Chart Parsing of Flowgraphs

Rudi Lutz
School of Cognitive and Computing Sciences
University of Sussex
Falmer, Brighton
England

Abstract

This paper will present a generalisation of chart parsing able to cope with the case where the object being parsed is a particular kind of diagram (a flowgraph) and the grammar is an appropriate type of graph grammar (a flowgraph grammar). A feature that often occurs in such diagrams is structure sharing. This paper also discusses the problem of diagram recognition in the case where structure sharing is allowed, noting that we want to permit structure sharing, but not enforce it.

1 Introduction and Motivation.

Many applications make use of diagrams to represent complex objects. Examples are electrical circuit diagrams and Programmer's Apprentice [Rich, 1981] style plan diagrams. In such applications it is often necessary to systematically recognise how some diagram has been pieced together from other diagrams. This is analogous to the parsing problem for strings, and this paper will present a generalisation of chart parsing [Thompson and Ritchie, 1984] able to cope with the case where the object being parsed is some kind of diagram (a flowgraph) and the grammar is an appropriate type of graph grammar (a flowgraph grammar). Often the various components of the diagrams can be regarded as producers of values which are fed as inputs to other components which in turn produce values to be passed on elsewhere. A feature that often occurs is structure sharing, when one component feeds one or more of its results to more than one other component (fan-out). In this situation the source component can be viewed as playing more than one role in the whole structure, and could have been duplicated so that separate copies of the component were responsible for each of these roles. This leads to no change in functionality, although there may be a loss in efficiency as measured by the number of components (electrical circuit case), or computational effort and code size (plan diagram case). This paper also discusses the problem of diagram recognition in the case where structure sharing is allowed, noting that we want to permit structure sharing, but not enforce it.

The symmetric case of structure-sharing arising through fan-in, rather than fan-out is not dealt with explicitly in this paper. However, the parsing algorithm is easily modified to cope with it, the neccessary modifications to the algorithm being similar to those needed for fan-out. Tha algorithm described in this paper has been implemented in POP-11.

2 Notation and Definitions.

Flowgraphs and flow grammars will be defined as special cases of plex languages and plex grammars first studied by Feder [1971]. A plex is a structure consisting of labelled nodes having an arbitrary number, n, of distinct attaching points, used to join nodes together. A node of this kind is called an n-attaching point entity (NAPE). Attaching points of NAPEs are not connected directly together, but are connected via intermediate points known as tie-points. A single tie-point may be responsible for connecting together two or more attaching points. If the direction of the connections is important then the plex is known as a directed plex. Many types of graph structure (e.g. webs [Pfaltz and Rosenfeld, 1969, Rosenfeld and Milgram, 1972], directed graphs, and indeed, strings) can be regarded as special cases of directed plexes. We will only consider the special case of directed plexes in which each NAPE's attaching points (from now on called ports) are subdivided into two mutually exclusive groups, known as input ports (restricted to only have incoming connections) and output ports (restricted to only have outgoing connections). We will further restrict ourselves to the special case in which each port of a NAPE is only connected to a single tie-point. This type of plex will be called a flowgraph and is a generalisation of Brotsky's [1984]use of the term. See Figure 1 (top) for an example of a simple flowgraph.

Just as a set of strings constitutes a language, so a set of plexes constitutes a plex language, and it is possible to define a plex grammar and the plex language generated by a plex grammar. Similar remarks apply to flowgraphs, webs, and graphs etc.

A production in a string grammar specifies how one string may be replaced by another, either in producing strings or in recognising them. In plex grammars the same is true but we encounter a difficulty (due to the 2-dimensional nature of plexes) not apparent in the string case. In the string case a production like

A ==> aXYb

applied to a string

....dAe..... (say)

results in the string

....daXYbe.....

and the question of how the replacement string is to be embedded in the host string in place of **A** never arises because there is a single obvious choice i.e. whatever is to the left of **A**

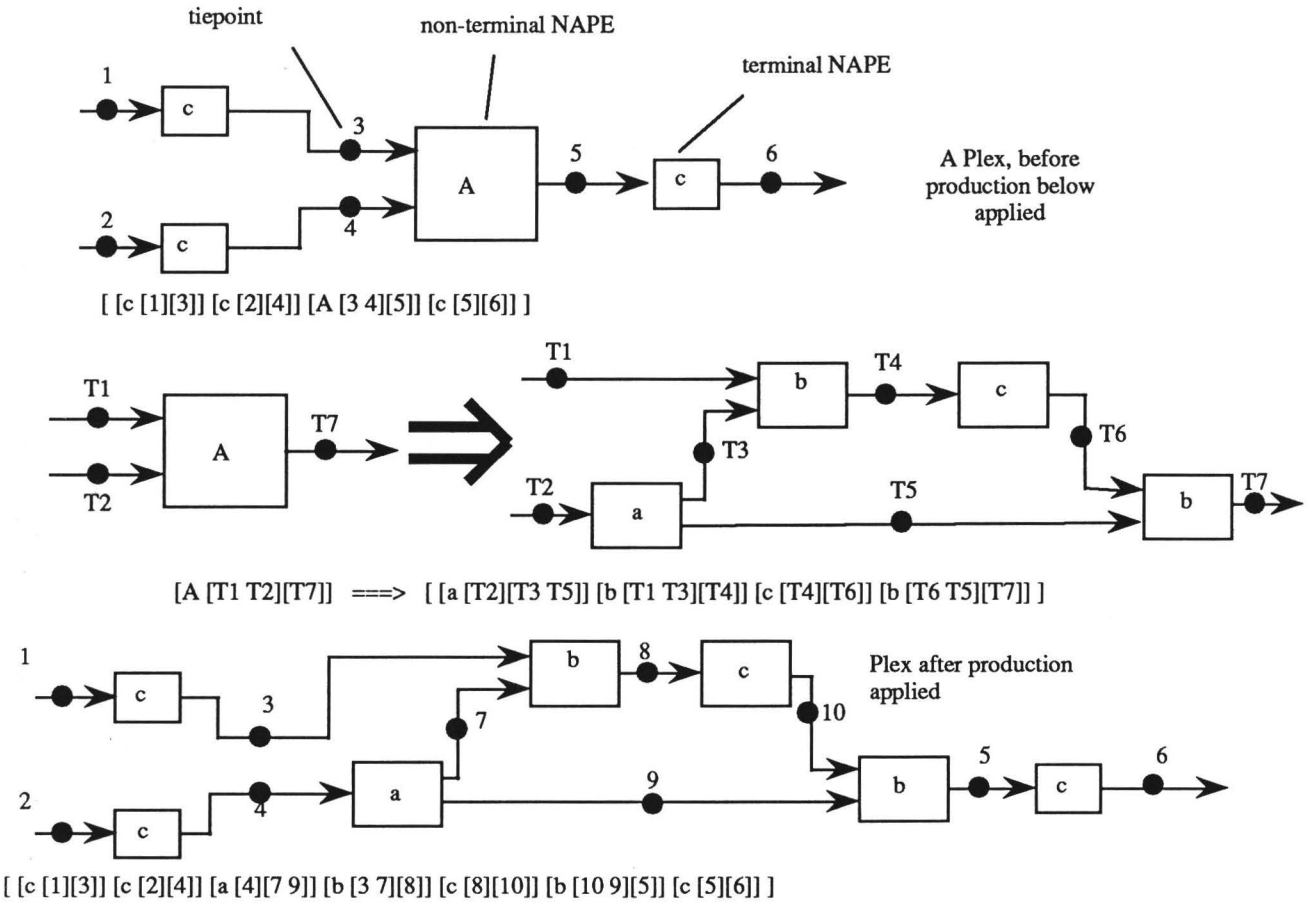

Figure 1

in the original string is to the left of the replacing string, and similarly on the right. In the graph case we no longer have this simple left-right ordering on the NAPEs and this question of embedding becomes much more complicated. Most of the discussion of this topic is in the web and graph grammar literature (e.g. [Pfaltz and Rosenfeld, 1969, Rosenfeld and Milgram, 1972]), but most of it applies (with some slight modifications) to the flowgraph case as well. The approach taken here is to specify with each production which tie-points on the left hand side correspond to which tie-points on the right and then connect everything connecting to one of these left hand tie-points (from the surrounding subgraph) to its corresponding right-hand tie-point.

We define a flowgraph grammar **G** to be a 4-tuple (**N**,**T**,**P**,**S**) where:

N is a finite non-empty set of NAPEs known as nonterminals.

T is a finite non-empty set of NAPEs known as terminals.

P is a finite set of productions.

S is a special member of **N** known as the initial (or start) NAPE

and the intersection of **N** and **T** must be empty.

If we arbitrarily order the input and output ports of a NAPE then each NAPE in a flowgraph can be represented in the form of a triple

(NAPE-label, input list, output list)

where NAPE-label is the label on the NAPE, and input list is a list in which the ith entry is the tie-point to which the ith input port is connected. Similarly the output list specifies to which tie-point each of the output ports is connected. Using this convention a complete flowgraph **G** can be represented as a set G^c (known as the component set) of such triples.

With the above conventions the productions in a flowgraph grammar have the general form

$$\mathbf{A}\ \mathbf{L_I}\ \mathbf{L_o} \text{ ==> } \mathbf{C}\ \mathbf{R_I}\ \mathbf{R_o}$$

where

A is known as the left-side structure, represented as a component set

C is known as the right-side structure, represented as a component set

$\mathbf{L_I}$ is the left-side input tie-point list

$\mathbf{R_I}$ is the right-side input tie-point list

$\mathbf{L_o}$ is the left-side output tie-point list, and

$\mathbf{R_o}$ is the right-side output tie-point list.

$\mathbf{L_I}$ and $\mathbf{R_I}$ must be of the same length, as must $\mathbf{L_o}$ and $\mathbf{R_o}$, and specify how an instance of the right-side structure is to be embedded into a structure **W** containing an instance of the left-side structure which is being rewritten according to the production. The rewriting and embedding is done as follows:

The instance of the left-side structure is removed from **W** and replaced by an instance of the right-side structure. Now, for each tie-point **X** in $\mathbf{L_I}$ any previous connections from NAPEs

in **W** to **X** are replaced by connections from the same attaching points of the same NAPEs to the corresponding tie-point in $\mathbf{R}_i$. The same is done for tie-points in $\mathbf{L}_o$ and $\mathbf{R}_o$. Note that one can eliminate the need for explicit storing of $\mathbf{R}_i$ and $\mathbf{R}_o$ by simply using the same variable names on the left and right hand sides of the production to denote corresponding tie-points.

Just as in the string case, by considering various restrictions on the form of **X** and **Y** in a production of the form:

X ==> Y

one can arrive at the notions of context-sensitive, context-free, and regular flowgraph languages [Ehrig, 1979]. In particular, restricting the productions to have a single NAPE in their left-side structure gives us the flowgraph equivalent of context-free string languages, and we will only concern ourselves with these from now on. In this case we no longer need to store $\mathbf{L}_i$ and $\mathbf{L}_o$ since the input and output lists of the single triple on the left of the production already specify this information. See Figure 1 for an example of the notation and of the rewriting process.

3 Chart Parsing of Context-free Flowgraphs.

In a chart parser, assertions about what has been found by the parsing algorithm are kept in a "database" known as the chart. Such assertions will be called covering patches (or simply patches), and are of two kinds - complete patches and partial patches. A complete patch is a statement that a complete grammatical entity (corresponding to some terminal or non-terminal symbol of the grammar) has been found. Partial patches are assertions that part of some grammatical entity has been found, and about what would need to be found in order to complete the grammatical entity concerned. One can think of a patch as being a closed loop drawn round some subgraph of the flowgraph, indicating that this subgraph corresponds to all or part of some grammatical entity as defined by the grammar. If we regard the right-side structures of rules as uninstantiated templates, then complete patches with non-terminal labels correspond to the occurrence of an instantiation of the right-side structure of some rule, thus forming an occurrence of the left-side structure of the rule. Partial patches correspond to partially instantiated instances of the right-side structure of some rule, and thus to partially recognised instances of the left-sidestructure of the rule. Each patch A contains the following information:

1) label(A) - the name of the grammatical entity corresponding to the patch, and is always one of the terminal or non-terminal symbols of the grammar.
2) inputs(A) - a set of input tie-points for the patch.
3) outputs(A) - set of output tie-points for the patch.
4) components(A) - a list of the other patches involved in making up this patch i.e. what other patches have been used to recognise this patch.
5) needed(A) - a description of what else needs to be found to complete the patch. In the case of a complete patch this will be empty, and for partial patches will be a flowgraph structure, represented as a list of triples.

For a partial patch, the input and output tie-points (i.e. those by which the patch connects to the surrounding flowgraph) are each subdivided into two categories - the set of active tie-points where the patch itself is still seeking other components to attach to these tie-points, and the set of inactive tie-points which are those which would be inputs or outputs of the patch were it complete. A NAPE needed by a partial patch will be called immediately needed if any of its tie-points are active. The components entry of a patch lists (instantiated versions of) those NAPEs in the right hand side of the rule which have been completely instantiated, and the needed entry lists uninstantiated (as yet) parts of the rule. Note that some of the tie-points in the needed entry may be instantiated. These are where the needed NAPEs connect to the ones already found. We will say that a partial patch A is extendable by a complete patch B (or that B can extend A) in the case where A immediately needs a patch of the same type as B and the instantiated tie-points in this needed patch do not conflict with any instantiations actually occurring in B.

The essence of the chart parsing strategy can then be stated as follows:

Every time a complete patch is added to the chart a search is made for any partial patches immediately needing a patch of the sort just added at the appropriate place. For each of these partial patches a new patch is made extending it by the complete one, and this new patch is then added to an agenda of patches to be processed at some appropriate time. Similarly, every time a partial patch is added to the chart a search is made for any complete patches which could be used to extend the partial patch just added, and any are found new patches are made which extend the partial one, and these are added to the agenda to be processed when appropriate. Note that patches are only ever added to the chart. They are never removed, thus avoiding the need to redo work that has been done before.

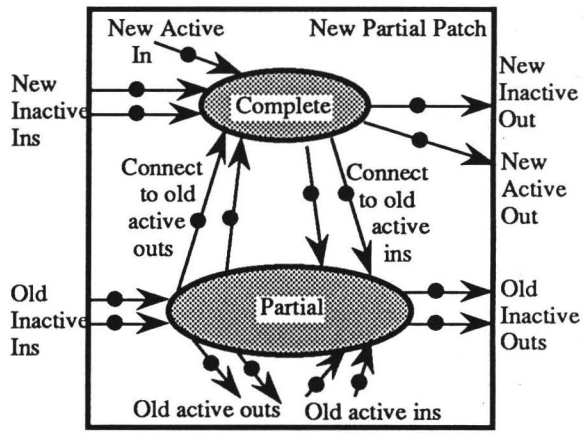

Figure 2

It should be clear from this that the basic operation of the algorithm is that of joining a complete patch to a partial patch to make a new enlarged patch. Fig. 2 shows a partial patch being joined to a complete patch to make a new patch (the enclosing box). The resulting patch has the same items in its components entry as the original partial patch plus the complete patch. Its needed entry is equal to that of the original partial patch minus the needed patch corresponding to the complete patch. Note that the matching of a needed patch to an actual complete patch may introduce further instantiations of tie-points in the needed entry of the new patch. On connecting the

two patches all the inactive tie-points of the partial patch remain inactive. Some of its active tie-points will correspond to tie-points of the complete patch (this is where the two patches actually join). Other active tie-points remain active in the new patch since it is still looking for other patches to attach to them. Of the complete patch's (input and output) tie-points some have already been mentioned i.e. those connecting directly to the partial patch. Others will become new inactive tie-points of the resulting patch since it will not be looking for anything to attach to them. However other (input and output) tie-points of the complete patch may now become active (viewed as belonging to the new patch) since it may now expect other patches to attach to them in order to complete itself. Provided all these distinctions are kept clear there is no great difficulty in implementing the joining operation.

The initialisation of the chart and the agenda now needs to be described. To begin with a complete patch is made for each of the terminal NAPEs in the original graph, and these are added to the agenda. If the algorithm is to be run top-down then an additional step is needed in which partial patches with empty components entries are made for every rule in the grammar whose left-side structure is labelled by the start symbol of the grammar. Each such rule leads to several such empty patches, one for each permutation of the input tie-points of the original graph. The inactive-inputs and active-outputs entries for each of these patches are the permuted inputs. The needed entry is just the right-side structure of the rule with any appropriate instantiations of the tie points occurring in it.

The complete algorithm is shown as Algorithm 1 below. When it terminates the parse is regarded as successful if the chart contains a complete patch for S, and the inputs and outputs entries are the same as the input and output tie-points of the graph being parsed.

The only remaining issue is how to organise the chart so that it can be searched efficiently. The chart is first of all divided into two parts, one for complete patches, and one for partial. The part for complete patches is organised as two arrays, one for indexing each patch by its inputs, and one for

```
initialise chart and agenda;
until the agenda is empty do
    pick a patch A from the agenda;
    unless A is already in the chart then
        add A to the chart;
        if A is complete then
            for each partial patch B in chart extendable by A do
                make a new patch extending B with A and put on agenda;
            endfor;
            if bottom-up then
                for each rule R in P such that rhs(R) has an input NAPE  labelled by label(A) do
                  for each such NAPE X in R do
                      make new empty patch B with label(B)=lhs(R) and
                          needed(B)=rhs(R) with instantiations dependent on match between X and A and
                          inputs(B)=inputs(A) and
                          active-outputs(B)=inputs(A);
                        add B to agenda;
                  endfor;
                endfor;
            endif;
        else
            for each complete patch B in chart which can extend A do
                make a new patch extending A with B and put on agenda;
            endfor;
            if top-down then
                for each object C immediately needed by A do
                      for each rule R in P with lhs(R)=label(C) do
                            make new empty patch B with label(B)=label(C) and
                                  needed(B)=rhs(R) with instantiations dependent on match between C and lhs(R) and
                                  inputs(B)=inputs(C) and
                                  active-outputs(B)=inputs(C);
                            add B to agenda;
                        endfor
                endfor
            endif
        endif
    endunless
enduntil;
```

Algorithm 1

indexing by its outputs. So each complete patch is entered several times into the chart, once for each of its inputs and outputs. For further efficiency each of the elements in these arrays is a hash table and the patches are actually entered into these hashed by their label. This enables efficient retrieval of all patches with a particular label at a particular place in the graph. In a similar fashion partial patches are entered into their part of the chart indexed by their input and output tie-points, and hashed by the labels of each of the patches they immediately need. Note that there may be several of these.

Finally, note that a similar trick can be used to store the grammar rules themselves in order to enable efficient retrieval of appropriate rules.

4 Structure-Sharing Flowgraphs.

As stated in the introduction we are also interested in the case where structure sharing is allowed. To make this more precise

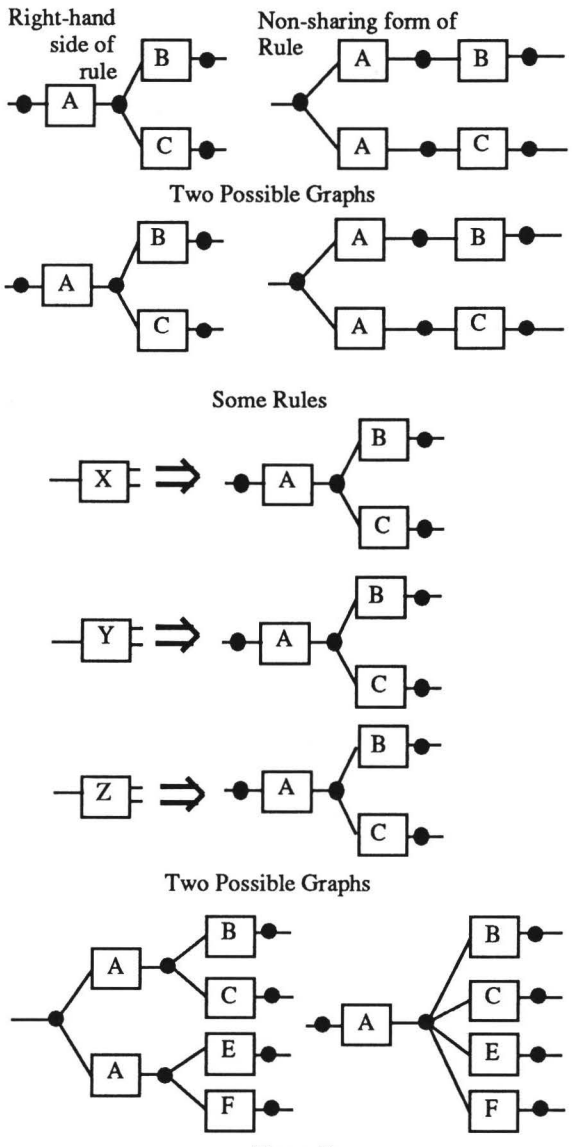

Figure 3

we define a relation **collapses** on the set of flowgraphs over some set of NAPEs by:

G2 **collapses** G1 iff G1 and G2 are flowgraphs, and $G1^c$ contains two triples of the form $T1=(A,(t_1,...,t_n),(x_1,...,x_m))$ and $T2=(A,(t_1,...,t_n),(y_1,...,y_m))$, and $G2^c$ can be obtained from $G1^c$ by removing these two triples and replacing them by a single triple of the form $T3=(A,(t_1,...,t_n),(z_1,...,z_m))$ and then replacing all occurrences of $x_1,...,x_m$ and $y_1,...,y_m$ by $z_1,...,z_m$ respectively throughout the remaining triples. In other words, G2 **collapses** G1 iff G1 contains two instances of some NAPE A (say) which have the same inputs, and G2 is identical to G1 except that the two instances of A have been replaced by a single instance of A (with the same inputs) and all NAPES which originally connected to the outputs of one or other of the two instances of A now connect to the single instance (in G2). This amounts to identifying the two instances of A and their corresponding tie-points.

The reflexive, transitive, symmetric closure of **collapses** is then an equivalence relation (share-equivalence) on the set of flowgraphs, and we then want any parsing algorithm which can recognise some graph G to also be able to recognise any flowgraphs share-equivalent to G. We also want the grammatical formalism used to be able to generate not only the flowgraphs derivable directly from the grammar, but also all share-equivalent flowgraphs. This can be done if we allow at any point in the generation of a flowgraph the replacement of the graph so far generated (G1) by any graph G2 for which either G1 **collapses** G2 or G2 **collapses** G1. A flowgraph grammar with the addition of this rewriting rule will be referred to as a structure sharing flowgraph grammar (a SSFG). Figure 3 illustrates several phenomena that can occur with SSFGs, and which motivated the above definition.

To see how the chart parsing algorithm can be modified to cope with SSFGs it should first be noted that for any flowgraph G there is a smallest flowgraph G_{min} which is share-equivalent to G. Secondly it should be noted that the right-side structure of any rule in a SSFG can be replaced by any flowgraph share-equivalent to it without altering the generative capacity of the grammar. We can therefore define a canonical form for a SSFG in which each rule of the form:

$$\mathbf{A ==> B}$$

has been replaced by the rule:

$$\mathbf{A ==> B_{min}}.$$

So the first change to the algorithm is actually to change the grammar to its canonical form, and to use this new form of the grammar for parsing. The second change is to the action of adding a complete patch to the chart. Previously the only check that was done was to see if the patch was already in the chart. Now the algorithm must additionally check that there is no other patch with the same label and the same inputs in the chart. If there is then the algorithm must collapse the new patch and the one that was there already into a single patch with a new set of output tie-points and identify the original outputs of the two patches with these new tie-points. Provided tie-points in the various triples making up the patches are represented as pointers to pointers to tie-points (rather than storing the tie-points directly in the triples) then simply changing the values of the second set of pointers will implement the identification universally throughout all patches in the chart. If the information that this has been done is needed by an application the algorithm can make a note of this fact either by annotating the tie-points

involved or by an assertion held separately.

5 Discussion

Although there is quite a lot of literature on the generative abilities of various types of graph grammar formalisms (see e.g. [Ehrig, 1979, Feder, 1971, Fu, 1974, Gonzalez and Thomason, 1978, Pfaltz and Rosenfeld, 1969, Rosenfeld and Milgram, 1972]), there is relatively little on parsing strategies, except for rather restricted classes of graph and web grammars [e.g.Della Vigna and Ghezzi, 1978]. In its top-down strictly left-to-right form chart parsing of context-free string languages corresponds to Earley's algorithm [Earley, 1970], which was generalised by Brotsky [1984] to parsing flowgraphs of the kind described here, except that his algorithm could not cope with fan-out at tie-points. However the approach taken here can also run bottom-up, and can cope with the case in which there is fan-out. Running bottom-up is particularly useful in applications in which we want to recognise as much as possible even though full recognition may be impossible (because of errors in the graph, or because the grammar is neccessarily incomplete). Zelinka [1986] has modified Brotsky's algorithm to cope with fan-out, but her algorithm only runs in a pseudo-bottom-up fashion by starting it running top-down looking for every possible non-terminal at every possible place in the graph. As discussed earlier the algorithm presented here is also easily modified to cope with structure sharing in a natural way, and indeed can also be easily generalised to run right-to-left as well as left-to-right. It can also be easily modified to cope with fan-in at tie-points.

A particular advantage of a chart parser is that it quite explicitly keeps a record of all partial patches it finds. This is useful in applications for which we may not just wish to verify that some graph can be generated from some grammar, but also to enable the system to make suggestions based on "near-miss" information about how to correct the graph. It is in such applications that it may be useful to modify the algorithm to run right-to-left as well, since this may enable one to find more "near-misses" (i.e. those missing their start NAPEs) than one would find if the parser only ran left-to-right.

The algorithm presented here runs in time polynomial in the size (measured by the number of tie-points T) of the graph being parsed provided that we do not allow the right-hand sides of rules to have "dangling" points (i.e. tie-points which are neither input or output tie-points of the rule, but which do not have both incoming and outgoing connections). If we allow these then there are graphs for which the algorithm will take time exponential in the size of the graph. Some intuition into why the algorithm runs in polynomial time can be gained if one considers the maximum number of possible patches that can be built. The algorithm only distinguishes patches which differ in at least one of their input tie-points, their output tie-points, or their label. Th e maximum number (K) of inputs andmaximum number (M) of outputs in a patch is determined by the grammar, as is the number of possible labels (L). So the number of possible patches is bounded above by the product of L and the number of possible ways of selecting at most K out of T tie-points, and the number of ways of choosing at most M out of T tie-points. This gives us $O(L.T^{K+M})$ patches altogether. Stemming from this fact a careful analysis then shows that the whole algorithm only takes timepolynomial in T. Full details will appear in a future paper. In this connection it should be noted that although the algorithm performs flowgraph recognition in polynomial time, it does not find all parses in polynomial time. This is because for some flowgraphs and some grammars there may well be an exponential number of parses (this is also true of Earley's algorithm operating on strings!). The algorithm will however find a parse if one exists. If an application requires all possible parses, then the algorithm can be modified to store any patch which is equal to one already in the chart in terms of its inputs, outputs, and label, but not equal in terms of its components, in an auxiliary data structure. At the end of the parsing there will then be enough information around in the chart to enable subsequent calculation of all the possible parses.

References

[Brotsky, 1984] Brotsky, D.C. An Algorithm for Parsing Flow Graphs. Technical Report AI-TR-704 MIT Artificial Intelligence Laboratory, 1984.

[Della Vigna and Ghezzi, 1978] Della Vigna, P. and Ghezzi, C. Context Free Graph Grammars. *Information and Control* 37, pp. 207-233, 1978.

[Earley, 1970] Earley J. An Efficient Context-Free Parsing Algorithm. *CACM* 13(2) pp.94-102, 1970.

[Ehrig, 1979] Ehrig H. Introduction to the Algebraic Theory of Graph Grammars (A Survey). *Graph Grammars and their Application to Computer Science and Biology.* (eds. Claus, V., Ehrig, H. and Rozenberg, G.) Lecture Notes in Computer Science, Springer-Verlag, 1979.

[Feder, 1971] Feder, J. Plex Languages. *Information Sciences*, Vol. 3, pp. 225-241, 1971.

[Fu, 1974] Fu, K.S. *Syntactic Methods in Pattern Recognition*, New York: Academic Press, 1974.

[Gonzalez and Thomason, 1978] Gonzalez, R.C. and Thomason, M.G. *Syntactic Pattern Recognition: An Introduction.* Addison-Wesley, 1978.

[Pfaltz and Rosenfeld, 1969] Pfaltz, J.L., and Rosenfeld, A. Web Grammars. Proc. IJCAI 1, pp. 609-619, 1969.

[Rich, 1981] Rich C. Inspection Methods in Programming MIT Artificial Intelligence Laboratory AI-TR-604, 1981.

[Rosenfeld and Milgram, 1972] Rosenfeld, A. and Milgram, D.L. Web Automata and Web Grammars. *Machine Intelligence* 7 pp.307-324 (eds. Meltzer, B. and Michie, D.) Edinburgh University Press, 1972.

[Thompson and Ritchie, 1984] Thompson H. and Ritchie, G. Implementing Natural Language Parsers. *Artificial Intelligence: Tools, Techniques, and Applications* pp.245-300 (eds. O'Shea, T. and Eisenstadt, M.) Harper and Row, 1984.

[Zelinka, 1986] Zelinka, L.M. Automated Program Recognition. MSc Thesis MIT Dept. Electrical Engineering and Computer Science, 1986.