辞書と言語モデルの効率のよい圧縮とかな漢字変換への応用

花岡 俊行, 田畑 悠介, 向井 淳, 小松 弘幸, 工藤 拓
Google 株式会社
{toshiyuki,tabata,mukai,komatsu,taku}@google.com

1　はじめに

Web コーパスを用いた言語処理のひとつの課題に, 自動構築された辞書や言語モデルの大きさが挙げられる. 特にかな漢字変換システムへ応用する場合, ユーザのコンピュータ上で動作させる必要があるため, 計算機資源の制約が大きく, 検索速度に対する要求も高い. 大容量の辞書や言語モデルとその検索速度を両立させるためには効率のよい圧縮が不可欠である.

我々が開発している Mozc[1](製品版 Gooogle 日本語入力) では, 大規模 Web コーパスより自動構築された辞書や言語モデルを用いて性能の向上を図っている.

本論文では, まず, Mozc における語彙数の増加が変換精度に与える影響について述べる. 次に, 語彙数の多い辞書や大きな言語モデルをユーザーのコンピュータ上で扱うための圧縮手法について述べる.

2　語彙数と変換精度の関係

従来のかな漢字変換システムでは, 医療辞書やトレンド辞書といった拡張辞書はデフォルトでは組み込まれておらず, ユーザが必要に応じて有効にする必要があった. その理由として, 大規模な語彙集合を扱うことが辞書インデックスの制約上難しかったことと, 副作用により精度が低下することが挙げられる. Mozc では, 大規模な Web コーパスから言語モデルを構築することで, 変換精度に対する副作用の問題を解決している [5]. Mozc における語彙数と変換精度の関係を調べるために, 以下のような簡単な調査を行った.

1. 単語コストのある値を定め, そのコストを超える単語を辞書から取り除く. 単語のコストは出現頻度を元にした言語モデルから決められており, コストが低いほど出現頻度が高い.

2. 語彙数を減らした辞書を用いてコーパスの文に対する変換精度 (BLEU スコアの平均) を測定する.

かな漢字変換の評価コーパスとしては Anthy[2] の言語モデルの学習に使用されている読みと変換結果のペアのデータ[3]を用いた. このデータはユーザから提供されたデータを多く含んでいるという特徴があり, 実際のユーザの入力単位などを反映した実用的な評価コーパスだと考えられる. 評価指数として, BLEU[4] を用いた. BLEU は機械翻訳の分野で広く使われている尺度であり, 正解データと変換結果の n-gram の一致度を元にした指標である.

結果を表 1 に示す. 一般的な傾向として, 語彙数が増えると変換精度が向上していることが分かる. 語彙 30 万以上では, 語彙数が 2 倍になるごとに BLEU スコアが 0.02 ポイント程度上昇している. 単純な語彙数の増加を変換精度の向上に還元できている点が, Mozc において豊富な語彙をサポートしたい動機となっている.

cost	#words	BLEU
10000	3035000	0.885
9000	2531000	0.883
8000	1497000	0.874
7000	709000	0.859
6000	317000	0.833
5000	127000	0.782
4000	44000	0.653
3000	15000	0.518

表 1: 語彙数と精度

3　辞書の圧縮

3.1　かな漢字変換のための辞書構造

Mozc の辞書の 1 エントリーは, 読み, 変換後文字列 (以下単語), コスト, 品詞 ID のタプルから構成されている. 読み, 単語はそれぞれ文字列であり, コストは単語の出現頻度に基づいて計算された 2byte の整数である. 品詞 ID は Mozc が採用しているクラス言語モデルのクラス ID に対応し, 単語を左から見たときと右から見たときの品詞 ID を考慮し, それぞれ 2byte, 計 4byte の整数で表現している.

かな漢字変換システムに用いる辞書のデータ構造として, 読みに対する Common Prefix Search (読み s に対し, s のプレフィックスとなる読みの検索) が要求される. 加えて Predictive Search (読み s に対し, プレフィックスが s となる読みの検索) や Reverse Lookup (単語から読みを逆方向に検索) をサポートしていれば, かな漢字変換システムの機能の一部である予測変換や再変換の実装に用いることができる.

Common Prefix Search や Predictive Search に適した辞書構造として, Trie がある. Trie は順序付き木構造であり, 長さ n のキーに対して $O(n)$ の計算量で Common Prefix Search, Predictive Search が行える. Trie の実装方法としては, Double Array[1] や LOUDS[3] 等がある.

3.2　Double Array

Mozc の初期の実装では Double Array[1] に基づくオープンソース Trie ライブラリ Darts [4] によって辞書を実装していた. Double Array は比較的小さな辞書表現で高速に検索できる特徴を有する. しかし, Mozc

[1] http://code.google.com/p/mozc/
[2] http://sourceforge.jp/projects/anthy/
[3] Anthy ソースツリー, calctrans/ 以下より入手できる
[4] http://chasen.org/~taku/software/darts/

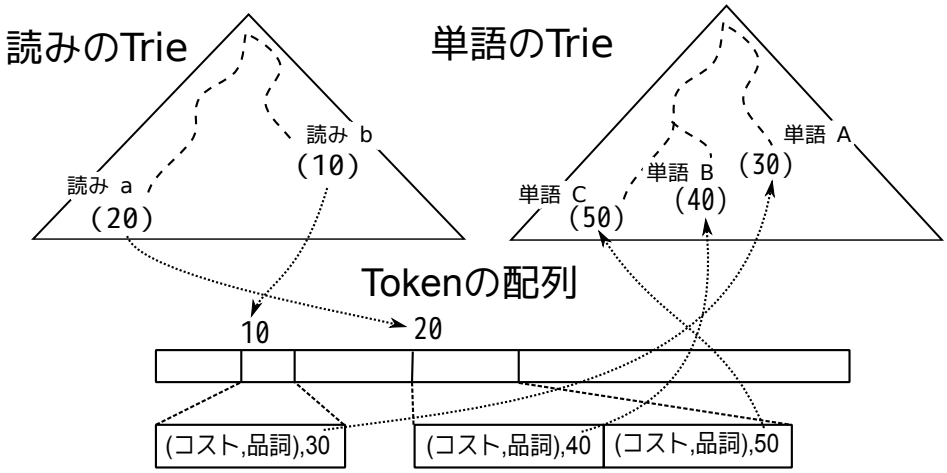

図 1: 辞書の模式図

に用いる語彙数が増えるにつれて辞書サイズが増大し, ユーザのコンピュータ上で動作させるという制約を満たせないと判断した.

3.3 LOUDS

現在の Mozc では, 辞書の構造として簡潔データ構造の一種である LOUDS[3] を用いている. 実装には, オープンソースのライブラリ rx[5]を使用している. LOUDS は Trie を一つの配列の簡潔データ構造で表現しており, Double Array に比べてより小さな領域で Trie を実現できる.

LOUDS では木を幅優先で辿り, ノードの次数の 1 と 1 個の 0 を並べたビット列で木を表現する. 図 2 に例を示す. 木の根には番兵 super root を足す. ここで,

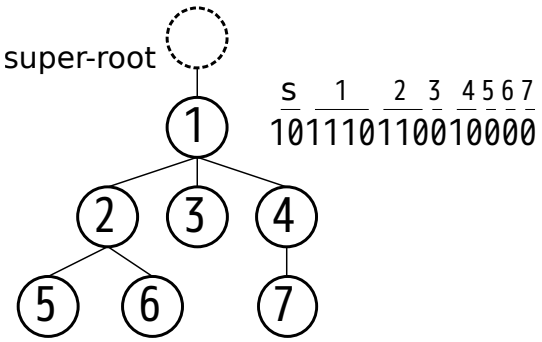

図 2: LOUDS における木の表現

i 番目の節点は 1 と 0 によって二回表現されており, 親から子を辿るときには i 番目の 1, 子の中では $(i+1)$ 番目の 0 によって表される. LOUDS では, 木に対する基本的な操作をビット配列上での演算として表現することができる. 例えば, 配列上で m 番目に対応するノードの最初の子は $select(0, rank(1, m))+1$, 次の兄弟は $m+1$, 親は $select(1, rank(0, m))$ としてアクセスできる. ここで, $rank(n, m)$, $select(n, m)$ はそれぞれ, m 番目以下にある n の数を求める関数, m 番目の n の場所を求める関数である. LOUDS ではノード数 n の木を $2n$ bit で表現しており, 空間効率性において優れている.

[5]http://sites.google.com/site/neonlightcompiler/rx/

3.4 Mozc の辞書構造

Mozc の辞書は読みと単語の文字列に対する二つの Trie, コストや品詞の単語情報 (以下 Token) の配列によって表現されている. 読みだけでなく, 変換結果の単語も Trie で表現することにより, 単語の共通部分を共有し圧縮率を高めるだけでなく, 単語から読みへの検索も可能となっている.

読み	単語	読み Trie 終端のノード ID	単語 Trie 終端のノード ID
a	A	20	30
b	B	10	40
b	C	10	50

表 2: 登録する単語の例

表 2 に示す辞書エントリーが, どのように Trie にエンコードされているかを図 1 に示す. 読みの Trie の終端ノードの ID が Token 配列のインデックスに対応する. Token 配列には同じ読みを持つ各単語のコスト, 品詞, 及び単語 Trie における ID が保存されている.

読みから単語への検索は, 次のような手順によって行われる.

1. 読みの Trie に対する各操作 (Common Prefix Search, Predictive Search) により, 読みの Trie におけるノード ID の集合を得る.
2. それぞれの ID より Token 配列を参照し, 対応する各単語のコスト, 品詞, 単語の Trie における ID を得る
3. 単語の Trie を葉から根の方向に走査し, 単語の文字列を得る.

同構造を使えば, 単語から読みへの検索も実行できる.

1. 単語の Trie に対する各操作 (Common Prefix Search, Predictive Search) により単語 Trie におけるノード ID の集合を得る.
2. Token 配列を線形探索し, 対応する各単語の コスト, 品詞, 読みの Trie における ID を得る.
3. 読みの Trie を葉から根の方向に走査し, 読みの文字列を得る.

Reverse Lookup (逆変換) の場合は, Token 配列に対する線形探索が必要となるため, 読みから単語への検索に比べて動作が遅くなる. ただし, 線形探索の結果を実行時にキャッシュすることで高速化を図っている[6].

3.5 その他の工夫

圧縮率を高めるために, 上記の構造に加えて, 以下に示す圧縮を組み合わせている.

1. Token 配列の圧縮
   頻度の高い品詞を短く表現できるように頻出品詞のテーブルを作成し, 頻出品詞に対してはそのインデックスのみを単語情報に記憶する. この結果, Token 配列も可変長配列となる[7].
2. 同値表現の圧縮
   読みと単語が同じものや, 読みをそのままカタカナにした単語は, 単語を Trie に登録せず, Token の中に同値表現を示すフラグのみを保存する.
3. 文字列の可変長圧縮
   Trie に登録する読み, 単語の文字列を, 文字の出現頻度に応じて長さを変えたエンコーディングによって表現する. 例えば, ひらがな・カタカナは 1 byte で, 頻度の高い漢字は 2 byte で, その他は 3 byte で表現する.

3.6 評価

圧縮率の比較のため, 各実装における領域量を表 3 に示す. LOUDS+Token 圧縮は, LOUDS に Token 配列の圧縮を適用し, すべての読み, 単語を UTF-8 エンコーディングで保存した場合, LOUDS+Token/同値表現圧縮は, さらに同値表現の圧縮を適用した場合, LOUDS+Token/同値表現/文字列圧縮は, 3.5 で示したすべての圧縮方法を組み合わせた場合である. 内訳の Token, 読み, 単語はそれぞれ Token 配列, 読み Trie, 単語 Trie のサイズを示している.

評価に用いた辞書ファイルには 1,345,900 エントリーが含まれており, テキストデータで 59.1 Mbyte である. 読み, 単語は, それぞれ重複を除き 866,100, 1,194,600 エントリーが含まれている.

Double Array による実装では単純なテキスト表現よりサイズが大きくなっている. ただし, 当時の実装では読みに対する Trie は表現しているものの, 単語に対してはそのまま記憶している.

LOUDS を用い, 読みと単語に対して Trie を作成した場合, 全体のサイズはテキスト表現の 1/3 程度となっている. さらに, 同値表現や文字列の圧縮を適用することで単語の Trie のサイズが減少することが分かる. また, 同値表現の圧縮を用いた場合, 単語 Trie に登録される単語の数が少なくなるため, 単語 Trie のサイズも減少する. 文字列圧縮を適用すると, 読み, 単語の Trie のサイズが共に減少している. 特に読みには主にひらがなしか出現しないため, 文字列圧縮の効果が大きく, サイズが 40 %程度まで減少している.

以上の圧縮により Mozc では 1,345,900 語を 13.3 Mbyte の領域量で表現し, Common Prefix Search, Predictive Search, 及び Reverse Lookup を実現している. 一単語当たりのサイズは約 10.4 byte となっている.

4 言語モデルの圧縮

4.1 Mozc の言語モデル

Mozc では, 言語モデルとしてクラス言語モデルを採用している [5]. クラス言語モデルは, 各クラスから単語が生成される確率 (単語生起確率) とクラス間の遷移確率 (クラス遷移確率) から構成される. 単語生起確率は辞書の各エントリーに整数のコストとして保存される. クラス遷移確率も実数値ではなく整数のコストとして表現している.

Mozc のクラス言語モデルは, 多くの単語を語彙化しているため, クラス数が 3000 にもなる [5]. しかし, 一般には連接しない遷移が多く, クラス遷移行列は疎行列となる. 疎行列の圧縮の例としては, 圧縮行格納方式 [2] などが知られている.

4.2 クラス遷移確率行列の圧縮

Mozc では, 行列のインデックスを表現した 8 分木とコストの配列によってクラス遷移確率行列を表現している. データ構造は次の手順で作成される.

1. 二次元の行列をインデックスの線形操作で一次元の配列に変換する. 一次元にしても疎行列の性質は失われない.
2. コストを持つ (コストが未定義ではない) インデックスのビット表現を 3bit ずつ分割する. 分割された値は 8 通り (3bit) の値をとる.
3. 分割したインデックスをノードにもつ 8 分木を作成し, 端点にコストを記憶する.

図 3 に, 一次元で表現されたインデックス 434, 406, 176 に対し, それぞれ cost0, cost1, cost2 が定義されているとき, これらがどのように 8 分木として表現されるかを示す. これらのインデックスは二進数表記でそれぞれ 110110010, 110010110, 010110000 であり, 3 bit ずつ区切るとそれぞれ $(6, 6, 2)$, $(6, 2, 6)$, $(2, 6, 0)$ である. 例えば, インデックス 434$(6, 6, 2)$ は, level0 のノードの 6bit 目, level1 のノードの 6bit 目, level2 のノード 2bit 目がそれぞれ 1 となり[8], 終端に cost0 が保存される.

[6] 再変換のインデックスを構築することも考えられるが, そもそも再変換の使用頻度は少ないことと, リソースの制約から, 実行時に構築している.

[7] 可変長配列は rx のモジュール rbx が提供している.

[8] 便宜上 0 オリジンとしている.

辞書表現方法	サイズ (Mbyte)	1 単語当たり (byte)	内訳 (単位は Mbyte)
プレーンテキスト	59.1	46.0	
Double Array	80.8	63.0	
LOUDS+Token 圧縮	20.5	16.0	Token: 8.5 読み: 5.8 単語: 6.2
LOUDS+Token/同値表現圧縮	18.3	14.2	Token: 7.9 読み: 5.8 単語: 4.6
LOUDS+Token/同値表現/文字列圧縮	13.3	10.4	Token: 7.9 読み: 2.4 単語: 3.0

表 3: 辞書の表現方法とサイズ

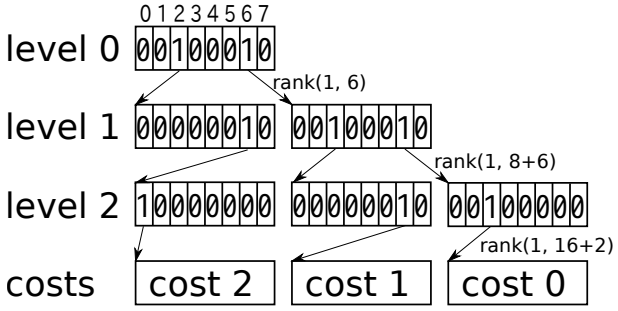

図 3: 言語モデル表現

各ノード間のポインタ (図 3 中の矢印) は, rank 操作を使えば明示的に記憶する必要がない. 各 level のノードをつなげて一つの大きなビット配列とみなした場合, 一般に, level k の m bit 目の子供は level $(k+1)$ では, $rank(1, m)$ 番目のノードに対応する.

4.3 評価

表 4 に各実装方法におけるクラス遷移確率行列のサイズを示す. サイズの測定には Mozc バージョン 1.0.558.102 に使用されているクラス遷移行列を用いた. クラスのサイズは 3019 であり, コストは 2byte で表現されている. そのうちコストが定義されている要素は 1,272,002 あり, およそ $14\%(\simeq 1,272,002/(3019\cdot 3019))$ の充填率となっている. 単純な配列表現では $3019 \cdot 3019 \cdot 2 = 18,228,722(\text{byte}) \simeq 17.4(\text{Mbyte})$ の領域を必要とする. 一方, 8 分木による表現では, 8 分木そのものに 0.51Mbyte, コストの配列に 2.4Mbyte $(= 1,272,002 \cdot 2\text{byte})$, 合計 2.9Mbyte となった. ランダムな時の下限とは, 密な要素がランダムに分布したと仮定したときに必要な記憶領域の下限値であり, インデックスを記録するのに 0.63Mbyte [9], コストの配列に 2.4Mbyte, 合計 3.1Mbyte となった.

単純な配列表現に比べて, 8 分木表現が 16%ほどの領域量で疎行列を圧縮できていることが分かる. また, ランダムな時の下限値に近い値となっており, 8 分木の手法が妥当なことを示している. 下限値よりも圧縮率が向上した理由として, 疎行列のインデックスに偏りがあったことが考察される.

[9] $\log_2 ({}_{3019\cdot 3019}C_{1272002})$ より求められる.

表現方法	サイズ Mbyte
配列表現	17.4
8 分木表現	2.9
ランダムな時の下限	3.1

表 4: 言語モデルの表現方法とサイズ

5 おわりに

本論文では, Mozc において豊富な語彙をサポートするに至った語彙数と変換精度の関係を示した. また, Mozc で用いられている大規模辞書や言語モデルをユーザーのコンピュータ上で扱うための手法について述べ, それぞれの手法における圧縮効率を評価した.

さらなる圧縮にはコスト値, すなわちロスありの圧縮も視野に入れる必要があるだろう. コスト値をどの程度圧縮すると, 変換精度にどの程度影響があるのかといった調査を行い, さらなる圧縮率の向上を目指したいと考える.

参考文献

[1] Jun-ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Trans. Softw. Eng.*, Vol. 15, pp. 1066–1077, 1989.

[2] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, Henk van der Vorst, and Long Restart. Templates for the solution of linear systems: Building blocks for iterative methods. 1993.

[3] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pp. 549–554. IEEE Computer Society, 1989.

[4] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu. Bleu: a method for automatic evaluation of machine translation. *ACL-2002: 40th Annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

[5] 工藤拓, 小松弘幸, 花岡俊行, 向井淳, 田畑悠介. 統計的かな漢字変換システム mozc. 言語処理学会 第 17 回年次大会, 2011.