自然言語処理 121－7
（1997. 9. 11）

ＧＬＲパーザの効率化に関する研究

植木正裕, 徳永健伸, 田中穂積
東京工業大学 大学院情報理工学研究科
{ueki,take,tanaka}@cs.titech.ac.jp

本論文は、ＧＬＲ法における圧縮共有の効率を改善する手法を提案する。ＧＬＲ法では、圧縮共有統語森により解析の高速化をはかっているが、冨田によるＧＬＲ法の実装では、アクションのコンフリクトによって枝分かれした複数のスタックの間で解析のタイミングがずれるために、共有できるシンボルの生成のタイミングもずれることがある。このため、スタックの完全な圧縮共有ができているとは限らず、同じ解析動作が重複して行なわれたり、完全な圧縮共有統語森が生成できないこともある。本論文で提案するＧＬＲパーズの手法では、枝分かれしたスタックの間での解析のタイミングを制御し、スタックの共有化のタイミングを早め、完全な圧縮共有統語森を作成することができる。これにより、解析の高速化と使用メモリ空間の削減を同時に実現できる。

Improvement of Efficiency of GLR Parsing

UEKI Masahiro, TOKUNAGA Takenobu, TANAKA Hozumi
Department of Computer Science
Tokyo Institute of Technology
{ueki,take,tanaka}@cs.titech.ac.jp

In this paper we propose a method to improve efficiency of GLR parsing. Tomita's GLR implementation uses two data structures, graph-structured stack and packed-shared parse forest. Both structures help us avoid applying same parsing action repeatedly to save parsing time and memory space. However, his implementation misses a chance to share data structures. To the contrary, our method not only keeps the advantage of Tomita's implementation but also allows data structures to be completely shared. Experiments show our method exceeds in both time and space efficiency compared to Tomita's method.

1 はじめに

自然言語処理では、形態素解析・構文解析・意味解析が必要になる。近年、構文解析と意味解析を統合する研究が行なわれているが、形態素解析と構文解析の統合に関しては研究が十分とはいえない。本研究で用いるＭＳＬＲシステムは、形態素解析と構文解析を完全に統合したシステムである [3, 4]。複数の解析を統合することの利点は、それぞれのレベルでの制約を同時に利用できることである。形態素解析と構文解析を統合すれば、形態素候補の選択の際に構文的な制約を利用できる。

ＭＳＬＲシステムでは、ＥＤＲ日本語単語辞書 [5] を用いて入力文中の単語の候補を抽出し、冨田によるＧＬＲ法 [2] を用いて形態素解析と構文解析を行なう。ＧＬＲ法では、文法からあらかじめＬＲ表を作成しておき、解析時にはＬＲ表を参照することで解析動作を決定する。ＭＳＬＲシステムでは、ＬＲ表を作成する際に、文法中の終端記号間の接続可能性に関する制約を用いて、ＬＲ表中の不必要な解析動作を削除しておく。これにより、構文解析を行なう過程で正しい形態素を選択することができる。

冨田によるＧＬＲ法では、グラフ構造化スタックと圧縮共有統語森という２つのデータ構造を用いることで、解析の効率化を図っている。ところが、冨田のアルゴリズム [2] では効率化が最大限には行なわれないことがある。本研究では、その問題点の指摘と解決法を述べる。

2 冨田によるＧＬＲ法

ＧＬＲ法では、解析動作に曖昧性がある場合、解析スタックを分岐させることにより、すべての解析動作を並行して行なう。解析が進むにつれ、分岐の数は組合せ的に増大するため、曖昧性の多い文の解析は難しい。冨田法では、解析の効率を上げるために、グラフ構造化スタックと圧縮共有統語森 (packed shared forest) という２つのデータ構造を用いる [2]。

グラフ構造化スタックは、複数のスタックトップに同じ記号が現れる場合に、スタックをマージすることで、スタックを木構造ではなくＤＡＧ構造にする。スタックをマージすることで、それ以降の重複した解析を回避できる。圧縮共有統語森は、２つの部分構文木が同じ非終端記号を親ノードに持ち、その部分木がカバーする終端記号列が同じ場合に、親ノードを共有する。

冨田法では、この２つのデータ構造を利用するために、merge, pack, share という３つの操作を用いている。ここでは、本稿の対象とするpack についてのみ簡単に説明する。

ＧＬＲ法では、解析が進む過程で曖昧性によりスタックを分岐するが、reduce によって複数のスタックに同じ非終端記号が現れることがある。

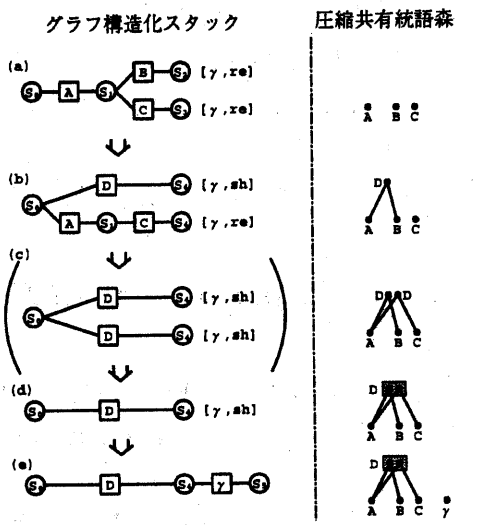


Fig. 1 スタックの pack

Fig. 1 の例ではスタックが２つに分岐している (a) が、図の上側の分岐では D → A, B (b)、下側の分岐では D → A, C によりともに D に reduce されている (c)。そこで、スタック上には S_0 — D — S_4 を１本だけ生成して残し、２つのスタックをマージする (d)。このような操作を pack とよぶ。スタックは解析動作

を制御するための中間状態を保持するだけなので、このようにスタックをマージすることで、以降の同じ解析動作を回避できるというメリットがある。

しかし、解析結果としては、D → A, B と D → A, C の2つの reduce によりできた2つの異なる構文木がいずれも必要である。そこで、圧縮共有統語森上には両方の構文木を生成しておく。ただし、この部分の違いだけで構文木全体を2つ別個に生成するのは空間的効率が悪い。そこで、2つの部分構文木で親ノード D を共有することにより、残りの構文木全体を共有する ((d) の右図参照)。これによって、以後の処理ではあたかも1つの木であるごとく解析を進めることができる。

3 冨田法の問題点

GLR法では、解析動作にコンフリクトがある場合に、スタックを分岐させてすべての解析動作を並行して行なうことはすでに述べた。入力文を読み進んだ段階ですでにスタックに分岐がある場合には、すべての分岐について解析を行なうことになる。冨田のアルゴリズムでは、まず各分岐ごとに深さ優先で reduce を行なう。すべての分岐に対する reduce が終わったところで、先読みの shift を行なう。ところが、この方法では、pack のタイミングが遅れたり、pack に失敗する場合があることが指摘されている [1]。

例として Fig. 2 の場合を考えてみたい。

この例では、最初スタックが2つに分岐している。まず上側の分岐について解析を行なう。F → B, D と G → A, F の2つのルールによる reduce が行なわれて S_0 — G — S_7 が生成される。上側の分岐での reduce はこれですべて終わったので、次に下側の分岐の解析を行なう。F → C, E と G → A, F による reduce が行なわれたところで、下の分岐にも S_0 — G — S_6 が生成されるが、上の分岐と同一のスタックになるのでこの時点で pack を行なう。

ところが、この例では F → B, D と F → C, E によって S_1 — F — S_6 という枝が2回生成され、そのため、G → A, F による reduce も2回行なわれてしまうことに注意したい。つまり、本当に曖昧なのは F を根とする構文木であるのにも関わらず、下側の分岐で reduce により F が得られたときには、上の分岐では G → A, F の reduce により F がすでに pop されているため、F での pack のタイミングを失する。

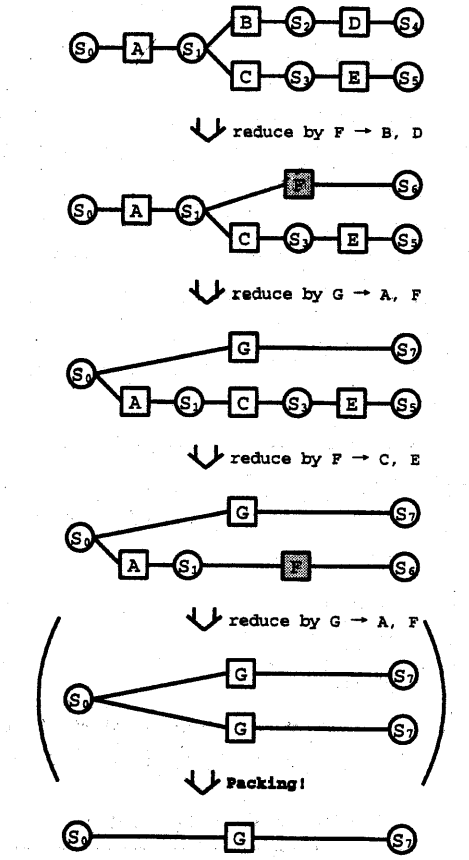


Fig. 2 タイミングの遅れによる pack の失敗

このように、冨田のアルゴリズムでは、分岐ごとの解析のタイミングにずれがあるために、pack のタイミングが遅れたり、文法によっては pack が起こらないことがあり得る。

ここで、pack がうまく行なわれない条件について、もう少し詳しく考察してみたい。そこで、次のような2つの文法を考えてみる。

S → A	S → A
A → A, A1	A → A1, A
A → A, A2	A → A2, A
A → A1	A → A1
A → A2	A → A2
A1 → a	A1 → a
A2 → a	A2 → a
文法 (a)	文法 (b)

2つの文法はどちらも a a a ... という文が解析可能で、文法の能力としては等価である。異なる点は再帰ルールが (a) 左再帰になっているか (b) 右再帰になっているかという点だけである。ところが、この2つの文法で解析を行なうと、後述する例から明らかなように、(b) のルールの場合には pack に失敗するので、生成される深さ1の部分構文木の数が極端に異なる (Fig. 3)。

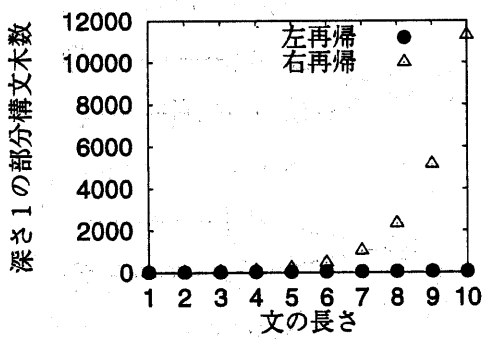
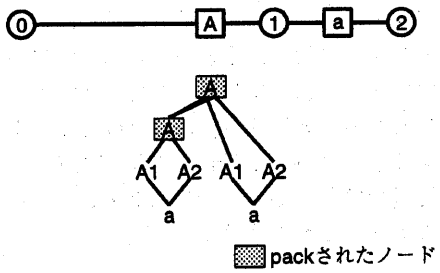


Fig. 3 再帰ルールによる差

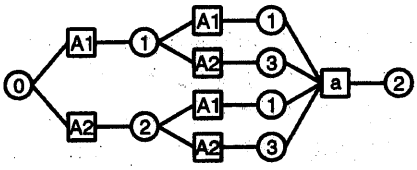
左再帰の場合には、入力を読み進むごとに、スタックトップでの reduce が行なわれ A が生成されるため、すぐに pack が行なわれる。

それに対して、右再帰の場合には、A への reduce はすぐには行なわれず、A1 または A2 という曖昧性を残したまま解析が進む。そのため、スタックは組合せ的に枝分かれを生じることになる。さらに入力を読み進み文末まで来ると、すべての分岐に対して深さ優先で再帰ルールによる reduce が行なわれる。reduce は S が生成されるまで一気に行なわれてしまうため、途中で生成される A での pack が冨田法では不可能である。これを具体例で示そう。

入力 "a a a" を文末まで読み進んだ直後のスタックを Fig. 3 に示す。



(a) 左再帰による解析



(b) 右再帰による解析

Fig. 4 解析スタックの違い

左再帰による解析では、組合せ的な曖昧性の情報は統語森にだけ残されていて、スタックは枝分かれしない。右再帰による解析では、曖昧性がスタック上にそのまま残されている。先読みの shift によりスタックトップだけは merge されているが、reduce が始まればすぐにまた分かれてしまう。

上記した例は再帰ルールを用いた極端な例だが、一般のルールでも Fig. 3 に示したように pack が行なわれない可能性がある。一般に冨田法では、右辺長が平均して短い場合には、pack の効果は小さくなる。

冨田法では、pack は空間的・時間的効率の両方に影響するので、pack を最大限行なうことが望ましい。そこで、本稿ではこの問題を解決するために次節に示すアルゴリズムを提案する。

4 アルゴリズムの改良

3 で指摘した問題点を解決するために本稿で提案するアルゴリズムの基本となる考え方を以下に示す。

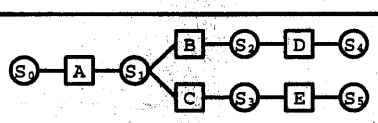
1. **スタックの分岐間での解析のタイミングを揃える**

   冨田のアルゴリズムの問題は、分岐によって解析の進み具合にずれができることであった。提案するアルゴリズムでは、各分岐ごとに reduce を深さ優先で行なうのではなく、reduce の範囲を制限することで、分岐間での解析のタイミングを制御する。

2. **圧縮共有統語森を過去の reduce 動作の履歴として参照する**

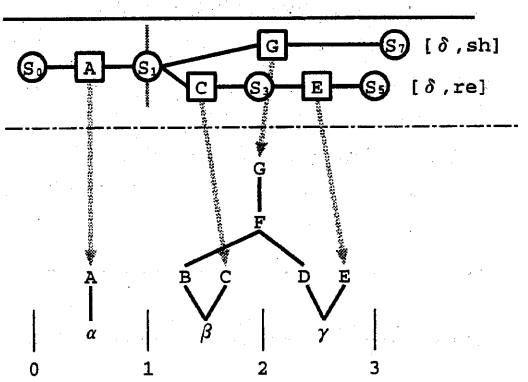
   後述するが、1 により Fig. 2 に示した pack のタイミングのずれを避けることができるが、文法中にユニットルールが含まれる場合、似たような失敗が生じることがある。これは、スタック上には過去の解析の履歴が残らないことに起因する。一方、圧縮共有統語森は過去の reduce 動作の履歴そのものであるので、これを参照することで、すでに reduce により pop され、スタック上には存在しない非終端記号の中に、pack 可能な非終端記号があるかどうかを調べる。

以下では、改良したアルゴリズムを用いた解析の様子を具体的に説明する。



解析過程 1

解析過程 1 のスタックの上側の分岐を用いた解析で F → B, D と G → F による reduce が行なわれるとの状態 2 になる。

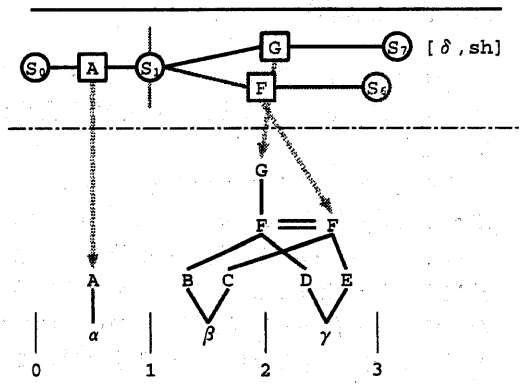


解析過程 2

上図の上半分がグラフ構造化スタック、下半分が圧縮共有統語森を表している。矢印は、スタックと統語森上の非終端記号の対応関係を表している。α, β, γ はそれぞれ 1 記号列であるとする。

このとき、reduce 範囲に制限がなければ、上の分岐での解析をそのまま続けることになるが、例では S_1 を越える reduce は禁止しているため、上の分岐の解析をここで一時中断する。

次に下側の分岐を用いた解析に移ると、F → D, E による reduce が行なわれる。

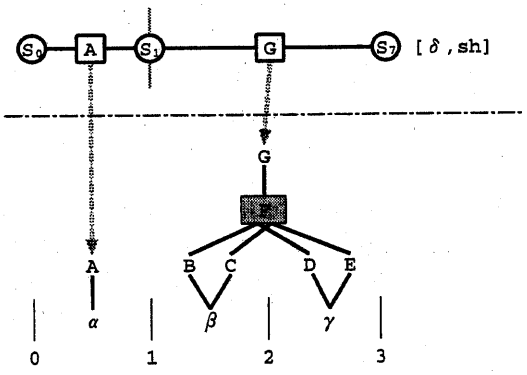


解析過程 3

このとき、冨田のアルゴリズムでは、スタック上にほかの F がないため、ここで新たに生成した F をスタック上に push するが、本ア

ルゴリズムでは圧縮共有統語森を参照して、カバーする終端記号の範囲が同じ、F という非終端記号が過去に存在したかどうかを調べる。

この例では、上側の分岐で F → B, D によってできた部分構文木が見つかるので、2 つの部分木で親ノード F を共有させ、スタック上には F は push しない。



解析過程 4

その結果、上図に示すスタックと統語森ができあがる。pack による変更は、F 以下の部分構文木の追加のみで、統語森のほかの部分やスタックには変更は必要ない。

5 評価実験

4 節で提案したアルゴリズムをＭＳＬＲシステム上に実装し、冨田法との比較実験を行なった。

まず、3 で例に挙げた再帰ルールについて、改善の効果があがっているかどうかを検証する。

冨田法では、右再帰の場合に部分構文木数が指数オーダーで増加していたが、本稿で提案した手法では右再帰・左再帰どちらの場合にも最大限に pack が行なわれ、部分構文木数はまったく同じになる。

次に、ルール数約 900 の日本語文法による解析実験を行なった。

評価用の文としては、ＥＤＲコーパスより 5,000 文をランダムに抽出した。そのうち、固有名詞や異表記などにより解析に失敗するもの、冨田法ではメモリがたりないなどの理由で解析できないものを除いた 1,226 文を用いて比較を行なった。

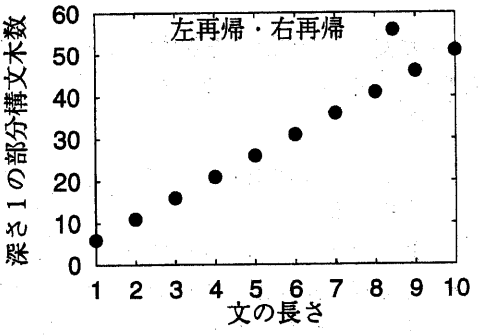


Fig. 5 再帰ルールでの解析結果

比較するパラメタとしては、グラフ構造化スタックのノード数、圧縮共有統語森の深さ 1 の部分構文木数を用いた。pack が行なわれることにより、スタックのマージが行なわれるため、pack の回数が多く、タイミングが早いほどスタックのノード数は少なくなる。また、pack により重複した解析動作を回避でき、無駄な部分構文木の生成も抑制される。

Fig. 3 と Fig. 4 は、冨田法でのノード数・構文木数を 1 としたときの本手法での結果を各文ごとにプロットしたものである。文が長くなるほど改良の効果が高いことがわかる。入力全体の平均では、スタックのノード数で 0.23、深さ 1 の部分構文木数で 0.46 まで削減できた。

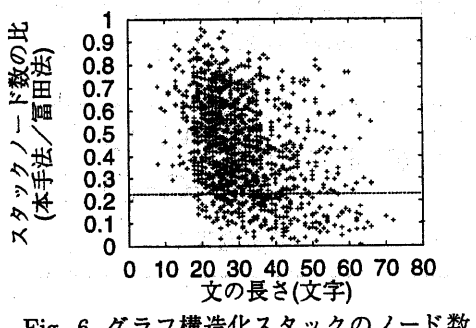


Fig. 6 グラフ構造化スタックのノード数

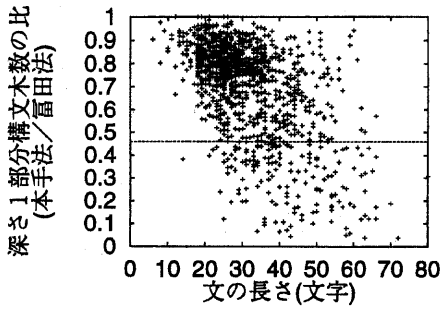


Fig. 7 深さ1の部分構文木数

6 結論

本稿では、冨田によるＧＬＲ法の問題を解消し、解析をより効率的に行なうための pack に関する新しい手法を提案した。実験により、本手法は空間的・時間的効率において、冨田の方法よりも大幅に改善されることが確認できた。また、冨田の方法ではメモリ不足により解析できなかった長文も解析が可能になった。

今回の実験では筆者らが開発した日本語文法を用いた。この文法は特に本手法の特性を考慮することなく作成されている。しかし、pack の効果は文法に依存する部分があるので、今後他の文法を用いて実験を行なう予定である。

References

[1] 伊東秀夫. LR 表を用いたチャートパージングアルゴリズム. 情報処理学会 自然言語処理研究会, Vol. 99, No. 7, pp. 49–56, 1 1994.

[2] Masaru Tomita and See-Kiong Ng. The generalized LR parsing algorithm. In *Generalized LR Parsing*, pp. 1–16. Kluwer Academic Publishers, 1991.

[3] 伴光昇, 福田譲, 白井清昭, 田中穂積. 圧縮統語森上での形態素解析候補の絞り込み - 品詞列統計情報の利用-. 1994 年度 人工知能学会全国大会 (第 8 回) 論文集, pp. 527–530, 6 1994.

[4] 植木正裕, 徳永健伸, 田中穂積. EDR 辞書を用いて日本語文の形態素解析と統語解析を行なうシステム. EDR 電子化辞書利用シンポジウム論文集, pp. 33–39, 7 1995.

[5] 日本電子化辞書研究所. EDR 電子化辞書利用マニュアル, 第 2.1 版, 1994.