E6-1

# 大域的グラフノード共有による単一化解析手法

Unification based parser using graph node global sharing

新納 浩幸
shinnou@trl.mei.co.jp
松下電器産業株式会社 情報通信東京研究所

素性構造の単一化処理の効率は、引数に当たるグラフのノードのコピー回数に大きく影響される。このため、従来、コピー回数を減少させる手法が考案されてきた。ここでは、Wroblewski のアルゴリズムにグラフのノードを共有させる手法を取り入れる。また、この際、各ノードに共有情報を加え、解析全体から見たノードの大域的な破壊可能性を検査することで、ノードのコピー回数を減少させることができることを示す。

## 1 はじめに

素性構造の単一化に基づいた文法を用いて自然言語処理システムを実現する場合、全体の処理の中で素性構造の単一化処理の占める割合が大きいため、単一化アルゴリズムの効率が全体の効率に大きな影響を与える [6]。

単一化アルゴリズムの効率化のボトルネックは素性構造を表すグラフのノードのコピーである。このために、コピーの回数を減らす手法が考案されている [2][5][3]。

この論文では、まず、コピーの回数を減らす以下の２つの方法を考案する。

- グラフノード共有による Wroblewski のアルゴリズム [5] の改良。
- 破壊可能なグラフの検知。

次に、これらを組み合わせた大域的グラフノード共有による単一化解析手法を提案する。

## 2 グラフノード共有による単一化

Wroblewski のアルゴリズムは、コピーが必要になった段階ではじめてコピーを作るアルゴリズムである（図１参照）。このアルゴリズムの単一化操作 unify2 は、単一化するノードがコピーによって作られたもの（以下コピーノードと呼ぶ）であれば破壊的な単一化操作である unify1 により単一化を行ない、そうでない場合、コピーノードを作り、その娘ノードに対して unify2 を再帰的に利用することで単一化を行ない、その結果を先のコピーノードに加えてゆく。

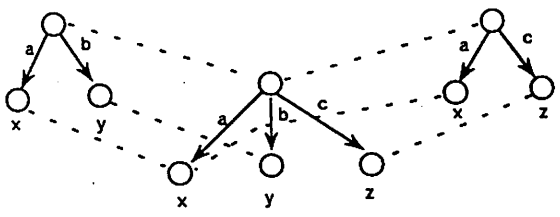


図１: Wroblewski の単一化

ここで提案する単一化操作 unify3 は、引数の２つのノードの単一化結果が、第１引数と同じである場合[1]、その第１引数を返し、異なる場合、第１引数がコピーノードであれば、第１引数の実体を破壊的に修正することで第１引数（ノードの *id*、ポインタ）を返し、コピーノードでない場合に、そこで新しいノードを作る手法である。

これは、図２のケースのように、ノード３以下のサブグラフをそのまま、あるノード（この場合、ノード７）の下にアペンドする場合、Wroblewski の単一化アルゴリズムではそのサブグラフごとコピーする必要があるが、ここで提案する手法では、図３のように、ノード３を共有した段階で、ノード３以下のサブグラフをそのまま利用するので、コピーする必要がなく、効率が良い。

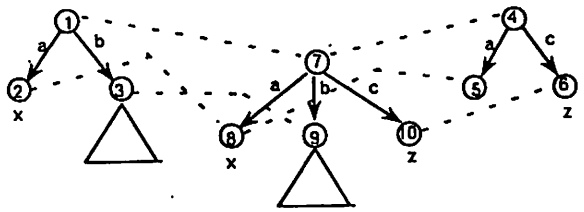


図２: サブグラフのコピーが必要な単一化

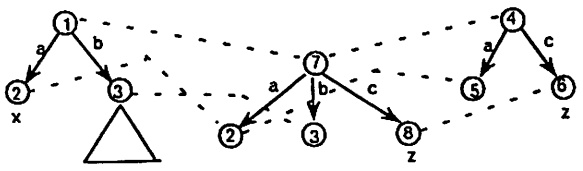


図３: サブグラフのコピーが不必要な単一化

ただし、あるサブグラフが共有された後で、パス表現により共有されているノードを書き変えなくてはいけない場合、その結果を親のノードに伝搬させてゆく必要がある。例えば、図４において、ノード３が、ノード４に書き変わる場合、ノード３の親

[1] どちらかの引数と同じであれば良いが、実際は一方のみに限定しておく方がよい。

ノードに当たるノード2も書き変えてゆく必要がある。

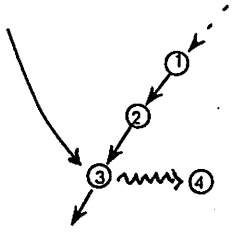


図4：親ノードの変更が必要なケース

Wroblewski のアルゴリズムでは、1度単一化されてできたノードは必ずコピーであるため、例えば、図4ではノード3を破壊的に作り直せば良く、上記の問題は発生しない。

ここでは、各グラフに対して、そのグラフのアサイクリックな地点になっているノード（複数の親ノードから参照されている娘ノード）の集合を持たせておくことにする。単一化処理終了後に保持してあるアサイクリックな地点になっているノードのforward情報を参照し、そのノードに書き変えが起こっている場合、単一化結果のグラフ中から書き変え前のノードを探し、必要に応じてグラフの作り変えを行なう。

## 3 グラフの破壊可能性

単一化処理で引数のグラフを破壊できないのは、そのグラフに対して複数回の単一化を行なう必要があるからである。2つの引数のうち一方は、文法であるために、常に破壊不可能であるが、もう一方のグラフは、句を表す構文意味構造であり、複数回のうちの最後の単一化ではグラフを破壊できる。

ここでは、冨田法 [4] に基づくパーサで、グラフ構造化スタック（ＧＳＳ）のスタックノードのリンク情報を参照することで、破壊可能なグラフを検知する。

具体的には、 reduce 操作時に pop するスタックノードが、ＧＳＳから解放できるとき、そのスッタクノードの保持するグラフ（素性構造）は、破壊可能になる。

スタックノードがＧＳＳから解放できるのは、以下の条件をすべて満たす時である。

条件0 全ての pop されるスタックノードは、最初に pop されるスタックノードが、他に読み込むべき形態素あるいは、他の操作がないときに解放可能。

例えば、スタックノードから複数の形態素を読み込める場合、最後の形態素を読み込んだ場合に関してだけ解放が可能になる。

また、ある形態素を読み込んだ時や、 reduce 操作によって、次の操作にコンフリクトが生じた場合、その最後の操作に限って解放が可能になる。

条件1 最初に pop されるスタックノードは、 pop される組が1組のとき解放可能。複数組の場合は、最後の操作時に解放可能。

条件2 最初に pop されるスタックノードならば、娘スタックノードを持たないときに解放可能。最初に pop されるスタックノード以外ならば、1つ前に pop されたスタックノードの娘スタックノードが1つだけである場合に解放可能。

条件3 最後に pop されるスタックノード以外ならば、複数の親スタックノードを持たない。

図5に例を示す。条件0は成立しているとする。ノード $c$ から形態素 $x$ を読み込む時、文法 $n$ により、ノード $d$、$c$、$b$ とノード $d$、$c$、$a$ が pop される場合、最初の $d$、$c$、$b$ によるリダクションでは、$d$ は、条件1により解放不可、$c$ は条件3より解放不可、$b$ は、条件1、2、3を満たすので、解放可能となる。このリダクションにより、図6のようなＧＳＳができた時、次の $d$、$c$、$a$ によるリダクションでは、$d$、$c$、$a$ すべて解放可能となる。

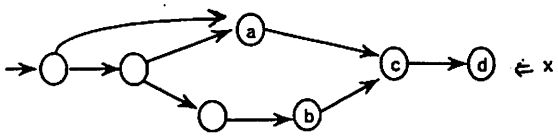


図5：リダクション前のＧＳＳ

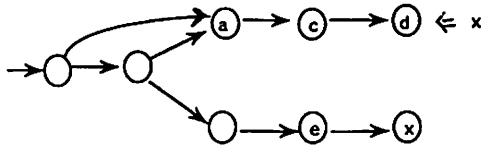


図6：リダクション後のＧＳＳ

## 4 大域的グラフノードの共有

2節の方法を利用する場合、あるノードは、そのグラフ内だけで参照されているノードではなく、大域的に他のグラフからも参照されている可能性がある。このため、2節、3節の方法を組み合わせて利用する場合、3節で示した単純なグラフ破壊を行なうことができない。

そこで、以下の設定を行なうことにより、上記の2つの手法を組み合わせて利用する方法を示す。

1. グラフの各ノードに新しい素性 *nodelink* を設け、その素性値にそのノードの大域的なすべての親ノードの個数を持たせる。

   これは解放可能なスタックノードの持つグラフでも、そのグラフのサブグラフが別のスタックノードに保持されているグラフのサブグラフになっている場合に、破壊するのを避けるために設けている。例で示すと図7のサブグラフＡを破壊しないために設けている。

2. 大域変数 *on_gss* を用意し、この変数には、ＧＳＳの各スタックノードが持つグラフのルートノードの *id* とその個数を保持させる。

   これは解放可能なスタックノードの持つグラフでも、そのグラフのサブグラフが別のスタック

ノードに保持されている場合に、破壊するのを避けるために設けている。例で示すと図8のサブグラフAを破壊しないために設けている。

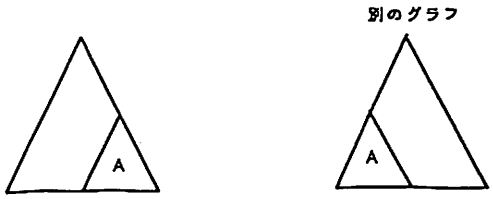


図7: *nodelink* からの破壊不可なサブグラフ

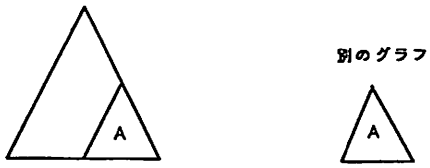


図8: *on_gss* からの破壊不可なサブグラフ

3. 単一化の関数に3番目の引数を設定する。

   単一化の関数は、第1、第2引数が単一化対象のグラフに対応するものであるが、3番目の引数として、1番目の引数のグラフが破壊可能かどうかを示すフラグを与えることにする。

ここで新たに示す単一化操作 unify4 は、2節で述べた unify3 の拡張である。unify3 では、引数のノードが破壊可能であれば、そのノードを破壊することで、単一化する。その際、破壊可能性のチェックは、unify2 と同様、その単一化の過程でコピーによって作られたものであるかどうかによる。unify4 では、さらに以下の条件を満たしていれば、そのノードが引数のグラフの一部であっても破壊可能とする。

条件A　unify4 の第3引数が *t*。

条件B　そのノードの *nodelink* が1以下。

条件C　そのノードの *id* が *on_gss* に登録されていない。

図9に unify4 のアルゴリズムを示す。

アルゴリズム中の arclist とは、ノードから出ているアークの名前とそのアーク先のノード *id* をペアにしたペアリストである。また、アルゴリズム中で使用されるプロシージャ intersectarcs は、2つの引数の arclist から共通のアークの集合を取り出し、プロシージャ complementarcs は、第2引数の arclist にあるアークで第1引数の arclist にないアークの集合を取り出す。また、プロシージャ destrablep は、引数で示されるノードが破壊可能かどうかを調べ、プロシージャ copy_subgraph は、引数で示されるノード以下のグラフをコピーする。

unify4 は、ノードの *nodelink* を破壊的に変更してゆくために、unify4 が *FAIL* した場合には、変更した *nodelink* をもとに戻す必要がある。このために、*nodelink* の増減は履歴を取っておく。

```
PROCEDURE top_unify4(id1,id2,dstr_p)
 id = unify4(id1,id2,dstr_p)
 IF id != FAIL  THEN
   親ノードの変更
   ELSE { 作成したグラフノードの解放 and
          nodelink の recover }
 ENDIF
ENDPROCEDURE

PROCEDURE unify4(id1,id2,dstr_p)
 必要なら id1,id2 を forward 先に取り替える。
 IF id1 isn't copy and id2 is copy THEN
   id1 <--> id2
 ENDIF
 al1 is arclist of id1 node
 al2 is arclist of id2 node
 IF al1 or al2 is symbol THEN
   IF al1 == al2 THEN  return id1
   ELSE exit fail
   ENDIF
 ELSE
  shared = intersectarcs(al1,al2)
  FOR each arc in shared DO  {
   c_id1 is value of arc in al1
   c_id2 is value of arc in al2
   c_d_p = destrablep(c_id1,dstr_p)
   id = unify4(c_id1,c_id2,c_d_p)
   ++{id -> nodelink}
   add pair (arc id) to new_arclist }
  new1 = complementarc(al2,al1)
  FOR each arc in new1 DO  {
   id is value of arc in al1
   ++{id -> nodelink}
   add pair (arc id) to new_arclist }
  new2 = complementarc(al1,al2)
  FOR each arc in new2 DO  {
   id is value of arc in al2
   copy_id = copy_subgraph(id)
   add pair (arc id) to new_arclist }
  CASE
   new_arclist == al1 :
     --{nodelink of nodes in al1}
     return id1
   id1 is copy :
     change arclist of id1 to new_arclist
     ++{id1 -> nodelink}
     return id1
   dstr_p is t :
     change arclist of id1 to new_arclist
     {id1 -> copy-flag} = T
     --{nodelink of nodes in al1}
     return id1
   otherwise :
     create new_node
     arclist of new_node is new_arclist
     return id of new_node
 ENDIF
ENDPROCEDURE
```

図9: unify4 アルゴリズム

解析では、reduce 操作が起こった時に、必要なスタックノードを pop する。このとき、3節で示した条件0～3を調べ、解放可能なスタックノードの場合、そのノードが保持するグラフのルートノード *id* に対して *on_gss* が持っている個数を1減らす。

その結果、0になれば、登録を削除し、unify4 の第3引数に $t$ を与える。

pop したすべてのグラフに対して、単一化処理が終了し、reduce 結果のグラフが得られたら、そのルートノード $id$ を $on\_gss$ に登録する。破壊可能なグラフに対して単一化が $FAIL$ した場合、*node-link* を参照しながら、そのグラフのノードを解放してゆく。

また、ＬＲ表から次の操作がない場合にも、スタックノードは解放されてゆくが、その場合も、*node-link* を参照しながら、そのスタックノードの保持するグラフのノードを解放してゆく。

## 5 実験

この論文で示した手法を評価するために、従来の単一化処理（unify1、unify2）と提案した単一化処理（unify3、unify4）に対して、素性構造の破壊可能性の検知を行なうパーサ（提案したパーサ）と、行なわないパーサ（従来のパーサ）を組合わせて、それぞれの処理時間を比較した。

（例文ａ） He saw a girl with his telescope $

| | 従来の単一化 | | 提案した単一化 | |
|---|---|---|---|---|
| | unify1 | unify2 | unify3 | unify4 |
| 従来のパーサ | 367 (372) | 336 (300) | 162 (38) | |
| 提案したパーサ | 331 (295) | 159 (100) | | 210 (21) |

（上段は実行時間 (msec)
下段括弧内はノードのコピー回数）

実験は、Sparc-Station-1 の KCL 上で、例文ａに対して、２０回解析し平均時間を測定した。用いた文法は１０個程度の単純なものである。

上記実験においては、2節、3節の手法により、コピー回数を減らすことができた。処理時間では、従来のパーサに対する unify2 と unify3 の処理時間から、2節の手法の効果が見られる。また、unify1 あるいは unify2 に対する従来のパーサと提案したパーサの処理時間から3節の手法の効果が見られる。

しかし、2節、3節の手法を組み合わせた場合（提案したパーサ×unify4）、*nodelink* の参照処理、増減処理の負担が大きかったために、よい結果が得られなかった。

一般に単一化が $FAIL$ する場合が多いと、unify1 と unify2 の差が大きくなり、単一化が成功する場合が多いと、unify2 と unify3 の差が大きくなる。今回の実験では、単一化が $FAIL$ する場合が少なく、成功する場合が多いために、従来のパーサに対して上記のような結果となった。

## 6 まとめと今後の研究

単一化解析の効率化として、グラフノード共有による Wroblewski のアルゴリズムの改良と、破壊可能なグラフの検知を考案し、それを組み合わせた解析方法を提案した。また、その有効性を実験により評価した。

unify3 は、単一化対象のグラフがツリー構造の場合に、Wroblewski の unify2 よりも早くなる。特に単一化が $FAIL$ しない場合に、その差が顕著になる。しかし、グラフにアサイクリックな構造が含まれると、2節で述べたように、グラフの作り直しの処理が起こる場合があり、完全な改良にはなっていない。ここでは、解析全体の平均的な効率を実現することを目的とした。

unify3 の欠点として、ノードの id とその実体との組をデータベースとして大域的に持っておく必要があり、そのために、ノードの数が増えると、そのデータベースを検索する処理が重くなることがある。ここでは、「いらなくなったノードは積極的に捨てる」、「ハッシュテーブルを用いて検索効率をあげる」という方針を立てている。

グラフの破壊可能性の検出は、曖昧性が増大していかないような解析で有効である。そのような解析では、スタックノードが共有される場合が少ないために、破壊可能な素性構造が多数出現する。

2節と3節の手法を組み合わせた場合、*nodelink* の参照、増減処理の負担が大きい点の他に、ＧＳＳのスタックノードの解放と、素性構造のグラフノードの解放を、適切な時期に逐次的に行なわなければならないという欠点がある。これはメモリ効率上は長所になっているが、5節の実験で示した他の組合せでは、解放自体も行なう必要がないために、メモリを犠牲にすれば、他の手法の方が効率的ではあると思われる。

今後の研究として、ここでの手法を packing 手法へ展開してゆきたい。

paking 手法は、曖昧性として生じる複数の素性構造を選言を用いて１つの素性構造として扱うことで解析効率を上げる手法である [1]。ここでの手法を利用することで、選言を用いて表された素性構造（グラフ）が破壊可能になるケースが増え、素性構造の奥深くに押し込められた選言を unpack する場合にも効率が落ちない可能性があると思われる。

## 参考文献


[1] Eisele,A. and Doerre,J. : Unification of Disjunctive Feture Descriptions, *ACL-88*, 7-10 (1988)

[2] Karttunen,L. and Kay,M. : Structure Sharing with Binary Trees, *ACL-85*,133-136A (1985)

[3] 加藤進，小暮潔 :「素性構造の単一化手法の効率」, 情報処理学会自然言語処理研究会,64-9 (1987)

[4] Tomita,M.: An Efficient Augmented Context-Free Parsing Algorithm, *Computational Linguistics*, 13,1-2,31-46 (1987)

[5] Wroblewski : Nondestructive graph unification, *AAAI-87*,581-587 (1987)

[6] 安川秀樹：ユニフィケーション文法,『自然言語の基礎理論』, 共立出版, p.p.109-143 (1986)