Efficient dictionary and language model compression for input method editors

Taku Kudo, Toshiyuki Hanaoka, Jun Mukai, Yusuke Tabata, and Hiroyuki Komatsu
Google Japan Inc.
{taku,toshiyuki,mukai,tabata,komatsu}@google.com

Abstract

Reducing size of dictionary and language model is critical when applying them to real world applications including machine translation and input method editors (IME). Especially for IME, we have to drastically compress them without sacrificing lookup speed, since IMEs need to be executed on local computers. This paper presents novel lossless compression algorithms for both dictionary and language model based on succinct data structures. Proposed two data structures are used in our product "Google Japanese Input"[1], and its open-source version "Mozc"[2].

1 Introduction

Statistical approaches to processing natural language have become popular in recent years. Input method editor is not an exception and stochastic input methods have been proposed and rolled out to real applications recently (Chen and Lee, 2000; Mori et al., 2006; Yabin Zheng, 2011). Compared to the traditional rule-based approach, a statistical approach allows us to improve conversion quality more easily with the power of a large amount of data, e.g., Web data. However, language models and dictionaries which are generated automatically from the Web tend to be bigger than those of manually crafted rules, which makes it hard to execute IMEs on local computers.

The situation is the same in machine translation. Language model compression is critical in statistical machine translation. Several studies have been proposed in order to scale language model to large data. Example includes class-based models (Brown et al., 1992), entropy-based pruning (Stolcke, 1998), Golomb Coding (Church et al., 2007) 2007) and randomized lossy compression (Talbot and Brants, 2008). The main focus of this research is how efficiently the language model, especially n-gram model, can be compressed. However, in Japanese input methods, lexicon plays more important role in actual conversion than language model, since users don't always type Japanese text by sentence, but by phrase or even by word. Lexicon coverage is one of the key factors with which to evaluate the overall usability of IMEs. In addition, modern Japanese input methods need to support a variety of features, including auto-completion and reconversion, which accelerate input speeds as well as help users to edit what they typed before. It is non-trivial to implement a compact dictionary storage which supports these complicated use cases.

In this work, we propose novel lossless compression algorithms for both dictionary and language model based on succinct data structures. Although the size of our dictionary storage approaches closer to the information-theoretic lower bound, it supports three lookup operations: common prefix lookup, predictive lookup and reverse lookup. Proposed two data structures are used in our product "Google Japanese Input", and its open-source version "Mozc".

2 Statistical approach to input method editors

The model of statistical input method editor is basically the same as those of statistical machine translation. An input is converted according to the probability distribution $P(W|S)$, where W is target output and S is source user input characters (e.g. Hiragana sequence). The probability $P(W|S)$ is usually decomposed as a product of language model $P(W)$ and reading model

[1] http://www.google.com/intl/ja/ime/
[2] http://code.google.com/p/mozc/

Proceedings of the Workshop on Advances in Text Input Methods (WTIM 2011), pages 19–25,
Chiang Mai, Thailand, November 13, 2011.

$P(S|W)$, corresponding to the language model and translation model in statistical machine translation.

$$W_{opt} = \operatorname*{argmax}_W P(W|S) \quad (1)$$
$$= \operatorname*{argmax}_W P(W)P(S|W) \quad (2)$$

In Mozc, we use a class language model for representing $P(W)$ to reduce overall memory footprint. The class corresponds to the part of speech of Japanese.

$$P(W) = \prod_i P(w_i|c_i)P(c_i|c_{i-1}), \quad (3)$$

where c_i is a word class (part of speech) of w_i. If we assume that reading probabilities are mutually independent, $P(S|W)$ could be rewritten as

$$P(S|W) = \prod_i P(s_i|w_i), \quad (4)$$

where $P(s_i|w_i)$ is the conditional probability that a Japanese word w_i is typed as Japanese Hiragana s_i. This probability can be estimated from a tagged corpus and/or a manually crafted dictionary. By combining $P(W)$ and $P(S|W)$, W_{opt} can be rewritten as

$$W_{opt} = \operatorname*{argmax}_W \prod_i P(s_i|w_i)P(w_i|c_i)P(c_i|c_{i-1}). \quad (5)$$

The first two terms $P(s_i|w_i)P(w_i|c_i)$ are context independent. This part can be represented as a tuple $d = \langle s, w, c, cost \rangle$, where cost is $-\log(P(s|w)P(w|c))$. $P(c_i|c_{i-1})$ is a transition probability of the class language model. For our convenience, we call the set of dictionary entries d as *dictionary* and transition probability as *language model* in this paper. Dictionary and language model are compressed with different algorithms.

3 Dictionary compression

3.1 General setting of dictionary lookup

We describe the general setting and structure of dictionary for Japanese IME.

- Dictionary D is a set of dictionary entries d_i, e.g., $D = \{d_1, \ldots, d_n\}$
- Dictionary entry d is a tuple of reading, word, part-of-speech id and cost. Part-of-speech id and cost are 16 bit integers in Mozc.

To implement a Japanese IME, the dictionary storage had better to support the following three operations.

- Common Prefix Lookup
  Given a query s, returns all dictionary entries whose reading parts are prefix of s. This operation allows us to build a lattice (word graph) for user input in $O(n)$, where n is the length of user input. A lattice represents all candidate paths or all candidate sequences of output token, where each token denotes a word with its part of-speech.
- Predictive Lookup
  Given a query s, returns all dictionary entries which have s as a prefix of reading. Modern Japanese IMEs, especially IMEs for mobile devices, provide a suggestion feature which predicts a word or phrase that the user wants to type in without the user actually typing it in completely. Predictive lookup is used in implementation of the suggestion feature.
- Reverse Lookup
  Basically the same as Common Prefix Lookup and Predictive Lookup, but the direction of lookup is opposite. Reverse lookup retrieves readings from words. Commercial input methods implement a feature called reconversion with which user can re-convert sentences or phrases already submitted to applications. To support this, Reverse Lookup is necessary.

A *trie* is a useful data structure which supports common prefix and predictive lookup in constant time. Each node in the trie encodes a character, and paths from the root to the leaf represent a word. A trie encodes a set of words with a small footprint when they share common prefixes.

3.2 Double Array

In the initial implementation of Mozc, we had used the Double Array trie structure for dictionary storage (Aoe, 1989). Double array is known to be the fastest implementation for a trie and supports every operation described in the previous section. However, one critical issue of Double Array is its scalability. Since Double Array uses an explicit pointer to represent a node in a trie, it uses $O(n \log(n))$ space to store a trie with n nodes.

3.3 LOUDS

In Mozc, we use a succinct data structure LOUDS trie for compact dictionary representation (Jacobson, 1989). The main idea of LOUDS is that a trie is encoded in a succinct bit array string which doesn't use any pointers to represent nodes. LOUDS stores an n-node ordinal tree as a bit array of $2n + 1$ bits.

A LOUDS bit string is constructed as follows. Starting from the root nodes, we walk through a trie in breadth-first order. When seeing a node having d children ($d > 0$), d "1"s and one " 0" are emitted. In addition to that, "10" is added to the prefix of the bit string, which represents an imaginary super root node pointing to the root node. For example, three "1"s and one "0" are emitted when seeing the node "1" in Figure 1.

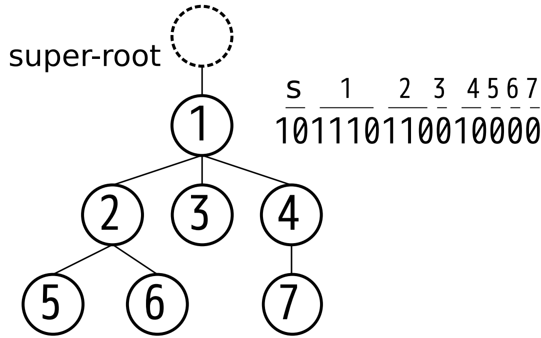

Figure 1: LOUDS trie representation

Navigation on the LOUDS trie is performed by rank and select operations on the bit array.

- $rank(k, i)$: Returns the number of $k \in \{0, 1\}$ bits to the left of, and including, position i.
- $select(k, i)$: Given an index i, returns the position of the ith $k \in \{0, 1\}$ bit in the bitstring.

Given a bit array of length k, rank and select can be executed in $O(1)$ time with $o(k)$ space. With rank and select operations, first child, next sibling and parent of m-th node can be computed as follows.

- $first\,child(m) = select(0, rank(1, m)) + 1$
- $next\,sibling(m) = m + 1$
- $parent(m) = select(1, 1 + rank(0, m))$

3.4 Space efficient dictionary data structure for Japanese IME

The key idea of Mozc's dictionary structure is that both readings and words are compressed in two independent LOUDS tries. This structure helps us not only to compress both readings and words by merging the common prefixes but to enable the reverse lookup required for Japanese IME. Dictionary entries associated with the pairs of reading and word are stored in a token array. A token stores part-of-speech id, cost and leaf node id associated with the word.

Figure 2 illustrates the dictionary data structure which encodes the dictionary entries shown in Table 1. Here we show how we perform forward lookup and reverse lookup on this dictionary.

Reading	Word	Leaf node id in reading trie	Leaf ndoe id in word trie
a	A	20	30
b	B	10	40
b	C	10	50

Table 1: Example of dictionary entries

Forward lookup (reading to word)

1. Given a query reading in Hiragana, retrieve a set of key node ids by traversing the reading trie.
2. By accessing the token array, obtain metadata of dictionary entries, e.g. POS and emission cost, and the set of word node ids.
3. By traversing the word trie from leaf to root, we can retrieve the word string.

Reverse lookup (word to reading)

1. Given a query word, retrieve a set of reading node ids by traversing the word trie
2. Unlike forward lookup, we cannot directly access the token array. Instead, we perform linear search over the token array to obtain the set of reading node ids.
3. By traversing the reading trie from leaf to root, we can retrieve the reading string.

Reverse lookup is generally slower than forward lookup because of linear search. This issue can easily be solved by caching the result of linear search. Since reconversion is only occasionally

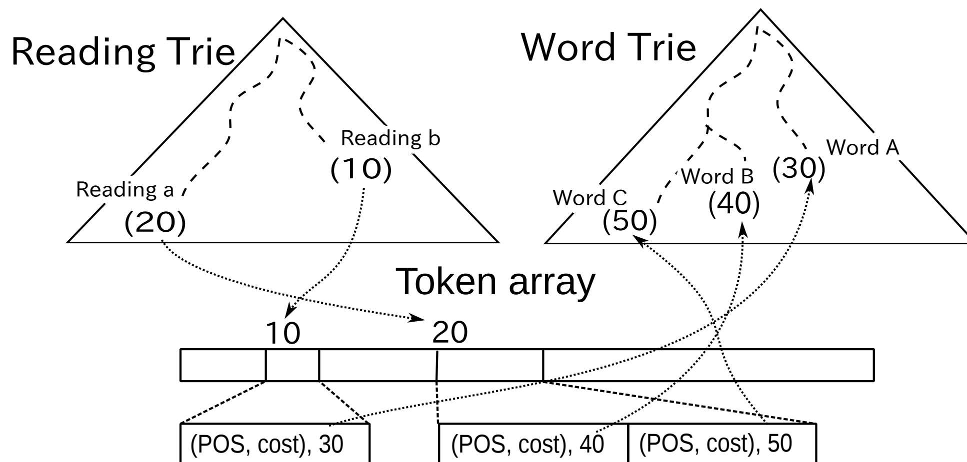

Figure 2: Dictionary structure

used, the cache is created on-the-fly when reconversion is requested to reduce the total amount of memory usage.

3.5 Additional heuristics for further compression

We combine the following three heuristics to perform further compression.

- String compression
  Mozc uses UTF-8 as an internal string representation, but not for storing the dictionary and language model. The reason is that it is not space-efficient to use UTF-8 directly, because UTF-8 encoding needs 3 bytes to encode Hiragana, Katakana and Kanji. Instead of UTF-8, we use a special encoding in the dictionary which encodes common Japanese characters, including Hiragana and Katakana, in 1 byte. With our new encoding, all Japanese characters are encoded in 1 or 2 byte.

- Token compression
  Part of speech distribution tends to be biased, since large portions of words are categorized as noun. By using shorter codes to represent frequent part-of-speech, we can compress the token arrays. With this compression, we have to encode the token array with variable length coding. For this purpose, we use a rx library [3], which also uses a succinct bit array structure.

- Katakana bit
  Converting Hiragana word to Katakana word is trivial in Japanese, as Hiragana and Katakana character have one-to-one mapping. We can remove all Katakana words from the word trie if a token have a Katakana bit. If a dictionary entry is a Hiragana to Katakana conversion, we set Katakana bit and do not insert the word in the word trie.

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

3.6 Experiments and evaluations

We compared our LOUDS based dictionary structure and three additional heuristics. Table 2 shows the total dictionary size and compression ratio against the plain text dictionary. LOUDS + Token is a LOUDS-based dictionary structure with token compression. LOUDS + Token/Katakana is a LOUDS-based dictionary with token compression and Katakana bit. LOUDS + all uses the all heuristics described in the previous section. Table 2 also shows the size of reading trie, word trie and token array in each dictionary.

The dictionary storage is reduced from 59.1MB to 20.5MB with LOUDS. On the other hand, space efficiency of Double Array trie is much worse than LOUDS. It uses about 80MB to encode all the words, which is larger than the original plain text. By combining three additional heuristics, the size of reading and word tries are drastically reduced. When using Katakana bit, the size of word trie is reduced since the Hiragana to Katakana entries are not registered into the word trie. The most effective heuristics for compressing the dictionary is string compression. Reading trie with string compression is about 40% the size of the original trie. Our succinct data structure encodes

Dictionary type	Size (Mbyte)	Size / word (byte)	detailed size (Mbyte)
plain text	59.1	46.0	
Double Array	80.8	63.0	
LOUDS+Token	20.5	16.0	Token: 8.5 Reading: 5.8 Word: 6.2
LOUDS+Token/Katakana	18.3	14.2	Token: 7.9 Reading: 5.8 Word: 4.6
LOUDS+all	13.3	10.4	Token: 7.9 Reading: 2.4 Word: 3.0

Table 2: Summary of dictionary compression

1,345,900 words in 13.3MByte (10.4 bytes per word) and supports common prefix, predictive and reverse lookup required for Japanese input methods.

4 Language model compression

4.1 Sparse matrix compression

As we described in the section 2, Mozc uses a class language model as base language model for conversion. The class basically corresponds to the part of speech of Japanese. Because a naive class language model is weak in capturing subtle grammatical differences, all Japanese function words (e.g. particles and auxiliary-verbs), frequent verbs, and frequent adjectives are lexicalized, where the lexical entry itself is assigned to unique class. With such an aggressive lexicalization, the number of classes amounts to 3000. One observation of the lexicalized transition probabilities is that the transition matrix becomes surprisingly sparse. More than 80% of transitions have 0 probability[4].

Several methods have been proposed for compressing a sparse matrix(Barrett et al., 1993). These algorithm are not optimal in terms of space, because they use a pointer to access to the indices. Our proposed data structure is based on succinct bit array which does not rely on any pointers.

Given a transition probabilities table $M[i, j]$, our succinct data structure is constructed as follows:

1. Converts the original two dimensional matrix into a one dimensional array with a linear conversion of the indices, e.g. $M[i, j] = A[size \cdot i + j]$, where $size$ is the number of classes.

2. Collects all probabilities and their indices having non-zero probability.

3. Represents the index in bit array.

4. Splits the bit array into sub bit arrays of size 3. Each split sub bits can represent 8 ($2^3 = 8$) different states.

5. Insert the split sub-bits into a tree structure. A node in the tree always has 8 children.

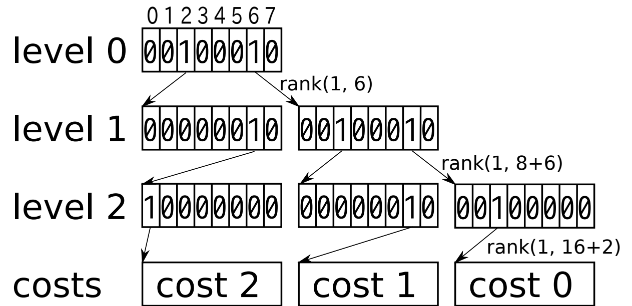

Figure 3: Succinct tree structure for class language model

Figure 3 shows how 1-dimensional indices 434, 406 and 176 are represented in a tree with 8 branches. Here we assume that indices 434, 406 and 176 have a transition log probability cost 0, cost1 and cost2 respectively. 434, 406 and 176 can be expressed as 110110010, 110010110, 010110000 in bit array. By splitting them into sub-arrays, they can be written as [6,6,2] (=110,110,010), [6,2,6] (=110,010,110), and [2,6,0] (=010,110,000). These sub arrays are inserted into a tree of depth 3. For example, when inserting the index 434 (6,6,2), the 6th bit of level 0 node, 6th bit of level 1 node, and 2nd bit of level 2 node are set to be 1. The leaf node stores the actual cost.

A key characteristic of this tree structure is that it is not necessary to store pointers (arrows in the Figure 3) from parent to children. If we were to create a bit array B_k by concatenating all nodes at level k, the child node of mth bit at level k is pointing to the $rank(1, m)$-th node at level $(k+1)$.

[4]Since Mozc's langauge model is generated from large amount of web data, we found that language model smoothing is not always necessary. We gave a reasonably big default cost to the transition having 0 probability.

4.2 Caching the transition matrix

One problem of our succinct tree structure for language model is that it requires huge numbers of bit operations in lookup. Our preliminary experiment showed that the lookup of matrix consumed about 50% of total decoding time (IME conversion time). To cope with this problem, we introduced a simple cache for transition matrix. Figure 4 shows a pseudocode of the cache. N is the size of cache.

```
argument: lid: class id of previous word
          rid: class id of current word
returns: cached cost (log probability)

N ← 1024  // cache size
size ←  number of classes

// initialization
cache_value[N] ← {−1, . . . , −1}
cache_index[N] ← {−1, . . . , −1}

function GetCachedTransitionCost(lid, rid)
begin
    // get the hash h from lid/rid.
    h ← (3 ∗ lid + rid) % N
    // i is the one dimensional index.
    i ← (lid · size + rid)
    if cache_index[h] ! = i then
        cache_index[h] ← i
        cache_value[h] ←
            GetRealTransitionCost(lid, rid)
    endif
    return cahce_value[h]
end
```

Figure 4: Pseudocode of cache

4.3 Experiments and evaluations

Table 3 shows the size of language model with different implementations. The size of class is 3019 and cost (transition probability) is encoded in 2 byte integer. We found that 1,272,002 probabilities are non-zero; i.e., around 86% elements have zero probability. If we use a naive two dimensional matrix, 18 Mbyte storage is required ($3019 \cdot 3019 \cdot 2 = 18,228,722$ byte). STL map requires more memory than baseline. Our proposed structure only uses 2.9 MB storage, where the succinct tree and cost tables consume 0.51 Mbyte and 2.4 Mbyte respectively. The "Random" is a theoretical lower bound of required storage, if we assume that the indices of non-zero probability are randomly selected. Our compact data structure is only 16% the size of the original matrix. Also, the size of our structure is close to the theoretical lower bound. The reason why the size becomes even smaller than the lower bound is that indices of non-zero probabilities are not actually randomly distributed in a real language model.

Table 4 shows the effect of the cache for the transition matrix. Even with a small cache size, the conversion speed is drastically improved. In practical, it is enough to set the cache size to be 512.

data structure	Size (Mbyte)
Two dimensional matrix	17.4
STL map	39.8
Succinct Tree (8-branches tree)	2.9
Random	3.1

Table 3: summary of language model compression

Cache size	conversion speed (sec/sentence)
0	0.0158
32	0.0115
128	0.0107
512	0.0102
1024	0.0099
4096	0.0097

Table 4: Cache size and conversion speed

5 Future work

This paper mainly focused on lossless algorithms. However, it would be possible to achieve more aggressive compression by introducing lossy compression. Furthermore, Mozc encodes all costs (log probabilities) in 2 byte integers. We would like to investigate how the compression of costs affects final conversion quality.

6 Conclusion

This paper presented novel lossless compression algorithms for dictionary and language model. Experimental results show that our succinct data structures drastically reduce space requirements by eschewing the space-consuming pointers used in traditional storage algorithms. In dictionary compression, we also proposed three methods to achieve further compression: string compression, token compression, and Katakana bit. In language

model compression, we showed that a naive cache algorithm can significantly improve lookup speed.

References

Junichi Aoe. 1989. An efficient digital search algorithm by using a double-array structure. *IEEE Trans. Softw. Eng.*, 15:1066–1077.

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. 1993. Templates for the solution of linear systems: Building blocks for iterative methods.

Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. 1992. Class-based n-gram models of natural language. *Computational Linguistics*, 18:467–479.

Zheng Chen and Kai-Fu Lee. 2000. A new statistical approach to chinese pinyin input. In *Proceedings of the ACL*, pages 241–247.

Kenneth Church, Ted Hart, , and Jianfeng Gao. 2007. Compressing trigram language models with golomb coding. In *In Proc. of EMNLP-CoNLL*.

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

Shinsuke Mori, Daisuke Takuma, and Gakuto Kurata. 2006. Phoneme-to-text transcription system with an infinite vocabulary. In *Proceedings of ACL*, ACL-44, pages 729–736.

Andreas Stolcke. 1998. Entropy-based pruning of back-off language models. In *In Proc. DARPA Broadcast News Transcription and Understanding Workshop*, page 270274.

David Talbot and Thorsten Brants. 2008. Randomized language models via perfect hash functions. In *In Proc. of ACL: HLT*.

Maosong Sun Yabin Zheng, Chen Li. 2011. Chime: An efficient error-tolerant chinese pinyin input method. In *In Proc. of IJCAI*.