統計的日英翻訳における依存構造に基づく事前並べ替えルール

丁 塵辰　酒主 佳祐　通事 寛奈　山本 幹雄
筑波大学 システム情報工学研究科
{tei,nushi,touji}@mibel.cs.tsukuba.ac.jp myama@cs.tsukuba.ac.jp

1　はじめに

日本語と英語のような語順が大きく相違する言語では、統計的機械翻訳（SMT）における語順並べ替えが問題点となっている。解決策の一つとしては、原言語文の語順を目的言語文の語順に並べ替えてから既存のSMT システムを適用するという事前並べ替え手法がある。事前並べ替え手法は、規則に基づく手法と統計に基づく手法の 2 種類に分けられる。前者は原言語・目的言語に対して言語学的な知識を用い人手で並べ替えルールを作成する。後者は統計的な並べ替えモデルを設計し大量のデータから学習する。規則に基づく手法は言語対ごとにルールの設計が必要とされ、強力かつ網羅的なルールを作成することはコストがかかる。しかし、効果的なルールを一度作ってしまえば、実行が高速で簡単に利用できる。統計に基づく手法は汎用性に富んでいるが、データからのモデル学習と学習済みモデルでの並べ替えに時間がかかり、性能が学習データに左右されるというが欠点がある。

本稿では、規則に基づく手法に注目し、3 種類の文節並べ替えルールと 2 種類の形態素移動ルールを提案した。実験により提案手法の有効性を示した。

2　先行研究

Head Finalization[1] は代表的な英日翻訳における規則に基づく事前並べ替え手法である。これは主辞駆動句構造文法（HPSG）に基づき、日本語の SOV といった語順に特化したルールで英日の語順調整問題をほぼ解決した。一方、この手法は日英翻訳には適用できない。日英翻訳でも多くの規則に基づく事前並べ手法が考案されている。例えば、形態素「は」に注目する手法 [2]、依存構造を利用する手法 [2, 3]、および述語項構造を利用する手法 [4, 5] が挙げられる。文構造解析の視点から、形態素のみを利用する手法 [2] は最も軽量であり、述語項構造を利用する手法 [4, 5] は深層の意味解析まで行うため処理が重くなる。本稿では [2]、[3] のように依存構造を利用した並べ替えルールを設計した。依存構造は既存の解析器で比較的高速・正確に得ることができる。

統計に基づく手法では、LADER[6] が代表的な手法であり、日英・英日翻訳上の有効性が示されている。独英翻訳で有効性が示されている手法 [7] もある。これらの手法は単語対応付き対訳文対から事前並べ替えモデルを学習する。学習データには自動生成されたデータより、人手で作成 [6] または機械で精選されたデータ [7] を用いることで良い性能が達成できる。

3　提案手法

従来の依存構造を利用する手法におけるルールは、主に「inter-chunk」と「intra-chunk」2 種類に分類できる。前者で文節[1]間、後者で文節内の形態素を並べ替える。しかし、形態素の移動をそれぞれの文節内に制限することで、正確な並べ替えができないケースが多い (図 1)。よって、提案手法では「inter-chunk」ルールを設ける一方、「intra-chunk」に限らず、「intra/extra-chunk」というさらに形態素の自由な移動を可能にするルールを提案する。Algorithm 1 は処理の流れを示す。1–2 行目の **for** 文は「inter-chunk」ルールによる文節間の並べ替え処理であり、3–4 行目の **for** 文は「intra/extra-chunk」による形態素の移動である。

文節間の並べ替えには、「動詞」、「名詞」、「コピュラ」の 3 種類の「inter-chunk」ルールを設計した。Algorithm 2 で示したように文節内の主辞である形態素の品詞により Algorithm 3–5 で示した各ルールを適用する。先行研究では、動詞文節の移動を中心にし、名詞に関する並べ替えがシンプルな「head-initialization」[2] のみを利用するものが多い。一方、英語では従属文は名詞の後ろに来るが、限定詞などは日本語の連体詞に相当し、同じく名詞の前に来る。これらの現象に正

[1] 本稿では、「文節」と「chunk」を同じ意味にする。

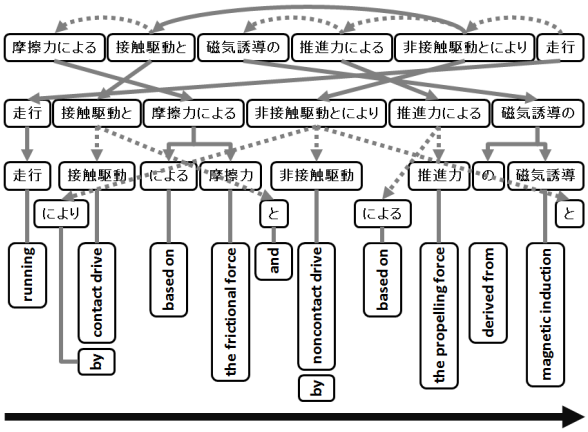

図 1: 日英翻訳における日本語文の事前並べ替えの一例。1 段から 2 段は「inter-chunk」の並べ替え（依存構造の破線矢印を反転する操作）。2 段から 3 段は形態素の並べ替え。子がない文節（「摩擦力による」、「磁気誘導の」）には、「intra-chunk」の移動（2，3 段間の実線矢印）のみで十分だが、子文節を持つ文節（「非接触駆動とによる」など）に対して、文節の分割により「extra-chunk」の移動（2，3 段間の破線矢印）がなければ正確な並べ替えが行えない。

確に対応するために、名詞ルールを設けた。さらに、日本語では、コピュラが名詞に形態素（「だ」、「です」、「である」）を接続することで表されるので、文節の主辞形態素が名詞ではあるが動詞のように扱う「コピュラ」ルールも設計した（Algorithm 2 の 4 行目の **if** 文で判別する）。

Algorithm 1 Over all process

Require: Dependency tree D

1: **for** each head chunk H in D **do**
2: reorder H and its modifier chunks by Algorithm 3, 4, 5, according to the type of H by Algorithm 2
3: **for** each head chunk H in D, top-down-wisely **do**
4: reorder morphemes of H by Algorithm 7, according to the segmentation of H by Algorithm 6

Algorithm 2 Chunk type

Require: chunk C

1: **if** C.head.func is "動詞" **then**
2: C is a Verb-chunk
3: **if** C.head.func is "名詞" **then**
4: **if** no copula morphemes succeeding **then**
5: C is a Noun-Chunk
6: **else**
7: C is a Copula-Chunk

形態素の並べ替えには、従来の「intra-chunk」移動に「extra-chunk」移動を加えて 1 つの枠組みにまとめた。Algorithm 6 のように、ある文節に対して、その外部の支配範囲の左・右端に移動すべき形態素（$ExPre/ExPost$）とその内部の左・右端に移動すべき形態素（$InPre/InPost$）を同定してから、Algorithm 7 を top-down に適用して形態素の移動を行う。具体的には英語の語順と一致させるように設計した。「用

Algorithm 3 Inter-chunk rule for Verb-chunk

Require: Verb-chunk H, set M of H's modifier chunks

1: $Sup, Pre, Core, Post \leftarrow [\,], [\,], [\,], [\,]$
2: **for** each chunk C in M **do**
3: **if** C.func.POS is "接続詞" **then**
4: Sup.append (C)
5: **for** each chunk C in M **do**
6: **if** C.func.(POS, lex) is ("助詞, 格助詞","が") **then**
7: $Core$.append (C)
8: $Core$.append (H)
9: **for** each chunk C in M **do**
10: **if** C.func.(POS, lex) is ("助詞, 格助詞","を") **then**
11: $Core$.append (C)
12: **for** each chunk C in M **do**
13: **if** C.func.(POS, lex) is ("助詞, 格助詞","に") **then**
14: $Core$.append (C)
15: **for** each chunk C in M **do**
16: **if** C is **not** in Sup **or** $Core$ **then**
17: **if** C.func.(POS, lex) is ("助詞, 係助詞","は") **or** C.func.POS is "助詞, 接続助詞" **or** C.end is "、" **then**
18: Pre.append (C)
19: **for** each chunk C in M **do**
20: **if** C is **not** in Sup **or** $Core$ **or** Pre **then**
21: $Post$.append (C)
22: **return** $Sup + Pre + Core + Post$

Algorithm 4 Inter-chunk rule for Noun-chunk

Require: Noun-chunk H, set M of H's modifier chunks

1: $Sup, Pre, Core, Post \leftarrow [\,], [\,], [\,], [\,]$
2: *Proc. of line 2 to 4 in Algorithm 3*
3: **for** each chunk C in M **do**
4: **if** C.func.POS is "並立助詞" **then**
5: $Core$.append (C)
6: **for** each chunk C in M **do**
7: **if** C.func.POS is "連体詞" **then**
8: $Core$.append (C)
9: **for** each chunk C in M **do**
10: **if** C.func.POS is "形容詞" **then**
11: $Core$.append (C)
12: **for** each chunk C in M **do**
13: **if** C.func.POS is "名詞" **then**
14: $Core$.append (C)
15: $Core$.append (H)
16: **for** each chunk C in M **do**
17: **if** C.func.(POS, lex) is ("助詞, 連体化","の") **then**
18: $Core$.append (C)
19: *Proc. of line 15 to 21 of Algorithm 3*
20: **return** $Sup + Pre + Core + Post$

Algorithm 5 Inter-chunk rule for Copula-chunk

Require: Copula-chunk H, set M of H's modifier chunks

1: $Sup, Pre, Core, Post \leftarrow [\,], [\,], [\,], [\,]$
2: *Proc. of line 2 to 8 of Algorithm 3*
3: *Proc. of line 15 to 21 of Algorithm 3*
4: **return** $Sup + Pre + Core + Post$

言」に対して基本的に文節内で形態素を移動するが、後続する「接続助詞」は支配範囲の一番先頭に移動する（Algorithm 6 の 3 行目の **if** 文)。「体言」に対して基本的に接続した形態素は支配範囲の一番先頭に移動するが、「並立助詞」は支配範囲の一番後ろに移動する（Algorithm 6 の 8 行目の **if** 文)。さらに句読点などは、該当範囲の最後に移動する。

Algorithm 6 Chunk Segmentation

Require: chunk C
1: $Core \leftarrow$ from the first to the head morph. of C
2: $ExPre, InPre, InPost, ExPost \leftarrow [\,], [\,], [\,], [\,]$
3: **if** C is Verb/Copula-chunk **or** C.head.POS is "形容詞" **then**
4: **if** C.func.POS is "助詞, 接続助詞" **then**
5: $ExPre$.append (C.func)
6: **else**
7: $InPre$.append (C.func)
8: **if** C is Noun-chunk **then**
9: **if** C.func.POS is "助詞, 並立助詞" **or** "接続詞" **then**
10: $ExPost$.append (C.func)
11: **else**
12: $ExPre$.append (C.func)
13: $InPost \leftarrow$ all the other unprocessed morph. in C

Algorithm 7 Intra/Extra-chunk rule for Morpheme

Require: Inter-chunk reordered tree D_r by line 1 to 2 of Algorithm 1, each chunk C in D_r is segmented by Algorithm 6
1: **for** each chunk C in D_r **do**
2: $L_m \leftarrow$ the most left modifier chunk of C in D_r
3: $R_m \leftarrow$ the most right modifier chunk of C in D_r
4: $L_m.ExPre \leftarrow C.ExPre + L_m.ExPre$
5: $R_m.ExPre \leftarrow R_m.ExPost + C.ExPost$
6: $C.Core \leftarrow C.InPre + C.Core + C.InPost$

4 評価実験

4.1 実験設定

提案した手法を NTCIR7 対訳コーパス [8] を用いた翻訳実験で評価した。[2] の形態素を利用する手法（M./Morph.）、[2]、[3] の依存構造を利用する手法（D./Dep.）、[4]、[5] の述語項構造を利用する手法（PAS)、及び LADER[6][2]を比較手法とした。

コーパスは MeCab[3] で形態素解析を行った。提案手法・[2]・[3] に利用した依存構造は CaboCha[4] の出力を用い、[4]・[5] に利用した述語項構造は SynCha[5] の出力を用いた。LADER の学習データには、[7] の対訳文精選手法を利用し、NTCIR7 のトレーニングセット上の 10〜30 単語の文と 31〜50 単語の文から、それぞれ上位 500 文を選び、計 1,000 文とした。

実験には日本語の IPA 品詞体系を利用して、前述のアルゴリズムに以下のような調整を行った。

- 「形容詞」は「イ形容詞」と「ナ形容詞」を含む。
- 「接続助詞」には「て」を除外。
- Algorithm 3 の 6, 10, 13 行目は「形式名詞」を除外。
- 「代名詞＋の」は「連体詞」にする。

翻訳には Moses[6] の Phrased-Based（PB）システムを利用した。翻訳モデルは「grow-diag-final-and」を用い、フレーズの単語数を 5 までにした。SRILM[7] で NTCIR7 トレーニングセット英語側上の Interpolated Modified Knense-Ney 5-gram 言語モデルを作成した。デコーディングは ttable-limit を 10 に、stack を 100 にした。915 文の dev セットで MERT を行い、1,381 文の fmlrun セットをテストセットにし、評価を行った。

4.2 実験結果と考察

表 1 は語順の一致性を示す指標である Kendall's τ[5] で、トレーニングセット上の各事前並べ替え手法の性能を比較し、提案手法の並べ替え性能を確認した。

表 2 はデコーディング時の distortion-limit（DL）の変化に従い、ベースライン（事前並べ替え無し)・各比較手法・提案手法のテストセット上の BLEU と RIBES を示している。提案手法の最高性能が各比較手法より上回っている。規則に基づく各既存手法において、[2] の形態素「は」のみに注目する手法は一番単純だが、DL の調整が必要となる。[5] の述語項構造を利用する手法は [2] と同じくらいの最高性能に達しているが、比較的小さい DL（6）の場合でも性能が劣化しない。これはより正確な並べ替えが実現されたからだと考えられる。一方、統計に基づく LADER[6] は言語学的な知識を用いず規則に基づく手法と同じ性能に達している。LADER の強力さが示されている。

表 3 に提案手法に利用していたルールを減らした場合の翻訳性能の推移を示す。比較的小さい DL の場合、性能劣化が顕著である。すなわち、並べ替えルールは小さい DL 時の翻訳性能への貢献が大きいと言える。

表 1: トレーニングセット上の Kendall's τ[5] の平均。（BL. は事前並べ替え無しのベースライン。）

BL.	M.[2]	D.[2]	[3]	[4]	[5]	[6]	Prop.
.489	.504	.484	.369	.444	.619	.618	**.671**

[2] http://www.phontron.com/lader/
[3] http://mecab.googlecode.com/svn/trunk/mecab/doc/index.html
[4] https://code.google.com/p/cabocha/
[5] http://www.cl.cs.titech.ac.jp/~ryu-i/syncha/
[6] http://www.statmt.org/moses/
[7] http://www.speech.sri.com/projects/srilm/

表 2: テストセット上の BLEU/RIBES。（DL はデコーディング時の distortion limit。太字は各列の BLEU/RIBES の最高値。提案手法の最高 BLEU 値（30.8）は他の比較手法の最高値と比べ $p \leq 0.01$ レベルの有意差がある（bootstrap 法による）。）

DL	Baseline	Morph.[2]	Dep.[2]	Dep.[3]	PAS[4]	PAS[5]	LADER[6]	Proposed
0	24.6/.645	23.6/.680	22.3/.672	24.8/.615	25.1/.651	28.2/.704	28.9/.705	30.3/.732
3	24.7/.648	25.5/.689	24.3/.679	25.1/.615	25.3/.652	28.1/.706	29.1/.705	30.2/.731
6	27.0/.662	28.6/.707	28.1/.702	26.6/.624	27.0/.667	29.1/.710	29.4/.707	30.2/.733
9	28.0/.672	**29.7**/**.715**	**28.6**/**.704**	27.5/.635	**28.3**/.681	**29.7**/**.716**	**29.8**/**.708**	30.6/**.735**
12	**28.6**/**.681**	29.3/.713	28.3/.702	**27.8**/**.641**	28.1/**.683**	29.4/.715	**29.8**/**.708**	**30.8**/.733
∞	28.5/.648	29.0/.670	27.4/.649	28.1/.630	**28.3**/.647	29.2/.679	29.1/.654	29.9/.697

表 3: 提案手法の各ルールと翻訳精度の関係。（テストセット上の BLEU/RIBES。VNC は 3 種類の「inter-chunk」ルール、$_{E \to I}$ は「extra-chunk」を「intra-chunk」に（Algorithm 6 の $ExPre$ と $InPre$、$ExPost$ と $InPost$ を合併）、$_{C \to N}$ は「コピュラ」ルールを「名詞」ルールに（Algorithm 2 の 4 行目の **if** 文を無効に）。）

DL	$\text{VNC}_{E \to I}$	VNC	$\text{VN}_{C \to N}$	V
0	29.1/.727	28.2/.720	27.7/.717	25.1/.664
3	29.2/.725	28.4/.720	28.0/.717	25.3/.664
6	29.7/.728	28.8/.724	29.1/**.723**	27.0/.678
9	30.2/**.731**	29.4/**.727**	**29.4**/**.723**	28.4/.688
12	**30.4**/.730	**29.5**/.723	29.1/.721	**28.5**/**.689**
∞	29.4/.679	28.3/.672	28.0/.664	28.1/.649

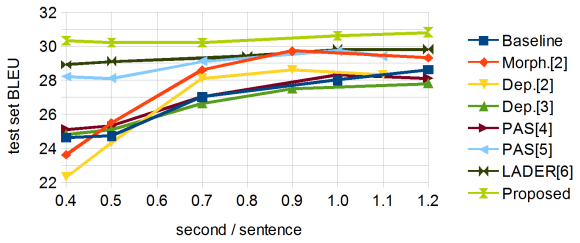

図 2: 各手法のテストセット上の文平均翻訳時間と BLEU の関係。（各線上の点は左から DL= 0/3/6/9/12）

翻訳速度は、Moses の PB システムは手法に関わらず DL のみに影響される。テストセット上の文平均翻訳時間は DL が 0/3/6/9/12/∞ の場合ではそれぞれ約 0.4/0.5/0.7/1.0/1.2/3.6 秒である。図 2 で翻訳速度と精度の関係を示す。Python の実装で各手法の事前並べ替え速度は、テストセット上の文平均処理時間が共に 0.002 秒以下であり、翻訳時間と比べ無視できる。依存構造解析に利用した CaboCha も速く、一文あたり 0.001 秒の速度で解析を行う。一方、述語項構造解析は時間がかかり、一文あたり 0.9 秒という翻訳時間と同等の処理時間となった。LADER について、前述した 1,000 文でのモデル学習[8]には約 3 日がかかった。学習済みのモデルで、トレーニングセット上の単語数が $(0, 30]/(30, 50]/(50, \infty)$ の文セット[9]の事前並べ替え[10]に、それぞれ約 3 日/10 日/3 週間かかった。LADER の短所は処理時間にあることが分かる。

[8]LADER のデフォルト設定。
[9]それぞれ約 $0.8M/0.7M/0.3M$ 文
[10]LADER のデフォルト設定、8 スレッドで処理。

5 おわりに

本稿では、依存構造を用いる統計的日英翻訳における事前並べ替えルールを提案し、実験により提案法の有効性を示した。これからの課題は、新聞コーパス上の実験・日中翻訳に適用することなどが挙げられる。

謝辞

[4] および [5] の実験用 Python スクリプトを提供して頂いた星野翔氏に誠に感謝申し上げます。

参考文献

[1] Hideki Isozaki, Katsuhito Sudoh, Hajime Tsukada, and Kevin Duh. HPSG-based preprocessing for English-to-Japanese translation. *ACM Transactions on Asian Language Information Processing*, Vol. 11, No. 3, 2012.

[2] Jason Katz-Brown and Michael Collins. Syntactic reordering in preprocessing for Japanese → English translation: MIT system description for NTCIR-7 patent translation task. In *Proc. of NTCIR*, pp. 409–414, 2008.

[3] Katsuhito Sudoh, Kevin Duh, Hajime Tsukada, Masaaki Nagata, Xianchao Wu, Takuya Matsuzaki, and Jun'ichi Tsujii. NTT-UT statistical machine translation in NTCIR-9 PatentMT. In *Proc. of NTCIR*, pp. 585–592, 2011.

[4] Mamoru Komachi, Yuji Matsumoto, and Masaaki Nagata. Phrase reordering for statistical machine translation based on predicate-argument structure. In *Proc. of IWSLT*, pp. 77–82, 2006.

[5] Sho Hoshino, Yusuke Miyao, Katsuhito Sudoh, and Masaaki Nagata. Two-stage pre-ordering for Japanese-to-English statistical machine translation. In *Proc. of IJCNLP*, pp. 1062–1066, 2013.

[6] Graham Neubig, Taro Watanabe, and Shinsuke Mori. Inducing a discriminative parser to optimize machine translation reordering. In *Proc. of EMNLP-CoNLL*, pp. 843–853, 2012.

[7] Jirí Navrátil Karthik Visweswariah Ananthakrishnan Ramanathan. A comparison of syntactic reordering methods for English-German machine translation. In *Proc. of COLING*, pp. 2043–2058, 2012.

[8] Atsushi Fujii, Masao Utiyama, Mikio Yamamoto, Takehito Utsuro, Terumasa Ehara, Hiroshi Echizen-ya, and Sayori Shimohata. Overview of the patent translation task at the NTCIR-7 workshop. In *Proc. of NTCIR*, pp. 389–400, 2008.