System Paper

Design and Structure of The Juman++ Morphological Analyzer Toolkit

Arseny Tolmachev[†], Daisuke Kawahara[††] and Sadao Kurohashi[††]

An NLP tool is practical when it is fast in addition to having high accuracy. We describe the architecture and the used methods to achieve 250× analysis speed improvement on the Juman++ morphological analyzer together with slight accuracy improvements. This information should be useful for implementors of high-performance NLP and machine-learning based software.

Key Words: *Word Segmentation, Morphological Analysis, Software Optimization*

1 Introduction

Languages with a continuous script, like Japanese and Chinese, do not have natural word boundaries in most cases. Natural language processing for such languages requires to segment text into words. Segmentation is commonly done jointly with part of speech (**POS**) tagging and the whole process is usually referred to as Morphological Analysis (**MA**).

Modern morphological analyzers achieve high accuracy (a tokenwise segmentation F1 score of $> .99$ for Japanese) on established domains like newspaper texts. However, when using them on out-of-domain or open domain data (like web texts), the accuracy decreases, and it is difficult to improve that accuracy without creating costly annotations by trained experts.

To increase the overall analysis accuracy, Morita, Kawahara, and Kurohashi (2015) have proposed using a combination of a feature-rich linear model with a recurrent neural network-based language model (**RNNLM**) for morphological analysis and implemented it as the initial version of Juman++. The combined model considers semantic plausibility of segmentation, and because of this has drastically reduced the number of intolerable analysis errors, achieving the state-of-the-art analysis accuracy on Jumandic (the JUMAN dictionary and segmentation standard, Kurohashi and Kawahara (2012)) based corpora. In this paper, we refer to it as **Juman++ V1** or simply **V1**. Unfortunately, its execution speed was extremely slow, and that limits the practical usage of Juman++ V1.

[†] Fujitsu Laboratories, Ltd.
[††] Kyoto University, Graduate School of Informatics

We have developed a morphological analysis toolkit (Tolmachev, Kawahara, and Kurohashi 2018) consisting of three components: a morphological analyzer and two support tools which help with the development of analysis models. The analyzer is a complete rewrite of core ideas of Juman++ V1, released as Juman++ V2[1] (Tolmachev and Kurohashi 2018). In this paper we refer to it as **Juman++ V2**, **V2**, or simply **Juman++**. Our reimplementation is more than 250 times faster than V1, reaching the speed of traditional analyzers, at the same time achieving better accuracy than V1.

Juman++ follows the dictionary-based morphological analyzer design. During the analysis, it looks up word candidates using a dictionary and unknown word handlers, and then builds a lattice by connecting adjacent word candidates. The path through the lattice with the highest score becomes the analysis result.

The main goal of Juman++ is to achieve a reasonable analysis speed while being flexible and powerful at the same time. MeCab (Kudo, Yamamoto, and Matsumoto 2004; Kudo 2018) achieves its high analysis speed by precomputing lattice node connection features into a 2D table, referring only to the precomputed table at the analysis time. This makes it impossible for MeCab to use *fully* lexicalized bigram feature templates with *a large number of instances*. This design decision also makes it impossible for MeCab to use surface-based connection features. We must remark that partial lexicalization of bigram features is supported by MeCab and is frequently used, mostly for auxiliary words.

Juman++, on the other hand, supports such features for the sake of analysis accuracy. We achieve a reasonable analysis speed by designing the architecture of the analyzer, in a way that the modern CPUs could efficiently execute its compiled code. In order to achieve a high analysis speed, each stage of the analysis was designed for efficiency. For example, the dictionary structure and feature representation are designed without string-based operations in computations. For in-memory lattice representation, we focus on improving CPU cache efficiency. Feature-based path scoring is done with generating a model-specific C++ code which is organized to mask memory latency by asynchronously prefetching the model weights. We also perform the aggressive beam trimming and deduplicate the paths through the RNN which would have the same representation.

This paper presents an insight into Juman++ internal architecture, design, and rationale for making those design decisions. We discuss the internal data structures and inner workings of Juman++ V2. When compared to MeCab, which performs a single 2D array access and summation with a combined unigram score (2 operations), Juman++ does significantly more

[1] The analyzer is available at https://github.com/ku-nlp/jumanpp

computations: it evaluates about 60 features for each lattice node while performing the beam search while being only 5 times slower. We believe that the information on how to perform a large number of computations efficiently in machine learning-based software will be useful for the further development of natural language processing tools.

The paper is structured as follows. In section 2 we discuss the problem of morphological analysis and present approaches. Section 3 discusses the internals of Juman++ and explains our decisions. In Juman++ we heavily exploit capabilities of modern CPUs, some of which are described in Appendix A. Section 4 discusses the training of Juman++ models. Section 5 evaluates and compares Juman++ to different morphological analyzers in three categories: model size, analysis speed, and analysis accuracy. Finally, section 6 gives the discussion on the remaining problems of morphological analysis and Juman++.

2 Morphological Analysis Overview

The Japanese language has a continous script: there are no delimiting spaces. However, the applications usually expect the text to be segmented into tokens. An example of segmentation is shown in Figure 1. In most cases, there exist several different ways to segment a given sentence. Still, most of these ways produce nonsense segmentation from the language perspective.

There are two general types of approaches defining what these tokens are: supervised and unsupervised. In the supervised approaches, the tokens are usually defined to be morphemes, defined by segmentation standards and related human-annotated corpora. In unsupervised approaches, there is no such apriori knowledge for algorithms. The segmentation is decided mechanically by the algorithm, often to minimize some information-theoretic measure. We focus on supervised segmentation in this work.

2.1 Morphemes and Morphological Analysis Approach Types

Generally speaking, the definition of morpheme in Japanese morphological analysis differs from the definition in European languages, and the term *word* is ill-defined for Japanese (Murawaki

外国人参政権が認められていない
↓
外国|人|参政|権|が|認め|られて|い|ない

Fig. 1 An example of Japanese sentence segmentation into morphemes

2019). In this paper we use the terms *morpheme* and *word* interchangeably as a unit defined by a segmentation standard.

For Japanese, there are several popular standards like IPADic, Jumandic, and UniDic. Because the way to segment text into morphemes is defined by humans, it matches with the human expectation of what a word is. When an underlying application will be interacting with users on a word level, tokens being human-compatible is an essential requirement for a segmentation.

Most Japanese analyzers use segmentation dictionaries that define corpus segmentation standards. They usually have rich part of speech information attached and are human-curated. One focus of segmentation dictionaries is to be consistent: it should be possible to segment a sentence using the dictionary entries only in a single correct way. Such dictionaries are often maintained together with annotated corpora. On the other hand, Chinese-focused systems do not put much focus on dictionaries. The dictionary is created by taking every unique word from the corpus (Kruengkrai, Uchimoto, Kazama, Wang, Torisawa, and Isahara 2009; Zheng, Chen, and Xu 2013), in contrast to Japanese where the corpora are usually accompanied by a dictionary. Still, almost all approaches use rich feature templates or additional resources such as pretrained character n-gram or word embeddings, which increase the model size.

Unsupervised approaches (Uchiumi, Tsukahara, and Mochihashi 2015; Mochihashi, Yamada, and Ueda 2009; Zhikov, Takamura, and Okumura 2010), on the other hand, decide segmentation without external definitions. They do not require annotated corpora to work, but the resulting segmentation does not match with the human expectation of a word. Still, unsupervised segmentation achieved better accuracy on underlying tasks where user interaction is on a sentence or text-level, like machine translation (Kudo and Richardson 2018).

There exist two main lines of approaches to supervised Japanese morphological analysis: pointwise and search-based. Pointwise approaches make a segmentation decision for each character, usually based on the information from its surroundings. Search-based approaches look for a maximum scored interpretation in some structure over the input sentence.

2.2 Pointwise Approaches

Pointwise approaches make a segmentation decision independently for each position. They can be seen as a sequence tagging task. Such approaches are more popular for Chinese.

KyTea (Neubig, Nakata, and Mori 2011) is an example of this approach in Japanese. It makes a binary decision for each character: whether to insert a boundary before it or not. It can be seen as a sequence tagging with {B, I} tagset. POS tagging is done after inferring segmentation. The decisions are made by feature-based approaches, using characters, character n-grams, character

type information, and dictionary information as features. KyTea can use word features obtained from a dictionary. It checks whether the character sequence before and after the current character forms a word from the dictionary. It also checks whether the current word is inside a word. Neural networks were shown to be useful for Japanese in this paradigm as well (Kitagawa and Komachi 2018). They use character embeddings, character type embeddings, character n-gram embeddings, and tricks to incorporate dictionary information into the model. However, they only consider the segmentation task and do not do any tagging.

Tolmachev, Kawahara, and Kurohashi (2019) have used a bootstrapping approach to create a purely neural-network pointwise analyzer that uses only character unigrams as input data. The model seems to learn the dictionary information implicitly from a large scale automatically annotated corpus. They have achieved slightly better accuracy for both segmentation and POS tagging than the underlying analyzer used for tagging a large corpus while having small model size. In such an approach it is, however, difficult to make the analyzer to recognize new words. The word should be added to the underlying analyzer, and then the whole huge corpus needs to be retagged before re-learning the neural network model.

2.3 Search-based Approaches

Search-based approaches induce a structure over a sentence and perform a search over it. A most frequently used structure is a lattice (Jiang, Mi, and Liu 2008; Manning and Schütze 1999) which contains all possible segmentation tokens as nodes. The lattice nodes are connected at the token boundaries. The search then finds the highest scoring path through the lattice. Another branch of search-based approaches splits decisions into transitions (starting a new token and appending a character to the token) and searches for the highest scoring chain of transitions. This also can be seen as dynamically constructing a lattice while performing the search in it at the same time.

Lattice-based approaches are dominant for the Japanese language. Most of the time, the lattice is based on words that are present in a segmentation dictionary and a rule-based component for handling out-of-dictionary words. An example of a lattice is shown in Figure 2. Usually, there are no machine-learning components in lattice creation, but the scoring can be machine-learning based. We believe that the availability of high quality consistent morphological analysis dictionaries is the reason for that. Still, the work of Kaji and Kitsuregawa (2013) is a counterexample of a lattice-based approach for Japanese, where the lattice is created using a machine-learning approach.

Traditional lattice-based approaches for Japanese use mostly POS tags or other latent infor-

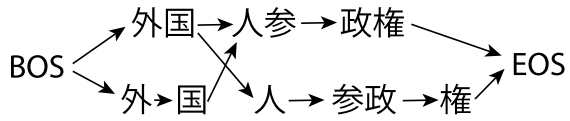

Fig. 2 Lattice for input sentence 外国人参政権

mation accessible from the dictionary to score paths through the lattice. JUMAN (Kurohashi 1994) is one of the first analyzers, which uses a hidden Markov model with manually-tuned weights for scoring. Lattice path scores are computed using connection weights for each pair of part of speech tags. ChaSen (Matsumoto, Takaoka, and Asahara 2008) is an improvement over JUMAN with adding automated weight tuning for the hidden Markov model (Asahara and Matsumoto 2000).

The most known and used morphological analyzer for Japanese is MeCab. Its ideas were reimplemented in a large number of analyzers like Kuromoji[2] and Sudachi (Takaoka, Hisamoto, Kawahara, Sakamoto, Uchida, and Matsumoto 2018). It builds a lattice and uses conditional random fields for learning the scoring function. MeCab is very fast: it can analyze almost 50k sentences per second while having good accuracy. The speed is realized by precomputing feature weights into a 2D array for bigram features and token weights for unigram and unknown node features. Unfortunately, precomputing the 2D array limits the kind and maximum variety of feature templates it is possible to use. For example, MeCab cannot use fully lexicalized features, and its bigram feature templates usually capture relations between POS tags. Partial lexicalization, usually of auxiliary words, is often used in MeCab features. When the bigram features start to contain a large number of diverse feature instances, the 2D array, and so the model itself, becomes very large. For example, the UniDic model for modern Japanese v2.3.0[3] (Den, Nakamura, Ogiso, and Ogura 2008) takes 5.5 GB because it uses many feature templates, which makes such an approach rather unwieldy.

Search-based models relying on POS-based features have problems when analyzing sequence of tokens with the same POS tag. One famous example is "外国人参政権", which has the correct segmentation "外国|人|参政|権", but the often produced segmentation is "外国|人参|政権". In a model with non-lexicalized bigram features, this is equivalent to making a decision whether P(外国|NN)P(人参|NN)P(政権|NN) is lower than P(外国|NN)P(人|NN)P(参政|NN)P(権|NN), where NN is the POS tag for common nouns. We believe that in general, if both sides contain

[2] https://www.atilika.org/
[3] https://unidic.ninjal.ac.jp/

words of similar frequency, it is undecidable. However, even lexical bigram information can help to solve most of such situations.

Another workaround to this problem is to register the whole sequence 外国人参政権 to the dictionary as a single super-token and handle such super-tokens in post-processing. JUMAN, ChaSen, and Sudachi support this approach. It can also be viewed as partial lexicalization. Still, this requires a lot of careful curating and super-token selection while not every possible situation is handleable is this way. Such super-tokens must also be not context-dependent. Context-dependent situations (e.g. "米原発" which can be segmented as "米原|発" or "米|原発") require surrounding information to produce the correct segmentation and can not be hard-coded.

3 Juman++ Internals

The logical subcomponents that contributed to the Juman++ speedup are the following:

(1) linear model score computation,
(2) beam search,
(3) RNN model.

While the last two subcomponents are rather small and self-contained, the first one is complex and forms the core of Juman++ design decisions: dictionary structure, feature computation hashing, code generation, and low level implementation details like prefetching and struct-of-arrays layout (described in the Appendix A.2) for the lattice. In this section we walk through the components in the order an input sentence goes through the analysis, starting with a general overview.

We provide the estimates of individual contributions to the overall speedup. Unfortunately, the rigorous experiment and comparison can not be performed because the components are highly intertwined and swapping one component for another means doing a large scale rewrite. Additionally, the development of Juman++ V2 did not happen in the order of the paper and it is difficult to use the revision history as a means for comparison.

3.1 Juman++ Analysis Overview

For an input sentence, the detailed description of Juman++ analysis steps is the following:

(1) Lookup dictionary-based lattice node candidates. This step uses the dictionary to emit triples of (token start, token end, entry pointer) for every dictionary candidate that can exist in the input string.
(2) Create unknown word node candidates. This stage also creates triples like the first stage.
(3) Build the lattice. This stage reads token information from the dictionary, computes token-

specific features and part of score which depends only on unigram features.

(4) Find top k highest scoring paths through the lattice using beam search. This stage computes the feature-based score for nodes.

(5) Perform RNN reranking of the top k paths.

(6) Output the analysis result.

The full morphological analysis score s for a sentence is a sum of per-node scores and is defined as

$$s \doteq \sum_{\text{nodes in lattice path}} s_l + \alpha(s_r + \beta), \tag{1}$$

where s_l is a *linear model* score for a lattice node, s_r is an *RNN model* score — non-normalized log-probabilities, α and β are scale and bias RNN hyperparamters. For computational efficiency, we use self-normalizing RNNLM which makes it possible to leave out the computation of the probability normalizer over the whole vocabulary. Still, the RNN scores are on the different range from that of the linear model. The RNN bias and scale parameters make the scores compatible with the linear model score. We also compute the RNN score only for paths with the top k linear model scores because the computational cost of applying RNN to the whole lattice is prohibitively high. The RNN itself is an optional component, its absence is equivalent to setting the scale $\alpha = 0$.

The linear score itself is defined as

$$s_l \doteq \sum_i \phi_i w_i, \tag{2}$$

where ϕ is an **n-gram feature** count vector and w is linear model weight vector. N-gram features are composed of **tokenwise features** using the templates defined in the spec. The presence of high-order n-gram features necessitates the usage of **beam search** to find the top-scoring path through the lattice. We must note that the current implementation of Juman++ does not use the RNN model during the beam search.

The training of Juman++ includes optimizing values of w and search for RNN-related hyperparameters. Juman++ can use both fully and partially annotated data for training of w.

Most of the computations at the analysis time happen during the path scoring. We adopt the struct-of-array layout of lattice-related data structures to improve the CPU cache efficiency of the entire process.

3.2 Juman++ Model

Juman++ by itself is only a toolkit for building morphological analyzers. The rules on how to perform the analysis are contained in a model, which is built from the following components:

(1) Analysis specification (**Spec**),
(2) Analysis dictionary,
(3) Training corpus,
(4) RNN model (optional).

Spec is a configuration for Juman++ instance, similar to feature templates used in other analyzers. It configures dictionary structure, feature templates, unknown word handling, and training loss. The documentation for the language and the configuration capabilities are available at Juman++ repository.[4] The dictionary contains the basic vocabulary the MA is going to recognize. For the analysis to work fast we need to *compile* the dictionary to the format which will enable fast lookup of dictionary information. We also have to train linear and RNN models to have the analysis result correct. The combination of these components forms a Juman++ analysis model. The model components and their interaction are shown in Figure 3.

3.3 Spec and Dictionary

In this subsection, we give a brief overview of the dictionary specification, and then we describe the dictionary compilation process logic, followed by the important implementation details.

3.3.1 Dictionary Specification

The first section of a spec defines which fields of a raw dictionary would be used for analysis and available for output. Based on that definition, we compile a raw dictionary into a representation that enables effective dictionary access at analysis time. An example of dictionary compilation is shown in Figure 4. Dictionary compilation has the following goals:

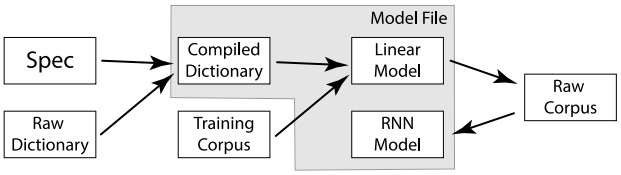

Fig. 3 Juman++ model file construction process

[4] https://github.com/ku-nlp/jumanpp/blob/master/docs/spec.md

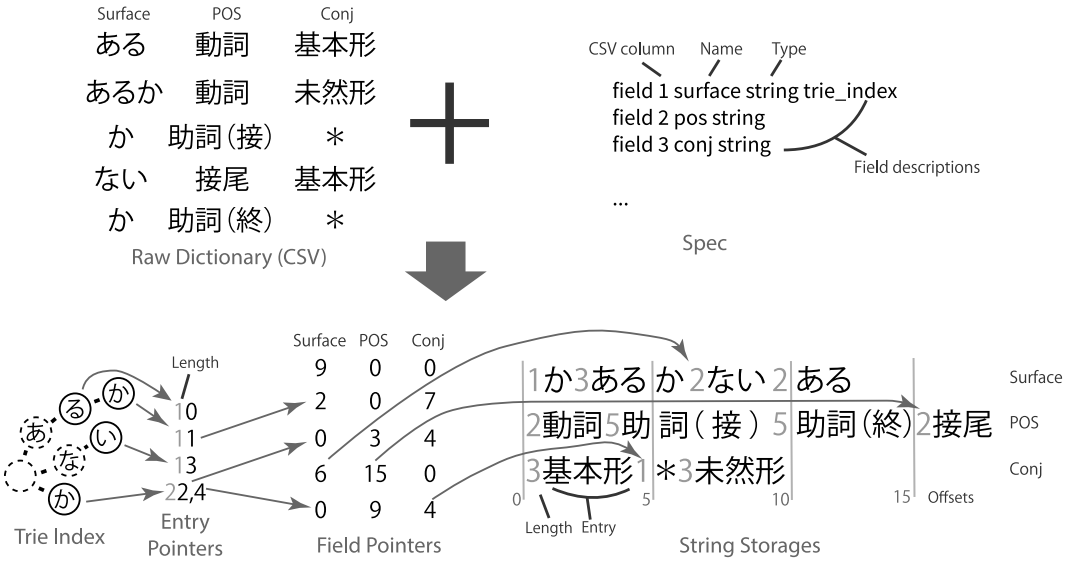

Fig. 4 Dictionary compilation. The actual information is black, explanations are gray. String and list lengths are blue. From the raw dictionary and spec field descriptions, we create compiled dictionary components: trie index, entry pointers, field pointers, and string storages. Trie index leaf nodes (non-leaf nodes are dashed here) point to a list of entry pointers (index keys can be duplicates). Each entry pointer refers to a row in the field pointer table, corresponding to a row in the raw dictionary. Field pointers refer to deduplicated column entries of the raw dictionary, packed in a column-specific string storage.

(1) During the lattice construction phase we need to look up the dictionary entries corresponding to surface string fragments. This process should be fast and not dependent on the dictionary size. Often, a trie-based approach is adopted for this objective.

(2) Handling strings as a sequence of characters or bytes is slow. On the other hand, feature computation for the lattice path scoring should be as fast as it can get. We would like to move as much computation as possible from the analysis step to the dictionary construction step. Our compiled dictionary design should allow us to treat all string entries in the dictionary as integers for the ease of further processing.

(3) Reducing the compiled dictionary size would also be beneficial.

(4) On-disk dictionary representation should be memory-mappable and should not require time-consuming processing logic at startup.

Raw dictionary is a RFC 4180 (Shafranovich 2005) CSV file. Each row corresponds to a single dictionary entry and columns correspond to the dictionary fields. Juman++ can use only a subset of columns from the raw dictionary. Namely, each of spec **field descriptions** selects a

single column from the dictionary.

Dictionary fields are typed. The handling of field data and its compiled representation depends on the field type. Juman++ supports four field types:

- Integers. 32-bit signed integer. While it is possible to use integers in a spec directly, this field type is used internally for storing values computed at dictionary compilation time.
- Strings. Strings are treated as categorical features: the strings themselves are replaced by integer pointers to the deduplicated values Most dictionary fields are usually strings.
- String lists. Dictionary entries can contain non-tabular information and this field type is useful for storing such information. The handling of individual strings is the same as the previous type, we also deduplicate the lists themselves.
- String key-value pair lists. Similar to the string list, but each entry holds two values.

3.3.2 Dictionary Compilation

For the compiled dictionary storage we adopt *column database* paradigm. We deduplicate string field contents and pack them, prefixed by their length, into **string storage**. Because the contents of such storage could be uniquely identified by their position inside the storage, we replace dictionary entries by 32-bit integer **field pointers** into respective string storages. This process is shown on Figure 4. For clarity, the figure treats the size of everything as 1. The actual implementation takes into the account variable-length UTF-8 string encoding and the integers are encoded using the variable-length LEB128 format (described later).

A spec must contain exactly one string field which would be used as a lookup key into the dictionary itself. Juman++ will build a double array trie index using the values of that field. There could be multiple entries with the same key. We handle that by pointing the trie index to the length-prefixed list of entry pointers.

By default, we use different string storages for different fields. However, Juman++ allows sharing string storages as well. For example, surface forms, readings, and dictionary forms usually contain the same entries. Sharing them greatly helps to reduce the size of the compiled dictionary.

3.3.3 Byte-level Storage and Alignment

Before this point, we have described the logical representation of the complied dictionary. However, the disk representation works with bytes and bits. Juman++ uses several techniques from search engines to store the dictionary on disk. Namely, we adopt variable length LEB128[5]

[5] https://en.wikipedia.org/wiki/LEB128

(little endian base-128) integer encoding (an example is shown in Table 1) for storing all integers (both pointers and lengths) in the compiled dictionary. With it, small-valued integers take a smaller number of bytes to store regardless of their original size. Reading small numbers in this representation is less computationally expensive as well. Furthermore, integers that are less than 127 are stored in a single byte having the same representation as lower 8 bits of the original 32-bit integer.

We exploit these facts about LEB128, which ultimately decrease compiled dictionary size and increase analysis speed. For example, we sort string storage contents by their frequency in the CSV file, in decreasing order. Because of this, more frequent strings will get closer to the beginning of the containing storage, meaning that they will have smaller pointer values, so they will effectively take less space in the field pointer table and could be decoded into the regular integers faster.

Juman++ also allows customizing **alignment** of string storage. By default, length-prefixed stored strings can start at any position. With the alignment of z, those starts can only be multiple of 2^z. Default alignment means $z = 0$. For aligned fields, the lower z bits of field pointers will always be zero and can be ignored. Thus, we bit-shift the pointers to the right by z and store only upper $32 - z$ bits as on-disk field pointer representation. An example of applying alignment to a string storage is shown in Figure 5. Aligning string storage makes all respective field pointers smaller, reaping the benefits of LEB128. Moreover, if a field contains only a limited and small set of values (e.g. POS tags), then choosing big enough alignment will guarantee that pointers for that field are always going to be stored in a single byte.

Alignment is a trade-off between using storage for actual strings or field pointers. At the same time, making the field pointer table smaller decreases the amount of memory to read during the analysis, increasing the probability of cache hits, resulting in ∼15% improved analysis speed compared to zero alignment in our Jumandic experiments. Juman++ includes a development

Table 1 An LEB128 encoding example.

Decimal	105			624485
Binary	1100101		10011000011101100101	
Split into 7-bit groups	1100101	0100110	0001110	1100101
Prepend 1 to non-last groups	01100101	00100110	10001110	11100101
In hexadecimal	0x65	0x26	0x8E	0xE5
On disk	0x65		0xE5	0x8E 0x26

The processing flows from the top to the bottom.

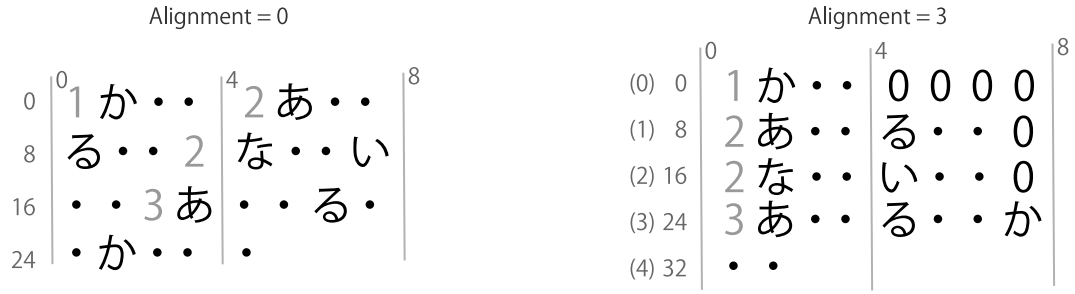

Fig. 5 String storage alignment example. Dots are additional bytes used by UTF-8 encoding for the respective symbols. Storage with alignment=3 can be indexed using the numbers in brackets.

tool that estimates changes in string storage size versus field pointer table size when changing the alignment for a chosen string storage to help users with making the correct decision.

When storing sequences of integers to the disk when their ordering does not matter (or they are already sorted), we store them in **delta encoding**. Instead of storing a sequence $a_1, a_2, a_3, ...$, $\forall j > i : a_j > a_i$ we store the differences between the successive elements: $a_1, a_2 - a_1, a_3 - a_2, ...$ Differences $a_{i+1} - a_i$ have smaller values than values a_i themselves, once more reaping the benefits of LEB128 encoding. We apply the delta encoding on top of LEB128 for entry pointer lists (which are already sorted), string list-typed fields and key parts of key-value list-typed fields (which we sort).

3.3.4 Representation Limitations

Our compiled dictionary representation uses signed 32-bit integers as pointers and because of that suffers from some limitations. Namely, in the current design, individual sizes of entry pointers, field pointers or string storage tables cannot exceed $2^{31} - 1$ bytes.

However, this does not mean that the dictionary size cannot be more than $2^{31} - 1$ bytes. For string fields, Juman++ cannot handle more than $2^{31} - 1$ bytes of *unique* string values for each field. In the case of Jumandic, a shared string storage for surface, the base form, and reading fields contain 1.7 M entries, taking 40 MB of space, less than 2% of the total limit. Because the size scales with unique entries, in practice the limit is not significant.

The field pointer table scales proportionally to the number of entries in the dictionary and can become a limit for truly enormous dictionaries. For the reference, Jumandic contains 1.8 M entries and its field pointer table takes only 32 MB, giving approximately 18 bytes per entry. It is difficult to give exact numbers on the maximum possible number of dictionary entries, but we believe that the current implementation can handle dictionaries with up to 100 M entries. It is

possible to align the field pointer table as well to step over the current restrictions, though field pointer table alignment is not implemented currently.

3.4 Features

Feature computation is a cornerstone for the analysis speed of Juman++. We use hashing for combining the values from the data into the final feature values. By arranging the computation order, we achieve a large reduction of duplicated calculations. Finally, we describe our reasons and gains from using code generation for feature computation and scoring.

3.4.1 Feature Layers

In Juman++ we ideologically distinguish features into three layers. The features of the top layers are computed using the values of the previous layer.

- Primitive, e.g. field values or other token-specific information. `pos` and `next_char` from Figure 6 are primitive features. They are represented by an unsigned 32-bit value.
- Tokenwise, namely a combination of one or more primitive features inside a single token. `[pos, next_char]`, `[pos]` and `[aux_word]` are tokenwise features. They are represented by an unsigned 64-bit value.
- N-gram, which is a combination of tokenwise features over different tokens.

Primitive features are defined for the current token, and come from two classes. The first one comes directly from the dictionary: they are values of integer and string-typed dictionary fields. For example, a part-of-speech or a surface string representation would be such a primitive feature. Primitive values of the second type either come from the analyzer input (e.g. characters near the current word) or are computed from the dictionary values (e.g. length of a surface field).

```
field 1 surface string trie_index
field 2 pos string

feature next_char = codepoint 1
feature aux_word = match [pos] with "助詞" then [surface, pos] else [pos]

ngram [pos]                      # No1
ngram [pos, next_char][pos]      # No2
ngram [aux_word][pos, next_char] # No3
ngram [pos][surface, pos][pos]   # No4
```

Fig. 6 An example of Juman++ spec with feature definitions

The full list of supported primitive features is available in spec format description documentation.

Juman++ has also support for **partial lexicalization**. See the *aux_word* feature definition in Figure 6. *Match* primitive feature is similar to an *if* statement. The feature itself is expanded in one of two lists, depending on the condition. The condition can match a list of fields to a list of values and is computed at *dictionary compile* time to save the computations at analysis time. Conditions for multiple *match* features are packed as distinct bits to an automatically generated integer-typed dictionary field.

Tokenwise features combine primitive features within a single token. We distinguish them because n-gram features can contain several identical combinations of primitive features. For example, Figure 6, n-grams 1, 2, 3 share the `[pos, next_char]` feature combination. We compute a 64-bit integer representation for each such combination. The value tf_t of a tokenwise feature t is computed as

$$tf_t = hash(t, n, seed_{tok}, pf_1, pf_2, ..., pf_n),$$

where n is the total count of primitive feature components of this tokenwise feature, $seed_{tok}$ is a hash seed value for tokenwise features and pf_i are respective primitive feature values. The hash function is discussed later.

N-gram features combine tokenwise features between several n-gram components into a 64-bit integer. The formula is similar to the one for tokenwise features.

$$nf_j = hash(j, n, seed_{ngram}, tf_0, tf_{-1}, ..., tf_{-n+1}),$$

where j is an index of n-gram feature, n is the order of the n-gram, $seed_{ngram}$ is seed value, tf_k is a value of tokenwise feature in the current path relative to the current position. tf_0 would be the value for the current token, tf_{-1} would point to the previous token, and so on. This enables us to get rid of identical computations.

3.4.2 Hashing

Because Juman++ uses hashing for computing features, we have created a hash function for the task. Usually, the hash functions are designed for hashing strings, operating on sequences of bytes, but it is not the case for Juman++. Instead, it operates on 64-bit integer values, producing a 64-bit result. The basic design of the function is that it takes its current state and a parameter and produces the next state. Hashing multiple values is done by applying the hashing function

in sequence:

$$hash(seed, a, b, c, ...) = \textsc{JppHash}(\textsc{JppHash}(\textsc{JppHash}(\textsc{JppHash}(seed, a), b), c), ...).$$

We adopt a lightweight hash function based on PCG (O'Neill 2014) algorithm for generating pseudorandom numbers, which itself is based on linear congruential generators. The function is shown on Figure 7. The total computations consist of two **XOR** operations, one 64-bit multiplication and the right shift. The right shift and the second **XOR** operations improve the randomness of lowest bits, which are used for accessing feature weights. Please refer to the PCG paper for the details.

3.4.3 Computation Order

As can be seen, tokenwise feature components of n-grams are consumed from the end to the beginning. This allows us to cut the total number of computations by performing identical parts of computations only once and achieve better cache locality. Figure 8 shows an example of a lattice. The node ない has two inbound paths, let us denote their last trigrams as ある-か-ない and BOS-あるか-ない.

While the overall scores for these paths should be different, they share some features. Namely, unigram features for ない are identical and that part of the score should be identical as well. Additionally, the parts of bigram and trigram hash states are identical too. Let's consider trigram

procedure JppHash($state, input$)
  $v \leftarrow state \oplus input$
  $v \leftarrow v \cdot$ `0x6eed0e9da4d94a4f`
  $t \leftarrow$ ShiftRight(v, 32)
  $v \leftarrow v \oplus t$
  return v
end procedure

Fig. 7 Juman++ hashing algorithm. It operates on 64-bit unsigned integers. $\oplus$ denotes a **XOR** operation.

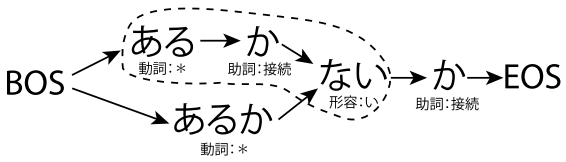

Fig. 8 Nodes used for the computation of a trigram features of the upper path at the node ない

features. $hash(j, 3, seed_{ngram}, tf_{ない}, tf_{か}, tf_{ある})$ and $hash(j, 3, seed_{ngram}, tf_{ない}, tf_{あるか}, tf_{\mathrm{BOS}})$ share prefixes until $tf_{ない}$ and thus will result in identical computations when computed naively. In contrast to that, we break the computation of n-gram features into steps, memorizing intermediate hash function state and sharing it between n-gram features with identical prefixes.

The order in which we perform feature and score computation at first glance is the following:

(1) Compute primitive features.
(2) Compute tokenwise features.
(3) Compute 1-gram features and respective parts of 2,3-grams.
(4) Compute score components which are provided by 1-gram features.
(5) Finish computing 2-gram features, respective score components and respective pars of 3-gram features.
(6) Finish computing 3-gram features and scores.

We compute scores for all nodes starting at a particular character boundary at the same time. Because of the struct-of-arrays layout for lattice structures, feature values for nodes of a boundary are stored in a single array. When performing computations in the specified order, we minimize the number of jumps between boundaries and have very high data locality in computations.

3.4.4 Code Generation

Juman++ has two versions of feature computations: *dynamic*, focusing on the clarity and correctness of the implementation and *static*, focusing on the execution speed. The dynamic implementation models the feature computation process using the object-oriented paradigm. Each of the feature kinds (primitive, tokenwise and ngrams) is modeled as an interface with corresponding implementations using the C++ inheritance and polymorphism. Because the polymorphism uses the virtual function dispatch, this results in a long sequence of indirect function calls, with each function being very short. A long sequence of indirect calls from a single call site, even non-changing between iterations, strains the branch predictor and can become a sequence of branch mispredictions resulting in a large overhead over the actual feature computation costs.

To achieve high performance, we provide static feature computation: a way to generate C++ code[6] which implements feature value and score computations inline, as defined by a spec. The code generation pursues the following main goals:

(1) Enable the compiler to perform the feature-specific optimizations.
(2) Exploit asynchronous and out-of-order nature of modern CPUs to mask the cost of random

[6] An example of generated code for Jumandic can be seen at https://git.io/fjvpy

feature accesses.

(3) Remove the overhead caused by indirect function calls.

While emitting the C++ code for feature computation and scoring automatically solves the third goal, the ways to achieve the first two and effects from them are discussed in more depth.

Enabling Compiler Optimizations

Our computation of tokenwise and n-gram features contain constants as the initial arguments to the hashing process. In the case of static feature computation, it opens the possibility for the compiler to perform the constant folding optimization, reducing the number of computations we need to perform at the runtime. An early V2 prototype performed the feature computations in the opposite order: constants after the values from the lattice nodes. By ordering the constants first we observed a significant decrease in analysis speed because of constant folding optimizations. While the constant folding was probably the most easily observable effect of compiler optimizations, inlining the logic also allows the compiler to perform other optimizations, for example, reordering code to improve register utilization.

Interleave Feature Computations and Memory Access

Because we use hashing for computing features, the model weight access pattern becomes random. Random access means that every data access is a cache miss with high probability. Fortunately, modern CPUs provide ways to **prefetch** data from memory to cache. User-controlled prefetching works by issuing special instructions, which are a no-op on the architectural level (so the computation result is the same), but communicate a hint to the CPU that the pointed memory is going to be accessed soon. Code-generation allows us to effectively exploit prefetching to effectively remove cache misses, or reduce their effect, during the score computation. We also take into account the out-of-order execution capabilities of modern CPUs by splitting the overall logic into multiple dependency chains.

We generate code using the skeleton shown in Figure 9. The main idea is to create a temporal delay between the computation of the actual feature value $f(n)$ and its usage for accumulating the weight contribution to the linear model score s_n. We accumulate the weight for the *previous* lattice node inside the boundary $n-1$, while computing the features for the current node n. Our implementation stores n-gram feature values in two buffers: one for the previous lattice node and one for the current one. Reordering code and utilizing prefetching gave about $2\times$ speedup over the prototype without these features.

We also localize the tokenwise feature access when computing n-gram features. Namely, the

tokenwise features are stored with the order shown in Figure 10. The numbers denote the rank of n-gram features that use the tokenwise features. Because the computation of bi-gram and tri-gram features happen separately, this ordering results in reduced CPU cache pressure.

In the case of unigrams, we additionally combine and interleave the first four steps of feature and score computation (primitive, tokenwise, n-gram feature components and scoring, see 3.4.3). This results in a larger temporal delay between the n-gram feature weight prefetch operation and the actual weight access and reduces the time an access operation needs to wait for the weight in the CPU reorder buffer. The generated code can be seen in the example at https://git.io/fjvpy from the line 944. The generated code for the first feature with annotations is shown in Figure 11.

Finally, interleaving opens a door for reducing the number of tokenwise features we need to store. In a spec there usually exist tokenwise features that are used only by unigram features. In the Jumandic-based model, about 30% of tokenwise features are like that. Such tokenwise features will not be used by *other nodes* in n-gram features, so we can skip storing them for the current node (when using code-generated scoring and feature computation logic). Their values never leave CPU registers, so this reduces overall CPU data cache pressure and analysis memory usage.

Our computations perform the score computations in floating point without quantizing the weights like MeCab. Because weight access is explicitly asynchronous in Juman++, the registers[7]

for node n at boundary **do**
  $s_{n-1} \leftarrow 0$
  for feature f in active ngram features **do**
    compute value of $f(n)$
    $s_{n-1} \leftarrow s_{n-1} + w_{f(n-1)}$
    prefetch $w_{f(n)}$
  end for
end for

Fig. 9 Skeleton of code-generated kernels

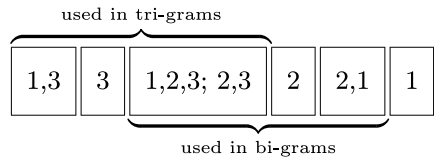

Fig. 10 Ordering of stored tokenwise features

[7] We are talking about microarchitectural physical registers instead of architectural logical registers in this

```
// pattern feature #8 (with unigram), usage=3
/* usage is a bitmap, this feature used by 1-gram (1) and 2-gram (2) */
constexpr jumanpp::core::features::impl::CopyPrimFeatureImpl pfobj_surface_{0};
/* constexpr forces the object to be constructed at the compile time. It is either
   completely optimized away or stored in the constant section of the binary. */
::jumanpp::u64 pf_surface_0 = pfobj_surface_.access(ctx, nodeInfo, entry);
/* primitive feature access */
auto fe_pat_hash_8 = ::jumanpp::util::hashing::FastHash1{}
  .mix(8ULL).mix(1ULL).mix(34359656763621376ULL);
/* Prefix of tokenwise feature: index, number of components and seed value */
fe_pat_hash_8 = fe_pat_hash_8.mix(pf_surface_0);
/* Mixing in primitive feature to the hash state */
::jumanpp::u64 fe_pat_8 = fe_pat_hash_8.result();
/* Finalizing tokenwise feature */
constexpr ::jumanpp::core::features::impl::UnigramFeature fng_uni_0_{0, 0, 8};
/* Instantiating the 1-gram (surface) feature.
   Number in the name of the n-gram feature corresponds to the order in the spec. */
auto value_fng_uni_0_ = fng_uni_0_.maskedValueFor(fe_pat_8, mask);
/* Instantiating n-gram feature */
float score_part_0 = weights.at(buf2.at(0)); // perceptron op
/* Computing linear model score component. The weight should be prefetched. */
weights.prefetch<::jumanpp::util::PrefetchHint::PREFETCH_HINT_T0>(value_fng_uni_0_);
/* Prefetching operation */
buf1.at(0) = value_fng_uni_0_;
/* Storing the computed n-gram feature value to a temporary buffer. */
patterns.at(8) = fe_pat_8;
/* storing the tokenwise feature value because it is used by 2-grams */
```

Fig. 11 Annotated code of interleaved computation. Annotations are in `/* comment blocks */`.

which are used for weights would be stored for a long time in the CPU reorder buffer. At the same time, feature hashing has rather high register pressure and high throughput because of high cache utilization, we would like those registers to be quickly reused. Modern CPU architectures often have two sets of physical registers: for integers and for floating points. Using floating point for weights allows the CPU to efficiently re-utilize integer registers during the hashing while keeping the weights in the floating point registers.

paragraph.For example, Intel Skylake microarchitecture has 180 integer physical registers and 168 floating point registers, but only 16 logical integer scalar registers and 16 vector registers (they can be both integer and floating point).See https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client) for details.

3.5 Unknown Word Handling

Dictionary-based morphological analyzers cannot contain every possible word in their dictionary. Instead, they devise means to handle unknown words. Usually, unknown words are handled with rules, e.g. grouping characters of the same type and using those entries as regular lattice nodes. The lattice search algorithm would then treat the unknown nodes in the same manner as the dictionary nodes for scoring purposes.

Juman++ follows the rule-based approach. The exact rules are defined in the spec using rule constructors. We implement five types of constructors:

(1) A single character of a specified type. This type of rules is used for not defined symbols. It is a good idea to have a catch-all rule of this type in the model because otherwise, Juman++ could fail to build a lattice for some input strings.

(2) A character sequence sharing the same character type. This is the most used type of rule for defining "regular" unknown words.

(3) A number. This constructor handles numbers, written with Kanji in literal (一万五千三百五十二) and numeric (一五三五二) styles and digits (15324). Kanji numbers in numeric styles can be separated in groups of three digits with the centered dot (一五・三五二) and digit numbers can be separated with commas (15,324).

(4) An onomatopoeia pattern (Sasano, Kurohashi, and Okumura 2013). We define the patterns as ABAB, ABCABC, and ABCDABCD where A, B, C, D are characters of the same and specified character class.

(5) A normalized dictionary entry. Following Sasano et al. (2013), we also add nodes which can be produced from the dictionary by inserting repeated characters (痛い→痛いいい), prolongation (はい→は〜い) or small "tsu" (かたい→かったい).

Primitive Feature Handling

To be able to score unknown nodes in the same manner as dictionary nodes, unknown word handlers need to provide the same primitive features as there exist for dictionary-based lattice nodes. The first four constructors use dictionary patterns: special dictionary entries that are not indexed for surface-lookup. The primitive features for unknown nodes, created by these handlers, are initialized from dictionary patterns and then a subset of primitive features, configured in the Spec, is replaced by a hash value of a surface string corresponding to the unknown lattice node. In the Jumandic we replace surface, the base form, and reading with the actual surface string of unknown handler. In the case of normalization, we use the dictionary entry which acts as the normalization result as the pattern, keeping the rest of handling the same.

Handlers also can communicate with the scoring procedure by setting values of *placeholder* primitive features. For example, the character sequence handler would set 1 to the placeholder in the case if the current unknown word contains a dictionary word as a prefix.

3.6 Beam Search and Trimming

Searching for the top-scored path in the lattice is done on a character boundary basis. For each boundary there are *left* lattice nodes, which contain words ending at the current boundary and *right* nodes, which contain words starting from the current boundary. Left nodes also contain paths ending at those nodes. A search step grows those paths, adding connections that cross the boundary between left and right nodes. The process is illustrated on Figure 12.

Because Juman++ uses trigram features, the usually employed Viterbi search will have $O(n^3)$ computational complexity on each segmentation boundary, which will completely overturn all benefits we gain from optimization. We employ the beam search instead. V1 uses node-local beams of width b_f, meaning that it keeps b_f paths with the top scores are kept for each lattice node. This configuration will be referred to as *full beam*.

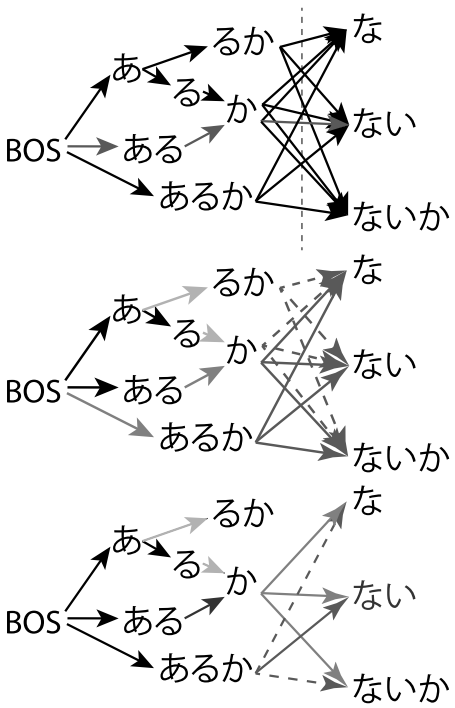

Fig. 12 A step of path search at a character boundary. Top: full beam, middle: left trim ($b_l = 2$), bottom: right trim ($m_l = 1$, $m_r = 1$). Trimmed connections are dashed.

Full beam considers every combination of *left* (ending on the boundary) and *right* (starts from the boundary) nodes on each character boundary. For each left node, it also considers the paths in the respective beam. This setup conservatively assumes that it is impossible to compare the scores of paths which end at different nodes, and has high computational cost. Still, in our experiments, most of the sentences contain several boundaries where there exist around 20–30 of both left and right nodes. In the full beam setup the computational complexity scales as the product of a number of left and right nodes at a character boundary. Moreover, the majority of the combinations are useless and can never be the correct analysis result.

To increase the analysis speed we have implemented more aggressive beam trimming. The first improvement is straightforward: instead of using all paths from all the left nodes, we form a global beam of width b_l for the boundary. The path scoring process can use the formed global beam instead of all the local beams. In this setting, the path scores become comparable for the identical sequences of surface characters.

Using the global beam for the left side of the boundary greatly helps to reduce the number of computations during the scoring. The number of right nodes can still be large. We further decrease the number of computations by ranking the right nodes and performing the full computations only to the perspective nodes. Namely, we rank the right nodes by evaluating their scores in the connection to top m_l paths from the left global beam. Then we evaluate the scores for the remaining paths, but we use only top-ranked m_r right nodes.

The full algorithm for this setting (which we refer to as *trimmed beam*) is:

(1) Form the *global beam* of width b_l from the local beams of left nodes (trimmed beam: left)
(2) Rank the right nodes using the top $m_l (< b_l)$ path candidates from the global beam (trimmed beam: rank)
(3) Compute the remaining path candidates over the boundary using the remaining $b_l - m_l$ paths of the global beam and top m_r best right nodes (trimmed beam: right)
(4) Form the local beams of width b_f for the right nodes using the results of steps 2 and 3

In our experiments, the trimmed beam configuration showed the same accuracy while having a significantly faster analysis speed. Moreover, it was sufficient to rank the right nodes using only $m_r = 1$ left paths. The accuracy results are discussed in detail in subsection 5.4. We also discuss the setting of analyzing out-of-corpus data, which had a larger number of situations when the trimmed beam result was different from the full beam.

3.7 RNN Language Model

An RNN language model helps the analyzer to be more accurate by adding additional lexicalization learned from a large-scale corpus. Juman++ uses a recurrent neural network-based language model (Mikolov 2012) to estimate semantic plausibility of different segmentation possibilities. Still, a complete lattice search with RNN is computationally expensive and is one of the reasons for the slow analysis speed of V1. Instead, V2 uses the RNN model only to rerank analysis candidates which remain in the beam of EOS node after beam search with the linear model.

Juman++ can evaluate RNN not only on the surface string but on any combination of dictionary string fields. We will denote such a combination as **RNN key**. For the Jumandic RNN key, we use a combination of word base form (without conjugations) with a rough part of speech tag.

Remember that Juman++ lattice nodes are unique for a combination of primitive features participating in n-gram computations. The RNN model is evaluated only on a subset of those primitive features, which mean that paths traversing through the distinct lattice nodes may have identical RNN key. Naively evaluating all paths lead to unnecessary computations, which decrease overall analysis speed. Because of this, we deduplicate possible RNN paths before performing the actual computations. This greatly reduces the number of computations for the RNN language model.

For out-of-dictionary keys of the RNN model, we use a length-based unknown word penalty instead of the raw RNN scores. Namely, the score is defined as

$$s_r = u_b + l \cdot u_l, \tag{3}$$

where u_b is a constant part of unknown RNN score hyperparameter, l is count of Unicode codepoints of in the surface of the node with out-of-dictionary RNN key and u_l is a length part of unknown RNN score hyperparameter.

Because the RNN model is independent of the analysis dictionary, the RNN representation lookup differs from the segmentation dictionary lookup. We build two separate indices for the RNN. The first one contains the concatenated field pointers in LEB128 encoding as the key and embedding id as the value. It is used to handle normal lattice nodes which are completely built from the dictionary. The second one uses the normal strings for the fields which unknown word handlers replace by surface and field pointers for the remaining fields. It is used to handle lattice nodes, which are contained in the language model but not in the segmentation dictionary.

4 Training

The training of the linear and RNN models for Juman++ is done independently. The parameters for mixing them are optimized after training both, using a hyperparameter search. We also use the hyperparameter search to optimize the training procedure for the linear model.

Juman++ uses a slightly peculiar data scheduling regime for the training. The full training process is divided into epochs, each of which iterates several times over the training data. The training data is also shuffled before the start of each epoch. The first iteration of each epoch uses the full beam for the analysis and the following iterations use the trimmed beam search. This scheduling improves the analysis stability for out-of-domain data.

Additionally, to reduce the model pollution with features corresponding to bad nodes, the first epoch uses only fully-annotated examples. The partially annotated examples are added to the training only from the second epoch when the model learns to rank perspective nodes higher than bad ones.

4.1 Linear Model

Juman++ can train the linear model using both fully-annotated and partially-annotated data. Data in the fully-annotated format define a sentence as a sequence of lattice nodes. Each node must contain all dictionary fields used in n-gram features.

Partially-annotated data contain sentences as sequences of characters with additional annotations. Annotations can be of three types: segment, no-segment, and word. Segment annotation forces a boundary between two Unicode codepoints. No-segment annotation is an opposite—it forces *an absence* of a boundary between two codepoints. Word annotation has two segment annotations at the beginning and the end of the word, no-segment annotations between them and optional tags that correspond to required values of dictionary fields for matching nodes.

Juman++ uses the Soft Confident Weighted (Wang, Zhao, and Hoi 2016) online learning algorithm to optimize model weights. For the fully-annotated examples the application is straightforward: we use the combination of all n-gram features in the path as sentence features. We update the model for the sentences where the correct analysis, as specified by the training example, falls off the beam.

We have also tried two early update strategies known to be useful for structured learning: maximum violation and beam falloff when the model is updated only for the subset of parameters starting from the beginning of the sentence and ending at the point of maximum violation (difference between top beam score and gold score) or where the correct analysis falls off the beam.

In our experiments, however, the partial update strategies showed worse accuracy than the full update strategy.

In the case of partially annotated examples, we only update features contained in partial violations: codepoint boundaries where the top-scored lattice node does not adhere to the partial annotation rules. As the correct node, we try to find several node candidates that satisfy the partial annotation rules, deduplicating features from them so that the features shared by multiple lattice nodes won't be overweighted.

4.2 RNN training

In the current implementation of Juman++, the RNN model is trained completely independently of the linear model. We use the faster-rnnlm software[8] for the RNN model training. It is first trained on a large scale automatically analyzed data and then fine-tuned on the human-segmented data. The resulting model is then used for the hyperparameter tuning.

4.3 Hyperparameters Tuning

For the training of Jumandic-based Juman++ models we optimize the following hyperparameters:

- Soft Confident Weighted learning hyperparameters
- Number of training epochs and in-epoch iterations
- RNN mixing parameters
- RNN unknown word hyperparameters

We use the Gaussian process-based Spearmint (Snoek, Larochelle, and Adams 2012) package for the hyperparameter tuning. The search is performed on the average of F1 scores for segmentation and POS-tagging over the 10-fold cross-validation on the training data. We have optimized hyperparameters in two groups: linear model training-related parameters and RNN-related parameters.

5 Experiments

The overall performance of a morphological analyzer consists of different factors. We compare the following performance components:

- Dictionary or model size,

[8] https://github.com/yandex/faster-rnnlm

- Analysis speed,
- Analysis accuracy.

For comparison, we use JUMAN, MeCab, and KyTea as our baselines.

In all the experiments we use the Jumandic segmentation dictionary with Kyoto University (KU) and Kyoto University Web Document Leads (KWDLC) corpora. JUMAN uses the Jumandic without expanded conjugation forms as it is, for other analyzers, we expand all possible conjugation forms with conjugation rules.

5.1 Dictionary Size

The compiled dictionary and model sizes for the different morphological analyzers are shown in the Table 2. All dictionaries except KyTea's were built using the same conjugation-expanded Jumandic. For KyTea we used only deduplicated entries consisting of surface, POS and sub-POS fields. V1 and V2 models do not include the RNN model size. We can see that the dictionary size of MeCab is larger than the raw dictionary file; however, that is the size of a trie index. Please note that its index size is a consequence of MeCab being very popular and backward compatible with its models. It is extremely easy to swap its current implementation of a double array trie with the API-compatible implementation darts-clone[9] which will reduce the index size in half. Juman++ V2 uses this implementation. V1 dictionary size is significantly larger than of other analyzers because it was using data structures that are optimized for shared memory interprocess communication for the on-disk storage of the compiled dictionary. KyTea stores the string feature representations and its models become relatively large because of that. For the KyTea model trained on the concatenation of KU and KWDLC corpora, the model size is even larger than the V2 model without an RNN.

We also analyze the size of a V2 Jumandic model in more detail. Table 3 contains a breakdown

Table 2 Dictionary and model size comparison

Analyzer	Dictionary size (MB)	Model size (MB)
Raw Dictionary	256	—
MeCab	*311	7.7
KyTea	—	200
V1	445	135
V2	158	16

[9] https://github.com/s-yata/darts-clone

Table 3 Juman++ Jumandic model size breakdown

Parts	Size, bytes	Model %	Dict %
Double Array Trie Index	33.12 M	8.11%	27.49%
Entry Pointers	7.92 M	1.94%	6.57%
Field Pointers	30.81 M	7.55%	25.57%
String storages			
surface, baseform, reading	38.35 M	9.40%	31.83%
pos	205	—	—
subpos	702	—	—
conjform	2,704	—	—
conjtype	860	—	—
canonic	2.87 M	0.70%	2.39%
features	5.17 M	1.27%	4.29%
features (list pointers)	2.23 M	0.55%	1.85%
Linear model	16.00 M	3.92%	N/A
RNN	271.71 M	66.56%	N/A

of a V2 Jumandic model.[10] The table shows that the dictionary information itself is represented very compactly. About half of the dictionary space is taken by the double array index. Using a more compact representation of the trie index can make the models even smaller, still, the total model size is dominated by the RNN, thus the resulting size reduction of the whole model will not be significant.

The Jumandic model shares the string storages for surface form, reading and a base conjugation form which makes the string storage with most diverse entries. On the other hand, the content of POS-related fields is not diverse at all, as it can possible to see by sizes of their string storages. The on-disk size of field pointers depends on the size of the respective string storage, and together with the field alignment, it is possible to achieve the size of one byte per POS-related field.

5.2 Analysis Speed

We used a computer with Intel i7-6850K CPU, 64 GB of RAM and Ubuntu 16.04 Linux for the analysis speed comparison. For the speed benchmarking we have downclocked the CPU frequency to 3 GHz (from the 3.6 GHz base frequency), disabled the Turbo Boost technology and used the performance CPU scaling governor which runs the CPU at the fixed frequency. The models

[10] This information is accessible as a result of `jumanpp --model-info` command for an arbitrary spec.

Table 4 Morphological analysis speed comparison

Analyzer	Speed (sents/s)	Ratio
JUMAN	8,802	1.00
MeCab	52,410	0.17
KyTea (Jumandic)	4,892	1.79
KyTea (Unidic)	1,995	4.41
V1 noRNN	27	328.82
V1 RNN	16	535.72
V2 noRNN	7,422	1.18
V2 RNN	4,803	1.83

were trained from scratch using the same Juman++ dictionary, the Kyoto University[11] (KU) and KWDLC[12] corpora for all morphological analyzers except JUMAN, which is not trainable. For KyTea we also report the throughput of Unidic-based models which also perform the reading estimation, and are available for download from the KyTea website. A Jumandic-based model for KyTea does not use the reading estimation feature of KyTea and it was learned using the default parameters. V1 uses the full beam of width $j = 5$. V2 uses trimmed beam with parameters $j = 5$, $k = 6$, $l = 1$, $m = 5$. All analyzers were using only a single thread.

Table 4 shows the analysis speed of the considered morphological analyzers and speed ratio as compared to JUMAN. The speed was measured by analyzing 50k sentences from a web corpus. We use the median time of five launches with identical parameters for computing the analysis speed. V2 noRNN is only 20% slower than JUMAN while having a considerably complex model. V2 RNN has 1.8 times the execution speed of JUMAN and is more than 250 times faster than V1.

5.3 Analysis Accuracy

Figure 13 shows F1 scores with 95% confidence intervals for both the KU and KWDLC corpora. Data in the tabular format is in Appendix B. A concatenation of training sections of both corpora was used to train a combined model; the reported scores are for the test sections. MeCab and V2 have hyperparameters optimized using Spearmint (Snoek et al. 2012). The confidence intervals are computed using bootstrap resampling with 10,000 iterations. Note that if sides of two confidence intervals overlap by half, it means the statistical significance of $p = 0.036$, and non-overlapping intervals mean statistical significance at least of $p = 0.006$ (Cumming and

[11] http://nlp.ist.i.kyoto-u.ac.jp/EN/index.php?Kyoto University Text Corpus
[12] http://nlp.ist.i.kyoto-u.ac.jp/EN/index.php?KWDLC

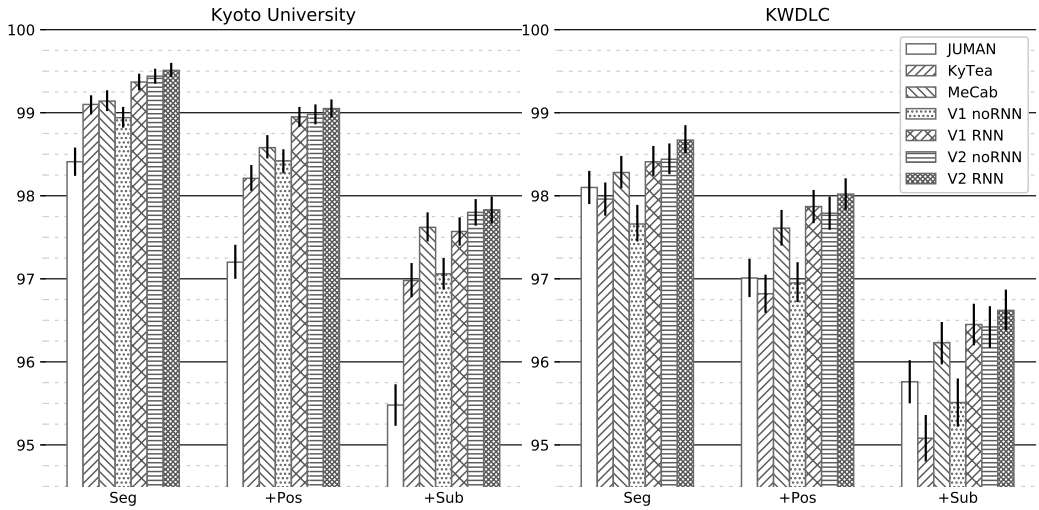

Fig. 13 F1 scores and 95% confidence intervals for accuracy of morphological analyzers on Jumandic-based corpora. Seg is segmentation; +Pos is correctly guessing the POS-tags after segmentation.

Finch 2005).

V2 RNN achieves a higher F1 score than the previous SOTA of V1 RNN. Even the scores of V2 noRNN are higher in some cases than those of V1 RNN. Note that the scores of V1 noRNN are one of the lowest, and thus we hypothesize that the number of training iterations of the V1 linear model was not sufficient. However, it was difficult to increase it because of a very slow analysis speed.

With V2, we could find an optimal number of iterations for learning the linear model with the best accuracy. The other reason for the improved accuracy for V2 is that it uses surface character and character type features.

5.4 Effects of Beam Trimming

We evaluate the effects of beam trimming on analysis accuracy by making an analysis experiment with different trimmed beam settings. We train multiple Juman++ models using different trimmed beam settings and compare their accuracy to each other and full beam setting. For the experiment, we use 10-fold cross-validation on the train section instead of the usual test-train split.

Figure 14 shows the average of the POS F1 score of different trimmed beam settings. Generally, when using the larger or equal beam size in test time than in the training time, the accuracy

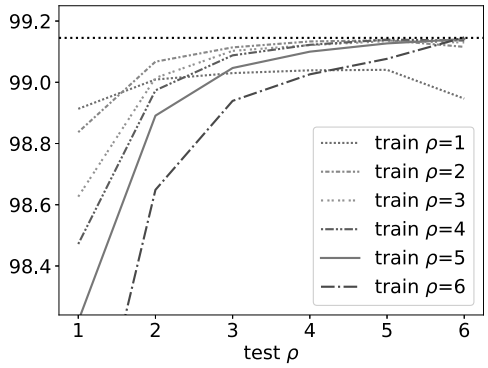

Fig. 14 Cross-validation test F1 score average on KU corpus for POS tags when using different timmed beam parameters $b_l = m_r = \rho$, $m_l = 1$. Dotted line at the top is F1 score in the full beam setting.

is not distinguishable from the full beam setting. We should note that training the model in the setting $\rho = 1$ (equivalent to Viterbi search on bigrams) fails to utilize the high-order features efficiently and achieves lower accuracy.

We also noticed that there exist sentences when the full and trimmed search configurations do not agree on the top-scoring path when analyzing random web sentences. To get a better picture, we analyzed a large number of sentences from a raw Japanese corpus, crawled from the web. On average, 0.38% of sentences had different analysis results, a relatively small number. Of those sentences, the full beam analysis was correct only in around 50% of the cases. The rest of sentences had both of analysis variants incorrect either because the dictionary did not support the language phenomena (20%) or lack of coverage by the training data (10%); the trimmed beam analysis was the correct one (10%); and other situations that were difficult to decide or impossible to analyze correctly like typos.

6 Discussion

While Juman++ achieves very high accuracy, there still exist some errors. We would like to discuss the most frequently occurring types.

Segmentation errors are the most visible ones for the users of MA. Juman++ often incorrectly segments katakana words. For example, ギネス|ビール, ストップ|ウオッチ or シベリアン|ハスキー are written in the corpus as single words. In these cases, the whole word is composed of the subcomponents and it is difficult to say that Juman++ segmentation is critically incorrect.

However, there are also cases of over-segmentation like ブル|ドーザー and ショベル|カー when the combined word is not composed of its parts and the segmentation does not make sense.

Another frequent case is an ambiguity in segmentation caused by a different part of speech. In Jumandic segmentation standard usages of na-adjectives when modifying verbs are not segmented (優位に|動く), but are segmented when に is a case marker (優位|に|立つ). This problem also manifests when the word is katakana. For example, Juman++ incorrectly treats リベラル as an adjective in the sentence 民主党 リベラル|に|よる高福祉. There exist very tricky cases like 準々決勝で|伊達|に|敗れ in a sentence about a sporting event. It is possible to decide the 100% correct segmentation only after checking the fact that a person named 伊達 participated in the event.

There exist compound words which can be segmented in different ways: 不快|感 or 不|快感; 工学|部 or 工|学部. While a non-corpus segmentation is registered as a segmentation error, this is more a problem of the segmentation standard than of an analyzer.

Lastly, there is a large number of over-segmented named entities like 新|民連 or デルタ|航空, especially in texts of the web domain. Sequences of characters of the same type are not as problematic as the entities of the second kind, containing characters of multiple classes (e.g. kanji and katakana or katakana and romaji). They could not be solved in the current paradigm of unknown word handling well. Nevertheless, we would like to argue that the named entities frequently consist of several morphemes. Here we have a conflict of how easy it is to use the results of the morphological analysis in other applications (handling sequences of tokens is more difficult than handling single tokens) versus the closeness to the uniformity of the tokenization. Unfortunately, the Jumandic does not make a clear decision at this point so conflicting situations arise in the segmentation.

Non-local Dependencies

One group of remaining accuracy problems is when the correct decision requires non-local information. The most frequent, albeit important such problem is the ambiguity between で being either a conjugation of copula だ or the instrumental case marker. For example, in the sentence “この湯豆腐と、あえ物、ごま豆腐、おひたしなどの精進料理で、体のしんまでポッカポカ。” で is the case marker, but the decision requires to take the overall sentence into the account. Moreover, punctuation does not help to make this decision.

Analyzers that make the local decisions are poorly fit to solve this problem. One possible approach would be to retain the ambiguity in the morphological analysis result and try to resolve this problem at the syntactic parsing level. It is, still, difficult to decide what kind of ambiguity

to keep and what to pass. There was an attempt to leave this ambiguity on the morphologic analysis level and try to resolve it as a syntactic parsing problem (Kawahara, Hayashibe, Morita, and Kurohashi 2017), but there was not much success. Accessing the global information about the sentence is key to solving this problem, so neural approaches could be helpful, but it was not thoughtfully studied yet.

Linear Weights Precomputation

While the current version of Juman++ does not precompute parts of linear model weights corresponding to unigram and bigram features to speed up computations, similarily to MeCab, it is possible. Still, not every feature can be precomputed, mostly because Juman++ allows us to use surface-based features which depend on the input string at the analysis time. Taking that into consideration, the design should give the user an ability to control the size of the precomputed weight table. Ultimately, this leads to significantly increased complexity of scoring and statically generated code. Also, the statically generated scoring procedure is going to depend on the decisions made at the precomputation time, compared to the current state when it depends only on the spec. Because of these considerations, the current version of Juman++ does not contain the functionality to precompute linear model weights.

It is, however, incorrect to think that Juman++ does not perform any type of precomputations. Converting the dictionary into a form that is easy to process is a form of precomputation. We additionally resolve conditional statements at the dictionary compile time.

Using SIMD and Vector Instructions

While Juman++ exploits out-of-order capabilities of modern CPUs, we do not exploit the SIMD parallelism in the linear model. RNN implementation uses Eigen library (Guennebaud, Jacob, et al. 2010) and is vectorized.

We did some experiments with vectorizing the feature and score computation; however, the scalar version was faster when using SSE/128-bit vectors and on the same level of performance for AVX2/256-bit vectors. We did not experiment with AVX-512 because of the lack of easily accessible AVX-512 hardware at the active development time of Juman++. One of the problems with them is the high latency of gather-type instructions needed for random access of model weights and the lack of gather-prefetch instructions. It would be interesting to explore this direction in the future.

Portability

The techniques proposed in this paper can be divided into two parts: architecture-dependent and architecture-independent. Everything related to the dictionary structure is architecture-independent. Other parts make assumptions about the underlying hardware, but most modern and widely used hardware satisfies our assumptions.

Linear congruent generators, which our hash function is based on, have better statistical properties when they have larger state space, and our implementation uses 64-bit integers because most widely used architectures are 64-bit and provide a large number of 64-bit integer registers with fast operations on such numbers. The algorithms should work with the 32-bit intermediate hash state, but we did not test it.

The rest of the improvements assume that the executing CPU will have out-of-order execution. Having a complex memory subsystem that supports prefetching and multiple asynchronous and simultaneous memory accesses usually stems from the out-of-order nature. Fortunately, nearly every modern CPU aimed at performance, in both server, desktop or mobile form-factors, has out-of-order execution. As far as we know, only extremely low-power or simple microcontrollers and CPU with special needs (e.g. ARM Cortex R series which have fixed latency for all instructions) are in-order. We believe that Juman++, in general, would be a bad match for tiny embedded systems not only because of applied optimization methods but also because such CPUs usually are not packaged with a large amount of RAM. Software prefetching, however, sometimes is supported on in-order CPUs as well, and can be used to improve the execution speed in such situations.

Juman++ and its techniques target the common characteristics of out-of-order CPUs, like asynchronous memory access, without being optimized for a specific microarchitecture. There is nothing prohibiting optimizations of Juman++ to work on ARMv8+/PowerPC CPUs in addition to x86_64 without difficult porting. Finally, we must remark that approaches to computer architecture like VLIW which put the decisions over instruction-level parallelism on the programmer and compiler could also be exploited similarly.

Low Accuracy on Sentence Level

While there exists an impression that the morphological analysis has extremely high accuracy (e.g. 99.5% F1 score for segmentation of newspaper texts), there exist some caveats. The evaluation is done on a token level and for Japanese, there exists a large number of *easy* tokens, mostly related to the grammar. Even JUMAN, as an example of an analyzer which uses an HMM model for analysis with mostly bi-gram features, gets the segmentation score above 98% F1. We

should note that greedy matching prefix dictionary strings do not perform well in the segmentation task and achieve only 79% F1 score. For suffixes (analyzing the end to the beginning) it is even worse with 68%.

On the other hand, when it comes to the sentence level the accuracy is not that high. For the KU corpus, it is 93% (127/1,783 contain segmentation errors) and for the KUDLC it is 88% (265/2,195). Moreover, for more complicated and noisy domains like user-generated texts in social media, it is even lower.

Being Robust to Input Errors

Most of the current morphological analyzers treat the input in a way so it does not contain errors. Unfortunately, real word texts contain typos and other linguistic phenomena. Some of them, like IME misconversions, are mostly Japanese-specific.

Lattice-based analyzers can't handle such situations well. For example, when the misconversion contains actual dictionary words, the analyzer would use those, producing nonsense analysis. Still, humans are pretty robust against such errors and can infer the original meaning of the sentence pretty easily. There exist prior work in this direction (Saito, Sadamitsu, Asano, and Matsuo 2014), but it is one of the open problems for the morphological analysis.

Approaches With and Without Dictionary

The current state of morphological analysis accuracy is happened because of the high quality of segmentation dictionaries for Japanese. Also, from the user point of view, it is easy to tune the analyzer for their task or domain by adding new words to the dictionary. Because the MeCab-based analyzers use primarily POS-based features for scoring, adding new words could be done without retraining the whole model. This, however, makes their models relatively large for large dictionaries.

Juman++, in theory, allows adding new words to the analysis dictionary. If we add new content to the end of string storages we will not change values of present field pointers, so the feature hash values will stay the same. The field pointer table itself can be completely rebuilt without retraining the linear and RNN models. This functionality, however, is not implemented presently because of time constraints.

Tolmachev et al. (2019) bootstrapped a neural network-based morphological analyzer based on a large-scale corpus analyzed by a traditional dictionary-based analyzer. Their approach did not model the dictionary explicitly, predicting segmentation and POS from the encoded unigram representations. Their approach, nevertheless, matched and in some cases surpassed

the bootstrapping analyzer, having a very compact model. This approach seems to learn a *word model*, preferring to output some tokens which make sense to humans, but have different segmentation in the corpus, e.g. 食材 which is segmented into two characters in the corpus. The downside to their approach is that adding new words to the model requires reanalyzing huge corpus and then retraining the model, making the process very cumbersome.

7 Conclusion

We described in detail the improvements of the Juman++ V2 morphological analyzer. It is more than 250 times faster than Juman++ V1 and achieves higher analysis accuracy. Compared to MeCab, even the version without RNN does significantly more computations: it always performs the beam search and accesses > 60 weights per lattice node while being only 5 times slower. We believe that the techniques used in the development of Juman++ V2 would be useful for other high-performance NLP tools.

Acknowledgement

The authors would like to thank Hajime Morita for implementing the first version of Juman++. We also thank anonymous reviewers, Hajime Morita, Tomoya Iwakura and Taku Kudo for their insightful comments and discussions which have greatly helped to improve this paper. The first author also would like to thank the Monbukagakusho MEXT program for scholarship support, Kyoto University Design School and Fujitsu Laboratories.

Experiment results of this paper were partially presented as the system demonstration at the 2018 Conference on Empirical Methods in Natural Language Processing (Tolmachev et al. 2018). The section on the morphological analysis is based on the paper presented at the meeting of North American Chapter of the Association for Computational Linguistics (Tolmachev et al. 2019).

Reference

Asahara, M. and Matsumoto, Y. (2000). "Extended Hidden Markov Model for Japanese Morphological Analyzer." Tech. rep. 54, The Special Interest Group Notes of IPSJ.

Cumming, G. and Finch, S. (2005). "Inference by Eye: Confidence Intervals and How to Read Pictures of Data." *The American Psychologist*, **60**, pp. 170–180.

Den, Y., Nakamura, J., Ogiso, T., and Ogura, H. (2008). "A Proper Approach to Japanese Morphological Analysis: Dictionary, Model, and Evaluation." In *Proceedings of the 6th International Conference on Language Resources and Evaluation (LREC 2008)*, pp. 1019–1024.

Guennebaud, G., Jacob, B., et al. (2010). "Eigen v3." http://eigen.tuxfamily.org.

Jiang, W., Mi, H., and Liu, Q. (2008). "Word Lattice Reranking for Chinese Word Segmentation and Part-of-Speech Tagging." In *Proceedings of the 22nd International Conference on Computational Linguistics - Volume 1*, COLING '08, pp. 385–392, USA. Association for Computational Linguistics.

Kaji, N. and Kitsuregawa, M. (2013). "Efficient Word Lattice Generation for Joint Word Segmentation and POS Tagging in Japanese." In *Proceedings of the 6th International Joint Conference on Natural Language Processing*, pp. 153–161, Nagoya, Japan. Asian Federation of Natural Language Processing.

Kawahara, D., Hayashibe, Y., Morita, H., and Kurohashi, S. (2017). "Automatically Acquired Lexical Knowledge Improves Japanese Joint Morphological and Dependency Analysis." In *Proceedings of the 15th International Conference on Parsing Technologies*, pp. 1–10, Pisa, Italy. Association for Computational Linguistics.

Kitagawa, Y. and Komachi, M. (2018). "Long Short-Term Memory for Japanese Word Segmentation." In *Proceedings of the 32nd Pacific Asia Conference on Language, Information and Computation*, Hong Kong. Association for Computational Linguistics.

Kruengkrai, C., Uchimoto, K., Kazama, J., Wang, Y., Torisawa, K., and Isahara, H. (2009). "An Error-driven Word-character Hybrid Model for Joint Chinese Word Segmentation and POS Tagging." In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1–Volume 1*, pp. 513–521. Association for Computational Linguistics.

Kudo, T. (2018). *Theory and Implementation of Morphological Analysis (In Japanese)*. Jissen · NLP Series. Kindaikagakusha.

Kudo, T. and Richardson, J. (2018). "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing." In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 66–71, Brussels, Belgium. Association for Computational Linguistics.

Kudo, T., Yamamoto, K., and Matsumoto, Y. (2004). "Applying Conditional Random Fields to Japanese Morphological Analysis." In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pp. 230–237.

Kurohashi, S. (1994). "Improvements of Japanese Morphological Analyzer JUMAN." In *Pro-*

ceedings of The International Workshop on Sharable Natural Language, 1994.

Kurohashi, S. and Kawahara, D. (2012). "Japanese Morphological Analysis System JUMAN 7.0 Users Manual.".

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT press.

Matsumoto, Y., Takaoka, K., and Asahara, M. (2008). "Morphological Analysis System ChaSen Users Manual. Version 2.4.3.".

Murawaki, Y. (2019). "On the Definition of Japanese Word." *CoRR*, **abs/1906.09719**.

Mikolov, T. (2012). *Statistical Language Models Based on Neural Networks*. Ph. d. thesis, Brno University of Technology, Brno.

Mochihashi, D., Yamada, T., and Ueda, N. (2009). "Bayesian Unsupervised Word Segmentation with Nested Pitman-Yor Language Modeling." In *ACL 2009, Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP, 2–7 August 2009, Singapore*, pp. 100–108.

Morita, H., Kawahara, D., and Kurohashi, S. (2015). "Morphological Analysis for Unsegmented Languages using Recurrent Neural Network Language Model." In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 2292–2297, Lisbon, Portugal. Association for Computational Linguistics.

Neubig, G., Nakata, Y., and Mori, S. (2011). "Pointwise Prediction for Robust, Adaptable Japanese Morphological Analysis." In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 529–533, Portland, Oregon, USA. Association for Computational Linguistics.

O'Neill, M. E. (2014). "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation." Tech. rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA.

Saito, I., Sadamitsu, K., Asano, H., and Matsuo, Y. (2014). "Morphological Analysis for Japanese Noisy Text Based on Character-level and Word-level Normalization." In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 1773–1782.

Sasano, R., Kurohashi, S., and Okumura, M. (2013). "A Simple Approach to Unknown Word Processing in Japanese Morphological Analysis." In *Proceedings of the 6th International Joint Conference on Natural Language Processing*, pp. 162–170, Nagoya, Japan. Asian Federation of Natural Language Processing.

Shafranovich, Y. (2005). "Common Format and MIME Type for Comma-Separated Values (CSV) Files." Rfc 4180, Internet Requests for Comments.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). "Practical Bayesian Optimization of Machine Learning Algorithms." In *NIPS*, pp. 2951–2959.

Takaoka, K., Hisamoto, S., Kawahara, N., Sakamoto, M., Uchida, Y., and Matsumoto, Y. (2018). "Sudachi: A Japanese Tokenizer for Business." In *Proceedings of the 11th International Conference on Language Resources and Evaluation (LREC 2018)*, pp. 2246–2249, Miyazaki, Japan. European Language Resources Association (ELRA).

Tolmachev, A., Kawahara, D., and Kurohashi, S. (2018). "Juman++: A Morphological Analysis Toolkit for Scriptio Continua." In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 54–59, Brussels, Belgium. Association for Computational Linguistics.

Tolmachev, A., Kawahara, D., and Kurohashi, S. (2019). "Shrinking Japanese Morphological Analyzers With Neural Networks and Semi-supervised Learning." In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2744–2755, Minneapolis, Minnesota. Association for Computational Linguistics.

Tolmachev, A. and Kurohashi, S. (2018). "Juman++ v2: A Practical and Modern Morphological Analyzer." In *Proceedings of 24th Annual Meeting of Japanese Natural Language Processing Society*, pp. 917–920, Okayama, Japan.

Uchiumi, K., Tsukahara, H., and Mochihashi, D. (2015). "Inducing Word and Part-of-Speech with Pitman-Yor Hidden Semi-Markov Models." In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26–31, 2015, Beijing, China, Volume 1: Long Papers*, pp. 1774–1782.

Wang, J., Zhao, P., and Hoi, S. C. (2016). "Soft Confidence-Weighted Learning." *ACM TIST*, **8** (1), p. 15.

Zheng, X., Chen, H., and Xu, T. (2013). "Deep Learning for Chinese Word Segmentation and POS Tagging." In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 647–657.

Zhikov, V., Takamura, H., and Okumura, M. (2010). "An Efficient Algorithm for Unsupervised Word Segmentation with Branching Entropy and MDL." In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pp. 832–842, Cambridge, MA. Association for Computational Linguistics.

Appendix

A Some Notes on Computer Architecture

Modern computers are complex. While the speed of CPUs was increasing very fast, memory latency did not increase that much. Because of this, modern CPUs have multi-level cache hierarchy with different access speeds. The CPUs also execute multiple instructions at the same time. When utilizing ineffective hardware utilization does not change the correctness of a program, it can drastically change the execution speed. We use some tricks that explicitly target the details of modern CPUs in the implementation of Juman++. This section provides some simplified details on modern computer architecture which are required to explain our decisions for Juman++. Juman++ for the lattice adopts struct-of-arrays data layout, which is described in this section.

A.1 CPUs and Memory

Modern computers have complex memory hierarchy and different memory access patterns can have different execution speeds. Table 5[13] shows approximate access latencies of different parts of modern hardware. While the absolute numbers are not precise, the orders of magnitude are correct.

It is easy to see that cache accesses are much faster than accesses to main memory, nevertheless the external storage. However, cache sizes are very limited. Modern Intel CPUs have 32 KB of L1 data cache per execution core, organized by 4,096 512-bit cache lines. L2 cache is 256 KB per core and is shared for code and data. L3 cache is usually shared by all the CPUs and the number

Table 5 Approximate latencies of the modern hardware

Category	Latency	Comment
L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	14× L1 cache
Main memory reference	100 ns	20× L2 cache, 200× L1 cache
Read 1 MB sequentially from memory	250,000 ns	
Read 1 MB sequentially from SSD	1,000,000 ns	4× memory
Disk seek	10,000,000 ns	
Read 1 MB sequentially from disk	20,000,000 ns	

[13] https://gist.github.com/jboner/2841832

is around 1.5 MB per core for server CPUs.

When a CPU executes an instruction accessing the memory, the data is first read from the L1 cache, if it does not contain the requested data then the request is forwarded to the L2, and then similarly to the L3. Only if the data is not contained in the L3, the access is forwarded to the main memory. The computation continues only after the data is fetched through all the cache levels. Data is evicted from caches based on a least frequently used heuristics. This means that accessing the same or close data will hit the cache, but random accesses will miss.

Also, it is possible to *prefetch* the data. By issuing a special instruction, we give the CPU a hint that a piece of data will be accessed soon, so it should get it from the memory and put it into the cache. Prefetching can be used to hide the latency of the main memory. In addition to *manual* prefetching, CPUs also detect frequently-used access patterns and automatically prefetch the data into the cache. Automatic prefetching is more effective because it does not require to dispatch additional instructions, but it can not detect random access patterns.

Modern high-performance CPUs are also *out-of-order*. They do not execute instructions sequentially (in-order), and they can execute multiple instructions each cycle. Instead, they convert instructions into a directed acyclical graph-like structure and traverse it in parallel. Inputs to an instruction become its dependencies in the graph. Converted instructions are placed into a reorder buffer. When all the inputs to the instruction in the buffer are ready, it is executed. For example, Intel Haswell microarchitecture has a reorder buffer of size 192,[14] meaning that there could be 192 instructions at the given time "in flight" and it can execute up to 4 instructions each cycle.

A.2 Memory Layouts and Cache Efficiency

Almost every programming language provides a means to pack the data into *structures* as a way to produce abstractions. They work by joining the data, possibly of a different kind, together. Struct data is usually laid sequentially after each other by language compilers and runtimes as well. When working with multiple similar data, it is often put into arrays: homogenous sequences of the data. This data layout is called *array-of-structs* (AoS).

Structs are an important building block of abstractions and are irreplaceable for building complex software. Still, structs can be large and when performing the array processing, programs tend to operate only on a subset of each struct. On the other hand, CPU caching works in cache line granularity. A CPU fetches data from the memory in the relatively large packets of fixed

[14] https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client)#Execution_engine

size (on the x86 architecture it is 512 bits–64 bytes). If an application uses large structs (e.g. larger than a cache line) and touches only a single 32-bit sized integer field from each of them, this means that the *efficient* cache utilization is very low (only 6.25% is used, the rest is wasted by the non-touched data) and there is a large number of main memory accesses, which are much slower than cache accesses.

Of course, this is an example of a pathological case and in the real world, the cache efficiency is not as bad. Still, high-performance code often adopts *struct-of-arrays* (SoA) data layout instead of usual array-of-structs. With this layout, the fields are packed together in an array for each field of the structure. Each struct becomes decomposed over several arrays. The difference between AoS and SoA layouts for a struct of 4 fields in a 4 element array is illustrated in Figure 15. With AoS layout the access patterns which touches only a subset of the fields at a time utilize the CPU caches much effectively. Moreover, hardware prefetchers of modern CPUs efficiently support SoA access patterns.

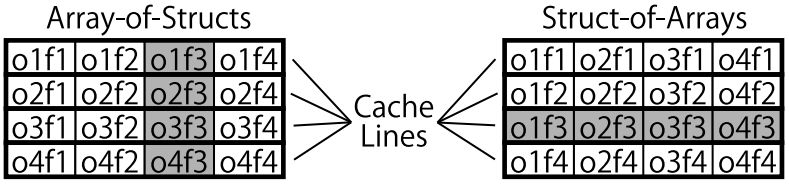

Fig. 15 Illustration of stuct-of-arrays and arrray-of-structs layouts of a 4-field object in a 4 element array. Gray is an accessed field. When using array-of-structs layout, accessing *f3* of all objects uses all four cache lines, however in stuct-of-arrays layout accesses are localized in a single cache line.

Table 6 F1 scores and 95% confidence intervals for accuracy of morphological analyzers on Jumandic-based corpora. Seg is segmentation; +Pos is correctly guessing the POS-tags after segmentation.

Analyzer	Kyoto University			KWDLC		
	Seg	+Pos	+Sub	Seg	+Pos	+Sub
JUMAN	$98.41^{98.58}_{98.24}$	$97.20^{97.40}_{96.99}$	$95.48^{95.73}_{95.23}$	$98.10^{98.30}_{97.90}$	$97.01^{97.24}_{96.78}$	$95.76^{96.02}_{95.50}$
KyTea	$99.10^{99.22}_{98.99}$	$98.21^{98.36}_{98.05}$	$96.98^{97.18}_{96.77}$	$97.96^{98.16}_{97.76}$	$96.82^{97.05}_{96.59}$	$95.08^{95.36}_{94.80}$
MeCab	$99.14^{99.26}_{99.01}$	$98.58^{98.71}_{98.43}$	$97.62^{97.79}_{97.44}$	$98.28^{98.47}_{98.08}$	$97.61^{97.82}_{97.39}$	$96.23^{96.49}_{95.98}$
V1 noRNN	$98.94^{99.06}_{98.81}$	$98.42^{98.57}_{98.28}$	$97.06^{97.25}_{96.87}$	$97.66^{97.87}_{97.43}$	$96.95^{97.18}_{96.70}$	$95.51^{95.80}_{95.22}$
V1 RNN	$99.37^{99.47}_{99.27}$	$98.95^{99.07}_{98.83}$	$97.57^{97.74}_{97.40}$	$98.41^{98.58}_{98.22}$	$97.87^{98.07}_{97.67}$	$96.45^{96.70}_{96.20}$
V2 noRNN	$99.44^{99.53}_{99.35}$	$98.98^{99.10}_{98.86}$	$97.80^{97.96}_{97.64}$	$98.44^{98.62}_{98.25}$	$97.79^{97.99}_{97.59}$	$96.42^{96.67}_{96.17}$
V2 RNN	$\mathbf{99.51}^{99.59}_{99.42}$	$\mathbf{99.05}^{99.16}_{98.94}$	$\mathbf{97.83}^{97.99}_{97.67}$	$\mathbf{98.67}^{98.83}_{98.49}$	$\mathbf{98.02}^{98.21}_{97.83}$	$\mathbf{96.62}^{96.86}_{96.37}$

B Accuracy Comparison Tabular Data

See Table 6. Each table entry consists of three numbers. The left number is the F1 score on the test data. Top right and bottom right numbers are upper and lower confidence interval bounds, respectively.

Arseny Tolmachev: Arseny Tolmachev received his B.S. in Ural Federal University, Ekaterinburg, Russia in 2010. He was working as an engineer in the Institute of Engineering Science, RAS (Ural Branch) on high-performance computing applications. From 2013 he became a student of Graduate School of Informatics, Kyoto University and received his M.S. there in 2016. He is currently a researcher in Fujitsu Laboratories, Ltd. His research interests center on natural language processing, deep learning, explainability of black-box models and efficient use of computers.

Daisuke Kawahara: Daisuke Kawahara received his B.S. and M.S. in Electronic Science and Engineering from Kyoto University in 1997 and 1999, respectively. He obtained his Ph.D. in Informatics from Kyoto University in 2005. He is currently an associate professor at the Graduate School of Informatics at Kyoto University. His research interests center on natural language processing, particularly knowledge acquisition and text understanding.

Sadao Kurohashi: Sadao Kurohashi received the B.S., M.S., and Ph.D. in Electrical Engineering from Kyoto University in 1989, 1991 and 1994, respectively. He has been a visiting researcher of IRCS, the University of Pennsylvania in 1994. He is currently a professor of the Graduate School of Informatics at Kyoto University. His research interests include natural language processing and knowledge acquisition/representation.

(Received October 29, 2019)
(Accepted December 25, 2019)