができる．そのため，参照とレコードに 4 bytes ずつ，ラベルに 1 byte を割り当てる場合，ノード数 n に対して，必要となる記憶領域は $9n$ bytes である．

図 1 のトライ辞書に対する二分木表現を**図 2** に示す．各ノードにおいて，右方向の矢印と *child*，上方向の矢印と *sib* が対応し，中に含まれる文字は *label* と対応する．ただし，葉については，右方向の矢印が存在せず，隣接する破線枠内の数値が *record* と対応する．

2.3 トライ辞書の構築アルゴリズム

トライ辞書に対するキーの登録は，入力キーの検索をおこない，探索失敗位置から新たな経路を定義することにより実現される．そのため，任意の順序でキーを登録する場合，二分木表現によるトライでは，キーの検索にかかる時間が問題となる．そこで，辞書順にキーを登録するという制約により，トライ辞書の構築における検索時間の問題を解決する．**図 3** は，辞書順にキーを登録するアルゴリズムであり，入力データのサイズに対して線形時間でトライ辞書を構築できる．

図 3 において，Trie::Build はトライ辞書を構築する手続きであり，キーが辞書順になるように，キーとレコードの組 *pair* を Trie::Insert に渡していく．一方，Trie::Insert はキーとレコードをトライ辞書に登録する手続きであり，前半のループでキーを検索し，後半のループで新たな経路を定義した後，レコードを登録する．なお，New は新たなノードを確保する関数であり，*key*.Length は *key* の長さを返す関数である．*key*[*key*.Length()] は終端記号を返すものとする．

```
Trie::Build(pairs)
  foreach pair in pairs ordered by keys
    Insert(pair.key, pair.record);

Trie::Insert(key, record)
  pos := 0;   node := root;
  while ( pos ≤ key.Length() )
    child := node.child;
    if ( child = nil or key[pos] ≠ child.label )
      break;
    pos := pos+1;   node := child;
  while ( pos ≤ key.Length() )
    node.child := New(
      sib => node.child, label => key[pos]);
    pos := pos+1;   node := node.child;
  node.record := record;
```

図 3 トライ辞書の構築アルゴリズム

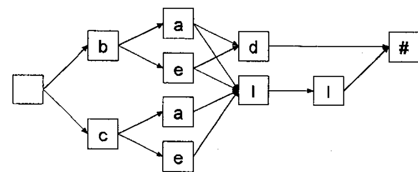

図 4 Directed Acyclic Word Graph (DAWG)

3. DAWG

3.1 DAWG の概要

DAWG はトライの共通部分木を併合して得られるグラフであり，同数のノードでも多くのキーを格納することができる．例として，図 1 のトライ辞書からレコードを取り除き，DAWG へと圧縮した結果を**図 4** に示す．図 1 において，'*d*' や '*l*' をラベルとするノード以降は共通部分木となっている．そのため，共通部分木の併合により，ノード数は 23 から 11 へと減少している．

トライから DAWG への圧縮は，検索アルゴリズムの変更を必要としないため，検索時間を悪化させることなく，大規模なデータの操作を可能とする．しかし，従来の研究では，レコードを必要としない用途を想定しており，大規模な DAWG 辞書を構築する効率的なアルゴリズムの提案はおこなわれていない†．

3.2 DAWG の静的な構築アルゴリズム

キー集合から DAWG を構築する方法としては，まず二分木表現によるトライを構築し，DAWG へと圧縮する手法が知られている [7]．この手法では，二分木表現におけるノードを DAWG のノードとして利用するため，異なる兄弟ノードを持つノードが併合されず，**図 5** のようになる．結果として，図 4 のような最適構造より多くのノードが必要となるものの，入力データのサ

†DAWG に関する従来研究の多くは接尾辞木に対する圧縮構造としての利用を想定しており [11, 12]，辞書としての利用を想定した研究は数少ない．

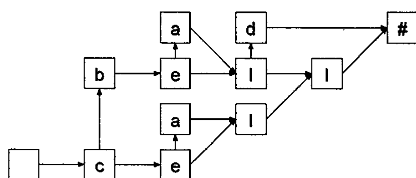

図 5 DAWG の内部表現

```
Dawg::Convert(trie)
  tables := empty;
  Classify(tables, trie.root);
  for ( i := 0; i < tables.Size(); i := i+1 )
    local_table := empty;
    foreach node in tables[i]
      matched_node := local_table.Find(node);
      if ( matched_node = nil )
        local_table.Insert(node);
      else
        node.UpdateInEdge(matched_node);

Dawg::Classify(tables, node)
  if (node = nil)
    return 0;
  child_dist := Classify(tables, node.child);
  sib_dist := Classify(tables, node.sib);
  max_dist := Max(child_dist, sib_dist);
  tables[max_dist].Insert(node);
  return max_dist+1;
```

図 6 トライから DAWG への圧縮アルゴリズム

イズに対して線形時間で DAWG を構築できる．

トライから DAWG への圧縮アルゴリズムを**図 6** に示す．Dawg::Convert はトライを DAWG へと圧縮する手続きであり，Dawg::Classify はバケットソートによりノードを整列する関数である．Dawg::Classify において，各ノードへの参照 *node* は，*child* と *sib* を辿って到達可能な葉の中で，最も遠くにある葉までの距離により分類され，*tables* に格納される．

図 6 において，*table*.Insert はノードを *table* に登録する手続きである．一方，*table*.Find は *child*, *sib*, *label* の等しいノードを *table* から検索する関数であり，該当するノードへの参照もしくは nil を返す．また，*node*.UpdateInEdge は，トライにおける *node* への参照を更新する手続きである．

線形時間で DAWG を構築するには，*local_table* の Insert と Find，*node*.UpdateInEdge を $O(1)$ の時間計算量で実現すればよい．そして，この条件を満たすには，*local_table* をハッシュ表により実装し，トライを構築する段階で，各ノードに自身への参照への参照を格納すればよい．

トライから DAWG への圧縮アルゴリズムは，構築時間において優れた手法である．しかし，各ノードに参照を一つずつ追加したトライを構築し，さらに各ノードへの参照を分類して格納する領域を確保しなければならないため，作業領域が大きくなるという欠点を持つ．参照に 4 bytes，ラベルに 1 byte を割り当てる場合，ノード数 n に対して，必要となる作業領域は $17n$ bytes であり，トライに割り当てられる領域の 2 倍近い．そのため，大規模なトライの圧縮に応用することは難しい．

3.3 DAWG の動的な構築アルゴリズム

DAWG の構築手法として，任意にキーを登録できる動的なアルゴリズムが提案されている [6]．しかし，キーを登録する度にノードの分割と併合をおこなうため，ノード数 n に対して，キー登録の時間計算量は $O(n)$ 以上となることが示されている．さらに，キー数 k が与えられるとき，DAWG 構築の時間計算量は $O(n \cdot k)$ 以上となる．そのため，登録するキーが少ない状況では実用的といえるものの，キー数 k が 10 万を超える規模になると，実用的な時間で DAWG を構築できないという欠点を持つ．

4. DAWG 辞書

4.1 DAWG 辞書の概要

本研究では，コーパスにおけるキーの出現頻度をレコードとして用いる場合など，重複レコードが多数存在する状況を想定し，DAWG を辞書として用いることを提案する．従来手法との違いは，共通部分木の判定において，参照とラベルだけでなく，レコードも基準として利用することである．そのため，レコードの重複が多いほど共通部分木が出現しやすくなり，圧縮率が高くなるという特徴を持つ．

図 1 のトライ辞書に対する DAWG 辞書を**図 7** に示す．図 4 の DAWG と比較すればノード数が 11 から 12 へと増加しているものの，元のトライ辞書と比較すればノード数が 23 から 12 へと減少しており，レコードが重複する状況における DAWG 辞書の有効性を示している．

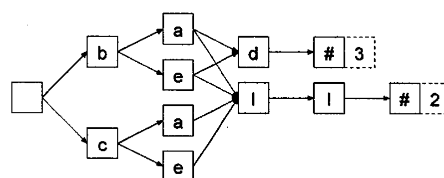

図 7 DAWG 辞書

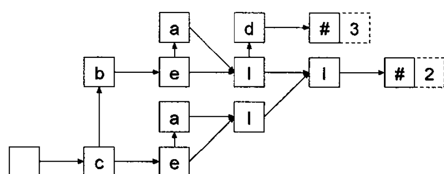

図 8 DAWG 辞書の内部表現

```
Dawg::Build(pairs)
  table := empty;
  stack := { root };
  foreach pair in pairs ordered by keys
    Insert(table, stack, pair.key, pair.record);
  Merge(table, stack, root);

Dawg::Insert(table, stack, key, record)
  pos := 0;   node := root;
  while ( pos ≤ key.Length() )
    child := node.child;
    if ( child = nil or key[pos] ≠ child.label )
      Merge(table, stack, node);
      break;
    pos := pos+1;   node := child;
  while ( pos ≤ key.Length() )
    node.child := New(
      sib => node.child, label => key[pos]);
    pos := pos+1;   node := node.child;
    stack.Push(node);
  node.record := record;

Dawg::Merge(table, stack, node)
  while ( stack.Top() ≠ node )
    unfixed_node := stack.Pop();
    matched_node := table.Find(unfixed_node);
    if ( matched_node = nil )
      table.Insert(unfixed_node);
      break;
    stack.Top().child := matched_node;
    delete unfixed_node;
  while ( stack.Top() ≠ node )
    table.Insert(stack.Pop());
```

図 9 DAWG 辞書の構築アルゴリズム

4.2 DAWG 辞書の構築アルゴリズム

本稿で提案する構築アルゴリズムは，トライを介することなく DAWG 辞書を構築する手法であり，辞書順にキーを登録する手続きの中で共通部分木の併合をおこなう．提案手法では，キーを逐次登録するため，すべてのキーをメモリ上に展開する必要がなく，また，トライを構築する必要がないため，少ない作業領域で大規模な DAWG 辞書を構築できる．さらに，従来の静的な構築アルゴリズムと同様に，入力データのサイズに対して線形時間で DAWG を構築することが可能である．

提案手法における DAWG の表現方法は，トライ辞書に対する二分木表現と同じであり，各ノードに *child*, *sib*, *label*, *record* をメンバとして持たせる．例として，図 7 の DAWG 辞書に対する内部表現を**図 8** に示す．図 4 の DAWG に対する図 5 の内部表現と同様に，最適構造と比較すればノード数が 12 から 13 へと増加しているものの，図 1 と比較すればノード数が 23 から 13 へと減少している．

DAWG 辞書の構築アルゴリズムは，**図 9** に示すように，図 3 の Trie::Build と Trie::Insert を拡張することにより得られる．図 9 において，Dawg::Build は DAWG 辞書を構築する手続きであり，Dawg::Insert はキーとレコードを DAWG 辞書に登録する手続きである．Dawg::Build と Dawg::Insert の下線部は更新箇所を表しており，共通部分木を検索するための索引構造 *table* と，未判定のノードを格納するためのスタック領域 *stack*，手続き Dawg::Merge の呼び出しが追加されている以外の変更点はないことを示している．なお，*stack*.Push は *stack* の頂点にノードへの参照を積む手続きである．

Dawg::Merge は共通部分木を検索・併合する手続きであり，直近に確保された葉から *node* に到達するまでのノードを併合の対象とする．共通部分木の併合をこのように単純化できる理由は，辞書順にキーを登録するという制約と，二分木表現を基にした内部表現にある．なお，*stack*.Top と *stack*.Pop は *stack* の頂点に積まれている要素を取得する関数であり，*stack*.Top は要素を *stack* に残すのに対し，*stack*.Pop は *stack* から要素を取り除く．

提案手法において，Dawg::Merge の呼び出しを除けば，Dawg::Build と Dawg::Insert の時間計算量は，Trie::Build と Trie::Insert の時間計算量に等しい．そのため，提案手法による DAWG 辞書の構築時間は，Dawg::Merge における *table*.Find と *table*.Insert の時間計算量に依存する．本稿では，二分探索木およびハッシュ表により *table* を実装する方法について説明する．

4.3 DAWG 辞書の構築における索引構造

二分探索木により索引構造を実装する場合，DAWG 辞書の各ノードを二分探索木のノードとして利用できるように，各ノードに左右の子への参照を持たせればよい．このとき，元になるトライ辞書のノード数 n と DAWG

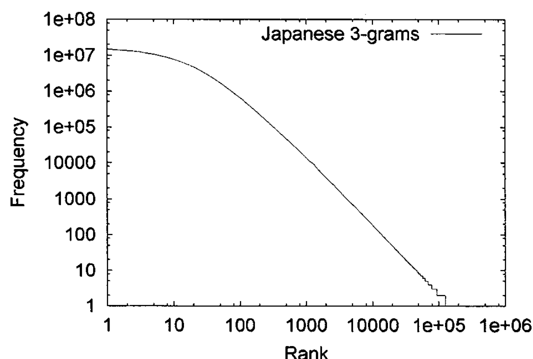

図 10 日本語 3-gram データにおける頻度分布

辞書のノード数 m に対して，DAWG 辞書構築の時間計算量は $O(n \log m)$ であり，作業領域は $17m$ bytes となる．なお，評価実験では，Left-Leaning Red-Black Tree[13] を二分探索木の実装として用いている．

一方，ハッシュ表により索引構造を実装する場合，DAWG 辞書構築の時間計算量は $O(n)$ であり，入力データのサイズに対して線形時間となる．しかし，ノードへの参照を格納するハッシュ表が作業領域として必要であり，十分なサイズを確保できない場合，衝突により性能が悪化する．評価実験では，充填率が 75%以上になる度にハッシュ表のサイズを 2 倍に拡張したため，作業領域は 14–20m bytes となっている．

5. 実験による評価

5.1 実験設定

提案手法の有効性を示すため，Pentium(R) Dual-Core 2.50GHz, 8GB RAM という構成のシステムを使用し，Linux (Ubuntu 8.10) 上で評価実験をおこなった．コーパスには，Google により公開されている日本語 n-gram データ [8] に含まれるすべての 3-gram 394,482,216 件を使用した．ただし，確保できる作業領域に制限があるため，ノード数に対して 4 億という上限を設定している．最大作業領域は，二分探索木を用いた場合で 6.8GB，ハッシュ表を用いた場合で 5.6GB である．このとき，ハッシュ表のサイズは $2^{29} = 536{,}870{,}912$ まで拡張され，充填率は 74.5%となる．

コーパスの 3-gram は辞書順に整列されており‡，ウェブコーパスにおける出現頻度が付与されている．そして，3-gram の頻度分布には，**図 10** に示されるように，Zipf の法則があてはまる．

評価実験では，以下のように，圧縮されたコーパスを復号しながら DAWG 辞書を構築した．

```
time gzip -cd 3gms/*.gz | ./build-dic
```

‡実験環境では，ランダムに並べ替えた 3-gram を Linux のコマンド sort で辞書順に整列するのに約 27 分を要する．

表 1 重複レコードによる登録可能キー数の増加

レコード	キー数 k	ノード数 n	k/n
固有 ID	74,029,090	400,000,001	0.185
頻度	230,859,628	399,999,996	0.577
対数頻度	368,084,494	399,999,992	0.920
なし	394,482,216	318,840,667	1.237

表 2 Linux のコマンド time による構築時間の計測結果

レコード	real	user	sys
固有 ID	7m04.2s	6m53.2s	0m10.9s
頻度	13m11.8s	12m53.3s	0m18.3s
対数頻度	16m04.3s	15m39.1s	0m24.7s
なし	14m07.7s	13m45.5s	0m22.1s

コーパスのサイズは，圧縮状態で 2,442,780,774 bytes であり，復号すると 9,419,643,438 bytes になる．

5.2 重複レコードと圧縮性能の関係

重複レコードの割合が圧縮性能に与える影響を調査するため，固有 ID，頻度，対数頻度，0 をレコードとし，ハッシュ表による実装を用いて DAWG 辞書を構築した．登録できたキー数を**表 1** に示し，Linux のコマンド time により計測した実行時間を**表 2** に示す．また，レコードを持たない DAWG 辞書の構築において，併合されるノードを計数したところ，トライ辞書にすべてのキーを登録すれば，ノード数は 2,127,080,491 になることが判明した．

実験結果より，トライを介する従来手法では作業領域が大きくなり，大規模な DAWG の構築は困難であることが分かる．また，重複するキーの割合が高くなるほど圧縮性能が向上し，同じノード数で格納できるキー数は 3.1–6.7 倍まで増加することが示されている．さらに，4 億のノードで構成される大規模な DAWG 辞書を 10–20 分程度の実時間で構築できることが分かる．以上のことから，重複レコードの多い大規模なトライ辞書の圧縮において，提案手法は有効であるといえる．

5.3 索引構造と構築時間の関係

提案手法による DAWG 辞書の構築時間を**図 11** と**図 12** に示す．時間の計測には C++の関数 std::clock を使用し，1,000 万件のキーを登録する度に，構築開始からの経過時間を求めた．

実験結果において，ハッシュ表が拡張される状況を除き，キー数の増加による構築時間の急激な悪化は確認できない．また，ハッシュ表を用いるとき，あらかじめノード数を予測できる状況では，最初に十分なサイズのハッシュ表を用意することにより，構築時間のさらなる短縮が可能である．そのため，ハッシュ表は索引構造として二分探索木より優れているといえる．

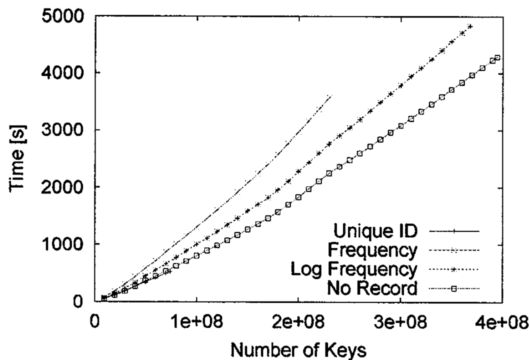

図 11 二分探索木による実装を用いた構築時間

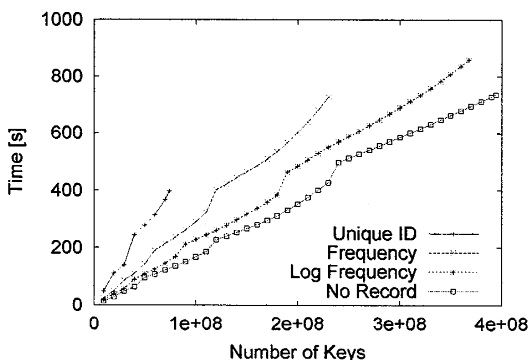

図 12 ハッシュ表による実装を用いた構築時間

5.4 辞書の規模とノード数の関係

キー数とノード数の関係を**図 13** に示す．固有 ID，頻度，対数頻度，0 をレコードとして用いたとき，ノード数の比は，キー数が 1,000 万件で 4.61 : 1.88 : 1.27 : 1.00 となり，キー数が 5,000 万件で 5.39 : 1.94 : 1.31 : 1.00 となっている．このことから，提案手法は大規模なトライ辞書に対して特に有効であることが分かる．

6. おわりに

本稿では，大規模なトライの圧縮構造として DAWG が有用であることを述べ，単純かつ効率的な構築アルゴリズムを提案した．提案手法は重複レコードの多いトライ辞書に有効であり，大規模な辞書を短時間でコンパクトに構築できることが実証されている．

提案手法による圧縮は，詳細な形態素情報や意味情報をレコードとする一般的な自然言語辞書には効果がないものの，単純な属性や重みをレコードとする辞書，あるいは統計的言語モデルや組み合わせ素性などに有効であり，大規模コーパスを利用する研究への応用が期待される．

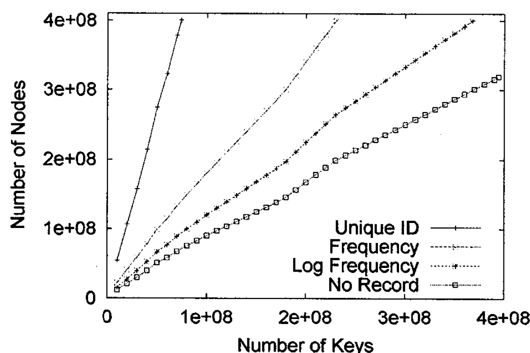

図 13 登録キー数によるノード数の変化

参考文献

[1] Edward Fredkin. Trie Memory. *Communications of the ACM.*, Vol. 3, No. 9, pp. 490–500, September 1960.

[2] Taku Kudo. MeCab: Yet Another Part-of-Speech and Morphological Analyzer. `http://mecab.sourceforge.net/`, February 2008.

[3] Tian Song, Wei Zhang, Dongsheng Wang, and Yibo Xue. A Memory Efficient Multiple Pattern Matching Architecture for Network Security. In *The 27th Conference on Computer Communications (INFOCOM 2008)*, pp. 166–170, Phoenix, AZ, USA, April 2008.

[4] 吉永直樹, 喜連川優. 組み合わせ素性に基づく分類器の高速化と係り受け解析への適用. 言語処理学会第 15 回年次大会発表論文集, pp. 28–31, March 2009.

[5] Andrew W. Appel and Guy J. Jacobson. The World's Fastest Scrabble Program. *Communications of the ACM.*, Vol. 31, No. 5, pp. 572–578, May 1988.

[6] 青江順一, 森本勝士, 長谷美紀. トライ構造における共通接尾辞の圧縮アルゴリズム. 電子情報通信学会論文誌, Vol. J75-D-II, No. 4, pp. 770–779, April 1992.

[7] Susumu Yata, Kazuhiro Morita, Masao Fuketa, and Jun-ichi Aoe. Fast String Matching with Space-efficient Word Graphs. In *Innovations in Information Technology (Innovations '08)*, pp. 79–83, Al Ain, United Arab Emirates, December 2008.

[8] 工藤拓, 賀沢秀人. Web 日本語 N グラム第 1 版. `http://www.gsk.or.jp/catalog/GSK2007-C/catalog.html`, 2007.

[9] Jun-ichi Aoe. An Efficient Digital Search Algorithm by Using a Double-Array Structure. *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066–1077, 1989.

[10] Jon L. Bentley and Robert Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 1997.

[11] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[12] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007.

[13] Robert Sedgewick. Left-leaning Red-Black Trees. `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`, September 2008.