2/11/87

Chart Parsing: Part 1

1. Problems with Backtrack Parsing

As we discussed above, the Prolog interpreter can be used directly as a parser for Definite Clause Grammars. The parsing strategy that it follows is a top-down, left to right, depth-first, backtracking one. There is a serious problem with backtrack parsing, and that is its inefficiency. A simple example will illustrate this point. Suppose a grammar contained the following two verb phrase expansions:

vp —> v[ditrans], np, pp.
vp —> v[ditrans], np, np. only difference is pp/np last clause.

Now suppose we are trying to parse a sentence such as (1):

(1) I sent the very pleasant double-glazing salesman that I met on holiday in Marbella last year a postcard.

The first rule will be chosen, and the rather long object noun phrase will be parsed as such. Then a prepositional phrase will be looked for, and of course there isn't one. Backtracking, the second verb phrase expansion will be chosen. All the work of parsing the object noun phrase will be repeated, and then finally 'a postcard' will be parsed as another noun phrase, and parsing will be successful. Because the results of an unsuccesful search are not stored, they must be recomputed. Although Prolog is very efficient, this failure to store well-formed constituents leads to a complexity of parsing that is exponential in the length of the input string, and for long sentences this is totally unacceptable.

2. Problems with Top-Down Depth-First Parsing

There is an even more serious problem with Prolog's search strategy. When a grammar contains a particular type of rule or combination of rules, parsing will fail to terminate altogether. For instance, a perfectly adequate grammar for possessives might contain the pair of rules:

np —> det, nom.
det —> np, ""s. search pattern infinitely circular.

Looking for a noun phrase, Prolog will first try to find a det. To find a det, the first thing to be found is a noun phrase. Thus control will arrive again at the same point in the grammar, without any of the input string having been consumed, and the computation will continue in this infinite loop until the machine resources allocated to the process are consumed and the computation is aborted. This is a general problem with any grammar that contains this sort of left recursion. Typically a grammar for English will contain several instances of left recursion, e.g.

np —> np, appos_mod.
vp —> vp, comp.
s —> s, conj, s.

but English is primarily a right branching language, and one can avoid the problem with a bit of contortion. Other languages, e.g. Japanese, which are primarily left-branching, have grammars that include left-recursive rules almost exclusively, and so the Prolog search strategy is totally unacceptable.

2/11/87

3. Bottom-up parsing

The left recursivity problem can be solved by the use of a bottom-up parsing algorithm. The simplest such algorithm is probably **shift-reduce**, which works as follows. Again, we make use of a stack. Symbols from the input string are read in and stacked. Each time we stack a symbol we look to see if there is a sequence of symbols on the top of the stack that matches the right hand side of a grammar rule. If there is, we may perform a reduction, that is, replace that string with the left hand symbol from the rule. If there is not, then we must shift the next symbol from the input string onto the stack, and repeat the process.

As with Prolog's top-down algorithm, this one is non-deterministic. There may be several rules whose right hand sides match the stack top. It may also be the case that although we can perform a reduction, we in fact need to perform a shift for successful recognition.

Also note in passing that there are types of grammar which will also cause this algorithm to cycle indefinitely, for instance, a grammar in which some symbol eventually derives itself. Another problem occurs when the grammar contains empty productions, since we can make an arbitrary number of reductions in which the empty string is reduced to itself. Neither of these problems, however, are as restrictive on the form of grammar as the left-recursive problem in top-down parsing.

The major drawback to this algorithm is one it shares with Prolog's backtracking, which is that time complexity of recognition is exponential in the length of the string. This is precisely because intermediate results computed on an unsuccessful search path are never stored, but must be recomputed even though the identical constituent may be discovered on the successful path.

The solution to this problem is to move some of the computational complexity from the control to the data structure. The space complexity of both backtracking parsers is linear in the length of the string, but by using a more complex data structure with a size bounded by n^2, we can reduce time complexity to polynomial. We will illustrate this with a bottom-up tabular parser, in which we ensure that we build no constituent more than once, at the expense of building some constituents that we do not need. Then we will go on to generalise the algorithm so that we build only what we need.

4. The Cocke-Kasami-Younger Algorithm

The following is a rational reconstruction of work by various people that is usually referred to as the CKY algorithm. The idea is to use a triangular matrix, from 0 to n, where n is the length of the string being parsed. This matrix is called a **chart**, or **well-formed substring table (wfst)**, and its indices correspond to the positions between the words in the sentence. The process of recognition is one of systematically filling in this matrix, so that there is an entry $C \in \text{chart}(i,j)$ if there is a constituent of type C spanning from position i to position j. The matrix is triangular since no constituent ends before it starts. Obviously then, recognition is successful if the final chart has the distinguished symbol $s \in \text{chart}(0,n)$.

To illustrate the workings of the algorithm, we first assume that the grammar is in a form in which all rules have at most two symbols on their right hand sides. Later, we will see how this constraint can be relaxed.

To compute an entry in the chart we use the equation:

$\alpha * \beta$: $\{\gamma_i \mid \alpha_i \in \alpha, b_i \in \beta, \alpha_i b \in \text{grammar}\}$ 2/11/87

Union

$$chart(i,j) = \bigcup_{i<k<j} chart(i,k) * chart(k,j)$$

where '$\alpha * \beta$' combines α and β according to the rules of the grammar. That is, it returns the set of phrases whose left daughter is from α and whose right daughter is from β. This algorithm can be performed in $O(n^3)$ time by choosing all combinations of i,j and k, each of which has n possible values. The complexity of the invariant is constant in the length of the input string. Since it depends only on the size of the grammar, it is known as the **grammar constant.**

One formulation of the algorithm is as follows:

```
for j <- 1 to n do
  chart(j-1,j) <- {A | A -> word_j}
  for i <- j-1 downto 0 do
    chart(i,j) <- chart(i,j) U *chart(i,j)
    for k <- i+1 to j-1 do
      chart(i,j) <- chart(i,j) U {chart(i,k) * chart(k,j)}

if s ∈ chart(0,n) then accept else reject
```

Obviously one can envisage alternative enumeration orders, but in practice this makes little difference to the efficiency of the algorithm.

An interesting point to notice in passing is the strong similarity between this algorithm and matrix multiplication, which originates in the similarity of the invariants, viz:

$$chart(i,j) = \bigcup_k chart(i,k) * chart(k,j)$$

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

Also note that nothing hinges on any particular way that we may care to represent the chart pictorially. Typically, computer scientists draw the matrix as such; computational linguists use a graph notation, in which the spaces between words are the vertices of the graph, and an edge labelled with C spans from vertex i to vertex j just in case C ∈ chart(i,j).

This algorithm is breadth-first, that is, it builds all constituents spanning a given portion of the input string before using any of them to build a larger constituent.

In fact, one of the drawbacks of the bottom-up strategy is that all constituents that are licensed by the grammar are built, regardless of whether they could be incorporated into a complete parse, that is, the algorithm is insufficently goal-driven. This problem will be rectified in a later section.

I will give an example of this algorithm running on the following grammar:

n → they v → are n → flying
v → flying a → flying n → planes

s → np vp vp → v np vp → v vp np → n
np → ap np np → vp ap → a vp → v

2/11/87

5. Dotted Rules

This algorithm assumes grammar rules have at most two symbols on their right-hand sides, whereas the sort of grammars that we want to write might contain rules of the form:

vp —> v[ditrans], np, pp.
vp —> v[ditrans], np, np.
vp —> v[raising], np, vp.

To generalise the above algorithm to work with grammars that have more than two symbols on their right hand sides, we introduce the notion of a dotted rule. The dotted rule is an entry in the chart, i.e. a type of edge, that embodies the idea of a partially found constituent. Instead of the multiplication step combining complete constituents according to the grammar, it combines one partial and one complete constituent according to information in the edges themselves.

So, the dotted rule indicates how much of a constituent has been found by taking a rule of the grammar and putting a dot between what has been found and what is yet to be found. Examples are:

vp → .v np pp
vp → v. np pp
vp → v np. pp
vp → v np pp.

The first of these represents a vp of which nothing has yet been found, and the last a vp which is complete. We use .vp and vp. as abbreviations for these respectively. The other two represent vps in various stages of completion. The multiplication rule now just takes the following form:

$A \rightarrow \alpha\ .X\ \beta\ *\ X. = A \rightarrow \alpha\ X.\ \beta$

where A is a non-terminal, X is a terminal or non-terminal, and α and β are any string of terminals and non-terminals, possibly empty. Possible instances of the rule are:

vp → .v np pp * v. = vp → v. np pp
vp → v. np pp * np. = vp → v np. pp
vp → v np. pp * pp. = vp → v np pp.

The problem we have now is how dotted rules get into the chart in the first place, and we will discuss alternative approaches to this below.