最小コスト法に基づく形態素解析における CPU キャッシュの効率化

神田峻介[1]　赤部晃一[1]　後藤啓介[1]　小田悠介[2]
[1]LegalOn Technologies Research [2]東北大学 データ駆動科学・AI 教育研究センター
{shunsuke.kanda,koichi.akabe,keisuke.goto}@legalontech.jp
yusuke.oda.c1@tohoku.ac.jp

概要

最小コスト法に基づく形態素解析において、木の探索やコスト行列の参照により発生するランダムアクセスは、CPU キャッシュ効率低下の原因となる。本稿では、参照の局所性を改善するデータ構造を提案し、解析速度の改善を図る。実験の結果、提案法を適用しなかった場合と比べ、提案法は 40%程度の時間短縮を達成した。また、提案法を組み込んだ形態素解析器 Vibrato[1)] を新たに開発し、既存のソフトウェアよりも高速に動作することを実証した。

1　はじめに

形態素解析は、自然言語を入力とし、形態素の列を返す処理である。日本語の情報検索やテキストマイニングなどの自然言語処理において、形態素解析は重要な前処理である。形態素解析を実現する代表的な手法の 1 つに最小コスト法 [1] があり、MeCab [2] や Sudachi [3] などの形態素解析器で利用されている。

最小コスト法の時間的ボトルネックの一つに、辞書の肥大化に伴う CPU キャッシュ効率の低下が挙げられる。単語列挙のための木構造の探索や、コスト計算のための行列参照では、メモリ上のランダムアクセスが頻繁に発生する。巨大な辞書では、このランダムアクセスがキャッシュミスを引き起こし、速度低下の原因となる。

本稿では、参照の局所性の良いデータ構造を設計し、キャッシュ効率を改善することで最小コストの高速化を達成する。IPADIC [4] や Neologd [5]、UniDic [6, 7] を使った実験により、その有効性を実証する。実験の結果、提案法を適用しなかった場合と比べて、提案法は 40%程度の高速化を達成した。

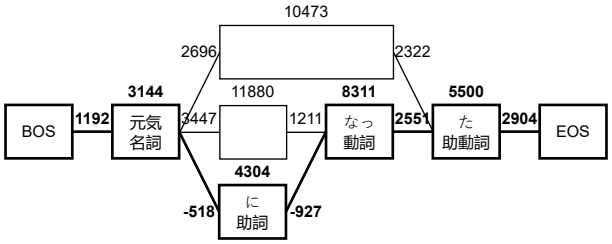

図 1　入力文「元気になった」についての形態素ラティス。太線はコストの和が最小となる経路を表す。

また、提案法を実装した形態素解析器 Vibrato を新たに開発した。実験により、Vibrato は MeCab の 2 倍以上の速度性能であることを実証する。

2　最小コスト法

2.1　アルゴリズム

最小コスト法は、以下の手順により入力文から形態素列[2)] を得るアルゴリズムである。

1. 入力文に現れる形態素をノードとしたグラフ構造（形態素ラティス）を構築
2. コストが最小となる経路を探索し、対応する形態素列を出力

図 1 に入力文「元気になった」から構築された形態素ラティスを示す。BOS と EOS は、文の先頭と末尾を表すダミーのノードである。形態素に対応するノードと形態素の並びの対応するエッジにはコストが割り当てられる。ノードのコストは、その形態素自体の出現しやすさを表す。エッジのコストは、その形態素の並びの出現しやすさを表す。以降では、形態素辞書やコスト値は所与として各手順を説明する。

形態素ラティスの構築　入力文が与えられたとき、形態素ラティスは以下のように構築される。

1）https://github.com/daac-tools/vibrato

2）本稿では、辞書のエントリを形態素と呼ぶ。

1. 入力文に現れる形態素を列挙（辞書引き）
2. 形態素をノードとして、文の位置について隣り合うノード同士をエッジで連結

辞書引きはトライ [8] という木構造を使って効率的に計算できる。詳しくは 3 節で解説する。

経路の探索　形態素ラティスの BOS から EOS までを繋ぐ、コストの和が最小となる経路を探索する。そして、その経路に対応する形態素列を解として出力する。例えば、図 1 では太線の経路が該当する。そのような経路は、ビタビアルゴリズム [9] を用いてエッジ数に線形の時間で計算できる。

2.2　高速化の方針

本研究では、最小コスト法のボトルネックは以下の 2 点であると指摘し、これらを解消することで解析速度の改善を試みる。

- 辞書引きでのトライの探索
- 経路探索中のコストの参照

これらに共通する点は、メモリ上の頻繁なランダムアクセスである。巨大な辞書では、このランダムアクセスがキャッシュミスを引き起こし、速度低下の原因となる。そこで、それぞれに参照の局所性が良いデータ構造を設計する。詳しくは、3 節と 4 節で解説する。

3　辞書引きのキャッシュ効率化

3.1　事前知識

多くの形態素解析器では、トライ [8] を使った共通接頭辞検索により辞書引きを実現する。トライとは文字列の接頭辞を併合して構築される木構造であり、図 2 に見られるようにエッジにラベルを付随したオートマトンの一種である。共通接頭辞検索とは、あるクエリ文字列の接頭辞として現れる登録文字列を列挙する操作であり、文字を使って根からノードを遷移することで実現される。形態素をトライに登録し、入力文の全ての開始位置から共通接頭辞検索を実行することで辞書引きは実現される。

トライを表現するデータ構造としてはダブル配列 [10] が広く用いられている。ダブル配列とは、ノードからノードへの探索を定数時間で実現する高速なデータ構造である。入力文の長さを N、登録形態素の最大長 K としたとき、ダブル配列を使った場合の辞書引きの実行時間は $O(NK)$ である。

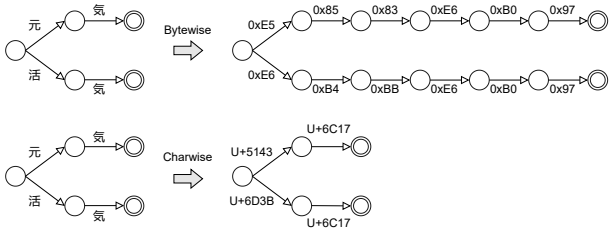

図 2　形態素「元気」「活気」を登録したトライ。文字列のエンコードが Unicode である場合の、上図は Bytewise で表現した例、下図は Charwise で表現した例を示す。

3.2　背景と問題点

ダブル配列の代表的な実装として、Darts[3] や Darts-clone[4] があり、MeCab や Sudachi などで用いられている。これらの共通点として、文字列をそのデータ表現のバイト列として扱う点が挙げられる。この方法を **Bytewise** と表記する。図 2 の上部に例を示す。

Bytewise の利点は、文字列のエンコードに関わらずバイト列として処理できることと、エッジラベルの種類数の上限が 256 に定まることである。これらは実装の汎用性や容易さに関係する。

Bytewise の欠点は、日本語などのマルチバイト文字を扱う場合にバイト列が長くなることである。図 2 の例では、Unicode で符号化された長さ 2 の形態素「元気」が、長さ 6 の UTF-8 バイト列として表現されている。これは「元気」を検索するときに 6 回のランダムアクセスが必要となることを意味し、キャッシュミス増加の原因となる。

3.3　改善案

文字列を Unicode のコードポイント値の系列として処理する。この改善案を **Charwise** と表記する。図 2 の下部に例を示す。

日本語の多くの文字は UTF-8 で 3 バイトを使って表現される。そのような文字について、Charwise では Bytewise に比べランダムアクセスの回数を 3 分の 1 に削減できる。ダブル配列では定数時間でノードの探索ができるので、計算量にも影響は無い。

Charwise の実装上の注意点は、エッジラベルの種類数が大きくなる点である。コードポイント値は `U+0000` から `U+10FFFF` までの 110 万種類を扱う。そのようなダブル配列は、素朴に実装すると構築速度やメモリ効率が低化する [11]。そこで、コード値を

3）http://chasen.org/~taku/software/darts/
4）https://github.com/s-yata/darts-clone

頻度順に割り当て直すことで、多くのコード値が小さい値になるように修正する。自然言語において文字の出現頻度には偏りがあるため、この修正で問題が解消されることが経験的に知られている [11]。

4 コスト参照のキャッシュ効率化

4.1 事前知識

形態素ラティスのエッジに付随したコストを連接コストとよぶ。連接コストは左側と右側ノードの素性情報のペアによって決定される。具体的には、形態素には左文脈 ID と右文脈 ID が割り当てられ、それら ID のペアによって連接コストは決定される。

本稿では、左文脈 ID の集合を $X = \{0, 1, \ldots, |X| - 1\}$、右文脈 ID の集合を $Y = \{0, 1, \ldots, |Y| - 1\}$ と定義する。連接コストは $|X| \times |Y|$ の連接表 M に保存され、文脈 ID ペア $(x, y) \in X \times Y$ の連接コスト値は表 M の (x, y) 成分 $M[x, y]$ に格納される。M は単純な二次元配列で実装され、$M[x, y]$ には値 x, y を使って定数時間でアクセスできる。

4.2 背景と問題点

現代書き言葉 UniDic [7] では素性を詳細に設計しており、結果として連接表が肥大化している。例えば、IPADIC v2.7.0 では $|X| = |Y| = 1,316$ で連接表のサイズは高々 3.3 MiB である。[5] それに対し、UniDic v3.1.0 では $|X| = 15,388, |Y| = 15,626$ で連接表のサイズは 459 MiB にもなる。事前実験（付録 A）では、解析時間は連接表のサイズに比例して増加する傾向が見られており、参照の局所性の悪化が解析速度の低下の原因になると考察される。

4.3 改善案

訓練コーパスを用いて文脈 ID の使用頻度を収集し、よく使用される順に若い文脈 ID を割り当て直す。この方法により、頻繁に使用される連接コスト値がメモリ上で近接する。使用頻度に大きな偏りがある場合、参照の局所性が改善する。図 3 に例を示す。本稿では、標準の文脈 ID を使用する場合を **Default**、頻度順にマッピングする場合を **Mapped** と表記する。

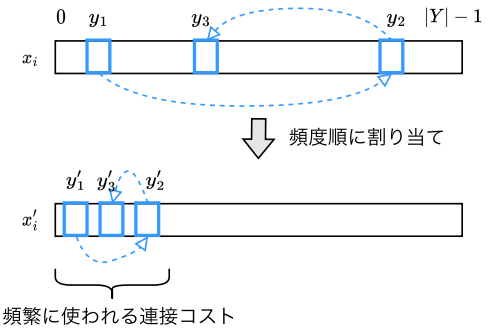

図 3 文脈 ID を頻度順に割り当て直した例。ある左文脈 ID x_i に対応する行と、x_i について右文脈 ID y_1, y_2, y_3 の順でコスト値を参照した様子を図示している。x'_i, y'_j は頻度順に割り当て直された文脈 ID を表す。y_1, y_2, y_3 が頻繁に使用される文脈 ID であった場合、y'_1, y'_2, y'_3 は小さい値となる。つまり、その連接コスト値は行の先頭付近に密集し、参照の局所性は改善する。

表 1 辞書の統計

辞書	形態素数	連接表 $\lvert X\rvert \times \lvert Y\rvert$	連接表 サイズ
ipadic-mecab	392,126	$1,316 \times 1,316$	3.3 MiB
ipadic-neologd	4,668,394	$1,316 \times 1,316$	3.3 MiB
unidic-mecab	756,463	$5,981 \times 5,981$	68 MiB
unidic-cwj	1,879,222	$15,388 \times 15,626$	459 MiB

5 実験

5.1 実験設定

形態素解析用の辞書は以下の 4 種類を評価した。

- ipadic-mecab (v2.7.0) [4]
- ipadic-neologd (2020-09-10) [5]
- unidic-mecab (v2.1.2) [6]
- unidic-cwj (v3.1.0) [7]

文章コーパスには BCCWJ v1.1 [12, 13] を使用した。マッピングの訓練にはコアデータ 60k 文を使用した。形態素解析の速度評価には、サブコーパスからランダムに抽出した 100k 文を使用した。

手法の実装にはプログラミング言語 Rust を用いた。コンパイルに使用した `rustc` は v1.63.0 であり、最適化フラグは `opt-level=3` である。計算機環境は、Intel Core i9-12900K @3.2–5.2GHz CPU (Cache-line: 64B, L1d: 640KiB, L2: 14MiB, L3: 30MiB), 64GiB RAM であり、OS は Ubuntu 22.04 である。実験は全てシングルスレッドで実施した。ダブル配列について、Bytewise の実装には Yada v0.5.0[6] を用いた。

5） 各コスト値は 2 バイトを使って保存される。

6） https://github.com/takuyaa/yada

表 2　各手法を適用した場合の解析時間 [ms]。括弧内の数値は Bytewise+Default からの変化率を表す。

辞書	Bytewise + Default	Charwise + Default	Charwise + Mapped
mecab-ipadic	390	343 (−12.1%)	330 (−15.4%)
mecab-neologd	538	479 (−11.1%)	467 (−13.3%)
unidic-mecab	616	567 (−8.0%)	499 (−19.0%)
unidic-cwj	1,012	949 (−6.3%)	587 (−42.1%)

表 3　L1 キャッシュへのデータ読み込みにおけるキャッシュミス回数 [10^6]。括弧内の数値は Bytewise+Default からの変化率を表す。

辞書	Bytewise + Default	Charwise + Default	Charwise + Mapped
ipadic-mecab	71.0	66.7 (−6.0%)	43.6 (−38.5%)
ipadic-neologd	86.0	79.9 (−7.1%)	56.3 (−34.6%)
unidic-mecab	136.1	130.3 (−4.3%)	71.4 (−47.5%)
unidic-cwj	231.8	226.3 (−2.4%)	104.6 (−54.9%)

Charwise の実装には Crawdad v0.3.0[7] を用いた。

5.2　提案法の分析

提案法の Charwise と Mapped を用いた場合の解析速度を評価する。評価データ 100k 文の解析に要した時間を表 2 に示す。結果は 10 回試行した平均であり、結果中に示す「変化率」は、$\frac{\text{変化先} - \text{変化元}}{\text{変化元}}$ で算出した値である。

文脈 ID が Default の場合での Bytewise と Charwise の解析時間を比較する。全ての辞書で Charwise が高速であり、IPADIC では 10%以上の改善が確認された。一方で、巨大な連接表を持つ unidic-cwj では、相対的にコスト参照のボトルネックの方が大きく、6.3%の改善に留まった。

続いて、文脈 ID の割り当てを Mapped に変更した場合と比較する。全ての辞書で Mapped が高速であり、特に連接表が最も大きい unidic-cwj では 42.1%の大きな改善が確認された。

表 3 に、評価データ 100k 文の解析時に発生した、L1 データキャッシュへの読み込み時のキャッシュミス回数を示す。計測には `perf` コマンドを使用し、結果は 100 回試行した平均である。

全ての場合において、提案法によりキャッシュミスの回数が削減している。Charwise 適用時には ipadic-neologd で、Mapped 適用時には unidic-cwj で改善率が最も大きく、提案法が巨大な辞書について効果的であることが確認できる。

これらの結果から、最小コスト法に基づく形態素解析器において、辞書引きとコスト参照のキャッシュ効率は解析速度に大きな影響を与えることが確認された。特に連接表が巨大な場合、コスト参照にかかる時間の割合が大きく、コスト参照のキャッシュ効率改善が解析の高速化に有効であった。

7）https://github.com/daac-tools/crawdad

表 4　形態素解析器の比較実験の結果。

ライブラリ	辞書	解析時間 [ms]	標準偏差 [ms]
Vibrato	mecab-ipadic	378	9.8
Vibrato	unidic-cwj	687	13.6
MeCab	mecab-ipadic	786	6.1
MeCab	unidic-cwj	1,562	11.8
Lindera	mecab-ipadic	1,027	17.5
Lindera	unidic-mecab	1,622	26.7
sudachi.rs	core	1,674	17.3

5.3　形態素解析器の比較

提案法の実装である Vibrato の速度性能を、既存の形態素解析器と比較し評価する。比較したソフトウェアは以下の 4 種類である。

- Vibrato v0.1.2
- MeCab v0.996[8]
- Lindera v0.16.1[9]
- sudachi.rs v0.6.4-a1[10]

Vibrato と MeCab は ipadic-mecab と unidic-cwj を用いて評価した。Lindera はライブラリでサポートされている ipadic-mecab と unidic-mecab を用いて評価した。sudachi.rs は形態素解析器 Sudachi [3] の Rust 移植であり、辞書には SudachiDict-core[11] を用いて評価した。計測には tokenizer-speed-bench[12] を用いた。

表 4 に結果を示す。IPADIC と UniDic の両方のケースについて、Vibrato が最速であることが確認できた。二番目に高速な MeCab と Vibrato を比較して、IPADIC で 2.1 倍、UniDic で 2.3 倍高速である。

6　おわりに

本稿では、形態素解析の高速化を目的とし、最小コスト法のキャッシュ効率化技法を提案した。実験により提案法の有効性を実証した。また、提案法を実装した形態素解析器 Vibrato を開発し、その速度性能を実証した。今後は機能面と性能面について、Vibrato のさらなる改良を予定している。

8）https://taku910.github.io/mecab/
9）https://github.com/lindera-morphology/lindera
10）https://github.com/WorksApplications/sudachi.rs
11）https://github.com/WorksApplications/SudachiDict
12）https://github.com/legalforce-research/tokenizer-speed-bench

参考文献

[1] 久光徹, 新田義彦. 接続コスト最小法による日本語形態素解析. 全国大会講演論文集, 人工知能及び認知科学, pp. 1–2, 1991.

[2] Taku Kudo, Kaoru Yamamoto, and Yuji Matsumoto. Applying conditional random fields to japanese morphological analysis. In **Proceedings of the 2004 conference on empirical methods in natural language processing**, pp. 230–237, 2004.

[3] Kazuma Takaoka, Sorami Hisamoto, Noriko Kawahara, Miho Sakamoto, Yoshitaka Uchida, and Yuji Matsumoto. Sudachi: a japanese tokenizer for business. In **Proceedings of the 11th International Conference on Language Resources and Evaluation**, 2018.

[4] Masayuki Asahara and Yuji Matsumoto. ipadic version 2.7.0 User's Manual. https://ja.osdn.net/projects/ipadic/docs/ipadic-2.7.0-manual-en.pdf, 2003.

[5] 佐藤敏紀, 橋本泰一, 奥村学. 単語分かち書き辞書 mecab-ipadic-neologd の実装と情報検索における効果的な使用方法の検討. 言語処理学会第 23 回年次大会 (NLP2017), pp. NLP2017–B6–1. 言語処理学会, 2017.

[6] 伝康晴, 小木曽智信, 小椋秀樹, 山田篤, 峯松信明, 内元清貴, 小磯花絵. コーパス日本語学のための言語資源 : 形態素解析用電子化辞書の開発とその応用. 日本語科学, Vol. 22, pp. 101–123, 2007.

[7] 岡照晃. CRF 素性テンプレートの見直しによるモデルサイズを軽量化した解析用 UniDic — unidic-cwj-2.2.0 と unidic-csj-2.2.0 —. 言語資源活用ワークショップ 2017 発表予稿集, 143–152.

[8] 青江順一. キー検索技法–IV: トライとその応用. 情報処理, Vol. 34, No. 2, 1993.

[9] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. **IEEE transactions on Information Theory**, Vol. 13, No. 2, pp. 260–269, 1967.

[10] Jun'ichi Aoe. An efficient digital search algorithm by using a double-array structure. **IEEE Transactions on Software Engineering**, Vol. 15, No. 9, pp. 1066–1077, 1989.

[11] Huidan Liu, Minghua Nuo, Longlong Ma, Jian Wu, and Yeping He. Compression methods by code mapping and code dividing for chinese dictionary stored in a double-array trie. In **Proceedings of 5th International Joint Conference on Natural Language Processing**, pp. 1189–1197, 2011.

[12] 前川喜久雄. 代表性を有する大規模日本語書き言葉コーパスの構築 (<特集>日本語コーパス). 人工知能, Vol. 24, No. 5, pp. 616–622, 2009.

[13] 国立国語研究所コーパス開発センター. 『現代日本語書き言葉均衡コーパス』利用の手引 第 1.1 版. https://clrd.ninjal.ac.jp/bccwj/doc.html, 2015.

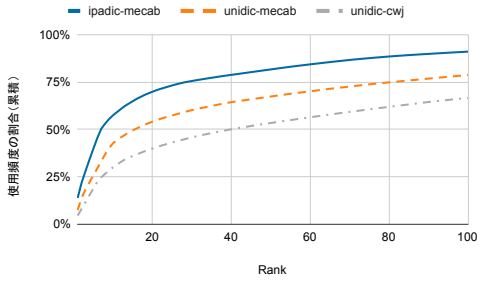

図 4　右文脈 ID の使用頻度上位 100 件の割合。値は累積を表し、上位 k 番目の値は 1 番目から k 番目までの割合の合計である。ipadic-neologd の結果は ipadic-mecab と僅差だったため省略している。

表 5　ダブル配列の実装を変化させた場合の解析時間 [ms]。括弧内の数値は Default からの変化率を表す。

辞書	Default	Ideal
mecab-ipadic	343	282 (–17.8%)
mecab-neologd	479	397 (–17.0%)
unidic-mecab	567	369 (–34.8%)
unidic-cwj	949	380 (–59.9%)

A　コスト参照についての事前実験

文脈 ID の使用頻度の調査. 文脈 ID の使用頻度の偏りについて調査する。各辞書について、訓練用テキストを解析したときの文脈 ID の使用頻度を算出した。右文脈 ID の使用頻度上位 100 件について、その割合を図 4 に示す。[13] 文脈 ID の使用頻度に大きな偏りが確認された。例えば、右文脈 ID が 15k 種類も定義されている unidic-cwj においても、上位 32 件が使用される割合は 47%である。

ボトルネックの調査. 巨大な連接表がボトルネックなのかを検証するために、連接表へのアクセスを完全に排除し、連接コスト値を全て 0 として解析した場合（**Ideal**）の結果とも比較する。表 5 に実験結果を示す。unidic-cwj で解析時間が 57%短縮した。定数時間の配列参照を取り除いただけの簡単な修正であることから、巨大な連接表への参照がいかにボトルネックであるかが推察できる。

B　訓練データに関する調査

文脈 ID の使用頻度について、均衡コーパスを用いて算出した場合と、ドメインに特化したコーパスを用いて算出した場合を比較し、訓練コーパスにより Mapped の性能が変化するのかを調査する。

実験設定. BCCWJ サブコーパスは、新聞（PN）や Yahoo!知恵袋（OC）などの 13 カテゴリから構成される [13]。各カテゴリから 10k 文ずつをサンプルし、13 種類の評価用データを構築する。そして、各カテゴリについて、以下の 2 つの方法でマッピングを訓練し、解析時間を比較する。

- 均衡データ：BCCWJ コアデータからサンプルし得られた 10k 文で訓練
- カテゴリ別：各カテゴリの評価用テキスト 10k 文について 5 分割交差検証

辞書には unidic-cwj を使用する。計算機環境など、その他の実験設定は 5 節と同じである。

13）　左文脈 ID の結果も大差は確認されなかった。

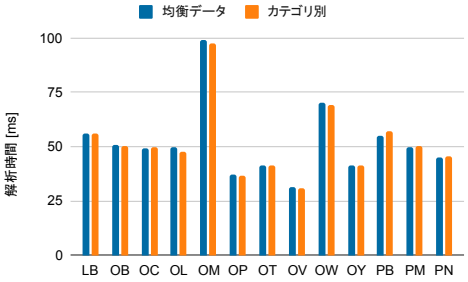

図 5　カテゴリ別の評価データについての解析時間。

表 6　各コーパスについて、頻繁に使用された右文脈 ID の上位 10 件。PN は新聞、LB は書籍、OC は Yahoo!知恵袋。ID は unidic-cwj で定義されている標準のものを表示。

ランク	コア	PN	LB	OC
1	850	850	850	850
2	6,742	6,742	6,742	6,742
3	9,663	9,663	9,663	9,663
4	11,540	11,540	11,540	11,540
5	14,497	14,497	14,497	14,497
6	2,250	2,250	2,250	2,250
7	11,383	11,383	11,383	11,383
8	6,980	6,980	14,775	14,775
9	12,392	12,392	12,392	2,410
10	14,775	3,917	2,410	12,392

解析時間に関する結果. 13 カテゴリについて、評価データ 10K 文の解析に要した時間を図 5 に示す。訓練データをドメインに特化しても、解析時間は大きく変化しないことが確認された。

高頻度な文脈 ID に関する分析. コアデータといくつかのカテゴリで頻繁に使用された右文脈 ID の上位 10 件を表 6 に示す。上位 7 件まではどのコーパスでも同一で、カテゴリによって使用される文脈 ID に大差は無い。

これらの右文脈 ID に対応した素性列を図 7 に示す。頻出 ID は品詞などの一部の情報によって識別可能な形態素に割り当てられている。そして、そのような形態素はドメインを問わずコーパスの大部分を占めている。

これらの結果は、「幅広いドメインのテキストを解析するために、均衡データから訓練したマッピングを一つ持っておけば十分である」ということを示唆している。つまり辞書を配布する場合、１種類のマッピング適用済みの辞書を配布すればよい。

表 7　unidic-cwj で頻繁に使用された素性列の例。

右文脈 ID	素性列
850	名詞, 普通名詞, サ変可能,*,*,*,*,*,*,*,*,*
6,742	名詞, 固有名詞, 人名, 一般,*,*,*,*,*,*,*,*
9,663	名詞, 固有名詞, 地名, 一般,*,*,*,*,*,*,*,*
11,540	名詞, 固有名詞, 一般,*,*,*,*,*,*,*,*
14,497	名詞, 普通名詞, 一般,*,*,*,*,*,*,*,*
2,250	感動詞, 一般,*,*,*,*,*,*,*,*,*
11,383	名詞, 普通名詞, 一般,*,*,*,*,*, 和,*,*,1,C3,*
6,980	名詞, 普通名詞, 一般,*,*,*,*,*, 漢,*,*,1,C3,*
12,392	接尾辞, 名詞的, 一般,*,*,*,*,*, 漢,*,*,*,C3,*
14,775	副詞,*,*,*,*,*,*,*, 和,*,*,*,1,*,*
3,917	名詞, 固有名詞, 人名, 名,*,*,*,*, 固,*,*,*,1,*,*
2,410	感動詞, 一般,*,*,*,*,*,*, 和,*,*,*,1,*,*