自然语言句法分析的顺序控制算法

宋柔 王鑫

(北京计算机学院, 100044)

THE SEQUENCIAL ALGORITHMS OF A GRAMMAR PARSER OF NATURAL LANGUAGES

Song Rou and Wang Xin

(Beijing Computer Institute, 100044)

ABSTRACT

The run speed of a system of natural language processing is an extremely important bearing on developing the system itself. One of the cores of the system is a grammar parser. This paper gives the general sequencial algorithms of simple backtraking, thorough backtracking and pruning, which are used in a grammar parser, and some thinking about implementing these algorithms. Our experiments and analysis show that these algorithms are very efficient.

摘 要

自然语言处理系统的运行速度对于系统本身的开发是至关重要的. 这类系统的核心之一是句法分析子系统. 本文讨论句法分析的一般性的顺序控制算法, 包括简单回溯算法、彻底回溯及其剪裁的算法, 以及算法的实现方法. 实验和分析表明, 这些算法的效率是相当令人满意的.

§1. 引 言

对于自然语言处理系统, 尤其是机器翻译系统来说, 处理速度是至关重要的. 这不仅是用户使用的需要, 也是开发系统本身的需要. 由于对象的复杂性和需求的长期性, 这类系统的开发几乎是无止境的, 而每一次修改都需要检查大量的例句和例文. 如果处

1990 年 4 月收到, 1990 年 7 月 25 日定稿. 宋柔 系北京计算机学院副教授, 现从事人工智能、程序设计方面的研究工作. 王鑫 系北京计算机学院讲师, 目前主要从事人工智能方面的研究工作.

理速度不够快，则开发新产品或改版所需时间会过长甚至无法忍受．目前许多机译系统开发者正受此困扰．本文就这类系统的核心部分 — 句法分析提出一般性的顺序算法．这里的“句法分析”是广义的，包括了通常的语义分析．我们已用自己研制的 LISP 语言 DCLISP 实现了此算法并用来支持了一个实验性的英汉机器翻译系统．该系统的部分规则使用 DCLISP 的动态编译机制进行了编译 [1][2]．在 SUPER–AT 微机上，它平均 1 秒钟可对 20 个英文句子进行句法分析并生成其汉义．如果加上硬盘输入，查字典和硬盘屏幕同时输出的时间，即完成翻译的全过程，则平均不到 1 秒钟可翻译 1 个英文句子．对比国内已发表的一些系统 ([3][4] 其中，我们系统的规则及字典的规模都与 [3] 相当，比 [4] 小，[3] 是 150 秒翻译一个句子，[4] 是一小时翻译 1000 单词), 这一结果是很令人满意的．

自然语言句法分析的任务是依照一定规则把单词或短语的序列转换成树结构，这些规则构成了系统的知识库．我们的讨论基于以下的知识库模型：知识库由若干独立的规则库组成．规则库是规则的序列，规则是动作的序列．除最后一个动作外，各动作都有进行某种检验的语义．动作之间在逻辑上是合取的关系．同一规则库的各规则之间在逻辑上是析取的关系．有四类动作：

(1) 简单检验．该动作检验待处理序列中开始处的若干项是否满足某些设定的性质，如果成功，则把这些项从待处理序列中取出并结合成一个树结构．否则失败．

(2) 规则库调用．该动作使系统的控制进入某个规则库，直至那里某一条规则的所有动作都成功地完成，此时该调用成功返回；或者那里所有的规则中都有动作不能成功，此时该调用失败返回．

(3) 生成树结构．该动作把当前规则中它前面那些动作所生成的若干结构加以改造，合成一个新的树结构．

(4) 规则库重叠调用．该动作类似于规则调用，只是在调用前把当前规则中它前面各动作所生成的若干结构推回待处理序列．

每条规则的最后一个动作只能是生成树结构或规则库重叠调用．

当系统的控制进入到一个规则库后，若某规则的各动作都能执行成功，则称该规则为当时的成功规则．在同一状态下，一个规则库中可能有一条以上的成功规则．这时有两种处理机制．第一种机制是在一个规则库中只找出一条成功规则，它的算法简单，运行效率高，但可能漏掉正确分析句子的机会．因为在几条成功规则中，有一些可能导致后文分析的失败．第二种机制是穷尽一切成功规则来分析句子，不会遗漏各种分析的可能，但算法复杂，运行效率低．

本文讨论顺序控制算法．相应于上面第一种的称为“简单回溯”，相应于第二种的称为“彻底回溯”．下面给出这两种算法并进行简单的讨论．

§ 2. 简单回溯算法

本算法涉及以下数据：(1) 待处理的序列，它被当作栈来使用，记作 a 栈．(2) 一个栈，用来存放规则中各动作成功执行后产生的树结构，记作 b 栈．(3) 一个栈，用来存放规则库调用时需要保护的信息，记作 r 栈．(4) 规则库．

这三个栈和规则库都被实现成为表．每个规则库是一个表，每一条规则为其中的子表，子表中列出的是各个动作．每个动作又是一个表，第一项是动作名，其余各项是参数，参数本身也可以是表．

为了易于描述算法，所有这些表的结构形式在本文中都用符号表达式来表示．其中 nil 表示空表；有下横线的小写字母标识符是表示其本身字符串所成的原子；其余小写字母标识符是栈名和指针名，它们的意义在使用时有解释；大写字母标识符是形式变量，表示一个表．

各动作表示如下：

规则库调用：　(call. OPS)

简单检验：　(test. OPS)

生成树结构：　(makenode. OPS)

规则库重叠调用：　(againcall. OPS)

其中 OPS 表示各动作的参数．例如，(test n v) 可用 (text. OPS) 表示，此时 OPS 表示表 (n v).

本算法用到下面 7 个指针，它们的内容决定了系统的状态：

astart: 新进入一个规则库时的 a 栈.

atop: 当前的 a 栈.

bstart: 新进入一个规则库时的 b 栈.

btop: 当前的 b 栈.

rtop: 当前的 r 栈.

rule: 当前使用的规则中，将要执行的各动作所成的表.

base: 当前使用的规则库中，正在使用的规则后面的各规则的表.

系统初始化时首先进入一个规则库．设它的形式为 (RULE0. BASE0), 即它的第一条规则用 RULE0 表示，其余规则所成的表用 BASE0 表示.

则各指针在初始所指的内容为：

astart=atop= 初始时的 a 栈，即待处理序列所构成的表

bstart=btop=nil

rtop=nil

rule=RULE0

base=BASE0

下面就 rule 所指内容的不同情况，说明各系统指针如何变化，从而描述该算法.

(1) 设 rule=((call. OPS). RULE). 它表示 rule 所指的动作的表中的第一个动作具有 (call. OPS) 的形式，即要求调用一个规则库，其余动作的表用 RULE 表示．又设被调用规则库的形式为 (RULE1. BASE1), 则

astart		atop
bstart		btop
rtop	⟸	(astart bstart RULE base. rtop)
rule		RULE1
base		BASE1

这里使用“ ⟸ ”的形式表示对若干指针的同时赋值．　⟸ 左边的是被赋的指针名；⟸ 右边的是赋进的内容，其中出现的指针名表示它在变化前所指的内容．本情况

有五个指针被同时赋值，比如使 astart 指针指向 atop 指针原所指的内容等. atop 指针与 btop 指针内容不变.

(2) 设 rule=((test. OPS). RULE).

(2.1) 设 (test. OPS) 执行成功. 并设有一个表 A1, atop=(. A1. A) (其意义下面有解释), (test. OPS) 将表 A1 中的项结合成一个树结构 B1, 则

atop		A
btop	⟸	(B1. btop)
rule		RULE

如果 X 代表一个表，我们总是把符号表达式中的. X 看作把该表的括号去掉后各项排成的序列. 因此，(.A1. A) 表示表 A1 与 A 并置的结果.

(2.2) 设 (test. OPS) 执行失败. 此时进行回溯.

(2.2.1) 设 base=(RULE1. BASE1), 则

atop		astart
btop	⟸	bstart
rule		RULE1
base		BASE1

(2.2.2) 设 base=nil.

(2.2.2.1) 设 rtop=(ASTART1 BSTART1 RULE1 BASE1. RTOP1), 则

astart		ASTART1
bstart		BSTART1
rtop	⟸	RTOP1
rule		((test fail))
base		BASE1

其中 (test fail) 是一个必定失败的简单检验动作.

(2.2.2.2) 设 rtop=nil, 则算法失败停止.

(3) 设 rule=((makenode. OPS)), 又设 btop=(. B1. bstart), 则 (makenode.OPS) 把表 B1 中的项改造后合成一个新的树结构，记为 B2.

(3.1) 设 rtop=(ASTART1 BSTART1 RULE1 BASE1.RTOP1), 则

astart		ASTART1
bstart		BSTART1
btop	⟸	(B2. bstart)
rtop		RTOP1
rule		RULE1
base		BASE1

(3.2) 设 rtop=nil, 此时必有 bstart=nil, 若 atop=nil, 则算法成功停止，B2 为运行结果，否则算法失败停止.

(4) 设 rule=((againcall, OPS)), 又设 btop=(. B1. bstart), 设 ((againcall. OPS) 要求调用一个规则库，其形式为 (RULE1. BASE1), 则

astart, atop		(. B1' . atop)
btop	⟸	bstart
rule		RULE1
base		BASE1

其中 B1' 表示表 B1 倒置的结果.

§ 3. 彻底回溯算法

在句法分析系统的运行过程中，规则库的调用踪迹形成一个树结构．在采用简单回溯算法时，只保留该树中的一条路径，r 栈中四个一组的指针组就是该路径上表示规则库调用的结点．为了彻底回溯，需要保存整个树结构．我们仍使用 r 栈，树中结点在 r 栈中按树的前序列表顺序排列，并使每个子结点到父结点有指针链接．于是，代表系统状态的指针需要增加一个，指的内容是 r 栈中从当前规则库的调用结点开始直到栈底的那一部分．该指针记为 rmove, 其初值为 nil. 调用规则库时，rmove 的内容也被保存进 r 栈．下面沿用简单回溯算法的描述方法，说明彻底回溯的算法．

(1) 设 rule=((call. OPS). RULE), 类似于 2 (1), 只是把

rtop ⟸ (astart bstart RULE base. rtop) 改成

rtop, rmove ⟸ (astart bstart RULE base rmove. rtop).

(2) 设 rule=((test. OPS). RULE).

(2.1) (test. OPS) 执行成功，则完全同于 2 (2.1).

(2.2) 设 (test. OPS) 执行失败．此时进行回溯．

(2.2.1) 设 base=(RULE1. BASE1), 则类似于 2 (2.2.1), 只是在指针同时赋值中增加一项： rmove ⟸ rtop

(2.2.2) 设 base=nil, 则与 2 (2.2.2) 相同，只是状态变化前 rtop 内容若不是 nil, 则设 rtop=(ASTART1 BSTART1 RULE1 BASE1 RMOVE1. RTOP1), 并且在指针同时赋值中增加一项： rmove ⟸ RTOP1.

(3) 设 rule=((makenode. OPS)), 又设当时 B 栈中前 n 项组成表 B1 (这里 n 是当时所用规则中动作的个数减 1), 设 btop=(. B1.B), 又设 (makenode. OPS) 把 B1 中各项改造后合成一个新的树结构，记作 B2.

(3.1) 设 rmove=(ASTART1 BSTART1 RULE1 BASE1 RMOVE1. RTOP1), 则

btop		(B2. B)
rule	⟸	RULE1
rmove		RMOVE1

(3.2) 设 rmove=nil, 则同于 2 (3.2).

(4) 设 rule=((againcall. OPS)), 则类似于 2 (4), 只是在指针同时赋值中增加一项：rtop, rmove ⟸ (astart bstart nil base rmove. rtop).

(5) 设 rule=nil.

(5.1) 设 rmove=(ASTART1 BSTART1 RULE1 BASE1 RMOVE1. RTOP1), 则

rule	⟸	RULE1
rmove		RMOVE1

(5.2) 设 rmove=nil, 此时 b 栈必然只有一项，记为 B. 若 atop=nil, 则算法成功停止，B 为运行结果，否则算法失败停止．

§ 4. 彻底回溯中的剪裁

句法分析系统应当具有对规则库彻底回溯的能力．但为了效率，应当提供手段使规

则库设计者能对回溯范围进行剪裁．剪裁手段可以设计多种．模仿 PROLOG 中的 cut 剪裁，可以在规则中添加一种动作，形式不妨就是 cut, 如果 rule=(cut. RULE), 则

当 rmove=rtop 时，（即当时所用的规则中该 cut 之前没有规则库调用的动作），则

base ⟸ nil
rule RULE

当 rmove 不等于 rtop 时，则设

rtop=(.X ASTART1 BSTART1 RULE1 BASE1 RMOVE1. rmove)

其中 X 是一个表，这一式子表示了当时 rtop 与 rmove 的关系．这时，进行如下的指针赋值：

astart ASTART1
bstart BSTART1
rtop ⟸ rmove
rule RULE
base nil

如果采用彻底回溯加剪裁的机制，且每条规则最后都加上 cut 动作，则效果相当于简单回溯，而且运行速度不会相去太远．

§ 5. 算法的实现

由于自然语言句子及其处理规则表示成动态的结构化数据比较合适，故用表处理能力较强的语言 LISP 或 PROLOG 来实现有关算法效果较好．用 LISP 实现简单回溯算法是十分合适的．每个规则库都可自动转换成一个 LISP 函数定义，定义体是 OR 表达式，其中每一项表示一条规则，它们是 AND 表达式．采用这一方式，r 栈的功能可以自动地由 LISP 系统内部的栈来实现，大部分指针赋值也在 LISP 系统的内部完成．我们曾以该算法为核心建立了一个英汉机器翻译实验系统，取得了较为满意的效果（见引言）．

PROLOG 的搜索机制恰好支持彻底回溯算法，但 PROLOG 的合一机制语义复杂，效率低，用于自然语言的句法分析是很大的浪费．比较理想的方案是设计一种比 LISP 和 PROLOG 层面低一些但具有它们的基本能力的语言，用它来开发各种自然语言处理系统．这方面的工作我们正在进行．

致谢：这一工作的背景是我们同黑龙江大学合作的机器翻译系统，该系统的词典和规则库是黑龙江大学研制的，他们的工作给我们提供了帮助．马希文教授对本项工作给予了指导．在此一并表示感谢．

参考文献

[1] 马希文，宋柔，一个动态编译的 LISP 系统 —DCLISP, 中国计算机学会人工智能学组 LISP 语言专题学术讨论会，1983 年 8 月．
[2] 宋柔，DCLISP 系统，计算机研究与发展，第 25 卷，第七期 (1988, 7), 第 37–42 页．
[3] 王开铸，郭威，张丽玫，HH–87 机器翻译系统，计算机学报，第 12 卷，第六期 (1989, 6), 第 472–476 页．
[4] 姚兆炜，张曙野，《译星》机器翻译系统，计算机世界周报，总第 205 期增版 (1988, 9), 第 40 页．