Chapter 11

Well-Formed Substring Tables (C)

11.1 Aims of this Chapter

- To explain the inefficiemcies of backtrack parsing and the need for the parser to have a memory of what it has already done. This motivates the use of both well-formed substring tables (this chapter) and charts (Chapter 12).
- To describe the CKY algorithm, the standard use of a well-formed substring table.

11.2 Problems with Backtrack Parsing

There is a serious problem with all of the interpreters given in the last chapter, which arises from their use of backtracking as a strategy for coping with the inherent non-determinism of parsing. The fact that well-formed constituents discovered on one branch of the search are lost if that branch eventually fails and may subsequently be recomputed leads to great inefficiency. A simple example will illustrate this point. Suppose a grammar contained the following two verb phrase expansions:

vp $\rightarrow$ v[ditrans] np pp
vp $\rightarrow$ v[ditrans] np np

Now suppose we are trying to recognise 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 recognition that is exponential in the length of the input string, and for long sentences this is totally unacceptable. In Chapter 8, we saw that the complexity of *parsing*, rather than *recognition* is at worst exponential for context-free grammars. Thus we know that if our English sentences have exponentially many analyses then there is nothing that can make parsing tractable. Here, however, we have an example where there is only one analysis but a search behaviour that is still pathological. The problem is that complexity of the Prolog strategy is at worst exponential *even for recognition*. In order to be efficient where there are not large numbers of analyses, we need to base our parsers on better recognisers.

The solution to this problem is to move some of the computational complexity from the control to the data structure. The space complexity of 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.

To avoid the repeated work involved in re-analysing the same constituents in different parts of the search space, it is necessary for a parser to store successfully parsed constituents, and use these stored results in preference to recomputing them. The abstract data structure used to store results is called a *well-formed substring table*, or *chart*. Since a tractable parsing algorithm (i.e. with polynomial complexity) must use a chart, there has been a considerable literature devoted to the question of parsing with a chart.

In presenting backtracking algorithms, we used (pure) Prolog, and let the latter's search algorithm take care of the housekeeping associated with backtracking. A chart parser can be implemented in Prolog, even pure Prolog, but since the programmer is forced to take over the housekeeping, there are not such advantages to using Prolog. In what follows, we will be less programming language-specific, and we will give algorithms in an ALGOL/PASCAL -like pseudo-code.

Again, to keep the presentation conceptually simple, we shall look first at an algorithm which requires grammars in Chomsky Normal Form. This will allow us to consider the major issues of correctness and complexity in their simplest form. Then we will see how the CNF restriction may be lifted in a completely general way.

11.3 The Cocke Kasami Younger (CKY) algorithm

The following is a rational reconstruction of work by various people that is usually referred to as the CKY algorithm. If a string to be parsed has n words, then the chart will be stored as a matrix size $(n+1)^2$, indexed from 0 to n. The indices represent string positions (between words and at either end), and the entries in the chart represent constituents of certain categories that have been found between those positions. The entries are therefore sets of nonterminal symbols. Here we will just consider the recognition problem. If we wished to extract parse trees from the chart, the entries would need to be more complicated.

The process of recognition is one of systematically filling in this matrix, so that the set $chart(i,j)$ is the set of all categories of constituents 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 the set $chart(0,n)$.

We must take some care about how we systematically fill in such a matrix. In particular, we must guarantee that a matrix entry is complete - that it contains all possible constituents over the string it covers - before we use it to compute a further entry. If this is not the case, a larger constituent looking for a certain phrase-type at a particular point will not know whether to look in the chart or try to parse the phrase from the beginning, and we will gain no benefit from the chart.

To ensure completeness, we will build all constituents ending at a certain point before we build any that end at a later point; and at a given point, we will build smaller constituents before we build larger ones. These two decisions make our version of the algorithm depth-first and bottom-up. It is a simple matter to change it to a breadth-first strategy.

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

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

The symbol $*$ (multiply) is an infix function that takes two sets of symbols $\{a_1, \ldots a_m\}$ and $\{b_1, \ldots b_p\}$ and returns the set $\{g_1, \ldots g_s\}$, containing all symbols g for which there is a rule $g \rightarrow \alpha\beta$, $\alpha \epsilon \{a_1, \ldots a_m\}$, $\beta \epsilon \{b_1, \ldots b_p\}$. Thus:

$$\{\text{np, n, verb, prep}\} * \{\text{np, s}\} = \{\text{vp, pp}\}$$

(assuming some appropriate grammar).

The CKY algorithm can be performed in cubic time by choosing all combinations of i, j and k, each of which has no more than n possible values. The complexity of the action performed in the innermost loop is constant in the length of the input string. (It is bounded by the square of the number of non-terminal symbols.) 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 $j$ from 1 going up to $n$ do
  set $chart(j-1,j)$ to $\{A \mid A \rightarrow word_j\}$
  for $i$ from $j-2$ going down to 0 do
    for $k$ from $i+1$ going up to $j-1$ do
      set $chart(i,j)$ to
        $chart(i,j) \cup (chart(i,k) * chart(k,j))$
if S $\epsilon$ $chart(0,n)$ then accept else reject
```

Intuitively,

- j is the point in the string up to which all complete phrases are being computed.

- i is a chosen starting point less than j - all complete phrases between i and j need to be computed.

- k is a chosen point between i and j - a phrase between i and j will be found if there are appropriate subphrases between i and k and between k and j.

Obviously one can envisage alternative enumeration orders, but as long as we don't try to use a chart entry before we have completed it, 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 actions in the inner loops, viz:

$$\begin{aligned} chart(i,j) &= \bigcup_{i<k<j} chart(i,k) * chart(k,j) \\ c_{ij} &= \sum_k a_{ik} b_{kj} \end{aligned}$$

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 $C \epsilon chart(i,j)$.

11.4 Ambiguity

The Appendix shows an example of the CKY algorithm running on the following grammar:

np → tigger	v → chases
n → dog	n → bone
n → garden	det → a
p → with	p → round
s → np vp	vp → v np
vp → vp pp	np → det n
np → np pp	pp → p np

A grammar like this assigns a Catalan$(N+1)$ number of parses to a sentence ending with N prepositional phrases, due to the ambiguity of attachment of a pp to a vp or np. (You should demonstrate this yourself by drawing the 5 parse trees for the sentence like 'Tigger chases a dog with a bone round a garden.'). Importantly, even though a constituent such as 'chases a dog with a bone' has two parses as a vp, with the CKY algorithm a higher constituent which uses that vp is not found or represented twice.

11.5 Making a Parser

So far, the algorithm described is for a recogniser that does not keep enough information for analyses to be reconstructed. To store this extra information but retain polynomial complexity, we need to represent the possible analyses in a factored form rather than by multiplying them out. We can do this by changing an element of a chart entry to be, instead of a simple category, a triple of the form:

$$< Category, Rule, Pos >$$

where each *Rule* is a rule of the grammar that justifies there being a phrase of the given category in this place and *Pos* is the position in the string where the two subphrases on the RHS meet. This, together with the information about the portion of the string covered by the chart entry is then enough information for a program to find the chart entries corresponding to the subphrases, recursively find their possible internal structures and so generate all possible analyses.

For such a modified system, the * operation needs to be redefined so as to ignore the extra structure information attached to its input categories, but to include extra information in its output:

$$\begin{aligned} & chart(i,k) * chart(k,j) = \\ & \quad \{< LHS, R, k > | \\ & \qquad \text{rule } R \text{ of the grammar is } LHS \rightarrow RHS_1\ RHS_2, \\ & \qquad \text{there is an element of the form } < RHS1, \_, \_ > \text{ in } chart(i,k) \text{ and} \\ & \qquad \text{there is an element of the form } < RHS2, \_, \_ > \text{ in } chart(k,j)\} \end{aligned}$$

Note that if there are several analyses of one of the subphrases, e.g. RHS_1, this is not reflected in a multiplication in the number of entries for the phrases containing it (in the construction of the set, multiple occurrences of $< LHS, R, k >$ will simply disappear). So, although the containing phrase may be found several times, it will only be represented once and so the multiple analyses of the subphrases cannot contribute to the work done in finding phrases containing that. In particular, when we consider non CNF grammars, this kind of chart representation will allow us to handle a grammar with a rule like:

S → S

(as well as conventional rules for S and other categories). Such a grammar provides an infinite number of analyses for any string that is an S, and so an enumeration of the possible parses would never terminate. Nevertheless, such an enhanced recogniser will always terminate and will be able to represent the infinite set of parses in a finite form (that can be subsequently enumerated for as long as one wishes).

11.6 Drawbacks of Bottom-up Parsing

One of the drawbacks of a 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 insufficiently goal-driven. For example, suppose our sentence was ‘the search for Spock was successful’. The phrase ‘search for spock’ would be found not only as part of a noun phrase, but also a present-tensed verb phrase. But a verb phrase can never directly follow a determiner such as ‘the’, so top-down prediction would allow us to prune the search tree. In Chapter 12, we will see how the use of a chart allows a parse to be driven by top-down prediction, while avoiding Prolog’s problems with non-termination.

11.7 Appendix: Trace of CKY algorithm

The following shows a trace of the CKY algorithm recognising the string:

```
  Tigger  chases  a   dog  with  a   bone  round  a   garden.
0       1       2   3    4     5   6     7      8   9        10
```

We show the values of the counters i, j and k given in the algorithm description, as well as the entries made to the chart as they are added. When for some value of k no additions are made to the chart, that value of k is not shown.

```
j = 1
   np from 0 to 1
j = 2
   v from 1 to 2
   i = 0
j = 3
   det from 2 to 3
   i = 1
   i = 0
j = 4
   n from 3 to 4
   i = 2
      k = 3
         np from 2 to 4
   i = 1
      k = 2
         vp from 1 to 4
   i = 0
      k = 1
         s from 0 to 4
j = 5
   p from 4 to 5
   i = 3
   i = 2
```

```
   i = 1
   i = 0
j = 6
   det from 5 to 6
   i = 4
   i = 3
   i = 2
   i = 1
   i = 0
j = 7
   n from 6 to 7
   i = 5
      k = 6
         np from 5 to 7
   i = 4
      k = 5
         pp from 4 to 7
   i = 3
   i = 2
      k = 4
         np from 2 to 7
   i = 1
      k = 2
         vp from 1 to 7
   i = 0
      k = 1
         s from 0 to 7
j = 8
   p from 7 to 8
   i = 6
   i = 5
   i = 4
   i = 3
   i = 2
   i = 1
   i = 0
j = 9
   det from 8 to 9
   i = 7
   i = 6
   i = 5
   i = 4
   i = 3
   i = 2
   i = 1
   i = 0
j = 10
   n from 9 to 10
   i = 8
      k = 9
         np from 8 to 10
   i = 7
      k = 8
         pp from 7 to 10
   i = 6
   i = 5
      k = 7
         np from 5 to 10
   i = 4
      k = 5
         pp from 4 to 10
   i = 3
   i = 2
      k = 4
         np from 2 to 10
   i = 1
      k = 2
         vp from 1 to 10
   i = 0
      k = 1
         s from 0 to 10
```

11.8 References

The formulation of the CKY algorithm is due to Martin, Church and Patil (1981), a very clear exposition of issues in chart parsing. See this paper for references to the original work.

- Martin, W.A., K.W.Church and R.S.Patil (1981) "Preliminary Analysis of a Breadth-First Parsing Algorithm" MIT Technical Report, MIT/LCS/TR-261

11.9 Quick Questions

Make sure that you can answer the following quick questions before you read on:

- Give an example where a backtrack recogniser will duplicate work unnecessarily.

- Is the CKY algorithm left-to-right or right-to-left?
- Is the CKY algorithm top-down or bottom-up?
- What is the computational complexity of recognition by the CKY algorithm?
- If in the CKY algorithm categories stored in the chart were replaced by parse trees, why would the complexity then be at worst exponential?

11.10 Exercises

Key:

`*` - These exercises should be completely mechanical once you have mastered the material. If you cannot do one of them after carefully re-reading the notes then you are having serious problems with the course material.

`**` - These exercises require some thought and you may need some hints. But you should fully understand a complete solution once you have seen or developed it.

`***` - These exercises require some significant extra thought or insight. You should spend some time thinking about these, but if you manage to come up with a solution you are doing exceptionally well.

11.1 Figures 11.1 and 11.1 represent empty "charts" for the CKY algorithm. Enter categories into these charts in the same way as the CKY algorithm would do for the following sentences:

sent a postcard
sent a man a boat
sent the man on the boat with the hat a postcard

You should use the grammar in `$csyn/lib/cfgram`.

11.11 What you should be able to do now

- Explain why backtrack parsing is less efficient than parsing with some kind of memory for found constituents.
- Describe the CKY algorithm and explain how it achieves its complexity characteristics.
- Exemplify the algorithm parsing a simple sentence.

FROM: / TO:	1	2	3	4
1				
2				
3				
4				

FROM: / TO:	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

Figure 11.1: Tables for CKY exercise

FROM: / TO:	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												

Figure 11.2: Tables for CKY exercise