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 泳田 正雄 青江 順一
Susumu Yata Kazuhiro Morita Masao Fuketa Jun-ichi Aoe
德島大学工学部 Faculty of Engineering, University of Tokushima
†日本学術振興会特別研究員 PD

1. はじめに


キー集合としての DAWG に関する研究では、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いためである。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、重複レコードの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、キーを辞書項に登録していくことにより、トライに介することなく 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 倍以下のキーを登録できことが示された。また、4 億のノードで構築される大規模な DAWG を 10-20 分程度の実時間で構築できことが確認された。

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方法を提案し、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める。

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、文字列の経路記号をラベルとして用いることにより、キー同士の識別が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、直後の数値が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]。

(第 2 分冊)
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になる場合のみrecordが必要となり、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4bytesずつ、ラベルに1byte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9nbytes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、左方向の矢印と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、左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えない、探索結果を保存せずに新たな経路を導き、新記録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導き、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[0..key.Length()]は辞書順に対応するものとする。

### 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る。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'i'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も多くのノードが必要となるものので、入力データのサ

---

1 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。
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を通っ
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から探索する関数であり、
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
照を一旦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4bytes、ラベルに1byteを割り当てる場合、
ノード数nに対して、必要な作業領域は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
与えられたとき、DAWG構築の時間計算量はO(n∗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性を持つ。

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され
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。
4.2 DAWG 詞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 詞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と、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 詞書を構築できる。さらに、従来の静的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 辞書内の内部表現を図 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 詞書を構築する手続きであり、Dawg::Insert はキーとレコードを DAWG 詞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、共通部分木を構築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 を実装する。Dawg::Merge の呼び出しが追加されている以外の変更点はないと示している。なお、stack:Pop 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、直接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で、二分木表現を基にした内部表現がある。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
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は作業領域に影響を与えるのであれば、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%以上の信頼にかかって、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 により開発されている日英プロジェクトデータ[8] に含まれるすべての 3-gram 394,822,166件を使用した。ただし、確保できる作業領域の制限があるため、ノード数に対して 4 倍という上限を設定している。最適作業領域は、二分探索木を用いた場合で 6.8G、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 は辞書順に整形されており、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 の頻度分布については、図 10 に示されるように、Zipf の法則が当てはまる。

評価実験では、以下の通り、圧縮されたコーパスを復号しながら DAWG 辞書を構築した。

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

![図 10 日本語 3-gram データにおける頻度分布](image)

表 1 重複レコードによる登録可能キー数の増加

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

表 2 Linux のコマンド time による構築時間の計測結果

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

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、合計されるノード数を計算したところ、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 個のノードで構築される大規模な DAWG 辞書を 10-20 分程度の実質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。
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。

参考文献