括弧制約の下での多段階まとめあげ統語解析[1]

宮田高志 † 橋田浩一 †‡
miyata.t@carc.aist.go.jp hasida.k@aist.go.jp
† 独立行政法人 科学技術振興機構, CREST
‡ 独立行政法人 産業技術総合研究所 サイバーアシスト研究センター

1 はじめに

現在使われている多くの自然言語解析システムは、形態素解析・統語解析・意味解析・背景知識に基づく推論といったモジュールを、逐次的に動作させるようなアーキテクチャをとっている。しかし多くの研究者が指摘してきたように、逐次的なアーキテクチャは後の処理のために膨大な曖昧性を保持しなければならず、効率が悪い。効率的に解析を行なうためには、背景知識を使って統語的曖昧性を解消したり、統語的構造を使って形態素の境界を制限したりすることができるように、各モジュールが同時並行的に動作すべきである。

また、最近の自然言語処理研究では固有名詞抽出や言い換えなど、先に挙げた分類にまたがるような部分問題も研究されている。これらの処理では複数の解析モジュールから得られる情報を利用しているため、統語解析や意味解析にこれらの処理結果をとり入れる時には処理が重複しないような工夫が必要となる。

このような問題を解決するために、多くの研究者が各解析処理を統一的に扱えるような枠組を提案してきた。制約論理や確率モデルに基づくもの [11, 4, 5] は理論的な背景が堅固であるので拡張もしやすく、有望であるように思われる。しかし、全てのモジュールを融合した単一の巨大なシステムを構築するというのは設計と保守の複雑さという観点から見て、現実的ではない。先の各解析における様々な知識はすでに専門化されており、一人の研究者が全てを網羅するような研究を行なうことはほとんど不可能である。これはシステムも分業で構築しなければならないことを意味する。実際、これらの解析を行なうシステムはすでに多くの研究者によってツールとして開発・公開されており、自分の専門以外の解析を行ないたい時はそれらのツールをブラックボックスとして使わざるを得ない状況になっている。

このように、さまざまな解析モジュールが互いに独立に開発されている状況において、各解析モジュールをできるだけ少ない修正で有機的に連携させるために、本論文では解析モジュール間の情報交換の基礎として**部分構造の指定**を利用することを提案する。これは、先に挙げたいくつかの解析において、ツールが整備されるくらい広く受け入れられている理論は概ね階層的な構造の概念を持っていることに基づく。具体的には、各ツールを、部分構造を明示する括弧を含んだ入力を受け付け、括弧に矛盾せずかつほとんど確実な解 (だけ) を、入力に対する括弧を付与する形で出力するように修正する。これによって不十分な情報のために大量の曖昧性を展開する必要はなくなり、固有名詞抽出のように複数のモジュールからの情報を利用するような処理も容易に統合可能である。

さらに、部分構造の指定は解析モジュール間の情報交換だけでなく、ユーザと解析システムとのインターフェースとしても役に立つ。例えば、近年、統計的手法に基づいた研究が盛んになるにつれて解析済みコーパスの需要が高まっているが、その作成にはあらかじめ自動解析した結果を人手で修正するという方法がとられることが多い。しかし自動解析の結果は多くの場合情報が詳細すぎて人間がチェックするのは困難であり、むしろ自動解析で間違えそうなところに先に括弧を付与する方が作業効率は高い。また、機械翻訳システムの誤りの原因が入力の解析エラーだということが分かれば、ユーザ自身が入力に括弧を付与することで、適切な翻訳を得られる可能性がある。一般に、各応用において解析器の辞書や文法を修正することはシステム全体への影響を考えるとためらわれることが多く、「この文だけ正しく解析できればよい」という場合も多い。

2 括弧を考慮した解析

本論文では hidden Markov model (HMM) を言語モデルとした形態素解析・文節まとめあげ解析・文節係り受け解析に対して、括弧で制約された入力を言語モデルに矛盾することなく扱えるように拡張する方法を示す。具体的には、次の三つの解析システムを拡張した[2]:

- 茶筌 [9] (形態素解析)
- YamCha[7] (文節まとめあげ解析)
- 南瓜 [8] (文節係り受け解析)

以下では簡単のため、最尤解だけを計算する場合を考えるが、複数解を計算するように修正することは容易である。

2.1 括弧制約を考慮した形態素解析

HMM を言語モデルとする形態素解析アルゴリズムでは、図 1 のような動的計画法が用いられる。ここで、(m_0, p_0) は文頭を表す特殊な形態素である[3]。

形態素より細かい括弧を付与することは考えないとすると、このアルゴリズムに括弧制約を導入するためには、D' を計算している $(*)$ の部分において、括弧をまたがないように $(c_i, \ldots, c_j)$ を制限すればよい。さらに次のように $(*)$ を修正すれば、() のかわりに [ ] を使うことで、形態素の区切りだけでなく「ちょうど形態素一つ分」の範囲も指定できる。

[1] Cascaded Chunk Parsing under Bracketing Constraints
MIYATA Takashi† HASIDA Kôiti†‡
† Japan Science and Technology Agency, CREST
‡ National Institute of Advanced Industrial Science and Technology

[2] YamCha, 南瓜は各ラベルを決定的に付与し、最尤解を計算しないので、厳密には HMM ではない。3 節の実行例で実装したシステムも YamCha, 南瓜をそのまま拡張したものになっているが、本論文で説明するように修正することは容易である。

[3] 遷移確率を一般の n-gram に拡張するには 2.2 節を参照。

```
if (c_i の左に '[' がある)
  D' = {((c_i, ..., c_k), p')};
  /* c_k は最初の ']' の直前の文字で、
     ((c_i, ..., c_k), p') ∈ D */
else
  D' = {(m', p') |m' = (c_i, ..., c_j),
                  (m', p') ∈ D, i < j ≤ n};
  /* ただし (c_i, ..., c_j) は括弧をまたがない */
```

2.2 括弧制約を考慮した文節まとめあげ解析

文節まとめあげ解析は、各形態素に B または I のラベルを付与することに帰着することで、HMM を言語モデルとすることができる [7, 12]。ここで B は文節の開始を、I は文節の途中もしくは最後を意味する。例えば、「逐次調査を依頼する」という文の二通りの解析は次のラベルに対応する:

(逐次 調査 を) (依頼 する)
B I I B I

(逐次) (調査 を) (依頼 する)
B B I B I

形態素解析では形態素より細かい括弧を付与することを考えなかったが、文節まとめあげ解析ではこのような仮定は適切とはいえない。形態素に関する括弧が入る時と入らない時があるだけでなく、文節内部の構造についても指定したいことがあるためである。このことは次のような問題を引き起こす:

入力	w_1((w_2	w_3	w_4)	w_5 (	w_6)	w_7)	w_8
可能なラベル	B	B	B	B,I	B	B,I	B,I	B
	B	B	I	B	B	B,I	B,I	B
	B	B	I	I	B	B,I	B,I	B
	B	B	I	I	I	B	B,I	B
	B	B	I	I	I	I	B	B
	B	B,I	I	I	I	I	I	B,I

上の場合、w_2 が I の時は w_3 から w_7 までのラベルも I でなければならないが、w_2 が B の時は必ずしもそうではない。このようにもとのアルゴリズムでは局所的にラベルを決めることができたのに対し、括弧を考慮すると局所的な条件だけではラベルを決めることはできなくなってしまう。しかし次のように考えると、わずかな修正で括弧制約を導入することができる。

1. 括弧と文節境界が交差するのは、対になった括弧のうち、(a) 開き括弧だけが文節に含まれるか、(b) 閉じ括弧だけが文節に含まれるか、の二つの場合しかない。

   a. … | … (… | …) …
   b. … (… | …) … | …

   いずれの場合も括弧の中に文節境界が入っている。逆に、括弧の中に文節境界がなければ (その括弧については) 文節境界と交差することはない。

2. 一方、括弧の中に文節境界が入っていても括弧と文節が交差しないためには、括弧の両端にも文節境界を設けなければならない。

   … |(… | …)| …

   つまり、括弧と文節境界が交差しないための必要十分条件は、(1) が真となることである。

   括弧の中に文節境界があるならば、括弧の両端にも文節境界がある。 (1)

3. (1) の対偶をとると、(2) となる。

   括弧の両端のどちらかに文節境界がないならば、括弧の中にも文節境界はない。 (2)

(1), (2) をもとに文 $\vec{S}$ の左から右へラベルを決めるアルゴリズムを作ると図 3 のようになる。ただし、括弧も一形態素とみなしていることに注意されたい。ここで、文脈 $\vec{c}_i$ は位置 i の "周辺にある" 品詞付き形態素の列、$\oplus$ は履歴の更新を表し、例えば $\Pr(L \mid \vec{c}, \vec{h})$ が 3 次の Markov 過程だとすると、以下のようになる[4]:

$$\vec{h}_i = (L_{i-2}, L_{i-1}, L_i)$$
$$\vec{h}_{i+1} = \vec{h}_i \oplus L_{i+1} \ = \ (L_{i-1}, L_i, L_{i+1})$$

また、`assign(tbl,prv,i,`$\vec{h}$`,L)` は位置 `i` までの部分解をもとに `i+1` 番目の形態素にラベルを付与する関数であり、添字 `i` と直前の履歴 $\vec{h}$ の対 `(i,`$\vec{h}$`)` を格納する。`depth` は連続する開き括弧の深さを表す変数であり、`depth>0` かどうかで開き括弧の直後かどうかのフラグも兼ねている。`start` は閉じ括弧の直後かどうかを表すフラグである。

2.3 括弧制約を考慮した多段階まとめあげ解析

2.3.1 単純な方法の問題点

括弧が明示された入力と矛盾しない解析を行なうためのもっとも単純な方法として、括弧ごとに解析器を再帰的に呼び出すことが考えられる。例えば、"I saw (a girl with a telescope) on Monday." という入力に対して、

- a girl with a telescope
- I saw a girl on Monday.

という二度の解析を行ない、二つ目の入力中の a girl の部分に最初の解析結果を埋め込む[5]。しかしこの方法には次のような問題がある:

- 切り出した部分は文とは限らない。文頭・文末を特別扱いするような構文解析器では、文以外の名詞句や (主語のない) 動詞句が入力された時に正しい結果が得られるという保証はない。
- 最近の統計的構文解析器では適用する規則の優先度を計算するために、親と子だけでなく孫や曾孫の情報まで参照する [3, 8]。上の例では、$S \leftarrow NP, V, NP_0, ADP.$ および $NP \leftarrow NP_0, ADP.$ という規則の優先度の計算に NP_0 の構造まで用いることがあるので、この部分が a girl である場合と a girl with a telescope である場合とでは、一般にこれらの規則の優先度は異なる。

2.3.2 多段階まとめあげ解析

多段階まとめあげ解析 (cascaded chunking) は、入力された単語列に対してまとめあげ (chunking) を何度も行なうことで、階層的な構造を決

[4] 文頭と文末の履歴 ($\vec{h}_0, \vec{h}_1, \ldots$ および $\vec{h}_n, \vec{h}_{n-1}, \ldots$) は適当な記号を補って長さをそろえるものとする。

[5] 句構造文法が与えられていれば、$M \leftarrow D_1, D_2, \ldots, D_n.$ のような規則に対して、$M \leftarrow (D_1, D_2, \ldots, D_n).$ のような規則を追加することとほぼ同じである。

```
* 0 4D 0/2 0.081032
車 名詞-一般
で 助詞-格助詞-一般
は 助詞-係助詞
* 1 2D 0/1 1.765846
こ 名詞-一般
を 助詞-格助詞-一般
* 2 3D 0/0 2.753827
運ぶ 動詞-自立
* 3 4D 0/1 0.000000
人 名詞-一般
を 助詞-格助詞-一般
* 4 -1O 0/0 0.000000
追う 動詞-自立
EOS
```

```
* 0 4D 0/1 2.383342
車 名詞-一般
で 助詞-格助詞-一般
* 1 2D 0/1 1.667978
* # <y>
* # <x mph="名詞-…">
はこ 名詞-一般
* # </x>
を 助詞-格助詞-一般
* 2 3D 0/0 2.753827
運ぶ 動詞-自立
* 3 4D 0/1 0.000000
人 名詞-一般
を 助詞-格助詞-一般
* 4 -1O 0/0 0.000000
追う 動詞-自立
* # </y>
EOS
```

図 4: 「車ではこを運ぶ人を追う」(左) および「車で<y><x mph="名詞-…">はこ</x>を運ぶ人を追う</y>」(右) の解析結果。「* 番号 …」で始まる行が文節の境界を表し、各数字は左から文節番号・係り先・主辞位置 / 関係語位置 (タグの行は除いて数える)・スコアを表す。

定する解析方法である [2]。図 2 で、一番外側の `while` ループの最後でチャンクの数を確認し、解析前と同じすなわち全ての文節に B ラベルが付与された時は、括弧と矛盾しないような最も右の文節のラベルを強制的に I に変更することに注意されたい。これは head final である日本語に固有のヒューリスティクスである。また、`assign(tbl,prv,i,`$\vec{h}$`,L)` は図 3 で定義したものと同じ関数で、文節列 $\vec{c}_{i+1}$ は直前 ($k-1$ 回目) のまとめあげにおいて作成された文節から計算されるものとする。

文節係り受けでは、形態素解析の時と同じく、係り受けより細かい解析は (すでに文節まとめあげ解析で済んでいるので) 考えなくてよい。このため、一般性を失うことなく、開き括弧および閉じ括弧の直後の文節は必ず B ラベルと決めてしまってもよく、アルゴリズムも図 2 中で (∗) の部分を、括弧の直後の場合は B だけに制限するだけでよい。

3 拡張したシステムについて

図 4(左) は「車ではこを運ぶ人を追う」を茶筌・YamCha・南瓜で解析させた結果である。この文に対して「はこ」がちょうど形態素一つであることを示すタグ <x> (品詞は mph という属性で指定) と、「車で」が「運ぶ」ではなく「追う」に係ることを示すためのタグ <y> を付与して解析すると図 4(右) が得られる[6]。なお、図 4 に示した通り、拡張した解析システムの出力は構文木ではなく、文節依存木である。これは、拡張するもととなった解析システム南瓜が文節依存木を出力するからであり、括弧を付与する時も文節依存木の意味で正しい部分構造を指定しなければならない。例えば「運ぶ人を」に対しては、「(((運ぶ) 人) を)」ではなく「((運ぶ) (人 を))」とすべきである。このように、何を “構成素” とするかは文法体系によって微妙に異なるので、注意が必要である[7]。

また、本論文でとりあげた三つの解析はそれぞれ前の結果を全て使うという依存関係があるために同時並行的に処理を進めることは難しいが、各解析器が追加する括弧に付与された確率を使って、係り受け解析まで終了した段階で確率の低い括弧を取り除いてもう一度形態素解析からやり直す、ということを繰り返せば統語的な情報を形態素解析にフィードバックすることも可能である。

4 関連研究

膨大な曖昧性をできるだけ効率よく扱うために、従来から主に構文解析において構造共有による曖昧性のパックに関して研究されてきた [1, 10]。HMM を言語モデルとする場合は束構造を使うことでこのような曖昧性を効率よく扱うことができ、本論文ではこれに括弧制約を導入しても効率が損なわれないことを示したが、一般のパック手法に関して同じように効率を損なわないようなアルゴリズムの拡張方法があるかどうかは、興味深い問題である。とくに統語解析でよく使われる言語モデルである語彙化された確率的文脈自由文法 [3, 6] に対しても効率のよい拡張方法が求められれば、これらは文節依存木ではなく構文木を出力するので、特別な修正なしに構成素単位で括弧を付与できるようになる。

謝辞 茶筌・YamCha・南瓜を公開している奈良先端科学技術大学院大学情報科学研究科松本研究室に感謝いたします。

参考文献

[1] R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. *Machine Intelligence*, Vol. 7, pp. 101–116, 1972.

[2] Thorsten Brants. Cascaded Markov models. In *Proc. of 6th EACL*, Bergen, Norway, 1999.

[3] Eugene Charniak. A maximum-entropy-inspired parser. In *Proc. of 6th ANLP and 1st NAACL*, pp. NAACL 132–139, USA, April–May 2000. (`http://www.cs.brown.edu/people/ec/`).

[4] Eugene Charniak and Solomon E. Shimony. Probabilistic semantics for cost based abduction. In *Proc. of 8th AAAI*, pp. 106–111, 1990.

[5] Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, Vol. 46, pp. 311–350, 1990.

[6] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proc. of 41st ACL*, pp. 423–438, Sapporo, July 2003.

[7] Taku Kudo and Yuji Matsumoto. Chunking with support vector machines. In *Proc. of 2nd NAACL*, Pittsburgh, PA, USA, 2001. (`http://cl.aist-nara.ac.jp/~taku-ku/software/yamcha/`).

[8] Taku Kudo and Yuji Matsumoto. Japanese dependency analysis using cascaded chunking. In *CoNLL 2002: Proc. of 6th COLING Post-Conference Workshops*, pp. 63–69, 2002. (`http://cl.aist-nara.ac.jp/~taku-ku/software/cabocha/`).

[9] Yuji Matsumoto, Akira Kitauchi, Tatsuo Yamashita, Yoshitaka Hirano, Hiroshi Matsuda, Kazuma Takaoka, and Masayuki Asahara. *Japanese Morphological Analysis System ChaSen version 2.2.1*. Computational Linguistics Laboratory, Graduate School of Information Science, Nara Institute of Science and Technology, December 2000. (`http://chasen.aist-nara.ac.jp/chasen/doc/chasen-2.2.1.pdf`).

[10] Yusuke Miyao. Packing of feature structures for efficient unification of disjunctive feature structure. In *Proc. of 37th ACL*, pp. 579–584, Maryland, USA, June 1999.

[11] Stefan Riezler. Quantitative constraint logic programming for weighted grammar applications. `http://xxx.lanl.gov/cmp-lg/9705006`, May 1997.

[12] Erik F. Tjong Kim Sang and Jorn Veenstra. Representing text chunks. In *Proc. of 6th EACL*, pp. 173–179, Bergen, Norway, June 1999.

[6] 下記 URL において、解析器の CGI および茶筌・YamCha・南瓜に対するパッチを公開している:
`http://www.carc.aist.go.jp/stella/bktpsr/demo.html`

[7] 文節は構成素 (の右端の部分) の後ろに付属語が付加されたものであるので、この場合に限れば、「閉じ括弧だけが文節に含まれる」という状況を許容することで、「(((運ぶ) 人) を)」に対して「((運ぶ) (人 を))」という解析を行なうようにアルゴリズムを修正することは可能である。

入力 長さ n の文字列 $\vec{S} = (c_1, c_2, \ldots, c_n)$,
形態素辞書 $D = \{(m, p) \mid m$ は形態素, p はその品詞 $\}$
品詞別形態素出現確率 $\Pr(m \mid p)$ および品詞の遷移確率 $\Pr(p' \mid p)$ のテーブル
出力 品詞付き形態素列 $\vec{T} = ((m_1, p_1), (m_2, p_2), \ldots, (m_{n'}, p_{n'}))$
手順
```
table[0,n]: 位置 i で終わる部分解析に対する最大確率を格納する、長さ n+1 の配列。
            各要素は品詞付き形態素 (m,p) をキーとする連想配列
prev[0,n]: 一つ前の品詞付き形態素を格納する、長さ n+1 の配列。
           各要素は品詞付き形態素 (m,p) をキーとする連想配列
table[0][(m0,p0)] = 1; prev[0][(m0,p0)] = nil;
for i = 0 to n-1
  foreach (m,p) ∈ keys_of(table[i]) {
    D' = {(m',p') | m' = (c_i,...,c_j), (m',p') ∈ D, i < j ≤ n};     ···(*)
    foreach (m',p') ∈ D' {
      j = i + length(m');
      if ((m',p') ∉ keys_of(table[j])) {
        table[j][(m',p')] = 0; prev[j][(m',p')] = nil; }
      newprob = Pr(m' | p') × Pr(p' | p) × table[i][(m,p)];
      if (table[j][(m',p')] < newprob) {
        table[j][(m',p')] = newprob; prev[j][(m',p')] = (m,p); }}
```

図 1: 動的計画法を用いた形態素解析アルゴリズム

入力 長さ n の品詞付き形態素列 $\vec{S} = ((m_1, p_1), (m_2, p_2), \ldots, (m_n, p_n))$,
品詞付き形態素列 $\vec{c}$ およびラベルの履歴 $\vec{h}$ の下での、ラベル L の出現確率 $\Pr(L \mid \vec{c}, \vec{h})$ のテーブル
出力 ラベル列 $\vec{L} = (L_1, L_2, \ldots, L_n)$
手順
```
table[0,n]: 位置 i までの部分解析に対する最大確率を保持する、長さ n+1 の配列。
            各要素はラベルの履歴 h⃗ をキーとする連想配列。
prev[0,n]: 一つ前の品詞付き形態素を格納する、長さ n+1 の配列。
           各要素は品詞付き形態素 (m,p) をキーとする連想配列。
skip[0,n]: 開き括弧の直後にラベル I を付与した時に対応する閉じ括弧まで I を付与したことを
           記録するための、長さ n+1 の配列。各要素は履歴 h⃗ をキーとする連想配列。
table[0][h⃗0] = 1; prev[0][h⃗0] = nil; depth = 0; start = false;
for i = 0 to n-1
  switch (S[i]) {
  case '(':  depth++; break;
  case ')':  start = true; break;
  default:
    foreach h⃗_i ∈ keys_of(table[i]) {
      if (h⃗_i ∈ keys_of(skip[i])) continue;
      assign(table,prev,i,h⃗_i,B);
      if (start && prev をたどって得られる最大確率を持つ部分解において、
                   depth==0 に対応する最初の開き括弧の二つ前までにラベル B がある)
        continue; // 制約 (1)
      else if (assign(table,prev,i,h⃗_i,I) == true && depth > 0) { // 制約 (2)
        k = (depth==0 に対応する最初の閉じ括弧の位置);
        prob = table[i][h⃗_i]; h⃗ = h⃗_i;
        for j = i to k { prob = Pr(I | c⃗_j, h⃗) × prob; h⃗ = h⃗ ⊕ I; }
        table[k][h⃗] = prob; prev[k][h⃗] = (i,h⃗_i);
        skip[i+1][h⃗_i⊕I] = k; }}
    depth = 0; start = false; break; }

function assign(tbl,prv,i,h⃗,L) {
  h⃗' = h⃗ ⊕ L;
  if (h⃗' ∉ keys_of(tbl[i+1])) {
    tbl[i+1][h⃗'] = 0; prv[i+1][h⃗'] = nil; }
  newprob = Pr(L | c⃗_{i+1}, h⃗) × tbl[i][h⃗]
  if (tbl[i+1][h⃗'] < newprob) {
    tbl[i+1][h⃗'] = newprob; prv[i+1][h⃗'] = (i,h⃗); return true; }
  return false; }
```

図 2: 括弧制約を考慮した文節まとめあげ解析アルゴリズム

入力 長さ n の文節列 $\vec{S} = (s_1, s_2, \ldots, s_n)$,
文節列 $\vec{c}$ およびラベルの履歴 $\vec{h}$ の下での、ラベル L の出現確率 $\Pr(L \mid \vec{c}, \vec{h})$ のテーブル
出力 ラベル列の列 $\mathbb{L} = (\vec{L}_1, \vec{L}_2, \ldots, \vec{L}_m)$。ただし、$\vec{L}_1$ の長さは n で、$\vec{L}_m$ は先頭が B で残りは I。
手順
```
table[1,n][0,n]: k 回目のまとめあげ解析における、位置 i までの部分解析に対する最大確率を
                 格納する、n × (n+1) の配列。
prev[1,n][0,n]: k 回目のまとめあげ解析における、一つ前のチャンク又は品詞付き形態素を格納する、
                n × (n+1) の配列。各要素は履歴 h⃗ をキーとする連想配列
k = 1; nchunks = n;
while (nchunks > 1) {
  table[k][0][h⃗0] = 1; prev[k][0][h⃗0] = nil;
  for i = 0 to n-1
    foreach h⃗_i ∈ keys_of(table[k][i])
      foreach L ∈ {B,I}    ···(*)
        assign(table[k],prev[k],i,h⃗_i,L);
  path = (prev[k] を辿って確率最大の解を構成する);
  if (path に含まれるチャンクの数 == nchunks)
    括弧に矛盾しない最も右の文節 path[?] のラベルを I に変更する。
  nchunks = (path に含まれる B ラベルの数); k++; }
```

図 3: 括弧制約を考慮した多段階まとめあげ解析アルゴリズム