Graph-structured Stack and Natural Language Parsing

Masaru Tomita
Center for Machine Translation
and
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

A general device for handling nondeterminism in stack operations is described. The device, called a *Graph-structured Stack*, can eliminate duplication of operations throughout the nondeterministic processes. This paper then applies the graph-structured stack to various natural language parsing methods, including ATN, LR parsing, categorial grammar and principle-based parsing. The relationship between the graph-structured stack and a chart in chart parsing is also discussed.

1. Introduction

A stack plays an important role in natural language parsing. It is the stack which gives a parser context-free (rather than regular) power by permitting recursions. Most parsing systems make explicit use of the stack. Augmented Transition Network (ATN) [10] employs a stack for keeping track of return addresses when it visits a sub-network. Shift-reduce parsing uses a stack as a primary device; sentences are parsed only by pushing an element onto the stack or by reducing the stack in accordance with grammatical rules. Implementation of principle-based parsing [9, 1, 4] and categorial grammar [2] also often requires a stack for storing partial parses already built. Those parsing systems usually introduce backtracking or pseudo parallelism to handle nondeterminism, taking exponential time in the worst case.

This paper describes a general device, a *graph-structured* stack. The graph-structured stack was originally introduced in Tomita's generalized LR parsing algorithm [7, 8]. This paper applies the graph-structured stack to various other parsing methods. Using the graph-structured stack, a system is guaranteed not to replicate the same work and can run in polynomial time. This is true for all of the parsing systems mentioned above; ATN, shift-reduce parsing, principle-based parsing, and perhaps any other parsing systems which employ a stack.

The next section describes the graph-structure stack itself. Sections 3, 4, 5 and 6 then describe the use of the graph-structured stack in shift-reduce LR parsing, ATN, Categorial Grammars, and principle-based parsing, respectively. Section 7 discusses the relationship between the graph-structured stack and chart [5], demonstrating that chart parsing may be viewed as a special case of shift-reduce parsing with a graph-structured stack.

2. The Graph-structured Stack

In this section, we describe three key notions of the graph-structured stack: splitting, combining and local ambiguity packing.

2.1. Splitting

When a stack must be reduced (or popped) in more than one way, the top of the stack is split. Suppose that the stack is in the following state. The left-most element, A, is the bottom of the stack, and the right-most element, E, is the top of the stack. In a graph-structured stack, there can be more than one top, whereas there can be only one bottom.

```
A --- B --- C --- D --- E
```

Suppose that the stack must be reduced in the following three different ways.

```
F <-- D E
G <-- D E
H <-- C D E
```

Then after the three reduce actions, the stack looks like:

```
              /-- F
             /
            /
A --- B --- C ---- G
       \
        \
         \-------- H
```

2.2. Combining

When an element needs to be shifted (pushed) onto two or more tops of the stack, it is done only once by combining the tops of the stack. For example, if "I" is to be shifted to F, G and H in the above example, then the stack will look like:

```
              /-- F --\
             /         \
            /           \
A --- B --- C ---- G ---- I
       \                /
        \              /
         \-------- H --/
```

2.3. Local Ambiguity Packing

If two or more branches of the stack turned out to be identical, then they represent local ambiguity; the identical state of stack has been obtained in two or more different ways. They are merged and treated as a single branch. Suppose we have two rules:

```
J <-- F I
J <-- G I
```

After applying these two rules to the example above, the stack will look like:

```
A --- B ---- C --- J
       \
        \
         \-- H --- I
```

The branch of the stack, "A-B-C-J", has been obtained in two ways, but they are merged and only one is shown in the stack.

3. Graph-structured Stack and Shift-reduce LR Parsing

In shift-reduce parsing, an input sentence is parsed from left to right. The parser has a stack, and there are two basic operations (actions) on the stack: shift and reduce. The shift action pushes the next word in the input sentence onto the top of the stack. The reduce action reduces top elements of the stack according to a context-free phrase structure rule in the grammar.

One of the most efficient shift-reduce parsing algorithms is *LR parsing*. The LR parsing algorithm pre-compiles a grammar into a parsing table; at run time, shift and reduce actions operating on the stack are deterministically guided by the parsing table. No backtracking or search is involved, and the algorithm runs in linear time. This standard LR parsing algorithm, however, can deal with only a small subset of context-free grammars called *LR grammars*, which are often sufficient for programming languages but clearly not for natural languages. If, for example, a grammar is ambiguous, then its LR table would have *multiple entries*, and hence deterministic parsing would no longer be possible.

Figures 3-1 and 3-2 show an example of a non-LR grammar and its LR table. Grammar symbols starting with "*" represent pre-terminals. Entries "sh *n*" in the action table (the left part of the table) indicate that the action is to "shift one word from input buffer onto the stack, and go to state *n*". Entries "re *n*" indicate that the action is to "reduce constituents on the stack using rule *n*". The entry "acc" stands for the action "accept", and blank spaces represent "error". The goto table (the right part of the table) decides to which state the parser should go after a reduce action. The LR parsing algorithm pushes state numbers (as well as constituents) onto the stack; the state number on the top of the stack indicates the current state. The exact definition and operation of the LR parser can be found in Aho and Ullman [3].

We can see that there are two multiple entries in the action table; on the rows of state 11 and 12 at the column labeled "*prep". Roughly speaking, this is the situation where the parser encounters a preposition of a PP right after a NP. If this PP does not modify the NP, then the parser can go ahead to reduce the NP to a higher nonterminal such as PP or VP, using rule 6 or 7, respectively (re6 and re7 in the multiple entries). If, on the other hand, the PP does modify the NP, then

```
-------------------------
 (1)    S --> NP VP
 (2)    S --> S PP
 (3)   NP --> *n
 (4)   NP --> *det *n
 (5)   NP --> NP PP
 (6)   PP --> *prep NP
 (7)   VP --> *v NP
-------------------------
```

Figure 3-1: An Example Ambiguous Grammar

State	*det	*n	*v	*prep	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	acc		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	re6,sh6	re6		9		
12				re7,sh6	re7		9		

Figure 3-2: LR Parsing Table with Multiple Entries (derived from the grammar in fig 3-1)

```
      /--------------S-------------------1--------------\
     /                                                    \
    /-------------S----------------1------\                \
   /                                       \                \
  /                /---------NP-------12-------\             \
 /                /                              \             \
0---NP--2---v---7---NP--12---p---6---NP--11---p---6---NP--11---p---6
 \               \          /     \                   /     /
  \-----S----1----\--------/       \---------NP---------/  /
                   \                                       /
                    \---------NP--------------6------------/
```

Figure 3-3: A Graph-structured Stack

the parser must wait (sh6) until the PP is completed so it can build a higher NP using rule 5.

With a graph-structured stack, these nondeterministic phenomena can be handled efficiently in polynomial time. Figure 3-3 shows the graph-structured stack right after shifting the word "with" in the sentence "I saw a man on the bed in the apartment with a telescope." Further description of the generalized LR parsing algorithm may be found in Tomita [7, 8].

4. Graph-structured Stack and ATN

An ATN parser employs a stack for saving local registers and a state number when it visits a subnetwork recursively. In general, an ATN is nondeterministic, and the graph-structured stack is viable as may be seen in the following example. Consider the simple ATN, shown in figure 4-1, for the sentence "I saw a man with a telescope."

After parsing "I saw", the parser is in state S3 and about to visit the NP subnetwork, pushing the current *environment* (the current state symbol and all registers) onto the stack. After parsing "a man", the stack is as shown in figure 4-2 (the top of the stack represents the current environment).

Now, we are faced with a nondeterministic choice: whether to return from the NP network (as state NP3 is final), or to continue to stay in the NP network, expecting PP post nominals. In the case of returning from NP, the top element (the current environment) is popped from the stack and the second element of the stack is reactivated as the current environment. The DO register is assigned with the result from the NP network, and the current state becomes S4.

At this moment, two processes (one in state NP3 and the other in state S4) are alive nondeterministically, and both of them are looking for a PP. When "with" is parsed, both processes visit the PP network, pushing the current environment onto the stack. Since both processes are to visit the same network PP, the current environment is pushed only once to both NP3 and S4, and the rest of the PP is parsed only once as shown in figure 4-3.

Eventually, both processes get to the final state S4, and two sets of registers are produced as its final results (figure 4-4).

5. Graph-structured Stack and categorial grammar

Parsers based on categorial grammar can be implemented as shift-reduce parsers with a stack. Unlike phrase-structure rule based parsers, information about how to reduce constituents is encoded in the complex category symbol of each constituent with *functor* and *argument* features. Basically, the parser parses a sentence strictly from left to right, shifting words one-by-one onto the stack. In doing so, two elements from the top of the stack are inspected to see whether they can be reduced. The two elements can be reduced in the following cases:

- `X/Y Y => X` (Forward Functional Application)
- `Y X\Y => X` (Backward Functional Application)
- `X/Y Y/Z => X/Z` (Forward Functional Composition)
- `Y\Z X/Y => X\Z` (Backward Functional Composition)

When it reduces a stack, it does so *non-destructively*; that is, the original stack is kept alive even after the reduce action. An example categorial grammar is presented in figure 5-1.

```
I          NP
saw        (S\NP)/NP
a          NP/N
man        N
with       (NP\NP)/NP, ((S\NP)\(S\NP))/NP
a          NP/N
telescope  N
```

Figure 5-1: An Example Categorial Grammar

The category, (S\NP), represents a verb phrase, as it becomes S if there is an NP on its left. The categories, (NP\NP) and (S\NP)\(S\NP), represent a prepositional phrase, as it becomes a noun phrase or a verb phrase if there is a noun phrase or a verb phrase on its left, respectively. Thus, a preposition such as "with" has two complex categories as in the

```
                                                            PP
                                                          /----\
      NP             v             NP                    /      |
(S1) ------> (S2) ------> (S3) ------> [S4]  <----/

                                              PP
                                            /----\
      det           n                      /      |
(NP1) -----> (NP2) -----> [NP3]  <----/
     \
      \ pron
       \-----> [NP4]                                 []: final states
                                                     (): non-final states
      p            NP
(PP1) -----> (PP2) -----> [PP3]

S1-NP-S2          A: Subj <-- *
S2-v-S3           C: (Subj-verb-agreement)
                  A: MV <-- *
S3-NP-S4          A: DO <-- *
S4-PP-S4          A: Mods <== *
NP1-det-NP2       A: Det <-- *
NP2-n-NP3         A: Head <-- *
NP3-PP-NP3        A: Qual <== *
NP1-pron-NP4      A: Head <-- *
PP1-p-PP2         A: Prep <-- *
PP2-NP-PP3        A: PrepObj <-- *
```

Figure 4-1: A Simple ATN for "I saw a man with a telescope"

```
bottom -------- S3 ------------ NP3
             [Subj: I          [Det: a
              MV:               Head:
               [root: see        [root: man
                tense: past]]     Num: Single]]
```

Figure 4-2: Graph-structured Stack in ATN Parsing "I saw a man"

```
bottom -------- S3 ------------ NP3 ------------ PP2 ---------- NP2
  \          [Subj: I          [Det: a          [Prep: with]    [Det: a]
   \          MV:               Head: man]        /
    \          [root: see                        /
     \          tense: past]]                   /
      \                                        /
       \----------- S4 -------------------/
                 [Subj: I
                  MV: [root: see
                       tense: past]
                  DO: [Det: a
                       Head: man]]
```

Figure 4-3: Graph-structured Stack in ATN Parsing "I saw a man with a"

```
bottom ---------- S4
        [Subj: I
         MV: [root: see
              tense: past]
         DO: [Det: a
              Head: man]
         Mods: [Prep: with
                PrepObj: [Det: a
                          Head: telescope]]]
        [Subj: I
         MV: [root: see
              tense: past]
         DO: [Det: a
              Head: man]
              Qual: [Prep: with
                     PrepObj: [Det: a
                               Head: telescope]]]
```

Figure 4-4: Graph-structured Stack in ATN Parsing "I saw a man with a telescope"

```
                /-------------- (S\NP)/N
               /
bottom ---- NP ---- (S\NP)/NP ---- NP/N
```

Figure 5-1: Graph-structured Stack in CG parsing
"I saw a"

```
                /-------------- (S\NP)/N -----\
               /                               \
bottom ---- NP ---- (S\NP)/NP ---- NP/N ------ N
         \     \                \
          \     \                \--------------- NP
           \     \
            \     \---------------------------- S\NP
             \
              \---------------------------------- S
```

Figure 5-2: Graph-structured Stack in CG parsing "I saw a man"

```
                /-------- (S\NP)/N -----\
               /                         \
bottom --- NP --- (S\NP)/NP --- NP/N --- N --\
         \    \            \                  \  /-- (NP\NP)/NP
          \    \            \---------- NP ----\ /
           \    \                               |---- ((S\NP)/(S\NP))/NP
            \    \---------------------- S\NP --/
             \                                 /
              \------------------------- S ---/
```

Figure 5-3: Graph-structured Stack in CG parsing "I saw a man with" example above. Nondeterminism in this formalism can be similarly handled with the graph-structured stack. After parsing "I saw a", there is only one way to reduce the stack; (S\NP)/NP and NP/N into (S\NP)/N with Forward Functional Composition. The graph-structured stack at this moment is shown in figure 5-1.

After parsing "man", a sequence of reductions takes place, as shown in figure 5-2. Note that S\NP is obtained in two ways (S\NP)/N N => S\NP and (S\NP)/NP NP => S\NP), but packed into one node with Local Ambiguity Packing described in section 2.3.

The preposition "with" has two complex categories; both of them are pushed onto the graph-structured stack, as in figure 5-3.

This example demonstrates that Categorial Grammars can be implemented as shift-reduce parsing with a graph-structured stack. It is interesting that this algorithm is almost equivalent to "lazy chart parsing" described in Pareschi and Steedman [6]. The relationship between the graph-structured stack and a chart in chart parsing is discussed in section 7.

6. Graph-structured Stack and Principle-based Parsing

Principle-based parsers, such as one based on the GB theory, also use a stack to temporarily store partial trees. These parsers may be seen as shift-reduce parsers, as follows. Basically, the parser parses a sentence strictly from left to right, shifting a word onto the stack one-by-one. In doing so, two elements from the top of the stack are always inspected to see whether there are any ways to combine them with one of the principles, such as augment attachment, specifier attachment and pre- and post-head adjunct attachment (remember, there are no outside phrase structure rules in principle-based parsing).

Sometimes these principles conflict and there is more than one way to combine constituents. In that case, the graph-structure stack is viable to handle nondeterminism without repetition of work. Although we do not present an example, the implementation of principle-based parsing with a graph-structured stack is very similar to the implementation of Categorial Grammars with a graph-structured stack. Only the difference is that, in categorial grammars, information about when and how to reduce two constituents on the top of the graph-structured stack is explicitely encoded in category symbols, while in principle-based parsing, it is defined implicitely as a set of principles.

7. Graph-structured Stack and Chart

Some parsing methods, such as chart parsing, do not explicitly use a stack. It is interesting to investigate the relationship between such parsing methods and the graph-structured stack, and this section discusses the correlation of the chart and the graph-structured stack. We show that chart parsing may be simulated as an exhaustive version of shift-reduce parsing with the graph-structured stack, as described informally below.

1. Push the next word onto the graph-structured stack.
2. Non-destructively reduce the graph-structured stack in all possible ways with all applicable grammar rules; repeat until no further reduce action is applicable.
3. Go to 1.

A snapshot of the graph-structured stack in the exhaustive shift-reduce parsers after parsing "I saw a man on the bed in the apartment with" is presented in figure 7-1 (slightly simplified, ignoring determiners, for example). A snapshot of a chart parser after parsing the same fragment of the sentence is also shown in figure 7-2 (again, slightly simplified). It is clear that the graph-structured stack in figure 7-1 and the chart in figure 7-2 are essentially the same; in fact they are topologically identical if we ignore the word boundary symbols, "*", in figure 7-2. It is also easy to observe that the exhaustive version of shift-reduce parsing is essentially a version of chart parsing which parses a sentence from left to right.

```
           /---------------S-------------------------------\
          /                                                \
         /-------------S--------------------\               \
        /                                    \               \
       /            /---------NP--------------\               \
      /            /                           \               \
bottom-----NP-----v------NP------p------NP------p------NP------p
      \            \            / \                            /
       \-----S------\----------/   \---------NP---------------/
                     \                                       /
                      \---------NP--------------------------/
```

Figure 7-1: A Graph-structured Stack in an Exhaustive Shift-Reduce Parser "I saw a man on the bed in the apartment with"

```
           /---------------S-------------------------------\
          /                                                 \
         /-------------S--------------------\                \
        /                                    \                \
       /                /--------NP-----------\                \
      /                /                       \                \
     *---NP---*---v---*---NP---*---p---*---NP---*---p---*---NP---*---p---*
      \                \      /         \                       /
       \-----S----------\----/           \----------NP---------/
                         \                                    /
                          \---------NP-----------------------/

        "I"     "saw"  "a man"   "on"  "the bed"  "in"  "the apt"  "with"
```

Figure 7-2: Chart in Chart Parsing "I saw a man on the bed in the apartment with"

8. Summary

The graph-structured stack was introduced in the Generalized LR parsing algorithm [7, 8] to handle nondeterminism in LR parsing. This paper extended the general idea to several other parsing methods: ATN, principle-based parsing and categorial grammar. We suggest considering the graph-structure stack for any problems which employ a stack nondeterministically. It would be interesting to see whether such problems are found outside the area of natural language parsing.

9. Bibliography

[1] Abney, S. and J. Cole.
A Government-Binding Parser.
In *Proceedings of the North Eastern Linguistic Society.* XVI, 1985.

[2] Ades, A. E. and Steedman, M. J.
On the Order of Words.
Linguistics and Philosophy 4(4):517-558, 1982.

[3] Aho, A. V. and Ullman, J. D.
Principles of Compiler Design.
Addison Wesley, 1977.

[4] Barton, G. E. Jr.
Toward a Principle-Based Parser.
A.I. Memo 788, MIT AI Lab, 1984.

[5] Kay, M.
The MIND System.
Natural Language Processing.
Algorithmics Press, New York, 1973, pages pp.155-188.

[6] Pareschi, R. and Steedman, M.
A Lazy Way to Chart-Parse with Categorial Grammars.
25th Annual Meeting of the Association for Computational Linguistics :81-88, 1987.

[7] Tomita, M.
Efficient Parsing for Natural Language.
Kluwer Academic Publishers, Boston, MA, 1985.

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

[9] Wehrli, E.
A Government-Binding Parser for French.
Working Paper 48, Institut pour les Etudes Semantiques et Cognitives, Universite de Geneve, 1984.

[10] Woods, W. A.
Transition Network Grammars for Natural Language Analysis.
CACM 13:pp.591-606, 1970.